XG Mode
User Guide

Version X-2005.09, September 2005

Comments?
Send comments on the documentation by going to http://solvnet.synopsys.com, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

Printed in the U.S.A.

Document Order Number: 38065-000 ZA
XG Mode User Guide, version X-2005.09

Contents

Preface

This preface includes the following sections:

What’s New in This Release

This section describes the new features, enhancements, and changes made to XG mode in version X-2005.09. These features and enhancements are available only in XG mode. In general, features added to DB mode in version X-2005.09 are also available in XG mode, but they are not documented in this guide.

New Features

In version X-2005.09, the following new features have been added to support XG mode:

- The following DFT Compiler features are now supported in XG mode:
    - BSD Compiler
    - SocBIST
- Design Compiler FPGA now supports XG mode

Enhancements

In version X-2005.09, XG mode includes the following enhancements:

- The following commands work on instances throughout the hierarchy, instead of just on instances within the current design:
  - Netlist editing commands
  - `ungroup`, `group`, and `uniquify` commands
  - `change_link` command
  - `set_size_only` command

  For more information, see "current_design Command" on page 1-11.

- The `compile_ultra` command automatically ungroups small hierarchies to improve the quality-of-results

- The `check_design` command has been enhanced to check for additional design problems

  For information about the `check_design` enhancements, see "Differences in Behavior" on page 3-2.

- Support for SDC version 1.5

  For details about SDC version 1.5, see the *Using the Synopsys Design Constraints Format Application Note*.

- Support for Milkyway-based DEF and PDEF generation

  The DEF and PDEF implementations are now consistent across the Galaxy platform. This capability requires the use of Milkyway reference libraries for the physical libraries.

  For more information about the Milkyway reference libraries, see "Physical Libraries" on page 5-2.

- The following DFT Compiler enhancements are available only in XG mode:

  - Ability to specify internal pins as test pins (`set_dft_drc_configuration -internal_pins`)
  - Ability to specify serially routed sequential cells as a scan group (`set_scan_group -serial_routed`)

Changes

In version X-2005.09, the following changes have been made to XG mode:

- XG mode is now the default mode for Design Compiler, DFT Compiler, Physical Compiler, and Power Compiler
- By default, Physical Compiler uses Milkyway reference libraries, rather than the .pdb physical libraries.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the product release notes in SolvNet.

To see the product release notes,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
3. Click Release Notes in the Main Navigation section (on the left), click the product name you want, then click the release you want in the list that appears at the bottom.

About This Application Note

This application note describes how to invoke the Synopsys synthesis tools in XG mode and the differences between running in XG mode and DB mode.

This application note does not discuss how to run the synthesis tools. For this type of information, see the product documentation.

Audience

This application note is for engineers who plan to run the Synopsys synthesis tools in XG mode.

Related Publications

For additional information about the Synopsys synthesis tools, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* `[`*`pin2 ... pinN`*`]`
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click SolvNet Help in the Support Resources section.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to XG Mode

This chapter provides an overview of the XG mode used by the Synopsys synthesis tools. The information in this chapter applies to all tools that support XG mode. Later chapters discuss aspects of XG mode that apply to specific products.

In version X-2005.09, the Synopsys synthesis tools provide two modes of operation:

- XG mode (default)

  This mode uses optimized memory management techniques that increase a tool’s capacity and can reduce runtime.

- DB mode

  This was the default mode for version W-2004.12 and earlier versions.

This book describes differences in behavior between XG mode and DB mode, as well as features that are available only in XG mode. For detailed information about tool usage, see the product documentation.

This chapter contains the following sections:

Products That Support XG Mode

XG mode is supported in most Synopsys synthesis products, including Design Compiler, Design Compiler FPGA, DFT Compiler (including BSD Compiler, SoCBIST, and Adaptive Scan Technology), Power Compiler, and Physical Compiler.

The ShadowLogic DFT feature of DFT Compiler is not supported in XG mode.

The following synthesis products are not supported in XG mode: Behavioral Compiler, Floorplan Manager, and the SIFF interface to the Mentor Falcon framework.

- For information about using Design Compiler in XG mode, see Chapter 3, “Using Design Compiler in XG Mode.”
- Design Compiler FPGA has no differences between XG mode and DB mode. For information about using Design Compiler FPGA, see *Design Compiler FPGA User Guide*.
- For information about using DFT Compiler (including BSD Compiler and SocBIST) in XG mode, see Chapter 4, “Using DFT Compiler in XG Mode.”
- For information about using Physical Compiler in XG mode, see Chapter 5, “Using Physical Compiler in XG Mode.”
- For information about using Power Compiler in XG mode, see Chapter 6, “Using Power Compiler in XG Mode.”

Supported Platforms

XG mode supports all existing hardware platforms, with the exception of HP32. (The Milkyway product does not support the HP32 platform.)

Licensing Requirements

XG mode does not require any special licensing.

Command Languages

In XG mode, all synthesis tools use the tool command language (Tcl). XG mode does not support the dcsh command language. If your existing scripts are written in dcsh, you must convert them to Tcl before running dc_shell in XG mode.

Synopsys provides the `dc-transcript` utility to help with the conversion to Tcl. The `dc-transcript` utility has the following limitations:

- When your script echoes the results of a dcsh `find` command, `dc-transcript` outputs the collection handle (the variable name) rather than the object names.

  For example, `dc-transcript` translates

  ```
  ports = find(port, "*")
  echo they are: ports
  ```

  as

```
set ports find port "*"
echo [concat they are: $ports
```

To fix this problem change `$ports` to `[get_object_name $ports]`.

- The `dc-transcript` utility does not always correctly handle multiple objects represented as a string.

  For example, `dc-transcript` translates

  ```
  find(net, "x\\?")
  ```

  as

  ```
  find net {x\?}
  ```

  To fix this problem, wrap the string value with the Tcl list command:

  ```
  find net [list {x\?}]
  ```

- The `dc-transcript` utility translates quotation marks to curly braces. In some cases, this will result in an incorrect translation.

  For example, `dc-transript` translates

  ```
  sh "echo s#/pattern## > out"
  ```

  as

  ```
  sh {echo s#/pattern## > out}
  ```

- If your script contains a very complex statement, `dc-transcript` might not be able to translate it.

  For example, `dc-transcript` cannot translate a statement such as

```
sh "awk 'BEGIN{flag=0}{if ($1=="Net" || $1=="Port") flag=1; \
   else if (flag==1 && $1!~/-/) print $1}' naming_temp.rep"
```

Libraries

This section describes the logical and physical libraries used in XG mode.

Logical Libraries

XG mode uses the same logical libraries (the library .db files) as DB mode. No changes are required to the logical libraries. To specify the logical libraries, set the `link_library` and `target_library` variables, just as you do in DB mode.

Physical Libraries

By default, XG mode uses the Milkyway reference library as the physical library. This is the same physical library that is used by the Jupiter and Astro tools. To specify the Milkyway reference library, set the `mw_reference_library` variable.

If you do not have a Milkyway reference library, you can generate one from your LEF library files or your .pdb library files. Use the Milkyway `read_lef` command to convert your LEF files or the Milkyway `read_plib` command to convert your .pdb files. For more information, see "Physical Libraries" on page 5-2.

To revert to using .pdb library files for the physical libraries, set the `use_pdb_lib_format` variable to true. If you use .pdb format physical libraries, you will not have access to the new Milkyway-based DEF and PDEF support.

Setup Variables

Before running a synthesis tool in XG mode, you must define the paths for the libraries and designs that you are using, just as you would in DB mode. Table 1-1 provides a minimum set of setup variables for XG mode. You can set these variables in the setup file (.synopsys_dc.setup), in a script, or interactively.

Table 1-1 XG Mode Setup Variables

Variable	Description
`search_path`	Defines the path used to locate the logical libraries, physical libraries, .db design files, and .ddc design files.
`link_library`	Defines the list of logical libraries searched to resolve references.

Table 1-1 XG Mode Setup Variables (Continued)

Variable	Description
`target_library`	Defines the list of logical libraries used to perform optimization.
`mw_design_library`	Specifies the location of the Milkyway design library. **Important:** In XG mode, you can access only one Milkyway design library in a single shell session.
`mw_reference_library`	Specifies the location of the Milkyway reference library. The order in the list implies priority for reference conflict resolution. If more than one reference library has a cell with the same name, the first reference library has precedence.
`mw_logic1_net`	Specifies the net used to tie off logic 1. You must set this variable if you are performing physical synthesis. If this variable is not set correctly, power nets might be converted to signal nets.
`mw_logic0_net`	Specifies the net used to tie off logic 0. You must set this variable if you are performing physical synthesis. If this variable is not set correctly, power nets might be converted to signal nets.

Supported Commands

The XG mode supports most, but not all, commands in dc_shell and psyn_shell. If you enter a command that is not supported in XG mode, an error message is generated.

For a complete list of commands and options that are not supported in XG mode, see Appendix A, “Command Differences.”

Differences in Behavior

Many commands and concepts are common to all synthesis tools. This section describes behavior differences that might affect any of the synthesis tools. Behavior differences that are product specific are included in the product-specific chapters.

The following sections describe the differences in behavior between XG mode and DB mode:

- read_* Command
- current_design Command
- get_object_name Command
- set_attribute Command
- filter_collection Command
- remove_annotated_delay Command
- Reference Objects
- Collections
- SDC Support
- Timing Path Attributes

read_* Command

In XG mode, when you load a design into memory, the tool also loads all libraries specified in the `link_library` variable, regardless of whether they are needed to link the design.

In DB mode, the libraries are read in during the link process. When all references are resolved, the link process ends and additional libraries, if any, are not read in.

Because the tool reads the libraries while loading the design, rather than during the link process, the memory usage and runtime required for loading the design might increase. However, there is a benefit to this: Unlike in DB mode, where peak memory usage occurs during optimization, in XG mode, peak memory usage occurs while the design is loading. Therefore you know immediately whether your design can be processed with the available memory.

current_design Command

The behavior of the `current_design` command differs in two ways between XG mode and DB mode:

- In runtime
- In collection preservation

The following sections describe these differences.

Runtime Differences

The runtime for the `current_design` command in XG mode is longer than in DB mode. Because of this difference in runtime, you should avoid writing scripts that use a large number of `current_design` commands, such as in a loop.

For example, the following script uses the `current_design` command in a loop to set the `fix_multiple_port_nets` attribute on all designs in memory:

```
set designs [get_designs]
foreach_in_collection d $designs {
   echo [get_object_name $d]
   current_design [get_object_name $d]
   set_fix_multiple_port_nets -all
}
```

In XG mode, this type of loop uses a very large amount of runtime. To reduce the runtime, apply the command to a collection of designs, as shown in the following example, rather than changing the current design within a loop:

```
set_fix_multiple_port_nets -all [get_designs]
```

As another example, to ungroup instances within a hierarchy (such as a DesignWare instance), use the following command to set the `ungroup` attribute on the instance instead of changing the current design and ungrouping the subdesign. When you compile the design, the instances with the `ungroup` attribute are automatically ungrouped.

```
set_ungroup instance true
```

To reduce the need to use the `current_design` command, the following commands work throughout the design hierarchy in XG mode (in DB mode they work only on instances in the current design or require the `-all_instances` option):

- Netlist editting commands

  These commands are used for incrementally editing a design that is in memory. Examples are `create_cell`, `create_net`, `connect_net`, `disconnect_net`, `create_port`, `remove_cell`, `remove_net`, `remove_port`, `remove_unconnected_ports`, `create_bus`, `remove_bus`, and `report_bus`. For a list of enhanced commands, see the *Design Compiler User Guide*, Chapter 5.

- The `ungroup`, `group`, and `uniquify` commands

  For detailed information, see Chapters 5 and 8 in the *Design Compiler User Guide*.

- `set_size_only` command

  In addition to accepting instance objects, the `-all_instances` option allows you to set the `size_only` attribute on a leaf cell when its parent design is instantiated multiple times.

For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 4.

- `change_link` command

  In addition to accepting instance objects, the `-all_instances` option allows you to make link changes for a leaf cell when its parent design is instantiated multiple times.

  For more information, see the *Design Compiler User Guide*, Chapter 5.

Collection Preservation Differences

In DB mode, all existing collections are maintained when you change the current design. You can access collections whether they were created in the current design or in another design.

In XG mode, you can access collections only within the current design in which they were created. When you change the current design, collections containing objects from the previous design are deleted. In most cases, Design Compiler automatically re-creates the deleted collections when you change the current design back to that design. However, Design Compiler does not re-create collections that contain the following design objects:

- Clusters
- Scan paths
- Timing paths

For example, the following command sequence works in DB mode but not in XG mode:

```
current_design top
set top_cells [get_cells -hier]
current_design mid
set sum_cells [add_to_collection [get_cells] $top_cells
```

In DB mode, the collection pointed to by the `sum_cells` variable contains the cells from both the top design and the mid design. In XG mode, the `top_cells` variable is undefined when top is not the current design, so the collection pointed to by the `sum_cells` variable contains only the cells from the mid design.

Because the collection variables associated with a design are re-created each time you run `current_design`, having a large number of collection variables can increase the `current_design` runtime. To reduce the runtime impact, use the `unset` command to delete collection variables that you no longer need. For example,

```
dc_shell-xg-t> set pins [get_pins]
# do things with $pins
dc_shell-xg-t> unset pins
```

get_object_name Command

In XG mode, the `get_object_name` command returns the complete path to an object; in DB mode the `get_object_name` command returns only the object name.

The following examples show the different results when you run the `get_object_name` command.

DB Mode Example

```
dc_shell-t> get_object_name [get_lib_pins class/NR4P/C]
C
```

XG Mode Example

```
dc_shell-xg-t> get_object_name [get_lib_pins class/NR4P/C]
class/NR4P/C
```

This difference makes it possible for you to nest the `get_object_name` command within another command when using XG mode. For example,

```
dc_shell-xg-t> get_attribute \
  [get_object_name [get_lib_pins class/NR4P/C]]] \
   pin_direction
```

In addition, in XG mode you can specify multiple objects as the argument to `get_object_name`. In DB mode, you must specify a single object as the argument.

set_attribute Command

In XG mode, the `set_attribute` command enforces the predefined attribute type and generates an error if you try to set an attribute with a value of an incorrect type. In DB mode, the `set_attribute` command does not perform type checking.

Note:

> To determine the predefined type for an attribute, use the `list_attributes -application` command. This command generates a list of all application attributes and their types.

For example, the `max_fanout` attribute has a predefined type of float. Suppose you enter the following command:

```
set_attribute lib/lcell/lpin max_fanout 1 -type integer
```

In XG mode, you get the following error message:

```
Error: data type does not match attribute type
```

In DB mode the tool simply accepts the command.

filter_collection Command

In XG mode, the `filter_collection` command verifies that the attribute specified in the filter expression is valid for the collection's object type and generates an error if you try to filter on an invalid attribute. In DB mode, the `filter_collection` command does not perform attribute checking.

Note:

> To determine the valid attributes for an object type, use the `list_attributes -application -class` *`object_type`* command. This command generates a list of all application attributes that apply to one of the following object types: design, port, cell, clock, pin, net, or lib.

For example, assume you enter the following command to filter a collection of library cells by specifying the `object_class` attribute (which is not a valid library cell attribute):

```
filter_collection [get_lib_cells mylib/inv] \
   "@object_class == cell"
```

In XG mode, you get the following error message:

```
Error: Unknown identifier: 'object_class' (FLT-005)
```

In DB mode the tool accepts the command and returns an empty collection.

remove_annotated_delay Command

In DB mode, you must use the same format to specify the annotated delay to be removed as you initially use to specify the delay. In XG mode, the requirement has been relaxed to allow a more general delay specification in the `remove_annotated_delay` command.

For example, suppose you set the annotated delay by using the following command:

```
shell-t> set_annotated_delay -from A -to B 10
```

In DB mode, you must remove the annotated delay by using the following command:

```
shell-t> remove_annotated_delay -from A -to B
```

In XG mode, you could also remove the annotated delay by using the following more general command:

```
shell-t> remove_annotated_delay -from A
```

Reference Objects

Reference objects are not supported in XG mode. However, in XG mode the `get_references` command provides similar functionality to using reference objects in DB mode.

In DB mode, the `get_references` command returns a collection of references that meet the specified requirements, and you operate on the references. In XG mode, the `get_references` command returns a collection of instances referred to by the specified reference, and you operate on the instances. Although the objects in the collection differ, the effect of operating on the collection is the same.

For example, in DB mode the following command returns a collection containing the reference AN2:

```
shell-t> get_references AN2
{"AN2"}
```

In XG mode, the same command returns a collection containing the instances in the current design that have the reference AN2:

```
shell-xg-t> get_references AN2
{U2 U3 U4}
```

If you use the `get_references` command in XG mode with a wildcard string and want to see the reference names, use the `report_cell` command.

```
shell-xg-t> report_cell [get_references AN*]

Cell                          Reference        Library            Area  Attributes
--------------------------------------------------------------------------------
U2                           AN2              lsi_10k         2.000000
U3                           AN2              lsi_10k         2.000000
U4                           AN2              lsi_10k         2.000000
U8                           AN3              lsi_10k         2.000000
```

Collections

Both DB and XG modes use Tcl collections to represent a group of design objects, but there are slight differences between the two modes. The following aspects of collection handling differ between the DB and XG modes:

- Invalidation of collections

  Whenever the objects in a collection become invalid or are no longer in the current scope, the collections that refer to those objects become invalid.

  Because of the different memory management techniques used in DB and XG modes, commands that invalidate design objects in one mode might not have the same effect in the other mode.

  For example, the `reset_design` command invalidates collections in DB mode but not in XG mode. The `current_design` command (when used to change the current design) invalidates collections in XG mode but not in DB mode. (For details about the effects of the `current_design` command on existing collections, see "Collection Preservation Differences" on page 1-13.)

  In either mode, if you need to use an invalidated collection, you must regenerate the collection.

- Query results

  In DB mode, querying the contents of a collection produces a comma-separated list of design objects (for example, {a, b, c, d...}). This is not valid syntax for a Tcl list, so the result cannot be used directly.

In XG mode, the query results are formatted as a Tcl list (for example, {a b c d ...}), so that you can directly use the results.

If the query returns a collection handle, rather than a printed list, the result is the same in both DB and XG mode.

SDC Support

In XG mode, Design Compiler supports version 1.5 of the Synopsys Design Constraints (SDC) format. In DB mode, Design Compiler supports SDC version 1.4. For details about the SDC versions, see the *Using the Synopsys Design Constraints Format Application Note*.

Timing Path Attributes

In DB mode, you can get detailed information about the data paths in your design by using the `get_timing_paths` command to generate a collection containing the data path, then querying the data path's attributes. (For more information about this capability, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.)

In XG mode, you can also get detailed information about the clock paths associated with the specified data path and information about the clock reconvergence pessimism removal (CRPR). To enable access to this information, you must specify the `-path_type full_clock_expanded` option when you run the `get_timing_paths` command. (The default, `-path_type full`, enables access only to the data path information.)

Table 1-2 shows the attributes supported on clock paths. Table 1-3 on page 1-22 shows the attributes supported on clock path points.

Table 1-2 Attributes Supported on Clock Paths

Attribute	Type	Notes
`arrival`	string	
`capture_clock_paths`	collection	This attribute is supported only in XG mode.
`clock_path`	Boolean	This attribute is supported only in XG mode.
`clock_uncertainty`	float	
`crpr_common_point`	collection	This attribute is supported only in XG mode.
`crpr_value`	float	This attribute is supported only in XG mode.
`endpoint`	string	
`endpoint_clock`	string	This attribute always has the same value as the `startpoint_clock` attribute.
`endpoint_clock_close_edge_type`	string	
`endpoint_clock_latency`	float	
`endpoint_clock_open_edge_type`	string	
`endpoint_recovery_time_value`	float	
`endpoint_setup_time_value`	float	
`full_name`	string	
`hierarchical`	Boolean	This attribute is always true for clock paths.

Table 1-2 Attributes Supported on Clock Paths (Continued)

Attribute	Type	Notes
`launch_clock_paths`	collection	This attribute is supported only in XG mode.
`object_class`	string	This attribute always has a value of `timing_path`.
`path_group`	string	The path group for the clock path is the same as the path group for the corresponding data path.
`path_type`	string	
`startpoint`	string	
`startpoint_clock`	string	
`startpoint_clock_latency`	float	
`startpoint_clock_open_edge_type`	string	
`startpoint_input_delay_value`	float	

Table 1-3 Attributes Supported on Clock Path Points

Attribute	Type	Notes
`arrival`	string	
`object_class`	string	This attribute always has a value of `timing_path`.
`object`	string	
rise_fall	string	

Invoking a Synthesis Tool in XG Mode

You can run the following user interfaces in XG mode:

- dc_shell
- Design Vision
- fpga_shell
- psyn_shell
- Physical Compiler Graphical User Interface (GUI)

Table 1-4 shows the command used to invoke each interface in XG mode and the prompt used in XG mode.

Table 1-4 XG Mode Tool Invocation Commands

User interface	Invocation command	XG mode prompt

dc_shell	`dc_shell-t`
	`dc_shell-xg-t`
	`dc_shell -tcl`
	`dc_shell -tcl -xg`
		dc_shell-xg-t>

Design Vision	`design_vision`
	`design_vision-xg`
	`design_vision -xg`
		design_vision-xg-t
		>

fpga_shell	`fpga_shell-t -xg`	fpga_shell-xg-t>

psyn_shell	`psyn_shell`
	`psyn_shell -xg`
		psyn_shell-xg-t>

Physical Compiler GUI	`psyn_gui`
	`psyn_gui -xg`
		psyn_gui-xg-t>

Determining the Mode

When you invoke a user interface in XG mode, the following message appears at startup:

```
Starting shell in XG mode...
```

In addition, when you are using a Tcl-based command-line interface, you can use the `shell_is_in_xg_mode` command to determine the current mode. If `shell_is_in_xg_mode` returns 1, the shell is running in XG mode. Otherwise the shell is running in DB mode.

2

XG Mode Design Database Formats

In DB mode, the .db format is the design database format used by all synthesis tools. In XG mode, the synthesis tools support two design database formats: .ddc and Milkyway. This chapter describes these formats in the following sections:

- Using the .ddc Format
- Using the Milkyway Format
- Using the .db Format
- Interfacing Between Synopsys Tools

Table 2-1 shows the support that each synthesis tool provides for the various database formats. The DB Milkyway format refers to the Milkyway design library written by the `write_mdb` command in DB mode. The XG Milkyway format refers to the Milkyway design library written by the `write_milkyway` command in XG mode.

Table 2-1 Supported Database Formats by Tool

Tool	.db		.ddc		DB Milkyway		XG Milkyway	
	Read	Write	Read	Write	Read	Write	Read	Write
dc_shell - DB mode (Design Compiler, DFT Compiler, Power Compiler)	X	X			X	X		
dc_shell - XG mode (Design Compiler, DFT Compiler, Power Compiler)	X[1]	X[1]	X	X	X		X[2]	X[3]
psyn_shell - DB mode (Physical Compiler, DFT Compiler, Power Compiler)	X	X			X	X		
psyn_shell - XG mode (Physical Compiler, DFT Compiler, Power Compiler)	X[1]		X	X[4]	X[5]		X	X[3]
Formality	X	X	X		X		X	
PrimeTime	X	X	X		X			
PrimePower	X	X	X		X			
Jupiter					X	X	X	X[6]
Astro					X	X	X	X[6]

1. *Not recommended. Use .ddc or Milkyway format instead for maximum efficiency.*
2. *Not recommended if your design contains physical information.*
3. *Requires a mapped design that does not contain multiple instances.*
4. *Saves logical information only.*
5. *Design constraints are not read. You must use the SDC file to reapply the constraints.*
6. *Existing design constraints are maintained but not updated.*

Using the .ddc Format

The .ddc format is similar to the .db format in that it is a single-file, binary format. The .ddc format stores design data in a more efficient manner than the .db format, enabling increased capacity. In addition, reading and writing files in .ddc format is faster than reading and writing files in .db format. The .ddc format stores only logical design information.

Note:

To maximize the capacity and performance improvements, use the .ddc format rather than the .db format when using XG mode.

The following sections contain information about how to use the .ddc format:

- Writing .ddc Files
- Limitations When Writing .ddc Files
- Reading .ddc Files
- Converting From .db Format to .ddc Format
- GUI Support

Writing .ddc Files

To save the design data in a .ddc file, use the `write -format ddc` command.

By default, the `write` command saves just the top-level design. To save the entire design, specify the `-hier` option.

If you do not use the `-output` option to specify the output file name, the `write -format ddc` command creates a file called *top_design*.ddc, where *top_design* is the name of the current design.

Limitations When Writing .ddc Files

The following limitations apply when you are writing .ddc files:

- The .ddc format saves only the logical design information. The .ddc format does not save physical information, such as physical constraints, floorplan data, or cell locations.
- If you load your design from a Milkyway database and then save the design in .ddc format, the .ddc file contains only the gate-level netlist description. You cannot perform RTL synthesis on this .ddc file.

Reading .ddc Files

To read the design data from a .ddc file, use the `read_ddc` command.

Note:

Like the .db format, the .ddc format is backward compatible (you can read a .ddc file that was generated with an earlier software version), but not forward compatible (you cannot read a .ddc file that was generated with a later software version).

Converting From .db Format to .ddc Format

To convert your design data from .db format to .ddc format, read the .db file into dc_shell or psyn_shell in XG mode, then save the design in .ddc format (`write -format ddc`). To realize the memory savings from using the .ddc format, you must exit the current shell, then restart the shell in XG mode and read the .ddc file.

GUI Support

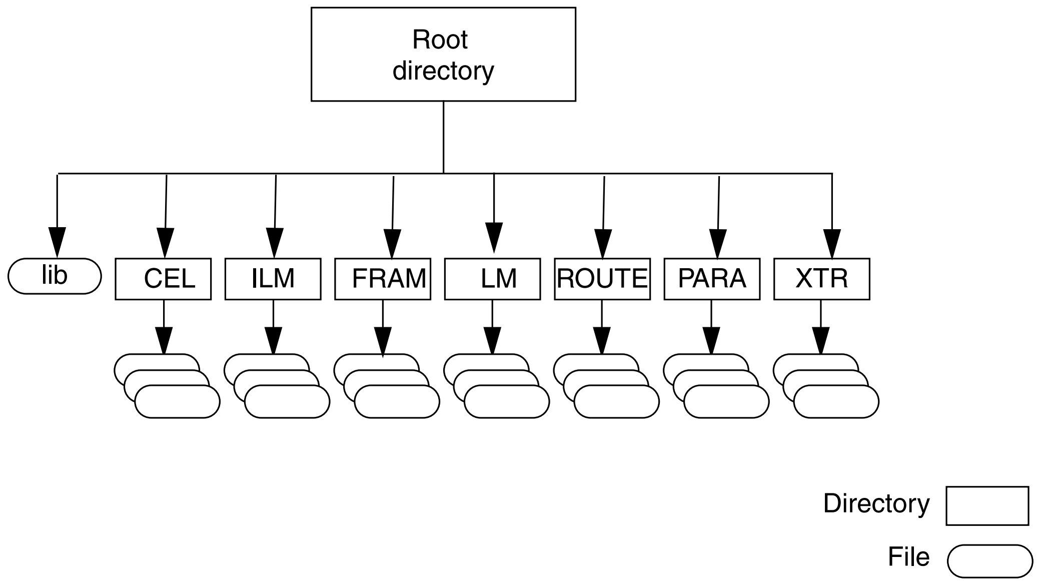
In XG mode, both Design Vision and the Physical Compiler GUI support the .ddc format.

To read or write .ddc files, select either Auto (if your file has the .ddc extension) or DDC from the Format list in the appropriate dialog box.

Using the Milkyway Format

The Milkyway format uses the directory structure shown in Figure 2-1 to store design data. This directory structure is referred to as the Milkyway design library. You specify the Milkyway design library for the current session by setting the `mw_design_library` variable to the root directory path.

Figure 2-1 Milkyway Design Library



The Milkyway format stores both logical and physical design information, but it requires a mapped design (the Milkyway format cannot store RTL information). In addition, the Milkyway format does not support designs that contain multiple instances.

The Milkyway format stores the following design information:

- Logical information, such as the netlist, design constraints, and design attributes
- Floorplan data, such as floorplan objects and physical constraints (for example, keepouts and bounds)
- Placement data

The format used to represent constraint information differs between the Milkyway format used in XG mode differs and the Milkyway format used in DB mode (as generated by the `write_mdb` command or by the Jupiter or Astro tool).

If you read a Milkyway design library that was created in the other mode, the design information is read, but the constraint information is not. You must reapply the constraints by reading the Synopsys Design Constraints (SDC) file.

Note:

> Jupiter and Astro can read the constraint information from the Milkyway design library regardless of which mode it was generated in.

The following sections contain information about how to use the XG mode Milkyway format:

- Creating a Milkyway Design Library
- Saving a Design in Milkyway Format
- Limitations When Writing Milkyway Format
- Reading a Design in Milkyway Format
- Limitations When Reading Milkyway Format
- Maintaining the Milkyway Design Library
- Converting From .db Format to Milkyway Format
- GUI Support

Creating a Milkyway Design Library

You must create a Milkyway design library before you can save your design in Milkyway format.

To create a Milkyway design library,

1. Specify the logical libraries by setting the `search_path`, `target_path`, and `link_library` variables.
2. Specify the physical libraries by setting the `mw_reference_library` variable.
3. Specify the location for the Milkyway design library by setting the `mw_design_library` variable.
4. Define the power nets by setting the `mw_logic0_net` and `mw_logic1_net` variables.

5. Run the `create_mw_design` command to create the Milkyway design library.

   When you run the `create_mw_design` command, you must use the `-tech_file` option to specify the Milkyway technology file. If you do not have a Milkyway technology file, see "Generating a Milkyway Technology File" on page 5-5 for information about extracting it from the Milkyway reference library.

   For detailed information about the `create_mw_design` command, see the *Physical Compiler User Guide, Volume 1*.

For an example script that shows how to create a Milkyway design library, see Example 2-1 on page 2-10.

Saving a Design in Milkyway Format

To save the design data in the Milkyway design library,

1. Specify the Milkyway reference library (`mw_reference_library` variable) and the Milkyway design library (`mw_design_library` variable).
2. Define the power nets by setting the `mw_logic0_net` and `mw_logic1_net` variables.

   If these variables are not set correctly, the power nets might be converted to signal nets.

3. Prepare the Milkyway design library.

   - If the Milkyway design library does not exist, create it as described in "Creating a Milkyway Design Library" on page 2-8.

- If you are writing to an existing Milkyway design library, prepare it for access by running the `set_mw_design` command.

4. Run the `write_milkyway` command to save the design data. You must use the `-output` option to specify the file name.

   ```
   psyn_shell-xg-t> write_milkyway -output file_name
   ```

Example 2-1 shows a sample command sequence for saving a design in Milkyway format.

Example 2-1 Saving a Design in Milkyway Format

```
# Define logical libraries
set search_path "dir1 dir2"
set target_library "logic_lib.db"
set link_library "* logic_lib.db"

# Define Milkyway reference library
set mw_reference_library mw_ref_lib

# Define Milkyway design library
set mw_design_library design_dir

# Define power nets
set mw_logic1_net VDD
set mw_logic0_net VSS

# Create Milkyway design library
create_mw_design -tech_file mw_ref.tf

# Save design data in Milkyway format
write_milkyway -output file_name
```

The `write_milkyway` command saves the design data for the current design in the CEL view of the Milkyway design library. The path for the design file is *design_dir*/CEL/*file_name*:*version*, where *design_dir* is the location you specified in `mw_design_library`.

By default, the `write_milkyway` command increments the version number of the Milkyway design file. To overwrite the latest version of the Milkyway design file instead of creating a new one, specify the `-overwrite` option.

If you plan to read the Milkyway design file created by `write_milkyway` into Jupiter or Astro, see "Interfacing Between Synopsys Tools" on page 2-26 for the requirements.

If you encounter problems when saving the design in Milkyway format, check for the following possible causes:

- The Milkyway design library does not exist

  It is not sufficient for the directory specified in the `mw_design_library` variable to exist. The specified directory must contain a Milkyway design library before you can save your design in Milkyway format. For information about creating a Milkyway design library, see "Creating a Milkyway Design Library" on page 2-8.

- The Milkyway design library is locked.

  If the Milkyway design library is being used by another user, you must wait to use the Milkyway design library. If there is a leftover lock file in the CEL directory, delete it as described in "Reading a Design in Milkyway Format" on page 2-12.

- The design is not mapped

  Because the Milkyway format describes physical information, it supports mapped designs only. You cannot use the Milkyway format to store design data for unmapped designs.

- The design contains multiples instances

  You must flatten or uniquify your design before saving it in Milkyway format.

Limitations When Writing Milkyway Format

The following limitations apply when you are writing your design in Milkyway format:

- The `write_milkyway` command saves the entire hierarchical design in a single Milkyway design file (CEL view). You cannot generate separate design files for each subdesign.
- When you save a design in Milkyway format, the `write_milkyway` command does not save the interface logic model (ILM) instances in the Milkyway design library. You must explicitly save each ILM (see "Using Interface Logic Models" on page 5-12).
- The constraints saved in the Milkyway design library cannot be read in DB mode (the `read_mdb` command) or by versions of Jupiter or Astro prior to W-2004.12.

  If you are using one of these tools, you must input the constraints by reading the golden SDC file for the design after reading the Milkyway design library. For more information, see "Limitations When Reading Milkyway Format" on page 2-17.

Reading a Design in Milkyway Format

To read a Milkyway design file,

1. Specify the logical libraries by setting the `search_path`, `target_library`, and `link_library` variables.

2. Specify the Milkyway reference library (`mw_reference_library` variable) and the Milkyway design library (`mw_design_library` variable).

3. Define the power nets by setting the `mw_logic0_net` and `mw_logic1_net` variables.

   If these variables are not set correctly, the power nets might be converted to signal nets.

4. Set up the Milkyway design library and update the search path by running the `set_mw_design` command.

5. Run the `read_milkyway` command to read the design.

   ```
   psyn_shell-xg-t> read_milkyway file_name
   ```

   Note:

   The `read_mdb` command is not supported in XG mode. For information about reading Milkyway design libraries created by `write_mdb`, see "Limitations When Reading Milkyway Format" on page 2-17.

Example 2-2 shows a sample command sequence for reading a design in Milkyway format.

Example 2-2 Reading a Design in Milkyway Format

```
# Define logical libraries
set search_path "dir1 dir2"
set target_library "logic_lib.db"
set link_library "* logic_lib.db"

# Define Milkyway reference library
set mw_reference_library mw_ref_lib

# Define Milkyway design library
set mw_design_library design_dir

# Define power nets
set mw_logic1_net VDD
set mw_logic0_net VSS

# Prepare Milkyway design library
set_mw_design

# Read design data in Milkyway format
read_milkyway file_name
```

When you read a design file from the Milkyway design library, the tool sets the top-level design as the current design and links the design.

By default, the `read_milkyway` command reads the latest version of the *file_name* design file from the location specified in `mw_design_library`. To read another version of the design file, specify the version with the `-version` option.

By default, the `read_milkyway` command generates a lock file to prevent other users from accessing the Milkyway design library. To open a Milkyway design library without generating a lock file, specify the `-read_only` option. This option prevents you from modifying the Milkyway design library, but allows other users to access the design.

When you end your session, the lock file is automatically deleted. However, if your session terminates abnormally or you end the session by using Control-c, the lock file is not deleted and you must manually remove it. For example,

```
% rm design_dir/CEL/my_design*.lock
```

If you encounter problems when reading a Milkyway design library, check for the following possible causes:

- The Milkyway design library is locked.

  If the Milkyway design library is being used by another user, you must wait to use the Milkyway design library. If there is a leftover lock file in the CEL directory, delete it as previously described.

- The physical library cannot be located.

  Verify that the physical library is defined and that its location is accurately defined in the search path.

- There is a mismatch between the site names in the Milkyway design library and the physical library.

  To synchronize the site names between the Milkyway design library and the physical library, use the `mw_site_name_mapping` variable to define the name mappings. You must set this variable before running the following commands: `read_db`, `read_ddc`, or `read_def`.

The syntax for setting this variable is

```
set mw_site_name_mapping \
   [list old_site_name new_site_name]
```

- The logical hierarchy data in the Milkyway design library is invalid.

  If the hierarchy data has been corrupted, you must use Astro to repair the Milkyway design library. You can use the following Milkyway consistency checking commands to validate your Milkyway design library: `dbCheckCellData`, `dbCheckNetlistVsFram`, and `astCheckHierPresConsistency`. For more information about these commands, see the Milkyway documentation.

Limitations When Reading Milkyway Format

If a Milkyway design file was not created in XG mode (it was created by `write_mdb` in DB mode or by the Jupiter or Astro tool), the design constraints are not loaded into memory when you use the `read_milkyway` command to read the Milkyway design.

Note:

This limitation also applies to a Milkyway design file that was updated in Jupiter or Astro, even if the design library was initially created in XG mode.

To ensure that all design constraints are loaded into memory, use one of the following methods to read a Milkyway design file that was not created in XG mode.

Method 1: Using the Milkyway and SDC Files

In XG mode, use `read_milkyway` to read the DB mode Milkyway design file, then reapply the design constraints by reading the golden SDC file:

```
psyn_shell-t> read_milkyway my_design
psyn_shell-t> source my_constraints.sdc
```

Method 2: Using the .db and DEF Files

To use this method, you must first convert the Milkyway design file into .db and Design Exchange Format (DEF) files in DB mode, then read the .db and DEF files in XG mode.

1. Invoke the tool in DB mode.

   In DB mode, use `read_mdb` to read the DB mode Milkyway design file, then save the design in .db and DEF formats.

   ```
   psyn_shell-t> read_mdb -cell_name my_design
   psyn_shell-t> current_design my_design
   psyn_shell> write -format db -hierarchy \
      -output my_design.db
   psyn_shell-t> write_def -output my_design.def
   ```

2. Invoke the tool in XG mode.

   In XG mode, read the .db and DEF files.

   ```
   psyn_shell-t-xg> read_db my_design.db
   psyn_shell-t-xg> link_physical
   psyn_shell-t-xg> read_def my_design.def
   ```

Maintaining the Milkyway Design Library

To maintain your Milkyway design library (for example, to delete unneeded versions of your design), you must use the Milkyway Environment tool. The Milkyway Environment tool is a graphical user interface (GUI) that enables manipulation of the Milkyway libraries.

This section describes how to perform the following tasks:

- Invoke the Milkyway Environment tool
- Open a Milkyway design library
- List the cells in the Milkyway design library
- Purge unneeded versions of cells from the Milkyway design library
- Delete cells from the Milkyway design library

For more information about using the Milkyway Environment tool, see the Milkyway documentation. For information about installing the Milkyway Environment tool, see the *Installation Guide*.

Invoking the Milkyway Environment Tool

To invoke the Milkyway Environment tool, enter

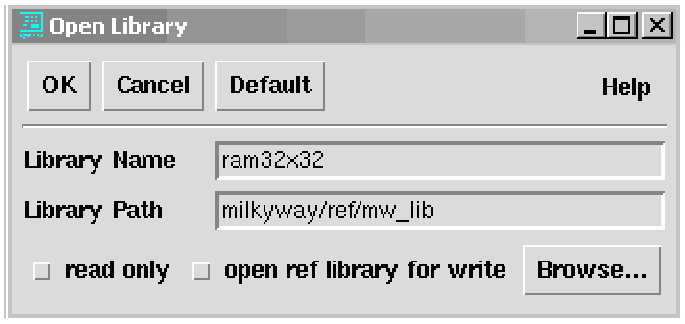
```
% Milkyway -galaxy
```

Opening a Milkyway Design Library

To open an existing Milkyway design library,

1. Choose Library > Open from the Milkyway menu bar.

The Open Library dialog box appears.



Library Name: ram32x32
Library Path: milkyway/ref/mw_lib
read only
open ref library for write

2. Enter the design library name (Library Name) and path (Library Path).

   Alternatively, you can click Browse to use the browse capability to select the design library. When you use the browse capability, Milkyway determines both the design library name and its path from your selection.

3. Click OK to open the specified design library.

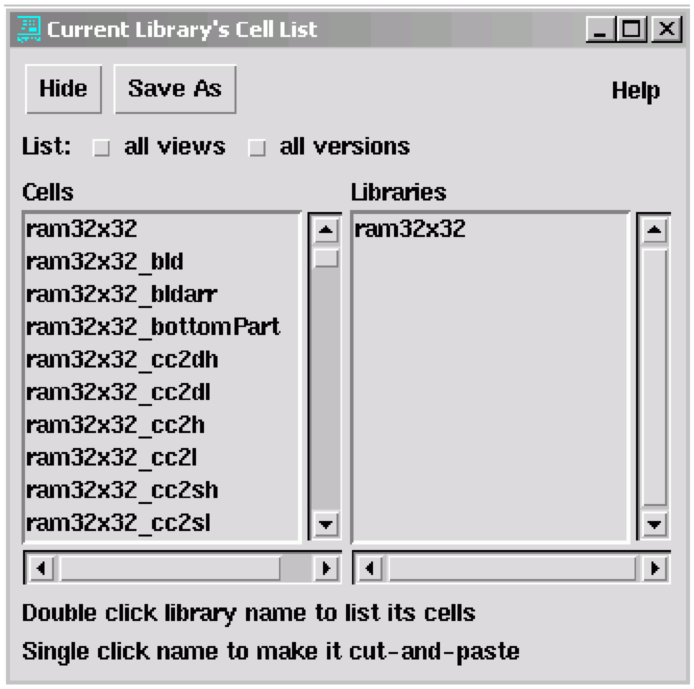
Note:
  You can have only one design library open at a time.

Listing the Cells

To list the cells in the current Milkyway design library,

- Choose Library > Show Cells from the Milkyway menu bar.

The Current Library's Cell List dialog box appears. By default, the view and version information is not displayed. To display all views of the cells, select "all views". To display all versions of the cells, select "all versions".



ram32x32
ram32x32_bld
ram32x32_bldarr
ram32x32_bottomPart
ram32x32_cc2dh
ram32x32_cc2dl
ram32x32_cc2h
ram32x32_cc2l
ram32x32_cc2sh
ram32x32_cc2sl

ram32x32

Double click library name to list its cells

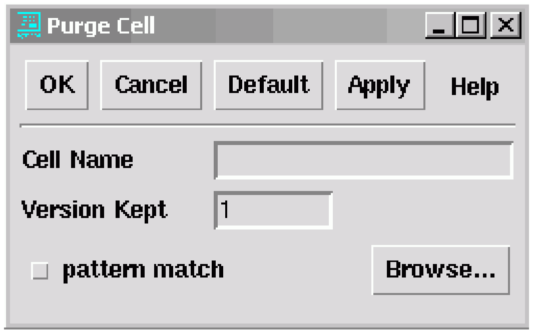
Single click name to make it cut-and-paste

Purging Versions from the CEL View

To purge old versions of cells in the current Milkyway design library,

1. Choose Cell > Purge from the Milkyway menu bar.

The Purge Cell dialog box appears.



2. Specify the name of the cell you want to purge (Cell Name).

   Alternatively, you can click Browse to use the browse capability to select the cell.

3. Specify the number of versions to keep (Version Kept) field.

   For example, if you specify 1, after purging the latest version of the cell remains. All older versions are deleted.

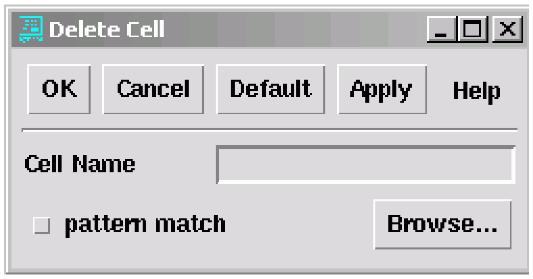
4. Click OK to delete the specified versions of the cell from the Milkyway design library.

Deleting a Cell

To delete a cell from the current Milkyway design library,

1. Choose Cell > Delete from the Milkyway menu bar.

   The Delete Cell dialog box appears.



2. Specify the name of the cell you want to delete (Cell Name).

   Alternatively, you can click Browse to use the browse capability to select the cell.

3. Click OK to delete all versions of the specified cell from the Milkyway design library.

Converting From .db Format to Milkyway Format

To convert your design data from .db format to Milkyway format, read the .db file into dc_shell or psyn_shell in XG mode, then save the design in Milkyway format (`write_milkyway`). For example,

```
psyn_shell-t-xg> set mw_cel_without_Fram_tech true
psyn_shell-t-xg> set mw_design_library design_dir

psyn_shell-t-xg> read_db my_design.db
psyn_shell-t-xg> write_milkyway -output my_design

psyn_shell-t-xg> remove_design -all
psyn_shell-t-xg> read_milkyway my_design
```

GUI Support

In XG mode, the Physical Compiler GUI supports the Milkyway format with menu options, but Design Vision does not.

To read a Milkyway design in the Physical Compiler GUI, choose File > Read. When the Read Designs dialog box opens, Milkyway is set as the design type. To read a design, specify the Milkyway design library in the “Library name” box, select the appropriate design (cell) and version, then click Read.

To write a Milkyway design in the Physical Compiler GUI, choose File > Save As. When the Write Designs dialog box opens, Milkyway is set as the design type. To write a design, specify the Milkyway design library in the “Library name” box, select the appropriate design (in the “Cell name” box), then click OK.

To read or write a Milkyway design library in Design Vision, you must run `read_milkyway` or `write_milkyway` from the command line.

Using the .db Format

The .db format is the internal database format used in DB mode. Although this format is supported in both DB and XG mode, for maximum capacity in XG mode, do not use the .db format in XG mode.

For more information about the .db format, see the Design Compiler documentation.

Writing .db Format

To save a design in .db format, you must use the `-xg_force_db` option.

```
psyn_shell-t-xg> write -format db -xg_force_db -hierarchy \
   -output my_design.db
```

If your design contains physical information or if you do not specify the `-xg_force_db` option, the `write -format db` command generates an error message.

Limitations When Writing .db Format

Because Physical Compiler does not annotate physical data to the .db file, you cannot write .db designs from psyn_shell.

Note:

In the Physical Compiler GUI, the Design type list in the Write Designs dialog box (File > Save As) lists DB as a valid option. This option is not valid in XG mode and should not be used.

Limitations When Reading .db Format

When you read a routed design in .db format, Physical Compiler loads the net topology and any vias defined in the physical library; however, Physical Compiler does not load any design-specific via definitions or power net topology.

To ensure that all routing information is loaded into memory, use the following process to read your .db design:

1. Invoke Physical Compiler in DB mode.
   a. Use the `read_db` command to read the design .db file.
   b. Use the `write_def` command to save the floorplan and routing information.
2. Invoke Physical Compiler in XG mode.
   a. Use the `read_db` command to read the design .db file.
   b. Use the `link_physical` command to link the physical library to your design.
   c. Use the `read_def` command to read the generated DEF file to input the floorplanning and routing information.

Interfacing Between Synopsys Tools

Table 2-2 shows the recommended format for exchanging design data between the various Synopsys tools. In general, .ddc is the recommended format for logical information, and Milkyway is the recommended format when the design contains physical information.

Table 2-2 Recommended Design Interface Formats

From	to dc_shell	to psyn_shell	to Jupiter/Astro
dc_shell	.ddc format	.ddc format	Milkyway format
psyn_shell	.ddc format	Milkyway format	Milkyway format
Jupiter or Astro	Milkyway format	Milkyway format	Milkyway format

Exporting Design Data to Jupiter or Astro

If you are exporting design data from dc_shell or psyn_shell to Jupiter or Astro, you must follow these steps:

1. Remove the multiport nets.

   To remove the multiport nets, run the `set_fix_multiple_port_nets -all` command before you run `compile` or `physopt`.

2. Ensure that the netlist is Verilog compliant.

   To ensure that the netlist is Verilog compliant, run the `change_names -rules verilog -hier` command before you run `write_milkyway`.

Example 2-3 shows a sample command sequence for exporting a design to Jupiter or Astro.

Example 2-3 Exporting a Design to Jupiter or Astro

```
# Remove the multi-port nets
set_fix_multiple_nets -all
physopt

# Ensure the netlist is Verilog compliant
change_names -rules verilog -hier

# Save the design in Milkyway format
write_milkyway -output my_design
```

Verifying Designs Compiled in XG Mode

There are no differences between the optimizations performed in XG mode and DB mode. Therefore, Formality behaves the same with designs compiled in either mode.

3

Using Design Compiler in XG Mode

This chapter describes the differences between running Design Compiler in XG mode and DB mode.

This chapter contains the following sections:

Differences in Behavior

The following differences in Design Compiler behavior exist between XG mode and DB mode:

- Effect of the OPT-100 error

  In XG mode, the OPT-100 error (command *xyz* terminated abnormally) might corrupt the design database. To prevent a script from performing further optimization on this data, dc_shell removes all designs from memory. If this error occurs, exit and restart the shell. In DB mode, you need not exit when this error occurs.

- Behavior of the `check_design` command

  In XG mode, the `check_design` command generates warnings for the following cases (in addition to the checks performed in DB mode):

  - Constant-driven outputs in the design—that is, an output that is driven by a logic constant cell or the `check_design` command is called post-compile on an originally unused output
  - A multidriver net connecting VDD directly to VSS
  - A multidriver net with constant drivers
  - Designs with no child cells or nets

  Additionally, the `check_design` command has a new option `-multiple_designs` that you can use to display multiply instantiated designs. By default, warning messages related to such designs are not reported.

- Behavior of the `uniquify` command

  In XG mode, the `uniquify` command removes the original design from memory after it creates the new, unique designs. The original design and any collections that contain it or its objects are no longer accessible.

  In DB mode, the original design remains in memory after you run the `uniquify` command.

- Behavior of `compile_ultra` command

  In XG mode, the `compile_ultra` command automatically ungroups small hierarchies to improve the quality-of-results for both timing and area. You can disable this feature by specifying the `-no_autoungroup` option when you run the `compile_ultra` command. For more information, see Chapter 8 of the *Design Compiler User Guide*.

- Constraints set on subdesigns

  In XG mode, if you are using a bottom-up compile flow, the constraints set on subdesigns are not preserved after you perform a top-level compile. To ensure that you are using the correct constraints, you should always reapply the subdesign constraints before compiling or analyzing a subdesign.

  Note:

  This behavior difference does not impact the top-level compile. The top-level constraints are always preserved.

The following difference in Design Vision behavior exist between XG mode and DB mode:

- Impact of changing the design netlist

  In XG mode, when you change the design netlist (for example, by using netlist editing commands, such as `change_link`) when the design schematic is open, Design Vision updates the schematic and maintains the current zoom level and pan position.

  In DB mode, Design Vision closes the design schematic.

  Note:

  > If you have a path schematic open when you change the netlist, Design Vision closes the path schematic in both XG mode and DB mode.

Using Automated Chip Synthesis in XG Mode

In XG mode, Automated Chip Synthesis uses the .ddc format to store the design files, rather than the .db format.

The default directories for the design files do not change when you use .ddc format instead of .db format; however, Automated Chip Synthesis provides unique file types for the .ddc files, so you can change the directories if you want to. Table 3-1 shows the .ddc file types. For more information about file types and customizing the directory locations, see the *Automated Chip Synthesis User Guide*.

Table 3-1 .ddc File Types

File type	Keyword	Default file location
Elaborated .ddc file	elab_ddc	$acs_work_dir/elab/db
Precompile .ddc file	pre_ddc	$acs_work_dir/*dest_dir*/db/pre_compile[1]
Postcompile .ddc file	post_ddc	$acs_work_dir/*dest_dir*/db/post_compile[1]

1. The dest_dir argument refers to the destination directory.

In XG mode, constraints are not preserved on subdesigns after performing a top-level compile; therefore, the postcompile .ddc files for the subpartitions do not contain constraints. If you reload a postcompile .ddc file for a subpartition, you must reapply the subpartition constraints (from the $acs_work_dir/*dest_dir*/constraints directory) before doing any analysis or further processing on the subpartition.

Unsupported Capabilities

In XG mode, Design Compiler does not support the following capabilities:

- Saved design budgets

  You cannot save design budgets in the design .db file. If a .db file contains design budgeting information, dc_shell ignores the information.

- Design verification

  The `compare_design`, `set_compare_design_script`, `reset_compare_design_script`, and `write_compare_design_script` commands are not supported in XG mode. In addition, XG mode does not support the `-verify`, `-verify_effort`, and `-verify_hierarchically` options with the `compile` or `translate` commands.

  To verify your design, use the Formality tool, rather than these dc_shell commands.

- Pad mapping

  The following commands are not supported in XG mode: `insert_pads`, `remove_pads`, `set_pad_type`, and `set_port_is_pad`.

4

Using DFT Compiler in XG Mode

This chapter describes the DFT Compiler flow using the new user interface introduced in XG mode. In XG mode, DFT Compiler uses the Unified Test Design Rule Checking (DRC) flow (which was introduced in version U-2003.06), so the scope of changes you see depends on whether you have previously converted to the Unified Test DRC flow. A converter script, `db2xg`, is provided to assist with the conversion.

This chapter contains the following sections:

Benefits of XG Mode

XG mode provides the following benefits for DFT Compiler:

- Half the runtime of DB mode
- Twice the capacity of DB mode
- New user interface that is more consistent and flexible than the DB mode user interface
- Enhanced reporting capabilities

Features Available in XG Mode

The following DFT Compiler features are available in XG mode:

- Unified Test DRC flow
- 1-Pass test synthesis
- Rapid scan synthesis
- Hierarchical scan synthesis
- Adaptive Scan Technology (including Hierarchical Adaptive Scan Synthesis)
- Test data volume reduction (TDVR)
- Automatic fixing of scan violations (AutoFix)
- Location-based scan ordering
- Timing-based scan ordering
- BSD Compiler
- SocBIST (including BIST-ready and Core Wrapping)

The ShadowLogic DFT feature is not available in XG mode.

The following capabilities are available only in XG mode:

- Ability to use internal pins as test pins (`set_dft_drc_configuration -internal_pins` command)
- Ability to specify a serially routed set of sequential cells as a scan group (`set_scan_group -serial_routed` command)

Overview of Scan Synthesis Command Changes

The main differences between the user interface in XG mode and in the Unified Test DRC flow in DB mode are the following:

- The command set used to identify the test-related ports and to set the test attributes on these ports

  DB mode provides four commands for this purpose. XG mode provides a single command, `set_dft_signal`.

- The introduction of the concept of descriptive and prescriptive views of the design-for-test (DFT) structures

  The descriptive view (`-view existing_dft`) describes how existing logic is used in test mode. The prescriptive view (`-view spec`) defines the DFT structures that you want DFT Compiler to insert into the design. Views are used for specifying the DFT signals (`*dft_signal` commands) and the scan paths (`*scan_path` commands).

  Note:

  If you do not specify a view, DFT Compiler uses the prescriptive view (`-view spec`).

- Addition of commands to report on *all* DFT specifications

  The ability to report on the DFT specifications enables you to verify the current specification at any point during exploration and implementation.

- Addition of commands to reset or remove *all* DFT specifications

  The ability to return the DFT specification to the default state enables interactive debugging and exploration of the DFT structures.

If you have not yet migrated to the Unified Test DRC flow, you will also see the following differences in test design rule checking:

- Test design rule checking requires a test protocol file.

  In the Unified Test DRC flow, the test protocol is no longer inferred during test design rule checking. You must explicitly run `create_test_protocol` (or `read_test_protocol`) before running test design rule checking.

- The same command, `dft_drc`, is used to perform test design rule checking in all contexts

  In the original DRC flow, the `rtldrc` command is used to perform test design rule checking for RTL designs, while the `check_dft` command is used to perform test design rule checking for gate-level designs. In the Unified Test DRC flow, the `dft_drc` is used to perform test design rule checking, regardless of the context (RTL design, gate-level pre-scan design, or gate-level post-scan design).

In addition to these changes, many DFT Compiler commands have been obsolete for some time but are still used. You must remove these commands from your scripts, because they do not function in XG mode. If you use these commands in XG mode, you will get an error message such as

```
dc_shell-xg-t> set_test_methodology
Error: Command 'set_test_methodology' is disabled.(CMD-080)
```

Table 4-1 shows the command correspondence between the different DFT Compiler user interfaces. In most cases, even if the command name is the same, the command options differ between XG mode and DB mode. For detailed information about these commands, see the man pages. For information about syntax differences between XG mode and DB mode, see Appendix A, “Command Differences.” For information about using the `db2xg` converter script to automatically update your script files, see “Using the db2xg Converter Script” on page 4-12.

Table 4-2 provides a list of obsolete DFT Compiler commands that no longer function in XG mode.

Table 4-1 DFT Compiler Command Correspondence

XG mode	DB mode (Unified Test DRC)	DB mode (Original DRC)
Scan Specification (Descriptive)		

`set_dft_configuration`
`-scan`
`-fix_clock`
`-fix_set -fix_reset`
	`set_dft_configuration`
	(not available)
	`-autofix`
	`-autofix`
		`set_dft_configuration`[1]

`reset_dft_configuration`	`remove_dft_configuration`	`remove_dft_configuration`
`report_dft_configuration`	`report_test` `-dft_configuration`	
`set_scan_configuration`[1]	`set_scan_configuration`[1]	`set_scan_configuration`

`set_scan_element false`	`set_scan_configuration`
	`-replace false`
		`set_scan_configuration`
		`-replace false`

`reset_scan_configuration`	`remove_scan_specification`	`remove_scan_specificatio` `n`
`report_scan_configuration`	(not available)	(not available)
`set_scan_state`	`set_scan_state`	`set_scan_state`

`set_dft_signal`
`-view existing_dft`
`-type ScanClock`
	`create_test_clock`	`create_test_clock`

Table 4-1 DFT Compiler Command Correspondence (Continued)

XG mode	DB mode (Unified Test DRC)	DB mode (Original DRC)
`set_dft_signal -view existing_dft -type Reset`	`set_signal_type test_asynch[_inverted]`	`set_signal_type test_asynch[_inverted ]`
`set_dft_signal -view existing_dft -type Constant`	`set_test_hold`	`set_test_hold`
`remove_dft_signal`	(not available)	(not available)
`report_dft_signal -view existing_dft`	`report_test -port`	`report_test -port`
`set_test_assume`[2]	`set_test_assume`	`set_test_assume`
`set_scan_path -view existing_dft`	(not available)	(not available)
`remove_scan_path`	(not available)	(not available)

Test Design Rule Checking

`create_test_protocol`	`create_test_protocol`	`create_test_protocol`
`read_test_protocol`	`read_test_protocol`	`read_test_protocol`
`remove_test_protocol`	(not available)	(not available)
`write_test_protocol`	`write_test_protocol`	`write_test_protocol`
`dft_drc`	`dft_drc`	`rtldrc` `check_dft` `check_scan` `check_test`

Table 4-1 DFT Compiler Command Correspondence (Continued)

XG mode	DB mode (Unified Test DRC)	DB mode (Original DRC)
Scan Specification (Prescriptive)		
`set_dft_signal -view spec`	`set_scan_signal`	`set_scan_signal`
`set_scan_bidi`	`set_scan_bidi`	`set_scan_bidi`
`set_scan_path`	`set_scan_path` `set_scan_segment`	`set_scan_path` `set_scan_segment`
`set_scan_replacement`	`set_scan_replacement`	`set_scan_replacement`
`set_scan_tristate`	`set_scan_tristate`	`set_scan_tristate`
`remove_scan_replacement`	`set_scan_replacement -remove`	`set_scan_replacement -remove`
Scan Preview		
`preview_dft`[1]	`preview_dft`	`preview_scan`
Scan Insertion		
`insert_dft`[1]	`insert_dft`[1]	`insert_scan` `insert_test`
`set_dft_insertion_configuration`[3] `-synthesis_optimization \ none` `-preserve_design_name`	`set_dft_optimization_configuration`[3] `-none` `-preserve_design_name`	`set_dft_optimization_configuration`[2] `-none` `-preserve_design_name`
`reset_dft_insertion_configuration`[3]	(not available)	

Table 4-1 DFT Compiler Command Correspondence (Continued)

XG mode	DB mode (Unified Test DRC)	DB mode (Original DRC)
Hierarchical Scan Synthesis		
`use_test_models`	(not available)	(not available)
AutoFix		
`set_autofix_ configuration`	(not available)	(not available)
`reset_autofix_ configuration`[3]	(not available)	(not available)
`report_autofix_ configuration`[3]	(not available)	(not available)
`set_autofix_element`	(not available)	(not available)
`reset_autofix_element`	(not available)	(not available)
`report_autofix_element`	(not available)	(not available)
Reporting		
`report_scan_path`	`report_test`	`report_test`

1. Options differ between XG mode and DB mode.

2. The set_test_assume command can be used only on output pins in XG mode. In addition, it no longer requires you to isolate the output pin (set_test_isolate is not required).

3. Command is one word; it is broken to fit in the table.

Table 4-2 Obsolete DFT Compiler Commands

Command	Notes
`create_test_patterns`	All Test Compiler ATPG functionality is obsolete. Migrate to TetraMAX.

Table 4-2 Obsolete DFT Compiler Commands (Continued)

Command	**Notes**
`create_testsim_model`	
`delete_test`	
`fault_simulate`	
`insert_test`	Use `insert_dft` instead.
`prepare_testsim_vectors`	
`report_test -dont_fault -mask_fault -testsim_timing -faults -class -coverage -incremental -atpg_conflicts`	
`restore_test`	
`set_min_fault_coverage`	
`set_scan`	
`set_scan_chain`	Replace with `set_scan_path`.
`set_scan_style`	Replace with `set_scan_configuration -style`.
`set_test_dont_fault`	
`set_test_isolate`	Replace with `set_scan_element false`.
`set_test_keep_fault_data`	
`set_test_mask_fault`	
`set_test_methodology`	
`set_test_require`	
`set_test_routing_order`	

Table 4-2 Obsolete DFT Compiler Commands (Continued)

Command	Notes
`set_test_unmask_fault`	
`set_testsim_output_strobe`	

Using the db2xg Converter Script

The `db2xg` script (which is located at $SYNOPSYS/auxx/syn/dftc/db2xg) converts your existing dctcl script to the XG mode command set. If your script uses dcsh syntax rather than dctcl syntax, run the `dc-transcript` script on your script before running `db2xg`. For more information about the `dc-transcript` utility, see the Design Compiler documentation.

You run the `db2xg` script from the system prompt. The syntax for running this script is

```
db2xg dctcl_script xg_script [-silent]
   [-help | -man | -version]
```

Argument	Description
dctcl_script	The name of the existing dctcl script to be converted.
xg_script	The name of the XG mode script generated by `db2xg`.
`-silent`	Specifies that the script should run silently. By default, the script prints warning messages during execution.
[`-help`]	Prints basic usage information for `db2xg`.
[`-man`]	Prints the man page for `db2xg`.
[`-version`]	Prints the version of the `db2xg` script.

Note:

The `db2xg` script is a Perl script. To run this script, you must be able to run Perl from /usr/bin/perl.

Known Limitations

The db2xg script has the following known limitations:

- Your dctcl script must use the Unified Test DRC flow.

  The db2xg script does not support the original DRC flow. For information about converting your script from the original DRC flow to the Unified Test DRC flow, see “Converting to the Unified Test DRC Flow” on page 4-16.

- The script fully supports only the multiplexed flip-flop scan style.

  If your design uses a scan style other than multiplexed flip-flop, the db2xg script correctly translates commands such as set_scan_style, but it might not correctly specify the test clocks.

- Scan path specifications might not translate correctly.

  In XG mode, the set_scan_path command is order dependent (it must occur after the test ports related to the path have been defined), so the script holds this command in memory until the script finds a command unrelated to scan specification, such as create_test_protocol, dft_drc, preview_dft, or insert_dft.

  Because db2xg waits to output the set_scan_path command, the result can be incorrect if the specification includes a Tcl variable (db2xg does not perform variable substitution) or occurs within a Tcl loop.

For example, assume that your dctcl script contains the following commands:

```
for {set i 1} {$i < 20} {incr i} {
   set_scan_path c${i} -chain_length 50
}
for {set j 1} {$j < 20} P{incr j} {
   set_scan_signal test_scan_in -port si${j} \
      -chain c${j}
   set_scan_signal test_scan_out -port so${j} \
      -chain c${j}
}
create_test_protocol
```

In this case, `db2xg` would convert the `set_scan_path` command incorrectly because it would not know that c${i} and c${j} resolve to the same value. Even if the same variable were used for the two loops, the generated `set_scan_path` command would be incorrect, because only a single command would be generated, and it would be outside the loop (so the variable would be undefined), as shown in the following code:

```
for {set i 1} {$i < 20} {incr i} {
}
for {set i 1} {$i < 20} P{incr i} {
   set_dft_signal -view spec -port si${i} \
      -type ScanDataIn
   set_dft_signal -view spec -port so${i} \
      -type ScanDataOut
}
set_scan_path c${i} -view spec -scan_data_in si${i} \
   -scan_data_out so${i} -exact_length 50
create_test_protocol
```

- Trailing characters after the actual command (for example, redirection specifications) might not be correctly translated.

  If a single command is translated into multiple commands, and the original command contains information after the actual command syntax, the additional information is added only to the last command in the generated script. This might result in an incorrect translation.

  For example, assume that your script contains the following command:

  ```
  report_test -scan_path > report.log
  ```

  The resulting commands in the generated script are

  ```
  report_scan_path -chains all
  report_scan_path -cells all > report.log
  ```

  In this case, only the cell report, and not the chain report, would be included in the log file.

- Multiline commands are merged into a single line during conversion.

  This limitation does not cause a problem for script execution; however, the generated script does not retain the formatting of the original script.

Because of these limitations, inspect the generated script to ensure that it is correct. You can use the enhanced reporting capabilities provided in XG mode to help with this task.

Converting to the Unified Test DRC Flow

The Unified Test DRC flow uses the same DRC engine as TetraMAX, thereby ensuring consistent rules and messages between DFT Compiler and TetraMAX.

This section briefly describes how to migrate to the Unified Test DRC flow. For detailed information about test design rule checking using the Unified Test DRC flow, see the following chapters in the *DFT Compiler User Guide Vol. 1: Scan (XG Mode)*:

- Chapter 2, "Running RTL Test Design Rule Checking"
- Chapter 4, "Pre-Scan Design Rule Checking"
- Chapter 5, "Architecting Your Test Design"

To migrate to the Unified Test DRC flow,

1. Add the `create_test_protocol` command to your script.

   In the Unified Test DRC flow, DFT Compiler no longer infers the test protocol during test design rule checking. You must explicitly run `create_test_protocol` (or `read_test_protocol`) before running test design rule checking.

   Before running the `create_test_protocol` command, you must define the test protocol, including the test clocks, resets, and constant values. For information about defining the test protocol, see the DFT Compiler documentation.

   By default, the `create_test_protocol` command does not infer test clocks and resets. For the best results (and reduced runtime), use the `set_dft_signal -view existing_dft` command to explicitly define these signals before running create_test_protocol. If you would rather have DFT Compiler infer these signals, specify the -infer_clock or -infer_aynch options when you run create_test_procotol.

2. Replace all test design rule checking commands with the dft_drc command.

   In the Unified Test DRC flow, the dft_drc command performs test design rule checking at all design stages, replacing the rtldrc, check_test, check_scan, and check_dft commands.

About the set_dft_signal Command

The set_dft_signal command replaces several commands from the Unified Test DRC flow. In XG mode, this command is used to identify all test signals, both those that are used in the existing logic (-view existing_dft) and those that are to be added (-view spec). You use the -type option to specify the type of test signal. Table 4-3 lists the valid keyword values.

Important:
The type values are case-sensitive.

Table 4-3 DFT Signal Type Keywords

Type keyword	Description	DB mode signal type
Constant	Test-hold pin (replaces `set_test_hold`)	(no equivalent)
MasterClock	System clock	test_clock
Reset	Asynchronous set or reset signal	test_asynch test_asynch_inverted
ScanClock	Shorthand for MasterClock + ScanMasterClock (for use with multiplexed flip-flop scan style only)	(no equivalent)
ScanDataIn	Scan input	test_scan_in
ScanDataOut	Scan output	test_scan_out
ScanEnable	Scan enable	test_scan_enable
ScanMasterClock	Master clock	test_scan_clock test_scan_clock_a
ScanSlaveClock	Slave clock in LSSD scan style	test_scan_clock_b
TestData	External test data pin	test_point_normal_data_source_async test_point_normal_data_source test_point_normal_data_source_clock test_point_clock
TestMode	Test mode pin	test_mode

Performing Scan Synthesis

When performing scan synthesis, you use a combination of the descriptive view (`-view existing_dft`) and the prescriptive view (`-view spec`). When you define existing signals that are used in test mode, you use the descriptive view. When you define the scan structure you want inserted, you use the prescriptive view.

As in DB mode, you follow these steps to perform scan synthesis in XG mode (after reading the RTL design):

1. Perform RTL test design rule checking.
2. Perform 1-pass scan synthesis.
3. Perform pre-scan test design rule checking.
4. Perform scan insertion.
5. Analyze the post-scan design.

The following sections provide sample scripts for performing these steps in XG mode.

Performing RTL Test Design Rule Checking

To perform RTL test design rule checking,

1. Define the test protocol.

   Use the `-view existing_dft` option with the `set_dft_signal` command to specify how existing signals are used in test mode.

2. Create the test protocol.

3. Run RTL test design rule checking.

Table 4-4 shows a sample script that performs RTL test design rule checking. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-4 RTL Test Design Rule Checking

XG mode script	DB mode script
`# Define the test protocol`	
`set_scan_configuration \`	`set_scan_configuration \`
`   -style multiplexed_flip_flop`	`   -style multiplexed_flip_flop`
`set_dft_signal -view existing_dft \`	`create_test_clock CLK \`
`   -type ScanClock \`	`   -waveform [list 45 55]`
`   -port CLK -timing [list 45 55]`	
`set_dft_signal -view existing_dft \`	`set_signal_type test_asynch_inverted`
`   -type Reset \`	`RESETN`
`   -port RESETN -active_state 0`	
`# Create the test protocol`	
`create_test_protocol`	`create_test_protocol`
`# Run RTL test design rule checking`	
`dft_drc`	`dft_drc`

Performing 1-Pass Scan Synthesis

To perform 1-pass synthesis, use the `compile -scan` command, just as you do in DB mode. There are no changes to this step.

Performing Pre-Scan Test Design Rule Checking

To perform pre-scan test design rule checking,

1. Create the test protocol.

   You previously defined the test protocol when you ran RTL test design rule checking (if you did not, you must do so now). The design has changed, so you must regenerate the test protocol.

2. Run pre-scan test design rule checking.

Example 4-1 shows a sample script that performs pre-scan test design rule checking. The commands to perform this task are the same in XG mode and DB mode.

Example 4-1   Pre-Scan Test Design Rule Checking

```
# Create the test protocol
create_test_protocol

# Run pre-scan test design rule checking
dft_drc
```

Performing Scan Insertion

To perform scan insertion,

1. Define the scan configuration.

   Use the `-view spec` option with the `set_dft_signal` and `set_scan_element` commands to define the scan structures that you want inserted into your design. For details about these commands, see the man pages.

   To specify the scan routing order, use the `set_scan_path -view spec` command. In DB mode, when you specify the routing order, you must specify the entire scan chain. In XG mode, you can specify cells at the beginning (`-head_elements`) or end (`-tail_elements`) of the scan chain. DFT Compiler inserts cells between the specified cells.

2. Preview the scan chains.

3. Insert the scan chains.

   To prevent DFT Compiler from renaming the subdesigns, run the `set_dft_insertion_configuration -preserve_design_name true` command before inserting the scan chains.

Table 4-5 shows a sample script that performs scan insertion. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-5 Scan Insertion

XG mode script	DB mode script
`# Define the scan configuration`	

`set_dft_signal -view spec \`
`   -type ScanDataIn -port TEST_SI`
	`set_scan_signal test_scan_in \`
	`   -port TEST_SI`

`set_dft_signal -view spec \`
`   -type ScanDataOut -port TEST_SO`
	`set_scan_signal test_scan_out \`
	`   -port TEST_SO`

`set_dft_signal -view spec \`
`   -type ScanEnable -port TEST_SE`
	`set_scan_signal test_scan_enable \`
	`   -port TEST_SE`

`# Preview the scan chains`	
`preview_dft`	`preview_dft`
`# Insert the scan chains`	
`insert_dft`	`insert_dft`

Analyzing the Post-Scan Design

To analyze the post-scan design,

1. Save the design and test protocol.
2. Run post-scan test design rule checking.
3. Report on the scan structures.

Table 4-6 shows a sample script that performs post-scan test design rule checking. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-6 Post-Scan Design Analysis

XG mode	DB mode
# Save the design and test protocol	
write -format ddc -hier \ -output *my_design*.ddc	write -format db -hier \ -output *my_design*.db
write_test_protocol \ -output my_*design*_final.spf	write_test_protocol \ -output my_*design*_final.spf
# Analyze the scan design	
dft_drc	dft_drc
report_scan_path -view existing_dft \ -chain all	report_test -scan
report_scan_path -view existing_dft \ -cell all	

Complete Scan Insertion Example

Table 4-7 shows a sample script that performs the complete scan insertion process. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-7 Complete Scan Insertion Script

XG mode script	DB mode script
# Read RTL design	
read_verilog *my_design*.v current_design *my_design* link	read_verilog *my_design*.v current_design *my_design* link
# Define the test protocol	
set_scan_configuration \ -style multiplexed_flip_flop	set_scan_configuration \ -style multiplexed_flip_flop

**set_dft_signal -view existing_dft \ **
   **-type ScanClock \ **
   -port CLK -timing [list 45 55]
	create_test_clock CLK \
	   -waveform [list 45 55]

**set_dft_signal -view existing_dft \ **
   **-type Reset \ **
   -port RESETN -active_state 0
	set_signal_type test_asynch_inverted
	RESETN

# Create the test protocol	
create_test_protocol	create_test_protocol
# Run RTL test design rule checking	
dft_drc	dft_drc
# Run 1-pass scan synthesis	
compile -scan	compile -scan
# Run pre-scan test design rule checking	
create_test_protocol dft_drc	

Table 4-7 Complete Scan Insertion Script (Continued)

XG mode script	DB mode script
`# Define the scan configuration`	

`set_dft_signal -view spec \`
`   -type ScanDataIn -port TEST_SI`
	`set_scan_signal test_scan_in \`
	`   -port TEST_SI`

`set_dft_signal -view spec \`
`   -type ScanDataOut -port TEST_SO`
	`set_scan_signal test_scan_out \`
	`   -port TEST_SO`

`set_dft_signal -view spec \`
`   -type ScanEnable -port TEST_SE`
	`set_scan_signal test_scan_enable \`
	`   -port TEST_SE`

`# Preview the scan chains`	
`preview_dft`	`preview_dft`
`# Insert the scan chains`	
`insert_dft`	`insert_dft`
`# Save the design and test protocol`	

`write -format ddc -hier \`
`   -output` ***`my_design`*****`.ddc`**
	`write -format db -hier \`
	`   -output` *`my_design`*`.db`

`write_test_protocol \`
`   -output m`*`y_design`*`_final.spf`
	`write_test_protocol \`
	`   -output m`*`y_design`*`_final.spf`

`# Analyze the scan design`	
`dft_drc`	`dft_drc`

`report_scan_path -view existing_dft \`
`   -chain all`
`report_scan_path -view existing_dft \`
`   -cell all`
	`report_test -scan`

Performing Scan Extraction

When performing scan extraction, you always use the descriptive view (`-view existing_dft`), because you are defining test structures that already exist in your design.

To perform scan extraction,

1. Define the scan input and scan output for each scan chain.

   Use the `-view existing_dft` option with the `set_scan_path` and `set_dft_signal` commands to define these relationships.

2. Define the test clocks, resets, and test mode signals.

   Use the `-view existing_dft` option with the `set_dft_signal` command to specify these signals.

3. Run `dft_drc` to extract the scan chains.

Table 4-8 shows a sample script that performs scan extraction. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Note:

You can use this flow to perform scan reordering after physical synthesis by running the `preview_dft -physical` and `insert_dft -physical` commands after you run the commands shown in the sample script.

Table 4-8 Scan Extraction

XG mode	DB mode
# Define the scan chains	
set_scan_configuration \	set_scan_configuration \
-style multiplexed_flip_flop	-style multiplexed_flip_flop
	set_scan_state scan_existing
set_dft_signal -view existing_dft \	set_signal_type test_scan_in TEST_SI \
-type ScanDataIn -port TEST_SI	-index 1
set_dft_signal -view existing_dft \	set_signal_type test_scan_out TEST_SO \
-type ScanDataOut -port TEST_SO	-index 1
set_dft_signal -view existing_dft \	set_signal_type test_scan_enable TEST_SE
-type ScanEnable -port TEST_SE	
set_scan_path chain1 \	
-view existing_dft \	
-scan_data_in TEST_SI \	
-scan_data_out TEST_SO	
# Define the test signals	
set_dft_signal -view existing_dft \	create_test_clock CLK -waveform {45 55}
-type ScanClock -port CLK \	
-timing [list 45 55]	
set_dft_signal -view existing_dft \	set_signal_type test_asynch_inverted \
-type Reset -port RESETN \	RESETN
-active_state 0	
# Extract scan chains	
create_test_protocol	create_test_protocol
dft_drc	dft_drc -infer_scan_structure
report_scan_path -view existing_dft \	report_test -scan
-chain all	
report_scan_path -view existing_dft \	
-cell all	

Using Rapid Scan Synthesis

Rapid scan synthesis is the process of stitching the scan chains together without optimizing them. In XG mode, use the `set_dft_insertion_configuration -synthesis_optimization none` command to prevent optimization during scan insertion.

Using Hierarchical Scan Synthesis

The hierarchical scan synthesis process is the same in XG mode as it is in DB mode. (For details about hierarchical scan synthesis, see Chapter 1, "Key DFT Flows and Methodologies" in the DFT Compiler User Guide Vol. 1: Scan (XG Mode).)

There are a few differences in command and variable names, which are detailed in this section.

To create test models during scan insertion, set the `test_xg_use_models` variable to `true` before running `insert_dft`. (DB mode uses the `test_use_test_models` variable for this purpose.)

To use test models at the top level,

- Set the `test_xg_use_models` variable to `true`.
- Use the `read_test_models` command to read the test models.
- Use the `use_test_models` command to specify which subdesigns use test models.

Use the `-true` *`design_list`* option to specify which subdesigns use test models. Use the `-false` *`design_list`* option to specify which subdesigns do not use test models.

To report the current settings, use the `report_use_test_models` command.

Table 4-9 shows a sample script that creates a test model. Table 4-10 shows a sample script that uses the test model. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-9 Hierarchical Scan Synthesis (Module Level)

XG mode	DB mode
`# Enable test model generation`	
`set test_xg_use_models true`	`set test_use_test_models true`
`# Perform scan insertion`	
`compile -scan`	`compile -scan`
`set_dft_signal ...`	`set_signal_type ...`
`create_test_protocol`	`create_test_protocol`
`dft_drc`	`dft_drc`
`set_scan_configuration ...`	`set_scan_signal ...`
`preview_dft`	`preview_dft`
`insert_dft`	`insert_dft`
`# Save test model`	
`write_test_model -output mod1.ctldb`	`write_test_model -output mod1.ctldb`

Table 4-10 Hierarchical Scan Synthesis (Top Level)

XG mode	DB mode
`# Enable test model generation`	
`set test_xg_use_models true`	`set test_use_test_models true`
`# Read design and test model`	
`read_test_model mod1.ctldb`	`read_test_model mod1.ctldb`
`read_verilog top.v`	`read_verilog top.v`
`# Enable use of test models throughout the design hierarchy`	
`use_test_model -true top`	
`# Perform scan insertion at the top level`	
`dft_drc`	`dft_drc`
`set_scan_configuration ...`	`set_scan_signal ...`
`preview_dft`	`preview_dft`
`insert_dft`	`insert_dft`

Using AutoFix

The AutoFix capability can automatically fix scan rule violations associated with uncontrollable clocks, uncontrollable asynchronous set signals, or uncontrollable asynchronous reset signals.

To use the AutoFix capability,

1. Enable the desired capabilities.

   You use the `set_dft_configuration` command to enable these capabilities. Table 4-11 shows the options used to control the various capabilities.

Table 4-11 Options to Enable AutoFix Capabilities

To enable fixing of	Use this option
Uncontrollable clocks	`-fix_clock enable`
Uncontrollable asynchronous reset signals	`-fix_reset enable`
Uncontrollable asynchronous set signals	`-fix_set enable`

2. Specify the control signals.

   You use the `set_autofix_configuration` command to specify global control signals. (To remove the specification, use the `reset_autofix_configuration` command.)

   You use the `set_autofix_element` command to specify the local control signals. (To remove the specification, use the `reset_autofix_element` command.)

   Note:

   Although the `set_autofix_configuration` and `set_autofix_element` commands exist in DB mode, their options and functionality differ in the two modes.

   Both commands have the same set of options. Table 4-12 defines the options used to fix clocks, asynchronous set signals, or asynchronous reset signals. For more information, see the man pages and the DFT Compiler documentation.

Table 4-12 *AutoFix Configuration Options*

Option	Description
`-type clock | reset | set`	You must specify the AutoFix type that uses this configuration. You can define different configurations for each AutoFix type and set of elements.
`-control_signal` *`signal_name`*	Specifies the name of the test mode control signal. This signal must be defined as either a TestMode or ScanEnable signal (with the `set_dft_signal` command). If you do not specify this option, DFT Compiler uses the TestMode signals in your design (as defined by `set_dft_signal`). If there are no TestMode signals in your design, DFT Compiler creates one.
`-test_data` *`signal_name`*	Specifies the name of the externally controllable test data signal. This signal must be defined as a TestData signal (with the `set_dft_signal` command). If you do not specify this option, DFT Compiler creates a new port.
`-include_elements` *`object_list`*	Specifies the set of design objects that are considered for fixing violations. By default, all objects are considered.
`-exclude_elements` *`object_list`*	Specifies the set of design objects that are not considered for fixing violations. By default, all objects are considered.

Table 4-13 shows a sample script that performs AutoFix. Both the XG mode and DB mode (Unified Test DRC flow) versions are provided for comparison. In the XG mode script, the commands that differ from DB mode are shown in bold.

Table 4-13 AutoFix

XG mode	DB mode
`# Enable AutoFix`	
`set_dft_configuration \ -fix_clock enable \ -fix_reset enable \ -fix_set enable`	`set_dft_configuration -autofix`
`# Perform scan insertion`	
`compile -scan`	`compile -scan`
`set_dft_signal ...`	`set_signal_type ...`
`create_test_protocol`	`create_test_protocol`
`dft_drc`	`dft_drc`
`set_scan_configuration ...`	`set_scan_signal ...`
`preview_dft -test_points all \ -show all`	`preview_dft -test_points all \ -show all`
`insert_dft`	`insert_dft`

Reporting

In XG mode, all DFT specification commands have corresponding reporting commands. To report what exists in the design, use the `-view existing_dft` option on the reporting command. To report what you have specified for insertion, use the `-view spec` option (this is the default).

Example 4-2 through Example 4-6 show sample reports from many of the XG mode reporting commands.

Example 4-2 A DFT Configuration Report

```
dc_shell-xg-t> report_dft_configuration

****************************************
Report : DFT configuration
Design : test
Version: 2003.12-DFT-POWER-BETA1
Date   : Fri Aug 22 16:10:05 2003
****************************************

DFT Structures                         Status
--------------                         --------
Scan:                                  Enable
Autofix:                               Enable
Test point:                            Disable
Logic BIST:                            Disable
Memory BIST:                           Disable
Wrapper:                               Disable
Integration:                           Disable
Boundary scan:                         Disable
```

Example 4-3 A Scan Configuration Report

```
dc_shell-xg-t> report_scan_configuration
****************************************
Report : Scan configuration
Design : SYNCH
Version: 2003.12-DFT-POWER-BETA1
Date   : Fri Aug 22 15:48:24 2003
****************************************

========================================
TEST MODE: Internal_scan
VIEW     : Specification
========================================
Chain count:                           Undefined
Scan Style:                            Multiplexed flip-flop
Maximum scan chain length:             Undefined
Preserve multibit segments:            True
Clock mixing:                          Not defined
Internal clocks:                       False
Add lockup:                            True
Insert terminal lockup:                False
Create dedicated scan out ports:       False
Shared scan in:                        0
Bidirectional mode:                    No bidirectional type
```

Example 4-4 A DFT Signal Report

```
dc_shell-xg-t> report_dft_signal -view existing_dft

****************************************
Report : DFT signals
Design : SYNCH
Version: 2003.12-DFT-POWER-BETA1
Date   : Fri Aug 22 15:48:51 2003
****************************************

========================================
TEST MODE: Internal_scan
VIEW     : Existing DFT
========================================
Port              SignalType          Active    Hookup    Timing
----------        ----------          ------    ------    ------
hrst_L            Reset               0         -         P 100.0 R 55.0 F 45.0
mrxc              ScanMasterClock     1         -         P 100.0 R 45.0 F 55.0
mrxc              MasterClock         1         -         P 100.0 R 45.0 F 55.0
clk3              ScanMasterClock     1         -         P 100.0 R 45.0 F 55.0
clk3              MasterClock         1         -         P 100.0 R 45.0 F 55.0
clk2              ScanMasterClock     1         -         P 100.0 R 45.0 F 55.0
clk2              MasterClock         1         -         P 100.0 R 45.0 F 55.0

dc_shell-xg-t> report_dft_signal -view spec
****************************************
Report : DFT signals
Design : SYNCH
Version: 2003.12-DFT-POWER-BETA1
Date   : Fri Aug 22 16:25:11 2003
****************************************

========================================
TEST MODE: Internal_scan
VIEW     : Specification
========================================
Port              SignalType          Active    Hookup    Timing
----------        ----------          ------    ------    ------
SI1               ScanDataIn          -         -         Delay 5.0
```

Example 4-5 Report on a User-Specified Scan Path

```
dc_shell-xg-t> report_scan_path -view spec -chain all

****************************************
Report : Scan path
Design : SYNCH
Version: 2003.12-DFT-POWER-BETA1
Date   : Fri Aug 22 15:50:07 2003
****************************************

========================================
TEST MODE: Internal_scan
VIEW     : Specification
========================================
Scan_path          ScanDataIn (h)      ScanDataOut (h)     ScanEnable (h)
---------------    ---------------     ---------------     ---------------
chain1             -                   -                   -
1
```

Example 4-6 An AutoFix Configuration Report

```
dc_shell-xg-t> report_autofix_configuration
 ****************************************
Report : Autofix configuration
Design : sample
Version: V-2004.06-SP1
Date   : Fri Jul  2 12:11:19 2004
****************************************
========================================
TEST MODE: all_dft
VIEW     : Specification
========================================
Fix type:                              Set
Fix method:                            Mux
Fix latches:                           Disable
Fix type:                              Clock
Fix latches:                           Disable
Fix clocks used as data:               Disable
Fix type:                              Internal_bus
Fix method:                            Enable_one
Fix type:                              External_bus
Fix method:                            Disable_all
```

Using BSD Compiler

In XG mode, BSD Compiler supports both boundary-scan insertion and compliance checking for IEEE Standards 1149.1-1993 and 1149.1-2001. It also supports BSDL file and test pattern generation.

Like the DFT Compiler user interface, the BSD Compiler user interface has been simplified in XG mode.

Note:
The `db2xg` script does not convert BSD Compiler commands.

This section covers the following topics:

- Boundary-Scan Design Flow
- Boundary-Scan Verification Flow
- Unsupported Capabilities

Boundary-Scan Design Flow

The process used for inserting boundary-scan logic is the same in XG mode as in DB mode; however many of the commands are different. In cases where the command is the same, the options might differ.

Table 4-16 shows the command correspondence between XG mode and DB mode for the commands used in the boundary-scan design flow. For detailed information about the boundary-scan design flow, see the *BSD Compiler User Guide (XG Mode)*.

Table 4-14 Boundary-Scan Design Commands

Task	**XG mode commands**	**DB mode commands**
Enable boundary-scan synthesis	`set_dft_configuration -bsd enable`	N/A
Read design	`read -format` *`fmt`*	`read -format` *`fmt`*
Specify pad cells	`define_dft_design`	`set_bsd_pad_design`
Specify data cells	`set_boundary_cell`	`set_bsd_data_cell`
Specify control cells	`set_boundary_cell`	`set_bsd_control_cell`
Specify boundary-scan cell order	`set_scan_path`	`set_bsd_path`
Specify boundary-scan signals (pre-insertion)	`set_dft_signal -view spec`	`set_bsd_signal`
Remove boundary-scan signal attributes	`remove_dft_signal`	`remove_bsd_signal`
Identify linkage ports	`set_bsd_linkage_port`	`set_bsd_linkage_port`
Define the boundary-scan configuration	`set_bsd_configuration`	`set_bsd_configuration`
Specify power-up reset of TAP controller	`set_bsd_power_up_reset`	`set_bsd_power_up_reset`
Configuring the device identification register	`set_bsd_instruction`	Set the following variables: `test_bsd_version_number`, `test_bsd_part_number`, and `test_bsd_manufacturer_id`
Define TAP controller interface	`define_dft_design`	`set_bsd_tap_element` `set_tap_elements`
Define test data register	`set_bsd_register`	`set_dft_signal` `set_scan_path`
Specify boundary-scan instructions	`set_bsd_instruction`	`set_bsd_instruction` `set_bsd_intest` `set_bsd_runbist`

Table 4-14 Boundary-Scan Design Commands (Continued)

Task	**XG mode commands**	**DB mode commands**
Preview boundary scan	`preview_dft -bsd_all`	`preview_bsd`
Insert boundary scan	`insert_dft`	`insert_bsd`
Save design	N/A	`write -donot_expand_dw`
Synthesize the boundary-scan design	N/A	`compile`

Boundary-Scan Verification Flow

The process used for verifying boundary-scan logic is the same in XG mode as in DB mode; however many of the commands are different. In cases where the command is the same, the options might differ.

Table 4-15 shows the command correspondence between XG mode and DB mode for the commands used in the boundary-scan verification flow. For detailed information about the boundary-scan verification flow, see the *BSD Compiler User Guide (XG Mode)*.

Table 4-15 Boundary-Scan Verification Commands

Task	**XG mode commands**	**DB mode commands**
Checking for existing test ports	`report_dft_signal`	`report_test -port`
Specify existing boundary-scan signals	`set_dft_signal -view existing_dft`	`set_bsd_port`
Remove boundary-scan signal attributes	`remove_dft_signal`	`remove_bsd_signal`
Identify linkage ports	`set_bsd_linkage_port`	`set_bsd_linkage_port`

Table 4-15 Boundary-Scan Verification Commands (Continued)

Task	XG mode commands	DB mode commands
Configure the compliance-enable ports	`set_bsd_compliance`	`set_bsd_compliance`
Report the compliance-enable configuration	N/A	`report_test -bsd`
Reset the compliance-enable ports	`reset_bsd_specification`	`remove_bsd_specification`
Define system clocks	`set_dft_signal`	`create_clock`
Enable automatic inferencing of boundary-scan instructions	`check_bsd -infer_instructions true`	`set_bsd_configuration -infer_instructions true`
Check compliance	`check_bsd`	`check_bsd`
Create boundary-scan patterns	`create_bsd_patterns`	`create_bsd_patterns`
Verify the boundary-scan specification	`preview_dft -bsd_all`	`report_test -bsd_configuration -bsd`
Remove the boundary-scan specification	`reset_bsd_specification`	`remove_bsd_specification`
Save the boundary-scan patterns	`write_test`	`write_test`
Generate BSDL file	`write_bsdl`	`write_bsdl`

Unsupported Capabilities

In XG mode, BSD Compiler does not support the following capabilities:

- The `write -donot_expand_dw` option

  In XG mode, the DesignWare boundary scan components are automatically linked to the design when you save the design.

- Generation of the boundary scan test patterns as a Verilog testbench

  You must save the boundary scan test patterns in STIL format. You can simulate these patterns with the VerilogDPV capablity.

- Integration with Physical Compiler

  In DB mode, BSD Compiler places the boundary scan registers close to the pad cells. This capability is not supported in XG mode.

- Scan through TAP

  In DB mode, BSD Compiler can connect the TAP signals to the internal scan enable and generate a test protocol that uses the TAP to control the internal scan chains. This capability is not supported in XG mode.

Using DFT Compiler DBIST Synthesis

The Synopsys deterministic built-in self test (DBIST) feature provides a method of testing digital logic more efficiently, while maintaining high test quality and minimizing the impact of test on designers. This feature includes X-tolerant DBIST and streaming DBIST (SDBIST).

The BIST insertion process consists of the following steps:

- BIST preparation
- BIST integration

In XG mode, you can perform these steps with a single `insert_dft` command. In DB mode, these are two separate steps. Other than this flow enhancement, the DBIST synthesis process is the same in both XG and DB mode; however some of the commands and options are different.

Note:
The `db2xg` script does not convert DBIST commands.

Table 4-16 shows the command correspondence between XG mode and DB mode for the commands used in DBIST synthesis. For detailed information about DBIST synthesis, see the *DBIST User Guide (XG Mode)*.

Table 4-16 DBIST Command Correspondence

Task	XG mode commands	DB mode commands
BIST preparation commands		
Specify BIST preparation step	set_dft_configuration -logicbist enable -wrapper enable set_logicbist_configuration -bist_ready true [-type xdbist]	set_dft_configuration -bist -core_wrapper set_bist_configuration -bist_ready [-type xdbist]
Specify insertion of control and observe points	set_dft_configuration -control_points enable -observe_points enable set_testability_configuration -type control_and_observe -max_test_points *n* -test_points_per_scan_cell *n*	set_dft_configuration -testability set_testability_configuration -method bist -max_control_points *n* -control_points_per_scan_cell *n*
Specify X-state propagation and bus fixing	set_dft_configuration -fix_xpropagation enable -fix_bus enable	set_dft_configuration -autofix -bist set_autofix_configuration -xprop true -bus true
BIST integration commands		
Specify BIST integration step	set_dft_configuration -logicbist enable set_logicbist_configuration -type dbist | xdbist -integration true	set_dft_configuration -dbist | -xdbist set_bist_configuration -integration

Table 4-16 DBIST Command Correspondence (Continued)

Task	XG mode commands	DB mode commands
One-step BIST synthesis commands		
Specify BIST synthesis	`set_dft_configuration -logicbist enable -wrapper enable` `set_logicbist_configuration -type dbist | xdbist -integration true`	N/A
Specify X-state propagation and bus fixing	`set_dft_configuration -fix_xpropagation enable -fix_bus enable`	N/A
Common commands		
Remove BIST configuration	`remove_dft_configuration`	`remove_dft_configuration`
Report on the BIST configuration	`report_logicbist_configuration`	`report_test -bist`
Create test protocol	`create_test_protocol`	`create_test_protocol`
Check BIST design rules	`dft_drc`	`dft_drc`
Preview test structures	`preview_dft`	`preview_dft`
Insert test structures (based on configuration)	`insert_dft`	`insert_dft`

GUI Support

In XG mode, the Test menu options and the DRC violation browser are disabled in Design Vision and the Physical Compiler GUI. You must invoke all DFT commands from the command line.

To debug your DRC violations, you can use the `dft_drc_interactive` Tcl procedure to invoke the TetraMAX GUI from within DFT Compiler. The TetraMAX GUI provides advanced DRC analysis capabilities.

You do not need a TetraMAX license to use these DRC analysis capabilities. However, you must have a fully mapped netlist and have access to simulation libraries that can be used by TetraMAX.

For more information about using the `dft_drc_interactive` procedure and a link to download it, see the SolvNet article, “Using TMAX for interactive debug inside DFT Compiler” (located at https://solvnet.synopsys.com/retrieve/012388.html).

5

Using Physical Compiler in XG Mode

This chapter describes the differences between running Physical Compiler in XG mode and DB mode.

This chapter contains the following sections:

Note:

The information provided in Chapter 1, “Introduction to XG Mode,” and Chapter 2, “XG Mode Design Database Formats,” applies to Physical Compiler as well. For detailed information about running Physical Compiler, see the Physical Compiler documentation.

Physical Libraries

By default, Physical Compiler uses the Milkyway reference library as the physical library in XG mode. This is the same physical library that is used by the Jupiter and Astro tools. To specify the Milkyway reference library, set the `mw_reference_library` variable.

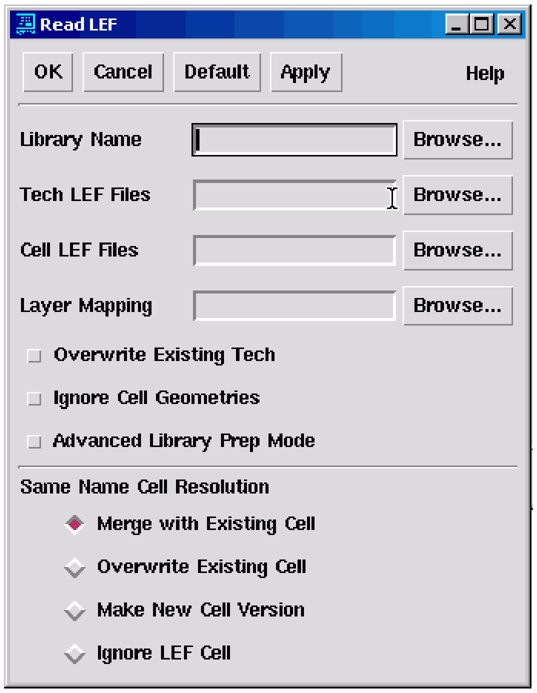
If you do not have a Milkyway reference library, you can use the Milkyway tool to generate one from your LEF library files or your .pdb library files. After generating the Milkyway reference library, you must generate a technology file (for use with the `create_mw_design` command). Physical Compiler also provides a way to revert to using .pdb format physical libraries. The following sections describe these tasks.

Generating a Milkyway Reference Library from LEF

To generate a Milkyway reference library from your LEF library files,

1. Choose Cell Library > LEF In from the Milkyway menu bar.

The Read LEF dialog box appears.



Library Name

Tech LEF Files

Cell LEF Files

Layer Mapping

☐ Overwrite Existing Tech

☐ Ignore Cell Geometries

☐ Advanced Library Prep Mode

Same Name Cell Resolution

[✓ Merge with Existing Cell]

Overwrite Existing Cell

Make New Cell Version

Ignore LEF Cell

2. Enter the reference library name (Library Name), the technology LEF files (Tech LEF Files), and the cell LEF files (Cell LEF Files).

   Alternatively, you can click Browse to use the browse capability to select the reference library and LEF files.

   If the same LEF file is used for both technology and cell information, enter the file name in both the Tech LEF Files and Cell LEF Files fields.

3. Click OK to generate the specified reference library.

4. Check the log file for warnings or errors.

   The log file is located in the directory where you invoked the Milkyway tool. The name of the log file is Milkyway.log.*date*, where date is the date and time when you invoked the Milkyway tool.

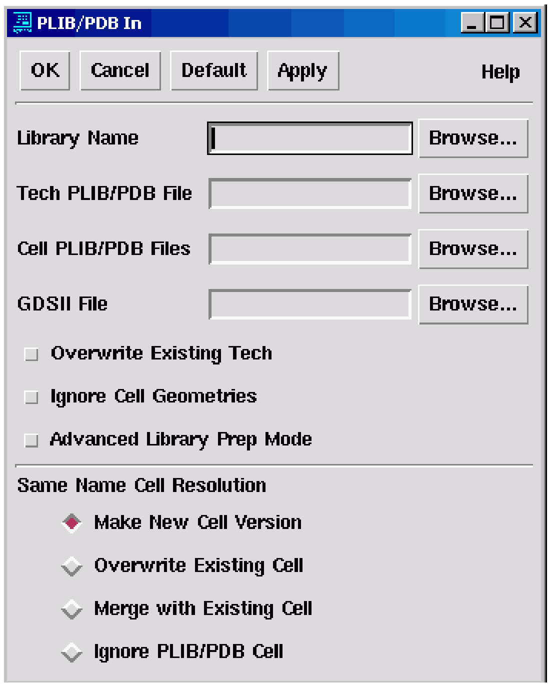
For more information about this process, including debugging information and information about the other fields in the Read LEF dialog box, see the Milkyway documentation.

Generating a Milkyway Reference Library from .pdb

To generate a Milkyway reference library from your .pdb library files,

1. Choose Cell Library > Import PLIB from the Milkyway menu bar.

   The PLIB/PDB In dialog box appears.



2. Enter the reference library name (Library Name), the technology .pdb file (Tech PLIB/PDB File), and the cell .pdb files (Cell PLIB/PDB Files).

   Alternatively, you can click Browse to use the browse capability to select the reference library and .pdb files.

Caution!

The technology .pdb file must contain layer definitions; otherwise the generated Milkyway reference library will be invalid.

If the same .pdb file is used for both technology and cell information, enter the file name in both the Tech PLIB/PDB File and Cell PLIB/PDB Files fields.

3. Click OK to generate the specified reference library.
4. Check the log file for warnings or errors.

   The log file is located in the directory where you invoked the Milkyway tool. The name of the log file is Milkyway.log.*date*, where date is the date and time when you invoked the Milkyway tool.

For more information about this process, including debugging information and information about the other fields in the PLIB/PDB In dialog box, see the Milkyway documentation.

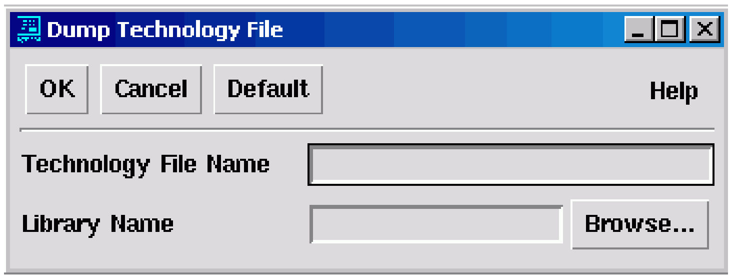
Generating a Milkyway Technology File

You use the Milkyway Environment tool to extract the technology file from the Milkyway reference library.

To extract the technology file,

1. Choose Library > Dump Tech File from the Milkyway menu bar.

The Dump Technology File dialog box appears.



2. Enter the technology file name (Technology File Name) and the Milkyway reference library name (Library Name).

   Alternatively, you can click Browse to use the browse capability to select the reference library.

3. Click OK to generate the specified technology file.

Using .pdb Libraries

The Milkyway reference library is the recommended physical library for the following reasons:

- A common physical library is used across the Galaxy platform
- The enhanced capabilities for reading and writing DEF and PDEF files require the Milkyway reference library

However, you can revert to using .pdb format physical libraries by setting the `use_pdb_lib_format` variable to true. When you use the .pdb physical libraries, you also revert to the previous DEF and PDEF implementations (you are not using the common Milkyway-based DEF and PDEF implementation).

If you set the `use_pdb_lib_format` variable to true, you must specify the physical library by setting the `physical_library` variable.

Supported Physical Design Flows

Physical Compiler supports the following flows in XG mode:

- Synopsys tool flow
- Third-party tool flow

This section provides an overview of these flows. You can find more information about these flows in the *Physical Compiler User Guide, Volume 1*.

Synopsys Tool Flow

If you are using only Jupiter, Physical Compiler, and Astro, you are using the Synopsys tool flow. This tool flow uses the Milkyway design library to store all design information.

Before running Physical Compiler, you must have the design floorplan that you created using Jupiter or Astro. The floorplan is saved in the Milkyway design library.

The script in Example 5-1 shows the basic commands for using this flow in XG mode. For more information about the Synopsys tool flow, see “Interacting With Other Synopsys Tools” in the *Physical Compiler User Guide, Volume 1*.

Example 5-1 Jupiter or Astro > Physical Compiler > Jupiter or Astro

```
# Set variables for running in XG mode (REQUIRED)
set mw_design_library design_dir
set mw_reference_library lib_dir
set mw_logic1_net VDD
set mw_logic0_net VSS

# Read floorplan
read_milkyway mydesign

# Set constraints
source myconstraints.sdc

# Perform physical optimization
physopt

# Save design database
write_milkyway -output mydesign
```

Third-Party Tool Flow

If you are using a tool other than Jupiter or Astro for floorplanning or routing, you are using the third-party tool flow. This flow uses DEF files to export the floorplanning and routing information.

Before running Physical Compiler, you must have the design floorplan that you created using a third-party tool. The design is saved in an ASCII format, such as Verilog. The floorplan is saved in DEF or PDEF format.

The script in Example 5-2 shows the basic commands for using the third-party flow in XG mode. For more information about the third-party flow, see “Interfacing with Third-Party Tools” in the *Physical Compiler User Guide, Volume 1*.

Example 5-2 Third-Party Flow

```
# Set variables for running in XG mode (REQUIRED)
set mw_design_library design_dir
set mw_reference_library lib_dir
set mw_logic1_net VDD
set mw_logic0_net VSS

# Read netlist
read_file -format fmt mydesign.fmt
link
link_physical

# Read floorplan
read_[p]def mydesign.[p]def

# Set constraints
source myconstraints.sdc

# Save design database
create_mw_design -tech_file mw_ref.tf
write_milkyway -output mydesign

# Perform physical optimization
physopt

# Save placement data
write_[p]def -output mydesign.[p]def

# Save design database
write_milkyway -output mydesign
```

Differences in Behavior

The following sections describe the differences in behavior between XG mode and DB mode.

- Checkpointing the Optimization Results
- Using Interface Logic Models
- Using Distributed Physical Synthesis
- GUI Enhancements

Checkpointing the Optimization Results

The process used for saving an intermediate design database (checkpointing) is the same in XG mode as in DB mode. You can do automatic checkpointing by setting the `physopt_checkpoint_stage` variable, or you can do manual checkpointing by pressing Control-c during optimization. However, there are differences in the checkpointing behavior.

The checkpointing capability has the following differences between XG mode and DB mode:

- In XG mode, checkpoint files are saved in the CEL view of the Milkyway design library, rather than in a .db file.

  Before using checkpointing, you must have created the Milkyway design library and specified its location by setting the `mw_design_library` variable. For information about creating a Milkyway design library, see "Creating a Milkyway Design Library" on page 2-8.

- The default file naming is different in XG mode than in DB mode.

In DB mode, the default file name is CHECKPOINT.db. You can specify a file name by setting the `compile_checkpoint_filename` variable before you run the `physopt` command.

In XG mode, the default file name is the current design name (*design_dir*/CEL/*current_design*:*version*, where *design_dir* is the location you specified in `mw_design_library)`. To specify the file name, set the `physopt_mw_checkpoint_filename` variable before you run the `physopt` command.

- You can save multiple checkpoint files in XG mode.

  In DB mode, you can save a single checkpoint file. Each successive checkpoint file overwrites the existing checkpoint file.

  In XG mode, you can save multiple checkpoint files. Each successive checkpoint file increments the version number. To conserve disk space, purge unneeded checkpoint files when you are done. For information about purging CEL versions, see "Purging Versions from the CEL View" on page 2-21.

Using Interface Logic Models

Interface logic model (ILM) behavior differs between XG mode and DB mode in the following ways:

- The interface logic models are stored in the ILM view of the Milkyway design library rather than in a .db file.
- When you save a design that contains ILMs, the ILMs are not saved. You must explicitly save each ILM.
- You do not read ILMs directly in XG mode; instead you specify the ILMs in the `link_library` variable and then Physical Compiler automatically loads them when you read the top-level design (in Milkyway format).

In addition, the following ILM capabilities are available only in XG mode:

- ILMs can be displayed in the GUI.
- The `create_ilm` command automatically handles `dont_touch` subblocks
- The `get_location` command can return the coordinates of pins on an ILM instance.
- You can perform on-route optimization on designs that contain ILMs.

For detailed information about using interface logic models in XG mode, see the *Interface Logic Model User Guide*.

Using Distributed Physical Synthesis

In XG mode, distributed physical synthesis uses the Milkyway format rather than the .db format. If you read your design in Milkyway format, distributed physical synthesis automatically uses the Milkyway format.

Note:

> If you read your design in .db format, distributed physical synthesis uses the .db format. For the best results, start with a Milkyway design, rather than a .db design. For information about converting your .db design to Milkyway format, see "Converting From .db Format to Milkyway Format" on page 2-23

GUI Enhancements

In XG mode, the Physical Compiler GUI provides three modes for displaying flylines:

- Pin to pin (default)

  This mode displays the flylines between two core cells.

- Macro to macro

  This mode displays the flylines between two macro cells.

- Pin to macro

  This mode displays the flylines between a core cell and a macro cell.

These flyline display modes are available only in XG mode.

Features Available Only in XG Mode

The following features are available only in XG mode:

- Milkyway-based DEF and PDEF support
- Relative placement

Milkyway-based DEF and PDEF Support

In XG mode, Physical Compiler uses the same Milkyway-based Design Exchange Format (DEF) and Physical Design Exchange Format (PDEF) reader and writer as the Astro and Jupiter products.

To use the Milkyway-based implementation to read and write DEF or PDEF files,

- The Milkyway Environment tool must be installed

  For information about installing the Milkyway Environment tool, see the *Installation Guide*.

- The physical libraries must be Milkyway reference libraries

  The Milkyway reference library is the default physical library. If you revert to using .pdb libraries, you cannot use the Milkyway-based DEF or PDEF implementation.

The same commands (`read_def`, `write_def`, `read_pdef`, and `write_pdef`) are used to read and write DEF and PDEF files, regardless of the implementation you use; however, the options differ. Table 5-1 lists the options for each command that are supported only in the Milkyway-based implementation.

Table 5-1 Options Supported Only in Milkyway-based Implementation

Command	Options
`read_def`	`-allow_physical_objects` `-lef_file_name`

`write_def`	`-regions_groups`
	`-macro`
	`-fixed_cell-`
	`-placed_cell`
	`-blockages`
	`-routed_net`
	`-diode_pins`
	`-notch_gap`
	`-floating_metal_fill`
	`-pg_metal_fill`
	`-lef_file_name`

`read_pdef`	`-allow_physical_objects`

For more information about using DEF and PDEF files, see Chapter 4, "Preparing Data for Physical Compiler," in the *Physical Compiler User Guide, Volume 1*.

Relative Placement

The Physical Compiler physical datapath with relative placement capability provides a way for you to create structures within psyn_shell in which you specify the relative column and row positions of instances with respect to each other. During placement and legalization, these structures, which are placement constraints called relative placement structures, are preserved and the cells in each structure are placed as a single entity. Relative placement is also called physical datapath and structured placement.

For details about using relative placement, see Chapter 12, “Physical Datapath with Relative Placement,” in the *Physical Compiler User Guide, Volume 1*.

Unsupported Capabilities

The following commands are not supported in XG mode:

- `compile_physical`
- `reoptimize_design`
- `read_mdb`

  Use the `read_milkyway` command to read the Milkyway design library. For information about reading Milkyway design libraries created with the `write_mdb` command, see “Limitations When Reading Milkyway Format” on page 2-17.

- `write_mdb`

  Use the `write_milkyway` command to save your design in a Milkyway design library. For more information, see "Saving a Design in Milkyway Format" on page 2-9.

- `change_site_name`

  Use the `mw_site_name_mapping` variable to define the name mappings. For more information about this variable, see "Reading a Design in Milkyway Format" on page 2-12.

In addition, the Physical Compiler clock tree synthesis capabilities are not supported in XG mode. To perform clock tree synthesis, use the clock tree synthesis features in Astro.

6

Using Power Compiler in XG Mode

This chapter describes the differences between running Power Compiler in XG mode and DB mode.

This chapter contains the following sections:

Benefits of XG Mode

XG mode provides the following benefits for Power Compiler:

- A 35 percent reduction in memory usage
- Up to 10 times faster runtime on some features
- New features that are available only in XG mode (for information about the new features, see "Features Available Only in XG Mode" on page 6-3)

Differences in Command Behavior

All Power Compiler commands behave the same in XG mode as they do in DB mode. However, if you use the DFT Compiler `hookup_testports` command in your flow, you need to be aware of a change in the method of identifying test ports and update your scripts accordingly.

In DB mode, you identify the test ports by using the `set_signal_type` command (for scan enable ports) or `set_test_hold` and `set_attribute` commands (for test mode ports). In XG mode, you must use the `set_dft_signal` command to identify the test ports. For more information about the `set_dft_signal` command, see Chapter 4, "Using DFT Compiler in XG Mode."

In addition, after you execute the `hookup_testports` command on a design in XG mode, if you save the design in .db format (which is not recommended), the resulting .db file is incompatible with DFT Compiler in DB mode. You can use the resulting .db file in DB mode only if you will not be using DFT Compiler.

Features Available Only in XG Mode

Power Compiler has added the following features that are available only in XG mode:

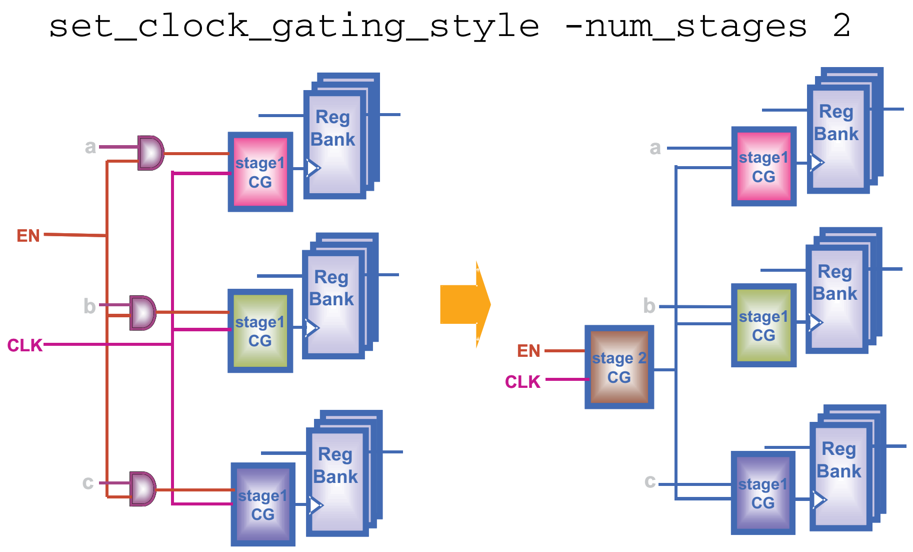
- Multistage clock gating
- Hierarchical clock gating
- Ability to reset clock-gating attributes
- Power analysis enhancements
- Stitching of power-gating signals

The following sections describe these features. For more information about these features, see the *Power Compiler User Guide*.

Multistage Clock Gating

When a clock-gating cell drives another clock-gating cell or a row of clock-gating cells, this structure is referred to as multistage clock gating. Multistage clock gating reduces power consumption by moving the clock gating closer to the root. Figure 6-1 shows a two-stage clock-gating structure.

Figure 6-1 Multistage Clock Gating



Power Compiler can automatically identify shared enable signals and use them to insert additional clock-gating levels. You can use this capability on either an RTL or gate-level design.

To enable multistage clock gating, run the `set_clock_gating_style -name_stages` *`cnt`* command before you run the `insert_clock_gating` command. For example,

```
dc_shell-xg-t> set_clock_gating_style -num_stages 2
dc_shell-xg-t> insert_clock_gating
```

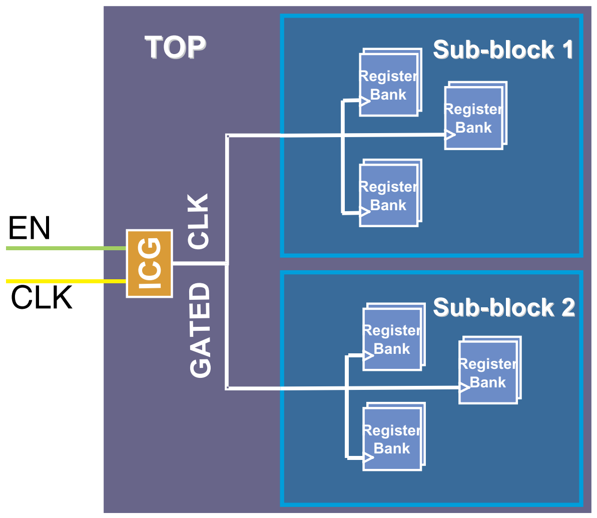
Hierarchical Clock Gating

Traditionally, the Power Compiler clock-gating technique extracts common enable conditions that are shared across registers in the same block.

With hierarchical clock gating, Power Compiler extracts common enable conditions shared across registers in different blocks. With this technique the tool looks for globally shared enables, thereby limiting the number of clock-gating cells inserted. Power Compiler inserts the hierarchical clock-gating cells in the current design when you run the `insert_clock_gating` command. Figure 6-2 shows an example of hierarchical clock gating.

Figure 6-2 Hierarchical Clock Gating

```
current_design TOP
insert_clock_gating -global
```



To enable hierarchical clock gating, specify the `-global` option when you run the `insert_clock_gating` command.

Resetting of Clock-Gating Attributes

Two new options, `-reset` and `-reset_only` *`object_list`*, have been added to the `identify_clock_gating` command. These options are supported only in XG mode.

The `-reset` option resets all clock-gating attributes in the design. The `-reset_only` *`object_list`* option resets the clock-gating attributes only on the specified cells and nets.

For more information about the `identify_clock_gating` command, see the *Power Compiler User Guide*.

Power Analysis Enhancements

The following power analysis enhancements are available only in XG mode:

- The `propagate_switching_activity` command was added to propagate switching activity data directly.

  In DB mode, you use the `report_power` command to propagate the switching activity data. In XG mode, the `propagate_switching_activity` command performs this task. For details about the `propagate_switching_activity` command, see the man page.

- The `-target_instance` option was added to the `read_saif` command.

  In DB mode, when you run the `read_saif` command the switching activity is annotated on the entire design. In XG mode, you can restrict the annotation to a specific instance by specifying the `-target_instance` option when you run the `read_saif` command.

- The `write_saif` command was added to save the switching activity data in Switching Activity Interchange Format (SAIF).

  To write the annotated switching activity to the SAIF file, enter

  ```
  dc-shell-xg-t> write_saif -output my_design.saif
  ```

  To include both annotated and propagated switching activity in the SAIF file, specify the `-propagated` option.

  To exclude state-dependent static probabilities and state-dependent or path-dependent toggle rates from the SAIF file, specify the `-exclude_sdpd` option.

  For more information about the `write_saif` command, see the man page.

Stitching of Power-Gating Signals

In both XG mode and DB mode, Power Compiler can insert state retention power-gating registers. In DB mode, you must manually stitch the power-gating signals. In XG mode, Power Compiler can automatically connect the sleep and wake signals on the registers to the power-gating signals.

To stitch the power-gating signals,

1. Identify the power-gating signals.

   Use the `set_power_gating_signal` command to identify the power-gating signals. You apply this command to top-level ports or to the output pins of instances in your design.

   If you do not identify existing power-gating signals, Power Compiler creates new top-level ports when you stitch the power-gating signals.

2. Stitch the power-gating signals.

   Use the `hookup_power_gating_ports` command to stitch the power-gating signals. This command connects the specified power-gating signals to the appropriate pins of the state retention power-gating registers. To control the naming of top-level ports added during stitching, specify the `-port_naming_style` or `-default_port_naming_style` option.

3. Report on the stitched signals.

   The `report_power_gating` command reports the following information about the stitched state retention power-gating registers: cell name, library cell name, power-gating style, power-gating pin (the library pin of the retention registers), and power-gating signal. If the power-gating signals are not stitched, `report_power_gating` displays the disabled value rather than the power-gating signal.

For more information about power gating, see the *Power Compiler User Guide*.

Example 6-1 shows a sample script used for stitching the power-gating signals.

Example 6-1 Stitching of Power-Gating Signals

```
dc_shell-xg-t> set_power_gating_signal ts \
   -library_pin SLEEP
dc_shell-xg-t> set_power_gating_signal tw \
   -library_pin WAKE
dc_shell-xg-t> hookup_power_gating_ports \
   -port_naming_style ts
dc_shell-xg-t> report_power_gating
```

A

Command Differences

The tables in this appendix list the command differences between DB mode (Tcl-based shells) and XG mode. Table A-1 on page A-2 shows the commands that are supported in DB mode but not in XG mode. Table A-2 on page A-9 shows the commands that are supported in XG mode but not in DB mode.Table A-3 on page A-15 shows the commands that are supported in both modes, but have different options in the two modes.

Table A-1 Commands Not Supported in XG Mode

Command	Note
all_cluster_cells	
all_clusters	
characterize_physical	
check_dft	Use `dft_drc` instead.
check_scan	Use `dft_drc` instead.
check_test	Use `dft_drc` instead.
compare_fsm	Command will never be supported in XG mode.
compile_clock_tree	
compile_physical	
create_cluster	
create_routing_path	
create_schematic	
create_test_clock	Use `set_dft_signal` instead.
create_test_patterns	Command will never be supported in XG mode.
create_test_schedule	
create_wire_load	
disconnect_scan_chains	
estimate_physical	
estimate_test_coverage	
extract	
get_clock_tree_attributes	

Table A-1 Commands Not Supported in XG Mode (Continued)

Command	Note
get_clock_tree_delays	
get_clock_tree_objects	
get_congested_regions	
get_design_parameter	
get_regions	
highlight_path	
infer_test_protocol	Command will never be supported in XG mode.
insert_bsd	Use `insert_dft` instead.
insert_scan	Use `insert_dft` instead.
library_analysis	
minimize_fsm	Command will never be supported in XG mode.
optimize_bsd	
parent_cluster	
plot	
preview_bsd	Use `preview_dft` instead.
preview_scan	Use `preview_dft` instead.
read_bsd_init_protocol	
read_bsd_protocol	
read_clusters	
read_init_protocol	
read_mdb	

Table A-1 Commands Not Supported in XG Mode (Continued)

Command	Note
read_trc_file	
reduce_fsm	Command will never be supported in XG mode.
remove_analysis_info	
remove_bsd_port	
remove_bsd_signal	
remove_bsd_specification	
remove_bsr_cell_type	
remove_clock_tree	
remove_clock_tree_balance_group	
remove_clock_tree_exceptions	
remove_clock_tree_options	
remove_clock_tree_root_delay	
remove_clusters	
remove_core_integration_configuration	
remove_core_wrapper_configuration	
remove_core_wrapper_specification	
remove_delay_calculation	
remove_dft_configuration	Use `reset_dft_configuration` instead.
remove_highlighting	
remove_port_configuration	
remove_wrapper_element	

Table A-1   Commands Not Supported in XG Mode (Continued)

Command	Note
reoptimize_design	
replace_fpga	
report_clock_tree	
report_clusters	
report_floorplan_macro_array	
report_floorplan_macro_options	
report_floorplan_options	
report_floorplan_pnet_options	
report_floorplan_port_options	
report_packages	
report_ph_region	
report_routability	
report_routing_options	
report_test	
report_xref	
reset_clock_tree_references	
rtl_analyzer	
rtldrc	
set_autofix_async	
set_autofix_clock	
set_bist_configuration	

Table A-1 Commands Not Supported in XG Mode (Continued)

Command	Note
set_bsd_bsr_element	
set_bsd_control_cell	
set_bsd_data_cell	
set_bsd_intest	
set_bsd_pad_design	
set_bsd_path	
set_bsd_port	
set_bsd_register	
set_bsd_runbist	
set_bsd_signal	
set_bsd_tap_element	
set_bsr_cell_type	
set_clock_tree_balance_group	
set_clock_tree_exceptions	
set_clock_tree_options	
set_clock_tree_references	
set_clock_tree_root_delay	
set_core_integration_configuration	
set_core_wrapper_cell	
set_core_wrapper_cell_design	
set_core_wrapper_configuration	

Table A-1 Commands Not Supported in XG Mode (Continued)

Command	Note
set_core_wrapper_path	
set_dft_optimization_configuration	
set_floorplan_macro_array	
set_floorplan_macro_options	
set_floorplan_options	
set_floorplan_pnet_options	
set_floorplan_port_options	
set_inverted_placement_keepout	
set_layer	
set_min_porosity	
set_pipeline_stages	
set_port_configuration	
set_rail_voltage	
set_routing_options	
set_scan_exclude	
set_scan_segment	
set_scan_signal	
set_scan_transparent	
set_signal_type	
set_tap_elements	
set_test_hold	

Table A-1 Commands Not Supported in XG Mode (Continued)

Command	Note
set_test_initial	
set_test_isolate	
set_test_model	
set_test_signal	
set_testability_element	
set_trc_configuration	
set_wired_logic_disable	
set_wrapper_element	
split_clock_gates	
trace_nets	
untrace_nets	
update_clusters	
update_script	
write_bsd_protocol	
write_clusters	
write_constraints	Command will never be supported in XG mode.
write_ibm_constraints	
write_layout_scan	
write_mdb	
write_testsim_lib	Command will never be supported in XG mode.
write_timing	Command will never be supported in XG mode.

Table A-2 Commands Supported in XG Mode Only

Command	Note
add_to_rp_group	
all_dont_touch	
all_preroute_checks	
all_rp_groups	
all_rp_hierarchicals	
all_rp_inclusions	
all_rp_instantiations	
all_rp_references	
begin_group_undo	
can_redo	
can_undo	
change_site_name	
check_target_library_subset	
create_rp_group	
define_lib_cell_class	
define_user_attribute	
disable_undo	
enable_undo	
end_group_undo	
extract_rp_group	
get_lib_cell_class	

Table A-2 Commands Supported in XG Mode Only (Continued)

Command	Note
get_power_domains	
get_rp_groups	
get_scan_chains_by_name	
get_voltage_areas	
gui_update_physical_model	
hookup_power_gating_ports	
initialize_mpc	
invalidate_undo	
last_redo_cmd_name	
last_undo_cmd_name	
order_rp_groups	
read_milkyway	
redo	
remove_boundary_cell	
remove_dft_equivalent_signals	
remove_fanout_load	
remove_from_rp_group	
remove_lib_cell_class	
remove_row_type	
remove_rp_group	
remove_rp_group_options	

Table A-2 Commands Supported in XG Mode Only (Continued)

Command	Note
remove_scan_link	
remove_scan_path	
remove_scan_replacement	
remove_target_library_subset	
remove_test_assume	
remove_test_point_element	
remove_user_attribute	
report_autofix_configuration	
report_autofix_element	
report_boundary_cell	
report_dft	
report_dft_clock_controller	
report_dft_configuration	
report_dft_design	
report_dft_equivalent_signals	
report_dft_signal	
report_dw_rp_group_options	
report_lib_cell_class	
report_logicbist_configuration	
report_scan_configuration	
report_scan_link	

Table A-2 Commands Supported in XG Mode Only (Continued)

Command	Note
report_scan_path	
report_scan_register_type	
report_scan_replacement	
report_scan_state	
report_target_library_subset	
report_test_assume	
report_test_point_element	
report_testability_configuration	
report_use_test_model	
report_wrapper_configuration	
reset_autofix_configuration	
reset_autofix_element	
reset_bsd_configuration	
reset_dft_clock_controller	
reset_dft_configuration	
reset_logicbist_configuration	
reset_mbist_configuration	
reset_mbist_controller	
reset_mbist_wrapper	
reset_scan_configuration	
reset_test_mode	

Table A-2 Commands Supported in XG Mode Only (Continued)

Command	Note
reset_testability_configuration	
reset_testbench_parameters	
reset_wrapper_configuration	
reshape_objects	
rp_group_inclusions	
rp_group_instantiations	
rp_group_references	
set_boundary_cell	
set_cell_type	
set_dft_equivalent_signals	
set_dw_rp_group_options	
set_inverted_placement_keepouts	
set_logicbist_configuration	
set_mbist_wrapper	
set_power_gating_signal	
set_rp_group_options	
set_scan_style	
set_target_library_subset	
set_test_dont_fault	
set_testbench_parameters	
set_user_attribute	

Table A-2 Commands Supported in XG Mode Only (Continued)

Command	Note
set_wrapper_configuration	
undo	
update_region	
update_voltage_area	
use_test_model	
write_dps	
write_dw_rp_group	
write_link_library	
write_milkyway	
write_rp_group	
write_scan_def	

Table A-3 Modified Commands

Command	Options in DB mode only	Options in XG mode only
change_link		-all_instances
check_bsd		-infer_instructions
check_budget	-no_environment	
check_design		-multiple_designs
check_dft	-overwrite_model	
check_scan	-overwrite_model	
check_test	-overwrite_model	
compile	-arch -background -host -xterm	
create_bsd_patterns	-stil	
create_ilm	-instances	-keep_parasitics
create_operating_conditions	-parameter1 -parameter2 -parameter3 -parameter4 -parameter5	
create_test_clock	-hookup	
define_test_mode	-existing -spec -view	-inherit
derive_regions		-guard_band_x -guard_band_y
derive_voltage_areas		-guard_band_x -guard_band_y

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
dft_drc	-infer_scan_structures	
drive_of	-wire_drive	-min
extract_ilm		-ilm_core -optimizable
filter	-dont_check_real_objects	
get_clocks	-exact -hierarchical	
get_clusters	-hierarchical	-flat -hier -of_objects
get_location		-rp_group
get_multibits	-exact -hierarchical	
get_path_groups	-exact	
get_pins		-leaf
get_placement_keepouts		-of_objects -within
get_scan_cells_of_chain		-test_mode
get_scan_chains		-test_mode
get_timing_paths		-path_type
get_wiring_keepouts		-of_objects -within
identify_clock_gating	-gated_elements -ungated_elements	-gated_element -reset -reset_only -ungated_element

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
insert_dft	-arch -background -dont_fix_constraint_violation s -host -ignore_compile_design_rule s -map_effort -no_scan -xterm	
list_test_modes	-existing -spec	
move_objects	-respect_mobility	-respect_rigidity
optimize_placement		-fix_drc -ignore_all_groups -worst_in_group
preview_dft	-bscan -no_scan	-bsd
propagate_constraints		-format
propagate_ilm		-parasitics
read_def	-design	-adjust_tracks -allow_physical_objects -lef_file_name
read_file		-ilm -rtl
read_parasitics	-format	-dont_write_to_db
read_pdef	-allow_physical_cells -allow_physical_ports -quiet -verbose	-allow_physical_objects

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
read_saif		-target_instance
read_test_protocol	-silent	-overwrite -section -test_mode -verbose
remove_clock_latency	-clock	
remove_dft_logic_usage	-test_modes	-test_mode
remove_dft_signal		-hookup_pin -port -test_mode -view
remove_port	-all	
remove_scan_specification	-all -bidirectionals -chain -link -segment -signal -test_mode -tristates	
remove_test_mode	-existing -spec -view	
remove_test_protocol		-design -test_mode
remove_wire_load_model	-cluster	
remove_wire_load_selection_group	-cluster	
report_annotated_delay	-min	

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
report_congestion	-coordinate	
report_constraint		-max_net_length
report_lib	-noise -noise_arcs	-yield
report_peak_noise	-explore_isolation	
report_port		-only_physical
report_region	-op_condition	-connection
report_saif		-annotated_flag
report_test	-assertions -bsd -clock -core_integration_configuratio n -dft_configuration -incremental -inst -nosplit -register_type -replacements	-autofix_configuration -bist -bist_config -dft -no_split -replacement -test_mode -testability_configuration
report_timing	-locations	
report_voltage_area	-op_condition	-connection
rotate_objects	-respect_mobility	-respect_rigidity
set_autofix_configuration	-async -async_fix -bus -clocks -data_is_clock -fix_async_with_scan_en -xprop	-exclude_elements -fix_data -fix_latch -include_elements -method -test_data -type

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
set_autofix_element	-async -clock	-control_signal -fix_data -fix_latch -method -test_data -type
set_bsd_compliance		-name -pattern
set_bsd_configuration	-flow -infer_instructions	-control_cell_max_fanout -integrate
set_bsd_instruction	-inst_enable -internal_scan -user_code_val	-capture_value -clock_cycles -signature -view
set_bsd_power_up_reset	-cell_name	
set_dft_clock_controller	-design	-design_name
set_dft_configuration	-autofix -bist -bscan -clock_controller -core_integration -core_wrapper -order -shadow_wrapper -testability -tester_checks	-boundary -bsd -control_points -fix_bidirectional -fix_bus -fix_clock -fix_reset -fix_set -fix_xpropagation -integration -logicbist -mbist -observe_points -scan -wrapper
set_dft_drc_configuration		-internal_pins

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
set_dft_insertion_configuration		-map_effort -preserve_design_name -route_scan_clock -route_scan_enable -route_scan_serial -synthesis_optimization -unscan
set_dft_logic_usage	-test_modes	-test_mode
set_dft_optimization_configuration	-def_out	
set_dft_signal	-hookup -sense	-active_state -freq_mult -hookup_pin -hookup_sense -internal_clocks -test_mode -type -view
set_disable_timing	-reset_loop_breaking_arcs	
set_dps_module_options		-no_opt
set_driving_cell		-none
set_macro_cell_bound_spot	-coordinate	-coordinates
set_mbist_configuration		-exclude_elements -include_elements
set_mbist_controller	-controller_names -ip_parameters -memory_cells	-instance -parameters -target
set_mpc_options		-dont_promote_layer
set_port_fanout_number	-max -min	

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
set_scan_configuration	-bidi_mode -dedicated_scan_ports -disable -existing_scan -external_tristates -insert_end_of_chain_lockup_latch -internal_tristates -longest_chain_length -methodology -minimize_hold_time_violation -multibit_segments -physical -prfile -prtool -rebalance -route -route_signals -scan_enable_per_domain -share_pins	-create_dedicated_scan_out_ports -domain_based_scan_enable -exclude_elements -insert_terminal_lockup -max_length -minimize_hold_time_violations -pipeline_fanout_limit -pipeline_scan_enable -preserve_multibit_segment -shared_scan_in -voltage_mixing
set_scan_element	-multibit	
set_scan_group		-serial_routed
set_scan_link		-test_mode
set_scan_path	-chain_length -clock	-class -exact_length -hookup -infer_dft_signals -input_wrapper_cells_only -ordered_elements -output_wrapper_cells_only -scan_data_in -scan_data_out -scan_enable -scan_master_clock -scan_slave_clock -view

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
set_scan_replacement	-remove	
set_testability_configuration	-control_points_per_scan_cell -max_control_points -max_observe_logic_area -max_observe_points -method -observe_points_per_scan_cell -power_saving_on -share_across_hierarchy -top_instance	-clock_signal -max_additional_logic_area -max_test_points -power_saving -test_points_per_scan_cell -type
set_wire_load_model	-cluster	
set_wire_load_selection_group	-cluster	
update_lib	-force	
write		-scenarios -xg_force_db
write_def	-default_sp_conn -tiehigh -tielow	-blockages -diode_pins -fills -fixed_cell -floating_metal_fill -gzip -lef_file_name -macro -notch_gap -pg_metal_fill -placed_cell -regions_groups -routed_net
write_file		-scenarios -xg_force_db

Table A-3 Modified Commands (Continued)

Command	Options in DB mode only	Options in XG mode only
write_lib	-mw_oc_type -mw_ref_lib	
write_pdef	-new_cells_only -no_attributes -no_hierarchy -v2 -v3	-no_legalize -unit
write_test	-cumulative -first -input -parallel -part_number -revision	-pattern_exec -test_bench_environment
write_test_model		-design
write_test_protocol		-design

Index