cadence®

Encounter® Test: Guide 1: Models

Product Version 12.1.101
February 2013

Printed in the United States of America.

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

Contents

A
Modeling Logic Structures and Attributes

List of Figures

Preface

About Encounter Test and Diagnostics

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

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

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

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

Typographic and Syntax Conventions

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

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

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

- Variables appear in Courier italic type.

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

- Optional arguments are enclosed in brackets.

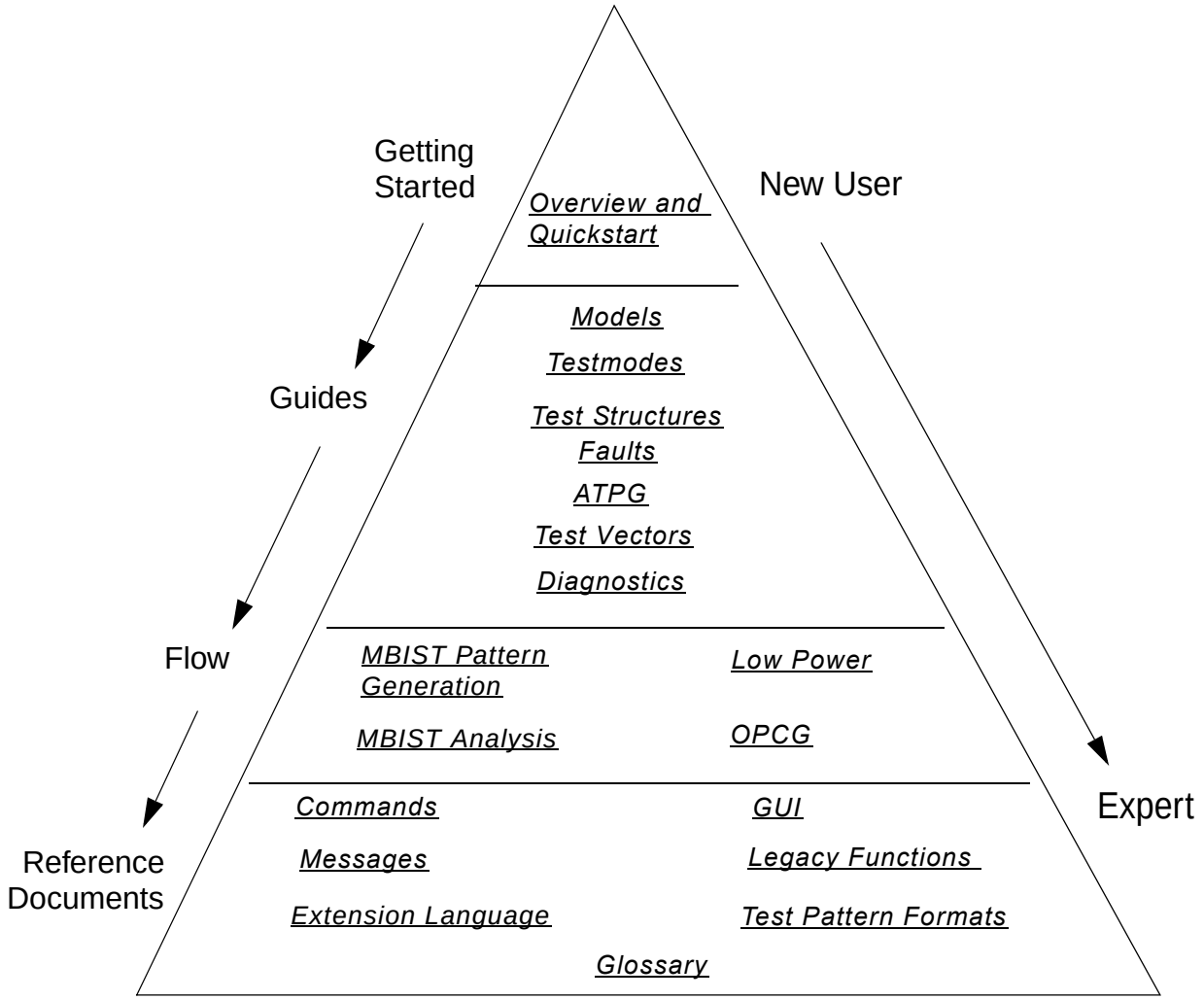
Example: `[simulation=gp|hsscan]`

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

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

Encounter Test Documentation Roadmap

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



Getting Help for Encounter Test and Diagnostics

Use the following methods to obtain help information:

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

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

Refer to the following in the *Graphical User Interface Reference* for additional details:

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

Contacting Customer Service

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

- Cadence Online Customer Support

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

- Cadence Customer Response Center (CRC)

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

- IBM Field Design Center Customers

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

Encounter Test And Diagnostics Licenses

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

Using Encounter Test Contrib Scripts

The files and Perl scripts shipped in the <*ET installation path*>/etc/tb/contrib directory of the Encounter Test product installation are not considered as "licensed materials". These files are provided AS IS and there is no express, implied, or statutory obligation of support or maintenance of such files by Cadence. These scripts should be considered as samples that you can customize to create functions to meet your specific requirements.

What We Changed for This Edition

- "Revisions for Version 12.1.100"

Revisions for Version 12.1.100

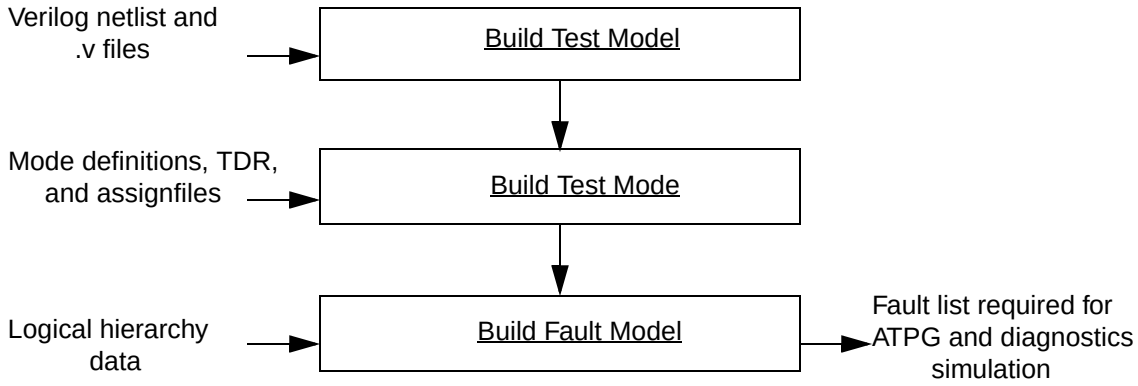
This a new guide for this release.

1

Encounter Test Modeling Overview

A test model must be created before generating test patterns or using diagnostics. This requires reading in the design, building a test mode (for example fullscan chains, JTAG IEEE 1149.1, compression logic), and assigning faults. This section briefly discusses the required inputs and processes within Encounter Test. Figure 1-1 on page 17 shows the typical high-level Encounter Test Modeling flow.

Figure 1-1 High Level Encounter Test Modeling Flow



Input for Building Encounter Test Model

Building a model for Encounter Test ATPG and diagnostics requires the following input:

- A Verilog netlist (with or without scan inserted)
- A structural Verilog technology library

- Information on how to access and control each of the unique test structures. This includes:
  - Scan enable, scan input and outputs, and system clock input pin information
  - BSDL information for 1149.1 JTAG
  - Chip initialization and clocking sequences for custom chip design test structures

Building a Test Model

Execute the following steps to build an Encounter Test model:

1. Build Model - This step reads a netlist and libraries and creates an optimized test model. The test model contains the logical hierarchy to facilitate design analysis. The following is an example command invocation:

   ```
   build_model cell=cellname designsource=filespec,...|dir1 [:dir2]...
   TECHLIB=file_spec  teiperiod=string WORKDIR=<directory>
   ```

   Refer to Build Model on page 21 for more information.

2. Build Test Mode - Define the test structures to be used during ATPG. Multiple testmodes can exist on the same design and are identified by unique names. The following is an example command invocation for a single test mode:

   ```
   build_testmode TESTMODE=name ASSIGNFILE=<pin file> WORKDIR=<directory>
   ```

   The specified `TESTMODE` name can be any arbitrary name.

   Note: If you use Encounter Test DFT Synthesis or the Cadence RTL compiler, the input data files required for building testmode are created automatically. The following are the automatically generated test modes:

- FULLSCAN - Use this test mode name for the industry standard direct pin access parallel fullscan chain test structure.
- 11491 - Use this test mode name for the JTAG 11491 test structure test mode.
- OPMISR and OPMISRPLUS - Use this test mode for the Encounter Test proprietary OPMISR compression logic.
- COMPRESSION - Use this test mode name for the Encounter Test proprietary XOR compression logic.

  Refer to section Building a Test Mode in *Encounter Test: Guide 1: Testmodes* for more information.

3. Build Fault Model - This step prepares a fault list for Encounter Test fault processing by defining the types of faults for ATPG, simulation, and diagnostics to target during analysis.

Static faults and (optionally) dynamic faults are assigned, and custom fault assignment such as no fault or logic-specific pattern faults can be applied. The following is an example command invocation:

```
build_faultmodel WORKDIR=<directory> includedynamic=yes cellfaults=no
```

Refer to section Building a Fault Model in *Encounter Test: Guide 4: Faults* for more information.

2

Build Model

Use the `build_model` command or Build Model GUI to read a netlist and build an Enounter model. Refer to "Static ATPG Use Model" in the *Encounter Test: An Overview and Quick Start* for an overview of the Encounter Test flow.

The following describes where the task to build a model is implemented in a design flow:

- Test Synthesis

  During the test synthesis flow, multiple test structures can be inserted into the design. Cadence RTL Compiler (RC) has a built-in DFT analysis to ensure that the design conforms to DFT guidelines. After synthesis is complete, a structural netlist is produced. Use the `build_model` command to read the netlist into Encounter Test to further analyze the design.

- ATPG

  Building a model is the first step in the process when performing ATPG for a design with scan structures inserted.

- Diagnostics

  A model is typically built after obtaining the failure data from the tester.

The Encounter Model is stored in the tbdata directory. There are actually two models built. A hierarchical model is created that is used by the schematic browser and interactive analysis features. This is saved in a file named `hiermodel2`. A flattened model is created from the hierarchical model for high performance functions like generating test patterns and performing diagnostics.

Building a Model

To perform *Build Model* using the graphical interface, refer to "Build Model" in the *Encounter Test: Reference: GUI*.

For a complete description of the `build_model` syntax, refer to "build_model" in the *Encounter Test: Reference: Commands*.

The syntax for the `build_model` command is given below:

```
build_model workdir=<mydir> designsource=<design file[, design file2,...]> \
            [techlib=<tech_-file[,tech_file2,...]>] [cell=<top_module_name>]
```

where:

- `workdir` = the name of the working directory
- `designsource` = the name of the design netlist file(s) or directories
- `techlib` = the name of the technology library file(s) or directories. This keyword is optional.
- `cell` = the name of the top-level module. This keyword is optional.

Supported Netlist and Technology Library Formats

The `build_model` command accepts the following data formats in the `designsource` and `techlib` keywords:

- Structural Verilog that can include UDPs and assign statements.
- Behavioral constructs are not supported.
- By default, Encounter Test searches for Verilog files ending with a `.etv, .v` or `.V` suffix
- Encounter Test 7.2 and later has limited support for Verilog parameters. These parameters may be used to define vector signals and fixed value strings, for example, 7'b1100zz1.
- Logic models already built by `build_model`
- Any file that is the container for other files or directories (must have a suffix of `.files`)
- EDIF 2.0.0 - files ending with `.edif` or `.EDIF` suffix
- VLSI Integrated Model (VIM)

Searching for Cells and Modules for Build Model

`build_model` constructs the model by selecting the top level cell (module) and recursively searching for instances and their corresponding cell definitions.

If the top level cell is not specified, `build_model` selects the last uninstantiated cell in the first designsource entry (presumed to be the top-level netlist).

To locate a specific cell definition, `build_model` searches the files and directories in the `designsource` keyword, followed by those in the `techlib` keyword. The order of the files and directories listed within each parameter is the search order used by Encounter Test. The first definition that contains contents (instances of other cells or nets) is used. If no cell is found that contains contents, the first definition without contents is used.

Within a directory, the search is performed in the following order:

- Files named `cell.etv`, `cell.v` or `cell.V`, where `cell` is the name of the specific module or cell for which Encounter Test is searching
- Compressed files named `cell.etv.suffix` or `cell.v.`*`suffix`*, where the suffix is `z`, `Z`, or `gz`
- Files named `cell.edif` or `cell.EDIF`
- Files named `cell.edif.suffix`, where the suffix is `z`, `Z`, or `gz`
- Existing Encounter Test models (files named `cell/hierModel2`)

Specify the `searchorder` keyword to define how `build_model` should search for cell definitions. The `firstdef` value selects the first definition found when processing the files and directories from the designsource and techlib specifications. Specify this value to select a blackbox definition that is first in the search order.

The `contents` keyword selects a cell definition that defines contents (instances of other cells or nets), even if a blackbox definition is found earlier in the search order.

Tip

Use the following procedure to specify long `designsource`/`techlib` path names.

a. Create a file named *`anything_you_want`*`.files`.

b. List the `designsource` files/directories to be included in the file and conform to the following guidelines:

- One name per line with no other punctuation, that is, no comments and no comma separated list.
- The file extension must be `.files`.

- ❍ if you list files on a single line with a ":" in between (and no white space), the program sends the files to the same server (similar to use of a colon on the command line).
- ❍ Do not delimit files with commas - the underlying application does not understand commas.

c. On the command line specify `designsource=<`*`filename`*`>.files techlib=<`*`filename`*`>.files`.

Note: Both may be placed in one file and specified with `designsource`.

The following is an example file and usage.

a. Create a file named `any.files` with the following content:

```
/afs/tda/home/mle/partsrc/verilog/useful
/afs/tda/home/mle/partsrc/verilog/full_stuff
```

b. Specify and run the command string `build_model designsource=any.files`

c. The command results in the following:

```
Build Model Controller starting:
Search Order  1:  "/afs/tda/home/mle/partsrc/verilog/useful"
Search Order  2:  "/afs/tda/home/mle/partsrc/verilog/full_stuff"
```

build_model Command Examples

Example 1: Single Design Netlist and Technology Library

```
build_model workdir=/local/dlx \
designsource=/local/source/dlx.v \
techlib=/local/techlibs/tech90.v
```

Example 2: Specifying Multiple Files

Separate multiple input files using a comma (no spaces allowed). A colon may also be used, but `build_model` considers colon separated entries as a single search entry.

```
build_model workdir=/local/dlx \
designsource=/local/source/dlx.v,/local/source/dlx_core.v \
techlib=/local/techlibs/tech90.v,/local/techlibs/memories.vep
```

Note: Encounter Test does not support verilog files with colon (:) as a part of the file name, such as `my:topcell.v`. This is because the colon is used as a separator while processing

a list of files. Therefore, `build_model` treats a file with a colon in its name as two separate files and exits with an error.

You can also use a mix of comma and colon to separate the multiple input files to `build_model`, as shown below:

```
build_model workdir=/local/dlx \
designsource=/local/source/test1.v:/local/source/test2.v,/local/techlibs/test3.v
```

Example 3: Using Include Files

Multiple files can be specified into a single file to minimize the number of entries on the `build_model` command line.

File name is `include.v` and contains the following two line

```
`include "/local/techlibs/tech90.v"
`include "/local/techlibs/memories.vep"
```

This file is used in the `build_model` command line as follows:

```
build_model workdir=/local/dlx \
designsource=/local/source/dlx.v,/local/source/dlx_core.v \
techlib=include.v
```

Another example of using Include Files is when you define the include file that contains the list of designsource files as shown below.

```
../file1.v

../file2.v

../../file3.v
```

In such a case, if a module is defined in multiple files (module A is defined in `file1.v` and `file3.v`), Encounter Test uses the last definition of this module. This is in contrast to how it works when you define the same on the command line. An example of the same is given below.

```
designsource=../file1.v,../file2.v,../../file3.v
```

On specifying the designsource files in the command line, Encounter Test uses the first definition of the specified module.

Example 4: Specifying Directories

You can specify a directory for the `designsource` or `techlib` keywords if the directory contains files of the form `cell.v`, where cell represents a module or cell used in the design:

For example, if `/local/memories` contains files:

- `memory1.v`
- `memory2.v`
- `memory3.v`

The design netlist contains instances of `memory1`, `memory2`, and `memory3`. The command may be specified as:

```
build_model workdir=/local/dlx \
          designsource=/local/source/dlx.v \
          techlib=/local/techlibs/tech90.v,/local/memories
```

Example 5: Specifying Top Level Module (Cell)

Encounter Test attempts to find the top level module in the design hierarchy by searching for the last uninstantiated module in the first file in the designsource keyword. To explicitly specify the top-level module, add the keyword `cell=<module name>`

```
build_model  workdir=/local/dlx \
      designsource=/local/source/dlx.v \
         techlib=/local/techlib/tech90.v \
                        cell=dlx_top
```

Example 6: Specifying ‘define to satisfy ‘if and ‘ifdef directives

In some cases, a technology library or design source might contain Verilog `‘if` and `‘ifdef` directives. In order to pass the `‘define` information to Encounter Test:

1. Create a file *<mydirective>* that contains the Verilog `‘define` statement.
2. Place this file in the `techlib` or `designsource` keyword, as required, before any of the files that need to recognize the `‘define` directive.
3. Use a colon (`:`) to separate all files in the `techlib` or `designsource` keyword that need to recognize the `‘define` directive.

For example, if file `/local/techlib/mydirective` contains `‘define` directive

```
build_model workdir=/local/dlx \
      designsource=/local/source/dlx.v \
      techlib=/local/techlibs/mydirective:/local/techlibs/tech90.v
```

Example 7: Using definemacro to specify ‘define constructs

You can use the `definemacro` keyword to specify a comma separated list of `‘define` constructs. Each of the specified `‘define` constructs is considered a global construct and processed by the `‘ifdef` constructs in the input designsource or techlib files when there is no `‘define` in any of the files.

In the following example:

```
build_model designsource=a.v:b.v:c.v techlib=tsmc13.v:techlib1.v \
            definemacro=text_macro
```

Specifying `text_macro` for the `definemacro` keyword is equivalent to defining the `‘define` construct as:

```
‘define text_macro
```

If `‘define text_macro` statement is not present in any of the source files, build_model parses the logic within the `‘ifdef text_macro` block in the source files.

Example 8: Modeling Blackboxes

Build Model typically generates an error message whenever a module definition is missing or has an incomplete pin specification. A definition of the module’s interface is required to model blackboxes. This requirement is exactly the same for NC simulation. This behavior can be overriden so that build_model automatically generates a blackbox module definition. Specify the `build_model` keyword `allowincomplete=yes` to automatically create a blackbox definition for missing cells. This keyword also adds missing pins on existing cells, whenever required.

The keyword `blackboxoutputs=<z|x|0|1>` allows you to specify the value used to tie blackbox outputs. The default is `x`.

```
build_model workdir=/local/dlx \
            designsource=/local/source/dlx.v \
            techlib=/local/techlib/tech90.v \
                allowincomplete=yes blacboxoutputs=x
```

Example 9: RAM/ROM Modeling

Encounter Test has special primitives for RAM and ROM models. Refer to “Building Memory Models for ATPG” for information on modeling the correct behaviors.

Example 10: Replacing a Cell in a Technology Library

Sometimes it is necessary to include an updated definition for a specific cell in the technology library. To include the new cell definition, place it ahead of the technology library in the `techlib` keyword and it will be selected before the definition in the base technology library.

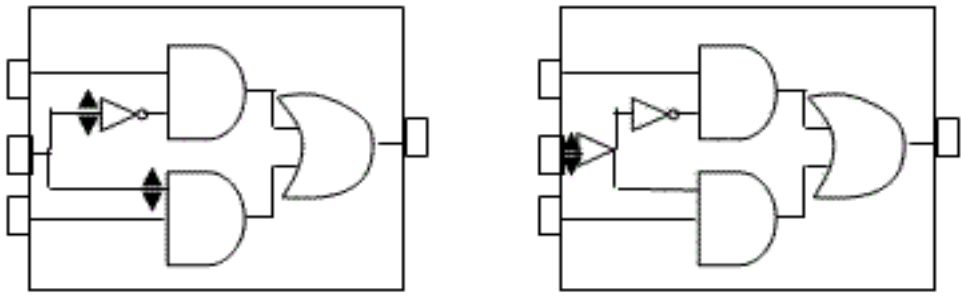
```
build_model workdir=/local/dlx \
            designsource=/local/source/dlx.v \
            techlib=/local/techlib/new_cell.v:/local/techlib/tech90.v
```

Example 11: Improving Fault Model Compatibility with Other ATPG Tools

Encounter Test provides comprehensive fault modeling capabilities including modeling at technology cell boundaries or at the primitive level. When modeling faults at the cell boundary, Encounter Test creates multiple pairs of faults when a cell fans out internally from a cell input pin. It is possible to generate a single pair of faults at the cell boundary by using the `industrycompatible=yes` keyword during build_model. This inserts a buffer into the flattened model so that the single pair of faults will be created at the cell boundary.

```
build_model workdir=/local/dlx \
           designsource=/loca/source/dlx.v \
           techlib=/local/techlib/tech90.v \
            industrycompatible=yes
```

Figure 2-1 Fault Model Compatibility with Other Tools



Example 12: Building from an Existing Model

The `build_model` command allows the use of a module within an existing Encounter Test working directory as an input. This can be specified with either the `designsource` or the

techlib keyword. This allows extraction of a lower-level entity from an existing Encounter Test model to perform additional analysis on a specific part of design.

```
build_model workdir=/local/sub_unit \
         designsource=/local/dlx/tbdata/hierModel \
         techlib=/local/techlib/tech90.v \
         cell=sub_unit
```

This is also a useful method to transfer an efficient binary version of a core's model or a technology library via tar or FTP of the tbdata directory. The following TEI binary files are required in tbdata:

- TEImsgOverrides - This fle is created by the build_model command and is an input to verify_test_structures. It contains message suppression information. This file should be saved for follow-on applications.
- TEIsourceLibPath - This file gets created when createpathfile=yes is specified with build_model command. This file contains the paths for designsource files, techlib files and the run directory RUNDIR from where the testpart was run (required when relative source/techilb path is given). This file is later used by the command write_eps.
- TEIexcludedAttr - This file contains a list of the unique attributes which are excluded during the creation of the hierAttributes file. These attributes are assumed to be not needed by any Encounter Test applications.
- TEIincludedAttr - This file contains a list of the unique attributes placed in the hierAttributes file. The hierAttributes file often contains attributes that are not used by Encounter Test (and can be dropped during Build Model process). The purpose of the TEIincludedAttr file is to a list the attributes in the hierAttributes file, which you can use as a basis for excluding attributes in future Build Model runs, by using the command line option TEICONTROLATTR.

This method can also be used to transfer protected proprietary models by cloaking them. Refer to "CLOAK Attribute" on page 39 for more information.

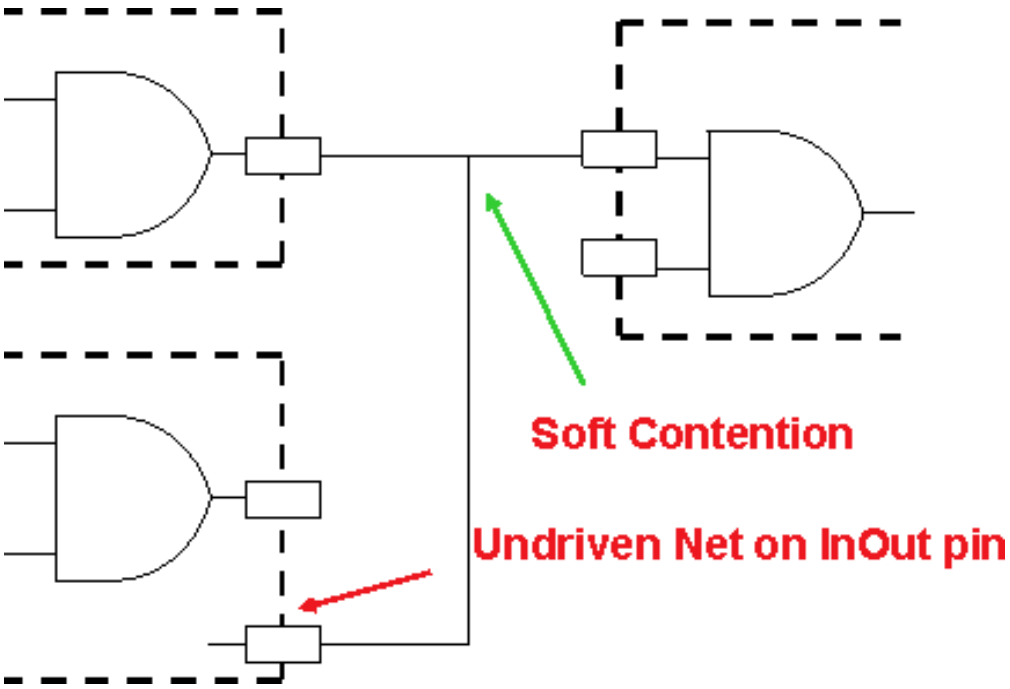
Note:

- The file name hierModel is optional unless there are other design sources, such as verilog, in the tbdata directory.
- Build_model requires both the hierModel and hierAttributes files from the tbdata directory, even though only the hierModel file is mentioned.

Example 13: Removing Contention on Unconnected Nets

Encounter Test uses a pessimistic approach while building a model. When the pin defined on a module is an undriven inout pin, it is assumed that the pin is driven by X. This can be an issue if the net it is also driven by another source (Figure 2-2 on page 30). This causes soft contention in ATPG and simulation.

Figure 2-2 Contention of Undriven Nets



Message TLM-105 helps identify the presence of this contention issue. Using the `build_model -i` option allows Encounter Test to build the model without the contention.

```
build_model -i workdir=/local/dlx \
     designsource=/loca/source/dlx.v \
     techlib=/local/techlib/tech90.v \
     industrycompatible=yes
```

Model Element Names

Each of the model elements has an index and can have a name. There are three forms of names that can be used in Encounter Test:

Full Proper Name

This is the fully specified name and is unique to one model element.

```
Block.f.l .topcell. nl.hierarchical_name
   (example: Block.f.l.TopBlock.nl.Block1)
Pin.f.l.topcell. nl.hierarchical_name
   (example: Pin.f.l.TopBlock.nl.Block1.pa)
Net.f.l .topcell. nl.hierarchical_name
   (example: Net.f.l.TopBlock.nl.nc)
```

Short Form Name

This is the *hierarchical_name* portion of the full proper name. This name is unique for a specific element type (for example, no two nets would have the same name). However, there is a possibility for the same name to exist for different element types (for example, if net “nc” and pin “PC” in the example were both named “c”, they would have the same short form name “c”).

Simple Name

This is the name without regard to hierarchy. This name is often ambiguous. In the example, the simple name of Block1's output pin is “pc” and the simple name of Block2's output pin is also “pc”.

Logic Values

In the Encounter Test logic model, logic values may appear on pins or nets in the model. The set of logic values used in Encounter Test are as follows:

1	Logic 1 (or hard 1) represents an electrical connection to a high voltage source on a specified pin or net.
0	Logic 0 (or hard 0) represents an electrical connection to a low voltage source on a specified pin or net.
X	Logic X (or unknown) represents an unknown value. This pin or net may be connected to a high voltage source, to ground, to a weak signal source, high impedance, or an oscillating or un-initialized source. Refer to “General View Circuit Values Information” in the *Graphical User Interface Reference* for related information.
Z	Logic Z (or high impedance) represents a non-driving value on a pin or a net.
H	Logic H (or weak 1) represents an electrical connection to an impedance to a high voltage source.
L	Logic L (or weak 0) represents an electrical connection to ground that is weaker than a logic 0.

W	Logic W (or weak X) represents an unknown weak value.
V	Logic V represents either an value of Logic 1 or Logic 0 where it is inconsequential whether the value is Logic 1 or Logic 0. This value can be displayed only when performing Report Faults.
Logic ~V	Logic ~V represents the opposite value of Logic V. This value can be displayed only when performing Report Faults.

Note: Logic values of *P* (pulsed) or *U* (uninitialized) may be seen on the circuit display during analysis.

Analyzing Build Model Log Results

The results of the build_model process are stored in a log file. By default, this file is located at `<work dir>/testresults/logs` and is named `log_build_model...`. To check and analyze the log results:

- Review the build_model log to verify that there are no Error or Warning [severe] messages.
- Verify that the cell (module) selected as the top level cell is correct. Build_model selects the highest cell definition with contents that it finds within the design source and technology library files. If this is different from the desired cell definition, specify the correct top-level cell name by using the `cell` keyword.

The model statistics printed before the message summary indicates the number of logic gates in the model and the counts for other significant model structures.

The following is a sample design summary. Refer to "Reporting Logic Model Information" on page 33 for more information on the Circuit Summary sections:

```
Circuit Summary
---------------

Hierarchical Model:                    Flattened Model:
    30955  Blocks                        18945  Blocks
    107517  Pins                         18945  Nodes
    69822  Nets

Primary Inputs:                        Primary Outputs:
    55  Input Only                        92  Output Only
    0 Input/Output                         0 Input/Output
```

```
    55  Total Inputs                          92  Total Outputs

Tied Nets:                                Dotted Nets:
    32 Tied to 0                             0  Two-State
    0 Tied to 1                              0  Three-State
    0 Tied to X                              0  Total Dotted Nets
    32 Total Tied Nets

Selected Primitive Functions:
    0  Clock Chopper (CHOP) primitives
    0  RAMs
    0  ROMs
    0  TSDs
    0  Resistors
    0  Transistors
    1711  MUX2s
    1  Latches
    1348  Flip-Flops
```

A quick review of this circuit data, especially the number of flip flops, RAMS, and interface pins, can be used as a positive indicator that the entire design hierarchy was successfully imported. Also, check the number of tied nets. A reason for an inordinate number of TIE X's might be the blackbox outputs that were not intended to be blackboxes, or should have been tied to high impedance (Z) to prevent contention when dotted to a common net. Refer to "Blackboxes on page 189" for more information.

Reporting Logic Model Information

The "report_model_statistics" command prints basic structural information about the logic model including:

- Circuit Statistics
- ATPG Constraints in the model
- RAM/ROM data
- Primary inputs

The following are some examples of the `report_model_statistics` command:

Example 1

```
report_model_statistics workdir=/local/dlx reportcircuit=yes
```

The following sample output is produced. Refer to "Report Model Statistics" in the *Encounter Test: Reference: GUI*for descriptions of the statistical categories.

```
Circuit Summary
---------------
```

```
Hierarchical Model:
     580  Blocks
    1893  Pins
     951  Nets

Primary Inputs:
      32  Input Only
       0  Input/Output
      32  Total Inputs

Tied Nets:
       0  Tied to 0
       2  Tied to 1
       0  Tied to X
       2  Total Tied Nets

Selected Primitive Functions:
       0  Clock Choppers
       4  RAMs
       0  ROMs
       0  TSDs
       0  Resistors
       0  Transistors

      38 L1 Latches
      38 L2 Latches
       0  L3 Latches
       0  L4 Latches
       0  L5 Latches
       0  Other Latches
      76 Total Latches

Flattened Model:
 535  Blocks
 535  Nodes

 Primary Outputs:
   7  Output Only
   0  Input/Output
   7  Total Outputs

 Dotted Nets:
   1  Two-State
   0  Three-State
   1  Total Dotted Nets
```

The following describes the categories listed in Circuit Summary:

- The number of primary I/O pins on the original design (these are the pins on Block 0 of the Hierarchical Model) can be calculated by adding the number of Input Only Primary Inputs, the number of Output Only Primary Outputs, and the number of Input/Output pins from either of the two columns.
- Adding the Total Inputs and Total Outputs gives the number of primary I/O pins in the Flattened Model, and this counts the Input/Output (bi-directional) pins twice.
- The Flattened Model section lists a number of blocks and nodes. The flat model blocks include each primitive, dot, primary input, and primary output. The nodes include each block (primitive or dot) output, primary input, and primary output. Note that the difference between the number of blocks and the number of nodes is only in the multiple-output primitives. Encounter Test supports two primitives that may have more than one output, RAM and ROM.
- The total number of RAM outputs and ROM outputs is calculated by subtracting the number of flat model blocks from the number of flat model nodes and adding the number of RAMs and ROMs.

- The numbers of various kinds of Tied Nets refer to flat model tie blocks. With respect to the hierarchical model, this is the addition of the number of logical nets that are tied and the number of unused pins that become tied in the flat model.
- Dotted Nets is the number of dot blocks in the flat model, which is the same as the number of multi-source logical nets in the `hierModel`, including nets connected to bi-directional I/O pins.

Example 2

```
report_model_statistics workdir=/local/dlx reportmem=all
```

This generates the following sample output:

```
RAM Information
---------------
   Index  Ports  Addr Pins  Data Width  Block Name
   -----  -----  ---------  ----------  ----------
   426     1      4          1           AR100DA.XPYRW_0196
   427     1      4          1           AR100DA.XPYRW_0197
   428     1      4          1           AR100DA.XPYRW_0198
   429     1      4          1           AR100DA.XPYRW_0199

RAM Information
---------------

Block 426 has 1 port with 4 address pins, data width 1,
and block name "AR100DA.XPYRW_0196".

   Port  Type   Addr 0  Data 0  Read  Write  Output 0
   ----  -----  -----   ------  ----  -----  --------
    1    R/W     1       5       6     7      1

Block 427 has 1 port with 4 address pins, data width 1,
and block name "AR100DA.XPYRW_0197".

   Port  Type   Addr 0  Data 0  Read  Write  Output 0
   ----  ----   ------  ------  ----  -----  --------
   1     R/W     1       5       6     7      1

Block 428 has 1 port with 4 address pins, data width 1,
and block name "AR100DA.XPYRW_0198".

   Port  Type   Addr 0  Data 0  Read  Write  Output 0
   ----  ----   ------  ------  ----  -----  --------
    1    R/W     1       5       6     7      1

Block 429 has 1 port with 4 address pins, data width 1,
and block name "AR100DA.XPYRW_0199".

   Port  Type   Addr 0  Data 0  Read  Write  Output 0
   ----  ----   ------  ------  ----  -----  --------
    1    R/W     1       5       6     7      1

Encounter Test Report Model Statistics completed.
```

```
Elapsed time      00:00:01.95 (hh:mm:ss.ss)
User time         00:00:00.13 (hh:mm:ss.ss)
System time       00:00:00.22 (hh:mm:ss.ss)
```

RAM/ROM Information

The RAM/ROM Information section lists a summary of the instances of RAMs or ROMs in the model. For each instance, a table is printed listing the number of ports, the type of memory cell, and the starting pin index relative to the beginning of the cell for each of the pin types (such as Address, Data, and Read Enable). Use either the hierarchical block index or the block name to view the RAM/ROM instance in the schematic browser.

Troubleshooting Common build_model Problems

The following table lists some of the common build_model problems and how to rectify them. For additional help on error messages, refer to the explanation and recommended action sections for each message in the Encounter Test: Reference: Messages or type the following command:

```
msgHelp <msgid>
```

Problem	Possible Cause(s)	Recommended Action(s)
Message "TEI-002"	The file or directory containing the missing cell was not specified in the `designsource` or `techlib` keywords.	1. Add the file or directory containing the missing cell and rerun build_model. 2. Add the `allowincomplete=yes` keyword so that build_model creates a blackbox for the missing cell. 3. If the missing cell is an Encounter Test primitive, provide the interface definition (module with input and output pins).
Incorrect module selected as the top-level cell	The design source contains multiple modules that are not instantiated by other modules.	Add the keyword `cell=<name>` to explicitly identify the top-level cell.

Problem	Possible Cause(s)	Recommended Action(s)
Message "TEI-269"	The name includes the string used to represent periods in the design.	Add the keyword `teiperiod=<string>` for period translation (where string does not appear in the design)
Message "TEI-220"	1. Behavioral Verilog is being used in the `designsource` or `techlib` keywords. 2. A syntax error exists.	1. For a memory cell, use "build_memory_model" to create an Encounter Test version of the cell or remove the behavioral verilog and specify `allowincomplete=yes` to automatically create a blackbox. 2. Correct the syntax error and rerun build_model.
Message "TEI-110"	1. The port is unused. 2. The cell of the port contains `$setuphold` constructs that make a logical connection not seen by build_model.	1. This warning can likely be ignored. 2. Update the cell to specify the connection explicitly.
Message "TEI-240"	`build_model` was unable to accurately determine the gate-level representation of the UDP.	View the schematic for the cell created by build_model. If it looks incorrect, create your own gate-level representation of the cell using Verilog and/or Encounter Test primitives, then place this cell ahead of the UDP in the `designsource` or `techlib` keywords, and rebuild the model.

Problem	Possible Cause(s)	Recommended Action(s)
Unexpected messages about a cell or module	`build_model` selected a cell definition other than the one that was expected.	If the model was built, use the `report_modules` command to check the location from where the module originated. Alternately, check the `build_model` error messages for the origin of the module. You may need to adjust the order of the files/directories in the `designsource` and `techlib` keywords.
The number of blocks in the model appears to be too small	1. Incorrect module selected as the top-level cell. 2. One of the module is incorrectly defined as a blackbox.	1. Add the keyword `cell=<name>` to explicitly identify the top-level cell. 2. Check the "TEI-010" message for any unintended blackboxes.

Encrypting a Model (Cloaking)

Cloaking is used to protect cores developed as IP to be used in other designs. If there is a concern over the disclosure of proprietary information, the core distributor does not need to deliver the Verilog source to the user, but instead deliver an Encounter model (binary) for the core. The cloaking attributes hide the core's contents from the user. Encounter Test can still access the core's contents for accurate fault modeling and simulation, but the user cannot view these.

Encounter Test supports two forms of cloaking, simple cloaking using the `CLOAK` attribute and password-based cloaking using the `TB_Cloak_Password` attribute.

Core Provider:

- Creates the IP
- Protects IP with Cloak attributes
- Creates logic model for IP
- Sends the logic model for IP to Core User

Core User:

- Includes logic model for IP in build_model designsource
- Cannot view contents for IP unless `TB_Cloak_Password` is specified

CLOAK Attribute

The `CLOAK` attribute is an optional cell attribute that controls whether the contents of the cell carrying this attribute can be viewed using the Encounter Test interactive logic model display capabilities. Specify `CLOAK=YES` to disable the viewing capability. After a cell is cloaked, there is no way to uncloak it and view its content. To view the cell, the model needs to be rebuilt using a version of the source (or hierarchical model) that does not contain the `CLOAK=YES` attribute.

The following is an example of Verilog with `CLOAK` attribute specified:

```
module CND2X4 (Z, A, B); //! CLOAK="YES"
input A, B;
output Z;
nand nand0 (Z_, B_, A_);
buf i00A ( A_ , A );
buf i01B ( B_ , B );
buf o02Z ( Z , Z_ );
endmodule
```

TB_Cloak_Password Attribute

The `TB_cloak_password = value` attribute implies that the model associated with the attribute should be cloaked, and also specifies the base value for the cloaking password. The password is used to uncloak the cell so that the user can view it. When `build_model` reads the `TB_cloak_password` attribute, it marks the model as cloaked (similar to the action described for "CLOAK Attribute"), encrypts the cloaking password, and saves the encrypted cloaking password in the logic model. The encryption protects against determination of the cloaking password just by looking at the contents of the logic model.

The following example demonstrates the specification of the `TB_cloak_password`:

```
module CND2X4 (Z, A, B); //! TB_Cloak_Password="openKimono"
input A, B;
output Z;
nand nand0 (Z_, B_, A_);
buf i00A ( A_ , A );
buf i01B ( B_ , B );
buf o02Z ( Z , Z_ );
endmodule
```

Uncloaking a Cell with TB_Cloak_Password

A time-sensitive cloaking password can temporarily uncloak cells. The "prepare_cloak_password" command requires the password specified in the `TB_cloak_password` attribute in the model for the cell as input. It creates a password that expires after a specified number of days. The maximum duration, which begins with the date the executable is run, is 30 days.

The following is an example of uncloaking using `TB_Cloak_Password`:

Core Provider:

```
prepare_cloak_password workdir=<workdir> password=openKimono pwduration=30
INFO (THM-406): Prepare Cloak Password program starting. [end THM_406]
INFO (THM-430): Time sensitive encrypted cloak PASSWORD = tqkyael.
           It will expire in 30 days [end THM_430]
INFO (THM-407): Prepare Cloak Password program completed. [end THM_407]
```

The time-sensitive password produced by the "prepare_cloak_password" command must then be exported as follows to uncloak the cell:

```
TBCLOAKPW_cellname=ts_pw
```

where `cellname` is the cell to be uncloaked and `ts_pw` is the time-sensitive password.

Core User:

```
setenv TBCLOAKPW_CND2X4 tqkyael
```

or

```
export TBCLOAK_PW_CND2X4 tqkyael
et &
```

Encrypting Verilog Files

The IEEE standard verilog parser in `build_model` supports all encrypted verilog files that are encrypted using the `ncprotect` command.

Encryption is supported at a module level of granularity, that is, if any part of a module is encrypted, Encounter Test considers the entire module as encrypted. The encrypted module is protected in the verilog design and none of the data is exposed.

Encounter Test protects and hides the following data:

- Verilog design contents of the encrypted module (except the boundary pin information of the encrypted module)
- Verilog source code
- Verilog source code line numbers
- Verilog design hierarchy shown in the ET GUI interface

You can make the encryption key-based by using the `ncprotect` command with different options to set the key. The `ncencryptkey` keyword of `build_model` accepts a path of a directory or a colon separated list of paths of directories containing a set of different keys. These keys are used by the IP provider to encrypt the verilog design file. Encounter Test requires these keys to allow `build_model` to execute successfully. The `ncencryptkey` keyword sets the *`NCPROTECT_KEYDB`* environment variable. This variable is then read by the `.ncvlog` to read the multiple key-based encrypted verilog file.

For more information, refer to "build_model" in the *Encounter Test: Reference: Commands*.

The encrypted verilog file has the following format:

```
`pragma protect begin_protected
20vaH6q+0/ao0bXcOP1HIlIezSFqezwssEuvftd2
DmJYIBJtGIeoEyw46uZtlJteFwJ+8Yt0xPeNs6p10
v92sNd8nlnxqFDicFqU9GhKN2wEM4W6V8Vqep/Wi/
6nXj5TqaSbJkUJ6JpHMjDYQD7nE4az8qltQUO0MFG
0tIk8cReSEdPrzT+Cug== OsMT6eEODXgrcz8fhiB
`pragma protect end_protected
```

where all the data between the begin and end pragma statements is encrypted/decrypted by using NC APIs. You can view the boundary pin names of the instances of an encrypted module.

Note: If the top level cell in the Encounter Test model is encrypted, then Encounter Test builds a model, and all Encounter Test applications will work as designed. However, applications that read data from Encounter Test and rely on internal names will not function properly. This is because exposure to downstream problems is much higher when the top level (chip) module is protected. Therefore, do not encrypt the top level module.

For key-based multiple encryptions in the verilog file:

- If the `ncencryptkey` keyword is not provided with the `build_model` command, the command fails and displays an error message. `ncvlog` also fails and notifies that the *`NCPROTECT_KEYDB`* environment variable is not set.
- If the `ncencryptkey` keyword is provided with the `build_model` command, and a correct and complete set of keys are not provided in the same format that is required by *`NCPROTECT_KEYDB`*, then the `build_model` fails and displays an

error message. `ncvlog` also fails and notifies that an error has occurred while decrypting the verilog file. The TEI-812 warning message is displayed and the encrypted part of the designsource is discarded.

- If the top cell is not found, the TEI-002 error message is displayed.

Following is the list of locations and files, where Encounter hides the encrypted data:

- Log files
- GUI interface
- Tool messages
- Data dumps
- Any knowledge inferred by analyzing the design
- Any output of Encounter Test designed for consumption by its downstream applications

The following are some of the limitations while working with encrypted verilog files:

- Encounter Test cannot generate parallel verilog patterns for any model in which one or more scan cells are part of encrypted verilog modules. Therefore, you will have to implement verilog simulation serially in such cases. This is because Encounter Test will not be able to refer to nets within encrypted modules, as would be required when producing parallel Verilog patterns.
- In the presence of one or more encrypted modules in the design, the `write_failures` command will include the string `name_is_protected` while writing the failures information. While reading back this information, `read_failures` will ignore the failures which will have such names containing the string `name_is_protected` and hence will report warning messages.
- Any ASCII data created by Encounter Test for models with protected Verilog will contain invalid names for any entities within the protected logic. Any downstream processing of such data by Encounter Test or any other tools may fail if they try to read these invalid names.

Optimizing the Logic Model

Technology libraries used by EncounterTest may contain logic that is necessary for verification and timing, but have no value for ATPG. The presence of such logic may hinder ATPG performance and require additional memory.

Removing Dangling Logic

Build Model automatically optimizes a design to remove unnecessary logic from the flat model. These conditions facilitate improved performance and reduce memory requirements during ATPG.

```
build_model optimize=dangling
```

Model optimization results in the following potential reductions:

- Dangling logic

  Logic that does not feed a pin on the technology cell boundary is excluded from the flat model. This technique results in fewer ignored faults in the fault model.

- Tied latch/flop ports

  Ports on flops or latches with tied clock inputs are removed if their tied block source is within the technology cell. This may result in reduced total and collapsed fault counts.

- Tied gate inputs

  Non-controlling, tied inputs to AND, NAND, OR, or NOR gates are removed as inputs if their tied block source is within the technology cell. This may result in reduced total and collapsed fault counts.

Note:

- **Schematic Display:** Optimized logic is represented on the schematic display in grey without the simulation logic values.
- **Expected Results:** Optimizations are applicable to all parts and will produce varied results based on the extent of dangling logic and/or tied inputs within the technology cells.

Building a Technology Library Model

The “build_library_model” command allows you to create a logic model that consists of one instance of each cell in a technology library. This model is useful when verifying the suitability of the technology library for ATPG. A typical library validation flow consists of running the following steps in the given order:

1. “build_library_model”
2. “build_testmode”
3. “verify_test_structures”

4. “create_logic_tests”
5. “write_vectors”

After patterns are generated with a library model, they are usually verified with an independent simulator such as ncsim.

Provide a list of cells to build a library model. Encounter Test creates an instance of each cell in the list. If a cell is listed more than once, Encounter Test creates multiple instances of the same cell in the library model.

The following is an example of building a library model using the `build_library_model` command:

```
build_library_model workdir=/local/lib_model \
    designsource=/local/techlib/tech90.v cellnames=/local/myListOfCells
```

where `cellnames` specifies a file containing a list of the technology cells to be included, one cell name per line. Cell names must use the same case as the technology library source.

Building Boundary Model

Build Boundary Model is used to create a model for a chip that consists of the subset of logic active in specified boundary scan external modes.

This small model for the chip is then used for Interconnect Test at the next higher level of the package (typically MCM).

To build a boundary model using command lines, refer to “build_boundary_model” in the *Encounter Test: Reference: Commands*.

To build a boundary model using the Graphical User Interface, refer to “Build Boundary Model” in the *Encounter Test: Reference: GUI*.

Restrictions

Build Boundary Model has the following restriction:

Read permission from the owner of a design is required if the design is not in your own directory.

Prerequisite Tasks

1. *Select* or *Create* the test mode for the chip using `BOUNDARY=EXTERNAL,MODEL` within the SCAN section of your mode definition input file.
2. Open the project for which the boundary model is to be built.

Naming Conventions for Boundary Model Block, Pin, and Net

The boundary `hierModel` produced by Build Boundary Model follows a naming convention for its elements to enable a correlation between them and the corresponding elements of the input chip `hierModel`. The blocks are named as `Bxxxxxxxx`, where `xxxxxxxx` is the block index in the full chip `hierModel`. Pins are named as `Bxxxxxxxx.pinname`, where `Bxxxxxxxx` is the block to which the pin attaches, and pinname is the simple name for the pin in the full chip `hierModel`. Nets are named as `Nzzzzzzzz`, where `zzzzzzzz` is the pin index of the pin contained in this net in the full chip `hierModel`.

Hierarchical and Flattened Model Characteristics

The Hierarchical Model is the view of the design when importing a model importer and as displayed by Encounter Test's graphical circuit display. Listed under the Hierarchical Model are the numbers of Blocks, Pins and Nets. The counts include every level of the hierarchy, so if you were to display block 0 in the circuit display, then explode it, and explode everything under that, and continue exploding until everything is exploded down to and including the primitive level. The number of blocks displayed, including “hier blocks,” is included in the block count. It is the same with the pins and the nets. Note that one logical net is counted multiple times in the `hierModel` if it crosses hierarchical boundaries. See “Block Actions” in the *Encounter Test: Reference: GUI* for related information.

The Flattened Model is what most Encounter Test applications work with. There are several differences between the two models. First, in the flat model, all the “hier blocks” disappear so that the only blocks left are the primitives. The pins and nets associated with those hier blocks also disappear. A logical net that crosses hierarchical boundaries and is counted as several nets in the Hierarchical Model becomes one net in the flat model. Thus, the flat model often has fewer blocks, pins, and nets than the `hierModel`.

Another difference between the Hierarchical Model and the Flattened Model is in how the dot blocks (wired circuits) are represented. In the `hierModel`, a single logical net has multiple source pins. In the flat model, a new block is created with one input per source pin of the original logical net. Each of these inputs of the “dot” block is connected to the corresponding source pin of the driving block. This tends to increase the number of blocks, pins, and nets in the flat model. See “Nets with Multiple Sources” on page 196 for related information.

A third difference between the Hierarchical Model and the Flattened Model is in the bi-directional primary I/O pins. A bi-directional pin is connected to a net that has internal sources and sinks, so that when the pin is acting as an input, there are multiple sources driving the net. If the externally applied signal is weaker than the internally applied signal, the resolved state on the net is controlled by the internal driver(s) and is different from the signal from the external source. To account for this, Encounter Test models a bi-directional pin by putting a "dot" block in the net and splitting the bi-directional I/O pin into a pure input pin and a pure output pin. The generated dot block is fed by all the original internal drivers connected to the net, plus the generated input pin, and it feeds all the original internal sinks, plus the generated output pin. Thus, the original I/O pin is counted twice, and there are extra blocks, pins, and nets in the flat model to represent the "dot."

A fourth difference between the Hierarchical Model and the Flattened Model is in the implicitly tied nets. Unused pins on the inputs of blocks in the `hierModel` will be connected to an automatically generated "tie" block in the flat model. Sourceless nets named "ZERO", "ONE", "GND", or "VDD" are assumed by Encounter Test to be tied to 0, 1, 0, or 1 respectively, and as needed, tie blocks are generated to drive these nets. Refer to "Sourceless Nets" on page 199 for additional information.

In the Circuit Summary, the number of primary I/O pins on the original circuit (these are the pins on Block 0 of the Hierarchical Model) can be calculated by adding the number of Input Only Primary Inputs, the number of Output Only Primary Outputs, and the number of Input/Output pins from either of the two columns. Adding the Total Inputs and Total Outputs gives the number of primary I/O pins in the Flattened Model, and this counts the Input/Output (bi-directional) pins twice as described earlier.

The Flattened Model lists a number of Blocks and a number of Nodes. The flat model Blocks include each primitive, each dot (described earlier), each primary input, and each primary output. The Nodes include each block (primitive or dot) output, each primary input, and each primary output. Note that the difference between the number of Blocks and the number of Nodes is only in the multiple-output primitives. Encounter Test supports two primitives that may have more than one output: RAM and ROM. By subtracting the number of flat model Blocks from the number of flat model Nodes and adding the number of RAMs and the number of ROMs, you have the total number of RAM outputs and ROM outputs in the circuit. Refer to "RAMs and ROMs" on page 121 for additional information.

The numbers of various kinds of Tied Nets actually refer to flat model tie blocks. With respect to the `hierModel`, this would be equal to the number of logical nets that are tied, plus the number of unused pins which become tied in the flat model.

Dotted Nets is the number of dot blocks in the flat model, which would be the same as the number of multi-source logical nets in the `hierModel`, including nets connected to bi-directional I/O pins.

Creating a Flattened Model

A flat model can be created from an existing hierarchical model by using Encounter Test's `build_flatmodel` command line executable. Refer to "build_flatmodel" in the *Command Line Reference* for a description of the syntax used to create a flat model.

Building Memory Models for ATPG

Creating and Using a Memory Model

Build Memory Model is a utility that assists in the creation of Verilog memory models for use in Encounter Test. This tool allows creation of a basic memory model with very little effort. Build Memory Model makes it easier to represent the content of a behavioral model or data-sheet as input to Encounter Test without spending a significant amount of time coding the low level Verilog primitives. By providing Verilog output, the Verilog models can be modified if desired.

Important

The resulting Verilog contains instances of Encounter Test primitives (such as latches and RAMs/ROMs). The Verilog can be read only by Encounter Test.

A memory model is created or modified using any of the following methods:

- Create a memory model by invoking `build_memory_model` and specify values via the utility interface
- Open an existing memory model in Tcl format by clicking *File* - *Open* `cellname.tcl` on the interface or by specifying an `infile` Tcl file parameter on the command line.
- Convert an existing memory model to a Verilog model using additional options for `build_memory_model`. Refer to "build_memory_model" in the *Encounter Test: Reference: Commands* for description of the command options.

Using the Memory Model Utility

Use any of the following methods to invoke the utility. These methods require inclusion of the Encounter Test executables in the `PATH`.

- At the command prompt:
  - Type `et|eta|ett|ediag -e build_memory_model`

  - Type `et|eta|ett|ediag -c`, press `Enter`, then type `build_memory_model` and press `Enter`.
  - Type `etet|eta|ett|ediag -tcl` to start the Tcl command line environment, then type `build_memory_model`.

- From the Encounter Test GUI *Command* input field, type `build_memory_model`.

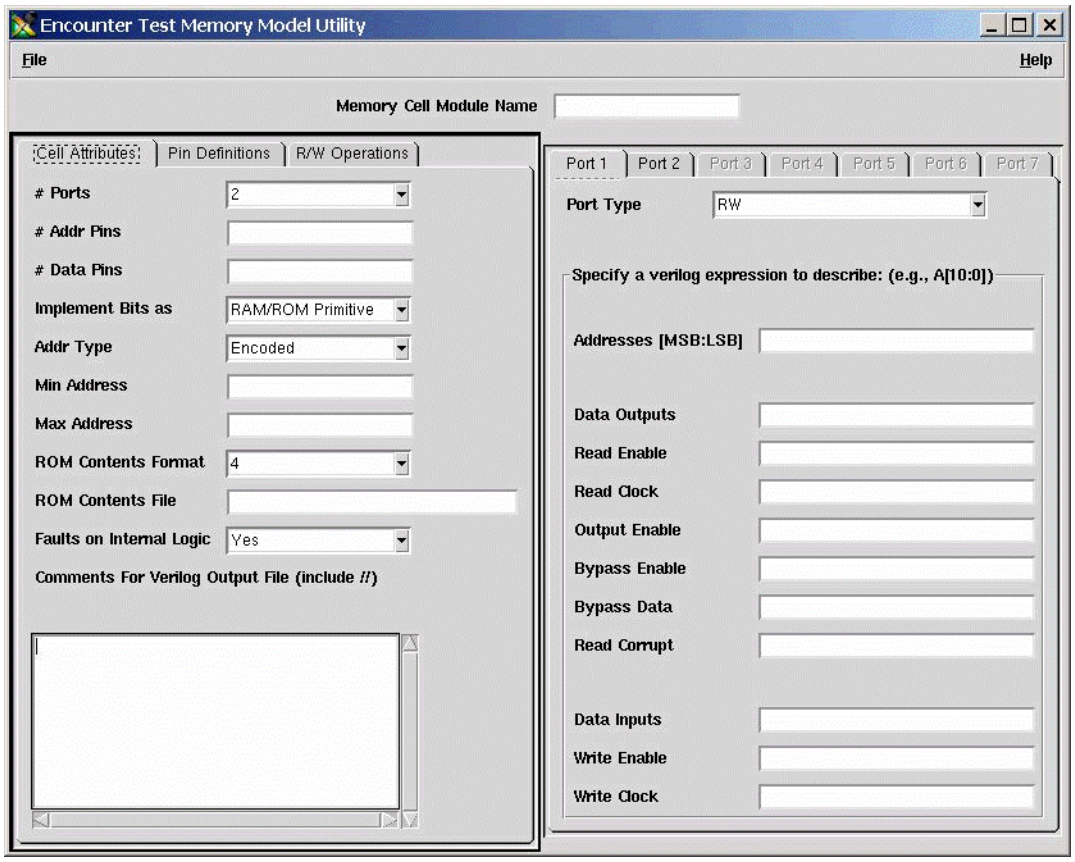
The form consists of two panes for data entry.

The data on the following screen examples in this section reflect the following scenario:

- Definition of a single port memory cell called sp_a16_d3.
- The memory cell contains 16 addresses and 3 data bits.
- The read and write operations are rising edge-triggered.
- There is an active high output enable signal.
- When disabled, the output of the memory is Z.

The initial screen displays *Cell Attributes* values as seen in Figure 2-3 on page 49.

Figure 2-3 Memory Model Utility - Cell Attributes



File Pull-down

The **File** pull-down lists the following selections:

- *New* prompts for save of the current session and resets all fields to default values. The *Memory Cell Module Name* field is cleared to allow a new cell name to be specified.
- *Open* displays the directory structure to allow navigation to and selection of an existing model.
- *Close* prompts for save of the current session and resets all fields to default values.
- *Save* will save the current values to the currently opened file.
- *Save As* allows the current file to be saved under a new name and/or extension. Files named `cellname.tcl` are recommended.
- *Generate Verilog* creates a gate-level Verilog model for the memory. A file name of `cellname.v` is recommended. An existing memory model can be converted in the

command line environment. Refer to "build_memory_model" in the *Encounter Test: Reference: Commands* for description of the command options.

- *Exit* terminates the Memory Model Utility session.

Cell Attributes

Enter the *Memory Cell Name* in the associated input field.

Enter *Cell Attributes*:

- *# Ports* - enter the number of ports in the RAM.
- *# Addr Pins* - enter the number of address pins
- *# Data Pins* - enter the number of data pins
- *Implement Bits as* - select one of the following drop-down options:
  - *RAM/ROM Primitive* to implement bits as a RAM/ROM primitive
  - *Latches* to model the RAM with an array of latches
  - *Latches Along Diagonal* to implement the latches to hold data only at address zero, the highest address, and any address that differs from these addresses by exactly one bit. This reduced model is useful for generating tests for logic surrounding a RAM using latch implementation without overhead associated with a fully specified latch model.
- *Addr Type* - select one of the following drop-down options if a latch implementation is selected:
  - *Encoded* if 2^n addresses are represented with n address pins. This is the default for memories that are modeled as a RAM/ROM primitive.
  - *Decoded* - if the addresses are already decoded, the number of address lines should be equal to the number of addresses.
- *Min Address* - enter the minimum valid address of non-power-of-2 arrays.
- *Max Address* - enter the maximum valid address of non-power-of-2 arrays.
- *ROM Contents Format* - enter the format of the data in the ROM contents file.
- *ROM Contents File* - enter the name of the ROM contents file.
- *Faults on Internal Logic* - click the drop-down and select one of the following methods for placement of faults in, out, or surrounding the memory cell. It is recommended to include faults to enable verification of the Verilog model.

- Select *Yes* to include faults inside the memory cell.
- Select *No* to exclude faults inside the memory cell.
- Select *Boundary* to place faults that surround the memory cell.

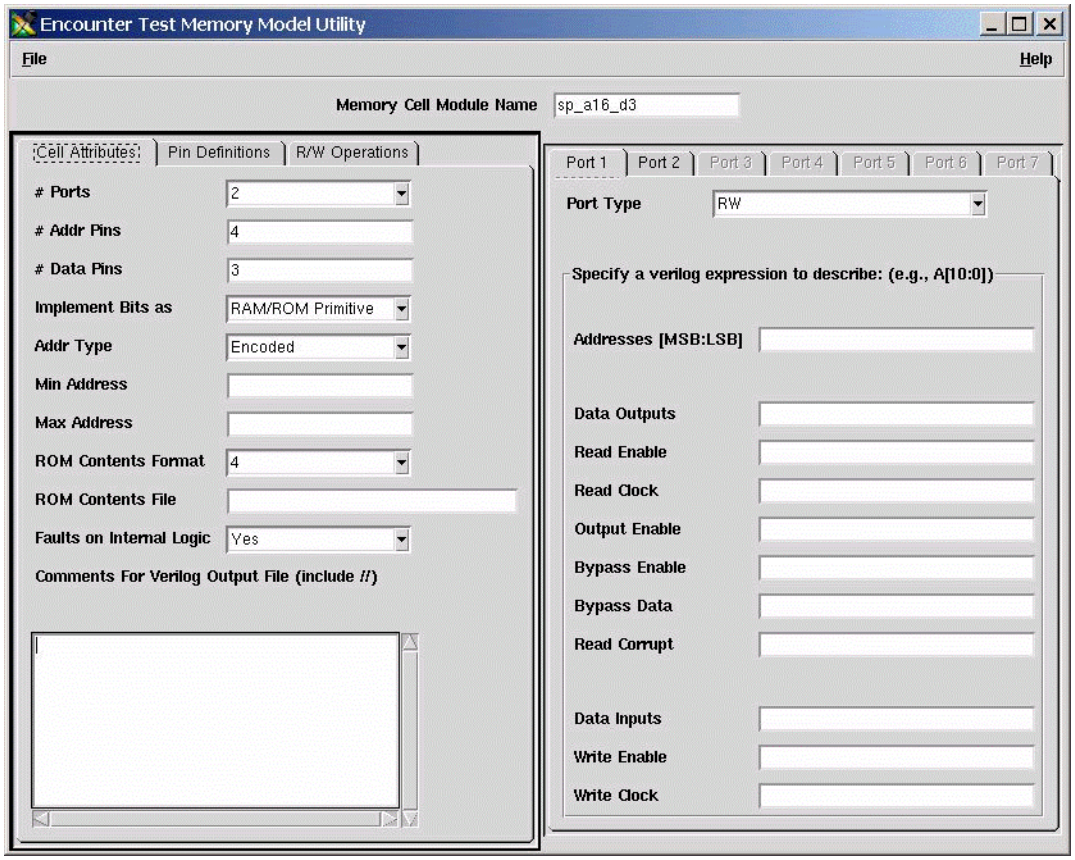
The faults can subsequently be removed by selecting *No* and regenerating the Verilog model.

Note: Leaving the faults in the model causes additional ATPG processing time.

- *Comments* - enter the comment, preceded by //, to be added at the beginning of Verilog file to record details such as location and date of creation of the cell.

Figure 2-4 shows *Cell Attributes* with fully specified values.

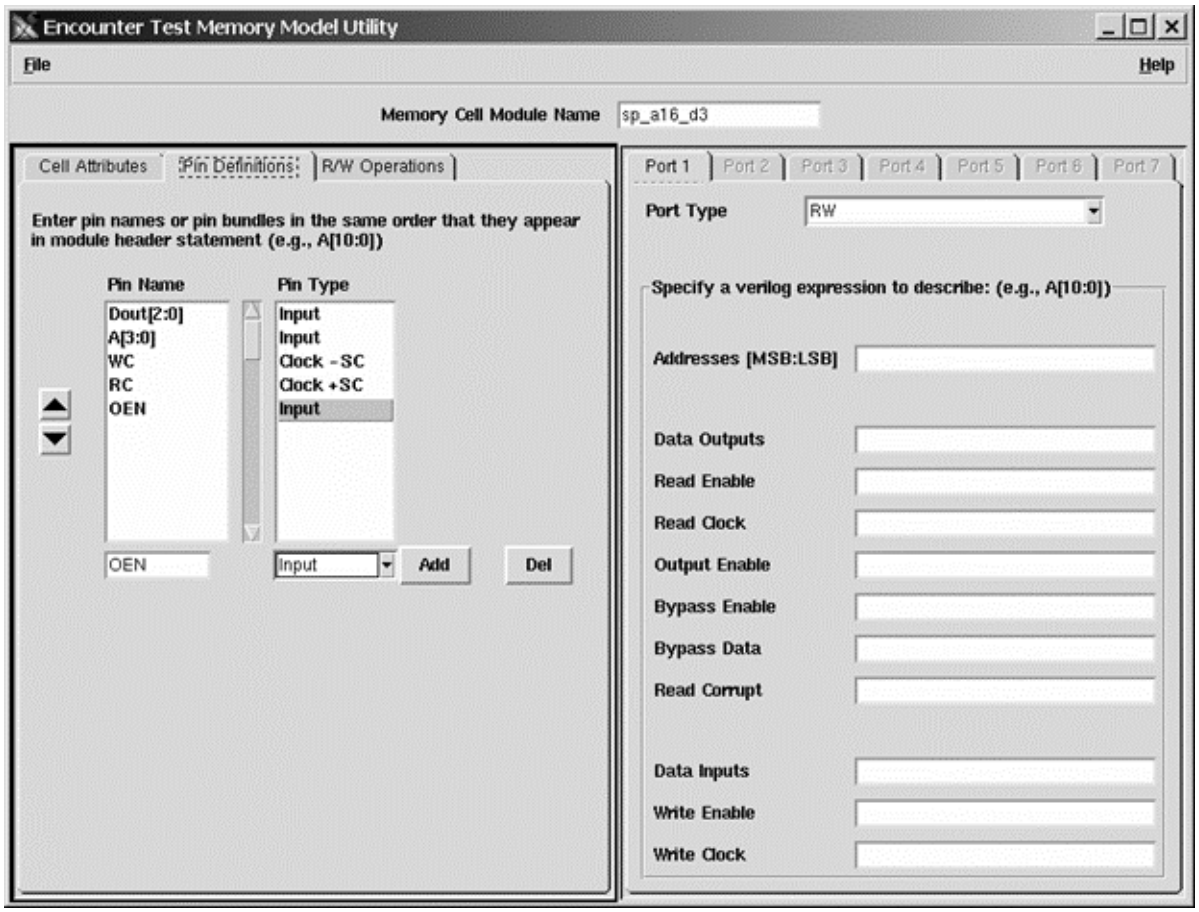
Figure 2-4 Example Cell Attributes Values



Pin Definitions

Click the *Pin Definitions* tab to display and enter pin definitions for the memory cell. The *Pin Definitions* fields are shown in Figure 2-5 .

Figure 2-5 Memory Model Utility - Pin Definitions



Enter *Pin Definitions*:

Note: Enter pin names in the same order of appearance as in the Verilog module header statement. It is important to match the order of pins with an existing behavioral description of the memory in case positional pin binding is used when the cell is instantiated in the design.

- Enter a pin name in the input field below the *Pin Name* list. The preceding figure shows `OEN` as the most recently specified pin.
- Select a corresponding *Pin Type* by clicking the drop-down below the *Pin Type* list. The following are the available selections:

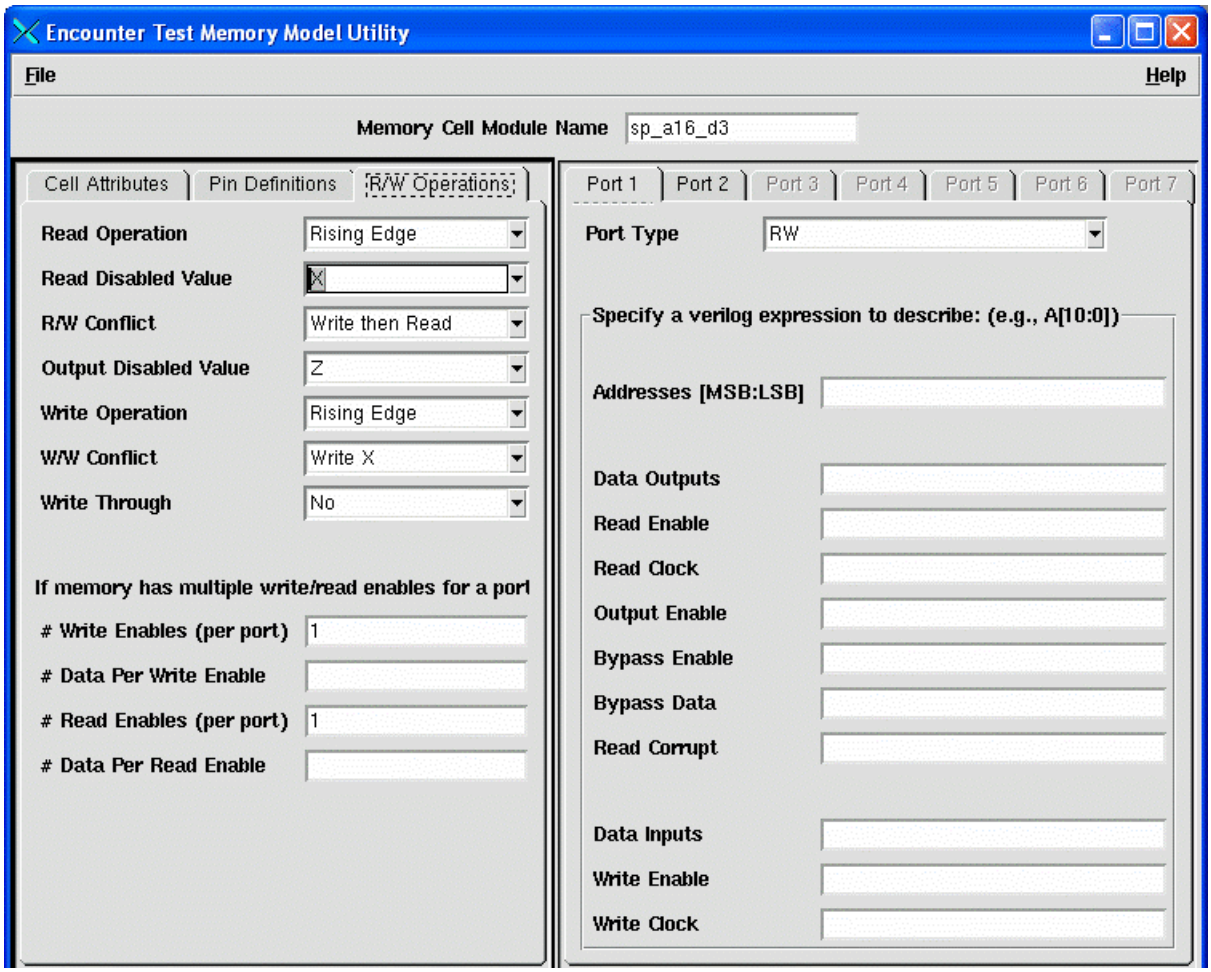
- *Input*
- *Output*
- *Clock -SC*
- *Clock +SC*
- *Input -TI*
- *Input +TI*

- Click *Add* to add the specified pin to the *Pin Name* list.
- Use one of the following methods to modify the content of the *Pin Name* list:
  - To remove a list entry, click and highlight a pin, and then click *Del*.
  - To change the order of a pin, click and highlight a list entry and then click the up or down arrow on the left of the list until the pin is placed in the desired position.
  - To select multiple continuous items, click the first item, press `Shift`, click the last item to highlight the range, and then release `Shift`. Use the up/down arrows or the *Del* button as desired.
  - To select non-continuous items, click the first item, press `Ctrl`, click and highlight additional items, and then release `Ctrl`. Use the up/down arrows or the *Del* button as required.

Read/Write Operations

Click the *R/W Operations* tab to display and set Read/Write operations for the memory cell. The *R/W Operations* fields are shown in Figure 2-6 on page 54.

Figure 2-6 Memory Model Utility - R/W Operations



Specify values for *R/W Operations*:

- *Read Operation* - click the drop-down and select one of the following:
  - *Rising Edge*
  - *Falling Edge*
  - *Combinational*
  - *Level Sensitive*

  Encounter Test RAM/ROM primitives are combinational when reading. If the read enable signal is at its active state, the RAM/ROM outputs can be seen. Otherwise the *Read Disabled Value* is seen on the RAM/ROM output. For non-combinational read operations, flip-flops or latches are used to model an edge-triggered or level sensitive read respectively. For non-combinational read operations, the *Read Disabled* value

defaults to hold. Some ATPG tools provide options to specify unique read disabled behavior when either the read clock or read enable signal is active while the other is inactive. To model this behavior, refer to "Translating Readoff Behavior From Other ATPG Tools" on page 62.

- *Read Disabled Value* - if the specified *Read Operation* is *Combinational*, click the drop-down to select one of the following values:
  - *0*
  - *1*
  - *X*
  - *Z*
  - *Hold*

  This value appears on the RAM outputs when the *Read Operation* is disabled. Note this only pertains to memories with a combinational read operation (not clocked).
- *R/W Conflict* - select the order of read and write operations when both are to be performed for the same address regardless of whether the read and write operations are performed by the same port. Click the drop-down and select one of the following:
  - *Write then Read*
  - *Read then Write*
  - *Read X then Write* (preserves memory contents)
  - *Write X then Read* (destroys memory contents)
- *Output Disabled Value* - click the drop-down and select one of the following values that should be seen when the Output Enable signal is inactive:
  - *X*
  - *Z*
  - *0*
  - *1*

  Select *Z* to indicate that three-state drivers are implemented on the outputs of the memory. Select *0*, *1*, or *X* to indicate that MUXes are inserted with the `D0` input tied to the appropriate value. `D1` is fed by the memory output and the Output Enable signal acts as the selector.
- *Write Operation* - click the drop-down and select one of the following:

- *Rising Edge*
- *Falling Edge*
- *Level Sensitive*

Encounter Test RAM/ROM primitives are level sensitive. To create a Rising or Falling Edge model, a latch is automatically inserted on each of the data and address inputs of the memory cell. The flip-flop behavior is modeled using master/slave pairs, where the inserted latch is the master, and the RAM/ROM primitive is the slave.

- *W/W Conflict* - click the drop-down and select one of the following behaviors to apply to multi-port RAMS with potential W/W conflicts:
  - *Last Port* , the default, allows the highest numbered, enabled port in the RAM (independent of time) to write.
  - *Write X* causes X to be written if more than one port is enabled for write at the same address.

    Using this option does not mean that the ATPG engine will not generate any W/W conditions in the patterns. The W/W conditions will cause the RAM to write an X, and this behavior might match with the simulation model. However, it does not guarantee that W/W conditions will be avoided during ATPG.
  - *Prohibit* adds a modeling constraint to the memory cell to prevent the Test Generator from producing patterns that enable multiple writes at the same time. This option is recommended if you want to completely avoid W/W conditions to a RAM (due to the inability of RAM silicon to handle such cases and resulting reliability issues).

    Note: This constraint does not depend on whether the addresses are the same. Any patterns simulated using the `Prohibit` constraint will cause three-state contention if multiple write clocks are enabled at the same time.
- *Write Through* - click the drop-down to select one of the following to specify whether to model the write through capability:
  - Select *No* if an explicit read operation must be performed in order to see the data on the output.
  - Select *Yes* to model the write through capability. The write operation causes the data to appear on the memory data output.
- *If memory has multiple write/read enables for a port* - specify the following:
  - *# Write Enables (per port)* - enter a number
  - *# Data Per Write Enable* - enter the number of allocated data bits per write enable

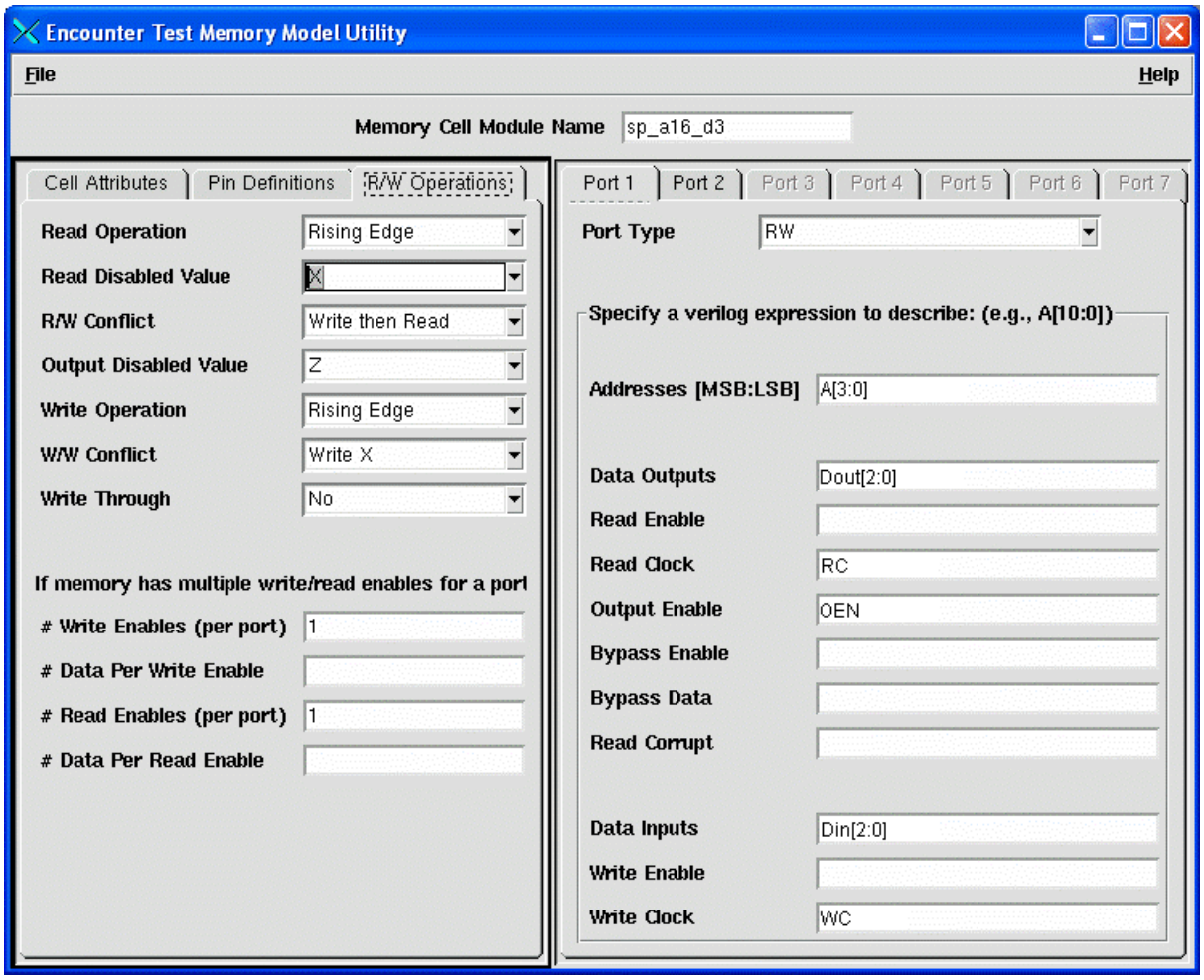
- *# Read Enables (per port)* - enter a number
- *# Data Per Read Enable* - enter the number of allocated data bits per read enable

Use the preceding selections if the memory has multiple write enable or read enable signals for a given port. Multiple enable signals are typically used when the memory has the capability to write one byte or one bit of a data word. For example, if the memory has 32 data bits (for example D[31:0]), and the memory is able to independently write each byte (8 bits) of data (D[31:24], D[23:16], D[15:8], D[7:0]), there would likely be 4 write enable signals (for example WE[3:0]) in the *Write Enable* field on the port definition pane.

Port Settings

Click the *Port 1* tab on the right hand pane to display and set port values. The right hand pane describes each of the ports in the memory. The *Port* fields are shown in Figure 2-7 .

Figure 2-7 Memory Model Utility - Port Settings



- *Port Type* - click the drop-down and select one of the following types:
  - *RW* to indicate a Read/Write port
  - *R* to indicate a Read port
  - *W* to indicate a Write port
  - *Set* to indicate a Set port

The following fields describe the memory port in terms of input and output pins. These fields must contain valid Verilog expressions. In the example shown in Figure 2-7 on page 58, the port behavior is relatively straight forward, however the fields can contain more complex expressions. Some examples follow:

- `~OEN` would indicate the *Output Enable* is active low.

- An *Addresses* value of `BIST ? TA[3:0] : A[3:0]` would indicate that a BIST enable pin (`BISTE`) is used to determine whether the address is specified by the `TA` address bus or the `A` address bus.

Note: Use bit-wise operators in lieu of logical operators when specifying these expressions.

Read Enable functions are ANDed with *Read Clock* functions. The same is true for *Write Enable* and *Write Clock* functions. The *Output Enable* function causes a three-state driver or a Mux to be placed on the output of the memory, depending on what value is seen with the *Output Enable* field disabled.

Tip

`Chip Enable` signals may be modeled by ANDing them with the `Read Enable` and/or `Write Enable` signal (for example, specifying `CE` and `WE` in the *Write Enable* field).

Use the *Bypass Enable* and *Bypass Data* fields if a bypass function is provided in the memory. Use the *Bypass Enable* field to describe the behavior that enables the bypass. Use the *Bypass Data* field to describe the data that is seen when the bypass is enabled.

Specify an expression in the *Read Corrupt* field. Whenever this expression is true, the values on the output of the memory will be set to X. It will not corrupt the contents of the memory.

Tip

The bypass function is implemented as a simple MUX. The *Bypass Enable* is used as the select, to choose between the RAM output or the *Bypass Data*. Some behavioral models are written such that the data is written to RAM, and at the same time, the data inputs are placed on the data output. Although the latter condition may be interpreted as a bypass, it should be modeled as a write-through operation. In this case, *Bypass Enable*/*Bypass Data* are not specified. The write-through is accomplished by selecting a *Write Through* of *Yes* on the *R/W Operations* tab.

Verifying the Memory Model

Use the following process to verify the memory model:

1. Run Build Model to import the Verilog memory model into Encounter Test. Refer to "Building a Model" on page 21.
2. Run Build Test Mode to create a test mode. Refer to Build Test Mode in *Encounter Test: Guide 2: Testmodes*. A `FULLSCAN` test mode should suffice.

3. Run Build Fault Model to create a fault model. Refer to Encounter Test Fault Model Overview in *Encounter Test: Guide 3: Faults*.
4. If using Encounter Test, run Create Tests to create ATPG tests for the memory model. Refer to "Performing Test Generation" in the *Encounter Test: Guide 5: ATPG*.
5. Run Write Vectors with output format of verilog to create Verilog format patterns.
6. Run `ncsim` with the resulting Verilog patterns and the *original behavioral memory model* (supplied by memory provider). Refer to SimVision documentation.

This process will verify that the patterns created by Encounter Test using the generated gate-level model produce the same results when simulated against the behavioral Verilog model. Although not an exhaustive test, it provides a high degree of confidence that the generated model is adequate for ATPG.

Creating Blackboxes

It is also possible to create a blackbox model for a memory cell. Use the following procedure to create a blackbox model:

1. Provide a name for the memory cell.
2. In the *Cell Attributes* tab, set # *Data Pins* and # *Addr Pins* to zero (`0`).
3. In the *Pin Definitions* tab, define the input and output pins in the same order as they should appear in the module header statement. Do not specify any other field.
4. Save the file (*File - Save As*).
5. Generate Verilog (*File - Generate Verilog*).

The created model contains only the module header and input/output statements. When read into Encounter Test, the blackbox memory will produce `X`s or `Z`s on its outputs.

Creating Blackboxes with Outputs Tied to Z

To create a blackbox memory cell where the outputs are tied to `Z` (rather than being left as `X` generators), create a blackbox using the procedure described in "Creating Blackboxes,"and specify to put three-state drivers on the cell's output pins. This is accomplished by providing the following additional information for each port in the appropriate *Port* tab:

1. Specify the *Data Outputs* by listing the output pins; for example, `Q[15:0]`.
2. Specify the *Output Enable* function as being tied to ground; for example `1'b0` or `gnd`.

Connect the *Output Enable* functions to a real input pin (such as `OEN`) so that it may be driven from a primary input, specify the Verilog expression that describes the output being enabled (`~OEN`). Note that if the `Output Enable` function is enabled, the `black_box` model will produce `X`s.

Creating Multiple Memories From a Template

In some cases, you might need to create several memories that have the same basic characteristics but vary only by the number of address or data pins on the memory. Encounter Test allows you to create multiple memories from one tcl specification for a memory model.

Using the command line to generate the verilog, you can override the module name, number of data pins, number of address pins, minimum address, maximum address, and the number of bit write enable signals. To create multiple memories:

1. Enter the data for the memory cell in the `build_memory_model` GUI using placeholders for the values that can edited:
   - Replace the number of address pins with `$ADDR`
   - Replace the number of data pins with `$DATA`
   - Replace the memory cell module name with `$MODULE`
   - Replace the number of bit write enable pins with `$BITWEN`
   - Replace any expression involving the number of address pins with an expression using `$ADDR` (for example, `addr_in[$ADDR-1:0]`)
   - Replace any expression involving the number of data pins with an expression using `$DATA` (for example, `data_in[$DATA-1:0]`)
   - Replace any expression involving the number of bit write enable pins with an expression using `$BITWEN` (for example, `wen[$BITWEN:0]`).
   - Replace the minimum address with `$MINADDR` (if applicable)
   - Replace the maximum address with `$MAXADDR` (if applicable)
2. Run "build_memory_model" with keywords to specify values for the placeholders:

   ```
   build_memory_model module=MN data=X addr=Y bitwen=Z  infile.tcl outfile.v
   ```

   where:
   - MN - the module name
   - X - the number of data pins

- Y - the number of address pins
- Z - the number of bit write enable pins

Note: If the expression involves a list of pins (for example, `{bwen[3],bwen[2],bwen[1],bwen[0]}`), replace the expression with the first and last entries only: (`{bwen[$BITWEN],bwen[0]}`). The remaining entries are filled in automatically.

Translating Readoff Behavior From Other ATPG Tools

Some ATPG tools express readoff behavior in terms of three values (`rx`, `ry`, `rz`), to represent various combinations of the read clock (rck) and read enable (ren) signals being inactive:

- `rx` - read clock inactive, read enable inactive
- `ry` - read clock active, read enable inactive
- `rz` - read clock inactive, read enable active

Possible values for rx, ry, and rz are:

- `0` - Logic 0
- `1` - Logic 1
- `X` - Logic X
- `Z` - High Impedance
- `H` - Hold previous value
- `H1` - Hold for 1 clock cycle then go to a specified readoff state

In `build_memory_model`, it is possible to represent these behaviors as well, using the asynchronous bypass pin and data fields. The following table describes how to map from the `rx`, `ry`, `rz` combinations to `build_memory_model` inputs.

Combinations	rx ren=0, rck=0	ry ren=0, rck=1	rz ren=1, rck=0	build_memory_model Specifications
rx=ry=rz(0,1,X,Z)	v	v	v	read operation=Combinational read disabled=v
rx=ry=rz(H)	H	H	H	read operation=Level Sensitive read disabled=Hold
rx,ry,rz(0,1,X,Z)	v1	v2	v3	read operation=Combinational read disabled=v1 bypass pin: rck ^ ren bypass data: ren ? v3:v2 OR read disabled=X(w/no bypass)
rx(H)ry,rz(0,1,X,Z)	H	v1	v2	read operation=Level Sensitive read disabled=Hold bypass pin: rck ^ ren bypass data: ren ? v2:v1
ry(H)rx,rz(0,1,X,Z)	v1	H	v2	read operation=Level Sensitive read disabled=Hold bypass pin: ~rck bypass data: ren ? v2:v1

Combinations	rx ren=0, rck=0	ry ren=0, rck=1	rz ren=1, rck=0	build_memory_model Specifications
rz(H)rx,ry(0,1,X,Z)	v1	v2	H	read operation=Level Sensitive read disabled=Hold bypass pin: ~ren bypass data: rck ? v2:v1
rx, rz (H) ry(0,1,X,Z)	H	v1	H	read operation=Level Sensitive read disabled=Hold bypass pin: rck & ~ren bypass data: v1
ry, rz (H) rx(0,1,X,Z)	v1	H	H	read operation=Level Sensitive read disabled=Hold bypass pin: ~ren & ~rck bypass data: v1
rx, ry (H) rz(0,1,X,Z)	H	H	v1	read operation=Level Sensitive read disabled=Hold bypass pin: ren & ~rck bypass data: v1

Combinations	**rx ren=0, rck=0**	**ry ren=0, rck=1**	**rz ren=1, rck=0**	**build_memory_model Specifications**

Notes:

1. To implement `H1`, substitute `H1` for `H` in the chart above and specify the `H1` value as the read disabled value.
2. Bypass data must be repeated once for each data bit for example, `{rck ? v2 : v1, rck ? v2 : v1, rck ? v2 : v1, ...}`
3. Bypass pins including `rclk` will produce WARNING [Severe] level messages in `verify_test_structures`
4. Rising Edge or Falling Edge Read Operations may be substituted for Level Sensitive

Securing Encounter Test Database

You can restrict access to an Encounter Test database (`tbdata`) to a specific user, internet address, or list of commands by securing the database. This helps you prevent unauthorized access to sensitive design data contained in the Encounter Test database (`tbdata`) when providing it to a chip manufacturer for defect localization.

The design data is stored in the Encounter Test model files, specifically the hierarchical model (`hierModel2` file).

To secure an Encounter Test database, run `build_model` by specifying the `AUTHPASSWORD` keyword. This adds an identifier string in the internal header of the hierarchical model to indicate the model is secured.

This prevents any previous versions of Encounter Test from accessing the model. Such a model is referred as a secured model and requires a password be specified for access. The password specified for the `AUTHPASSWORD` keyword for the `build_model` command is referred to as a master control password.

An encrypted version of the master control password is saved in an Authorization Data Base (ADB). This contains all access authorization information and must be available to access the secured model. The ADB is stored in a file named `adb.gz` in the `tbdata` subdirectory of the working directory (WORKDIR). The file is stored in a binary format which further secures the passwords (beyond encryption). The initial ADB setting allows all Encounter Test commands to be executed providing the master control password is specified.

You can also optionally use the `grant_access` command to generate new access control passwords using the master control password. This command accepts a variety of options for restricting access for a password. The `grant_access` command adds encrypted information about the value of the password, the attributes associated with the password, and an encrypted version of the master control password for internal use to the ADB. Upon successful granting of access to a password, the `grant_access` command adds a new entry at the end of the ADB. Therefore, if you specify an existing password, the associated information will be not be replaced, but appended to the ADB, which means that the granting of access is cumulative with respect to a specific password. After granting the desired access, you can then deliver the updated ADB to the third party along with the information about the password(s). The third party can then access the database using the password, provided all the restrictions associated with that password are met.

Refer to grant_access in the *Encounter Test: Reference:Commands* for more information.

Note: The `grant_access` command allows specifying the same value for both the `AUTHPASSWORD` and `accesspassword` keywords. This could result in an unusable Authorization Database (ADB) as the original authorization password could be redefined in such a way as to disallow subsequent invocations of `grant_access`. In that case, you will have to rerun `build_model` with the `AUTHPASSWORD` keyword.

3

Edit Model

Editing a Model

The "edit_model" command allows you to update a logic model after it has been built. You can update the structural model data (blocks, pins, and nets) as well as the attribute data such as fault assignment directives.

To edit a model using the graphical interface, refer to "Edit Model" in the *Encounter Test: Reference: GUI.*

The examples given below show some simple command-line scenarios using edit_model. "Edit Model File Syntax on page 213" contains a complete list of the editing commands.

Tip

Editing a model might cause some or all results to be removed. If you have any existing test data that you want to keep, it is recommended that you copy the design to another directory before editing. However, this is not mandatory.

Edit Model Process Flow

```
build_model  cell=top  designsource=<source Verilog>  techlib=<techlib Verilog>
<other params>
edit_model    editfile=<file name>  <other params>
build_testmode …….
```

The result is that the logic structure is edited in the internal representation of the design within Encounter Test. Edits can affect how Encounter Test views the logic and creates test patterns. These edits do not affect how other tools view the design, including NCsim when simulating the test patterns created by Encounter Test.

Example 1: Add a new net connection

```
edit_model workdir=<directory> editfile=<filename>
```

where:

<filename> contains the following statements:

```
add pin p1 to cell c1 direction input;
add net n1 to cell c1;
connect pin p1 to net n1 cell c1;
connect pin a instance I to net n1 cell c1;
```

Note: The edit applies to all instances of cell c1.

Example 2: Delete a pin

```
edit_model workdir=<directory> editfile=<filename>
```

where *<filename>* contains the following statement:

```
delete pin p1 cell c1 direction input;
```

Example 3: Add Attribute FAULTS=NO to a cell

```
edit_model workdir=<directory> editfile=<filename>
```

where *<filename>* contains the following statement:

```
add ATTRIBUTE FAULTS = "NO" to cell c1;
```

Example 4: Insert a Test Point

Test points inserted by "edit_model" help evaluate the effectiveness of a test point when considering adding it to a design. Only the Encounter Test model and not the designsource netlist is updated when a test point is inserted. `edit_model` test points are connected directly to primary inputs or outputs, and not to scannable flops or latches.

```
edit_model workdir=<directory> editfile=<filename>
```

where *<filename>* contains the following statement:

```
add test point observe pin inst1.inst2.pin1;
```

Example 5: Print Model Edit History

The model edit history lists all model edit commands performed on a design since the model was built.

```
edit_model workdir=<directory> -p
```

Example 6: Blackbox all instances of a specific cell/module

This will blackbox all instances of a given module. This will work on techlib and designsource modules. If you use this via `build_model` you can also enter `blackboxoutputs=x|z|X|Z|0|1` parameter to control the value driven by the black box. The default value is `x`.

```
edit_model workdir=<directory> editfile=<filename>
```

where *<filename>* contains the following statement:

```
BLACKBOX CELL <module name>;
```

Example editfile entry: `BLACKBOX CELL sdffnq_f1_hd;`

This will blackbox all instances of `sdffnq_f1_hd cell`.

Edit Model Restrictions

The following cannot be edited but can be overridden in a test mode define or assignment file that is used as input to Build Test mode:

- BIST attributes (such as PRPG and MISR)
- OPCG PPI, cutpoints, or PPI test functions
- Clockdomain properties
- Cloaked objects. Refer to "CLOAK Attribute" on page 39 for additional information.

Automatic reprocessing in the GUI can recreate only the open test mode. If the open test mode requires a parent test mode and the editing causes both testmodes to be removed, you need to turn off the automatic testmode creation and recreate the test modes yourself using the `editAssignFile. testmode` as the `assignfile` input, found in the tbdata directory.

To edit the `hierModel`, you can edit the pins nets and instances within any specific cell. To cross hierarchical boundaries, edit the effected cells of a given hierarchy separately. Model edit syntax supports only simple instances, pins, and nets within a cell.

Important

Any pin, net, or block name that corresponds to an edit model keyword needs to be in quotes in order to be parsed properly in the Model Edit file.

Edit Model Considerations

1. When you reinitialize the design after a structural edit, the fault model, and all test modes and their corresponding data will be removed.

Tip

If there is existing data you want to keep, edit a copy of the model.

2. During edit, a message appears above the *Edits Complete Reinitialize* button on the *Edits in Progress* window to indicate the data that will be removed when the design is reinitialized. Pay close attention to this message before you press the *Edits Complete Reinitialize* button because there is no way to back out after the button is clicked.

   If you do not want to lose your data, do one of the following:

   - Copy the design directory (tbdata) to another directory before pressing the *Edits Complete Reinitialize* button.
   - Cut the list of pending edits from the window and put it into a file (quadruple click will pick up the entire window). Then use *Cancel Edits* to cancel all the pending edits. Copy the design directory to another directory. Use the command `edit_model` with the file you created as input to apply the edits.
   - Message analysis does not use the new data. It continues to use the old model data. Therefore, you must complete the edits, reinitialize, and rerun the application that created the messages to use the edits.

     If you continue to use message analysis, the display will not include objects you have deleted or added. Therefore, the display may end up looking strange. But, it will show you the area of the design that was a problem prior to the edits. Tracing will add in the objects you added to that area of the display.

     Test function attributes may be included as attributes in the design source and then overridden in the test mode. The resolved value for the test mode is stored in globalData, and can be edited using `Edit Test Function Attributes` when that test mode is selected. To avoid confusion over the value in the design source and the resolved value, if a test mode is open, test function attributes are not available as model properties on pins/nets.

     Changing the attribute that was entered in the design source does not change the resolved value for any existing test modes. However, it would change the value for any new test modes that didn't explicitly override the value. Therefore, if no test mode is selected (open test mode `NONE`), test functions are available as model properties. Any existing test modes will have to be rebuilt to take advantage of the new values.

To edit the test functions as model properties, open test mode `NONE` prior to using *Edit Cell*. When your edits are complete, rebuild any existing test modes that you want to use with the new value.

- To add an instance of a cell that doesn't exist in the model, you must first create the new cell and then add the instance. Currently, there is no way to include additional cells from a design source such as Verilog.

4

Delete Test Model

Deleting a Logic Model

This action removes all files and deletes the working directory. Designs residing in Encounter Test may be removed at any time.

To delete a model using the graphical interface, refer to "Delete Model" in the *Encounter Test Reference: GUI*.

To delete a model using command lines, refer to "delete_model" in the *Encounter Test: Reference: Commands*.

An example of the delete_model command is given below:

```
delete_model workdir=/local/dlx
```

A

Modeling Logic Structures and Attributes

Using Verilog with Encounter Test

Encounter Test `build_model` command has the following two Verilog language parsers:

- The Encounter Test legacy Verilog parser. This parser is the default if `build_model` detects pre-2001 Verilog attributes in the input design source. Refer to "Attribute Syntax Prior to Verilog 2001" on page 89 for more information.

  ```
  build_model vlogparser=et
  ```

- The IEEEstandard parser will accept and import most of the IEEE Std 1364-2001 Verilog language standards. This parser is the same as the used by Cadence NC-SIM and is the default parser if `build_model` does not detect pre-2001 attributes in the input design source. Refer to "Attribute Syntax Using Verilog 2001" on page 90 for more information.

  ```
  build_model vlogparser=ieeestandard
  ```

The primary difference between the two Verilog parsers is the way in which the attributes are specified. Adding Model Attributes on page 88 contains more information on how to code attributes in each of the two forms.

Refer to "Adding Model Attributes" on page 88 if you have attributes such as fault assignment/deletion inserted within the net list or cell libraries before migrating from the legacy Verilog parser. If you are not migrating, the use of the IEEE Standard parser should be transparent.

Verilog Processing Limitations

The following are Encounter Test restrictions/requirements:

- Structural Verilog only

  Encounter Test only reads the structural information from the Verilog data and creates a hierarchical model from this information. Encounter Test tolerates other information contained in the Verilog data such as timing information, behavioral information, or assertions, but does not propagate into the hierarchical model.

- Parameter restrictions

  The support for parameter constructs is limited to the IEEE 2001 standard (NC Verilog) parser. Such constructs are not supported by the legacy parser.

  The following limitations exist:

  - Parameters are supported only for the purpose of defining bussed nets, ports, tied ports, and some references to the bussed nets/ports, as shown below:

    ```
    parameter BITS =15;
    output [BITS:0] A;  /* defines a bussed output port A with range of 15:0 */
    output [BITS-1:0] B; /* math op (subtraction) defines a bussed output port
    B with range of 14:0 */
    ```

    Any other parameter definitions, such as character and multi-value parameters, are parsed but not used. Some examples of such parameters are shown below:

    ```
    parameter databits="ON";
    parameter Z=( 2110:3050:5530 );
    ```

  - Real numbers are not supported as parameter values and are truncated to integers, as shown below, with an appropriate warning message.

    ```
    parameter PARM=5.2;  /* will be truncated to 5 */
    ```

    Similarly, binary, hexadecimal, and octal numbers are also converted to integers and then used as parameter values.

  - An `x` value (`1'bx`) or a `z` value (`1'bz`) is supported only to represent a tied net or port. Any other usage will result in an error message and termination of the run if it is used in a situation that affects the way a model is built.

    ```
    parameter wordx={BITS{1'bx}};/*example of parameter used in a parameter*/
                          /*15{1'bx} will just mean logic X to Build Model*/
    ```

  - defparam or specparam statements for defining parameters are not supported.

  - You cannot change a parameter value when you instantiate a module.

  - For parameters involving multiconcatenation, as shown below:

    ```
    parameter Z = {3{2'b1}};
    parameter Z = {X{2'b1}};
    ```

    - The multiplication factor (such as `3` in the example above) can only be an integer, hexadecimal, or an octal value.

    - The rightmost operand (such as `2'b1` in the example above) can only be a binary value. Hexadecimal and octal values are not supported.

    - If the binary value is an `x` or `z` type, the multiconcatenation effect will not be there. For example, `3{1'bx}` will be reduced to `1'bx`.

- Any binary value other than `x` or `z`, as shown below, will be converted to its corresponding integer value. For example, the following parameter value will be converted to 3 replications of the 2 bit binary value 1 (010101 binary = 21 decimal).

  ```
  Z = {{3{2'b1}};
  ```

  Therefore, net assignments such as shown below:

  ```
  wire [0:1] w1 = Z;
  ```

  where `Z = 2'bz;` will generate only one TSD block for the 0th index value of `w1`. That is, only `w1[0]` will be tied to a `z` value, and the other vectors of `w1` will remain untied.

- `'ifdef` Restrictions

By default, Encounter Test recognizes the state of a `'define` only within the scope of the file in which it resides. Therefore, a `'define` value set in one file will have no effect on an `'ifdef` directive that appears in another verilog file. There are two ways in which this can be overriden:

- Using a Colon Separated List of Files

  Encounter Test will recognize the state of a `'define` variable in any number of subsequent input files if the file list passed to `build_model` is separated by colons (:) instead of a comma. However, the `'define` statement must appear before any of the `'ifdef` directives that you want it to affect.

  See Example 6: Specifying ‘define to satisfy ‘if and ‘ifdef directives on page 26 for information on how to use a `'define` directive.

- Using the `definemacro` keyword

  You can specify the `'define` constructs as a comma separated list for the `definemacro` keyword. In this case, Encounter Test processes the logic within the `'ifdef` constructs in all the input source files.

  See “Example 7: Using definemacro to specify ‘define constructs” on page 27 for more information on how to use the `definemacro` keyword.

- Vectors

The Verilog language syntax allows multiple bit nets and register data types, known as vectors, to be declared by specifying a range. Encounter Test does not support vector constructs, therefore, vectors are converted into scalers with a suffix that specifies the bit index within the original vector. This bit index is placed within square brackets, as shown in the example below.

```
Verilog : input [0:3] data_bus --> Encounter Test : data_bus[0]
```

```
data_bus[1]
data_bus[2]
data_bus[3]
```

- Constant Values (1’b0)

  Verilog allows a pin of an instantiated module, gate, or primitive to be connected to a constant value. In this case, Encounter Test generates an instance of a TIUP or TIDN block using the generated instance name as above. Encounter Test also generates a net name of the form `NET$ nnn` where *nnn* is a generated net index starting at 1 and counting up for each generated net name.

- Constant Nets (Supply)

  In Verilog, some nets are defined as constant nets. In Encounter Test, these nets are modeled as nets driven by a TIUP or TIDN block. Encounter Test generates an instance name, as above, for the instances of TIUP or TIDN.

- Escape Characters

  Encounter Test preserves the escape character used in Verilog in the Encounter Test name. Therefore, a Verilog name of the form `\BUS.5` remains `\BUS.5` in Encounter Test. There is a semantic difference between escaped and unescaped names.

  Note: When using vector notation with an escaped name, the Verilog standard requires a delimiter between the name and the bracketed bit number. If there is no space, Encounter test considers the name as a scalar name. For example, `\x@yz [15]` (with a space between the `z` and `[`) is the 15th strand of the bussed object `x@yz`. `\x@yz[15]` without the space is just a scalar name, that is, not a vector notation.

- Reserved Keywords

  Reserved keywords in Verilog must not be used to annotate instance, pin, or net names in the netlist. All of the IEEE Std 1364-2001 Verilog language restrictions on the use of keywords are applicable to Encounter Test. A syntax error is issued if Build Model encounters such a case.

- Protected Verilog

  Encounter Test does not support Verilog encrypted by the `‘protect` directive. It must be unencrypted. Encounter test has it’s own internal IP protection feature called cloaking. Refer to “Encrypting a Model (Cloaking)” on page 38 for more information.

- Include Directives

  Verilog `‘include` directives are supported, but filenames must be fully specified.

Mapping Verilog to Encounter Test Primitives

This section describes the automatic mapping of Verilog primitives and other constructs to Encounter Test primitives. An Encounter Test primitive is instantiated in either of the following ways:

- A Verilog primitive is mapped to an Encounter Test primitive automatically during `build_model`.
- A Verilog UDP is mapped to an equivalent set of Encounter Test primitives. (See the table below)
- A Verilog Continuous Assignment Statement is mapped to an equivalent set of Encounter Test primitives. (See the table below)
- The primitive is instantiated explicitly by the designer in a netlist or technology cell.

The following table shows the mapping of input Verilog primitives to their Encounter Test equivalent model.

Note: In the table:

- n is the number of data inputs. The number of input ports is part of the Encounter Test primitive name
- XORs with > 4 inputs are modeled as a collection of four input XOR gates
- Multi-output buffers or inverters are modeled as a cell containing multiple buffers or inverters

Verilog Primitive	Encounter Test
and	AND_verilogPrim_n
nand	NAND_verilogPrim_n
or	OR_verilogPrim_n
nor	NOR_verilogPrim_n
xor	XOR_verilogPrim_n
xnor	XNOR_verilogPrim_n

The following two Verilog primitives can be defined with `n` outputs. Encounter Test allows only one output to be defined. Definition of multiple outputs for either Verilog primitive in a library cell model generates the Error Message TEI-118.

Verilog Primitive	Encounter Test
xor	XOR_verilogPrim_n
xnor	XNOR_verilogPrim_n

Note: In the table:

- `n` is the number of data inputs
- XORs with more than four inputs are modeled as a collection of 4 input XOR gates

- Multi-output buffers or inverters are modeled cells containing multiple buffers or inverters.

Primitives Implemented As Cells Containing Encounter Test Primitives

Verilog	Encounter Test	Implementation
pulldown	PULLDOWN_verilogPrim	TIE0 with resistor
pullup	PULLUP_verilogPrim	TIE1 with resistor
bufif0	BUFIF0_verilogPrim	TSD with inverted control
bufif1	BUFIF1_verilogPrim	TSD
rbufif0	RBUFIF0_verilogPrim	TSD with inverted control feeding resistor
rbufif1	RBUFIF1_verilogPrim	TSD feeding resistor
notif0	NOTIF0_verilogPrim	TSD with inverted control feeding inverter
notif1	NOTIF1_verilogPrim	TSD feeding inverter
nmos	NMOS_verilogPrim	NFET
rnmos	RNMOS_verilogPrim	NFET feeding resistor
pmos	PMOS_verilogPrim	PFET
rpmos	RPMOS_verilogPrim	PFET feeding resistor
cmos	CMOS_verilogPrim	NFET dotted with PFET
rcmos	RCMOS_verilogPrim	NFET dotted with PFET, feeding resistor

Primitives Mapped As Uni-Directional Using Encounter Test Primitives

Verilog	Encounter Test	Implementation
tran	tran_verilogPrim	wire
rtran	rtran_verilogPrim	buffer feeding resistor
tranif0	tranif0_verilogPrim	TSD with inverted control

Primitives Mapped As Uni-Directional Using Encounter Test Primitives

Verilog	Encounter Test	Implementation
rtranif0	rtranif0_verilogPrim	TSD with inverted control feeding resistor
tranif1	tranif1_verilogPrim	TSD
rtranif1	rtranif1_verilogPrim	TSD feeding resistor

Note: The bidirectional aspect of these primitives is not modeled in Encounter Test.

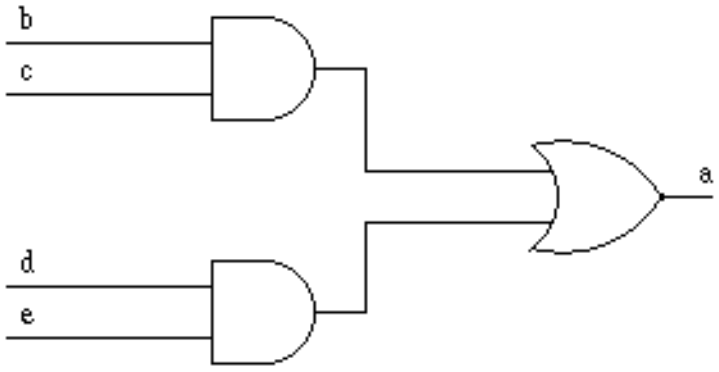
Verilog Continuous Assignment Statements

Encounter Test provides limited support for continuous assignment statements. Encounter Test implements continuous assignments as a series of logic gates connected to the net(s) on the left-hand-side of the assignment statement.

Example 1: Boolean Expression

```
assign a = (b & c) | (d & e)
```

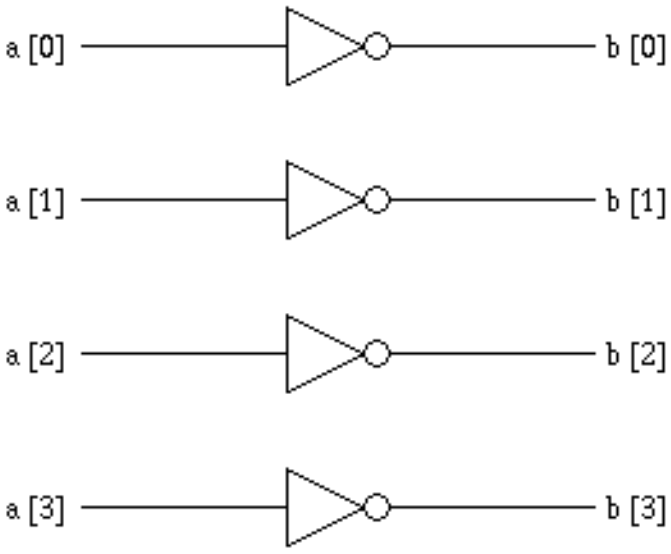
produces:



Example 2: Bussed Expressions

```
input [0 :3] a ;
output [0 :3] b ;
assign b = ~a ;
```

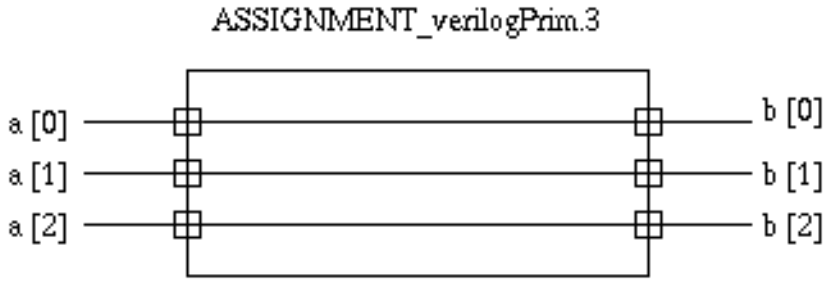
produces:



Example 3: Assignment

```
input [0 :2] a;
wire [0 :2] b;
assign b = a;
```

produces:



Supported Operators

The following operators can be used in continuous assignments statements, with the highest precedence operators listed first. It is recommended you use parenthesis to explicitly determine operator precedence.

Operator	Description
[ ]	part select
{ }	concatenation
~	bit-wise not
&	bit-wise AND, Reduction AND
|	bit-wise OR, Reduction OR
^	bit-wise XOR, Reduction XOR
^ ~, ~ ^	bit-wise XNOR, Reduction XNOR
~ &	Reduction NAND
~ |	Reduction NOR
? :	Conditional
!	Logical NOT
&&	Logical AND
||	Logical OR
=	Logical equality
!=	Logical inequality

Note: Only nets and constant values may be used as operands. Registers are not supported.

Operators Not Supported

Currently, the following operators are not supported:

Operator	Description
+,-,*/-	Arithmetic

Operator	Description
%	Modulus
>, <, >=, <=	Relational
< <	Shift left
>>	Shift right

For more information on continuous assignment statements, refer to the *Verilog-XL Reference Manual* published by Cadence.

Processing User Defined Primitives

Build Model accepts User Defined Primitives (UDPs) and processes them to create a gate-level model. This gate-level model is then verified using the states in the UDP table to ensure that simulation of the gate-level model will produce the same results as the UDP. Both combinational and sequential UDPs are accepted.

Note: Encounter Test removes faults from the internals of the synthesis of user-defined primitives. In addition, Encounter Test also removes faults from the internals of cells created by synthesis of Verilog primitives that do not map directly to Encounter Test primitives (for example, `bufif0`).

This section provides some examples of User Defined Primitives (UDPs) processed by Build Model.

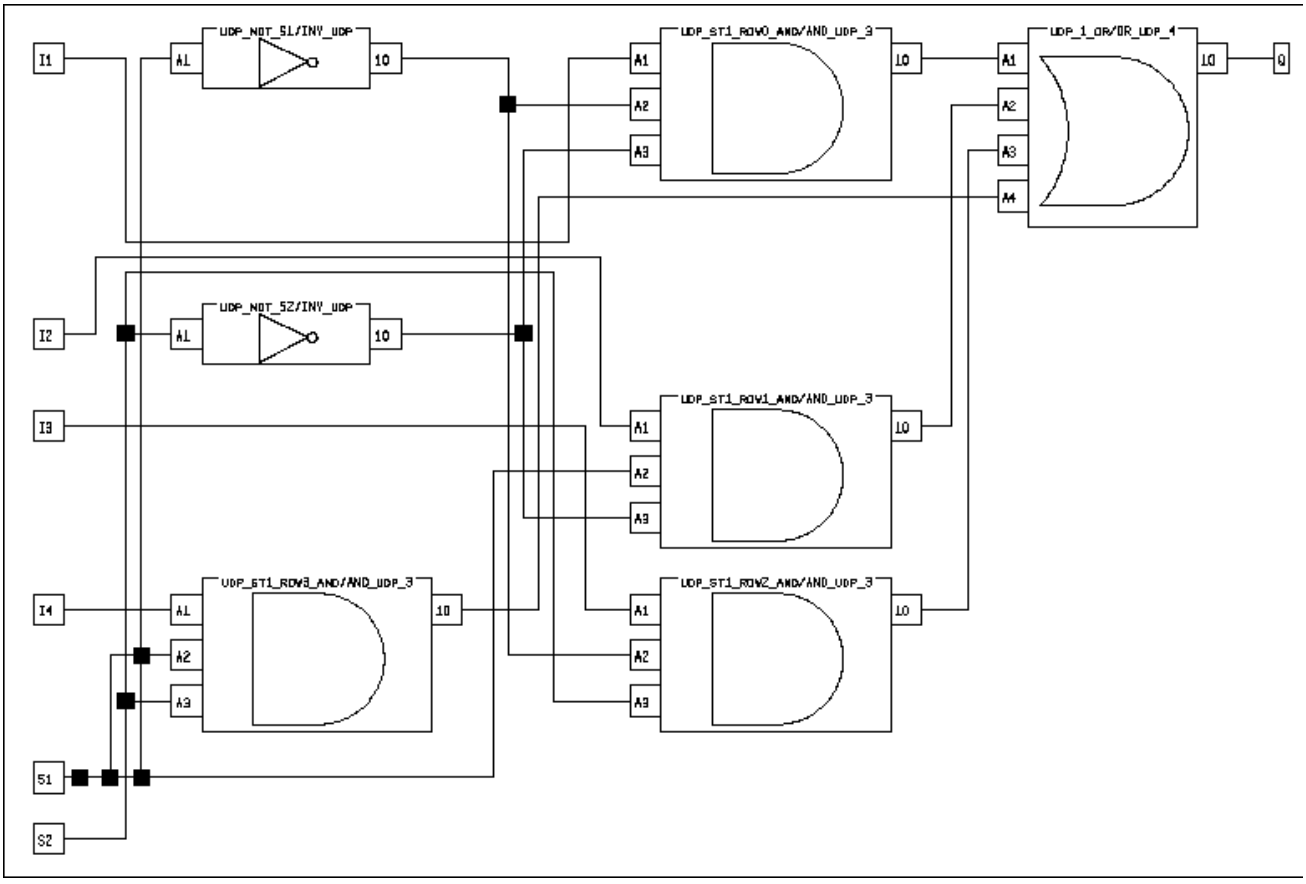
Combinational UDPs

The following UDP represents MUX with 1 output, 4 data inputs, and 2 select inputs:

```
primitive MUX41 (Q, I1, I2, I3, I4, S1, S2);

output Q;
input  I1, I2, I3, I4, S1, S2;

table

//  I1  I2  I3  I4   S1 S2  :  Q
    0   ?   ?   ?    0    0  :  0  ;
    1   ?   ?   ?    0    0  :  1  ;
    ?   0   ?   ?    1    0  :  0  ;
    ?   1   ?   ?    1    0  :  1  ;
    ?   ?   0   ?    0    1  :  0  ;
    ?   ?   1   ?    0    1  :  1  ;
    ?   ?   ?   0    1    1  :  0  ;
    ?   ?   ?   1    1    1  :  1  ;
    0   0   0   0    ?    ?  :  0  ;
```

```
        1   1   1   1     ?     ?  :  1  ;
        0   0   ?   ?     ?     0  :  0  ;
        1   1   ?   ?     ?     0  :  1  ;
        ?   ?   0   0     ?     1  :  0  ;
        ?   ?   1   1     ?     1  :  1  ;
        0   ?   0   ?     0     ?  :  0  ;
        1   ?   1   ?     0     ?  :  1  ;
        ?   0   ?   0     1     ?  :  0  ;
        ?   1   ?   1     1     ?  :  1  ;
    endtable
    endprimitive
```

The gate-level model created by Encounter Test looks like this:

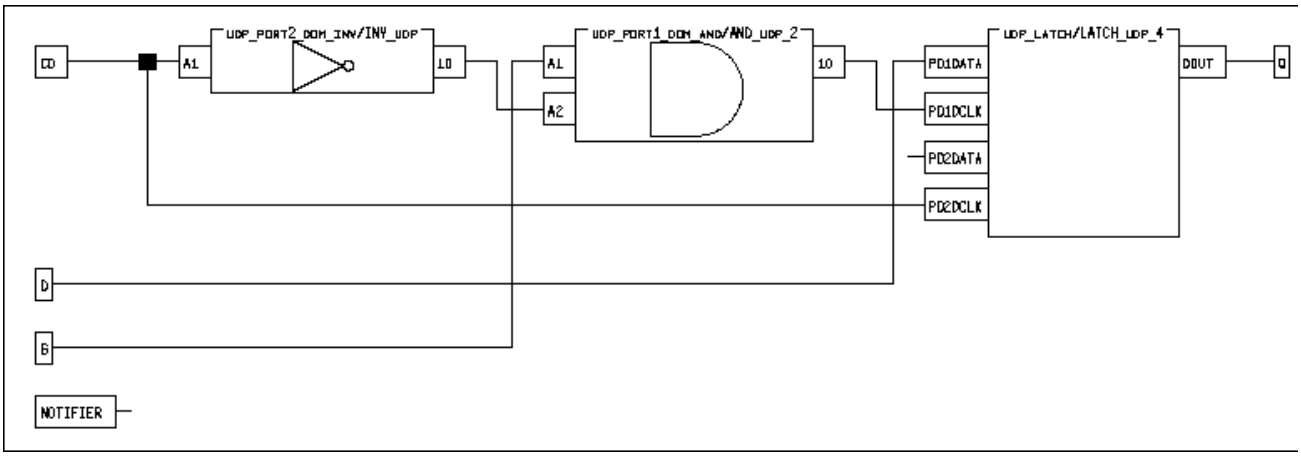


Encounter Test also processes sequential UDPs. The following example describes a latch with a reset.

```
primitive LD3 (Q, D, G, CD, NOTIFIER);

//
// latch with reset
//

output Q;
reg    Q;
```

```
input  D,
       CLK,
       CD, NOTIFIER;

table
//       D       CLK     CD     NOTIFIER       : Qt  :  Q(t+1)

         0      (0x)     0       ?             : 0   :  0    ;
         1      (0x)     0       ?             : 1   :  1    ;
         0      (x1)     0       ?             : ?   :  0    ;
         1      (x1)     0       ?             : ?   :  1    ;

         *       0       0       ?             : ?   :  -    ;

         ?       ?       1       ?             : ?   :  0   ; // asynchronous clear

         ?     (?0)      ?       ?            :  ?   :  -   ;
         ?     (1x)      ?       ?            :  ?   :  -   ;

        (?0)     1       0       ?            :  ?   :  0   ;
        (?1)     1       0       ?            :  ?   :  1   ;
         0      (01)     0       ?            :  ?   :  0   ;
         1      (01)     0       ?            :  ?   :  1   ;

         ?        ?      ?       *            :  ?   :  x  ;
    endtable
    endprimitive
```

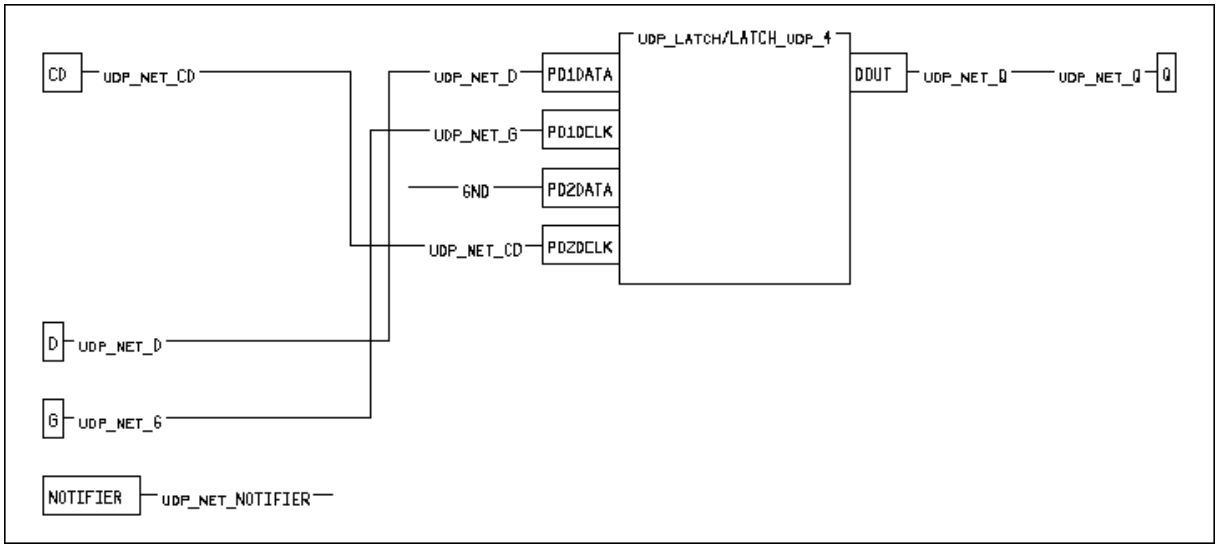
The gate level model created by Encounter Test looks like this:



Port Dominance

Whenever CD is active (1), the latch is cleared, regardless of whether or not the clock to the data port (CLK) is active. Additional logic is included in the Encounter Test model to ensure that this "Port Dominance" is modeled accurately. Port Dominance can be ignored when processing sequential UDPs by specifying `UDPDOMINANCE=NO` as a command line parameter to `build_model`.

If `UDPDOMINANCE=NO` is specified, the gate level model produced by Encounter Test would look like this:



Defining Set/Reset Ports

`build_model` supports creation of an alternate model for flip-flop Verilog UDPs. The generation of the model for Verilog UDPs defining a Set or Reset port or defining both a Set and Reset port can be controlled by using the keyword `udpsetresetx=yes` as a command line parameter to `build_model`.

Specify `yes` to create a model that will allow the flop or latch to be clocked when the Set is OFF and the Reset is X, or when the Set is X and the Reset is OFF. By default, `build_model` logic generation for Verilog UDPs does not allow a set-reset flop or latch to be clocked if both the Set and Reset are logic X.

Adding Model Attributes

An attribute is a piece of logic model data of the form *`keyword=value`*. It contains additional information about a logic model object. Attributes may be attached to the following:

- Module or cell
- Instance of a module, cell or primitive
- Pin
- Instance of a pin

Note:

1. Instance attributes (and instance pin attributes) are associated with the instance within the containing cell. That is, all usages of the containing cell have an instance inside them that has the attribute. Encounter Test does not support attributes for unique instances.
2. Attribute names and values are case sensitive.
3. Place all attributes values within double quotes.

Attribute Syntax Prior to Verilog 2001

Specify `build_model verilogparser=et` for pre-2001 Verilog attributes.

When using the legacy Verilog parser, use the following extended comment syntax to specify attributes:

```
//! [range] attribute="value"
```

where `[range]` is an optional specification of the strand(s) of a bus that will inherit the attribute and its value.

The location of the comment determines the object to which the attribute is associated.

Note: The attribute is essentially still a comment, therefore, it must appear at the end of the line.

Attribute Locations Prior to Verilog 2001

Module or Cell Attributes

```
module modname(?); //!  attribute="value"
```

Pin Attributes

```
input p1, p2, p3;  //! attribute="value"
                   // applies to p3 only
output [7:0] data; //! [3] attribute="value"
                   //  applies to data [3] or strand 3
                   //! [1:2] attribute="value"
                   //  applies to strands 1 and 2 of data
```

Note: Strand is the specification of specific elements in the Verilog syntax.

Instance Attributes

```
and (z, a, b, c);  //! attribute="value"
             //  applies to the and gate instance
memory1 meminst (...); //! attribute="value"
                     //  applies to instance meminst
```

Pin Instance Attributes

```
and (z, //! attribute="value"
        //  applies to output pin on instance of and gate
     a, b, c);
memory1 meminst (...,.D(datain), //! [0:7] attribute="value"
                                 //  applies to strands 0-7 of pin
                                 //  D on instance meminst
                 ...);
```

The following is an example of specifying pin instance attributes:

```
Module FOO (Y, p1, p2, p3); //!  TYPE=?MACRO"
input p1, p2, p3;  //! PFLT="01"
   // Attribute applies only to p3
```

Attribute Syntax Using Verilog 2001

Specify `build_model vlogparser=IEEEstandard` for Verilog 2001 design source.

Section 2.8 of the IEEE Std 1364-2001 Verilog Standard specifies the following syntax for attributes:

```
attribute_instance::=(*attr_spec{,attr_spec}*)
attr_spec::=
attr_name=constant_expression
| attr_name
attr_name::=
identifier
```

Attribute Locations Using Verilog 2001

In Verilog 2001, attributes occur before the object they are associated with:

- Module or Cell Attributes

```
(* attribute="value" *) module modname(...);
```

- Pin Attributes

```
(*attribute="value" *) input p1, p2, p3;
                   // applies to p1, p2 and p3
(* \attribute1[3] ="value" *)
(* \attribute2[1:2] ="value" *)
output [7:0] data;
                   // attribute 1 applies to strand 3
                   // attribute 2 applies to strands 1 and 2
```

- Instance Attributes

```
(* attribute="value" *) and (z, a, b, c);
        //  applies to the and gate instance
(* attribute="value" *) memory1 meminst (...);
        //  applies to instance meminst
```

- Pin Instance Attributes

```
and ((* attribute="value" *) z,
        //  applies to output pin on instance of and gate
     a, b, c);
memory1 meminst (..., (* \attribute[0:7] ="value" *) .D(datain),
                 ...);
                    //  applies to strands 0-7 of pin
                    //  D on instance meminst
```

Summary of Attributes

The following table lists the attributes supported by Encounter Test:

Table A-1 Encounter Test Attributes

Attribute Name	Value	Location	Description
CELLTYPE (also TYPE)	CELL, MACRO, SUPER, CHIPLET, CHIP, MODULE, CARD, BOARD, FRAME, MACHINE	module	Specifies the level of the hierarchy that is being defined. Refer to "Specifying the Cell Level for Graphical Display" on page 103 for more information.

Attribute Name	Value	Location	Description
CHECKSUM	string	module	Records level information about the library cell for audit purposes.
CIO	YES/NO	pin	Identifies a bidirectional pin on the top level. Refer to "Overriding CIO Attribute" on page 158 for more information.
CLOCKED_FROM	NO	instance	Specifies to TSV that apparent race conditions from the LATCH or RAM to other capture points should not be flagged as errors. Refer to "Flag Data Races with CLOCKED_FROM and CLOCKED_TO Attributes" in the *Encounter Test: Guide 3: Test Structures* for more information.
CLOCKED_TO	NO	instance	Specifies to TSV that apparent race conditions from other sources to the LATCH or RAM should not be flagged as errors. Refer to "Flag Data Races with CLOCKED_FROM and CLOCKED_TO Attributes" in the *Encounter Test: Guide 3: Test Structures* for more information.
CONTENTS	name	instance	Specifies the name of the ROM contents file. Refer to "CONTENTS Attribute" on page 133 for more information.

Attribute Name	Value	Location	Description
CONTENTS_FORMAT	1/2/3/4	instance	Specifies the format of the data in the ROM contents file. Refer to "CONTENTS_FORMAT Attribute" on page 133 for more information.
CORRELATE	pin name and polarity	pin	Specifies that a pin should be switched either in phase or out of phase with the specified correlated pin. Refer to "Specifying Differential I/O and Other Correlated Pins" on page 204 for more information.
DATALEVL	string	module	Records level information about the library cell for audit purposes. Refer to "DATALEVL Attribute" in *Encounter Test: Reference: Legacy Functions* for more information.
DFLT	NO	pin instance	Specifies whether driver faults should be removed from a driver block. Removes both DRV0 and DRV1 faults. Refer to Modeling Defects (Faults) Concepts in *Encounter Test: Guide 3: Faults* for more information.
DFN	AND/OR/ T/NO	pin or pin instance	Specifies the dot function that should be used if the pin is connected to a multi-source net. Refer to "Nets with Multiple Sources" on page 196 for more information.

Attribute Name	Value	Location	Description
ET_CHOPPER	RISKY	instance	Specifies that the invalid clock chopper, for which the attributed instance is the reconvergent block, should not be simulated with pessimistic simulation. It is to be treated as risky; simulated as normal logic.
ET_UNCONNECTED	Internal, External, Both	cell	Identifies that a pin is not to be connected to internal logic, to external logic, or both. Refer to "Identifying Unconnected Pins with the ET_UNCONNECTED Attribute" on page 171 for more information.
FAULTS	NO	instance or module	Specifies that no faults should be created for the instance of the entity being defined in the structure. Refer to Modeling Defects (Faults) Concepts in *Encounter Test: Guide 3: Faults* for more information.
FAULTS	filename	module or instance	Name of the pattern fault definition file. Refer to Modeling Defects (Faults) Concepts in *Encounter Test: Guide 3: Faults* for more information.
HF_MIN_PULSE_WIDTH	number	instance	High function minimum pulse width - used for AC testing. Refer to "HF_MIN_PULSE_WIDTH Attribute" on page 160 for more information.

Attribute Name	Value	Location	Description
HIGHADDR	number	instance	Specifies the highest address accessible for the RAM or ROM. Refer to "HIGHADDR Attribute" on page 132 for more information.
KEEP_MSG	list of TSV message numbers	module or instance	Specifies that specific TSV messages should be kept for this entity in the hierarchy. Refer to "Override TSV Message Inhibition with KEEP_MSG Attribute" in the *Encounter Test: Guide 3: Test Structures* for more information.
LOWADDR	number	instance	Specifies the lowest address accessible for the RAM or ROM. Refer to "LOWADDR Attribute" on page 131 for more information.
LPD	YES	instance	Specifies that the lower port dominates if multiple ports of the RAM are being written to the same address. Refer to "LPD Attribute" on page 131 for more information.
LTYPE	L1, L2, L3, L4, L5	instance	Specifies the LSSD latch type. Refer to "LTYPE Attribute" in *Encounter Test: Reference: Legacy Functions* for more information.

Attribute Name	Value	Location	Description
MACRO	YES	interface	Defines the logic model entity that should be tested as an embedded macro. Refer to "Macro MPR Isolation and Test" in *Encounter Test: Reference: Legacy Functions* for more information.
MACRO_MIC_NAME	string	interface	Identifies the name of the macro isolation control (MIC) file for the macro. Refer to "Macro MPR Isolation and Test" in *Encounter Test: Reference: Legacy Functions* for more information.
MACRO_CORR	keyword value pairs	pin or instance	Identifies a primary input pin or scannable latch at the correspondence point for a pin at the boundary of a macro. Refer to "Macro MPR Isolation and Test" in *Encounter Test: Reference: Legacy Functions* for more information.
MACRO_TYPE	string	module	Identifies the specific isolation requirements specified in the MIC file that should be used for this macro. Refer to "Macro MPR Isolation and Test" in *Encounter Test: Reference: Legacy Functions* for more information.

Attribute Name	Value	Location	Description
MASKOUT	0/1/X/ OVERLAY Note: OVERLAY is for ROM only.	instance	Specifies the value that should be placed on the output data pins of a RAM or ROM if an address outside the valid range is specified. Refer to "MASKOUT Attribute" on page 132 for more information.
MISR	list of LFSR cell positions for an on-product MISR	module	Defines the LFSR polynomial. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.
MISR_NET	netname	interface	Identifies the net which is connected to the last latch in the MISR LFSR. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.
ORIGIN	string	module	Created by Encounter Test `Import`, records the format of the library data. Refer to "ORIGIN Attribute" in the *Encounter Test: Reference: Legacy Functions* for more information.

Attribute Name	Value	Location	Description
PFLT	O/1/NO/01	pin instance	Specifies whether stuck-at-0, stuck-at-1, both stuck-at-0 and stuck-at-1, or no stuck-at faults should be created for the pin. Refer to Pin Fault Specifiation Using Verilog in *Encounter Test: Guide 3: Faults* for more information.
PRPG	list of LFSR cell positions	module	Defines the LFSR polynomial for an on-product PRPG. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.
PRPG_NET	netname	module	Identifies the net which is connected to the last latch in the PRPG LFSR. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.
RAISE_MSG_SEVERITY	list of TSV message numbers	module or instance	Specifies that specific TSV messages should be printed with higher severity codes. Refer to "Raise TSV Message Severity with RAISE_MSG_SEVERITY Attribute" in the *Encounter Test: Guide 3: Test Structures* for more information.

Attribute Name	Value	Location	Description
READOFF	0/1/X	instance	Specifies the value that should be placed on the data output pins of the RAM or ROM when the read enable is off. Refer to "READOFF Attribute" on page 130 for more information.
RFLT	NO	pin instance	Specifies whether receiver faults should be removed from a receiver block. Refer to Pin Fault Specifiation Using Verilog in *Encounter Test: Guide 3: Faults* for more information.
SUPPRESS_MSG	list of TSV message numbers	module or instance	Specifies that specific TSV messages should be suppressed for this entity in the hierarchy. Refer to "Suppress or Eliminate TSV Messages with SUPPRESS_MSG Attribute" in the *Encounter Test: Guide 3: Test Structures* for more information.
TB_MISR	netname	module	Alternative method to MISR_NET for identifying the net which is connected to the last latch in the MISR LFSR. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.

Attribute Name	Value	Location	Description
TB_PRPG	netname	module	Alternative method to PRPG_NET for identifying the net which is connected to the last latch in the PRPG LFSR. Refer to "PRPG and MISR Attributes" in the *Encounter Test: Reference: Legacy Functions* for more information.
TB_VERILOG_SCAN_PIN	string	instance or pin instance	Controls selection of scan, stim, and measure points in exported Verilog test vectors. Refer to "Using the TB_VERILOG_SCAN_PIN Attribute" in the *Encounter Test: Guide 5: ATPG* for more information.
TERM	0/1	pin or pin instance	Product termination to be applied for wired net. Refer to "Internal Three-State Termination" on page 148 for more information.
TFLT	O/1/NO/01	pin instance	Specifies whether slow-to-rise, slow-to-fall, both slow-to-rise and slow-to-fall, or no transition faults should be created for the pin. Refer to Pin Fault Specification Using Verilog in *Encounter Test: Guide 3: Faults* for more information.
TIE	1/0/X	pin or pin instance	Specifies the tied value of the net connected to the pin if it has no source. Refer to "Sourceless Nets" on page 199 for more information.

Attribute Name	Value	Location	Description
TTERM	0/1	pin or pin instance	Tester termination to be applied. Used for open drain drivers. Refer to "External Three-State Termination" on page 158 for more information.
UNLEV	string	module	Used to ensure that all design source files are kept in sync. Refer to "UNLEV Attribute" in the *Encounter Test: Reference: Legacy Functions* for more information.
ZDLY	YES/NO	primitive instance	Specifies that the block should be treated as zero delay when simulating with the General Purpose Simulator. Refer to "Using the ZDLY Attribute" in the *Encounter Test: Guide 5: ATPG* for more information.
OPTIMIZE	NO	module	Used with a value of "NO" to any Verilog module statement to turn off optimization for that individual module. All parent modules will inherit the specification and will also not be optimizied. Refer to "Optimizing the Logic Model" on page 42 for more information.

Identifying Technology Library Cells

A technology cell is an important layer of hierarchy in the Encounter Test model. Technology cell boundaries are important for True-Time Delay Test because they indicate where delays

are expected to be found. They are also important when creating a cell boundary fault model (build_faultmodel `cellfaults=yes`), because faults are placed on the primitives along the cell boundary.

By default Encounter Test assigns an identifier to each level in the hierarchy. The lowest level is primitive. Technology cells include primitive cell types and may include instances of other technology cells. Encounter Test allows up to 31 levels of hierarchy to be identified as a technology cell. A user can control the assignment of cell types in the hierarchy through the `build_model` editfile or by attributes within the net list. Verilog language allows the specific assignment of library CELL levels, as described in the next section.

Verilog Specification

In Verilog, the `‘celldefine` and `‘endcelldefine` directives are commonly used to identify a technology cell. Encounter Test identifies every module between `‘celldefine` and `‘endcelldefine` directives as a technology cell.

```
‘celldefine
module aiocell(z,a,b,c,d); <- A technology cell
...
endmodule
module andcell(z,a,b,c); <- A technology cell
endmodule
primitive mux(z,d0,d1,sel); <- Not a technology cell
...
endprimitive
‘endcelldefine
```

It is also possible to identify a technology cell by adding an attribute to the module definition for a cell:

Verilog 2001 Syntax

```
(* TYPE="CELL" *) module aoicell(z,a,b,c,d);
...
endmodule
```

Pre-Verilog 2001 Syntax

```
module aoicell(z,a,b,c,d); //! TYPE="CELL"
```

This attribute has the same effect as the `‘celldefine` directive.

Note:

1. `CELLTYPE` is a synonym for `TYPE`. If a module contains both attributes, Encounter Test uses `CELLTYPE` and ignores `TYPE`.
2. Both the keyword (`TYPE`) and the value (`CELL`) are case sensitive.
3. The quotes are required around the value `CELL`.
4. For more information on valid values for the `TYPE` attribute, see Specifying the Cell Level for Graphical Display on page 103.

Rules for Technology Cells

The `build_model` command uses the following rules to determine if a layer of hierarchy should be treated as a technology cell:

1. A technology cell must contain only primitives, UDPs, or other technology cells.
2. If a module contains only other technology cells (no primitives), it must either have a `TYPE=CELL` or `CELLTYPE=CELL` attribute or be within a `‘celldefine` directive.

Any module not meeting these criteria is treated as a module (RLM) by the `build_model` command.

Note: Other values for the TYPE attribute may be specified to distinguish other layers of design hierarchy beyond a module (RLM). They are useful to control tracing and displaying entities in the schematic browser. For more information on valid values for the TYPE attribute, see Specifying the Cell Level for Graphical Display on page 103.

Specifying the Cell Level for Graphical Display

The `TYPE` attribute is also used to limit the default display of a design. This attribute allows you to display the design down to the technology book level. The technology library supplier may include a `TYPE=CELL` on each of the technology library cells. To control the display at a higher level than that, you can include TYPE on any cell in your design.

There are no restrictions on the values that can be specified in the `TYPE` attribute, but the following values are recommended.

Attribute Value	Meaning
`primitive`	Primitive cell
`cell`	Technology Library Cell

Attribute Value	Meaning
`macro`	Design Macro Cell
`super`	Second level of design macro
`chiplet`	A subdivision of a chip
`chip`	The top level of a hierarchy which defines a chip
`module`	The top level of the chip-substrate design hierarchy
`card`	The top level of the card design hierarchy
`board`	The top level of the printed design board design hierarchy
`frame`	A subassembly of a machine
`machine`	The top level of the machine description hierarchy

Notes:

1. Encounter Test looks for the `CELLTYPE` attribute to find TYPE information before the `TYPE` attribute. This allows other Design Automation systems to use the `TYPE` attribute name for other purposes.
2. The same value for the `TYPE` attribute may be used at different levels of hierarchy. For example, a technology library may offer a MUX cell with a TYPE of CELL. The MUX cell may also be used as a building block in a D flip-flop cell which also has a TYPE of CELL.

Boolean Primitives

Encounter Test has the following basic boolean primitives:

- AND
- NAND
- OR
- NOR
- XOR

- XNOR
- BUF
- INV
- TIUP
- TIDN
- TIX

Important

Every Encounter Test primitive that is instantiated in a design must have a module interface definition provided as input to `build_model` to enable pin binding. An example interface definition is:

```
module AND_2(Z, A, B);  // Encounter Test Primitive
  output Z;
  input A, B;
endmodule
```

The interface definition is not required if a Verilog primitive is instantiated. Mapping Verilog primitives to Encounter Test primitives is done automatically and is described in "Mapping Verilog to Encounter Test Primitives" on page 79.

Primitive Names

Primitives may be named in either of the following ways:

`function_n`
`function_string_n`

where:

- `n` is the number of primitive inputs.
- `function` is the primitive's name (from the list above).
- `string` is any character string.

Note: Primitive function names are case insensitive.

For example, AND_2, And_2, and and_myprimitive_2 are all valid primitive names.

Rules for Pin Names

- No restriction on input pins - By convention, input pins on primitives mapped from Verilog primitives (such as and, or) start with A0 and increase to the number of inputs, minus 1 (such as A0, A1, A2).
- No restriction on output pins - By convention, output pins on primitives mapped from Verilog primitives (such as and, or) start with 01 and increase to the number of outputs, minus 1 (such as 01, 02).
- If the primitive is mapped from a Verilog primitive (such as and, or), pins are assumed to be defined in the order of output followed by inputs, so that relative pin binding may be used.
- If the Encounter Test primitive is instantiated directly, an interface definition must be provided for the primitive to identify the pin order.

Supported Logic Values for Boolean Primitives

To improve the efficiency of simulation and test generation within Encounter Test, the Encounter Test Boolean primitives support only a subset of the logic values defined in Encounter Test.

The following table describes an input logic value to a Boolean primitive, and how a Boolean primitive interprets the input logic value.

INPUT LOGIC VALUE	INTERPRETED LOGIC VALUE
0	0
1	1
X	X
Z	X
L	0
H	1
W	X

There are some cases where the hardware implementation of a Boolean cell can accommodate some of the more complicated logic values that Encounter Test uses. In these cases, the cell must be modeled with a combination of Boolean primitives and transistor primitives.

TIUP Primitive Function

The TIUP primitive has no inputs, and only a single output. The output value is a constant logic 1. Encounter Test recognizes the name TIEUP as well as TIUP for this primitive.

TIDN Primitive Function

The TIDN primitive has no inputs, and only a single output. The output value is a constant logic 0. Encounter Test recognizes the names TIEDN and TIDWN as synonyms for TIDN.

TIX Primitive Function

The TIX primitive has no inputs, and only a single output. The output value is a constant logic X. Encounter Test recognizes TIEX as a valid synonym for TIX.

BUF Primitive Function

The BUF primitive must have only one input and one output. The output value is the same as the input value. Notice that the BUF primitive changes the input value only if the input value is not one of the strong logic values.

INPUT	OUTPUT
0	0
1	1
X	X
Z	X
L	0
H	1
W	X

INV Primitive Function

The INV primitive must have only one input and one output. The output value is the inverse of the input value. Notice that the INV primitive changes weak logic values to strong logic values. There are several synonyms that Encounter Test recognizes for the INV primitive, namely NBUF, I, N, and NOT.

INPUT	OUTPUT
0	1
1	0
X	X
Z	X
L	1
H	0
W	X

AND Primitive Function

The AND primitive carries out a logical AND of its inputs. The only synonym for AND is A.

```
        |  INPUT 2
INPUT1  | 0   1   X   Z   L   H   W
--------|---------------------------
  0     | 0   0   0   0   0   0   0
  1     | 0   1   X   X   0   1   X
  X     | 0   X   X   X   0   X   X
  Z     | 0   X   X   X   0   X   X
  L     | 0   0   0   0   0   0   0
  H     | 0   1   X   X   0   1   X
  W     | 0   X   X   X   0   X   X
```

NAND Primitive Function

The NAND primitive does a logical AND of its inputs, and inverts the result. Synonyms for NAND are AI and ANDN.

```
        |  INPUT 2
INPUT1  | 0   1   X   Z   L   H   W
--------|---------------------------
  0     | 1   1   1   1   1   1   1
  1     | 1   0   X   X   1   0   X
  X     | 1   X   X   X   1   X   X
  Z     | 1   X   X   X   1   X   X
  L     | 1   1   1   1   1   1   1
  H     | 1   0   X   X   1   0   X
  W     | 1   X   X   X   1   X   X
```

OR Primitive Function

The OR primitive performs a logical OR of its inputs. The only synonym for OR is O.

```
        |  INPUT 2
INPUT1  | 0   1   X   Z   L   H   W
--------|---------------------------
  0     | 0   1   X   X   0   1   X
  1     | 1   1   1   1   1   1   1
  X     | X   1   X   X   X   1   X
  Z     | X   1   X   X   X   1   X
  L     | 0   1   X   X   0   1   X
  H     | 1   1   1   1   1   1   1
  W     | X   1   X   X   X   1   X
```

NOR Primitive Function

The NOR primitive does a logical OR of its inputs, and inverts the result. OI is a synonym for NOR.

```
        |  INPUT 2
INPUT1  | 0   1   X   Z   L   H   W
--------|---------------------------
  0     | 1   0   X   X   1   0   X
```

1	0	0	0	0	0	0	0
X	X	0	X	X	X	0	X
Z	X	0	X	X	X	0	X
L	1	0	X	X	1	0	X
H	0	0	0	0	0	0	0
W	X	0	X	X	X	0	X

XOR Primitive Function

The XOR primitive carries out a logical EXCLUSIVE OR of its inputs. Encounter Test allows up to 256 of inputs to an XOR primitive to improve the efficiency of the test generators.

	INPUT 2						
INPUT1	0	1	X	Z	L	H	W
0	0	1	X	X	0	1	X
1	1	0	X	X	1	0	X
X	X	X	X	X	X	X	X
Z	X	X	X	X	X	X	X
L	0	1	X	X	0	1	X
H	1	0	X	X	1	0	X
W	X	X	X	X	X	X	X

Note: Encounter Test can create automatic pattern faults for XOR primitives that model real defects more accurately than traditional pin faults. See Encounter Test F ault Model Overview in *Encounter Test: Guide 3: Faults* for more information.

XNOR Primitive Function

The XNOR primitive carries out an EXCLUSIVE OR of its inputs and inverts the result. A synonym for XNOR is XORI. Encounter Test limits the number of inputs to an XNOR primitive to four to improve the efficiency of the test generators. If an XNOR is defined with more than four inputs, Encounter Test issues an error message and treats the cell as an X generator.

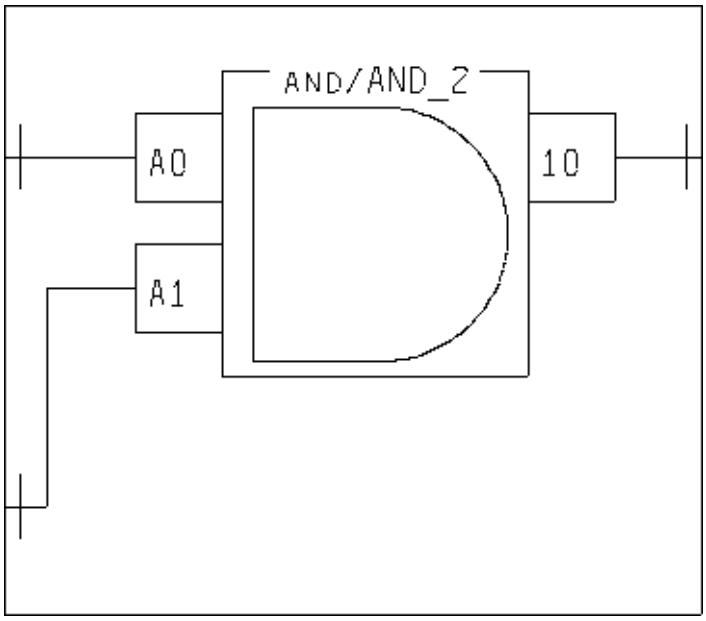
	INPUT 2						
INPUT1	0	1	X	Z	L	H	W
0	1	0	X	X	1	0	X
1	0	1	X	X	0	1	X
X	X	X	X	X	X	X	X
Z	X	X	X	X	X	X	X
L	1	0	X	X	1	0	X
H	0	1	X	X	0	1	X
W	X	X	X	X	X	X	X

Examples of Boolean Primitives

The following examples show possible technology cells which use Encounter Test Boolean primitives.

Simple Two Input AND

Figure A-1 Two Input AND Schematic



Example: Using Verilog Primitives

```
module top (out, in1, in2);
output out;
input in1, in2;
  and i1(out, in1, in2);
endmodule
```

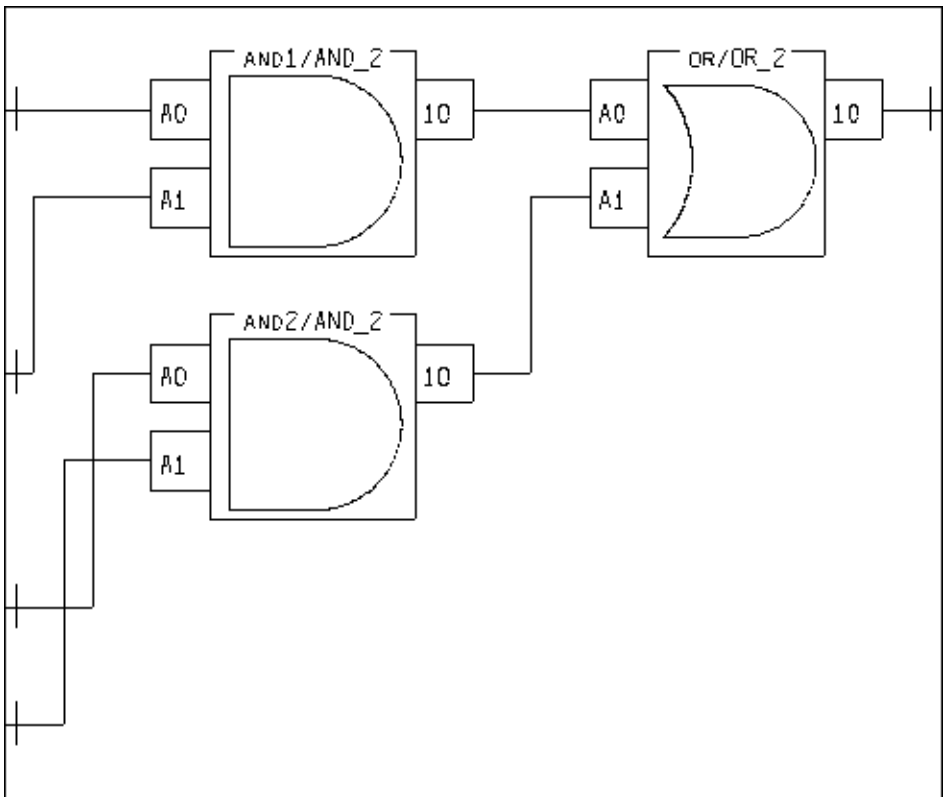
Example: Using Encounter Test Primitives

```
module top (out, in1, in2);
output out;
input in1, in2;
  AND_2 i1(out, in1, in2);
endmodule
```

```
// ET Primitive Definition Required
module AND_2(Z,A0,A1)
output Z;
input   A0, A1;
endmodule
```

2x2 AND-OR Cell

Figure A-2 2X2 AND-OR Schematic



Example: Using Verilog Primitives

```
module ao2x2 (out, in1, in2, in3, in4);
output out;
input in1, in2, in3, in4;
wire  and1out, and2out;
and   and1(and1out, in1, in2);
and   and2(and2out, in3, in4);
or    or1(out, and1out, and2out);
endmodule
```

Example: Using Encounter Test Primitives

```
module ao2x2 (out, in1, in2, in3, in4);
output out;
input in1, in2, in3, in4;
wire  and1out, and2out;
AND_2 and1(and1out, in1, in2);
AND_2 and2(and2out, in3, in4);
OR_2  or1(out, and1out, and2out);
endmodule
// ET Primitive Definitions Required
module AND_2(Z,A0,A1)
output Z;
input   A0, A1;
endmodule
module OR_2(Z,A0,A1)
output Z;
input   A0, A1;
endmodule
```

MUXes

The __MUX2 primitive represents a two-to-one multiplexor.

Primitive Name

`__MUX2`

or

`__MUX2_string`

where `string` is a user-specified character string.

Pin Names

The __MUX2 primitive has three inputs in this order: DATA0, DATA1, and SEL. It has one output, DOUT.

When the SEL input is zero (0), the value of the DATA0 input pin is placed on the DOUT pin. When the SEL input is one (1), the value of the DATA1 input pin is placed on the DOUT pin.

The truth table for the __MUX2 primitive is shown in the following figure.

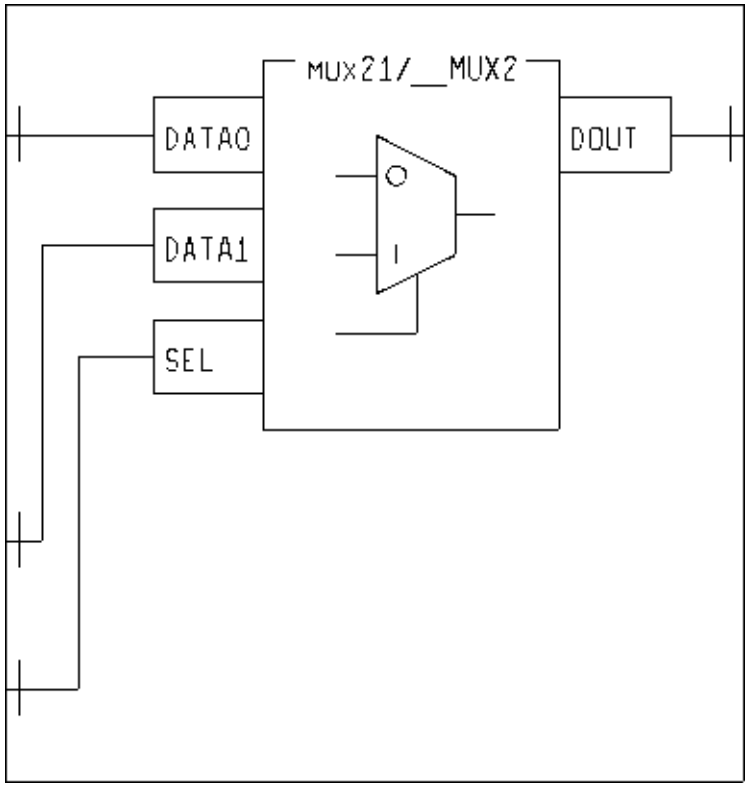
Figure A-3 _MUX2 Primitive Truth Table

SEL	DATA0	DATA1	DOUT
0,L	DATA0	-	DATA0
1,H	-	DATA1	DATA1
-	1	1	1
-	0	0	0

Note: Encounter Test can create automatic pattern faults for MUX primitives that model real defects more accurately than traditional pin faults. See Encounter Test Fault Model Overview in *Encounter Test: Guide 3: Faults* for more information.

Examples of MUX Primitive

Figure A-4 Bit Multiplexer Schematic



Example1: Using Encounter Test MUX Primitive

```
module top (out, in1, in2, select);
output out;
input in1, in2, select;
```

```
__MUX_2 MUX21(out, in1, in2,select);
endmodule

module __MUX_2(DOUT, DATA0, DATA1, SEL);
output DOUT;
input   DATA0, DATA1, SEL;
endmodule
```

Note: In Example 1, a module interface definition for a __MUX2 must be provided.

Example2: Mux Instance Using Continuous Assignment

```
module top (out, in1, in2, select);
output out;
input in1, in2, select;
out = select ? in2 : in1;
endmodule
```

Note: In Example 2, a module interface definition for a __MUX2 is automatically generated.

LATCH Primitive

The Encounter Test latch primitive takes pairs of clock and data inputs, and produces a single output. Each clock and data input pair is called a latch port.

Pin Naming Conventions

Encounter Test distinguishes the different input pins of a latch by using the pin name specified on the latch. Clock pins are named `Pxx`DCLK and data pins are named `Pxx`DATA, where *xx* is the latch port index. The clock pin on the first latch port is called `P01DCLK` and the data pin on the third latch port is called `P03DATA`.

There are no restrictions on the output pin name of the latch.

Latch Behavior

The simplest version of the latch primitive is a single port latch, which has a clock pin, a data pin, and an output pin. The output pin has the same value as the input pin when the clock is at logic 1, and retains its original value when the clock is at logic 0.

Note: A test mode must be created prior to simulating latch primitive values.

CLOCK	DATA	Prev.Value	OUTPUT
0	-	p	p
1	d	-	d
X	d	d	d
X	d	not d	X

The latch primitive becomes slightly more complex when multiple latch ports exist. The only complexity occurs when more than one clock is at logic 1 at the same time. If all the data pins are the same at all latch ports whose clocks are on, then the latch gets that data value. Otherwise, the latch gets an X.

Port 1		Port 2			
CLOCK	DATA	CLOCK	DATA	Prev.Value	OUTPUT
0	-	0	-	p	p
0	-	1	d	-	d
0	-	X	d	d	d
0	-	X	d	not d	X
1	d	0	-	-	d
1	d	1	d	-	d
1	d	1	not d	-	X
1	d	X	d	-	d
1	d	X	not d	-	X
1	d	X	-	not d	X
X	d	0	-	d	d
X	d	0	-	not d	X
X	d	1	d	-	d
X	d	1	-	not d	X
X	d	1	not d	-	X
X	d	X	d	d	d
X	d	X	not d	-	X
X	d	X	-	not d	X

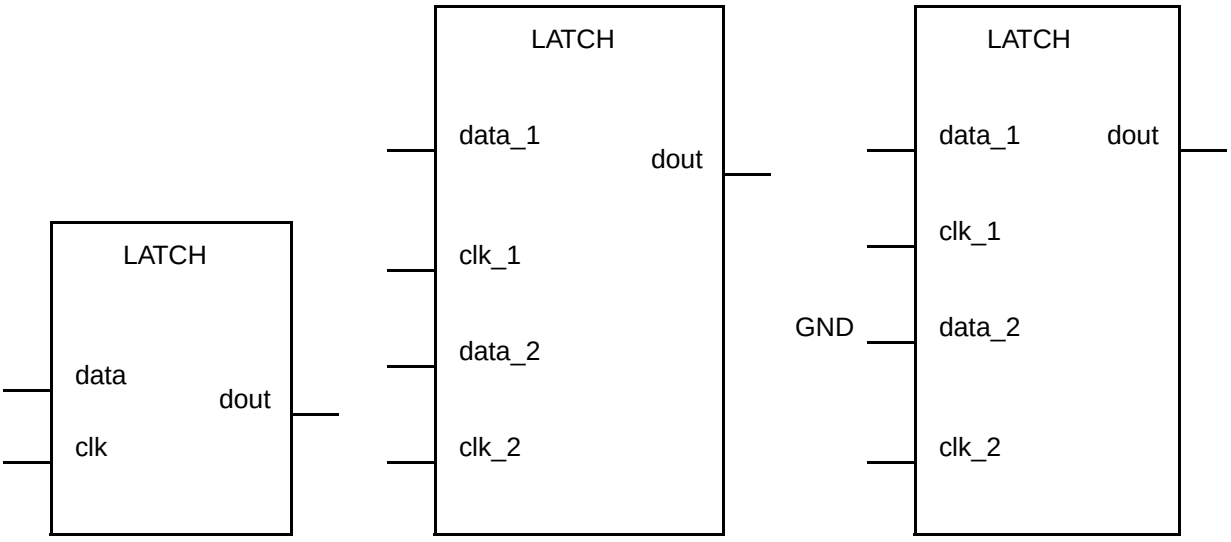
The same reasoning applies to latches which have more than two ports.

Encounter Test recognizes that a cell is a latch primitive by a cell name of LATCH or LATCH_*xxx*. Encounter Test must also be able to distinguish the function of the input pins, in other words which port the pin is associated with, and whether the pin is a clock or a data pin for that port.

Note: Encounter Test can create automatic pattern faults for LATCH primitives that model real defects more accurately than traditional pin faults. See Encounter Test Fault Model Overview in *Encounter Test: Guide 3: Faults* for more information.

LATCH Primitive Example

Figure A-5 LATCH Primitive Examples



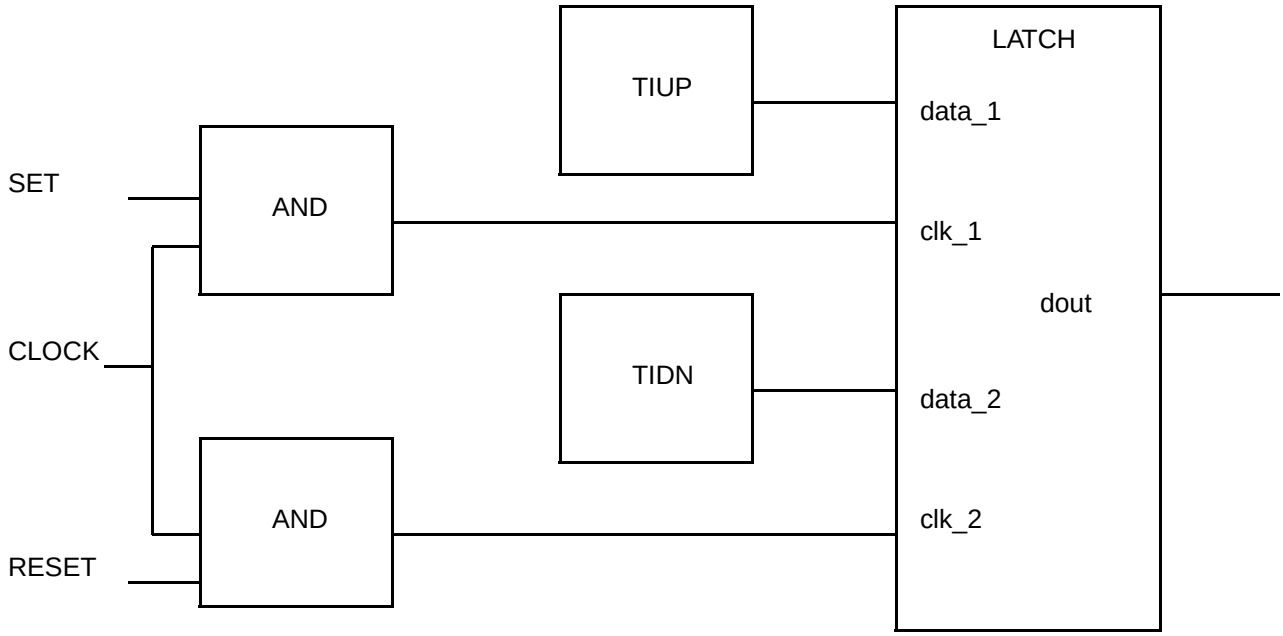
Suppose you want to model a clocked set-reset latch whose behavior is depicted in the truth table of Figure A-6.

Figure A-6 Truth Table for a Clocked Set-Reset Latch

```
set   reset  clock  latch value  |  new latch value
-------------------------------|------------------
-      -      0      v           |       v
0      0      1      v           |       v
0      1      1      v           |       0
1      0      1      v           |       1
1      1      1      v           |       X
```

The set-reset latch can be modeled using the latch primitive as shown in Figure A-7.

Figure A-7 A Clocked Set-Reset Latch Modeled with the Latch Primitive



Verilog Source

Example1: Using Encounter Test Latch Primitive

```
module clocked_srlatch(out, set, reset, clock);
output out;
input set, reset, clock;
wire clocked_set;
wire clocked_reset;
and s_and_c(clocked_set,set,clock);
and r_and_c(clocked_reset,reset,clock);
LATCH_2 sr_latch(out, 1'b1, clocked_set, 1'b0, clocked_reset);
endmodule
```

```
module LATCH_2 (DOUT, P01DATA, P01DCLK, P02DATA, P02DCLK);
output DOUT;
input   P01DATA, P01DCLK, P02DATA, P02DCLK;
endmodule
```

Note: In Example 1, the interface definition for LATCH_2 must be provided.

Example 2: Using UDP Instance

```
module clocked_srlatch(out, set, reset, clock); output out;
input set, reset, clock;
wire clocked_set;
wire clocked_reset;
and s_and_c(clocked_set,set,clock);
and r_and_c(clocked_reset,reset,clock);
latch2_udp sr_latch(out, 1'b1, clocked_set, 1'b0, clocked_reset);
endmodule


primitive latch2_udp(q, d1, c1, d2, c2);
   input d1, c1, d2, c2;
   output q;
   reg q;
   table
// d1   c1   d2   c2  : q  : q+1
    1   1    ?    0   : ?  : 1      // port 1 clock on
    0   1    ?    0   : ?  : 0
    ?   0    1    1   : ?  : 1      // port 2 clock on
    ?   0    0    1   : ?  : 0
    ?   0    ?    0   : ?  : -      // both clocks off
    1   X    ?    0   : 1  : 1      // clock at X
    0   X    ?    0   : 0  : 0
    ?   0    1    X   : 1  : 1
    ?   0    0    X   : 0  : 0
    1   1    1    1   : ?  : 1      // both clocks on
    0   1    0    1   : ?  : 0
    1   X    1    X   : 1  : 1      // both clocks X
    0   X    0    X   : 0  : 0
  endtable
endprimitive
```

Note: In Example 2, the interface definition for primitive LATCH_udp_2 is created when the UDP is converted to a gate-level model.

Flip-Flops

Encounter Test supports a flip-flop meta-primitive that can be created either from Verilog User-Defined Primitives (UDPs) or through instantiation when coding a library cell or design netlist.

The flip-flop meta-primitive acts like a regular primitive, except that it is made up of two cascading latches and other control logic as needed. The details of the meta-primitive are stored in the logic model and in reports generated by Encounter Test.

The meta-primitive is the preferred method for modeling a DFF. However, latch-based models must be used when multiple edge-sensitive ports exist or when you need to model defects more accurately within the library cell

Naming Convention

The __DFF meta-primitive is characterized by the following unique naming conventions:

- `__`*`type`*`DFF[_`*`portString`*`]`,
  - *`type`* is specified as either `r` (rising) or `f` (falling).
  - *`portString`* contains the following characters for each of the set and clear ports:

    `c` - Non-dominant, asynchronous clear port

    `C` - Dominant, asynchronous clear port

    `s` - Non-dominant, asynchronous set port

    `S` - Dominant, asynchronous set port

The upper/lower case of the port designation determines the relative dominance of the asynchronous ports. For example, the flip-flop __rDFF_sC would represent a rising edge flip-flop with an asynchronous set port and a dominant, asynchronous clear port.

Use one of the following methods to indicate that set and clear ports have no relative dominance:

- Specify two upper case characters, for example `__rDFF_SC`
- Specify two lower case characters, for example `__rDFF_cs`

Note: The edge-triggered port is always dominated by a set or clear port. The set and clear inputs to the __xDFF meta-primitive are active high.

Pins on the _DFF meta-primitive are named using the following conventions:

`Q` Output

`CLR` Asynchronous clear

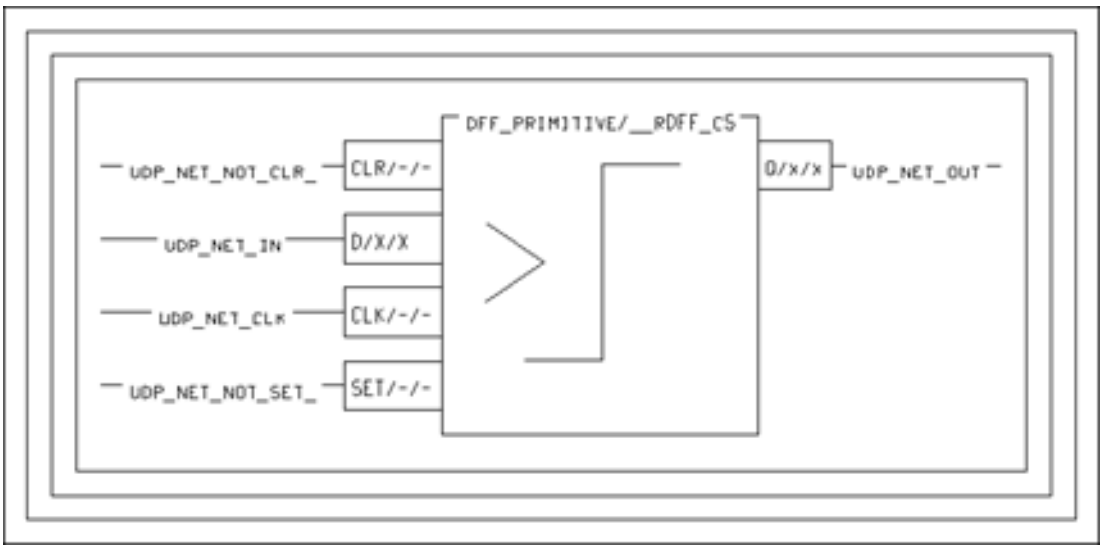
`D` Data input to edge-triggered port

CLK	Clock input to edge-triggered port
SET	Asynchronous set

Note: Only one edge-triggered port is allowed.

Flip-flop Example 1

Figure A-8 __rDFF Meta Primitive



Verilog Source: Using Encounter Test DFF metaprimitive

```
module use_dff (out, clear, data, clock, set);
   input clear, data, clock, set;
   output out;
wire clear_bar, set_bar;
not (clear_bar,clear);
not (set_bar,set);
__rDFF_CS  DFF_PRIMITIVE (.Q(out), .CLR(clear_bar), .D(data),
                          .CLK(clock), .SET(set_bar));
endmodule
```

Flip-flop Example 2

The following example shows the Verilog source for using the flip-flop primitive to define a scan dff with both set and reset (active low):

```
`celldefine
module SDFFSRX1 (Q, QN, D, SI, SE, CK, SN, RN);

output Q, QN;
input D, SI, SE, CK, SN, RN;
    udp_dff_cs I0 (Q, n1, CK, RN, SN);
    udp_mux I1 (n1, D, SI, SE);
    not     I3 (QN, Q);
endmodule
`endcelldefine

`celldefine
module udp_dff_cs (out, in, clk, clr_, set_);
    output out;
    input  in, clk, clr_, set_;
    not     clr_inv ( clr , clr_ );
    not     set_inv ( set , set_ );
    __rDFF_cs I0 ( .Q(out) , .D(in) , .CLK(clk) , .CLR(clr) , .SET(set) );

endmodule
`endcelldefine

`celldefine
module udp_mux (out, in, s_in, s_sel);
    output out;
    input  in, s_in, s_sel;
     __MUX2 I0( .DOUT(out) , .DATA0(in) , .DATA1(s_in) , .SEL(s_sel) );
endmodule
`endcelldefine
```

RAMs and ROMs

Encounter Test provides RAM and ROM primitives to model memory arrays. These primitives encapsulate the storage cell array and the address decoding. A typical technology memory cell contains additional logic beyond the function provided by the RAM and ROM primitives. This logic must be modeled around the RAM/ROM primitive to achieve the complete memory cell function.

ROM (Read Only Memory)

The ROM primitive supports the definition of multiple read ports, which consist of:

- Address Inputs
- Data Outputs
- Read Enable Signal

Primitive Name

`ROM_Axx_Dyyy_ttt`

where:

- *xx* is the number of address pins
- *yyy* is the number of data output pins
- *ttt* is a character string containing a port type for each port on the ROM. There is a maximum of 512 ports therefore this character string can be up to 512. Ports must be numbered consecutively starting with 1. The only port type that applies to a ROM is `R` for read.

Example:

```
ROM_A06_D032_RR
```

This primitive name identifies a two-port ROM with 6 address pins (64 addresses) and 32 data output pins.

Pin Names

Address Pins: *PxxxAyy*

where:

- *xxx* is the port number. Port numbers start with 1
- *yy* is the address pin number (starting with 00)

Data Output Pins: *Pxxx_yyy*

where:

- *xxx* is the port number. Port numbers start with 1
- *yyy* is the data output pin number (starting with 000)

Read Enable Pins: *PxxxREAD*

where *xxx* is the port number

ROM Contents

A ROM's data values are specified in a separate file. The file is associated with the ROM by adding a `CONTENTS` attribute to the ROM instance. The CONTENTS_FORMAT attribute

indicates how the data is formatted in the file. This attribute is also added to the ROM primitive instance. Refer to "CONTENTS Attribute" on page 133 and "CONTENTS_FORMAT Attribute" on page 133 for more information. Some examples are provided below.

ROM Function

The ROM primitive's read operation is not latched up (and is sometimes called combinational). That is, when the read enable signal is 1, the Data Outputs contain the data values at the specified Address. When the read enable signal is 0, the Data Outputs go to logic X. The read off value can be changed to 1 or 0 by adding a `READOFF` attribute to the ROM instance. Refer to "READOFF Attribute" on page 130 for a more complete description of the READOFF attribute.

The table below describes the behavior of a ROM primitive port:

Address	Read Enable	Readoff Attribute	Data Output
--	0	not specified	X
--	0	specified	readoff value
known	1	--	values from CONTENTS file
partially X	1	--	X
known	X	--	X

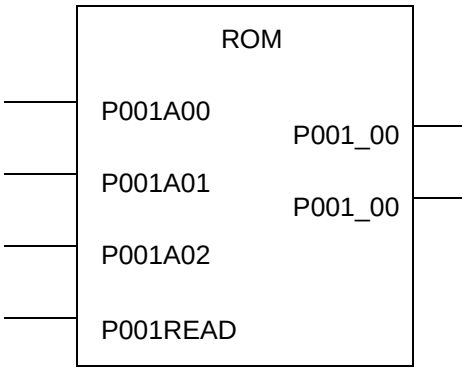
Note: If the Read Enable for multiple ports is 1 at the same time, data outputs with different values will be set to logic X.

ROM Address Bit Positions

For a ROM primitive, bit 0 is always the low-order bit of an address:

```
                P001A00   P001A01   P001A02
Address 6:         0         1         1
```

Figure A-9 ROM Primitive Example



1 Port ROM with
8 Addresses and
2 bit wide Data

Command Line Syntax:

```
build_model workdir=<directory> rompath=<directory>
```

where `rompath` is the directory containing a ROM contents file.

Verilog Syntax: Using Encounter Test ROM Primitive

```
module top (out, addr, read);
    output [0:1] out;
    input  [0:2] addr;
    input  read;
(* CONTENTS="my_contents.file" *)
    ROM_A03_D002_R rom_inst(out[0], out[1], addr[0], addr[1], addr[2], read);
endmodule

module ROM_A03_D002_R(P001_000, P001_001, P001A00, P001A01, P001A02,
                      P001READ);
    output P001_000, P001_001;
    input  P001A00, P001A01, P001A02, P001READ;
endmodule
```

Note: The ROM primitive interface definition must be provided to determine the appropriate pin ordering.

Alternative Bus Syntax (Verilog only):

```
module top (out, addr, read);
    output [0:1] out;
    input  [0:2] addr;
      input  read;
      (* CONTENTS="my_contents.file" *)
```

```
      ROM_A03_D002_R rom_inst(out, addr, read);
endmodule
module ROM_A03_D002_R(P001_[0:1], P001A[0:2], P001READ);
      output [0:1] P001_;
      input  [0:2] P001A;
      input  P001READ;
endmodule
```

Note: The ROM primitive interface busses (for example P001_[0:1]) are expanded (to pins named P001_000 and P001_001) during `build_model`.

RAM (Random Access Memory)

The RAM primitive supports four types of ports with the following characteristics:

Port Type	Address Pins	Data Input Pins	Data Output Pins	Read Enable	Write Clock
Read	X		X	X	
Write	X	X			X
Read/ Write	X	X	X	X	X
Set		X			X

Each port on a RAM primitive must contain the same number of address and data pins. Like the ROM primitive, the RAM primitive read operation is combinational. When the read enable pin is 0, the data on the output of the RAM is the readoff value. The write operation is level sensitive. That is, when the write clock is 1, the data at the specified address is written into the RAM. If the data changes while the clock is 1, the new data is written to the specified address.

Primitive Name

`RAM_Axx_Dyyy_ttt`

where:

- *xx* is the number of address pins
- *yyy* is the number of data output pins

- *ttt* is a character string containing a port type for each port on the ROM. There is a maximum of 512 ports therefore this character string can be up to 512. Ports must be numbered consecutively starting with 1. Valid port types are:
  - R - Read Only Port
  - W - Write Only Port
  - B - Read/Write Port (Both)
  - S - Set Port

Example:

The following is a primitive name for a three-port RAM with 6 address pins (64 addresses) and 32 data output pins.

```
RAM_A06_D032_RBS
```

The first port is read only, the second port is a read/write port, and the third port is a set port.

Pin Names

Address Pins: *PxxxAyy*

where:

- *xxx* is the port number. Port numbers start with 1
- *yy* is the address pin number (starting with 00)

Data Input Pins: *PxxxDyyy*

where:

- *xxx* is the port number. Port numbers start with 1
- *yyy* is the data input pin number (starting with 000)

Data Output Pins: *Pxxx_yyy*

where:

- *xxx* is the port number. Port numbers start with 1
- *yyy* is the data output pin number (starting with 000)

Read Enable Pins: *PxxxREAD*

where *xxx* is the port number. Port numbers start with 1

Write Clock Pins: *PxxxWCLK*

where *xxx* is the port number. Port numbers start with 1

RAM Function

The tables below describe the behavior of the various RAM primitive ports:

- Read Port

Address	Read Enable	Readoff Attribute	Data Output
--	0	not specified	X
--	0	specified	readoff value
known	1	--	values from CONTENTS file
partially X	1	--	X
known	X	--	X

- Write Port

Address	Write Clock	Data Input	RAM Contents	Data Output
--	0	--	No change	No change
known	1	Din	Din	No change
partially X	1,X	--	All X(1)	No change
known	X	Din	Din or X(2)	No change

- Read-Write Port

Address	Write Clock	Data Input	Read Enable	RAM Contents	Data Output
--	0	--	0	No change	X (or readoff)
known	0	--	1	No change	RAM contents
partially X	1,X	--	0	All X(1)	No change
partially X	1,X	--	1,X	All X(1)	No change
known	X	Din	0	Din or X(2)	No change
known	X	Din	1	Din or X(2)	Din or X(2)
known	X	Din	X	Din or X(2)	X
known	1	Din	1	Din	Din

- Set Port

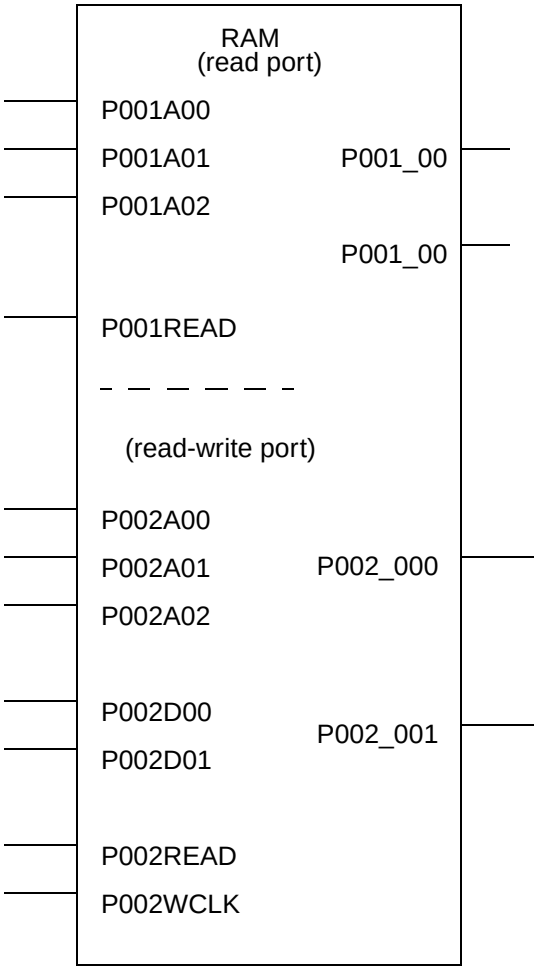
Write Clock	Data Input	RAM Contents

```
0     --    | No change
1     Din   | All Din(3)
X     Din   | Din or X(2)
```

Note:

1. All X - the entire RAM goes to X.
2. Din or X - data bits, where Din matches the RAM's contents, remain unchanged and the data bits that differ go to X.
3. All Din - All Addresses are written with Din.

Figure A-10 RAM Primitive Example



2 Port RAM with
8 Addresses and
2 bit wide Data

Verilog Syntax: Using Encounter Test RAM Primitive

```
    module top (dout1, dout2, addr1, addr2, din2, r1, r2, w2);
      output [0:1] dout1, dout2;
      input  [0:2] addr1, addr2;
      input  [0:1] din2;
      input  r1, r2, w2;
      RAM_A03_D002_RB ram_inst(dout1[0], dout1[1],
                               dout2[0], dout2[1],
                               addr1[0], addr1[1], addr1[2], r1,
                               addr2[0], addr2[1], addr2[2],
                                din2[0],  din2[1],  r2, w2);
    endmodule

module RAM_A03_D002_RB(P001_000, P001_001, P002_000, P002_001,
                       P001A00,  P001A01,  P001A02,  P001READ,
                       P002A00,  P002A01,  P002A02,
                       P002D000, P002D001, P002READ, P002WCLK);

     output P001_000, P001_001, P002_000, P002_001;
     input  P001A00, P0001A01, P001A02;
     input  P002A00, P002A01, P002A02;
     input P002D000, P002D001;
     input P001READ, P002READ, P002WCLK;
endmodule
```

Note: The RAM primitive interface definition must be provided to determine the appropriate pin ordering.

Alternative Bus Syntax (Verilog only):

```
    module top (dout1, dout2, addr1, addr2, din2, r1, r2, w2);
      output [0:1] dout1, dout2;
      input  [0:2] addr1, addr2;
      input  [0:1] din2;
      input  r1, r2, w2;
      RAM_A03_D002_RB ram_inst(dout1, dout2, addr1, r1,
                               addr2, din2,  r2,    w2);
    endmodule

module RAM_A03_D002_RB(P001_[0:1], P002_[0:1], P001A[0:2],
                         P001READ,P002A[0:2], P002D[0:1],
                         P002READ,P002WCLK);
     output [0:1] P001_, P002_;
     input  [0:2] P001A, P002A;
     input  [0:1] P002D;
     input  P001READ, P002READ, P002WCLK;
endmodule
```

Note: The RAM primitive interface busses (for example P001_[0:1]) are expanded (to pins named P001_000 and P001_001) during `build_model`.

Tips to remember when defining a RAM or ROM primitive:

- All ports except set ports must have address pins. If a single RAM or ROM is defined by several ports, each port must have the same number of address pins.
- Read and read/write ports must have data output pins. Write, read/write, and set ports must have data input pins. You must use the same number of data input pins and data output pins on every port of the RAM or ROM.
- Up to 32 address pins are supported,
- Up to 512 ports are supported.
- Up to 999 data pins are supported for each port.
- Up to 4095 output pins are supported (total of all read or r/w ports).
- Behavioral models can be transitioned to gate-level models for input to ATPG by using the Encounter Test memory modeling utility. Refer to Creating and Using a Memory Model on page 47 for more information.

To model RAMs that functionally exceed 999 data pins or 4095 output pins, you can "slice" the data pins until they fit into the primitive, then use multiple primitive slices until all the bits in the width of the RAM are covered.

RAM and ROM Attributes

There are several attributes that affect the way the RAM or ROM primitives are used in Encounter Test.

Note: The CLOCKED_FROM and CLOCKED_TO attributes apply to RAMs and ROMs as well as latches.

READOFF Attribute

The READOFF attribute defines the value that should appear on data output pins of a RAM or ROM when the read enable pin is at logic 0. The valid values for the READOFF attribute are 0, 1, or X. The default value for READOFF is X. The READOFF attribute should appear on the instance of the RAM or ROM primitive.

LPD Attribute

When multiple write ports attempt to write into the same address of a RAM at the same time, a write conflict occurs. Normally, Encounter Test ATPG would not create such tests on purpose, but this may happen by accident. Encounter Test simulators will by default compare the data values being written into the RAM and keep those bit values that match on all ports writing to that address and substitute an unknown value (X) wherever corresponding bits along the data width of the RAM conflict (0 versus 1 or known versus X). The only other write conflict behavior available for a RAM primitive is to use the LPD attribute as described below.

The LPD attribute identifies a multi-port RAM which obeys a lower-port dominance rule. If the attribute is specified on a multi-port RAM, and two or more ports attempt to write to the same address of the RAM at the same time, then the lower port (the port with the higher port number) will win. The lower-port dominance behavior is guaranteed only when all of the ports that attempt to write to the same address are clocked with the same clock pulse. If there are two different clocks controlling the write to the same address of a RAM at the same time, then the clock pulse which falls last will win.

The only value for the LPD attribute is `YES`. The LPD attribute should appear on the instance of a RAM primitive.

Any behavior other than the default writing of X where the data bits conflict or using lower port dominance must be implemented by logic surrounding the RAM primitive. For example, if behavior is needed that writes X into the selected RAM address on all bits of the width instead of just conflicting bits, address compare logic can be implemented that would force (at least) one of the ports being written to go to all X on its data pins.

Specifying Incomplete RAMs and ROMs

The size of a RAM or ROM primitive is defined with the number, *n*, of address pins on each port. This, in turn, implies that words in the memory starting at address 0, and ending at one less than two to the power of *n* are all available for accessing. However, there are some implementations of RAMs and ROMs which limit their address space. Encounter Test uses three attributes to define these limits.

LOWADDR Attribute

The LOWADDR attribute defines the lowest address which is accessible on the RAM or ROM. If it is not specified, the default lowest address is address 0. The LOWADDR attribute must have a value that is a number between zero and the highest possible address. It appears on the instance of the RAM or ROM primitive.

HIGHADDR Attribute

The HIGHADDR attribute defines the highest address which is accessible on the RAM or ROM. If it is not specified, the default high address is one less than two to the power of `n`. The HIGHADDR attribute must have a value that is a number between zero and the highest possible address. It appears on the instance of the RAM or ROM primitive.

MASKOUT Attribute

The MASKOUT attribute defines a single logic value that will appear on all data output pins of a RAM or ROM if the address specified is less than the LOWADDR or greater than the HIGHADDR attribute specified for the memory primitive. The default value of MASKOUT is X. The valid values for MASKOUT are 0, 1, or X. For ROM primitives only, the MASKOUT attribute can also be specified as OVERLAY, which indicates that addresses outside the valid range of addresses will access the values stored at the address obtained by inverting the high-order bit of the invalid address. The MASKOUT attribute appears on the instance of the RAM primitive.

Specifying ROM Contents

Since ROMs are read-only memories, Encounter Test must have a way of determining what the contents of the ROM memory is when the chip is being tested. The contents of a ROM primitive are defined to Encounter Test through a special ROM contents file. The definition of a usage of a ROM primitive must contain a CONTENTS attribute, which defines the file name of the ROM contents file. When the design is read into Encounter Test, the directory which contains the ROM contents file must be specified. Encounter Test then reads the ROM contents file to understand the values that are in ROM memory.

Notes:

1. The specification of ROM contents is not mandatory but can help raise test coverage in ATPG or design verification.
2. An error message is issued if not enough input data is specified and the ROM contents file is padded with "X".
3. An error message is issued if too much input data is specified and the extra data is ignored.

The ROM is specified to Build Model using the `build_model ROMPATH=` command. Refer to the following for related information:

- build_model in the *Encounter Test: Reference*: *Commands*

- Paths Setup in the *Encounter Test: Reference: GUI*
- "ROM (Read Only Memory)" on page 121

CONTENTS Attribute

The CONTENTS attribute defines the name of the contents file for a ROM. The CONTENTS attribute should appear on the instance of a ROM primitive.

CONTENTS_FORMAT Attribute

The CONTENTS_FORMAT attribute defines the style of ROM contents syntax to expect when reading the file. Valid values are 1, 2, 3, or 4. It defaults to 1 if not specified. The CONTENTS_FORMAT attribute should appear on the instance of a ROM primitive.

For flexibility, Encounter Test allows four different formats for specifying ROM contents in the ROM contents file. Any of the four syntax styles can be used, but the CONTENTS_FORMAT attribute should clearly identify which style (1, 2, 3, or 4) is to be expected when the file is read by Encounter Test. If CONTENTS_FORMAT is not specified, the original Encounter Test format (1) is assumed. The following sections describe the four valid formats for specifying a ROM contents file.

ROM Contents File Syntax 1

If the CONTENTS_FORMAT attribute on the ROM instance is either specified or defaulted to 1, the ROM contents file is expected to conform to the following syntax:

- The first line of the file must contain a number, w indicating the data bit width of the ROM.
- The second line must contain a number, m, the number of addresses in the ROM. This number must be a power of two, and defines the number of lines to follow.
- The third and following lines of the file contain the contents of the ROM for a specific address.
  - The third line of the file represents the contents of address 0,
  - the fourth line of the file represents the contents of address 1,

    and so on, up to the...
  - m+2 line, which represents address m-1, the last addressable word in the ROM.

These lines contain *w* characters, where *w* is the data bit width of the ROM. Each character can be a 0, 1, or X to define the value of the corresponding ROM bit. No leading or embedded blanks are allowed. The first character of the line corresponds to ROM bit 0, the second to ROM bit 1, and so on.

Note: For addresses which are not implemented, the entire line should be set to the value to appear on the output pins when this address is read.

The following is an example of the ROM contents file for a ROM with three data pins, and two address pins (or four addressable words).

Figure A-11 A ROM Contents File Example

```
3
4
000
001
010
011
```

ROM Contents File Syntax 2

If the CONTENTS_FORMAT attribute on the ROM instance is specified as 2, the ROM contents file is expected to conform to a Verilog style syntax as described below:

- Line comments begin with "//" and continue through the end of that line.
- General comments begin with "/*" and continue until the "*/" comment end delimiter. General comments can span multiple lines.
- All white space is ignored, including spaces, new-lines, tabs, form-feeds and under-score "_" characters.
- The first valid character found (0, 1 or X) is assigned to ROM bit 0 of the lowest valid address (usually address 0, but can be different if the LOWADDR attribute was specified).
- Subsequent values are assigned to successively numbered bits of that address (if any) until all bits of that address have been assigned a value. The next value is then assigned to ROM bit 0 of the next higher address.
- The values are assigned to successively higher addresses until the highest valid address is filled (as specified by HIGHADDR) or as defaulted (2 to the power of *n* minus 1).

The following is an example of the ROM contents file for a ROM with four data pins, and three address pins (or eight addressable words); only addresses 0 through 5 are valid.

Figure A-12 Another ROM Contents File Example

```
/*************************************************************************
** This ROM contents example requires the CONTENTS format              **
**  property be specified to a value of 2. It is for a ROM that has    **
**  3 address inputs, but only implements the addresses 0 through 5.   **
** It is presumed that the HIGHADDR property is specified to a value   **
**  of 5. The ROM is 4 bits wide. Note that the first value for any address **
**  is the value for bit 0 and the last value for that address is for bit 3. **
*************************************************************************/

// This bit appears on the output pin P001_000
//   |
//   |
    1111  //  This word is assigned to address 0
    1110  1101  1100  1011   // Words for addresses 1 - 4
    1010  //  This is for address 5, the last valid address
//      |
//      |
//      This bit appears on the output pin P001_003

/*end of ROM contents file example  */
```

ROM Contents File Syntax 3

If the CONTENTS_FORMAT attribute on the ROM instance is specified as 3, the ROM contents file is expected to conform to a HEX syntax as described below:

- Line comments begin with "//" and continue through the end of that line.
- General comments begin with "/*" and continue until the "*/" comment end delimiter. General comments can span multiple lines.
- All white space (spaces, new-lines, tabs, form-feeds) and under-score "_" characters are ignored.
- If the data width of the ROM is a multiple of 4:
  - The lowest valid address (either 0 or LOWADDR) is filled as follows:

- The first valid HEX character(0,1,2,3,4,5,6,7,8,9,A,a,B,b,C,c, D,d,E,e,F,f) is converted to binary and:
  - The left-most bit is assigned to data bit 0,
  - Successive bits are assigned to data bit 1, data bit 2, and data bit 3.
- Subsequent HEX characters are converted to binary and assigned to successively numbered bits of this address (if any) until all bits of this address have been assigned a value.

- Each subsequent address is filled by processing subsequent HEX characters in the same manner until the highest valid address (either HIGHADDR or defaulted to ((2 to the power of *n*) minus 1) is filled.

  Note: *n* is equal to the number of address lines.

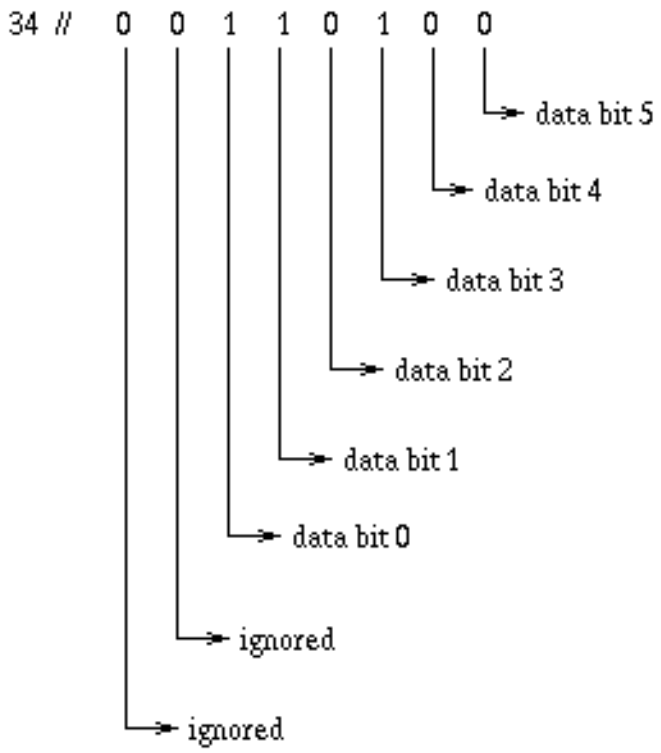
- If the data width of the ROM is NOT a multiple of 4:
  - The lowest valid address (either 0 or LOWADDR) is filled as follows:
    - The first valid HEX character is converted to binary and:
      - If the data width divided by 4 has a remainder of 3, the left-most bit is ignored and the remaining 3 bits are assigned to data bit 0, data bit 1 and data bit 2.
      - If the data width divided by 4 has a remainder of 2, the 2 left-most bits are ignored and the remaining 2 bits are assigned to data bit 0 and data bit 1.
      - If the data width divided by 4 has a remainder of 1, the 3 left-most bits are ignored and the remaining bit is assigned to data bit 0.

      Note: It is expected that the leading ignored bits will be zero. If not, a warning message is issued.
    - Subsequent HEX characters are converted to binary and assigned to successively numbered bits of this address (if any) until all bits of this address have been assigned a value.
  - Each subsequent address is filled by processing subsequent HEX characters in the same manner until the highest valid address (either HIGHADDR or defaulted to ((2 to the power of *n*) minus 1) is filled.

    Note: *n* is equal to the number of address lines.

The following ROM contents file example, for `CONTENTS_FORMAT=3`, is data for one address when the ROM has a data width of 6.

Figure A-13 ROM Contents File, CONTENTS_FORMAT=3, data width of 6



ROM Contents File Syntax 4

If the CONTENTS_FORMAT attribute on the ROM instance is specified as 4, the ROM contents file is expected to conform to a HEX syntax as described below:

- Line comments begin with "//" and continue through the end of that line.
- General comments begin with "/*" and continue until the "*/" comment end delimiter. General comments can span multiple lines.
- All white space (spaces, new-lines, tabs, form-feeds) and under-score "_" characters are ignored.
- If the data width of the ROM is a multiple of 4:
  - The lowest valid address(either 0 or LOWADDR) is filled as follows:
    - The first valid HEX character(0,1,2,3,4,5,6,7,8,9,A,a,B,b,C,c,D,d E,e,F,f) is converted to binary and:
      - The left-most bit is assigned to the highest numbered data bit (width-1).

- Successive bits are assigned to data bit (width-2), data bit (width-3), and data bit(width-4).

❍ Subsequent HEX values are converted to binary and assigned to decreasing numbered bits of this address (if any) until all bits down to data bit 0 of this address have been assigned a value.

❑ Each subsequent address is filled by processing subsequent HEX characters in the same manner until the highest valid address (either HIGHADDR or defaulted to ((2 to the power of *n*) minus 1) is filled.

Note: *n* is equal to the number of address lines.

■ If the data width of the ROM is NOT a multiple of 4:

❑ The lowest valid address (either 0 or LOWADDR) is filled as follows:

❍ The first valid HEX character is converted to binary and:

- If the data width divided by 4 has a remainder of 3, the left-most bit is ignored and the successive 3 bits are assigned to data bit(width-1), data bit(width-2), and data bit(width-3).

- If the data width divided by 4 has a remainder of 2, the 2 left-most bits are ignored and the successive 2 bits are assigned to data bit(width-1) and data bit(width-2).

- If the data width divided by 4 has a remainder of 1, the 3 left-most bits are ignored and the remaining bit is assigned to data bit (width-1).

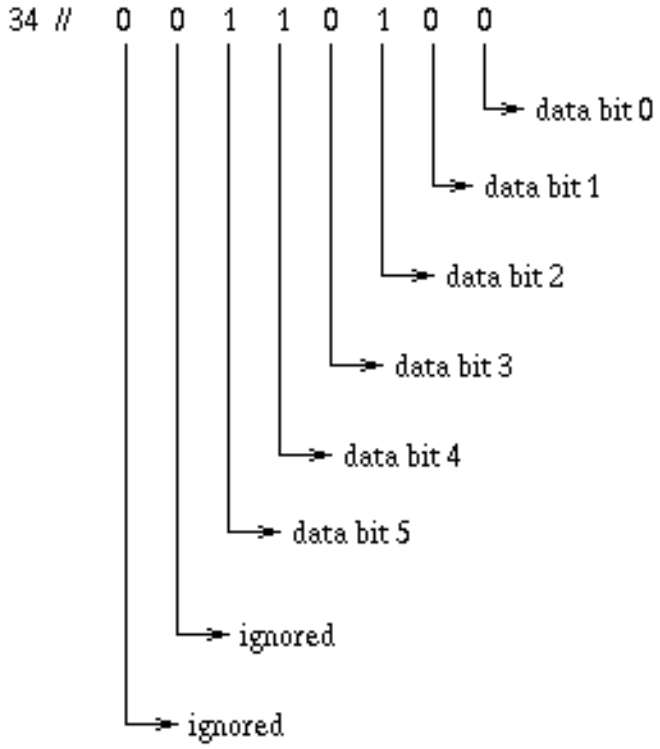
Note: It is expected that the leading ignored bits will be zero. If not, a warning message is issued.

❍ Each subsequent address is filled by processing subsequent HEX characters in the same manner until the highest valid address (either HIGHADDR or defaulted to ((2 to the power of *n*) minus 1) is filled.

Note: *n* is equal to the number of address lines.

The following ROM contents file example, for `CONTENTS_FORMAT=4`, is data for one address when the ROM has a data width of 6.

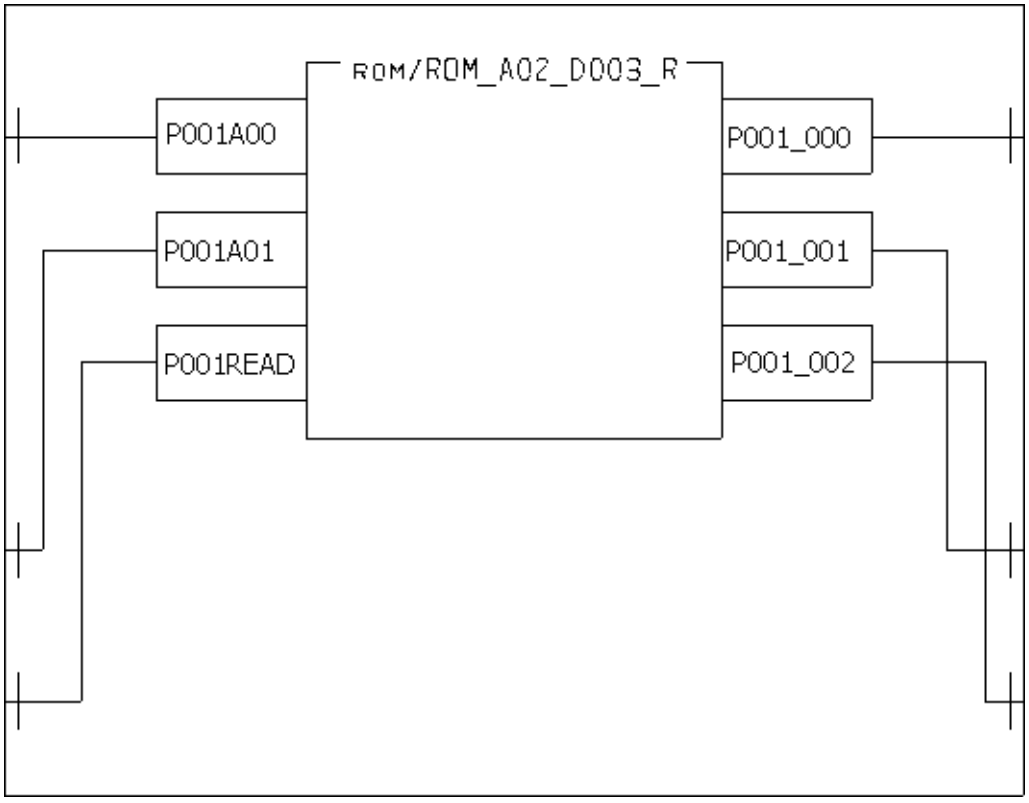
Figure A-14 ROM Contents File, CONTENTS_FORMAT=4, data width of 6



Example of ROM with Contents

The following example shows the definition of a three by four ROM.

Figure A-15 Example ROM with Contents Schematic



This usage of the ROM primitive calls for file ROM0123 to contain the contents of the ROM. Assume the file ROM0123 is defined as in "ROM Contents File Syntax 1" on page 133. Then, if we put a logic 1 on pin RD, and a logic 1 on pin A0 and a logic 0 on pin A1, we are asking what the value of word 1 is in the ROM. Here, Encounter Test assigns a logic 0 to pin Q0, a 0 to pin Q1, and a 1 to pin Q2.

Note: If the ROM contents cannot be found during Build Model, Encounter Test will presume the ROM contents are completely unknown (logic X).

Example of ROM with CONTENTS

The following example shows the usage of the three by four ROM defined in the preceding example.

Command Line Syntax

```
build_model workdir=<directory> ? rompath=<directory>
```

Verilog Syntax

The following is a sample format for the ET parser:

```
// ****************************************************************
// ROM interface definition required for pin binding
// ****************************************************************

module ROM_A03_D004_R (P001_000, P001_001, P001_002, P001_003,
                       P001A00, P001A01, P001A02, P001READ);

 output P001_000, P001_001, P001_002, P001_003;
 input  P001A00, P001A01, P001A02;
 input  P001READ;

endmodule

// ****************************************************************
//   Module that contains ROM instance
// ****************************************************************

module rom_book_1 (out_bus, addr_bus, read_enab);

 output [3:0] out_bus;
 input  [2:0] addr_bus;
 input  read_enab;

 ROM_A03_D004_R rom_i1

    //! CONTENTS="rom_a03_d004_r.contents"
    //! CONTENTS_FORMAT="2"

  ( .P001_000 (out_bus[0]),
    .P001_001 (out_bus[1]),
    .P001_002 (out_bus[2]),
    .P001_003 (out_bus[3]),
    .P001A00 (addr_bus[0]),
    .P001A01 (addr_bus[1]),
    .P001A02 (addr_bus[2]),
    .P001READ (read_enab) ) ;

endmodule
```

The following is a sample IEEE 2001 format for the IEEEstandard parser:

```
(* CONTENTS="rom_a03_d004_r.contents" *)
(* CONTENTS_FORMAT="2" *)
ROM_A03_D004_R rom_i1
   (.P001_000 (out_bus[0]),
    .P001_001 (out_bus[1]),
    .P001_002 (out_bus[2]),
    .P001_003 (out_bus[3]),
    .P001A00 (addr_bus[0]),
    .P001A01 (addr_bus[1]),
    .P001A02 (addr_bus[2]),
    .P001READ (read_enab));
```

Three-state Logic

This section describes the Encounter Test primitives that create three-state logic values, and discusses examples of cells that use three-state logic.

Three-State Driver (TSD) Primitive

The TSD primitive has two inputs and a single output. The inputs consist of a data input and an enable input. If the enable input is a logic 0, then the output is Z. If the enable input is a logic 1, the TSD functions like the BUF Primitive Function.

Note:

1. When the enable pin of a TSD is at logic 1, the input data is treated as a Boolean value instead of a three-state value. Therefore, weak values become strong, and Z becomes X.
2. Encounter Test assigns a dot function of T to all TSD outputs.

Primitive Name

```
TSD
```

or

```
TSD_string
```

Pin Names

Input Pins: DATA, ENABLE

Output Pin: any name

TSD Function

```
              DATA
  TSD|  0  1  X  Z  L  H  W
  ---|--------------------
E  0|  Z  Z  Z  Z  Z  Z  Z
N  1|  0  1  X  X  0  1  X
A  X|  X  X  X  X  X  X  X
B  Z|  X  X  X  X  X  X  X
L  L|  Z  Z  Z  Z  Z  Z  Z
E  H|  0  1  X  X  0  1  X
   W|  X  X  X  X  X  X  X
```

Note: Encounter Test can create automatic pattern faults for TSD primitives that model real defects more accurately than traditional pin faults. See Encounter Test Fault Model Overview in *Encounter Test: Guide 3: Faults* for more information.

Example Using TSD Primitive

```
//*****************************************************************
// Top Level Module
//*****************************************************************
module top(out, in, enable);
  output out;
  input in, enable;
  TSD i1(out, in, enable);
endmodule
//*****************************************************************
// Encounter Test TSD Primitive
//*****************************************************************
module TSD (DOUT, DATA, ENABLE);
  input DATA, ENABLE;
  output DOUT;
endmodule
```

Note: Encounter Test TSD interface definition is required for pin binding.

Example Using Verilog bufif1 Primitive

```
//*****************************************************************
// Top Level Module
//*****************************************************************
module top(out, in, enable);
  output out;
  input in, enable;
  bufif1 i1(out, in, enable);
endmodule
```

NFET Primitive

The NFET primitive is almost the same as the TSD primitive, with the exception that weak signals are passed through the NFET primitive when the enable pin is at logic 1. The NFET primitive has two inputs and a single output.The inputs consist of a data input and an enable input. If the enable input is a logic 0, then the output is Z. If the enable pin is a logic 1, NFET moves the data input to the output.

Note:

1. Encounter Test assigns a dot function of T to all NFET outputs.
2. Encounter Test treats the NFET primitive as a unidirectional gate. The values on the output pin do not affect the values on the input pin.

Primitive Name

```
NFET
```

or

```
NFET_string
```

Pin Names

Input Pins: DIN, NGATE

Output Pin: any name

NFET Function

```
              DIN
  NFET  |  0  1  X  Z  L  H  W
  ------|--------------------
NGATE  0|  Z  Z  Z  Z  Z  Z  Z
  N    1|  0  1  X  Z  L  H  W
  G    X|  X  X  X  Z  W  W  W
  A    Z|  X  X  X  Z  W  W  W
  T    L|  Z  Z  Z  Z  Z  Z  Z
  E    H|  0  1  X  Z  L  H  W
       W|  X  X  X  Z  W  W  W
```

Example 1: Using NFET Primitive

```
//*********************************************************************
// Top Level Module
//*********************************************************************
module top(out, in, enable);
  output out;
  input in, enable;
  NFET i1(out, in, enable);
endmodule
//*********************************************************************
// Encounter Test NFET Primitive
//*********************************************************************
module NFET (DOUT, DIN, NGATE);
  input DIN, NGATE;
  output DOUT;
endmodule
```

Note: Encounter Test NFET interface definition is required for pin binding.

Example 2: Using Verilog nmos Primitive

```
//*********************************************************************
// Top Level Module
//*********************************************************************
```

```
module top(out, in, enable);
  output out;
  input in, enable;
  nmos (out, in, enable);
endmodule
```

PFET Primitive

The PFET primitive is almost the same as the NFET primitive, with the exception that the enable pin allows data to pass through the PFET device when it is at logic 0 instead of logic 1.

The PFET primitive has two inputs and a single output. The inputs consist of a data input and an enable input. If the enable input is a logic 1, then the output is Z. If the enable pin is a logic 0, the PFET moves the data input to the output.

Note:

1. Encounter Test assigns a dot function of T to all PFET outputs.
2. Encounter Test treats the PFET primitive as a unidirectional gate. The values on the output pin do not affect the values on the input pin.

Primitive Name

```
PFET
```

or

```
PFET_string
```

Pin Names

Input Pins: DIN, PGATE

Output Pin: any name

PFET Function

```
             DIN
  PFET| 0  1  X  Z  L  H  W
  ----|-------------------
     0| 0  1  X  Z  L  H  W
  P  1| Z  Z  Z  Z  Z  Z  Z
  G  X| X  X  X  Z  W  W  W
  A  Z| X  X  X  Z  W  W  W
  T  L| 0  1  X  Z  L  H  W
  E  H| Z  Z  Z  Z  Z  Z  Z
     W| X  X  X  Z  W  W  W
```

Example 1: Using PFET Primitive

```
//*****************************************************************
// Top Level Module
//*****************************************************************
module top(out, in, enable);
  output out;
  input in, enable;
  PFET i1(out, in, enable);
endmodule
//*****************************************************************
// Encounter Test PFET Primitive
//*****************************************************************
module PFET (DOUT, DIN, NGATE);
  input DIN, NGATE;
  output DOUT;
endmodule
```

Note: Encounter Test PFET interface definition is required for pin binding.

Example 2: Using Verilog pmos Primitive

```
//*****************************************************************
// Top Level Module
//*****************************************************************
module top(out, in, enable);
  output out;
  input in, enable;
  pmos (out, in, enable);
endmodule
```

RESISTOR Primitive

The RESISTOR primitive converts a strong signal to a weak signal. It has a single input value and a single output value.

Primitive Name

```
RESISTOR
```

or

```
RESISTOR_string
```

Pin Names

Input Pin: any name

Output Pin: any name

Resistor Function

```
INPUT   |  OUTPUT
--------|--------
  0     |   L
  1     |   H
  X     |   W
  Z     |   Z
  L     |   L
  H     |   H
  W     |   W
```

Example

```
//*********************************************************************
// Top Level Module with RESISTOR instance
//*********************************************************************

module top (OUT, IN);
  input IN;
  output OUT;
  RESISTOR i1(OUT,IN);
endmodule

//*********************************************************************
// Encounter Test RESISTOR Primitive Interface
// Port names for the RESISTOR primitive are arbitrary
//*********************************************************************

module RESISTOR (OUT, IN);
  input IN;
  output OUT;
endmodule
```

Note: Encounter Test RESISTOR interface definition is required for pin binding.

Three-State Contention

Hard contention occurs when two sources of a three-state net are driving opposing hard logic values onto the net (for example, 0 vs. 1 or 1 vs. 0).

Soft contention occurs when one source of a three-state net drives a hard logic value (0 or 1) onto the net, and another source drives the net with a hard unknown value (X), therefore, producing a potential hard conflict.

Note:

1. If a net with a logic value of 0 is dotted with the output of a TSD, NFET, or PFET, whose data is 0, and whose enable is X, then the three-state net must have the value of hard 0. The same applies for a logic value of 1.

2. Encounter Test provides keywords for use during ATPG to avoid or detect three-state contention.

Internal Three-State Termination

The three-state termination deals with the Boolean logic value that appears on a net when the enable pins on all TSD, NFET, and PFET primitives that drive the net are off. In other words, when no device drives the net, termination determines the value to be seen at the sinks of the net.

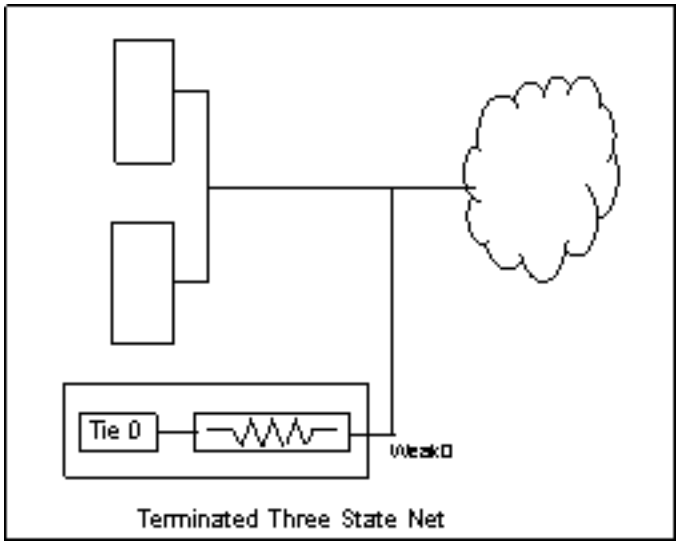
Note: "External Three-State Termination" on page 158 discusses a three-state net driven by a pin at the top level of the hierarchy.

The following are two ways to model the termination of internal three-state nets:

- Add a Pull-up/Pull-down resistor

The simplest method to terminate a three-state net is to add a pull-up or pull-down resistor to the net. A pull-up or pull-down resistor can be modeled using a verilog primitive or with a TIUP or TIDN primitive that feeds a RESISTOR primitive.

Figure A-16 Terminated Three-State Net



Example 1: Verilog Pulldown Primitive

```
module top(...);
...
unit1 inst1(..., three_state_net, ...);
unit2 inst2(..., three_state_net, ...);
pulldown three_state_net);
...
endmodule
```

Example 2: User Defined Pulldown Cell

```
module top(...);
...
unit1 inst1(..., three_state_net, ...);
unit2 inst2(..., three_state_net, ...);
pulldown_cell inst3(three_state_net);
...
endmodule


module pulldown_cell(out);
output out;
resistor(out,1'b0);
endmodule
```

- Add a TERM attribute

Encounter Test also supports the TERM attribute on pins or usage pins on a three-state net. The value of the TERM attribute can be either 0 or 1. Encounter Test checks to make sure that all the TERM attributes on a single net have the same value.

Example 4: TERM Attribute on a Usage Pin

```
module top(...);
...
unit1 inst1(..., three_state_net, //! TERM="0"
           ...);
unit2 inst2(..., three_state_net, //! TERM="0"
           ...);
...
endmodule
```

Note: On using Verilog strength specifications:

1. `build_model` converts the weak0/weak1 and pull0/pull1 strength specifications on nets or gate instances in Verilog source to WEAK values.
2. The strong and supply strength identifiers are converted to STRONG values.
3. To treat the pull0/pull1 strength identifier as a STRONG value in the model, specify `PULL=strong` on the `build_model` command line.

Note: Encounter Test models the TERM attribute as a pull-up or pull-down resistor. That is, the TERM attribute is modeled as an extra source to the net that contains a weak 0 or a weak 1.

Example Three-State Cell Descriptions

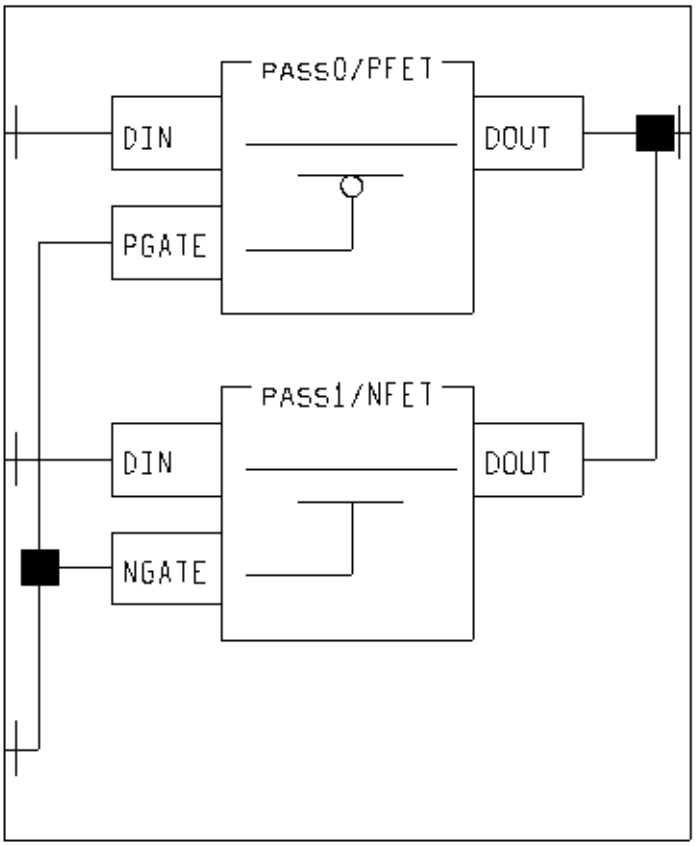
The following examples demonstrate modeling using three-state logic.

Pass-Gate MUX

The pass-gate multiplexor demonstrates the use of transistor primitives in a cell library.

Note: It is possible to define the same function that a pass-gate MUX performs using only combinational primitives in Encounter Test. (See Figure A-4 on page 113 for an example.) The technology library coder must weigh the advantage of the more exact pass-gate MUX design against the simpler, easy-to-test combinational MUX design.

Figure A-17 Pass Gate MUX Schematic



Verilog Source

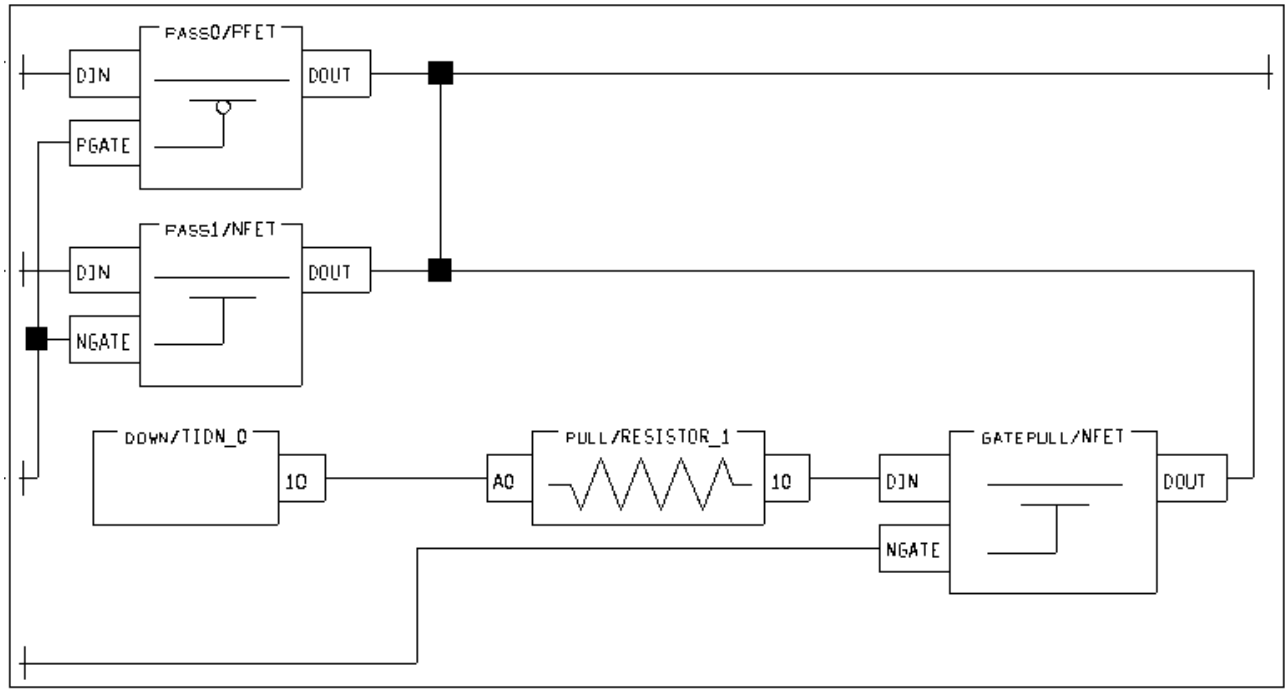
```
module PFET(DOUT, DIN, PGATE);
output DOUT;
input DIN, PGATE;
endmodule

module NFET(DOUT, DIN, NGATE);
output DOUT;
input DIN, NGATE;
endmodule
```

Note: Encounter Test primitives require module interface definition for pin binding.

Pass-Gate MUX with Test Circuitry

The pass-gate MUX defined above is a simple design, but contains some inherently untestable logic. Resolution of a high-impedance state on the output net to a known value is required to improve testability. The standard method for resolving high-impedance is through a pull-up or pull-down resistor. However, pull-up and pull-down resistors may slow the output time-to-rise or time-to-fall, and may require extra current (and affect IDDq tests). One way to resolve these problems is to provide a gated pull-up or pull-down resistor, controlled by a test input. The following cell illustrates this logic.

Figure A-18 Pass-Gate MUX with Test Circuitry Schematic



Verilog Source

```
module top(out, d0, d1, sel, test);
output out;
input d0, d1, sel, test;
PFET pass0(out,d0,sel);
NFET pass1(out,d1,sel);
TIDN down (tie0);
RESISTOR_1 pull(weak0,tie0);
NFET gatepull(out,weak0,test);
endmodule

module PFET(DOUT, DIN, PGATE);
output DOUT;
input DIN, PGATE;
endmodule

module NFET(DOUT, DIN, NGATE);
output DOUT;
input DIN, NGATE;
endmodule


module RESISTOR(\10 , A0);
output \10 ;
input A0;
endmodule

module TIDN(\10 );
output \10 ;
endmodule
```

Note: Encounter Test primitives require module interface definition for pin binding.

Modeling Other Logic

Keeper Devices

Encounter Test provides support for keeper devices. A keeper device is a weak feedback on a three-state net (see Figure A-19 on page 153). The feedback may be gated such that it does not always contribute to the three-state dot (see Figure A-20 on page 153). In some designs, a certain type of keeper may be able to hold only a specific logic value (0 or 1) and cannot hold the other logic value (see Figure A-21 on page 154 and Figure A-22 on page 154). Although it is not considered good practice, it is possible for a single three-state net to have more than one keeper device attached to it. In such cases, the keepers may be redundant and could reduce the testability of the design.

Figure A-19 A keeper device attached to a three-state net

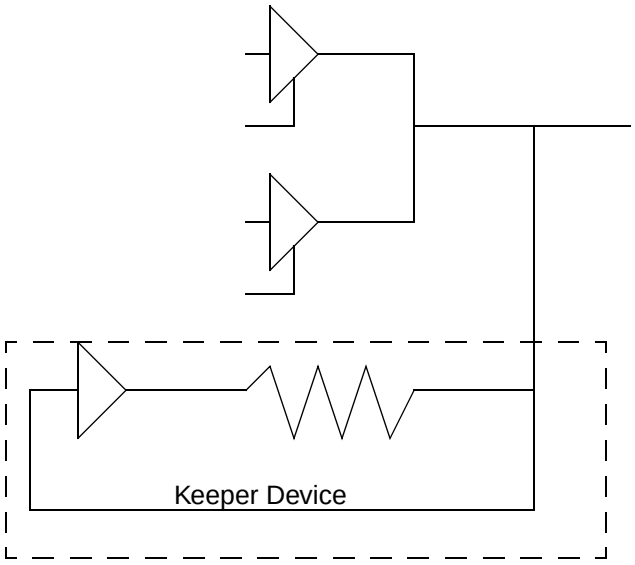


Figure A-20 A keeper device with enable input

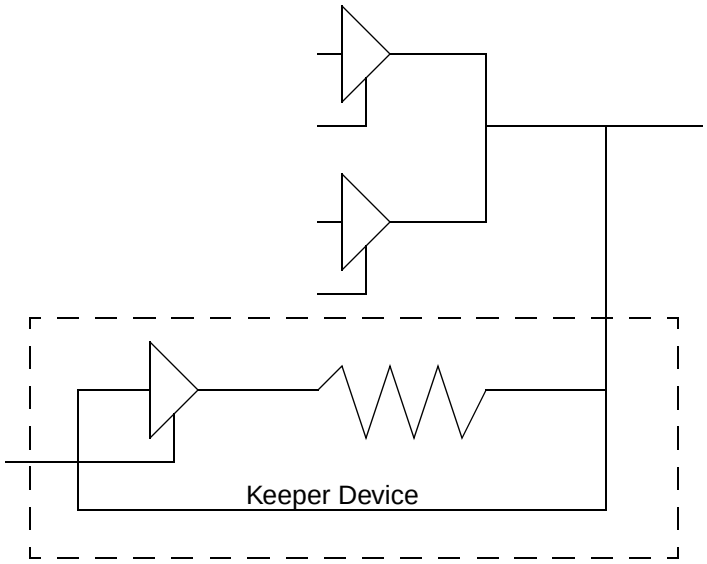


Figure A-21 A keeper device that keeps only a logic 1

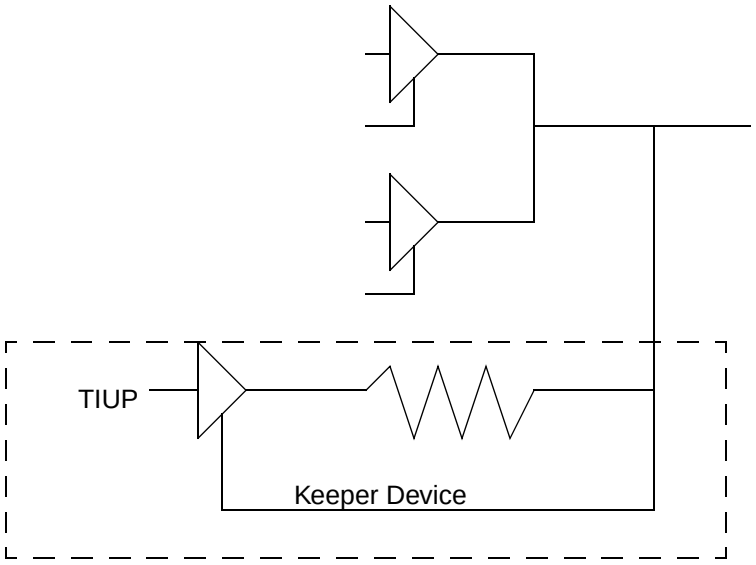
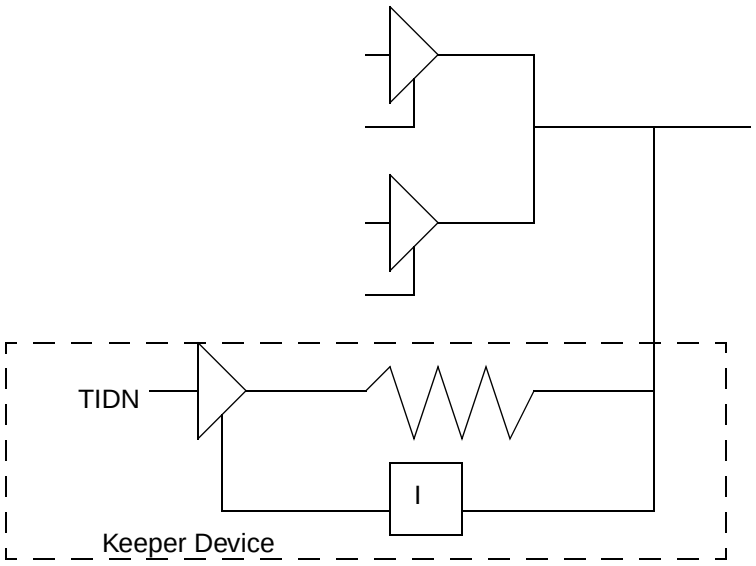


Figure A-22 A keeper device that keeps only a logic 0



Keeper devices may be used in a design for several reasons:

- As a (cheap) latch

  If a net is driven by a three-state device and timing constraints are such that the net has to hold a valid signal for a longer time than the strong devices are enabled, then a keeper device can be used as a cheaper solution than a (clocked) latch.

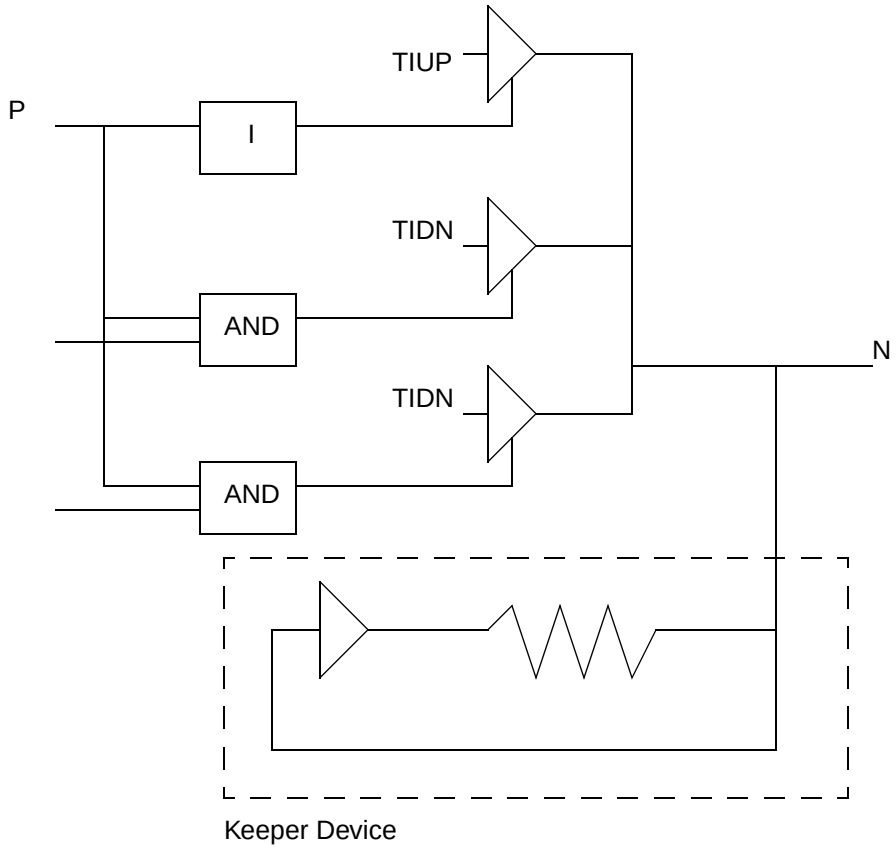
- To avoid high current

  If a three-state net is allowed to float, the downstream receiving logic may see an indeterminate value. This value may be interpreted as a zero (0) by some transistors and as a one (1) by other transistors. This can create paths between Vdd and ground, resulting in high current (burn-out) conditions. Placing a weak device on the three-state net eliminates any indeterminate logic values. A weak constant value (termination) may achieve similar results, but a weak feedback tends to use less power. The feedback draws current through the resistor only while the net is switching, but a terminating resistor draws current whenever the net is being actively driven to the value opposite of the termination. Also, a keeper is less likely than a terminating resistor to slow the switching time of the net.

- To model dynamic logic

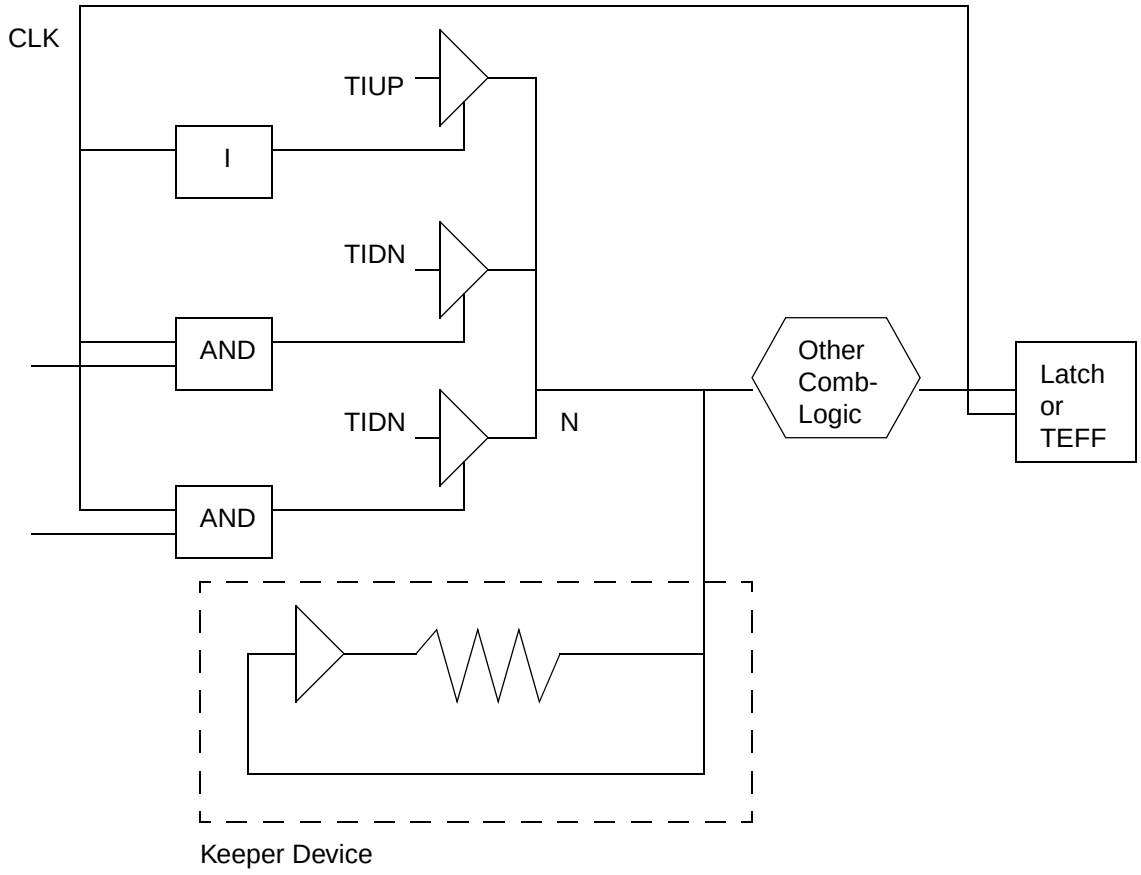
  In many dynamic logic designs, a (three-state) node is actively driven to a logic value (typically 1) during a precharge phase. The precharge phase is followed by an evaluation phase in which zero or more strong devices are enabled to “discharge” the node (typically to 0). If the logic wants the resolved node value to stay at the precharged (1) value, the enable inputs all stay off. In order for the precharged value to stay on the node, the node may have either an attached capacitor or a weak feedback. Encounter Test models this kind of design with weak feedback (keeper devices), as shown in Figure A-23.

Figure A-23 A Precharge Design with Keeper. Node N is Precharged to 1 when Signal P is Zero.



In most cases, the precharge and evaluate phases are synchronized by the system clock. (See Figure A-24 for an example.) Typically the precharge turns ON when the clock is OFF at the downstream latches. When the clock turns ON, precharge ends and evaluate begins. In these designs it is important for the evaluate stage to complete its ripple through the logic up to the latches before the clock turns back OFF at the latches, with enough extra time for the minimum setup and hold times for the latches. Dynamic logic such as this is inherently racy and requires highly accurate timing to ensure it all works. When the clock turns OFF, the precharge begins for the next cycle, and the data must be latched up before the precharge effects ripple through the logic to change the latch data input. Encounter Test zero delay simulators assume that the clock going OFF at the latch will win this race; however, unit delay simulators will have trouble if the models are not designed for unit delay simulation.

Figure A-24 A clocked precharge Design. Node N is precharged to 1 when Signal CLK is zero.



The keeper devices described, such as those shown in Figure A-19 on page 153 through Figure A-24 on page 157, are essentially non-scannable latches. These keeper devices are often not controlled by system clocks. In many cases, these “latches” will be asynchronous - they will not be strongly clocked with a known, stable state. This exposes simulators to potentially racy designs which may be susceptible to glitches.

I/O Cells

Most I/O cells in a technology library can be mapped directly to an Encounter Test buffer or inverter primitive. This section deals with exceptions to this rule.

Pin Directions

Specification

Each pin defined in Verilog must have an explicit pin direction. The pin must be either an input pin, an output pin, or a bidirectional pin. Encounter Test uses the explicit pin directions to check the semantics of the logic model, and to control the display of the schematic. Other Encounter Test software does not rely on the explicit pin directions as they process the model. Instead, the software examines all pins connected to a net to determine each pin's direction. Encounter Test can tolerate incorrect specification of pin directions in the model source, but incorrect pin directions make analysis and problem solving more difficult.

There are two cases where Encounter Test must use the explicit pin directions:

- Pins at the top level of the hierarchy.
- Pins at the boundary of a cell which is not a primitive, but which has no contents.

In these cases, Encounter Test uses the pin directions specified in the model source. If the pin directions in the model source are incorrect, Encounter Test will produce incorrect results.

Overriding CIO Attribute

The CIO attribute may be attached to an output pin as an alternate method for defining a bidirectional pin. The preferred method for defining a bidirectional pin is with the use of the pin-directions. Encounter Test will change the pin direction of an output pin to a bidirectional pin for an output pin which has the CIO attribute with a value of YES.

The CIO attribute can also be used to convert a bidirectional pin into an output only pin. Here, the value for the CIO attribute must be NO.

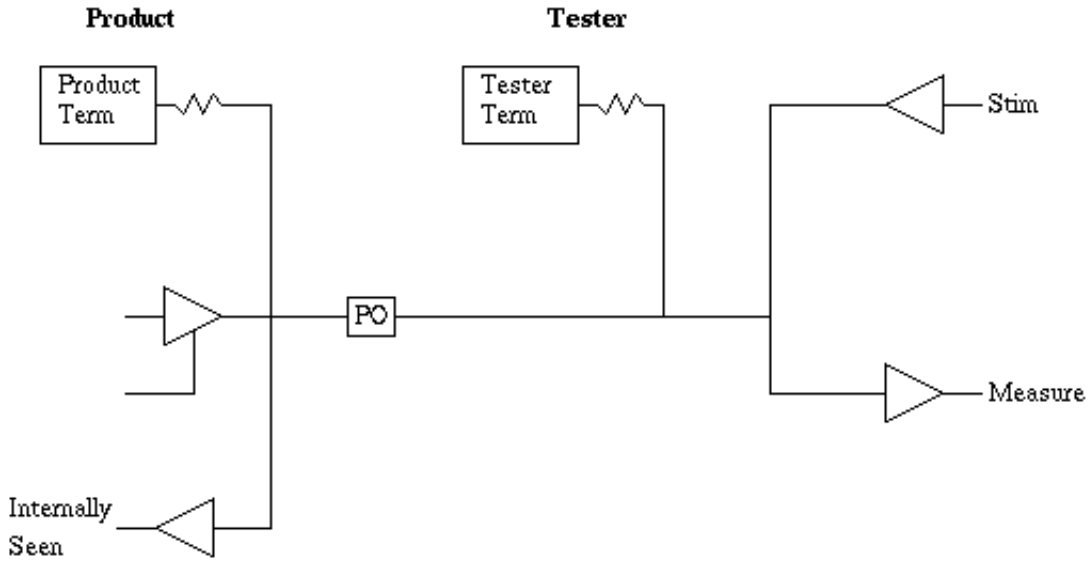
External Three-State Termination

Three-state termination of internal nets is discussed in "Internal Three-State Termination". The more difficult three-state termination questions arise when the three-state net connects to a bidirectional pin at the top level of the hierarchy. In this case, the function of the tester probe which contacts the bidirectional pin must be taken into account. The tester probe can provide both a new source and a new sink to the three-state net. In the worst case, a three-state net can have sources from:

- One or more TSDs or NFETs or PFETs
- Termination on the product from a pull-up or pull-down resistor on the product

- Termination in the tester from a pull-up or pull-down resistor attached to the tester probe
- A tester stimulated value

Figure A-25 External Three-State Termination Example



Given all of these sources, Encounter Test uses the following ground-rules to determine the value of the three-state net.

- If the tester is stimulating the bidirectional pin, it will be contributing a hard logic value which will drive the net.
- A hard value may also come from an enabled TSD, NFET or PFET on the product if one or more of these primitives are enabled.
- If there is more than one hard logic value driving the net, and the hard values are different, a three-state burn-out will result. Options in the Encounter Test vector generators prevent three-state burn-outs.
- If no hard logic values are driving the net, the value at the sinks of the net depends on either the product termination or the tester termination.
- The Tester Description Rule (TDR) defines whether the product termination takes precedence or the tester termination takes precedence. For more information, see Tester Description Rule (TDR) File Syntax in *Encounter Test: Guide 2: Testmodes*.

Encounter Test determines which value the tester uses to terminate the net from either of the following:

- If there is a TTERM attribute on the usage of a pin driving the net, or on the bidirectional pin at the top level, the TTERM value is used to calculate the tester provided termination value.

  Note: For historical reasons, if a net is terminated with a TTERM value, the value becomes a hard logic value rather than a weak logic value.

- If there is no TTERM attribute, Encounter Test uses the termination options specified as a parameter when the test patterns are generated to determine the tester termination value.

Once the value of the three-state net is determined, Encounter Test determines the values at the sinks of the net, as follows:

- For sinks of the net internal to the product, the TDR specifies whether the tester probe can corrupt a weak logic value. If the TDR states `INTERNALLY SEEN=X`, then all weak values on the three-state net become logic X at internal sources. If `INTERNALLY SEEN=TERM` appears in the TDR, then the value of the net propagates to internal sinks of the net.
- The other sink of the net is the test probe. The logic value that Encounter Test predicts at the test probe depends on what the tester can measure, as defined in the TDR. If the TDR says `MEASURE HIGH_Z = TERM`, then weak values on the three-state net are converted to strong values at the tester probe. If the TDR says `MEASURE HIGH_Z = Z`, then weak values on the three-state net are converted to logic Z at the tester probe.
- The termination support in Encounter Test allows the tool to generate tests for the enable stuck faults on the off-chip driver.

Note: In all cases described above, you must know if the target tester and the tester software can support the termination values and `measure HIGHZ` statement.

HF_MIN_PULSE_WIDTH Attribute

The `HF_MIN_PULSE_WIDTH` attribute can be specified as an instance attribute in a technology library cell. The value of this attribute indicates the minimum pulse width that will correctly propagate through this cell. The `HF_MIN_PULSE_WIDTH` attribute is typically used to indicate that a specific technology cell can support a narrower clock pulse width than the technology generally supports. When Encounter Test creates timed test patterns, the minimum pulse width for a test which uses this cell is determined by the maximum of:

- The functional pulse width determined by the topology

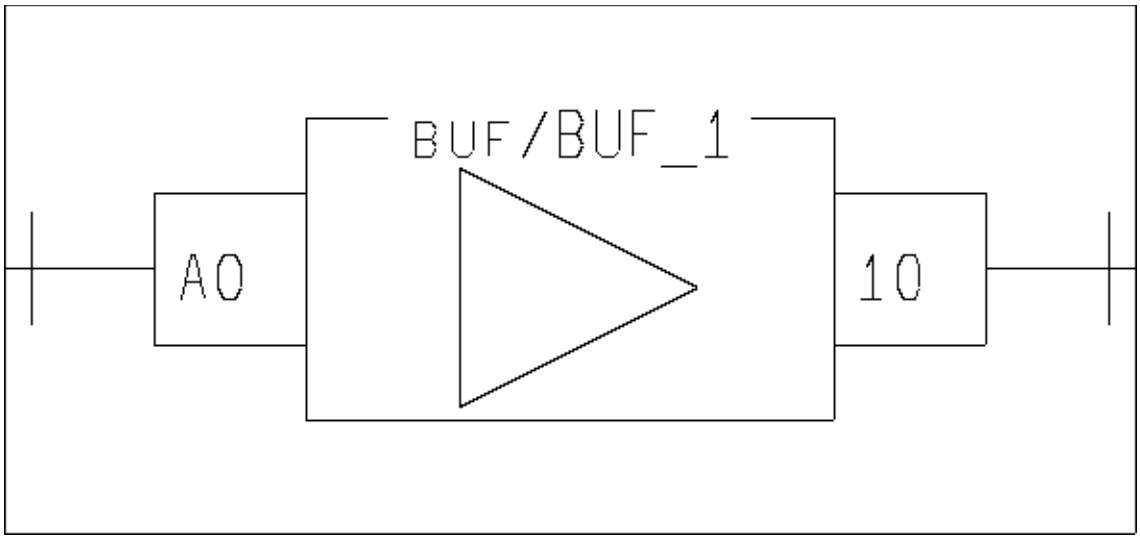
- The `HF_MIN_PULSE_WIDTH` of the cell
- The `HF_MIN_PULSE_WIDTH` of the tester (TDR) if it exists or the `MIN_PULSE_WIDTH` of the tester (TDR) if `HF_MIN_PULSE_WIDTH` does not exist.

Example I/O Cells

The following examples demonstrate the more common I/O cell modeling techniques in Encounter Test.

Two-State Receiver

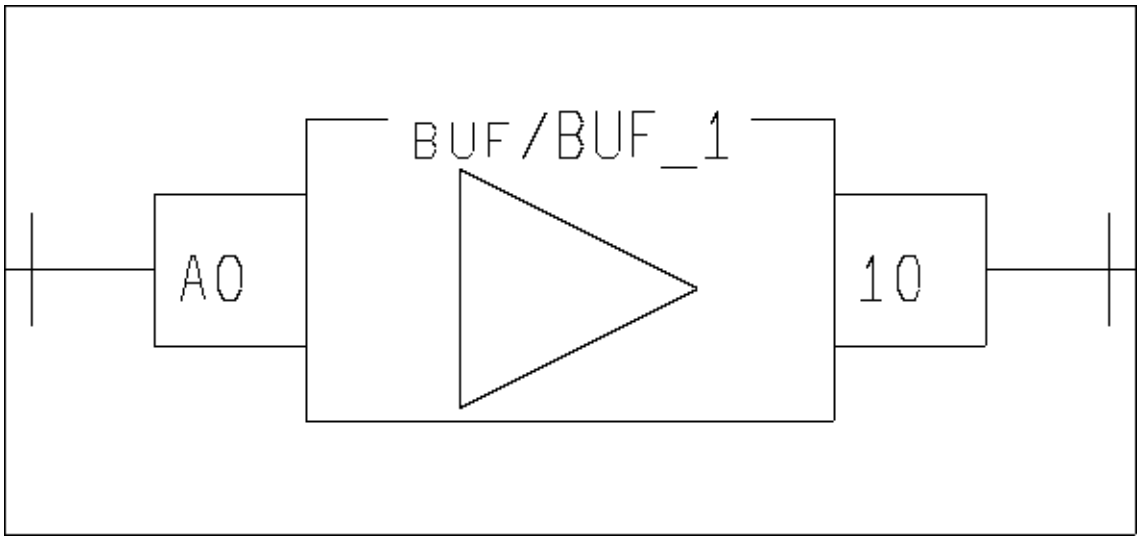
Figure A-26 Two-State Receiver Schematic



Refer to Two-State Receiver Source to view the corresponding source.

Two-State Driver

Figure A-27 Two-State Driver Schematic

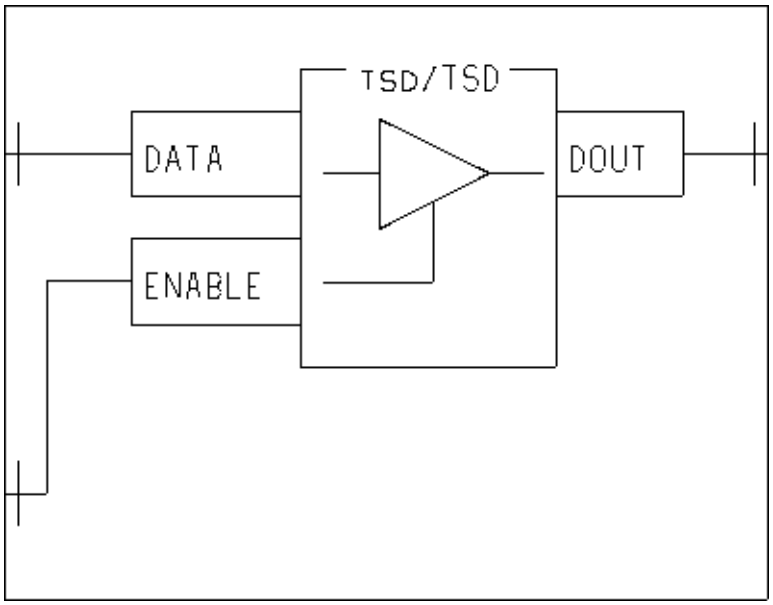


Refer to Two-State Driver Source to view the corresponding source.

Three-State Driver

This example shows a typical three-state off-chip driver. The two inputs are D (data) and E (enable), and the output is Y. The typical three-state driver is a direct map onto the TSD primitive.

Figure A-28 Three-State Driver Schematic



Refer to Three-State Driver Source to view the corresponding source.

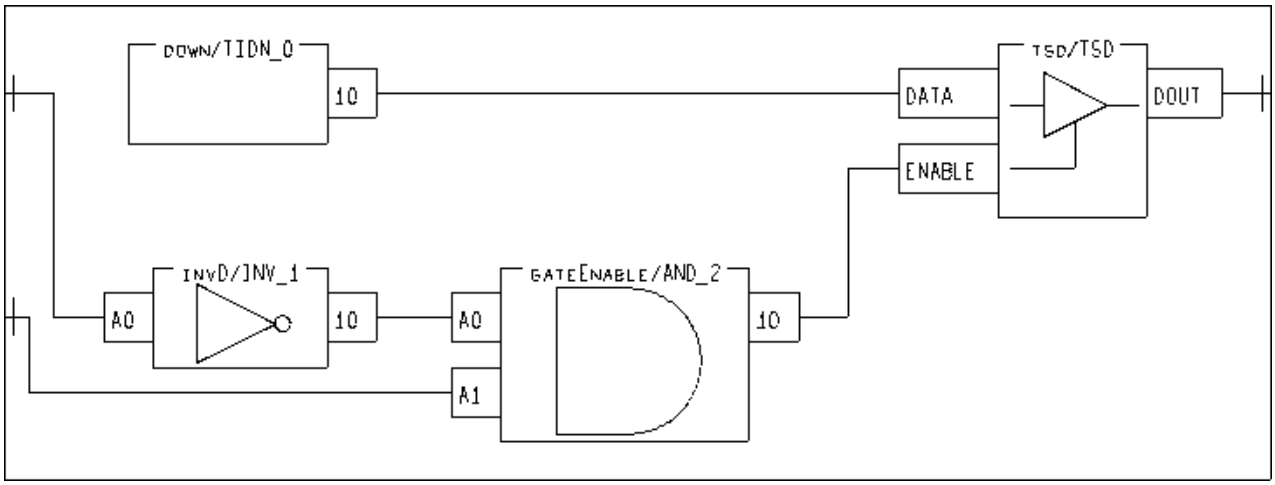
Open Drain Driver

The open drain off-chip driver can be modeled in Encounter Test with a TSD which can produce either high impedance or a hard logic 0. A truth table for this cell is as follows.

D	E	Y
-	0/L	Z
0/L	1/H	0
1/H	1/H	Z
X/W/Z	1/H	X
-	X/W/Z	X

Note: The TTERM attribute on the output pin of this cell requires a tester termination to 1. This tester termination allows the tester to distinguish between a hard 0 and high impedance, and makes this driver testable.

Figure A-29 Open Drain Driver Schematic



Refer to Open Drain Driver Source to view the corresponding source.

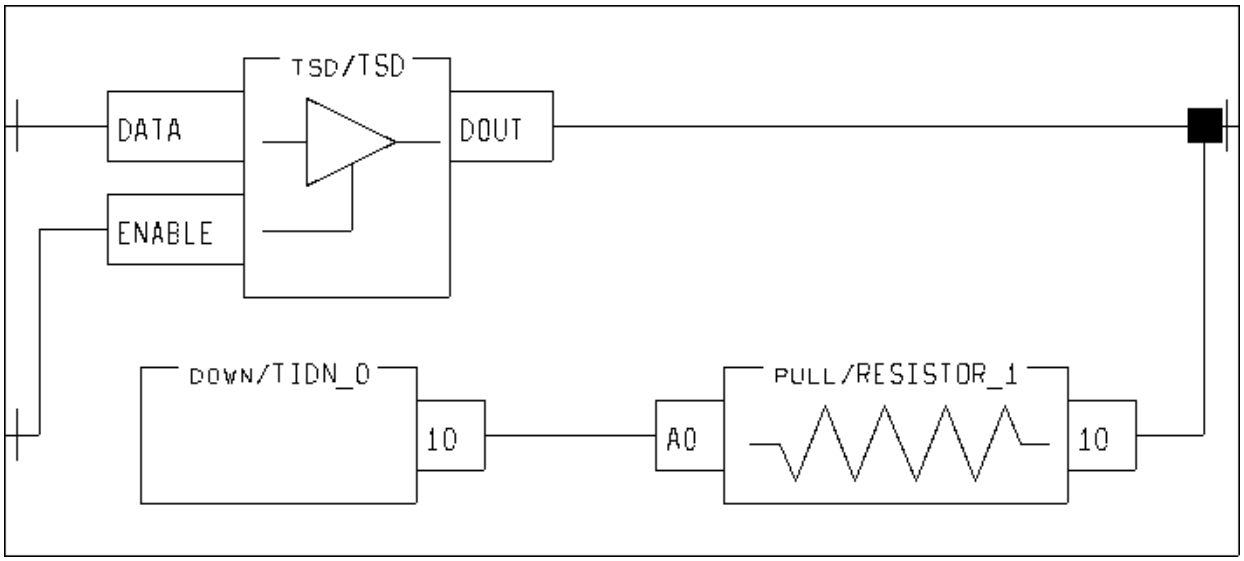
Three-State Driver with Pull-Down Resistor

A pull-down resistor on a three-state driver is modeled by simply adding a resistor fed by a net whose source is ground. A truth table for this cell is as follows:

```
                D
     | 0  1  X  Z  L  H  W
-----|-------------------
    0| L  L  L  L  L  L  L
    1| 0  1  X  X  0  1  X
    X| X  X  X  X  X  X  X
    E  Z| X  X  X  X  X  X  X
    L| L  L  L  L  L  L  L
    H| 0  1  X  X  0  1  X
    W| X  X  X  X  X  X  X
```

Note: The L (weak 0) in the truth table represents the pull down value that is calculated at the cell. What is seen outside the cell depends on whether the net connected to the pin goes to other cells, or goes to a bidirectional pin at the top level of the hierarchy. The signal may be interpreted as either a 0, a Z, or an X depending on where the signal goes, and the capabilities of the tester.

Figure A-30 Three-State Driver with Pull-Down Schematic



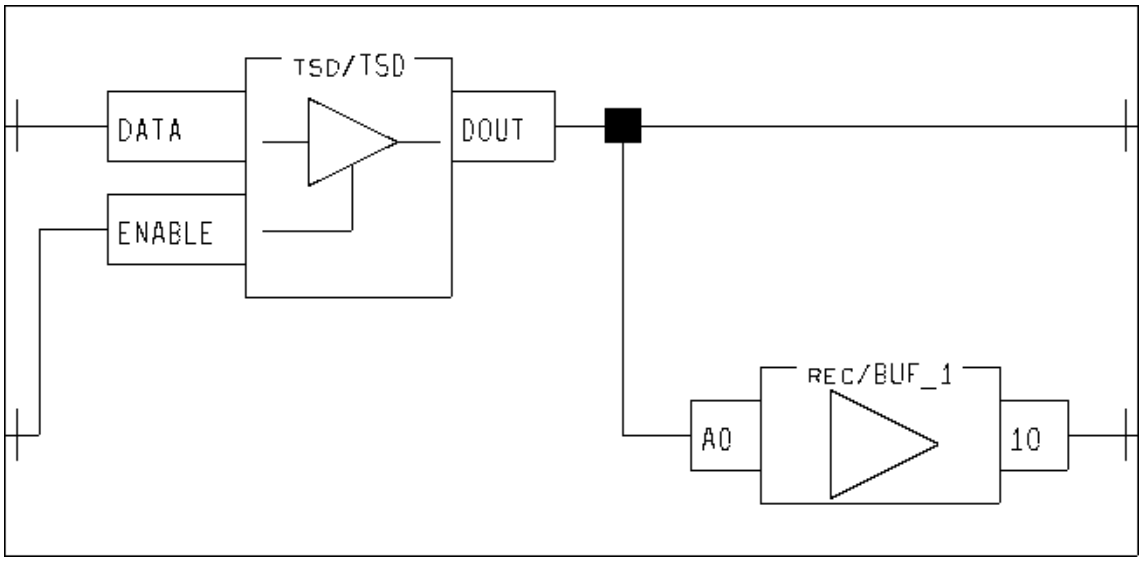
Refer to Three-State Driver with Pull-Down Source to view the corresponding source.

Bidirectional Driver

The bidirectional driver cell demonstrates a bidirectional pin at the cell level. The cell can act as either a driver or a receiver depending on the value of E. When E is a 1, the outputs of the cell, Y and Z, are driven by the input to the cell D. When E is a 0, then the bidirectional input Y drives the output Z. A truth table for this cell is as follows:

D	E	Y	Z
-	0/L	0	0
-	0/L	1	1
-	0/L	X	X
-	0/L	L	0
-	0/L	H	1
-	0/L	W	X
-	0/L	Z	X
0/L	1/H	0	0
1/H	1/H	1	1
X/W/Z	1/H	X	X
-	X/W/Z	X	X

Figure A-31 Bidirectional Driver Schematic



Refer to Bidirectional Driver Source to view the corresponding source.

Termination Values

Table A-2 on page 166 shows how Encounter Test determines the correct values to use during simulation of three-state primary input and output logic. Descriptions of the values are listed after the table.

Table A-2 Resolution of Internal and External Values due to Termination

Product Parameters								Results	
Product Term	**Product Tester Term**	**Global Tester Term**	**TDR Term Dominance**	**TDR Internally Seen**	**TDR Measure High_Z**	**Pin is Active**	**Pin is Tester Contacted**	**Internal Value Propagated**	**External Value Measure**
None	None	None	N/A	N/A	N/A	No	N/A	X	N/A
None	None	None	N/A	N/A	N/A	Yes	Yes	X	Z
None	None	0/1	N/A	Term	N/A	No	Yes	0/1	N/A
None	None	0/1	Tester	N/A	N/A	No	Yes	0/1	N/A

Product Parameters								Results	
Product Term	**Product Tester Term**	**Global Tester Term**	**TDR Term Dominance**	**TDR Internally Seen**	**TDR Measure High_Z**	**Pin is Active**	**Pin is Tester Contacted**	**Internal Value Propagated**	**External Value Measure**
None	None	0/1	Product	X	Term	No	No	X	N/A
None	None	0/1	N/A	Term	Term	Yes	Yes	0/1	0/1
None	None	0/1	Tester	N/A	Term	Yes	Yes	0/1	0/1
None	None	0/1	Product	X	Term	Yes	Yes	X	0/1
None	None	0/1	N/A	Term	Z	Yes	Yes	0/1	Z
None	None	0/1	Tester	N/A	Z	Yes	Yes	0/1	Z
None	None	0/1	Product	X	Z	Yes	Yes	X	Z
None	0/1	N/A	N/A	N/A	N/A	No	No	X	N/A
None	0/1	N/A	N/A	N/A	N/A	No	Yes	0/1	N/A
None	0/1	N/A	N/A	N/A	N/A	Yes	Yes	0/1	0/1
0/1	None	None	N/A	Term	N/A	No	N/A	0/1	N/A
0/1	None	None	N/A	X	N/A	No	N/A	X	N/A
0/1	None	None	N/A	Term	Term	Yes	Yes	0/1	0/1
0/1	None	None	N/A	Term	Z	Yes	Yes	0/1	Z
0/1	None	None	N/A	X	Term	Yes	Yes	X	0/1
0/1	None	None	N/A	X	Z	Yes	Yes	X	Z
0/1	None	0/1	N/A	Term	N/A	No	N/A	0/1	N/A
0/1	None	0/1	Tester	N/A	N/A	No	Yes	0/1	N/A
0/1	None	0/1	N/A	X	N/A	No	No	X	N/A
0/1	None	0/1	Product	X	N/A	No	N/A	X	N/A
0/1	None	0/1	N/A	Term	Term	Yes	Yes	0/1	0/1
0/1	None	0/1	Tester	N/A	Term	Yes	Yes	0/1	0/1
0/1	None	0/1	Product	X	Term	Yes	Yes	X	0/1

Product Parameters								Results	
Product Term	**Product Tester Term**	**Global Tester Term**	**TDR Term Dominance**	**TDR Internally Seen**	**TDR Measure High_Z**	**Pin is Active**	**Pin is Tester Contacted**	**Internal Value Propagated**	**External Value Measure**
0/1	None	0/1	N/A	Term	Z	Yes	Yes	0/1	Z
0/1	None	0/1	Tester	N/A	Z	Yes	Yes	0/1	Z
0/1	None	0/1	Product	X	Z	Yes	Yes	X	Z
0/1	None	1/0	Tester	N/A	N/A	No	Yes	1/0	N/A
0/1	None	1/0	N/A	Term	N/A	No	No	0/1	N/A
0/1	None	1/0	Product	Term	N/A	No	N/A	0/1	N/A
0/1	None	1/0	N/A	X	N/A	No	No	X	N/A
0/1	None	1/0	Product	X	N/A	No	N/A	X	N/A
0/1	None	1/0	Tester	N/A	Z	Yes	Yes	1/0	Z
0/1	None	1/0	Tester	N/A	Term	Yes	Yes	1/0	1/0
0/1	None	1/0	Product	Term	Z	Yes	Yes	0/1	Z
0/1	None	1/0	Product	Term	Term	Yes	Yes	0/1	0/1
0/1	None	1/0	Product	X	Z	Yes	Yes	X	Z
0/1	None	1/0	Product	X	Term	Yes	Yes	X	0/1

- None

  The value for this parameter was specified as none or there is no product termination.

- 0/1, 1/0

  Values can be either 0 or 1, however order can be important when termination values differ between product and tester termination

- Tester

  Tester is the value for this parameter.

- Product

  Product is the value for this parameter.

- Term

  Termination is the value for this parameter.

- N/A

  Not Applicable. Due to other options, this value is not used in determining the results.

- Product Term

  The value of this parameter is based on attributes found in the design during Build Model. "TERM" can be specified on pins in the netlists or assumed when using resistive pull-ups or pull-downs in the design. The TERM attribute signifies that the PRODUCT itself contains termination on this pin. In the following example, Product Term would be 0 for all parts that use bufif1. Valid options are "none, 0, or 1". If the TERM attribute is not specified on a pin, it defaults to "none".

  ```
  bufif1 tsd
      ( Y //! TERM = "0"
        , D , E ) ;
  ```

- Product Tester Term

  The value of this parameter is based on attributes found in the design during Build Model. "TTERM" can be specified on pins in the netlists that will be used during simulation. The TTERM attribute indicates that the product itself does not have termination but it is required that the tester provides it. In the following example, Product Tester Term would be 0 for all parts that use notif1. Valid options are "none, 0, or 1". If the TTERM attribute is specified on a pin, it defaults to "none".

  ```
  notif1 tsd
       ( Y //! TTERM = "0"
         , D , E ) ;
  ```

- Global Tester Term

  This option is specified during Logic Test Generation and Simulation . Global Termination is the selection of which tester termination capability to use during a specific ATPG experiment. The value selected must be one of the TDR-specified tester capabilities or else a message is issued. "globalterm" indicates the termination to be applied on three-state primary output pins and bi-directional pins when the internal three-state driver goes to high-impedance. `globalterm=either` allows the use of either 1 or 0 termination when attempting to resolve a fault. If both zero and one are specified in the TDR, the default for globalterm is either.

  ```
  TDR Ex:  TERMINATION
                      ZERO, ONE
  Ex: create_logic_tests... globalterm=none ...
  ```

- TDR Term Dominance

  The value of this parameter is defined in the TDR. It indicates whether tester supplied termination should be applied to pins which already have product termination, and if so, which will dominate. Dominance applies only when the tester can provide termination. It is optional. Valid options are "tester or product". If no DOMINANCE attribute is specified in the TDR, it defaults to "tester".

  TESTER indicates that tester-supplied termination is applied to all three-state output pins regardless of any product-supplied (via the TERM attribute) value for a pin. This is the default if

  DOMINANCE is not specified. PRODUCT indicates that tester-supplied termination is applied only to pins without product-supplied termination.

  ```
  Ex:  DOMINANCE = TESTER
  ```

- TDR Internally Seen

  The value of this parameter is defined in the TDR. It specifies whether the resolved termination value for an external three-state net should be propagated into the internal (receiver) logic for bidirectional three-state pins, or that an unknown, pessimistic assumption should be made. Valid options are "X or termination". The default is to use termination if there is any.

  TERMINATION specifies that internal logic should use the terminated value for a high impedance (Z) on an external three-state net. This is the default if INTERNALLY_SEEN is not specified. Note that the termination value can be supplied either by the product and/or the tester. If they both supply termination, the TDR DOMINANCE specification identifies how to resolve the termination when the values conflict.

  X specifies that even if an external three-state net has termination on it, a pessimistic assumption should be made that forces a high impedance (Z) to be seen as an unknown (X) value by the internal (receiver) logic. This always applies to product termination; however, tester termination is considered strong enough to propagate internally unless the TDR specifies `DOMINANCE=product`, in which case tester termination is considered to be weak.

  ```
  Ex:  INTERNALLY_SEEN = X
  ```

- TDR Measure High_Z

  The value of this parameter is defined in the TDR. This is used to specify how high-impedance (Z) values are to be measured by the tester. Valid options are "term or Z". There is no default value and a value is required for each TDR.

  TERM specifies that the product or tester termination should be used to convert high_Z values into logic zero or one values.

Z specifies that regardless of product or tester termination, if all three-state drivers feeding a product I/O pin are inhibited, the measure response should be Z (high impedance).

```
Ex: MEASURE HIGH_Z = Z
```

- Pin is Active

  A pin is determined to be active if its value can propagate to a measurable point based on test mode constraints (+TIs, etc.). An output or bi-directional pin can only be active if it is tester contacted. Note that a primary input pin may be active even if it is not tester contacted.

- Pin is Tester Contacted

  A pin is determined to be contacted based on a set of rules, which include the test mode BOUNDARY setting, and whether or not it is defined to have some test function (TF). If BOUNDARY is `internal` then the pin must have a TF in order to be assumed contacted. If BOUNDARY is `external` or `no`, then all pins are contacted regardless of TF. An exception to this is when there are more product pins than the tester support, as defined by the TDR TEST_PINS FULL_FUNCTION_PIN_LIMIT, in which case Encounter Test may use parametric measurement units (PMUs) to contact the pins.

- Internal Value Propagated

  These are the values the simulator will propagate during simulation for the given options on the right.

- External Value Measure

These are the values the simulator will measure on Primary Outputs for the given options on the right.

Identifying Unconnected Pins with the ET_UNCONNECTED Attribute

Use the `ET_UNCONNECTED` attribute to identify that a pin is not to be connected to internal logic, to external logic, or both. This allows you to define a cell with the correct interface to match the physical model, and not model that behavior in the model without getting warning messages about the pins being unconnected. The values are:

- `internal` to indicate no net is connected to the pin inside the cell; but the pin will be connected to logic from the outside
- `external` to indicate no net will be connected to the pin from the outside, but there is a connection inside the cell
- `both` to indicate no net will be connected to the pin on either side

The following is an `ET_UNCONNECTED` coding example:

```
(* ET_UNCONNECTED="internal" *) input A;
(* ET_UNCONNECTED="external" *) input B;
(* ET_UNCONNECTED="both" *) output Z;
```

When these attributes are included in the cell definition, the `build_model` command, run with `reportverbose=yes`, will generate messages if the connection does not match the property. For example, if no net is connected to output `Z`, there will be no message since it matches the property. However, if a net is connected to output `Z` a warning message will be produced to indicate that the property was violated.

In addition, pins identified with this attribute will not be included in the inactive logic count that is produced during build_testmode.

Note: These attributes are to be included in the netlist/library description of the cell; they cannot be entered with assign statements during `build_testmode`.

Clock Shaping Designs

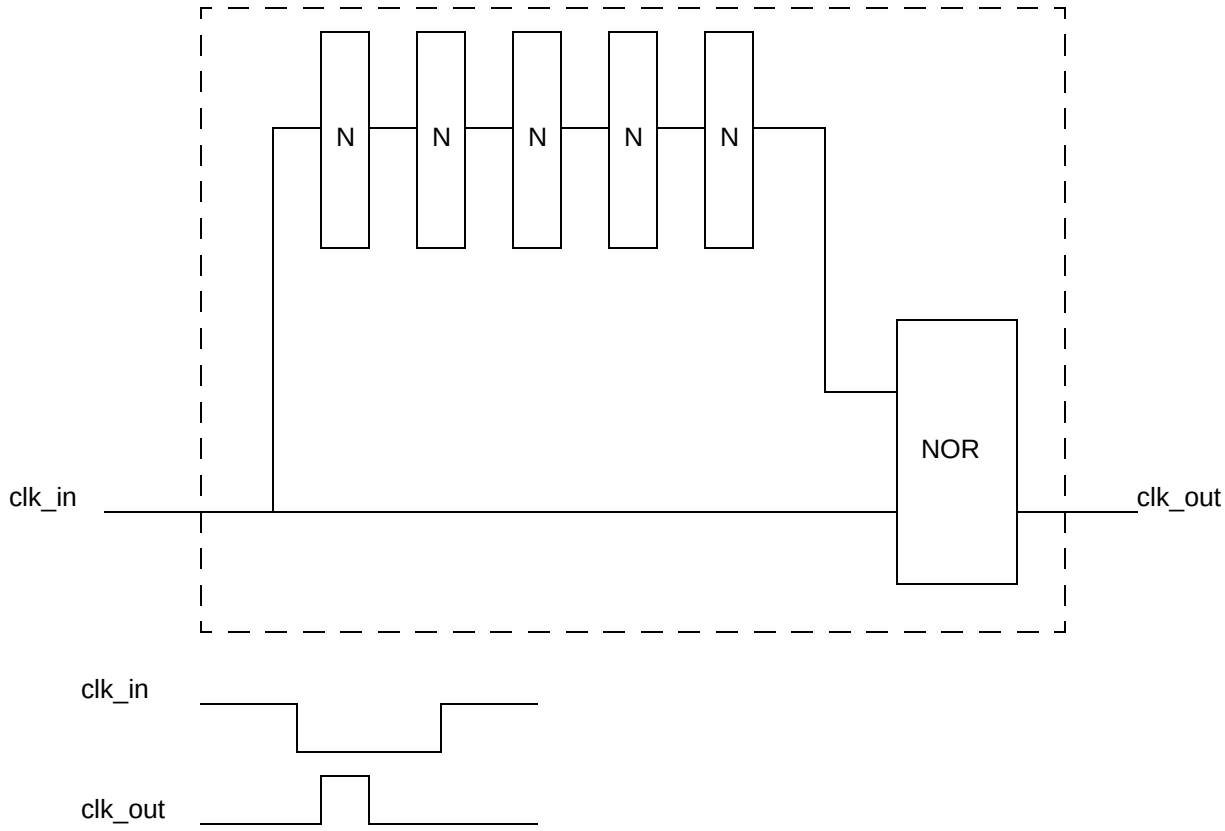
Clock Chopper Primitives

The clock chopper primitives (CHOPT, CHOPL, NCHOPT, NCHOPL) are used to identify a clock chopper design.

A clock chopper is a special design that produces a pulse of a defined duration from a single signal transition at its input. A typical clock chopper design is shown in Figure A-32 on page 173. The out-of-phase reconvergent paths create a hazard at the output block, and the delay inherent in the longer path (the five inverters) holds the non-controlling value of its output block for some time duration after the input transition occurs. The difference in path lengths (in terms of delay) determines the width of the pulse that is produced. By changing the inversion at the output, either a negative-going pulse (quiescent 1) or a positive-going pulse (quiescent 0) can be produced. The block function (AND or OR) at the reconvergence point also affects the polarity of the output pulse and (along with the inversion in the input paths) determines whether the clock chopper is rising-edge or falling-edge sensitive.

Encounter Test recognizes two types of clock choppers: leading-edge and trailing-edge sensitive. The determination of leading- or trailing-edge is based on the relationship of the clock polarity and whether the design is rising- or falling-edge sensitive. So, if the clock's quiescent (stable) state is zero and the chopper produces a pulse on a 1 to 0 transition (falling-edge sensitive) it is a trailing-edge sensitive clock chopper.

Figure A-32 Example Clock Chopper Design



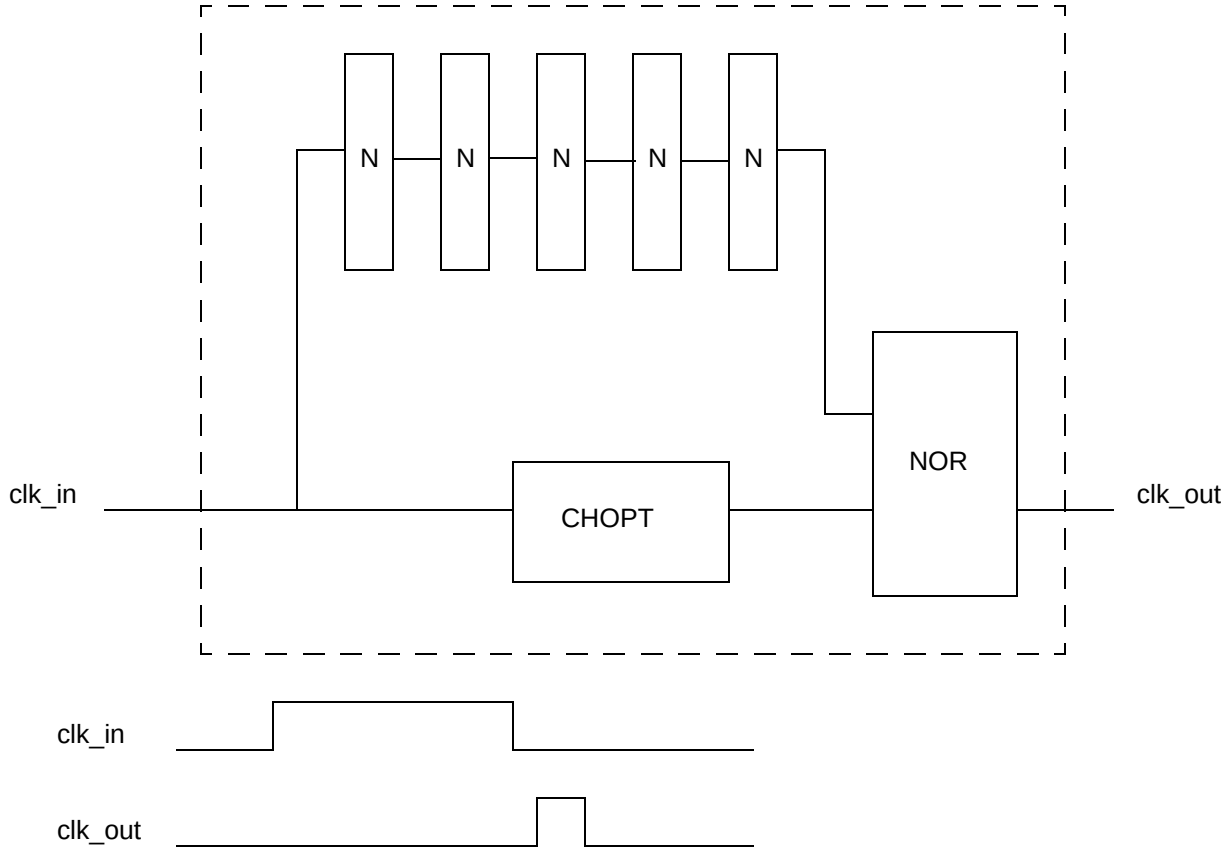
The clock chopper designs require special modeling for correct handling in Encounter Test. Without the correct identification of clock choppers, Encounter Test test generators and high-speed simulators will see this logic as redundant (producing a constant output value). They would not realize that the design can generate a pulse that can clock latches and RAMs.

To identify the clock chopper, Encounter Test uses special primitives CHOPT and NCHOPT for trailing-edge choppers, and CHOPL and NCHOPL for leading-edge choppers. CHOPT and CHOPL function as non-inverting buffers; NCHOPT and NCHOPL function as inverting buffers. The placement of the CHOP block in the design is at the input of the reconvergent block, in the longer path for CHOPL and NCHOPL and in the shorter path for CHOPT and NCHOPT.

See "Clock Shaping Circuitry" on page 175 for additional information.

Figure A-33 shows the Encounter Test model of a trailing edge chopper.

Figure A-33 Encounter Test Model for a Trailing Edge Clock Chopper



Implicit Clock Choppers

An implicit clock chopper is a clock chopper that is modeled to be built without using CHOPL or CHOPT primitives. The mode definition statement IMPLICIT CHOPPERS must be set to *yes* and additional guidelines must also be met. Refer to the following for details:

- IMPLICIT CHOPPERS in *Encounter Test: Guide 2: Testmodes*
- LSSD "Guideline TB.7 - Clock Choppers" in the *Encounter Test: Reference: Legacy Functions*
- GSD "Guideline TG.9 - Clock Choppers" in the *Encounter Test: Guide 3: Test Structures*
- Limitation "Guideline TBL.4 - Clock Choppers" in the *Encounter Test: Guide 3: Test Structures*

■ "Clock Shaping Circuitry" on page 175

Clock Shaping Circuitry

To generate test patterns, test pattern generators generally require extra information about the clock circuitry. Encounter Test gets this information from test function attributes on pins at the top hierarchical level of the design. The clock information can then be propagated to the rest of the design, based on the function of the primitives that the clock pulses go through.

The natural way to analyze clock circuitry is to use the path delays and timing characteristics of the clocking logic. Since the primary simulator in Encounter Test is a zero-delay simulator, timing characteristics cannot be used. Therefore, Encounter Test uses a different primitive to model circuitry which can alter clock pulses to get the extra information it needs.

The best example of clock shaping circuitry which needs extra information is a clock chopper design. Clock choppers take a clock through a delay, and reconverge the clock through a combinational gate, by using the correct reconvergent function, and the proper inversions through either the delay or non-delay paths. A clock pulse is produced on the leading or the trailing edge of the original pulse.

Clock choppers are useful for driving level sensitive latches and RAMs in a high speed digital system where clock skew may cause unintended race conditions.

Clock Chopper Primitives

The clock chopper primitives in Encounter Test perform as buffers or inverters. The clock chopper primitive must appear as the last block before the gate where the input and delayed clock reconverge.

- CHOPL Primitive - Leading Edge Chopper

  The CHOPL primitive acts as a buffer.

- NCHOPL Primitive - Inverting Leading Edge Chopper

  The NCHOPL primitive acts as an inverter.

- CHOPT Primitive - Trailing Edge Chopper

  The CHOPT primitive acts as a buffer.

- NCHOPT Primitive - Inverting Trailing Edge Chopper

  The NCHOPT primitive acts as an inverter.

See the "Clock Chopper Primitives" on page 172 for additional information.

Clock choppers that are modeled without using CHOPL or CHOPT primitives are referred to as implicit clock choppers. The mode definition statement IMPLICIT CHOPPERS must be set to *yes* and additional guidelines must also be met. Refer to the following:

- IMPLICIT CHOPPERS in *Encounter Test: Guide 2: Testmodes*
- The rules listed for LSSD "Guideline TB.7 - Clock Choppers" in the *Encounter Test: Reference: Legacy Functions*.
- The rules listed for GSD "Guideline TG.9 - Clock Choppers" in the *Encounter Test: Guide 3: Test Structures*
- The rules listed for Limitation "Guideline TBL.4 - Clock Choppers" in the *Encounter Test: Guide 3: Test Structures*

Valid Clock Chopper Configurations

Encounter Test identifies valid clock choppers in the entire design by default. Invalid choppers are identified only if they are not fed by an invalid chopper.

In addition to allowing a clock to be chopped either through a leading or trailing edge clock chopper, Encounter Test also supports certain combinations of clock choppers. The following clock chopper configurations are acceptable (to Encounter Test):

- Leading-Edge feeds Leading-Edge

  If the output of a leading-edge clock chopper is fed into another leading-edge clock chopper, the resulting wave forms are assumed to be identical as far as zero-delay simulation is concerned. Any number of leading-edge clock choppers can be cascaded.

- Leading-Edge feeds Trailing-Edge

  In this configuration, when the output of the leading-edge chopper falls, it causes the trailing-edge chopper to emit a pulse. Thus, a single (leading) edge on the clock primary input will cause two internal pulses to be generated in succession.

- Trailing-Edge feeds Leading-Edge

  In this configuration, when the trailing edge chopper emits a pulse, the leading edge chopper will emit a pulse at the same time (during zero-delay simulation).

Encounter Test does not currently support a trailing-edge chopper feeding to another trailing-edge chopper - including if there are intervening leading-edge choppers. This configuration would allow an arbitrary number of internally generated pulses from a single clock primary input edge. Currently, such sophisticated clock generation mechanisms must be dealt with by using on-product clock generation (OPCG) support.

Note: While Encounter Test supports the existence of clock choppers and can generate tests and simulate in their presence, the ATPG engines currently take some short cuts in dealing with them. These short cuts may result in poor fault coverage in cases where a clock is chopped and feeds to memory elements where one memory element feeds the other.

Algorithm for Modeling Clock Choppers in Encounter Test

Clock choppers are modeled such that the clock input to the chopper design is split (fans out), along a short and a long path and re-converges at an AND/NAND gate or at an OR/NOR gate.

1. There are four criteria to consider when modeling a clock chopper design.
    a. Do you need a leading or a trailing edge clock chopper? Refer to "Clock Shaping Design Examples" on page 180 for references.
    b. Is the reconvergent block a NAND/AND or a NOR/OR gate? Referring to Figure A-34 on page 181, the reconvergent block is the right most block in the design.
    c. What is the Stability (OFF) state at the point in the chopper design where the clock fans out? Referring to Figure A-34 on page 181, this is the left most point in that example.
    d. Do you want the output clock pulse to be in phase or out of phase in relation to the input clock pulse?
2. Rules for Modeling Leading Edge Clock Chopper Designs:
    a. To model a leading edge clock chopper, you must always use the CHOPL or NCHOPL Encounter Test primitive and place the primitive in the longer path.
    b. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of zero, and if the reconvergent block is an AND or a NAND gate, then place an inversion into the longer path. This can be accomplished either by using an NCHOPL primitive or by using a CHOPL primitive and an odd number of explicit inverter cells in that path. See Figure A-34 on page 181 and Figure A-35 on page 181 for examples.
    c. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of one, and if the reconvergent block is an AND or a NAND gate, then place an inversion into the shorter path. This can be accomplished by using an odd number of explicit inverter cells in that path. See Figure A-36 on page 182 for an example.
    d. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of zero, and if the reconvergent block is an OR or a NOR gate, then place an inversion into the shorter path. This can be accomplished by using an odd

number of explicit inverter cells in that path. See Figure A-37 on page 182 for an example.

e. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of one, and if the reconvergent block is an OR or a NOR gate, then place an inversion into the longer path. This can be accomplished either by using an NCHOPL primitive or by using a CHOPL primitive and placing an odd number of explicit inverter cells in that path. See Figure A-38 on page 183 and Figure A-39 on page 183 for examples of each method.

3. Rules for Modeling Trailing Edge Clock Chopper Designs:

a. To model a trailing edge clock chopper, you must always use the CHOPT or NCHOPT Encounter Test primitive and place the primitive in the shorter path.

b. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of zero, and if the reconvergent block is an AND or a NAND gate, then place an inversion into the shorter path. This can be accomplished by either using an NCHOPT primitive or by using a CHOPT primitive and an odd number of explicit inverter cells in that path. See Figure A-40 on page 184 and Figure A-41 on page 185 for examples of each method.

c. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of one, and if the reconvergent block is an AND or a NAND gate, then place an inversion into the longer path. This can be accomplished by using an odd number of explicit inverter cells in that path. See Figure A-42 on page 185 for an example.

d. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of zero, and if the reconvergent block is an OR or a NOR gate, then place an inversion into the longer path. This can be accomplished by using an odd number of explicit inverter cells in that path. See Figure A-43 on page 186 for an example.

e. If the clock stability value at the point in the chopper design where the clock fans out is a logic value of one, and if the reconvergent block is an OR or a NOR gate, then place an inversion into the shorter path. This can be accomplished by either using an NCHOPT primitive or by using a CHOPT primitive and an odd number of explicit inverter cells in that path. See Figure A-44 on page 186 and Figure A-45 on page 187 for examples.

4. Generic Rules for Modeling All Clock Chopper Designs

a. The output of the chopper primitive MUST feed directly to and ONLY one of the inputs of the reconvergent block such that there is no additional fanout from the Encounter Test chop Primitive.

b. To get an in-phase chopped clock at the output, in relation to the input clock pulse:

- with a clock stability of 0, use an AND or a NOR gate as the reconvergent block.
- with a clock stability of 1, use a NAND or an OR gate as the reconvergent block.

c. To get an out-of-phase chopped clock at the output, in relation to the input clock pulse:

- with a clock stability of 0, use a NAND or an OR gate as the reconvergent block.
- with a clock stability of 1, use an AND or a NOR gate as the reconvergent block.

The example of a leading edge clock chopper design in Figure A-34 on page 181 assumes that the clock at the fan-out point in the chopper design is at logic value of 0. As the reconvergent block is an AND gate, the model is coded according to rule 2a using an nchopl as the inversion block in the longer path.

Verilog example of a leading edge clock chopper design whose clock input is at a logic value of "0" in the stability state.

```
module nchopl (out, in);

 input in;
 output out;

endmodule

module tbb_CLKCHOPL (out, in);

  input in;
  output out;

  buf buf1 (b1_out, in);
  buf buf2 (b2_out, b1_out);
  nchopl buf3 (chop_out, b2_out);
  and recombine (out, chop_out, in);

endmodule
```

Verilog example of a trailing edge clock chopper design whose clock input is at a logic value of "0" in the stability state.

```
module nchopt (out, in);

input in;
output out;

endmodule
```

```
module tbb_CLKCHOPT (out, in);

  input in;
  output out;

  buf buf1 (b1_out, in);
  buf buf2 (b2_out, b1_out);
  nchopt chop (chop_out, in);
  and recombine (out, b2_out, chop_out);

endmodule
```

Clock Shaping Design Examples

The following examples show typical usages of the clock chopper primitive.

Leading Edge Clock Chopper

The best way to demonstrate the leading edge clock chopper is to show wave forms of the input pulse and the output pulse.

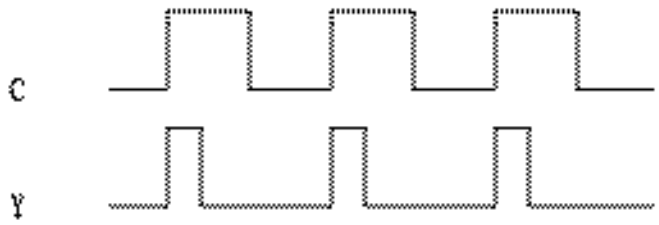
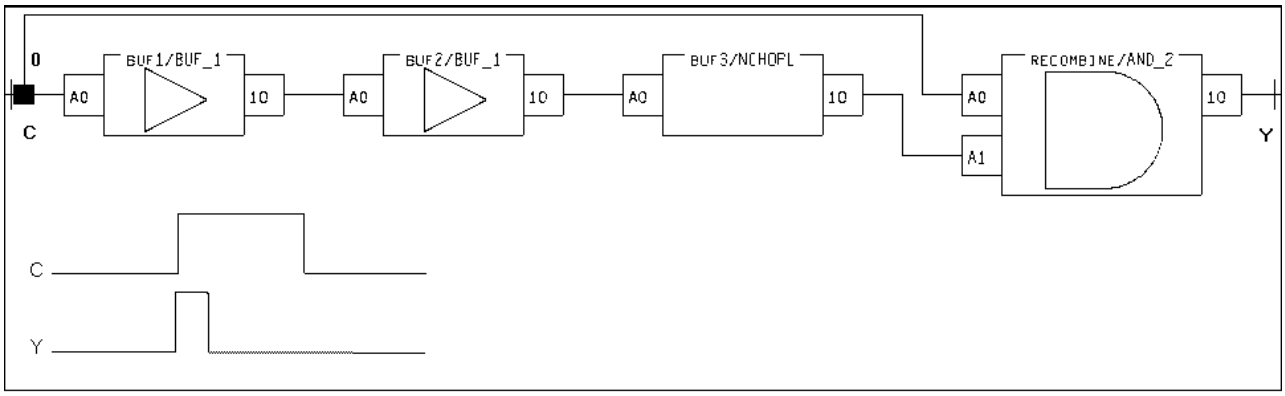
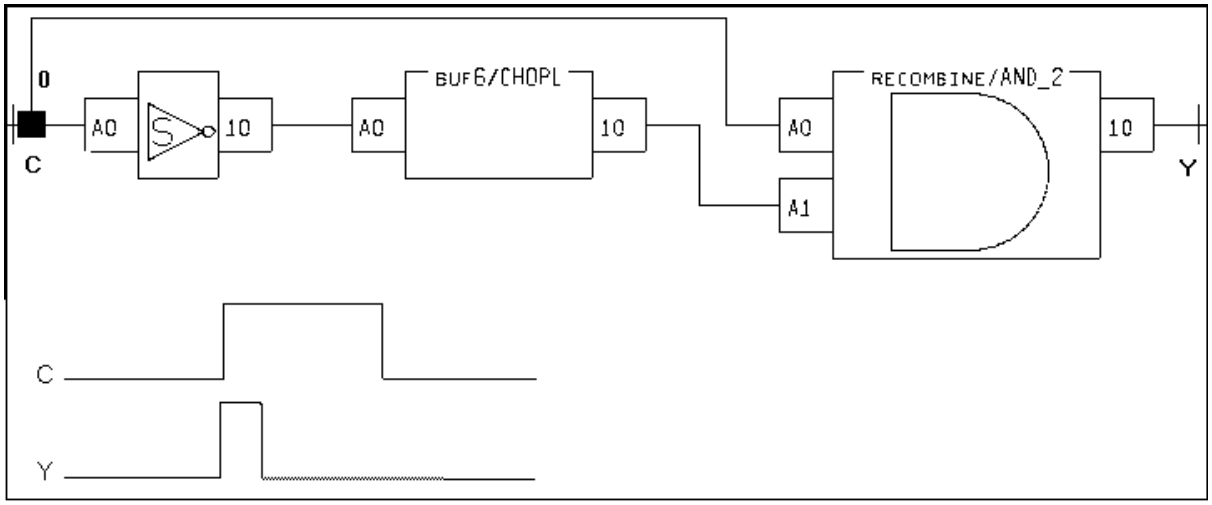


Figure A-34 Leading Edge Clock Chopper Schematic, Example 1



Refer to "Leading Edge Clock Chopper Source, Example 1" on page 235 to view the corresponding source example.

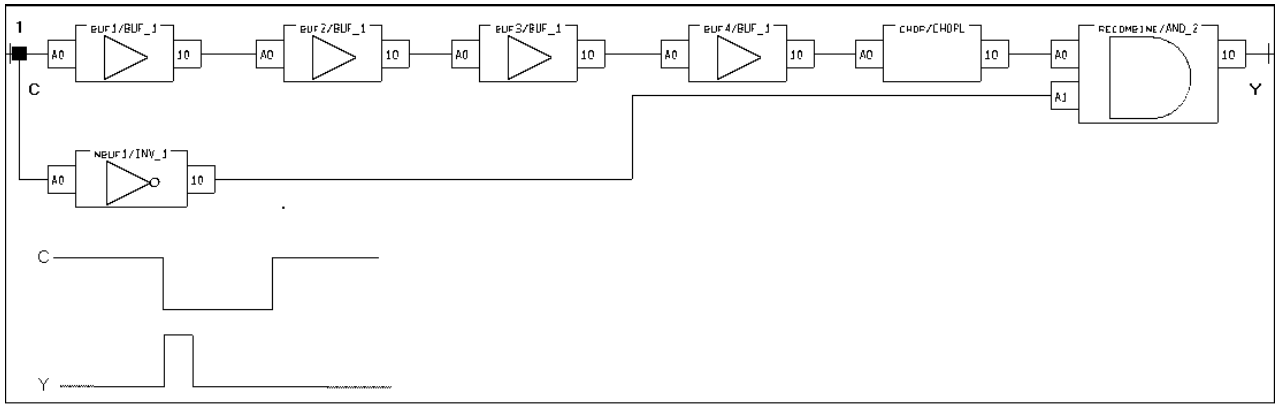
Figure A-35 Leading Edge Clock Chopper Schematic, Example 2



Note: Inverters are suppressed in this schematic as denoted by the "S".

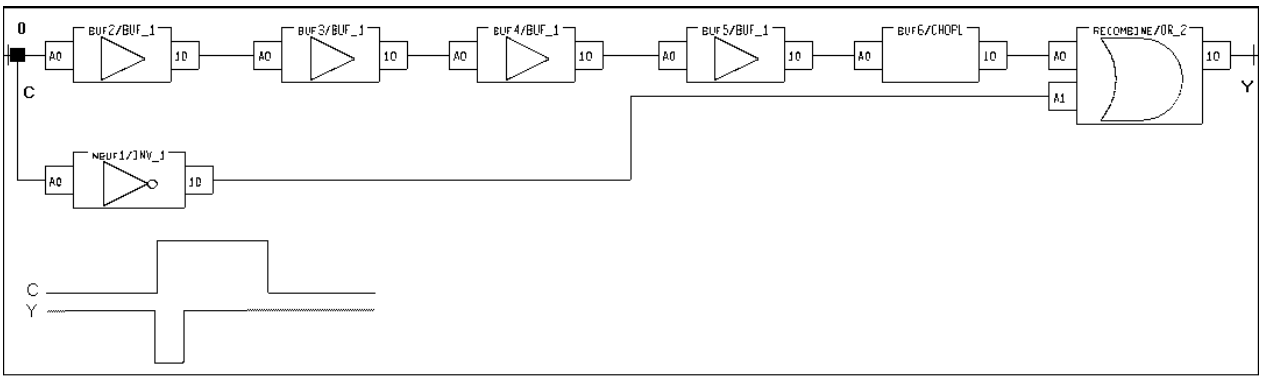
Refer to "Leading Edge Clock Chopper Source - Example 2" on page 236 to view the corresponding source example.

Figure A-36 Leading Edge Clock Chopper Schematic, Example 3



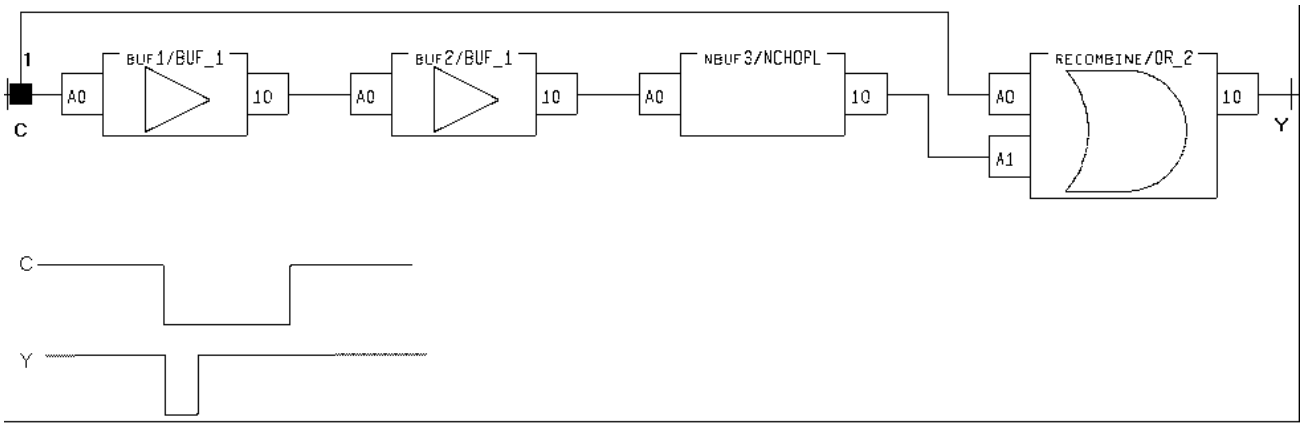
Refer to “Leading Edge Clock Chopper Source - Example 3” on page 237 to view the corresponding source example.

Figure A-37 Leading Edge Clock Chopper Schematic, Example 4



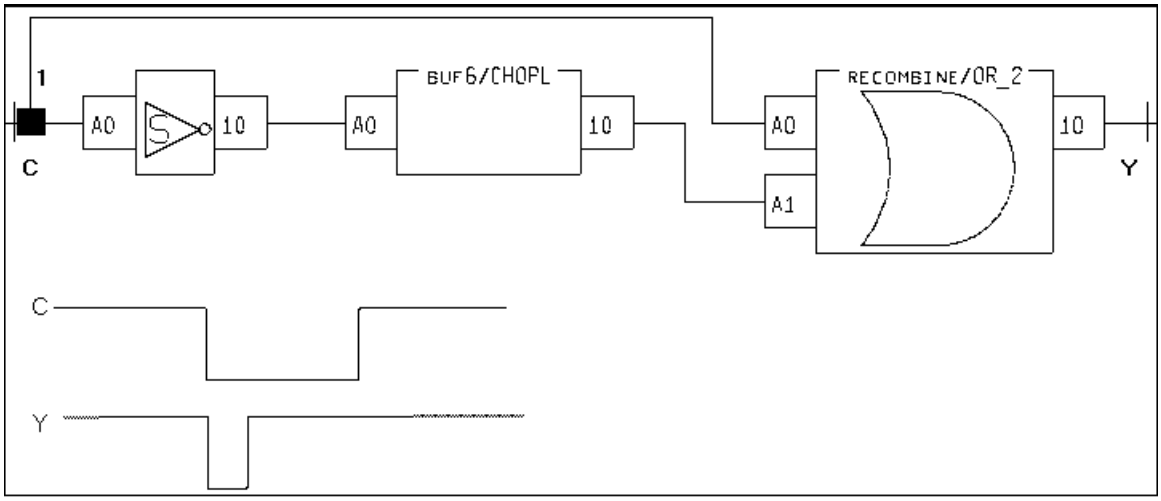
Refer to “Leading Edge Clock Chopper Source - Example 4” on page 238 to view the corresponding source example.

Figure A-38 Leading Edge Clock Chopper Schematic, Example 5



Refer to "Leading Edge Clock Chopper Source - Example 5" on page 239 to view the corresponding source example.

Figure A-39 Leading Edge Clock Chopper Schematic, Example 6



Note: Inverters are suppressed in this schematic as denoted by the "S"

Refer to "Leading Edge Clock Chopper Source - Example 6" on page 240 to view the corresponding source example.

Trailing Edge Clock Chopper

The following waveforms show the input pulse and the output pulse of a trailing edge clock chopper.

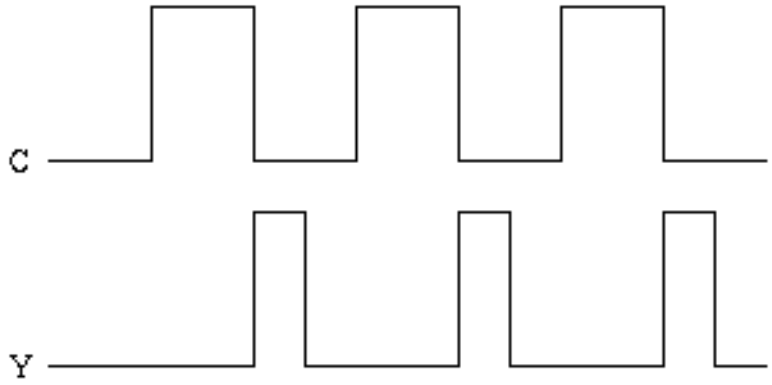
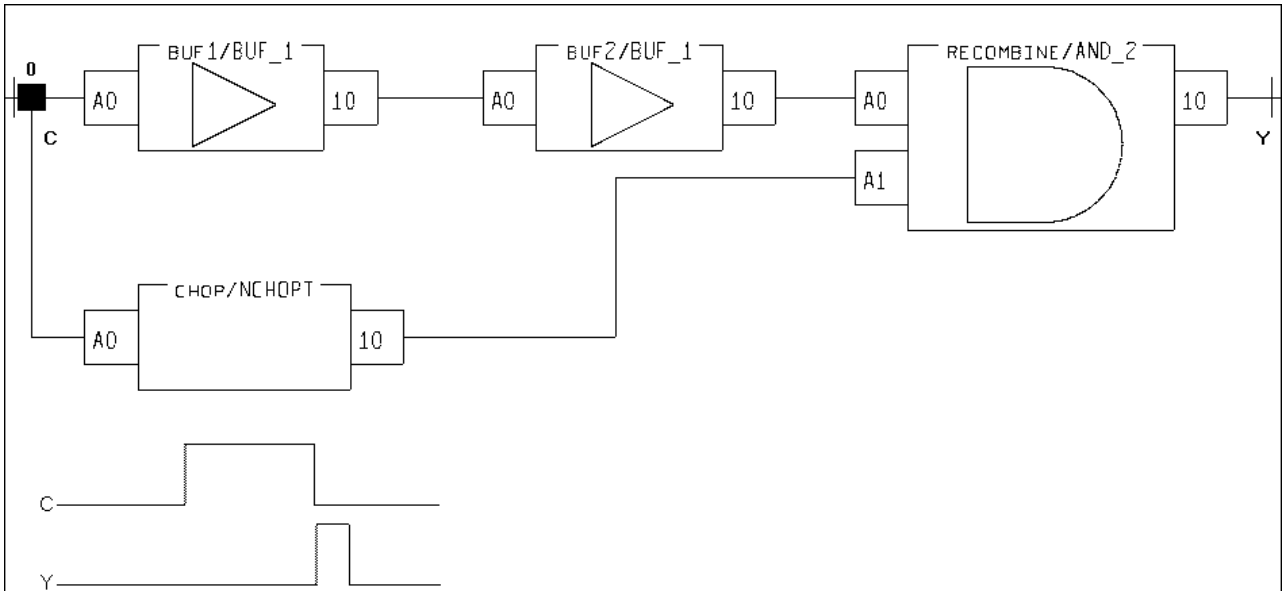
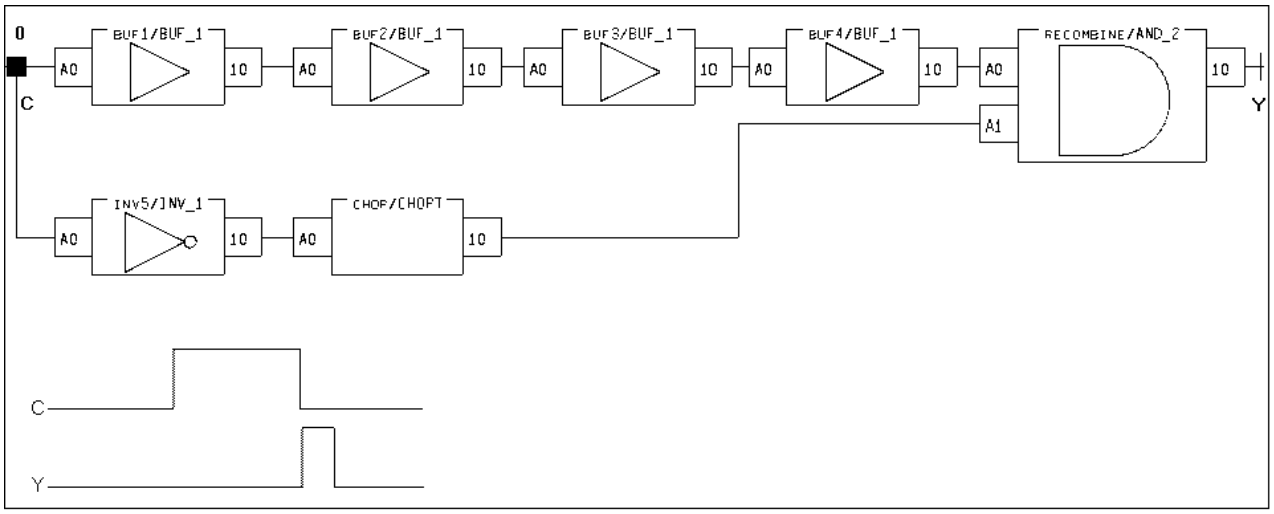


Figure A-40 Trailing Edge Clock Chopper Schematic, Example 1



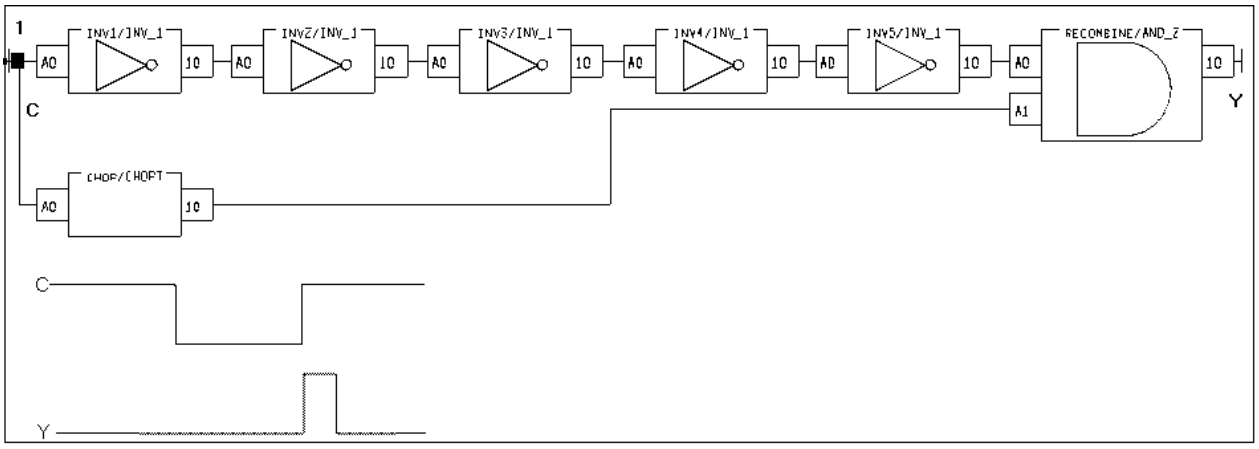
Refer to "Trailing Edge Clock Chopper Source, Example 1" on page 241 to view the corresponding source example.

Figure A-41 Trailing Edge Clock Chopper Schematic, Example 2



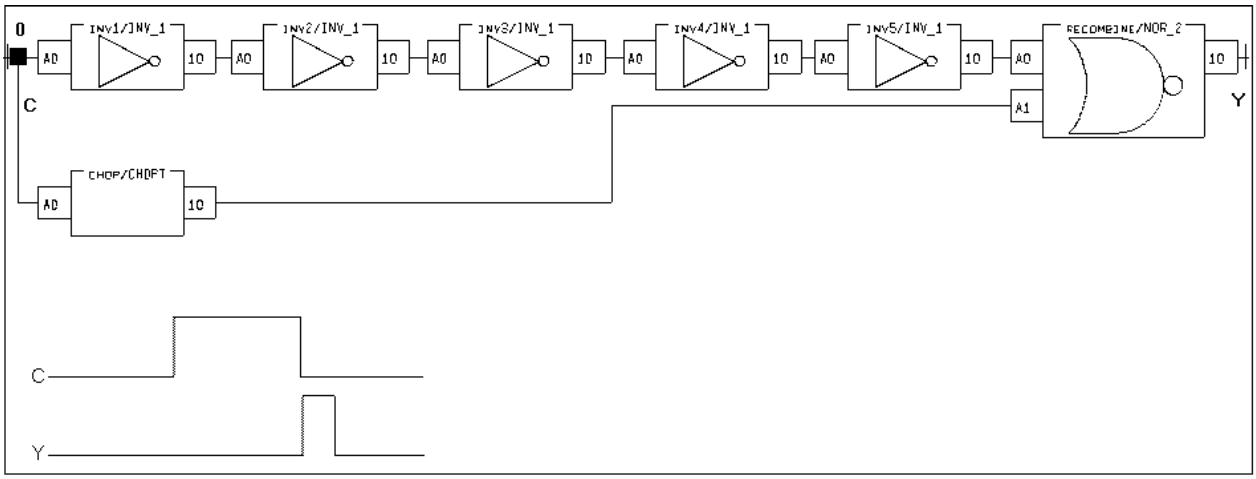
Refer to "Trailing Edge Clock Chopper Source - Example 2" on page 243 to view the corresponding source example.

Figure A-42 Trailing Edge Clock Chopper Schematic, Example 3



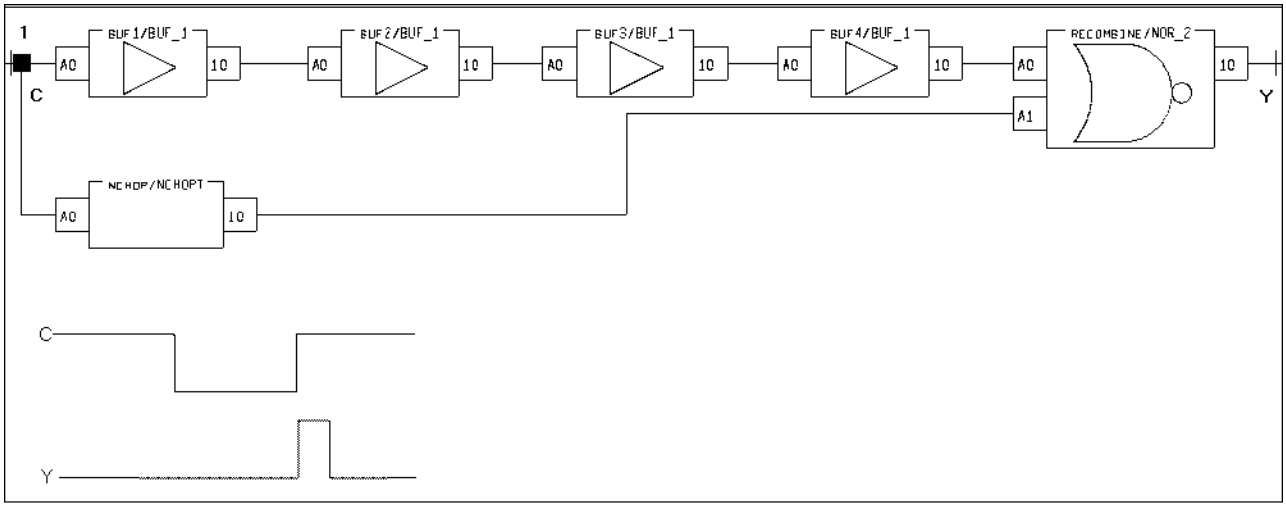
Refer to "Trailing Edge Clock Chopper Source - Example 3" on page 244 to view the corresponding source example.

Figure A-43 Trailing Edge Clock Chopper Schematic, Example 4



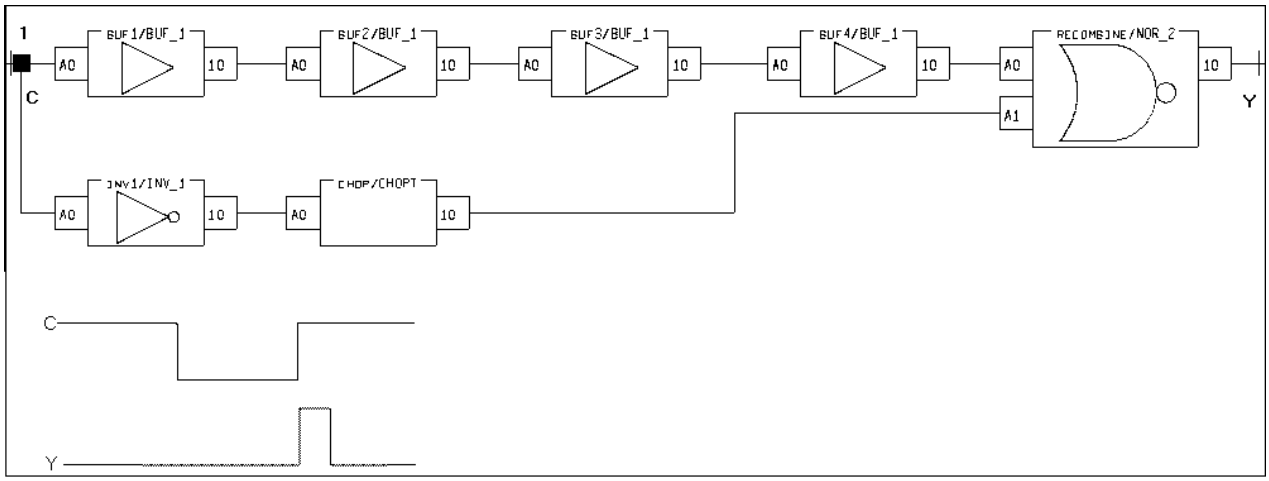
Refer to "Trailing Edge Clock Chopper Source - Example 4" on page 245 to view the corresponding source example.

Figure A-44 Trailing Edge Clock Chopper Schematic, Example 5



Refer to "Trailing Edge Clock Chopper Source - Example 5" on page 246 to view the corresponding source example.

Figure A-45 Trailing Edge Clock Chopper Schematic, Example 6



Refer to "Trailing Edge Clock Chopper Source - Example 6" on page 247 to view the corresponding source example.

Clock Splitter

A clock splitter produces two non-overlapping pulses from a single input pulse. One way to create a clock-splitter is by combining a leading edge clock chopper with a trailing edge clock chopper, as shown in the following waveforms.

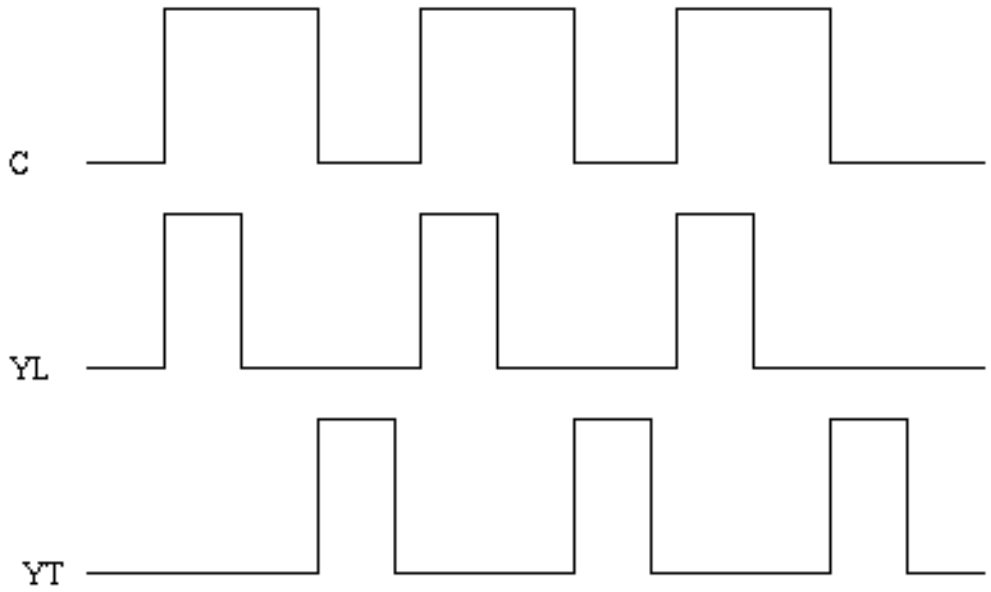
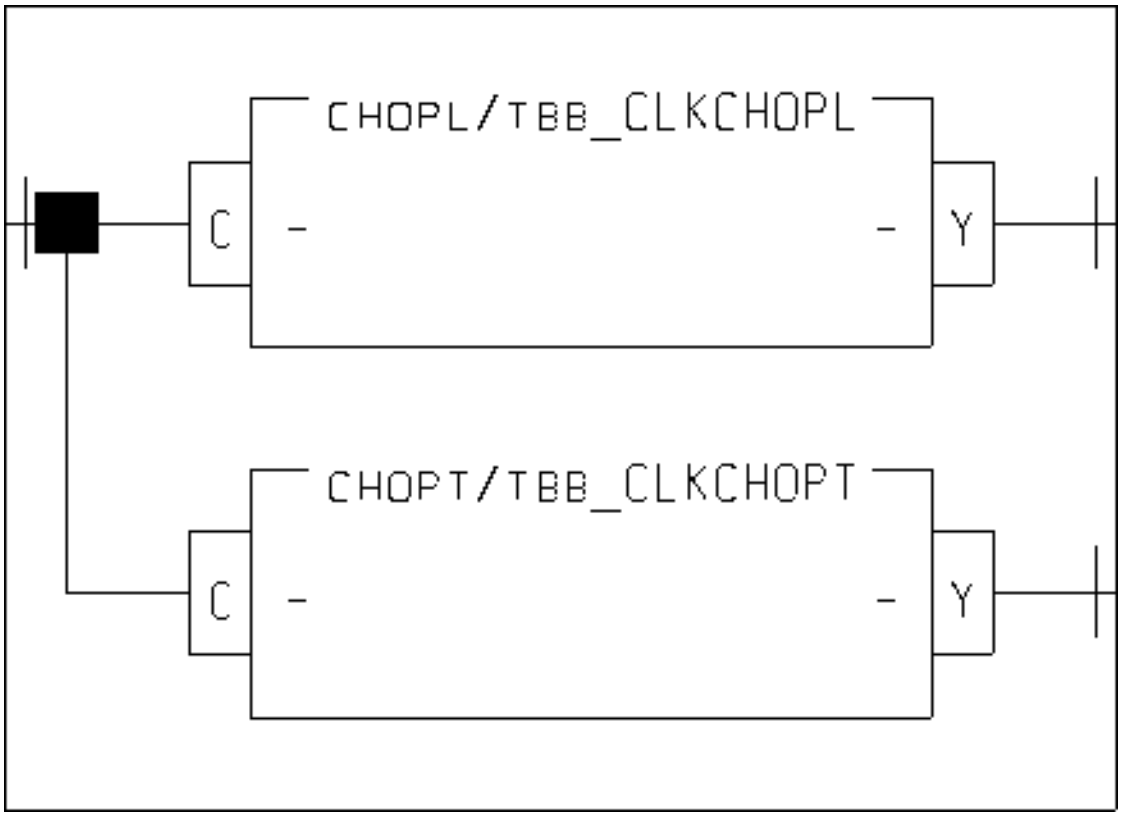


Figure A-46 Clock Splitter Schematic



Refer to Clock Shrinker Source on page 248 to view the corresponding source example.

Clock Shrinker

There are other methods of changing the nature of the clock that do not require special primitives in Encounter Test. The following waveforms show a clock-shrinker design which recombines the clock without inverting.

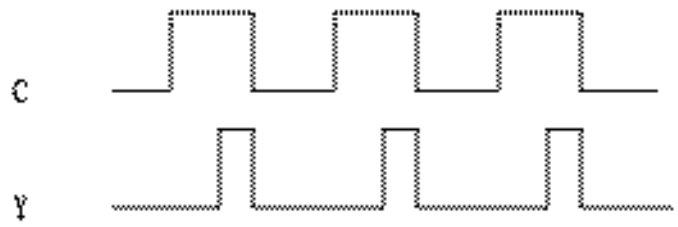
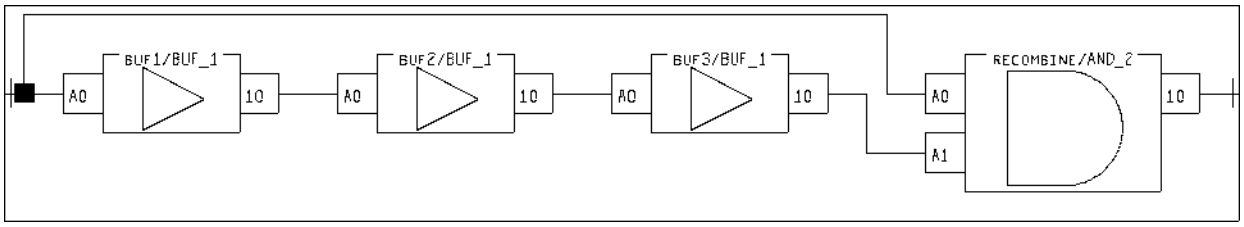


Figure A-47 Clock Shrinker Schematic



Refer to Clock Shrinker Source on page 248 to view the corresponding source examples.

Blackboxes

Blackboxes are modeled using a variety of techniques:

- Module Definition with No Contents

  If a defined module does not contain any content, `build_model` treats it as a blackbox.

  Example:

```
module top_level(out, in1, in2);
     output out;
     input in1, in2;
     memory_cell insst1(t, in1, in2);
endmodule

module memory_cell (out, in1, in2);
     output out;
     input in1, in2;
endmodule
```

- Missing Module Definition

  If a module definition is not provided, `build_model` terminates by default. To allow missing modules to be created automatically, specify `allowincomplete=yes` on the `build_model` command.

  Example: Command Line:

```
build_model workdir=<workdir> source=<source> techlib=<techlib>
allowincomplete=yes
```

  Design Source

```
module top_level(out, in1, in2);
   output out;
   input in1, in2;
```

```
   memory_cell (out, in1, in2);
endmodule
// No definition for memory_cell is provided
```

Note: Encounter Test attempts to determine the correct pin directions from the usage of the missing module. Occasionally an input pin is incorrectly identified as an output pin causing 3-state contention on a net. It is recommended to check the pin directions associated with missing modules through the Schematic Browser.

- Behavioral Module Definition

  If a behavioral module definition is provided, `build_model` ignores the behavioral verilog constructs. If no structural gates or nets are defined in the behavioral model, the resulting module will be a blackbox.

  Note: Modules with contents are selected before blackboxes.

If the same module is defined twice, once with contents and once as a blackbox, Encounter Test always uses the definition with contents, regardless of which definition comes first in the `source` or `techlib` keywords.

Blackbox Output Pin Values

By default, blackbox outputs are tied to logic X. This can cause soft contention in some cases such as a multi-source net fed by a blackbox. To change this default behavior, use the keyword `blackboxoutputs=[0|1|x|z]` during `build_model`. In GUI, set the required logic value for the *Treat blackbox outputs as source of* option in the *Build Model - Advanced* window. Each black box output is modeled with a TIEX (the default), a TIUP, a TIDN, or a configuration that drives a Z.

You can specify the value for each individual blackbox cell and each individual output/bidi port of a blackbox cell within the design using the `BLACKBOXOUTPUTS` attribute. The syntax of the `BLACKBOXOUTPUTS` attribute value is similar to the `blackboxoutputs` keyword. If you do not specify the logic value for a black box cell or the port, `build_model` continues to use the value provided in the `blackboxoutputs` keyword.

Define the `BLACKBOXOUTPUTS` attribute for the cell or the port in the same way as you would define any other attribute used by Encounter Test when building the logic model. Specify the attribute on the port definition or the black box cell interface definition in the design source, or specify an edit file with `addAttribute` statement as input to `build_model`. You can also add the attribute on a black box cell or the port using the `edit_model` command.

Blackbox Command Example

```
build_model workdir=<workdir> source=<source> allowincomplete=yes
blackboxoutputs=z
```

Figure A-48 Default Value for Blackbox Outputs is X

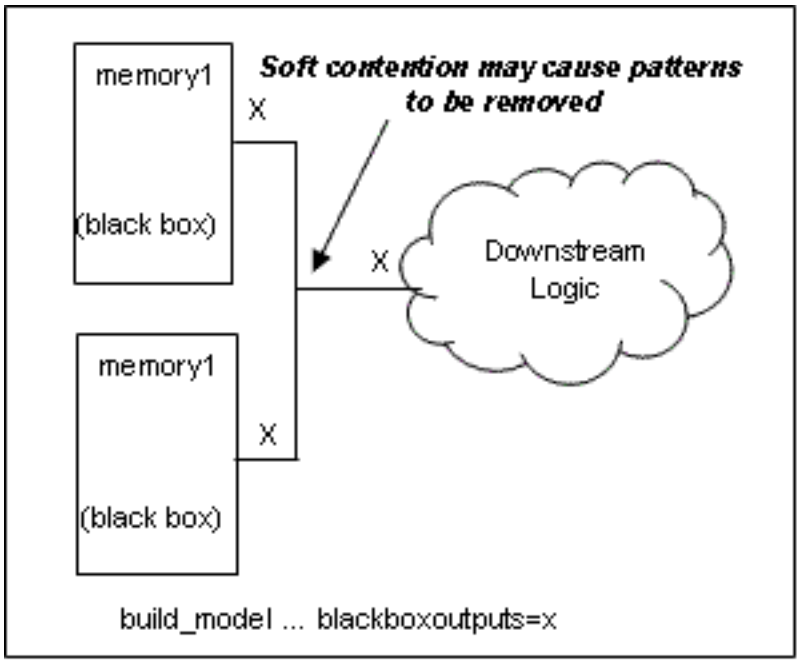
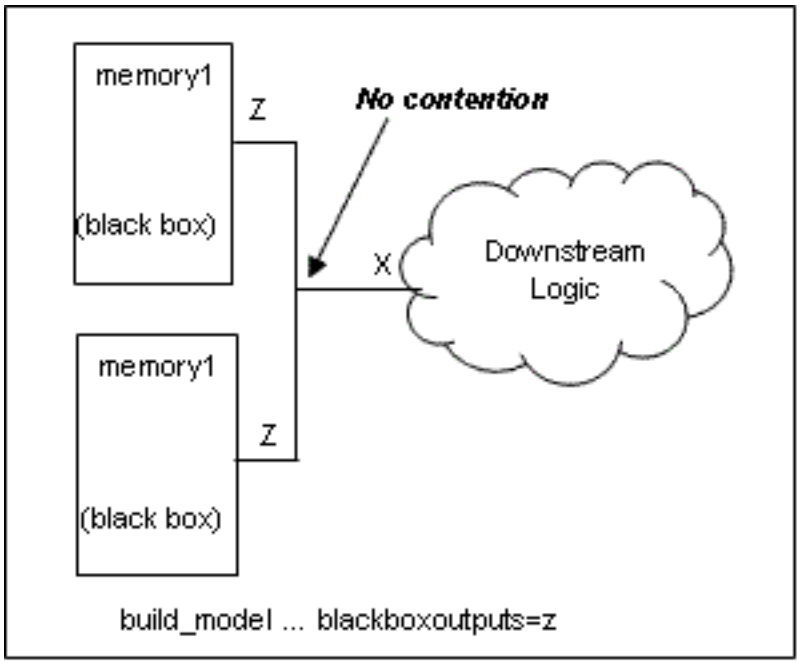


Figure A-49 Blackbox with blackboxoutputs=z



Blackbox Faults

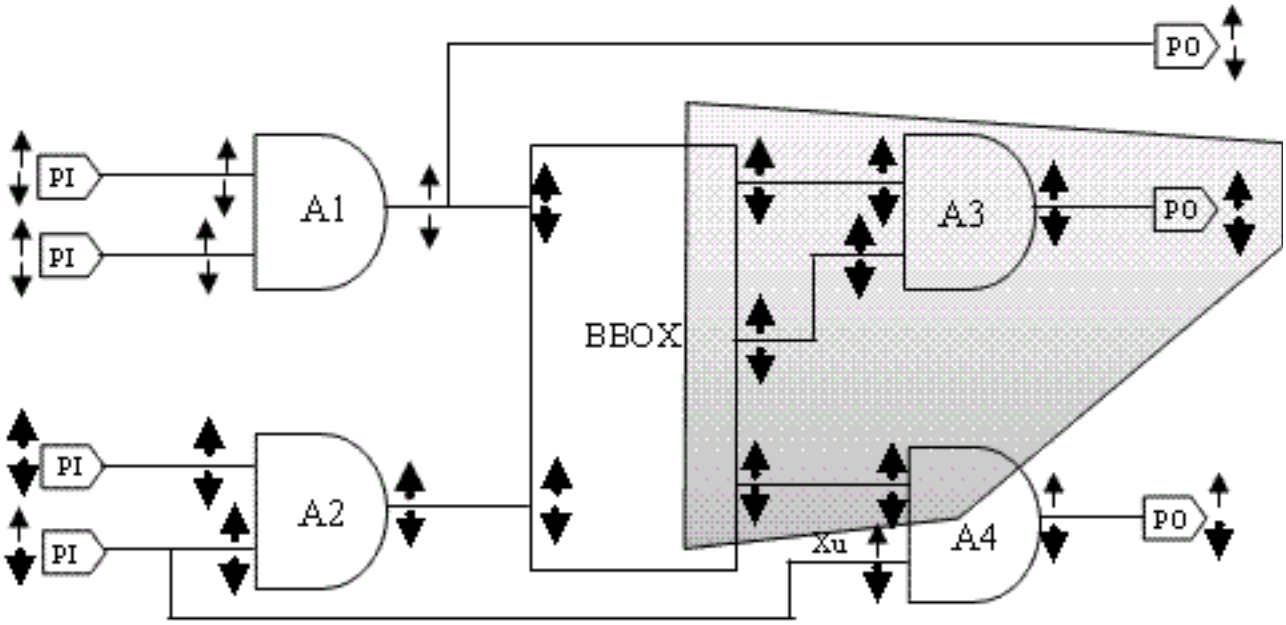
Encounter Test classifies blackbox faults as untested faults and includes these in the global fault counts and fault coverage calculations.

Faults Downstream From Blackbox Outputs

Blackbox outputs are almost always tied to X (or Z). When propagated downstream, some number of gates may be tied to X as well. All faults along a path that is tied to X are included in the fault model and are identified as untestable undetermined. When a testmode is built, those faults that are observable will be reclassified as untestable due to a tiex.

The following figure represents faults on gates controlled on tiex:

Figure A-50 Faults Untestable Due to Tie X

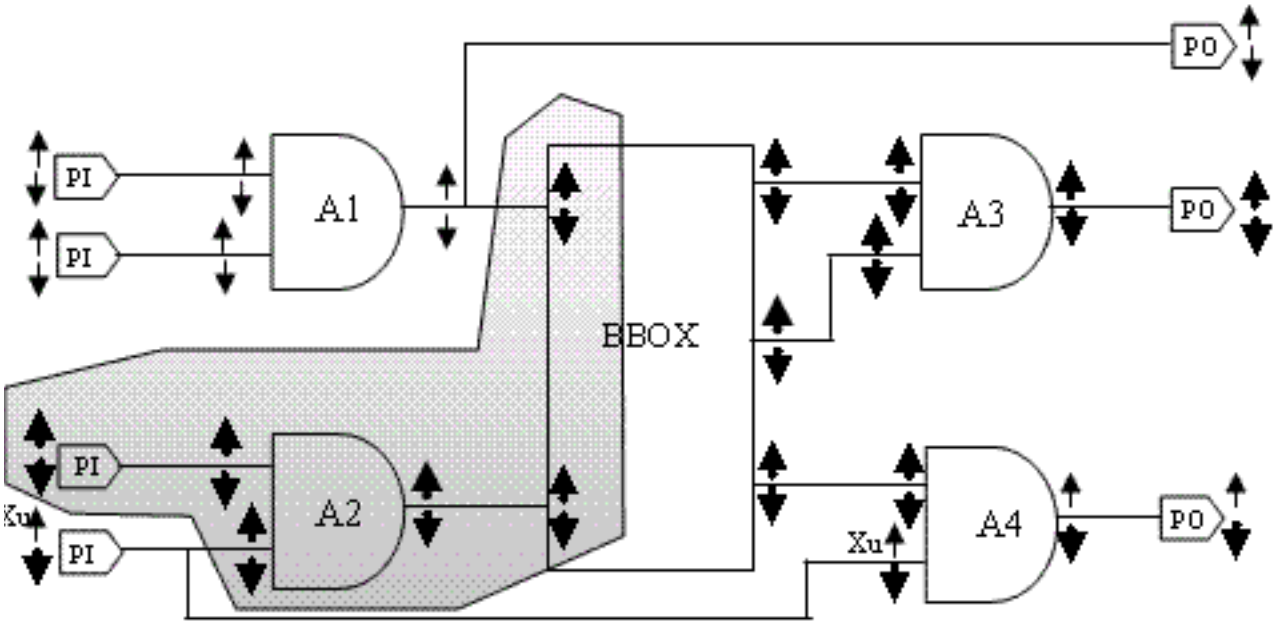


Faults Upstream From Blackbox Inputs

Faults that feed only into a blackbox are located on dangling logic. Such faults are marked as untestable undetermined. As other (non-blackbox) cells may also contain input pins that are not modeled internally, any logic that feeds only to a cell input pin that is not modeled internally is considered untestable undetermined. This applies to both technology cell inputs and inputs on higher level entities.

The following figure represents faults feeding unmodeled cell inputs:

Figure A-51 Faults Feeding Unmodeled Cell Inputs

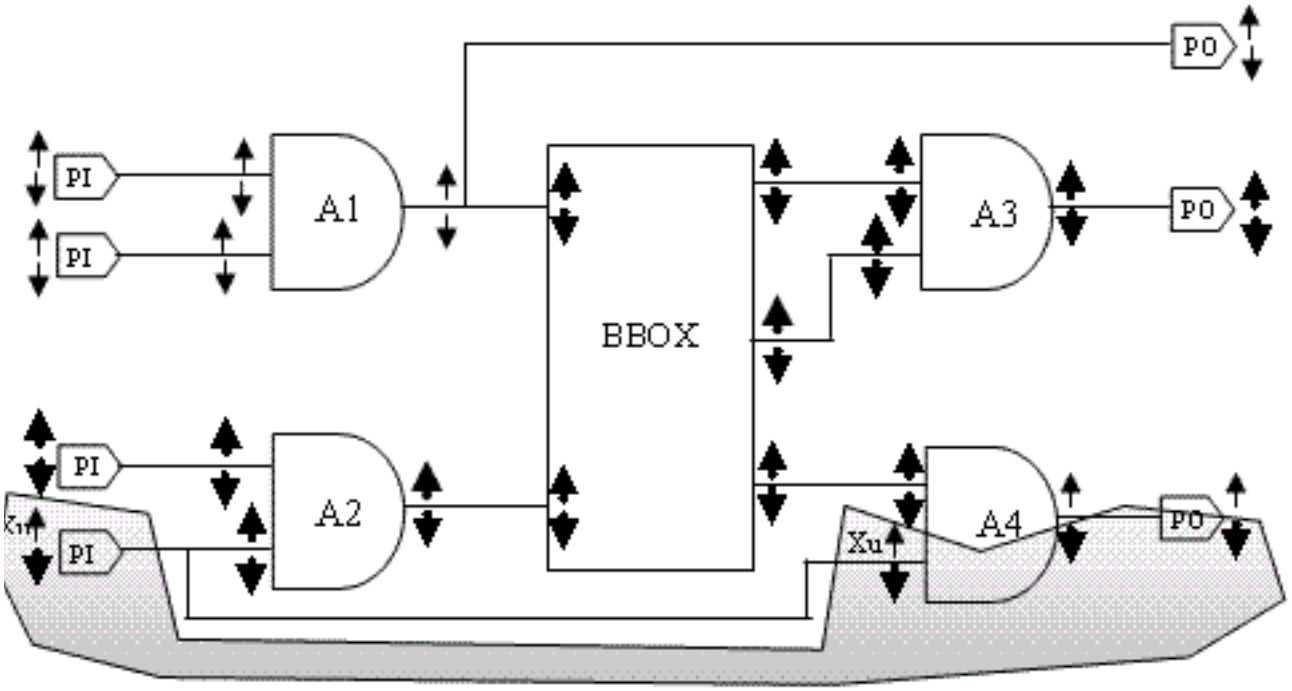


Faults That Require a Blackbox Output to Propagate

Any faults that require a blackbox output to be set to a known value so that the fault can be propagated are either marked as untestable undetermined or identified as possibly testable at best (PTAB) due to a Tie X (designated as Xu in report_faults). This is also true if the source of the Tie X is from a technology cell with contents and one or more of the output pins tied to X. When a testmode is built, those untestable undetermined faults that are observable are reclassified as untestable due to tiex.

The following figure represents faults requiring a value tied to X:

Figure A-52 Faults Requiring Value Tied to X



Faults on Cell Outputs Tied to 1 or 0

In a model with blackbox outputs tied to 1 or 0, faults of the same polarity as the tie value are ignored, and the opposite polarity faults are identified as untested.

Modeling Faults for Blackbox Ports

To enable modeling faults for black box input ports and module input ports that are not modeled internally to the module, specify the attribute `INPUTFAULTS` on either a module statement or on an individual module I/O port definition.

The logic model that will enable modeling faults for a module input port if `blackboxinputfaults` is set to `no` and either the module statement or the module input port has the attribute `INPUTFAULTS=YES`.

However, the logic model that will not enable modeling faults for a module port if `blackboxinputfaults` is set to `yes` and either the module statement or the module input port has the attribute `INPUTFAULTS=NO`.

Refer to "build_model" in the *Command Line Reference* for more information on the `blackboxinputfaults` keyword.

Adding Cutpoints to Blackbox Outputs

It is possible to add cutpoints to blackbox outputs during build_testmode. Adding cutpoints may make some previously ignored faults around blackboxes testable. These faults will be marked ATPG untestable. By default, ATPG does not re-process faults that are ATPG untestable. However, if `reprocessuntestables` is set to `yes` for the create tests commands, ATPG reprocesses these faults and if possible, generate tests for them.

Nets with Multiple Sources

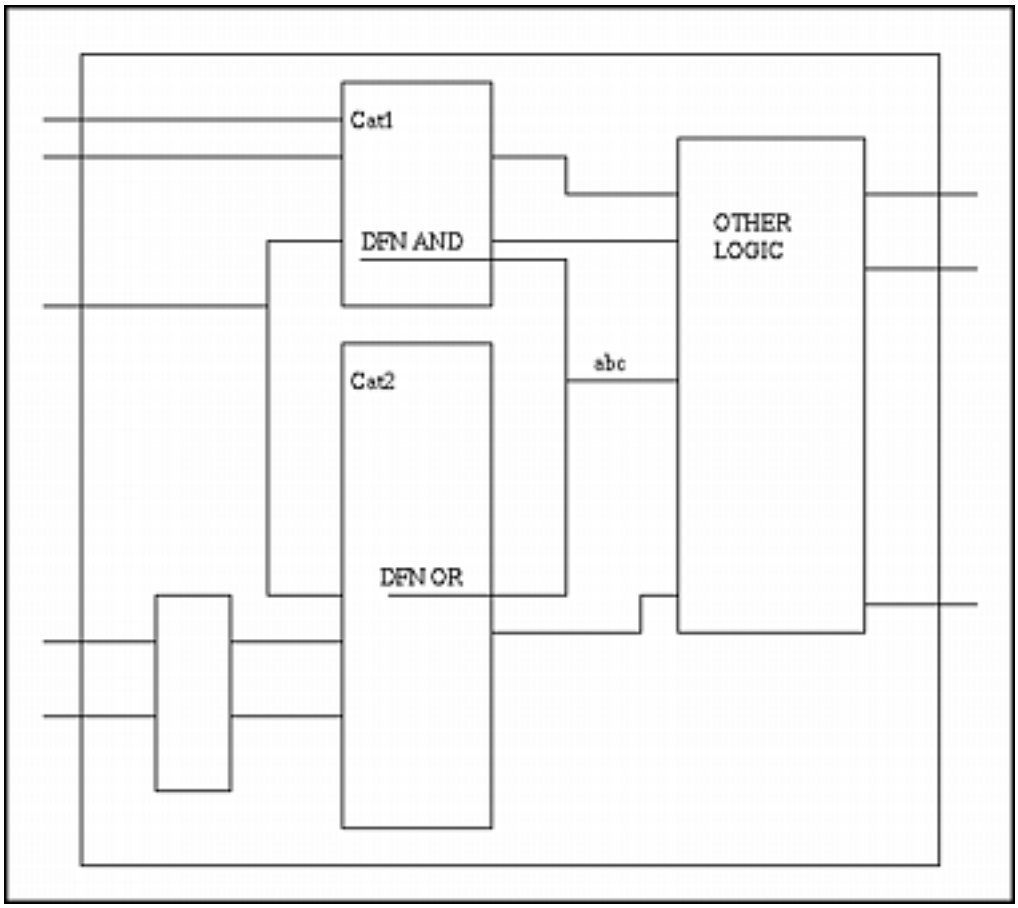
The way of deriving the value of a multi-source net when different source values are applied to it is called its dot function (DFN). The dot function is specified in the following ways:

- A DFN attribute applied to one or more of the cell pins or instance pins on the net determines the dot function.
- Verilog wand, wor, tri net declarations determine the dot function (`build_model` inserts the appropriate DFN attributes).
- The default dot function specified in the build_model keyword `defaultdfn` determines the dot function for all the multi-source nets without DFN attributes.

Note:

- Specify `DFN=NO` to identify a pin that should not be connected to a multi-source net. The `build_model` command checks this condition and issues an error message if required.
- Any multi-source net driven by a three-state primitive is assigned a DFN of T (tri-state).
- If conflicting DFN attributes are found, a DFN of T is modeled.
- DFN need not exist for all pins on a net.

Figure A-53 Example of Conflicting DOT Functions



DFN Attribute Values

AND

The dot function of AND assumes that a logic zero overrides all other logic values.

Note: Encounter Test forces a DFN of T for nets with weak logic values or high-impedance. Therefore, only Boolean logic values will appear as sources of a net with DFN values of AND.

```
AND | 0  1  X
----|---------
0   | 0  0  0
1   | 0  1  X
X   | 0  X  X
```

OR

The dot function of OR assumes that a logic one overrides all other logic values.

Note: Encounter Test forces a DFN of T for nets with weak logic values or high-impedance. Therefore, only Boolean logic values will appear as sources of a net with DFN values of OR.

```
OR | 0  1  X
---|----------
0  | 0  1  X
1  | 1  1  1
X  | X  1  X
```

T (three-state)

The T or three-state dot assumes that strong logic values override weak logic values. If two conflicting strong values appear at the dot, a direct path from voltage to ground exists, and a three-state burnout will occur.

```
T | 0  1  X  Z  L  H  W
--|-------------------
0 | 0  X  X  0  0  0  0
1 | X  1  X  1  1  1  1
X | X  X  X  X  X  X  X
Z | 0  1  X  Z  L  H  W
L | 0  1  X  L  L  W  W
H | 0  1  X  H  W  H  W
W | 0  1  X  W  W  W  W
```

Note: T must be used if the pin is sourced by a three-state driver or transistor primitive. If a DFN other than T is used, Encounter Test issues an error message and sets the dot function to T.

NO

A DFN of NO indicates that this pin may not be a source of a multi-sourced net.

Examples

Example 1: Using a Verilog Wired Net Declaration

```
module top(...);
...
wand three_state_net;
...
unit1 inst1(..., three_state_net, ...);
unit2 inst2(..., three_state_net, ...);
...
endmodule
```

A dot function of AND is modeled.

Example 2: Using a DFN attribute

```
module top(...);

...
wire three_state_net;
unit1 inst1(..., three_state_net, //! DFN="O"
            ...);
unit2 inst2(..., three_state_net, ...);
...
endmodule
```

A dot function of OR is modeled.

Example 3: Using the default DFN keyword

Verilog Source

```
module top(...);
...
wire three_state_net;
unit1 inst1(..., three_state_net, ...);
unit2 inst2(..., three_state_net, ...);
...
endmodule
```

Command Line

```
build_model ... defaultdfn=OR ...
```

A dot function of OR is modeled.

Sourceless Nets

Encounter Test uses several rules to determine how a net should perform when it becomes sourceless in the hierarchical model of a design.

Encounter Test looks for a TIE attribute on all pins and usage pins that drive the net. The valid values for the TIE attribute are 1, 0, or X. If a valid TIE attribute is found, the net will take on a constant value, as specified by the TIE attribute.

If the net is sourceless and has no TIE properties, Encounter Test examines the net name. The net names that Encounter Test recognizes are shown in the following table:

Logic 0 Source	Logic 1 Source
ZERO	ONE

Logic 0 Source	Logic 1 Source
zero	one
VSS	PWR
vss	pwr
vss!	pwr!
VSS!	PWR!
GND	VDD
gnd	vdd
GND!	VDD!
gnd!	vdd

If the net is sourceless, has no TIE attribute, and its netname is not recognized by Encounter Test, the net becomes a constant net tied to the default tie value. The default tie value is a logic X unless some other default tie value is specified when running `build_model`.

A net is also tied if it is sourceless and its name appears in the `build_model` net voltage file. See "Using a Net Voltage File to Define Power and Ground Nets" on page 201 for details on using a net voltage file.

Examples of Specifying a Default TIE Value

Specifying a default tie value for all sourceless nets

```
build_model workdir=/local/dlx \
              designsource=/loca/source/dlx.v \
              techlib=/local/techlib/tech90.v \
              defaulttie=0
```

Specifying a tie value for a specific net (Verilog 2001)

```
module techcell (z, a, b, c);
     output z;
     // a is tied to 1 if it is not connected when techcell is used
     (*  TIE="1" *) input a;
     input b,c;
     ...
endmodule
```

Specifying a tie value for a specific net (Pre-Verilog 2001)

```
module techcell (z, a, b, c);
     output z;
     // a is tied to 1 if it is not connected when techcell is used
     input a;    //! TIE="1"
     input b,c;
     ...
endmodule
```

A Sourceless Net Tied Due To Netname

```
module techcell (z, a, b, c);
        output z;
        input a,b,c;
        wire vdd;
        and (z, a, b, c, vdd);  // fourth input is tied to logic 1
endmodule
```

Using a Net Voltage File to Define Power and Ground Nets

A net voltage file is an alternate voltage file which redefines the set of net names `build_model` uses when determining sources of power and ground. Specify this file to override the default set of net names used by `build_model`. Be aware of the following when using a net voltage file:

- If the default set of net names is still required, they must also be specified in the net voltage file.
- If specifying either power or ground, `build_model` retains and applies the defaults for either the unspecified power or unspecified ground.

Use the following commands to specify the net names in the net voltage file:

- `define_ground_nets -nets` *`net_list;`*

  Use this command to specify the set of nets, which when sourceless, will be connected to ground (Tied to 0). When you use this command, the `build_model` process Ties to 0 any net ending in one of the strings identified in *`net_list`*.

- `define_power_nets -nets` *`net_list;`*

  Use this command to specify the set of nets, which when sourceless, will be connected to power (Tied to 1). When you use this command, the `build_model` process Ties to 1 any net ending in one of the strings identified in *`net_list`*.

The following are the syntax rules for net voltage file:

- Each specification of `define_ground_nets` or `define_power_nets` statement must end with a semicolon.
- The `-nets` option in the syntax is optional.
- Braces { } around a `net_list` are optional. Net names in `net_list` must be separated by whitespace and the list should be terminated by a semicolon, as shown in the following example:

  ```
  define_power_nets -nets {mypwr1 mypwr2 mypwr3};
  ```

- `net_list` may span across multiple lines in the file.
- Multiple forms of comments are accepted. Slash-asterisk (`/* */`) and slash-slash (`//`) ignore rest of line, and a slash-asterisk block comments that spans lines.
- Syntax errors are identified and the parsing of the file continues looking for the next valid piece of input data.
- Syntax errors do not cause `build_model` to fail. Default actions are performed and the model is built. However, errors may need to be corrected.

Refer to build_model in the *Command Line Reference* for more information.

The following is an example of a net voltage file:

```
define_ground_nets -nets mygnd;
define_power_nets -nets mypwr;
```

Modeling Implied Sources of Power and Ground

The following conditions are applied when specifying the advanced `build_model` keyword `impliedgndvdd=yes`.

Encounter Test adds sources of Power and Ground based on the same net name recognition, including the use of a net voltage file, as is used for modeling sourceless nets. All sources defined in the gnd/vdd net will be dotted together using a DOTT primitive and a source representing ground (TIDN primitive) or power (TIUP primitive). This method ensures three-state contention detection when neither the data input nor the enable input of the TSD driver is wired to ground. This has the potential to produce a logic 1 dotting with ground that will cause three-state contention. All sinks defined in the gnd/vdd net will be fed by the TIDN/TIUP primitive.

Several examples follow:

Figure A-54 Example 1 of Wiring Unused Drivers/Receivers to Ground and Resulting flatModel

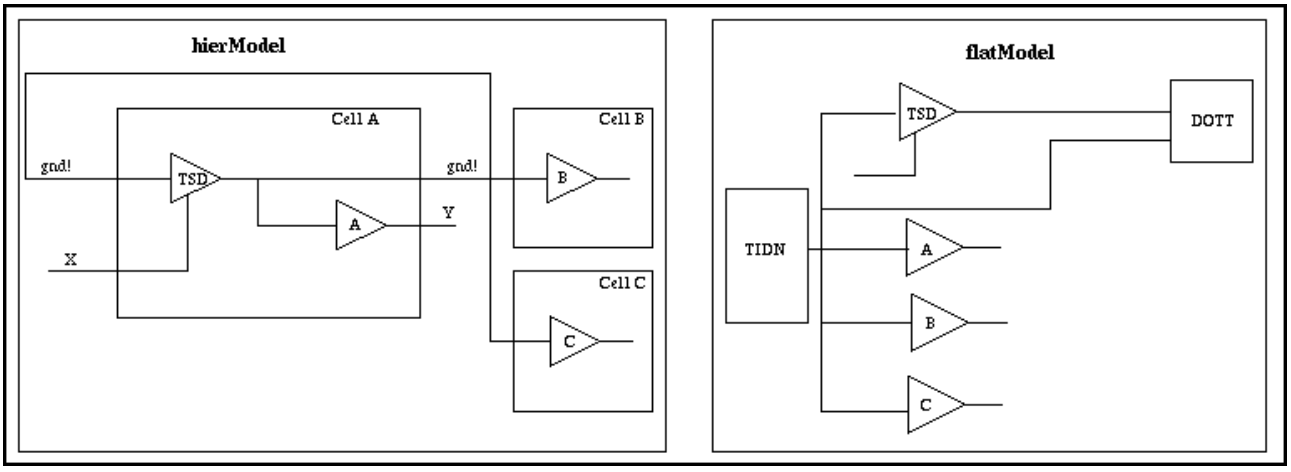
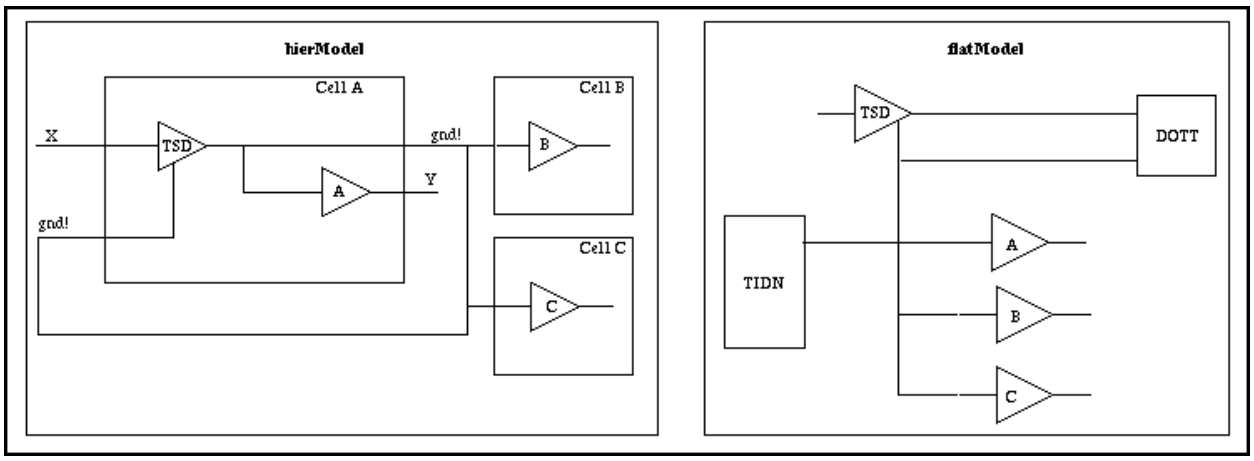
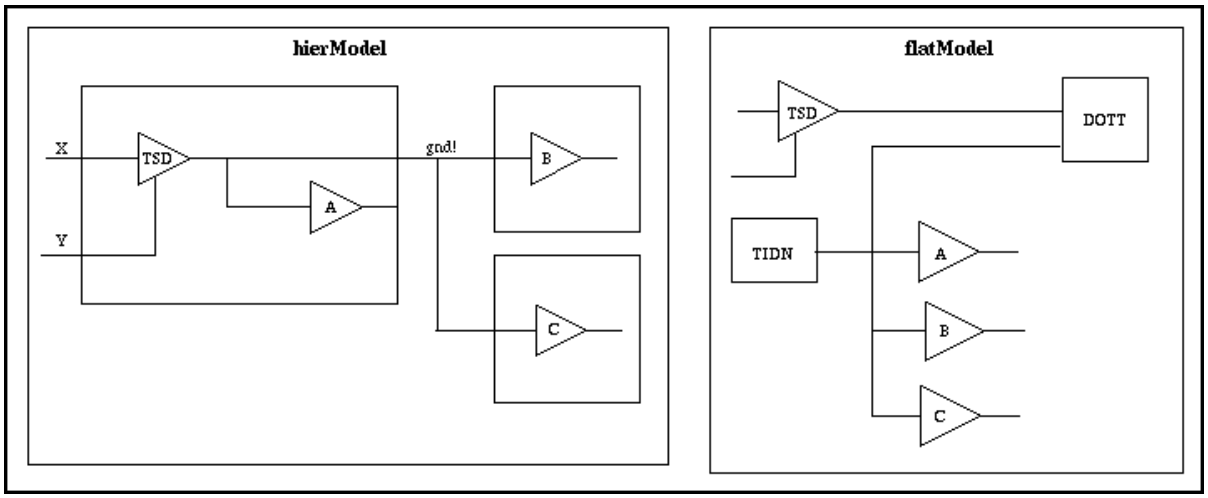


Figure A-55 Example 2 of Wiring Unused Drivers/Receivers to Ground and Resulting flatModel



The resulting flatModel differs in case neither the data nor the enable of the TSD driver is connected to ground, as shown in the following figure.

Figure A-56 Example of Neither Data nor Enable Connected to Ground and Resulting flatModel



Specifying Differential I/O and Other Correlated Pins

Encounter Test provides the ability to correlate pins to one another. This capability is often used to model differential I/O pins. One pin is called the representative pin and the others are the correlated pins. Correlated pins always share the same (or complementary) logic value as their representative pin.

There are three ways to identify correlated pins:

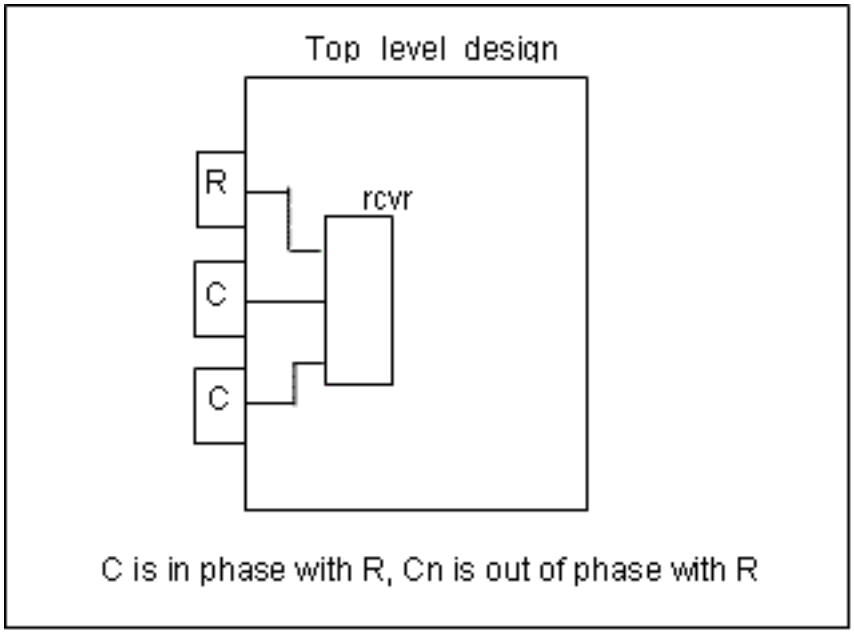
- By adding a CORRELATE attribute to the top-level module correlated pin(s) specified in the design source
- By adding a CORRELATE attribute to technology I/O cell pins connected directly to top-level module pins, specified in the technology library source
- By using a CORRELATE statement in the mode definition file or assign file

Note:

- Pins correlated by an attribute in the design source or technology library are correlated for all test modes.
- Pins correlated by a test mode CORRELATE statement are correlated only for the test mode.
- When referencing correlated pins (as in a test sequence, linehold file, or test function pin assignment), specify only the representative pin and not the correlated pins.

Correlated Pins Examples

Figure A-57 Module With Correlated Pins



Example 1: Attributes on the Top Level Design (pre-Verilog 2001 Syntax)

```
module top_level_design(..., R, C,Cn,...);
     input R;
     input C;  //! CORRELATE="R";
     input Cn; //! CORRELATE="-R";
     ...
endmodule
```

Example 2: Attributes on Technology Cell Pins (Verilog 2001 Syntax)

```
module top_level_design(..., R, C, Cn,...);
     input R;
     input C;     // No correlate specification at top level
     input Cn;
     receiver rcvr(out1, out2, R, C, Cn);
     ...
endmodule

// Technology Library Cells
module receiver(out, outn, rin, cin, cnin);
     output out, outn;
     input rin;
     (* CORRELATE="rin" *)  input cin;  //! Propagates to top level
```

```
     (* CORRELATE="-rin" *) input cnin;
     ...
endmodule
```

Example 3: Correlate Statement in Assign File

Verilog Source

```
module top_level_design(..., R, C, Cn,...);
     input R;
     input C;         // No correlate specification in design source
     input Cn;
     ...
endmodule
```

Assign File: myassign

```
assign pin R test function -SC;
...
correlate C  +R;
correlate Cn -R;
...
```

Command Line

```
build_testmode ... assignfile=myassign ...
```

Applying Logic Model Constraints

Encounter Test supports the following types of constraints:

- Logic Model constraints - Logic added to the model to prevent certain patterns from being generated. Three-state contention occurs if the constraint is violated.
- Lineholds - An input file used in ATPG that restricts the values applied to the specified primary inputs and scannable flops.
- SDC constraints - A file used in delay test generation to identify false paths and multi-cycle paths. Tests are not generated along these paths.
- Delay Test Timing Constraints - The information used in delay test generation resulting from automatic analysis of the delay model to identify paths exceeding the specified release/capture timing.

This section describes the Logic Model constraints that can be specified in the following ways:

- Using a constraint file containing constraint statements. The filename is specified on the `build_model` command.
- Specifying model attributes on a module statement in the design source.

Constraint File Syntax

```
[cellName:] constraintName expression;
```

where:

- `cellName`: The name of the cell (or module) that contains the nets being constrained. If omitted, the-top level cell is used.
- `constraintName`: A name given to the constraint. It is used as the instance name of the cell within cellName that contains the generated constraint logic.
- `expression`: The condition that must be true in order to satisfy the constraint (preventing contention).

  An expression is one of the following:

  - An arbitrary boolean expression. The expression may combine netname(s), the logical operators ^, &, and |, in addition to the provided functions such as oneHot, equiv, and so on.
  - One of the following functions using a list of netnames as arguments:

Function	Description
oneHot	exactly one net at logic 1
oneCold	exactly one net at logic 0
zeroOneHot	zero or one nets at logic 1
zeroOneCold	zero or one nets at logic 0
equiv	all nets have same value
bitwiseOneHot	bitwise version of oneHot
bitwiseOneCold	bitwise version of oneCold
bitwiseZeroOneHot	bitwise version of zerOneHot
bitwiseZeroOneCold	bitwise version of zerOneCold
bitwiseEquiv	bitwise version of equiv

Note:

- Line comments are allowed in a constraint file and start with //, --, or #.
- Block comments of the form /* ? */ are also allowed.

Constraints File Example

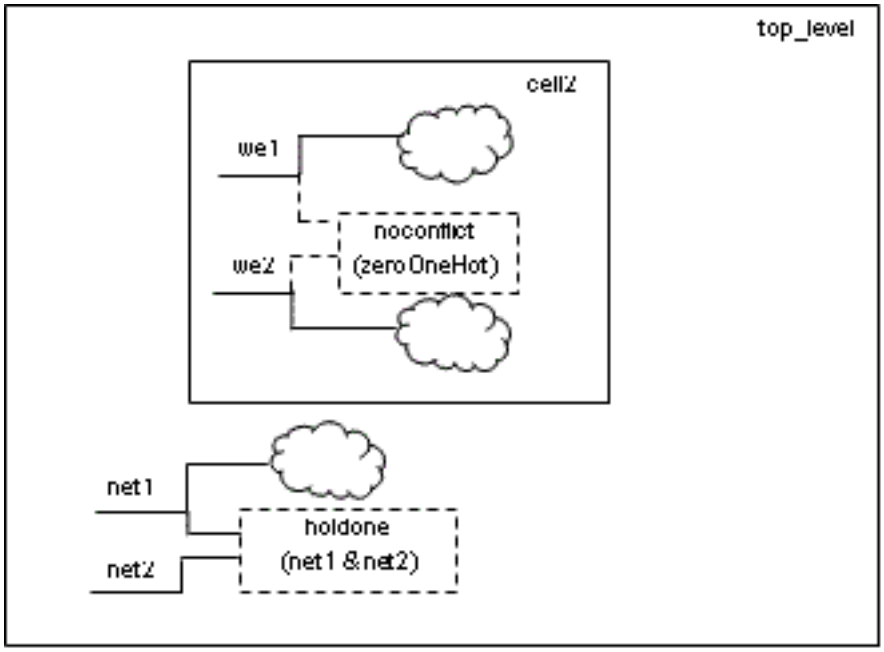
Constraints File myconstraints

```
cell2: noconflict zeroOneHot(we1, we2);
holdone net1 & net2;
```

Constraints Command Example:

```
build_model ... constraints=myconstraints ...
```

Figure A-58 Logic Model with Constraints



In the preceding figure:

- The constraint noconflict is inserted into all instances of module cell2. we1 and we2 should never be logic 1 at the same time otherwise contention will occur, causing the pattern to be thrown out.
- The constraint holdone is added to top_level because no containing cell was specified. If either net1 or net2 becomes 0, the constraint will be violated causing contention.

Note: A logic value of X on a net being constrained will also violate the constraint, causing soft contention. Depending on the `contentionreport` keyword setting used on the ATPG commands, these patterns may also be thrown out.

Defining Constraints with Model Attributes

In this method, model attributes are placed on the cell that contains the constraint. The syntax used is similar to the constraint file statements:

```
CONSTRAINTn="expression"
```

where:

- `n` = an integer (starting with 1), enumerating the constraint within the cell. The values for `n` must be consecutive.
- expression = a constraint expression as defined in "Constraint File Syntax" on page 207.

Attribute Constraint Example

Using attributes to model the constraints in Figure A-58 on page 208,

```
module top_level(...);
   //! CONSTRAINT1="net1 & net2"
   ...
endmodule;

module cell2(...); //! CONSTRAINT1="zeroOneHot(we1,we2)"
   ...
endmodule
```

Bus Syntax

The constraints file and model attributes use bus syntax to create constraints.

Example 1: Bitwise oneHot Using Bus Syntax

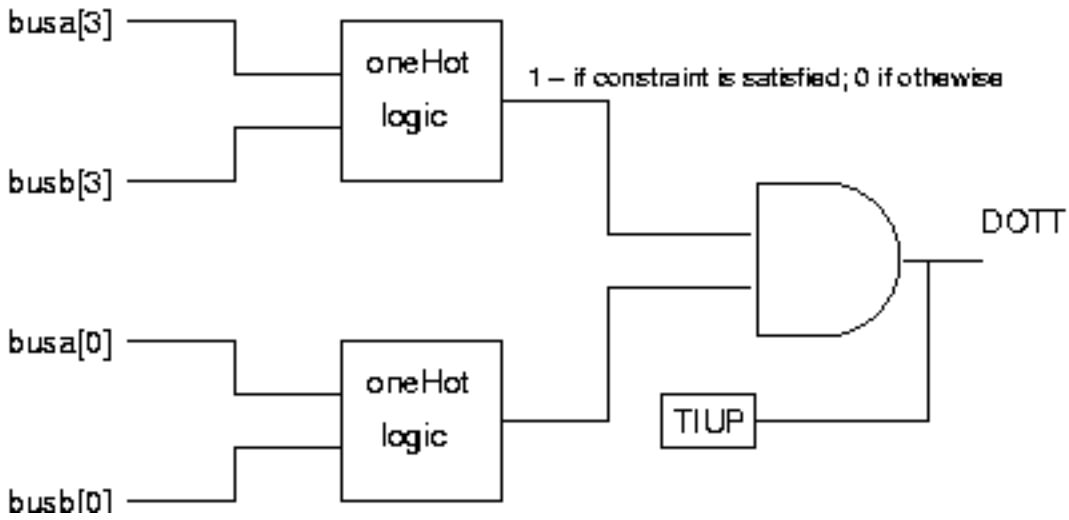
You can generate a bitwise oneHot constraint across bits 0-3 given the following Verilog example where a constraint is created between each bit bus:

```
module my_module (out, busa, busb);
//! CONSTRAINT= bitOneHot(busa[3:0],busb[3:0])
input [3:0]  busa;
input [3:0]  busb;
```

Note: Encounter Test does not support a mixture of bussed and scalar representations of bitwise constraints specifications. For example, the following examples are not supported:

```
bitOneHot(busa[3:0],busb[3:0],{scalarA,scalarB,scalarC})
bitOneCold(busa[3:2,0],busb[2:0])
```

Figure A-59 Bitwise OneHot Constraint



Example 2: Using Bus Syntax with non-Bitwise Constraints

```
module my_module (o, a, b, c);
//! CONSTRAINT= "oneHot(a[0:4])&zeroOneCold(b[4:2])"
//! CONSTRAINT= "oneHot(c[5:4], neta)"
// oneHot constraint across c[5], c[4], and neta
   input [4:0] a;
   input [7:0] b;
   input [5:0] c;
   wire neta;
   ...
endmodule
```

Removing Instance-Based Constraints

You can remove constraints on an instance-based selection list using Encounter Test's *Model Edit* function. The following examples depict “before” and “after” views for removing instance-based constraints. Refer to “Encrypting a Model (Cloaking)” on page 38 for additional information on *Model Edit* and associated syntax.

Figure A-60 Removing an Instance-Based Constraint, Example 1

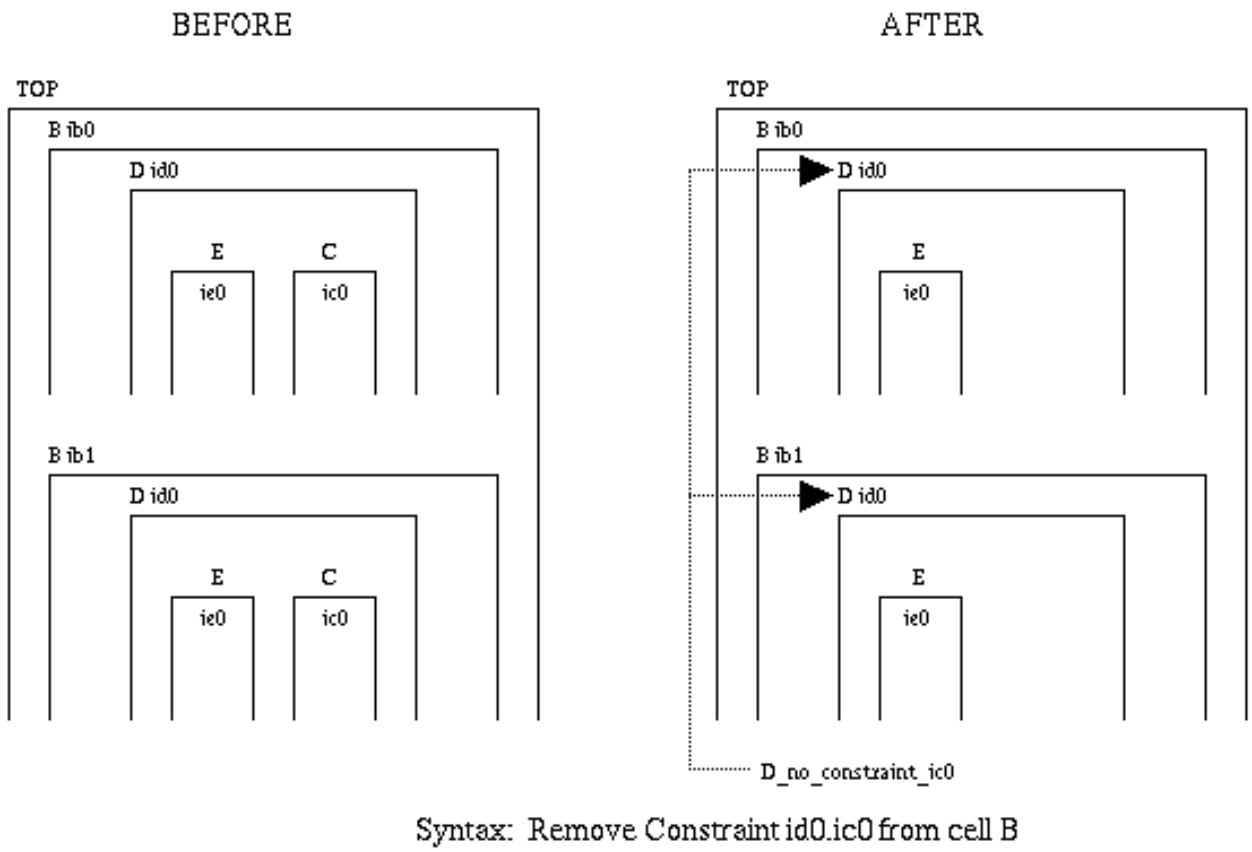
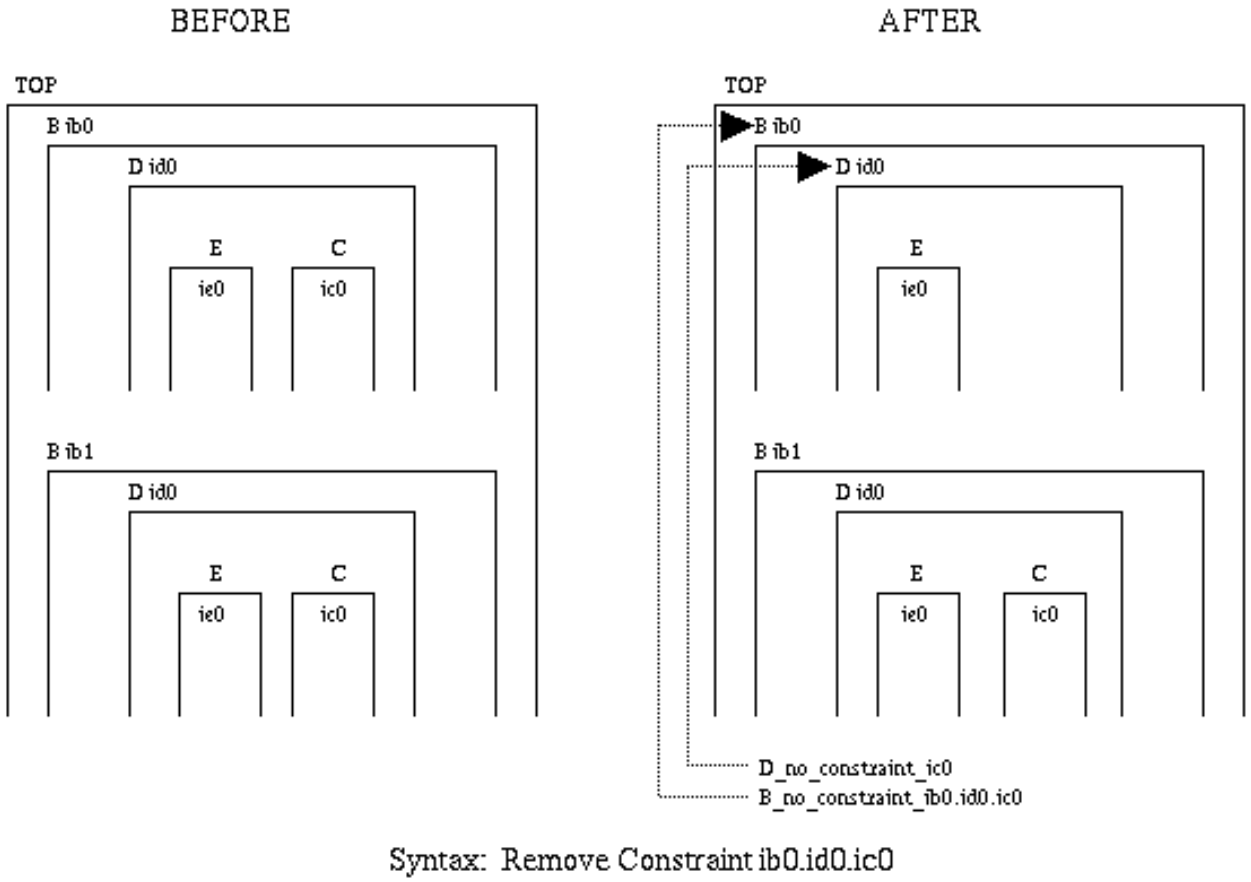


Figure A-61 Removing an Instance-Based Constraint, Example 2



Syntax: Remove Constraint ib0.id0.ic0

B

Edit Model File Syntax

This section describes the syntax of the edit model language. This language may be used to perform model edit actions using the `edit_model` command (see `edit_model` in the *Encounter Test: Reference: Commands*).

For `edit_model` examples, refer to "Encrypting a Model (Cloaking)" on page 38.

Note: For complete list of the common attributes supported by Encounter Test, refer to "Summary of Attributes" on page 91.

Comment Syntax

The following syntax statements are supported for comments:

- `/* */`
- `//`
- `--`

Language Statements

- The language is case insensitive. Therefore, although the documentation shows the statement elements in uppercase, they really can be entered in any case. Note that the names of cells, pins, nets, or properties must be in the same case as they are in the model.
- Each language statement must end with a ;
- Portions of the statements shown in square brackets [ ] are optional, they are simply included for readability.
- The word `EQUAL` can be substituted for the `=`.
- The optional prepositions in the syntax (for example `TO`, `IN`, `FOR`) are interchangeable so you can modify the preposition to fit your own style.

ADD

Use ADD to add pins, cells, nets, instances, attributes and/or test points to the model. The ADD statements are as follows:

Syntax	Description
ADD PIN *pinName1* [TO] CELL *cellName1* DIRECTION *pinDirection1;*	Adds a pin to a cell. ■ *pinName1* is the name of the pin to be added ■ *cellName1* is the name of the cell to which the pin is attached ■ *pinDirection1* is the direction of the pin (Input, Output, or InOut)
ADD CELL *cellName1;*	Adds a new cell to the model. ■ *cellName1* is the name of the cell
ADD NET *netName1* [TO] CELL *cellName1;*	Adds a net to the cell. ■ *netName1* is the name of the net to be added ■ *cellName1* is the name of the cell in which the net is contained.
ADD INSTANCE *instanceName1* [OF] CELL *cellName1* [TO] CELL *cellName2;*	Add an instance of a cell to another cell. ■ *instanceName1* is the name of the instance ■ *cellName1* is the name of the cell being instanced ■ *cellName2* is the name of the cell in which the instance is to be contained
ADD ATTRIBUTE *attrName1* [=] *attrValue1* [ON] PIN *pinName1* [ON] CELL *cellName1;*	Add a pin attribute. ■ *attrName1* is the name of the attribute ■ *attrValue1* is the value of the attribute ■ *pinName1* is the pin on which the attribute is to be placed ■ *cellName1* is the cell to which the pin is attached

Syntax	Description
`ADD ATTRIBUTE attrName1 [=] attrValue1 [ON] PIN pinName1 [ON] INSTANCE instanceName1 [OF] CELL cellName1;`	Add a pin instance attribute. ■ *attrName1* is the name of the attribute ■ *attrValue1* is the value of the attribute ■ *pinName1* is the pin on which the attribute is to be placed ■ *instanceName1* is the name of the instance of the cell to be edited ■ *cellName1* is the cell to which the pin is attached
`ADD ATTRIBUTE attrName1 [=] attrValue1 [ON] NET netName1 [ON] CELL cellName1;`	Add a net attribute. ■ *attrName1* is the name of the attribute ■ *attrValue1* is the value of the attribute ■ *netName1* is the net on which the attribute is to be placed ■ *cellName1* is the cell in which the net is contained
`ADD ATTRIBUTE attrName1 [=] attrValue1 [ON] INSTANCE instanceName1 [OF] CELL cellName1;`	Add an instance attribute. ■ *attrName1* is the name of the attribute ■ *attrValue1* is the value of the attribute ■ *instanceName1* is the instance on which the attribute is to be placed ■ *cellName1* is the cell which is instanced
`ADD ATTRIBUTE attrName1 [=] attrValue1 [ON] CELL cellName1;`	Add a cell attribute. ■ *attrName1* is the name of the attribute ■ *attrValue1* is the value of the attribute ■ *cellName1* is the cell to which the attribute applies

Syntax	Description

`ADD PRIMITIVE FUNCTION` *`simfunc`*
`PIN` *`pinName inputNetList;`*
	Add a primitive to the pin.
	■ *`simfunc`* is one of these: `and, or, nand, nor, xor, xnor, buf, inv, tiup, tidn, tix, mux` (case insensitive)
	■ *`pinName`* is a hierarchical pin name (instance specific). Specify the pin on the net closest to the insertion point of the primitive. This is important when the pin's net fans out to multiple sinks.
	■ *`inputNetList`* is in the following form:
	`INPUT netName [INPUT` *`netName`* `...]`
	`netName` is a hierarchical net name (instance specific). Each *`netName`* instance represents an input connection to the primitive gate.
	Input connections must be specified in the order defined by the Encounter Test primitive. Specify `INPUT ""` for an input to leave it unconnected.
	An example on the usage of this function is shown below:
	`ADD PRIMITIVE FUNCTION and PIN`
	`Pin.f.l.core.nl.out2_reg.SI INPUT`
	`Net.f.l.core.nl.n_28 INPUT`
	`Net.f.l.core.nl.DFT_sdi_1;`

Syntax	Description
`ADD PRIMITIVE FUNCTION` *`simfunc`* `NET` *`netName inputNetList;`*	Add a primitive to the source pin of the net. In case of multiple sources, one source pin is selected arbitrarily. ■ *`simfunc`* is one of these: `and, or, nand, nor, xor, xnor, buf, inv, tiup, tidn, tix, mux` (case insensitive) ■ *`netName`* is a hierarchical net name (instance specific). ■ *`inputNetList`* is in the following form: `INPUT netName [INPUT` *`netName`* `...]` Each *`netName`* instance represents an input connection to the primitive gate. Input connections must be specified in the order defined by the Encounter Test primitive. Specify `INPUT ""` for an input to leave it unconnected.
`ADD TEST POINT` *`testPointType`* `[AT] PIN [=]` *`pinName1;`*	Add a test point. ■ *`testPointType`* is CONTROL0, CONTROL1, or OBSERVE ■ *`pinName1`* is the name of the pin where the test point is to be inserted

ASSIGN

Syntax	Description
`ASSIGN PIN` *`hierpinName1`* `TEST FUNCTION [=]` *`tf1;`*	Add the test function attribute to a pin.

CHANGE

Syntax	Description
`CHANGE ATTRIBUTE attrName1 [=] attrValue1 [TO] attrValue2 [ON] CELL cellName1;`	Change the value of a cell attribute. ■ *attrName1* is the existing name of the attribute ■ *attrValue1* is the existing value of the attribute ■ *attrValue2* is the new value of the attribute ■ *cellName1* is the name of the cell to which the attribute applies.
`CHANGE ATTRIBUTE attrName1 [= ] attrValue1 [TO] attrValue2 [ON] INSTANCE instanceName1 [ON] CELL cellName1;`	Change the value of a attribute on an instance. ■ *attrName1* is the existing name of the attribute ■ *attrValue1* is the existing value of the attribute ■ *attrValue2* is the new value of the attribute ■ *instanceName1* is the name of the instance to which the attribute applies ■ *cellName1* is the name of the cell which is instanced

Syntax	Description
`CHANGE ATTRIBUTE` *`attrName1`* `[=]` *`attrValue1`* `[TO]` *`attrValue2`* `[ON] NET` *`netName1`* `[ON] CELL` *`cellName1;`*	Change a net attribute value. ■ *`attrName1`* is the existing name of the attribute ■ *`attrValue1`* is the existing value of the attribute ■ *`attrValue2`* is the new name of the attribute ■ *`netName1`* is the name of the net to which the attribute applies ■ *`cellName1`* is the name of the cell in which the net is contained
`CHANGE ATTRIBUTE` *`attrName1`* `[=]` *`attrValue1`* `[TO]` *`attrValue2`* `[ON] PIN` *`pinName1`* `[ON] CELL` *`cellName1;`*	Change a pin attribute value. ■ *`attrName1`* is the existing name of the attribute ■ *`attrValue1`* is the existing value of the attribute ■ *`attrValue2`* is the new value of the attribute ■ *`pinName1`* is the name of the pin to which the attribute applies ■ *`cellName1`* is the name of the cell to which the attribute is attached.

Syntax	Description
CHANGE ATTRIBUTE *attrName1* [=] *attrValue1* [TO] *attrValue2* [ON] PIN *pinName1* [ON] INSTANCE *instanceName1* [OF] *cellName1;*	Change the value of a pin instance attribute. ■ *attrName1* is the existing name of the attribute ■ *attrValue1* is the existing value of the attribute ■ *attrValue2* is the new value of the attribute ■ *pinName1* is the name of the pin to which the attribute applies ■ *instanceName1* is the name of the instance of the cell to which the pin is attached. ■ *cellName1* is the name of the cell which is instanced.
CHANGE ATTRIBUTE NAME [FROM] *attrName1* [TO] *attrName2* [ON] CELL *cellName1;*	Rename a cell attribute. ■ *attrName1* is the existing name of the attribute ■ *attrName2* is the new name of the attribute ■ *cellName1* is the name of the cell to which the attribute applies.
CHANGE ATTRIBUTE NAME [FROM] *attrName1* [TO] *attrName2* [ON] INSTANCE *instanceName1* [ON] CELL *cellName1;*	Rename a attribute on an instance. ■ *attrName1* is the existing name of the attribute ■ *attrName2* is the new name of the attribute ■ *instanceName1* is the name of the instance to which the attribute applies ■ *cellName1* is the name of the cell which is instanced

Syntax	Description
CHANGE ATTRIBUTE NAME [FROM] *attrName1* [TO] *attrName2* [ON] NET *netName1* [ON] CELL *cellName1;*	Rename a net attribute. ■ *attrName1* is the existing name of the attribute ■ *attrName2* is the new name of the attribute ■ *netName1* is the name of the net to which the attribute applies ■ *cellName1* is the name of the cell in which the net is contained
CHANGE ATTRIBUTE NAME [FROM] *attrName1* [TO] *attrName2* [ON] PIN *pinName1* [ON] CELL *cellName1;*	Rename the pin attribute ■ *attrName1* is the existing name of the attribute ■ *attrName2* is the new name of the attribute ■ *pinName1* is the name of the pin to which the attribute applies ■ *cellName1* is the name of the cell to which the attribute is attached.
CHANGE ATTRIBUTE NAME [FROM] *attrName1* [TO] *attrName2* [ON] PIN *pinName1* [ON] INSTANCE *instanceName1* [OF] CELL *cellName1;*	Rename a pin instance attribute. ■ *attrName1* is the existing name of the attribute ■ *attrName2* is the new name of the attribute ■ *pinName1* is the name of the pin to which the attribute applies ■ *instanceName1* is the name of the instance of the cell to which the pin is attached ■ *cellName1* is the name of the cell which is instanced
CHANGE CELL NAME [FROM] *cellName1* [TO] *cellName2;*	Rename a cell. ■ *cellName1* is the existing name of the cell ■ *cellName2* is the new name of the cell

Syntax	Description
`CHANGE INSTANCE NAME [FROM] instanceName1 [TO] instanceName2 [IN] CELL cellName1;`	Rename an instance. ■ *instanceName1* is the existing name of the instance ■ instanceName2 is the new name of the instance ■ *cellName1* is the cell which contains the instance to be changed
`CHANGE NET NAME [FROM] netName1 [TO] netName2 [IN] CELL cellName1;`	Rename a net. ■ *netName1* is the existing name of the net ■ *netName2* is the new name of the net ■ *cellName1* is the name of the cell in which the net is contained
`CHANGE PIN pinName1 DIRECTION [FROM] pinDirection1 [TO] pinDirection2 [ON] CELL cellName1;`	Change the direction of an existing pin. ■ *pinName1* is the name of the pin ■ *pinDirection1* is the current direction of the pin ■ *pinDirection2* is the new direction of the pin ■ *cellName1* is the name of the cell to which this pin is attached.
`CHANGE PIN NAME [FROM] pinName1 [TO] pinName2 [ON] CELL cellName1;`	Rename a pin. ■ *pinName1* is the current name of the pin ■ *pinName2* is the new name for the pin ■ *cellName1* is the name of the cell to which the pin is attached

Syntax	Description
`CHANGE TEST FUNCTION [FROM] tf1 [TO] tf2 [ON] PIN hierpinName1;`	Change the test function on the specified pin. ■ *hierpinName1* is the hierarchical pin name of the pin containing the test function to be changed. ■ *tf1* is the existing test function ■ *tf2* is the new test function
`CHANGE TEST POINT TYPE [FROM] testPointType1 [TO] testPointType2 PIN [=] pinName1;`	Change the type of an existing test point. ■ *testPointType1* is CONTROL0, CONTROL1, or OBSERVE ■ *testPointType2* is CONTROL0, CONTROL1, or OBSERVE ■ *pinName1* is the name of the pin where the test point to be deleted exists.
`CHANGE TEST POINT PIN NAME [FROM] pinName1 [TO] pinName2;`	Change the location of an existing test point from one pin to another pin. ■ *pinName1* is the name of the pin where the test point to be currently exists. ■ *pinName2* is the name of the pin to which the test point is to be moved.
`CHANGE INSTANCE CELL [FOR] INSTANCE instanceName1 [IN] CELL cellName1 [TO] CELL cellName2 ;`	Change the cell name of an instance. ■ *instanceName1* is the name of the instance ■ *cellName1* is the cell containing instanceName1 ■ *cellName2* is the new cell name for the instance **Note:** The current cell name (which is inferred) and the new cell name for the instance must have the same number of input and output pins.

CONNECT

Use `CONNECT` to connect pins to nets.

Syntax	Description
`CONNECT PIN` *`pinName1`* `[TO] NET` *`netName1`* `[IN] CELL` *`cellName1;`*	Connect a cell I/O pin to a net. ■ *`pinName1`* is the name of the cell I/O pin ■ *`netName1`* is the name of the net to connect to the pin ■ *`cellName1`* is the name of the cell to which the pin is attached and in which the net is contained.
`CONNECT PIN` *`pinName1`* `[ON] INSTANCE` *`instanceName1`* `[TO] NET` *`netName1`* `[IN] CELL` *`cellName1;`*	Connect an instance pin to a net ■ *`pinName1`* is the name of the pin ■ *`instanceName1`* is the name of the instance the pin is connected to ■ *`netName1`* is the name of the net to connect to the pin ■ *`cellName1`* is the name of the cell in which the net and the instance are contained

Note: An existing pin in the model must first be disconnected from its connecting net before being connected to a new net.

COPY

Use `COPY` to copy cells in a model.

Syntax	Description
`COPY CELL` *`cellName1`* `[TO] CELL` *`cellName2;`*	Copy an existing cell to create a new cell. ■ *`cellName1`* is the name of the existing cell. ■ *`cellName2`* is the name of the new cell to be created by copying cellName1

DELETE

Use `DELETE` to delete pins, cells, nets, instances, and attributes in a model.

Syntax	Description
`DELETE ATTRIBUTE attrName1 [= attrValue1 ] [ON] PIN pinName1 [ON] CELL cellName1;`	Delete an existing pin attribute. ■ *attrName1* is the name of the existing attribute to be deleted ■ *attrValue1* is the value of the existing attribute ■ *pinName1* is the name of the pin to which the attribute applies ■ *cellName1* is the name of the cell to which the pin is connected
`DELETE ATTRIBUTE attrName1 [= attrValue1 ] [ON] PIN pinName1 [ON] INSTANCE instanceName1 [OF] CELL cellName1;`	Delete an existing pin instance attribute. ■ *attrName1* is the name of the existing attribute to be deleted ■ *attrValue1* is the value of the existing attribute ■ *pinName1* is the name of the pin to which the attribute applies ■ *instanceName1* is the name of the instance to which the pin is attached ■ *cellName1* is the name of the cell which is instanced

Syntax	Description
`DELETE ATTRIBUTE attrName1 [= attrValue1 ] [ON] NET netName1 [IN] CELL cellName1;`	Delete an existing net attribute ■ *attrName1* is the name of the existing attribute to be deleted ■ *attrValue1* is the value of the existing attribute ■ *netName1* is the name of the net to which the attribute applies ■ *instanceName1* is the name of the instance in which the net is contained ■ *cellName1* is the name of the cell which is instanced
`DELETE ATTRIBUTE attrName1 [= attrValue1 ] [ON] INSTANCE instanceName1 [IN] CELL cellName1;`	Delete an existing instance attribute. ■ *attrName1* is the name of the existing attribute to be deleted ■ *attrValue1* is the value of the existing attribute ■ *instanceName1* is the name of the instance to which the attribute applies ■ *cellName1* is the name of the cell that contains the instance
`DELETE ATTRIBUTE attrName1 [= attrValue1 ] [ON] CELL cellName1;`	Delete an existing cell attribute. ■ *attrName1* is the name of the existing attribute to be deleted ■ *attrValue1* is the value of the existing attribute ■ *cellName1* is the name of the cell to which the attribute applies
`DELETE CELL cellName1;`	Delete an existing cell. ■ *cellName1* is the name of the cell to be deleted

Syntax	Description
`DELETE INSTANCE instanceName1 [OF] CELL cellName1 [FROM] CELL cellName2;`	Delete an existing instance of a cell. ■ *instanceName1* is the name of the existing instance to be deleted ■ *cellName1* is the name of the cell which is instanced ■ *cellName2* is the name of the cell in which the instance is contained
`DELETE NET netName1 [FROM] CELL cellName1;`	Delete an existing net. ■ *netName1* is the name of the existing net ■ *cellName1* is the name of the cell in which the net is contained
`DELETE PIN pinName1 [FROM] CELL cellName1 DIRECTION pinDirection1;`	Delete an existing pin. ■ *pinName1* is the name of the existing pin. ■ *cellName1* is the name of the cell to which the pin is attached ■ *pinDirection1* is the direction of the pin to be deleted (input, output, inout)
`DELETE TEST POINT testPointType [ON] PIN [=] pinName1;`	Delete a test point that was added on the specified pin. ■ *testPointType* is CONTROL0, CONTROL1, or OBSERVE ■ *pinName1* is the name of the pin where the test point to be deleted exists.

DISCONNECT

Syntax	Description
`DISCONNECT PIN pinName1 [FROM] NET netName1 [IN] CELL cellName1;`	Disconnect a cell I/O pin from a net. ■ *pinName1* is the name of the existing pin to be disconnected ■ *netName1* is the net to disconnect ■ *cellName1* is the name of the cell to which the pin is attached and in which the net is contained.
`DISCONNECT PIN pinName1 [ON] INSTANCE instanceName1 [FROM] NET netName1 [IN] CELL cellName1;`	Disconnect an instance pin from a net. ■ *pinName1* is the name of the existing pin to be disconnected ■ *instanceName1* is the name of the instance to which the pin is connected ■ *netName1* is the net to disconnect ■ *cellName1* is the name of the cell in which the net and the instance are contained

Note: An existing pin in the model must first be disconnected from its connecting net before being connected to a new net.

EDIT

Syntax	Description
`EDIT ROM CONTENTS [USING] ROMPATH [=] romPath1;`	Refreshes the read-only memory contents from new ROM content files. ■ *romPath1* is the path to be searched for the new ROM contents.

REMOVE

Use `REMOVE` to remove instance-based constraints in the model.

Syntax	Description
`Remove Constraint ibo.id0._constraint_Property_x`	Removes `_constraint_Property_x` from Cell d, Instance id0 which is in Cell b, instance ib0
`Remove Contraint id0._constraint_Property_x from cell b`	Removes `_constraint_Property_x` from Cell d in all Instances of Cell b

move pin RSI_SI[1] ON INSTANCE COMPACTOR to net PSI2_IM in cell XOR_COMPRESSION_RJK;

MOVE

Syntax	Description
`move pin RSI_SI[0] ON INSTANCE COMPACTOR to net PSI1_IM IN cell XOR_COMPRESSION_ABC`	Moves an instance-based pin in a cell. ■ `RSI_SI[0]` is the pin name ■ `PI1_IM` is the name of the net within the cell.

C

Design Source Examples

This section lists design source examples that correspond to schematic depictions for “Example I/O Cells” on page 161and “Clock Shaping Design Examples” on page 180.

I/O Cell Examples

This section lists source examples for:

Two-State Receiver Source

This example corresponds to Figure A-26 on page 161.

```
module
     tbb_REC
         ( D

         , Y);
     //! TYPE="CELL"
input D,
output Y;

     BUF_1
      \$blk_buf
         (.\10 ( Y )
          ,.A0 ( D )

         );
endmodule
```

Two-State Driver Source

This example corresponds to Figure A-27 on page 162.

```
module
      tbb_DRV_2ST
            ( D
            , Y);
      //! TYPE="CELL"
      input D,
      output Y;

      BUF_1
          \$blk_buf
                (.\10 ( Y )
                ,.A0 ( D )
                );
endmodule
```

Three-State Driver Source

This example corresponds to Figure A-28 on page 163.

```
module
      tbb_DRV_3ST
          ( D
          , E
          , Y);
      //! TYPE="CELL"

      input D, E;
      input Y;

      TSD
        \$blk_tsd
                ( .DATA ( D )
              , .ENABLE ( E )
              , .DOUT ( Y )
              );
endmodule
```

Open Drain Driver Source

This example corresponds to Figure A-29 on page 164.

```
module
      tbb_DRV_OD
              ( D
              , E
              , Y);
        //! TYPE="CELL"

      TIDN_0
        $blk_down
              ( .\10 (\$net_delay1)
```

```
            );

      INV_1
        $blk_invd
            ( .\10 (\$net_delay2)
            , .A0 ( D )
            );

      AND_2
        $blk_gateEnable
            (.\10 (\$net_delay3)
            , .A0 (\$net_delay2)
            , .A1 ( E )
            );

      TSD
        $blk_tsd
            ( .DOUT ( Y )
            ,.DATA (\$net_delay1)
            , .ENABLE (\$net_delay3)
            );

endmodule
```

Three-State Driver with Pull-Down Source

This example corresponds to Figure A-30 on page 165.

```
module
      tbb_DRV_PD
            ( D
            , E
            , Y );
       //! TYPE="CELL"

      TSD
        $blk_tsd
            ( .DOUT ( Y )
            , .DATA ( D )
            , .ENABLE ( E)
            );

      TIDN_0
        $blk_down
            (.\10 (\$net_delay1)
            );

      RESISTOR_1
        $blk_pull
            ( .\10 ( Y)
            , .A0 (\$net_delay1)
            );
endmodule
```

Bidirectional Driver Source

This example corresponds to Figure A-31 on page 166.

```
module
      tbb_IO
            ( D
            , E
            , Y
            , Z );

           //! TYPE="CELL"

      TSD
        $blk_tsd
            ( .DOUT ( Y )
            , .DATA ( D )
            , .ENABLE ( E)
            );

      BUF_1
        $blk_rec
            ( .\10 ( Z )
             , .A0 ( Y )
            );

endmodule
```

Clock Chopper Examples

This section lists source examples for:

Refer to "Clock Shaping Circuitry" on page 175for conceptual information.

Leading Edge Clock Chopper Source, Example 1

This example corresponds to Figure A-34 on page 181.

Verilog Example

```
module
   NCHOPL
         (\10
         ,A0
         );

   input  A0
         ;

   output \10
         ;

endmodule

module
   tbb_CLKCHOPL
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   NCHOPL
```

```
      \$blk_nbuf3
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly2 )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( C )
         ,.A1 ( \$net_slowclock )
         );

endmodule
```

Leading Edge Clock Chopper Source - Example 2

This example corresponds to Figure A-35 on page 181.

Verilog Example

```
module
   CHOPL
         (\10
         ,A0
         );

   input  A0
         ;

   output \10
         ;

endmodule

module
   tbb_CLKCHOPL2
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   CHOPL
      \$blk_buf6
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly5 )
         );

   INV_1
      \$blk_nbuf1
         (.\10 ( \$net_dly1 )
```

```
         ,.A0 ( C )
         );

   INV_1
      \$blk_nbuf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   INV_1
   \$blk_nbuf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   INV_1
      \$blk_nbuf4
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

   INV_1
      \$blk_nbuf5
         (.\10 ( \$net_dly5 )
         ,.A0 ( \$net_dly4 )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( C )
         ,.A1 ( \$net_slowclock )
         );

endmodule
```

Leading Edge Clock Chopper Source - Example 3

This example corresponds to Figure A-36 on page 182.

Verilog Example

```
module
   tbb_CLKCHOPL3
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="+SC"
         ;

   output Y
         ;

   BUF_1
```

```
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   BUF_1
      \$blk_buf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

    BUF_1
      \$blk_buf4
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

   CHOPL
      \$blk_chop
         (.\10 ( \$net_chopclk )
         ,.A0 ( \$net_dly4 )
         );

   INV_1
      \$blk_nbuf1
         (.\10 ( \$net_invpath )
         ,.A0 ( C )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_chopclk )
         ,.A1 ( \$net_invpath )
         );

endmodule
```

Leading Edge Clock Chopper Source - Example 4

This example corresponds to Figure A-37 on page 182.

Verilog Example

```
module
   tbb_CLKCHOPL4
         (Y
         ,C
         );
      //! TYPE="CELL"
```

```
   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf3
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   BUF_1
      \$blk_buf4
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   BUF_1
      \$blk_buf5
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

    CHOPL
      \$blk_buf6
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly4 )
         );

    INV_1
      \$blk_nbuf1
         (.\10 ( \$net_invclock )
         ,.A0 ( C )
         );

    OR_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
         ,.A1 ( \$net_invclock )
         );

endmodule
```

Leading Edge Clock Chopper Source - Example 5

This example corresponds to Figure A-38 on page 183.

Verilog Example

```
module
   tbb_CLKCHOPL5
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="+SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   NCHOPL
      \$blk_nbuf3
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly2 )
         );

   OR_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( C )
         ,.A1 ( \$net_slowclock )
         );

endmodule
```

Leading Edge Clock Chopper Source - Example 6

This example corresponds to Figure A-39 on page 183.

Verilog Example

```
module
   tbb_CLKCHOPL6
         (Y
         ,C
         );
    //! TYPE="CELL"

   input  C
```

```
    //! K_FLAG="+SC"
         ;

   output Y
         ;

   CHOPL
    \$blk_buf6
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly5 )
         );

   INV_1
      \$blk_nbuf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_nbuf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   BUF_1
      \$blk_nbuf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   BUF_1
      \$blk_nbuf4
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

   BUF_1
    \$blk_nbuf5
         (.\10 ( \$net_dly5 )
         ,.A0 ( \$net_dly4 )
         );

   OR_2
     \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( C )
         ,.A1 ( \$net_slowclock )
         );

endmodule
```

Trailing Edge Clock Chopper Source, Example 1

This example corresponds to Figure A-40 on page 184.

Verilog Example

```
module
   NCHOPT
         (\10
         ,A0
         );

   input  A0
         ;

   output \10
         ;

endmodule

module
   tbb_CLKCHOPT
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly1 )
         );

   NCHOPT
      \$blk_buf2
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly1 )
         );

   NCHOPT
      \$blk_nchop
         (.\10 ( \$net_chopclock )
         ,.A0 ( C )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
          ,.A1 ( \$net_chopclock )
         );
```

```
endmodule
```

Trailing Edge Clock Chopper Source - Example 2

This example corresponds to Figure A-41 on page 185.

Verilog Example

```
module
   CHOPT
         (\10
         ,A0
         );

   input  A0
         ;

   output \10
         ;

endmodule

module
   tbb_CLKCHOPT2
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   BUF_1
      \$blk_buf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   BUF_1
      \$blk_buf4
```

```
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly3 )
         );

   CHOPT
      \$blk_chop
         (.\10 ( \$net_chopclock )
         ,.A0 ( \$net_invclk )
         );

   INV_1
      \$blk_inv5
         (.\10 ( \$net_invclk )
         ,.A0 ( C )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
         ,.A1 ( \$net_chopclock )
         );

endmodule
```

Trailing Edge Clock Chopper Source - Example 3

This example corresponds to Figure A-42 on page 185.

Verilog Example

```
module
   tbb_CLKCHOPT3
         (Y
         ,C
         );
       //! TYPE="CELL"

   input  C
      //! K_FLAG="+SC"
         ;

   output Y
         ;

   CHOPT
      \$blk_chop
         (.\10 ( \$net_chopclock )
         ,.A0 ( C )
         );

   INV_1
      \$blk_inv1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );
```

```
   INV_1
      \$blk_inv2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   INV_1
      \$blk_inv3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   INV_1
      \$blk_inv4
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

   INV_1
      \$blk_inv5
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly4 )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
         ,.A1 ( \$net_chopclock )
         );

endmodule
```

Trailing Edge Clock Chopper Source - Example 4

This example corresponds to Figure A-43 on page 186.

Verilog Example

```
module
    tbb_CLKCHOPT4
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="-SC"
         ;

   output Y
         ;

   CHOPT
      \$blk_chop
         (.\10 ( \$net_chopclock )
```

```
         ,.A0 ( C )
         );

   INV_1
      \$blk_inv1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   INV_1
      \$blk_inv2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   INV_1
      \$blk_inv3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   INV_1
      \$blk_inv4
         (.\10 ( \$net_dly4 )
         ,.A0 ( \$net_dly3 )
         );

   INV_1
      \$blk_inv5
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly4 )
         );

   NOR_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
         ,.A1 ( \$net_chopclock )
         );

endmodule
```

Trailing Edge Clock Chopper Source - Example 5

This example corresponds to Figure A-44 on page 186.

Verilog Example

```
module
   tbb_CLKCHOPT5
         (Y
         ,C
         );
      //! TYPE="CELL"

   input  C
      //! K_FLAG="+SC"
```

```
        ;

   output Y
        ;

   BUF_1
      \$blk_buf1
        (.\10 ( \$net_dly1 )
        ,.A0 ( C )
        );

   BUF_1
      \$blk_buf2
        (.\10 ( \$net_dly2 )
        ,.A0 ( \$net_dly1 )
        );

   BUF_1
      \$blk_buf3
        (.\10 ( \$net_dly3 )
        ,.A0 ( \$net_dly2 )
        );

   BUF_1
      \$blk_buf4
        (.\10 ( \$net_slowclock )
        ,.A0 ( \$net_dly3 )
        );

   NCHOPT
      \$blk_nchop
        (.\10 ( \$net_chopclock )
        ,.A0 ( C )
        );

   NOR_2
      \$blk_recombine
        (.\10 ( Y )
        ,.A0 ( \$net_slowclock )
        ,.A1 ( \$net_chopclock )
        );

endmodule
```

Trailing Edge Clock Chopper Source - Example 6

This example corresponds to Figure A-45 on page 187.

Verilog Example

```
module
   tbb_CLKCHOPT6
        (Y
        ,C
        );
      //! TYPE="CELL"
```

```
   input  C
      //! K_FLAG="+SC"
         ;

   output Y
         ;

   BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

   BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

   BUF_1
      \$blk_buf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   BUF_1
      \$blk_buf4
         (.\10 ( \$net_slowclock )
         ,.A0 ( \$net_dly3 )
         );

   CHOPT
      \$blk_chop
         (.\10 ( \$net_chopclock )
         ,.A0 ( \$net_dly4 )
         );

   INV_1
      \$blk_inv1
         (.\10 ( \$net_dly4 )
         ,.A0 ( C )
         );

   NOR_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( \$net_slowclock )
         ,.A1 ( \$net_chopclock )
         );

endmodule
```

Clock Shrinker Source

This example corresponds to Figure A-46 on page 188.

Verilog Example

```
module
    tbb_CLOCKSHRINK
         (Y
         ,C
         );
      //! TYPE="CELL"

    input  C
      //! K_FLAG="-SC"
         ;

    output Y
         ;

    BUF_1
      \$blk_buf1
         (.\10 ( \$net_dly1 )
         ,.A0 ( C )
         );

    BUF_1
      \$blk_buf2
         (.\10 ( \$net_dly2 )
         ,.A0 ( \$net_dly1 )
         );

    BUF_1
      \$blk_buf3
         (.\10 ( \$net_dly3 )
         ,.A0 ( \$net_dly2 )
         );

   AND_2
      \$blk_recombine
         (.\10 ( Y )
         ,.A0 ( C )
         ,.A1 ( \$net_dly3 )
         );

endmodule
```

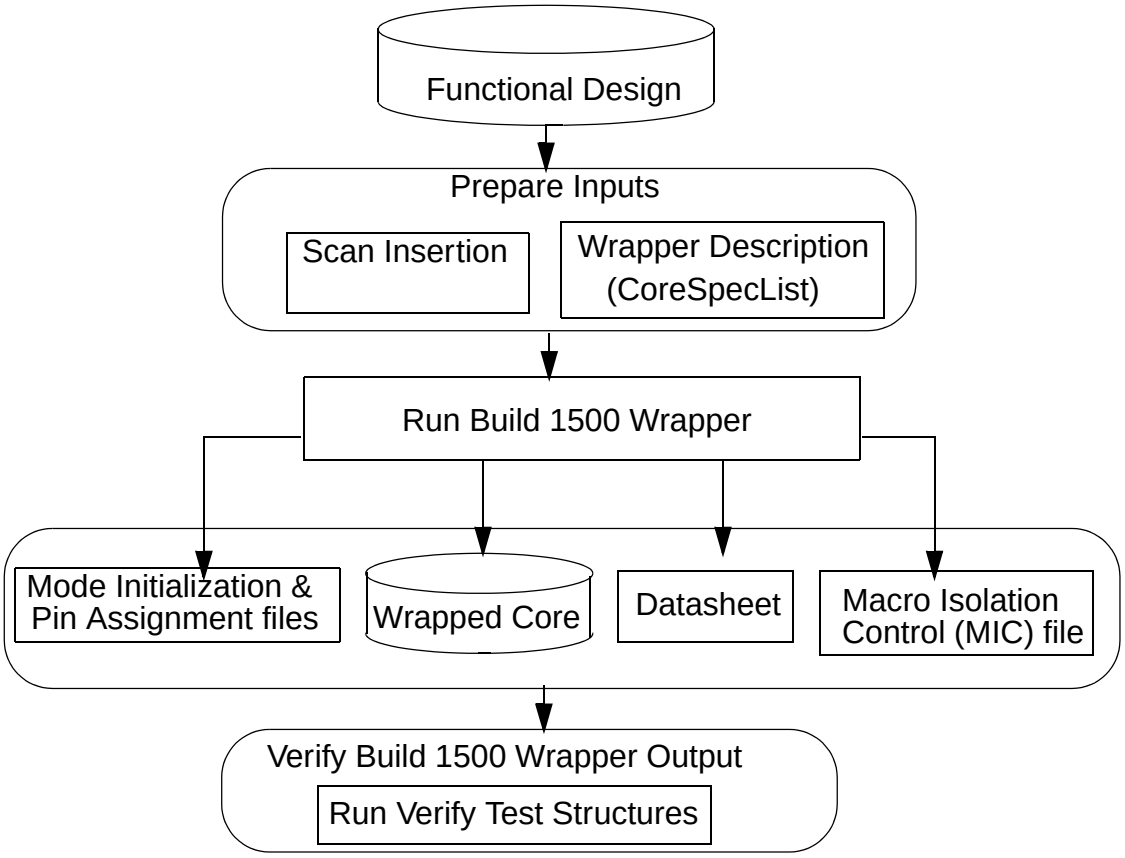
D

IEEE 1500 Core Wrapping Logic

The IEEE 1500 standard adds specialized circuitry that standardizes a test interface between the core and the design in which the core is embedded (SoC). This circuitry is called a *wrapper* since it provides a test interface which *wraps around* the core. Once the core has been instantiated in a SoC, this wrapper can be used to test the core as well as the user-defined logic between cores.

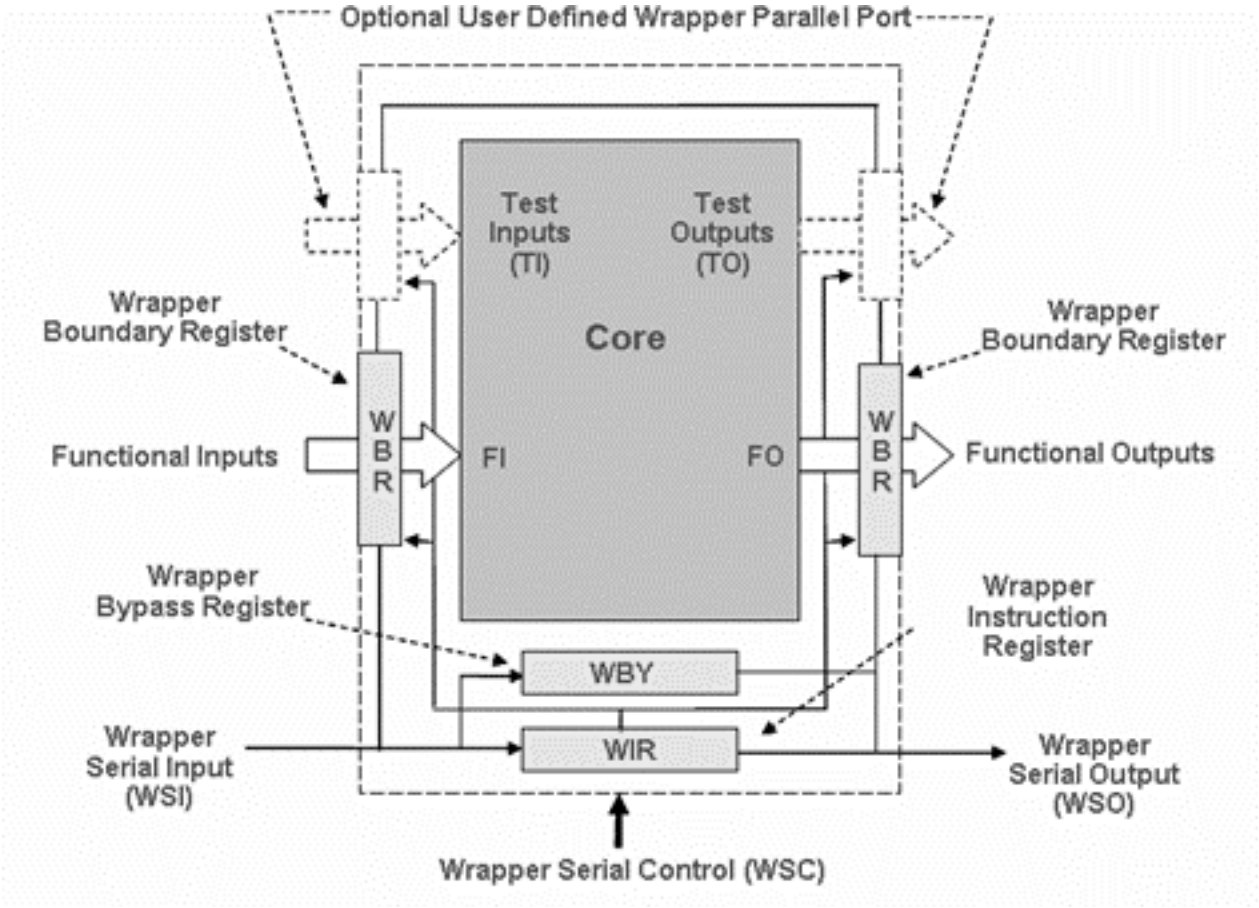
Figure D-1 depicts a use model for Build 1500 Wrapper .

Figure D-1 Build 1500 Wrapper Use Model



The architecture of the IEEE 1500 wrapper circuitry that is created is similar to the architecture shown in Figure D-2.

Figure D-2 IEEE 1500 "Wrapper" Logic Added Around a Core



The following hardware elements are illustrated in Figure D-2:

- A Wrapper Boundary Register (WBR) that consists of one or more *Wrapper Boundary Cells* which interface core functional terminals to the SoC logic.
- A Wrapper Bypass Register (WBY) that provides a minimum length scan bypass of the WBR.
- A Wrapper Instruction Register (WIR) and decode logic that controls the operation of the wrapper circuitry.
- A mandatory Wrapper Serial Port (WSP) that consists of a Wrapper Scan Input (WSI), a Wrapper Scan Output (WSO), and a set of Wrapper Serial Control (WSC) pins.

- An optional Wrapper Parallel Port (WPP) which is also a collection of wrapper terminals that is enabled only when a wrapper *parallel* instruction is present in the WIR.

Components of IEEE 1500 Wrapper Circuitry

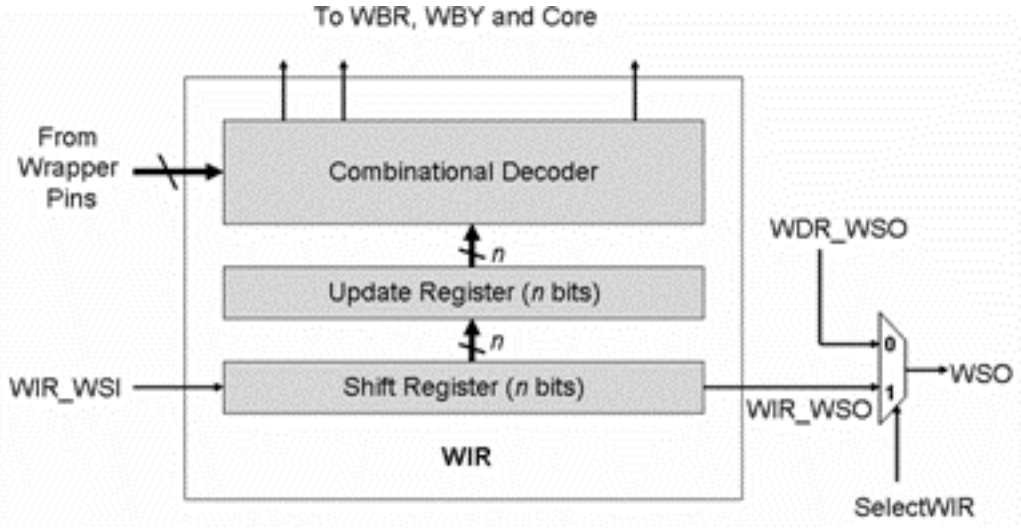
The elements of the IEEE 1500 Wrapper are described in this section.

Wrapper Instruction Register (WIR)

The modes of operation of WBR, WBY, and core depend on the instruction that is loaded into the WIR. Each instruction in the WIR corresponds to a test mode in which either the core or the logic external to the core is tested.

Build 1500 Wrapper creates a WIR that contains a serial Shift Register, an Update Register with parallel load capability, and decode logic that generates control signals to the WBR, WBY, and core. Figure D-3 on page 253 shows a block diagram of the WIR that is created.

Figure D-3 WIR Structure



The WIR supports the following instructions (mandatory instructions are denoted by boldface):

- `WS_BYPASS` - The WBY is selected between WSI and WSO, while the WBR cells are placed in functional mode. This is a mandatory instruction and is implemented regardless of whether it is specified in the CoreSpecList file. The all-zero opcode is

reserved for the WS_BYPASS instruction. Any unused opcode also maps to the WS_BYPASS instruction.

- WS_EXTEST - The WBR is selected between the WSI and WSO path, and placed in outward facing mode. This instruction applies test data to the SoC from the WBR cells on the core outputs and captures data from the SoC into the WBR cells on the core inputs. This is a mandatory instruction and is implemented regardless of whether it is specified in the CoreSpecList file.

- WP_EXTEST - This instruction is similar in functionality to the WS_EXTEST instruction except that the Wrapper Parallel Port (WPP) terminals are used to control the WBR operation. This is an optional instruction as per the standard, and is implemented only if specified.

- WP_INTEST - The WBR is used to apply test data to the core and also capture response from the core outputs. The Wrapper Parallel Port (WPP) terminals are used to control the WBR and core operation during this parallel instruction. This is an optional instruction as per the standard, and is implemented only if specified.

- WS_INTEST_SCAN - This is a serial instruction that places the WBR and core scan chains in the WSI-WSO path. Similar to WP_INTEST, this instruction is used to test the core. The core scan chains are daisy-chained along with the WBR, and lock-up latches are added where appropriate. This is an optional instruction as per the standard, and is implemented by default only if the WP_INTEST instruction is not specified. The standard mandates that at least one instruction should be implemented that tests the core.

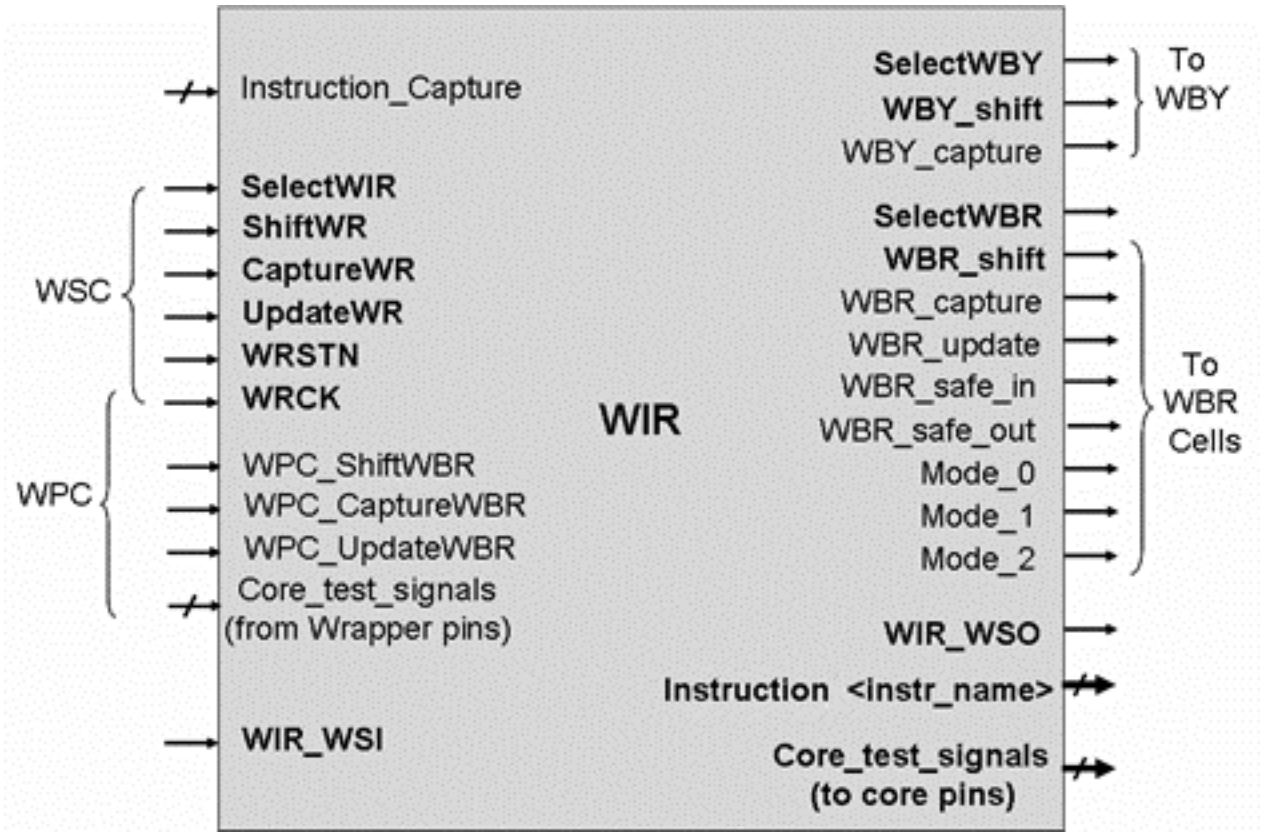
- WP_INTEST_COMPRESSION - This instruction will cause the insertion of an XOR compression macro within the wrapper shell. The WBR and core scan chains will be segmented appropriately and connected to WPP terminals through the compression macro.

The minimum size of a WIR instruction opcode is 2 bits. Unless otherwise specified, the default implemented instructions are WS_BYPASS, WS_EXTEST, and WS_INTEST_SCAN. A minimum 2 bits each are required to implement the WIR's Shift and Update registers. More bits may be required depending on the specified instructions.

If the core has test signals that are static in one or more core test modes, then these signals are provisioned with a WIR control so that the static value can be applied by the WIR.

Figure D-4 on page 255 shows a list of created input and output pins on the WIR. Pin names that are in bold-face are present on the WIR regardless of the instructions and the WBR cell types that are required. Other pins will be present depending on the implemented WIR instructions and selected WBR cell types.

Figure D-4 I/O Pins of the WIR



The following are descriptions of the pins:

- `Instruction_capture_`<n-1:0> (Optional) - Input pins that allow user signal values to be captured in the WIR Shift Register when CaptureWR and SelectWIR are asserted and there is a pulse on WRCK. Pins will exist only if specifying the data to be captured into the WIR. The width of this bus will equal the number of bits in the WIR Shift Register.
- The following are mandatory Wrapper Serial Control (WSC) signals:
  - `WRCK` - An ungated clock that feeds the clock inputs of the serial shift register and the parallel update register.
  - `WRSTN` - An active low wrapper reset signal that resets the WIR and puts the wrapper into functional mode.
  - `SelectWIR` - Active high signal that selects the WIR between the WSI-WSO pins of the wrapper.

- ShiftWR, CaptureWR, UpdateWR- Active high signals that enable shift, capture and update of the register that is selected between the wrapper WSI-WSO pins by the current instruction.
- AUXCKn - The clock signals of the core that are required in order to shift and capture test data within the core internal scan chains are categorized as auxiliary clocks. They will be part of at least either the WSC or WPC since only the WS_INTEST_SCAN and WP_INTEST instructions are currently supported, and the user has to implement at least one of these two instructions. These clock signals are not shown in the preceding figure since they are directly connected to the core, and do not require gating.

- Optional Wrapper Parallel Control (WPC) signals. Refer to "Wrapper Parallel Port (WPP)" on page 259.
- The WIR_WSI and WIR_WSO signals that hook up to the mandatory WSI and WSO terminals on the wrapper
- Control signals connected to WBY:
  - WBY_shift - active high signal that enables shifting of the WBY register
  - WBY_capture (Optional) - active high signal that allows capture of data into the WBY register. The pin will exist only if the data to be captured into the WBY is defined.
  - SelectWBY - active high signal that selects the WBY between the WSI-WSO pins provided the SelectWIR signal is active low.
- Signals corresponding to the currently loaded instruction in the WIR. Since these signals are one-hot encoded, only one of the signals is currently active high.
  - INSTR_*instr_name* - active high when instruction instr_name has been loaded into the WIR. There is one pin each for WS_BYPASS and WS_EXTEST. WIR pins may also be present for WS_INTEST_SCAN, WP_EXTEST and WP_INTEST.
- SelectWBR control signal that indicates the WBR has been selected. The signal goes high when the WBR has been selected by the current Serial or Parallel instruction.
- Control signals connected to all WBR cells in the design, regardless of whether it is an input WBR cell or an output WBR cell.
  - WBR_shift - active high when the WBR is in shift mode.
  - WBR_capture (Optional) - active high when data is captured in the WBR. If no WBR cell in the WBR has a CaptureEn pin on its cell interface, this signal is not present on the WIR output.

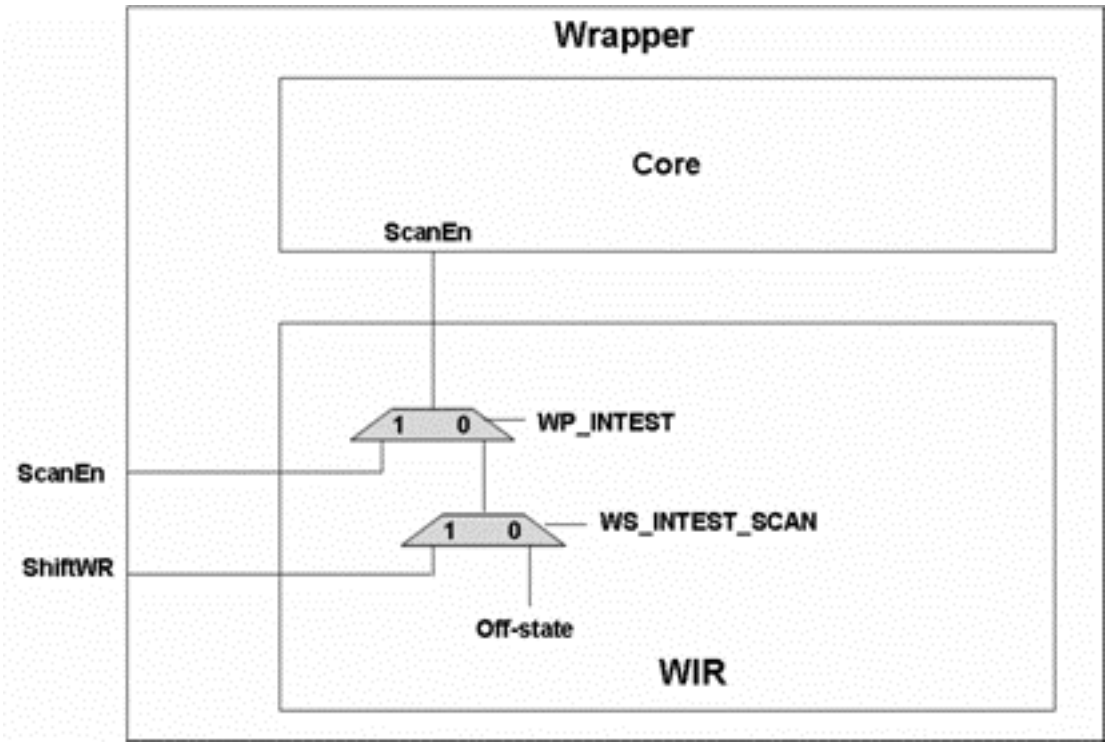
- `WBR_update` (Optional) - active high when the data in the WBR is updated. This signal exists only if there is at least one WBR cell in the wrapper that supports the update event, and thus has an `UpdateEn` input pin.

- Control signals to WBR cells that allow the cells to provide hard-wired safe (or Guarded) data at their CFO pins.
  - `WBR_safe_in` (Optional) - This signal is connected to the `SafeEn` input of the WBR cells that are on core input terminals. The signal is active high when the WBR cells are required to provide safe (Guarded) data at their output. For this signal to be implemented, there should exist at least one WBR cell on the core inputs that supports providing safe (Guarded) data.
  - `WBR_safe_out` (Optional) - This signal is connected to the `SafeEn` input of the WBR cells that are on core output terminals. The signal is active high when the WBR cells need to provide safe (Guarded) data at their output. For this signal to be implemented, there should exist least one WBR cell on the core outputs that supports providing safe (Guarded) data.
- Control signals to configure WBR cells into their various states for different instructions. Note that depending on the WBR cell types that need to be inserted in the wrapper, only one or all three of these pins will exist on the WIR.
  - `Mode_0` - This signal will be logic high for instructions {`WS_INTEST_SCAN`, `WP_INTEST`, `WS_EXTEST`, `WP_EXTEST`}. Otherwise it will be at a logic low level.
  - `Mode_1` - This signal will be logic high for instructions {`WS_INTEST_SCAN`, `WP_INTEST`}. Otherwise it will be at a logic low level.
  - `Mode_2` - This signal will be logic high for instructions {`WS_EXTEST`, `WP_EXTEST`}. Otherwise it will be at a logic low level.
- `Core_test_signals` (to core) - These are test signals of the core that control the core during the different core test modes. The IEEE P1500 standard dictates that test signals of the core that statically enable one or more core test modes shall be provisioned with WIR control. Build 1500 Wrapper uses the following strategy for test signals:
  - If a test signal of the core has a static value during each test mode of the core (that is, during all variations of `INTEST`) then this pin on the core is exclusively controlled by the WIR. The decoding logic ensures that during instructions that test the core, this test signal is provided the required static value. During other instructions (i.e., non-`INTEST` flavors) and during functional mode, unless specific values are provided, the pin on the core will be provided the *off-state* value of this test signal, which is the value of that signal in functional mode. Note that this pin on the core is not provided with a corresponding port on the Wrapper.

- If a test signal of the core is a scan enable, then this signal has a *dynamic* value during the core test modes, since the scan enable may change value during a test mode. During Serial instructions that test the core, gating logic is added inside the WIR so that the `ShiftWR` signal feeds the pin on the core that corresponds to this test signal. This ensures that only the WSC terminals are used during Serial instructions. During Serial instructions that do not test the core, the WIR ensures that the core pin is held to its off-state (functional state).
- If Parallel instructions that test the core are implemented, a port is created on the wrapper that corresponds to this signal. The WIR logic ensures that this wrapper port has exclusive control over the core pin during a parallel `INTEST` instruction. This port will thus become part of the WPC terminals. This is to enable the WPC terminals to operate the core scan chains during `WP_INTEST`. During functional operation, this signal is gated by the WIR to its off-state. Note that if `WP_INTEST` is not implemented, then a dedicated port on the wrapper corresponding to the core's enable pin is not created.
- If a test signal of the core is static during some test modes (instructions), and dynamic during other test modes, a combination of the above two cases is performed. During the test modes when this signal has a static value, the WIR provides the controlling value. While this is a dynamic test signal, depending on whether it is a Serial or Parallel instruction that tests the core, either the WIR or the Wrapper port provides the test value for this signal.

Figure D-5 on page 259 shows an example of the logic that feeds a dedicated scan enable (`SE`) pin on the core. Note that this figure only illustrates the functionality, and this may not be the actual hardware that will be inserted in the WIR.

Figure D-5 Example of Logic Controlling a Dedicated Scan Enable of the Core



Wrapper Parallel Port (WPP)

The WPP provides a parallel access mechanism to the Wrapper that facilitates significant reduction in the test time. The IEEE P1500 standard mandates that only the WPP should be used to control the operation of the WBR and/or core scan chains during parallel instructions. The WPP consists of one or more (usually more than one) Wrapper Parallel Input (WPI) terminals, an equal number of Wrapper Parallel Output (WPO) terminals, and some Wrapper Parallel Control (WPC) terminals. Since implementation of the parallel instructions is optional, the WPP is optional as well.

The WPI and WPO terminals are used to access the WBR and core internal scan chains during parallel instructions. During the `WP_EXTEST` instruction, the WPI and WPO terminals are used to shift test data in and out of the WBR. During `WP_INTEST`, the WPI and WPO terminals are used to access the WBR and core internal scan chains.

The WPC signals are used to control the operation of the WBR and core internal scan chains during Parallel instructions. The standard mandates that non-clock WSP terminals cannot be used as part of the WPP. In order to be compliant with the standard, the implemented WPC signals are as follows.

- WRCK - An ungated clock that feeds the clock inputs of the WBR cells.
- AUXCKn - The clock signals of the core that are required in order to shift and capture test data within the core internal scan chains are categorized as auxiliary clocks. Note that both the WRCK and the AUXCKn signals are reused from the WSC signals. They will not be part of the WPC if WP_INTEST is not needed.
- WPC_ShiftWBR - An active high signal that enables the shifting of data through the WBR cells during a Parallel instruction. Its operation is similar to the WSP's ShiftWR terminal.
- WPC_CaptureWBR - An active high signal that enables capture of data in the WBR cells during a Parallel instruction. Its operation is similar to the WSP's CaptureWR terminal.
- WPC_UpdateWBR (Optional) - An active high signal that enables the update of data within the WBR cells during a Parallel instruction. This WPC terminal will be implemented only if the WBR contains cells that have the Update storage element.
- Core_test_signals (Optional) - These wrapper ports are needed to operate the core during parallel INTEST instructions. If only WP_EXTEST is implemented, there will not be any dedicated ports on the wrapper that correspond to the core's test signals. This is because during serial instructions, only the WSP is used to operate the core (if necessary). Note that these dedicated ports on the wrapper will have the same name as the core's test signals to which they correspond. If the core has a pin that is a scan enable during at least one parallel INTEST instruction, then this pin will have a corresponding port on the wrapper. "Wrapper Instruction Register (WIR)" on page 253 describes how the core's test signals can be accessed from the wrapper ports during parallel instructions.

Note that the non-clock WPC terminals (WPC_ShiftWBR, WPC_CaptureWBR, WPC_UpdateWBR, and Core_test_signals) on the Wrapper are inputs to the WIR. This is necessary in order to add the gating logic that allows these signals access to the WBR and core scan chains only during parallel instructions. The WRCK and AUXCKn signals do not require gating.

During a parallel instruction that requires shifting of the WBR, and when the WPC_ShiftWBR pin on the Wrapper is asserted, the decoding logic inside the WIR ensures that there is an active high value on the WBR_shift pin of the WIR. Once the WBR_shift signal is asserted, the WBR cells are in shift mode, and any pulses on the WRCK will load test data into the WBR cells through the WPI terminals, and unload test data through the WPO terminals.

Similarly, asserting the WPC_CaptureWBR signal asserts the WBR_capture pin on the WIR, and this puts the WBR cells into capture mode. Provided the Wrapper has WBR cells that respond to the Update event, asserting the WPC_UpdateWBR signal will allow the WBR cells to respond to the Update event.

Note that if the WRCK pulses, the WBR cells will hold state if none of the WPC_ShiftWBR, WPC_CaptureWBR, and WPC_UpdateWBR signals are active.

During WP_INTEST, asserting the Core_test_signals that correspond to the core's scan enables will put the core's scan chains into shift mode. Any pulses on the AUXCKn terminals of the Wrapper will load test data into the core's scan chains through the WPI terminals and unload the response data through the WPO terminals. WPC_ShiftWBR and Core_test_signals can then be de-asserted in order to capture the response of the core to the test data. Note that WPC_CaptureWBR has to be asserted (to put the WBR in capture mode) before applying the capture clocks (WRCK and AUXCKn). Once the response has been captured, WPC_ShiftWBR and the Core_test_signals need to be asserted to shift out the response data.

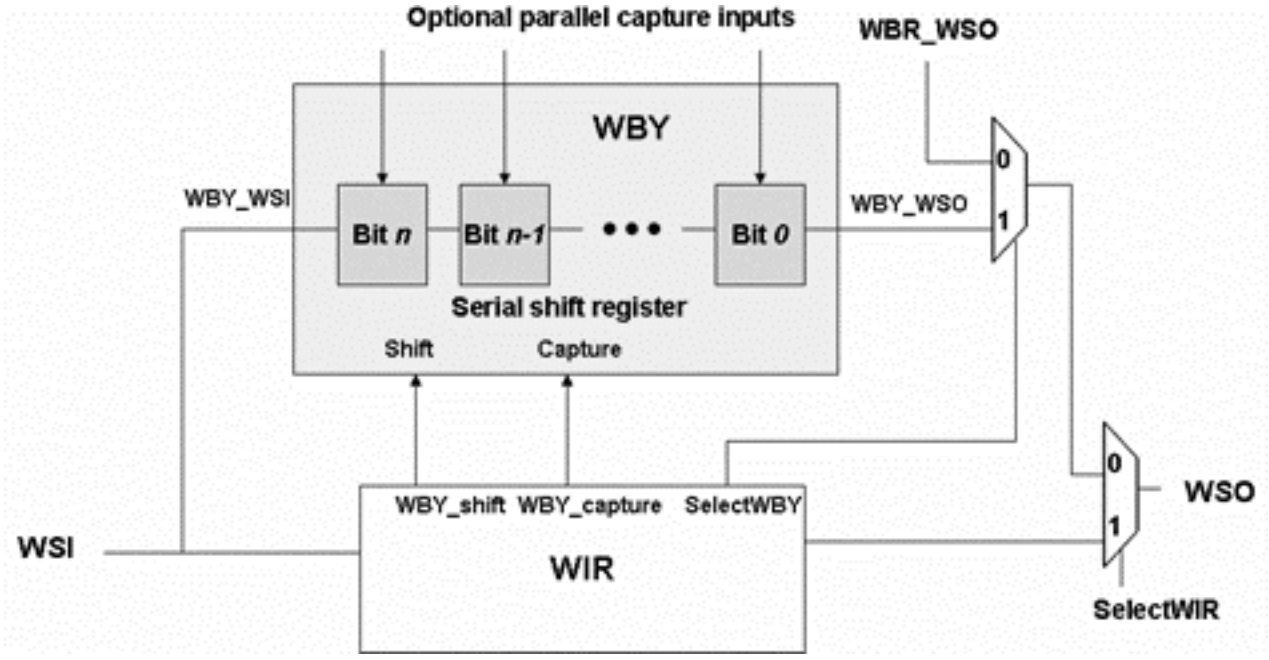
Note that the design of the WPC does not allow the WPP and WSP to be used at the same time since the WRCK clock terminal is included in both the WSP and WPP ports.

Wrapper Bypass Register (WBY)

The WBY is a mandatory register, and is always selected between the WSI-WSO pins during the WS_BYPASS instruction. The WBY contains a serial shift register. Since the purpose of a WBY is to provide a minimum length shift path through the Wrapper, it is recommended that the shift register be 1 bit long, though a longer bit length may be specified. Due to the lack of an update register, the WBY cannot respond to the Update event, but does respond to the Shift event. The WBY does respond to the Capture event if so specified.

Figure D-6 on page 262 illustrates the block diagram of the WBY.

Figure D-6 WBY Block Diagram



The `WBY_shift` pin of the WIR is connected to the `Shift` pin on the WBY. The `WRCK` connected to the WBY is an ungated clock. The `SelectWBY` pin of the WIR is connected to the select line of the MUX at the output of the WBY. When the `WS_BYPASS` instruction is loaded into the WIR, the select line of the MUX selects the `WBY_WSO` as feeding the output of the MUX. Any data shifted in through the WSI is loaded into the WBY through `WBY_WSI`, and data shifted out through `WBY_WSO` goes out of the Wrapper through the WSO terminal. Since the WSC is used during this instruction, `ShiftWR` needs to be active in order to shift data through the WBY. This causes the `WBY_shift` pin on the WIR to have an active high value, and puts the WBY into shift state. Data can be shifted through the WBY by repeated pulses of the `WRCK`.

As per the IEEE 1500 standard, the parallel capture inputs of the WBY are optional. Any core pins or constant values may be specified for capture in the WBY. The WIR will have a `WBY_capture` pin be connected to the `Capture` pin on the WBY. An active high value on `CaptureWR` puts the WBY into capture mode, and any subsequent pulse on `WRCK` captures data into the WBY shift register. Note that if the data to be captured into the WBY is not specified, neither the `WBY_capture` nor the `Capture` pin will exist.

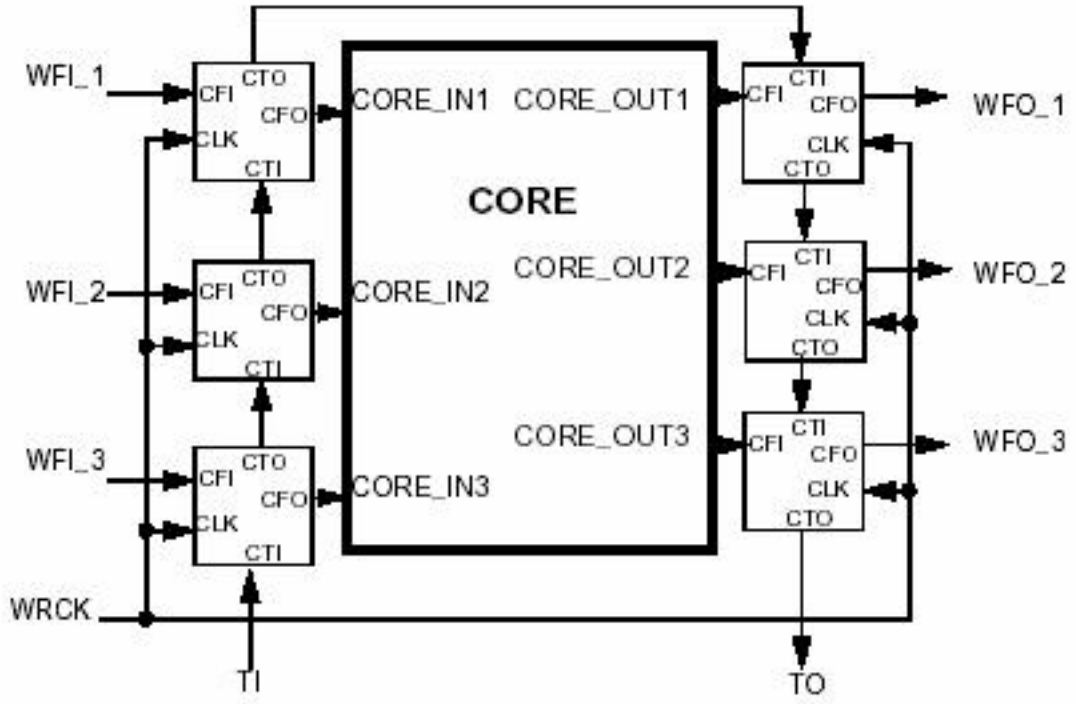
Wrapper Boundary Register (WBR)

The WBR is a mandatory shift register comprised of boundary cells (WBR cells) on the functional pins of the core. Each core functional terminal, except clocks and resets, is provided with a WBR cell. Each WBR cell can be placed into various modes such that the core either gets test data from the WBR cell or functional data from the SoC. The response of the core to the test data can also be captured into the WBR and shifted out for analysis. The test data is shifted in and out of the WBR either through the WSI-WSO terminals, or through the Wrapper Parallel Input and Output (WPI, WPO) terminals.

The boundary cells are shipped with Encounter Test. Currently, user-defined boundary cells are not supported. Specify a supported WBR cell type to connect to a core functional port. If not specified, a WBR cell is chosen by default.

Depending on the current instruction in the WIR, the WBR is configured either as a single serial shift register or will be split up into multiple chains. During Serial instructions, the WBR will act as a single scan chain consisting of all the WBR cells. During the `WS_INTEST_SCAN` instruction, the core internal scan chains are daisy-chained along with the WBR cells to form a single scan chain. During Serial instructions, the WSI and WSO are used to shift test data in and out of the WBR, and the WSC is used for test control. Figure D-7 on page 264 (Figure 12.1 in the IEEE 1500 standard) shows an example of the WBR in a serial configuration where `TI` and `TO` are connected to `WSI` and `WSO` respectively.

Figure D-7 Example of WBR Serial Configuration



The `SelectWBR` signal from the WIR indicates whether the WBR has been selected by the current Serial or Parallel instruction. The clock to the WBR cells is not gated inside the WIR, and comes directly from the Wrapper ports. Refer to "Wrapper Instruction Register (WIR)" on page 253 for descriptions of the control signals coming out of the WIR that control the operation of the WBR cells.

Segmentation of the WBR During Parallel Processing

During the optional Parallel instructions, the WBR can be split up into one or more scan segments, and the Wrapper Parallel Input and Output (`WPI`/`WPO`) terminals are used to access these segments of the WBR. Note that the WPC will be used to control the `WBR` and/or core operation during Parallel instructions. Refer to "Wrapper Parallel Port (WPP)" on page 259 for a description of the WPC signals.

To configure the WBR for either serial or parallel instructions, MUXes are inserted between the WBR cells to allow either the `WSI`/`WSO` or the `WPI`/`WPO` terminals to access the WBR. For example, MUX m1 in Figure D-8 on page 265 is necessary to select between the `WSI` and `WPI[0]` terminals.

Figure D-8 Example of WBR Segmentation During Parallel Instructions

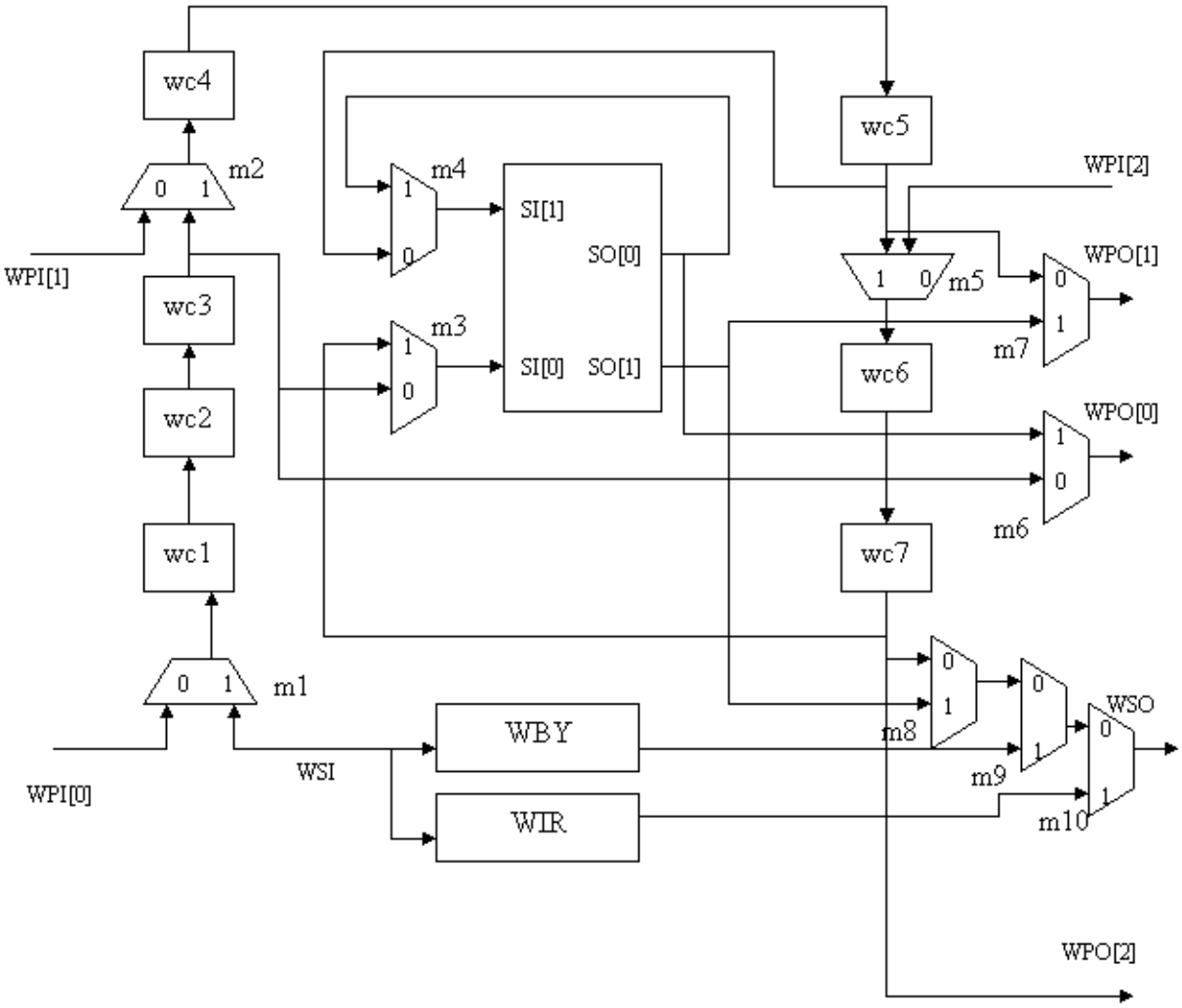


Table D-1 MUX Control Signals During Instructions that Select the WBR

Instruction	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
WP_EXTEXT	0	0	X	X	0	0	0	X	X	X
WP_INTEST	0	0	0	0	0	1	1	X	X	X
WS_INTEST_SCAN	1	1	1	1	1	X	X	0	0	0
WS_EXTEST	1	1	X	X	1	X	X	0	0	0

Figure D-8 on page 265 (similar to examples in the IEEE 1500 standard) shows an example of how the WBR has been configured into 3 segments during the WP_EXTEST instruction, with the WPI[0]. WPI[1], and WPI[2] terminals feeding these segments. Table D-1 on page 265 shows the values to be placed on the select line of the MUXes to allow the WPI terminals access to the WBR segments.

During WP_INTEST, the core internal scan chains are also placed in the scan path formed by the WPI-WPO terminals. The example in Figure D-8 on page 265 shows how the WBR and core scan chains form three segments and receive data through three WPI terminals. The WPI[0] and WPI[1] terminals have some wrapper cells in their path followed by a core scan chain. Since there are only 2 core scan chains, the third WPI terminal, WPI[2], has only wrapper cells on its path. If there were more core scan chains they would be distributed evenly over the number of WPI terminals. Table D-1 on page 265 shows the values that need to be placed on the MUXes to allow the WPI/WPO terminals access to the WBR and core scan chains.

Since different parallel instructions may choose to segment the WBR differently, MUXes also are inserted in order to choose between these different configurations.For example, MUX m2 in Figure D-8 on page 265 ensures that during WP_EXTEST, WBR cell WC4 receives data from terminal WPI[1]. During other instructions, it will receive data from the previous WBR cell, WC3.

To properly control the operation of these MUXes during different instructions, control logic needs to be generated. Given how the WBR needs to be segmented, and using the Inst_<i*nstr_name*> signals coming out of the WIR, the required MUXes and their control logic will be generated by this application.

Default segmentation of the WBR can be done if specifying the width of the WPI/WPO bus (*n*). During WP_EXTEST the WBR is segmented by dividing the number of WBR cells by the bus width *n*. During WP_INTEST, the core scan chains are evenly distributed over the *n* WPI terminals and some of the scan chains may be daisy-chained as necessary. If there are less scan chains than the bus width, one WPI terminal is associated with each scan chain input and there will be some WPI terminals that are not associated with any scan chain. The WBR cells are always evenly divided over the remaining WPI terminals.

Verifying Build 1500 Wrapper Output

Use Verify Test Structures (TSV) to verify compliance to the instruction. One mode initialization sequence file and one pin assignment file are generated for each instruction (test mode) that is implemented in the wrapper. Encounter Test uses the mode initialization sequence to load the instruction opcode into the 1500 WIR. The pin assignment file describes the ATPG test functions for the relevant test pins on the wrapper core.

Each instruction implemented within the wrapper can be considered a separate test mode. Build a testmode for each instruction implemented within the 1500 wrapped core and run Verify Test Structures on each test mode to ensure that the correct wrapper register for a particular instruction is being identified as a scan chain, and is of the correct length. For example, during the `WS_BYPASS` instruction, it is desirable to select the Wrapper Bypass Register between WSI and WSO, with a 1 bit length. View the Verify Test Structures logs for each instruction to verify the expected design intent is met.

Important

If the core is a blackbox during `build_model`, you will not be able to verify the `WS_INTEST_SCAN` and `WP_INTEST` instructions since the core internal scan chains need to be visible for TSV to verify that they are correctly connected to the wrapper level pins/chains.

The wrapper shell itself is not scan inserted. Scan insertion on the wrapper shell is not necessary since the wrapper flops are already stitched in a manner so that they behave as scan chains during the different instructions. This behavior is verified by Verify Test Structures.

Index

Symbols

A

B

C

D

E

F

H

I

K

L

M

R

S

T

U

V

W