IC Compiler
Hierarchical Design Methodology

Application Note

Version A-2007.12, March 5, 2008

Comments?

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

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, and Vera are registered trademarks of Synopsys, Inc.

Trademarks (™)

Active Parasitics, AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BOA, BRT, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, DC Expert, DC Professional, DC Ultra, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, Direct RTL, Direct Silicon Access, Discovery, Dynamic-Macromodeling, Dynamic Model Switcher, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Formal Model Checker, FoundryModel, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSIM^{plus}, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Milkyway, ModelSource, Module Compiler, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Raphael, Raphael-NES, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, Softwire, Source-Level Design, Star-RCXT, Star-SimXT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

Hierarchical Design Methodology Overview

In previous releases, the Synopsys Galaxy Design Platform supported hierarchical design methodology by using JupiterXT for design planning and IC Compiler for implementation.

Starting with version A-2007.12, IC Compiler supports the complete hierarchical design methodology with following features:

- Hierarchical design planning
- Block-level implementation
- Interface logic model (ILM) and FRAM model generation
- Top-level implementation

Benefits of the Hierarchical Design Methodology

The hierarchical design methodology can be used to divide and conquer large designs. In following scenarios, you can consider using hierarchical methodology:

1. Machine (memory) capacity or runtime issues
2. Structural and parallel design with team cooperation
3. Design intellectual property (IP) reuse and integration
4. Design engineering change orders (ECO)

However, you must also understand that a hierarchical methodology usually implements the design with lower utilization as compared to a flat design approach.

Hierarchical Design Support in Galaxy Design Platform

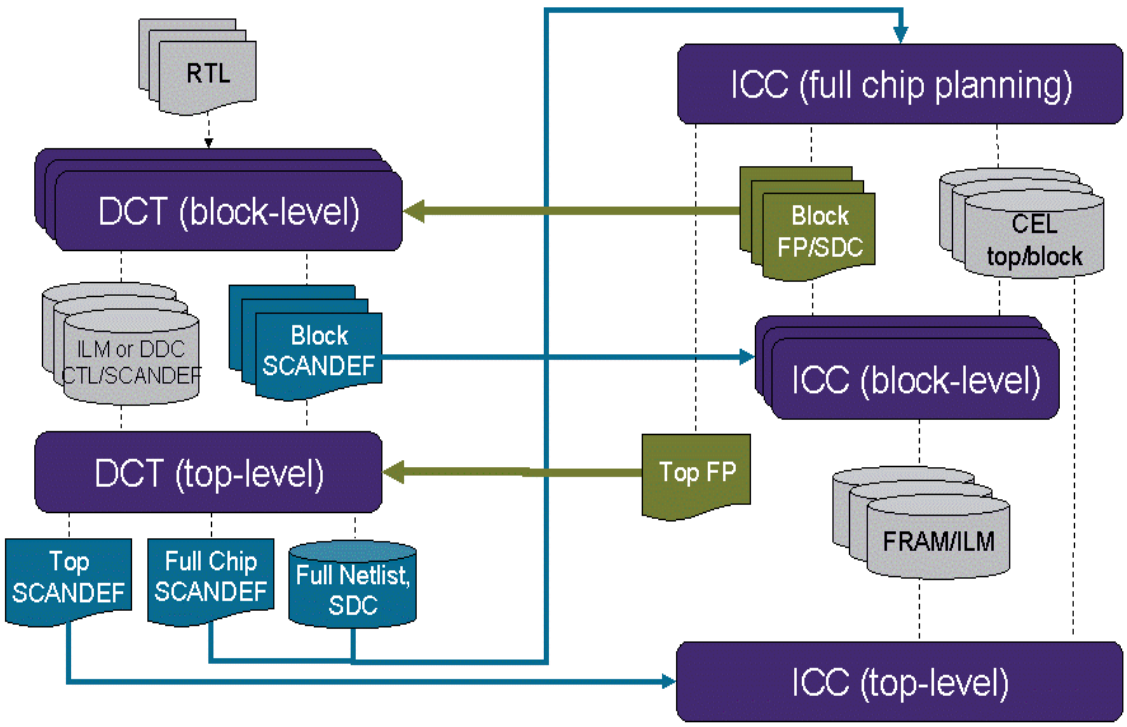
Design Compiler and IC Compiler are two of the key components of the Synopsys Galaxy Design Platform. Both tools have complete hierarchical design support.

In topographical mode, Design Compiler understands physical hierarchy while performing RTL synthesis. The virtual placement used by Design Compiler topographical mode supports top-level optimization. You can use a bottom-up flow with ILMs for hierarchical design in Design Compiler topographical mode. You can create the ILMs in either Design Compiler topographical mode or in IC Compiler. For details, see the Design Compiler topographical mode documentation.

In IC Compiler, the hierarchical methodology includes design planning, block-level implementation, and top-level implementation. You can use IC Compiler to perform both feasibility and detailed implementation flows on hierarchical designs. IC Compiler also supports hierarchical design with advanced features such as multivoltage, multicorner-multimode (MCMM), signal integrity (SI), design-for-test (DFT), abutted floorplans, and black-box handling.

Hierarchical Design Flow

Using Design Compiler topographical mode and IC Compiler, the hierarchical flow can implement a design hierarchically from RTL to GDSII. A recommended hierarchical flow is shown in the following figure:



1. Use Design Compiler topographical mode to compile the design from RTL to a gate-level netlist.

   You can use either a top-down or bottom-up approach. The netlist does not have to include scan.

2. Use IC Compiler to perform full-chip design planning using the full netlist and Synopsys design constraints (SDC) from Design Compiler topographical mode.

   Generate the physical hierarchy. For each block, output the floorplan file and SDC budgets. For the top level, output the floorplan file.

3. Use Design Compiler topographical mode to compile the design using a bottom-up flow.

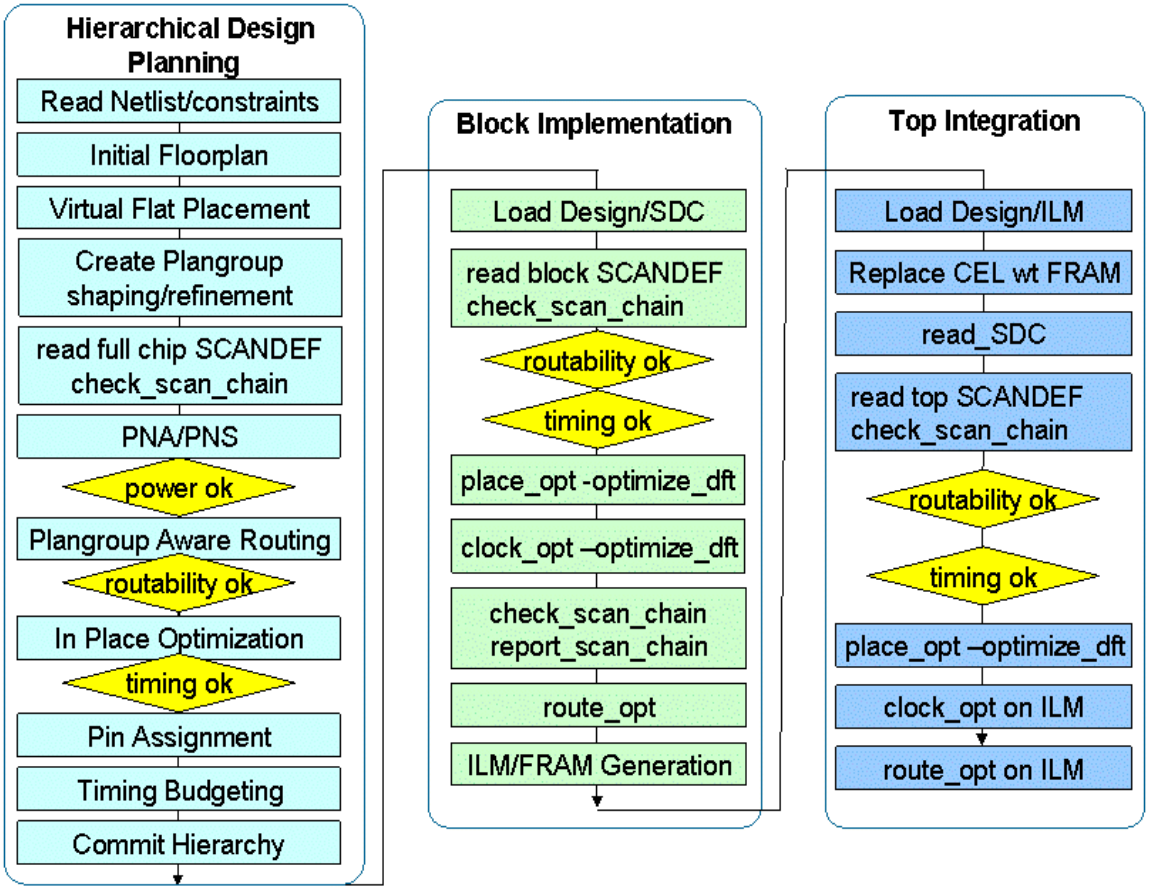
   Each block-level compile run uses the floorplan file and SDC budgets from design planning. After compiling each block, generate an ILM. Use the ILMs for the top-level compile run. These runs (both block level and top level) should include scan. Generate a SCANDEF file for each block, the top level only, and the full chip.

4. Use IC Compiler to perform design planning using the full-chip netlist and the full-chip SCANDEF file.

   Refine the hierarchical floorplan and handle the additional pins from scan insertion. For each block. implement the block using the block-level SCANDEF from Design Compiler, and then generate an ILM and FRAM model. After the blocks are complete, implement the top-level design using the ILM and FRAM models with the top-level SCANDEF file.

IC Compiler Hierarchical Design Flow

The IC Compiler hierarchical design flow (shown in the following figure) consists of three steps: hierarchical design planning, block implementation, and top-level integration. For details about these steps, see *the IC Compiler User Guide*.



IC Compiler Hierarchical Methodology

The IC Compiler hierarchical methodology consists of the following steps:

- Design partitioning
- Hierarchical floorplanning
- Timing closure
- Top-level implementation

The following sections describe these steps.

Design Partitioning

The first step of hierarchical design is to determine the physical partitioning. Consider following factors when deciding on the partitioning:

- Size

  The size of the physical partition, or block, should be appropriate for the flat IC Compiler implementation flow. A large block size can result in long turnaround time. A small block size can result in too many block jobs. Block sizes in same design should be similar so that the block jobs finish with similar runtime.

- Data paths

  The design should be partitioned using its functional units for verification and simulation purposes. You should also consider top-level connectivity and minimization of block pin counts to avoid congestion and timing issues.

- Floorplan style

  Different floorplan styles need different physical hierarchy to support them. An abutted floorplan can have no or very little top-level logic. A channeled floorplan can either have a lot of top-level logic or have only glue logic at the top level.

- Connection with the hierarchical Design Compiler topographical mode flow

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

When you create the physical hierarchy, it is best to use the existing modules in the original logical hierarchy for the physical hierarchy. If you cannot use the logical hierarchy, you can create new hierarchy modules in IC Compiler.

In the design planning phase, a physical partition is represented by a plan group. A plan group is an exclusive placement constraint to place cells of same physical partition together. It has to be defined on a module in the logical hierarchy that needs to be physically implemented and can contain only cells that belong to that module. You can manually define the location of a plan group, or you can use virtual flat placement to determine the location.

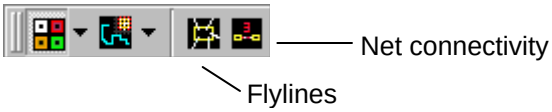
When you are ready to implement the physical partition, you commit the plan group into a soft macro, which inherits the shape and size of the plan group.

To help you decide on the physical partitions, IC Compiler provides the following features:

- Layout window
- Hierarchy browser
- Virtual flat placement
- Plan group shaping

Layout Window

In the layout window, you can use the Analysis toolbar shown below to display the flylines and net connectivity of selected cells or plan groups.



Hierarchy Browser

You can open the hierarchy browser in either the main window or the layout window. It displays the hierarchy tree of the netlist. For each logical module, it also displays columns of related information such as the reference name, the number of cells, the number of pins, the physical area, and the utilization (if the associated plan group has been created).

The hierarchy browser has a pop-up menu that you open by right-clicking in the hierarchy browser. You can use this pop-up menu to perform the following tasks:

- Add or hide columns of information
- Color the cells of specified modules or plan groups

  Cell coloring can help you identify and analyze the cell distribution of the module.

- Modify the physical hierarchy

  If you cannot use the original logical hierarchy for the physical partitions, you can use the `merge_fp_hierarchy` and `flatten_fp_hierarchy` commands to modify the logical hierarchy, which are accessible from the pop-up menu.

  The `merge_fp_hierarchy` command can merge only logical modules or cells that belong to same parent module. To merge modules that have different parent modules, you need to flatten their parent modules until the targets to merge are under same parent module.

  The hierarchy manipulation commands have options to help you automatically convert the original SDC constraints to match the new hierarchy.

Virtual Flat Placement

You can use virtual flat placement (the `create_fp_placement` command) to identify logical hierarchy modules that are feasible to use as physical partitions. When you do not know how to do the physical partitioning, run virtual flat placement without any partition constraints. The log file generated the `create_fp_placement` command contains information similar to the following:

```
…
Auto detecting hierarchy nodes for grouping ...
block I_TOP/I_BLENDER_3
block I_TOP/I_SDRAM_TOP
block I_TOP/I_BLENDER_2
block I_TOP/I_PCI_TOP
block I_TOP/I_BLENDER_1
block I_TOP/I_BLENDER_4
Above hierarchy nodes are selected for grouping
…
```

The cells of the suggested logical modules will stay close together in the placement result. This feature is called *hierarchy gravity*. The suggested physical partitioning is based on size, so that the physical hierarchy is well balanced.

If you define the physical partitions (plan groups) before virtual flat placement, hierarchy gravity does not suggest any modules, but follows the user constraints.

Plan Group Shaping

After virtual flat placement, you can place and shape plan groups based on the cell distribution (the `shape_fp_blocks` command). After placement and shaping, you can adjust the boundaries of the plan group to finalize the floorplan. IC Compiler supports both rectangular and rectilinear plan groups.

In hierarchical designs that have block boxes or soft macros, the tool can automatically adjust their shapes along with plan groups.

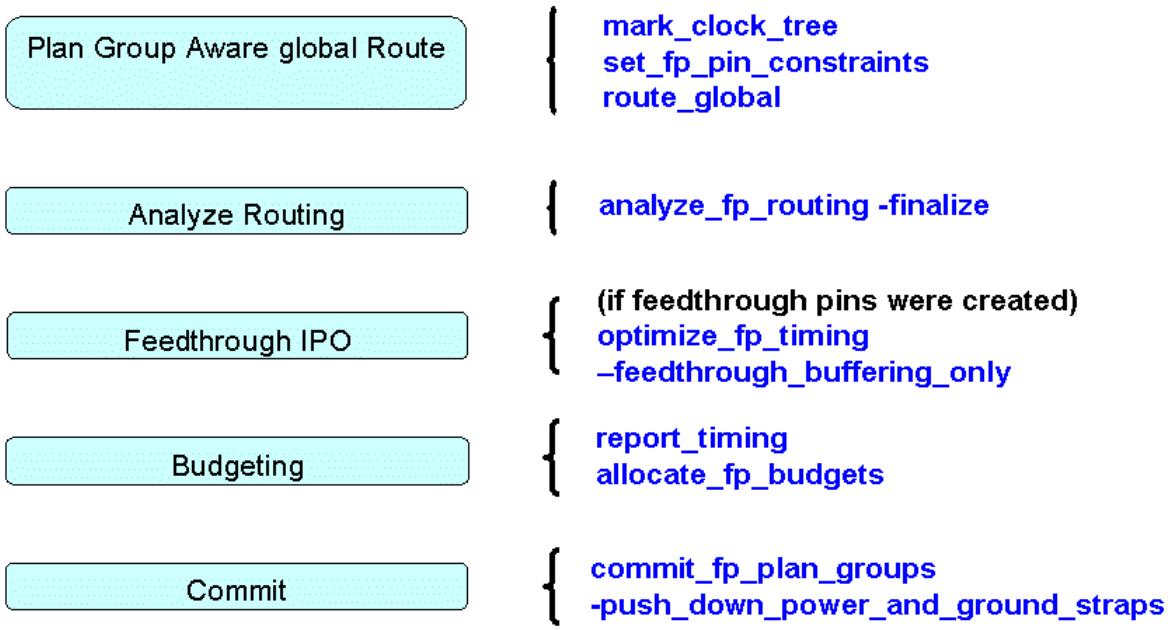
Hierarchical Floorplanning

Hierarchical floorplanning includes the following steps:

- Assigning pins
- Padding plan groups
- Shielding plan groups or soft macros
- Creating soft macros from the plan groups
- Pushing down physical objects from the chip-level into the blocks

Assigning Pins

Before you can perform pin assignment, you must finish the chip-level floorplan, placement, and netlist optimization. The following figure shows the steps and corresponding commands to do pin assignment in IC Compiler.



To perform pin assignment,

1. Use the `mark_clock_tree` command to identify the clock nets.

   By doing this, pin assignment can give high priority to clock pins and avoid feedthroughs on clock nets.

   If your design planning flow includes clock planning, it can also be used to determine the clock pin locations.

2. Define the pin assignment constraints by using the `set_fp_pin_constraints` command.

   By default, feedthrough nets are not created. To enable the creation of feedthrough nets, use the following command:

   ```
   set_fp_pin_constraints –allow_feedthroughs on
   ```

3. Use plan-group-aware routing to suggest pin locations.

   To enable plan-group-aware routing in version A-2007.12-SP1 and later versions, use the following command:

   ```
   set_fp_flow_strategy –plan_group_aware_routing true
   ```

   Note:
   The variable used to enable plan-group-aware routing in previous versions can still be used.

   When plan-group-aware routing is enabled the `route_global` command recognizes that plan groups and routes internal to a plan group will not cross plan group boundaries. Routes that cross plan groups make minimal intersections with the plan group boundaries and do not crisscross. The intersection becomes the pin location of the plan groups. You can analyze the pin assignment by checking the topology of related global routes.

   To see the feedthrough nets projected by plan-group-aware routing, use the `analyze_fp_routing –feedthrough_nets` command.

4. Use the `analyze_fp_routing –finalize` command to save the pin locations in the Milkyway design database.

   If feedthrough creation is enabled and feedthrough nets are needed, the tool creates new pins for the feedthrough nets in the logical hierarchy and saves the pin locations.

Important:
After you run the `analyze_fp_routing -finalize` command, changes to the global routes no longer affect pin assignment.

5. If feedthrough creation is enabled and feedthrough pins were added in the previous step, use the `optimize_fp_timing -feedthrough_buffering_only` command to insert buffers on the feedthrough nets.

   Inserting buffers on the feedthrough nets avoids assign statements in the Verilog output file. It also helps to improve the timing on interblock timing paths.

6. Commit the plan groups into soft macros by using the `commit_fp_plan_groups` command.
   This command also places pins physically on the soft macro boundaries. After this, you can visually see that the pins are assigned.

7. Report any pin assignment violations by using the `check_fp_pin_assignment` command.

   You can also use the `check_fp_pin_alignment` command to check pin alignment issues.

8. Refine the pin assignment, if necessary.

   You can refine the pin assignment either by manually changing the pin assignment in the GUI or by using the `place_fp_pins` command to reassign individual pins.

   In the layout window, you can move, align, and sort pin locations, as well as remove pin overlaps. To access these functions, choose File > Task > Design Planning > Pin Assignment.

   To place pins on soft macros based on top-level connectivity, use the `place_fp_pins` command.

   You can also improve pin locations based on cell placement inside the soft macro by using the `place_fp_pins -block_level` command. For more information about block-level pin assignment or improvement, see SolvNet article 021179.

Padding Plan Groups

In the hierarchical design planning phase, you should pad the plan group boundaries by using the `create_fp_plan_group_padding` command. This command prevents cells from being placed in the space around the plan group boundaries. If cells are placed too close to the plan group (later soft macro) boundaries may cause congestion and implementation issues.

The plan group padding is visible in layout window. The plan group padding is dynamic, which means the padding follows the changes of plan group boundaries. After the plan groups are committed into soft macros, their padding is automatically converted into regular placement blockages.

Shielding Plan Groups or Soft Macros

To avoid congestion and crosstalk issues between a block and the top level, you can create shielding around plan group and soft macro boundaries. The `create_fp_block_shielding` command creates rectangular metal layers inside or outside plan groups and soft macros, or both.

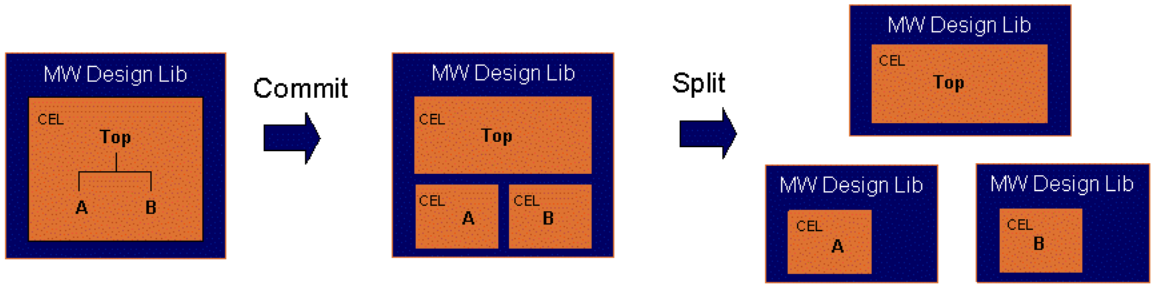
Usually, you can selectively create layers of metal to block the router from routing long nets along the block boundaries. Sometimes, even if you block layers in the preferred direction, nets can still be routed along the boundary using nonpreferred direction routing. In that case, you can consider blocking all layers. Shielding is created along the soft macro boundaries, but it leaves narrow openings to access the pins.

Creating Soft Macros

As described above, the `commit_fp_plan_groups` command commits plan groups into soft macros. Each soft macro is CEL view of the block in same Milkyway design library. All the related design information, including the netlist, floorplan, pin assignment, and power plan, is saved in the CEL view. You can use the CEL view directly for block implementation in IC Compiler.

To flatten an existing soft macro into the top level, use the `uncommit_fp_soft_macros` command.

When you commit the soft macros, the tool creates a CEL view for each of the blocks in same Milkyway design library. To allow different people to work on different blocks without interfering with each other, you can use the `split_mw_lib` command to create a new Milkyway design library for each block and the top-level design, as shown in the following figure:



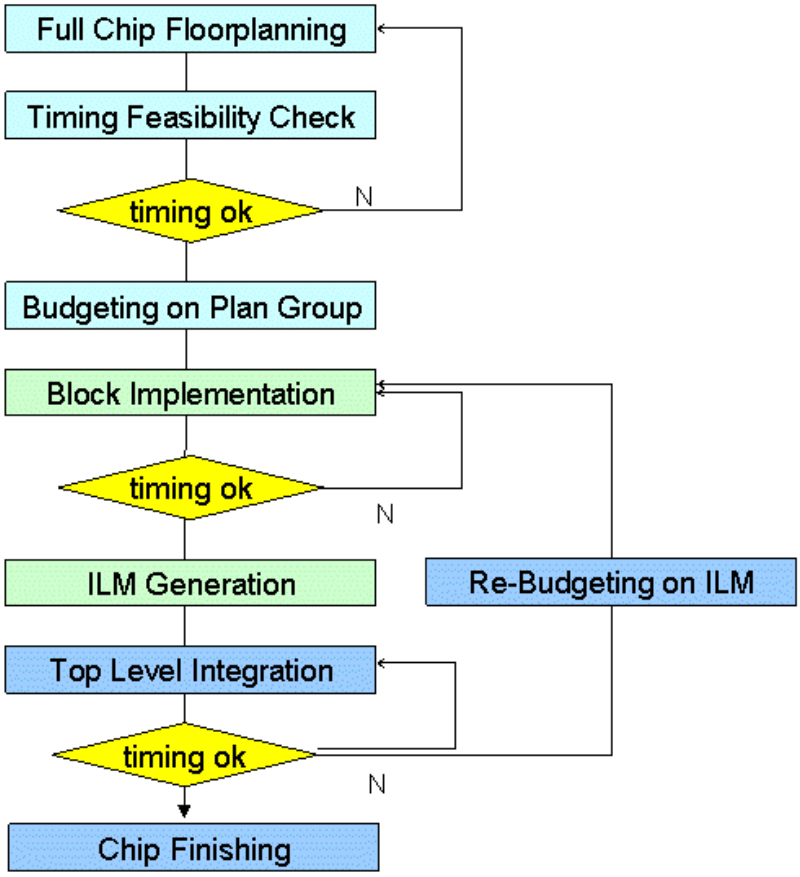
After block-level implementation, you can link the CEL views for the blocks in different design libraries to the top-level design by using the `set_mw_lib_reference` command.

Pushing Down Physical Objects

In the hierarchical design planning phase, most of the floorplanning and netlist changes are done at the full-chip level, and then are inherited by the soft macros after committing the plan groups. However, in some cases you must make design changes at the top level and then push them into existing soft macros. You can do this by using the `push_down_fp_objects` command. This command can push the following objects down into the soft macros: power plan changes, floorplan objects (such as blockages and routing guides), detail routes, and logic cells into soft macros. You can also push objects up from a soft macro to the top level by using the `push_up_fp_objects` command.

Timing Closure

Timing closure in a hierarchical design is more challenging than that in a flat design. In a hierarchical design, timing optimization might not be able to see or change all the logic of the timing paths that are causing violations. Therefore, hierarchical design needs a more sophisticated flow to achieve timing closure.



The flow shown in the previous figure is the recommended IC Compiler flow to close timing in a hierarchical design. It includes the following tasks:

- Early chip-level timing feasibility check
- Timing budgeting and block-level implementation
- Top-level integration and timing rebudgeting
- Signal integrity budgeting

Early Chip-Level Timing Feasibility Check

In the early design planning phase, you should check the timing of the entire design to identify issues. The timing feasibility check includes using the `check_timing` and `report_timing` commands to report unconstrained

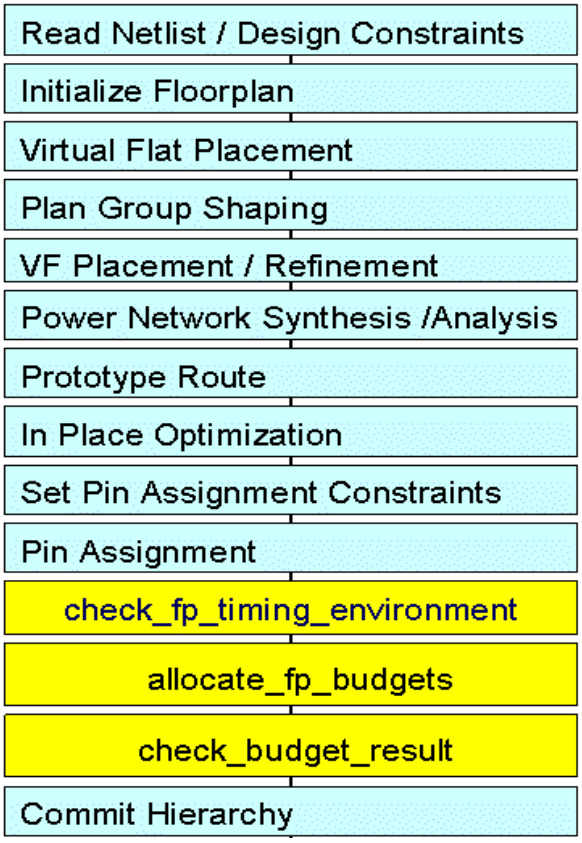
paths and timing slack, and the `check_fp_timing_environment` command to check zero wire delay timing and timing bottlenecks. It can also include other commands that you think might identify timing issues.

If the early timing check shows serious timing violations, you should use in-place optimization (the `optimize_fp_timing` command) to improve timing. When doing timing optimization, this command honors the plan groups defined in the hierarchical design, as well as the scan chain connections from the SCANDEF. The result predicts critical paths that you will face in the implementation phase. Negative slack on interblock paths after in-place optimization should not exceed 20 percent of the clock cycle time.

You should analyze the results, identify the root causes of the timing issues, such as the floorplan, SDC constraints, or netlist, and resolve them before proceeding to the next steps.

Timing Budgeting and Block-Level Implementation

After the design passes the early timing feasibility check, you can perform timing budgeting by using the `allocate_fp_budgets` command on plan groups. The following figure shows the timing budgeting steps in the design planning flow:



Timing budgeting on plan groups runs with full-chip timing. It honors pin locations and feedthrough nets assigned to the plan groups. Reasonable timing budgets result from good chip-level timing.

After committing the hierarchy by using the `commit_fp_plan_groups` command, the created soft macros do not have timing models. You might encounter warning messages such as the following:

```
Warning: Unable to resolve reference 'PCI_TOP' in TOP_DESIGN'.
(LINK-5)
Warning: Unable to resolve reference 'RISC_CORE' in 'TOP_DESIGN'.
(LINK-5)
Warning: Unable to resolve reference 'SDRAM_TOP' in 'TOP_DESIGN'.
(LINK-5)
```

These messages do not affect physical design operations, such as refining pin assignment.

If you want to perform an early timing check at the top level after committing the hierarchy,

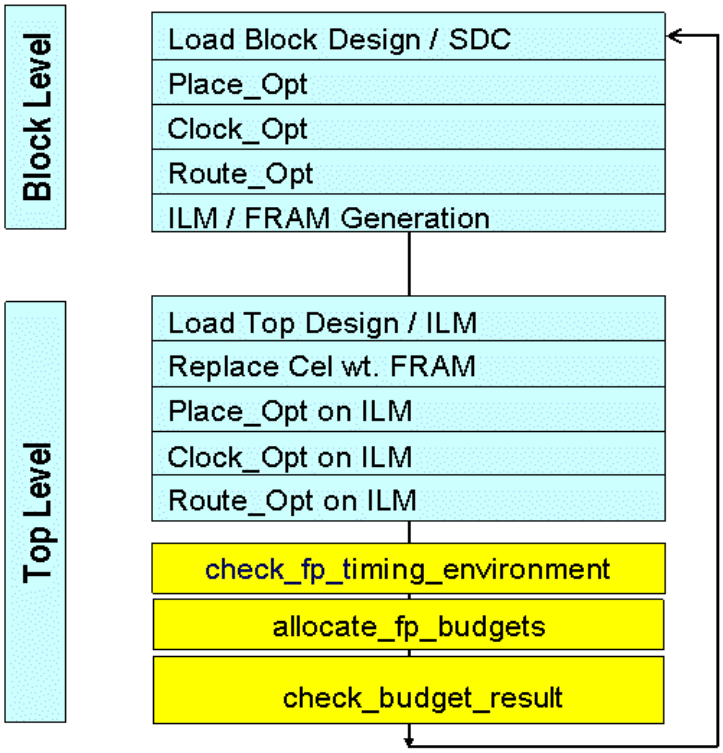
1. Save the soft macro CEL views by using the `save_mw_cel -hier` command.
2. Close the soft macro CEL views by using the `close_mw_cel` command, which leaves open only the CEL view for the top-level design.
3. Generate ILMs for the soft macros from the top level by using the `create_ilm_models` command.
4. Run a top-level timing report using the ILMs.
   This top-level timing report uses virtual-route-based timing only, as there is no global route or parasitic information for the ILMs. For accurate top-level timing, finish the global route or implementation flow inside each soft macro before generating the ILMs.

Block level implementation uses soft macros and SDC budgets. See the IC Compiler Reference Methodology document for details about the flat implementation flow. Because the timing slack and environment constraints outside the soft macro cannot be accurately predicted in timing budgeting, the block implementation should focus on violations on internal paths. Defining separate path groups for input and output paths helps with this.

You can also use the timing budgeting results from the early design planning phase to drive block-level Design Compiler topographical mode runs. In this case, you should ensure that the SDC constraints are appropriate for RTL synthesis. For more information about the RTL synthesis constraints, see the Design Compiler topographical mode documentation for more information.

Top-Level Integration and Timing Rebudgeting

Top-level integration optimizes top-level designs with block-level ILMs. If the design cannot meet top-level timing on timing paths connected to ILMs, consider rebudgeting the design using ILMs, as shown in the following figure.



Because the soft macros have been fully implemented, rebudgeting on ILM should limit changes for soft macros by using the following budgeting method:

`allocate_fp_budgets -fixed_delay_blocks {`*`list`*`} …`

The soft macros specified in the list remain unchanged (fixed delay). Timing budgeting pushes negative slack into soft macros that are not in the list. Those soft macros receive tighter constraints so that the incremental block-level run will fix the remaining violations.

Signal Integrity Budgeting

Timing budgeting can consider crosstalk effects across soft macro boundaries when generating SDC constraints. It uses noise-induced delay to allocate slack. The effective aggressor driving strength for input pins and coupling capacitance outside soft macros are stored with soft macros, so that those crosstalk effects are honored in the block-level implementation.

To enable signal integrity budgeting, set the `enable_hier_si` variable to true before running timing budgeting.

Top-Level Integration

After block-level implementation, generate ILMs for top-level integration. For details about generating ILMs, see the *IC Compiler User Guide* and Online Help.

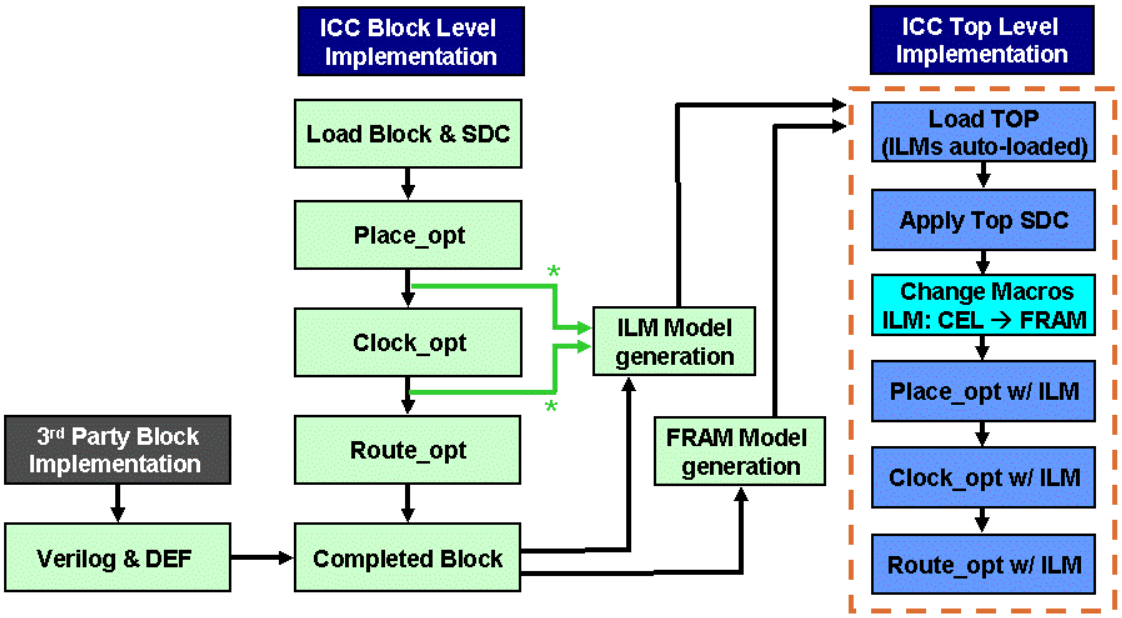
ILMs support various floorplan styles, such as

- Feedthrough nets
- Abutted floorplans
  In this case, top-level nets might have zero wire length due to abutted pins.
- Nested ILMs
  This allows you to create three or more levels of physical hierarchy in the design.

If the ILMs are in the same design library as the top-level design, the ILMs are automatically linked with the design. If the ILMs are in separate design libraries, they must be set as reference libraries of the top-level design by using the `set_mw_lib_reference` command. The name of the ILM must match the reference cell name instantiated by the top-level design.

In addition to the ILMs, you must also create a FRAM view for each block to allow over-the-block routing and to enable more accurate interconnect timing. Before starting top-level integration, use the `change_macro_view` command to change the CEL view to the FRAM view. Without the FRAM view of the soft macro, routing resources over the soft macro are completely blocked.

The following figure shows the top-level ILM flow.



* Early exploration using ILM

You can generate ILMs in IC Compiler after block-level implementation or you can generate them on blocks implemented by third-party tools. After each major step in block-level implementation, you can generate ILMs for early exploration at the top level.

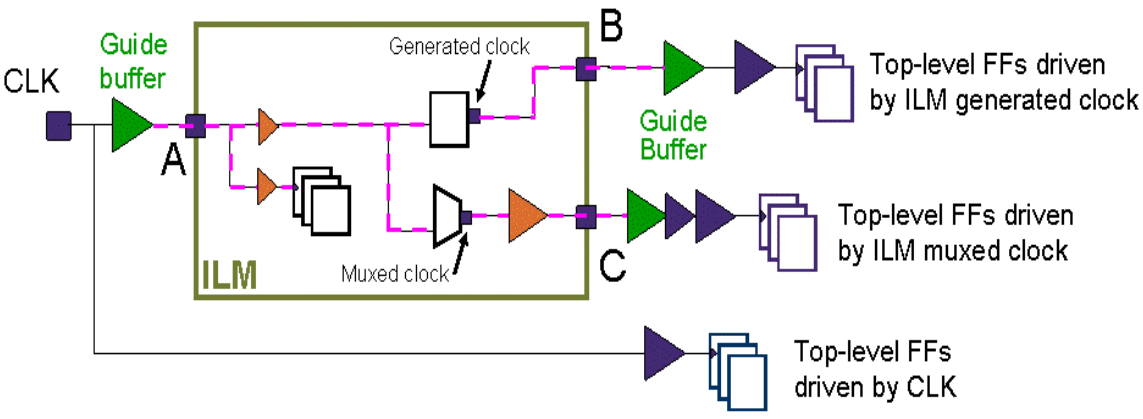
Clock Tree Extraction In ILMs

In version A-2007.12-SP2 and later versions, default ILM generation extracts the longest and shortest paths on each clock tree, in addition to the clock trees connected to the interface logic. Using the abstracted clock trees results in a smaller ILM than with full clock trees, while still providing accurate top-level clock timing.

Prior to version A-2007.12, default ILM generation extracted only those clock trees that connect to interface logic and top-level clock tree synthesis required full clock trees. To workaround this issue, you have two choices: extract the full clock tree by using the `create_ilm -keep_full_clock_tree` command, at the cost of a larger ILM; or enable top-level clock tree synthesis to use the default ILMs by setting the `cts_force_ilm_keep_full_clock_tree` variable to false, at the cost of less accurate top-level clock skew results possibly reduced clock tree QoR.

Top-Level Clock Tree Synthesis Using ILMs

Top-level clock tree synthesis merges the clock trees inside the ILM into the top-level clock trees to measure clock skew and insertion delay. However, clock nets that belong to ILM clock trees are marked with `dont_buffer_net` attributes. To isolate these nets from the top-level nets, clock tree synthesis adds guide buffers outside the ILM clock inputs and outputs, as show in the following figure.



In some cases, the guide buffers can affect the clock tree synthesis QoR. To allow clock tree optimization to optimize or remove these guide buffers, set the `cto_remove_ilm_guide_buffer` variable to true.

Clock definitions and exceptions inside the ILM are automatically propagated to the top-level for clock tree synthesis.

Advanced Hierarchical Design Feature Support

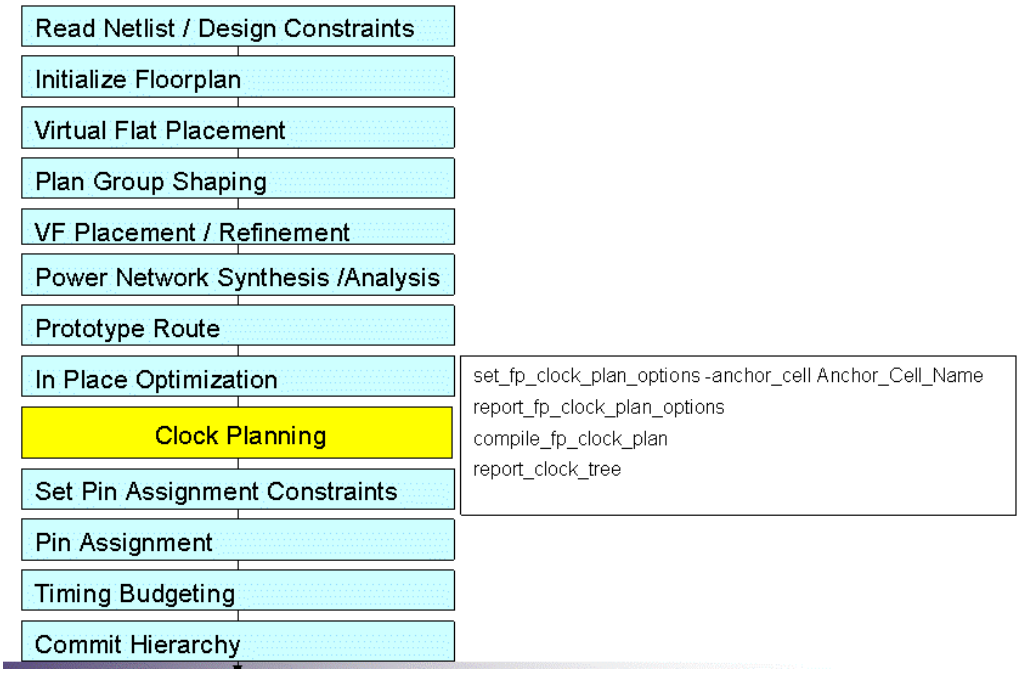
IC Compiler supports advanced design features, such as hierarchical DFT methodology with Design Compiler topographical mode, hierarchical multivoltage design flow, and hierarchical multicorner-multimode (MCMM) design flow. For details, see the Design Compiler topographical mode documentation and *IC Compiler User Guide*.

The virtual flat flow discussed in this document is the recommended hierarchical design flow. However, IC Compiler also supports other hierarchical flows, such as the top-down flow, bottom-up flow, and black box flow. Abutted hierarchical floorplans are also supported in IC Compiler. As compared to the virtual flat flow, these flows follow similar steps with minimal changes.

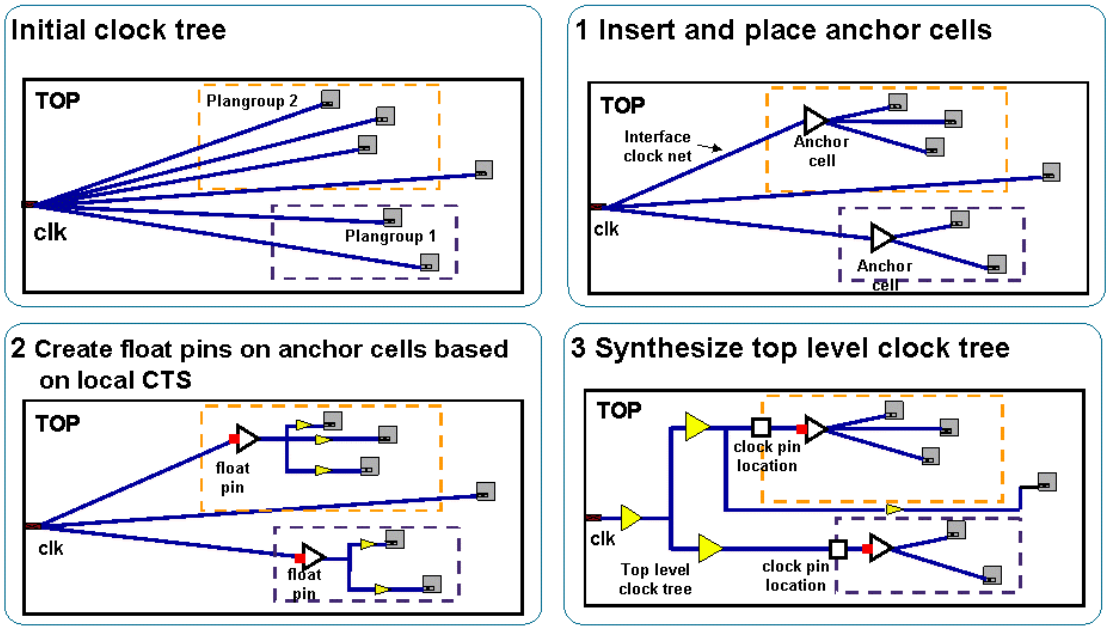
Clock Planning

Clock planning is an optional step in the design planning flow. You can use clock planning to estimate the clock tree insertion delay and skew in a hierarchical design. It also helps you determine the optimal clock pin locations on blocks. For details about clock planning, see the clock planning documentation in the *IC Compiler User Guide*.

If you use clock planning, run it before pin assignment as shown in the following figure.



Clock planning inserts anchor cells to isolate the clock trees inside the plan group from the top-level clock tree. It then runs fast clock tree synthesis for each plan group to estimate the clock skew and insertion delay. The clock tree synthesis results are annotated on floating pins defined on the anchor cells. Clock planning synthesizes the top-level clock tree to the anchor cell floating pins. The following figures show these steps.



For hierarchical timing closure, clock planning is also used to generate realistic clock latency through budgeting to fix timing violations.

Top-level timing violations result not only from delays on combinational logic between registers, but also from clock skew between launching and capturing registers. In a hierarchical design, clock tree synthesis inside each soft macro cannot consider clock skews with the top level and other soft macros. In the top-level integration phase, clock skews among different soft macros can cause set up and hold violations on interblock paths. It is very hard to fix these violations without changing the soft macros.

To estimate the chip-level clock skew issues for block-level implementation, consider using clock planning.

Timing budgeting with and without clock planning generates different SDC constraints. The following table gives a comparison of the budgeting results.

SDC Constraints	No Clock Planning	With Clock Planning
Environment	**Same**	**Same**
Clock Definition	**Same real clock** create_clock [get_ports clk] –name clk	**Same real clock** create_clock [get_ports clk] –name clk
Virtual Clock	**For Timing exception** create_clock –name clk_virtual1	**For Timing exception** Create_clock –name clk_virtual1 **For Clock Latency Budgets** create_clock –name clk__v__in create_clock –name clk__v__out
Clock Latency	**From Full Chip SDC** set_clock_latency 0.25 [get_clock clk] …	**Source and Network from CP** set_clock_latency –source … [get_clock clk] Set_clock_latency … [get_clock clk] set_clock_latency … [get_clock clk__v__in] (set_clock_latency –source … [get_clock clk__v__in]) …
Slack Budget	**No latency budgets** set_input_delay –clock clk 1.5 [get_pins A]	**Consider latency budgets** set_input_delay –clock clk__v__in 1.5 [get_pins A]
Timing Exceptions	**Same**	**Same**

In both cases, the SDC constraints have the same environment and real clock definitions. The virtual clocks used for timing exceptions are also the same. All the virtual clocks use same naming convention: [$real_clock]_virtual[1-9].

However, for timing budgeting run after clock planning virtual clocks are also generated for clock latency budgets. Those virtual clocks use the following naming convention: [$real_clock]__v__[in|out].

For clock latency constraints, timing budgeting without clock planning uses the ideal latency from the full-chip SDC constraints. Timing budgeting with clock planning uses the source and network latency from the clock planning results.

For slack budgets, budgets without clock planning will have the input and output delays defined with reference to real clocks. Clock planning budgets define input and output delays with reference to the related virtual clocks that are for latency budgets. One clock planning budget example is shown below:

```
…
### Clock Definition
create_clock –name clk [get_ports clk]
### Clock Latency
set_clock_latency –source … [get_clock clk]
set_clock_latency … [get_clock clk]
…
### Clock latency and slack budgets on Pin A
create_clock –name clk__v__in
set_clock_latency 2.6 [get_clock clk__v__in]
…
set_input_delay –clk clk__v__in 1.5 [get_port A]
…
```

The timing path through port A not only has input delay of 1.5, but also a clock latency 2.6 from clk__v__in. Block-level optimization honors these constraints to identify timing violations. Without clock latency budgets, the same path might not be identified as a critical path until you perform top-level clock tree synthesis.

For more details about clock latency budgets, see the timing budgeting documentation in the *IC Compiler User Guide*.

IC Compiler Hierarchical Reference Methodology

The hierarchical reference methodology is part of the IC Compiler reference methodology release. It provides you with a full set of scripts to go through design planning, block-level implementation, and top-level integration. You can use it to understand the major and commonly used commands and options in the hierarchical flow.

You can get the reference methodology scripts from SolvNet article 021624.