SIEMENS EDA

Tessent™ Cell Library Manual

Software Version 2023.4
Document Revision 31

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Revision History ISO-26262

Revision	Changes	Status/ Date
31	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Dec 2023
30	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Aug 2023
29	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Jun 2023
28	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Mar 2023

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

List of Figures

List of Tables

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.

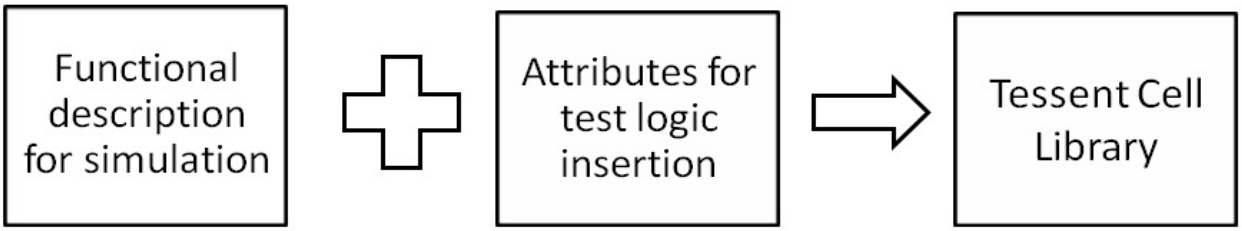
If you are currently using the old ATPG library, you can continue using it with ATPG tools such as Tessent™ FastScan and Tessent™ TestKompress without making any changes. If you want to use a Tessent library in the LV Flow, you must add and certify appropriate attributes for the tools to be used.

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.

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, existing customers do not need 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 and want to create a complete library with models and attributes using the new syntax, refer to section "Attributes and the Tessent Cell Library" on page 373 for information about mapping the old attribute spelling to new attribute spelling. However, it may be easier to write out the merged Tessent cell library to output an old set of library files in the new syntax and see the changes from the old syntax.

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 "How to Generate Simulation Models."

   Note

   If you have an existing LV Flow or cell 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 "How to Add 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."

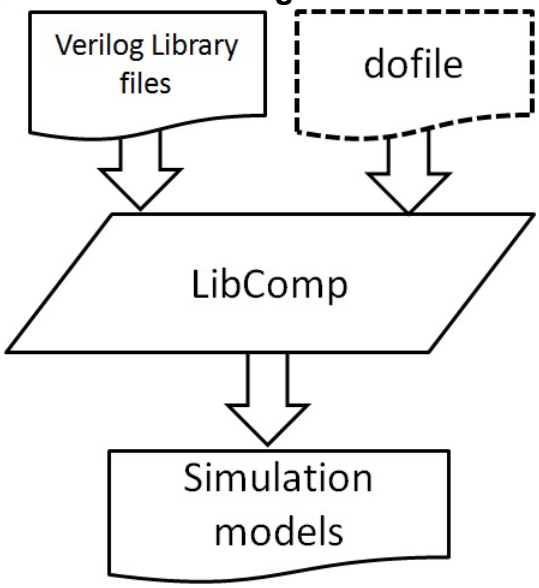
   Tip

   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 373."

How to Generate 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 *Tessent Shell Reference Manual*. For additional information on creating models using LibComp, see Create Tessent Simulation Models Using LibComp."

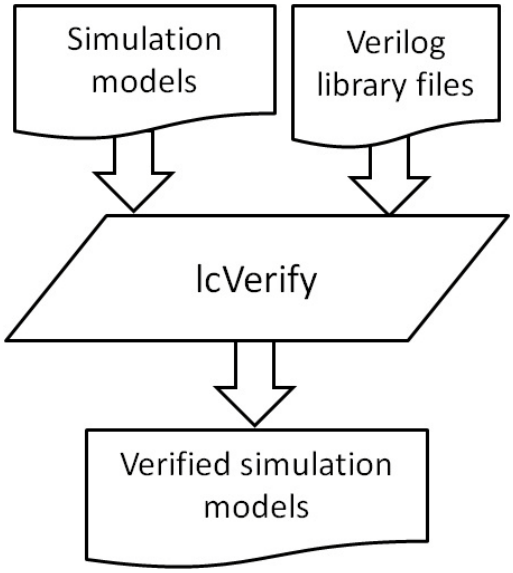
To get a quick reminder of the invocation arguments for LibComp before running it, enter the following on the command line:

```
libcomp -help
```

How to Verify 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*** *cell_library_name Verilog_library_names*

For more information, see "Verification of Tessent Simulation Models".

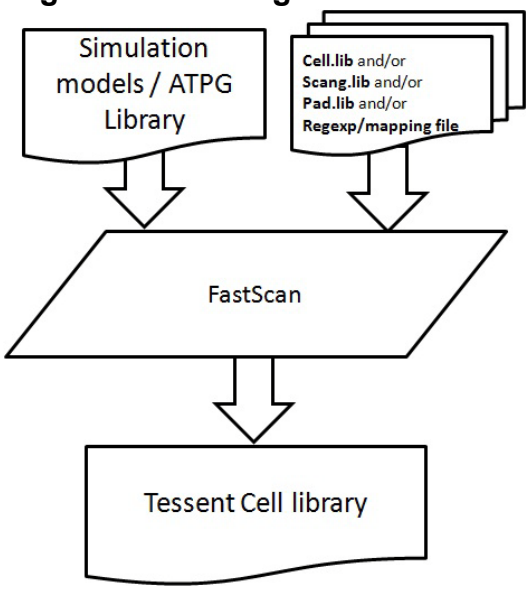
How to Add Cell Attributes to Simulation Models

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 "How to Define Cell Information" on page 95.

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



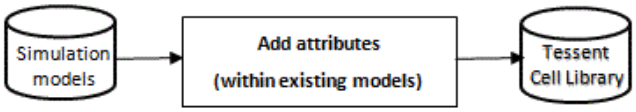
- Method 1 — Extract cell information from Liberty. See "Create Tessent Insertion Attributes Using Liberty".
- Method 2 — Define cell attributes using simple and regular expressions in attribute definition syntax. See "Cell Attribute Population With Simple and Regular Expressions."
- Method 3— Manually 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 the other listed Methods. See "Manual Creation of Cell Attributes."
- Method 4— Merge pre-existing attributes from LV Flow in *cell.lib*, *pad.lib*, *scang.lib* files with pre-existing cell 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 3 of section "Primitive and Attribute Examples" on page 158.

Cell Attribute Population With Simple and Regular Expressions

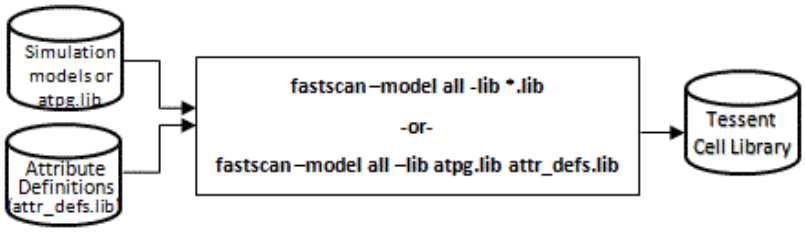
You can define cell attributes in an attribute definition file that is then read in during a library parser invocation and merged with the simulation models (or cell library models) to create a single source library.

This is shown in Figure 1-6 below. In the attribute definition file, you can (1) simply list every model and cell attribute and, (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 158."

For complete information on using regular expressions to update attributes with pattern matching, see section "Cell Selection."

When the simulation models (or cell 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 cell library to create a Tessent cell library.

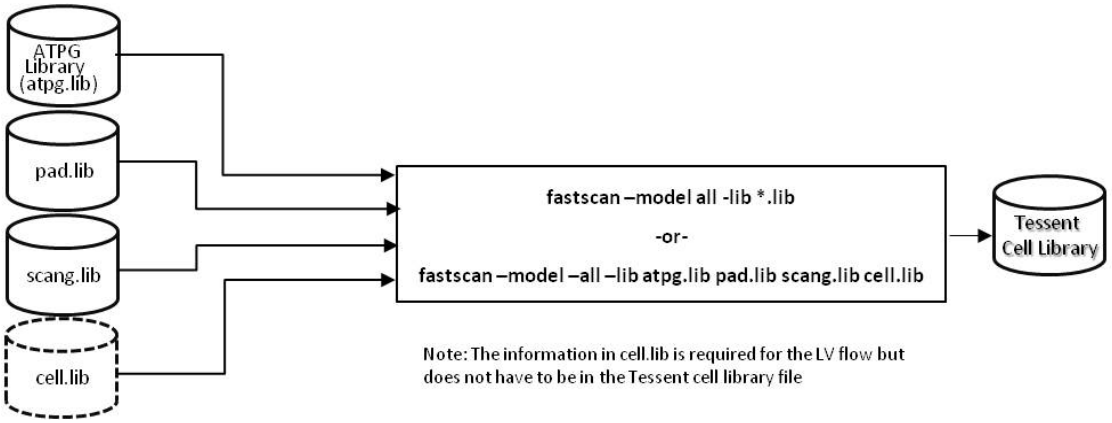
Figure 1-7 illustrates the method used to merge LV libraries. The LV libraries already have attributes. This approach is shown in Method 4 of section "Primitive and Attribute Examples" on page 158.

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 cell library, you can generate one from Verilog files as described in "How to Generate 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

1. You can write out a single library file of all of the library files loaded at invocation, by performing the following steps:
2. 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

3. Once inside the tool, run 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

4. For more information on the invocation arguments, see the tessent shell command in the *Tessent Shell Reference Manual*.
5. For more information on the write_cell_library command, see write_cell_library in the *Tessent Shell Reference Manual*.

Chapter 2
Library Model Creation

The Tessent tools require simulation models for test simulations. For some of these models (only those appropriate to use when the Tessent test tools insert logic for testing the system hardware) attributes for those models and their pins may also be needed.

This chapter illustrates the process of creating a simulation model with the necessary attributes and the model attributes and pin attributes when those are appropriate.

One frequently misunderstood model attribute is “cell_type”. The “cell_type” attribute is never needed for ATPG or simulation, but only for Tessent test tools to know the model is appropriate to insert to implement test access logic. For a cell_type statement to be appropriate, the model must be a model of a cell, and its transfer function must meet the expectations of the test tools for that specified cell_type. For example, any multiplexer with an inverting data transfer should not be given a cell_type = mux. The test tools expect (and require) a non-inverting data transfer from all of the mux inputs to the mux output. Another frequently misunderstood cell_type is “scan_cell”. “cell_type = scan_cell” should only be placed on a small subset of all scanned sequential cells. Neither nonscan <=> scan replacement nor scan chain stitching use “cell_type = scan_cell” information. Test point insertion, on the other hand, does want an appropriate cell to use for a test point, as indicated by “cell_type = scan_cell”. Such a cell should be a simple DFF with a scan mux in front of it. There should be only one data_in input going to the mux, only one scan_in input, and one scan_enable or scan_enable_inv controlling which input of the mux is transferred to the DFF D input. Finally, there must be a non-inverting scan_out output for “cell_type = scan_cell” to be appropriate. The examples in this chapter illustrate appropriate cells of various cell types, and their model and pin function attributions needed if they are appropriate for insertion for test logic.

How to Create a Positive-Edge (Nonscan) DFF Cell

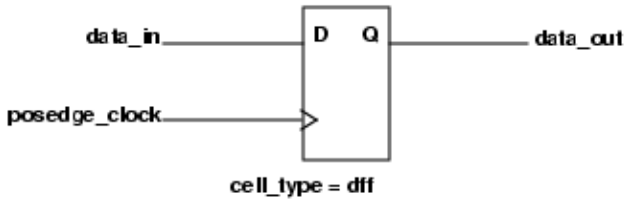
Assume you want to model a positive-edge DFF. You need to first translate the DFF from Verilog and then add the Tessent attributes used for test insertion. The model is required for almost all flows. The Tessent attributes are needed for test logic insertion and for scan replacement and stitching to create scan chains within nonscan netlists.

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;

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. If you are going to use Tessent tools for scan stitching or test insertion, you should add the attributes to the models that correspond to the cells illustrated in this chapter, using one of the methods described in section “Cell Library” on page 93. You would not add attributes such as cell_type to models that do not match the functions illustrated. For example, an AOI (AND OR INVERT) or a mux with inverting datapath would not be given a cell_type attribute or pin_type attribute. After the attributes are added, the final

Tessent Cell Library contains the following contents; green font indicates the attributes that it is recommended you add:

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

library_format_version = 9;

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
// "dft_cell_selection_name".
// CAVEAT: If multiple posedge_dff are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.
```

```
dft_cell_selection(dft_cell_selection_name)
{
   posedge_dff = pos_dff_cell;
}
```

Buffer

This section describes the attributes required for a buffer cell.

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

The green font indicates the attributes that Siemens EDA recommends you add.

Clock Buffer

This section describes the attributes required for a clock buffer cell.

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

The red font indicates the attributes you must add.

Buffer Cell With Unused Inputs

This section describes the attributes required for a buffer cell.

Model attributes: cell_type = buffer

If a buffer has more than one unused input, the cell_type must be declared to distinguish from an Input only IO Pad.

Pin attributes: unused

Buffer With Unused Inputs

This section describes the attributes required for a buffer cell with more than one unused input.

```
model buff
(A, VDD, VSS, Y)
(
cell_type = buffer:
model_source = verilog_module;
input (A) ( )
input (VDD) (unused) // Unused attribute avoids warning messages
input (VSS) (unused) // Unused attribute avoids warning messages
output (Y) ()
   (
       primitive = _buf buf_inst (A, Y);
   )
)
```

The red font indicates the attributes you must add. The green font indicates the attributes that Siemens EDA recommends you add.

Inverter

This section describes the attributes required for an Inverter cell.

```
model Inv
(A, Y)
(
cell_type = inverter:
model_source = verilog_module;
input (A) ( )
output (Y) ()
   (
       primitive = _inv Inv_inst (A, Y);
   )
)
```

The green font indicates the attributes that Siemens EDA recommends you add.

Clock Inverter

This section describes the attributes required for an Clock Inverter cell.

```
model Clk_Inv
(A, Y)
(
cell_type = clock_inverter:
model_source = verilog_module;
input (A) ( )
output (Y) ()
   (
       primitive = _inv Inv_inst (A, Y);
   )
)
```

The red font indicates the attributes you must add.

Inverter Cell With Unused Inputs

This section describes the attributes required for an inverter cell.

Model attributes: cell_type = inverter

If an inverter has more than one unused input, the cell_type must be declared to distinguish from an inverting Input only IO Pad.

Pin attributes: unused

The red font indicates the attributes you must add. The green font indicates the attributes that Siemens EDA recommends you add.

Inverter With Unused Inputs

This section describes the attributes required for a Inverter cell with more than one unused input.

```
model invert
(A, VDD, VSS, Y)
(
cell_type = inverter:
model_source = verilog_module;
input (A) ( )
input (VDD) (unused) // Unused attribute avoids warning messages
input (VSS) (unused) // Unused attribute avoids warning messages
output (Y) ()
   (
       primitive = _inv inv_inst (A, Y);
   )
)
```

The red font indicates the attributes you must add. The green font indicates the attributes that Siemens EDA recommends you add.

Pad Cell

This section describes the attributes required for a pad cell.

Model attributes:

```
cell_type = pad
```

Pin attributes:

```
pad_enable_high or pad_enable_low
pad_to_pad
pad_pad_io, pad_to_io or pad_from_io
pad_from_pad
pad_data_inv
pad_input_enable_high or pad_input_enable_low
pad_open_drain or pad_open_source
pad_tied0
pad_tied1
```

Input Pad

This section describes the attributes required for an Input Pad cell.

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_from_io pad_from_pad;
```

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

```
model ipad
(PAD, C)
(
model_source = verilog_module;
cell_type = pad;

input (PAD) ( pad_from_io; )
output (C) ( pad_from_pad; )
(
instance = buf02 ipad_inst (PAD, C);
)
) // end model ipad
```

An Inverting Input Pad cell requires the attributes described below.

Model attributes: cell_type = pad;

Illustrated Pin attributes: pad_from_iopad_from_pad pad_data_inv;

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

```
model Inverting_Pad
  (C_b,PAD)
(
  cell_type = pad;
  input (PAD) ( pad_from_io;)
  output (C_b) (pad_from_pad; pad_data_inv)

  (
   primitive = _inv P5 (PAD, C_b);
  )
)
```

An Input Pad cell with Active High Input Enable requires the attributes described below.

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_from_io pad_from_pad pad_input_enable_high;
```

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

```
model ipad_ien
(PAD, C, IEN)
(
   model_source = verilog_module;
   cell_type = pad;

   input (PAD) ( pad_from_io; )
   input (IEN) ( pad_input_enable_high; )
   output (C)  ( pad_from_pad; )
(
   // Input a 0 (low voltage) to core when input disabled.
   primitive = _and mlc_and (PAD, IEN, C);
)
) // end model ipad_ien
```

An Input Pad cell with Active Low Input Enable requires the attributes described below.

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_from_io pad_from_pad pad_input_enable_low;
```

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

```
model ipad_ienb
(PAD, C, IENB)
(
   model_source = verilog_module;
   cell_type = pad;

   input (PAD) ( pad_from_io; )
   input (IEN) ( pad_input_enable_low; )
   output (C)  ( pad_from_pad; )
(
   // Input a 1 (high voltage) to core when input disabled.
   primitive = _or mlc_or (PAD, IENB, C);
)
) // end model ipad_ien
```

A Differential Input Pad cell requires the attributes described below (note that pad_diff_voltage might be pad_diff_current).

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_from_io  pad_diff_voltage  pad_from_io_invpad_from_pad;
```

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

```
model Diff_Pad
  (C,PADP,PADN)
(
  cell_type = pad;

  input (PADP) ( pad_from_io; pad_diff_voltage)
  input (PADN) ( pad_from_io_inv; )
  output (C) ( pad_from_pad )

  (
    primitive = _buf P1 (PADP, diff_net);
    primitive = _inv P2 (PADN, diff_net);
    primitive = _buf P3 (diff_net, C);
   )
)
```

You can see another "Differential Input Pad Example" in the *Support for IEEE 1149.6 Boundary Scan* manual.

An input pad with alternative from pad ports going to the core (only one of which is connected outside the pad) is specified using two modes, as follows.

```
model ipad_alt_from_pad
(PAD, C, CH)
(
model_source = verilog_module;
cell_type = pad;
mode (
  input (PAD) ( pad_from_io; )
  output (C) ( pad_from_pad; )
  output (CH) ( pad_open; )
)  // end mode 1, for instantiations where only C is connected outside
mode (
  input (PAD) ( pad_from_io; )
  output (C) ( pad_open; )
  output (CH) ( pad_from_pad; )
)  // end mode 2, for instantiations where only CH is connected outside

input (PAD) ( )
output (C) ( )
output (CH) ( )
(
  instance = buf02 ipad_inst_1 (PAD, C);
  instance = buf02 ipad_inst_2 (PAD, CH);
)
) // end model ipad_alt_from_pad
```

Output Pad

This section describes the attributes required for an Output Pad cell.

Model attributes: cell_type = pad;

Illustrated Pin attributes: pad_to_pad pad_enable_highpad_to_io

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

```
model opad
(I, OEN, PAD)
(
model_source = verilog_module;
cell_type = pad;

input (I) ( pad_to_pad; )
input (OEN) ( pad_enable_high; )
output (PAD) ( pad_to_io; )
(
instance = tri01 opad_inst (I, OEN, PAD);
)
) // end model opad
```

An open_drain output pad, with output Z from PAD when DO_EN = 0 (input mode) or when DO_EN = 1 (output mode) and outputting a logical 1 (becomes Z for open_drain), should be specified as below.

```
model open_drain_output_pad
 (DO, DO_EN, PAD)
(
   model_source = verilog_module;
   cell_type = pad;
   mode(
     pin (DO) (pad_to_pad; )
     pin (DO_EN) (pad_enable_high; )
     pin (PAD) (pad_pad_io; pad_open_drain)
   )
   input (DO) ( )
   input (DO_EN) ( )
   inout (PAD) ( )
   (
     // Turn off _tsh when DO is 1 to get Z out
     primitive = _inv mlc_1_to_z_inv (DO, DOinv);

     // Turn off unless DO=0 and DO_EN=1
     primitive = _and mlc_1_to_z (DOinv, DO_EN, open_drain_enable);

     primitive = _tsh mlc_tsh_1 (DO, open_drain_enable, PAD);
   )
)
```

You can see a "Differential Output Pad Example" in the *Support for IEEE 1149.6 Boundary Scan* manual.

Inout Pad

This section describes the attributes required for an Inout Pad cell.

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_enable_high pad_to_pad pad_pad_io pad_from_pad
```

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

```
model iopad
(OEN, PAD, I, C)
(
model_source = verilog_module;
cell_type = pad;

input (OEN) ( pad_enable_high; )
input (I) ( pad_to_pad; )
inout (PAD) ( pad_pad_io; )
output (C) ( pad_from_pad; )
(
instance = ipad iopad_ipad_inst (PAD, C);
instance = opad iopad_opad_inst (I, OEN, PAD);
)
) // end model iopad
```

An inout pad with active low input enable requires the attributes specified as follows:

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_enable_high pad_to_pad pad_pad_io pad_from_pad pad_input_enable_low
```

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

```
model iopad_ienb
(OEN, PAD, I, C, IENB)
(
model_source = verilog_module;
cell_type = pad;

input (OEN) ( pad_enable_high; )
input (I) ( pad_to_pad; )
input (IENB) ( pad_input_enable_low; )
inout (PAD) ( pad_pad_io; )
output (C) ( pad_from_pad; )
(
instance = ipad_ienb iopad_ipad_inst (PAD, C, IENB);
instance = opad iopad_opad_inst (I, OEN, PAD);
)
) // end model iopad_ienb
```

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

An inout pad with active high input enable requires the attributes specified as follows:

Model attributes:

```
cell_type = pad;
```

Illustrated Pin attributes:

```
pad_enable_high pad_to_pad pad_pad_io pad_from_pad pad_input_enable_high
```

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

```
model iopad_ien
(OEN, PAD, I, C, IEN)
(
model_source = verilog_module;
cell_type = pad;

input (OEN) ( pad_enable_high; )
input (I) ( pad_to_pad; )
input (IEN) ( pad_input_enable_high; )
inout (PAD) ( pad_pad_io; )
output (C) ( pad_from_pad; )
(
instance = ipad_ien iopad_ipad_inst (PAD, C, IEN);
instance = opad iopad_opad_inst (I, OEN, PAD);
)
) // end model iopad_ien
```

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

An inout pad with alternative from pad ports going to the core (only one of which is connected outside the pad) is specified using two modes, as follows:

```
model iopad_alt_from_pad
(OEN, PAD, I, C, CH)
(
model_source = verilog_module;
cell_type = pad;
mode (
  input (OEN) ( pad_enable_high; )
  input (I) ( pad_to_pad; )
  inout (PAD) ( pad_pad_io; )
  output (C) ( pad_from_pad; )
  output (CH) ( pad_open ))
// end mode 1, for instantiations where only C is connected outside
mode (
  input (OEN) ( pad_enable_high; )
  input (I) ( pad_to_pad; )
  inout (PAD) ( pad_pad_io; )
  output (C) ( pad_open; )
  output (CH) ( pad_from_pad )
)  // end mode 2, for instantiations where only CH is connected outside
```

```
input (OEN) ( )
input (I) ( )
inout (PAD) ( )
output (C) ( )
output (CH) ( )
(
  instance = ipad iopad_ipad_inst (PAD, C);
  instance = ipad iopad_ipad_inst (PAD, CH);

  instance = opad iopad_opad_inst (I, OEN, PAD);
)
) // end model iopad_alt_from_pad
```

An open_drain inout pad, with output Z from PAD when DO_EN = 0 (input mode) or when DO_EN = 1 (output mode) and outputting a logical 1 (becomes Z for open_drain), is specified as follows:

```
model open_drain_bidi_pad
 (DI, DO, DO_EN, PAD)
(
   model_source = verilog_module;
   cell_type = pad;
   mode(
     pin (DI) (pad_from_pad; )
     pin (DO) (pad_to_pad; )
     pin (DO_EN) (pad_enable_high; )
     pin (PAD) (pad_pad_io; pad_open_drain)
   )
   input (DO) ( )
   input (DO_EN) ( )
   inout (PAD) ( )
   output (DI) ()
   (
     primitive = _buf mlc_buf_1 (PAD, DI);

     // Turn off _tsh when DO is 1 to get Z out
     primitive = _inv mlc_1_to_z_inv (DO, DOinv);
     // Turn off unless DO=0 and DO_EN=1
     primitive = _and mlc_1_to_z (DOinv, DO_EN, open_drain_enable);

     primitive = _tsh mlc_tsh_1 (DO, open_drain_enable, PAD);
   )
)
```

An open_source inout pad, with output Z from PAD when DO_EN = 0 (input mode) or when DO_EN = 1 (output mode) and outputting a logical 0 (becomes Z for open_source), is specified as follows:

```
model open_source_bidi_pad
(DI, DO, DO_EN, PAD)
(
   model_source = verilog_module;
   cell_type = pad;
   mode(     pin (DI) (pad_from_pad; )
     pin (DO) (pad_to_pad; )
     pin (DO_EN) (pad_enable_high; )
     pin (PAD) (pad_pad_io; pad_open_source)
   )
   input (DO) ( )
   input (DO_EN) ( )
   inout (PAD) ( )
   output (DI) ()
   (
     primitive = _buf mlc_buf_1 (PAD, DI);
     // Turn off unless DO=1 and DO_EN=1
     primitive = _and mlc_1_to_z (DO, DO_EN, open_source_enable);
     primitive = _tsh mlc_tsh_1 (DO, open_source_enable, PAD);
   )
)
```

An inout pad that is programmable (using pad_tied attributes with cell inputs tied correspondingly where pad cell is instantiated in netlist) to be an inout pad with no pulls, or an inout pad with pullup, or an inout pad with pulldown is specified as follows:

```
model IOPUPD_with_modes
  (DIN, DOUT, ENB, PAD,
   ENB_IN, PD_EN, PU_ENB)
(
  model_source = verilog_module;
  cell_type = pad;
  mode(
    input (DOUT) (pad_to_pad; )
    input (ENB) (pad_enable_low; )
    input (ENB_IN) (pad_input_enable_low; )
    input (PD_EN) (pad_tied0; )
    input (PU_ENB) (pad_tied1; )
    inout (PAD) (pad_pad_io; )
    output (DIN) (pad_from_pad; )
  )
  mode(
    input (DOUT) (pad_to_pad; )
    input (ENB) (pad_enable_low; )
    input (ENB_IN) (pad_input_enable_low; )
    input (PD_EN) (pad_tied1; )
    input (PU_ENB) (pad_tied1; )
    inout (PAD) (pad_pad_io; pad_pull0; )
    output (DIN) (pad_from_pad; )
  )
  mode(
    input (DOUT) (pad_to_pad; )
    input (ENB) (pad_enable_low; )
    input (ENB_IN) (pad_input_enable_low; )
    input (PD_EN) (pad_tied0; )
    input (PU_ENB) (pad_tied0; )
    inout (PAD) (pad_pad_io; pad_pull1; )
    output (DIN) (pad_from_pad; )
  )

  input (DOUT) ( )
  input (ENB) ( )
  input (ENB_IN) ( )
  input (PD_EN) ( )
  input (PU_ENB) ( )
  inout (PAD) ( )
  output (DIN) ( )
  (
    primitive = _tie1 mlc_tie1_1 (v_vdd);
    primitive = _tie0 mlc_tie0_1 (v_gnd);
    primitive = _tsl mlc_tsl_1 (DOUT, ENB, PAD);
    primitive = _rpmos mlc_rpmos_1 (v_vdd, PU_ENB, PAD);
    primitive = _rnmos mlc_rnmos_1 (v_gnd, PD_EN, PAD);
    primitive = _inv mlc_inv_1 (ENB_IN, en_in);
    primitive = _and mlc_and_1 (PAD, en_in, DIN);
  )
) // end model IOPUPD_with_modes
```

Buffer and Input Pad

This section describes the attributes required for a Buffer and Input Pad cell.

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

The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

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

Programmable Pad

This section describes the attributes required for a Programmable Pad cell

```
model modepad
   (io, fp, tp, en, modein)
(
  model_source = verilog_module;
  cell_type = pad;
  mode(
    pin (io) (pad_pad_io; )
    pin (fp) (pad_from_pad; )
    pin (modein) (pad_tied1; )
  )
  mode(
    pin (io) (pad_pad_io; )
    pin (tp) (pad_to_pad; )
    pin (en) (pad_enable_high; )
    pin (modein) (pad_tied0; )
  )

  input (tp) ( )
  input (en) ( )
  input (modein) ( )
  inout (io) ( )
  output (fp) ( )
  (
    primitive = _inv mlc_inv_1 (modein, modeininv);
    primitive = _and mlc_and_1 (andin, modein, fp);
    primitive = _and mlc_and_2 (en, modeininv, anden);

    primitive = _buf mlc_buf_1 (io, andin);
    primitive = _tsh mlc_tsl_1 (tp, anden, io);
  )
) // end model modepad
```

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 are illustrated. Others are the same except the cell_type and the primitive implementing the function change from “and” to “nand” for a cell_type = nand illustration, and similarly for the other symmetric cell types. Also, only cell_type = or and cell_type = and can have data_in_inv pin functions. For the clock_or cell, change the cell_type “clock_and” to “clock_or”.

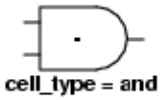
AND Cells

This section describes the attributes required for an AND cell.

Illustrated Model attributes: cell_type = and

Pin attributes: data_in data_in_inv clock_in data_out

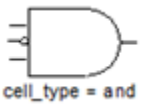
Figure 2-2. 2-Input AND Cell



The final Tessent Cell Library is shown here. The green font indicates the attributes that it is recommended you add.

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

Figure 2-3. 3-Input AND Cell



The final Tessent Cell Library is shown here. The green font indicates the attributes that it is recommended you add.

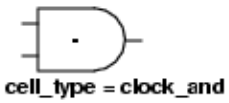
```
model data_and3
  (out, in1, in2, in3)
(
  model_source = verilog_module;
  cell_type = and;
  input (in1) (data_in )
  input (in2) (data_in_inv )
  input (in3) (data_in )
  output (out) (data_out)
  (
      primitive = _inv  inv_inst (in2, in2_inv);
      primitive = _and  and_inst ( in1, in2_inv, in3, out );
  )
)
```

Note that cell_type is learned for non-inverting symmetric functions if a mix of inverting and non-inverting inputs exists. This is because these cell types can be used for intercepts, and intercepts must be non-inverting cell_type, with at least one non-inverting input to use for the intercept. Although the cell below is modeled as a NOR with the 1st and 3rd inputs inverted, it is learned and used for intercept purposes as a non-inverting 'and' cell_type, with the other inputs (in this case the 2nd input) inverted.

```
model data_nor3
  (out, in1, in2, in3)
(
  model_source = verilog_module;
  cell_type = and;
  input (in1) (data_in )
  input (in2) (data_in_inv )
  input (in3) (data_in )
  output (out) (data_out)
  (
    primitive = _inv   inv_inst1 (in1, in1_inv);
    primitive = _inv   inv_inst3 (in3, in3_inv);
    primitive = _nor  nor_inst ( in1_inv, in2, in3_inv, out );
  )
)
```

Similarly, a cell with _nand inside, with some inputs inverting, is learned as “cell_type = or;” with the other inputs inverted (those that were non-inverting originally going into the _nand primitive).

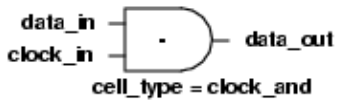
Figure 2-4. 2-Input Clock AND Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
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 "dft_cell_selection_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(dft_cell_selection_name) {
   and = data_and3, data_and2; // For data path.
   clock_and = clk_and;        // For clock path.
}
```

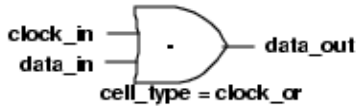
Figure 2-5. 2-Input Clock AND Cell With Explicit Clock Pin



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
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 "dft_cell_selection_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(dft_cell_selection_name) {
   and = data_and3, data_and2; // For data path.
   clock_and = clock_and_cell;        // For clock path.
```

Figure 2-6. 2-Input Clock OR Cell With Explicit Clock Pin



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
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 "dft_cell_selection_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(dft_cell_selection_name) {
   or = data_or3, data_or2; // For data path.
   clock_or = clock_or_cell;        // For clock path.
}
```

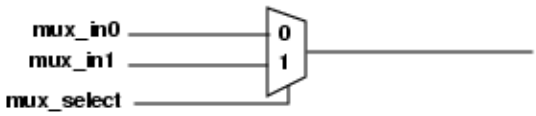
MUX Cell

This section describes the attributes required for a multiplexer.

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 2-7. 2:1 MUX Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
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 "dft_cell_selection_name".
//
```

```
dft_cell_selection(dft_cell_selection_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.
}
```

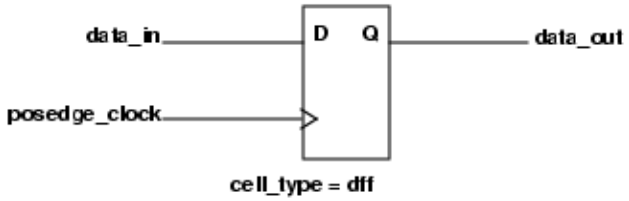
Positive-Edge (Nonscan) DFF Cell

This section describes the attributes required for a positive-edge nonscan DFF.

Model attributes: cell_type = dff

Pin attributes: data_in posedge_clock data_out

Figure 2-8. Positive-Edge DFF Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; the green font indicates the attributes that it is recommended you add.

```
model pos_dff_cell
  (q, clk, din)
(
  cell_type = dff;  // Added cell_type attribute.
  // Following required if tessent scan to be used for nonscan=>scan
  //   replacement. Can be provided by read_liberty, else must be
  //   provided when tessent cell library created.
  // To replace pos_dff_cell with pos_scan_dff_cell scan cell :
  scan_equivalent = pos_scan_dff_cell;

  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
// "dft_cell_selection_name".
// CAVEAT: If multiple posedge_dff are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.

dft_cell_selection(dft_cell_selection_name)
{
   posedge_dff = pos_dff_cell;
}
```

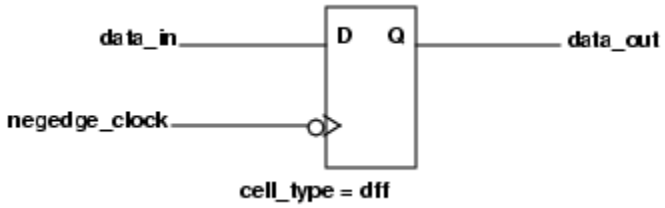
Negative-Edge (Nonscan) DFF Cell

This section describes the attributes required for a negative-edge nonscan DFF.

Model attributes: cell_type = dff

Pin attributes: data_in negedge_clock data_out

Figure 2-9. Negative-Edge Nonscan DFF Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; the green font indicates the attributes that it is recommended you add.

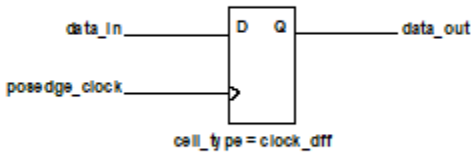
```
model neg_dff_cell
  (q, clk, data)
(
  cell_type = dff;  // Added cell_type attribute.
 // Following required if tessent scan to be used for nonscan=>scan
//   replacement.  Can be provided by read_liberty, else must be
//   provided when tessent cell library created.
// To replace neg_dff_cell with neg_scan_dff_cell scan cell :

  scan_equivalent = neg_scan_dff_cell;

  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
// "dft_cell_selection_name".
// CAVEAT: If multiple negedge_dff are specified for the same
// dft_cell_selection_name, the last parsed is kept as
// the one to use.

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

Positive-Edge Clock DFF Cell

This section describes the attributes required for a positive-edge clock DFF.

Pin attributes: data_in posedge_clock data_out

Figure 2-10. Positive-Edge Clock DFF Cell



The final Tessent Cell Library is shown here. The red font indicates attributes that you must add. The green font indicates the attributes that it is recommended you add.

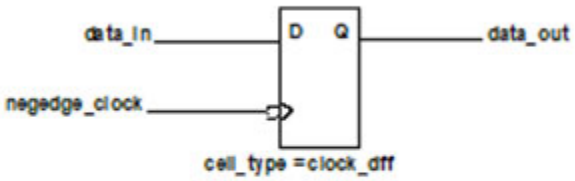
```
model pos_clk_dff_cell
  (q, clk, din)
(
  cell_type = clock_dff;  // Added cell_type attribute.

  input (clk) (posedge_clock) // Added pin attribute (function)
  input (din) (data_in)       // Added pin attribute (function)
  output (q) (data_out)       // Added pin attribute (function)
  (
    primitive = _dff dff_inst ( , , clk, din, q, );
  )
)

//******************* CELL SELECTION ADDED ************************
// Ensure that Tessent test tools use pos_clk_dff_cell if they insert a
// posedge clock dff. Sometimes, users have multiple technology cells
// (low power and non low power for example) and they can specify
// one posedge and one negedge clock dff for each technology. In this
// example, only one technology exists, with the name
// "dft_cell_selection_name".
// CAVEAT: If multiple posedge_clock dff are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.

dft_cell_selection(dft_cell_selection_name)
{
   posedge_clock_dff = pos_clk_dff_cell;
}
```

Negative-Edge Clock DFF Cell

This section describes the attributes required for a negative-edge clock DFF.

Pin attributes: data_in negedge_clock data_out

Figure 2-11. Negative-Edge Clock DFF Cell



The final Tessent Cell Library is shown here. The red font indicates attributes that you must add. The green font indicates the attributes that it is recommended you add.

```
model neg_clk_dff_cell
  (q, clk, din)
(
  cell_type = clock_dff;  // Added cell_type attribute.

  input (clk) (negedge_clock) // Added pin attribute (function)
  input (din) (data_in)       // Added pin attribute (function)
  output (q) (data_out)       // Added pin attribute (function)
  (
    primitive = _dff dff_inst ( , , clk, din, q, );
  )
)

//******************* CELL SELECTION ADDED ************************
// Ensure that Tessent test tools use neg_clk_dff_cell if they insert a
// negedge clock dff. Sometimes, users have multiple technology cells
// (low power and non low power for example) and they can specify
// one posedge and one negedge clock dff for each technology. In this
// example, only one technology exists, with the name
// "dft_cell_selection_name".
// CAVEAT: If multiple negedge_clock dff are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.

dft_cell_selection(dft_cell_selection_name)
{
   negedge_clock_dff = neg_clk_dff_cell;
}
```

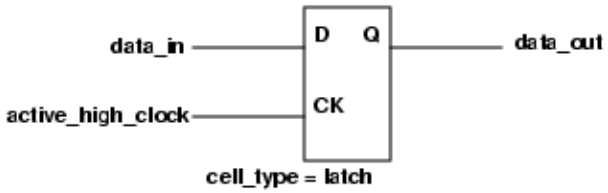
Active High Latch Cell

This section describes the attributes required for an active high Latch cell.

Model attributes: cell_type = latch

Pin attributes: data_in active_high_clock data_out

Figure 2-12. 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
// "dft_cell_selection_name".
// CAVEAT: If multiple active_high_latch are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.

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

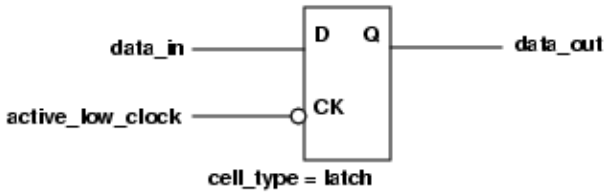
Active Low Latch Cell

This section describes the attributes required for an active low Latch cell.

Model attributes: cell_type = latch

Pin attributes: data_in active_low_clock data_out

Figure 2-13. Active Low Latch Cell



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
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
// "dft_cell_selection_name".
// CAVEAT: If multiple active_high_latch are specified for the same
// dft_cell_selection_name, the last parsed is kept as the
// one to use.

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

MUX Scan Cell

This section describes the attributes required for a MUX scan cell.

The red font indicates the attributes you must add; the green font indicates the attributes that it is recommended you add. For more examples, see Example Scan Definitions.

Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;

cell_type = scan_cell;
// Only used if scan cell is appropriate for a test point
cell_type = prohibited;
// Prevents the scan cell from being used as a test point.
```

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

This section describes the attributes required for a positive-edge mux scan cell.

Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;

cell_type = scan_cell;
// Only used if scan cell is appropriate for a test point.
```

Illustrated Pin attributes:

```
data_in
scan_in
scan_enable
posedge_clock
scan_out
```

Figure 2-14. Positive-Edge MUX DFF Scan Cell

The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
model mux_scan
  (q, clk, din, sin, sen)
(
  model_source = verilog_module;
// Following is required for tessent scan shift register scan=>nonscan
// replacement. To replace mux_scan by nonscan pos_dff_cell :
  nonscan_equivalent = pos_dff_cell; /

// Only use for simple mux scan cell appropriate as test point.
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
// dft_cell_selection_name, the last parsed is kept as the one
// to use.

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

Positive-Edge MUX DFF Scan Cell (Non Test Point)

This section describes the attributes required for a positive-edge mux scan cell.

Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;

cell_type = prohibited;
// Prevents using this scan cell for a test point.
```

Illustrated Pin attributes:

```
data_in
scan_in
scan_enable
posedge_clock
scan_out
```

Figure 2-15. Positive-Edge MUX DFF Scan Cell (Non Test Point)

The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
model mux_scan
  (q, clk, din, sin, sen)
(
  model_source = verilog_module;
  nonscan_model = pos_dff_cell; // Non-scan to be replaced
  cell_type = prohibited;
  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) );
  )
)
//*************** DO NOT ADD WITHIN CELL SELECTION ***************
// Because this scan cell is not to be used as a test point, do not
// add the cell as a posedge_scan_cell in the dft_cell_selection.
dft_cell_selection(dft_cell_selection_name) {
   //Do not declare as: posedge_scan_cell = mux_scan;
}
```

Negative-Edge MUX Scan Cell

This section describes the attributes required for a negative-edge mux scan cell.

Model attributes:

```
// Required to replace scan with nonscan, for example,
// for shift register scan replacement.
```

```
nonscan_equivalent = nonscan_model_name;

cell_type = scan_cell;
// Only used if scan cell is appropriate for a test point.
```

Illustrated Pin attributes:

```
data_in
scan_in
scan_enable
negedge_clock
scan_out
```

Figure 2-16. Negative-Edge MUX DFF Scan Cell

The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
model neg_mux_scan
   (q, clk, din, sin, sen)
(
  model_source = verilog_module;
// Following is required for tessent scan shift register scan=>nonscan
// replacement. To replace mux_scan by nonscan neg_dff_cell :
  nonscan_equivalent = neg_dff_cell;

  cell_type = scan_cell;
// Only use for simple mux scan cell appropriate as test point.
  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
// dft_cell_selection_name, the last parsed is kept as the one
// to use.

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

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

This section describes the attributes required for a positive-edge MUX scan cell with an active low scan_enable and inverting scan_out pin.

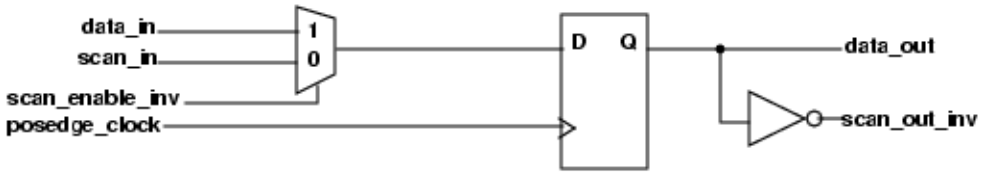
Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;

cell_type = scan_cell;
// Only used if scan cell is appropriate for a test point.
```

Illustrated Pin attributes:

```
data_in
scan_in
scan_enable_inv
posedge_clock
data_out
scan_out_inv
```

Figure 2-17. 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 must add; green font indicates the attributes that it is recommended you add.

```
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 );
  )
)
//******************* CELL SELECTION ADDED ************************
// To ensure that Tessent test tools use pos_mux_scan if they insert
// a posedge scan cell.
// CAVEAT: If multiple posedge_scan_cell are specified for the same
// dft_cell_selection_name, the last parsed is kept as the one
// to use.

dft_cell_selection(dft_cell_selection_name)
{
   posedge_scan_cell = pos_mux_scan;
}
```

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

This section describes the attributes required for a positive-edge MUX scan cell with inverting and non-inverting scan_out.

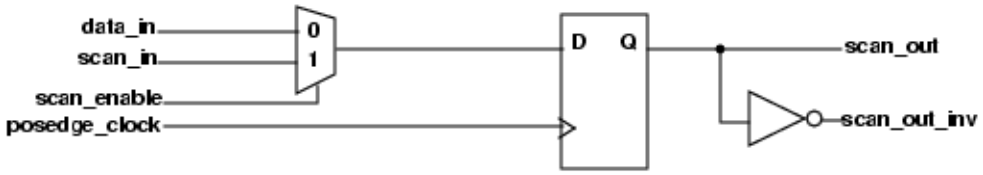
Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;
cell_type = scan_cell;
// Only used if scan cell is appropriate for a test point.
```

Illustrated Pin attributes:

```
data_in
scan_in
scan_enable_inv
posedge_clock
scan_out
scan_out_inv
```

Figure 2-18. 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 must add; green font indicates the attributes that it is recommended you add.

```
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 );
  )
)
//******************* CELL SELECTION ADDED ************************
// To ensure that Tessent test tools use pos_mux_scan_both_out if
// they insert a posedge scan cell.
// CAVEAT: If multiple posedge_scan_cell are specified for the same
// dft_cell_selection_name, the last parsed is kept as the one
// to use.

dft_cell_selection(dft_cell_selection_name)
{
   posedge_scan_cell = pos_mux_scan_both_out;

}
```

Positive-Edge MUX Scan Cell With Retention Enable

This section describes the attributes required for a positive-edge mux scan cell with retention enable.

Model attributes:

```
// Required to replace scan with nonscan, for example,
//      for shift register scan cell replacement.

nonscan_equivalent = nonscan_model_name;
cell_type = prohibited;
// Prevents using this scan cell for a test point.
```

Illustrated Pin attributes:

```
data_in
posedge_clock
scan_in
scan_enable_inv
retention_enable
scan_out
```

The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that it is recommended you add.

```
model retention_scan_cell
 (D, CLK, SI, SSB, SLEEP, Q)
(
model_source = verilog_module;
 cell_type = prohibited; // cannot be used for test point
 input (D) (data_in;)
 input (CLK) (posedge_clock)
 input (SI) (scan_in)
 input (SSB) (scan_enable_inv)
  input (SLEEP) (retention_enable)
 output (Q) (scan_out)
  (
    primitive = _inv mlc_inv_1 (SSB, SSBn;
    primitive = _mux mlc_mux_1 (D, SI, SSBn, Dint);
    primitive = _inv mlc_inv_2 (CLK, CLKn);
    instance = latch_p (Qint, viol_0, CLKn, Dint);
    primitive = _inv mlc_inv_3 (SLEEP, sleepn);
    primitive = _and mlc_and_4 (sleepn, CLK, GCLK);
    instance = latch_p (Q, viol_0, GCLK, Qint);
  )
) // end model retention_scan_cell

//*************** DO NOT ADD WITHIN CELL SELECTION ***************
// Because this scan cell is not to be used as a test point, do not
// add the cell as a posedge_scan_cell in the dft_cell_selection.
dft_cell_selection(dft_cell_selection_name) {
   //Do not declare as: posedge_scan_cell = retention_scan_cell;
}
```

Clock Shaper Cell

This section describes the attributes required for a clock shaper cell.

Model attributes: cell_type = clock_shaper

Pin attributes: enable0 or enable0_inv; enable1 or enable1_inv; active_high_reset or active_low_reset; clock_in; clock_out

simulation_functions: clock_shaper_reset_active or clock_shaper_reset_inactive

Note

When an enable is inverting from the cell enable0 input to the clock_out, its attribute is enable0_inv; otherwise it is enable0. The asynchronous input is required and is arbitrarily called a reset. See the full discussion below.

Definition of Clock Shaper

The clock shaper contains two latches (no DFFs).

The clock_in goes to the clock of both latches, which are mutually exclusive in the sense that when one holds (Q=Q) the other is transparent (Q=D); that is, clock_in is inverting going to one latch clock pin and non-inverting to the other.

Suppose act_lo_lat is the latch that is transparent when the clock is 0, and act_hi_lat is the latch that is transparent when the clock is 1. When the clock is 0, the only cell output, clock_out, is a function of only the act_hi_lat state. When the clock is 1, clock_out is a function of only the act_lo_lat state.

For the clock_shaper with “simulation_function=clock_shaper_reset_active” (get_dft_cell clock_shaper -with_asynch_enable), the cell asynchronous input, when asserted, makes the clock_out equal to the clock_in whether the clock is 1 or 0. Therefore, when the clock_in is 0, if clock_out = act_hi_lat Q, the asynchronous input connects to the act_hi_lat R (Reset) latch input to make Q=0. Otherwise, if clock_out = act_hi_lat QB when clock_in is 0, the asynchronous input connects to the act_hi_lat S (Set) latch input to make QB=0. Similarly, when the clock_in is 1, if clock_out = act_lo_lat Q, the asynchronous input connects to the act_lo_lat S (Set) latch input to make Q=1. Otherwise, if clock_out = act_lo_lat QB when the clock is 1, the asynchronous input connects to the act_lo_lat R (Reset) latch input to make QB=1.

For the clock_shaper with “simulation_function=clock_shaper_reset_inactive” (get_dft_cell clock_shaper -with_asynch_enable), the cell asynchronous input, when asserted, makes the clock_out=0 whether the clock is 1 or 0. Therefore, when the clock_in is 0, if clock_out = act_hi_lat Q, the asynchronous input connects to the act_hi_lat R (Reset) latch input to make Q=0. Otherwise, if clock_out = act_hi_lat QB when clock_in is 0, the asynchronous input connects to the act_hi_lat S (Set) latch input to make QB=0. Similarly, when the clock_in is 1, if clock_out = act_lo_lat Q, the asynchronous input connects to the act_lo_lat R (Reset) latch input to make Q=0. Otherwise, if clock_out = act_lo_lat QB when the clock is 1, the asynchronous input connects to the act_lo_lat S (Set) latch input to make QB=0.

The act_lo_lat D input is a function of only the enable1 cell input. The act_hi_lat D input is a function of only the enable0 input. The polarity is with respect to clock_out, so if the path from enable0 through act_hi_lat to clock_out is inverting, the function attribute is enable0_inv, otherwise, the function attribute is enable0. Similarly, if the path from enable1 through act_lo_lat to clock_out is inverting, the function attribute is enable1_inv; otherwise, the function attribute is enable1.

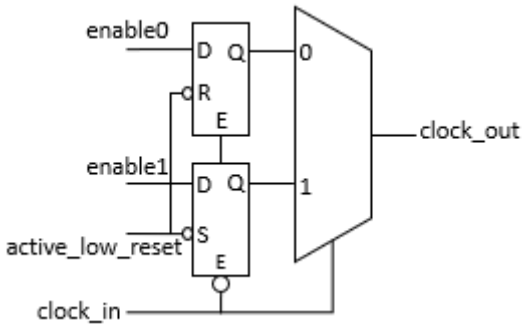
Clock Shaper Cell With Inverting Asynch Enable

This section describes the attributes required for a clock shaper cell with an inverting asynch enable.

Model attributes: cell_type = clock_shaper

Pin attributes: enable0 or enable0_inv; enable1 or enable1_inv; active_high_reset or active_low_reset; clock_in; clock_out

Figure 2-19. Clock Shaper Cell With Inverting Asynch Enable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
model clock_shaper_actlo_reset_active
   (clk_out, en0, en1, resetb, clk_in)
(
  cell_type = clock_shaper;
  simulation_function = clock_shaper_reset_active;

  input (en0) (enable0)
  input (en1) (enable1)
  input (resetb) (active_low_reset)
  input (clk_in) (clock_in)
  output (clk_out) (clock_out)
  (
     primitive = _inv (resetb, reset);
     primitive = _inv (clk_in, clk_inb);
     // Inverted reset goes to active high set of active lo latch.
     //    en1 to its D input.
     primitive = _dlat actlo_lat (reset,  , clk_inb, en1, actlo_q,  );
     // Inverted reset goes to active high reset of active hi latch.
     //    en0 to its D input.
     primitive = _dlat acthi_lat (  , reset, clk_in, en0, acthi_q,  );
     // active hi q goes to in0 of mux, active lo q goes to in1.
     primitive = _mux (acthi_q, actlo_q, clk_in, clk_out);
   )
)
```

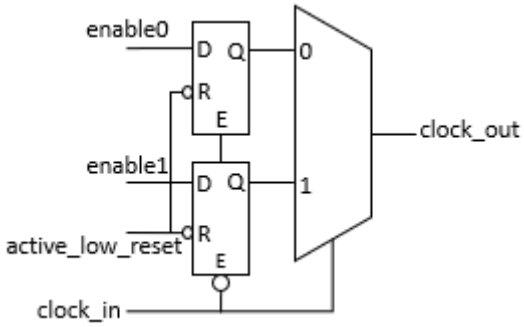
Clock Shaper Cell With Inverting Asynch Disable

This section describes the attributes required for a clock shaper cell with an inverting asynch disable.

Model attributes: cell_type = clock_shaper

Pin attributes: enable0 or enable0_inv; enable1 or enable1_inv; active_high_reset or active_low_reset; clock_in; clock_out

Figure 2-20. Clock Shaper Cell With Inverting Asynch Disable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
model clock_shaper_actlo_reset_inactive
   (clk_out, en0, en1, resetb, clk_in)
(
  cell_type = clock_shaper;
  simulation_function = clock_shaper_reset_inactive;

  input (en0) (enable0)
  input (en1) (enable1)
  input (resetb) (active_low_reset)
  input (clk_in) (clock_in)
  output (clk_out) (clock_out)
  (
     primitive = _inv (resetb, reset);
     primitive = _inv (clk_in, clk_inb);
     // Inverted reset goes to active high reset of active lo latch.
     //    en1 to its D input.
     primitive = _dlat actlo_lat ( , reset, clk_inb, en1, actlo_q,  );
     // Inverted reset goes to active high reset of active hi latch.
     //    en0 to its D input.
     primitive = _dlat acthi_lat ( , reset, clk_in, en0, acthi_q,  );
     // active hi q goes to in0 of mux, active lo q goes to in1.
     primitive = _mux (acthi_q, actlo_q, clk_in, clk_out);
   )
)
```

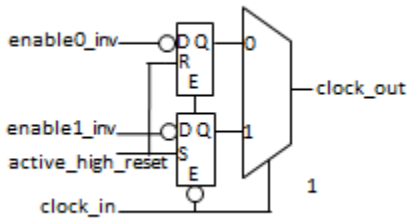
Clock Shaper Cell with Inverting Enables

This section describes the attributes required for a clock shaper cell with inverting enables.

Model attributes: cell_type = clock_shaper

Pin attributes: enable0 or enable0_inv; enable1 or enable1_inv; active_high_reset or active_low_reset; clock_in; clock_out

Figure 2-21. Clock Shaper Cell With Inverting Enables



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
model clock_shaper_actlo_enables
   (clk_out, en0b, en1b, reset, clk_in)
(
  cell_type = clock_shaper;
  simulation_function = clock_shaper_reset_active;

  input (en0b) (enable0_inv)
  input (en1b) (enable1_inv)
  input (reset) (active_high_reset)
  input (clk_in) (clock_in)
  output (clk_out) (clock_out)
  (
     primitive = _inv (clk_in, clk_inb);
     primitive = _inv (en0b, en0);
     primitive = _inv (en1b, en1);
     // reset goes to active high set of active lo latch.
     // inverted enable1 to its D input.
     primitive = _dlat actlo_lat (reset,  , clk_inb, en1, actlo_q,  );
     // reset goes to active high reset of active hi latch.
     //    en0 to its D input.
     primitive = _dlat acthi_lat (  , reset, clk_in, en0, acthi_q,  );
     // active hi q goes to in0 of mux, active lo q goes to in1.
     primitive = _mux (acthi_q, actlo_q, clk_in, clk_out);
   )
)
```

Clock Gating Cell

This section describes the attributes required for a clock_gating_or and clock_gating_and cell.

Model attributes: cell_type = clock_gating_or or cell_type = clock_gating_and

Required Pin attributes: func_enable or func_enable_inv; test_enable or test_enable_inv; clock_in clock_out

Optional Pin attributes: asynch_enable or asynch_enable_inv; asynch_disable or asynch_disable_inv

Additional Output Pins: The clock gating cell clock_out pin attribute can be automatically assigned during clock gating learning if at most two output pins exist on the cell. Otherwise, you must assign the clock_out pin attribute.

Note

When the test_enable or test_enable_inv pin name starts with “te” or “se” and the func_enable or func_enable_inv does not start with “te” or “se”, then the pin function is not required. (The “te” and “se” are case-insensitive).

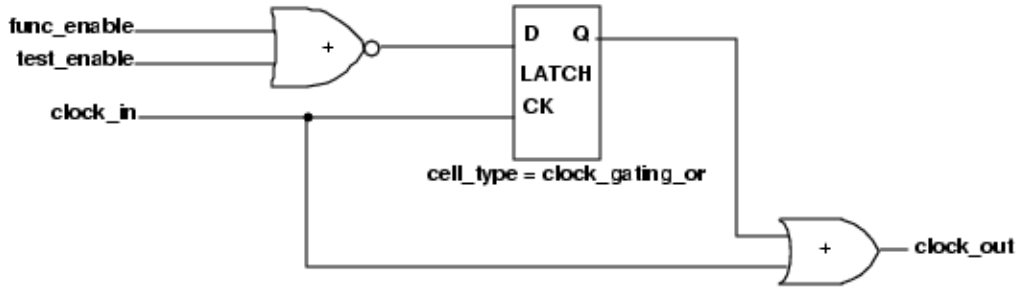
Clock Gating OR Cell With Both Enables

This section describes the attributes required for a clock gating OR.

Model attributes: cell_type = clock_gating_or

Pin attributes: func_enable test_enable clock_inclock_out

Figure 2-22. clock_gating_or With test_enable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

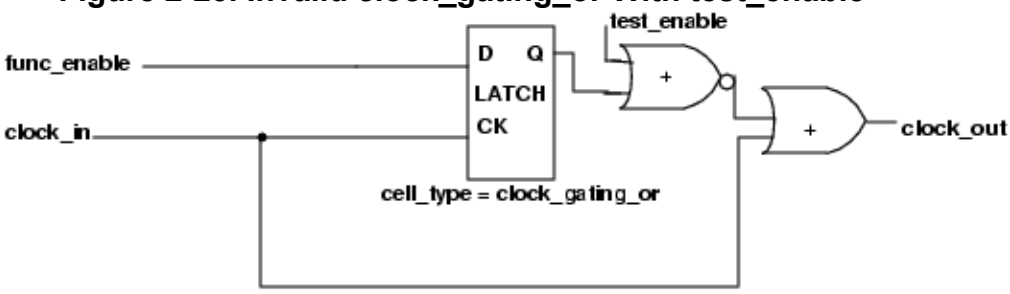
```
model negedge_clk_gater_te
  (clk_out, clk_in, en, test_en)
(
  model_source = verilog_module;
  cell_type = clock_gating_or;

  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 2-23 below is an example of an Invalid Clock Gating OR cell.

Figure 2-23. Invalid clock_gating_or With test_enable



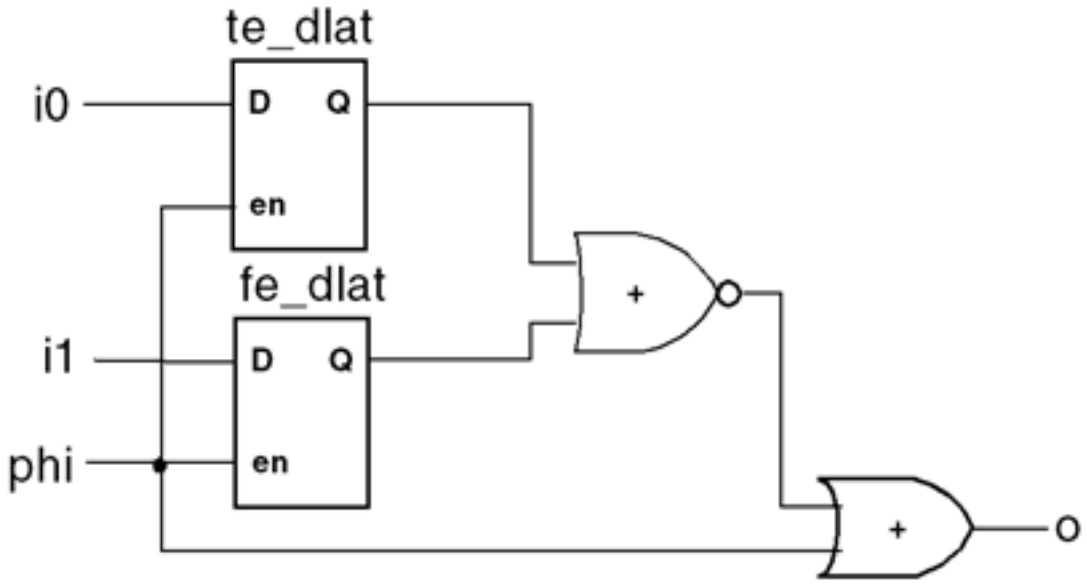
Two-Latch Clock Gating OR Cell With Both Enables

This section describes the attributes required for a two-latch clock gating OR.

Model attributes: cell_type = clock_gating_or

Pin attributes: func_enable test_enable clock_inclock_out

Figure 2-24. Two-Latch clock_gating_or With test_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; the green font indicates the attributes that it is recommended you add.

```
model two_latch_clock_gating_or (o,i0,i1,phi) (
   cell_type = clock_gating_or;
   input (i0) (test_enable)
   input (i1) (func_enable)
   input (phi) (clock_in)
   output (o) (clock_out)
   (
      primitive = _dlat te_dlat (,,phi,i0,q0,);
      primitive = _dlat fe_dlat (,,phi,i1,q1,);
      primitive = _nor lat_q_nor (q0,q1,q);
      primitive = _or out_gat (q,phi,o);
   )
)
```

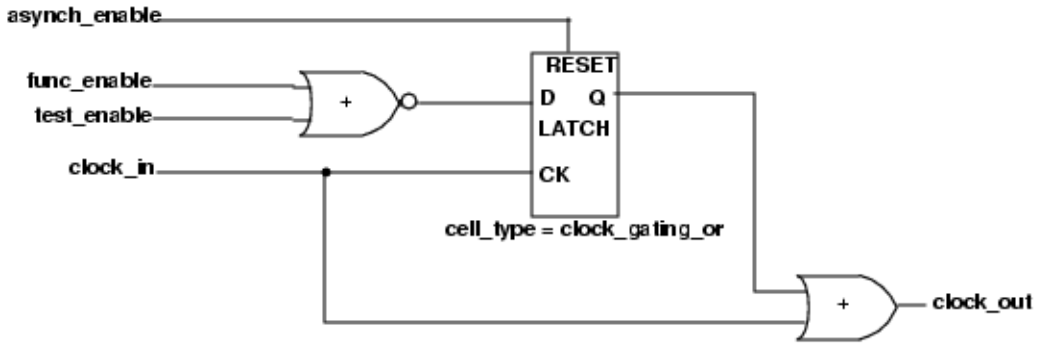
Clock Gating OR Cell With Both Enables and Asynch Enable

This section describes the attributes required for a clock gating OR.

Model attributes: cell_type = clock_gating_or

Pin attributes: func_enable test_enable asynch_enable clock_inclock_out

Figure 2-25. clock_gating_or With test_enable and asynch_enable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
model negedge_clk_gater_te
  (clk_out, clk_in, en, test_en, asynch_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 );
      primitive = _dlat lat_inst ( , asynch_en, clk_in, enabled_b, q_int,
   ); // _dlat has active high reset
      primitive = _or  or_inst ( q_int, clk_in, clk_out );
)
)
```

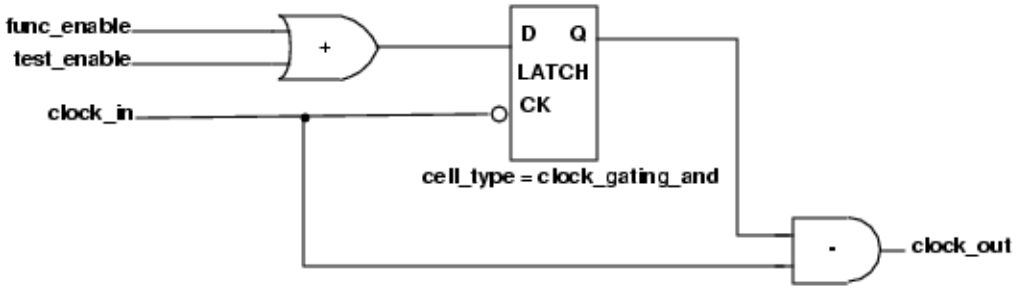
Clock Gating AND Cell With Both Enables

This section describes the attributes required for a clock gating AND.

Model attributes: cell_type = clock_gating_and

Illustrated Pin attributes: func_enable test_enable clock_inclock_out

Figure 2-26. clock_gating_and With test_enable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

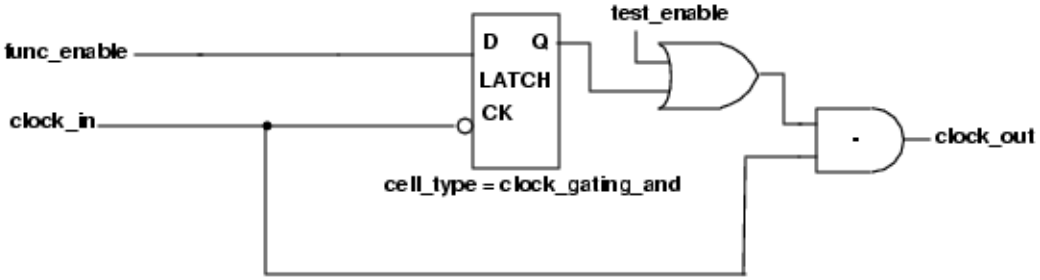
```
model posedge_clk_gater_te
  (clk_out, clk_in, en, test_en)
(
  model_source = verilog_module;
  cell_type = clock_gating_and;

  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 2-27 below is an example of an invalid Clock Gating AND cell.

Figure 2-27. Invalid clock_gating_and With test_enable



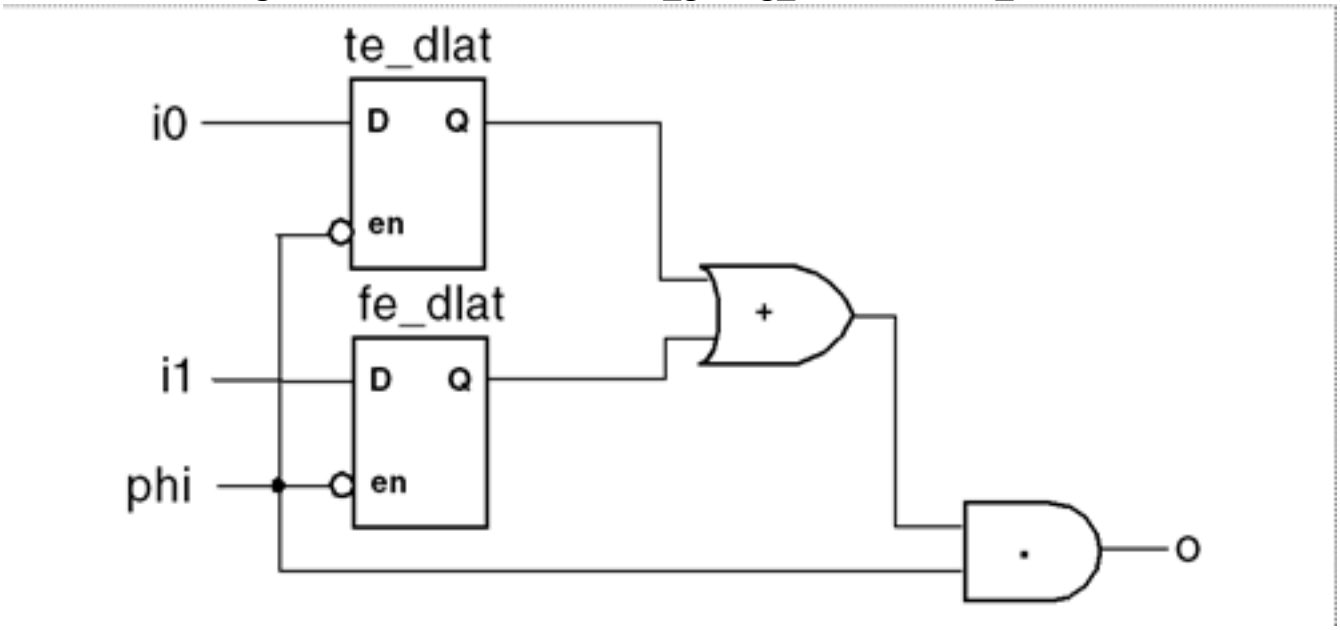
Two-Latch Clock Gating AND Cell With Both Enables

This section describes the attributes required for a two-latch clock gating AND.

Model attributes: cell_type = clock_gating_and

Pin attributes: func_enable test_enable clock_inclock_out

Figure 2-28. Two-Latch clock_gating_and With test_enable



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; the green font indicates the attributes that it is recommended you add.

```
model  two_latch_clock_gating_and (o,i0,i1,phi) (
   cell_type = clock_gating_and;
   input (i0) (test_enable)
   input (i1) (func_enable)
   input (phi) (clock_in)
   output (o) (clock_out)
   (
      primitive = _inv clk_inv (phi,phib);
      primitive = _dlat te_dlat (,,phib,i0,q0,);
      primitive = _dlat fe_dlat (,,phib,i1,q1,);
      primitive = _or lat_q_or (q0,q1,q);
      primitive = _and out_gat (q,phi,o);
   )
)
```

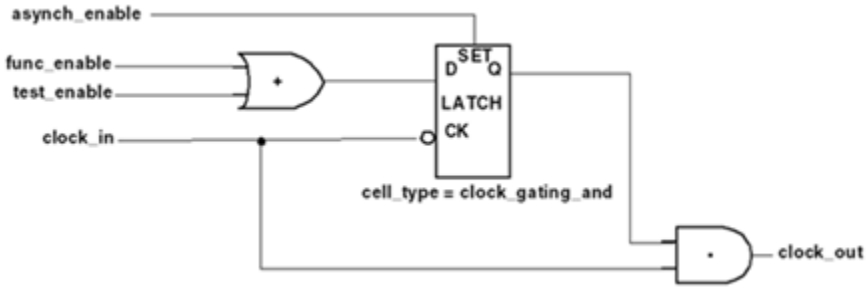
Clock Gating AND Cell With Both Enables and Asynch Enable

This section describes the attributes required for a clock gating AND.

Model attributes: cell_type = clock_gating_and

Illustrated Pin attributes: func_enable test_enable clock_inclock_out

Figure 2-29. clock_gating_and With test_enable and asynch_enable



The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

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

  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 = _or  or_inst ( en, test_en, enabled );
    primitive = _dlat  lat_inst ( asynch_en, ,clk_in, enabled, q_int, );
      //dlat has active_high set
    primitive = _and  and_inst ( q_int, clk_in, clk_out );
  )
)
```

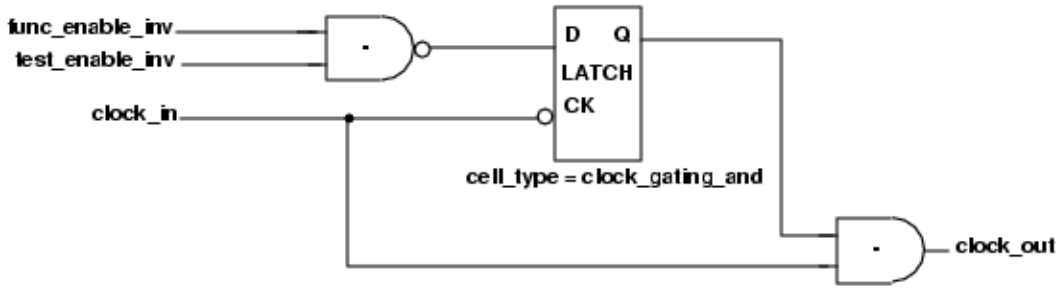
Clock Gating AND Cell With Inverting Enables

This section describes the attributes required for a clock gating AND.

Model attributes: cell_type = clock_gating_and

Illustrated Pin attributes: func_enable_inv test_enable_inv clock_inclock_out

Figure 2-30. clock_gating_and With test_enable



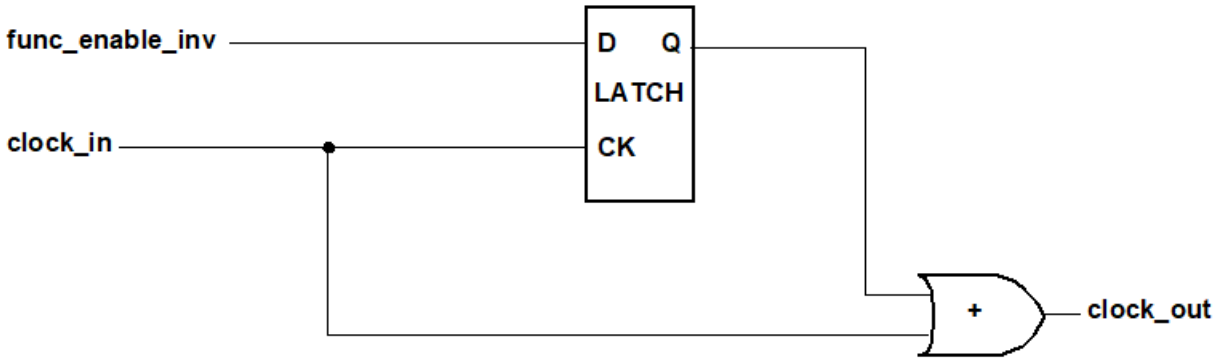
The final Tessent Cell Library is shown here. The green font indicates the attributes that Siemens EDA recommends you add.

```
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 );
  )
)
```

Clock Gating OR Cell With One Enable (Non-Test Only)

This section describes the attributes required for a clock gating OR cell with one enable pin.

Figure 2-31. func_only_clock_gating_or With func_enable_inv



```
model negedge_non_test_clk_gater
  (clk_out, clk_in, en_b)
(
  model_source = verilog_module;
  cell_type = prohibited; // Assigned automatically.
  input (clk_in) (clock_in)
  input (en_b) (func_enable_inv)
  output (clk_out) (clock_out)
  (
    instance = platch_udp  lat_inst ( q_int, clk_in, en_b );
    primitive = _or  or_inst ( q_int, clk_in, clk_out );
  )
)
```

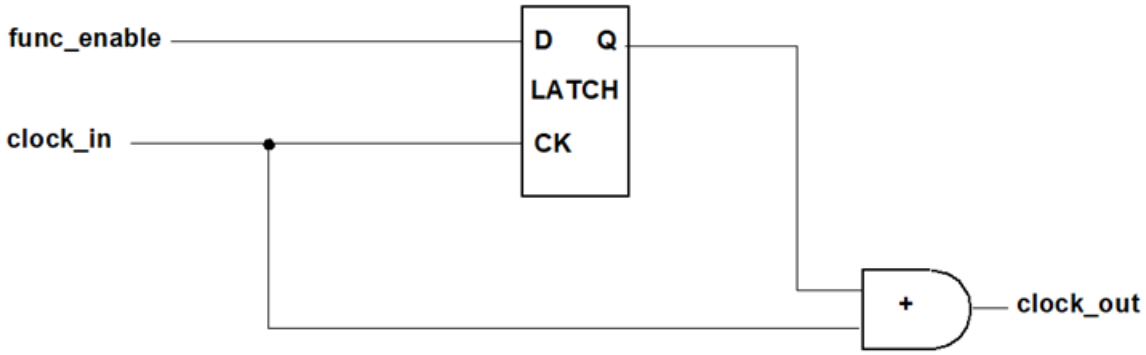
Note

The tool rejects this cell as a cell_type = clock_gating_or, but the tool assigns a cell_type = prohibited because you cannot use this cell for test insertion as a clock gater for test logic. Also, the tool assigns simulation_function = func_only_clock_gating_or.

Clock Gating AND Cell With One Enable (Non-Test Only)

This section describes the attributes required for a clock gating AND cell with one enable pin.

Figure 2-32. func_only_clock_gating_and With func_enable




```
model posedge_non_test_clk_gater
  (clk_out, clk_in, en)
(
  model_source = verilog_module;
  cell_type = prohibited; // Automatically assigned
  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 );
  )
)
```

Note

The tool rejects this cell as a cell_type clock_gating_and, but the tool assigns a cell_type = prohibited because you cannot use this cell for test insertion as a clock gater for test logic. Also, the tool assigns simulation_function = func_only_clock_gating_and.

Synchronizer Cell

This section describes the attributes required for a synchronizer cell.

Model attributes: cell_type = synchronizer_cell; sync_cell_length = 2 or 3, or 4

Required Pin attributes: data_in; posedge_clock or negedge clock; data_out

A synchronizer cell is needed by test insertion only when a clock crosses clock domains in inserted test logic. If a synchronizer cell does not exist and test insertion requires it, one is created from DFF cells. The number of DFFs in a synchronizer cell, called length, must be at least 2 and at most 6.

A cell_type = synchronizer_cell that is appropriate to insert into a netlist for test logic must have a non-inverting system data transfer. So, it must always have one “data_in” and cannot have a “data_in_inv”. If a nonscan synchronizer cell, there must also be a “data_out”. In a scanned synchronizer_cell, there must be either a “data_out”, a “scan_out”, or both. There is no need to indicate a “data_out_inv” pin on a sycnchronizer cell — it is not used and the attribute is removed. If a “data_out” exists, that is used for the system data transfer during test, else a “scan_out” must exist, and that is used for the system data transfer during test. If scanned, either a “scan_out”, “scan_out_inv”, or both must exist, and the stitcher uses one of these to connect to the shift path of the scan chain.

The result of the above is that the following is supported for a nonscan synchronizer_cell:

```
'data_in', 'data_out'
'data_in', 'data_out', additional unattributed output.
```

A scanned synchronizer cell must have a scan_in, scan_enable or scan_enable_inv, and at least one scan_out or scan_out_inv. When it has only one output used for both system and scan, that should have a “scan_out” attribute. It can have both a noninverting data_out, as well as another scan output (“scan_out” or “scan_out_inv”). For the system path, the following combinations are supported for a scanned synchronizer cell, with the “scan_out” being used for system when no “data_out” is specified:

```
'data_in', 'scan_out'
'data_in', 'scan_out', 'scan_out_inv'
'data_in', 'data_out', 'scan_out'
'data_in', 'data_out', 'scan_out_inv'
```

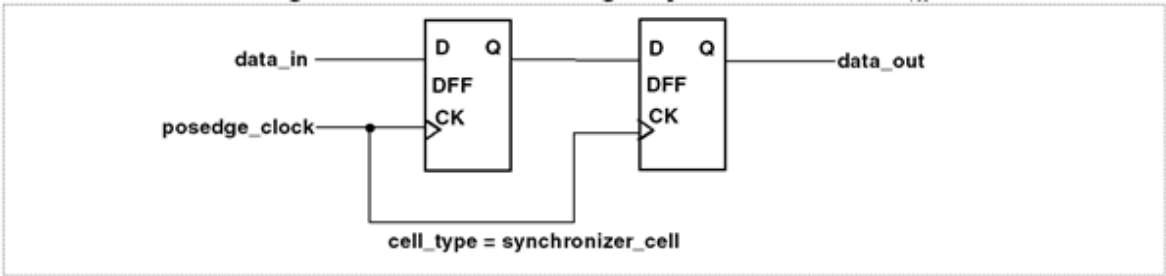
Positive-Edge Synchronizer Cell

This section describes the attributes required for a positive-edge synchronizer cell.

Model attributes: cell_type = synchronizer_cell

Pin attributes: data_in posedge_clock data_out

Figure 2-33. Positive-Edge Synchronizer Cell



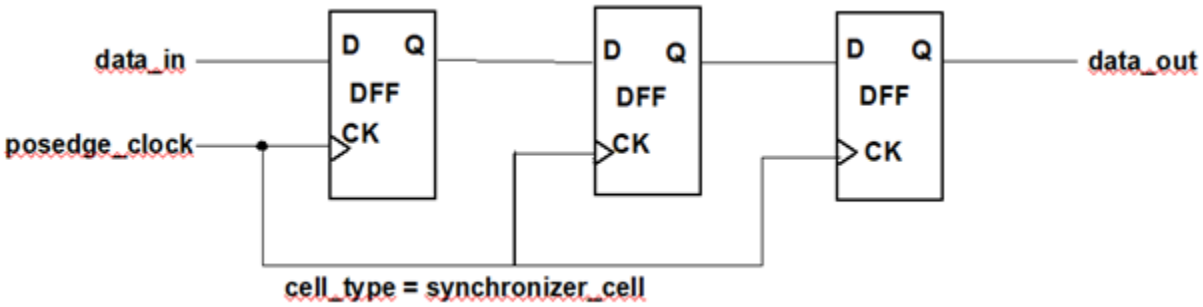
The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
model posedge_synch
  (dout, clk, din)
(
  model_source = verilog_module;
  cell_type = synchronizer_cell;  // Added cell_type
  sync_cell_length = 2;

  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;
}
```

Model attributes: cell_type = synchronizer_cell

Pin attributes: data_in posedge_clock data_out

Figure 2-34. Positive-Edge Synchronizer Cell (3 DFF)



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
model posedge_synch
  (dout, clk, din)
(
  model_source = verilog_module;
  cell_type = synchronizer_cell;  // Added cell_type
  sync_cell_length = 3;

  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_int1, clk, din );
    instance = pos_dff_udp_wrap  middle_dff ( q_int2, clk, q_int1 );
    instance = pos_dff_udp_wrap  back_dff ( dout, clk, q_int2 );
  )
)
//**********  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;
}
```

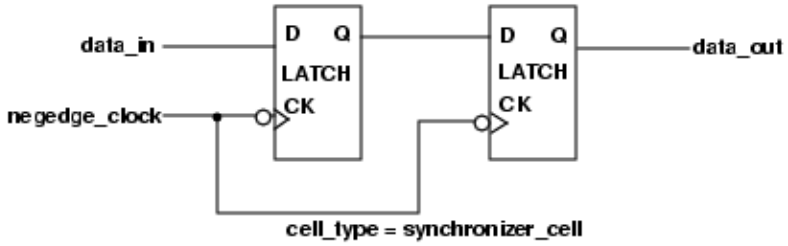
Negative-Edge Synchronizer Cell

This section describes the attributes required for a negative-edge synchronizer cell.

Model attributes: cell_type= synchronizer_cell

Pin attributes: data_in negedge_clock data_out

Figure 2-35. Negative-Edge Synchronizer Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
model negedge_synch
  (dout, clk, din)
(
  model_source = verilog_module;
  cell_type = synchronizer_cell;
  sync_cell_length = 2;

  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;
}
```

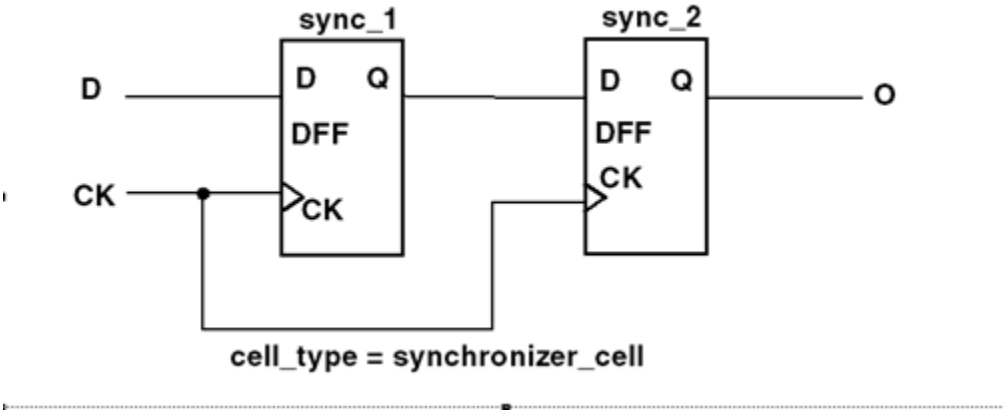
Positive-Edge Non-Scan and Scan Synchronizer Cells

This section describes the attributes required for a positive-edge non-scan synchronizer cell and scan cell replacement.

Model attributes: cell_type = synchronizer_cell

Pin attributes: data_in posedge_clock data_out

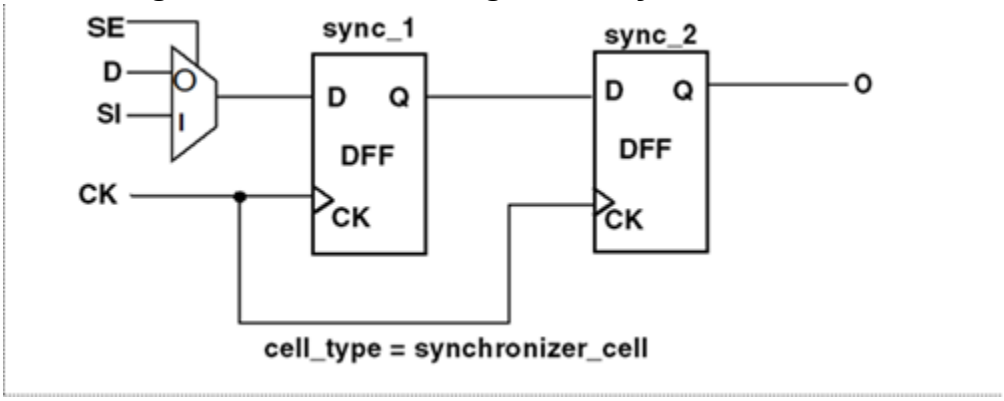
Figure 2-36. Positive-Edge Non Scan Synchronizer Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
model sync_cell
  (CK, D, O)(
  (  model_source = verilog_module;
  cell_type = synchronizer_cell;
  sync_cell_length = 2;
  simulation_function = synchronizer_cell;

  input (CK) ( posedge_clock; )
  input (D) ( data_in; )
  output (O) ( data_out; )
(
    instance = dff sync_1 (D, CK, n2,   );
    instance = dff sync_2 (n2, CK, O,   );
  )
) // end model sync_cell
//**********  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 = sync_cell;
}
```

Figure 2-37. Positive-Edge Scan Synchronizer Cell




```
model scan_sync_cell
  (CK, D, SI, SE,   O)
(
  nonscan_model = sync_cell;
  model_source = verilog_module;
  cell_type = synchronizer_cell;
  sync_cell_length = 2;
  simulation_function = synchronizer_cell;
  scan_length = 2;
  nonscan_model = sync_cell;

  input (CK) ( posedge_clock; )
  input (D) ( data_in; )
  input (SI)( scan_in; )
  input (SE) ( scan_enable; )
  output (O) ( scan_out; )
  (
    primitive = _mux scan_mux (D, SI, SE, n1);
    instance = dff sync_1 (n1, CK, n2,   );
    instance = dff sync_2 (n2, CK, O,   );
  )
) // end model scan_sync_cell
```

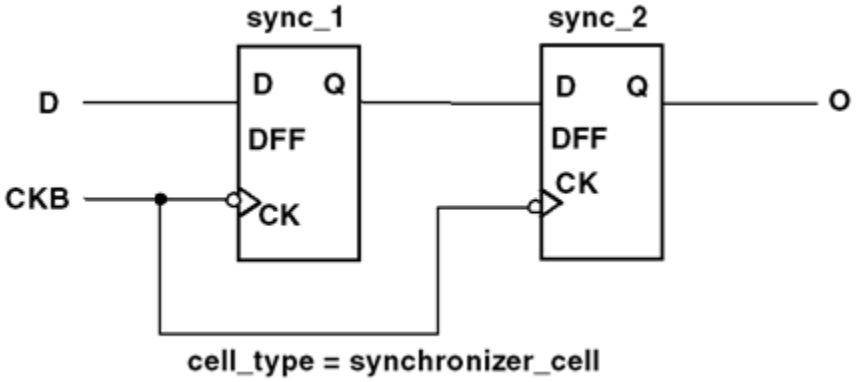
Negative-Edge Non-Scan and Scan Synchronizer Cells

This section describes the attributes required for a negative-edge non-scan synchronizer cell and scan cell replacement.

Model attributes: cell_type = synchronizer_cell

Pin attributes: data_in posedge_clock data_out

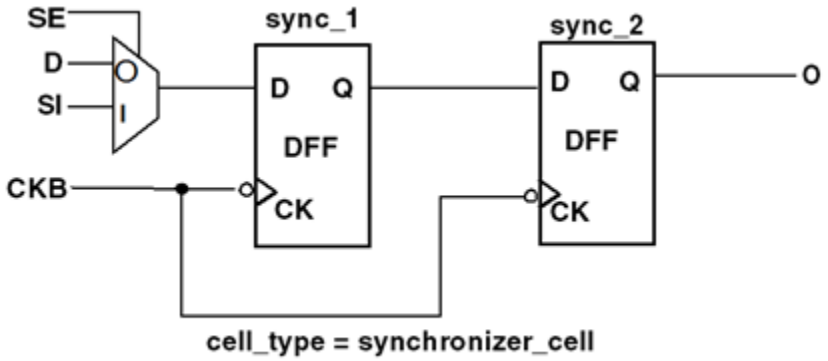
Figure 2-38. Negative-Edge Non Scan Synchronizer Cell



The final Tessent Cell Library is shown here. The red font indicates the attributes you must add; green font indicates the attributes that Siemens EDA recommends you add.

```
model nck_sync_cell  (CKB, D, O)
(
  model_source = verilog_module;
  cell_type = synchronizer_cell;
  sync_cell_length = 2;
  simulation_function = synchronizer_cell;

  input (CKB) ( negedge_clock; )
  input (D) ( data_in; )
  output (O) ( data_out; )
  (
    instance = nck_dff sync_1 (D, CKB, n2,   );
    instance = nck_dff sync_2 (n2, CKB, O,   );
  )
) // end model nck_sync_cell

//**********  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) {
   negedge_synchronizer_cell = nck_sync_cell;
}
```

Figure 2-39. Negative-Edge Scan Synchronizer Cell




```
model nck_scan_sync_cell
  (CKB, D, SI, SE,   O)
(
  nonscan_model = nck_sync_cell;
  model_source = verilog_module;
  cell_type = synchronizer_cell;
  sync_cell_length = 2;
  simulation_function = synchronizer_cell;
  scan_length = 2;
  nonscan_model = nck_sync_cell;

  input (CKB) ( negedge_clock; )
  input (D) ( data_in; )
  input (SI) ( scan_in; )
  input (SE) ( scan_enable; )
  output (O) ( scan_out; )
  (
    primitive = _mux scan_mux (D, SI, SE, n1);
    instance = nck_dff sync_1 (n1, CKB, n2,   );
    instance = nck_dff sync_2 (n2, CKB, O,   );
  )
) // end model nck_scan_sync_cell
```

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).

Also, 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, and Tessent TestKompress. These library models can also be used in the LV Flow.

This chapter includes the following topics:

How to Define 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 that typically applies to all subsequent processing, *model* level information that applies only to a particular model, and *pin* level information that applies only to a particular pin.

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

model model_name (list_of_pins) ( ) // model level
   cell_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:

```
model equivalent_cell = defined_model_name;
regexp_verbosity = {verbose | silent};
cell_attributes()
...
dft_cell_selection (dft_cell_selection_name_1) ()
dft_cell_selection (dft_cell_selection_name_2) ()
...
```

For more information on defining library level information, see “How to Define 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 “How to Define 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 parentheses are required even if the pin or intern has no pin or net attributes to declare.

If there are attributes, they are mentioned within the second set of parentheses 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 below:

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

For information on defining pin level information, see "Pin Attributes."

Note

The *list_of_<some>_attributes* argument is described in section "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 quotation marks.

Legal Block Delimiters

You can use either braces 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 (<dft_cell_selection_name>) ()
   or
dft_cell_selection (<dft_cell_selection_name>){}


cell_attributes ()
   or
cell_attributes {}
```

How to Define 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:

```
model equivalent_cell = defined_model_name;
regexp_verbosity = {verbose | silent};

dft_cell_selection (dft_cell_selection_name_1) ( ) // See Cell Selection
dft_cell_selection (dft_cell_selection_name_2) ( )
...
dft_cell_selection (dft_cell_selection_name_k) ( )

cell_attributes( // See Cell Attributes
   regexp = {anchors | no_anchors | glob | off};
   cell ("modelname_regexp") (
   // model_level_attribute statements,
      // see Model-Level Attribute Descriptions.
   // For legal pin_level attributes, see 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_level_attribute statements,
      // see Model-Level Attribute Descriptions.
   // For legal pin_level attributes, see 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 How to Define Cell Models
   // model_level_attribute statements,
      // see Model-Level Attribute Descriptions.
   // For legal pin_level attributes, see 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 Hardware Definitions
   )
) // end model
...
```

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

- *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 "How to Reuse 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 cell name and pin name matches resulting from regular expression matching.
  - silent — Specifies to not report matching cell names or matching pin names resulting from regular expression matching. This is the default.

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_level_attributes* syntax is described in "Model-Level Attribute Descriptions."
- Pin attributes are described in "Pin Attributes."

Cell Selection

A dft_cell_selection is optional. If not provided, a default dft_cell_selection is created for test insertion use as well as for the get_dft_cell command. The highest priority for selecting a default dft_cell_selection is the safest and simplest cell. If an explicit dft_cell_selection is parsed, no default is created. You can create an empty, explicit dft_cell_selection to prevent creation of a default. The rules for selecting a cell for the default dft_cell_selection are described in the Rules for Selecting the Default dft_cell_selection section below, following the dft_cell_selection syntax.

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 (dft_cell_selection_name)(
   clock_buffer = model_name;
   clock_inverter = model_name;
   clock_mux = model_name_2_data_input, model_name_3_data_input,... ;
   clock_shaper = model_name;
   posedge_clock_dff = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   negedge_clock_dff = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   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_no_asynchs, model_name_asynch_enable,
                             model_name_asynch_disable, model_name_both;
   clock_gating_or = model_name_no_asynchs, model_name_asynch_enable,
                             model_name_asynch_disable, model_name_both;
   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_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   negedge_dff = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   active_high_latch = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   active_low_latch = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   posedge_synchronizer_cell = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   negedge_synchronizer_cell = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   posedge_scan_cell = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   negedge_scan_cell = model_name_no_asynchs, model_name_set,
                                    model_name_reset, model_name_both;
   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:

- **dft_cell_selection_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

dft_cell_selection_name, they are treated as if they are part of the same dft_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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_shaper**

  Specifies the clock_shaper for the specified dft_cell_selection_name of the enclosing wrapper. Must point to a valid clock shaper cell (model). See the cell_type clock_shaper in the Model-Level Attribute Descriptions for details.

- **negedge_clock_dff**

  Specifies the negedge_clock_dff for the specified dft_cell_selection_name of the enclosing wrapper. Must point to a valid clock dff cell (model). See the cell_type clock_dff in the Model-Level Attribute Descriptions for details.

- **posedge_clock_dff**

  Specifies the posedge_clock_dff for the specified dft_cell_selection_name of the enclosing wrapper. Must point to a valid clock dff cell (model). See the cell_type clock_dff in the Model-Level Attribute Descriptions for details.

- **clock_and**

  Specifies the clock_and for the specified dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_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 dft_cell_selection_name of the enclosing wrapper. Must point to a valid mux scan_cell (model) that is appropriate to use for a test point. 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 dft_cell_selection_name of the enclosing wrapper. Must point to a valid mux scan_cell (model) that is appropriate to use for a test point. See the cell_type scan_cell in the Model-Level Attribute Descriptions for details.

- **default_bcell_libs**

  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 in the *Embedded Test Hardware Reference*.

For more information, see Appendix A, Attributes and the Tessent Cell Library.

Rules for Selecting the Default dft_cell_selection

The highest priority cell that exists is selected for each default dft_cell_selection. In the examples below, the priorities are listed from P1, the highest priority to P32, the lowest priority. The selection criteria are on the left, in order of importance.

- First, a cell with a user assigned cell type is always selected if one exists, over a cell with no user-assigned cell type.
- Second, a cell referenced by a read_verilog instance in a module outside a `celldefine is always selected if one exists over a cell not so referenced.
- Next, a cell is selected if it has unused inputs, asynch inputs, or tied inputs, with tied inputs being the lowest priority of these, asynchs next, followed by only additional unused input, and finally no additional inputs of any kind.
- The last criterion is whether the transfer (as indicated by the output pin function) is inverting or non-inverting, with the latter being higher priority.

To illustrate, P5 has a user assigned cell type, is also referenced by a read_verilog file as an instance within a module outside a `celldefine, has one or more asynch inputs but no tie inputs, and has a non-inverting transfer function (has a ‘scan_out’ rather than only a ‘scan_out_inv’ for example). P32 has no user assigned cell type, is not referenced by a Verilog instance outside a `celldefine, has at least one tied input, and has only an inverting transfer (for example, only a ‘scan_out_inv’ output for a scan_cell, say, or only a ‘data_out_inv’ output for a dff or latch).

User assigned cell type:

No {unused, asynch, tie} inputs.

P1. Non-inverting transfer function (for example, cell with a ‘data_out’).

P2. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Unused inputs, but no {asynch, tie}.

P3. Non-inverting transfer function (for example, cell with a ‘data_out’).

P4. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Asynch inputs, but no {tie}.

P5. Non-inverting transfer function (for example, cell with a ‘data_out’).

P6. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Tie inputs.

P7. Non-inverting transfer function (for example, cell with a ‘data_out’).

P8. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Not referenced by read_verilog instance outside `celldefine.

No {unused, asynch, tie} inputs.

P9. Non-inverting transfer function (for example, cell with a ‘data_out’).

P10. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Unused inputs, but no {asynch, tie}.

P11. Non-inverting transfer function (for example, cell with a ‘data_out’).

P12. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Asynch inputs, but no {tie}.

P13. Non-inverting transfer function (for example, cell with a ‘data_out’).

P14. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Tie inputs.

P15. Non-inverting transfer function (for example, cell with a ‘data_out’).

P16. Inverting transfer function (for example, cell with only ‘data_out_inv’).

No user assigned cell_type.

Referenced by read_verilog instance outside `celldefine.

No {unused, asynch, tie} inputs.

P17. Non-inverting transfer function (for example, cell with a ‘data_out’).

P18. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Unused inputs, but no {asynch, tie}.

P19. Non-inverting transfer function (for example, cell with a ‘data_out’).

P20. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Asynch inputs, but no {tie}.

P21. Non-inverting transfer function (for example, cell with a ‘data_out’).

P22. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Tie inputs.

P23. Non-inverting transfer function (for example, cell with a ‘data_out’).

P24. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Not referenced by read_verilog instance outside `celldefine.

No {unused, asynch, tie} inputs.

P25. Non-inverting transfer function (for example, cell with a ‘data_out’).

P26. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Unused inputs, but no {asynch, tie}.

P27. Non-inverting transfer function (for example, cell with a ‘data_out’).

P28. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Asynch inputs, but no {tie}.

P29. Non-inverting transfer function (for example, cell with a ‘data_out’).

P30. Inverting transfer function (for example, cell with only ‘data_out_inv’).

Tie inputs.

P31. Non-inverting transfer function (for example, cell with a ‘data_out’).

P32. Inverting transfer function (for example, cell with only ‘data_out_inv’).

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_level_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_level_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 "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 cell libraries using simple 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 braces 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 regexp is set to anchors, they do not match.
- glob — This value converts “*” to “.*” and “?” to “.” only. (This is not true globbing.) This value enables back reference support with globbing wildcards, because regexec 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 enables 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 (regexp off) model that specified the attribute in a *cell_attributes* section determines the value. Else, if only regexp models (that must be in a *cell_attributes* section) specify the value, the last regexp to specify the value determines the value assigned.

- **cell| model**

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

```
cell (“modelname_regexp”) (
   input (“pinname_regexp”) (list_of_input_attributes)
   output (“pinname_regexp”) (list_of_output_attributes)
   inout (“pinname_regexp”) (list_of_inout_attributes)
) // end cell
```

Note

The *list_of_<some>_attributes* argument is described in “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 Method 2 in "Pin Attributes" on page 138.

How to Associate read_cell_library Attributes With read_verilog Modules

There are two syntactic ways you can associate cell and port attributes with read_cell modules.

In both cases, when associating information with a read_verilog "module fred" that is inside some file read via the read_verilog command, **there is no "model fred" hardware description (outside the cell_attributes { }) read in any file via the read_cell_library command**. However, **the module and port attributes** to be associated with the Verilog module "fred" must be read in via the read_cell_library command, and the association occurs because "fred" is referenced as the cell name in the read_cell_library file with the attributes, and no read_cell_library model "fred" had its internal hardware defined, that is, no "model fred" was defined outside a "cell_attributes { }".

1. Read attributes from one or all of the LV formatted files cell.lib, scang.lib, and pad.lib.
2. Read attributes from the cell_attributes{ } section of the tessent cell library syntax.

This syntax is outlined in the section How to Define a Cell Library.

This section shows how a regular expression can be used to associate one set of attributes with many cells (those whose module name matches the regular expression), and also how (with regexp = off) to associate one set of attributes with a single cell.

For example, an LV "Cell (BMX13A)" in a pad.lib would have its attributes associated with a read_verilog "module BMX13A". Alternatively, with regexp = off, a "cell (BMX13A)" inside a Tessent cell library file's "cell_attributes { }" would have its attributes associated with a read_verilog "module BMX13A". Finally, using regular expressions, a "cell ("^BMX.*$")" inside a Tessent cell library file's "cell_attributes { }" would have its attributes associated with any read_verilog module that starts with "BMX". The anchors "^" and "$" surrounding the module name characters ensure only modules that start with "BMX" (rather than also those that have those characters in the middle of the name) match the regular expression and are attributed. The ".*" following "BMX" means zero or more following characters regardless of what they may be. See man regexp for a complete definition of regular expressions. For those unfamiliar with regular expressions, but who want to use "*" wildcards, the "regexp = glob;" statement inside the "cell_attributes { }" enables that form of matching.

Most attributes, regardless of whether LV or Tessent format, are available via TCL or get_dft_cell. Pad mode (Usage in LV format) information, and nonscan mapping information are exceptions. However, the Pad mode/Usage information is available and can be used by pad extraction, and the nonscan equivalent information is available and can be used for test insertion for scan cell replacement of nonscan equivalents in the netlist.

Information from the cell.lib or dft_cell_selection is available via TCL or the Tessent Shell get_dft_cell command.

To illustrate, assume that a read_verilog file had a module “CKD46” as follows :

```
module CKD46 ( A , Y );
input   A ;
output  Y ;
  buf  bu1 (Y, A);
endmodule
```

Further, a “read_cell_library cell.lib” was issued where the LV format inside cell.lib contained the following:

```
 cellLibrary(tgc4000) {
  CellsToUseOnFunctionalClockPaths {
    ClockBuffer ( CKD46 ) {
      port ( A ): Input;
      port ( Y ): Output;
    }
```

Then get_dft_cell (where Tessent syntax must be used) could be used to get the clock_buffer (ClockBuffer in LV syntax) as shown by the following transcript showing the command and its output :

```
//  command: puts "[get_name [get_dft_cell clock_buffer ] ]"
 CKD46
```

To illustrate regexp attribution, assume that a read_cell_library file contained the following:

```
regexp_verbosity = silent;
// Prevents warning messages about ABC40
//... cells with no 'oeb' or 'dout' ports
cell_attributes {
   Cell ("^ABC40.*$") {
      input (oeb) (pad_enable_low)
      input (dout) (pad_to_pad)
   }
}
```

Further, assume that a read_verilog file contained a Verilog definition of “module ABC40LP40_35”.

Then, the following Tessent transcript illustrates the pad_function attributes thereby associated with the ports.

```
// command: foreach_in_collection i [get_ports -of_module  ABC40LP40_35] {
  puts  "port name: [get_name $i]  \
  pad_function value: [get_attr_val $i -name pad_function]"
}
port name: dout  pad_function value: to_pad
port name: oeb  pad_function value: enable_low
```

How to Define 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 |
    clock_dff | clock_shaper | dff | latch | synchronizer_cell |
    scan_cell | pad | prohibited };
  simulation_function = { clock_gating_and | clock_gating_or |
    func_only_clock_gating_and | func_only_clock_gating_or |
    clock_shaper_reset_active | clock_shaper_reset_inactive | 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);
  nonscan_equivalents = model_name1, modelname2 ...;
  scan_equivalents = model_name1, modelname2 ...;
  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_two_state_output_enable_high]
      [pad_two_state_output_enable_low]
      [pad_init_data_dot6] [pad_init_data_inv_dot6]
      [pad_init_posedge_clock_dot6] [pad_init_negedge_clock_dot6]
      [pad_init_enable_high_dot6] [pad_init_enable_low_dot6]
      [pad_ac_mode_dot6]
      [pad_data_inv]       [pad_diff_voltage] [pad_diff_current]
      [pad_from_io] [pad_from_io_inv] [pad_tied0] [pad_tied1]
      [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 cell library
// primitives and instances
  (
    //hardware_statements, see Hardware Definitions
  )

) // 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 (...
...
)
```

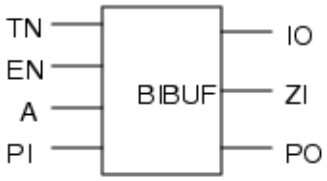
- **model_name** Specifies the name of the library cell in your design data. When translating from Verilog, the model_name should match the Verilog module name, or UDP primitive name. The *model_name* argument enables you to reference the cell.
- **list_of_pins** Specifies the interface pins on the cell boundary that 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

Model-level attributes define characteristics of individual models.

This section describes the following model-level attributes:

input(port_name_list) (input_port_attributes)

The port_name_list that 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 "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 "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 "Pin Attributes."

sync_cell_length

Keyword that only applies to cell_type = synchronizer_cell. It must be 2, 3, or 4 to reflect the number of cycles (latency) from the data_in capture to the data_out launch.

cell_type

Keyword that identifies the cell (model) function. The value "pad" is informative. All other legal values specify the cell transfer function and typically also require use of pin functions when inserting test logic circuitry 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 |
clock_shaper | dff | clock_dff | latch | synchronizer_cell | scan_cell |
pad | prohibited};
```

clock_gating_and

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 two enable input pins, the first with either func_enable attribute (clocking enabled when input is 1 or high) or

func_enable_inv attribute (enabled when 0/low), and the second with either test_enable attribute (clocking enabled when input is 1 or high) or test_enable_inv attribute (enabled when 0/low). The cell should have a single output with clock_out attribute. Optionally, you can also immediately enable the cell by an additional asynch_enable input (clocking enabled when input is 1 or high) or asynch_enable_inv (enabled when 0 or low). You can also optionally immediately disable the cell 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, refer to *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 the tool learns the cell_type and sets it automatically. It also learns 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).

clock_gating_or

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 (clocking 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, you can also enable the cell by an additional test_enable input (clocking enabled when input is 1 or high) or test_enable_inv (enabled when 0 or low). You can also immediately enable the cell by an additional asynch_enable input (clocking enabled when input is 1 or high) or asynch_enable_inv (enabled when 0 or low). You can also optionally immediately disable the cell 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, refer to *Application Note Timing Constraints andClock 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).

buffer

Specifies a buffer to be inserted on data paths. For more information, refer to Add Cell Models in the *Tessent Shell Reference Manual*.

clock_buffer

Specifies a buffer to be inserted on clock paths. For more information, refer to Add Cell Models in the *Tessent Shell Reference Manual*.

inverter

Specifies an inverter cell to be inserted on data paths. For more information, refer to the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

clock_inverter

Specifies an inverter cell to be inserted on clock paths.

and

Specifies an And cell to be inserted on data paths. For more information, refer to the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

clock_and

Specifies an And cell to be inserted on clock paths.

nand

Specifies a Nand cell to be inserted on data paths. For more information, refer to NAND on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

or

Specifies an Or cell to be inserted on data paths. For more information, refer to OR in the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

clock_or

Specifies an Or cell to be inserted on clock paths.

nor

Specifies a Nor cell to be inserted on data paths. For more information, refer to NOR on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

xor

Specifies an Xor cell to be inserted on data paths. For more information, refer to XOR on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

mux

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 MUX on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

clock_mux

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.

clock_shaper

A cell that enables an incoming clock, clock_in, to be held low for any number of cycles on the cell output, clock_out, or to be held high for any number of cycles. The clock_out is synchronized with the clock_in so that edges on clock_out can only occur on an edge of clock_in (for simulation_function=clock_shaper_reset_active) or is set to 0 (for simulation_function=clock_shaper_reset_inactive).

dff

Used by 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 or the Add Control Points command. For more information, refer to DFF on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

clock_dff

Used by 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 a negedge_clock or posedge_clock attribute, and an input pin with a data_in attribute. The cell should have a single output with a data_out attribute.

latch

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, refer to LATCH on the Add Cell Models reference page in the *Tessent Shell Reference Manual*.

synchronizer_cell

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. The number of DFFs in a synchronizer cell, called length, must be at least 2 and at most 6.

scan_cell

Used by Tessent Shell flows to indicate a mux scan cell that is appropriate to use for a test point. 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. It must have a single data_in input and all scan_out or scan_out_inv outputs must be a function of that single input when scan enable is deasserted (in system operation).

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 |
func_only_clock_gating_and | func_only_clock_gating_or |
clock_shaper_reset_active | clock_shaper_reset_inactive |
buffer | inverter | or | and | nandn | nor | xor | mux |
dff | latch | synchronizer_cell | scan_cell | tie0 | tie1};
```

func_only_clock_gating_and

Specifies a latched-enabled clock gating cell that forces the clock low when disabled. The referenced cell (model) must have an input pin with a clock_in attribute, and a single enable pin with a func_enable (enables when that port is 1) or func_enable_inv (enables when that port is 0) attribute. The cell should have a single output with clock_out attribute.

func_only_clock_gating_or

Specifies a latched-enabled clock gating cell that forces the clock high when disabled. The referenced cell (model) must have an input pin with a clock_in attribute, and a single enable pin

with a func_enable (enables when that port is 1) or func_enable_inv (enables when that port is 0) attribute. The cell should have a single output with clock_out attribute.

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 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 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 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 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

The nonscan_model statement is only necessary if the nonscan pin name differs from the corresponding pin name on the scan cell it is to be associated with. If that is not needed, consider using the nonscan_equivalents and scan_equivalents attributes instead. The nonscan_model is a 2-way association, associating the nonscan => scan model replacement, as well as the scan => nonscan replacement (such as can occur in a shift register of scan cells, where all but the first mux scan cell can be replaced by nonscan and the system shift path used for scan as well). Often, the best nonscan => scan replacement differs from the best scan => nonscan replacement, and only the scan_equivalents plus nonscan_equivalents inside the nonscan and scan models respectively cause the best replacement in all cases.

nonscan_model is an optional attribute placed inside a scan cell model 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:

```
nonscan_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 *nonscan_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)
      ...
      nonscan_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);
```

nonscan_equivalents

Optional attribute placed inside a scan cell model that specifies the nonscan cell model names that can replace the scan cell. The syntax is:

```
nonscan_equivalents = nonscan_model_name1, nonscan_model_name2, ....;
```

The first nonscan model name is used to replace the scan cell inside which this statement occurs, if it exists in the Tessent cell library. All nonscan model pin names must also exist on a scan model that can be replaced by the nonscan model. For more information on how to automatically populate the nonscan_equivalents, see “Create Tessent Insertion Attributes Using Liberty”, and “Method 2” in “Primitive and Attribute Examples” for regular expression usage.

scan_equivalents

Optional attribute placed inside a nonscan cell model that specifies the scan cell model names that can replace the nonscan cell. The syntax is:

```
scan_equivalents = scan_model_name1, scan_model_name2, ....;
```

The first scan model name is used to replace the nonscan cell inside which this statement occurs, if it exists in the Tessent cell library. All nonscan model pin names must also exist on a scan model that replaces the nonscan model. For more information on how to automatically populate the scan_equivalents, see "Create Tessent Insertion Attributes Using Liberty", and "Method 2" in "Primitive and Attribute Examples" for regular expression usage.

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-bitscan 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 does not cause DRC tracing rule violations.

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_two_state_output_enable_high]
[pad_two_state_output_enable_low]
[pad_init_data_dot6] [pad_init_data_inv_dot6]
[pad_init_posedge_clock_dot6]  [pad_init_negedge_clock_dot6]
[pad_init_enable_high_dot6]  [pad_init_enable_low_dot6]
[pad_ac_mode_dot6]
[pad_data_inv] [pad_diff_voltage] [pad_diff_current]
[pad_from_io] [pad_from_io_inv] [pad_tied0] [pad_tied1] [pad_open]
) // end input section of mode
```

  For information on specific input attributes statements, see “Pin Attributes” on page 138.

- **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 “Pin Attributes” on page 138.

- **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 inout section of mode
```

  For information on specific inout attributes statements, see “Pin Attributes” on page 138.

- **mode directionless pin attributes**

```
pin(pin_name)(// Following are legal mode directionless pin
attributes
any attribute in the above mode pins
) //end directionless pin section of mode
```

For information on specific pad pin attributes statements, see “Pin Attributes” on page 138

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 Questa™ SIM 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 enable 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 Questa SIM 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;
```

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
```

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;
```

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

```
model_source = lv_generic;
```

power_retention_cell

Used during retention test generation to declare a library model as a retention cell. During ATPG, this cell should be treated as a retention cell.

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 “Pin Attributes.” The hardware section is shown here for completeness, and is discussed in “Hardware Definitions.”

```
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 a 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; a space is required at the end to designate the end of the name). In the cell 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 quotation marks.

For example, given the simple Verilog netlist:

```
module test ( i1, o1);
   input i1;
   output o1;
   inv1b u1( .\a# (i1), .o (o1) );
endmodule
```

With the Verilog cell library model shown here:

```
module inv1b ( \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 inv1b (\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 inv1b ("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 that indicates that a defining pin name follows is outside the double quotation marks 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 144.

Hardware Definitions

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

Table 3-1 lists the hardware definition that can be applied to models and their pins. Each attribute in the table is described in detail following the table.

Table 3-1. Hardware

Attribute Statement	Used for Pin Types	Description
primitive	Intern, Input, Output, Inout	See "primitive".
instance	Intern, Input, Output, Inout	See "instance".
bus_keeper	Intern, Output, Inout	See "bus_keeper".
function	Intern, Output, Inout	See "function".

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 192 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 cell 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 *TessentScan 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);
   )
)
```

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 enable you to further define the internal structure of the model precisely, as shown below:

```
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)
```

Note

Inside a single mode (LV usage) a given pad_attribute can only occur on a single port, with the exception of pad_keep_tracing. The pad_keep_tracing exception is allowed because pad_keep_tracing does not specify the port's pad_function, but is used when a pad is composed of multiple models to indicate that a pad_function should be found on another model connected to the port with a keep_tracing attribute. Because more than one pad_function can be deferred to another model, multiple ports can have the pad_keep_tracing attribute.

Table 3-2 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 the section "Attribute Descriptions" that follows this table.

Table 3-2. 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".
assert_capture	Input	See "assert_capture = assert_value.

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
assert_pattern	Input	See "assert_pattern = assert_value.
assert_shift	Input	See "assert_shift = assert_value".
asynch_disable	Input	See "asynch_disable".
asynch_disable_inv	Input	See "asynch_disable_inv".
asynch_enable	Input	See "asynch_enable.
asynch_enable_inv	Input	See "asynch_enable_inv".
clock_in	Input	See "clock_in".
clock_out	Output	See "clock_out".
data_in	Input	See "data_in".
data_in_inv	Input	See "data_in_inv".
data_out	Output	See "data_out".
data_out_inv	Output	See "data_out_inv".
func_enable	Input	See "func_enable".
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_select0".
mux_select0	Input	See "mux_select0".
mux_select1	Input	See "mux_select1".
mux_select2	Input	See "mux_select2".
negedge_clock	Input	See "negedge_clock".

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
posedge_clock	Input	See "posedge_clock".
power_isolation_external	Input, Output	See "power_isolation_external".
power_isolation_internal	Input, Output	See "power_isolation_internal".
retention_enable	Input	See "retention_enable"
retention_enable_inv	Input	See "retention_enable_inv"
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".
scan_out_no_polarity	Output	See "scan_out_no_polarity".
test_enable	Input	See "test_enable".
test_enable_inv	Input	See "test_enable_inv".
tie0	Input	See "tie0".
tie1	Input	See "tie1".
Pad Function (See Note above)		
pad_ac_mode_dot6	Input	See "pad_ac_mode_dot6".
pad_data_inv	Input, Output	See "pad_data_inv".
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".

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
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 *Note: Deprecated.	Input, Inout	See "pad_init_clock_dot6".
pad_init_data_dot6	Input, Inout	See "pad_init_data_dot6".
pad_init_data_inv_dot6	Input, Inout	See "pad_init_data_inv_dot6".
pad_init_enable_high_dot6	Input, Inout	See "pad_init_enable_high_dot6".
pad_init_enable_low_dot6	Input, Inout	See "pad_init_enable_low_dot6".
pad_init_negedge_clock_dot6	Input, Inout	See "pad_init_negedge_clock_dot6".
pad_init_posedge_clock_dot6	Input, Inout	See "pad_init_posedge_clock_dot6".
pad_input_enable_high	Input	See "pad_input_enable_high".
pad_input_enable_low	Input	See "pad_input_enable_low".
pad_keep_tracing	Input, Output	See "pad_keep_tracing".

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
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	Input, Inout	See "pad_test_data_dot6".
pad_test_data_inv_dot6	Input, Inout	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".
pad_two_state_output_enable_high	Input	See "pad_two_state_output_enable_high"
pad_two_state_output_enable_low	Input	See "pad_two_state_output_enable_low"
Special Drive		
pad_open_drain	Output, Inout	See "pad_open_drain".

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
pad_open_source	Output, Inout	See "pad_open_source".
Pull Drive		
pad_pull0	Input, Output, Inout	See "pad_pull0".
pad_pull1	Input, Output, Inout	See "pad_pull1".
Undriven Input State		
pad_input_extern0	Output	See "pad_input_extern0".
pad_input_extern1	Output	See "pad_input_extern1".
pad_input_keeper	Output	See "pad_input_keeper".
pad_input_open0	Output	See "pad_input_open0".
pad_input_open1	Output	See "pad_input_open1".
pad_input_openx	Output	See "pad_input_openx".
pad_input_pull0	Output	See "pad_input_pull0".
pad_input_pull1	Output	See "pad_input_pull1".
Attribute		
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".

Table 3-2. Pin Attributes (cont.)

Attribute	For Pin Direction	Description
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 3-2 on page 138 to determine the pin type the attribute can be attached to.

1. **active_high_clock**

   Level clock and polarity.

2. **active_high_reset**

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

3. **active_high_set**

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

4. **active_low_clock**

   Level clock and polarity.

5. **active_low_reset**

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

6. **active_low_set**

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

7. **assert_shift = assert_value**

   Specifies that the input pin should be at the assert_value throughout shift. DRC checks are done on all netlist instances defined by a model with one or more pins with assert_shift declared.

   You can introspect scalar port values of assert_shift assert_value. You can specify any of the following for assert_value:

   - A simple scalar constant from {0, 1, Z} or their one bit equivalent {1’b0, 1’b1, 1’bZ}.

- A vector constant composed of any of the above plus X or x to mean no assertion for the corresponding bit of the vector port.
- "stable" – can initially be any value from {0,1} for Boolean inputs, {0,1,Z} for rare Z-consuming inputs such as nmos/pmos/cmos channels, but then must remain fixed at that value.

8. **assert_capture = assert_value**

   Specifies that the input pin should be at the assert_value throughout capture. DRC checks are done on all netlist instances defined by a model with one or more pins with assert_capture declared.

   You can introspect scalar port values of assert_capture assert_value. You can specify any of the following for assert_value:

   - A simple scalar constant from {0, 1, Z} or their one bit equivalent {1'b0, 1'b1, 1'bZ}.
   - A vector constant composed of any of the above plus X or x to mean no assertion for the corresponding bit of the vector port.
   - "stable" – can initially be any value from {0,1} for Boolean inputs, {0,1,Z} for rare Z-consuming inputs such as nmos/pmos/cmos channels, but then must remain fixed at that value.

9. **assert_pattern = assert_value**

   Specifies that the input pin should be at the assert_value throughout the entire pattern. DRC checks are done on all netlist instances defined by a model with one or more pins with assert_pattern declared.

   You can introspect scalar port values of assert_pattern assert_value. You can specify any of the following for assert_value:

   - A simple scalar constant from {0, 1, Z} or their one bit equivalent {1'b0, 1'b1, 1'bZ}.
   - A vector constant composed of any of the above plus X or x to mean no assertion for the corresponding bit of the vector port.
   - "stable" – can initially be any value from {0,1} for Boolean inputs, {0,1,Z} for rare Z-consuming inputs such as nmos/pmos/cmos channels, but then must remain fixed at that value.

10. **asynch_enable**

    Optional attribute used for ClockGating cells to define the asynchronous or immediate clocking enable when the so attributed port has a logic 1.

11. **asynch_enable_inv**

Optional attribute used for ClockGating cells to define the asynchronous or immediate clocking enable when the so attributed port has a logic 0.

12. **asynch_disable**

    Optional attribute used for ClockGating cells to define the asynchronous or immediate clocking disable when the so attributed port has a logic 1.

13. **asynch_disable_inv**

    Optional attribute used for ClockGating cells to define the asynchronous or immediate clocking disable when the so attributed port has a logic 0.

14. **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 connect the clock to the clock_in pin and the enable to the data_in pin.

15. **clock_out**

    Used to indicate gated clock output of ClockGating cells.

16. **data_in**

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

17. **data_in_inv**

    Optional attribute that specifies an inverted data input pin of cell types “or” and “and”.

18. **data_out**

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

19. **data_out_inv**

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

20. **func_enable**

    Optional attribute used for ClockGating cells to define the functional or system enable.

21. **func_enable_inv**

    Optional attribute used for ClockGating cells to define the functional or system enable.

22. **max_fanout**

    For Tessent Scan, Add Cell Models Buf –max_fanout.

```
max_fanout = integer_greater_than_1
```

23. **must_be_0_to_shift**

    Ignored on input. Output for write_cell_library for any scan cell input that must be 0 for a datapath to exist from scan_in to one or more scan_out or scan_out_inv. Not output for scan_enable_inv, or for active_high asynchs, that always must be 0 to shift, but only for additional cell input constraints necessary for shift.

24. **must_be_1_to_shift**

    Ignored on input. Output for write_cell_library for any scan cell input that must be 1 for a datapath to exist from scan_in to one or more scan_out or scan_out_inv. Not output for scan_enable, or for active_low asynchs, that always must be 1 to shift, but only for additional cell input constraints necessary for shift.

25. **mux_in0**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux_select* pin is 0.

26. **mux_in1**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux_select* pin is 1.

27. **mux_in2**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 2.

28. **mux_in3**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 3.

29. **mux_in4**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 4.

30. **mux_in5**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 5.

31. **mux_in6**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 6.

32. **mux_in7**

    Used for cell_type *mux*. Indicates that the pin is selected when the *mux select* pins have the value 7.

33. **mux_out**

    Optional for mux cell. If specified, indicates a non-inverting mux output.

34. **retention_enable**

    Used for an input that causes a sequential cell to retain its state when the input is high and not respond to its clock or asynch inputs. This attribute causes the model-level power_retention_cell attribute to also be set. The impact of specifying this attribute for one or more retention enable inputs of a library cell is the assignment of cell_type = prohibited. See the description of the prohibited attribute.

    If a cell_type is specified by the user, a warning is issued and the cell_type is changed to prohibited. If the cell is referenced in a dft cell selection, a warning is issued and it is removed from the dft cell selection.

35. **retention_enable_inv**

    Used for an input that causes a sequential cell to retain its state when the input is low and not respond to its clock or asynch inputs. This attribute causes the model-level power_retention_cell attribute to also be set. The impact of specifying this attribute for one or more retention enable input of a library cell is the assignment of cell_type = prohibited. See the description of the prohibited attribute

    If a cell_type is specified by the user, a warning is issued and the cell_type is changed to prohibited. If the cell is referenced in a dft cell selection, a warning is issued and it is removed from the dft cell selection.

36. **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.

37. **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.

38. **mux_select**

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

39. **mux_select0**

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

40. **mux_select1**

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

41. **mux_select2**

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

42. **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, and so on.

43. **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 also affect other library instances that 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 that 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);
       )
    )
    ```

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

    Each pin can have the characteristic of stuck-at-1 and stuck-at-0 faults. This attribute enables 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 are considered.
no_fault = sa1        \\ Specifies that no stuck-at-1 fault is
                      \\ considered at the specified pin. Only
                      \\ stuck-at-0 faults are 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) ()
 ...
 )
```

45. **open**

    Specifies an output pin that should be left unconnected when the cell is inserted as test logic.

46. **pad_ac_mode_dot6**

    When present, specifies that ETAssemble adds the ACM subclass to the pad. This mode is optional. For details, refer to ACMode in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*.

47. **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.

48. **pad_diff_current**

    Specifies that the pad is a differential voltage or differential current type cell. Used for differential.

49. **pad_diff_voltage**

Specifies that the pad is a differential voltage or differential current type cell. Used for differential.

50. **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.

51. **pad_enable_low**

    Specifies the active low enable pins for bidirectional or output pads. For more information, see Function in the *ETAssemble Tool Reference* manual.

52. **pad_force_disable**

    Used for cell_type pad.

53. **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 is forced to input even if the direction of the associated top pin is inout.

54. **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.

55. **pad_from_pad**

    Used for cell_type pad.

56. **pad_from_sje_mux**

    Used for cell_type MUX. Select Jtag Enable.

57. **pad_from_sji_mux**

    Used for cell_type pad. Select Jtag Input.

58. **pad_from_sjo_mux**

    Used for cell_type pad. Select Jtag Output.

59. **pad_init_clock_dot6**

    Required for all input and bidirectional AC pads. For details, refer to the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*.

Note

This attribute is deprecated. It has been replaced by pad_init_posedge_clock_dot6, pad_init_negedge_clock_dot6, pad_init_enable_high_dot6, or pad_init_enable_low_dot6.

60. **pad_init_posedge_clock_dot6**

    Required for all input and bidirectional AC pads. For details, refer to initClk in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*. It replaces pad_init_clock_dot6 because it also provides the clock capturing edge.

61. **pad_init_negedge_clock_dot6**

    Required for all input and bidirectional AC pads. For details, refer to initClk in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*. It replaces pad_init_clock_dot6 because it also provides the clock capturing edge.

62. **pad_init_enable_high_dot6**

    Required for all input and bidirectional AC pads. For details, refer to initClk in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*. It replaces pad_init_clock_dot6 because it also provides the clock capturing level.

63. **pad_init_enable_low_dot6**

    Required for all input and bidirectional AC pads. For details, refer to initClk in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*. It replaces pad_init_clock_dot6 because it also provides the clock capturing level.

64. **pad_init_data_dot6**

    Required for all input and bidirectional AC pads. For details, refer to InitData in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan.*

65. **pad_init_data_inv_dot6**

    Required for differential input and bidirectional AC pads. For details, refer to InitDataInv in the section “Pad Library File” in *Support for IEEE 1149.6 Boundary Scan*.

66. **pad_input_enable_high**

    Needed only when a pad contains logic inside that can gate off the path from the pad_pad_io port to the pad_from_pad port, to prevent switching going into the core. Specifies the pad input that enables the pad_pad_io to pad_from_pad path when the pad_input_enable_high pin is high. For details, see “Pad Cell Input Path Considerations for Boundary Scan Testing” in the *Tessent Boundary Scan User’s Manual.*

67. **pad_input_enable_low**

Needed only when a pad contains logic inside that can gate off the path from the pad_pad_io port to the pad_from_pad port, to prevent switching going into the core. Specifies the pad input that enables the pad_pad_io to pad_from_pad path when the pad_input_enable_low pin is low. For details, see "Pad Cell Input Path Considerations for Boundary Scan Testing" in the *Tessent Boundary Scan User's Manual.*

68. **pad_keep_tracing**

    Used for cell_type pad to enable 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.

69. **pad_open**

    Specifies the condition of a pin should be open for the pad to operate in a given mode (Usage).

70. **pad_open_drain**

    Used for cell_type pad.

71. **pad_open_source**

    Used for cell_type pad.

72. **pad_pad_io**

    Specifies the input/output connection from a top pin to its pad.

73. **pad_pad_io_inv**

    Specifies the input/output connection from the second top pin of a differential pair to its pad.

74. **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.

75. **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.

76. **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.

77. **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.

78. **pad_select_jtag_in**

    Specifies the connection for the control signal, usually coming from the TAP controller, that 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.

79. **pad_select_jtag_out**

    Specifies the connection for the control signal, usually coming from the TAP controller that 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.

80. **pad_test_data_dot6**

    Used for pad for 1149.6 AC test.

81. **pad_test_data_inv_dot6**

    Used for pad for 1149.6 AC test.

82. **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).

83. **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).

84. **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 is forced to output even if the direction of the associated top pin is inout.

85. **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.

86. **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.

87. **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.

88. **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.

89. **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.

90. **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.

91. **pad_two_state_output_enable_high**

    Needed to support pads with both an openDrain or openSource and an enable pin. For more information, see Function in the *ETAssemble Tool Reference* manual.

92. **pad_two_state_output_enable_low**

    Needed to support pads with both an openDrain or openSource and an enable pin. For more information, see Function in the *ETAssemble Tool Reference* manual.

93. **pad_input_keeper**

    A “kept” state memory of the last strongly driven logic state.

94. **pad_input_pull0**

    A weak 0 internal pull-down.

95. **pad_input_pull1**

    A weak 1 internal pull-up.

96. **pad_input_extern0**

    A 0 external pull- or tie-down for a signal not connected to a driving source.

97. **pad_input_extern1**

A 1 external pull- or tie-up for a signal not connected to a driving source.

98. **pad_input_open0**

    A received value of 0 when the signal is undriven.

99. **pad_input_open1**

    A received value of 1 when the signal is undriven.

100. **pad_input_openx**

    An indeterminate value when the signal is undriven.

101. **posedge_clock**

    Edge-triggered clock and polarity.

102. **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.

103. **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.

104. **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.

105. **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.

106. **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.

107. **scan_out_no_polarity**

    Deprecated DFTA scan_definition syntax specified a polarityless scan_out for a cell. Now, scan_out and scan_out_inv (relative to scan_in) are supported for the tessent

syntax. When parsing an old DFTA scan_definition, scan_out_no_polarity is assigned to all scan out pins initially. These are typically converted to either “scan_out” or “scan_out_inv” by learning, but in rare cases, the polarityless attribute is retained due to inability to learn a unique path parity for all paths from “scan_in” to the “scan_out_no_polarity” pin (all inverting or all non-inverting).

108. **test_enable**

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

For classic Siemens EDA 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 the original clock, set and reset; or the test clock, test set, and test reset.

109. **test_enable_inv**

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

110. **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.

111. **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.

112. **unused**

Specifies that a pin is not connected internally. By default, the tool issues a message 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 tool suppresses the warning message:

```
model test (IN1, IN2, OUT) (
   input (IN1) ()
   input (IN2) (unused)
   output (OUT) (data_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 four methods described here:

- **Method 1** — This method uses read_liberty to extract appropriate scan equivalents for a nonscan cell.

  An example invocation for method 1 is shown here:

  read_liberty utmc13.lib

- **Method 2** — Define the attributes using the *cell_attributes* syntax. This enables you to use regular expressions to define attributes on multiple cells and pins in a single expression.

  a. **Example 1:**

  In this example, a separate file *adk.attributes* is used whose contents are shown below.

  When processing the file, the tool assigns any model whose name starts with “sff” the nonscan_equivalent “dff” due to the first model level statement (attribute). Any pins in a matching model whose names match one of the names below (that 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 (
   model ("sff.*") (
      nonscan_equivalent = "dff";
      cell_type = scan_cell;
      input (SE) (scan_enable)
      input (SI) (scan_in)
      input (CLK)(posedge_clock)
      input (D)  (data_in)
      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* (*regexp=anchors* being the default). 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 regex** 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 enables 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_equivalent = dff;
      cell_type = scan_cell;
      input (SE) (scan_enable;)
      input (SI) (scan_in;)
      input (CLK)(posedge_clock)
      input (D)  (data_in;)
      output(Q)  (scan_out;)
      output(QB) (scan_out_inv;)
   )
)
```

You can use regular expressions to associate nonscan_equivalents that match the scan model name pattern by using back reference ("*d*") notations, where "*d*" is a

non-zero decimal number. For example, you can use “\1” to reference the match within the first set of parens, “\2” to reference the match within a second set of parens, and so on.

A single set of parens exists in the following example and the regular expression will match as shown in the table.

```
cell ("sff(|s|r|sr)"){
  nonscan_equivalent = "dff\1";
  cell_type = scan_cell;
  input ("SE") (scan_enable;)
  input ("SI") (scan_in;)
  input (CLK)  (posedge_clock)
  input ("D")  (data_in;)
  output ("Q") (scan_out;)
  output ("QB")(scan_out_inv;)
}
```

You can populate the scan_equivalent in the nonscan model using the same method:

```
cell ("dff(|s|r|sr)") {
   scan_equivalent = "sff\1";
}
```

cell	regular expression match	nonscan_equivalent	scan_equivalent
sff	-	dff	sff
sffs	s	dffs	sffs
sffr	r	dffr	sffr
sffsr	sr	dffsr	sffsr

The example shows a reciprocal relation between the nonscan_equivalent and scan_equivalent models, which is not common. Usually, the best nonscan_equivalent for a scan cell does not have that scan cell as its scan_equivalent because a better match for the nonscan_equivalent exists to meet output drive and area requirements.

b. **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* is assigned the *pad_to_pad* attribute.
- Similarly, any input pin beginning with *ipp_in* is assigned the *pad_from_pad* attribute and so on for each of the *output*, *input*, and *inout* statements.

- In the second cell declaration, model *FD1T* has a model-level attribute *nonscan_equivalent = FD1* added to its output, *FD2T* is 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* is assigned *nonscan_equivalent = DFF* instead of *nonscan_equivalent = FD3* because it is overridden in the third cell by the wrapper's *nonscan_equivalent* attribute statement.
- Because model *FD3T* matches “FD([1-9)]T(.*)” in the second cell, *FD3T* has 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* also has a *scan_out_inv* attribute placed on pin *SO_B* in the third cell declaration; this attribute is not placed on any of the other cells even if they have inverted output pins named *SO_B* because their model name does 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_equivalent = "FD\1\2";
      cell_type = scan_cell;
      output ("SO")( scan_out; )
      input ("TE") ( scan_enable; )
      input ("TI") ( scan_in; )
   }
   cell ("FD3T") { // third cell
      nonscan_equivalent = "DFF";
      cell_type = scan_cell;
      output ("SO_B") (scan_out_inv; )
   }
}
```

- **Method 3** — Edit to populate with attributes and replace directly.

```
model dff (D, CLK, Q, QB) (
   cell_type = dff;
   scan_equivalent = sff;
   input (D) (data_in;)
   input (CLK) (posedge_clock;)
   output (Q) (data_out;)
   output (QB) (data_out_inv;)
   (
     instance = dff_p (Q, viol_0, CLK, D);
     primitive = _inv mlc_inv_1 (Q, QB);
   )
) // end model dff

model sff (D, SI, SE, CLK, Q, QB) (
   cell_type = scan_cell;
   nonscan_equivalent = dff;
   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;)
   (
     instance = mux2 (int_res_D, SI, D, SE);
     instance = dff_p (Q, viol_0, CLK, int_res_D);
     primitive = _inv mlc_inv_1 (Q, QB);
   )
)
```

- **Method 4** — 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 below:

```
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 method 4 is shown here:

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

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

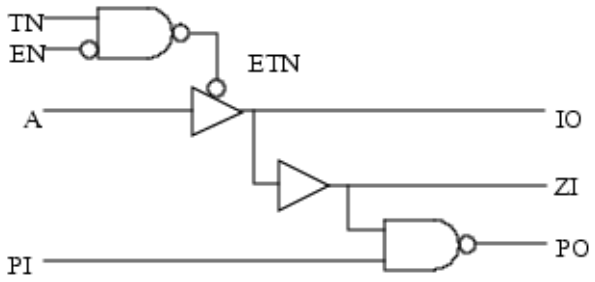
To write out a populated single file library, see the write_cell_library command.

Example 2

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);
   )
)
```

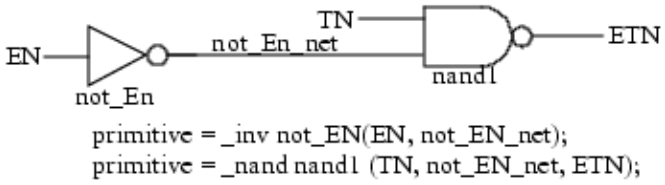
Figure 3-2. Combinational Logic



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:

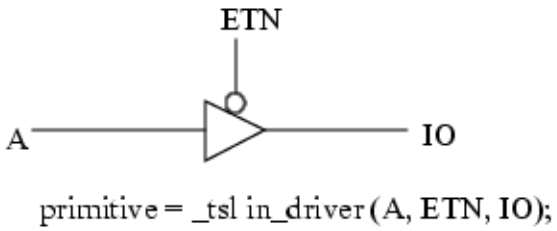
- An internal node such as not_en_net or ETN can be implied by referencing a name in a port list that is not an input, output, or inout pin of the model. For example:

Figure 3-3. Implying an Internal Node



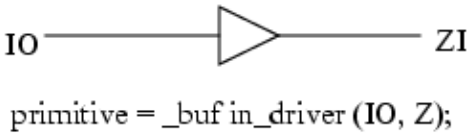
- For the tri-state buffer with input and active low pins, you can use a primitive attribute statement, "_tsl out_driver (A,ETN,IO)" (tsl = tri-state low), to model the driver of the bidirectional pin IO. Note that the internal node ETN is treated as the enable pin for the tri-state buffer because it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

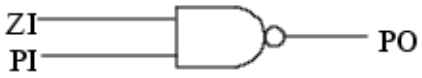
Figure 3-5. Non-Inverting Buffer



  Note that _buf converts 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

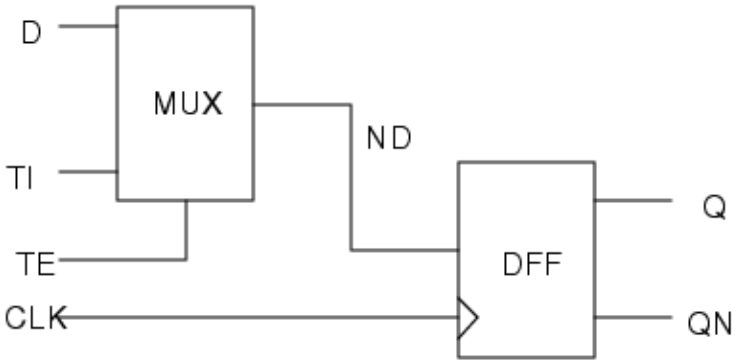


primitive = _nand nand2 (ZI, PI, PO);

```
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:

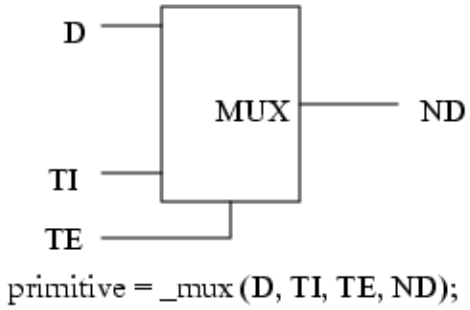
Figure 3-7. Mux-DFF Scan Cell



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

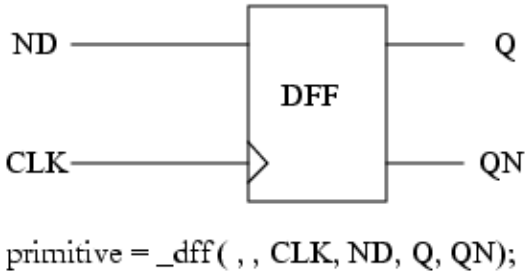
- Use the primitive statement "_mux(D, TI, TE, ND)" to describe the multiplexer. Syntax: primitive =_mux(I0, I1, CNT, OUT) where OUT = CNT*I1 + ~CNT*I0 + I1*I0.

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 required instantiation, so you should leave an empty pin to indicate this.

Figure 3-9. The DFF



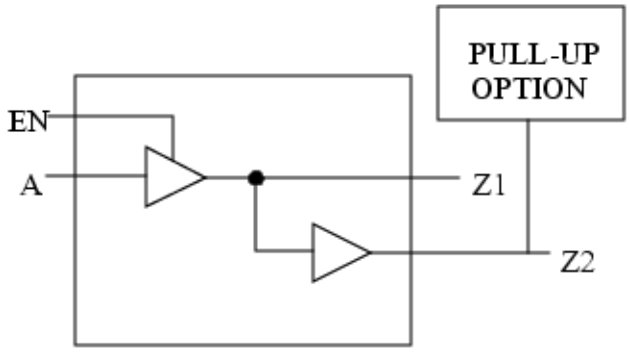

```
model SDFF(D, CLK, TI, TE, Q, QN) (
   input(D, TI, TE) ()
   input(CLK) ( )
   output(Q,QN) ( )
   (
      primitive = _mux(D, TI, TE, ND);
      primitive = _dff(, , CLK, ND, Q, QN);
   )
)
```

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, is 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)

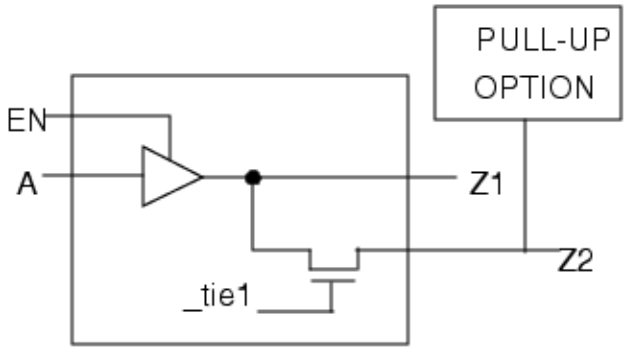


When this model is compiled, a combinational buffer is created between output pin Z1 and output pin Z2. The effect of modeling this way stops a Z state that can propagate to output pin Z1 from propagating to output pin Z2. The buffer causes the Z in its input to become an X (unknown) value at Z2. If there is an external pull up/down gate connected to output pin Z2, the effect of the pull up/down does 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)

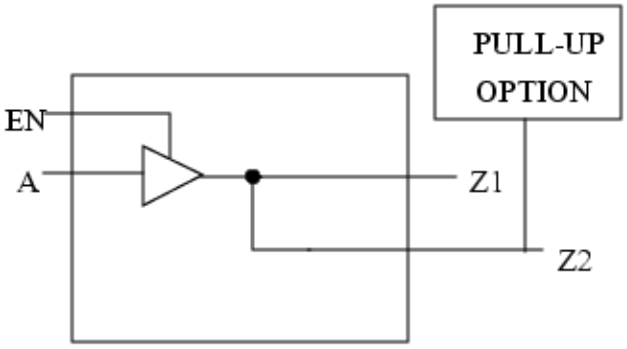


When this model is compiled, a wired on switch is created for output pin Z2, and a Z state can propagate to pin Z2 through the switch. However, if there is an external pull up/down gate connected to output pin Z2, the effect of the pull up/down does not show up at output pin Z1.

- 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 are created for output pin Z1 and output pin Z2, respectively. If output pin Z1 is a Z, Z2 also a Z. If there is an external pull up/down gate connected to output pin Z2, the effect of the pull up/down shows up at output pin Z1, and vice versa.

Internal Faults

You can place internal faults on cell models.

By default, the tool places faults on all interface pins on the boundary of the model. You have the option to model the cell without faults on any of these interface pins. If the cell model is complex and you want to model faults on some of the pins inside the cell, you can do this with primitive and instance attribute statements.

Three attribute statements describe the connectivity of cell models: function, primitive, and instance. Because 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.

If a cell is internally faulted, the tool does not apply faults on the boundary of the top level (read_verilog) instantiation of a (read_cell_library) cell model, but only to instances inside that cell model.

Cell model “fault =” statements control what is faulted inside. If no “fault =” statements occur within any level of hierarchy inside an internally faulted model, but all instances are named down to, and including, primitive instances, the primitive instance boundaries (with full pathnames down to, and including, the primitive instance name) have faults modeled on the boundary.

The tool ignores the “fault =” statements unless you have run the “set_internal_fault on” command.

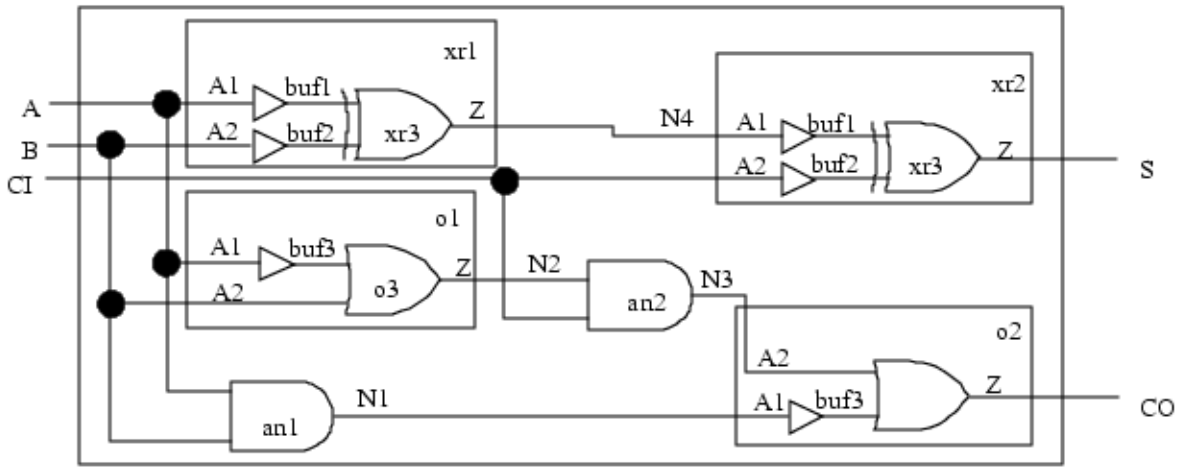
Fault sites must have fully specified pathnames down to, and including, the fault site itself. If any “instance =” statement within a cell has no instance name, no faults can occur on its boundary or within that unnamed instance. If a “primitive =” statement within a cell has no instance name, no faults can occur on its boundary (or inside, because it is a primitive and has no internal sites).

Examples

Example 1

Figure 3-13 shows an example using internal faults:

Figure 3-13. Internal Faults on the U1 Instance of the “adder” Model



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. You can describe the library model description for the adder with internal faulting as follows:

```
model adder(CI, A, B, S, CO)  (
   input(A, B, CI)  ()
   output(S) ()
   output(CO) ()
   (
      fault=internal; // Default value if no 'fault =' statement exists
      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=buff1 buf3(A1, N1);
      fault=boundary;
      primitive=_or o3(N1, A2, Z:nf1);
   )
)
model buff1(I, Z)  (
   input(I)  ()
   output(Z)  ()
   (
      fault = boundary;
      primitive = _buf buf4(I, Z);
   )
```

```
)
```

When internal faulting is enabled, and you add all faults to this example using the add_faults command, the tool places these faults as follows:

1. The tool places stuck-at-0 and stuck-at-1 faults in the primary inputs and primary outputs:

   ```
   /A
   /B
   /CI
   /S
   /CO
   ```

2. The tool places faults on the internals of instance xr1. This instance name refers to the library model xor2. Within library model xor2, it places no-faults on the instance name buf1, and stuck-at-0 and stuck-at-1 faults 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
   ```

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

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

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

3. The tool places faults on the boundary and the internals of instance xr2. This instance name refers to the internals and boundary of library model xor2. It places stuck-at-0 and stuck-at-1 faults on the boundary of library model xor2:

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

   Within library model xor2, it places no-faults on the instance name buf1. It also places faults 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
   ```

   Because library model xor2 has an instance name xr3 but has no-fault attribute statement, then by default it sets the instance_fault attribute to boundary. For the XOR primitive, IN0 and IN1 are the input pin names, OUT is the output pin name. However,

it places a no-fault attribute on the first pin of the XOR primitive (IN0). So, it only places stuck-at-0 and stuck-at-1 faults on the IN1 and OUT pins of the XOR primitive:

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

4. The tool places faults on the internals of instance an1. The instance name refers to the library model and2. However, because the library model contains an AND primitive and an instance_fault attribute statement set to none, it does not place any no instance_faults.

5. The tool places faults on the internals of instance o1. This instance name refers to the library model or2. Within library model or2 it places faults 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, it places faults 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
   ```

   It places faults 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, it places a stuck-at-1 no-fault attribute statement on the output pin of the OR primitive (OUT). So, it only places a stuck-at-0 fault 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. The tool places faults 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, it places faults only on the boundary of instance o2. Though instance o2 refers to library model or2 and has internal faults, it places no-faults 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.

You should declare arrays before they are referenced in the *primitive*, *instance*, or *function* statements. The '<' and '>' symbols are only required if a multi-bit part of an array needs to be referenced but not for single-bit index references or full-array references.

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

```
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, you must use a bit select / index 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, you can use the simple form:

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

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.

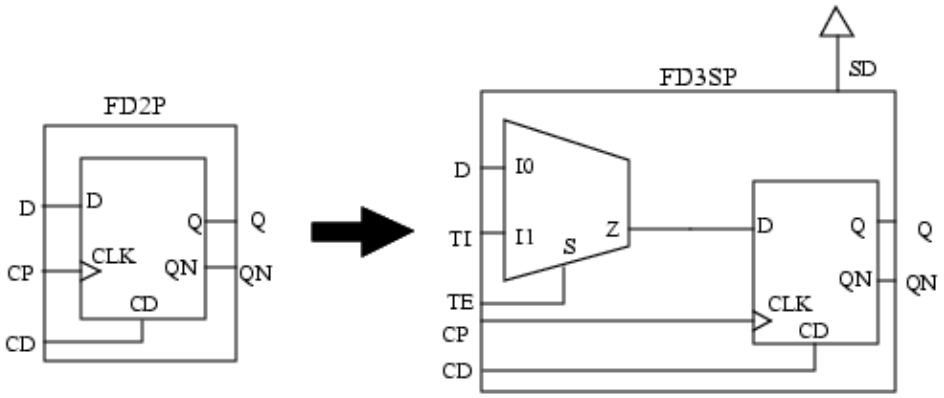
Basic Example

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

```
model FD3SP(D, CP, TI, TE, CD, Q, QN)  (
   nonscan_model = FD2P;
   cell_type = scan_cell;
   input (CD) (active_high_reset)
   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



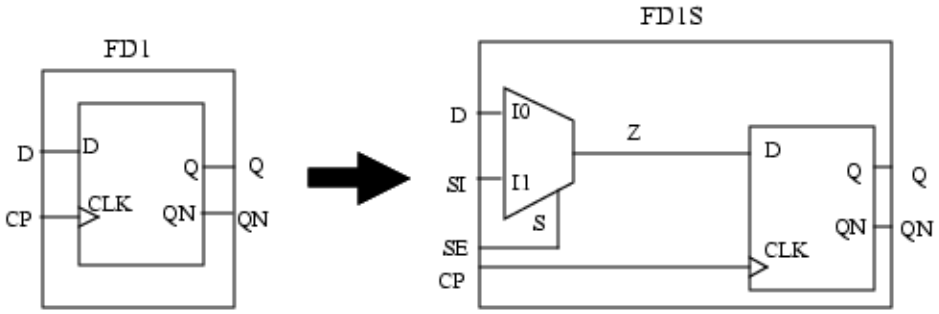
MUX-Scan Cell

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



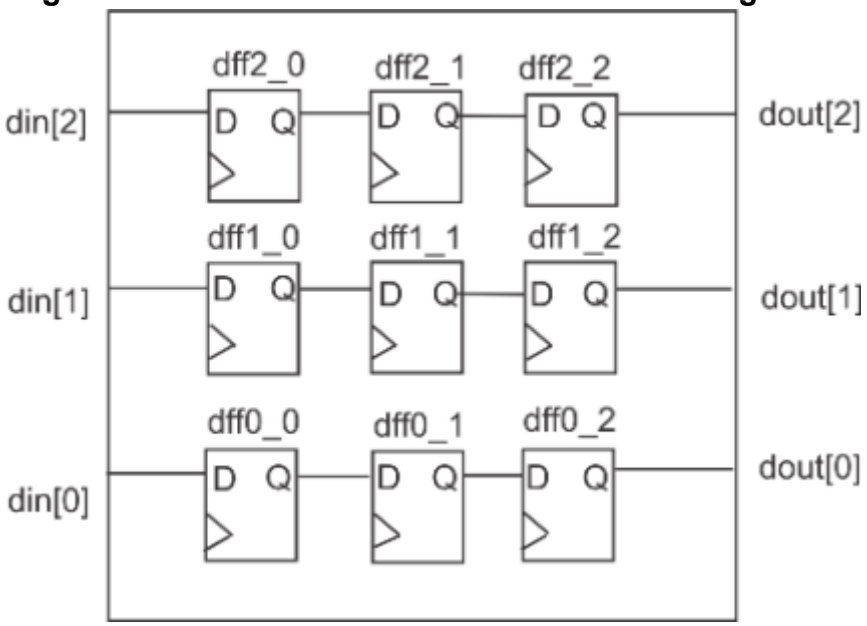
Non-Scan Cell With Vector Scan Segments

The following is an example of a non-scan cell with vector scan segments.

```
model three_scan_segment  (dout, clk, din)
(
   input (clk) ( )
   input (din) (array = 2:0)
   output (dout) (array = 2:0)
   (
      primitive = _dff dff2_0 ( , , clk, din[2], dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2],  );

      primitive = _dff dff1_0 ( , , clk, din[1], dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1],  );

      primitive = _dff dff0_0 ( , , clk, din[0], dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0],  );
   )
)
```

Figure 3-16 shows this non-scan cell with vector scan segments defined above.

Figure 3-16. Non-Scan Cell With Vector Scan Segments



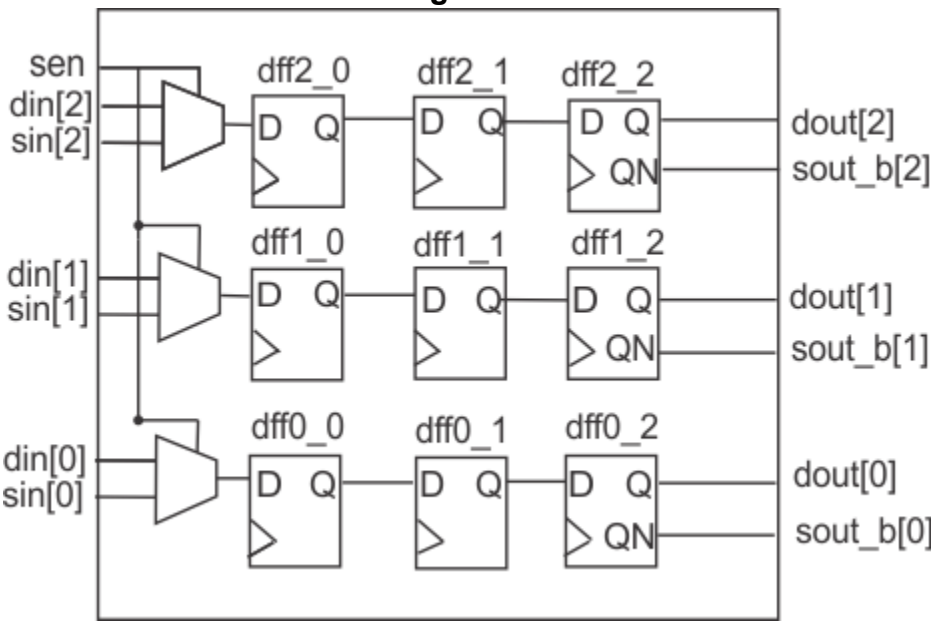
Scan-Cell Replacement for Non-Scan Cell With Vector Scan Segments

The following is an example definition of a non-scan cell with vector scan segments.

```
// Scan_in scan segments are implicitly defined.  For an array of > 1 bit,
// when a scan_in, scan_out or scan_out_inv, and scan_enable or
// scan_enable_inv port function attributes are declared,
// as for "input (sin)" below, then the rightmost/LS bit (sin[0]
// in this case) is assigned to scan_segment [0],(so "sin[0]" is
// implicitly assigned a "scan_in[0]" port function attribute).
// Similarly, port "sin[1]" is implicitly assigned port function
// "scan_in[1]", and port "sin[2]" is implicitly assigned port
// function "scan_in[2]". Note that if the array indices of
// port "sin" were declared as "array = 0:2", then the LS bit
// would be "sin[2]" which would still be assigned LS scan_segment
// index, which is always [0], so it would have implicit port
// attribute "scan_in[0]". Similarly, "sin[1]" would implicitly
// be "scan_in[1]", and finally MS bit "sin[0]" would implicitly
// be "scan_in[2]".

model three_scan_segment_scan
   (dout, sout_b, clk, din, sin, sen)
(
   scan_length = 3;

   input (sin) (array = 2:0; scan_in)
   input (sen) (scan_enable)
   input (clk) ( )
   input (din) (array = 2:0)
   output (dout) (array = 2:0)
   output (sout_b) (array = 2:0; scan_out_inv)
   (
      primitive = _mux mux2 (din[2], sin[2], sen, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);

      primitive = _mux mux1 (din[1], sin[1], sen, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
       primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);

      primitive = _mux mux0 (din[0], sin[0], sen, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);
   )
)
```

Figure 3-17 shows this non-scan to scan cell with vector scan segments replacement defined above.

Figure 3-17. Scan-Cell Replacement for Non-Scan Cell With Vector Scan Segments



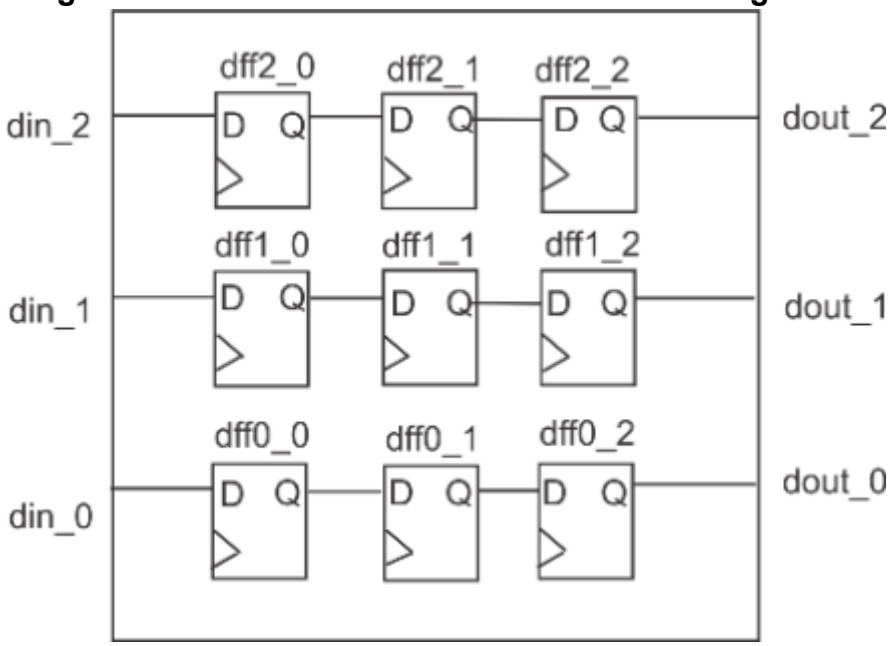
Non-Scan Cell With Scalar Scan Segments

The following is an example of a non-scan cell with scalar scan segments.

```
model three_scan_segment_scalar_inputs
   (dout_2, dout_1, dout_0, clk, din_2, din_1, din_0)
(
   input (clk) ( )
   input (din_2) ( )
   input (din_1) ( )
   input (din_0) ( )
   output (dout_2) ( )
   output (dout_1) ( )
   output (dout_0) ( )
   (
      primitive = _dff dff2_0 ( , , clk, din_2, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout_2,  );

      primitive = _dff dff1_0 ( , , clk, din_1, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout_1,  );

      primitive = _dff dff0_0 ( , , clk, din_0, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout_0,  );
   )
)
```

Figure 3-18 shows this non-scan cell with scalar scan segments defined above.

Figure 3-18. Non-Scan Cell With Scalar Scan Segments



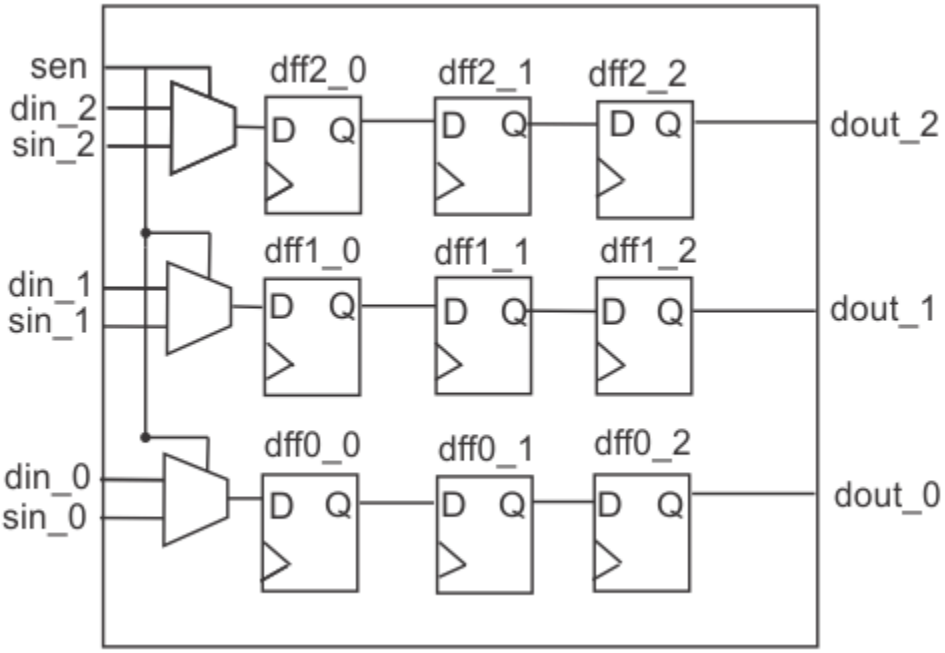
Scan-Cell Replacement for Non-Scan Cell With Scalar Scan Segments

The following is an example definition of a non-scan cell with scalar scan segments.

```
model three_scan_segment_scalar_inputs_scan
   (dout_2, dout_1, dout_0, clk, din_2, din_1, din_0, sin_0, sin_1, sin_2,
sen)
(
   scan_length[2] = 3;
   scan_length[1] = 3;
   scan_length[0] = 3;
   // Could also simply have scan_length = 3;
   //-- or -- scan_length[2:0] = 3;

   input (clk) ( )
   input (din_2) ( )
   input (din_1) ( )
   input (din_0) ( )
   input (sin_2) (scan_in[2])
   input (sin_1) (scan_in[1])
   input (sin_0) (scan_in[0])
   input (sen) (scan_enable)
   output (dout_2) (scan_out[2])
   output (dout_1) (scan_out[1])
   output (dout_0) (scan_out[0])
   (
      primitive = _mux mux2 (din_2, sin_2, sen, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout_2,  );

      primitive = _mux mux1 (din_1, sin_1, sen, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout_1,  );

      primitive = _mux mux0 (din_0, sin_0, sen, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout_0,  );
   )
)
```

Figure 3-19 shows this non-scan to scan cell with scalar scan segments replacement defined above

Figure 3-19. Scan-Cell Replacement for Non-Scan Cell With Vector Scan Segments



Scan-Cell Replacement for Non-Scan Cell With Scan Enable per Scan Segment

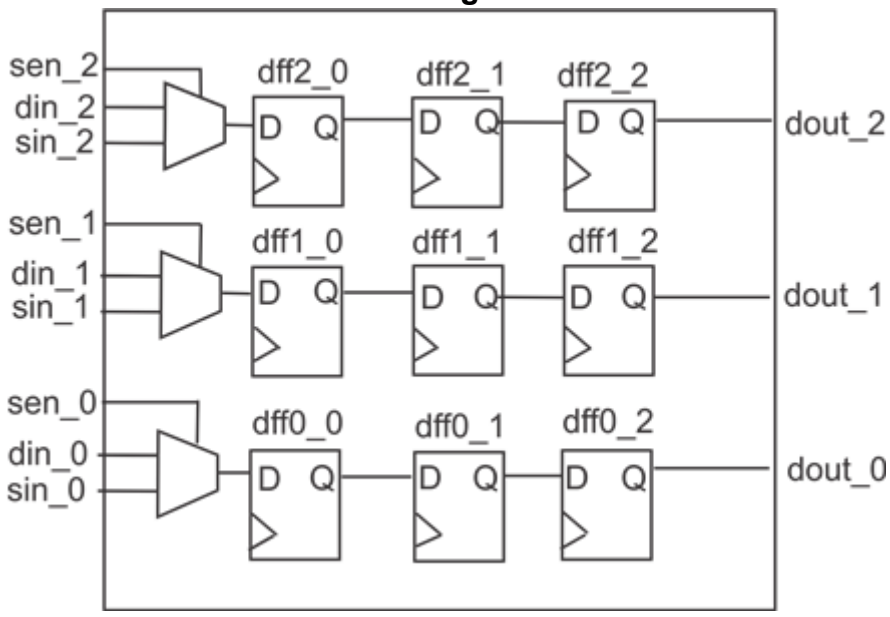
The following is an example definition of a non-scan cell with scan enable per scan segment.

Note

There must be a single scan enable for all scan segments or a scan enable per scan segment.

```
model three_scan_segment_three_sen_scalar_inputs_scan
   (dout_2, dout_1, dout_0, clk, din_2, din_1, din_0, sin_0, sin_1, sin_2,
sen_2, sen_1, sen_0)
(
   scan_length[2] = 3;
   scan_length[1] = 3;
   scan_length[0] = 3;
   // Could also simply have scan_length = 3;
   //-- or -- scan_length[2:0] = 3;

   input (clk) ( )
   input (din_2) ( )
   input (din_1) ( )
   input (din_0) ( )
   input (sin_2) (scan_in[2])
   input (sin_1) (scan_in[1])
   input (sin_0) (scan_in[0])
   input (sen_2) (scan_enable[2])
   input (sen_1) (scan_enable[1])
   input (sen_0) (scan_enable[0])
   output (dout_2) (scan_out[2])
   output (dout_1) (scan_out[1])
   output (dout_0) (scan_out[0])
   (
      primitive = _mux mux2 (din_2, sin_2, sen_2, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout_2,  );

      primitive = _mux mux1 (din_1, sin_1, sen_1, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout_1,  );

      primitive = _mux mux0 (din_0, sin_0, sen_0, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout_0,  );
   )
)
```

Figure 3-20. Scan-Cell Replacement for Non-Scan Cell With Scan Enable per Scan Segment



Mux-Scan Cell With Varying Length Scan Segments

The following is a snippet of a mux-scan cell with varying length scan segments.

```
model three_scan_segment_scan_different_length
   (dout, sout_b, clk, din, sin, sen)
   (
      scan_length[1:0] = 3;
      scan_length[2] = 2;
      .........
      .........
)
```

Mux-Scan Cells With Multiple Scan Enable Port Functions

The following are examples of mux-scan cells with multiple scan enable port functions.

```
model four_scan_segment_sen_range_model
  (dout, sout_b, clk, din, sin, sen0, sen1)
(
   scan_length = 3;

   input (sin) (array = 3:0; scan_in)
   input (sen0) (scan_enable[1:0])
   input (sen1) (scan_enable[3:2])
   input (clk) ( )
   input (din) (array = 3:0)
   output (dout) (array = 3:0)
   output (sout_b) (array = 3:0; scan_out_inv)
   (
      primitive = _mux mux0 (din[0], sin[0], sen0, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);

      primitive = _mux mux1 (din[1], sin[1], sen0, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);

      primitive = _mux mux2 (din[2], sin[2], sen1, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);

      primitive = _mux mux3 (din[3], sin[3], sen1, mux3_out);
      primitive = _dff dff3_0 ( , , clk, mux3_out, dff3_0net, );
      primitive = _dff dff3_1 ( , , clk, dff3_0net, dff3_1net, );
      primitive = _dff dff3_3 ( , , clk, dff3_1net, dout[3], sout_b[3]);

   )
)

model four_scan_segment_sen_array_range_model
  (dout, sout_b, clk, din, sin, sen)
(
   scan_length = 3;

   input (sin) (array = 3:0; scan_in)
   input (sen) (array = 1:0; index(1)(scan_enable[3:2]);
   index(0)(scan_enable[1:0]))
   input (clk) ( )
   input (din) (array = 3:0)
   output (dout) (array = 3:0)
   output (sout_b) (array = 3:0; scan_out_inv)
   (
      primitive = _mux mux0 (din[0], sin[0], sen[0], mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);

      primitive = _mux mux1 (din[1], sin[1], sen[0], mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);
```

```
      primitive = _mux mux2 (din[2], sin[2], sen[1], mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);

      primitive = _mux mux3 (din[3], sin[3], sen[1], mux3_out);
      primitive = _dff dff3_0 ( , , clk, mux3_out, dff3_0net, );
      primitive = _dff dff3_1 ( , , clk, dff3_0net, dff3_1net, );
      primitive = _dff dff3_3 ( , , clk, dff3_1net, dout[3], sout_b[3]);

   )
)


model four_scan_segment_sen_index_model
  (dout, sout_b, clk, din, sin, senB, senA)
(
   scan_length = 3;

   input (sin) (array = 3:0; scan_in)
   input (senA) (scan_enable[2]; scan_enable[0])
   input (senB) (scan_enable[3]; scan_enable[1])
   input (clk) ( )
   input (din) (array = 3:0)
   output (dout) (array = 3:0)
   output (sout_b) (array = 3:0; scan_out_inv)
   (
      primitive = _mux mux0 (din[0], sin[0], senA, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);

      primitive = _mux mux1 (din[1], sin[1], senB, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);

      primitive = _mux mux2 (din[2], sin[2], senA, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);

      primitive = _mux mux3 (din[3], sin[3], senB, mux3_out);
      primitive = _dff dff3_0 ( , , clk, mux3_out, dff3_0net, );
      primitive = _dff dff3_1 ( , , clk, dff3_0net, dff3_1net, );
      primitive = _dff dff3_3 ( , , clk, dff3_1net, dout[3], sout_b[3]);

   )
)

model four_scan_segment_sen_array_index_model
  (dout, sout_b, clk, din, sin, sen)
(
   scan_length = 3;

   input (sin) (array = 3:0; scan_in)
   input (sen) (array = 1:0; index(1)(scan_enable[3];scan_enable[1]);
```

```
index(0)(scan_enable[2];scan_enable[0]))
   input (clk) ( )
   input (din) (array = 3:0)
   output (dout) (array = 3:0)
   output (sout_b) (array = 3:0; scan_out_inv)
   (
      primitive = _mux mux0 (din[0], sin[0], sen[0], mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);

      primitive = _mux mux1 (din[1], sin[1], sen[1], mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);

      primitive = _mux mux2 (din[2], sin[2], sen[0], mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);

      primitive = _mux mux3 (din[3], sin[3], sen[1], mux3_out);
      primitive = _dff dff3_0 ( , , clk, mux3_out, dff3_0net, );
      primitive = _dff dff3_1 ( , , clk, dff3_0net, dff3_1net, );
      primitive = _dff dff3_3 ( , , clk, dff3_1net, dout[3], sout_b[3]);

   )
)

model four_scan_segment_sen_inv_range_model
  (dout, sout_b, clk, din, sin, sen0, sen_inv_1)
(
   scan_length = 3;

   input (sin) (array = 3:0; scan_in)
   input (sen0) (scan_enable[1:0])
   input (sen_inv_1) (scan_enable_inv[3:2])
   input (clk) ( )
   input (din) (array = 3:0)
   output (dout) (array = 3:0)
   output (sout_b) (array = 3:0; scan_out_inv)
   (
      primitive = _mux mux0 (din[0], sin[0], sen0, mux0_out);
      primitive = _dff dff0_0 ( , , clk, mux0_out, dff0_0net, );
      primitive = _dff dff0_1 ( , , clk, dff0_0net, dff0_1net, );
      primitive = _dff dff0_2 ( , , clk, dff0_1net, dout[0], sout_b[0]);

      primitive = _mux mux1 (din[1], sin[1], sen0, mux1_out);
      primitive = _dff dff1_0 ( , , clk, mux1_out, dff1_0net, );
      primitive = _dff dff1_1 ( , , clk, dff1_0net, dff1_1net, );
      primitive = _dff dff1_2 ( , , clk, dff1_1net, dout[1], sout_b[1]);

      primitive = _inv sen_inv (sen_inv_1, sen1);

      primitive = _mux mux2 (din[2], sin[2], sen1, mux2_out);
      primitive = _dff dff2_0 ( , , clk, mux2_out, dff2_0net, );
      primitive = _dff dff2_1 ( , , clk, dff2_0net, dff2_1net, );
      primitive = _dff dff2_2 ( , , clk, dff2_1net, dout[2], sout_b[2]);
```

```
        primitive = _mux mux3 (din[3], sin[3], sen1, mux3_out);
        primitive = _dff dff3_0 ( , , clk, mux3_out, dff3_0net, );
        primitive = _dff dff3_1 ( , , clk, dff3_0net, dff3_1net, );
        primitive = _dff dff3_3 ( , , clk, dff3_1net, dout[3], sout_b[3]);

    )
)
```

How to Define 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.

How to Reuse a Model Definition

This section describes how you can reuse a model definition.

Many times a library includes 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;
```

How to Read 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 quotation marks. 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 therefore defines “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 a cell library model named “and” is present, it is ignored by the tool because the Verilog reader recognizes this model as a primitive that the tool already understands.

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

Table 3-3. Supported Verilog Primitives

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

Notes:

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. For more information, refer to “Transistor Modeling Limitations”.

Note

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

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

Supported Primitives

This section contains 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 are testable in ATPG. The best practice is to use the tool's highest level built-in gate primitives.

The supported primitives are listed below and are described in the following sections.

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 3-4. AND Truth Table

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

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-21. AND Gate

Example 2:

```
model vec_AND3 (I1, I2, I3, O) (
  input (I1, I2, I3) (array = 2:0)
  output (O) (array = 2:0)
  (
    primitive = _and (I1, I2, I3, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_AND3
  (I1, I2, I3, O)
(

  input (I1) ( array = 2:0; )
  input (I2) ( array = 2:0; )
  input (I3) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _and (I1[2], I2[2], I3[2], O[2]);
    primitive = _and (I1[1], I2[1], I3[1], O[1]);
    primitive = _and (I1[0], I2[0], I3[0], O[0]);
  )
) // end model vec_AND3
```

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 3-5. NAND Truth Table

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

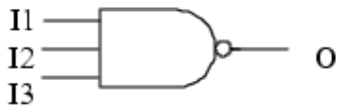
Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-22. NAND Gate



Example 2:

```
model vec_NAND3 (I1, I2, I3, O) (
  input (I1, I2, I3) (array = 2:0)
  output (O) (array = 2:0)
  (
    primitive = _nand (I1, I2, I3, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_NAND3
  (I1, I2, I3, O)
(

  input (I1) ( array = 2:0; )
  input (I2) ( array = 2:0; )
  input (I3) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _nand (I1[2], I2[2], I3[2], O[2]);
    primitive = _nand (I1[1], I2[1], I3[1], O[1]);
    primitive = _nand (I1[0], I2[0], I3[0], O[0]);
  )
) // end model vec_NAND3
```

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 3-6. OR Truth Table

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

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-23. OR Gate

Example 2:

```
model vec_OR3 (I1, I2, I3, O) (
  input (I1, I2, I3) (array = 2:0)
  output (O) (array = 2:0)
  (
    primitive = _or (I1, I2, I3, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_OR3
  (I1, I2, I3, O)
(

  input (I1) ( array = 2:0; )
  input (I2) ( array = 2:0; )
  input (I3) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _or (I1[2], I2[2], I3[2], O[2]);
    primitive = _or (I1[1], I2[1], I3[1], O[1]);
    primitive = _or (I1[0], I2[0], I3[0], O[0]);
  )
) // end model vec_OR3
```

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 3-7. NOR Truth Table

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

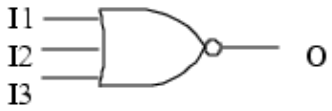
Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example:

```
model NOR3(I1, I2, I3, O) (
   input(I1, I2, I3) ()
   output(O) ()
   (
      primitive = _nor nor_1(I1, I2, I3, O);
   )
 )
```

Figure 3-24. NOR Gate



Example 2:

```
model vec_NOR3 (I1, I2, I3, O) (
  input (I1, I2, I3) (array = 2:0)
  output (O) (array = 2:0)
  (
    primitive = _nor (I1, I2, I3, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_NOR3
  (I1, I2, I3, O)
(

  input (I1) ( array = 2:0; )
  input (I2) ( array = 2:0; )
  input (I3) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _nor (I1[2], I2[2], I3[2], O[2]);
    primitive = _nor (I1[1], I2[1], I3[1], O[1]);
    primitive = _nor (I1[0], I2[0], I3[0], O[0]);
  )
) // end model vec_NOR3
```

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 3-8. Inverter Truth Table

IN	OUT
0	1
1	0
X	X

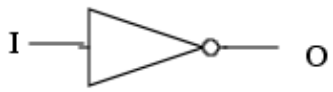
Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-25. Inverter



Example 2:

```
model vec_INV1 (I,O) (
   input (I) (array=2:0)
   output (O) (array=2:0)
   (
      primitive = _inv (I, O);
   )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_INV1
  (I, O)
(

  input (I) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _inv (I[2], O[2]);
    primitive = _inv (I[1], O[1]);
    primitive = _inv (I[0], O[0]);
  )
) // end model vec_INV1
```

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 3-9. Buffer Truth Table

IN	OUT
0	0
1	1
X	X

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-26. Buffer

Example 2:

```
model vec_BUF1 (I,O) (
   input (I) (array=2:0)
   output (O) (array=2:0)
   (
      primitive = _buf (I, O);
   )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_BUF1
  (I, O)
(

  input (I) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _buf (I[2], O[2]);
    primitive = _buf (I[1], O[1]);
    primitive = _buf (I[0], O[0]);
  )
) // end model vec_BUF1
```

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 3-10. XOR Truth Table

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

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-27. XOR Gate

Example 2:

```
model vec_XOR (A, B, Z) (
  input (A,B) (array=2:0)
  output (Z) (array = 2:0)
  (
     primitive = _xor (A, B, Z);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_XOR
  (A, B, Z)
(

  input (A) ( array = 2:0; )
  input (B) ( array = 2:0; )
  output (Z) ( array = 2:0; )
  (
    primitive = _xor (A[2], B[2], Z[2]);
    primitive = _xor (A[1], B[1], Z[1]);
    primitive = _xor (A[0], B[0], Z[0]);
  )
) // end model vec_XOR
```

XNOR Gate

The primitive used to model an 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 3-11. XNOR Truth Table

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

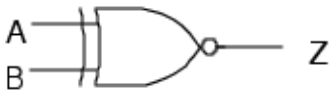
Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

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

Figure 3-28. XNOR Gate



Example 2:

```
model vec_XNOR (A, B, Z) (
  input (A,B) (array=2:0)
  output (Z) (array = 2:0)
  (
     primitive = _xnor (A, B, Z);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_XNOR
   (A, B, Z)
(

  input (A) ( array = 2:0; )
  input (B) ( array = 2:0; )
  output (Z) ( array = 2:0; )
  (
    primitive = _xnor (A[2], B[2], Z[2]);
    primitive = _xnor (A[1], B[1], Z[1]);
    primitive = _xnor (A[0], B[0], Z[0]);
  )
) // end model vec_XNOR
```

Tri-State Buffer With Active Low Control

The primitive used to model a tri-state buffer with an active low control is **_tsl**.

The syntax of the primitive attribute statement is as follows:

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

Table 3-12. TSL Truth Table

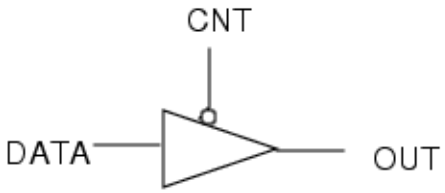
IN	CNT	OUT
0	0	0
1	0	1
X/Z	0	X
0/1/X/Z	1	Z
0	X/Z	L[1]
1	X/Z	H[2]
X/Z	X/Z	X

1. These simulate like Verilog L (0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
2. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

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

Figure 3-29. 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 3-13. TSLI Truth Table

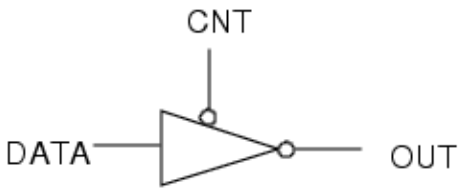
IN	CNT	OUT
0	0	1
1	0	0
X/Z	0	X
0/1/X/Z	1	Z
0	X/Z	H[1]
1	X/Z	L[2]
X/Z	X/Z	X

1. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
2. These simulate like Verilog L (0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

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

Figure 3-30. 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 an active high control is **_tsh**.

The syntax of the primitive attribute statement is as follows:

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

Table 3-14. TSH Truth Table

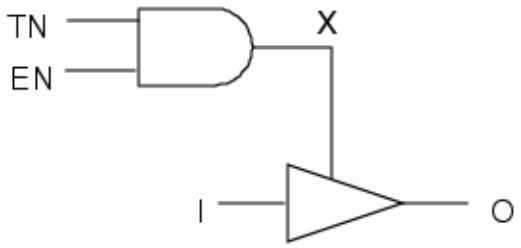
IN	CNT	OUT
0	1	0
1	1	1
X/Z	1	X
0/1/X/Z	0	Z
0	X/Z	H[1]
1	X/Z	L[2]
X	X/Z	X

1. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
2. These simulate like Verilog L (0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

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

Figure 3-31. 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**.

The syntax of the primitive attribute statement is as follows:

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

Table 3-15. TSHI Truth Table

IN	CNT	OUT
0	1	1
1	1	0
X/Z	1	X
0/1/X/Z	0	Z
0	X/Z	H[1]
1	X/Z	L[2]
X	X/Z	X

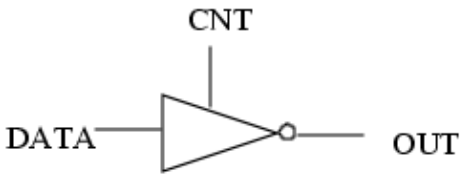
1. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

2. These simulate like Verilog L (0 or Z) and if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

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

Figure 3-32. Inverted Tri-State Buffer With Active High Control



Multiplexer

The primitive used to model a two-to-one multiplexer is **_mux**.

The syntax of the primitive attribute statement is:

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

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

Table 3-16. MUX Truth Table

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

Table 3-16. MUX Truth Table (cont.)

IN0	IN1	CNT	OUT
0	1	X	X
1	0	X	X

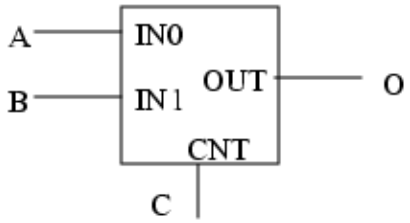
Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Example 1:

```
model MUX1(A, B, C, O) (
   input(A) ()
   input(B) ()
   input(C) ()
   output(O) ()
   (
      primitive = _mux consensus_mux_1(A, B, C, O);
   )
)
```

Figure 3-33. Multiplexer



Example 2, shared select vector:

```
model vec_mux_shared_select (A, B, SEL, O) (
  input (A, B) (array = 2:0)
  input (SEL) ( )
  output (O) (array = 2:0)
  (
     primitive = _mux (A, B, SEL, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_mux_shared_select
  (A, B, SEL, O)
(

  input (A) ( array = 2:0; )
  input (B) ( array = 2:0; )
  input (SEL) ( )
  output (O) ( array = 2:0; )
  (
    primitive = _mux (A[2], B[2], SEL, O[2]);
    primitive = _mux (A[1], B[1], SEL, O[1]);
    primitive = _mux (A[0], B[0], SEL, O[0]);
  )
) // end model vec_mux_shared_select
```

Example 3, select per bit vector mux:

```
model vec_mux_select_per_bit (A, B, SEL, O) (
  input (A, B, SEL) (array=2:0)
  output (O) (array=2:0)
  (
     primitive = _mux (A, B, SEL, O);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_mux_select_per_bit
  (A, B, SEL, O)
(

  input (A) ( array = 2:0; )
  input (B) ( array = 2:0; )
  input (SEL) ( array = 2:0; )
  output (O) ( array = 2:0; )
  (
    primitive = _mux (A[2], B[2], SEL[2], O[2]);
    primitive = _mux (A[1], B[1], SEL[1], O[1]);
    primitive = _mux (A[0], B[0], SEL[0], O[0]);
  )
) // end model vec_mux_select_per_bit
```

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 enables you to define a D flip-flop with a single pair or multiple pairs of clock and data inputs.

Table 3-17 shows the truth table for D flip-flop primitives for Tessent FastScan and Tessent TestKompress. This is the default and is equivalent to the command set_simulation_options -set_reset_dominate_clock ON.

Table 3-17. 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 set_xclock_handling command with the ‘X’ option, the tool treats these values as potential rising transitions.					

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

Table 3-18 shows the alternative truth table for the D flip-flop primitive for Tessent FastScan and Tessent TestKompress when you disable set/reset port dominance using the set_simulation_options -set_reset_dominate_clock OFF switch.

Table 3-18. Alternative D Flip-Flop Primitive Table

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
1	01	0	1	X	X

Table 3-18. Alternative D Flip-Flop Primitive Table (cont.)

D1	CLK1	SET	RESET	Q	QN
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.					

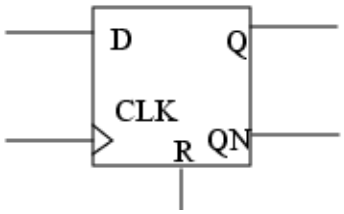
If the primitive is used to model a multiple port D flip-flop, the behavior is not affected by the attributes set_clock_conflict and reset_clock_conflict. The default behavior for Tessent FastScan and Tessent TestKompress of a multiple port D flip-flop is as follows:

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_podedge_DFF (d, clk, q) (
   input (d) ()
   input (clk) ()
   output (Q) ()
   (  // 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) (
   input (d) ()
   input (clk_b) ()
   output (Q) ()
   (  // All Tessent 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) (
   input(D) ()
   input(CLK) ()
   input(R) ()
   output(Q) ()
   output(QN) ()
   (
         primitive = _dff( , R, CLK, D, Q, QN);
   )
)
```

Figure 3-34. 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) (
   input (setb) ()
   input (rstb) ()
   input (d) ()
   input (ck) ()
   output (q) ()
   output (qb) ()
   (
      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);
   )
 )
```

Vector DFF Example:

```
model vec_dff (D, CLK, Q) (
  input (D) (array=2:0)
  input (CLK) ( )
  output (Q) (array=2:0)
  (  // set (if any), reset (if any) and clock are shared across all bits,
     // and must be 1-bit references.
     primitive = _dff ( , , CLK, D, Q,  );
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_dff
  (D, CLK, Q)
(

  input (D) ( array = 2:0; )
  input (CLK) ( )
  output (Q) ( array = 2:0; )
  (
    primitive = _dff (  ,  , CLK, D[2], Q[2],  );
    primitive = _dff (  ,  , CLK, D[1], Q[1],  );
    primitive = _dff (  ,  , CLK, D[0], Q[0],  );
  )
) // end model vec_dff
```

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 enables you 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 Tessent FastScan and Tessent TestKompress and is shown in the primitive table (Table 3-19).

When two _dlat primitives are connected in a master-slave configuration to implement a D flip-flop, the tool merges them into a single _dff primitive as part of library processing. When necessary, the tool flattens the hierarchical levels containing the two latches in the library model containing both latches.

The result of the merging is that it simplifies DRC. There is no impact on ATPG or simulation except rare cases where they run faster because clock -off simulation is no longer required. Also, write_cell_library reflects the post-merged models.

Table 3-19. 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	0	0	0	Q	QN
1	1	0	1	0[1]	1[1]
				X[2]	X[2]
0/1/X	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	0	1	0	1	0
0/1/X	0/1/X	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_clock switch to OFF.

Note

The tool converts Z values to X using a ZVAL gate before arriving at Boolean primitive inputs that are driven by tristateable nets. The ZVAL gate passes 0 as 0, 1 as 1, X as X, and Z as X.

The default behavior for 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_clock" is set to OFF, then Q and QN are unknown.

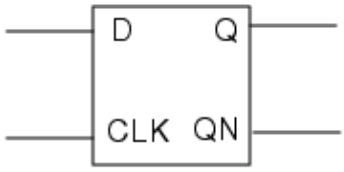
Note

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) (
   input (d) ()
   input (clk) ()
   output (Q) ()
   (  // 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) (
   input (d) ()
   input (clk_b) ()
   output (Q) ()
   (  // All Tessent 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) (
   input (D) ()
   input (CLK) ()
   output (Q) ()
   output (QN) ()
   (
      primitive = _dlat( , , CLK, D, Q, QN);
   )
 )
```

Figure 3-35. 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) (
   input (setb) ()
   input (rstb) ()
   input (d) ()
   input (ck) ()
   output (q_out) ()
   output (QN) ()
   (
      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);
   )
 )
```

Vector Latch Example:

```
model vec_latch (D, CLK, Q) (
  input (D) (array=2:0)
  input (CLK) ( )
  output (Q) (array=2:0)
  (  // set (if any), reset (if any) and clock are shared across all bits,
     // and must be 1-bit references.
     primitive = _dlat ( , , CLK, D, Q,  );
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_latch
  (D, CLK, Q)
(

  input (D) ( array = 2:0; )
  input (CLK) ( )
  output (Q) ( array = 2:0; )
  (
    primitive = _dlat (  ,  , CLK, D[2], Q[2],  );
    primitive = _dlat (  ,  , CLK, D[1], Q[1],  );
    primitive = _dlat (  ,  , CLK, D[0], Q[0],  );
  )
) // end model vec_latch
```

XDET

The **_xdet** primitive detects if the value on a pin is X.

The syntax of the primitive attribute statement is as follows:

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

The **_xdet** primitive has different behaviors for DRC and ATPG, as shown in Table 3-20.

For DRC, the output OUT is modeled as an unknown value (X) when input IN is X. For ATPG, the output OUT is high (1) when input IN is X.

For both DRC and ATPG, the output OUT is low (0) when the input IN is driven to a binary value or the high impedance state (Z).

Table 3-20. XDET Truth Table

IN	OUT for DRC	OUT for ATPG
X	X	1
0/1/Z	0	0

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 3-21. 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

Table 3-21. WIRE Truth Table (for two inputs) (cont.)

IN0/IN1	0	1	X	Z*	Weak 0	Weak 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.						

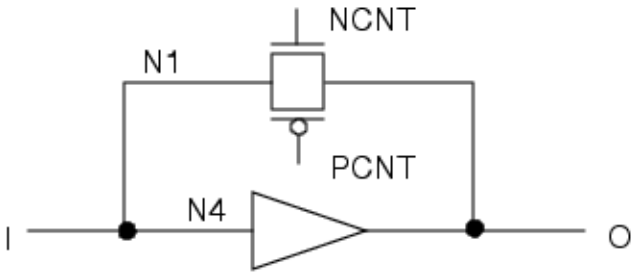
Note

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 3-36. 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 3-22. 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 3-37. 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 that 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 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 (VDD);
```

Example 1:

```
model VDD (vdd) (
  output (vdd) ( )
  (
     primitive = _tie1 (vdd);
  )
)
```

Example 2:

```
model vec_ONES (ones) (
  output (ones) (array=2:0)
  (
     primitive = _tie1 (ones);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_ONES
  (ones)
(

  output (ones) ( array = 2:0; )
  (
    primitive = _tie1 (ones[2]);
    primitive = _tie1 (ones[1]);
    primitive = _tie1 (ones[0]);
  )
) // end model vec_ONES
```

Ground Signal

The primitive used to model a ground signal is **_tie0**.

The syntax of the primitive attribute statement is as follows:

```
primitive = _tie0 (vss);
```

Example:

```
model VSS (vss) (
  output (vss) ( )
  (
     primitive = _tie0 (vss);
  )
)
```

Example 2:

```
model vec_ZEROS (zeros) (
  output (zeros) (array=2:0)
  (
     primitive = _tie0 (zeros);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_ZEROS
  (zeros)
(

  output (zeros) ( array = 2:0; )
  (
    primitive = _tie0 (zeros[2]);
    primitive = _tie0 (zeros[1]);
    primitive = _tie0 (zeros[0]);
  )
) // end model vec_ZEROS
```

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 1:

```
model pessimism (in, selectX, Xout) (
  input (in, selectX) ( )
  output (Xout) ( )
  (
     primitive = _tiex (tiex_net);
     primitive = _mux (in, tiex_net, selectX, Xout);
  )
)
```

Example 2:

```
model vec_pessimism (in, selectX, Xout) (
  input (in) (array = 2:0)
  input (selectX) ( )
  output (Xout) (array = 2:0)
  intern (tiex_net) (array = 2:0)
  (
     primitive = _tiex (tiex_net);
     primitive = _mux (in, tiex_net, selectX, Xout);
  )
)
```

The above model is expanded into the following for downstream consumption:

```
model vec_pessimism
  (in, selectX, Xout)
(

  intern (tiex_net) ( array = 2:0)
  input (in) ( array = 2:0; )
  input (selectX) ( )
  output (Xout) ( array = 2:0; )
  (
    primitive = _tiex (tiex_net[2]);
    primitive = _tiex (tiex_net[1]);
    primitive = _tiex (tiex_net[0]);
    primitive = _mux (in[2], tiex_net[2], selectX, Xout[2]);
    primitive = _mux (in[1], tiex_net[1], selectX, Xout[1]);
    primitive = _mux (in[0], tiex_net[0], selectX, Xout[0]);  )
) // end model vec_pessimism
```

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);
```

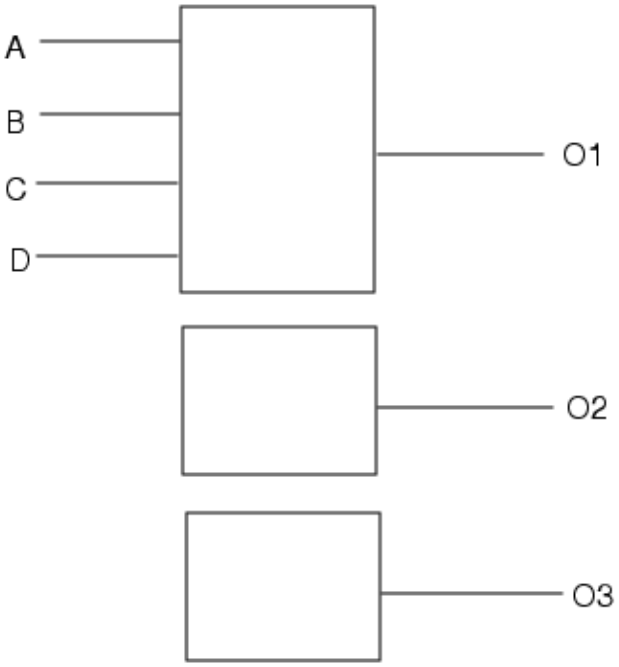
Table 3-23. UNDEFINED Truth Table

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 3-38. 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 3-24. NMOS Truth Table

I	EN	O
0/1/X/Z	0	Z
0	1	0
1	1	1
Z	0/1/X/Z	Z
X	1	X
0	X/Z	L[1]
1	X/Z	H[2]
X	X/Z	X

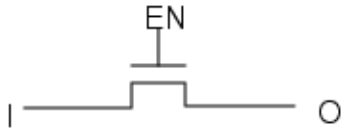
1. These simulate like Verilog L (0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

2. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

```
model NMOS1(I, EN, O) (
   input(I, EN) ()
   output(O) ()
   (
      primitive = _nmos(I, EN, O);
   )
)
```

Figure 3-39. 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 3-25. PMOS Truth Table

I	EN	O
0/1/X/Z	1	Z
0	0	0
1	0	1
Z	0/1/X/Z	Z
X	0	X
0	X/Z	L[1]
1	X/Z	H[2]
X	X/Z	X

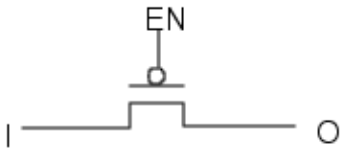
1. These simulate like Verilog L (0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

2. These simulate like Verilog H (1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

```
model PMOS1(I, EN, O) (
   input(I, EN) ()
   output(O) ()
   (
      primitive = _pmos(I, EN, O);
   )
)
```

Figure 3-40. 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 3-26. RNMOS Truth Table

I	EN	O
0/1/X/Z	0	Z
0	1	Weak 0
1	1	Weak 1
Z	0/1/X/Z	Z
X	1	Weak X
0	X/Z	Weak L[1]
1	X/Z	Weak H[2]
X	X/Z	Weak X

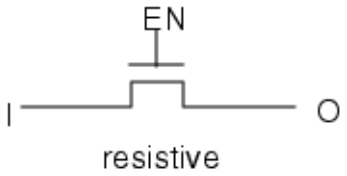
1. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

2. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

```
model RNMOS(I, EN, O) (
   input(I, EN) ()
   output(O) ()
   (
      primitive = _rnmos(I, EN, O);
   )
)
```

Figure 3-41. Unidirectional Resistive PMOS Transistor



Unidirectional Resistive PMOS Transistor

The primitive used to define a resistive PMOS transistor is **_rpmos**.

The syntax of the primitive attribute statement is as follows:

```
primitive = _rpmos optional_inst_name(IN, PCNT, OUT);
```

Table 3-27. RPMOS Truth Table

IN	PCNT	OUT
0/1/X/Z	1	Z
0	0	Weak 0
1	0	Weak 1
Z	0/1/X/Z	Z
X	0	Weak X
0	X/Z	Weak L[1]
1	X/Z	Weak H[2]
X	X/Z	Weak X

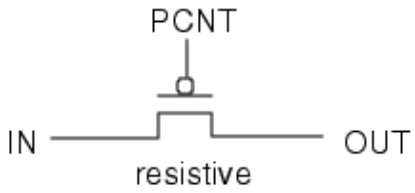
1. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

2. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

```
model RPMOS1(IN, PCNT, OUT) (
   input(IN, PCNT) ()
   output(OUT) ()
   (
      primitive = _rpmos(IN, PCNT, OUT);
   )
)
```

Figure 3-42. Unidirectional Resistive NMOS Transistor



Unidirectional Resistive CMOS Transistor

The keyword used to define a resistive CMOS transistor (that can be turned on when E is high or P is low) is **_rcmos**.

The syntax of the primitive attribute statement is:

```
primitive = _rcmos optional_inst_name(I, E, P, O);
```

Table 3-28. RCMOS Truth Table

I	E	P	O
0/1/X/Z	0	1	Z
0	1	0/1/X	Weak 0
1	1	0/1/X	Weak 1
Z	0/1/X/Z	0/1/X	Z
X	1	0/1/X	Weak X
0	0/1/X/Z	0	Weak 0
1	0/1/X/Z	0	Weak 1
X	0/1/X/Z	0	Weak X
0	0	X	Weak L[1]
1	0	X	Weak H[2]

Table 3-28. RCMOS Truth Table (cont.)

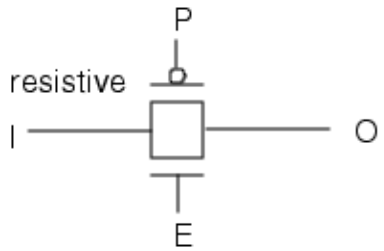
I	E	P	O
X	0	X	Weak X
0	X	1	Weak L[3]
1	X	1	Weak H[4]
X	X	1	Weak X

1. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
2. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
3. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
4. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

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 CMOS Transistor

The primitive used to model a CMOS transistor (that 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 3-29. CMOS Truth Table

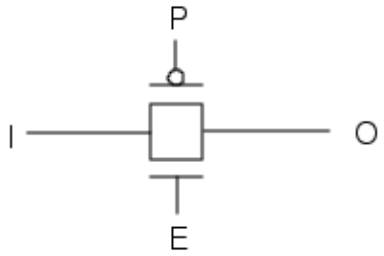
I	E	P	O
0/1/X/Z	0	1	Z
0	1	0/1/X/Z	0
1	1	0/1/X/Z	1
Z	0/1/X/Z	0/1/X/Z	Z
X	1	0/1/X/Z	X
0	0/1/X/Z	0	0
1	0/1/X/Z	0	1
X	0/1/X	0	X
0	0	X	L[1]
1	0	X	H[2]
X	0	X	X
0	X	1	L[3]
1	X	1	H[4]
X	X	1	X

1. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
2. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
3. These simulate like weak Verilog L (weak 0 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.
4. These simulate like weak Verilog H (weak 1 or Z) if they drive a BUS that cannot have contention, else like Verilog X, but always but display as X in reports and schematic views.

Example:

```
model CMOSX1(I, E, P, O) (
   input(I, E, P) ()
   output(O) ()
   (
      primitive = _cmos (I, E, P, O);
   )
)
```

Figure 3-44. Unidirectional CMOS Transistor



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 enable multiple pulses with different effective timing to be generated. You can assume that the combinational delays of the circuit are such that all paths that need to stabilize between different pulses from choppers 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 <= delay < 64K and the width must be in the range 1 <= 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 apply:

- You can only clock a sequential element at most once in any cycle (C10).
- Each sequential element has only a single state value, so you can only capture it by sinks that are either always clocked before or always clocked after the source, not a mixture (D10).

A limitation of this feature is that you cannot use any clock pin driving pulse generators in a clock procedure with other clocks. The output of a pulse generator must not propagate to a PO. (Any such PO is classified as a clock PO. However, as the output of a pulse generator is at X during clock PO pattern simulation, it is likely that some test coverage is 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 is used to cover this situation, too.

You can have no more than 31 unique events associated with the pulsing of any clock pin. This means that after counting the rising and falling edge events of the clock, you may use 29 additional discrete times 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 tool schedules rising and falling edge events at the output of the pulse generator to create events in the order defined by their delay and width. It generates additional simulation steps to simulate the output changes of pulse generators. It stabilizes all internally generated events before it simulates the next test procedure event and advances the time. In this sense, the delay and width attributes are in units of deltas that are infinitesimally small compared to the time units used to define test procedures.

The tool requires the input to a pulse generator at a binary value when all clock pins are in their inactive state and constrained PIs are placed at the constrained value. If the input value is a 1 under these conditions, it flags the pulse generator with the “inactive-high” property, and the parallel pattern simulator considers 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.

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 that 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 "RAM and ROM Test Overview" 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 Reference Manual*.

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 :

```
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].

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 239.

The content of the CRAM is provided through an initialization file. See "Initialization Files for RAM and ROM" on page 247 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 cell library must define a valid initialization file. If the model does not define an initialization file, the tool treats the CRAM as TIEX for ATPG, which lowers 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

You may use the following attributes 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**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 Checks" section of the *Tessent Scan and ATPG User's Manual*.

- **init_file= "file_name";**

  This optional attribute specifies the file, in Siemens EDA modelfile format, defining initial memory values. Specify the full path name to the initialization file if the file is not located in the directory where the model containing the init_file statement is located. Environment variables such as $init_dir are expanded before attempting to open the file. See Example in the RAM/ROM Library Primitives section above for illustration.

- **edge_trigger= [R | W | RW];**

  This optional attribute specifies the edge trigger values of the read 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.

- **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.

- **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.

Table 3-30. read_write_conflict States

	Values on Read Port	Value Stored in Memory
NW	New Data	New Data
XW	X	New Data
OW	Old Data	New Data
XX	X	X
OX	Old Data	X

- **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 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 enables 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.

  - 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)
```

Table 3-31. write_write_conflict Option 1

Option	Result	Notes
same_address_bit_data_consensus	1XX0	If writing to same word (address), 1XX0 written
same_address_port_data_consensus	XXXX	If writing to same word (address), XXXX written
same_address_x_port	XXXX	If writing to same word (address), XXXX written
always_x_words	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)
```

Table 3-32. write_write_conflict Option 2

Option	Result	Notes
same_address_bit_data_consensus	1010	If writing to same word (address), 1010 written
same_address_port_data_consensus	1010	If writing to same word (address), 1010 written
same_address_x_port	XXXX	If writing to same word (address), X all bits
always_x_words	XXXX	Always write all bits X if multiple writes

Example 3

```
_write port A is writing data 1010 to an address (word) A
_write port B is writing data 1010 to same address (word) B
```

Table 3-33. write_write_conflict Option 3

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

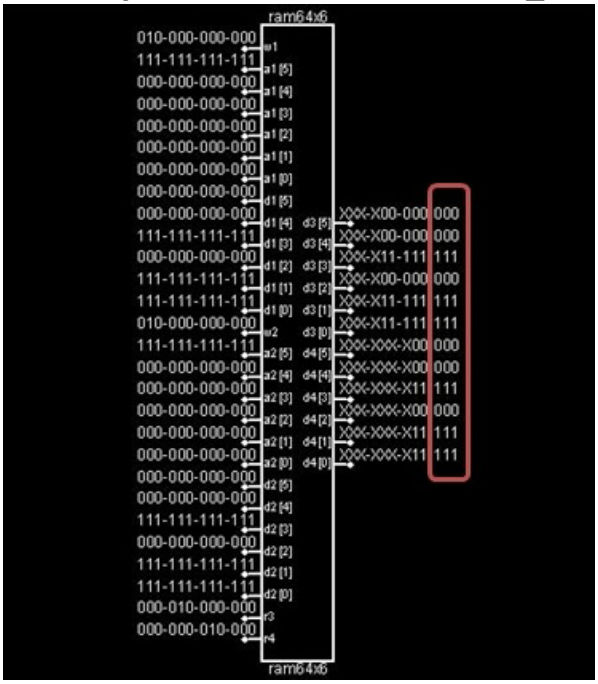
Table 3-33. write_write_conflict Option 3 (cont.)

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

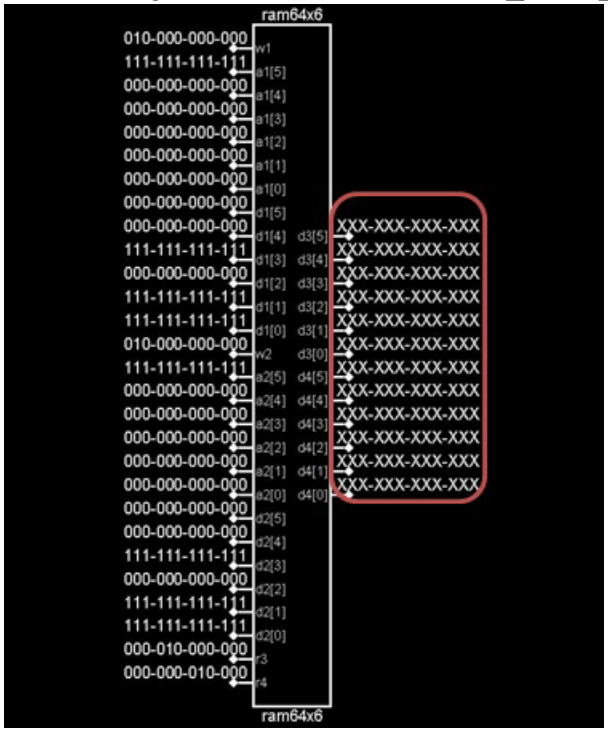
Figure 3-45 and Figure 3-46 below illustrate the effect of using the write_write_conflict attribute.

Figure 3-45. 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 3-46. 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.

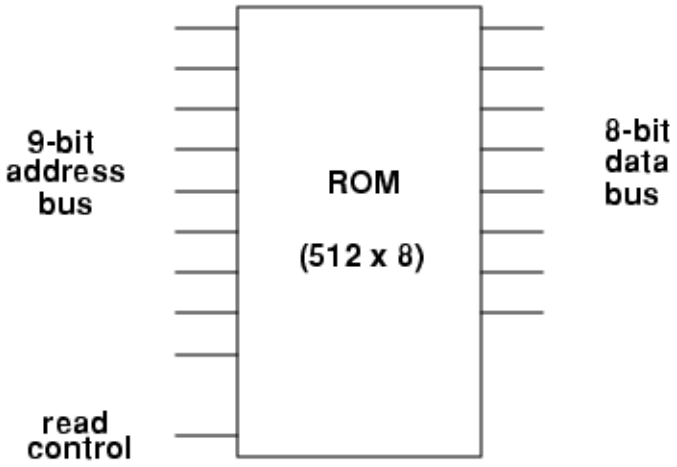
RAM and ROM Basics

A ROM is an array of memory cells whose contents are accessible through the activities of one or more read ports.

Each read port 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 3-47 shows a ROM.

Figure 3-47. 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)*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 3-47 uses addresses in the range 0-511 and can access 512 8-bit words.

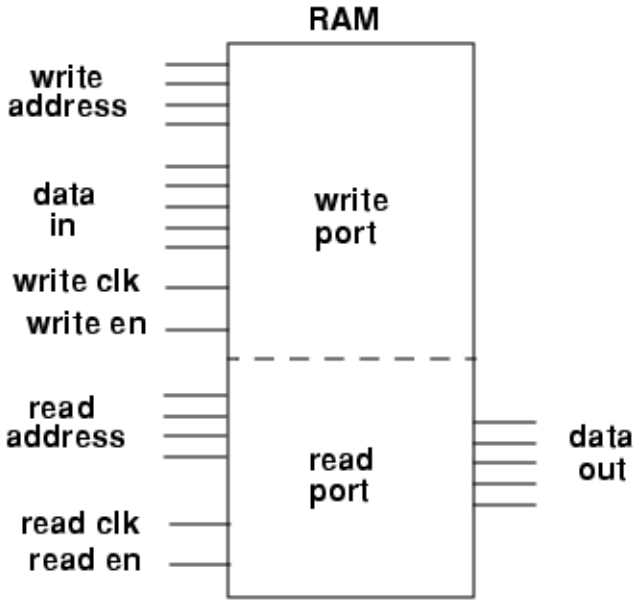
Before you read values from a ROM, the contents of the ROM must be initialized. This is accomplished through the use of a ROM initialization file. This is discussed in in the *TessentScan and ATPG User's Manual*.

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 in the *Tessent Scan and ATPG User'sManual*.

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, A1…AN, and they must also have the same number of data lines, D1…DM. Figure 3-48 shows a block diagram of a RAM.

Figure 3-48. 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 249). Often, oen is not needed to model the RAM of interest, so this input was not shown in Figure 3-48.

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 Siemens EDA 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.

Here is an example of using an environment variable in an init_file pathname.

```
setenv init_dir = test
// Issued in the UNIX O.S. before invoking tessent
// Show that directory test exists where the read_cell_library model
//   exists, and contains rom.init_file
ls test
rom.init_file
// Can use an absolute path instead
setenv init_dir /top_dir/mid_dir/lower_dir/test

// Inside a read_cell_library file
model ROM8X2 (A1,Ren1, D1) (
   input(Ren1 ) ()
   input(A1) (array = 2:0;)
   output(D1) (array = 1:0;)
   (
      data_size = 2;
      address_size = 3;
      read_off = X;
      min_address = 0;
      max_address = 7;
      //setenv init_dir string substituted below to open init_file
      init_file = "$init_dir/rom.init_file";
      primitive = _rom(_read(Ren1,A1,D1));
   )
)
```

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, the tool assumes that the port is always reading. If the read enable is defined, by default it behaves as follows. The tool assumes it to be active high. When the read enable line is active, it places the values of the memory cells associated with the current port address on the data outputs if the address is valid. If the current address is invalid, it sets all outputs of the port 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, it sets all outputs of the port to X. If the read enable line is low (inactive), it sets all outputs of the port 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.

You must ensure the number of address lines in each port are equal to the number specified by the address_size attribute. Order the address lines so that the most significant address lines are given first. Ensure the number of data lines in each port are equal to the number specified by the data_size attributes. Order the data lines so that the first data input line corresponds to the first data output line, and so on. Make sure the data line ordering is 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

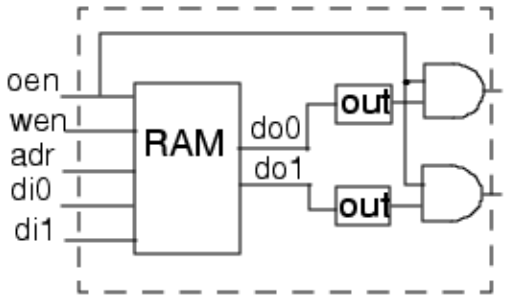
read{w,x,y,z}(oen,rclk,ren,address,out_data)

Each read port of a **_cram** can have up to five pins. The first three are the control pins that are described in the following list:

1. oen - This is the output enable signal that the tool uses to control accessibility of the _cram output. If the signal is high, you can access the output. Otherwise, it disables the output. 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 3-49 shows.

Figure 3-49. 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.

2. 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.

3. ren - This is the read enable, a signal that 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 required 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).

Set and Reset Lines for _ram and _cram

The **_ram** and **_cram** primitives may have a set or reset input that is active high. If the set line is high, the tool sets all the memory cells of the RAM to 1. If the reset line is high, it sets all the memory cells of the RAM to 0. If either the set or reset input is at an X, or if both are high, it sets all the memory cells of the RAM to X. If the set or reset lines are not used, you must still insert the comma delimiters 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 tool loads the memory location specified by the current port address with data present on the data in lines if the address is valid. If the address is invalid, it ignores the write operation. 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 are accessed by the address to X. If the input data and the memory cell value are the same, the memory cell value is not 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, the tool sets those memory cells 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. You must ensure the address lines are ordered such that the most significant address lines are given first. Ensure the number of data lines in each port are equal to the number specified by the data_size attributes. Order the data lines such that the first data in line corresponds to the first data out line, and so on. Also, make sure the data line ordering is 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:

- The tool does not allow static pass-through testing.
- The RAM must successfully pass design rule A1 if you want to use it during ATPG or fault simulation. Otherwise, the tool treats it 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 that 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.

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 Xs the current addresses 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, you must specify the output enable pin in the _read_write port.

The behavior of the port is to allow either read or write in each cycle (not both), but it is not possible to perform any form of pass-through test using the RAM. In the case that multiple RAMs share a common data bus, it is not possible to transfer data from one RAM to another using the bus.

Note

Tessent FastScan and Tessent TestKompress 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 is no danger of creating an incorrect pattern, although a certain amount of pessimism is introduced into the simulation. To support a bidirectional pin, the tool requires exceptional behavior in flattening.

For a read/write port, a read write conflict on the same port is 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 recognizes the case where a RAM port is stable because it has 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

Two 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.

- 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 not 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 enables the tool to perform ATPG for a RAM whose read enable lines are always active.
- 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 is not overwritten by another port at the same time, during the ATPG process.

Chapter 4
Create Tessent Insertion Attributes Using Liberty

This chapter describes how to extract cell information from Liberty for Tessent test tools to be able to insert combinational and sequential cells for test logic. This includes cell_type and pin_function as well as information for nonscan replacement by a scan equivalent cell or scan replacement by a nonscan equivalent cell and stitching the scan cells into a scan chain.

The read_liberty command is used to extract the information.

The tool gathers the cell attributes from read_liberty and populates them onto the read_cell_library model of the same name. This occurs when it needs the populated cell information, which is when you issue the first write_cell_library, get_dft_cell, or set_current_design command. It is advised that you make a separate read_liberty, read_cell_library, write_cell_library run as in the example below, then examine any warnings and other messages produced, and possibly even edit the cell library output by that command. (For example, to move a scan_equivalent to the front of the scan_equivalents list because it only uses the front scan cell to replace a nonscan cell during test insertion, or to move a nonscan_equivalent to the front of the nonscan_equivalents list for the same reason.) If you need to make manual edits, it is advisable to reiterate read_cell_library and write_cell_library, to ensure the manual edits were done correctly. For iterations, omit read_liberty because the cell attribute information from that command is already populated onto the models/cells written out by write_cell_library.

Liberty cells with “dont_use = true;” attributes have “cell_type = prohibited;” populated into the Tessent cell model to prevent their use for test insertion.

See “set_cell_library_options -liberty_ignore_dont_use” for information on how to change the default for a Liberty dont_use cell, if you want to enable the tool to use such cells for test logic insertion.

Example:

```
read_cell_library tessent_cell.tcellib
read_liberty liberty.lib
write_cell_library tessent_cell.tcellib
```

This chapter includes the following topics:

General Limitations

This section describes the general limitations on the extraction of cell information from Liberty that apply to all cells. For more information, see the *Liberty Reference Manual*, which is an Open Source document.

- No Liberty bus group range support.
- Do support Liberty

  ```
  type (<bus_type_name>) { ...}
  ```

  when combined with a bit blasted bus declaration of the form:

  ```
  bus (<bus_name> {
      bus_type:<bus_type_name>;
      pin (<bus_name>[0]){..};
      pin (<bus_name>[1]) {..};
  }
  ```

  with a single bit pin declaration inside the bus group for each bit of the bus.

  For example:

  ```
  library("mylib_io_rcmax") {
    type("myBus_1_0") {
      base_type : "array" ;
      bit_from : 1 ;
      bit_to : 0 ;
      bit_width : 2 ;
      data_type : "bit" ;
      downto : "true" ;
    }
    cell("myCell") {
      bus("drive_select") {
        bus_type : "myBus_1_0" ;
        pin("drive_select[1]") {
          direction : "input" ;
          always_on : false ;
          default_state : "00" ;
          capacitance : 0.003706 ;
        }
        pin("drive_select[0]") {
          direction : "input" ;
          always_on : false ;
          capacitance : 0.003699 ;
        }
      }
      ...
  }
  ```

- No powerdown support.

- No x_function support.

Although not a limitation, the tool assigns “cell_type=prohibited;” for any Liberty cell with the “dont_use=true;” statement inside the cell’s group. This prevents test insertion from using such a cell to implement test logic.

See “set_cell_library_options -liberty_ignore_dont_use” for information on how to change the default for a Liberty dont_use cell, if you want to enable the tool to use such cells for test logic insertion.

Liberty Pad Information

This section describes the information extracted from Liberty related to pad cells.

Pad Limitations

In addition to the General Limitations that apply to all cells, the following limitations apply only to the extraction of pad cell information:

- No differential pad support.
- No programmable pad support.
- No AC pad support.
- Single pad_from_pad. Only one pin inside the Liberty pad cell references the is_pad pin in its equations.

Pad Capabilities

The following Tessent cell port pad_function attributes can be extracted from Liberty:

- pad_pad_io for a bi-directional pad.
- pad_to_io for an output pad.
- pad_from_io for an input pad.
- pad_to_pad for an output or bi-directional pad. pad_data_inv also if inverting.
- pad_from_pad for an input or bi-directional pad. pad_data_inv also if inverting.
- pad_enable_high for an output or bi-directional pad.
- pad_enable_low for an output or bi-directional pad.

The following Tessent cell port special drive attributes can be extracted from Liberty:

- pad_open_drain for an output pad or a bi-directional pad.
- pad_open_source or an output pad or a bi-directional pad.

- pad_pull1 for an output pad or a bi-directional pad.
- pad_pull0 for an output pad or a bi-directional pad.

Liberty Combinational Cell Information

This section describes the information extracted from Liberty related to combinational cells.

Combinational Cell Limitations

In addition to the General Limitations that apply to all cells, the following limitations apply only to the extraction of combinational cell information:

- cell_type = clock_buffer cannot be inferred from Liberty.
- cell_type = clock_inverter cannot be inferred from Liberty.
- cell_type = clock_and cannot be inferred from Liberty.
- cell_type = clock_or cannot be inferred from Liberty.
- cell_type = clock_mux cannot be inferred from Liberty.

Combinational Cell Capabilities

The tool translates a Liberty cell into a Tessent cell library model using the output or bi-directional pin function. Using this model, the simulation_function of the cell can be learned.

From the translated Tessent cell library model, it also assigns the learned simulation_function type as the cell_type. It also assigns any required port function as is always done during learning. This enables test insertion tools to insert the cells where needed.

The tool merges the cell_type and port functions from the Liberty translated model onto the read_cell_library model or read_verilog module of the same name as the Liberty cell name.

Liberty Sequential Cell Information

This section describes the information extracted from Liberty related to sequential cells.

Sequential Cell Limitations

In addition to the General Limitations that apply to all cells, the following limitations apply only to the extraction of sequential cell information:

- No latch_bank support.
- No synchronizer_cell extraction.

- Scan cell must have same pin names as nonscan to become its scan_equivalent.
- Nonscan cell must have same pin names as scan to become its nonscan_equivalent.
- The dff “clocked_on” equation can only be a function of one literal (a single true or complemented variable).

  In all Liberty cells encountered in Liberty source files, the "clocked_on" equation has been an expression of a single literal, such as "CLK" for a posedge clock, or "!CLK" for a negedge clock. When automatically associating scan_equivalents from read_liberty, so that scan replacement can replace a nonscan cell with a system function equivalent scan cell, and when associating nonscan_equivalents from read_liberty so that nonscan replacement can replace a scan cell with a system equivalent nonscan cell, currently only single literal clock equivalence is supported. In the very unlikely event that a clock enable is inside the cell, so the "clocked_on" equation is, for example, "CLK * CLKEN", no scan_equivalents are automatically created for such a nonscan cell (or nonscan_equivalents for a scan cell). You can add a scan_equivalents statement to the nonscan dff model created by write_cell_library after processing of all the read_liberty commands has completed, to enable nonscan cell replacement in a netlist before scan stitching. Similarly, you can add a nonscan_equivalents statement to the scan cell model created by write_cell_library after processing all of the read_liberty commands has completed, to enable scan cell replacement in a netlist before scan stitching (such as in a shift register where every bit has been scanned).
- S-R Latch Liberty Equation Requirement.

  The tool creates a cell model for an S-R latch from Liberty only if clock and data are explicitly tied off to 0. This is not needed for normal read_liberty flows because Tessent flows do not use S-R latches for test insertion. The tool reports a warning message for an S-R latch that does not meet the recognition requirement that the data_in and enable are tied explicitly, as follows :

  ```
  // Warning: Cell 'S_R_Latch' is a Latch cell with no data_in. Cannot build model.
  ```

  Example:

  ```
  cell ("S_R_Latch") {
    latch ("IQ", "IQN") {
    clear_preset_var1 : "H" ;
    clear : !resetb" ;
    clear_preset_var2 : "H" ;
    preset : "!setb" ;
    /* Must also explicity tie off the clock and data to 0 (inactive) */
    data_in : "0" ;
    enable : "0" ;
  }
  ```

- cell_type = clock_dff cannot be inferred from Liberty.

Sequential Cell Capabilities

The tool translates the Liberty cell into a Tessent cell library model using the output or bi-directional pin function. Using this model, it can learn the simulation_function of the cell.

It also uses the dff group and latch group nextstate, preset, and clear equations in the translation to a Tessent cell library model. In addition, it extracts the following test_cell pin functions from Liberty.

- scan_enable or scan_enable_inv
- scan_in
- scan_out or scan_out_inv

From the translated Tessent cell library model, it assigns the learned simulation_function type as the cell_type. Also, it assigns any required port function as is always done during learning. This enables test insertion tools to insert the cells where needed.

The tool merges the cell_type and port functions from the Liberty translated model onto the read_cell_library model or read_verilog module of the same name as the Liberty cell name.

The tool determines the scan_equivalents for a nonscan cell in the following manner. Cells with both a dff group and a test_cell group are scan cells. It initially determines the nonscan cells that a scan cell can replace by an equivalence check of the nonscan cell with the scan cell when the scan cell's scan enable is de-asserted. It uses the Liberty cell area and output pin drives to order the functionally equivalent scan cells for a nonscan cell. This list then becomes the scan_equivalents statement inside the nonscan cell model. The first scan model in the scan_equivalents list inside the nonscan model is the cell that replaces instances of that nonscan model in the netlist, if replacement is done within the Tessent Shell flow. It determines the nonscan_equivalents similarly, and the first nonscan model in the nonscan_equivalents list inside the scan model is the cell that replaces instances of that scan model in the netlist if replacement is done within the Tessent Shell flow.

The "set_cell_library_options -use_liberty_filenames_for_scan_equivalents" command determines whether a scan cell from a different Liberty file than the nonscan cell can become a scan equivalent for the nonscan cell, or a nonscan from a different Liberty file than the scan cell can become a nonscan_equivalent for the scan cell. The default is on, meaning that it cannot become an equivalent. The tool does not place any scan equivalents for which no read_cell_library model exists in the scan_equivalents list for a nonscan cell. Similarly, it does not place any nonscan equivalents for which no read_cell_library model exists in the nonscan_equivalents list for a scan cell.

For a Liberty integrated clock gating cell, the tool can extract cell_type and port functions if the Liberty cell uses either of the following statements:

- clock_gating_integrated_cell : latch_posedge_precontrol;

- clock_gating_integrated_cell : latch_negedge_precontrol;

Additionally, it can create and verify a Tessent cell library model. Typically, it uses a statetable (for details, see Liberty documentation). If Liberty uses a latch group, then use the following statement:

clock_gating_integrated_cell :generic;

For the extraction of a clock gating cell to be useable for test, a pre-control test enable must exist in the cell, and the following four attributes must exist, one on each of the four pins of the cell:

- clock_gate_clock_pin

  // The clock input to the cell

- clock_gate_enable_pin

  // The system enable of the cell, which independently but synchronously enables clocking

- clock_gate_test_pin

  // The test enable of the cell, which independently but synchronously enables clocking

- clock_gate_out_pin

  // The clock output from the cell

Chapter 5
Create Tessent Simulation Models Using LibComp

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, and Tessent TestKompress. You can also use LibComp to convert TetraMax Primitives to Verilog Primitives.

This chapter includes the following topics:

Simulation Model Creation

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 -dof 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
```

How to Find 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 blackboxed):

   ```
   "PARTIALLY TRANSLATED MODULE"
   ```

How to Find Blackboxes with Vectored Outputs

The LibComp tool leaves vectored outputs on blackboxes undriven.

Undriven outputs default to _tiex in ATPG runs. In some cases, partial models with only some outputs blackboxed may result. Blackboxes with vectored outputs are preceded with a “Blackboxed PO” message in the cell library.

Use the following command to display a list of the blackboxed models with vectored outputs in a cell library:

grep "Blackboxed PO." output.atpglib

Reconciling System Verilog reg and Verilog Keyword Compiling Issues

You can encounter Questa SIM compilation and loading errors with LibComp and lcVerify output under certain conditions.

- **SystemVerilog** — Modules using reg as interconnect. LibComp does not support System Verilog syntax or semantics except for using reg as interconnect, which the tool enables 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 lcverify invocation.

Reserved Verilog Keywords

If the Verilog source uses Verilog reserved words (for example, “int” or “do”), then Questa SIM can have compilation and loading errors. In this case, you must use the -no_sv switch so Questa SIM can compile and load.

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 that do not contain simulation library primitives (for example, _MUX), then you must explicitly issue the -sv switch to enable Questa SIM 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 enable 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 defines 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 is 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 5-1. Command Summary

Command	Description
Dofile	Executes the commands contained within the specified file.
Exit	Terminates the application session.
Help	Displays the usage syntax for the specified command.
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 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 Differential Clocks	Changes the default UDP latch translation for a single data input with two clocks.
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.
Set Excessive Pull_delay	Specifies a delay threshold for changing a weak driver with excessive delay to either a _tiex or an _undefined 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 _mux if either one or both consensus terms is missing from the UDP table.

Table 5-1. Command Summary (cont.)

Command	Description
Set System Mode	Changes the tool mode.
Set Undefined Instance	Specifies whether LibComp outputs a simulation model or a blackbox when a Verilog instance references a module name for which there is no defining module in the Verilog source.
Set Verification	Specifies which tool it uses for verification or disables verification.
Set X_from_known Combinational_udp	Specifies whether LibComp outputs a simulation model or a blackbox 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.

Dofile

Scope: All modes

Sequentially runs the commands contained in the specified file.

Usage

DOFile ***filename***

Arguments

- ***filename***

  A required string that specifies the name of the file that contains the commands to run.

Description

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 run 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 running it. If an error is encountered due to any command, the Dofile command stops its running and displays an error message. You can enable the Dofile command to continue regardless of errors with the "set_tcl_shell_options -abort_dofile_on_error off" 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 "How to Find Unsupported Constructs in Partial Models" topic in this manual.

Examples

The following example orderly runs all the commands from the *command_file* file:

```
dofile command_file
```

Exit

Scope: All modes

Terminates the application session and returns to the operating system.

Usage

EXIt

Arguments

None

Description

Use this command for interactive sessions and in dofiles.

Examples

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

Displays the usage syntax for the specified command.

Usage

HELp [*command_name*]

Arguments

- *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, LOAd, SET, or WRIte.

  If you do not supply a *command_name,* the default is to display a list of all the command names.

Description

The Help command provides quick access to either information about a specific command, to a list of commands beginning with at specific key word, or to a list of all the commands.

Run

Scope: Translation mode

Starts the compilation process on all models currently in the compilation list.

Usage

RUN

Arguments

None

Description

This command causes all Verilog modules and primitives that are added by the add models command to be translated. This includes all the Verilog source in files listed on the libcomp invocation line following “libcomp” using the default dofile, which issues the "add models -all" command. When the run command is issued, all Verilog source added is translated into Tessent cell library source format for use in Tessent tools. The run command then causes lcverify to be invoked which in turn invokes ATPG to generate patterns for the translated models, and then a Verilog testbench to be simulated against the original Verilog source with the values predicted by ATPG's simulator using the translated Tessent syntax models. Any commands modifying default translation or verification behavior must be issued before the run command.

Examples

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
```

Set Asynchronous Control_Logic

Scope: Setup mode

Specifies whether output dominance logic is used during the translation of sequential UDP memory modules.

Usage

SET ASynchronous
Control_logic {**-OUTput_dominance_logic**| **-NO_OUTput_dominance_logic**}

Arguments

- **-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.

Description

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 required 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 5-2. 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

Examples

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 Topics

Run

Set BB Outputs

Scope: All modes

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.

Usage

SET BB Outputs [-Tiex| -Float]

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 blackbox.

- -Float

  An optional literal that specifies that each bidi or output bit of a blackboxed model is to be left undriven (floating).

Examples

The following example instructs the tool to leave blackbox output or bidi floating.

```
set bb outputs -float
```

Related Topics

Set Empty_module Outputs

Set Differential Clocks

Scope: Setup mode.

Changes the default UDP latch translation for a single data input with two clocks.

Usage

set differential clocks none

Arguments

- none

  A required switch that changes the default UDP latch translation.

Description

By default, a latch with one data input and two clocks is assumed to be a differential clock pair latch, if either of the clocks can capture the data. To change this default to create an ORed clock model, the Libcomp dofile should contain the command “set differential clocks none” before the run command that causes models to be translated.

Examples

set differential clocks none

Set Empty_module Outputs

Scope: All modes

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.

Usage

SET EMpty_module Outputs [-Float -Tiex]

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.

Description

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* that preserves the ability to match Verilog simulation without modifying the simulation models using the set_tied_signals command in the TK/FS dofile.

Examples

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

Specifies a delay threshold for changing a weak driver with excessive delay to either a _tiex or an _undefined simulation model.

Usage

SET EXcessive Pull_delay *delay_time*

Arguments

- *delay_time*

  A optional non-negative integer specifying weak drivers' excessive delay in time units. The default is 100 time units.

Description

Using this command 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.

Examples

The following example sets the delay to 1000 time units.

```
set excessive pull_delay 1000
```

Set Floating Net_type

Scope: All modes

Specifies how LibComp translates floating nets.

Usage

SET FLoating Net_type
[NONE| X| Z]

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.

Description

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.

Examples

The following example ensures that each net has a _tiex driver if no others.

```
set floating net_type X
```

Set Hold Check

Scope: All modes

Specifies how sequential UDPs with hold check violations are translated.

Usage

SET HOld Check ON[-fix_asynch_fails | -black_box_asynch] | OFf

Arguments

- ON

  A literal that enables asynchronous hold checks. This is the default.

- -fix_asynch_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_asynch

  A switch that outputs a blackbox for models that fail a hold check. Prevents simulation mismatches, but causes fault coverage problems if you use blackboxes 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.

Description

It is common to erroneously specify Verilog UDPs in the rows saying that the state should hold when you omit some UDP input changes.

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 correctly holds 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 (cell library) is created which assumes that is 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 enables the data port(s) to work but goes to X if the erroneously defined asynch is ever asserted. This matches the Verilog and masks 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 blackboxed 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 blackbox 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).

Examples

Example 1

The following example blackboxes asynchronous modules that fail hold check.

```
set hold check on -black_box_asynch
```

Example 2

The following example turns off hold checks.

```
set hold check off
```

Related Topics

Run

Set Instance Portlist

Scope: Setup mode

Specifies naming conventions used for module instances.

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

Prior to the v8.2009_2 release, LibComp and the cell 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, cell 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.

Examples

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

Specifies whether LibComp outputs the model_source statement for every model created.

Usage

SET MOdel Source **ON**|**OFf**

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.

Description

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 that enables those models to be used by pre-v8.2009_3 versions of Tessent TestKompress/Tessent FastScan.

Examples

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
```

Set MUX Nonconsensus_logic

Scope: All modes

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.

Usage

SET MUX NONconsensus_logic
{**EXPLicit**| **NONE | ALL**}

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.

Examples

The following example instructs LibComp to return to pre-v8.2009_4 behavior.

```
SET MUX NONconsensus_logic NONE
```

Set Partial Translation

Scope: All modes

Changes the default behavior from blackboxing modules with constructs that are not understood, to behavior that translates as much of the module that is understood.

Usage

SET PArtial Translation {OFF | ON}

Arguments

- ON

  A required literal that changes the default behavior from blackboxing modules with constructs that are not understood, to behavior that translates as much of the module that is understood. A warning is issued for each construct not understood. Also, above the model when output, a PARTIALLY TRANSLATED MODULE comment displays. You are responsible for ensuring such models are adequate and for adding any additional hardware required. Sometimes, nothing is required because the construct not understood has no critical simulation impact. In rare cases, the missing hardware can drive a bus, and ATPG may not be able to satisfy contention. In such cases, you should add a _tieZ to drive the bus, or add the hardware that was not translated.

- OFF

  The default at tool invocation, so this switch is not needed, but is provided for completeness.

Examples

The following command changes the default behavior from blackboxing modules with constructs that are not to behavior that translates as much of the module that is understood.

```
set partial translation on
```

Set System Mode

Scope: All modes

Specifies whether the tool enters the Setup or Translation mode.

Usage

SET SYstem Mode **Setup**| **Translation**

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.

Examples

The following example puts the tool in Translation mode.

```
set system mode translation
```

Related Topics

Run

Set Undefined Instance

Scope: All modes

Specifies whether LibComp outputs an simulation model or a blackbox when a Verilog instance references a module name for which there is no defining module in the Verilog source.

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 blackbox model when a Verilog instance references a module name for which there is no defining module in the Verilog source.

Description

By default, LibComp outputs a simulation model. You can specify the -black_box option that causes the module to be blackboxed.

When undefined instances are encountered, LibComp sets verification to off (Set Verification -OFf) because Questa SIM fails to load an incompletely defined simulation model and, therefore, verification is not possible.

Examples

The following example specifies to output undefined modules as blackboxes.

```
set undefined instance -blackbox
```

Set Verification

Scope: Setup mode

Enables or disables verification tool usage after library compilation.

Usage

SET VErification
{**FASTscan** | **TESTKompress** | **OFf**}
[-SV | -NO_SV]

Arguments

- **FASTscan**

  A required literal that sets Tessent FastScan as the verification tool. This is the default.

- **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 Questa SIM 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 Questa SIM 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.

Description

Specifies which tool is used for verification or disables the verification of a generated simulation library (cell 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 "Verification of Tessent Simulation Models".

Examples

The following example disables verification during compilation runs.

```
set verification off
```

Related Topics

Dofile

Run

Set X_from_known Combinational_udp

Scope: All modes

Specifies whether LibComp outputs a simulation model or a blackbox for combinational UDPs that output X from a known input combination.

Usage

SET X_from_known
Combinational_udp {atpg_model| black_box}

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 blackbox model for combinational UDPs that output X from some known input combinations.

Description

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.

Examples

The following example defaults to outputting a simulation model:

```
set x_from_known combinational_udp
```

System

Scope: All modes

Executes one operating system command without exiting the currently running application.

Usage

SYStem ***os_command***

Arguments

- ***os_command***

  A required string that specifies any legal operating system command.

Examples

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.

Writes the simulation models created during compilation to a specified filename.

Usage

WRIte LIbrary ***filename***[-Replace]

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.

Examples

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 Topics

Run

Limitations and Examples

This section describes many of the limitations encountered with translating Verilog using LibComp. Examples are given to illustrate how to write a simplified test view form of Verilog, that can then be translated using LibComp. Although LibComp can be surprisingly good at translating UDPs, its limitations in this area are also discussed. On the other hand, the behavioral translation capabilities are quite limited, and the somewhat severe limitations in this area are also discussed.

Transistor Modeling Limitations

The following limitations apply to transistor modeling.

- Tessent simulators do not have bidirectional primitives. LibComp uses a _wire model for a Verilog tran or rtran primitive.

  The LibComp _wire model connects the net at each end, which creates the same simulation result as with the Verilog tran primitive. However, the LibComp _wire model does not insert a _pull for the rtran Verilog primitive, which may cause simulation mismatches. If you need the weakening of a value through the rtran primitive, you should not use the _wire model created by LibComp.

- If a Verilog module has only a tran primitive between two inout ports, the simulation of the resulting LibComp model produces an X on these ports. However, when you instantiate the Verilog module, the LibComp model simulation matches the Verilog results.

  In the following example, the module transistor gets 0% coverage since neither port has a source nor sink connected. However, when the module is connected (module use_transistor and backwards_transistor), the same module produces 100% coverage with no simulation mismatches.

```
`celldefine
module transistor (dst, src);
inout src, dst;
tran (dst, src);
endmodule

module use_transistor (out, in);
input in;
output out;
wire to_tran, from_tran;
not (to_tran, in);
transistor tran1 (from_tran, to_tran);
not (out, from_tran);
endmodule

module backwards_transistor (out, in);
input in;
output out;
not (to_tran, in);
transistor tran1 (to_tran, from_tran);
not (out, from_tran);
endmodule
`endcelldefine

Libcomp tran model :

model transistor
(dst, src)
(
model_source = verilog_module;
inout (dst) ( )
inout (src) ()
(
    primitive = _wire mlc_tran_1 ( src, dst );
)
)
```

The LibComp/Icverify results below show that the LibComp _wire model for the Verilog tran primitive works when instantiated in the use_transistor and backwards_transistor modules. The results for the transistor module demonstrate the coverage limitation of the _wire model without any connections, which does not matter as it is not a realistic usage.

```
Model: transistor
---------------------
Fault Statistics for instance transistor
----------------------------------------
                     #faults         transistor
Fault Classes        (total)         transistor
-------------------- --------------- transistor
FU (full)                  2         transistor
------------------  ---------------- transistor
UU (unused)             2 (100.00%)  transistor
------------------------------------ transistor
Coverage                             transistor
  ------------------                 transistor
  test_coverage               0.00% transistor
  fault_coverage              0.00% transistor
  atpg_effectiveness        100.00% transistor
------------------------------------ transistor

-------------------------------------------------------
Model : backwards_transistor
---------------------
Fault Statistics for instance backwards_transistor
----------------------------------------
Fault Classes              #faults     backwards_transistor
                           (total)     backwards_transistor
---------------------  ---------------- backwards_transistor
  FU (full)                   4        backwards_transistor
  -------------------  ---------------- backwards_transistor
  DS (det_simulation)      4 (100.00%) backwards_transistor
------------------------------------- backwards_transistor
Coverage                              backwards_transistor
  ------------------                  backwards_transistor
  test_coverage              100.00%  backwards_transistor
  fault_coverage             100.00%  backwards_transistor
  atpg_effectiveness         100.00%  backwards_transistor
------------------------------------- backwards_transistor
```

```
-------------------------------------------------------
Model : use_transistor
---------------------
Fault Statistics for instance use_transistor
----------------------------------------
Fault Classes              #faults     use_transistor
                           (total)     use_transistor
---------------------  --------------- use_transistor
  FU (full)                   4        use_transistor
  -------------------  --------------- use_transistor
  DS (det_simulation)      4 (100.00%) use_transistor
------------------------------------- use_transistor
Coverage                               use_transistor
  -------------------                  use_transistor
  test_coverage              100.00%   use_transistor
  fault_coverage             100.00%   use_transistor
  atpg_effectiveness         100.00%   use_transistor
------------------------------------- use_transistor

Verification Summary:
     3 Total Models
ALL PASSED for all patterns.
  All known model output values predicted by ATPG agreed
  with Verilog sim.
```

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 multiplexer UDP into a _mux that has consensus terms in its simulation.

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.

Sequential UDP Translation

Libcomp does not translate sequential UDPs when the clock input dominates the asynchronous inputs.

Typically, the asynchronous inputs dominate the clock input for sequential UDPs. In the rare case where the opposite is true, Libcomp fails to translate the UDP. In such cases, you have to translate the UDP manually or contact Siemens EDA Support for help in translation.

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 that cannot happen in hardware. Because there are some LV flows where an X destroys 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 costs 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 that cannot happen in hardware. If the X's have been properly bounded as required for the LV flows, a lighter simpler Boolean model is sufficient.

UDP With Both Consensus Terms

This is a UDP for a mux that 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

This section describes some 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 that 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 that 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 that 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 Multiplexer (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 (cell library) output for the string "BlackBox". In most cases, when LibComp encounters a UDP that cannot be translated, a blackbox 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

This is an example of a latch with 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 Set/Reset Logic Gated Off By Scan Enable

This is a Verilog example of a MUX Scan cell.

```
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 asynchronious 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
```

DFF Example

This is a Verilog example of a DFF.

```
//
// 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

This is a Verilog example of a D Latch.

```
//
// 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
```

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 "How to Find Unsupported Constructs in Partial Models".

Behavioral Constructs

The LibComp tool does not support Verilog constructs with certain behavioral constructs.

These are as follows:

- Case statements
- Tasks
- Functions
- Initial blocks
- Loops
- Always blocks except very simple DFF, Latch, or Ram behavioral models

  For a DFF, a single always block such as:

  ```
  always @ (posedge clk)  Q = D;
  // For a reg Q.  Q and D can be vectors of same size.
  ```

  For a Latch, a single always block such as:

  ```
  always @) (clk or D)  if (clk) Q = D;  // For an activeHI latch.
  ```

  For simple memories, see the RAM Examples.

Structural Constructs

The LibComp tool does not support Verilog constructs with certain structural constructs.

These are as follows:

- 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

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, and so on 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 that can cause pulls to activate, 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 does not know that exercising the pins can cause false simulation mismatches, and therefore results 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” and so on. 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 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 enable 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 that indicate legal differential (or other operation modes) can exist. Such inputs are often only used to cause X output values 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 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 inouts (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 that 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” is 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 constrains 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 *Tessent Shell Reference Manual*.

I/O Pad Code Example

This is a Verilog example of an I/O pad.

```
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 enables it to produce
  //   and measure Z values at PAD.
  //   This is undesireable 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).
  //
  //  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 memory limitations are described in this section.

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 or equal to 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 blackboxed. 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 does not translate a memory with a single address line.

$readmemh and $readmemb

LibComp translates files containing $readmemh or $readmemb.

The translation occurs 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

This is a Verilog example of a ROM.

```
`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 contains 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

This section describes Verilog examples of various types of RAMs.

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 Tristate 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 writes-through 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, 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 Level Ports With Separate Clocks and Write-Thru

```
//
// level sensitive RAM with constantly enabled READ port
//
module ram512x11 (addr, din, wen, dout);

  parameter databits = 11;
  parameter addrbits = 9;
  parameter addrmax = (1<<addrbits)-1;

  input wen;
  input [addrbits-1:0] addr;
  input [databits-1:0] din;

  output [databits-1:0] dout;
  reg [databits-1:0] dout;

  reg [databits-1:0] mymem [0:addrmax];

  supply1 ren;
  event WRITE_OP;

  always @ (wen or addr or din) if (wen) begin
    mymem[addr] = din;
    #0; ->WRITE_OP;
  end

  always @ (ren or addr or WRITE_OP)
    if (ren) dout = mymem[addr];

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 LO).
//   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
        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
  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,  // Datat 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
        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
        else begin
          Mem[A_1] <= (Mem[A_1] & web_1) | (Din_1 & ~web_1);
        end
      end
    end
  end

  assign Dout_1 =  (bypass_1) ? Din_1 : Dout_1_reg;

endmodule
```

Positional Parameter Override Usage to Modify the Previous RAM Model

```
// Rather than create a different module for each different data 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
// uniquification, 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 Holds in-memory data when not writing.
      _write { , , } (ck_1, mlc_and_5[71], A_1, Din_1[71]),
      // Following write port Holds in-memory data when not writing.
      _write { , , } (ck_0, mlc_and_6[71], A_0, Din_0[71]),
      // Following read port Holds output data after reading.
      _read { ,H,H,H} ( , ck_1, mlc_and_3, A_1, Qreg_1[71]),
      // Following read port Holds 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 Holds in-memory data when not writing.
        _write { , , } (ck_1, mlc_and_5[7], A_1, Din_1[7]),
        // Following write port Holds in-memory data when not writing.
        _write { , , } (ck_0, mlc_and_6[7], A_0, Din_0[7]),
        // Following read port Holds output data after reading.
        _read { ,H,H,H} ( , ck_1, mlc_and_3, A_1, Qreg_1[7]),
        // Following read port Holds 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
  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
        else begin
          Mem[A_1] <= (Mem[A_1] & web_1_int) | (Din_1 & ~web_1_int);
        end
      end
    end
  end

  assign Dout_1 =  (bypass_1) ? Din_1 : Dout_1_reg;

endmodule
```

How to Create and Use Shared Submodels Using Libcomp

This section describes how you can create several files of Verilog models, using Verilog `celldefine modules and UDPs to define the lower levels of hierarchy in multiple libraries. To do this, translate the Verilog sources by providing the file with the upper modules as well as the shared Verilog files, and get a separate Tessent Cell Library file from each translation.

Alternatively, you can create a single translation of the lower-level files, and provide those files as well as the appropriate higher-level module translation files in a subsequent read_cell_library for downstream Tessent tools to use. To use those cell libraries, however, you must remember to provide both sets of files when using the downstream flows.

Assume that there are two higher level module files (or sets of files or directories), high_level_1.v and high_level_2.v, as well as a lower level directory low_level_dir containing the supporting Verilog source files to be translated.

First, invoke **libcomp** to translate the low_level_dir source:

```
libcomp low_level_dir -dofile -log logfile_name
// No filename following -dofile uses the recommended default dofile.
```

Once the resulting file provides adequate coverage and produces no mismatches during Verilog simulation, rename the libcomp.atpglib translated output file to low_level.atpglib.

When translating each of the higher level Verilog source files using libcomp, you must provide the Verilog source and the translated files to enable Verilog simulation verification of the translated higher level files.

```
libcomp high_level_1.v -verify_model_file low_level.atpglib \
-verify_verilog_directory low_level_dir -dofile -log ..
```

Once you finish translating, rename libcomp.atpglib to high_level_1.atpglib. This can be a set of files or directories separated by spaces after “libcomp”. For example:

```
libcomp high_level_1a high_level_1b .. -verify_model_file \
low_level.atpglib -verify_verilog_directory low_level_dir -dofile ..
```

Continue translating each high level file (or set of files/directories). For example:

```
libcomp high_level_2.v -verify_model_file low_level.atpglib \
-verify_verilog_directory low_level_dir -dofile -log ..
```

When you finish, rename libcomp.atpglib to high_level_2.atpglib. Again, this can be a set of files or directories specified with space separation after “libcomp”. For example:

```
libcomp high_level_2a high_level_2b .. -verify_model_file \
  low_level.atpglib -verify_verilog_directory low_level_dir -dofile ..
```

Chapter 6
Verification of 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

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.

Tip

: You should verify functionality of the simulation models when you generate the libraries with LibComp (see "Create Tessent Simulation Models Using LibComp"), 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 Questa SIM 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.

Specify Which Tool Performs Verification

By default, Tessent FastScan performs the verification of Tessent cell libraries.

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 Tessent TestKompress.
- **lcVerify** — To use Tessent FastScan for verification, use the *lcVerify* command from Linux/UNIX shell. You specify Tessent TestKompress during lcVerify invocation. Tessent FastScan is the default.

Verification Prerequisites

This section describes the prerequisites that are required for verification.

These are:

- Verilog source library must be available.
- Tessent FastScan or other specified verification tool must be available.
- Questa SIM 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.

How to Run 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 Cell Library using Invocation :
://  $Tessent_Tree_Path/bin/lcVerify -no_scan_rams -no_atpg_prims
//   tessent.mtCellLib
//  design_verify.v
```

How to Verify 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.

How to Interpret the Verification Results

The verification process produces the results files in the simulation library parent directory.

These files are:

- **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 Questa SIM
  - DRC violations
  - Fault and test coverage
- **sim.log** — Full transcript of the Tessent FastScan and Questa SIM runs
- **transcript** — Transcript of the Questa SIM 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
----------------------------------------
                          #faults   #faults
fault class               (coll.)   (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
----------------------------------------
                           #faults  #faults    almost_xor_dff_no_controls
fault class                (coll.)  (total)    almost_xor_dff_no_controls
-----------------------    -------  -------    almost_xor_dff_no_controls
FU  (full)                       6        6    almost_xor_dff_no_controls
-----------------------    -------  -------    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
-----------------------   -------   -------    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
---------------------------------------------- almost_xor_dff_no_controls

-----------------------------------------------------------
Model : nonsense_model
--------------------
Fault Statistics for instance nonsense_model
----------------------------------------
                        #faults   #faults       nonsense_model
```

```
fault class              (coll.)    (total)       nonsense_model
------------------------ -------    -------       nonsense_model
FU (full)                      8          8       nonsense_model
------------------------ -------    -------       nonsense_model
UU (unused)                    6          6       nonsense_model
AU (atpg_untestable)           2          2       nonsense_model
------------------------ -------    -------       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
-----------------------------------------       nonsense_model

------------------------------------------------------------
Model : xor_dff_no_controls
---------------------
Fault Statistics for instance xor_dff_no_controls
-----------------------------------------
                         #faults    #faults       xor_dff_no_controls
fault class              (coll.)    (total)       xor_dff_no_controls
------------------------ -------    -------       xor_dff_no_controls
FU (full)                      8          8       xor_dff_no_controls
------------------------ -------    -------       xor_dff_no_controls
DS (det_simulation)            7          7       xor_dff_no_controls
DI (det_implication)           1          1       xor_dff_no_controls
------------------------ -------    -------       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
-----------------------------------------       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 cell 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
```

How to Debug 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 6-1. 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.
Questa SIM 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 sim.log file and typically contains enough information (line numbers and brief descriptions of errors) for you to start debugging the Verilog.
The Questa SIM 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 sim.log file for debugging information. Check the vlog transcript to ensure the model(s) compiled without errors; compile errors 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 add_clocks definition for the clock input or a missing add_input_constraints for a notifier input in the fastscan.do.cat file. Correct the definition in the fastscan.do.cat file and rerun verification.
Sequential Verilog UDP seems to be a valid Latch or DFF, but LibComp blackboxes 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.

How to Re-simulate Verilog Only

To troubleshoot Verilog simulation issues, you can simulate just the Verilog portion of the verification with the <*Tessent_Tree_Path*>/bin/ script.

The run_verify script contains the commands and arguments to perform just the Questa SIM portion of verification.

See the following topics for more information on using run_veify to simulate the Verilog source library.

Prerequisites for Simulating Verilog Only

This section describes the prerequisites that must be satisfied before you can use run_verify to simulate the Verilog source library.

- Questa SIM must be available.
- lcVerify must be run initially on the Verilog source and simulation models. For more information, see "How to Assess 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.

How to Simulate 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 reinvoke 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 (#).

How to Fix 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.

How to Improve 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.

How to Troubleshoot One Model at a Time

As an aid in troubleshooting, you can 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.

How to Assess the Impact of Low Coverage

If you cannot raise coverage for a particular model in your library, use the Linux 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.

How to Locate 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, that 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
mxdprbsb1qb
mxdprbsb2qb
```

How to Re-run 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 (#).

How to Model for Optimal Test Coverage

Typically, you want the test coverage to be as a 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:

Ignored or Blackboxed Models

Due to the variety of design configurations possible with UDPs, there are UDPs that LibComp cannot process.

Because blackbox 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 “How to Define Cell Information”.

How to Anticipate 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 off state you want on the input to the internal instance after passing through the logic.

How to Verify Pad Models

Typically Pad Models have low coverage even when the model is good. This section describes how to verify that the required functionality exists in the translated model.

IO pads cannot be translated completely by LibComp for digital test, because they are analog in nature, and all sorts of dynamic DRCs are included in the Verilog modules.The main thing to know about using Libcomp for IO pads is:

- Use the fault_coverage_... files.
- Use the au_uc_uo_faults.libcomp file.

For IO pads, the only thing that matters for digital test is: "do they Input to the core and do they Output from the core" -- IO is the important feature for digital test, not the coverage of the analog parts, or modeling of the complex parts.

LcVerify leaves behind fault coverage files, one per decade of coverage. Each file has the names of the models with coverage in that decade. These filenames start with "fault_coverage_".

For example:

```
fault_coverage_10_to_20_percent_models
fault_coverage_20_to_30_percent_models
fault_coverage_30_to_40_percent_models
```

Often, low coverage models have no pins that connect to/from the core, and such models are unimportant for digital test.Assume that the 30_to_40_percent file lists a model named "EXAMPLE_PAD_MODEL" and has connections to or from the core. In such a case, the "au_uc_uo_faults.libcomp" file generated when LcVerify runs is crucial. Note that LibComp automatically invokes LcVerify. Here is an excerpt from that module (from the Verilog source):

```
module  EXAMPLE_PAD_MODEL (I, CK, PADP, PADN, OE_P, OE_N, O_P, O_N);
 input  I;    // data input from core
 input  CK;   // data input from core
 inout PADP;  // PAD: positive
 inout PADN;  // PADP: Negative
 input OE_P;  // active high output enable for PADP
 input OE_N;  // active high output enable for PADN
 output O_P;  // data input from PADP to core
 output O_N;  // data input from PADN to core
```

Typically, the only thing that matters is the IOs going to/from the core.Inputs "I", "CK" are of interest. Outputs "O_P" and "O_N" are of interest. This has differential inputs from outside the chip. If you get coverage of those pins, the IO pad does I's and O's and give good results for testing the core. Parametric tests are for testing the rest of the IO pad. Digital scan based testing is not used to test the rest of the IO pad.

Searching in the au_uo_uc_faults file for the module name "EXAMPLE_PAD_MODEL" produces the following:

```
stssled03[1303] grep EXAMPLE_PAD_MODEL au_uc_uo_faults.libcomp
     0,  AU, "/EXAMPLE_PAD_MODEL/OE_P";
     0,  EQ, "/EXAMPLE_PAD_MODEL__OE_P";
     1,  EQ, "/EXAMPLE_PAD_MODEL__OE_P";
     0,  AU, "/EEXAMPLE_PAD_MODEL/OE_N";
     0,  EQ, "/EXAMPLE_PAD_MODEL__OE_N";
     1,  AU, "/EXAMPLE_PAD_MODEL__OE_N";
     1,  EQ, "/EXAMPLE_PAD_MODEL/OE_N";
```

There is no module pin ending in “/I” or “__I”. That means the output from the core is working, and the core is observable at this pad, because /I comes into the pad from the core, then to the tester. Similarly, there is nothing ending in “/CK” or “__CK” so that is also working. Finally, the differential inputs going to the core (that go out of the pad to the core, and so are core module inputs) are also working. We know that because there is no “/O_P” or “__O_P” in the *au_uo_uc_faults* file, and also no “O_N” or “__O_N”. So both work.The names ending in “__<pinname>” are the fake top level PIs and POs we create to instantiate the pad as an instance. The names ending in “/<pinname>” are the actual pad pins that the fake pins connect to. They are always EQ (EQuivalent). There is no way to test one for a sa0 or sa1 and not test the other. Similarly, if one is untestable, the other must also be untestable. Typically, every model in the fault coverage files above 40% have the input to the core and output from the core working when the *au_uo_uc_faults* file is checked.

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
lcverify	Invokes lcVerify and verifies the cell library models generated by LibComp.
libcomp	Invokes the LibComp utility to compile a Verilog library into cell library format.

lcverify

Runs lcVerify and verifies the cell library models generated by LibComp

Usage

lcverify [-fastscan | -testkompress] ***cell_library_path verilog_library_path***
[-model *model_name*]
[+*verilog_plus_args*]
[-vsim_options *vsim_options*]
[-atpg_prims | -no_atpg_prims]
[-lv_generics | -no_lv_generics]
[-atpg_only]
[-save_vsim | -no_save_vsim]
[-scan_rams | -no_scan_rams]
[-sv | -no_sv]
[-ignore_startup_file]
[-help]

Arguments

- -fastscan | -testkompress

  An optional literal that determines which tool generates the test patterns and simulates the models. Tessent FastScan is the default.

- ***cell_library_path***

  A required string that specifies a file or directory containing the cell library models to verify.

- ***verilog_library_path***

  A required string that specifies a file or directory containing the source Verilog modules to verify the cell library models against.

- -model *model_name*

  An optional switch and string pair that specifies verification of a single model in a cell library. For additional details, refer to "Verification of Tessent Simulation Models".

- +*verilog_plus_args*

  An optional list of options that are passed verbatim to vlog compilations. For example, +define+view1+view2 would cause ifdef view1 and ifdef view2 source to be compiled. Must not follow -vsim_options without intervening lcverify invocation minus argument switch.

- -vsim_options *vsim_options*

  An optional switch and string pair that specifies all of the options to be handed to vsim. For example, it must include '-voptargs=+acc' to cause vsim to be invoked in the same way it would be invoked without the lcverify -vsim_options. It must be followed by end of

invocation line, or another lcverify invocation minus argument switch to terminate the gathering of vsim_option switches for the vsim invocation.

- -atpg_prims

  An optional switch (default) that enables Questa SIM to simulate Verilog with TMax or Siemens EDA cell primitives referenced as instances in the Verilog.

- -no_atpg_prims

  An optional switch that disables the use of the cell primitives definition file to verify cell model libraries with cell primitives. By default, the tool uses the cell primitives definition file 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 Questa SIM.

  Caution

  This switch may prevent Questa SIM from simulating the Verilog modules. You should either remove the cell primitives from the existing libraries or use the script located in <*Tessent_Tree_Path*>*/lib/tmax_to_verilog.pl* to convert cell primitives to Verilog primitives. For more information, refer to “Simulation Model Creation”.

- -lv_generics

  An optional switch that enables Questa SIM to simulate Verilog with LV leaves referenced as instances in the Verilog.

- -no_lv_generics

  An optional switch that should be used if your library has no LV leaf modules (with the names such as LV_MUX.)

- -atpg_only

  An optional switch that should be used to obtain ATPG coverages, but skip Verilog simulation.

- -save_vsim

  An optional switch that prevents deleting the files that enable rerunning Questa SIM without rerunning Tessent FastScan (without reinvoking lcverify). This switch also saves *pat*.ascii.

- -no_save_vsim

  An optional switch that deletes the files that enable rerunning Questa SIM without rerunning FastScan (without reinvoking lcverify). Also deletes ATPG patterns file pat.ascii.

- -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.

- -no_scan_rams

  An optional switch that specifies that even if RAMs exist, do not create RAM scan chains or test procedure file when verifying a library with RAMs. Uses older style PI/PO ram connections.

- -sv

  An optional switch that invokes Questa SIM with System Verilog compilation for .v files. Required when a reg is used as interconnect (such as in a port list). Refer to "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. Refer to "Reconciling System Verilog reg and Verilog Keyword Compiling Issues".

- -ignore_startup_file

  An optional switch that instructs the tool to not read the .tessent_startup file during invocation.

  If a .tessent_startup file exists where the tool is invoked, and the -ignore_startup_file switch is not used, a Warning is output to the transcript and also to sim.log file saying that a .tessent_startup file exists, and is used by ATPG. Invoke the tool with the -ignore_startup_file switch to prevent its use. If no .tessent_startup file exists, or you invoke the tool with the -ignore_startup_file switch, no Warning is issued.

- -help

  An optional switch that displays a description of all the lcverify invocation switches.

Description

lcVerify uses Tessent FastScan to generate test patterns and simulate the models; it also uses Questa SIM to simulate the Verilog source modules and then compares the simulated values to ensure models function as intended. You must generate cell models with a matching version of LibComp.

By default, lcVerify runs automatically from libcomp as it generates the models. For more information, refer to "Verification of Tessent Simulation Models".

Examples

The following example invokes lcVerify in stand-alone mode to verify the *lib10.atpg* cell library with the *lib10.v* Verilog source library.

```
<Tessent_Tree_Path>/bin/lcverify lib10.cell lib10.v
```

libcomp

Runs the LibComp utility to compile a Verilog library into cell library format.

Usage

libcomp *library_path…*{-DOfile[*dofile_name*]} [+*vlog_directive* ...] [-NO_SCAN_rams]
[-TESTkompress] [-LOgfile *logfile_name*[-Replace]]
[-SV | -NO_SV] [-Help| -MANual | -VERSion | -Usage] [-NO_DEFine_simple_views]
[-ignore_startup_file]

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.

- -TESTkompress

  An optional switch that sets LibComp to use Tessent TestKompress when verifying cell 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 Questa SIM 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 Questa SIM 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 Questa SIM. 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.

- -ignore_startup_file

  An optional switch that instructs the tool to not read the .tessent_startup file during invocation.

  If a .tessent_startup file exists where the tool is invoked, and the -ignore_startup_file switch is not used, a Warning is output to the transcript and also to sim.log file saying that a .tessent_startup file exists, and is used by ATPG. Invoke the tool with the -ignore_startup_file switch to prevent its use. If no .tessent_startup file exists, or you invoke the tool with the -ignore_startup_file switch, no Warning is issued.

Description

Invokes the LibComp utility to compile a Verilog library into cell library format.

If no arguments are specified, LibComp invokes in interactive mode.

For more information, see "Create Tessent Simulation Models Using LibComp".

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

TESSENT	lv_rtl_primitives	functional
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 want to retain using the *+vlog_directive* switch documented below.

Examples

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 Questa SIM 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 cell library syntax.

This appendix describes the attributes in the Tessent cell library and maps the LV and cell library attributes to the Tessent cell library attributes.

Tessent Pin Function and Pad_Function Attributes

The following table lists each Tessent pin functions attribute, the tools that use it, and any dependencies.

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

Table A-1. Tessent Pin Function Attributes (cont.)

Attribute	Used by Tools	Required by
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
mux_in5	Tessent Scan	cell_type = mux
mux_in6	Tessent Scan	cell_type = mux
mux_in7	Tessent Scan	cell_type = mux
mux_out	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.
posedge_clock	LV Flow, Tessent Scan	cell_type = dff/latch/scan_cell..
power_isolation_internal	LV Flow	None

Table A-1. Tessent Pin Function Attributes (cont.)

Attribute	Used by Tools	Required by
power_isolation_external	LV Flow	None
retention_enable	Tessent	Retention cell
retention_enable_inv	Tessent	Retention cell
scan_enable	LV Flow, Tessent Scan	Scan cell
scan_enable_inv	LV Flow, Tessent Scan	Scan cell
scan_in	LV Flow, Tessent Scan	Scan cell
scan_out	LV Flow, Tessent Scan	Scan cell
scan_out_inv	LV Flow, Tessent Scan	Scan cell
scan_out_no_polarity	LV Flow, Tessent Scan	Scan cell From old scan_definition scan_out (had no polarity)
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

The following table lists each Tessent pin pad_functions attribute, the tools that use it, and any dependencies.

Table A-2. Tessent Pin Pad_Function Attributes

Attribute	Used by Tools	Required by
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

Table A-2. Tessent Pin Pad_Function Attributes (cont.)

Attribute	Used by Tools	Required by
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 *Note: Deprecated.	Boundary Scan	AC test
pad_init_posedge_clock_dot6	Boundary Scan	AC test
pad_init_negedge_clock_dot6	Boundary Scan	AC test
pad_init_enable_high_dot6	Boundary Scan	AC test
pad_init_enable_low_dot6	Boundary Scan	AC test
pad_init_data_dot6	Boundary Scan	AC test
pad_init_data_inv_dot6	Boundary Scan	AC test
pad_input_enable_low	Boundary Scan	None
pad_input_enable_high	Boundary Scan	None
pad_keep_tracing	Boundary Scan	None
pad_nonjtag	Boundary Scan	None
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

Table A-2. Tessent Pin Pad_Function Attributes (cont.)

Attribute	Used by Tools	Required by
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
pad_two_state_output_enable_high	Boundary Scan	None
pad_two_state_output_enable_low	Boundary Scan	None

Tessent Pin Special Drive and Pull Drive Attributes

The following table lists each Tessent pin special drive attribute, the tools that use it, and any dependencies.

Table A-3. Tessent Pin Special Drive Attributes

Attribute	Used by Tools	Required by
pad_open_drain	Boundary Scan	None
pad_open_source	Boundary Scan	None

The following table lists each Tessent pin pull drive attribute, the tools that use it, and any dependencies.

Table A-4. Tessent Pin PullDrive Attributes

Attribute	Used by Tools	Required by
pad_pull0	Boundary Scan	None
pad_pull1	Boundary Scan	None

Tessent Pin/Port Undriven Input State Attributes

The following table lists each Tessent pin/port special_undriven_state attribute, the tools that use it, and any dependencies.

- Only one undriven_input_state pin attribute can be declared per mode. You can have one in each mode, or one in some modes only, or none.
- The attribute declared in a mode can only be on an input IO (pad_from_io) or bidi IO (pad_pad_io) pin.
- When declared on bidi IO (pad_pad_io), only “pad_input_open0” or “pad_input_open1” can be declared.

IEEE 1149.1-2013 Standard Specification of Undriven Input State Attributes

```
<input or disable spec> ::= <input spec> | <disable spec>
<input spec> :: = EXTERN0 | EXTERN1 | PULL0 | PULL1 | OPEN0 |
OPEN1 | KEEPER | OPENX
```

B.8.14.3.8 <input spec> element

For a <cell info> with a <function> **INPUT** or **CLOCK**, the <port ID> element is the name of a signal received into the component. The <input spec> is required, and the value specifies the behavior of the receiving circuits when that signal is not driven. For example, the source of the signal on the board may be disabled, the connection of the pin to the board may be open due to a defect, or the pin may not be connected on the board, or the pin map for the package may mark the pin as **OPEN**.

Table A-5. Tessent Pin/Port Special Undriven State Attributes

Attribute	Used by Tools	Required by
pad_input_keeper	Boundary Scan	See IEEE Spec. above
pad_input_pull0	Boundary Scan	See IEEE Spec. above
pad_input_pull1	Boundary Scan	See IEEE Spec. above
pad_input_extern0	Boundary Scan	See IEEE Spec. above
pad_input_extern1	Boundary Scan	See IEEE Spec. above
pad_input_open0	Boundary Scan	See IEEE Spec. above
pad_input_open1	Boundary Scan	See IEEE Spec. above
pad_input_openx	Boundary Scan	See IEEE Spec. above

Cell Library Mappings

The following table maps the LV Flow *cell.lib* attributes and syntax to the new Tessent cell library syntax and provides any additional needed description.

Table A-6. 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-6. 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 *clock_*. 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> ;

Table A-6. cell.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
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.
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

The following table 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-7. 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 =	

Table A-7. scang.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
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;
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	

Table A-7. scang.lib Attributes Mapped to Tessent Cell Library Attributes

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
scanoutnot	scan_out_inv	
clockenable	clock_enable	LogicVision internal.
input0	mux_in0	
input1	mux_in1	
select	mux_select	
open	unused	
clockn	clockn	
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

The following table 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-8. 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.

Table A-8. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
<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. For example, my_bs_in_cell = input_cell(...);
O	out_cell	Boundary scan cell class. For example, my_bs_out_cell = output_cell(...);
IO	inout_cell	Boundary scan cell class. For example, my_bs_io_cell = inout_cell(...);
EN	enable_cell	Boundary scan cell class. For example, 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.

Table A-8. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
II	inv_in	Boundary scan cell subclass. Inverted input pad (functional path).
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	Disabled scan cell subclass. IO class. Unavailable 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-8. 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 *model* 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{}} that 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-8. 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{...}}} that 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.
twoStateOutputEnableHigh	pad_two_state_output_enable_high	Pin attribute
twoStateOutputEnableLow	pad_two_state_output_enable_low	Pin 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.

Table A-8. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
padIOInv	pad_pad_io_inv	For second pin of differential pair.
fromIOInv	pad_from_io_inv	For second pin of differential pair.
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. Deprecated. Replaced by initPosedgeClk, initNegedgeClk, initEnableHigh, or initEnableLow.
initPosedgeClk	pad_init_posedge_clock_dot6	1149.6. Replaces InitClk for posedge clocks.
initNegedgeClk	pad_init_negedge_clock_dot6	1149.6 Replaces InitClk for negedge clocks.
initEnableHigh	pad_init_enable_high_dot6	1149.6 Replaces InitClk for active High clocks.
initEnableLow	pad_init_enable_low_dot6	1149.6 Replaces InitClk for active Low clocks.
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.

Table A-8. pad.lib Attributes Mapped to Tessent Cell Library Attributes (cont.)

Tessent LV Flow Syntax	New Tessent Cell Library Syntax	Description and Comments
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 Old Cell Library Mappings

The following table maps the old Tessent Scan ATPG libraries attributes and syntax to the new Tessent Cell library syntax and provides any additional needed description.

Table A-9. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Old Tessent Scan Syntax	New Tessent Cell Library Syntax	Description and Comments
scan_definition()	Omitted.	Most moved to pin attributes; remaining are model level statements or consolidated.
length = integer;	scan_length = integer;	Defined inside model: “model model_name() (*defined here*)” Required if > 1 bit shift path in cell.

Table A-9. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Old Tessent Scan Syntax	New Tessent Cell Library Syntax	Description and Comments
nonscan_model = model_name (list_of_pins); maps to both:	nonscan_model = model_name;	If nonscan pin names are the same on the scan model, the (list_of_pins) can be omitted.
	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.
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 *type* name for cell_type= INV;
cell_type = BUF	cell_type = buffer;	New *type* name for cell_type= BUF;
cell_type = AND	cell_type = and;	No change to *type* name for cell_type= and;
cell_type = NAND	cell_type = nand;	No change to *type* name.
cell_type = OR	cell_type = or;	No change to *type* name.
cell_type = NOR	cell_type = nor;	No change to *type* name.

Table A-9. Tessent Scan Attributes Mapped to Tessent Cell Library Attributes

Old Tessent Scan Syntax	New Tessent Cell Library Syntax	Description and Comments
cell_type = XOR	cell_type = xor;	No change to *type* 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;	
cell_type = SCANCELL < >	Obsolete.	Now must explicitly declare as cell_type = scan_cell.

Appendix B
Getting Help

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 Siemens EDA Support.

The Tessent Documentation System

At the center of the documentation system is the InfoHub that supports both PDF and HTML content. From the InfoHub, you can access all locally installed product documentation, system administration documentation, videos, and tutorials. For users who want to use PDF, you have a PDF bookcase file that provides access to all the installed PDF files.

Note

See New Cloud-Based Documentation Available in the 2024.1 Release for details of an upcoming change to the Tessent documentation system.

For information on defining default HTML browsers, setting up browser options, and setting the default PDF viewer, refer to the *Siemens® Software and Mentor® Documentation System* manual.

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.
- **File System —** Access the Tessent InfoHub or PDF bookcase directly from your file system, without invoking a Tessent tool. For example:

  HTML:

  ```
  firefox <software_release_tree>/doc/infohubs/index.html
  ```

  PDF:

  ```
  acroread <software_release_tree>/doc/pdfdocs/_tessent_pdf_qref.pdf
  ```

- **Application Online Help —** You can get contextual online help within most Tessent tools by using the “help -manual” tool command. For example:

  > help dofile -manual

  This command opens the appropriate reference manual at the “dofile” command description.

New Cloud-Based Documentation Available in the 2024.1 Release

Siemens EDA is excited to announce that with the upcoming release of Tessent 2024.1, we are providing an improved method to access your Siemens EDA product documentation. The default option will now serve your product documentation from the Support Center, giving you immediate access to the latest, release-specific documentation and eliminating the current requirement to install product documentation as part of the local software installation.

For easy access, we offer the option of viewing Support Center documentation using a proxy server on your network. This server also removes the need for your users to have a Support Center account or to log into Support Center to view documentation.

Siemens EDA understands that some customers use our products on restricted networks without internet access. For those customers, we are pleased to offer the option to download and set up the Siemens Documentation Server to view the documentation package locally on that network.

Note

The Siemens EDA documentation InfoHub will be deprecated as part of this transition. We will provide more information on this new documentation system as part of the Tessent 2024.1 release.

We are committed to providing you with the best experience possible while using Siemens EDA products, and we are confident that this change will enhance your access to product documentation.

Global Customer Support and Success

A support contract with Siemens EDA is a valuable investment in your organization’s success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

https://support.sw.siemens.com

If your site is under a current support contract, but you do not have a Support Center login, register here:

https://support.sw.siemens.com/register

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.