SpyGlass® CDC Submethodology (for GuideWare 2017.12)

Version N-2017.12-SP2, June 2018

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

Introduction to SpyGlass CDC Methodology

Clocks that are asynchronous with respect to each other may reach different flip-flops at slightly different times in each cycle. This timing uncertainty may cause setup and hold-time violations randomly in the design resulting in functional failure in an SoC.

Such issues cannot be detected by using traditional verification methods, such as simulation and static timing analysis. You can detect them by using static clock-domain-crossing analysis and verification of SpyGlass CDC solution.

SpyGlass CDC solution enables you to detect clock-domain crossings at the RTL level and ensure that proper synchronization is added in the circuit.

This document introduces a methodology that you can use to verify clock-domain crossing (CDC) issues in your design by using the SpyGlass® tool suite. The document is useful for novice and advanced users of SpyGlass. Advanced users can proceed directly to the relevant sections of the document.

The following table describes the sections covered in this document:

Topic	**Information**
The CDC Issues	Describes basic CDC problems, such as metastability and complex synchronizers.
Using SpyGlass CDC Methodology to Solve CDC Problems	Describes a step-by-step solution towards a SpyGlass CDC-clean design by using any of the following flows: ● *SpyGlass CDC Methodology Flow* ● *SpyGlass CDC Hierarchical Verification Flow*

Goals of SpyGlass CDC Methodology

SpyGlass CDC methodology is integrated within GuideWare for different field of use. Below is a summary of goals (of SpyGlass CDC solution) deployed in various field of use of GuideWare. The set of goals (of SpyGlass CDC solution) used for each GuideWare stage is the same.

SpyGlass CDC solution: Block						
	Goals					
GuideWare Stage	**cdc_setup**	**cdc_setup_check**	**clock_reset_integrity**	**cdc_verify_struct**	**cdc_verify**	**cdc_abstract**
initial_rtl	O	M	M	M	-	-
rtl_handoff	O	M	M	M	M	M
netlist_handoff	O	M	M	M	M	M

SpyGlass CDC solution: SoC								
	Goals							
GuideWare Stage	**cdc_setup**	**cdc_top_down**	**cdc_setup_check**	**clock_reset_integrity**	**cdc_abstract_validate**	**cdc_verify_struct**	**cdc_verify**	**cdc_abstract**
initial_rtl	O	-	M	M	O	M	-	-
rtl_handoff	O	O	M	M	O	M	-	M
netlist_handoff	O	O	M	M	O	M	-	M
layout_handoff	O	O	M	M	O	M	-	-

NOTE: *M means mandatory and O means optional.*

In addition to the static SpyGlass CDC verification described here, SpyGlass CDC solution can be used to perform dynamic SpyGlass CDC verification. For dynamic SpyGlass CDC verification, SpyGlass CDC solution can generate simulation directives using Ac_meta01 rule that will inject metastability errors during simulation using your own testbench. These directives may cause additional simulation mismatches due to the effect of metastability.

Tool and Methodology Version

- SpyGlass Version: Version N-2017.12-SP2
- GuideWare Version: 2017.12

References

- SpyGlass CDC Rules Reference Guide
- SpyGlass Explorer User Guide

SpyGlass CDC Terminologies

The terminologies used in SpyGlass CDC are defined in the following table:

Terminology	**Description**
Clock domain	Refers to the clocks that have a constant phase relationship with each other. Typically, a clock, its inverted form, and its divided form is considered to be in the same domain. Divided forms have a constant phase relationship until the division ratios have a common factor. Divide-by-2 and divide-by-4 have constant phasing but divide-by-3 and divide-by-5 do not have constant phasing.
CDC (Clock Domain Crossing)	Refers to the path connecting a sequential element, flip-flop, primary input, or black box controlled by one clock domain to another sequential element, flip-flop, primary input, or black box clocked by another clock domain.
Synchronizer	Refers to the part of a design that transfers signal values across clock domains
Quasi-static	Refers to flip-flops that take constant values in a design. They may change values during setup and initialization of the design, or may change value when a block powers on or power off. Often, quasi-static flip-flops do not require synchronizers even if they are involved in clock domain crossings.
LCM	Refers to the least common multiple to identify a common clock period for a design with multiple clocks of different periods.
Correlated Signals	These are the signals whose combined values are used in the design. An example of such signals is state vector signals.

The CDC Issues

Clocks that are synchronous with respect to each other are known as same domain clocks, and clocks that are asynchronous to each other are known as different domain clocks.

Edges of clocks coming from the same clock domain are always aligned for all registers in the design and for all time throughout design run. As a result, if setup and hold time for a flip-flop input is considered, there is no risk in capturing the data of the flip-flop throughout the design.

However, clocks from different domains may reach different flip-flops at different times in each cycle during design run. This timing uncertainty may cause random setup and hold-time violations. Such problems may result in the following CDC issues:

- *Metastability*
- *Data Hold in Fast-to-Slow Crossings*
- *Data Correlation and Race Conditions*
- *Complex Synchronizers*
- *Issues Related to Reset Synchronization*

Metastability

Metastability is the design problem in which metastable values are created and propagated due to setup and hold-time issues in an asynchronous crossing.

The following figure shows an example of such an issue:

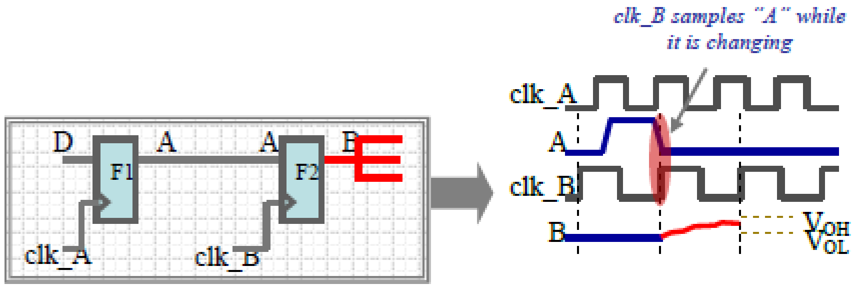


FIGURE 1. Metastability Issue

In the above figure, the metastable waveform generated at B is subject to interpretation by each branch in the fan-out of B.

One gate in a fan-out can perceive the metastable wave as the logical value 1 while another fan-out perceives the same net as 0. This free interpretation causes functional failure in the design.

To remove metastability, use the following approaches:

- Control signal synchronization

  Control signals crossing clock domains are typically synchronized by using multi-flop synchronizers. In such cases, multiple stages of flip-flops transform the metastable values to a *cleaner* 0 or 1 before it is passed to a downstream logic.

- Data signal synchronization

  Data signals are synchronized by using enable techniques where the data is first stabilized on the crossing path and then the destination flip-flop is enabled to capture the stable data (so the setup and hold time is not violated).

Traditionally, a clock domain crossing is seen as a path from one memory element to another. However, designers typically design interfaces that involve data, address, and control lines implementing complex synchronization protocols. For example, *Figure 2* illustrates a common FIFO where data is stored and read from a memory (data), pointers are designed to access the memory (address) for either read or write, and finally control logic that computes empty/full flags (control) is designed to ensure coherency and prevent metastability.

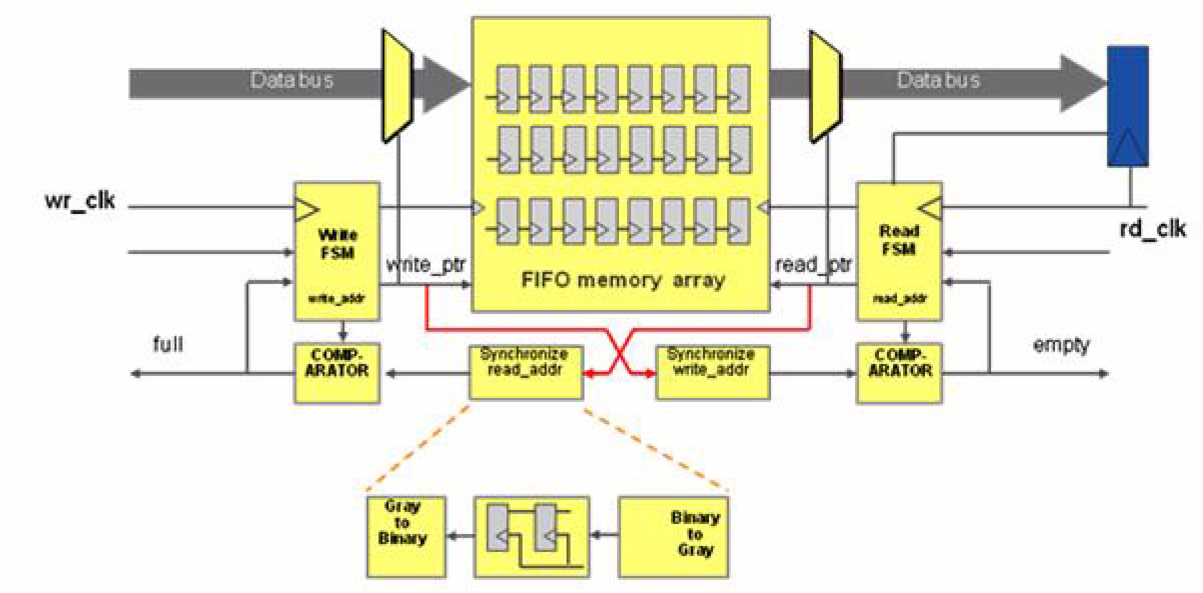


FIGURE 2. FIFO synchronizer involving data, address, and control logic

The key concept in common data synchronization techniques is a link between the sender and receiver of the data which ensures that the data is not captured while it is changing as this will cause an asynchronous event to propagate which can cause metastability. Based on this observation, an asynchronous interface is composed of a set of source signals, a destination signal (can be a bus), and a set of control logic. At least part of the control logic is responsible for proper synchronization. *Figure 3* illustrates a generic data crossing and signals involved in the synchronization.

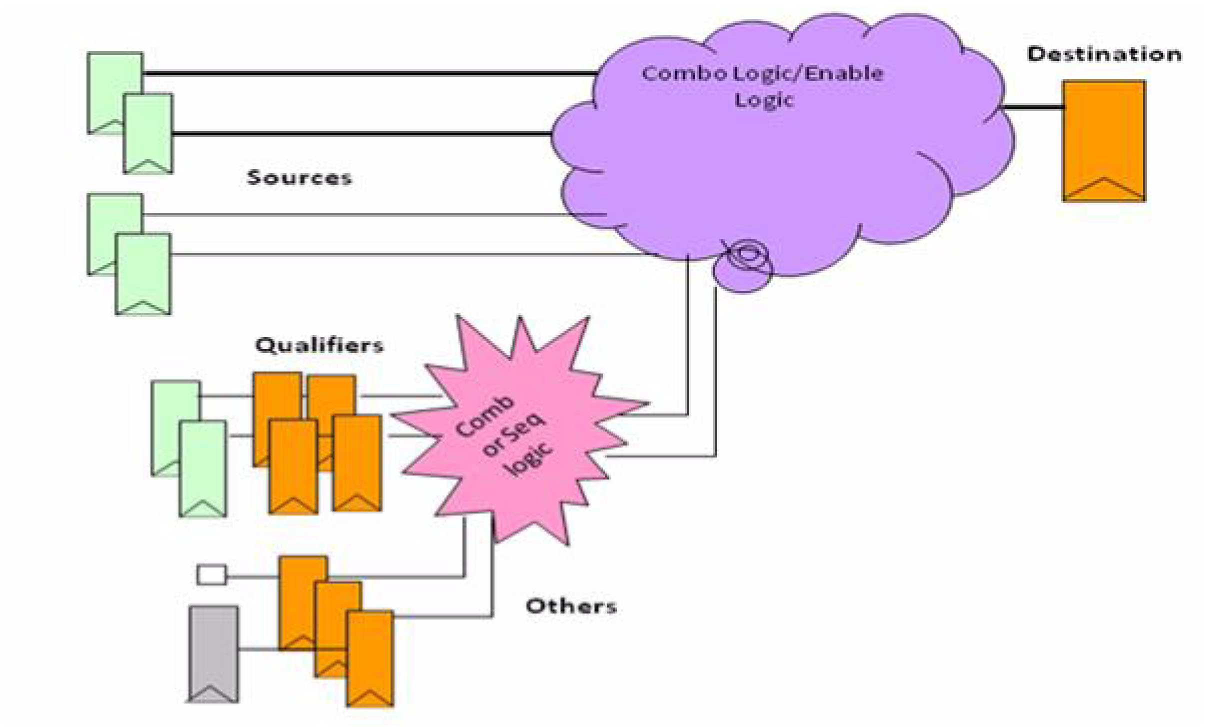


FIGURE 3. Generic Data Crossing

The following signals are typically involved in a crossing:

- **Destination of the crossing**: Single-bit or multi-bit signal receiving data from one or multiple domains
- **Source of the crossing**: Data, Address, Control signals crossing clock domain without being flip-flopped in the destination domain. There can be multiple sets of sources from different domain crossing to the destination
- **Qualifier**: A control signal, typically from the source domain synchronized in the destination domain (typically using multi-flop synchronization technique) responsible for ensuring that the source is stable when captured by the destination

- **Signals from the destination domain**: These signals are used as a control or for data computation/transformation purposes needed for the design.

  Such signals are typically not important for synchronization verification, however, it is not a good idea to perform complex computation, or bring complex control logic on a crossing. Increased combinational logic on a crossing increases the risk of asynchronous glitch, especially after synthesis and optimization. We recommend performing data transformation/computation in the source domain or in the destination domain and keeping the asynchronous interface very simple

NOTE: *Some custom interfaces such as source synchronous interfaces where the source is itself generated synchronously to the destination may not comply with the common synchronization techniques described here. Such structures are relatively rare and may require custom verification approaches.*

As is implied by such an interface, the synchronization protocol must ensure that the data is stable when the qualifier is asserted to trigger data capture. This is a functional requirement that cannot be validated by looking at a design structure alone.

Furthermore, the above functional requirement only ensures that the source will not violate timing that can cause metastability. If any combinational logic on the crossing glitches, it may cause either metastability or generation of an undesired pulse.

SpyGlass CDC Verification performs a fast structural analysis that identifies elements of synchronization that indicates user intent to synchronize the crossing in order to prevent metastability as well as glitch issues.

SpyGlass CDC verification solution relies on the following concepts to declare a crossing as synchronized:

- **Presence of qualifier signals**: One or more signals coming from the source domain and synchronized to the destination using multi-flop synchronizer or a user-specified synchronizer cell. Qualifiers may be present beyond sequential logic such as a receiver state machine.

NOTE: *Synchronous resets may have all characteristics of a qualifier. They can be signals generated in a source domain, synchronized in a destination domain, and even gating the data capture into the destination. It is very important to define synchronous resets for SpyGlass CDC solution to avoid false positives. As stated above, functional verification must complement the structural verification to guarantee synchronization correctness. Indeed, although synchronous resets have full appearance of a qualifier they will fail functional data/enable or data-hold checks.*

- **Gating mechanism**: The qualifier must converge with the source in order to block the transfer when the source is being set up. SpyGlass CDC solution accepts common gating mechanisms such as AND and MUX synchronizers. An XOR gate is not considered as an acceptable gating mechanism as it allows asynchronous transfer regardless of the value of a qualifier feeding the gate.

  Furthermore, to avoid glitches, re-convergences are allowed only before the crossing is gated. Any re-convergence of the same asynchronous source after it is gated is prone to glitch and is not accepted. This includes cases where each branch of the re-convergence is synchronized. *Figure 4* illustrates a re-convergence of the source, although each path is gated by a qualifier, the crossing is not considered as synchronized based on a pessimistic assumption that the re-convergence can produce a glitch. Indeed, the logic in this figure represents a XOR and this crossing must not be considered as a synchronized crossing.

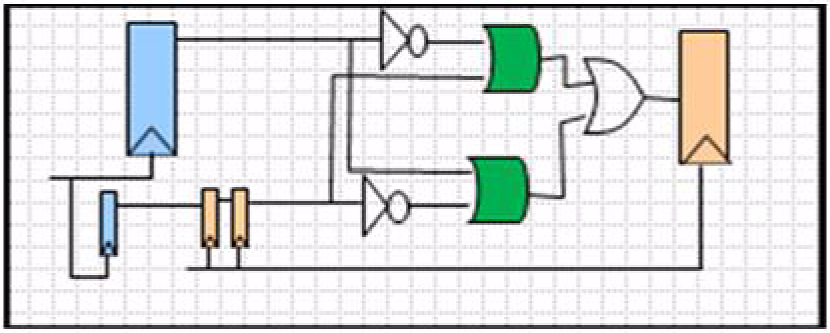


FIGURE 4. Synchronization not accepted

If source re-convergences are synchronized after they converge, then there is no glitch risk and the crossing is considered as synchronized. Such structures are considered as properly synchronized.

Note that there might be properly synchronized crossings where re-convergences exist on each branch of the crossing. However, such complex logic on a crossing is a risky design style that can cause glitches, especially after optimization and synthesis transformations. Due to the pessimistic nature of the analysis, such synchronized crossings are not considered as synchronized.

In case a data is found to be unsynchronized, it is important to understand the nature of the failure and the way to fix it. Following are possible reasons for synchronization failure:

- Lack of a qualifier: No qualifier converges with the crossing. This may be due to the presence of an unsynchronized control signal that is intended to qualify a crossing.
- Invalid gating of the crossing with a valid qualifier: This happens for example when the crossing and a qualifier converges on a XOR gate.
- A source diverges to multiple paths that are synchronized separately (refer to *Figure 3*)
- A source converges with the qualifier before the qualifier feeds the synchronizing gate
- Two sources from different domains converge before they are synchronized

Some simple examples of control and data synchronizers are shown in *Figure 5*:

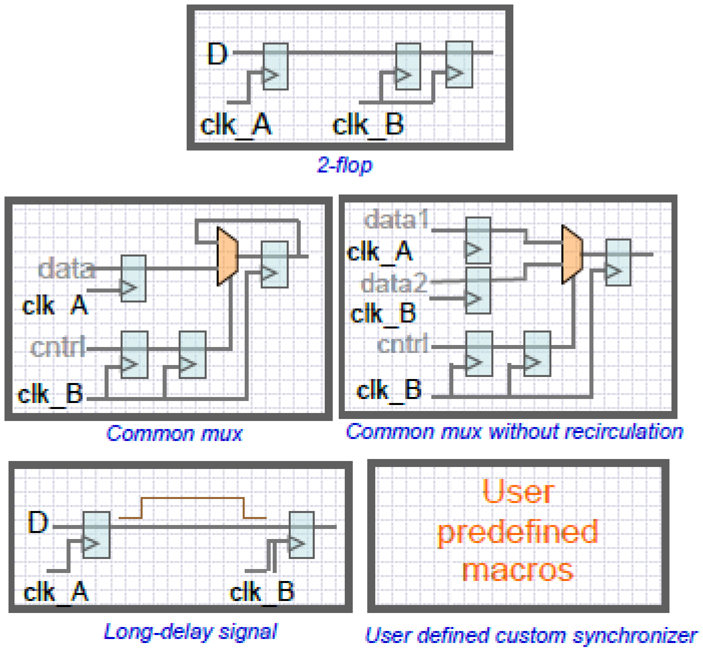


FIGURE 5. Common synchronization schemes

Data Hold in Fast-to-Slow Crossings

This issue appears when a short pulse generated in a fast clock domain is fed in a slow clock domain. In such cases, short signals may miss the active edge of the slow clock domain and they are not captured in the destination.

The following figure shows the data hold problem in fast-to-slow crossings:

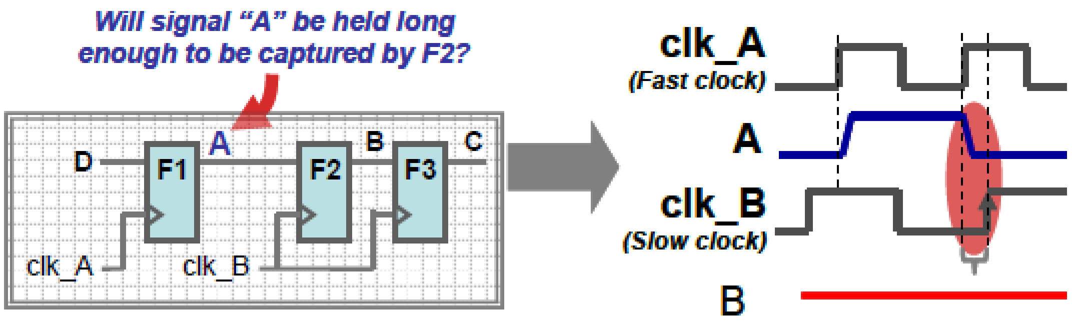


FIGURE 6. Example of Data Hold in Fast-to-Slow Crossings

To fix such issues, use the following approaches:

- Use a custom circuit to extend the pulse for at least one complete cycle of the slow destination clock.

  You must verify all the fast-to-slow crossings and ensure that such extenders exist and no short pulse is generated in the destination.
- In case of enabled flip-flops involved in a crossing, ensure that the data is stable before the enable is asserted and the data does not change when the enable is on.

Data Correlation and Race Conditions

If a source remains stable for long, its value is transferred to the destination. However, if the design has metastability issues, this transfer may not happen immediately. This can cause problems for *Correlated Signals* such that one or more signals are deferred relative to others.

This results in loss of correlation, which results in an unknown state at destination, thereby causing functional failure.

The following figure shows the example of such problem:

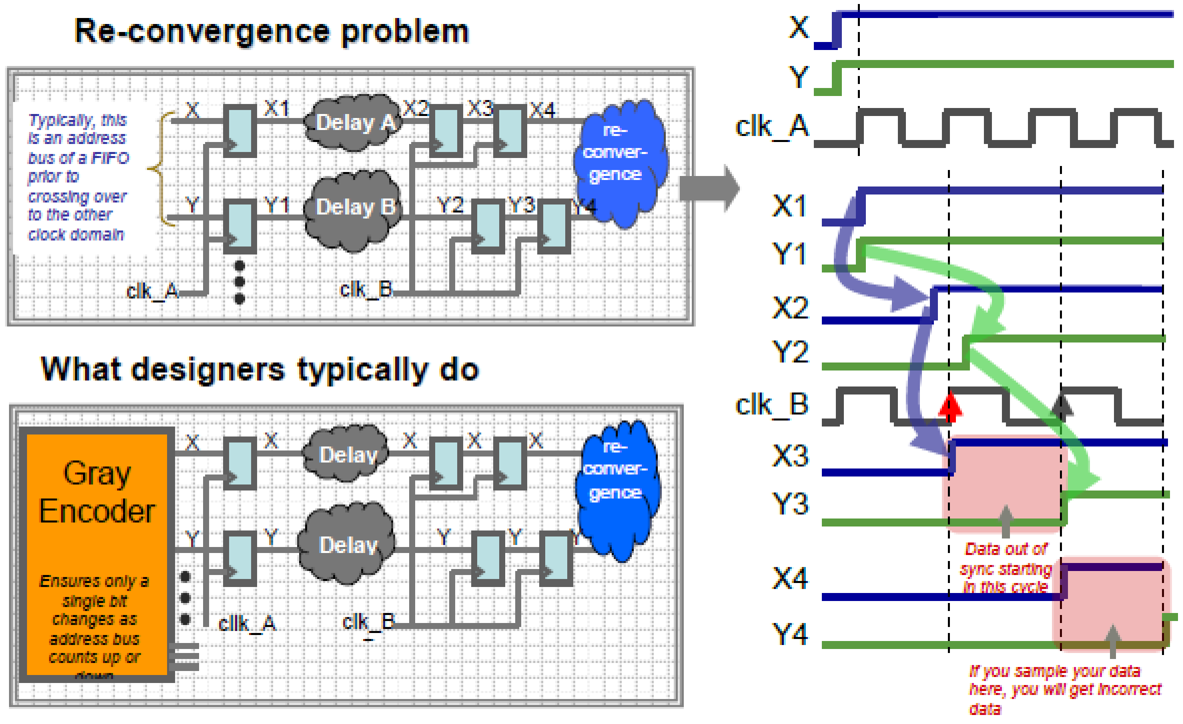

Re-convergence problem
Typically, this is an address bus of a FIFO prior to crossing over to the other clock domain
X
X1
Delay A
X2
X3
X4
Y
Y1
Delay B
Y2
Y3
Y4
re-conver-gence
clk_A
clk_B
What designers typically do
Gray Encoder
Ensures only a single bit changes as address bus counts up or down
Delay
clk_A
clk_B
X
Y
clk_A
X1
Y1
X2
Y2
clk_B
X3
Y3
X4
Y4
Data out of sync starting in this cycle
If you sample your data here, you will get incorrect data

FIGURE 7. The re-convergence problem and a typical solution using gray coding

To fix this problem, introduce a gray encoder, which ensures that only a single bit is changed at a time.

You must ensure that *Correlated Signals* are gray encoded before they cross clock domains. You can identify such signals where independent signals are converging and are used in the same combinational logic, or when a bus is used as a state vector or a memory pointer.

Complex Synchronizers

FIFO mechanisms are often used to transfer data from one domain to another.

The following figure shows a FIFO synchronizer architecture:

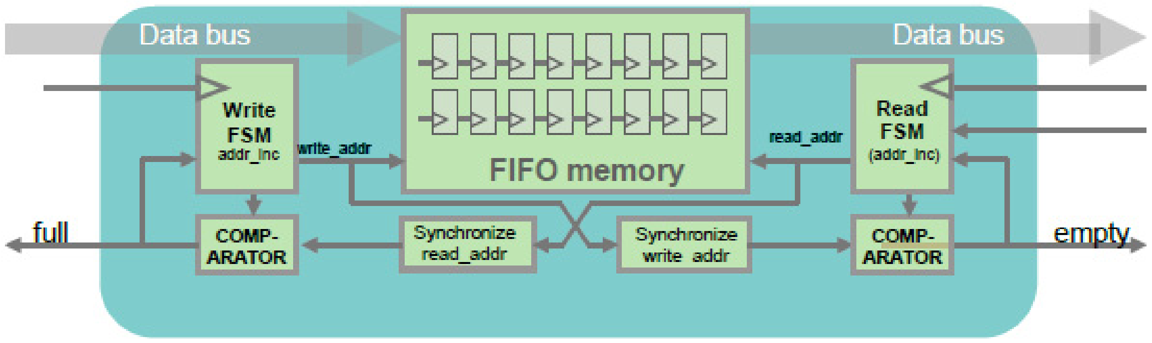


FIGURE 8. FIFO synchronization scheme

For proper data transfer, it is important that the full and empty flags are generated on time and are not delayed or corrupted due to the pointers crossing clock domains. It is also important that the read and write FSMs make use of the full and empty flags to prevent writing into a full FIFO or reading from an empty FIFO.

Issues Related to Reset Synchronization

Reset synchronizers are especially built to avoid metastability while de-asserting a reset signal. Such synchronizers must be verified for both metastability and functionality to avoid reset failures.

The following figure shows the example of reset synchronization:

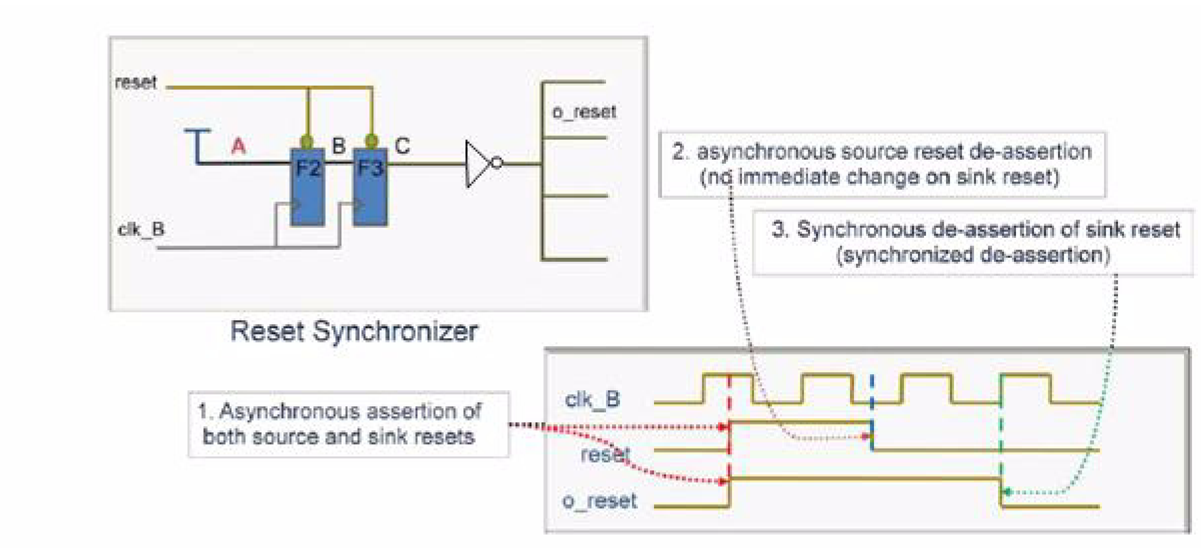

reset
A
B
C
F2
F3
clk_B
o_reset
Reset Synchronizer
2. asynchronous source reset de-assertion (no immediate change on sink reset)
3. Synchronous de-assertion of sink reset (synchronized de-assertion)
1. Asynchronous assertion of both source and sink resets
clk_B
reset
o_reset

FIGURE 9. Reset Synchronization

Using SpyGlass CDC Methodology to Solve CDC Problems

This section provides a step-by-step solution to make an SoC free from CDC issues by using any of the following flows:

- *SpyGlass CDC Methodology Flow*
- *SpyGlass CDC Hierarchical Verification Flow*

SpyGlass CDC Methodology Flow

The following figure illustrates this flow to achieve a SpyGlass CDC clean SoC:

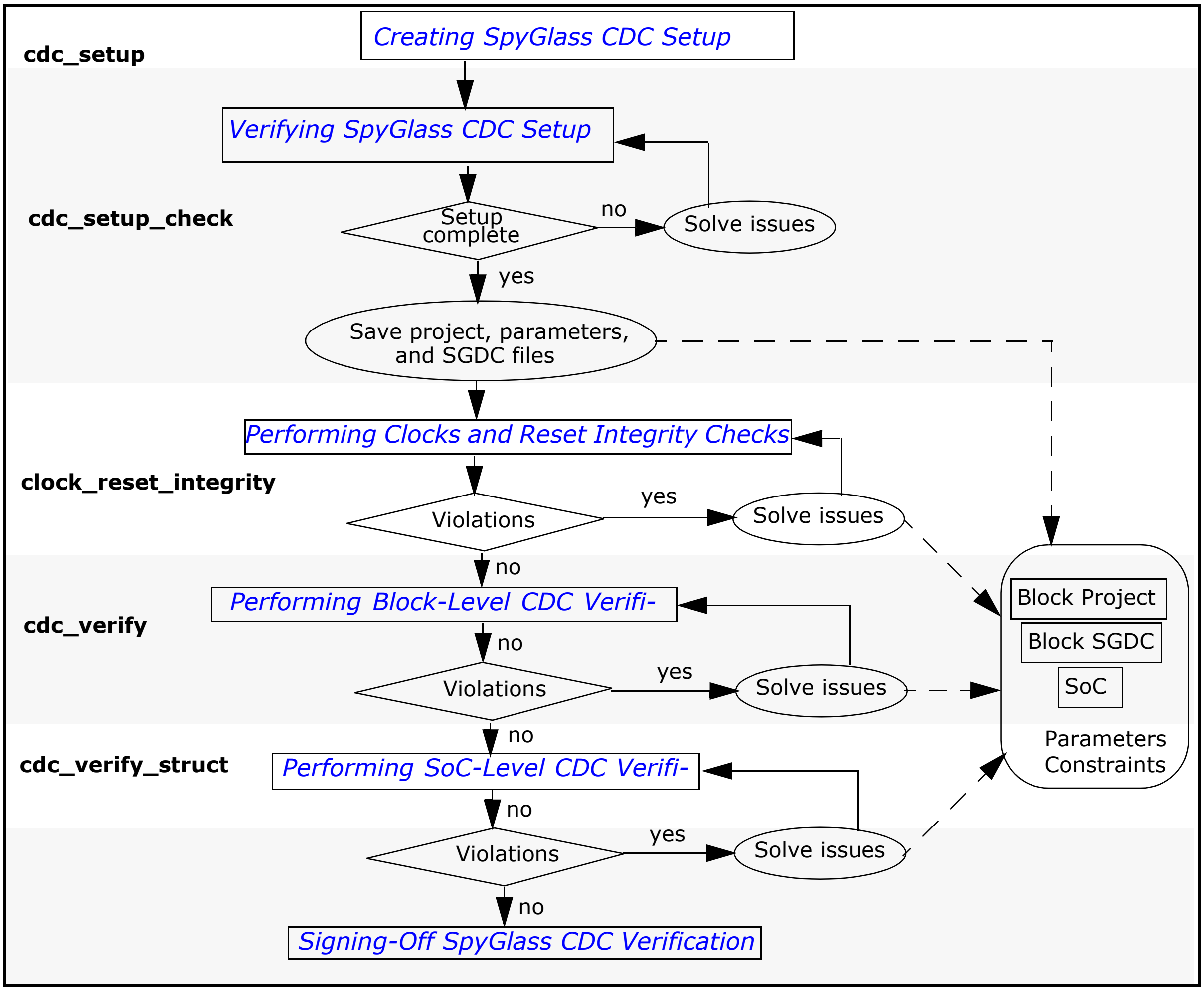

cdc_setup
Creating SpyGlass CDC Setup
Verifying SpyGlass CDC Setup
cdc_setup_check
Setup complete
no
Solve issues
yes
Save project, parameters, and SGDC files
Performing Clocks and Reset Integrity Checks
clock_reset_integrity
Violations
yes
Solve issues
no
Performing Block-Level CDC Verifi-
cdc_verify
no
Violations
yes
Solve issues
no
cdc_verify_struct
Performing SoC-Level CDC Verifi-
no
Violations
yes
Solve issues
no
Signing-Off SpyGlass CDC Verification
Block Project
Block SGDC
SoC
Parameters
Constraints

FIGURE 1. The SpyGlass CDC Methodology Flow

The following table shows the stages and their corresponding goals to achieve a SpyGlass CDC-clean design while using this flow:

Stage	**Summary**	**Goals**
Creating SpyGlass CDC Setup	Specify constraints, parameters, and other design components required for accurate and complete SpyGlass CDC verification.	cdc_setup
Verifying SpyGlass CDC Setup	Check for the correctness and completeness of the setup.	cdc_setup_check
Performing Clocks and Reset Integrity Checks	Fix clock and reset integrity problems	clock_reset_integrity
Performing Block-Level CDC Verification	Fix block-level violations to make the block SpyGlass CDC clean. This stage involves the following tasks: ● *Fixing Ac_sync_group Rule Violations* ● *Fixing Violations Related to Convergence* ● *Fixing Violations Related to Glitches* ● *Fixing Violations for Data Hold Checks* ● *Fixing Data Hold Issues in Synchronized Data Crossings* ● *Fixing Violations Related to Data Correlation and Race Conditions* ● *Fixing Violations Related to Reset Synchronization and Deassertion* ● *Fixing Violations Related to FIFO Recognition and Verification*	cdc_verify
Performing SoC-Level CDC Verification	Verify the SoC using the verified blocks	cdc_verify_struct
Signing-Off SpyGlass CDC Verification	View reports and sign-off SpyGlass CDC Verification on the SoC	-

Creating SpyGlass CDC Setup

In this stage, you specify constraints, parameters, and other design components required for accurate and complete SpyGlass CDC verification.

Create a setup in the following ways:

- *Writing Constraints*

- *Translating SDC Commands to SGDC Commands*
- *Predicting Constraints*
- *Running the cdc_setup Goal*
- *Generating Block-Level Constraints from SoC Level*
- *Generating Clocks*

NOTE: *You can also perform a setup by using The Setup Manager of SpyGlass CDC.*

Writing Constraints

Define constraints in an SGDC file if you have knowledge of block constraints.

Translating SDC Commands to SGDC Commands

Use the `sdc2sgdc` project file command to translate block-level SDC commands to their corresponding SGDC constraints.

Predicting Constraints

Run the *cdc_setup_check* goal to generate constraints. The *Clock_info15* rule of this goal generates constraints.

You must review these constraints before using them during SpyGlass CDC verification.

Running the cdc_setup Goal

Run the *cdc_setup* goal to generate clocks (*clock* constraint) and resets (*reset* constraint) in a design.

After running this goal:

- Understand the design-clocks architecture by checking the reported clocks and resets.
- Resolve the clocks of black boxes by specifying a path through the black boxes by using the *assume_path* constraint.

- Copy the *autoclocks.sgdc* and *autoresets.sgdc* in a new constraint file and edit the clocks and resets to provide valid clock/reset sources.

  Remove all the non clocks and non resets from the file.
- Provide the *set_case_analysis* constraint to add the known case-analysis values under which you want to perform SpyGlass CDC analysis.

Generating Block-Level Constraints from SoC Level

You generate block-level constraints from the SoC-level constraints in the *SpyGlass CDC Hierarchical Verification Flow*.

Run the *cdc_top_down* goal to generate block-level constraints.

Note that using the generated block-level constraints for *Generating Abstract View in SpyGlass CDC* without *Performing Block-Level CDC Verification* may mask design bugs. For example, if a top module propagates to the `P` pin (of the `A` domain) of a block, you must verify that the `P` pin is feeding flip-flops in the `A` domain within the block or it is synchronized to another domain in which the pin is used.

SpyGlass CDC solution can also generate a block's peripheral domain information from within a block relying on the flip-flops interacting with these pins. For details, refer to the documentation of the rule Clock_info15 in SpyGlass CDC Rules Reference Guide.

Generating Clocks

You can specify derived clocks or generated clock by using the `generated_clock` constraint. These are the clocks that traverse from the output (hierarchical pin or net) of sequential elements.

To enable SpyGlass consider this constraint, set the `enable_generated_clocks` parameter to `yes`. When you set this parameter to `yes`, the following occurs:

- The specified `generated_clock` constraints are considered during SpyGlass analysis.
- The derived clock information is generated in the form of `generated_clock` constraints in the *generated_clocks.sgdc* and *cdc_setup_generated_clocks.sgdc* files.

This happens when the `use_inferred_clocks` parameter is set to `yes`.

Verifying SpyGlass CDC Setup

After *Creating SpyGlass CDC Setup*, run the *cdc_setup_check* goal to correctness and completeness of the setup.

Fix all violations reported in this stage to avoid false violations in later stages. For example, you must ensure the following:

- All flip-flops are receiving a clock.
- *set_case_analysis* is properly defined so that multiple clocks do not control the same flip-flop. See *Specifying set_case_analysis*.
- Multiple clocks are not defined on the same clock path.
- Periods, edges, and domains are defined properly for clocks.

Constraining Clock Trees

Constraint clock tree by:

- *Constraining Clock Nets*
- *Specifying set_case_analysis*

Constraining Clock Nets

Check the *Clock_info03a* violations to locate the clock-tree parts to which top-level clocks do not reach. This occurs because of:

- Missing *clock* constraints in an SGDC file.
- Presence of black boxes through which a clock cannot propagate.

  Black boxes appear because their structural information is missing or they have incorrect case analysis settings.

Specifying set_case_analysis

MUXes in clock trees use different clocks for different operating modes of the design. Configure MUXes by setting an operating mode by applying the *set_case_analysis* constraint on the MUX select pin.

Consequences of Not Configuring MUXes

Consider the scenario in the following figure:

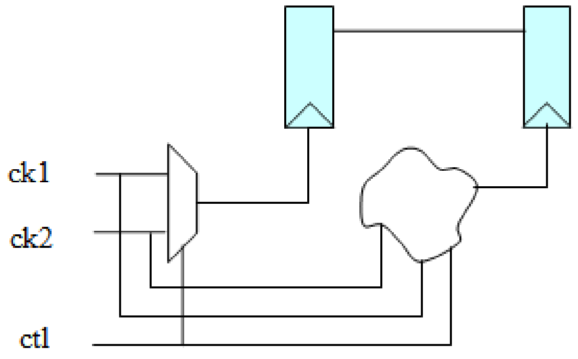


FIGURE 2. Configuring the MUX

In the above scenario, if you do not configure the MUX by applying *set_case_analysis* on its select pin, multiple clocks may drive the same flip-flop. As a result, SpyGlass may infer the path between these flip-flops as asynchronous crossings even if these paths are synchronous. This results in false unsynchronized violations, which results in noise and more time for CDC verification closure.

Fixing Violations for Non Configured MUXes

Check for the *Clock_info05* violations that report cases where you should define *set_case_analysis* on the MUX select pin.

Refer to the *Clock Setup* window that shows MUXes involved in clock paths from where you can interactively define a value for MUX selects.

Fixing Setup-Related Sanity Checks

The setup verification performs basic sanity checks on the constraints specified in an SGDC file. These checks are always run to check for design-object existence and constraints correctness.

You can identify the violations of these checks with the *SGDC_* prefix in their names.

For information on these checks, refer to SpyGlass CDC Rules Reference Guide.

Performing Clocks and Reset Integrity Checks

Run the *clock_reset_integrity* goal to fix clock and reset integrity problems.

This step ensures that clocks and resets are properly defined, and they are free of glitches, race conditions, and other hazards.

If you do not have the information about clocks and resets, you must run the setup. For details, see *Creating SpyGlass CDC Setup*.

Performing Block-Level CDC Verification

Run the *cdc_verify* goal to perform SpyGlass CDC verification at the block level.

This step uses all the information gathered while *Creating SpyGlass CDC Setup* and *Performing Clocks and Reset Integrity Checks* to perform SpyGlass CDC verification at block level.

This section covers the following topics:

- *Focusing on Certain Violations on Priority Basis*
- *Reducing Noise*
- *Dealing with Functional Checks*
- *Waiving Violations*

Focusing on Certain Violations on Priority Basis

You may initially see a large number of reported CDC issues. It is important to approach them in a systematic way. This enables you to quickly reach to a handful of issues that you may need to consider.

The issues listed in the following sections cover the majority of important violations you should fix on priority:

- *Fixing Ac_sync_group Rule Violations*
- *Fixing Violations Related to Convergence*
- *Fixing Violations Related to Glitches*
- *Fixing Violations for Data Hold Checks*
- *Fixing Data Hold Issues in Synchronized Data Crossings*
- *Fixing Violations Related to Data Correlation and Race Conditions*
- *Fixing Violations Related to Reset Synchronization and Deassertion*
- *Fixing Violations Related to FIFO Recognition and Verification*

Fixing Ac_sync_group Rule Violations

This group of rules performs an architectural design analysis and presents an architectural view of design crossings.

You must fix the following violations of this group first before fixing the other violations:

- *Ac_unsync01* rule violations: These violations report scalar unsynchronized crossings. Such crossings act as control signals that synchronize complex data crossings.
- *Ac_unsync02* rule violations: These violations report data crossings where no valid synchronization is found.

Run SpyGlass again with newly added constraints and verify that all *Ac_unsync01* and *Ac_unsync02* violations are fixed.

Debugging Ac_unsync Violations

Use the following pointers to debug and fix such violations:

- Open the spreadsheet to view all the violations of a rule. Look for common reasons or common sources in the spreadsheet.

  Use filtering and sorting in the spreadsheet view to isolate common factors between violations. If you are using a naming methodology for static signals, filter by source name in the spreadsheet.
- Open the *Incremental Schematic* to view the cause for unsynchronized crossings. Check for the reason for such crossings in the violation message or spreadsheet.
- Check for the presence of qualifiers or potential qualifiers for a crossing.

- Filter violations by using the `cdc_false_path` constraint.
- Specify output net names for source and destination flip-flops.
- Check for mode or control-status registers that are static or quasi-static.
- Do not waive such violations. Use the *`cdc_false_path`* constraint instead to filter certain unsynchronized crossings in a design.

  Note that the violations of the other rules that honor this constraint may get filtered due to this constraint specification.

False Ac_unsync Violations

If the *clock* and *set_case_analysis* constraints are not properly defined during setup, you may see false *Ac_unsync01* and *Ac_unsync02* violations. For information on specifying these constraints during setup, see *Constraining Clock Trees*.

Such violations also appear due to configuration registers and other quasi-static signals need not to be synchronized. Section Noise Reduction describes various tools SpyGlass provides to reduce false violations and find real synchronization bugs faster.

Conditions for an Unsynchronized Crossing

Consider the following figure:

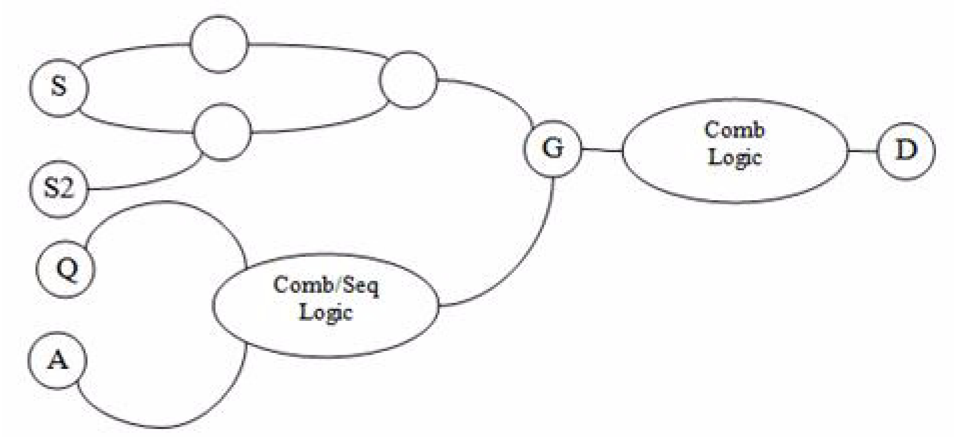


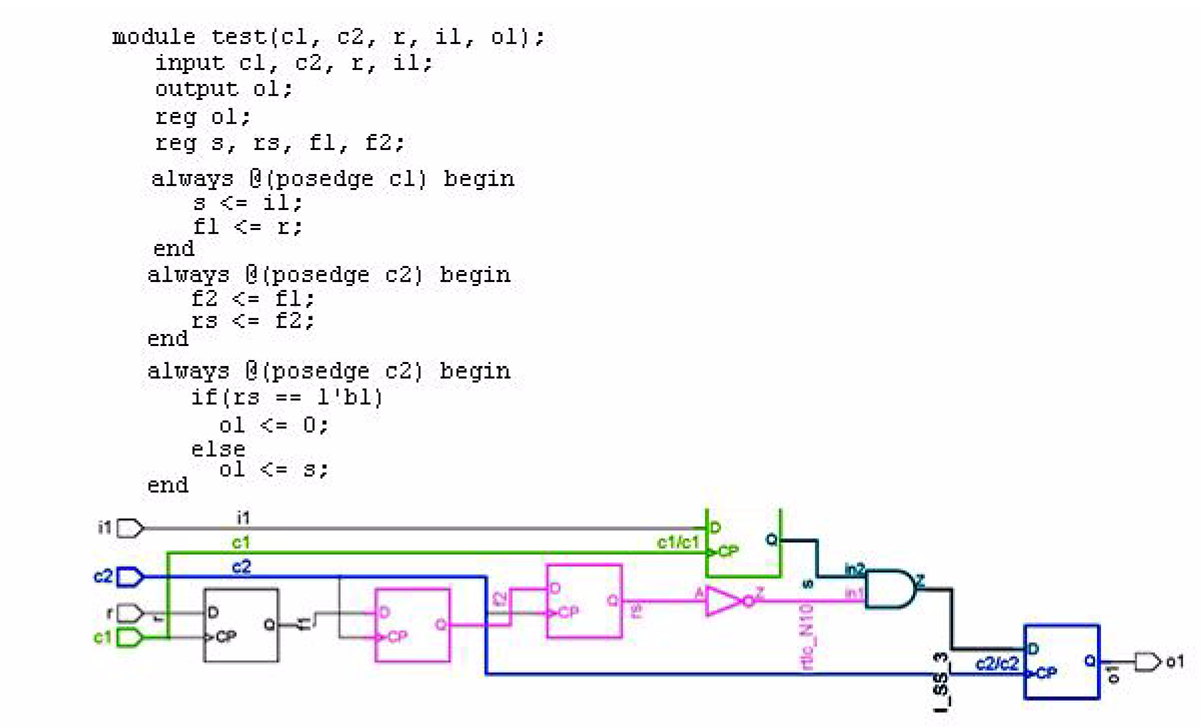
FIGURE 3. Requirements for a synchronized crossing

In the above figure, a crossing is considered as unsynchronized if one of the following conditions is false for the source S:

- The `Q` qualifier whose source domain is the same as that of S exists and converges with S on the `G` gate.
- The type of the G gate is consistent with the type specified by the `enable_and_sync`, `enable_mux_sync`, and `enable_clock_gate_sync` parameters.
- If S fans out to multiple gates, all the fan-out points must converge before G. In *Figure 3*, the two divergent paths from S converge before G, so this condition is met.
- The input of the synchronizing gate G that is driven by Q is not driven by another source of the crossing. In *Figure 3*, `A` should not be the source for the crossing. Flip-flops in the domain of the destination are allowed.
- If another source `S2` converges with `S` before `G`, `S2` must be in the same domain as that of `S`.
- The path from `Q` to `G` is considered based on the value of the `enable_delayed_qualifier` parameter.

Declaring Synchronous Resets

Consider the following figure:



```
module test(c1, c2, r, i1, o1);
   input c1, c2, r, i1;
   output o1;
   reg o1;
   reg s, rs, f1, f2;
   always @(posedge c1) begin
      s <= i1;
      f1 <= r;
   end
  always @(posedge c2) begin
      f2 <= f1;
      rs <= f2;
  end
  always @(posedge c2) begin
      if(rs == 1'b1)
        o1 <= 0;
      else
        o1 <= s;
  end
```

FIGURE 4. Example of a crossing with a reset synchronizer

In the above figure, the source `s` is reported as synchronized with the qualifier `rs`.

By looking at the Verilog description, it is clear that `rs` is the output of a reset synchronizer. Since the user missed to declare the input `r` as a reset, `rs` is considered as a qualifier.

To avoid such issues, declare all synchronous resets in the SGDC file to avoid considering their reset synchronizers as qualifier. Therefore, specify the following constraint to fix the issue in the above example:

```
reset -name r -sync
```

Fixing Violations Related to Convergence

Check for the *Ac_conv01*, *Ac_conv02*, *Ac_conv03*, *Ac_conv04*, and

Ac_conv05 violations.

Convergence issues can occur when multiple signals cross from one domain to another but they are separately synchronized.

Fixing Violations Related to Glitches

Check for the *Ac_glitch** or *Clock_glitch** violations.

These rules report glitch-prone logic that can lead to problems similar to synchronization issues.

Fixing Violations for Data Hold Checks

Check for the *Ac_cdc01* violations.

Such violations indicate potential problems in signals or data crossing typically from a fast clock domain to a slower clock domain where data sent may have already changed by the time the capturing clock arrives.

The *Dealing with Functional Checks* section provides further detail on how to debug such functional checks.

Fixing Data Hold Issues in Synchronized Data Crossings

Check for the *Ac_datahold01a* violations.

Such violations report clock domain crossings where data can be unstable while the enable is active. For every data change, the enable should be activated to capture the new data and should be deactivated before the next data is loaded.

The *Dealing with Functional Checks* section provides further detail on how to debug such functional checks.

Fixing Violations Related to Data Correlation and Race Conditions

Check for gray-code violations, such as *Ac_cdc08*, *Ac_conv01*, and *Ac_conv02*.

Convergence of signals, such as control buses can cause major problems if they are not implemented using approved methods.

Typically, with control buses crossing clock domains, designers implement gray code schemes to handle such issues. Using a gray-encoded implementation for control bus signals ensures that only one bit of the control signal changes during any one clock cycle.

For debug and analysis of the gray encoding check and other functional checks, see *Dealing with Functional Checks*.

Fixing Violations Related to Reset Synchronization and Deassertion

Check for the following rule violations:

Rule	Violation Reported
Ar_unsync01	Reports unsynchronized reset signals in the design
Ar_sync01	Reports synchronized reset signals in the design
Ar_asyncdeassert01	Reports if reset signal is asynchronously de-asserted
Ar_syncdeassert01	Reports if reset signal is synchronously de-asserted or not de-asserted at all
Reset_sync02	Reports asynchronous reset signals that are generated in asynchronous clock domain

Fixing Violations Related to FIFO Recognition and Verification

Check for the Ac_fifo01 and Ac_sync_group rule violations.

SpyGlass can automatically identify FIFOs. FIFO recognition may produce following results:

- Fully recognized FIFOs: This is the case if memory and pointers of a FIFO are identified
- Partially recognized FIFOs/Memory: A 2-dimensional memory or a lib/sglib memory identified by SpyGlass for which read/write pointers were not identified.
- Disabled: When fa_msgmode is set to none.

FIFO recognition will help SpyGlass CDC verification as follows:

- Metastability violations reduction (Ac_unsync02 violations reduction): Typically, a FIFO memory is clocked by write clock and the data is read out of memory in a read domain. This situation creates a clock domain crossing from write domain to the read domain that will potentially be reported as unsynchronized (Ac_unsync02 violation). FIFO recognition will help in reducing such metastability violations (the crossing will be reported as properly synchronized by Ac_sync02 rule). You can control FIFO based Ac_unsync02 filtering with enable_fifo option. If the option is set to “strict”, only fully recognized FIFOs will contribute to Ac_unsync02 violations reduction. If enable_fifo is set to “soft”, partially recognized FIFOs/Memory will also lead to Ac_unsync02 violations reduction. Reading data out of a memory is not necessarily safe and may be subject to metastability; so usage of enable_fifo set to soft is not advised unless you are sure that the control logic around the memories provide sufficient margin between the data being written into the memory and the read request out of the memory. List of FIFOs recognized in a design is given by Rule Ac_fifo01.

- Functional verification of FIFOs: For all fully recognized FIFOs, SpyGlass performs functional check to make sure the FIFO will not overflow or underflow. FIFO overflow/underflow violations are reported in Ac_fifo01 rule.

SpyGlass recognizes commonly used FIFO architectures where memory and pointer counters can be identified. FIFOs cannot be extracted from a netlist design as the counters are mapped into gate level netlist. SpyGlass provides “fifo” constraint that can be used to provide FIFO attributes that would help FIFO recognition and verification. The fifo constraint can be used to provide FIFO attributes, such as memory and/or pointers in a constraint file (SGDC). Here is an example of “fifo” constraint:

```
fifo -memory "uart_top.u13.u4"
```

For debug and closure of FIFO and other functional checks, see *Dealing with Functional Checks*.

Reducing Noise

You can reduce noise by:

- *Setting Parameters*
- *Setting Constraints*
- *Filtering Violations in a Spreadsheet*

Setting Parameters

For a particular design or project, set the following parameters to reduce the number of violations:

- `allow_combo_logic`

  Use this parameter to allow combinational logic between synchronizers.

Combinational logic on a crossing can create a glitch. It is harmless in a synchronous circuit. However, its presence in an asynchronous crossing may cause unwanted pulses causing functional failures.

- `cdc_reduce_pessimism`

  Use this parameter to filter out violations by setting this parameter to appropriate values.

- `clock_reduce_pessimism`

  Use this parameter to control clock-domain propagation and consequently control SpyGlass CDC solution violations.

Setting Constraints

Specify the following constraints to reduce noise:

- `cdc_false_path`

  Specify this constraint to filter certain unsynchronized crossings in a design. An example of such crossings is configuration and other quasi-static registers that do not need synchronizers.

  Using this constraint, you can specify the paths that the `Ac_sync_group` rules should not check for clock crossings. This reduces the number of violations reported on that path. The following is an example of `cdc_false_path`:

```
cdc_false_path -from block1.flop1 -to block2.flop2
cdc_false_path -from block1.clk1
cdc_false_path -from config_module::fifo_config_reg[1]
```

  The first line filters out the `flop1`-to-`flop2` crossing from `Ac_sync_group` violations. The second constraint eliminates all the crossings from flip-flops controlled by `clk1` regardless of their destination flip-flops.

- `reset -sync`

  If you are using a synchronous reset at the crossing or synchronizer flip-flops, you can specify these resets using the reset constraint with sync argument (reset –sync). This allows combinational gates generated due to synchronous reset logic in the crossing or synchronizer path.

  By default, synchronous reset gate will be considered as combinational logic and a crossing will be considered unsynchronized.

Filtering Violations in a Spreadsheet

From `Ac_unsync01` and `Ac_unsync02` violations header you can access a spreadsheet view of all violations. In this spreadsheet, you can sort or filter violations based on several criteria (e.g. source or destination clocks, reason of failures, etc.). Explore the violations in the spreadsheet to determine false violations due to configuration registers, unconstrained paths, etc. You can select all such violations and request `cdc_false_path` constraint generation from the spreadsheet window; `cdc_false_path` constraints will prevent these violations from being reported in subsequent runs.

Dealing with Functional Checks

Functional verification of clock-domain crossings is an important aspect of SpyGlass CDC verification. Many critical bugs causing SoC spins are because of gray-encoding failure, FIFO failure, and other types of functional problems in clock-domain crossings.

Functional checks are more CPU-intensive than structural checks.

Focusing on Failed or Partially-Proved Checks

A functional check reports any of the following status:

Status	**Description**
FAILED	Refers to functional checks that failed. For such cases, SpyGlass provides a simulation trace that you can view in the waveform viewer. To open the waveform viewer, double-click the violation and click the waveform viewer icon.

PASSED	Refers to checks that passed. SpyGlass reports a message for such checks only if fa_msgmode is set to pass or all. These checks are reported with the INFO severity. This status indicates a proper functionality proof of SpyGlass CDC solution.
PP (Partially Proved)	Refers to checks that could not be concluded. SpyGlass provides the number of cycles that have been explored during which no violation has been found. Similar to passed checks, these checks are reported only if fa_msgmode contains "pp" or "all"; by default both failed and partially proved results are reported. These checks are reported with the WARNING severity. See *Dealing with Partially-Proved Checks*.

Focus on failed and partially-proved checks as they may represent real design bugs.

Dealing with Partially-Proved Checks

Set the `fa_atime` parameter to increase the amount of time that SpyGlass spends on validating a single property.

Dealing with Long Run Times

Formal verification is exhaustive and involves complex functional analysis of a design. The complexity of functional analysis increase with the number of asynchronous clocks in a design.

It is recommended to perform functional verification only where it is required. Avoiding unnecessary functional verification requires *Creating SpyGlass CDC Setup* properly and *Reducing Noise*.

You can deal with long run times in the following ways:

- *Constraining Resets*
- *Dealing With Clock Frequencies*

Constraining Resets

Consider a synchronous reset always converges with a data/control signal through a simple gate, such as an AND gate. This type of convergence, although reported by *Ac_conv01* and *Ac_conv02*, can be considered as safe as long as the reset is static.

In this case, you should constraint the synchronous reset by using the `reset -sync` constraint. By doing so, you can reduce the number of *Ac_conv01* and *Ac_conv02* violations reported because of synchronous reset convergence. This consequently reduces the run time by preventing formal verification of such convergences.

Dealing With Clock Frequencies

Clock frequencies may greatly affect the complexity of functional analysis.

To understand how clock frequencies affect the functional analysis process, consider two clocks running with the 17 ns period and 13 ns period, respectively.

If the rising edges of the two clocks are aligned at the time 0 ns, the next time the rising edges will again be aligned corresponds to 221 ns (the LCM of two clock periods). This means that the design behaves asynchronously for 221 ns.

Any functional analysis that repeats itself many times (for proving a property, for example) analyzes the design for at least this period of time. This means it performs many evaluations of logic in the design. This time period is called the *Design Virtual Cycle*. A high design virtual cycle makes it hard to verify design functionality.

In some cases, if functional analysis enter into long design runs, modify clock periods to reduce the LCM. Consider the following example.

The device A has two asynchronous clocks: clk_33 (clock period - 33 ns) and clk_100 (clock period - 100 ns). If you specify these clock periods in an SGDC file, the LCM of the two clock periods is 3300 ns (33x100), which is quite large.

If you specify the 100 ns clock in the SGDC as the 99 ns clock, the design virtual cycle reduces to 99 ns. Note that changing the clock frequency by this amount affects the behavior of the design, and therefore the change should not be considered unless necessary.

SpyGlass reports the design virtual cycle in terms of the number of fastest clock cycles and the number of non-overlapping edges of all clocks covered by the design virtual cycle.

Note that the gray-encoding check is a relatively local check as the logic for gray encoding is purely combinatorial and should not depend on the frequency. In this case, frequency numbers are not important. If frequency/period information is not provided, then SpyGlass assumes all clocks (clocks for which a period is not defined) as having a 10 ns period.

Debugging Functional Checks

A failed functional check generates a waveform indicating the circumstances of the failure.

To view the waveform viewer, double-click on the violation and click the waveform viewer icon.

Initially, a small set of signals are loaded in the waveform viewer. These signals are a good starting point for debugging. To check the signals in the vicinity of a signal, right-click on that signal and select the *fan-in* option from the shortcut menu. Select all or part of these signals and click *OK* to load their waveform in the viewer.

Note that you can cross-probe between the waveform viewer and the RTL-viewer.

Removing False Violations of Functional Checks

False violations appear if the design is not constrained properly or SpyGlass considers an inappropriate initial state of the design.

Constraining the Design Properly

Reset signals are used to initialize the design and they are usually disabled during functional checks. For example, a gray-encoding check may fail due to a reset signal being asserted in the middle of a binary count.

To prevent functional checks failure due to reset toggling, define the reset signal by using the `reset` constraint in an SGDC file.

If you want the reset to be considered as any other input during function check, declare the reset as soft by specifying the `-soft` argument with the `reset` constraint.

Specifying the Correct Initial State of the Design

Validate the *Ac_initstate01* rule message to know the initial state used during functional verification.

Functional checks may fail or pass depending on the initial states considered by SpyGlass.

Waiving Violations

It is recommended to use the `cdc_false_path` constraint to reduce the number of false violations.

However, if you want to remove a specific violation that does not have any global impact of discarding a path, waive that violation. For example, you may waive a *Clock_info03a* violation.

You can waive violations before or after SpyGlass analysis, as described below:

- Before analysis, specify the `waive` constraint to waive violations on a block that you do not want to analyze.
- After analysis, waive a violation that are safe to be ignored.

NOTE: *Apply waivers to only those rules that do not directly involve a synchronizer.*

Performing SoC-Level CDC Verification

Run the *cdc_verify_struct* goal to perform SpyGlass CDC verification on the SoC.

If you are using the *SpyGlass CDC Hierarchical Verification Flow*, specify the SGDC files representing the abstract views of blocks with the SoC-level files while performing SoC-level verification.

This step verifies all structural issues in SpyGlass CDC solution on the SoC.

All violations, including those from the rules *Ac_unsync01*, *Ac_unsync02*, *Ac_conv01*, *Ac_conv02*, and *Ac_conv03* should be analyzed and resolved.

Signing-Off SpyGlass CDC Verification

- Open SpyGlass CDC report from the GUI pull-down menu, **Report->clock-reset->CDC-report**, and review the content as follow

- Examine the assumptions; the SpyGlass CDC report header contains all parameters that make the verification optimistic (e.g. use of `allow_combo_logic`). All optimistic assumptions need to be justified and documented.
- Check if all verification goals have been run and if there are any violations left unsolved. All such violations need to be justified and documented.

SpyGlass CDC Hierarchical Verification Flow

Unlike the *SpyGlass CDC Methodology Flow*, in this flow you use abstract views of blocks while *Performing SoC-Level CDC Verification*. Using abstract views reduce SpyGlass CDC verification run time by focusing on SpyGlass CDC solution issues on block boundaries only.

Use this flow in the following cases:

- **Large SoCs**

  Performing SpyGlass CDC verification on large SoCs having 100M+ gates and many clocks can be time consuming. For such designs, use the SpyGlass CDC hierarchical verification flow for faster SpyGlass CDC verification sign-off.

- **Distributed Environment for SoC Development**

  In such environment, IPs are developed or acquired from different design teams and SoC integration happens in a different location. In such cases:

  - Block owners verify blocks and handoff the abstract views of these blocks (along with the blocks) to the SoC integration team.
  - SoC integration team uses abstract views without worrying about the block content.

    If the abstract view of some blocks is not available, the SoC team does the following:

    - Generate the abstract views for such blocks.
    - Migrate constraints from top level to block level. For details, see *Generating Block-Level Constraints from SoC Level*.
    - Consider such blocks as glue logic by specifying them with the `ip_block` constraint.

The following figure illustrates this flow:

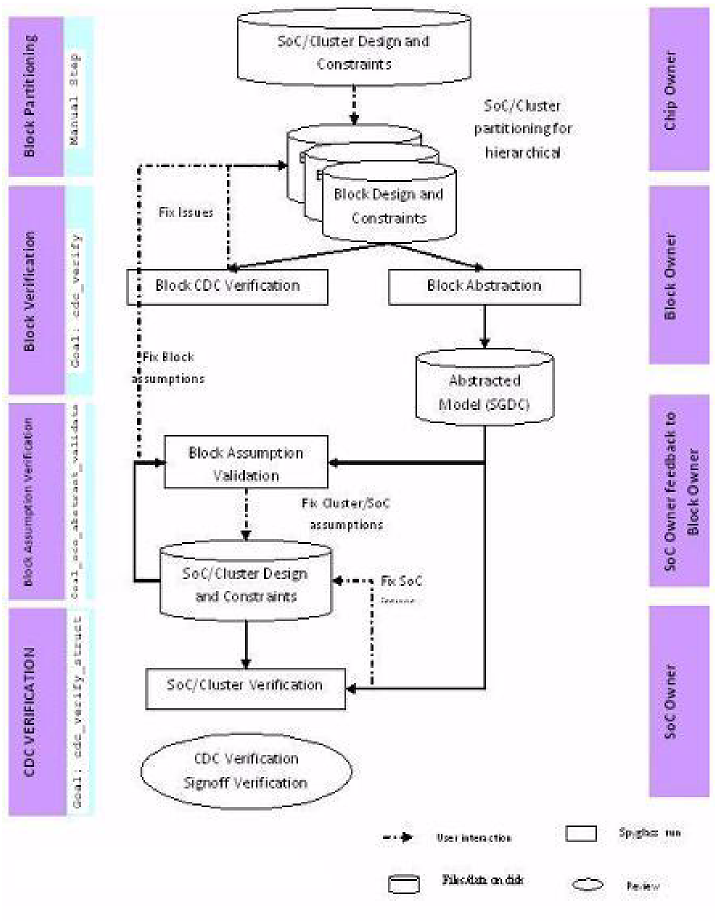

Block Partitioning
Manual Step
Block Verification
Goal: cdc_verify
Block assumption verification
Goal: soc_abstract_validate
CDC VERIFICATION
Goal: cdc_verify_struct
SoC/Cluster Design and Constraints
SoC/Cluster partitioning for hierarchical
Block Design and Constraints
Fix Issues
Block CDC Verification
Block Abstraction
Fix Block assumptions
Abstracted Model (SGDC)
Block Assumption Validation
Fix Cluster/SoC assumptions
SoC/Cluster Design and Constraints
Fix SoC
SoC/Cluster Verification
CDC Verification Signoff Verification
User interaction
Spyglass run
Files/data on disk
Review
Chip Owner
Block Owner
SoC Owner feedback to Block Owner
SoC Owner

FIGURE 5. Abstract Bottom Up SoC level CDC verification flow

The following table shows the steps and their corresponding goals used in this flow:

TABLE 1 Steps in SpyGlass CDC Hierarchical Verification Flow

Steps	**Summary**	**Goals**
Identifying the Blocks to Abstract in SpyGlass CDC	Identify the blocks whose abstract view should be created. This abstract view is used while *Performing SoC-Level CDC Verification*.	-
Creating SpyGlass CDC Setup	Capture block constraints, such as clocks, input domains, resets, and other assumptions on the inputs	cdc_setup_check
Verifying SpyGlass CDC Setup	Check for the correctness and completeness of the setup.	cdc_setup_check
Performing Clocks and Reset Integrity Checks	Fix clock and reset integrity problems	clock_reset_integrity
Performing Block-Level CDC Verification	Verify all the sub blocks. The input constraints captured while *Creating SpyGlass CDC Setup* dictate the quality of the block verification. If an input is in a given domain then it should feed the flip-flops in the same domain or be synchronized before being used in a different domain. However, on the output side, constraints, such as domains may be neglected, as those will be identified during verification and generation of abstract model.	cdc_verif
Generating Abstract View in SpyGlass CDC	Create an abstract view for a block	cdc_abstract
Performing Abstract View Validation in SpyGlass CDC	Validate block assumptions against the higher-level hierarchy constraints	cdc_abstract_validate

TABLE 1 Steps in SpyGlass CDC Hierarchical Verification Flow

Steps	**Summary**	**Goals**
Performing SoC-Level CDC Verification	Verify the SoC using the abstract view of blocks	cdc_verif_struct
Signing-Off SpyGlass CDC Verification	View reports and sign-off SpyGlass CDC Verification on the SoC	-

Identifying the Blocks to Abstract in SpyGlass CDC

Decide the blocks to verify before moving to the verification of a higher-level hierarchy.

Typically, for a full SoC, verifying the first level instances (often referred to as a clusters, or sub-modules) before moving to the SoC verification is good enough. However, if the size and complexity of a sub-module is so that the verification may take long time, 5M+ gates with dozens of asynchronous clocks, then it is a good idea to further partition the sub-module for verification before verifying the SoC.

Note that it is important to consider single clock blocks while verifying SpyGlass CDC solution of a design instantiating the block. For example, if an input of a single clock block is coming from another clock domain, then the block must synchronize the input before using it. Furthermore, if a multi-flop synchronizer feeds into a single clock module, it may converge with other multi-flop synchronizers within the module or further down after exiting the block.

Generating Abstract View in SpyGlass CDC

Run the *cdc_abstract* goal to generate the abstract view of a block. This goal runs the *Ac_abstract01* rule that generates the abstract view of a block.

The abstract view is an SGDC file that is used by the SoC owner while *Performing SoC-Level CDC Verification*. The abstract view captures all the block-level constraints. It also propagates synchronizer information (crossing information) to IOs by capturing this information in `abstract_port` constraints.

NOTE: *Abstraction is performed on a top-level module, hence set_option top <block-name> must be specified during block level verification run.*

Quality of Abstract View

The quality of an abstract view depends on the quality of *Performing Block-Level CDC Verification*.

If a block is not properly verified or the block constraints are incorrect or incomplete, clocks, domains, and other information assumed at block boundaries may be incorrect. This may result in false violations and mask real design issues.

Although SpyGlass CDC solution provides utilities to generate block assumptions (clocks, domains, etc.) automatically, this is not recommended for SpyGlass CDC verification sign-off. The user can review the abstract model, and adjust the blocks assumptions if needed

Performing Abstract View Validation in SpyGlass CDC

Run the *cdc_abstract_validate* goal after providing the block and its abstract view by using the following command:

```
sgdc -import <block-name> <block-abstract-view-SGDC-file>
```

During abstract view validation, the abstract view of a block is validated in context of an SoC. Provide the abstract view of a block by specifying the following command in the SoC-level SGDC file:

The abstract view contains block-level assumptions, such as clocks, resets, and domains on block inputs. These assumptions are validated with the constraints of the higher-level hierarchy.

The following points describe some examples of inconsistencies reported during abstract view validation:

- A block constraint associates two inputs to the same domain. However, these domains are controlled by different clocks in the higher-level hierarchy. Such issues are captured and fixed during validation.

- An abstract view defines a port to be equal to 1, while the higher-level block constraints causes the port to be equal to 0.

Approach to Fix Violations During Abstract View Validation

For a correct verification of an SoC, all violations reported during abstract block validation should be analyzed and fixed.

There are the following ways to fix these violations:

- If the SoC-level constraints are incorrect that caused the violations

  In this case, modify these constraints and rerun the *cdc_validation* goal.
- Block-level constraints are incorrect

  In this case, modify the incorrect constraints and repeat the following steps:
  - ❒ *Performing Block-Level CDC Verification*
  - ❒ *Generating Abstract View in SpyGlass CDC*
  - ❒ *Performing Abstract View Validation in SpyGlass CDC*

Examples of Fixing Violations During Abstract View Validation

Example 1

Consider the following schematic of a design:

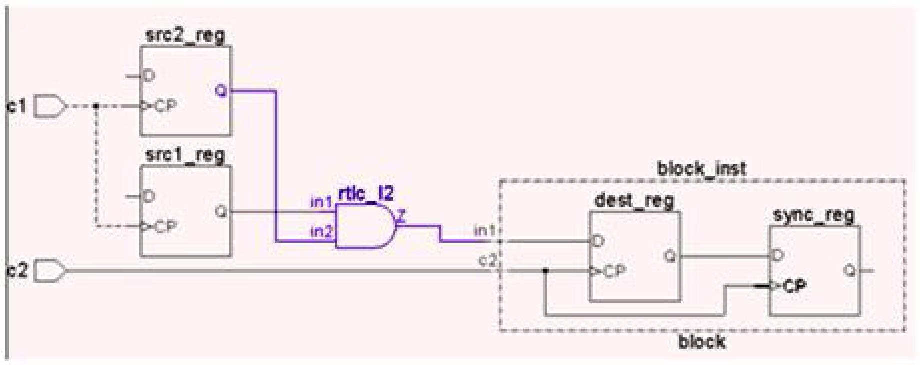


FIGURE 6. Example of Abstract View Validation

In the above schematic, `block` is correctly synchronizing an asynchronous signal using multi-flop synchronizer. However, the transmitter at the top-level is driving a combinational logic that is failing the block assumption that an asynchronous source signal should not be having combinational logic (it should be glitch free).

To fix such violation, latch the source signal in a flip-flop (on source domain) before it is sent to receiver block.

Example 2

The following example shows the violation that should be fixed in setup:

```
SGDC_set_case_analysis_validation02     Warning     test.v
2720    2     Simulated value '0' reaches to port
'txhcfc_en[3:0]' of block instance 'test.block_inst (block:
'block') however no set_case_analysis is specified in block
level constraint file
```

The above violation is suggesting that top-level constant is reaching the block port and in block constraints, set_case_analysis is not defined on the port. This will result in inaccurate SpyGlass CDC solution analysis of the block, because in absence of correct constants, either it may ne noisier or may miss certain violations.

NOTE: *Perform setup and setup-check at SoC level, prior to block validation step.*

Points to be Considered in the Hierarchical CDC Flow

Consider the following points:

- *Loss of Information While Generating the Abstract View*
- *Functional checks in an abstraction-based bottom-up methodology*

Loss of Information While Generating the Abstract View

Generating Abstract View in SpyGlass CDC results in loss of information.

During abstraction, the functionality information of the block is removed and the crossing information is preserved. Therefore, you cannot use abstract views to verify design functionality. However, you can verify metastability, convergence, block-to-block or block-to-top connectivity, and other structural issues.

Abstract views generated by SpyGlass CDC contain information using which you can perform:

- All structural synchronization checks
- Limited checks for multi-sampled signals (reported by *Clock_sync05*) and multi-synchronized crossings (reported by *Clock_sync09*).

Functional checks in an abstraction-based bottom-up methodology

Typically, synchronization circuitry is implemented in, sub-modules as opposed to a top-level SoC design. Therefore, verifying the functionality of sub-modules may be sufficient to cover critical functional issues such as gray encoding, FIFO correctness, etc.

In case functional verification is needed across module boundaries, capture module assumptions within each module. If a signal crossing module boundary is generating a multi-cycle pulse, the sender should verify that the pulse generated must be of a certain width and the receiver can assume that the pulse has the given width. SpyGlass CDC solution does not verify the sender assertion for the signal width. However, SpyGlass CDC solution can understand the signal width as an assumption for the receiver block while doing SpyGlass CDC verification of the block. The user can provide an OVL (Open Verification Library) assumption for the receiver side and verify the width of the pulse from the sender using an assertion based verification tool. For more details on OVL, refer to the Accelera Open Verification Library site at http://verificationlib.org.

Design Styles and Management

This section describes the following:

- *Handling Clock and Reset Nets Propagating Through Black Boxes*
- *Handling Clock Tree IPs*

Handling Clock and Reset Nets Propagating Through Black Boxes

One way to extend the clock domain propagation through a black box instance is to specify which output pins belong to the same clock domain as a particular input pin. This can be done by using the assume_path constraint.

Consider the following example:

```
assume_path -name BBOX -input d -output q qbar
```

The above specification indicates that the paths exist between input pin d and output pins q and qbar of the black box design unit BBOX.

Handling Clock Tree IPs

Typically, blocks, such as blocking blocks, PLLs, DLLs, and oscillators are analog, or at least have a non-synthesizable model. Section *Creating SpyGlass CDC Setup* describes a way to identify any such black boxes and solve them.

With regards to PLLs, they are generally black boxed; put the clock constraints at the appropriate output pins, with the domain set equal to the domain of the clock driving the input pin. An alternate (and possibly better) approach is to use the assume_path constraint as discussed earlier.

I/O cells are generally easy to identify because they either appear at the top level of the design, or inside a special block dedicated to I/O cells. Generally, each I/O cell has a modest number of I/O pins, one of which is typically called a PAD. I/O cells do have .lib models, but typically the model does not contain a function description because I/O cells are not optimized during synthesis.

The simplest way to deal with I/O cells is to black box them if possible. Do all your analysis from the inbound side of the I/O cells. It is possible to set the clock and other constraints on internal nets, so this should work fine. Even if the user wants to analyze through I/O cells, start with this approach and get the analysis as fine-tuned as you can before incorporating the I/O cell structure. You will find that this approach delivers useful results faster and with minimal manual intervention.

With regard to memories, it is important to understand that the only memories, which are natively recognized by SpyGlass, are inferred memories, that is, 2-dimensional arrays that appear on the left-hand side of an assignment, inside a sequential block. Instantiated memories are simply black boxes. All other memories will be reported as either black boxes (if no description is supplied) or un-synthesizable modules (if the memory size exceeds mthresh). For all the un-synthesizable modules for which memory size exceeds mthresh, SYNTH_5273 warning is generated.

In such cases, you should resolve those warnings by increasing the mthresh value.

It is quite common in a simulation to infer large memories (for example, 64k) with an intention of later replacing them with an instantiated memory. This can cause a big problem in synthesis, which blows inferred memories into one flip-flop per bit, causing memory explosion, and performance issues. SpyGlass provides the set_option mthresh <value> project file command (works only for Verilog) to handle this problem. With this command, SpyGlass will add up all the bits in a module and will black box (not synthesize) the module if it contains more than the specified number of bits (defaults to 4096 bits).

Limitations of the Hierarchical CDC Verification Flow

Following are the limitations of the hierarchical verification flow:

- Reset synchronizers propagated to block output ports are not abstracted.
- If a top-level port goes to different domain flip-flops inside an abstracted block, it is not reported by the *Clock_sync05* rule. Similarly, if a source flip-flop is synchronized multiple times inside such block, it is not reported by the *Clock_sync09* rule.
- The *Reset_sync01*, *Reset_sync03*, *Reset_sync04*, and *Clock_glitch01* rules do not support the *abstract_port* constraint. These rules support the *input* constraint.

  In such cases, use the *Ar_sync_group* rules instead of the *Reset_sync01*, and *Reset_sync03* rules.
- If the *abstract_port -sync* constraint qualifies a crossing inside a block, SpyGlass does not generate the *abstract_port -sync* inactive constraint at the output port of the block during abstraction of the block.
- If an input port is driving a multi-flop synchronizer inside a block, the *Clock_info15* rule generates the *abstract_port* constraint with a virtual clock and the *-combo no* argument.

  During constraints validation, if such port is driven by a clock that is same as the destination clock, the *SGDC_abstract_validation04* rule reports a false violation.

- Considerations for multi-mode analysis with respect to the hierarchical SoC flow.

  A block may operate in multiple modes. In the current abstraction flow, a block needs to be abstracted in each mode and used at the higher level of hierarchy separately.

  However, if a block has many modes or it can be parameterized, and is instantiated in a higher level of hierarchy multiple times with different parameters, the model can be dropped from abstraction.

  In such cases, constraint the module by using the *ip_block* constraint.

Recommended Guidelines to Perform SpyGlass CDC Verification

Using a systematic and step-by-step approach, it is possible to sign off SpyGlass CDC verification using SpyGlass. It is important to solve the last violation reported by SpyGlass to make sure no bug of SpyGlass CDC solution is left.

Following are some guidelines to follow:

- It is recommended to run all the SpyGlass CDC checks first at the RTL. Complex synchronization schemes, such as FIFO and handshake should be verified at RTL only.

  FIFOs may not be detected on post-synthesis and post-layout netlist designs.
- For large designs, it is recommended to use the divide and conquer technique where you first perform SpyGlass CDC checks on design blocks and then use the hierarchical CDC verification flow run on the complete SoC.

Appendix

This appendix covers the following topics:

- *Rules of the cdc_setup Goal*
- *Rules of the clock_reset_integrity Goal*
- *Rules in the cdc_verify Goal*
- *Rules in the cdc_abstract_validate Goal*
- *The Setup Manager of SpyGlass CDC*

Rules of the cdc_setup Goal

The *cdc_setup* goals runs the following rules:

Rule	**Description**
Clock_info03a	Reports unconstrained clock nets
Clock_info03b	Reports flip-flops, latches, or clock gating cells whose data pins are tied to a constant value
Clock_info03c	Reports flip-flops or latches where the clock/enable pin is set to a constant
Clock_info05	Reports MUX descriptions where two or more clock signals converge
Clock_info05b	Reports clock signals converging at a combinational gate other than a MUX
Clock_info15	Reports port domain information
Reset_info09a	Reports unconstrained asynchronous reset nets
Info_Case_Analysis	Provides schematic highlight of propagated values.
Clock_check07	Reports clock domains that reach another clock domain
Clock_converge01	Reports a clock signal whose multiple fan-outs converge
Reset_check03	Reports synchronous reset signals that are being used as active high as well as active low
Reset_check10	Reports asynchronous resets used as non-reset signals
Reset_check11	Reports asynchronous resets used as both active-high and active-low
Reset_check12	Reports flops/latches/sequential element that do not get active reset during power on reset
Clock_info18	Reports unconstrained ports summary
Ac_resetvalue01	Reports missing '-value' field in 'reset' constraint

Rules of the clock_reset_integrity Goal

The *clock_reset_integrity* goal runs the following rules:

Rule	Description
Clock_info05b	Potential glitch in clock tree due to clocks converging on combination gate (other than a MUX)
Clock_check01	Potential glitch in clock tree due to unexpected gates in clock tree (e.g. XOR gate in clock tree)
Clock_check04	Both positive and negative edges of clocks used in a same design
Reset_check01	Reset usage check against sync/async_set_reset pragma
Reset_check02	Glitches in reset paths due to unexpected gates (e.g. XOR gate in reset tree)
Reset_check03	Both positive and negative edges of synchronous reset used in a same design
Reset_check04	Both positive and negative edges of asynchronous reset used in a same design
Reset_check06	High fan-out reset nets not driven by placeholder cell
Reset_check07	Glitches on reset paths due to combinational logic on reset tree
Clock_Reset_check01	Glitches due to unwanted gates on clock or reset trees
Clock_Reset_check02	Race between flip-flop output and its clock/reset
Clock_Reset_check03	Race between flip-flop clock and reset
Info_Case_Analysis	Information on case-analysis to help debug violations
ClockEnableRace	Race between clock and enable of a same flip-flop
Clock_Reset_info01	Clock and reset usage matrix for information
Clock_glitch02	Gated clocks with improper enable logic
Clock_glitch03	Clock re-convergence at MUX
Clock_glitch04	Glitches due to combination logic driving flip-flops clock pin
Clock_converge01	Reports a clock signal whose multiple fan-outs converge

Rules in the cdc_verify Goal

The goal *cdc_verify* runs the following rules in addition to the *Ac_sync_group* rules:

Rule	**Description**
Clock_sync05	Reports primary inputs that are multi-sampled
Clock_sync06	Reports primary outputs driven by multiple clock domain flip-flops or latches
Clock_sync09	Reports signals that are synchronized more than once in the same destination domain
Ar_unsync01*	Reports unsynchronized reset signals in the design
Ar_sync01*	Reports synchronized reset signals in the design
Ar_asyncdeassert01*	Reports if reset signal is asynchronously de-asserted
Ar_syncdeassert01*	Reports if reset signal is synchronously de-asserted or not de-asserted at all
Reset_sync02	Asynchronous reset should not be generated in asynchronous clock domain
Reset_sync04	Asynchronous resets synchronized more than once in the same clock domain
Ac_cdc01a*	Data hold in multi-flop synchronized fast-to-slow crossing
Ac_datahold01a*	Reports synchronized data clock domain crossings where data can be unstable
Ac_conv01*	Check for sequential convergence of properly synchronized control crossings
Ac_conv02*	Check for combinational convergence of properly synchronized control crossings
Ac_conv03*	Convergence of synchronized signals from different source domains
Ac_cdc08*	Gray encoding of control bus crossing clock domains
Ac_fifo01*	FIFO overflow and underflow checks
Info_Case_Analysis	Provides schematic highlight of propagated values
Ac_clockperiod01*	Reports missing '-period' or '-edge' fields in 'clock' constraint

Ac_clockperiod02*	Reports clocks whose periods are rounded off by SpyGlass for lower design cycle
Ac_clockperiod03*	Reports correlated clocks whose design cycle is greater than the threshold value.
Ac_initstate01*	Reports a valid state of the design from which the formal analysis would actually start.
Ar_syncrst_validation*	Verifies user-defined synchronous resets
Ac_crossing01*	Generate spreadsheet for Crossing Matrix view
Ac_glitch03	Reports clock domain crossings subject to glitches

NOTE: * *means the rules and parameters that require Advanced CDC License.*

The cdc_verify goal also includes all the rules of cdc_setup_check goal. These are added to verify any new constraints, which may be added during verification.

Rules in the cdc_abstract_validate Goal

The cdc_abstract_validate goal runs the following rules:

Rule	**Description**
SGDC_abstract_port_validation01	Checks that the domain defined for a port is consistent with the domain that drives it from the higher-level block
SGDC_abstract_port_validation02	Verifies that a port with -sync specified is driven by a synchronizer from the higher-level block
SGDC_abstract_port_validation03	Verifies that the clocks of the synchronizer (source and destination) defined in abstract_port match those in the higher-level block
SGDC_abstract_port_validation04	Verifies that the combo parameter specified in abstract_port constraint matches what drives the port from the top-level block
SGDC_cdc_false_path_validation01	Verifies that the -from and -to clocks of a the cdc_false_path constraint are different in the top-level block
SGDC_clock_validation01	Verifies that no clock propagates to a port of the block if no clock constraint is defined in the abstract model
SGDC_clock_validation02	Verifies that a clock propagates to a port of the block if a clock constraint is defined in the abstract model
SGDC_clock_domain_validation01	Verifies that two or more ports that have the same domain in the abstract model receive the same clock from the top-level block
SGDC_clock_domain_validation02	Verifies that two or more ports that have different clocks domain in the abstract model receive different clocks from the top-level block
SGDC_define_reset_order_validation01	Verifies that the resets defined in from and to fields of the define_reset_order constraint are driven by resets in the higher-level block
SGDC_define_reset_order_validation02	Verifies that the resets defined in from and to fields of the define_reset_order constraint are driven by different resets in the higher-level block

Rule	Description
SGDC_input_validation01	Verifies that the domain of the clock defined in input/abstract_port constraint matches the domain of the clock that drives the port in the higher-level block
SGDC_input_validation02	Verifies that if no input/abstract_port constraint is defined, then the port is not driven by a flip-flop in the higher-level block
SGDC_num_flops_validation01	Verifies that the clocks specified in from_clk and to_clk of num_flops constraints are not the same in the higher-level block
SGDC_num_flops_validation02	Verifies that the number of flip-flops in the num_flop constraints for a clock pair in the abstract model matches the number of flip-flops for the corresponding pair in the higher-level block
SGDC_reset_validation01	Verifies that a port with no reset constraint in the abstract model is not driven by a reset in the higher-level block
SGDC_reset_validation02	Verifies that a port with a rest constraint in the abstract model is driven by a reset in the higher-level block
SGDC_reset_validation03	Verifies that top and block level asynchronous and synchronous reset types are not conflicting
SGDC_reset_validation04	Verifies that the active value of a reset for a port defined in the abstract model matches the value of the reset that drives the port in the higher-level block
SGDC_qualifier_validation01	Verifies that the clocks specified in from_clk and to_clk of a qualifier constraint are not the same in the higher-level block
SGDC_qualifier_validation02	Verifies that if a port does not have a qualifier constraint in the abstract model, then no qualifier drives the port in the higher-level block
SGDC_set_case_analysis_validation01	Verifies that the value of a set_case_analysis constraint on a port in the abstract model matches the value propagated to the port in the higher-level block

Rule	Description
SGDC_set_case_analysis_validation02	Verifies that if a port does not have a set_case_analysis constraint in the abstract model, then no constant value is propagate to that port in the higher-level block
SGDC_virtualclock_validation01	Verifies the validity of block-level virtual clock with higher-level clocks

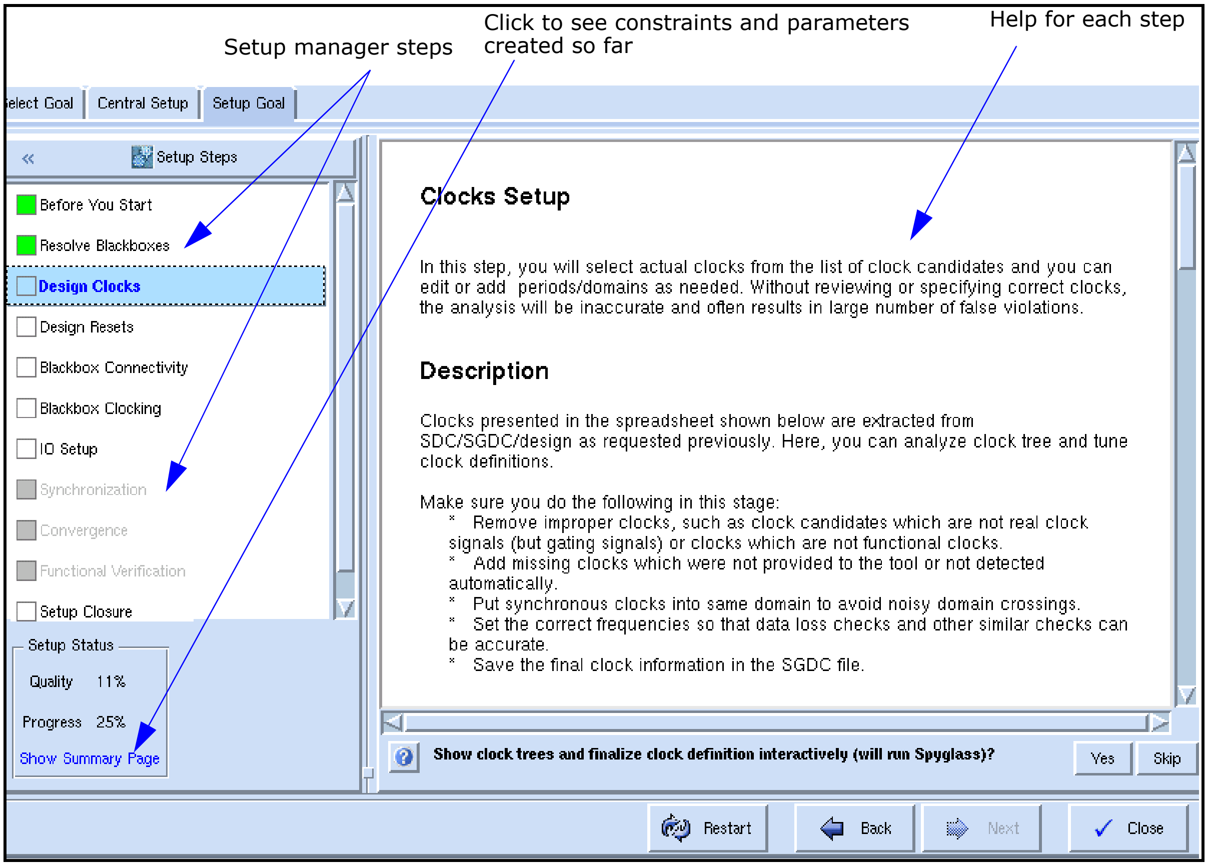
The Setup Manager of SpyGlass CDC

The setup manager guides designers with little tool and design knowledge to achieve a design setup as complete as possible. It enables you to:

- Extract and complete clocks and reset definitions in a design.
- Configure black boxes.
- Set boundary assumptions (IO assumptions).
- Define appropriate synchronization practices for the given design.

The quality of a setup dictates the quality of SpyGlass CDC analysis. Incorrect or incomplete setup cause many false violations or mask design bugs.

The following figure shows the setup manager:



Clocks Setup

In this step, you will select actual clocks from the list of clock candidates and you can edit or add periods/domains as needed. Without reviewing or specifying correct clocks, the analysis will be inaccurate and often results in large number of false violations.

Description

Clocks presented in the spreadsheet shown below are extracted from SDC/SGDC/design as requested previously. Here, you can analyze clock tree and tune clock definitions.

Make sure you do the following in this stage:
* Remove improper clocks, such as clock candidates which are not real clock signals (but gating signals) or clocks which are not functional clocks.
* Add missing clocks which were not provided to the tool or not detected automatically.
* Put synchronous clocks into same domain to avoid noisy domain crossings.
* Set the correct frequencies so that data loss checks and other similar checks can be accurate.
* Save the final clock information in the SGDC file.

Show clock trees and finalize clock definition interactively (will run Spyglass)?

FIGURE 1. Setup Manager of SpyGlass CDC

In the above wizard, if a step is not relevant for the current design or project, it appears disabled or hidden.

Before proceeding to setup verification, ensure that all domains and frequency information for each clock is properly defined during clock setup.

NOTE: *The default mode in the setup manager of SpyGlass CDC solution allows only some of the features namely, "Clocks", "Black Box", "Resets", "IO Setup", and "Setup Closure". To use all the features of the setup manager of SpyGlass CDC solution, you can select the "Advanced mode" option from the "Before You Start" step.*

NOTE: *Frequency information is needed for functional checks only. If a design can operate with a range of frequencies, identify the worst and best frequencies that cover all* *corner cases and run CDC verification with each frequency setting.*

Invoking the Setup Manager

To invoke the setup manager, perform the following steps:

1. Select a SpyGlass CDC goal under the *Select Goal* tab.
2. Click the *Setup Goal* tab.
3. Click the *Run Setup Wizard* button.

After performing the above steps, the first screen of the setup-manager wizard appears.

Limitations of the Setup Manager

Following are the known limitations of the Setup Manager of SpyGlass CDC:

- If clocks and other constraints are specified in an SGDC file and clocks are also created by the *Clock Setup* step in the SGDC file in the *Setup Manager*, the *Setup Manager* only considers the generated SGDC file.

  It is recommended that you consolidate both the SGDC files. You can take clocks from the generated SGDC file and other constraints from the SGDC file specified by you.
- The *Reset Setup* step does not have the interactive setup similar to the *Clock Setup* step. It creates the *autoresets.sgdc* file.

  It is recommended that you review the *autoresets.sgdc* file and add/ delete/modify the `reset` constraints from this file.
- In the VHDL and mixed flow, if the SGDC file (which has the `sdc_data` constraint) has *`<entity.architecture>`* in `current_design`, and you perform the following steps, clocks will not be used by setup step of the SpyGlass CDC goals:

  a. Select the `cdc_verif_base` goal in the Console GUI.
  b. Click on the *Setup* tab.
  c. Choose to import constraints from an SDC file.

  To solve this problem, use *`<entity>`* in `current_design` instead of

`<entity.architecture>`.

- Auto-save is not supported in the *IO Setup* step of the *Setup Manager*.

  If you complete a step and perform the next steps, and then go to the previous step that is completed and choose to skip that completed step, the *Setup Manager* highlights that step in red color.