

Tessent® Cell Library Manual

Software Version 2014.2

June 2014



This manual is part of a fully-indexed Tessent documentation set. To search across all Tessent manuals, click on the “binocular” icon or press Shift-Ctrl-F. Note that this index is not available if you are viewing this PDF in a web browser.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

About Tessent Libraries	13
Overview	14
Generating Simulation Models	15
Verifying Simulation Models	16
Adding Cell Attributes to Simulation Models	17
Generating a Tessent Cell Library	19
Certifying a Tessent Cell Library	20

Chapter 2

Creating Library Models	21
Positive-Edge (Nonscan) DFF Cell	22
Negative-Edge (Nonscan) DFF Cell	26
Active High Latch Cell	27
Active Low Latch Cell	28
MUX Scan Cell	29
Positive-Edge MUX Scan Cell	30
Negative-Edge MUX Scan Cell	32
Positive-Edge MUX Scan Cell With Active Low Scan Enable and Inverting Scan Out	34
Positive-Edge MUX Scan With Inverting and Non-inverting scan_out	35
Buffer and Input Pad	36
Symmetric Function (and, nand, or, nor, xor) Cells	37
AND Cells	38
MUX Cell	42
Clock Gating Cell	46
Clock Gating OR Cell with Both Enables	47
Clock Gating OR Cell with Both Enables and Asynch Enable	49
Clock Gating OR Cell with Func Enable Only	50
Clock Gating AND Cell With Both Enables	52
Clock Gating AND Cell With Func Enable Only	54
Clock Gating AND Cell with Func Enable and Asynch Enable	55
Clock Gating AND Cell with Inverting Enables	57
Positive-Edge Synchronizer Cell	58
Negative-Edge Synchronizer Cell	59

Chapter 3

Cell Library	61
Defining Cell Information	62
Cell Library Overview	62
Supported Library Syntax	63
Defining a Cell Library	65
Defining Cell Models	74

Defining Hardware	91
Defining Instance Attributes	96
Defining Pin Attributes	98
Internal Faults	120
Support of Arrays Within Library Models	124
Example Scan Definitions	125
Defining Macros	127
Reusing a Model Definition	127
Reading Multiple Libraries	128
Verilog Primitives	129
Supported Primitives	131
AND Gate	133
NAND Gate	134
OR Gate	134
NOR Gate	136
Inverter	137
Buffer	138
XOR Gate	138
XNOR Gate	140
Tri-State Buffer with Active Low Control	141
Inverted Tri-State Buffer with Active Low Control	142
Tri-State Buffer with Active High Control	143
Inverted Tri-State Buffer with Active High Control	144
Multiplexer	145
D Flip-Flop	146
D Latch	149
One Time Unit Delay Element (FlexTest Only)	151
Feedback Inverter	152
XDET	152
Wire Element	154
Pull-Up or Pull-Down Device	155
Power Signal	156
Ground Signal	156
Unknown Signal	157
High Impedance Signal	157
Undefined	158
Unidirectional NMOS Transistor	159
Unidirectional PMOS Transistor	159
Unidirectional Resistive NMOS Transistor	161
Unidirectional Resistive PMOS Transistor	162
Unidirectional Feedback NMOS Transistor	163
Unidirectional Feedback PMOS Transistor	164
Unidirectional CMOS Transistor	165
Unidirectional Resistive CMOS Transistor	166
Unidirectional Resistive Feedback CMOS Transistor	167
Pulse Generators with User Defined Timing	168
RAM and ROM	169

Chapter 4

Using LibComp to Create Tessent Simulation Models	189
Creating Simulation Models	190
Finding Unsupported Constructs in Partial Models	190
Finding Black Boxes with Vectored Outputs	191
Reconciling System Verilog reg and Verilog Keyword Compiling Issues	191
Accounting for Reserved Verilog Keywords	191
Accounting for Reg and Wire for Interconnect	191
Support for Verilog Parameter Overrides	192
LibComp Command Summary	193
LibComp Command Descriptions	195
Add Models	196
Delete Models	197
Dofile	198
Exit	199
Help	200
Report Models	201
Run	202
Set Asynchronous Control_Logic	203
Set Behavioral Processing	205
Set BB Outputs	206
Set Dofile Abort	207
Set Empty_module Outputs	208
Set Excessive Pull_delay	209
Set Floating Net_type	210
Set Hold Check	211
Set Instance Portlist	213
Set Model Source	215
Set MUX Nonconsensus_logic	216
Set System Mode	217
Set Undefined Instance	218
Set Verification	219
Set X_from_known Combinational_udp	220
System	221
Write Library	222
UDP Limitations and Examples	223
2-1 Mux Translation	223
General 3-Valued Limitations	223
LibComp Limitation - Complex Asynchronous Logic	227
LibComp Limitation - Verilog Construct Support	229
DFF Example	230
D Latch Example	230
I/O Pad Limitations and Examples	231
I/O Pad Code Example	233
Memory Limitations and Examples	234
Memories	234
Verilog Constructs	234
Arrays	235

\$readmemh and \$readmemb	235
ROM Example	235
RAM Examples	236
Chapter 5	
Verifying Tessent Simulation Models	259
Verification Overview	259
Specifying Which Tool Performs Verification	260
Verification Prerequisites	260
Running Verification from the Shell	260
Verifying a Single Simulation Model	261
Interpreting the Verification Results	261
verify.results File Example	261
Debugging Models	264
Re-simulating Verilog Only	265
Prerequisites for Simulating Verilog Only	265
Simulating Verilog Only	265
Fixing DRC Violations	266
Improving Test Coverage	266
Troubleshooting One Model at a Time	266
Assessing the Impact of Low Coverage	266
Locating Low-Coverage Models	267
Re-running the Tessent FastScan Portion of Verification	267
Modeling for Optimal Test Coverage	268
Handling Ignored or Blackboxed Models	268
Anticipating the Effects of Internal Gating on Clocks	268
Chapter 6	
Using ETLibCertify in the LV Flow	269
Certification Process Overview	269
Using ETLibCertify	269
<technology_name>.etlibcertify Configuration File	270
Certification Process	271
Example of Using ETLibCertify	274
Chapter 7	
Shell Command Dictionary	277
Shell Command Descriptions	277
etlibcertify	278
lcverify	279
libcomp	281
Appendix A	
Attributes and the Tessent Cell Library	285
Tessent Pin Function Attributes	286
Tessent Pin Special Drive Attributes	289
Cell Library Mappings	290
Tessent LV Flow Library Mappings	292

Table of Contents

Tessent LV Flow Pad Library Mappings.....	294
Tessent Scan ATPG Library Mappings.....	300
Appendix B	
Converting TetraMax Primitives to Verilog Primitives	303
Appendix C	
Reference for the .etlibcertify Input File	305
Cell	306
CellFilter.....	307
ConstantPinConnections.....	308
ETLibCertify.....	309
LogicHigh.....	310
LogicLow	311
TessentCellLib	312
SimModelDir	313
SimModelFile	314
VerilogOptionFile.....	315
Appendix D	
Reference for ETLibCertify Runtime Options	317
-log	318
-outDir.....	319
-mode	320
Appendix E	
Getting Help	321
Documentation.....	321
Mentor Graphics Support.....	322
Third-Party Information	
End-User License Agreement	

List of Figures

Figure 1-1. Tessent Cell Library Overview	14
Figure 1-2. Generating Simulation Models	15
Figure 1-3. Verifying Simulation Models	16
Figure 1-4. Adding Cell Attributes	17
Figure 1-5. Text Editor Method	18
Figure 1-6. Attribute Definition File Method	18
Figure 1-7. Library Merge Method	19
Figure 1-8. Certifying a Tessent Cell Library	20
Figure 2-1. Positive-Edge DFF Cell	22
Figure 2-2. Negative-Edge nonscan DFF Cell	26
Figure 2-3. Active-high Latch Cell	27
Figure 2-4. Active Low Latch Cell	28
Figure 2-5. Positive-Edge MUX DFF Scan Cell	30
Figure 2-6. Negative-Edge MUX DFF Scan Cell	32
Figure 2-7. Positive-Edge MUX Scan Cell With Active Low Scan Enable and Inverting scan_out	34
Figure 2-8. Positive-Edge MUX Scan Cell With Inverting and Non-inverting scan_out	35
Figure 2-9. 2-input AND Cell	38
Figure 2-10. 3-input AND Cell	38
Figure 2-11. 2-input Clock AND Cell	39
Figure 2-12. 2-input Clock AND Cell with Explicit Clock Pin	39
Figure 2-13. 2-input Clock OR Cell with Explicit Clock Pin	40
Figure 2-14. 2:1 MUX Cell	42
Figure 2-15. clock_gating_or with test_enable	47
Figure 2-16. Invalid clock_gating_or with test_enable	48
Figure 2-17. clock_gating_or with test_enable and asynch_enable	49
Figure 2-18. clock_gating_or	50
Figure 2-19. clock_gating_and with test_enable	52
Figure 2-20. Invalid clock_gating_and with test_enable	53
Figure 2-21. clock_gating_and	54
Figure 2-22. clock_gating_and with asynch_enable	55
Figure 2-23. clock_gating_and With test_enable	57
Figure 2-24. Positive-Edge Synchronizer Cell	58
Figure 2-25. Negative-Edge Synchronizer Cell	59
Figure 3-1. Bidirectional Buffer	76
Figure 3-2. Combinational Logic	115
Figure 3-3. Implying an Internal Node	116
Figure 3-4. Tri-State Buffer	116
Figure 3-5. Non-Inverting Buffer	116

List of Figures

Figure 3-6. Two-input NAND Gate.	117
Figure 3-7. Mux-DFF Scan Cell	117
Figure 3-8. The MUX	118
Figure 3-9. The DFF	118
Figure 3-10. Tri-State Gate (_buf primitive)	119
Figure 3-11. Tri-State Gate (_nmos primitive)	119
Figure 3-12. Tri-State Gate (_wire primitive)	120
Figure 3-13. Internal Faults	121
Figure 3-14. General Scan Definition Replacement Example.	126
Figure 3-15. Mux-Scan Definition Replacement Example	127
Figure 3-16. AND Gate	133
Figure 3-17. NAND Gate.	134
Figure 3-18. OR Gate.	135
Figure 3-19. NOR Gate	136
Figure 3-20. Inverter	137
Figure 3-21. Buffer	138
Figure 3-22. XOR Gate	139
Figure 3-23. XNOR Gate.	140
Figure 3-24. Tri-State Buffer with Active Low Control	141
Figure 3-25. Inverted Tri-State Buffer with Active Low Control	142
Figure 3-26. Tri-State Buffer with Active High Control	143
Figure 3-27. Inverted Tri-State Buffer with Active High Control.	144
Figure 3-28. Multiplexer	145
Figure 3-29. D Flip-Flop	148
Figure 3-30. D Latch	150
Figure 3-31. One Time Unit Delay Element	151
Figure 3-32. Feedback Inverter	152
Figure 3-33. Wire Element	154
Figure 3-34. Pull-Up or Pull-Down Device.	155
Figure 3-35. Undefined Functional Block	158
Figure 3-36. Unidirectional NMOS Transistor	159
Figure 3-37. Unidirectional PMOS Transistor.	160
Figure 3-38. Unidirectional Resistive PMOS Transistor.	161
Figure 3-39. Unidirectional Resistive NMOS Transistor	162
Figure 3-40. Unidirectional Feedback NMOS Transistor	163
Figure 3-41. Unidirectional Feedback PMOS Transistor	164
Figure 3-42. Unidirectional CMOS Transistor	165
Figure 3-43. Unidirectional Resistive CMOS Transistor	166
Figure 3-44. ROM	170
Figure 3-45. RAM	171
Figure 3-46. Example of a RAM without write_write_conflict	181
Figure 3-47. Example of a RAM with write_write_conflict	182
Figure 3-48. Flattened RAM Model with oen Set to 0	184
Figure 6-1. Complete Syntax of the .etlibcertify Configuration File.	270
Figure 6-2. Example scanCellSummary.final	273

Figure 6-3. Example padCellSummary.final	273
Figure 6-4. Example .etlibcertify File	274

List of Tables

Table 3-1. Cell_types	78
Table 3-2. Simulation_functions	82
Table 3-3. Hardware	91
Table 3-4. Instance Attributes	96
Table 3-5. Pin Attributes	98
Table 3-6. Supported Verilog Primitives	129
Table 3-7. AND Truth Table	133
Table 3-8. NAND Truth Table	134
Table 3-9. OR Truth Table	134
Table 3-10. NOR Truth Table	136
Table 3-11. Inverter Truth Table	137
Table 3-12. Buffer Truth Table	138
Table 3-13. XOR Truth Table	138
Table 3-14. XNOR Truth Table	140
Table 3-15. TSL Truth Table	141
Table 3-16. TSLI Truth Table	142
Table 3-17. TSH Truth Table	143
Table 3-18. TSHI Truth Table	144
Table 3-19. MUX Truth Table	145
Table 3-20. D Flip-Flop Primitives	146
Table 3-21. Alternative D Flip-Flop Primitive Table	146
Table 3-22. D Latch Primitive Table	149
Table 3-23. DELAY Truth Table	151
Table 3-24. INVF Truth Table	152
Table 3-25. XDET Truth Table	153
Table 3-26. WIRE Truth Table (for two inputs)	154
Table 3-27. PULL Truth Table	155
Table 3-28. UNDEFINED Truth Table	158
Table 3-29. NMOS Truth Table	159
Table 3-30. PMOS Truth Table	159
Table 3-31. RN MOS Truth Table	161
Table 3-32. RPMOS Truth Table	162
Table 3-33. NMOSF Truth Table	163
Table 3-34. PMOSF Truth Table	164
Table 3-35. CMOS Truth Table	165
Table 3-36. RCMOS Truth Table	166
Table 3-37. RCMOSF Truth Table	167
Table 4-1. Command Summary	193
Table 4-2. Output Dominance Logic	203
Table 5-1. Debugging Models	264

Table 7-1. Shell Command Summary	277
Table 7-2. LibComp Views	281
Table A-1. Tessent Pin Function Attributes	286
Table A-2. Tessent Pin Special Drive Attributes	289
Table A-3. cell.lib Attributes Mapped to Tessent Cell Library Attributes	290
Table A-4. scang.lib Attributes Mapped to Tessent Cell Library Attributes	292
Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes	294
Table A-6. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes	300

Chapter 1

About Tessent Libraries

A Tessent cell library is an integrated library that contains functionality information used for simulation by the Tessent tools, as well as DFT cell insertion attributes used for test logic insertion.

If you are currently using the ATPG library, you can continue using it with ATPG tools such as Tessent® FastScan and Tessent® TestKompress without making any changes. If you want to use a Tessent library in the LV Flow, you will need to add and certify appropriate attributes for the tools to be used.

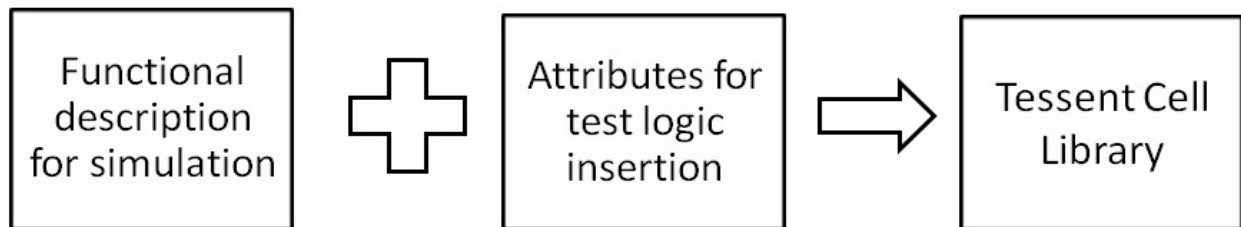
This manual describes the cell library with the pre-2012.3 library syntax for describing hardware, and the new test insertion and LV Flow attribute syntax. The manual also describes how to use ETLibCertify to verify that the library models and attributes are appropriate and adequate for use in the LV Flow.

Generating Simulation Models.....	15
Verifying Simulation Models	16
Adding Cell Attributes to Simulation Models.....	17
Generating a Tessent Cell Library.....	19
Certifying a Tessent Cell Library.....	20

Overview

The Tessent LV Flow and ATPG tools can utilize information about the function of a cell referenced in a netlist (usually originating from Verilog source files), and about the test attributes of netlist cells (usually originating from a Liberty file, a set of existing files used for the LV Flow, or a knowledgeable user). This information can all be stored in a single library source referred to as a *Tessent cell library*.

Figure 1-1. Tessent Cell Library Overview



Although you may want to create such an integrated library, it is unnecessary for existing customers to make any changes to support existing flows. Customers currently using the LV Flow using the LogicVision *cell.lib*, *pad.lib*, *scang.lib*, and scanModels/directories can continue using it. Customers currently using the ATPG flow with only ATPG models and no attributes can continue to do so. Tessent® Scan (scan insertion) users who utilize the old ATPG library scan_definition(...) syntax to define test attributes can continue to do that.

If you are an existing LV customer that wishes to create a complete library with models and attributes using the new syntax, you should refer to section “[Attributes and the Tessent Cell Library](#)” on page 285; this section will help you transition by mapping the old attribute spelling to new attribute spelling. However, you will probably find it easier to write out the merged Tessent cell library (as described in step 4 below) to output an old set of library files in the new syntax and see what the new spellings have become.

Be aware that it is not necessary to understand and populate all of the library, model, and pin attributes described in this document. The flow you are using dictates whether it is useful to populate an attribute. By referring to the attribute description in this manual, you can obtain more information about the attribute to help determine if it is needed for a specific application.

Creating a single Tessent cell library containing both test simulation models and test attributes usually requires you to perform all of the following steps:

1. Generate and verify simulation models (typically from Verilog source files). For information on performing this step, see the following section “[Generating Simulation Models](#).”

Note

If you have an existing LV Flow or ATPG library used for test simulations that you are converting to an attributed Tessent cell library, you won't need to do this first step, but can simply use that library.

2. Populate those simulation models with *attributes* used by test logic insertion tools. For information on performing this step, see the following section “[Adding Cell Attributes to Simulation Models](#).”
3. Write out the new merged Tessent cell library. For information on performing this step, see the following section “[Generating a Tessent Cell Library](#).”
4. Certify that the attributes and models can be correctly converted to LV Flow library files (LogicVision library files) for use in those flows. For information on performing this step, see the “[Using ETLibCertify in the LV Flow](#)”.

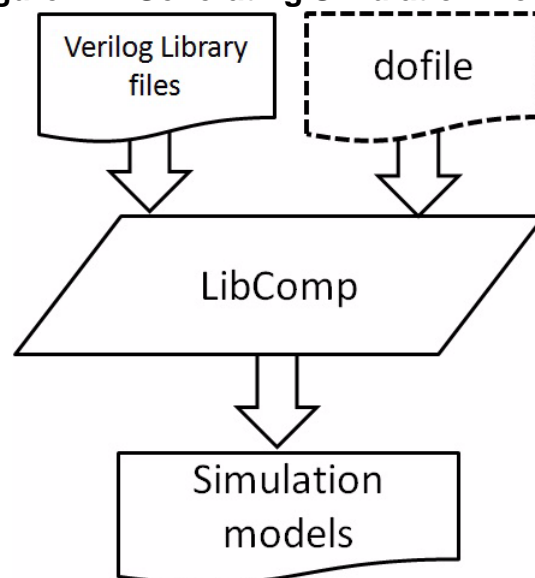


Tip: Appendix A contains three tables that map all of the LV Flow attributes to the new Tessent cell library attributes. For more information, see “[Attributes and the Tessent Cell Library](#)” on page 285.”

Generating Simulation Models

To create a library of simulation models from a Verilog netlist or library of Verilog modules, you invoke the LibComp tool on the Verilog source library typically using the unnamed default dofile provided by LibComp.

Figure 1-2. Generating Simulation Models



For example:

Tessent_Tree_Path/bin/libcomp verilog_source -dofile -log log_file

For more information on the invocation arguments, see the [libcomp](#) shell command in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*. For additional information on creating models using LibComp, see “[Using LibComp to Create Tessent Simulation Models](#).”

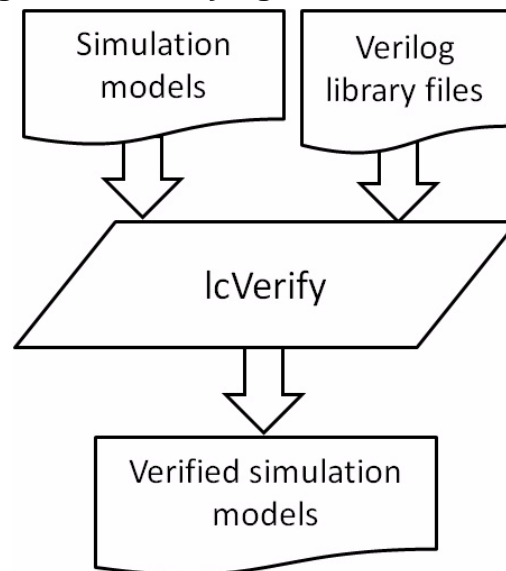
To get a quick reminder of the invocation arguments for LibComp before running it, enter the following on the command line:

```
libcomp -help
```

Verifying Simulation Models

By default, LibComp runs verification as it generates simulation models. If you manually create or edit a library model, you can run lcVerify from a UNIX/Linux command prompt to verify the model.

Figure 1-3. Verifying Simulation Models



For example:

Tessent_Tree_Path/bin/lcverify ATPG_library_name Verilog_library_names

For more information, see “[Verifying Tessent Simulation Models](#)”.

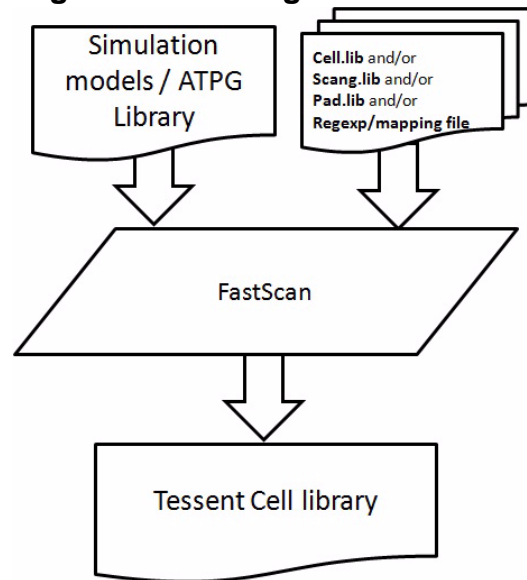
Adding Cell Attributes to Simulation Models

You can add cell attributes to your simulation models (typically generated by LibComp) to create a fully populated Tessent cell library.

The set of pre-defined recognized cell attributes are described in detail in section “[Defining Cell Information](#)” on page 62.

You can add cell attributes to your models using one of the following methods, depending on the types of library files you are converting:

Figure 1-4. Adding Cell Attributes



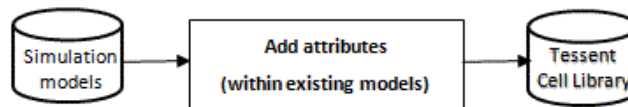
- Method 1 — Manually create the Tessent cell library by using a text editor to define all attributes or to add to the attributes after creating a library using Method 2 or Method 3. See “[Manual Creation of Cell Attributes](#).”
- Method 2 — Define cell attributes using simple and regular expressions in attribute definition syntax. See “[Cell Attribute Population With Simple and Regular Expressions](#).”
- Method 3 — Merge pre-existing attributes from LV Flow in *cell.lib*, *pad.lib*, *scang.lib* files with pre-existing ATPG library data or simulation models. See “[LogicVision Libraries](#).”

Manual Creation of Cell Attributes

You can use a text editor to create a Tessent cell library by adding attribute information directly to the simulation models

This is illustrated in [Figure 1-5](#) below.

Figure 1-5. Text Editor Method



This approach is shown in [Method 1](#) of section “[Primitive and Attribute Examples](#)” on page 111.”

Cell Attribute Population With Simple and Regular Expressions

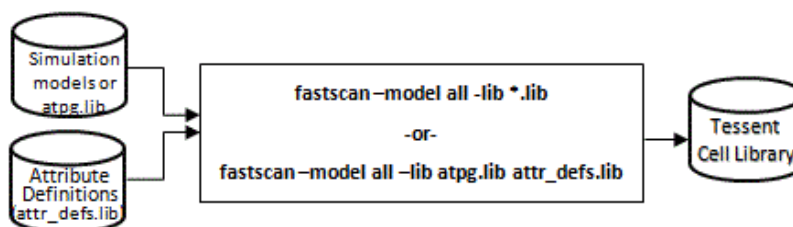
You can define cell attributes in an attribute definition file that is then read in during a library parser invocation and merged with the simulation models (or ATPG library models) to create a single source library.

This is shown in [Figure 1-6](#) below. In the attribute definition file, you can (1) simply list every model and cell attribute and/or, (2) use regular expressions to populate the library models with attributes. Both of these approaches are shown in [Method 2](#) of section “[Primitive and Attribute Examples](#)” on page 111.”

For complete information on using regular expressions to update attributes with pattern matching, see section “[Cell Selection](#).”

When the simulation models (or ATPG library models) and attribute definition file are loaded, the tool is ready to write out the cell library. For information on writing out the Tessent cell library, see ‘[Generating a Tessent Cell Library](#).’

Figure 1-6. Attribute Definition File Method



LogicVision Libraries

If you have existing LV libraries, you can merge this data with simulation models or an existing ATPG library to create a Tessent cell library.

Merging LV libraries is illustrated in [Figure 1-7](#). The LV libraries are already populated with attributes. This approach is shown in [Method 3](#) of section ““[Primitive and Attribute Examples](#)” on page 111.”

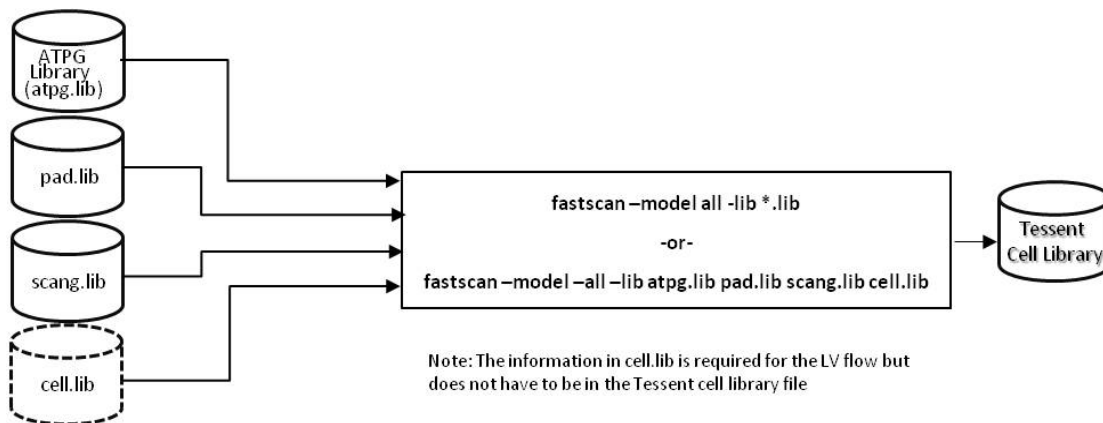
During tool invocation, you use the `-lib` argument to load all of the libraries and attribute files you want merged into one Tessent cell library. When these are loaded, the tool is ready to write out the cell library. For information on writing out the Tessent cell library, see ‘[Generating a Tessent Cell Library](#).’

Note



If you don't have an existing ATPG library, you can generate one from Verilog files as described in “[Generating Simulation Models](#).”

Figure 1-7. Library Merge Method



Generating a Tessent Cell Library

Use this procedure to generate a single library file from one or more library files.

Prerequisites

- One or more libraries exists.

Procedure

You can write out a single library file of all of the library files loaded at invocation, by performing the following steps:

1. Invoke the tool and specify the libraries you want to be merged into the Tessent cell library on the invocation line. For example:

Tessent_Tree_Path/bin/fastscan -model all -lib file_names

2. Once inside the tool, execute the [write_cell_library](#) command to write out the populated single file library. You can do this interactively after invocation or by invoking the tool with a dofile containing the command (and typically also an exit command) within the dofile. For example:

write_cell_library library_name.celllib

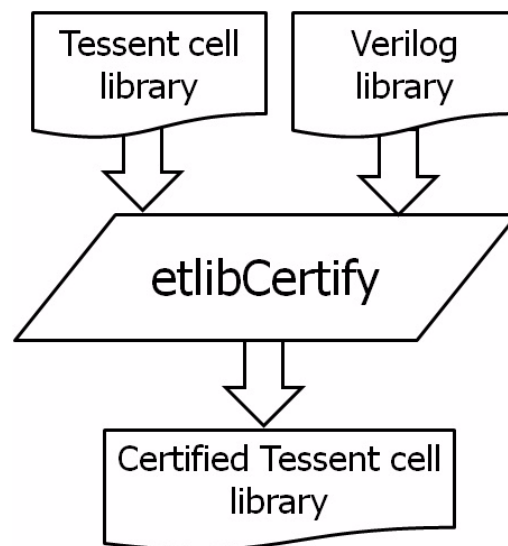
For more information on the invocation arguments, see the [tessent](#) shell command in the *Tessent Shell Reference Manual*.

For more information on the `write_cell_library` command, see [write_cell_library](#) in the *Tessent Shell Reference Manual*.

Certifying a Tessent Cell Library

After verification of the cell functionality, you can invoke ETLibCertify to verify the pin and cell attributes of the library cells and certify them for use in the Tessent Flow.

Figure 1-8. Certifying a Tessent Cell Library



Note

The Tessent Cell library file (or files) and Verilog files are provided to `etlibCertify` via a configuration file named *technology_name.etlibcertify*.

For example:

Tessent_Tree_Path/bin/etlibCertify technology_name

Chapter 2

Creating Library Models

The Tessent tools require simulation models for test simulations and attributes for those models and their pins to allow test tools to insert them as test logic. This chapter illustrates the process of creating a simulation model with the necessary attributes and required model and pin attributes.

The source of each step is illustrated for the “[Positive-Edge \(Nonscan\) DFF Cell](#)” on page 22. All subsequent cell types illustrate the final result but do not show the source for each step.

Positive-Edge (Nonscan) DFF Cell

Assume you want to model a positive-edge DFF. You need to first translate the DFF from Verilog and then add the Tessent attributes.

This process is described in the following procedure:

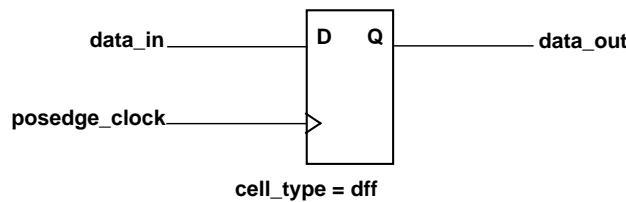
Model attributes:

cell_type = dff

Pin attributes:

data_in
posedge_clock
data_out

Figure 2-1. Positive-Edge DFF Cell



1. Begin with the Verilog source that describes a positive-edge nonscan DFF flip-flop.

```
primitive pos_dff_udp (q, clk, data, noti);
    input data;
    input clk;
    input noti;
    output q;
    reg q;
    table
    //clk data noti : q : q+ ;
        r  0  ?   : ? : 0 ; // Clock in 0
        r  1  ?   : ? : 1 ; // Clock in 1
        f  ?  ?   : ? : - ; // Hold when posedge clock falls
        ?  *  ?   : ? : - ; // Hold when data changes
        ?  ?  *   : ? : X ; // Go to X if timing violation signaled.
    endtable
endprimitive

`celldefine
    module pos_dff_cell (q, clk, din);
        output q;
        input  clk;
        input  din;
        reg notifier;
        wire din_delayed, clk_delayed;
        pos_dff_udp dff_inst (q, clk_delayed, din_delayed, notifier);

        // Note that specify blocks are ignored, except for connections
        // implied by $setuphold() statements when 8th and 9th inputs are
```

```
// specified, as below. Test simulations ignore timing.

specify
    $setuphold(posedge clk,posedge din,0, 0,notifier,, ,
        clk_delayed,din_delayed);
    $setuphold(posedge clk,negedge din,0, 0,notifier,, ,
        clk_delayed, din_delayed);
    (clk => q) = (`ifdef unit_delay 1 `else 1 `endif ,`ifdef
        unit_delay 1 `else 1 `endif );
    $width(negedge clk,1,0,notifier);
    $width(posedge clk,1,0,notifier);
endspecify
endmodule
`endcelldefine
```

2. Execute LibComp on the Verilog to produce a Tessent cell library model for each Verilog module and primitive.

```
*****
// File Type:      Tessent Cell Library
// Generated by:    Tessent Libcomp
// Tool Version:    2012.3-snapshot_2012.06.03_05.02
// Tool Build Date: Sun Jun 03 05:16:23 GMT 2012
//
*****

library_format_version = 9;
array_delimiter = "[";

model pos_dff_udp
    (q, clk, data, noti)
    (
        model_source = verilog_udp;
        input (clk) (posedge_clock; ) // Posedge Triggered Clock.
        input (data) (data_in; )
        input (noti) (no_fault = sa0 sa1; used = false;)
        // Notifier.
        output (q) (data_out; )
        (
            primitive = _dff mlc_dff ( , , clk, data, q, );
        )
    )

model pos_dff_cell
    (q, clk, din)
    (
        model_source = verilog_module;
        input (clk) ( )
        input (din) ( )
        output (q) ( )
        (
            instance = pos_dff_udp dff_inst
                ( q, clk_delayed, din_delayed, notifier );
            primitive = _buf mlc_buf_1 ( clk, clk_delayed );
            primitive = _buf mlc_buf_2 ( din, din_delayed );
        )
    )
```

3. Add the attributes to the models using one of the methods described in section “[Cell Library](#)” on page 61. After the attributes are added, the final Tessent Cell Library contains the following contents; **red** font indicates the attributes you added:

```
*****
// File Type:      Tessent Cell Library
// Generated by:    Tessent Libcomp
*****

library_format_version = 9;
array_delimiter = "[";

model pos_dff_udp
  (q, clk, data, noti)
  (
    model_source = verilog_udp;

    input (clk) (posedge_clock; ) // Posedge Triggered Clock.
    input (data) (data_in;)
    input (noti) (no_fault = sa0 sa1; used = false;)
    // Notifier.
    output (q) (data_out;)
    (
      primitive = _dff mlc_dff ( , , clk, data, q, );
    )
  )

// Note that this is exact copy of libcomp.atpglib (libcomp
// translation of Verilog)with additional information as noted in
// comments below. These can be added by several methods, to be
// discussed later.

model pos_dff_cell
  (q, clk, din)
  (
    cell_type = dff; // Added cell_type attribute.
    model_source = verilog_module;
    input (clk) (posedge_clock) // Added pin attribute (function)
    input (din) (data_in) // Added pin attribute (function)
    output (q) (data_out) // Added pin attribute (function)
    (
      instance = pos_dff_udp dff_inst
        ( q, clk_delayed, din_delayed, notifier);
      primitive = _buf mlc_buf_1 ( clk, clk_delayed );
      primitive = _buf mlc_buf_2 ( din, din_delayed );
    )
  )

//***** CELL SELECTION ADDED *****
// Ensure that Tessent test tools use pos_dff_cell if they insert a
// posedge dff. Sometimes, users have multiple technology cells
// (low power and non low power for example) and they can specify
// one posedge and one negedge dff for each technology. In this
// example, only one technology exists, with the name
// "technology_name".
// CAVEAT: If multiple posedge_dff are specified for the same
```



```
// technology_name, the last parsed will be kept as the one to use.  
  
dft_cell_selection(technology_name) {  
    posedge_dff = pos_dff_cell;  
}
```

Negative-Edge (Nonscan) DFF Cell

A negative-edge nonscan DFF requires the attributes described below.

Model attributes:

cell_type = dff

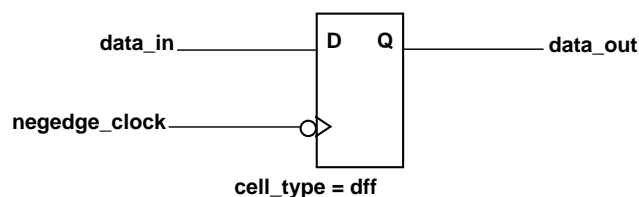
Pin attributes:

data_in

negedge_clock

data_out

Figure 2-2. Negative-Edge nonscan DFF Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model neg_dff_cell
  (q, clk, data)
  (
    cell_type = dff;
    model_source = verilog_module;
    input (clk) (negedge_clock)
    input (data) (data_in)
    output (q) (data_out)
    (
      instance = neg_dff_udp dff_inst
        ( q, clk_delayed, data_delayed, notifier );
      primitive = _buf mlc_buf_1 ( clk, clk_delayed );
      primitive = _buf mlc_buf_2 ( data, data_delayed );
    )
  )
  //***** CELL SELECTION ADDED *****
  // Ensure that Tessent test tools use neg_dff_cell if they insert a
  // negedge dff. Sometimes, users have multiple technology cells
  // (low power and non low power for example) and they can specify
  // one posedge and one negedge dff for each technology. In this
  // example, only one technology exists, with the name
  // "technology_name".
  // CAVEAT: If multiple negedge_dff are specified for the same
  // technology_name, the last parsed will be kept as the one to use.

  dft_cell_selection(technology_name) {
    negedge_dff = neg_dff_cell;
  }
```

Active High Latch Cell

An active high Latch cell requires the attributes described below.

Model attributes:

cell_type = latch

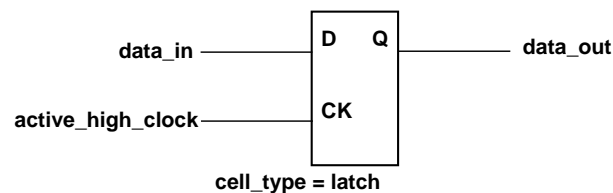
Pin attributes:

data_in

active_high_clock

data_out

Figure 2-3. Active-high Latch Cell



```
model pos_lat_cell
    (q, clk, data)
    (
        cell_type = latch;
        model_source = verilog_module;
        input (clk) (active_high_clock)
        input (data) (data_in)
        output (q) (data_out)
        (
            instance = pos_lat_udp dff_inst
                ( q, clk_delayed, data_delayed, notifier );
            primitive = _buf mlc_buf_1 ( clk, clk_delayed );
            primitive = _buf mlc_buf_2 ( data, data_delayed );
        )
    )
    //***** CELL SELECTION ADDED *****
    // To ensure that Tessent test tools use pos_lat_cell if they insert
    // activeHI latch. Sometimes, users have multiple technology cells
    // (low power and non low power for example) and they can specify
    // one activeHI and one activeLO latch for each technology. In this
    // example, only one technology exists, with the name
    // "technology_name".
    // CAVEAT: If multiple active_high_latch are specified for the same
    // technology_name, the last parsed will be kept as the one to use.

    dft_cell_selection(technology_name) {
        active_high_latch = pos_lat_cell;
    }
}
```

Active Low Latch Cell

An active low Latch cell requires the attributes described below.

Model attributes:

cell_type = latch

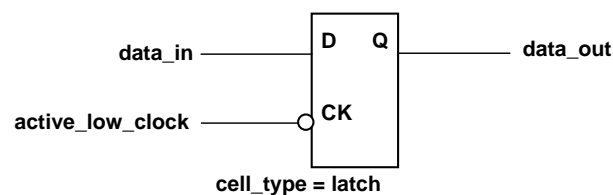
Pin attributes:

data_in

active_low_clock

data_out

Figure 2-4. Active Low Latch Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model neg_lat_cell
    (q, clk, data)
    (
        cell_type = latch;
        model_source = verilog_module;
        input (clk) (active_low_clock)
        input (data) (data_in)
        output (q) (data_out)
        (
            instance = neg_lat_udp dff_inst
                ( q, clk_delayed, data_delayed, notifier );
            primitive = _buf mlc_buf_1 ( clk, clk_delayed );
            primitive = _buf mlc_buf_2 ( data, data_delayed );
        )
    )
    //***** CELL SELECTION ADDED *****
    // To ensure that Tessent test tools use neg_lat_cell if they insert
    // activeLO latch. Sometimes, users have multiple technology cells
    // (low power and non low power for example) and they can specify
    // one activeHI and one activeLO latch for each technology. In this
    // example, only one technology exists, with the name
    // "technology_name".
    // CAVEAT: If multiple active_high_latch are specified for the same
    // technology_name, the last parsed will be kept as the one to use.

    dft_cell_selection(technology_name) {
        active_low_latch = neg_lat_cell;
    }
```

MUX Scan Cell

A cell_type scan_cell requires the attributes described below.

Model attributes:

nonscan_model = *nonscan_model_name*; // Required to insert scan for nonscan
cell_type = scan_cell; // Required if no nonscan_model statement, else implied.

General Pin attributes:

data_in
scan_in
scan_enable or scan_enable_inv
posedge_clock or negedge_clock
scan_out and/or scan_out_inv

Positive-Edge MUX Scan Cell

A positive-edge mux scan cell requires the attributes described below.

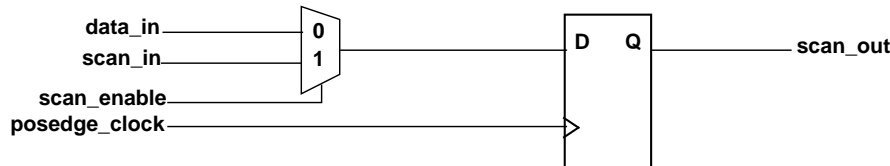
Model attributes:

`nonscan_model = nonscan_model_name;` // Required to insert scan for nonscan
`cell_type = scan_cell;` // Required if no nonscan_model statement, else implied.

Illustrated Pin attributes:

`data_in`
`scan_in`
`scan_enable`
`posedge_clock`
`scan_out`

Figure 2-5. Positive-Edge MUX DFF Scan Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model mux_scan
  (q, clk, din, sin, sen)
  (
    model_source = verilog_module;
    nonscan_model = pos_dff_cell; // Non-scan to be replaced
    cell_type = scan_cell;
    input (clk) (posedge_clock)
    input (din) (data_in)
    input (sin) (scan_in)
    input (sen) (scan_enable)
    output (q) (scan_out)
    (
      instance = data_mux mux_scan_mux ( .dout(mux_net),
        .sel(sen), .d1(sin), .d0(din) );
      instance = pos_dff_udp_wrap mux_scan_dff ( .q(q),
        .clk(clk), .din(mux_net) );
    )
  )
  //***** CELL SELECTION ADDED *****
  // To ensure that Tessent test tools use a particular cell if they
  // insert a posedge scan cell.
  // CAVEAT: If multiple posedge_scan_cell are specified for the same
  // technology_name, the last parsed will be kept as the one to use.

  dft_cell_selection(technology_name) {
    posedge_scan_cell = mux_scan;
  }
```


Negative-Edge MUX Scan Cell

A negative-edge mux scan cell requires the attributes described below.

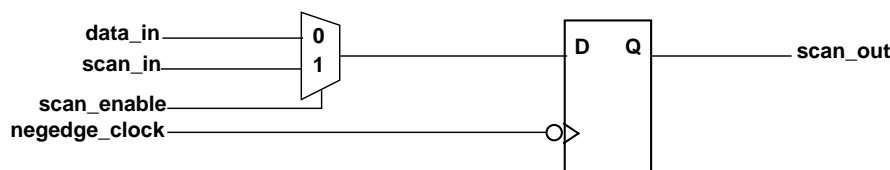
Model attributes:

nonscan_model = nonscan_model_name; // Required to insert scan for nonscan
cell_type = scan_cell; // Required if no nonscan_model statement, else implied.

Illustrated Pin attributes:

data_in
scan_in
scan_enable
negedge_clock
scan_out

Figure 2-6. Negative-Edge MUX DFF Scan Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model neg_mux_scan
  (q, clk, din, sin, sen)
  (
    model_source = verilog_module;
    nonscan_model = neg_dff_cell; // Non-scan to be replaced
    cell_type = scan_cell;
    input (clk) (negedge_clock)
    input (data) (data_in)
    input (sin) (scan_in)
    input (sen) (scan_enable)
    output (q) (scan_out)
  )
  (
    instance = data_mux mux_scan_mux
      ( .dout(mux_net), .sel(sen), .d1(sin), .d0(data) );
    instance = neg_dff_udp_wrap mux_scan_dff
      ( .q(q), .clk(clk), .din(mux_net) );
  )
)
//***** CELL SELECTION ADDED *****
// To ensure that Tessent test tools use neg_mux_scan if they insert
// a negedge scan cell.
// CAVEAT: If multiple negedge_scan_cell are specified for the same
// technology_name, the last parsed will be kept as the one to use.
```



```
dft_cell_selection(technology_name) {  
  negedge_scan_cell = neg_mux_scan;  
}
```

Positive-Edge MUX Scan Cell With Active Low Scan Enable and Inverting Scan Out

A positive-edge MUX scan cell with an active low scan_enable and inverting scan_out pin requires the attributes described below.

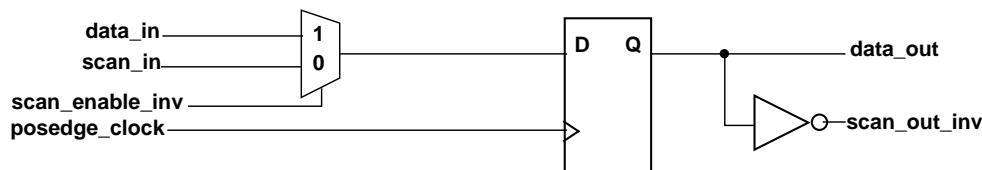
Model attributes:

nonscan_model = nonscan_model_name; // Required to insert scan for nonscan
cell_type = scan_cell; // Required if no nonscan_model statement, else implied.

Illustrated Pin attributes:

data_in
scan_in
scan_enable_inv
posedge_clock
data_out
scan_out_inv

Figure 2-7. Positive-Edge MUX Scan Cell With Active Low Scan Enable and Inverting scan_out



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model pos_mux_scan_inv_out
(q, qb, clk, din,
 sin, sen)
(
  model_source = verilog_module;
  cell_type = scan_cell;
  nonscan_model = pos_dff_cell; // Nonscan replaced
  input (clk) (posedge_clock)
  input (din) (data_in)
  input (sin) (scan_in)
  input (sen) (scan_enable_inv)
  output (q) (data_out)
  output (qb) (scan_out_inv)
  (
    instance = data_mux mux_scan_mux
      ( .dout(mux_net), .sel(sen_b), .d1(din), .d0(sin) );
    instance = pos_dff_udp_wrap mux_scan_dff
      ( .q(q), .clk(clk), .din(mux_net) );
    primitive = _inv inv_out ( q, qb );
  )
)
```

Positive-Edge MUX Scan With Inverting and Non-inverting scan_out

A positive-edge MUX scan cell with inverting and non-inverting scan_out requires the attributes described below.

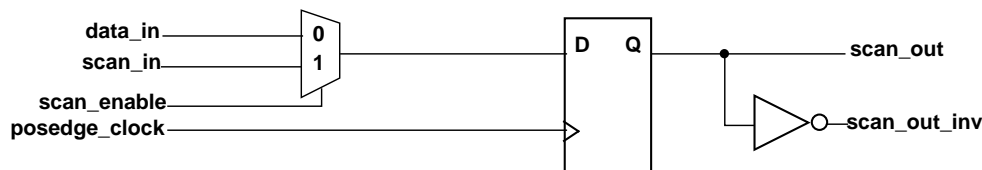
Model attributes:

nonscan_model = nonscan_model_name; // Required to insert scan for nonscan
cell_type = scan_cell; // Required if no nonscan_model statement, else implied.

Illustrated Pin attributes:

data_in
scan_in
scan_enable
posedge_clock
scan_out
scan_out_inv

Figure 2-8. Positive-Edge MUX Scan Cell With Inverting and Non-inverting scan_out



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model pos_mux_scan_both_out
(q, qb, clk, din,
 sin, sen)
(
  model_source = verilog_module;
  nonscan_model = pos_dff_cell; // Nonscan replaced
  cell_type = scan_cell;
  input (clk) (posedge_clock)
  input (din) (data_in)
  input (sin) (scan_in)
  input (sen) (scan_enable)
  output (q) (scan_out)
  output (qb) (scan_out_inv)
  (
    instance = data_mux mux_scan_mux
      ( .dout(mux_net), .sel(sen), .d1(sin), .d0(din) );
    instance = pos_dff_udp_wrap mux_scan_dff
      ( .q(q), .clk(clk), .din(mux_net) );
    primitive = _inv inv_out ( q, qb );
  )
)
```

Buffer and Input Pad

A Buffer and Input Pad cell requires the attributes described below.

```
model buf
(A, Y)
(
  cell_type = buffer:
  model_source = verilog_module;
  input (A) ( )
  output (Y) ( )
  (
    primitive = _buf buf_inst (A, Y);
  )
)
```

```
model ipad
(PAD, C)
(
  cell_type = pad: // required to distinguish from buffer cell
  model_source = verilog_module;
  input (PAD) ( )
  output (C) ( )
  (
    primitive = _buf ipad_inst (PAD, C);
  )
)
```

Symmetric Function (and, nand, or, nor, xor) Cells

Test tools use five types of symmetric combinational cells: {and, nand, or, nor, xor} as well as {clock_and and clock_or}. They can have any number of inputs but must have at least two.

Each of these cells require only a cell_type attribute:

Model attributes:

```
cell_type = and; cell_type = clock_and; cell_type = nand;  
cell_type = or; cell_type = clock_or; cell_type = nor; cell_type = xor;
```

Pin attributes:

None

Only a cell_type = and cell and cell_type = clock_and cell will be illustrated. Others are the same except the cell_type and the primitive implementing the function would change from “and” to “nand” for a cell_type = nand illustration, and similarly for the other symmetric cell types. For the clock_or cell, change the cell_type “clock_and” to “clock_or”.

AND Cells

An AND cell requires the attributes described below.

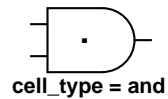
Illustrated Model attributes:

cell_type = and

Pin attributes:

None

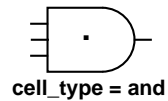
Figure 2-9. 2-input AND Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model data_and2
  (out, in1, in2)
  (
    model_source = verilog_module;
    cell_type = and;
    input (in1) ( )
    input (in2) ( )
    output (out) ( )
    (
      primitive = _and and_inst ( in1, in2, out );
    )
  )
)
```

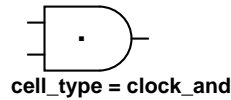
Figure 2-10. 3-input AND Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model data_and3
  (out, in1, in2, in3)
  (
    model_source = verilog_module;
    cell_type = and;
    input (in1) ( )
    input (in2) ( )
    input (in3) ( )
    output (out) ( )
    (
      primitive = _and and_inst ( in1, in2, in3, out );
    )
  )
)
```

Figure 2-11. 2-input Clock AND Cell



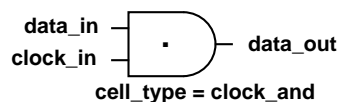
The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model clk_and
  (clk_out, clk_1, clk_2)
  (
    model_source = verilog_module;
    cell_type = clock_and;
    input (clk_1) ( )
    input (clk_2) ( )
    output (clk_out) ( )
    (
      primitive = _and and_inst ( clk_1, clk_2, clk_out );
    )
  )

  /******* CELL SELECTION ADDED *****/
  // Ensure that Tessent test tools use data_and3, data_and2, ...
  // if they insert an AND gate in a data path, but use clock_and
  // if they insert an AND gate in a clock path.
  // Sometimes, users have multiple technology cells (low power
  // and non low power for example) and they can specify a data
  // AND (and/or clock AND) for each technology. In this example,
  // only one technology exists, with the name "technology_name".
  //
  // Note that multiple cell_type=and can be specified, but only one
  // 2-input, one 3-input, etc. This applies to cell_type=clock_and
  // as well.

  dft_cell_selection(technology_name) {
    and = data_and3, data_and2; // For data path.
    clock_and = clk_and;       // For clock path.
  }
}
```

Figure 2-12. 2-input Clock AND Cell with Explicit Clock Pin



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model clock_and_cell
  (clk_out, clk_enable, clk_in)
  (
    model_source = verilog_module;
    cell_type = clock_and;
    input (clk_enable)(data_in)
    input (clk_in)(clock_in)
  )
}
```

```

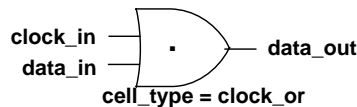
output (clk_out)(data_out)
(
    primitive = _and and_inst ( clk_enable, clk_in, clk_out);
)

//***** CELL SELECTION ADDED *****/
// Ensure that Tessent test tools use data_and3, data_and2, ...
// if they insert an AND gate in a data path, but use clock_and
// if they insert an AND gate in a clock path.
// Sometimes, users have multiple technology cells (low power
// and non low power for example) and they can specify a data
// AND (and/or clock AND) for each technology. In this example,
// only one technology exists, with the name "technology_name".
//
// Note that multiple cell_type=and can be specified, but only one
// 2-input, one 3-input, etc. This applies to cell_type=clock_and
// as well.

dft_cell_selection(technology_name) {
    and = data_and3, data_and2; // For data path.
    clock_and = clock_and_cell; // For clock path.
}

```

Figure 2-13. 2-input Clock OR Cell with Explicit Clock Pin



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```

model clock_or_cell
    (clk_out, clk_in, clk_enable)
    (
        model_source = verilog_module;
        cell_type = clock_or;
        input (clk_in)(clock_in)
        input (clk_enable)(data_in)
        output (clk_out)(data_out)
        (
            primitive = _or and_inst (clk_in, clk_enable, clk_out);
        )
    )

//***** CELL SELECTION ADDED *****/
// Ensure that Tessent test tools use data_or3, data_or2, ...
// if they insert an OR gate in a data path, but use clock_or
// if they insert an OR gate in a clock path.
// Sometimes, users have multiple technology cells (low power
// and non low power for example) and they can specify a data
// OR (and/or clock OR) for each technology. In this example,
// only one technology exists, with the name "technology_name".
//
// Note that multiple cell_type=or can be specified, but only one

```



```
// 2-input, one 3-input, etc. This applies to cell_type=clock_or  
// as well.  
  
dft_cell_selection(technology_name) {  
    or = data_or3, data_or2; // For data path.  
    clock_or = clock_or_cell; // For clock path.  
}
```

MUX Cell

A multiplexer requires the attributes described below.

Model attributes:

cell_type = mux

Pin attributes:

mux_in0

mux_in1

mux_in2

mux_in3

mux_in4

mux_in5

mux_in6

mux_in7

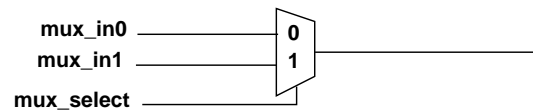
mux_select // for 2:1 mux only

mux_select0

mux_select1

mux_select2

Figure 2-14. 2:1 MUX Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model data_mux
    (mux_out, sel, d0, d1)
    (
        model_source = verilog_module;
        cell_type = mux;
        input (sel) (mux_select)
        input (d0) (mux_in0)
        input (d1) (mux_in1)
        output (mux_out) ( ) // mux_out attribute optional
        (
            instance = mux_udp mux_inst (mux_out, sel, d0, d1);
        )
    )

model clk_mux
    (mux_out, clk_sel, clk_0, clk_1)
    (
        model_source = verilog_module;
        cell_type = mux;
```

```

input (clk_sel) (mux_select)
input (clk_0) (mux_in0)
input (clk_1) (mux_in1)
output (mux_out) ( ) // mux_out attribute optional
(
    instance = mux_udp mux_inst (mux_out, clk_sel, clk_0, clk_1);
)
)

model data_mux3
    (Z, A, B, C, S1, S0)
    (
        model_source = verilog_module;
        cell_type = mux;

        input (A) ( mux_in0 )
        input (B) ( mux_in1 )
        input (C) ( mux_in2 )
        input (S1) ( mux_select1 )
        input (S0) ( mux_select0 )
        output (Z) ( )
        (
            instance = mux_udp mux_inst0 (mlc_sel_eq_0_net0, C, S1, Z);
            instance = mux_udp mux_inst1 (A, B, S0, mlc_sel_eq_0_net0);
        )
    )
)

model clock_mux3
    (Z, A, B, C, S1, S0)
    (
        model_source = verilog_module;
        cell_type = clock_mux;

        input (A) ( mux_in0 )
        input (B) ( mux_in1 )
        input (C) ( mux_in2 )
        input (S1) ( mux_select1 )
        input (S0) ( mux_select0 )
        output (Z) ( )
        (
            instance = mux_udp mux_inst0 (mlc_sel_eq_0_net0, C, S1, Z);
            instance = mux_udp mux_inst1 (A, B, S0, mlc_sel_eq_0_net0);
        )
    )
)

model data_mux4
    (Z, A, B, C, D, S1, S0)
    (
        model_source = verilog_module;
        cell_type = mux;

        input (A) ( mux_in0 )
        input (B) ( mux_in1 )
        input (C) ( mux_in2 )
        input (D) ( mux_in3 )
        input (S1) ( mux_select1 )
        input (S0) ( mux_select0 )
    )
)

```

```
output (Z) (    )
(
    instance = mux_udp mux_inst0 (C, D, S0, mlc_sel_eq_1_net1);
    instance = mux_udp mux_inst1 (mlc_sel_eq_0_net1,
mlc_sel_eq_1_net1, S1, Z);
    instance = mux_udp mux_inst2 (A, B, S0, mlc_sel_eq_0_net1);
)
)

model clock_mux4
(Z, A, B, C, D, S1, S0)
(
    model_source = verilog_module;
    cell_type = clock_mux;

    input (A) ( mux_in0 )
    input (B) ( mux_in1 )
    input (C) ( mux_in2 )
    input (D) ( mux_in3 )
    input (S1) ( mux_select1 )
    input (S0) ( mux_select0 )
    output (Z) (    )
    (
        instance = mux_udp mux_inst0 (C, D, S0, mlc_sel_eq_1_net1);
        instance = mux_udp mux_inst1 (mlc_sel_eq_0_net1,
mlc_sel_eq_1_net1, S1, Z);
        instance = mux_udp mux_inst2 (A, B, S0, mlc_sel_eq_0_net1);
    )
)

model data_mux5
(Z, A, B, C, D, E, S2, S1, S0)
(
    model_source = verilog_module;
    cell_type = mux;

    input (A) ( mux_in0 )
    input (B) ( mux_in1 )
    input (C) ( mux_in2 )
    input (D) ( mux_in3 )
    input (E) ( mux_in4 )
    input (S2) ( mux_select2 )
    input (S1) ( mux_select1 )
    input (S0) ( mux_select0 )
    output (Z) (    )
    (
        instance = mux_udp mux_inst0 (mlc_sel_eq_0_net0, E, S2, Z);
        instance = mux_udp mux_inst1 (C, D, S0, mlc_sel_eq_1_net2);
        instance = mux_udp mux_inst2 (mlc_sel_eq_0_net2,
mlc_sel_eq_1_net2, S1, mlc_sel_eq_0_net0);
        instance = mux_udp mux_inst3 (A, B, S0, mlc_sel_eq_0_net2);
    )
)

model data_mux8
(Z, A, B, C, D, E, F, G, S2, S1, S0)
```

```
(
  model_source = verilog_module;
  cell_type = mux;

  input (A) ( mux_in0 )
  input (B) ( mux_in1 )
  input (C) ( mux_in2 )
  input (D) ( mux_in3 )
  input (E) ( mux_in4 )
  input (F) ( mux_in5 )
  input (G) ( mux_in6 )
  input (H) ( mux_in7 )
  input (S2) ( mux_select2 )
  input (S1) ( mux_select1 )
  input (S0) ( mux_select0 )
  output (Z) ( )
  (
    instance = mux_udp mux_inst5 (mlc_sel_eq_0_net1,
mlc_sel_eq_1_net1, S1, mlc_sel_eq_1_net0);
    instance = mux_udp mux_inst6 (G, H, S0, mlc_sel_eq_1_net1);
    instance = mux_udp mux_inst4 (E, F, S0, mlc_sel_eq_0_net1);
    instance = mux_udp mux_inst0 (mlc_sel_eq_0_net0,
mlc_sel_eq_1_net0, S2, Z);
    instance = mux_udp mux_inst1 (C, D, S0, mlc_sel_eq_1_net2);
    instance = mux_udp mux_inst2 (mlc_sel_eq_0_net2,
mlc_sel_eq_1_net2, S1, mlc_sel_eq_0_net0);
    instance = mux_udp mux_inst3 (A, B, S0, mlc_sel_eq_0_net2);
  )
)

//***** CELL SELECTION ADDED *****/
// Ensure that Tessent test tools use data_mux if they insert a Mux
// in a data path, but use clk_mux if they insert a Mux in a
// clock path. Sometimes, users have multiple technology cells
// (low power and non low power for example) and they can specify a
// data Mux (and/or clock Mux) for each technology. In this example,
// only one technology exists, with the name "technology_name".
//

dft_cell_selection(technology_name) {
  mux = data_mux, data_mux3, data_mux4, data_mux5, data_mux8;
  // For data path
  clock_mux = clk_mux, clock_mux3, clock_mux4; // For clock path.
}
```

Clock Gating Cell

A clock_gating_or and clock_gating_and require the attributes described below.

Model attributes:

cell_type = clock_gating_or or cell_type = clock_gating_and

Pin attributes:

func_enable or func_enable_inv

test_enable or test_enable_inv

asynch_enable or asynch_enable_inv

asynch_disable or asynch_disable_inv

clock_in

clock_out

Clock Gating OR Cell with Both Enables

A clock gating OR requires the attributes described below.

Model attributes:

cell_type = clock_gating_or

Pin attributes:

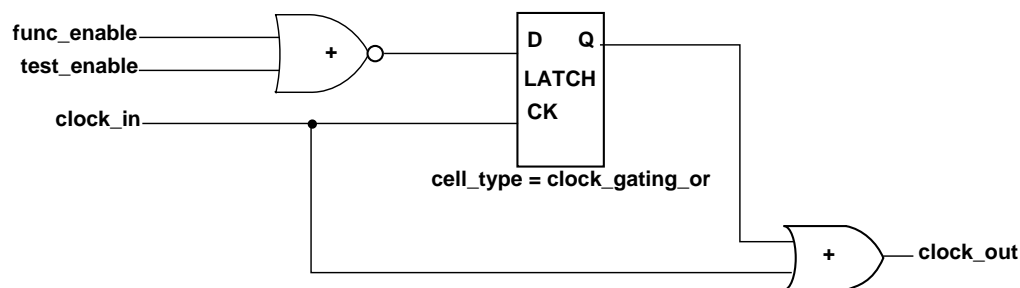
func_enable

test_enable

clock_in

clock_out

Figure 2-15. clock_gating_or with test_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model negedge_clk_gater_te
  (clk_out, clk_in, en, test_en)
  (
    model_source = verilog_module;
    cell_type = clock_gating_or;

    input (clk_in) (clock_in)
    input (en) (func_enable)
    input (test_en) (test_enable)
    output (clk_out) (clock_out)
    (
      primitive = _nor nor_inst ( en, test_en, enabled_b );
      instance = platch_udp lat_inst ( q_int, clk_in, enabled_b );
      primitive = _or or_inst ( q_int, clk_in, clk_out );
    )
  )
)
```

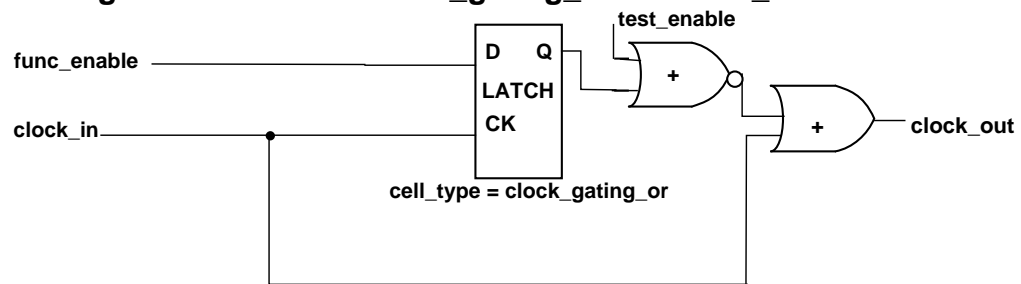
Note



A clock_gating_or cell does not support a test_enable after the latch. It is only supported on the input side of the latch.

Figure 2-16 below is an example of an Invalid Clock Gating OR cell.

Figure 2-16. Invalid clock_gating_or with test_enable



Clock Gating OR Cell with Both Enables and Asynch Enable

A clock gating OR requires the attributes described below.

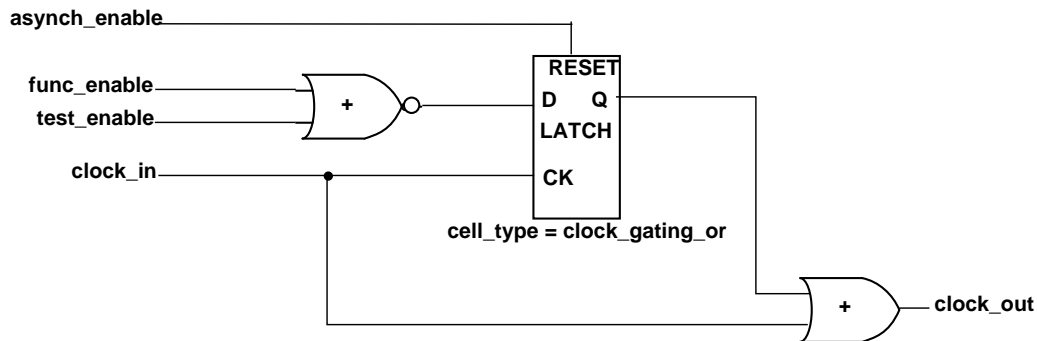
Model attributes:

cell_type = clock_gating_or

Pin attributes:

func_enable
 test_enable
 asynch_enable
 clock_in
 clock_out

Figure 2-17. clock_gating_or with test_enable and asynch_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model negedge_clk_gater_te
    (clk_out, clk_in, en, test_en)
    (
        model_source = verilog_module;
        cell_type = clock_gating_or;

        input (clk_in) (clock_in)
        input (en) (func_enable)
        input (test_en) (test_enable)
        input (asynch_en) (asynch_enable)
        output (clk_out) (clock_out)
    )
    (
        primitive = _nor nor_inst ( en, test_en, enabled_b );
        instance = _dlat lat_inst ( , asynch_en, clk_in, enabled_b, q_int, );
        primitive = _or or_inst ( q_int, clk_in, clk_out );
    )
)
```

Clock Gating OR Cell with Func Enable Only

A clock gating OR requires the attributes described below.

Model attributes:

cell_type = clock_gating_or

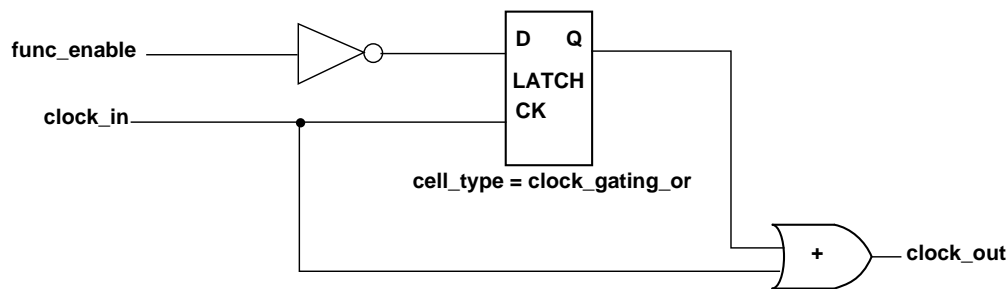
Illustrated Pin attributes:

func_enable

clock_in

clock_out

Figure 2-18. clock_gating_or



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model negedge_clk_gater
  (clk_out, clk_in, en)
  (
    model_source = verilog_module;
    cell_type = clock_gating_or;

    input (clk_in) (clock_in)
    input (en) (func_enable)
    output (clk_out) (clock_out)
    (
      primitive = _inv not_inst ( en, en_b );
      instance = platch_udp lat_inst ( q_int, clk_in, en_b );
      primitive = _or or_inst ( q_int, clk_in, clk_out );
    )
  )
  //***** CELL SELECTION ADDED *****
  // Ensure that Tessent test tools use negedge_clk_gater if they
  // insert a negedge clock gater.
  // Users can specify a (potentially different) synchronizer_
  // cell for each technology.
  // In this example, technology "tech_1" should use clock gaters with
  // only system enable, "tech_2" should use clock gaters with both
  // system enable and test enable.

  dft_cell_selection(tech_1) {
    clock_gating_or = negedge_clk_gater;
  }
```

```
dft_cell_selection(tech_2) {  
    clock_gating_or = negedge_clk_gater;  
}
```

Clock Gating AND Cell With Both Enables

A clock gating AND requires the attributes described below.

Model attributes:

cell_type = clock_gating_and

Illustrated Pin attributes:

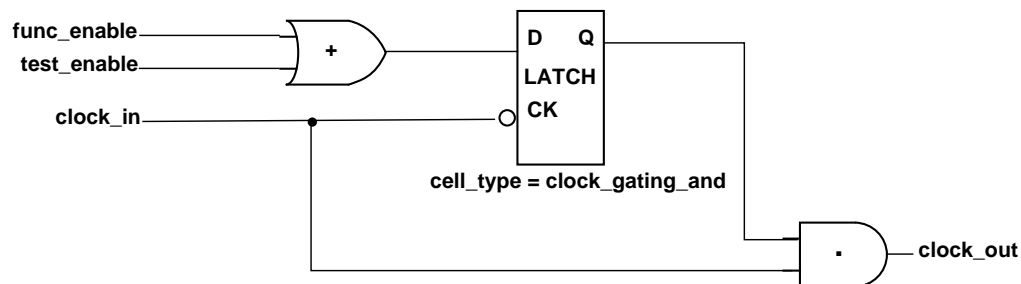
func_enable

test_enable

clock_in

clock_out

Figure 2-19. clock_gating_and with test_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model posedge_clk_gater_te
  (clk_out, clk_in, en, test_en)
  (
    model_source = verilog_module;
    cell_type = clock_gating_and;

    input (clk_in) (clock_in)
    input (en) (func_enable)
    input (test_en) (test_enable)
    output (clk_out) (clock_out)
    (
      primitive = _or or_inst ( en, test_en, enabled );
      instance = nlatch_udp lat_inst ( q_int, clk_in, enabled );
      primitive = _and and_inst ( q_int, clk_in, clk_out );
    )
  )
)
```

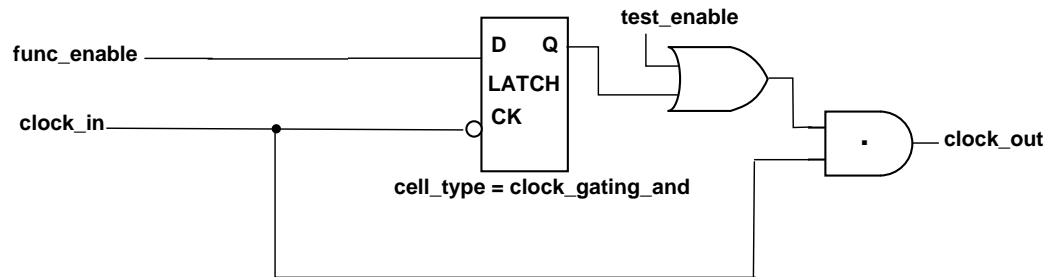
Note



A clock_gating_and cell does not support a test_enable after the latch. It is only supported on the input side of the latch.

Figure 2-20 below is an example of an invalid Clock Gating AND cell.

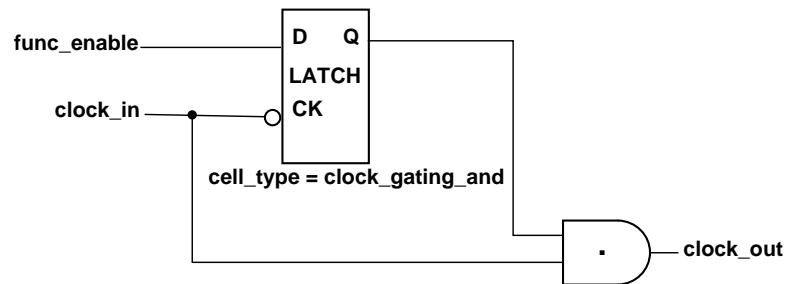
Figure 2-20. Invalid clock_gating_and with test_enable



Clock Gating AND Cell With Func Enable Only

A clock gating AND requires the attributes described below.

Figure 2-21. clock_gating_and



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model posedge_clk_gater
    (clk_out, clk_in, en)
(
    model_source = verilog_module;
    cell_type = clock_gating_and;

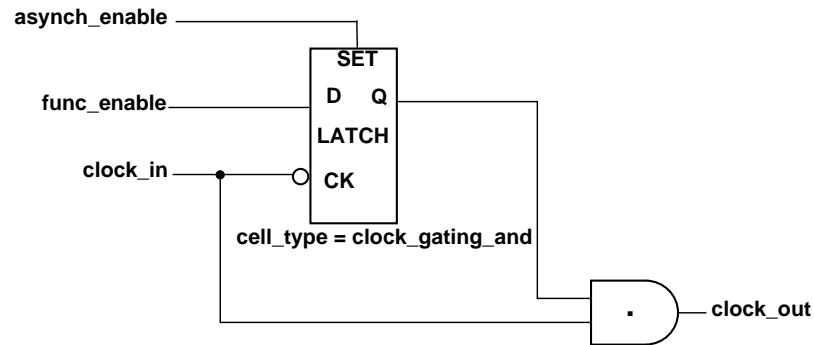
    input (clk_in) (clock_in)
    input (en) (func_enable)
    output (clk_out) (clock_out)
    (
        instance = nlatch_udp lat_inst ( q_int, clk_in, en );
        primitive = _and and_inst ( q_int, clk_in, clk_out );
    )
)
//***** CELL SELECTION ADDED *****
//***** CELL SELECTION ADDED *****
// Ensure that Tessent test tools use posedge_clk_gater if they
// insert a posedge clock gater.
// Sometimes, users have multiple technology cells (low power and
// non low power for example) and they can specify a (potentially
// different) synchronizer_cell for each technology.
// In this example, technology "tech_1" should use clock gaters with
// only system enable, "tech_2" should use clock gaters with both
// system enable and test enable.

dft_cell_selection(tech_1) {
    clock_gating_and = posedge_clk_gater;
}
dft_cell_selection(tech_2) {
    clock_gating_and = posedge_clk_gater_te;
}
```

Clock Gating AND Cell with Func Enable and Asynch Enable

A clock gating AND requires the attributes described below.

Figure 2-22. clock_gating_and with asynch_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model posedge_clk_gater
  (clk_out, clk_in, en)
  (
    model_source = verilog_module;
    cell_type = clock_gating_and;

    input (clk_in) (clock_in)
    input (en) (func_enable)
    input (asynch_en) (asynch_enable)

    output (clk_out) (clock_out)
    (
      instance = _dlat lat_inst ( asynch_en, , clk_in, en, q_int, );
      primitive = _and and_inst ( q_int, clk_in, clk_out );
    )
  )
  //***** CELL SELECTION ADDED *****
  //***** CELL SELECTION ADDED *****
  // Ensure that Tessent test tools use posedge_clk_gater if they
  // insert a posedge clock gater.
  // Sometimes, users have multiple technology cells (low power and
  // non low power for example) and they can specify a (potentially
  // different) synchronizer_cell for each technology.
  // In this example, technology "tech_1" should use clock gaters with
  // only system enable, "tech_2" should use clock gaters with both
  // system enable and test enable.

  dft_cell_selection(tech_1) {
    clock_gating_and = posedge_clk_gater;
  }
  dft_cell_selection(tech_2) {
    clock_gating_and = posedge_clk_gater_te;
  }
}
```


Clock Gating AND Cell with Inverting Enables

A clock gating AND requires the attributes described below.

Model attributes:

cell_type = clock_gating_and

Illustrated Pin attributes:

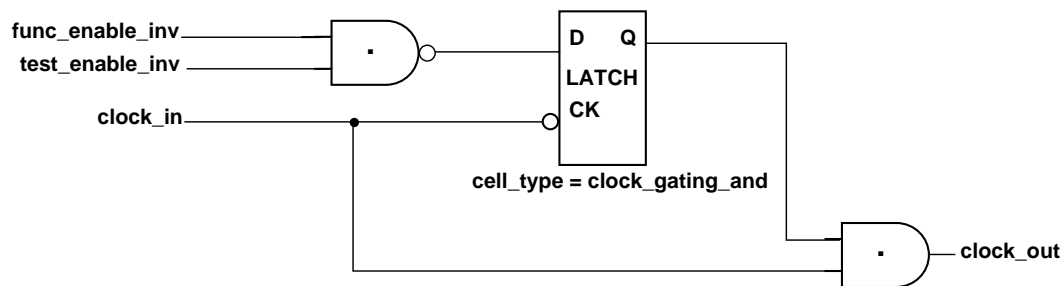
func_enable_inv

test_enable_inv

clock_in

clock_out

Figure 2-23. clock_gating_and With test_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model posedge_clk_gater_te
  (clk_out, clk_in, en, test_en)
  (
    model_source = verilog_module;
    cell_type = clock_gating_and;

    input (clk_in) (clock_in)
    input (en_b) (func_enable_inv)
    input (test_en_b) (test_enable_inv)
    output (clk_out) (clock_out)
    (
      primitive = _nand nand_inst ( en_b, test_en_b, enabled );
      instance = nlatch_udp lat_inst ( q_int, clk_in, enabled );
      primitive = _and and_inst ( q_int, clk_in, clk_out );
    )
  )
)
```

Positive-Edge Synchronizer Cell

A positive-edge synchronizer cell requires the attributes described below.

Model attributes:

cell_type = synchronizer_cell

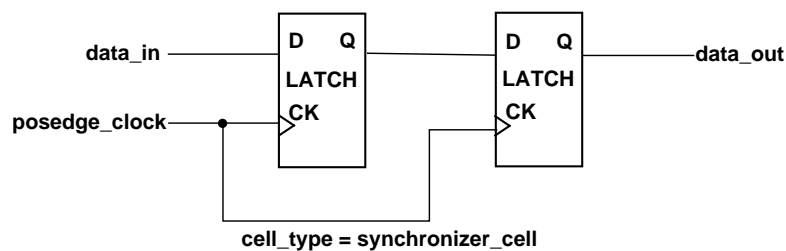
Pin attributes:

data_in

posedge_clock

data_out

Figure 2-24. Positive-Edge Synchronizer Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model posedge_synch
  (dout, clk, din)
  (
    model_source = verilog_module;
    cell_type = synchronizer_cell; // Added cell_type

    input (clk) (posedge_clock) // Added pin attribute
    input (din) (data_in) // Added pin attribute
    output (dout) (data_out) // Added pin attribute
    (
      instance = pos_dff_udp_wrap front_dff ( q_int, clk, din );
      instance = pos_dff_udp_wrap back_dff ( dout, clk, q_int );
    )
  )
  //***** CELL SELECTION ADDED *****
  // Ensure that Tessent test tools use posedge_synch if they insert a
  // synchronizer_cell in a data transfer between clock domains.
  // Sometimes, users have multiple technology cells (low power and
  // non low power for example) and they can specify a (potentially
  // different) synchronizer_cell for each technology.
  dft_cell_selection(tech_1) {
    posedge_synchronizer_cell = posedge_synch;
  }
}
```

Negative-Edge Synchronizer Cell

A negative-edge synchronizer cell requires the attributes described below.

Model attributes:

cell_type = synchronizer_cell

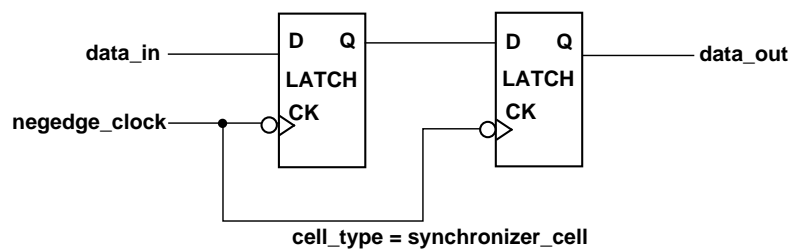
Pin attributes:

data_in

negedge_clock

data_out

Figure 2-25. Negative-Edge Synchronizer Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you added.

```
model negedge_synch
  (dout, clk, din)
  (
    model_source = verilog_module;
    cell_type = synchronizer_cell;

    input (clk) (negedge_clock)
    input (din) (data_in)
    output (dout) (data_out)
  )
  (
    instance = neg_dff_udp_wrap front_dff ( q_int, clk, din );
    instance = neg_dff_udp_wrap back_dff ( dout, clk, q_int );
  )
)
//***** CELL SELECTION ADDED *****
// Ensure that Tessent test tools use negedge_synch if they insert a
// synchronizer_cell in a data transfer between clock domains.
// Sometimes, users have multiple technology cells (low power and
// non low power for example) and they can specify a (potentially
// different) synchronizer_cell for each technology.
dft_cell_selection(tech_1) {
  negedge_synchronizer_cell = negedge_synch;
}
```


Chapter 3

Cell Library

This chapter describes how to define test simulation models and macros for a cell, how to define test attributes for a cell and its ports or pins, and how to include other library files by reference (usually as the definition of sub-models referenced to build larger more complex simulation models).

In addition, this chapter provides functional descriptions of each primitive, and the pin order for instantiating each primitive within a library model for primitives supported by Tessent Scan, Tessent FastScan, FlexTest, and Tessent TestKompress. These library models can also be used in the LV Flow.

This chapter includes the following topics:

Defining Cell Information	62
Cell Library Overview	62
Supported Library Syntax	63
Defining a Cell Library	65
Defining Cell Models	74
Defining Hardware	91
Defining Instance Attributes	96
Defining Pin Attributes	98
Internal Faults	120
Support of Arrays Within Library Models	124
Example Scan Definitions	125
Defining Macros	127
Reusing a Model Definition	127
Reading Multiple Libraries	128
Verilog Primitives	129
Supported Primitives	131

Defining Cell Information

The cell library file describes library cells to be used during RTL generation or test logic insertion, and also for test and simulation.

This section first briefly describes the organization of the cell library file, the required syntax for each of the major sections in the cell library, and describes how the sections fit in a typical cell library. Subsequent sections provide complete information about each of these sections and examples.

Cell Library Overview

As shown in the following example, the cell library is typically composed of *library* level information which typically applies to all subsequent processing, *model* level information which applies only to a particular model, and *pin* level information which applies only to a particular pin.

```
array_delimiter = "["; // library level
model equivalent_cell = defined_model_name; // library level
regexp_verbosity = {verbose | silent}; // library level
cell_attributes() // library level
...
dft_cell_selection (cell_technology_name) ( ) // library level

model model_name (list_of_pins) ( ) // model level
    cell_type = ... // model level
    simulation_function = ... // model level
    scan_type = ... // model level
    (input (pin_name) (...)) // pin level
    ...
```

A brief syntactic overview showing these sections and how they fit within a typical cell library is presented here first. More detailed discussions and examples follow.

Library Definition Section Overview

The library level defines information that is not specific to a particular model but applies to all cells and processing that occur after the information is read by the library parser:

```
array_delimiter = "[";
model equivalent_cell = defined_model_name;
regexp_verbosity = {verbose | silent};
cell_attributes()
...
dft_cell_selection (cell_technology_name_1) ( )
dft_cell_selection (cell_technology_name_2) ( )
...
```

For more information on defining library level information, see “[Defining a Cell Library](#).”

Model Definition Section Overview

The model level defines information about a single cell as shown below.

```
model model_name (list_of_pins)
  model_attribute statements
  input (input_pins) (list_of_input_attributes)
  // Usually more input statements (can be one per pin)
  inout (inout_pins) (list_of_inout_attributes)
  // Can be more inout statements (can be one per pin)..
  output (output_pins) (list_of_output_attributes)
  // Can be more output statements (can be one per pin)
  intern (net_names) (list_of_net_attributes)
  // Needed for internal vector net only, and typically only
  // the array net attribute is used (to declare the dimensions).
  (
    instance_statement; ... // Instances of other library models
    primitive_statement; ... // Instances of supported primitives
    ...
  )
```

For information on defining model level information, see “[Defining Cell Models.](#)”

Pin Definition Section Overview

The first set of parentheses shown above after input, inout, output, or intern surround a list of pin names (or net names for the intern statement). The second set of parens are required even if the pin or intern have no pin or net attributes to declare.

If there are attributes, they are mentioned within the second set of parens, and apply to all pins (or nets) in the first set. Multiple attributes for the same pin(s) are separated by semicolons within the list as shown here:

```
pin_attribute1; pin_attribute2; ... // Attributes for this pin
```

For information on defining pin level information, see “[Defining Pin Attributes.](#)”

Note



The *list_of_<some>_attributes* argument is described in section “[Defining Pin Attributes](#)” where <some> can be *input*, *output*, or *inout*.

Supported Library Syntax

The legal characters for user-defined names and legal block delimiters are described below along with examples.

Legal Characters for User-defined Names

The legal characters for user-defined names such as `model_name`, `input_pins`, `intern_nodes`, `inout_pins`, and so on are letters (Aa-Zz), numbers (0-9), and the underscore character “_”. If any other character is used, the entire name must be enclosed in double quotes.

Legal Block Delimiters

You can use either curly brackets or round brackets (parentheses) as legal block delimiters.

Here are some examples:

```
model model_name (list_of_pins) (model attributes)
or
model model_name (list_of_pins) {model attributes}
```

```
input (pin_name) (list_of_input_attributes)
or
input (pin_name) {list_of_input_attributes}
```

```
dft_cell_selection (<cell_technology_name>) ()
or
dft_cell_selection (<cell_technology_name>){}
```

```
cell_attributes ()
or
cell_attributes {}
```


Defining a Cell Library

All cell information related to the definition of the library itself is grouped together in the library description.

Here is the syntax of the library description:

```
array_delimiter = "[";
model_equivalent_cell = defined_model_name;
regexp_verbosity = {verbose | silent};

dft_cell_selection (cell_technology_name_1) ( ) // See Cell Selection
dft_cell_selection (cell_technology_name_2) ( )
...
dft_cell_selection (cell_technology_name_k) ( )

cell_attributes( // See Cell Attributes
    regexp = {anchors | no_anchors | glob | off};
    cell ("modelname_regexp") (
        // model_attribute statements, see Model-Level Attribute Descriptions.
        // For legal pin_level attributes, see Defining Pin Attributes
        input ("pinname_regexp") (list_of_input_attributes)
        output ("pinname_regexp") (list_of_output_attributes)
        inout ("pinname_regexp") (list_of_inout_attributes)
    ) // end cell
    ...
    regexp = off;
    cell (modelname) (
        // model_attribute statements, see Model-Level Attribute Descriptions.
        // For legal pin_level attributes, see Defining Pin Attributes
        input (pinname) (list_of_input_attributes)
        output (pinname) (list_of_output_attributes)
        inout (pinname) (list_of_inout_attributes)
    ) // end cell
    ...
) // end cell_attributes

model model_name (list_of_pins) ( // See Defining Cell Models
    // model_attribute statements, see Model-Level Attribute Descriptions.
    // For legal pin_level attributes, see Defining Pin Attributes
    input (pin_name) (list_of_input_attributes)
    output (pin_name) (list_of_output_attributes)
    inout (pin_name) (list_of_inout_attributes)
    (
        //hardware_statements, see Defining Hardware
    )
) // end model
...
```

The following list describes each of the global library attributes in more detail:

- *array_delimiter*

Keyword that specifies the use of arrays in library models. For more information, see [“Support of Arrays Within Library Models.”](#)

- *model new_modelname = defined_model_name;*

Statement that specifies to make a copy of a cell already fully described in the library and assign the *new_modelname* to it.

Note

The copy occurs prior to processing `cell_attributes()` or other sources of attributes for the defined model. Only attributes already part of the defined model description (that is, within its syntactic model definition scope) are inherited by the new model when copied. For more information, see [“Reusing a Model Definition.”](#)

- *regexp_verbosity*

Keyword that specifies the reporting that occurs during regular expression matching within *cell_attributes* statements as follows:

- *verbose* — Specifies to report pin name matches resulting from regular expression matching. By default, matching pin names are not reported.
- *silent* — Specifies to not report matching cell names resulting from regular expression matching. By default, matching cell names are reported.

See additional description as follows:

- The *dft_cell_selection* syntax is described in [“Cell Selection.”](#)
- The *cell_attributes* syntax is described in [“Cell Attributes.”](#)
- The *model* syntax is described in [“Model Statement Descriptions.”](#)
- The *model_attributes* syntax is described in [“Model-Level Attribute Descriptions.”](#)
- Pin attributes are described in [“Defining Pin Attributes.”](#)

Cell Selection

The `dft_cell_selection` keyword occurs at the library syntax level and defines the list of cells to use for test logic insertion.

Here is the `dft_cell_selection` syntax:

```
dft_cell_selection (cell_technology_name) (
    clock_buffer = model_name;
    clock_inverter = model_name;
    clock_mux = model_name_2_data_input, model_name_3_data_input, ... ;
    clock_and = model_name_2_input, model_name_3_input, ... ;
    clock_or = model_name_2_input, model_name_3_input, ... ;
    clock_gating_and = model_name;
    clock_gating_or = model_name;
    buffer = model_name;
    inverter = model_name;
    and = model_name_2_input, model_name_3_input, ... ;
    nand = model_name_2_input, model_name_3_input, ... ;
    or = model_name_2_input, model_name_3_input, ... ;
    nor = model_name_2_input, model_name_3_input, ... ;
    xor = model_name_2_input, model_name_3_input, ... ;
    mux = model_name_2_data_input, model_name_3_data_input, ... ;
    posedge_dff = model_name;
    negedge_dff = model_name;
    active_high_latch = model_name;
    active_low_latch = model_name;
    posedge_synchronizer_cell = model_name;
    negedge_synchronizer_cell = model_name;
    posedge_scan_cell = model_name;
    negedge_scan_cell = model_name;
    default_bcell_libs = (
        bcell_lib_name1 = class_name [subclass_list];
        bcell_lib_name2 = class_name [subclass_list];
        ...
        bcell_lib_nameN = class_name [subclass_list];
    ) //end default_bcells_libs
) //end dft_cell_selection
```

The following list describes each of the `dft_cell_selection` statements in more detail:

- cell_technology_name**
 A user-provided name that determines the technology of the cells inside the named `dft_cell_selection()` statement. If multiple `cell_selection` statements have the same `cell_technology_name`, they are treated as if they are part of the same `cell_selection()` and the union of all such named `cell_selections` is kept. If a selection is repeated with a different `model_name`, a warning is issued and the last parsed is kept as the selection for that technology.
- clock_buffer**
 Specifies the `clock_buffer` for the specified `cell_technology_name` of the enclosing wrapper. Must point to a valid buffer cell (model). See the `cell_type` `clock_buffer` in the [Model-Level Attribute Descriptions](#) for details.

- **clock_inverter**
Specifies the clock_inverter for the specified cell_technology_name of the enclosing wrapper. Must point to a valid inverter cell (model). See the cell_type clock_inverter in the [Model-Level Attribute Descriptions](#) for details.
- **clock_mux**
Specifies the clock_mux for the specified cell_technology_name of the enclosing wrapper. Must point to a valid mux cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type clock_mux in the [Model-Level Attribute Descriptions](#) for details.
- **clock_and**
Specifies the clock_and for the specified cell_technology_name of the enclosing wrapper. Must point to a valid and cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type clock_and in the [Model-Level Attribute Descriptions](#) for details.
- **clock_or**
Specifies the clock_or for the specified cell_technology_name of the enclosing wrapper. Must point to a valid or cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type clock_or in the [Model-Level Attribute Descriptions](#) for details.
- **clock_gating_and**
Specifies the clock_gating_and for the specified cell_technology_name of the enclosing wrapper. Must point to a valid clock_gating_and cell (model). See the cell_type clock_gating_and in the [Model-Level Attribute Descriptions](#) for details.
- **clock_gating_or**
Specifies the clock_gating_or for the specified cell_technology_name of the enclosing wrapper. Must point to a valid clock_gating_or cell (model). See the cell_type clock_gating_or in the [Model-Level Attribute Descriptions](#) for details.
- **buffer**
Specifies the buffer for the specified cell_technology_name of the enclosing wrapper. Must point to a valid buffer cell (model). See the cell_type buffer in the [Model-Level Attribute Descriptions](#) for details.
- **inverter**
Specifies the inverter for the specified cell_technology_name of the enclosing wrapper. Must point to a valid inverter cell (model). See the cell_type inverter in the [Model-Level Attribute Descriptions](#) for details.
- **and**
Specifies the and for the specified cell_technology_name of the enclosing wrapper. Must point to a valid and cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type and in the [Model-Level Attribute Descriptions](#) for details.

- **nand**
Specifies the nand for the specified cell_technology_name of the enclosing wrapper. Must point to a valid nand cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type nand in the [Model-Level Attribute Descriptions](#) for details.
- **or**
Specifies the or for the specified cell_technology_name of the enclosing wrapper. Must point to a valid or cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type or in the [Model-Level Attribute Descriptions](#) for details.
- **nor**
Specifies the nor for the specified cell_technology_name of the enclosing wrapper. Must point to a valid nor cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type nor in the [Model-Level Attribute Descriptions](#) for details.
- **xor**
Specifies the xor for the specified cell_technology_name of the enclosing wrapper. Must point to a valid xor cell (model). You can specify at most one 2-input cell, one 3-input cell, and so on. See the cell_type xor in the [Model-Level Attribute Descriptions](#) for details.
- **mux**
Specifies the mux for the specified cell_technology_name of the enclosing wrapper. Must point to a valid mux cell (model). You can specify at most one 2-data input cell, one 3-data input cell, and so on. See the cell_type mux in the [Model-Level Attribute Descriptions](#) for details.
- **negedge_dff**
Specifies the negedge_dff for the specified cell_technology_name of the enclosing wrapper. Must point to a valid dff cell (model). See the cell_type dff in the [Model-Level Attribute Descriptions](#) for details.
- **posedge_dff**
Specifies the posedge_dff for the specified cell_technology_name of the enclosing wrapper. Must point to a valid dff cell (model). See the cell_type dff in the [Model-Level Attribute Descriptions](#) for details.
- **active_high_latch**
Specifies the active_high_latch for the specified cell_technology_name of the enclosing wrapper. Must point to a valid latch cell (model). See the cell_type latch in the [Model-Level Attribute Descriptions](#) for details.
- **active_low_latch**
Specifies the active_low_latch for the specified cell_technology_name of the enclosing wrapper. Must point to a valid latch cell (model). See the cell_type latch in the [Model-Level Attribute Descriptions](#) for details.

- **posedge_synchronizer_cell**
Specifies the `posedge_synchronizer_cell` for the specified `cell_technology_name` of the enclosing wrapper. Must point to a valid `synchronizer_cell` cell (model). See the `cell_type synchronizer_cell` in the [Model-Level Attribute Descriptions](#) for details.
- **negedge_synchronizer_cell**
Specifies the `negedge_synchronizer_cell` for the specified `cell_technology_name` of the enclosing wrapper. Must point to a valid `synchronizer_cell` cell (model). See the `cell_type synchronizer_cell` in the [Model-Level Attribute Descriptions](#) for details.
- **posedge_scan_cell**
Specifies the `posedge_scan_cell` for the specified `cell_technology_name` of the enclosing wrapper. Must point to a valid mux scan_cell (model). See the `cell_type scan_cell` in the [Model-Level Attribute Descriptions](#) for details.
- **negedge_scan_cell**
Specifies the `negedge_scan_cell` for the specified `cell_technology_name` of the enclosing wrapper. Must point to a valid mux scan_cell (model). See the `cell_type scan_cell` in the [Model-Level Attribute Descriptions](#) for details.
- **default_bcell_libs**
Used by the LV Flow only. Specifies the custom boundary-scan cells used in a design and their correspondence to pad cells. The specified boundary-scan cells are instantiated when a corresponding pad cell is identified in the netlist. This attribute only applies to the `cell_type pad`. The syntax of this attribute is:

```
default_bcell_libs = (  
    bcell_lib_name1 = class_name (subclass_list);  
    bcell_lib_name2 = class_name (subclass_list);  
    ...  
    bcell_lib_nameN = class_name [subclass_list]);
```

- **bcell_lib_nameX** — Specifies the name of a boundary-scan cell.
 - **class_name** — Specifies one of the following literal cell types:
 - in_cell** — Specifies `bcell_lib_nameX` cell is an input cell.
 - out_cell** — Specifies `bcell_lib_nameX` cell is an output cell.
 - inout_cell** — Specifies `bcell_lib_nameX` cell is a bidirectional cell.
 - enable_cell** — Specifies `bcell_lib_nameX` cell is an enable cell.
 - **subclass_list** — A syntactically optional comma-separated list enclosed in round brackets that specifies that `bcell_lib_nameX` is one or more of the following: `mux_inside_pad`, `two_state`, `diff_current`, `diff_voltage`, `force_disable`, `pull0`, `pull1`, `ac_dot6`, `acm_dot6`, `from_pad_dot6`, `inv_in`, `inv_out`, `sample_only`, `muxed_out`, `nonjtag`, `enable_low`, `enable_high`, `hold`, `unused_in_direction`, `unused_out_direction`, `open_drain`, `open_source`, `sample_pad`. For more information on a specific subclass, see [DefaultBcells](#) in the *ETAssemble Tool*

Reference manual and [Subclasses for Complex Boundary-Scan Cells](#) in the
Embedded Test Hardware Reference.

For more information, see Appendix A, [Attributes and the Tessent Cell Library](#)

Cell Attributes

All cell attribute information may be grouped together in the library description

The cell attributes syntax is described below

```
cell_attributes(  
  regexp = {anchors | no_anchors | glob};  
  cell ("modelname_regexp") (  
    model_attributes  
    input ("pinname_regexp") (list_of_input_attributes)  
    output ("pinname_regexp") (list_of_output_attributes)  
    inout ("pinname_regexp") (list_of_inout_attributes)  
  ) // end cell  
  ...  
  regexp = off;  
  cell (modelname) (  
    model_attributes  
    input (pinname) (list_of_input_attributes)  
    output (pinname) (list_of_output_attributes)  
    inout (pinname) (list_of_inout_attributes)  
  ) // end cell  
  ...  
) // end cell_attributes
```

The following list describes each component of the *cell_attributes* statement.

Note



The *list_of_<some>_attributes* argument is described in section “[Defining Pin Attributes](#)” where *<some>* can be *input*, *output*, or *inout*.

- **cell_attributes | model_attributes**

Keyword that defines a syntactic section that can only contain attributes. These attributes are associated with models and pins in ATPG cell libraries using simple and/or regular expression matching. Either *cell_attributes* or *model_attributes* are legal keywords for this use and can be used interchangeably.

Note



You can use curly brackets in place of round brackets (parentheses) for the *cell_attributes* statement.

- **regexp**

A statement within the *cell_attributes* section that specifies the style of regular expression matching to be used in all subsequent expressions until the next regexp statement within the same *cell_attributes* syntactic section is encountered. The syntax is:

```
regexp = {anchors | no_anchors | glob | off};
```

- **anchors** — This is the default at the start of each new *cell_attributes* section. When regexp is set to *anchors*, the regexp string is surrounded by '^' '\$' to prevent substring

matches. As an example, this ensures "sff.*" matches only names that start with "sff", rather than matching any name with "sff" anywhere in it.

- **no_anchors** — This value enables substring matching. That is, "fre" matches "offreg" because the latter contains "fre" within it. If **regex** is set to **anchors**, they will not match.
- **glob** — This value converts "*" to ".*" and "?" to "." only. (This is not true globbing.) This value allows back reference support with globbing wildcards, because **regex** is always used after the substitutions. In this case, only escaped "*" and "?" are not converted. For example: "a*" would not be converted but "a*" and "a*" would be converted. For more information, refer to the UNIX man pages for **regex** and **7 regex**.
- **off** — This value allows simple namestring comparisons to determine matches. This option provides a way of providing attribute-only information for a single unique model name and applies to the last model name in the case of duplicate models, because only that model is preserved for attribute processing.

Note



In the event of conflicting attribute values, which may happen, for example, if attributes in a *cell_attributes* section conflict with attributes already on a model or elsewhere, the tool determines the attribute value as follows:

If a model outside any *cell_attributes* section specifies that attribute, that model determines the value.

Else, if that model does not specify that attribute, the last simple (**regex** off) model that specified the attribute in a *cell_attributes* section determines the value.

Else, if only **regex** models (which must be in a *cell_attributes* section) specify the value, the last **regex** to specify the value determines the value assigned.

• **cell | model**

Keyword that identifies a single cell model in the ATPG library to which regular expression matching will be applied. The syntax is as follows:

```
cell ("modelname_regex") (
    input ("pinname_regex") (list_of_input_attributes)
    output ("pinname_regex") (list_of_output_attributes)
    inout ("pinname_regex") (list_of_inout_attributes)
) // end cell
```

Note



The *list_of_<some>_attributes* argument is described in [“Defining Pin Attributes.”](#)

- **modelname_regexp | pinname_regexp**

Specifies the regular expression to be applied during pattern matching. The *modelname_regexp* (or *pinname_regexp* expression) is surrounded by double quotes. The quoted *modelname_regexp* (or *pinname_regexp* statement) is contained within parentheses as shown in the following examples:

```
cell ("modelname_regexp")
input ("pinname_regexp")
```

For more examples of regular expressions, see “[Method2](#)”.

Defining Cell Models

Model-level attributes are associated with the model in whose model section they are defined.

All statements following the initial *model model_name* statement and before the final *model* ending bracket are considered model-level attributes unless they are nested inside brackets, either “()” or “{}”. For example, attributes following input, output, or inout pin names are inside their own brackets and are, therefore, not model-level attributes, but pin-level attributes.

Model-level attributes can be of two different broad classes:

- Port attributes
- Cell attributes

Port attributes describe the direction of the model pins as well as information about the pins for test insertion. See "Defining Pin Attributes".

All information related to a cell model is grouped together in the model description as follows:

```
model model_name (list_of_pins) (
  model_source = verilog_module;
  cell_type = { clock_gating_and | clock_gating_or |
    buffer | inverter | or | and | nand | nor | xor | mux |
    clock_buffer | clock_inverter | clock_and | clock_or | clock_mux |
    dff | latch | synchronizer_cell | scan_cell | pad | prohibited };
  simulation_function = { clock_gating_and | clock_gating_or |
    buffer | inverter | or | and | nand | nor | xor | mux |
    dff | latch | synchronizer_cell | scan_cell | tie0 | tie1 };
  pad_ac;
  pad_nonjtag;
  pad_ac_lp_time = <time_in_seconds>;
  pad_ac_hp_time = <time_in_seconds>;
  pad_ac_hp_on_chip;
  bcell_lib_name = <bcell_lib_name>
  nonscan_model = model_name | model_name (list_of_pins);
  scan_length = integer;
  gated_out_scan_cell;
  mode(
    pad_ac;
    pad_ac_lp_time = <time_in_seconds>;
```

```

pad_ac_hp_time = <time_in_seconds>;
pad_ac_hp_on_chip [Yes | No];
input (pin_name) (
    [pad_to_pad] [pad_pad_io] [pad_pad_io_inv] [pad_enable_high]
    [pad_enable_low] [pad_to_sje_mux_low] [pad_to_sje_mux_high]
    [pad_to_sji_mux] [pad_to_sjo_mux] [pad_select_jtag_enable]
    [pad_select_jtag_in] [pad_select_jtag_out]
    [pad_init_data_dot6] [pad_init_clock_dot6]
    [pad_init_data_inv_dot6] [pad_ac_mode_dot6]
    [pad_data_inv] [pad_diff_voltage] [pad_diff_current]
    [pad_from_io] [pad_from_io_inv] [pad_tie0] [pad_tie1]
    [pad_open])
) // end input section of mode

input (pin_name2) ()
...
input (pin_nameN) ()

output (pin_name) (
    [pad_from_pad] [pad_pad_io] [pad_pad_io_inv]
    [pad_from_sje_mux] [pad_from_sji_mux] [pad_from_sjo_mux]
    [pad_sample_pad] [pad_data_inv] [pad_diff_voltage]
    [pad_diff_current] [pad_open_drain] [pad_open_source]
    [pad_to_io] [pad_to_io_inv] [pad_pull0] [pad_pull1];
) // end output section of mode
output (pin_name2) ()
...
output (pin_nameN) ()

inout (pin_name) (
    [pad_pad_io] [pad_pad_io_inv] [pad_diff_voltage]
    [pad_diff_current] [pad_open_drain]
    [pad_open_source] [pad_pull0] [pad_pull1]
) // end inoutsection of mode

inout (pin_name2) ()
...
inout (pin_nameN) ()

) // end mode

// Completed model level statements, including pad modes if any.
// Define model interface port directions, and attributes.

intern (intern_nodes) (intern_attributes)
input (pin_names) (list_of_input_attributes)
output (pin_names) (list_of_output_attributes)
inout (pin_names) (list_of_inout_attributes)

// Define hardware simulation function using ATPG library
// primitives and instances
(
    //hardware_statements, see Defining Hardware
)

) // end model_name

```

Model Statement Descriptions

The components of the cell model definition are described in more detail in this section.

The cell model definition consists of the following parts:

- **model**

Keyword that defines a single library cell in the technology library. The model statement requires two components: the *model_name* and the *list_of_pins*. The syntax is as follows:

```
model model_name (...  
    ...  
)
```

- **model_name**

Specifies the name of the library cell in your design data. When translating from Verilog, the *model_name* should match the Verilog module name, or UDP primitive name. The *model_name* argument allows you to reference the cell.

- **list_of_pins**

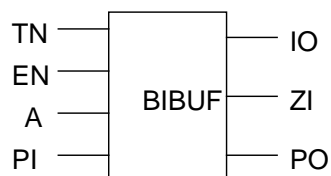
Specifies the interface pins on the cell boundary which can include input, output, and bidirectional pins. The syntax is as follows:

```
model model_name (list_of_pins) (  
    ...  
)
```

For example, the model name, BIBUF, for the bidirectional buffer and its interface pins IO, A, EN, TN, PI, ZI, and PO in the model statement is specified as follows:

```
model BIBUF (IO, A, EN, TN, PI, ZI, PO) (  
    ...  
)
```

Figure 3-1. Bidirectional Buffer



In another example, the model name SDFF is assigned to a scan D flip-flop and the names D, CLK, TI, TE, Q, and QN are assigned to its interface pins as follows:

```
model SDFF(D, CLK, TI, TE, Q, QN) (  
    .....  
)
```

Model-Level Attribute Descriptions

The model-level attributes are described in this section.

Listed below are the model-level attributes:

- **input (port_name_list) (input_port_attributes)**

The `port_name_list` which follows the keyword “input” and is contained within the first set of parentheses can be a single name, or a comma-separated list of names. For example, a single port declaration and a two port declaration would be, respectively:

```
input (in1)
input (in1, in2)
```

The `input_port_attributes` list is contained in the second set of parentheses. These attributes apply to all ports in the list. For that reason, ports like the `scan_enable` of a `scan_cell` must be listed in a separate input statement because only that pin has the `scan_enable` pin attribute.

If the port is a vector, its size must be declared in the second set of parentheses. For example, to declare an 8-bit port named `addr`, you use the following syntax:

```
input (addr) (array = 7:0)
```

For information on specific model-level input pin attributes (normal pin attributes outside a mode section), including more details on the array declaration, see “[Defining Pin Attributes](#).”

- **output (pin_name) (output_pin_attributes)**

The *input* statement description also applies to the output statement. For information on specific model-level output attributes (normal pin attributes outside a mode section), see “[Defining Pin Attributes](#).”

- **inout (pin_name) (inout_pin_attributes)**

The *input* statement description also applies to the inout statement. For information on specific model-level inout attributes (normal pin attributes outside a mode section), see “[Defining Pin Attributes](#).”

- **cell_type**

Keyword that identifies the cell (model) function. `pad` is informative. All other legal values specify the cell transfer function and typically also require pin functions to be utilized when test logic circuitry is inserted into the design to make it more testable. All the `cell_type` except `buffer`, `inverter`, `or`, `and`, `nand`, `nor`, `xor`, and `pad` must also have pin function attributes to identify the appropriate pin connections for test logic insertion.

The syntax is as follows:

```
cell_type = {clock_gating_and | clock_gating_or |
  buffer | inverter | or | and | nand | nor | xor | mux |
  clock_buffer | clock_inverter | clock_and | clock_or | clock_mux |
  dff | latch | synchronizer_cell | scan_cell | pad | prohibited};
```

Table 3-1 lists all the cell_types, which are described in detail following this table.

Table 3-1. Cell_types

cell_type	Description
clock_gating_and	See “ clock_gating_and ”.
clock_gating_or	See “ clock_gating_or ”.
buffer	See “ buffer ”.
inverter	See “ inverter ”.
or	See “ or ”.
and	See “ and ”.
nand	See “ nand ”.
nor	See “ nor ”.
xor	See “ xor ”.
mux	See “ mux ”.
clock_buffer	See “ clock_buffer ”.
clock_inverter	See “ clock_inverter ”.
clock_and	See “ clock_and ”.
clock_or	See “ clock_or ”.
clock_mux	See “ clock_mux ”.
dff	See “ dff ”.
latch	See “ latch ”.
synchronizer_cell	See “ synchronizer_cell ”.
scan_cell	See “ scan_cell ”.
pad	See “ pad ”.
prohibited	See “ prohibited ”.

- **clock_gating_and**

Used by LV Flow (ETAssemble, and ETAnalysis). Specifies a latched-enabled clock gating cell that forces the clock low when disabled. The referenced cell (model) must have an input pin with clock_in attribute, and another input pin with either func_enable (enabled when input is 1 or high) or func_enable_inv (enabled when 0/low) attribute. The cell should have a single output with clock_out attribute. Optionally, the cell can also be enabled by an additional test_enable input (enabled when input is 1 or high) or test_enable_inv (enabled when 0 or low). The cell can also be optionally immediately enabled by an additional asynch_enable input (enabled when input is 1 or high) or asynch_enable_inv (enabled when 0 or low).

The cell can also be optionally immediately disabled by an additional `asynch_disable` input (disabled when input is 1 or high) or `asynch_disable_inv` (disabled when 0 or low). For more information, see *Application Note Timing Constraints and Clock Tree Synthesis*.

Note



If a `clock_gating_and` cell has a single functional enable, or if it has two enables with assigned pin functions, then the `cell_type` is learned and set automatically. It is also learned if it has two enables and exactly one of them starts with “te”, and that input is considered the `test_enable` (or `test_enable_inv` depending on the polarity).

o **clock_gating_or**

Used by LV Flow (ETAssemble, and ETAnalysis). Specifies a latched-enabled clock gating cell that forces the clock high when disabled. The referenced cell (model) must have an input pin with `clock_in` attribute, and another input pin with either `func_enable` (enabled when input is 1 or high) or `func_enable_inv` (enabled when 0/low) attribute. The cell should have a single output with `clock_out` attribute. Optionally, the cell can also be enabled by an additional `test_enable` input (enabled when input is 1 or high) or `test_enable_inv` (enabled when 0 or low). The cell can also be optionally immediately enabled by an additional `asynch_enable` input (enabled when input is 1 or high) or `asynch_enable_inv` (enabled when 0 or low). The cell can also be optionally immediately disabled by an additional `asynch_disable` input (disabled when input is 1 or high) or `asynch_disable_inv` (disabled when 0 or low). For more information, see *Application Note Timing Constraints and Clock Tree Synthesis*.

Note



If a `clock_gating_or` cell has a single functional enable, or if it has two enables with assigned pin functions, then the `cell_type` is learned and set automatically. It is also learned if it has two enables and exactly one of them starts with “te”, and that input is considered the `test_enable` (or `test_enable_inv` depending on the polarity).

o **buffer**

Used by LV Flow (ETAssemble, and ETAnalysis), and by Tessent Scan. Specifies a buffer to be inserted on data paths. For more information, see [Buffer](#) in the *ETAssemble Tool Reference* manual or in the [Add Cell Models](#) in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

o **clock_buffer**

Used by LV Flow (ETAssemble, and ETAnalysis), and by Tessent Scan. Specifies a buffer to be inserted on clock paths. For more information, see [ClockBuffer](#) in the *ETAssemble Tool Reference* manual or in the [Add Cell Models](#) in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **inverter**

Used by LV Flow (ETAssemble, and ETAnalysis), and by Tessent Scan. Specifies an inverter cell to be inserted on data paths. For more information, see [Inverter](#) in the *ETAssemble Tool Reference* manual or on the [Add Cell Models](#) reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **clock_inverter**

Used by LV Flow (ETAssemble, and ETAnalysis), and by Tessent Scan. Specifies an inverter cell to be inserted on clock paths. For more information, see [ClockInverter](#) in the *ETAssemble Tool Reference* manual.

- **and**

All widths can be used by LV Flow (ETAssemble). 2-input can be used by LV Flow ETAnalysis and Tessent Scan. Specifies an And cell to be inserted on data paths. For more information, see [AND2](#) in the *ETAssemble Tool Reference* manual or [AND](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **clock_and**

All widths can be used by LV Flow (ETAssemble). 2-input can be used by LV Flow ETAnalysis and Tessent Scan. Specifies an And cell to be inserted on clock paths. For more information, see [ClockAnd](#) in the *ETAssemble Tool Reference* manual.

- **nand**

All widths can be used by LV Flow (ETAssemble). 2-input can be used by Mentor Tessent Scan. Specifies an Nand cell to be inserted on data paths. For more information, see [NAND](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **or**

All widths can be used by LV Flow(ETAssemble). 2-input can be used by LV Flow ETAnalysis and Mentor Tessent Scan. Specifies an Or cell to be inserted on data paths. For more information, see [Or2](#) in the *ETAssemble Tool Reference* manual or [OR](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **clock_or**

All widths can be used by LV Flow(ETAssemble). 2-input can be used by LV Flow ETAnalysis and Mentor Tessent Scan. Specifies an Or cell to be inserted on clock paths. For more information, see [ClockOr](#) in the *ETAssemble Tool Reference* manual.

- **nor**

All widths can be used by LV Flow (ETAssemble). 2-input can be used by Mentor Tessent Scan. Specifies an Nor cell to be inserted on data paths. For more information, see [NOR](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **xor**

All widths can be used by LV Flow (ETAssemble). 2-input can be used by Mentor Tessent Scan. Specifies an Xor cell to be inserted on data paths. For more information, see [XOR](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **mux**

Used by LV Flow and Tessent Scan. Specifies a 2-to-1 data path multiplexer. The cell (model) must have an input with mux_select attribute, an input with mux_in0 attribute, and an input with mux_in1 attribute. For more information, see [Multiplexer](#) in the *ETAssemble Tool Reference manual* or [MUX](#) on the Add Cell Models reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **clock_mux**

Used by LV Flow and Tessent Scan. Specifies a 2-to-1 clock path multiplexer. The cell (model) must have an input with mux_select attribute, an input with mux_in0 attribute, and an input with mux_in1 attribute. For more information, see [Multiplexer](#) in the *ETAssemble Tool Reference manual*.

- **dff**

Used by LV flow ETAssemble and Tessent Scan to describe a cell with only a DFF cell function, with non-inverting data_in to data_out. To be usable, the model must also have an input pin with negedge_clock or posedge_clock attribute, and an input pin with data_in attribute. The cell should have a single output, ideally with data_out attribute, but that pin attribute is optional if the cell has a single output pin. In Tessent Scan, this attribute works in combination with the [Add Observe Points](#) command, the [Add Control Points](#) command or [Set Lockup Cell](#) command. For more information, see DFF on the [Add Cell Models](#) reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **latch**

Used by LV flow ETAssemble and Tessent Scan. Identifies a cell with a latch cell function, with non-inverting data_in to data_out. To be usable, the model must also have an input pin with active_high_clock or active_low_clock attribute, and an input pin with data_in attribute. The cell should have a single output, ideally with data_out attribute, although that pin attribute is optional if the cell has a single output pin. For more information, see LATCH on the [Add Cell Models](#) reference page in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

- **synchronizer_cell**

Used by LV Flow (ETAssemble). Identifies a cell with a clock domain synchronization function (2-bit shift register without parallel access), with noninverting data_in to data_out. To be usable, the model must also have a posedge_clock input pin attribute, a data_in input pin attribute. The cell should have a single output with data_out attribute.

- **scan_cell**

Used by Tessent Shell flows to indicate a mux scan cell. To be usable, the model must also have an input pin with scan_in, an output pin with scan_out or scan_out_inv, and an input pin with scan_enable or scan_enable_inv.

- **pad**

Used to indicate that the cell is a pad. Helpful to prevent insertion of logic outside the IC.

- **prohibited**

Used to indicate that the cell_type cannot be learned and set automatically. It effectively prevents insertion of the model as a cell by test insertion tools.

Note



libcomp ... -PROHIBIT_cell_types causes libcomp to place a “cell_type=prohibited;” statement inside each model written to the output file.

- **simulation_function**

Keyword that identifies the model function as learned by simulation. The statement listing the simulation_function is written automatically by the write_cell_library command. Although simulation_function is parsed the value is discarded and re-learned. Pin functions corresponding to the simulation_function are automatically assigned as well.

The syntax is as follows:

```
simulation_function = {clock_gating_and | clock_gating_or |
    buffer | inverter | or | and | nand | nor | xor | mux |
    dff | latch | synchronizer_cell | tie0 | tie1};
```

Table 3-2 lists all the cell_types, which are described in detail following this table or Table 3-1.

Table 3-2. Simulation_functions

simulation_function	Description
clock_gating_and	See “clock_gating_and”.

Table 3-2. Simulation_functions

simulation_function	Description
clock_gating_or	See “ clock_gating_or ”.
buffer	See “ buffer ”.
inverter	See “ inverter ”.
or	See “ or ”.
and	See “ and ”.
nand	See “ nand ”.
nor	See “ nor ”.
xor	See “ xor ”.
mux	See “ mux ”.
dff	See “ dff ”.
latch	See “ latch ”.
synchronizer_cell	See “ synchronizer_cell ”.
scan_cell	See “ scan_cell ”.
tie0	See “ tie0 ”.
tie1	See “ tie1 ”.

- **tie0**

A single output model whose output value is 0 due to internal ties in the model.

- **tie1**

A single output model whose output value is 1 due to internal ties in the model.

- **pad_ac**

Specifies the AC subclass. This keyword is defined at the model level if it applies to all LV Flow programmable pad modes defined for this model. Otherwise, it must appear only within the *mode* attribute section that it applies to. For more information, see [PadAttribute](#) in the *ETAssemble Tool Reference* manual.

- **pad_ac_lp_time = time_in_seconds**

Specifies the minimum Low pass time constant of an AC input cell. Can be overridden by the mode section. This attribute is only defined here if it applies to all LV Flow programmable pad modes defined for this model that do not specify a different value. Otherwise, it must appear only within the *mode* attribute section that it applies to.

The *time_in_seconds* argument is specified in seconds in exponential real form: 1.0e-7 or 4.73e-08. Either one of or both of the *pad_ac_lp_time* or *pad_ac_hp_time* attributes

must be specified. For more information, see [ACTiming](#) in the *ETAssemble Tool Reference* manual.

- **pad_ac_hp_time**

Specifies the minimum High pass time constant of an AC input cell.

Can be overridden by the mode section. This attribute is only defined here if it applies to all LV Flow programmable pad modes defined for this model that do not specify a different value. Otherwise, it must appear only within the *mode* attribute section that it applies to.

The *time_in_seconds* argument is specified in seconds in exponential real form: 1.0e-7 or 4.73e-08. Either one of or both of the *pad_ac_lp_time* or *pad_ac_hp_time* attributes must be specified. For more information, see [ACTiming](#) in the *ETAssemble Tool Reference* manual.

- **pad_ac_hp_on_chip**

Specifies that the AC High pass filter associated to an AC input pin is on the device rather than on the board. By default, the pin is on the board. Can be overridden by the mode section. This attribute is only defined here if it applies to all LV Flow programmable pad modes defined for this model that do not specify a different value. Otherwise, it must appear only within the *mode* attribute section of the modes that it applies to. For more information, see [ACTiming](#) in the *ETAssemble Tool Reference* manual.

- **bcell_lib_name**

Specifies a pre-existing embedded pad or boundary-scan cell. For more information, see [BlibCell](#) in the *ETAssemble Tool Reference* manual.

- **nonscan_model**

Optional attribute that specifies the non-scan cell model name using one of the following syntaxes:

```
nonscan_model = model_name;
```

```
nonscan_model = model_name (list_of_pins);
```

In the unlikely event that the same scan model replaces multiple nonscan models, they are listed in one comma-separated statement, as shown here:

```
nonscan_model = model_name1, model_name2, model_name3 (list_of_pins);
```

The scan model is typically an extension of the nonscan model with additional test pins. If the pin names of the nonscan model are retained in the scan model, as is typical, only the nonscan model name needs to be specified. For example:

```
non_scan_model = DFF;
```

When some of the nonscan pin names are changed in the scan model, correspondence between the non-scan model and scan model can be determined in two ways, positionally, or by pinname syntax, both using a *list_of_pins*. The *list_of_pins* argument

maps all the pins in the nonscan model to the corresponding pin in the scan model (the pin that has the same pin function). The number of pins in the `list_of_pins` is typically the same as that in the non-scan model. If the number is not the same, you must tie the extra pins either high or low using the `tie1` or `tie0` attributes.

The `list_of_pins` argument can be a complete, comma-separated positional list. In this case, the pins of the non-scan models map by position to named pins of the scan model.

For example, the original model definition of a DFF might list the interface pins (Q, QN, CK, D) while the corresponding scan model SDFF lists the interface pins (CP, D, SI, SE, Q, and QB). The *non_scan_model* attribute establishes pin mapping between the DFF and SDFF by symbolically instantiating the nonscan model using the following syntax:

```
model SDFF (CP, D, SI, SE, Q, QB)
...
non_scan_model = DFF(Q, QB, CP, D);
...
```

Alternately, the *list_of_pins* argument can use a *.pinname* syntax in which all pins are assumed to be the same between the non-scan model and scan model except for the pins explicitly listed in the *list_of_pins* argument using the *.pinname* syntax:

```
(.non-scan_pinname(scan_pinname))
```

For example, if all non-scan pinnames are the same name in the scan cell except that non-scan pin “I” is named “D” in the scan model, the `list_of_pins` argument is `(.I(D))`.

You can map multiple non-scan models to one scan model by listing multiple non-scan models and their pin lists, separated by commas as follows:

```
nonscan_model = model1(in1, in2, in3, out1, out2),
               model2(i1, i2, i3, o1, o2),
               model3(in1, in2, in3, out1);
```

In the case of multiple non-scan models mapping to one scan model, if you rip up existing scan circuitry, Tessent Scan replaces the scan model in the netlist with the first non-scan model defined by the `nonscan_model` attribute.

- **scan_length**

Used by Tessent Scan when balancing scan chains. Optional attribute that only applies to models containing scan cells (models with a *nonscan_model* statement), and which specifies the length of the scan chain in the library model. The integer value must be larger than zero. If this attribute is not specified, the default value for a scan cell model is 1.

A scan cell with a length greater than 1 is referred to as a *multi-bit scan cell*. In a multi-bit scan cell, the number of scan cell ports do not change; there is still a single scan input port, scan output port, and scan enable port. This is because the cascading of the sequential cells are assumed to be done by connecting the scan I/O ports of each sequential cell to form an internal scan chain inside the multi-bit scan cell. The data input and output ports, however, are multiplied by the *scan_length* value.

Tessent Scan does not check the connections of the sequential cells in the multi-bit scan cell. You need to be sure that stitching multi-bit scan cells in scan chains will not cause DRC tracing rule violations.

If the sequential cells within the multi-bit scan cell are already stitched into scan chains, an alternate method for defining internal scan chains is to use the [Add Sub Chains](#) command with the *-library_model* switch. For more information on this method, see the [Scan Insertion, ATPG, and Diagnosis Reference Manual](#).

- **gated_out_scan_cell**
Used by Tessent Scan. Tessent Scan specifies that the system output is gated off inside the scan cell during shift. For more information, see the [Tessent Scan and ATPG User's Manual](#).
- **mode (**
Specifies the beginning of a pad mode or pad usage section that can be used to define one of several different modes for a single pad model, each with its own mode section inside the single model syntax. Each mode section ends with a closing right parentheses. You can have zero or more mode statements. The mode section is only applicable to pad cells.
 - **pad_ac**
Specifies the AC subclass for this mode only.
 - **pad_ac_lp_time**
Specifies for this model only the minimum Low pass time constant of an AC input cell. The *time_in_seconds* argument is specified in time units.
 - **pad_ac_hp_time**
Specifies for this mode only the minimum High pass time constant of an AC input cell. The *time_in_seconds* argument is specified in time units.
 - **pad_ac_hp_on_chip**
Specifies for this mode only that the AC High pass filter associated with an AC input pin is on the device rather than on the board. By default, the tool assumes on the board.
 - **mode input pin attributes**

```
input (pin_name)( // Following are legal mode input pin attributes
[pad_to_pad][pad_pad_io][pad_pad_io_inv][pad_enable_high]
[pad_enable_low][pad_to_sje_mux_low][pad_to_sje_mux_high]
[pad_to_sji_mux] [pad_to_sjo_mux][pad_select_jtag_enable]
[pad_select_jtag_in][pad_select_jtag_out]
[pad_init_data_dot6][pad_init_clock_dot6]
[pad_init_data_inv_dot6][pad_ac_mode_dot6]
[pad_data_inv][pad_diff_voltage][pad_diff_current]
[pad_from_io][pad_from_io_inv][pad_tie0][pad_tie1][pad_open]
) // end input section of mode
```

For information on specific input attributes statements, see “[Defining Pin Attributes](#)” on page 98.

- **mode output pin attributes**

```
output(pin_name)( // Following are legal mode output pin attributes
[pad_from_pad][pad_pad_io][pad_pad_io_inv]
[pad_from_sje_mux][pad_from_sji_mux] [pad_from_sjo_mux]
[pad_sample_pad][pad_data_inv] [pad_diff_voltage][pad_diff_current]
[pad_open_drain][pad_open_source][pad_to_io][pad_to_io_inv]
[pad_pull0][pad_pull1]
) // end output section of mode
```

For information on specific output attributes statements, see “[Defining Pin Attributes](#)” on page 98.

- **mode inout pin attributes**

```
inout(pin_name)(// Following are legal mode inout pin attributes
[pad_pad_io][pad_pad_io_inv][pad_diff_voltage][pad_diff_current]
[pad_open_drain][pad_open_source][pad_pull0][pad_pull1]
) //end inoutsection of mode
```

For information on specific inout attributes statements, see “[Defining Pin Attributes](#)” on page 98.

- **model_source statement**

This statement is optional, but because Libcomp generates it automatically, it often occurs in libraries. This statement is used only for library verification flows invoked with -model to ensure that an appropriate Verilog test harness netlist can be created to instantiate each model for exercising by ATPG and ModelSim simulations.

Because it is illegal to instantiate Verilog primitives by pinname (even UDPs), the test harness instantiates UDPs as positional instances for verification.

If model_source is from a parameter override, the model does not have an explicit Verilog module in the source that matches it. This occurs because these models were uniquified to match the module, as if it were defined with the overridden parameter values, to allow valid tests to be created. Therefore, modules from a parameter override are not instantiated in the Verilog test harness created for verification simulation because no matching Verilog module for ModelSim to compile for simulation exists. Although you cannot test these modules as stand-alone functionality, they are tested in place when the Verilog module with the instance override is tested: that is, both in the ATPG run using the uniquified model to compensate for the fact that it has no override simulation mechanism, and in the Verilog simulation using the Verilog override simulation mechanism.

The Verilog source that created the translated model is identified using exactly one of the following four legal *model_source* statements within any model:

- If the model was originally a User Defined Primitive table, then:

If not blackboxed

```
model_source = verilog_udp;
```

Else

```
model_source = verilog_udp_black_box;
```

- o If the model has one or more ports in the defining portlist that Verilog considers an unnamed port such as “A[0]” or “B[3:0]”, that instance must be instantiated positionally:

If not blackboxed

```
model_source = verilog_unnamed_port_module
```

Else

```
model_source = verilog_unnamed_port_module_black_box
```

- o If the model represents a Verilog instance's implied module created by a parameter override and therefore has no explicit corresponding Verilog module, then:

If not blackboxed

```
model_source = verilog_parameter_override;
```

Else

```
model_source = verilog_parameter_override_black_box;
```

- o If the model is defined as an LV Flow generic, such as LV_MUX, then:

```
model_source = lv_generic;
```

Model Definition Examples

The following are examples that show how to define library cells using correct syntax.

Example 1 — Interface Pins Declarations

For BIBUF, assign A, PI, EN, and TN as input pins; IO as a bidirectional pin; ZI and PO as output pins as follows:

```
model BIBUF(IO, A, EN, TN, PI, ZI, PO) (  
    input (A, PI, EN, TN)...  
    inout (IO)...  
    output (ZI)...  
    output (PO)...  
)
```

Example 2 — Interface Pins Declarations

For SDFF, define TI, TE, D, and CLK as input pins, and Q and QN as output pins as follows:

```
model SDFF(D, CLK, TI, TE, Q, QN) (  
    input (D, TI, TE)...  
    input (CLK)...  
    output (Q, QN)...  
)
```


Example 3 — Scan Cell

The following example shows a model description that uses the *nonscan_model* statement to specify a mux scan cell that can be used to replace a nonscan cell named “dff” by test logic circuitry insertion tools. Note that the clock, data, and test functions are attributed to the ports along with the nonscan cell (model) it can replace to produce scan-testable hardware. These pin-level attributes are discussed in “[Defining Pin Attributes](#).” The hardware section is shown here for completeness, and is discussed in “[Defining Hardware](#).”

```
model sff (D, SI, SE, CLK, Q, QB) (
    cell_type = scan_cell;
    nonscan_model = dff;
    input (D) (data_in)
    input (CLK) (posedge_clock)
    input (SI) (scan_in)
    input (SE) (scan_enable)
    output (QB) (scan_out_inv) // Inverted from scan_in. Connect here ..
    output (Q) (scan_out) // .. or to this noninverting scan_out
    ( // Hardware section
        primitive = _mux (D, SI, SE, _D);
        primitive = _dff (, , CLK, _D, Q, QB);
    )
) //end model
```

Example 4 — Escaping Special Characters

When an model name, net name, or pin name (identifier) contains special characters and needs to be escaped, the Verilog usage is: “\a# ” (without the double quotes -- note that a space is required at the end to designate the end of the name). In the ATPG library model, you can use the Verilog escaped name convention, or you can refer to that identifier without the backslash or final space, but with the enclosing double-quotes.

For example, given the simple Verilog netlist:

```
module test ( i1, o1);
    input i1;
    output o1;
    invlb u1( .\a# (i1), .o (o1) );
endmodule
```

With the Verilog cell library model shown here:

```
module invlb ( \a# , o);
    input \a# ;
    output o;
    not (o, \a# );
endmodule
```

The Tessent cell library model of this Verilog cell can be either of the following:

```
model invlb (\a# , o) ( // Uses Verilog escaped name convention
    input (\a# ) ()
    output (o) ()
    ( // Verilog escaped names start with '\\' and end with ' '
        primitive = _inv (\a# , o); // Note space to end name.
    )
)

model invlb ("a#", o) ( // Uses double quotes convention
    input ("a#") ()
    output (o) ()
    (
        primitive = _inv ("a#", o);
    )
)
```

In another example, given either form of the following netlist fragment:

```
model special_chars_1 // defining model
    (\clk# , d, o)
...

model special_chars_2 // defining model
    ("clk#", d, o)
...
```

If another model instantiates the instance above using named connections, the name containing the special characters should also be in escaped or quoted form, as shown here:

```
instance = special_chars_1 sp_ch_inst_1
    (. \clk# (clk_net), .d(data_net), .o(out_net) );

instance = special_chars_2 sp_ch_inst_2
    (."clk#"(clk_net), .d(data_net), .o(out_net) );
```

Note that only the characters of the name are escaped or quoted. The period which indicates that a defining pinname follows is outside the double quotes because it is not part of the defining pin's name. For more information on specifying the *instance* attribute statement, see the section [“Attribute Descriptions”](#) on page 101.

Defining Hardware

This section describes the hardware definition that can be applied to models and their pins.

[Table 3-3](#) lists the hardware definition that can be applied to models and their pins. Each attribute in the table is described in detail in the “[Hardware Definitions](#)” section following the table.

Table 3-3. Hardware

Attribute Statement	Used for Pin Types	Description
primitive	Intern, Input, Output, Inout	See “ Defining Instance Attributes ”.
instance	Intern, Input, Output, Inout	See “ instance ”.
bus_keeper	Intern, Output, Inout	See “ bus_keeper ”.
function	Intern, Output, Inout	See “ function ”.

Hardware Definitions

This section describes the hardware definitions that can be defined for models and their pins.

Refer to [Table 3-3](#) to determine the pin type the attribute can be attached to.

primitive

The *primitive* attribute statement instantiates predefined primitive elements such as latches and flip-flops, for use in modeling cell functionality for simulation.

Although optional, instance names are recommended to facilitate locating specific gates and instances in reports and visual displays.

Refer to “[Supported Primitives](#)” on page 131 for all tool-supported primitives.

The format and syntax for the *primitive* attribute statement is as follows:

```
primitive = _primitive_name [instance_name] (<list_of_nets>);
```

The following is an example of the usage of a *primitive* attribute statement:

```
model and2(out, in1, in2) (  
    input (in1, in2) (  
        output (out) (  
            (  
                primitive = _and and1x (in1, in2, out);  
            )  
        )  
    )  
)
```

Here is an example of the *primitive* attribute statement with explicit *intern* net declarations. Note, these are only required if attributes, such as “array = ” to declare a vector, need to be specified for an internal net.

```
model andnor1(A1, A2, B1, B2, ZN) (  
    input(A1, A2, B1, B2) (  
        intern(N1, N2) (  
            output(ZN) (  
                (  
                    primitive = _and an1 (A1, A2, N1);  
                    primitive = _and an2 (B1, B2, N2);  
                    primitive = _nor nr1 (N1, N2, ZN);  
                )  
            )  
        )  
    )  
)
```

This example which has only scalar nets inside it could more simply be described without *intern* statements as follows:

```
model andnor1(A1, A2, B1, B2, ZN) (  
    input(A1, A2, B1, B2) (  
        output(ZN) (  
            (  
                primitive = _and an1 (A1, A2, N1);  
                primitive = _and an2 (B1, B2, N2);  
                primitive = _nor nr1 (N1, N2, ZN);  
            )  
        )  
    )  
)
```

A primitive attribute statement cannot have an `instance_name` if there is a [function](#) statement in the library model. You should consider avoiding the function statement so that instantiations of primitives and other models inside the model definition can be named. If more than one primitive statement exists in a library model, you must specify instance names for either all or for none of the primitive statements.

instance

The *instance* attribute statement refers to another defined library model.

The format and syntax for the *instance* attribute statement is as follows:

```
instance = model_name [instance_name] (<list_of_nets>);
```

The *model_name* refers to another model name defined in the library. The *list_of_nets* refers to the boundary pins of the *model_name* or internal nets. The entire *list_of_nets* must be specified as either a positional or pinname connection; a pinname connection does not rely on position as shown here:

```
instance = model_name (.pinX(net1), .pinY(net2), ..., .pinZ(netN));
```

Instance pin connections are the default when LibComp translates Verilog modules. See the [Set Instance Portlist](#) command or information on how to obtain instance positional connections for backward compatibility with v8.2009_1 and earlier versions of the ATPG library parser.

Here is an example using the *instance* attribute statement:

```
model andnor2(A1, A2, A3, B1, B2, B3, ZN) (  
    input(A1, A2, A3, B1, B2, B3) ()  
    output(ZN) ()  
    (  
        instance = and3 (.A1(A1), .A2(A2), .A3(A3), .Z(N1));  
        instance = and3 (.A1(B1), .A2(B2), .A3(B3), .Z(N2));  
        instance = nor2 (.A1(N1), .A2(N2), .ZN(ZN));  
    )  
)  
  
model and3(A1, A2, A3, Z) (  
    input(A1, A2, A3) ()  
    output(Z) ()  
    (  
        primitive = _and (A1, A2, A3, Z);  
    )  
)  
  
model nor2(A1, A2, ZN) (  
    input(A1, A2) ()  
    output(ZN) ()  
    (  
        primitive = _nor (A1, A2, ZN);  
    )  
)
```

The *instance_name* argument is an optional user-defined name. It is recommended for reference purposes. Note that an instance attribute statement cannot have an instance name if there is a [function](#) statement described in the library model. If there is more than one instance attribute statement in a library model, either instance names must be given for all instance attribute statements, or no instance names can be given in any (within that same model). This also applies if there are primitive and instance attribute statements in the same library model (all must have instance names in their statements or none). In the example below, the top model has instance names and the lower two models do not. This example is meant to illustrate the rule; typically, the lower models would also be given instance names.

Here is an example of the *instance* attribute statement with instance positional connections:

```
model andnor2(A1, A2, A3, B1, B2, B3, ZN) (  
    input(A1, A2, A3, B1, B2, B3) ()  
    output(ZN) ()  
    (  
        instance = and3 (.A1(A1), .A2(A2), .A3(A3), .Z(N1));  
        instance = and3 (.A1(B1), .A2(B2), .A3(B3), .Z(N2));  
        instance = nor2 (.A1(N1), .A2(N2), .ZN(ZN));  
    )  
)
```

```
model and3 (A1, A2, A3, Z) (
    input (A1, A2, A3) ( )
    output (Z) ( )
    (
        primitive = _and (A1, A2, A3, Z);
    )
)

model nor2 (A1, A2, ZN) (
    input (A1, A2) ( )
    output (ZN) ( )
    (
        primitive = _nor (A1, A2, ZN);
    )
)

model mixed_insts (in1, in2, in3, and_out, nand_out) (
    input (in1, in2, in3) ( )
    output (and_out, nand_out) ( )
    (
        instance = and3 and_inst (in1, in2, in3, and_out);
        primitive = _inv inv_inst (and_out, nand_out);
    )
)
```

bus_keeper

The `bus_keeper` attribute statement models the ability of a net or pin to retain its previous binary state when it is not driven.

The format of this attribute statement is:

```
output|inout|intern (<name>) (
    bus_keeper = <zhold | zhold0 | zhold1>;
)

// For example, to define a cell/model output that holds old values when
// not driven, that pin is attributed with the bus_keeper's zhold value

output (hold_out) (bus_keeper = zhold;)
```

`zhold` retains the previous binary state, `zhold0` retains only a preceding 0 state, and `zhold1` retains only a preceding 1 state. If the input value (driver) of the bus is not Z, the output value is the same as the input value. In other words, the driven output or internal net simply receives the value of the driver, as if no `bus_keeper` were present.

If the input value is Z (all drivers are high impedance), the following occurs:

- If the previous value is retained by the `bus_keeper`'s hold type, the output value is set to the previous value (the output or net retains its old value).

- If the previous value is not retained due to the bus_keeper's hold type (such as a previous value of 1 for a zhold0), the output is set to Z (the output or net becomes Z like the driver).
- If the previous value is X, the output value is set to X.

If multiple bus_keeper attributes are used on a net, their effect is additive (zhold0 plus zhold1 acts like a zhold, holding either previous value). If a non-tristate net is assigned a bus_keeper attribute, a warning message is issued. For information on bus keeper analysis during rules checking, refer to “[Bus Keeper Analysis](#)” in the *Tessent Scan and ATPG User's Manual*.

function

The *function* attribute describes the function of the model's output pin's internal nodes in terms of the model's input pins, output pins, bidirectional pins, and other internal nodes.

There can be only one function per output, inout, or intern because syntactically that is the only way to connect the output of a function. Note, using this statement prevents you from using names and can have a negative impact as compared to using named primitives and instances when reporting messages pertinent to a specific internal instance. You define the function of internal nodes by using legal operators in a Boolean expression as follows:

```
output|inout|intern (<name>) (
    function = boolean_expression;
)
```

The legal Boolean operators are:

- ! invert following expression
- * logical AND operation
- + logical OR operation

Here is an example of legal Boolean operators. In this example, the internal node is ETN and the function of ETN is “!(TN*!EN)”. The function of the output ZI is defined with the function attribute statement “function = IO”. When the library model is compiled, a combinational buffer is created:

```
model BD4T(IO, A, EN, TN, PI, ZI, PO) (
    input(A, PI, EN, TN) ()
    output(ZI) (function = IO;)
    output(PO) (function = !(ZI * PI);)
    intern(ETN) (function = !(TN * !EN);)
    inout(IO) ( )
    (
        primitive = _tsl (A, ETN, IO);
    )
)
```

The following example is the same model using no function statements. You could easily add instance names before the left paren of each primitive statement:

```
model BD4T(IO, A, EN, TN, PI, ZI, PO) (
    input(A, PI, EN, TN) ( )
    output(ZI) ( )
    output(PO) ( )
    intern(ETN) ( )
    inout(IO) ( )
    (
        primitive = _buf (ZI, IO);
        primitive = _nand (ZI, PI, PO);
        primitive = _inv (EN, notEN);
        primitive = _nand (TN, notEN, ETN);
        primitive = _tsl (A, ETN, IO);
    )
)
```

Defining Instance Attributes

This section describes the attributes that can be applied to models and their pins.

[Table 3-4](#) lists, in alphabetical order, the attributes that can be applied to models and their pins. Each attribute is described in detail in section “[Instance Attributes](#)” following the table.

Table 3-4. Instance Attributes

Attribute Statement ¹	Used for Pin Types ²	Description
reset_clock_conflict	Intern, Output, Inout	See “ reset_clock_conflict ”.
set_clock_conflict	Intern, Output, Inout	See “ set_clock_conflict ”.

1. If no attribute is defined for pin groups, () must be entered after the pin field to indicate the attribute field in the model.
2. The *input* attributes are optional.

Instance Attributes

The following sections describe the attribute statements that can be defined for instances and their pins.

Refer to [Table 3-4](#) to determine the pin type the attribute can be attached to.

reset_clock_conflict

This attribute only applies to single ported D flip-flops and **only impacts FlexTest simulations**.

For sequential primitive D flip-flops that contain a single reset pin, the values of Q and Qbar become unknown if the input data is 1 when both the reset and clock pins are active at the same time. This attribute statement allows users to force the values of Q and Qbar to 0 and 1 respectively, when unknown values occur as shown here:


```
reset_clock_conflict = q_qbar_value;
```

The possible values of `q_qbar_value` are XX and 01 (reset-dominated clock). In Tessent FastScan and Tessent TestKompress, XX is used. In FlexTest, if this attribute is not used, the default value is 01.

The `reset_clock_conflict` statement should be placed before the sequential primitive statement and can only affect single data-clock port sequential primitives. Also, for each sequential primitive, there can only be one conflict attribute given. This attribute has no effect in Tessent Scan, Tessent FastScan, or Tessent TestKompress.

For example:

```
model FD2S(D, CP, CD, TI, TE, Q, QN) (
  input(D, TI, TE) ()
  intern(ND) (function = D * !TE + TI * TE;)
  input(CP) (posedge_clock;)
  input(CD) ()
  output(Q,QN) ()
  (
    reset_clock_conflict = 01;
    primitive = _dff (, CD, CP, ND, Q, QN);
  )
)
```

set_clock_conflict

This attribute only applies to single ported D flip-flops and **only impacts FlexTest simulations**.

For sequential primitive D flip-flops that contain a single set pin, the values of Q and Qbar become unknown if the input data is 0 when both the set and clock pins are active at the same time. This attribute statement allows users to force the values of Q and Qbar to 1 and 0 respectively, when the situation of unknown values occurs. To accomplish this, the following statement is provided:

```
set_clock_conflict = q_qbar_value;
```

The possible values of `q_qbar_value` are XX and 10 (set-dominated clock). In Tessent FastScan and Tessent TestKompress, XX is used. In FlexTest, if this attribute is not used, the default value is 10.

The `set_clock_conflict` statement should be placed before the sequential primitive statement, and can only affect single data-clock port sequential primitives. Also, for each sequential primitive, there can only be one conflict attribute given. This attribute has no effect in Tessent Scan, Tessent FastScan, or Tessent TestKompress.

Note



The `set_clock_conflict` and the `reset_clock_conflict` attributes only apply to single port D flip-flops and only impact FlexTest simulations.

Defining Pin Attributes

The final step in defining pin attributes is to create internal connectivities in the model by assigning attributes to the interface pins and internal nodes.

The interface pins and internal nodes are connected to individual elements such as combinational or sequential elements. You may use Boolean expressions to create combinational elements, or primitive attributes to build sequential elements. Additional attribute statements allow you to further define the internal structure of the model precisely as shown here.

```
model model_name (list_of_pins) (
    intern (intern_nodes) (intern_attributes)
    input (input_pins) (input attributes)
    inout (inout_pins) (inout attributes)
    output (output_pins) (output attributes)
```

[Table 3-5](#) lists, in alphabetical order, the attributes that can be applied to models and their pins. Each attribute in the table is described in detail in section [“Attribute Descriptions”](#) which follows this table.

Table 3-5. Pin Attributes

Attribute	For Pin Direction	Description
Function		
active_high_clock	Input	See “active_high_clock” .
active_high_reset	Input	See “active_high_reset” .
active_high_set	Input	See “active_high_set” .
active_low_clock	Input	See “active_low_clock” .
active_low_reset	Input	See “active_low_reset” .
active_low_set	Input	See “active_low_set” .
asynch_enable	Input	See “asynch_enable” .
asynch_enable_inv	Input	See “asynch_enable_inv” .
asynch_disable	Input	See “asynch_disable” .
asynch_disable_inv	Input	See “asynch_disable_inv” .
clock_in	Input	See “clock_in” .
clock_out	Output	See “clock_out” .
data_in	Input	See “data_in” .
data_out	Output	See “data_out” .
data_out_inv	Output	See “data_out” .
func_enable	Input	See “func_enable” .

Table 3-5. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
func_enable_inv	Input	See “ func_enable_inv ”.
mux_in0	Input	See “ mux_in0 ”.
mux_in1	Input	See “ mux_in1 ”.
mux_in2	Input	See “ mux_in2 ”.
mux_in3	Input	See “ mux_in3 ”.
mux_in4	Input	See “ mux_in4 ”.
mux_in5	Input	See “ mux_in5 ”.
mux_in6	Input	See “ mux_in6 ”.
mux_in7	Input	See “ mux_in7 ”.
mux_out	Output	See “ mux_out ”.
mux_select	Input	See “ mux_select ”.
mux_select0	Input	See “ mux_select0 ”.
mux_select1	Input	See “ mux_select1 ”.
mux_select2	Input	See “ mux_select2 ”.
negedge_clock	Input	See “ negedge_clock ”.
posedge_clock	Input	See “ posedge_clock ”.
power_isolation_internal	Input, Output	See “ power_isolation_internal ”.
power_isolation_external	Input, Output	See “ power_isolation_external ”.
scan_enable	Input	See “ scan_enable ”.
scan_enable_inv	Input	See “ scan_enable_inv ”.
scan_in	Input	See “ scan_in ”.
scan_out	Output	See “ scan_out ”.
scan_out_inv	Output	See “ scan_out_inv ”.
test_enable	Input	See “ test_enable ”.
test_enable_inv	Input	See “ test_enable_inv ”.
tie0	Input	See “ tie0 ”.
tie1	Input	See “ tie1 ”.
Pad Function		
pad_ac_mode_dot6	Input	See “ pad_ac_mode_dot6 ”.
pad_data_inv	Input, Output	See “ pad_data_inv ”.

Table 3-5. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
pad_diff_current	Input, Output, Inout	See “ pad_diff_current ”.
pad_diff_voltage	Input, Output, Inout	See “ pad_diff_voltage ”.
pad_enable_high	Input	See “ pad_enable_high ”.
pad_enable_low	Input	See “ pad_enable_low ”.
pad_force_disable	Input	See “ pad_force_disable ”.
pad_from_io	Input	See “ pad_from_io ”.
pad_from_io_inv	Input	See “ pad_from_io_inv ”.
pad_from_pad	Output	See “ pad_from_pad ”.
pad_from_sje_mux	Output	See “ pad_from_sje_mux ”.
pad_from_sji_mux	Output	See “ pad_from_sji_mux ”.
pad_from_sjo_mux	Output	See “ pad_from_sjo_mux ”.
pad_init_clock_dot6	Input	See “ pad_init_clock_dot6 ”.
pad_init_data_dot6	Input	See “ pad_init_data_dot6 ”.
pad_init_data_inv_dot6	Input	See “ pad_init_data_inv_dot6 ”.
pad_keep_tracing	Input, Output	See “ pad_keep_tracing ”.
pad_pad_io	Input, Output, Inout	See “ pad_pad_io ”.
pad_pad_io_inv	Input, Output, Inout	See “ pad_pad_io_inv ”.
pad_sample_pad	Output	See “ pad_sample_pad ”.
pad_select_jtag_enable	Input	See “ pad_select_jtag_enable ”.
pad_select_jtag_in	Input	See “ pad_select_jtag_in ”.
pad_select_jtag_out	Input	See “ pad_select_jtag_out ”.
pad_test_data_dot6	Output	See “ pad_test_data_dot6 ”.
pad_test_data_inv_dot6	Output	See “ pad_test_data_inv_dot6 ”.
pad_to_io	Input	See “ pad_to_io ”.
pad_to_io_inv	Input	See “ pad_to_io_inv ”.
pad_to_pad	Input	See “ pad_to_pad ”.
pad_to_sje_mux_low	Input	See “ pad_to_sje_mux_low ”.
pad_to_sje_mux_high	Input	See “ pad_to_sje_mux_high ”.
pad_to_sji_mux	Input	See “ pad_to_sji_mux ”.
pad_to_sjo_mux	Input	See “ pad_to_sjo_mux ”.

Table 3-5. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
Special Drive		
pad_open_drain	Output, Inout	See “ pad_open_drain ”.
pad_open_source	Output, Inout	See “ pad_open_source ”.
pad_pull0	Input, Output, Inout	See “ pad_pull0 ”.
pad_pull1	Input, Output, Inout	See “ pad_pull1 ”.
Other		
max_fanout	Output	See “ max_fanout ”.
no-fault (instance)	Input, Intern, Output, Inout	See “ no-fault (instance/primitive pins) ”.
no-fault (model pins)	Input, Intern, Output, Inout	See “ no-fault (model pins) ”.
open	Input, Output, Inout	See “ open ”.
pad_open	Input	See “ pad_open ”.
pad_tied0	Input	See “ pad_tied0 ”.
pad_tied1	Input	See “ pad_tied1 ”.
unused	Intern, Input, Output, Inout	See “ unused ”.

Attribute Descriptions

The following sections, presented in alphabetical order, describe the attribute statements that can be defined for models and their pins.

Refer to [Table 3-5](#) on page 98 to determine the pin type the attribute can be attached to.

- **active_high_clock**

Level clock and polarity.

- **active_high_reset**

Resets pin and polarity. Pin is active when equal to 1.

- **active_high_set**

Sets pin and polarity. Pin is active when equal to 1.

- **active_low_clock**

Level clock and polarity.

- **active_low_reset**

Resets pin and polarity. Pin is active when equal to 0.

- **active_low_set**

Sets pin and polarity. Pin is active when equal to 0.

- **asynch_enable**

Optional attribute used for ClockGating cells to define the asynchronous or immediate enable.

- **asynch_enable_inv**

Optional attribute used for ClockGating cells to define the asynchronous or immediate enable.

- **asynch_disable**

Optional attribute used for ClockGating cells to define the asynchronous or immediate disable.

- **asynch_disable_inv**

Optional attribute used for ClockGating cells to define the asynchronous or immediate disable.

- **clock_in**

Required attribute used for ClockGating cells to define the clock pin of the cell. Also optional attribute to indicate the clock pin of a clock_and or a clock_or cell. If specified, the test insertion tools will connect the clock to the clock_in pin and the enable to the data_in pin.

- **clock_out**

Used to indicate gated clock output of ClockGating cells.

- **data_in**

Optional attribute that specifies the name of the data input pin of several cell types, including the scan_cell cells.

- **data_out**

Optional for single output cell. If specified, indicates the path from data_in to this pin is non-inverting.

- **data_out_inv**

Optional for single output cell. If specified, indicates the path from data_in to this pin is non-inverting.

- **func_enable**
Optional attribute used for ClockGating cells to define the functional or system enable.
- **func_enable_inv**
Optional attribute used for ClockGating cells to define the functional or system enable.
- **max_fanout**
For Tessent Scan, Add Cell Models Buf –max_fanout.
`max_fanout = integer_greater_than_1`
- **mux_in0**
Used for cell_type *mux*. Indicates that the pin is selected when the *mux_select* pin is 0.
- **mux_in1**
Used for cell_type *mux*. Indicates that the pin is selected when the *mux_select* pin is 1.
- **mux_in2**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 2.
- **mux_in3**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 3.
- **mux_in4**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 4.
- **mux_in5**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 5.
- **mux_in6**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 6.
- **mux_in7**
Used for cell_type *mux*. Indicates that the pin is selected when the mux select pins have the value 7.
- **mux_out**
Optional for mux cell. If specified, indicates a non-inverting mux output.
- **scan_enable**

Specifies the name of the scan enable pin associated with the mux-scan cell, and the non-inverting polarity. When this pin is 1, the cell is in scan (shift) mode and captures the scan_in value. When the pin is 0, the cell is in system mode and captures the data_in value.

- **scan_enable_inv**

Specifies the name of the scan enable pin associated with the mux-scan cell. When this pin is 0, the cell is in scan (shift) mode and captures the scan_in value. When the pin is 1, the cell is in system mode and captures the data_in value.

- **mux_select**

Used for cell_type *mux*. Determines whether cell output = mux_in0 or cell output = mux_in1.

- **mux_select0**

Used for cell_type *mux* when it has more than two data inputs. It indicates the least significant select input.

- **mux_select1**

Used for cell_type *mux* when it has more than two data inputs. It indicates the next to least significant select input.

- **mux_select2**

Used for cell_type *mux* when it has five or more data inputs. It indicates the most significant select input.

- **negedge_clock**

Edge-triggered clock that changes state on falling edge. A pin with a negedge_clock or a posedge_clock attribute is required by many cell types such as dff, synchronizer_cell, scan_cell etc.

- **no-fault (instance/primitive pins)**

Typically, users use the tool default of faulting library cell boundaries for determining fault sites. The *no-fault* attribute only applies in the rare case where you want to explicitly determine fault sites inside library cells; these are referred to as internal faults. If the instance and primitive attribute statements have instance names, they can be faulted or not faulted by the fault attribute statement. Assume that somehow the instance and primitive attribute statements are faulted at the boundary and you want to not fault certain instance pins or primitive pins.

For library instances, no-faults on pins can be controlled by their defining library model. The side effect is that no-fault on model pins will also affect other library instances which are instantiated from the model, as in a hierarchical library description. The syntax is as follows:


```
node_name : <nf0 | nf1 | nf>
```

Here, *node_name* can be model pin names or internal node names which appear in the instance or primitive attribute statements. The keywords *nf0*, *nf1*, and *nf* stand for no-fault at 0, no-fault at 1, and no-fault at 0 and 1, respectively.

Here is an example with the *nf* no-fault attribute statement within instance attribute statements:

```
model AO2(A, B, C, D, Z) (
    input (A, B, C, D)    ()
    output(Z)            ()
    (
        instance = AN2 U2(C, D:nf1, CD);
        instance = NR2 U3(AB, CD, Z:nf);
        instance=AN2 U1(A:nf0, B, AB);
    )
)
```

- **no-fault (model pins)**

Each pin can have the characteristic of stuck-at-1 and stuck-at-0 faults. This attribute allows you to exclude any stuck-at fault at specified pins.

The syntax is as follows:

```
no_fault = sa0      \\ Specifies that no stuck-at-0 fault is
                    \\ considered at the specified pin. Only
                    \\ stuck-at-1 will be considered.
no_fault = sa1      \\ Specifies that no stuck-at-1 fault is
                    \\ considered at the specified pin. Only
                    \\ stuck-at-0 faults will be considered.
no_fault = sa0 sa1  \\ Specifies that no stuck-at-0 and
                    \\ stuck-at-1 faults are considered at the
                    \\ specified pin.
```

An example of the *no-fault* attribute statement is as follows:

```
model FD2(D, CP, CD, Q, QN) (
    input (D) (no_fault = sa0;)
    input (CP) (posedge_clock;)
    input (CD) ()
    ... )
```

This is the same example as the previous one using the *no-fault* attribute statement for instance/primitive pins. In this example, the syntax is exactly analogous to that described for instance pins for the [no-fault \(instance/primitive pins\)](#) attribute, but is used in the model portlist rather than the instance portlist.

```
model FD2(D:nf0, CP, CD, Q, QN) (
    input (D) ()
    input (CP) (posedge_clock;)
    input (CD) ()
    ...
)
```

- **open**
Specifies a pin that should be left unconnected when the cell is inserted as test logic.
- **pad_ac_mode_dot6**
When present, specifies that ETAssemble adds the ACM subclass to the pad. This mode is optional. For details, refer to the section “Pad Library File” in *Application Note Support for IEEE 1149.6 Boundary Scan*.
- **pad_data_inv**
Specifies that data on this pin is inverted. This attribute is applicable only for cell pins with either *fromPad* or *toPad* functions.
- **pad_diff_current**
Specifies that the pad is a differential voltage or differential current type cell. Used for differential.
- **pad_diff_voltage**
Specifies that the pad is a differential voltage or differential current type cell. Used for differential.
- **pad_enable_high**
Specifies the active high enable pins for bidirectional or output pads. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual.
- **pad_enable_low**
Specifies the active high enable pins for bidirectional or output pads. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual.
- **pad_force_disable**
Used for cell_type pad.
- **pad_from_io**
Specifies the input connection from an input or inout top pin to its pad. If the pad description has this cell pin function but neither toIO nor padIO cell pin function, then the BSDL top pin direction will be forced to input even if the direction of the associated top pin is inout.
- **pad_from_io_inv**
Specifies the input connection from the second top pin of a differential pin pair which direction is input or inout to its (differential) pad.
- **pad_from_pad**
Used for cell_type pad.

- **pad_from_sje_mux**
Used for cell_type MUX. Select Jtag Enable.
- **pad_from_sji_mux**
Used for cell_type pad. Select Jtag Input.
- **pad_from_sjo_mux**
Used for cell_type pad. Select Jtag Output.
- **pad_init_clock_dot6**
Required for all input and bidirectional AC pads. For details, refer to the section “Pad Library File” in *Application Note Support for IEEE 1149.6 Boundary Scan*.
- **pad_init_data_dot6**
Required for all input and bidirectional AC pads. For details, refer to the section “Pad Library File” in *Application Note Support for IEEE 1149.6 Boundary Scan*.
- **pad_init_data_inv_dot6**
Required for differential input and bidirectional AC pads. For details, refer to the section “Pad Library File” in *Application Note Support for IEEE 1149.6 Boundary Scan*.
- **pad_keep_tracing**
Used for cell_type pad to allow more flexible logic around IO pads. Guides the trace when logic intervenes between sites where an attribute might typically be expected to the downstream site where it occurs.
- **pad_open**
Specifies the condition of a pin should be open for the pad to operate in a given mode (Usage).
- **pad_open_drain**
Used for cell_type pad.
- **pad_open_source**
Used for cell_type pad.
- **pad_pad_io**
Specifies the input connection from an input or inout top pin to its pad. If the pad description has this cell pin function but neither toIO nor padIO cell pin function, then the BSDL top pin direction will be forced to input even if the direction of the associated top pin is inout.
- **pad_pad_io_inv**

Specifies the input connection from the second top pin of a differential pin pair which direction is input or inout to its (differential) pad.

- **pad_pull0**

Specifies that a tri-state bidirectional or output pad is pulled to ground using a resistor internal to the device. The effect of specifying a *pull0* attribute on a pad is two-fold. In the BSDL file generated by ETAssemble, the **DisableResult** value corresponding to the pin associated with this pad is *pull0*. During the TAP simulation, the *Output* test algorithm compares the output of the pin with a strong logic 0.

- **pad_pull1**

Specifies that a tri-state bidirectional or output pad is pulled to VDD using a resistor internal to the device. The effect of specifying a *pull1* attribute on a pad is two-fold. In the BSDL file generated by ETAssemble, the **DisableResult** value corresponding to the pin associated with this pad is *pull1*. During the TAP simulation, the *Output* test algorithm compares the output of the pin with a strong logic 1.

- **pad_sample_pad**

Specifies that the pin is a test-only buffered copy of the fromPad output pin. For more information, see description of the samplePad [Function](#) in the *ETAssemble Tool Reference* manual.

- **pad_select_jtag_enable**

Specifies the connection for the SJEMux control signal, usually connected directly to the selectJtagOutput pin of the TAP, except when the pin is specified as an auxiliary output pin. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. SJE mux select input.

- **pad_select_jtag_in**

Specifies the connection for the control signal, usually coming from the TAP controller, which controls the updating of the boundary-scan cell's internal register. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. SJI/SJO selects.

- **pad_select_jtag_out**

Specifies the connection for the control signal, usually coming from the TAP controller which controls the updating of the boundary-scan cell's internal register. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. SJI/SJO selects.

- **pad_test_data_dot6**

Used for pad for 1149.6 AC test.

- **pad_test_data_inv_dot6**

Used for pad for 1149.6 AC test.

- **pad_tied0**
Specifies the condition of a pin should be tied to 0 (LV tiedLow) for the pad to operate in a given mode (LV Usage).
- **pad_tied1**
Specifies the condition of a pin should be tied to 1 (LV tiedHigh) for the pad to operate in a given mode (LV Usage).
- **pad_to_io**
Specifies the output connection from an output or inout top pin to its pad. If the pad description has this cell pin function but neither fromIO nor padIO cell pin function, then the BSDL top pin direction will be forced to output even if the direction of the associated top pin is inout.
- **pad_to_io_inv**
Specifies the output connection from the second top pin of a differential pin pair whose direction is output or inout to its (differential) pad.
- **pad_to_pad**
Specifies the input/output connections from/to a pad and the core module. For more information, see [Attribute](#) in the *ETAssemble Tool Reference* manual.
- **pad_to_sje_mux_low**
Specifies the test enable signal from the boundary-scan EN cell. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. Select Jtag Enable.
- **pad_to_sje_mux_high**
Specifies the test enable signal from the boundary-scan EN cell. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. Select Jtag Enable.
- **pad_to_sji_mux**
Specifies the output/input on the pad cells that are connected to the Select Jtag Input (SJI) or output (SJO) multiplexer. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. Select Jtag Input.
- **pad_to_sjo_mux**
Specifies the output/input on the pad cells that are connected to the Select Jtag Input (SJI) or output (SJO) multiplexer. For more information, see [Function](#) in the *ETAssemble Tool Reference* manual. Select Jtag Output.
- **posedge_clock**
Edge-triggered clock and polarity.
- **power_isolation_internal**

Identifies the pin on the Power Isolation cell that is connected to the internal logic of the ELTCORE. For more information, see [PowerIsoInt](#) in the *ETAssemble Tool Reference* manual.

- **power_isolation_external**

Identifies the pin on the Power Isolation cell that is connected to the primary input or primary output port of the ELTCORE. For more information, see [PowerIsoExt](#) in the *ETAssemble Tool Reference* manual.

- **scan_in**

Required attribute for a scan cell that specifies the name of the scan input pin of the scan cell. Multiple scan_in pins are not supported.

- **scan_out**

Required attribute for a scan cell that specifies the pin is the shift scan out pin. The path from scan_in to this scan_out pin is a non-inverting path. See Note below.

- **scan_out_inv**

Required attribute for a scan cell that specifies the pin is the shift scan out pin. The path from scan_in to this scan_out pin is an inverting path. See Note below.

Note



You must specify at least one of scan_out or scan_out_inv for a scan cell. Both are allowed if the cell has both inverting and non_inverting outputs.

- **test_enable**

For the LV Flow, specifies the test enable pin on the clock gating cell.

For classic Mentor tools, specifies a new test enable pin in the scan model that does not exist in the non-scan model. In this case, the test enable is used as a selector to choose either original clock, set and reset; or test clock, test set and test reset.

- **test_enable_inv**

For the LV Flow only. Specifies the test enable pin on the clock gating cell. This is an optional attribute.

- **tie0**

Specifies to tie off the pin to this value when inserting cell as test logic. Used when a scan model has more pins, such as extra set or reset lines, than the non-scan model to which it maps. This attribute specifies that the extra pin should be tied low after the scan is inserted.

- **tie1**

Specifies to tie off the pin to this value when inserting cell as test logic. Used when a scan model has more pins, such as extra set or reset lines, than the non-scan model to which it maps. This attribute specifies that the extra pin should be tied high after the scan is inserted.

- **unused**

Specifies that a pin is not connected internally. By default, a message is issued if an input pin or intern is unused. You can use the unused attribute to suppress the warning message for a particular pin or intern. The syntax is as follows:

```
unused
```

Here is an example of an unused pin, where the warning message will be suppressed:

```
model dummy(IN1, IN2, OUT) (  
    input (IN1) (  
        input (IN2) (unused)  
        output (OUT) ( )  
        (  
            primitive = _buf (IN1, OUT);  
        )  
    )  
)
```

Primitive and Attribute Examples

The following are examples of primitives and the attributes used to describe them.

Example 1

In this example, the non-scan model for the original unattributed *dff* model is shown followed by its scan equivalent *sff*.

```
model dff (D, CLK, Q, QB) (  
    input (D, CLK) (  
        output(Q, QB) (  
            (  
                // Note that asynch set and reset nets are each left out  
                // via ' , ' indicating the _dff below has no such pins.  
                primitive = _dff( , , CLK, D, Q, QB);  
            )  
        )  
    )  
)  
  
model sff (D, SI, SE, CLK, Q, QB) (  
    input (D, SI, SE, CLK) ( )  
    output(Q, QB) (  
        (  
            primitive = _mux n2 (D, SI, SE, mux_D_net);  
            instance = dff n3 (mux_D_net, CLK, Q, QB);  
        )  
    )  
)
```

)

To indicate pin and cell attributes for the scan cell, you use one or more of the three methods described here:

- **Method 1**— Edit to populate with attributes and replace directly.

```
model sff (D, SI, SE, CLK, Q, QB) (
    nonscan_model = dff;
    cell_type = scan_cell;
    input (D) (data_in)
    input (SI) (scan_in)
    input (SE) (scan_enable)
    input (CLK) (posedge_clock)
    output (Q) (scan_out)
    output (QB) (scan_out_inv)
    (
        primitive = _mux n2 (D, SI, SE, mux_D_net);
        primitive = _dff n3 ( , , CLK, mux_D_net Q, QB);
    )
)
```

- **Method 2** — Define the attributes using the *cell_attributes* syntax. This allows you to use regular expressions to define attributes on multiple cells and pins in a single expression. In this example, a separate file *adk.attributes* is used whose contents are shown below.

When the file is processed, any model whose name starts with “sff” is assigned the nonscan_model “dff” due to the first model level statement (attribute). Any pins in a matching model whose names match one of the names below (which are also processed as regular expressions) are given the attribute indicated. For example, an input name “SE” is attributed as scan_enable. If “SE” were replaced by “SE.*” below, any pin starting with “SE” would be attributed as scan_enable. (There should be only one such pin on a model but other models may use SEN, SENABLE, and so on and still be attributed if the regexp is used). Model and pin names are case sensitive, so their regular expressions are also case sensitive.

```
cell_attributes (
    regexp = anchored;
    model ("sff.*") (
        nonscan_model = dff;
        cell_type = scan_cell;
        input (D) (data_in)
        input (SI) (scan_in)
        input (SE) (scan_enable) // Can also have regexp pinname, e.g.
                                // "SE.*" for all pins that start with "SE"
        output (Q) (scan_out)
        output (QB) (scan_out_inv)
    )
)
```

The following message, assuming *regexp_verbosity* is not set to *silent*, indicates a model match and shows the anchored string created from the quoted string above that was used

by *regexp*. This makes it easier to find that unintended models have been attributed, and if so, the regular expression can be adjusted. Refer to the Unix man page using the **man 7 regexp** shell command for a complete definition of the regular expression syntax supported.

```
// Full match of model_attribute ( model '^sff.*$' on line 3
// of file 'data/./adk.attributes', with model 'sff' on line 520
// of file 'data/./adk.atpg'.
```

This method also allows the use of simple (non-regexp) *cell_attributes* expressions. It is the same as just illustrated except that *regexp* is set to *off*, and simple names must always be used (not regular expressions) for both model and pin names. The *regexp = off* statement must precede the model statement and applies to every model following it within the same *cell_attributes* section or until another *regexp* statement is encountered. Note that regular expressions are not legal library model or pin names and so must be quoted. However, when *regexp* is off, you can use simple (unescaped) model and pin names directly without quotes.

```
cell_attributes (
  regexp = off;
  model (sff) (
    nonscan_model = dff;
    cell_type = scan_cell;
    input (D) (data_in)
    input (SI) (scan_in)
    input (SE) (scan_enable)
    output(Q) (scan_out;)
    output(QB) (scan_out_inv;)
  )
)
```

- **Method 3**— This method uses pre-existing *cell.lib*, *scang.lib* and *pad.lib* libraries. These can be in the same file or specified in a different file using the *-lib* switch. The library parser recognizes these by their syntactic wrappers as shown here:

```
cellLibrary (<libname>) {...} // cell.lib syntax expected inside {...}
library (<libname>) {...} // scang.lib syntax expected inside {...}
padLibrary (<libname>) {...} // pad.lib syntax expected inside {...}
```

An example invocation for this method 3 is shown here:

tessent -shell -lib atpg.lib cell.lib pad.lib scang.lib

For complete syntax descriptions of these wrappers, refer to the *ETAssemble Reference Manual* and to the *ETPlanner Reference Manual*.

To write out a populated single file library, see the [write_cell_library](#) command.

Example 2

In this example, the cell_type *pad* is added to the model definitions in the output file of the following matching models: *pad_srr_hv*, *pad_multv_hv*, *pad_msr_hv*, *pad_ae_hv*, and *pad_lo_hv*.

- Any output pin on any of these models with a pinname of *ipp_do* will be assigned the *pad_to_pad* attribute.
- Similarly, any input pin beginning with *ipp_in* will be assigned the *pad_from_pad* attribute and so on for each of the *output*, *input*, and *inout* statements.

The last two cells declare scan cell attributes as described here:

- In the second cell declaration, model *FD1T* will have a model-level attribute *nonscan_model = FD1* added to its output, *FD2T* will be assigned *FD2*, and so on through *FD9T*. This assumes that *FD1T* through *FD9T* exist and therefore would match the regexp.
- In the third cell declaration, *FD3T* will be assigned *nonscan_model = DFF* instead of *nonscan_model = FD3* because it is overridden in the third cell by the wrapper's *nonscan_model* attribute statement.
- Because model *FD3T* matches “FD([1-9])T(.*)” in the second cell, *FD3T* will have attributes *scan_out*, *scan_enable*, and *scan_in* placed on pins *SO*, *TE*, and *TI* respectively (assuming the pinnames and direction for each exist).
- *FD3T* will also have a *scan_out_inv* attribute placed on pin *SO_B* in the third cell declaration; this attribute will not be placed on any of the other cells even if they have inverted output pins named *SO_B* because their model name will not match “FD3T” (model names are unique in ATPG libraries).

```
// Note: Single line comments must start with "//" and are terminated
// by <cr>/eol. These can occur at the end of a statement, as shown here:
//
// input (<pinname>) ( ) // Comment is legal after end statement
//
// Variable line comments start with a '/*' and are terminated by '*/'.
//
// /* This is a variable-line comment. */
//
// It is especially useful for commenting out entire models at times.
// However, it can be within a syntactic statement also, such as follows:
// instance = my_model inst_name (.A(net_in), .B(D)/*output*/, .C(Y));
//
// End Note -- continuing with Example.
```

```
regexp_verbosity = verbose; // Helps debug pin name regexp.
```

```
cell_attributes {
    cell ("pad_(ssr|multv|msr|ae|lo)_hv") { // first cell
        cell_type = pad;
        output ("ipp_do") ( pad_to_pad; )
    }
}
```

```

    input ("ipp_in.*") ( pad_from_pad; )
    input ("ipp_obe")   ( pad_enable_high; )
    inout ("(pad|pad_p)") ( pad_pad_io; )
    inout ("pad_n")     ( pad_pad_io_inv; )
}

cell ("FD([1-9])T(.*)") { // second cell
    nonscan_model = "FD\1\2";
    cell_type = scan_cell;
    output ("SO") ( scan_out; )
    input ("TE") ( scan_enable; )
    input ("TI") ( scan_in; )
}

cell ("FD3T") { // third cell
    nonscan_model = "DFF";
    cell_type = scan_cell;
    output ("SO_B") (scan_out_inv; )
}
}

```

Example 3

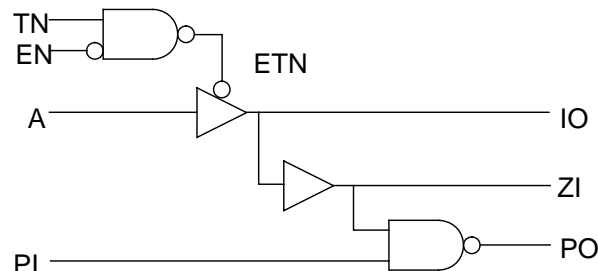
Figure 3-2 illustrates the inout and output attribute assignments with the bidirectional buffer, BIBUF, whose hardware description is described here:

```

model BIBUF(IO, A, EN, TN, PI, ZI, PO) (
    input(A, PI, EN, TN) (no_fault = sa0;)
    inout(IO) ()
    output(ZI) ()
    output(PO) ()
    (
        primitive = _inv not_EN(EN, not_EN_net);
        primitive = _nand nand1(TN, not_EN_net, ETN);
        primitive = _tsl out_driver(A, ETN, IO);
        primitive = _buf in_driver(IO, ZI);
        primitive = _nand nand2(ZI, PI, PO);
    )
)

```

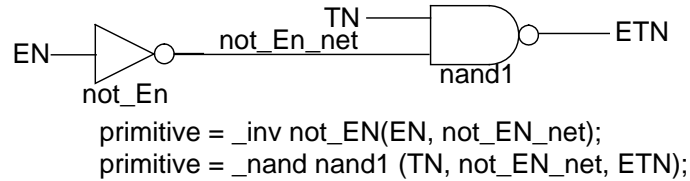
Figure 3-2. Combinational Logic



First, you examine the internal structure of the model and identify two 2-input NAND gates, one tri-state buffer, and one non-inverting buffer. Based on the structures of individual elements, assign attributes as follows:

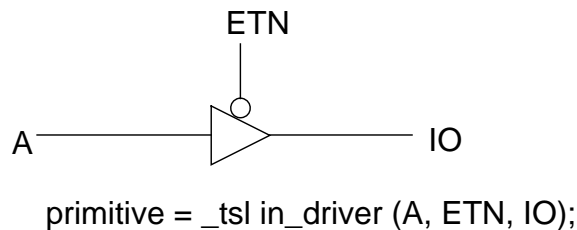
- An internal node such as not_en_net or ETN can be implied by referencing a name in a port list that is not an input, output, or inout pin of the model. For example:

Figure 3-3. Implying an Internal Node



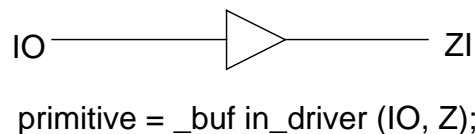
- For the tri-state buffer with input and active low pins, you can use a *primitive* attribute statement, “_tsl out_driver (A,ETN,IO)” (tsl = tri-state low), to model the driver of the bidirectional pin IO. Note that the internal node ETN is treated as the enable pin for the tri-state buffer because it is the second input to the _tsl primitive.

Figure 3-4. Tri-State Buffer



- For the non-inverting buffer whose input comes from pin IO simply use the *primitive* attribute statement “primitive = _buf in_driver (IO, ZI)” to describe it. Therefore, ZI will be an X state, if IO is a Z state.

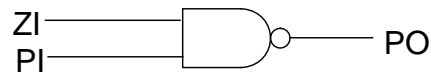
Figure 3-5. Non-Inverting Buffer



Note that _buf will convert a Z at IO into an X at ZI. If Z must be passed, use a wired ON nmos primitive or an _wire primitive.

- For the two-input NAND gate, use the *primitive* attribute statement “primitive = _nand nand2 (ZI, PI, PO)” to describe the logic driving the output pin PO.

Figure 3-6. Two-input NAND Gate

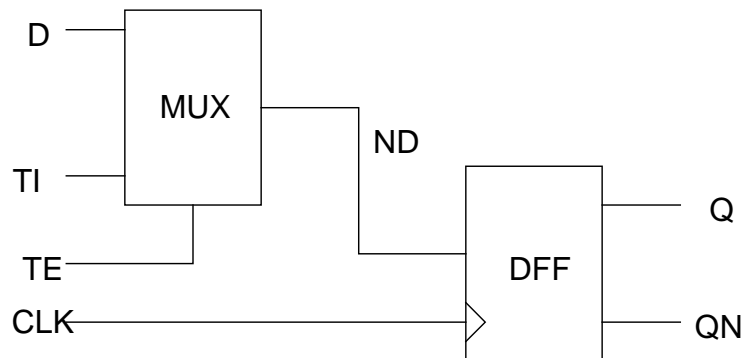


```
primitive = _nand nand2 (ZI, PI, PO);
```

```
model BIBUF(IO, A, EN, TN, PI, ZI, PO)
  input(A, PI, EN, TN) (no_fault = sa0;)
  inout(IO) ( )
  output(ZI) ( )
  output(PO) ( )
  (
    primitive = _inv not_EN(EN, not_EN_net);
    primitive = _nand nand1(TN, not_EN_net, ETN);
    primitive = _tsl out_driver(A, ETN, IO);
    primitive = _buf in_driver(IO, ZI);
    primitive = _nand nand2(ZI, PI, PO);
  )
)
```

Another example illustrating the hardware description of a mux DFF scan cell follows:

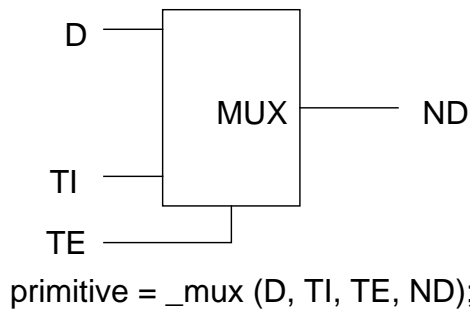
Figure 3-7. Mux-DFF Scan Cell



Based on the internal structure of SDFF, you will have to create one multiplexer and one D flip-flop as follows:

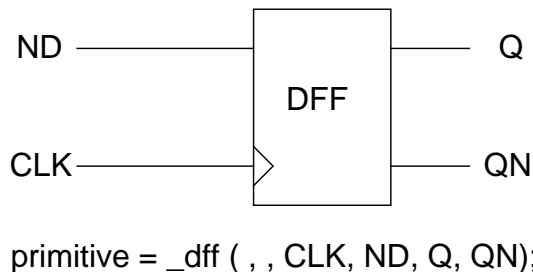
- Use the *primitive* statement “_mux(D, TI, TE, ND)” to describe the multiplexer. Syntax: `primitive =_mux(I0, I1, CNT, OUT)` where $OUT = CNT * I1 + \sim CNT * I0 + I1 * I0$.

Figure 3-8. The MUX



- Use the *primitive* statement "`_dff (, , CLK, ND, Q, QN)`" to describe the D flip-flop. Syntax: `primitive =_dff(SET, RESET, CLK, DATA, Q, QN)`. SET and RESET pins do not exist on the desired instantiation, so you should leave an empty pin to indicate this.

Figure 3-9. The DFF



```
model SDFF(D, CLK, TI, TE, Q, QN) (
    input(D, TI, TE) ( )
    input(CLK) ( )
    output(Q,QN) ( )
    (
        primitive = _mux(D, TI, TE, ND);
        primitive = _dff( , , CLK, ND, Q, QN);
    )
)
```

Note

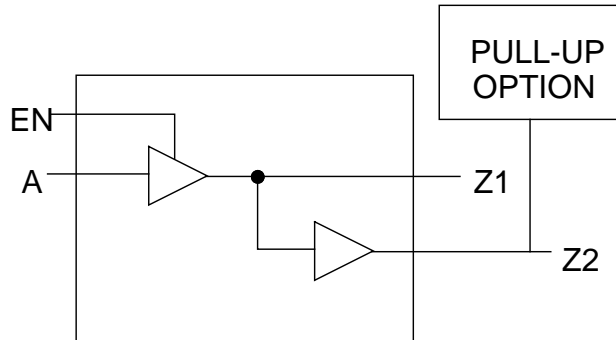


There is a clarification for the usage of a single input `_wire` primitive, wired ON nmos primitive, and the `_buf` primitive attribute statement. The following example, which is a tri-state gate feeding two primary output pins, will be used to explain the differences when different attribute statements are chosen for describing cell function.

```
model TS(A, EN, Z1, Z2) (
    input(A, EN) ( )
    output(Z1) ( )
    output(Z2) ( )
    (
        primitive = _tsh(A, EN, Z1);
        primitive = _buf(Z1, Z2);
    )
)
```

)
)

Figure 3-10. Tri-State Gate (_buf primitive)

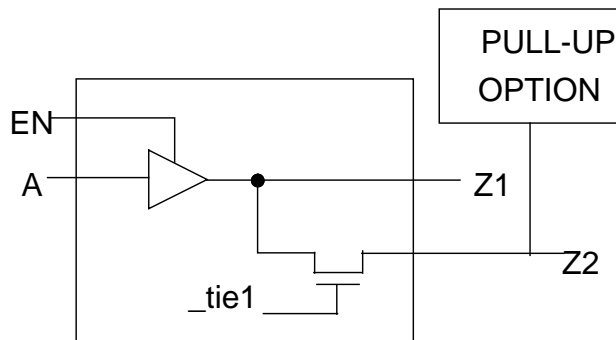


When this model is compiled, a combinational buffer is created between output pin Z1 and output pin Z2. The effect of modeling this way stops a Z state which can propagate to output pin Z1 from propagating to output pin Z2. The buffer causes the Z in its input to become an X (unknown) value at Z2. If there is an external pull up/down gate connected to output pin Z2, the effect of the pull up/down will not show up at output pin Z1.

- Here is an example using the wired ON nmos primitive:

```
model TS(A, EN, Z1, Z2) (
  input(A, EN) ( )
  output(Z1) ( )
  output(Z2) ( )
  (
    primitive = _tsh driver (A, EN, Z1);
    primitive = _tie1 one (one_net);
    primitive = _nmos pass_Z (Z1, one_net, Z2);
  )
)
```

Figure 3-11. Tri-State Gate (_nmos primitive)



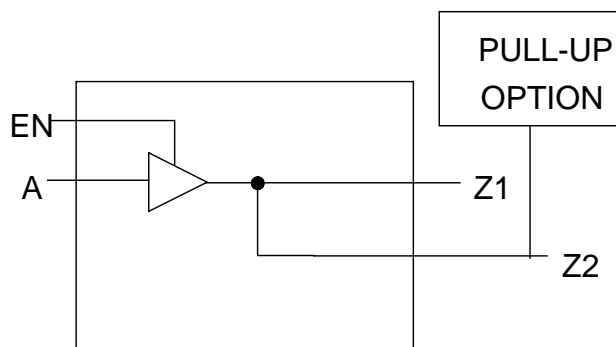
When this model is compiled, a wired on switch will be created for output pin Z2, and a Z state can propagate to pin Z2 through the switch. However, if there is an external pull

up/down gate connected to output pin Z2, the effect of the pull up/down will not show up at output pin Z1.

- Finally, here is an example using the `_wire` primitive:

```
model TS(A, EN, Z1, Z2) (  
    input(A, EN) (  
    output(Z1) (  
    output(Z2) (  
    (  
        primitive = _tsh (A, EN, Z1);  
        primitive = _wire (Z1, Z2);  
    )  
)
```

Figure 3-12. Tri-State Gate (`_wire` primitive)



When this model is compiled, buses will be created for output pin Z1 and output pin Z2, respectively. If output pin Z1 is a Z, Z2 will also be a Z. If there is an external pull up/down gate connected to output pin Z2, the effect of the pull up/down will show up at output pin Z1, and vice versa.

Internal Faults

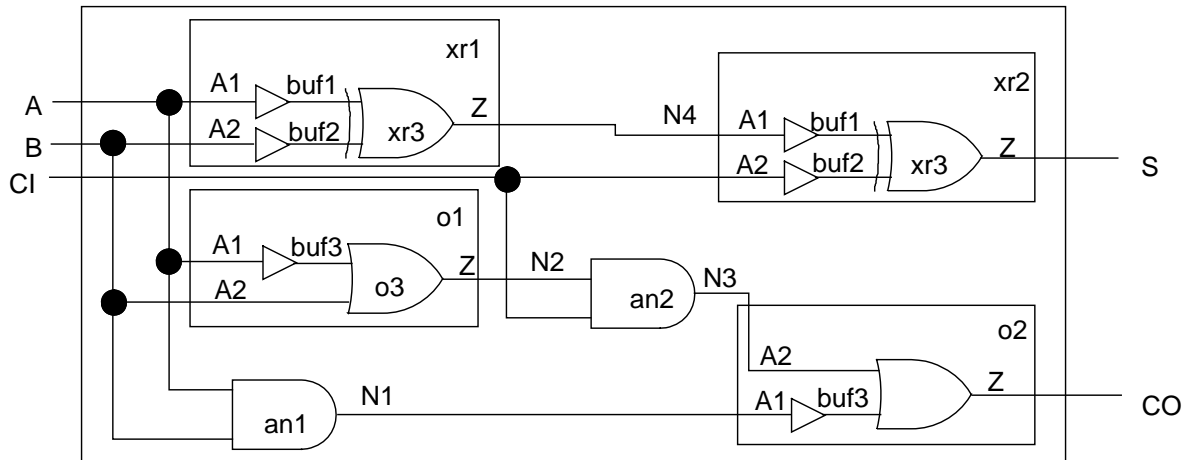
This section describes how to place internal faults.

By default, faults are placed on all interfaced pins of a cell model. Any of these interfaced pins can be selected not to be faulted. If the cell model is complex and the user wants to fault some of the pins inside the cell, this can be accomplished with *primitive* and *instance* attribute statements.

Three attribute statements describe the connectivity of cell models: *function*, *primitive*, and *instance*. Since, there is no instance name and pin name associated with the *function* attribute statement, there is no way to place faults on the function. The *primitive* and *instance* attribute statements allow instance names in order to handle faulting internal pins. The *fault* and *no-fault* attribute statements describe how to handle faulting or not faulting internal pins.

Figure 3-13 is an example using internal faults:

Figure 3-13. Internal Faults



In this example, a netlist models an adder and contains one instance, U1, which refers to the library name “adder”. The primary inputs are A, B, and CI. The primary outputs are S and CO. The library model description for the adder can be described with internal faulting as follows:

```
model adder(CI, A, B, S, CO) (
  input(A, B, CI) ()
  output(S) ()
  output(CO) ()
  (
    fault=internal;
    instance=xor2 xr1(A, B, N4);
    instance=and2 an1(A, B, N1);
    instance=or2 o1(A, B, N2);
    fault=boundary internal;
    instance=xor2 xr2(N4, CI, S);
    fault=boundary;
    primitive= _and an2(N2, CI, N3);
    instance=or2 o2(N1, N3, CO);
  )
)
model xor2(A1, A2, Z) (
  input(A1, A2) ()
  output(Z) ()
  (
    primitive=_xor xr3(N1:nf, N2, Z);
    fault=none;
    instance= buff1 buf1(A1, N1);
    fault=boundary;
    primitive= _buf buf2(A2, N2);
  )
)
model and2(A1, A2:nf0, Z) (
  input(A1, A2) ()
  output(Z) ()
  (
    fault = none;
    primitive = _and an3(A1, A2, Z);
  )
)
```

```

    )
  )
  model or2(A1, A2, Z) (
    input(A1, A2) ()
    intern(N1) ()
    output(Z) ()
    (
      fault=internal;
      instance=buf1 buf3(A1, N1);
      fault=boundary;
      primitive=_or o3(N1, A2, Z:nf1);
    )
  )
  model buf1(I, Z) (
    input(I) ()
    output(Z) ()
    (
      fault = boundary;
      primitive = _buf buf4(I, Z);
    )
  )
)

```

When all faults are added to this example, using the `add_faults` command, these faults are placed as follows:

1. Stuck-at-0 and stuck-at-1 faults are placed on the primary inputs and primary outputs:

```

/A
/B
/CI
/S
/CO

```

2. Faults are placed on the internals of instance `xr1`. This instance name refers to the library model `xor2`. Within library model `xor2`, no-faults are placed on the instance name `buf1`. Stuck-at-0 and stuck-at-1 faults are placed on the boundary of the buffer primitive with instance name `buf2`. For the buffer primitive, `IN` is the input pin name, and `OUT` is the output pin name:

```

/U1/xr1/buf2/IN
/U1/xr1/buf2/OUT

```

Since library model `xor2` has an instance name `xr3`, and has a no-fault attribute statement, then by default, the *instance_fault* attribute is set to boundary. For the XOR primitive, `IN0` and `IN1` are the input pin names, `OUT` is the output pin name.

However, a *no-fault* attribute statement is placed on the first pin of the XOR primitive (`IN0`). So, stuck-at-0 and stuck-at-1 faults are only placed on the `IN1` and `OUT` pins of the XOR primitive:

```

/U1/xr1/xr3/IN1
/U1/xr1/xr3/OUT

```

3. Faults are placed on the boundary and the internals of instance xr2. This instance name refers to the internals and boundary of library model xor2. Stuck-at-0 and stuck-at-1 faults are placed on the boundary of library model xor2:

```
/U1/xr2/A1
/U1/xr2/A2
/U1/xr2/Z
```

Within library model xor2, no-faults are placed on the instance name buf1. Faults are placed on the boundary of the buffer primitive with instance name buf2. For the buffer primitive, IN is the input pin name, and OUT is the output pin name:

```
/U1/xr2/buf2/IN
/U1/xr2/buf2/OUT
```

Since library model xor2 has an instance name xr3, but has *no-fault* attribute statement, then by default, the instance_fault attribute is set to boundary. For the XOR primitive, IN0 and IN1 are the input pin names, OUT is the output pin name. However, a no-fault attribute is placed on the first pin of the XOR primitive (IN0). So, stuck-at-0 and stuck-at-1 faults are only placed on the IN1 and OUT pins of the XOR primitive:

```
/U1/xr2/xr3/IN1
/U1/xr2/xr3/OUT
```

4. Faults are placed on the internals of instance an1. The instance name refers to the library model and2. However, since the library model contains an AND primitive and a *instance_fault* attribute statement set to none, no instance_faults are placed.
5. Faults are placed on the internals of instance o1. This instance name refers to the library model or2. Within library model or2, faults are placed on the internals of instance buf3. This instance name refers to the library model buff1. Within library model buff1, stuck-at-0 and stuck-at-1 faults are placed on the boundary of the buffer primitive with the instance name buf4. For the buffer primitive, IN is the input pin name, and OUT is the output pin name:

```
/U1/o1/buf3/buf4/IN
/U1/o1/buf3/buf4/OUT
```

Faults are placed on the boundary of the OR primitive with instance name o3. For the OR primitive, IN0 and IN1 are the input pin names, OUT is the output pin name. However, a stuck-at-1 *no-fault* attribute statement is placed on the output pin of the OR primitive (OUT). So, only a stuck-at-0 fault is placed on the OUT pin of the OR primitive:

```
/U1/o1/o3/IN0
/U1/o1/o3/IN1
/U1/o1/o3/OUT    (stuck-at-0 fault only)
```

6. Faults are placed on the boundary of the AND primitive with instance name an2. For the AND primitive, IN0 and IN1 are the input pin names, OUT is the output pin name:

```
/U1/an2/IN0  
/U1/an2/IN1  
/U1/an2/OUT
```

7. Finally, faults are placed only on the boundary of instance o2. Though instance o2 refers to library model or2 and has internal faults, no-faults are placed within the library model. The boundary pins for library model or2 are A1, A2, and Z:

```
/U1/o2/A1  
/U1/o2/A2  
/U1/o2/Z
```

Support of Arrays Within Library Models

To support arrays in library models, an *array* attribute statement can be used in the *input*, *output*, *inout*, and *intern* statements.

The array syntax is as follows:

```
array = start : end;
```

Array is the keyword; start and end are integers greater than or equal to 0. If start is greater than end, the array is in descending order; otherwise, the array is in ascending order.

Arrays should be declared before they are referenced in the *primitive*, *instance*, or *function* statements. The symbols '<' and '>' are reserved for the array delimiters. If the user wants to redefine the array delimiter after the library models are parsed, the *array_delimiter* statement can be used. The syntax is as follows:

```
array_delimiter = "<>" | "()" | "{}" | "[]";
```

Array_delimiter is the keyword, and this statement is only defined once and must be used before any library models with the *array* attribute statement are defined. If this statement is not defined in the library, "<>" will be assumed.

Here is an example using the *array* attribute statement and *array_delimiter* statement:

```
array_delimiter = "[]";  
model RAM1(W1, A1, D1, R2, A2, D2) (  
  input(W1, R2) ()  
  input(A1, A2) (array = 5: 0;)  
  input(D1) (array = 0 : 4;)  
  output(D2) ()  
  (  
    data_size = 5;  
    address_size = 6;  
    read_off = 0;  
    min_address = 0;  
    max_address = 63;
```

```

        primitive = _ram U1 ( , ,
            _write(W1, A1, D1),
            _read(R2, A2, D2)
        );
    )
) // end model RAM1

```

Support for Array Index and Array Range

For any input/output/inout (pinname) (array = .. where the array is > 1 bit, a bit select / index must be used to identify when only a subset of bits has an attribute.

For example:

```

input (in_arr_1) ( array = 2:0;
                  index(0) (tie1);
                  index(2:1) (tie0)
                )

```

would specify that the LS bit, in_arr_1[0], should be tied to 1 if the cell is inserted by test tools, and the MS bit in_arr_1[2], as well as in_arr_1[1], should be tied to 0.

If all bits of an array have the same attribute, the simple form:

```

input (in_arr_2) (array = 1:0; tie1; )

```

can be used to declare that every bit of the array has that attribute.

Example Scan Definitions

Because it is required for scan insertion, the design library should provide information for mapping non-scan models to their associated scan cell models.

This information is found in the model description of a scan cell. The specific scan information includes the scan input pin, scan output pin, scan enable pin and the mapping of scan cell model to its non-scan cell model.

The following subsections contain example scan definitions for various types of cells.

Basic Example

The following is a general example of the usage of several of the attributes:

```

model FD3SP(D, CP, TI, TE, CD, SD, Q, QN) (
    nonscan_model = FD2P;
    cell_type = scan_cell;
    input (SD) (tie1)
    input (D) (data_in)
    input (TI) (scan_in)
    input (TE) (scan_enable)
    input (CP) (posedge_clock)
)

```

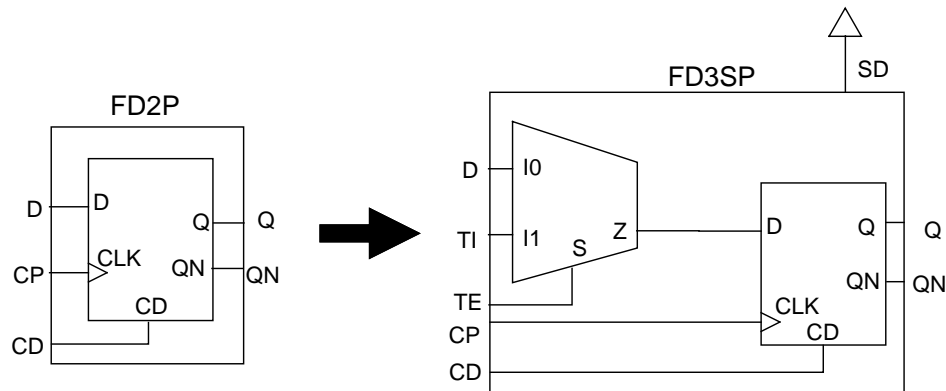
```

output (Q) (scan_out)
output (QN) (scan_out_inv)
(
    <instance and primitive attributes>
)
)

```

Figure 3-14 shows the non-scan to scan cell replacement that is defined in the preceding scan definition.

Figure 3-14. General Scan Definition Replacement Example



MUX-Scan Cell

The following is an example definition for a MUX-Scan cell:

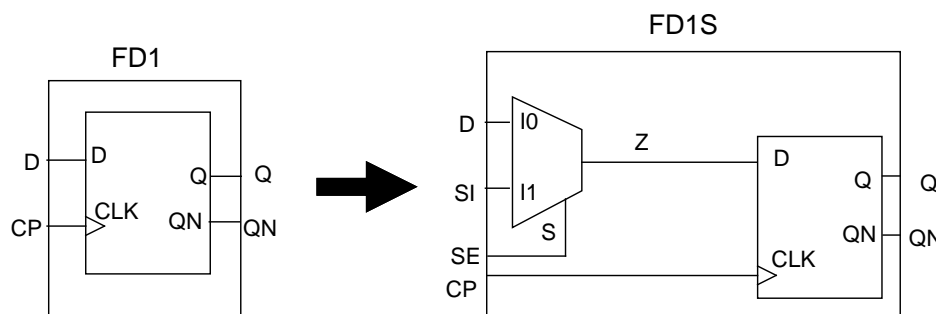
```

model FD1S(D, CP, SI, SE, Q, QN) (
    nonscan_model = FD1;
    cell_type = scan_cell;
    input (SI) (scan_in)
    input (SE) (scan_enable)
    input (CP) (posedge_clock)
    output (Q) (scan_out)
    output (QN) (scan_out_inv)
    (
        <instance and primitive attributes>
    )
)

```

Figure 3-15 shows this non-scan to scan cell replacement defined above.

Figure 3-15. Mux-Scan Definition Replacement Example



Defining Macros

Design libraries for the DFT products support macro descriptions.

The syntax for a macro description, which is similar to a model description, is as follows:

```
macro macro_name (list_of_pins) (
    input (input_pins) ...
    output (output_pins) ...
    inout (inout_pins) ...
    intern (internal_nodes) ...
    ( ...
    )
)
```

Macro descriptions support nearly all the statements that model descriptions support, with the following restrictions:

- Macros can be referenced by other macros, but not other models.
- *Function* attribute statements are not allowed.
- *Primitive* attribute statements are not allowed.
- Instance names are required.

If macros are used to describe scan cell models, Tessent Scan expands the macro into modules when writing Genie format output. When writing any other format, Tessent Scan writes out the macro as a separate module.

Reusing a Model Definition

This section describes how you can reuse a model definition.

Many times a library will include several components with the same function but different timing characteristics. The DFT library needs only the functional information for a cell, not the

timing. Therefore, to simplify model creation for cells with the same logic functions, you can use the following statement at the library level where a normal macro or model would be defined. The syntax of the statement is as follows:

```
model new_model_name = defined_model_name ; // Same ports and hardware
macro new_macro_name = defined_macro_name ; // Same ports and hardware
```

The *new_model_name* (or *new_macro_name*) argument following the model (or macro) keyword specifies a cell name that is functionally equivalent to a model named *defined_model_name*, which is a model that is fully described elsewhere in the library.

An example follows. Note that the TBUF model is fully described, while a functionally equivalent model, TBUFH, has its port order, port names, port directions, and the transfer function between ports defined by referencing those of TBUF.

```
// =====
// fastscan model: tbuf
// =====
model TBUF (X, A, ENB) (
    input(A, ENB) ()
    output(X) (
        primitive = _tsl a (A,ENB, X);
    )
)
// =====
// fastscan model: tbufh
// =====
model TBUFH = TBUF;
```

Reading Multiple Libraries

This section describes how to read multiple libraries within one main library.

In the custom design environment, all design cells may not be created and maintained by a single user or group. To avoid having to maintain one complete library (which may be created by concatenating all subsets of the libraries) and many subsets of libraries consistently, you can specify reading multiple libraries within one main library by adding the following statement to the library:

```
#include "library_filename"
```

There should be no space between “#” and “include”, and the library filename should be enclosed in double quotes. This statement can only be placed between model descriptions and cannot be placed inside a model description.

The library parses the *library_filename* file as if it were inserted in the position where the *#include* statement occurs. This order is only important if duplicate model names exist in multiple libraries; in this case, the last one parsed is used as the definition of a particular model name. Any references to that model name inherit that model’s ports and hardware definition. Here is an example that first uses the “#include” statement to read two files (parsing the models

and macros in the order they are read) and then reads the model an2. The an2 model is parsed after the two include files, and will therefore define “an2” if a duplicate an2 model exists in either of the #include files:

```
#include "/home/users/library/set1.lib"
#include "/home/users/library/set2.lib"

model an2(A1, A2, X)    (
    input(A1, A2)      ()
    output(X)          ( )
    (
        ...
    )
)
...
```

Verilog Primitives

Tessent tools understand Verilog primitives without requiring an ATPG model to be created.

For example, the Verilog “and” directly maps to the built-in primitive “and” in DFT tools, and therefore no model is needed. Even if an ATPG library model named “and” is present, it will be ignored by the tool because the Verilog reader will recognize this model as a primitive which the tool already understands.

A list follows of the Verilog primitives that the DFT tools handle directly without requiring an ATPG library model:

Table 3-6. Supported Verilog Primitives

and	nand	notif1	rcmos	xnor
buf	nmos	or	rnmos	xor
bufif0	nor	pmos	rpmos	
bufif1	not	pulldown	rtran ¹	
cmos	notif0	pullup	tran ²	

1. The _pull primitive is unidirectional, and none of the test simulation primitives can model a bi-directional resistive transfer in the general case.

2. The tool uses a built-in unidirectional ATPG model for this primitive; thus, the ATPG behavior is not the same as the Verilog primitive. Take care that the built-in model is suitable for your purposes.

Note



UDPs can be parsed and synthesized by the DFT tools, however it is recommended that LibComp be used to create ATPG library models from UDP tables.

Note



Take care when using tran and rtran Verilog primitives as the Verilog simulator treats them differently than the test simulators.

Supported Primitives

The following pages contain descriptions, truth tables, and examples of the tool-supported primitives. When defining a primitive, you must understand the pin sequence of the primitive. The sequence of the pin names is important to the primitive definition. You must use a comma as a separator to keep the fixed pin sequence format for any unused pin in the primitive.

The library supports regular and resistive primitives. The drive strength of the outputs for regular and resistive primitives are different. The possible output drive strengths of a regular primitive are: 0, 1, X (unknown), and Z (high impedance). The possible output drive strengths of a resistive primitive are: weak 0, weak 1, weak X (unknown), and Z (high impedance).

Note

Use transistor primitives only under carefully controlled conditions. Building models from transistors to match the simulation or actual gate representation does not guarantee the models will be testable in ATPG. The best practice is to use the tool's highest level built-in gate primitives.

The following is a list of the supported primitives:

- [“AND Gate”](#) on page 133
- [“NAND Gate”](#) on page 134
- [“OR Gate”](#) on page 134
- [“NOR Gate”](#) on page 136
- [“Inverter”](#) on page 137
- [“Buffer”](#) on page 138
- [“XOR Gate”](#) on page 138
- [“XOR Gate”](#) on page 138
- [“XNOR Gate”](#) on page 140
- [“Tri-State Buffer with Active Low Control”](#) on page 141
- [“Inverted Tri-State Buffer with Active Low Control”](#) on page 142
- [“Tri-State Buffer with Active High Control”](#) on page 143
- [“Inverted Tri-State Buffer with Active High Control”](#) on page 144
- [“Multiplexer”](#) on page 145
- [“D Flip-Flop”](#) on page 146
- [“D Latch”](#) on page 149

- [“One Time Unit Delay Element \(FlexTest Only\)”](#) on page 151
- [“Feedback Inverter”](#) on page 152
- [“XDET”](#) on page 152
- [“Wire Element”](#) on page 154
- [“Pull-Up or Pull-Down Device”](#) on page 155
- [“Power Signal”](#) on page 156
- [“Ground Signal”](#) on page 156
- [“High Impedance Signal”](#) on page 157
- [“Undefined”](#) on page 158
- [“Unidirectional NMOS Transistor”](#) on page 159
- [“Unidirectional PMOS Transistor”](#) on page 159
- [“Unidirectional Resistive NMOS Transistor”](#) on page 161
- [“Unidirectional Resistive PMOS Transistor”](#) on page 162
- [“Unidirectional Feedback NMOS Transistor”](#) on page 163
- [“Unidirectional Feedback PMOS Transistor”](#) on page 164
- [“Unidirectional CMOS Transistor”](#) on page 165
- [“Unidirectional Resistive CMOS Transistor”](#) on page 166
- [“Unidirectional Resistive Feedback CMOS Transistor”](#) on page 167
- [“Pulse Generators with User Defined Timing”](#) on page 168
- [“RAM and ROM”](#) on page 169

AND Gate

The primitive used to model an AND gate is **_and**. The syntax of the primitive attribute statement is as follows:

```
primitive = _and optional_inst_name (IN0, IN1, ..., INn, OUT);
```

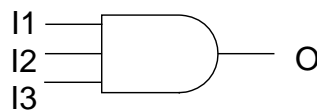
Table 3-7. AND Truth Table

IN0	IN1	OUT
0	0/1/X/Z	0
0/1/X/Z	0	0
1	1	1
1/X/Z	X/Z	X
X/Z	1/X/Z	X

Example:

```
model AND3(I1, I2, I3, O) (
  input(I1, I2, I3) ()
  output(O) ()
  (
    primitive = _and and_inst(I1, I2, I3, O);
  )
)
```

Figure 3-16. AND Gate



NAND Gate

The primitive used to model a NAND gate is **_nand**. The syntax of the primitive attribute statement is as follows:

```
primitive = _nand optional_inst_name(IN0, IN1, ..., INn, OUT);
```

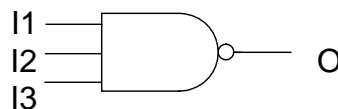
Table 3-8. NAND Truth Table

IN0	IN1	OUT
0	0/1/X/Z	1
0/1/X/Z	0	1
1	1	0
1	X/Z	X
X/Z	1	X

Example:

```
model NAND3(I1, I2, I3, O) (
  cell_type = nand;
  input(I1, I2, I3) ()
  output(O) ()
  (
    primitive = _nand nand_inst(I1, I2, I3, O);
  )
)
```

Figure 3-17. NAND Gate



OR Gate

The primitive used to model an OR gate is **_or**. The syntax of the primitive attribute statement is as follows:

```
primitive = _or optional_inst_name(IN0, IN1, ..., INn, OUT);
```

Table 3-9. OR Truth Table

IN0	IN1	OUT
0	0	0
0/1/X/Z	1	1

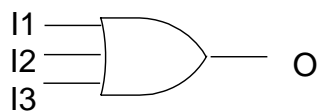
Table 3-9. OR Truth Table

IN0	IN1	OUT
1	0/1/X/Z	1
X/Z	0/X/Z	X
0/X/Z	X/Z	X

Example:

```
model OR3(I1, I2, I3, O) (
  cell_type = or;
  input(I1, I2, I3) ()
  output(O) ()
  (
    primitive = _or my_or(I1, I2, I3, O);
  )
)
```

Figure 3-18. OR Gate



NOR Gate

The primitive used to model a NOR gate is **_nor**. The syntax of the primitive attribute statement is as follows:

```
primitive = _nor optional_inst_name(IN0, IN1, ..., INn, OUT);
```

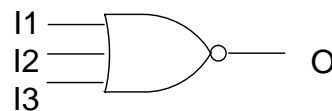
Table 3-10. NOR Truth Table

IN0	IN1	OUT
0	0	1
0/1/X/Z	1	0
1	0/1/X/Z	0
X/Z	0/X/Z	X
0/X/Z	X/Z	X

Example:

```
model NOR3(I1, I2, I3, O) (  
  cell_type = nor;  
  input(I1, I2, I3) ()  
  output(O) ()  
  (  
    primitive = _nor nor_1(I1, I2, I3, O);  
  )  
)
```

Figure 3-19. NOR Gate



Inverter

The primitive used to model an inverter is **_inv**. The syntax of the primitive attribute statement is as follows:

```
primitive = _inv optional_inst_name(IN, OUT);
```

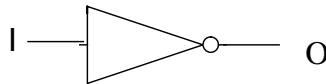
Table 3-11. Inverter Truth Table

IN	OUT
0	1
1	0
X/Z	X

Example:

```
model INV1(I, O) (
    cell_type = inverter;
    input(I) ()
    output(O) ()
    (
        // Example that uses no unique instantiation name.
        primitive = _inv (I, O);
    )
)
```

Figure 3-20. Inverter



Buffer

The primitive used to model a buffer is **_buf**. The syntax of the primitive attribute statement is as follows:

```
primitive = _buf optional_inst_name(IN, OUT);
```

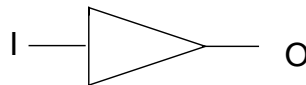
Table 3-12. Buffer Truth Table

IN	OUT
0	0
1	1
X/Z	X

Example:

```
model BUF1(I, O) (
    cell_type = buffer;
    input(I) ()
    output(O) ()
    (
        primitive = _buf my_buffer(I, O);
    )
)
```

Figure 3-21. Buffer



XOR Gate

The primitive used to model a XOR is **_xor**. The syntax of the primitive attribute statement is as follows:

```
primitive = _xor optional_inst_name(IN0, IN1, ..., INn, OUT);
```

Table 3-13. XOR Truth Table

IN0	IN1	OUT
0	0	0
0	1	1
1	0	1
1	1	0
X/Z	0/1/X/Z	X

Table 3-13. XOR Truth Table

IN0	IN1	OUT
0/1/X/Z	X/Z	X

Example:

```

model XOR1(A, B, Z) (
  cell_type = xor;
  input(A, B) ()
  output(Z) ()
  (
    primitive = _xor xor_1(A, B, Z);
  )
)

```

Figure 3-22. XOR Gate



XNOR Gate

The primitive used to model a XNOR is **`_xnor`**. The syntax of the primitive attribute statement is as follows:

```
primitive = _xnor optional_inst_name(IN0, IN1, ..., INn, OUT);
```

Using this primitive is more efficient than using functions.

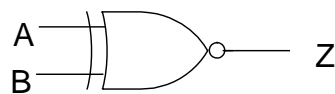
Table 3-14. XNOR Truth Table

IN0	IN1	OUT
0	0	1
0	1	0
1	0	0
1	1	1
X/Z	0/1/X	X
0/1/X	X/Z	X

Example:

```
model XNOR1(A, B, Z) (  
    input(A, B) (  
        output(Z) (  
            (  
                primitive = _xnor(A, B, Z);  
            )  
        )  
    )  
)
```

Figure 3-23. XNOR Gate



Tri-State Buffer with Active Low Control

The primitive used to model a tri-state buffer with an active low control is `_tsl`, and the syntax of the primitive attribute statement is as follows:

```
primitive = _tsl optional_inst_name(IN, CNT, OUT);
```

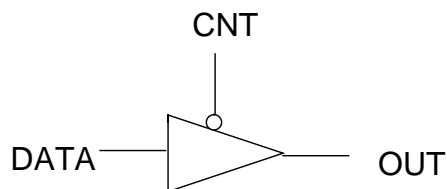
Table 3-15. TSL Truth Table

IN	CNT	OUT
0	0	0
1	0	1
Z	0	X
0/1/X/Z	1	Z
0/1/X/Z	X	X

Example:

```
model TSL1(DATA, CNT, OUT) (
  input(DATA, CNT) ()
  output(OUT) ()
  (
    primitive = _tsl act_lo_en(DATA, CNT, OUT);
  )
)
```

Figure 3-24. Tri-State Buffer with Active Low Control



Inverted Tri-State Buffer with Active Low Control

The primitive used to model an inverted tri-state buffer with an active low control is `_tsli`. The syntax of the primitive attribute statement is as follows:

```
primitive = _tsli optional_inst_name(IN, CNT, OUT);
```

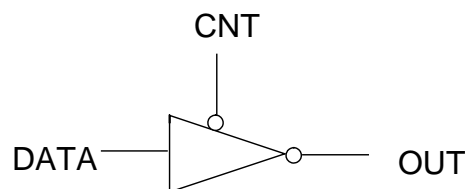
Table 3-16. TSLI Truth Table

IN	CNT	OUT
0	0	1
1	0	0
Z	0	X
0/1/X/Z	1	Z
0/1/X/Z	X	X

Example:

```
model TSLI1(DATA, CNT, OUT) (  
    input(DATA, CNT) (  
        output(OUT) (  
            (  
                primitive = _tsli(DATA, CNT, OUT);  
            )  
        )  
    )  
)
```

Figure 3-25. Inverted Tri-State Buffer with Active Low Control



Tri-State Buffer with Active High Control

The primitive used to model a tri-state buffer with a active high control is **_tsh**, and the syntax of the primitive attribute statement is as follows:

```
primitive = _tsh optional_inst_name(IN, CNT, OUT);
```

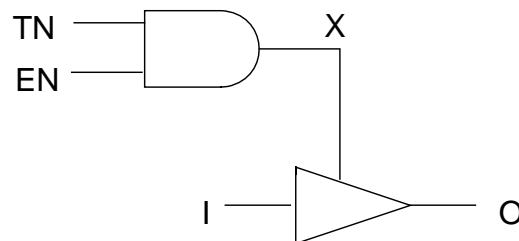
Table 3-17. TSH Truth Table

IN	CNT	OUT
0	1	0
1	1	1
Z	1	X
0/1/X/Z	0	Z
0/1/X/Z	X	X

Example:

```
model TSH1(I, EN, TN, O) (
  input(I, EN, TN) ()
  output(O) ()
  (
    primitive = _and(TN, EN, X);
    primitive = _tsh(I, X, O);
  )
)
```

Figure 3-26. Tri-State Buffer with Active High Control



Inverted Tri-State Buffer with Active High Control

The primitive used to model an inverted tri-state buffer with an active high control is **_tshi**, and the syntax of the primitive attribute statement is as follows:

```
primitive = _tshi optional_inst_name(IN, CNT, OUT);
```

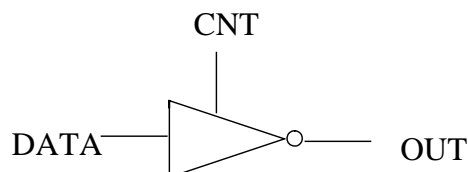
Table 3-18. TSHI Truth Table

IN	CNT	OUT
0	1	1
1	1	0
Z	1	X
0/1/X/Z	0	Z
0/1/X/Z	X	X

Example:

```
model TSHI1(DATA, CNT, OUT) (  
    input(DATA, CNT) ()  
    output(OUT) ()  
    (  
        primitive = _tshi(DATA, CNT, OUT);  
    )  
)
```

Figure 3-27. Inverted Tri-State Buffer with Active High Control



Multiplexer

The primitive used to model a two-to-one multiplexer is **_mux**, and the syntax of the primitive attribute statement is:

```
primitive = _mux optional_inst_name(IN0, IN1, CNT, OUT);
```

The output signal will be the same as input signal “IN0” when control signal CNT is low. The output signal will be the same as input signal “IN1” when the control signal CNT is high. Using this primitive is more efficient than using functions.

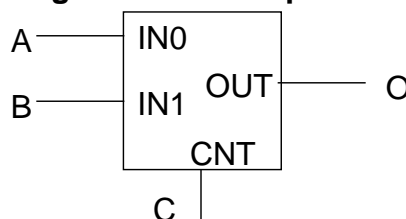
Table 3-19. MUX Truth Table

IN0	IN1	CNT	OUT
0	0/1/X/Z	0	0
1	0/1/X/Z	0	1
0/1/X/Z	0	1	0
0/1/X/Z	1	1	1
Z	0/1/X/Z	0	X
0/1/X/Z	Z	1	X
1	1	X	1
0	0	X	0
0	1	X	X
1	0	X	X

Example:

```
model MUX1(A, B, C, O) (
  cell_type = mux;
  input(A) (mux_in0)
  input(B) (mux_in1)
  input(C) (mux_select)
  output(O) ()
  (
    primitive = _mux consensus_mux_1(A, B, C, O);
  )
)
```

Figure 3-28. Multiplexer



D Flip-Flop

The keyword used to define a single or multiple port D flip-flop is `_dff`. It defines a DFF where *both asynchs are activeHI*, and whose clocks are posedge-triggered. The syntax of the primitive attribute statement is as follows:

```
primitive = _dff optional_inst_name(SET, RESET, CLK1, D1,  
                                   CLK2, D2, ..., CLKn, Dn, Q, QN);
```

This primitive allows users to define a D flip-flop with a single pair or multiple pairs of clock and data inputs. If this primitive is used to model a single port D flip-flop, the behavior in FlexTest may be modified by the attributes `reset_clock_conflict` and `set_clock_conflict`, which are defined in “[Defining Pin Attributes](#)” on page 98. Tessent FastScan, Tessent TestKompress, and Tessent Scan are not affected by these attributes.

[Table 3-20](#) shows the truth table for D flip-flop primitives for Tessent FastScan, Tessent TestKompress, and FlexTest. This is the default; the `set_simulation_options` -Set_reset_dominate_port switch is set to ON

Table 3-20. D Flip-Flop Primitives

D1	CLK1	SET	RESET	Q	QN
0	01	0	0/1/X	0	1
1	01	0/1/X	0	1	0
X	01	0	0	X	X
0/1/X	10/1X/X0	0	0	Q	QN
0/1/X	10/1X/X0	0	1	0	1
0/1/X	0/1/X	0	1	0	1
0/1/X	10/1X/X0	1	0	1	0
0/1/X	0/1/X	1	0	1	0
0/1/X	0/1/X	1	1	X	X
When CLK1 = 0X or X1, by default there is no clock transition and the DFF retains its previous values. But when you use the <code>set_xclock_handling</code> command with the ‘X’ option, the tool treats these values as potential rising transitions.					

[Table 3-21](#) shows the alternative truth table for the D flip-flop primitive for Tessent FastScan, Tessent TestKompress, and FlexTest when you disable set/reset port dominance using the `set_simulation_options` -Set_reset_dominate_port OFF switch.

Table 3-21. Alternative D Flip-Flop Primitive Table

D1	CLK1	SET	RESET	Q	QN
0	01	0	0/1/X	0	1

Table 3-21. Alternative D Flip-Flop Primitive Table

D1	CLK1	SET	RESET	Q	QN
1	01	0/1/X	0	1	0
X	01	0	0	X	X
0/1/X	10/1X/X0	0	0	Q	QN
1	01	0	1	X	X
0/1/X	10/1X/X0	0	1	0	1
0/1/X	0/1/X	0	1	0	1
0	01	1	0	X	X
0/1/X	10/1X/X0	1	0	1	0
0/1/X	0/1/X	1	0	1	0
0/1/X	0/1/X	1	1	X	X

When CLK1 = 0X or X1, by default there is no clock transition and the DFF retains its previous values. But when you use the [set_xclock_handling](#) command with the 'X' option, the tool treats these values as potential rising transitions.

If the primitive is used to model a multiple port D flip-flop, the behavior is not affected by the attributes `set_clock_conflict` and `reset_clock_conflict`. The default behavior for FlexTest, Tessent FastScan, and Tessent TestKompress of a multiple port D flip-flop is as follows:

1. If only one set, reset, or one of the clocks is active, Q and QN are well defined.
2. If more than one set, reset, or clock lines is active, and if the values captured by the active clocks, set, or reset are the same, Q and QN are well defined. Otherwise, Q and QN are unknown.

Examples:

```

model simple_posedge_DFF (d, clk, q) (
    cell_type = dff;
    input (d) (data_in)
    input (clk) (posedge_clock)
    output (Q) (data_out)
    ( // Leave 1st two (set and reset) pins empty to indicate not present
      // Leave the last (QN) pin empty to indicate no QB
      primitive = _dff ( , , clk, d, q, ) ;
    )
)

model simple_negedge_DFF (d, clk_b, q) (
    cell_type = dff;
    input (d) (data_in)
    input (clk_b) (negedge_clock)
    output (Q) (data_out)
    ( // All Mentor test primitives have activeHI asynchs and activeHI

```

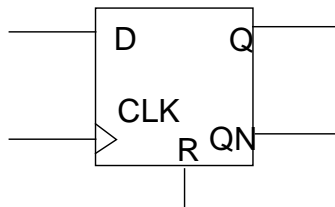
```

        //      or posedge clocks, so must invert cell input to model
        //      activeLO or negedge clock.
        primitive = _inv (clk_b, clk_net);
        primitive = _dff ( , , clk_net, d, q, );
    )
)

model active_high_reset_posedge_dff (D, CLK, R, Q, QN) (
    cell_type = dff;
    input(D) (data_in)
    input(CLK) (posedge_clock)
    input(R) (active_high_reset)
    output(Q) (data_out)
    output(QN) (data_out_inv)
    (
        primitive = _dff( , R, CLK, D, Q, QN);
    )
)

```

Figure 3-29. D Flip-Flop



In the above example, the D flip-flop does not have a set pin; therefore, it is required to have a comma as the separator to indicate the absence of the set pin.

```

model setb_rstb_flop (setb, rstb, ck, d, q, qb) (
    cell_type = dff;
    input (setb) (active_low_set)
    input (rstb) (active_low_reset)
    input (d) (data_in)
    input (ck) (posedge_clock)
    output (q_out) (data_out)
    output (QN) (data_out_inv)
    (
        primitive = _inv set_inv (setb, set_net);
        primitive = _inv reset_inv (rstb, reset_net);
        primitive = _dff lat (set_net , reset_net, ck, d, q, qb);
    )
)

```

D Latch

The keyword used to define a single or multiple port D latch is **_dlat**. It defines a latch where *both asynchs* and the clock input(s) are all *activeHI*. The syntax of the primitive attribute statement is as follows:

```
primitive = _dlat optional_inst_name(SET, RESET, CLK1, D1,  
                                     CLK2, D2, ..., CLKn, Dn, Q, QN);
```

This primitive allows users to define a D latch with a single pair or multiple pairs of clock and data inputs. The default behavior of a single port D latch is the same for FlexTest, Tessent FastScan, and Tessent TestKompress and is shown in the primitive table (Table 3-22).

Table 3-22. D Latch Primitive Table

Di	CLKi	SET	RESET	Q	QN
0	1	0	0	0	1
1	1	0	0	1	0
X	1	0	0	X	X
0/1/X/Z	0	0	0	Q	QN
1	1	0	1	0 ¹	1 ¹
				X ²	X ²
0/1/X/Z	0	0	1	0	1
0	1	0	1	0	1
0	1	1	0	1 ¹	0 ¹
				X ²	X ²
1	1	1	0	1	0
0/1/X/Z	0	1	0	1	0
0/1/X/Z	0/1/X/Z	1	1	X ²	X ²

1. Default value.

2. Value if you disable set/reset port dominance by setting the [set_simulation_options](#) command's -Set_reset_dominate_port switch to OFF.

The default behavior for FlexTest, Tessent FastScan, and Tessent TestKompress of a multiple port D latch is as follows:

1. If only one set, reset, or one of the clocks is active, Q and QN are well defined.
2. If more than one set, reset, or clock lines is active and if the values captured by the active clocks, set, or reset are the same, Q and QN are well defined. Otherwise, for Tessent FastScan and Tessent TestKompress, if the Set Simulation command's -set_reset_dominate_port is set to OFF, then Q and QN are unknown.

Note



You can use the [set_xclock_handling](#) command with the 'X' option to make the behavior of the `_dff` and `_dlat` primitives more pessimistic when the clock is set to an X value. This command has no effect on other primitives.

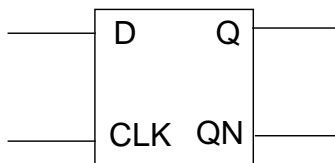
Examples:

```
model simple_active_high_lat (d, clk, q) (
    cell_type = latch;
    input (d) (data_in)
    input (clk) (active_high_clock)
    output (Q) (data_out)
    ( // Leave 1st two (set and reset) pins empty to indicate not present
      // Leave the last (QN) pin empty to indicate no QB
      primitive = _dlat ( , , clk, d, q, ) ;
    )
)

model simple_negedge_DFF (d, clk_b, q) (
    cell_type = latch;
    input (d) (data_in)
    input (clk_b) (active_low_clock)
    output (Q) (data_out)
    ( // All Mentor test primitives have activeHI asynchs and activeHI
      // or posedge clocks, so must invert cell input to model
      // activeLO or negedge clock.
      primitive = _inv (clk_b, clk_net);
      primitive = _dlat ( , , clk_net, d, q, );
    )
)

model active_high_latch (CLK, D, Q, QN) (
    cell_type = latch;
    input (D) (data_in)
    input (CLK) (active_high_clock)
    output (Q) (data_out)
    output (QN) (data_out_inv)
    (
        primitive = _dlat( , , CLK, D, Q, QN);
    )
)
```

Figure 3-30. D Latch



In the preceding example, the first two commas in the primitive statement are inserted because there is no set or reset pin in this D latch.

```
model setb_rstb_latch (setb, rstb, ck, d, q, qb) (
    cell_type = latch;
    input (setb) (active_low_set)
    input (rstb) (active_low_reset)
    input (d) (data_in)
    input (ck) (active_high_clock)
    output (q_out) (data_out)
    output (QN) (data_out_inv)
    (
        primitive = _inv set_inv (setb, set_net);
        primitive = _inv reset_inv (rstb, reset_net);
        primitive = _dlat lat (set_net , reset_net, ck, d, q, qb);
    )
)
```

One Time Unit Delay Element (FlexTest Only)

The keyword used to model one time-unit delay is **_delay**, and the syntax of the primitive attribute statement is as follows:

```
primitive = _delay optional_inst_name(IN, OUT);
```

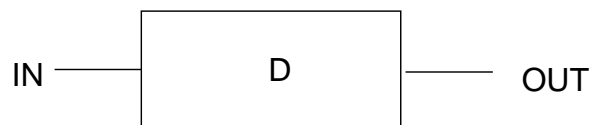
Table 3-23. DELAY Truth Table

IN (Previous State)	OUT
0	0
1	1

Example:

```
model DEL1(IN, OUT) (
    input(IN) ()
    output(OUT) ()
    (
        primitive = _delay(IN, OUT);
    )
)
```

Figure 3-31. One Time Unit Delay Element




Feedback Inverter

The primitive used to model a feedback inverter is **_invf**. The syntax of the primitive attribute statement is as follows:

```
primitive = _invf optional_inst_name(IN, OUT);
```

Table 3-24. INVF Truth Table

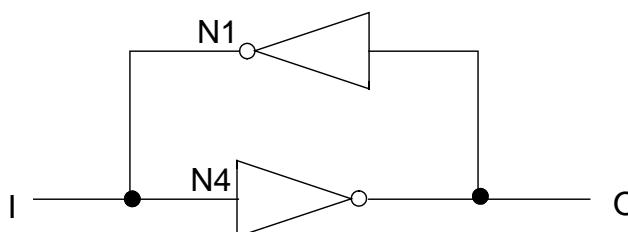
IN (Previous State)	OUT
0	Weak 1
1	Weak 0
X	Weak X
Z	Weak X

Note  The previous state of IN is X for Tessent FastScan and Tessent TestKompress. This primitive can only be used in a feedback path.

Example:

```
model INV_INVX(I, O) (
  input(I) ()
  output(O) ()
  (
    primitive = _wire(I, N1, N4);
    primitive = _invf(O, N1);
    primitive = _inv (N4, O);
  )
)
```

Figure 3-32. Feedback Inverter



XDET

The **_xdet** primitive is used to detect if the value on a pin is X. The syntax of the primitive attribute statement is as follows:

```
primitive = _xdet (IN, OUT);
```


The output OUT is high (1) when input IN is X. The output OUT is low (0) when the input IN is driven to a binary value or the high impedance state (Z) as shown in the truth table below:

Table 3-25. XDET Truth Table

IN	OUT
0/1/Z	0
X	1

Example:

```
model x-detector(RETN, detection_flag) (  
  input (RETN) ()  
  output (detection_flag) ()  
  (  
    primitive = _xdet(RETN, detection_flag);  
  )  
)
```

Wire Element

The primitive used to model signals wired together is **_wire**. The syntax of the primitive attribute statement is as follows:

```
primitive = _wire (IN0, IN1, ..., INn, OUT);
```

Table 3-26. WIRE Truth Table (for two inputs)

IN0/IN1	0	1	X	Z*	Weak 0	Weak 1
0	0	X	X	0	0	0
1	X	1	X	1	1	1
X	X	X	X	X	X	X
Z*	0	1	X	Z	0	1
Weak 0	0	1	X	0	0	X
Weak 1	0	1	X	1	X	1
* If there is a Z state at the input, then wire is treated as a bus.						

Note

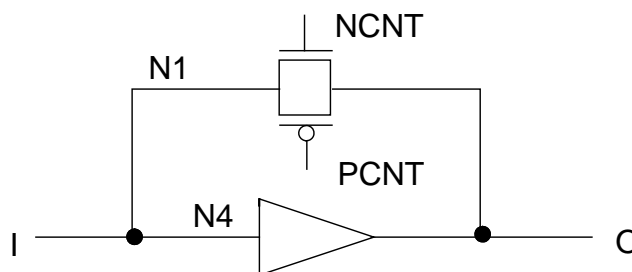


Even if an instance name is given, no-faults are placed on this primitive.

Example:


```
model MEM(I, O) (
  input(I) ()
  output(O) ()
  ( // For illustration only. Not advisable for test view.
    primitive = _wire(I, N1, N4);
    primitive = _tie1(NCNT);
    primitive = _tie0(PCNT);
    primitive = _rcmosf(O, NCNT, PCNT, N1);
    primitive = _buf(N4, O);
  )
)
```

Figure 3-33. Wire Element



Pull-Up or Pull-Down Device

The primitive used to model a pull-up or pull-down device, or any weak signal that must let a stronger signal fight (drive a different value) and win (the strong known value remains known) is **_pull**. It is placed at the output of the gate driving the weak signal onto the net where fighting occurs. For example, if a weak tie0 is driven through a switch onto a net where fighting occurs, the pull should be placed on the output of the switch, not the output of the _tie0 primitive, when modeling for ATPG simulation.

Note  A pull gate must be placed where its output drives the point of fighting with the stronger signal. ATPG simulations do not propagate weak signal strengths.

The syntax of the primitive attribute statement is as follows:

```
primitive = _pull (IN, OUT);
```

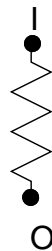
Table 3-27. PULL Truth Table

IN	OUT
1	Weak 1
0	Weak 0
X	Weak X
Z	Z

Example:

```
model PULLX(I, O) (  
    input(I) (  
        output(O) (  
            (  
                primitive = _pull(I, O);  
            )  
        )  
    )  
)
```

Figure 3-34. Pull-Up or Pull-Down Device



Note



The tools simulate pull gate transitions in one tester cycle. If you are modeling a very weak IO pad pullup/down which cannot pull up (down) in one tester cycle, leave the pull gate out, or use a `_tiex` as an input, so the ATPG tools do not predict a known value and cause simulation mismatches and/or tester failures.

Power Signal

The primitive used to model a power signal is `_tie1`. The syntax of the primitive attribute statement is as follows:

```
primitive = _tie1 (OUT);
```

Example:

```
model HOLD_RCMOSF(I, O) (  
  input(I) (  
    output(O) (  
      (  
        primitive = _wire(I, N1, N4);  
        primitive = _tie1(NCNT);  
        primitive = _tie0(PCNT);  
        primitive = _rcmosf(O, NCNT, PCNT, N1);  
        primitive = _buf(N4, O);  
      )  
    )  
  )  
)
```

The `_tie1` primitive is supported in macro descriptions. When writing out a netlist in Tessent Scan, this primitive will be converted to language-specific descriptions.

Ground Signal

The primitive used to model a ground signal is `_tie0`. The syntax of the primitive attribute statement is as follows:

```
primitive = _tie0 (OUT);
```

Example:

```
model HOLD_RCMOSF(I, O) (  
  input(I) (  
    output(O) (  
      (  
        primitive = _wire(I, N1, N4);  
        primitive = _tie1(NCNT);  
        primitive = _tie0(PCNT);  
        primitive = _rcmosf(O, NCNT, PCNT, N1);  
        primitive = _buf(N4, O);  
      )  
    )  
  )  
)
```

The `_tie0` primitive is supported in macro descriptions. When writing out a netlist in Tessent Scan, this primitive will be converted to language-specific descriptions. For example, `_tie0` will be converted to the `supply0` declaration in Verilog.

Unknown Signal

The primitive used to model an unknown signal is `_tiex`. The syntax of the primitive attribute statement is as follows:

```
primitive = _tiex (OUT);
```

Example:

```
model UN1(N, P, X) (  
    input(N, P) ()  
    output(X) ()  
    (  
        primitive = _xnor(N, P, N1);  
        primitive = _tiex(U);  
        primitive = _inv(P, Pnot);  
        primitive = _and (N, Pnot, U, N2);  
        primitive = _xor(N1, N2, N3);  
        primitive = _tshi(N, N3, X);  
    )  
)
```

High Impedance Signal

The primitive used to model a high impedance signal is `_tiez`. Typically, it is not needed in model descriptions. The main use occurs around IO pads because Verilog simulates an undriven (floating) net as Z, ATPG simulates it as X (by default, although that can be changed via runtime commands). Sometimes, the default X simulation causes IO pads with known Verilog simulations to produce X in ATPG simulations due to bus contention, or prevents patterns from being generated due to bus contention. Explicit use of a `_tiez` driver rather than a floating net can ensure that even the default ATPG simulation with no modification matches the Verilog.

The syntax of the primitive attribute statement is as follows:

```
primitive = _tiez (OUT);
```

Example:

```
model HIGHZ(Zout) (  
    output(Zout) ()  
    (  
        primitive = _tiez(Zout); // Zout is always Z/floating value.  
    )  
)
```

Undefined

The primitive used to model an undefined functional block is **_undefined**. The syntax of the primitive attribute statement is as follows:

```
primitive = _undefined optional_inst_name(IN0, IN1, ..., INn, OUT);
```

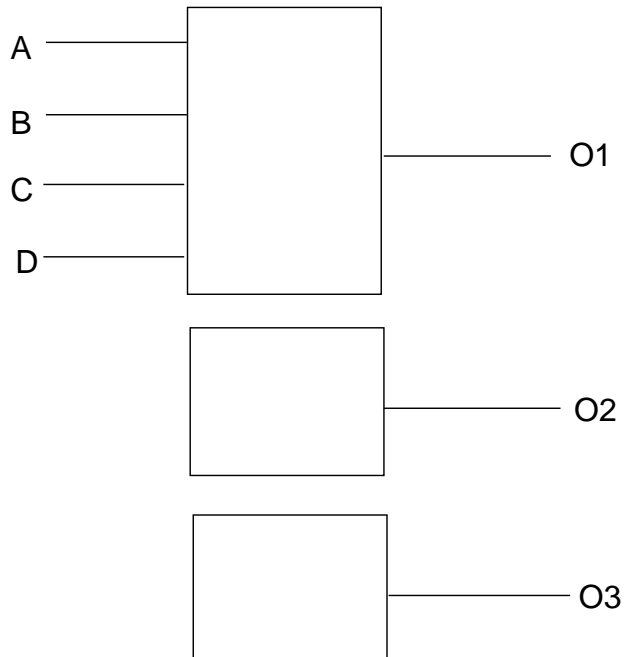
Table 3-28. UNDEFINED Truth Table

IN0	IN1	...	INn	OUT
0/1/X	0/1/X	...	0/1/X	X

Example:

```
model UNKNOWN1(A, B, C, D, O1, O2, O3) (
  input(A, B, C, D) ()
  output(O1) ()
  output(O2) ()
  output(O3) ()
  ( // _tiex has no inputs, so absorb all input using _undefined.
    primitive = _undefined(A, B, C, D, O1);
    primitive = _tiex(O2);
    primitive = _tiex(O3);
  )
)
```

Figure 3-35. Undefined Functional Block



Unidirectional NMOS Transistor

The primitive used to model a NMOS transistor is **_nmos**. The syntax of the primitive attribute statement is as follows:

```
primitive = _nmos optional_inst_name(I, EN, O);
```

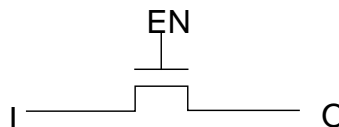
Table 3-29. NMOS Truth Table

I	EN	O
0/1/X/Z	0	Z
0	1	0
1	1	1
Z	1	Z
X	1	X
0/1/X/Z	X	X

Example:

```
model NMOS1(I, EN, O) (
    input(I, EN) ()
    output(O) ()
    (
        primitive = _nmos(I, EN, O);
    )
)
```

Figure 3-36. Unidirectional NMOS Transistor



Unidirectional PMOS Transistor

The primitive used to model a PMOS transistor is **_pmos**. The syntax of the primitive attribute statement is as follows:

```
primitive = _pmos optional_inst_name(I, EN, O);
```

Table 3-30. PMOS Truth Table

I	EN	O
0/1/X	1	Z
0	0	0

Table 3-30. PMOS Truth Table

I	EN	O
1	0	1
Z	0	Z
X	0	X
0/1/X	X	X

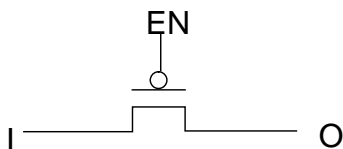
Example:

```

model PMOS1(I, EN, O) (
    input(I, EN) ()
    output(O) ()
    (
        primitive = _pmos(I, EN, O);
    )
)

```

Figure 3-37. Unidirectional PMOS Transistor



Unidirectional Resistive NMOS Transistor

The primitive used to model a resistive NMOS transistor is `_rnmos`. The syntax of the primitive attribute statement is as follows:

```
primitive = _rnmos optional_inst_name(I, EN, O);
```

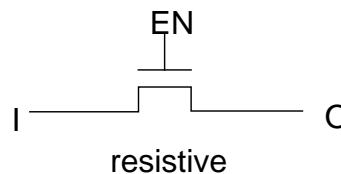
Table 3-31. RNMOS Truth Table

I	EN	O
0/1/X	0	Z
0	1	Weak 0
1	1	Weak 1
Z	1	Z
X	1	Weak X
0/1/X	X	Weak X

Example:

```
model RNMOS(I, EN, O) (
    input(I, EN) ()
    output(O) ()
    (
        primitive = _rnmos(I, EN, O);
    )
)
```

Figure 3-38. Unidirectional Resistive PMOS Transistor



Unidirectional Resistive PMOS Transistor

The primitive used to define a resistive PMOS transistor is **_rmos**. The syntax of the primitive attribute statement is as follows:

```
primitive = _rmos optional_inst_name(IN, PCNT, OUT);
```

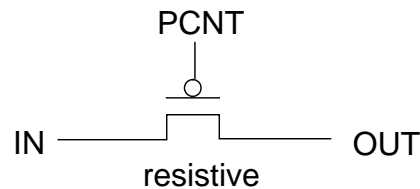
Table 3-32. RPMOS Truth Table

IN	PCNT	OUT
0/1/X	1	Z
0	0	Weak 0
1	0	Weak 1
Z	0	Z
X	0	Weak X
0/1/X	X	Weak X

Example:

```
model RPMOS1(IN, PCNT, OUT) (
    input(IN, PCNT) ()
    output(OUT) ()
    (
        primitive = _rmos(IN, PCNT, OUT);
    )
)
```

Figure 3-39. Unidirectional Resistive NMOS Transistor



Unidirectional Feedback NMOS Transistor

The primitive used to model a feedback NMOS transistor is `_nmosf`. The syntax of the primitive attribute statement is as follows:

```
primitive = _nmosf (I, CNT, O);
```

Table 3-33. NMOSF Truth Table

I (Previous State)	CNT	O
0/1/X	0	Z
0	1	Weak 0
1	1	Weak 1
Z	1	Z
X	1	Weak X
0/1/X	X	Weak X

Note

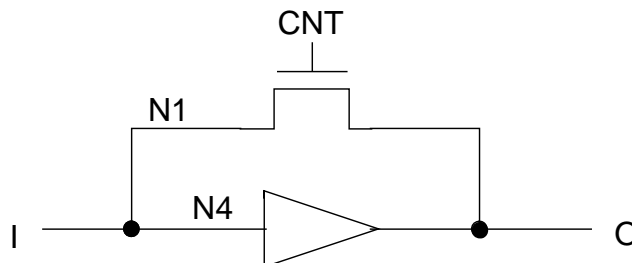


The previous state of “I” is X for Tessent FastScan and Tessent TestKompress. This primitive can only be used in a feedback path.

Example:

```
model HOLD_NMOSF(I, O) (
  input(I) ()
  output(O) ()
  (
    primitive = _wire(I, N1, N4);
    primitive = _nmosf(O, CNT, N1);
    primitive = _tie1(CNT);
    primitive = _buf (N4,O);
  )
)
```

Figure 3-40. Unidirectional Feedback NMOS Transistor



Unidirectional Feedback PMOS Transistor

The primitive used to model a feedback PMOS transistor is **_pmosf**. The syntax of the primitive attribute statement is as follows:

```
primitive = _pmosf optional_inst_name(I, CNT, O) ;
```

Table 3-34. PMOSF Truth Table

I (Previous State)	CNT	O
0/1/X	1	Z
0	0	Weak 0
1	0	Weak 1
Z	0	Z
X	0	Weak X
0/1/X	X	Weak X

Note

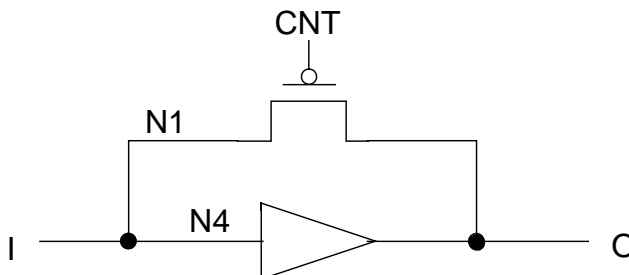


The previous state of “I” is X for Tessent FastScan and Tessent TestKompress. This primitive can only be used in a feedback path.

Example:

```
model HOLD_PMOSE(I, O) (
  input(I) ()
  output(O) ()
  (
    primitive = _wire(I, N1, N4);
    primitive = _pmosf(O, CNT, N1);
    primitive = _tie0(CNT);
    primitive = _buf (N4,O);
  )
)
```

Figure 3-41. Unidirectional Feedback PMOS Transistor



Unidirectional CMOS Transistor

The primitive used to model a CMOS transistor (which can be turned on when E is high or P is low) is **_cmos**. The syntax of the primitive attribute statement is as follows:

```
primitive = _cmos optional_inst_name(I, E, P, O);
```

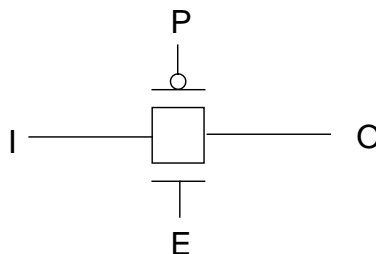
Table 3-35. CMOS Truth Table

I	E	P	O
0/1/X	0	1	Z
0	1	0/1/X	0
1	1	0/1/X	1
Z	1	0/1/X	Z
X	1	0/1/X	X
0	0/1/X	0	0
1	0/1/X	0	1
Z	0/1/X	0	Z
X	0/1/X	0	X
0/1/X	0	X	X
0/1/X	X	1	X

Example:

```
model CMOSX1(I, E, P, O) (
  input(I, E, P) ()
  output(O) ()
  (
    primitive = _cmos (I, E, P, O);
  )
)
```

Figure 3-42. Unidirectional CMOS Transistor



Unidirectional Resistive CMOS Transistor

The keyword used to define a resistive CMOS transistor (which can be turned on when E is high or P is low) is **_rcmos**, and the syntax of the primitive attribute statement is:

```
primitive = _rcmos optional_inst_name(I, E, P, O);
```

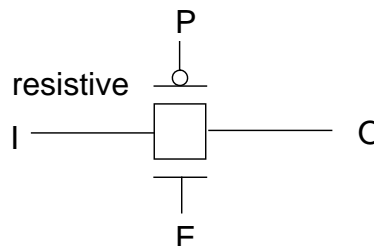
Table 3-36. RCMOS Truth Table

I	E	P	O
0/1/X	0	1	Z
0	1	0/1/X	Weak 0
1	1	0/1/X	Weak 1
Z	1	0/1/X	Z
X	1	0/1/X	Weak X
0	0/1/X	0	Weak 0
1	0/1/X	0	Weak 1
Z	0/1/X	0	Z
X	0/1/X	0	Weak X
0/1/X	0	X	Weak X
0/1/X	X	1	Weak X

Example:

```
model RMOSX1(I, E, P, O) (
    input(I, E, P) ()
    output(O) ()
    (
        primitive = _rcmos (I, E, P, O);
    )
)
```

Figure 3-43. Unidirectional Resistive CMOS Transistor



Unidirectional Resistive Feedback CMOS Transistor

The primitive used to model a resistive feedback CMOS transistor (which can be turned on when NCNT is high or PCNT is low) is **_rcmosf**, and the syntax of the primitive attribute statement is:

```
primitive = _rcmosf optional_inst_name(I, NCNT, PCNT, O);
```

Table 3-37. RCMOSF Truth Table

I (Previous State)	NCNT	PCNT	O
0/1/X	0	1	Z
0	1	0/1/X	Weak 0
1	1	0/1/X	Weak 1
Z	1	0/1/X	Z
X	1	0/1/X	Weak X
0	0/1/X	0	Weak 0
1	0/1/X	0	Weak 1
Z	0/1/X	0	Z
X	0/1/X	0	Weak X
0/1/X	0	X	Weak X
0/1/X	X	1	Weak X

Note



The previous state of I is X for Tessent FastScan and Tessent TestKompress. This primitive can only be used in a feedback path.

Example:

```
model HOLD_RCMOSF(I, O) (
  input(I) ()
  output(O) ()
  (
    primitive = _wire(I, N1, N4);
    primitive = _rcmosf(O, NCNT, PCNT, N1);
    primitive = _tie1(NCNT);
    primitive = _tie0(PCNT);
    primitive = _buf(N4, O);
  )
)
```

Pulse Generators with User Defined Timing

Tessent FastScan and Tessent TestKompress support pulse generators with multiple timed outputs. This is useful in cases when pulse generators have only a single output, and user-specified delay and width attributes allow multiple pulses with different effective timing to be generated. You can assume that the combinational delays of the circuit will be such that all paths which need to stabilize between different pulses from choppers will have time to stabilize. The syntax of the primitive attribute statement is:

```
primitive = _pulse_generator {delay, width} (clk_in,output);
```

- Delay and width variables are required attributes. The value of the delay must be in the range $0 \leq \text{delay} < 64K$ and the width must be in the range $1 \leq \text{width} < 64K$.

In the flattened data structure, for the primitive pulse generator:

```
primitive= _pulse_generator { 5, 10 } ( ck, a_clk )
```

the `report_gates` command displays the `pulse_generator` as:

```
command: report_gates -type pgen
/test PGEN
  IN      I
  OUT     O
  delay = 5 width = 10
```

The same checks will apply:

- Any sequential element can be clocked at most once in any cycle (C10).
- Intermediate values (from transparent capture cells) cannot be propagated to a PO (D11).
- Each sequential element has only a single state value, so it can only be captured by sinks that are either always clocked before or always clocked after the source. (not a mixture) (D10).

A limitation to this feature is that any clock pin driving pulse generators will not be able to be used in a clock procedure with other clocks. The output of a pulse generator must not propagate to a PO. (Any such PO will be classified as a clock PO. However, as the output of a pulse generator will be at X during clock PO pattern simulation, it is likely that some test coverage will be lost in this case.)

The output of a pulse generator must not connect to the input of a pulse generator through any path. The existing T17 rule will be used to cover this situation, too.

There can be no more than 31 unique events associated with the pulsing of any one clock pin. This means that after counting the rising and falling edge events of the clock, 29 additional discrete times may be used for rising and falling edge events generated from pulse generators.

For simulation of test procedures, a pulse generator outputs a 1 when there is a rising edge event at its input. The rising and falling edge events at the output of the pulse generator are scheduled to create events in the order defined by their delay and width. Additional simulation steps are generated to simulate the output changes of pulse generators. All internally generated events are stabilized before the next test procedure event is simulated and the time advanced. In this sense, the delay and width attributes are in units of deltas which are infinitesimally small compared to the time units used to define test procedures.

The input to a pulse generator at a binary value is required when all clock pins are in their inactive state, and constrained PIs are placed at the constrained value. In the event that the input value is a 1 under these conditions, the pulse generator is flagged with the “inactive-high” property, and the parallel pattern simulator will consider an input 0 to be the pulse-generating event.

There is the potential in this capability to significantly increase the amount of DRC simulation required, by creating many different edge times from different pulse generators. This is important when assigning delays and widths.

There is an increased risk that you may encounter scan chain tracing limitations using this capability, due to the ability to generate large numbers of different timed events from only a small number of shift procedure events. To work around this, additional workspace memory can be allocated.

In order to model the effect of timed outputs, Design Rules Checking identifies sequential elements as having transparent capture capability. DRC and simulation behave as if the outputs of the pulse generators are external clock pins, which are pulsed in sequence by a clock procedure.

RAM and ROM

Because the RAM and ROM primitives have some similar characteristics, they are combined into this subsection. However, a ROM is a subset of the functionality of a RAM. Because ROM is somewhat simpler than RAM, it is described first. The added complexities of RAM primitives are discussed following the description of ROM.

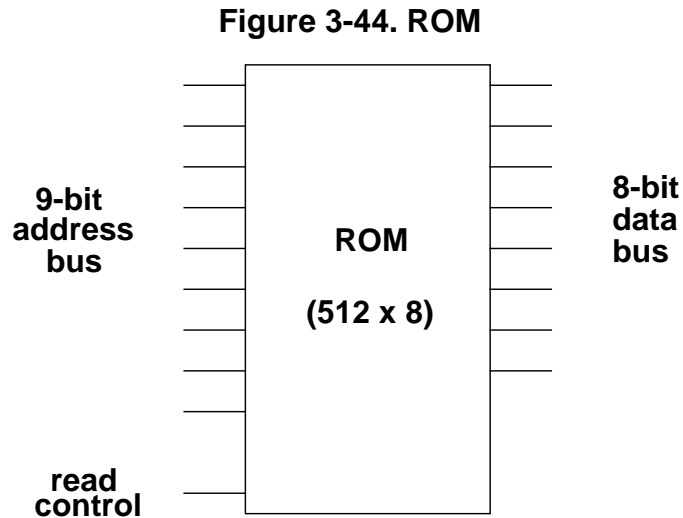
This section discusses RAM and ROM behavior and modeling concerns. For information on test strategies for RAM and ROM, refer to [“Testing RAM and ROM”](#) in the *Tessent Scan and ATPG User’s Manual*. For information on RAM rules checking, refer to [“RAM Rules \(A Rules\)”](#) in the *Tessent Shell User’s Manual*.

RAM and ROM Basics

A ROM is an array of memory cells whose contents are accessible through the activities of one or more read ports. Each of these read ports has an associated set of inputs. The set of inputs for each read port includes one or more read control lines, N read address lines, and M data output

lines. Each read port must have the same number of address lines, as well as the same number of data outputs.

Figure 3-44 shows a ROM.



Address lines identify which column of cells (set of values) to be placed on the data output lines. A ROM can store values into $((2^N) \times M)$ memory cells. M values at a time are placed on the outputs. The possible values you can place on the address lines are within the range of 0 to $((2^N)-1)$. The example in Figure 3-44 uses addresses in the range 0-511 and can access 512 8-bit words.

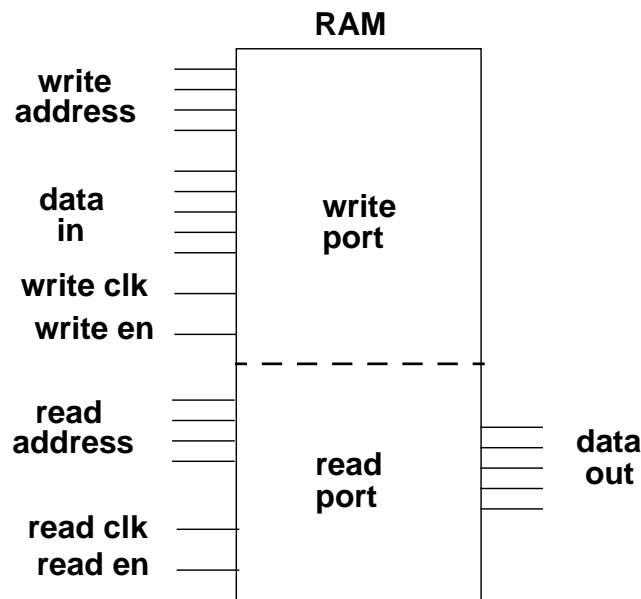
Before you read values from a ROM, the contents of the ROM must be initialized. This is accomplished through the use of a ROM initialization file. This is discussed in “[Basic ROM/RAM Rules Checking](#)” in the *Tessent Scan and ATPG User’s Manual*.

To turn on the read operation, activate the read control line(s). This places the value stored at the location specified by the address lines on the data output lines. When the read operation is off (not activated), X's are placed at the outputs, unless you specify a different behavior for the read off state, using the **read_off** attribute.

ROMs are modeled as strictly combinational gates; that is, they do not contain any sequential behavior. Two simulation gates, ROM and OUT, model the behavior of a ROM once the ROM model is flattened. ATPG simulation gates and model flattening are discussed in “[Model Flattening](#)” in the *Tessent Scan and ATPG User’s Manual*.

A RAM is similar to a ROM, with the addition of data write capabilities. Like a ROM, a RAM contains read ports and data output lines. However, it also contains write ports and data input lines. A RAM can have any number of read and write ports. Each port has its own separate inputs and outputs. All ports must have the same number of address lines, $A_1 \dots A_N$, and they must also have the same number of data lines, $D_1 \dots D_M$. Figure 3-45 shows a block diagram of a RAM.

Figure 3-45. RAM



The read operation of a RAM is identical to that of a ROM. However, to read a RAM value, you must first write a value to the specified location. To perform a write operation, you must place the proper address on the write address lines, place the proper data on the data in lines, and activate the write operation (typically, turn on write enable and pulse write clock). To model RAM behavior, the tools use RAM and OUT simulation gates in the flattened design. The flattened model may require additional gates, depending on how you define the output enable (oen) signal (see [page 183](#)). Often, oen is not needed to model the RAM of interest, so this input was not shown in [Figure 3-45](#).

RAM/ROM Library Primitives

This section discusses the library primitives used to model ROM and RAM. In each of the primitive descriptions that follow, the items inside the () denote pins that comprise the specified port. Additionally, within the **_cram** primitive, the items inside the {} denote optional port attributes. Read and write port behavior specified in the model description, is described in more detail in the next section.

ROM Library Primitive

The library primitive used to model ROM is **_rom**. The syntax of the primitive attribute statement is:

```
primitive = _rom (_read(REN, Aij, ..., Ai1, Ai0,
                      Dij, ..., Di1, Di0));
```

Note



The address and data line ordering specified from left to right must match the left to right ordering of the lines specified in the ROM init file. The DFT tools do not make any assumptions about ordering (for example, MSB to LSB).

Example:

```
model ROM2(Ren1, A1[2], A1[1], A1[0], D1[1], D1[0],
           Ren2, A2[2], A2[1], A2[0], D2[1], D2[0]) (
  input(Ren1, A1[2], A1[1], A1[0]) ()
  input(Ren2, A2[2], A2[1], A2[0]) ()
  output(D1[1], D1[0], D2[1], D2[0]) ( )
  (
    data_size = 2;
    address_size = 3;
    read_off = X;
    min_address = 0;
    max_address = 7;
    init_file = "rom.init_file";
    primitive = _rom(
      _read(Ren1, A1[2], A1[1], A1[0], D1[1], D1[0]),
      _read(Ren2, A2[2], A2[1], A2[0], D2[1], D2[0])
    );
  )
)
```

Or, you can model the same ROM using the **array** construct as follows:

```
model ROM2 (Ren1, A1, D1, Ren2, A2, D2) (
  input(Ren1, Ren2) ()
  input(A1,A2) (array = 2:0;)
  output(D1,D2) (array = 1:0;)
  (
    data_size = 2;
    address_size = 3;
    read_off = X;
    min_address = 0;
    max_address = 7;
    init_file = "rom.init_file";
    primitive = _rom(
      _read(Ren1,A1,D1),
      _read(Ren2,A2,D2)
    );
  )
)
```

This example shows a 2-port ROM with three address lines and two data lines. The read enable for the first port is named Ren1. The address lines for the first port, given with highest order first, are A1[2], A1[1], and A1[0]. The data lines for the first port are D1[0] and D1[1]. The read enable for the second port is named Ren2. Likewise, the address lines are A2[2], A2[1], and A2[0], and the data lines are D2[0] and D2[1].

When the read operation is off, X's are placed on the out gates. The addresses allowed on the address lines are in the range of 0 to 7. The initialization values to be placed on the ROM are found in a file called *rom.init_file* in the library directory.

The attributes **data_size** and **address_size** are required. The attributes **read_off**, **min_address**, **max_address**, and **init_file** are optional.

Basic RAM Library Primitive

The library primitive used to model simple RAM is **_ram**. The syntax of the primitive attribute statement is:

```
primitive = _ram (SET, RESET,
    _read(REN, An, ..., A1, A0, Dn, ..., D1, D0),
    _write(WEN, Aij, ..., Ai1, Ai0, Dij, ..., Di1, Di0))
```



Note

The address and data line ordering specified from left to right must match the left to right ordering of the lines specified in the RAM init file. The DFT tools do not make any assumptions about ordering (for example, MSB to LSB).

Example 1:

```
model RAM1(W1, A1[2], A1[1], A1[0], D1[2], D1[1], D1[0],
    R2, A2[2], A2[1], A2[0], D2[2], D2[1], D2[0]) (
    input(W1, A1[2], A1[1], A1[0], D1[2], D1[1], D1[0]) ()
    input(R2, A2[2], A2[1], A2[0]) ()
    output(D2[2], D2[1], D2[0]) ()
    (
        data_size = 3;
        address_size = 3;
        read_off = 0;
        min_address = 0;
        max_address = 7;
        edge_trigger = w;
        init_file = "ram.init_file";
        primitive = _ram(, ,
            _write(W1, A1[2], A1[1],
                A1[0], D1[2], D1[1], D1[0]),
            _read(R2, A2[2], A2[1],
                A2[0], D2[2], D2[1], D2[0])
        );
    )
)
```

This example shows a RAM gate with one write port and one read port, and no set or reset lines. The edge-triggered enable line of the write port is W1. The three-bit address includes lines A1[2], A1[1], and A1[0]. The three-bit data input includes lines D1[2], D1[1], and D1[0]. The address space is 0 to (2**3)-1.

The read port enable line is R2. The read port address lines are A2[2], A2[1], and A2[0]. The data out lines include D2[2], D2[1], and D2[0].

Example 2:

```

array_delimiter = "<>";

// RAM128 model
model ram128 (DOUT,ADD,CS,DIN,RD,WR) (
    input(ADD) (array = 7 : 0;)
    input(DIN) (array = 15 : 0;)
    input(CS, RD, WR) ()
    intern(DATAIN) (
        array = 15:0;
        primitive = _dlat D1 (,,RD,DIN<15>,DATAIN<15>,);
        primitive = _dlat D2 (,,RD,DIN<14>,DATAIN<14>,);
        primitive = _dlat D3 (,,RD,DIN<13>,DATAIN<13>,);
        primitive = _dlat D4 (,,RD,DIN<12>,DATAIN<12>,);
        primitive = _dlat D5 (,,RD,DIN<11>,DATAIN<11>,);
        primitive = _dlat D6 (,,RD,DIN<10>,DATAIN<10>,);
        primitive = _dlat D7 (,,RD,DIN<9>,DATAIN<9>,);
        primitive = _dlat D8 (,,RD,DIN<8>,DATAIN<8>,);
        primitive = _dlat D9 (,,RD,DIN<7>,DATAIN<7>,);
        primitive = _dlat D10 (,,RD,DIN<6>,DATAIN<6>,);
        primitive = _dlat D11 (,,RD,DIN<5>,DATAIN<5>,);
        primitive = _dlat D12 (,,RD,DIN<4>,DATAIN<4>,);
        primitive = _dlat D13 (,,RD,DIN<3>,DATAIN<3>,);
        primitive = _dlat D14 (,,RD,DIN<2>,DATAIN<2>,);
        primitive = _dlat D15 (,,RD,DIN<1>,DATAIN<1>,);
        primitive = _dlat D16 (,,RD,DIN<0>,DATAIN<0>,);
    )
    intern(WR_CS) (
        primitive = _and AN1 (CS,WR,WR_CS);
    )
    output(DOUT) ( array = 15:0;
        min_address = 0;
        max_address = 128;
        data_size = 16;
        address_size = 8;
        primitive = _ram RAM1 (,,
            _read(CS,ADD,DOUT),
            _write(WR_CS,ADD,DATAIN));
    )
)

```

Comprehensive RAM Primitive

The primitive used to model complex RAM (CRAM) reading and writing capabilities is **cram**. The syntax of the primitive attribute statement is:

```

primitive = _cram (SET, RESET,
    _read{w,x,y,z} (oen,rclk,ren,address,out_data)
    _write{x,y,z} (wclk,wen,address,in_data))

```

The **cram** primitive may have zero or more write ports and one or more read ports. If it has no write ports, the tool treats the CRAM as a read-only CRAM during test pattern generation. The port types are described in more detail in the section, “[Attributes of RAM/ROM Primitives](#)” on page 177.

The content of the CRAM is provided through an initialization file. See “[Initialization Files for RAM and ROM](#)” on page 182 for more information.

Example CRAM:

```
model CRAM1(Wclk1, WA1[2], WA1[1], WA1[0],
            Din1[2], Din1[1], Din1[0],
            Rclk1, RA1[2], RA1[1], RA1[0],
            Dout1[2], Dout1[1], Dout1[0],
            REN1, WEN1) (
    input(Wclk1, WA1[2], WA1[1], WA1[0],
          Din1[2], Din1[1], Din1[0]) ()
    input(Rclk1, RA1[2], RA1[1], RA1[0], REN1, WEN1) ()
    output(Dout1[2], Dout1[1], Dout1[0]) ()
    (
        edge_trigger = r;
        data_size = 3;
        address_size = 3;
        read_off = h;
        min_address = 0;
        max_address = 7;
        init_file = "ram.init_file";
        primitive = _cram(, ,
            _write{,,} (Wclk1, WEN1, WA1[2], WA1[1], WA1[0],
                       Din1[2], Din1[1], Din1[0]),
            _read{,H,H,H} (, Rclk1, REN1, RA1[2], RA1[1], RA1[0],
                           D2[2], D2[1], D2[0])
        );
    ) )
```

Example CRAM modeled using the **array** construct:

```
model CRAM1(Wclk1, WA1, Din1, Rclk1, RA1, REN1, WEN1, Dout1) (
    input (Wclk1, Rclk1, REN1, WEN1) ()
    input (WA1,RA1) (array = 2:0;)
    input (Din1) (array = 2:0;)
    output (Dout1) (array = 2:0) ;
    (
        edge_trigger = r;
        data_size = 3;
        address_size = 3;
        read_off = h;
        min_address = 0;
        max_address = 7;
        init_file = "ram.init_file";
        primitive = _cram (, ,
            _write{,,} (Wclk1, WEN1, WA1, Din1),
            _read{,H,H,H} (,Rclk1, REN1, RA1, Dout1)
        );
    )
)
```

Example CRAM with multiple ports:

```
model 2_port_ram (Clk1, WEN1, A1, Din1, Dout1,
                  Clk2, WEN2, A2, Din2, Dout2) (
```

```

input (Clk1, WEN1, Clk2, WEN2) ()
input (A1, A2) (array = 3:0;)
input (Din1, Din2) (array = 3:0;)
output (Dout1, Dout2) (array = 3:0;)
(
    primitive = _inv I1 (WEN1, REN1);
    primitive = _inv I2 (WEN2, REN2);
    // Define the ram using the Comprehensive Ram primitive & statements.
    data_size = 4;
    address_size = 4;
    min_address = 0;
    max_address = 15;
    edge_trigger = rw;
    read_write_conflict = xw;

    // Usually, rams do not have set or reset so leave off 1st two pins.
    primitive = _cram 2_port_cram ( , ,
        //port 1
            _write{,,} (Clk1, WEN1, A1, Din1),
            _read{,H,H,H} (,Clk1, REN1, A1, Dout1),
        //port 2
            _write{,,} (Clk2, WEN2, A2, Din2),
            _read{,H,H,H} (,Clk2, REN2, A2, Dout2)
    ); // end primitive port list

) // end hardware section
) // end model

```

Example CRAM with write port definition removed to make it read-only:

```

model READ_ONLY_CRAM1(Wclk1, WA1, Din1, Rclk1, RA1, REN1, WEN1, Dout1) (
    input (Wclk1, Rclk1, REN1, WEN1) ()
    input (WA1,RA1) (array = 2:0;)
    input (Din1) (array = 2:0;)
    output (Dout1) (array = 2:0)
    (
        edge_trigger = r;
        data_size = 3;
        address_size = 3;
        read_off = h;
        min_address = 0;
        max_address = 7;
        init_file = "ram.init_file";
        primitive = _cram ( , ,
            _read{,H,H,H} (, Rclk1, REN1, RA1, Dout1)
        );
    )
)

```

Another way to create a read-only CRAM is to tie the write ports of a regular CRAM to the inactive state. To be usable for test generation in Tessent FastScan or Tessent TestKompress, a read-only CRAM must meet the following requirements:

- The contents of the CRAM cannot be disturbed by any test procedures except test setup.

- The set and reset ports (and write ports when defined) must be inactive and stable during capture.
- The CRAM model in the ATPG library must define a valid initialization file. If the model does not define an initialization file, the tool will treat the CRAM as TIEX for ATPG, which will lower test coverage.

Note

Tessent FastScan and Tessent TestKompress are the only tools that support read-only CRAMs for test generation.

Attributes of RAM/ROM Primitives

The following attributes may be used within the RAM and ROM model descriptions:

- **data_size = number;**
This required attribute specifies the width of the data outputs.
- **address_size = number;**
This required attribute specifies the width of the address inputs.
- **primitive = [_rom | _ram | _cram];**
This required attribute specifies the library primitive used by the RAM or ROM being defined.
- **array = start_number: end_number;**
This optional attribute specifies the width of wide address or data pins.
- **min_address = number;**
This optional attribute specifies the minimum valid address. The default is 0.
- **max_address = number;**
This optional attribute specifies the maximum valid address. The default is $(2^{**}\text{address_size}) - 1$.
- **read_off = [0 | 1 | X | H];**
This optional attribute specifies the data output values if the read enable line is off. The options are 0, 1, hold, or X, which is the default. For the **_rom** primitive, this value must be X. For the **_cram** primitive, this attribute does not apply because the X, Y, Z attributes specify its read_off behavior. For more information on requirements of the read_off attribute relative to the **_ram** primitive, refer to the [“Basic ROM/RAM Rules Checking”](#) section of the *Tessent Scan and ATPG User’s Manual*.
- **init_file = “file_name”;**
This optional attribute specifies the file, in Mentor Graphics modelfile format, defining initial memory values. Specify the full path name to the initialization file if the file is not located in the directory from where you invoked Tessent FastScan or Tessent TestKompress.

- **edge_trigger = [R | W | RW];**

This optional attribute specifies the edge trigger values of the read and/or write lines. R indicates the read lines are positive edge-triggered whereas W indicates the write lines are positive edge-triggered. RW indicates both are positive edge-triggered. The default is neither read nor write are positive edge-triggered.

Note



The **_rom** primitive does not support this attribute.

For the **_cram** primitive, only the 1st control (the read clock for **_read** ports and the write clock for **_write** ports) can be edge-triggered. The **_cram** read and write enables, if they are used, are always level sensitive.

- **address_type = <encode|decode>;**

This optional attribute is used only for the **_cram** primitive to specify whether the address lines are encoded or decoded. Encoded is the default.

Note



The **_cram** primitive does not support decoded address buses for FlexTest.

- **read_read_conflict = [R|X];**

This optional attribute specifies the behavior when two or more **_read** ports are active on the same address at the same time. If this attribute is set to R, the normal read is carried out. If the attribute is set to an X, X is placed at the outputs. R is the default.

Note



FlexTest does not support this attribute because it simulates each read operation. Thus, its behavior for this case is “R”, regardless of the value of the addresses.

- **read_write_conflict = [NW|XW|OW|XX|OX];**

This optional attribute specifies the behavior when a **_read** and a **_write** port are both active on the same address. If the address is different, the normal read/write operations are performed. If the address is the same, simulation is defined by the value of the attribute. The first character defines how the read is performed, and the second character defines how the write is performed. N=new, O=old, X=x values, and W= normal write operation. NW is the default. For example, if the attribute is set to NW, then the new value is read and the new value is written.

Note



FlexTest always does “NW” independent of addresses; that is, it does not support XW|OW|XX|OX.

- **write_contention = [true|false];**
This optional attribute specifies the behavior when two or more **_write** ports are active at the same time. If set to true, all (independent of address) multiple writes are prohibited by this attribute. False is the default.
- **overwrite = [true|false];**
This optional attribute is used only if the **write_contention** attribute is not set to true. This attribute defines the behavior when multiple ports are writing to the same address. If set to true, and if the addresses are different, both writes are carried out. If the address is the same, precedence is given to the last port defined in the model (data at the other write port is completely ignored). False is the default.

If set to false, and if the addresses are different, all writes are carried out. If the address is the same, the write that is performed depends on the data at the active write ports. If data differs at the active ports, an X is written. Otherwise, the same data is at all ports, so it is written to them all. For this attribute, Tessent FastScan, FlexTest, and Tessent TestKompress exhibit the same behavior.
- **write_write_conflict = {same_address_bit_data_consensus |
same_address_port_data_consensus |
same_address_x_port |
always_x_words |
prohibit_multiple_writes};**

This optional attribute is used for the RAM primitive to control multiple write ports writing to the same address. The attribute allows more precise matching of Verilog modeling pessimism. The **write_write_conflict** attribute has the following options:

- **same_address_bit_data_consensus**

An optional literal that specifies that if multiple write ports are writing to the same address, and the write ports have different values, then the RAM writes X for each bit that disagrees. If all write ports have the same value, then the RAM writes that value. If not writing to the same address, the RAM writes normally. This is the default.
- **same_address_port_data_consensus**

An optional literal that specifies that if multiple write ports are writing to the same address, and any bits in the write ports disagree, then the RAM writes X for all bits of the write ports. If all bits in the write ports agree, then the RAM writes that value. If not writing to same address, the RAM writes normally.
- **same_address_x_port**

An optional literal that specifies that if multiple write ports are writing to the same address, the RAM writes X for all bits of the ports. If not writing to same address, the RAM writes normally.
- **always_x_words**

An optional literal that specifies to always write all bits X when multiple ports write.

- o `prohibit_multiple_writes`

An optional literal that specifies to not create a pattern that writes to multiple ports with the same address simultaneously.

Note



`write_write_conflict` is a new attribute that replaces `write_contention` and `overwrite`. `write_contention` and `overwrite` are allowed for backward compatibility, but you cannot use either of them at the same time as `write_write_conflict`.

Examples of the `write_write_conflict` Attribute

The following examples show the results of using the various options for the `write_write_conflict` attribute.

Example 1

`_write` port A is writing data 10X0 to an address (word)

`_write` port B is writing data 11X0 to same address (word)

Option	Result	Notes
<code>same_address_bit_data_consensus</code>	1XX0	If writing to same word (address), 1XX0 written
<code>same_address_port_data_consensus</code>	XXXX	If writing to same word (address), XXXX written
<code>same_address_x_port</code>	XXXX	If writing to same word (address), XXXX written
<code>always_x_words</code>	XXXX	Always write all bits X if multiple writes

Example 2

`_write` port A is writing data 1010 to an address (word)

`_write` port B is writing data 1010 to same address (word)

Option	Result	Notes
<code>same_address_bit_data_consensus</code>	1010	If writing to same word (address), 1010 written
<code>same_address_port_data_consensus</code>	1010	If writing to same word (address), 1010 written
<code>same_address_x_port</code>	XXXX	If writing to same word (address), X all bits
<code>always_x_words</code>	XXXX	Always write all bits X if multiple writes

Example 3

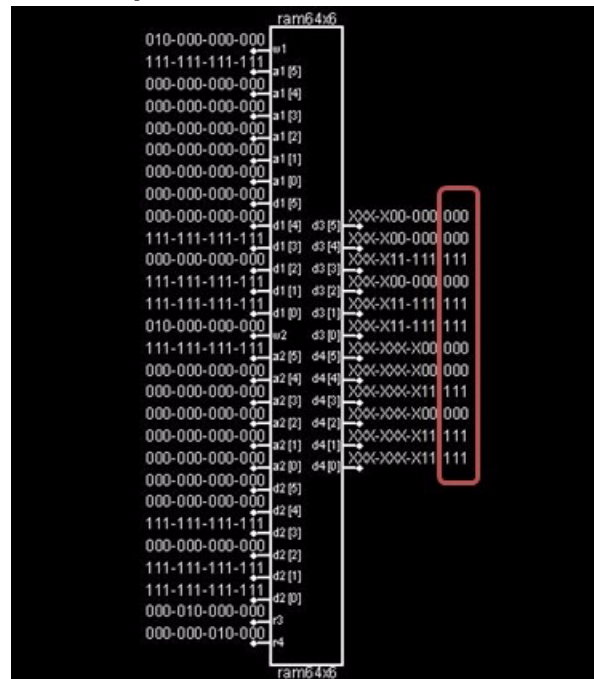
_write port A writing data 1010 to an address (word) A
_write port B writing data 0101 to different address (word) B)

Option	Result
same_address_bit_data_consensus	1010 written to address A, 0101 to address B
same_address_port_data_consensus	1010 written to address A, 0101 to address B
same_address_x_port	1010 written to address A, 0101 to address B
always_x_words	XXXX written to both address A and address B

Example 4

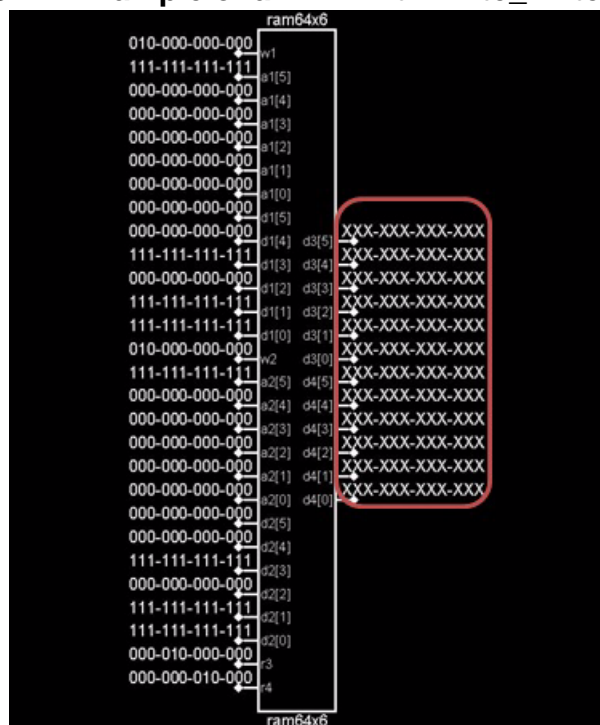
Figure 3-46 and Figure 3-47 below illustrate the effect of using the write_write_conflict attribute.

Figure 3-46. Example of a RAM without write_write_conflict



Notice that the output data ports for this RAM have known (binary) values for the read cycle in Cycle 4.

Figure 3-47. Example of a RAM with write_write_conflict



Notice that the output data ports for this RAM are masked for all cycles when the attribute `write_write_conflict` is set to `same_address_x_port`.

Initialization Files for RAM and ROM

An initialization file may be used to define the initial values of the memory cells of the RAM and ROM. The supported format of this file is the Mentor Graphics ROM/RAM modelfile format. For a detailed description of this format, refer to the [read_modelfile](#) command in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*. After you create the modelfile, you use an `init_file` attribute within the RAM or ROM model description to specify the file. Alternatively, you can use the `read_modelfile` command to read in the initialization file.

ROM and RAM Port Behavior

This section describes the port behaviors for the `_rom`, `_ram`, and `_cram` library primitives.

Read Port Behavior for `_rom` and `_ram`

You use a `_read` keyword for each read port of the ROM or RAM. Each read port contains an ordered list of pins separated by commas. If you omit a pin, you must still specify the comma delimiter. When you define the pins, the read control line(s) must be first, followed by the address lines, and then the data out lines.

The xclock handling does not affect ram or rom primitive simulation, even if the primitive is edge-triggered. This command affects `_dff` and `_lat` primitives only.

The read enable line is optional for ROM. If it is not defined, it is assumed that the port is always reading. If the read enable is defined, by default it behaves as follows. It is assumed to be active high. When the read enable line is active, the values of the memory cells associated with the current port address are placed on the data outputs--if the address is valid. If the current address is invalid, all outputs of the port are set to X. Additionally, when either the read enable line is at X or the read enable line is active and any address line is at an X state, all outputs of the port are set to X. If the read enable line is low (inactive), all outputs of the port are set to X. You can change some of this default behavior by using attributes in the RAM or ROM model description. For example, you can change the behavior of the ROM when reading is inactive by using the **read_off** attribute.

X handling for ram and rom primitives does not depend on triggering. The level and edge-triggered ports have the same X behavior.

The number of address lines in each port must be equal to the number specified by the **address_size** attribute. The address lines must be ordered so that the most significant address lines are given first. The number of data lines in each port must be equal to the number specified by the **data_size** attributes. The data lines must be ordered so that the first data input line corresponds to the first data output line, and so on. The data line ordering must also be consistent with the data ordering specified in the initialization file.

You can use the **edge_trigger** attribute to specify that the read lines of all RAM read ports are edge-triggered. This specifies for the RAM to only read during the rising edge of an edge-triggered read line. RAM with edge-triggered read lines must also set the value of the **read_off** attribute to hold. This indicates the read port is capable of holding the values at its outputs when the read line is off. Failure to satisfy this condition results in an error condition during design flattening.

You cannot use the **edge_trigger** attribute with ROMs; an error condition results during design flattening.

Read Port Behavior for `_cram`

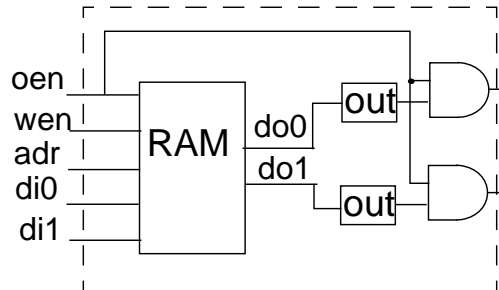
```
_read{w,x,y,z}(oen,rclk,ren,address,out_data)
```

Each read port of a **_cram** can have up to five pins. The first three are the control pins, which are described in the following list:

- **oen** - This is the output enable signal that is used to control accessibility of the `_cram` output. If the signal is high, the output is accessible. Otherwise, the output is disabled. You can assign a value to this signal using the **w** attribute that is within the `{ }` of the **_read** statement. The choices are 0, 1, X (default), Z, or H (hold previous value).

If you specify 0, the tool adds AND gates after each OUT gate in the flattened model, as Figure 3-48 shows.

Figure 3-48. Flattened RAM Model with oen Set to 0



Likewise, if you specify 1, X, Z, or H, the tool adds OR, MUXed TieX, tri-stateable driver, or D latch gates, respectively.

- **rclk** - This is the read clock, which is the signal that activates reading of the RAM data. You can use the **edge_trigger** attribute to specify whether the signal is edge-triggered or level sensitive. You must specify this clock pin if the signal is edge-triggered or if you specified the read enable pin. If you do not specify this signal, the default behavior is always active.
- **ren** - This is the read enable, a signal which can also activate reading of the RAM data. If the RAM has only one signal that activates reading, you must specify this signal as a read clock pin (rclk).

The read enable pin is assumed to be level sensitive. If you do not specify this pin, the default behavior is always active.

Normally, the RAM data is accessible when both the read enable and read clock signals are active. You can use the **x**, **y**, and **z** attributes within the { } of the **_read** statement to specify the desired behavior when either or both of these signals are inactive. The **x** attribute specifies the behavior when both are inactive, the **y** attribute specifies the behavior when only ren is inactive, and the **z** attribute specifies the behavior when only rclk is inactive. The choices for behavior of the read port values are 0, 1, X, H (hold previous values), H1 (hold previous values for one clock cycle, then become X), and PR (possible read, outputs that would change if a read were done are set to X).

Note



FlexTest does not support the H1 and PR options.

Set and Reset Lines for `_ram` and `_cram`

The `_ram` and `_cram` primitives may have a set and/or reset input that is active high. If the set line is high, all the memory cells of the RAM are set to 1. If the reset line is high, all the memory

cells of the RAM are set to 0. If either the set or reset input is at an X, or if both are high, all the memory cells of the RAM are set to X. If the set or reset lines are not used, the comma delimiters must still be inserted in the primitive definition.

Write Port Behavior for `_ram`

Each **`_write`** port contains an ordered list of pins. When you define the pins, you must specify the write enable first, followed by the address lines, and then the data in lines.

The `xclock` handling does not affect ram primitive simulation, even if the primitive is edge-triggered. This command affects `_dff` and `_lat` primitives only.

By default, the behavior of the RAM write port is as follows. The write enable line is active high. When the write enable line is active, the memory location specified by the current port address is loaded with data present on the data in lines--if the address is valid. If the address is invalid, the write operation is ignored. When the write enable line is at X or the write enable line is active and any address line is at an X state, Tessent FastScan and Tessent TestKompress set the memory cells that will be accessed by the address to X. FlexTest does the same if the input data value differs from the memory cell of the RAM. If the input data and the memory cell value are the same, the memory cell value will not be changed.

X handling for the ram primitive does not depend on triggering. The level and edge-triggered ports have the same X behavior.

When multiple write ports are active at the same time, and they attempt to write conflicting values to the same memory cell, those memory cells are set to X (unless the **`overwrite`** attribute is used). The **`overwrite`** attribute gives precedence to the last **`_write`** port defined within the **`_ram`** primitive.

The number of address lines in each port must be equal to the number specified by the **`address_size`** attribute. The address lines must be ordered so that the most significant address lines are given first. The number of data lines in each port must be equal to the number specified by the **`data_size`** attributes. The data lines must be ordered so that the first data in line corresponds to the first data out line, and so on. The data line ordering must also be consistent with the data ordering specified in the initialization file.

You can use the **`edge_trigger`** attribute to specify that the write lines of all write ports are edge-triggered. This specifies for the RAM to only write during the rising edge of an edge-triggered write line. For RAMs with edge-triggered write lines, the following rules apply:

- Static pass-through testing is not allowed.
- The RAM must successfully pass design rule A1 in order to be used during ATPG or fault simulation. Otherwise, it is treated as a tie-X gate.
- Patterns pulse the write control line after forcing the primary inputs to make sure the address and data in inputs are stable.

Write Port Behavior for `_cram`

```
_write{x,y,z} (wclk,wen,address,in_data)
```

You use a **`_write`** keyword for each write port of the **`_cram`**. The pin list for a write port contains four pins separated by commas. If you omit a pin, you must still specify the comma delimiter. The first two are the control pins, which are described in the following list:

- **wclk** - This signal, which is *not* optional, activates writing to the RAM. You can use the **edge_trigger** attribute to specify whether the signal is edge-triggered or level sensitive.
- **wen** - This signal, which is assumed to be level sensitive, also activates writing. If not specified, the default value is active.

When both the write enable and write clock signals are active, the normal write operation is performed. Additionally, you can specify the behavior when either or both of these signals are inactive by using the **x**, **y**, and **z** attributes. The **x** attribute specifies the behavior when both are inactive; the **y** attribute specifies the behavior when wen is inactive; and the **z** attribute specifies the behavior when wclk is inactive. The choices for cell values are 0, 1, X, H (contents not changed, the default), and PW (possible write--cells which would change if a write were done are set to X).

The xclock handling does not affect cram primitive simulation, even if the primitive is edge-triggered. This command affects `_dff` and `_lat` primitives only.

It is possible to change the simulation behavior of RAM models with data hold capability. In cases where it is required to model a RAM, which has data hold capability that does not introduce latency, you can use the `add_capture_handling` command to define a data-hold RAM as a source of new data -- this will indicate that latency is to be removed. For more information, see the [add_capture_handling](#) command description in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

X handling for the cram primitive does not depend on triggering. The level and edge-triggered ports have the same X behavior.

An X on a `_cram` write port control, clock or enable, will X the current address[es] even if the other control is off/0, and even if the triggering is edge. An X on an address bit covers both 0 and 1 values, so at least two rows are Xd if any encoded write address bit and any write control bit is X.

Read_Write Port Behavior for `_cram`

You can use the `_read_write` port primitive, if a read port and a write port have the same address and data lines. The primitive is defined as follows:

```
_read_write {rw, rx, ry, rz, wx, wy, wz} (oen, rclk, ren, wclk, wen, address, data);
```

Here, rw, rx, ry, and rz are attributes used to specify the read port behavior as described in `_read` port. However, rw (the attribute for specifying the behavior of output enable) has a different default value if it is not specified, and the default is Z, which is the only legal value for rw. The wx, wy, and wz are attributes used to specify the write port behavior.

The first five pins of the `_read_write` port are output enable (oen), read clock (rclk), read enable (ren), write clock (wclk), and write enable (wen). The order is significant. Also, the output enable pin must be specified in the `_read_write` port.

The behavior of the port will be to allow either read or write in each cycle (not both), but it will not be possible to perform any form of passthru test using the RAM. In the case that multiple RAMs share a common data bus, it will not be possible to transfer data from one RAM to another using the bus.

Note

Tessent FastScan and Tessent TestKompress will report an A5 rule violation in this case.

The xclock handling does not affect cram primitive simulation, even if the primitive is edge-triggered. This command affects `_dff` and `_lat` primitives only.

Provided contention checking is performed; there will be no danger of creating an incorrect pattern, although a certain amount of pessimism will be introduced into the simulation. In order to support a bidirectional pin, exceptional behavior will be required in flattening.

For a read/write port, a read write conflict on the same port will always be treated as read X, write X. This is independent of the attribute controlling conflicts between the other ports of the `_cram`.

X handling for the cram primitive does not depend on triggering. The level and edge-triggered ports have the same X behavior.

An example of a ram model that uses `_read_write` port is shown below:

```
model RAM1 (W1,A1,R2,D1) (  
    input (W1,R2) ()  
    input (A1) (array = 4:0;)  
    inout (D1) (array = 0:4)  
    (  
        data_size = 5;  
        address_size = 5;  
        min_address = 0;  
        max_address = 31;  
        primitive = _cram( , ,  
            _read_write (R2,R2,,W1,,A1,D1) // Single read port  
        ); // end primitive statement  
    ) // end hardware section  
    ) // end model
```

DRC for RAMs

Edge-triggered ports: DRC will recognize the case where a RAM port is stable due to having an edge-triggered clock. This supports using opposite edges of the same signal to clock multiple ports of the same RAM. This is not supported for level sensitive ports.

RAM sequential patterns require that a RAM be kept stable across multiple scan load operations. (for example, no write can occur during scan shift). Further, if a RAM has data hold capability at its read port, the read port must also not be clocked during scan shift. These requirements are also checked by existing DRCs.

ROM Limitations

The following restrictions apply to ROM modeling:

- The **_rom** primitive does not support the **edge_trigger** attribute.
- The **_rom** primitive only supports the **read_off** attribute value of X.

RAM Limitations

To simplify the ATPG process, there are two restrictions that should have an insignificant impact on the test coverage.

1. If there is a read operation requirement at a RAM, all of its write operations must be at its off-state. This restriction reduces the efforts to make sure what is read will not be overwritten at the same time during the ATPG process. However, if there is a write operation requirement, read operation can be at any state. This allows us to do ATPG for a RAM whose read enable lines are always active.
2. If there is a write operation requirement at one port, all the write operations of other ports must be at their off-states. This restriction reduces the efforts to make sure what is written at one port will not be overwritten by another port at the same time, during the ATPG process.

Chapter 4

Using LibComp to Create Tessent Simulation Models

This chapter describes how to create Tessent simulation models using LibComp. LibComp translates Verilog modules into simulation models for use with Tessent Scan, Tessent FastScan, FlexTest, and Tessent TestKompress. This chapter includes the following topics:

Creating Simulation Models	190
Finding Unsupported Constructs in Partial Models	190
Finding Black Boxes with Vectored Outputs	191
LibComp Command Summary	193
LibComp Command Descriptions	195
UDP Limitations and Examples	223
I/O Pad Limitations and Examples	231
Memory Limitations and Examples.....	234

Creating Simulation Models

To create simulation models, invoke the LibComp tool on the Verilog source library using the default dofile provided by LibComp. For example:

```
Tessent_Tree_Path/bin/libcomp verilog_source -dofile -log log_file
```

For more information on the invocation arguments, see the [libcomp](#) shell command.

To get a quick reminder of the invocation arguments for the **libcomp** shell command before running it, enter the following on the command line:

```
libcomp -help
```

Finding Unsupported Constructs in Partial Models

When unsupported constructs are encountered, the LibComp tool continues to translate the supported pieces of the module into a partial model. Unsupported constructs contain a “not translated/supported” message in the partial model. You can then locate the unsupported construct and edit the partial model as needed to complete it.

To find the unsupported constructs within partial models:

1. Invoke the LibComp tool on the Verilog library and create a logfile. For example:

```
Tessent_Tree_Path/bin/libcomp library_path -log log_file -replace
```

The library is translated and a log file detailing the output including unsupported constructs is generated.

2. From the UNIX/Linux command line, search the logfile for the unsupported message:

```
grep "not translated/supported." my.log
```

Each instance of the “not translated/supported” warning message in the logfile is displayed in the module name and the line number of the unsupported construct.

If a simulation model was partially translated, the following message is displayed in the model:

```
"BlackBox"  
or
```

```
"EDIT & place _cram" ...
```

Also, the following message is displayed above the model (unless it was completed black boxed):

```
"PARTIALLY TRANSLATED MODULE"
```

Finding Black Boxes with Vectored Outputs

The LibComp tool leaves vectored outputs on black boxes undriven. These undriven outputs default to `_tiex` in ATPG runs. In some cases, partial models with only some outputs black boxed may result. Black boxes with vectored outputs are preceded with a “Blackboxed PO” message in the ATPG library.

Use the following command to display a list of the black boxed models with vectored outputs in an ATPG library:

```
grep "Blackboxed PO." output.atpglib
```

Reconciling System Verilog reg and Verilog Keyword Compiling Issues

You can encounter ModelSim compilation and loading errors with LibComp and lcVerify output under the following conditions:

- **System Verilog** — Modules using `reg` as interconnect. LibComp does not support System Verilog syntax or semantics except for using `reg` as interconnect, which the tool allows and handles correctly.
- **Verilog** — Source file contains Verilog reserved keywords, for example `int` or `do`.

You reconcile these either of these issues by using the `-sv` or `-no_sv` switch with one of the following methods:

- The [Set Verification](#) command.
- The [libcomp](#) invocation.
- The [leverify](#) invocation.

Accounting for Reserved Verilog Keywords

If the Verilog source uses Verilog reserved words (for example, “`int`” or “`do`”), then ModelSim can have compilation and loading errors. In this case, you must use the `-no_sv` switch so ModelSim can compile and load.

Accounting for Reg and Wire for Interconnect

In normal Verilog, only a wire can be used for interconnect, but in System Verilog, either a wire or reg can be used. Often, simulation library primitives are used in modules that also use `reg` as interconnect. In this case, the LibComp default is to use the `-sv` switch if `-atpg_lib_prims` are

needed (these often use reg as interconnect) and the -no_sv switch otherwise, unless you explicitly invoke with the -sv or -no_sv switch using one of the previous three methods.



Note

In the unlikely event that a reg is used for interconnect in Verilog libraries which do not contain simulation library primitives (for example, _MUX), then you must explicitly issue the -sv switch to allow ModelSim to compile and load.

Support for Verilog Parameter Overrides

Verilog parameters are compile time definitions of constants that are typically used to define the width of the IO and internal variables. Parameter overrides allow users to widen or narrow hardware when instantiating their definition and are useful to prevent replicating functionality where only the operand widths change.

LibComp now supports Verilog parameter overrides to widen and narrow operands. It creates a module using the original module name, appended with the parameter name and override value, for each changed parameter. For example:

```
// Defining module with parameters

module ramcore_16x6 (wba, wa, din, ra, dout);  parameter databits = 6;
parameter addrbits = 4;  parameter addrmax  = (1<<addrbits) - 1; ....
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    ...

// Instantiating module with overrides

module ram16x8( DO0, DO1, DO2, DO3, DO4, DO5, ..
    ramcore_16x6 #(8) u1 ( ...
```

This example re-defines the first parameter, *databits*, to be 8 rather than the originally defined 6; this causes the memory to be 8 bits wide rather than 6 bits. When LibComp translates *ram16x8*, it will define a model, *mlc_ramcore_16x6__databits_6*, whose name is created from the original defining module name, with “mlc_” prepended to indicate this is a name LibComp generated (not found in the original Verilog), and with each changed parameter and its value appended, as shown here:

```
model mlc_ramcore_16x6__databits_8
    (wba, wa, din, ra, dout)
    (
        model_source = verilog_parameter_override; ...
```

LibComp outputs a **model_source** statement to indicate the model was created due to a parameter override. These type of models are not instantiated explicitly for verification because no corresponding Verilog module exists to verify against. However, it will be tested in situ when the model *ram16x8* is verified.

The following limitations exist for Verilog parameter overrides:

- Current support is for positional overrides only and not for Verilog-named overrides.
- Current support is limited to integers to widen hardware; it does not extend to other constant types such as overriding a \$readmemb/h ROM initialization file or other uses of overrides.

LibComp Command Summary

The following table contains a brief summary of the LibComp commands.

Table 4-1. Command Summary

Command	Description
Add Models	Adds models from the currently loaded library into the set of models for compilation.
Delete Models	Removes a model from the current compilation set.
Dofile	Executes the commands contained within the specified file.
Exit	Terminates the application session.
Help	Displays the usage syntax for the specified command.
Report Models	Reports the models that have been added but not deleted.
Run	Starts the compilation process.
Set Asynchronous Control_Logic	Specifies whether output dominance logic is used during the translation of sequential UDP memory modules.
Set Behavioral Processing	Specifies whether the Verilog behavioral always statement and always block processing are processed.
Set BB Outputs	Specifies whether LibComp drives each bit of a bidi or output of a blackbox model from a _tiex gate or leaves those outputs as floating.
Set Dofile Abort	Specifies whether the tool aborts or continues dofile execution if an error condition is detected.
Set Empty_module Outputs	Specifies whether LibComp leaves each bit of a bidi or output of an empty module undriven (floating) or drives each bit from a _tiex gate.

Table 4-1. Command Summary

Command	Description
Set Excessive Pull_delay	Specifies a delay threshold for changing a weak driver with excessive delay to either a <code>_tiex</code> or an <code>_undefined</code> simulation library model.
Set Floating Net_type	Specifies how LibComp translates floating nets.
Set Hold Check	Specifies whether hold checks is performed.
Set Instance Portlist	Specifies naming conventions used for module instances.
Set MUX Nonconsensus_logic	Specifies whether LibComp outputs logic that precisely matches the Verilog UDP for any mux that does not have both consensus terms or, outputs <code>_mux</code> if either one or both consensus terms is missing from the UDP table.
Set System Mode	Changes the tool mode.
Set Undefined Instance	Specifies whether LibComp outputs a simulation model or a black box when a Verilog instance references a module name for which there is no defining module in the Verilog source.
Set Verification	Specifies which tool is used for verification or disables verification.
Set X_from_known Combinational_udp	Specifies whether LibComp outputs a simulation model or a black box for combinational UDPs that output X from a known input combination.
System	Passes the specified command to the operating system for execution.
Write Library	Writes the simulation models to a file.

LibComp Command Descriptions

This section describes each LibComp command in alphabetical order.

Use the line continuation character “\” when application commands extend beyond the end of a line in a dofile. The line continuation character improves the readability of dofiles and helps with the command line entry of multiple-argument commands.

Add Models

Scope: Setup mode

Prerequisites: The Verilog library that contains the cell model(s) that you want to add must be loaded.

Usage

ADD MOdels [*model_name...* | -ALI]

Description

Adds models from the currently loaded library into the set of models for compilation.

The Add Models command lets you specify one or more models to add to the compile list. You can also specify for the tool to add all models found in the netlist.

Arguments

- *model_name*
A repeatable string that specifies the name of the model to add into the set of models for compilation.
- -ALI
A switch that specifies to add all models from the currently loaded library into the set of models for compilation.

Example

The following example adds all models from the loaded library into the set of models for compilation.

```
add models -all  
set system mode translation  
run
```

Related Commands

[Delete Models](#)
[Report Models](#)

[Run](#)

Delete Models

Scope: Setup mode

Usage

DELeTe MOdels [*model_name...* | -ALL]

Description

Removes one or more specified models or all models from the current compilation set.

Arguments

- *model_name*
A repeatable string that specifies the name of the model to remove from the current compilation set.
- -ALL
A switch that specifies to remove all models from the current compilation set.

Example

The following example adds all models from the currently loaded library and then deletes one particular model from that library.

```
add models -all  
delete models ram8x4
```

Related Commands

[Add Models](#)
[Report Models](#)

[Run](#)

Dofile

Scope: All modes

Usage

DOfile *filename*

Description

Sequentially executes the commands contained in the specified file. This command is especially useful to issue a series of commands. Rather than executing each command separately, you can place them in a file and then execute them with the Dofile command. You can also place comment lines in the file by starting the line with a double slash (//); lines preceded with a double slash (//) are ignored.

The Dofile command sends each command expression (in order) to the tool which in turn displays each command line from the file before executing it. If an error is encountered due to any command, the Dofile command stops its execution and displays an error message. You can enable the Dofile command to continue regardless of errors with the [Set Dofile Abort](#) command.

The dofile <*your_tessent_software_tree*>/lib/tools/libcomp/libcomp.do.default can be copied and used as is or modified. For more information, see the “[Finding Unsupported Constructs in Partial Models](#)” topic in this manual.

Argument

- *filename*

A required string that specifies the name of the file that contains the commands to execute.

Example

The following example orderly executes all the commands from the *command_file* file:

```
dofile command_file
```

Related Command

[Set Dofile Abort](#)

Exit

Scope: All modes

Usage

EXIt

Description

Terminates the application session and returns to the operating system. Use this command for interactive sessions and in dofiles.

Example

The following example quits the tool without saving the current compilation set.

```
add model -all  
set system mode translation  
run  
exit
```

Help

Scope: All modes

Usage

HELp [*command_name*]

Description

Displays the usage syntax for the specified command. The Help command provides quick access to either information about a specific command, to a list of commands beginning with a specific key word, or to a list of all the commands.

Argument

- *command_name*

An optional string that either specifies the name of the command for which you want help or specifies one of the following keywords whose group of commands you want to list: ADD, DELEte, LOAd, SET, REPort, or WRItE.

If you do not supply a *command_name*, the default is to display a list of all the command names.

Report Models

Scope: Setup mode

Prerequisites: The Verilog library that contains the cell model(s) that you want to add must be loaded.

Usage

REPort MOdels [*model_name...* | -ALI]

Description

Reports the models that have been added but not deleted; these models will be translated when the run command is issued.

Arguments

- *model_name*
A repeatable string that specifies the name of the model to report.
- -ALI
A switch that specifies reporting add all models from the currently loaded library.

Example

The following example reports all models from the loaded library into the set of models for compilation.

report models -all

Related Commands

[Add Models](#)
[Delete Models](#)

[Run](#)

Run

Scope: Translation mode

Usage

RUN

Description

Starts the compilation process on all models currently in the compilation list. The list is created using the [Add Models](#) and [Delete Models](#) commands.

Example

The following example starts the compilation process on the models added to the compilation set.

```
Tessent_Tree_Path/bin/libcomp my_library.v -dofile -log my_log.log -rep  
add models -all  
set system mode translation  
run
```

Related Commands

[Add Models](#)
[Delete Models](#)

Set Asynchronous Control_Loic

Scope: Setup mode

Usage

SET ASynchronous Control_loic
 {-OUTput dominance logic| -NO_OUTput_dominance_logic}

Description

Specifies whether output dominance logic is used during the translation of sequential UDP memory modules.

This command applies only to sequential UDP memory models when both the Set and Reset signals are asserted. Use this command before the Run command to create the desired simulation models.

There are four possible Q output values that a UDP can produce when both Set and Reset are asserted. By default, the LibComp tool models each output as described in the following table.

Table 4-2. Output Dominance Logic

UDP Behavior	Model Dominance Logic	Q Output Value
Both Set and Reset dominant	No AND gating dominance logic used.	X
Reset dominant	AND gate used to de-assert Set when Reset is active.	0
Set dominant	AND gate used to de-assert Reset when Set is active.	1
No dominance	AND gates used to de-assert both Set and Reset when other asserted.	Hold

- **-OUTput dominance logic**
 Required switch that produces models that use dominance AND logic when both Set and Reset are asserted on a sequential UDP modules. This is the default.
- **-NO_OUTput_dominance_logic**
 Required switch that produces models that output an X value for sequential UDP modules when both Set and Reset are asserted, regardless of signal dominance.

Example

The following example creates a model that always produces an X value for UDPs when both Set and Reset are asserted.

```
Tessent_Tree_Path/bin/libcomp my_library.v -dofile -log my_log.log -rep  
add model -all  
set asynchronous control_logic -NO_OUTput_dominance_logic  
set system mode translation  
run
```

Related Commands

[Run](#)

Set Behavioral Processing

Scope: All modes

Usage

SET BEhavioral Processing **ON** | **OFF**

Description

Specifies whether or not the tool processes a Verilog behavioral always statement always block processing, and black box modules containing such blocks.

Arguments

- **ON**
A required literal that translates simple memories, DFFs, latches, muxes, and TSDs implied by simple behavioral constructs.
- **OFF**
A required literal that black boxes any module containing an always block, and skips any attempt at translating code within those blocks. This is the default.

Example

The following example instructs the tool to not process any Verilog always block.

```
set behavioral processing off
```

Related Commands

[Run](#)

Set BB Outputs

Scope: All modes

Usage

SET BB Outputs [-Tiex | -Float]

Description

Specifies whether LibComp drives each bit of a bidi or output of a blackbox model from a `_tiex` gate or leaves those outputs as floating.

Arguments

- -Tiex
An optional literal that specifies that each bidi or output bit of a blackboxed model is to be driven from a `_tiex` gate and comments are placed in the blackbox model identifying it as a black box.
- -Float
An optional literal that specifies that each bidi or output bit of a blackboxed model is to be left undriven (floating).

Example

The following example instructs the tool to leave blackbox output or bidi floating.

```
set bb outputs -float
```

Related Topics

[Set Empty_module Outputs](#)

Set Dofile Abort

Scope: All modes

Usage

SET DOfile Abort **ON** | **OFF**

Description

Specifies whether the tool aborts or continues dofile execution if an error condition is detected. The Set Dofile Abort command stops processing and reports any line numbers causing errors in the dofile. However, the Set Dofile Abort command enables you to turn this functionality off so that the tool continues to process all commands in the dofile.

Arguments

- **ON**
A required literal that halts the execution of a dofile upon the detection of an error. This is the default.
- **OFF**
A required switch that forces dofile processing to complete all commands in a dofile regardless of error detection.

Example

The following example sets the `set_dofile_abort` command off to ensure that all commands in *test1.dofile* are executed.

```
set system mode translation
set dofile abort off
dofile test1.dofile
```

Related Commands

[Dofile](#)

Set Empty_module Outputs

Scope: All modes

Usage

SET EMpty_module Outputs [-Float | -Tiex]

Description

Specifies whether LibComp leaves each bit of a bidi or output of an empty module undriven (floating) or drives each bit from a _tiex gate.

If you specify the *-Float* option, you can use the [set_tied_signals](#) command in the Tessent TestKompress/Tessent FastScan dofile to simulate these pins as Z; this behavior mimics Verilog. The default in Tessent TestKompress and Tessent FastScan is to _tiex undriven bidis/outputs.

If you specify the *-Tiex* option, you cannot simulate these pins as Z in Tessent TestKompress/Tessent FastScan; this can sometimes cause bus contention issues around IO pads.

The default behavior in LibComp is *-Float* which preserves the ability to match Verilog simulation without modifying the simulation models using the [set_tied_signals](#) command in the TK/FS dofile.

Arguments

- **-Float**
An optional literal that specifies that each bidi or output bit of an empty module is to be left undriven (floating); comments are placed at the end of the module indicating that it came from an empty module and is not a blackbox resulting from a failed translation.
- **-Tiex**
An optional literal that specifies that each bidi or output bit of an empty module is to be driven from a _tiex gate.

Example

The following example specifies that the bidi or output bits of empty modules are to be driven from a _tiex gate.

```
set empty_module outputs -tiex
```

Related Topics

[Set BB Outputs](#)

Set Excessive Pull_delay

Scope: All modes

Usage

SET EXcessive Pull_delay *delay_time*

Description

Specifies a delay threshold for changing a weak driver with excessive delay to either a `_tiex` or an `_undefined` simulation model. This can prevent tester issues and simulation mismatches from appearing when drivers with excessive delay exceed the simulation cycle.

By default, LibComp detects excessive switching delay for weak pull-up and pull-down drivers on an integrated circuit's input or bidirectional pads. Using the Set Excessive Pull_delay command, you can specify a delay threshold in time units (*delay_time*) for these weak drivers. If a driver's delay is excessive (equal or greater than the threshold), then LibComp converts the driver to one of the following simulation models:

- **`_tiex`** — If the driver has no inputs.
- **`_undefined`** — If the driver has inputs.

The default delay is 100 time units, which converts weak drivers having a delay equal to or exceeding 100 time units. If you set the *delay_time* to 0 (zero), then LibComp converts any weak driver or net to drive weak X values rather than known values.

Note



If you set *delay_time* to a large integer (for example, 1000000), ATPG obtains input values from a weak driver instead of an I/O pad. This setting potentially cause simulation mismatches if the drivers delay exceeds the simulation cycle.

Argument

- *delay_time*

A optional non-negative integer specifying weak drivers' excessive delay in time units. The default is 100 time units.

Example

The following example sets the delay to 1000 time units.

```
set excessive pull_delay 1000
```

Set Floating Net_type

Scope: All modes

Usage

SET FLoating Net_type [NONE | X | Z]

Description

Specifies how LibComp translates floating nets.

By default, LibComp leaves floating nets floating to provide flexibility. The default for ATPG is to treat floating nets as driven by _tiex; you can use the [set_tied_signals](#) to change that to _tiez without editing the library. You may need to do this for some IO pad models to prevent unsolvable bus contention issues for ATPG.

If this command is set to X or Z, floating nets without a driver are driven by/tied to X or Z, respectively, in the LibComp library output; these nets are not affected by the set_tied_signals command.

Arguments

- NONE
A literal that specifies to translate floating (undriven) nets literally, leaving them floating.
- X
A literal that specifies to create an explicit _tiex primitive to drive undriven model output and inout pins.
- Z
A literal that specifies to create an explicit _tiez primitive to drive undriven model output and inout pins.

Example

The following example ensures that each net has a _tiex driver if no others.

```
set floating net_type X
```

Related Topics

[set_tied_signals](#)

Set Hold Check

Scope: All modes

Usage

SET HOLD Check ON [-fix_async_fails | -black_box_async] | Off

Description

Specifies how sequential UDPs with hold check violations are translated.

Arguments

- ON
A literal that enables asynchronous hold checks. This is the default.
- -fix_async_fails
A switch that fixes models that fail asynchronous hold check. LibComp creates a model that outputs an X to match the verilog. The data ports are usable on such models. This switch only fixes asynchronous hold check failures. This is the default.
- -black_box_async
A switch that outputs a black box for models that fail a hold check. Prevents simulation mismatches, but causes fault coverage problems if black boxes are used for ATPG.
- Off
A literal that disables hold checks and outputs models that may cause simulation mismatches when an input changes in the Verilog. Hold fails are output in the transcript.

Set Hold Check Functionality

It is common that Verilog UDPs are erroneously specified in that the rows saying that the state should hold when some UDP input changes is omitted.

This is especially prevalent for asynchronous set and reset deasserting (going from set or reset to off, or the deasserted value). This is not how the hardware really works, but the UDP error may cause simulation mismatches if the offending input changes during simulation. This is because the simulation model will correctly hold the state, and disagree with the incorrect Verilog result. For an asynchronous input, often it is tied off outside, and so the error cannot cause a simulation mismatch, but the potential is there if a simulation model (ATPG library) is created which assumes that will be the case.

There are three possibilities if the hold check occurs on a set or reset (an asynchronous input).

- Do not make such a hold check but instead create a simple model (what was intended but not defined properly by the Verilog) and hope for no mismatches. That is accomplished by using Set Hold Check Off.
- A safer option, and the default, is to create a remodel that will allow the data port(s) to work but will go to X if the erroneously defined asynch is ever asserted. This will match

the Verilog and mask its error if it occurs. This is accomplished by a fanout from the offending asynchronous input to both the set and reset of the simulation model, so that if asserted, both set and reset of the ATPG model assert, and it goes to X.

- Clock or data input hold fails cause black boxed models unless Set Hold Check Off is issued. There is no reasonable way to protect the model in those cases, so only the simple, dangerous model, or the black box options are available for UDPs where one or more of those inputs fail to hold. Clocks must hold when deasserting with all other clocks and any asynchs at their off (nonasserted / hold) values. Data must hold for both changes (rise or fall) when all clocks and asynchs are at their off (hold) value.

The best solution is to fix the Verilog UDP, which is simply wrong in almost all cases (especially for asynchronous fails).

Example 1

The following example black boxes asynchronous modules that fail hold check.

```
set hold check on -black_box_async
```

Example 2

The following example turns off hold checks.

```
set hold check off
```

Related Commands

[Run](#)

Set Instance Portlist

Scope: Setup mode

Usage

SET INstance Portlist { **-PIN_names** [**-EXclude udp**] } | **-POSitional**

Arguments

- **-PIN_names**
A required literal that specifies that pinname connections should be used for instances of modules and undefined instances. This is the default.
- **-EXclude udp**
A required literal that specifies that instances whose definition is a Verilog UDP should not be pinname, but positional. This is the default.
- **-POSitional**
A required literal that specifies that all instance portlists should be positional. This is backward compatible with pre-v8.2009_2 releases.

Description

Specifies naming conventions used for module instances.

Prior to the v8.2009_2 release, LibComp and the ATPG library were restricted to positional connections in the portlist of an instance. For example:

```
instance = zt8binv i1 ( rden1, rden1_b ); // first instance
```

In the v8.2009_2 and all subsequent releases, ATPG library syntax is extended to include pinname connections. For example:

```
instance = zt8binv i1 ( .a(rden1), .o1(rden1_b) ); // second instance
```

For a defining model with port (pin) names as shown here, the first instance connects *rden1* to the first pin in the defining model's portlist; the second instance connects *rden1* to the port (pin) named “a” in the defining model's portlist, regardless of its position:

```
model zt8binv (a, o1)
```

By default, LibComp uses pinname connections for instances whose definition is missing (in which case automatic verification must be set off due to incomplete input), and for any instantiation of a defined module.

Instances of primitives remain positional. Instances defined by a Verilog UDP default to positional connections when translated by LibComp. However, because UDPs become an entire model rather than a single primitive, pinname portlist connections are legal and supported for those instances as well.

Example 1

The following example shows the default setting at the time of tool invocation.

```
set instance portlist -pin_names -exclude_udp
```

Example 2

The following example specifies to use pinnames on all instances of modules, including UDPs.

```
set instance portlist -pin_names
```

Example 3

The following example specifies to use positional connections for all instances. This behavior is consistent with the behavior of pre-2009_2 releases.

```
set instance portlist -positional
```

Set Model Source

Scope: All modes

Usage

SET MModel Source **ON** | **OFF**

Description

Specifies whether LibComp outputs the [model_source](#) statement for every model created.

This source statement is useful to guide automatic verification to avoid flow issues when compiling libraries in the Verilog simulator.

In releases prior to v8.2009_3, the Tessent TestKompress/Tessent FastScan library parser cannot read models containing the **model_source** statement. When you specify the *-Off* option, LibComp eliminates this statement from the simulation models it created which allows those models to be used by pre-v8.2009_3 versions of Tessent TestKompress/Tessent FastScan.

Arguments

- **ON**
A required literal that specifies that LibComp outputs the [model_source](#) statement for every model created.
- **OFF**
A required literal that specifies that LibComp eliminates the [model_source](#) statement for every model created.

Example

The following example instructs LibComp to create models that can be read by pre-v8.2009_3 versions of Tessent TestKompress/Tessent FastScan.

```
set model source off
```

Related Commands

Set MUX Nonconsensus_logic

Scope: All modes

Usage

SET MUX NONconsensus_logic {**EXPLicit** | **NONE** | **ALL**}

Description

Specifies the LibComp behavior that occurs when all, none, or one consensus terms are omitted from the UDP or, specifies to always outputs _mux irrespective of which consensus terms are present.

Arguments

- **EXPLicit**

A required literal that specifies that LibComp issues a warning and outputs _mux when both consensus terms are omitted from the UDP; in this case, LibComp assumes the user forgot to include them. If only one consensus term is present, LibComp assumes the user intends a one consensus term implementation and uses nonconsensus logic for the remodel. If both consensus terms are explicitly stated to produce X out of the 2-1 mux in the UDP, nonconsensus logic is used.

- **NONE**

A required literal that specifies to return to pre-v8.2009_4 behavior and always outputs _mux irrespective of which consensus terms are present. Nonconsensus logic is never used for any 2-1 mux remodel.

- **ALL**

A required literal that can be specified if the simplified _mux model causes mismatches when select is X and both data are known and agree, or if the user intended to produce an X in these cases but did not explicitly state that in the UDP.

Example

The following example instructs LibComp to return to pre-v8.2009_4 behavior.

SET MUX NONconsensus_logic NONE

Related Commands

Set System Mode

Scope: All modes

Usage

SET SYstem Mode **Setup** | **Translation**

Description

Specifies whether the tool enters the Setup or Translation mode.

Arguments

- **Setup**
A required literal that specifies to enter the Setup system mode from translation. By default the LibComp tool invokes in the setup mode. Within this mode, you can open Verilog libraries, select models for compilation, and setup options to control the compilation process. When you re-enter Setup mode from translation mode, the tool destroys any existing compiled network. When you leave Setup mode, the tool converts each added model to a network. The tool also performs pre-compilation checks.
- **Translation**
A literal that specifies to enter the Translation mode from Setup mode. When you enter Translation mode, the tool performs model checking for compilation.

Example

The following example puts the tool in Translation mode.

```
set system mode translation
```

Related Commands

[Add Models](#)

[Run](#)

Set Undefined Instance

Scope: All modes

Usage

SET UNdefined Instance [-ATpg_model | -BLack_box]

Arguments

- -ATpg_model
An optional literal that specifies to output an ATPG model using a _tsh (tri-state buffer) simulation model primitive when a Verilog instance references a module name for which there is no defining module in the Verilog source. This is the default.
- -BLack_box
An optional literal that specifies to output a black box model when a Verilog instance references a module name for which there is no defining module in the Verilog source.

Description

Specifies whether LibComp outputs an simulation model or a black box when a Verilog instance references a module name for which there is no defining module in the Verilog source.

By default, LibComp outputs a simulation model. You can specify the -black_box option which causes the module to be blackboxed.

When undefined instances are encountered, LibComp sets verification to off ([Set Verification](#) -Off) because ModelSim will fail to load an incompletely defined simulation model and, therefore, verification is not possible.

Example

The following example specifies to output undefined modules as blackboxes.

```
set undefined instance -blackbox
```

Set Verification

Scope: Setup mode

Usage

SET VERification {**FASTscan** | **FLEXtest** | **TESTKompress** | **Off**} [-SV | -NO_SV]

Description

Specifies which tool is used for verification or disables the verification of a generated simulation library (ATPG library) after the initial compilation. This is most often done in a LibComp dofile.

Note



You can run just the LibComp verification step with the lcVerify script. For more information, see [“Verifying Tessent Simulation Models”](#).

- **FASTscan**
A required literal that sets Tessent FastScan as the verification tool. This is the default.
- **FLEXtest**
A required literal that sets FLEXtest as the verification tool.
- **TESTKompress**
A required literal that sets Tessent TestKompress as the verification tool.
- **Off**
A required literal that disables verification during compilation runs.
- **-SV**
An optional switch that invokes ModelSim with System Verilog compilation for .v files. Required when a reg is used as interconnect (such as in a port list). See [“Reconciling System Verilog reg and Verilog Keyword Compiling Issues”](#) for complete information.
- **-NO_SV**
Prevents invoking ModelSim with -sv (System Verilog) for .v files. Required when the -sv switch causes compile errors due to modules containing reserved Verilog keywords. See [“Reconciling System Verilog reg and Verilog Keyword Compiling Issues”](#) for complete information.

Example

The following example disables verification during compilation runs.

```
set verification off
```

Related Commands

[Dofile](#)

[Run](#)

Set X_from_known Combinational_udp

Scope: All modes

Usage

SET X_from_known Combinational_udp { atpg_model | black_box }

Description

Specifies whether LibComp outputs a simulation model or a black box for combinational UDPs that output X from a known input combination.

By default, LibComp outputs a simulation model that uses a buffered tri-state (_tsh) simulation primitive with the following attributes:

- The model's enable logic matches the UDP's X rows.
- The model's data input logic matches the UDP's known (Boolean) rows.

You override this default behavior by specifying the black_box argument with this command.

Arguments

- atpg_model
An optional literal specifying the outputting of a simulation model using a _tsh (tri-state buffer) simulation primitive for combinational UDPs that output X from some known input combinations. This is the default.
- black_box
An optional literal specifying the outputting of a black box model for combinational UDPs that output X from some known input combinations.

Example

The following example defaults to outputting a simulation model:

```
set x_from_known combinational_udp
```

System

Scope: All modes

Usage

SYStem *os_command*

Description

Executes one operating system command without exiting the currently running application.

Argument

- *os_command*
A required string that specifies any legal operating system command.

Example

The following example displays the current working directory without exiting the tool:

system pwd

Write Library

Scope: Translation mode

Prerequisite: You must perform a compilation run before you issue this command.

Usage

WRItE LIbrary *filename* [-Replace]

Description

Writes the simulation models created during compilation to a specified filename.

Arguments

- *filename*
A required string that specifies the name of the file to which the tool writes compiled models.
- -Replace
An optional switch that forces the tool to overwrite the compiled library file if a file by that name already exists.

Example

The following example compiles the models in the Verilog library *vlib.v* then saves the compiled models to a file.

```
Tessent_Tree_Path/bin/libcomp vlib.v -dofile -log my_log.log -rep  
add model -all  
set system mode translation  
run  
write library vlib.atpg -replace
```

Related Commands

[Run](#)

UDP Limitations and Examples

LibComp focuses primarily on the translation of User-Defined Primitives (UDP) and gate-level component descriptions, such as combinational/sequential UDPs and gate-level and switch-level constructs.

2-1 Mux Translation

Libcomp translates a 2-1 multiplexor UDP into a `_mux` that has consensus terms in its simulation. Note: If select is X but the data agree on what the output value should be, the output becomes that value. When a UDP has a single consensus term, LibComp assumes a `_mux` should not be used and outputs logic gates implementing a one consensus term mux. Also, when both terms are missing, LibComp assumes this is an oversight of the user and outputs a `_mux`. You can explicitly state that the two consensus terms should output X in the UDP or you can use the [Set MUX Nonconsensus_logic](#) command to change this behavior.

General 3-Valued Limitations

If a UDP specifies all Boolean combinations of inputs, no attempt is made to produce an X output for cases where some inputs are X. There may be situations where the Verilog models contain unknowns which cannot happen in hardware. Since there are some LV flows where an X will destroy the signature, it is necessary that the X's be bounded so they cannot propagate downstream. If the UDP pessimism is not really valid in hardware, it will cost a lot to create the models to produce the X's, then bound them with more hardware to kill the X's, especially if it is to take care of some situation which cannot happen in hardware. If the X's have been properly bounded as required for the LV flows, a lighter simpler Boolean model will be sufficient.

UDP With Both Consensus Terms

This is a UDP for a mux which has both consensus terms. This model is highly recommended.

You can use an "X" rather than a "?" for the S input value for the last two consensus rows. They are functionally equivalent, but the "?" makes it clear that S is a DontCare when data agree.

```

primitive mux_both_consensus_udp
`protect
(Y, S, A, B);
output Y;
input S, A, B;

table
//      S  A  B   : Y;
//      -----
      0  0  ?   : 0 ; // Select A
      0  1  ?   : 1 ; // Select A
      1  ?  0   : 0 ; // Select B
      1  ?  1   : 1 ; // Select B

```

```

        ? 0 0 : 0 ; // consensus term (both data in are 0)
        ? 1 1 : 1 ; // consensus term (both data in are 1)

endtable
`endprotect
endprimitive

```

LibComp outputs the following for this UDP:

```

model model mux_both_consensus_udp
(Y, S, A, B)
(
  model_source = verilog_udp;
  input (S, A, B) ( )
  output (Y) (
    primitive = _mux mlc_gate0 (A, B, S, Y);
  )
)

```

Partial Mux Examples - Missing Consensus Terms

Following are examples and default solutions.

Case 1 - Mux Missing 11 Consensus Term

```

primitive mux_missing_11_consensus_udp
(Y, S, A, B);
output Y;
input S, A, B;

table
//      S  A  B  : Y ;
//      -----
//      0  0  ?  : 0 ; // Select A
//      0  1  ?  : 1 ; // Select A
//      1  ?  0  : 0 ; // Select B
//      1  ?  1  : 1 ; // Select B
//      ?  0  0  : 0 ; // consensus term (both data in are 0)
//      // consensus term MISSING (both data in are 1)

endtable
endprimitive

```

LibComp outputs the following for this UDP which has only the 00 consensus term:

```

model mux_missing_11_consensus_udp
(Y, S, A, B)
(
  model_source = verilog_udp;
  input (S, A, B) ( )
  output (Y) (
    primitive = _inv mlc_not_S_gate (S, mlc_not_S);
    primitive = _and mlc_D0_gate (A, mlc_not_S, mlc_D0_net);
    primitive = _and mlc_D1_gate (B, S, mlc_D1_net);
    primitive = _or mlc_out_gate (mlc_D0_net, mlc_D1_net, Y);
  )
)

```



```
)  
)
```

Case 2 - Mux Missing 00 Consensus Term

```
primitive mux_missing_00_consensus_udp  
  (Y, S, A, B);  
  output Y;  
  input S, A, B;  
  
  table  
  //      S  A  B   : Y ;  
  //      -----  
      0  0  ?   : 0 ; // Select A  
      0  1  ?   : 1 ; // Select A  
      1  ?  0   : 0 ; // Select B  
      1  ?  1   : 1 ; // Select B  
      ?  1  1   : 1 ; // consensus term (both data in are 1)  
      // consensus term MISSING (both data in are 0)  
  
  endtable  
endprimitive
```

LibComp outputs the following for this UDP which has only the 11 consensus term:

```
model mux_missing_00_consensus_udp  
  (Y, S, A, B)  
  (  
    model_source = verilog_udp;  
    input (S, A, B) ( )  
    output (Y) (  
      primitive = _inv mlc_not_S_gate (S, mlc_not_S);  
      primitive = _or mlc_D0_gate (A, S, mlc_D0_net);  
      primitive = _or mlc_D1_gate (B, mlc_not_S, mlc_D1_net);  
      primitive = _and mlc_out_gate (mlc_D0_net, mlc_D1_net, Y);  
    )  
  )
```

Case 3 - Mux Missing Both Consensus Terms

```
primitive mux_missing_both_consensus_udp  
  (Y, S, A, B);  
  output Y;  
  input S, A, B;  
  
  table  
  //      S  A  B   : Y ;  
  //      -----  
      0  0  ?   : 0 ; // Select A  
      0  1  ?   : 1 ; // Select A  
      1  ?  0   : 0 ; // Select B  
      1  ?  1   : 1 ; // Select B  
      // BOTH consensus terms MISSING !!  
  
  endtable  
endprimitive
```

LibComp outputs the following for this UDP which has neither consensus term:

```
model mux_missing_both_consensus_udp
  (Y, S, A, B)
  (
    model_source = verilog_udp;
    input (S, A, B) ( )
    output (Y) (
      // Warning: UDP appears to implement _mux function, but is missing
      // consensus. Both terms missing, but not explicitly X in UDP, so
      // risking use of _mux remodel.
      primitive = _mux mlc_gate0 (A, B, S, Y);
    )
  )
```

Case 4 -- Mux With Explicit X Terms

```
primitive mux_both_consensus_explicitly_x_udp
  (Y, S, A, B);
  output Y;
  input S, A, B;

  table
  //      S  A  B  : Y ;
  //      -----
  //      0  0  ?  : 0 ; // Select A
  //      0  1  ?  : 1 ; // Select A
  //      1  ?  0  : 0 ; // Select B
  //      1  ?  1  : 1 ; // Select B
  //      X  0  0  : X ; // 00 consensus => X
  //      X  1  1  : X ; // 11 consensus => X

  endtable
endprimitive
```

LibComp translates the preceding UDP as follows:

```
model mux_both_consensus_explicitly_x_udp
  (Y, S, A, B)
  (
    model_source = verilog_udp;
    input (S, A, B) ( )
    output (Y) (
      primitive = _inv mlc_not_S_gate (S, mlc_not_S);
      primitive = _or mlc_anticonsensus_gate (S, mlc_not_S,
mlc_anticonsensus_net);
      primitive = _or mlc_D0_gate (A, S, mlc_D0_net);
      primitive = _or mlc_D1_gate (B, mlc_not_S, mlc_D1_net);
      primitive = _and mlc_out_gate (mlc_anticonsensus_net, mlc_D0_net,
mlc_D1_net, Y);
    )
  )
```

Mux Example With Consensus Terms

```
//
// 2-1 Multiplexor (single select line)
//
// CAVEAT: This mux *never* creates Z out (converted to X out).
// Otherwise, function is equivalent to : mux_out = select?d1:d0;
//   select==1 => mux_out=d1
//   select==0 => mux_out=d0
//   Both consensus terms (select==X, but known out if data agree).

module mux (mux_out, select, d1, d0);
  input d0, d1, select;
  output mux_out;
  mux_primitive _mux_inst (mux_out, select, d0, d1); endmodule

primitive mux_primitive (mux_out, select, d0, d1);
  input select, d0, d1;
  output mux_out;

  table
  // sel  d0  d1  :  out
    0    0   ?   :   0 ; // Select==0 => out = d0.
    0    1   ?   :   1 ;

    1    ?   0   :   0 ; // Select==1 => out = d1.
    1    ?   1   :   1 ;

    ?    0   0   :   0 ; // 0-consensus term
    ?    1   1   :   1 ; // 1-consensus term

  endtable

endprimitive
```

LibComp Limitation - Complex Asynchronous Logic

Complex asynchronous set/reset logic is not supported. You use structural gates external to the UDP in Verilog.

The LibComp tool only handles simple buffered or inverted sets and resets. If a scan cell has both an asynchronous set and asynchronous reset, and also uses scan_enable to gate off set and reset during shift, the asynchronous logic for scan_enable should be specified outside the UDP, and a simple (inverted or buffered) set and reset input should be all of the asynchronous gating logic that is included in the UDP table describing the scan cell's function. You should run the LibComp tool on all UDPs and check for any messages in <logfile_name> if invoked using "...libcomp... -log <logfile_name>", or check the simulation models (ATPG library) output for the string "BlackBox". In most cases, when LibComp encounters a UDP that cannot be translated, a black box model is output rather than a bad model.

Mux Scan DFF With Complex Asynchronous Logic

Create a Mux Scan DFF with Simple Asynchronous Logic using a UDP. (The following example uses Mux UDP and DFF UDP rather than one mux-scan UDP to show an alternative to the earlier single UDP mux-scan DFF.) Then implement complex asynchronous logic outside of the UDP in a module, and connect structurally to an instantiation of the simple-asynchronous UDP placed inside the module.

Example - Mux Scan DFF

The latch has the same asynchronous limitation.

DFF with active high reset and active high set. Neither is dominant, so asserting both produces an X state value.

```
primitive dff_activeHI_set_and_reset( q, s, r, c, d );
  input s, r, c, d;
  output q;
  reg q;
  table
  // s  r  c  d : q : q+;
  //-----
    1  0  ?  ? : ? : 1 ; // Assert asynchronous Set.
    0  1  ?  ? : ? : 0 ; // Assert asynchronous Reset.
    1  1  ?  ? : ? : x ; // Assert both set and reset result Unknown.
    0  ?  r  0 : ? : 0 ; // Clock in a 0 from d (cannot assert set).
    ?  0  r  1 : ? : 1 ; // Clock in a 1 from d (cannot assert reset).
    0  0  0  ? : ? : - ; // Hold when controls & clock off.
    ?  ?  ?  * : ? : - ; // Hold when data changes.
    ?  0  ?  1 : 1 : 1 ; // When d=q, next state same whether clocked
    0  ?  ?  0 : 0 : 0 ; //   or not clocked if controls consistent.
  endtable
endprimitive
```

Example - Mux Scan Cell With Asynchronous Gated Off By Scan Enable

```
module mux_scan_dff_with_complex_asynch_logic ( q, clk, d, si, sen, rst_,
set_ );
  input clk, d, si, sen, rst_, set_;
  output q;

  wire rst_net, set_net, d_net;

  // Use Verilog nor primitives for complex asynchronous logic
  //   outside UDP. This cannot be placed inside UDP that the LibComp
  //   tool automatically translates (current limitation).
  //   The complex gating is used to gate off the asynchronous set and
  //   reset while scanning (sen = 1). Using DFF UDP with activeHI set
  //   and reset, so NOR each activeLO asynchronous input with sen
  //   to disable asynchs when scanning (even if asynch inputs asserted).
```

```
nor rst_nor (rst_net, sen, rst_); // Reset if rst_ asserted in
                                // system mode (sen = 0).
nor set_nor (set_net, sen, set_); // Set if set_ asserted in
                                // system mode (sen = 0).

// Use mux UDP defined near beginning of this file. Creating a
// structural mux-scan DFF.

mux_x_consensus_udp mux_scan_gate (d_net, sen, d, si); // select si
                                                         when sen=1 (scanning).

// Use DFF UDP defined earlier in this file (with activeHI set and
// reset).
dff_activeHI_set_and_reset dff_gate (q, set_net, rst_net, clk, d_net);

endmodule
```

LibComp Limitation - Verilog Construct Support

The LibComp tool supports only a subset of Verilog constructs. When unsupported constructs are encountered, the supported pieces of the module are made into a partial simulation model. You can then locate and complete the model. For more information, see [“Finding Unsupported Constructs in Partial Models”](#).

Behavioral Constructs

The LibComp tool does not support Verilog constructs with the following behavioral constructs:

- Always blocks
- Case statements
- Tasks
- Functions
- Initial blocks
- Loops

Structural Constructs

The LibComp tool does not support Verilog constructs with the following structural constructs:

- Parameters that are expressions (For example, expression “a & b” in the actual parameter of a port)
- Modules or primitives with undefined instances
- Non-Logical operation in the continuous assign statement

- Array, Vector, \$<functions> and integer data structures/calls
- Non-Assign operations and real types in the continuous assign statement

DFF Example

```
//
// DFF
//   ActiveHI set    ActiveHI reset  posedge clock
//

module dff (q, set, reset, clock, data);
    input set, reset, clock, data;
    output q;
    dff_primitive dff_inst (q, set, reset, clock, data); endmodule

primitive dff_primitive ( q, set, reset, clock, data ); input set, reset,
clock, data; output q; reg q;

table
// s    r    c    d : q : q+;
//-----
(01)  0    ?    ? : ? : 1 ; // Assert asynchronous Set.
    1 (10) ?    ? : ? : 1 ; // Set/Reset asserted then Reset deasserts.
    0 (01) ?    ? : ? : 0 ; // Assert asynchronous Reset.
(10)  1    ?    ? : ? : 0 ; // Set/Reset asserted then Set deasserts.
(01)  1    ?    ? : ? : x ; // Assert both set and reset.
    1 (01) ?    ? : ? : x ; // Assert both set and reset.
(10)  0    ?    ? : ? : - ; // Hold when deassert set, reset inactive.
    0 (10) ?    ? : ? : - ; // Hold when deassert reset, set inactive.
    0   ? (01) 0 : ? : 0 ; // Clock in 0 (set must be known inactive).
    ?   0 (01) 1 : ? : 1 ; // Clock in 1 (reset must be known inactive).
    ?   ? (10) ? : ? : - ; // Clock falling can never change state.
    ?   0    ?    1 : 1 : 1 ; // d=q=1, reset deasserted, q remains 1.
    0   ?    ?    0 : 0 : 0 ; // d=q=0, set deasserted, q remains 0.
    ?   ?    ?    * : ? : - ; // Data changing can never change DFF state.
endtable
endprimitive
```

D Latch Example

```
//
// D Latch
//   ActiveHI set    ActiveHI reset  ActiveHI clock
//

module dlat (q, set, reset, clock, data);
    input set, reset, clock, data;
    output q;
    dlat_primitive _dlat_inst (q, set, reset, clock, data); endmodule

primitive dlat_primitive ( q, set, reset, clock, data ); input set,
reset, clock, data; output q; reg q;

table
```

```
// s    r    c    d : q : q+;
//-----
1    0    0    ? : ? : 1 ; // Assert asynchronous Set.
0    1    0    ? : ? : 0 ; // Assert asynchronous Reset.
1    1    ?    ? : ? : x ; // Assert both asynchs result Unknown.
0    ?    1    0 : ? : 0 ; // Clock in 0 from d (cannot assert set).
?    0    1    1 : ? : 1 ; // Clock in 1 from d (cannot assert reset).
?    0    ?    1 : 1 : 1 ; // When d=q=1, next state 1 if not reset.
0    ?    ?    0 : 0 : 0 ; // When d=q=0, next state 0 if not set.
0    0    0    ? : ? : - ; // Hold when all controls & clock inactive.

endtable
endprimitive
```

I/O Pad Limitations and Examples

This section describes how switch level modeling, bidirectional primitives, and strength constructs are handled.

Almost all I/O pads are built upon the following basic building block. In this section, the term “I/O pad” refers to all the parts, but the term “PAD” is often the name of the bidirectional pin on an I/O pad. In this manual, all caps “PAD” means the pin, and lower case “I/O pad” means the entire module.

Port list pin order can vary arbitrarily, but the pin functions assigned to input, inout, and output are always the same for a basic I/O pad. When an I/O pad is embedded in hardware (called the “core”), the I/O pad can output data from the core to the outside world using a bidirectional PAD pin. The data and the output enable controlling it are inputs to the I/O pad. If enabled, the I/O pad is in output mode, and the external world should be high impedance at PAD, so that PAD becomes the value of data is output from the I/O pad.

An I/O pad inputs data (when embedded in a core) into the bidirectional PAD and on into the core through a data output pin of the I/O pad model (there is usually a buffer in the path from the bidirectional PAD pin to this output pin).

Often, there are pulls, resistors, capacitors, etc. all modeled in Verilog. Scan testing is not used to test that hardware (parametric testing is best for that), so a simple model that is adequate to do I/O is typically best for scan testing.

As a result, this basic I/O pad model is often best to use even for much more complex I/O pads, once such pads are tied off by being embedded, or “programmed”. If the model is to be verified using the lcVerify tool (a subsidiary of LibComp), then additional pins which can cause pulls to activate, and/or other non-modeled functionality to be exercised, should all be pin constrained in the dofile to restrict the ATPG from accidentally random filling these pins. Unless constrained, the tool will not know that exercising the pins can cause false simulation mismatches, and will therefore result in false simulation mismatches. See [Pin Constraints Required for Verification](#) for how to accomplish this.

Strength Propagation

Often, there are inputs that indicate an external supply. These inputs are often named “E3V” or “E2_5V” etc. Typically, if powered ON, the simpler I/O pad illustrated below emerges (or at least is more closely approximated), with one or more switches effectively becoming wires (always ON). During scan test, everything should be powered ON, and tests will fail otherwise, so a simpler “powered ON” remodel for scan test can simplify ATPG. However, if you are going to verify such simplified remodels, see the [Pin Constraints Required for Verification](#). If pulls are modeled, this becomes critical, due to the lack of strength propagation in ATPG. Any transistors between an output driver node with a weak pull and the PAD node must be removed in the remodel to allow the strong output driver and PAD to correctly cause bus contention if fighting, but for the weak signal to yield to them both unless they are Z. This can only be done if all the fighting drivers and weak ones are one node (or a set of wired nodes). Any intervening transistors can be problematic.

Also, input pins which indicate legal differential (or other operation modes) can exist. Such inputs are often only used to cause X output values, and/or to cause messages from always blocks. Typically, these should be tied off outside during test to the appropriate values to prevent spurious unwanted events and/or messages, so a simple remodel ignoring them can be appropriate. However, if you are going to verify such simplified remodels, constrain such pins (see `add_input_constraints`).

Once processed for ATPG stand alone verification (not embedded in the netlist yet), the top level module's inputs becomes PIs, its outputs becomes POs, and its inout (bidirectional pins) become split into a PI and PO half with the same (original Verilog) pin's name used for both. When embedded, usually only the bidirectional PAD pin reaches the top level netlist, and the other pins are connected to core logic or tied off to “program” the I/O pad.

Pin Constraints Required for Verification

To verify the library, you must add pin constraints to any model input that must be other than 0 for proper operation. The `lcVerify` tool ties PIs that go nowhere to LO (0) currently, so pins which must be tied to 1, Z, or X for the model to be valid need be constrained to that value in the ATPG dofile.

During verification, pin “pin_name” on module “module_name” will be represented by a fake wrapper pin using the string “module_name__pin_name” (separated by TWO underscores). So, after running LibComp to get an ATPG dofile “fastscan.do.cat” in your local directory, you edit that and at the top add a line such as:

```
add_input_constraints module_name__pin_name C1 // Constant 1 (HI)
// pin constraint
```

The following will constrain pin “ENBI” of module (model) “TUD1ZE1000” to HI (1) for all ATPG patterns, after being added to the ATPG dofile used for verification.


```
add_input_constraints TUD1ZE1000__ENBI C1
```

For more information, see the [add_input_constraints](#) command in the *Scan Insertion, ATPG, and Diagnosis Reference Manual*.

I/O Pad Code Example

```
module module_name (
    data_out, output_enable, PAD, data_in
);

    // The data being output from the core to the PAD (outside world) thru
    // this I/O pad.
    input data_out;

    // The output enable for the data. Enabled in output mode. Disabled in
    // input mode.
    input output_enable;

    // The bidirectional pad where data goes in/out.
    inout PAD;

    // The data being input to the core from the PAD (outside world).
    output data_in;

    // Connect the output TriState Driver (TSD) directly to the
    // bidirectional pin. Do not include intervening wired ON MOS
    // switches, etc. from the more complex Verilog simulation model.
    // Also do not include weak pullup loops (buskeepers), or any
    // other programmable functionality. However, remember to pin
    // constrain any inputs enabling such pulls to OFF for the
    // lcVerify library verification if it is to be used.
    // Alternatively, alter the dofile to "Set External Z Handling X".
    // It is "Z" in the verification dofile, which allows it to produce
    // and measure Z values at PAD.
    // This is undesirable if the pulls can be enabled, so either
    // disable them, or have it convert Z's to X at PAD for measures
    // by changing the dofile command option. One prevents the pull
    // from happening, and the other predicts X (don't measure)
    // in cases where no strong driver is driving (and therefore the
    // unmodeled pull could cause a mismatch with the predicted Z from
    // an ATPG pattern's PAD PO value).
    //
    // Will assume activeHI output enable. If activeLO, use bufif0
    // instead of bufif1.

    bufif1 output_driver (PAD, data_out, output_enable);

    // Buffer the PAD input from the external world.

    buf input_buffer (data_in, PAD);

endmodule
```

Memory Limitations and Examples

LibComp has the following memory limitations:

- [Memories](#)
- [Verilog Constructs](#)
- [Arrays](#)
- [\\$readmemh and \\$readmemb](#)

Memories

LibComp only supports memories with vector addresses; that is, addresses with an address line bit that is greater than 1.

For example, the following notation is not supported:

```
reg 7:0 my_mem [0:0]; // A one-word memory - 1 bit address line
reg 7:0 my_mem [0:1]; // A two-word memory - 1-bit address line
```

The following notation is supported if j and k are both greater than 0 AND a greater than 1 difference exists between j and k :

```
reg <anything> my_mem [j:k];
```

Verilog Constructs

LibComp does not translate Verilog models containing tasks, functions, case statements, or loops. For example, if the Verilog model contains the following conditional loop:

```
if (address_is_not_all_known)
  for all words in memory
    memory[addr] = all X (Unknown)
```

an error message is issued indicating the model is not translated and the model may be black boxed. You should create a simple, but simulatable, Verilog description that does not check for incorrect operation and X out the memory. The model should only contain control logic and simple always blocks for the read and write ports. Initialization requirements (such as ignoring the first j clocks in a memory) should not be included in the models, but instead, specified in a test_setup procedure that pulses the appropriate clocks j times before starting the test.

Before remodeling Verilog behavioral models, use LibComp to experiment with a construct or form of description to ensure the expression form can be translated by LibComp.

Arrays

LibComp only recognizes an array as a memory if its index (address) is at least two bits. Therefore, it will not translate a memory with a single address line.

\$readmemh and \$readmemb

LibComp translates files containing \$readmemh and/or \$readmemb if:

- The system call contains the filename in the module with memory.
- The memory has at least a read always block.
- \$readmemh and \$readmemb are in an initial block as a simple statement or a simple if-else construct.

The following ROM and RAM examples show Verilog source that can be translated by LibComp. Some of the examples also contain comments about memory limitations.

ROM Example

```
`timescale 1ns / 1ns

module example_ROM (Q, CK ,CSN, A);

parameter
    Words = 768,
    Bits = 8,
    Addr = 10;

parameter
    InitFileName = "user_init_file.dat";

output [Bits-1 : 0] Q;
input [Addr-1 : 0] A;
input  CK, CSN;

reg [Bits-1 : 0] Mem [Words-1 : 0];
reg [Bits-1 : 0] Qreg;

// Note: InitFileName "user_init_file.dat" should be translated and
// provided in ATPG format if ROM will contain such data during scan
// test, and you wish to exploit that known data.
// Otherwise, all internal ROM bits are Unknown (X).

initial
begin
    $readmemb(InitFileName, Mem, 0, Words-1);
end

// ROM is simply a Ram with only a Read Port.
always @ (posedge CK)
```

```
begin

    if (CSN == 1'b0)
    begin
        Qreg <= Mem[A];
    end

end

// Outside CK control, so only _wire / assignment relation, not DFF
//   or LATch.
assign Q = Qreg;

endmodule
```

RAM Examples

Single Posedge Ports With Separate Port Clocks

```
//
// Simple RAM with edge triggered read and write ports,
//   and common (shared) address inputs.
//
module ram1024x8 (wclk, rclk, a, din, dout);

    parameter databits = 8;
    parameter addrbits = 10;
    parameter addrmax = (1<<addrbits) - 1;

    input wclk, rclk;
    input [addrbits-1:0] a;
    input [databits-1:0] din ;
    output [databits-1:0] dout;

    // Memory is declared as a reg.
    reg [databits-1:0] mymem [0:addrmax];

    reg [databits-1:0] dout ;

    // Edge triggered write port clocked by wclk.
    always @ (posedge wclk) mymem[a] <= din;

    // Edge triggered read port clocked by rclk.
    always @ (posedge rclk) dout <= mymem[a];

endmodule
```

Single Level Ports With Write-Thru and Trisate Output Enable

```
//
// RAM with tristateable output.
// Common address for read/write, level sensitive for both read and
// write ports, active HI chip select CS, active HI output
// enable OE, and active low write enable WEB.
//
// Note that an event is used to cause write-thru (writing to
// some address while reading from it causes output to immediately
// reflect new data written if the output is enabled).
//
module ram128x32 (DO, DI, A, WEB, OE, CS);

    parameter databits = 32;
    parameter addrbits = 7;
    parameter addrmax = (1<<addrbits) - 1;

    output [databits-1:0] DO;
    input [databits-1:0] DI;
    input [addrbits-1:0] A;
    input WEB, OE, CS;

    reg [databits-1:0] memory [0:addrmax];
    reg [databits-1:0] DO;

    and u0 (OEN, CS, OE);
    and u1 (WEN, CS,!WEB);

    event WRITE_OP;

    // Write Port
    // Level sensitive, so could respond to Address, data, or
    // clock (strobe).
    always @ (WEN or A or DI)
        if (WEN) begin // Active HI write clock
            memory[A] = DI;
            #0; ->WRITE_OP; // Causes write-thru (in case output is enabled)
        end

    // Read Port
    // Always read. Output can respond to output enable, data (if enabled),
    // or write-thru event (if enabled).
    // Output TSD (Tristate Drivers pass memory output if OEN is HI, else
    // output high impedance (Z).
    always @ (OEN or A or WRITE_OP)
        if (OEN) // Active HI output enable (TSD enabled HI).
            DO = memory[A];
        else
            DO = 32'hZ;

endmodule
```

Single Posedge Write Level Read Ports With Write-Thru

```
//
// Simple RAM with common address, activeLO level
//   sensitive read, posedge write.
// R/W contention behavior of this example is new, in
//   other words, if write to address being read from,
//   it will write-thru to the outputs immediately.
//
module ram500x8 (wclk, ren, a, din, dout);

    parameter databits = 8;
    parameter addrbits = 9;
    parameter addrmax  = 499;

    input wclk, ren;
    input  [addrbits-1:0] a;
    input  [databits-1:0] din ;

    output [databits-1:0] dout;
    reg [databits-1:0] dout ;

    reg [databits-1:0] mymem [0:addrmax];

    event WRITE_OP; // Used to cause write-thru to always read port

    // posedge triggered write port
    always @ (posedge wclk) begin
        mymem[a] = din;
        #0; ->WRITE_OP; // Cause always read port below to awaken.
    end

    // level clocked read port, reads when "ren" LO.
    always @ (ren or a or WRITE_OP)
        if (!ren) dout = mymem[a] ;

endmodule
```

Single Posedge Ports With Separate Port Clocks

```
//
// Simple RAM with separate R/W address, separate posedge
//   read and write clocks.
//
module ram256x4 (wclk, wa, din, rclk, ra, dout);

    parameter databits = 4;
    parameter addrbits = 8;
    parameter addrmax  = (1<<addrbits) - 1;

    input wclk, rclk;
    input  [addrbits-1:0] wa, ra;
    input  [databits-1:0] din ;

    output [databits-1:0] dout;
```

```

reg [databits-1:0] dout ;

reg [databits-1:0] mymem [0:addrmax];

// write when "wclk" rises.
always @ (posedge wclk) mymem[wa] = din;

// read when "rclk" rises.
always @ (posedge rclk) dout = mymem[ra] ;

endmodule

```

Single Level Ports With Separate Port Clocks

```

//
// LIMITATION:
// Only simple variables supported inside always if (expression).
// If logical combination of enabling signals required, create
// a structural logic gate or separate continuous assign
// expression, and use that as the enabling signal.
// The following example uses a Verilog "and" gate to create such
// a signal.
//
// Posedge write, enabled by single input.
// Posedge read, enabled by AND of two inputs.
// Separate read and write addresses.
//

module ram48x4 (
    CS,    // Chip Select -- activeHI
    wclk,  // Posedge Write CLoCK
    wen,   // Write Enable -- activeHI
    wa,    // Write Address
    DI,    // write Data In
    RCLK,  // Posedge Read CLoCK
    REN,   // Read Enable -- activeHI
    RA,    // Read Address
    DO     // read Data Out
);

parameter databits = 4;
parameter addrbits = 6;
parameter addrmax  = 47;

input wclk, RCLK, wen, REN, CS;
input  [addrbits-1:0] wa, RA;
input  [databits-1:0] DI ;

output [databits-1:0] DO;
reg [databits-1:0] DO;

reg [databits-1:0] mymem [0:addrmax];

// Write when "wclk" rises if Enabled.
always @ (posedge wclk)
    if (wen) // Enable condition.
        mymem[wa] = DI;

```

```
        and u2 (read_en, CS, REN);    // Enable read only if both CS and REN HI.
        // Read when "rclk" rises if Enabled
        always @ (posedge RCLK)
            if (read_en)    // Enable condition.0-
                DO = mymem[RA];

    endmodule
```

Single Posedge Ports With Write Enable

```
//
// Separated posedge read and write clocks, separate
// read and write addresses.
// ActiveHI write enable. No read enable.
//
module ram256x8 (wclk, wen, wa, din, rclk, ra, dout);

    parameter databits = 8;
    parameter addrbits = 8;
    parameter addrmax = (1<<addrbits) - 1;

    input wclk, wen, rclk;
    input [addrbits-1:0] wa, ra;
    input [databits-1:0] din ;

    output [databits-1:0] dout;
    reg [databits-1:0] dout ;

    reg [databits-1:0] mymem [0:addrmax];

    // Write when "wclk" rises if "wen" is HI (1).
    always @ (posedge wclk)
        if (wen)
            mymem[wa] = din;

    // Read when "rclk" rises.
    always @ (posedge rclk)
        dout = mymem[ra] ;

endmodule
```

Single Level Ports With Single Read/Write Control and Bit-Blasted Wrapper

```
//
// RAM that uses a core and a wrapper to accomplish
// bit-blasted address and data pins.
//
//
// The original Verilog uses vectors. Need single bit interface.
// Can simply create bit blasted wrapper below.
// It has a shared readHI/writeLO level clock wba, but an
// independent read port address and write port address.
//
```



```

module ramcore_16x6 (
    wba, // write when wba LO, read when HI.
    wa,  // write address
    din, // data in
    ra,  // read address
    dout // data out
);

parameter databits = 6;
parameter addrbits = 4;
parameter addrmax  = (1<<addrbits) - 1;

input wba;
input [addrbits-1:0] wa, ra;
input [databits-1:0] din ;

output [databits-1:0] dout; // dout is an output register.
reg [databits-1:0] dout;

reg [databits-1:0] mymem [0:addrmax];

// wba causes a level sensitive write when LO (0),
always @ (wba or wa or din)
    if (!wba)
        mymem[wa] = din;

// wba causes a level sensitive read when HI (1).
always @ (wba or ra)
    if (wba)
        dout = mymem[ra];

endmodule

// The bit-blasted wrapper.
//
module ram16x6( DO0, DO1, DO2, DO3, DO4, DO5,
    WA0, WA1, WA2, WA3, RA0, RA1, RA2, RA3,
    DI0, DI1, DI2, DI3, DI4, DI5, WBA);
input WA0, WA1, WA2, WA3, RA0, RA1, RA2, RA3,
    DI0, DI1, DI2, DI3, DI4, DI5, WBA;
output DO0, DO1, DO2, DO3, DO4, DO5;

// Unblast (Concatenate) I/O bits into vector in portlist of ram
// instance.
// Could also use wire declaration continuous assign concatenation,
// then reference vector wire name in portlist.
ramcore_16x6 u1 (
    .wba(WBA),
    .wa( {WA3,WA2,WA1,WA0} ),
    .din( {DI5, DI4, DI3, DI2, DI1, DI0} ),
    .ra( {RA3, RA2, RA1, RA0} ),
    .dout( {DO5, DO4, DO3, DO2, DO1, DO0} ) );

endmodule

```

Single Level Ports With Read_Off 1 Output and Tristate Output Enable

```
//
// RAM that outputs Z if output disabled,
//   outputs 1 if enabled but not reading,
//   and memory contents if enabled and reading.
//   Level sensitive write and read ports.
//
module ram64x128 (
    wen,    // Write ENable (clock). activeHI level.
    wa,     // Write Address.
    din,    // Data IN.
    ren,    // Read ENable (clock). activeHI level.
    ra,     // Read Address.
    dout,   // Data OUT. Z, or 1, or a word.
    oe      // Output Enable -- activeHI
);

parameter databits = 128;
parameter addrbits = 6;
parameter addrmax  = (1<<addrbits) - 1;

input wen, ren, oe;
input  [addrbits-1:0] wa, ra;
input  [databits-1:0] din ;

output [databits-1:0] dout;
reg [databits-1:0] dout_reg ;

reg [databits-1:0] mymem [0:addrmax];

event WRITE_OP;

//
always @ (wen or wa or din) if (wen) begin
    mymem[wa] = din;
    #0; -> WRITE_OP; /* signal event */
end

always @ (ren or ra or WRITE_OP)
    if (ren) dout_reg = mymem[ra] ;
    else dout_reg = 128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

// activeHI output enable
always @ (oe or dout_reg)
    if (oe) dout = dout_reg;
    else dout = 128'bZ;

endmodule
```

Single Posedge Ports With Memory Bypass

```
//
// RAM with Data In to Data Out MUX bypass implied.
// Shared posedge clock. Separate read and write addresses.
// Separate activeHI read and write enables.
//
module bypass_RAM (
    DO,      // Data Out
    RA,      // Read Address
    WA,      // Write Address
    DI,      // Data In
    WE,      // Write Enable -- activeHI
    RE,      // Read Enable -- activeHI
    CLK,     // posedge shared clock
    BYPASS   // If 1, output DI rather than memory word.
);

output [15:0] DO;
input  [15:0] DI;
input  [3:0] RA,WA;
input  CLK, WE, RE, BYPASS;

reg [15:0] memory [0:15];

reg [15:0] DO_REG;

// Posedge clock with activeHI write enable.
always @(posedge CLK)
    if (WE)
        memory[WA] = DI;

// Posedge clock with activeHI read enable.
always @(posedge CLK)
    if (RE)
        DO_REG = memory[RA];

// 2-1 mux per data bit implied by following.
assign DO = BYPASS ? DI : DO_REG ;

endmodule
```

Single Level Bidirectional Ports With Write-Thru and Output Enable

```
//
// Bidirectional data bus ram.
// Dual Posedge read & write ports. ActiveHI chip port selects.
// ActiveHI write enables (read enabled if L0).
// ActiveHI output enables (oen_0, oen_1).
//
module bidi_dual_port_ram (clk, cs_0, cs_1, wen_0, wen_1,
    addr_0, addr_1, data_0, data_1, oen_0, oen_1);

// Warning: Do *not* change parameters in instantiations
// (not supported). Only use them for convenience of
```

```
// creating differing width modules. Create one module
// per unique physical memory (if addr_size, data_size,
// or mem_size differ), and reference the appropriate
// module name in Verilog memory instantiations of any
// memory module definition (such as this one).

parameter data_0_size = 8 ;
parameter addr_size = 8 ;
parameter mem_size = 1 << addr_size;

input clk, cs_0, cs_1, wen_0, wen_1, oen_0, oen_1;
input [addr_size-1:0] addr_0 ;
input [addr_size-1:0] addr_1 ;

inout [data_0_size-1:0] data_0 ;
inout [data_0_size-1:0] data_1 ;

reg [data_0_size-1:0] data_0_out ;
reg [data_0_size-1:0] data_1_out ;
reg [data_0_size-1:0] mem [0:mem_size-1];

// Port 0 write
always @ (posedge clk) begin
    if ( cs_0 && wen_0 ) begin
        mem[addr_0] <= data_0;
    end
end

// Port 1 write
always @ (posedge clk) begin
    if (cs_1 && wen_1) begin
        mem[addr_1] <= data_1;
    end
end

// Port 0 Read.
always @ (posedge clk) begin
    if (cs_0 && !wen_0 && oen_0) begin
        data_0_out <= mem[addr_0];
    end else begin
        data_0_out <= 0;
    end
end

// Port 1 Read.
always @ (posedge clk) begin
    if (cs_1 && !wen_1 && oen_1) begin
        data_1_out <= mem[addr_1];
    end else begin
        data_1_out <= 0;
    end
end

// If in output mode, drive memory out, else highZ. Both ports.
assign data_0 = (cs_0 && oen_0 && !wen_0) ? data_0_out : 8'bz; assign
data_1 = (cs_1 && oen_1 && !wen_1) ? data_1_out : 8'bz;

endmodule
```

Single Posedge Ports With Port and Per-Bit Write Enables and Write_Thru

```
//
// ActiveLO write enable per bit ram with write thru (single read/write
// port).
//   Port enable reads when HI, enables write when LO.
//   Writes data bit whose corresponding (LS to MS) position in the
//   write enable per bit (web) is LO, else web bit is HI and word retains
//   pre-write value in that bit.
//   If bypass is 1,
module enable_per_bit_write_thru (data_out, csb, clk,
    port_web, addr, data_in, web, bypass);

    parameter bit_size = 8;
    parameter addr_size = 9;
    parameter mem_size = 1<<addr_size;

    output [bit_size-1 : 0] data_out;

    input csb, clk, port_web, bypass;
    input [addr_size-1 : 0] addr;
    input [bit_size-1 : 0] data_in;

    input [bit_size-1 : 0] web;

    reg [bit_size-1 : 0] Mem [mem_size-1 : 0];
    reg [bit_size-1 : 0] data_outreg;

    // Note that the read and write can be in separate ports.
    always @ (posedge clk) begin
        if (csb == 1'b0) begin
            if (bypass == 1'b0) begin
                if (port_web == 1'b1) begin
                    data_outreg <= Mem[addr];
                end
            end
            else begin
                // Write only data_in bits where web is LO.
                // In Verilog, the Hold is implemented by a read then writeback.
                Mem[addr] <= (Mem[addr] & web) | (data_in & ~web);

                // Express write-thru by immediately repeating RHS expression.
                // Can also use "Mem[addr]" on right rather than expression.
                data_outreg <= (Mem[addr] & web) | (data_in & ~web);
            end
        end
    end

    // if bypass, data_out = data_in; else data_out = last value read or
    // written.
    assign data_out = bypass ? data_in : data_outreg;

endmodule
```

Dual Posedge Ports With Separate Clocks

```
//
// Ports 1, 2 Write Ports.  Ports 3, 4 Read Ports.
//   Each with its own posedge clock and own address.
//
module ram64x12 (
    w1,  // Write clock for port 1 (posedge)
    a1,  // Address for port 1
    d1,  // Data into port 1

    w2,  // Write clock for port 2 (posedge)
    a2,  // Address for port 2
    d2,  // Data into port 2

    r3,  // Read clock for port 3 (posedge)
    a3,  // Address for port 3
    d3,  // Data out from port 3

    r4,  // Read clock for port 4 (posedge)
    a4,  // Address for port 4
    d4   // Data out from port 4
);

parameter databits = 12;
parameter addrbits = 5;
parameter addrmax  = (1<<addrbits) - 1;

input w1,w2,r3,r4;
input  [addrbits-1:0] a1, a2, a3, a4;
input  [databits-1:0] d1, d2 ;

output [databits-1:0] d3, d4;
reg [databits-1:0] d3, d4 ;

reg [databits-1:0] mymem [0:addrmax];

always @ (posedge w1)
    mymem[a1] = d1;

always @ (posedge w2)
    mymem[a2] = d2;

always @ (posedge r3)
    d3 = mymem[a3] ;

always @ (posedge r4)
    d4 = mymem[a4] ;

endmodule
```

Dual Posedge Write Level Read Ports With Separate Clocks

```
//
// RAM with dual READ/WRITE ports. Posedge write, level
//      sensitive read. All 4 ports have their own
//      clock and address.
//      If either Write Port writes to address being read
//      by a Read Port, write-thru new data to output
//      (uses event WRITE to do this).
//
module ram64x6 (
    w1, // posedge Write clock for port 1.
    a1, // write Address for port 1
    d1, // Data into port 1

    w2, // posedge Write clock for port 2.
    a2, // write Address for port 2.
    d2, // Data into port 2

    r3, // level sensitive Read clock for port 3.
    a3, // read Address for port 3
    d3, // Data out from port 3

    r4, // level sensitive Read clock for port 4.
    a4, // read Address for port 4
    d4  // Data out from port 4
);

parameter databits = 6;
parameter addrbits = 6;
parameter addrmax  = (1<<addrbits) - 1;

input w1,w2,r3,r4;
input [addrbits-1:0] a1, a2, a3, a4;
input [databits-1:0] d1, d2 ;

output [databits-1:0] d3, d4;
reg [databits-1:0] d3, d4 ;

reg [databits-1:0] mymem [0:addrmax];

event WRITE;

// Port 1 posedge Write port.
always @ (posedge w1) begin
    mymem[a1] = d1;
    #0; ->WRITE; // Cause write-thru if appropriate.
end

// Port 2 posedge Write port.
always @ (posedge w2) begin
    mymem[a2] = d2;
    #0; ->WRITE; // Cause write-thru if appropriate
end

// Port 3 level sensitive Read port.
```

```
always @ (r3 or a3 or WRITE)
  if (r3) d3 = mymem[a3] ;

// Port 4 level sensitive Read port.
always @ (r4 or a4 or WRITE)
  if (r4) d4 = mymem[a4] ;

endmodule
```

Dual Posedge Write Level Read Ports With Separate Clocks and Tristate Output Enable

```
//
// Falling edge sensitive write ports, level sensitive
//      read ports, separate tristate output, Chip Select
//
module ram50x20 (w1,a1,d1, w2,a2,d2, r3,a3,d3,oe3, r4,a4,d4,oe4, cs);

  parameter databits = 20;
  parameter addrbits = 6;
  parameter addrmax  = 49;

  input w1,w2,r3,oe3,r4,oe4, cs;
  input [databits-1:0] d1, d2;
  input [addrbits-1:0] a1, a2, a3, a4;

  output [databits-1:0] d3, d4;
  reg [databits-1:0] d3, d3_reg, d4, d4_reg;

  reg [databits-1:0] mymem [0:addrmax];
  event WRITE;

  /* internal control logic terms */

  and u1 (readena1, cs, !r3);
  and u2 (readena2, cs, !r4);
  and u3 (outena1,  cs, !oe3);
  and u4 (outena2,  cs, !oe4);

  /* write ports, edge sensitive active high */

  always @(posedge w1) if (cs) begin
    mymem[a1] = d1; #0; ->WRITE; end
  always @(posedge w2) if (cs) begin
    mymem[a2] = d2; #0; ->WRITE; end

  /* read ports, level sensitive */

  always @(readena1 or a3 or WRITE)
    if (readena1) d3_reg = mymem[a3];
  always @(readena2 or a4 or WRITE)
    if (readena2) d4_reg = mymem[a4];

  /* output enables, qualified by chip select */

  always @(outena1 or d3_reg)
```



```

        if (outena1) d3 = d3_reg; else d3 = 20'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZZ ;
    always @(outena2 or d4_reg)
        if (outena2) d4 = d4_reg; else d4 = 20'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZZ ;

endmodule

```

Dual Level Ports With Separate Clocks and Write-Thru

```

//
// RAM with non-default contention behavior
//
module ram252x7 (w1,a1,d1, w2,a2,d2, r3,a3,d3, r4,a4,d4);

    parameter addrbits = 8;
    parameter addrmax  = 251;
    parameter databits = 7;

    input w1,w2,r3,r4;
    input [addrbits-1:0] a1, a2, a3, a4;
    input [databits-1:0] d1, d2;

    output [databits-1:0] d3, d4;
    reg [databits-1:0] d3, d4;

    reg [databits-1:0] mymem [0:addrmax];

    event WRITE_OP;

    // Level sensitive Write port 1
    always @ (w1 or a1 or d1)
        if (w1) begin
            mymem[a1] = d1;
            #0; ->WRITE_OP; /* signal event */
        end

    // Level sensitive Write port 2
    always @ (w2 or a2 or d2)
        if (w2) begin
            mymem[a2] = d2;
            #0; ->WRITE_OP; /* signal event */
        end

    // Level sensitive Read port 3
    always @ (r3 or a3 or WRITE_OP)
        if (r3)
            d3 = mymem[a3] ;

    // Level sensitive Read port 4
    always @ (r4 or a4 or WRITE_OP)
        if (r4)
            d4 = mymem[a4] ;

endmodule

```

Dual Posedge Bidirectional Ports With Read_Off 0 and Output Enable

```
//
// Bidirectional data bus ram.
// Asynchronous read and write.
// If chip selected then :
// if wen HI writes, else if oen HI reads.
//

module bidi_ram (cs, wen, addr, data, oen);

// Warning: Do *not* change parameters in instantiations
// (not supported). Only use them for convenience of
// creating differing width modules. Create one module
// per unique physical memory (if addr_size, data_size,
// or mem_size differ), and reference the appropriate
// module name in Verilog memory instantiations of any
// memory module definition (such as this one).

parameter addr_size = 8 ;
parameter data_size = 8 ;
parameter mem_size = 1 << addr_size;

input cs, wen, oen;
input [addr_size-1:0] addr;

inout [data_size-1:0] data ;

reg [data_size-1:0] data_out ;
reg [data_size-1:0] mem [0:mem_size-1];

// The data being output from the core to the PAD (outside world) thru //
// Level sensitive write.
always @ (addr or data or cs or wen)
begin
    if ( cs && wen ) begin // if writing/input mode..
        mem[addr] = data;
    end
end

// Level sensitive read.
always @ (addr or cs or wen or oen)
begin
    if (cs && !wen && oen) begin // if output mode..
        data_out = mem[addr];
    end
end

// If output mode, drive bidi data port, else shut off drivers.
assign data = (cs && !wen && oen) ? data_out : 'bz; //

endmodule
```

Dual Posedge Separate Clocks Ports With Port and Per-Bit Write Enables

```
//
// ActiveLO write Enable per Bit ram. Dual read / Dual write.
// Separate posedge port clocks. Port enables write LO/ read HI.
// Write enable per bit (web) controls which bits written when
// writing. Bits of word where web LO are written, else hold.

module dual_port_enable_per_bit_ram ( Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1, web_0, web_1,
    bypass_0, bypass_1);

    parameter bit_size = 6;
    parameter addr_size = 8;
    parameter mem_size = 1<<addr_size;

    output [bit_size-1 : 0] Dout_0;
    output [bit_size-1 : 0] Dout_1;

    input csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, bypass_0, bypass_1;

    input [addr_size-1 : 0] A_0;
    input [addr_size-1 : 0] A_1;

    input [bit_size-1 : 0] Din_0;
    input [bit_size-1 : 0] Din_1;

    input [bit_size-1 : 0] web_0;
    input [bit_size-1 : 0] web_1;

    reg [bit_size-1 : 0] Dout_0_reg;
    reg [bit_size-1 : 0] Dout_1_reg;

    reg [bit_size-1 : 0] Mem [mem_size-1 : 0];
    reg [bit_size-1 : 0] Qreg_0, Qreg_1;

    // Note that the Dout_0_reg can be in a separate always block.
    // Also, the Port0 read and write can be in separate blocks.
    always @ (posedge ck_0) begin
        Dout_0_reg <= Qreg_0; // Always update port's out pipe.
        if (csn_0 == 1'b0) begin // ActiveLO chip select
            if (bypass_0 == 1'b0) begin // ActiveHI memory bypass.
                if (wen_0 == 1'b1) begin // ActiveHI port read enable.
                    Qreg_0 <= Mem[A_0]; // Port 0 read.
                end
            end
            else begin // ActiveLO port write enable.
                // web_0 has one activeLO enable bit per data bit.
                // Following writes Din if corresponding bit is LO,
                // else it holds old value (reads then writes back).
                Mem[A_0] <= (Mem[A_0] & web_0) | (Din_0 & ~web_0);
            end
        end
    end
end
```

```
// If bypass, Dout = Din, else Dout = piped/registered mem out.
assign Dout_0 = (bypass_0) ? Din_0 : Dout_0_reg;

// Port 1 is exactly like above port 0. See comments above.
always @ (posedge ck_1) begin
    Dout_1_reg <= Qreg_1;
    if (csn_1 == 1'b0) begin
        if (bypass_1 == 1'b0) begin
            if (wen_1 == 1'b1) begin
                Qreg_1 <= Mem[A_1];
            end
        end
        else begin
            Mem[A_1] <= (Mem[A_1] & web_1) | (Din_1 & ~web_1);
        end
    end
end

assign Dout_1 = (bypass_1) ? Din_1 : Dout_1_reg;

endmodule
```

Using Positional Parameter Overrides to Modify Above RAM

```
// Rather than create a different module for each different data and/or
// word size of an otherwise identical RAM, positional overrides can be
// used and are supported in LibComp. The following modifies the previous
// write enable per bit RAM to be two different sizes than the original
// defining module, but otherwise they operate identically due to reuse
// of the same Verilog functionality.
// Because there is no explicit defining model as required by ATPG, such
// a model is created and referenced in the translation. This requires
// uniquifying the original module name. The changed parameters and
// their new value are used to create a unique module name for each
// different size of module encountered due to parameter overrides.
// Unchanged parameters are not used in the uniquification of the name.
// CAVEAT: Only positional parameter overrides are supported. Named
// parameter overrides remain unsupported at this time.

module dual_port_enable_per_bit_ram_128x72
    (Dout_0, Dout_1, csn_0, csn_1,
     ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1, web_0, web_1,
     bypass_0, bypass_1);

    parameter bit_size = 72;
    parameter addr_size = 7;
    parameter mem_size = 1<<addr_size;
    output [bit_size-1 : 0] Dout_0;
    output [bit_size-1 : 0] Dout_1;

    input csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, bypass_0, bypass_1;

    input [addr_size-1 : 0] A_0;
    input [addr_size-1 : 0] A_1;
```

```

input [bit_size-1 : 0] Din_0;
input [bit_size-1 : 0] Din_1;

input [bit_size-1 : 0] web_0;
input [bit_size-1 : 0] web_1;

// Note that positional overrides used to change size of ram instantiated.
// Named parameter overrides are not supported.
// Defining module is shown in prior example, and must have the number of
// bits as the first parameter, and the number of words as the second.

dual_port_enable_per_bit_ram #(bit_size, addr_size) override_inst(Dout_0,
    Dout_1, csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0,
    Din_1, web_0, web_1, bypass_0, bypass_1);

endmodule

// LibComp output excerpt from above Verilog module showing name
// unification, and redeclaration of array sizes.

model dual_port_enable_per_bit_ram_128x72 (Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1, web_0, web_1,
    bypass_0, bypass_1)
(
    model_source = verilog_module;
    input (csn_0) ( )
    input (csn_1) ( )
    ...
    output (Dout_1) (array = 71 : 0; instance =
        mlc_dual_port_enable_per_bit_ram_bit_size_72_addr_size_7
        override_inst (.Dout_0(Dout_0), .Dout_1(Dout_1), .csn_0(csn_0),
        .csn_1(csn_1), .ck_0(ck_0), .ck_1(ck_1), .wen_0(wen_0), .wen_1(wen_1),
        .A_0(A_0), .A_1(A_1), .Din_0(Din_0), .Din_1(Din_1), .web_0(web_0),
        .web_1(web_1), .bypass_0(bypass_0), .bypass_1(bypass_1) );
    )
)

model mlc_dual_port_enable_per_bit_ram_bit_size_72_addr_size_7 (Dout_0,
    Dout_1, csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1,
    web_0, web_1, bypass_0, bypass_1)
(
    model_source = verilog_parameter_override;
    intern (Qreg_0) (array = 71 : 0;)

    ...
    input (A_0) (array = 6 : 0;)
    input (A_1) (array = 6 : 0;)
    input (Din_0) (array = 71 : 0;)
    ...
    primitive = _cram Mem_0 ( , ,
    ...
    primitive = _cram Mem_71 ( , ,
        // Following write port will Hold in-memory data when not writing.
        _write { , , } (ck_1, mlc_and_5[71], A_1, Din_1[71]),
        // Following write port will Hold in-memory data when not writing.
        _write { , , } (ck_0, mlc_and_6[71], A_0, Din_0[71]),
        // Following read port will Hold output data after reading.

```

```
        _read { ,H,H,H} ( , ck_1, mlc_and_3, A_1, Qreg_1[71]),
        // Following read port will Hold output data after reading.
        _read { ,H,H,H} ( , ck_0, mlc_and_1, A_0, Qreg_0[71])
    );
)
)

// Same defining module as above but RAM is different size. If it had
// been same override values (same size), then above ATPG remodel would
// have been reused. Only one model created per unique set of parameters
// of the defining module.

module dual_port_enable_per_bit_ram_32x8 ( Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1, web_0, web_1,
    bypass_0, bypass_1);

    parameter bit_size = 8;
    parameter addr_size = 5;
    parameter mem_size = 1<<addr_size;

    output [bit_size-1 : 0] Dout_0;
    output [bit_size-1 : 0] Dout_1;

    input csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, bypass_0, bypass_1;

    input [addr_size-1 : 0] A_0;
    input [addr_size-1 : 0] A_1;

    input [bit_size-1 : 0] Din_0;
    input [bit_size-1 : 0] Din_1;

    input [bit_size-1 : 0] web_0;
    input [bit_size-1 : 0] web_1;

    // Note that positional overrides used to change size of ram instantiated.
    // Named parameter overrides are not supported.
    // Defining module is shown in prior example, and must have the number of
    // bits as the first parameter, and the number of words as the second.

    dual_port_enable_per_bit_ram #(bit_size, addr_size)
        override_inst(Dout_0, Dout_1, csn_0, csn_1, ck_0, ck_1, wen_0, wen_1,
            A_0, A_1, Din_0, Din_1, web_0, web_1, bypass_0, bypass_1);

endmodule

// LibComp output excerpt from above Verilog module showing name
// uniquification, and redeclaration of array sizes.
// This is remodel of uses above uniquified model for ATPG purposes.

model dual_port_enable_per_bit_ram_32x8
(Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1,
    A_0, A_1, Din_0, Din_1,
    web_0, web_1, bypass_0, bypass_1)
(
    model_source = verilog_module;
    input (csn_0) ( )
    input (csn_1) ( )
```

```

...
output (Dout_1) (array = 7 : 0;
    instance = mlc_dual_port_enable_per_bit_ram__bit_size_8_addr_size_5
        override_inst (.Dout_0(Dout_0), .Dout_1(Dout_1), .csn_0(csn_0),
            .csn_1(csn_1), .ck_0(ck_0), .ck_1(ck_1), .wen_0(wen_0), .wen_1(wen_1),
            .A_0(A_0), .A_1(A_1), .Din_0(Din_0), .Din_1(Din_1),
            .web_0(web_0), .web_1(web_1), .bypass_0(bypass_0), .bypass_1(bypass_1)
        );
    )
)

model mlc_dual_port_enable_per_bit_ram__bit_size_8_addr_size_5
    (Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1,
    A_0, A_1, Din_0, Din_1,
    web_0, web_1, bypass_0, bypass_1)
(
    model_source = verilog_parameter_override;
    intern (Qreg_0) (array = 7 : 0;)
    intern (Dout_0_reg) (array = 7 : 0;)
    ...
    input (A_0) (array = 4 : 0;)
    input (A_1) (array = 4 : 0;)
    input (Din_0) (array = 7 : 0;)
    ...
    primitive = _cram Mem_0 ( , ,
    ...
    primitive = _cram Mem_7 ( , ,
        // Following write port will Hold in-memory data when not writing.
        _write { , , } (ck_1, mlc_and_5[7], A_1, Din_1[7]),
        // Following write port will Hold in-memory data when not writing.
        _write { , , } (ck_0, mlc_and_6[7], A_0, Din_0[7]),
        // Following read port will Hold output data after reading.
        _read { ,H,H,H} ( , ck_1, mlc_and_3, A_1, Qreg_1[7]),
        // Following read port will Hold output data after reading.
        _read { ,H,H,H} ( , ck_0, mlc_and_1, A_0, Qreg_0[7])
    );
)
)

```

Dual Posedge Separate Clocks Ports With Port and Per-Byte Write Enables

```

//
// ActiveLO write Enable per Byte ram. Dual read / Dual write.
// Separate posedge port clocks. Port enables write LO/ read HI.
// Write enable per byte (web) controls which bytes written when
// writing. Bytes of word where corresponding web is LO are written,
// while bytes whose web is HI hold previously written value.

module dual_port_enable_per_byte_ram ( Dout_0, Dout_1, csn_0, csn_1,
    ck_0, ck_1, wen_0, wen_1, A_0, A_1, Din_0, Din_1, web_0, web_1,
    bypass_0, bypass_1);

    output [31 : 0] Dout_0;
    output [31 : 0] Dout_1;

```

```
input csn_0, csn_1, ck_0, ck_1, wen_0, wen_1, bypass_0, bypass_1;

input [9:0] A_0;
input [9:0] A_1;

input [31:0] Din_0;
input [31:0] Din_1;

input [3:0] web_0;
input [3:0] web_1;

reg [31:0] Dout_0_reg;
reg [31:0] Dout_1_reg;

reg [31:0] Mem [1023 : 0];
reg [31:0] Qreg_0, Qreg_1;

// Expand to enable per bit, so can express writes as Boolean
// equation below. Each web bit controls a byte (8 bits) of
// data -- so replicate each 8 times to create enable per bit.
wire [31:0] web_0_int = { {8{web_0[3]}}, {8{web_0[2]}}, {8{web_0[1]}},
{8{web_0[0]}} };
wire [31:0] web_1_int = { {8{web_1[3]}}, {8{web_1[2]}}, {8{web_1[1]}},
{8{web_1[0]}} };

// Note that the Dout_0_reg output pipe register can be in a separate
always block.
// Also, the Port0 read and write can be in separate blocks.
always @ (posedge ck_0) begin
    Dout_0_reg <= Qreg_0; // Always update port's out pipe.
    if (csn_0 == 1'b0) begin // ActiveLO chip select
        if (bypass_0 == 1'b0) begin // ActiveHI memory bypass.
            if (wen_0 == 1'b1) begin // ActiveHI port read enable.
                Qreg_0 <= Mem[A_0]; // Port 0 read.
            end
        else begin // ActiveLO port write enable.
            // web_0_int has one activeLO enable bit per data bit.
            // Following writes Din if corresponding bit is LO,
            // else it holds old value (reads then writes back).
            Mem[A_0] <= (Mem[A_0] & web_0_int) | (Din_0 & ~web_0_int);
        end
    end
end
// If bypass, Dout = Din, else Dout = piped/registered mem out.
assign Dout_0 = (bypass_0) ? Din_0 : Dout_0_reg;

// Port 1 is exactly like above port 0. See comments above.
always @ (posedge ck_1) begin
    Dout_1_reg <= Qreg_1;
    if (csn_1 == 1'b0) begin
        if (bypass_1 == 1'b0) begin
            if (wen_1 == 1'b1) begin
                Qreg_1 <= Mem[A_1];
            end
        end
    end
end
```



```
        else begin
            Mem[A_1] <= (Mem[A_1] & web_1_int) | (Din_1 & ~web_1_int);
        end
    end
end
end
end

assign Dout_1 = (bypass_1) ? Din_1 : Dout_1_reg;

endmodule
```


Chapter 5

Verifying Tessent Simulation Models

This chapter describes how to verify the functionality of the Tessent Cell libraries and provides a few guidelines to improve the quality of your Tessent simulation models. This chapter includes the following topics:

Verification Overview	259
Specifying Which Tool Performs Verification	260
Verification Prerequisites	260
Running Verification from the Shell	260
Interpreting the Verification Results	261
Debugging Models	264
Re-simulating Verilog Only	265
Fixing DRC Violations	266
Improving Test Coverage	266
Modeling for Optimal Test Coverage	268

Verification Overview

Functionality verification of the Tessent Cell libraries consists of simulating and testing the library models, and comparing these models to the Verilog source modules to confirm parallel functionality. When the functionality does not match, the simulation model fails verification, and the application returns simulation mismatches for the failing model.

i **Tip:** You should verify functionality of the simulation models when you generate the libraries with LibComp (see “[Using LibComp to Create Tessent Simulation Models](#)”), or when you manually edit or add new models to an existing library.

By default, LibComp runs verification as it generates simulation models. You can also run verification on an existing simulation models from a UNIX/Linux shell.

A utility, lcVerify, performs the verification of the simulation models using the following steps:

1. Uses Tessent FastScan to generate and simulate test patterns for the simulation models.
2. Uses ModelSim® to simulate the Verilog source library using the same Tessent FastScan test patterns.

3. Compares the simulation results of the simulation models and the Verilog source and outputs a logfile detailing simulation mismatches and statistics you can use to improve the testability and performance of the simulation models.

To ensure robust simulation models, you should correct all simulation mismatches and raise test coverage to as close to 100% as possible.

Specifying Which Tool Performs Verification

By default, Tessent FastScan performs the verification. You can change which tool is used for verification using the following utilities:

- **LibComp** — To change the verification tool LibComp uses, use the [Set Verification](#) command in LibComp to specify FlexTest or Tessent TestKompress.
- **lcVerify** — To use Tessent FastScan for verification, use the *lcVerify* command from Linux/UNIX shell. You specify FlexTest or Tessent TestKompress during lcVerify invocation. Tessent FastScan is the default.

Verification Prerequisites

- Verilog source library must be available.
- Tessent FastScan or other specified verification tool must be available.
- ModelSim must be available.
- LibComp must be used to generate the initial simulation models. LibComp generates setup files required by the lcVerify utility. The LibComp and lcVerify utilities used should be from the same software release.

Running Verification from the Shell

When running verification from a shell, you should reuse the same verification arguments that LibComp uses when verifying a library. LibComp outputs how the tool invokes the verification as a comment in the *transcript/log_file* and to stdout.

For example, if you invoked LibComp using the following syntax:

```
$Tessent_Tree_Path/bin/libcomp design_verify.v -dofile -log my_log.log \
-replace
```

then LibComp outputs the following invocation line in the *my_log.log* log file:

```
// Verifying ATPG Library using Invocation :
// $Tessent_Tree_Path/bin/lcVerify -no_scan_rams -no_atpg_prims
// tessent.mtCellLib
```

```
// design_verify.v
```

Verifying a Single Simulation Model

By default, the verification process validates all simulation models in a simulation library. In some cases, you might want to verify only a single model, for example verifying a simplified view of a memory module using LibComp with internal hierarchical modules that do not match this simplified view.

You direct lcVerify to verify a single model in a library using the following invocation syntax:

```
Tessent_Tree_Path/bin/lcVerify my_dft_library.atpg my_verilog_source.v \  
-model model_name
```

In the ATPG tool you invoke with the -model switch, you can also use the [report_statistics](#) command's -model switch to report the statistics for this single named model.

Interpreting the Verification Results

The verification process produces the following results files in the simulation library parent directory:

- **verify.results** — Summary of the following key statistics for the run:
 - Simulation mismatches or the differences between the values Tessent FastScan simulated for the simulation library and the values simulated for the original Verilog library by ModelSim
 - DRC violations
 - Fault and test coverage
- **sim.log** — Full transcript of the Tessent FastScan and ModelSim runs
- **transcript** — Transcript of the ModelSim run

verify.results File Example

You should review this file first to identify simulation models with low coverage, DRC violations, and simulation mismatches.



Note

There are two metrics on the right side of the statistics -- first the collapsed statistics (coll) and then the full or total statistics (total). Only the (total) numbers should be used for coverage assessment. The (total) numbers accurately reflect true coverage for single instances of smaller modules defined in libraries.

The following example shows the contents of a *verify.results* file.

```
Library Verification Run
Verifying tessent.mtCellLib
Run at Wed Oct  1 12:29:23 2008
```

```
    The following 1 Models were Completely BlackBoxed:
    nonsense_model
```

```
-----
Summary Statistics for Library tessent.mtCellLib
-----
```

```
Fault Statistics for Library tessent.mtCellLib
-----
```

fault class	#faults (coll.)	#faults (total)
FU (full)	22	22
DS (det_simulation)	12	12
DI (det_implication)	1	1
PT (posdet_testable)	1	1
UU (unused)	6	6
AU (atpg_untestable)	2	2
test_coverage	81.25%	81.25%
fault_coverage	59.09%	59.09%
atpg_effectiveness	95.45%	95.45%

```
-----
Test Pattern Statistics for Library tessent.mtCellLib
-----
```

#test_patterns	6
#clock_sequential_patterns	6
#simulated_patterns	6

```
-----
Model : almost_xor_dff_no_controls
-----
```

```
Fault Statistics for instance almost_xor_dff_no_controls
-----
```

fault class	#faults (coll.)	#faults (total)	almost_xor_dff_no_controls
FU (full)	6	6	almost_xor_dff_no_controls
DS (det_simulation)	5	5	almost_xor_dff_no_controls
PT (posdet_testable)	1	1	almost_xor_dff_no_controls
test_coverage	83.33%	83.33%	almost_xor_dff_no_controls
fault_coverage	83.33%	83.33%	almost_xor_dff_no_controls
atpg_effectiveness	83.33%	83.33%	almost_xor_dff_no_controls

```
-----
Model : nonsense_model
-----
```

Fault Statistics for instance nonsense_model

fault class	#faults (coll.)	#faults (total)	nonsense_model
FU (full)	8	8	nonsense_model
UU (unused)	6	6	nonsense_model
AU (atpg_untestable)	2	2	nonsense_model
test_coverage	0.00%	0.00%	nonsense_model
fault_coverage	0.00%	0.00%	nonsense_model
atpg_effectiveness	100.00%	100.00%	nonsense_model

Model : xor_dff_no_controls

Fault Statistics for instance xor_dff_no_controls

fault class	#faults (coll.)	#faults (total)	xor_dff_no_controls
FU (full)	8	8	xor_dff_no_controls
DS (det_simulation)	7	7	xor_dff_no_controls
DI (det_implication)	1	1	xor_dff_no_controls
test_coverage	100.00%	100.00%	xor_dff_no_controls
fault_coverage	100.00%	100.00%	xor_dff_no_controls
atpg_effectiveness	100.00%	100.00%	xor_dff_no_controls

***** Fault (pessimistic) Coverage Summary by Decile *****

See file fault_coverage_0_to_10_percent_models for a list of models in 0 to 10% decile, etc. for each nonNULL decile.

0% to 10% --- 1 models.
80% to 90% --- 1 models.
90% to 100% -- 1 models.

Verification Summary:

3 Total Models

ALL PASSED for all patterns.

All known model output values predicted by ATPG agreed with Verilog sim. In some cases, this only means ATPG could not exercise the model. See below.

Always check transcript for model translation messages.

Always check the output ATPG library file with remodels for the strings:

"BlackBox", "EDIT & place _cram", and "PARTIALLY TRANSLATED MODULE".

Always check sim.log for untestable Ram & DRC messages.

Always check low coverage models to see if low coverage is expected

(IO pads may have 40% coverage and be good, whereas 85% may be bad for a mux scan model).

See the Fault Coverage Summary above these messages and associated

files for the model names with low coverage. For model specific coverages, check the `sim.log` file. Look for " *** FINAL COVERAGE STATISTICS ***" in `sim.log` to find overall coverage. Also shown beside "Summary Statistics for Library". in screen/transcript output. Module specific (`-instance`) reports follow overall.

The following 1 Models were Completely BlackBoxed:
`nonsense_model`

Debugging Models

Be aware that simulation mismatches can be caused by a number of errors.

Review the `verify.results` file and note the names of the models that failed verification and the cause of the failure, and then, use the information in the following table to debug the models.

Table 5-1. Debugging Models

Symptom	Possible Solution
Tessent FastScan is unable to read a model.	Fix the model or comment it out.
Duplicate modules in the Verilog library.	Eliminate the duplicates or change their names.
ModelSim compiler (vlog) cannot compile a model.	This is usually due to a Verilog syntax error or modeling issue. A transcript of the compiler's run is recorded in the <code>sim.log</code> file and typically contains enough information (line numbers and brief descriptions of errors) for you to start debugging the Verilog.
The ModelSim simulator (vsim) cannot successfully load a model	This is usually due to a Verilog issue. Refer to the transcript of the simulators run in the <code>sim.log</code> file for debugging information. Check the vlog transcript to ensure the model(s) compiled without errors; compile errors will often result in load errors.
Verilog source is a Sequential UDP and the ATPG model is correct, but the verification still fails.	The cause of the failure is very likely a missing <code>add_clocks</code> definition for the clock input or a missing <code>add_input_constraints</code> for a notifier input in the <code>fastscan.do.cat</code> file. Correct the definition in the <code>fastscan.do.cat</code> file and rerun verification.
Sequential Verilog UDP seems to be a valid Latch or DFF, but LibComp black boxes it.	Search the LibComp transcript for HOLD CHECK messages. The Hold Check message is output when a UDP modeling a latch or D Flip Flop fails to compile because of a minor error. When such a UDP is encountered, the HOLD CHECK message is output to the transcript followed by a description of what LibComp needs to successfully compile the model.

For more troubleshooting information, see the “[Potential Causes of Simulation Mismatches](#)” section of the *Tessent Scan and ATPG User’s Manual*.

Re-simulating Verilog Only

To troubleshoot Verilog simulation issues, you can simulate just the Verilog portion of the verification with the `<Tessent_Tree_Path>/bin/run_verify` script. The `run_verify` script contains the commands and arguments to perform just the ModelSim portion of verification.

See the following topics for more information on using `run_verify` to simulate the Verilog source library:

- [Prerequisites for Simulating Verilog Only](#)
- [Simulating Verilog Only](#)

Prerequisites for Simulating Verilog Only

The following prerequisites must be satisfied before you can use `run_verify` to simulate the Verilog source library:

- ModelSim must be available.
- lcVerify must be run initially on the Verilog source and simulation models. For more information, see “[Assessing the Impact of Low Coverage](#)”.
- Verilog source library must be available.
- lcVerify must be run using the `-save_vsim` switch, even if LibComp has been run earlier. This is necessary to preserve the files needed for Verilog simulation. By default, these simulation files are not saved by LibComp in lcVerify runs.

Simulating Verilog Only

The LibComp and lcVerify scripts output the Tessent FastScan invocation line in the `sim.log` verification output file. You can cut and paste this invocation line and substitute the `<mgc_dft_tree>` with your `Tessent_Tree_Path` to reinvoked only the Verilog simulation portion of a verification run and analyze coverage only. You can find this invocation line in the `sim.log` after the ATPG section and before the Verilog simulation section.

The following shows an example Verilog simulation invocation line from the `sim.log`:

```
# Note: Invoking vsim with :  
#      vsim -novopt MGC_DFT_LIB_ALL_pat_v_ctl -t 1ns -c -do  
#      "run -all; quit" +nospecify +nowarnTSCALE
```

When copying and pasting this invocation, ensure you omit the leading pound sign (#).

Fixing DRC Violations

You can find detailed information about the DRC violations in your simulation models in the *sim.log* file. The *sim.log* file contains a transcript of the Tessent FastScan run including the DRC messages. These messages usually include a DRC identification number.

You may decide a particular DRC violation is acceptable, but your decision should be based on an understanding of the violation and its effect on the testability of the model and the library.

Improving Test Coverage

Test coverage loss due to ineffective models may hinder the test coverage for any design that uses the library. A maximum attainable coverage for a design can only be achieved if a library has maximum attainable coverage. It is normal for IO pad model coverage to be much lower than UDPs, so maximum attainable coverage is relative to the type of models being tested.

The *results.verify* file lists the overall fault statistics for the simulation library. If the library was generated using LibComp, V8.2003_1.10 or later, the *results.verify* file also lists the fault statistics individually for each model in the library. In this case, the first step in troubleshooting low coverage should be to determine from this list which models have less than desirable coverage. For example, if most IO pad models have 40% to 60% coverage any models with only 20% coverage would be suspect.

Troubleshooting One Model at a Time

As an aid in troubleshooting, split out low coverage models into their own files. One way to do this is to open the simulation library in a text editor, and copy and paste each model description of interest into a new text editing window and save it as a file. Be sure to use a naming convention for the individual model files that indicates what each contains. For example, *model_name.atpg*. When each low coverage model is in its own file, you can focus your troubleshooting efforts on one model at a time.

Run Tessent FastScan on each model file in turn, using the same commands used in the earlier lcVerify run.

Assessing the Impact of Low Coverage

If you cannot raise coverage for a particular model in your library, use the UNIX **grep** command to find out how many instances of the model are used in your design(s). If there are relatively few instances, the impact of the low coverage model on the design's overall coverage may be low enough to ignore.

Locating Low-Coverage Models

After verification, you can find information concerning low-coverage models either at the end of the transcript or log, or the *verify.results* file in a Fault Coverage Summary. This summary is useful to locate low-coverage models in large libraries without searching the *sim.log* file manually.

The following is an example Fault Coverage Summary:

```
***** Fault (pessimistic) Coverage Summary by Decile *****
See file fault_coverage_0_to_10_percent_models for a list
of models in 0 to 10% decile, etc. for each nonNULL decile.
0% to 10% --- 2 models.
60% to 70% --- 1 models.
70% to 80% --- 4 models.
80% to 90% --- 20 models.
90% to 100% -- 705 models.
-----
-----
```

Additionally, at the end you can find a list of BlackBoxes, if any, which will explain some 0 percent coverages as in the following example:

```
The following 2 Models were Completely BlackBoxed:
mxsdprbs1q
mxsdprbs2q
```

The tool also writes out files (one per listed decile in the Fault Coverage Summary) in the results directory containing this information as in the following example:

```
fault_coverage_0_to_10_percent_models
fault_coverage_80_to_90_percent_models
fault_coverage_60_to_70_percent_models
fault_coverage_90_to_100_percent_models
fault_coverage_70_to_80_percent_models
```

Each file has a list of the modules/models within that coverage decile. The following example shows the contents for the *fault_coverage_70_to_80_percent_models* file:

```
List of all 4 models with this coverage decile.
mxiao31x1a_UDPOB
glat
mxdprbsblqb
mxdprbsb2qb
```

Re-running the Tessent FastScan Portion of Verification

The LibComp and lcVerify scripts output the Tessent FastScan invocation line at the top of the *sim.log* verification output file. You can cut and paste this invocation line and substitute the

`<mgc_dft_tree>` with your *Tessent_Tree_Path* to reinvoke only the ATPG portion of a verification run and analyze coverage only.

The following shows an example invocation line from the *sim.log*:

```
# Note: Invoking fastscan with :  
# <Tessent_Tree_Path>/bin/fastscan -dof fastscan.do.cat -lib  
# tessent.mtCellLib -load_warnings -sensitive -scan_rams -model all
```

When copying and pasting this invocation, ensure you omit the leading pound sign (#).

Modeling for Optimal Test Coverage

Typically, you want the test coverage to be as high as possible. Also, the simulation library should not have any models that lcVerify is unable to process. See the following topics for recommended practices to achieve high coverage, efficient simulation models:

Handling Ignored or Blackboxed Models

Due to the variety of design configurations possible with UDPs, there are UDPs that LibComp cannot process. Because black box outputs are tied to X, they generally result in some AU faults during ATPG.

To create valid models, you must manually convert UDP models that are blackboxed or ignored by LibComp. For more information on creating simulation models manually, see “[Defining Cell Information](#)”.

Anticipating the Effects of Internal Gating on Clocks

Be aware, when you define clocks using the `add_clocks` command, that the tool understands the off state you specify to be the clock’s off value at a primary input to the model. If there is gating logic between this input and an instance within the model, take care that the off state you specify produces the desired off state on the input to the internal instance after passing through the logic.

Chapter 6

Using ETLibCertify in the LV Flow

The ETLibCertify tool enables you to certify a set of scan models together with their associated *scang.lib*, *pad.library*, and *cell.library* files. These scan models and library files can be generated manually, with LibComp or any third-party tool. The certification process is done without the use of Liberty files.

This chapter follows this sequence:

Certification Process Overview	269
Using ETLibCertify	269
<technology_name>.etlibcertify Configuration File	270
Certification Process	271
Example of Using ETLibCertify	274

Certification Process Overview

ETLibCertify verifies the correctness of a set of scan models together with their associated *scang.lib*, *pad.library*, and *cell.library* library files.

The certification process verifies that:

- Rules checking can be performed on the scan models.
- The library files are sufficiently populated for basic use.
- The scan models are compatible with the Verilog simulation models of the cells.

Using ETLibCertify

To invoke *ETLibCertify*, you must provide a configuration file — [*<technology_name>.etlibcertify Configuration File*](#). For example, if you have a configuration file with the name *Dolphin.etlibcertify*, the following command line instructs ETLibCertify to create certification workspaces for the Dolphin technology:

```
etlibcertify Dolphin
```

The configuration file specifies the input data (files and/or directories) for which ETLibCertify must create the certification workspace(s) — *Verilog Simulation models*.

ETLibCertify will invoke Tessent FastScan to convert a *TessentCellLib* file into a single *ScanModelFile*.

<technology_name>.etlibcertify Configuration File

The input to ETLibCertify is a single technology-specific configuration file. The root name of the configuration file must match the name of the technology and have the suffix *.etlibcertify* attached.

The configuration file specifies the files and/or directories for which ETLibCertify must create certification workspace(s).

Complete syntax for *.etlibcertify* is provided in [Figure 6-1](#). For detailed file information, refer to [Appendix Reference for the .etlibcertify Input File](#).

Figure 6-1. Complete Syntax of the .etlibcertify Configuration File

```
ETLibCertify (<ICTechnologyName>) {  
    SimModelDir: <dirName>; // Repeatable  
    SimModelFile: <fileName>; // Repeatable  
    VerilogOptionFile: <fileName>;  
    TessentCellLib: <fileName>; // Repeatable  
    ConstantPinConnections { // To declare pins on cells that  
                            // must be tied high or low.  
        Cell (CellNameRegExpList) { //Repeatable  
            LogicLow: <pinNameRegExpList>;  
            LogicHigh: <pinNameRegExpList>;  
        } // End of Cell wrapper  
    } // End of ConstantPinConnections wrapper  
    CellFilter {  
        <CellNameRegExpList>: Exclude; // Repeatable  
    } // End of CellFilter wrapper  
} // End of ETLibCertify wrapper
```

You can specify model directories, model files, and *TessentCellLib* files in any order. They are processed in the order in which they are listed. Consequently, ETLibCertify ignores any duplicate models in directories or files specified later in the configuration file.

Certification Process

ETLibCertify performs the following tasks upon invocation:

- Verifies that the functional description, the scan, pin and cell attributes and the dft_cell_selection section of all cells in the Tessent cell library are correct. Also verifies the port list of each cell.
- Creates the following certification workspaces:
 - *Scan Model Certification Workspace* — to verify the cell library description of the functionality and attributes against the simulation models.
 - *Pad Cell Certification Workspace* — to verify that the attributes of a pad are correct. For the padCertify workspace to be created, at least one pad cell must be defined.

ETLibCertify creates certification workspaces that are populated with all of the files necessary to perform the certification. The contents of these workspaces include the following:

- A netlist that instantiates all relevant cells.
- A Makefile for executing all steps in the certification process (either single step or as a group).
- All side files that are needed by the tools that are used in the certification process.

Scan Model Certification Workspace


ETLibCertify performs *Scan Model Certification* by navigating to the *scanCertify* directory and executing the certification targets as a group (**make all**).

The Makefile for the *Scan Model Certification* consists of the following make targets that are executed one after another when the **make all** command is run:

Usage	Description
make designe	Runs designExtract on the top-level module to create .tcm file.
make scang	Performs a flop replacement of regular flops with scan flops.
make rulea	Extracts flat model on post-scan circuit.
make fastscan	Generates ATPG vectors for all cells including the scan chains.
make testbench	Generates testbench for simulating the ATPG vectors.
make sim	Simulates the post-scan ATPG vectors.
make display_sim_results	Shows the content of all simulation result files.
make summary_report	Inspects the log files and reports whether certification succeeded.

Group Targets

```
make all                (make designe, make scang, make rulea,  
                           make fastscan, make testbench, make sim,  
                           summary_report)
```

 **Note** **make scang** (marked in **Red**) is not present when scan attributes are not included in the Tessent Cell Library.

Pad Cell Certification Workspace

ETLibcertify performs certification of the pad cells by navigating to the *padCertify* directory and executing the certification targets as a group.

The Makefile for the *padCertify* workspace consists of the following make targets which are executed one after another when the **make all** command is run:

Usage	Description
<hr/>	
make embedded_test	Generates a TAP and boundary scan to exercise the pads.
make designe	Runs designExtract on the top-level module to create .tcm file.
make testbench	Generates the jtagVerify testbench.
make sim	Simulates the jtagVerify testbench.
make summary_report	Inspects the log files and reports whether certification succeeded.

Group Targets

```
make all                (make embedded_test, make designe,  
                           make testbench, make sim, make summary_report).
```

Other targets

```
make pin_order_file    Generates a pin order file.  
make config_etSignOff Generates ETVerify input configuration file.
```

Summary Reports

At the end of each certification process, ETLibCertify generates a summary report that provides the certification status of each cell:

- For the *Scan Certification process*, the report file is named *scanCellSummary.final* and is placed in the *outDir* directory within the *scanCertify* directory.
- For the *pad.library/cell.library Certification process*, the report file is named *padCellSummary.final* and is placed in the *outDir* directory within the *padCertify* directory.

You can use the **make_summary_report** make target to regenerate the report. [Figure 6-2](#) and [Figure 6-3](#) illustrate the results of *Scan* and *Pad Library Certifications*. The explanation of the summary reports follows the figures.

Figure 6-2. Example scanCellSummary.final

Cell Name	Cell Type	Scan Model Status
dti_mx4lq	Latch	Certified
dti_mx4lqx1	Latch	Certified
dti_mx4soffq	Flip-flop	Certified
dti_mx4soffqx1	Scan flip-flop	Certified
dti_nand2i1	Combinational	Certified
dti_nand2i1x1	Combinational	Certified
dti_nand2i1x2	Combinational	Certified

Figure 6-3. Example padCellSummary.final

Cell Name	Direction	Pad Cell Status
iopad	Inout	Certified
ipad	Input	Certified
ipad_buf	Input	Certified
opad	Output	Certified

The different statuses that might be reported in the *scanCellSummary* and the *padCellSummary* reports are explained below:

- *Certified* — The cells do not cause errors while being processed by the Mentor Graphics tools and do not cause any simulation fail.
- *No scan equivalent* — A non-scan flop is not certified if no scanable equivalent flop has been specified in the *scang.library* file.
- *Not certified* — Sequential cells are not certified if no *scang.lib* file is provided.
- *Empty* — The scan model does not contain any logic gates and will not be certified. This might happen for cells that have no functional logic.
- *Excluded* — The scan model is specified in the [CellFilter](#) wrapper and has not been certified.
- *FAIL: not in scan chain* — This failure only applies to cells of type *Scan flip-flop*. If such a cell does not appear in the scan chain then the Mentor Graphics tools have not recognized this cell as a flip-flop cell, or have failed to stitch the flip-flop into the scan chain. You might want to check the *scang.log* and the *rulea.log* files for any error messages for this cell.

- *FAIL: not simulated* — This failure is reported if there is no evidence that the cell was actually simulated.
 - During the *Scan Certification process*, the *.ruleainfo* file is inspected to make sure that there are pins that connect to the cell. If there are no such pins then it appears that the cell instance has been pruned by ruleAnalyze. For instance, this would happen if the cell does not have an output pin.
 - In the *Pad Certification process*, comments in the simulation log file show that either a value is applied to an input pad (*Setting*) or verified on an output pad (*Checking*). If no such comment is found for a pad cell, the pad cell is reported as *not simulated*.

Typically, this failure happens because the cell contains no digital logic circuitry. If the cell never appears in a netlist then no scan model for the cell is needed.

Note



When there are holding scan flip-flops in your library then their status will be *FAIL: not simulated* at the end of the certification process. Certification of holding scan flops is currently not supported.

- *FAIL: simulation* — This failure is reported if there is a compare failure in the simulation on an output net of this cell. A simulation failure indicates that the scan model is inconsistent with the Verilog simulation model. Inspecting the simulation waveforms might help pinpoint the cause of the discrepancy.
- *FAIL: OTHER* — This failure is reported if ETLibCertify reports an error when it processes the cell and instantiates it in the certification netlist. The error message for this cell in the ETLibCertify log file provides details on the reason for the failure.

Example of Using ETLibCertify

Figure 6-4 presents an example of a *dolphin.etlibcertify* file that specifies the locations of scan models, simulation models, library files, and some additional information for the Dolphin technology:

Figure 6-4. Example .etlibcertify File

```
ETLibCertify(dolphin) {  
  SimModelDir: dolphin/verilog;  
  SimModelFile: dolphin/verilog/pads.v;  
  VerilogOptionFile: dolphin/simoptions;  
  TessentCellLib: dolphin/atpg.lib;  
  ConstantPinConnections {  
    Cell("AB*L") {  
      LogicLow: "*VSS, *PD";  
      LogicHigh: "*VDD, *PU";  
    }  
  }  
}
```

```
CellFilter {  
    "DE.*[FG]": Exclude;  
}  
}
```

Using the information in the *dolphin.etlibcertify* file, ETLibCertify performs the following tasks:

- Invokes *Tessent FastScan* to generate scan models for the cells in the *dolphin/atpg.lib* file.
- Reads the description of the scan models in *dolphin/scan* directory and in the *dolphin/scan/pads.v* file, as well as the scan models generated by Tessent FastScan.
- Sets up the *Scan Model Certification workspace* with the name *scanCertify* and the *pad cell Certification workspace* with the name *padCertify*. All scan models are instantiated in the netlist in the *Scan Model Certification workspace*. All cells that are identified as pad cells are instantiated in the netlist in the *pad.library/cell.library Certification workspace*.

The information in the *ConstantPinConnections* wrapper is used to connect the ports of the instantiated cells. In both netlists, ports of cells which names start with *AB* and end with *L* are tied off in a special way. If the name of the port of such a cell ends with *VSS* or *PD*, the port is tied to logic 0. If the name of the port ends with *VDD* or *PU*, the port is tied to logic 1. The ports and cells that are not specified in the *ConstantPinConnection* wrapper are connected using information from the library files and scan model descriptions.

The information in the *CellFilter* wrapper is used to exclude cells from being processed during certification. All cells whose names start with *DE* and end with *F* or *G* will be excluded. These cells are not instantiated in the netlist and will not be certified. They will be listed in the summary report with the status *Excluded*.

The creation of the workspaces is completed by generating the Makefiles and the side files. The names of the *dolphin/verilog* directory, the *dolphin/verilog/pads.v* file, and the *dolphin/simoptions* file are listed in a side file that is used for simulation.

After the ETLibCertify run is complete, you need to execute the certifications by issuing the command **make all** in each certification workspace.

Chapter 7

Shell Command Dictionary

This chapter describes the Tessent Cell library commands. For quick reference, the commands appear alphabetically with each beginning on a separate page.

Shell Command Descriptions

The notational conventions used to describe the shell commands are the same as those used in other parts of the manual. Do not enter any of the special notational characters (such as, {}, [], or |) when typing the command.

[Table 7-1](#) contains a summary of the shell commands described in this chapter.

Table 7-1. Shell Command Summary

Command	Description
etlibcertify	Invokes the ETLibCertify tool and certifies a set of scan models together with their associated scang.lib, pad.library, and cell.library files.
lcverify	Invokes lcVerify and verifies the ATPG library models generated by LibComp.
libcomp	Invokes the LibComp utility to compile a Verilog library into ATPG library format.

etlibcertify

Usage

etlibcertify **technology_file**

Description

Invokes the ETLibCertify tool and certifies a set of scan models together with their associated *scang.lib*, *pad.library*, and *cell.library* files.

These scan models and library files can be generated manually with LibComp or any third-party tool.

The certification process is done without the use of Liberty files.

Arguments

- **technology_file**

An optional string that specifies a single technology-specific configuration file. The configuration file specifies the input data (files and/or directories) for which ETLibCertify must create the certification workspace(s). The root name of the configuration file must match the name of the technology and have the suffix *.etlibcertify* attached. For example: *dolphin.etlibcertify*.

Example

The following example invokes the ETLibCertify tool.

etlibcertify dolphin.etlibcertify

lcverify

Usage

```
lcverify [-FASTSCAN | -FLEXTEST | -TESTKOMPRESS] atpg_library_path  
         verilog_library_path [-MODEL model_name] [-NO_ATPG_PRIMs]  
         [-NO_SCAN_RAMs] [-SAVE_VSIM] [-SV | -NO_SV] [-HELP]
```

Description

Invokes lcVerify and verifies the ATPG library models generated by LibComp.

lcVerify uses Tessent FastScan to generate test patterns and simulate the models; it also uses ModelSim to simulate the Verilog source modules and then compares the simulated values to ensure models function as intended. The ATPG models must be generated with a matching version of LibComp.

By default, lcVerify runs automatically from [libcomp](#) as the models are generated. For more information, see “[Verifying Tessent Simulation Models](#)”.

Arguments

- **-FASTSCAN | -FLEXTEST | -TESTKOMPRESS**
An optional literal that determines which tool generates the test patterns and simulates the models. Tessent FastScan is the default.
- ***atpg_library_path***
A required string that specifies a file or directory containing the ATPG library models to verify.
- ***verilog_library_path***
A required string that specifies a file or directory containing the source Verilog modules to verify the ATPG library models against.
- **-MODEL *model_name***
An optional switch and string pair that specifies verification of a single model in an ATPG library. For additional details, see “[Verifying Tessent Simulation Models](#)”.
- **-NO_ATPG_PRIMs**
An optional switch that disables the use of the ATPG primitives definition file to verify ATPG model libraries with ATPG primitives. By default, the ATPG primitives definition file is used during simulation.

Use this switch to run lcverify for verification only when the Verilog source contains Verilog module names “and”, “or”, “_TIEX”, and so on (see the *atpg_lib_prims.v* file for a complete list) to prevent the display of duplicate module messages from ModelSim.

Caution



This switch may prevent ModelSim from simulating the Verilog modules. You should either remove the ATPG primitives from the existing libraries or use the script located in `<Tessent_Tree_Path>/lib/tmax_to_verilog.pl` to convert ATPG primitives to Verilog primitives. For more information, see [“Creating Simulation Models”](#).

- **-NO_SCAN_RAMs**
An optional switch that disables the insertion of scan chains and gating around RAMs when verifying a library. Uses older style PI/PO RAM connections.
- **-SAVE_VSIM**
An optional switch that prevents deleting the files that allow rerunning ModelSim without rerunning Tessent FastScan (without reinvoking lcverify). This switch also saves *pat.ascii*.
- **-SV**
An optional switch that invokes ModelSim with System Verilog compilation for .v files. Required when a reg is used as interconnect (such as in a port list). See [“Reconciling System Verilog reg and Verilog Keyword Compiling Issues”](#).
- **-NO_SV**
An optional switch that disables the -sv (System Verilog) switch for .v files. Required when the -sv switch causes compile errors due to modules containing reserved Verilog keywords. See [“Reconciling System Verilog reg and Verilog Keyword Compiling Issues”](#).
- **-HELP**
An optional switch that displays a description of all the lcverify invocation switches.

Example

The following example invokes lcVerify in stand-alone mode to verify the *lib10.atpg* ATPG library with the *lib10.v* Verilog source library.

```
<Tessent_Tree_Path>/bin/lcverify lib10.atpg lib10.v
```


libcomp

Usage

```
libcomp library_path... {-DOfile [dofile_name]} [+vlog_directive ...] [-NO_SCAN_rams]
[-FLEXtest] [-TESTkcompress] [-LOGfile logfile_name [-Replace]] [-SV | -NO_SV] [-Help |
-MANual | -VERsion | -Usage] [-NO_DEFine_simple_views]
```

Description

Invokes the LibComp utility to compile a Verilog library into ATPG library format.

If no arguments are specified, LibComp invokes in interactive mode.

For more information, see “[Using LibComp to Create Tessent Simulation Models](#)”.

[Table 7-2](#) lists default macros that you can use in the netlist to control how the design is compiled.

Table 7-2. LibComp Views

MGC_TESSENT	lv_rtl_primitives	functional
TETRAMAX	FAST_FUNC	syntest

Note



LibComp warns you about any macros it defines. If your test view should not include any of the macros listed in [Table 7-2](#), you should disable all of these macros using the `-NO_DEFine_simple_views` switch; then `+define+..` any individual macros whose views you wish to retain using the `+vlog_directive` switch documented below.

Arguments

- *library_path*
An optional repeatable string that specifies the file or directory populated with library cells. The library must be a Verilog-formatted library.
- `-DOFile` *dofile_name*
An optional switch and optional string pair that specifies the name of the dofile to execute upon invocation. If you issue `-dofile` without specifying a *dofile_name* or by specifying a “+” (plus), “-” (minus), or end-of-line, LibComp uses the default dofile at `<Tessent_Tree_Path>/lib/tools/libcomp/libcomp.do.default`.
- `+vlog_directive`
An optional switch and single string that specifies Verilog compiler directives. This argument primarily supports defining alternative Verilog views: specifically, using “`+define+string`” at invocation is equivalent to editing the first file and adding a line with only ``define string` for the first line of the Verilog source before compilation. Note a tick character must precede the *define* statement. See “[Example 4](#).”

You must precede the *vlog_directive* with a “+” (plus).

- -NO_SCAN_rams

An optional switch that invokes Tessent FastScan and surrounds RAMs with PI/PO rather than inserting scan gating and chains around each RAM, and scan testing them.

- -FLEXtest

An optional switch that sets LibComp to use FlexTest when verifying ATPG models. By default, LibComp uses Tessent FastScan for verification.

- -TESTkcompress

An optional switch that sets LibComp to use Tessent TestKcompress when verifying ATPG models. By default, LibComp uses Tessent FastScan for verification.

- -LOGfile *logfile_name*

An optional switch and string pair that writes the session information to a file you specify. By default, LibComp outputs the session information to the display. This logfile includes the banner specifying the tool, version, date, and platform for the session.

- -REPlace

An optional switch that overwrites the -Logfile *logfile_name* if a log file of the same name already exists.

- -SV

An optional switch that invokes ModelSim with System Verilog compilation for .v files. Required when a reg is used as interconnect (such as in a port list). See “[Reconciling System Verilog reg and Verilog Keyword Compiling Issues](#)” for complete information.

- -NO_SV

An optional switch that prevents invoking ModelSim with -sv (System Verilog) for .v files. Required when the -sv switch causes compile errors due to modules containing reserved Verilog keywords. See “[Reconciling System Verilog reg and Verilog Keyword Compiling Issues](#)” for complete information.

- -Help

An optional switch that displays a description of all the LibComp invocation switches.

- -MANual

An optional switch that opens the bookcase of DFT documentation.

- -VERSion

An optional switch that displays the version of your LibComp software.

- -NO_DEFine_simple_views

An optional switch that prevents automatic definition of views in Verilog source modules for translation and for LibComp’s verification using ModelSim. The default is to automatically define the views listed in [Table 7-2](#). It is recommended that you not use this switch unless you are sure that a simpler view is inappropriate for test. Be aware that it is

almost never appropriate to use the more difficult view for test, which is what happens when you use this switch.

Example 1

The following example invokes the LibComp utility on a Verilog library file named “*lib10.v*” and runs the default dofile.

```
% <Tessent_Tree_Path>/bin/libcomp lib10.v -dof -log my.log
```

The default dofile is located at *Tessent_Tree_Path/lib/tools/libcomp/libcomp.do.default*. The output from this command is written to the screen and to the file “*my.log*”.

Example 2

The following example invokes the LibComp utility on a Verilog library file named “*lib10.v*” and runs the user-edited dofile “*lib10.do*” upon invocation.

```
% <Tessent_Tree_Path>/bin/libcomp lib10.v -dof lib10.do
```

Example 3

The following example invokes LibComp in interactive mode:

```
% libcomp
```

It results in the following:

```
// Note: Libcomp invocation missing -dofile argument.
// Invoked interactively.
//
SETUP>
```

Example 4

In this example, the Verilog source file (*source.v*) contains the following lines. (Note, the *ifdef*, *else*, and *endif* statements are immediately preceded by the tick character.)

```
`ifdef functional_view
... simpler, Verilog functional view ...
`else
... default, complex Verilog simulation view with many pessimism checks
...
`endif
```

Libcomp cannot translate the more complex view because of unsupported constructs. However, LibComp can translate the simpler view and this view’s verification using ModelSim with the following invocation:

```
<Tessent_Tree_Path>/bin/libcomp source.v +define=functional_view \
-dofile ...
```


Appendix A

Attributes and the Tessent Cell Library

The Tessent cell library definition supports all of the LV library attributes and syntax as well as the ATPG library syntax. This section describes the attributes in the Tessent cell library and maps the LV and ATPG library attributes to the Tessent cell library attributes.

Tessent Pin Function Attributes.....	286
Cell Library Mappings	290
Tessent LV Flow Library Mappings	292
Tessent LV Flow Pad Library Mappings	294
Tessent Scan ATPG Library Mappings.....	300

Tessent Pin Function Attributes

Table A-1 lists each Tessent pin functions attribute, the tools that use it, and any dependencies on it. `mux_select0`

Table A-1. Tessent Pin Function Attributes

Attribute	Used by Tools	Required by
active_high_clock	LV Flow ¹ , Tessent Scan	Latch-based cell with level sens clock
active_high_reset	None ²	None
active_high_set	None	None
active_low_clock	LV Flow, Tessent Scan	Latch-based cell with level sens clock
active_low_reset	None	None
active_low_set	None	None
asynch_enable	LV Flow	LV Flow clock gating cell
asynch_enable_inv	LV Flow	LV Flow clock gating cell
asynch_disable	LV Flow	LV Flow clock gating cell
asynch_disable_inv	LV Flow	LV Flow clock gating cell
clock_in	LV Flow	LV Flow clock gating cell
clock_out	LV Flow	LV Flow clock gating cell
data_in	LV Flow, Tessent Scan	LV Flow mux scan system data input
data_out	LV Flow	Pipeline flop
data_out_inv	LV Flow	Pipeline flop
func_enable	LV Flow	LV Flow
func_enable_inv	LV Flow	LV Flow
max_fanout	Tessent Scan	cell_type = buffer only for balancing
mux_in0	LV Flow, Tessent Scan	cell_type = mux
mux_in1	LV Flow, Tessent Scan	cell_type = mux
mux_in2	Tessent Scan	cell_type = mux
mux_in3	Tessent Scan	cell_type = mux
mux_in4	Tessent Scan	cell_type = mux

Table A-1. Tessent Pin Function Attributes (cont.)

Attribute	Used by Tools	Required by
mux_in5	Tessent Scan	cell_type = mux
mux_in6	Tessent Scan	cell_type = mux
mux_in7	Tessent Scan	cell_type = mux
mux_select	LV Flow, Tessent Scan	cell_type = mux
mux_select0	Tessent Scan	cell_type = mux
mux_select1	Tessent Scan	cell_type = mux
mux_select2	Tessent Scan	cell_type = mux
negedge_clock	LV Flow, Tessent Scan	Negedge clocked cell types
no-fault (instance)	ATPG ³	None
no-fault (model pins)	ATPG	None
open	LV Flow	Test-inserted cells with extra pins.
pad_ac_mode_dot6	Boundary Scan	AC test
pad_data_inv	Boundary Scan	None
pad_diff_current	Boundary Scan	None
pad_diff_voltage	Boundary Scan	None
pad_enable_high	Boundary Scan	None
pad_enable_low	Boundary Scan	None
pad_force_disable	Boundary Scan	None
pad_from_io	Boundary Scan	None
pad_from_io_inv	Boundary Scan	None
pad_from_pad	Boundary Scan	None
pad_from_sje_mux	Boundary Scan	None
pad_from_sji_mux	Boundary Scan	None
pad_from_sjo_mux	Boundary Scan	None
pad_init_clock_dot6	Boundary Scan	AC test
pad_init_data_dot6	Boundary Scan	AC test
pad_init_data_inv_dot6	Boundary Scan	AC test
pad_nonjtag	Boundary Scan	None

Table A-1. Tessent Pin Function Attributes (cont.)

Attribute	Used by Tools	Required by
pad_open	Boundary Scan	Pin condition identifying mode/Usage.
pad_pad_io	Boundary Scan	None
pad_pad_io_inv	Boundary Scan	None
pad_parametric	Boundary Scan	None
pad_sample_only	Boundary Scan	None
pad_sample_pad	Boundary Scan	None
pad_select_jtag_enable	Boundary Scan	None
pad_select_jtag_in	Boundary Scan	None
pad_select_jtag_out	Boundary Scan	None
pad_test_data_dot6	Boundary Scan	AC test
pad_test_data_inv_dot6	Boundary Scan	AC test
pad_tied0	Boundary Scan	Pin condition identifying mode/Usage.
pad_tied1	Boundary Scan	Pin condition identifying mode/Usage.
pad_to_io	Boundary Scan	None
pad_to_io_inv	Boundary Scan	None
pad_to_pad	Boundary Scan	None
pad_to_sje_mux_low	Boundary Scan	None
pad_to_sje_mux_high	Boundary Scan	None
pad_to_sji_mux	Boundary Scan	None
pad_to_sjo_mux	Boundary Scan	None
posedge_clock	LV Flow, Tessent Scan	cell_type = dff/latch/scan_cell..
power_isolation_internal	LV Flow	None
power_isolation_external	LV Flow	None
scan_enable	LV Flow, Tessent Scan	cell_type = scan_cell
scan_enable_inv	LV Flow, Tessent Scan	cell_type = scan_cell
scan_in	LV Flow, Tessent Scan	cell_type = scan_cell

Table A-1. Tessent Pin Function Attributes (cont.)

Attribute	Used by Tools	Required by
scan_out	LV Flow, Tessent Scan	cell_type = scan_cell
scan_out_inv	LV Flow, Tessent Scan	cell_type = scan_cell
test_enable	Tessent Scan	None
test_enable_inv	Tessent Scan	None
tie0	LV Flow, Tessent Scan	Cells that need ties for test insertion
tie1	LV Flow, Tessent Scan	Cells that need ties for test insertion
unused	LV Flow, Tessent Scan, ATPG	Cell input pins not connected to anything inside.

1. LV Flow.
2. Not used by any tools. For information purposes only.
3. Automatic Test Pattern Generation

Tessent Pin Special Drive Attributes

Table A-2 lists each Tessent pin special drive attribute, the tools that use it, and any dependencies on it

Table A-2. Tessent Pin Special Drive Attributes

Attribute	Used by Tools	Required by
pad_open_drain	Boundary Scan	None
pad_open_source	Boundary Scan	None
pad_pull0	Boundary Scan	None
pad_pull1	Boundary Scan	None

Cell Library Mappings

Table A-3 maps the LV Flow *cell.lib* attributes and syntax to the new Tessent cell library syntax and provides any additional needed description.

Table A-3. cell.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
And2 -or- And2 (<cellName>) {}	and = model_name;	2-input AND cell. A model is the same as a cell. For example: and2 = <cellName>;
Port (<portName>):<portFunction>;	input (<portName>) (pinFunction; ...); output (<portName>) (pinFunction; ...); inout (<portName>) (pinFunction; ...);	Moved to attributes on model pin of same <portName>:
Input	input (portName) (cell_in)	Deduced from pin direction.
Output	output (portName) (cell_out)	Deduced from pin direction.
LogicHigh	input (portName) (tie1)	Attribute.
LogicLow	input (portName) (tie0)	Attribute.
Open	output (portName) (unused)	Attribute.
Buffer (<cellName>) {}	buffer = model_name;	Usage: buffer = <cellName>;
Inverter (<cellName>) {}	inverter = model_name;	Usage: inverter = <cellName>;
Or2 (<cellName>) {}	or = model_name;	Usage: or = <cellName>;
Multiplexer (<cellName>) {}	mux = model_name;	Usage: mux = <cellName>;
Input0	input (portName) (mux_in0)	Attribute.
Input1	input (portName) (mux_in1);	Attribute.
Select	input (portName) (mux_select);	Attribute.

Table A-3. cell.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
RetimingFlop (<cellName>) {}	retiming_flop = model_name;	Defines the cell to be used as a retiming flip-flop in the design. Now obsolete; replaced by dff.
UpdateGroupDelayElement (<cellName>) {}	update_group_delay_element = model_name;	Defines cell to be used as a delay element to control ground bounce during the update boundary-scan update cycle. Now obsolete; replaced by buffer.
CellsToUseOnFunctionalPaths {}	Omitted.	Reconstruct from dft_cell_selection (<selection_name1>) {...} cells that start with <i>clock_</i> . All these attributes are contained within this section.
ClockBuffer (<cellName>) {}	clock_buffer = model_name;	Usage: clock_buffer = <cellName> ;
ClockInverter (<cellName>) {}	clock_inverter = model_name;	Usage: clock_inverter = <cellName> ;
ClockMultiplexer (<cellName>) {}	clock_mux = model_name;	Usage: clock_mux = <cellName> ;
ClockOr (<cellName>) {}	clock_or = model_name;	Usage: clock_or = <cellName> ;
ClockAnd (<cellName>) {}	clock_and = model_name;	Usage: clock_and = <cellName> ;
ClockGatingANDCell (<cellName>) {}	clock_gating_and = model_name;	Usage: clock_gating_and = <cellName> ;
ClockGatingORCell (<cellName>) {}	clock_gating_or = model_name;	Usage: clock_gating_or = <cellName> ;
Clock	input (portName) (clock_in);	For clock_gating cell pin.
ClockGated	output (portName) (clock_out);	For clock_gating cell pin.

Table A-3. cell.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
FuncEnable	input (portName) (func_enable);	For clock_gating cell pin.
FuncEnableInv	input(portName) (func_enable_inv);	For clock_gating cell pin.
AsynchEnable	input (portName) (asynch_enable);	For clock_gating cell pin.
AsynchEnableInv	input (portName) (asynch_enable_inv);	For clock_gating cell pin.
AsynchDisable	input (portName) (asynch_disable);	For clock_gating cell pin.
AsynchDisableInv	input (portName) (asynch_disable_inv);	For clock_gating cell pin.
TestEnable	input (portName) (test_enable);	For clock_gating cell pin.
TestEnableInv	input (portName) (test_enable_inv);	For clock_gating cell pin.

Tessent LV Flow Library Mappings

Table A-4 maps the Tessent LV Flow *scang.lib* attributes and syntax to the new Tessent cell library syntax and provides any additional needed description.

Table A-4. scang.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
Module (<module_name>)	model (<model_name>)	
Type:	cell_type =	
buffer	buffer	Deduced from pin direction. Usage: cell_type = buffer;
inverter	inverter	Usage: cell_type = inverter;
noscan	nonscan	Usage: cell_type = nonscan;
scan	scan_cell	Usage: cell_type = scan_cell;
pipeline	pipeline	Usage: cell_type = pipeline;

Table A-4. scang.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
latch	latch	Usage: cell_type = latch;
XOR	xor	Usage: cell_type = xor;
OR	or	Usage: cell_type = or;
AND	and	Usage: cell_type = and;
MUX	mux	Usage: cell_type = mux;
tieHigh	tie1	Usage: cell_type = tie1;
tieLow	tie0	Usage: cell_type = tie0;
NAND	nand	Usage: cell_type = nand;
NOR	nor	Usage: cell_type = nor;
ScanEquivalent: module_name;	Find scan model whose name is module_name and add: nonscan_model = this_model_name;	For equating cells, not pins. Equating pins on a scan equivalent cell is listed below.
Pin (pinName) {pin_type_attribute}	input (pin_name) (pin_type_attribute) output (pin_name (pin_type_attribute) inout (pin_name) (pin_type_attribute)	All pin information now inside second set of parens after declaration of pin direction.
input	data_in	
output	data_out	
scanenable	scan_enable	
scanenablenot	scan_enable_inv	
scanin	scan_in	
scanout	scan_out	
scanoutnot	scan_out_inv	
clockenable	clock_enable	LogicVision internal.
input0	mux_in0	
input1	mux_in1	
select	mux_select	
open	unused	
clockn	clockn	

Table A-4. scang.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
clockp	clockp	
datain	data_in	
dataout	data_out	
dataoutnot	data_out_inv	
resetn	active_low_reset	Pin_type_attribute.
resetp	active_high_reset	Pin_type_attribute.
ScanEquivalent: <scanpinName>;	Find scan model from Module() level ScanEquivalent and add: nonscan_model = <model_name> (list_of_pins);	Only differing scan pin names for a scan cell need to be specified in the single list_of_pins. For example: pin (I) {scanEquivalent: D;} becomes .I(D) in list.

Tessent LV Flow Pad Library Mappings

[Table A-5](#) maps the Tessent LV Flow *pad.lib* attributes and syntax to the new Tessent cell library syntax and provides any additional needed description.

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
VHDLPackage: <packageName>;	dft_cell_selection (vhdl_package_name = <packageName>;	Now obsolete.
DefaultBcells { }	dft_cell_selection (default_bcell_libs())	Inside library level dft_cell_selection () section.
<BcellNamei>:	model_name =	model_name of the boundary scan cell. The following row lists the classes.
class (subclass1, ...);	class (subclass1, ...);	Kept class (subclass_list); syntax as is, with optional parens and subclass_list.
I	in_cell	Boundary scan cell class. e.g. my_bs_in_cell = input_cell(...);

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
O	out_cell	Boundary scan cell class .e.g. my_bs_out_cell = output_cell(...);
IO	inout_cell	Boundary scan cell class .e.g. my_bs_io_cell = inout_cell(...);
EN	enable_cell	Boundary scan cell class. e.g. my_bs_en_cell = enable_cell(...);
M	mux_inside_pad	Boundary scan cell subclass. SJI/SJO mux inside pad cell.
2	two_state	Boundary scan cell subclass.
DC	diff_current	Boundary scan cell subclass.
DV	diff_voltage	Boundary scan cell subclass.
FD	force_disable	Boundary scan cell subclass.
P0	pull0	Boundary scan cell subclass.
P1	pull1	Boundary scan cell subclass.
AC	ac_dot6	Boundary scan cell subclass. LV Flow "AC". 1149.6 compatible pad.
ACM	acm_dot6	Boundary scan cell subclass. LV Flow "ACM" 1149.6 Has ACMode input pin.
NFP	from_pad_dot6	Boundary scan cell subclass. 1149.6 Has from_pad output.
II	inv_in	Boundary scan cell subclass. Inverted input pad (functional path).

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
IO	inv_out	Boundary scan cell subclass. Inverted output pad (functional path).
S	sample_only	Boundary scan cell subclass. Use sample-only bscan cell.
MO	muxed_out	Boundary scan cell subclass.IO class. Pad only has SJO mux.
NJ	nonjtag	Boundary scan cell subclass. Pad forces non-Jtag pin (no bscan cell).
0	enable_low	Boundary scan cell subclass. Subclass. EN class only.
1	enable_high	Boundary scan cell subclass.EN class only.
H	hold	Boundary scan cell subclass. Output (O), bidirectional (IO), and control (EN) classes of boundary-scan cells.
DI	disabled_in	Boundary scan cell subclass. IO class. Disabled functional path.
DO	disabled_out	Boundary scan cell subclass. IO class. Disabled functional path.
OD	open_drain	Boundary scan cell subclass.
OS	open_source	Boundary scan cell subclass.
SP	sample_pad	Boundary scan cell subclass. Pad has buffered out of functional input.

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
Cell (<cellName>){ }	model (model_name) (cell_type = pad;)	Include all of the following inside <i>model</i> as model statements or pin attributes. Explicitly type as pad inside model.
New cell level non_jtag	pad_nonjtag;	Can occur at cell/model level or on individual fromPad, toPad, pins.
PadAttribute:AC;	pad_ac;	Can occur at cell/model level or within mode section.
BlibCell:<cellName>;	bcell_lib_name = model_name;	
ACTiming{ }	Omitted.	Encoded in name so can retrieve.
LP_time: <time_in_seconds>;	pad_ac_lp_time = <time_in_seconds>;	Can be at Cell (shown here) or at Usage level (shown below).
HP_time: <time_in_seconds>;	pad_ac_hp_time = <time_in_seconds>;	Can be at Cell (shown here) or at Usage level (shown below).
HP_onChip: Yes (No);	pad_ac_hp_on_chip; // Omitted if No.	Can be at Cell (shown here) or at Usage level (shown below).
Usage { }	mode ()	Following are inside Cell {Usage{ }} which becomes model (mode ()).
PadAttribute: AC;	pad_ac;	Can occur at cell/model level or within usage/mode section.
ACTiming{ }	Omitted.	Encoded in name so can retrieve.
LP_time: <time_in_seconds>;	pad_ac_lp_time = <time_in_seconds>;	Can be at Cell (shown here) or at Usage level (shown below).
HP_time: <time_in_seconds>;	pad_ac_hp_time = <time_in_seconds>;	Can be at Cell (shown here) or at Usage level (shown below).

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
HP_onChip: Yes (No);	pad_ac_hp_on_chip; // Omitted if No.	Can be at Cell (shown here) or at Usage level (shown below).
Pin (<pinName>){ }	input (pinName) () output (pinName) () inout (pinName) ()	Following in Cell{ Usage{ Pin{ ... } } } which becomes for an input pin for example: model (mode (input (...)))
Function:	Omitted.	Can reconstruct from attribute names.
forceDisable	pad_force_disable	Pin or model attribute.
fromPad	pad_from_pad	Pin or model attribute.
toPad	pad_to_pad	Pin or model attribute.
enableHigh	pad_enable_high	Pin or model attribute.
enableLow	pad_enable_low	Pin or model attribute.
fromSJIMux	pad_from_sje_mux	Pin or model attribute.
toSJEMuxLow	pad_to_sje_mux_low	Pin or model attribute.
toSJEMuxHigh	pad_to_sje_mux_high	Pin or model attribute.
toSJIMux	pad_to_sji_mux	Pin or model attribute.
fromSJIMux	pad_from_sji_mux	Pin or model attribute.
toSJOMux	pad_to_sjo_mux	Pin or model attribute.
fromSJOMux	pad_from_sjo_mux	Pin or model attribute.
selectJtagEnable	pad_select_jtag_enable	Pin or model attribute.
selectJtagInput	pad_select_jtag_in	Pin or model attribute.
selectJtagOutput	pad_select_jtag_out	Pin or model attribute.
fromIO	pad_from_io	Pin or model attribute.
toIO	pad_to_io	Pin or model attribute.
padIO	pad_pad_io	Pin or model attribute.
padIOInv	pad_pad_io_inv	For second pin of differential pair.
fromIOInv	pad_from_io_inv	For second pin of differential pair.

Table A-5. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
toIOInv	pad_to_io_inv	For second pin of differential pair.
InitData	pad_init_data_dot6	1149.6.
InitClk	pad_init_clock_dot6	1149.6.
TestData	pad_test_data_dot6	1149.6.
InitDataInv	pad_init_data_inv_dot6	1149.6.
TestDataInv	pad_test_data_inv_dot6	1149.6. For second pin of differential pair.
ACMode	pad_ac_mode_dot6	1149.6
Attribute:	Omitted.	Can reconstruct from attribute names.
sampleOnly	pad_sample_only	Pin attribute.
samplePad	pad_sample_pad	Pin attribute.
nonJTAG	pad_nonjtag	Pin attribute.
inverted	pad_data_inv	Pin attribute. For unidirectional pads.
DV	pad_diff_volt	Pin attribute.
DC	pad_diff_current	Pin attribute.
openDrain	pad_open_drain	Pin attribute.
openSource	pad_open_source	Pin attribute.
parametric	pad_parametric	Pin attribute.
pull0	pad_pull0	Pin attribute.
pull1	pad_pull1	Pin attribute.
Condition:	Omitted.	Can reconstruct from attribute names.
tiedLow	tie0	Pin condition that produces this mode.
tiedHigh	tie1	Pin condition that produces this mode.
open	unused	Pin left unconnected in this mode.

Tessent Scan ATPG Library Mappings

Table A-6 maps the Tessent Scan ATPG libraries attributes and syntax to the new Tessent cell library syntax and provides any additional needed description.

Table A-6. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Tessent Scan Syntax	New Library Syntax	Description and Comments
scan_definition()	Omitted.	Most moved to pin attributes; remaining are model level statements or consolidated.
mux_scan	scan_cell	For example: scan_definition (type = mux_scan;) Becomes: cell_type = scan_cell;
length = integer;	scan_length = integer;	Defined inside model: "model model_name() (defined here)" Required if > 1 bit shift path in cell.
nonscan_model = model_name (list_of_pins);	nonscan_model = model_name;	If nonscan pin names are the same on the scan model, the (list_of_pins) can be omitted.
maps to both:	nonscan_model = model_name (positional_or_pinname);	Otherwise, instantiate nonscan inside scan: (.nonscan_pinname (scan_pinname), ...) or positional: (scan_pin, ..., ...)
data_in = pin_name;	input (pin_name) (data_in;)	Pin type attribute. System data input. Semicolon inside parens is optional.
scan_in = pin_name;	input (pin_name) (scan_in)	Pin type attribute. Scan data input for all scan types.
scan_out = pin_name1, pin_name2;	output (pin_name1) (scan_out) // non-inverting output (pin_name2) (scan_out_inv) // inverting	Pin type attribute. Scan output for all scan types.
scan_enable = pin_name;	input (pin_name) (scan_enable)	Pin type attribute.
scan_enable_inverted = pin_name;	input (pin_name) (scan_enable_inv)	Pin type attribute.
tie0 = pin_name, ...;	input (...) (tie0) ...	Pin type attribute. Placed on each pin in original list.

Table A-6. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Tessent Scan Syntax	New Library Syntax	Description and Comments
tie1 = pin_name, ...;	input (...) (tie1) ...	Pin type attribute. Placed on each pin in original list.
test_enable = name;	input (pin_name) (test_enable)	Pin type attribute.
cell_type = INV	cell_type = inverter;	New <i>type</i> name for cell_type= INV;
cell_type = BUF	cell_type = buffer;	New <i>type</i> name for cell_type= BUF;
cell_type = AND	cell_type = and;	No change to <i>type</i> name for cell_type= and;
cell_type = NAND	cell_type = nand;	No change to <i>type</i> name.
cell_type = OR	cell_type = or;	No change to <i>type</i> name.
cell_type = NOR	cell_type = nor;	No change to <i>type</i> name.
cell_type = XOR	cell_type = xor;	No change to <i>type</i> name.
cell_type = MUX <sel d0 d1>	cell_type = mux; input (...) (mux_select) input(...) (mux_in0) input(...) (mux_in1)	Pin function now on pin attributes. Model level attribute specifies overall cell or model function. Pin attributes specify pin functions.
cell_type = DLAT <clk d> should now be expressed as :	cell_type = active_high_latch active_low_latch ; input (...) (active_low_clock active_high_clock) input (...) (data_in)	Pin function now on pin attributes. active_low_clock active_high_clock used to encode both which pin is "clk" and active high or active low.
cell_type = DLAT <clk d> translates to :	cell_type = dlat; input (...) (clock_in) input (...) (data_in)	Because no clock polarity, the pin function attribute is polarityless.
cell_type = DFF <clk d> should now be expressed as :	cell_type = posedge_dff negedge_dff; input (...) (posedge_clock negedge_clock) input (...) (data_in)	Pin function now on pin attributes. posedge negedge used to encode both which pin is "clk" and posedge or negedge.
cell_type = DFF <clk d> translates to :	cell_type = dff; input (...) (clock_in) input (...) (data_in)	Because no clock polarity, the pin function attribute is polarityless.
cell_type = CLKBUF	cell_type = clock_buffer;	

Table A-6. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Tessent Scan Syntax	New Library Syntax	Description and Comments
cell_type = SCANCELL < >	Obsolete.	Now must explicitly declare as cell_type = scan_cell.

Appendix B

Converting TetraMax Primitives to Verilog Primitives

Although LibComp translates TetraMax primitives into a valid Mentor Graphics ATPG library, the primitives are undefined modules in Verilog and prevent simulation, which is needed to verify the translated ATPG library. LibComp provides an ATPG primitives definition file (*atpg_lib_prims.v*) that defines the mux (multiplexor), dlat (D latch), and dff (D flip-flop) ATPG (or extended Verilog) primitives referenced in Verilog for simulation with ModelSim. However, some libraries contain different width Boolean primitives (and, or, and so on) that are not defined in the *atpg_lib_prims.v* file. Therefore, if you want to verify ATPG libraries using LibComp's verification, you must either remove the ATPG primitives from the source libraries or run the *tmax_to_verilog.pl* script located in `<Tessent_Tree_Path>/lib/tools/libcomp/tmax_to_verilog.pl`. This script only converts the tieX and Boolean primitives into Verilog primitives. Limitations of the script are included in the *tmax_to_verilog.pl* file.

Appendix C

Reference for the .etlibcertify Input File

The detailed reference information for the following syntax of the *.etlibcertify* input file is provided in sections below:

```
ETLibCertify (<ICTechnologyName>) {  
    SimModelDir: <dirName>; // Repeatable  
    SimModelFile: <fileName>; // Repeatable  
    VerilogOptionFile: <fileName>;  
    TessentCellLib: <fileName>; // Repeatable  
    ConstantPinConnections { // Declares pins on cells that  
        // must be tied high or low.  
        Cell (CellNameRegExpList) { //Repeatable  
            LogicLow: <pinNameRegExpList>;  
            LogicHigh: <pinNameRegExpList>;  
        } // End of Cell wrapper  
    } // End of ConstantPinConnections wrapper  
    CellFilter {  
        <CellNameRegExpList>: Exclude; //Repeatable  
    } // End of CellFilter wrapper  
} // End of ETLibCertify wrapper
```

Cell

The *Cell* wrapper allows you to specify cells which have pins that need to be tied high or low.

Syntax

The following syntax specifies this wrapper:

```
Cell (<cellNameRegExpList>) {  
    LogicLow: <pinNameRegExpList>;  
    LogicHigh: <pinNameRegExpList>;  
}
```

where *cellNameRegExpList* is a regular expression for matching names of cells which have pins that need to be tied off.

Default Value

None

Usage Conditions

This wrapper is used in the *ConstantPinConnections* wrapper of the .etlibcertify file.

This wrapper can be repeated.

Example

Below is an example of a *Cell* wrapper that specifies that cells which names start with *AB* and end with *L* have ports that need to be tied off in a special way:

```
ETLibCertify (dolphin) {  
    ConstantPinConnections {  
        Cell ("AB*L") {  
            LogicLow:  "*VSS, *PD";  
            LogicHigh: "*VDD, *PU";  
        }  
    }  
}
```

CellFilter

The *CellFilter* wrapper is used to exclude cells from being processed during certification. These cells are not instantiated in the netlist and will not be certified. They will be listed in the summary report with the status *Excluded*.

Syntax

The following syntax specifies this wrapper:

```
CellFilter {  
    <CellNameRegExpList>: Exclude;  
}
```

where *CellNameRegExpList* is a colon-separated list with the priority going from left to right.

Default Value

None

Usage Conditions

This wrapper is used in the *ETLibCertify* wrapper and can be specified once. The *Exclude* line can be repeated multiple times.

Example

The following example specifies that all cells which names start with *DE* and end with *F* or *G* will be excluded from the certification:

```
ETLibCertify (dolphin) {  
    CellFilter {  
        "DE.*[FG]": Exclude;  
    }  
}
```

ConstantPinConnections

The *ConstantPinConnections* wrapper allows you to declare pins on cells that must be tied high or low.

Syntax

The following syntax specifies this wrapper:

```
ConstantPinConnections {  
  Cell (<cellNameRegExpList>) {  
    LogicHigh: <pinNameRegExpList>;  
    LogicLow: <pinNameRegExpList>;  
  }  
}
```

Default Value

None

Usage Conditions

This wrapper is used in the *ETLibCertify* wrapper of the .etlibcertify file.

Example

Below is an example of a *ConstantPinConnections* wrapper that specifies cells with names starting with *AB* and ending with *L* are tied off in a special way. If the port name of such a cell ends with *VSS* or *PD*, the port is tied to logic 0. If the name of the port ends with *VDD* or *PU*, the port is tied to logic 1:

```
ETLibCertify (dolphin) {  
  ConstantPinConnections {  
    Cell ("AB*L") {  
      LogicLow: "*VSS, *PD";  
      LogicHigh: "*VDD, *PU";  
    }  
  }  
}
```

ETLibCertify

The *ETLibCertify* wrapper specifies the name of the technology and is the top-most wrapper of the *.etlibcertify* file. This wrapper encloses all information pertinent to the technology that the ETLibCertify tool needs.

Syntax

The following syntax specifies this wrapper:

```
ETLibCertify (<ICTechnologyName>) {  
    .  
    .  
    .  
}
```

Usage Conditions

This is the top-most wrapper of the *.etlibcertify* file.

Example

Below is an example of a *dolphin.etlibcertify* file that specifies the locations of scan models, simulation models, library files, and some additional information for the Dolphin technology:

```
ETLibCertify (dolphin) {  
    SimModelDir: dolphin/verilog;  
    SimModelFile: dolphin/verilog/pads.v;  
    VerilogOptionFile: dolphin/simoptions;  
    TessentCellLib: dolphin/atpg.lib;  
    ConstantPinConnections {  
        Cell ("AB*L") {  
            LogicLow: "*VSS, *PD";  
            LogicHigh: "*VDD, *PU";  
        }  
    }  
}
```

LogicHigh

The **LogicHigh** property specifies that the port of the cell needs to be tied to logic 1.

Syntax

The following syntax specifies this property:

```
LogicHigh: <pinNameRegExpList>;
```

where *pinNameRegExpList* is a comma-separated list with the priority going from left to right.

Default Value

None

Usage Conditions

This property is used in the **ConstantPinConnections: Cell** wrapper of the .etlibcertify file.

Example

Below is an example of a **ConstantPinConnections** wrapper that specifies that ports with names ending with *VDD* or *PU* need to be tied to logic 1 for the selected cells:

```
ETLibCertify (dolphin) {  
    ConstantPinConnections {  
        Cell ("AB*L") {  
            LogicHigh: "*VDD, *PU";  
        }  
    }  
}
```

LogicLow

The **LogicLow** property specifies that the port of the cell needs to be tied to logic 0.

Syntax

The following syntax specifies this property:

```
LogicHigh: <pinNameRegExpList>;
```

where *pinNameRegExpList* is a comma-separated list with the priority going from left to right.

Default Value

None

Usage Conditions

This property is used in the **ConstantPinConnections: Cell** wrapper of the .etlibcertify file.

Example

Below is an example of a **ConstantPinConnections** wrapper that specifies that ports with names ending with VSS or PD need to be tied to logic 0 for the selected cells:

```
ETLibCertify (dolphin) {  
    ConstantPinConnections {  
        Cell ("AB*L") {  
            LogicLow: "*VSS, *PD";  
        }  
    }  
}
```

TessentCellLib

The ***TessentCellLib*** property allows you to specify a Mentor Graphics ATPG library. When this property is specified, Tessent FastScan is invoked to convert the ATPG library to a scan model library and a set of library files (*scang.lib*, *cell.library*, and *pad.library*). The scan model library and library files are then used in the certification process.

Syntax

The following syntax specifies this property:

```
TessentCellLibLib: <fileName>;
```

Default Value

None

Usage Conditions

This property is used in the ***ETLibCertify*** wrapper of the *.etlibcertify* file.

This property can be repeated.

Example

Below is an example of specifying the location of the ATPG library in the *dolphin.etlibcertify* file that will be converted to a scan model library to be used in the certification process:

```
ETLibCertify (dolphin) {  
    TessentCellLib: dolphin/atpg.lib;  
}
```

The generated Verilog scan model library is a single file, named *atpg.scan.v*, and is placed in the *outDir* directory.

SimModelDir

The *SimModelDir* property points to directories containing the simulation models for memories, library cells, and design files.

Syntax

The following syntax specifies this property:

```
SimModelDir: <directoryName>;
```

Default Value

None

Usage Conditions

This property is used in the *ETLibCertify* wrapper of the *.etlibcertify* file.

These usage conditions apply to this property:

- This property can be repeated.
- At least one of the *SimModelDir* property or *SimModelFile* is required. A simulation model for each scan model or pad cell must be provided in one of the simulation libraries.

Example

In this example, the directory where the simulation models for library cells are located is *dolphin/verilog*:

```
ETLibCertify (dolphin) {  
    SimModelDir: dolphin/verilog;  
}
```

SimModelFile

The *SimModelFile* property points to files containing simulation models for memories or library cells.

Syntax

The following syntax specifies this property:

```
SimModelFile: <fileName>;
```

Default Value

None

Usage Conditions

This property is used in the *ETLibCertify* wrapper of the *.etlibcertify* file.

These usage conditions apply to this property:

- This property can be repeated.
- At least one of *SimModelFile* or *SimModelDir* is required. A simulation model for each scan model or pad cell must be provided in one of the simulation libraries.

Example

In this example, the file that contains simulation models for library cells is *dolphin/verilog/pads*:

```
ETLibCertify (dolphin) {  
    SimModelFile: dolphin/verilog/pads.v;  
}
```

VerilogOptionFile

The *VerilogOptionFile* property allows you to point to a file containing simulations options and directives for simulating the technology library cells.

Syntax

The following syntax specifies this property:

```
VerilogOptionFile: <fileName>;
```

Default Value

None

Usage Conditions

This property is used in the *ETLibCertify* wrapper of the *.etlibcertify* file.

Example

In this example, the file that contains simulations options and directives for simulating the technology library cells is *dolphin/simoptions*:

```
ETLibCertify (dolphin) {  
    VerilogOptionFile: dolphin/simoptions;  
}
```


Appendix D

Reference for ETLibCertify Runtime Options

The detailed reference information for the following ETLibCertify runtime options is provided in sections below:

```
etLibcertify <technology> \  
  -log <logFileName> \  
  -outDir <directory> \  
  -mode (Scan) | nonScan
```

where <technology> specifies the name of the technology for which scan models are to be certified.

-log

The **-log** option specifies the name of the log file.

Syntax

The following syntax specifies this wrapper:

```
-log <logFileName>
```

Default Value

The default is *etlibcertify.log*.

Usage Conditions

The name of the log file must specify the relative path to the current working directory or a full path name. The directory in which the log file is to be placed must exist or must be specified with the **-outDir** option.

Example

The following example specifies that the name of the log file is *outDir/Dolphin_etlibcertify.log*:

```
etlibcertify Dolphin -log outDir/Dolphin_etlibcertify.log
```

-outDir

The **-outDir** option specifies the name of the directory in which to place the log file as well as all intermediate files.

Syntax

The following syntax specifies this wrapper:

```
-outDir <directory>
```

Default Value

The default is *outDir*.

Usage Conditions

These usage conditions apply:

- If the specified directory exists then the log file and intermediate files will be placed in that directory, potentially overwriting existing files of the same name.
- If the specified directory does not exist then ETLibCertify will create a new directory.

Example

The following example specifies that the log file and intermediate output files need to be placed in *newOutDir*:

```
etlibcertify Dolphin -outDir newOutDir
```

-mode

The **-mode** option identifies the mode of operation in which ETLibCertify runs.

Syntax

The following syntax specifies this option:

```
-mode (Scan) | nonScan
```

where valid values are as follows:

- *nonScan* — instructs ETLibCertify to only process combinational cells.
- *Scan* — instructs ETLibCertify to process all cells, including sequential cells (flip-flops and latches).

Default Value

The default is *Scan*.

Usage Conditions

These usage conditions apply:

- The *nonScan* mode is useful for customers that don't have access to the full set of tools used in the LV flow. For example, a customer licensed for memory BIST but not for logic BIST can use this mode to generate certified *scan models* for the combinational cells that are used in the Memory BIST flow.
- This option does not influence the impact of the *CellFilter* Wrapper in the .etlibcertify configuration file. In *nonScan* mode the *CellFilter* wrapper is applied to the combinational cells that are left after excluding all sequential cells.

Example

In this example ETLibCertify generates *scan models* for combinational cells only:

```
etlibcertify Dolphin-mode nonScan
```

Related Information

[Reference for the .etlibcertify Input File](#)

[CellFilter Wrapper](#)

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

Documentation

The Tessent software tree includes a complete set of documentation and help files in PDF format. Although you can view this documentation with any PDF reader, if you are viewing documentation on a Linux file server, you must use only Adobe® Reader® versions 8 or 9, and you must set one of these versions as the default using the MGC_PDF_READER variable in your *mgc_doc_options.ini* file.

For more information, refer to “[Specifying Documentation System Defaults](#)” in the *Managing Mentor Graphics Tessent Software* manual.

You can download a free copy of the latest Adobe Reader from this location:

<http://get.adobe.com/reader>

You can access the documentation in the following ways:

- **Shell Command** — On Linux platforms, enter **mgcdocs** at the shell prompt or invoke a Tessent tool with the -Manual invocation switch. This option is available only with Tessent Shell and the following classic point tools: Tessent FastScan, Tessent TestKompress, Tessent Diagnosis, and DFTAdvisor.
- **File System** — Access the Tessent bookcase directly from your file system, without invoking a Tessent tool. From your product installation, invoke Adobe Reader on the following file:

```
$MGC_DFT/doc/pdfiles/_bk_tessent.pdf
```
- **Application Online Help** — You can get contextual online help within most Tessent tools by using the “help -manual” tool command:

> help dofile -manual

This command opens the appropriate reference manual at the “dofile” command description.

Mentor Graphics Support

Mentor Graphics software support includes software enhancements, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service.

For details, refer to this page:

<http://supportnet.mentor.com/about>

If you have questions about a software release, you can log in to SupportNet and search thousands of technical solutions, view documentation, or open a Service Request online:

<http://supportnet.mentor.com>

If your site is under current support and you do not have a SupportNet login, you can register for SupportNet by filling out a short form here:

<http://supportnet.mentor.com/user/register.cfm>

All customer support contact information is available here:

<http://supportnet.mentor.com/contacts/supportcenters/index.cfm>

Third-Party Information

For information about third-party software included with this release of Tessent products, refer to the [*Third-Party Software for Tessent Products*](#).



End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.
3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are

linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. BETA CODE.

- 4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. RESTRICTIONS ON USE.

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.

7. LIMITED WARRANTY.

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor

Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). EXCEPT TO THE EXTENT THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. INFRINGEMENT.

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

11.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. TERMINATION AND EFFECT OF TERMINATION.

12.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or

any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

- 12.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
13. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.