SpyGlass®-CDC Methodology Series

CDC-Clean Design Sub-Methodology

Updated: March 31, 2008

Existing SpyGlass customers: please check the Methodology subdirectory of your SpyGlass installation to see if there is an updated version

Table of Contents

Sub-methodology for CDC-Clean Design Using SpyGlass-CDC

1 CDC-Clean Design – The need

Clocks that are asynchronous with respect to each other may reach different flops at slightly different times in each cycle during the execution of the design. This timing uncertainty may cause setup and hold-time violations randomly in the design. Setup and hold time violations can cause functional failure in the chip.

This issue cannot be completely identified using traditional verification methods, such as simulation and static timing analysis. Static clock domain crossing analysis and verification is the most efficient way of verifying CDC correctness.

The need is to detect clock domain crossing at RTL level and make sure proper synchronization has been added in the circuit.

2 Introduction

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

We begin by introducing basic problems associated with CDCs in section 3. Section 4 describes the approach to solving CDC problems.

In Section 4 we describe the overall framework of the SpyGlass CDC methodology.

Section 4.1 discusses the knowledge you must have before beginning analysis and the methods required to make CDC analysis more manageable, such as partitioning the design into smaller design units and providing SpyGlass with synthesizable RTL.

Subsequent sections lay out the individual methodology steps. Sections 4.2 and 4.3, respectively, describe SpyGlass setup for CDC verification, and setup checks. One important theme of these sections is the need to provide additional design information that is not apparent in the RTL description. This information is required to constrain the design for correct analysis and is specified in the SpyGlass Design Constraints (SGDC) file.

Section 4.4 describes the actual process of CDC verification and debugging, covering main CDC issues. This section explains how to run CDC rules, how to minimize the number of false violations being reported, and how to debug violations.

Section 4.5 describes an additional class of SpyGlass CDC checks, dealing with implementation-related CDC issues.

Finally, section 4.6 discusses issues you might face while integrating the partitioned design units using a bottom-up approach—issues such as how to handle the PLL, I/O pads, memories, and so on. This will help you smooth the transition while integrating the partitioned design.

This document covers main use model for CDC verification; SpyGlass has capabilities not covered by this document, for more details on all SpyGlass CDC capabilities please refer to the SpyGlass Clock-Reset Rules reference.

2.1 Tool Versions

- SPYGLASS VERSION 4.0.1
- Templates referred to in this document are available at the following path: $SPYGLASS_HOME/Methodology/Clock-reset/

2.2 References

- SpyGlass Clock-Reset Rules reference
- SpyGlass Predictive Analyzer User Guide

2.3 Terminology

Clock domain: Refers to clocks that have constant phase relationship with each other. Typically a clock, its inverted form, and its divided form are considered to be in the same domain. . Divided forms only have a constant phase relationship as long as the division ratios have a common factor. A divide-by-2 and divide-by-4 have constant phasing but a divide-by-3 and divide-by-4 do not.

CDC: Clock Domain Crossing, refers to a path connecting a sequential element/flop/primary input/black box controlled by one clock domain to another sequential element/flop/primary input/black box clocked by another clock domain.

Synchronizer: Part of a design which ensures proper transfer of signal values across clock domains.

Quasi-static: Refers to flops that are taking constant value during regular operation of a design (they may change value during setup and initialization of the design; or may change value when a block powers on or power off, etc.). Often, quasi-static flops do not require synchronizers even if they are involved in clock domain crossings.

LCM: Least common multiple used in this document to identify a common clock period for a design with multiple clocks with different periods.

3 Concept – The CDC Problem

Clocks that are synchronous with respect to each other are referred to as same-domain clocks; clocks that are asynchronous to each other are in different clock domains. Edges of clocks coming from the same clock domain are always *aligned* for all registers in the design and for all time throughout the execution of a design. As a result, if setup and hold time for a flop input is honored, there is no risk in capturing the data for the flop throughout the design. On the other hand, clocks from different domains may reach different flops at slightly different times in each cycle during the execution of the design. This timing uncertainty may cause random setup and hold-time violations. This issue cannot be completely identified using traditional verification methods, such as simulation and static timing analysis. Static clock domain crossing analysis and verification is the most efficient way of verifying CDC correctness.

There are four main problems associated with CDCs:

- Metastability
- Data hold in fast-to-slow crossings
- Data correlation and race conditions
- Issues related to complex synchronizers

3.1 Metastability

Metastable values are created and propagated due to setup and hold-time violations in an asynchronous crossing. The issue is illustrated below. A metastable waveform generated at B is subject to interpretation by each branch in the fanout of B. One gate in a fanout can perceive the metastable wave as a logical value 1 while another fanout will see the same net being 0. This free interpretation will cause functional failure in the design. The functional failure will arise randomly in the execution of the design:

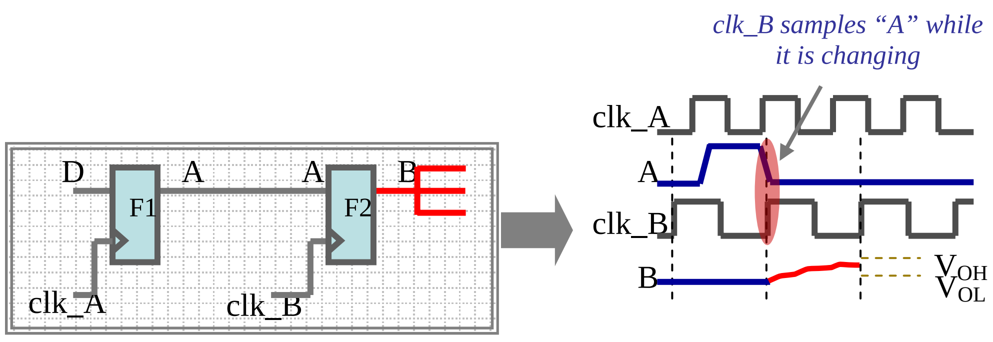


Figure 1: Metastability in CDC: metastable wave at "B" maybe interpreted as a logical 1 or 0 by downstream logic

Solutions that prevent such metastability from being propagated can be categorized as follows:

1- Control signal synchronization: control signals crossing clock domains are typically synchronized using multi-flop synchronizers. Multiple stages of flops will transform the metastable value to a “cleaner” 0 or 1 before it is passed to downstream logic.

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

Common synchronization schemes used for both control and data signals by most designers are illustrated below:

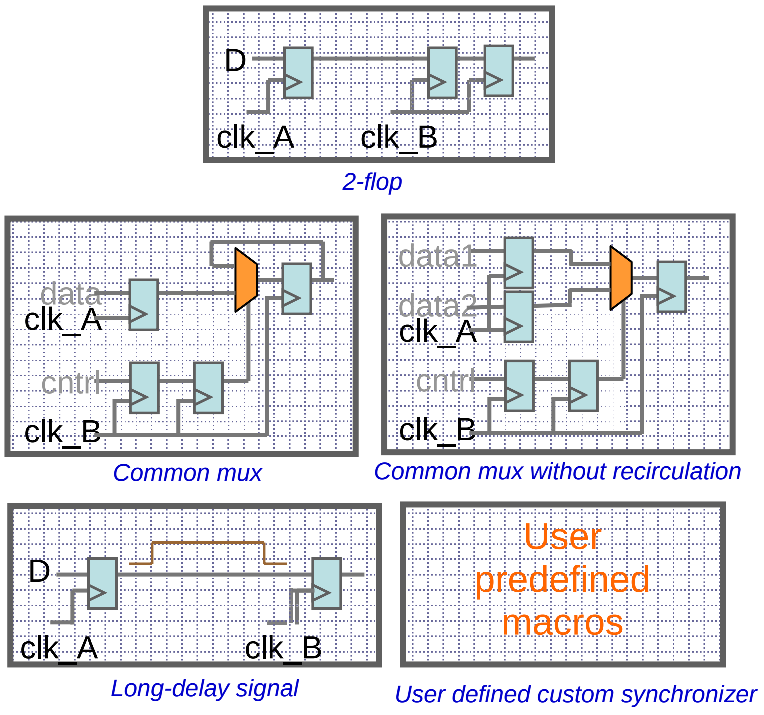


Figure 2: Common synchronization schemes

Common mux without enable is a generalized recirculation mux where one of the inputs of the mux is coming from destination domain instead of the flop receiving the mux. The enable in this case is synchronizing the data coming from the asynchronous input of the mux.

In addition, more complex handshake and FIFO synchronization techniques are used for data transfer, especially across blocks in a design.

In all cases, the presence of such synchronizers on each asynchronous crossing needs to be verified to prevent metastability propagation causing random functional failure.

3.2 Data Hold in Fast-to-Slow Crossings

This problem arises when a short pulse generated in a fast clock domain is fed into a slow clock domain. Under such circumstances the short signals may miss the active edge of the slow clock domain and will not be captured in the destination. The following figure illustrates the data-hold problem in fast-to-slow crossings:

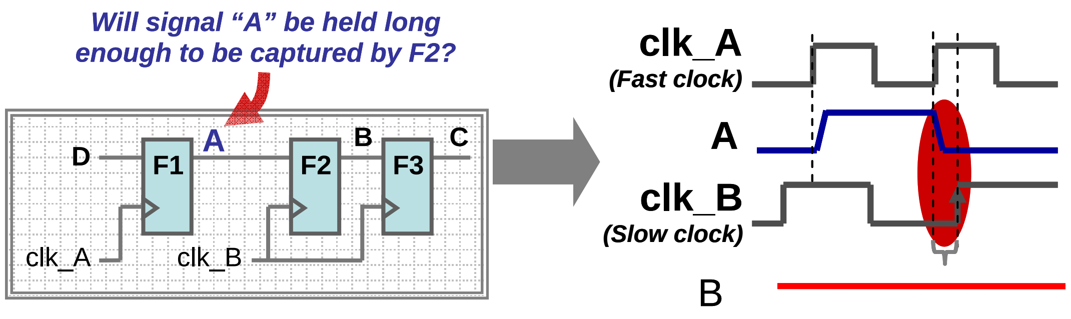


Figure 3: signal "A" not held long enough for slow clock clk_B, causing B to miss the pulse

Typically a handshake or custom circuit is used to extend the pulse for at least one full cycle of the slow destination clock. All fast-to-slow crossings need to be functionally verified to make sure that such extenders exist and no short pulse is generated in the destination.

Similarly in case of enabled flops involved in a crossing; it is important to make sure that the data is stable before the enable is asserted and the data does not change when the enable is on. In case of a handshake this property is maintained by the request-acknowledge protocol which will stabilize the data before sending a request.

3.3 Data Correlation and Race Conditions

As mentioned above, the presence of synchronizers prevents metastability and settles the metastable value to a "clean" 0 or 1 value before feeding it to the downstream logic. Assuming that the source remains stable long enough, its value will be transferred to the destination. However, this transfer may not happen immediately, due to metastability, and this can cause problems for multiple signals that are correlated (i.e., their combined value matters, like a state vector) and crossing clock domains. One or more signals may be deferred relative to others. This results in a loss of correlation, which will lead to unknown state generation in the destination and will cause random functional failure.

This "reconvergence" problem is illustrated in Figure 4.

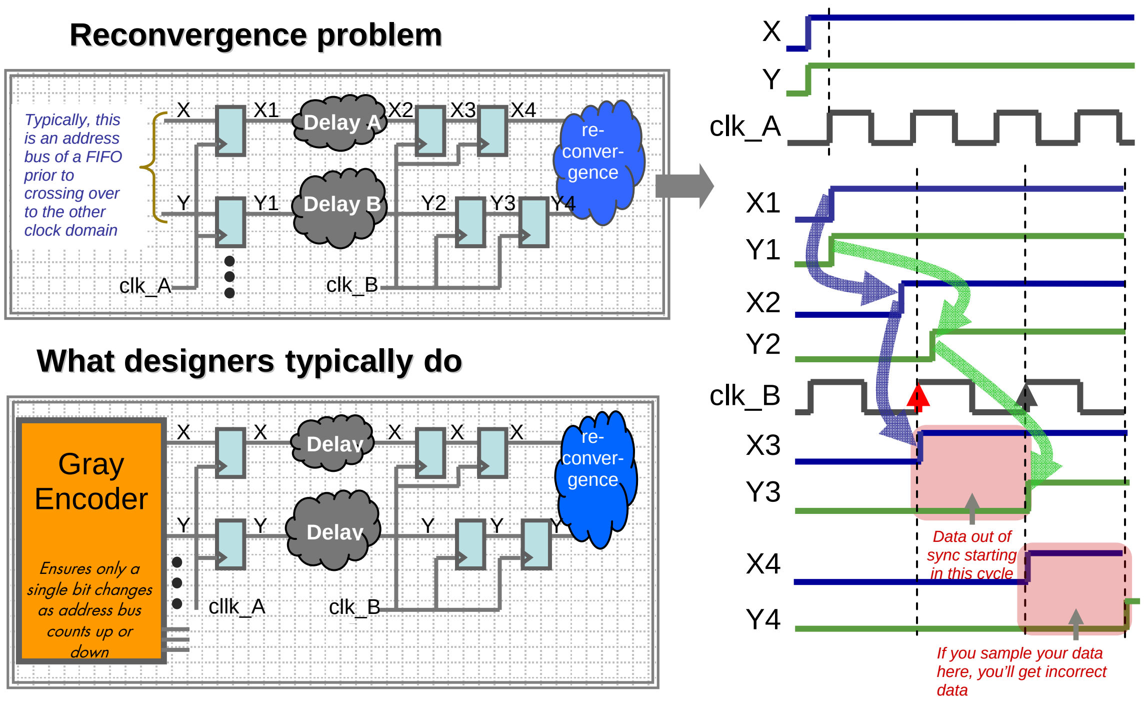

Reconvergence problem
Typically, this is an address bus of a FIFO prior to crossing over to the other clock domain
X
Y
X1
Y1
Delay A
Delay B
X2
X3
X4
Y2
Y3
Y4
re-conver-gence
clk_A
clk_B
What designers typically do
Gray Encoder
Ensures only a single bit changes as address bus counts up or down
cllk_A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

Figure 4: The reconvergence problem and a typical solution using gray coding

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

It is important to verify that correlated signals are gray encoded before crossing clock domains. Correlated signals may be identified wherever independent signals are converging and being used in the same combinational logic, or when a bus is used as a state vector or a memory pointer.

3.4 Complex Synchronizers

FIFOs and handshake mechanisms are often used to transfer data from one domain to another.

The following figure illustrates typical FIFO synchronizer architecture.

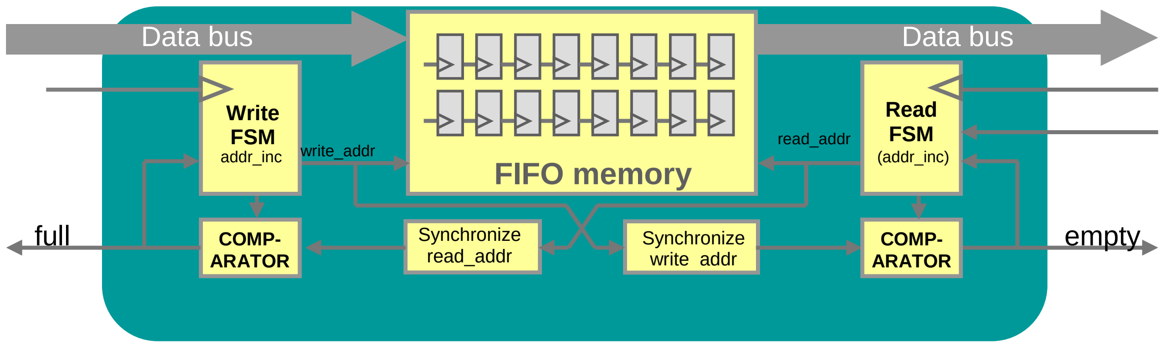


Figure 5: FIFO synchronization scheme

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.

Similarly, handshakes typically involve 2 FSMs controlling requests and acknowledge mechanism to stabilize data and capture it at appropriate time. This structure needs to be verified from a metastability perspective as well as specific handshaking protocol perspective.

3.5 Additional CDC Issues

In addition to the above CDC issues, there are other problems associated with clock domain crossings. For example, when a signal is synchronized more than once in the same destination domain, it can create a race condition or a data-correlation problem. Yet another class of problems is related to implementation of clocks, resets and crossings. For more information on other CDC issues, refer to the SpyGlass Clock-Reset Rules reference.

4 Approach

The following figure illustrates the recommended steps for block-level CDC verification using SpyGlass. Chip level CDC verification follows the same steps with minor variations as described in section 4.4.4. The following sections of this document will detail each step of this flow.

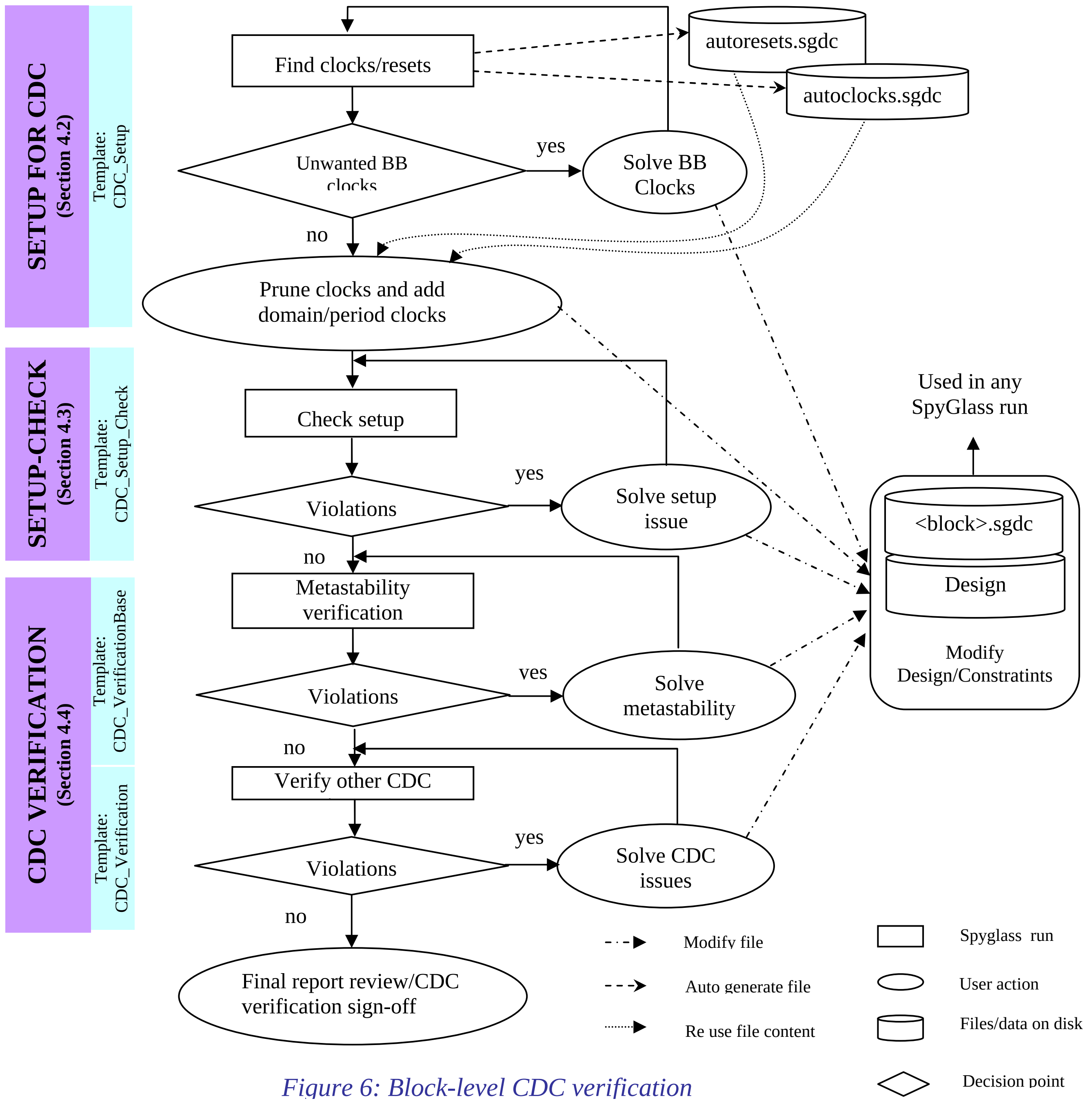


Figure 6: Block-level CDC verification

4.1 Before You Begin Verification

Before you perform CDC verification, you need to acquire some knowledge about the clocks and resets in the design, either by yourself or with the help of a design owner. First, determine whether the analysis needs to be carried out at the full-chip level or at the block level. Unless the full chip is small enough (500K-1M gates), CDC verification should be completed at the block level then on the full chip.

Even at the block level, we recommend that you use a bottom-up approach. Then, as you make progress, start integrating the blocks one-by-one, and progressively move up the hierarchy until you reach the top level of the device. Although the analysis can be run on multi-million gates design, verification can be more productive and can be concluded sooner if the partitions/modules size is limited to 500K to Million gates.

As you integrate the design units, you need to ensure that the integration has been done correctly. You may have some questions such as:

- How to handle the third-party IP.
- How to handle PLLs.
- How to handle the I/O pads and the memories in the design.
- How to handle instantiated cells such as clock-gating cells

We will discuss these questions in section 4.6; initially, our purpose is to perform CDC verification at the block level. Because we are initially working at the block level, you might want to additionally verify the following:

- Does the block have one clock or multiple clocks?
- Are the clocks synchronous with respect to each other?

If the block has only one clock or has multiple clocks but all of them belong to the same domain, you do not need to check for CDC problems in the block. However, you may want to check the validity of your assumptions (e.g. finding if single clock is feeding all flops in the design) by running setup check (second step of CDC verification methodology).

> Note: An exception to the above is synchronous FIFO verification. SpyGlass identifies and verifies both synchronous FIFOs (where FIFO read and write pointers are in the same domain) and asynchronous FIFOs (where FIFO read and write pointers are from different domains).
>
> After you identify the block on which you want to perform the CDC verification, you will have to provide SpyGlass with the design information that is not apparent in the RTL description, such as clock net and reset net information. This is done through the SpyGlass Design Constraints (SGDC) file.

4.2 Setup for CDC Verification

This step consists of defining clocks and resets for the block. Once the initial setup is completed, it will be checked for consistency and correctness during the setup-check step, where operating modes and set-case-analysis will be defined.

If you have clocks and/or other constraints in SDC format, they can be important into SpyGlass. How to read a SDC file into SpyGlass is described later in this section.

SpyGlass provides a template, CDC_Setup, that packages rules needed for setup. Following are the main rules and parameters in CDC_Setup template:

- Clock_info01: Identify clock candidates
- Reset_info01: Identify reset candidates
- Parameter ignore_latches: Ignores clocks driving only the latch enable pins

Running CDC_Setup template will create an inferred clock constraints file called autoclocks.sgdc and an inferred resets constraint file called autoresets.sgdc. The above two files can be found in the /spyglass-reports/clocks-reset directory. These files should be concatenated and saved at another location (we recommend a file called <block>.sgdc in the current working directory) because the auto-generated files will be overwritten during the next SpyGlass run.

Here is an example showing concatenation of the two automatically generated files:

```
cat spyglass_reports/clock_reset/auto* > <block>.sgdc
```

The following is an example of a concatenated constraints file (actual file lines are separated by comments describing their meanings):

```
current_design block
```

The above indicates the top level module for which the constraints are defined.

```
clock -name "block.clka" -domain domain1 -value rtz
clock -name "block.clkb" -domain domain2 -value rtz
clock -name "block.clkc" -domain domain3 -value rtz
```

For the current design (block) three clocks have been inferred: clka, clkb, and clkc. Each clock assumes a different clock domain. Because the clocks take a different domain, they are assumed to be asynchronous to each other. As explained later in this document, you can combine the clocks in the same domain by assigning the same domain name to them.

```
current_design block
```

Because the reset constraint specified below applies to the same block, we can delete this occurrence of current_design.

```
reset -name "block.reset" -value 1
```

The reset keyword is used to apply a reset to the block. For functional checks resets are initially assumed to be active, allowing flops to be initialized, and then become inactive for functional analysis purposes. Reset inactivation prevents false violations due to resets (e.g. gray encoding failure due to reset transitioning a vector from 11 to 00). Reset constraints are also used for other critical checks such as Reset_sync01 to enforce that the reset is synchronously deactivated. For more information on reset constraint refer to the Clock-Reset Rules Reference.

The generated constraints files may include some control signals in addition to the real clocks and resets. For instance, if enable and clock are combined through complex combinational logic and it is not possible to differentiate between the two, both enable and clock maybe reported as probable clocks in the generated constraints file. However, the automatically generated files will not include the clock sources that are generated within the block[1]. You should review each inferred clock and reset and remove those signals that are clearly not real candidates. It is recommended that you look at the Clock_info01 and Reset_info01 rule messages in the SpyGlass Design Environment's Incremental Schematic (IS) view to review the inferred signals.

If the block can be configured to operate in *n* number of ways, you will need to perform CDC verification *n* number of times, each with a different configuration. For example, if the block can be configured as a 16-bit device in mode 1 and as a 32-bit device in mode 2, you will need to perform the CDC verification twice, once for each mode.

Such modes have to be added to the constraints file. The setup check described in the next section will help in defining set-case-analysis that constrains clock propagation through clock trees.

Another important task during setup is to validate black box clocks. If a black box clock is a valid clock source (e.g. PLL clock), its domain needs to be provided and no further action is needed. If a black box clock is derived from another source clock (e.g. a clock port traversing a black box representing a functional block for which a synthesizable model is not provided), the path through black box should be provided to SpyGlass using assume_path constraint (see section 5.1). If no black box is present in the design, use SpyGlass command line option "-nobb" to enforce the assumption. For more detail on "-nobb" refer to SpyGlassPredectiveAnalyzerUserGuide.pdf.

For the command/files level detail of the step, please refer to the section 5.1.

If you have constraints defined in SDC format, SpyGlass can read the file and use relevant constraints for CDC verification. SpyGlass can either directly read and use constraints from SDC file or it can translate CDC related constraints of SDC into a SGDC file that can be used in subsequent runs.

To be able to read a SDC file into SpyGlass, you need to follow the following steps (assume block1.sdc and block2.sdc are two SDC files that need to be used for CDC verification of a top level module "top"):

- Create a SGDC file that contains the following lines:

  ```
  current_design top
  sdcschema –file block1.sgdc block2.sgdc
  ```

- Run spyglass with the following command line:

  ```
  spyglass –verilog <rtl_files> -policy=none –sdc2sgdc –sgdc sdc.sgdc
  ```

  (Replace "policy=none" by "policy=clock-reset" to use the SDC constraints on the fly and continue with CDC verification)

- Copy resulted sgdc file spyglass_reports/sdc2sgdc/sdc2sgdc.sgdc file in your working directory

- Examine the content and adjust or add new constraints as needed and use it for CDC analysis.

[1] Typically, this is not a problem at the block level.

Note that only following commands of SDC are translated into SGDC:

- “create_clock”
- “create_generated_clock” – this will be commented in output SGDC. In current run, this command will not be used.
- “set_input_delay”
- “set_output_delay”
- “set_case_analysis”

Note that any error in SDC file will result in empty SGDC file creation and halt of further translation or analysis. Furthermore, while translating SDC commands into SGDC commands, SpyGlass will perform existence and sanity checks; if such checks fail an empty SGDC file will be generated and no further CDC analysis will be carried out.

4.3 Setup Check

Defining clocks and resets provides initial constraints out of which the setup can be completed using capabilities described in this section. The following aspects of setup need to be completed before starting CDC verification:

- Make sure that each flop is receiving one and only one clock
- Make sure that one clock definition is not masked by another clock definition
- Make sure that constraints (set_case_analysis, clocks, resets) are correct and not conflicting with each other
- Assign domain and frequency information for each clock (frequency information is needed for functional checks only. If the 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)

SpyGlass helps to check the above aspects of CDC setup. The exceptions to this are definitions of clock domains and frequency, which are known only to the designer (assuming that a block is first created and top-level clocks are not available). This information must therefore be entered by the designer.

In addition to the above checks, SpyGlass performs basic constraints sanity check. When a constraint file is provided, SpyGlass will do a series of sanity checks for design objects existence and constraints correctness. These built-in rules are always run and would produce fatal errors if any constraint is not correctly specified. Such rules are prefixed with “SGDC_”. These rules apply to any step of the methodology as long as a constraint file is provided. Fix the issues and apply all the constraints correctly before doing any further analysis. For more information on SGDC sanity checks refer to Clock Reset Rules Reference.

4.3.1 Rules-Checking Setup

Once an initial set of constraints has been defined, run SpyGlass’s CDC_Setup_Check template to check for setup issues mentioned above. CDC_Setup_Check includes following rules:

Checks	**Description**
Clock_info03	Unclocked flops
Clock_info05	Clock convergences

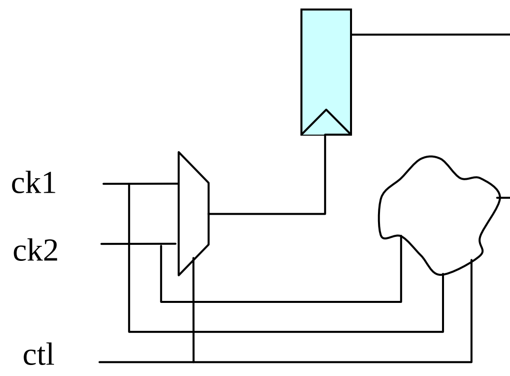
Checks	Description
Clock_info05a	MUX selects to be constrained in clock trees
Clock_info14	Clock tree color map in schematic
Clock_info15	Report port domain info
Reset_info09	Unconstrained resets
Info_Case_Analysis	Provides schematic highlight of propagated values.
Clock_check07	Clocks defined on the path of other user clocks (pre-requisite; does not appear in template)
Ac_sanity04	Conflicting constraints in SGDC file (pre-requisite; does not appear in template)

For a full description of these rules, refer to the SpyGlass Clock-Reset Rules Reference.

Check for Clock_info03a violations, which flag the parts of the clock tree that the top-level clock sources do not reach. Typically, this is caused by the missing clock constraints in the SGDC file or black boxes[2] in the clock tree through which the clock cannot be propagated due to missing structural information or incorrect case analysis settings. This means that the clock gates or muxes are incorrectly sensitized.

An important aspect of clocks setup is constraining clock trees. Clock trees often use muxes combining various clocks for various operating modes of the design. In such cases, you need to define the operating mode (set-case-analysis for mux control) by constraining the select of the mux using the set_case_analysis constraint command in the constraints file. If these muxes are not configured, then multiple clocks may drive the same flop. In this case many flop-to-flop paths may be seen as asynchronous crossings even if in the actual design execution these paths are synchronous (due to proper configuration). This situation may lead to a high noise level, leading to higher CDC verification closure time. You can choose to analyze the design under multiple modes simultaneously (by not constraining the clock tree completely, and letting multiple clock domains controlling a same flop), however closing CDC verification may become a harder task: SpyGlass will do a pessimistic interpretation of crossings, consequently if two clocks reach two flops through two different clouds of logic, the two flops will be considered to belong to two different domains. This may lead to false unsynchronized crossing violations.

[2] How to handle black boxes will be addressed in section 9



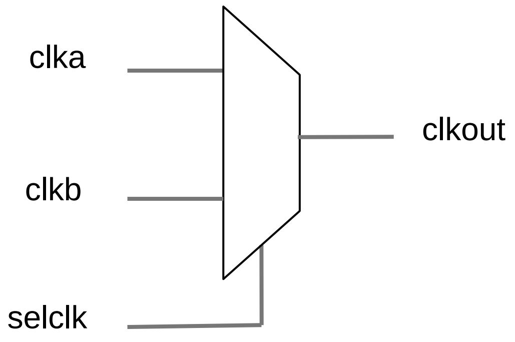
Example of clocks convergence: without any constraint on ctl, SpyGlass will consider the path between the two flops as clock domain crossing, if not synchronized this will result in CDC violation.

Clock_info05a reports all occurrences where you need to define a set_case_analysis constraint. If proper set-case-analysis is defined, a single clock will reach each flop.

The set_case_analysis command can be applied to ports, internal nets and configuration registers that are assumed to be constants during the analysis.

The keyword set_case_analysis does not appear in the automatically generated SGDC files, but is important for correct setup.

If you see a Clock_info05a rule under the INFO category in the message window, then you will need to specify a set_case_analysis constraint in your constraints file.



Example 1

In the above example, the selclk pin has been constrained, using the set_case_analysis constraint, to the value 0. This implies that clka is being propagated to clkout.

```
set_case_analysis –name "block.selclk" –value 0
```

The propagation of case analysis can be observed using the Info_Case_Analysis rule. This can be added to the Clock_info05a schematic highlight by first selecting the Clock_info05a rule message and then, while holding down the **<Ctrl>** key on the keyboard, double-clicking the Info_Case_Analysis rule message to add the information on how the case analysis has propagated through the design.

While analyzing the Setup_checks results, it is worth reviewing how the clock and reset trees have propagated. This can be done by viewing the Propagate_clocks and Propagate_resets messages in the Incremental Schematic (IS). In this mode, the fanout of the clock to each block can be seen. The IS will show the first instance of the flop/latch connected to the clock that was detected in the block to verify that the clock drives into the block. However, only one flop per block is shown, in order to keep the schematic manageable and understandable. The fanout of each branch is also annotated in the IS (for example, an annotation of **F:234** on the fanout of a branch indicates that the branch drives a total of 234 flops).

4.3.2 Assigning Domain and Frequency Information for Clocks

Clocks declared in the constraints file are associated with a different domain and hence are assumed to be asynchronous with respect to each other. If clocks are synchronous to each other, then the user will need to modify the constraints file and modify the argument to the –domain parameter.

An example of clocks in the same domain is a block which receives a clock as well as the clock divided by two as input. In such cases, the two clocks may be considered to be in the same domain.

In the example below, clka and clkb are synchronous to each other. clka has a period of 10 (the units are not important as these are relative numbers; let’s say 10ns), while clkb has a period of 20.

```
clock -name "block.clka" -domain domain1 –period 10 -value rtz
clock -name "block.clkb" -domain domain1 –period 20 -value rtz
```

For the command/files level detail of this step, please refer to the section 5.2 .

4.4 CDC Verification

This section describes how to verify and debug the main CDC verification issues. First, it describes block-level verification. Next, it provides tips to help debug and analyze CDC issues using SpyGlass. Finally, it describes top-level CDC verification by automatic abstraction of blocks for which CDC issues have already been investigated.

4.4.1 Block-level verification

The following checks offered by SpyGlass cover a large spectrum of CDC problems, including the four main CDC issues discussed in section 3 above as well as other critical CDC problems. These rules are packaged in two templates, CDC_VerificationBase and CDC_Verification. CDC_VerificationBase is composed of the following:

- Clock_sync01, Clock_sync02 rules for meta-stability check
- enable_handshake, enable_fifo and sync_reset parameters

CDC_VerificationBase has to be run and all violations need to be closed before the CDC_Verification template is invoked. CDC_Verification includes the same parameters (enable_handshake, enable_fifo, sync_reset) and the rules in the following table:

Checks	Description
Clock_sync01	Reports signals crossing clock domains without being synchronized
Clock_sync02	Reports signals crossing clock domains that are properly synchronized
Clock_sync02p	Reports domain crossings synchronized using a user-specified pattern
Clock_sync03a	Reports converging signals originating from same source domain that are separately synchronized into a single destination domain and performs gray encoding check on those signals.
Clock_sync03b	Reports convergence of signals originating from different clock domains.
Clock_sync05	Reports primary inputs that are multi-sampled
Clock_sync06	Reports primary outputs that are multi-transitioned
Clock_sync08	Reports bus signals that are not synchronized using common enable
Clock_sync08a	Reports multi-flop synchronized bus bits where double flop outputs belong to the same bus
Clock_sync09	Reports signals that are synchronized more than once in the same destination domain
Reset_sync01	Reports asynchronous reset that is not synchronously de-asserted relative to a clock
Reset_sync02	Asynchronous reset should not be generated in asynchronous clock domain
Ac_cdc01a	Data hold in multi-flop synchronized fast-to-slow crossing
Ac_cdc01b	Data hold in properly synchronized fast-to-slow crossings not using multi-flop synchronizer
Ac_cdc08	Gray encoding of control bus crossing clock domains
Ac_fifo01	FIFO overflow and underflow checks
Ac_handshake01	Data hold between request and data
Ac_handshake02	Proper request and acknowledge sequencing
Info_Case_Analysis	Provides schematic highlight of propagated values

CDC_VerificationBase and CDC_Verification templates also include all the rules of CDC_Setup_Check template. These are added to verify any new constraints, which may be added during verification.

CDC_VerificationBase contains Clock_sync01/Clock_sync02 rules for meta-stability check. Violations for these rules should be resolved before other checks are run using CDC_Verification. This sequence is also illustrated in CDC verification flow in figure 6. CDC_Verification template is a superset of the base template; base clock crossings and synchronization analysis is a prerequisite for most of the CDC verification rules (e.g. Ac_cdc01, Clock_sync03a, etc.).

For both CDC_VerificationBase and CDC_Verification templates there exists a more strict version that will run CDC verification under the strict options to generate all possible issues with no assumptions about the design or user intent. CDC_VerificationBaseStrict and CDC_VerificationStrict templates can be enabled later in CDC verification process when issues reported by CDC_Verification/CDC_VerificationBase are addressed and number of violations are reasonable. Many options are used in strict versions of the templates, for instance option one_cross_per_dest is set to no which will force analysis of all crossings (by default only one source is analyzed for each destination flop; with this option crossings from all sources to a same destination is reported) but may cause significant increase in number of violations.

SpyGlass recognizes the most common type of synchronization schemes for asynchronous clock domain crossings and reset activation and deactivation schemes. For a detailed list of synchronizer schemes recognized by SpyGlass, see the SpyGlass Clock-Reset Rules reference manual.

Because we will be performing synchronization checks initially at the block level, the number of issues being reported will be more manageable than running verification on the full chip.

Most of these checks are based on the basic synchronization check, Clock_sync01 (e.g., Ac_cdc08 does the gray encoding check on all control buses crossing clock domains and synchronized by any technique other than multi-flop). Therefore, it is important to first resolve Clock_sync01 violations before investigating other checks. For this purpose a separate template is provided; CDC_VerificationBase includes only Clock_sync01/Clock_sync02. This template can be run and all Clock_sync01 violations should be fixed before the CDC_Verification template is invoked to check all aspects of CDC verification.

For complex blocks with more than five clocks where the number of crossings could be high, it is recommended to run the template in two modes:

- Stuctural: In this mode, in addition to invoking CDC_Verification template, you set the command line option fa_msgmode to none. With this option, SpyGlass will not perform functional checks and therefore the run time will be smaller. Here, you should particularly focus on reducing number of convergence violations (e.g. by adjusting CDC setup) as well as other violations where functional checks will be performed. With this preliminary filtering of structural violations you will significantly reduce functional checks run time.
- Structural and functional: In this mode, the CDC_Verification template can be used without any additional option allowing both structural and functional checks to be performed.

4.4.1.1 Synchronization check for metastability control

It is very important to close CDC setup before starting metastability verification. If clocks and/or set-case-analysis are not properly defined, many metastability violations may be reported which will take much more time to close. The metastability check is performed by rule Clock_sync01. If the number of violations is small, you can directly debug the violations using the schematic viewer and RTL cross-probing capabilities. Typically, for large blocks you may get a high number of violations. Many of these violations may be false violations due to configuration registers and other quasi-static signals that do not need to be synchronized. Section 4.4.2 describes various tools SpyGlass provides to reduce false violations and find real synchronization bugs faster.

4.4.1.2 Data hold checks across fast-to-slow clock domains

The data hold check in a fast-to-slow crossing is performed by rule Ac_cdc01. The Ac_cdc01a and Ac_cdc01b rules map to Clock_sync02 rule. The Ac_cdc01 rule set considers only fast-to-slow crossings and does not check for slow-to-fast or same frequency crossings reported by Clock_sync01 and Clock_sync02.

The Ac_cdc01 rule is only performed on a fast-to-slow clock crossing and not on the slow-to-fast crossings.

Section 4.4.3.1 provides further detail on how to debug functional checks such as the data-hold check.

4.4.1.3 Data Enable Sequencing verification

Ac_cdc04 rule reports 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 de-activated before the next data is loaded. There should be sufficient margin between the data change and the change. By default a margin of one clock edge is assumed which can be changed by using the rule parameter "fa_holdmargin". Ac_cdc04 is off by default in 3.9.3 release and is not part of the CDC templates.

Section 4.4.3.1 provides further detail on how to debug functional checks such as the data-hold or data enable sequencing checks.

4.4.1.4 Data correlation and race conditions - gray code checks

Convergence of signals such as control buses can cause major problems if not implemented using the 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. Ac_cdc08 ensures that any control bus crossing clock domains is gray encoded, while Clock_sync03a looks for converging signals originating from same source domain that are separately synchronized signals into a single destination domain and ensures that they are gray encoded.

For debug and analysis of the gray encoding check and other functional checks see section 4.4.3.1.

4.4.1.5 Complex synchronization schemes: FIFO recognition and verification

SpyGlass can automatically identify FIFOs. It is not always possible to recognize FIFOs automatically, FIFO recognition may produce following two results:

- Fully recognized FIFOs: This is the case if a FIFO's memory and pointers are identified
- Partially recognized FIFOs/Memory: A 2-dimentional memory or a lib/sglib memory identified by SpyGlass for which read/write pointers were not identified.

FIFO recognition will help CDC verification as follow:

- Metastability violations reduction (Clock_sync01 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 (Clock_sync01 violation). FIFO recognition will help in reducing such metastability violations (the crossing will be reported as properly synchronized by Clock_sync02 rule). You can control FIFO based Clock_sync01 filtering with enable_fifo option. If the option is set to "strict", only fully recognized FIFOs will contribute to Clock_sync01 violations reduction. If enable_fifo is set to "soft", partially recognized FIFOs/Memory will also lead to Clock_sync01 violations reduction. Reading data out of a memory is not necessarily safe and may be subject to metastability; so usage of "enable_fifo=soft" is not advised unless you are absolutely 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 a memory, and pointers counters can be identified. FIFOs can not 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. "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 detail of fifo constraint please refer to Clock Reset Rules Reference.

For debug and closure of FIFO and other functional checks, refer to section 4.4.3.1.

4.4.1.6 Complex synchronization schemes: handshake recognition and verification

Handshaking is a standard protocol adopted in many designs to transfer data across different clock domains. By enabling the –enable_handshake parameter, the Clock_sync02 rule identifies and reports the crossings that are synchronized using handshake. It also identifies the request and acknowledge signals corresponding to the data transfer, which are displayed in the schematic viewer. Ac_handshake01 and Ac_handshake02 perform functional checks on handshake circuits.

The Ac_handshake01 rule checks that the data signal is held stable for the period during which the request signal is active. The Ac_handshake02 rule checks that the request and acknowledge signals are alternating.

4.4.2 Noise reduction

One of the major challenges in CDC verification is to manage high number of violations. The methodology described in this document is tuned toward noise reduction by requiring specific setup and setup check steps. Typically, in large designs, skipping proper setup may lead to an order of magnitude higher number of violations. In addition to setup, SpyGlass provides a set of specific tools to filter false violations and get quickly to real design bugs. Typically, Clock_sync01 is generating the highest number of violations as it is checking all crossings in a design. In turn the number of Clock_sync01 violations will influence the quality, correctness, and noise level of other CDC checks such as Clock_sync03.

Following steps need to be followed to reduce noise:

1. Use automatic noise reduction capabilities:

   Many inter-block crossings may be synchronized using handshake or FIFO synchronizers. Make sure the options –enable_handshake and –enable_fifo are enabled. In large designs, handshakes and FIFOs are the main synchronization technique; if the above options are not enabled, a significantly higher number of Clock_sync01 violations may occur.

   Refer to section 4.4.1.5 and 4.4.1.6 for more details of the above options.

2. Use global parameters that fit your design style to reduce noise:

   Here are the most important parameters that can be used to set the tool for a given design style/project and which allow to reduce the number of violations[3].

   - allow_combo_logic

     Combinational logic on the crossing can create a glitch, which is harmless in a synchronous circuit, but if used in an asynchronous crossing may cause unwanted pulses, causing functional failures. By default, SpyGlass does not allow combinational logic to be present between synchronizers. However, this parameter can be enabled with the command line switch -allow_combo_logic=yes or by setting the value of the parameter to "yes" in the SpyGlass Design Environment. Please refer to the allow_combo_logic parameter in the Clock-Reset Rule Parameters section of the SpyGlass Clock-Reset Rules Reference for further details.

   - sync_reset

     While allow_combo_logic is a very high-risk practice, using a synchronous reset involving a single gate at the crossing or between synchronizer flops may be considered an acceptable practice. If this is an acceptable practice for you, then set the sync_reset parameter to “yes.” sync_reset option is turned on in the templates supporting this methodology.

[3] For further details on parameters for Clock_sync01 and other rules, refer to the Clock-Reset Rules Reference

- cdc_reduce_pessimism: Can be used to filter out violations as follow:
    - bbox: crossings involving black boxes at source or destination will be ignored. You can use this option once you have reviewed all black boxes and know crossings from/to the black boxes are synchronized
    - output_not_used: Crossings where the destination is blocked (output flop feeding a mux where the select of the mux is set to a constant not selecting the flop path) will be ignored
    - hanging_net: Crossing where the destination flop is not feeding any logic
    - mbit_macro: Rules Clock_sync03a and Clock_sync03b will ignore convergences where a multi-ouput macro (adder, comparator, etc.) exists in the paths of convergence. Default setting is on.
    - no_convergence_at_syncreset': Rules Clock_sync03a and Clock_sync03b will ignore convergences on synchronous resets specified by 'reset –sync' constraint. Default setting is on.
    - no_convergence_at_enable: Rules Clock_sync03a and Clock_sync03b will ignore convergences occurring on data and enable of a flip-flop. Default setting is on.
- clock_reduce_pessimism: Can be used to control clock domains propagation and consequently control CDC violations:
    - mux_sel: clocks will not be propagated through select of muxes
    - latch_en: clocks will not be propagated through enable of latches. Default setting
- pattern_match

  With the use of the pattern-matching feature, you can configure the clock-reset policy to recognize a user-specific synchronization scheme. For more information on pattern matching refer to application note AN_PatternMatch.pdf in the release tree under /doc.
- one_cross_per_dest:
  By default SpyGlass reports one violation for each destination even if there are multiple sources crossing clock domains into a same destination. You can set one_cross_per_dest=no to see all crossings for a given crossing's destination. However, by turning this option off number of violations may increase significantly

1. **Use local constraints to reduce noise:**
   Local constraints can be defined in the SGDC file to filter out specific unsynchronized crossings considered as acceptable by the designer. An example of such crossings may be configuration and other quasi-static registers which do not need synchronizers. The SpyGlass CDC solution provides a constraint, cdc_false_path, as the main CDC filtering capability. The cdc_false_path constraint allows you to specify the paths so that the Clock_sync01 and Clock_sync02 rules do not check for clock crossings. This reduces the number of violations reported on the 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
```

The first line will filter out the flop1-to-flop2 crossing from Clock_sync01 violations. The second constraint will eliminate all crossings from flops controlled by clk1 regardless their destination flops. Note that cdc_false_path impacts Clock_sync01 as well as other rules working on clock domain crossings such as Clock_sync03a, Clock_sync08 etc. The cdc_false_path constraint supports regular expressions. For further details, refer to the Clock-Reset Rules Reference. In addition to cdc_false_path, waivers and filters can be used to waive messages as a post-process operation on violations reported by SpyGlass. Waivers and filters are described in section 4.4.3.2.

2. **Interactive noise reduction:**
   From Clock_sync01 violations header you can access a spreadsheet view of all violations. In this spreadsheet you can sort or filter violations based on several criterions (e.g. source or destination clocks, reason of failures, etc.). Explore Clock_sync01 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 to be reported in subsequent runs.

4.4.3 Tips for Verification

4.4.3.1 Special considerations for functional checks

Functional verification of clock domain crossings is an important aspect of CDC verification. Many critical bugs causing chip spins are attributed to gray encoding failure, handshake or FIFO failure, and other types of functional problems in the clock domain crossings. When verifying these aspects of clock domain crossings you should be aware of the following situations.

4.4.3.1.1 Dealing with checks that do not complete

Functional checks are more CPU-intensive than structural checks. The outcomes of functional checks are as follow:

- Fail: For a CDC check that failed, SpyGlass provides a simulation trace that can be loaded in the waveform viewer by activating the violation and clicking on the waveform viewer icon (next to schematic viewer icon in the GUI).
- Pass: SpyGlass will produce a message only if fa_msgmode=pass or fa_msgmode=all. Passed checks are reported with severity info. You can also see these messages in the report file, accessible from the GUI pull-down menu Report->clock-reset->adv_cdc. You do not have to worry about these messages as they indicate proper CDC functionality proof.
- Partially analyzed: These are instances of CDC checks that are unconcluded. SpyGlass provides the number of cycles that have been explored during which no violation has been found. Similar to passed checks, partially analyzed checks are reported only if fa_msgmode contains “pa” or “all”; by default both failed and

partially analyzed results are reported. Messages for partially analyzed results are reported as warnings.

Both failed and partially analyzed checks require user attention as they may represent real design bugs. When dealing with functional checks that do not complete--that is, checks that are reported as partially analyzed--you can do the following:

- Increase the amount of time that SpyGlass spends on validating a single property. Currently, the default run time is set to 20 seconds per functional check. The parameter used to change the run time is called fa_atime.
- Change the engine selection for functional checks. This changes the way verification is done. SpyGlass provides an option, -fa_solvemethod, to invoke various engines performing functional verificaiton. This option takes 3 values (1, 2, 3) and depending on the design one or another may conclude the check.

4.4.3.1.2 Dealing with long run times

Functional analysis complexity increases with the number of asynchronous clocks in a design. It is recommended to perform functional verification where it is needed and avoid useless functional verification. This can be achieved by proper setup and structural noise reduction described in previous section. For instance, a synchronous reset will always converge with a data/control signal through a simple gate (e.g. AND gate). This type of convergence, although reported by Clock_sync03a, can be considered as safe as long as reset can be considered as static. By properly constraining synchronous resets (use reset constraint with –sync option) you can reduce number of Clock_sync03a violations due to synchronous reset convergence and consequently reduce formal verification run time by preventing formal verification of such convergences.

Formal verification is exhaustive and involves complex functional analysis of a design. Clock frequencies may greatly affect the complexity of functional analysis. T**o** understand how clock frequencies affect the functional analysis process, consider two clocks running with a 17ns period and a 13ns period, respectively. If the rising edges of the two clocks are aligned at time 0ns, then the next time the rising edges will again be aligned corresponds to 221ns (the LCM of two clock periods). This means that the design behaves asynchronously for 221ns. Any functional analysis process that would exploit the repetition (for proving a property, for example) would have to analyze the design at least for this period of time, which may correspond to many evaluations of logic in the design. We refer to this period as the Design Virtual Cycle.

In some cases if you run into long design runs, it may make more sense to modify the clock periods to reduce the LCM. Let's take an example.

Device A has two asynchronous clocks: clk_33 is 33MHz and clk_100MHz is 100MHz. If you specify these clock periods in the SGDC file, the LCM of the two clock periods is (33x100) 3300ns, which is quite large. If you specify the 100MHz clock in the SGDC as being 99MHz, then the Design Virtual Cycle has been reduced to 99 ns. Note that changing the clock frequency by this amount has impacted the behavior of the design and therefore the change shouldn't be considered unless absolutely necessary. If such a change has been introduced it has to be documented.

SpyGlass reports a Design Virtual Cycle in terms of the number of fastest clock cycles, as well as 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 since 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 10ns period.

4.4.3.1.3 Debugging functional CDC checks

In addition to RTL cross-probing and schematic highlights, failure of a functional check will generate a waveform indicating the circumstances of the failure. Once a violation is activated, click on the waveform-icon (close to the schematic icon) to activate the waveform viewer. Initially, a small set of signals are loaded in the waveform; these signals are considered to be a good starting point for debugging the waveform. You can right-click on a signal in the waveform viewer and select “fanin” from the pop-up menu to see the set of signals in the immediate vicinity of the selected signal for which a waveform is available. Select all or part of these signals and click on “OK” to load their waveform in the viewer. Note that you can cross-probe between the waveform viewer and the RTL-viewer.

4.4.3.1.4 Main reasons of false functional checks violations

Functional checks violations are always genuine under given constraints. To avoid false functional violations make sure design is properly constrained. Below are some of the most important constraints impacting outcome of functional checks:

1. Reset constraint: reset signals are used to initialize the design and they are generally disabled during functional checks. For instance a gray encoding check may fail due to a reset signal being asserted in the middle of a binary count. In order to prevent functional checks failure due to reset toggling, define the reset signal in a sgdc file (e.g. reset –name rst –value 0). If you want the reset to be treated as any other input during function check you can declare the reset as a “soft” reset (e.g. reset –name rst –value 0 –soft).
2. Initial state: SpyGlass identifies an initial state automatically and uses it as starting state for any functional checks. Functional checks may fail or pass depending on the initial state(s) used in functional verification. Always validate the initial state before investigating functional failures. Flop values for initial state as well as how the initial state is obtained by spyglass are provided in rule Ac_initstate01.

4.4.3.2 Waivers and Filters

Waivers and filters are the means of reducing the number of false violations being reported. For CDC violations, it is recommended that you use cdc_false_path constraints as described above rather than waivers. However, there are violations where cdc_false_path cannot be applied; in these cases waivers may be used. For example a Clock_info03a violation may be waived. Waivers can be used in two ways, either during pre-processing or during post-processing:

- **Applying waivers during pre-processing –** In case you do not want to view the CDC issues reported in a block that you don’t intend to analyze, apply a waiver on the block before analysis.

- **Applying waivers during post-processing** – As you analyze the reported violations, if a violation is determined to be acceptable, apply a waiver on it. For example, if a register is used for test-only purposes and a Clock_info03a violation is reported, and which you want to ignore, then apply a waiver.

4.4.4 Design Unit Integration and Chip-Level CDC Verification

As you progressively sign off each block for synchronization issues, you need to start integrating them. In most cases you still want to verify inter-block crossings or crossings into/from IP, but not crossings within IP. SpyGlass provides a way to perform CDC verification at the boundary of IP, but not within IP, by using the following constraint (in an SGDC file):

```
IP_block –name module1
```

The effect of this constraint is illustrated in the following figure (the pink flops belong to the top module; light blue flops are the boundary flops of the IP block; the green flop is IP block's internal flop):

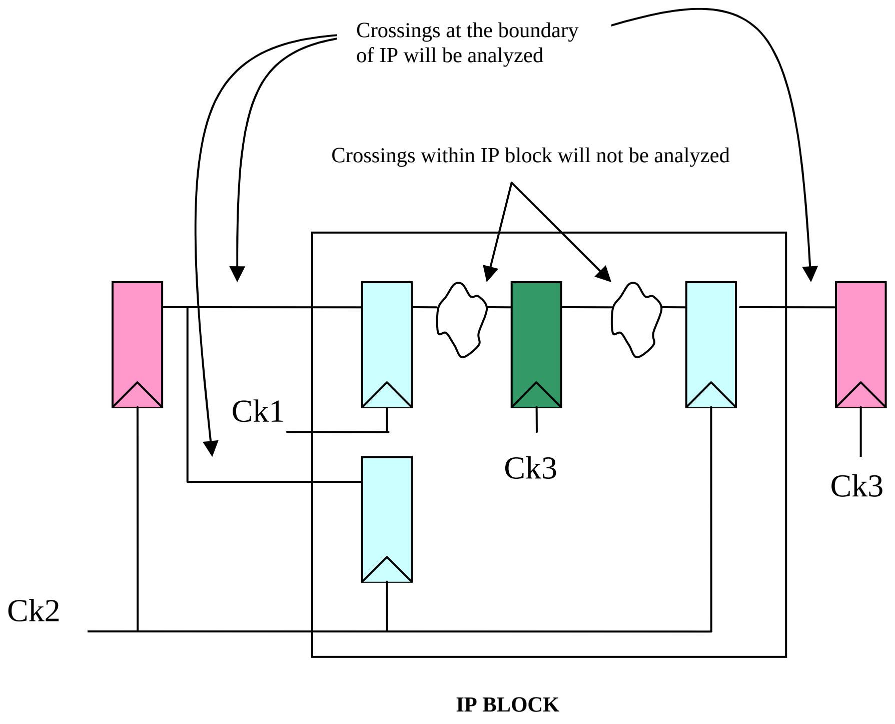

Crossings at the boundary of IP will be analyzed
Crossings within IP block will not be analyzed
Ck1
Ck3
Ck3
Ck2
IP BLOCK

Figure 7: Effect of IP Block Constraint

This constraint applies to all applicable CDC rules: Clock_sync01, Ac_cdc01, Ac_fifo01, etc. Bottom up CDC verification using IP_block will assure completeness of verification and can be considered as complete as top level, flat verification. This is true because checks validated at block level remain valid at top level. For instance, if a gray encoding check has passed (proved to be correct) at block level, it remains correct at higher level of hierarchies.

Note that the IP_block approach is different from black-boxing a module. When a module is black-boxed, no crossings are analyzed within, from or to the block. In rare cases where you want to black-box IP so that no crossings involving the IP block are reported, you can use the -stop parameter as follows:

```
spyglass –stop module1, module2,…
```

When block-level CDC verification is signed off, chip-level CDC verification can be carried out in the same way, except that all blocks that have been signed off will be IP_block'd. All other steps of setup, verification and debugging are the same.

For the command/files level detail of this step, please refer to the section 5.4.

4.5 Implementation Checks

In addition to the critical CDC issues covered in section 3 and addressed by the checks described in section 4.4, there are a number of other important CDC issues that must be considered. These are typically related to implementation of clocks, resets and crossings. SpyGlass includes a number of checks for these issues. The implementation checks can be run after CDC verification described in previous section has been completed. Implementation checks are carried out on the clocks and resets in the design. For example, the following checks are carried out on clocks:

- Are there latches used within the clock tree?
- Are bus bits being used as clocks?

On reset, the following check is carried out:

- Are unexpected gates found in the reset tree?

The violations when flagged are not too difficult to trace and debug, and hence this section is rather short.

Implementation checks are gathered in CDC_Implementation template. Following are main rules in this template:

Checks	**Description**
Clock_info05b	Clocks converging on a gate other than a mux
Clock_check01	Latches, tristate gates, or XOR/XNOR gates found in a clock tree
Clock_check02	High fanout clock nets not driven by specified placeholder cells
Clock_check03	Bus-bits used as clocks
Clock_check04	Mixed clock edges used in the design
Clock_check05	Deep ripple clock-dividers
Info_Case_Analysis	Provides schematic highlight of propagated values.
Reset_check01	(Verilog only) Reset signals that are not being used in the same mode as their respective pragma mode
Reset_check02	Latches, tristate signals, or XOR/XNOR gates found in a reset tree
Reset_check03	Reset signals that are being used at both levels to set or reset the flip-flops synchronously
Reset_check04	Reset signals that are being used both as asynchronous reset and synchronous reset

Checks	Description
Reset_check06	High fanout reset nets not driven by specified placeholder cells
Reset_check07	Reset signals driven by combinational logic

In addition, SpyGlass provides other, less commonly used, checks. For more information on these checks refer to Clock-Reset Rules reference.

4.6 Design Styles and Management

4.6.1 Handling Clock and Reset Nets That Propagate 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.

4.6.2 Handling PLLs, DLLs, Oscillators and Other Clock-Tree IPs

Typically, these blocks are analog, or at least have a non-synthesizable model. Section 4.2 describes a way of identifying any such black boxes and how to 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 /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.

If you want to perform a complete analysis through the I/O cells, you will need to find or create models for those cells. In many cases, you will find that the simulation models for the I/O cells are synthesizable, or UDP-based. In such cases, go to the spyglass_lc step (make sure to include the simulation models if the I/O models are UDP-based). Otherwise, you will need to manually create nearly equivalent synthesizable models.

With regard to memories, it is important to first 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 unsynthesizable modules (if the memory size exceeds mthresh). For all the unsynthesizable 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 flop per bit, causing memory explosion and performance issues. SpyGlass provides the following parameters to handle this problem:

1. Add the –mthresh parameter (works only for Verilog). 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).
2. Add the –handlememory parameter (works for both Verilog and VHDL). SpyGlass does an intelligent compression on memories before synthesizing, thereby allowing full analysis of the module/architecture. It works on most memories but will not work well on modules which initialize the full memory or on CAM-like structures.

Note: If both options are supplied, the -handlememory option overrides the –mthresh option.

Note: The second method is not recommended as it will impact functional checks and will interfere with FIFO recognition.

5 Step-by-Step Solution

5.1 Setup for CDC Verification

***Objective**:* Using SpyGlass, define clocks and resets and resolve black boxes as a preparation step for setup check and clock domain crossing verification.

1. Run CDC_Setup template:

- Select CDC_Setup template
- Enable Save Restore feature using –enable_save_restore command-line option
- Run SpyGlass as follows:

```
spyglass -template Clock-reset/Setup/CDC_Setup –f <command_files> <design_files>
```

2. Resolve black box clocks:

- Launch the SpyGlass GUI to analyze clock trees
- Look for black box clocks in Clock_info01 violations (search for "Black Box clock" in message window).
- For each black box on clock path take following actions:
    1. If a synthesizable module for the black box exists, then provide it to SpyGlass for the next run.
    2. Determine if the black box output reported as a clock is indeed a clock signal (requires design knowledge):
        a. If the answer is yes, do one of the following:
            i. If the module is stopped (using –stop), and you know which black box input should be routed to the specific black box output reported as a clock, then set assume_path between the input and the specific output of the black box; or
            ii. Define the signal as a clock for CDC verification.
        b. If the answer is no, remove the signal from the list of clocks for CDC verification.
    3. If the above two cannot solve the black-box for a clock; you can black box the higher level module, if this is not masking too much logic for CDC analysis, by using –stop, then provide assume_path on the "stopped" module which will be now accepted by SpyGlass.

Modified clock constraints and added constraints are saved in a new SGDC file for further pruning and CDC verification.

3. Restart at 1 if black boxes are solved:

- Rerun CDC_Setup template if new assume_path constraints have been introduced
- Rerun CDC_Setup template if new RTL code is provided for black boxes

4. Eliminate clock candidates that are not actual clocks (pruning):

- Review the list of candidate clocks in autoclocks.sgdc and remove signals that you know are not clocks (requires design knowledge). Possible candidates for filtering are:
    1. Test clocks (assuming you want to do CDC verification in functional mode)
    2. Control signals
- Add set_case_analysis constraints if the above analysis indicates that case-analysis should be considered during CDC verification (requires design knowledge).
- For cases where you need further analysis to understand the design before making a decision, take following steps:
    1. Launch SpyGlass GUI to analyze violations
    2. Go to specific Clock_info01 messages that you want to further analyze (use find context-menu from message tree window).
    3. Explore the message by cross-probing to RTL and tracing in the schematic viewer.

5. ***Assign clock domains and frequencies:***
    - For remaining clocks not filtered out by previous sections, provide domain and period information.
6. ***Eliminate reset candidates that are not actual resets:***
    - Open autoresets.sgdc generated by the previous SpyGlass run.
    - Review all resets and remove resets that you know are not resets (requires design knowledge).
    - For cases where you need further analysis to understand the design before making a decision, take following steps:
        1. Run SpyGlass in debug mode.
        2. Go to specific Reset_info01 messages that you want to further analyze (use the Find pop-up menu from message tree window).
        3. Explore the message by cross-probing to RTL and tracing in schematic viewer.
    - Save the pruned list of resets in an existing or new SpyGlass constraints file.

5.2 Setup Check

Objective: Using SpyGlass, check and further refine constraints created in the previous step.

The SpyGlass tool suite has capabilities to traverse the design and identify possible areas where set_case_analysis constraints are needed. SpyGlass will also identify any missing clock constraints.

1. ***Run CDC_Setup_Check template***
    - Select CDC_Setup_Check
    - Enable Save Restore feature using –enable_save_restore command-line option
    - Run SpyGlass

2. ***Resolve the* Clock_info03a *violations by solving all unconstrained flops.***
   - Activate Clock_info03a violation reporting a flop not being controlled by any clock defined in the constraint file
   - Analyze and trace the RTL code and/or the schematic to find a clock source that is driving the flop
   - Add clock constraints to your SGDC file.
   - Waive any remaining violations if you don't care about some flops (e.g. test-only flops). Note that such flops will be excluded from CDC analysis.
3. ***Resolve* Clock_info05a *violations by constraining muxes on the clock paths.***
   - Activate Clock_info05a violation.
   - Analyze the circuit (the MUX and its control) in the schematic window.
   - Use the set_case_analysis constraint to resolve the violation.
   - Use the Info_case_analysis to analyze the impact of set_case_analysis values in the circuit (activate the violation and open schematic).
4. ***Resolve* Clock_check07 *messages by removing redundant clock definitions in SGDC file.***
   - Activate Clock_check07 violation
   - Remove one of the two clocks, reported in the violation message, from the SGDC file.
5. ***Resolve* Ac_sanity04 *violations by resolving conflicting constraints.***
   - Activate Ac_sanity04 violation. This rule reports conflicting constraints in a file named OverConstrain.info in the $CWD/<vdb-name>_reports/clock-reset directory.
   - Perform root cause analysis of conflicting constraints. Often, these violations are due to conflicting set_case_analysis (e.g. an input of an AND gate set to 0 while its output is set to 1). Another possible cause of violations is a set_case_analysis set on a clock or reset path.
   - Fix conflicting settings (requires design knowledge)
6. **Rerun CDC_Setup_Check as needed:**
   - Rerun setup-check and proceed through the same steps (1 to 6), after one or multiple changes have been introduced in the constraint file or in the RTL code.
   - The process will stop when violation count for all the above checks has been reduced to 0.

5.3 CDC Verification

Objective: Verify clock domain crossings, fix issues, and sign off verification.

This step uses all the information gathered in the first three steps to perform the actual CDC verification. The output will be a list of possible clocking problems. This section shows how to step through the messages and determine which ones need to be fixed by changing the RTL and which can be filtered using SpyGlass options or the waiving/filtering mechanism. For the sake of completeness, the following four aspects of CDC verification are considered:

1. Metastability problems

2. Data hold problems in fast-to-slow crossings
3. Data correlation issues
4. Complex synchronizer verification (FIFO/handshake recognition and FIFO verification)

1. ***Choose the proper templates/ parameters to run the validation.***

   The most common options are:

   - sync_reset='yes' (allows synchronous resets in the metastability flops). Default in template
   - allow_combo_logic='no' (default; changing to yes will allow combinatorial logic in the data path between two clock domains.
   - enable_handshake='yes' (recognize handshake synchronization and reduce Clock_sync01 violations)
   - enable_fifo='strict' (recognize FIFO synchronization and reduce Clock_sync01 violations)
   - one_cross_per_dest=no (to make sure all crossings to a same destination is reported)

   You can chose to run strict template which would set the above options in a strict mode to analyze all crossings with no assumptions or run the template where the options are set with realistic/optimistic assumptions.

2. ***Run CDC_VerificationBase template:***

   - Select CDC_VerificationBase (or CDC_VerificationBaseStrict) template
   - Enable Save Restore feature using –enable_save_restore command-line option
   - Run SpyGlass

3. ***Resolve the* Clock_sync01 *violations***

   - If there are only a few messages, they can be examined in the GUI (proceed to next step). If there are a lot of messages and you need to sort, filter and categorize messages, right click on Clock_sync01 violations header and select "Open Spreadsheet" to explore violations in a spreadsheet. For individual crossings analysis proceed with next step.
   - Activate violation and bring up the Incremental Schematic to analyze violation.

     The source domain clock and the source flop will be highlighted in one color, the destination domain clock and the destination instance will be highlighted in another color and the data path crossing will be in third color.
   - Filter any messages using the cdc_false_path –to/-from constraint.

     cdc_false_path –from <source-flop> -through <internal-net> -to <destination-flop>
     For source and destination flops, output net name is expected.

     Look for mode or control-status registers that are static or quasi-static.

     Do not use waivers for Clock_sync01 violations. Use the cdc_false_path constraint. There are other rules that will honor this constraint and remove other potential violations.

- Fix any real bug found as a result of this exercise. Typically, add multi-flop synchronizers for control signals and enable flops for data synchronization with proper enable control circuitry.
- Rerun SpyGlass with newly added constraints and verify all Clock_sync01 violations are fixed.

4. ***Restart in step 2. until all Clock_sync01 are solved***
5. ***Run CDC_Verification template:***
   - Select CDC_Verification (or CDC_VerificationStrict) template
   - Enable Save Restore feature using –enable_save_restore command-line option
   - Run SpyGlass
6. ***Resolve the Ac_cdc08/Clock_sync03a violations for gray coding issues.***
   - Activate violation.
   - Bring up Incremental Schematic and waveform viewer.
   - Navigate waveform viewer to find the cause of failure.
   - Look for potential reset signals causing a violation. If you do not want reset to cause a functional failure, provide them in the constraint file.
   - Fix any real bug found during this exercise. Typically, gray encode correlated signals that are crossing clock domains.
7. ***Resolve the Ac_fifo01 violation indicating FIFO overflow or underflow.***
   - Activate violation.
   - Read the message carefully as it indicates whether it is an overflow or underflow.
   - Bring up the waveform viewer.
   - Investigate the cause of failure by navigating the waveform
   - Look for potential reset/clear signal causing such violation. Provide the reset/clear in the constraint file as a reset.
   - Look at the initial state values in the waveform viewer; if not correct, provide correct initial state in the constraints file or provide a VCD file from which an initial state can be loaded.
   - Fix any real bug found during this exercise. Often, gray encoding or data hold violation on FIFO pointers are causing such violation.
8. ***Restart at step 5 until all violations are resolved***
   - Rerun verification if design is changed.
   - Rerun verification if any constraints have changed or new constraints have been added.
   - Rerun setup if assume paths have changed.

5.4 Design unit integration and Chip-Level CDC verification:

- Add IP_block constraint in SGDC file for each block previously analyzed for CDC

- Start all steps, for setup, setup-check, and CDC verification at chip level. This can be integrated as part of block level CDC verification (considering the chip itself as a block).

5.5 Report review and CDC verification signoff

- Open CDC-report from the GUI pull-down menu Report->clock-reset->CDC-report for each block and review the content as follow
- Examine the assumptions; the 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 templates have been run and if there are any violations left unsolved. All such violations need to be justified and documented.

6 Conclusion

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

SpyGlass provides other important checks (e.g. Reset_sync01 for reset synchronization check) that are not detailed in this document. However, the CDC templates mentioned in this document covers them. For more details on these rules please refer to the SpyGlass Clock-Reset Rules reference.

SpyGlass also provides special checks (e.g. DeltaDelay01 to prevent simulation mismatch due to unbalanced clock trees as seen by simulators) that are not part of any template and are not described in this document. For more details on such rules please refer to the SpyGlass Clock-Reset Rules reference.

7 Appendix: CDC Template and Verification Level Matrix

The following table lists various CDC templates recommended to be used at different verification levels:

Stage	Template	Recommended CDC Templates for different verification levels		
		Block	IP	Chip/Sub-chip level
Stage	CDC_Setup	✓	✓	✓
	CDC_Setup_Check	✓	✓	✓
	CDC_Implementation	✓	✓	✓
	CDC_VerificationBase	✓	✓	✓
	CDC_VerificationBaseStrict	✓	✓	✓
	CDC_Verification	✓	✓	✓
Post-Synthesis/Pre-Layout	CDC_VerificationStrict	✓	✓	✓
	CDC_Setup	✓	✓	✓
	CDC_Setup_Check	✓	✓	✓
	CDC_Implementation	✓	✓	✓
	CDC_VerificationBase	✓	✓	✓
	CDC_VerificationBaseStrict	x	x	x
	CDC_Verification	x	x	x
Post Layout	CDC_VerificationStrict	x	x	x
	CDC_Setup	✓	✓	✓
	CDC_Setup_Check	✓	✓	✓
	CDC_Implementation	✓	✓	✓
	CDC_VerificationBase	✓	✓	✓
	CDC_VerificationBaseStrict	x	x	x
	CDC_Verification	x	x	x
	CDC_VerificationStrict	x	x	x

Guidelines:

- All CDC checks are first recommended to be run at RTL. Complex synchronization schemes like FIFO/HandShake should be verified at RTL only.
- FIFOs and handshakes may not be detected on Post-Synthesis and Post-Layout netlist designs.
- CDC_VerificationBaseStrict and CDC_VerificationStrict should be run during final RTL Handoff stages only.
- CDC_VerificationBaseStrict and CDC_VerificationStrict templates are not recommended to be run on Chip level directly without proper setup.
- For large designs, it is recommended to use the divide and conquer technique where you first perform CDC checks on blocks and then, using IP_Block constraint, run on full chip. (one block at a time)

- A large flat design should not be consumed by all templates. Therefore, it is recommended to first do CDC_VerificationBase followed by CDC_Verification. (one template at a time).

End of Document