SpyGlass® Base Submethodology (for GuideWare 2017.12)

Version N-2017.12-SP2, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information
©2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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.

Trademarks
Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Third-Party Links
Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Report an Error

The SpyGlass Technical Publications team welcomes your feedback and suggestions on this publication. Please provide specific feedback and, if possible, attach a snapshot. Send your feedback to *spyglass_support@synopsys.com*.

Contents

Preface 7
- **About This Book** 7
- **Contents of This Book** 8
- **Typographical Conventions** 9

Sub-methodology for Lint Clean Design 11
- **Early Design Closure - The Need** 11
- **Introduction** 12
  - Tool and Methodology Versions 12
  - Terminology 12
- **Concept - Challenges in Development of an SoC Design** 15
  - Challenges Involved During New RTL Block/Sub-system Development 16
  - Challenges Involved in the Selection of Third Party or Internal Legacy IP 20
  - Challenges Involved During SoC Integration 21
- **Approach - How to Make Your Design Ready** 25
  - During RTL Development 25
  - During SoC Integration and Implementation 33
- **Conclusion** 44

Preface

About This Book

The SpyGlass Base Sub-methodology Guide describes a methodology for simulation, synthesis, connectivity, and basic structural readiness.

Contents of This Book

The SpyGlass Base Sub-Methodology Guide consists of the following chapter:

Chapter	Describes...
Sub-methodology for Lint Clean Design	Methodology for simulation, synthesis, connectivity, and basic structural readiness

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	OUT <= IN;
Reworked example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

Sub-methodology for Lint Clean Design

Early Design Closure - The Need

IC designs go through several transformations in a typical RTL-to-layout flow, and as they do, a number of verification steps (simulation, synthesis, equivalence checking, etc.) are performed to ensure that the design intent is preserved. Such early design analysis ensures the design can be verified and implemented right from the start, preventing any time consuming iterations.

Introduction

Early analysis of a design for simulation, synthesis, connectivity, and structural readiness requires a methodology that will guide designers through each step in the design flow. This will not only improve the QoR and facilitate handoff, but will also reduce expensive re-spins and iterations. This document introduces a methodology to make your design ready for RTL handoff, netlist handoff, or SoC integration. The SpyGlass® tool suite is an industry standard for early design closure in IC design flows. SpyGlass analyzes design intent (RTL, netlist & constraints) as soon as it is captured and enables programmed handoff of design assumptions.

This submethodology relates to “simulation, synthesis, connectivity and structural readiness (aka lint)”. The GuideWare Reference Methodology provides a jump-start for design groups with SpyGlass goals readily usable out-of-the-box at various phases of IC design flow (RTL, Netlist, and Chip Integration design phases). The GuideWare Reference Methodology can be configured to map to customer specific design style and handoff requirements. For more details of GuideWare Reference Methodology, please refer to the documentation as part of this installation.

Tool and Methodology Versions

SpyGlass: Version N-2017.12-SP2

GuideWare: 2017.12

Terminology

This section defines some commonly used terms that have a specific meaning in the SpyGlass environment.

- **Rule**: In SpyGlass environment, a ‘Rule’ represents the atomic unit of RTL analysis and checking performed by the SpyGlass software. Although a ‘Rule’ can be configured, it cannot be further sub-divided to select what analysis is performed.
- **Parameter**: In SpyGlass environment, a ‘Parameter’ is like an option to a rule that dictates the rule behavior. Parameters are typically used to make the rule do specific or detailed analysis of the RTL.

- **Goal**: A SpyGlass goal is a collection of relevant rules that are grouped together to perform a specific task. In addition to the rule list, a goal may further configure the rule parameters and redefine severity labels assigned to these rules. SpyGlass software release contains a useful set of many widely applicable goals. However, a user may fine-tune existing goals or create new goals to meet their specific design and workflow needs.
- **Sub-methodology**: A SpyGlass sub-methodology is a set of relevant goals that are grouped together to achieve a particular design goal. In addition to software, these sub-methodologies contain detailed documentation to assist customer in understanding specific usage and debug nuances. This documentation is released as part of 'SpyGlass Methodology Series', and consist of following documents additionally:
  - SpyGlass CDC Methodology
  - SpyGlass Constraints Methodology
  - SpyGlass DFT Methodology
  - SpyGlass Power Methodology
  - SpyGlass TXV Methodology

  The 'SpyGlass Methodology series' contain a rich and proven set of industry experience in design analysis, and most of SpyGlass sub-methodologies have been in production use for many years, by a wide-spectrum of customers and design teams.
- **Violation Message**: A violation message (or simply a message) is unit of SpyGlass reporting. When a SpyGlass 'rule' detects a design condition not consistent with the rule requirement, it reports each such occurrence as a (violation) message. In addition to text message, such report usually contains other supporting data, such as back-reference in RTL source code where such problem originates, schematic highlight of the problem, detailed tables and graphs (as in power activity over time), waveform for a formal 'witness' (such as a false path proven to be not a false path), and so on.
- **Severity**: A SpyGlass violation message is tagged with an attribute, called severity, which helps to identify the criticality of reported message, within the context of a goal and sub-methodology being run. SpyGlass supports four main severity classes: FATAL, ERROR, WARNING, and INFO. A SpyGlass rule or goal can define a (severity) text label belonging to one of the above classes, and attach it to a rule.

- **Waivers**: A SpyGlass 'waiver' is a method for user to review a rule (violation) message and flag a specific occurrence (or set of occurrences) as acceptable in context of their design and workflow. This is a very important mechanism to flag an apparently non-compliant design scenario as intended and verified by actual design or verification engineer. In the SoC design workflow, the SpyGlass waivers play a very significant role both in Block regressions and in Block handoff to SoC integration and implementation teams.
- **SGDC**: SGDC is an abbreviation for 'SpyGlass Design Constraints', and is used to capture additional designer intent of the block/SoC functionality which are not obvious at RTL/netlist. SGDC is used for capturing a wide variety of design intent, related with clock domain crossing, power, testability, etc.
- **BaseSpyGlass**: 'BaseSpyGlass' is a sub-methodology for basic design checks related to RTL design connectivity, simulation/ synthesis readiness, design structural issues, design profiling, and basic clock/ reset integrity checks. Within SpyGlass software release, these goals can be found within 'SPYGLASS_HOME/Methodology/BaseSpyGlass' directory.
- **Console**: Console is the enhanced user interface available in SpyGlass for configuring, selecting, and running the GuideWare methodology (as well as any other custom or standard methodology).

Concept - Challenges in Development of an SoC Design

The development of large SoC designs typically involves integration of various disparate sub-systems or blocks. Most of these blocks are sourced from legacy designs or third party IP providers. A few blocks may involve significant changes before they are used in the final SoC. In some cases, a new block is developed from scratch. All these blocks are finally stitched together to develop large SoC design(s).

The development process of large designs is divided into various stages, as shown in the following figure:

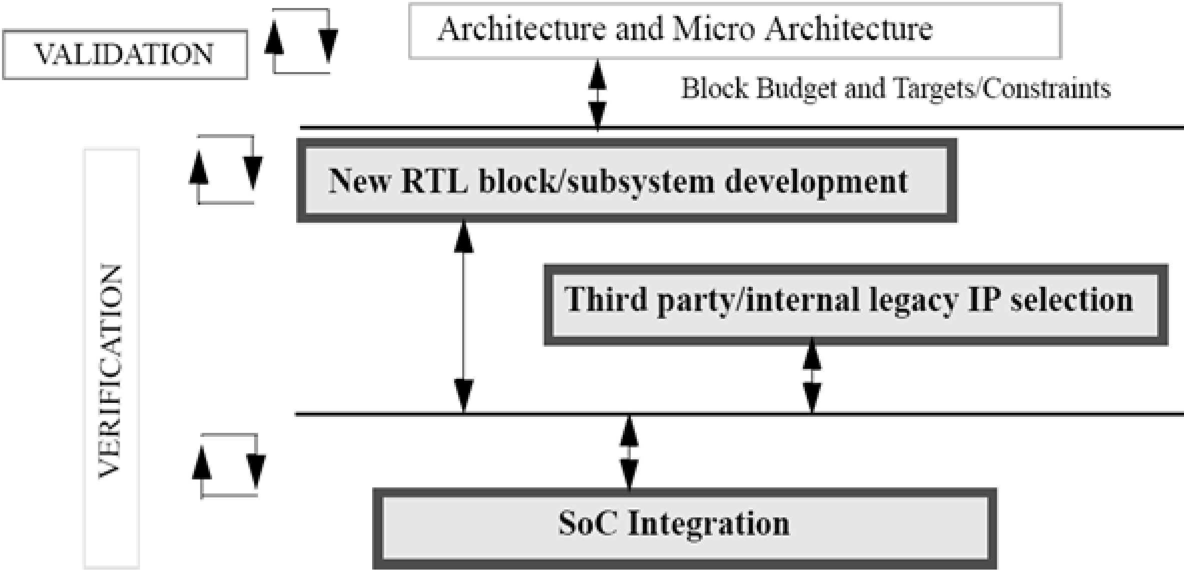


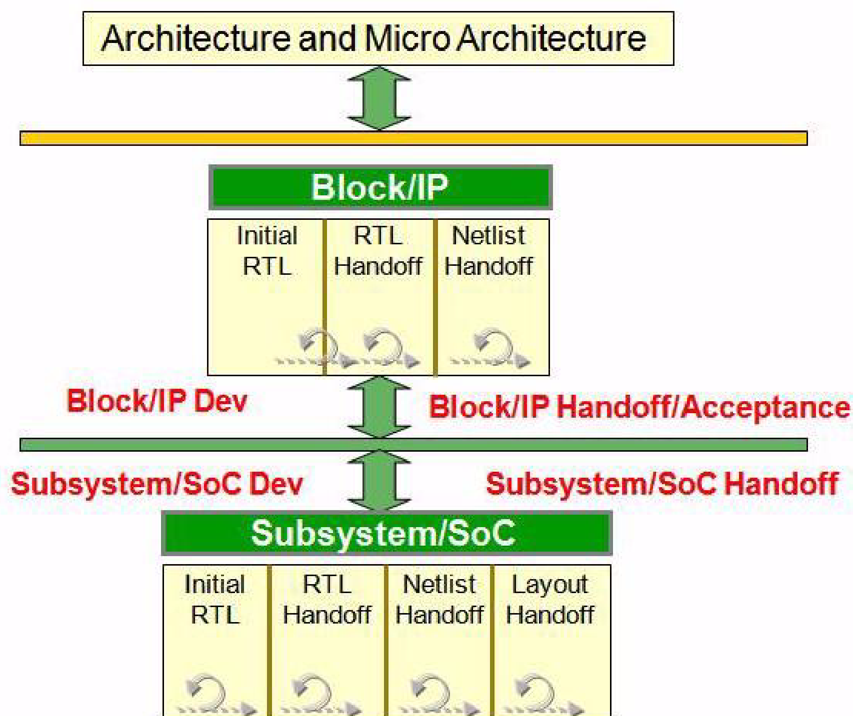
FIGURE 1. Design Development Flow

Challenges Involved During New RTL Block/Sub-system Development

In the development flow process, you can use the GuideWare Reference Methodology to achieve your design goals for the following:

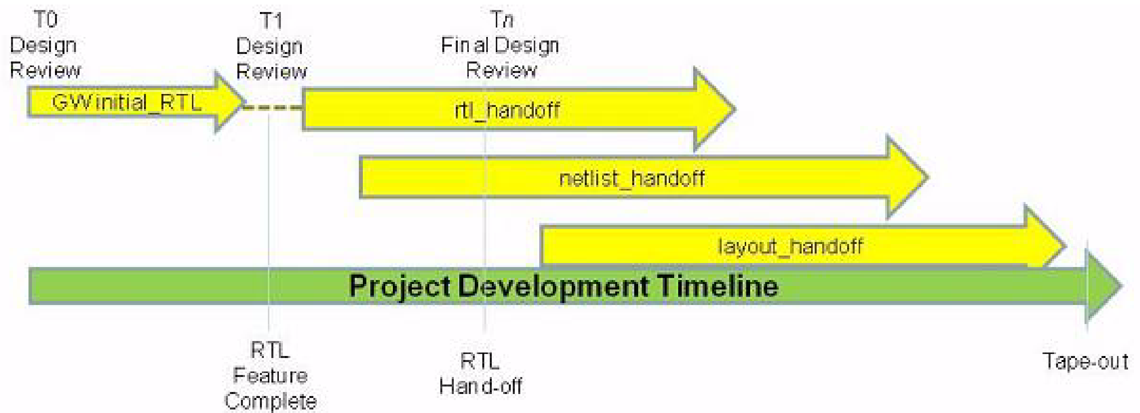
- *Block/IP*
- *SoC RTL*
- *SoC Netlist*

The above fields of use are highlighted in the following figure:



Each design goal is directly addressed by pre-packaged SpyGlass goals. These goals have been tested and fine-tuned for high impact results and minimal noise.

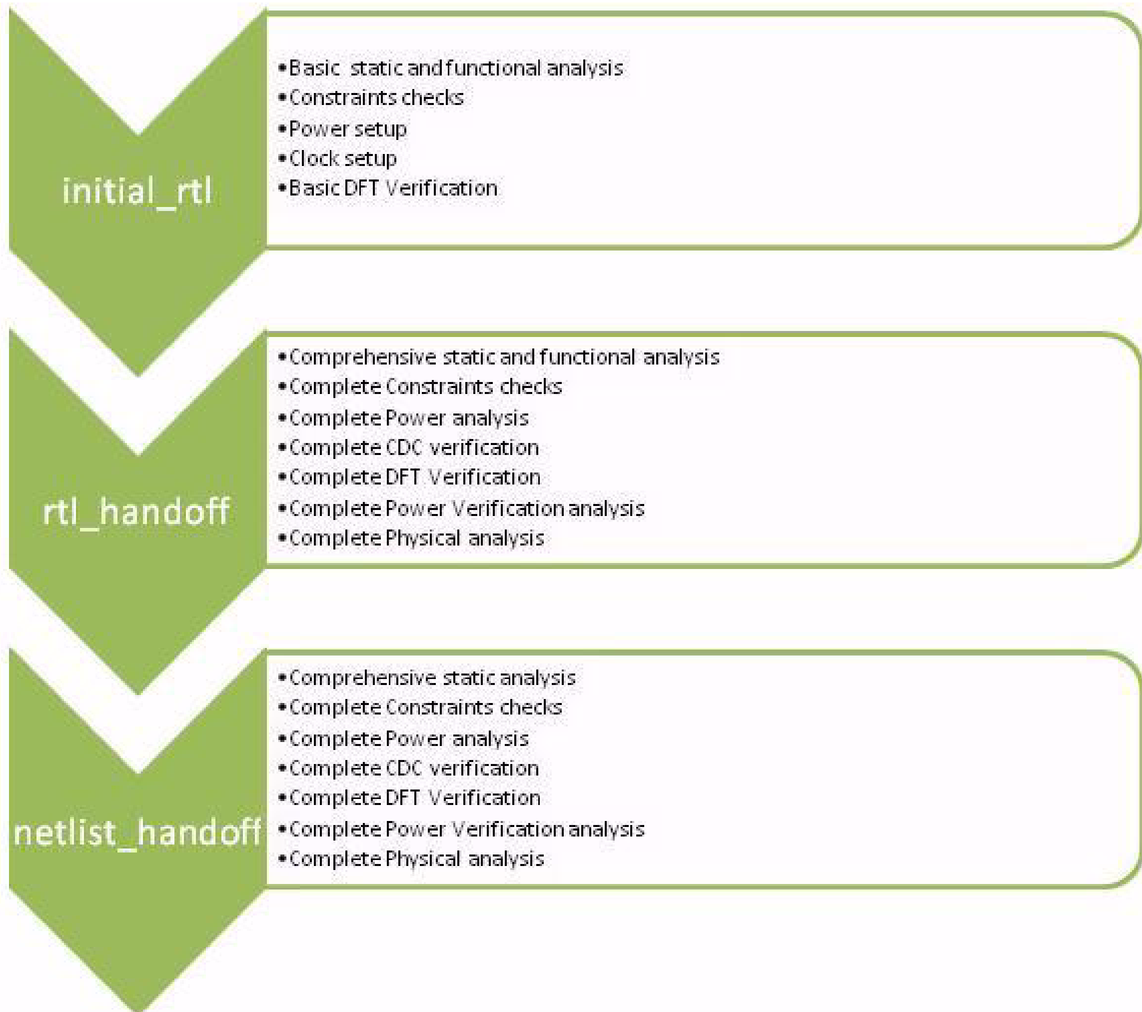
The development process of large designs is divided into various stages, as shown in the following figure:



Block/IP

This stage involves the development of a new RTL. The process of the development of a new RTL goes through progressive RTL refinement starting with simpler goals that meet the functional requirements, such as functional correctness and simulation and synthesis readiness of the code. As the RTL code and design constraints mature, the design goals evolve to performance, testability, and meeting handoff requirements.

In this stage, the GuideWare methodology recommends the following flow:



The above flow represents an ecosystem of goals. You can customize this flow based on your specific requirements and workflow of your design project.

The following sections describe the details of each stage.

- *Initial RTL Development*
- *RTL Handoff*
- *Netlist Handoff*

Initial RTL Development

The initial RTL design goal set contains a set of checks for the stage of the project when the RTL is still in coding development and may not be functionally complete. The idea is not necessarily to be clean all at once, but provide a starting point for getting to the clean RTL.

The design team faces the following lint related challenges during this stage:

- Issues related with correct code capture
- Issues related with simulation and synthesis
- Issues with basic connectivity
- Issues related with basic structure like combinational loops and multiple drivers

GuideWare recommends the designer focus first on getting to lint_rtl clean, and then work on the clock/reset correctness, basic DFT coding correctness, and check SDC constraints. By the time the design reaches a "feature complete" milestone, ideally the initial_rtl goals have been run and are clean. At this point, the design would progress to the rtl_handoff stage where additional requirements are added to the existing set of checks.

RTL Handoff

The rtl_handoff goals are a super-set of the initial_rtl goals. This stage contains the complete set of recommended RTL Handoff checks.

The design team faces the following lint related challenges during this stage:

- Issues related with verification regressions and associated bug fixes

- Issues related with incomplete handoff
- Providing closure on various implementation issues, such as synthesizability, timing, constraints, clock domain crossings, testability, congestion/routing, and power management

An incomplete handoff results in expensive and unpredictable error-prone iterations during the SoC integration phase. Handoff is assumed to be the hand-off from the RTL design team to the post-synthesis implementation team or hand-off to System Integration (sub-system or SoC) integration. Since the hand-off process is typically iterative, it is not necessarily expected that all goals will be clean at the first hand-off, but at least the issues will be known and can be communicated to the consumers downstream.

Netlist Handoff

The netlist_handoff goals are designed to check post-synthesis netlist prior to layout. These checks are ideal for hand-off to the backend physical implementation team or ASIC handoff.

SoC or Sub-system Integration

During Soc or sub-system integration the design architect needs to stitch the block IPs. These block IPs may have been developed internally or selected from a third party vendor. Depending on the extent of reuse of these IPs, some of them may not have gone through the process of lint cleaning. This is typically seen in legacy IPs which have existed in the prior incarnations of the design. This creates new challenges during the integration phase.

Challenges Involved in the Selection of Third Party or Internal Legacy IP

While selecting an IP, either internal or external, the design teams are usually concerned about the following challenges:

- Understanding the profile of an IP

  The information about IP profile is critical for effective integration of an IP into the target SoC. This information includes the usual IP statistics

about approximate block size, number of flip-flops and latches, ROM/ memory and other large structures used in the IP block, overall clock and reset architecture, voltage/ power domain, and so on.

- Identifying specific risks associated with an IP

  Some of the challenges that must be considered are unsynchronized clock-domain crossing, inaccurate or incomplete timing exception specification, inconsistency within SDC or across SDC and RTL description, and errors in clock gating/ isolation-logic or level-shifting logic, if applicable.

- Estimating the adaptability of an IP

  The challenge involved in estimating the extent to which an IP is adaptable for a given target application might relate to testability, power domain, and voltage domain adaptation, and other fine-tuning to meet SoC performance targets (if applicable).

Challenges Involved During SoC Integration

During SoC integration, the integration team faces the following challenges:

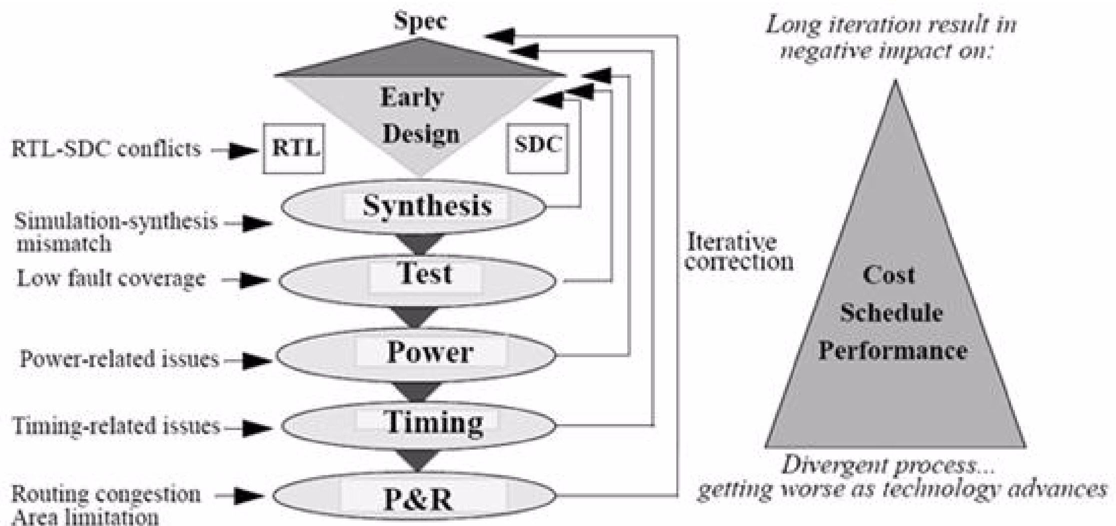
- Issues related with functional verification and implementation
- Issues related with interconnection of blocks
- Issues related with clock and reset planning, I/Os, floor planning, testability, JTAG, scan chains, and power management

Issues in the early stages of design development usually surface as critical bugs in the late stages of design implementation. Such issues, if not resolved in the early stages, result in iterations that are costly and time-consuming.

For example, at a particular stage of design development, you might get feedback about synthesizability, testability, or power from implementation or integration team. This may require you to go back to a previous stage, and resolve those issues. Once those issues are resolved, there might be another issue in some RTL block, which might cause another global iteration through the process.

Essentially, resolving an issue late in the design cycle might require multiple iterations. In addition, resolving an issue might lead to introducing another issue in the block/subchip. Figure 2 shows that fixing issues that

RTL designers encounter at different stages of development is an iterative process. It also shows that identifying an issue late in the development stage negatively impacts the project cost, schedule, and performance.



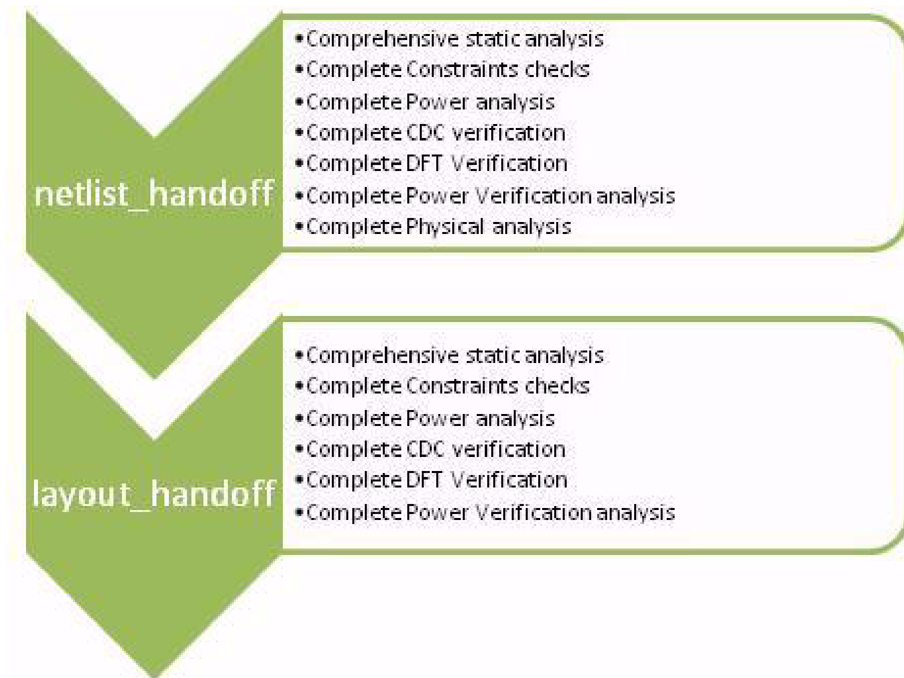
SoC RTL

This stage involves the verification of an SoC design or a subset of design (subsystem) that has been integrated by using various blocks. This field of use involves checks related to inter-block/inter-IP issues and consistency across blocks. In addition, it ensures that block constraints are consistent with SoC constraints.

The following image illustrates the key checks performed for each design stage:

SoC Netlist

The following image illustrates the key checks performed for for each design stage:



Netlist Handoff

The netlist handoff goals are intended to check a design which is ready for floor-planning, layout, and backend implementation.

Layout Handoff

The layout handoff goals are intended to check a design which has gone through floor-planning, layout, and so on, in preparation for tape-out.

Approach - How to Make Your Design Ready

During RTL Development

The process of the development of a new RTL goes through progressive RTL refinement starting with simpler goals that meet the functional requirements, such as functional correctness and simulation and synthesis readiness of the code. As the RTL code and design constraints mature, the design goals evolve to performance, testability, and meeting handoff requirements. In this field of use, the methodology recommends a four-stage flow.

Initial RTL Development

During this stage, an initial version of the RTL is completed, and an initial set of SGDC constraints are available. This stage involves basic structural and sanity checks of the design (and constraints, wherever appropriate). In addition, issues related to connectivity, synthesizability, preliminary clocks, and reset integrity issues, such as glitches and clock-MUXing are also checked during this stage.

For this stage, methodology recommends a set of goals that can be used by individual RTL designers to correct the issues within their own desktop environment before simulation and synthesis tasks can begin. These goals are recommended to be used quite frequently. In some cases, designers use these goals before checking-in their RTL code. Waivers, if any, should be captured on an ongoing basis.

This stage may involve some micro-architectural changes related with bus widths, RAM/ROM usage, and clock phase/frequency refinements. It is important to ensure that the proposed micro-architectural changes are reflected in the RTL without any adverse impact on the implementation issues.

RTL Handoff

This is the final completion and handoff stage for the RTL. By this stage, it is assumed that the RTL has already been refined as per the methodology. Most checks are applicable at this point before backend implementation

begins. During this stage, the micro-architecture and majority of the logic is stable. SpyGlass goals are used to perform handoff checks with appropriate waiver definitions.

At this milestone, the block is expected to be clean and all the necessary inputs are expected to be in place before you perform the final SpyGlass run. It is also expected that the user is able to share the setup, constraints, waivers, reports, and so on, with the customer.

Netlist Handoff

This stage when the handoff RTL is synthesized and netlist is handed off for backend implementation. All structural checks at RTL handloff are applicable here. In addition certain ERC checks are appropriate at this stage. This netlist is used by many groups as a starting point for their tasks (such as floorplanning, test insertion, power estimation, and reduction analysis). SpyGlass goals are used to perform handoff checks with appropriate waiver definitions.

The following table describes recommended Base SpyGlass goals for each of the three stages of the new Block/IP development.

#	Goal Name	Initial_ RTL	RTL_ Handoff	Netlist Handoff	Description
1	lint_rtl			N/A	This goal checks: Basic connectivity issue in the design, such as floating input, width mismatch, etc. simulation issues in the design, such as incomplete sensitivity list, incorrect use of block/ non-blocking assignments, potential functional errors and possible simulation hang & simulation race cases. unsynthesizable constructs in the design and code that can cause RTL vs. gate simulation mismatch. structural issues in design that affect the post-implementation functionality or performance of the design. Examples include multiple drivers, high fan-in MUX, and synchronous/asynchronous use of resets. These checks should be run after every change in RTL code prior to code-check-in.
2	lint_netlist	N/A	N/A		This is similar to lint_rtl but applicable only for netlist_handoff

#	Goal Name	Initial_ RTL	RTL_ Handoff	Netlist Handoff	Description
3	design_audit	Optional	Optional	Optional	The aim of this goal is to gather statistics of the design. This information may not be needed when RTL is still being actively coded. However, when RTL is somewhat complete, this information is useful to get an overall profile of the design. The audit report has the following information: • Basic Design Data Section • Top-Level Design Units Section • Black Boxes Section • Gray Boxes Section • Unsynthesizable Design Units Section • Parameters/Generics Section • Macros Section (For Verilog only) • Library Section (For VHDL only) • Line-size Section • Design Hierarchy Section • Design Size Statistics Section • Control Signals Section Design Elements Statistics Section
4	Clock_reset_ integrity				The aim of this goal is to check the integrity of clock and reset logic in a design. This includes fixing basic clock issues, race and glitch issues and reset logic issues.

#	Goal Name	Initial_ RTL	RTL_ Handoff	Netlist Handoff	Description
5	lint_rtl_fast	Optional	N/A	N/A	This goal is a subset of lint_rtl that checks for design readiness for simulation and potential simulation-synthesis mismatches. The will limit checks to pure RTL and elaboration and runs faster since it will not synthesize the design.
6	lint_abstract	N/A			This goal is used to create an abstract view of the RTL or netlist of the block when it is ready for handoff. This implies the block has been cleaned using the prior goals. The abstract view provides interface level information of the block, so that during the SoC level lint check the block RTL/netlist don't have to be read. Instead the abstract provides all relevant block information for faster SoC analysis. The block abstract view along with all constraints and waiver files is handed off by the block owner to the SoC integrator.

Waiving Messages

During design analysis, you may want to suppress the display of certain violation messages that may not represent a serious problem or messages that you may want to ignore at that point of time. You can suppress the display of such messages by using waivers. Waivers need to be applied cautiously. Waivers are written in a separate file and can be used with the modified source files as long as the modifications do not invalidate the design constraints.

Use the waive command to waive a message by various categories, such as by source files, by design units, by rules, etc. The waive command specifications are supplied in a waiver file.

A sample file containing waive command specifications is shown below:

```
################# Sample SpyGlass Waiver Commands
##############

##### Single Option (File/DU/Rule) Waivers ############

## Waive all violations for a design file or set of design
files

waive -file "/apps/rtl/imp_controller.v"


## Waive all violations for a design module or group of
design

## modules

waive -du "ahb_transmit"


## Waive all violations of a rule or group of rules

waive -rule "W164a"


################### Double Option Waivers
#########################

## Waive all violations of a particular rule for a design
module/

## file

waive -file "/apps/rtl/imp_controller.v" -rule "W164a"

waive -du "ahb_transmit" -rule "W164a"


################### Multi Options Waivers
#########################

## Waive all violation of warning severity related to a
particular

## net/design object for a design unit.
```

```
waive -regexp -du "ahb_tran" -severity="warning" -msg
".*test.*"


## Waive all clock rule violations arising due to black boxes
for a

## group of design units

waive -regexp -du "ahb_.*" -rule "Clk_Gen01a" -msg ".*black-
box.*" -comment "This is a comment for review purposes"
```

While creating waiver, it is recommended to adhere to the following guidelines.

- Avoid using waivers with line numbers. When a design changes, the line numbers can move causing the waivers to be invalidated and possibly applied at the wrong place. Consider using pragma rather than a waive command in such cases. e.g. `wire w1, w2;    //spyglass disable W120, W121`
- Always include `-rule` with `-msg` option. This will facilitate migration in case the message changes. The migration script will be able to map the old message to the new format.
- Do not use just `-rule` (i.e. "`waive -rule W123`"). This will increase runtime, since waive is a post processing step. If you do not want the rule to be run, use `set_goal_option ignorerules <rule-name>` option instead.
- Apply regular expressions carefully. Regular expression give the user the flexibility, but can result in longer run time
  - Limit regular expression to the design object. Do not apply regular expression on the static part of a message. E.g. if the message is "Signal '`a.b.n`' has multiple simultaneous drivers", the static part of the message (underlined) is "Signal *<signal_name>* has multiple simultaneous drivers". Apply regular expression only to *<signal_name>*, which is the variable part of the message.

    ```
    waive -du BLOCKA -rule W415 -regexp -msg "Signal*" - Not
    recommended

    waive -du BLOCKA -rule W415 -regexp  -msg "Signal '.*n'
    ```

```
has multiple simultaneous driver" - Recommended
```

- ❒ Limit using the `-regexp` option, if other fields can also result in multiple matches. To limit regular expression to a single field, include them in `m%<string>%` quotes. Consider the waiver command

  ```
  waive -du BLOCKA -rule W415 -regexp  -msg "Signal '.*n'
  has multiple simultaneous driver"
  ```

  If the design contains other blocks like `BLOCKAA`, or `BLOCKA1`, the waiver example shown above will get applied to all the blocks where the signal names match, which may not be intended. This should be replaced by

  ```
  waive -du BLOCKA -rule W415 -msg m%Signal '.*n' has
  multiple simultaneous driver%
  ```

- ❒ When message contains special characters like "*" or "?", which have a special meaning in regular expressions ('*' match any string, '?' match single character), enclose the message in q%% quotes to prevent any incorrect interpretation of these special characters in the message.
- ❒ Use "^" and "$" to anchor the match to beginning or end of line, if required

- Escape using backslash (\) existing meta characters (`[ ] * . - + | ? ^ $ \`) before starting. Always use the –comment field to comment your waivers
- Review your waivers using waiver report. Utilize the message counts in the user interface to make sure waiver is not over applied

These guidelines will facilitate maintenance, migration of waiver across release and reuses of waivers, when a block is being integrated. For details on using the `waive` command, refer to the *SpyGlass Explorer User Guide*.

During SoC Integration and Implementation

During SoC design or a subset of design (sub-system) that has been integrated by using various blocks, consistency across blocks is required. This field of use involves checks related to inter-block/inter-IP issues. In addition, it ensures that block constraints are consistent with SoC constraints. In this field of use, the methodology recommends a four-stage flow:

SoC / Sub-system Integration (of RTL Blocks)

During this stage, the SoC/sub-system integration team assembles the RTL blocks and IPs to form a SoC/sub-system. These RTL blocks are usually designed by different teams. The design teams may also use third party or legacy IPs.

The goals used during this stage target the following objectives:

- Check the complete design intent captured in individual blocks and their assembly
- Correct various inter-block issues, such as combinational loops and unconnected ports

During this stage, the intent is to clean the RTL before production level synthesis begins. The following table shows goals for SoC RTL stage:

#	Goal Nam	Intial RTL	RTL Handoff	Description
1	lint_rtl			This goal checks • Basic connectivity issue in the design, such as floating input, width mismatch, etc. • simulation issues in the design, such as incomplete sensitivity list, incorrect use of block/ non-blocking assignments, potential functional errors and possible simulation hang & simulation race cases. • unsynthesizable constructs in the design and code that can cause RTL vs. gate simulation mismatch. • structural issues in design that affect the post-implementation functionality or performance of the design. Examples include multiple drivers, high fan-in MUX, and synchronous/asynchronous use of resets. These checks should be run after every change in RTL code prior to code-check-in.
2	design_audit	Optional	Optional	The aim of this goal is to gather statistics of the design. This information may not be needed when RTL is still being actively coded. However, when RTL is somewhat complete, this information is useful to get an overall profile of the design. The audit report has the following information: • Basic Design Data Section • Top-Level Design Units Section • Black Boxes Section • Gray Boxes Section • Unsynthesizable Design Units Section • Parameters/Generics Section • Macros Section (For Verilog only) • Library Section (For VHDL only) • Line-size Section • Design Hierarchy Section • Design Size Statistics Section • Control Signals Section • Design Elements Statistics Section

#	Goal Nam	Intial RTL	RTL Handoff	Description
3	clock_reset_integrity			The aim of this goal is to check the integrity of clock and reset logic in a design. This includes fixing basic clock issues, race and glitch issues and reset logic issues
4	lint_rtl_fast	Optional	N/A	This goal is a subset of lint_rtl that checks for design readiness for simulation and potential simulation-synthesis mismatches. The will limit checks to pure RTL and elaboration and runs faster since it will not synthesize the design.
5	lint_abstract	N/A		This goal is used to create an abstract view of the RTL of the subs-system when it is ready for handoff. This implies the sub-system has been cleaned using the prior goals. The abstract view provides interface level information of the sub-system, so that during the SoC level lint check the sub-system RTL doesn't have to be read. Instead the abstract provides all relevant information for faster SoC analysis. The sub-system abstract view along with all constraints and waiver files is handed off by the block owner to the SoC integrator.
6	lint_abstract_validate	N/A		During the integration stage the SoC integrator may be using the abstract view of the block or the sub-system instead of the RTL/netlist of the block. This goal ensures that all assumptions made during the block level analysis to generate the abstract view matches the SoC level constraints (case analysis).

SoC / Sub-system Integration (of Netlist Blocks)

Netlist Handoff

This netlist is used by many groups as a starting point for their tasks (such as floorplanning, test insertion, power estimation, and reduction analysis). During this stage, third party tools modify the preliminary netlist for scan and BIST insertion and power-related gating. This version of netlist is

known as pre-layout netlist by most of the design teams. The goals used during this stage ensure that the original design intent is not adversely impacted during these modifications.

The goals and sub-methodologies recommended for this stage ensure the integrity of the complete SoC-level netlist from ERC perspective.

Layout Handoff

During this phase, the SoC post layout netlist is closest to silicon. It is important to ensure final integrity of this post-layout netlist before tape-out. Recommended goals allow the designer to ensure integrity of post-layout netlist during the ECOs and before the final handoff for tape-out.

The following table describes recommended Base SpyGlass goals for SoC Netlist:

#	Goal Name	Netlist_ Handoff	Layout_ Handoff	Description
1	lint_netlist			This goal checks • Basic connectivity issue in the design, such as floating input, width mismatch, and so on. • simulation issues in the design, such as incomplete sensitivity list, incorrect use of block/ non-blocking assignments, potential functional errors and possible simulation hang & simulation race cases. • unsynthesizable constructs in the design and code that can cause RTL vs. gate simulation mismatch. • structural issues in design that affect the post-implementation functionality or performance of the design. Examples include multiple drivers, high fan-in MUX, and synchronous/asynchronous use of resets. These checks should be run after every change in RTL code prior to code-check-in.

2	design_audit	Optional	Optional	The aim of this goal is to gather statistics of the design. The audit report has the following information: • Basic Design Data Section • Top-Level Design Units Section • Black Boxes Section • Gray Boxes Section • Unsynthesizable Design Units Section • Parameters/Generics Section • Macros Section (For Verilog only) • Library Section (For VHDL only) • Line-size Section • Design Hierarchy Section • Design Size Statistics Section • Control Signals Section • Design Elements Statistics Section
3	clock_reset_integrity			The aim of this goal is to check the integrity of clock and reset logic in a design. This includes fixing basic clock issues, race, and glitch issues and reset logic issues
4	lint_abstract			This goal is used to create an abstract view of the netlist of the sub-system when it is ready for handoff. This implies the sub-system has been cleaned using the prior goals. The abstract view provides interface level information of the sub-system, so that during the SoC level lint check the sub-system netlist doesn't have to be read. Instead the abstract provides all relevant information for faster SoC analysis. The sub-system abstract view along with all constraints and waiver files is handed off by the block owner to the SoC integrator.
5	lint_abstract_validate			During the integration stage the SoC integrator may be using the abstract view of the block or the sub-system instead of the netlist of the block. This goal ensures that all assumptions made during the block level analysis to generate the abstract view matches the SoC level constraints (case analysis).

Managing Design Hierarchy

During SoC or sub-system, SpyGlass enables you to specify the portions of your design that you want to consider and/or exclude from the scope of SpyGlass analysis. You can specify this information by making some design units as the top-level design units and by stopping some design units. Consider the example below.

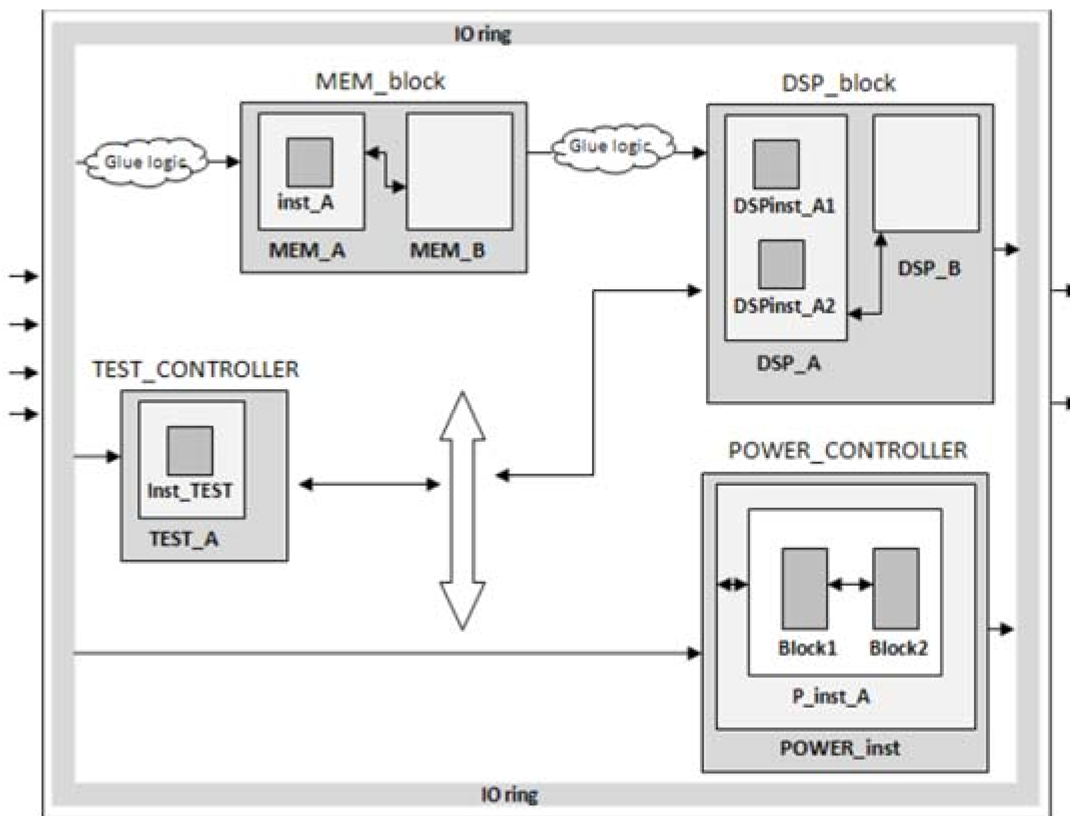


FIGURE 2.

By default, all the design units (`MEM_BLOCK`, `DSP_block`, `TEST_CONTROLLER`, and `POWER_CONTROLLER`) as well as the design units instantiated directly/indirectly within these design units are considered for SpyGlass analysis. Now, among these units, you can specify

the top-level design units that should be considered for SpyGlass analysis by using the following command in the Console project file:

```
set_option top <du-name>
```

To exclude some design units from the scope of SpyGlass analysis, specify the following command in the Console project file:

```
set_option stop <du-name>
```

Example 1:

```
set_option top DSP_block
set_option stop DSP_A
```

Only the design unit, `DSP_B`, is considered for SpyGlass analysis

Example 2:

```
set_option top POWER_CONTROLLER
set_option stop Block1
```

The design units, `POWER_inst`, `P_inst_A`, and `Block2` are considered for SpyGlass analysis.

Example 3:

```
set_option stop P_inst_A
```

Only `P_inst_A` is excluded from the scope of SpyGlass analysis. All the other design units including the design units instantiated directly/indirectly within `P_inst_A` are considered for SpyGlass analysis, since you have not specified the top-level DU name.

If the user does not wish to stop, he can provide an abstracted ILM (interface level model) model for the block.

On the other hand, if SoC integrator is running the tool with a top-down approach (or even with bottom up approach), they can use following options such that they have the control to run the tool on specific IP or design units (DU) even though some other lower level or parallel level DU/IPs are not complete.

```
set_option checkip <du-name>
```

This option specifies the design units (the specified unit and all its sub-hierarchy) on which the rule-checking should be done. When you specify this option in console project file, SpyGlass not only considers the design unit specified by this option for rule-checking, but also considers all those design units starting from the top in the hierarchy till the design unit specified by this option.

Consider an example, as shown in the above hierarchical block view:

Now, if you specify `set_option checkip Mem_block`, SpyGlass would analyze the modules, `TOP`, `Mem_block`, `Mem_A` and `Mem_B`, and will consider only these modules for rule-checking.

Additionally, there is another option named `checkdu`, which allows analysis of only that specific design unit and not on any design unit which is below its hierarchy. All the design units instantiated under the design unit specified by this option are treated as gray boxes.

For the rest of the design units, rule-checking will be bypassed.

When you specify this option in project file, SpyGlass not only considers the design unit specified by this option for rule-checking, but also considers all those design units starting from the top in the hierarchy till the design unit specified by this option.

For example, based on above figure, if you want to analyze only the logic present at top level and not anything inside the sub-blocks, you can either do it using '`set_option stop`' but there you need to specify individual sub-block names. Instead of that, you can specify:

```
set_option checkdu top
```

That will make the tool analyze only the top-level logic and will make all the sub-blocks treated as gray boxes. Similarly, if you apply '`set_option checkdu DSP_block`', that means only glue logic at top and `DSP_block` will be analyzed for rule checking.

You can also use combination of `checkdu` and `checkip` but please note that checkdu is given preference over checkip if these options are specified for same design unit.

Aligning Block Level Waivers

During SoC or sub-system development process, waivers generated either for RTL development or IP block qualification can be used.

Hierarchical Waivers

During SoC integration, if the block has already been verified using SpyGlass, then chip-level designers can use all the waivers of a block specified by the block-level designer. This can be implemented by using the `-import` option of the waive constraint, as shown below:

```
waive -import <block-name> <block-waive-file>
```

The above command imports the waiver file, *<block-waiver-file>*, of the block, *<block-name>*.

You can specify waivers for individual blocks separately in the top-level chip by specifying multiple waive `-import` constraint specifications. You can also specify multiple waiver files for a given block in multiple `waive -import` constraint specifications.

Consider the following example in which `B1` and `B2` are the two blocks inside the top-level chip, and `B1.swl` and `B2.swl` are the waiver files applied to these blocks, respectively:

```
waive -import B1 B1.swl
waive -import B2 B2.swl
```

Waiving an IP

During SoC integration, if an IP block has no SpyGlass waivers, then the user can waive all the violations on the IP (including all modules underneath it) and only focus on issues at the integration level. This can be achieved using the `waive -ip` command as shown below.

```
waive -ip <ip_name|<ip_list>|<logical_library_name> [-rule|-
rules <rule_list>] [-msg <message>] [-severity <label>] [-
except <rule_list>]
```

Consider the following example in which `B1` and `B2` are the two blocks inside the top-level chip. To waive all message in the hierarchy below `B1` and `B2`, use:

```
waive -ip B1, B2
```

To waive message from a particular rule, use

```
waive -ip B1, B2 -rule CombLoop
```

Aligning Block Level Constraints

During SoC or sub-system development process, SGDC (SpyGlass Design Constraints) constraints created during RTL development or IP block qualification can be used.

Hierarchical Constraints

This chip-level or sub-system level SGDC file should contain the sgdc -import command(s) for block-level SGDC file(s) that need to be imported. For example, a chip-level SGDC file may contain the following specification:

```
current_design <du-name>
sgdc -import <block-name> <block-level-SGDC-file>
```

The <block-name> can be specified in any of the following formats:

- module
- entity
- entity.architecture

To create the migrated SGDC run, SpyGlass with the option

```
set_option gen_hiersgdc yes
```

This will generate a file *<module-name>.sgdc* in the *gen_hiersgdc/ spyglass_reports/imported_sgdc* directory. The above specification can be given multiple times for different blocks in the same chip-level SGDC file. The top-level SGDC file that is generated includes all the migrated block level SGDC files. This file also contains those migrated SGDC commands that are common in two or more block-level SGDC output files. In subsequent chip-level analysis, you should specify the generated top-level SGDC file instead of migrated block level SGDC files.

Understanding Constraints in Block Level Scope

SpyGlass allows scoping mechanism in SGDC commands. The scoping mechanism is implemented by using the `::` operator. Consider the

following example:

```
current_design sub-system
set_case_analysis -name M::i1.i2.net -value 0
```

In the above example, `M::` specifies the scoping mechanism, which means to find all instances of module M in:

- All instances of design unit, sub-system if that design unit is not a top-level design unit
- Design unit, sub-system, if it is a top-level design unit

Then, the value 0 is applied on net `i1.i2.net` in all these instances.

If a value generated due to scoping conflicts with an explicit value specified by the user, the value generated by scoping is deleted. This provides you the flexibility to override one or more specifications generated through scoping.

Scoping can also be done via the `current_design` command. Consider the following example:

```
current_design top
…
current_design reset_sync_block
reset -name srst_pin -value 1 -sync
```

This will put a `reset` constraint on the pin `srst` or each instance of `reset_sync_block`.

Using Block Level Abstract

An abstract view is a representative model of a block that contains relevant block information required during SoC-level verification.
For example, it contains block information, such as combinational path details, boundary registers and related clock/reset information, domain information, and boundary constraints used to analyze the block.
An abstract view contains such information in the form of SGDC constraints. SpyGlass provides a way to generate and consumes these abstract view. For more details, please refer to the *SpyGlass SoC Methodology Guide*.

Conclusion

As the chip complexity rises, various design issues cause silicon re-spin risk and poorer chip quality in terms of area, power, and timing. SpyGlass® is a powerful and extendible tool for analyzing Hardware Description Language (HDL) designs.

SpyGlass recognizes the issues related with synthesis, simulation, test, power, clocks, and constraints at an early stage (RTL or netlist). In addition, it guides you to fix and optimize your design and constraints that results in:

- Fewer synthesis iterations
- Higher test coverage
- Lower power consumption
- Properly implemented clock gating and voltage island strategies
- Faster timing closure with correct SDC, false paths, and multi-cycle paths
- No silicon re-spin due to clock domain crossing issues

SpyGlass can analyze designs written in languages, Verilog (including SV) and VHDL. In addition, SpyGlass supports mixed-language and DEF designs. SpyGlass supports both RTL and netlist abstraction for analysis. You can use SpyGlass to perform any of the several industry standard HDL analysis and assessment programs, including OpenMORE™ and STARC™. You can also use SpyGlass to analyze HDL source code early in the design stage for syntax, semantic, and structural content, and perform complex checks later in the development process. For example, you might initially use SpyGlass to check Register Transfer Level (RTL) HDL descriptions. Later, you might use it to analyze gate-level designs or designs that include both RTL and structural descriptions.

SpyGlass provides the following features:

- Support for Verilog (including SV) and VHDL
- Support for rich suite of built-in rules including the following checks:
- File checks, such as file names, design units per file, and headers
- Naming checks on signals, ports, parameters, constants, clocks and other constructs
- Style and related checks

- Coding for synthesis and related checks
- Design practice and related checks
- Area, timing and synchronization checks
- Clock and reset checks
- SpyGlass DFT, SpyGlass Power Verify, SpyGlass Constraints related checks
- Support for a variety of report format options

Where as having a product is first step, without a proper methodology that suits the customer design flow, it is not effective. Users do not know which rules to apply at what stage. Too many rules applied to a stage leads to too many violations, only few of which are really critical. This creates a barrier in adoption.

In this document, we have laid out a recommended step-by-step methodology that applies to generic design flow.