Copyright Notice for the Command-Line Editing Feature
© 1992, 1993 The Regents of the University of California. All rights reserved. This code is derived from software contributed to Berkeley by Christos Zoulas of Cornell University.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright Notice for the Line-Editing Library
© 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:
1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.
Creating RDL Net Shields ......................................................... 5-24

6. Managing Design Blocks
Exploring the Design Hierarchy ............................................... 6-3
Creating Module Boundaries .................................................. 6-3
Committing Design Blocks ..................................................... 6-6
Creating Block Placement Abstracts ........................................ 6-7
Moving the Origin of a Block .................................................. 6-9

7. Shaping Design Blocks and Macro Placement
Setting Macro Constraints ...................................................... 7-3
Creating Relative Placement Constraints for Macros and Macro Arrays ........................................ 7-5
Setting Macro Keepouts ........................................................ 7-7
Shaping Blocks ..................................................................... 7-8
Creating Block Shaping Constraints ....................................... 7-10
Creating the Block Grid ......................................................... 7-16
Creating the Initial Macro Placement ................................. 7-18

8. Performing Power Planning
Creating and Connecting Power Nets .................................. 8-5
Defining PG Via Masters ....................................................... 8-5
Creating Power and Ground Ring Patterns ................................ 8-6
Creating Power and Ground Mesh Patterns ............................ 8-7
Creating Power and Ground Macro Connections ...................... 8-8
Creating Power and Ground Standard Cell Rails ....................... 8-9
Creating Channel and Alignment Power Straps ....................... 8-10
Creating Complex Composite Patterns .................................. 8-10
Creating Center-Aligned Vias ................................................ 8-11
Creating Bridging Straps ...................................................... 8-13
Creating Via Bridging Straps .................................................. 8-16
Creating Tapering Straps ...................................................... 8-19
Creating a Checkerboard Via Pattern .............................................. 8-22
Defining Power Plan Regions ......................................................... 8-25
Setting the Power Plan Strategy ....................................................... 8-28
  Strategy Settings for the -pattern Option ...................................... 8-30
  Strategy Settings for the -blockage Option .................................... 8-30
  Strategy Settings for the -extension Option .................................... 8-31
Creating Via Rules Between Different Strategies ............................. 8-31
Instantiating the Power Plan ......................................................... 8-32
Pattern-Based Power Network Routing Example ............................... 8-33
Inserting and Connecting Power Switches ..................................... 8-35
  Setting Resistance Values for Power Switch Cells ......................... 8-38
Trimming the Power and Ground Mesh ............................................ 8-38
Checking Power Network Connectivity and DRC .............................. 8-40
  Checking Power Network Connectivity ......................................... 8-40
  Validating DRC in the Power Network ......................................... 8-42
Performing Distributed Power Network Routing .............................. 8-43
  Preparing for Distributed Power Network Routing ......................... 8-44
  Performing Distributed Power Network Routing ............................. 8-45

9. Performing Global Planning

Creating Bundles and Bundle Constraints ................................. 9-3
  Bundle Constraint Example ....................................................... 9-6
Creating Routing Corridors .......................................................... 9-7
Creating Global Routes Within Routing Corridors ......................... 9-9
Adding Repeater Cells ................................................................ 9-10
Pushing Down Repeater Cells ...................................................... 9-10

10. Performing Pin Assignment

Creating Global Pin Constraints .................................................... 10-4
Creating Individual Pin Constraints ............................................. 10-5
Reading Pin Constraints ............................................................. 10-6
1. Contents

Topological Map ................................................................. 10-7
Feedthrough Control ......................................................... 10-9
Physical Pin Constraints ..................................................... 10-10
Pin Spacing Control ........................................................... 10-12
Block Pin Constraints ......................................................... 10-13

Topological Constraint Examples ........................................ 10-14
Creating a Direct Connection Between Blocks with Topological Constraints .............................. 10-14
Specifying a Partial Path ...................................................... 10-15

Performing Global Routing for Pin Placement ......................... 10-15
Creating a Channel Congestion Map ...................................... 10-16
Placing Pins ........................................................................ 10-17
Checking Pin Placement ........................................................ 10-20
Writing Pin Constraints ......................................................... 10-30
Writing Out Feedthroughs Inserted During Design Planning ............... 10-31
Reporting Estimated Wire Length ......................................... 10-32
Moving Objects Between the Top and Block Level ....................... 10-33

11. Performing Timing Budgeting

Creating Block Timing Abstracts ............................................. 11-4
Performing Virtual In-Place Optimization ............................... 11-6
Applying Manual Budget Constraints .................................... 11-9
Updating Budget Information ............................................... 11-10
Writing Out Budget Information ............................................ 11-13
Preface

This preface includes the following sections:

- About This User Guide
- Customer Support
About This User Guide

The Synopsys IC Compiler II tool provides a complete netlist-to-GDSII design solution, which combines proprietary design planning, physical synthesis, clock tree synthesis, and routing for logical and physical design implementations throughout the design flow.

This guide describes the IC Compiler II implementation and integration flow. For more information about the IC Compiler II tool, see the following companion volumes:

- *IC Compiler II Library Preparation User Guide*
- *IC Compiler II Implementation User Guide*
- *IC Compiler II Data Model User Guide*
- *IC Compiler II Timing Analysis User Guide*
- *IC Compiler II Graphical User Interface User Guide*

Audience

This user guide is for design engineers who use the IC Compiler II tool to implement designs.

To use the IC Compiler II tool, you need to be skilled in physical design and synthesis and be familiar with the following:

- Physical design principles
- The Linux or UNIX operating system
- The tool command language (Tcl)

Related Publications

For additional information about the IC Compiler II tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- Design Compiler®
- IC Validator
- PrimeTime® Suite
Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the IC Compiler II Release Notes on the SolvNet site.

To see the IC Compiler II Release Notes,

1. Go to the SolvNet Download Center located at the following address:
   https://solvnet.synopsys.com/DownloadCenter
2. Select IC Compiler II, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Courier</strong></td>
<td>Indicates syntax, such as write_file.</td>
</tr>
<tr>
<td><strong>Courier italic</strong></td>
<td>Indicates a user-defined value in syntax, such as write_file design_list.</td>
</tr>
<tr>
<td><strong>Courier bold</strong></td>
<td>Indicates user input—text you type verbatim—in examples, such as prompt&gt; write_file top</td>
</tr>
<tr>
<td>[]</td>
<td>Denotes optional arguments in syntax, such as write_file [-format fmt]</td>
</tr>
<tr>
<td>...</td>
<td>Indicates that arguments can be repeated as many times as needed, such as pin1 pin2 ... pinN.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Ctrl+C</td>
<td>Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.</td>
</tr>
<tr>
<td>\</td>
<td>Indicates a continuation of a command line.</td>
</tr>
<tr>
<td>/</td>
<td>Indicates levels of directory structure.</td>
</tr>
<tr>
<td>Edit &gt; Copy</td>
<td>Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.</td>
</tr>
</tbody>
</table>
Customer Support

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

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

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 support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”

• Send an e-mail message to your local support center.
  o E-mail support_center@synopsys.com from within North America.
  o Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

• Telephone your local support center.
  o Call (800) 245-8005 from within North America.
  o Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
Introduction to Design Planning

The IC Compiler II tool is designed for efficient design planning, placement, routing, and analysis of very large designs. You can use the design planning and chip-level analysis capabilities to perform initial floorplanning and feasibility analysis.

For more information, see the following topics:

- Design Planning Overview
- Hierarchical Design Planning Flow
- Design Planning at Multiple Levels of Physical Hierarchy
Design Planning Overview

Design planning is an integral part of the RTL to GDSII design process. During design planning, you assess the feasibility of different implementation strategies early in the design flow. For large designs, design planning helps you to “divide and conquer” the implementation process by partitioning the design into smaller, more manageable pieces for more efficient processing.

Floorplanning is the process of partitioning logical blocks into physical blocks, sizing and placing the blocks, performing a floorplan-level placement of macros and standard cells, and creating a power plan. The goal of floorplanning is to increase the efficiency of downstream physical design steps to enable a robust, optimized design. To generate accurate budget estimates for the physical blocks, you generate timing estimates to guide timing budget allocation between the blocks and top-level cells in the design. Floorplanning can also be an iterative process that reshapes blocks, creates a new cell placement, reallocates timing budgets, and rechecks top-level timing until an optimal floorplan is created.

An effective floorplan helps to achieve timing closure by placing blocks to minimize critical path length and reduce routing congestion. The challenge is to create a floorplan with good area efficiency and a small footprint, while leaving sufficient area for routing.

Similarly, design planning supports pattern-based power planning, including low-power design techniques that can be used in multivoltage designs. Using pattern-based power planning, you can create different voltage areas within the design and define a power plan strategy for each voltage area. The tool creates the power and ground mesh based on your specification, and it connects the macros and standard cells to the mesh. You can quickly iterate on different power plan strategies to investigate different implementations and select the optimal power plan.

The IC Compiler II tool supports complete hierarchical design planning for both channeled and abutted layout styles. For more information, see Supported Types of Floorplans.

Design planning in the IC Compiler II tool provides the following features for developing hierarchical and flat designs, such as

- Multi-Level Physical Hierarchy Floorplanning
- Advanced abstraction management for processing large designs
- Guided floorplanning with a graphical Task Manager
- Data flow analysis for macro management and placement assistance
- Fast placement of macros and standard cells
- Pattern-based power planning
- Power network description using UPF
Hierarchical Design Planning Flow

The hierarchical design planning flow provides an efficient approach for managing large designs. By dividing the design into multiple blocks, different design teams can work on different blocks in parallel, from RTL through physical implementation. Working with smaller blocks and using multiply instantiated blocks can reduce overall runtime.

Consider using a hierarchical methodology in the following scenarios:

- The design is large, complex, and requires excessive computing resources to process the design in a flat form.
- You anticipate problems that might delay the delivery of some blocks and might cause the schedule to slip. A robust hierarchical methodology accommodates late design changes to individual blocks while maintaining minimal disruption to the design schedule.
- The design contains hard intellectual property (IP) macros such as RAMs, or the design was previously implemented and can be converted and reused.

After the initial design netlist is generated in Design Compiler topographical mode, you can use the hierarchical methodology for design planning in the IC Compiler II tool. Design planning is performed during the first stage of the hierarchical flow to partition the design into blocks, generate hierarchical physical design constraints, and allocate top-level timing budgets to lower-level physical blocks.
The flow to implement a hierarchical design plan is shown in Figure 1-1.

**Figure 1-1 Hierarchical Design Planning Flow**

- Read design data (read_verilog_outline)
- Initialize design (initialize_floorplan)
- Place I/Os (place_io)
- Explore hierarchy and commit blocks (commit_block)
- Create block placement abstracts (create_abstract)
- Read top- and block-level UPF (load_upf)
- Shape blocks (shape_blocks)
- Create initial placement (create_placement)
- Plan buses and repeaters
- Plan power network
- Plan hierarchical interfaces (place_pins)
- Create block timing abstract (create_abstract)
- Budget timing (estimate_timing)
- Write design data

**See Also**
- Design Partitioning
- Deciding on the Physical Partitions

**Design Partitioning**

After reading in the netlist and initializing the floorplan, you can determine the physical partitioning for your hierarchical design project. When deciding on the physical partitions, consider the following factors:
• Size
  Partition your design with blocks of similar size. Small blocks should be grouped and large blocks should be divided when appropriate.

• Function
  Partition your design using its functional units for verification and simulation purposes. Consider top-level connectivity and minimal block pin counts to avoid congestion and timing issues.

• Floorplan style
  Different floorplan styles require different physical hierarchies to support them. An abutted floorplan style has no top-level logic and a channeled floorplan has either a small or large amount of top-level logic.

• Common hierarchy with Design Compiler topographical mode
  To exchange SCANDEF information at the block level and the top level, the physical hierarchy used in Design Compiler topographical mode must also be used in the IC Compiler II tool.

Deciding on the Physical Partitions

The IC Compiler II tool provides the following features to help you decide on the physical partitions:

• Using the Hierarchy Browser
  You can use the hierarchy browser to navigate through the design hierarchy, to examine the logic design hierarchy, and to display information about the hierarchical cells and logic blocks in your design. You can select the hierarchical cells, leaf cells, or other objects in layout or schematic views. The viewer provides a tree browser to help in understanding the design hierarchy, and information about the blocks in the design such as the utilization, number of standard cells, and so on.
  For more information, see Exploring the Design Hierarchy.

• Committing Blocks
  After you decide on an initial partitioning, you can commit the blocks. Blocks are committed early in the floorplanning flow, and abstract views are used to generate an initial floorplan and timing budgets for the design.
  For more information about committing blocks, see Committing Design Blocks.
Design Planning at Multiple Levels of Physical Hierarchy

Large, complex SoC designs require hierarchical layout methodologies capable of managing multiple levels of physical hierarchy at the same time. Many traditional design tools -- including physical planning, place and route, and other tools -- are limited to two levels of physical hierarchy: top and block. The IC Compiler II tool provides comprehensive support for designs with multiple levels of physical hierarchy, resulting in shorter time to results, better QoR, and higher productivity for physical design teams. Use the `set_hierarchy_options` command to enable or disable specific blocks and design levels of hierarchy for planning.

IC Compiler II provides support in several areas to accommodate designs with multiple levels of physical hierarchy:

**Data Model**

The data model in the IC Compiler II tool has built-in support for multiple levels of physical hierarchy. Native physical hierarchy support provides significant advantages for multi-level physical hierarchy planning and implementation. When performing block shaping, placement, routing, timing, and other steps, the tool can quickly access the specific data relative to physical hierarchy needed to perform the function.

**Block Shaping**

In a complex design with multiple levels of physical hierarchy, the block shaper needs to know the target area for each sub-chip, the aspect ratio constraints required by hard macro children, and any interconnect that exists at the sibling-to-sibling, parent-to-child, and child-to parent interfaces. For multi-voltage designs, the block shaper needs the target locations for voltage areas. These requirements add additional constraints for the shaper to manage. For multi-level physical hierarchy planning, block shaping constraints on lower level sub-chips must be propagated to the top level; these constraints take the form of block shaping constraints on parent sub-chips. To improve performance, the shaper does not need the full netlist content that exists within each sub-chip or block.

The IC Compiler II data model provides block shaping with the specific data required to accomplish these goals. For multi-voltage designs, the tool reads UPF and saves the power intent at the sub-chip level. The tool retrieves data from the data model to calculate targets based on natural design utilization or retrieves user-defined attributes that specify design targets.

**Cell and Macro Placement**

After block shaping, the cell and macro placement function sees a global view of the interconnect paths and data flow at the physical hierarchy boundaries and connectivity to macro cells. With this information, the tool places macros for each sub-chip at each level of hierarchy. Because the tool understands the relative location requirements of interconnect
paths at the boundaries at all levels, sufficient resources at the adjacent sub-chip edges are reserved to accommodate interconnect paths. The placer anticipates the needs of hierarchical pin placement and places macros where interconnect paths do not require significant buffering to drive signals across macros.

The placer models the external environment at the boundaries of both child and parent sub-chips by considering sub-chip shapes, locations, and the global macro placements. Using this information, the placer creates cell placement jobs for each sub-chip at each level of hierarchy. By delegating sub-chip placement across multiple processes, the tool minimizes turnaround time while maximizing the use of compute resources.

**Power Planning**

For power planning, the IC Compiler II tool provides an innovative pattern-based methodology. Patterns describing construction rules -- widths, layers, and pitches required to form rings and meshes -- are applied to different areas of the floorplan such as voltage areas, groups of macros, and so on. Strategies associate single patterns or multiple patterns with areas. Given these strategy definitions, the IC Compiler II tool characterizes the power plan and automatically generates definitions of strategies for sub-chips at all levels. A complete power plan is generated in a distributed manner. Because the characterized strategies are written in terms of objects at each sub-chip level, power plans can be easily re-created to accommodate floorplan changes at any level.

**Pin Placement**

With block shapes formed, macros placed, and power routed, pin placement retrieves interface data from all levels and invokes the global router to determine the optimal location to place hierarchical pins. The global router recognizes physical boundaries at all levels to ensure efficient use of resources at hierarchical pin interfaces. Pins are aligned across multiple levels when possible. Like all IC Compiler II operations, the global router comprehends multiply instantiated blocks (MIBs) and creates routes compliant with each MIB instantiation. To place pins for MIBs, the pin placement algorithm determines the best pin placement that works for all instances, ensuring that the pin placement on each instance is identical. Additionally, pin placement creates feedthroughs for all sub-chips, including MIBs, throughout the hierarchy. The global router creates feedthroughs across MIBs, determines feedthrough reuse, and connects unused feedthroughs to power or ground as required.

**Timing Budgeting**

The IC Compiler II tool estimates the timing at hierarchical interfaces and creates timing budgets for sub-chips. The timing budgeter in IC Compiler II creates timing constraints for all child interface pins within the full chip, the parent and child interfaces for mid-level sub-chips and the primary pins at lowest level sub-chips. The entire design can proceed with placement and optimization concurrently and in a distributed manner.
To examine critical timing paths in the layout or perform other design planning tasks, you can interactively view, analyze, and manually edit any level of the design in a full-chip context. You can choose to view top-level only or multiple levels of hierarchy. When viewing multiple levels, interactive routing is performed as if the design is flat. At completion, routes are pushed into children and hierarchical pins are automatically added.
Splitting Constraints

To generate timing budgets, chip-level Synopsys Design Constraints (SDC) and UPF files must be partitioned into top-level and block-level files. The IC Compiler II tool uses the top-level constraints to constrain the top-level logic when abstract views of blocks are used.

For more information, see the following topics on splitting constraints:

- Split Constraints Flow
- Split Constraints Output Files
- Split Constraints Example 1
- Split Constraints Example 2
- Running Tasks in Parallel
- Monitoring Distributed Tasks
Split Constraints Flow

If you have only chip-level constraints for your design, you can use the `split_constraints` command to partition the full-chip SDC and UPF constraints and create separate top-level and block-level constraint files. After running `split_constraints`, the top-level constraint file contain the top-level UPF and all top-level and top-to-block boundary timing, without block-level internal constraints. The block-level constraint files are used during block-level optimization.

The block-level constraint files generated by the `split_constraints` command contain internal constraints for the block and should be applied only one time. You must apply either manually generated constraints or constraints generated by the `split_constraints` command before applying timing budgets.

Note that if you already have block-level SDC and UPF constraints for your design, you can ignore this section and continue without running the `split_constraints` command.

To split the chip-level SDC and UPF constraints,

1. Read in the full Verilog netlist.
   ```
   icc2_shell> read_verilog -top leon3mp leonmp3.v
   ```
   Note that the split constraints flow requires that you read the full Verilog netlist with the `read_verilog` command. You cannot use the `read_verilog_outline` command for this step.

2. Load and commit the UPF constraint file for the entire design.
   ```
   icc2_shell> load_upf leon3mp.upf
   icc2_shell> commit_upf
   ```
   The `load_upf` command reads in the UPF constraints and the `commit_upf` command applies the UPF constraints to the design. The `commit_upf` command also tries to resolve any conflicts between the power intent specified in the UPF file and the actual power and ground nets in the design.

3. Setup the timing environment for the design.
   For a design with multiple modes and corners, you must use one or more of the following commands:
   ```
   create_corner
   create_mode
   read_sdc
   set_corner_status
   set_parasitic_parameters
   set_process_number
   set_temperature
   set_voltage
   ```
If your design uses UPF to describe the power network, you must use the set_voltage command for all supply net groups defined in the UPF file.

4. Reset any previous budget constraints and specify the blocks for which to write out constraints.

   icc2_shell> set_budget_options -reset
   icc2_shell> set_budget_options -add_blocks {u0_0 u0_1 u0_2 u0_3}

   If your design contains multiply instantiated blocks, all instances must be specified with the set_budget_options command.

5. Split the timing constraints and UPF file into separate files for each block.

   icc2_shell> split_constraints

   If your design contains multiple levels of physical hierarchy, specify the intermediate levels of hierarchy with the -design_subblocks option as follows:

   icc2_shell> split_constraints -design_subblocks {block_1 block_2}

   The split_constraints command supports several options to control how the constraints are split. To split only the SDC file or the UPF file, use the -sdc_only option or -upf_only options. To write out constraint files only for certain modes, use the -modes option. To write out constraint files only for certain corners, use the -corners option. To compress the output constraint files, use the -compress gzip option. To specify the directory into which to write the output, use the -output directory_name option.

Split Constraints Output Files

Figure 2-1 shows the directory structure created by the split_constraints command for the leon3mp design.

**Figure 2-1** Constraint Files After split_constraints

In this example, leon3mp is the top-level design and leon3s, leon3s_2 and leon3s_3 are the modules. The split_constraints command creates the following files for the top-level
design and for each module that is specified with the `set_budget_options -add_blocks` command:

- **top.tcl**: Provides corner and mode definitions, and sources all other files in the correct order. Source only this file to apply the split constraints for the specified block.

- **design.tcl**: Provides design-wide constraints with mode and corner associations. Contains `set_corner_status` commands to configure analysis settings for mode and corner combinations.

- **mode_modename.tcl**: Provides mode-specific constraints for clocks, exceptions, latencies, and so on. The tool creates one Tcl file for each defined mode.

- **corner_cornername.tcl**: Provides corner-specific constraints for derating factors, parasitic definitions, and PVT values. The tool creates one Tcl file for each defined corner.

- **top.upf**: UPF constraints for the block.

The `split_constraints` command also creates the constraint mapping file, mapfile, which lists the SDC and UPF constraint files for each block. The content of the mapfile file for this example is as follows:

```
leon3s   SDC   leon3s/top.tcl
leon3s   UPF   leon3s/top.upf
leon3s_2 SDC   leon3s_2/top.tcl
leon3s_2 UPF   leon3s_2/top.upf
leon3s_3 SDC   leon3s_3/top.tcl
leon3s_3 UPF   leon3s_3/top.upf
```

---

**Split Constraints Example 1**

The design for Example 1 is shown in Figure 2-2.

**Figure 2-2  Example 1 Design**

This design example uses the following chip-level SDC constraints file.
Example 2-1  Chip-Level SDC Constraints File

create_clock -period 1 [get_ports clk]
set_input_delay 0.5 [get_ports in1]
set_multicycle_path 2 -from ff1/clk -through i1/a1/y

The `split_constraints` command partitions the chip-level SDC constraints file into the following top-level and block-level `mode_modename.tcl` constraint files.

Example 2-2  Top-Level Constraint File after `split_constraints`

# Top-level constraints
create_clock -period 1 [get_ports clk]
set_input_delay 0.5 [get_ports in1]
set_multicycle_path 2 -from ff1/clk -through i1/a1/y

Example 2-3  i1 Constraint File after `split_constraints`

# Block-level constraints
create_clock -add -name "clk" -period 1 [get_ports clk]
clock -name virtual_clk -period 1
set_multicycle_path 2 -from virtual_clk -through [get_ports in] -through a1/y

Split Constraints Example 2

The design for Example 2 is shown in Figure 2-3.

Figure 2-3  Example 2 Design

This design example uses the following top-level SDC constraints file.

Example 2-4  Chip-Level SDC Constraints File

create_clock -period 1 [get_ports clk]
set_input_delay -clock [get_clocks clk] 0.5 [get_ports din]
set_output_delay -clock [get_clocks clk] 0.5 [get_ports dout]

The `split_constraints` command partitions the chip-level SDC constraints file into the following top-level and block-level `mode_modename.tcl` constraint files.
Example 2-5  Top-Level Constraint File After split_constraints

```tcl
# Top-level constraints
create_clock -name clk -period 1 [get_ports {clk}]
set_input_delay -clock [get_clocks {clk}] 0.5 [get_ports {din}]
set_output_delay -clock [get_clocks {clk}] 0.5 [get_ports {dout}]
```

Example 2-6  Block i1 Constraint File after split_constraints

```tcl
# Block-level constraints
create_clock -name "clk" -period 1 [get_ports {clk}]
set_false_path -from [get_pins {ff2/CK}] -to [get_pins {ff3/D}]
```

Running Tasks in Parallel

To efficiently run the same task on several blocks in your design, you can use the `run_block_script` command to enable distributed processing and perform the tasks in parallel. The `run_block_script` command accepts a Tcl script and applies the commands in the script to the blocks you specify.

For example, to generate the timing abstracts in parallel for the blocks in your design,

1. Write the script to run on the specified blocks in the design.

   The following example performs the steps necessary to generate timing abstracts for the blocks in the design. The script opens the block, applies constraints, inserts feedthroughs and creates the timing abstract. You can add additional commands to the script as needed to perform other operations.

   ```tcl
   Example 2-7  Example of run_block_script

   Example 2-8  Timing Abstract Script
   ```

   Note:

   If your design contains multiple levels of physical hierarchy, the `create_abstract` command converts only blocks at the lowest level of hierarchy into abstract views; intermediate levels of hierarchy are kept as design views.
# Perform design setup
source ../scripts/setup.tcl

# Open the block
open_block $block_libfilename:$block_refname.design

# Merge previous changes to the abstract into the top-level
merge_abstract

# Source block-level constraints
source ./split/${block_refname}/top.tcl

# Insert feedthroughs
add_feedthrough_buffers

# Create the block-level timing abstract
create_abstract -estimate_timing

# Save the block
save_lib
close_lib

The **run_block_script** command sets four variables to help you simplify and
generalize your script: **block_refname**, **block_libfilename**, **top_libname**, and **work_dir**. The **block_refname** variable contains the reference name of the block being processed. The **block_libfilename** variable contains the full path name of the library for the block being processed. **top_libname** contains the name of the library for the top-level design that contains the blocks being processed. **work_dir** is a subdirectory named **block_refname** in the directory specified by the **-work_dir** option.

2. Run the **run_block_script** command to process each block in parallel by using the script created in step 1.

The following example specifies the settings for distributed processing with the **set_host_options** command and runs the **create_block_timing_abstract.tcl** block script on blocks u1, u2, u3 and u4 in parallel by using the Load Sharing Facility (LSF). The command writes data and log files to the **create_block_timing_abstract** directory.

```
icc2_shell> set_host_options -name lsf -submit_command "bsub -q amd64"
icc2_shell> run_block_script \
    -script create_block_timing_abstract.tcl -cells {u1 u2 u3 u4} \ 
    -host_options lsf4 -work_dir create_block_timing_abstract
```

To control the order in which blocks are processed, use the **-run_order** option with the **top_down** or **bottom-up** argument. When you specify **-run_order top-down**, the tool delays processing for a child block until all parent blocks for the child block are processed. When you specify **-run_order bottom-up**, the tool begins by processing the child blocks, then processes the parent blocks. By default, blocks are processed in a bottom-up order.
Monitoring Distributed Tasks

The IC Compiler II tool supports a Distributed Processing Manager to help you monitor the progress of distributed tasks as they run. The Distributed Processing Manager communicates with distributed commands such as `create_abstract` to provide up-to-date status. When a task completes, the GUI updates to show that the task is finished.

To use the Distributed Processing Manager to monitor running jobs,

1. Start the Distributed Processing Manager with the `run_monitor_gui` command.

   ```shell
   icc2_shell> run_monitor_gui
   10948
   ```

   The command returns the process id for the dpmanager task created by the command.

2. Start the distributed task.

   ```shell
   icc2_shell> create_abstract -placement \
     -host_options block_script -all_blocks
   Submitting job for block leon3s ...
   Submitting job for block leon3s_2 ...
   Submitting job for block leon3s_3 ...
   All tasks created.
   ```
3. Monitor the running task in the Distributed Processing Manager window as shown in Figure 2-4.

**Figure 2-4  Distributed Processing Manager**

When all tasks are complete, the Distributed Processing Manager updates the Status column to COMPLETED for all job/Task IDs, and updates the status in the Job History window from not done to done.

You can open a maximum of two Distributed Processing Manager windows. To open a second Distributed Processing Manager window, use the `run_monitor_gui` command a second time. To kill any open Distributed Processing Manager windows and restart a single window, use the `run_monitor_gui -kill` command.
The Distributed Processing Manager monitors the progress of the following commands, if they are running in distributed mode:

- `create_abstract -all_blocks | -blocks`
- `create_placement -floorplan`
- `estimate_timing`
- `merge_abstract`
- `route_global -floorplan -virtual_flat`
- `shape_blocks`
Creating a Floorplan

This section describes how to create and refine a floorplan from an existing netlist or floorplan description. The floorplan describes the size of the core; the shape and placement of standard cell rows and routing channels; standard cell placement constraints; and the placement of peripheral I/O, power, ground, corner, and filler pad cells.

For more information, see the following topics:

• Supported Types of Floorplans
• Reading the Verilog Netlist
• Creating an Initial Floorplan
• Creating an L-, T-, or U-Shaped Floorplan
• Creating a Complex Rectilinear Floorplan
• Adjusting the Floorplan Aspect Ratio
• Flipping the First Row in the Floorplan
• Updating the Floorplanning Without Disturbing Specified Object Types
• Validating the FinFET Grid
• Reporting Floorplan Information
• Reading and Writing the Floorplan Information
Supported Types of Floorplans

The IC Compiler II tool supports different floorplan styles to meet the requirements of your design. The channeled, abutted, and narrow-channel floorplan styles are described in the following sections.

- Channeled Floorplans
- Abutted Floorplans
- Narrow-Channel Floorplans

Channeled Floorplans

Channeled floorplans contain spacing between blocks for the placement of top-level macro cells, as shown in the floorplan example in Figure 3-1. The spacing enables the tool to place standard cells between blocks.

Figure 3-1 Channeled Floorplan

Abutted Floorplans

In the abutted floorplan style, blocks are touching and the tool does not allocate space for macro cell placement between blocks. Designs with abutted floorplans do not require top-level optimization, as all logic is pushed down into the blocks. However, abutted floorplans might require over-the-block routing to meet design requirements. This floorplan style also requires more attention to clock planning and feedthrough management. Routing congestion might also become an issue for abutted floorplan designs. Due to the difficulties in implementing an abutted floorplan, the narrow-channel floorplan style is often used to minimize the spacing between blocks.
An example of an abutted floorplan is shown in Figure 3-2.

Figure 3-2  Abutted Floorplan

Narrow-Channel Floorplans

The narrow-channel floorplan style provides a balance between the channeled floorplan and abutted floorplan styles. You can abut certain blocks in the design where top-level cells are not needed. In other areas, you can reserve a channel between certain blocks for top-level clock routing or other special purpose cells. An example of a narrow-channel floorplan is shown in Figure 3-3.

Figure 3-3  Narrow-Channel Floorplan
Reading the Verilog Netlist

To read the Verilog netlist and begin floorplanning,

1. Create the library to contain the design by using the `create_lib` command. If the library already exists, use the `open_lib` command to open the library.

   ```
   icc2_shell> set libs [glob -directory ${lib_dir} *.ndm]
   icc2_shell> create_lib -technology technology_file.tf \
   -ref_libs $libs lib/design.ndm
   ```

2. Read the Verilog netlist with the `read_verilog_outline` command and specify the appropriate options.

   ```
   icc2_shell> read_verilog_outline -top leon3mp leon3mp_io_fc.v
   ```

The Verilog netlist outline reader in the IC Compiler II tool efficiently processes very large designs. To conserve resources and minimize turnaround time, the netlist reader creates an outline view of the blocks in the different levels of design hierarchy. The outline design contains hierarchy, but no leaf cells or nets.

The `read_verilog_outline` command does not partition the design. To partition the design, you commit the design blocks with the `commit_block` command and expand the blocks to their full netlist representations with the `expand_outline` command. For more information on committing blocks, see “Committing Design Blocks” on page 6-6.

You can also process a design in the full netlist flow by using the flat netlist reader. For information about reading a design with the flat netlist reader, see the `read_verilog` command.

See Also

• IC Compiler II Library Preparation User Guide

Creating an Initial Floorplan

After reading the Verilog netlist, use the `initialize_floorplan` command (or choose Floorplan Preparation > Floorplan Initialization from the Task Assistant in the GUI) to define a floorplan boundary based on your design criteria. You can create a floorplan by specifying the length ratio between the floorplan edges, the length of the core or die edges, the aspect ratio, or the exact width and height. This step uses the outline view for the top-level design.

To create the floorplan,

1. Use the `initialize_floorplan` command with the appropriate options to create the initial floorplan.
2. If needed, refine the floorplan by running the `initialize_floorplan` command again with different settings for the utilization ratio, side lengths, core-to-boundary spacing, and other settings.

   The `initialize_floorplan` command creates the floorplan according to the options you specify. The floorplan is shown in *Figure 3-4*.

   ![Floorplan With Core Offset of 100](image)

---

### Creating an L-, T-, or U-Shaped Floorplan

To create an L-, T-, or U-shaped floorplan,

1. Use the `initialize_floorplan` command with the `shape` option to create the floorplan. Supported shapes are R (rectangular), L, T, and U. The following example creates an L-shaped floorplan as shown in *Figure 3-5*.

   ```
   icc2_shell> initialize_floorplan -core_utilization 0.7 -shape L -orientation N -side_ratio {1 1 1} -core_offset {100.0} -flip_first_row true -coincident_boundary true
   ```

   2. Adjust the length of the sides if needed by changing the arguments to the `-side_ratio` or `-side_length` options and reissue the `initialize_floorplan` command.

   To further change the shape of the floorplan, you must set the dimensions of the floorplan boundary by specifying the `-side_ratio {side_a side_b ...}` option or the `-side_length {side_a side_b ...}` option. See *Figure 3-6* for a cross reference between the position of the number in the argument and the edge on the floorplan shape. This information is also available in the Floorplan Task Manager.
Chapter 3: Creating a Floorplan

Creating a Complex Rectilinear Floorplan

Create a complex floorplan shape by using the initialize_floorplan command with the -boundary option. Specify the -control_type die or -control_type core option of the -boundary option to specify the coordinates of the die or the core.

The following example creates an H-shaped floorplan by setting the coordinates of the floorplan die.

```plaintext
icc2_shell> initialize_floorplan -core_utilization 0.7 -shape U \ 
   -orientation N -side_ratio {2 1 1 1 1 2} -core_offset 100 \ 
   -coincident_boundary true
```
Adjusting the Floorplan Aspect Ratio

To create a floorplan with different side lengths, specify an aspect ratio with the \texttt{-side\_ratio} option. To set the aspect ratio of the floorplan,

1. Determine the ratio between the width and height of the floorplan.
2. Use the \texttt{initialize\_floorplan} command with the \texttt{-side\_ratio} option to create a floorplan with the specified width-to-height ratio.

\begin{verbatim}
icc2_shell> initialize_floorplan -control_type die \  
   -core_offset 100 \  
   -boundary { {0 0} {0 3000} {1000 3000} {1000 2000} \ 
               {2000 2000} {2000 3000} {3000 3000} {3000 0} \ 
               {2000 0} {2000 1000} {1000 1000} {1000 0} }
\end{verbatim}

Figure 3-7 shows a floorplan created by using the previous command.

Figure 3-7 Complex Rectilinear Floorplan
Flipping the First Row in the Floorplan

To avoid flipping the first row in the floorplan, specify the `-flip_first_row false` option as shown in the following example.

```
icc2_shell> initialize_floorplan -flip_first_row false
```

To create these floorplans by using the Floorplan Task Assistant in the GUI,

1. Open the Floorplan Task Assistant by selecting Task > Task Assistant in the GUI and selecting the Floorplan Initialization panel under Floorplan Preparation.
2. Select Core side ratio and enter a width/height value in the Aspect ratio box.
3. Click Preview to display a wireframe preview of the floorplan.
4. Click Apply to create the floorplan.

To create the floorplans shown in Figure 3-8, enter 3.0 in the Aspect ratio box to create the floorplan on the left, or enter 0.33 to create the floorplan on the right.
In Figure 3-9, the layout image on the left shows the layout created by using the `-flip_first_row true` option, and the layout image on the right shows the floorplan created by using the `-flip_first_row false` option. By default, the `initialize_floorplan` command flips the first row.

**Figure 3-9  Floorplan With First Row Flipped and Not Flipped**

To create an offset between the boundary and the core, specify the `-core_offset` option with the `initialize_floorplan` command as shown in the following example.

```
icc2_shell> initialize_floorplan -core_utilization 0.8 \
   -core_offset {50 100}
```

**Figure 3-10** shows a zoomed-in view of a floorplan created by using the `initialize_floorplan` command with the `-core_offset` option. This option sets the distance between the floorplan core and the floorplan boundary. The `-core_offset` option accepts a list of offsets, where each offset corresponds to an edge of the floorplan. See the `initialize_floorplan` man page for information about mapping the argument position to an edge of the floorplan.
Updating the Floorplanning Without Disturbing Specified Object Types

After creating the initial floorplan with the initialize_floorplan command, repeat the command with different options to experiment with different floorplan shapes and characteristics. To retain the placement or shape for specified object types when changing the floorplan, use one or more of the -keep_* options.

- **-keep_boundary**: Retain the floorplan boundary when updating the floorplan.
- **-keep_pg_route**: Retain power and ground routes when updating the floorplan.
- **-keep_detail_route**: Retain all routes except power and ground routes when updating the floorplan.
- **-keep_placement {block}**: Retain block placement when updating the floorplan.
- **-keep_placement {io}**: Retain terminal and pad placement when updating the floorplan.
- **-keep_placement {macro}**: Retain macro placement when updating the floorplan.
• **-keep_placement {std_cell}**: Keep standard cell placement when updating the floorplan.

• **-keep_placement {physical_only}**: Keep the placement for physical only cells when updating the floorplan; the cells can be identified with the following commands:

  ```
  icc2_shell> get_cells -filter "design_type==physical_only"
  icc2_shell> get_cells -filter "design_type==fill"
  ```

• **-keep_placement {all}**: Retain terminal, pad, macro, block, standard cell, and physical-only objects when updating the floorplan options.

---

### Validating the FinFET Grid

The IC Compiler II tool supports a FinFET grid to guide the placement of library cells that contain FinFET devices. The x- and y-spacing and offset values for the FinFET grid are specified in the technology file.

To report information about the FinFET grid, use the `report_grids -type finfet` command as follows:

```
icc2_shell> report_grids -type finfet
FinFET Grid               Description
---------------------------------
Top Design                leon3mp.ndm:leon3mp.outline
FinFET Grid defined       true
FinFET Grid X Pitch       0.152
FinFET Grid Y Pitch       1.672
FinFET Grid X Offset      0
FinFET Grid Y Offset      0
```

To check for placement or boundary violations with respect to the FinFET grid, use the `check_finfet_grid` command as follows:

```
icc2_shell> check_finfet_grid
... Reporting violations in block leon3mp ...
Error: boundary point (1425.304 1810.933) of hard macro 'u0_0/pd/rf0_x2' is not on FinFET grid. (DPCHK-002)
Error: boundary point (1425.304 1927.973) of hard macro 'u0_0/pd/rf0_x1' is not on FinFET grid. (DPCHK-002)
...```

To check specific object types, specify only those objects to the `check_finfet_grid` command as follows:
icc2 shell> `check_finfet_grid -objects [get_cells -hierarchical \
-filter "design_type==pad"]`
Reporting violations in block leon3mp ...
Error: boundary point (-40 0) of io pad 'u_p/errorn_pad_P1'
is not on FinFET grid. (DPCHK-002)
Error: boundary point (-40 0) of io pad 'u_p/pllref_pad_P1'
is not on FinFET grid. (DPCHK-002)
...

The following commands are aware of the FinFET grid in IC Compiler II:

- create_blackbox
- create_grid
- create_io_guide
- create_io_ring
- create_macro_array
- create_placement
- create_power_switch_array
- create_power_switch_ring
- create_site_array
- create_site_row
- initialize_floorplan
- place_io
- report_grids
- set_block_grid_references
- set_grid
• set_signal_io_constraints
• shape_blocks

---

**Reporting Floorplan Information**

Use the `report_design -floorplan` or `report_design -all` command to report the following information for the floorplan:

- Area: Core and chip area.
- Site row: Name, width, height, number of rows, number of tiles, and area for each site row.
- Blockages: Count and area for each blockage type, including hard placement blockages, soft placement blockages, hard macro blockages, partial placement blockages, routing blockages, category blockages, reserved placement group blockages, shaping blockages, routing for design rule blockages, placement blockages which only allow buffers, and no-register blockages for each blockage type.
- Power domains: Power domain name, voltage area name, and primary power and ground nets for each power domain.
- Voltage areas: Name, number of shapes, area, target utilization, and bounding box coordinates for each voltage area.
- Group bounds: Number of dimensionless group bounds.
- Move bounds: Name, area, utilization, and bounding box for each hard, soft, and exclusive move bound.
- Route guide: Count and area for each extra detour region, route guide over in-chip overlay cell layers, river routing guide, all double vias in the route guides, access preference route guide, default route guide, switched direction only route guide.
- Multibit registers: Number of multibit registers.
- Reserved placement groups: Number of top-level reserved placement groups, number of reserved placement cells, reserved placement cell area, and reserved placement cell area percentage for each reserved placement group.
- Layers: layer name, direction, pitch, default width, minimum width, minimum spacing, and minimum spacing between the same nets for each layer.
Reading and Writing the Floorplan Information

After you create the floorplan, you can save a Tcl script that describes the floorplan by using the write_floorplan command, or save a DEF file by using the write_def command.

- Reading DEF Files
- Writing the Floorplan and DEF Files

Reading DEF Files

To read a DEF file, use the read_def command. The following command reads the leon3.def DEF file.

```
icc2_shell> read_def leon3.def
```

To analyze the syntax of the input DEF files before annotating the objects on the design, specify the -syntax_only option.

```
icc2_shell> read_def -syntax_only design_name.def
```

Writing the Floorplan and DEF Files

To write the current floorplan, use the write_floorplan command as shown in the following example.

```
icc2_shell> write_floorplan
```

The write_floorplan command creates a directory named floorplan and writes out the floorplan.def and floorplan.tcl files to the directory. The floorplan.def file contains the physical layout, connectivity, and design constraint information for the design. The floorplan.tcl file contains additional design intent that cannot be described in the DEF file, such as hard macro keepout margins. In addition, the floorplan.tcl file contains commands to initialize the design and read the floorplan.def file. You can control the information written to the files by using command options.

To create only the DEF file, use the write_def command. For example, to create a compressed DEF file, specify the write_def command with the -compress option.

```
icc2_shell> write_def -compress gzip design.def
```
Handling Black Boxes

The IC Compiler II tool supports black boxes for modules in the design planning flow. A black box is a module that has no netlist, has only a partial netlist, or is not bound to a higher level module. Use black boxes to continue design planning without requiring a complete netlist for all modules. The tool supports block boxes for both physical and timing purposes, and for all stages of design planning. Note that only physical hierarchies can be black boxes in IC Compiler II.

The flow to create black box physical models and timing models is shown in Figure 4-1.

*Figure 4-1  Black Box Flow*

```
Identify potential black boxes  
(get_attribute)

Create black box references  
(create_blackbox)

Create black box timing models  
(create_blackbox_*; set_blackbox_*)

Commit the black box timing models  
(commit_blackbox_timing)
```
For more details, see the following topics:

- Identifying Potential Black Boxes
- Creating Black Box References
- Creating a Black Box Timing Model
- Black Box Timing Example
Identifying Potential Black Boxes

After reading in the netlist, use design attributes to identify empty, partial, or missing modules in the design. For each module in the hierarchy, the tool saves the number of standard cells and the number of macros in the module. If the module contains no standard cells or macros, the module might be a black box. Use the `hierarchy_std_cell_count` and `hierarchy_hard_macro_count` attributes to identify modules with no standard cells and no macro cells as follows.

```shell
icc2_shell> get_cells -hierarchical * \n    -filter {hierarchy_std_cell_count==0 && hierarchy_hard_macro_count==0} {u0_3}
```

If the netlist references a module that is not defined in the netlist or in a library, the tool sets the `is_unbound` attribute to true for the missing module. Use the following command to identify missing modules in the design:

```shell
icc2_shell> get_cells -hierarchical * -filter "is_unbound==true" {u0_3}
```

Several other attributes can be used to identify black boxes. The following list contains design attributes that can be used to find black boxes:

- `has_timing_model`
- `hierarchy_hard_macro_count`
- `hierarchy_has_shadow_netlist`
- `hierarchy_has_shadow_netlist_only`
- `hierarchy_pad_cell_count`
- `hierarchy_physical_only_cell_count`
- `hierarchy_std_cell_count`
- `hierarchy_switch_cell_count`
- `is_unbound`
- `target_boundary_area`

Creating Black Box References

After identifying the modules which to treat as black boxes, use the `create_blackbox` command to create a black box physical hierarchy for the cell. The command also updates the reference for the specified cell. Note that you can convert empty, partial, or missing modules to black boxes after reading in the netlist, or after committing the blocks.
The following example creates a black box in the current library for instance u0_3 and sets a target boundary area of 2000000. The layout window after creating the black box is shown in Figure 4-2:

```
icc2_shell> create_blackbox -target_boundary_area 2000000 u0_3
```

**Figure 4-2  Black Box in the GUI**

To find black boxes in the design created with the `create_blackbox` command, or to check whether a specific module is a black box, check the `design_type` attribute of the module as follows:

```
icc2_shell> get_cells -hierarchical * -filter {design_type==black_box} {u0_3}
icc2_shell> get_attribute [get_cells u0_3] design_type black_box
```
The GUI provides annotations in the Hierarchy Browser and Properties Editor to help identify black boxes as shown in Figure 4-3.

**Figure 4-3  Black Box Display in Hierarchy Browser and Properties Editor**

To modify the area set by the `create_blackbox -target_boundary_area area` command, set the `target_boundary_area` attribute directly on the black box cell with the following command:

```
icc2_shell> set_attribute -objects [get_cells u0_3] \
    -name target_boundary_area -value 1500000 {u0_3}
```

For black boxes created from a partial Verilog netlist, you can specify a target utilization with the following command:

```
icc2_shell> set_attribute -objects [get_cells u0_3] \
    -name target_utilization -value 0.5 {u0_3}
```

Both the `target_boundary_area` and `target_utilization` attributes are honored by the `shape_blocks` command during block shaping.

### Creating a Black Box Timing Model

A black box contains no timing information by default. Any path that begins or ends at a black box is unconstrained. IC Compiler II provides commands to create a timing model for a black box, which allows the module to be used for timing budgeting and optimization.
To create a black box timing model,

1. Define the clock ports, drive and load.

   icc2_shell> set_blackbox_clock_port ...
   icc2_shell> set_blackbox_port_drive ...
   icc2_shell> set_blackbox_port_load ...

2. Set the clock network delay, pin-to-pin delays, setup and hold constraints.

   icc2_shell> create_blackbox_clock_network_delay ...
   icc2_shell> create_blackbox_constraint ...
   icc2_shell> create_blackbox_delay ...
   icc2_shell> create_blackbox_drive_type ...
   icc2_shell> create_blackbox_load_type ...

3. Commit the timing model to the black box.

   icc2_shell> commit_blackbox_timing 1

You create timing model for a black box by specifying the model ports, the setup and hold constraints on the inputs, the clock-to-output path delays, and the input-to-output path delays. You can also specify the loads on input ports and the drive strength of output ports.

Black box timing information is stored in the design view. When an abstract is created from a black box, specially marked cells, nets, and connections are created on-the-fly within the abstract view. This approach enables timing on an empty or partial netlist and allows newly created feedthroughs to be timed immediately without re-creating the models.

For a partial black box, only the unconnected ports can be constrained with black box timing constraints. Other ports on the partial black box should be constrained by using Synopsys Design Constraints (SDC) files.

When generating timing budgets, the tool maintains the timing you specified with the black box timing commands. To allow the budgeter to adjust the black box timing budgets, set the plan.budget.bbt_fixed_delay application option as follows:

   icc2_shell> set_app_options -name plan.budget.bbt_fixed_delay -value false
   plan.budget.bbt_fixed_delay false

---

**Black Box Timing Example**

The following example shows the commands used to create a black box timing model. The resulting model is shown in Figure 4-4.

```plaintext
set_blackbox_clock_port [get_ports {clk1_in clk1_out clk2_in}]
create_blackbox_load_type -lib_cell buf1 lt1
set_blackbox_port_load -type lt1 -factor 3 [get_ports in1]
create_blackbox_clock_network_delay -value 3.5 clk_1
```
create_blackbox_delay -rise_from in1 -fall_to out1 -value 5.0
create_blackbox_delay -fall_from in1 -rise_to out1 -value 5.2
create_blackbox_delay -from in2 -to out2 -clock main_clock -value 0.4
create_blackbox_constraint -setup -edge rise -from clk2_in -to in3 -value 1.2
create_blackbox_constraint -hold -edge rise -from clk2_in -to in3 -value 0.8
create_blackbox_delay -from clk1_in -to clk1_out -value 1.1
create_blackbox_constraint -edge rise -from clk1_in -to in3 -clock main_clock -value 0.5
create_blackbox_constraint -edge rise -from clk1_in -to in2 -value 1.3
create_blackbox_delay -rise_from clk1_in -to out2 -value 2.2
create_blackbox_delay -fall_from clk1_in -to out3 -value 1.7
commit_blackbox_timing

Figure 4-4  Example Black Box Timing Model
Planning I/Os and Flip-Chip Bumps

After creating the floorplan, you can instantiate the I/O drivers and bump cells for your design. The IC Compiler II tool supports advanced features for unconstrained and constraint-based placement of I/O drivers and flip-chip bump cells. You can create I/O placement constraints and specify the ordering, xy coordinate, and placement side for each I/O. The tool supports both package-driven and die-driven I/O placement flows, and you can adapt the tool for different chip packaging scenarios. Different I/O driver styles, such as pads with integrated drivers, are supported by the tool.

Chip I/Os can be placed along the chip periphery or in islands within the chip core. Constraints include signal constraints and power constraints to specify I/O adjacency requirements and power requirements. You can perform I/O placement by using Tcl commands and GUI interfaces.
The flow for instantiating I/O drivers and pads is shown in Figure 5-1.

**Figure 5-1  I/O Planning Flow**

1. **Create I/O rings or guides**
   (create_io_ring, create_io_guide)
2. **Create bump pad arrays**
   (create_bump_array)
3. **Create power I/O constraints**
   (set_power_io_constraints)
4. **Create signal I/O constraints**
   (set_signal_io_constraints)
5. **Create matching types**
   (create_matching_type)
6. **Place I/O drivers**
   (place_io)
7. **Write I/O placement constraints**
   (write_io_constraints)
8. **Route RDL nets**
   (route_rdl_flip_chip)
9. **Optimize RDL nets**
   (optimize_rdl_routes)
10. **Create RDL route shields**
    (create_rdl_shields)

For more details, see the following topics:
- Creating I/O Rings or Guides
• Creating Arrays of Bump Cells
• Placing Bump Cells in Predetermined Locations
• Creating Power I/O Placement Constraints
• Creating Signal I/O Placement Constraints
• Assigning I/O Pads to Bumps With Matching Types
• Placing I/Os and Writing Constraints
• Routing RDL Nets
• Optimizing RDL Routes
• Creating RDL Net Shields
Creating I/O Rings or Guides

The IC Compiler II tool supports I/O guides, which form a placement area for I/O drivers. You can create an I/O ring by placing four I/O guides along the periphery of the floorplan with four `create_io_guide` commands, or by using one `create_io_ring` command. The IC Compiler II tool also supports multiple I/O rings and supports multiple I/O guides on the same side of the die.

You must create I/O guides or I/O rings before placing I/O cells. The following commands create a ring containing four I/O guides around the periphery of the floorplan and retrieve the names of the I/O guides created by the `create_io_ring` command.

```
icc2_shell> create_io_ring -name outer_ring -corner_height 300
{outer_ring}
icc2_shell> get_io_guides
{outer_ring.left outer_ring.bottom outer_ring.right outer_ring.top}
```

You can also use four `create_io_guide` commands to create I/O guides on the left, top, right, and bottom sides of the die.

```
icc2_shell> create_io_guide -name "ring_left" -line {{0 300} 3000} \\ -side left
{ring_left}
icc2_shell> create_io_guide -name "ring_top" -line {{300 3640} 3000} \\ -side top
{ring_top}
icc2_shell> create_io_guide -name "ring_right" -line {{3680 3380} 3000} \\ -side right
{ring_right}
icc2_shell> create_io_guide -name "ring_bottom" -line {{3380 0} 3000} \\ -side bottom
{ring_bottom}
icc2_shell> get_io_guides
{ring_left ring_top ring_right ring_bottom}
```

Alternatively, you can use the Floorplan Task Assistant to create the I/O rings and I/O guides in the GUI. Select Task > Task Assistant, then select Floorplan Preparation > I/O Planning > I/O Ring and Guide. Enter the information for the I/O ring or I/O guide and click Apply to create the ring or guide.
I/O Ring and I/O Guide Tasks

Use options with the `create_io_ring` and `create_io_guide` commands to control how the ring is created:

- Assign a name to the ring with the `-name` option. The name is used as a prefix for the I/O guide names.
  ```
  icc2_shell> create_io_ring -name outer_ring
  icc2_shell> get_io_rings
  {outer_ring}
  icc2_shell> get_io_guides
  {outer_ring.left outer_ring.bottom outer_ring.right outer_ring.top}
  ```

- Create a ring inside an existing ring with the `-inside` option. The existing ring is used as the outer boundary for the new ring.
  ```
  icc2_shell> create_io_ring -name inner -inside outer_ring
  ```

- Specify the distance between the inner and outer ring boundaries with the `-offset` option.
  ```
  icc2_shell> create_io_ring -name inner -inside outer -offset 500
  ```

- Create a ring within a specific bounding box with the `-bbox` option.
  ```
  icc2_shell> create_io_ring -bbox {{1000 1000} {2000 2000}}
  ```

- Specify a list of pad cell instances to assign to the ring with the `-pad_cell_list` option.
  ```
  icc2_shell> create_io_ring -pad_cell_list {pad_iopad_0 pad_iopad_1}
  ```

- Include specific I/O guides in the ring with the `-guides` option.
  ```
  icc2_shell> get_io_guides
  {ring_left ring_top ring_right ring_bottom}
  icc2_shell> create_io_ring -guides [get_io_guides] {default_io_ring1}
  ```

- Specify the minimum distance between the startpoint of the I/O guide and the first I/O driver, and the minimum distance between the endpoint of the I/O guide and the last I/O driver with the `-offset` option of the `create_io_guide` command.

Editing, Reporting, and Removing I/O Guides and I/O Rings

- Add I/O pad cells to an I/O guide with the `add_to_io_guide` command.
  ```
  icc2_shell> add_to_io_guide guide_left {pad_iopad_0 pad_iopad_1}
  ```

- Create a collection of I/O guides with the `get_io_guides` command and create a collection of I/O rings with the `get_io_rings` command.
Creating Arrays of Bump Cells

The IC Compiler II tool supports bump cell placement for flip-chip designs. You can place bump cells based on placement coordinates in a DEF or AIF file, or by creating a bump array with the `create_bump_array` command. You should place bump cells before placing I/O driver cells. In the die-driven flow, you create a region for bump cell placement and let the tool determine the locations for the bump cells. The following example creates five bump array regions, one each for the left, top, right, bottom, and center die areas, and fills each region with BUMP library cells.

```
icc2_shell> create_bump_array -lib_cell BUMP -delta {140 140} -bbox {{410 450} {1175 3278}} -name left
icc2_shell> create_bump_array -lib_cell BUMP -delta {140 140} -bbox {{2620 450} {3325 3278}} -name right
icc2_shell> create_bump_array -lib_cell BUMP -delta {140 140} -bbox {{1100 450} {2650 1125}} -name bottom
icc2_shell> create_bump_array -lib_cell BUMP -delta {140 140} -bbox {{1100 2555} {2650 3250}} -name top
icc2_shell> create_bump_array -bbox {{1100 1150} {2650 2550}} -pattern staggered_1 -lib_cell {BUMP} -delta {140 140} -name pg_bump
```

Figure 5-2 shows the layout after running the previous commands.
Bump Array Tasks

Use options with the `create_bump_array` command to control how the bump array is created:

- Specify the horizontal and vertical spacing between adjacent rows with the `-delta` option; `-delta` and `-lib_cell` are required options.
  
  ```bash
  icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150}
  ```

- Create a name for the bump array with the `-name` option.
  
  ```bash
  icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \
  -name pg_bump
  ```

- Specify a rectangular bounding box to contain the bump cells with the `-bbox` option.
  
  ```bash
  icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \
  -bbox {{1000 1000} {3000 3000}}
  ```
• Create the bump array within a complex rectilinear shape with the -boundary option.

```shell
icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \\
-boundary {{1000 1000} {3000 1000} {3000 3000} {1000 3000}}
```

• Limit the bump array size by number of columns or number of rows with the -repeat option. Bumps are centered in the die or bounding box.

```shell
icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \\
-repeat {5 5}
```

• Set the spacing between the lower-left corner of bump array bounding box and the lower-left corner of the bump cell instance at column 0 and row 0 with the -origin option.

```shell
icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \\
-origin {1000 1000}
```

• Specify the orientation of each bump cell in the bump array with the -orientation option.

```shell
icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \\
-orientation N
```

Legal orientation values are N, W, S, E, FN, FS, FE, and FW.

• Specify whether the bump cells occupy each possible location or alternate locations with the -pattern inline | staggered_1 | staggered_2 option.

```shell
icc2_shell> create_bump_array -lib_cell BUMP -delta {150 150} \\
-pattern staggered_1
```

The inline keyword places bump cells at each point in the array specified by the -delta option. The staggered_1 keyword places bump cells at points where the sum of the column index and the row index is even. The staggered_2 keyword places bump cells at points where the sum of the column index and the row index is odd. Figure 5-3 shows a small bump placement example that uses these three placement patterns.
Placing Bump Cells in Predetermined Locations

If the locations of the bump cells are determined by the constraints of the package, use the `read_aif` command to read in a list of bump cells and locations. The location information is typically read from a DEF or AIF file, which contains x- and y-coordinates and bump cell names for the bump cells in the block.

`icc2_shell> read_aif bumps.aif`

A short example from an AIF file is as follows:

```
[DATABASE]
TYPE=AIF
VERSION=2.0
UNITS=UM
[DIE]
NAME=leon3mp_die
WIDTH=3840.575700
HEIGHT=3690.006000
[PADS]
BUMP N = POLY -17.000 46.500 -17.000 46.475 ...
[NETLIST]
;Netname Pad#    Type    Pad_X       Pad_Y       Ball#
-  left_0_0      BUMP N  -1463.7878  -1348.5030
-  left_0_1      BUMP N  -1323.7878  -1348.5030
...  
-  right_0_0     BUMP N  746.2122    -1348.5030
-  right_0_1     BUMP N  886.2122    -1348.5030
...  
-  bottom_0_0    BUMP N  -773.7878   -1348.5030
-  bottom_0_1    BUMP N  -633.7878   -1348.5030
...  
```

**Figure 5-3  Bump Array Placement Pattern Options**

- pattern inline (default)
- pattern staggered_1
- pattern staggered_2
Creating Power I/O Placement Constraints

Power I/O placement constraints specify spacing and other requirements for power I/O drivers. Use the `set_power_io_constraints` command to set constraints for the placement of I/O driver pads as shown in the following example.

```bash
icc2_shell> set_power_io_constraints -io_guide_object [get_io_guides *] -ratio {{3 VDD_NS} {7 VSS_NS}} -reference_cell {VDD_NS VSS_NS}
```

Power I/O constraints set successfully

The `set_power_io_constraints` command defines how the `place_io` command places driver pads within I/O guides. The preceding example specifies a constraint for all I/O guides in the design, limits the number of signal pads between successive VDD_NS pads to three, and limits the number of signal pads between the successive VSS_NS pads to seven. Figure 5-4 shows a small section of the lower-left corner of the layout after running the `set_power_io_constraints` and `place_io` commands.
Creating Power I/O Placement Constraints

**Power I/O Constraint Tasks**

Use options with the `set_power_io_constraints` command to further control power signal I/O placement.

- Specify up to two library cell names for the power driver pads with the `-reference_cell` option.

  ```bash
  icc2_shell> set_power_io_constraints -reference_cell {VDD_NS VSS_NS}
  ```

- Specify the list of I/O guides for which to apply the constraint with the `-io_guide_object` option.

  ```bash
  icc2_shell> set_power_io_constraints -reference_cell {VDD_NS VSS_NS} -io_guide_object {guide_top}
  ```

- Assign more than one power pad to a given bump cell with the `-share` option.

  ```bash
  icc2_shell> set_power_io_constraints -reference_cell {VDD_NS VSS_NS} -share {{2 VDD_NS} {3 VSS_NS}}
  ```

In this example, `-share {{2 VDD_NS} {3 VSS_NS}}` allows up to two VDD power pads to drive a single VDD bump cell, and three VSS power pads to drive a single VSS bump cell.
• Define the number of signal pads that can be placed between successive power pads with the -ratio option.

    icc2_shell> set_power_io_constraints \
      -reference_cell {VDD_NS VSS_NS} -ratio {{7 VDD_NS} {6 VSS_NS}}

    In this example, the tool places no more than seven I/O pads between VDD_NS pad cells, and no more than six I/O pads between VSS_NS pad cells.

• Specify the maximum spacing between successive power pads of the same type with the -spacing option.

    icc2_shell> set_power_io_constraints -reference_cell {VDD_NS VSS_NS} \
      -spacing {{100 VDD_NS} {100 VSS_NS}}

• Specify the maximum distance between the starting point of the I/O guide and the closest edge of the first power pad of the specified type with the -offset option.

    icc2_shell> set_power_io_constraints -reference_cell {VDD_NS VSS_NS} \
      -offset {{100 VDD_NS} {100 VSS_NS}}

    If you do not specify constraints with the set_power_io_constraints command, the place_io command places the power pads evenly throughout the I/O guide.

Creating Signal I/O Placement Constraints

To create signal I/O placement constraints to specify the ordering and spacing for signal I/O drivers, use the set_signal_io_constraints command. In the following example, the command specifies an order-only constraint for the placement of three signal I/O drivers in the _default_io_ring1.left I/O guide: u_p/clk_pad_P1, u_p/sdclk_pad_P1, and u_p/test_so_4_pad_P1.

    icc2_shell> set_signal_io_constraints -constraint {{order_only} \
      -io_guide_object _default_io_ring1.left} \
      {u_p/clk_pad_P1 u_p/sdclk_pad_P1 u_p/test_so_4_pad_P1}

    Signal IO constraints set successfully

    1

The -constraint {constraint_spec} option supports different formats for specifying the placement of signal I/O drivers. Use the -constraint {{order-only} pad_1 pad_2 ...} option to place the pads in the specified order. Note that additional spacing and other pad cells might be inserted between adjacent pads. To create a fixed space between I/O drivers, insert a spacing value in microns between the signal names. For example, the -constraint {pad_1 10 pad_2 20 pad_3} option inserts a 10 micron space between pads pad_1 and pad_2, and a 20 micron space between pads pad_2 and pad_3. To place each signal I/O driver at a specified pitch, use the -constraint {{pitch} pad_1 pad_2 ...} option format.
Alternatively, you can write out the signal I/O placement constraints with the `write_io_constraints -filename constraint_file_name` command, then use the `set_signal_io_constraints -file constraint_file_name` command to load the constraints. The constraint file is useful when there is a large number of signal constraints to specify. The following example loads signal I/O placement constraints from the `signal_io_constraints.txt` constraints file.

```
icc2_shell> set_signal_io_constraints -file signal_io_constraints.txt
Signal I/O constraints set successfully

icc2_shell> shell cat signal_io_constraints.txt
    _default_io_ring1.left
    {order_only}
    u_p/clk_pad_P1
    u_p/sdclk_pad_P1
    u_p/test_so_4_pad_P1;
```

Signal ordering begins at the lower-left pad for the left edge, the upper-left pad for the top edge, the upper-right pad for the right edge, and the lower-right pad for the bottom edge as shown in Figure 5-5.

*Figure 5-5 Signal I/O Constraints Side Ordering*
In the following example, a signal I/O constraints file creates an ordering and spacing constraint for the top I/O guide in the ring. The constraint creates an 80 micron gap between the start of the I/O guide and the first signal I/O pad. A 55 micron pitch is used to place the Pad_1, Pad_2, and Pad_3 I/O drivers. The constraint creates a 10 micron spacing between the Pad_3 and Pad_4 I/O drivers, a 40 micron spacing between Pad_4 and Pad_5, a 60 micron spacing between Pad_5 and Pad_6, and no spacing between Pad_6, Pad_7, and Pad_8. Example 5-1 shows the signal I/O constraint file, and Figure 5-6 shows the layout results after running the place_io command.

Example 5-1  I/O Constraints File Example

    ring.top
    {80}
    {{55} Pad_1 Pad_2 Pad_3}
    {10}
    {Pad_4 40 Pad_5 60 Pad_6}
    {Pad_7 Pad_8}
    ;

Figure 5-6  Ring.top I/O Placement

Assigning I/O Pads to Bumps With Matching Types

Before routing flip-chip drivers to flip-chip nets, you can associate flip-chip drivers and bump cells by designating a matching type. The matching type for a cell has two primary functions: provide support for driver-to-bump pairing and provide support for driver placement. The tool assigns flip-chip drivers to flip-chip bump cells that have the same matching type.

The following example uses two create_matching_type commands. The first command associates pad cells pad_iopad_46 through pad_iopad_49 with bump cells in row 25. The second command associates pad cells pad_iopad_42 through pad_iopad_45 with bump cells in row 24. The -uniquify option specifies the number of pad pins that can be assigned to a single bump cell; a value of 0 specifies that each bump cell can be assigned to only one pad cell. The place_io command applies the matching type assignments to the pad and
bump cells; this command is required to view the updated flylines with the RDL flylines panel.

```plaintext
cicc2_shell> create_matching_type -name match1 -uniquify 0 
left_25_0 left_25_1 left_25_2 left_25_3 
pad_iopad_46 pad_iopad_47 pad_iopad_48 pad_iopad_49

cicc2_shell> create_matching_type -name match2 -uniquify 0 
left_24_0 left_24_1 left_24_2 left_24_3 
pad_iopad_42 pad_iopad_43 pad_iopad_44 pad_iopad_45
```

Figure 5-7 shows the RDL flyline connections created by the previous commands. View RDL flylines in the layout by choosing View > Flylines > RDL Flylines in the GUI.

**Figure 5-7  RDL Flylines Between Matching Cells**

Other Matching Type Command Examples

Use the following commands to modify, report, and delete matching types.

- Return a collection of matching types with the `get_matching_types` command.

  ```plaintext
  icc2_shell> get_matching_types
  {SIGNAL POWER}
  ```

- Create a collection of cells that are assigned a specified matching type name with the `get_cells` command and `matching_type.name` attribute.
• Add I/O pad or bump cells to the matching type with the `add_to_matching_type` command.

```shell
icc2_shell> add_to_matching_type match2 {pad_iopad_40 pad_iopad_41}
```

• Write out your user-defined matching type assignments with the `write_matching_types` command.

```shell
icc2_shell> write_matching_types -file_name user_matching.tcl
```

If you did not create any matching type assignments, the file is empty.

• Write out the default matching types assigned by the tool with the `write_matching_types -from_existing_assignment` command.

```shell
icc2_shell> write_matching_types -from_existing_assignment -file_name existing.tcl
```

```shell
icc2_shell> sh head existing.tcl
```

```bash
create_matching_type -name existing_assignment_0 -uniquify 1 [list [get_cells { bottom_0_0 }]
 [get_pins { __added_power_driver_1/VDDPAD }]]
create_matching_type -name existing_assignment_1 -uniquify 1 [list [get_cells { bottom_0_1 }]
 [get_pins { sdram_DQ_iopad_59/PADIO }]]
create_matching_type -name existing_assignment_2 -uniquify 1 [list [get_cells { bottom_0_10 }]
 [get_pins { sdram_DQ_iopad_38/PADIO }]]
...```

• Remove one or more matching types with the `remove_matching_types` command.

```shell
icc2_shell> remove_matching_types {match2}
```

```shell
icc2_shell> remove_matching_types [get_matching_types]
```

• Remove specific I/O pad or bump cells with the `remove_from_matching_type` command.

```shell
icc2_shell> remove_from_matching_type match2 {pad_iopad_42}
```

• Create a list of matching types and associated I/O pad or bump cells with the `report_matching_types` command.
Placing I/Os and Writing Constraints

After setting power and signal I/O placement constraints, you can place the I/O drivers and write out the placement constraints.

To place I/Os and write out the placement constraints,

1. Place the I/Os with the `place_io` command.

   icc2_shell> place_io

2. Write out the I/O placement constraints file based on the current I/O placement with the `write_io_constraints` command.

   icc2_shell> write_io_constraints -filename io_constraints.txt

   icc2_shell> shell cat io_constraints.txt
   _default_io_ring1.left
   {{order_only} u_p/test_si_13_pad_P1
   u_p/data_pad_1_x0_3_P1
   u_p/data_pad_2_x0_3_P1
   u_p/data_pad_2_x0_2_P1
   ...

The `place_io` command supports options to control how I/Os are placed. Use the `-io_guide io_guide_list` option to place I/Os only in the specified I/O guides. If the I/O guide is not long enough to contain all pads, the pads are placed after the endpoint of I/O guides. Use the `-rule rule_name` option to specify a single routing spacing rule for redistribution layer (RDL) routes. Use the `-incremental` option to place additional I/O drivers for nonflip-chip designs.

Note that the `place_io` command removes existing filler cells if the `physical_status` property is not set to `fixed`. This condition occurs when all pads are considered during flip-chip I/O placement. If only a fraction of pads are incrementally re-placed, then filler cells are not removed.

If you dedicate a matching type to a collection of only bump cells, the tool will not connect these bump cells to I/O drivers. Use this feature to reserve certain bump cells for power or ground and avoid connecting the bump cells to I/O driver cells. If you apply a matching type to I/O drivers, I/O driver pins, and I/O driver terminals, but don't assign a matching type for
bumps, the place_io command skips bump assignment and continues to place other I/Os. If an I/O driver pin is already connected to a bump through a two-pin net, the pad pin will remain connected to the bump. For isolated pad cells which are not assigned to an I/O guide, the tool locates the closest unassigned bump cell and calculates a flyline route between the pad cell and bump cell. For groups of pad cells not within an I/O guide, the tool creates a virtual I/O guide.

The write_io_constraints command supports options to control the name and contents of the output file. Use -filename file_name option to specify the output constraints file name. Use the -format order_only | spacing | pitch | fixed option to specify the type of constraints to write to the output file. The order_only argument writes only pad order information to the file. The spacing argument writes the spacing values between pads and the pad cell names to the file. The pitch argument writes pad cell locations by using cell pitch instead of spacing values and requires that all pad cells be placed evenly in the I/O guide. The fixed argument specifies that all pad cells are fixed at their current locations.

Routing RDL Nets

After placing the bump and I/O driver cells, you can route the redistribution layer (RDL) nets. To route the RDL nets,

1. Define the routing rules for the RDL net routes. If you plan to add net shields for the RDL routes, specify the -shield_spacings and -shield_widths options as described in Creating RDL Net Shields.

  icc2_shell> create_routing_rule rdlrule -widths {MRDL 2} \
   -spacings {MRDL 1}

2. Apply the routing rule to the flip-chip bump cells.

   icc2_shell> set rdlnets [get nets -of_objects [get_pins \
   -of_objects [get_cells -filter "design_type==flip_chip_pad"]]]

   [...] 

   icc2_shell> set_routing_rule [get nets $rdl_nets] -rule rdlrule

3. Set the flip_chip.route.layer_routing_angles application option to 90_degree or 45_degree to specify the routing type for the RDL layer.

   icc2_shell> set_app_options \
   -name flip_chip.route.layer_routing_angles \
   -value {{MRDL 90_degree}}

1

4. Route the RDL nets.
Figure 5-8 shows the layout result produced by the previous commands. The RDL nets are routed at 90-degree angles and connect the I/O drivers and bump cells.

Other RDL Routing Tasks

Use the following commands and application options to perform different RDL routing tasks.

- Create 45-degree routes by setting the flip_chip.route.layer_routing_angles application option to 45_degree before running the route_rdl_flip_chip command.

```
icc2_shell> set_app_options
    -name flip_chip.route.layer_routing_angles
    -value {{MRDL 45_degree} {M9 45_degree}}
```

```
icc2_shell> route_rdl_flip_chip -layers {MRDL M9} -nets $rdl_nets
```
The tool creates RDL routes as shown in Figure 5-9.

Figure 5-9  45-Degree RDL Routes

- Create an RDL route on an adjacent layer that is parallel to the current route by using the `split_rdl_routes` command.

```
icc2_shell> split_rdl_routes -nets {pad[48] pad[49]} \
   -mode adjacent_layer -via_interval 25 -from_layers MRDL \ 
   -to_layers M9
```

The tool adds a parallel route on the M9 layer and inserts vias as shown in the upper two routes in Figure 5-10.

Figure 5-10  Split Routes
• Split an existing route into multiple thin routes on the same layer with the `split_rdl_routes` command and the `-widths`, `-spacings`, and `-number_of_routes` options.

```shell
icc2_shell> split_rdl_routes -nets {pad[46] pad[47]} \
    -widths {MRDL {2 2 2}} -spacings {MRDL {2 2}} \
    -number_of_routes {MRDL 3}
```

The tool splits the existing route into multiple thin routes as shown in the lower two routes in Figure 5-10.

• Create length-matched routes for the specified routes by using the `route_rdl_differential` command.

```shell
icc2_shell> route_rdl_differential -layers {MRDL} \
    -nets {pad[41] pad[43]}
```

The tool creates routes for the specified nets as shown in Figure 5-11.

![Figure 5-11 Differential Routes](image)

• Create a tapered route by setting taper distances and widths with the `create_routing_rule` command. Create the tapered route with the `route_rdl_flip_chip` command.

```shell
icc2_shell> create_routing_rule taperrule \ 
    -widths {MRDL 10 M9 10} -spacings {MRDL 10 M9 10} \ 
    -rdl_taper_distances {MRDL 40 M9 40} \ 
    -rdl_taper_widths {MRDL 5 M9 5}

icc2_shell> set_routing_rule [get_nets $rdl_nets] -rule taperrule

icc2_shell> route_rdl_flip_chip -layers {MRDL M9} -nets $rdl_nets
```

The tool creates the tapered routes as shown in Figure 5-12.
• Remove all RDL routes with the `remove_routes -rdl` command.

```
icc2_shell> remove_routes -rdl
Successfully removed 321 route shapes.
```

Optimizing RDL Routes

After creating RDL routes with the `route_rdl_flip_chip` command or by using the GUI, use the `optimize_rdl_routes` command to improve the RDL routing patterns by reducing the number of U- and Z-shaped routes in the block and create additional space for RDL power routes. Perform the following tasks to optimize RDL routes:

• Improve RDL routing.

```
icc2_shell> optimize_rdl_routes -nets $rdl_nets
```
Figure 5-13 shows the layout result before and after running the previous command.

**Figure 5-13  RDL Route Optimization**

- Move signal routes to create additional routing area for power and ground routes with the `-reserve_power_resources` option.

  ```
  icc2_shell> optimize_rdl_routes -layer {MRDL} -nets $rdl_nets \ 
  -reserve_power_resources true
  ```

After running the command with this option, the amount of U- and Z-shaped routes might increase. Figure 5-14 shows the layout result before and after running the previous command.
Creating RDL Net Shields

RDL net shields provide isolation for RDL nets. To create RDL routes and shield the routes,

1. Set the `flip_chip.route.shielding_net` application option to define the shielding net connection.

   ```shell
   icc2_shell> set_app_options -list {flip_chip.route.shielding_net VSS}
   ```

2. Create the routing rule and specify the shield spacings and shield widths.

   ```shell
   icc2_shell> create_routing_rule shieldrule \ 
     -shield_spacings {MRDL 2 M9 2} -shield_widths {MRDL 5 M9 5} \ 
     -widths {MRDL 6 M9 5}
   ```

3. Assign the routing rule to the RDL nets.

   ```shell
   icc2_shell> set_routing_rule $rdl_nets -rule shieldrule
   ```

4. Route the RDL nets.

   ```shell
   icc2_shell> route_rdl_flip_chip -layers {MRDL M9} -nets $rdl_nets
   ```

5. Create the RDL net shields.

   ```shell
   icc2_shell> create_rdl_shields -layers {MRDL M9} -nets $rdl_nets
   ```

Figure 5-15 shows the layout result produced by the previous commands.
RDL Net Shielding Options

Use the following options with the `create_rdl_shields` command to control how the shields are created.

- List the RDL nets in a file and specify the file name with the `-nets_in_file` option.
  
  ```
  icc2_shell> create_rdl_shields -layers {MRDL M9} \ 
  -nets_in_file rdlnets.txt
  icc2_shell> sh cat rdlnets.txt
  pad[40]
  pad[41]
  ...
  ```

- Enable or disable shielding on bumps with the `-shield_on_bump` option.
  
  ```
  icc2_shell> create_rdl_shields -layers {MRDL M9} -nets $rdl_nets \ 
  -shield_on_bump true
  ```

  Figure 5-16 shows the layout result with the `-shield_on_bump` option set to true (left) and false (right).

Figure 5-16  Bump Shields -shield_on_bump Option
• Enable or disable shielding on via ties with the \texttt{--shield\_via\_tie} option.
• Remove floating shields after creating them with the \texttt{--trim\_floating true} option.
To manage design blocks, the IC Compiler II tool supports operations to easily partition your design and commit a logical hierarchy cell to a physical hierarchy block early in the design flow. After committing to blocks, you can create multiple optimized, abstract views for design blocks that contain only the information needed to perform placement, timing, and other tasks. This approach enables you to minimize the system requirements needed to efficiently distribute and process very large designs.
The flow to commit blocks and create abstracts is shown in Figure 6-1.

**Figure 6-1  Manage Block Flow**

![Diagram of the manage block flow]

For more details, see the following topics:

- Exploring the Design Hierarchy
- Creating Module Boundaries
- Committing Design Blocks
- Creating Block Placement Abstracts
- Moving the Origin of a Block
Exploring the Design Hierarchy

The IC Compiler II tool provides tools to explore your design hierarchy and determine which logical hierarchy cells to commit to physical hierarchy blocks. The hierarchy browser in the GUI displays a view of the design hierarchy and provides other information about each hierarchy cell. To explore the design hierarchy with the Hierarchy Browser,

1. Choose View > Hierarchy Browser in the GUI.
2. Expand or collapse the hierarchy by clicking the [+] symbols next to the hierarchy cells.
3. (Optional) Click and drag the column headers to rearrange the table.
4. Click the block name to perform additional operations on the block.

An example hierarchy as displayed in the Hierarchy Browser is shown in Figure 6-2.

Figure 6-2  Hierarchy Browser

<table>
<thead>
<tr>
<th>Logical Hierarchy</th>
<th>utilization</th>
<th>number_of_pins</th>
<th>hard_macro</th>
<th>hierarchy</th>
<th>std_cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>T leon3mp</td>
<td>0.220</td>
<td>261</td>
<td>0</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>M u_p</td>
<td></td>
<td>693</td>
<td>0</td>
<td>0</td>
<td>254</td>
</tr>
<tr>
<td>M u_m</td>
<td></td>
<td>14288</td>
<td>0</td>
<td>4</td>
<td>2852</td>
</tr>
<tr>
<td>MB u0_3</td>
<td>0.220</td>
<td>6344</td>
<td>0</td>
<td>18</td>
<td>16492</td>
</tr>
<tr>
<td>MB u0_2</td>
<td>0.220</td>
<td>6333</td>
<td>0</td>
<td>18</td>
<td>16785</td>
</tr>
<tr>
<td>MB u0_1</td>
<td>0.220</td>
<td>6339</td>
<td>0</td>
<td>18</td>
<td>3762</td>
</tr>
<tr>
<td>MB u0_0</td>
<td>0.220</td>
<td>6339</td>
<td>0</td>
<td>18</td>
<td>3762</td>
</tr>
</tbody>
</table>

In the Hierarchy Browser, the top level contains six cells: u_p, u_m, u0_0, u0_1, u0_2, and u0_3. The figure shows the cell name, utilization, number of pins, and number of hard macros and number of standard cells.

Creating Module Boundaries

In preparation for committing logical hierarchy cells to physical hierarchy blocks, you can create module boundaries in the GUI. Module boundaries represent uncommitted cells (modules) in the logical hierarchy. Using module boundaries, you can quickly assess the approximate block size that is required for a given utilization for a top-level block, and arrange module boundaries by hand within the floorplan for rough estimation of block placement. You can also set different parameters to control which top-level blocks are treated as module boundaries in this view.
To create module boundaries,

1. Use the `explore_logic_hierarchy` command and specify the cell names for which to create module boundaries.

   ```
   icc2_shell> explore_logic_hierarchy -create_module_boundary -cell {u0_0 u0_1 u0_2 u0_3}
   ```

   The tool creates module boundaries based on the options you specify. By default, the tool determines the size of the boundary based on the utilization of the top-level floorplan. After creating the module boundaries, the tool updates the display in the GUI as shown in Figure 6-3. The original floorplan boundary is shown on the left, the I/Os are located above the floorplan boundary, the module boundaries are placed at the bottom right, and any unplaced macros are placed above the module boundaries.

   **Figure 6-3  Module Boundaries**

2. (Optional) Experiment with different block placements by selecting a module boundary and moving it into the floorplan boundary as shown in Figure 6-4.
3. Reset the module boundary positions or remove the module boundaries to explore other hierarchies.
   
   \texttt{icc2\_shell> explore\_logic\_hierarchy -organize}
   \texttt{icc2\_shell> explore\_logic\_hierarchy -remove}

   The \texttt{-organize} option resets the locations the module boundaries to the original locations created by the \texttt{explore\_logic\_hierarchy -create\_module\_boundary} command as shown in Figure 6-3. The \texttt{-remove} option removes all module boundaries. You can also combine the \texttt{-remove} and \texttt{-cell} options to remove specific module boundaries.

4. Repeat the flow by creating module boundaries for other cells to explore different hierarchies.

To create the initial module boundaries with the GUI,

1. Open the Hierarchy Exploration panel in the GUI by choosing View > Assistants > Hierarchy Exploration, or select Floorplan Preparation > Hierarchy Exploration in the Task Assistant.

2. Select the blocks for which to create module boundaries.

3. Click Create Module Boundary from the context menu as shown in Figure 6-5.

   The tool creates module boundary for the selected logic hierarchies.
Creating Module Boundaries for Flat Designs

The `explore_logic_hierarchy` command can also create module boundaries for a flat design. Set the `plan.place.hierarchy_by_name` application option to `true`; this enables the `explore_logic_hierarchy` command to extract virtual hierarchies by name and create move bounds for the virtual hierarchies. Any existing module boundaries are removed before creating the new move bounds. To create individual move bounds for each module, specify the modules explicitly with the `-cell` option as follows:

```
icc2_shell> explore_logic_hierarchy -create_module_boundary \ 
              -cell {I_ORCA_TOP/I_PCI_TOP I_ORCA_TOP/I_SDRAM_TOP}
```

To create a single move bound to group two or more modules, specify the `-virtual_group` option as follows:

```
icc2_shell> explore_logic_hierarchy -name group1 \ 
              -virtual_group {I_ORCA_TOP/I_PCI_TOP I_ORCA_TOP/I_SDRAM_TOP}
```

Use the `get_attribute` command as follows to report the individual modules associated with the grouped move bound:

```
icc2_shell> get_attribute [get_bounds group1] -name flat_hier_names
```

Committing Design Blocks

In the IC Compiler II design planning flow, you commit logical hierarchy cells to physical hierarchy blocks early in the design process. You can use the connectivity of the cells, size of the cells, and other factors to determine which logical hierarchy cells to commit. After deciding which logical hierarchy cells to commit to physical hierarchy blocks, you can convert them to an abstract representation for placement, power planning, pin placement and timing budgeting. Interactive tools support your analysis by displaying the net connectivity between different modules.
To commit a block in the design,

1. (Optional) Create a new library to contain the block and save the library. This enables each block to be processed in parallel by using distributed processing.

   \[
   \text{icc2\_shell}\> \text{create\_lib} \; \ldots/\text{lib}/\text{leon3s\_ndm} \; \text{-technology saed90nm\_tf} \; \text{-ref\_libs} \; \$\text{libs} \\
   \text{icc2\_shell}\> \text{save\_lib} \\
   \text{icc2\_shell}\> \text{set\_ref\_libs} \; \text{-add} \; \text{leon3s\_ndm}
   \]

2. Commit the block. The block is saved to the specified library.

   \[
   \text{icc2\_shell}\> \text{commit\_block} \; \text{-library} \; \text{leon3s\_ndm} \; \text{leon3s}
   \]

In this example, the command creates a new physical hierarchy for the block named leon3s and writes the block to the leon3s\_ndm library. The GUI updates to show outline views for the committed blocks.

---

### Creating Block Placement Abstracts

After committing the logical hierarchy cells to physical hierarchy blocks, you can create block placement abstracts in preparation for block shaping and initial macro placement. Block placement abstracts are lightweight representations of physical hierarchy blocks that contain no timing information and enable faster initial macro placement. You must load the full netlist representation of the top-level design, minus the details for the committed blocks, before creating block placement abstracts.

To create block placement abstracts,

1. Set the constraint mapping file.

   \[
   \text{icc2\_shell}\> \text{shell} \; \text{cat} \; \text{split/mapfile} \\
   \text{leon3s} \; \text{SDC} \; \text{leon3s/top.tcl} \\
   \text{leon3s} \; \text{UPF} \; \text{leon3s/top.upf} \\
   \text{leon3s\_2} \; \text{SDC} \; \text{leon3s\_2/top.tcl} \\
   \text{leon3s\_2} \; \text{UPF} \; \text{leon3s\_2/top.upf} \\
   \text{leon3s\_3} \; \text{SDC} \; \text{leon3s\_3/top.tcl} \\
   \text{leon3s\_3} \; \text{UPF} \; \text{leon3s\_3/top.upf} \\
   \text{leon3mp} \; \text{SDC} \; \text{leon3mp/top.tcl} \\
   \text{leon3mp} \; \text{UPF} \; \text{leon3mp/top.upf}
   \]

   \[
   \text{icc2\_shell}\> \text{set\_constraint\_mapping\_file} \; \text{split/mapfile}
   \]

   The constraint mapping file is generated automatically by the \text{split\_constraints} command and specifies a list of blocks and their associated SDC and UPF constraint files. See \text{Split Constraints Output Files} for more information about the constraint files generated by the \text{split\_constraints} command.
2. Verify that the currently loaded top-level block representation is “outline”. If the current representation is already “design”, proceed to step 3.

```plaintext
icc2_shell> current_design
{leon3mp.ndm:leon3mp.outline}
```

The `current_design` command reports the current block name (leon3mp), representation (outline), and library that contains the design (leon3mp.ndm).

3. Expand the outline representation for the top-level block with the `expand_outline` command and save the library.

```plaintext
icc2_shell> expand_outline
...
icc2_shell> current_design
{leon3mp.ndm:leon3mp.design}
icc2_shell> save_lib -all
```

In this example, the `expand_outline` command reads the netlist and expands the top level. Note that the representation reported by the `current_design` command is changed from "outline" to "design".

**Note:**
You must maintain the original Verilog file in the same location referenced by the `read_verilog_outline` command, or make the file available in the `search_path`, for the command to function correctly.

4. Set the host options for distributed computing.

```plaintext
icc2_shell> set_host_options -name block_script host_settings
```

**Note:**
You can verify the current host option settings with the `check_host_options` command.

5. Create the block placement abstracts for all blocks by using distributed processing.

```plaintext
icc2_shell> create_abstract -placement -host_options block_script -all_blocks
Submitting job for block leon3s ...
Submitting job for block leon3s_2 ...
... Running distributed create_abstract ...
...
```

**Note:**
If your design contains multiple levels of physical hierarchy, the `create_abstract` command converts only blocks at the lowest level of hierarchy into abstract views; intermediate levels of hierarchy are kept as design views.
You can create block placement abstracts without using distributed processing by omitting the -host_options option. To create block placement abstracts for only a set of blocks, use the -blocks {blocks} option instead of the -all_blocks option.

---

**Moving the Origin of a Block**

By default, the {0 0} location for a given block is at the lower-left corner of the block, if the block is placed without rotation or other transformation. Note that if you reshape the block by moving an edge that includes the origin, the origin remains at the original location and does not move.

To move the origin for the current top-level block to a new location, use the move_block_origin command. The location you specify is relative to the current origin. For example, to move the origin of the current top-level block to the center of the block, where the current origin is at the lower-left corner and the bounding box for the block is {0 0} {1000 1000}, use the following command:

```
icc2_shell> move_block_origin -to {500 500}
```
Shaping Design Blocks and Macro Placement

The block shaping flow refines the boundary for the block based on the rough rectangular or rectilinear shape defined during hierarchy exploration. When creating the block shape, the tool considers design constraints such as target utilization for the block, channel width and keepout settings, while minimizing feedthroughs and interface wire lengths. You can create an optional block grid for your design; the tool aligns block shapes to the grid.
The flow to shape blocks is shown in Figure 7-1.

**Figure 7-1  Block Shaping Flow**

1. Set macro constraints
   \[\text{set\_macro\_constraints}\]
2. Create macro keepouts
   \[\text{create\_keepout\_margin}\]
3. Create shaping constraints
4. Shape blocks
   \[\text{shape\_blocks}\]
5. Create initial placement
   \[\text{create\_placement\_floorplan}\]

For more details, see the following topics:

- Setting Macro Constraints
- Creating Relative Placement Constraints for Macros and Macro Arrays
- Setting Macro Keepouts
- Shaping Blocks
- Creating the Initial Macro Placement
Setting Macro Constraints

Before block shaping, you can specify macro placement constraints and keepout margins for some or all blocks and macros. Placement constraints limit the orientations in which macros can be placed. To set macro placement constraints,

1. Create a collection of macro cells.
   The following example assigns a variable to a collection containing all macro cells in the design.
   
   ```
   icc2_shell> set macro_cells [get_cells -physical_context -filter "is_hard_macro && !is_physical_only" -quiet]
   ```

2. Specify the `set_macro_constraints` command with the appropriate arguments.
   The following example limits the placement orientation for all hard macros to R0 or R180.
   
   ```
   icc2_shell> set_macro_constraints -allowed_orientations {R0 R180} $macro_cells
   ```

3. (Optional) Use the `report_macro_constraints` command to verify the current constraint settings.
   Use one or more of the `-allowed_orientations`, `-preferred_location`, `-alignment_grid`, and `-align_pins_to_tracks` options to `report_macro_constraints` to limit the report to only those constraints.
   
   ```
   icc2_shell> report_macro_constraints
   ****************************************
   Report : report_macro_constraints
   Design : leon3mp
   ****************************************
   macro       allowed                   legal
   name        orientations              orientations
   ----------------------------------------------------
   u0_0/pd_switchable rf0_x2
   R0,R180                   all
   u0_0/pd_switchable rf0_x1
   R0,R180                   all
   ...
Use the following options with the `set_macro_constraints` command to constrain the placement of hard macros and I/O cells in the design:

**Table 7-1  Specifying Macro Constraint Options**

<table>
<thead>
<tr>
<th>To do this</th>
<th>Use this option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place macros by aligning the signal pins of the macro to the tracks for</td>
<td><code>-align_pins_to_tracks</code></td>
</tr>
<tr>
<td>the layer the pin shape belongs to</td>
<td></td>
</tr>
<tr>
<td>Align macros to a specified placement grid (placement grids are created</td>
<td><code>-alignment_grid grid_name</code></td>
</tr>
<tr>
<td>with the <code>create_grid</code> command)</td>
<td></td>
</tr>
<tr>
<td>Specify a list of allowed orientation values that further restrict legal</td>
<td><code>-allowed_orientations {list}</code></td>
</tr>
<tr>
<td>placement orientations specified in the library</td>
<td></td>
</tr>
<tr>
<td>Only apply the <code>-alignment_grid</code> or <code>-align_pins_to_tracks</code> option</td>
<td>`-alignment_orientation_set R0</td>
</tr>
<tr>
<td>specified in the same command to the specified macros placed in the</td>
<td></td>
</tr>
<tr>
<td>specified orientation set, either R0: {R0, R180, MX, MY}, or R90: {R90,</td>
<td></td>
</tr>
<tr>
<td>R270, MXR90, MYR90}</td>
<td></td>
</tr>
<tr>
<td>Assign a coordinate on the macro that must be on-grid</td>
<td><code>-alignment_point {x y}</code></td>
</tr>
<tr>
<td>Specify a preferred placement location for a specified macro, where x</td>
<td><code>-preferred_location {x_range y_range}</code></td>
</tr>
<tr>
<td><em>range</em> and y_range are values from zero to one</td>
<td></td>
</tr>
</tbody>
</table>
Creating Relative Placement Constraints for Macros and Macro Arrays

To constrain a macro or macro array to be placed with respect to another object in the design, use the \texttt{set\_macro\_relative\_location} command. The \texttt{create\_placement\_floorplan} command honors the constraints set by the \texttt{set\_macro\_relative\_location} command and places the target macro or macro array according to the specified offset distance from the anchor object. To create the constraint,

1. Set the constraint with the \texttt{set\_macro\_relative\_location} command.

\begin{verbatim}
icc2\_shell> \texttt{set\_macro\_relative\_location } \\
\hspace{1cm} \texttt{-target\_object {u\_m/u\_m\_pd/dsu0/x0\_mem0\_ram0\_1\_x1} \} } \\
\hspace{1cm} \texttt{-target\_orientation MXR90 -target\_corner bl } \\
\hspace{1cm} \texttt{-anchor\_object {u\_m/u\_m\_pd/dsu0/x0\_mem0\_ram0\_0\_x1} \} } \\
\hspace{1cm} \texttt{-anchor\_corner bl -offset {200 200}}
\end{verbatim}

2. (Optional) Verify that the constraint was set correctly with the \texttt{report\_macro\_relative\_location} command.

\begin{verbatim}
icc2\_shell> \texttt{report\_macro\_relative\_location} \\
.../x0\_mem0\_ram0\_1\_x1  MXR90  bl  .../x0\_mem0\_ram0\_0\_x1  bl  200  200
\end{verbatim}

\textbf{Figure 7-2} shows the layout after running the \texttt{set\_macro\_relative\_location} and \texttt{create\_placement\_floorplan} commands.
The `set_macro_relative_location` command constrains the placement of an object with respect to the anchor object. If the anchor object is not specified, the constraint is with respect to the parent-level design that contains the target object. The following anchor objects are supported:

- Hard Macro
- Macro Array
- Block
- IO Pad
- Block Pin
- Core Area
- Move Bound
- Voltage Area

You can change the relative location constraints with the following `set_macro_relative_location` command options:

- Specify which corner of the anchor object to use when creating the offset between the anchor object and target object; valid corner values are bl (bottom left), br (bottom right), tl (top left), and tr (top right).
  ```shell
  icc2_shell> set_macro_relative_location -anchor_corner bl
  ```

- Specify the instance name of the object to use as the reference location for the target object.
  ```shell
  icc2_shell> set_macro_relative_location -anchor_object {u1/u2}
  ```
• Specify the x- and y-offset for the target object with respect to the anchor object.

```bash
icc2_shell> set_macro_relative_location -offset {200 200}
```

• Specify the corner of the target macro to apply the constraint; valid corner values are the same as for the `-anchor_corner` option.

```bash
icc2_shell> set_macro_relative_location -target_corner tr
```

• Specify the instance name of the target macro to constrain.

```bash
icc2_shell> set_macro_relative_location -target_object {u3/u4}
```

• Specify the orientation of the target macro; valid orientations are R0, R90, R180, R270, MX, MXR90, MY, and MYR90. The orientation that you specify must be an allowed orientation for the macro.

```bash
icc2_shell> set_macro_relative_location -target_orientation R0
```

• Specify a scalable offset ratio, positive or negative, from the anchor object.

```bash
icc2_shell> set_macro_relative_location -offset_type scalable \
    -offset {0.5 0.5}
```

The scalable offset is calculated using the following formula:

```
target position = anchor position + offset * (scale_edge_length - used_length)
```

---

**Setting Macro Keepouts**

Keepout margins maintain spacing around blocks and macros and help prevent congestion and DRC errors. The following example creates a collection of all hard macros, then uses the `create_keepout_margin` command to create four types of keepout margins:

```bash
icc2_shell> set all_hm [get_cells -hierarchical \
    -filter "is_hard_macro==true"]
icc2_shell> create_keepout_margin -type hard \
    -tracks_per_macro_pin 0.05 $all_hm
icc2_shell> create_keepout_margin -type hard_macro \
    -tracks_per_macro_pin 0.5 $all_hm
icc2_shell> create_keepout_margin -type routing_blockage \
    -layers "M2 M3 M4" -tracks_per_macro_pin 0.05 $all_hm
icc2_shell> create_keepout_margin -type soft \
    -tracks_per_macro_pin 1.0 $all_hm
```

The `create_keepout_margin` command supports several options to control the keepout margins. To create a keepout margin around the outside of the block or macro, use the `-outer {left bottom right top}` option. To create a keepout margin around the inside of the block or macro, use the `-inner {left bottom right top}` option. To create a keepout margin based on a specified number of wire tracks, use the
-tracks_per_macro_pin  tracks option. When you use this option, the command multiplies the number of tracks you specify by the number of pins on that side of the macro to calculate the keepout margin size. You can further restrict the size of the keepout by specifying the -min_padding_per_macro size and -max_padding_per_macro size options together with the -tracks_per_macro_pin option. To specify the layers on which to apply a route_blockage keepout, use the -layers {layer_list} option.

The tool supports four types of keepout margins: hard, soft, hard_macro, and route_blockage. Table 7-2 summarizes these four types.

<table>
<thead>
<tr>
<th>Keepout Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard (default)</td>
<td>Prevents standard cells and hard macros from being placed within the keepout margin. The placement blockage moves together with the macro.</td>
</tr>
<tr>
<td>soft</td>
<td>Prevents standard cells and hard macros from being placed within the keepout margin. However, the placer can move cells into the keepout during optimization.</td>
</tr>
<tr>
<td>hard_macro</td>
<td>Prevents the placement of hard macros or other hard macro keepout margins within the keepout margin. This type affects only hard macro placement and ensures a minimum channel between hard macros. The entire macro and keepout are kept within the site array. You can use this keepout type to allocate space for routing to the pins of the macro.</td>
</tr>
<tr>
<td>route_blockage</td>
<td>Prevents the power and ground (PG) router from inserting via arrays that block access to macro pins. Requires a list of layers to include in the blockage.</td>
</tr>
</tbody>
</table>

**Shaping Blocks**

After committing the logical hierarchy cells to physical hierarchy blocks, you can create a rough placement of the physical hierarchy blocks. The tool shapes and places the physical blocks, including power domains and voltage areas, based on the specified utilization, channel size and constraints.

To place and shape the committed physical hierarchy blocks,

1. Set the block placement options with the set_shaping_options command.

   icc2_shell> set_shaping_options -keep_top_level_together true

   Any options set with the set_shaping_options command remain active during the session. You can use the report_shaping_options command to validate the current shaping settings.
2. Shape and place the blocks with the `shape_blocks` command.

```plaintext
icc2_shell> shape_blocks -channels true
```

In this example, the `-channels true` option directs the command to insert channels between blocks. Note that `shape_blocks` command options are not persistent. Figure 7-3 shows the floorplan before and after block shaping.

![Figure 7-3 Before (left) and After (right) Block Shaping](image)

During block shaping, the tool minimizes the number of feedthroughs and minimizes interface wire lengths. The command uses macro packing to place the top-level macros. The tool also aligns the block shape to the block grid, if specified. If you created block placement abstracts as described in Creating Block Placement Abstracts, you can use the abstracts to reduce the runtime required for placement.

3. Report any violations that occurred during block placement with the `report_block_shaping` command.
The `report_block_shaping` command produces a quality of results (QoR) report for the current block shaping result. Use the command to check for block overlaps, blocks outside the core boundary, excessive flyline crossings. The command also produces estimates for the number of unaligned pins and net detours.

4. Repeat the flow from step 1 to adjust the shaping options and produce a higher quality result.

The `shape_blocks` command supports different options to modify and refine the shaping result. Use the `-channels true | false` option to specify whether the command inserts channels between different blocks or between the blocks and the core boundary. Use the `-constraint_file file_name` option to apply utilization, preferred locations, grouping and fixed shape constraints for each block. Use the `-incremental congestion_driven | target_utilization_driven` option to apply incremental shaping with a priority on congestion or target utilization.

---

### Creating Block Shaping Constraints

To specify preferred block and voltage area placement locations for the `shape_blocks` command, create a block shaping constraint file and specify the file with the `-constraint_file` option. The tool sizes and places the specified regions based on the target utilization, channel size, and other constraints you specify in the file.

The shaping constraints file supports the following syntax:
# Global shaping constraints
utilization { target: 0.2; }
guard_band { width: 0; height: 0; }
channel_size { size=5; }

# TOP constraints
# block constraints cannot be nested
block TOP {
  # Define channel constraints for TOP
  # and blocks within TOP
  # Channel constraints can be defined for
  # block_inst, voltage_area, group, or current_block
  channel_size {
    # Define constraint for TOP/U1
    block_inst U1;
    # Default channel is 20 micron
    # You can also specify "min = 20, max = 20"
    size = 20;
    # Left and right channels are between 30 and 50 microns
    left, right: min = 30, max = 50;
  }
  # Create a guard band constraint for voltage area VA1
  # of 50 microns, and set the voltage area boundary
  # to {{0 0} {300 600}}
  voltage_area VA1 {
    contents: voltage_area VA2, block_inst U3;
    # Define spacing around voltage area VA2
    guard_band {
      width: 50; height: 50;
    }
    boundary {
      type: rigid;
      shape: {{0 0} {300 600}};
    }
  }
  # Define spacing around any other voltage area
  # beneath TOP
  guard_band {
    width: 40;
    height: 40;
  }
}
# Define a group that contains a set of blocks or
# voltage areas within TOP
define_group GROUP1 {
  # Organize blocks and voltage areas in a row or column
  # Legal directions are:
  # east (from left to right)
  # west (from right to left)
  # north (from bottom to top)
  # south (from top to bottom)
  arrange_in_array {
    contents: block_inst U1, block_inst U2, voltage_area VA3;
    direction: east;
  }
  # Specify a relative placement location
  # for instance U4 in the middle of TOP block,
  # alignment_point is optional
  arrange_in_box {
    block_inst U4;
    location: x = 0.5, y = 0.5;
    alignment_point: x = 1, y = 0;
  }
  # aspect_ratio specifies the relative width
  # and height. Use aspect_ratio together with
  # arrange_in_box
  aspect_ratio {
    block_inst U1;
    target: width = 1, height = 2;
  }
}

# BLOCK2 constraints
block BLOCK2 {
  # Reserve a space between the current block
  # boundary and any regions shaped inside the block
  boundary_channel_size {
    current_block;
    left: min = 3, max = 5;
    right, bottom: min = 5, max = 10;
  }
  # Specify a 50% target utilization
  # and a maximum of 70%
  utilization {
    target: 0.5;
    max: 0.7;
  }
}

The following example uses the shaping constraints file to organize blocks in the design. The shaping constraints file specifies two groups that contain five blocks each. The five
blocks are stacked horizontally, and the two groups are stacked vertically. A guard band of 100 is applied between the blocks.

```bash
icc2_shell> shell cat shaping.con
# Shaping constraint file
channel_size { size=100; }
block top {
  define_group UPPER {
    arrange_in_array {
      contents: block_inst u0, block_inst u1, block_inst u2,
      block_inst u3, block_inst u4;
      direction: east;
    }
  }
  define_group LOWER {
    arrange_in_array {
      contents: block_inst u5, block_inst u6, block_inst u7,
      block_inst u8, block_inst u9;
      direction: east;
    }
  }
  define_group TOP {
    arrange_in_array {
      contents: group UPPER, group LOWER;
      direction: south;
    }
  }
}
icc2_shell> shape_blocks -constraint_file shaping.con
```

Figure 7-4 shows the layout after running the `shape_blocks` command.
The following example specifies a fixed boundary for a block in the design. The boundary is specified with the `shape:` and `type: rigid` constraints. The coordinates are relative and the tool can move the shape to a new location in the parent block. The location of the block within the design is specified by `arrange_in_array` or `arrange_in_box` constraints.
# Shaping constraints with fixed block

channel_size { size=50; }

block block100a {
  boundary {
    type: rigid;
    shape: {{350 500} {350 1360} {550 1360}  
             {550 900} {850 900} {850 500}};
  }
}

block top {
  define_group UPPER {
    arrange_in_array {
      contents: block_inst u0, block_inst u1, block_inst u2,  
                block_inst u3, block_inst u4;
      direction: east;
    }
  }

  define_group LOWER {
    arrange_in_array {
      contents: block_inst u5, block_inst u6, block_inst u7,  
                block_inst u8, block_inst u9;
      direction: east;
    }
  }

  define_group TOP {
    arrange_in_array {
      contents: group UPPER, group LOWER;
      direction: south;
    }
  }
}

Figure 7-5 shows the layout after running the shape_blocks command. Note the shape of the block100a block, instantiated in the design as instance u0.
Creating the Block Grid

For designs that contain multiply instantiated blocks (MIBs), you must create a block grid to enable the tool to properly align MIB instances with other design objects. Proper MIB alignment ensures that objects can be successfully pushed down into blocks while maintaining alignment. During block shaping with the `shape_blocks` command, the tool aligns MIB instances with the block grid.

The block grid can be derived from power plan strategy or derived from existing site rows in the floorplan. You can also define the grid manually by specifying the origin and grid spacing. After creating the grid, you must associate the block references with a block grid to ensure that all instances align with the grid in the same way.
To create the block grid from existing site rows and power ground strategy settings,

1. Use the `create_pg_mesh_pattern` and `set_pg_strategy` commands to define the power grid (it is not necessary to instantiate the grid with the `compile_pg` command).

2. Create the grid by using the `create_grid` command with the `-site_rows` and `-pg_strategy` options.

   ```shell
   icc2_shell> create_grid grid1 -type block \
   -site_rows [get_site_rows] -pg_strategy s_mesh1
   Calculating block grid:
   ...
   {grid1}
   ```

   If your design contains a site array instead of site rows, use the following command to create the grid:

   ```shell
   icc2_shell> create_grid grid1 -type block \
   -site_arrays [get_site_array] -pg_strategy s_mesh1
   Calculating block grid:
   ...
   {grid1}
   ```

3. (Optional) Verify that the grid was created correctly with the `report_grids` command.

   ```shell
   icc2_shell> report_grids
   ****************************************
   Report : report block grid
   Design : leon3mp
   ****************************************
   Auto Block grid grid1:
   Step in X direction: 20.6720 micron
   Step in Y direction: 26.7520 micron
   X offset: 19.6200 micron
   Y offset: 13.7680 micron
   Allowed orientations: R0 MY MX R180
   Derived from site rows and PG strategies.
   Strategies Used       Layers:
   s_mesh1              all layers
   1
   ```

4. Associate the block grid with the MIB references by using the `set_block_grid_references` command.

   ```shell
   icc2_shell> set_block_grid_references -designs {leon3s} \
   -grid grid1
   ```

   Alternatively, you can manually create the block grid by specifying the x- and y- offsets and step sizes for the grid. Use the `report_grids` and `set_block_grid_references` commands to verify the grid and associate the grid with the MIB references. Create the grid manually as follows:
creating_grid -type block grid2 -x_step 20.6720 \
(grid2)

Note that you do not need to specify the power planning patterns or strategy if you create the grid manually.

Creating the Initial Macro Placement

After shaping the blocks, you can create an initial global macro placement with the create_placement command. To create the initial macro placement,

1. (Optional) Set the host options for the creating the floorplan-level placement with the create_placement command.

   icc2_shell> set_host_options -name distributed ...

   Use the set_host_options command to create the setting for distributed processing.

2. (Optional) Review the settings for the application options for the placer and make any necessary changes.

   You can use the report_app_options plan.place* command to report the current placer-related settings. Use the set_app_options command to change a setting.

3. Use the create_placement command with the -floorplan option to perform global macro placement.

   icc2_shell> create_placement -floorplan -host_options distributed

4. Validate the placement with the report_placement command.

   icc2_shell> report_placement -physical_hierarchy_violations all \
-wirelength all -hard_macro_overlap

   Report: report_placement
   Design: leon3mp
   ****************************************

   Wire length report (all)
   ================
   wire length in design leon3mp: 2267874.946 microns.
   number of nets with unassigned pins: 1512
   wire length in design leon3mp
   (see through blk pins): 3602323.139 microns.
5. (Optional) If the design contains overlaps, generate a complete list of overlapping instances with the \texttt{report\_placement} command.

\begin{verbatim}
icc2_shell> report_placement -verbose high > report.txt
\end{verbatim}

You can examine the report that is created by the \texttt{report\_placement} command and investigate the overlaps.

6. (Optional) To remove the placement created with the \texttt{create\_placement} command, use the \texttt{reset\_placement} command.

\begin{verbatim}
icc2_shell> reset_placement
\end{verbatim}

The \texttt{create\_placement} command supports a number of options to control how placement is performed. Use the \texttt{-floorplan} option to perform an initial, coarse placement with fast runtime. Use the \texttt{-effort low | medium | high} option to specify the level of effort to apply to the placement and tradeoff runtime and quality of results. To enable distributed processing, use the \texttt{-host\_options} option.

To create a congestion-driven placement, use the \texttt{-congestion} options with the \texttt{-floorplan} option. When you use the \texttt{-congestion} option, you can also control the level of effort with the \texttt{-congestion\_effort low | medium | high} option. Congestion-driven placement can perform three types of congestion reduction:

- Reduce the congestion over and between the macros only
- Reduce congestion between standard cells
- Reduce congestion both between macros and between standard cells

To specify which type of congestion reduction is used, specify the \texttt{plan.place.congestion\_driven\_mode} application option with the value \texttt{macro}, \texttt{std\_cell}, or \texttt{both}. The following example specifies that congestion reduction is applied only between hard macros:
To update the placement after changing the boundaries of macro blocks or voltage areas, use the \-incremental option with the create_placement command. If you specify create_placement -floorplan -incremental, the current macro placement is retained if the macros have a legal placement. If you specify create_placement -floorplan -congestion, the command can move macros to create wider channels for reduced routing congestion.

The create_placement command supports the following application options to control placement:

- **plan.place.floorplan.auto_generate_blockages**: Controls whether to automatically create soft placement blockages in channels between hard macros.
- **plan.place.floorplan.auto_generate_blockages_smooth_rectilinear**: Controls whether slightly rectilinear macros are considered to be the shape of their rectangular bounding box when determining blockages.
- **plan.place.auto_generate_hard_blockage_channel_width**: Specifies the maximum channel width and height when automatically creating a hard blockage.
- **plan.place.auto_generate_soft_blockage_channel_width**: Specifies the maximum channel width and height when automatically creating a soft blockage.
- **plan.place.auto_tl_max_density**: Controls whether to allow clumping of standard cells during top-level placement of a design that contains blocks.
- **plan.place.congestion_driven_mode**: Specifies the type of cell to consider for congestion reduction: macros, standard cells, or both.
- **plan.place.default_keepout**: Controls whether the tool creates a default hard_macro type keepout for all macros.
- **plan.place.place_inside_blocks**: Controls whether the placer places cells within blocks, or only at the top level.
- **plan.macro.auto_macro_array_max_height**: Specifies the maximum height of an automatically generated macro array.
- **plan.macro.auto_macro_array_max_num_cols**: Specifies the maximum number of columns in an automatically generated macro array.
- **plan.macro.auto_macro_array_max_num_rows**: Specifies the maximum number of rows in an automatically generated macro array.
- **plan.macro.auto_macro_array_max_width**: Specifies the maximum width of an automatically generated macro array.
plan.macro.auto_macro_array_minimize_channels: For macros with pins on only one side, specifies whether macros are flipped to abut the pins with the pins of neighboring macros, or flipped to abut the sides without pins.

plan.macro.auto_macro_array_size: Specifies the amount of array packing to use during hard macro placement: none, low, medium, or high.

plan.macro.create_temporary_pg_grid: Specifies a command or script to create a temporary power grid for congestion estimation.

plan.macro.grid_error_behavior: Specifies what to do if the tool fails to assign grids to macros: continue, macro_place_only, or quit.

plan.macro.macro_place_only: Specifies whether to stop the placement when macro locations are decided.

plan.macro.pin_shape_track_match_threshold: Specifies the maximum pin shape width, on a per layer basis, to consider for color matching.
Performing Power Planning

Power planning, which includes power network routing and power network analysis, is required to create a design with good power integrity. A design with a robust power and ground (PG) grid reduces IR drop and electromigration by providing an adequate number of power and ground pads and rails. The power plan can be used to assess the routing resources consumed by the power nets and to determine the impact on routability due to the power plan. You can experiment with different power plans or fine-tune the existing power plan by modifying the command option settings and regenerating the power plan.
The flow to create a power plan for your design is shown in Figure 8-1.

Figure 8-1  Pattern-Based Power Planning Flow

The pattern-based power planning flow separates the physical implementation details (layer, width, spacing) of the power ring and mesh from the regions of the design where the structures are inserted. In a typical design, you run the flow multiple times. The first pass defines and creates the power rings, the second pass defines and creates and the power mesh, and so on.

The create Pg ring pattern, create Pg mesh pattern, create Pg macro conn pattern, and create Pg std cell conn pattern commands create pattern specifications that define the physical implementation details of the power plan and associate a width, spacing, offset, and via rule specification with the metal layers.
in your design. After defining the patterns, they can be easily reused among different power regions in the design.

The `set_pg_strategy` command assigns a pattern specification to specific power nets and regions in the design. You can also define the offset for the start of the pattern, the power strap extension specification, and the blockage specification with this command.

You can create and implement complex via rules during pattern-based power planning. The `set_pg_via_master_rule` command defines the contact code, offset for the contact code within the via structure, cut spacing, and other parameters for the custom via structure. Power via masters are associated with a specific power plan strategy by specifying the `-via_rule` option with the appropriate `create_pg_*_pattern` command.

The `create_pg_*_pattern` and `set_pg_strategy` commands specify the power plan, but they do not modify the design. After creating the pattern and strategy definitions, use the `compile_pg` command to instantiate the power structures into the design. To make changes to the power plan after creating the power rings and meshes, remove the rings and meshes with the `compile_pg -undo` command and make changes to the via rules, pattern specifications, or power plan strategies.

After instantiating the power plan, you can validate the integrity of the power plan by checking for DRC and connectivity violations. Use the `check_pg_drc` command to report any power structures that violate the routing design rules. Use the `check_pg_connectivity` command to check the physical connectivity of the power network.

These topics are described in the following sections:

- Creating and Connecting Power Nets
- Defining PG Via Masters
- Creating Power and Ground Ring Patterns
- Creating Power and Ground Mesh Patterns
- Creating Power and Ground Macro Connections
- Creating Power and Ground Standard Cell Rails
- Creating Channel and Alignment Power Straps
- Creating Complex Composite Patterns
- Defining Power Plan Regions
- Setting the Power Plan Strategy
- Creating Via Rules Between Different Strategies
- Instantiating the Power Plan
• Pattern-Based Power Network Routing Example
• Inserting and Connecting Power Switches
• Trimming the Power and Ground Mesh
• Checking Power Network Connectivity and DRC
• Performing Distributed Power Network Routing
Creating and Connecting Power Nets

To begin pattern-based power planning, define the power and ground nets in your design and connect them to the power and ground pins. If you use UPF to describe your power network, you can skip these steps.

1. Define the power and ground nets with the `create_net` command.

   ```
   icc2_shell> create_net -power VDD
   {VDD}
   icc2_shell> create_net -ground VSS
   {VSS}
   ```

2. Connect the power and ground nets to power and ground pins with the `connect_pg_net` command.

   ```
   icc2_shell> connect_pg_net -net VDD [get_pins -physical_context *VDD]
   icc2_shell> connect_pg_net -net VSS [get_pins -physical_context *VSS]
   ```

   If your design has a UPF description of the power network, you can specify `connect_pg_net -automatic` to derive the power and ground nets directly from the power domain specification and perform the connection.

Defining PG Via Masters

If your design requires a via master that is not defined in the technology file, you can create a via master for your power network with the `set_pg_via_master_rule` command. You can use the new via master in the power network by referencing it with the `-via_rule` option of the `create_pg_composite_pattern`, `create_pg_macro_conn_pattern`, `create_pg_mesh_pattern`, or `create_pg_ring_pattern` command.

To create a new via master for the power network, use the `set_pg_via_master_rule` command as shown in the following example.

```
icc2_shell> set_pg_via_master_rule VIA78_2x2 -contact_code VIA78 \
       -via_array_dimension {2 2} -offset {3 1}
```
To use the new rule in your design, specify the via rule name after the `via_master` keyword for the `create_pg_composite_pattern`, `create_pg_macro_conn_pattern`, `create_pg_mesh_pattern`, and `create_pg_ring_pattern` commands. For example, the following command uses the VIA78_2x2 via rule defined by the previous command.

```plaintext
icc2_shell> create_pg_mesh_pattern mesh_pat
   -layers {layer_spec}
   -via_rule {
       {{layers: M6} {layers: M7} {via_master: default}}
       {{layers: M7} {layers: M8} {via_master: VIA78_2x2}}
   }
```

### Creating Power and Ground Ring Patterns

The ring pattern specifies the horizontal and vertical layer names, ring width values, spacing values, vias, and corner bridging to use to create the power and ground ring. You can create ring patterns around the core, design blocks, macros, power and ground regions, or rectilinear polygons that you specify. To define the ring pattern, use the `create_pg_ring_pattern` command as follows:

```plaintext
icc2_shell> create_pg_ring_pattern ring_pat -horizontal_layer M7 
   -horizontal_width {10} -horizontal_spacing {2} 
   -vertical_layer M8 -vertical_width {10} 
   -vertical_spacing {2} -corner_bridge false
```

Alternatively, use the Task Assistant to define the ring pattern.

1. Select Task > Task Assistant in the GUI.
3. Click the Ring tab.
4. Enter the pattern name, layer names, widths, and other ring parameters in the Task Assistant.
5. Click Apply to create the pattern.

You can also define the width, spacing, and other values for the `create_pg_ring_pattern` command by using parameters. To create a parameter, include the `-parameters` option followed by the parameter name, and replace the value for the command option with a parameter name. Parameters are identified by the parameter name preceded by the (`@`) character.

The values for the parameters remain undefined until they are set with the `set_pg_strategy` command. Parameters are set in the same order they are specified by the `-parameters` option. Using this approach, you can reuse the pattern specified by the command and apply different parameter values with the `set_pg_strategy` command.
The following example uses parameters to define the metal layer, width, spacing, and corner bridge settings for the ring pattern.

```bash
icc2_shell> create_pg_ring_pattern ring_pat -horizontal_layer @hlayer \
   -horizontal_width {@hwidth} -horizontal_spacing {@hspace} \
   -vertical_layer @vlayer -vertical_width {@vwidth} \
   -vertical_spacing {@vspace} -corner_bridge @cbridge \
   -parameters {hlayer hwidth hspace vlayer vwidth vspace cbridge}

icc2_shell> set_pg_strategy ring_strat -core \
   -pattern {{name: ring_pat} {nets: {VDD VSS}} \ 
   [offset: 3 3] [parameters: {M7 10 2 M8 10 2 true}]}) \
   -extension [{stop: design_boundary}]
```

---

## Creating Power and Ground Mesh Patterns

The mesh pattern specifies the horizontal and vertical layer names, metal width values, metal spacing values, metal pitch, vias, and wire trimming to use to create the power and ground mesh. To define the mesh pattern, use the `create_pg_mesh_pattern` command as follows:

```bash
icc2_shell> create_pg_mesh_pattern mesh_pat -layers {
   {vertical_layer: M8} {width: 5}
   [spacing: interleaving] [pitch: 32])
   {vertical_layer: M6} {width: 2}
   [spacing: interleaving] [pitch: 32])
   {horizontal_layer: M7} {width: 5}
   [spacing: interleaving] [pitch: 28.8])}
   -via_rule {
   [layers: M6} [layers: M7} [via_master: default])
   [layers: M8} [layers: M7] [via_master: VIA78_3x3])}
```

Alternatively, use the Task Assistant to define the mesh pattern.

1. Select Task > Task Assistant in the GUI.
3. Enter the pattern name and define the via rule.
4. Select the layer, direction, width, spacing, pitch, and offset values.
5. Click the entry in the table row to edit the setting.
6. Click Preview to view the Tcl command in the Tcl Command box.
7. Click Run Tcl Command to create the mesh pattern.
You can also define the settings for the power mesh by using parameters as described in Creating Power and Ground Ring Patterns. The following example uses parameters to define the metal width values for the `create_pg_mesh_pattern` command.

```
icc2_shell> create_pg_mesh_pattern mesh_pat -layers {
    {vertical_layer: M8} {width: @width8}
    {spacing: interleaving} {pitch: 32})
    {vertical_layer: M6} {width: @width6}
    {spacing: interleaving} {pitch: 32})
    {horizontal_layer: M7} {width: @width7}
    {spacing: interleaving} {pitch: 28.8})
    -via_rule {
        {layers: M6} {layers: M7} {via_master: default})
        {layers: M8} {layers: M7} {via_master: VIA78_3x3})
    -parameters {width6 width7 width8}
icc2_shell> set_pg_strategy mesh_strat -core -pattern {
    {name: mesh_pat} {nets: {VDD VSS}}
    {parameters: {32 28.8 32}}0
```

Creating Power and Ground Macro Connections

The macro connection pattern specifies the power and ground nets, routing direction, metal layers, layer width, layer spacing, metal pitch and other information to create connections to macros. To define the macro connection pattern, use the `create_pg_macro_conn_pattern` as in the following example.

```
icc2_shell> create_pg_macro_conn_pattern macro_pat -nets {VDD VSS} \\
    -direction horizontal -width 1 -layers M5 -spacing minimum \\
    -pitch 5 -pin_conn_type long_pin
```

Alternatively, use the Task Assistant to define the standard cell rail pattern.

1. Select Task > Task Assistant in the GUI.
3. Click the HM scattered pin, HM long pin, or HM ring pin tab depending on the type of hard macro pin connection.
4. Enter the pattern name, layer, width, and other information in the form.
5. Click Apply to create the pattern.

You can also define the settings for the standard cell rails by using parameters as described in Creating Power and Ground Ring Patterns. The following example uses parameters to define the metal width for the standard cell rail pattern, and assigns values to the parameters with the `set_pg_strategy` command.
Creating Power and Ground Standard Cell Rails

The standard cell rail pattern specifies the metal layers, rail width, and rail offset to use to create the power and ground rails for the standard cell rows. To define the standard cell rail pattern, use the `create_pg_std_cell_conn_pattern` as in the following example.

```
icc2_shell> create_pg_std_cell_conn_pattern rail_pat -layers {M1} \  -rail_width {0.2 0.2}
icc2_shell> set_pg_strategy rail_strat -core \  -pattern {{name: rail_pat} {nets: VDD VSS} \  {parameters: {0.2 0.2}}}
```

Alternatively, use the Task Assistant to define the standard cell rail pattern.

1. Select Task > Task Assistant in the GUI.
3. Click the Std cell tab.
4. Enter the pattern name, layer name, rail width, and offset in the Task Assistant.
5. Click Apply to create the pattern.

If you do not define a rail width, the command uses the width of the standard cell power and ground pins as the width.

You can also define the settings for the standard cell rails by using parameters as described in Creating Power and Ground Ring Patterns. The following example uses parameters to define the metal width for the standard cell rail pattern, and assigns values to the parameters with the `set_pg_strategy` command.

```
icc2_shell> create_pg_std_cell_conn_pattern rail_pat -layers {M1} \  -rail_width {@wtop @wbottom} -parameters {wtop wbottom}
icc2_shell> set_pg_strategy rail_strat -core \  -pattern {{name: rail_pat} {nets: VDD VSS} \  {parameters: {0.2 0.2}}}
```
Creating Channel and Alignment Power Straps

The special pattern defines the structure for the following types of power ground straps:

- Straps placed in channels
- Straps and vias inserted to connect and align terminals
- Straps inserted to align power switches
- Straps inserted to align physical-only and filler cells

Only one pattern type can be specified within a single create_pg_special_pattern command. To define these patterns, use the create_pg_special_pattern command as in the following example.

```
icc2_shell> create_pg_special_pattern channel_pattern \
   -insert_channel_straps {{layer: M6} {direction: vertical} \ 
   {width: 2} \ 
   {channel_between_objects: {macro placement_blockage voltage_area} \ 
   }}
```

Alternatively, use the Task Assistant to define the special pattern.

1. Select Task > Task Assistant in the GUI.
2. Select PG Planning > Special Pattern in the Task Assistant.
3. Click the Channel straps tab. The Switch alignment, Terminal alignment, Physical cell alignment, and Max stdcell distance tabs are also available.
4. Enter the information for the form, such as the library name, layer name, and other settings.
5. Click Apply to create the pattern.

Creating Complex Composite Patterns

This section uses examples to create complex PG patterns using the create_pg_composite_pattern command.

- Creating Center-Aligned Vias
- Creating Bridging Straps
- Creating Via Bridging Straps
• Creating Tapering Straps
• Creating a Checkerboard Via Pattern

Creating Center-Aligned Vias

The following example creates horizontal power straps on layer M3 and vertical power straps on layer M8. Stacked vias connect the horizontal and vertical straps, and the vias are pushed toward the center of each VDD/VSS vertical power strap pair to save routing resources beneath the straps. The `create_pg_wire_pattern` command defines the base pattern, and the `create_pg_composite_pattern` command defines the parameters for the horizontal and vertical PG routes.

*Figure 8-2  Power Straps with Center-Aligned Vias*

This example defines two via master rules: `via_5x6_shift_right` and `via_5x6_shift_left`. For each via, the `-offset` option is applied to shift the via from center by 0.8 microns.

The `-via_rule` option of the `create_pg_composite_pattern` command specifies which via is placed at each intersection of layers M3 and M8 as shown in *Figure 8-3*. In this example, the `-add_patterns` option instantiates two `create_pg_wire_pattern` definitions: the vertical power strap pattern on layer M8 (`pattern_id: 1`), and the horizontal power strap pattern on layer M3: (`pattern_id: 2`). The `create_pg_composite_pattern` also refers to four net ids for the two PG nets: VSS (`net_id: 1` and `4`) and VDD (`net_id: 2` and `3`).
The via_5x6_shift_right via is inserted at the intersections of \texttt{pattern\_id: 1} and \texttt{net\_id: 1} of the first pattern (layer M8 and net VSS), and \texttt{pattern\_id: 2} and \texttt{net\_id: 2} of the second pattern (layer M3 and net VSS); location (A) in the figure. The via_5x6_shift_right via is also inserted at \texttt{pattern\_id: 1} and \texttt{net\_id: 3} (layer M8 and net VDD), and \texttt{pattern\_id: 2} and \texttt{net\_id: 1} (layer M3 and net VDD); location (D) in the figure. Similarly, the remaining statements define vias at locations (B) and (C).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{annotated_vias}
\caption{Annotated Center-Aligned Vias}
\end{figure}

The commands to create this power plan are as follows:

\begin{verbatim}
set_pg_via_master_rule via_5x6_shift_right \\
  -via_array_dimension {5 6} -offset {0.8 0}
set_pg_via_master_rule via_5x6_shift_left \\
  -via_array_dimension {5 6} -offset {-0.8 0}
create_pg_wire_pattern wire_base -layer @l -direction @d \\
  -width @w -spacing @s -pitch @p -parameters {l d w p s}
# For the vertical straps on M8 defined by pattern\_id: 1
# - net\_id: 1 is VSS
# - net\_id: 2 is VDD
# - net\_id: 3 is VDD
# - net\_id: 4 is VSS
#
# For the horizontal straps on M3 defined by pattern\_id: 2
# - net\_id: 1 is VDD
# - net\_id: 2 is VSS
\end{verbatim}
create_pg_composite_pattern center_aligned_vias -nets {VDD VSS} \ 
  -add_patterns [ \ 
    {pattern: wire_base} {nets: VSS VDD VDD VSS} \ 
    {parameters: {M8 vertical 3 30 [1 4 1]}{offset: 4}} \ 
    {pattern: wire_base} {nets: VDD VSS} \ 
    {parameters: {M3 horizontal 2 15 2}}{offset: 4}} \ 
  ] \ 
  -via_rule [ \ 
    {{pattern_id: 1}{net_id: 1}} \ 
    {{pattern_id: 2}{net_id: 2}} \ 
    {via_master: via_5x6_shift_right}} \ 
    {{pattern_id: 1}{net_id: 2}} \ 
    {{pattern_id: 2}{net_id: 1}} \ 
    {via_master: via_5x6_shift_left}} \ 
    {{pattern_id: 1}{net_id: 3}} \ 
    {{pattern_id: 2}{net_id: 1}} \ 
    {via_master: via_5x6_shift_right}} \ 
    {{pattern_id: 1}{net_id: 4}} \ 
    {{pattern_id: 2}{net_id: 2}} \ 
    {via_master: via_5x6_shift_left}} \ 
    {intersection: undefined}{via_master: NIL} \ 
  ] \ 
set_pg_strategy core_patterns \ 
  -pattern {{name: center_aligned_vias} \ 
    {nets: VDD VSS } {offset: {5 5}}} -core \ 

cOMPile_Pg -strategies core_patterns

Creating Bridging Straps

The following example creates bridging straps to connect parallel vertical power straps. This power plan contains the following features:

- Three VDD power straps of width 2 microns, 4 microns, and 2 microns form a group of vertical straps on layer M6
- Three M6 VSS power straps of width 2 microns, 4 microns, and 2 microns also form a group of vertical straps on layer M6
- The groups of three vertical VDD straps are connected together with a 2 micron wide strap on layer M3 with a pitch of 15 microns, length of 12 microns, and 5x1 stacked via arrays between M6 and M3
- The groups of three vertical VSS straps are connected together with a 2 micron wide strap on layer M5 with a pitch of 15 microns, length of 12 microns, and 5x3 via arrays between M5 and M6

The bridging straps and via structures used to connect the straps are shown in Figure 8-4 and Figure 8-5.
The `set_pg_via_master_rule` command defines the via rules for the 5x1 and 5x3 via arrays. The `create_pg_composite_pattern` command defines a power plan that uses two distinct wire patterns: one pattern called `wire_base` used to define the VDD and VSS vertical straps, one pattern called `segment_base` used to define both the M3 layer straps that connect VDD and the M5 layer straps that connect VSS. The wire patterns are referenced using the `{pattern: wire_base}` and `{pattern: segment_base}` statements.

The `-via_rule` option to `create_pg_composite_pattern` defines what vias are inserted at different metal intersections in the power plan. The `{{layers: M6} {layers: M5} {via_master: via65_5x3}}` statement inserts a `via65_5x3` via master between any intersection of layers M6 and M5 in the power plan.
The -via_rule statement \{{\text{intersection: undefined}} \ {\text{via_master: NIL}}\} ensures that only the specified vias are inserted in the power plan. For designs that contain many wire patterns, this statement ensures that no vias are inserted between undefined layer pairs; by default, vias are created for undefined layer pairs.

The commands used to implement this power plan are as follows:

```plaintext
set_pg_via_master_rule via63_5x1 \
  -contact_code {VIA34 VIA45 VIA56} \
  -via_array_dimension {5 1}

set_pg_via_master_rule via65_5x3 \
  -contact_code {VIA56} \
  -via_array_dimension {5 3}

create_pg_wire_pattern wire_base -layer @l -direction @d \
  -width @w -spacing @s -pitch @p -parameters {l d w p s}

create_pg_wire_pattern segment_base -layer @l -direction @d \
  -width @w -low_end_reference_point 0 \
  -high_end_reference_point @len -pitch @p \
  -parameters {l d w len p}

create_pg_composite_pattern bridging_straps -nets {VDD VSS} \
  -add_patterns { 
    {{pattern: wire_base} {nets: VDD VSS} \
     {parameters: {M7 horizontal 2 60 2}}{offset: 13}} \n    {{pattern: wire_base} {nets: VDD VDD VSS VSS VSS} \
     {parameters: {M6 vertical {2 4 2 4 2} 45 {2 2 4 2 2}}} \
     {offset: 4}} \n    {{pattern: segment_base} {nets: VDD} \
     {parameters: {M3 horizontal 2 12 {45 15}}} {offset: {3 2}}}} \
    {{pattern: segment_base} {nets: VSS} \
     {parameters: {M5 horizontal 2 12 {45 15}}} {offset: {19 10}}}) \ 
  -via_rule { 
    {{layers: M7}{layers: M6} {via_master: default}} \
    {{layers: M6}{layers: M5} {via_master: via65_5x3}} \
    {{layers: M6}{layers: M3} {via_master: via63_5x1}} \
    {{intersection: undefined} {via_master: NIL}} 
  }

set_pg_strategy core_patterns \
  -pattern {{name: bridging_straps} \
    {nets: VDD VSS} {offset: {5 5}}} -core

compile_pg -strategies core_patterns
```
Creating Via Bridging Straps

The following example creates a complex, shifted via bridging structure to connect the vertical VDD straps on layer M8 to the horizontal VDD straps on layer M3. The horizontal and vertical VSS straps use a standard stacked via to connect layers M8 and M3.

Figure 8-6 Composite Bridging Patterns

For the VDD power straps, a 13x13 via array connects the vertical straps on layer M8 with the short horizontal metal segment on layer M7. On the same M7 segment, a smaller 6x12 stacked via array connects M7 with the horizontal VDD strap on layer M3.
The composite pattern technique allows you to shift the via locations and open additional vertical routing tracks to reduce routing jogs and improve timing. The via array is typically inserted in the center of intersection between the M7 layer metal segment and the M3 layer horizontal strap. In this example, the via is shifted to the right with the `-offset {3 0}` option of the `set_pg_via_master_rule` command. The offset is shown in Figure 8-8. Vertical routing tracks on layers M6 and M4 that are beneath the M8 layer VDD straps are not used for PG and can be used for signal routing.
The `set_pg_via_master_rule` command defines the via rule for the 12x6 stacked via array. The `create_pg_composite_pattern` command defines a composite pattern that uses two distinct wire patterns: `wire_base` and `segment_base`. The `wire_base` pattern specifies the vertical straps on M8 and the horizontal straps on M3 for VDD and VSS. The `segment_base` pattern specifies the short 10 micron wide horizontal segments for net VDD on layer M7. The wire patterns are referenced using the `{pattern: wire_base}` and `{pattern: segment_base}` statements.

The `-via_rule` option specifies where vias are inserted for each layer intersection. Consider the following definition for the right-shifted via73_12x6_shift_right stacked via:

```
{[pattern_id: 2]{net_id: 1}}
{[pattern_id: 3]{net_id: 1}}
{via_master: via73_12x6_shift_right} {between_parallel: true}
```

In the previous definition, `{pattern_id: 2}{net_id: 1}` refers to the second pattern specified by the `-add_patterns` option: `{pattern: wire_base} ... {parameters: {M3 horizontal ...}}`, and the first net specified by the `-nets` option: VDD. Likewise, `{pattern_id: 3}{net_id: 1}` refers to the third pattern definition: `{pattern: segment_base} ... {parameters: {M7 horizontal ...}}`, and the first net: VDD. Based on this definition, the tool inserts a via73_12x6_shift_right stacked via at the intersection of the M3 and M7 straps. The `{between_parallel: true}` statement allows the tool to insert the vias between parallel straps.

The commands to create this power plan, including the via master rules, are as follows:

```
set_pg_via_master_rule via73_12x6_shift_right \
 -contact_code {VIA34 VIA45 VIA56 VIA67} \
 -via_array_dimension {12 6} -offset {3 0}
create_pg_wire_pattern wire_base -layer @l -direction @d \
 -width @w -spacing @s -pitch @p -parameters {l d w p s}
create_pg_wire_pattern segment_base -layer @l -direction @d \
 -width @w -low_end_reference_point 0 \
 -high_end_reference_point @len -pitch @p \
 -parameters {l d w len p}
```
create_pg_composite_pattern via_bridging -nets {VDD VSS} \\  -add_patterns { \  {{pattern: wire_base} {nets: VDD VSS} \  {parameters: {M8 vertical 4 15 2}}{offset: 4}} \  {{pattern: wire_base} {nets: VDD VSS} \  {parameters: {M3 horizontal 2 20 4}}{offset: 4}} \  {{pattern: segment_base} {nets: VDD} \  {parameters: {M7 horizontal 4 10 {30 20}}} {offset: {2 4}}} \  } \  -via_rule { \  {{pattern_id: 1}{net_id: 1}} \  {{pattern_id: 2}{net_id: 1} {via_master: NIL}} \  {{pattern_id: 1}{net_id: 2}} \  {{pattern_id: 2}{net_id: 2} {via_master: default}} \  {{pattern_id: 1}{net_id: 1}} \  {{pattern_id: 3}{net_id: 1} {via_master: default}} \  {{pattern_id: 2}{net_id: 1}} \  {{pattern_id: 3}{net_id: 1} \  {via_master: via73_12x6_shift_right} {between_parallel: true}} \  {{intersection: undefined} {via_master: NIL}} \  } \\

set_pg_strategy core_patterns \  -pattern {{name: via_bridging} \\  {nets: VDD VSS } {offset: {5 5}}} -core

compile_pg -strategies core_patterns

---

**Creating Tapering Straps**

The following example creates a power structure with narrowed power straps at the center of the die. Vertical VDD and VSS straps are created on layer M6 and horizontal VDD and VSS straps are created on layer M7. At the periphery of the die, horizontal M7 straps are 10 microns wide and vertical M6 straps are 8 microns wide. At the center of the die, the M7 straps narrow to 5 microns and M6 straps narrow to 4 microns. The green square annotates the bounding box {{500 500} {1000 1000}}.
The power plan begins with three `create_pg_wire_pattern` commands to create three wire pattern definitions. These commands define the pattern layer, direction, width, and spacing in a particular region of the routing boundary. The `first_wire` wire pattern defines a pattern from the left and bottom of the routing boundary as specified by the `-extend_low` boundary option, and extends to the point specified by the `-high_end_reference_point @pt1` argument. The `center_wire` wire pattern extends from the `-low_end_reference_point @pt1` and ends at the `-high_end_reference_point @pt2` points. The `last_wire` pattern starts from the point specified by
-low_end_reference_point @pt2 and extends to the -extend_high boundary, which is the top or right edge of the routing boundary.

The two create_pg_composite_pattern commands are used to define two different relationships between the patterns. The create_pg_composite_pattern tapering_base command associates the previous wire patterns with the left, center, and right areas of the routing boundary, and the bottom, middle, and top areas respectively. The second command, create_pg_composite_pattern tapering_straps, specifies the actual values for the PG route layer, direction, side width, center width, side spacing, and center spacing using the previous create_pg_composite_pattern tapering_base definition to create horizontal and vertical PG routes.

Note that the tapering_straps composite pattern uses another composite pattern called tapering_base. Composite pattern can be defined hierarchically to create more complex patterns for a demanding design requirement.

The set_pg_strategy command passes the actual x- and y-locations where the tapered straps should begin and end, together with the signal names for the PG straps. These values are specified by using Tcl variables.

The commands used to implement this power plan are as follows.

```
create_pg_wire_pattern first_wire -layer @l -direction @d \
    -width @w -spacing @s -extend_low boundary \
    -high_end_reference_point @pt1 -parameters {l d w s pt1}
create_pg_wire_pattern center_wire -layer @l -direction @d \
    -width @w -spacing @s -low_end_reference_point @pt1 \
    -high_end_reference_point @pt2 -parameters {l d w s pt1 pt2}
create_pg_wire_pattern last_wire -layer @l -direction @d \
    -width @w -spacing @s -low_end_reference_point @pt2 \
    -extend_high boundary -parameters {l d w s pt2}
```
create_pg_composite_pattern tapering_base \ 
  -parameters {layer dir side_w center_w side_s center_s pt1 pt2} \ 
  -add_patterns {{pattern: first_wire} \ 
  {parameters: {@layer @dir @side_w @side_s @pt1}} \ 
  {pattern: center_wire} \ 
  {parameters: {@layer @dir @center_w @center_s @pt1 @pt2}} \ 
  {pattern: last_wire} \ 
  {parameters: {@layer @dir @side_w @side_s @pt2}} \ 
}

create_pg_composite_pattern tapering_straps \ 
  -parameters {h_pt1 h_pt2 v_pt1 v_pt2} \ 
  -add_patterns { \ 
  {pattern: tapering_base} \ 
  {parameters: {M7 horizontal 10 5 2 7.5 @h_pt1 @h_pt2}} \ 
  {pitch: {0 30}}} \ 
  {pattern: tapering_base} \ 
  {parameters: {M6 vertical 8 4 2 6 @v_pt1 @v_pt2}} \ 
  {pitch: {25 0}}} \ 
}

set x_start 500
set x_end 1000
set y_start 500
set y_end 1000
set c_offset 5
set_pg_strategy core_patterns -core \ 
  -pattern [list {name: tapering_straps} \ 
  {nets: VDD VSS } \ 
  {parameters: $x_start $x_end $y_start $y_end} \ 
  [list offset: [list 5 $c_offset]]]

compile_pg -strategies core_patterns

Creating a Checkerboard Via Pattern

The following example creates horizontal power straps on layer M5 and vertical power straps on layer M4 for net VDD. Vias are inserted at alternating locations and form a checkerboard pattern. The create_pg_wire_pattern command defines the base pattern, and the create_pg_composite_pattern command defines the parameters for the horizontal and vertical PG routes.
The single `create_pg_wire_pattern` command creates a wire pattern named `wire_base`. The `create_pg_composite_pattern` command instantiates the `wire_base` wire pattern two times; once for the vertical straps on layer M4 and once for the horizontal straps on layer M5. Each pattern defines a pair of nets, `{VDD VDD}`. The pitch between the pairs of nets is 20 microns; the two nets are evenly distributed between the pitch due to the `-spacing interleaving` option of the `create_pg_wire_pattern` command. Note that `interleaving` is passed as a parameter by the `{parameters: {M4 vertical 2 20 interleaving}}` statement in the `create_pg_composite_pattern` command.

The `-via_rule` option specifies which via is placed at each intersection of layers M4 and M5. The following figure is annotated with the `pattern_id` and `net_id` numbering for this power plan. In this example, `{pattern_id: 1}` refers to the first pattern in the `-add_patterns` option: the vertical power straps on layer M4. `{pattern_id: 2}` refers to the second pattern: the horizontal power straps on layer M5. `{pattern_id: 1} {net_id: 1}` refers to the first net following the `nets:` keyword listed for the first pattern specification.

Based on this specification, vias are placed at the intersections of `{pattern_id: 1} {net_id: 1}` and `{pattern_id: 2} {net_id: 1}`. Vias are also placed at the intersections of `{pattern_id: 1} {net_id: 2}` and `{pattern_id: 2} {net_id: 2}`. At other locations, `{via_master: NIL}` specifies that no via is placed at those intersections. The `{intersection: undefined} {via_master: NIL}` statement specifies that no other vias should be inserted anywhere in the power plan.
The commands to create this power plan are as follows:
Defining Power Plan Regions

In the pattern-based power network routing flow, power plan regions define areas of the design used to create the power rings and power straps to form the power network. To create a power plan region, use the `create_pg_region` command as shown in the following example.

```
icc2_shell> create_pg_region region1 \   
    -exclude_macros {macro1 macro2} -core -expand -35 \   
    -macro_offset 35 {region1}
```

Alternatively, use the Task Assistant to define the power plan region.

1. Select Task > Task Assistant in the GUI.
2. Select PG Planning > PG Region in the Task Assistant.
3. Enter the region name, region area, expansion parameters, and other information in the Task Assistant.
4. Click Preview to display the Tcl command you specified.

5. Click Apply to create the power plan region.

To create a power plan region that is bounded by a polygon, specify the -polygon option followed by the coordinates of the polygon. The following example creates a power plan region named region_polygon bounded by the coordinates (2000,2000), (4000,2000), (4000,4000), and (2000,4000).

```tcl
icc2_shell> create_pg_region region_polygon \\
    -polygon {{2000 2000} {4000 2000} {4000 4000} {2000 4000}}
```

To create a power plan region that excludes specified macros, use the -exclude_macros option and specify the list of macros to exclude. You can create an offset around the macro by including the -macro_offset option. In the following example, the tool creates an indentation in the region to exclude the specified macro.

```tcl
icc2_shell> create_pg_region region_exclude -core \\
    -exclude_macros {u_l/u_m/ahbram0/aram_1/x0_t0} -macro_offset 50
```

Figure 8-13  Region Excluding a Macro

To create a PG region that surrounds macros in the design, use the -group_of Macros option. You can create an offset by including the -expand option. In the following example,
the tool creates a region around the specified macros and expands the region by 50 microns.

```
icc2_shell> set macros {u_l/u_m/ahbram0/aram_0/x0_t0
    u_l/u_m/dsu0/x0/mem0_ram0_0 x0 x0_t0
    u_l/u_m/ahbram0/aram_2/x0_t0
    u_l/u_m/dsu0/x0/mem0_ram0_0_x1_x0_t0}

icc2_shell> create_pg_region region_surround -group_of_macros $macros \ 
    -expand 50
{region_surround}
```

**Figure 8-14  Region Surrounding a Group of Macros**

If you run the `create_pg_region` command and specify an existing region name, the command deletes the existing region, reuses the region name, and creates the new region definition. You can also create a copy of an existing region and make modifications to the region copy by using the `-update` option. The next example copies the `region_surround` power plan region from the previous example, expands the region, and removes the jogs.

```
icc2_shell> create_pg_region region_copy -update region_surround \ 
    -expand 75 -remove_jog {expand: 50}
```
To list all the power plan regions in the design, use the \texttt{report\_pg\_regions} command. To list only a single region, specify the region name after the command. The following example reports the currently defined power plan regions in the design.

\begin{verbatim}
icc2\_shell> \texttt{report\_pg\_regions}
Region: region\_surround
Points: \{(2202.360 3092.270) (2202.360 3575.890) \ldots \}
Region: region\_copy
Points: \{(2127.360 3017.270) (2127.360 4647.760) \ldots \}
\end{verbatim}

To delete a power plan region, use the \texttt{remove\_pg\_regions} command followed by the region name. To remove all regions, use the \texttt{remove\_pg\_regions -all} command. The following example removes the power plan region named \texttt{region\_copy}.

\begin{verbatim}
icc2\_shell> \texttt{remove\_pg\_regions region\_copy}
PG region region\_copy is removed.
\end{verbatim}

\section*{Setting the Power Plan Strategy}

After defining the ring, mesh, macro connection, and standard cell rail patterns for your design, you can associate the patterns with the power plan regions or other areas of the design by using the \texttt{set\_pg\_strategy} command. The command also defines the pattern offset, how the patterns extend past the boundaries of the power plan region or area, and routing blockages where the pattern is not created.
To define the power plan strategy, use the `set_pg_strategy` command as in the following example.

```
icc2_shell> set_pg_strategy ring_strat -core \
   -pattern {{name: ring_pattern} {nets: {VDD VSS}} \ 
   {offset: {3 3}} {parameters: {M7 10 2 M8 10 2 true}}} \ 
   -extension {{stop: design_boundary}}
```

Alternatively, use the Task Assistant to define the standard cell rail pattern.

1. Select Task > Task Assistant in the GUI.
3. Enter the strategy name, PG nets, pattern name, routing area, and other settings in the Task Assistant.
4. Click Apply to apply the strategy.

In the previous example, the `set_pg_strategy` command defined a strategy that associated the ring_pattern pattern with the ring_strat strategy. The following example defines a strategy that associates the mesh_pattern pattern for the power mesh defined in Creating Power and Ground Mesh Patterns with the mesh_strat strategy. The `-extension` option extends the power mesh to the outermost ring.

```
icc2_shell> set_pg_strategy mesh_strat -core \
   -extension {{stop: outermost_ring}} \ 
   -pattern {{pattern: mesh_pattern} {nets: {VDD VSS}}}
```

The following example defines a strategy that associates the macro_pattern macro connection pattern defined in Creating Power and Ground Macro Connections with the macro_strat strategy.

```
icc2_shell> set_pg_strategy macro_strat -core \ 
   -pattern {{pattern: hm_pattern} {nets: {VDD VSS}}}
```

The following example defines a strategy that associates the rail_pattern standard cell connection pattern defined in Creating Power and Ground Standard Cell Rails with the rail_strat strategy.

```
icc2_shell> set_pg_strategy rail_strat -core \ 
   -pattern {{pattern: std_cell_rail} {nets: VDD VSS}}
```

The following sections summarize the specification keywords and their function for the `-pattern`, `-blockage` and `-extension` options.
Strategy Settings for the -pattern Option

Arguments to the -pattern option define how the specified pattern is created in the design. The following list describes the available arguments. Note that not all arguments are valid for all pattern types.

- The {name: pattern_name} argument assigns the pattern name to use with this strategy. The name is the pattern name you created with the create_pg_*_pattern pattern_name command.

- The {nets: {net1 net2 ...}} argument specifies the power and ground net names for the strategy. You can also define placeholder names in the create_pg_*_pattern command and set the actual net names using this argument.

- The {offset: {x_offset y_offset}} argument specifies the x- and y-offset to apply when creating the first pattern. This argument is not used for scattered pin macro connection patterns or standard cell rail patterns.

- The {offset_start: boundary | {x y}} argument specifies the starting point for the pattern offset. This argument is not used for scattered pin macro connection patterns or standard cell rail patterns.

- The {parameters: value1 value2 ...} argument specifies the ordered list of values for the parameters defined with the -parameters option of the create_pg_*_pattern command.

- The {skip_sides: {side1 side2 ...}} argument defines the sides to omit when creating the power ring. This argument is valid only for ring patterns.

- The {side_offset: {side: number} {offset: offset}} argument defines the offset for each specified side in the ring pattern. This argument is valid only for ring patterns.

Strategy Settings for the -blockage Option

Arguments to the -blockage option define areas where the pattern should not be created. A blockage is defined with three arguments: nets, layers, and the area specification (voltage_areas, macros, polygon, blocks, or pg_regions). You can provide multiple blockage specifications within a single set_pg_strategy command. The following list describes the available arguments. Note that not all arguments are valid for all pattern types.

- The {layers: {layer1 layer2 ...}}: argument specifies the layers to avoid routing over the blockage.

- The {nets: {net1 net2 ...}} argument specifies the power and ground net names to avoid routing over the blockage. This list of nets is a subset of the nets specified by the -pattern option.
• The area is defined by the `pg_regions, blocks, macros, pg_regions, polygon, or voltage_areas` argument, followed by a list of elements or points.

---

**Strategy Settings for the -extension Option**

Arguments to the `-extension` option define how the power mesh extends beyond the specified power plan region or design area. The extension specification enables a connection between the power structure and other power structures in the design. An extension is defined with the `direction, layers, nets, side, and stop` arguments. You can define multiple extensions in the same `set_pg_strategy` command. The following list describes the available arguments. Note that some arguments are invalid for some commands.

• The `{direction: {L R T B}}` argument limits the direction to extend the power mesh to only the specified directions.

• The `{layers: {layer1 layer2 ...}}` argument limits the extension to only the specified layers.

• The `{nets: {net1 net2 ...}}` argument specifies the power and ground net names to avoid routing over the blockage. This list of nets is a subset of the nets specified by the `-pattern` option.

• The `{side: {side_a side_b ...}}` argument limits power ring or macro connection extension to only the specified sides.

• The `{stop: first_target | innermost_ring | outermost_ring | pad_ring | design_boundary | design_boundary_and_generate_pin | distance_in_microns}` argument defines the stopping point for the extension.

---

**Creating Via Rules Between Different Strategies**

A complex power plan might require that you connect combinations of several different power rings, meshes, power rails, and other structures. Use the `set_pg_strategy_via_rule` command to set the via insertion rules when inserting vias between different power plan strategies.

```
icc2_shell> set_pg_strategy_via_rule via_rule1 -via_rule {
    {'{[strategies: strat1] {layer: M2}}
    {'{[strategies: strat2] {layer: M3}}
    {'{via_master: VIA23_FAT}}
    {'{intersection: undefined} {via_master: nil}}
}"
```
The preceding example creates a via rule that inserts the VIA23_FAT via between shapes on layer M2 defined by strategy strat1 and shapes on layer M3 defined by strategy strat2. New vias are omitted between other metal layer intersections.

Alternatively, use the Task Assistant to define the via rule.

1. Select Task > Task Assistant in the GUI.
2. Select PG Planning > Create PG in the Task Assistant.
3. Click the Via rule tab.
4. Enter the rule name.
5. Select Advanced, click Define and define the details of the via rule.
6. Click Apply to create the via strategy.

Instantiating the Power Plan

The create_pg_*_pattern and set_pg_strategy commands define the power plan, but do not add power straps or rails to the design. To instantiate the power plan, use the compile_pg command with the -strategies option and specify the strategy name as in the following example.

```
icc2_shell> compile_pg -strategies ring_strat
... 
Successfully compiled PG.
```

The command instantiates the power plan defined by the specified strategy and checks for any DRC violations created by the power plan. In a typical flow, you create the ring, mesh, rail, or macro pattern, associate the pattern with a power plan strategy by using the set_pg_strategy command, instantiate the pattern with the compile_pg command, then repeat the flow for the next pattern type in the power plan.

If you specified the power plan incorrectly, or if the power plan contains an error, you can remove the power plan with the compile_pg -undo command as shown in the following example. The compile_pg -undo command removes only the power and ground network created by the most recent compile_pg command.

```
icc2_shell> compile_pg -undo
```

To ignore DRC violations while creating the power network, include the -ignore_drc option. The compile_pg -ignore_drc command creates the power network and reports DRC violations created by the power plan, but does not remove power straps or vias that create DRC violations.
Pattern-Based Power Network Routing Example

The following example is a complete pattern-based power network routing script.

```bash
# Create the power and ground nets and connections
create_net -power VDD
create_net -ground VSS
connect_pg_net -net VDD [get_pins -physical_context *VDD]
connect_pg_net -net VSS [get_pins -physical_context *VSS]

# Create the power and ground ring pattern
create_pg_ring_pattern ring_pattern -horizontal_layer @hlayer \ 
  -horizontal_width (@hwidth) -horizontal_spacing (@hspace) \ 
  -vertical_layer @vlayer -vertical_width (@vwidth) \ 
  -vertical_spacing (@vspace) -corner_bridge @cbridge \ 
  -parameters {hlayer hwidth hspace vlayer vwidth vspace cbridge}

# Set the ring strategy to apply the ring_pattern
# pattern to the core area and set the width
# and spacing parameters
set_pg_strategy ring_strat -core \ 
  -pattern {{name: Ring_pattern} {nets: {VDD VSS}} \ 
    {offset: {3 3}} {parameters: {M7 10 2 M8 10 2 true}}} \ 
  -extension {{stop: design_boundary}}

# Create the ring in the design
compile_pg -strategies ring_strat

# Define a new via rule, VIA78_3x3, for the power mesh
set_pg_via_master_rule VIA78_3x3 -contact_code VIA78 \ 
  -via_array_dimension {3 3}

# Create the power and ground ring mesh pattern
create_pg_mesh_pattern mesh_pattern -layers \ 
  {{vertical_layer: M8} {width: 5} \ 
    {spacing: interleaving} {pitch: 32}} \ 
  {{vertical_layer: M6} {width: 2} \ 
    {spacing: interleaving} {pitch: 32}} \ 
  {{horizontal_layer: M7} {width: 5} \ 
    {spacing: interleaving} {pitch: 28.8}}} \ 
  -via_rule \ 
    [{{layers: M6} {layers: M7} {via_master: default}} \ 
     {{layers: M8} {layers: M7} {via_master: VIA78_3x3}}]
```
# Set the mesh strategy to apply the mesh_pattern
# pattern to the core area. Extend the mesh
# to the outermost ring
set_pg_strategy mesh_strat -core -pattern {{pattern: mesh_pattern} 
{nets: {VDD VSS}}} -extension {{stop: outermost_ring}}

# Create the mesh in the design
compile_pg -strategies mesh_strat

# Create the power and ground ring mesh pattern
create_pg_mesh_pattern mesh_pattern -layers {
    {{vertical_layer: M8} {width: @width8} 
    {spacing: interleaving} {pitch: 32}}
    {{vertical_layer: M6} {width: @width6} 
    {spacing: interleaving} {pitch: 32}}
    {{horizontal_layer: M7} {width: @width7} 
    {spacing: interleaving} {pitch: 28.8}})
-via_rule {
    {{layers: M6} {layers: M7} {via_master: default}}
    {{layers: M8} {layers: M7} {via_master: VIA78_3x3}}
-parameters {width6 width7 width8}

# Set the mesh strategy to apply the mesh_pattern
# pattern to the core area. Extend the mesh
# to the outermost ring
set_pg_strategy mesh_strat -core -pattern {
    {pattern: mesh_pattern} {nets: {VDD VSS}}
    {parameters: {32 28.8 32}})
-extension {{stop: outermost_ring}}

# Create the mesh pattern in the design
compile_pg -strategies mesh_strat

# Create the connection pattern for macro
# power and ground pins
create_pg_macro_conn_pattern macro_pattern 
-pin_conn_type scattered_pin

# Set the macro connection strategy to
# apply the macro_pattern pattern to
# the core area
set_pg_strategy macro_strat -core 
-pattern {{pattern: macro_pattern} 
{nets: {VDD VSS}}}

# Connect the power and ground macro pins
compile_pg -strategies macro_strat
Inserting and Connecting Power Switches

In a design that contains shut-down power domains, you can insert power switches to control the power within the power domain. To create an array of power switches, use the **create_power_switch_array** command as follows:

```
icc2_shell> create_power_switch_array \
    -power_switch leon3_misc_sw \
    -y_pitch 3.344 -x_pitch 20.672
```

To create a power switch ring, use the **create_power_switch_ring** command as follows:

```
icc2_shell> create_power_switch_ring \
    -power_switch leon3_misc_sw \
    -x_offset 1.601 -y_offset 3.278 \
    -x_pitch 40 -y_pitch 40 \
    -prefix leon3_p0_sw_ring
```

Common options with the **create_power_switch_array** and **create_power_switch_ring** commands control how the commands insert power switches. Use the **-power_switch** option to specify the power switch strategy name. Use the **-lib_cell** option to explicitly specify the name of the power switch library cell to insert. Use the **-prefix** option to specify an instance name prefix for the switch cells. Use the **-voltage_area**, **-voltage_area_shape**, or **-boundary** option to define the shape or bound

---

# Create a new 1x2 via
set_pg_via_master_rule via16_1x2 -via_array_dimension {1 2}

# Create the power and ground rail pattern
create_pg_std_cell_conn_pattern rail_pattern -layers {M1}

# Set the power and ground rail strategy
# to apply the rail_pattern pattern to the
# core area
set_pg_strategy rail_strat -core \
    -pattern [{pattern: rail_pattern} {nets: VDD VSS}]

# Define a via strategy to insert via16_1x2 vias
# between existing straps and the new power rails
# specified by rail_strat strategy on the M6 layer
set_pg_strategy_via_rule rail_rule -via_rule {
    [{existing: strap} {layers: M6}]
    {strategies: rail_strat} {via_master: via16_1x2}]
    {intersection: undefined} {via_master: nil})

# Insert the new rails
compile_pg -strategies rail_strat -via_rule rail_rule
within which to place the switch cells. Use the \(-x\_pitch, -y\_pitch, -x\_offset, \) and \(-y\_offset\) options to control the switch placement spacing and offset.

Use the \(-orient\) option to specify the orientation of the switch cell. In a power switch array, the same orientation is applied throughout the array. In a power switch ring, switch cells placed on the top and right edges are rotated accordingly. The \(-pattern\) option specifies the name of the pattern to be used to place switch cells. The pattern is defined by the \texttt{set_power_switch_placement_pattern}\ command. Use the \(-snap\_to\_site\_row\) option to control whether the power switches are snapped to site rows. By default, power switch cells are snapped to site rows.

For power switch arrays, use the \(-siterow\_offset\) and \(-siterow\_pitch\) options to place the power switches based on site rows instead of \(y\)-offset and \(y\)-pitch. Instead of using \(-x\_offset\) and \(-y\_offset\), you can specify the startpoint of the power switch array with the \(-offset\_start\) option. The \(-checkerboard\ even | odd\) option creates a checkerboard placement pattern. Use either the \(-pg\_straps\) or \(-pg\_strategy\) option to specify the vertical power and ground wires with which to align the power switches. If needed, use the \(-align\_marker\) option to control the horizontal offset of the power switch cells to the wires. Otherwise, the wires are aligned to corresponding power switch pins.

For power switch rings, use the \(-continue\_pattern\) option to avoid restarting the placement pattern on adjacent edges and continue the pattern specified by the \(-pattern\) option. Use the \(-start\_point\) and \(-end\_point\) options to specify the starting and ending points of a partial ring. Use the \(-filler\_cells\) option to specify the list of filler cells to be placed into gaps in the power switch ring. Use the \(-inner\_corner\_cell, \) \(-inner\_corner\_cell\_orient, -outer\_corner\_cell,\) and \(-outer\_corner\_cell\_orient\) options to specify the corner library cell name and orientation to use when placing cells at the inner (concave) and outer (convex) corners of the ring.

The \(-pattern\) option specifies a predefined pattern to use when placing power switches. The pattern is defined by the \texttt{set_power_switch_placement_pattern}\ command, and is replicated across the power switch ring or throughout the power switch array. For power switch arrays, pattern-based power switch insertion automatically connects SLEEP and ACKNOWLEDGE pins within the pattern and avoids complicated Tcl scripts to create the connections. To create a power switch placement pattern, use the \texttt{set_power_switch_placement_pattern}\ command as follows:

\begin{verbatim}
icc2_shell> set_power_switch_placement_pattern -name pattern1 \
  -driver buf1 -direction vertical \ 
  -placement_type array \ 
  -pattern {buf1 {SPACE 10} {buf2 {SPACE 5} 3}} \ 
  -connect_mode daisy
\end{verbatim}

Options to the \texttt{set_power_switch_placement_pattern}\ command control how the pattern is defined. Use the \(-name\) option to specify the name of the pattern. Use the \(-placement\_type\ ring | array\) option to specify whether the pattern applies to a power switch ring or array. Use the \(-driver\) option to assign a driver cell for the pattern. Use the \(-direction\ horizontal | vertical\) option to specify the placement direction. Use the
-connect_mode hfn | daisy option to specify whether the power switches are connected in high-fanout mode (hfn) or daisy-chain mode. Use the -pattern option to specify the actual placement pattern. Use the -port_net_name option to set the net name prefix for the nets in the connection pattern.

After inserting the power switch ring or array, you can connect the control pins of the power switches with the connect_power_switch command as follows:

```bash
icc2_shell> connect_power_switch \\
    -source [get_pins u0_2/pd_switchable/sw_ctl] -mode hfn \\
    -port_name p2_sd -object_list [get_cells u0_2/.* -regexp \\
    -filter "ref_name == HEAD2X16_HVT"]
```

Options to the connect_power_switch command control how the power switches are connected. Use the -source option to specify the control signal within the power domain for the power switch. Use the -mode hfn | daisy | fishbone option to specify the connection strategy for the power switches as shown in Figure 8-16. Use the -port_name option to specify the base name of the switch ports when a new port is created. Use the -object_list option to specify the switch cells to connect. See the man page for additional options.

Figure 8-16  Power Switch Fanout Configurations
Setting Resistance Values for Power Switch Cells

To set resistance values for power switch cells, use the `plan.pna.power_switch_resistance` application option and specify pairs of instance names and resistance values.

The following example sets a resistance value of 10 ohms on cell HEAD2X16_HVT and a resistance value of 15 ohms on cell HEAD2X16_LVT.

```shell
icc2_shell> set_app_options 
   -name plan.pna.power_switch_resistance 
   -value {{HEAD2X16_HVT 10} {HEAD2X16_LVT 15}}
```

Alternatively, you can set the resistance attribute directly on cell instances. The following example sets the `power_switch_resistance` attribute to specify a resistance value of 10 ohms on instances of switch cell SWITCH_1 and a resistance value of 20 ohms on instances of switch cell SWITCH_2.

```shell
icc2_shell> set_attribute {$switch_cell_1} 
   -name power_switch_resistance -value 10
icc2_shell> set_attribute {$switch_cell_2} 
   -name power_switch_resistance -value 20
```

Note that the attribute has a higher priority than the application option.

Trimming the Power and Ground Mesh

Your power plan might contain protruding or dangling shapes after creating the power mesh. To remove these extra shapes, use the `trim_pg_mesh` command. The command trims all power and ground routes up to the closest via metal enclosure on the same layer for the same net. Figure 8-17 shows a section of a design before and after running the `trim_pg_mesh` command.
Use options with the `trim_pg_mesh` command to control how the mesh is trimmed.

- Trim only the specified net names or layers with the `-nets` and `-layers` options. You can also use the `-shapes` option and specify a collection of net shapes. The following two commands are equivalent.

```bash
icc2_shell> trim_pg_mesh -nets {VDD} -layers {M5}
icc2_shell> trim_pg_mesh -shapes [get_shapes -filter "layer_name==M5 && owner.name==VDD"]
```

- Trim only nets of the specified type, including ring, stripe, `lib_cell_pin_connect`, `macro_pin_connect`, and `macro_conn`.

```bash
icc2_shell> trim_pg_mesh -types {ring stripe}
```

- Trim the net to the wire boundary (default) or the via boundary with the `-trim_to` option.

```bash
icc2_shell> trim_pg_mesh -trim_to target_wire # left image
icc2_shell> trim_pg_mesh -trim_to via # right image
```
The IC Compiler II tool supports commands to check the connectivity of the power ground network and verify the DRC compliance. The following sections describe the commands used to perform power network validation.

### Checking Power Network Connectivity and DRC

The IC Compiler II tool supports commands to check the connectivity of the power ground network and verify the DRC compliance. The following sections describe the commands used to perform power network validation.

### Checking Power Network Connectivity

Use the `check_pg_connectivity` command to perform a detailed connectivity check and ensure that each block, macro, pad, and standard cell is connected to the power ground network as follows:
The `check_pg_connectivity` command provides options to control how the check is performed. Use the `-nets` option to limit the check to a specific collection of power ground nets. Use the `-write_connectivity_file` option to write out the results of the connectivity check to a text file. Use the `-check_macro_pins one | all | none` option to check the power ground connections to macro cells. The `one` argument checks for at least one connection, `all` specifies that all power ground pins must be connected, and `none` specifies that this check is skipped. Use the `-check_block_pins one | all | none` option to check the power ground connection for block pins. Use the `-check_pad_pins one | all | none` option to check the power ground connection for pad pins. Use the `-check_std_cell_pins true | false` option to enable or disable power ground network checking for standard cells.

When you specify the `-write_connectivity_file` option, the tool writes out a file containing detailed results of the connectivity check. For each power net and each subnetwork for each net, the tool reports the number of wires, vias, standard cells, hard macros, I/O pads, terminals, and hierarchical blocks connected to the net, and the number of floating connections for each element. A short example from the output file is as follows:

```
********************************
*** Net VDD Connectivity Stats ***
********************************
Number of disjoint networks: 1
*********(main network)**********
   Number of wires: 4826
   Number of vias: 377204
   Number of std cells: 914
   ...
   Number of floating std cells: 3633
     u_m/u_power_controller_top/all_sd_driver,
     u_m/u_power_controller_top/p0_sd_driver,
     ...
```
To create a collection of floating objects in the power network, save the returned value of the `check_pg_connectivity` command to a variable. The following example runs the `check_pg_connectivity` command, saves the collection of floating objects to the `floating_objects` variable, and selects the floating objects in the GUI.

```
icc2_shell> set floating_objects [check_pg_connectivity]
Loading cell instances...
Number of Standard Cells: 9392
... 
icc2_shell> change_selection $floating_objects
```

Validating DRC in the Power Network

To check the DRC compliance of routing objects related to the power and ground network, use the `check_pg_drc` command. For a given net, the command checks for DRC violations with respect to the following related objects:

- Pins of nets of any type, including pins that are unassigned
- Routing blockages and keepouts that might restrict power ground routing
- Metal and via routing shapes that are not assigned to any net
- Routing objects associated with other power ground nets that are not connected to the current net being evaluated
- Routing objects of clock nets

The following example uses the `check_pg_drc` command to verify DRC compliance for objects affected by the VDD power net. The command writes out a text report and a database that you can review by using the error browser in the GUI.

```
icc2_shell> check_pg_drc -nets VDD
Total number of errors found: 4375008
  1406 shorts on CO
  25082 shorts on M1
...
Description of the errors can be seen in
gui error set "DRC_report_by_check_pg_drc"
```

You can use multiple threads to shorten the runtime for the `check_pg_drc` command. Use the `set_host_options -max_cores n` command to set the maximum number of processing cores to use when checking power ground DRC.

The `check_pg_drc` command supports options to control which nets and which area of the design the tool checks for DRC violations. Use the `-nets` option to limit the checking to a specified collection of power and ground nets. Use the `-ignore_clock_nets true` option to skip checking of routing objects for clock nets. Use the `-coordinates` option to limit the check to a specified bounding box. Use the `-bottom_layer` and `-top_layer` options to limit
the check to a specified range of layers. Use the -no_gui option to skip writing the error browser database for the GUI. Use the -output option to write out a text version of the DRC check result to a file. A short example of the DRC text file is as follows:

===============
Error type: insufficient space between two same-net via-cuts
Violated rule: Cut-Min_spacing_Rule
Layer: VIA5
Minimal spacing: 0.07
Actual spacing: 0.05 (south - north)
Spacing box: {3408.42 1571.75} {3408.46 1571.8}
Via #1: box = {3408.41 1571.7} {3408.46 1571.75}
    net = VDD
Via #2: box = {3408.42 1571.8} {3408.47 1571.85}
    net = VDD
===============
Error type: illegal overlap of net-shapes
...

You can review the errors reported by the check_pg_drc command in the GUI. Select Error > Error Browser in the GUI, and select DRC_report_by_check_pg_drc as the Data Name. The tool loads the DRC data in the Error Browser.

By default, power network routing ignores signal routes during DRC checking to maximize performance. To enable the compile_pg, create_pg_strap and create_pg_vias commands to validate the DRC correctness of power straps and vias with respect to signal routes, set the plan.pgroute.honor_signal_route_drc application option to true as follows:

icc2_shell> set_app_options -name plan.pgroute.honor_signal_route_drc -value true

Performing Distributed Power Network Routing

The IC Compiler II tool supports commands to characterize and write out the current power network strategy and pattern settings, then generate the power network through distributed processing. Distributed power network routing reduces the runtime needed to create a power network for your design.
Preparing for Distributed Power Network Routing

To perform distributed power network routing, you must write out the current power plan constraints with the `characterize_block_pg` command and use the `run_block_compile_pg` command to create the power plan by using a distributed computing network.

The following example writes out the current strategy and pattern settings to the pgout directory and defines a script to be run by the `run_block_compile_pg` command.

```
icc2_shell> characterize_block_pg \
-init_pg_script pg_script.tcl -output_directory pgout
... Characterize block PG constraints.
... Characterize PG constraints for block u0_0 in script ./pgout/leon3s_pg.tcl.
... Characterize top-level PG constraints.
... Characterize PG constraints for top-level design in script ./pgout/top_pg.tcl.
Create PG mapping file ./characterize_output/pg_mapfile
```

The `characterize_block_pg` command writes a power plan script for each reference block. Each script contains commands such as `create_pg_mesh_pattern`, `create_pg_ring_pattern`, and so on to create the power ground connection patterns. The scripts also contain `set_pg_strategy` commands to associate the patterns with different areas of the design, and `set_pg_via_master_rule` and `set_pg_strategy_via_rule` commands to define rules to create vias.

If you specified the `-compile_pg_script pg_script` option, the tool writes out a map file that associates the power planning scripts with the reference blocks. The `pg_script` file can contain setup commands and application options for creating the power ground routing, but it should not contain via master rules, patterns, strategies, or strategy via rule commands. During distributed power network creation, the map file is associated with the design by using the `set_constraint_mapping_file` command. In this example, the map file written by the `characterize_block_pg` commands is as follows:

```
leon3s   PG_CONSTRAINT   ./leon3s_pg.tcl
leon3s_2  PG_CONSTRAINT  ./leon3s_2_pg.tcl
leon3s_3  PG_CONSTRAINT  ./leon3s_3_pg.tcl
leon3s   COMPILE_PG     ./pg_script.tcl
leon3s_2  COMPILE_PG    ./pg_script.tcl
leon3s_3  COMPILE_PG    ./pg_script.tcl
leon3mp  COMPILE_PG     ./pg_script.tcl
```
Performing Distributed Power Network Routing

After generating the power plan constraint files, use the run_block_compile_pg command to create the power plan by using distributed processing. To create the power plan,

1. Assign the map file with the set_constraint_mapping_file command.
   
   icc2_shell> set_constraint_mapping_file pgout/pg_mapfile

2. Set the host options for distributed processing with the set_host_options command.
   
   icc2_shell> set_host_options -name block_script host_settings

3. Run distributed power planning with the run_block_compile_pg command.
   
   icc2_shell> run_block_compile_pg -host_options block_script

   ... Create PG for blocks : leon3s leon3s_2 leon3s_3
   ... found instance u0_1 for design leon3s
   ... Submitting job for block leon3s ...
   ... All tasks created.
   ... Successfully ran distributed PG creation.

   1

The run_block_compile_pg command uses distributed processing to create the power plan. If you omit the -host_options option, the run_block_compile_pg command process each block sequentially.
Performing Global Planning

To improve global routing between design blocks during the design planning phase, the IC Compiler II tool supports operations to group similar nets, create global routing corridors, and push global route objects into blocks. Global planning enables you to better manage routing resources between blocks. You typically use global bus planning after running block shaping and macro placement.
The flow to perform global bus planning is shown in Figure 9-1.

Figure 9-1 Pin Assignment Flow

Create bundles and bundle constraints
(create_bundle)
(set_bundle_pin_constraints)

Create routing corridors
(create_routing_corridor)

Create targeted global routes
(route_group)

Add repeater cells
(add_buffer_on_route)

Push down repeater cells
(push_down_objects)

For more details, see the following topics:

- Creating Bundles and Bundle Constraints
- Creating Routing Corridors
- Creating Global Routes Within Routing Corridors
- Adding Repeater Cells
- Pushing Down Repeater Cells
Creating Bundles and Bundle Constraints

Net bundles are used to manage bus signals and other related signal nets. You can use bundles to group signals together, create a bundle object, and trace their flylines in the layout. Any net can be placed into a bundle. Signals within the bundle can be rearranged and sorted to create custom pin ordering. Bundles can be created based on standard bus naming patterns. Bundles can contain other bundles.

During pin placement, the `place_pins` command routes bundle nets together through the block edge and honors bundle constraints such as pin spacing, pin width, and so on.

To create an explicit bundle by specifying individual net names,

1. Use the `create_bundle` command and specify the bundle name and the signals in the bundle.
   
   ```
   icc2_shell> create_bundle \
   {Bundle1}
   ```

2. (Optional) Verify the net contents of the bundle with the `report_bundles` command.
   
   ```
   icc2_shell> report_bundles [get_bundles Bundle1]
   ```

   ```
   Begin bundle report ..
   Bundle NAME: 'Bundle1'
   Number of objects in bundle 4
   Bundle Type 'net'
   Objects in bundle
   Position = 0, net sd[0]
   ...
   ```

To automatically create bus bundles based on nets with similar prefixes,

1. Use the `create_bundles_from_patterns` command and specify the prefix of the net group, or omit the `-net_name_prefix` option to automatically create multiple net bundles for all net groups.
   
   ```
   icc2_shell> create_bundles_from_patterns -net_name_prefix ahbmi
   Information: There are 1 new bundles defined. (LED-102)
   {B_ahbmi}
   ```

2. (Optional) Verify the contents of the bundle with the `report_bundles` command.
   
   ```
   icc2_shell> report_bundles [get_bundles {B_ahbmi}]
   ```

   ```
   Begin bundle report ..
   Bundle NAME: 'B_ahbmi'
   Number of objects in bundle 24
   ...
   ```

The `create_bundles_from_patterns` command supports several options to control how the bundles are created. Use the `-net_name_prefix prefix` option to specify a set of net
names for a bundle. To specify the ordering of the net bundles, use the `-net_order
ascending | descending` option. To constrain the minimum and maximum number of nets
in a bundle, use the `-minimum_nets min` and `-maximum_nets max` options.

If you omit the `-net_name_prefix` option, the `create_bundles_from_patterns` command
creates a bundle by identifying signals that contain a bus index pattern. The supported bus
index patterns are \([i], (i), <i>, _i, :i\) and \({i}\), where "i" is the bus index. To limit the set of
recognized bus patterns, specify the `-no_braces`, `-no_brackets`, `-no_angle_brackets`,
`-no_underlines`, `-no_colons`, or `-no_parentheses` options.

To modify an existing bundle by adding or removing nets, use the `add_to_bundle` and
`remove_from_bundle` commands.

You can also use the Net Assistant dialog box in the GUI to create or modify bundles of nets
as follows:

1. Select View > Net Assistant in the GUI.
2. Select the nets to add to the bundle.
3. Click Create Bundle.
4. Enter the bundle name.
5. Sort the list by selecting the sort options and clicking the Sort button, or by dragging and
dropping the signals to reorder the list.
6. Click OK to create the bundle.
Figure 9-2 shows the Net Assistant and Create Bundle dialog boxes.

**Figure 9-2  Net Assistant and Create Bundle**

Note that the Net Assistant can display flylines associated with the nets in the bundle. In the Net Assistant, click the net bundle entry to analyze, then click Highlight or Highlight Flylines. The layout updates to show the flylines for the bundle.

To return a collection of bundles, use the `get_bundles` command. To remove a bundle from the design, use the `remove_bundles` command.

Bundle pin constraints restrict the placement of net bundles on a physical design block. You can use the constraints to keep bundle bits together, specify pin ordering, specify the allowed layers for bundles, and so on. To create a bundle constraint,

1. Use the `set_bundle_pin_constraints` command and specify the constraint options.

```plaintext
icc2_shell> set_bundle_pin_constraints -keep_pins_together true
```
2. (Optional) Verify the constraint settings with the `report_bundle_pin_constraints` command.

```shell
icc2_shell> report_bundle_pin_constraints
******************************
Report : report_bundle_pin_constraints
Design : leon3mp
******************************

Constraints on all cells and all bundles
----------------------------------------
Keep pins together: true
```

The `set_bundle_pin_constraints` command supports several options to constrain bundles. To create the constraint on specific bundles, use the `-bundles bundles` option. To apply the constraints to specific cells, use the `-cells cells` option. To apply the constraints only at the top level, use the `-self` option. To specify whether bundle pins are kept together, use the `-keep_pins_together true | false` option. To specify a region on the block where the bundle pins should be placed, use the `-sides side and -range {start end}` options. To control the ordering of the pin placement for signals in the bundle, use the `-bundle_order off | ordered | increasing | decreasing | equal-distance` option. To allow or disallow feedthroughs for a bundle, use the `-allow_feedthroughs true | false` option. To control which layers can be used for the bundle pins, use the `-allowed_layers metal_layers` option. To specify the number of wire tracks between adjacent pins, use the `-pin_spacing number_of_tracks` option. To specify the minimum distance between adjacent pins in microns, use the `-pin_spacing_distance spacing_distance` option.

---

**Bundle Constraint Example**

The following example uses bundle constraints to control the global routing of a bus. A 256-bit bus is split into two individual 128-bit buses. Bits 0 through 127 are routed from block A to block D through block B, and bits 128 through 255 are routed from block A to block D through block C. Each bundle is constrained independently and is routed separately. The commands to create this routing are as follows and the simplified layout result is shown in Figure 9-3.

```shell
icc2_shell> create_bundle -name bundle1 [get_nets \ 
   {Bus_0 Bus_1 ... Bus_127}]
icc2_shell> set_bundle_pin_constraints ... -bundle bundle1 \ 
   -allowed_layers {M3 M5} -cell Block B
icc2_shell> create_bundle -name bundle2 [get_nets \ 
   {Bus_128 Bus_129 ... Bus_255}]
icc2_shell> set_bundle_pin_constraints ... -bundle bundle2 \ 
   -allowed_layers {M3 M5} -cell Block C
```
Creating Routing Corridors

Routing corridors provide added control over the routing topology of individual nets or a bundle of nets. During global routing, IC Compiler II routes the associated nets through the routing corridors. Routing corridors are not exclusive; you can route other nets through the corridor, even if they are not assigned to the corridor. You can define a unique minimum and maximum layer for each corridor, or use the default minimum and maximum layers for the design. Note that during optimization, the tool does not legalize cells within a routing corridor.

The IC Compiler II tool supports two types of routing corridors: complete and partial. A complete corridor is a corridor which encloses all the pins of the nets assigned to that corridor. A partial corridor is a corridor that has pins (of nets assigned to the corridor) that are physically placed outside of the corridor area. Figure 9-4 shows a complete corridor and a partial corridor.
To create and verify a routing corridor,

1. **Use the `create_routing_corridor` to specify the routing corridor.**

   ```
   icc2_shell> create_routing_corridor -name corridor_a \\
   -boundary {{1925 2170} {1980 3260}} \\
   -min_layer_name M1 -max_layer_name M4 \\
   -object [get_nets irqo[0]]
   ```

2. **(Optional) Verify the routing corridor with the `report_routing_corridors` command.**

   ```
   icc2_shell> report_routing_corridors \\
   [get_routing_corridors corridor_a]
   *******************************************************
   Report : report_routing_corridors
   Design : leon3mp
   *******************************************************
   CORRIDOR NAME: corridor_a
   Shape Name       min/max       Shape
   -----------------------------------------------
   CORRIDOR_SHAPE_0 M1/M4       {1925 2170} {1980 3260}
   -----------------------------------------------
   Shapes Connected: yes
   Shapes Cover Pins and Ports: no
   Objects: irqo[0]
   ```
Alternatively, you can use the Create Routing Corridor tool in the GUI to create the corridor.

1. Select the nets to associate with the new routing corridor.

2. Select Create > Routing Corridor in the GUI.

3. Change any additional settings as needed, such as the minimum or maximum layer, the allowed drawing angles, snap settings, and so on.

4. Draw the routing corridor by clicking the mouse at the vertices of the corridor.

The IC Compiler II tool supports several commands to create, modify, and remove routing corridors. To add nets to a routing corridor, use the `add_to_routing_corridor` command. To extend the shape of the routing corridor, use the `create_routing_corridor_shape` command. To return a collection of routing corridors, use the `get_routing_corridors` command. To return a list of shapes that are contained in the routing corridor, use the `get_routing_corridor_shapes` command. To remove nets from a routing corridor, use the `remove_from_routing_corridor` command. To modify a routing corridor by removing a portion of the shape, use the `remove_routing_corridor_shapes` command.

---

**Creating Global Routes Within Routing Corridors**

After creating routing corridors, you can use the routing corridors to guide global routing. To limit global routing to only specified nets in the design, use the `route_group` command with the `-nets` and `-global_planning true` options.

To create a routing corridor and to globally route a specific net through the corridor:

1. Use the `create_routing_corridor` command to create the corridor.

   ```shell
   icc2_shell> create_routing_corridor -name corridor_a \
   -boundary {{1942 3219} {2026 2105}} \
   -object [get_nets dbgi[69]]
   ```

2. If the routing corridor is a partial corridor, set the `route.global.connect_pins_outside_routing_corridor` application option to `true` to allow the global router to connect to pins outside the corridor. This step is not required for complete corridors which cover all pins on the route.

   ```shell
   icc2_shell> set_app_options \
   -name route.global.connect_pins_outside_routing_corridor \
   -value true
   route.global.connect_pins_outside_routing_corridor true
   ```

3. Globally route the net that is associated with the corridor by using the `route_group` command with the `-global_planning` option set to `true`.

   ```shell
   icc2_shell> route_group -global_planning true -nets [get_nets \
   -of_object [get_routing_corridors corridor_a]]
   ```
Adding Repeater Cells

In the global planning flow, you can perform repeater insertion with the `add_buffer_on_route` command. Use this command after routing globally planned nets with the `route_group` command. Repeater insertion is done before pushing objects into blocks and creating pins on blocks.

```shell
icc2_shell> add_buffer_on_route \
   -allow_insertion_over_cell [get_cells i_block1 i_block2] \
   -repeater_distance 150 -lib_cell stdcell/buffer1
```

The `add_buffer_on_route` command automatically detects whether the buffer cell is an inverter or not, according to the specified library cell. The command respects placement blockages, soft macros, and hard macros by default, and allows you to add repeaters in specified soft macros or hard macros. You can use the `add_buffer_on_route` command on incomplete routes, and the command can be used during global route, track assignment and detail routing.

Options to the `add_buffer_on_route` command control how the buffers are inserted. Use the `-location`, `-user_specified_buffers`, `-repeater_distance`, or `-repeater_distance_length_ratio` options to control the positioning of buffers and the number of buffers to insert. Use the `-scaled_by_layer` and `-scaled_by_width` options to scale the distance between the buffers based on metal layer or based on the default width of the metal layer. Use the `-allow_insertion_over_cell` option to insert buffers on top of blocks. You can use the `push_down_objects` command later in the flow to push the buffers into blocks. The `add_buffer_on_route` command supports many more options; see the man page for more information.

Note that the command inserts buffers but does not legalize them. You must legalize the buffers with the `legalize_placement` command at the top and block design levels. In addition, you must use perform engineering change order (ECO) routing with the `route_eco` command to completely repair a net that has been modified by the `add_buffer_on_route` command.

Pushing Down Repeater Cells

After inserting repeater cells in the global planning flow, you can push the repeater cells into the blocks they cover with the `push_down_objects` command. The command supports concurrent cell and routing push down. After objects are pushed down, the tool sets the `shadow_status` property to `pushed_down`. Other possible `shadow_status` property values are `copied_down`, `copied_up`, `normal`, `pulled_up`, `pushed_down`, and `virtual_flat`.
To push down a net and create feedthroughs,

1. Set pin constraints to enable feedthroughs on the net.
   
   icc2_shell> set_individual_pin_constraints \
   -nets [get_nets dbgi[69]] -allow_feedthroughs true

2. Use the `push_down_objects` command to push down the net.
   
   icc2_shell> push_down_objects [get_nets \ 
   -of_objects [get_routing_corridors corridor_a]]

To push down the repeater cells and the nets connected to those cells, use the `push_down_objects` command and specify the repeater cell instances.

   icc2_shell> push_down_objects [get_cells eco_*]

In the previous example, the `push_down_objects` command pushes down all nets associated with routing corridor `corridor_a` into the blocks that are crossed by the net. Note that feedthrough creation must be enabled. Figure 9-5 shows the design block diagram before and after pushing down the route. The tool creates a pure feedthrough in Block2 and creates physical pins for detailed routed nets at the intersection of the net and the block boundaries.

Figure 9-5  Pushdown Global Route

Before pushdown

After pushdown

---

Chapter 9: Performing Global Planning
Pushing Down Repeater Cells

9-11
Performing Pin Assignment

The IC Compiler II tool provides extensive control over pin placement and feedthrough creation during floorplanning. The tool supports global pin placement constraints, individual pin constraints, and bus signal bundling for pin placement to provide flexibility and high quality-of-results. After performing an initial pin placement, you can improve the result by incrementally modifying the pin placement on specific blocks.
The flow to generate pin constraints, place pins, incrementally modify pin placement, and validate placement results is shown in Figure 10-1.

**Figure 10-1 Pin Assignment Flow**

Create global pin constraints  
(set_block_pin_constraints)

Create individual pin constraints  
(set_individual_pin_constraints)

Create bundles and bundle constraints  
(create_bundle)  
(set_bundle_pin_constraints)

Create and read pin constraint file  
(read_pin_constraints)

Place pins  
(place_pins)

Verify pin placement  
(check_pin-placement)  
(report_pin-placement)  
(report_feedthroughs)

Write pin constraints  
(write_pin_constraints)

For more details, see the following topics:

- Creating Global Pin Constraints
- Creating Individual Pin Constraints
- Reading Pin Constraints
- Topological Constraint Examples
- Performing Global Routing for Pin Placement
- Creating a Channel Congestion Map
• Placing Pins
• Checking Pin Placement
• Writing Pin Constraints
• Writing Out Feedthroughs Inserted During Design Planning
• Reporting Estimated Wire Length
• Moving Objects Between the Top and Block Level
Creating Global Pin Constraints

Global pin constraints restrict the metal layers, spacing, block sides, and the creation of feedthroughs for the blocks in your design. The tool supports feedthroughs for both normal and multiply instantiated blocks (MIB). To create global pin constraints,

1. Use the `set_block_pin_constraints` command and specify the constraint options.

   ```
   icc2_shell> set_block_pin_constraints -pin_spacing 1 \\ 
   -allowed_layers {M2 M3 M4 M5 M6 M7 M8 M9}
   ```

2. (Optional) Verify the constraint settings with the `report_block_pin_constraints` command.

   ```
   icc2_shell> report_block_pin_constraints
   ```

3. If needed, remove block pin constraints with the `remove_block_pin_constraints` command and reapply the constraint with the `set_block_pin_constraints` command.

Alternatively, use the Task Assistant to define the block pin constraint.

1. Select Task > Task Assistant in the GUI.
2. Select Pin Assignment > Block-based Constraints.
3. Click the Block-based Constraint Table.
4. Enter the constraints in the table and click Apply.

The `set_block_pin_constraints` command supports several options to control how the pin constraints are applied. To restrict the pin placement to a limited set of layers, use the `-allowed_layers {layers}` option and specify the layers on which to place pins. To create additional offset from the corner of a block to the closest pin, use the `-corner_keepout_num_tracks tracks` or `-corner_keepout_distance distance` option and specify the offset distance in wire tracks or microns. To set the spacing between adjacent pins, use the `-pin_spacing tracks` or `-pin_spacing_distance distance` option and specify the number of wire tracks or microns between pins. To specify the
allowable block sides where pins can be placed, use the \texttt{-sides} \{\textit{side_numbers}\} and \texttt{-exclude_sides} \{\textit{side_numbers}\} options. To enable or disable feedthroughs, use the \texttt{-allow_feedthroughs} \texttt{true} | \texttt{false} option. To restrict the constraints to a subset of blocks in the design, use the \texttt{-cells} \{\textit{cell_list}\} option. To control how to place pins that cannot be placed on a legal wire track, use the \texttt{-hard_constraints} \texttt{constraint_type} option.

---

**Creating Individual Pin Constraints**

Use individual pin constraints to control pin placement details on a pin-by-pin basis. Individual pin constraints take precedence over global pin constraints. To create individual pin constraints,

1. Use the \texttt{set_individual_pin_constraints} command and specify the constraint options.

   \begin{verbatim}
   icc2_shell> set_individual_pin_constraints -pins {u0_1/rstn} \\
                 -sides 4 -width {{M3 3.00} {M5 5.00}} -length {{M3 3.00} {M5 5.00}}
   \end{verbatim}

2. Verify the constraint settings with the \texttt{report_individual_pin_constraints} command.

   \begin{verbatim}
   icc2_shell> report_individual_pin_constraints
   ****************************
   Report : report_individual_pin_constraints
   Design : leon3mp
   ***************************
   Pin constraints report on nets
   ===============================
   ...
   Pin constraints report on pins
   =================================
   u0_1/rstn:
   \{sides \{4\}\} \{width \{\{M3 3.00\} \{M5 5.00\}\}\}
   \{length \{\{M3 3.00\} \{M5 5.00\}\}\}
   ...
   \end{verbatim}

3. If needed, remove individual pin constraints with the \texttt{remove_individual_pin_constraints} command and reapply the constraint with the \texttt{set_individual_pin_constraints} command.

Alternatively, use the Task Assistant to define the individual pin constraint.

1. Select Task > Task Assistant in the GUI.

2. Select Pin Assignment > Pin-based Constraints.
3. Click the Pin-based Constraint Table.
4. Enter the constraints in the table and click Apply.

Setting Individual Pin Constraint Options

Use options with the `set_individual_pin_constraints` command to enable the following actions:

- Restrict the pin placement to the specified set of pins, nets, ports, cells, and layers
  
  ```
  icc2_shell> set_individual_pin_constraints -pins pins -nets nets \
  -ports ports -cells cells -allowed_layers layers
  ```

- Set the number of wire tracks between adjacent pins
  
  ```
  icc2_shell> set_individual_pin_constraints -pin_spacing 5
  ```

- Set the distance in microns between adjacent pins
  
  ```
  icc2_shell> set_individual_pin_constraints \
  -pin_spacing_distance 2
  ```

- Specify the sides of the block that can be used to place the pin
  
  ```
  icc2_shell> set_individual_pin_constraints -sides {1 2 3}
  ```

- Specify the width or length of the pin as layer-width or layer-length pairs
  
  ```
  icc2_shell> set_individual_pin_constraints \ 
  -width {{M3 1} {M5 2}} -length {{M3 2} {M5 4}}
  ```

- Create a specified distance between the starting point for the block edge and the first pin on that edge
  
  ```
  icc2_shell> set_individual_pin_constraints -offset 10
  ```

- Set the location in the top-level design where the pin should be created
  
  ```
  icc2_shell> set_individual_pin_constraints -pins {u0_1/rstn} \ 
  -location {1430 1914}
  ```

Note that if you specify multiple conflicting pin constraints for the same pin with the `set_individual_pin_constraints` command, the last constraint takes precedence.

Reading Pin Constraints

The pin constraints file describes the preferred connection path, feedthrough control, pin constraint information, and pin spacing control information for the design. To read the pin constraints file, use the `read_pin_constraints` command and specify the constraints file name as follows:
icc2_shell> read_pin_constraints -file_name constraints.txt

After reading the constraints, use the `report_individual_pin_constraints` command to view the constraints.

The pin constraints file is divided into sections, and each section of the file begins with a section header and ends with a footer. Within each section, constraints are surrounded by curly brackets and each record description ends with a semicolon (;). Empty lines are ignored, and comments begin with a pound sign (#).

---

**Topological Map**

The topological map section specifies routing topologies for different nets in the design. Add constraints to this section to specify how nets should be routed through blocks during pin placement. Pin constraints specified in the topological map section take precedence over global pin constraints.

Each record begins with the `Nets` or `Bundles` keyword and contains one or more net or bundle names followed by pairs of objects. The keyword and object names are surrounded by curly brackets. Use an asterisk (*) as a wildcard character to specify nets with similar names. An asterisk is not supported as a wildcard character for bundle names. Objects can be a block cell, a top-level port, or any cell instance pin. Each `Nets` or `Bundles` record is terminated with a semicolon (;). The general syntax for the topological map section is as follows:

```plaintext
start topological map;
{Nets net_names}
{{Port port_name_1} {Cell cell_name_1}
    {Cell cell_name_1} {Cell cell_name_2}
    {Cell cell_name_2} {Pins pin_name_1}};
{Bundles bundle_names}
{{Cell cell_name_1} {Cell cell_name_2}};
end topological map;
```

For block objects specified by the `Cell` keyword, you can include side, offset, and layer constraints for the block pins by using the `side`, `offset`, and `layer` keywords. The side constraint specifies the side number for the block where the pins are inserted. The side number is a positive integer that starts from 1. Given any block with a rectangular or rectilinear shape, the lowest leftmost edge is side number 1. The sides are numbered consecutively as you proceeded clockwise around the shape.

The offset specifies the distance in microns from the starting point of a given edge to the routing cross-point on that edge in clockwise direction. The starting point for edge number 1 is the lower vertex, the starting point for edge number 2 is the left vertex, and so on. You must specify the side number when you specify an offset. The layer constraint specifies the metal layers to use for the pins.
The general syntax for a block object constraint is as follows:

```plaintext
{{Cell cell_name_1} {sides side_number} {offset offset_1} {layers {layer_1 layer_2 ...}}}  
```

The following example connects Net_A to blocks Cell_A, Cell_C, and Cell_E, and forces the routing between Cell_A and Cell_C to pass through Cell_B.

```plaintext
start topological map;
{Nets Net_A}
  {{Cell Cell_A} {Cell Cell_B}}
  {{Cell Cell_B} {Cell Cell_C}}
  {{Cell Cell_A} {Cell Cell_E}};
end topological map;
```

The following example connects top-level port A and pin C/a, and forces the route to pass through Cell_B if Net_A does not connect to Cell_B.

```plaintext
start topological map;
{Nets Net_A}
  {{Port A} {Cell Cell_B}}
  {{Cell Cell_B} {Pins C/a}};
end topological map;
```

The following example uses curly brackets and a wildcard for net grouping. The constraint creates a similar routing pattern from Cell_A to Cell_B to Cell_C for nets Net_A, Net_B and all nets with net names that begin with DataIn.

```plaintext
start topological map;
{Nets Net_A Net_B DataIn*}
  {{Cell Cell_A} {Cell Cell_B}}
  {{Cell Cell_B} {Cell Cell_C}};
end topological map;
```

The following example routes the Bundle1 and Bundle2 net bundles through the Cell_A and Cell_B blocks.

```plaintext
start topological map;
{Bundles Bundle1 Bundle2}
  {{Cell Cell_A} {Cell Cell_B}};
end topological map;
```

The following example sets side, offset, and layer topological constraints for pins that connect to Net_A. The routing connects Cell_A through side 1 at offset 20.18 on layer M2 to Cell_B through side 3 at offset 40.28 on layer M4. The routing also connects Cell_B through side 1 at offset 30.28 on layer M4 to Cell_C through side 3 at offset 15.88 on layer M2.
Feedthrough Control

The feedthrough control section specifies the blocks on which to allow or prevent feedthroughs. Pin constraints specified in the feedthrough control section take precedence over global pin constraints. Each record contains one or more net or bundle names within a pair of curly brackets, followed by a Feedthrough or NoFeedthrough keyword, followed by block cell names within curly brackets.

Note that you can route signals over blocks without creating feedthroughs. Disable feedthroughs for the block and net by specifying the NoFeedthrough keyword in the feedthrough control section, and create a routing corridor with the create_routing_corridor command. When you create pins with the place_pins command, the signal is routed over the block and feedthroughs are not created for the specified signal.

The general syntax is as follows. Note that this is only a syntax example; you typically do not specify both the Feedthrough and NoFeedthrough keywords for the same net.

```
start feedthrough control;
{Nets net_name}
# Specify either Feedthrough or NoFeedthrough
# but not both for a given net
  {Feedthrough {cell_name_1 cell_name_2 cell_name_3}
   {NoFeedthrough {cell_name_4 cell_name_5 cell_name_6};
  Bundles bundle_name}
    {Feedthrough {cell_name_7 cell_name_8 cell_name_9}
     {NoFeedthrough {cell_name_10 cell_name_11 cell_name_12};
  end topological map;
```

The following rules apply to feedthrough control:

- For a given net, constraint statements that appear later in the file override earlier statements.
- The Feedthrough and NoFeedthrough keywords specify whether feedthroughs are allowed or disallowed for either specified block cells or for the entire design.
If block cells are specified, all other block cells receive the opposite feedthrough constraint value. Therefore, there is no need for more than one \texttt{Feedthrough} or \texttt{NoFeedthrough} constraint line for a given net.

- Topological map constraints can take priority over feedthrough control statements. For example, A->B and B->C forces a feedthrough on B, regardless of whether feedthroughs are allowed on B.

A single \texttt{Feedthrough} or \texttt{NoFeedthrough} statement completely specifies the allowed feedthrough modules for the net. Feedthrough statements are not cumulative. If multiple feedthrough statements exist for the same net, the last feedthrough statement overrides the earlier statements and a warning message is issued.

\texttt{Feedthrough} and \texttt{NoFeedthrough} statements override the default feedthrough permission set by the \texttt{set block pin constraints -allow_feedthroughs true | false} command for the specified nets.

The statement \{ \texttt{Feedthrough A B} \} does not force the creation of feedthroughs on A and B. Instead, the statement allows feedthroughs on blocks A and B, while restricting feedthroughs on all other blocks. To force a specific feedthrough path, you must use topological constraints.

If there are feedthrough constraint conflicts between topological map constraints and previous feedthrough constraints, the feedthrough constraints from the constraint file take priority. If there are conflicts between feedthrough constraints and topological map constraints, topological map constraints take priority.

The following example enables feedthroughs for net Xecutng_Instrn[0] on blocks I_ALU, I_REG_FILE, and I_STACK_TOP.

```plaintext
start feedthrough control;
{Nets Xecutng_Instrn[0]}
  {Feedthrough {I_ALU I_REG_FILE I_STACK_TOP}};
end feedthrough control;
```

The following example prevents feedthroughs for net Xecutng_Instrn[0] on blocks I_ALU, I_REG_FILE, and I_STACK_TOP.

```plaintext
start feedthrough control;
{Nets Xecutng_Instrn[0]}
  {NoFeedthrough I_ALU I_REG_FILE I_STACK_TOP};
end feedthrough control;
```

\textbf{Physical Pin Constraints}

The physical pin constraints section specifies location and layer constraints for individual nets and pins. The general syntax is as follows:
start physical pin constraints;
(Net net_name) {cell cell_name} 
(layers {layer1 layer2}) {sides {side1 side2}}
{offset offset1} {order {order_spec}}
{start start_offset} {end end_offset}
{width layer1 width1 layer2 width2}
{length layer1 length1 layer2 length2};
{Pins pin_name} {reference reference_cell}
{off_edge} {location {x y}};
end physical pin constraints;

Supported keywords are: net, pins, reference, sides, layers, offset, order, start, end, off_edge, location, width, and length. The side constraint specifies on which side number the pin should be inserted for the specified block cell. The side number is a positive integer that starts from 1. Given any rectangular or rectilinear shape, the lower leftmost edge is side number 1. The sides are numbered consecutively as you proceeded clockwise around the shape. The shape refers to the reference boundary shape for the block.

The offset constraint specifies the distance in microns from the starting point of a specific edge to the routing cross-point on that edge in the clockwise direction. The starting point for side 1 is the lowest vertex, the starting point for side 2 is the left vertex, and so on. Since the offset is specific for each edge, you must specify the side information with the offset.

Each record in the physical pin constraints section is contained within curly brackets. The first element in the record can be a top-level net or a port for a reference block. If the record is a port object, the next record should be the reference block name associated with the port.

In the following example, ports C and D on reference block REFA are constrained. This provides flexibility to use the constraint both inside the block design and at the top level.

start physical pin constraints;
(Net E) {cell A} {layers M2} {sides 2};
(Net E) {cell B} {layers M4} {sides 4};
{Pins C} {reference REFA} {layers M2} {sides 2};
{Pins D} {reference REFA} {layers M2} {sides 2}
 {width 0.8} {length 0.8};
{Pins F} {reference REFA} {off_edge} {location {10 10}};
end physical pin constraints;

You can set order constraints in the physical pin constraint section for a collection of pins. The following example specifies explicit pin ordering for pins A and B. Pin A must be 2 wiretracks away from pin B and must use layer M4. Pins A, B, C and D must be placed within the region between 20 to 40 microns from the starting point of side 3 for block RISC_CORE. Because pins C and D are not specified with the {order} constraint, pins C and D can be anywhere within the 20 to 40 micron region on side 3.

start physical pin constraints;
{reference RISC_CORE}
{sides 3} {start 20}
 {end 40} {{order {A {spacing 2} {layers M4} B}} {pins {C D}}};
end physical pin constraints;
If the top-level block cells are instances of multiply instantiated blocks (MIBs), and there is 
more than one record for each port on the MIB’s reference block, the last constraint record 
takes priority over previous constraint records.

---

**Pin Spacing Control**

The pin spacing control section specifies pin spacing constraints on individual edges for 
block cells. Each record specifies the pin spacing constraints for one or more edges on one 
or more layers for a particular block cell. The constraint contains the `reference`, `side`, and 
`layer` keywords. The general syntax is as follows:

```
start pin spacing control;
{reference ref_cell_name} {sides {side_a side_b}}
{layer_a spacing_a}{layer_b spacing_b};
end pin spacing control;
```

The `reference` keyword specifies the block reference that this constraint applies to. For a 
block cell, this constraint refers to the reference cell name, not the cell instance name, if it 
constrains the top level.

The `sides` keyword specifies one or more sides of the block. If there is more than one side, 
specify the side number within curly brackets separated by white space. Use an asterisk (*) 
as a wildcard character to specify that all sides are allowed.

The `layer` and `spacing` constraints are specified as pairs. You can specify multiple 
layer-spacing pairs. The layer value is the metal layer name and the spacing value is the 
minimum number of wire tracks between adjacent pins for the specified layer. Only the 
specified layers are used for pin placement; unspecified layers are not used by the pin 
placer. In addition, only layers that are allowed in the corresponding reference blocks are 
allowed for pin placement.

Keywords are case-insensitive. The reference name and the layer name are case-sensitive. 
If a line contains a syntax error, the line is ignored. The command issues warnings or errors 
for any lines that are ignored.

If the pin spacing control section contains conflicting constraints for the same edge, same 
layer and the same block, the last constraint takes priority and the command issues an error 
or warning message.

In the following example, the CPU block cell is constrained to a minimum pin spacing of 2 
wire tracks on layer M2, a minimum pin spacing of 3 wire tracks on layer M4, and a minimum 
pin spacing of 3 wire tracks on layer M6 for sides 2 and 4 of the block. If the pins are placed 
on sides 2 or 4, the pins must be placed on layer M2, M4, or M6. Note than when 
reference, sides, and layer-spacing pairs are given, unspecified layers are strictly not 
used by the tool for pin placement on the specified block sides. If the pins are placed on
sides other than 2 or 4, the pins must honor the block pin constraints applied to the CPU block. Note that individual or bundle pin constraints can override these constraints.

```
start pin spacing control;
{reference CPU} {sides {2 4}} {M2 2} {M4 3} {M6 3};
end pin spacing control;
```

### Block Pin Constraints

The block pin constraints section specifies pin constraints that apply to all the pins of the specified block cell. You can specify block-related pin constraints for feedthroughs, layers, spacing, corner keepouts, hard pin constraints, sides, and exclude sides. The same constraint can be set with the `set_block_pin_constraints` Tcl command. Supported keywords are: `reference`, `sides`, `exclude_sides`, `layers`, `feedthrough`, `spacing`, `spacing_distance`, `corner_keepout_distance`, `corner_keepout_num_tracks`, and `hard_constraints`.

The general syntax is as follows:

```
start block pin constraints;
{reference reference_name}
{layers {layer_name_1 layer_name_2 ...}}
{feedthrough true | false}
{corner_keepout_num_tracks number_of_tracks}
{spacing number_of_tracks}
{hard_constraints {spacing location layer}}
{exclude_sides {side_1 side_2}};
end block pin constraints;
```

The following example sets pin constraints on references DATA_PATH and ALU:

```
start block pin constraints;
{reference DATA_PATH}
{layers {METAL3 METAL4 METAL5}}
{feedthrough true}
{corner_keepout_num_tracks 1}
{spacing 5}
{exclude_sides {2 3}};

{reference ALU}
{layers {METAL3 METAL4 METAL5}}
{feedthrough true}
{hard_constraints {spacing location layer}}
{sides {2 3}};
end block pin constraints;
```
Topological Constraint Examples

The following sections show brief usage examples for topological constraints. The constraints are implemented with the \texttt{place_pins} command.

Creating a Direct Connection Between Blocks with Topological Constraints

The following example contains three blocks: A, B, and C. The specification requires that net Net1 connect blocks A and C without creating a feedthrough in block B. To implement this design, create the following \texttt{constraints.txt} pin constraints file and specify the \texttt{read_pin_constraints -file_name constraints.txt} and \texttt{set_block_pin_constraints -allow_feedthroughs true} commands, before running the \texttt{place_pins} command.

\begin{verbatim}
start topological map;
{Nets Net1}
  {Cell A} {Cell C};
end topological map;
\end{verbatim}

The layout result is shown in \textbf{Figure 10-2}.

\textit{Figure 10-2}  \textit{Direct Connection Between Blocks}

Note that you can create the same topology for a group of nets by specifying net names, separated by white space, within the curly brackets. You can also use the asterisk character (\texttt{*}) as a wildcard.
Specifying a Partial Path

The following example contains six blocks: A, B, C, X, Y, and Z. The specification requires that port Net1In connect to block A, and block A connect to block B. No feedthroughs are allowed on blocks X and Y. To implement this feedthrough routing, create the following constraints.txt pin constraints file.

```
start topological map;
{Nets Net1}
{Port Net1In} {Cell A}
{Cell A} {Cell B};
end topological map;
start feedthrough control;
{Nets Net1}
{NoFeedthrough X Y};
end feedthrough control;
```

The layout result is shown in Figure 10-3.

Figure 10-3 Partial Path Specification

Note that Net1 has pins only in block A and block C, and the global router routes the net by creating a feedthrough in block B. The NoFeedthrough statement in the constraints file prevents the tool from creating a feedthrough in blocks X or Y, and the global router creates a feedthrough in block C instead.

Performing Global Routing for Pin Placement

If your design contains multiply instantiated blocks or existing feedthroughs, you can perform global routing on the top level and on child blocks in your design before placing block pins to aid pin placement. In the top-level design view, the blocks can be either design or abstract views.

To perform global routing on all the blocks in your design,

1. (Optional) Set host options for distributed processing.
icc2_shell> set_host_options -name distributed ... 

2. Use the route_global command with the -virtual_flat all_routing option.

   icc2_shell> route_global -floorplan true -host_options distributed \ 
                   -virtual_flat all_routing 

In the top-level design view, the blocks can be either design or abstract views. You can place pins and create feedthroughs for designs with top-level terminal or cell objects that are placed inside the block boundary. To reuse global routing that exist in the design, use the -use_existing_routing option with the place_pins command. Note that the place_pins -use_existing_routing command supports designs with or without multiply instantiated blocks. You should also use the -use_existing_routing option to retain global routes that you created in the global planning flow with the route_group command.

To limit global routing and reduce runtime when iterating on different pin placements for a small collection of pins, use the -nets_to_exclude_from_routing option with the place_pins command. The command does not perform global routing for the specified nets.

To check for unused feedthroughs, use the check_feedthroughs -unused_feedthroughs command. You can use the -congestion_map_only option to create congestion maps without writing the global routes, after running the route_global command.

Creating a Channel Congestion Map

Before performing pin placement, you can identify potentially congested areas between macros or block boundaries by creating a channel congestion map. The channel congestion map is based on the global route. To generate the channel congestion map, use the create_channel_congestion_map command as follows. The -channel_width_threshold option specifies the maximum channel width when generating the congestion map; channels larger than the specified width are not considered. The command first performs a global route on the design by using the route_global command.

   icc2_shell> create_channel_congestion_map -channel_width_threshold 10 

To view the congestion map, select View > Map > Channel Congestion in the GUI. The IC Compiler II tool also provides the What-if Channel Congestion panel for interactive analysis of congested channels. Figure 10-4 shows a channel congestion map and the What-if Channel Congestion panel.
After defining pin placement constraints, use the `place_pins` command to perform global routing and place pins on the blocks in the design. You can limit pin placement to specific layers, cells, nets, pins, and ports, and perform incremental pin placement. The following command places pins on all blocks within the current design.

```
icc2_shell> place_pins
```

To place pins in your design,

1. (Optional) Set application options as needed.
   
   ```class
   icc2_shell> set_app_options -name plan.pins.incremental -value true
   plan.pins.incremental true
   ```

2. (Optional) Specify the number of processor cores to use for multithreading.
   
   ```class
   icc2_shell> set_host_options -max_cores 4
   ```

3. Use the `place_pins` command to places pins on all blocks within the current design.
   
   ```class
   icc2_shell> place_pins
   ```
Alternatively, use the Task Assistant to place pins.

1. Select Task > Task Assistant in the GUI.

2. Select Pin Assignment > Place Pin/Global Route/Congestion and select the Placement tab.

3. Choose the Incremental or Place only top-level pins options if required.
   - If you choose the Incremental option, you must first remove all existing feedthroughs with the `remove_feedthroughs` command before proceeding.

4. Click Apply to place the pins.

Use options with the `place_pins` command to control how the pins are placed.

- Place pins only on specific cells, nets, pins, or ports by using the `-cells`, `-nets`, `-pins`, `-ports` option.
  ```
  icc2_shell> place_pins -cells {I_ALU}
  icc2_shell> place_pins -nets [get_nets Oprnd_B*]
  icc2_shell> place_pins -pins [get_pins I_DATA_PATH/Oprnd_A*]
  icc2_shell> place_pins -ports [get_ports]
  ```

- Place pins only on the current block by using the `-self` option.
  ```
  icc2_shell> place_pins -self
  ```

- Avoid placing pins that connect to a specific collection of nets by using the `-exclude_nets` option.
  ```
  icc2_shell> place_pins -exclude_nets [get_nets Oprnd_B*]
  ```

- Avoid global routing for a collection of nets by using the `-nets_to_exclude_from_routing` option. This option might be useful to reduce runtime when iterating on different pin placements for a small collection of pins.
  ```
  icc2_shell> place_pins \
  -nets_to_exclude_from_routing [get_nets RegPort_B*]
  ```

For multiply instantiated block (MIB) instances, pin placement considers all MIB instances during global route and pin assignment. Pin placement for MIBs considers blockages, routing congestion, and wire length. You can perform pin placement on selected MIB instances by specifying their instance names on the `place_pins` command line. Pin placement honors all user-specified blockages for all MIB instances, and ignores the congestion around unselected MIB instances.

The `place_pins` command supports the following application options to control pin placement.

- `plan.pins.allow_pins_in_narrow_macro_channels`: Specifies whether the global router is permitted to route through narrow channels between blocks, between hard macros,
and between blocks and hard macros. This option also controls whether the pin placer is allowed to place pins in narrow channel areas.

`plan.pins.exclude_clocks_from_feedthroughs`: Specifies whether feedthroughs are allowed on clock pins.

`plan.pins.exclude_inout_nets_from_feedthroughs`: Specifies whether feedthrough ports are created on nets that connect to multiple driver pins or nets without driver pins, or nets that connect to inout pins.

`plan.pins.incremental`: When true, specifies that pins should stay in their existing locations. The tool moves the pin within the distance defined by the `plan.pins.pin_range` and the layers specified by the `plan.pins.layer_range` application options, if the wire length, congestion, and pin alignment do not degrade.

Note:
If you perform incremental pin placement, you must first remove existing feedthroughs with the `remove_feedthroughs` command. During incremental placement, the tool can move pins along the block edge to create a layout with better quality of results (QoR). The tool can also change the layer used for the pin.

`plan.pins.layer_range`: Specifies the maximum allowable layer displacement from the original pin layer.

`plan.pins.new_port_name_tag`: Specifies the naming suffix for newly created feedthrough ports and feedthrough nets; this setting overrides the default of _FEEDTHRU_#.

`plan.pins.new_port_name_tag_style`: Specifies whether the port name tag defined by the `plan.pins.new_port_name_tag` application option is inserted at the beginning or end of the net or port name.

`plan.pins.new_split_nets_use_block_names`: Specifies whether the driver and receiver block names are inserted into the feedthrough net name.

`plan.pins.pin_range`: Specifies the maximum allowable pin location displacement from the original pin location when the `plan.pins.incremental` application option is set to true.

`plan.pins.reserved_channel_threshold`: Specifies the maximum channel size for routing, where a channel is the space between a block or module on which pins can be placed and any adjacent module, hard macro, or I/O pad.

`plan.pins.retain_bus_name`: Specifies how new feedthrough nets for buses are named if they are not marked as bus ports or nets in the database.

`plan.pins.synthesize_abutted_pins`: Specifies whether new pins are created on neighboring abutted blocks and are connected to the original single pins.
Checking Pin Placement

After placing pins, you should verify the placement and investigate any issues with pin alignment, pin spacing, feedthroughs, and other pin placement problems. To verify pin placement,

1. Use the `check_pin_placement` command and specify the verification options.

   ```bash
   icc2_shell> check_pin_placement -pin_spacing true -layers true
   --------------------------- Start Of Pin Layer Check ---------------------------
   No violation has been found
   --------------------------- End Of Pin Layer Check ---------------------------
   --------------------------- Start Of Missing Pin Check ---------------------------
   Port ahbi_HGRANT__0_ on cell u_l/u0_3 has no pin
   Port ahbi_HGRANT__1_ on cell u_l/u0_3 has no pin
   --------------------------- End Of Missing Pin Check ---------------------------
   --------------------------- Start Of Pin Spacing Check ---------------------------
   No violation has been found
   --------------------------- End Of Pin Spacing Check ---------------------------
   --------------------------- Start Of Pin Short Check ---------------------------
   No violation has been found
   --------------------------- End Of Pin Short Check ---------------------------
   ```
2. Use the `check_feedthroughs` command to check for unused feedthroughs, reused feedthroughs, and feedthrough constraint violations.

```bash
icc2_shell> check_feedthroughs -unused_feedthroughs \
   -reused_feedthroughs -net_constraints -topo_constraints
```

Un-used feedthrough ports:

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Instance Name</th>
<th>Pin Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>leon3s</td>
<td>U2/u0_0</td>
<td>U2/u0_0/ahbso__FEEDTHRU_1[0]</td>
</tr>
</tbody>
</table>

Un-used feedthrough summary:

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Instance Name</th>
<th># of unused</th>
<th># of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>leon3s</td>
<td>U2/u0_0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Re-used feedthrough ports:

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Instance Name</th>
<th>Pin Name</th>
</tr>
</thead>
</table>

Re-used feedthrough summary:

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Instance Name</th>
<th># of reused</th>
<th># of total</th>
</tr>
</thead>
</table>

Checking Feedthrough Constraint Violations

Error: The net ahbso[0] violates feed-through constraints on cell u0_1 with feed-through ports: u0_1/...

Error: The net ahbso[0] violates feed-through constraints on cell U2/u0_0 with feed-through ports: U2/...

violating feedthrough ports:

<table>
<thead>
<tr>
<th>Instance Name</th>
<th>Pin Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0_1</td>
<td>u0_1/ahbso__FEEDTHRU_1[0]</td>
</tr>
</tbody>
</table>

| U2/u0_0       | U2/u0_0/ahbso__FEEDTHRU_1[0] |

--- Start Of Check Feedthrough Net Constraints Summary ---

Number of Feedthrough ports checked:18
Number of Violating Feedthrough ports:2

<table>
<thead>
<tr>
<th>Inst Name</th>
<th>Number of Ports</th>
<th>Number of Violating Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0_1</td>
<td>6345</td>
<td>1</td>
</tr>
</tbody>
</table>
### Chapter 10: Performing Pin Assignment

#### Checking Pin Placement

**U2/u0_0 | 6345 | 1**

**U1/u0_2 | 6344 | 0**

**U1/u0_3 | 6348 | 0**

--- End Of Check Feedthrough Net Constraints Summary ---

End of feed-through constraint violations

Checking Topological Constraint Violations

Error: The net ahbso[0] violates topological constraints by not having a direct connection between cells u0_1 and ...

Error: The net ahbso[0] violates topological constraints by not having a direct connection between cells U1/u0_3 ...

Checked a total of 1 nets with topological constraints, found 1 nets with violations

End of topological constraint violations

3. If your design contains multiply instantiated block (MIB) instances, verify pin placement with the `check_mib_for_pin_placement` command.

   icc2_shell> `check_mib_for_pin_placement -swapped_connections -top_level_terminal_locations`

4. Use the `report_pin_placement` command to save the block name, layer, side, and offset for the current pin placement for each block.

   icc2_shell> `report_pin_placement > pin_placement.rpt`

5. Use the `report_feedthroughs` command to generate a list of feedthrough nets and their associated block pins.

   icc2_shell> `report_feedthroughs -reporting_style net_based`

Alternatively, use the Task Assistant to verify the pin placement and create the pin and feedthrough reports.

1. Select Task > Task Assistant in the GUI.
2. Select Pin Assignment > QoR Analysis and click the Check tab.
3. Select the checking options as needed to limit the pin verification.
4. Click Apply to check pin placement.
5. Select Pin Assignment > QoR Analysis and click the Report tab.
6. Select reporting options as needed to limit the report to specific nets or blocks.
7. Click Apply to generate the pin placement report.

   Note that the `check_feedthroughs` and `report_feedthroughs` commands are not available in the Task Assistant.

**Figure 10-5 Check Pin QoR Task Assistant Dialog Boxes**

### Checking Pin Alignment

Use options with the `check_pin_placement` command to control pin alignment checking as follows:

- Limit alignment checking to only two-pin nets only of the specified connection types: BLOCK_TO_BLOCK, BLOCK_TO_MACRO, BLOCK_TO_IO, or ALL. The default is ALL.
  ```shell
  icc2_shell> check_pin_placement -alignment true \ 
  -connection_type BLOCK_TO_BLOCK
  ```

- Check pin alignment within a specified tolerance, or identify aligned pins that have a physical block or hard macro between the pins; the default alignment tolerance is zero.
  ```shell
  icc2_shell> check_pin_placement -alignment true \ 
  -alignment_tolerance_distance 5
  ```

- Report nets that are aligned, but the pins are placed on different layers
  ```shell
  icc2_shell> check_pin_placement -alignment true \ 
  -layer_mismatch_only true
  ```

- Print out additional information about the nets connected to the violating pins and report detailed net information about pins with alignment violations; include the `-alignment_report_file filename` option to write the output to a file
  ```shell
  icc2_shell> check_pin_placement -alignment true \ 
  -alignment_report_file my_alignment_report.txt
  ```
icc2_shell> check_pin_placement -alignment true \ 
-exclude_unplaced_objects true

• Limit the checking to the specified connection type: ALL, BLOCK_TO_BLOCK, BLOCK_TO_MACRO, or BLOCK_TO_IO when checking alignment; by default, ALL is used

icc2_shell> check_pin_placement -alignment true \ 
-connection_type BLOCK_TO_BLOCK

• Limit the alignment check to two-pin nets that are aligned but have a physical block or hard macro placed between the two pins

icc2_shell> check_pin_placement -alignment true \ 
-blocked_only true
• Write a table of misalignment offsets and the number of pins in each offset range

```plaintext
icc2_shell> check_pin_placement -alignment true \
         -alignment_histogram true

... Total of 17 alignment violations
------Start Of Alignment Histogram----------
Total Two-Pin Nets: 65
Total Misaligned Nets: 17
Average Misalignment Distance: 24.3668um
Standard Deviation: 48.3um
Maximum: 151.048um

-----------------------------
Low Violation (um) High Violation (um) Number
-----------------------------
  0      50      14
  50     100      1
 100     150      1
 150     200      1
-----------------------------

------End Of Alignment Histogram----------
```

Saving Output from `check_pin_placement`

Use options with the `check_pin_placement` command to specify how the information from the pin check is saved:

• Specify the name of the error view for pin placement checking

```plaintext
icc2_shell> check_pin_placement -error_view pincheck.err
```

• Specify the name of a report file in which to write the console output from the `check_pin_placement` command; by default, the tool writes output only to the console

```plaintext
icc2_shell> check_pin_placement -filename pinchecks.txt
```

Enabling Various Pin Placement Checks

Use options with the `check_pin_placement` command to limit pin checking to the specified checks or change the default pin check behavior. By default, the command skips these checks unless otherwise specified.

• Check the pins only of the specified cell type: BLOCK, HARD_MACRO, EXTRACTED_TIMING_MODEL, or ALL. The default is BLOCK.

```plaintext
icc2_shell> check_pin_placement -cell_type HARD_MACRO
```

• Check the pins only of the specified type: PG_PINS, SIGNAL_PINS, or ALL. The default is SIGNAL_PINS.

```plaintext
icc2_shell> check_pin_placement -pin_type ALL
```
- Check for corner keepout violations on the edges of physical blocks
  \texttt{icc2\_shell> check\_pin\_placement -corner\_keep\_out true}

- Check if pins are placed only on allowed layers
  \texttt{icc2\_shell> check\_pin\_placement -layers true}

- Skip checking for block pins without a pin shape; by default, this check is performed
  \texttt{icc2\_shell> check\_pin\_placement -missing false}

- Restrict the check to the specified nets; by default, the tool checks all block pins
  \texttt{icc2\_shell> check\_pin\_placement -nets \{clk\}}

- Check for off-edge pins
  \texttt{icc2\_shell> check\_pin\_placement -off\_edge true}

- Check if any pins violate individual pin offset constraints
  \texttt{icc2\_shell> check\_pin\_placement -offset true}

- Check for pin-order constraint violations
  \texttt{icc2\_shell> check\_pin\_placement -order true}

- Check if the center of the pin is inside an associated pin blockage and if the pin shape is on the layer associated with the blockage
  \texttt{icc2\_shell> check\_pin\_placement -pin\_blockage true}

- Create an ordered list of pins that might have nonoptimal placement and create a longer routing path than necessary
  \texttt{icc2\_shell> check\_pin\_placement -pin\_detour true \-detour\_tolerance 1.5}

- Check if the center of the pin is within its associated pin guide and whether the pin is placed on a layer allowed by the pin guide
  \texttt{icc2\_shell> check\_pin\_placement -pin\_guide true}

- Check whether pins have adequate spacing between them
  \texttt{icc2\_shell> check\_pin\_placement -pin\_spacing true}

- Check for pins with coloring conflicts
  \texttt{icc2\_shell> check\_pin\_placement -pin\_mask\_constraint true}

- Check for pins that fail to honor their length constraints, width constraints, or double patterning pin length requirements
  \texttt{icc2\_shell> check\_pin\_placement -pin\_size true}
• Check for pins shorted with PG or signal preroutes; the -shorts check does not check for these shorts
  icc2_shell> check_pin_placement -pre_route true

• Check for pins that are outside the routing corridor; this option also check for pins that are on a different layer than specified by the routing corridor
  icc2_shell> check_pin_placement -routing_corridor true

• Check only top-level terminals and ignore block pins; by default, the command checks all block pins
  icc2_shell> check_pin_placement -self

• Disable checking for shorted or overlapping pins; by default, the shorts check is enabled
  icc2_shell> check_pin_placement -shorts false

• Check for pin side placement violations
  icc2_shell> check_pin_placement -sides true

• Check for single, unroutable pins under different conditions; valid values are connected, unconnected, and all
  icc2_shell> check_pin_placement -single_pin all

• Check for pin stacking violations
  icc2_shell> check_pin_placement -stacking true

• Report all the pins that were inserted by the tool to avoid single pins in an abutted design
  icc2_shell> check_pin_placement -synthesized_pins true

• Checks that pins are centered on a wire track
  icc2_shell> check_pin_placement -wire_track true

• Checks that pins satisfy track width constraints
  icc2_shell> check_pin_placement -wide_track true
Checking Pin Placement on Multiply Instantiated Blocks

Use the check_mib_for_pin_placement command to check multiply instantiated blocks (MIBs) for pin placement issues.

- Check for pins across MIB instances where the MIB block pin does not connect to the same number or pins or same type of pins.

  \[\text{icc2\_shell} > \text{check\_mib\_for\_pin\_placement} \ -\text{asymmetric\_connections}\]
  \[\text{---------- Start Of Logical Asymmetry Check ----------}\]
  \[\text{Error: Port pin\_b on MIB Reference Module CPU has asymmetric connections. (DPPA-205)}\]

- Check MIB instances for two-pin nets that do not connect to the same pin across all instances of the reference MIB block. Swapped connections are a subset of asymmetric connections.

  \[\text{icc2\_shell} > \text{check\_mib\_for\_pin\_placement} \ -\text{swapped\_connections}\]
  \[\text{---------- Start Of Swapped Connection Check ----------}\]
  \[\text{Error: Port pin\_a on MIB Reference Module CPU is a potentially swapped port. (DPPA-203)}\]

- Check for top-level terminals that connect to the same MIB pin, but at different locations on the MIB instances.

  \[\text{icc2\_shell} > \text{check\_mib\_for\_pin\_placement} \ -\text{top\_level\_terminal\_locations}\]
  \[\text{---------- Start Of Top Level Terminal Check ----------}\]
  \[\text{Error: Port A[0] on MIB Reference Module dlm\_test2 connects to top level terminals which are not all at the same location. (DPPA-204)}\]

- Check only pins on specific cells, or pins only connected to specific nets.

  \[\text{icc2\_shell} > \text{check\_mib\_for\_pin\_placement} \ -\text{asymmetric\_connections} \ -\text{cells} \ \{U\_MIB2\_2/U\_DLM\_TEST2\}\]
  \[\text{icc2\_shell} > \text{check\_mib\_for\_pin\_placement} \ -\text{asymmetric\_connections} \ -\text{nets} \ \{U\_MIB1\_2/A\}\]

Checking Feedthroughs

Use options with the check_feedthroughs command to specify how feedthroughs are checked and what information is reported:

- Specify a collection of blocks for feedthrough analysis

  \[\text{icc2\_shell} > \text{check\_feedthroughs} \ -\text{cells} \ \{u0\_1\}\]

- Write out a list of feedthroughs that are reused across MIB instances

  \[\text{icc2\_shell} > \text{check\_feedthroughs} \ -\text{reused\_feedthroughs}\]
• Check for feedthrough violations on each feedthrough net on each block and for net bundles
  icc2_shell> check_feedthroughs -net_constraints

• Check for redundant feedthroughs
  icc2_shell> check_feedthroughs -redundant

• Check feedthrough nets that connect top-level ports in the current block
  icc2_shell> check_feedthroughs -self

• Check topological constraints for each feedthrough net
  icc2_shell> check_feedthroughs -topo_constraints

• Check for unused feedthroughs
  icc2_shell> check_feedthroughs -unused_feedthroughs

• Include repeater and buffer cells in feedthrough tracing when timing libraries are loaded
  icc2_shell> check_feedthroughs -include_buffered

For net bundles created with the create_bundle or create_bundles_from_patterns command, or by using the GUI, the check_pin_placement command checks that the bundle meets the pin constraints. Bundle pin constraints are created with the set_bundle_pin_constraints command, and the check_pin_placement command checks spacing, layer, and offset constraints for the bundle. Note that individual pin constraints take precedence over bundle pin constraints, and bundle pin constraints take precedence over block pin constraints.

Use the report_pin_placement command to generate a list of pins on blocks in the design. The report contains the block name, layer, block side, and offset for each pin. To limit the report to a specified set of pins, nets, ports, or cells, use the -pins pins, -nets nets, -ports ports, or -cells cells option. Use the -self option to report pin placement for only the top-level ports. Use the -format {layer side offset cell} option to limit the report content to one or more of layer, side, offset or block name.

Use the report_feedthroughs command to generate a report that contains feedthrough nets and their associated block pin names for each feedthrough. To limit the report to a specified set of feedthrough nets, use the -nets option. Use the -include_buffered option to report feedthroughs which contain buffer cells. Use the -include_original_feedthroughs option to report feedthroughs inserted by the place_pins command and feedthroughs that previously existed in the original netlist. Use the -reporting_style net_based option to generate a single report containing feedthrough net names and associated block pins for each feedthrough in the design. Use the -reporting_style block_based option to generate multiple feedthrough reports, one
for each design block. The report generated with the -reporting_style block_based option contains a list of feedthrough net names. Use the -self option to limit feedthrough reporting to only the current block; by default, the command checks all planning levels.

## Writing Pin Constraints

Pin constraint and feedthrough information can be saved to file and used as input for incremental pin placement and feedthrough generation. To write the pin constraints for the current design, use the `write_pin_constraints` command with the appropriate options.

```bash
icc2_shell> write_pin_constraints -from_existing_pins \
   -topological_map {side offset layer} \
   -file_name pin_constraints.map
```

Alternatively, use the Task Assistant to write the pin constraints.

1. Select Task > Task Assistant in the GUI.
2. Select Pin Assignment > Read/Write Pin Constraints and click the Write tab.
3. Select the Side, Layer, and Offset options in the Topological map section.
4. Click Apply to write out the constraints.

Use the `write_pin_constraints` command to write out topological and physical pin constraints for the current block. To limit the output to a specified set of pins, nets, ports, cells, or bundles, use the `-pins pins`, `-nets nets`, `-ports ports`, `-cells cells`, or `-bundles bundles` option. Use the `-exclude_nets` option to exclude specified nets from the list of pins constraints written by this command. Use the `-self` option to write out pin constraints for the top-level ports. Specify the output file name with the `-file_name` option.

Use the `-topological_map {side layer offset offset_range offset_range layer_range layer_range}` option to write out only the specified fields to the topological map section of the constraint file. If you specify the offset_range argument, the command adds the offset_range value to, and subtracts the offset_range value from, the current offset position. This provides additional flexibility for the pin placer during incremental pin placement. For example, if the current offset is 20 and you specify `-topological_map (offset offset_range 5)`, the command writes the offset portion of the constraint as `{offset {15 25}}`. If you specify the layer_range argument, the command adds layers above and below the current layer to the output constraint. For example, if the current layer is M4 and you specify `-topological_map {layer layer_range 2}`, the command adds two layers above and two layers below the current pin layer and writes the layers portion of the constraint as `{layers {M2 M3 M4 M5 M6}}.

In the following example, the command writes out the topological map section of the constraints file, including net name, pin name, cell name, side, layer range, and offset range.
for each constraint. The constraints are based on the existing pin placement. The pin_constraints.map generated by this command is also shown.

```
icc2_shell> write_pin_constraints -from_existing_pins \ 
   -file_name pin_constraints.map \ 
   -topological_map {side offset offset_range 5 layer layer_range 1}
```

```shell
icc2_shell> shell head pin_constraints.map
```

```
START TOPOLOGICAL MAP;
{n832} 
{{pin u_m/U1192/Y}} {{cell u0_0} 
 {layers {M5 M6 M7}} {sides 4} 
 {offset {756.280 766.280}}});
{n1319} 
{{pin U45/Y}} {{cell u0_0} 
 {layers {M5 M6 M7}} {sides 4} 
 {offset {1013.464 1023.464}}});
...
```

Use the `-physical_pin_constraint {side layer offset offset_range offset_range layer_range layer_range} option to write out only the specified fields to the physical pin constraints section of the constraint file. The offset_range and layer_range arguments have the same meaning as in the topological map section.

In the following example, the command writes out the physical pin constraints section of the constraints file, including pin name, reference name, side, layer, and offset range for each constraint. The constraints are based on the existing pin placement. The physical_constraints.map generated by this command is also shown.

```
icc2_shell> write_pin_constraints \
   -file_name physical_constraints.map -from_existing_pins \
   -physical_pin_constraint {side offset offset_range 5 layer}
```

```shell
icc2_shell> shell cat physical_constraints.map
```

```
START PHYSICAL PIN CONSTRAINTS;
{pins clk} {reference leon3s} 
 {layers M5} {sides 3} {offset {694.720 704.720}};
{pins rstn} {reference leon3s} 
 {layers M6} {sides 4} {offset {756.280 766.280}};
{pins ahbi[0]} {reference leon3s} 
 {layers M2} {sides 4} {offset {171.536 181.536}};
{pins ahbi[1]} {reference leon3s} 
 {layers M4} {sides 4} {offset {171.384 181.384}};
...
```

---

**Writing Out Feedthroughs Inserted During Design Planning**

The `place_pins` and `push_down_objects` commands insert feedthroughs into blocks based on the constraints you provide. To write out a script that contains Tcl commands to re-create the feedthrough topology in another tool session, use the `write_shadow_eco`
command as shown in the following example. The example also shows the first few lines of
the script written by the command; you can source this script in another session of IC
Compiler to re-create the feedthroughs. Use the -command_style dc option to write out a
script that is compatible with Design Compiler.

```
icc2_shell> write_shadow_eco -command_style icc2 \
    -output feedthroughs.tcl
    1

icc2_shell> shell cat feedthroughs.tcl
#original net: ahbsi[31]
create_net U1/ahbsi__FEEDTHRU_0[31]
disconnect_net -net U1/ahbsi[31] [get_pins U1/u0_3/ahbsi[118]]
connect_net -net U1/ahbsi__FEEDTHRU_0[31] [get_pins U1/u0_3/ahbsi[118]]
create_pin -direction out U1/u0_2/ahbsi__FEEDTHRU_1[31]
connect_net -net U1/ahbsi__FEEDTHRU_0[31] [get_pins U1/u0_2/ahbsi__FEEDTHRU_1[31]]
connect_net -net U1/u0_2/ahbsi[118] [get_pins U1/u0_2/ahbsi__FEEDTHRU_1[31]]
...
```

---

### Reporting Estimated Wire Length

After placing the pins with the `place_pins` command, report the wire length for individual
nets or for the entire design with the `get_estimated_wirelength` command. The
command produces a wire length estimate by creating a virtual route for the net across all
design hierarchies. Use the `set_hierarchy_options` command to limit the design
hierarchies considered by the `get_estimated_wirelength` command. The
`get_estimated_wirelength` command should be used after placing block pins with the
`place_pins` command.

The following example reports the estimated wire length for the entire design and for only
the `din[0]` and `dout[1]` nets.

```
icc2_shell> get_estimated_wirelength
Total wirelength of net is: 4117.640
1

icc2_shell> get_estimated_wirelength -nets [get_nets {din[0] dout[0]}]
The length of net din[0] is: 138.478
The length of net dout[0] is: 137.485
1
```
Moving Objects Between the Top and Block Level

You can use the set_push_down_object_options, push_down_objects, set_pop_up_object_options, and pop_up_objects commands to move cells, routing, route guides, routing corridors, blockages, and pin guides between the top level and block level in your design. Use these commands to plan hierarchical interfaces using fully routed signals or wire stubs as pin markers and move routing objects that have no associated net.

To push objects from the top level to the block level,

1. Specify the types of objects to move, and how to copy them, with the set_push_down_object_options command.

   icc2_shell> set_push_down_object_options -object_type signal_routing \-block_action {copy create_pin_shape}

2. (Optional) Verify the option settings with the report_push_down_object_options command.

   icc2_shell> report_push_down_object_options \-object_type signal_routing

   ****************************************
   Report : report_push_down_object_options
   ****************************************
   --- push_down_objects options for signal_routing ---
   Top action: remove
   Block action(s): {copy create_pin_shape}
   Routing overlap check: false
   Ignore misalignment: false

3. (Optional) Check the nets for potential routing issues before pushing down to the block level with the check_objects_for_push_down command.

   icc2_shell> check_objects_for_push_down \[get_nets -of [get_cells CLK_LV*]]

   0 nets determined to have problems
   Overall runtime: 0.008u 0.000s 0:00.01e 100.9%

4. Push the specified objects from the top level to the block level with the push_down_objects command.

   icc2_shell> push_down_objects [get_nets -of [get_cells CLK_BUFFER*]]

   Examined 8 nets for push-down
   Pushed down routing for 8 nets
   Overall runtime: 0.008u 0.000s 0:00.01e 106.6%

Note that the push_down_objects command creates electrically equivalent (EEQ) pins on the block that a feedthrough net crosses over. To create feedthrough ports instead of EEQ pins, specify the -allow_feedthroughs true option with the set_individual_pin_constraints command. To allow feedthroughs on only certain types of nets during push down, apply a filter as shown in the following example.
icc2_shell> set_individual_pin_constraints -nets [get_nets * \ 
   -filter "net_type == signal"] -allow_feedthroughs true

To pop up objects from the block level to the top level,

1. Specify the types of objects to move, and how to copy them, with the set_pop_up_object_options command.

   icc2_shell> set_pop_up_object_options -object_type blockage \ 
   -block_action keep

2. (Optional) Verify the option settings with the report_pop_up_object_options command.

   icc2_shell> report_pop_up_object_options -object_type blockage

   ****************************************
   Report : report_pop_up_object_options
   ****************************************
   --- pop_up_objects options for blockage ---
   Block action: keep

3. Set the current design to the block design and create a collection of objects to pop up.

   icc2_shell> current_design alu
   icc2_shell> set placement_blockages [get_placement_blockages]

4. Set the current design to the top-level block.

   icc2_shell> current_design top

5. Pop up the objects from the specified blocks to the top level with the pop_up_objects command.

   icc2_shell> pop_up_objects \ 
   -blocks [get_cells u1/u2] $placement_blockages

   --- Process cell u1/u2: Popping up blockages from block block1 ---
   Block action: keep
   Number of Placement blockages processed: 6
   Overall runtime: 0.017u 0.000s 0:00.02e 103.7%

The set_push_down_object_options and set_pop_up_object_options commands support options to specify which objects are moved between the top level and the block level and how the objects are copied. Use the -object_type {pg_routing signal_routing routing_guide routing_corridor blockage pin_guide pin_blockage cells} option to specify one or more object types to copy. Use the -top_action keep | remove option to specify how the objects are treated at the top level. Use the -block_action {copy | keep | create_pin_shape} option to specify how objects are treated at the block level. The exact copy behavior for the -top_action and -block_action options depends on the object type; see the man pages for the push_down_objects and pop_up_objects commands for more information. Use the -routing_overlap_check true | false option to specify whether the tool performs an overlap check of routing objects before moving them.
The set_push_down_object_options command supports additional options for moving objects from the top level to the block level. Use the -ignore_misalignment true | false option to specify whether the tool pushes down objects if there are misaligned objects among multiple instantiated blocks. Use the -location_based_terminal_naming true | false option to specify whether the command creates a unique pin name based on the layer and location of the pin. Use the -collinear_margin distance option to treat parallel PG straps outside the block boundary but within the specified distance setting as collinear routes. These PG straps are created in the child block as copies of the original top-level strap, and the original strap is left untouched at the top level. Use the -pin_meet_fatwire_rule true to invoke fat-wire spacing rules within the block.

The check_objects_for_push_down command checks the routes or charging stations to be pushed down for potential issues. The command checks for collinear routes and vias, problems with physical routing that are incompatible with the logical definition of the net, and incomplete routing at the top level. Charging stations are checked for potential UPF issues. The check_objects_for_push_down command uses the option settings specified by the set_push_down_object_options command when checking the design.
Performing Timing Budgeting

To support a hierarchical design flow, the IC Compiler II tool provides timing budgeting to allocate timing among the blocks in the design. The budgeting process begins by partitioning the chip-level timing constraints into top-level and block-level constraints. To speed the budget creation process, the tool creates lightweight timing abstract representations for the blocks in the design. Each block is virtually optimized to negate the effects of high-fanout nets and provide more accurate timing estimations. The timing budgeter uses the optimized timing for each block to derive new budgets. In the final step, top-level and block-level timing budgets are written out in preparation for further block-level optimization. The timing budgeter is fully multimode and multcorner aware, and handles designs with multiply instantiated blocks.
The flow to perform timing budgeting is shown in Figure 11-1.

Figure 11-1  Timing Budget Flow

Create block timing abstracts
(create_abstract)

Perform virtual in-place optimization
(estimate_timing)

Apply manual budget constraints
(set_pin_budget_constraints)
(set_boundary_budget_constraints)

Update budget constraints
(compute_budget_constraints)

Write out timing budgets
(write_budgets)

To generate timing budgets, chip-level Synopsys Design Constraints (SDC) and UPF files must be partitioned into top-level and block-level files. The IC Compiler II tool uses the top-level constraints to constrain the top-level logic when abstract representations of blocks are used. See Split Constraints Flow for more information about partitioning chip-level constraint and UPF files.

You typically perform timing budgeting on a design with module pins that are already placed. Note that timing budgeting step requires separate constraints for the top-level and block-level logic. See Split Constraints Flow for more information about splitting chip-level constraints into top-level and block-level constraints.

To efficiently run the same task on several blocks in your design, you can use the run_block_script command to enable distributed processing and perform the tasks in parallel. The run_block_script command accepts a Tcl script and applies the commands in the script to the blocks you specify. For more information about running and monitoring parallel tasks, see Running Tasks in Parallel and Monitoring Distributed Tasks.

For more information, see the following topics on budgeting:

- Creating Block Timing Abstracts
- Performing Virtual In-Place Optimization
• Applying Manual Budget Constraints
• Updating Budget Information
• Writing Out Budget Information
Creating Block Timing Abstracts

To update timing budgets quickly and efficiently in the IC Compiler II tool, use timing abstract representations for the blocks in your design. Timing abstracts are smaller, more efficient block representations that are useful for generating timing budgets. The timing abstracts are created from blocks with pins already placed. There is no need to re-create cell placement within the blocks before creating the timing abstracts. To create block timing abstracts in parallel by using distributed computing, see Running Tasks in Parallel.

To create a block timing abstract,

1. Open the design representation for the block with the `open_block` command.
   
   `icc2_shell> open_block leon3mp.ndm:leon3s.design`

2. Merge any changes that were made to the abstract view into the full design view with the `merge_abstract` command.
   
   `icc2_shell> merge_abstract`

   If you placed the pins on your block by using an abstract representation, you must use the `merge_abstract` command to copy the pin information into the full design view.

3. Source the constraints for the block.
   
   `icc2_shell> source split/leon3s/top.tcl`

   Note that this step applies internal constraints for the block and is run only one time. You must apply either manually generated constraints or constraints generated by the `split_constraints` command before applying timing budgets. If the constraints were generated by using the `split_constraints` command, the top.tcl constraints file is stored in the `split/block_name` directory.

4. Insert feedthrough buffers with the `add_feedthrough_buffers` command.
   
   `icc2_shell> add_feedthrough_buffers`

5. Create the block timing abstract with the `create_abstract` command and include the `-estimate_timing` option.
   
   `icc2_shell> create_abstract -estimate_timing`

   Note:
   - If your design contains multiple levels of physical hierarchy, the `create_abstract` command converts only blocks at the lowest level of hierarchy into abstract views; intermediate levels of hierarchy are kept as design views.

6. Save the changes to the design.
   
   `icc2_shell> save_lib`
The `create_abstract -estimate_timing` command runs virtual timing optimization on the interface logic and annotates the abstract view with the result. You can omit the `-estimate_timing` option if you performed virtual timing optimization earlier in the flow.

Create separate abstracts for the black box blocks in your design by using the following steps:

1. Open the design representation for the block with the `open_block` command.
   
   ```
   open_block lib.ndm:cpu.design
   ```

2. Define the blocks that do not use black boxes and create a list of block references.
   
   ```
   set non_bb_blocks $DP_BLOCK_REFS
   foreach bb $DP_BB_BLOCK_REFS {
     set idx [lsearch -exact $non_bb_blocks $bb]
     set non_bb_blocks [lreplace $non_bb_blocks $idx $idx]
   }
   ```

3. Create a list of instances that do not use black boxes.
   
   ```
   set non_bb_insts ""
   foreach ref $non_bb_blocks {
     set non_bb_insts "$non_bb_insts [get_object_name [get_cells -hier -filter ref_name==$ref]]"
   }
   ```

4. Identify all black box instances at the lowest level of hierarchy.
   
   ```
   set non_bb_for_abs [get_attribute -objects [filter_collection [get_cells $non_bb_insts] "!has_child_physical_hierarchy"] -name ref_name]
   ```

5. Create abstracts for all non black boxes at the lowest hierarchy levels.
   
   ```
   create_abstract -estimate_timing -blocks [list $non_bb_for_abs]
   ```

6. Load constraints and create abstracts for black box blocks.
   
   ```
   load_block_constraints -blocks [list $DP_BB_BLOCK_REFS] -type SDC
   create_abstract -blocks [list $DP_BB_BLOCK_REFS]
   ```

You can use the `report_abstracts` command to report information about the abstract representations, including leaf cell count, compression percentage, and available modes and corners as shown in the following example.
Performing Virtual In-Place Optimization

Virtual in-place optimization optimizes the top-level and block interface logic in the design without changing the design netlist. The optimization begins by examining the timing between top-level I/O and block interfaces. To improve timing, the tool identifies nets with a large number of fanouts, then resizes drivers and inserts buffers as needed to improve timing. The changes are not permanent and the tool only annotates the design with the new timing information for later processing. Using this technique, the tool can create more accurate timing budgets earlier in the design process without requiring a full design optimization.
To perform virtual in-place optimization on the top-level design and interface logic,

1. Open the top-level design.
   
   \texttt{icc2\_shell}\> \texttt{open\_block orca\_ndm\_ORCA}

2. Read in the constraint mapping file.
   
   \texttt{icc2\_shell}\> \texttt{shell cat split/mapfile}
   
   BLENDER\_0 CLKNET BLENDER\_0/clocknets.tcl
   BLENDER\_0 SDC BLENDER\_0/top.tcl
   BLENDER\_1 CLKNET BLENDER\_1/clocknets.tcl
   BLENDER\_1 SDC BLENDER\_1/top.tcl
   BLENDER\_2 BTM BLENDER\_2/blender.btm
   ...
   ORCA CLKNET ORCA/clocknets.tcl
   ORCA SDC ORCA/top.tcl

   Create the constraint mapping file in a text editor or by running the \texttt{split\_constraints} command. The file specifies a list of blocks and their associated constraint files. See Split Constraints Output Files for more information about the constraint files generated by the \texttt{split\_constraints} command.

3. Source the top-level constraints created by the \texttt{split\_constraints} command.
   
   \texttt{icc2\_shell}\> \texttt{source ./split/ORCA/top.tcl}

   The top-level constraints, such as clock definitions and exceptions, are necessary to constrain the virtual in-place optimization. Note that the top-level constraints were previously saved to the \texttt{split/design\_name/top.tcl} file by the \texttt{split\_constraints} command. If you are using the full netlist flow, source the full-chip SDC file instead of the top-level SDC file.

4. Run virtual in-place optimization to optimize the top-level and interface logic.
   
   \texttt{icc2\_shell}\> \texttt{estimate\_timing}

   The \texttt{estimate\_timing} command creates a new corner, \texttt{estimated\_corner}, and optimizes the top-level interface logic by using the block-level timing abstracts. The \texttt{estimated\_corner} corner is similar to other process corners, such as best case and worst case. However, the delays in the \texttt{estimated\_corner} corner represent the delays in the virtually optimized design. To apply the optimized delays to the current corner and avoid creating the \texttt{estimated\_corner} corner, use the \texttt{-force\_user\_corner} option to update the current corner with the optimized delay information.

   To run the virtual in-place optimization task more efficiently on large designs, you can use the \texttt{-host\_options} \texttt{option\_name} option to enable distributed processing. To achieve good quality-of-results, the timing constraints must be reasonable.

5. Perform a timing check to identify any large top-to-block or block-to-top timing violations.
   
   \texttt{icc2\_shell}\> \texttt{report\_timing -modes [all\_modes] -corners estimated\_corner}
After running the `estimate_timing` and `report_timing` commands, the timing report now contains timing information for the new corner named `estimated_corner`. The paths reported for `estimated_corner` are annotated with additional information that indicate how the tool calculated the timing values:

- "E" or "e" indicate that the timing value is generated based on the `estimate_timing` command.
- "A" indicates that the timing value is generated based on the `create_abstract -estimate_timing` command.
- "a" indicates that the timing value is generated based on static timing analysis from the `create_abstract -estimate_timing` command.
- "S" indicates that the timing value originates from an SPEF or SDF file.

The “Delta Incr” column shows the difference in delay for each output on the path between the current corner and the `estimated_corner` corner. The “Analysis” column reports any changes made to the net during estimated timing optimization. The following examples are possible changes made by the `estimate_timing` command.

- Size: None no changes were made
- Size: BUF1 The buffer was resized; the new buffer is BUF1
- Buff: Short net The net is too short and buffering cannot improve timing
- Buff: 3 BUF1 added Three BUf1 cells were added to the net
The following is a short example of a timing report, with annotations for estimated_corner:

```
icc2_shell> report_timing -modes [all_modes] -corners estimated_corner
*****************************************************
Report : timing -path_type full -delay_type max
         -max_paths 1 -report_by design
Options: --
Design : ORCA
Version:
Date : Wed May 25 15:44:30 2016
*****************************************************
Startpoint: I_DATA_PATH/I_0_ (...)
Endpoint: I_ALU/Lachd_Result_reg_15_ (...) 
Mode: s1.mode
Corner: estimated_corner
Scenario: s1.mode::estimated_corner
Path Group: CLK
Path Type: max

Point Incr Path Delta Analysis
Incr
-----------------------------------------------------------------
 clock CLK (rise edge) 0.00 0.00
 clock network delay (ideal) 0.00 0.00
 I_DATA_PATH/I_0_/CP (senrq1) 0.00 0.00 r 0.00
 I_DATA_PATH/I_0_/Q (senrq1)
 I_ALU/U56/Z (bufbd4) 0.14 e 0.50 f -0.01 Size: bufbd7
 I_ALU/U34/ZN (inv0d4) 0.05 e 0.56 r -0.00 Size: invbdf
 I_ALU/U36/ZN (invbd2) 0.06 e 0.62 f -0.03 Size: invbda
```

### Applying Manual Budget Constraints

In the timing budgeting flow, you can create the block-level budgets automatically with the tool or manually by writing individual constraints. Using the automatic approach, the `compute_budget_constraints` command derives the constraints based on the gate delays in the design. In the manual approach, you enter the budget constraints and source them before creating the block budgets. You can combine these approaches and augment the automatic constraints with your own manual constraints. By using manual budget constraints, you can create block budgets that are independent of actual circuit timing. This feature provides you with full control when creating the block budgets.

Pin budget constraints are defined with the `set_pin_budget_constraints` command and specify how the timing budgeter allocates the available path delay through a pin. Pin budget constraints are created automatically for every hierarchical pin in the current design with the
compute_budget_constraints command. For example, the following command constrains the timing budgeter to assign 40% of the available path delay through pin u0_0/rstn to the inside of the block.

```shell
icc2_shell> set_pin_budget_constraints [get_pins u0_0/rstn] \\
    -internal_percent 40
```

Boundary budget constraints are defined with the `set_boundary_budget_constraints` command and specify boundary conditions for block pins in terms of driving cells and loads. You can associate boundary budget constraints with specific pins by using the `set_pin_budget_constraints` command as shown in the last line of the following example. Boundary budget constraints are also created automatically for every hierarchical pin in the current design with the `compute_budget_constraints` command.

```shell
icc2_shell> set_boundary_budget_constraints -name OutputLoad \ 
    -default -load_capacitance 0.15
icc2_shell> set_boundary_budget_constraints -name OutputLoad \ 
    -corner RCworst -load_capacitance 0.15
icc2_shell> set_boundary_budget_constraints \ 
    -name driving_cell_buf -driving_cell NBUFFX2_RVT
icc2_shell> set_pin_budget_constraints \ 
    -late_boundary driving_cell_buf -inputs u0_0/*
```

Latency budget constraints are defined with the `set_latency_budget_constraints` command and define expected or target values for top-level clock latencies. Latency budget constraints are created automatically for each nonvirtual clock in the current circuit when you use the `compute_budget_constraints -latency_targets actual` command. The following example constrains the budgeter to use a latency of 5ns for signal clk when performing budgeting calculations.

```shell
icc2_shell> set_latency_budget_constraints -corner RCworst \ 
    -early_latency 5 -late_latency 5 -clock clk
```

---

**Updating Budget Information**

After performing virtual in-place optimization on the top-level logic and block-level abstracts, you can generate updated timing budgets for the blocks in your design. Block timing budgets typically contain the following constraints:

- Data path I/O constraints created with the `set_input_delay` and `set_output_delay` commands
- Virtual clocks referenced in the data path I/O constraints
- Virtual clock latencies for source and network
• Real clock latencies for source only
• I/O electrical constraints created with set_driving_cell, set_load, and other commands

To generate updated timing budgets,

1. Specify the blocks to budget.
   
   icc2_shell> set_budget_options -add_blocks {u0_0 u0_1 u0_2 u0_3}

2. Update the timing budgets with the compute_budget_constraints command.
   
   icc2_shell> compute_budget_constraints -estimate_timing

   The -estimate_timing option sets the plan.budget.estimate_timing_mode application option to true and specifies that budgeting is performed on the estimated_corner corner. Based on this application option setting, the report_budget command reports timing information for the estimated_corner corner. The compute_budget_constraints command creates pin budget constraints for every hierarchical pin in the current circuit implementation.

   The compute_budget_constraints command supports options to fine-tune the budgeting task and update budget values only for specified pins in the design. Using these options, you can iteratively refine the timing budget for your design. To update budgets only for specified pins, use the -pins pin_list option. To update budgets only for inputs to blocks, use the -inputs option. To update budgets only for block outputs, use the -outputs option. To update budgets for pins which have no current budgets, use the -unspecified option. To update budgets for feedthrough pins, use the -feedthrough option. To update budgets for pins with a slack value less than a specified value, use the -slack slack_value option.

   Note that you should only use the -estimate_timing option when you want to use optimized delay values produced by the estimate_timing command. If your design timing is already optimized, run the compute_budget_constraints command without the -estimate_timing option.

3. Report the updated budgets with the report_budget command.
   
   icc2_shell> report_budget -html budget_report.html

   The budget report contains a comprehensive summary of timing budgets for each block specified by the set_budget_options -add_blocks command. The HTML-format report contains a list of timing path segments and summary sections for block timing, clock timing, timing to block boundary pins, and path segments. Figure 11-2 shows the first few lines of the HTML-format budget report generated by the report_budget command.
To further investigate timing budgets for individual paths, you can use the `-through` `pin_name` and `-warning_pins` options with the `report_budget` command. The `report_budget -through pin_name` command reports the budget of the worst-slate path through the specified budget pin and shows the proportion of the delay consumed by each segment.

The following example reports the worst-slate path through the `u0_0/n114__FEEDTHRU_0` pin.

```bash
icc2_shell> report_budget -through u0_0/n114__FEEDTHRU_0
```

```
****************************************
Report : report_budget -through u0_0/n114__FEEDTHRU_0
Module : leon3mp
Mode : func
****************************************
Path:  (S= 1.031 / D= 2.303( 0.70) / B= 3.333)  pci_clk.top_r->clk_r
      From top level startpoint
      Segment  (S= 0.182 / D= 0.248( 0.00) / B= 0.430)  13%
      Through input pin u0_0/n114__FEEDTHRU_0
      Segment  (S= 0.226 / D= 0.385( 0.00) / B= 0.612)  18%
      Through output pin u0_0/n114__FEEDTHRU_2
        Segment  (S= 0.045 / D= 0.015( 0.00) / B= 0.060)  2%
      ...
```

The `report_budget -warning_pins` command reports additional details about potential problems in the design or timing budget. The report lists pins with seemingly impossible constraints, pins with missing budget constraints, and pins with no timing paths. The following example uses the `-warning_pins` option to create a report and saves the report to the `warning_pins.txt` file in the local directory.
Writing Out Budget Information

After updating the timing budgets with the `compute_budget_constraints` command, use the `write_budgets` command to write out the updated timing constraints.

```
icc2_shell> write_budgets -blocks {u0_0 u0_1 u0_2 u0_3} -force
```

The `write_budgets` command writes out the constraints for the block references specified by the `set_budget_options -add_blocks` command. The `write_budgets` command creates a directory for each block reference name, writes a constraint file for each corner and mode, and writes a top.tcl file that sources the other constraint files for the block. The corner constraint files are named `corner_cornername.tcl` and the mode constraint files are named `mode_modename.tcl`.

Budgets apply only to the boundary paths of the blocks, not to the internal segments. Whenever you generate a new budget with the `write_budgets` command, you must source the top.tcl budget file in the block. You can apply budget files multiple times, whenever the budgets are changed or updated. When you source a new budget file, the tool overwrites any existing boundary constraints from any previously sourced budget file.

To write I/O constraints for all blocks, use the `-all_blocks` option of the `write_budgets` command. To write constraints for only a specified set of blocks, use the `-blocks (block_list)` option. To write the output constraint files to a specific directory, use the `-output directory_name` option. To overwrite an existing output directory, use the `-force` option.

Figure 11-3 shows the contents of the budgets directory created by the `write_budgets` command.
During block development later in the flow, apply the constraints written by the `write_budgets` command by sourcing the top.tcl file in the budgets directory.

Note that the constraints generated by the `write_budgets` command should be sourced after you source the constraints created by the `split_constraints` command. For example, to apply constraints to the leon3s block, open the block and use the following steps.

1. **Read in the constraint mapping file for the SDC and UPF files.**
   ```shell
   icc2_shell> shell cat split/mapfile
   leon3s  SDC  leon3s/top.tcl
   leon3s  UPF  leon3s/top.upf
   ...
   icc2_shell> set_constraint_mapping_file split/mapfile
   1
   ``

2. **Source the constraints for the block generated by the `split_constraints` command.**
   ```shell
   icc2_shell> source split/leon3s/top.tcl
   ``

3. **Read in the mapping file for the budgets.**
   ```shell
   icc2_shell> shell cat budgets/mapfile
   leon3s  BUDGET  leon3s/top.tcl
   leon3s_2 BUDGET  leon3s_2/top.tcl
   leon3s_3 BUDGET  leon3s_3/top.tcl
   icc2_shell> set_constraint_mapping_file budgets/mapfile
   1
   ``

4. **Source the I/O budget constraint.**
   ```shell
   icc2_shell> source budgets/leon3s/top.tcl
   ``

**Application Options for Budgeting**

The `write_budgets` command supports the following application options:

- `plan.budget.all_design_subblocks`: Retains the internal constraints of subblocks in the budget output by using the design representation instead of the abstract representation.
**plan.budget.all_full_budgets:** Controls whether the `write_budgets` command writes out only incremental boundary constraints or writes out full constraints for a budgeted block.

**plan.budget.allow_top_only_exceptions:** Allows exceptions created by the `set_false_path` and `set_multicycle_path` commands at the top level, without regenerating or modifying block-level constraints.

**plan.budget.hold_buffer_margin:** Constrains one segment of the path so that it meets the path hold constraint with the specified delay margin.

**plan.budget.pessimistic_driving_cells:** When calculating setup budgets at block inputs, adjusts delays so that the receiving block is charged only with the driving cell delay that is caused by an excess capacitive load.

**plan.budget.write_hold_budget:** Includes `set_input_delay -min` and `set_output_delay -min` constraints in the output budget to constrain hold optimization at the block boundaries.