cadence®

Jasper Clock Domain Crossing Verification App User Guide

Product Version 2023.09
September 2023

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

Preface

This manual is an introduction to the Jasper™ Clock Domain Crossing Verification (CDC) App. It provides setup and flow information to help you successfully integrate CDC sign-off into your verification efforts and is intended for verification engineers who need to exhaustively verify clock domain crossing.

This preface provides a general introduction to this manual and contains the following sections:

Cadence® Design Systems, Inc. prohibits the use of our software in a way that does not comply with our written guidelines and documentation.

About the CDC App

As the number of clock domains in a design explode due to dynamic power optimization, clock domain crossing issues must be addressed earlier in the verification flow. The CDC App provides formal and simulation-based solutions to this challenge and enables comprehensive CDC sign-off in an efficient, intuitive flow. It manages clock complexity by automatically inferring clock domain crossing intent from the design and comprehensively analyzing structural, functional, and convergence issues. It also provides an integrated debugging environment with access to advanced debugging options, including schematics, graphs, and waveform analysis with Visualize™.

How This Guide Is Organized

This guide is organized as follows:

- Chapter 1, "Preface"
  Provides a general introduction to this manual.
- Chapter 2, "CDC and RDC Concepts"
  Introduces key concepts, briefly describes common synchronizer schemes, and introduces CDC verification phases.

- Chapter 3, "CDC App Flow and GUI Orientation"
  Provides an overview of the CDC App general use flow.
- Chapter 4, "Capturing the CDC Intent"
  Discusses CDC configuration, constraints, and user-defined synchronizers.
- Chapter 5, "Running CDC Analysis"
  Provides description and procedures for running the structural analysis.
- Chapter 6, "Analyzing the Results"
  Describes the CDC App integrated debugging environment.
- Chapter 7, "Bottom-Up Hierarchical Flow"
  Describes the CDC App integrated debugging environment.
- Chapter 8, "CDC Violations"
  Provides a table of all CDC App violations with associated tags.
- Chapter 9, "Customizing the Rules File"
  Provides instructions for customizing the CDC rules file.
- Chapter 10, "Supported SDC Commands"
  Provides a complete list of supported SDC commands.

Conventions Used in Jasper Apps Documents

The following tables list conventions used in syntax and text.

Jasper Apps Syntax Conventions	
Convention	**Definition**
`Courier font`	Indicates text you will type on the command line.
`-underscore_separation`	
	Indicates a command switch. Switches are not case-sensitive.
`[ ]`	Indicates optional arguments. Do not type the square brackets.

`|`	Indicates a choice (a logical OR) among alternatives. Do not type the vertical bar.
`\`	The backslash character (\) at the end of a line indicates that the command you are entering continues on the next line.
`*`	Indicates the preceding argument appears zero or more times per command.
`+`	Indicates the preceding argument appears one or more times per command.
‘ ’	Indicates the enclosed character(s) should be explicitly included on the command line. Do not type the single quotation marks.
<*Italics*>	Indicates a command option that you will replace with a valid value. Do not type the angle brackets.
`( )`	Used as a grouping convention. Do not type the parentheses. For example, parentheses in the following syntax indicate that if you use the `-bbox` option, you will follow it with either the `0` or `1`. `[-bbox ( 0 | 1 )]`

Jasper Apps Text Conventions	
Convention	**Definition**
`Courier font`	This style indicates: • Text you will type in GUI fields • Commands and options • Filenames and paths • Code samples
Italics	User interface items such as button and field names.
Menu – Option	GUI command sequence; that is, click on a menu followed by an option. Example: *Help – Command Reference Manual* Meaning: Click on the *Help* menu and choose the option *Command Reference Manual*.

Blue or Black text	Hyperlinked cross-reference. When you view the PDF version of Jasper manuals from a computer screen, click on the blue or black text to view related information.

Customer Support

If you have a problem using the CDC App or the documentation, you can submit a customer case to Cadence Support or contact your local Field Application Engineer (FAE). When doing so, provide enough information about the problem so that it can be investigated efficiently. Describe the problem in full, give the version of the software you are using, and state the exact circumstances in which the problem occurs.

Being able to reproduce the internal error with a Tcl script and associated design files is the best way to ensure a quick fix by the Cadence Research and Development staff. The `capture_testcase` command is a way to bundle up most of what the staff needs to see. This command supports optional switches to add relevant files and screen shots. Refer to the command reference manual or type `help capture_testcase -gui` in the GUI.

If the internal error is not easily reproduced, you can gather and send the following information, which might be helpful.

- Log files – Jasper Apps automatically generates log files with the extension `.log`. The `jg.log` file contains all commands and tool messages. The `jg_console.log` file also includes messages related to license events, tool idle state changes, analysis session activity, forced timeout settings in effect, database auto-save, and tool exit. Hence, log file information identifies which step in the session triggered the issue.
- Machine and OS information – Use the `checkMachine.sh` script under the tool installation's `/bin` directory to output machine and OS information. This information documents the environment to use to reproduce the issue.
- File `hs_pid_errXXXX.log` – Jasper Apps often generates an `hs_pid_errXXXX.log` file along with a core file when an internal error is triggered. This file usually shows a stack trace of the relevant code involved in the internal error.

  ⚠ You must have write permission in the target directory for this file to be generated.

- Contents of `/etc/redhat-release` (Linux® only) – On Red Hat Linux systems, this file contains a text description of the Red Hat version information. This is usually much easier to interpret

for kernel information than the output of `uname -a`.

> ⚠ While not officially supported by Jasper Apps, the Fedora™ OS equivalent of this file is `/etc/fedora-release`.

This section contains the following subsections:

- Cases
- Using Cadence Online Learning and Support

Cases

Jasper Apps cases are your way of asking questions, getting solutions, and reporting problems. Unless told otherwise, Cadence Support staff will respond to your case. If Cadence support cannot answer your question, Cadence research and development personnel will get involved.

It is important to specify the severity level of the service request as accurately as possible. Cases have one of three levels of severity:

- Critical – You cannot proceed without a solution to the issue.
- Important – You can proceed, but you need a solution to the issue.
- Minor – You prefer to have a solution, but you can wait for it.

> ⓘ You can request Support to increase the severity level of an issue. Therefore, do not use Critical unless resolution of an issue is absolutely necessary and urgently required.

Using Cadence Online Learning and Support

Cadence Online Learning and Support provides a Jasper landing page with a CDC product tab and links to Rapid Adoption Kits (RAKs), troubleshooting articles, and videos.

In addition, Cadence encourages you to submit cases using Cadence Online Learning and Support. With Cadence Online Learning and Support you can also track your open cases. To use Cadence Online Learning and Support to submit a case:

1. If you do not yet have a Cadence Online Learning and Support account, go to (support.cadence.com) and follow the and click *Register Now* under the *New User* heading. You must provide a valid `HostID` for any Cadence product (for example, Jasper Apps). The

`HostID` is contained in the `SERVER` line of your Cadence product license file.

> ⚠ If you already have a Cadence Online Learning and Support account, then you only need to update your preferences to include a valid `HostID` for a Cadence product.

2. Log in to Cadence Online Learning and Support, and on the upper left side of the page click *Create Case* under the *Cases* heading. A form is presented for submission of your service request. Select *Jasper Apps* in the *Product* list box and click *Continue*. Follow the online instructions to complete the Service Request.

Creating Group Privileges in Cadence Online Learning and Support

Sometimes it is beneficial to view the cases of others on your project.

To create group privileges in Cadence Online Learning and Support:

1. Open a Cadence Online Learning and Support service request by clicking *Create Case* under the *Cases* heading.
2. Select *Jasper Apps* in the *Product* list box and click *Continue*.
3. Fill in the required fields in the form presented.
4. Explain in the *Stated Problem* text box that you want to create a group of users.
5. In the *People to notify upon Case creation* field, include the email addresses of the users you want to have group privileges.

   > ⚠ Each person receiving group privileges must have a Cadence Online Learning and Support account.

6. Click *Submit Case* to complete the case.

Visit http://www.cadence.com/support/Pages/default.aspx to learn more about Cadence Global Customer Support and the Support Offerings we provide. For more details about our support process, visit http://www.cadence.com/support/Pages/support_process.aspx.

This section lists locations for sources of information.

To learn more about features and commands, refer to the *Jasper Apps Command Reference Manual* and *Jasper Platform and Formal Property Verification App User Guide*. You can access these manuals from the tool's *Help* menu. Other resources available from the Jasper Apps *Help* menu include the following:

- Stand-alone PDF guides that describe the Visualize™ GUI, guide engine selection, and so forth.
- Searchable HTML Tcl command help

  ⚠ You can also type `help` on the command line for a list of all commands or use `help <command_name> -gui` for documentation on a specific command.

In addition, Cadence offers Cadence Online Learning and Support, a searchable knowledgebase that includes release version information, quick tip videos, answers to frequently asked questions, application notes, manuals, tutorials, and other resources.

New and updated resources are continually added to the site. This site requires a one-time registration. To register, go to Cadence Online Learning and Support and follow the new-user registration instructions.

1

CDC and RDC Concepts

With the addition of multiple functionalities, modern system-on-chip (SoC) designs are becoming more complex. To handle this complex logic, these chips typically contain multiple asynchronous clock domains, and signals are frequently transferred from one clock domain to another. In hardware, all clock domain crossing signals are often subject to metastability effects that can cause two fundamental issues as follows: 1) metastability propagation and 2) data loss and/or corruption. Traditional methods like RTL simulation or static timing analysis alone are not sufficient to verify that the data is transferred consistently and reliably across clock domains. As a result, many CDC-related bugs go undetected until the post-silicon verification stage, which necessitates costly re-spins.

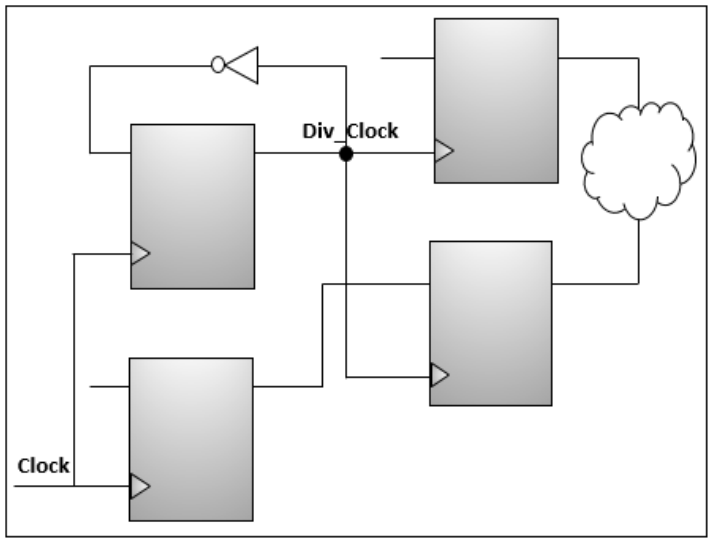
The Jasper CDC App is a combination of formal and simulation-based solutions that enables you to do comprehensive CDC sign-off. This chapter introduces you to some important CDC concepts to help you better understand the challenges associated with CDC verification. It includes the following sections:

- Clock Domains
- Metastability
- Glitch
- Data Coherency - Convergence
- Divergence
- Reset Domain Crossing (RDC)
- Common Synchronization Schemes

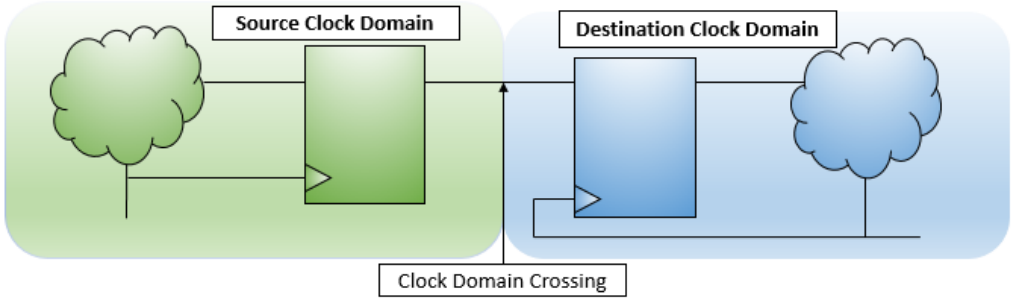
> ⓘ At various points, this user guide references CDC units. Units are always a design element, that is, flops, top inputs and outputs, black-boxed inputs and outputs, and signals with stopats.

Clock Domains

A clock domain is defined as that part of the design driven by either a single clock or clocks that have constant phase relationships. Synchronous clocks have constant phase relationships, but asynchronous clocks have variable phase and variable time relationships.

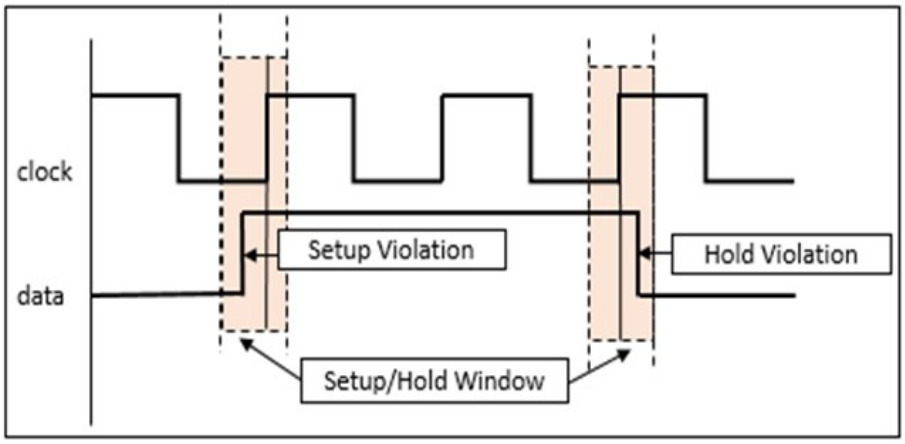


Clock domain crossings occur when a signal crosses between two asynchronous clock domains, and the signal crossing from one domain to another is referred to as a CDC pair.

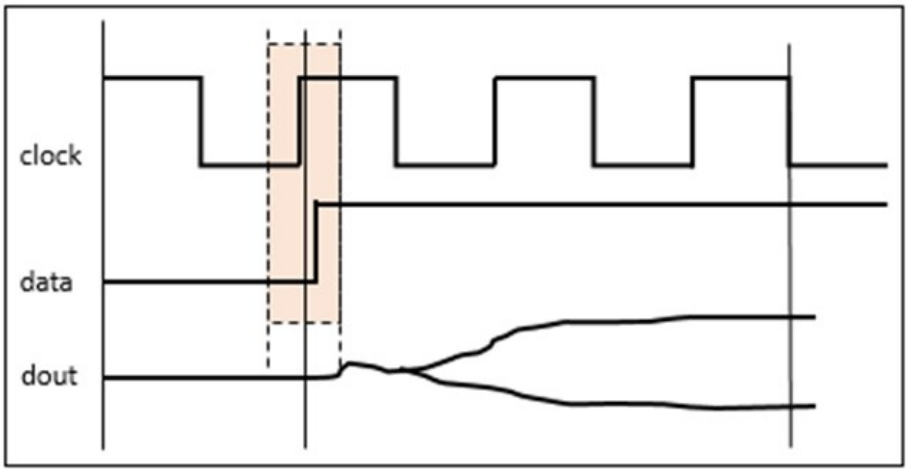


Metastability

Each flip-flop has a *setup* time, the time before the clock edge, and a *hold* time, the time after the clock edge, and data is expected to be ready by setup time and stable during hold time.



When setup and hold conditions are violated, the output of a flip-flop becomes unstable, and after an unpredictable delay, the value of the flip-flop can settle either way (`1` or `0`). This phenomenon is called metastability.



While you cannot prevent metastability in asynchronous designs, you can use synchronizers to prevent the forward propagation of metastable values, data loss or corruption caused by metastability, and data coherency issues related to convergence. See "Common Synchronization Schemes".

Glitch

Any logic in the CDC path can cause glitches and create functional errors downstream. For example, in the figure below, different delays in paths `T1` and `T2` cause a glitch, which causes `T3` to momentarily go high. This transition can be picked up by the sampling clock and cause the downstream logic to behave differently than expected (see Figure 2-1).

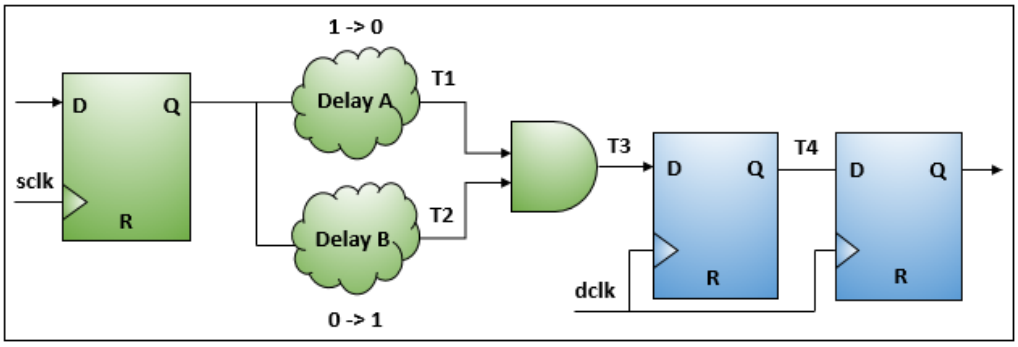
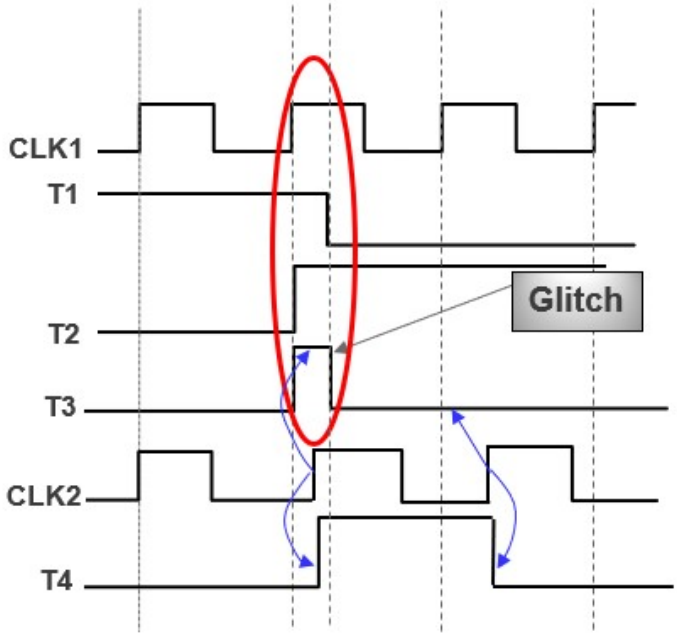


Figure 1.1: Effect of Combo Logic on CDC Path



Data Coherency – Convergence

Convergence is defined as a group of signals or different bits of the same signal coming from the same or different clock domains and converging into a combinatorial logic in the destination clock after synchronization. See the figures below for illustrations of both scenarios.

Figure 1.2: Convergence of Different Source Signals

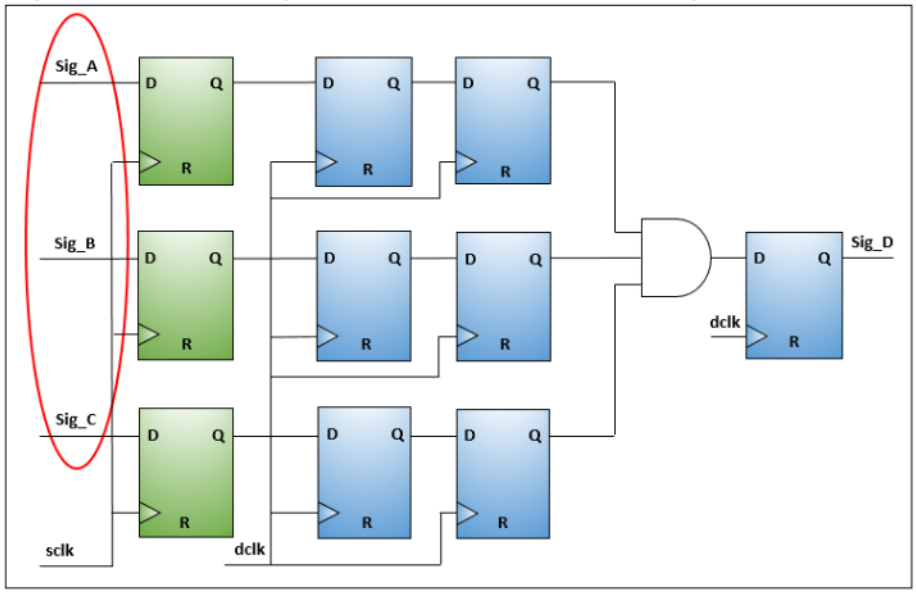
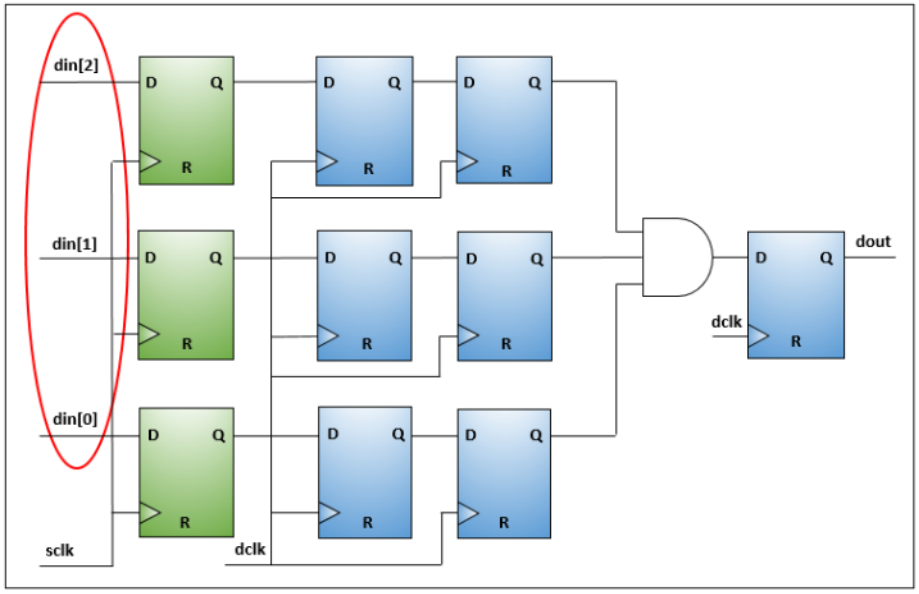


Figure 1.3: Convergence of Different Bits of the Same Source Signals



The tool reports convergence issues in terms of CDC groups. A CDC group consists of the CDC pairs that are converging into the combinatorial logic and the flop after it.

Divergence

A divergent logic to multiple synchronization paths might cause functional errors. For example, in the following figure, the propagation delay and different metastable settling times might cause `FSM1_en` and `FSM2_en` to start at different times. Avoid this type of structure by fanning out a single FSM enable after synchronization to both the FSMs.

Figure 1.4: Divergence of CDC Signal

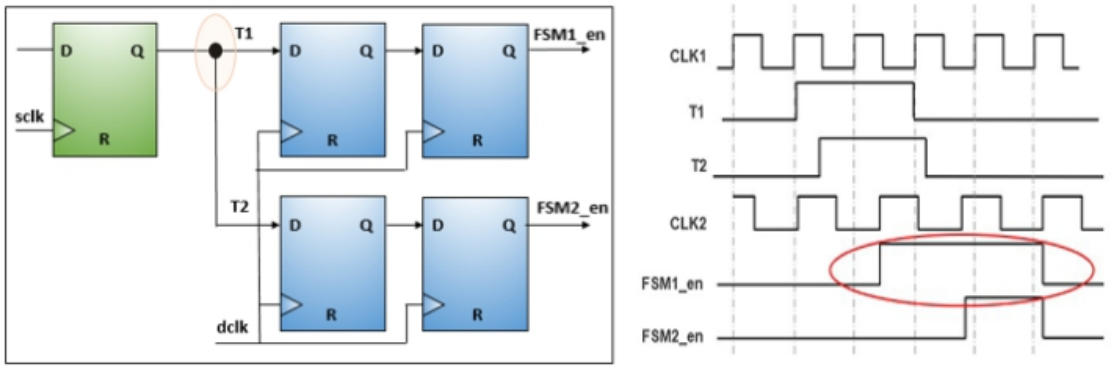
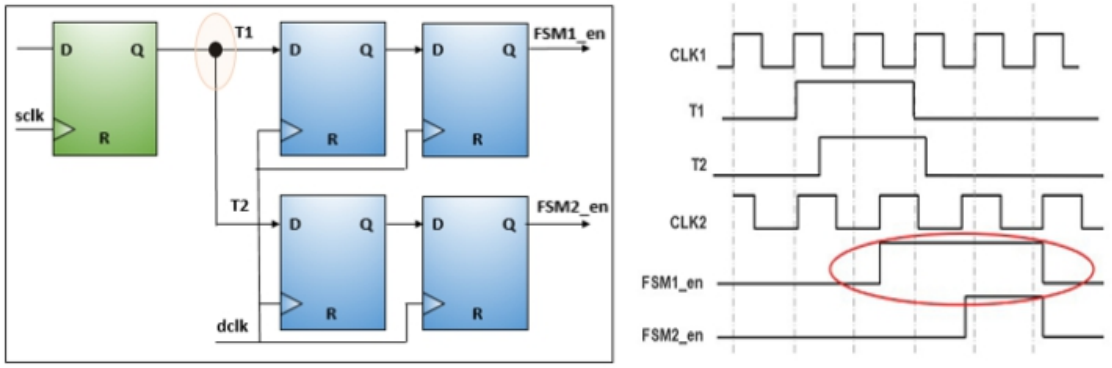


Figure 1.4: Divergence of CDC Signal



In addition, metastable signals are unstable signals that need proper synchronization. For example, in the figure below, `OUT1` and `OUT2` emerge at different clock edges because the settling and latching values of these metastable signals to the two flops happen at different times.

Figure 1.6: Divergence of Metastable Signal

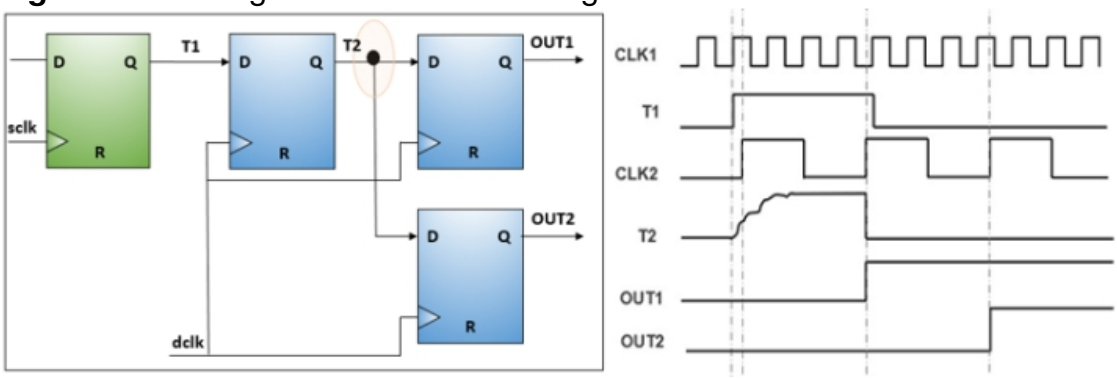
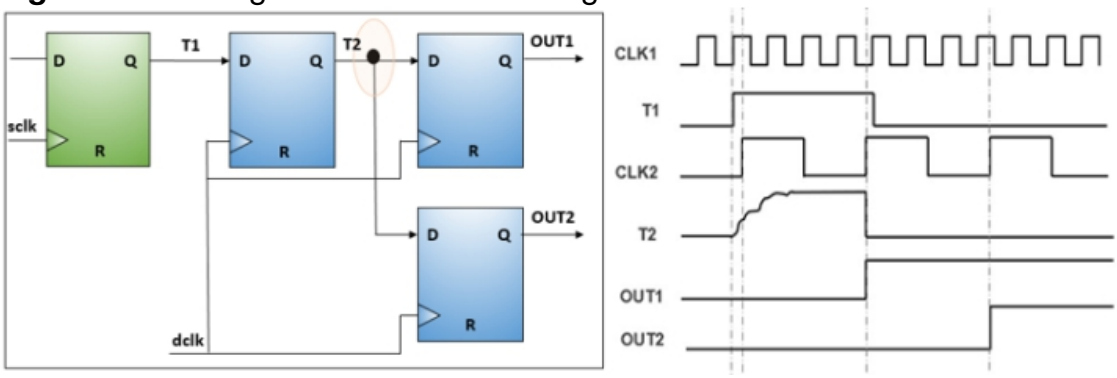


Figure 1.6: Divergence of Metastable Signal



Reset Domain Crossing (RDC)

Reset analysis analyzes the reset tree of the design and identifies two types of issues: 1) scenarios in which a reset signal crosses from one clock domain to another and 2) scenarios in which there is a reset domain crossing between the flops in the same clock domain. Both scenarios can cause metastability problems and should, therefore, be reported as reset violations.

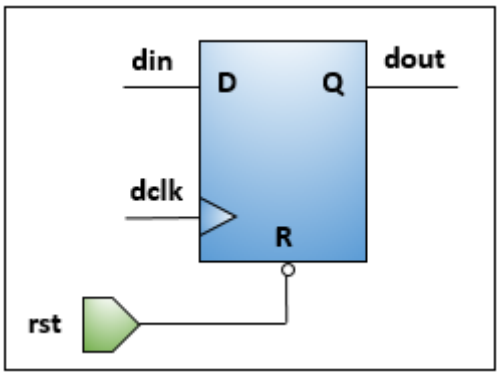
This section introduces the following RDC concepts:

- Reset Signal Crosses from One Clock Domain to Another
- RDC Between the Flops in the Same Clock Domain

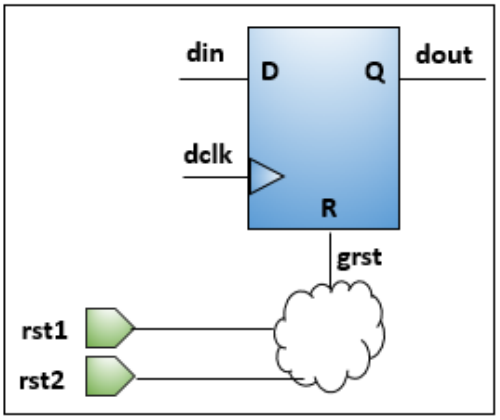
Reset Signal Crosses from One Clock Domain to Another

Disregarding buffers, the tool considers the signal connected to the flop's reset pin the reset signal of a flop. Consider the following cases:

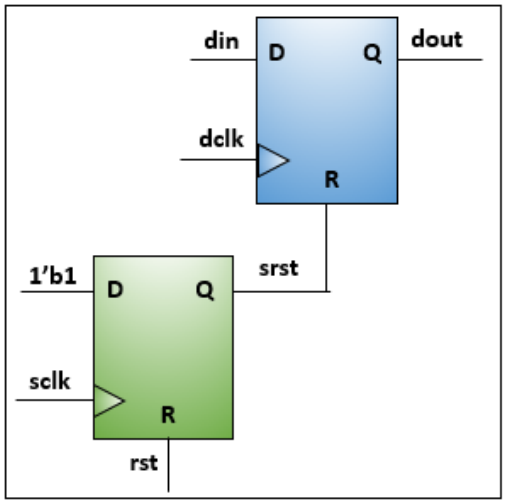
- Primary input in reset path – In this case, if rst is a primary input and is asynchronous to the destination clock domain `dclk`, `dout` is assigned its reset value when `rst` is asserted asynchronously; thus, the reset might be deasserted before the recovery time of the flop causing `dout` to become metastable. Hence, a reset pair is reported between `rst` and `dout`.



- Combinational logic in the reset path – In this case, CDC considers the output of the combinational logic `grstas` the reset signal and a reset pair is reported between `grst` and `dout`.



- Flop in the reset path – In this case, CDC considers the output of `srst` as the reset signal and the reset pair is reported between `srst` and `dout`.



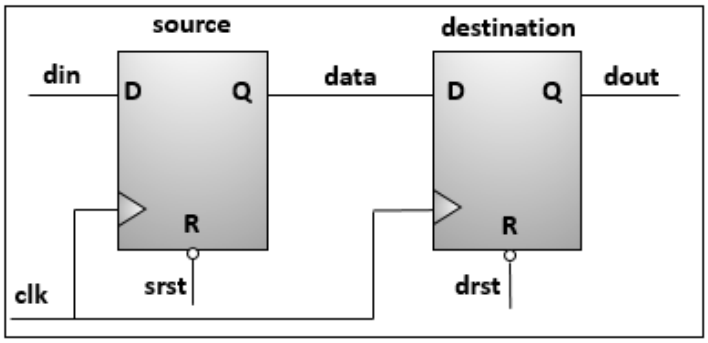
A clock crossing reset pair is reported between a flop and its asynchronous reset signal as follows.

- If the flop and its reset signal are in the same clock domain or the reset signal is the output of a valid reset synchronizer, the reset pair is reported as `Passed`.
- If the flop and its reset signal are in different clock domains, the reset pair shows `Failed` status.
- If the reset pair contains violations, it shows a `Failed` status.

⚠ For a list of reset violations, see *Reset Checks* in Chapter 7, "CDC Violations."

RDC Between the Flops in the Same Clock Domain

Metastability can be propagated from the reset path when the reset of the source register is different from the reset of the destination register even though the data path is in the same clock domain. Additionally, if the source flop has an asynchronous reset signal and the destination flop does not, the tool reports a reset violation.



During reset assertion (`srst`), if the reset value of the destination flop is different from the value in the source flop output (`data`), the destination flop can become metastable if it is not being reset at the same time. To address this problem, the design should always use the same reset in a clock domain or both resets must be asserted simultaneously. The order of reset assertion can be specified using reset order analysis (see "Reset Order Analysis").

Common Synchronization Schemes

While you cannot prevent metastability in asynchronous designs, you can use synchronizers to prevent the forward propagation of metastable values and data loss or corruption caused by metastability. This section provides a brief introduction to commonly used synchronization schemes and includes the following sections:

- Control-Based Synchronization Schemes
- Data Synchronization Schemes
- Reset Synchronization Schemes

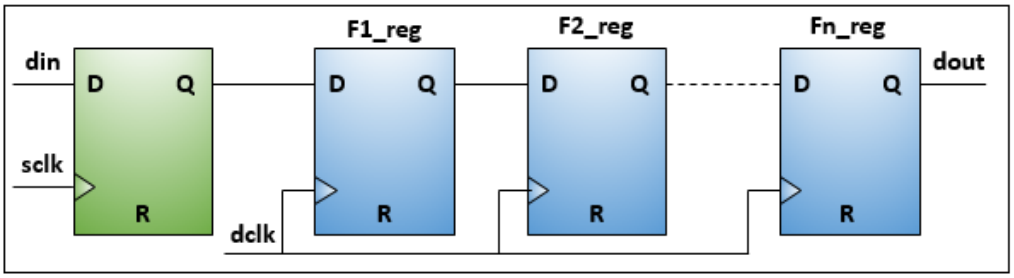
Control-Based Synchronization Schemes

Control-based synchronizers include the following:

- NDFF
- PULSE
- EDGE

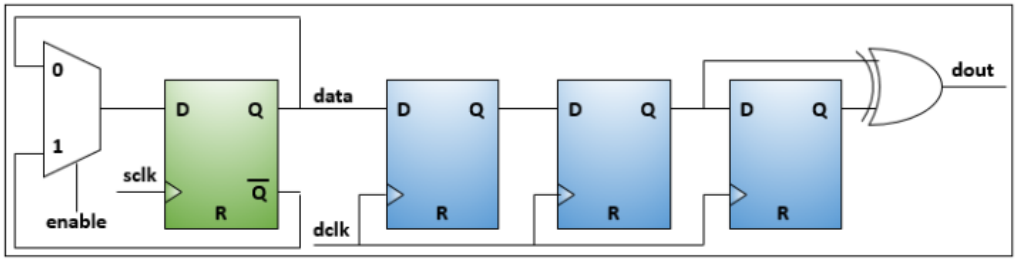
NDFF

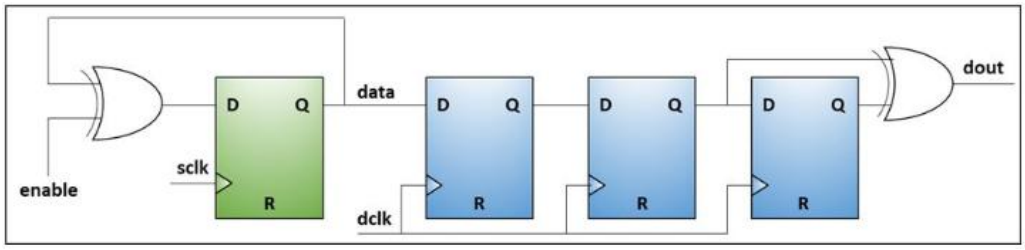
NDFF synchronizers check single-bit crossovers like control paths. The simplest NDFF synchronizers use 2 DFFs, but you can add more flops to increase the Mean Time Between Failure (MTBF).



PULSE

Pulse synchronizers are used to synchronize pulses entering the destination clock domain from the source clock domain. The basic function of a pulse synchronizer is to take a single clock-wide pulse from the source clock domain and create a single clock-wide pulse in the destination domain. The tool supports both single-bit and multi-bit signals in Pulse schemes.





EDGE

Edge synchronizers detect the rising or falling edge of the input and generate a one-clock-cycle-wide active high or active low pulse to the output. The tool supports both single-bit and multi-bit signals in Edge schemes.

Figure 1.8: Rising Edge Active High

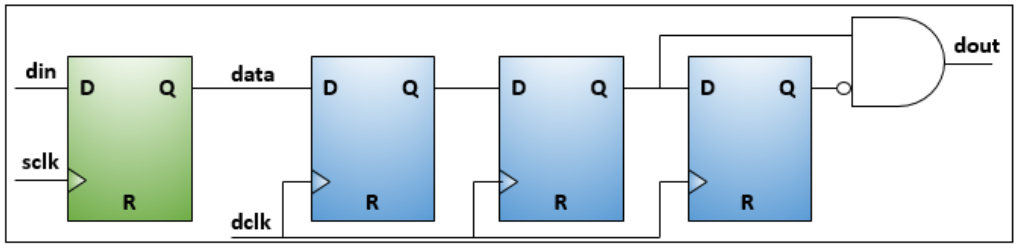


Figure 1.9: Rising Edge Active Low

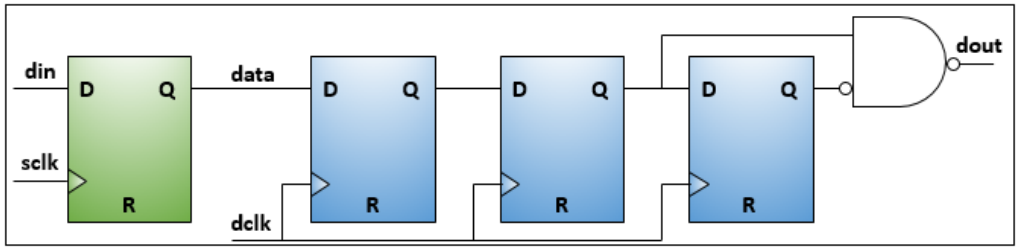


Figure 1.10: Falling Edge Active High

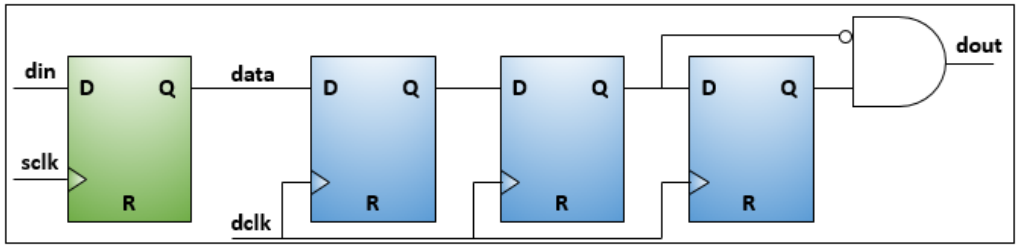
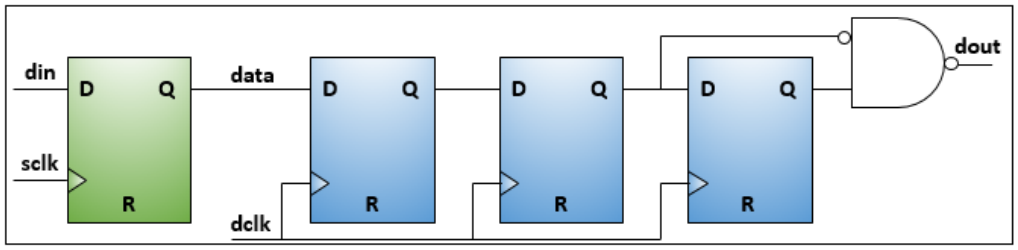


Figure 1.11: Falling Edge Active Low



Data Synchronization Schemes

Data synchronizers include the following:

- MUX_NDFF and MUX_PULSE
- Handshake
- FIFO
- Synchronization Enabler
- Glitch Protector (AND-Based Synchronizer)
- NDFF_BUS

MUX_NDFF and MUX_PULSE

MUX synchronizers are typically used for data paths. Open-loop synchronization ensures that data is captured without acknowledgment and when the MUX enable is active. MUX schemes use a synchronized control signal as the select line of the MUX. The MUX_NDFF scheme uses an NDFF to synchronize the control signal and the MUX_PULSE scheme uses a pulse to synchronize the control signal. This control signal can be the selector pin of a MUX gate or can be the enable condition of a clock gating as shown in the following figures.

Figure 1.12: MUX_NDFF

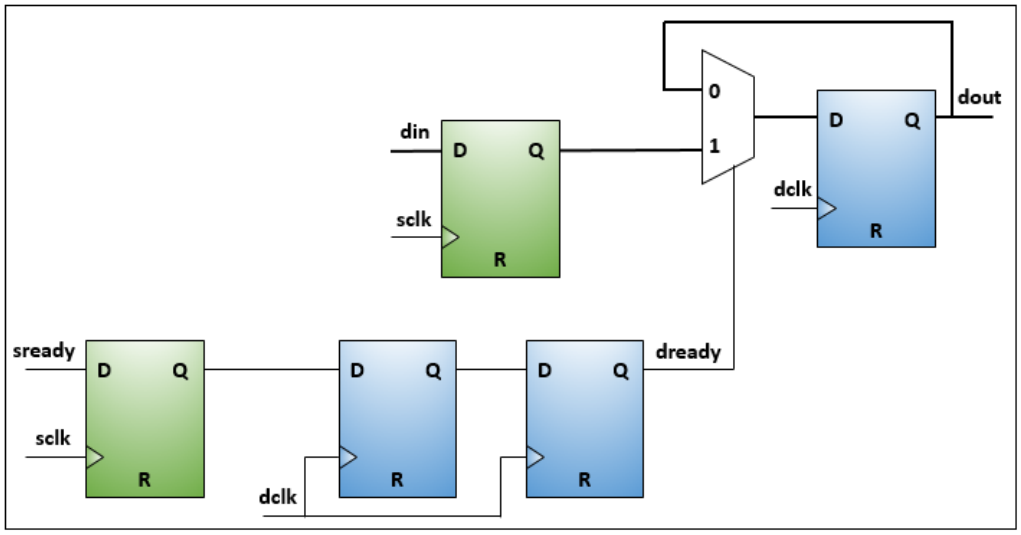
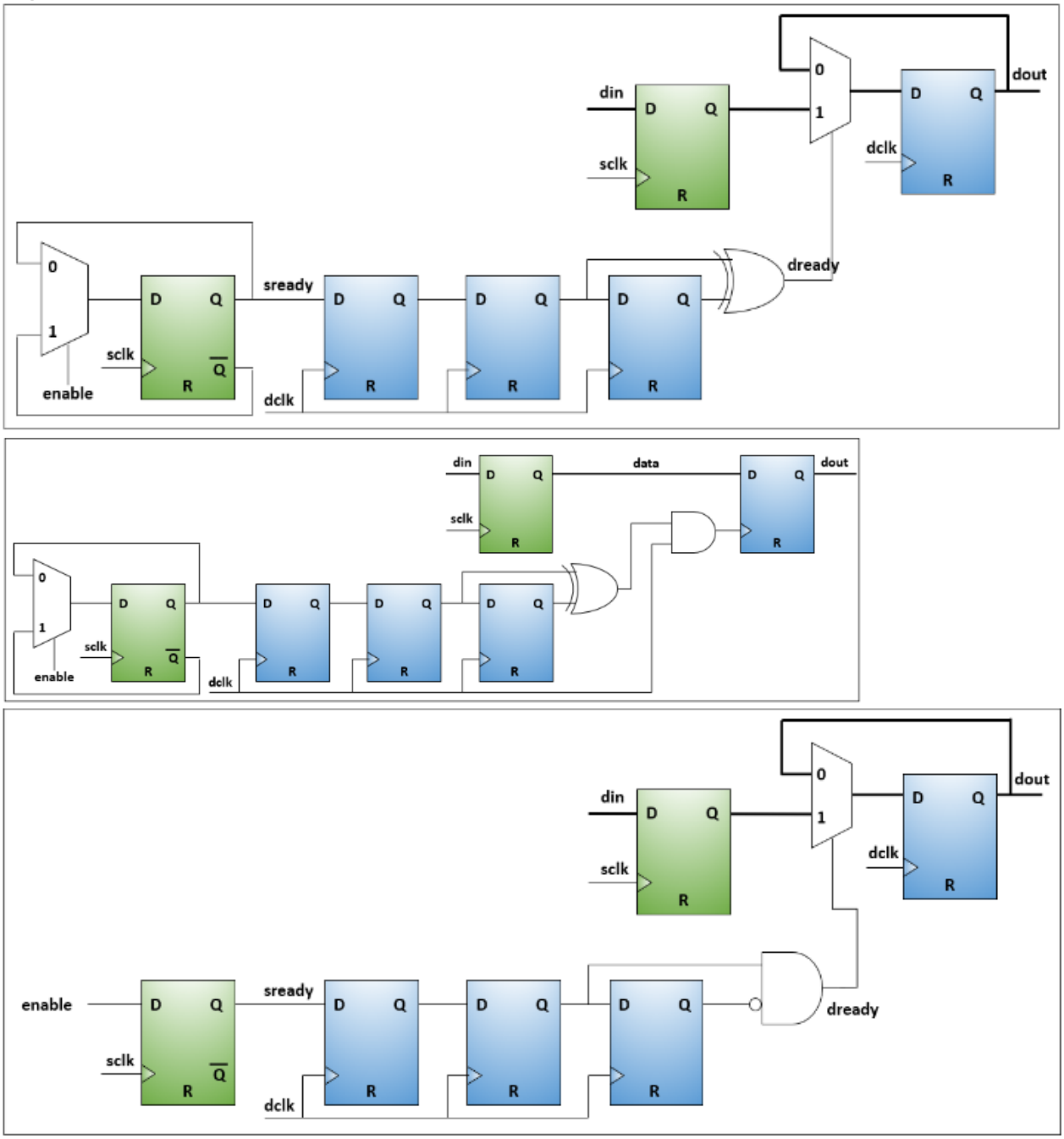


Figure 1.13: MUX_PULSE



Handshake

Handshake synchronization is a closed-loop synchronization method that requires acknowledgment of receipt of the signal that crosses the CDC boundary.

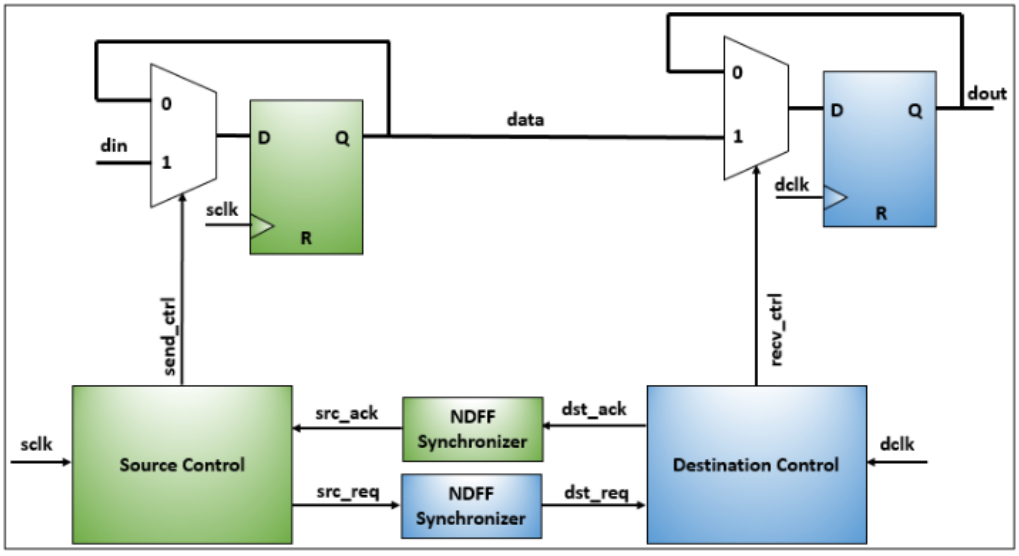
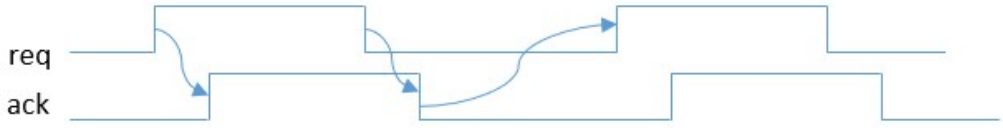


Figure 1.14:
Phase Handshake Protocol

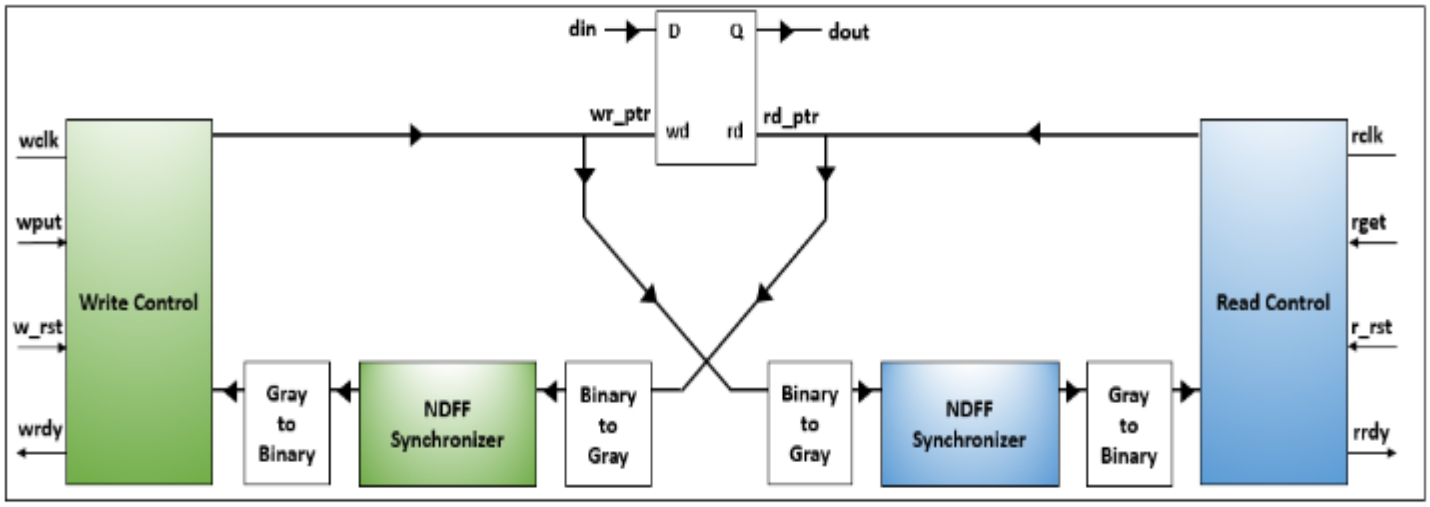


With this type of synchronization, the source and destination modules use a simple request-acknowledgment protocol as follows (also see Figure 2-12):

- The sender should continue to assert the `req` signal until `ack` is received at the sender's end.
- The sender should not assert a new request `req` until `ack` for the previous transfer has been de-asserted.

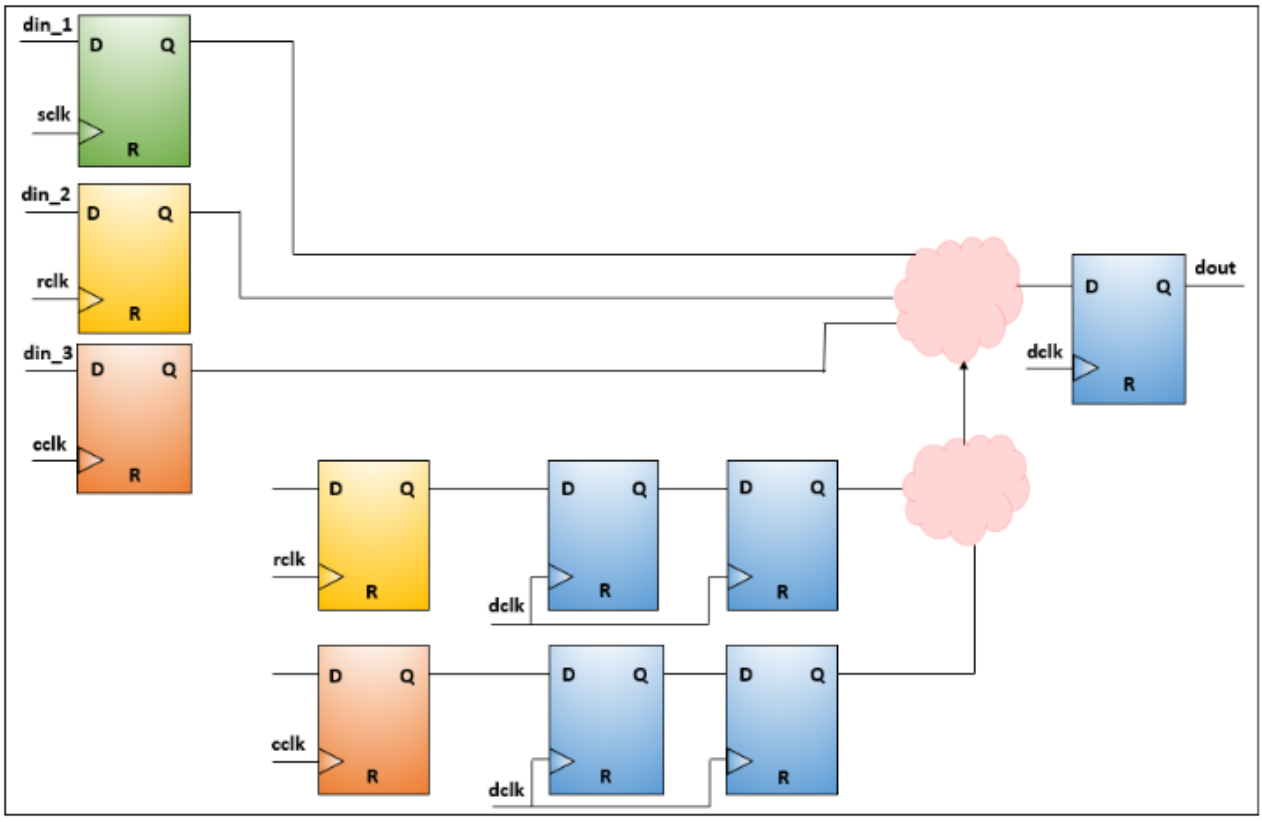
FIFO

FIFO synchronizers are used to transfer data between clock domains. Data is written into the FIFO from the source clock domain, which never writes when the FIFO is full, and read from the FIFO in the destination clock domain, which never reads when the FIFO is empty. The gray-coded read and write pointers are passed to the alternate clock domain to generate full and empty status flags.



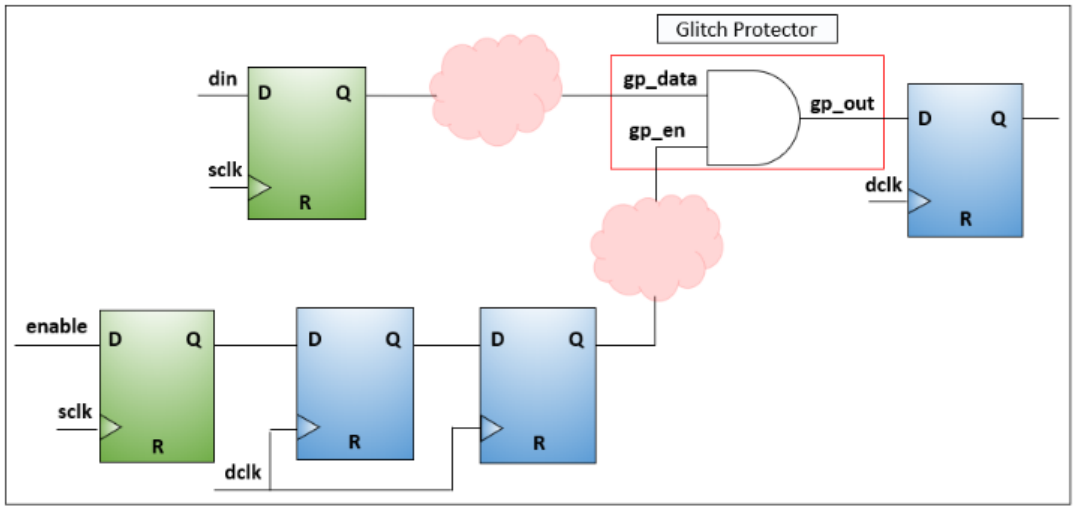
Synchronization Enabler

Use the synchronization enabler scheme to synchronize a data path between one or more source domains and a specified destination domain. With this scheme, you inform the tool of the signal that controls the data going from one or more domains to another.



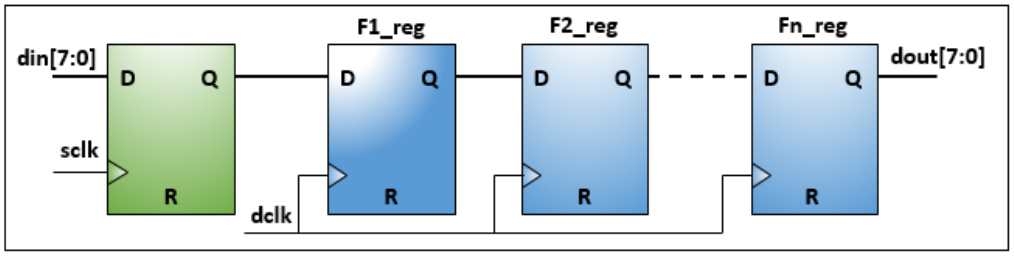
Glitch Protector (AND-Based Synchronizer)

Use glitch protector schemes to add a custom module that acts as a data synchronizer. You must provide the glitch protector module name before automatic scheme detection.



NDFF_BUS

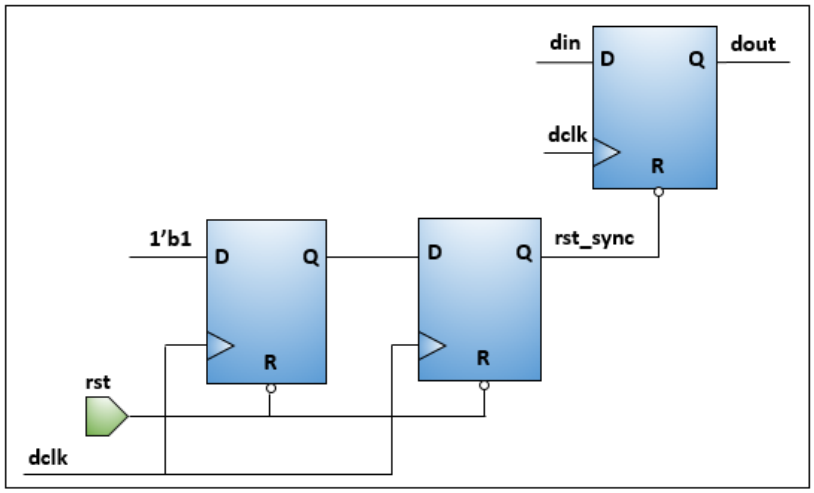
Use NDFF_BUS schemes to synchronize a wide signal from the source domain to the destination domain. Each bit of the wide signal is synchronized individually by the NDFF_BUS scheme.



Reset Synchronization Schemes

In standard reset synchronization, the reset assertion is asynchronous to the clock, but when the reset signal is de-asserted, it happens synchronously with regard to the synchronizer clock. Thus, when reset is asserted, the output of the reset synchronizer assumes the reset value asynchronously, resetting the target flops. When the reset is de-asserted, the output of the synchronizer assumes the input value synchronously after two active clock edges, de-asserting the target flops' resets.

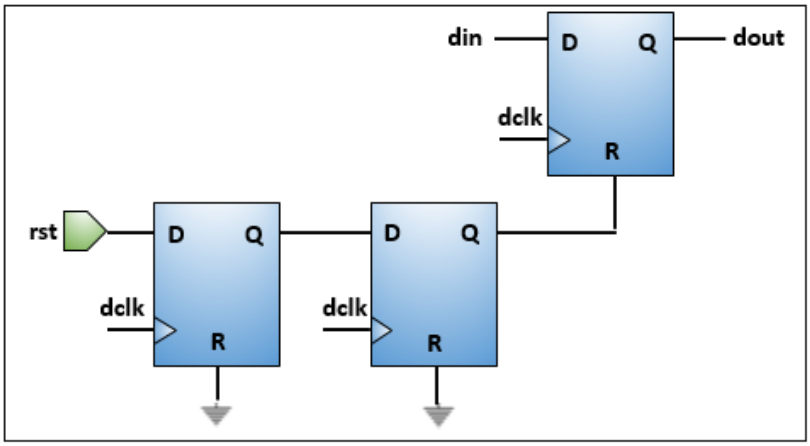
Figure 1.15: Standard Reset Synchronizer



In the above example, `rst` is an asynchronous reset signal driving the reset input of a synchronizer clocked by `dclk`. The output of the reset synchronizer, `rst_sync`, is asserted immediately when `rst` is asserted. When `rst` is de-asserted, `rst_sync` is de-asserted after two cycles of `dclk`. This allows the target flops two additional cycles to come out of reset.

In direct reset synchronization, the reset signal drives the data pin of the flop and flops of the synchronizer are non-resettable. Both assertion and de-assertion happen synchronously with respect to the clock.

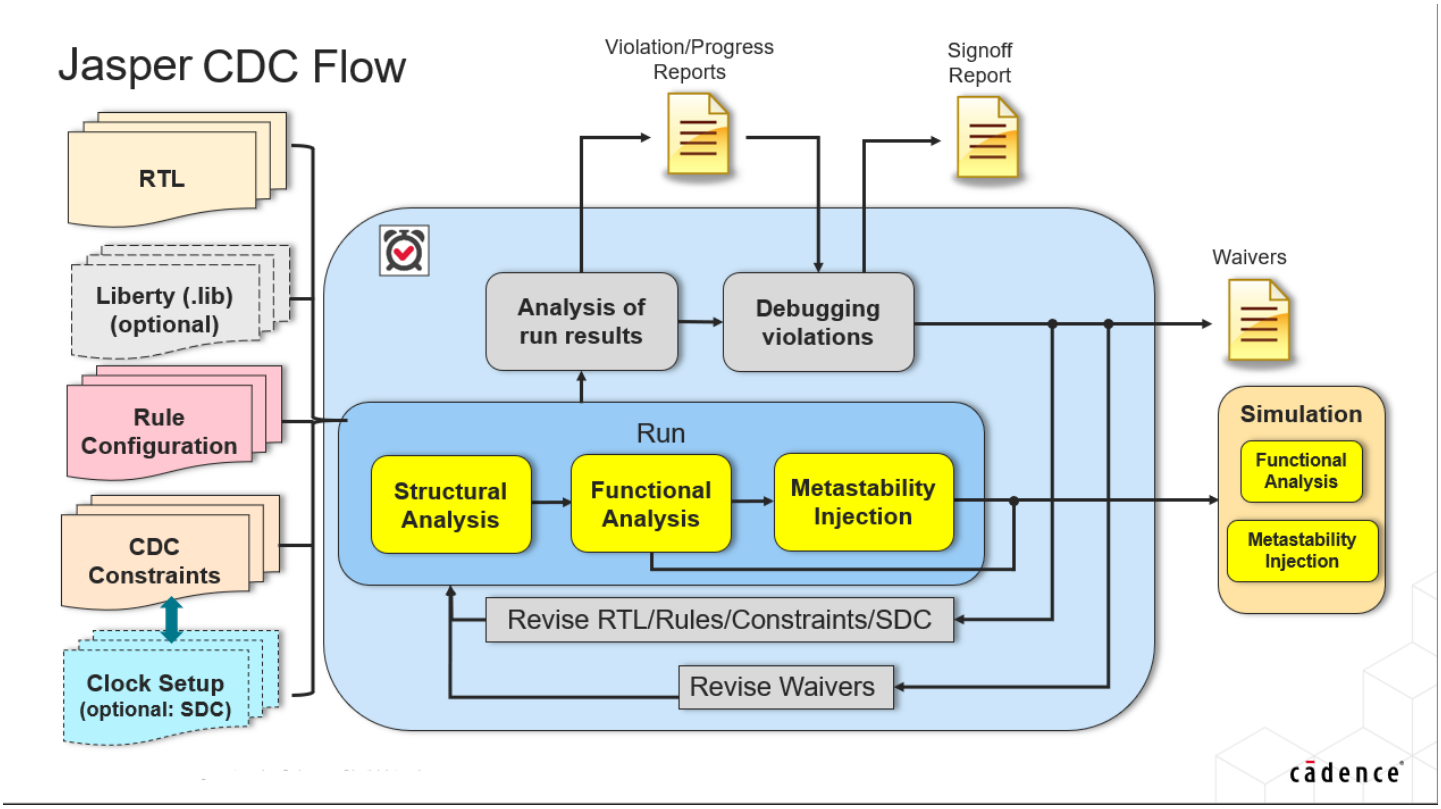
Figure 1.16: Direct Reset Synchronizer



For more information on common synchronization schemes, click the following link to navigate to the Jasper landing page on Cadence Online Learning and Support: https://support.cadence.com/jasper. From there, click the CDC button, scroll down, and follow the "Defining User-Defined and Custom Synchronizers" link under "Application Notes".

2

CDC App Flow and GUI Orientation



The above diagram shows the Jasper CDC flow. The following sections describe the inputs required and different execution phases during a complete CDC analysis run.

Inputs Required

Jasper requires the following inputs to run a CDC analysis:

- RTL – This is the design on which CDC analysis needs to run.
- Liberty files – This is an optional input. If available, you can load Liberty files for standard cells or hard macros like PHY or memory models.

- Rule configuration – A rule file containing the rule and parameter configurations. The rule settings determine which checks are performed during CDC analysis.
- CDC constraints – This includes clock, reset definitions, constraints on signals (constant, static, mutually exclusive, gray coded, and so forth), and user-defined synchronizers.
- SDC – This is the most common way to declare clock specifications, clock relationships, and clock association for ports. However, this can be an optional input if you have declared all specifications using native Jasper commands.

CDC analysis is divided into three different phases as follows:

Structural Analysis

The structural analysis phase includes running the following checks:

- Clock/Reset tree checks
- CDC synchronization checks
- Convergence/glitch checks
- Reset synchronization checks
- Reset domain crossing checks

Once the structural analysis is done, the tool reports violations indicating issues with the CDC structures in the design. You can analyze the structural analysis results and debug the violations. Based on your analysis, you can fix the RTL or the CDC setup, or waive the violation if the reported scenario is safe. Once the structural analysis is complete and the design is structurally clean, the next step is to run the functional CDC analysis to guarantee that the CDC structures in the design are functioning as per expectation.

Functional Analysis

The functional analysis phase includes the following:

- Generation and proof of protocol checks
- Validation of signal configuration and conditional waivers
- Automatic waivers

The structural and functional analysis guarantees that the CDC structures are working as expected.

However, analyzing the combined effect of metastability through different synchronized paths on the design can only be achieved by doing a metastability aware verification of the design.

Metastability Analysis

The metastability aware analysis phase includes the following:

- Modeling and Injection of metastability effect
- Proving user-defined properties in presence of metastability effects in a formal environment

Link to Simulation

The functional CDC analysis or metastability-aware verification can also be done in simulation. CDC helps you to perform the following CDC verification in simulation:

- Functional CDC verification
  - Export protocol checks, signal configurations checks, and conditional waivers checks
  - Run these checks in simulation
- Metastability-aware simulation
  - Export timing violation monitors and metastability injection modules to simulation
  - Include the Jasper generated monitors and injection modules in simulation and make the simulation environment metastability aware
  - Randomly inject metastability effect during simulation
  - Validate existing simulation test cases in presence of metastability effect

Sample Run File

A sample Tcl script that illustrates the general flow and includes keys commands follows. For complete help, see the *Jasper Apps Command Reference Manual* (*Help – Command Reference Manual*).

```
clear -all

#Analyzing RTL
analyze -verilog sample.v

#Loading Liberty File
```

```
liberty -load sample.lib

## Loading Rule File
config_rtlds -rule -load customized_rule_file.def

#Elaborating Top module
elaborate -top top -bbox_m { ip_1 ip_2}

#Loading SDC
read_sdc sample.sdc

#Clock Declaration (Optional if SDC is not used)
#check_cdc -clock_domain -virtual_clock vclk
#clock clk_1
#clock clk_2
#clock vclk

#Clock Rating (Optional if SDC is not used)
#config_rtlds -port {ip_1.data} -clock clk_1
#config_rtlds -port {ip_1.dout} -clock vclk
#config_rtlds -port {ip_2.dout} -clock clk_2

#Reset Declaration
config_rtlds -reset -async rstn1 -polarity low
config_rtlds -reset -async rstn2 -polarity low
config_rtlds -reset -async rstn3 -polarity low
config_rtlds -reset -async rstn4 -polarity low
config_rtlds -reset -synchronized rstn5 -clock clk_1 -polarity low

#Assumptions and Assertions
assume { @(posedge clk_2) $changed(inp3_reg) |-> $stable(inp3_reg)[*3]}

#Adding Signal Configuration
config_rtlds -signal -constant {{inp1_reg 1'b1}}
config_rtlds -signal -static {{inp2_reg}} -condition {en == 1'b1} -verify

#Setting the CDC Parameters (optional if custom rule file is not loaded)
#config_rtlds -rule -parameter { autowaive_rdc_with_ndff = false } -tag RDC_RS_DFRS
#config_rtlds -rule parameter {strict_ndff_detection = true}

#Reset Rating
config_rtlds -port {ip_1.data} -reset rstn4

## Joining clock domains
config_rtlds -clock -group -sync {vclk clk_1} -name CLK_GRP

#Find Clock Domains
check_cdc -clock_domain -find

#Find CDC pairs
check_cdc -pair -find

#Adding User-defined Schemes
check_cdc -scheme -add sync_enabler -map {{enable en} {srcdomainlist clk_1 clk_2} {dstdomain
clk_3}}
```

```
clk_sff

#Find Schemes
check_cdc -scheme -find

#Find Convergence
check_cdc -group -find

#Reset Analysis
check_cdc -reset -find

#Functional Checks
check_cdc -protocol_check -generate

check_cdc -protocol_check -prove
check_cdc -protocol_check -export -file protocol_checks.svp -force

#Proving Signal Configuration
config_rtlds -signal -prove

#Validate Waivers
check_cdc -waiver -generate
check_cdc -waiver -prove

#Metastability Injection in Formal
check_cdc -metastability -inject
check_cdc -metastability -prove

#Report Generation
file mkdir $PROJ_DIR/reports
check_cdc -report pairs -file $PROJ_DIR/reports/pairs.csv -force
check_cdc -report violations -file $PROJ_DIR/reports/violations.csv -force
check_cdc -report rules -file $PROJ_DIR/reports/rules.csv -force
```

CDC App GUI Orientation

This section provides an overview of the CDC App GUI. Use Figure 3-1, and Figure 3-2, and Table 3-1 to familiarize yourself with the components of the main window.

Figure 2.1: CDC App Main Window

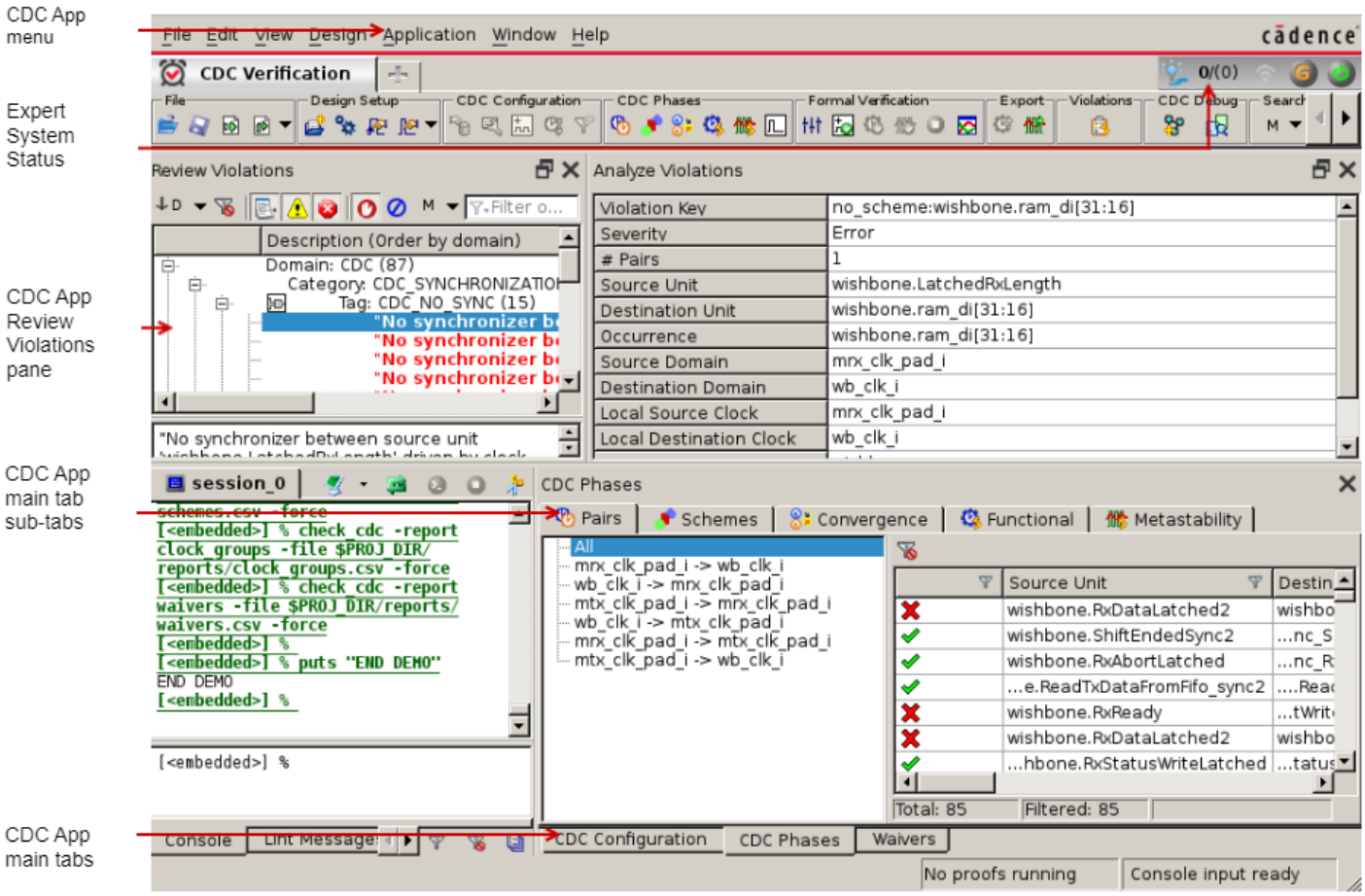


Figure 2.2: CDC App Toolbar

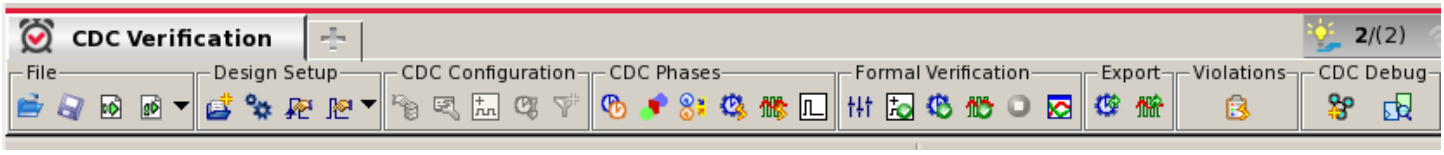


Table 3-1 CDC App GUI Components

Window Component	Function
Application menu	This menu includes the *Configure CDC Checks, Add Signal Configuration, Generate Report, Waivers*, and *Proof* options. Use these options as follows: • *Configure CDC Checks* – Access the *CDC Verification Configuration* dialog box to enable or disable CDC, RDC, or integration checks. • *Add Signal Configuration* – Define one or more signals as constant, static, mutually exclusive, or gray coded. • *Generate Report* – Generate a violations report. • *Waivers* – Access the *Add Waiver* dialog box or export waivers. • *Proof* – Prove signal configuration, protocol checks, or metastability, or access the ProofGrid Manager.

Toolbar	The toolbar includes generic and CDC App toolbars with the following button groups: • *Design Setup* – Set up the design and environment. Use the down arrow to access additional setup and session options. • *CDC Configuration* – Load the port configuration; configure CDC checks; define one or more signals as constant, static, or mutually exclusive, or gray coded; automatically detect all clock domains in the design; or create a filter based on GUI filters. • *CDC Phases* – Find CDC pairs and schemes, identify structural convergence issues, generate protocol checks for previously detected schemes, inject metastability into the synchronizers in the COI of all user properties, and find reset signals. ⚠ For ease of use, the icons on the associated *CDC Configuration* and *CDC Phases* tab sub-tabs mirror the buttons on this toolbar, and clicking these buttons brings the associated sub-tab table to focus if applicable. • *Formal Verification* – Specify proof settings, prove signal configuration, prove protocol checks, verify the result of the metastability injection, stop all jobs in the current session, and access the ProofGrid™ Manager. • *Export* – Export filtered checks or metastability candidates to simulation. • *Violations* – Generate violations reports. • *CDC Debug* – Show CDC graph plus schematic or open the Schematic Viewer. Just above the toolbar on the right is the Jasper Expert System status bar. For additional information, access the *Expert System User Guide* from the tool (*Menu – Application Guides*).
Review Violations, Review Waiver Effects, and Analyze Violations tables	By default, violations are organized by domain and collapsed to provide a summary of reported violations. Double-click, click the + to the left of a domain, or select *Expand All* from the context menu to expand all. You can also use the *Toggle Order Preference* drop-down menu to order the results by *Domain*, *Tag*, *Message*, *Filename*, *Severity*, or *Instance*. Once you have expanded the list to view associated violations for each tag,

	you can click on a message to view a brief description of the highlighted violation in the field at the bottom of the *Review Violations* table or right-click and choose *Open Tag Help* to view the full message details. Click on a category or tag in the *Review Violations* table to see the details of all violations within that category or tag in the *Analyze Violations* table to the right. Click on an individual violation within a tag to view the details of that violation only in the *Analyze Violations* table. Details presented for individual violations are context dependent. For example, an unclocked port violation (PRT_IS_UNCLK) displays the *Port Type* among other details, and an undeclared clock violation (CLK_NO_DECL) displays a *Potential Clock Source*. Use the buttons at the top of the *Review Violations* table to filter by *Info*, *Warning*, or *Fatal*/*Error*. And to include or exclude waivers from the violations view, click the *Show/Hide Waived Violations* button at the top of the violation tree. Clicking the *Show/Hide Violations* button to the left of the *Show/Hide Waived Violations* button hides all violations that have *not* been waived, which allows you to view waived violations only. To the right of these buttons, there is a text filter with a drop-down menu that provides the option to filter the tree contents by *Domain*, *Tag*, or *Message*. To view the details for a specified waiver only, right-click on a waiver in the *Waivers* table and choose *Show Related Violations*. Doing so filters the violations table, which becomes the *Review Waiver Effects* table, to show only violations affected by the selected waivers. Clicking the *Exit Waiver View* button that appears at the top left of the *Review Violation Effects* table returns you to the full list of violations. ⚠ These tables can be undocked independently and repositioned or made separate windows. To undock a table, simply click and drag it outside the GUI, or click the pop-out button at the top right corner of the window. Click this button again to return the table to its original position.
CDC Configuration tab	The *CDC Configuration* tab includes the following sub-tabs: • *Port Configuration* – Includes information about all design ports. Use this table to configure top inputs and outputs and black-boxed inputs and outputs.

	• *Signal Configuration* – Displays constant and static signals. ⚠ During CDC configuration, you might need to define one or more signals as constant, static, or mutually exclusive toggle to resolve problems in the design. For example, to select one of the inputs of a clock mux, the mux select needs to be tied to a constant value. See "Signal Configuration". • *Clock Configuration* – Displays all clock signals in the left-hand pane along with the source type indicating the declaration source of the clock, that is, an SDC or TCL file. Click the *Clock Analysis* button located at the far right of this tab to view the relationships between the clocks selected from the clock table. • *CDC Rules* – This table displays the specifics of the default parameters for structural analysis. It contains a column header for *Rule Type* (Pair, Scheme, Group, Reset, and Config) and for each parameter, for example, *CDC Pair Logic*, *CDC Pair Fanout*, *N Min*, *N Max*, *Sync Chain Logic*, and so forth. Move your cursor and hover over the contents of the first populated cell under each rule header to access a tooltip that defines the rule and lists the default and other valid options. You can modify rule parameters based on your design. All rule modifications are applied globally. • *Filters* – Displays the specifics of all user-added filters. Access the full command reference from the command line (`help check_cdc -gui`) and click on `check_cdc -filter` for additional details. • *User-Defined Schemes* – Displays information for all user-defined and custom schemes. Since not all schemes can be automatically detected, you can add user-defined schemes as needed. Access the full command reference from the command line (`help check_cdc -gui`) and click on `check_cdc -scheme` for additional details. Also see "User-Defined Synchronizers".
CDC Phases tab	The *CDC Phases* tab provides information on the different stages of the CDC analysis and includes the following sub-tabs: • *Pairs* – The left-hand pane displays a tree of all CDC pairs. Click on a CDC pair to see the related information in the right-hand pane. Related information includes source and destination clocks; source and

destination units; pair classification (that is, data, control, or data and control); rule violations; and scheme. Green checks indicate that the CDC pairs passed all path rules, red Xs indicate rule violations, blue "not" icons indicate CDC pairs that you have waived, and orange "not" icons indicate pairs that the tool has automatically waived. Use the *Pairs* table context menu to view a schematic, see associated rule violations in the *Review Violations* table, view a scheme in the *Schemes* sub-tab, or view the source.

- *Schemes* – The left-hand pane displays a tree of all CDC pairs. Click on a CDC pair to see the scheme information in the right-hand pane. Scheme information includes scheme, scheme type, detection type, and violation. A third pane at the bottom of the *Schemes* sub-tab includes information from the *Pairs* sub-tab.
- *Convergence* – The left-hand pane displays a tree that lists all identified convergence issues. Click on a group of issues (or expand the group [+] and click on a specific issue) to see the CDC group information in the right-hand pane. Group information includes CDC group, convergence type, whether the convergence came from the same source domain or signal, and depth. A third pane at the bottom of the *Convergence* sub-tab includes information from the *Pairs* sub-tab.
- *Functional* – The left-hand pane displays a tree that lists all CDC checks. Click on a CDC pair (or expand the pair [+] and click on a scheme) to view a table with the names of the associated functional checks. From this table, you can use the context menu to verify properties, view associated rule violations in the *Review Violations* table, Visualize a failing property, show a property graph, show a scheme schematic, or view a witness trace.
- *Metastability* – This tab includes a task table with a summary of proof results, a property table with assertions and a set of assumptions that represent the formal verification environment, and a table detailing CDC pair information. Prove properties before injecting metastability, and then use the toolbar wizards to inject metastability and verify whether the properties still pass (that is, are immune to metastability effect).

Waivers tab	The *Waivers* table displays information on added waivers. When you right-click on a waiver in the Waivers table and choose *Show Related Violations*, the *Review Violations* table enters the *Review Waiver Effects* view, which is automatically filtered to show only violations affected by the selected waivers. The *Exit Waiver View* button that appears at the top left of the *Review Waiver Effect* table returns you to the full list of violations. Use the *Waive all violations that match the filters* button on the *Analyze Violations* toolbar, which appears when domains, categories, or tags are selected from the *Review Violations* table, to waive all CDC pairs that match any filters you have created using the GUI column filters.
Design Hierarchy tree	The *Design Hierarchy* tree displays the source file hierarchy. ⚠ This pane can be accessed by clicking the *Show Design Hierarchy* button on the far right of the CDC App toolbar. From this pane, you can use the context menu (right-click) to do any of the following: • Open the Source Browser window with a focus on the instance you selected in the tree • Expand (or collapse) all levels of the hierarchy, or click individual expand (+) or collapse (–) buttons to view various levels of the hierarchy • Access the Design Information window with context-sensitive details • Visualize the selected instance in WaveEdit mode or show the related graph or schematic view. To see proof details for a module or entity with embedded properties, move your cursor and hover over the module or instance name to reveal the tooltip.

3

Capturing the CDC Intent

This chapter describes some configuration options, explains how to declare constraints, and introduces user-defined synchronizers:

Setting Up the Design

This section discusses setting up the design and includes the following topics:

Analyzing and Elaborating the Design

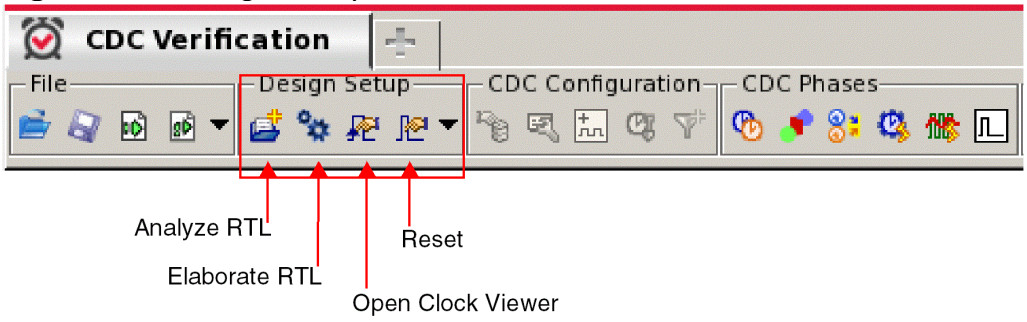
This section provides information on analyzing and elaborating the design and includes the following sections:

- SDC Flow
- Clock Analysis Modes in Jasper CDC

Design Setup Toolbar

Use the *Design Setup* group buttons or related Tcl commands to set up the design and environment. The down arrow provides access to additional setup and session options.

Figure 3.1: Design Setup Toolbar



⚠ For instructions on analyzing and elaborating your design, see "Chapter 3: Specifying the Environment" of the *Jasper Platform and Formal Property Verification App User Guide*.

Liberty Files

CDC can automatically infer clock associations from Synopsys® Liberty™ files for hard macros, but these files must be loaded *before* elaboration. Load the files with the `liberty -load` command. The typical flow follows:

```
% analyze...
// Design configuration, including loading one or more Liberty files
...
% liberty -load libfile1.lib
...
% liberty -load libfileM.lib.gz
...
% elaborate...
...
% check_cdc -clock_domain -find -use_liberty_information
```

ⓘ Run `liberty -load` *before* elaboration.

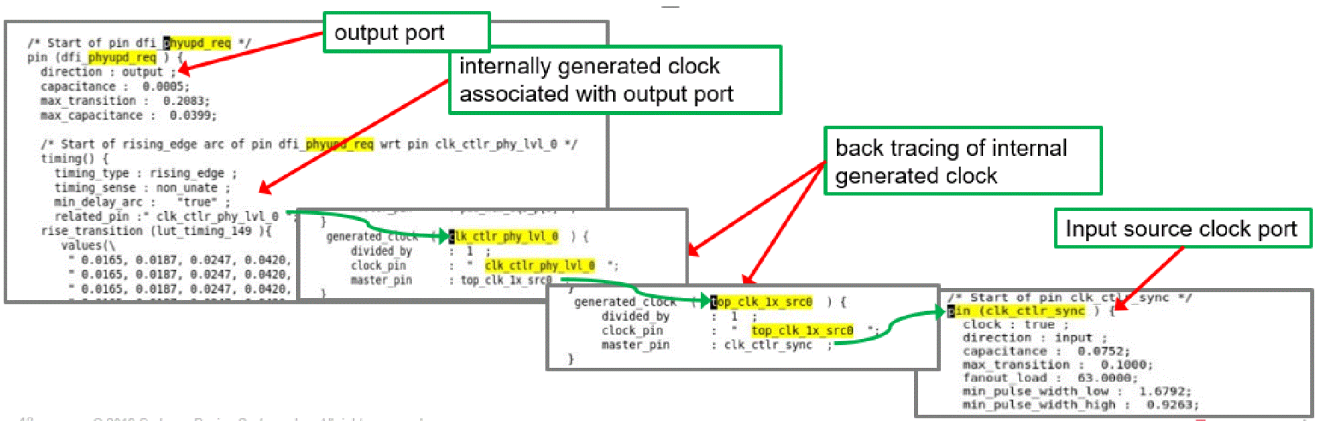
Note:

- See the full command help (`help check_cdc -gui`) for the known limitations of this command.
- Beginning with the 2021.06 release, the `-use_liberty_information` switch is no longer required. CDC now extracts the clock association for Liberty ports automatically.

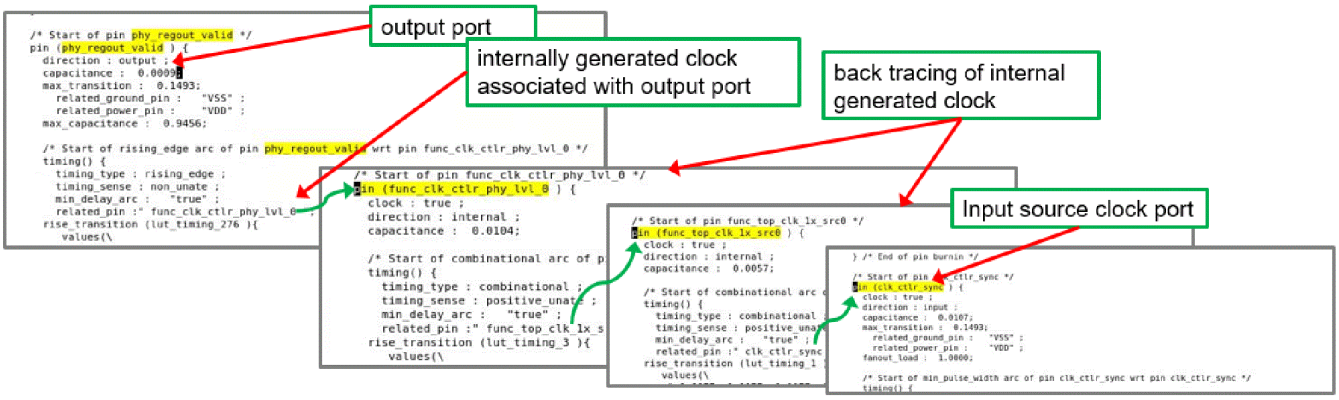
Support for Internally Generated Clocks

The clocks inside a Liberty file can be generated internally in the following three ways:

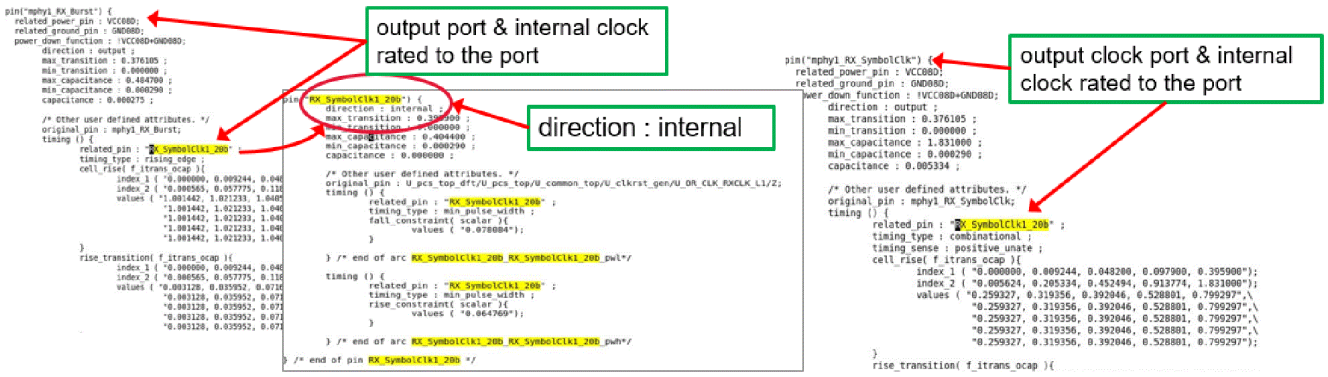
- **Using the generated_clock attribute** – In this scenario, the internal clock is generated from an input clock source. CDC recursively traces back to find the source pin of the internally generated clock. The boundary ports are then associated with the input source clock of the generated clock.



- **By internal pin in Liberty –** In this scenario, the internal clock is not a generated clock, but a clock related to one of the input source clocks. The tool traces back recursively to get to the source pin of the internal clock. The Liberty boundary ports are associated to the input source clock of the internal clock.



- **Virtually –** In this scenario, the internal clock is not related to any input source clocks. As a result, CDC creates a virtual clock for the internally generated clock. The Liberty boundary ports related to the internally generated clock are associated to the virtual clock. The virtual clock created is shown as a virtual clock domain in the CDC clock domain table with signal type set to `liberty`. The name of the virtual clock is `[instance name].[pin name]`.



SDC Flow

Beginning with the 2021.06 release, the Jasper CDC App can read native SDC files and provide the following support:

- Capability for Tcl-layered SDC
- Option to reload a new or updated SDC file without clearing the rest of the design results

To load the SDC file in your design setup, use the following command:

```
read_sdc (sdc_file_name | -clear)
```

- *sdc_file_name* : Specifies the SDC file to be loaded
- `-clear`: Clears the SDC constraint information from a previously read SDC file

When you read an SDC file with this command, the Jasper console displays a summary report that prints the count of successful, failed and ignored executions for all commands that are processed. An example follows:

```
================================================================================
                          Command Execution Report
================================================================================
          command name          | successes | failures  |  ignored  |   total
--------------------------------------------------------------------------------
all_clocks                      |     2     |     0     |     0     |     2
create_clock                    |     4     |     0     |     0     |     4
current_design                  |     3     |     0     |     0     |     3
foreach_in_collection           |    11     |     0     |     0     |    11
get_clocks                      |   127     |    78     |     0     |   205
get_pins                        |   153     |     0     |     0     |   153
get_ports                       |    31     |     0     |     0     |    31
set_case_analysis               |   153     |     0     |     0     |   153
set_clock_groups                |     1     |     0     |     0     |     1
set_clock_uncertainty           |     0     |     0     |     2     |     2
set_input_delay                 |     9     |    78     |     0     |    87
set_max_capacitance             |     0     |     0     |     1     |     1
set_max_fanout                  |     0     |     0     |     1     |     1
set_max_transition              |     0     |     0     |     1     |     1
set_output_delay                |   118     |     0     |     0     |   118
--------------------------------------------------------------------------------
```

The console summary along with the reason of failures (if any) in the SDC file can be found in the log file located at the following path:

```
% jgproject/sessionLogs/session_N/cte.log
```

⚠ `session_N` represents the current Jasper session.

Note:

- When the tool encounters an unsupported SDC command, it issues a warning that points to the line and file where the command was used. This is shown as a failure in the command execution summary report.
- When processing `create_clock` and `create_generated_clock` commands, the tool determines the clock factor with respect to the internal clock by determining the time period of the clock specified with the optional `-period` switch.

Starting with 2021.12, you can also provide SDC configurations in the design using the following command:

```
% read_sdc -config {{<config_command1> <config_val1>} {<config_command2 config_val2>}}
```

For example, to specify the register naming style as <register_name>_reg/Q in the SDC, you can run the following command:

```
% read_sdc -config {hdl_reg_naming_style %s_reg%s}
```

If you want to specify the generated instance name separator as "_", you can run the following command:

```
% read_sdc -config {hdl_generate_separator _}
```

If you want to specify the bus naming convention as <bus_name>_<index>_, you can run the following command:

```
% read_sdc -config {hdl_array_naming_style %s\_%d\_}
```

⚠ These commands should be provided before reading the SDC file using `read_sdc` command.

For a complete list of supported SDC commands, see Chapter 9, "Supported SDC Commands."

Auto-Normalization of Clock Ratios

In SDC, the clock frequency is specified using `-period` and/or `-waveform` arguments of the `create_clock` command. In Jasper, clock frequencies are specified using the `-factor` argument to the `clock` command. Jasper CDC uses a clock period normalization algorithm to translate the SDC clock periods to Jasper clock factors in order to run functional checks. This algorithm tries to match the SDC clock periods to Jasper clock factors by keeping those factors below 10 and also keeping the clock proportions as accurate as possible. The final clock factors calculated may result in one or more clocks having identical factors, although the clock periods declared in SDC were different.

Clock Analysis Modes in Jasper CDC

There are two clock analysis modes that are supported in Jasper CDC. They are:

- STA Mode – This is the default behavior of the tool where all the clocks are considered to be synchronous with each other unless explicitly specified.
- Async Mode – In this mode, all the clocks are considered to be asynchronous to each other.

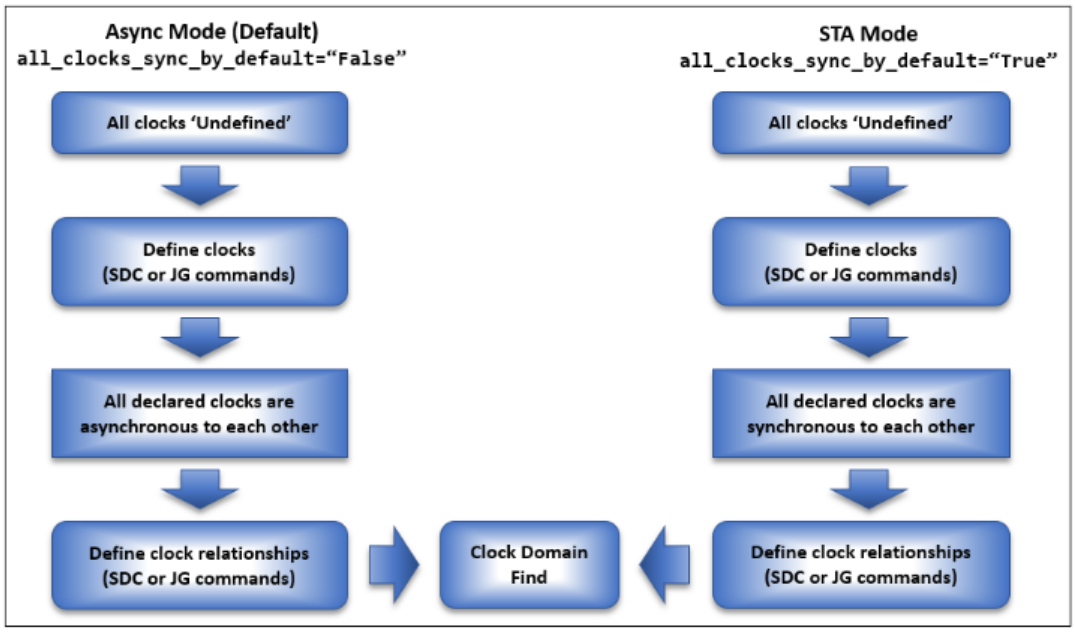
Any synchronous relationship should be explicitly specified.

The clock analysis mode can be set either by the following Tcl command or by enabling the following parameter in the CDC rule file.

```
% config_rtlds -rule -parameter {all_clocks_sync_by_default = true|false}
% params CDC_APP {all_clocks_sync_by_default = True|False}
```

Setting the parameter `all_clocks_sync_by_default` to `true` enables the STA clock analysis mode, and setting it to `false` enables the Async clock analysis mode. The default value of the parameter is `true`.

⚠ The above change (Tcl command or rule file change) must be completed before running the `elaborate` command.



Async Mode

Async Mode is the default behavior of the tool. All clocks declared using the Jasper `clock` command are, by default, considered to be asynchronous to each other unless explicitly specified.

For example, consider the following scenario:

```
% clock clk1
% clock clk2
```

The two clocks `clk1` and `clk2` defined using the Jasper `clock` command will be considered

asynchronous to each other unless explicitly specified as synchronous using the `config_rtlds -clock { } -group { }` command or by enabling the STA mode.

⚠ Beginning with the 2022.03 release, CDC considers clocks declared through SDC as synchronous to each other by default, irrespective of the parameter setting.

STA Mode

When the parameter `all_clocks_sync_by_default` is set to `true`, the STA mode is enabled. In this mode, the tool considers all declared clocks (both native and SDC) as synchronous to each other. You need to explicitly specify any asynchronous relationship between the clocks. This provides better compatibility and consistency with the way clock relationships are specified in SDC.

Since this behavior is consistent with STA tools and all clocks will be considered synchronous by default, asynchronous clock relationships can be defined using the `set_clock_groups -async` command in SDC or `config_rtlds -clock { } -group { }` command in Jasper CDC run file.

For example, consider the following two relationships declared in SDC:

```
% set_clock_groups -name "async1" -async -group [list [get_clocks clk1] [get_clocks clk2]] -group [list [get_clocks clk3] [get_clocks clk4]]
% set_clock_groups -name "async2" -async -group [get_clocks clk1] -group [get_clocks clk2]
```

Before any `set_clock_groups` command is run, all clocks are synchronous to each other in the STA mode. The first command indicates that `clk1` and `clk2` are asynchronous to clocks `clk3` and `clk4`. The second command specifies `clk1` and `clk2` to be asynchronous to each other. The table below illustrates the final relationships between all clocks:

*****	clk1	clk2	clk3	clk4	clk5
clk1	*****	async	async	async	sync
clk2	async	*****	async	async	sync
clk3	async	async	*****	sync	sync
clk4	async	async	sync	*****	sync
clk5	sync	sync	sync	sync	*****

In the above example, as no clock relationship has been defined by `clk5` clock, the tool considers `clk5` to be synchronous to all the other declared clocks in STA mode.

The same relationships can be achieved using the following Jasper commands:

```
% config_rtlds -clock -group {{clk1 clk2} {clk3 clk4}}

% config_rtlds -clock -group {{clk1} {clk2}}
```

*****	clk1	clk2	clk3	clk4	clk5
clk1	*****	async	async	async	sync
clk2	async	*****	async	async	sync
clk3	async	async	*****	sync	sync
clk4	async	async	sync	*****	sync
clk5	sync	sync	sync	sync	*****

In the above example, since no clock relationship has been defined by `clk5` clock, the tool considers `clk5` to be synchronous to all the other declared clocks in STA mode.

Clock Declaration

If you are not using the SDC flow, you can use the Jasper command to declare clocks. This section provides information on declaring clocks and includes the following sections:

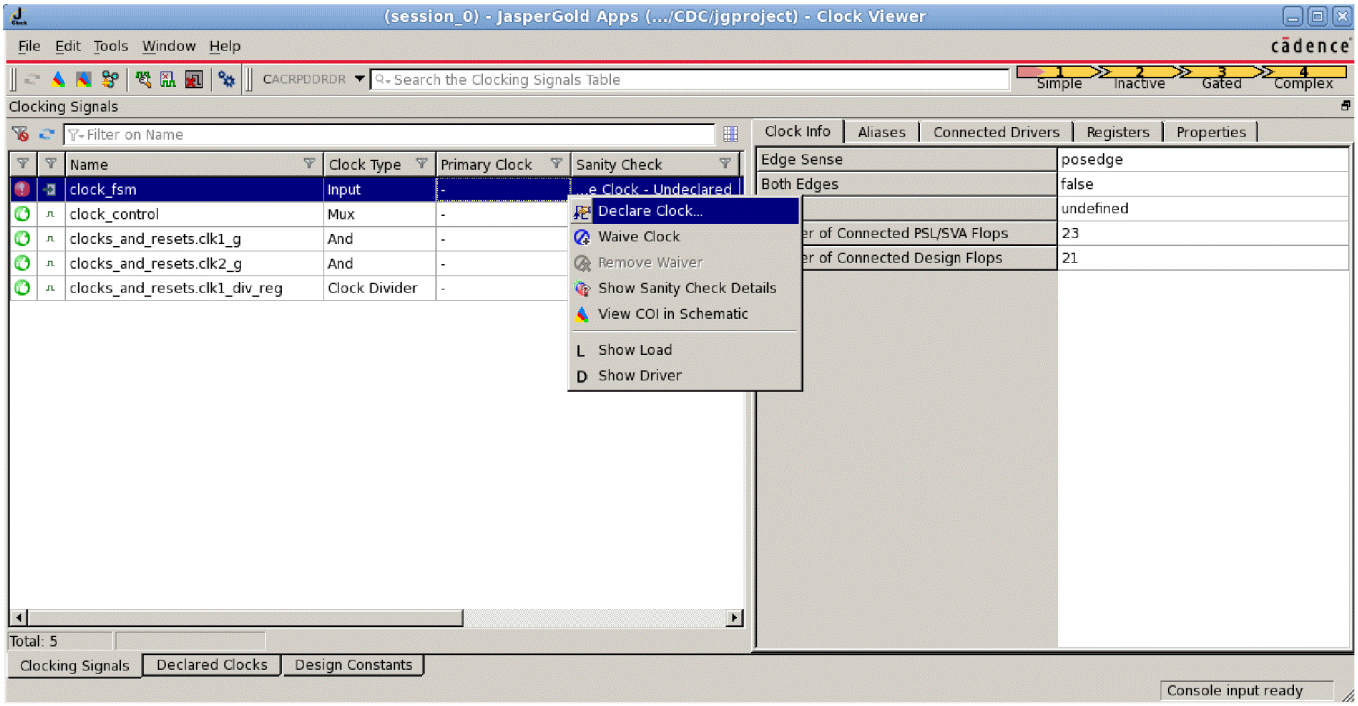
- Jasper Clock
- Declaring Internal Signals as Clocks
- Virtual Clocks
- Defining User-Defined ICG Cells

Jasper Clock GUI

To declare a clock from the Jasper GUI, do the following:

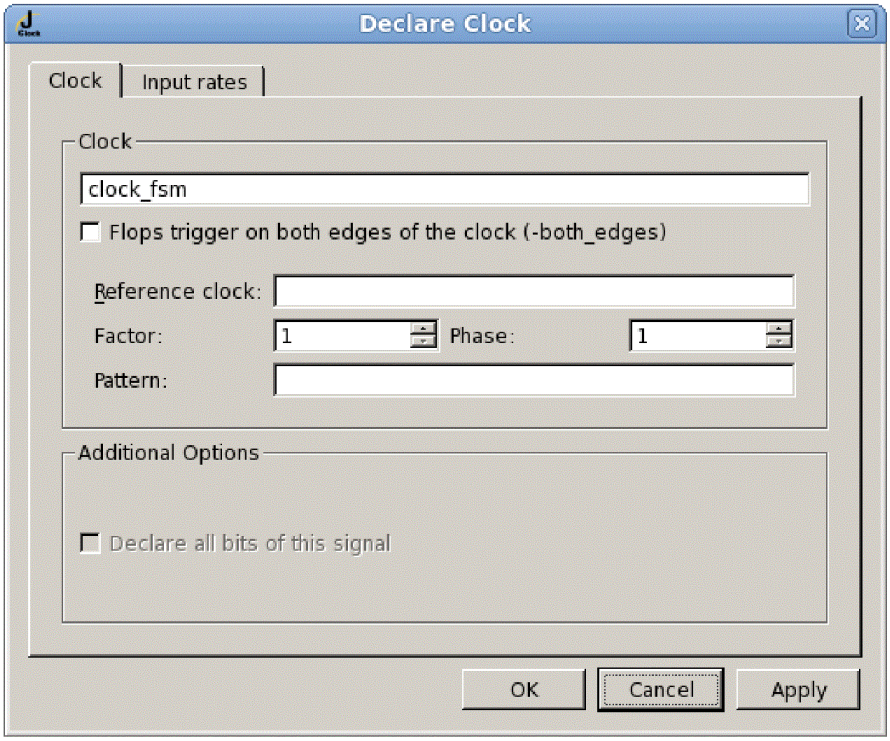
1. Click the *Open Clock Viewer* button on the CDC *Design Setup* toolbar (Figure 4-1).

   The Clock Viewer opens, providing a centralized location for clock environment information.



2. Right-click on an undeclared clock signal and choose *Declare Clock.*

   The *Declare Clock* dialog box opens with the clock name pre-filled.



3. Specify the *Factor* and *Phase* if you want to change the pre-filled default values.
4. Click *OK*.

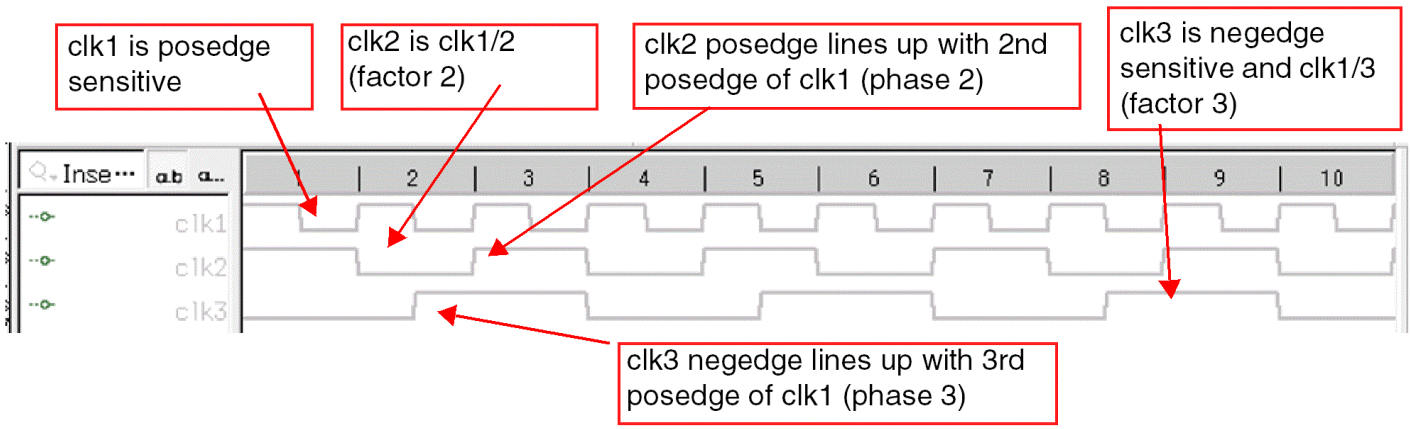
You can also declare clocks from the command line using the Jasper `clock` command as follows:

```
// Positive edge active clock
% clock clk
// Negative edge active clock
% clock ~clk
```

Specifying Clock Factor and Phase

In the following example, all clocks are relative to formal engine cycles (implicit):

```
// All clocks relative to formal engine cycles
% clock clk1
% clock clk2  -factor 2 -phase 2
% clock ~clk3 -factor 3 -phase 3
```



In the next example, which is equivalent to the previous example, all clocks are relative to the specified reference clock `clk1`:

```
// All clocks relative to clk1
% clock clk1
% clock clk1 clk2  2 2
% clock clk1 ~clk3 3 3
```

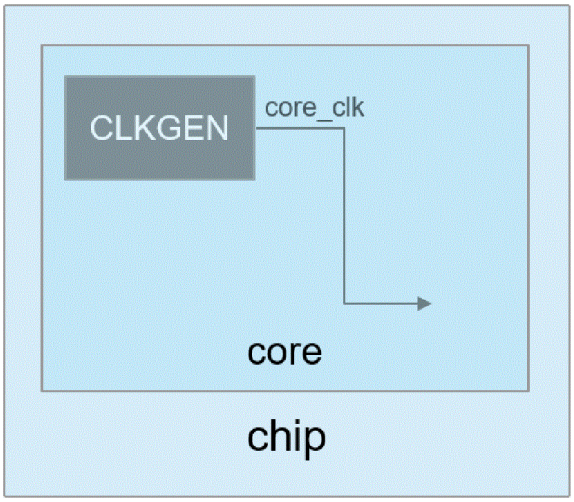
Note:

- When you specify a reference clock, the `-factor` and `-phase` switches are not allowed. Instead, the factor and phase are represented by *N* only.
- For complete details on the `clock` command, they `help clock -gui` on the command line.

Declaring Internal Signals as Clocks

Only signals without a driver can be declared as a clock. If the signal is an output of a module, black-box the module or apply an environmental stopat. See the following example:

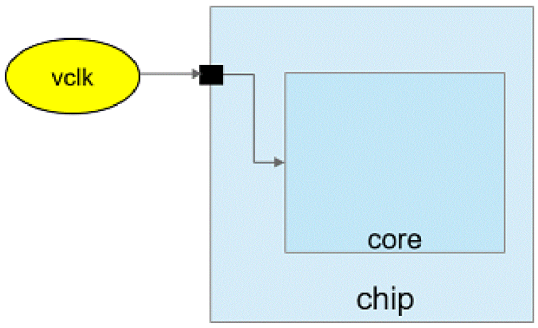
```
% stopat -env chip.core.clkgen.core_clk
% clock chip.core.clkgen.core_clk
```



Virtual Clocks

Use `check_cdc -clock_domain -virtual_clock` *`name_list`* to create virtual clocks that represent clock domains external to the design. The tool treats these as separate clock domains and reports CDC crossings between virtual clocks and design clocks. See the following example:

```
% check_cdc -clock_domain -virtual_clock vclk
% clock vclk
```



CDC creates a virtual clock for any internally generated clock in a Liberty file. This virtual clock is in a new clock domain. The Liberty output ports connected to the internally generated clock are rated in the corresponding virtual clock.

The virtual clock generated from Liberty is shown in the *Clock Domains* table with *Signal Type* Liberty. The virtual clock name takes the following format: `[instance_path].[pin_name]`.

Defining User-Defined ICG Cells

There are several ways in which Integrated Clock Gated (ICG) cells can be implemented in the RTL with some of them being non-standard, making it difficult for the tool to detect those. With 2022.06, support for user-defined ICG cells has been enhanced.

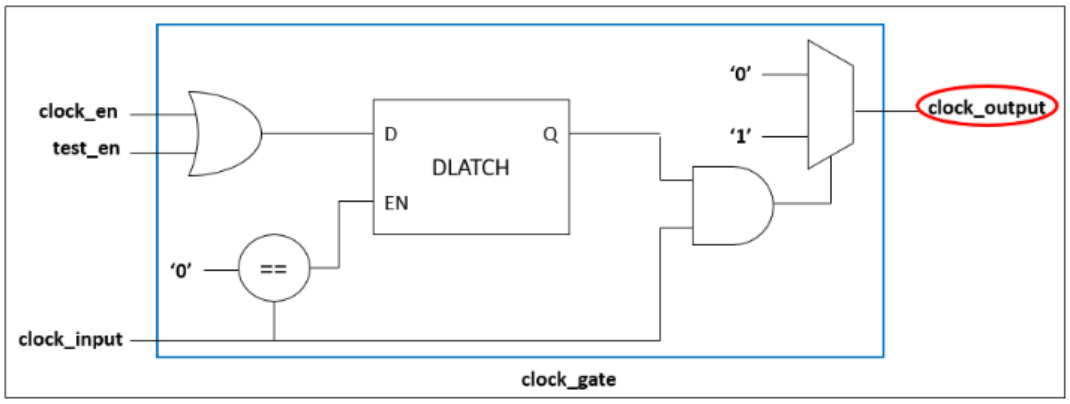
To specify a user-defined ICG cell, use the following command:

```
% check_cdc -clock_domain -icg -add -module <module_name> -map {{clkin
<input_clock_pin>} {enable <clock_enable_pin>} {testenable <test_enable_pin>} {clkout
<output_clock_pin>}}

All formal parameters are mandatory.
```

Example:

In the design below, the tool might not be able to recognize this particular type of ICG cell by default.



For the tool to recognize it, you can add it as a user-defined ICG-cell using the following command:

```
% check_cdc -clock_domain -icg -add -module clock_gate -map {{clkin
clock_input} {enable clock_en} {testenable test_en} {clkout clock_output}}
```

To list all the ICG cells that have been identified by the tool (both user-defined and automatic), you can use the following command:

```
% check_cdc -list icg_cells
```

Reset Declaration

This section provides information on declaring reset and includes information on the following:

- Reset Declaration with "config_rtlds -reset"
- Using Internal Signals as Reset
- Declaring Virtual Resets

Reset Declaration with "config_rtlds -reset"

Beginning with the 2019.06 release, you can choose to declare reset signals using the `config_rtlds -reset` command. The syntax of this command follows:

```
config_rtlds -reset ( -sync signal_list -clock clk_signal |-async signal_list
                     |-synchronized signal_list -clock clk_signal)
                    -polarity (high | low)
```

- Use `-sync` to specify design resets as synchronous (synchronous assertion and de-assertion).

- Use `-async` to specify design resets as asynchronous (asynchronous assertion and de-assertion)
- Use `-synchronized` to specify design resets as equivalent to the output of a reset synchronizer (asynchronous assertion and synchronous de-assertion)

  ⚠ The `-clock` switch is required when declaring a reset signal as synchronous or synchronized

- Use `-polarity` to specify if the reset is active high or active low.

The advantages of using `config_rtlds -reset` to declare resets include the following:

- You can declare all three types of resets (asynchronous, synchronized, and fully synchronous) using `config_rtlds -reset`, but with the Jasper `reset` command, you can declare only asynchronous resets or fully synchronous reset if the reset is associated with a clock.
- Declaring resets with `config_rtlds` avoids any conflict in declaring flop outputs as constants during reset analysis.
- Reset signals are not assumed to be constant during structural analysis.
- Lower order resets can toggle during functional analysis.

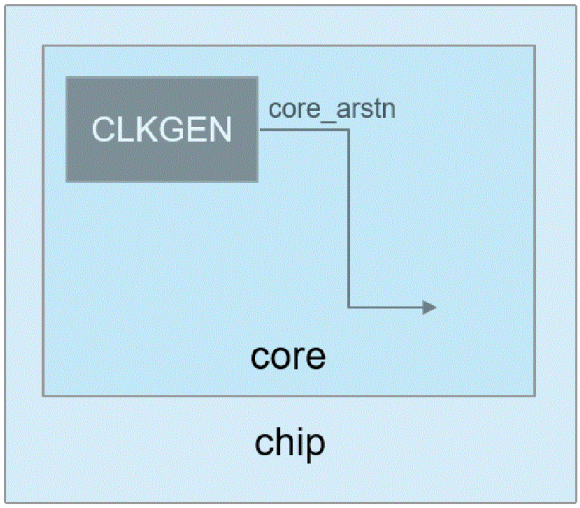
⚠
- Resets declared using `config_rtlds -reset` command will be used for structural analysis. Any resets declared using the Jasper `reset` command will be ignored.
- The tool will automatically infer the resets for functional analysis based on reset declarations and reset order.
- All resets declared using the `config_rtlds -reset` command must be declared before running `check_cdc -clock_domain -find`.

Using Internal Signals as Reset

Only signals without a driver can be declared as reset. If the signal is an output of a module, black-box the module or apply an environmental stopat. If the signal is the output of a flop, apply an environmental stopat.

See the following example:

```
% stopat -env chip.core.clkgen.core_arstn
% config_rtlds -reset -async chip.core.clkgen.core_arstn -polarity low
```

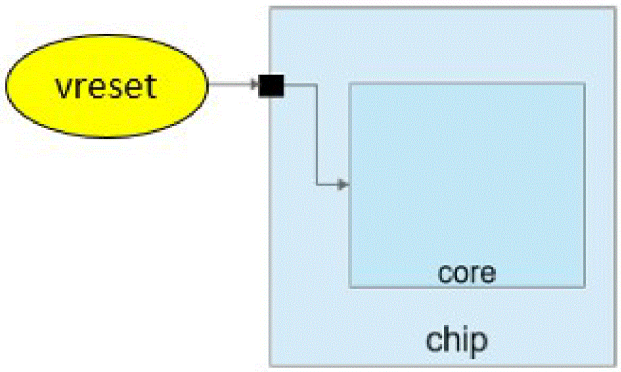


Declaring Virtual Resets

Virtual resets represent a reset source which exists outside the block under consideration for CDC analysis. The tool treats these as separate reset domains and reports reset domain crossings between virtual resets and design resets.

Declare virtual resets with the `config_rtlds -reset` command as follows:

```
% config_rtlds -reset -async {vreset} -polarity low -virtual
```



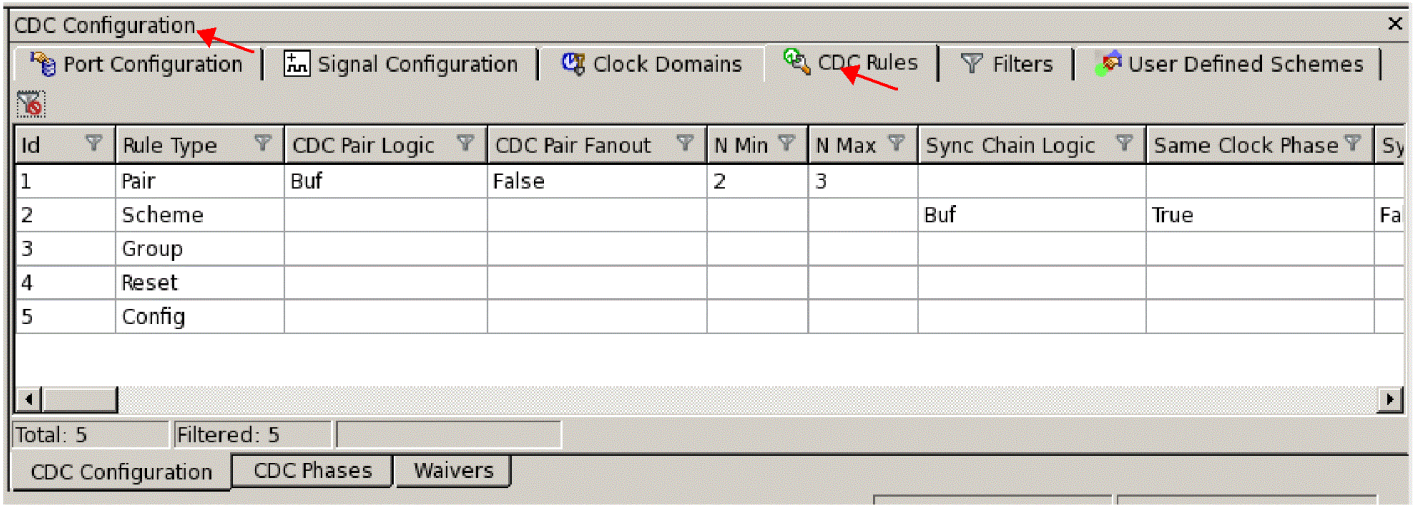
CDC App Path Rule Configuration

This section provides information on the CDC App path parameter settings, finding the default values for each, and command examples for changing the default behavior. It includes the following sections:

CDC Path Parameters Overview

The CDC App path parameters define the criteria for each crossing between source and destination clock domains to ensure proper synchronization of the CDC signals. By default, the tool has a value set for each parameter, but during CDC App setup, you can change the values to specify what the tool considers as violation criteria.

The *CDC Configuration* tab *CDC Rules* sub-tab displays the CDC App path parameters and their values.



The parameters are divided into five categories as follows:

- `pair` – Parameters pertaining to CDC pairs
- `scheme` – Parameters pertaining to synchronization schemes
- `group` – Parameters pertaining to a group of signals involved in convergence

- `reset` – Parameters pertaining to reset issues
- `config` – Parameters pertaining to configuration options

CDC App Rules File

A rules file is a regular text file that contains the definitions of the CDC rule categories and messages. You can find the default rules file `cdc.def`, which contains CDC category and definition defaults, at the following location:

```
install_dir/etc/res/rtlds/rules/cdc.def
```

This rules file can be customized to meet corporate, team-wide, and user-specific design purification requirements. You can change the severity of rules, enable or disable rules individually, and customize different rule parameters. You can also add new categories and definitions to customize the CDC rules file. For additional details on using statements to customize the rules file, see Chapter 8, "Customizing the Rules File."

Modifying Path Parameters

There are two ways to modify the default parameter settings: 1) from the CDC App rules file and 2) from the command line.

From the Rules File

ⓘ These changes must be made *before* elaboration.

To modify the path parameters from the CDC App rules file, use the `params` statement as shown in the following examples:

- Pass the value of a parameter `parameter_name` associated with a rule `tag`. For example, to specify the logic type allowed on the CDC path, use the following statement:

  ```
  params CDC_PR_LOGC {cdc_pair_logic="Buf"}
  ```

  ⚠ If the same parameter is assigned two or more different values, the last assignment takes precedence.

- Specify that the tool automatically waive structural violations due to constant or static values handling as follows:

  params CDC_APP {apply_auto_waivers="True"}

- Set the severity of the rule. The severity can be `error`, `warning`, or `info`. To set the severity of the CLK_IS_CNST to `error`, use the following statement:

  params CLK_IS_CNST {severity=error}

  ⚠ You cannot downgrade the default severity of a message with this parameter. For example, the default severity of `CLK_IS_CNST` is `warning`. You cannot change it to `info`.

From the Command Line

To modify path parameters from the command line, use the `config_rtlds -rule -parameter` command as follows:

```
% config_rtlds -rule -parameter {key=value}+ -tag tag_list
```

Configuring CDC Checks

ⓘ Checks configuration must be completed before elaboration.

You can configure CDC checks from the rules file, the command line, or the GUI as follows:

- Use the `status` statement to configure individual checks from the rules file.
  For example, to disable the PRT_IS_UNCK rule within the CDC_CONFIGURATION category, modify the `cdc.def` file as follows:

  ```
  category CDC_CONFIGURATION
  {
  PRT_IS_UNCK {severity = error} {msg = "msg="Port '%s' is not associated with any clock"}
  {status=off}
  ....
  }
  ```

  For additional details, see Chapter 8, "Customizing the Rules File."

- Enable or disable individual CDC checks, categories of checks, and/or entire domains from

the command line with the following command:

```
config_rtlds -rule ((-enable |-disable) (-tag |-category |-domain [all])
{items_name_list})
```

See the full command help for additional information (`help config_rtlds -gui`).

- The *CDC Verification Configuration* dialog box provides the option to specify which domains, categories, and/or individual checks you want to run. To access the *CDC Verification Configuration* dialog box, click the *Configure CDC Checks* button on the *CDC Configuration* toolbar.

⚠ For a full list of CDC checks, see the Chapter 7, "CDC Violations."

Signal Configuration

You can constrain the design by declaring signals as constant, static, mutually exclusive, gray encoded, or externally synchronized. You can also add false paths to specify paths that make their corresponding CDC pairs considered inactive.

To apply signal configuration, you can use `config_rtlds -signal` as follows :

```
% config_rtlds -signal ( -constant {{signal_name value}+} [-verify [-condition
expression]] [-force]
                        |-static signal_list [-verify [-condition expression]] [-force]
                        |-exclusive signal_list [-verify [-condition expression]]
                        |-gray_code signal_list [-verify]
                        |-controllable signal_list
                        [-mode (global | functional | shift | capture)]
                        [-override] [-condition expression]
                        |-false_path ( -from list |-to list |-through list)+ [-regexp]
                        |-inactive_destination -source_reset list -destination_clock
list (-on_assert|-on_deassert)
```

- Use `-force` to overwrite previous configurations.
- Use `-verify` to specify that the constraint should be proven as part of constraint validation checks using `config_rtlds -signal -prove`.
- Use the `-condition` switch to define a precondition for the constant, static, or exclusive signals for the constraint validation check. It can be used with `-verify` only. Using it without `-`

`verify` will result in an error.

- Use the `-override` switch to resolve conflicts that might exist when defining signal configurations by applying stopats explicitly.

⚠ Currently, the `gray_code` constraint does not support any precondition.

The old command `check_cdc -signal_config` for specifying signal configurations is still maintained for backward compatibility, but you are strongly recommended to use the config_rtlds -signal command for all signal configuration declarations.

```
% check_cdc -signal_config ( -add_constant {{signal_name value}+} [-force]
                            |-add_static signal_list [-force]
                            |-add_exclusive signal_list
                            |-add_gray_code signal_list
                            |-add_false_path ( -from list | -to list | -through list)+
[-regexp]
                            |-add_inactive_destination -source_reset list -
destination_clock list ( -on_assert|-on_deassert )
```

- Use `-force` to overwrite the existing applied configuration.

⚠ All signal configurations declared using `check_cdc -signal_config` command will work as assumptions only during CDC analysis. These constraints cannot be proven. The constraint validation flow will only be possible to run if you are declaring constraints with `config_rtlds -signal` command.

Difference between the two commands:

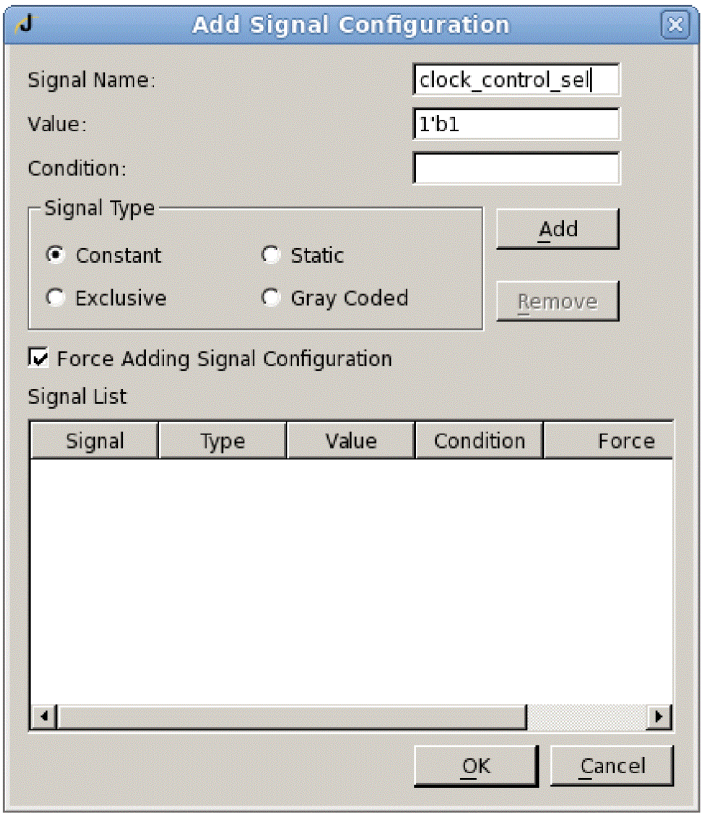
- Stopats are applied on internal nets to resolve the conflicts that might exist between the reset value and configuration applied. While using the `check_cdc -signal_config` command, the stopats can be applied automatically by setting the `add_stopat_on_internal_constants` parameter to `true`. However, while using the `config_rtlds -signal` command, the stopat needs to be explicitly added using the `-override` switch.

This section details GUI procedures and specifies Tcl commands for declaring constraints. It includes information on the following:

Constants

To declare a port as constant, do the following:

1. Click the Add signal configuration button.
   The *Add Signal Configuration* dialog box opens.

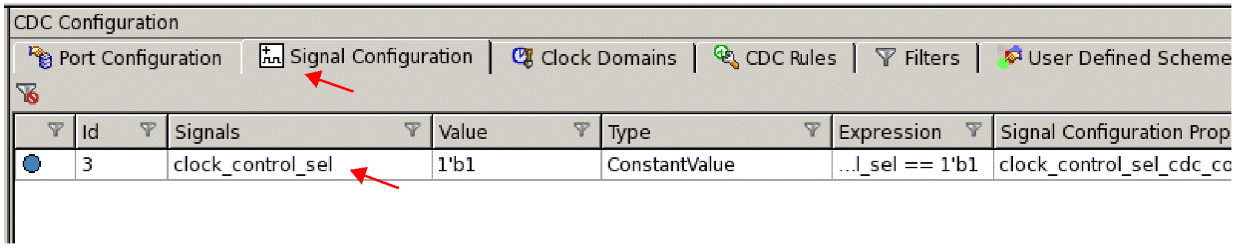


2. Enter the signal name in the *Signal Name* field.
3. Complete the *Value* field.

4. Specify the *Signal Type* by selecting the *Constant* radio button.
5. Click the *Force Adding Signal Configuration* check box if you want the current configuration to overwrite any existing configuration.
6. Click *Add*.

   ⚠ A constant declaration without conditions on undriven nets (primary inputs or stopat nets) is always considered an assumption.

7. Click *OK*.
   The tool adds the configuration details to the *CDC Configuration* tab's *Signal Configuration* tab.



Associated Tcl Command

```
% check_cdc -signal_config -add_constant {{clock_control_sel 1'b1}}
```

OR

```
% config_rtlds -signal -constant {{clock_control_sel 1'b1}}
```

Conditional Constants

You can define a precondition for the constant signal configuration by using the `-condition` switch as follows:

```
% check_cdc -signal_config -add_constant {{signal_name <value>}} -condition
<expression>
```

⚠ The tool treats all constraints declared with `check_cdc -signal_config` as assumptions during structural analysis. No signal configuration checks are generated for these constraints even if a precondition is specified. To run constraint validation or signal configuration checks, declare constraints using the `config_rtlds -signal` command.

OR

```
% config_rtlds -signal -constant {{signal_name <value>}} -condition <expression> -
verify
```

⚠ By default, the tool applies all constraints as assumptions during structural analysis irrespective of any precondition specified. When running constraint validation or signal configuration checks using `config_rtlds -signal -prove`, the signals marked with the switch `-verify` are converted into assertions under the task *RTLDS_signal_config_properties.*

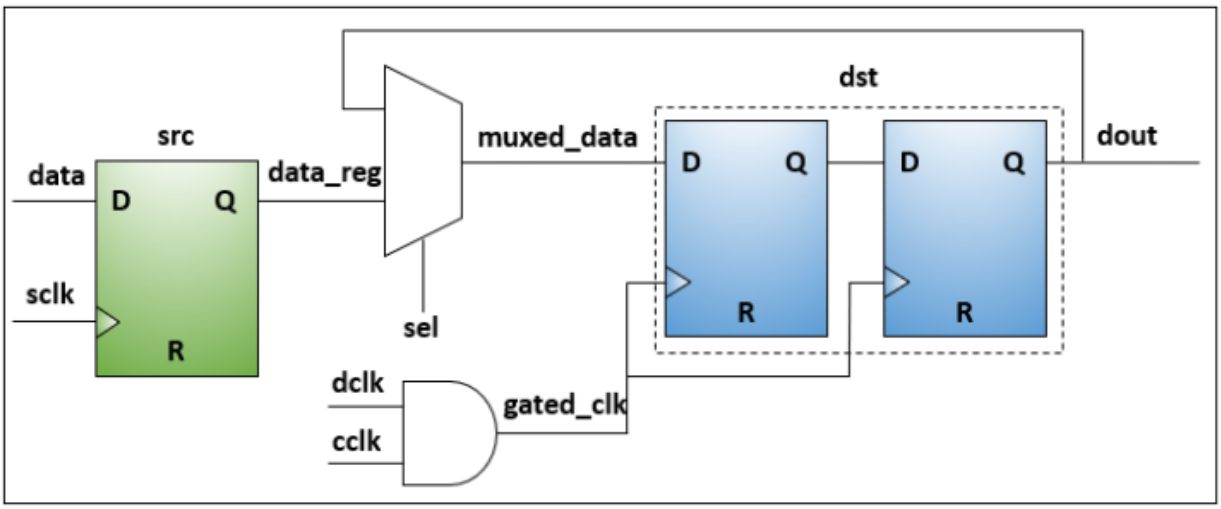
Example:

In the example below, `data_reg` has been declared as constant based on a condition that it will be `1'b0` only if the `gated_clk` remains low.

```
% check_cdc -signal_config -add_constant {{data_reg 1'b0}} -condition {!gated_clk}
```

OR

```
% config_rtlds -signal -constant {{data_reg 1'b0}} -condition {!gated_clk} -verify
```



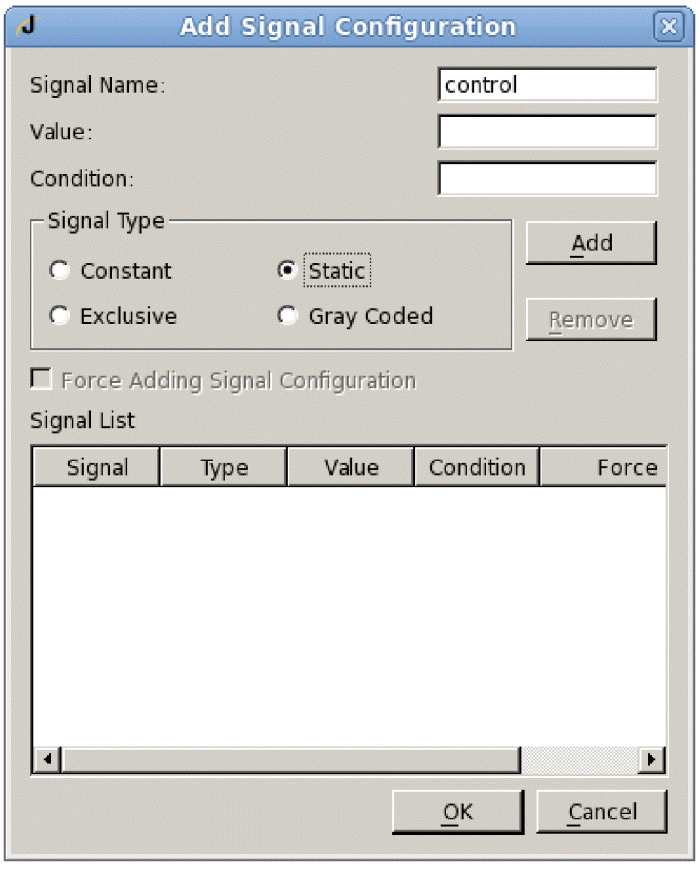
When you use the `config_rtlds -signal -prove` command, the tool attempts to prove that the signal `data_reg` remains constant 1'b0 when `gated_clk` remains low. If the check fails, the tool reports a `CND_NO_CONS` violation.

⚠ Preconditions are necessary when you want to verify the conditional nature of constraints during the functional analysis. During proof, if the constraint check fails when the precondition is true, the tool reports `CND_NO_CONS` or `SIG_NO_STAT` or `SIG_NO_EXCL` violations depending on whether you applied constant, static, or mutually toggle exclusive signal configuration. The failing check indicates that the signal you are defining as constant or static or mutually toggle exclusive is actually not constant or static or mutually toggle exclusive under the given condition.

Pseudo Statics

To declare a port as static, do the following:

1. Click the Add signal configuration button.
   The *Add Signal Configuration* dialog box opens.

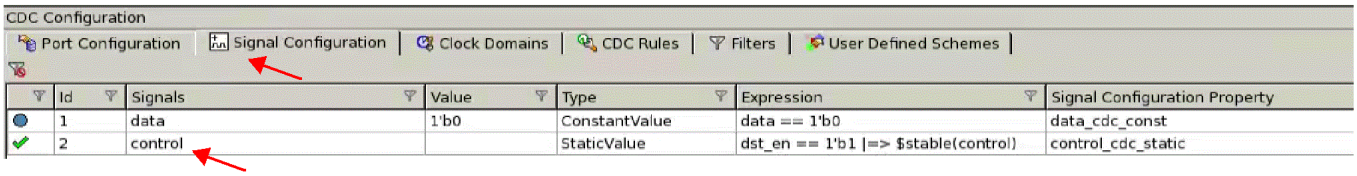


2. Enter the signal name in the *Signal Name* field.
3. Complete the *Condition* field if you want to specify a precondition, which is used for functional checks.

4. Specify the *Signal Type* by selecting the *Static* radio button.
5. Click the *Force Adding Signal Configuration* check box if you want the current configuration to overwrite any existing configuration.
6. Click *Add*.

   ⚠ Static declarations without condition on undriven nets (that is, primary inputs or stopat nets) are considered assumptions.

7. Click *OK*.
   The tool adds the configuration details to the *CDC Configuration* tab's *Signal Configuration* tab.



Associated Tcl Command

```
% check_cdc -signal_config -add_static {{control}} -condition {dst_en == 1'b1}
```

⚠ The tool treats all constraints declared with `check_cdc -signal_config` as assumptions during structural analysis. No signal configuration checks are generated for these constraints even if a precondition is specified. To run constraint validation or signal configuration checks, declare constraints using the `config_rtlds -signal` command.

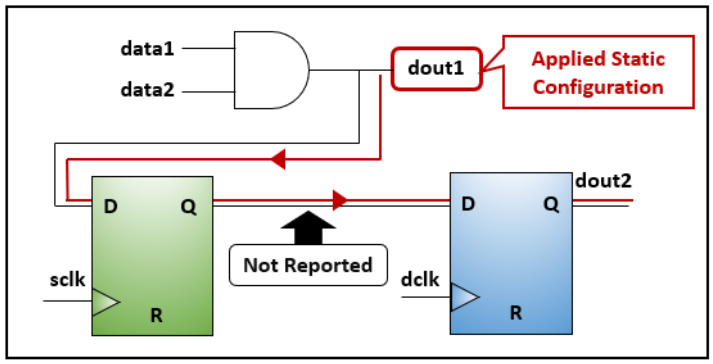
OR

```
% config_rtlds -signal -static {{control}} -condition {dst_en == 1'b1} -verify
```

⚠ By default, the tool applies all constraints as assumptions during structural analysis irrespective of any precondition specified. When running constraint validation or signal configuration checks using `config_rtlds -signal -prove`, the signals marked with the switch `-verify` are converted into assertions under the task *RTLDS_signal_config_properties*.

⚠

- All CDC pairs that have been declared as static are marked as inactive by the tool. You can find a list of inactive signals by running `check_cdc -list inactive_pairs`, or enable the `CDC_PR_INAC` rule with the command `config_rtlds -rule -enable -tag CDC_PR_INAC` to view the related INFO messages on the Violation tree.
- Ensure that the signal being declared as static is either a primary input or on a register. Applying static configurations on a primary output can result in other connected nets becoming static, which can prevent actual CDC crossings from being reported. See the image below.



Mutually Toggle Exclusive

You can define a set of source signals (output of the source flop) as mutually exclusive, which means that only one of these signals can toggle at the same time.

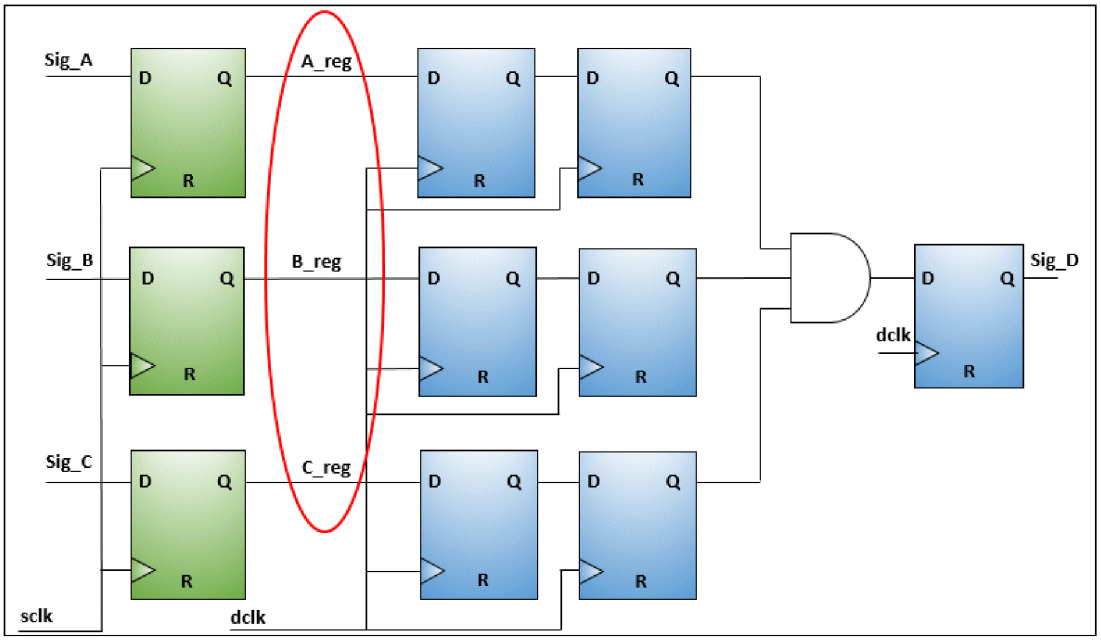
If there is more than one signal converging into a combinational logic in the destination domain, but not more than one can toggle at the same time, you can safely waive the convergence violation. Thus, when you declare signals mutually exclusive, the tool attempts to apply automatic waivers.

For example, to specify signals `A_reg`, `B_reg`, and `C_reg` as mutually toggle exclusive, use one of the following commands:

```
% check_cdc -signal_config -add_exclusive {A_reg B_reg C_reg}
```

OR

```
% config_rtlds -signal -exclusive {A_reg B_reg C_reg} -condition {enable == 1'b1} -
verify
```



The tool uses this information to apply automatic waivers to applicable CDC groups during convergence detection.

The signals must be either single-bit signals, which might or might not be bits of a bus, or individual bits of a multi-bit bus. See the following examples:

- Single-bit signals: `sigA`, `sigB`, `sigC`
- Bits of different buses: `busP[0]`, `busQ[7]`, `busR[3]`
- Individual bits of a multi-bit bus: `sigX[0]`, `sigX[1]`, `sigX[2]`, `sigX[3]`, or `sigX`

> ⚠ By default, the tool applies all constraints as assumptions during structural analysis irrespective of any precondition specified. When running constraint validation or signal configuration checks using `config_rtlds -signal -prove`, the signals marked with the switch `-verify` are converted into assertions under the task *RTLDS_signal_config_properties*.

Note:

- The mutually exclusive toggle relationship cannot be specified between two multi-bit buses.
- The tool issues a warning message during the convergence phase when the signal list you have provided does not follow the listed specifications. In this case, the mutex signals declared will have no effect in the auto waivers flow.

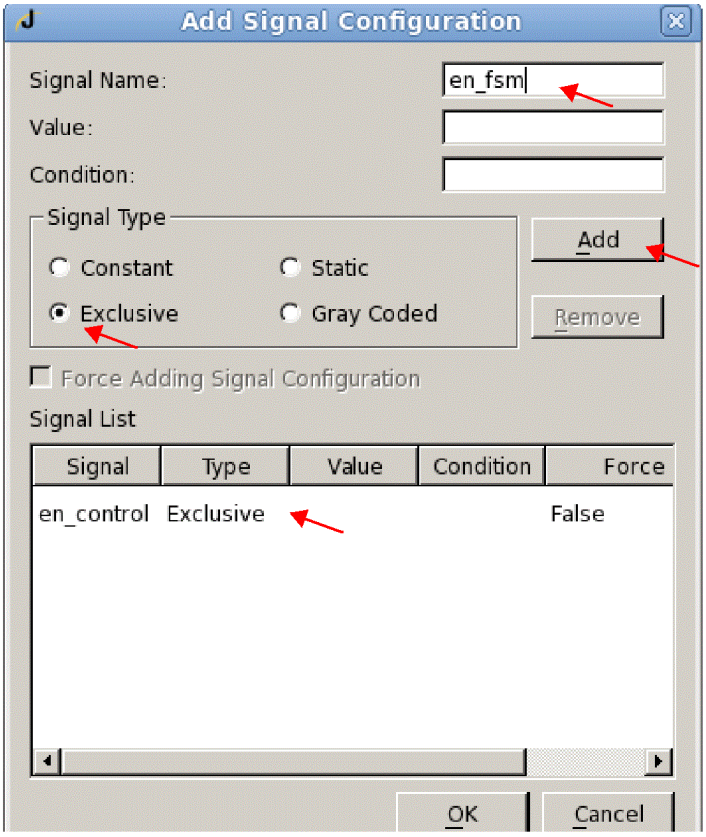
- You can declare the mutually exclusive toggle relation during the CDC Configuration phase only.

Associated GUI Procedure

To use the *Add Signal Configuration* dialog box to declare signals mutually exclusive, do the following:

1. Click the Add signal configuration button.

   The *Add Signal Configuration* dialog box opens.



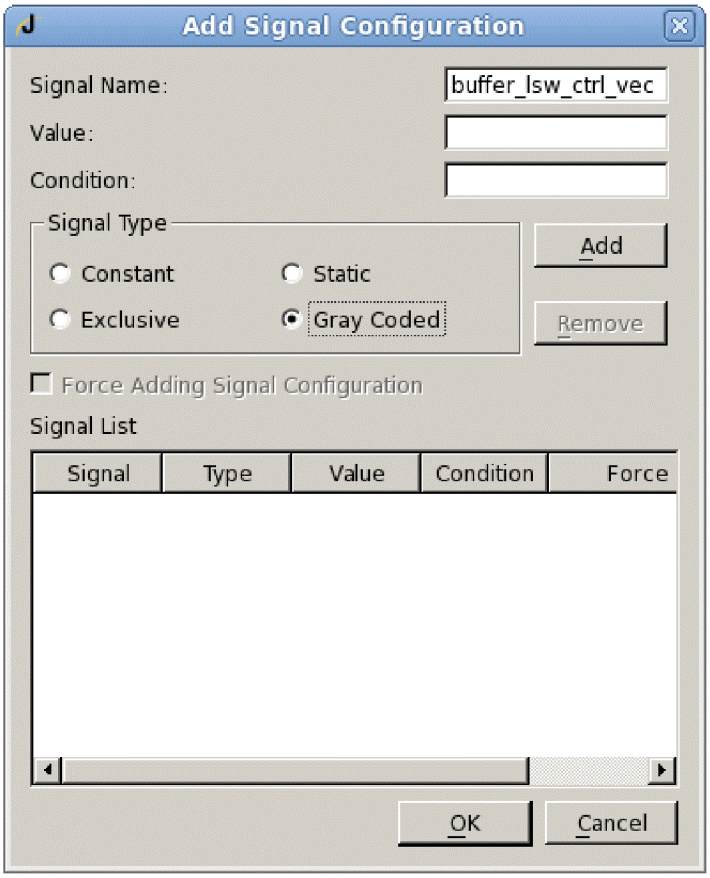
2. Enter the signal name in the *Signal Name* field.
3. Specify the *Signal Type* by selecting the *Exclusive* radio button.
4. Specify an expression in the *Condition* field if you want to use it as a precondition, which is used for functional checks.
5. Click the *Force Adding Signal Configuration* check box if you want the current configuration to overwrite any existing configuration.

6. Click *Add*.
   The single is added to the *Signal List*.
7. Specify a second signal in the *Signal Name* field.
   You must specify at least two signals.
8. Click *Add*.
9. Specify additional signals in the *Signal Name* field as required.
10. Click *Add*.
11. Click *OK*.

Gray Coded

You can also use the *Add Signal Configuration* dialog box to create user-defined gray encoded checks on bus signals as follows:

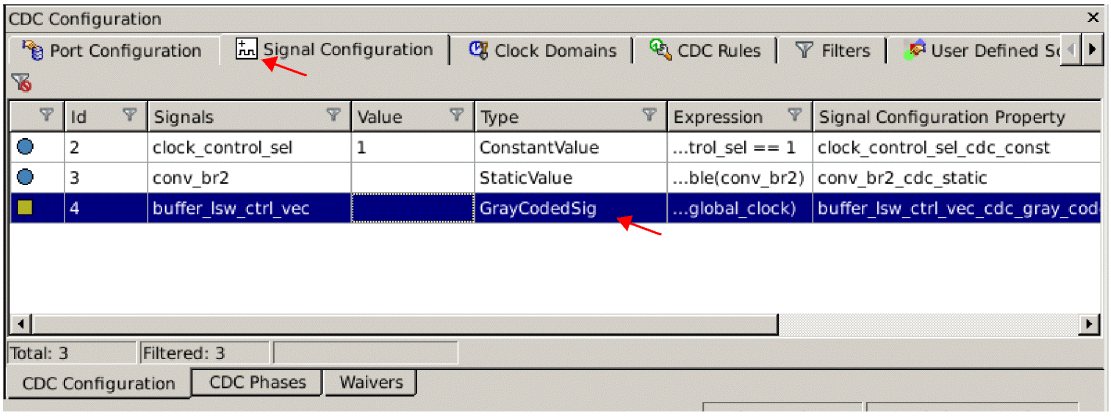
1. Click the Add signal configuration button.
   The *Add Signal Configuration* dialog box opens.



2. Enter the signal name in the *Signal Name* field.
3. Specify the *Signal Type* by selecting the *Gray Coded* radio button.
4. Click the *Force Adding Signal Configuration* check box if you want the current configuration to overwrite any existing configuration.
5. Click *Add*.

   ⚠ The tool does not honor gray-coded constraints on non-bus signals.

6. To add additional gray-encoded signals, type another signal name in the *Signal Name* field and repeat steps 2-5.
7. Click *OK*.
   The tool adds the signal to the *CDC Configuration* tab's *Signal Configuration* sub-tab as type `GrayCodedSig`.



If a signal is undriven, the tool generates an assumption; otherwise, the tool generates an assertion that you can prove with the `check_cdc -signal_config -prove` command. If the property is not proven, the tool generates a signal configuration violation.

CDC False Path

You can specify paths that should make their corresponding CDC pairs considered inactive by defining those paths as *false* paths. Specify false paths with the following commands:

```
% check_cdc -signal_config -add_false_path
    [-from {<object>+}]
    [-to {<object>+}]
    [-through {<object>+}]
    [-regexp]

OR

% config_rtlds -signal -false_path
    [-from {<object>+}]
    [-to {<object>+}]
    [-through {<object>+}]
    [-regexp]
```

`<object>` can be a primary clock, internal clock net, internal signal, instance port, and so forth. At least one argument, `-from`, `-to`, or `-through`, is mandatory

⚠ If there is a mismatch of signal name in the `-to{}` or `-from{}` signal list, the tool issues a warning `(WCDC115)` indicating the absence of the signal in the design and considers the command invalid. However, depending on the requirements, you can upgrade this `warning` to an `error`.

If you included `-regexp`, the tool interprets `<object>` a regular expression.

See the following examples:

```
// All CDC paths from clk1
% check_cdc -signal_config -add_false_path -from clk1
OR
% config_rtlds -signal -false_path -from clk1

// All CDC paths between clk2 and clk1
% check_cdc -signal_config -add_false_path -from clk2 -to clk1
OR
% config_rtlds -signal -false_path -from clk2 -to clk1

// All CDC paths starting from pair
% check_cdc -signal_config -add_false_path -to pair.* -regexp
OR
% config_rtlds -signal -false_path -top pair.* -regexp
```

Specifying Inactive Constraints for Reset

You can use the following commands to specify that a certain reset be active only when the clock is inactive. Inactive constraints allow you to eliminate violations that can be ignored if the clock is inactive during reset assertion or de-assertion

```
% check_cdc -signal_config -add_inactive_destination
                           -source_reset <reset_signals>
                           -destination_clock <clock_signals> (-on_assert | -on_deassert)

OR

% config_rtlds -signal -inactive_destination
                       -source_reset <reset_signals>
                       -destination_clock <clock_signals> (-on_assert | -on_deassert)
```

The tool waives the following reset violations based on the inactive constraint:

- Source reset asserted while destination clock is inactive: RDC_RS_DFRS, RST_RS_RIDP
- Source reset de-asserted while destination clocks is inactive: RST_NO_SYNC

To generate the waivers, run `check_cdc -waiver -generate`. You can also use the following configuration command to choose to generate unconditional or conditional waivers:

```
set_cdc_inactive_destination_unconditional_waiver (false | true)
```

Setting this configuration to `true`, generates unconditional waivers, and setting it to `false`, generates conditional waivers. By default, the tool generates unconditional waivers (`true`). With conditional waivers (`false`), you can verify that the resets are active only when the clock is gated.

Inactive destinations can be reported on the console or in a file using one of the following commands:

- `check_cdc -list inactive_destinations`
- `check_cdc -report inactive_destinations -file <filename> [-force]`

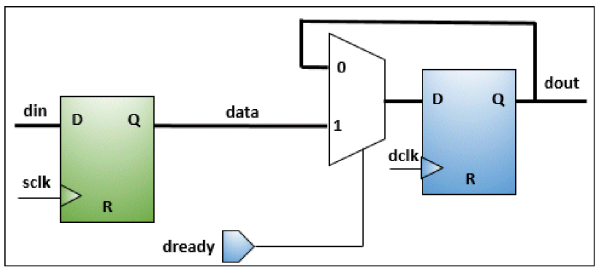
Externally Synchronized

When specifying the clock domain for a port, you can use the optional `-external_sync` switch to indicate that the port is correctly synchronized into the specified clock by means of control synchronizers present outside the block under analysis. You can also provide one or more source domains with the optional `-source_domain` switch. See the command below:

```
% config_rtlds -port <signal_list> -external_sync -destination_clock <clock_list> [-source_clock <clock>] [-add]
```

- Use `-source_clock` to specify the source clock from which these ports are synchronized.
- Use `-add` to define clock associations for the specified signals incrementally.

Currently, CDC uses this information exclusively for the detection of certain types of composite synchronization schemes, such as the automatic MUX_NDFF detection scheme and the user-defined glitch protector and sync enabler detection schemes. The purpose of this approach is to allow the control paths to be externally synchronized, which can be used as one of the building blocks of composite schemes.



Given the case above, the following command allows the tool to identify the structure as a MUX_NDFF scheme since it considers port `dready` equivalent to the output of the control synchronizer required for the scheme:

```
% config_rtlds -port dready -destination_clock dclk -external_sync -source_clock {sclk}
```

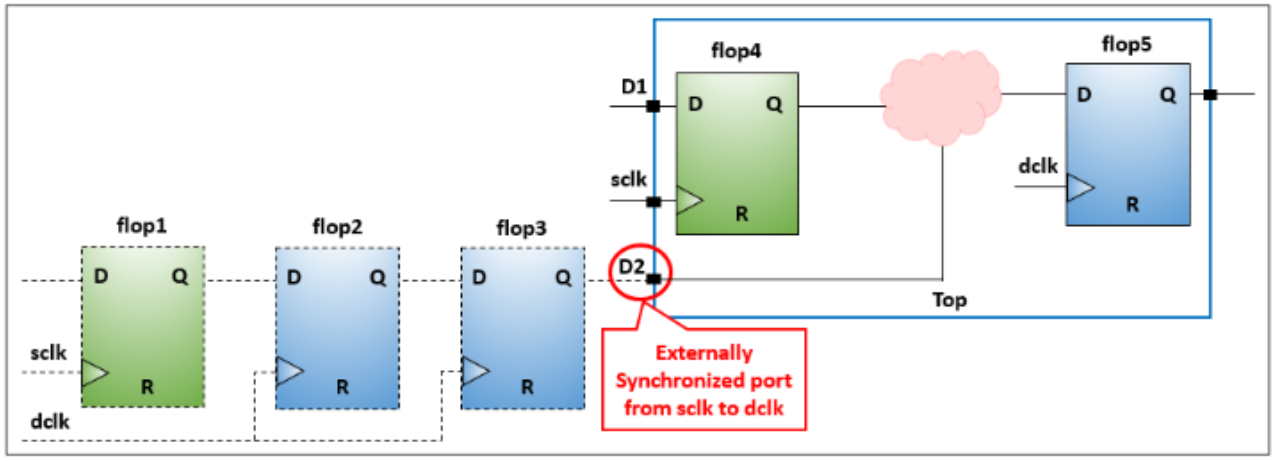
Without this definition, the clock domain crossing `sclk-dclk` remains uncovered, and the tool reports the corresponding CDC_NO_SYNC violation.

Example 1: Defining a primary input port as externally synchronized

In the example below, the port `D2` of the `Top` module has been defined as externally synchronized using the following command:

```
% config_rtlds -port D2 -external_sync -destination_clock {dclk} -source_clock {sclk}
```

This indicates that port `D2` of the `Top` module is being driven by an external synchronizer in `dclk` clock domain with source clock domain being `sclk`. Thus, the tool does not report a `CDC_NO_SYNC` violation between port `D2` and `flop5` but does report a violation between `flop4` and `flop5`.

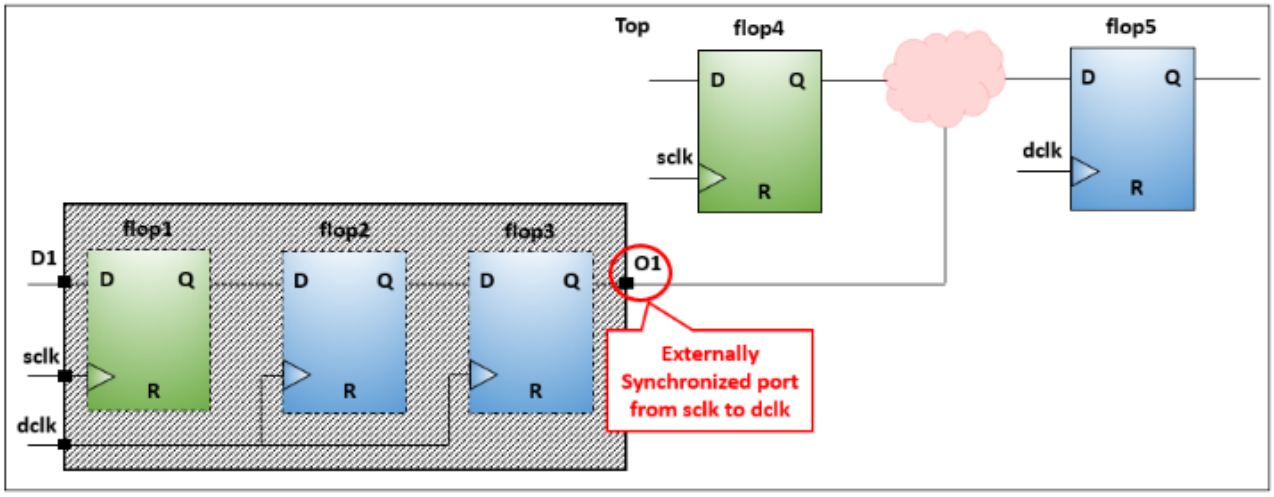


Example 2: Defining a blackbox output port as externally synchronized

In the example below, the port `O1` output of a blackbox module has been defined as externally synchronized using the following command:

```
% config_rtlds -port O1 -external_sync -destination_clock {dclk} -source_clock {sclk}
```

This indicates that the output port of the blackbox has been synchronized in the destination clock domain `dclk` with source clock `sclk` inside the blackbox itself. Thus, the tool reports a `CDC_NO_SYNC` violation between `flop4` and `flop5` only, but does not report a violation between `O1` and `flop5`.



Example 3: Defining a primary input as externally synchronized with multiple source clocks

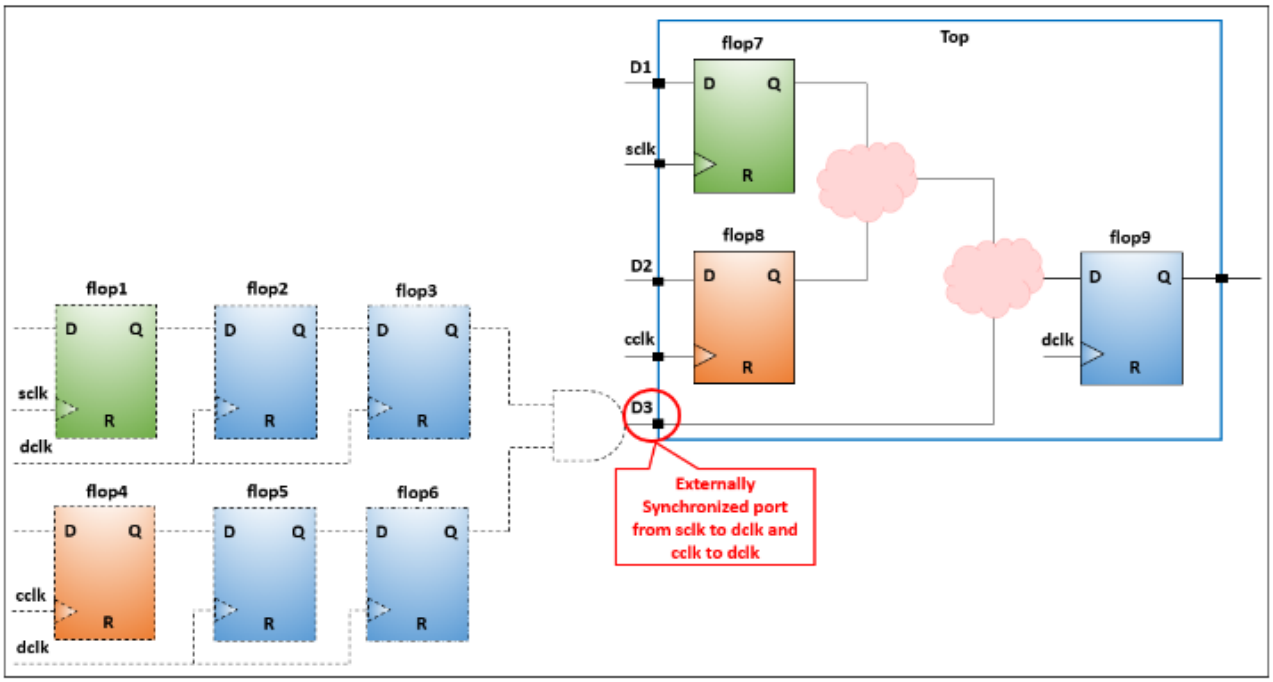
In the design below, the primary input port `D3` of the `Top` module has been defined as externally synchronized using the following commands:

```
% config_rtlds -port D3 -external_sync -destination_clock {dclk} -source_clock {sclk}
```

```
% config_rtlds -port D3 -external_sync -destination_clock {dclk} -source_clock {cclk} -add
```

The `-add` switch provides the option to add multiple source clocks to the externally synchronized port.

This indicates that the input port `D3` of the top module has been externally synchronized in the destination clock domain `dclk` with two source units, `sclk` and `cclk` clock domains.



Handling Unclocked Signals

Prior to the 2021.06 release, CDC automatically rated the clock associations of all boundary ports. However, 2021.06 introduces a new clock analysis algorithm, and the tool no longer tries to infer clock association for any port. Instead, you must explicitly specify the clock ratings.

Thus, all the primary input, output, or black-box input and output ports are reported as `unclocked` if no clock association has been explicitly specified, and all unclocked ports are considered synchronous to all declared clocks. Hence, no CDC crossings are reported to or from any of these unclocked ports to the rest of the design.

Internal Abstract Clocks

The 2021.06 release introduces two new internal abstract clocks that are user accessible and can be used to rate the ports:

- `sync_to_all` represents a clock that is synchronous to all the clocks in the design
- `async_to_all` represents a clock that is asynchronous to all the clocks in the design

Specifying Clock Association from Command Line

You can specify clock associations for ports (top input and output, black-boxed input and output, and signals with stopats) using the `config_rtlds -port` command as follows:

```
% config_rtlds -port signal_list -clock (clock_list | sync_to_all | async_to_all) [-add]
```

- Use `-add` to define the clock associations for the specified signals incrementally
- Use `sync_to_all` to associate the ports or signals to a clock synchronous to all other declared clocks in the design.
- Use `async_to_all` to associate the ports or signals to a clock asynchronous to all the other declared clocks in the design

⚠ The `check_cdc -clock_domain -port` command has been deprecated. Use `config_rtlds -port` instead.

```
Example:

Consider a port A which has been associated to two clock clk1 and clk2
% config_rtlds -port {A} -clock {clk1 clk2}

To add a clock association with the existing one, you can use the -add switch
% config_rtlds -port {A} -clock {clk3} -add

However, if you do not use the -add switch and specify another clock association for the same
port, the tool will ignore all the previous ratings and associate the port with the last
definition. Now port A will only be associated to clk4
% config_rtlds -port {A} -clock {clk4}

Rating port B with async_to_all clock
% config_rtlds -port {B} -clock async_to_all

Rating port C with sync_to_all clock
% config_rtlds -port {C} -clock sync_to_all
```

If you want to rate a set of ports, for example, all the primary inputs/outputs and so forth, as asynchronous to the entire design, use the following parameter:

```
% treat_data_ports_as_async ( all | none | primary_inputs | primary_outputs |
primary_inouts | bbox_inputs
| bbox_outputs}
```

By default, the parameter is set to `none`. But you can change the default value by either using the `config_rtlds -rule` command or your custom rule file.

⚠ The parameters `treat_boundaries_as_unclocked` and `treat_undeclared_as_sync` have been deprecated. Use `treat_data_ports_as_async` instead.

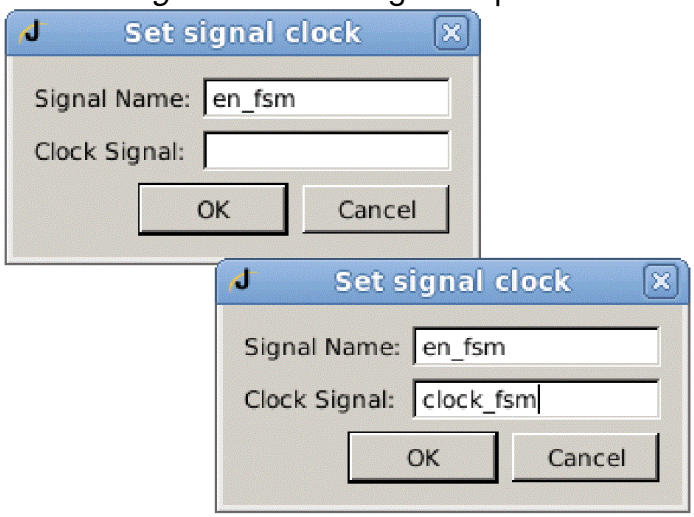
Specifying Clock Association from GUI

Constant ports are not affected by this analysis and remain in the sync domain. Otherwise, this flow handles static values and the tool tries to assign a valid clock domain for them.

To set the signal clock for an unrated signal, do the following:

1. From the *Port Configuration* tab, right click on a signal and choose the *Set Clock for Port* context-menu option.

   The *Set Signal Clock* dialog box opens with the *Signal Name* field pre-filled.



2. Specify the signal clock in the *Clock Signal* field.
3. Click *OK*.

> These commands must be run *before* `check_cdc -clock_domain -find.`

Clock Synchronicity Declaration

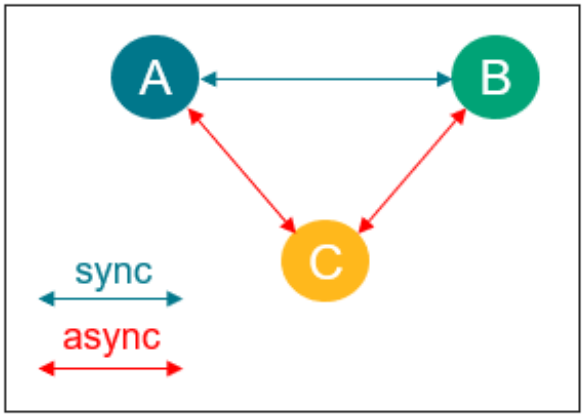
To define the relationship between declared clocks, you can use the following Jasper command. This command makes the flow consistent with the SDC command flow.

```
% config_rtlds -clock -group (-async |-sync |-physically_exclusive |-
logically_exclusive) {{clock_list}+} [-name <name>]
```

For example, consider the following command, which declares the relationship among three clocks `clkA`, `clkB`, and `clkC`:

```
% config_rtlds -clock -group -async {{clkA clkB} {clkC}}
```

Here, the tool considers both `clkA` and `clkB` as asynchronous to clkC. However, no relationship is set by the command between `clkA` and `clkB`. The relationship between `clkA` and `clkB` is determined by other commands or the default value of the parameter `all_clocks_sync_by_default`.



⚠ The synchronous clock relationships are non-transitive in both STA and non-STA (or async) mode. That is, if a derived clock is specified to be synchronous to another clock, this does not influence the relationship that might exist between the parent of the derived clock and the other clock.

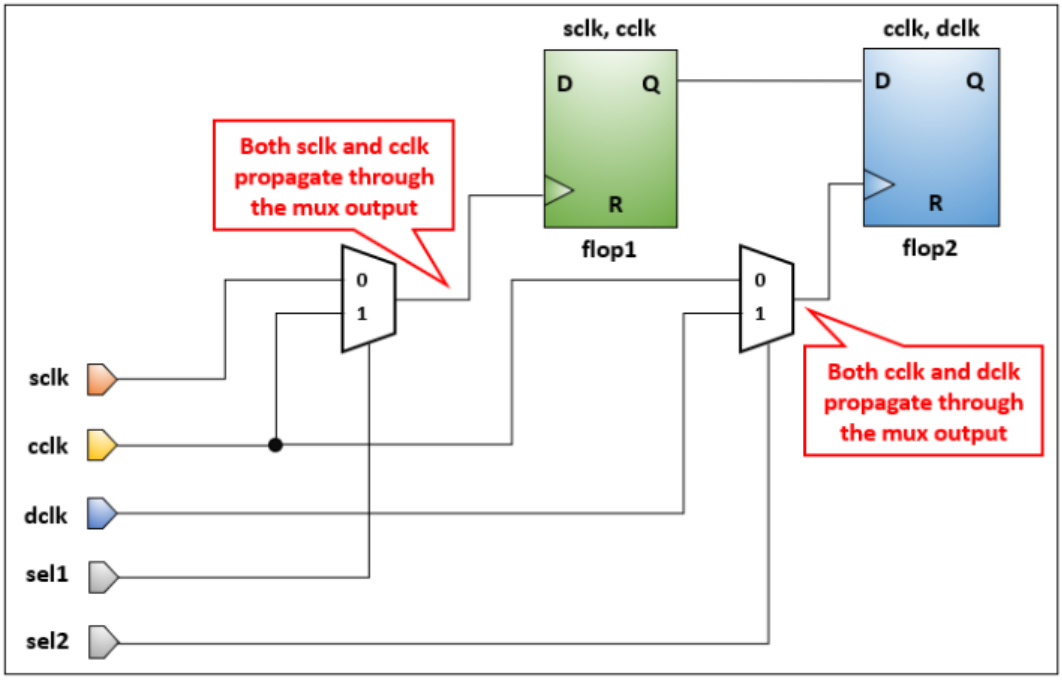
Handling Multi-Clock Scenarios

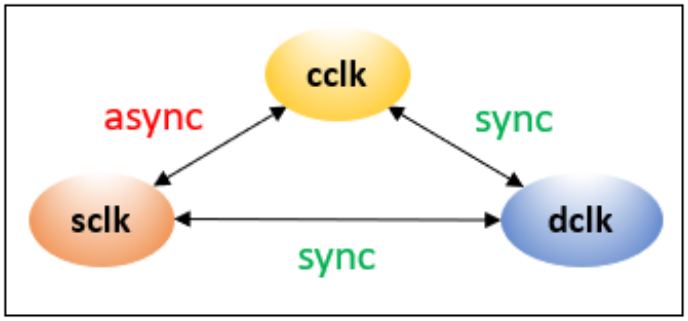
Beginning with the 2021.06 release, CDC pairs are reported between primary clocks only, filtering out all synchronous clocks. If multiple clocks control the source or destination of a CDC pair, the synchronous relationships are evaluated independently for each pair of primary clocks, and all the synchronous pairs of the clocks are omitted from the reporting.

Muxed Clocks

Example 1:

In the scenario below, `flop1` is driven by `sclk` and `cclk` clocks, and `flop2` is driven by `cclk` and dclk clocks. However, since clocks `cclk` and `dclk` and `sclk` and dclk have synchronous relationships defined, the tool reports the CDC pair between `sclk` and cclk clocks only because they are asynchronous to each other.





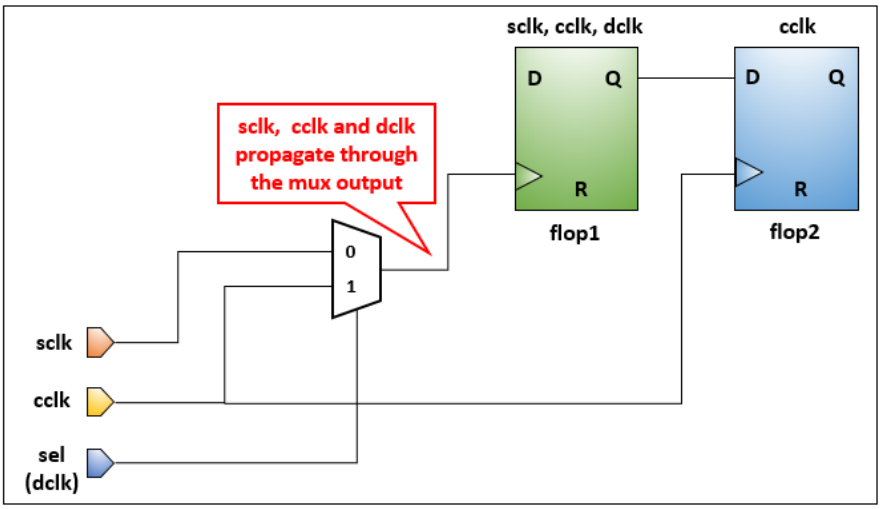
⚠ As the synchronous clock relationship is non-transitive, defining `cclk-dclk` and `sclk-dclk` as synchronous does not result in `sclk-cclk` clocks becoming synchronous to each other in both modes unless explicitly specified.

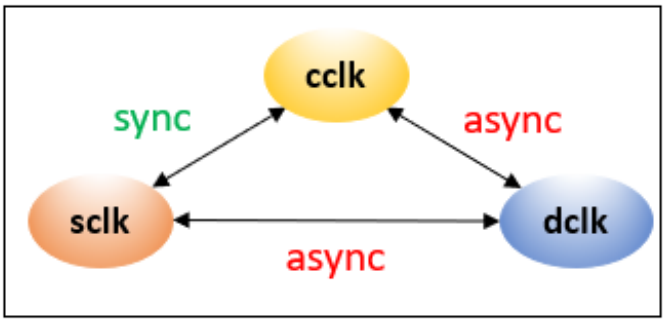
Example 2:

In the scenario below, `flop1` is being driven by output of a clocked mux with input `sclk` and cclk and a select line `sel` in the clock domain `dclk`, whereas, `flop2` is being driven by the clock `cclk` only. The tool propagates all three clocks `sclk, dclk, cclk` to the clocked mux output. However, since `sclk` and `cclk` are synchronous to each other, only one CDC pair will be reported between the clocks `dclk-cclk`.

Also, the tool will report a `CKS_NO_CONS` violation indicating the select line of the clocked mux is

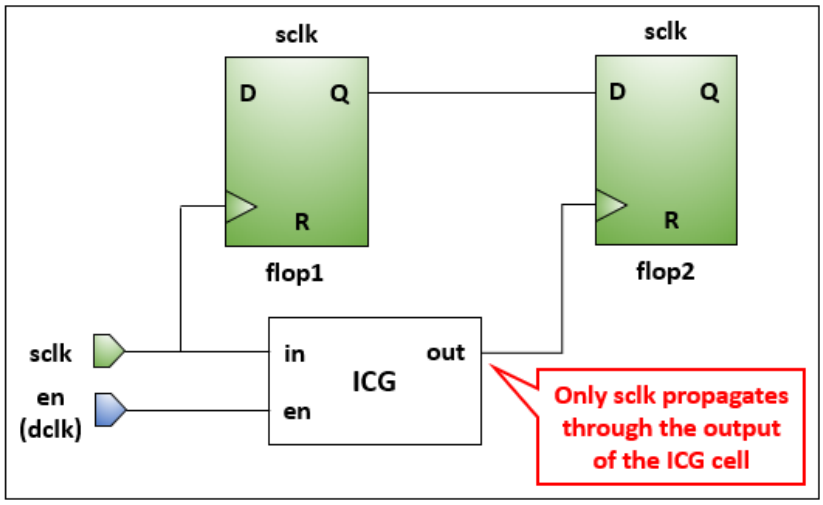
unconstrained.





Gated Clocks – ICG Cells

In the scenario below, `flop1` is driven by `sclk` clock, whereas `flop2` is driven by the output of the ICG, which has `sclk` as the input clock. The tool does not report any CDC pair between `flop1` and `flop2` because both are driven by the same clock signals.

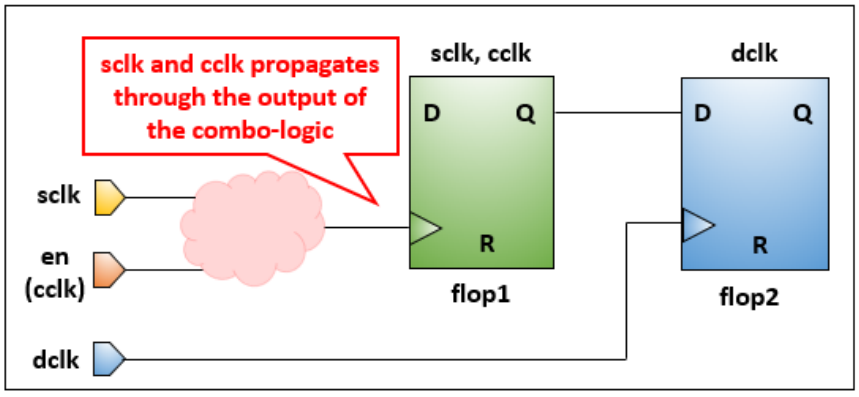


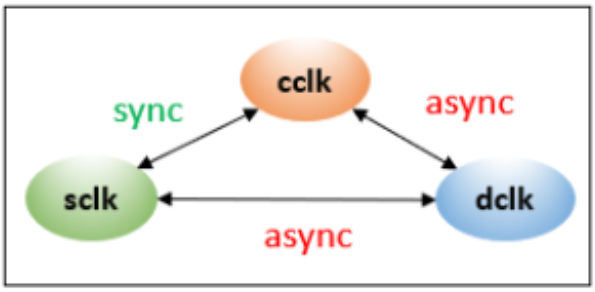
⚠ CDC does not propagate the clock domain of the `enable` signal through the ICG output. However, the tool reports a `CDC_PR_LTCH` violation indicating that different clocks are driving the inputs of the latch.

Discrete Gated Clocks

Example 1:

In the design below, `flop1` is driven by a gated clock with primary clock input `sclk` and enable signal associated to `cclk` clock whereas, `flop2` is driven by `dclk` clock. As both clocks `sclk` and `dclk` are asynchronous to `dclk`, the tool reports a CDC crossing between them. Also, since `sclk` and `cclk` are synchronous to each other, no violation is reported at the output of the combo-logic.

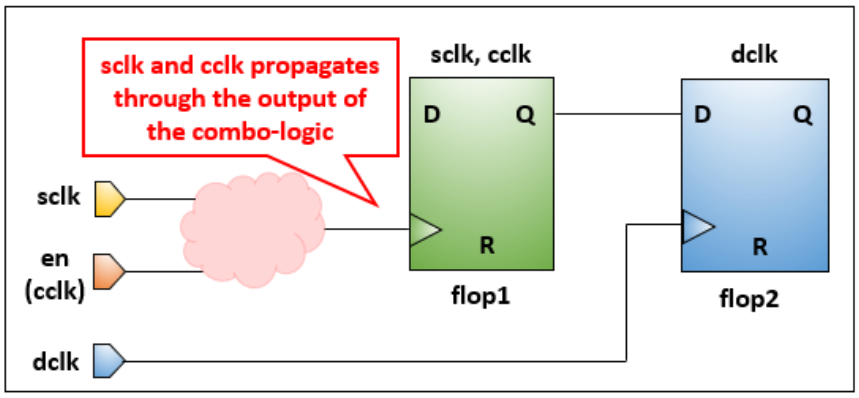


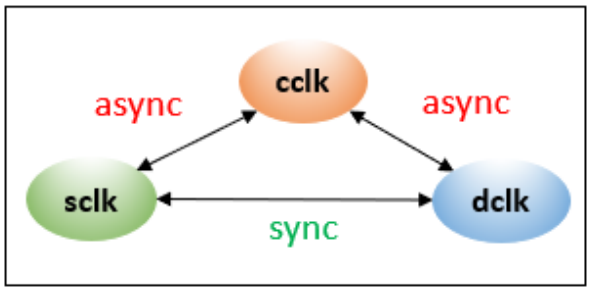


Example 2:

In the design below, `flop1` is driven by a gated clock with primary clock input `sclk` and an enable signal associated to clock `cclk`, whereas, `flop2` is driven by `dclk` clock. Since the tool does propagate the enable signal clock to the combo-logic output, both `sclk` and `cclk` are propagated.

Clocks `sclk` and `cclk` being asynchronous to each other causes the tool to report a `CLK_GT_ASYN` violation at the output of the combo-loop indicating the convergence of asynchronous clock signals. Also, as both `sclk` and `dclk` are synchronous to each other, a CDC crossing is reported only between `cclk` and `dclk`.



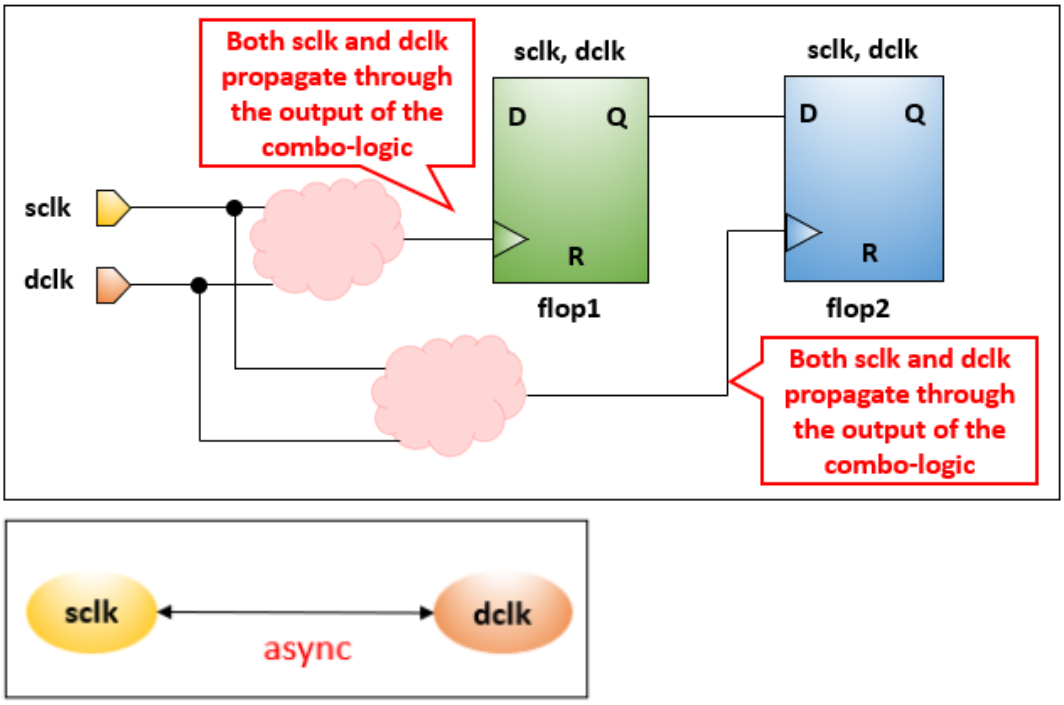


⚠ By default, the tool propagates the clock domains of the select or the enable signal in both muxed and gated clock (not latched) respectively.

Combinational Clocks

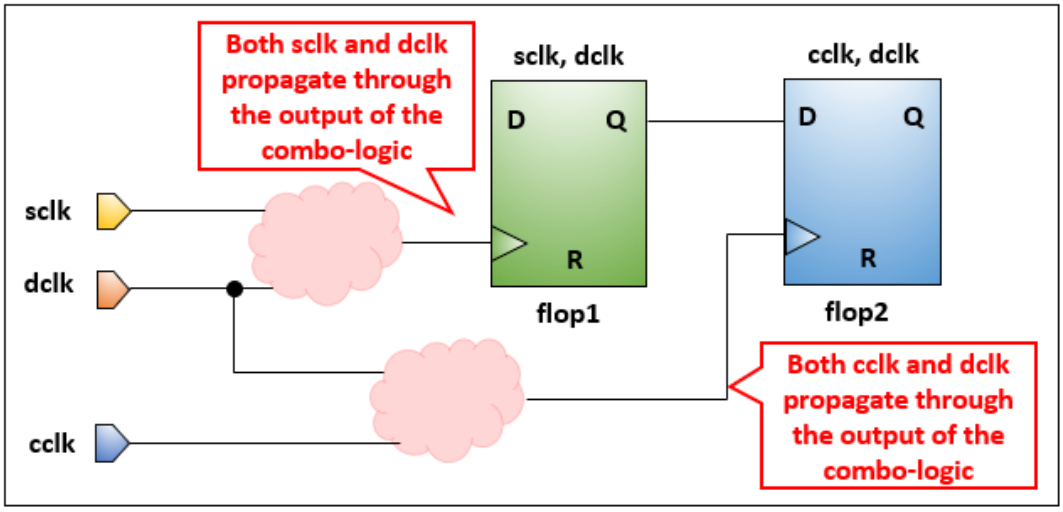
Example 1:

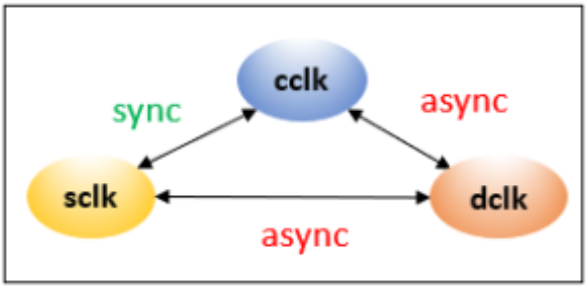
In the scenario below, both the flops (`flop1` and `flop2`) are driven by the same asynchronous clocks but with different combo-logics. As the combo-logics are different, the output of each combinational logic would also be different, causing the tool to report a CDC crossing between the two flops.



Example 2:

In the scenario below, `flop1` is driven by two converging clocks `sclk` and `dclk` whereas, `flop2` is driven by two converging clocks `dclk` and `cclk`. Since, `sclk` and cclk clocks are synchronous to each other, the tool reports CDC crossings only between (`sclk` and `dclk)` and between (`dclk` and `cclk)`.





⚠ The parameter `auto_join_muxed_gated_clock` has been deprecated. The tool no longer joins the parent clock signals of muxed, discrete gated, or combinational clocks into new clock domains.

Also, the tool reports a `CLK_IS_CONV` violation to highlight the convergence of asynchronous clocks.

Also, if you want to view the clock domain of a signal, you can use the following command:

```
% check_cdc -debug -show_clock_domain <signal_name>.
```

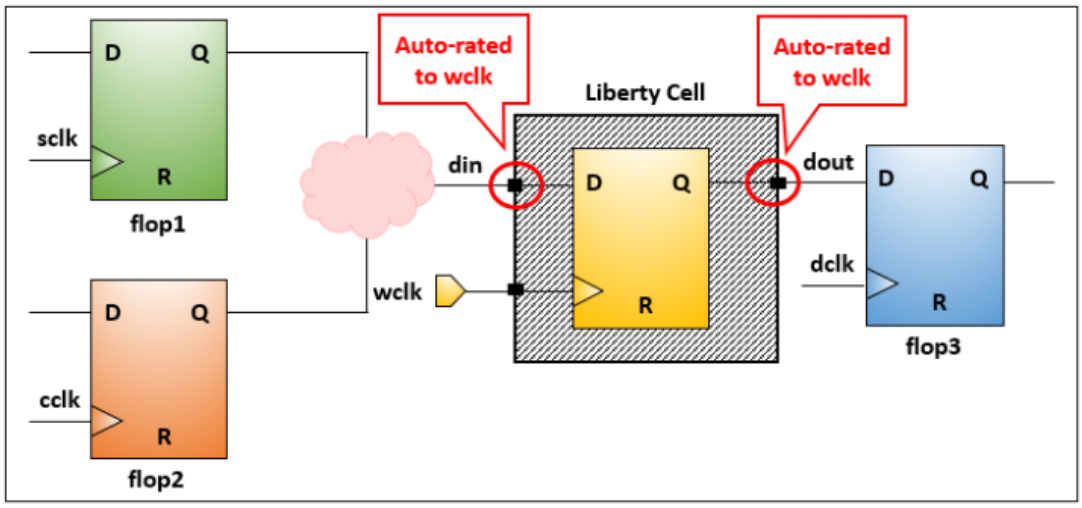
⚠ Refer to the "Running CDC Analysis" chapter for more details.

Liberty Cells

Example 1 : Sequential arc from clock input to data input and data output

In the design below, the `din` and `dout` ports of the Liberty cell automatically get rated to the `wclk` domain by the tool. Thus, the tool reports a `CNV_ST_GLCH` violation between `din` and `flop1` and `flop2`

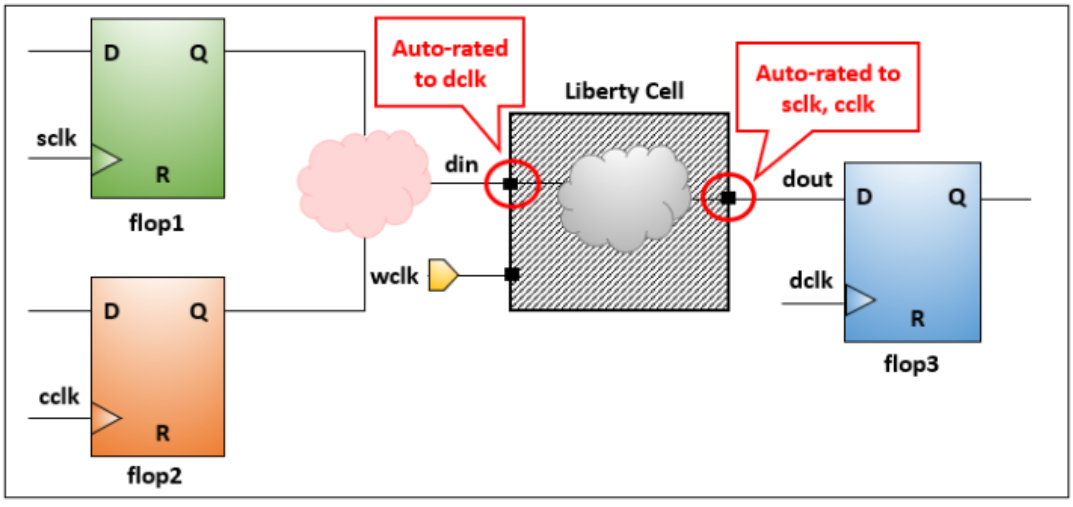
and a CDC_NO_SYNC violation between dout and flop3.



Example 2 : Combinational arc from data input to data output

In the design below, the tool cross rates the ports. The din port of the Liberty cell is automatically rated to the dclk clock domain, whereas the dout port is automatically rated to sclk, cclk clock domains. This allows the tool to catch CDC-related violations on both sides of the Liberty cell. At the din port, a CNV_ST_GLCH violation is reported, whereas on the dout port, a CDC_NO_SYNC violation is reported.

However, if the cross rating was not provided and din was associated to sclk, cclk clock domains the tool may end up missing the CNV_ST_GLCH violation as all the liberty cells are black-boxed by the tool and only clock association information is extracted under the current behavior.



Handling Black Boxes

To create an abstract model for blackboxed modules, create module-based clock and reset ratings for blackbox ports. To rate a list of ports with a clock and/or reset or with regard to any input data port, use the following syntax:

```
% config_rtlds -port <signal_list> -module <module_name> -input <signal_list> | -clock
<signal_list> | -reset <signal_list> [-add]
```

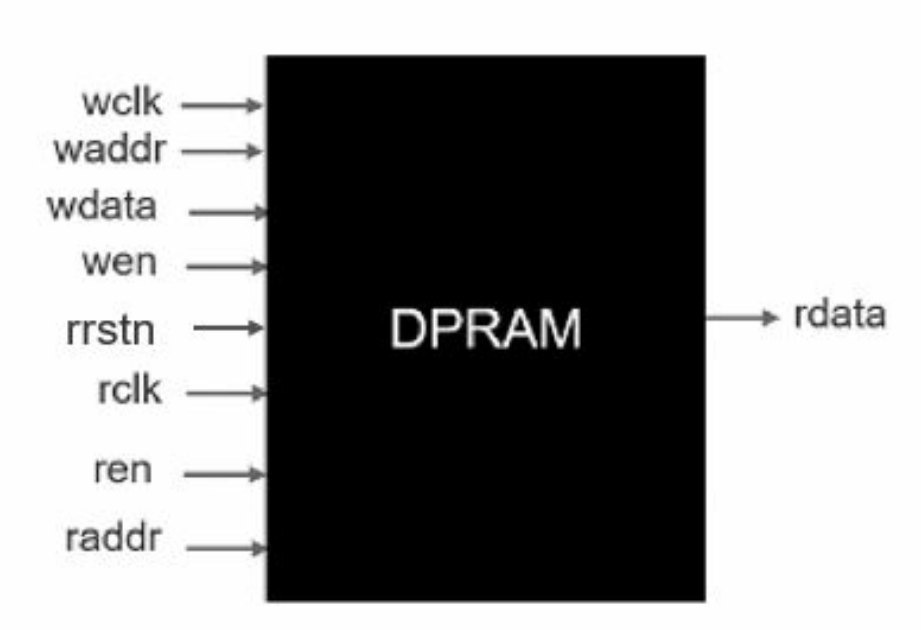
- Use the combination of `-module` and `-clock` to rate the module data ports specified in the `-port` argument with the clocks connected to the module clock ports specified in the `-clock` argument. These ratings are applied to all instantiations of the given module by rating the instance data ports to the primary clocks that are actually connected to the instance's clock ports.
- Use the combination of `-module` and `-reset` to create a reset rating between the data ports specified in the `-port` argument and the reset signals connected to the module reset ports specified in the `-reset` argument.
- Use the combination of `-module` and `-input` to copy both clock and reset rating, if it exists, from the pin specified in the `-input` argument to the ports specified in the `-port` argument.

See the following examples:

```
// Associate read signals with the same clock domain as the clock driving 'rclk'
% config_rtlds -port {raddr ren rdata} -module dpram -clock rclk

// Associate read signals with the same reset domain as the reset driving 'rrstn'
% config_rtlds -port {raddr ren rdata} -module dpram -reset rrstn

// Associate write signals with the same clock and reset domain as wclk
% config_rtlds -port {waddr wdata wen} -module dpram -input wclk
```



User-Defined and Custom Synchronizers

In general, the tool identifies CDC synchronizers when you run `check_cdc -scheme -find` and Reset Synchronizers when you run `check_cdc -reset -find`. However, in some cases, the tool might be unable to identify a scheme even when it belongs to one of the automatically detectable types. Any such unidentified synchronizer is reported as a violation, which adds to the noise. To reduce this noise, you can add unidentified synchronizers as user-defined schemes.

The remainder of this section offers additional information about user-defined and custom schemes as follows:

- User-Defined Module-Based Synchronizers
- User-Defined Instance-Based Synchronizers
- Custom Synchronizers

User-Defined Module-Based Synchronizers

Use the module-based approach when the synchronizer is contained in its own RTL module. In this case, synchronizer detection is based on the detection of any instance of that particular module. For module-based specification of user-defined or custom schemes, provide the key signals of the scheme within the scope of the module. Both ports and internal signals in the module can be used in the specification of the scheme.

Module-based synchronizers can be added to the design setup from the command line as well from the GUI by mapping the module signals/ports to the formal signal list.

Adding Module-Based Synchronizers from the Command Line

To add a module-based synchronizer in your design, use the following command:

```
check_cdc -scheme -add <scheme_name> -module <module_name> -map {{formal_signal signal}+}
```

In the command above,

- `scheme_name` is one of the pre-defined schemes that you want to add in your design.
- `module_name` is the name of the module that contains the scheme RTL.
- `formal_signal` names the relevant signals inside each synchronizer. The tool maps these to actual RTL signals in your design for each instance of a synchronizer.
- `signal` is the name of the port/signal inside the module that is to be mapped to the specific formal signal.

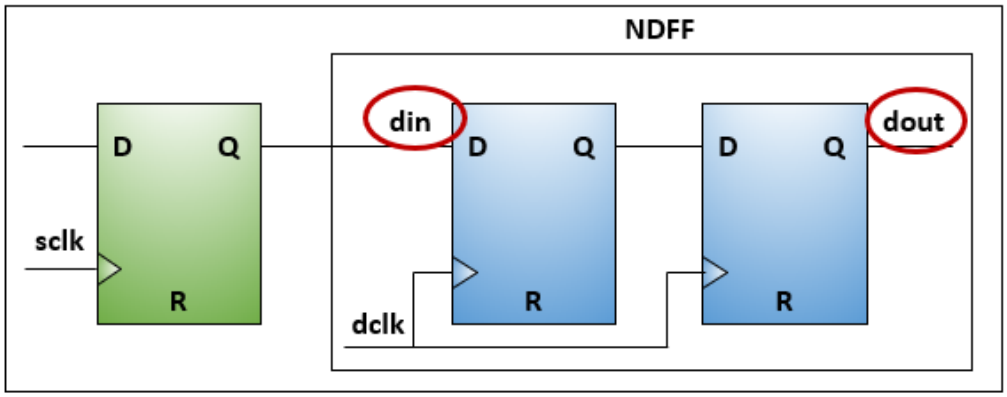
Mapping Synchronizer Schemes

This section describes how to map synchronizer logic to one of the pre-defined synchronizers using the `check_cdc -scheme -add` command.

⚠ All user-defined and custom schemes need to be specified before running `check_cdc -scheme -find`.

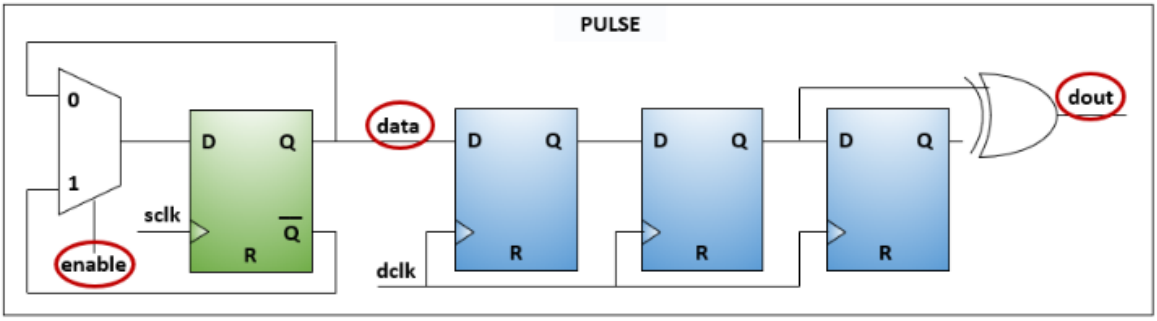
NDFF

Formal Signals	`data, dout`
Associated Command	`check_cdc -scheme -add ndff -module ndff -map {{dout dout} {data din}}`



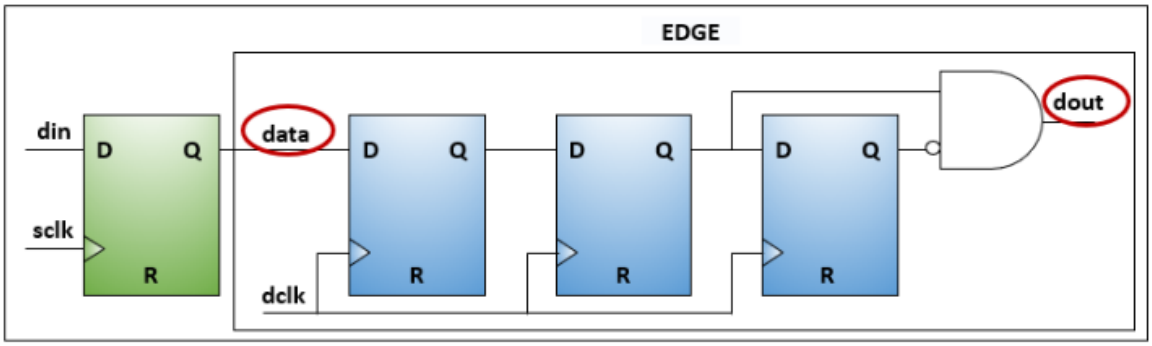
PULSE

Formal Signals	`data, dout, enable`
Associated Command	`check_cdc -scheme -add pulse -module PULSE -map {{data data} {dout dout} {enable enable}}`



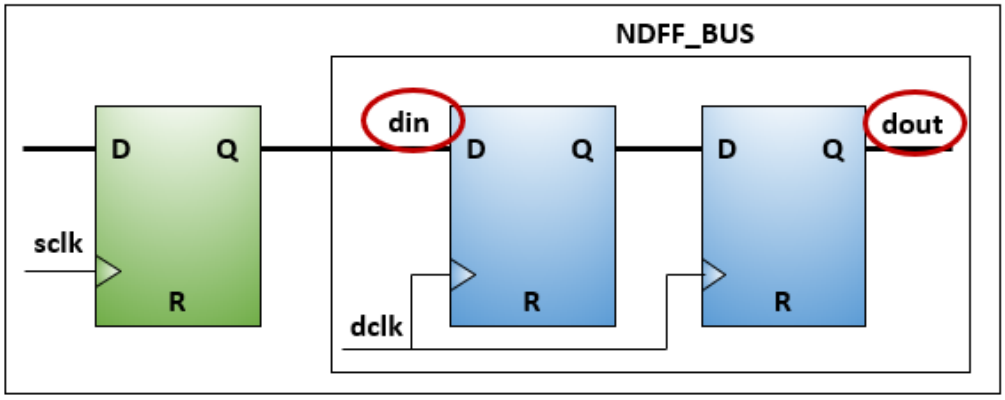
EDGE

Formal Signals	`data, dout, activity (high | low), sensitivity (rising | falling)`
Associated Command	`check_cdc -scheme -add edge -module EDGE -map {{data data} {dout dout} {activity high} {sensitivity rising}}`



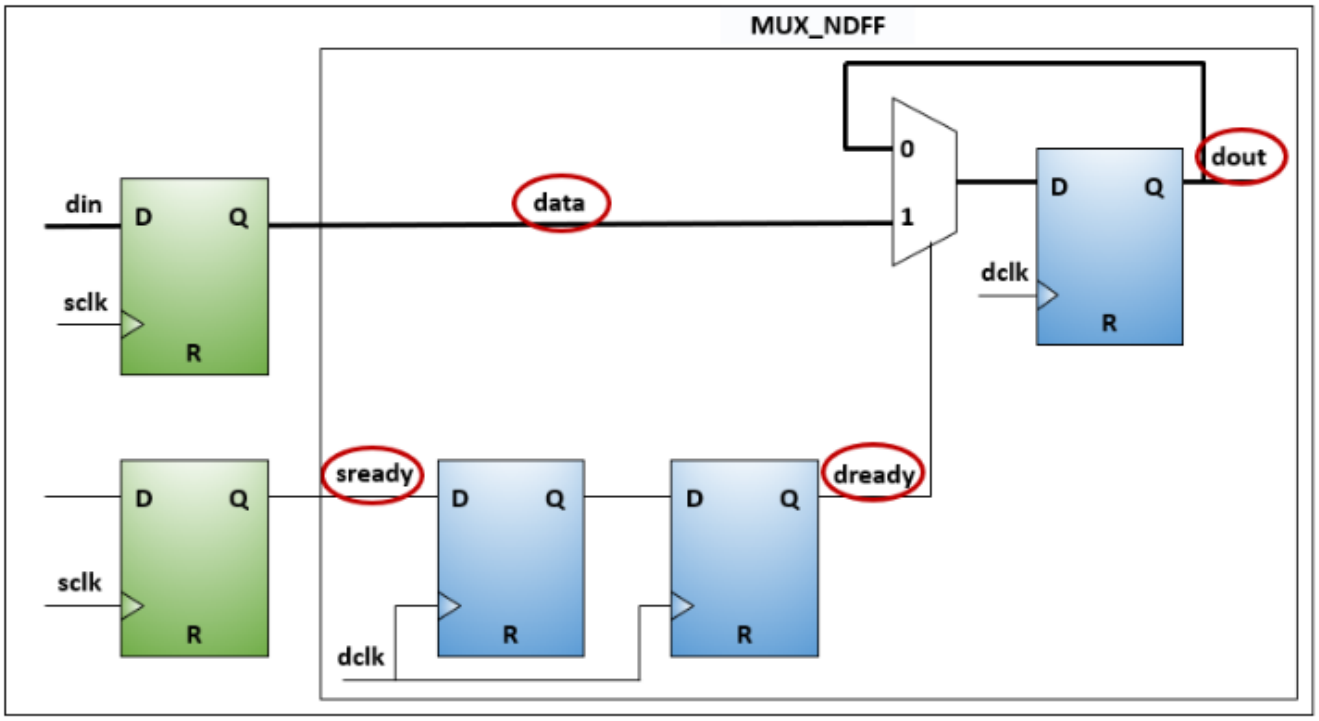
NDFF_BUS

Formal Signals	`data, dout`
Associated Command	`check_cdc -scheme -add ndff_bus -module NDFF_BUS -map {{dout dout} {data din}}`



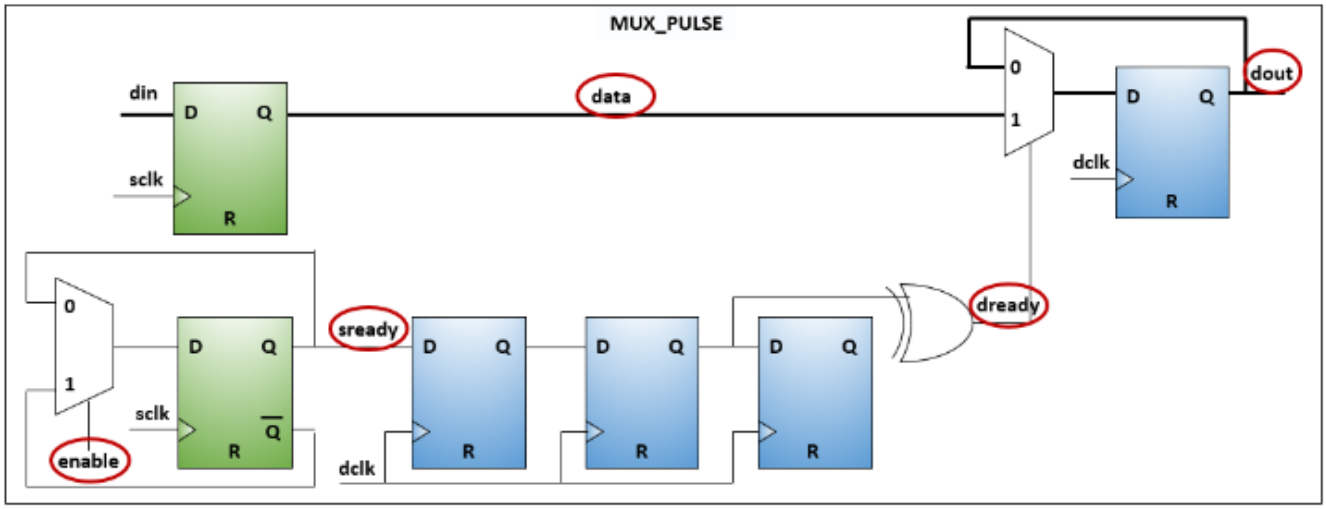
MUX_NDFF

Formal Signals	`data, dout, sready, dready`
Associated Command	`check_cdc -scheme -add mux_ndff -module MUX_NDFF -map {{data data} {dout dout} {sready sready} {dready dready}}`



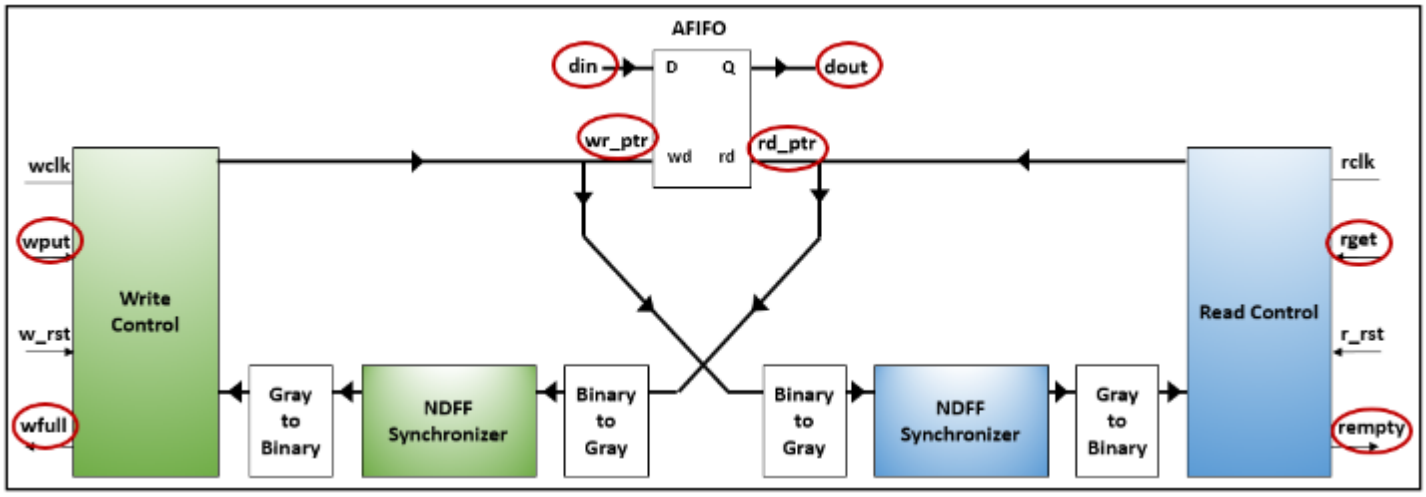
MUX_PULSE

Formal Signals	`data, dout, sready, dready, enable`
Associated Command	`check_cdc -scheme -add mux_pulse -module MUX_PULSE -map {{data data} {enable enable} {sready sready} {dready dready} {dout dout}}`



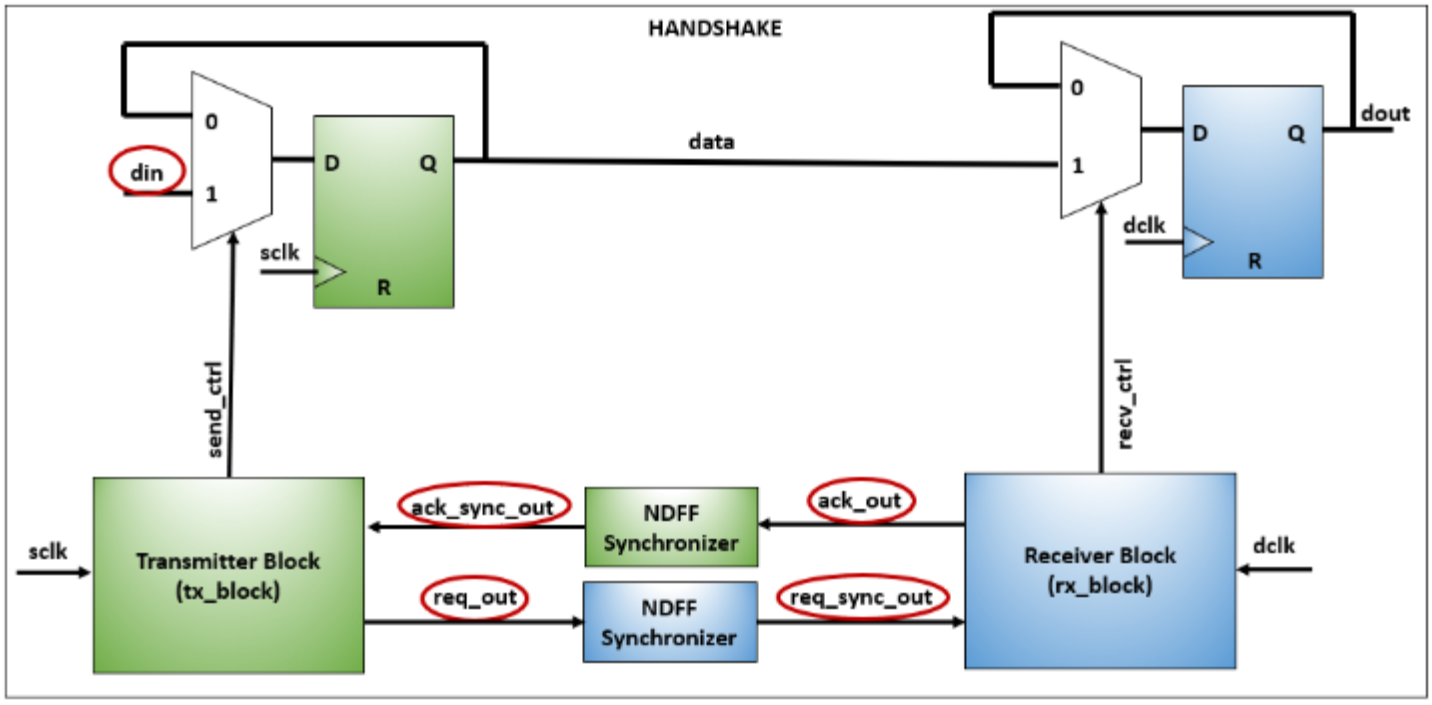
FIFO

Formal Signals	`rdata, wdata, wptr, rptr, wfull, rempty, winc, rinc`
Associated Command	`check_cdc -scheme -add fifo -module AFIFO -map {{rdata dout } {wdata din} {wptr wr_ptr} {rptr rd_ptr} {wfull wfull} {rempty rempty} {winc wput} {rinc rget}}`



HANDSHAKE

Formal Signals	`data, sreq, dreq, dack, sack`
Associated Command	`check_cdc -scheme -add handshake -module HANDSHAKE -map {{data din} {sreq tx_block.req_out} {dreq rx_block.req_sync_out} {dack rx_block.ack_out} {sack tx_block.ack_sync_out}}`

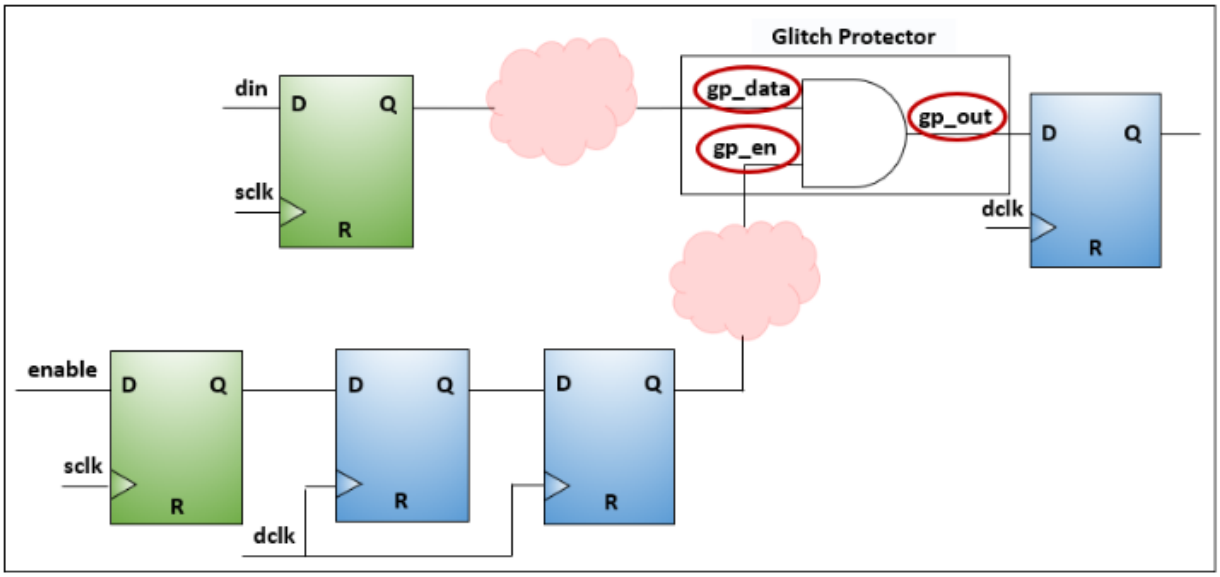


GLITCH PROTECTOR

The tool detects a glitch protector scheme only when at least one unit connected to the control input is the output of a control synchronizer. After a glitch protector synchronizer has been added, no `CDC_PR_FOUT` or `CDC_PR_LOGC` violations are reported for the pairs related to that synchronizer.

Formal Signals	`dout, gpdata, control`
Associated Command	`check_cdc -scheme -add glitch_protector -module gp -map {{ dout gp_out } { gpdata gp_data } { control gp_en }}`

⚠ Mandatory formal signals for glitch protector synchronizer include `dout` only. You might also need to supply the `GPData` and `control` signals if the tool cannot infer them.



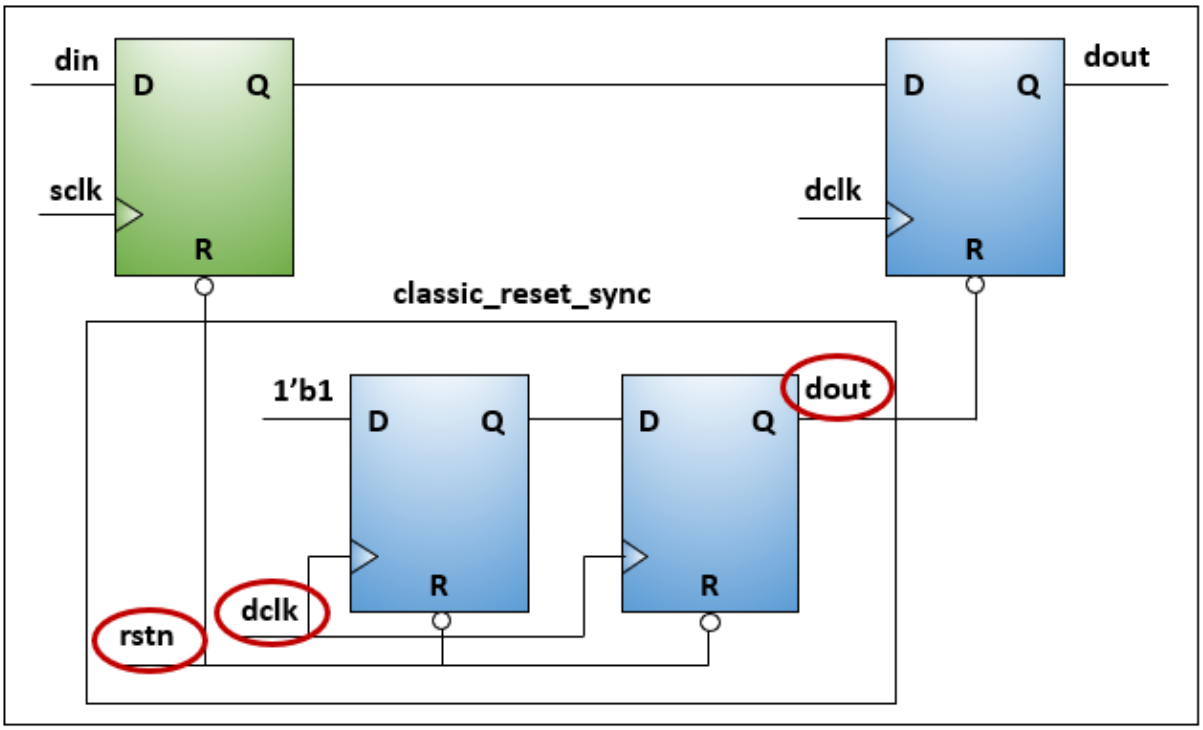
Note:

- If the source domain of the control signal is not synchronized with the source domain of the data, the tool generates an `SYN_GP_ASIP` violation.
- If CDC units connected to the `control` signal are not synchronized with the destination, the tool generates an `SYN_GP_ASOP` violation.
- You can define specific instances of a glitch protector module as a valid synchronizer scheme.

CLASSIC RESET SYNCHRONIZER

Formal Signals	`dout/snrst, srst, dclk`

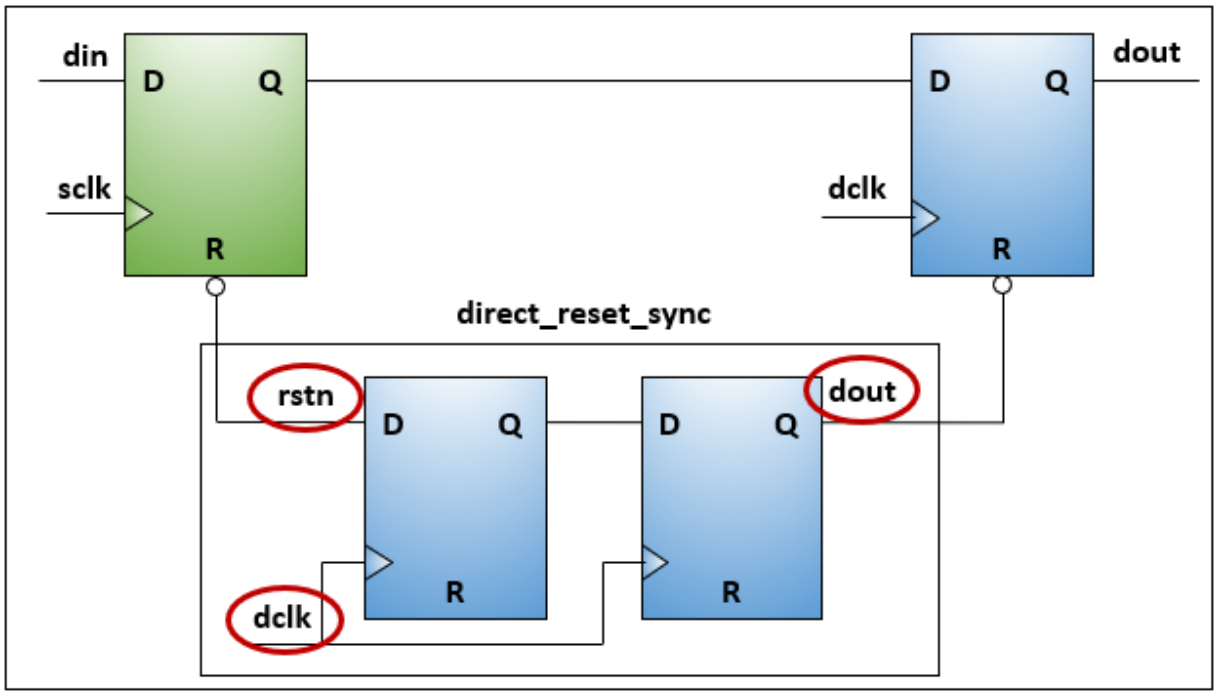
Associated Command	`check_cdc -scheme -add reset -module classic_reset_sync -map {{dout dout} {srst rstn} {dclk dclk}}`
	or
	`check_cdc -scheme -add reset -module classic_reset_sync -map {{snrst dout} {srst rstn} {dclk dclk}}`



DIRECT RESET SYNCHRONIZER

Formal Signals	`dout/fsrst, srst, dclk`

Associated Command	`check_cdc -scheme -add reset -module direct_reset_sync -map {{dout dout} {srst rstn} {dclk dclk}}`
	or
	`check_cdc -scheme -add reset -module direct_reset_sync -map {{fsrst dout} {srst rstn} {dclk dclk}}`



⚠

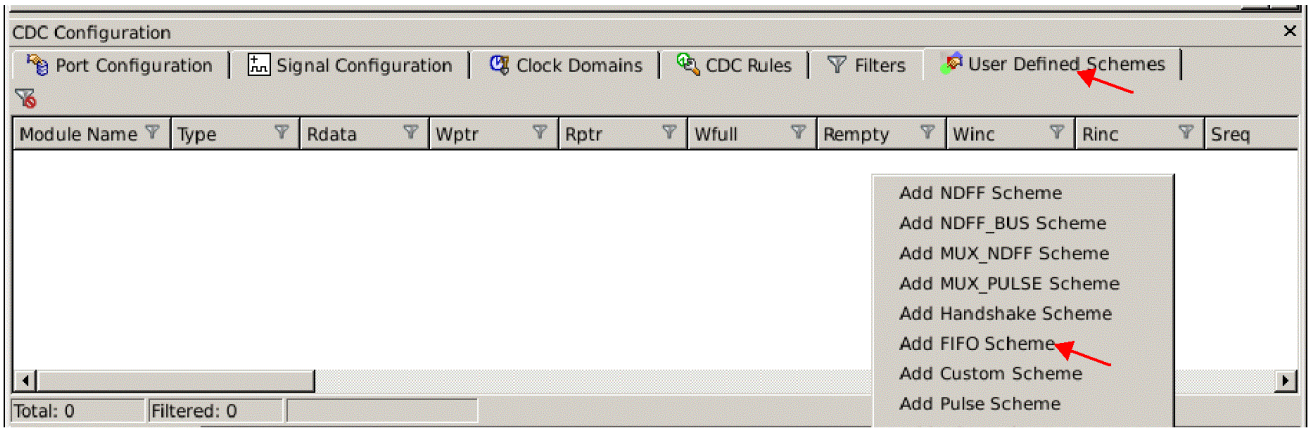
- The tool can detect the same module or instance as direct and classic reset synchronizer if both the formal parameters `snrst` and `fsrst` have been set.
- The same module or instance can contain multiple `snrst` and `fsrst` outputs and can also be used in conjunction with the previous formal parameter `dout`
- The tool can detect the same module declared as both user-defined NDFF and classic/direct reset synchronizer schemes. With the current tool behavior, this may result in some of the reset schemes being counted as NDFF scheme.

Adding Module-Based Synchronizers from the GUI

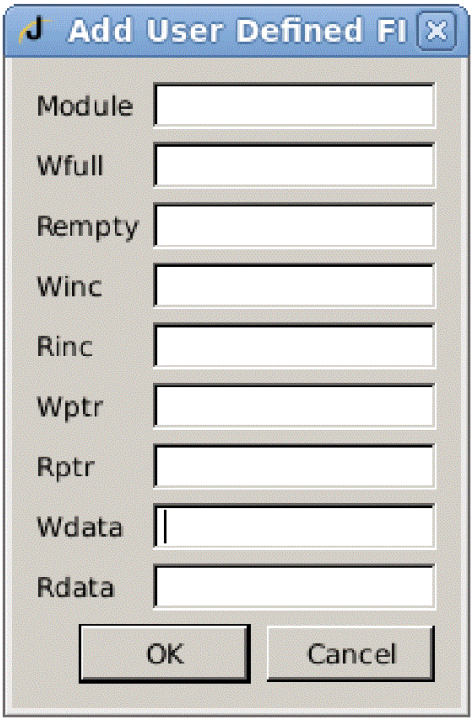
The module-based synchronizers can also be added from the GUI. The following example shows how to add a FIFO. However, other synchronizers can be added in similar fashion.

To add module `AFIFO` as a user-defined FIFO synchronizer, do the following:

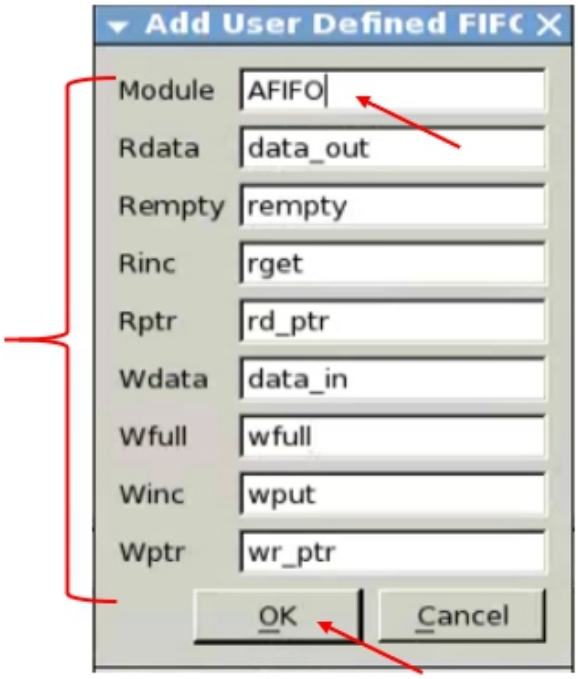
1. Access the *CDC Configurations* tab *User-Defined Schemes* sub-tab.
2. Right click on the blank table and select *Add FIFO Scheme*.



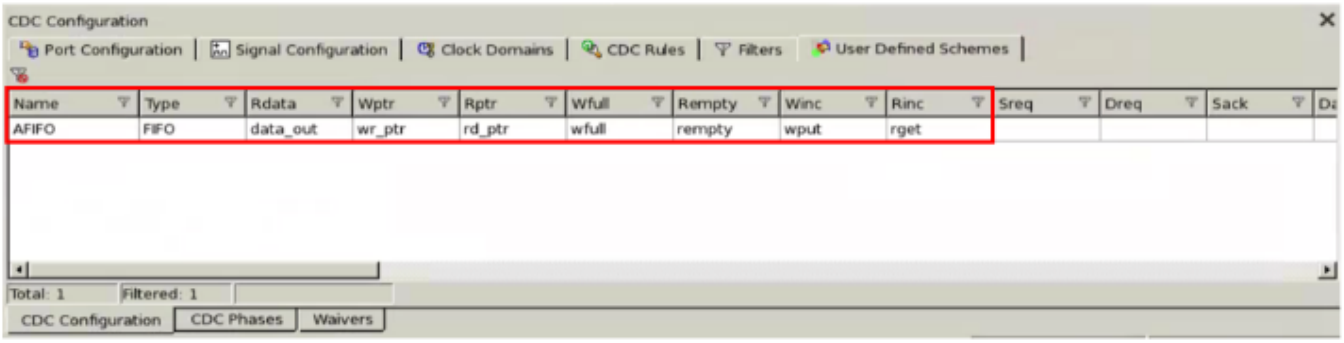
The *Add User-Defined FIFO Scheme* dialog box box opens with the formal parameter names listed.



3. Enter the module name in the *Module* field.
4. Enter the actual RTL signal names in the given fields (see the following figure for a completed dialog box).



5. Click *OK*.
   The tool adds the module `AFIFO` as a user-defined FIFO synchronizer, and it appears in the *User-Defined Schemes* table.



Henceforth, the tool considers all instances of `AFIFO` a FIFO synchronizer and automatically generates the FIFO synchronizer functional checks for every instance of `AFIFO`.

User-Defined Instance-Based Synchronizers

Use the instance-based approach when the synchronizer logic is not contained in a single RTL module. For instance-based specification of user-defined or custom schemes, provide the key signals of the scheme within the scope of the top-level module.

For instance-based schemes, any CDC pair whose source or destination is one of the signals provided during the specification is covered by the scheme. For example, you can manually add an instance-based synchronizer by mapping the actual RTL signals of an instance to the formal signal names.

To add an instance-based scheme in your design, use the following command:

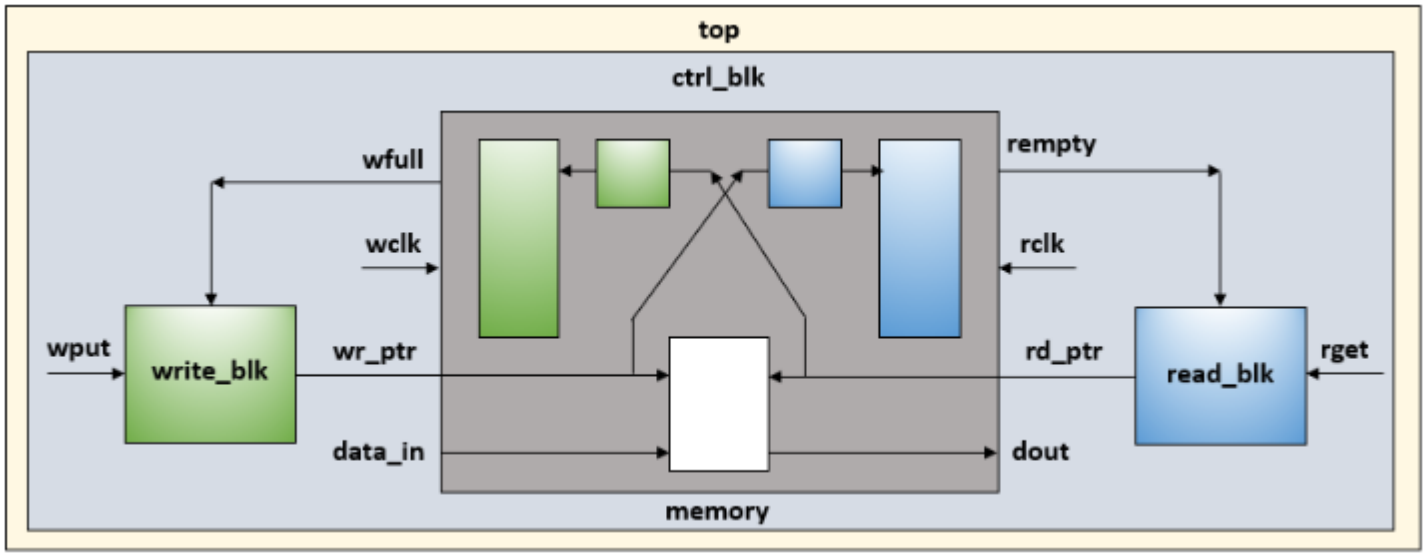
```
check_cdc -scheme -add <scheme_name> -map {{formal_signal signal}+}
```

In the command above,

- `scheme_name` is one of the pre-defined schemes that you want to add in your design.
- `formal_signal` names the relevant signals inside each synchronizer. The tool maps these to actual RTL signals in your design for each instance of a synchronizer.
- `signal` is the name of the port/signal with full hierarchy that is to be mapped to the specific formal signal.

See the following example, which adds an instance-based FIFO in the design setup:

```
check_cdc -scheme -add fifo -map {{rdata ctrl_blk.memory.dout}
                                  {wdata ctrl_blk.memory.data_in}
                                  {wptr ctrl_blk.write_blk.wr_ptr}
                                  {rptr ctrl_blk.read_blk.rd_ptr}
                                  {wfull ctrl_blk.memory.wfull}
                                  {rempty ctrl_blk.memory.rempty}
                                  {winc ctrl_blk.write_blk.wput}
                                  {rinc ctrl_blk.read_blk.rget}}
```



Synchronization Enabler Schemes

There is one exception to this rule; synchronization enabler schemes cover every CDC pair going from the specified source domain to the destination domain that is controlled by the synchronization enabler signal as well as the CDC control pair synchronizing the enabler signal (if it is the case).

Use the synchronization enabler scheme to synchronize a data path between one or more source domains and a specified destination domain. With this scheme, you inform the tool of the signal that controls the data going from one or more domains to another.

Adding an Instance-Based Sync Enabler

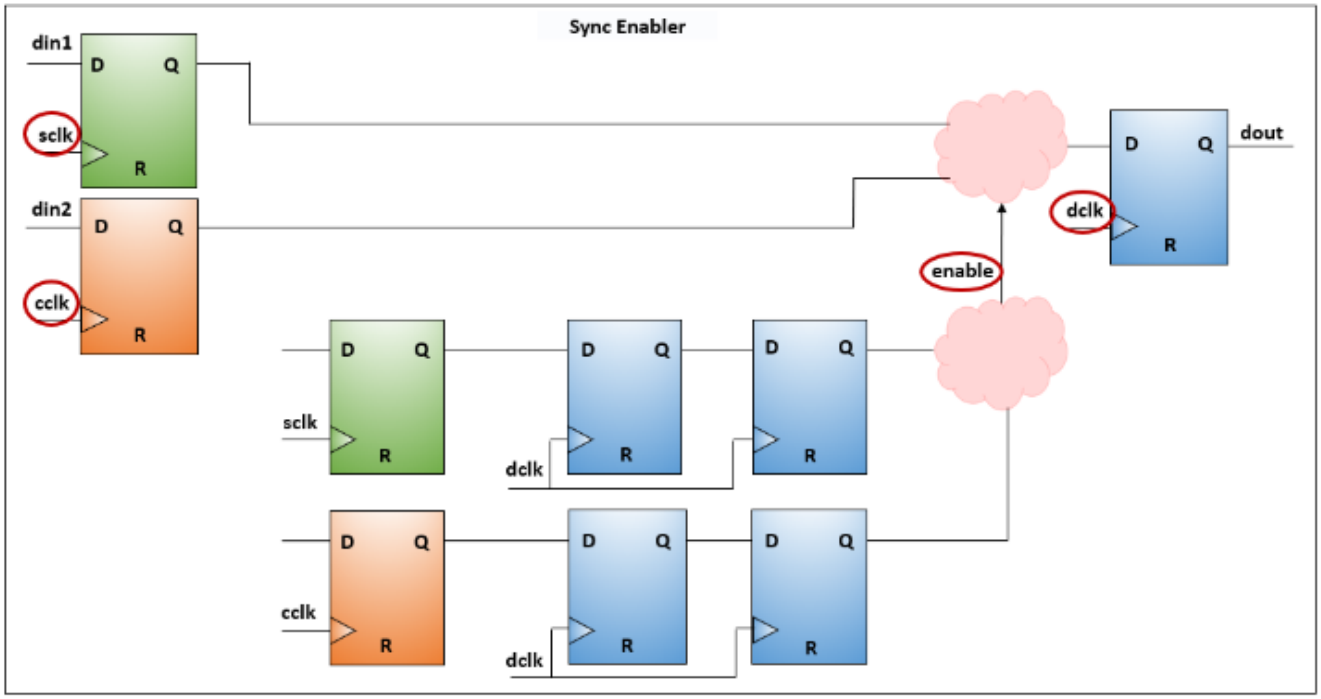
For user-defined `sync_enabler` schemes, the tool explores the fanin cone of the provided sync enabler signal until it reaches a boundary defined by one or more of the following elements:

1. Constant or static CDC units
2. CDC units known as the output of control synchronizers from any of the specified source domains into the destination domain
3. Externally synchronized ports associated with the destination domain

Note:

- The presence of an element of the second type automatically renders the sync enabler scheme structurally invalid.
- For the third element type above, at least one synchronizer from each source domain into the destination domain must be found for the sync enabler to be structurally valid.

Formal Signals	`enable, (srcdomain | srcdomainlist), dstdomain`
Associated Command	`check_cdc -scheme -add sync_enabler -map {{enable enable} {srcdomainlist {sclk cclk}} {dstdomain dclk}}`



When you run `check_cdc -scheme -find`, the tool processes all the synchronization enabler schemes after every other type of supported scheme and considers any remaining uncovered data pairs from the source domain(s) to the destination domain covered if they are controlled by the specified enable signal. The tool shows these pairs as *Passed* without any associated violation even if they have a `CDC_PR_LOGC` violation. Conversely, invalid synchronization enabler schemes fail with an `SYN_SE_INVL` or `CDC_NO_SYNC` violation.

⚠ If the tool is unable to determine a valid data path qualified by the user-specified enable signal, it issues a warning informing you that the sync enabler has not been added to the design.

Note:

- The tool assigns the destination unit name to `sync_enabler` schemes.
- The tool validates the enable signal during the detection phase and not during the specification phase.

Custom Synchronizers

The CDC App supports twelve pre-defined synchronizer schemes. Ten of these, NDFF, NDFF_BUS, MUX_NDFF, MUX_PULSE, Handshake, FIFO, Pulse, Edge, and two reset synchronizers (Classic and Direct), can be automatically identified by the tool or defined by the user. The other two, glitch protector and synchronization enabler, are strictly user-defined.

In some scenarios, designs might use synchronizer cells that do not map to any of the pre-defined types. In such cases, the tool reports violations since it does not recognize these synchronizer cells as valid synchronizers. To avoid this, you can add these cells as custom synchronizer schemes so that the tool accepts them.

To create a custom scheme, use the following command syntax:

```
check_cdc -scheme -create scheme_type -formal_list {formal_signals}
```

- `formal_signals` are names for the relevant signals inside each synchronizer. it is important to have `dout` (output of the synchronizer) in the list. The tool maps these to the actual RTL signals in your design for each instance of a synchronizer.
- `scheme_type` provided must be a new type. If you provide a name that conflicts with one that already exists in the library, the tool errors out.

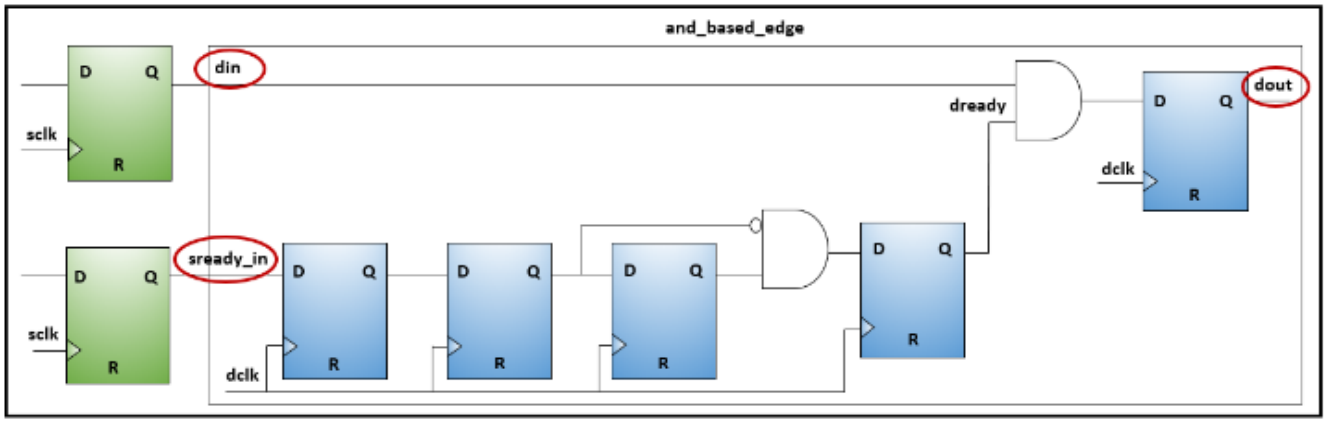
⚠ A valid formal signal name can have only letters, numbers, and underscores. If you provide an invalid formal signal, the tool issues an error.

Once the new scheme type is created, you can add it as a module-based scheme using the regular `check_cdc -scheme -add` command. Map the formal signals of the scheme to actual RTL signals in your design as follows:

```
check_cdc -scheme -add scheme_type -map {signal_mapping} -module module_name
```

Example:

To understand how you might add a custom scheme in the design and verify its functionality, consider the following example.

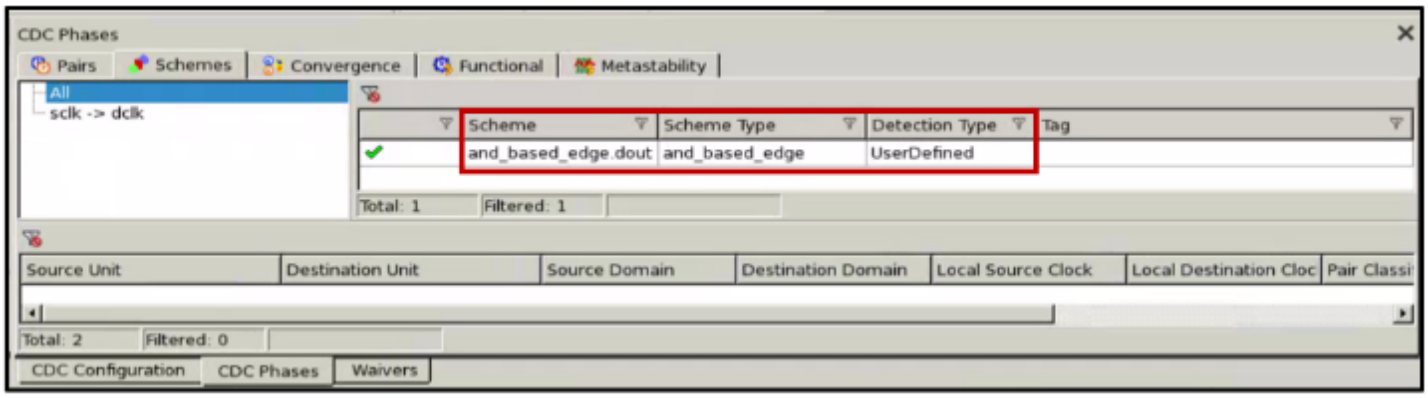


First, add the scheme to the tool scheme library using the following command:

```
check_cdc -scheme -create and_based_edge -formal_list {Dout Data Enable}
```

Then add this scheme as a module-based scheme before running the `check_cdc -scheme -find` command:

```
check_cdc -scheme -add and_based_edge -module and_based_edge -map {{Dout dout} {Data din} {Enable sready_in}}
```



Protocol Checks For Custom Synchronizers

Since a custom synchronizer is an ad-hoc scheme that does not map to any of the pre-defined types, the tool does not generate functional checks for a custom synchronizer. However, you can add custom protocol checks as follows:

```
check_cdc -protocol_check -add expression_name -expression expression_template -scheme scheme_type
```

In the command above:

- `expression_template` must be a valid SVA or PSL property. To build this template, you must replace the signal names with the corresponding formal signal. All formal signals used in the template must be related to the synchronizer in question and should be preceded by a `%` character. The tool then replaces all instances of `%formal_signal` by the actual formal signal when creating the protocol checks for the synchronizer instance.
- `expression_name` refers to the name of the property you are verifying. The name specified should be new and should not coincide with the existing names. Otherwise, the tool issues an error and does not verify the property defined.
- `scheme_type` is the custom scheme name for which you are creating the protocol check.

For the `and-based-edge` example detailed above, the command might be as follows:

```
check_cdc -protocol_check -add abe_stability -expression {@(posedge %dclk} (%Enable)
##4 $stable(%Data) until_with !(%Enable)} -scheme and_based_edge
```

⚠ If the custom scheme defined does not adhere to the protocol check defined, the tool reports a `USR_FN_PROP` violation.

4

Running CDC Analysis

This chapter discusses running the CDC analysis and includes the following sections:

Structural Analysis

This section describes procedures for running structural analysis and includes the following main topics:

Identifying and Manipulating Clock Domains

This section includes the following sections:

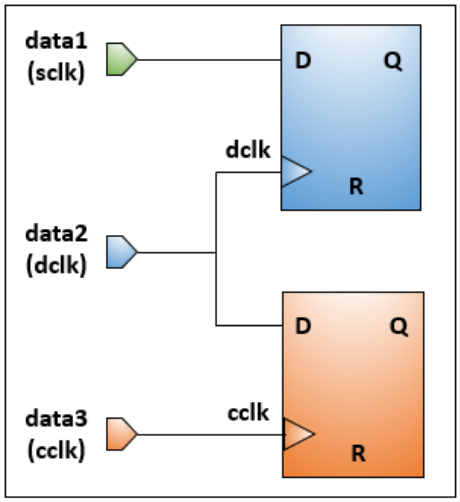
Handling Ports Used as Both Clock and Data

There are scenarios where the same port is used both as a clock and as a data signal. The tool handles this scenario in the following manner:

- The clock rating of the port is propagated to the flops in its fanout.
- If the port is used as data in the downstream logic, it is treated as data during the analysis.
- If the port is used as a clock in the downstream logic, it is treated as a clock during the analysis. The tool will report a `CLK_NO_DECL` violation on all such potential clocks.
- Rating of unrated ports is determined by the parameter `treat_data_ports_as_async`.

Example 1: Rated Ports

In the example below, there are three data ports (`data1, data2, data3`) and three clock ports with all the three clocks (`sclk, dclk, cclk`) declared and asynchronous to each other. Each of the data ports is rated to one of the declared clocks.



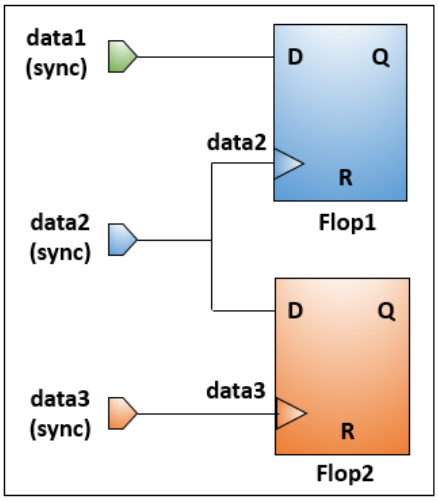
As data port `data2` (rated to clock `dclk)` is used both as a clock and as a data signal in the downstream logic and the data port `data3` (rated to clock `cclk)` is used as a clock in the downstream logic, the tool will behave in the following manner:

- Report `CLK_NO_DECL` violation on `data2` and `data3` as both the ports are being used as clocks.
- Report CDC Pairs Between `data1-data2(sclk-dclk)` and `data2-data3(dclk-cclk)` as all of them are rated in different clocks that are asynchronous to each other.
- The clock pins of the flop will use the clock ratings of data port `data2` and `data3` for carrying

out the analysis.

Example 2: Unrated Ports

In the example below, there are three data ports `(data1, data2, data3)` and no rating has been provided to any of these ports. All these ports will be treated as synchronous to the entire design. Also, the parameter `treat_data_ports_as_async` is set to `none`.

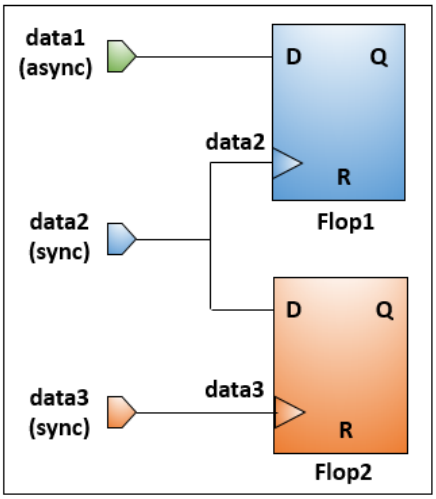


As data port `data2` is unrated and is used as both clock and data signal in the downstream logic and the data port `data3` is used as a clock in the downstream logic, the tool will behave in the following manner:

- Report `CLK_NO_DECL` violations on `data2` and `data3`, as both of them are being used as clock.
- `Flop1` will be driven by `data2` clock domain and `Flop2` will be driven by `data3` clock domain.
- No CDC Pair will be reported as both the clock domains `data2` and `data3` are synchronous to each other. This happens as all the undeclared clocks are considered to be synchronous to each other by default.

Example 3: Unrated Ports with `treat_data_ports_as_async` set to `all`

In the design below, there are three data ports `(data1, data2, data3)`, and the parameter `treat_data_ports_as_async` has been set to `all`. This will consider all the unrated ports as asynchronous to the entire design.



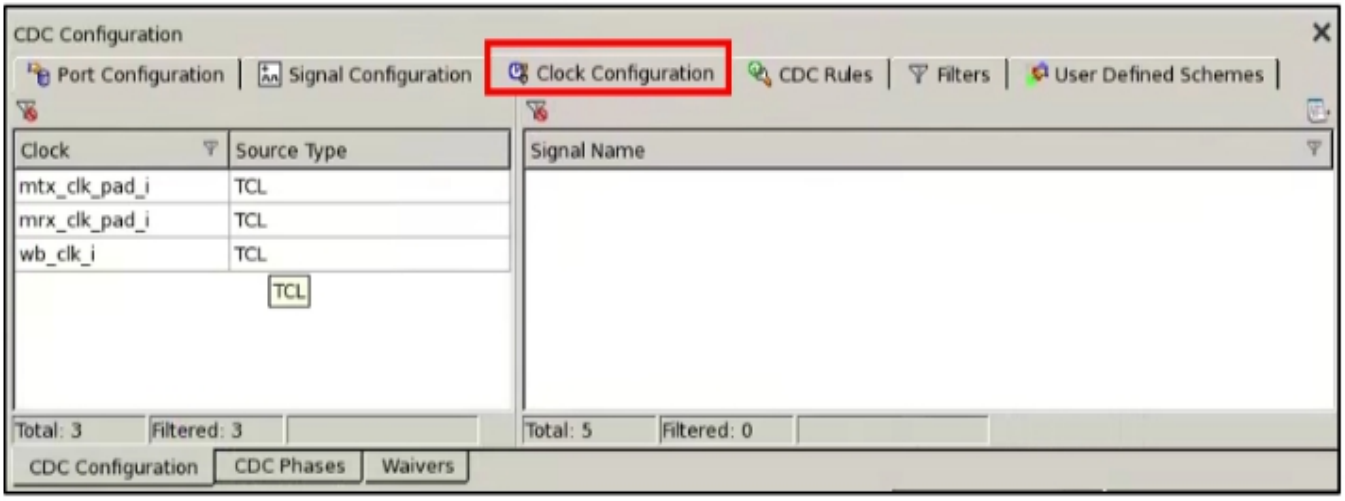
As the data port `data2` is unrated and is used as both clock and data signal in the downstream logic, and the data port `data3` is used as a clock in the downstream logic, the tool will behave in the following manner:

- Report `CLK_NO_DECL` violations on `data2` and `data3`, as both of them are being used as clocks.
- Data port `data1` will be asynchronous, whereas, `data2` and `data3` will be synchronous since the tool considers all the undeclared clocks to be synchronous.
- No CDC pair will be reported as both the clock domains `data2` and data3 are synchronous to each other.

Populating the Clock Configuration Tab

To populate the *Clock Configuration* tab, do one of the following:

- Click the *Find Clock Domains* button on the *CDC Configuration* toolbar.
- Type the command `check_cdc -clock_domain -find` on the command line.



The *Clock Configuration* tab displays all clock domains in the left-hand pane, along with a source column indicating the declaration source of that particular clock.

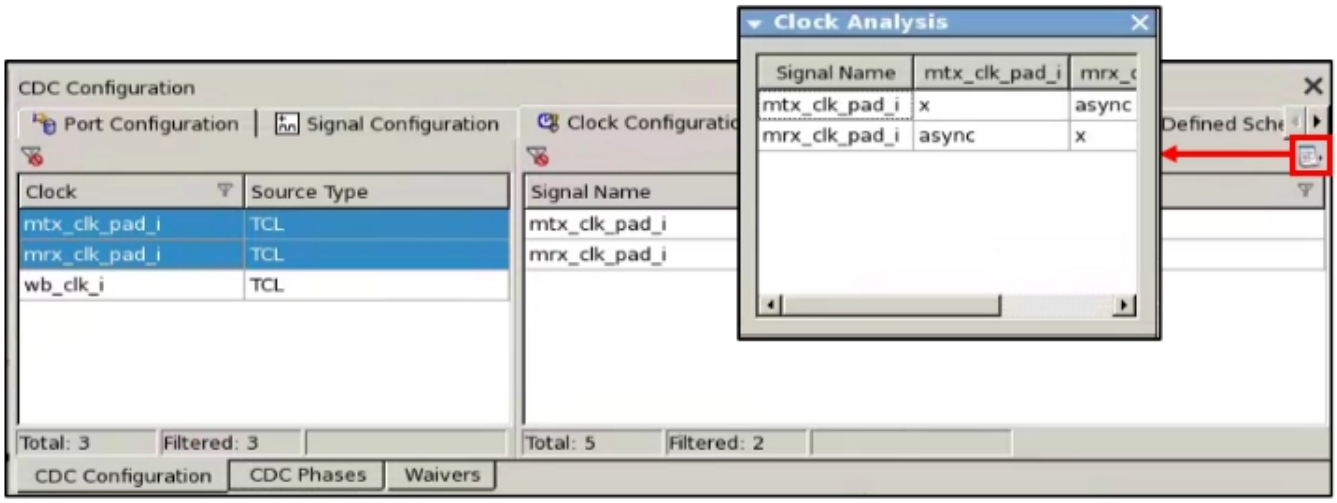
Viewing Clock Relationships from the GUI

You can review the clock relationships from the GUI using the *Clock Analysis* button.

To view the clock relationships, follow the steps below:

1. Select the clocks from the *CDC Configuration* tab *Clock Configuration* sub-tab.
2. Click on the *Clock Analysis* button located at the right of the *Clock Configuration* tab.

   A clock matrix window opens, displaying the clock relationships between the selected clocks.



Finding the Clock Domain of a Signal

You can use the following command after running `check_cdc -clock_domain -find` to return the clock domain of a specified net:

```
check_cdc -debug -show_clock_domain <signal_name>
```

This command returns a Tcl dictionary that lists the inferred clock domains for the target signal. If different bits of the signal have different clock domains, each bit is a different key in the Tcl dictionary. If all bits of the signal have the same clock domains, the full signal is the key in the Tcl dictionary. See the following example:

```
% check_cdc -debug -show_clock_domain clocks_and_resets.clk1_div_reg
clocks_and_resets.clk1_div_reg clock_control1
```

This feature to annotate the clock domain in the schematic for all signals in all debugging views or return it on the command line is controlled by the following command. It is set to `true` by default.

```
% check_cdc -debug -clock_domain_all_nets (true | false)
```

Finding the Reset Domain of a Signal

You can use the following command after running `check_cdc -reset -find` to return the reset domain of a specified net:

```
check_cdc -debug -show_reset_domain <signal_name>
```

This command returns a Tcl dictionary that lists the inferred reset domains for the target signal. If different bits of the signal have different clock domains, each bit is a different key in the Tcl dictionary. If all bits of the signal have the same reset domains, the full signal is the key in the Tcl dictionary. See the following example:

```
% check_cdc -debug -show_reset_domain rxethmac1.ByteCntEq6
rxethmac1.ByteCntEq6 mrx_rst_pad_i
```

This feature to annotate reset domain in the schematic viewer for all signals in all debugging views or returning it on the command line is controlled by the following command. It is set to `true` by default.

```
% check_cdc -debug -reset_domain_all_nets (true | false)
```

Structural Analysis – CDC

Structural checks identify issues related to synchronizers, convergence, and combinational logic to confirm that all CDC signals have been properly synchronized. This section provides information on CDC App pairs, schemes, and convergence analysis and includes the following sections:

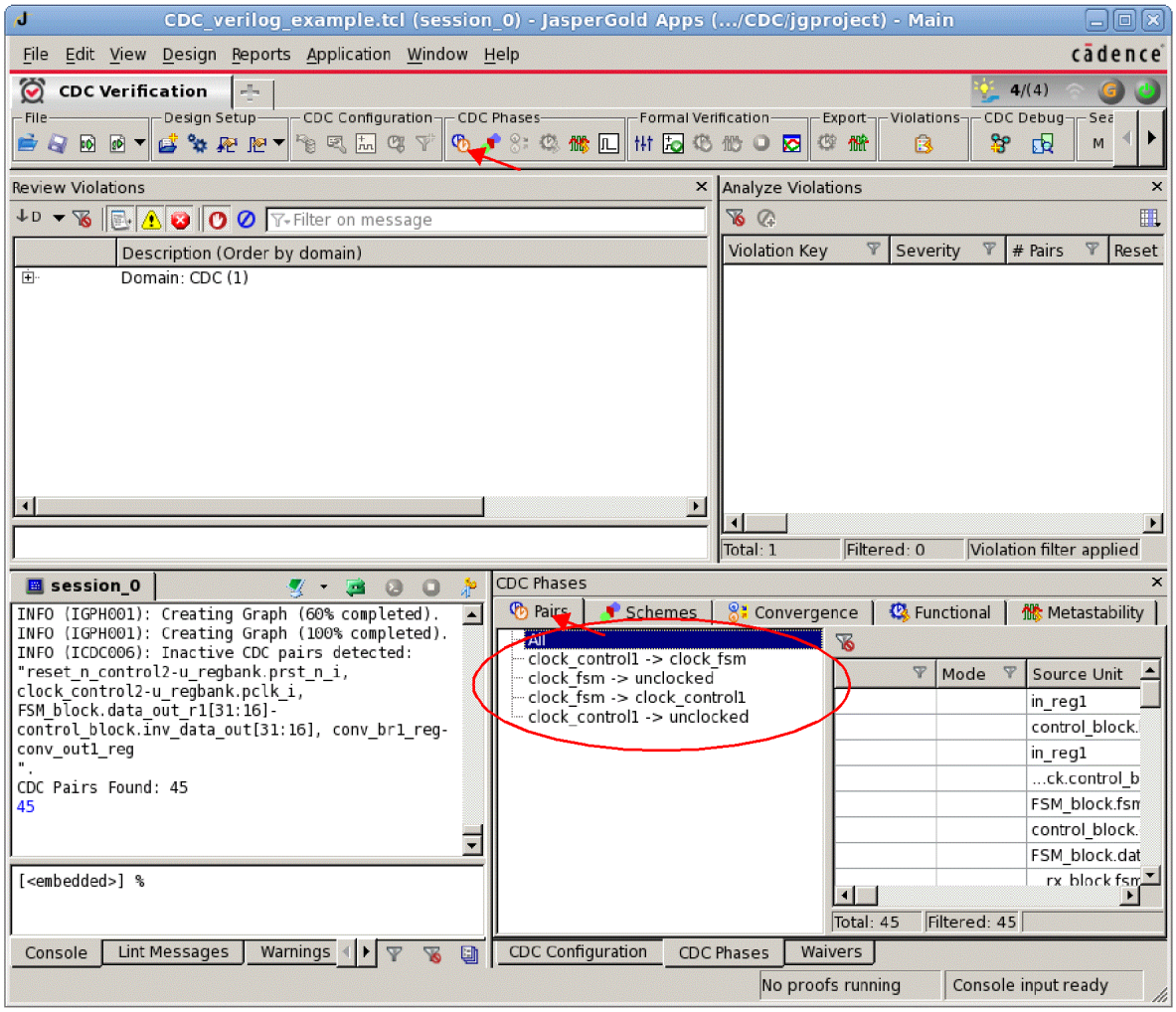
- Finding CDC Pairs
- Finding CDC Synchronizers
- Finding Convergence

Finding CDC Pairs

To populate the *Pairs* tab, do the following:

- Click the *Find CDC Pairs* button on the *CDC Phases* toolbar.

  In the *Pairs* tab, all clock domain crossings are listed in a tree on the left and the CDC pairs for each clock crossing are displayed on the right.

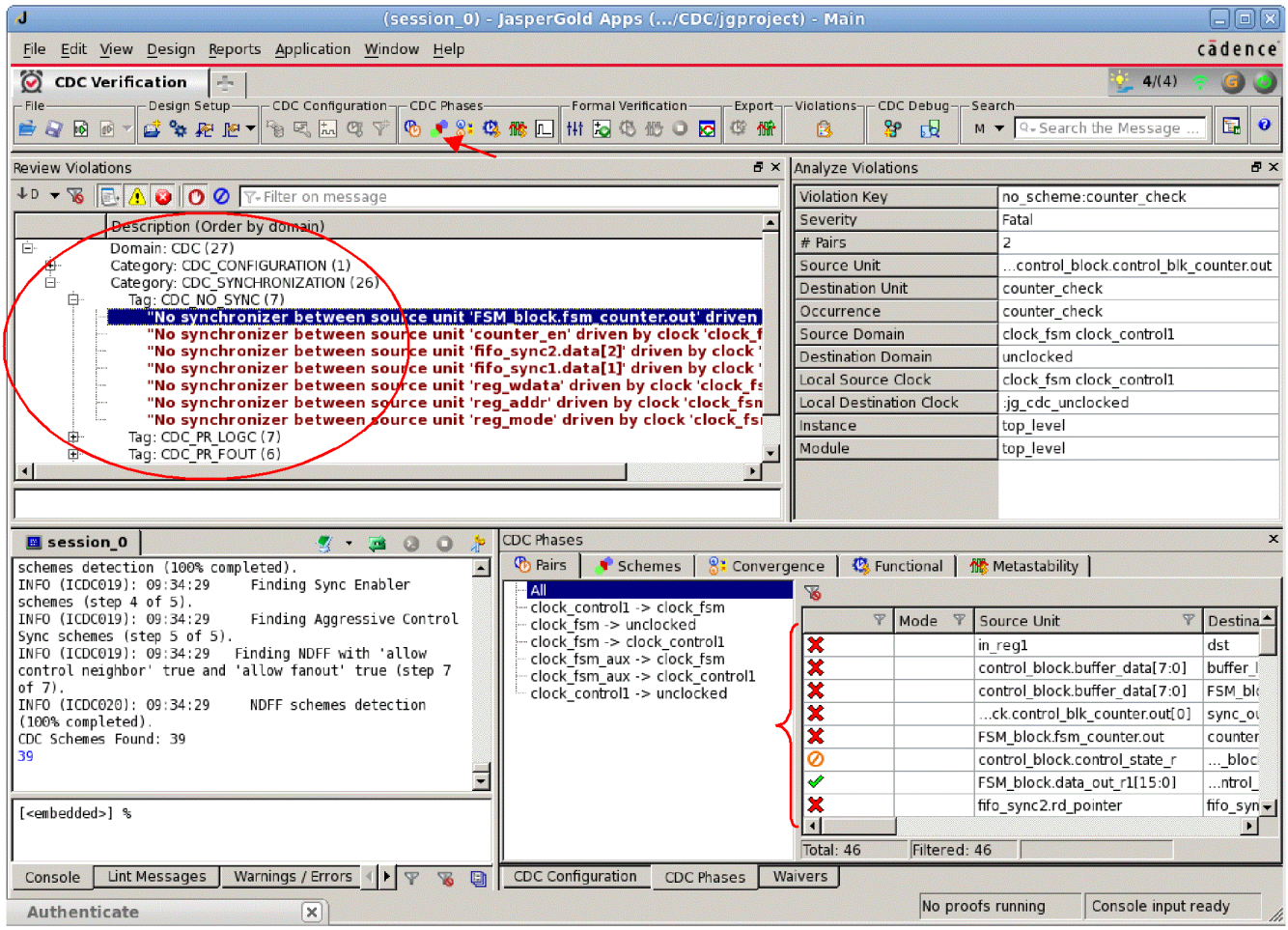


Finding CDC Synchronizers

To automatically identify all synchronizers on these paths, do the followings:

1. Click the *Find Schemes* button on the *CDC Phases* toolbar.
   The CDC violations shown in the *Review Violations* table go from one to 23, and the icons in the left-most column of the right-hand pane of the *Pairs* tab now indicate the status of the CDC pair. Red Xs denote paths that have violated path rules, blue "not" circles denote violations that you have waived, orange "not" circles denote violations that have been automatically waived, and green check marks denote paths that have passed all path rules.

   ⚠ See "Waiving Violations" for additional information about user-defined and automatic waivers.

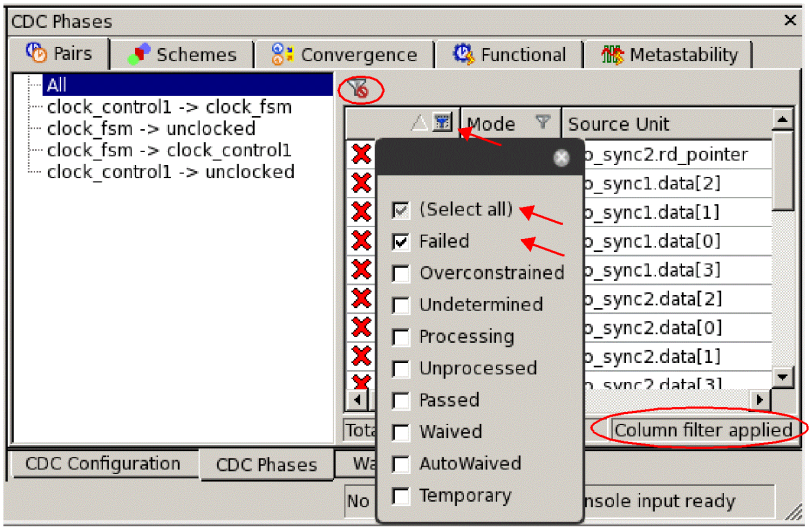


2. To view only failing pairs from the *Pairs* table, do the following:

   a. Click on the *Change filter options* button in the heading of the *Status* column to open the filter dialog box (see the figure below).

   b. Click *Select all* to deselect all, and then select *Failed* to see only failed schemes.

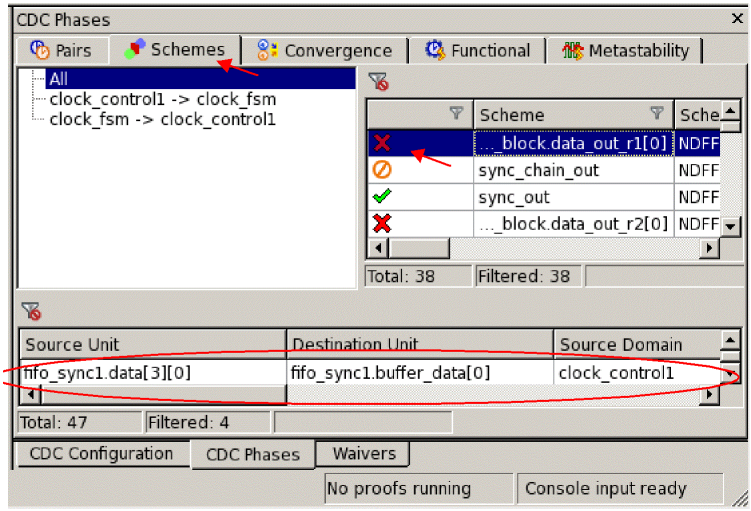
      The table contents are filtered and the status bar shows *Column filter applied*.



c. To restore the complete list, click the *Reset all filters* button.

3. To view the details of the schemes detected, click the *Schemes* tab.

   The left-hand pane displays a tree of all CDC pairs, and the right-hand pane shows all synchronizers the tool identified. When you highlight a scheme, a third pane at the bottom of the *Schemes* sub-tab shows the pairs table.

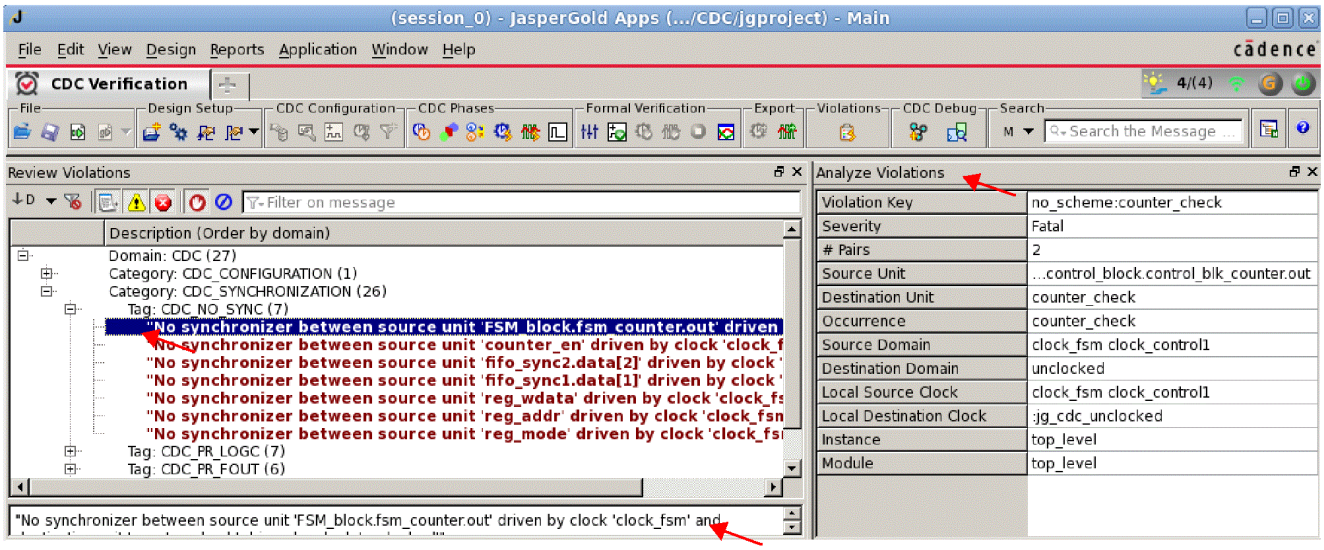


Understanding CDC Violations

To learn more about CDC violations, do the following:

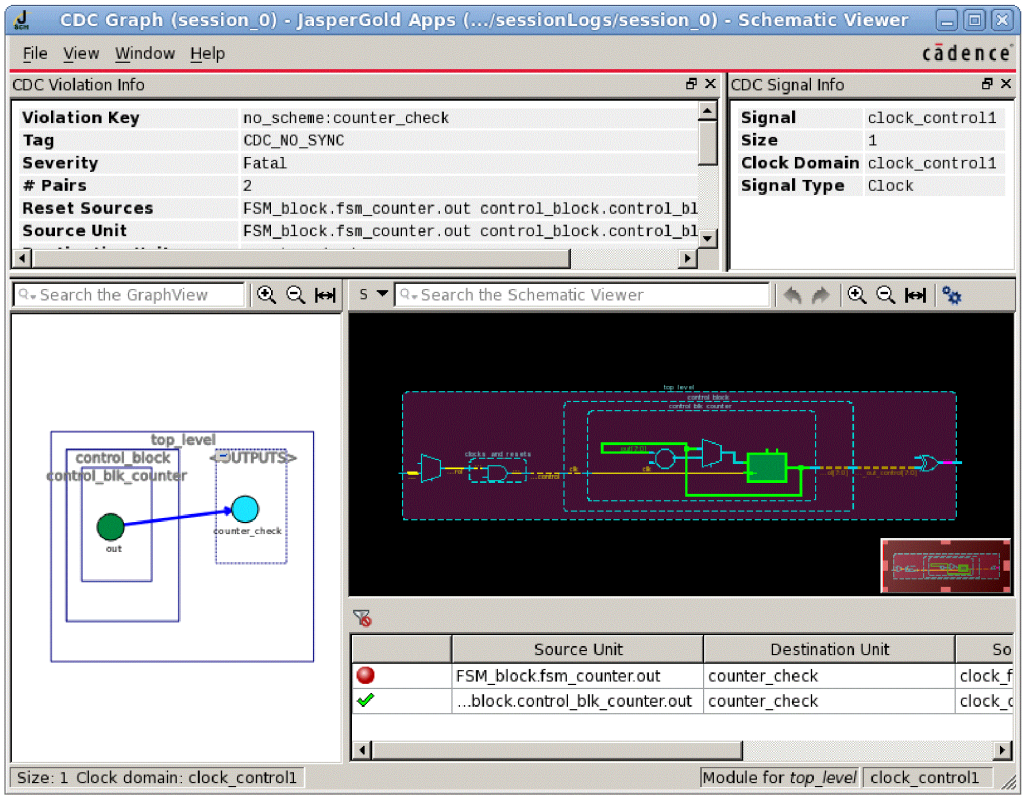
1. Click on a violation in the *Review Violations* table.

   The short message appears at the bottom of the *Review Violations* table, and the details of the violation are shown in the *Analyze Violations* table.



2. Right-click on the violation and choose *Open Tag Help* to see the full tag help if available. This provides access to the violation description in the *CDC Checks Reference*.
3. Right-click on the violation of interest and choose *Show Schematic*.

   An integrated debugging view with schematic and graph opens. See "Structural Violations" for additional information.



Fixing a Violation

You can fix a violation by modifying the RTL or by altering the path rule settings for the problematic CDC pair. You might also choose to waive violations (see "Waiving Violations").

Finding Convergence

The *Convergence* tab displays any glitch or convergence issues that might exist in the design. Structural glitches report potential glitch sources in a clock domain (for example, combinational logic in the CDC path). Convergence issues identify cases where CDC signals from *different* sources converge into combinational logic in the destination clock domain or where CDC signals from the *same* source (for example, a bus vector) converge into combinational logic in the destination domain.

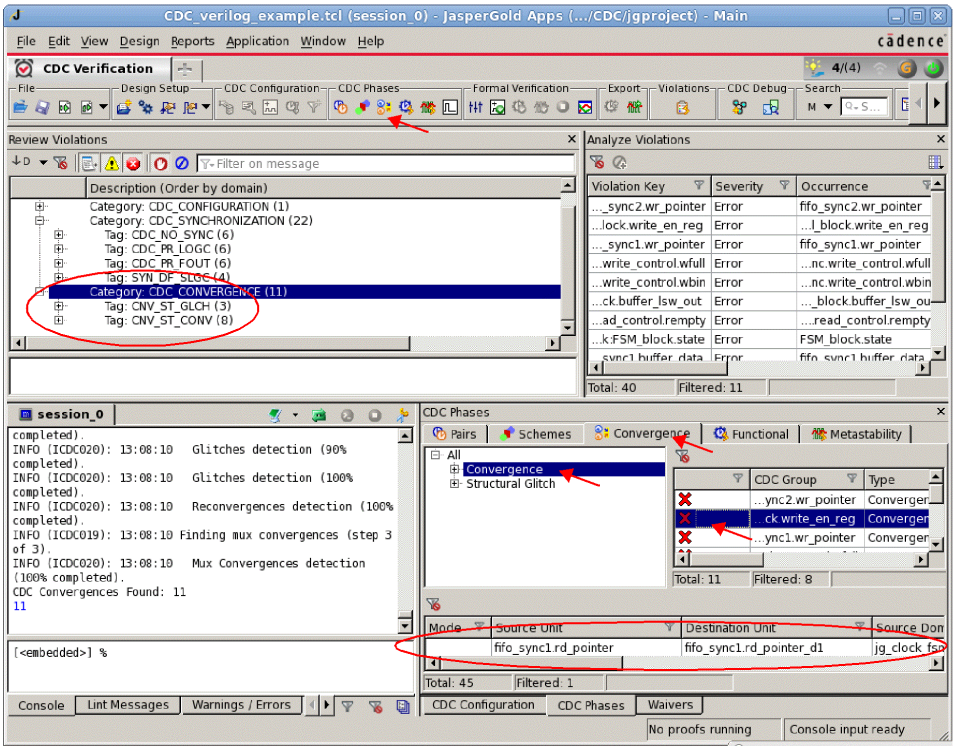
This section provides information on CDC App convergence analysis and includes the following:

- Identifying Convergence Issues
- Defining Structural Cells
- Defining Enable-Based Convergence Schemes

Identifying Convergence Issues

To populate the *Convergence* tab, do one of the following:

- Click the *Find Convergence* button on the *CDC Phases* toolbar.
- Type the command `check_cdc -group -find` in the console.
  The tool shows convergence issues and structural glitches as sets on the tree in the left-hand pane of the *Convergence* tab. The convergence points are shown in the *CDC Group* column on the right-hand pane and all CDC pairs involved in the CDC group are shown in the pairs table at the bottom of the *Convergence* tab.



Defining Structural Cells

The presence of combinational logic on the CDC or reset path can cause the tool to report violations, indicating the possibility of glitches that can propagate through the design. To avoid glitches, structural cells and modules are used to construct the combinational logic. This can reduce spurious violations resulting in noise reduction.

To specify a glitch-free structural cell or module in the design, use the following command:

```
% check_cdc -structural_cell -add -module {mycell1 mycell2}
```

Also, any previously added module can be removed with the command below:

```
% check_cdc -structural_cell -remove -module mycell2
```

View the defined structural cells using the following command:

```
% check_cdc -list structural_cells
```

Defining Enable-Based Convergence Schemes

During convergence analysis, the tool tries to remove convergence violations by using defined `enabled_based` schemes. A scheme can remove a convergence violation if all the data paths associated with the violation are controlled by the enable signal provided in the declaration of the scheme and the enable signal is synchronous to the convergence point. If there is at least one data path that is not controlled by the scheme, or the enable scheme is asynchronous to the convergence point, the convergence point is not covered by that scheme.

Hence, if only one converging path is selected at a time, you can use the following command to specify the enable-based convergence scheme:

```
% check_cdc -group -scheme -add enable_based -map {{enable <signal>}}
```

⚠ All convergence schemes should be defined prior to running the `check_cdc -group -find` command.

Viewing Convergence Schemes

You can view the defined convergence schemes using one of the following commands:

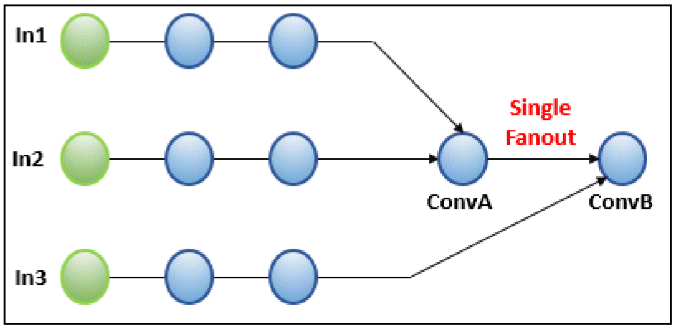
- `check_cdc -list convergence_schemes`
- `check_cdc -report convergence_schemes -file <filename>`

Grouping Convergence Issues

CDC can group convergence violations. It detects all the pairs involved in the convergence but avoids reporting duplicate convergence points that differ only in depth. See the following scenarios.

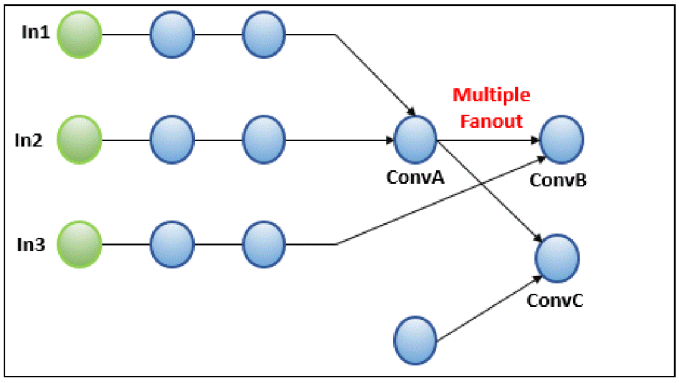
Scenario 1: Fully Contained Outputs

In the scenario below, convergent points `ConvA` and `ConvB` have different source signals, which causes the tool to report violations on both nodes.



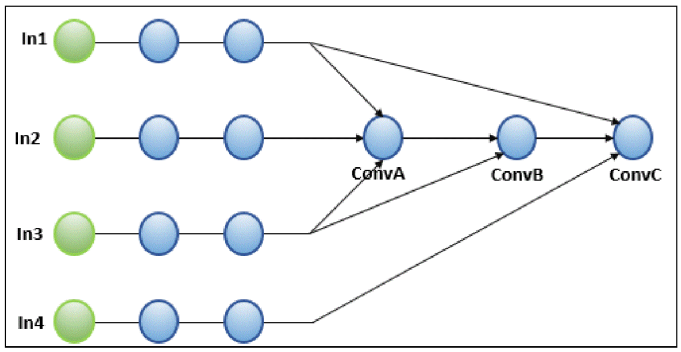
Scenario 2: Partially Contained Outputs

In the scenario below, convergence violations are reported only on convergent points `ConvA` and `ConvB`. Convergence at `ConvC` contains the same source signals as `ConvA` along with another fully synchronous node, which causes the tool to discard the violation under duplication criteria.



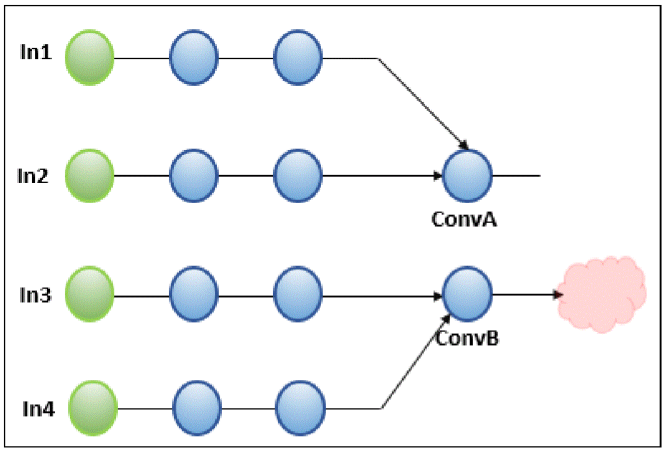
Scenario 3: Multi-Depth Convergence

In the scenario below, convergence violations are reported only on convergent points `ConvA` and `ConvC`. The tool discards convergence at `ConvB` under duplication criteria as it contains the same source signals as `ConvA`.



No Convergences with Disconnected Outputs

CDC does not report convergence violations on disconnected nets because they do not allow metastable values to be propagated in the design, and thus, can be safely discarded. For example, in the scenario below, a convergence violation is reported at `ConvB`, but no convergence violation is reported at `ConvA`.



Grouping FSM Convergence Violations

You can use the CDC parameter `group_fsm_convergences` to reduce the noise in convergences related to FSMs. When you enable this parameter, CDC convergences occurring on FSM signals are aggregated to a single violation on all of the bits. The aggregated violation is reported if at least one of the FSM transitions has more than one controller or there is a a state where the size of the set of the controllers of its outgoing transitions is greater than one. Otherwise, the aggregated violation is automatically waived with the comment `FSM state register driven by one asynchronous source`.

While debugging an aggregated FSM convergence, you can see the list of FSM transitions affected

by each one of the convergences roots through the *FSM Transitions* column in the pairs table at the bottom of the schematic window. Furthermore, a Boolean *FSM* column in the group report identifies whether the group is related to an FSM. When the `group_fsm_convergences` parameter is disabled, the value of this column is always *False*.

Enable or disable this parameter as follows:

```
config_rtlds -rule -parameter {group_fsm_convergences = true | false} -domain CDC
```

The default is `false`.

Resolving Convergence Issues

To open a schematic of a violation, do the following:

- Right-click on a convergence violation in the *Review Violations* table and choose *Show Schematic*.

  ⚠ You can also right-click on a violation in the *Convergence* table and choose *View in Schematic*.

  A schematic plus graph view opens (see "Structural Violations" for additional information).

Fix convergence issues by using a different synchronizer or choosing to automatically waive violations.

Structural Analysis – RDC

Reset domain crossings (RDC) are a well-known source of metastability in digital designs. Assertion of an asynchronous reset in a flop might cause the data input of subsequent flops to change asynchronously with respect to their clocks.

This section describes CDC App options for specifying reset association for ports and includes the following topics:

- Finding the Reset Domain of a Signal
- Reset Order Analysis
- Manually Specifying Reset Association

Finding Reset Domain of a Signal

You can use the following command after running `check_cdc -reset -find` to return the clock domain of a specified net:

```
check_cdc -debug -show_reset_domain <signal_name>
```

This returns a Tcl dictionary that lists the inferred reset domains for the target signal. If different bits of the signal have different reset domains, each bit is a different key in the Tcl dictionary. If all bits of the signal have the same reset domains, the full signal is the key in the Tcl dictionary.

For example:

```
% check_cdc -debug -show_reset_domain clocks_and_resets.clk1_div_reg
clocks_and_resets.clk1_div_reg reset_n_control1
```

⚠ The behavior of the above command is controlled by the following command:

```
% check_cdc -debug -reset_domain_all_nets <true | false>
```

Reset Order Analysis

One possibility for addressing the structural problem presented by reset associations is to ensure that resets on both sides of an RDC are asserted in the correct order, that is, the latter flop needs to be reset by the time the reset of the first flop is asserted. During structural reset analysis, you can specify reset order information using the `check_cdc -reset -set_order` command. CDC then uses the information you have provided to automatically waive the appropriate RDC violations.

Reset Order Command

The basic command syntax for the reset order command follows:

```
check_cdc -reset -set_order <reset_signal_list>
```

The reset signal list is a list of lists that reflects how different resets are asserted with respect to each other. Resets appearing later in the list are asserted before resets appearing earlier in the list. Resets that are asserted simultaneously can be specified by grouping them together in a second-level list. See the following example:

```
% check_cdc -reset -set_order {ARST1 ARST2 {ARST3 ARST4}}
```

With this command, you inform the tool that `ARST3` and `ARST4` are asserted simultaneously and are already reset by the time `ARST2` is asserted, which is already reset by the time `ARST1` is asserted.

Note:

- You must specify the reset order during the CDC setup phase. You cannot run `check_cdc -reset -set_order` after running reset analysis with `check_cdc -reset -find`.
- `check_cdc -reset -find` uses the reset order information you specify to automatically waive non-problematic RDC violations. Use the `check_cdc -reset -set_order -preview` command to get the set of reset domain crossings the tool will automatically waive as a result of the reset order definitions provided.
- If no reset order has been specified, the tool will consider all the declared resets to be of the highest priority.
- Reset order information can be specified in successive `check_cdc -reset -set_order` commands. The tool keeps an internal representation of the cumulative effect of all the reset order definitions provided.
    - CDC issues warning WCDC005 if a new `check_cdc -reset -set_order` command contradicts previous reset order relationships between resets. The previous relationships are overridden, and the last definition provided prevails.
    - If a new reset order command creates a circular dependency among resets, the tool issues the error ECDC001, and the entire new reset order definition is discarded.
    - Use the `check_cdc -list reset_order` command to get the set of reset order definitions you provided.

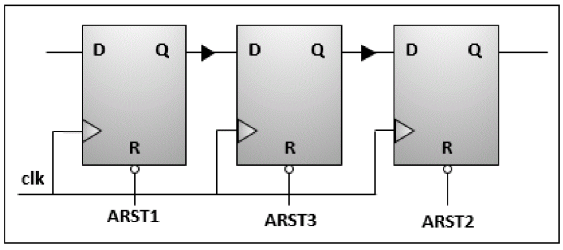
See the following examples of RDC violations based on the reset setup specified with the following command:

```
% check_cdc -reset -set_order {ARST1 ARST2 {ARST3 ARST4}}
```

Example 1

Crossing `ARST1->ARST3` is not problematic, considering that `ARST3` will already be reset by the time `ARST1` is asserted. The tool automatically waives this RDC violation.

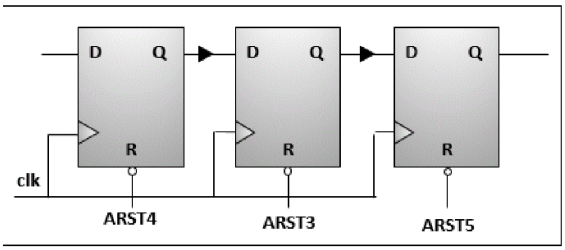
The tool reports crossing `ARST3->ARST2` as an RDC violation since there is no guarantee that `ARST2` is reset by the time `ARST3` is asserted (only the opposite has been specified).



Example 2

Crossing `ARST4->ARST`3 is not problematic, considering that `ARST4` and `ARST3` are asserted simultaneously. The tool automatically waives this reset violation.

Reset domain crossings between resets for which no order relationship is specified are not automatically waived. Thus, the tool would report a violation for a crossing between `ARST3->ARST5` since it cannot infer anything about the reset order of `ARST5`.



Example 3

When CDC cannot trace each flop's asynchronous reset pin back to a single reset signal, the tool reports an RDC violation, as shown in the two scenarios below:

Figure 4.1:
Crossing ARST1,ARST2→ARST1

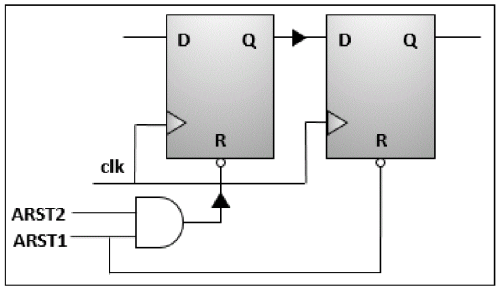
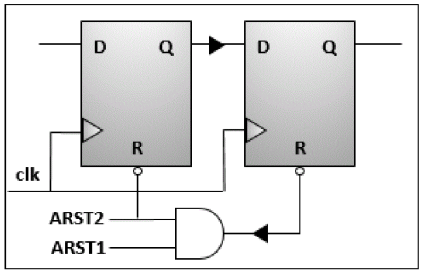


Figure 4.2: Crossing ARST2→ARST1,ARST2



Manually Specifying Reset Association

When you issue `check_cdc -reset -find` or click the *Find Reset Signals* button on the *CDC Phases* toolbar, the tool infers reset domain information for ports automatically unless. Based on the reset order, the tool decides whether an RDC is valid or not. For invalid reset domain crossings, the tool reports a `RDC_RS_DFRS` violation.

If you want to override the tool-inferred reset association, you can manually specify the reset association for any port with the following command:

```
check_cdc -reset -port ({signal_list} -reset_signal {signal_list}
```

⚠ The reset association influences RDC analysis.

ⓘ To get more information on structural analysis for CDC and RDC, click the following link to navigate to the Jasper landing page on Cadence Online Learning and Support: https://support.cadence.com/jasper. From there, click the CDC button, scroll down, and follow the "Checkpoint Methodology in Jasper CDC" link under "Application Notes".

Functional Analysis

Structural analysis guarantees that the design is structurally verified from the CDC point of view. Its success is dependent on the quality of clock and reset specifications, CDC constraints, and the validity of the design assumptions used in dispositioning the violations. However, it does not confirm whether the design is functionally correct or not, that is, whether all synchronizers are functioning correctly or all constraints are valid.

This section provides information on validating constraints, generating protocol checks, and running

those in a formal environment or in simulation.

Constraint Validation

Constraints play an important role while performing CDC analysis on the design. However, since incorrect constraints might invalidate the entire CDC analysis, it is critical to ensure that the correct constraints have been applied.

To prove the signal configurations, use the following command:

```
% config_rtlds -signal -prove
```

- If a signal configuration is applied on an undriven net, it is treated as an assumption by the tool.
- Otherwise, the tool generates an assertion under the task *RTLDS_signal_config_properties*.
- Some conditions trigger additional verification for pseudo nature of the constraints (mainly statics, constants, and mutually toggle exclusive signals).

⚠ The tool treats all constraints declared with `check_cdc -signal_config` as assumptions during structural analysis. No signal configuration checks are generated for these constraints even if a precondition is specified. To run constraint validation or signal configuration checks, declare constraints using the `config_rtlds -signal` command.

Protocol Check Command Syntax

The CDC App is capable of automatically generating functional checks to verify the transfer protocols for each identified synchronization scheme. These functional checks confirm that the data from the source clock domain is properly captured by the destination clock domain without loss or corruption. They also check whether the control signals associated with the complex synchronizers are functioning correctly.

Use the following command syntax to generate and verify functional checks for all previously detected CDC schemes:

```
check_cdc -protocol_check ( -generate
                          |-prove
                          |-export -file file_name
```

- Use `-generate` to automatically create the properties.
- Use `-prove` to validate the properties created.
- Use `-export` to export the properties to a file.

Note:

- Currently, the tool generates functional checks only for schemes that have no structural violations.
- Checks are based on property templates (see "Common Synchronization Schemes" for related template diagrams.)
- For details of the protocol checks generated for every synchronization scheme, refer to the *CDC Checks Reference*.
- This section includes the basic syntax only. Type `help check_cdc -gui` on the command line to see the full `check_cdc` command syntax.

Running Protocol Checks in Formal

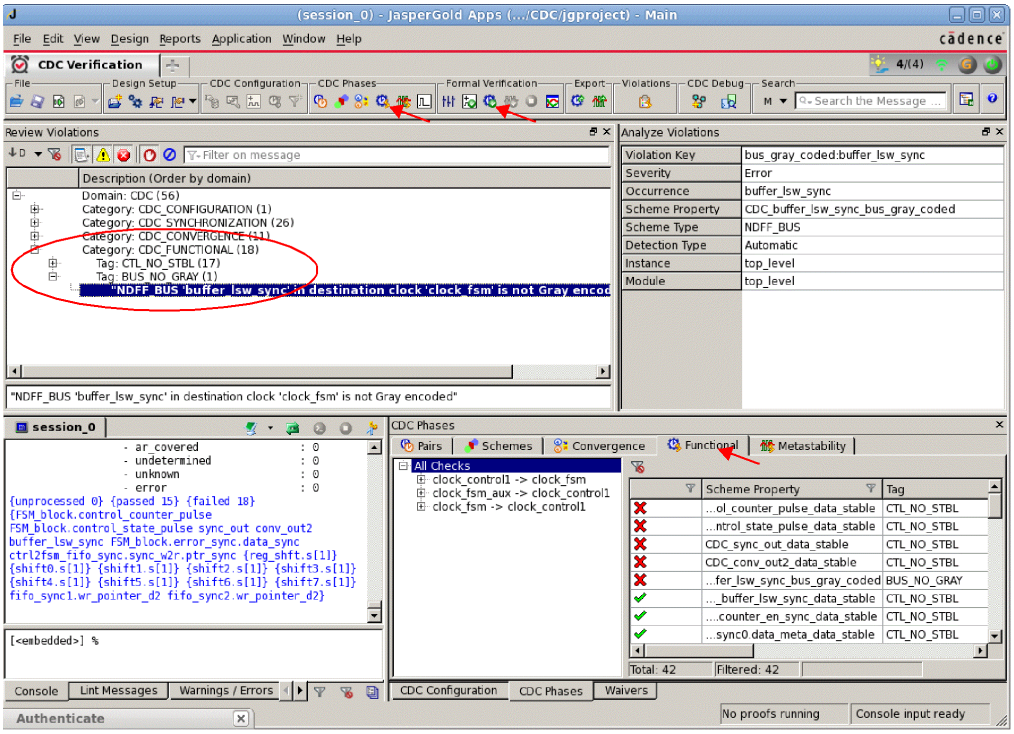
To populate the *Functional* tab, which displays all CDC checks and the associated details, do the following:

Click the *Generate Protocol Checks* button on the *CDC Phases* toolbar (see Figure 5-14). The associated Tcl command follows:

```
% check_cdc -protocol_check -generate
```

To verify the checks, click the *Prove Protocol Checks* button on the CDC App *Formal Verification* toolbar. Icons in the left-most column of the right-hand pane of the *Functional* tab indicate the status of the proof. The associated Tcl command follows:

```
% check_cdc -protocol_check -prove
```



To better understand why a check failed, do the following:

- Right-click on a failing property in the *Functional* table and choose *View Violation Trace*. A Visualize window and graph view opens (see "Functional Violations").
- Use Visualize debugging features to debug the trace.

For information on Visualize debugging features, access the Visualize manual available from the tool's Help menu (*Help – Application Guides – Jasper Visualize GUI Features*) or consult the "Visualize" chapter in the *Jasper Platform and Formal Property Verification App User Guide.*

> ⚠ If you have added assumptions to the `<embedded>` task that you want to apply to the CDC protocol checks or other CDC-related functional checks, consider using the following command to link all user-defined assumptions in the `<embedded>` task to `<CDC_*>` tasks: `task -link (<task_name>)+ [-to <inheriting_task_name>]`

You must run this command before running the CDC functional analysis.

Running Protocol Checks in Simulation

To export the generated assertions to simulation, do the following:

Click the *Export Filtered Checks to Simulation* button on the CDC App *Export* wizard. The *Export File* dialog box opens. Enter a file name.

Click *Save*. The tool generates an encrypted assertion file. The associated Tcl command for exporting protocol checks is as follows:

```
% check_cdc -protocol_check -export -file <file_name> [-force]
                                  [-property {protocol_check_name_list} [-regexp]

                                    |-default_clock <default_clock_name>

                                    |-filter filter_id

                                  [-default_disable_cond <expression>]
```

- Use `-file` to export the specified file. And use -force to overwrite the existing file
- Use `-property` to specify the list of schemes properties you want to export. -regexp can be used to consider the property name as a regular expression.
- Use `-default_clock` to specify the default assertion clock.
- Use `-filter` to export the properties that match the filter criteria
- Use `-default_disable_cond` to set which disable condition is prepended to exported properties

However, to export waiver conditions and signal configurations, use the following command:

```
% check_cdc -export -type (signal_config_property | waiver_cond_property) [-
user_assertions_only] -default_clock <clock_name> -file <file_name> -force
```

- Use `-type` to specify the report type you want generated to the specified file.
- Add `-user_assertions_only` to export only the user-defined conditional waivers if violations are also covered by automatic waiver.
- Use `-default_clock` to specify the default assertion clock.

⚠ The `-default_clock` option allows you to specify the clock to which you would like to export all functional checks (protocol checks, signal configurations, and waiver conditions) to simulation. If you do not provide the default clock, then the tool randomly picks the fastest clock as the sampling clock.

From your UNIX®1 terminal, include the generated assertion file in your simulation file list and run the simulation.

Metastability Analysis

Regular verification approaches like simulation or formal do not take into account the effect of metastability during functional verification. As a result, any change in the functional behavior of the design due to metastability is not caught during the verification process. To introduce the effects of metastability in functional verification, the tool needs to detect timing violations and model the metastable behavior of flops. Jasper CDC allows you to extend your regular formal- or simulation-based verification to inject random metastability into the design. If any assertion or test case that was passing starts to fail after metastability injection, it indicates that the design is not immune to metastability effects.

The Metastability tab includes three panes, a task table with a summary of proof results, a table of assertions and assumptions that represent the formal verification environment of the design, and a table that displays relevant CDC pairs. This section provides information on the CDC App metastability analysis and includes information on the following:

- Metastability Injection (MSI) Flow in Formal
- Metastability Injection (MSI) Flow in Simulation

Metastability Injection (MSI) Flow in Formal

To run the MSI flow in formal, you need user-defined assertions that are passing, which indicate that design functionalities are working properly. The objective of the MSI flow is to prove these assertions in the presence of metastability.

This section includes information on the following:

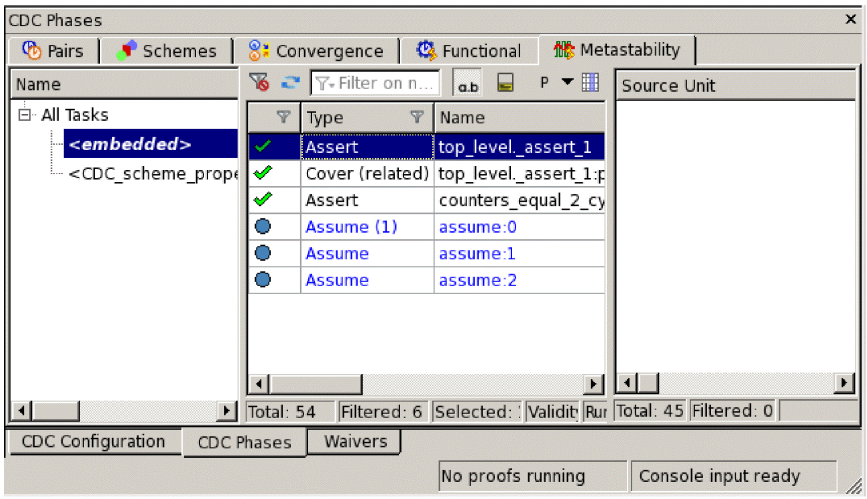
- Injecting and Verifying Metastability
- Selectively Enabling/Disabling Metastability Injection
- Understanding Metastability Failures

Injecting and Verifying Metastability

To determine whether properties are immune to the metastability effect, do the following:

Go to the *CDC Phases* tab *Metastability* sub-tab.

Right-click on one or more user-defined properties and click *Prove Property*.
This step confirms all the properties pass before injecting metastability. In the example shown below, all properties pass.
Figure 4.3: Passing Properties



To inject metastability effect into the CDC flops in the COI of the passing properties, click the *Inject Metastability in User properties* button on the *CDC Phases* wizard.
This resets the property status.

You can also use the following command to inject metastability:

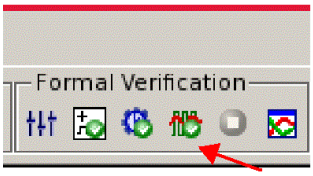
```
% check_cdc -metastability -inject [-include_reset] [-include_protocol_check] [-include_inactive_pairs]
```

- `-inject` automatically injects metastability in all the user properties
- Use `-include_reset` to inject metastability considering reset pairs
- Use `-include_protocol_check` to include protocol checks that were automatically generated by the tool as targets for metastability injection

- Use `-include_inactive_pairs` to also model the metastability effect in inactive CDC pairs

> ⚠ Beginning with 2022.03, if only the control paths are to be included for injecting metastability, the command above should be run after the `check_cdc -scheme -find` command. However, if you want to include metastability in both control and data paths, the command should be run before running `check_cdc -scheme -find`.

Click the *Prove User Properties with Metastability* button on the CDC App *Formal Verification* wizard.

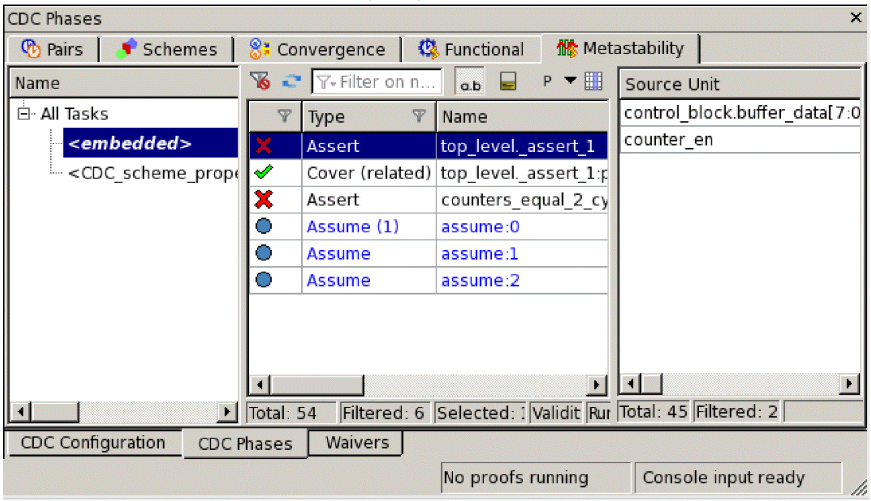


You can also use the following command to prove metastability:

```
% check_cdc -metastability -prove [-property <property_list>]
```

If an assertion fails in the presence of metastability, you can conclude that the failure is because of metastability effect.

Figure 4.4:
Failures after Metastability Injection



Selectively Enabling/Disabling Metastability Injection

User-defined properties might fail due to metastability injection in the formal environment. To better understand the root cause of the failure, you might want to selectively disable a few injection points. This can help you find a CEX with injection in a minimum set of crossings in the COI of the property.

To disable metastability injection at certain crossings, use the following commands:

```
% check_cdc -metastability -enable [-pair |-destination |-source | -property]
<list_of_keys>
```

```
% check_cdc -metastability -disable [-pair |-destination | -source |-property]
<list_of_keys>
```

```
% check_cdc -metastability -injection_status [-pair |-destination | -source |-property]
<list_of_keys>
```

- Use -`disable` to disable metastability injection in one or more clock-domain crossings.
- Use -`enable` to re-enable metastability injection in one or more clock-domain crossings.
- Use -`injection_status` to get the status of metastability injection
- Use -`pair` to select one or more crossings directly by providing their CDC pair keys as a list.
- Use `-source` or `-destination` to select all the crossings to one or more sources or destinations, providing their CDC Unit keys as a list or a wildcard expression. Use `check_cdc -list units` to find the available CDC Unit keys.
- Use -`property` to select all the crossings in the COI of one or more user properties by providing their User Property keys as a list.

Understanding Metastability Failures

All failures due to metastability injection are reported under the `Metastability` category in the *Violations* tree.

To debug a failure, do either of the following:

- Right-click on a failure in the *Metastability* sub-tab and choose *View Violation Trace*.
- Right-click on a violation in the *Violations* table and choose *Visualize Trace*.
  A Visualize window plus graph view opens (see "Metastability Violations" for additional information).

Metastability Injection (MSI) Flow in Simulation

ⓘ The MSI flow in simulation works with Cadence Xcelium simulator only.

The MSI flow in simulation requires that you have simulation test cases that are passing. This indicates that the design functionalities are working properly. The objective of the MSI flow is to run these test cases in the presence of metastability.

This section includes information on the following:

- Exporting MSI Files to Simulation from the GUI
- Exporting MSI Files Simulation from the Command Line
- Running a Metastability-Aware Simulation
- Customizing Setup and Hold Times
- Debugging Metastability Injection in Simulation
- Combining Metastability Injection Models in a Single Simulation
- Combinational Glitch Injection

Exporting MSI Files to Simulation from the GUI

To export the generated insertion points to simulation from the GUI, do the following:

1. Click the *Export Metastability Candidates to Simulation* button on the CDC App *Export* wizard.

   The *Export File* dialog box opens.

   ⚠ Though you must specify a directory to continue, the tool ignores this information. This dialog box will be removed in a future release.

2. Select a directory from the list and click *Choose*.

   The *Export to Simulation* dialog box opens.

3. Enter a value for the *Time Window* parameter.
4. Click *OK*.

In the session folder, the tool generates the files.

5. From your UNIX terminal, run the simulation using the generated metastability injection files.

Note:

- Metastability injection in simulation works on top of CDC pairs and not CDC synchronizers. Thus, all CDC pairs, including unsynchronized signals, quasi-static signals, and so forth, are considered for metastability injection in simulation. Therefore, you can export the MSI files immediately after finding CDC pairs if you choose.

Exporting MSI Files Simulation from the Command Line

Alternatively, you can export the generated injection points to simulation from the command line as follows:

```
% check_cdc -metastability -export -file <file_name> [-force]
                            ( -time_window ( <N> | <time_unit> | <N> <time_unit> ) [-dir
<dir_name>]
                                [-tolerance (N | time_unit | N time_unit)]
                                [-report] [-include_comboglitch] [-include_reset] [-
include_inactive_pairs]
                                [-max_operands <N>]
```

- Use `-force` to overwrite files in an existing directory.
- Use `-time_window` to specify the time window as follows:
  - *N* is mandatory if you do not specify the time unit.
    - If you use *N* and do not specify a *`time_unit`*, the default is nanoseconds (ns).
    - *N* is a natural non-zero number.
  - *`time_unit`* is mandatory if you do not specify *N*.
    - If you use *`time_unit`* and do not specify *N*, the default is `1`.
    - *`time_unit`* is `s`, `ms`, `us`, `ns`, `ps`, or `fs`.
  - If you override the setup and hold times when binding the metastability injection model to your design, you need to specify the value in seconds using scientific notation. For example, `bind top : dut meta_injection #(.TSETUP_clk(5e-9))`

`i_meta_injection();` overrides the value of the setup window associated with clock `clk` to be `5ns`.

- Use `-tolerance` to specify the time window after the active clock edge during which any timing violation will be ignored for injection
  - *N* is mandatory if you do not specify the time unit.
    - If you use "N" and do not specify a `time_unit`, the default is nanoseconds (ns).
    - N is a natural non-zero number.
  - `time_unit` is mandatory if you do not specify *N*
    - If you use `time_unit` and do not specify *N*, the default is 0.
    - `time_unit` is `s, ms, us, ns, ps`, or `fs`.
- Use `-dir` to export the simulation files to the specified directory. If you do not specify a directory, the simulation files are written to the session directory.
- Use `-report` to generate a report with the results of the `-export` command, including the list of exported and discarded candidates.
- Use `-include_comboglitch` to explicitly generate the extra RTL for combo-glitch mode.

  ⚠ `+define+CDC_COMBOGLITCH` has no effect unless you generate the model with this switch.

- Use `-include_reset` to explicitly generate the extra RTL for a simplified model of injection due to asynchronous reset de-assertion. See "Metastability Injection due to Asynchronous Reset De-Assertion" for additional information.
- Use `-include_inactive_pairs` to also include inactive CDC pairs in the model for metastability injection in simulation. When you do not use this switch, the tool exports active CDC pairs only.
- Use `-max_operands` to specify the maximum number of operands in the fan-in of a metastability injection point. Default value is 64.

Note:

- CDC groups injection points exported to simulation by destination and displays these by

destination in the report generated with the `-report` option. For example, even when two different CDC pairs `data1[1:0]->q[1:0]` and `data2[1:0]->q[3:2]` are shown in the Pairs tab, a single grouped injection point `q[3:0]` is exported to simulation and shown in the metastability injection report. Consider this fact when noting divergences between the number of CDC pairs reported and the number of candidates for metastability injection exported to simulation.

- When metastability is exported to simulation, there are various candidates (points) on which metastability is injected. One such candidate might be the one that monitors variable data types inside a VHDL entity. For scenarios where out-of-module referencing is required, CDC tries to find an equivalent signal (not a variable data type) for VHDL variable data types by tracing forward or backward. If no equivalent signal is found, CDC discards the signal and generates the following warning: `WCDC031 "unsupported hierarchical reference to VHDL variable data type".`
- CDC refrains from exporting to simulation any metastability injection point that has more than 64 operands in its fan-in by default. This prevents longer run times while processing the combinatorial clouds on the CDC paths that might end up in exporting thousands of signals, making the analysis in simulation unfeasible. You can change the limit with the help of the `-max_operands` switch during export.

Running a Metastability-Aware Simulation

When exporting metastability to simulation, the tool generates the following two files (in the session folder by default or in the folder specified with `-dir`):

- `global_meta_injection.sv`: This file contains shared utility code used by the metastability injection model.
- `<top>_meta_injection.sv`: <*top*> is the name of the top module analyzed in Jasper. This file contains the metastability injection model for the particular design analyzed in Jasper. The whole model is contained inside a module called `<top>_meta_injection` in the file.

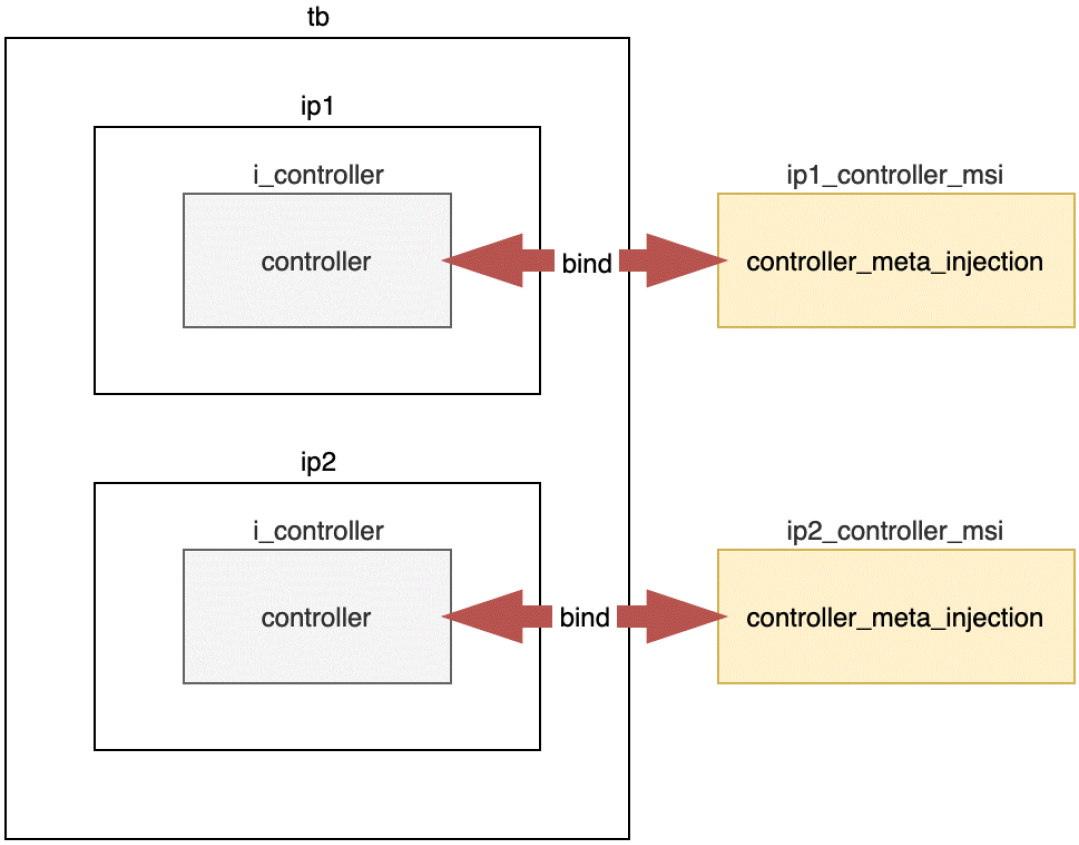
You must include both files when running a simulation, and you must also bind any instance of the `<top>` module to the corresponding metastability injection model to make them metastability-aware.

Since the file `global_meta_injection.sv` contains code used by the metastability injection model in `<top>_meta_injection.sv`, you must include them in this order when calling the simulator.

Example

Top module controller is analyzed in Jasper and files `global_meta_injection.sv` and `controller_meta_injection.sv` are generated after exporting metastability to simulation.

The simulation testbench has a top module called `tb`, and the module `controller` is instantiated twice inside the testbench hierarchy, at `tb.ip1.i_controller` and `tb.ip2.i_controller`, as shown in the following figure.



To bind the metastability injection model to the appropriate instances, you can include the following statements in a file (`bind.sv`, for example):

```
bind controller : tb.ip1.i_controller
controller_meta_injection ip1_controller_msi();
bind controller : tb.ip2.i_controller
controller_meta_injection ip2_controller_msi();
```

When calling the simulator, you can then combine the design files, the metastability injection files, and the bind file together as follows:

```
irun -sv <verilog_design_files> \
     -v93|v200x <vhdl_design_files> \
     global_meta_injection.sv \
     controller_meta_injection.sv \
     +extbind+bind.sv \
     -access rwc
```

ⓘ If your design has UDPs and you are getting compilation errors with Xcelium, try running `set_library_cell_optimization 2` before running `analyze` in CDC. The rest of the flow for exporting MSI files from Jasper remains same as detailed earlier.

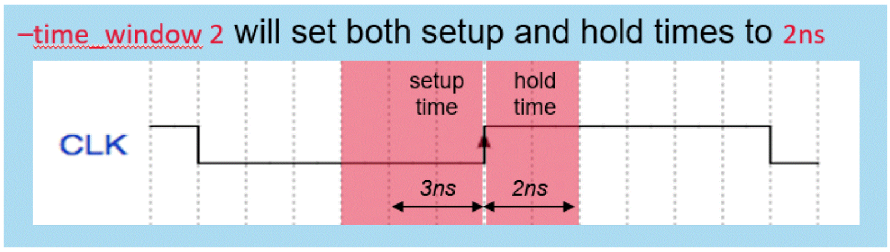
⚠ The switch `-hier_path` in the `check_cdc -metastability -export` command is deprecated. Its function, specifying to which instance the metastability injection model needs to be attached, is now performed by the bind statements described above.

Customizing Setup and Hold Times

The metastability injection model includes separate values for the setup and hold times of flops triggered by different clocks in the design. All are defined as parameters of the module `<top>_meta_injection`, included in the model file `<top>_meta_injection.sv` (where `<top>` is the name of the top module analyzed in Jasper).

By default, all the setup and hold time parameters are set to the value you provide with the `-time_window` switch of the `check_cdc -metastability -export` command. However, their values can be individually overwritten when the model is instantiated in the simulation testbench, specifying them in seconds using scientific notation as shown below:

```
bind controller : tb.ip1.i_controller controller_meta_injection #(.TSETUP_dclk(3e-9),.THOLD_dclk(1e-9)) ip1_controller_msi();
```



The bind statement above sets the setup and hold times as `3ns` and `1ns`, respectively, for all flops triggered by clock `dclk` that can suffer metastability in the model `ip1_controller_msi`.

Debugging Metastability Injection in Simulation

For every injection point (destination flop of a CDC pair) exported, the tool generates a metastability injection model. These models monitor the d-input of the flop for timing violations and randomly inject metastability if a timing violation occurs. Each model contains several injection witness signals that can be plotted in the simulator waveform viewer to understand the design behavior under metastability.

This section discusses debugging metastability injection in simulation and includes the following topics:

- Hierarchy of a Metastability Injection Model
- Injection Enables
- Injection Witness Signals
- Metastability Injection Messages
- Redirecting Injection Messages to a File
- META BEHAVIOR Tag
- META BEHAVIOR for Setup Violations
- META BEHAVIOR for Hold Violations

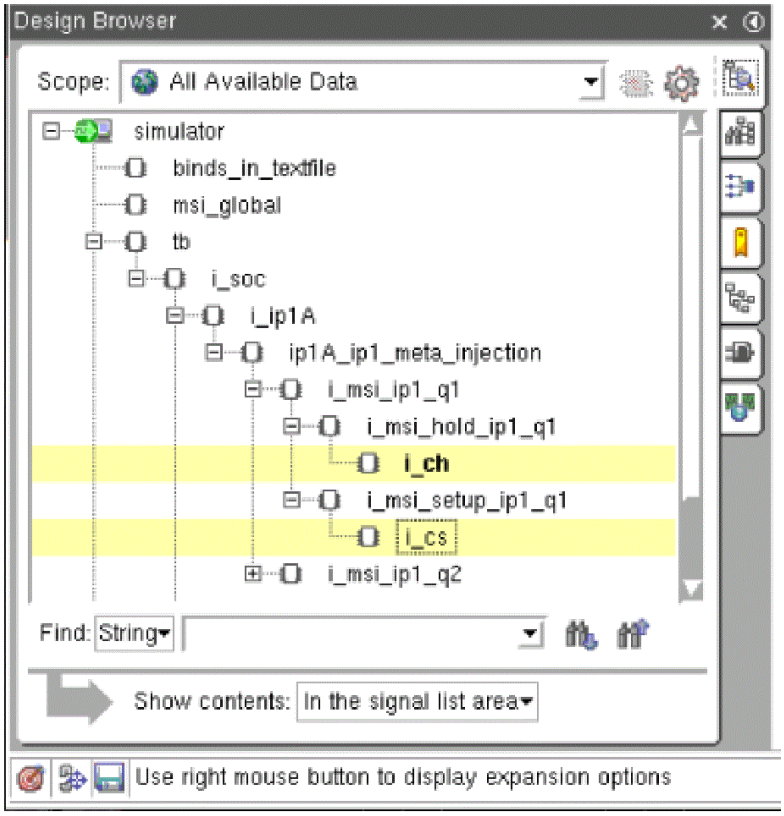
Hierarchy of a Metastability Injection Model

Metastability injection models generated by the tool are organized hierarchically, following some naming conventions. Being aware of the hierarchy and naming conventions will help you debug metastability related behavior.

In the content that follows, `<top>` is the name of the top module analyzed in Jasper, `<model_instance>` is the instance name given to the model when instantiated using the bind construct, and `<register>` is the path to each of the specific registers that can suffer metastability.

The hierarchy inside a metastability injection model includes the following four nested levels:

1. Wrapper module for the whole model:
    - Module name: `<top>_meta_injection`
    - Instance name: `<model_instance>`
2. Wrapper module for a specific register:
    - Module name: `msi_<top>_<register>`
    - Instance name: `i_msi_<top>_<register>`
3. Wrapper modules for setup/hold violations for a specific register:
    - Module names: `msi_setup_<top>_<register>` and `msi_hold_<top>_<register>`, respectively
    - Instance names: `i_msi_setup_<top>_<register>` and `i_msi_hold_<top>_<register>`, respectively
4. Setup/hold timing violation checkers:
    - Module names: `check_setup_violation` and `check_hold_violation`, respectively
    - Instance names: `i_cs` and `i_ch`, respectively (as shown below)



Injection Enables

Each wrapper module for specific registers inside the model have setup and hold injection enables exposed. Their full paths inside the model are as follows:

```
<model_instance>.i_msi_<top>_<register>._setup_inj_en
```

```
<model_instance>.i_msi_<top>_<register>._hold_inj_en
```

These signals have been exposed to allow a basic means of enabling/disabling metastability injection for specific registers during a simulation. By default, they are `true`, that is, metastability injection is enabled for every register in the model, but you can override this value by using force statements.

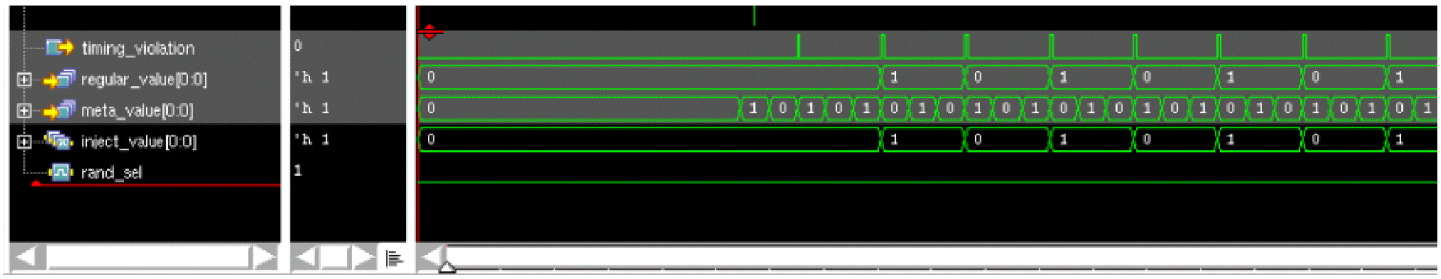
Injection Witness Signals

Several internal variables of the setup/hold timing violation checkers are available for ease of debugging the metastability injection behavior. All can be found at the following paths inside the model's hierarchy:

```
<model_instance>.i_msi_<top>_<candidate>.i_msi_setup_<top>_<candidate>.i_cs
<model_instance>.i_msi_<top>_<candidate>.i_msi_setup_<top>_<candidate>.i_ch
```

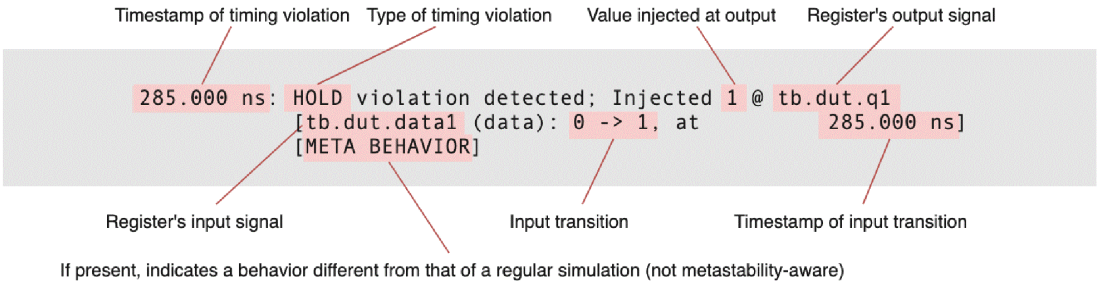
The purpose of each signal or variable is indicated below:

- `data_last_change`: Timestamp of the data's last change
- `clock_last_change`: Timestamp of the clock's last change
- `delta`: Last computed difference between data_last_change and clock_last_change
- `regular_value`: The value sampled by the register in a regular simulation (not metastability-aware)
- `meta_value`: This value can be considered as an indication that a specific data change can produce a metastable behavior, different from that of a regular simulation. This is not the value that gets injected when a timing violation happens (see `inject_value` below), but it is used to randomize `inject_value`. The key ideas follow:
  - A given data change inside the setup/hold window is considered as an injection opportunity if `regular_value` and `meta_value` are different from each other.
  - Only the bits that are different between `regular_value` and `meta_value` are randomized to generate the value actually injected (`inject_value`).
- `inject_value`: The actual value forced at the register's output as the result of a timing violation
- `rand_sel`: Random boolean value used to choose between the regular and meta values (done bit-wise) to decide the final `inject_value`
- `timing_violation`: A pulse that indicates that data has changed causing a timing violation; its width is TW (the time window, that is, either the setup or hold times)
- `meta_behavior`: Boolean value that indicates whether the value (randomly) injected is different from that of a regular simulation (not metastability-aware)



Metastability Injection Messages

Once the simulation runs the with the MSI files `global_meta_injection.sv` and `<top>_meta_injection.sv`, the tool reports metastability injections in the simulation log as follows:



In addition, at the beginning of the simulation, each timing violation checker module issues a message informing the value of the time window used to detect metastability injection opportunities. For example:

```
tb.i_soc.i_ip1A.ip1A_ip1_meta_injection.i_msi_ip1_q1.i_msi_setup_ip1_q1.i_cs: Time Window (TW) =
2.000 ns
```

These messages follow the hierarchy and naming conventions described above. In the previous example, the tool is informing that the time window for the setup violation checker (`i_msi_setup_ip1_q1.i_cs`) of register `q1` of metastability injection model for module `ip1` (`i_msi_ip1_q1`) is 2 ns.

Redirecting Injection Messages to a File

Alternatively, you can redirect the injection messages to a file instead of including them in the simulation log. To do so, set the following macro when calling the simulator:

```
+define+METALOGFILE=<file>
```

⚠ File `global_meta_injection.sv` contains a top-level module `msi_global` used for allowing the model to open and write the injection opportunity messages to the file provided. This module needs to be identified by the simulator as a top-level module; otherwise you see a `*E,CUVUNF: Hierarchical name component lookup failed at 'msi_global'` elaboration error when trying to run a metastability aware simulation.

As a general rule, just take care when your simulation contains VHDL top-level design units, since Verilog and SystemVerilog top-level modules are determined automatically (module `msi_global` among them):

- If your simulation testbench contains a single VHDL top-level design units, use the `-vhdltop <design_unit>` option when calling the simulator.
- If your simulation testbench contains multiple VHDL top-level design units, use one `-top <design_unit>` option for each of the top-level units.

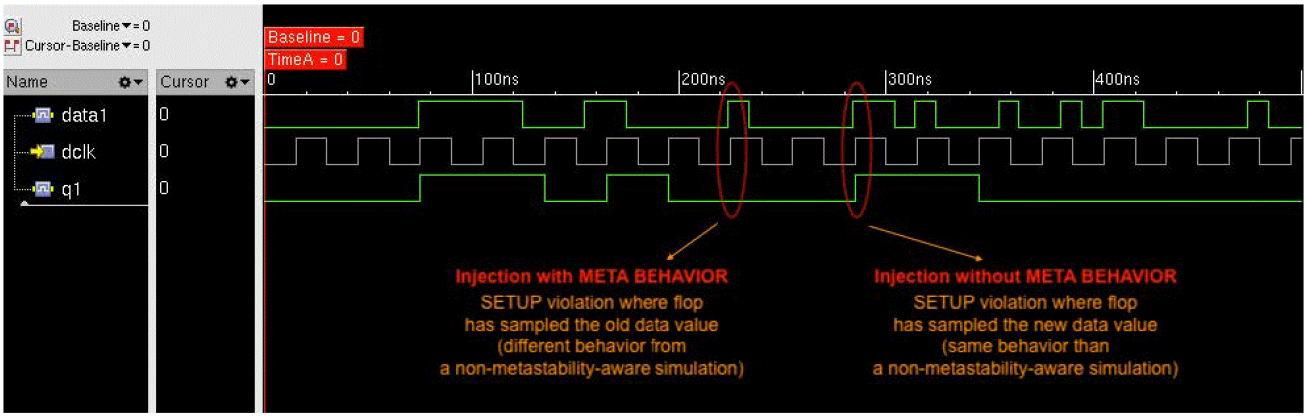
Refer to the simulator documentation for additional details on how top-level modules are determined in a simulation.

META BEHAVIOR Tag

As mentioned above, the tool adds a META BEHAVIOR tag in the message corresponding to metastability injections that caused an injection to happen due to a timing violation. This way, you can differentiate between an injection opportunity that occurred without any injection versus an injection opportunity with an actual injection, as shown in the examples below.

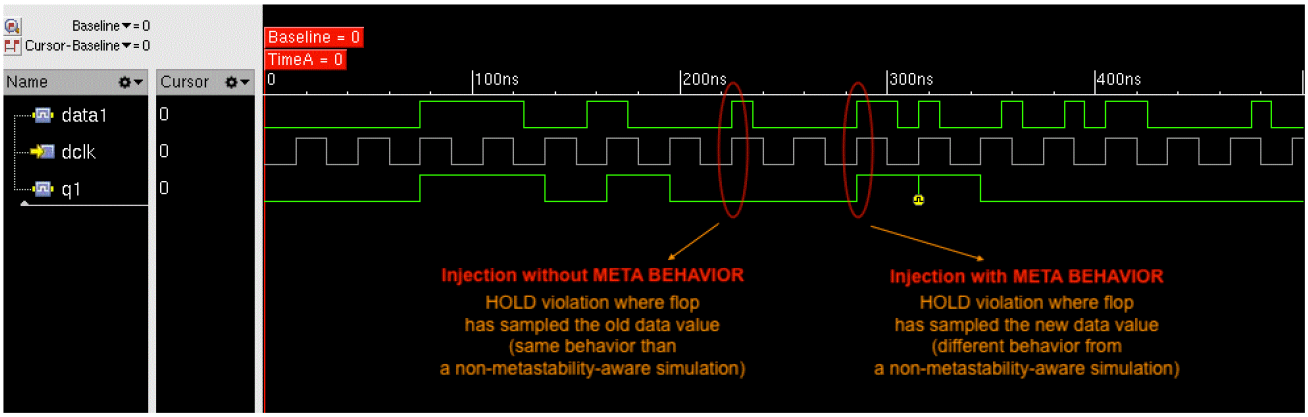
META BEHAVIOR for Setup Violations

```
225.000 ns: SETUP violation detected; Injected 0 @
tb.dut.q1
           [tb.dut.data1 (data): 0 -> 1, at 224.000 ns]
           [META BEHAVIOR]
285.000 ns: SETUP violation detected; Injected 1 @
tb.dut.q1
           [tb.dut.data1 (data): 0 -> 1, at 284.000 ns]
```



META BEHAVIOR for Hold Violations

```
225.000 ns: HOLD violation detected; Injected 0 @
tb.dut.q1
          [tb.dut.data1 (data): 0 -> 1, at 225.000 ns]
285.000 ns: HOLD violation detected; Injected 1 @
tb.dut.q1
          [tb.dut.data1 (data): 0 -> 1, at 285.000 ns]
          [META BEHAVIOR]
```



Metastability Injection Release Time

Metastability injections are achieved by forcing random values at the output of flops suffering timing violations. By default, the model releases such nodes one fentosecond after the random value is forced (so the flop can resume its normal operation). If flops in your design are modeled with some CLK-to-Q delay, this might create a race condition between the design and the metastability injection model when writing a value to the output of a flop. In such scenarios, the result of a metastability-aware simulation can be unexpected and difficult to debug (flops suffering metastability injection do not actually see their output forced to a random value).

To avoid this, you can override the `1 fs` default value for the release time by defining the `INJECTION_RELEASE_DELAY_FS` macro when running the simulation. The value specified is interpreted in fentoseconds. For example, including `+define+INJECTION_RELEASE_DELAY_FS=1000000` sets the release time to be `1 ns`.

Metastability Injection due to Asynchronous Reset De-Assertion

Asynchronous reset de-assertion is another well known source of metastability. When the asynchronous reset of a flop is de-asserted inside a recovery/removal time window around the clock's active edge, its output can go metastable if the input being sampled by the flop is different from the reset value. As a result, the flop can randomly sample the new input or remain with its reset value.

When exporting metastability to simulation, you can optionally include a simplified model for the injection due to asynchronous reset de-assertion if you add the `-include_reset` switch to `check_cdc -metastability -export`.

> This simplified model monitors the flop's asynchronous reset pin, without analyzing whether the changes seen are actually asynchronous or synchronous to the flop's clock. As a result, spurious injections due to asynchronous reset pins being synchronously de-asserted are known limitation of the model.

Most of what has previously been described about the data-path injection also applies to the metastability injection due to async reset de-assertion. Any relevant difference is explained below:

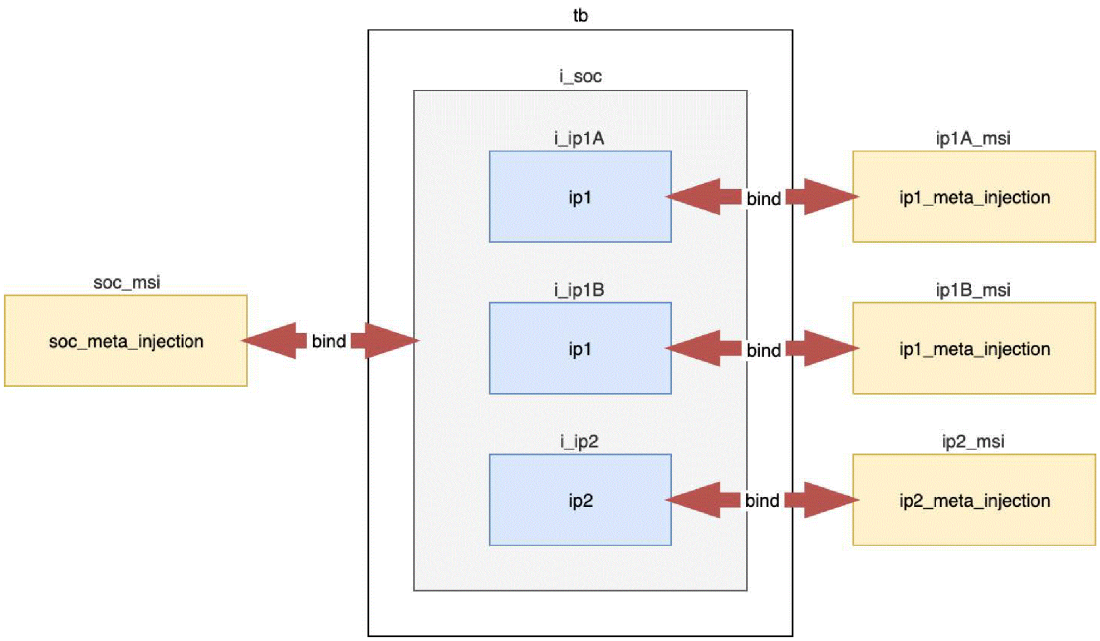
- Customizing recovery and removal times: By default, recovery and removal times are set to the value you provide with the `-time_window` switch of the `check_cdc -metastability -export` command. Their values can be individually overwritten when the model is instantiated in the simulation testbench, specifying them in seconds using scientific notation (for example, `bind controller : tb.ip1.i_controller controller_meta_injection # (.TRECOVERY_dclk(3e- 9),.TREMOVAL_dclk(1e-9)) ip1_controller_msi();`).
- Levels 2, 3, and 4 in the "Hierarchy of a Metastability Injection Model" are slightly different for the reset-path injection model:
  - Wrapper modules/instances for specific registers are called `msi_reset_<top>_register` and `i_msi_reset_<top>_<register>`, respectively.
  - Wrapper modules/instances for recovery/removal violations for a specific register are called `msi_recovery_<top>_<register> / msi_removal_<top>_<register>` and `i_msi_recovery_<top>_register> / i_msi_removal_<top>_<register>`, respectively.
  - Recovery/removal violation checkers are called `check_recovery_violation / check_removal_violation` and `i_crcv / i_crmv`, respectively.
- *Injection enables* for recovery/removal violations can be found in the following paths:
  - `<model_instance>.i_msi_reset_<top>_<register>._recovery_inj_en`
  - `<model_instance>.i_msi_reset_<top>_<register>._removal_inj_en`
- *Internal variables* of the model: `reset_last_change` contains the timestamp of the reset's last *de-assertion*.
- *Metastability injection messages* for recovery/removal violations include the type of violation, the value injected at the output of the flop, and whether or not the value injected is different from that of a regular simulation (non-metastability aware).

Combining Metastability Injection Models in a Single Simulation

The process described in "Running a Metastability-Aware Simulation" can also be used with multiple metastability injection models generated for different subsystems of a larger design. Basically, this is a two-phase process where 1) you need to analyze each subsystem in Jasper separately to generate its corresponding metastability injection model for simulation, and 2) you combine all the different metastability injection models together with your simulation testbench in a call to the simulator where the models are bound to the appropriate instances in the testbench.

Example

A SoC design (top module named `soc`) contains two different IP blocks (top modules named `ip1` and `ip2`, respectively), where `soc` actually contains two instances of `ip1` (`i_ip1A`) and `i_ip1B`) and one instance of `ip2` (`i_ip2`).



To set up a metastability-aware simulation where the SoC is instantiated as `i_soc` inside the simulation testbench module `tb` (as shown in the diagram above), perform the following steps:

1. Analyze `ip1` in Jasper, and generate its metastability injection model. This creates files `global_meta_injection.sv` and `ip1_meta_injection.sv`.
2. Analyze `ip2` in Jasper, and generate its metastability injection model. This creates files `global_meta_injection.sv` and `ip2_meta_injection.sv`.
3. Analyze `soc` in Jasper, black-boxing modules `ip1` and `ip2` (since their models have already

been created), and generate the metastability injection model for `soc`. This creates files `global_meta_injection.sv` and `soc_meta_injection.sv`.

4. Create a separate file to bind the different models to the appropriate instances in the simulation testbench, including bind statements like the following:

```
bind ip1 : tb.i_soc.i_ip1A ip1_meta_injection
ip1A_msi();
bind ip1 : tb.i_soc.i_ip1B ip1_meta_injection
ip1B_msi();
bind ip2 : tb.i_soc.i_ip2 ip2_meta_injection ip2_msi();
bind soc : tb.i_soc soc_meta_injection soc_msi();
```

5. Finally, call the simulator including all the necessary files, as follows:

```
xrun -sv <verilog_design_files> \
     -v93|v200x <vhdl_design_files> \
     global_meta_injection.sv \
     ip1_meta_injection.sv \
     ip2_meta_injection.sv \
     soc_meta_injection.sv \
     +extbind+bind.sv \
     -access rwc
```

ⓘ As shown above, you just need to include one instance of the file `global_meta_injection.sv`, since its content is fixed and independent of the corresponding metastability injection model. In addition, you must include it before all the model files, since it contains code used by the latter.

You must take care not to include the same clock-domain crossings in more than one metastability injection model. This is the reason, in step 3 above, the module `soc` is analyzed with instances of modules `ip1` and `ip2` black boxed. Otherwise, metastability injection for any crossing inside those instances would be duplicated (once in the models created for `ip1` and `ip2`).

⚠ As mentioned in "Running a Metastability-Aware Simulation", setup and hold times can be customized on a per-instance basis. To do so, simply overwrite the default value of the corresponding parameters in the appropriate bind statements.

Combinational Glitch Injection

Due to the potential for glitches generated by logic, including combinational logic in a synchronization path is generally against CDC design guidelines. However, due to some design constraints, designers might be sufficiently confident that the combinational logic is glitch-free to ignore reported violations. These potential violations, however, are normally masked during RTL simulation only to become apparent at a later stage, for example, gate-level simulation, as a source of metastability problems.

To reduce this risk, the CDC App includes a pessimistic metastability injection mode in simulation, which allows potential combinational glitches to create metastability injection opportunities at the flops sitting at the output of the combinational logic. That is, when you enable this pessimistic injection mode, the tool considers potential glitches due to changes in the CDC path signals violating the setup/hold times of the destination register. The relative order of the changes in the CDC path signals is randomized internally by the tool, thus creating the opportunity for glitches to be present at the output of the combinational logic.

Example

As shown in the figure below, there is some combinational logic, a simple AND gate between source flops data_a and `data_b` and the destination flop `q`.

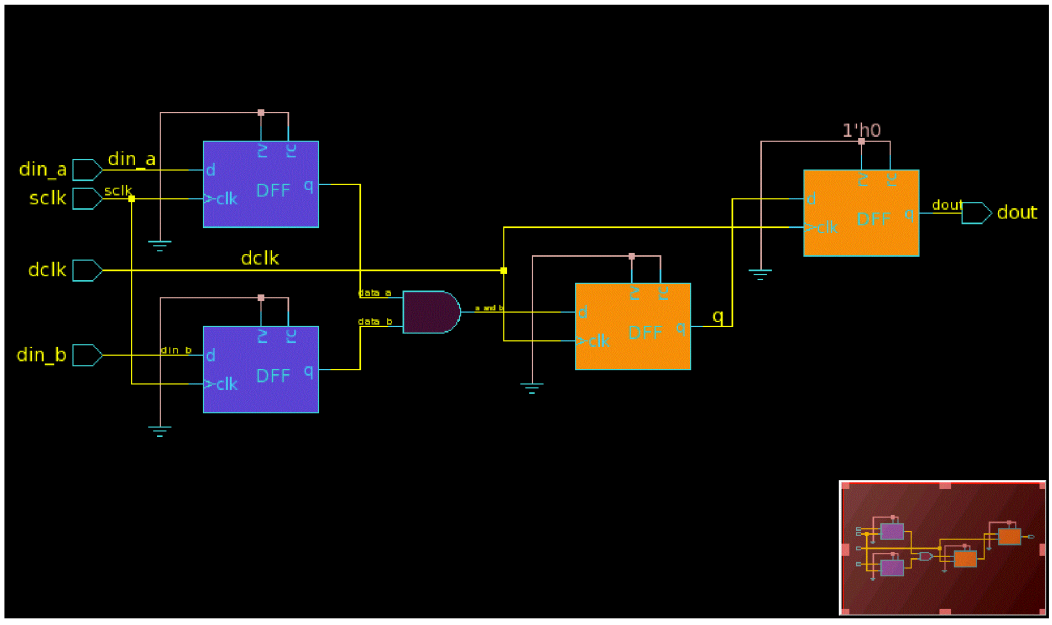
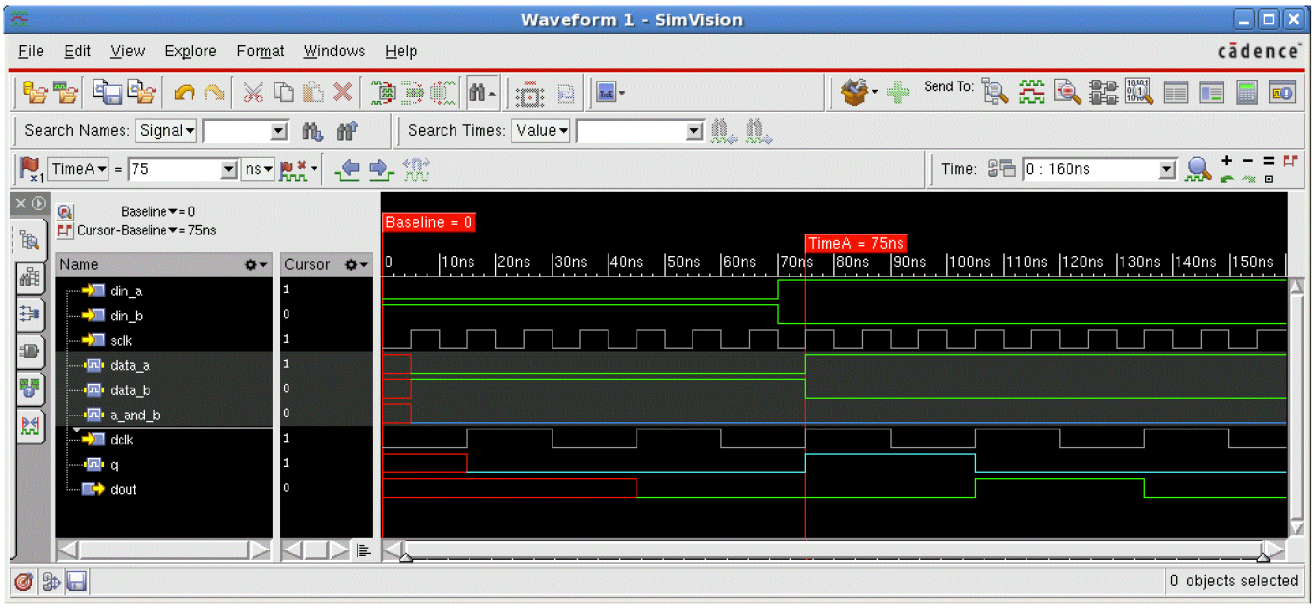


Figure 5-5 shows a potential glitch happening at TimeA = 75 ns. Inputs to the combinational logic `data_a` and `data_b` are simultaneously changing from `0` to `1` and from `1` to `0`, respectively. In an RTL simulation, however, these changes do not affect the output of the combinational logic (signal

a_and_b), which remains at 0. Therefore, no metastability injection opportunity should be generated at the destination flop q, even though data_a and data_b are changing precisely at the active edge of destination clock dclk.

However, with combinational glitch injection mode enabled, changes in data_a and data_b are internally randomized, simulating the chance of having a glitch at a_and_b and therefore creating a metastability injection opportunity at the destination flop, as shown in Figure 5-5. Note that q becomes 1 at TimeA = 75 ns even though its input a_and_b remains at 0.

Figure 4.5: Potential Glitch in the Combinational Logic



Enabling Combinational Glitch Injection

Enable the combinational glitch injection mode by setting the macro CDC_COMBOGLITCH when calling the simulator as follows:

```
irun \
  -sv <verilog_design_files> \
  -v93|v200x <vhdl_design_files> \
  <path_to_session_folder>/global_meta_injection.sv \
  <path_to_session_folder>/<top>_meta_injection.sv \
  +extbind+bind.sv \
  +define+CDC_COMBOGLITCH \
  -access rwc
```

⚠ : `+define+CDC_COMBOGLITCH` has no effect unless you add the `-include_comboglitch` switch to the `check_cdc -metastability -export` command.

1) UNIX is a registered trademark of The Open Group.

5

Analyzing the Results

This chapter provides information on debugging CDC violations, waiving violations, and generating reports. It include the following sections:

- Debugging CDC Violations
- Waiving Violations
- Generating Reports

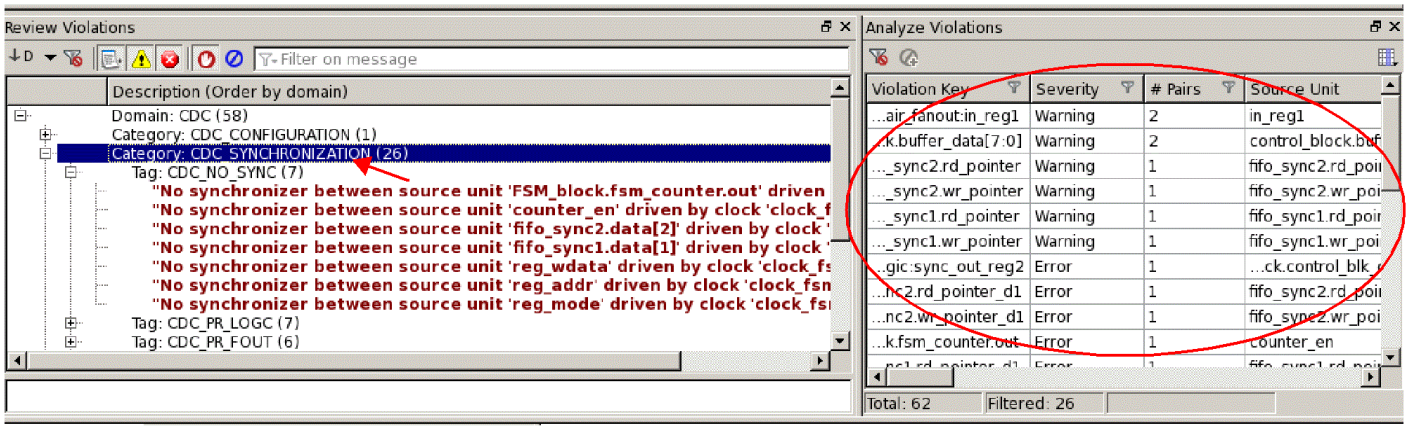
Debugging CDC Violations

This section provides information on debugging CDC violations and includes the following topics:

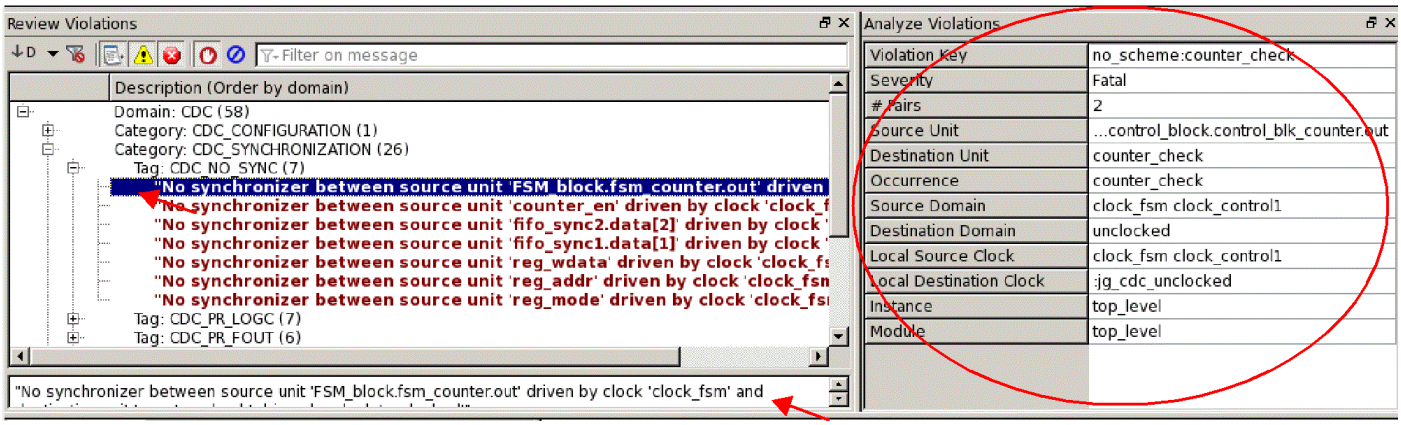
- CDC Violation Tree
- Filtering Violations
- Violation Grouping
- CDC Debugging Environment
- Schematic+Graph and Visualize

CDC Violation Tree

The CDC App finds and reports violations during every phase of clock domain crossing analysis. The *Review Violations* pane provides a list of all violations organized by domain and category. When you click on a category or tag, details on all violations within that category or tag are listed in the table to the right.



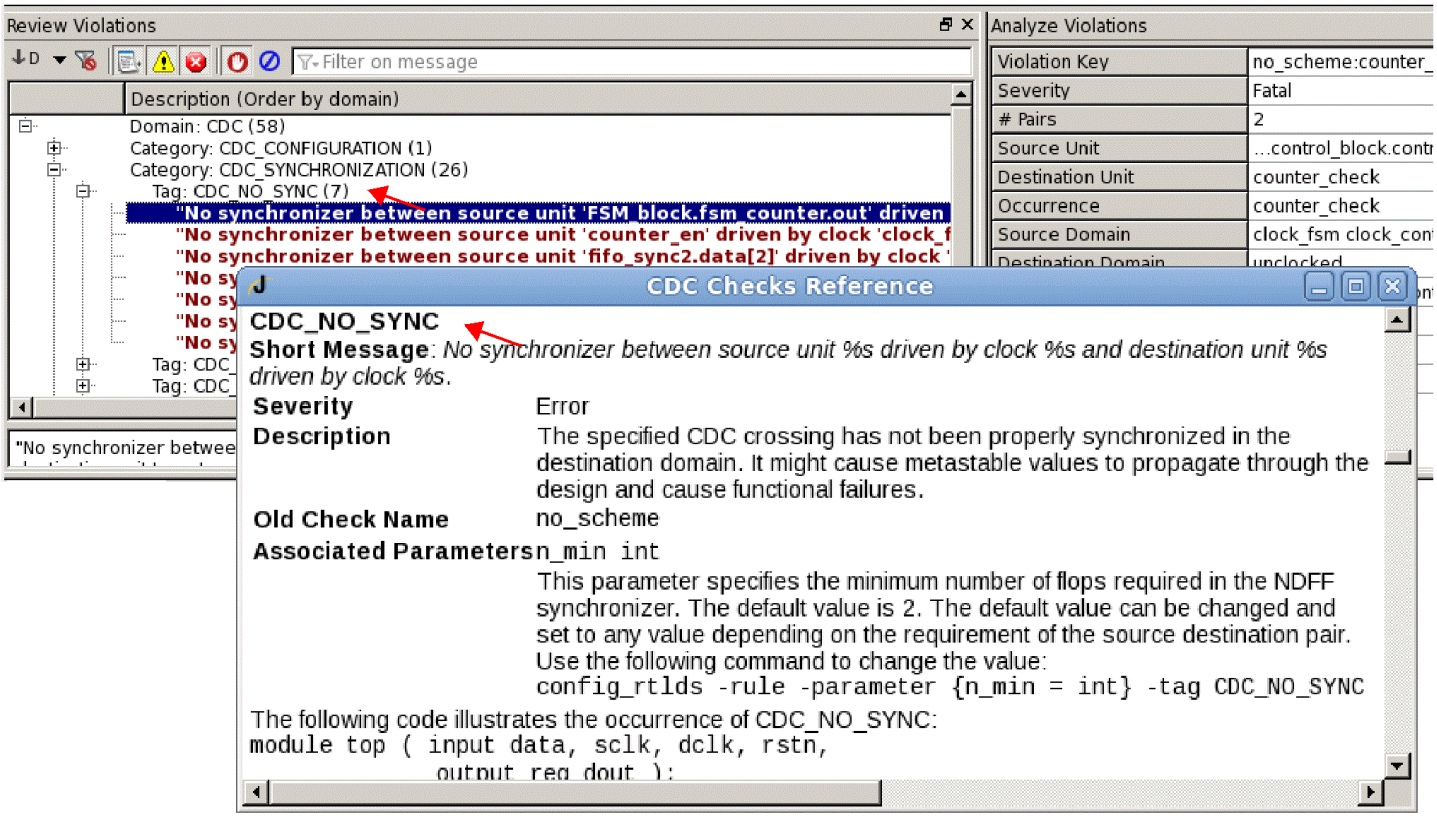
When you click on an individual violation, the details of that violation only are listed to the right.



In addition, you can view the short message for individual violations in the field at the bottom of the *Review Violations* table or right-click on a violation and choose *Open Tag Help* to see the full message details from the *CDC Checks Reference* (Figure 6-1).

> ⚠ The document that you access with this option is an HTML document that includes all checks for which message details are currently available. *Open Tag Help* simply takes you to the details for the specified check.

Figure 5.1: Viewing Message Details



Open the debugging environment by right-clicking on any of the individual violations and selecting *Show Schematic* or *Visualize Trace*.

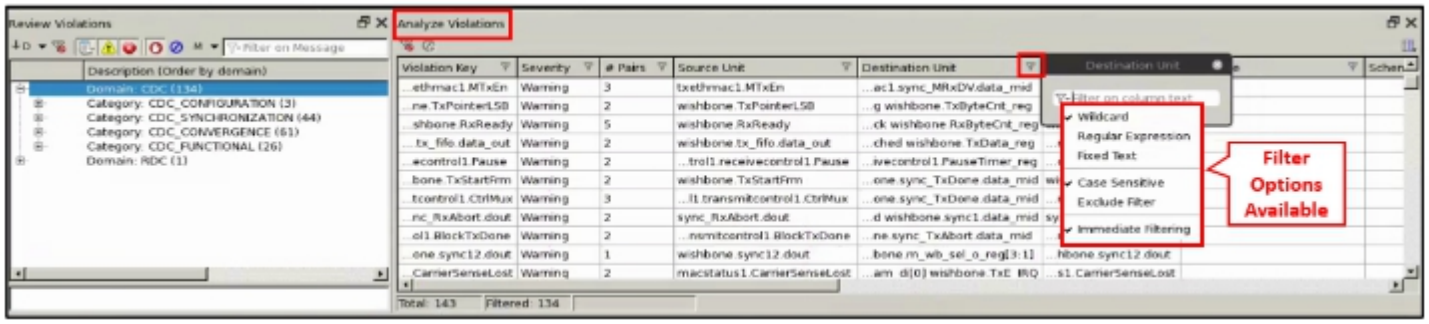
Filtering Violations

The CDC App provides the option to filter the violations reported in the violation tree by matching the column filters that have been applied.

The filters options are as follows:

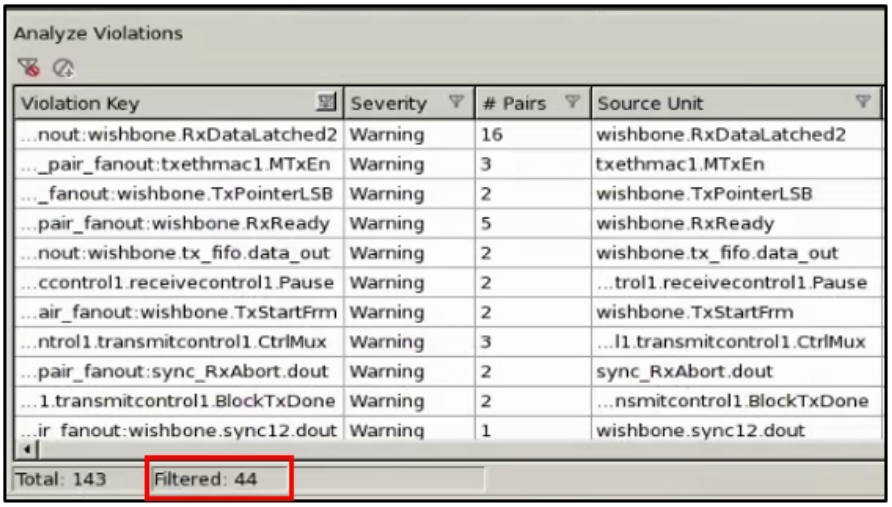
- Use of wildcard or regular expression that exactly matches or is contained in the available data
- Use of wildcard or regular expression that does not match the specified string in the available data

Filters can be applied on all the columns in the *Analyze Violations* table.

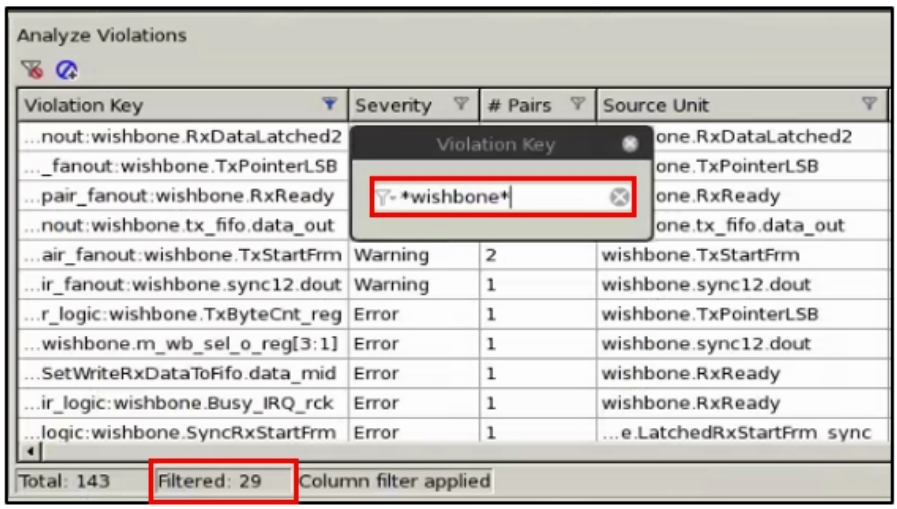


Example:

In the figure below, you can see that a total of 44 violations of a certain tag have been reported.

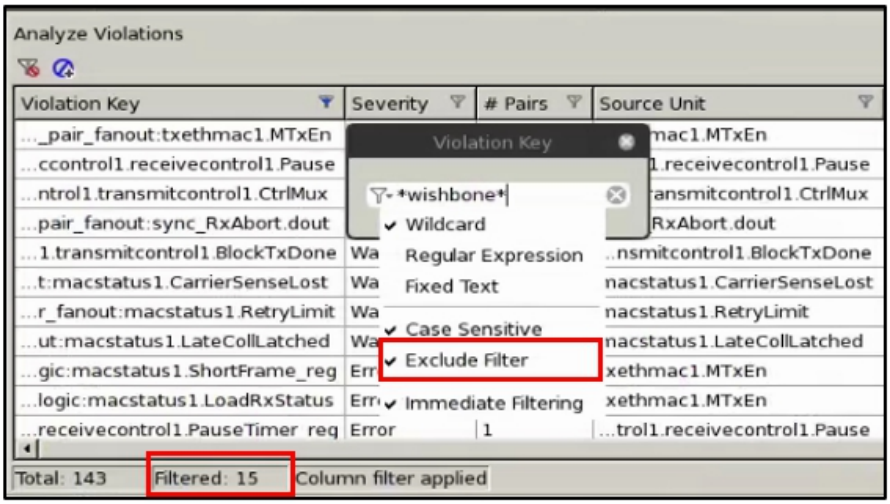


Matching Filters: When you enter **wishbone** in the *Violation Key* column filter, the tool reports only those violations that match this expression, and the count is reduced to 29.



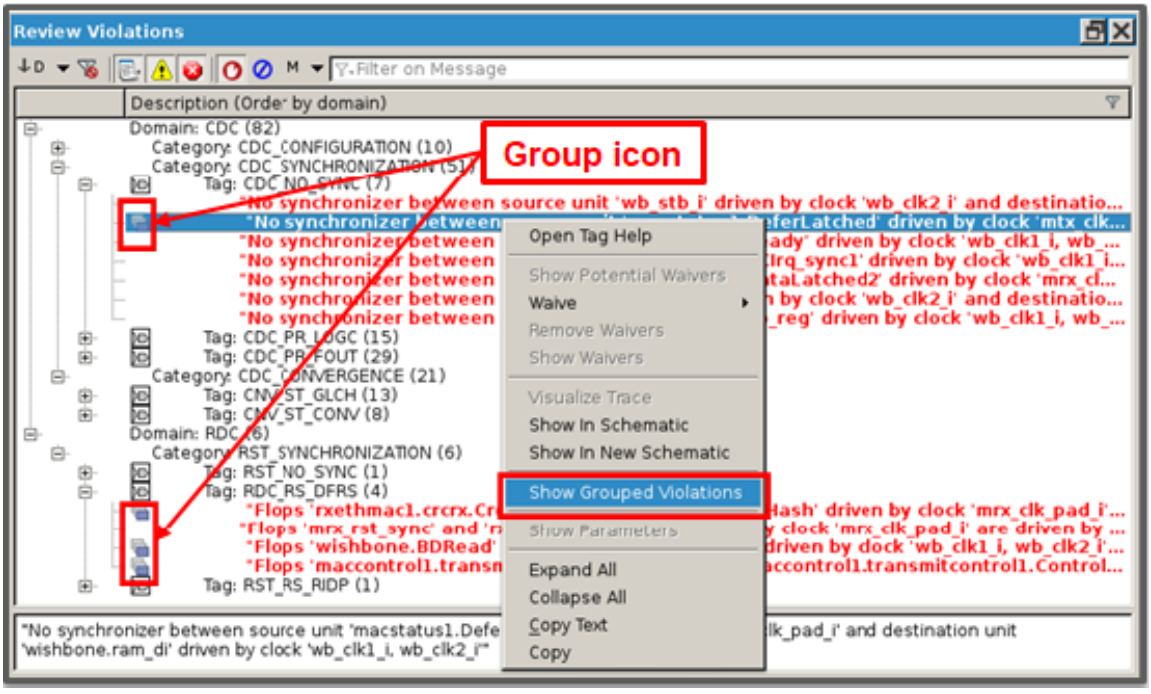
Does Not Match Filters: When you specify the *Exclude Filter* option, the tool reports only those

violations that do not match the specified expression **wishbone**, and the count is reduced to 15.

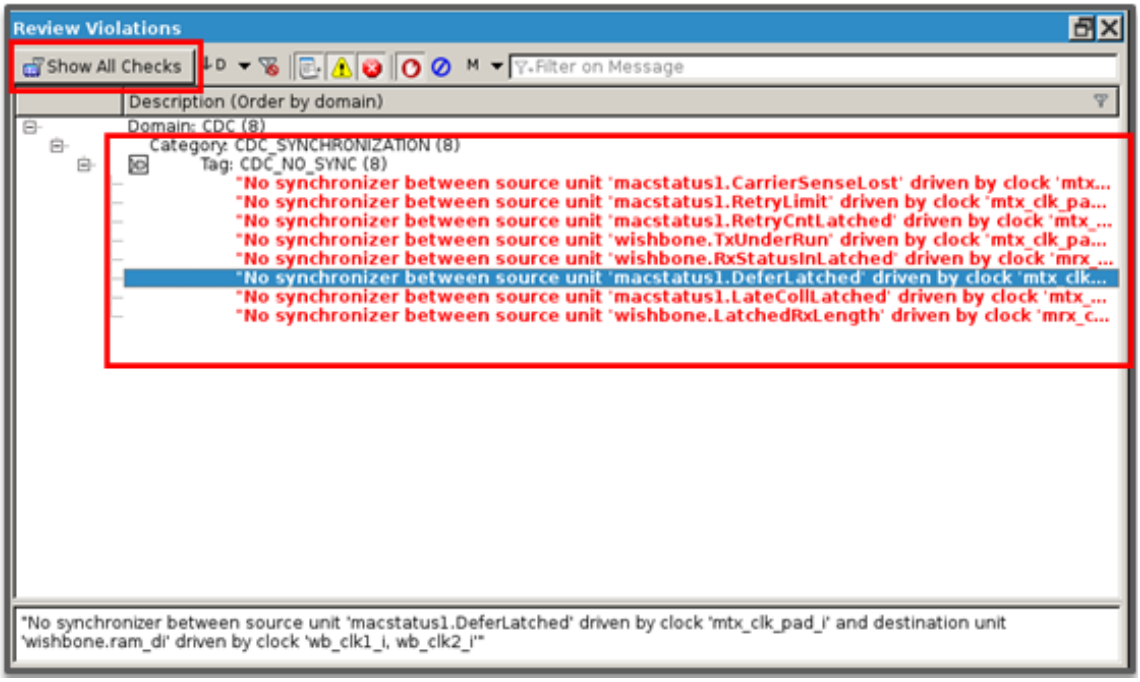


Violation Grouping

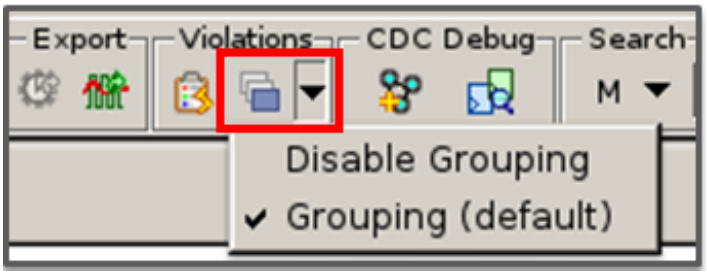
From 2023.06 onwards, the CDC app groups violations under the same tag based on common root cause. A group of violations is displayed as a single message on the Violation tree with a special icon representing a group of violations. Violation grouping reduces the time spent on dispositioning violations since debugging one violation in the group is enough to understand the root cause. Fixing one of the violations has a high probability of fixing the other violations in the group, thereby increasing productivity.



You can review all violations in a group by the *Show Grouped Violations* context-menu. In the expanded view, you can debug or waive any violation. To return to the previous view containing all violations, click on the *Show All Checks* button.



The violations are grouped by default. However, you can enable or disable grouping from the drop-down menu under *Violations* on the CDC toolbar.



To enable or disable grouping from the command line, you can run the following command anytime during the run.

```
% check_cdc -violation -group (none | basic)
```

⚠ Since the CDC app now groups violations based on common root cause, violations like `CDC_NO_SYNC`, `CDC_PR_LTCH` that can involve multiple pairs are not merged anymore. Multi-pair violations that were reported in versions prior to 2023.06 will now be grouped. Hence, from 2023.06 onwards, you don't need to split such multi-pair violations for the purpose of waiving. The command `check_cdc -violation -split` has been deprecated.

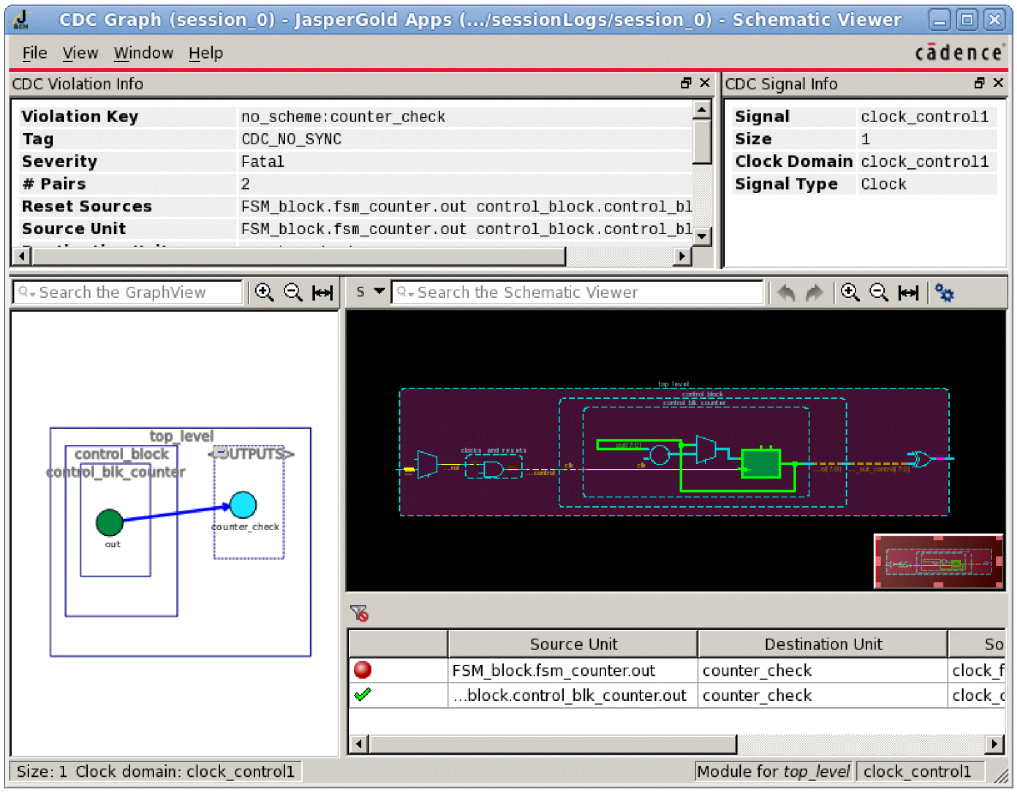
CDC Debugging Environment

The CDC App provides an advanced debugging environment where multiple debugging views, such as schematic, graph, waveform, and source code browser, are integrated together for an enhanced debugging experience. The *Show Schematic* or *Visualize Trace* context-menu options in the *Review Violations* table open a debugging view suitable for the particular violation type. For example, any structural violation is best debugged using the schematic viewer. Thus, CDC opens an integrated schematic+graph to facilitate the debugging process.

Structural Violations

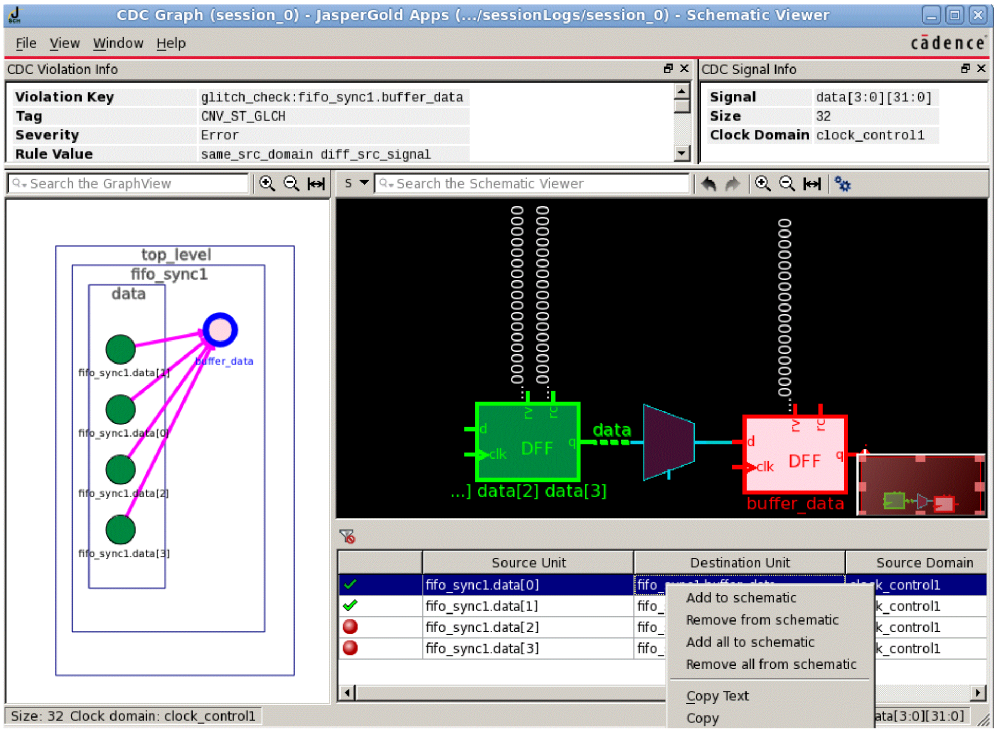
When you choose *Show Schematic* on a structural violation, the tool opens a schematic+graph view for debugging. While the schematic view shows the relevant section of the design related to the violation, the graph shows an abstracted view of the same logic. Both views are updated automatically based on any additional information plotted on either side. A *Violation Info* pane at the top of the schematic plus graph window shows all violation-related information, and the *Signal Info* pane shows relevant signal information. Use the context-menu of either view to perform various actions, for example, view fanin or fanout and open the source code browser.

⚠ The CDC Schematic Viewer is disabled for convergence violations with greater than 1000 signals. For these violations, CDC opens the graph only.



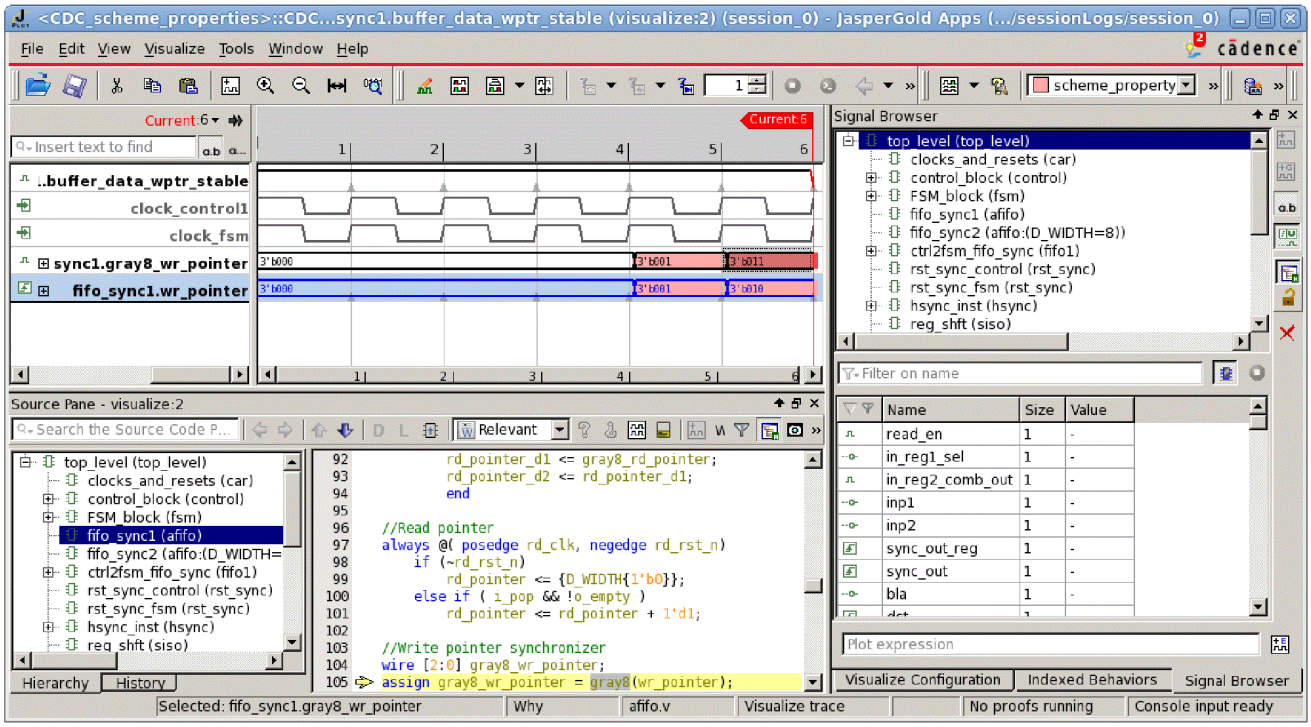
Structural Violations with More than One Associated CDC Pair

When you choose *Show Schematic* on a convergence or pair violation, the tool opens a schematic+graph view for debugging that automatically loads the signals related to the first CDC pair only (Figure 6-5). At the bottom of the schematic is a table with the source and destination unit, source and destination domain, and status of all CDC pairs in the violation. You can add signals related to any CDC pair by right-clicking on a signal in this table and choosing *Add to schematic.* You can also remove signals from the schematic by right-clicking and choosing *Remove from schematic*. Or you can use this context menu to add or remove all signals at once. When applicable, the CDC graph shows only the nodes related to the CDC pair.



Functional Violations

When you choose *Visualize Trace* on a functional violation, the tool opens a Visualize window, providing a waveform-based debugging environment. With Visualize, you can do various *What if* analyses, and since Visualize is coupled with a source browser, signals can be plotted to the waveform from the source browser.



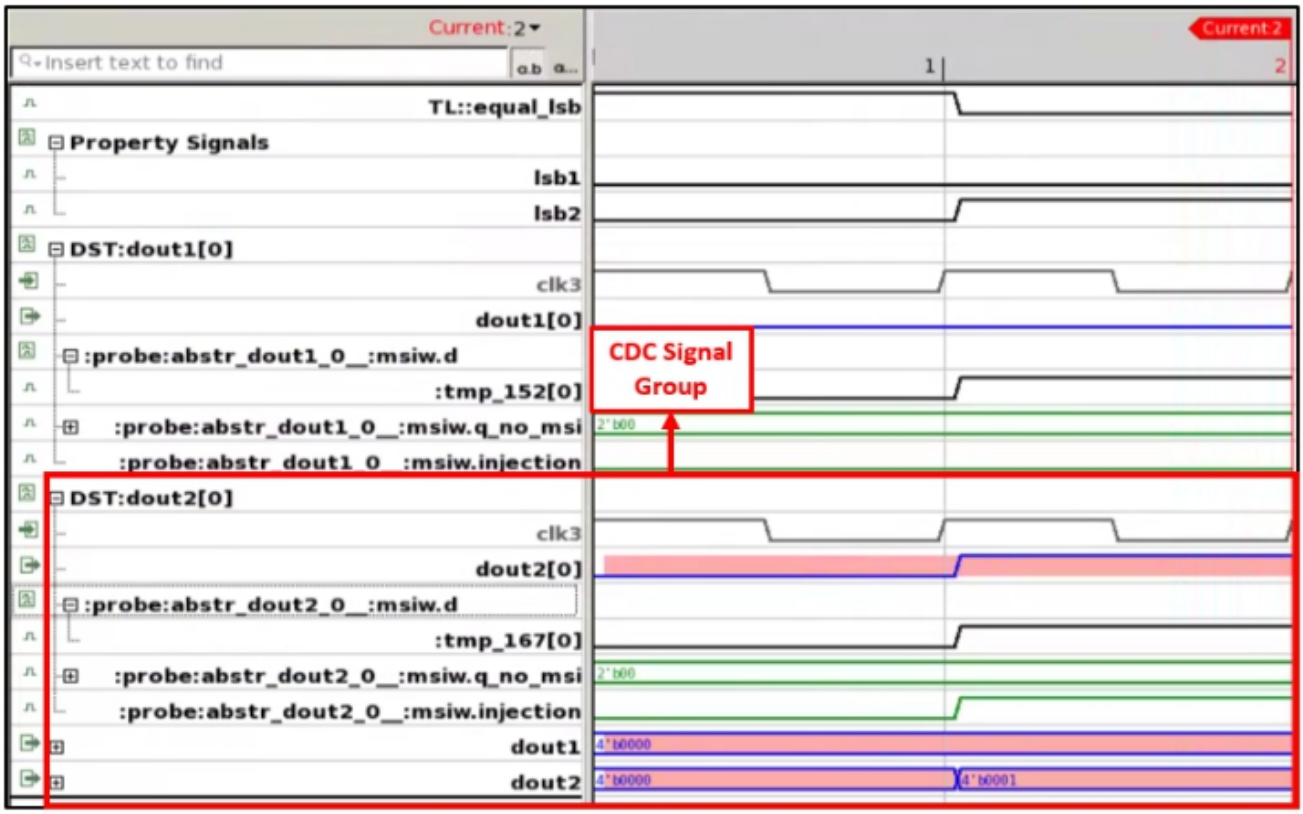
Metastability Violations

When you choose *Visualize Trace* on a metastability violation, the tool opens the debugging environment for analyzing property failures due to metastability injection. This debugging environment is the same as the functional violation debugging environment discussed above, but it includes additional customization for metastability related information. To facilitate the debugging process, the tool highlights the signals that have been affected due to metastability with red in specific clock cycles. This helps you concentrate on specific cycles for the root cause analysis.

A group of CDC signals visible in the Visualize window helps you understand a Metastability CEX. The group contains the following signals:

- D-Input of the destination flop
- Destination clock
- Two versions of destination flop output (with and without) metastability Injection
- Injection witness signal

This helps you investigate the MSI behavior at each crossing.



Schematic+Graph and Visualize

This section discusses CDC schematics and graphs in greater detail. For additional information on waveform analysis with Visualize, see *Help – Application Guides – Jasper Visualize GUI Features* or the "Visualize" chapter in the *Jasper Platform and Formal Property Verification App User Guide*.

- Schematics
- Graphs

Schematics

You can access simple clock schematics, structural schematics, or scheme schematics from the context menus of the following tables:

- *CDC Configuration* tab – *Clock Domains* and *Port Configuration* tables (Right-click on a signal and choose *View in Schematic*.)
- *CDC Phases* Tables

- *Pairs* (Right-click on a CDC pair and choose *View in Schematic*.)
- *Schemes* (Right-click on a scheme and choose *View in Schematic*.)
- *Convergence* (Right-click on a CDC group and choose *View in Schematic*.)
- *Functional* (Right-click on a scheme property and choose *Show Scheme Schematic*.)

Note:

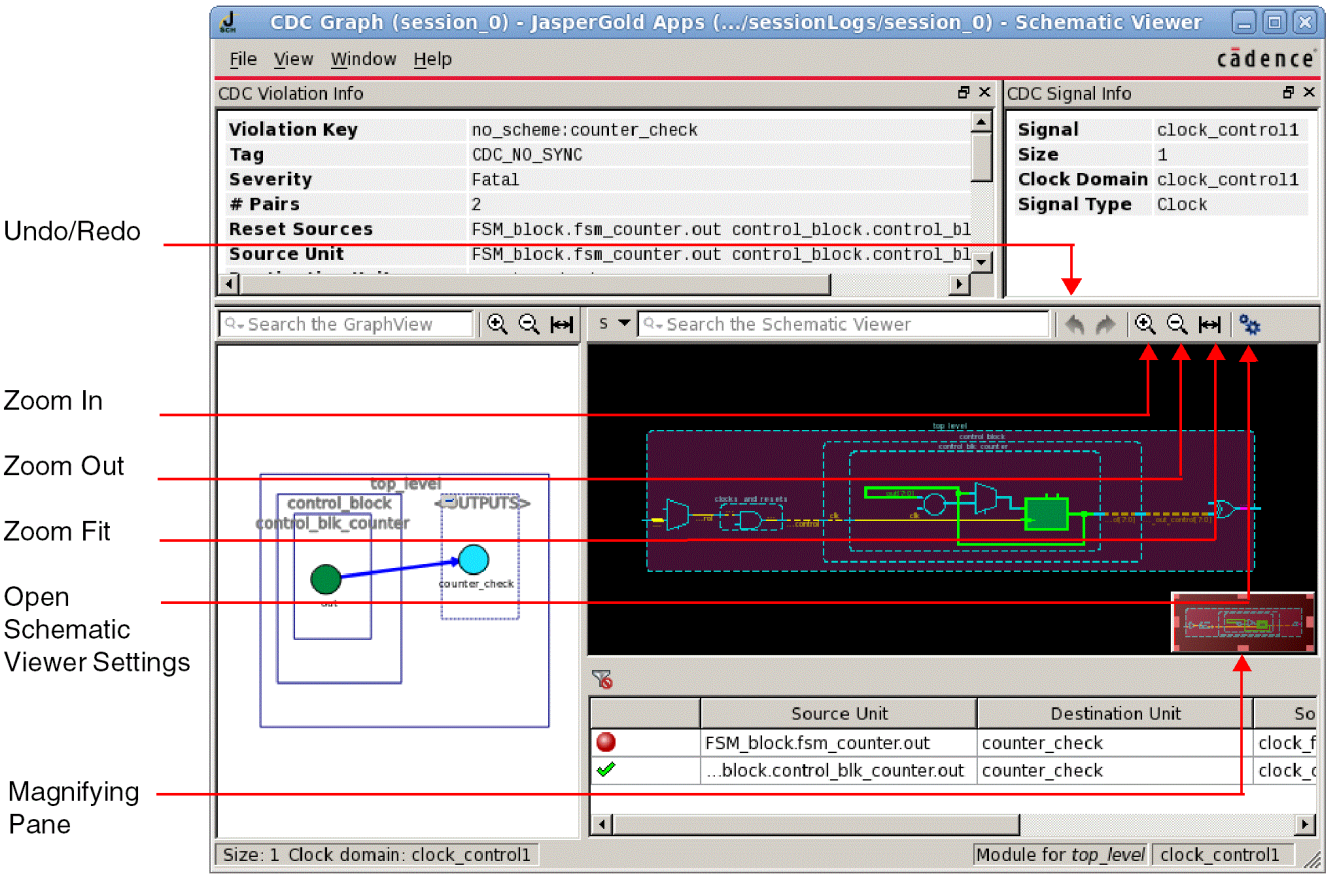
- By default, structural schematics include graphs, but the following discusses features of the schematic viewer only. See "Graphs" for a brief discussion of CDC graphs.
- Structural schematics are disabled for convergence violations with greater than 1000 signals. For these violations, CDC opens graphs only.
- For additional information on the Jasper Apps Schematic Viewer, see the "Schematic Viewer" chapter in the *Jasper Platform and Formal Property Verification App User Guide*.

The *Schematics* section includes information on the following:

- Schematics (Overview)
- Schematics (Icons)
- Schematics (Highlighting CDC Units)
- Signal Info Table (Reset and Clock)
- Signal Info Table (Manually Rated Ports)
- Signal Info Table (Constant or Static Signal Configuration)
- Signal Info Table (Static Value Propagation)
- Signal Info Table (Multi-Bit Signal)
- Tracing the Fanin/Fanout

Schematics (Overview)

See the following for an overview of the CDC schematic.



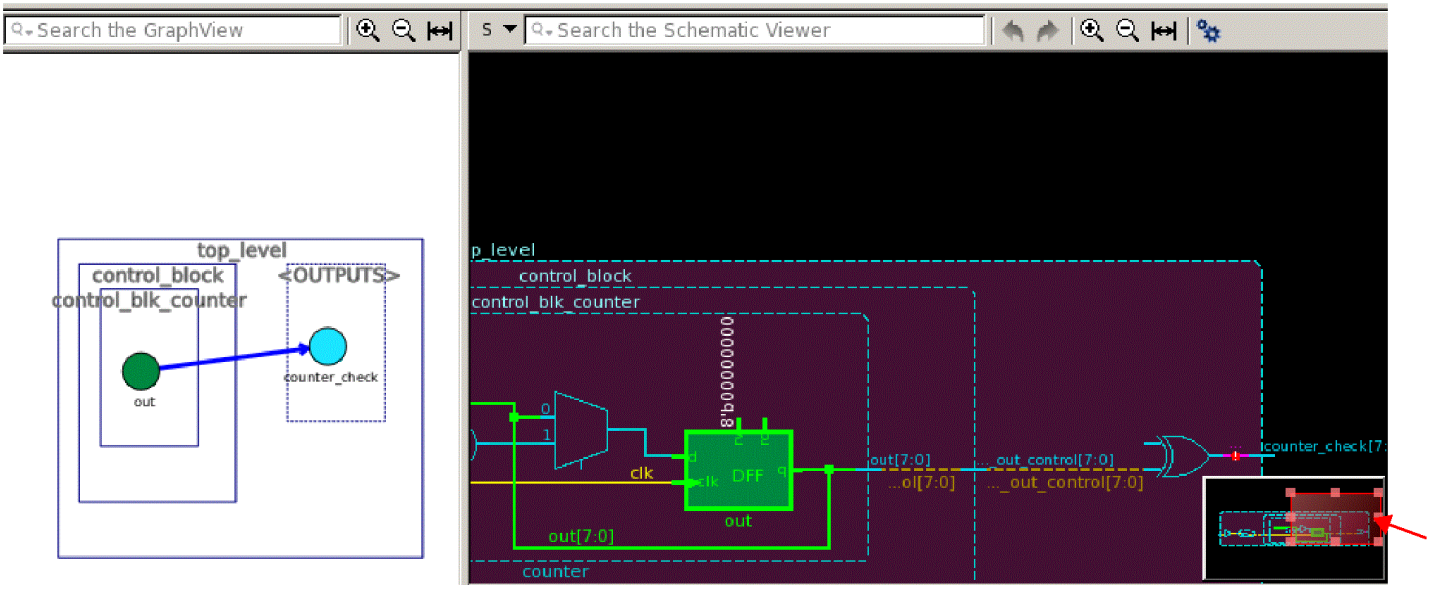
To undo or redo the last action only, click on the *Undo* or *Redo* button.

To zoom in on a specific element of the schematic, do any of the following:

- Left-click on the schematic and drag the mouse to select the section you want to zoom in on.
- Left-click over the magnifying pane to grab the magnifier and drag it to the desired view (see Figure 6-2).
- Click on the *Zoom In* button.
- Use the wheel on your mouse to zoom in and out.

Click the *Open Schematic Viewer Settings Dialog* button to open the *Schematic Viewer Settings* dialog box. This dialog box provides the option to specify the behavior when double-clicking on a wire or pin. You can choose the default, which is the equivalent of the *Add Objects* context menu option, or you can choose to *Add objects until instance boundary, flop, or port.*

Figure 5.2: Structural Schematic Magnifier Pane



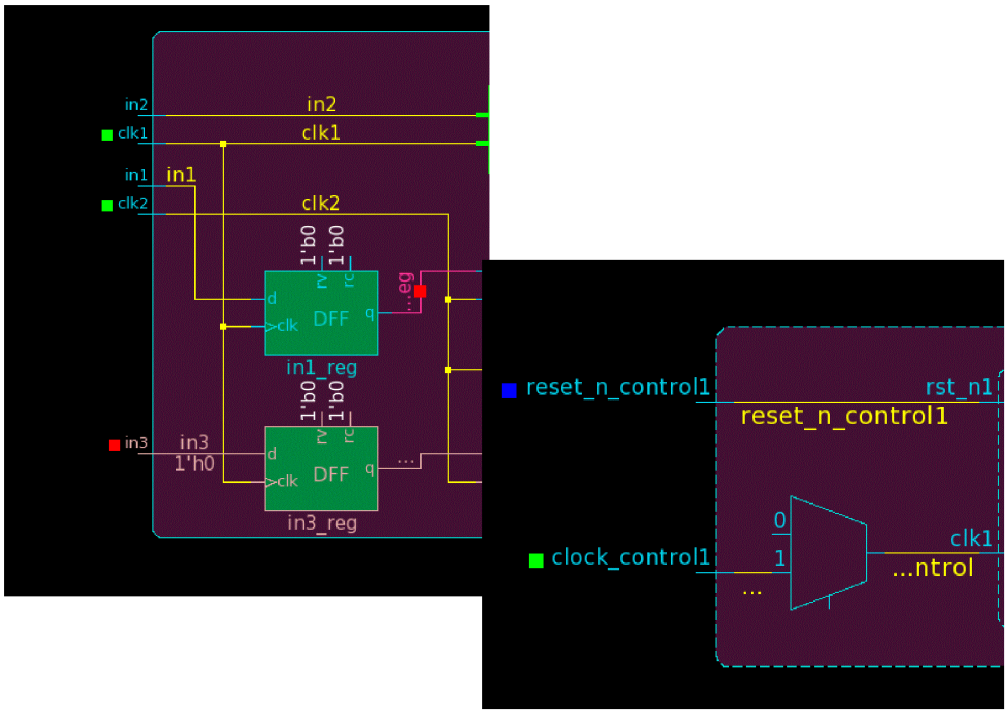
Schematics (Icons)

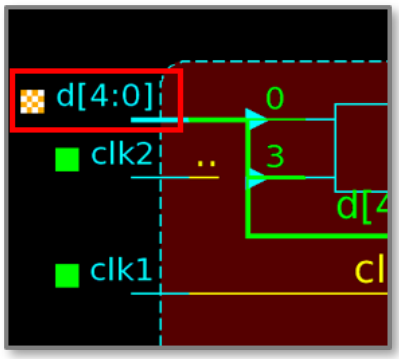
The CDC schematic uses color-coded icons as follows to identify various elements:

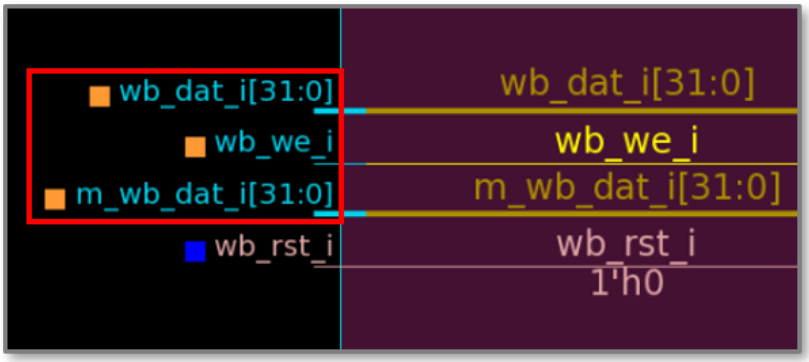
- Green square – Clocks
- Blue square – Resets
- Red square – Constant and static signal nodes
- Solid orange square – Ports with user-defined clock association (all bits manually rated)
- Orange square grid – Ports with user-defined clock association (some bits manually rated and some unclocked)
- Red "skip" icon – Clocks stopped by the `set_clock_sense -stop_propagation` command

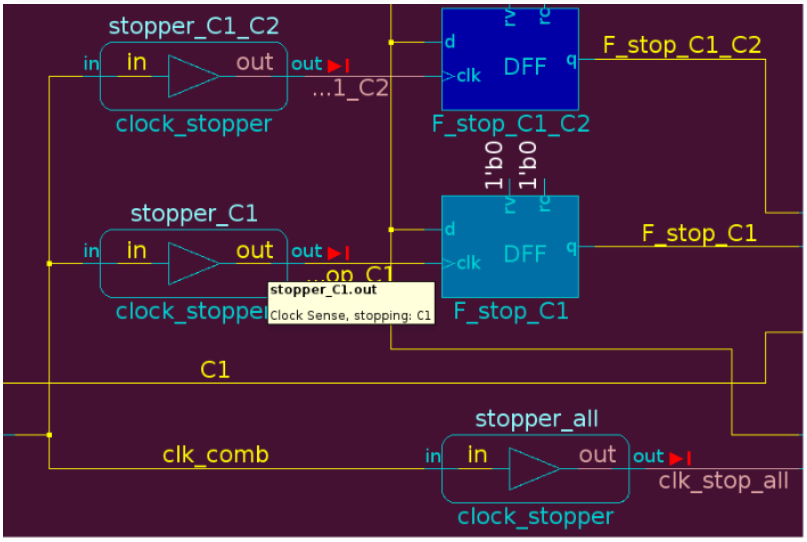
For multi-bit signals, each bit of the bus contributes at most one icon, and there can be a maximum of three icons on a certain bus. Reset or clock has priority over signal configuration, that is, a constant clock bit shows green only and not red and green.

Figure 5.3: Colored-Coded Icons


in2
clk1
in1
clk2
in3
1'h0
DFF
in1_reg
in3_reg
reset_n_control1
rst_n1
clock_control1
clk1
...ntrol

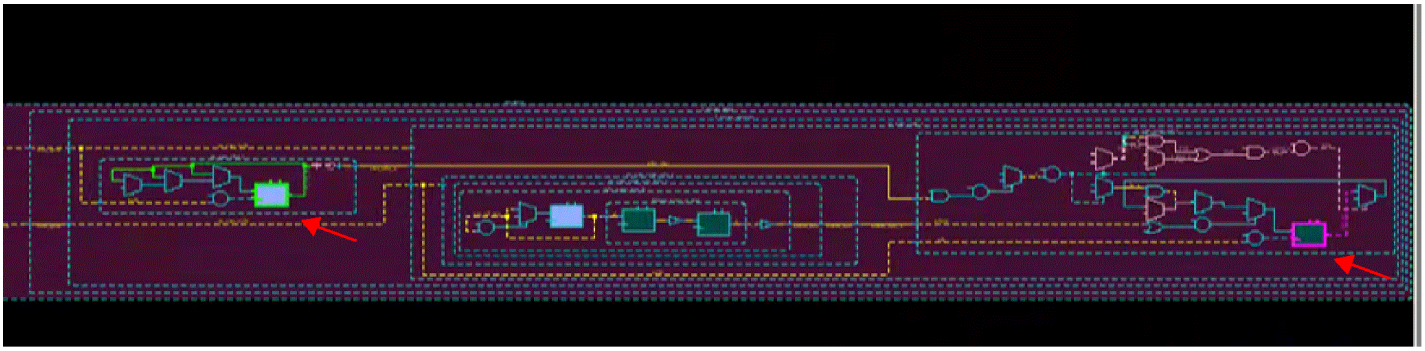

d[4:0]
clk2
clk1
0
3


wb_dat_i[31:0]
wb_we_i
m_wb_dat_i[31:0]
wb_rst_i
1'h0


stopper_C1_C2
clock_stopper
F_stop_C1_C2
stopper_C1
F_stop_C1
C1
stopper_all
clk_comb
clk_stop_all
DFF

Schematics (Highlighting CDC Units)

The schematic viewer uses thicker borders to highlight source and destination units as shown in the figure below. The CDC source unit is highlighted in green, and the CDC destination unit in dark pink.

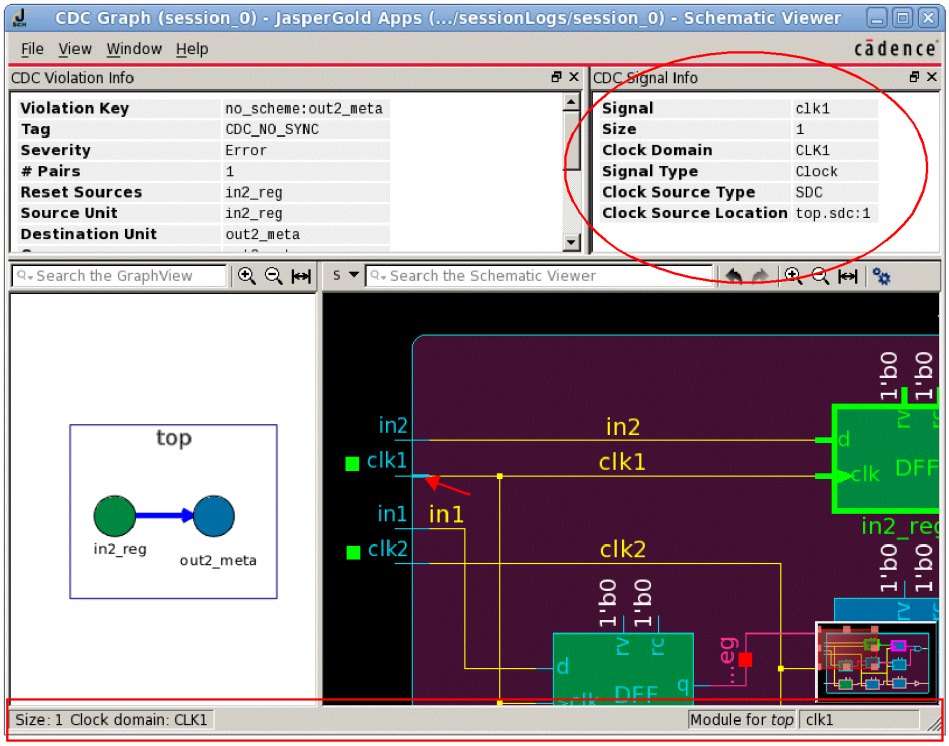


Signal Info Table (Reset and Clock)

The *Signal Info* table provides information on signal name, size, and clock domain, which is also displayed in the schematic footer. If the selected net is a reset, clock, or multi-bit signal that contains reset or clock bits, the signal info table also includes additional information as follows:

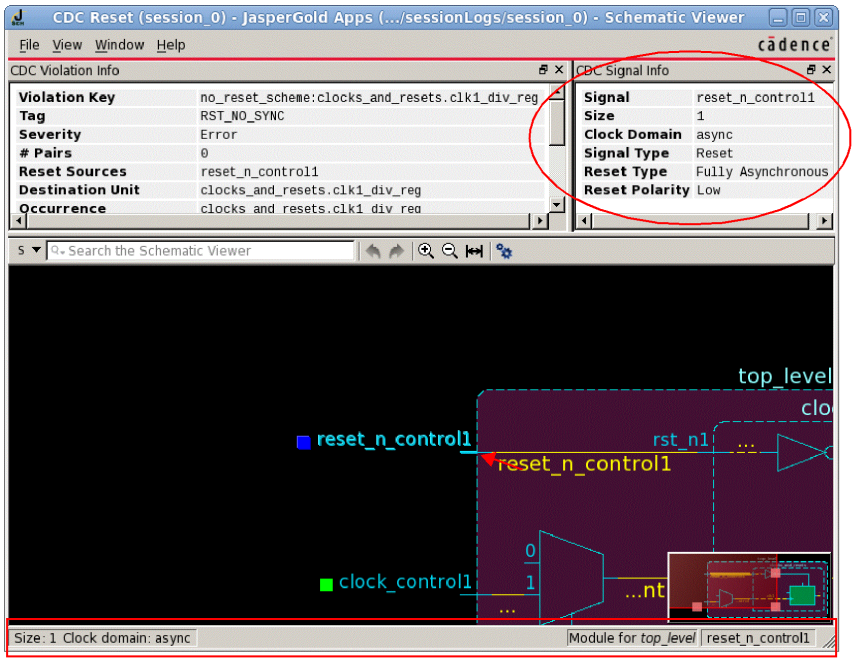
- If the selected net is a user-declared clock, the *Signal Type* is *Clock*. In addition, if the declared clock is an SDC declared clock, the tool displays two additional entries are shown below:
    - *Clock Source Type – SDC*
    - *Clock Source Location* – Relevant only when the *Clock Source Type* is *SDC* (shows the value that represents the source location of the declaration in the SDC file)

Figure 5.4: Signal Info Table (Clock)



- If the selected net is a declared reset, the *Signal Type* is *Reset*. In addition, if the reset is declared using config_rtlds -reset, the tool displays the following information:
  - *Reset Type* – *Fully Asynchronous*, *Fully Synchronous*, or *Sync Deassert*
  - *Reset Clock* – Represents the clock associated with the reset (relevant for synchronized and sync RTLDS resets)
  - *Reset Polarity*
  - *Reset Signal Source* – Can be *Tcl* or *Console*
  - *Reset Signal Location* – Relevant only when the *Reset Signal Source* is *Tcl*

Figure 5.5: Signal Info Table (Reset)



Note:

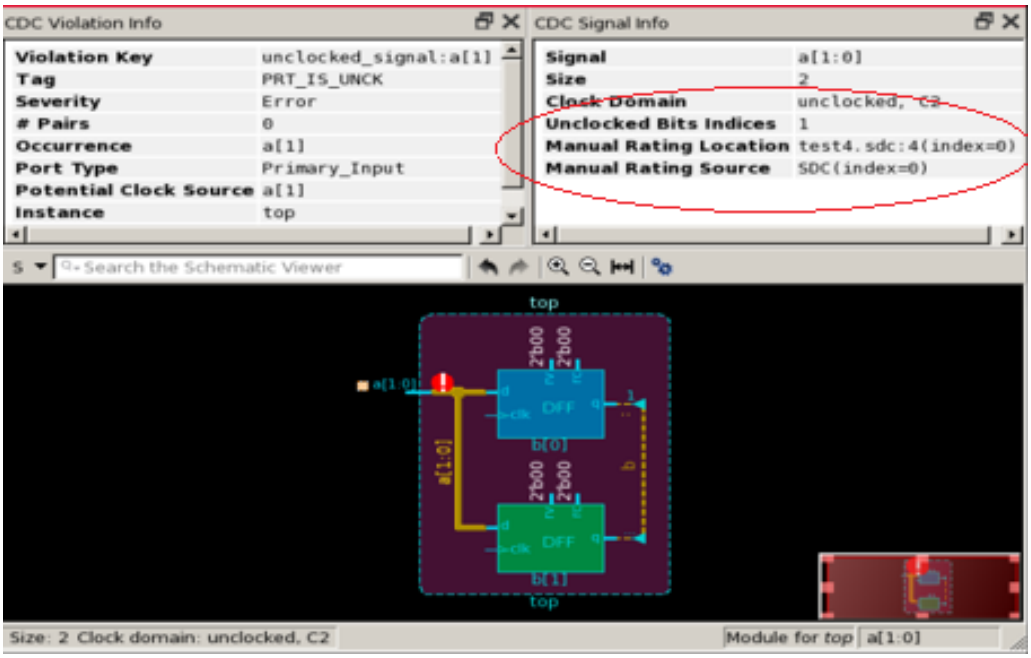
- Only the source of `config_rtlds` resets and SDC clocks are shown (this data is not shown for Jasper resets and clocks).
- This reset information can be seen only after running the reset phase, that is `check_cdc -reset -find`.

Signal Info Table (Manually Rated Ports)

For manually rated ports, the *Signal Info* table includes additional information as follows:

- *Manual Rating Location* – Relevant only when the rating is through SDC or through a sourced file that includes `check_cdc -clock_domain -port` commands. The format is `<filename>:<linenumber>` of the rating command.
- *Manual Rating Source* – Represents the source type of the rating, which can be *Console*, *TCL*, or *SDC*.
- *Unclocked Bits Indices* – Lists the indices of the bits that are unclocked in the selected port.

A solid orange icon indicates that all the bits of the port are manually rated, and an orange square grid icon indicates that some of the port's bits are manually rated and some are unclocked.



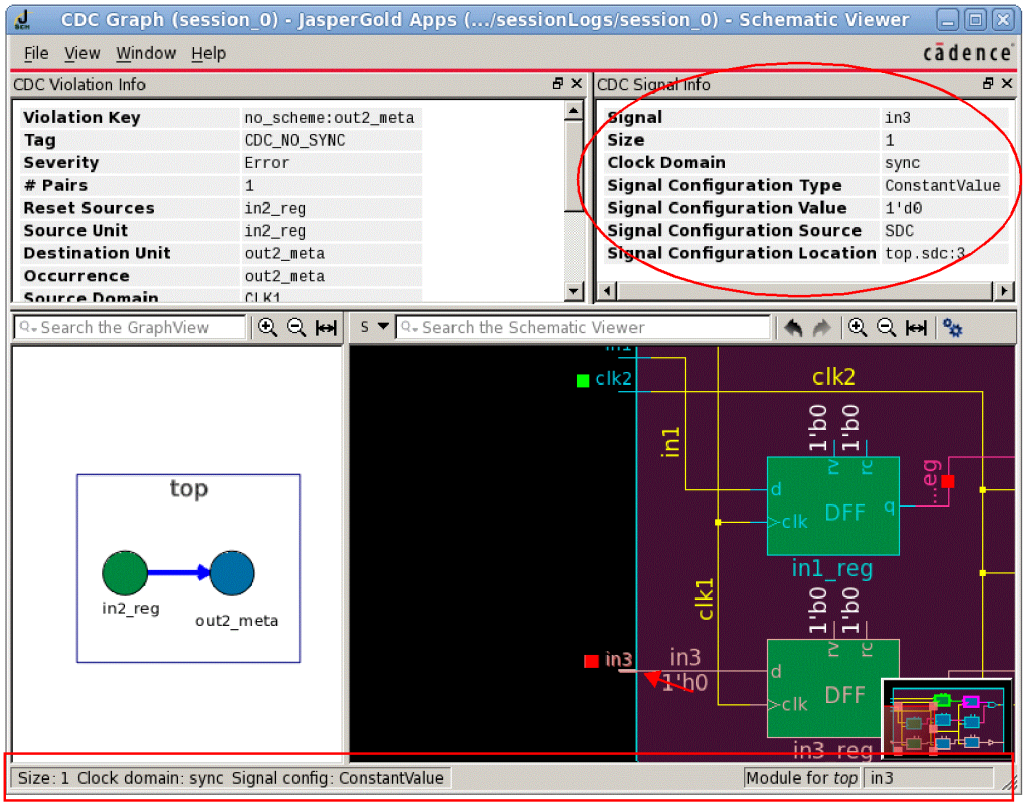
You can manually rate a port using either the `check-cdc -clock_domain -port` or SDC `set_input_delay` and `set_output_delay` commands.

Signal Info Table (Constant or Static Signal Configuration)

For constant or static signals, the *Signal Info* table includes information on the origin of the signal configuration. The table shows the type of the configuration, the value of the signal (if it is a constant), and the origin of the configuration (source type and location). See the following for specifics:

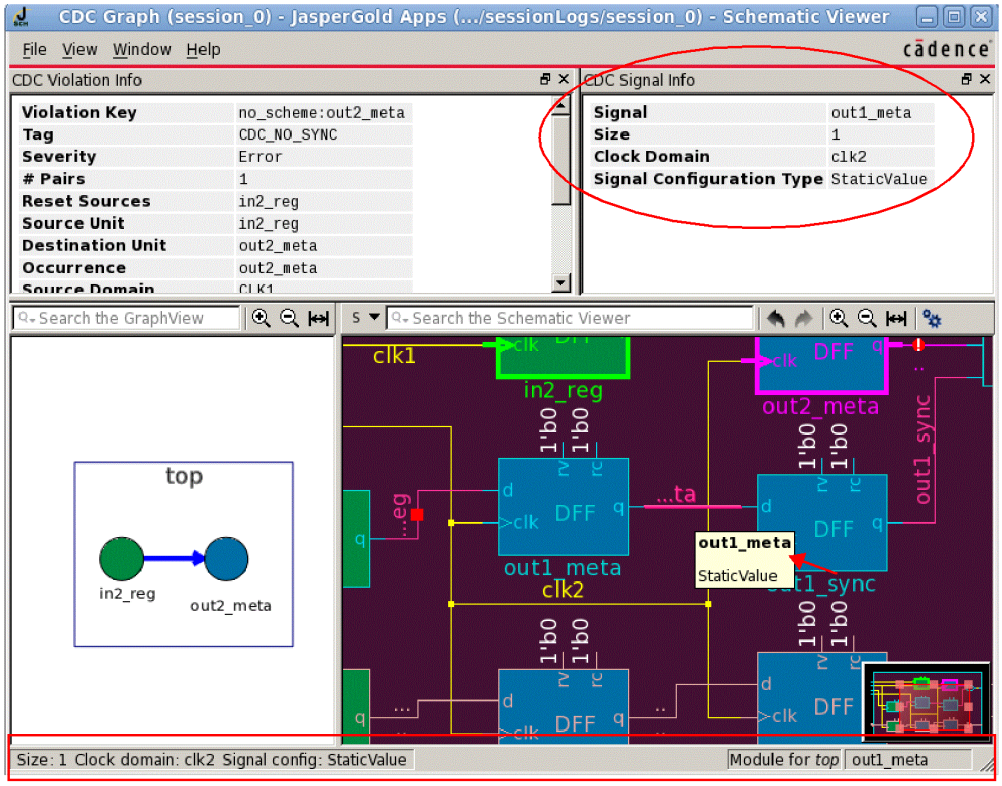
- When you declare a signal configuration from the GUI schematic or console, the source type is *Console* and the source location is empty.
- When you source a file that contains the signal configuration command using `source <file>`, the source location is <*file*>:<*line*>, where `<file>` is the sourced file and `<line>` is the number of the line containing the configuration command in `<file>`.
- When you source an SDC file that contains SDC commands that are translated to CDC, the source location is the name of the SDC file and the number of the line containing the SDC command. Further, the source type in this case is SDC.

  ⚠ If you export the SDC file to the translated Tcl file and then source the exported file, the source location contains the name of the exported file and the source type is Tcl.



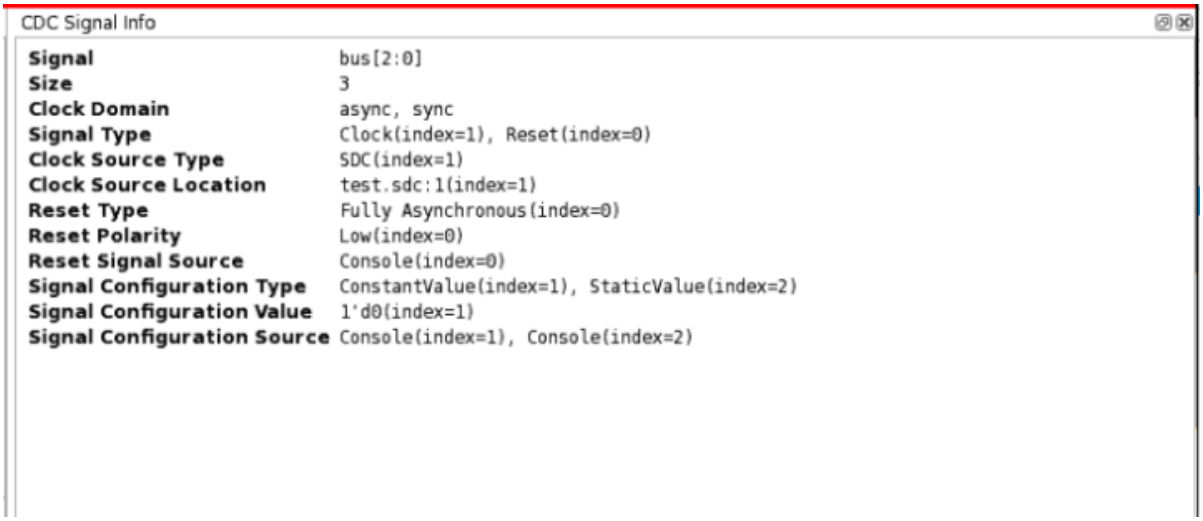
Signal Info Table (Static Value Propagation)

For static value propagation, static nets are colored dark pink, the tool tip shows the *StaticValue*, and the status bar shows *Signal Config: StaticValue*.



Signal Info Table (Multi-Bit Signal)

For multi-bit signals, the table shows the indexes that contribute to the information displayed so you can trace a single bit through a bus to better understand bit mapping.

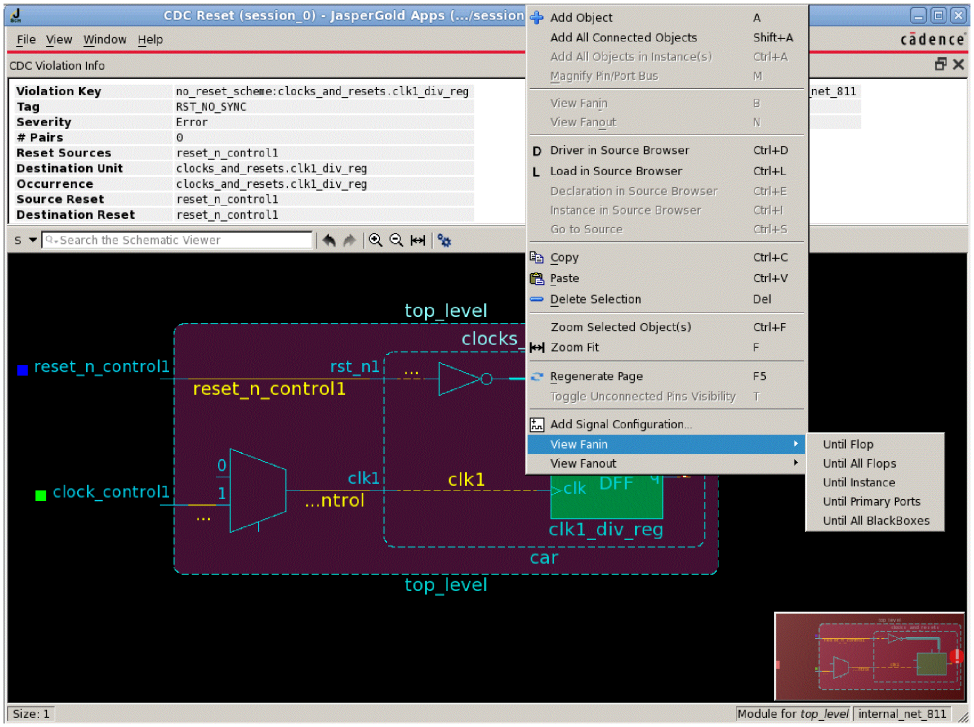


Tracing the Fanin/Fanout

The CDC schematic viewer provides several options for viewing fanin or fanout as follows:

- Until Flop
- Until All Flops
- Until Instance
- Until Primary Port
- Until All BlackBoxes

To access these options, right-click on the relevant object and choose *View Fanin* or *View Fanout.*



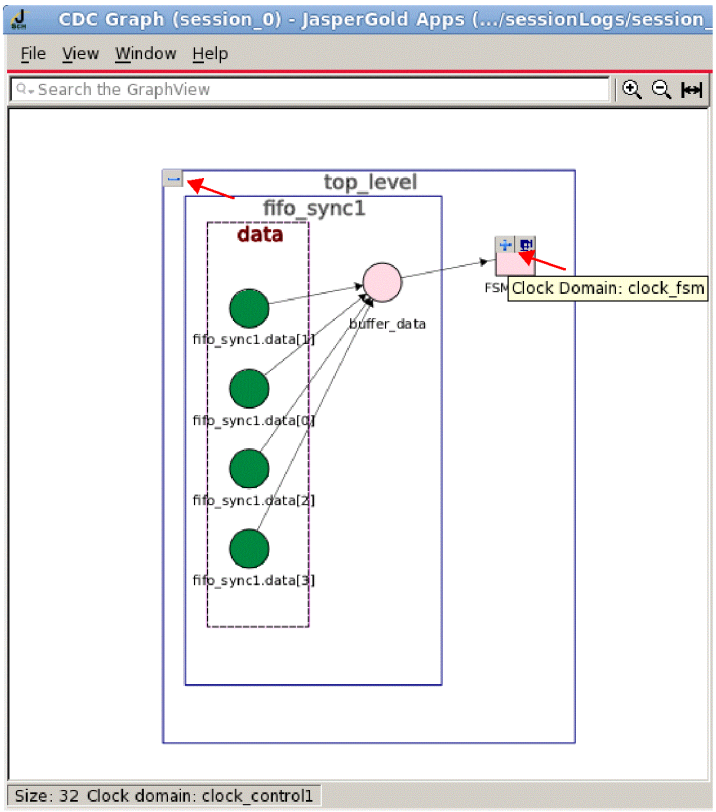
Graphs

You can access a graph of a property or violation in any of the following ways:

- Click the *Show CDC Graph and Schematic* button on the *CDC Debug* wizard.

- Right-click on a violation in the *Pairs, Schemes*, or *Convergence* tables and choose *View in Schematic*.
- Right-click on a violation in the *Analyze Violations* table and choose *Debug Violation*.
- Right-click on a property in the *Functional* or *Metastability* table and choose *Show Property Graph* or *Show Graph*.

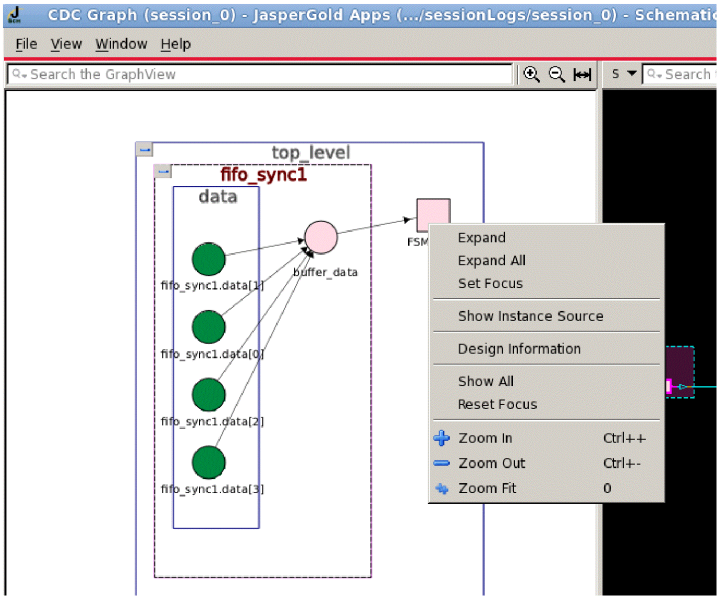
The graph presents a high-level view where nodes represent flops, arrows represent paths, and different colors represent different clock domains. You can expand or collapse detail for square nodes, which represent blocks, by hovering over the block and clicking on the (+) or (–) that appears in the top left corner.



In addition to various zoom functions, context-menu options (right-click) for unexpanded blocks include the following:

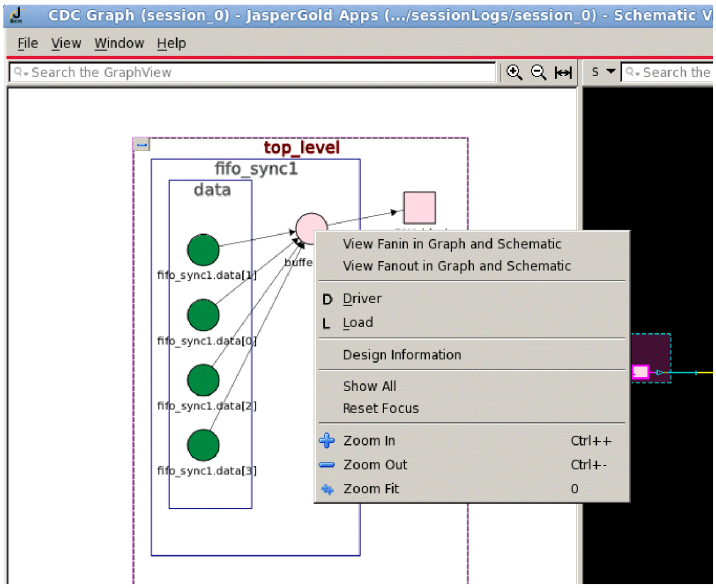
- *Expand* – Expands the selected node
- *Expand All* – Expands all nodes in the block
- *Set Focus* – Sets the focus to the selected instance node
- *Source Instance Source* – Opens the source browser for the instance

- *Design Information* – Opens Design Information for the selected node
- *Show All* – Expands all
- *Reset Focus* – Collapses any expanded instance on the current graph and resets the focus to the top instance



Additional context-menu options for expanded nodes include the following:

- *View Fanin in Graph and Schematic* – Shows the fanin for the selected node and expands the schematic accordingly
- *View Fanout in Graph and Schematic* – Shows the fanout for the selected node and expands the schematic accordingly
- *Driver* – Opens the source code browser and highlights the driver for the selected node
- *Load* – Opens the source code browser and highlights the load for the selected node



To get more information on debugging CDC violations, click the following link to navigate to the Jasper landing page on Cadence Online Learning and Support: https://support.cadence.com/jasper. From there, click the CDC button, scroll down, and follow the "Debugging Jasper CDC Violations" link under "Application Notes".

Waiving Violations

Though you can resolve violations in various ways, for example, by altering the RTL, modifying the structural path rule attributes, or using a different synchronizer, you can also choose to waive violations if you determine that it is safe to do so. You can waive single violations, groups of violations, violations that satisfy a specified expression, or you can automatically waive violations.

This section includes procedures for the following:

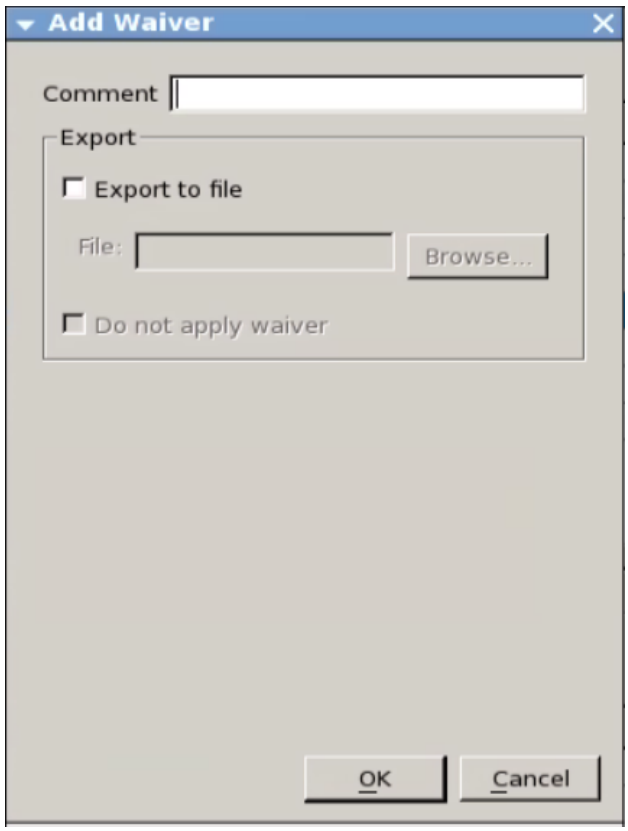
- Waiving Single Violations
- Waiving Groups of Violations
- Handling Multi-Pair Violation Waiver
- Waiving Structural Violations by Hierarchy

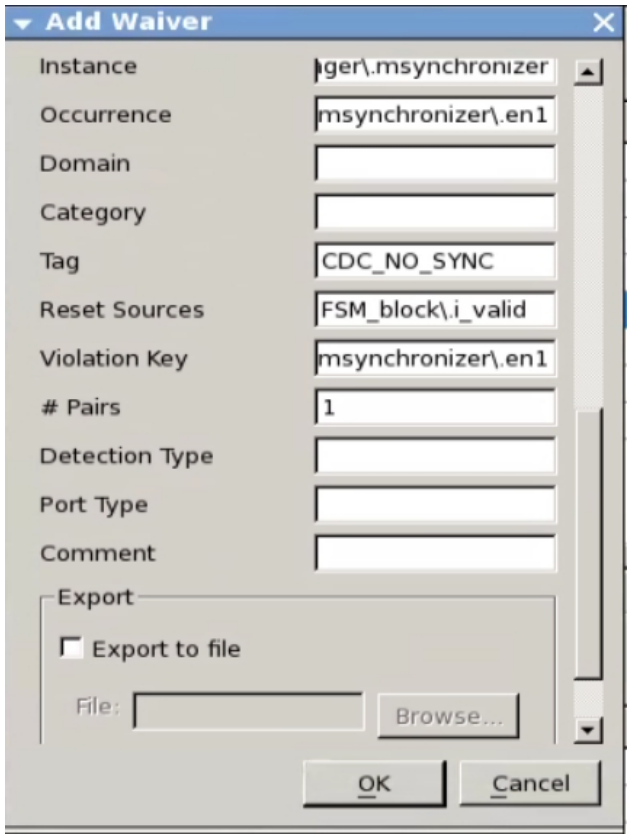
Waiving Single Violations

To waive a single violation, do the following:

1. Right-click on a violation in the *Review Violations* table and choose *Waive – Add Waiver Comment* or *Waive – Open Add Waiver Dialog*.

   The *Add Waiver* dialog box opens (see the following figures).





2. Add a comment to the *Comment* field.

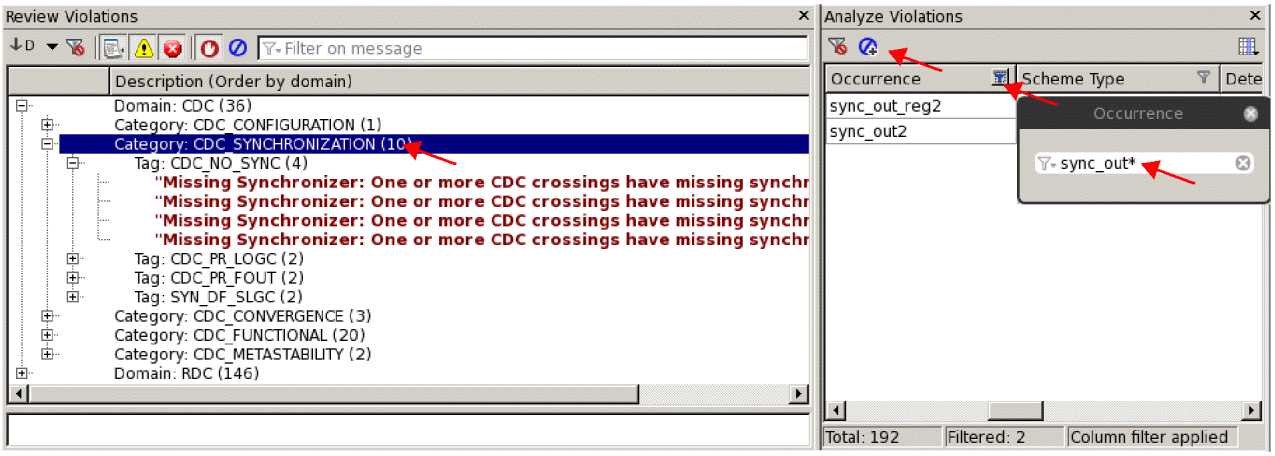
   ⚠ This field is mandatory regardless of which *Waive* option you choose. After you have added a comment, you can edit it with the following command if necessary: `check_cdc -waiver -edit <waiver_id> -comment <new_comment>`

3. Select the *Export to file* check box to choose to export the given waiver to a specified file.
4. Specify the file in the *File* field.
5. Select the *Do not apply waiver* check box to choose not to apply the waiver in the current session.
6. Click *OK*.
   The violation is waived, and the status icon changes from red to blue.

Waiving Groups of Violations

To waive a group of violations, you can set GUI filters and then use the *Waive all violations that match the filters* button on the *Analyze Violations* toolbar. To set a GUI filter, do the following:

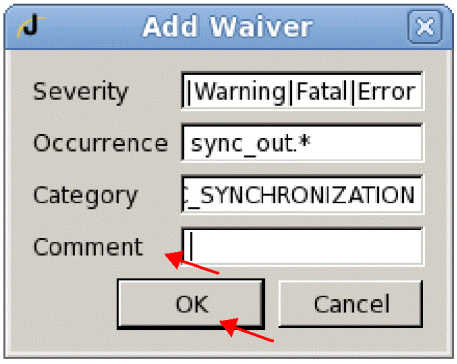
1. Choose which category of violations you want to filter in the *Review Violations* table.
2. Chose which column you want to filter from the *Analyze Violations* table and click on the *Change filter options* button at the top right of the column heading.
   A control box appears.



3. Enter text to filter the contents of the column by wildcards (`*` and `?`).
   For example, enter `sync_out*` in the *Occurrence* filter box to filter out all violations except those related to `sync_out*` occurrences (see the figure above).

> ⚠ All filter criteria support a wildcard expression by default, but you can use `-regexp` or `-literal` with `check_cdc -filter -add` to specify that the criteria be filtered by regular expression or fixed string instead.

4. Click on the *Waive all violations that match the filters* button on the *Analyze Violations* toolbar.
   The *Add Waiver* dialog box appears.



5. Complete the *Comment* field, which is mandatory.
6. Click *OK*.

The icons associated with the waived failures change from red to blue, indicating these pairs have been waived, and the tool adds waived failures to the *Waivers* tables.

Handling Multi-Pair Violation Waiver

There are certain violations reported in the CDC App that involve multiple CDC pairs merged in a single violation. Such violations need to be handled differently when compared to other single pair violations reported by the tool.

This section describes how to handle multi-pair violation waivers and how to waive such violations.

Waiving Multi-Pair Violations

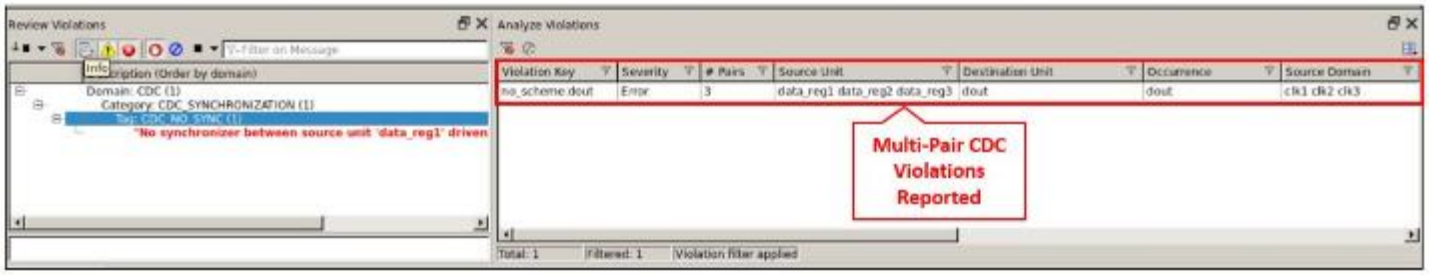
If the tool reports multi-pair violations and you want to waive only a set of pairs in the merged violation, you can use the following command:

```
% check_cdc -waiver -add -filter filter_id -comment <comment> [-split]
```

Use `-split` to automatically split multi-pair violations based on the source and/or destination unit of the provided filter and waive only those pairs that match the filter criteria.

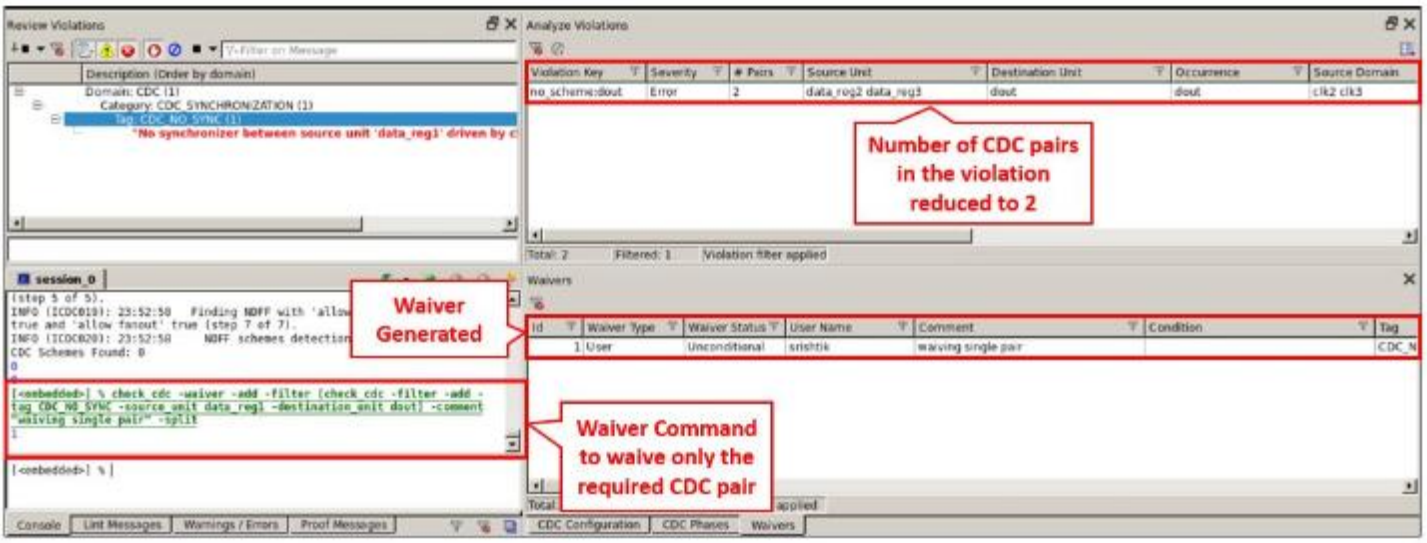
Example:

In the following example, a multi-pair `CDC_NO_SYNC` violation containing three merged CDC pairs has been reported by the tool.



However, only one CDC pair needs to be waived as per the design requirement. To do that, use the following command:

```
% check_cdc -waiver -add -filter [check_cdc -filter -add -tag CDC_NO_SYNC -source_unit
data_reg1 -destination_unit dout] -comment "waiving single pair" -split
```



Preventing Multi-Pair Violation Waivers

As shown above, there can be scenarios where you do not want to waive an entire violation. Instead, you want to waive only those pairs that are safe. However, there might be some waiving filters applied that cause the entire merged violation to be waived rather than the required subset of CDC pairs of that merged violation. To handle such scenarios and prevent multi-pair violations from being waived in their entirety, you can use the following command:

```
set_cdc_no_waiver_for_multipair_violation (true | false)
```

By default, this configuration is set to `false`. When set to `true`, the tool prevents the waiver of any multi-pair CDC violation without splitting it. Also, if you try to waive such violations when the switch is set to `true`, the tool issues a warning stating the violation could not be waived. In the waivers table, it is reported as `unwaived`.

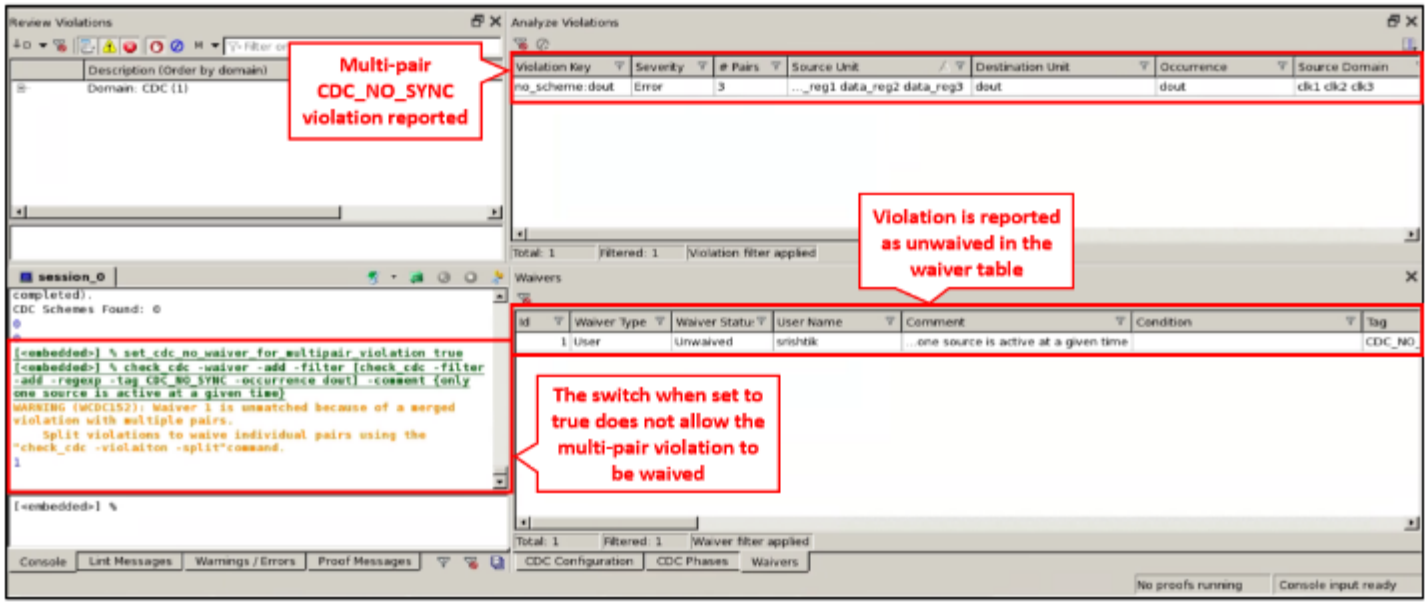
Example:

In the following scenario, a multi-pair `CDC_NO_SYNC` violation has been reported by the tool. To prevent the tool from waiving the violation, set the switch to true as follows:

```
% set_cdc_no_waiver_for_multipair_violation true
```

Setting the switch to `true` prevents you from waiving the entire violation, and thus, the tool issues a warning stating the violation could not be waived. The violations in such scenarios can be waived by using the command as suggested in the section above.

Also, since the violation does not get waived, it is reported as `unwaived` in the waivers table.



Waiving Structural Violations by Hierarchy

If there are blocks in the design (like third-party IPs or legacy IPs) where performing CDC analysis might not be very useful, you can choose to automatically waive all violations that are contained within that particular block. However, the tool continues to report all boundary violations, that is:

- Any violation originating outside the IP but ending inside
- Any violation originating inside the IP but ending outside

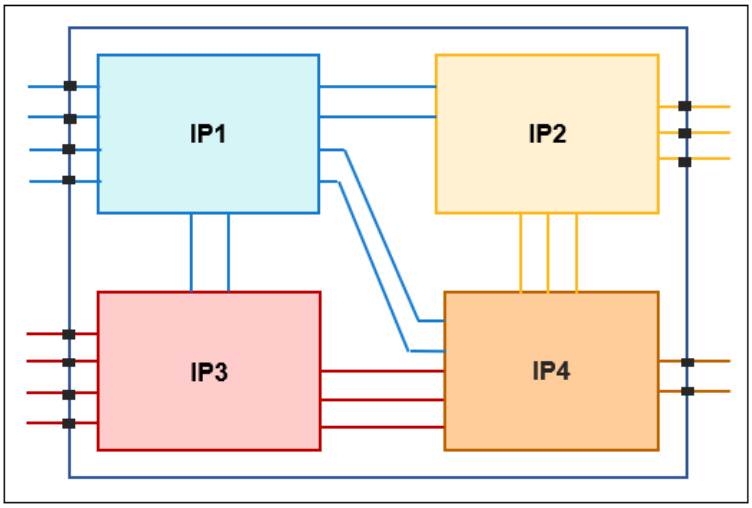
Use the following command to specify the modules or instances that the tool should skip when you issue the `check_cdc <-phase> -find` command:

```
% check_cdc -waiver -add -hierarchy <module or instance> [-filter filter_id]
```

The optional `-filter` switch can be used in scenarios where you want to waive only a selected set of tags for that particular module or instance rather than waiving all the violations reported for that particular module or instance.

Example:

For illustration, consider the following SoC, which has four IPs inside. Out of these four IPs, two are 3rd party IPs and one is a Legacy IP that has been signed-off multiple times.

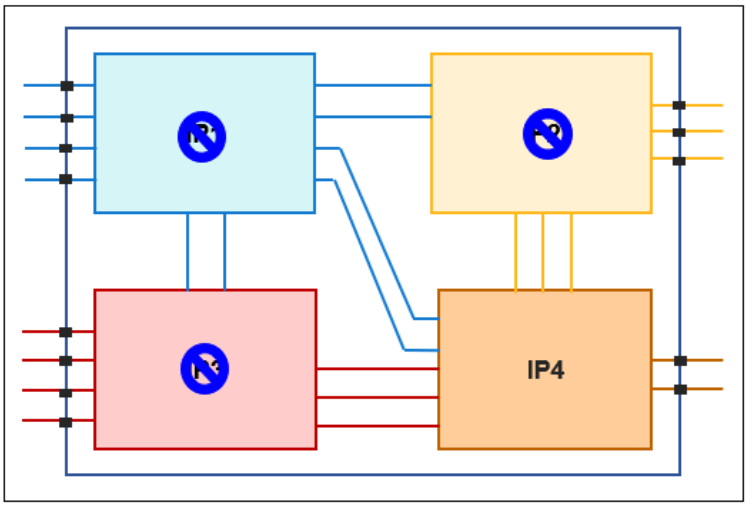


Carrying out CDC analysis on these three IPs would not be very beneficial. Thus, you can apply the waive hierarchy feature on these IPs and automatically waive all the violations contained completely inside the IPs. To use the waive hierarchy feature on these three IPs, use the following commands:

```
% check_cdc -waiver -add -hierarchy IP1

% check_cdc -waiver -add -hierarchy IP2

% check_cdc -waiver -add -hierarchy IP3
```



⚠

- You must run the `check_cdc -waiver -add -hierarchy` before generating any violations, that is, before running any `check_cdc <-phase> -find` command. If you run this command after running any phase command, all the violations associated with that phase are not waived.
- You must waive each hierarchy explicitly because this command does not accept a list of modules or instances.

Criteria CDC Uses for Waiving Violations under Waive Hierarchy

The criteria CDC uses for waiving the violations using the waive hierarchy feature are different for each of the violation categories reported in the tool. Different categories consider different CDC/RDC units or configuration sources depending on what is relevant for the violation under consideration.

ⓘ Use the following command to enable the waive hierarchy feature: `% check_ cdc -waiver -add -hierarchy IP`

For the CDC_CONFIGURATION category, the criteria is as follows:

Violation Type	Associated Tags	Items Considered

Configuration Violations	`CKS_IS_CONV`
	`CLK_IS_CONV`
	`CLK_GT_ASYN`
		Configuration Sources and **Convergence Gate**
		If the configuration source and the convergence gate belong to the same IP, the violation is waived under waive hierarchy; otherwise, it is reported at the SoC level.

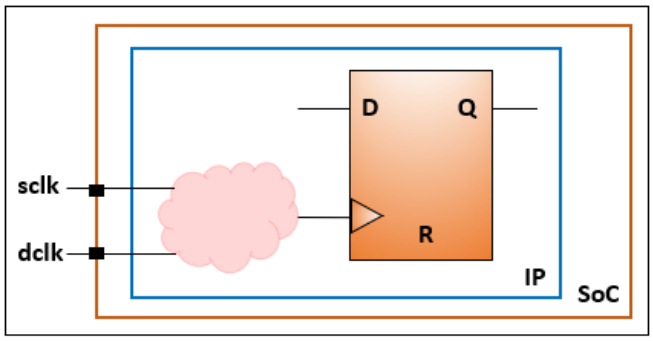
Configuration Violations	`CKS_NO_CONS`
	`CLK_IS_CNST`
	`CLK_NO_DECL`
	`CND_NO_CONS`
	`LTC_NO_INIT`
	`PRT_IS_UNCK`
		Configuration Sources
		If the configuration source is outside the IP, the violation is reported at the SoC level; otherwise, the violation is waived under waive hierarchy.

Example 1:

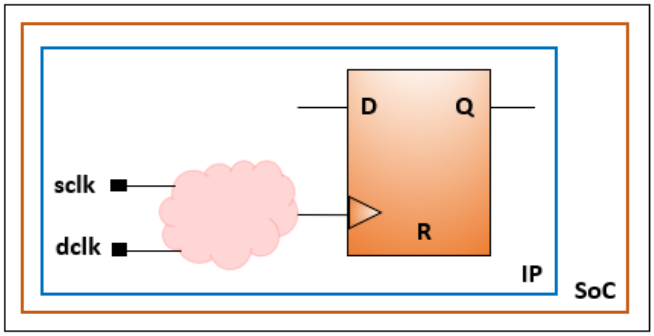
In the design below, the clocks (`sclk` and `dclk`) are defined at the SoC but the convergence happens inside the IP. Since both the configuration sources (clocks) and convergence gate belong to a different hierarchy, the tool reports a `CLK_IS_CONV` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when both the clocks (`sclk` and `dclk`) are declared inside the IP and the convergence also happens inside the same IP, the `CLK_IS_CONV` violation is waived under waive hierarchy by the tool.

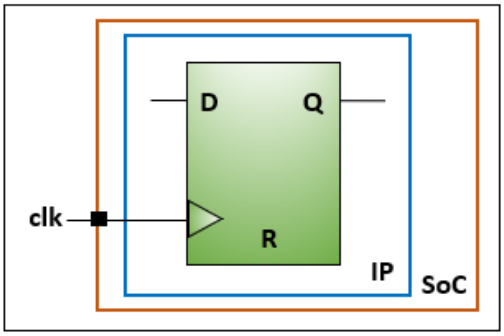


Example 2:

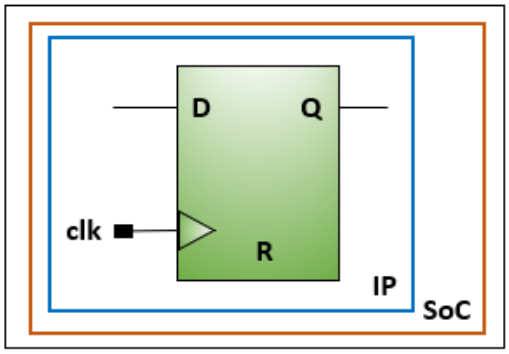
In the design below, the clock (`clk`) is an SoC-level clock but has not been declared. Since the configuration source (clock) is at the SoC level, the tool reports a `CLK_NO_DECL` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when the undeclared clock is contained completely inside the IP, the tool waives `CLK_NO_DECL` violation under waive hierarchy for this IP.



For the CDC_SYNCHRONIZATION category, the criteria is as follows:

Violation Type	Associated Tags	Items Considered
Pair Violations	CDC_NO_SYNC CDC_PR_FOUT CDC_PR_INAC CDC_PR_LOGC CDC_PR_LTCH	**Source Flops/Ports** and the **Destination Flops/Ports** If either the source flops/ports or the destination flops/ports exist outside the IP, the violation is reported at the SoC level; otherwise, the violation is waived under waive hierarchy.
Scheme Violations	SYN_DF_CKPH SYN_DF_FOUT SYN_DF_RDND SYN_DF_SLGC SYN_GP_ASIP SYN_GP_ASOP SYN_SE_INVL	**Scheme Flops**. The violations are waived by the tool if all the scheme flops exist inside the IP; otherwise, the violations are reported at the SoC level.

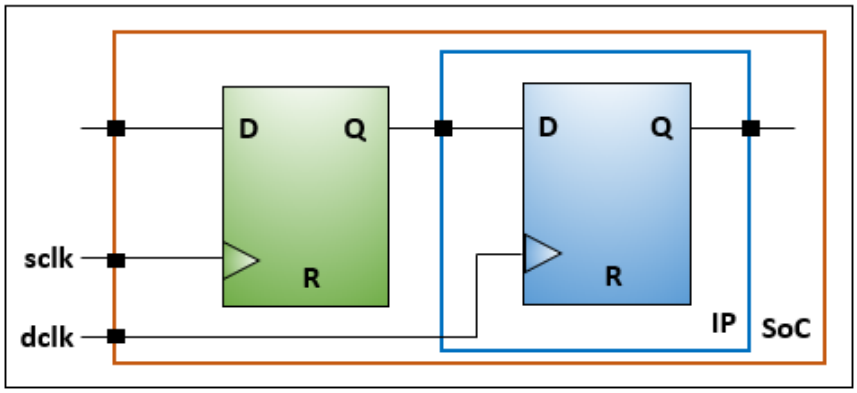
Example 1:

In the design below, the source flop exists at the SoC level but the destination flop exists inside the IP . Since the source and destination flops belong to different hierarchies, the tool reports a CDC_NO_SYNC violation at the SoC level

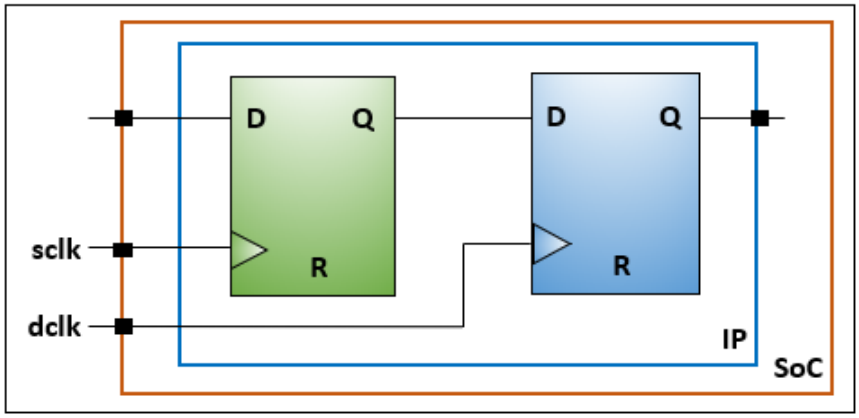
even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when both the source and destination flops belong to the same IP, the CDC_NO_SYNC violation is waived under waive hierarchy by the tool.

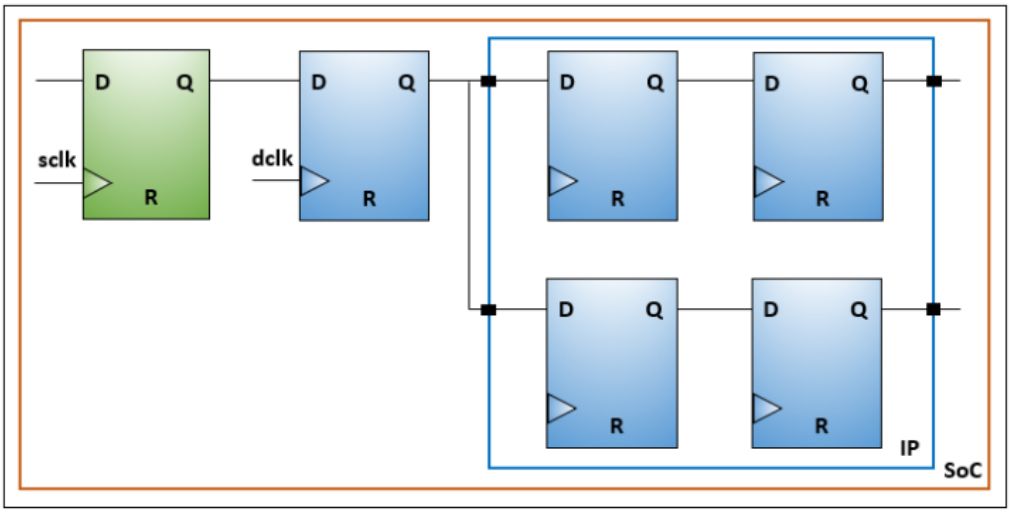


Example 2:

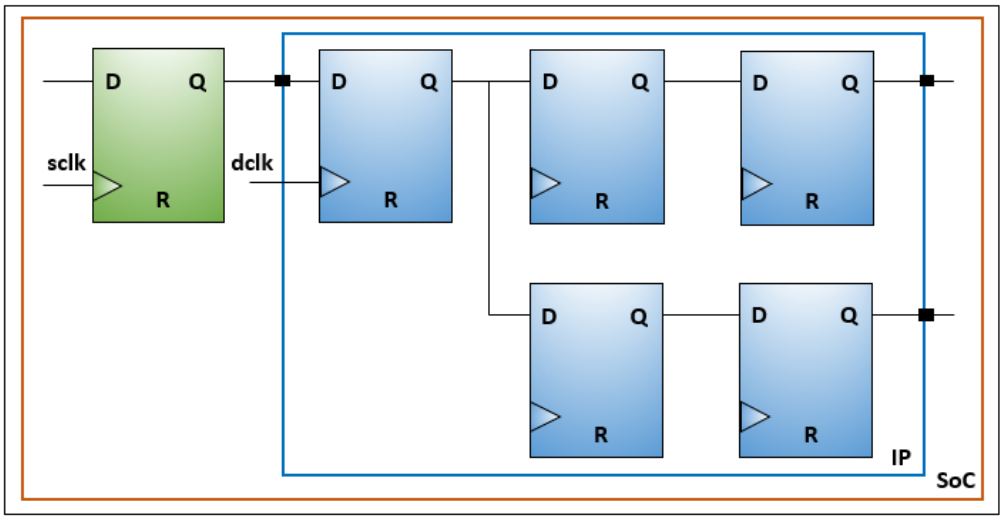
In the design below, some of the scheme flops exist at the SoC level and some exist inside the IP . Since all scheme flops belong to different hierarchies, the tool reports an SYN_DF_FOUT violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when all the scheme flops are contained completely inside the IP, the tool waives the `SYN_DF_FOUT` violation under waive hierarchy for this IP.



For the CDC_CONVERGENCE category, the criteria is as follows:

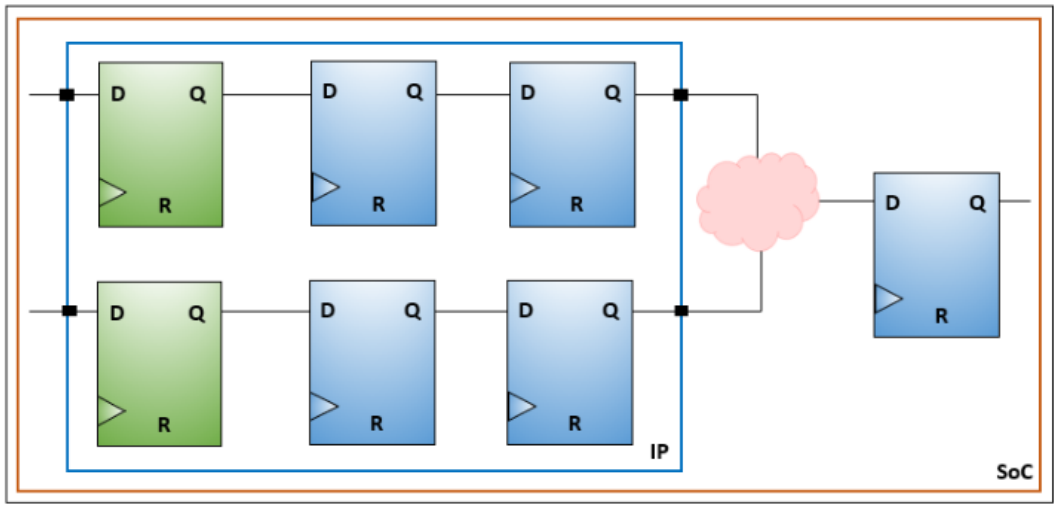
Violation Type	Associated Tags	Items Considered
Convergence Violations	`CNV_ST_CONV` `CNV_ST_GLCH`	**All Pairs Flops/Ports** and **Convergence Flops/Ports**. No violation is reported at the SoC level if all the pair and the convergence flops/ports exist inside the IP. However, if any of the flops/ports exist outside the IP, a violation is reported at the SoC level.

Example:

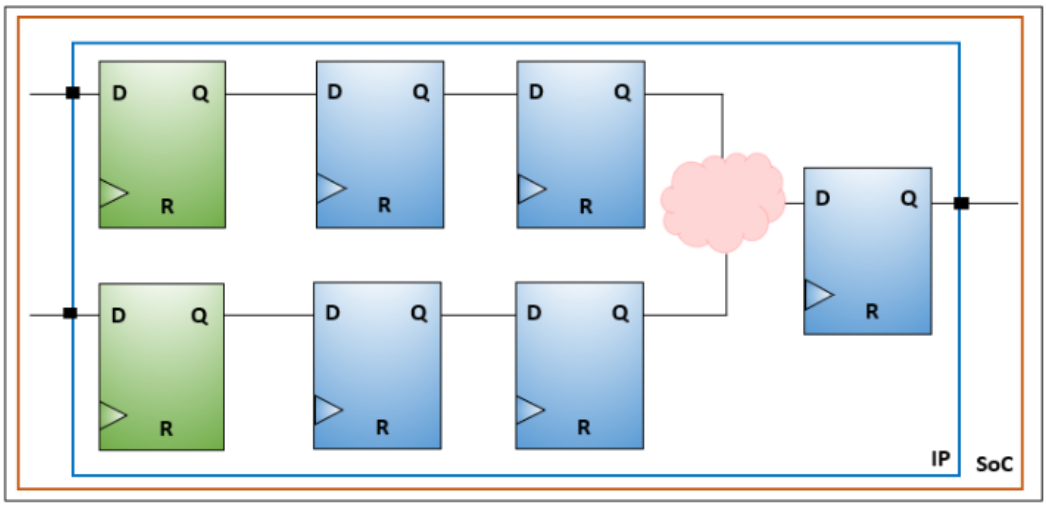
In the design below, the source flops and synchronizers exist inside the IP, but the convergence gate and the destination flop exist at the SoC level. As pair flops and convergence flops and gates belong to different hierarchies, the tool reports a `CNV_ST_CONV` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when all the flops (that is, the pair and the destination flops along with the convergence gate) are contained completely inside the IP, the tool waives the CNV_ST_CONV violation under waive hierarchy for this IP.



For the RST_CONFIGURATION category, the criteria is as follows:

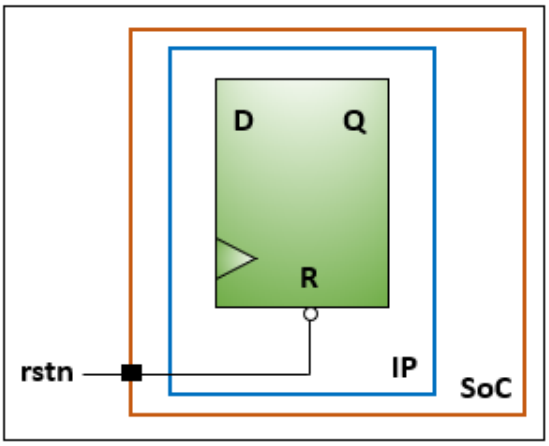
Violation Type	Associated Tags	Items Considered
Reset Configuration Violations	RST_NO_DECL RST_RS_FSAR	**Configuration Sources** and **Destination Flops/Ports** If the configuration source and the destination flops/ports exist inside the IP, the violation is waived under waive hierarchy; otherwise, it is reported at the SoC level.
Reset Configuration Violations	RST_RS_CONV	**Configuration Sources** and **Convergence Gate** If the configuration source and the convergence gate exist inside the IP, the violation is waived under waive hierarchy; otherwise, it is reported at the SoC level.

Example 1:

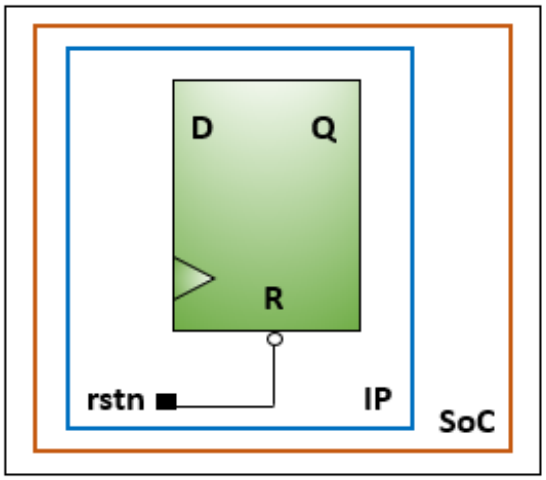
In the design below, the reset (rst) is an SoC-level reset but has not been declared. Since the configuration source (reset) is at the SoC level, the tool reports an RST_NO_DECL violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when the undeclared reset is contained completely inside the IP, the tool waives the `RST_NO_DECL` violation under waive hierarchy for this IP.

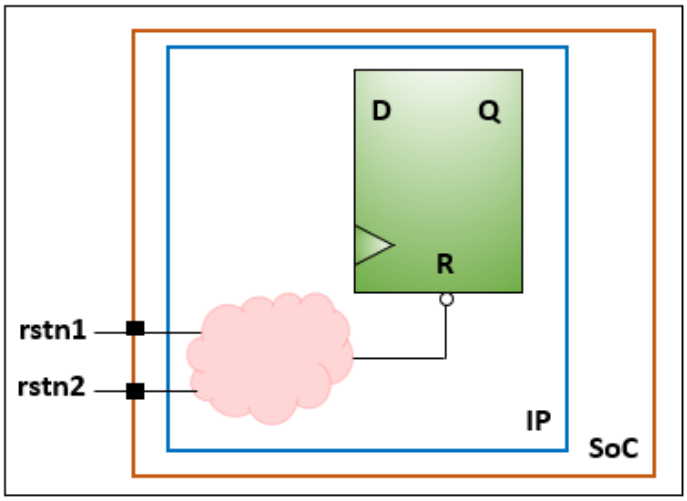


Example 2:

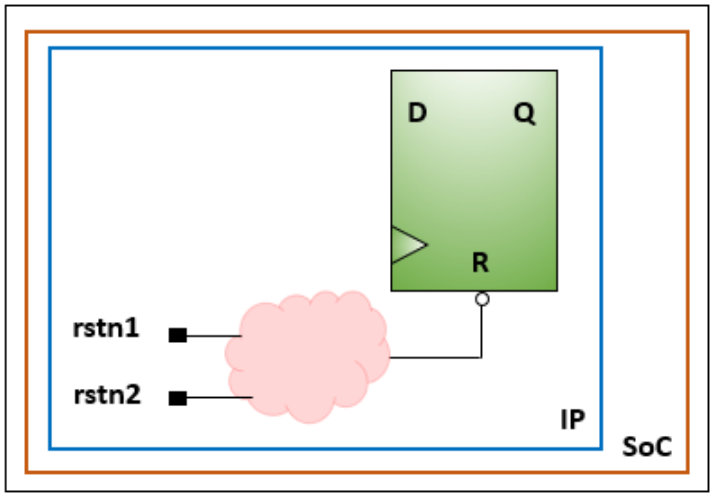
In the design below, the resets (`rstn1` and rstn2) are defined at the SoC but the convergence happens inside the IP. Since both the configuration sources (resets) and convergence gate belong to different hierarchies, the tool reports an `RST_RS_CONV` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when both the resets (rstn1 and rstn2) are declared inside the IP and the convergence also happens inside the same IP, the `RST_RS_CONV` violation is waived under waive hierarchy.



For the RST_SYNCHRONIZATION category, the criteria is as follows:

Violation Type	Associated Tags	Items Considered
Reset Path Violations	`RST_NO_SYNC` `RST_RS_RIDP`	**Reset Path Source Flops/Ports** and **Reset Path Destination Flops/Ports**. If either the reset path source flops/ports or the reset path destination flop/ports exist outside the IP, the violation is reported at the SoC level; otherwise, the violation is waived under waive hierarchy.
Reset Pair Violations	`RDC_RS_DFRS` `RST_RS_INAC`	**Source Flop/Port** and the **Destination Flop/Port**. If either the source flop/port or the destination flop/port exist outside the IP, the violation is reported at the SoC level; otherwise, the violation is waived under waive hierarchy.

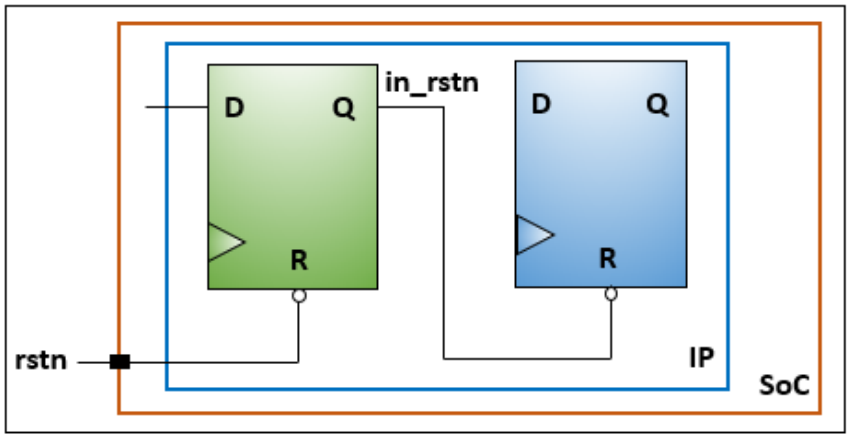
Reset Scheme Violations	`SYN_RS_CKPH`
	`SYN_RS_SMDR`
	`SYN_RS_SLGC`
	`SYN_RS_RDND`
	`SYN_RS_FOUT`
		Scheme Flops.
		Violations are waived if all the scheme flops exist inside the IP; otherwise, violations are reported at the SoC level.

Example 1:

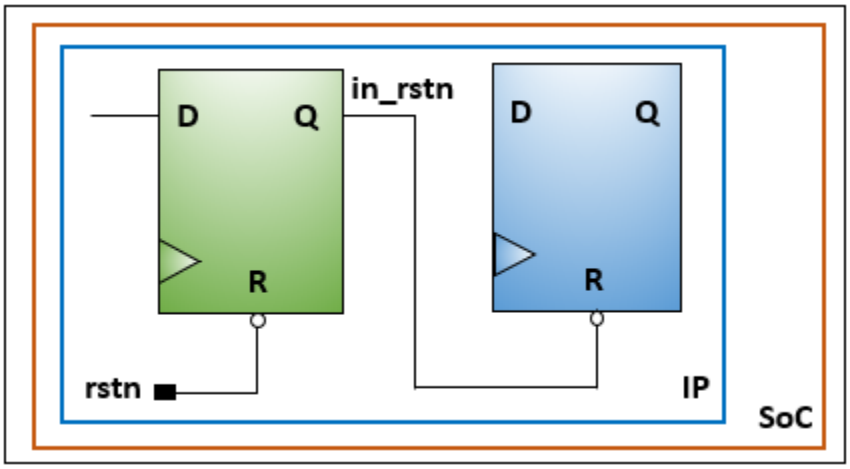
In the design below, the reset signal (`rstn`) declared at SoC is driving a flop inside the IP. As the declared reset source and the destination flop belong to different hierarchies, the tool reports an `RST_NO_SYNC` on the `rstn` signal at the SoC level. Since the reset signal (`in_rstn`) is inside the IP and is driving the flop inside the IP. The tool waives `RST_NO_SYNC` on `in_rstn` under waive hierarchy.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, when the declared reset signal (`rstn`) is declared inside the IP and the destination flop that it is driving also exists inside the same IP, the tool waives the `RST_NO_SYNC` violation reported on `rstn` under waive hierarchy. Similarly, as reset signal (`in_rstn`) is inside the IP and the destination flop also exists in the same IP, the `RST_NO_SYNC` violation reported on `in_rstn` is waived under waive hierarchy.

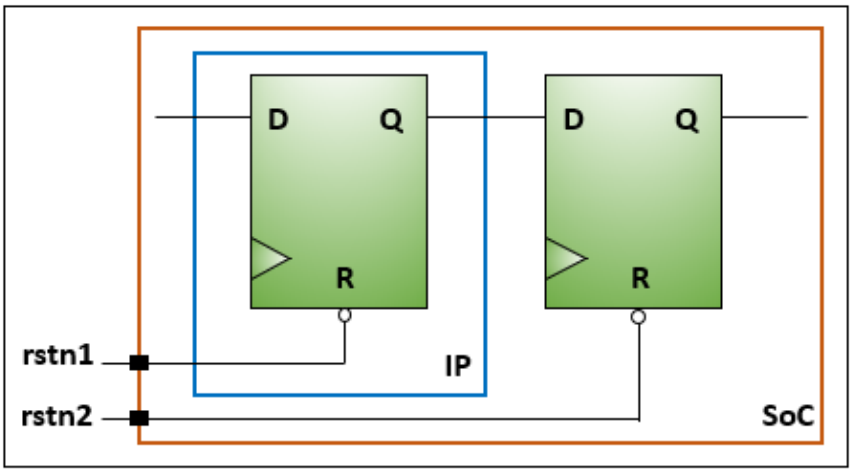


Example 2:

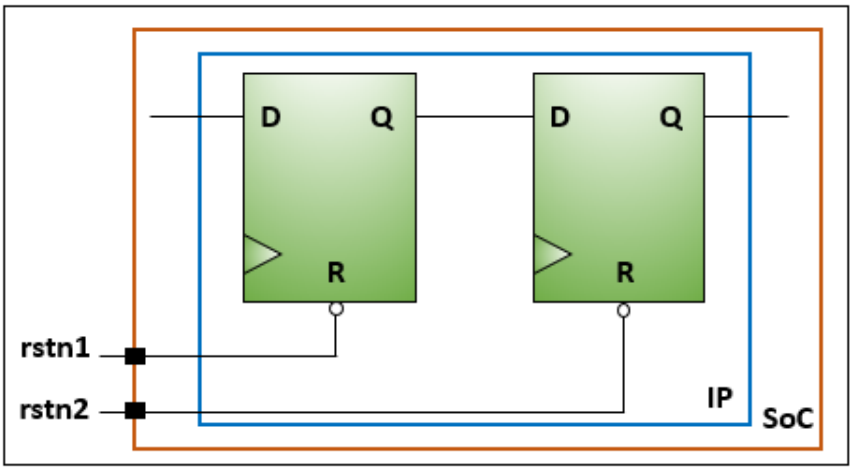
In the design below, the source flop exists inside the IP whereas the destination flop exist at the SoC level. Since the source and the destination flop of the reset pair belong to different hierarchies, the tool reports an `RDC_RS_DFRS` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



However, if both the source and the destination flop exist inside the same IP, then the `RDC_RS_DFRS` violation is waived under waiver hierarchy.

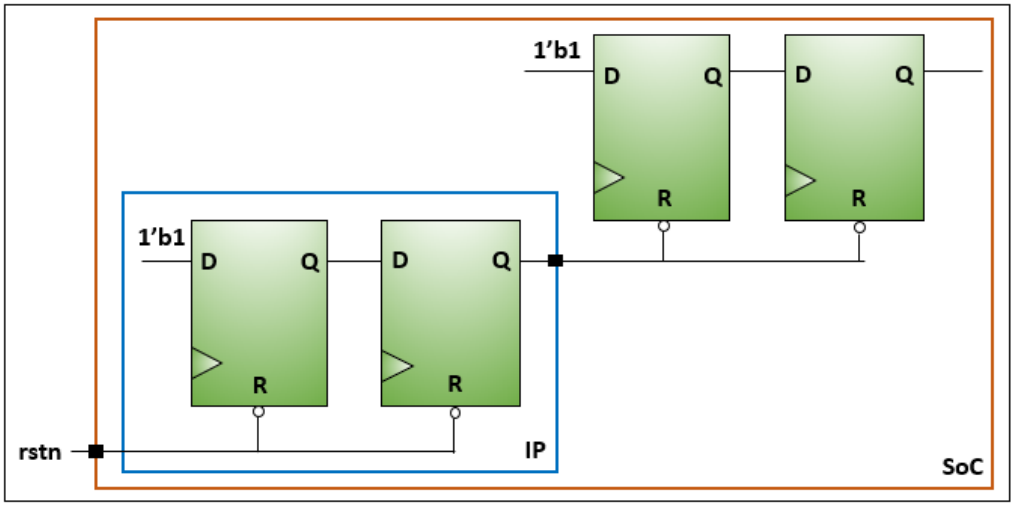


Example 3:

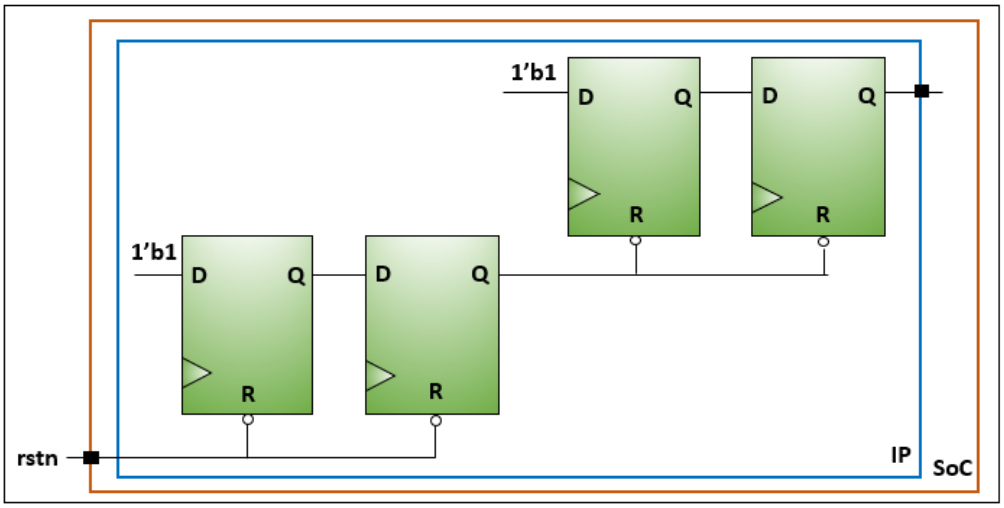
In the design below, there are two reset synchronizers connected back to back, with one synchronizer at the IP level and other at the SoC level. As these synchronizers belong to different hierarchies, the tool reports a `SYN_RS_RDND` violation at the SoC level even if you have used the waive hierarchy feature.

To apply waive hierarchy on the IP, use the following command:

```
% check_ cdc -waiver -add -hierarchy IP
```



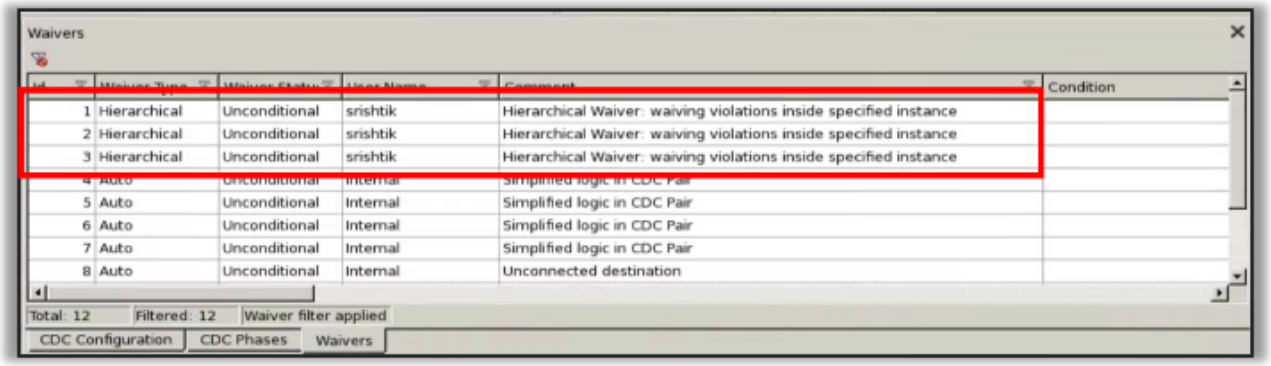
However, if both synchronizers exist inside the same IP, the `SYN_RS_RDND` is waived under waive hierarchy.



Reviewing Hierarchical Waivers

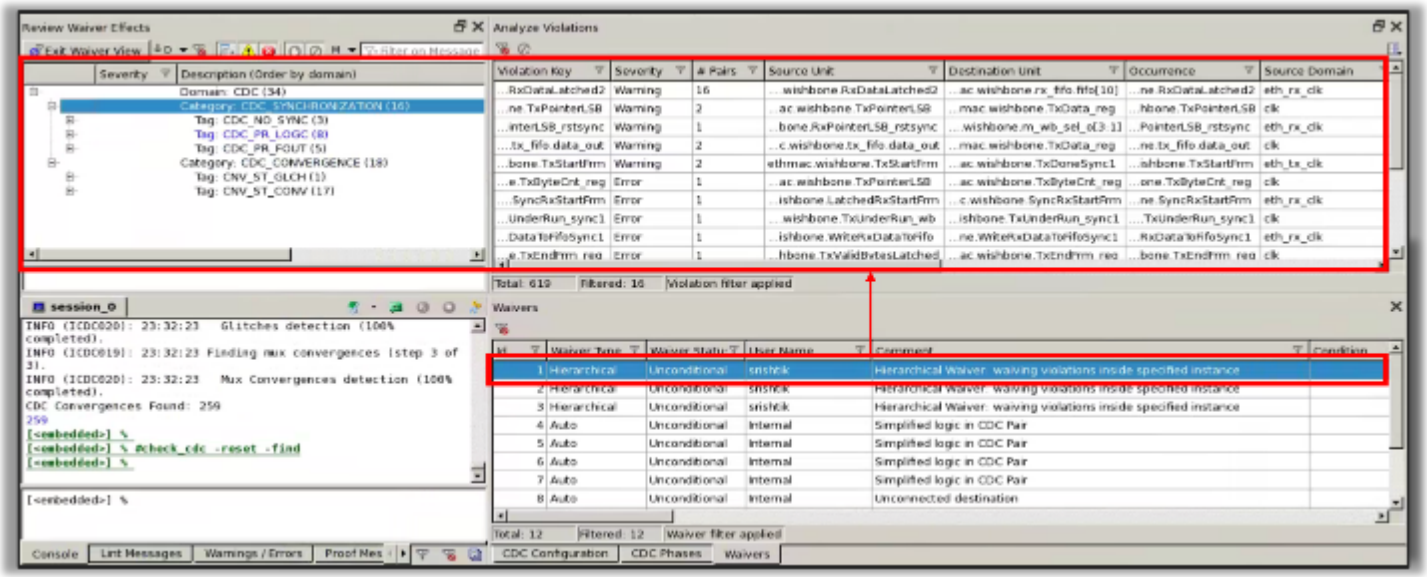
For ease of identification, all violations waived with the `check_cdc -waiver -add -hierarchy` command include the prefix Hierarchical Waiver in the waive comment,

Also, these waivers are reported as *Hierarchical* in the *Waiver Type* column of the *Waivers* table.



All violations associated with a particular module/instance are grouped under a single waiver. To view all waived violations associated with a particular block, select the waiver from the waiver table.

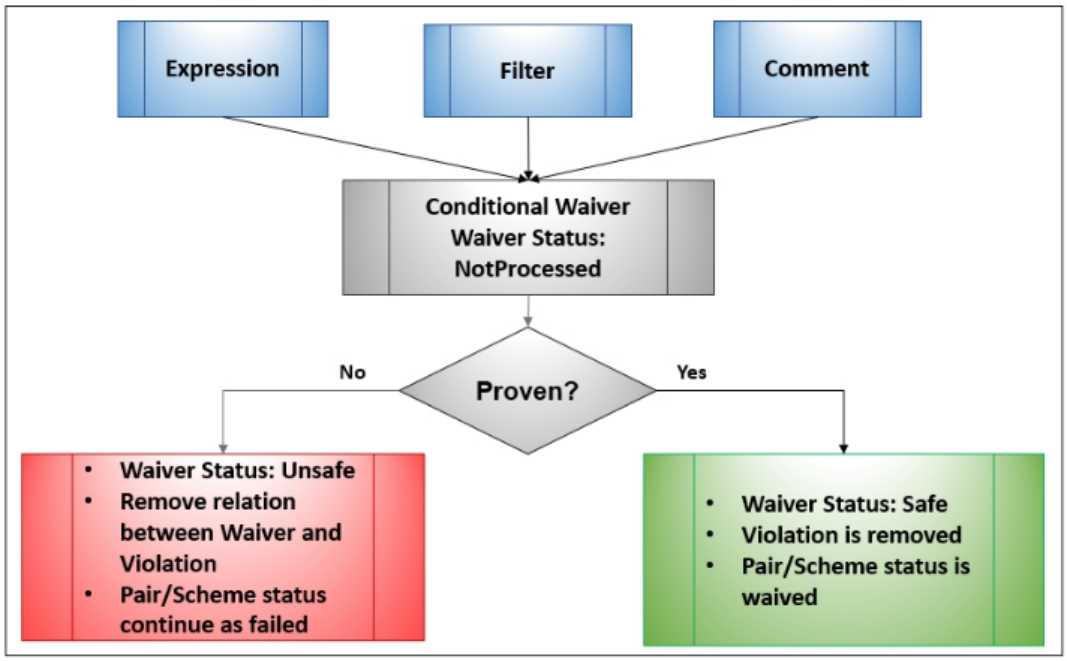
This populates the Review Violation Tree with the associated waived violations.



Conditional Waivers

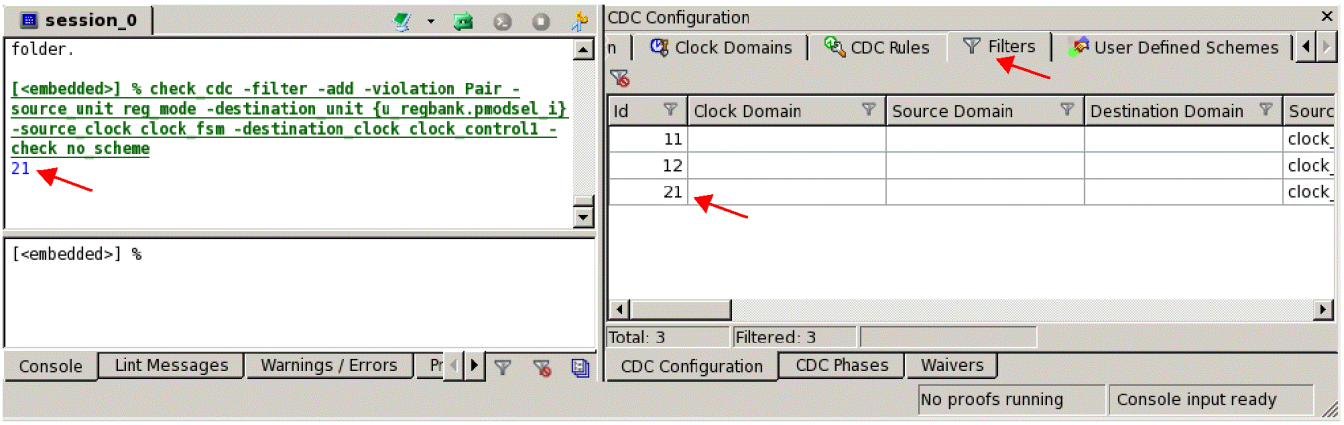
CDC provides the option to specify an SVA expression as a waiver condition and then validate the condition with a formal proof. Once proven, the waiver is considered safe; otherwise, it is considered as unsafe.

The following flowchart shows the process by which the conditional waivers are defined and processed.

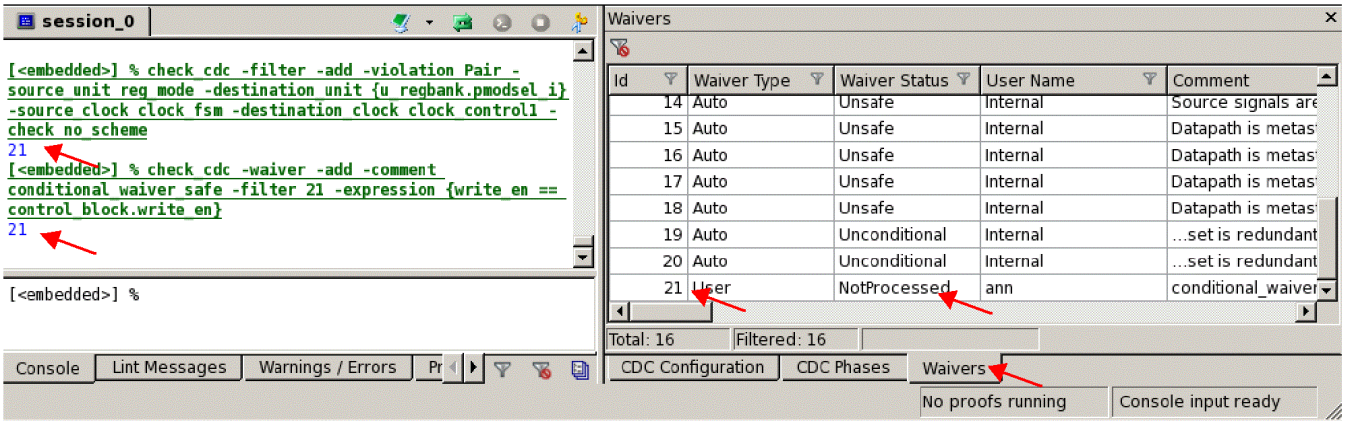


See the following procedure to better understand the steps defined in the flowchart.

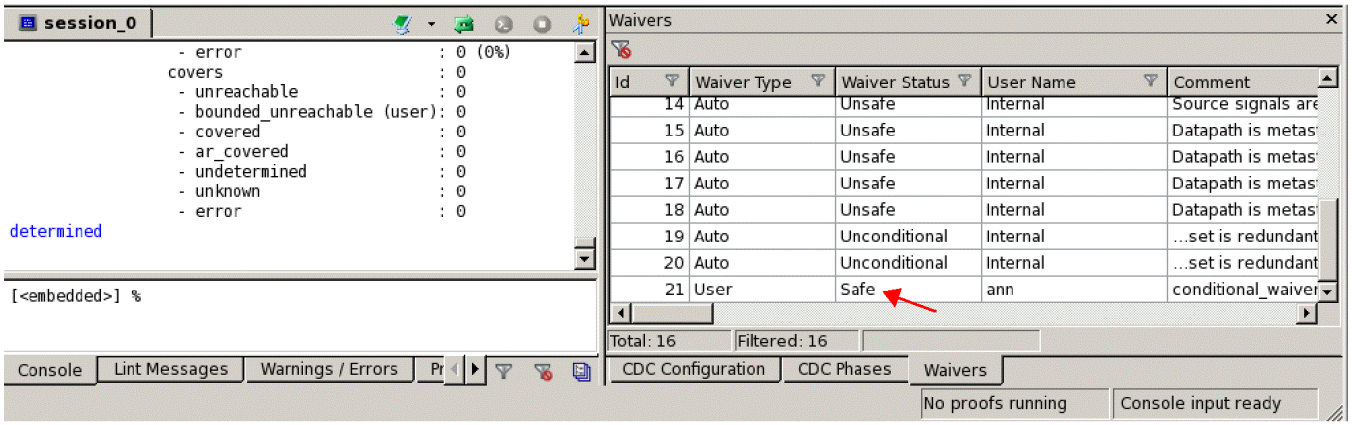
1. Add the filter on which the waiver will be based (see `check_cdc -filter -add...` and the following figure).



2. Use the following command syntax to add the conditional waiver:
   `check_cdc -waiver -add -comment <comment> -filter <filter_id> (-expression <sva_expression> |-expression_list <exp_list>)`
   The tool adds the waiver to the *Waivers* tab as `NotProcessed` since you have not yet run the proof.



3. Run `check_cdc -waiver -prove`.
   The `NotProcessed` status becomes `Safe` if the condition is proven and the waived violation is removed from the violations table.



Selectively Proving Conditional Waivers

The above procedure detailed how to prove all conditional waivers generated for a particular design. However, in some scenarios you might want to selectively prove a set of waivers. You can do this in through the command line or from the GUI.

Through Command Line

Use the following command to prove a selective set of conditional waivers:

```
% check_cdc -waiver -prove -id {id_list}
```

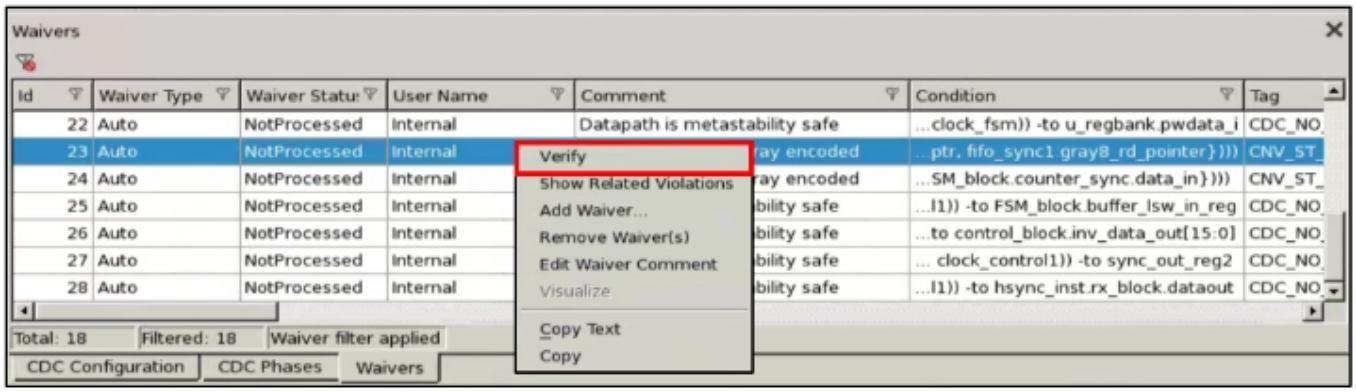
- Provide a non-empty list of waivers IDs.
- The tool issues a warning message for each of the non-conditional waivers specified in the

list, and ignores them.

Through GUI

To prove a conditional waiver with `NotProcessed` status from the GUI, do the following:

1. Right-click on a `NotProcessed` waiver and select the *Verify* option.
   If the condition is proven, the status changes to `Safe`; otherwise, it becomes `Unsafe`.



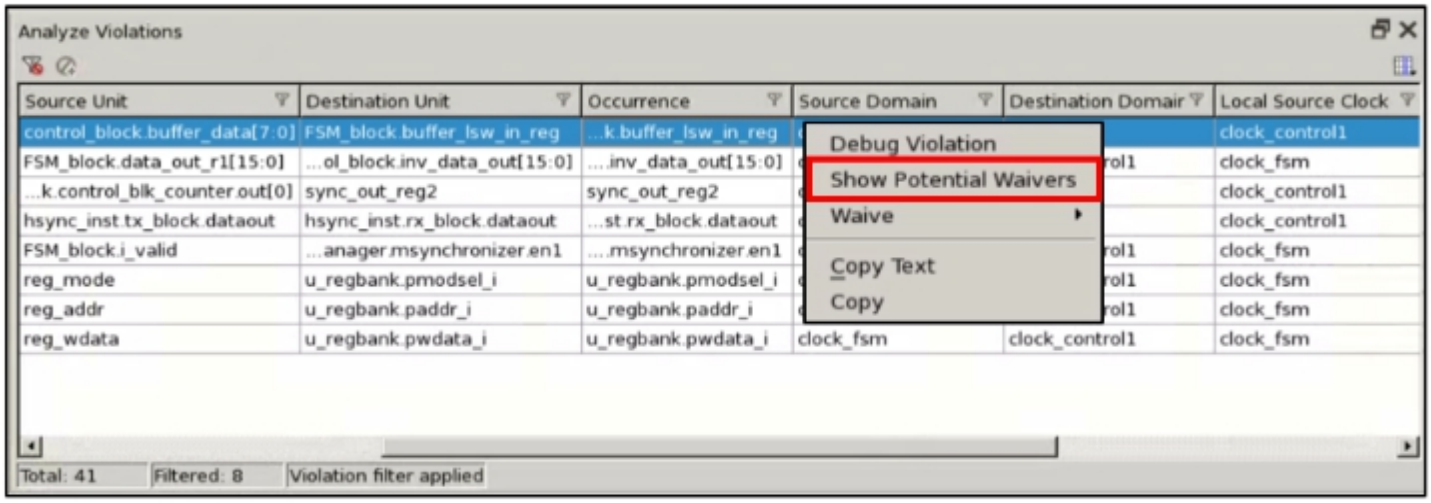
2. To see all the related violations of a conditional waiver, right-click on the waiver and choose *Show Related Violations*.
   This reveals related violations that have not yet been waived, that is, the conditional waiver is still in the `NotProcessed` state.

Viewing Potential Waivers

Potential waivers associated with violations are conditional waivers that are still not processed, and if proven, can be waived. To view potential waivers, do the following:

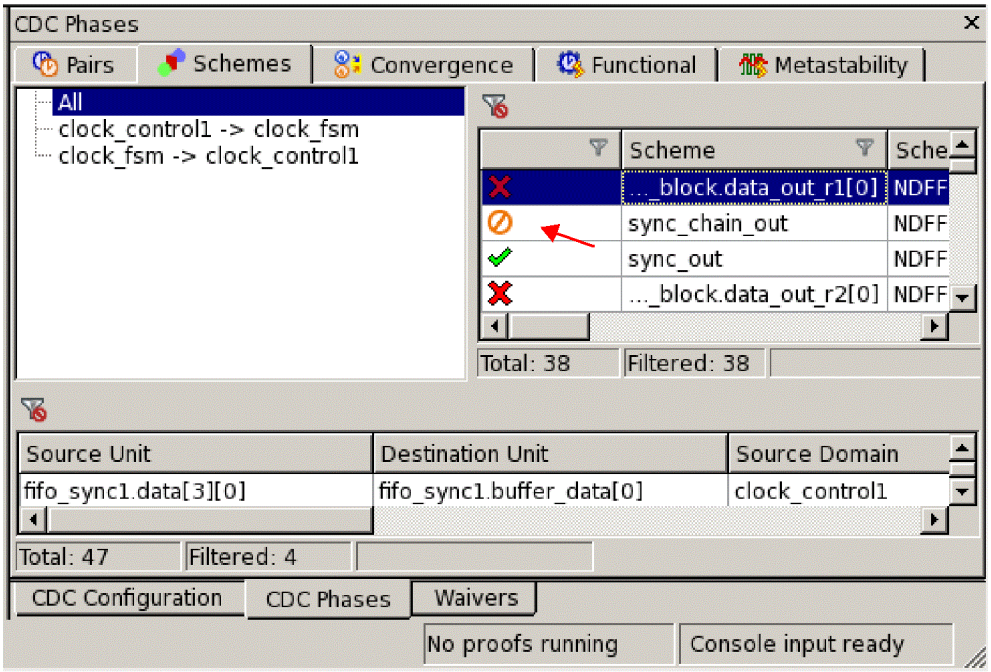
- Right-click on a waiver from the Review Violation tree or the Analyze Violations table and select *Show Potential Waivers* from the context menu.

⚠ The *Show Potential Waivers* option is enabled only if the selected entry is not a waived violation and there are potential waivers for it.



Automatic/Safe Waiver Flow

The tool automatically waives structural violations that are a result of logic simplification. An orange "not" circle indicates violations that have been automatically waived.



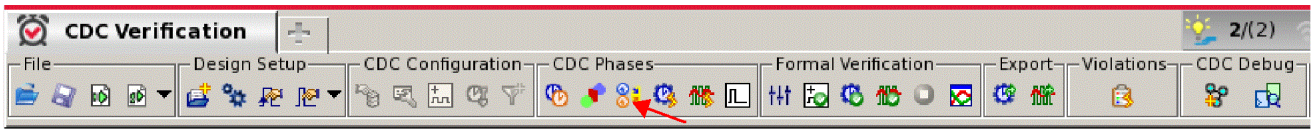
To determine whether automatic waivers are due to static, constant, or mutually toggle exclusive signals, you can run `check_cdc -waiver -why <waiver_id>`. This command lists the signals that might be causing an automatic waiver, or returns an empty list if no related signals are identified.

Additionally, the tool can automatically generate gray encoding checks for converging signals and

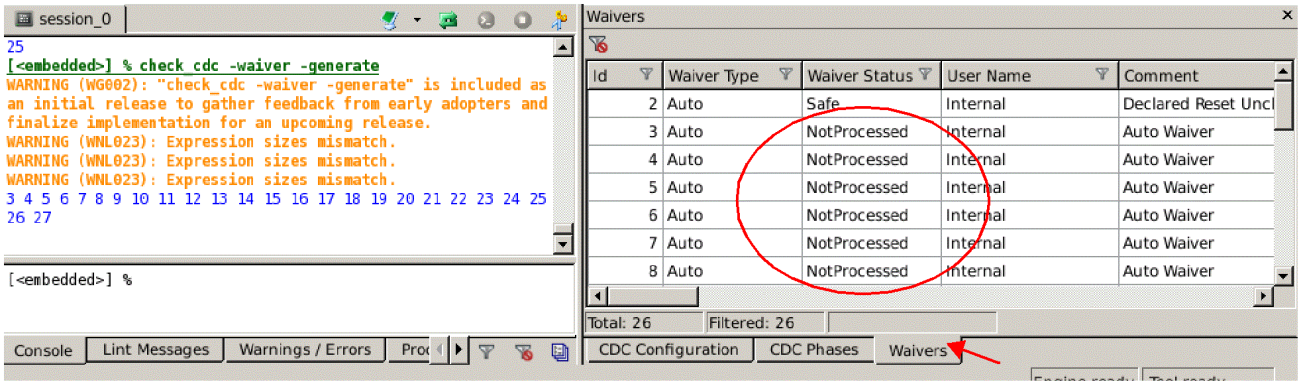
validate whether these are genuine violations. If proven, these violations are automatically waived.

To generate and verify automatic waivers, do the following:

1. Click the *Find Convergence* button on the *CDC Phases* toolbar.



2. Run `check_cdc -waiver -generate`.
   The tool adds the automatic waivers to the *Waivers* tab as `NotProcessed` since you have not yet run the proof.



3. Run `check_cdc -waiver -prove`.
   The `NotProcessed` status becomes `Safe` if the condition is proven and `Unsafe` if the tool finds a counterexample.

Exporting and Importing Waivers

To reuse waivers in a different CDC session, export the waivers with the following command:

```
% check_cdc -waiver -export -file <file_name> [(-user | -imported | -hierarchical)]
                                              [(-safe | -unsafe | -unmatched | -
unconditional | -unwaived | -not_processed) ] -do_not_apply
```

- Use `-user` to export only waivers added in the current session.
- Use `-imported` to export only waivers imported from scripts in the current session.
- Use `-hierarchical` tp export only hierarchical waivers.
- Use `-safe, -unsafe, -unmatched, -unconditional, -unwaived, -not_processed` to export

waivers of the specified statuses only.

- Use `-do_not_apply` to prevent applying the waiver in the current session.

⚠ The `-do_not_apply` option is also available in the *Add Waiver Comment* and *Open Add Waiver* dialog boxes. First select the *Export to file* check box, and then specify a file and select the *Do not apply waiver* check box.

By default, CDC exports all waivers.

Import the file in a subsequent run as a regular Tcl file using the `include` command:

```
% include <file_name>
```

⚠ You might encounter duplication or unmatched waiver issues while sourcing waiver files from one run to another. To ease the process of locating the offending file, CDC provides the file name and the line number where the problem occurs. All such issues are reported as warnings in the console.

Generating Reports

This section presents information on generating reports and includes the following topics:

- Generating Reports from the GUI
- Generating Reports Through Command Line
- Scripting Interface

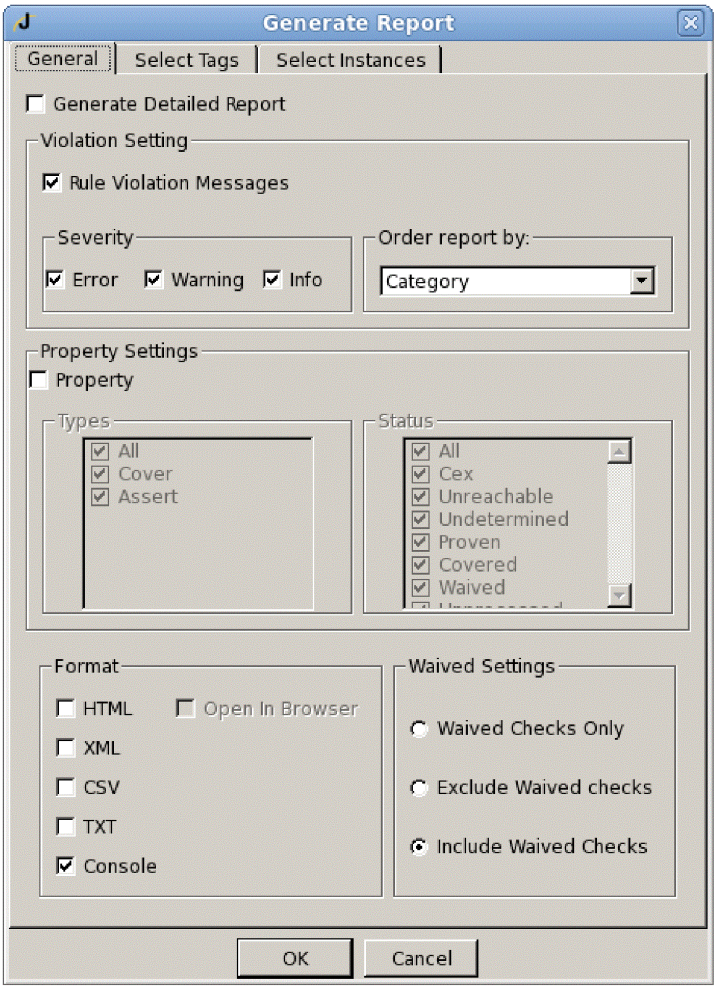
ⓘ Access to CDC HTML reports requires Firefox® 17 or above.

Generating Reports from the GUI

To access the *Generate Report* dialog box, do the following:

1. Click the *Generate Report* button on the CDC App *Violations* toolbar.

The *Generate Report* dialog box appears with the *General* tab active.



2. Click *Generate Detailed Report* for a detailed report. By default, the tool generates a summary report.
3. Under *Violation Settings* do the following:
    - Deselect *Rule Violation Messages* if you do not want violation messages included in the report. All reports include violation messages by default.
    - Select the check boxes for all violation severities you want reported. The default is all.
    - Use the drop-down menu under *Order report by* to choose how you want the report organized. Options include *Category*, *Filename*, *Severity*, *Label*, and *Instance*.

4. Under *Property Settings*, do the following:
   - Select the *Property* check box to access the options in this group and generate a properties report.
   - To focus on properties of a specific type, select one or more *Types* check boxes. The default is all types.
   - To focus on properties of a specific status, select one or more *Status* check boxes. The default is all statuses.
5. Under *Format*, select one or more of the following formats for the report:
   - *HTML*

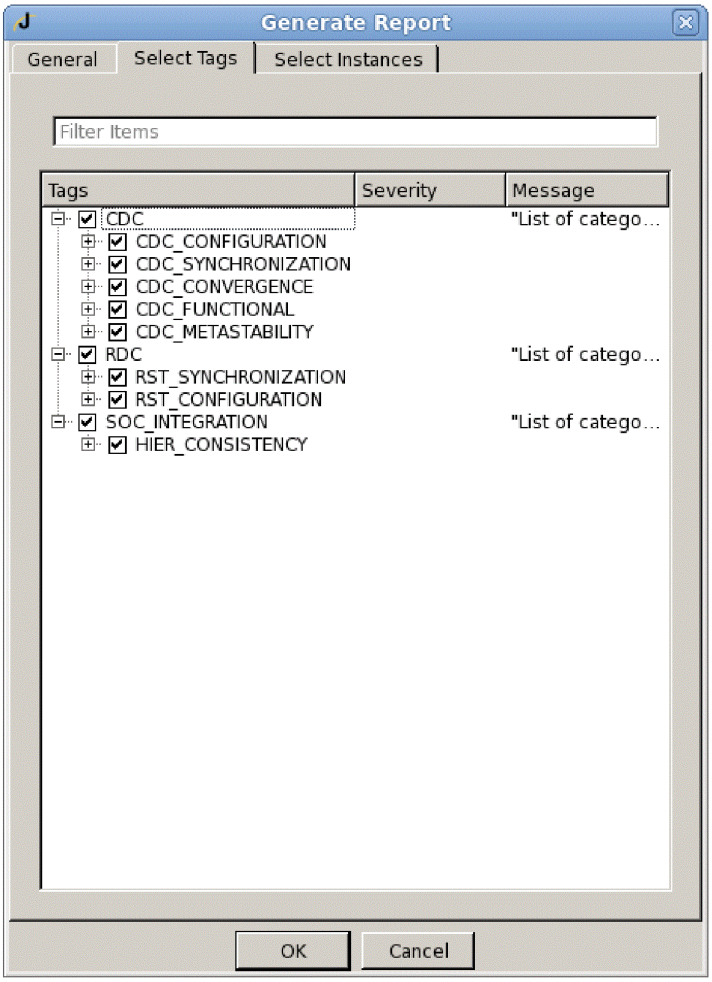
     ⚠ If you choose this option, the *Open in Browser* option is automatically selected. Deselect *Open in Browser* if you prefer not to immediately open the report in a browser.

   - *XML*
   - CSV
   - *Text*
   - *Console*

     ⚠ The *Console* option generates a report to the console only and is the default. This is the equivalent of issuing `check_cdc -report` without `-html`, `-xml`, or `-csv`.

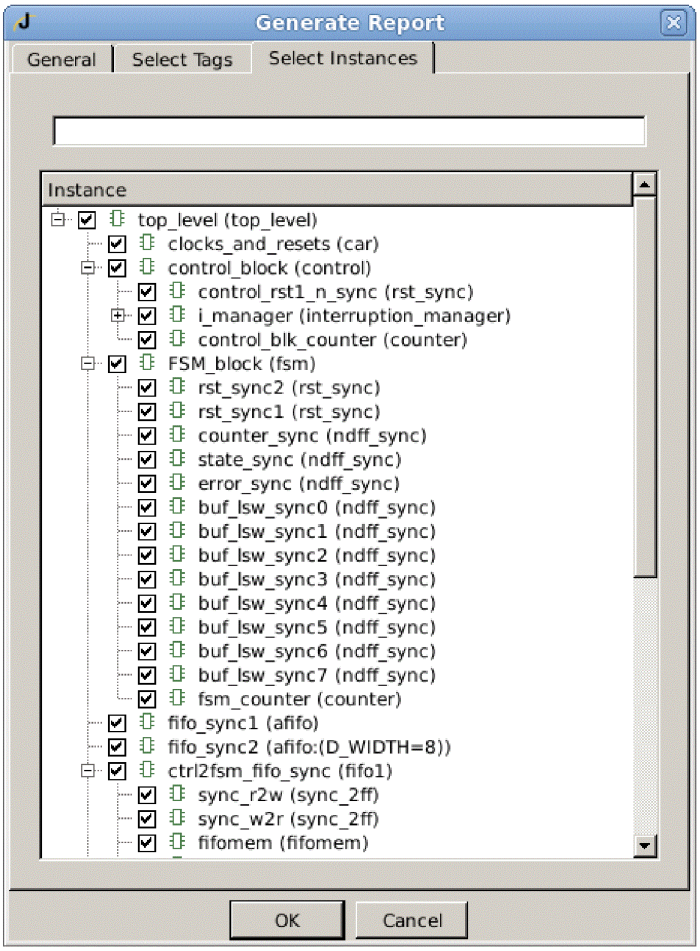
6. Under *Waived Settings*, do the following:
   - Select the *Waived checks only* check box if you want to generate a report of waived checks only.
   - Select the *Exclude waived checks* check box if you want to generate a report that excludes waivers.
   - Select the *Include waived checks* check box if you want to generate a report that includes waivers. This option is the default.
7. Click the *Select Tags* tab, and do any of the following:
   - Use the *Filter items* field to filter the results list.
   - Use the context menu (right-click) to expand or collapse all or to enable or disable all.

- Click on the plus sign (+) to expand a domain or category or click on the minus sign (-) to collapse a domain or category.
- Select the check boxes for those domains, categories, and/or tags that you want included in the report.
- Deselect the check boxes for those domains, categories, and/or tags that you do not want included in the report.



8. Click the *Select Instances* tab, and do any of the following:
   - Use the *Filter items* field to filter the results list.
   - Use the context menu (right-click) to expand or collapse all, enable or disable all, or enable or disable a the selected hierarchy.
   - Click on the plus sign (+) to expand an instance or click on the minus sign (-) to collapse an instance.

- Select the check boxes for those instances that you want included in the report.
- Deselect the check boxes for those instances that you do not want included in the report.



9. When you are ready to generate the report, click *OK* from any tab.
   If you have selected any format other than *Console*, a *Save Report* dialog box opens.
10. Complete the name field and click *Save*.

Generating Reports from Command Line

Reports can be generated using the following command:

```
check_cdc -report [[( -html [-launch_html_browser] [-source_path_mapping paths_mapping]
                       -txt |-xml |-csv] ]
                     [-file file_name [-force]
                     [-domain domain_list]
                     [-category category_list]
                     [-tag tag_list]
                     [-instance instance_list]
                     [-regexp]
                     [-waived (none | only [-hierarchical] [-imported] [-user])]
                     [-grouping (none | basic)]
                     [-property
                         [-status (  cex | unreachable | proven
                                   | covered | undetermined | unprocessed)]
                         [-type (assert | cover)]
                         [-task task] ]
                     [-violation
                         [-severity (error | warning | info | fatal)]
                         [-order (  category | tag | filename | instance(module) | label
                                  | severity)] ]
                     [-waiver
                         [-waiver_status ( safe | unsafe | unconditional | not_processed |
unmatched | unwaived)]
                         [-waiver_type (user | imported | hierarchical | auto)] ]
                     [-detailed]
```

Generating Violation Summary

If you use the following command, the tool generates a violation summary report in the console:

```
% check_cdc -report
```

If you specify the file name using the `-file` switch in the above command, then a file containing the violation summary is generated.

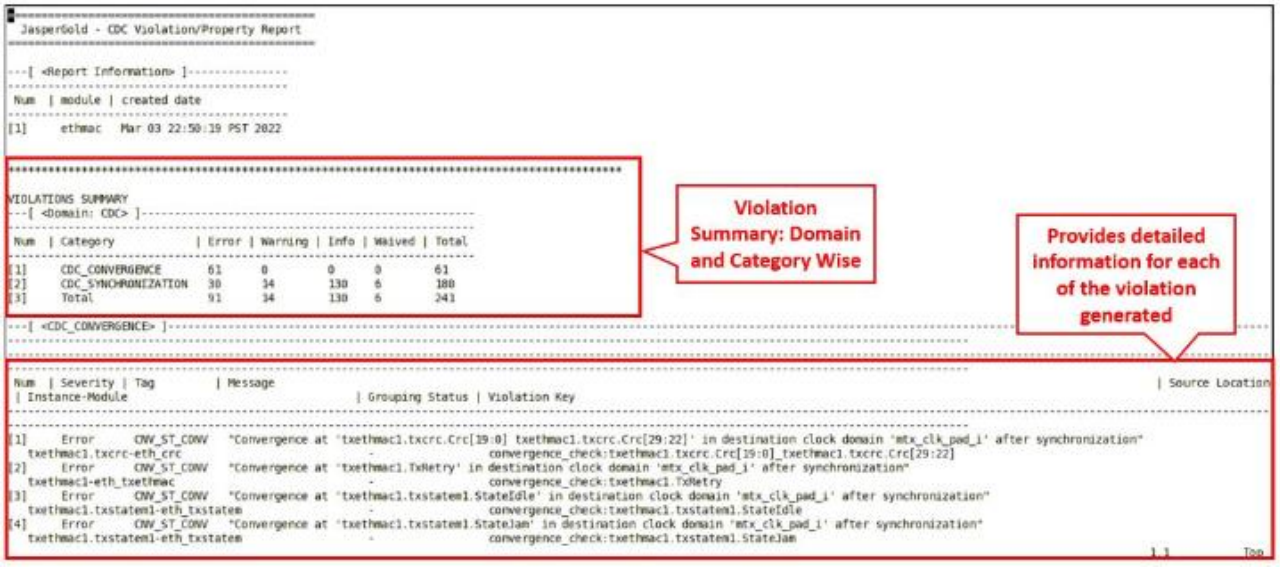
```
% check_cdc -report -file summary.txt
```

⚠ By default, all files are generated in the `.txt` format. If you want to generate a file in some other supported format, you can specify it with the switches listed in the command block.

Generating Detailed Violation Report

To generate a detailed violation report in the file, use the `-detailed` option as follows:

```
% check_cdc -report -detailed -file violation_report
```



Customized Violation Reports

Violation reports can be customized according to the requirement.

- If the report needs to be generated for a particular domain only, use the `-domain` switch along with the list of domain(s):

  ```
  % check_cdc -report -detailed -domain CDC -file domain_violation_report -force
  ```

- If the report needs to be generated for some categories, use the `-category` switch along with the list of categories:

  ```
  % check_cdc -report -detailed -category {CDC_CONFIGURATION CDC_SYNCHRONIZATION} -file category_violation_report -force
  ```

- If the report needs to be generated for some specific tags, use the `-tag` switch along with the list of tags:

```
% check_cdc -report -detailed -tag {PRT_IS_UNCK CLK_NO_DECL CLK_IS_CNST
CKS_NO_CONS CDC_NO_SYNC CDC_PR_LOGC} -file tag_violation_report -force
```

- If the report needs to be generated for some specific instances, use the `-instance` switch along with the list of instances:

  ```
  % check_cdc -report -detailed -instance IP1 -file ip1_violation_report -force
  ```

- If the report needs to be generated for a specific pattern matching a particular category, instance, or tag, use the `-regexp` switch:

```
% check_cdc -report -detailed -regexp -instance eth* -file eth_violation_report -
force
```

- If you do not want to include waived violations in the violation report, use the following command:

  ```
  % check_cdc -report -detailed -waived none -file violation_report -force
  ```

- If you want to report the violations in a grouped format, use the following command:

```
% check_cdc -report -violation -order category -detailed -grouping basic -file
grouped_violation_report.txt -force
```

Customized Waived Violation Reports

- To generate an exclusive report for waived violations, use the following command:

  ```
  % check_cdc -report -detailed -waived only -file waived_violations -force
  ```

- Like violation reports, waived violations reports can be customized depending on the type and status. If you want to generate a waiver report for specific waiver types, use the following commands:

  - `% check_cdc -report -detailed -waived only -hierarchical -file hierarchical_waivers`
  - `% check_cdc -report -detailed -waived only -imported -file imported_waivers`
  - `% check_cdc -report -detailed -waived only -user -file user_waivers`
  - `% check_cdc -report -detailed -waived only -safe -file safe_waivers`
  - `% check_cdc -report -detailed -waived only -unconditional -file unconditional_waivers`

Customized Waiver Reports

- To generate an exclusive waiver report based on either status, type, or both, use the following command:
- `% check_cdc -report -waiver`
  `[-waiver_status ( safe | unsafe | unconditional | not_processed | unmatched | unwaived)]`
  `[-waiver_type (user | imported | hierarchical | auto)]`

Scripting Interface

You can also list all CDC objects on the Jasper console using the `check_cdc -list` command. To list detected objects for a specified type, use the following syntax:

```
check_cdc -list ( pairs | violations | domains | rules
                | filters | signoff | signal_config
                | user_defined_schemes | waivers
                | scheme_properties | domain_crossings
                | groups | schemes | checks | units
                | clock_signals | inactive_pairs
                | clock_groups | clock_matrix | setup
                | ports | reset_order | reset_definitions
                | reset_paths | reset_schemes
                | inactive_reset_pairs | rdc_pairs
                | clock_synchronicity | virtual_clocks | icg_cells
                | structural_cells | virtual_resets | convergence_schemes
                | clock_sense | false_paths)
               [-filter filter_id]
```

Since all information in the GUI is listed, you can create a number of scripts using the `-list` information.

Accessing Information by Scheme Name

For example, to access all the information for a certain scheme, pass the key (scheme name) as index: `dict get $DICTIONARY scheme_name`

For example,

```
set dictionary [check_cdc -list schemes]
dict get $dictionary FSM_block.control_counter_pulse
```

returns

```
Status Passed Scheme FSM_block.control_counter_pulse {Scheme Type} Edge {Detection
Type} Automatic Violation {} {Connection Map} {{DQ1 {FSM_block.counter_sync.data_meta}}
{DQ1RV {1'b0}} {CombOut {FSM_block.control_counter_pulse}} {DclkIsFastest {1}}
{DclkPolarity {1}} {Dout {FSM_block.control_counter_pulse}} {Data
{FSM_block.counter_sync.data_in}} {Drst {~FSM_block.counter_sync.rst_n}} {Dclk
{FSM_block.counter_sync.clk}}} {NDFF Chain} {{FSM_block.counter_sync.data_meta
FSM_block.counter_sync.data_sync FSM_block.control_counter_reg}} Pairs
control_block.control_counter_r-FSM_block.counter_sync.data_meta
```

To list all keys, use `dict keys $dictionary`, which in this example, returns the following:

```
sync_chain_out sync_out sync_out2 conv_out2 control_block.counter_en_sync
FSM_block.buf_lsw_sync0.data_meta FSM_block.buf_lsw_sync1.data_meta
FSM_block.buf_lsw_sync2.data_meta FSM_block.buf_lsw_sync3.data_meta
FSM_block.buf_lsw_sync4.data_meta FSM_block.buf_lsw_sync5.data_meta
FSM_block.buf_lsw_sync6.data_meta FSM_block.buf_lsw_sync7.data_meta
FSM_block.error_sync.data_sync hsync_inst.tx_block.ack_sync.out_sync {reg_shft.s[1]}
{shift0.s[1]} {shift1.s[1]} {shift2.s[1]} {shift3.s[1]} {shift4.s[1]} {shift5.s[1]}
{shift6.s[1]} {shift7.s[1]} {control_block.inv_data_out[15:0]}
control_block.i_manager.msynchronizer.data_sync hsync_inst.rx_block.dataout
ctrl2fsm_fifo_sync.rdata fifo_sync1.buffer_data fifo_sync2.buffer_data buffer_lsw_sync
FSM_block.control_counter_pulse FSM_block.control_state_pulse
```

Accessing Information Inside a Scheme

With the scheme dictionary, you can access any information inside a scheme providing a key.

For example,

```
set scheme_info [dict get $dictionary FSM_block.control_counter_pulse]
dict get $scheme_info Pairs
```

returns

```
control_block.control_counter_r-FSM_block.counter_sync.data_meta
```

Counting Violations

Also, if you want to count how many `sync_chain_logic` violations the tool found, you can use the following script:

```
set scheme_list [check_cdc -list schemes]
set schemes {}
foreach key [dict keys $scheme_list] {
if {[lsearch [dict get [dict get $scheme_list $key] \ Violation] sync_chain_logic] >= 0} {
lappend schemes $key
}
}
puts $schemes
```

Filtering and Removing a Scheme Type

Use `check_cdc -list -filter` *`filter_id`* to list only the IDs of the objects related to the specified filter.

For example, if you want to remove all schemes of type `edge`, you can create a filter as follows,

```
set filter_id [check_cdc -filter -add -scheme_type edge]
```

and then use the filter to list only `edge` schemes:

```
set edge_schemes [check_cdc -list schemes -filter $filter_id]
```

The result might be as follows:

```
FSM_block.control_counter_pulse {Status Passed Scheme FSM_block.control_counter_pulse
{Scheme Type} Edge {Detection Type} Automatic Violation {} {Connection Map} {{DQ1
{FSM_block.counter_sync.data_meta}} {DQ1RV {1'b0}} {CombOut
{FSM_block.control_counter_pulse}} {DclkIsFastest {1}} {DclkPolarity {1}} {Dout
{FSM_block.control_counter_pulse}} {Data {FSM_block.counter_sync.data_in}} {Drst
{~FSM_block.counter_sync.rst_n}} {Dclk {FSM_block.counter_sync.clk}}} {NDFF Chain}
{{FSM_block.counter_sync.data_meta FSM_block.counter_sync.data_sync
FSM_block.control_counter_reg}} Pairs control_block.control_counter_r-
FSM_block.counter_sync.data_meta} FSM_block.control_state_pulse {Status Passed Scheme
FSM_block.control_state_pulse {Scheme Type} Edge {Detection Type} Automatic Violation
{} {Connection Map} {{DQ1 {FSM_block.state_sync.data_meta}} {DQ1RV {1'b0}} {CombOut
{FSM_block.control_state_pulse}} {DclkIsFastest {1}} {DclkPolarity {1}} {Dout
{FSM_block.control_state_pulse}} {Data {FSM_block.state_sync.data_in}} {Drst
{~FSM_block.state_sync.rst_n}} {Dclk {FSM_block.state_sync.clk}}} {NDFF Chain}
{{FSM_block.state_sync.data_meta FSM_block.state_sync.data_sync
```

```
FSM_block.control_state_reg}} Pairs control_block.control_state_r-
FSM_block.state_sync.data_meta}
```

To remove the schemes, you can now use the dictionary keys:

```
foreach key [dict keys $edge_schemes] {check_cdc -scheme -remove $key}
```

Once a particular scheme is removed, the tool tries to automatically infer the scheme. If no valid synchronizing scheme is detected, CDC_NO_SYNC violations are generated on the CDC crossings.

6

Bottom-Up Hierarchical Flow

With large SoC designs, it can be cumbersome to perform flat analysis. Not only can flat analysis result in a huge violation count, which might take a lot of time to disposition, but it might also cause some scalability problems. In such scenarios, you can leverage the bottom-up hierarchical methodology to perform CDC/RDC analysis at the SoC level. The methodology is as follows:

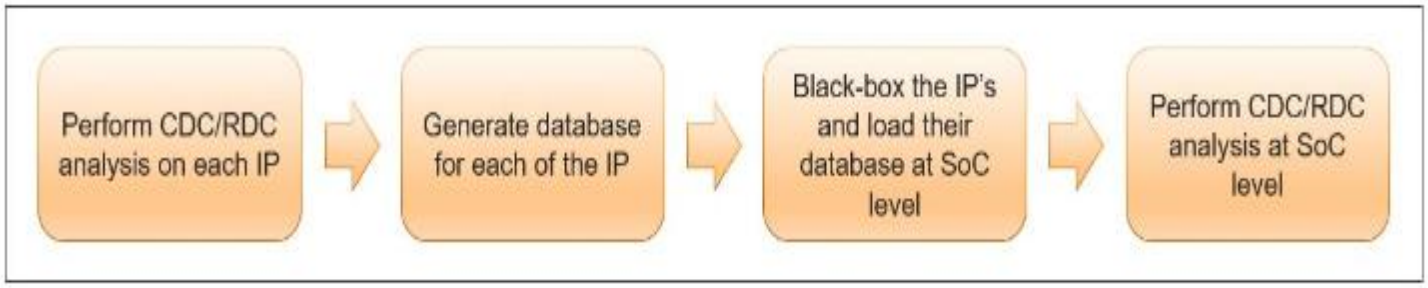
- IP and SoC can be developed in parallel.
- Once the IPs have been verified in the standalone environment, they can be integrated at the higher level.
- The integration of IPs at SoC level helps the SoC integrator to identify the inconsistencies that might exist between the IP and SoC-level setup.

This chapter introduces you to the bottom-up hierarchical flow. The topics covered follow:

Understanding the Flow

The steps you need to perform while using the bottom-up hierarchical flow follow:

1. Perform the CDC/RDC analysis at the IP level
2. Generate the databases for each of the IPs
3. Blackbox the IPs and loading the databases at the SoC level
4. Complete CDC/RDC analysis at the SoC level



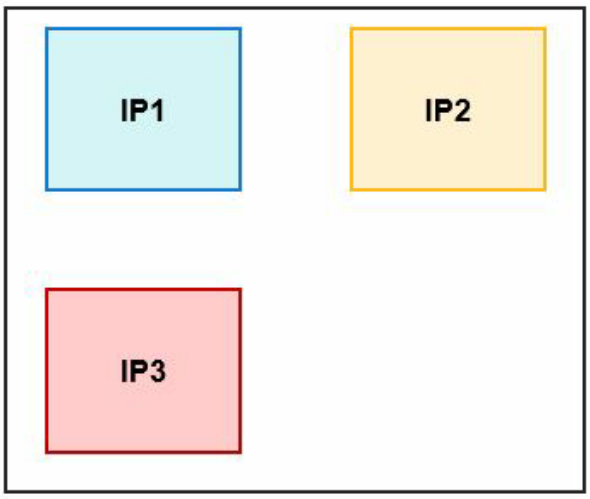
This section provides information regarding the various steps you need to follow while implementing the bottom-up hierarchical flow. The topics covered follow:

- Performing CDC/RDC Analysis IP Level
- Generating Databases
  - Reviewing Information Stored Inside the Database
- Loading Databases at the SoC Level
- Validating the Loaded Databases

Performing CDC/RDC Analysis at Each IP

The first step in the bottom-up hierarchical flow is to ensure that the IP blocks being used at the SoC or sub-system level are CDC and RDC clean. Thus, you should perform the complete analysis including both the structural and functional analysis. Ensure that all the IPs have been configured properly, all the CDC crossings are synchronized, and there are no convergence violations. Similarly, ensure that all reset-related violations have been dispositioned and that the design is free of any RDC crossings.

In the example below, there are three IPs that are CDC and RDC clean. All three IPs have been developed in the standalone environment by their respective designers and will later be integrated by the SoC integrator at the SoC level.



Generating Databases

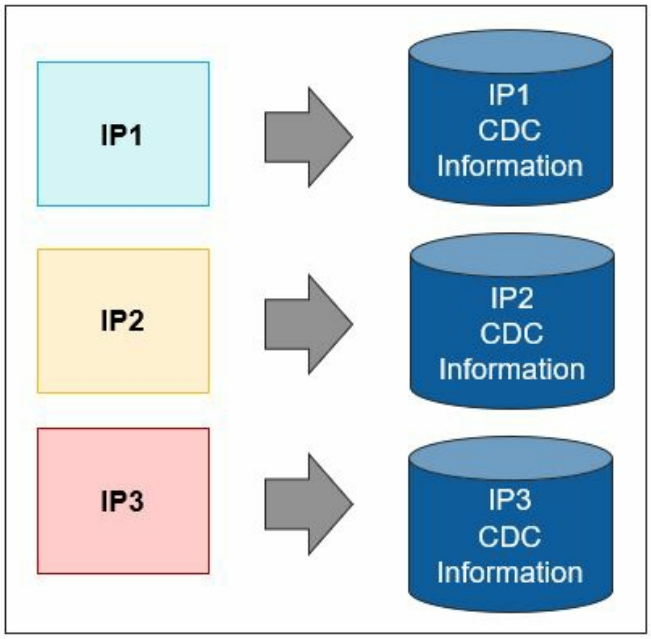
Once the CDC and RDC analysis is complete at the IP level, you can generate the database for each IP individually using the following command:

```
% check_cdc -export -database -file <file_name> [-force]
```

- Use `-database` to export the current CDC database to the specified file, which can later be loaded in the design using another command.
- Use `-file` to export the database to a specified file.
- Use `-force` to overwrite an existing database file.

For example, the databases for the three IPs can be generated using the following commands:

```
% check_cdc -export -database -file database_folder/ip1.jdb -force
% check_cdc -export -database -file database_folder/ip2.jdb -force
% check_cdc -export -database -file database_folder/ip3.jdb -force
```



⚠ The databases generated using `check_cdc -export` are different from the databases generated using `check_cdc -save`. For the bottom-up hierarchical flow, all databases need to be generated using the `check_cdc -export` command only.

Reviewing Information Stored Inside the Database

Once the databases have been generated, you can review the information that has been saved using the following command:

```
% check_cdc -hierarchical -report <report_name> -info <database_name>
```

Use `-report` to generate a report from the database specified with the `-info` switch.

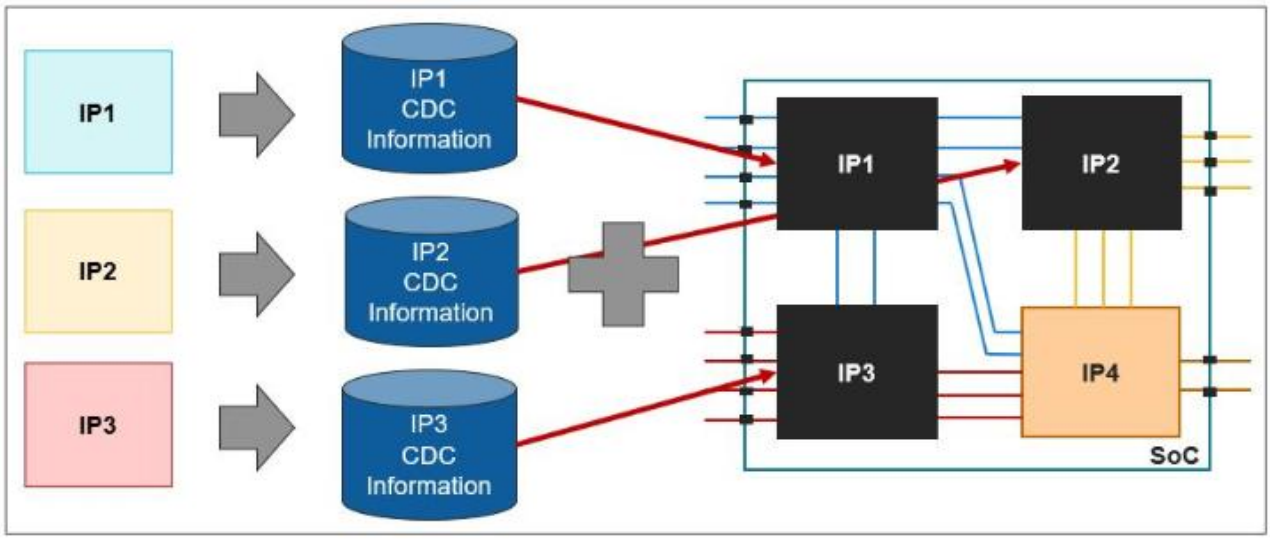
Loading Databases at the SoC Level

CDC/RDC clean IPs whose databases have been generated need to be blackboxed at the SoC level. To blackbox the IPs, you need to specifically mention the instance or module names in the `elaborate` command as follows:

```
% elaborate -top <top_name> -bbox_i {instance_list} -bbox_m {module_list}
```

For example, the three IPs whose databases have been generated can be blackboxed as follows:

```
% elaborate -top soc -bbox_m {IP1 IP2 IP3}
```



After blackboxing the IPs, you can use the following command to load the databases at the SoC level:

```
% check_cdc -hierarchical -load (-directory <directory name> |-file <file name or pattern> |-instance <instance name>)
```

- Use `-load` to load and open database files.
  - Use `-directory` with `-file` to open all the database files in the directory that match the file name or pattern list.
  - Use `-file` to open database files from the list.
  - Use `-instance` with `-file` to associate a specified file with the instance and store this association.

The files specified in the command above should be valid CDC databases generated with `check_cdc -export -database` command.

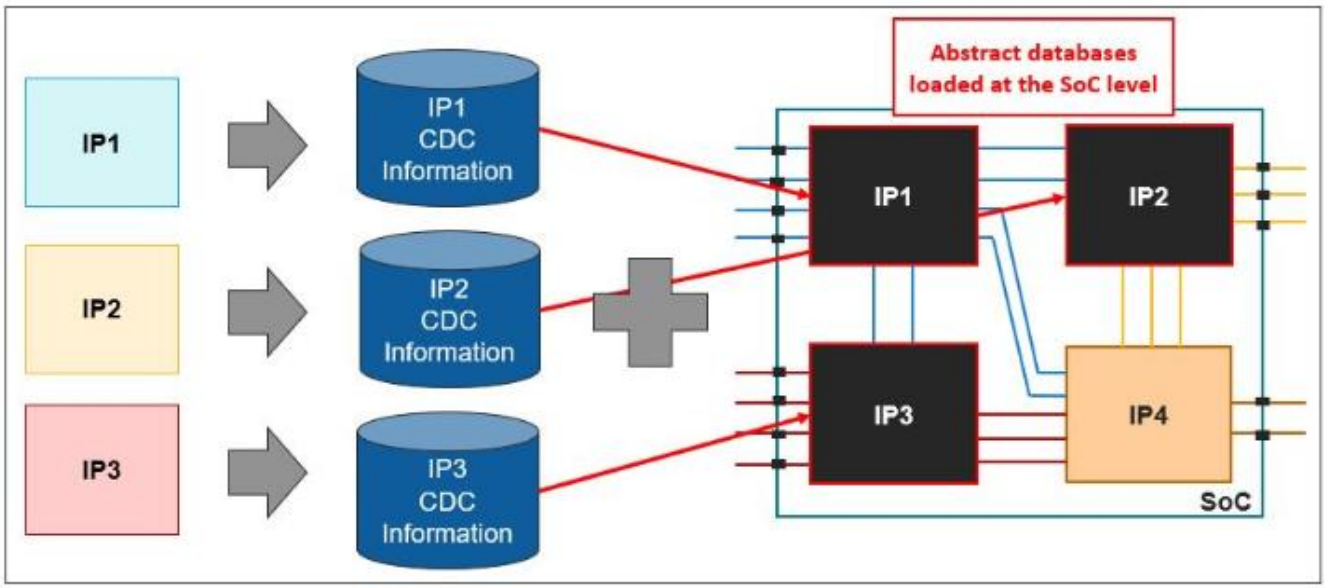
⚠
- The databases can be loaded in the design at any time after the `elaborate` command and before running `check_cdc -clock_domain -find`.
- You cannot load hierarchical CDC databases if their corresponding module instantiation in the elaborated design does not have a compatible port interface. If the tool finds no matching module in the elaborated design, the `check_cdc -hierarchical -load` command fails with downgradable error ECDC110.

For example, the three IPs whose databases have been generated can be loaded at the SoC level

as follows:

```
% check_cdc -hierarchical -load -directory {database_folder} -file {*.jdb}
```

The above command loads all the abstract databases present inside the `database_folder` and matching the pattern `"*.jdb"`



Validating the Loaded Databases

After the entire design has been analyzed and elaborated and all the abstract databases have been loaded in the design, it is important to verify that all databases have been loaded correctly before conducting the CDC and RDC analysis.

You can verify the correctly loaded databases as follows:

- Ensuring no HDB_IN_LDFL violation is reported
- Using the list command
- Viewing the databases in the schematic viewer

Understanding the HDB_IN_LDFL Violation

The `HDB_IN_LDFL` violation is reported when the tool is unable to correctly load the abstract database in the design due to the following reasons:

- Unmatched module or parameter – The parameter value at the IP and SoC level do not match.

- Unable to open the file – The user-specified database could not be found when trying to load a hierarchical database.
- File is corrupted – The database you are trying to load has been corrupted.
- File is not a valid CDC file – The database has been generated by a command other than `check_cdc -export -database`.
- Unsupported Version – The database being loaded was exported from an older Jasper version.

These violations are reported only when errors `ECDC100` to `ECDC105` are downgraded to warnings, allowing all the fixes in the databases to be done together. Otherwise, all these errors are generated individually, and you are required to fix the incorrectly loaded database before moving forward.

The errors can be downgraded to warnings as follows:

```
set_message -warning ECDC100

set_message -warning ECDC101

set_message -warning ECDC102

set_message -warning ECDC103

set_message -warning ECDC104

set_message -warning ECDC105
```

⚠ The `HDB_IN_LDFL` violation is reported automatically while running the `check_cdc -clock_domain -find` command.

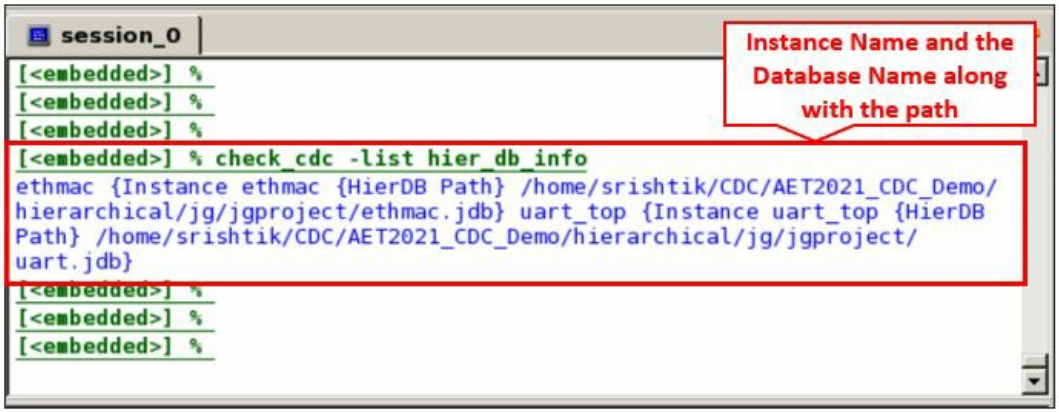
It is important to debug the `HDB_IN_LDFL` violations before analyzing the rest of the violations that have been reported. To get more information about the violation, you can refer to the *CDC Checks Reference*.

Using the List Command

To verify that all the databases have been loaded correctly for each of the IP instances, use the following command:

```
% check_cdc -list hier_db_info
```

The command returns a Tcl dictionary containing information regarding the databases that have been successfully loaded in the design along with the path and the instance name for which the database has been loaded.



Viewing Abstract Databases in the Schematic

Another way to confirm that all your databases have been loaded successfully in the design is through the schematic viewer. All the IPs/modules that have been blackboxed and for which the databases have been successfully loaded are highlighted in red in the schematic viewer.

Debugging Violations

This section provides information on debugging hierarchical violations. It includes the following topics:

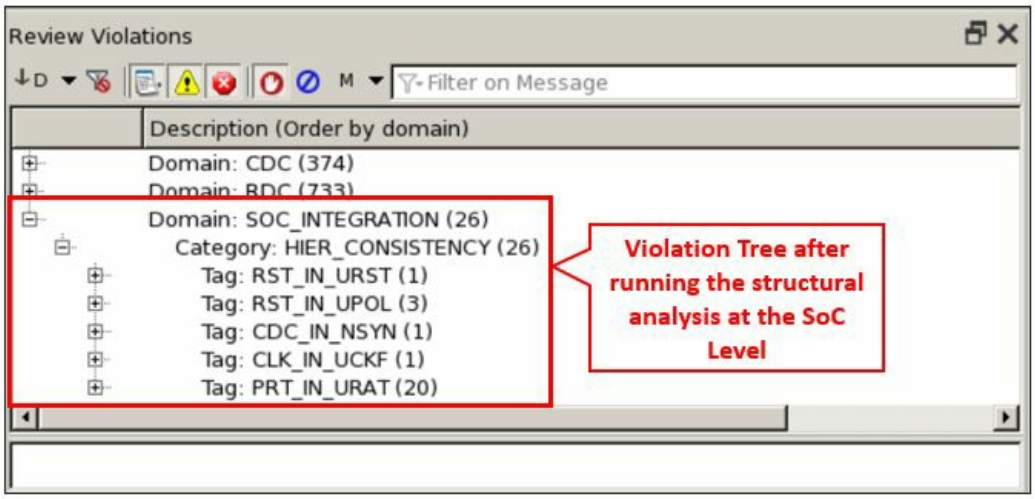
- Violation Tree
- Schematic Viewer

Violation Tree

Once it has been confirmed that all the abstract databases have been loaded correctly in the design, you can run the CDC analysis at the SoC level.

In terms of the commands and the flow, performing CDC/RDC analysis at the SoC level for bottom-up hierarchical flow is exactly the same as for flat analysis with the only difference being that the tool reports a new domain of violations `SOC_INTEGRATION` in the violation tree.

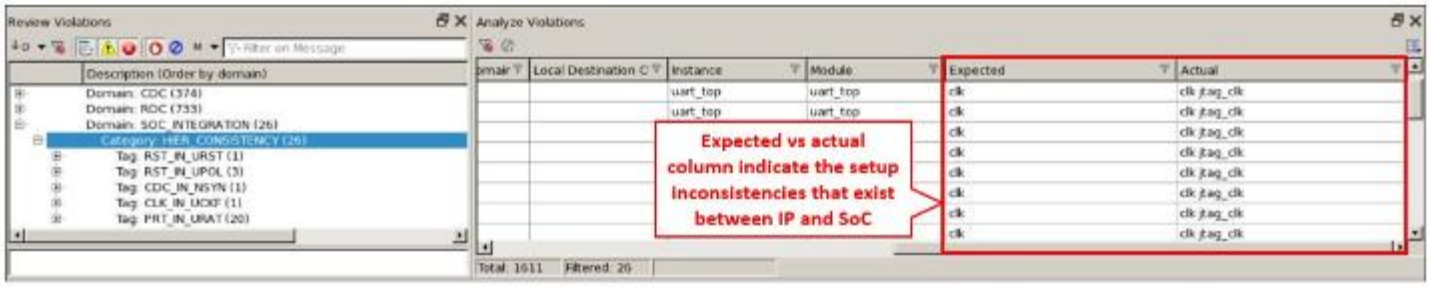
Since one of the main purposes of using the bottom-up hierarchical methodology is to catch the inconsistencies in the setup between the IP and SoC level, all violations associated with such differences are reported under the `SOC_INTEGRATION` domain; whereas, the rest of the violations that are encountered by the tool while running the analysis are reported under CDC and RDC domain violations as applicable.



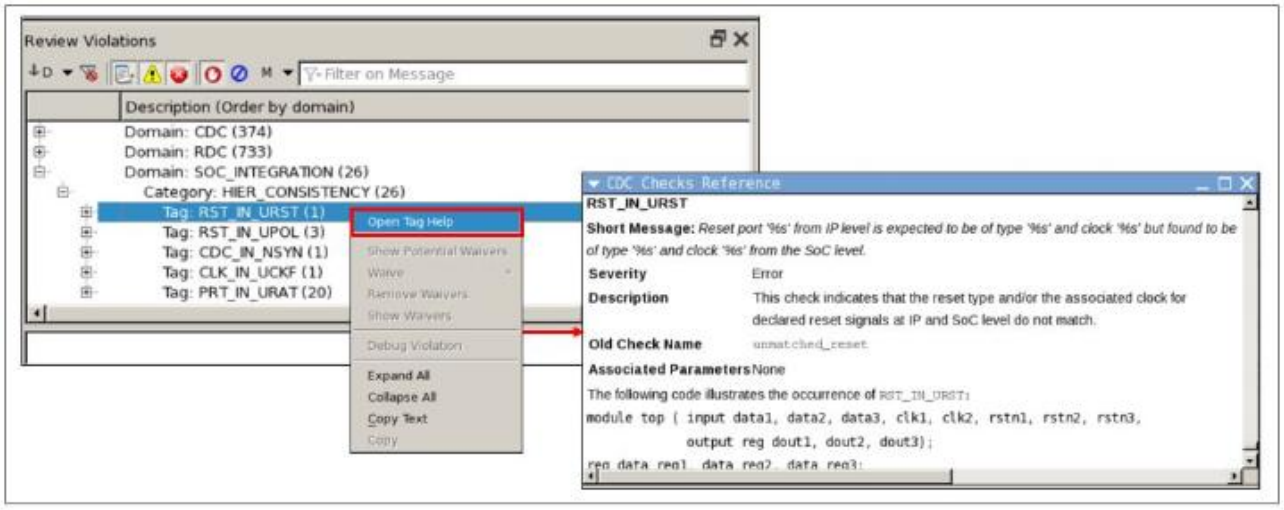
Moreover, to understand the inconsistency between the IP and the SoC level setup, two new columns *Expected* and *Actual* have been added specifically for `SOC_INTEGRATION` domain violations in the *Analyze Violation Table.*

- The *Actual Column* indicates the setup information at the SoC level.

- The *Expected Column* indicates the configuration that has been applied to that particular port at the IP level.



Before debugging violations, it is important to understand why a particular violation is being reported. For that, see the *CDC Reference Guide*. To view the reference manual information for each tag from the GUI, right-click on the violation tag and select *Open Tag Help*. The help provides a detailed description of the violation, a small example to illustrate a scenario in which the violation is reported, and information on possible courses of action that can help with debugging the violation under consideration.

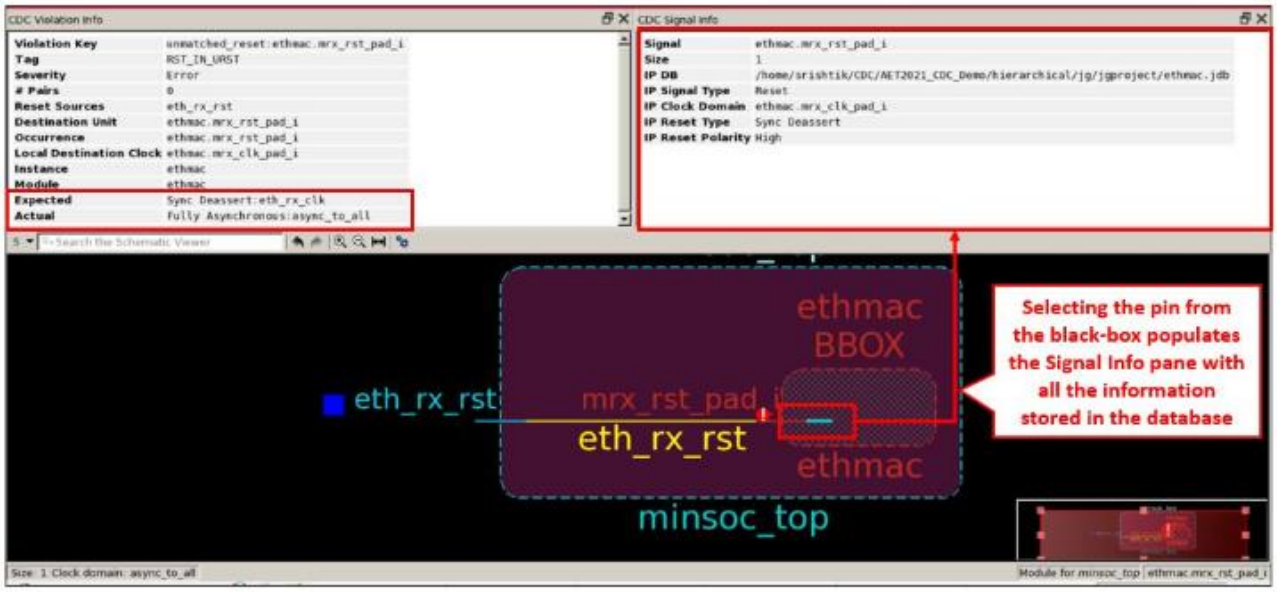


Schematic Viewer

The process for debugging violations reported under `SOC_INTEGRATION` category is the same as that for any other CDC/RDC violation reported by the tool. However, since the bottom-up hierarchical methodology uses a blackboxing approach, some additional features and information is available in the schematic view to make debugging easier.

When you open the schematic viewer, you see a schematic view along with the populated *Violation Info Pane* and *Signal Info Pane*. The information available in the *Signal Info Pane* is variable and depends on the pin or wire being selected from the schematic.

In the schematic view, if you select a pin or a port from the schematic on which a violation is being reported, the *Signal Info Pane* is populated with all the details necessary to debug the violation that have been stored in the database for that particular pin.



⚠ The other features available for debugging CDC and RDC structural violations are still applicable and available for all the `SOC_INTEGRATION` violations that are reported by the CDC App.

7

CDC Violations

This appendix provides a full list of CDC violations (see Table 7-1). For detailed discussion of these violations and possible courses of action for addressing them see the CDC Checks Reference (*Help – Application Guides – CDC Checks Reference*).

⚠ Violations are listed in the order they appear in the default `cdc.def` file located at `<installation>/etc/res/rtlds/rules/cdc.def`.

Table 7-1 CDC Violations

Configuration Checks	Severity	Description
PRT_IS_UNCK	Error	The specified signal is not associated with any clock signal. The tool considers every unclocked signal as an independent clock domain and all the connections are reported as clock domain crossings.
CLK_GT_ASYN	Error	This check indicates that the enable and clock signal converging in the clock path are asynchronous to each other.
CLK_IS_CNST	Warning	Specifies the primary inputs declared as clock ports are driven by a constant value, making the clock inactive. A constant clock can cause genuine CDC crossings to be inactive

CLK_NO_DECL	Error	The specified signal is a primary input driving clock ports of the sequential elements in the design, but it has not been declared as a clock.
SIG_NO_STAT	Error	The specified signal is declared as static but is not actually static.
SIG_NO_GRAY	Error	A signal declared as gray encoded is actually not gray encoded.
SIG_NO_EXCL	Error	The specified signals are declared as exclusive but are not actually exclusive.
CND_NO_CONS	Warning	A signal declared as constant with condition is actually not constant.
CKS_NO_CONS	Warning	The select pin of a clock mux is not constrained as a constant value.
CLK_IS_CONV	Error	The check identifies scenarios where one or more clocks converge into a combinational logic before driving the clock input of a flop.
CKS_IS_CONV	Error	One or more select signals diverge into multiple paths and converge at the combinational logic before driving the select input of a clock mux.

LTC_NO_INIT	Error	A latch is disabled (latch.EN = 0). If latch.EN = RTL constant 0, CDC reports a violation, indicating an uninitialized latch. If latch.EN = non-RTL constant 0 (for example, assumption, set_case_analysis, and so forth), the tool reports a violation but automatically waives it. In this case, the tool assumes that the initialization process by which the latch.EN went to 0 is verified by the user.
SYN_NO_TARG	Info	This check indicates that the sync enabler scheme has not been added as there are no target CDC pairs that have been identified by the tool.
Structural Checks	**Severity**	**Description**
CDC_NO_SYNC	Error	The specified CDC crossing has not been properly synchronized in the destination domain.
CDC_PR_LOGC	Error	This check indicates the existence of combinational logic on the specified CDC path.
CDC_PR_FOUT	Warning	The CDC path fans out to multiple destinations and might include scenarios in which a single bit from the CDC source fan outs to multiple bits of the same CDC destination signals.
CDC_PR_INAC	Info	The tool identifies constant propagation on the CDC path causing the CDC pair to be inactive.

CDC_PR_LTCH	Error	All the sources of a latch are being driven by different domains.
SYN_DF_SLGC	Error	This check indicates the existence of a combinational logic between the flops in the synchronizer chain of the specified NDFF scheme.
SYN_DF_CKPH	Error	The flip flops in the specified synchronizer are driven by different clock phases.
SYN_DF_FOUT	Error	The tool has identified fanouts in the NDFF synchronizer chain.
SYN_DF_RDND	Warning	A CDC pair is synchronized more than once with a NDFF in the same clock domain.
SYN_SE_INVL	Error	CDC has identified an invalid sync enabler
SYN_GP_ASIP	Error	The tool has identified an invalid input for the specified glitch protector synchronizer.
SYN_GP_ASOP	Error	The tool has identified an invalid output for the glitch protector synchronizer.
Convergence Checks	**Severity**	**Description**
CNV_ST_GLCH	Error	Two or more signals of the source clock domain converge into a combinational logic on the CDC path. The logic on the CDC path can cause glitches in the design.

CNV_ST_CONV	Error	There is convergence of multiple paths from the same or different source clock domains after synchronization into a combinational logic before or after the sequential elements in the destination clock domain.
CNV_ST_BCNV	Error	This check identifies scenarios where the tool finds the potential convergence of one or more bits of the source signal after synchronization outside the module.
Functional Checks	**Severity**	**Description**
CTL_NO_STBL	Error	The signal from the source clock domain becomes unstable before it can be captured in the destination clock domain.
PLS_NO_STBL	Error	The output from source clock domain becomes unstable before it can be captured in the destination clock domain.
DAT_NO_STBL	Error	The data becomes unstable in the destination domain during the data transfer phase.
REQ_NO_HOLD	Error	The request signal is deasserted before receiving acknowledgment from the destination clock domain.
NRQ_WO_DACK	Error	The sender submits a new request before the acknowledgment for the previous transfer is de-asserted.

ACK_WO_SREQ	Error	The signal is deasserted before the acknowledgment is received from the destination clock domain.
NAK_WO_SREQ	Error	The receiver asserts a new acknowledgment before a new request is received.
DAT_HS_STBL	Error	The data from the sender becomes unstable before it receives an acknowledgment from the receiver.
PSH_ON_FULL	Error	It is possible to write to a full FIFO.
POP_ON_EMTY	Error	It is possible to read from an empty FIFO.
WPT_NO_GRAY	Error	This check indicates that the gray encoding for the specified write pointer of the specified FIFO has been failed.
RPT_NO_GRAY	Error	The gray encoding for the specified read pointer of the specified FIFO has failed.
BUS_NO_GRAY	Error	The gray encoding for a data bus synchronized using an NDFF_BUS synchronizer has failed.
PLS_IP_WDTH	Error	The input of the pulse synchronizer is more than one source clock wide.
PLS_OP_WDTH	Error	The output of the pulse synchronizer is more than one destination clock wide.
PLS_NO_TOGL	Error	The toggle is not working as expected.

GPD_NO_STBL	Error	The data path in a glitch protector becomes unstable when the enable signal is asserted in the destination clock.
SYN_NO_STBL	Error	The sync enabler data path becomes unstable when the enable signal is asserted in the destination clock.
INC_FN_PROP	Error	The functional protocol checks for the synchronizer could not be generated.
Metastability Checks	**Severity**	**Description**
CHK_MS_FAIL	Error	A user-defined property has failed due to metastability injection.
INJ_MS_FAIL	Error	Metastability injection has failed.
Reset Checks	**Severity**	**Description**
RST_RS_CONV	Error	There are convergence issues in the reset tree during reset analysis. Reset convergence happens when one reset signal diverges and combines into a combinational logic or a group of reset signals combine into a combinational logic before driving the reset pin of the destination unit.
RST_NO_DECL	Error	The specified reset has not been declared.
RST_NO_SYNC	Error	The specified reset pair has not been synchronized.

RST_RS_FSAR	Error	This check indicates that the reset signal is forcing a flop to be permanently stuck at reset due to external constraints.
RDC_RS_DFRS	Error	The connected flops in the same clock domain are driven by different asynchronous/synchronized reset signals.
RDC_RS_GLCH	Error	More than one flop from different reset domains is combining into a combo logic.
RST_RS_INAC	Info	The tool has identified an inactive reset pair.
RST_RS_RIDP	Error	The specified reset signal is driving the data pins of one or more flops/latches.
SYN_RS_CKPH	Error	Some elements of the reset synchronizer are driven by different clock phase.
SYN_RS_SMDR	Error	The flops of the specified reset synchronizer do not have the same reset signal.
SYN_RS_SLGC	Error	This check indicates the existence of combinational logic on the sync chain.
SYN_RS_RDND	Warning	The reset signal from source to destination clock domain is synchronized more than once in the destination domain.
RST_RS_FOUT	Error	The fanning out of the asynchronous reset signals to multiple points in the same destination clock domain.

SoC Integration Checks	Severity	Description
RST_IN_UDEF	Error	This check indicates that the input port declared as reset at the IP level is not being driven by a reset signal at the SoC level.
RST_IN_URST	Error	This check indicates that the reset type and/or the associated clock for declared reset signals at IP and SoC level do not match.
RST_IN_UPOL	Error	This check indicates whether the reset polarity for the reset signals at both the IP and SoC level is same.
RST_IN_RORD	Error	This check indicates whether the reset order defined for declared reset signals at the IP level matches the reset order defined at the SoC level.
PRT_NO_CONS	Error	This check indicates whether a constant port at the IP level has been declared as constant at the SoC level.
PRT_DF_CONS	Error	This check indicates whether the value of the port at the IP level matches the port value at the SoC level.
CDC_IN_NSYN	Error	This check indicates whether the output signal of an IP needs to be synchronized at the SoC level to prevent metastable values from being propagated through the design.

CDC_IN_UDEF	Error	This check indicates whether the input clock port at the IP level is connected to a clock at the SoC level.
CLK_IN_UCKF	Error	This check indicates whether the clock signal at the IP level has the same clock factor at the SoC level.
CLK_IN_UREL	Error	This check indicates that there is a mismatch in the clock relationship between the IP and the SoC level clocks. It is reported in scenarios where the mismatch in relationship can cause metastable values to be propagated.
CLK_IN_WREL	Warning	This check indicates that there is a mismatch between the clock relationships at the IP and SoC level. It is reported in scenarios where a mismatch in clock relationship is safe and might not cause metastable problems in the design.
INV_IN_ASUM	Error	This check indicates that the assumptions applied on the boundary signals at the IP level cannot be validated at the SoC level.
INV_IN_PCND	Error	This check indicates that the condition defined on the IP level to make the conditional waiver pass cannot be validated at the SoC level.

INT_IN_PROP	Warning	This check indicates that the assumptions defined on the internal nets of the IP could not be proven at the SoC level.
PRT_IN_URAT	warning	This check indicates the mismatch in clock association of black box input or primary output ports and their signal drivers.
HDB_IN_LDFL	Error	This check indicates that the hierarchical database could not be loaded correctly at the SoC level.

8

CDC Rule File

CDC rules can be customized to meet corporate, team-wide, and user-specific design purification requirements. You can change the severity of rules, enable or disable rules individually, define parameters of naming convention rules, and customize different rule parameters. You can also add new categories and definitions to customize the CDC rules file.

A rules file is a regular text file that contains the definitions of the CDC rule categories and messages. You can find the default rules file `cdc.def`, which contains CDC category and definition defaults, at the following location:

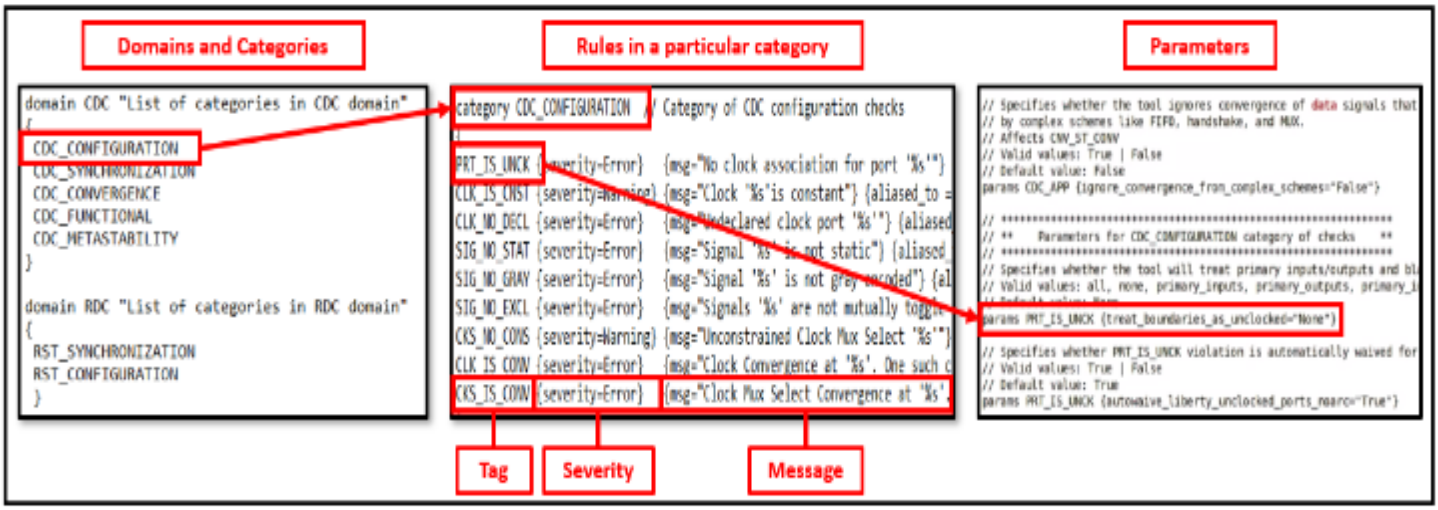
```
install_dir/etc/res/rtlds/rules/cdc.def
```

This chapter includes information on the following:

CDC Rule File Organization

The CDC rule file is divided into two major segments. They are as follows:

- Rules Section – Contains a set of rules that are reported by the tool in the violation tree.
- Parameters Section – Contains a list of parameters that control the reporting of rules in the violation tree.



Rules Section Organization

Domains

The first segment containing the CDC rules is divided into three major domains CDC, RDC, and SOC_INTEGRATION.

- The CDC domain contains specific categories that report violations related to structural and functional CDC analysis of the design.
- The RDC domain contains specific categories that report violations related to structural RDC analysis of the design.
- The SOC_INTEGRATION domain contains specific categories that report violations related to hierarchical CDC and RDC analysis of the design

⚠ SOC_INTEGRATION domain has been kept for compatibility with future versions and is currently not available in the tool.

```
domain CDC "List of categories in CDC domain"
{
 CDC_CONFIGURATION
 CDC_SYNCHRONIZATION
 CDC_CONVERGENCE
 CDC_FUNCTIONAL
 CDC_METASTABILITY
}

domain RDC "List of categories in RDC domain"
{
 RST_SYNCHRONIZATION
 RST_CONFIGURATION
 }

domain SOC_INTEGRATION "List of categories in Hierarchical Domain"
{
 HIER_CONSISTENCY
}
```

Categories

Each domain is further divided into categories to address a specific area of the design analysis. Similarly, each category contains a specific set of rules that are reported as violations in the violation tree. Domain categories are explained below.

CDC Domain Categories:

- CDC CONFIGURATION – contains rules that check whether the design has been properly configured
- CDC SYNCHRONIZATION – contains rules that check whether every CDC path has been synchronized properly
- CDC CONVERGENCE – contains rules that check whether the design is glitch and convergence free
- CDC FUNCTIONAL – contains rules to check the functionality of the synchronizers being used in the design
- CDC METASTABILITY – contains rules to check the behavior of metastability injection in the design

RDC Domain Categories:

- RST CONFIGURATION – contains rules to check whether the reset signals have been declared properly and do not have any convergence issues before driving the clear pins of the flops
- RST SYNCHRONIZATION – contains rules to check whether every reset crossing has been synchronized properly

Rules

Each rule includes information on the severity, message, and the status of the check (`on` or `off`):

- Severity – defines the level of severity with which the violation in reported in the design
  There are three severity categories supported by the tool. They are ERROR, WARNING, and INFO.
- Message – provides explanation of the rule along with information of the point of occurrence of the violation
- Status – controls the reporting of the rules in the violation tree
  If the value is `off`, the tool does not report this violation in the design.

Parameters Section Organization

The second segment of the CDC rule file contains parameters that control the reporting of violations in the design. Each parameter contains the information described below:

- Description – explains the parameter functionality
- Default Value – indicates the default behavior of the parameter
- Supported Values – provides values that can be used to change the default behavior of parameter according to design requirements
- Affected Checks – provides a list of checks that are affected by this parameter
- Command Line – to change the default behavior of the parameter

The parameters are segregated and mapped to various rule categories depending on their functionality. The categories are explained below:

- Overall CDC Parameters – contains a list of parameters that affect the overall design analysis but are not associated to any specific category or check
- CDC CONFIGURATION Parameters – contains a list of parameters that affect the design configuration
  In this category, you will find every parameter associated with a certain rule present in the CDC CONFIGURATION category.
- CDC SYNCHRONIZATION Parameters – contains a list of parameters that control the synchronization aspect of the design analysis
  In this category, you will find every parameter associated with a certain rule present in the

CDC SYNCHRONIZATION category.

- CDC Domain Parameters –contains a list of parameters that affect mainly convergence analysis of the design
  In this category, the parameters are either associated with a certain rule present in the CDC CONVERGENCE category or the overall CDC design analysis.
- Overall RDC Parameters – contains a list of parameters that affect the overall RDC design analysis but are not associated to any specific category or check
- RST SYNCHRONIZATION Parameters – contains a list of parameters that control the synchronization aspect of the RDC design analysis
  In this category, you will find every parameter associated with a certain rule present in the RST SYNCHRONIZATION category.

⚠ For more details on parameters and rules, you can refer to CDC Check Reference Document.

Customizing CDC Rule File

Add new categories and definitions to customize the CDC rules file using the following statements:

- include
- softinclude
- severity
- message
- status
- aliased_to
- params

include

Use `include` to include another rules file within a custom rules file. The syntax of the include

statement follows:

```
include (filename | default)
```

Use `default` to include the default CDC rules file located at `install_dir/etc/res/rtlds/rules/cdc.def` within your custom rules file. For example, to load the default CDC rules file along with your custom rules file, do the following:

```
include default
```

`filename` specifies the name of the rules file you want to include within a custom rules file. For example, to include a local team's rules file `team.def` within the custom rules file, use the following statement:

```
include /usr/local/share/team.def
```

⚠ The location of the `team.def` file is relative to the path of the custom rules file. The rule files support Unix environment variables. If a rule file includes another rule file whose path is specified using an environment variable and the file is edited, the environment variable is retained as is and is not expanded.

softinclude

The `softinclude` statement is the same as the include statement except that the tool prints no error messages if the included file does not exist. The syntax of the `softinclude` statement follows:

```
softinclude (filename | default)
```

severity

The following three predefined severity parameters are recognized by CDC for all messages: `error`, `warning`, and `info`. You can use these predefined parameters to modify the default severity of predefined and custom messages.

For example, the following rules file changes the severity of the `CLK_IS_CNST` rule, which checks if a clock is driven by a constant value, to an error.

```
// Include the default rules file.

include default

// Change the severity of the CLK_IS_CNST to error.
```

CLK_IS_CNST {severity=Error} {msg="Clock '%s' is constant"}

⚠ If there are multiple changes to the severity of a message, the last one takes precedence. By default, a message with a predefined severity of error cannot be downgraded to a warning.

message

CDC provides the `msg` parameter so you can rephrase existing short messages.

For example, the `PRT_IS_UNCK` short message follows:

```
PRT_IS_UNCK {severity=error} {msg="No clock association for the '%s' port '%s'"}
```

You can modify the message in your customized rule file as shown below:

```
// Rephrase PRT_IS_UNCK short message.
PRT_IS_UNCK {severity = error} {msg="'%s' port '%s' is unclocked"}
```

⚠ The `PRT_IS_UNCK` rule used in the above example has two arguments represented by `%s`. The edited message must have the exact arguments as the original message or the tool returns an error.

status

All domains, categories, and most of the rules are enabled by default. To disable an entire domain or category, you must use `config_rtlds -rule -disable (-category <category_name_list> |-domain <domain_name_list>` or you can remove the domain/category along with the associated rules from the custom rule file.

To disable individual rules, you can specify the `status` parameter inside the category definition as shown below:

```
category <category_name>
{
<tag> <severity> <short_message> {status=off}
}
```

For example, to disable the `PRT_IS_UNCK` check within the `CDC_CONFIGURATION` category, modify your customized rule file as follows:

```
category CDC_CONFIGURATION
{
PRT_IS_UNCK {severity = error} {msg = "msg="No clock association for the '%s' port '%s'"}
{status=off}
....
}
```

Alternatively, you can specify the `status` param for a check at the end of the CDC rules file as follows:

```
params <tag> {status=off}
```

For example, you can add the following statement to the end of your custom rule file to disable the `PRT_IS_UNCK` rule:

```
params PRT_IS_UNCK {status=off}
```

aliased_to

The `aliased_to` statement renames a tag only. Using the `aliased_to` statement, you can rename the pre-defined tag to match any rules you might already have in place. The syntax of the `aliased_to` statement follows:

```
old_tag... {aliased_to=new_tag}
```

The following example renames the predefined rule `PRT_IS_UNCK` to a user-defined rule unclocked_signal

```
// Include the default rules file.
include default

// Rename PRT_IS_UNCK to unclocked_signal.
category CDC_CONFIGURATION
{
PRT_IS_UNCK {severity = error} {msg = "No clock association for the '%s' port '%s'"} {aliased_to
= unclocked_signal}
}
```

You can also rename multiple tags with a single message identifier using the `aliased_to` keyword. The following example renames the predefined rules CDC_PR_LOGC and CNV_ST_GLCH to a user-defined rule `glitch_check`.

```
// Include the default rules file.
include default

// Rename CDC_PR_LOGC and CNV_ST_GLCH to glitch_check.
category CDC_SYNCHRONIZATION
{
CDC_PR_LOGC {severity=Error}   {msg="Combo Logic on CDC path between source unit '%s' driven by
clock '%s' and destination unit '%s' driven by clock '%s'"}{aliased_to=glitch_check}
}
category CDC_CONVERGENCE
{
CNV_ST_GLCH {severity=Error}   {msg="Potential glitch at '%s' in source clock domain '%s'"}
{aliased_to=glitch_check}
}
```

You can specify the same tag to any number of message identifiers that have different long and short messages. You can choose to retain the same long or short message for all the aliased tags or specify a different message to each of them. However, you must exercise caution when using the `aliased_to` and `params` keywords together. Consider the following example:

```
// Rename PRT_IS_UNCK to unclocked_signal.
{
PRT_IS_UNCK {severity = error} {msg = "No clock association for '%s' port '%s'"}
{aliased_to=unclocked_signal}
}}
params unclocked_signal {msg = "'%s' port '%s' is unclocked"}
```

The tool ignores the above customization because you cannot replace an existing message identifier with a new one (`PRT_IS_UNCK` with `unclocked_signal`) and then change its parameter using `params`. You need to use the original tag for specifying parameters even if you have replaced the message identifier with a new name. This is because the message parameter changes done by `params` keyword are always processed first.

To change the `msg` parameter of the `PRT_IS_UNCK` check, use the following code instead:

```
// Rename PRT_IS_UNCK to RULE1.
{
PRT_IS_UNCK {severity = error} {msg = "msg="No clock association for the '%s' port '%s'"}
{aliased_to=unclocked_signal}
}}
params PRT_IS_UNCK {msg = "'%s' port '%s' is unclocked"}
```

params

A parameter usually consists of a name-value pair and is associated with a specific message identifier. The `params` statement passes information to the code that implements a specified rule. The syntax of the `params` statement follows:

```
params <tag> {parameter_name=value | severity=value}
```

Following are several examples using the `params` statement. For a complete list of CDC parameters, see the *Jasper CDC Checks Reference* (*Help – Application Guide – CDC Checks Reference*).

- Pass the value of a parameter `parameter_name` associated with a rule `tag`. For example, to specify the logic type allowed on the CDC path, use the following statement:

  ```
  params CDC_PR_LOGC {cdc_pair_logic="Buf"}
  ```

  ⚠ If the same parameter is assigned two or more different values, the last assignment takes precedence.

- Specify that the tool automatically waive structural violations due to constant or static values handling as follows:

  ```
  params CDC_APP {apply_auto_waivers="True"}
  ```

- Set the severity of the rule. The severity can be `error`, `warning`, or `info`. To set the severity of the `CLK_IS_CNST` to `error`, use the following statement:

  ```
  params CLK_IS_CNST {severity=error}
  ```

  ⚠ You cannot downgrade the default severity of a message with this parameter. For example, the default severity of `CLK_IS_CNST` is `warning`. You cannot change it to `info`.

For additional information on parameters affecting individual rules, see the *CDC Checks Reference* available from the tool (*Help – Application Guides – CDC Checks Reference*).

⚠ Once you have created a custom rules file, load the file using the `config_rtlds -rule -load <rule_file>` command. Enure the file is loaded before running the `elaborate` command as there might be some parameters that affect the design elaboration.

9

Supported SDC Commands

The following is the list of SDC commands that are supported by the CDC App.

- `add_to_collection`
- `all_clocks`
- `all_inputs`
- `all_instances`
- `all_outputs`
- `append_to_collection`
- `compare_collections`
- `copy_collections`
- `create_clock`
- `create_generated_clock`
- `current_design`
- `current_instance`
- `filter_collection`
- `foreach_in_collection`
- `get_cells`
- `get design`
- `get_designs`
- `get_clocks`
- `get_generated_clocks`
- `get_lib_cells`

- get_lib_pins
- get_nets
- get_object_name
- get_pins
- get_ports
- get_property
- index_collection
- query_objects
- range_collection
- remove_from_collection
- remove_input_delay
- remove_output_delay
- reset_case_analysis
- reset_clock
- reset_clock_groups
- reset_clock_sense
- reset_disable_timing
- reset_generated_clock
- reset_input_delay
- reset_mode
- reset_output_delay
- reset_path_exception
- set_case_analysis
- set_clock_groups
- set_clock_sense

- set_disable_timings
- set_global
- set_input_delay
- set_mode
- set_multicycle_path
- set_output_delay
- set_sense
- size_of_collection
- sort_collection

List of Naming Style Configuration SDC Commands supported in CDC:

- hdl_array_naming_style
- hdl_bit_blasted_port_style
- hdl_bus_naming_style
- hdl_bus_wire_naming_style
- hdl_generate_index_style
- hdl_generate_separator
- hdl_instance_array_naming_style
- hdl_parameter_naming_style
- hdl_record_naming_style
- hdl_reg_naming_style
- hdl_rename_cdn_flop_pins