Vivado Design Suite User Guide

Using Constraints

UG903 (v2020.2) February 9, 2021

XILINX

The following table shows the revision history for this document.

Navigating Content by Design Process	Added section.
General Release Updates	N/A

Table of Contents

XILINX

Chapter 1

Introduction

The Xilinx® Vivado® Integrated Design Environment (IDE) uses Xilinx Design Constraints (XDC), and does not support the legacy User Constraints File (UCF) format.

There are key differences between Xilinx Design Constraints (XDC) and User Constraints File (UCF) constraints. XDC constraints are based on the standard Synopsys® Design Constraints (SDC) format. SDC has been in use and evolving for more than 20 years, making it the most popular and proven format for describing design constraints.

VIDEO: *For training on migrating UCF constraints to XDC, see the Vivado Design Suite QuickTake Video: Migrating UCF Constraints to XDC.*

If you are familiar with UCF but new to XDC, see the "Differences Between XDC and UCF Constraints" section in the Migrating UCF Constraints to XDC chapter of the *ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 1]. That chapter also describes how to convert existing UCF files to XDC as a starting point for creating XDC constraints.

IMPORTANT: *XDC has fundamental differences from UCF that must be understood in order to properly constrain a design. The UCF to XDC conversion utility is not a replacement for properly understanding and creating XDC constraints. Each XDC constraint is described in this User Guide.*

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:

- Dedicated Hardware Resources
- IP and Sub-Module Constraining with XDC

XDC constraints are a combination of industry standard Synopsys Design Constraints (SDC version 1.9) and Xilinx proprietary physical constraints.

XDC constraints have the following properties:

- They are not simple strings, but are commands that follow the Tcl semantic.
- They can be interpreted like any other Tcl command by the Vivado Tcl interpreter.
- They are read in and parsed sequentially the same as other Tcl commands.

You can enter XDC constraints in several ways, at different points in the flow.

- Store the constraints in one or more XDC files.

  To load the XDC file in memory, do one of the following:

  - Use the `read_xdc` command.
  - Add it to one of your project constraints sets. XDC files only accept the `set`, `list`, and `expr` built-in Tcl commands. See Appendix A, Supported XDC and SDC Commands for a complete list of supported commands.
- Generate the constraints with an unmanaged Tcl script.

  To execute the Tcl script, do one of the following:

  - Run the `source` command.

- Use the `read_xdc -unmanaged` command.
- Add the Tcl script to one of your project constraints sets.

TIP: *Unlike XDC files, unmanaged Tcl scripts can include any common Tcl command for selecting design objects and defining design constraints, including conditional and looping control structures.*

IMPORTANT: *The Vivado Design Suite allows you to mix XDC files and Tcl scripts in the same constraints set. Modified constraints are saved back to their original location only if they originally came from an XDC file, and not from an unmanaged Tcl script. A constraint generated by a Tcl script is not managed by the Vivado Design Suite and cannot be interactively modified. For more information, see Chapter 2, Constraints Methodology.*

IMPORTANT: *For XDC constraints, there is a difference in behavior between the commands `source` and `read_xdc`. The constraints imported with the `source` command are not saved in the checkpoint in the same order as they are imported. The constraints imported with `read_xdc` are saved first and then those imported with `source`. To save all the constraints in the same order as they are applied to the design, use `read_xdc -unmanaged` instead of `source`.*

To validate the syntax or impact of a particular constraint after loading your design in memory, use the Tcl console and the Vivado Design Suite reporting features. This is particularly powerful for analyzing and debugging timing constraints and physical constraints.

XILINX

Chapter 2

Constraints Methodology

Design constraints define the requirements that must be met by the compilation flow in order for the design to be functional on the board. Not all constraints are used by all steps in the compilation flow. For example, physical constraints are used only during the implementation steps (that is, by the placer and the router).

Because the Xilinx® Vivado® Integrated Design Environment (IDE) synthesis and implementation algorithms are timing-driven, you must create proper timing constraints. Over-constraining or under-constraining your design makes timing closure difficult. You must use reasonable constraints that correspond to your application requirements.

The Vivado IDE allows you to use one or many constraint files. While using a single constraint file for the entire compilation flow might seem more convenient, it can be a challenge to maintain all the constraints as the design becomes more complex. This is usually the case for designs that use several IP cores or large blocks developed by different teams.

After the timing and physical constraints have been imported, independent of the number of source files or whether the design is in Project or Non-Project mode, all the constraints can be exported as a single file with the `write_xdc` command. The constraints are written to the specified output file in the same order that they were read into the project or design. The command line option `write_xdc -type` can be used to select a subset of constraints (timing, physical, or waiver) to export.

RECOMMENDED: *Xilinx recommends that you separate timing constraints and physical constraints by saving them into two distinct files. You can also keep the constraints specific to a certain module in a separate file.*

You can add your Xilinx Design Constraints (XDC) files to a constraints set during the creation of a new project, or later, from the Vivado IDE menus.

Figure 2-1 shows two constraint sets in a project, which are single- or multi-XDC. The first constraint set includes two XDC files. The second constraint set uses only one XDC file containing all the constraints.

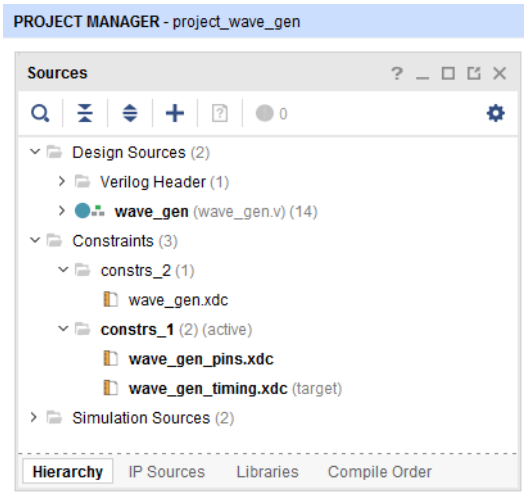


Figure 2-1:

IMPORTANT: *If your project contains an IP that uses its own constraints, the corresponding constraint file does not appear in the constraints set. Instead, it is listed along with the IP source files.*

You can also add Tcl scripts to your constraints set as unmanaged constraints or unmanaged Tcl scripts. The Vivado Design Suite does not write modified constraints back into an unmanaged Tcl script. Tcl scripts and XDC files are loaded in the same sequence as displayed in the Vivado IDE (if they belong to the same `PROCESSING_ORDER` group) or as reported by the command `report_compile_order -constraints`.

An XDC file or a Tcl script can be used in several constraints sets if needed. For more information on how to create and add constraint files and constraints sets to your project, see Working with Constraints in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 2].

In Non-Project Mode, you must read each file individually before executing the compilation commands.

The example script below shows how to use one or more XDC files for synthesis and implementation.

Example Script:

```
read_verilog [glob src/*.v]
read_xdc wave_gen_timing.xdc
read_xdc wave_gen_pins.xdc
synth_design -top wave_gen -part xc7k325tffg900-2
opt_design
place_design
route_design
```

In designs using Dynamic Function eXchange (DFX), it is common to synthesize parts of the design in an Out-of-Context (OOC) approach. When such a flow is used, some constraints can be specified for the OOC synthesis only. For example, clocks that propagate at the input boundary of the blocks must be defined when the blocks are synthesized OOC. These clocks are defined inside an OOC XDC file.

In Project Mode:

```
add_file constraints_ooc.xdc
set_property USED_IN {synthesis out_of_context} [get_files constraints_ooc.xdc]
```

The Out-of-Context can also be set on the XDC file through the GUI (property on file `constraints_ooc.xdc`).

In Non-Project Mode:

```
read_xdc -mode out_of_context constraints_ooc.xdc
```

By default, all XDC files and Tcl scripts added to a constraint set are used for both synthesis and implementation. Set the `USED_IN_SYNTHESIS` and `USED_IN_IMPLEMENTATION` properties on the XDC file or the Tcl script to change this behavior. This property can take the value of either TRUE or FALSE.

IMPORTANT: *The `DONT_TOUCH` attribute does not obey the properties of `USED_IN_SYNTHESIS` and `USED_IN_IMPLEMENTATION`. If you use `DONT_TOUCH` properties in the synthesis XDC, it is propagated to implementation regardless of the value of `USED_IN_IMPLEMENTATION`.*
For more information about the DONT_TOUCH attribute, refer to *RTL Attributes, page 59**.*

IMPORTANT: *If any module (IP/BD/...) is synthesized in Out-Of-Context (OOC) mode, the top-level synthesis run infers a black box for these modules. Hence, the top-level synthesis constraints will not be able to reference objects such as pins, nets, cells, etc., that are internal to the OOC module. If some top-level constraints refer to objects inside any OOC module, you may need to split the constraints into 2 files: one XDC file for Synthesis (USED_IN_SYNTHESIS=TRUE / USED_IN_IMPLEMENTATION=FALSE) and one XDC file for implementation (USED_IN_SYNTHESIS=FALSE / USED_IN_IMPLEMENTATION=TRUE). There is no such limitation during implementation since the netlists from the OOC module DCPs are linked with the netlist produced when synthesizing the top-level design files, and the Vivado Design Suite resolves the black boxes. The XDC output products that were generated for use during implementation are applied along with any user constraints.*

For example, to use a constraint file for implementation only:

1. Select the constraint file in the Sources window.
2. In the Source File Properties window:
   a. Uncheck **Synthesis**.
   b. Check **Implementation**.

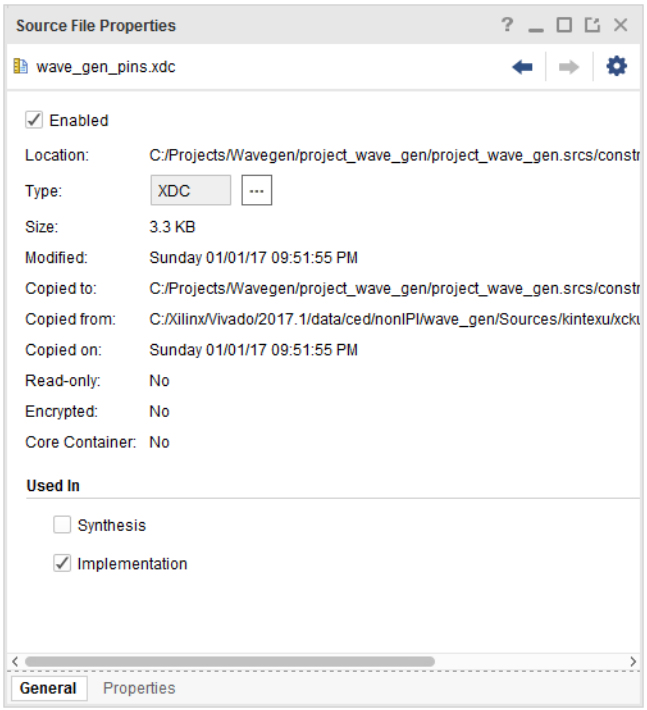


Figure 2-2:

The equivalent Tcl commands are:

```
set_property USED_IN_SYNTHESIS false [get_files wave_gen_pins.xdc]
set_property USED_IN_IMPLEMENTATION true [get_files wave_gen_pins.xdc]
```

When running Vivado in Non-Project Mode, you can read in the constraints directly between any steps of the flow. The properties `USED_IN_SYNTHESIS` and `USED_IN_IMPLEMENTATION` do not matter in this mode.

The following compilation Tcl script shows how to read two XDC files for different steps of the flow:

```
read_verilog [glob src/*.v]
read_xdc wave_gen_timing.xdc
synth_design -top wave_gen -part xc7k325tffg900-2
read_xdc wave_gen_pins.xdc
opt_design
place_design
route_design
```

Table 2-1:

`wave_gen_timing.xdc`	Before synthesis	• Synthesis • Implementation
`wave_gen_pins.xdc`	After synthesis	• Implementation

TIP: *The constraints read in after synthesis are applied in addition to the constraints read in before synthesis.*

Because XDC constraints are applied sequentially, and are prioritized based on clear precedence rules, you must review the order of your constraints carefully. For more information, see Chapter 7, XDC Precedence.

Note: If multiple physical constraints are conflicting, the latest constraint wins. For example, if an I/O port gets assigned a different location (LOC) through multiple XDC files, the latest location assigned to the port takes precedence.

The Vivado IDE provides full visibility into your design. To validate your constraints step by step:

1. Run the appropriate report commands.
2. Review the messages in the Tcl Console or the Messages window.

RECOMMENDED: *Whether you use one or several XDC files for your design, organize your constraints in the following sequence.*

```
## Timing Assertions Section
# Primary clocks
# Virtual clocks
# Generated clocks
# Clock Groups
# Bus Skew constraints
# Input and output delay constraints

## Timing Exceptions Section
# False Paths
# Max Delay / Min Delay
# Multicycle Paths
# Case Analysis
# Disable Timing

## Physical Constraints Section
# located anywhere in the file, preferably before or after the timing constraints
# or stored in a separate constraint file
```

Note: The case analysis constraints that change the clock relationships or clock propagation should be defined prior to defining the generated clocks. This includes the case analysis defined on clock buffers that result in the output clock of the buffer to be impacted by the case analysis.

Start with the clock definitions. The clocks must be created before they can be used by any subsequent constraints. Any reference to a clock before it has been declared results in an error and the corresponding constraint is ignored. This is true within an individual constraint file, as well as across all the XDC files (or Tcl scripts) in your design.

The order of the constraint files matters. You must be sure that the constraints in each file do not rely on the constraints of another file. If this is the case, you must read the file that contains the constraint dependencies last. If two constraint files have interdependencies, you must either merge them manually into one file that contains the proper sequence, or divide the files into several separate files and order them correctly.

The Vivado IDE constraints manager saves any edited constraint back to its original location in the XDC files, but not in Tcl scripts. Any new constraint is saved at the end of the XDC file marked as `target`. In many cases, when your constraints set contains several XDC files, the target constraint file is not the last file in the list, and will not be loaded last when opening or reloading your design. As a consequence, the constraints sequence saved to constraint source files can be different from the one you had previously in memory.

IMPORTANT: *You must verify that the final sequence stored in the constraint files still works as expected. If you must modify the sequence, you must modify it by directly editing the constraint files. This is especially important for timing constraints.*

In a project flow without any IP, all the constraints are located in a constraints set. By default, the order of the XDC files (or Tcl scripts) displayed in the Vivado IDE defines the read sequence used by the tool when loading an elaborated or synthesized design into memory. The file at the top of the list is read in first, and the bottom one is read in last. You can change the order by simply selecting the file in the IDE, and moving it to the desired place in the list.

For example, in Figure 2-3, the file `wave_gen_pin.xdc` was moved to before the file `wave_gen_timing.xdc` by using drag and drop.

Figure 2-3:

The equivalent Tcl command is:

```
reorder_files -fileset constrs_1 -before [get_files wave_gen_timing.xdc] \
  [get_files wave_gen_pins.xdc]
```

Table 2-2:

`wave_gen_timing.xdc`	1	2
`wave_gen_pins.xdc`	2	1

In Non-Project Mode, the sequence of the `read_xdc` calls determine the order in which the constraint files are evaluated.

Constraint Files Order with IP Cores

Many IP cores are delivered with one or more XDC files. When such IP cores are generated within your RTL project, their XDC files are also used during the various design compilation steps.

For example, Figure 2-4 shows that one of the IP cores in the project comes with an XDC file.

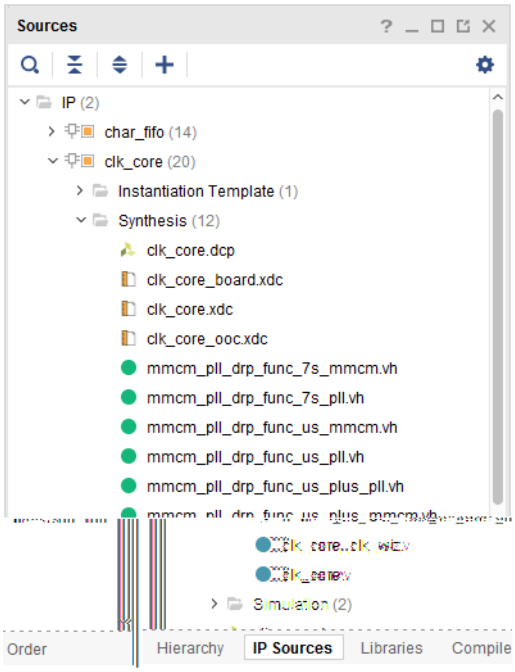


Figure 2-4:

By default, IP XDC files are read in before the user XDC files. Processing it in this way allows an IP to create a clock object that can be referenced in the XDC. It also allows you to overwrite physical constraints set by an IP core because the user constraints are evaluated after the IP. There is an exception to this order for the IP cores that have a dependency on clock objects being created by the user or by another IP (for example, `get_clocks -of_objects [get_ports clka]`). In this case, the IP XDC is read after the user files.

This behavior is controlled by the `PROCESSING_ORDER` property, set for each XDC file:

- EARLY: Files that must be read first
- NORMAL: Default
- LATE: Files that must be read last

An IP XDC will have its `PROCESSING_ORDER` property set to either EARLY or LATE. No IP delivers XDC files that belong to the NORMAL constraints group. For user XDC (or Tcl) files that belong to the same PROCESSING_ORDER group, their relative order displayed in the Vivado IDE determines their read sequence. The order within the group can be modified by moving the files in the Vivado IDE constraints set, or by using the `reorder_files` command.

For IP XDC files that belong to the same PROCESSING_ORDER group, the order is determined by import or creation sequence of the IP cores. This order cannot be changed after the project has been created.

Finally, the relative order between user groups and IP XDC PROCESSING_ORDER groups are as follows:

1. User Constraints marked as EARLY
2. IP Constraints marked as EARLY (default)
3. User Constraints marked as NORMAL
4. IP Constraints marked as LATE (contain clock dependencies)
5. User Constraints marked as LATE

Note: IP XDC files that have their `PROCESSING_ORDER` set to LATE (in order to be processed after the user constraints) are named `<IP_NAME>_clocks.xdc`.

The following figure shows an example of how to set the `PROCESSING_ORDER` property:

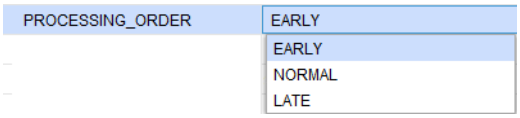


Figure 2-5:

The equivalent Tcl command is:

```
set_property PROCESSING_ORDER EARLY [get_files wave_gen_pins.xdc]
```

RECOMMENDED: *Use the `report_compile_order -constraints` command in the Tcl console to report the XDC files read sequence determined by the tool based the properties mentioned above, including IS_ENABLED, USED_IN_SYNTHESIS, and USED_IN_IMPLEMENTATION.*

Note: When an IP is synthesized Out of Context, the IP provides, when needed, an _occ.xdc file which contains the default clock definition. The _ooc.xdc has the USED_IN property set to "synthesis out_of_context implementation" (order does not matter). During the Out Of Context synthesis, the _occ file is always processed before all other constraints.

To change the read order of an XDC file or unmanaged Tcl script in a constraints set:

1. In the Sources window, select the XDC file or Tcl script you want to move.
2. Drag and drop the file to the desired position in the constraints set.

For the example shown in Figure 2-3, the equivalent Tcl command is:

```
reorder_files -fileset constrs_1 -before [get_files wave_gen_timing.xdc] \
  [get_files wave_gen_pins.xdc]
```

In Non-Project Mode, the sequence of the `read_xdc` or `source` commands determines the order the constraint files are read.

If you use an IP core that comes with constraints, two groups of constraints are handled automatically as follows:

- Constraints that do not depend on clocks are grouped in an XDC file with PROCESSING_ORDER set to EARLY,
- Constraints that depend on clocks are grouped in an XDC file with PROCESSING_ORDER set to LATE.

By default, user XDC files belong to the PROCESSING_ORDER NORMAL group. They are loaded after EARLY XDC files and before LATE XDC files. For each PROCESSING_ORDER group, IP XDC files are loaded in the same sequence as how the IP cores are listed in the IP Sources window. For example, the following figure shows one of the project IP cores that comes with an XDC file.

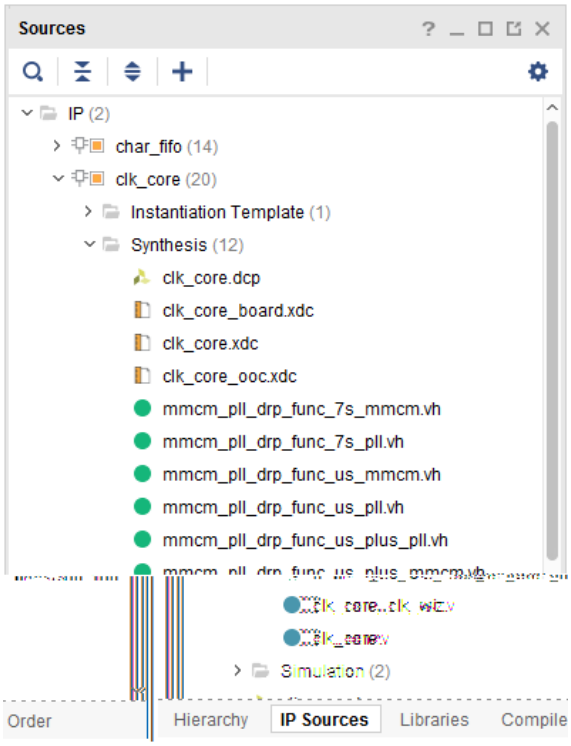


Figure 2-6:

When you open your design, the log file shows that the IP XDC file was loaded last:

```
Parsing XDC File [C:/project_wave_gen_hdl.srcs/sources_1/ip/clk_core/clk_core.xdc]
for cell 'clk_gen_i0/clk_core_i0/inst'
Finished Parsing XDC File
[C:/project_wave_gen_hdl.srcs/sources_1/ip/clk_core/clk_core.xdc] for cell
'clk_gen_i0/clk_core_i0/inst'
Parsing XDC File
[C:/project_wave_gen_hdl.srcs/sources_1/ip/char_fifo/char_fifo/char_fifo.xdc] for
cell 'char_fifo_i0/U0'
```

```
Finished Parsing XDC File
[C:/project_wave_gen_hdl.srcs/sources_1/ip/char_fifo/char_fifo/char_fifo.xdc] for
cell 'char_fifo_i0/U0'
Parsing XDC File
[C:/project_wave_gen_hdl.srcs/constrs_1/imports/verilog/wave_gen_timing.xdc]
Finished Parsing XDC File
[C:/project_wave_gen_hdl.srcs/constrs_1/imports/verilog/wave_gen_timing.xdc]
Parsing XDC File
[C:/project_wave_gen_hdl.srcs/sources_1/ip/char_fifo/char_fifo/char_fifo_clocks.xdc
] for cell 'char_fifo_i0/U0'
Finished Parsing XDC File
[C:/project_wave_gen_hdl.srcs/sources_1/ip/char_fifo/char_fifo/char_fifo_clocks.xdc
] for cell 'char_fifo_i0/U0'
Completed Processing XDC Constraints
```

Unlike with the User XDC files, you cannot directly change the read order of the IP XDC files that belong to the same PROCESSING_ORDER group. If you must modify the order, do the following:

1. Disable the corresponding IP XDC files (IS_ENABLED set to false).
2. Copy their content.
3. Paste the content into one of the XDC files included in your constraints set.
4. Update the copied IP XDC commands with the full hierarchical netlist object path names wherever needed. Doing so is required because the IP XDC constraints are written in such a manner that they can be scoped to the IP instance.
5. Review the `get_ports` queries that are processed in a special way for scoped constraints. For more information on XDC scoping, see Constraints Scoping, page 68.

The Vivado IDE provides several ways to enter constraints. Unless you directly edit the XDC file in a text editor, you must open a design database (elaborated, synthesized or implemented) in order to access the constraints windows in the Vivado IDE.

You must have a design in memory to validate your constraints during editing. When you edit a constraint using the Vivado IDE user interface, the equivalent XDC command is issued in the Tcl Console in order to apply it in memory. An edited timing constraint must be applied in memory before it can be saved to the XDC file.

Before you can run synthesis or implementation, you must save the constraints in memory back to an XDC file that belongs to the project. The Vivado IDE prompts you to save your constraints whenever necessary.

Do one of the following to save your constraints manually:

- Click **Save Constraints**.
- Select **File** > **Constraints** > **Save**.

Note: When you save the in-memory constraints, a dialog box opens to remind you that this could cause the synthesis and implementation to go out of date. Select the **Remember Preference** check box on this dialog box to disable future instances of this warning.

When you run these commands, Vivado does the following:

- Saves all new constraints to the XDC file marked `target` in the constraints set associated with your design.
- Saves all edited constraints back to the XDC file from which they originated.

  Note: The constraints management system preserves the original XDC files format as much as possible.

Figure 2-7 shows the recommended flow options. Do not use both options at the same time. Mixing these options might cause you to lose constraints. The recommended flow options are:

- User Interface Option
- Hand Edit Option

User Interface Option

Because the Vivado IDE manages your constraints, you must not edit your XDC files at the same time. When the Vivado IDE saves the memory content, the following occurs:

- The modified constraints replace the original constraints in their original file.
- The new constraints are appended to the file marked as `target`.
- All manual edits in the XDC files are overwritten.

Hand Edit Option

When you use the Hand Edit option, you are in charge of editing and maintaining the XDC files. While you will probably use the Tcl Console to verify the syntax of some constraints, you must discard the changes made in memory when closing or reloading your design.

In case of a conflict when saving the constraints, you are prompted to choose one of the following:

- Discarding the changes made in memory
- Saving the changes in a new file
- Overwriting the XDC files

Constraints creation is iterative. You can use IDE editors in some cases, and hand edit the constraint files in others.

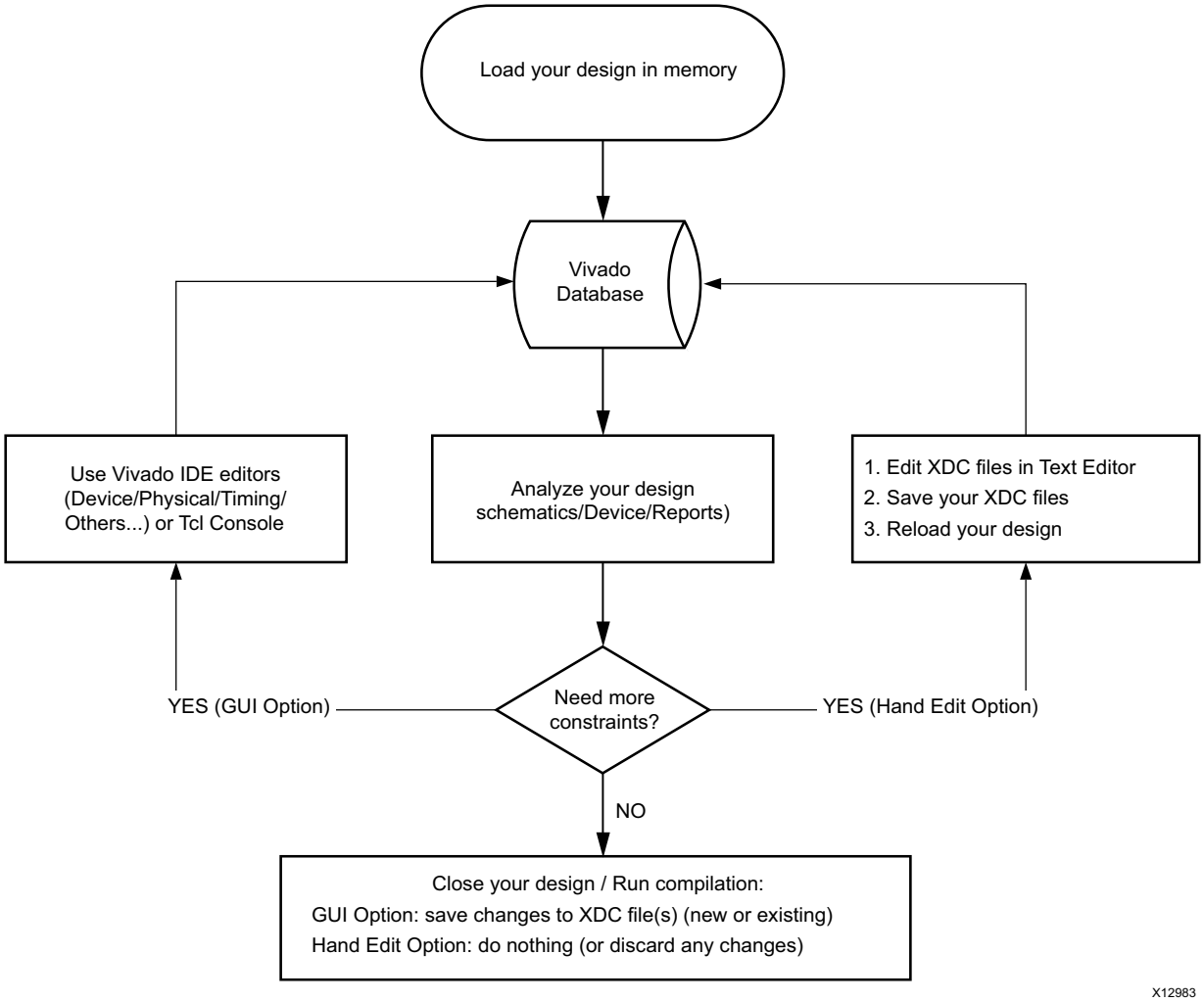


Figure 2-7:

Within each iteration described in Figure 2-7, do not use both options at the same time.

If you switch between the two options, you must first save your constraints or reload your design, to ensure that the constraints in memory are properly synchronized with the XDC files.

To create and edit existing top-level ports placement when using the RTL Analysis, Synthesis, or Implementation views:

1. Select the I/O Planning pre-configured layout.

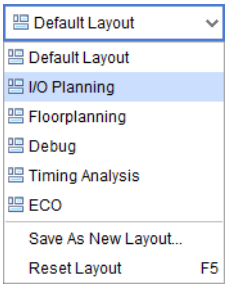


Figure 2-8:

2. Open the windows shown in Table 2-3.

Table 2-3:

Device	View and edit the location of the ports on the device floorplan.
Package	View and edit the location of the ports on the device package.
I/O Ports	Select a port, drag and drop it to a location on the Device or Package view, as well as review current assignment and properties of each port.
Package Pins	View the resource utilization in each I/O bank.

For more information on Pin Assignment, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 3].

To create and edit Pblocks when using the RTL Analysis, Synthesis, or Implementation views:

1. Select the Floorplanning pre-configured layout.

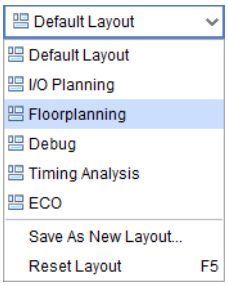


Figure 2-9:

2. Open the windows shown in Table 2-4.

Table 2-4:

Netlist	Select the cells to be assigned to a Pblock.
Physical Constraints	Review the existing Pblocks and their properties.
Device	Create or edit the shape and location of your Pblocks in the device.

To create cell placement constraints on a particular BEL or SITE:

1. Select the cell in the Netlist view.
2. Drag and drop the cell to the target location in the Device view.

For more information on Floorplanning, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

The Timing Constraints Wizard identifies missing timing constraints on a synthesized or implemented design. It analyzes the netlist, the clock nets connectivity, and the existing timing constraints in order to provide recommendations as per the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 5]. Three categories of constraints are covered by the following 11 pages of the wizard, followed by a summary. The following steps are included:

- Clocks
  - Primary clocks
  - Generated clocks
  - Forwarded clocks
  - External feedback delays
- Input and output ports
  - Input delays
  - Output delays
  - Combinatorial delays
- Clock domain crossings
  - Physically exclusive clock groups
  - Logically exclusive clock groups with no interaction
  - Logically exclusive clock groups with interaction
  - Asynchronous clock domain crossings
- Constraints Summary

During each step, you can accept the recommended constraints or modify the list by checking or unchecking each of the proposed constraints. However, unchecking recommended constraints early in the wizard can prevent the identification of other missing constraints in subsequent steps. For example, if you decide to skip the creation of a clock, the wizard will not identify and recommend any constraints that refer to this clock or its auto-derived clocks.

The final page of the wizard provides a summary of the constraints that will be created. You can click on each individual hyperlink to see the constraints details, or visualize the new constraints in the Timing Constraints window after exiting the wizard.

You can also choose to generate the following recommended reports upon clicking **Finish** to verify that the design is completely and properly constrained:

- **Create Timing Summary report**: Timing slack is reported with the new constraints, in addition to a `check_timing` report. Timing violations will likely display if the period or I/O delay constraints that you entered are too difficult.
- **Create Check Timing report**: This report identifies missing or inappropriate constraints by running the `check_timing` command.
- **Create DRC Report using only Timing Checks**: this report runs the Timing DRCs.

IMPORTANT: *The newly added constraints are automatically saved to the Target XDC file unless you click **Cancel**. You can edit or delete the new constraints in the Timing Constraints window after exiting the wizard.*

The Timing Constraint Wizard does not recommend a constraint if it introduces unsafe timing analysis. Also, the wizard does not fix inappropriate constraints that already existed when loading the design in memory. Nevertheless, some invalid constraints might become valid after creating all the missing clocks when using Vivado Design Suite in project mode; for more details, see Constraints Processing Order and Invalid Constraints, below. Also, after using the wizard, if `check_timing` or `report_drc` still flag some constraints issues, it is usually due to a constraint problem that already existed in the source XDC files. You must address these problems directly instead of using the wizard to resolve them.

clocks, then re-applies the invalid constraints from `c.xdc` before proceeding to the next step and discovering other missing constraints.

- **Non-project or Design Check Point (DCP) modes**: You cannot specify a target XDC file in these modes, so the Timing Constraints wizard recommends and applies new constraints at the last position of the constraints sequence. This is equivalent to entering new constraints in the Tcl Console or via the Timing Constraints window. In these modes, the wizard does not attempt to re-apply invalid constraints. If the new constraints need to be applied earlier in the overall constraints sequence in order to resolve constraints dependencies or precedence issues, you must edit the constraints sequence manually.

  Here is an example of how to manually edit constraints.

  a. Create new constraints using the Vivado Design Suite.

  b. Run one of the following commands:

     ```
     write_xdc -exclude_physical timing_constraints.xdc
     ```

     ```
     write_xdc -type timing timing_constraints.xdc
     ```

  c. Edit `timing_constraints.xdc` to move the new constraints higher in the XDC file.

  d. Save the file.

  e. Run the following command:

     ```
     reset_timing
     ```

  f. Read the edited timing constraints file by typing:

     ```
     read_xdc timing_constraints.xdc
     ```

You can review the updated timing constraints sequence using the Timing Constraints window. After reviewing the new constraints, you can save the sequence into the DCP.

Reporting Features Available When the Wizard is Open

When the Timing Constraints wizard is open, it prevents most actions in the Vivado IDE, including using the Tcl Console or running timing analysis, in order to avoid database discrepancies. The wizard window is always in front of the other Vivado IDE windows. If you need to access the Vivado IDE menus or windows, you must move the wizard window to the side.

Only the following features are available while the Timing Constraints wizard is open:

- Reporting and visualizing the clock networks

Most pages of the wizard have buttons to generate and access the clock network report in order to visualize the clock topologies, their source point, and the shared segments for some of the clocks.

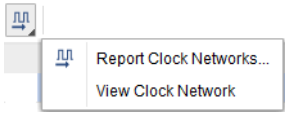


Figure 2-10:

Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4] for more details about the clock network report.

- Searching a name in source files or an object in the design in memory

  The Find and Find In Files dialog boxes are available from the **Edit** menu. You can use these dialog boxes to retrieve some information about the design while entering the constraints in the wizard.

- Creating and Viewing schematics

  You can select design objects in the main Vivado IDE window and visualize them in schematics. All schematics features are available. Only the last step of the Timing Constraints wizard, Asynchronous Clock Domain Crossings, supports convenient schematics cross-probing when selecting one or several entries in the Timing Paths tab.

  Refer to the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 7] for more info on using schematics.

- Visualizing constraints in memory with the Timing Constraints window

  Each page of the wizard includes a tab that shows the existing constraints of the same type as recommended by the step. This is convenient for quickly reviewing the details of constraints already created in the XDC files. For a complete view of all timing constraints in memory, the Timing Constraints window shows the full sequence of constraints, organized by XDC file, including scoping information. It also displays the invalid constraints.

Constraints Editing within the Wizard

Each step of the wizard can recommend several constraints. Depending on the constraint, you must take one of the following actions:

- Uncheck the constraints you do not want to create, using one of the following methods:
  - Remove each constraint from the list, one at a time, by unchecking each line.
  - Remove all constraints by unchecking the upper left check box of the table.

TIP: *Alternatively, you can right-click the constraint, and select Do Not Create Constraint, as shown in Figure 2-11.*

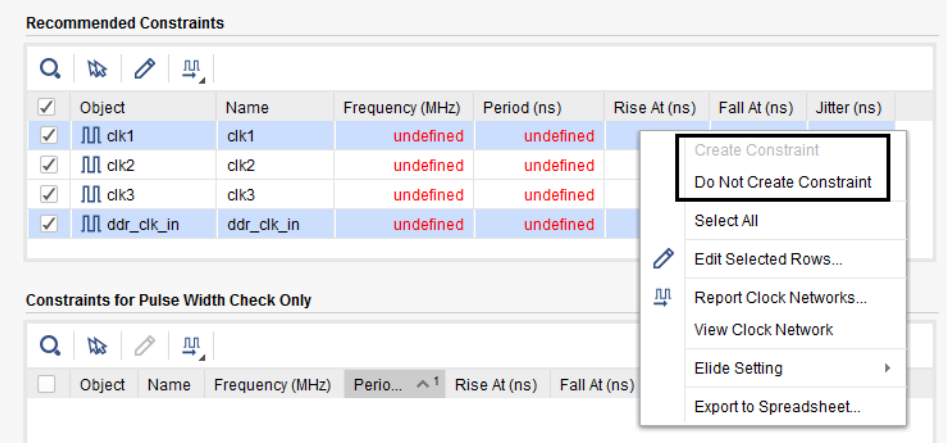


Figure 2-11:

In Figure 2-12, `clk1` and `ddr_clk_in` are unchecked and will be skipped.

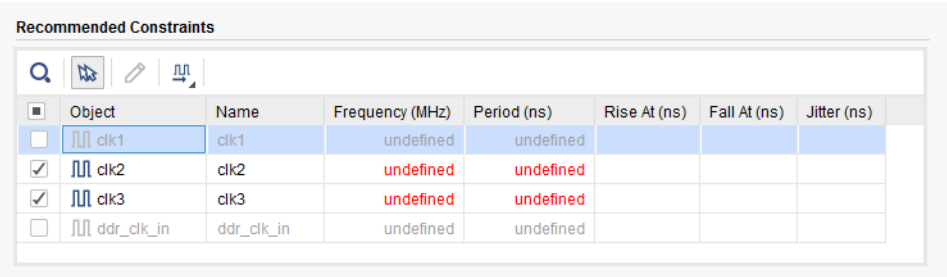


Figure 2-12:

- Enter the missing values by clicking on the cells that show **undefined** (for example, the **Frequency** or **Period** value for `clk2` and `clk3` in Figure 2-12).

  You can edit several constraints at the same time by selecting the corresponding rows and clicking the **Edit Selected Rows** button, as shown in Figure 2-13.

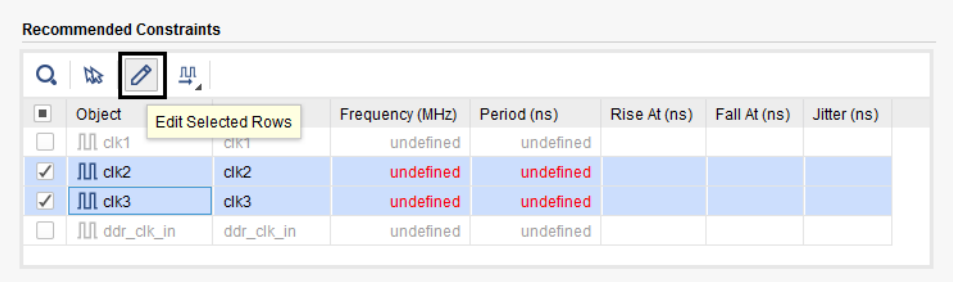


Figure 2-13:

Next, fill out any required fields, such as **Frequency** and **Period** as shown in Figure 2-14.

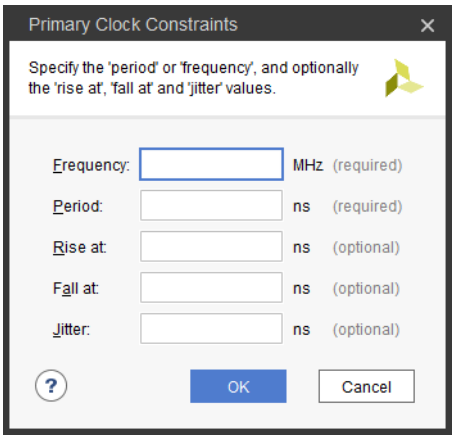


Figure 2-14:

Editing multiple constraints at a time is particularly helpful for input and output delay constraints.

- Simply review the constraints if no action is required.

When all the checked recommended constraints have been reviewed and completed, click **Next** to proceed to the next page. Any entries that you missed prevent the wizard from moving to the next step.

You can use the **Back** button to revisit a page. If you edit any constraint on a previous page and click **Next**, the wizard re-analyzes the design and recommends new constraints accordingly. In most cases, the previously recommended constraints not affected by the change are reinstated. If you only view a previous page without modifying any of its recommended constraints, the wizard does not re-run any analysis, which usually saves runtime.

IMPORTANT: *You cannot use the Timing Constraints wizard to edit existing timing constraints. Instead, you must use the Timing Constraints window.*

Constraints Recommended by the Wizard

Two categories of clocks are identified by the wizard, as shown in Figure 2-15.

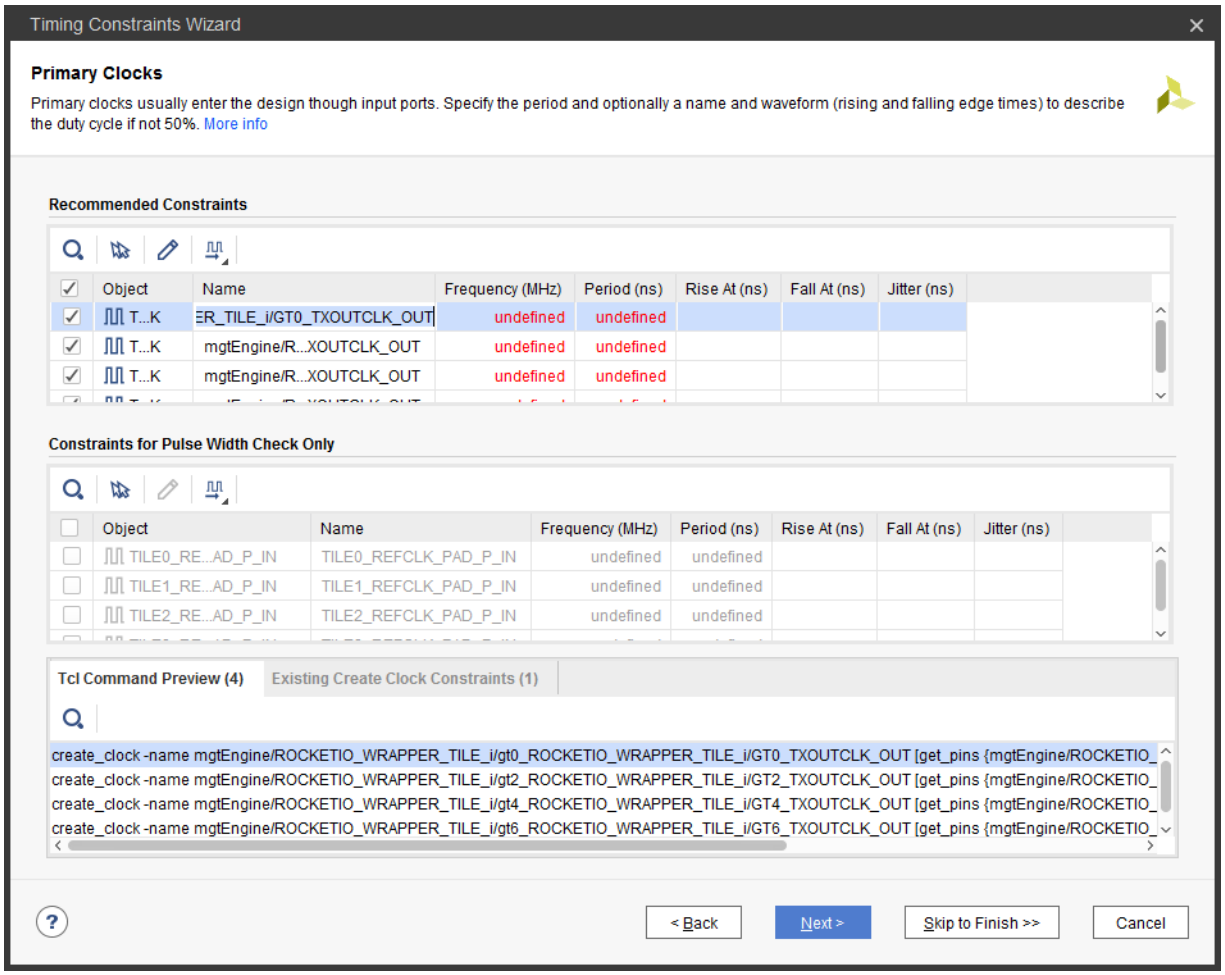


Figure 2-15:

- The primary clocks needed for computing the timing slack for setup, hold, recovery, and removal checks appear in the **Recommended Constraints** table.
- The clocks only needed for performing pulse width checks (`min_period`, `max_period`, `max_skew`, `min_low_pulse_width`, and `min_high_pulse_width`) appear in the **Constraints For Pulse Width Check Only** table. By default, these clocks are unchecked because they are only used for reporting purposes and do not influence the implementation tools quality of result.

The wizard automatically identifies the proper clock source point for the constraint. In most cases, the clock source point is an input clock port, and in some special cases it is the output of a primitive that does not have a timing arc. For example, in 7 series devices, the

wizard identifies missing primary clocks on the output of GT_CHANNEL primitives. For UltraScale™ devices, the Vivado Design Suite is able to auto-derive the GT_CHANNEL output clocks based on the incoming clock characteristics and the GT_CHANNEL configuration and connectivity. Consequently, the wizard recommends primary clocks located upstream from the GT_CHANNEL cells on the design boundary.

The Timing Constraints wizard recommends the creation of a generated clock on the output of a sequential cell when it drives the clock pins of other sequential cells either directly or through some interconnect logic. Unlike PLL or MMCM, user logic cannot multiply the frequency of the master clock, so the wizard only offers the option to specify a division coefficient, as shown in Figure 2-16.

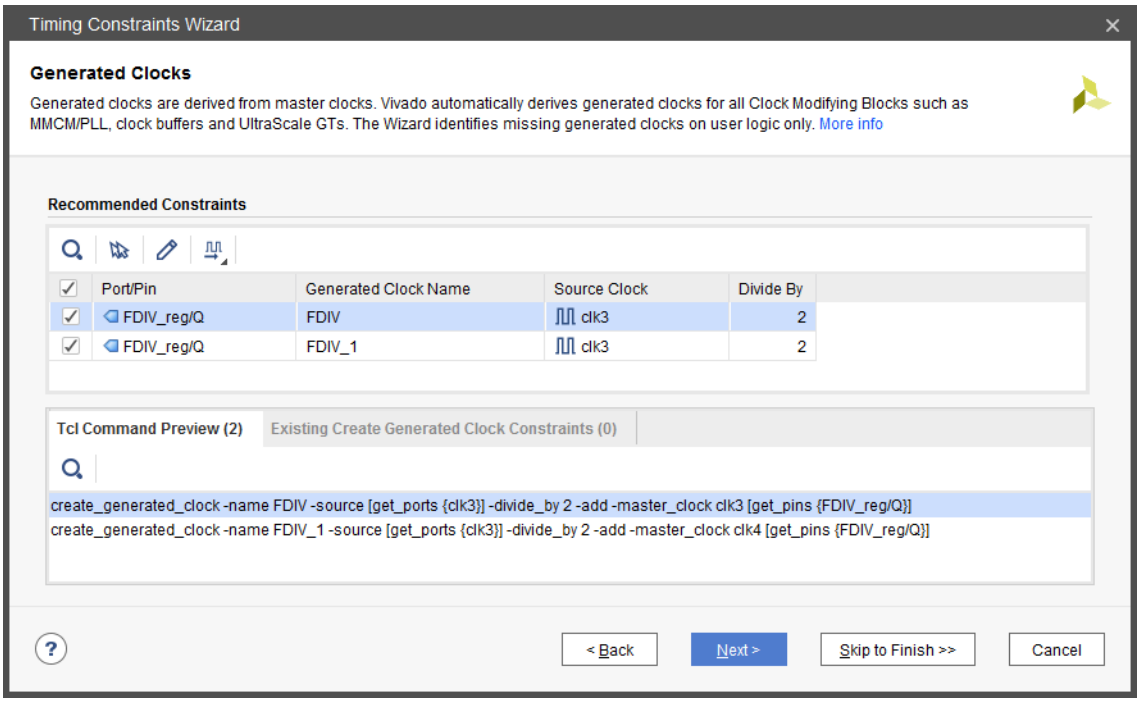


Figure 2-16:

When several master clocks reach the generated clock source point, the wizard creates all the corresponding generated clocks, using unique names and clear reference to individual master clocks. Figure 2-16 illustrates the scenario where two clocks (clk3 and clk4) reach the sequential cell FDIV_reg. Consequently, two generated clock constraints (FDIV and FDIV_1) are recommended.

TIP: *Some clocking topologies, such as cascaded registers on the clock path, might require that you run the Timing Constraints wizard multiple times to discover all the missing generated clocks.*

The Timing Constraints wizard recommends generated clock constraints on output ports that are driven by double data-rate registers with constant inputs. Based on the input constant connectivity, the generated clock phase is adjusted to either positive (0 degree phase shift) or inverted (180 degree phase shift). The master clock used in the constraint is the clock that reaches the clock pin of the double data-rate register. See the Source Clock column of the Recommended Constraints table in Figure 2-17.

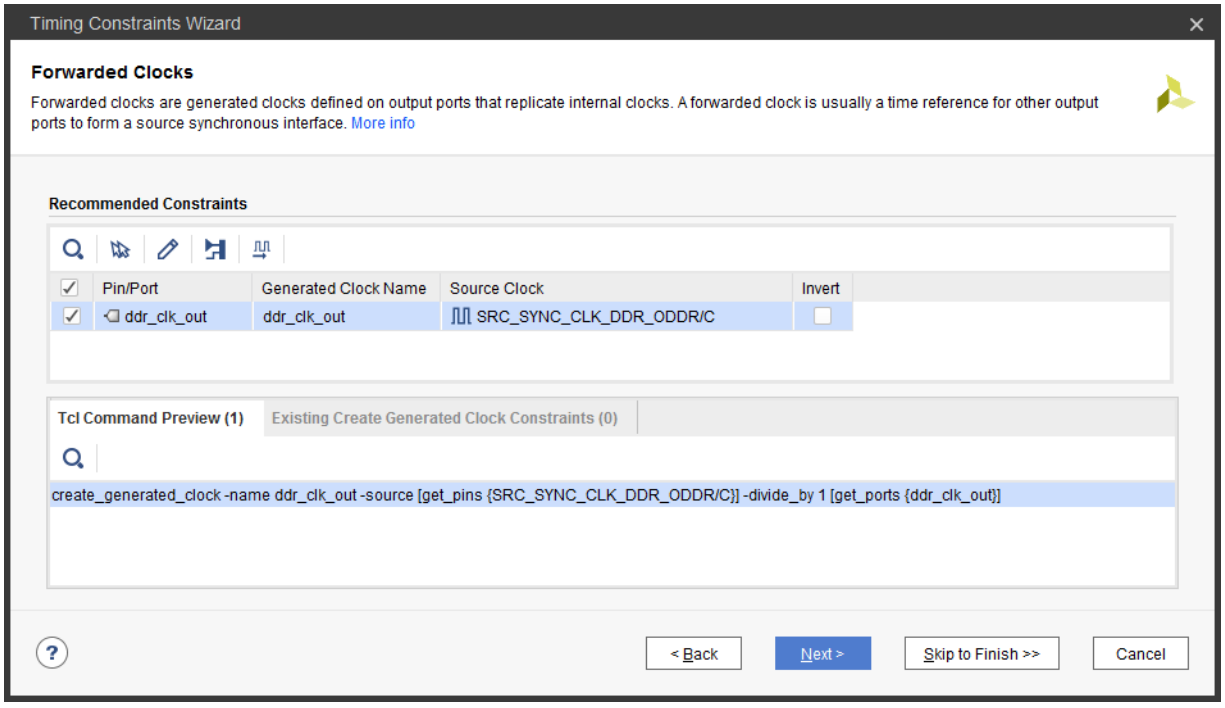


Figure 2-17:

For the 7 series device family, the topology recognized by the wizard is shown in Figure 2-18. There is no restriction on the nature of the master clock or the output buffer.

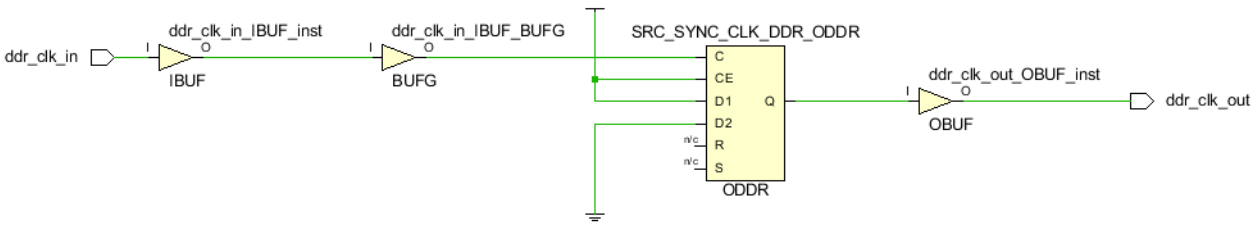


Figure 2-18:

For the UltraScale device family, the ODDR and ODDRE1 primitives are automatically retargeted to OSERDESE3 with the property `ODDR_MODE=TRUE`. The wizard recognizes the topology shown in Figure 2-19, where `OSERDESE3/D[0]` is connected to `1` and `OSERDESE3/D[4]` is connected to `0` (no phase-shift).

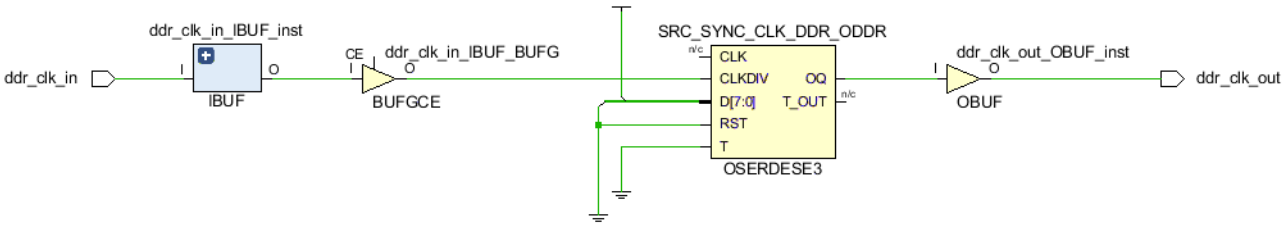


Figure 2-19:

The Timing Constraints wizard analyzes the feedback loop connectivity of the MMCM and PLL cells present in the design. External delay constraints (min and max) are recommended when the `CLKFBIN` and `CLKFBOUT` pins are connected to the design ports through IO buffers and the MMCM or PLL property `COMPENSATION=EXTERNAL`. Figure 2-20 illustrates the recommended External Delay constraints.

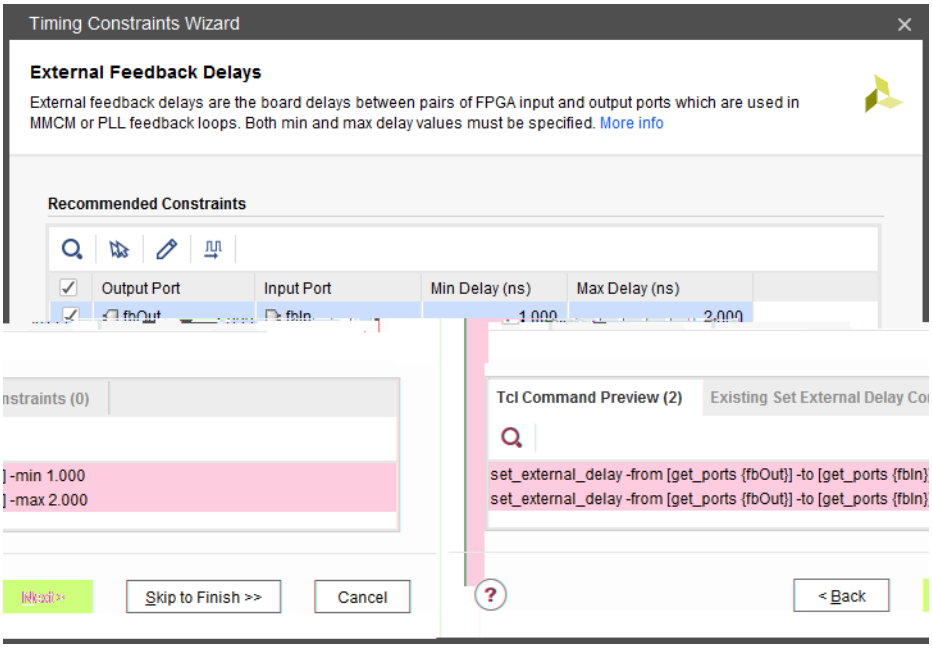


Figure 2-20:

Figure 2-21 illustrates a typical MMCM with external feedback path circuit.

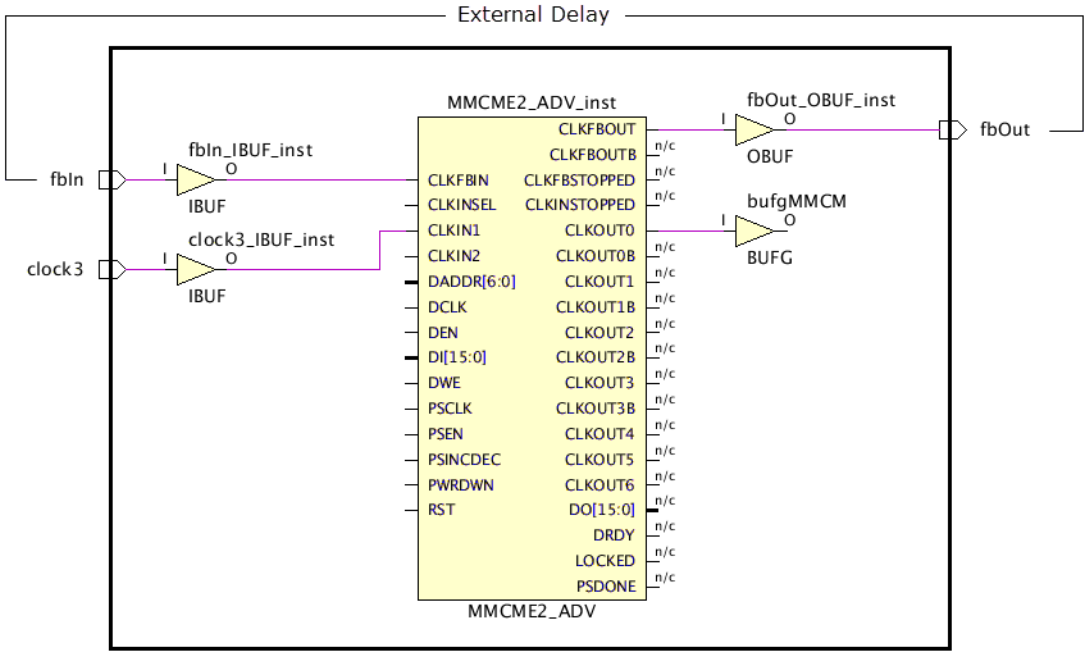


Figure 2-21:

In the current Vivado Design Suite release, the Timing Constraints wizard cannot recommend external delay constraints when there is a sequential cell in the feedback path, such as ODDR, which is used for generating a forwarded clock. In this case, you must create the external delay constraints manually or using the Timing Constraints window after exiting the wizard.

The Timing Constraints wizard analyzes all paths from input ports to identify their destination clock inside the design and their active edges. Based on this information, the wizard recommends basic system synchronous input delay constraints that are based on the XDC templates available in the Vivado IDE (see XDC Templates, page 56 for templates). The waveform associated with the selected template is displayed at the bottom of the window in the Waveform tab when you select a constraint entry in the Recommended Constraints table.

Figure 2-22 shows an example of several input constraints proposed by the wizard and partially edited by the user.

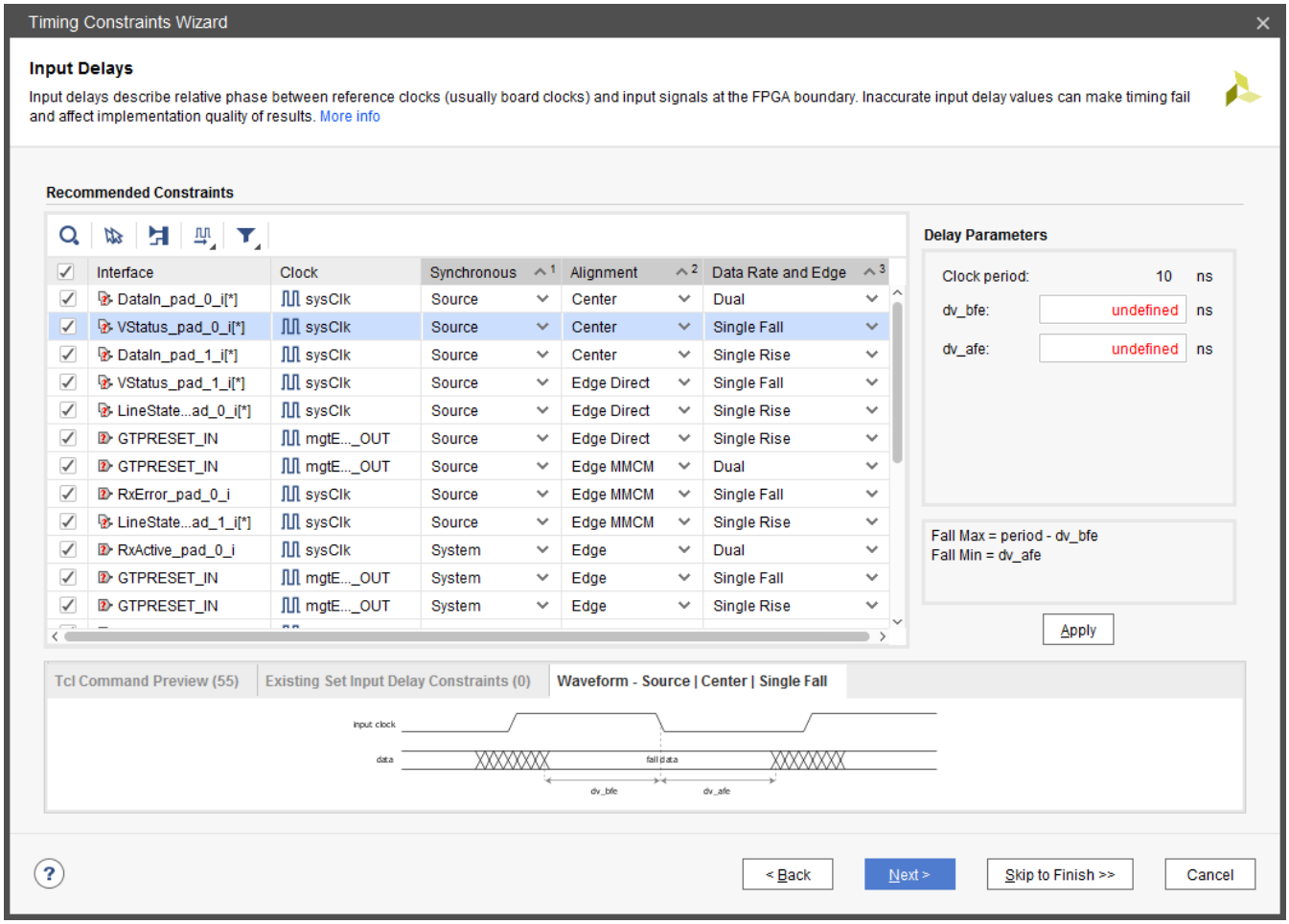


Figure 2-22:

For each constraint, you can edit three characteristics in order to specify the appropriate waveform that corresponds to the actual interface timing on the board:

- **Synchronous** describes the nature of the clock-data relationship.
  - **System** (for System Synchronous interface): use this setting when the data is launched and captured by different clock edges that are 1 period or ½ period apart.
  - **Source** (for Source Synchronous interface): use this setting when the data is launched and captured by the same clock edge.
- **Alignment** describes the data transition alignment with respect to the active clock edge.
  - For System Synchronous interfaces only:
    - **Edge**: use this setting when the clock and data transition at the same time.

    - For Source Synchronous interfaces only:
        - **Center**: use this setting when the clock transitions in the middle of the data valid window.
        - **Edge Direct**: use this setting when the clock transitions at the beginning of the data valid window.
        - **Edge MMCM**: use this setting when the clock transitions at the end of the data valid window.
- **Data Rate and Edge** describes the active clock edges constrained by the template. The default value recommended by the wizard is based on the active clock edges of the capturing sequential cell.
    - **Single Rise**: use this setting for cases where only the rising clock edges launch the data outside the FPGA.
    - **Single Fall**: use this setting for cases where only the falling clock edges launch the data outside the FPGA.
    - **Dual**: use this setting for cases where both rising and falling clock edges launch the data outside the FPGA.

The recommended clock is usually the board clock related to the input path sequential cell. When the input path internal clock is an MMCM or PLL generated clock, the board clock that drives the MMCM or PLL is used as the input constraint reference clock. The only exceptions exist when the internal clock waveform and the board clock waveform are not identical, such as the following scenarios:

- Different period scenario

  The input constraint references a virtual clock that has the same waveform as the internal clock so that the setup analysis is performed with a 1 cycle path requirement. The virtual clock is automatically created.

- Positive phase-shift clock scenario

  The wizard uses a virtual clock as the reference clock. The virtual clock is automatically created with the same waveform as the board clock. In addition, the wizard also specifies a multicycle path constraint between the virtual clock and the internal clock to adjust the default analysis to 1 period + the amount of phase-shift for setup. The combination of the virtual clock and the multicycle path constraint provides simpler constraints for the Vivado Design Suite timer to handle and can only affect input ports that reference to the virtual clock.

  Note that for a negative phase-shift, the virtual clock and the multicycle path constraint are not needed because the default setup path requirement is 1-cycle minus the amount of phase-shift.

The wizard does not allow you to change the reference clock selected for the constraint. To do so, you must manually edit the XDC files or use the Timing Constraints window after exiting the wizard.

After you select the proper template, enter the delay parameter values in the Delay Parameters panel located on the right hand side of the wizard and then click **Apply** to validate the entries.

The input delay equations are displayed below the delay parameter fields and on some of the template waveforms. Figure 2-23 shows the Delay Parameters panel for the DDR System Synchronous interface template.

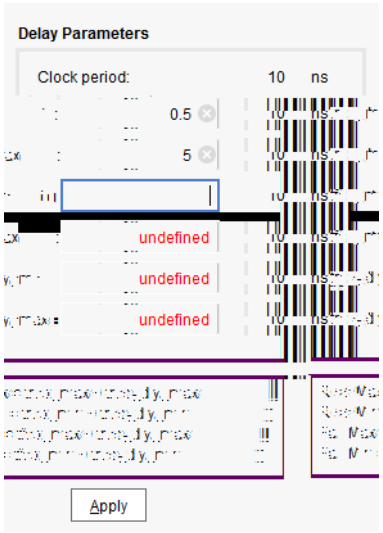


Figure 2-23:

To accelerate the delay parameter entry task, you can select and edit several constraints with same clock and same template at once.

After the constraints have been completed and applied, you can review their corresponding Tcl syntax in the Tcl Command Preview tab or you can click **Next** to proceed to the next step.

TIP: *The Timing Constraints wizard skips input ports with a false path constraint. This is particularly useful for skipping asynchronous resets that usually do not have a known phase relationship with any clock of the design. The false path constraint can only be created outside the wizard.*

Similar to the Input delays step, the Timing Constraints wizard analyzes the paths to all output ports to identify their source clocks inside the design and their active edges. The template selection rules are the same as described in Input Delays. Figure 2-24 shows several output constraints proposed by the wizard and partially edited by the user.

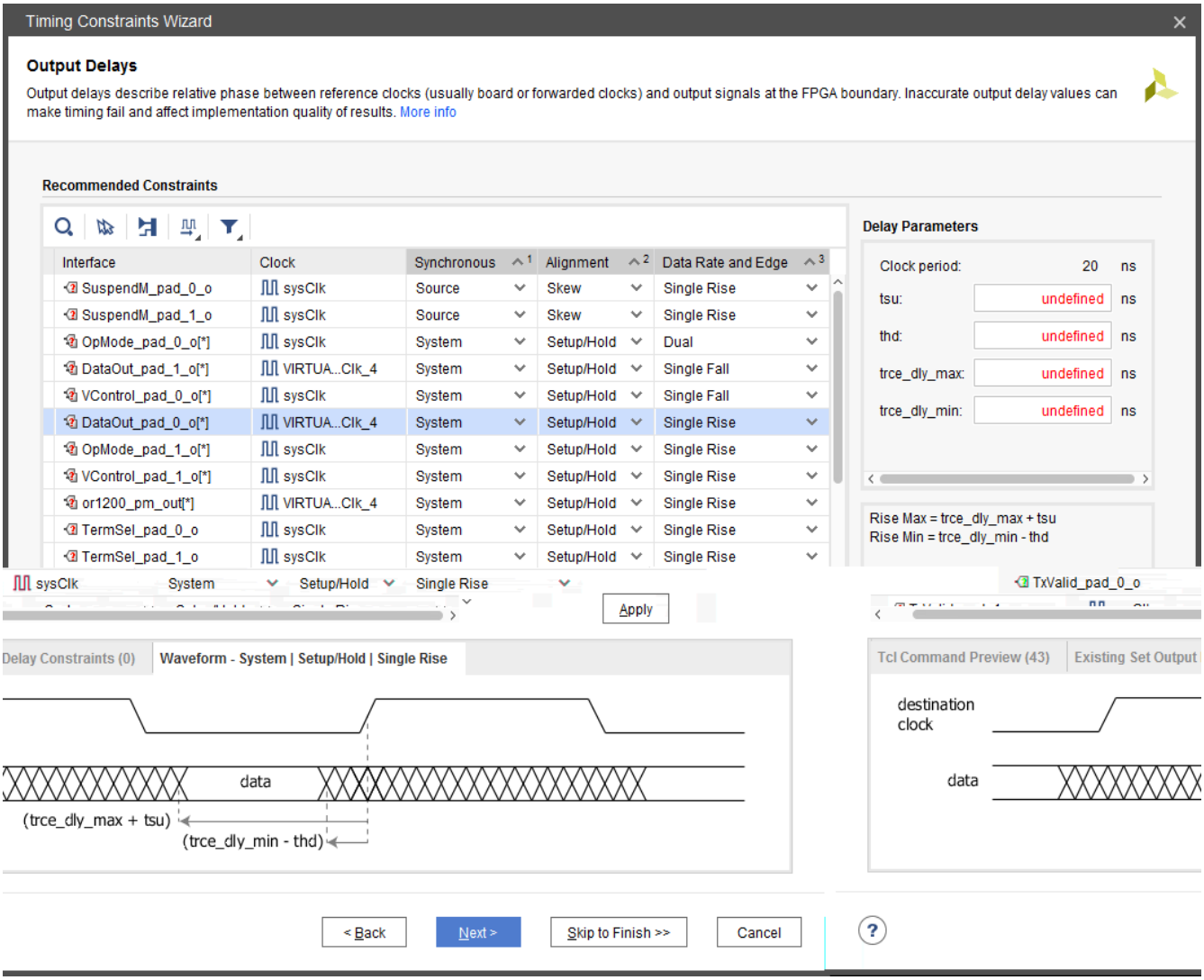


Figure 2-24:

For each constraint, three characteristics can be edited in order specify the appropriate waveform that corresponds to the actual interface timing on the board:

- **Synchronous** describes the nature of the clock-data relationship (see Input Delays, page 34 for more details).
- **Alignment** describes the data transition alignment with respect to the active clock edge.
  - **Setup/Hold**: use this setting when the template delay parameters are specified based on the data valid window timing characteristics outside the FPGA.
  - **Skew** (Source Synchronous only): use this setting when the template delay parameters are specified based on the skew requirements on the output pin of the FPGA.
- **Data Rate and Edge** describes the active clock edges constrained by the template (see Input Delays, page 34 for more details).

As with recommended input delay constraints, the reference clock is typically the board clock, except in the following cases:

- The board clock and the output path internal clock have different clock periods.

  The output constraint references a virtual clock that has the same waveform as the internal clock so that the setup analysis is performed with a 1-cycle path requirement. The virtual clock is automatically created.

- The output path internal clock has a negative phase-shift compared to the board clock.

  The wizard uses a virtual clock as the reference clock. The virtual clock is automatically created with the same waveform as the board clock. In addition, the wizard also specifies a multicycle path constraint between the virtual clock and the internal clock to adjust the default analysis to 1 period + the amount of phase-shift for setup. The combination of the virtual clock and the multicycle path constraint provides simpler constraints for the Vivado Design Suite timer to handle and can only affect output ports that reference to the virtual clock.

  Note: For a positive phase-shift, the virtual clock and the multicycle path constraint are not needed because the default setup path requirement is 1 cycle minus the amount of phase-shift.

- A forwarded clock has been identified for timing the output path based on the shared clocking connectivity.

  The forwarded clock must have been created during the third step of the wizard "Forwarded Clocks," or else the board clock or a virtual clock will be used as the output delay constraint reference clock.

Figure 2-25 shows a basic example of an output source synchronous path along with its forwarded clock for the 7 series family. Both ODDR/OSERDES instances are connected to the same clock net (highlighted in blue). The `ck_vsf_clk_2` generated clock is already defined on the `vsf_clk_2` output port.

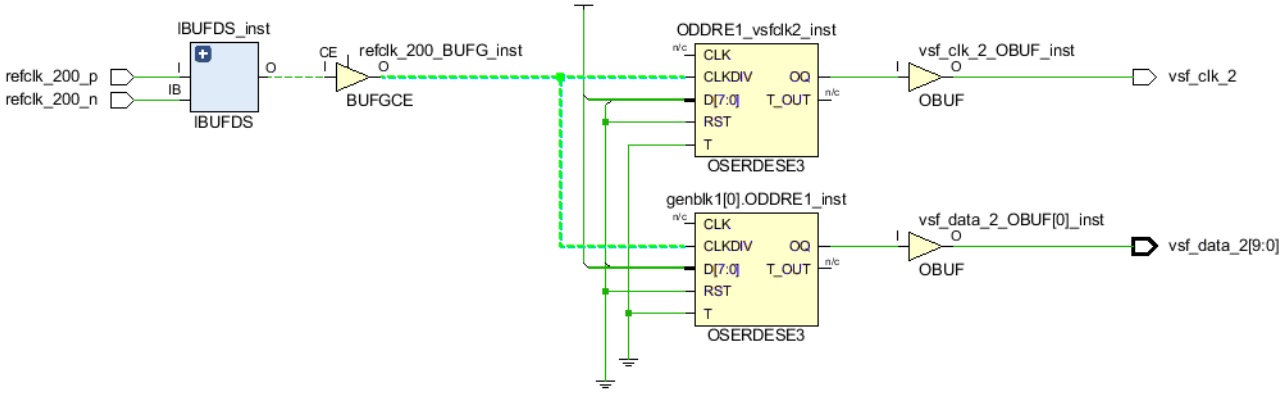


Figure 2-25:

Figure 2-26 shows the corresponding constraints in the wizard.

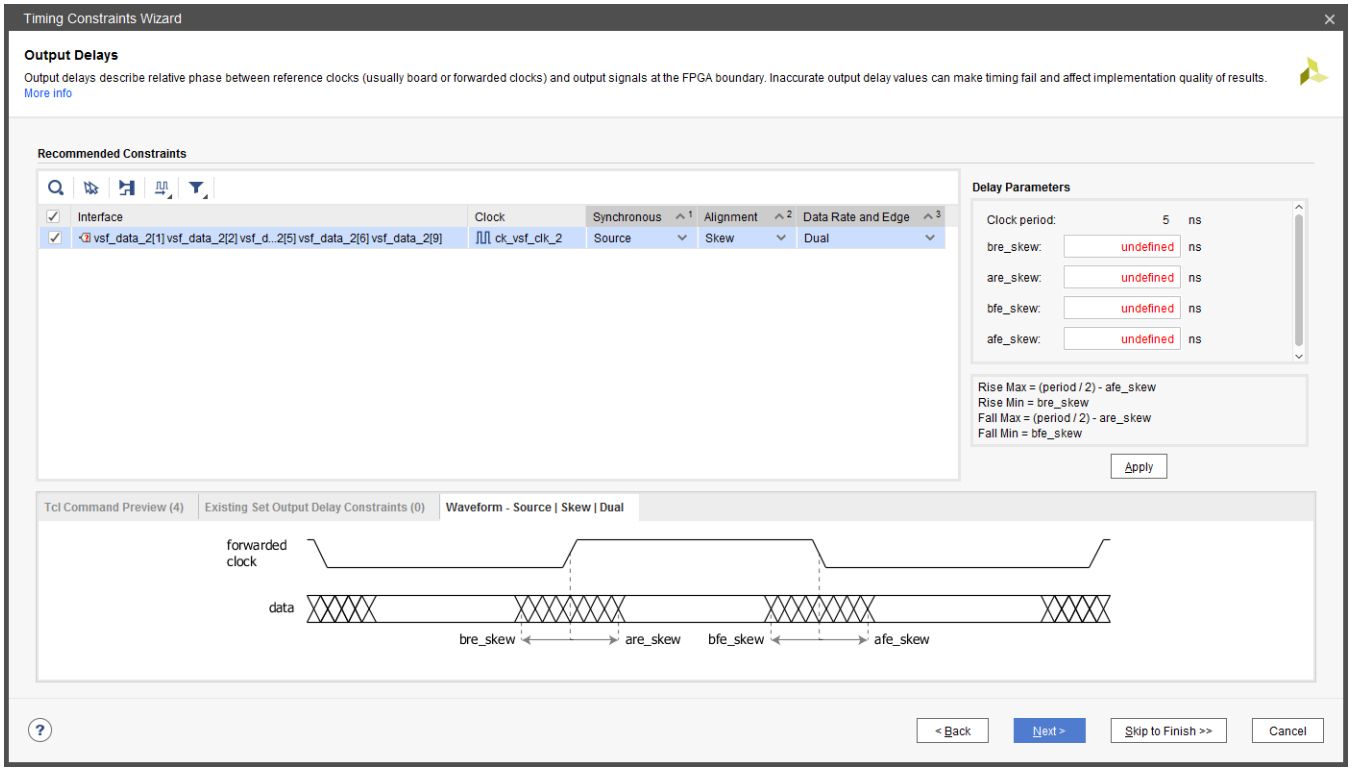


Figure 2-26:

After you select the proper template, you must enter the delay parameters values. To accelerate the delay parameter entry task, you can select and edit several constraints with same clock and same template at once. After the constraints have been completed and applied, you can review their corresponding Tcl syntax in the Tcl Command Preview tab or you can click **Next** to proceed to the next step.

TIP: *The Timing Constraints wizard skips output ports with a false path constraint. The false path constraint can only be created outside the wizard.*

Some paths propagate directly from input ports to output ports without being captured inside the device by a sequential cell. If an input port is connected to both an output port and a sequential cell, the Timing Constraints wizard does not recommend combinational constraints between the input/output port pair, because the input port should have been constrained during the Input Delay step. For the combinational paths, the wizard recommends to define a virtual clock along with input and output delays on the design ports as shown in Figure 2-27.

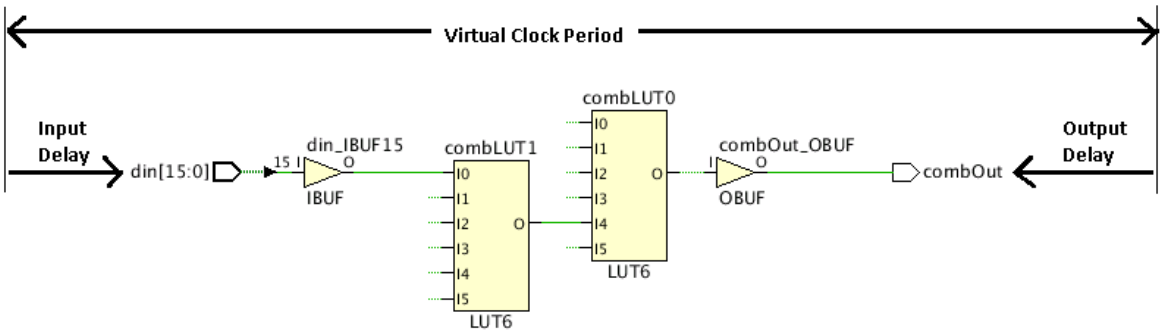


Figure 2-27:

The final combinational path delay constraints are:

- For setup analysis:

  virtual clock period - max input delay - max output delay

- For hold analysis:

  0 - min output delay - min input delay

The virtual clock period must be modified so that it is greater than the largest combinational delay constraint across all constrained combinational paths. Figure 2-28 shows the delay entries needed per input/output ports pair.

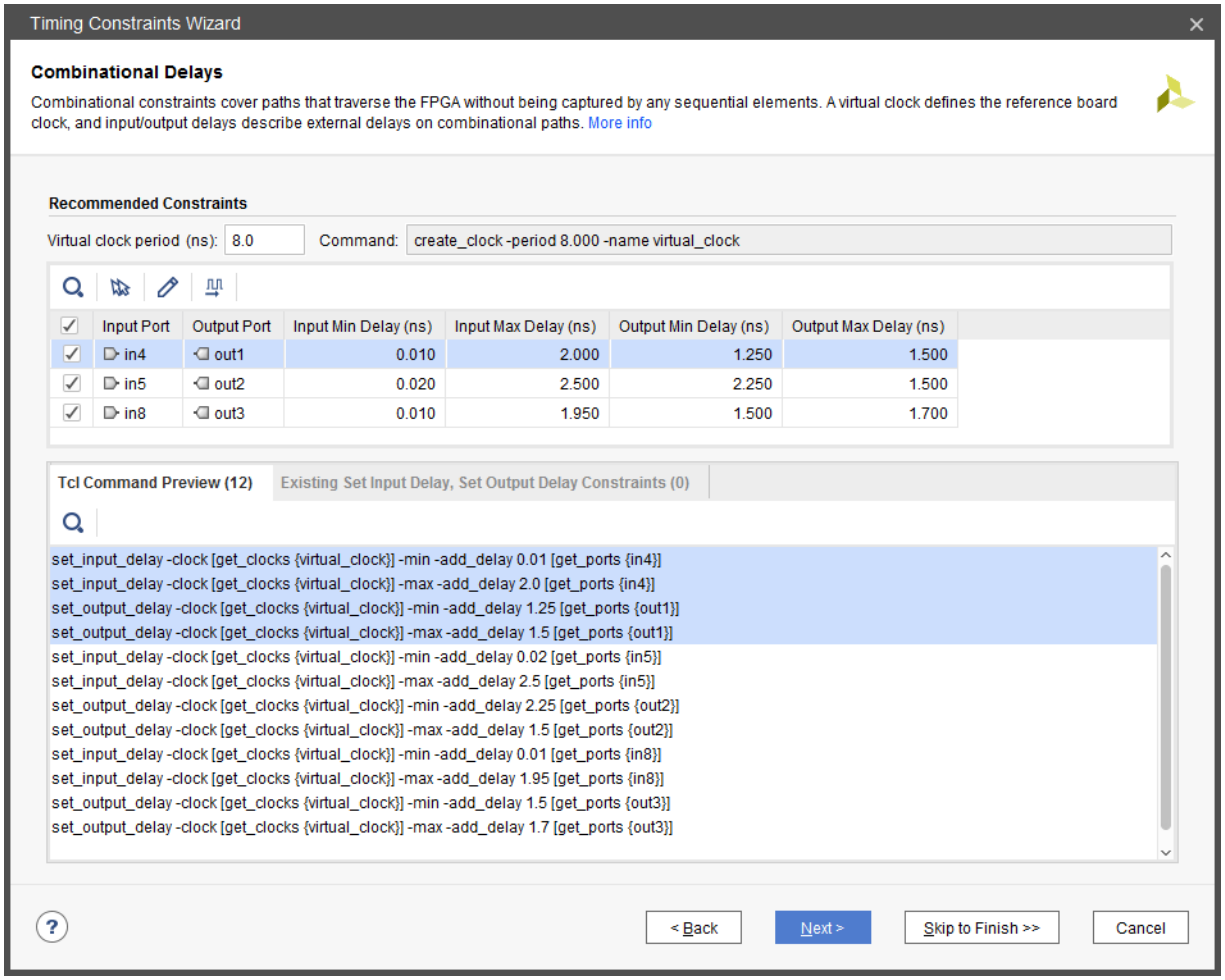


Figure 2-28:

None of the input and output delay constraints override existing ones. If a given port has multiple delay constraints with respect to the same clock, the smallest value of all constraints is used by the Vivado Timing analysis feature during hold analysis, and the largest one during setup analysis.

After all delay entries have been filled, you can click **Next** to proceed to the next step.

TIP: *Alternatively, you can constrain combinational paths using the* `set_max_delay` *and* `set_min_delay` *commands outside the Timing Constraints wizard.*

Physically exclusive clocks are clocks that are defined on the same source point and propagate on the same clock tree. Figure 2-29 shows an example where two primary clocks are defined on the same input port.

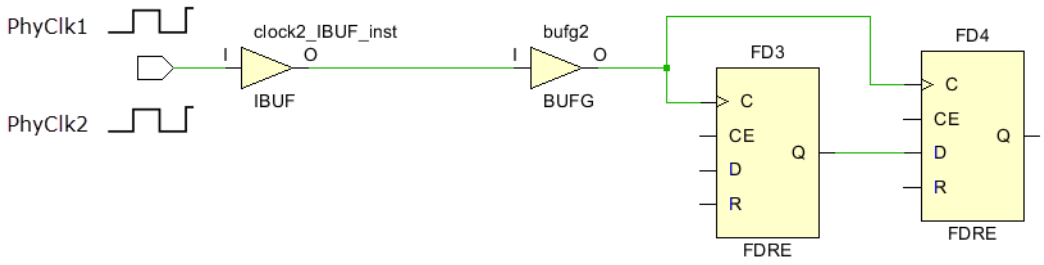


Figure 2-29:

While their overlap is convenient for timing several application modes with one design and constraint database, these clocks and their children generated clocks should never be timed together. The Timing Constraints wizard identifies such clocks and recommends a clock groups constraint to prevent unnecessary timing analysis on the clock domain crossing paths, as shown in Figure 2-30.

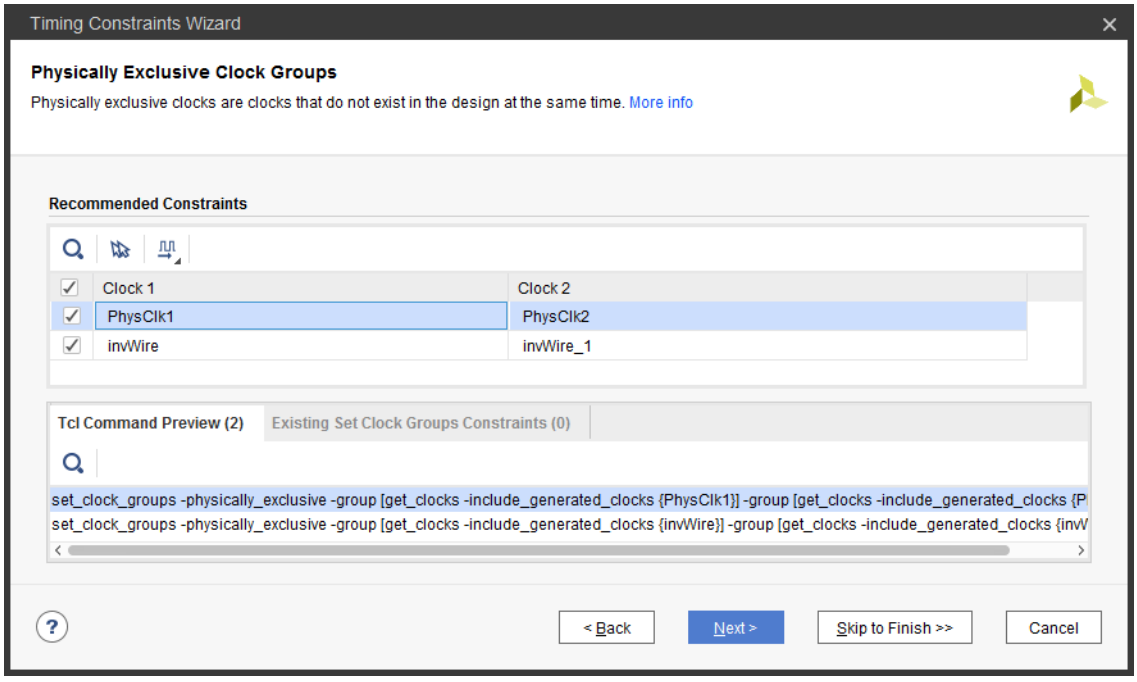


Figure 2-30:

Logically exclusive clocks are clocks that are defined on different source points but share part of their clock tree due to a multiplexer or other combinational logic. The Timing Constraints wizard identifies such clocks and recommends a clock groups constraint directly on them when they do not have timing paths between each other except for the logic connected to their shared clock tree. Figure 2-31 shows an example of two clocks, `clkA` and `clkB`, which are defined on different input ports and start overlapping on the output of a `BUFGMUX`.

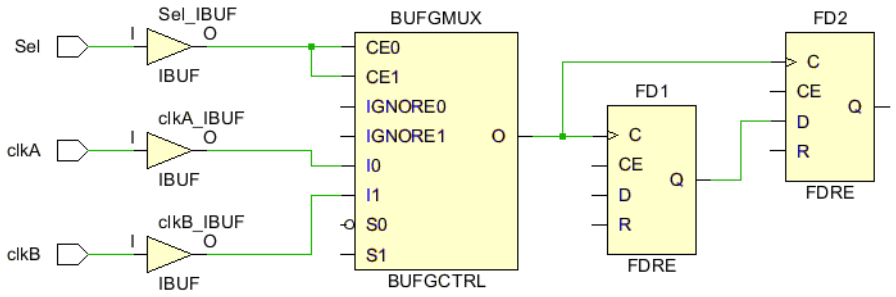


Figure 2-31:

The Timing Constraints wizard identifies logically exclusive clocks that have timing paths between each other elsewhere than just on the logic connected to the shared clock tree. Figure 2-32 shows an example where `clkA` and `clkB` have a shared clock tree portion, and also have a timing path from the shared clock tree to `clkA` only.

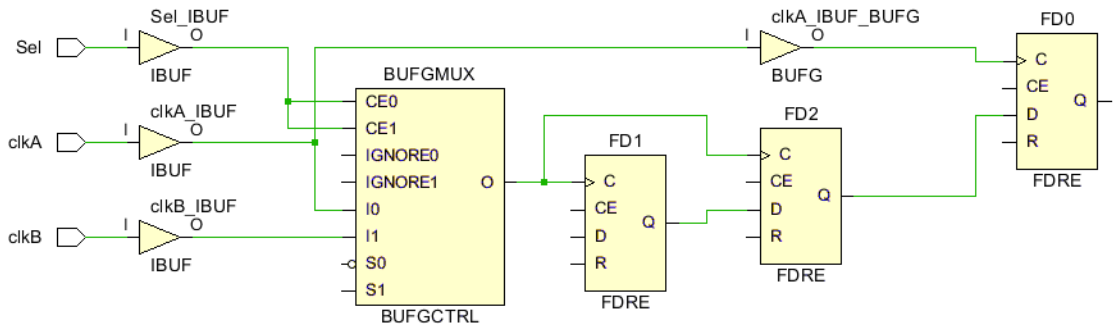


Figure 2-32:

Because only the clock domain crossing paths of the shared clock tree must be ignored, the wizard recommends to create generated clocks that are copies of `clkA` and `clkB` but that only exist on the shared clock tree. The clock groups constraint is applied to the generated clocks only, so that the paths outside the logic of the shared clock tree can still be normally timed. Figure 2-33 illustrates the wizard recommended constraints for the example above.

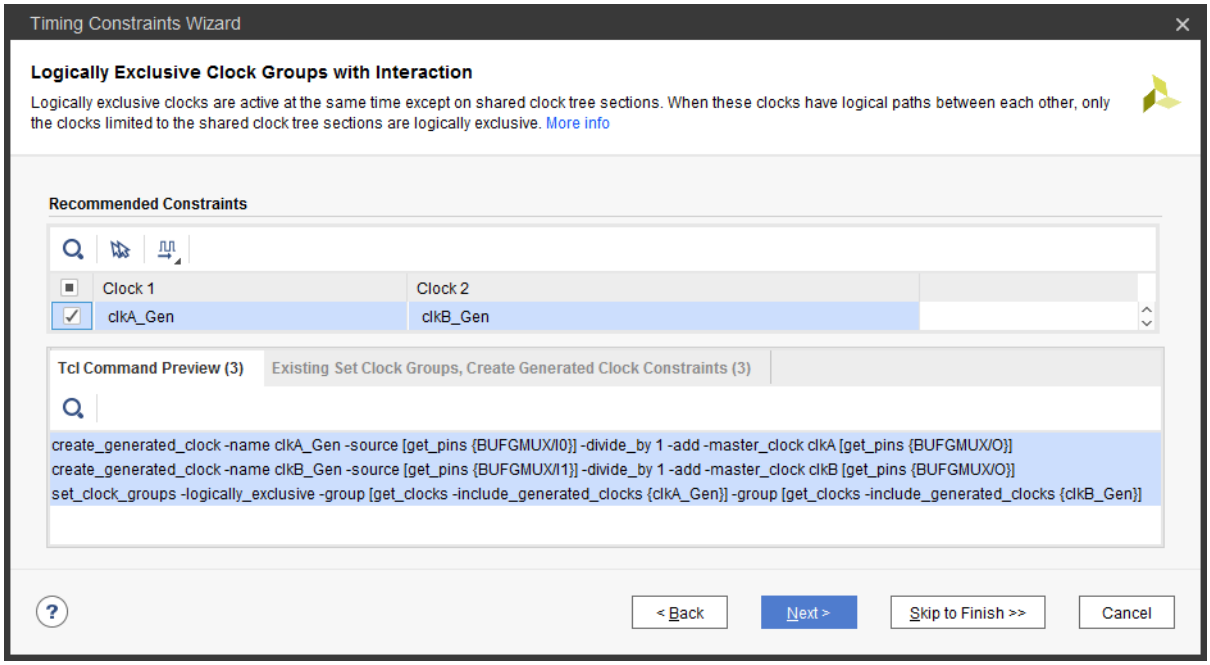


Figure 2-33:

The Timing Constraints wizard analyzes the topology of clock domain crossing (CDC) paths between asynchronous clocks and recommends clock groups or false path constraints whenever it is safe to do so.

Asynchronous clocks are clocks with no known phase relationship, which typically happens when they do not share the same primary clock or do not have a common period. For this reason, slack computation on asynchronous CDC paths is not accurate and cannot be trusted. Due to potentially large skew between asynchronous clocks, the timing quality-of-result can be heavily impacted and prevent proper timing closure if any of the asynchronous CDC paths is timed. You are responsible for adding timing exceptions on these paths, such as `set_clock_groups`, `set_false_path`, or `set_max_delay -datapath_only` to either completely ignore timing analysis or just ignore the clock skew and uncertainty. Also, the design must implement proper CDC circuitry to prevent metastability.

In the Vivado Design Suite, the wizard only identifies flip-flop-based synchronizers for synchronous data and asynchronous reset. For an example of such synchronizers, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

Figure 2-34 shows an example of the recommended and non-recommended constraints tables.

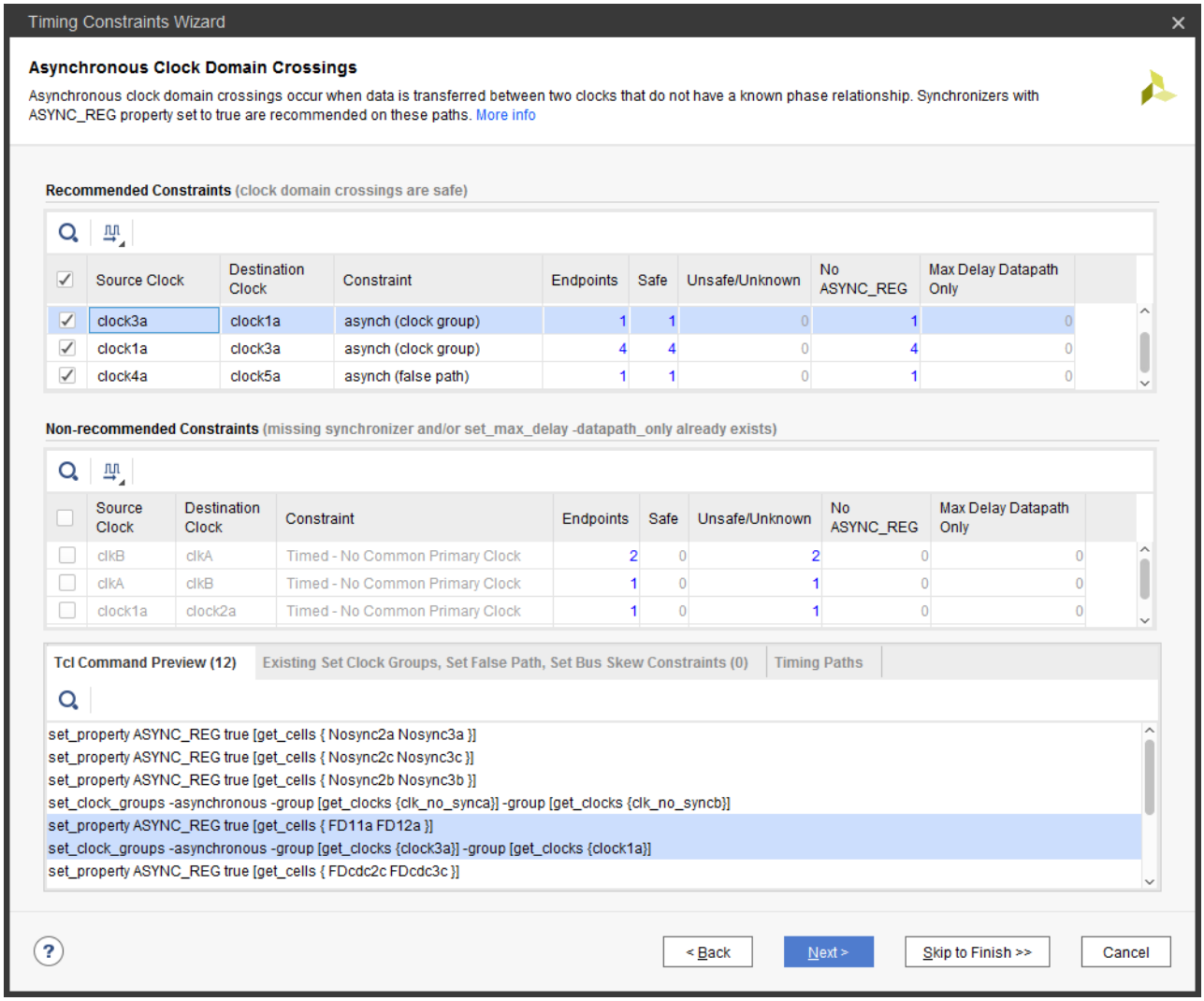


Figure 2-34:

The columns in both tables display the following information:

- **Source Clock**: this is the clock of the CDC paths start points identified by the wizard.
- **Destination Clock**: this is the clock of the CDC paths endpoints identified by the wizard.

- **Constraint**: this column shows either the dominant timing exception or the characteristics of the clock relationship when there is no exception.
  - In the Recommended Constraints table, the wizard anticipates that the constraints will be created and displays the new constraint:
    - **asynch (clock groups)** for the cases where it is safe to ignore timing in both directions, in which case a `set_clock_groups` constraint is created
    - **asynch (false path)** when it is only safe to ignore the paths in one direction, in which case a `set_false_path` constraint is created
  - In the Non-recommended Constraints table, the Timing Constraints wizard displays how the CDC paths are timed before eventually applying a clock group or false path exception:
    - **Timed - No Common Primary Clock**
    - **Timed - No Common Period**
    - **MaxDelay DataPath** for the case where at least 1 path is covered by a `set_max_delay -datapath_only` constraint and all other paths are covered by false path constraints
- **Endpoints**: the number of CDC path endpoints identified by the wizard.
- **Synchronized (with ASYNC_REG)**: the number of endpoints properly synchronized, with the `ASYNC_REG` property set to `true` on all synchronizer flip-flops.
- **Synchronizer without ASYNC_REG**: the number of synchronizers where at least one flip-flop does not have the `ASYNC_REG` property set to `true`.
- **Unknown**: the number of CDC path endpoints where the wizard did not find a synchronizer.
- **Max Delay Datapath Only**: the number of CDC path endpoints that are constrained with a `set_max_delay -datapath_only` constraint.

The table entries contain cross-probing links whenever applicable. When you click on a number, the corresponding CDC paths are listed in the Paths tab at the bottom of the window. You can select one or several CDC paths and click on the Schematic (**F4**) button to display the logic of the path(s) in the main Vivado IDE window.

Recommended Asynchronous Clock Groups Constraints

The Timing Constraints wizard recommends a `set_clock_groups -asynchronous` constraint between two clocks when the following conditions are present:

- All paths have synchronizers in both directions.
- No path is covered by a `set_max_delay -datapath_only` in either direction (`set_clock_groups` has higher precedence and overrides any existing `set_max_delay`).

Non-Recommended Asynchronous Clock Groups Constraints

The Timing Constraints wizard provides a table with constraints that are not enabled by default because they are not recommended for one of the following reasons:

- At least one path is missing a synchronizer in either direction.
- At least one path is covered by `set_max_delay -datapath_only` in either direction.

You can decide to activate any of these constraints when working on an early version of the design, and then revisit the CDC paths and their constraints later when finalizing your design.

CDC Synchronizers and ASYNC_REG Property

Xilinx recommends that all synchronizer flip-flops have their `ASYNC_REG` property set to `true` in order to preserve the synchronizer cells through any logic optimization during synthesis and implementation, and to optimize their placement for best Mean Time Between Failures (MTBF) statistics. For any clock group constraints that are enabled in both tables (either by default or by the user), the wizard sets to `true` any missing `ASYNC_REG` property.

Refer to the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 11] for detailed information about the `ASYNC_REG` property.

Completing the CDC Analysis and Constraints

The Timing Constraints wizard does not recognize some valid CDC topologies that are not based on simple synchronizers. The `report_cdc` command provides a powerful and more comprehensive view of the CDC paths that need structural correction in order to become safe. Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4] for detailed information about `report_cdc`.

For the cases where the wizard does not recommend a constraint due to the presence of some `set_max_delay -datapath_only`, the other CDC paths that are normally timed must be reviewed individually and possibly ignored by additional false path constraints. The creation of point-to-point false path constraints must be done in the XDC file, in the Tcl Console, or in the Timing Constraints window after exiting the wizard.

The final page of the Timing Constraints wizard summarizes the new constraints that will be applied and saved at the end of the Target XDC file when you click **Finish**. Click each hyperlink to see the details of the constraints. Figure 2-35 below shows an example of the Constraints Summary page.

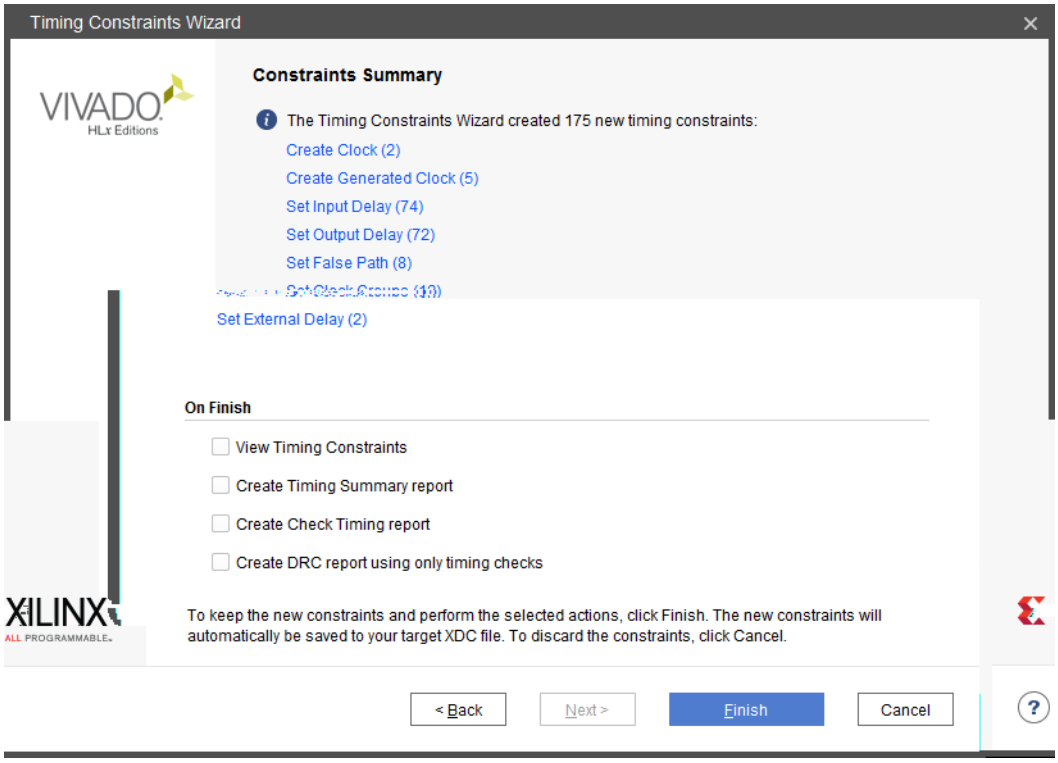


Figure 2-35:

The Timing Constraints window is available for Synthesized and Implemented designs only. For elaborated design constraints, you must use and edit XDC files directly. For more information, see Creating Synthesis Constraints, page 59.

You can open the Timing Constraints window using one of the following three options, as shown in Figure 2-36:

- Select **Window > Timing Constraints**.
- In the Synthesis section of the Flow Navigator panel, select **Synthesized Design > Edit Timing Constraints**.
- In the Implementation section of the Flow Navigator panel, select **Implemented Design > Edit Timing Constraints**.

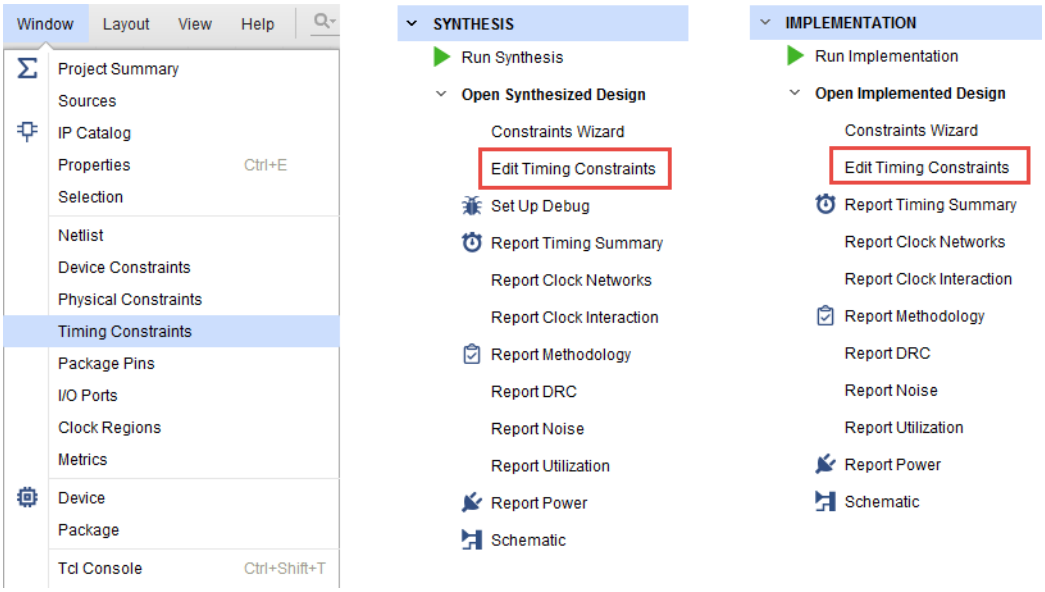


Figure 2-36:

The Timing Constraints window displays the timing constraints in memory, in either the same sequence as in the XDC files and Tcl scripts, or the same sequence in which you entered them in the Tcl Console.

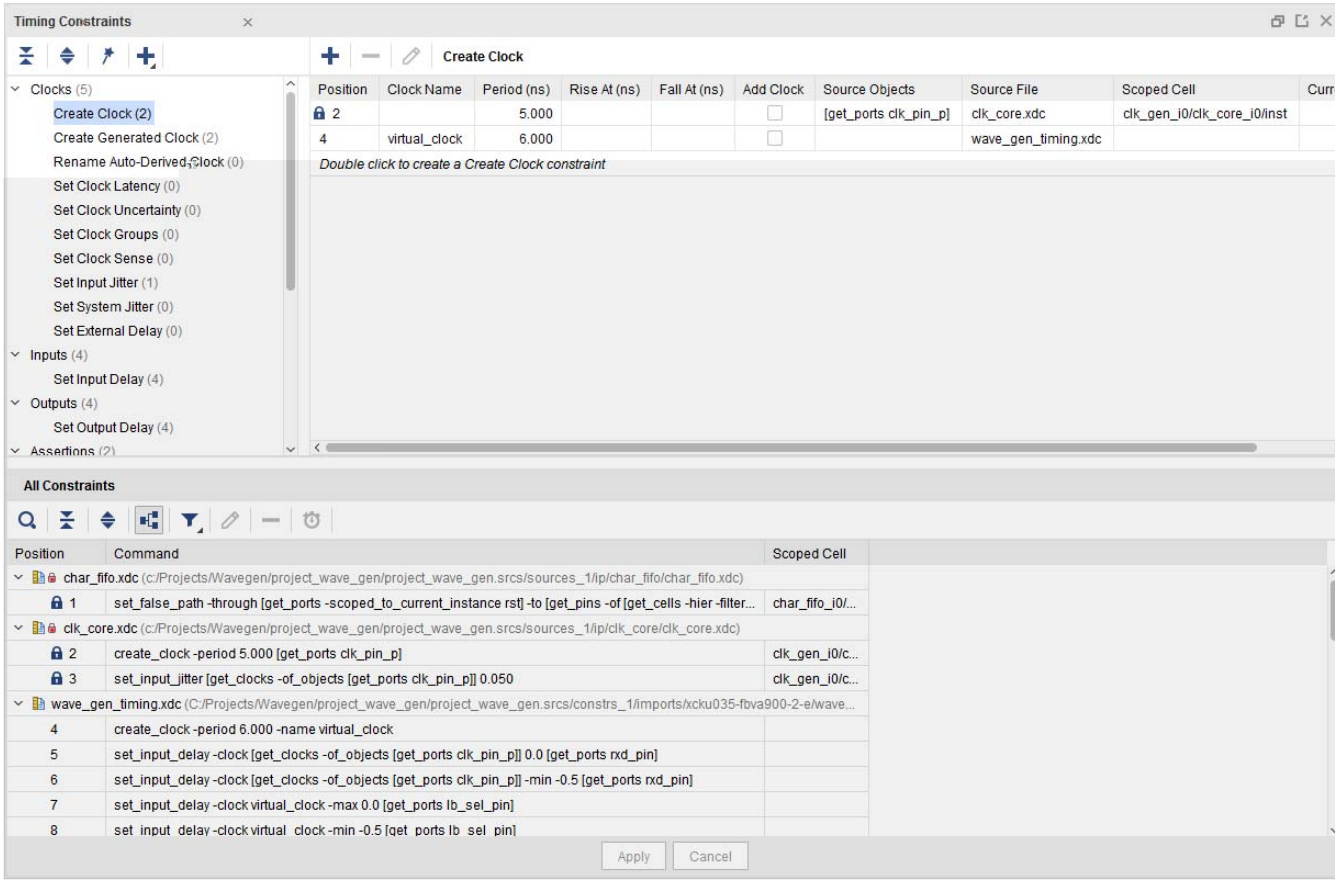


Figure 2-37:

Some of the constraints cannot be edited from this window. They are marked with the XDC **No Edit** icon .

Timing Constraints Spreadsheet

The timing constraints spreadsheet displays the details of all existing constraints of a specific type. Use the timing constraints spreadsheet to review and edit constraint options.

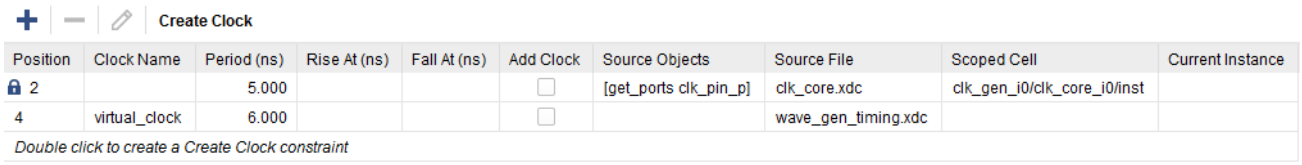


Figure 2-38:

The two last columns of the panel show:

- Source File: The name of the XDC file or Tcl script the constraint comes from
- Scoped Cell: The name of the current instance when the constraint was applied. This name usually corresponds to an IP instance which is delivered with dedicated constraints. For more information, see Constraints Scoping, page 68.

A new constraint of the selected type can be created by double clicking the last line of the spreadsheet. The corresponding constraint creation dialog opens and lets you fill in the details of the new constraint. Click **OK** to apply the constraint in memory and close the window. A new line in the spreadsheet shows the new constraint information.

You can edit any existing constraint by modifying the values directly in the spreadsheet. After you have finished editing, click **Apply** to apply the modified constraints in memory.

IMPORTANT: *Applying a new or modified constraint does not save it in the XDC file. You must click **Save Constraints** to save it.*

IMPORTANT: *IP constraints cannot be edited or deleted. In order to modify a constraint delivered with an IP, you must disable the corresponding IP XDC file, copy the constraint to your XDC file, and edit the constraint as desired.*

Constraints Creation, Grouped by Category

When you select a constraint type, the corresponding spreadsheet appears on the right sub-window panel. This allows you to view all the constraints of the same type that have already been created.

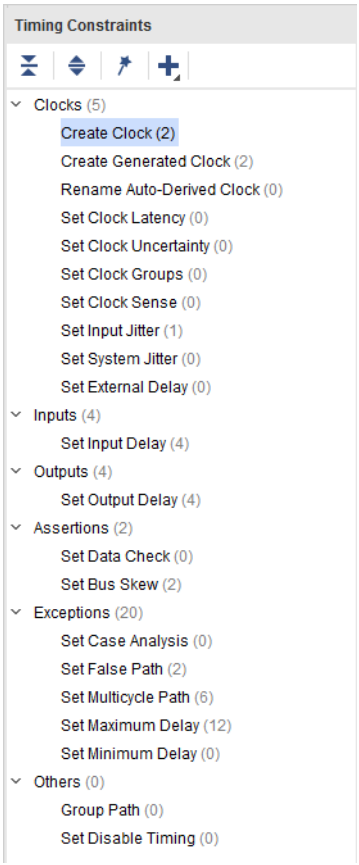


Figure 2-39:

To create a new constraint, double click the name of the target constraint. A dialog box allows you to specify the value for each option. When you click **OK**, the tool does the following:

1. Validates the syntax.
2. Applies the syntax to the memory.
3. Adds the new constraint at the end of the spreadsheet.
4. Adds the new constraint at the end of your complete list of constraints.

All Constraints

The bottom of the window displays the complete list of constraints loaded in memory, in the same sequence as they were applied. The constraints are grouped in accordance with the XDC file or the Tcl script from which they originated. When an XDC file is scoped to a particular hierarchical cell, the cell name is displayed next to the file name.

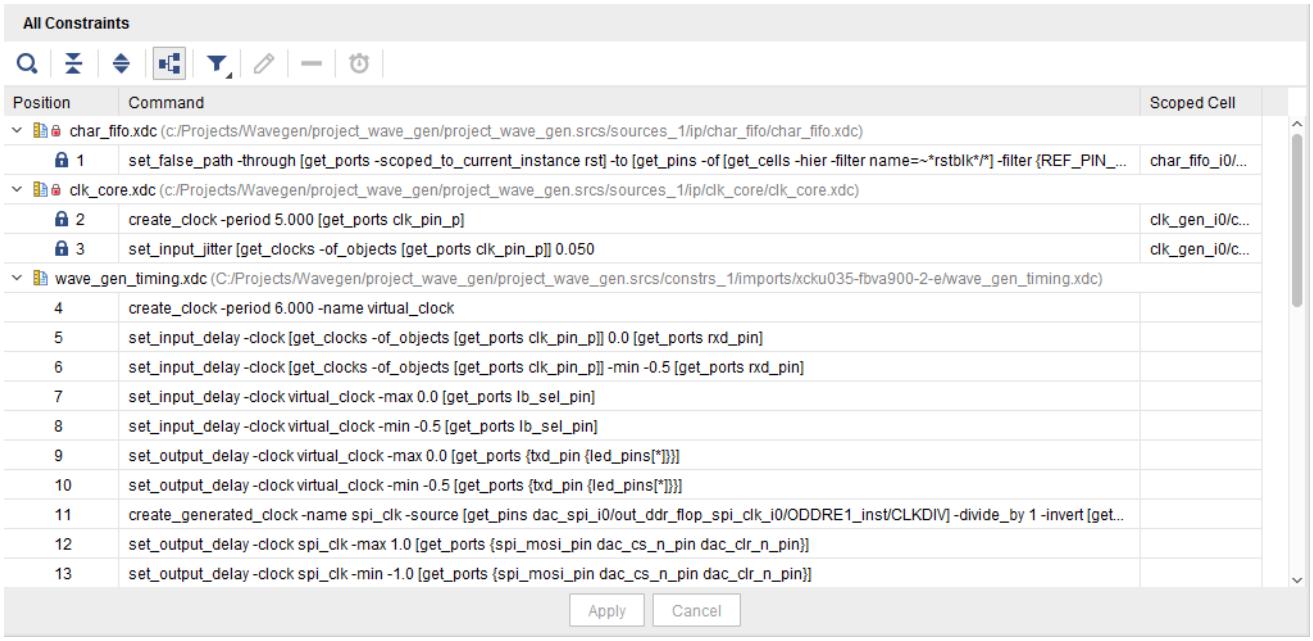


Figure 2-40:

You can expand and collapse the constraints for each associated source file, or completely by clicking the two corresponding buttons on the left side of the panel.

Figure 2-41:

TIP: *The collapsed view provides a compact overview of which constraints file are loaded in memory, and where the scoping mechanism is used. The same information is available through the* `report_compile_order -constraints` *command.*

De-select the **Group by Source** icon to switch the view to a table in which the source constraint file and the scoped cell information appears in the two right columns.

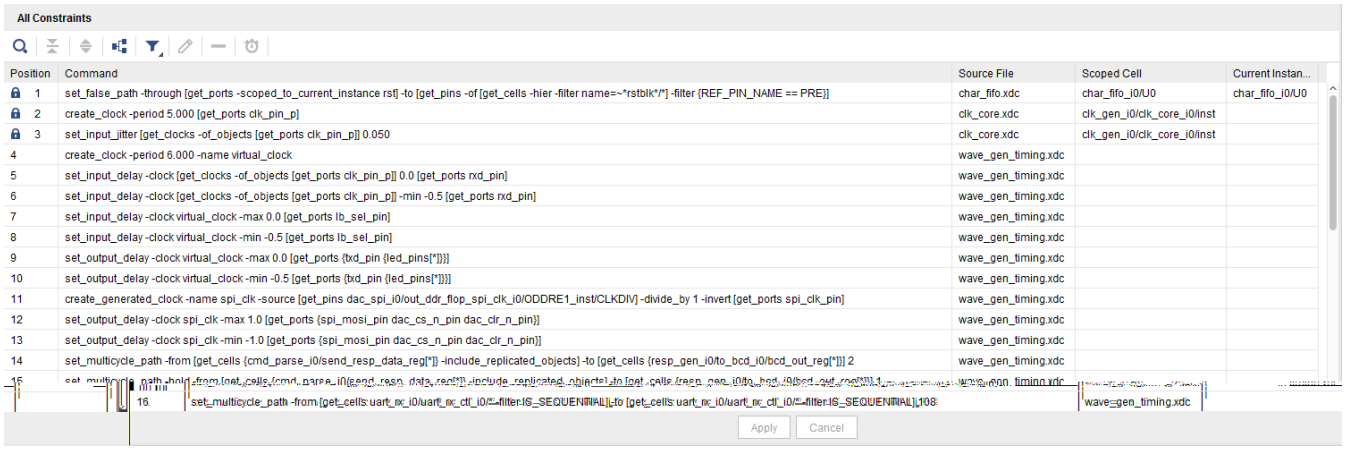


Figure 2-42:

- To delete a constraint, select it and click **X**.
- To edit a constraint that is not read-only, use the spreadsheet view. After your changes have been registered by the tool, you must click **Apply** to refresh the constraints in memory.
- To add new constraints, use the dialog boxes as previously described, or type the constraints in the Tcl console. The new constraint appears at the end of the list in a group named `<unsaved_constraints>`.

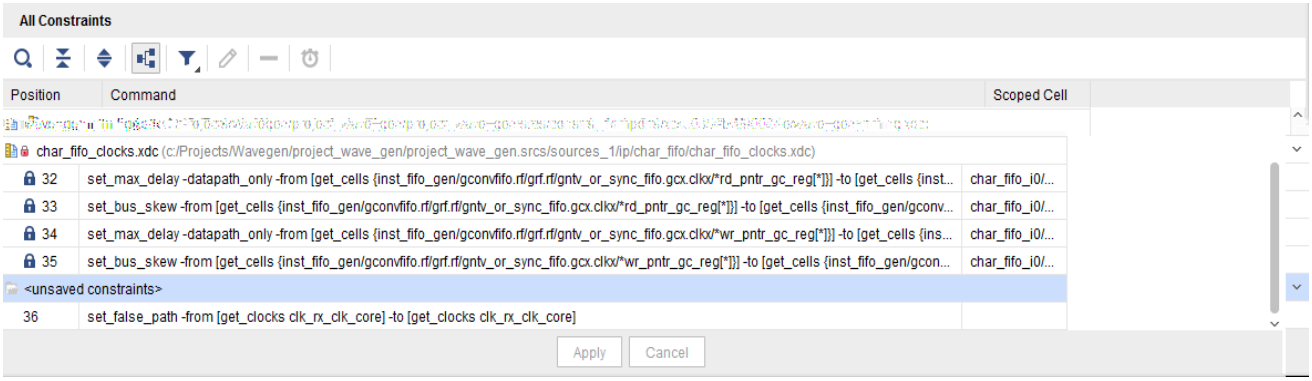


Figure 2-43:

When saving the constraints, the new constraints are saved at the end of the XDC file marked as `target`. If there is no target XDC file in the constraint set associated with the design in memory, or if there is only a Tcl script in the constraint set, you are prompted to specify where to save the constraints.

Regularly save your constraints. Click **Save**, or select **File** > **Constraints** > **Save**.

IMPORTANT: *New and modified constraints cannot be saved back to a Tcl script.*

CAUTION! *Do not enter new constraints in the Tcl Console if any constraints in the Timing Constraints window have not yet been applied. The final constraints order in the editor can become different from the constraints order in memory. In order to avoid any confusion, you must re-apply all constraints each time you edit an existing constraint.*

You can access XDC templates by selecting **Tools** > **Language Templates**.

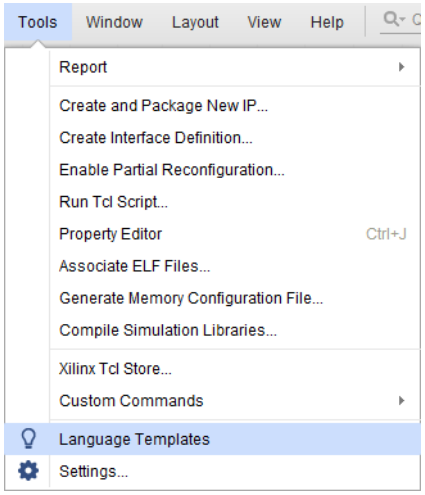


Figure 2-44:

XDC Template Contents

The XDC templates include:

- The most common timing constraints, such as clock definitions, jitter, input/output delay, and exceptions
- Physical constraints
- Configuration constraints

Using XDC Templates

To use an XDC template:

1. Select the template you want to use.
2. Copy the text displayed in the Preview window.
3. Paste the text in your XDC file.
4. Replace the generic strings with actual names from your design or with appropriate values.

Advanced XDC Templates

Some advanced templates such as System Synchronous and Source Synchronous I/O delay constraints require you to set some Tcl variables to capture the design requirements. The Tcl variables are used in the actual `set_input_delay` and `set_output_delay` constraints.

You must verify that all necessary values have been filled instead of using the default values.

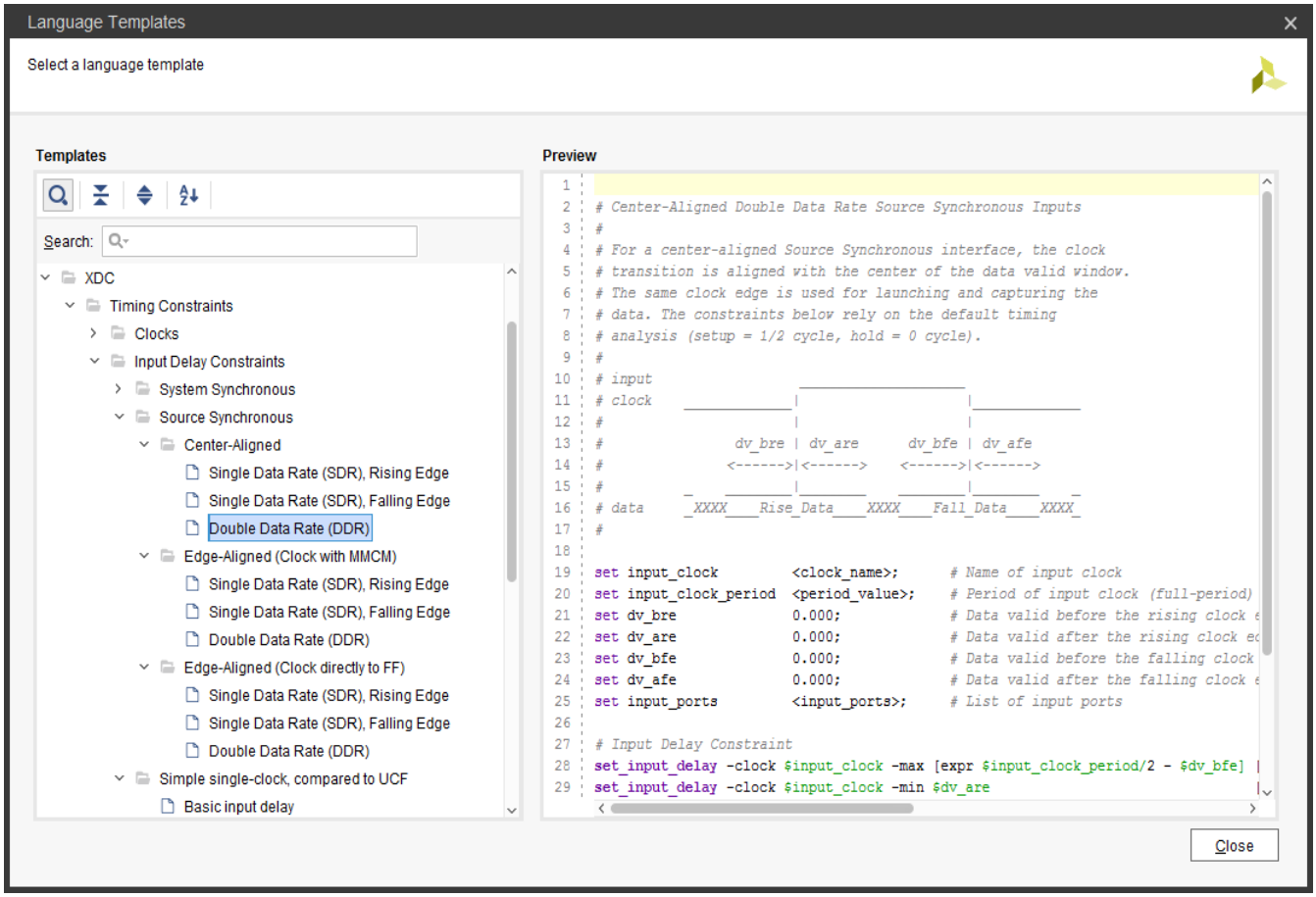


Figure 2-45:

The Vivado Synthesis transforms the RTL description of your design into a technology mapped netlist. This process happens in several steps, and includes a number of timing-driven optimizations.

Xilinx® FPGAs include many logic features that can be used in many different ways. Your constraints are needed to guide the synthesis engine towards a solution that meets all the design requirements at the end of implementation.

There are four categories of constraints for the Vivado IDE synthesis:

- RTL Attributes
- Timing Constraints
- Elaborated Design Constraints

RTL attributes must be written in the RTL files. They usually choose the mapping style of certain part of the logic, as well as preserving certain registers and nets, or controlling the design hierarchy in the final netlist.

For more information, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

IMPORTANT: *The `DONT_TOUCH` attribute does not obey the properties of `USED_IN_SYNTHESIS` and `USED_IN_IMPLEMENTATION`. If you use `DONT_TOUCH` properties in the synthesis XDC, it is propagated to implementation regardless of the value of `USED_IN_IMPLEMENTATION`.*

For more information about `USED_IN_SYNTHESIS` and `USED_IN_IMPLEMENTATION`, Refer to *Synthesis and Implementation Constraint Files, page 11*.

`DONT_TOUCH` attribute example:

```
set_property DONT_TOUCH true [get_cells fsm_reg]
```

Timing constraints must be passed to the synthesis engine by means of one or more XDC files. Only the following constraints related to setup analysis have any real impact on synthesis results:

- `create_clock`
- `create_generated_clock`
- `set_input_delay`
- `set_output_delay`
- `set_clock_groups`
- `set_false_path`
- `set_max_delay`
- `set_multicycle_path`

Physical and configuration constraints are ignored by the synthesis algorithms.

RECOMMENDED: *When you create the first version of your synthesis XDC, use simple timing constraints to describe the high-level design requirements.*

At this point in the flow, the net delay modeling is still not very accurate. The main goal is to obtain a synthesized netlist which meets timing, or fail by a small amount, before starting implementation. In many cases, you will have to go through several XDC and RTL modification iterations before you can reach this state.

The RTL-based XDC creation iteration is shown in Figure 2-46. It is based on the utilization of the Elaborated design to find the object names in your design that you want to constrain for synthesis.

You must use the Tcl Console to validate the syntax of the XDC commands before saving them in the XDC files. With the elaborated design, you can create constraints, query clocks, and query design objects, but you cannot run any timing report command.

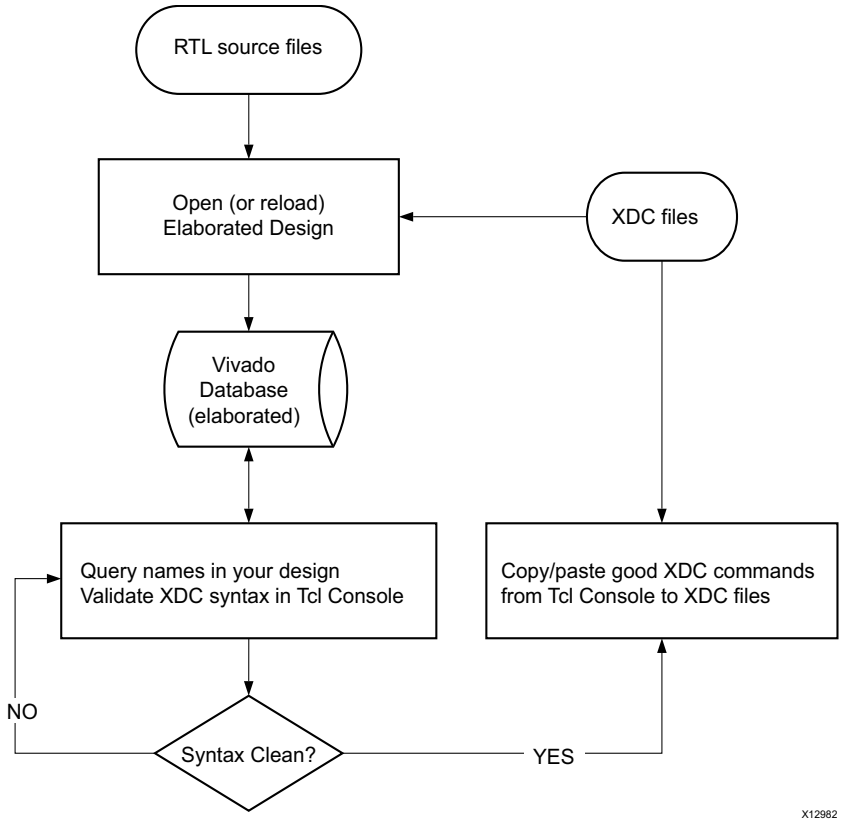


Figure 2-46:

Design objects that are safe to use when writing constraints for synthesis are:

- Top level ports
- Manually instantiated primitives (cells and pins)

Some RTL names are modified or lost during the creation of the elaborated design. Following are the most common cases:

- Single-Bit Register Names
- Multi-Bit Register Names
- Absorbed Registers and Nets
- Hierarchical Names

By default, the register name is based on the signal name in the RTL, plus the `_reg` suffix.

For example, for a signal defined as follows in VHDL and Verilog, the instance name generated during the elaboration is `wbDataForInputReg_reg`:

```
VHDL: signal wbDataForInputReg : std_logic;
Verilog: reg wbDataForInputReg;
```

Figure 2-47 shows the schematic of the register, and its pins. It is possible to define a constraint on the register instance or its pins.

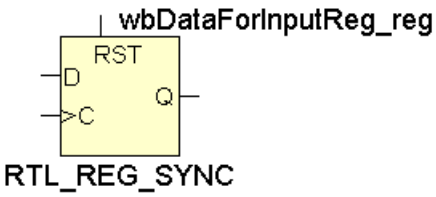


Figure 2-47:

By default, the register name is based on the signal name in the RTL, plus the `_reg` suffix. You can only query and constrain individual bits of the multi-bit register in your XDC commands.

For example, for a signal defined as follows in VHDL and Verilog, the instance names generated during the elaboration are `loadState_reg[0]`, `loadState_reg[1]`, and `loadState_reg[2]`:

```
VHDL: signal loadState: std_logic_vector(2 downto 0);
Verilog: reg [2:0] loadState;
```

Figure 2-48 shows the schematic of the register. The multi-bit register appears as a vector of single-bit registers. The vector is represented in a compact way whenever possible in the schematics. Each individual bit can also be displayed separately.

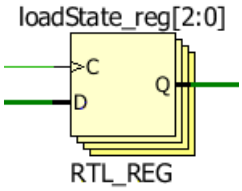


Figure 2-48:

You can only constrain each register individually or as a group by using the following patterns:

- Register bit `0` only

```
loadState_reg[0]
```

- All register bits

```
loadState_reg[*]
```

IMPORTANT: *You cannot query the multi-bit register, or more generally any multi-bit instance, by using the pattern* `loadState_reg[2:0]`.

Because the names above also correspond to the names in the post-synthesis netlist, any constraint based on them will most probably work for implementation as well.

Some registers or nets in the RTL sources can disappear in the elaborated design (or synthesized design) for various reasons. For example, memory block, DSP or shift register inference requires absorbing several design objects into one resource. Instead of using these objects to define constraints, try to find other connected registers or nets that you can use.

Unless you plan to force Vivado synthesis to keep the complete hierarchy of your design, some or all levels of the hierarchy can be flattened during synthesis. For more information, see the `-flatten_hierarchy` information at this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8].

RECOMMENDED: *Use fully resolved hierarchical names in your synthesis constraints where all the hierarchical levels are explicitly written ("/" character) instead of using implicit matching ("*" character). They are more likely to be matching the final netlist names regardless of the hierarchy transformations.*

For example, consider the following register located in a sub-level of the design.

Elaborated Design Example:

```
inst_A/inst_B/control_reg
```

During synthesis (assuming no special optimization is performed on this register), you can get either flat or hierarchical name depending on the tool options or the design structure.

Instance name in a flat netlist:

```
inst_A/inst_B/control_reg     (F)
```

Instance name in a hierarchical netlist:

```
inst_A/inst_B/control_reg     (H)
```

There is no obvious difference because the / character is also used to mark flattened hierarchy levels. You will notice the difference when querying the object in memory. The following commands will return the netlist object for F but not H:

```
% get_cells -hierarchical *inst_B/control_reg
% get_cells inst_A*control_reg
```

In order to avoid problems related to hierarchical names, Xilinx recommends that you do the following:

- Use `get_*` commands without the `-hierarchical` option.
- Mark explicitly with the forward-slash (/) character all the levels of hierarchy as they show in the elaborated design view.

Examples Without Hierarchical Option:

- This option works for both flat and hierarchical netlists:

```
% get_cells inst_A/inst_B/*_reg
% get_cells inst_*/inst_B/control_reg
```

- Another option is:

```
% get_cells  -hier -filter {NAME =~ inst_A/inst_B/*_reg}
% get_cells  -hier -filter {NAME =~ inst_*/inst_B/control_reg}
```

CAUTION! *(1) Do not attach constraints to hierarchical pins during synthesis for the same reason as explained above for hierarchical cells. (2) Do not attach constraints to nets connecting combinatorial logic operators. They will likely be merged into a LUT and disappear from the netlist.*

RECOMMENDED: *Regularly save your XDC files after editing, and reload the Elaborated design in order to make sure the constraints in memory and the constraints in the XDC files are the same. After running synthesis, load the synthesized design with the same synthesis XDC in memory, and run timing analysis by using the timing summary report.*

Some pre-synthesis constraints might no longer apply properly because of the transformations performed by synthesis on the design. To resolve these problems, do the following:

1. Find the new XDC syntax that applies to the synthesized netlist.
2. Save the constraints in a new XDC file to be used during implementation only.
3. Move the synthesis constraints that can no longer be applied to a separate XDC file that will be used for synthesis only.

After you have a synthesized netlist, you can load it into memory together with the XDC files or Tcl scripts enabled for implementation. You must review the messages issued by the tool when loading the XDC in order to verify and correct any constraint that cannot be applied.

In some cases, the object names in the synthesized netlist are different from the names in the elaborated design. If this is the case, you must recreate some constraints with the corrected names, and save them in an implementation-only XDC file.

After the tool can properly load all the XDC files, you can run timing analysis in order to:

- Add missing constraints, such as input and output delay.
- Add timing exceptions, such as false paths, multicycle paths, and min/max delay constraints.
- Identify large violations due to long paths in the design and correct the RTL description.

You can use the same base constraints as during synthesis, and create a second XDC file to store all new constraints specific to implementation. You can choose to save physical and configuration constraints in a separate XDC file.

Note: In project mode, opening a synthesized design results in linking the netlist(s) from the post-synthesis DCP(s) to build the full top-level hierarchical netlist. All XDC constraints marked for implementation are also automatically loaded. This enables you to verify the implementation constraints on the full synthesized design. This means that if the implementation constraints are modified, the opened synthesized design goes out of date, not the synthesized run. The GUI shows a small banner and provides the option to reload the design.

The netlist-based XDC iteration is shown in Figure 2-49.

During synthesis, some registers are replicated to improve the design performance. The user XDC constraints are not modified by the synthesis engine to include the replicated cells. If a timing constraint is attached to an object replicated by Vivado Synthesis, the replicated cells are not always covered by the XDC constraints depending on how the constraint is written, which can later impact the implementation quality of results.

When using Vivado Synthesis, the `get_cells` and `get_pins` commands provide a mechanism to automatically include the replicated objects.

For example, `set_false_path –from [get_cells –hierarchical *rx_reg]` can be rewritten as follows to also safely include the replicated objects during implementation:

```
set_false_path -from [get_cells -hierarchical *rx_reg -include_replicated_objects]
```

The command line option `-include_replicated_objects` relies on the property `ORIG_CELL_NAME` set on the replicated objects. The following query commands return the original cells with the replicated cells:

```
get_cells -include_replicated_objects *rx_reg
get_cells -include_replicated_objects [get_cells -hier -filter {NAME =~ *rx_reg}]
get_cells -hierarchical -filter {NAME =~ *rx_reg || ORIG_CELL_NAME =~ *rx_reg}
```

The `-filter` option always applies after the collection of objects is built. It is not recommended to use `-filter` with `-include_replicated_objects` when the filtering expression refers to the property `NAME`. In such scenarios, the replicated objects are not returned when they do not match the pattern specified for `NAME`. For example, the syntax below does not return replicated objects matching `*reg_replica*`:

```
get_cells -include_replicated_objects -filter {NAME =~ *rx_reg}
```

Xilinx recommends running the Methodology checks (`report_methodology`) and reviewing the XDCV-1 and XDCV-2 check messages to identify constraints that need to be updated with the `get_cells/get_pins -include_replicated_objects` option.

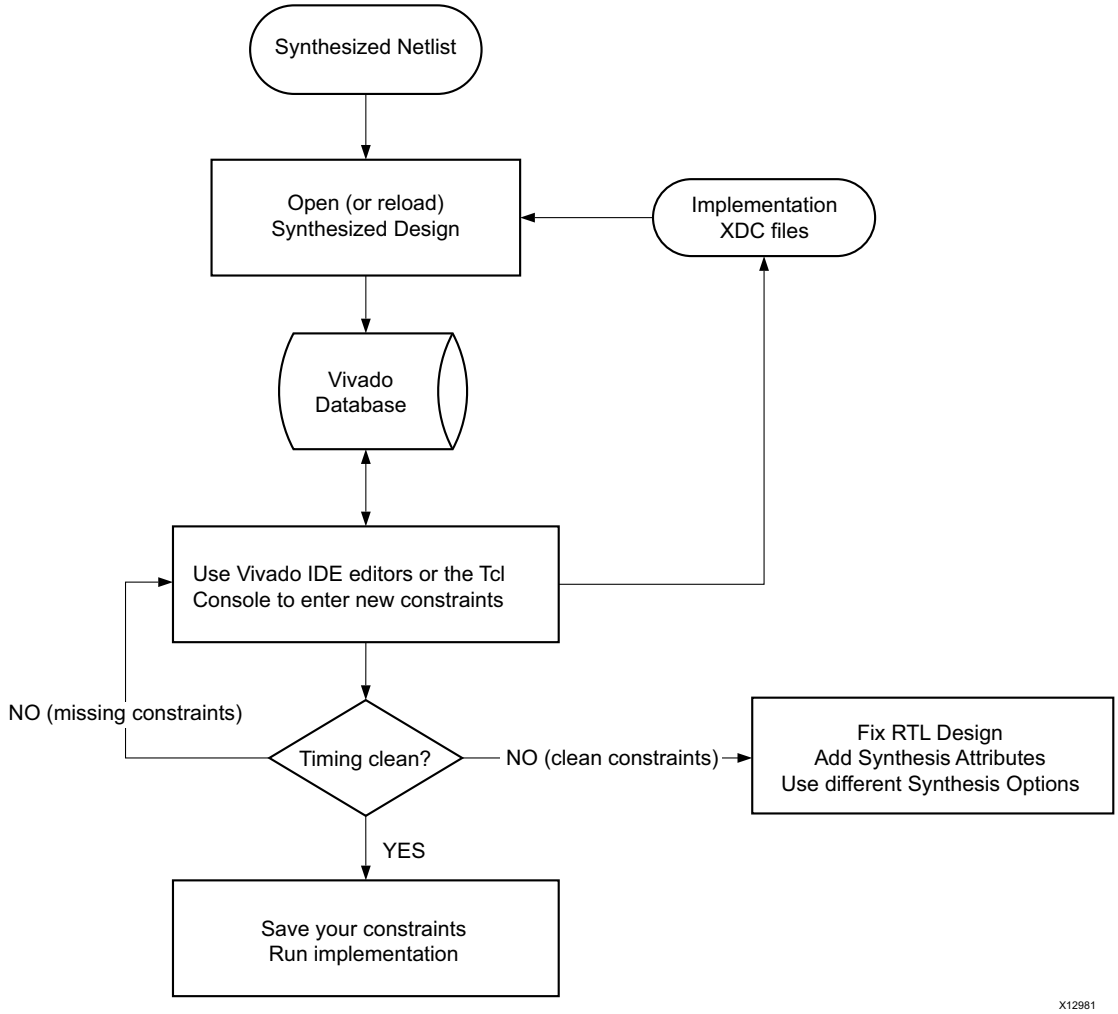


Figure 2-49:

Before proceeding to implementation, you must verify that your design does not include any major timing violation. The place-and-route tools can fix most reasonable timing violations, but they cannot fix fundamental design issues that make timing closure impossible.

RECOMMENDED: *Revisit the RTL to reduce the number of logic levels on the violating paths and to clean up the clock trees in order to use dedicated clock resources and minimize the skew between related clocks. You can also add synthesis attributes and use different synthesis options.*

For more information, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 8], or this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 9].

When using Out-Of-Context (OOC) synthesis mode, the OOC modules (IP/BD/DFx/...) are inferred as a black-box inside the top level. This means that the netlist objects inside the

OOC modules are not accessible by the top-level constraints. This may require the top-level constraints for synthesis to be different from the constraints for implementation. In Project Mode, this can be done by creating a specific XDC file for synthesis and setting the properties USED_IN_SYNTHESIS=TRUE & USED_IN_IMPLEMENTATION=FALSE on it. The top-level XDC for implementation should have USED_IN_SYNTHESIS=FALSE.

The only objects accessible from the black-boxes are the input and output ports. This limits the type of timing constraints that the top-level can specify when referring to a black-box.

Some of the limitations for the top-level constraints from OOC synthesis are:

- Auto-derived clocks generated inside the OOC module cannot be renamed.
- Clock names defined inside the OOC module cannot be referred to. The clock propagating to the output of the OOC module is named based on the net connected to the port of the module, not from the name it has inside the module, even if the clock is renamed inside the module XDC.
- If the top-level constraints need to refer to the clock coming out of an OOC module, it should use a query such as 'get_clocks -of_objects [get_pins <MODULE_OOC_OUTPUT_CLOCK_PORT>]'.

The constraints from a particular XDC file can be optionally scoped to a specific module, to specific cells of your design, or both, if needed. This is convenient for creating and applying constraints to a sub-level of your design without having any information about the top-level. The block-level constraints must be developed independently from the top-level constraints, and must be as generic as possible so that they can be used in various contexts. They must also not affect any logic that is beyond the block boundaries. By default, all the IP cores from the Vivado IP Catalog generated within a Vivado Design Suite project use this mechanism to load their constraints in memory.

The constraints scoping mechanism is activated by specifying the following properties on the XDC files:

- SCOPED_TO_REF: This property takes the name of a module (or entity). The constraints are applied to ALL instances of the specified module (or entity) only,
- SCOPED_TO_CELLS: This property takes a list of hierarchical cell names. The constraints are scoped and applied to each hierarchical cell individually,
- SCOPED_TO_REF + SCOPED_TO_CELLS: If both these properties are specified, the constraints are applied to each cell of the `SCOPED_TO_CELLS` list, located inside the module (or entity) specified by `SCOPED_TO_REF`.

These properties are automatically set by the Vivado Design Suite for IP cores added to your RTL project by means of the IP Catalog.

Setting XDC File Scoping Properties Example

Figure 2-50 shows the `uart_tx_i0` cell, an instance of the `uart_tx module`, which includes two hierarchical cells, `uart_tx_ctl_i0` and `uart_baud_gen_tx_i0`.

The project includes an XDC file `uart_tx_ctl.xdc` to constrain the `uart_tx_ctl` module.

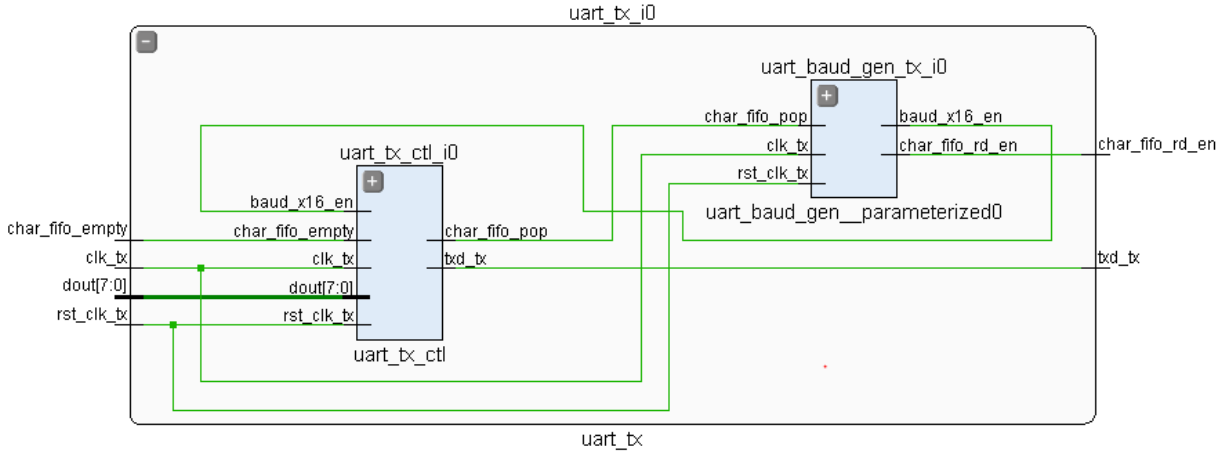


Figure 2-50:

Following are three equivalent Tcl examples to use the scoping properties on `uart_tx_ctl.xdc`. The same values can be set in the Properties windows of the XDC file in the Vivado IDE.

```
# Using the reference module name only:
set_property SCOPED_TO_REF uart_tx_ctl [get_files uart_tx_ctl.xdc]

# Using the cell name only:
set_property SCOPED_TO_CELLS uart_tx_i0/uart_tx_ctl_i0 [get_files uart_tx_ctl.xdc]

# Using both the uart_tx reference module and uart_tx_ctl_i0 instance:
set_property SCOPED_TO_REF uart_tx [get_files uart_tx_ctl.xdc]
set_property SCOPED_TO_CELLS uart_tx_ctl_i0 [get_files uart_tx_ctl.xdc]
```

When using Vivado Design Suite in Non-Project Mode, you can use the `read_xdc` command with the `-ref` and `-cells` options to achieve the same result:

```
# Using the reference module name only:
read_xdc -ref uart_tx_ctl uart_tx_ctl.xdc
# Using the cell name only:
read_xdc -cells uart_tx_i0/uart_tx_ctl_i0 uart_tx_ctl.xdc
# Using both the uart_tx reference module and uart_tx_ctl_i0 instance
read_xdc -ref uart_tx -cells uart_tx_ctl_i0 uart_tx_ctl.xdc
```

When a module is instantiated multiple times in the design, the module is uniquified during synthesis. After the synthesis, each instance of the RTL module points to a different module name. To apply some XDC constraints to all the instances of the original RTL module, the property `ORIG_REF_NAME` should be used instead of the property `REF_NAME`. For example:

```
set_property SCOPED_TO_REF [get_cells -hierarchical -filter {ORIG_REF_NAME ==
uart_tx_ctl}] [get_files uart_tx_ctl.xdc]
read_xdc -ref [get_cells -hierarchical -filter {ORIG_REF_NAME == uart_tx_ctl}]
uart_tx_ctl.xdc
```

Except for ports, constraints scoping relies on the `current_instance` mechanism, which is part of the Synopsys Design Constraints (SDC) standard. When setting the scope to a lower level of the design hierarchy with the `current_instance` command, only the objects included in that level or below can be returned by the object query commands.

The only exceptions are with timing clock objects and netlist ports:

- Timing clocks are defined by `create_clock` or `create_generated_clock`. They are visible throughout the design regardless of the current instance setting. The `get_clocks` command can query clocks that are not present in the current instance, or that propagate beyond the current instance. Xilinx does not recommend defining timing exceptions on clocks when creating scoped constraints unless they are fully contained in the current instance. For a clock to be available for reference in an XDC, the clock must have already been defined. This might require changing the order of the XDC files in the project.
- Top-level ports are returned by the `get_ports` command when the scope is set to a lower level instance with the `current_instance` command. But when reading an XDC file scoped to a lower-level instance with the `read_xdc -ref/-cells` command or when loading a design after setting the SCOPED_TO_REF/SCOPED_TO_CELLS file properties, the `get_ports` command behavior is different:
  - The port names to be used with `get_ports` are the port names of the scoped instance interface, not the top-level port names.
  - If a scoped instance port is directly connected to a top-level port through the hierarchy of the design, the top-level port is returned by the `get_ports` command and the constraint is applied to the top-level port.
  - If there is any leaf cell, including IO and clock buffers, between the scoped instance port and the top-level ports, the `get_ports` command becomes a `get_pins` command and returns the hierarchical scoped instance pin.

The XDC scoping mechanism is used for reading all Vivado Design Suite IP constraint files. Figure 2-51, and Figure 2-52, show the two examples of how the `get_ports` commands are treated when reading in the IP-level XDC using this methodology.

In Figure 2-51, the I/O buffer is instantiated inside the IP and the IP interface pin is directly connected to a top-level port (regardless of the hierarchy). When the XDC for the IP is applied, the argument of the `get_ports` command is automatically replaced with the top-level port.

This enables setting physical properties such as a LOC or IOSTANDARD at the IP level and having them be placed on the top-level port where they need to be. This is accomplished without the IP knowing the name of the top-level ports of the design.

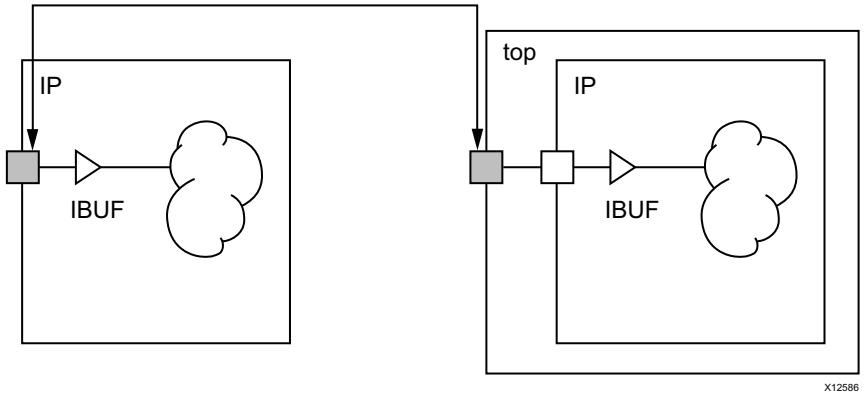


Figure 2-51:

In Figure 2-52, the IP does not contain an I/O buffer, so the synthesis engine infers one between the IP interface pin and the top-level port. Consequently, the `get_ports` is converted to a `get_pins` of the IP interface pin (for example, a hierarchical pin) when the XDC is applied.

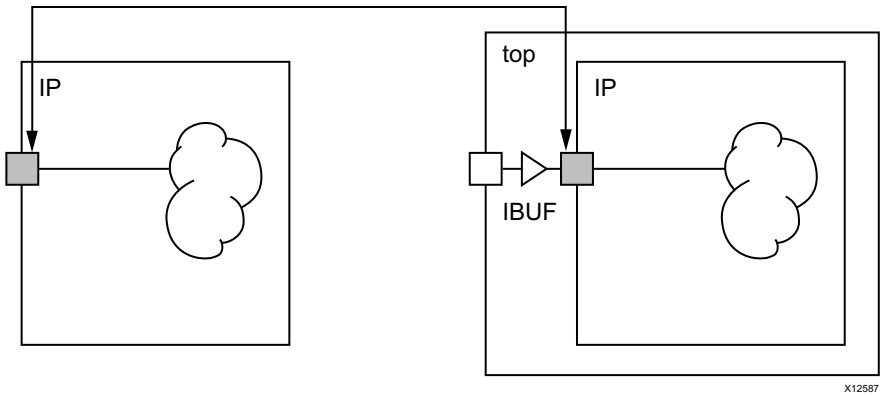


Figure 2-52:

This capability is very useful for creating constraints on the interface of an IP or a sub-level module without knowing the names of the top-level design.

If the scoped XDC file includes constraints that can only be applied to top-level ports but the IP instance is not directly connected to top-level ports, the Vivado Design Suite XDC reader will return errors. For example, the following constraints can only be applied to top-level ports, and not hierarchical pins of your design:

- `set_input_delay/set_output_delay`
- `set_property IOSTANDARD`

When using Package IP to create IP and use it from the Vivado IP catalog, XDC constraints can also be packaged for inclusion. Any IP in the Vivado Design Suite is plug-and-play, that is, the IP does not require a sample project from which you must cut and paste constraints to complete your top-level design constraints. Instead, the IP can be packaged with an XDC file that was developed for the IP as if it were a stand alone, top-level design. The Vivado tools take care of reading the constraints appropriately when the IP is instantiated in the project using the IP catalog.

Similarly, you can develop constraints for a sub-module of your design, and use the same scoping mechanism as IP cores by setting the `SCOPED_TO_REF/SCOPED_TO_CELLS XDC` file properties appropriately in a project flow, or use the `read_xdc -ref/-cells` command in Non-Project Mode.

For this flow to work smoothly, the XDC constraints must be written so that the effects of the constraints stay local to the IP or sub-module instance. The Vivado tools can set the scope of queries to a specific level of the hierarchy as seen previously in Constraints Scoping, page 68. When developing constraints for an IP or a sub-level module, you must understand the behavior of the query commands:

- Cell/net/pin objects queries are limited to the scoped instance and its sub-levels:
  - `get_cells/get_nets/get_pins <name pattern>`
  - The `NAME` property of the object shows the full hierarchical path of the object relative to the top-level and not just the scoped instance. If you use the `-filter` option of the `get_*` commands on the `NAME` property, you must use the glob string match operator and provide a pattern which starts with a `*`. For example:

    ```
    get_nets -hierarchical -filter {NAME =~ *clk}
    ```
- `get_ports` returns a top-level port if the port of the block/IP is directly connected to a top-level port. Otherwise, `get_ports` returns a hierarchical pin
- Netlist helper commands are also scoped:
  - `all_ffs`, `all_latches`, `all_rams`, `all_registers`, `all_dsps`, `all_hsios` return only instances included in the current instance.
- IO helper commands *cannot* be used at all in a scoped XDC:
  - `all_inputs`, `all_outputs`
- Clock commands are *not* scoped and will return all timing clocks of your design.
  - `get_clocks`, `all_clocks`

- Top-level and local clock objects can be queried by probing the netlist with `get_clocks -of_objects`.
  - Retrieve a clock entering the current instance by using `get_clocks -of_objects [get_ports <interfacePinName>]`
  - Retrieve a clock automatically generated inside the current instance by using `get_clocks -of_objects [get_pins <instName/outPin>]`, where `instName` is a clock generator instance.
- Querying any object in the design is possible using the `-of_objects` option:
  - Example: `get_pins -leaf -of_objects [get_nets local_net]`
- Queries are supported for top-level ports connected to the current instance interface nets:
  - `get_ports -of_objects [get_nets <scoped_instance_net>]`
- Queries of IP/sub-module interface pins are not allowed:
  - `"get_pins clk"` returns an error.
- Path tracing commands are also scoped:
  - `all_fanin/all_fanout` traverses the scoped design and stops at its boundary.
- Use `get_cells/get_pins/get_nets` with the most specific pattern instead of using the `all_registers` command with the `-clock` option to query all the cells connected to a particular clock. The returned list can be very large while only a few objects need to be constrained. This can impact the runtime negatively.

To avoid negatively impacting the top-level design, it is important to make sure that timing constraints written for the IP or sub-module do not propagate beyond its boundary, except for clock definition in some cases.

For example, consider the case in which a false path constraint is defined in the IP XDC between two clocks that come into the IP. The IP includes proper circuitry for asynchronous clock boundaries, but perhaps not for the rest of the design. This is a problem if the two clocks are related and must be timed together in the rest of the design in order to have proper hardware functionality.

Also, as discussed in Chapter 7, XDC Precedence, a timing exception defined in the IP XDC file can have higher precedence than top-level constraints and can override them, which is undesired. To avoid this situation, Xilinx recommends that you apply the constraints to netlist objects local to the IP. In the case of a false path between two global clocks, the false path must be applied from a group of startpoint cells inside the IP to another group of endpoint cells inside the IP as well. This technique is referred to as point-to-point exceptions instead of global exceptions.

The block-level constraints must comply with the following rules:

1. Do not define clocks in the block-level constraints if they are expected to be created at the top level of the design.

   Instead they can be queried inside the block using the `get_clocks -of_objects` command. This command returns all the clocks that traverse a particular object in the design.

   Example:

   ```
   set blockClock [get_clocks -of_objects [get_ports clkIn]]
   ```

   If a clock needs to be defined inside the block, it must be on an input/inout port that is driving an instantiated input/inout buffer, or on the output of a cell that creates/transforms a clock (except for MMCM/PLL or special buffers that are automatically handled by the timing tools).

   Examples:

   - Input clock with input buffer
   - Clock Divider
   - GT recovered clock

2. Specify input and output delay only if the port is directly connected to the top-level port and the I/O buffer is instantiated inside the IP.

   Example:

   - Input data ports with input buffers
   - Output data ports with output buffers

3. Do not define timing exceptions between two clocks that are not bounded to the IP.
4. Do not refer to clocks by name as the name may vary based on the top-level clock names or if the block is instantiated multiple times.
5. Do not add placement constraints if the block can be instantiated multiple times in a same top-level design.

When writing timing constraints, it is important to keep the constraints simple and specify them on the relevant netlist objects only. Inefficient constraints result in larger runtime and larger memory consumption. Inefficient constraints can also result in a design improperly constrained as timing exceptions can unexpectedly cover more paths than expected and collide with other constraints.

A timing constraint is efficient when the number of objects provided to the constraint is as small as possible to accurately and safely cover the desired timing paths. Most of the time, the full efficiency cannot be obtained as the list of objects are typically built from some pins or cells name patterns. However, the minimum number of objects should always be the target when building the list of objects for a timing exception.

Vivado provides several ways to get feedback on the timing exceptions:

- The methodology check XDCB-1 (`report_methodology`) reports the timing constraints that reference large collections of objects (over 1000).
- The Report Exception command (`report_exceptions`) provides coverage and collision information on the timing exceptions that have been defined.

Xilinx recommends that you carefully analyze the following reports:

- `report_exceptions -coverage`

  This report provides a logical path coverage for each timing exception. The number of objects passed to the timing exception are compared to the number of startpoints and endpoints effectively covered. You should review constraints that have significant differences between the number of objects and the number of startpoints/endpoints.

- `report_exceptions -ignored`

  This report provides the list of timing constraints overridden by other timing constraints (for example, a `set_false_path` overridden by `set_clock_group`). You should review the overridden constraints for correctness or remove the useless constraints.

- `report_exceptions -ignored_objects`

  This report provides the list of startpoints and endpoints that are ignored due to, for example, inexistent paths from those startpoints or to those endpoints.

Optimizing Pin Queries

Since there are several times mores pins than cells in the design, using `get_pins` instead of `get_cells` can have a significant impact on the runtime. The runtime degradation can be experienced when processing XDC constraints (for example, `open_checkpoint` runtime) or when executing a Tcl script. Xilinx recommends leveraging the relationship between pin and cell objects to improve the runtime for large number of pin queries.

Instead of finding a list of pins based on their names among all pins in the design, it is more efficient to first find the cells of the desired pins, and then refine the query by filtering the desired pins of the cells returned by the first query, as described below.

Original pin query:

```
get_pins -hier * -filter {NAME=~xx*/yy*}
```

Recommended efficient pin query:

```
get_pins -filter {REF_PIN_NAME=~yy*} -of [get_cells -hier xx*]
```

Alternate recommended pin query:

```
get_pins -filter {REF_PIN_NAME=~yy*} -of [get_cells -hier * -filter {NAME=~xx*}]
```

For example, consider the following constraint:

```
set_max_delay 15 -from [get_pins -hier -filter {NAME=~*/aclk_dpram_reg*/*/CLK}] \
                 -to   [get_cells -hier -filter {NAME=~*/bclk_dout_reg*}] \
                 -datapath_only
```

The constraint above can be re-written as follows to significantly improve the query runtime, especially for larger designs:

```
set_max_delay 15 -from [get_pins -of [get_cells -hier -filter
                     {NAME =~ *aclk_dpram_reg*/*}] -filter {REF_PIN_NAME == CLK}] \
                 -to [get_cells -hier bclk_dout_reg*] \
                 -datapath_only
```

The following are some additional query recommendations:

- Avoid queries using `all_registers` whenever possible, as they tend to create large collections of objects. Such queries should be replaced by cells/pins queries with appropriate name patterns.
- When `all_registers` must be used and the query is gathering all the sequential elements from a clock domain, `all_registers -clock` can sometimes have equivalent coverage as directly using a clock object.

  For example, the two commands below are equivalent in terms of coverage. However, the second form using `get_clocks` is far more efficient because the multicycle path constraint references a single clock object instead of potentially hundreds of thousands of sequential elements.

  Original:

  ```
  set_multicycle_path -from [all_inputs] -to [all_registers -clock clk1]
  ```

  Optimal:

  ```
  set_multicycle_path -from [all_inputs] -to [get_clocks clk1]
  ```

IMPORTANT: *Starting with the Vivado Design Suite 2018.3, the `all_registers` command only returns primitives that have at least one Setup/Hold/Recovery/Removal timing arc that is enabled and a CLK->Q timing arc. This means that buffers such as BUFGCE and BUFGCE_DIV are not returned anymore by the `all_registers` command.*

When loading the timing constraints in memory, the timing engine validates each new constraint and prints messages to flag potential problems. Some timing constraints partially invalidate the timing database (also referred as timing graph) and some other timing constraints require an up-to-date timing database in order to be properly applied. Once the timing database is out of date, subsequent timing updates are needed, for instance, to update auto-derivation clocks or to disable certain timing paths in the design. The XDC commands which query the clocks or which traverse the design to query netlist objects require an up-to-date timing database.

Interleaving constraints and commands that impact the timing database state can be runtime intensive as the timing information gets invalidated and updated multiple times.

For runtime optimization, Xilinx recommends that you order the timing constraints and queries carefully. The table below lists the XDC constraints and commands that have an impact on the timing graph.

Table 2-5:

create_clock	set_bus_skew	all_fanout
create_generated_clock	set_clock_groups	all_fanin
set_case_analysis	set_clock_latency	get_clocks
set_clock_sense	set_false_path	get_generated_clocks
set_clock_uncertainty	set_input_delay	all_clocks
set_disable_timing	set_input_jitter	Any constraint with the –clock option
set_external_delay	set_min_delay	
set_propagated_clock	set_max_delay	
	set_max_time_borrow	
	set_multicycle_path	
	set_system_jitter	

One of the most runtime intensive combinations is `set_disable_timing` with `all_fanout` or `all_fanin`. Such combinations should be avoided. For example:

```
set_disable_timing –from <pin> -to [all_fanout …]
set_disable_timing –from [all_fanin …] -to <pin>
```

Based on Table 2-5 above, the optimal constraints order for runtime optimization is:

1. XDC constraints `set_disable_timing`, `set_case_analysis`, and `set_external_delay`.
2. Constraints that have an impact on the timing graph.
3. Constraints that do not require timing graph updates.

TIP: *When the same query is done in multiple places, it is recommended that you save the result of the query inside a Tcl variable and refer to that Tcl variable when it is needed.*

For example, the following sequence of constraints is not optimal.

```
create_clock –name clk1
create_generated_clock –name genclk1 –master_clock [get_clocks -of [get_pins ...]]
set_disable_timing ...
create_clock –name clk2
set_false_path -from [get_clocks -of [get_pins ff1/C]]
set_case_analysis ...
create_clock –name clk3
set_max_delay -to [get_clocks -of [get_pins ff2/C]]
```

The following shows a more optimal and runtime efficient sequence.

```
set_disable_timing ...
set_case_analysis ...
create_clock -name clk1
create_clock -name clk2
create_clock -name clk3
create_generated_clock -name genclk1 -master_clock [get_clocks -of [get_pins ...]]
set_false_path -from [get_clocks -of [get_pins ff1/C]]
set_max_delay -to [get_clocks -of [get_pins ff2/C]]
```

Defining Clocks

In digital designs, clocks represent the time reference for reliably transferring data from register to register. The Xilinx® Vivado® Integrated Design Environment (IDE) timing engine uses the clock characteristics to compute timing path requirements and report the design timing margin by means of the slack computation.

For more information, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

Clocks must be properly defined in order to get the maximum timing path coverage with the best accuracy. The following characteristics define a clock:

- It is defined on the driver pin or port of its tree root, which is called the source point.
- Its edges are described by the combination of the period and the waveform properties.
- The period is specified in nanoseconds. It corresponds to the time over which the waveform repeats.
- The waveform is the list of rising edge and falling edge absolute times, in nanoseconds, within the clock period. The list must contain an even number of values. The first value always corresponds to the first rising edge. Unless specified otherwise, the duty cycle defaults to 50% and the phase shift to 0ns.

As shown in Figure 3-1, the clock `Clk0` has a 10 ns period, a 50% duty cycle and 0 ns phase. The clock `Clk1` has 8 ns period, 75% duty cycle (high time is 6 ns out of 8 ns) and a 2 ns rising edge phase shift.

```
Clk0: period = 10, waveform = {0 5}
Clk1: period = 8, waveform = {2 8}
```

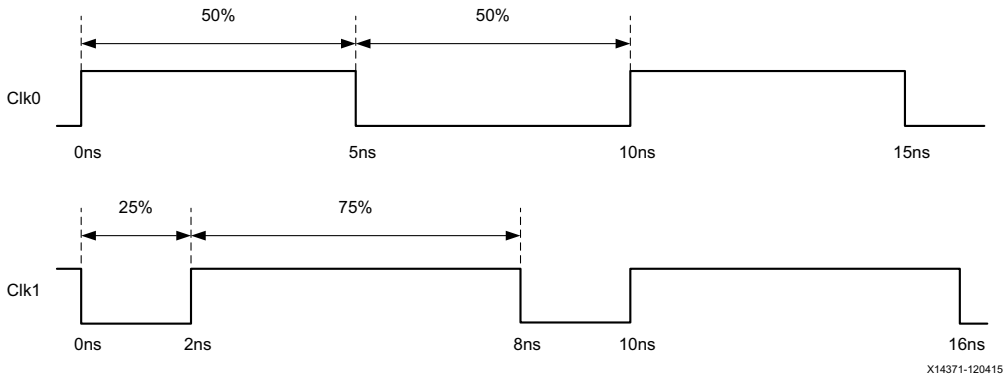


Figure 3-1:

The period and waveform properties represent the ideal characteristics of a clock. When entering the FPGA and propagating through the clock tree, the clock edges are delayed and become subject to variations induced by noise and hardware behavior. These characteristics are called clock network latency and clock uncertainty.

The clock uncertainty includes:

- Clock jitter (see Clock Jitter)
- Phase error
- Any additional uncertainty that you have specified (see Additional Clock Uncertainty)

By default, the Vivado IDE always treats clocks as propagated clocks, that is, non-ideal, in order to provide an accurate slack value which includes clock tree insertion delay and uncertainty.

The dedicated hardware resources of Xilinx FPGAs efficiently support a large number of design clocks. These clocks are usually generated by an external component on the board. They usually enter the device through an input port.

They can also be generated by special primitives called Clock Modifying Blocks, such as:

- MMCM
- PLL
- BUFR

They can also be transformed by regular cells such as LUTs and registers.

The following sections describe how to best define clocks based on where they originate.

A primary clock is a board clock that enters the design through an input port or a gigabit transceiver output pin (for example, a recovered clock).

A primary clock can be defined only by the `create_clock` command.

Note: Primary clocks must be defined on a gigabit transceiver output only for Xilinx® 7 series FPGAs. For UltraScale and UltraScale+™ devices, the timer automatically derives clocks on the GT output ports.

A primary clock must be attached to a netlist object. This netlist object represents the point in the design from which all the clock edges originate and propagate downstream on the clock tree. In other words, the source point of a primary clock defines the time zero used by the Vivado IDE when computing the clock latency and uncertainty used in the slack equation.

IMPORTANT: *The Vivado IDE ignores all clock tree delays coming from cells located upstream from the point at which the primary clock is defined. If you define a primary clock on a pin in the middle of the design, only part of its latency is used for timing analysis. This can be a problem if this clock communicates with other related clocks in the design, because the skew, and consequently the slack, value between the clocks can be inaccurate.*

Primary clocks must be defined first, because other timing constraints often refer to them.

As shown in Figure 3-2, the board clock enters the device through the port `sysclk`, then propagates through an input buffer and a clock buffer before reaching the path registers.

- Its period is 10 ns.
- Its duty cycle is 50%.
- Its phase is not shifted.

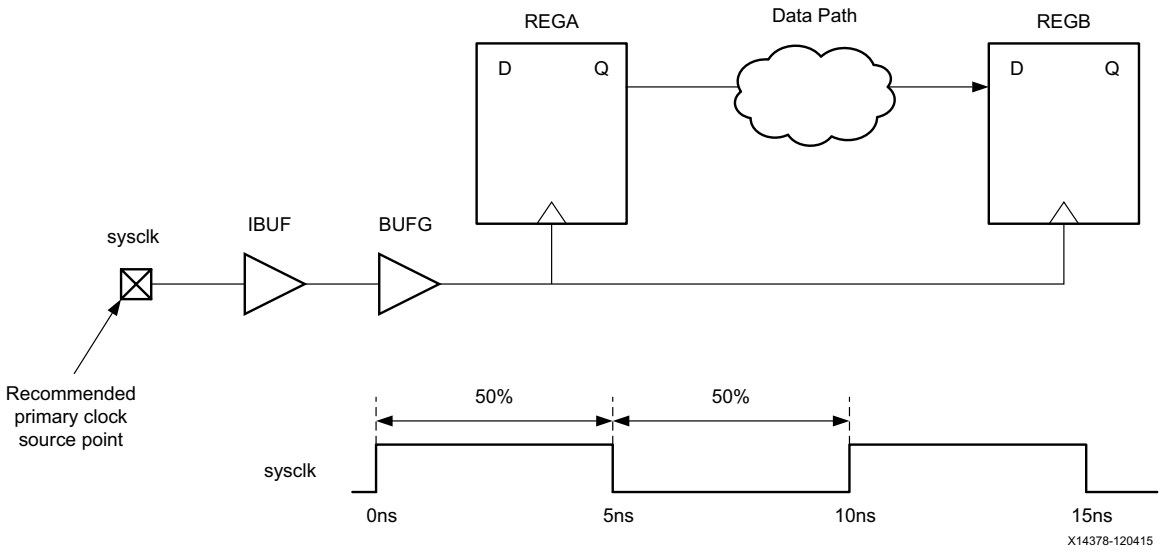


Figure 3-2:

RECOMMENDED: *Define the board clock on the input port, not on the output of the clock buffer.*

Corresponding Xilinx Design Constraints (XDC):

```
create_clock -period 10 [get_ports sysclk]
```

Similar to `sysclk`, a board clock `devclk` enters the device through the port `ClkIn`.

- Its period is 10 ns.
- Its duty cycle is 25%.
- It is phase shifted by 90 degrees.

Corresponding XDC:

```
create_clock -name devclk -period 10 -waveform {2.5 5} [get_ports ClkIn]
```

Figure 3-3 shows a transceiver gt0, which recovers the clock `rxclk` from a high speed link on the board. The clock `rxclk` has a 3.33 ns period, a 50% duty cycle and is routed to an MMCM, which generates several compensated clocks for the design.

When defining `rxclk` on the output driver pin of `GT0`, all the generated clocks driven by the MMCM have a common source point, which is `gt0/RXOUTCLK`. The slack computation on paths between them uses the proper clock latency and uncertainty values.

```
create_clock -name rxclk -period 3.33 [get_pins gt0/RXOUTCLK]
```

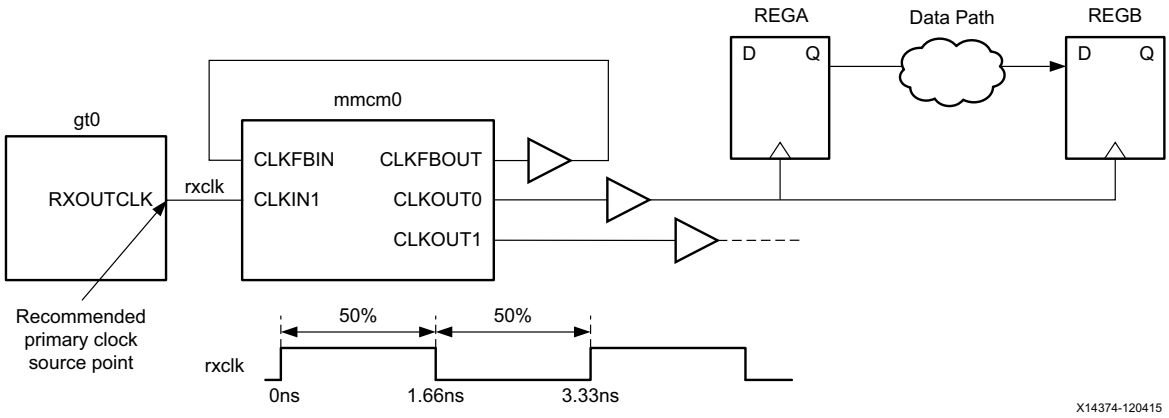


Figure 3-3:

In Figure 3-4, a differential buffer drives the PLL. In such a scenario, the primary clock must only be created on the positive input of the differential buffer. Creating a primary clock on each of the positive/negative inputs of the buffer would result in unrealistic CDC paths. For example:

```
create_clock -name sysclk -period 3.33 [get_ports SYS_CLK_clk_p]
```

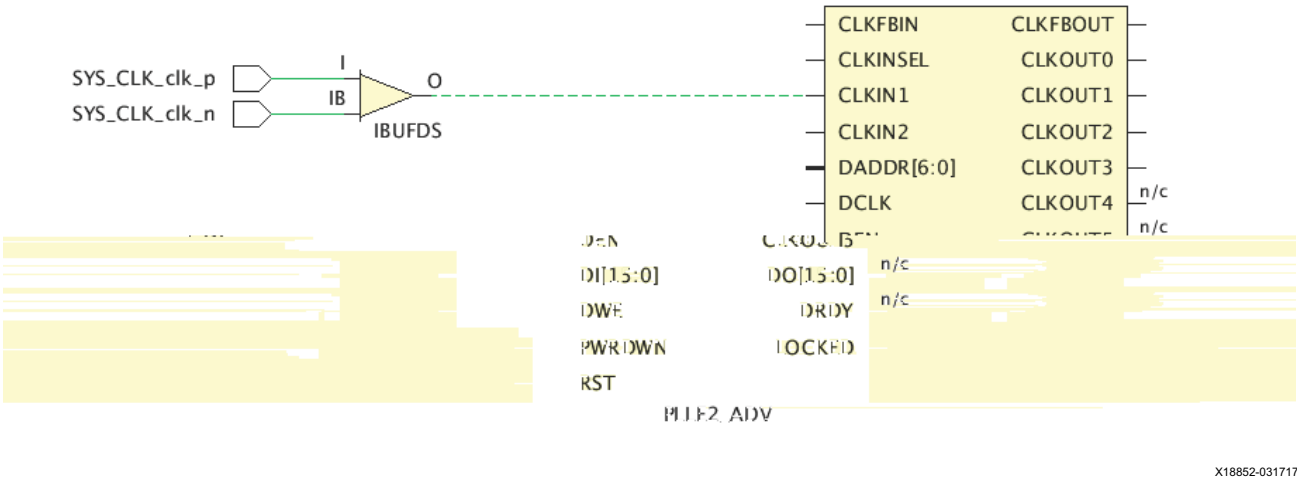


Figure 3-4:

A virtual clock is a clock that is not physically attached to any netlist element in the design.

A virtual clock is defined by means of the `create_clock` command without specifying a source object.

A virtual clock is commonly used to specify input and output delay constraints in one of the following situations:

- The external device I/O reference clock is not one of the design clocks.
- The FPGA I/O paths are related to an internally generated clock that cannot be properly timed against the board clock from which it is derived.

  Note: This happens when the ratio between the two periods is not an integer. which leads to a very tight and unrealistic timing path requirement.
- You want to specify different jitter and latency only for the clock related to the I/O delay constraints without modifying the internal clocks characteristics.

For example, the clock `clk_virt` has a period of 10 ns and is not attached to any netlist object. The `[<objects>]` argument is not specified. The `-name` option is mandatory in such cases.

```
create_clock -name clk_virt -period 10
```

The virtual clocks must be defined before being used by the input and output delay constraints.

This section discusses generated clocks and includes:

- About Generated Clocks
- User Defined Generated Clocks
- Automatically Derived Clocks
- Renaming Auto-Derived Clocks

Generated clocks are driven inside the design by special cells called Clock Modifying Blocks (for example, an MMCM), or by some user logic.

Generated clocks are associated with a master clock. The `create_generated_clock` command considers the start point of the master clock. The master clock can be a primary clock or another generated clock.

Generated clock properties are directly derived from their master clock. Instead of specifying their period or waveform, you must describe how the modifying circuitry transforms the master clock.

The relationship between a master clock and a generated clock can be any of the following:

- A simple frequency division
- A simple frequency multiplication
- A combination of a frequency multiplication and division in order to obtain a non-integral ratio (usually done by MMCM and PLL)
- A phase shift or a waveform inversion
- A duty cycle transformation
- A combination of all the above

RECOMMENDED: *Define all primary clocks first. They are needed for defining the generated clocks.*

Note: To compute the latency for the generated clock, the tool traces both sequential and combinational paths between the source pin of the generated clock and the source pin of the master clock. In some cases, it might be desirable to only trace through combinational paths to calculate the generated clock latency. You can do this using the `-combinational` command line option.

A user defined generated clock is:

- Defined by the `create_generated_clock` command.
- Attached to a netlist object, preferably the clock tree root pin.

Specify the master clock using the `-source` option. This indicates a pin or port in the design through which the master clock propagates. It is common to use the master clock source point or the input clock pin of generated clock source cell.

IMPORTANT: *The `-source` option accepts only a pin or port netlist object. It does not accept clock objects.*

Example One: Simple Division by 2

The primary clock `clkin` has a period of 10 ns. It is divided by 2 by the register REGA which drives other registers clock pin. The corresponding generated clock is called `clkdiv2`.

Two equivalent constraints are provided below:

```
create_clock -name clkin -period 10 [get_ports clkin]

# Option 1: master clock source is the primary clock source point
create_generated_clock -name clkdiv2 -source [get_ports clkin] -divide_by 2 \
  [get_pins REGA/Q]

# Option 2: master clock source is the REGA clock pin
create_generated_clock -name clkdiv2 -source [get_pins REGA/C] -divide_by 2 \
  [get_pins REGA/Q]
```

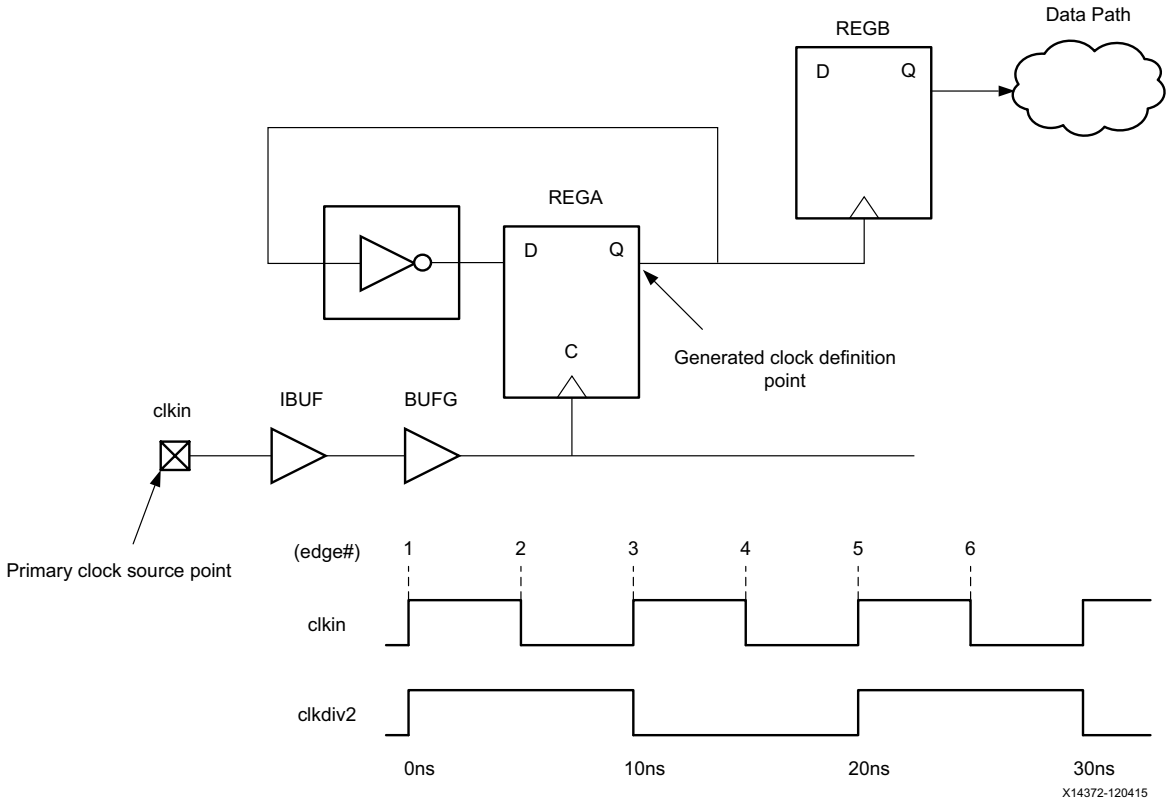


Figure 3-5:

Example Two: Division by 2 With the -edges Option

Instead of using the `-divide_by` option, you can use the `-edges` option to directly describe the waveform of the generated clock based on the edges of the master clock. The argument is a list of master clock edge indexes used for defining the position in time of the generated clock edges, starting with the rising clock edge.

The following example is equivalent to the generated clock defined in Example One: Simple Division by 2.

```
# waveform specified with -edges instead of -divide_by
create_generated_clock -name clkdiv2 -source [get_pins REGA/C] -edges {1 3 5} \
  [get_pins REGA/Q]
```

Example Three: Duty Cycle Change and Phase Shift with -edges and -edge_shift Options

Each edge of the generated clock waveform can also be individually shifted by a positive or negative value by using the `-edge_shift` option. Use this option only if a phase shift is needed.

The `-edge_shift` option cannot be used at the same time as any of the following:

- `-divide_by`
- `-multiply_by`
- `-invert`

Consider the master clock `clkin` with a 10 ns period and a 50% duty cycle. It reaches the cell `mmcm0` which generates a clock with a 25% duty cycle, shifted by 90 degrees. The generated clock definition refers to the master clock edges 1, 2, and 3. These edges respectively occur at 0ns, 5ns, and 10ns. To obtain the desired waveform, shift the first and the third edges by 2.5ns.

```
create_clock -name clkin -period 10 [get_ports clkin]
create_generated_clock -name clkshift -source [get_pins mmcm0/CLKIN] -edges {1 2 3} \
   -edge_shift {2.5 0 2.5} [get_pins mmcm0/CLKOUT]
# First rising edge:   0ns + 2.5ns = 2.5ns
# Falling edge:        5ns + 0ns   = 5ns
# Second rising edge: 10ns + 2.5ns = 12.5ns
```

Note: The `-edge_shift` values can be positive or negative.

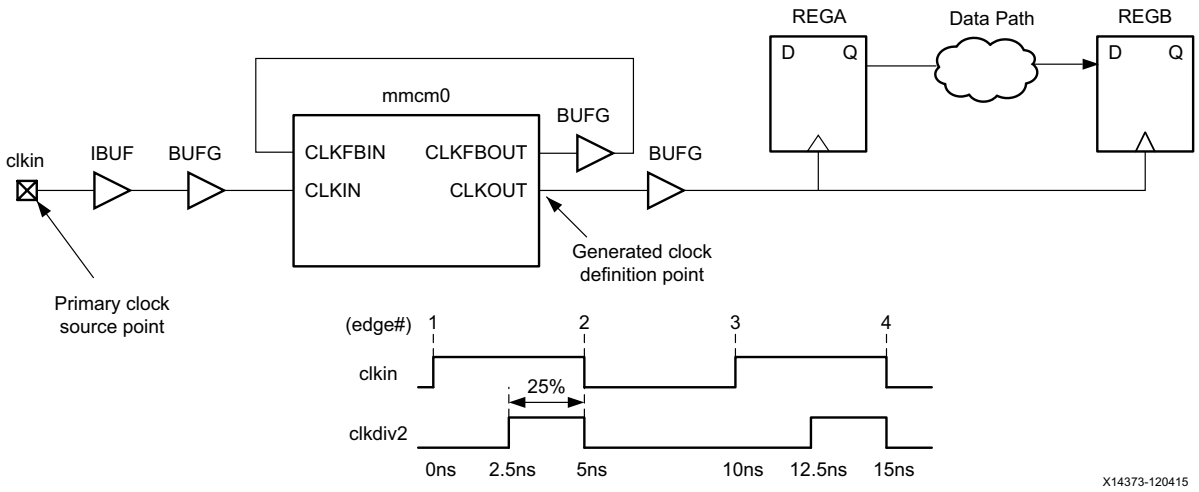


Figure 3-6:

Example Four: Using Both -divide_by and -multiply_by at the Same Time

The Vivado IDE allows you to specify both `-divide_by` and `-multiply_by` at the same time. This is an extension to standard Synopsys Design Constraints (SDC) support. This is

particularly convenient for manually defining clocks generated by MMCM or PLL instances, although Xilinx recommends that you let the engine create these constraints automatically.

For more information, see Automatically Derived Clocks.

Consider the `mmcm0` cell as in Example Three: Duty Cycle Change and Phase Shift with -edges and -edge_shift Options above, and assume that it multiplies the frequency of the master clock by 4/3. The corresponding generated clock definition is:

```
create_generated_clock -name clk43 -source [get_pins mmcm0/CLKIN] -multiply_by 4 \
   -divide_by 3 [get_pins mmcm0/CLKOUT]
```

If you create a generated clock constraint on the output of an MMCM or PLL, it is better to verify that the waveform definition matches the configuration of the MMCM or PLL.

Example Five: Tracing the Master Clock through Combinational Arcs Only

In this example, assume that the master clock drives both a register-based clock divided-by-2 and a clock multiplexer that can select the master clock or the divided-by-2 clock from the register clock divider. In this scenario, there are two paths from the master clock to the generated clock, which are through a sequential arc and through a combinational arc. We want to create a generated clock on the multiplexer output that reflects the latency of the combinational path from the master clock through the multiplexer. This is done by using the `-combinational` command line option:

```
create_generated_clock -name clkout -source [get_pins mmcm0/CLKIN] -combinational
[get_pins MUX/O]
```

Example Six: Forwarded Clock Driven by ODDR

In this example, a forwarded clock is created on the output port driven by an ODDR cell. The forwarded clock references the master clock driving the ODDR/CLKDIV pin and has the same period as the master clock (`-divide_by 1`):

```
create_generated_clock -name ck_vsf_clk_2 \
    -source [get_pins ODDRE1_vsfclk2_inst/CLKDIV] -divide_by 1 [get_ports vsf_clk_2]
```

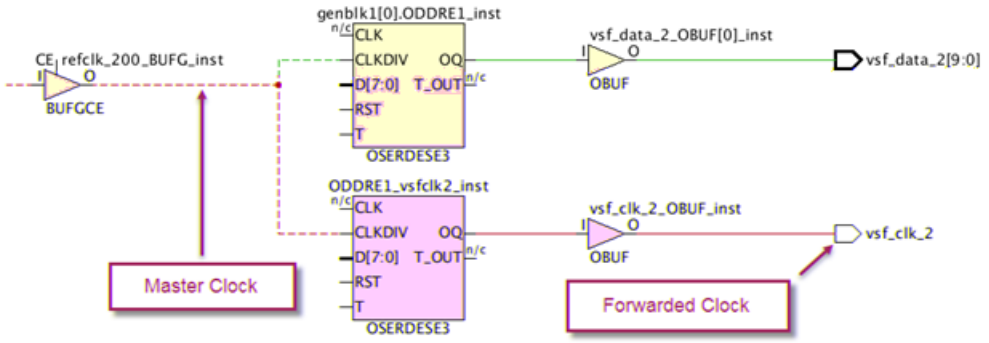


Figure 3-7:

Automatically derived clocks are also called auto-generated clocks. The Vivado IDE automatically creates the constraint for these on the output pins of the Clock Modifying Blocks (CMBs), provided the associated master clock has already been defined.

In the Xilinx 7 series device family, the CMBs are:

- MMCM*/ PLL*
- BUFR
- PHASER*

In the Xilinx UltraScale™ device family, the CMBs are:

- MMCM* / PLL*
- BUFG_GT / BUFGCE_DIV
- GT*_COMMON / GT*_CHANNEL / IBUFDS_GTE3
- BITSLICE_CONTROL / RX*_BITSLICE
- ISERDESE3

An auto-generated clock is not created if a user-defined clock (primary or generated) is also defined on the same netlist object, that is, on the same definition point (net or pin). The auto-derived clock is named with the segment name in the top-most hierarchy of the net that is connected to the definition point.

Automatically Derived Clock Example

The following automatically derived clock example is a clock generated by an MMCM.

The master clock `clkin` drives the input `CLKIN` of the MMCME2 instance `clkip/mmcm0`. The name of the auto-generated clock is `cpuClk` and its definition point is `clkip/mmcm0/CLKOUT`.

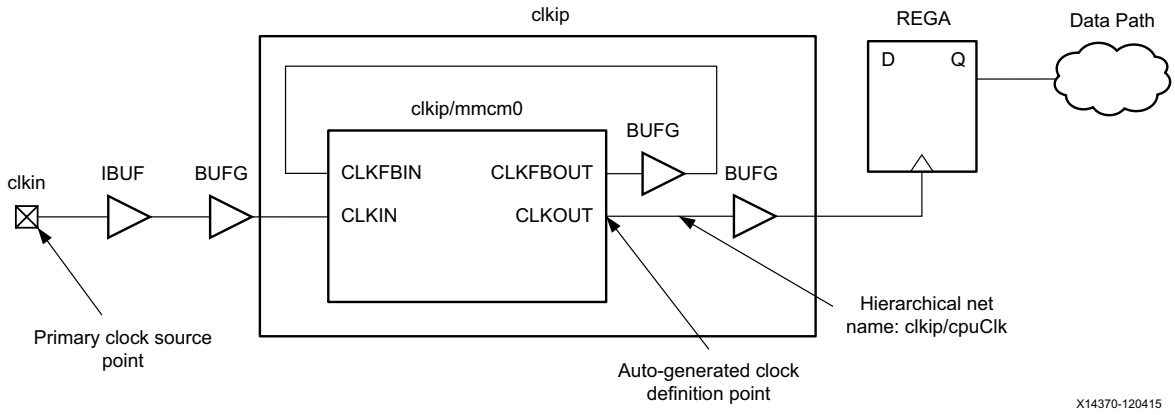


Figure 3-8:

TIP: *Use the `get_clocks -of_objects <pin/port/net>` command to query an auto-generated clock without knowing its name. This will make your constraint or script independent of the clock name changes.*

Local Net Names

If the CMB instance is located inside the hierarchy of the design, the local net name (that is, the name without its parent cell name) is used for the generated clock name.

For example, for a hierarchical net called `clkip/cpuClk`:

- The parent cell name is `clkip`.
- The generated clock name is `cpuClk`.

Name Conflicts

In case of name conflict between two auto-generated clocks, the Vivado IDE adds unique suffixes to differentiate them, such as:

- `usrclk`
- `usrclk_1`
- `usrclk_2`
- `...`

To force the name of the generated clocks:

- Choose unique and relevant net names in the RTL, or
- Use `create_generated_clock` to force the name of the generated clocks.

It is possible to rename the generated clocks that are automatically created by the tool. The renaming process consists in calling the `create_generated_clock` command with a limited number of parameters:

```
create_generated_clock -name new_name [-source master_pin] [-master_clock
master_clk] source_object
```

The arguments that must be specified are the new generated clock name and the source object of the generated clock. The source object of the generated clock is the object where the auto-derived clock is created (CMB output pin, GT output pin for UltraScale, and so on). The `-source` and `-master` parameters must be used only when more than one clock propagates through the source pin in order to remove any ambiguity.

IMPORTANT: *If any of the `-edges`/`-edge_shift`/`-divide_by`/`-multiply_by`/`-combinational`/`-duty_cycle`/`-invert` options is passed to the `create_generated_clock` command, the generated clock is not renamed. Instead a new generated clock is created with the specified characteristics.*

IMPORTANT: *When a module (IP/BD/DFx/...) is synthesized Out-Of-Context, the module is inferred as a black-box when the top level is synthesized and the module internal pins and clock names are not anymore accessible. In that scenario, the top level XDC constraints used for synthesis cannot refer to a clock name or rename an auto-derived clock that is generated inside the module. With OOC synthesis, the top-level timing constraints must point to the OOC clocks through the module ports that propagate those clocks. This can be done using some queries such as 'get_clocks -of_objects [get_pins <OOC_MODULE_OUTPUT_CLOCK_PORT>]. The XDC constraints used for implementation do not have this limitation since the entire design is rebuilt before the XDC constraints are applied.*

Limitations

- Auto-derived clocks can only be renamed at the pin where they originate, such as at the output of the Clock Modifying blocks (PLL, MMCM, . . .). For example, an auto-derived clock cannot be renamed at the output of a BUFG even though it propagates through it.
- Primary clocks or user-defined generated clocks cannot be renamed. Only auto-derived clocks can be renamed with this mechanism.
- The `source_object` must match the object where the auto-derived clock is created.

An error is returned if the tool cannot rename the generated clock. The master clock must also exist at the time the renaming is done. The auto-derived clocks can be renamed at any time inside the XDC, even after they have been referenced by some timing constraints.

For example, below is an abstract of `report_clocks` for the generated clock at the output pins of an MMCM:

```
====================================================
Generated Clocks
====================================================

Generated Clock    : clkfbout_clk_core
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Multiply By        : 1
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKFBOUT}

Generated Clock    : clk_rx_clk_core
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Multiply By        : 1
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT0}

Generated Clock    : clk_tx_clk_core
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Edges              : {1 2 3}
Edge Shifts        : {0.000 0.500 1.000}
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT1}
```

The three commands below illustrate the command line options that must be specified to rename the three auto-derived clocks at the output of the MMCM:

```
create_generated_clock -name clk_rx [get_pins
clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT0]
create_generated_clock -name clk_tx [get_pins
clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT1]
create_generated_clock -name clkfbout [get_pins
clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKFBOUT]
```

After the renaming, below is the abstract from the `report_clocks`:

```
====================================================
Generated Clocks
====================================================
Generated Clock    : clkfbout
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Multiply By        : 1
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKFBOUT}

Generated Clock    : clk_rx
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Multiply By        : 1
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT0}

Generated Clock    : clk_tx
Master Source      : clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKIN1
Master Clock       : clk_pin_p
Edges              : {1 2 3}
Edge Shifts        : {0.000 0.500 1.000}
Generated Sources  : {clk_gen_i0/clk_core_i0/inst/mmcm_adv_inst/CLKOUT1}
```

This section discusses Clock Groups and includes:

- About Clock Groups
- Clock Categories
- Asynchronous Clock Groups
- Exclusive Clock Groups

The Vivado IDE times the paths between all the clocks in your design by default, unless you specify otherwise by using clock groups or false path constraints. The `set_clock_groups` command disables timing analysis between groups of clocks that you identify, and not between the clocks within a same group. Unlike with the `set_false_path` constraint, timing is ignored on both directions between the clocks.

Multiple groups of clocks can be specified using the `-group` option multiple times. If none of the clocks in a group exist in the design, the group becomes empty. The `set_clock_groups` constraint stays valid only when at least two groups are valid and not empty. If only one group remains valid and all the other groups are empty, then the `set_clock_groups` constraint is not applied and an error message is generated.

Use the schematic viewer or the Clock Networks Report to visualize the topology of the clock trees, and determine which clocks must not be timed together. You can also use the Clock Interactions Report to review the existing constraints between two clocks, and determine whether they share the same primary clock -- that is, they have a known phase relationship -- or identify the clocks with no common period (unexpandable).

CAUTION! *Ignoring timing analysis between two clocks does not mean that the paths between them will work properly in hardware. In order to prevent metastability, you must verify that these paths have proper re-synchronization circuitry, or asynchronous data transfer protocols.*

This section discusses synchronous, asynchronous, and unexpandable clocks.

Synchronous Clocks

Two clocks are *synchronous* when their relative phase is predictable. This is usually the case when their tree originates from the same root in the netlist, and when they have a common period.

For example, a generated clock and its master clock that have a period ratio of 2 are *synchronous* because they propagate through the same netlist resources up to the generated clock source point, and have a common period of 2 cycles. They can be safely timed together.

Asynchronous Clocks

Two clocks are asynchronous when it is impossible to determine their relative phase.

For example, two clocks generated by separate oscillators on the board and entering the FPGA by means of different input ports have no known phase relationship. They must therefore be treated as asynchronous. If they were generated by the same oscillator on the board, this would not be true.

In most cases, primary clocks can be treated as asynchronous. When associated with their respective generated clocks, they form asynchronous clock groups.

Unexpandable Clocks

Two clocks are not expandable when the timing engine cannot determine their common period over 1000 cycles. In this case, the worst setup relationship over the 1000 cycles is used during timing analysis, but the timing engine cannot ensure this is the most pessimistic case.

This is typically the case between two clocks with an odd fractional period ratio. For example, consider two clocks, `clk0` and `clk1`, generated by two MMCMs that share the same primary clock:

- `clk0` has a 5.125 ns period.
- `clk1` has a 6.666 ns period.

Their rising clock edges do not realign within 1000 cycles. The timing engine uses a setup path requirement of 0.01 ns on the timing paths between the two clocks. Even if the two clocks have a known phase relationship at their clock tree root, their waveforms do not allow safe timing analysis between them.

As with asynchronous clocks, the slack computation appears normally, but the value cannot be trusted. For this reason, unexpandable clocks are often assimilated to asynchronous clocks. Both clock categories must be treated the same way for constraining and clock-domain crossing circuitry.

Asynchronous clocks and unexpandable clocks cannot be safely timed. The timing paths between them can be ignored during analysis by using the `set_clock_groups` command.

IMPORTANT: *The `set_clock_groups` command has higher priority over the regular timing exceptions. If you need to constrain and report some paths between asynchronous clocks, you must use the timing exceptions only, and not `set_clock_groups`.*

Asynchronous Clock Groups Examples

- The primary clock `clk0` is defined on an input port and reaches an MMCM which generates the clocks `usrclk` and `itfclk`.
- A second primary clock `clk1`

Such clocks are called *exclusive clocks*. Constrain them as such by using the options of `set_clock_groups`:

- `-logically_exclusive`
- `-physically_exclusive`

Exclusive Clock Groups Example

An MMCM instance generates `clk0` and `clk1` which are connected to the BUFGMUX instance `clkmux`. The output of `clkmux` drives the design clock tree.

By default, the Vivado IDE analyzes paths between `clk0` and `clk1` even though both clocks share the same clock tree and cannot exist at the same time.

You must enter the following constraint to disable the analysis between the two clocks:

```
set_clock_groups -name exclusive_clk0_clk1 -physically_exclusive \
  -group clk0 -group clk1
```

The following options are equivalent in the context of Xilinx FPGAs:

- `-physically_exclusive`
- `-logically_exclusive`

The physically and logically labels refer to various signal integrity analysis (crosstalk) modes in ASIC technologies which is not needed for Xilinx FPGAs.

In addition to defining the clock waveforms, you must specify predictable and random variations related to the operating conditions and environment.

After propagating on the board and inside the FPGA, the clock edges arrive at their destination with a certain delay. This delay is typically represented by:

- The source latency (delay before the clock source point, usually, outside the device)
- The network latency

The delay introduced by the network latency (also called insertion delay) is either automatically estimated (pre-route design) or accurately computed (post-route design).

Many non-Xilinx timing engines require the SDC command `set_propagated_clock` to trigger the computation of propagation delay along the clock trees. The Vivado tool does not require this command. Instead, it computes the clock propagation delay by default:

- All clocks are considered propagated clocks.
- A generated clock latency includes the insertion delay of its master clock plus its own network latency.

For Xilinx FPGAs, use the `set_clock_latency` command primarily to specify the clock latency outside the device.

set_clock_latency Example

```
# Minimum source latency value for clock sysClk (for both Slow and Fast corners)
set_clock_latency -source -early 0.2 [get_clocks sysClk]
# Maximum source latency value for clock sysClk (for both Slow and Fast corners)
set_clock_latency -source -late 0.5 [get_clocks sysClk]
```

Clock Jitter

For ASIC devices, clock jitter is usually represented with the clock uncertainty characteristic. However, for Xilinx FPGAs, the jitter properties are predictable. They can be automatically computed by the timing analysis engine, or be specified separately.

Input jitter is the difference between successive clock edges with respect to variation from the nominal or ideal clock arrival times. The input jitter is an absolute value and represents variations on each side of the clock edge.

Use the `set_input_jitter` command to specify input jitter for each primary clock individually. You cannot specify the input jitter on a generated clock directly. The Vivado IDE timing engine automatically computes the jitter that a generated clock inherits from its master clock.

- For the case in which the generated clock is driven by a MMCM or a PLL, the input jitter is replaced with a computed discrete jitter.
- For the case the generated clock is created by a combinatorial or sequential cell, the generated clock jitter is the same as its master clock jitter.

System jitter is the overall jitter due to power supply noise, board noise, or any extra jitter of the system.

Use the `set_system_jitter` command to set only one value for the whole design, that is, all the clocks.

The following command sets a +/-100 ps jitter on the primary clock propagating through input port clkin:

```
set_input_jitter [get_clocks -of_objects [get_ports clkin]] 0.1
```

Note: The impact of input jitter and system jitter in the overall calculation of the clock uncertainty is not trivial and does not follow a single equation. The calculation of the clock uncertainty is path-dependent and depends on the clocking topology, the clock-pair involved in the path, the presence or not of an MMCM/PLL on the clock tree, and other considerations. However, the text and GUI of the Report Timing command expose the breakdown of the clock uncertainty for each timing path.

Additional Clock Uncertainty

Use the `set_clock_uncertainty` command to define additional clock uncertainty for different corner, delay, or particular clock relationships as needed. This is a convenient way to add extra margin to a portion of the design from a timing perspective.

The inter-clock uncertainty always takes precedence over simple clock uncertainty, regardless of the order of the constraints. In the following example, although a simple clock uncertainty of 1.0 ns is defined last on clock `clk1`, the timing paths from clock `clk1` to clock `clk2` are constrained with a 2.0 ns clock uncertainty.

```
set_clock_uncertainty 2.0 -from [get_clocks clk1] -to [get_clocks clk2]
set_clock_uncertainty 1.0 [get_clocks clk1]
```

When an inter-clock uncertainty is defined between two clock domains, make sure to constrain all the possible interactions of clock domains:

- `clk1` to `clk2`
- `clk2` to `clk1`

XILINX

Chapter 4

Constraining I/O Delay

To accurately model the external timing context in your design, you must give timing information for the input and output ports. Because the Xilinx® Vivado® Integrated Design Environment (IDE) recognizes timing only within the boundaries of the FPGA, you must use the following commands to specify delay values that exist beyond these boundaries:

- `set_input_delay`
- `set_output_delay`

The `set_input_delay` command specifies the input path delay on an input port relative to a clock edge at the interface of the design.

VIDEO: *For training on input delay, see the* Vivado Design Suite QuickTake Video: Setting Input Delay.

When considering the application board, the input delay represents the phase difference between:

a. The data propagating from an external chip through the board to an input package pin of the FPGA, and

b. The relative reference board clock.

Consequently, the input delay value can be positive or negative, depending on the clock and data relative phase at the interface of the device.

Note: Input delays can also be set on internal data pins such as, STARTUPE3/DATA_IN[0:3] (UltraScale+™ devices).

Although the `-clock` option is optional in the Synopsys Design Constraints (SDC) standard, it is required by the Vivado IDE. The relative clock can be either a design clock or a virtual clock.

RECOMMENDED: *When using a virtual clock, use the same waveform as the design clock related to the input ports inside the design. This way, the timing path requirement is realistic. Using a virtual clock is convenient for modeling different jitter or source latency scenarios without modifying the design clock.*

The Input Delay command options are:

- Min and Max Input Delay Command Options
- Clock Fall Input Delay Command Option
- Add Delay Input Delay Command Option

Min and Max Input Delay Command Options

The `-min` and `-max` options specify different values for:

- Min delay analysis (hold/removal)
- Max delay analysis (setup/recovery).

If neither is used, the input delay value applies to both min and max.

Clock Fall Input Delay Command Option

The `-clock_fall` option specifies that the input delay constraint applies to timing paths launched by the falling clock edge of the relative clock. Without this option, the Vivado IDE assumes only the rising edge of the relative clock.

Do not confuse the `-clock_fall` option with the `-rise` and `-fall` options. These options refer to the data edge and not to the clock edge.

Add Delay Input Delay Command Option

The `-add_delay` option must be used if:

- A max (or min) input delay constraint exists, and
- You want to specify a second max (or min) input delay constraint on the same port.

This option is commonly used to constrain an input port relative to more than one clock edge, as, for example, DDR interfaces.

You can apply an input delay constraint only to input or bi-directional ports, excluding clock input ports, which are automatically ignored. You cannot apply an input delay constraint to an internal pin.

The following examples present typical uses of the `set_input_delay` command options. For additional information about input delay constraint methodology, refer to this link in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 5].

Input Delay Example One

This example defines an input delay relative to a previously defined `sysClk` for both `min` and `max` analysis.

```
> create_clock -name sysClk -period 10 [get_ports CLK0]
> set_input_delay -clock sysClk 2 [get_ports DIN]
```

Input Delay Example Two

This example defines an input delay relative to a previously defined virtual clock.

```
> create_clock -name clk_port_virt -period 10
> set_input_delay -clock clk_port_virt 2 [get_ports DIN]
```

Input Delay Example Three

This example defines a different input delay value for min analysis and max analysis relative to `sysClk`.

```
> create_clock -name sysClk -period 10 [get_ports CLK0]
> set_input_delay -clock sysClk -max 4 [get_ports DIN]
> set_input_delay -clock sysClk -min 1 [get_ports DIN]
```

Input Delay Example Four

To constrain pure combinational paths between I/O ports, an input and output delay must be defined on the I/O ports relative to a previously defined virtual clock.

The example below sets a 5 ns (10 ns - 4 ns - 1 ns) constraint on the combinational path between ports DIN and DOUT:

```
> create_clock -name sysClk -period 10 [get_ports CLK0]
> set_input_delay -clock sysClk 4 [get_ports DIN]
> set_output_delay -clock sysClk 1 [get_ports DOUT]
```

Refer to Combinatorial Delays, page 41 for further information about constraining combinational paths using the Timing Constraints wizard.

Input Delay Example Five

This example specifies input delay value relative to a DDR clock.

```
> create_clock -name clk_ddr -period 6 [get_ports DDR_CLK_IN]
> set_input_delay -clock clk_ddr -max 2.1 [get_ports DDR_IN]
> set_input_delay -clock clk_ddr -max 1.9 [get_ports DDR_IN] -clock_fall -add_delay
> set_input_delay -clock clk_ddr -min 0.9 [get_ports DDR_IN]
> set_input_delay -clock clk_ddr -min 1.1 [get_ports DDR_IN] -clock_fall -add_delay
```

This example creates constraints from data launched by both rising and falling edges of the `clk_ddr` clock outside the device to the data input of the internal flip-flop that is sensitive to both rising and falling clock edges.

Input Delay Example Six

This example specifies the clock and input delay on the STARTUPE3 internal pins (UltraScale+ devices) to time the paths from STARTUPE3 to the fabric.

```
> create_generated_clock -name clk_sck -source [get_pins -hierarchical
*axi_quad_spi_0/ext_spi_clk] [get_pins STARTUP/CCLK] -edges {3 5 7}
> set_input_delay -clock clk_sck -max 7 [get_pins STARTUP/DATA_IN[*]] -clock_fall
> set_input_delay -clock clk_sck -min 1 [get_pins STARTUP/DATA_IN[*]] -clock_fall
```

For more information on timing constraints for STARTUPE3, refer to the *AXI Quad SPI v3.2 LogiCORE IP Product Guide* (PG153) [Ref 6].

The `set_output_delay` command specifies the output path delay of an output port relative to a clock edge at the interface of the design.

VIDEO: *For training on output delay, see the* Vivado Design Suite QuickTake Video: Setting Output Delay.

When considering the application board, this delay represents the phase difference between:

a. The data propagating from the output package pin of the FPGA, through the board to another device, and

b. The relative reference board clock.

The output delay value can be positive or negative, depending on the clock and data relative phase outside the FPGA.

Note: Output delays can also be set on internal data pins such as, STARTUPE3/DATA_OUT[0:3] (UltraScale+ devices).

Although the `-clock` option is optional in the SDC standard, it is required by the Vivado Design Suite tools.

The relative clock can either be a design clock or a virtual clock.

RECOMMENDED: *When using a virtual clock, use the same waveform as the design clock related to the output ports inside the design. This way, the timing path requirement is realistic. Using a virtual clock is convenient for modeling jitter or source latency scenarios without modifying the design clock.*

The Output Delay command options are:

- Min and Max Output Delay Command Options
- Clock Fall Output Delay Command Option
- Add Delay Output Delay Command Option

Min and Max Output Delay Command Options

The `-min` and `-max` options specify different values for min delay analysis (hold/removal) and max delay analysis (setup/recovery). If neither is used, the output delay value applies to both min and max.

Clock Fall Output Delay Command Option

The `-clock_fall` option specifies that the output delay constraint applies to timing paths captured by a falling clock edge of the relative clock. Without this option, the Vivado IDE assumes only the rising edge of the relative clock (outside the device) by default.

Do not confuse the `-clock_fall` option with the `-rise` and `-fall` options. These options refer to the data edge not the clock edge.

Add Delay Output Delay Command Option

You must use the `-add_delay` option if:

- A max output delay constraint already exists, and
- You want to specify a second max output delay constraint on the same port.

The same is true for a min output delay constraint. This option is commonly used to constrain an output port relative to more than one clock edge, as, for example, rising and falling edges in DDR interfaces, or when the output port is connected to several devices that use different clocks.

IMPORTANT: *You can apply an output delay constraint only to output or bi-directional ports. You cannot apply an output delay constraint to an internal pin.*

The following examples present typical uses of the `set_output_delay` command options. For additional information about output delay constraint methodology, refer to this link in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 5].

Output Delay Example One

This example defines an output delay relative to a previously defined `sysClk` for both `min` and `max` analysis.

```
> create_clock -name sysClk -period 10 [get_ports CLK0]
> set_output_delay -clock sysClk 6 [get_ports DOUT]
```

Output Delay Example Two

This example defines an output delay relative to a previously defined virtual clock.

```
> create_clock -name clk_port_virt -period 10
> set_output_delay -clock clk_port_virt 6 [get_ports DOUT]
```

Output Delay Example Three

This example specifies output delay value relative to a DDR clock with different values for `min` (hold) and `max` (setup) analysis.

```
> create_clock -name clk_ddr -period 6 [get_ports DDR_CLK_IN]
> set_output_delay -clock clk_ddr -max 2.1 [get_ports DDR_OUT]
> set_output_delay -clock clk_ddr -max 1.9 [get_ports DDR_OUT] -clock_fall
-add_delay
> set_output_delay -clock clk_ddr -min 0.9 [get_ports DDR_OUT]
> set_output_delay -clock clk_ddr -min 1.1 [get_ports DDR_OUT] -clock_fall
-add_delay
```

This example creates constraints from data launched by both rising and falling edges of the `clk_ddr` clock outside the device to the data output of the internal flip-flop sensitive to both rising and falling clock edges.

Output Delay Example Four

This example specifies the clock and output delay on the STARTUPE3 internal pins (UltraScale+ devices) to time the paths from the fabric to STARTUPE3.

```
> create_generated_clock -name clk_sck -source [get_pins -hierarchical
*axi_quad_spi_0/ext_spi_clk] [get_pins STARTUP/CCLK] -edges {3 5 7}
> set_output_delay -clock clk_sck -max 6 [get_pins STARTUP/DATA_OUT[*]]
> set_output_delay -clock clk_sck -min 1 [get_pins STARTUP/DATA_OUT[*]]
```

For more information on timing constraints for STARTUPE3, refer to the *AXI Quad SPI v3.2 LogiCORE IP Product Guide* (PG153) [Ref 6].

Timing Exceptions

A timing exception is needed when the logic behaves in a way that is not timed correctly by default. You must use a timing exception command any time you want the timing handled differently (for example, for logic that only has the result captured every other clock cycle by design).

The Xilinx® Vivado® Integrated Design Environment (IDE) supports the timing exceptions commands shown in Table 5-1.

Table 5-1:

`set_multicycle_path`	Indicates the number of clock cycles required to propagate data from the start to the end of a path.
`set_false_path`	Indicates that a logic path in the design should not be analyzed.
`set_max_delay` `set_min_delay`	Sets the minimum and maximum path delay value. This overrides the default setup and hold constraints with user specified maximum and minimum delay values.

Note: For runtime consideration, Vivado tools do not provide on-the-fly analysis for conflicting timing exceptions. Run `report_exceptions` for full analysis and reporting of the timing exceptions.
For more information, refer to *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

The Multicycle Path constraint allows you to modify the setup and hold relationships determined by the timer, based on the design clock waveforms. By default, the Vivado IDE timing engine performs a single-cycle analysis. This analysis can be too restrictive, and can be inappropriate for certain logic paths.

The most common example is the logical path that requires more than one clock cycle for the data to stabilize at the endpoint. If the control circuitry of the path startpoint and endpoint allows it, Xilinx recommends that you use the Multicycle Path constraint to relax the setup requirement.

The hold requirement might still maintain the original relationship, depending on your intent. This helps the timing-driven algorithms to focus on other paths that have tighter requirements and that are challenging. It can also help in reducing runtime.

The `set_multicycle_path` command is used to modify the path requirement multipliers (for setup analysis, hold analysis, or both) with respect to the source clock or the destination clock.

set_multicycle_path Syntax

The syntax of the `set_multicycle_path` command with the basic options is:

```
set_multicycle_path <path_multiplier> [-setup|-hold] [-start|-end]
  [-from <startpoints>] [-to <endpoints>] [-through <pins|cells|nets>]
```

You must specify the `<path_multiplier>`. The default values used by the timer are:

- `1` for setup analysis (or recovery)
- `0` for hold analysis (or removal)

The hold relationship is tied to the setup relationship. Use the following formula to retrieve the number of hold cycles for most common cases:

```
Hold cycles = <setup path multiplier> - 1 - <hold path multiplier>
```

- By default, the setup path multiplier is defined with respect to the destination clock. To modify the setup requirement with respect to the source clock, use the `-start` option.
- Similarly, the hold path multiplier is defined with respect to the source clock. To modify the hold requirement with respect to the destination clock, use the `-end` option.

Note: For a definition of the relevant terms, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

IMPORTANT: *There are two hold relationships for each setup relationship. (1) The first hold relationship ensures that the setup launch edge is not captured by the edge arriving before the active capture edge. (2) The second hold relationship ensures that the edge after the active launch edge is not captured by the active capture edge. The timing analysis tool calculates both hold relationships but only the most restrictive is kept during analysis and reporting. See Figure 5-1.*

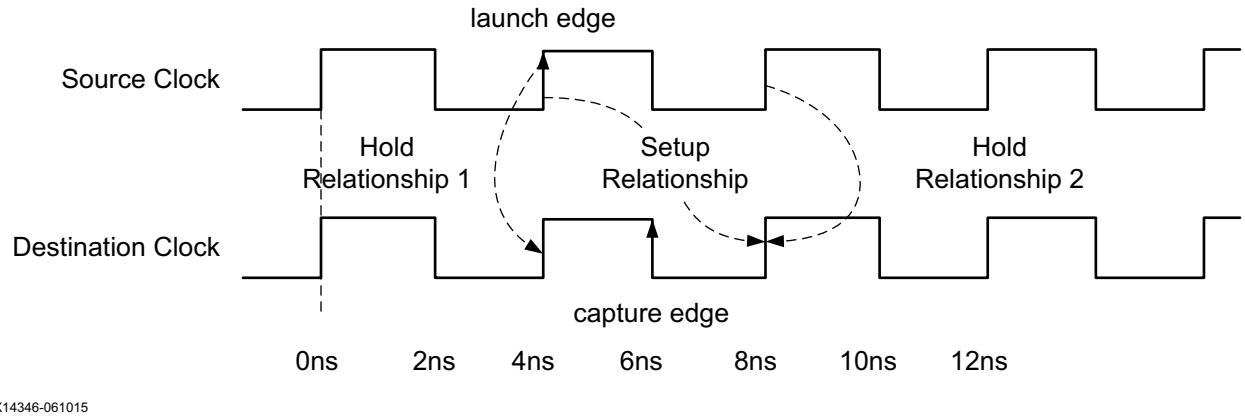


Figure 5-1:

IMPORTANT: *The `-start` and `-end` options have no apparent effect when applying a Multicycle Path constraint on paths clocked by the same clock or clocked by two identical clocks (that is, when the clocks have the same waveform with or without a phase shift).*

Table 5-2 summarizes how the active launch and capture edges are affected by the `-end` and `-start` options.

Table 5-2:

	<---- (backward)	----> (forward) (default)
	----> (forward) (default)	<---- (backward)

IMPORTANT: *The `-setup` option of the `set_multicycle_path` command does not only modify the setup relationship. It also affects the hold relationships which are always tied to the setup relationships. If the hold relationship is to be restored back to its original position, another `set_multicycle_path` specification would be needed with `-hold`.*

Note: A Multicycle constraint can be set on a single path, on multiple paths, or even between two clocks.

The following sections cover the common Multicycle Path constraint scenarios and illustrate the impact of the setup and hold multipliers and the `-start` and `-end` options on the timing path requirement.

A Multicycle constraint defined within the same clock domain or between two clocks with the same waveform (no phase-shift) works the same way. See Figure 5-2.

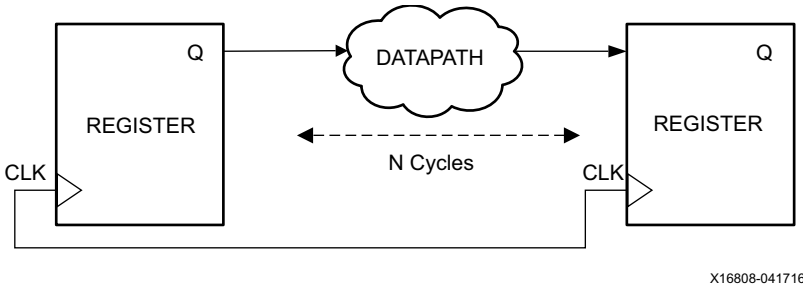


Figure 5-2:

The default Setup and Hold relationships that are resolved by the Static Timing Analysis (STA) tool are shown in Figure 5-3.

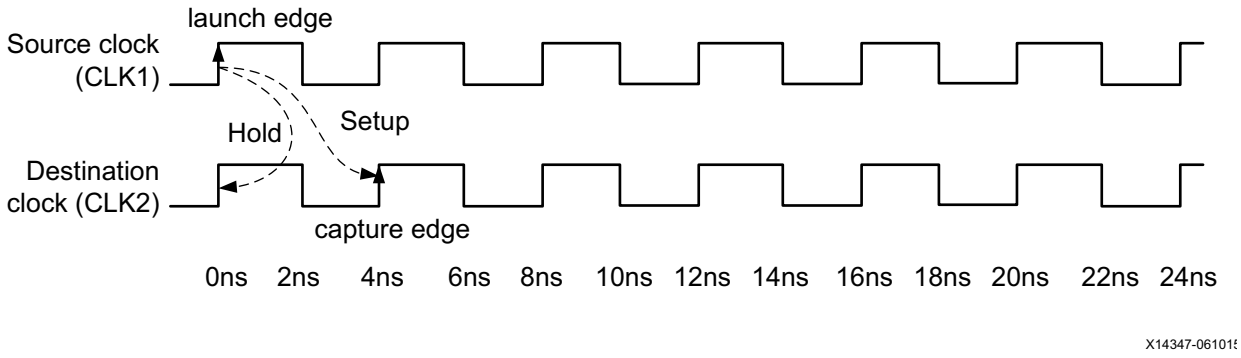


Figure 5-3:

The Setup and Hold timing requirements are:

- Setup check

  $T_{Datapath(max)} < TCLK_{(t=Period)} - T_{Setup}$

- Hold check

  $T_{Datapath(min)} > TCLK_{(t=0)} + T_{Hold}$

Relaxing Setup While Maintaining Hold

Figure 5-4 shows a path between two flip-flops that are enabled every two cycles. It is safe to define a Multicycle Path constraint on this path to indicate that the first edge of the destination clock is not active, and only the second edge of the destination clock will capture a new data.

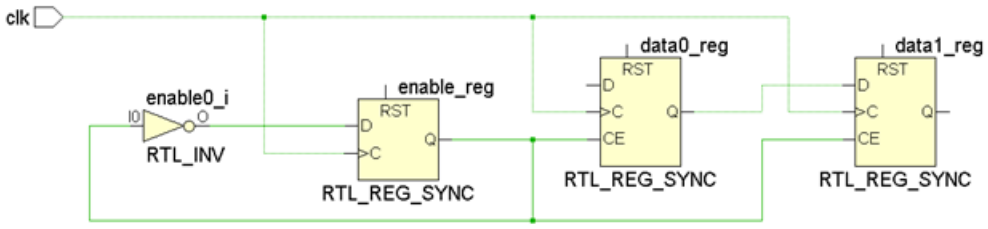


Figure 5-4:

The following constraint establishes a new setup relationship:

```
set_multicycle_path 2 -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

This link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4] describes how the hold relationships are derived from the setup relationships. When modifying the setup relationship, the hold relationships are also modified to follow the changes in the setup launch and capture edges.

IMPORTANT: *If the new hold requirements become too aggressive, it will likely result in difficult timing closure. It is the your responsibility to relax the hold requirement assuming it is safe for the design.*

In the same example as Figure 5-4, after moving the setup check to the second capture edge, the hold check is automatically moved to the first capture edge (that is, one clock period before the setup check).

Figure 5-5 shows how both the setup and hold relationships have changed when only the setup path multiplier has been defined with the Multicycle Path constraint.

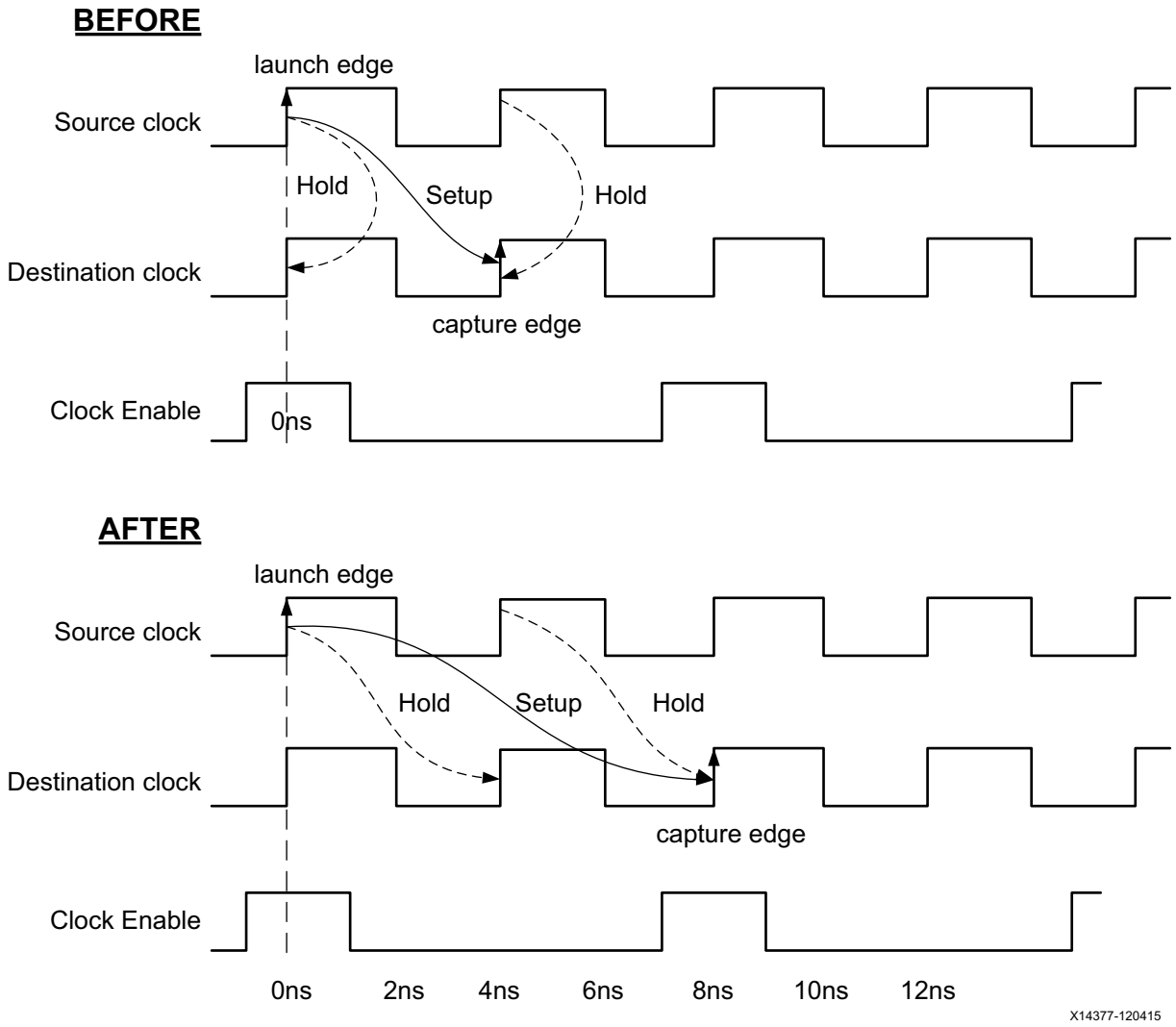


Figure 5-5:

Holding the data in the `data0_reg` for one cycle is not needed for this path to be functional due to the clock enable. In this case, Xilinx recommends changing the hold relationship back to the original, which is between the same launch and capture edges. To do so, you must add a second Multicycle Path constraint that modifies the hold check only:

```
set_multicycle_path 1 -hold -end -from [get_pins data0_reg/C] \
    -to [get_pins data1_reg/D]
```

The `-end` option is used with `set_multicycle -hold` command because the edges of the capture clock must be moved backward.

Note: Because the launch and capture clocks have the same waveforms, the `-end` option is optional. Moving the capture edges backward result in the same hold relationship as moving the launch edges forward. To simplify the expressions, the `-end` option has been removed from the next two examples.

Figure 5-6 shows the updated setup and hold relationships after applying both Multicycle Path constraints.

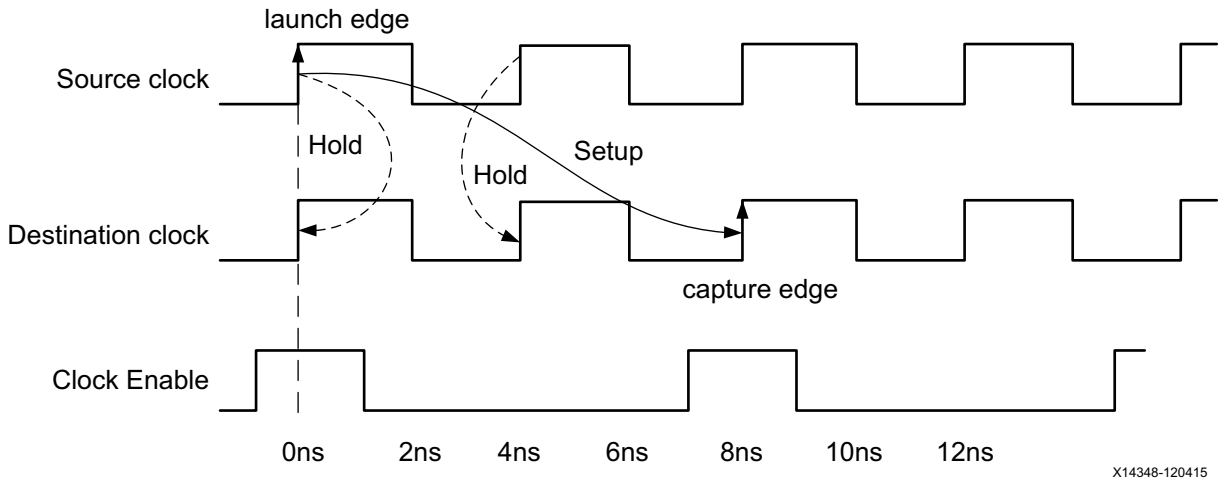


Figure 5-6:

To summarize this example, the following constraints are necessary to properly define a multicycle path of two (2) between `data0_reg/C` and `data1_reg/D`:

```
set_multicycle_path 2 -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
set_multicycle_path 1 -hold -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

For a multicycle with a setup multiplier of four (4), the constraints are:

```
set_multicycle_path 4 -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
set_multicycle_path 3 -hold -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

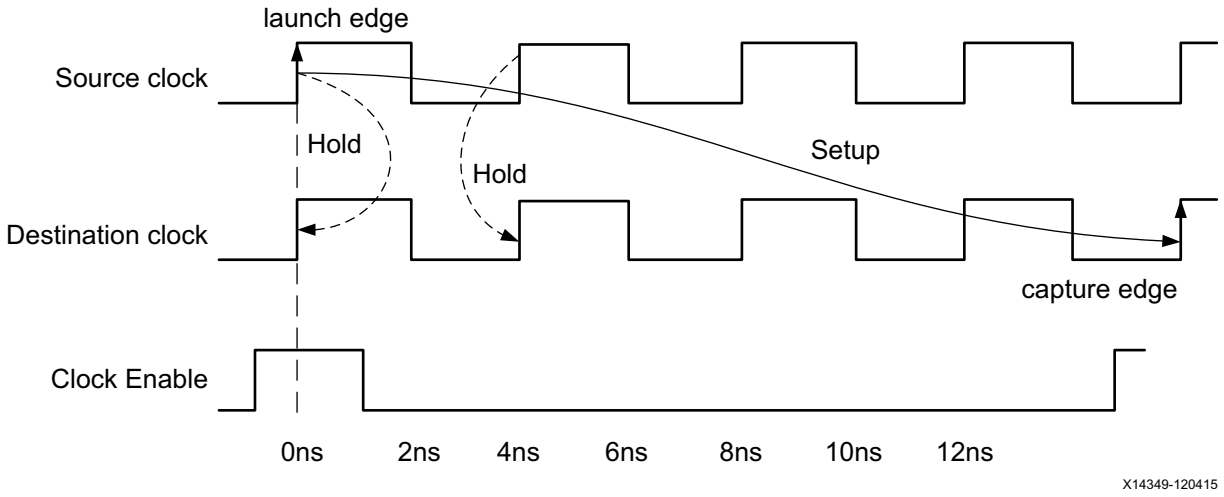


Figure 5-7:

Moving the Setup

The following examples show the results of moving the setup:

- Example One: Setup=5 / Hold Moved Accordingly
- Example Two: Setup=5 / Hold=4

Example One: Setup=5 / Hold Moved Accordingly

Let's assume that the setup path multiplier is set to five (5). Because the hold path multiplier is not specified, the hold relationship is derived from the setup launch and capture edges:

```
set_multicycle_path 5 -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

By default, the setup multiplier is applied against the capture clock. This results in moving the edge on the capture clock forward. The setup capture edge comes after five clock periods instead of just one. Because no hold multiplier has been specified, the edge of the capture clock used for the hold check stays the edge that arrives one cycle before the active edge used for the setup check.

The edges on the launch clock do not change for the setup and hold relationships.

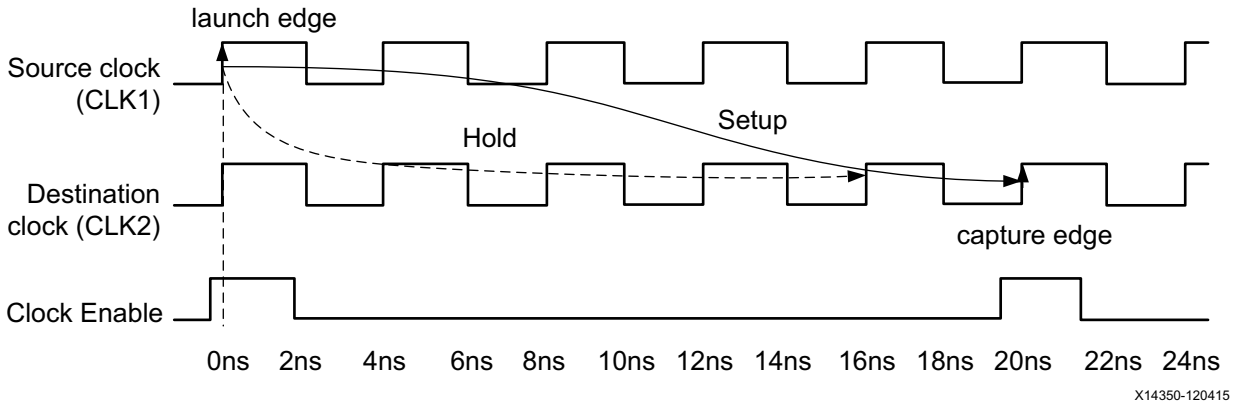


Figure 5-8:

With a four-cycle hold requirement, the timing-driven implementation tools usually have to insert a large amount of delay in the datapath in order to meet hold timing in both Slow and Fast timing corners. This results in unnecessary area and power consumption. For this reason, it is important to relax the hold requirement when possible.

In this example design, the clock enable signal provides the safety to not have to hold the data in the `data0_reg` for four cycles without risking metastability. Example Two: Setup=5 / Hold=4 describes how the hold requirement can be relaxed.

Example Two: Setup=5 / Hold=4

This example assumes that the following are defined:

- A setup multiplier of five (5)
- A hold multiplier of four (4) (that is, 5-1)

This corresponds to a transfer between two sequential cells when a new data is launched and captured every five (5) cycles.

```
set_multicycle_path 5 -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
set_multicycle_path 4 -hold -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

By default, the setup multiplier is applied against the destination clock, which in this case results in moving the capture edge forward to the fifth cycle instead of the first cycle. Accordingly, by default, the hold check follows the setup check.

On specifying the second command, the hold multiplier is applied against the source clock, which in this case results in moving the launch edge forward to the fourth cycle.

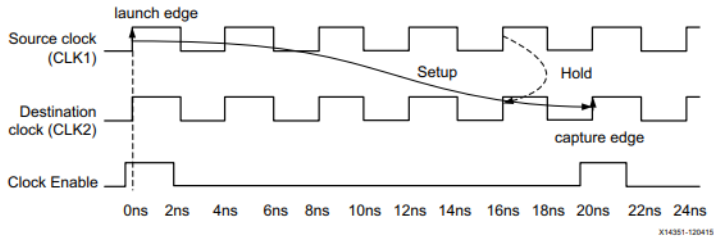


Figure 5-9:

Because both source and destination clocks have the same waveforms, and are phase-aligned, Figure 5-9 is equivalent to Figure 5-10.

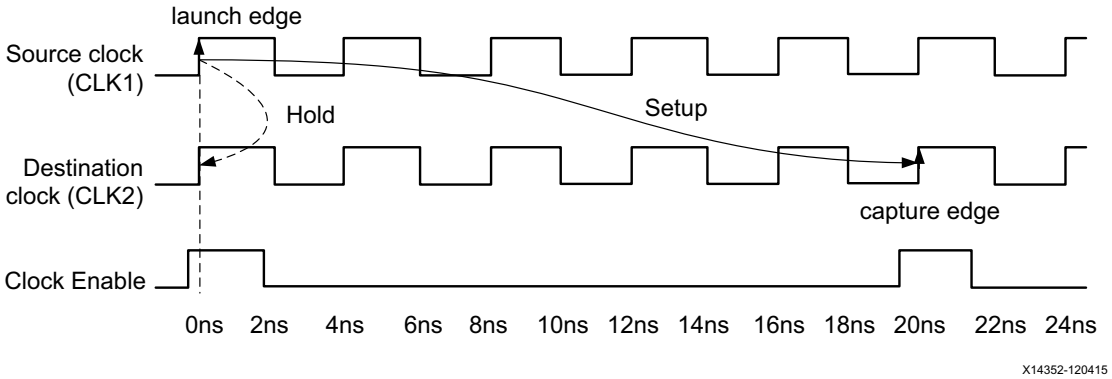


Figure 5-10:

IMPORTANT: *In general, within a clock domain or between two clocks with the same waveform, when a setup multiplier of `N` is defined, define a hold multiplier of `N-1` (most common case) as shown below.*

```
set_multicycle_path N -setup -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
set_multicycle_path N-1 -hold -from [get_pins data0_reg/C] -to [get_pins data1_reg/D]
```

Sometimes a timing constraint must be defined between two clock domains that havethe same clock period, but a phase-shift between the two clocks. In those cases, it is critical to understand the default setup and hold relationships used by the timing engine. If not carefully adjusted, the phase-shift between two clocks might result in over constraining the logic between the two clock domains.

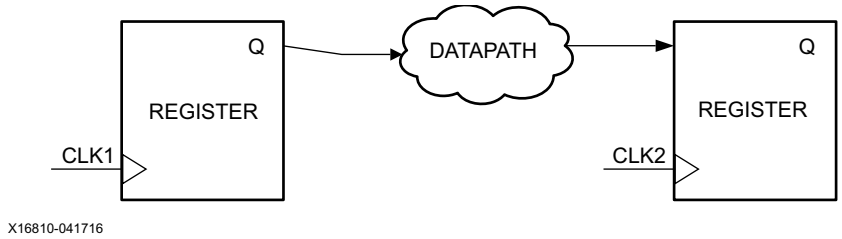


Figure 5-11:

For example, assume the following:

- The two clocks CLK1 and CLK2 have the same waveform.
- CLK2 is shifted by +0.3 ns.

The setup relationship is calculated by the timing engine by looking at all the edges on both waveforms and selecting the two edges on the launch and capture clocks that result in the stricter constraint.

Because of the clocks phase-shift, the setup and hold relationships used by the timing engine might not be those expected. See Figure 5-12.

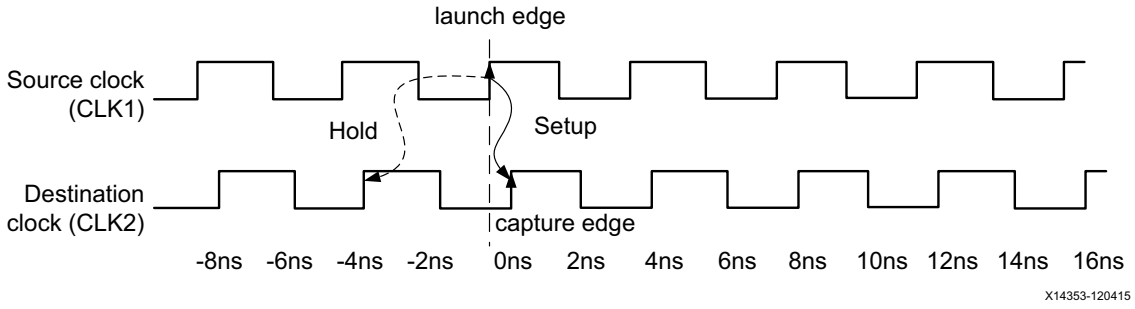


Figure 5-12:

In this example, the setup constraint due to the phase-shift is 0.3 ns. This makes it almost impossible to achieve timing closure. On the other hand, the hold check is -3.7 ns, which is too lenient.

The setup and hold edges must be adjusted to match your intent. This is done by adding a Multicycle constraint with a setup multiplier of two (2):

```
set_multicycle_path 2 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
```

This results in moving the capture edge for the setup requirement forward by one cycle. The default edge for the hold is derived from the setup requirement. It does not need to be specified.

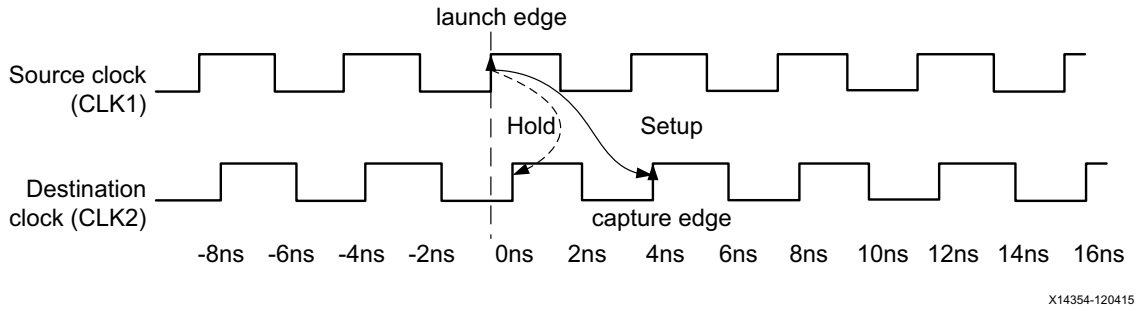


Figure 5-13:

In the case of negative phase-shift, as shown in Figure 5-14, between the two clock domains, the launch and capture edges used for the setup and hold checks are similar to those from the previous section (single clock domain, no phase-shift).

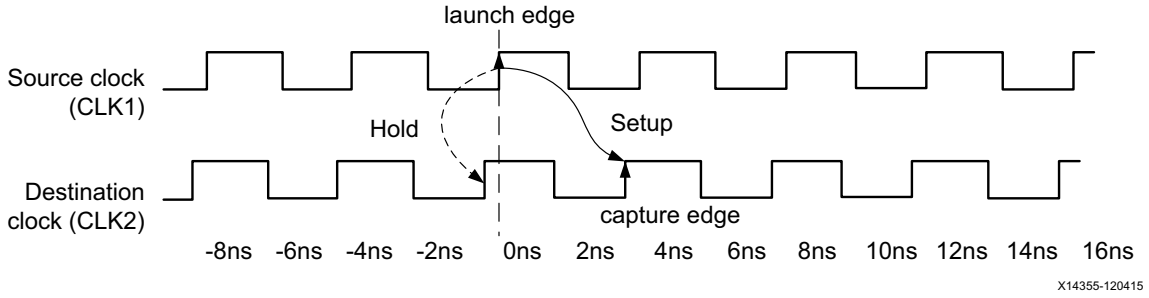


Figure 5-14:

For a negative phase-shift, a Multicycle constraint is typically not needed to counter-balance the effect of the phase-shift. An exception occurs if the phase-shift is so large that the clock launch or capture edges must be adjusted to keep realistic setup and hold requirements.

In this scenario, the launch clock CLK1 is the slow clock; the capture clock CLK2 is the fast clock. See Figure 5-15.

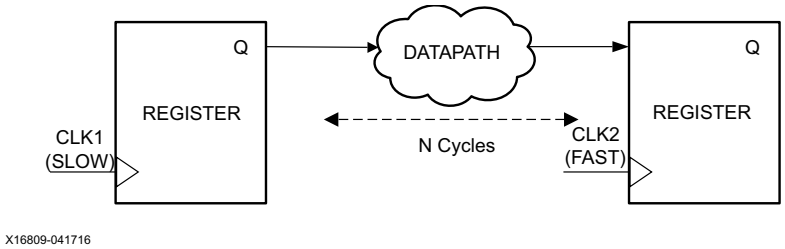


Figure 5-15:

For example, assume the following:

- CLK2 is three times the frequency of CLK1
- A clock enable signal on the receiving registers allows a Multicycle constraint to be set between both clocks. See Figure 5-16.

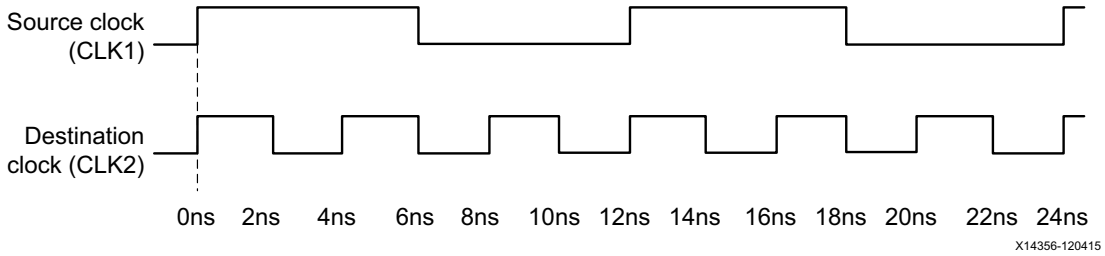


Figure 5-16:

The setup and hold relationships that are resolved by the STA tool when no multicycle is applied are shown in Figure 5-17.

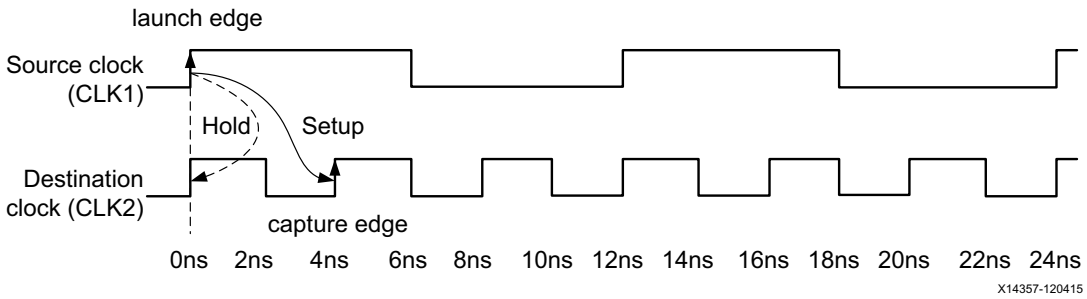


Figure 5-17:

Example One: Setup=3 / Hold Moved Accordingly

For example, assume that only a setup multiplier of three (3) is defined.

```
set_multicycle_path 3 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
```

The consequence of the setup multiplier is to move the edge of the capture clock used for setup check forward by two (2) cycles (that is, 3-1 cycles). Because no hold multiplier has been specified, the hold relationship is derived by the tool from the setup launch and capture edges. The launch clock active edge is not modified by the Multicycle constraint.

The setup and hold relationships after the multicycle are shown in Figure 5-18.

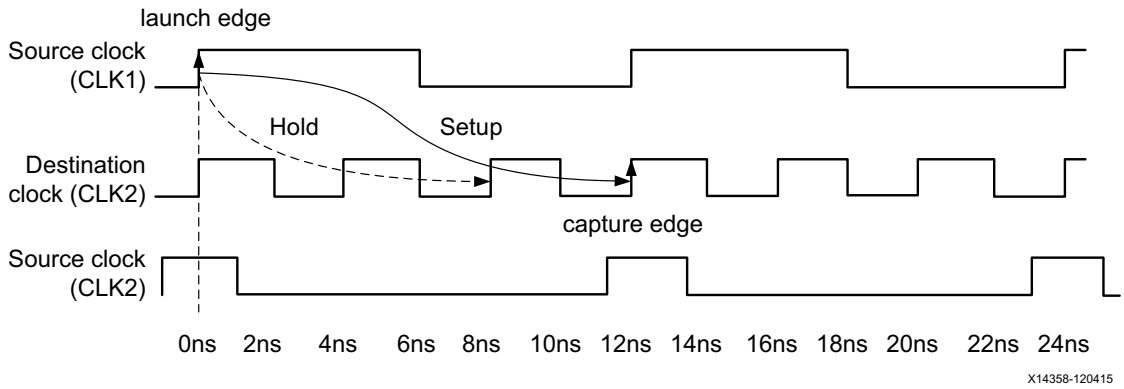


Figure 5-18:

There is no need to hold the data in the launch registers for one cycle of CLK2 for this path to be functional. Doing so adds unnecessary logic, which increases area and consumes power.

Because the receiving registers have a clock enable signal, it is safe to relax the hold requirement without risks of metastability.

Example Two: Setup=3 / Hold=2 (-end)

To relax the hold requirement for the previous example, the capture clock edge for the hold relationship must be moved backward by two (2) clock cycles. This is done by specifying the `-end` option with the `set_multicycle_path -hold` command:

```
set_multicycle_path 3 -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
set_multicycle_path 2 -hold -end -from [get_clocks CLK1] -to [get_clocks CLK2]
```

TIP: *If `-end` is not specified with `set_multicycle_path -hold`, then the launch clock edge is instead moved forward. This does not result in the intended hold requirement.*

As in Example One: Setup=3 / Hold Moved Accordingly, the setup multiplier moves the edge of the capture clock used for setup check forward by two (2) cycles (that is, 3-1 cycles).

The setup and hold relationships after the two Multicycle constraints are shown in Figure 5-19.

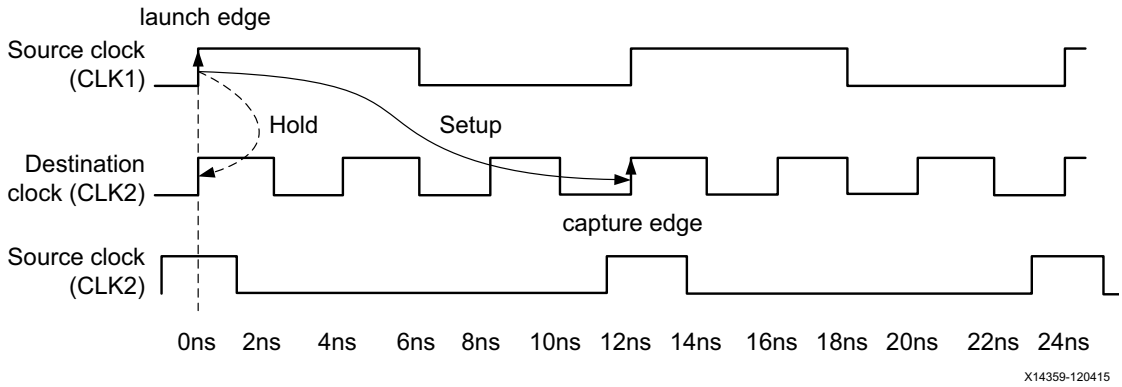


Figure 5-19:

IMPORTANT: *For a SLOW-to-FAST clock domain crossing, when a setup multiplier of `N` is defined, define a hold multiplier of `N-1` against the capture clock (`-end`) (most common case) as shown in the following code example.*

```
set_multicycle_path N -setup -from [get_clocks CLK1] -to [get_clocks CLK2]
set_multicycle_path N-1 -hold -end -from [get_clocks CLK1] -to [get_clocks CLK2]
```

In the following scenario, the launch clock CLK1 is the fast clock and the capture clock CLK2 is the slow clock. See Figure 5-20.

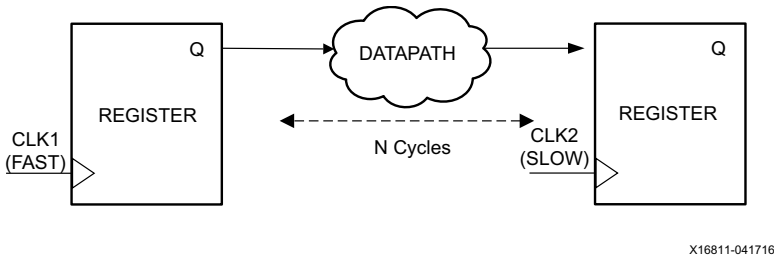


Figure 5-20:

In the next example, the launch clock CLK1 is the fast clock. The capture clock CLK2 is the slow clock. Assume that CLK1 is three (3) times the frequency of CLK2. See Figure 5-21.

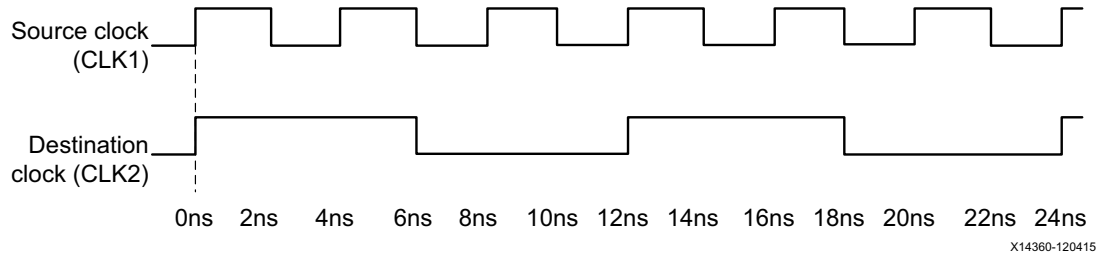


Figure 5-21:

The setup and hold relationships that are resolved by the STA tool when no multicycle is applied are shown in Figure 5-22.

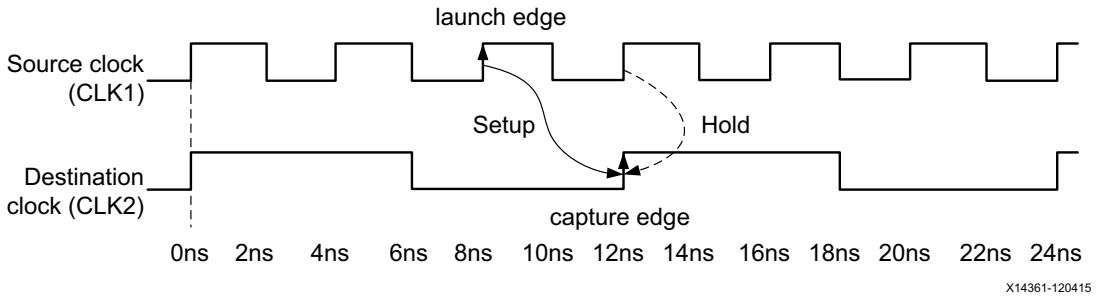


Figure 5-22:

Example: Setup=3 (-start) / Hold=2

This example assumes the following:

- A setup multiplier of three (3) is defined against the launch clock (`-start`).
- A hold multiplier of one (1) is defined.

Example:

```
set_multicycle_path 3 -setup -start -from [get_clocks CLK1] -to [get_clocks CLK2]
set_multicycle_path 2 -hold -from [get_clocks CLK1] -to [get_clocks CLK2]
```

The consequence of defining the setup multiplier against the launch clock (`-start`) is to move the edge of the launch clock used for setup check backward by two (2) cycles (that is, 3-1 cycles). However, because a hold multiplier is defined against the launch clock (default `-start` option with `-hold`), the edge of the launch clock that is used for the hold relationship is moved forward by two (2) cycles.

For both setup and hold checks, the capture clock edge does not change. See the following figure.

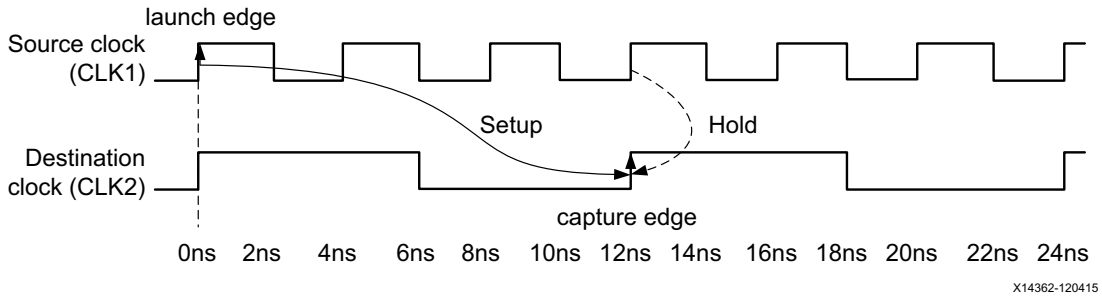


Figure 5-23:

IMPORTANT: *For a FAST-to-SLOW clock domain crossing, define a setup multiplier of `N` against the launch clock (`-start`) with a hold multiplier of `N-1` (most common case). See the following example:*

```
set_multicycle_path N -setup -start -from [get_clocks CLK1] -to [get_clocks CLK2]
set_multicycle_path N-1 -hold -from [get_clocks CLK1] -to [get_clocks CLK2]
```

Table 5-3 summarizes the previous results.

Table 5-3:

Same clock domain or between synchronous clock domains with same period and no phase-shift	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -from CLK1 -to CLK2
Between SLOW-to FAST synchronous clock domains	set_multicycle_path N -setup -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -end -from CLK1 -to CLK2
Between FAST-to SLOW synchronous clock domains	set_multicycle_path N -setup -start -from CLK1 -to CLK2 set_multicycle_path N-1 -hold -from CLK1 -to CLK2

Note: The `get_clocks` command has been omitted in Table 5-3 to simplify the expressions.

A false path is a path that topologically exists in the design but either: (1) is not functional; or (2) does not need to be timed. Consequently, the false paths should be ignored during timing analysis.

VIDEO: *For training on the advanced timing exceptions, including false paths, see the* Vivado Design Suite QuickTake Video: Advanced Timing Exceptions - False Path, Min-Max Delay and Set_Case_Analysis.

Examples of false paths include:

- Clock domain crossings in which double synchronizer logic has been added
- Registers that might be written once at power up
- Reset or test logic
- Ignore paths between the write and asynchronous read clocks of an asynchronous distributed RAM (when applicable)

Figure 5-24 shows an example of a non-functional path. Because both multiplexers are driven by the same select signal, the path from Q to D does not exist, and should be defined as a false path.

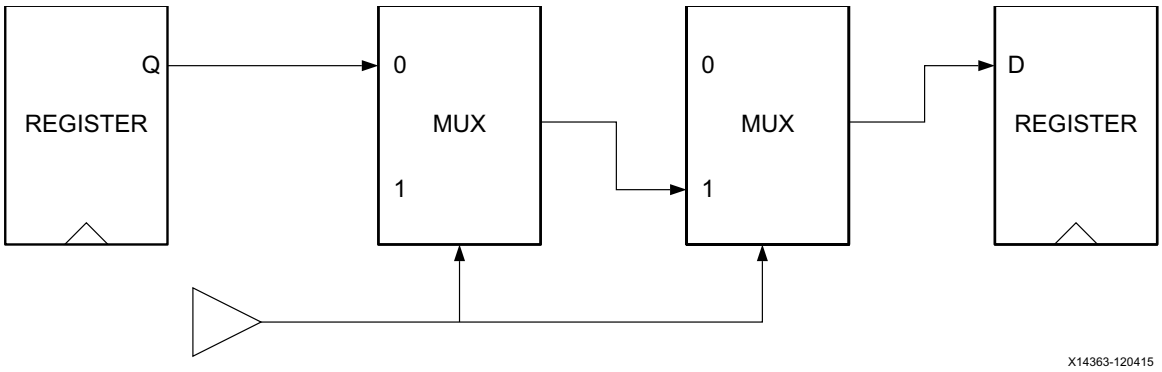


Figure 5-24:

TIP: *Use a Multicycle constraint in place of a False Path constraint when: (1) your intent is only to relax the timing requirements on a synchronous path; but (2) the path still must be timed, verified and optimized.*

Reasons to remove false paths from the timing analysis include:

- **Decrease Runtime**

  When false paths have been removed from the timing analysis, the tool does not need to time or optimize those non-functional paths. Having non-functional paths visible to the timing and optimization engines can result in a large runtime penalty.

- **Enhance Quality of Results (QOR)**

  Removing false paths can greatly enhance the Quality of Results (QOR). The quality of the synthesized, placed, and optimized design is greatly impacted by the timing issues that the tool tries to solve.

  If some non-functional paths have timing violations, the tool might try to fix those paths instead of working on the real functional paths. Not only might the design unnecessarily increase in size (such as logic cloning), but the tool might skip fixing real issues because non-functional paths have larger violations that overshadow other real violations. The best results are always achieved with a realistic set of constraints.

False paths are defined inside the tool with the Xilinx Design Constraints (XDC) command `set_false_path`:

```
set_false_path [-setup] [-hold] [-from <node_list>] [-to <node_list>] \
    [-through <node_list>]
```

You can use the following additional options to the command to fine tune the path specification. For detailed information about all supported command line options, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 10].

- The list of nodes for the `-from` option should be a list of valid startpoints. A valid startpoint is a clock object, a clock pin of a sequential element, or an input (or inout) primary port. Multiple elements can be provided.
- The list of nodes for the `-to` option should be a list of valid endpoints. A valid endpoint is a clock object, an output (or inout) primary port, or a sequential element input data pin. Multiple elements can be provided.
- The list of nodes for the `-through` option should be a list of valid pins, ports, or nets. Multiple elements can be provided.

CAUTION! *Be careful when using `-through` option without `-from` and `-to` because it removes from timing analysis any path going through this list of pins or ports. Be especially careful when the timing constraints are designed for an IP or a sub-block, but then used in a different context or a larger project. Many more paths than expected could be removed when `-through` is used alone.*

The order of the `-through` option is important. See the following examples.

For example, the following two commands are different:

```
set_false_path -through cell1/pin1 -through cell2/pin2
set_false_path -through cell2/pin2 -through cell1/pin1
```

The following example removes the timing paths from the reset port to all the registers:

```
set_false_path -from [get_port reset] -to [all_registers]
```

The following example disables the timing paths between two asynchronous clock domains (for example, from clock CLKA to clock CLKB):

```
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
```

The previous example disables the paths from clock CLKA to clock CLKB. Paths from clock CLKB to clock CLKA are not disabled. Accordingly, disabling all the paths between the two clock domains in either direction requires two `set_false_path` commands:

```
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
set_false_path -from [get_clocks CLKB] -to [get_clocks CLKA]
```

IMPORTANT: *Although the previous two `set_false_path` examples perform what is intended, when two or more clock domains are asynchronous and the paths between those clock domains should be disabled in either direction, Xilinx recommends using the `set_clock_groups` command instead:*
`set_clock_groups -group CLKA -group CLKB`

In the non-functional path example shown in Figure 5-24, the false path can be set using the `-through` option instead of using the `-from` or `-to` option. See Figure 5-25.

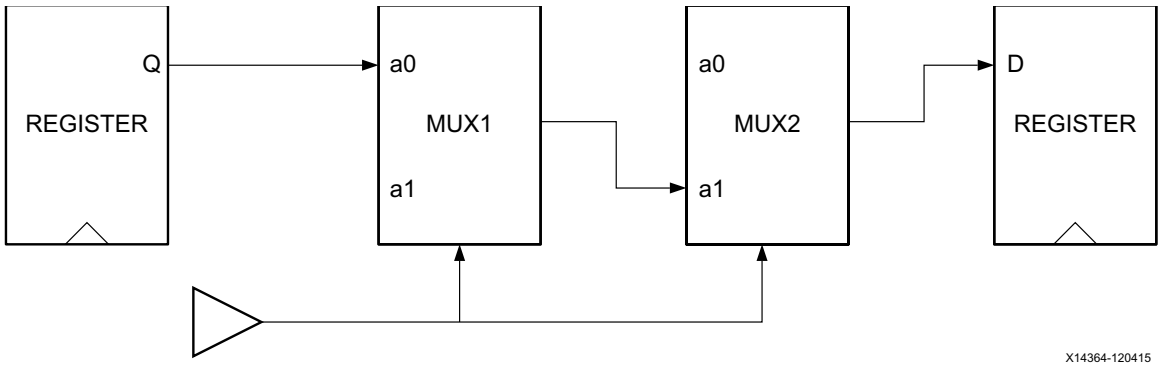


Figure 5-25:

This ensures that all the paths going through the path shown above are selected without needing to find specific patterns for the startpoints and endpoints.

```
set_false_path -through [get_pins MUX1/a0] -through [get_pins MUX2/a1]
```

Note: The order of the `-through` option is important. In the above example, the order ensures that the false paths go through pin `MUX1/a0` first and then pin `MUX2/a1`.

Another common example is with asynchronous dual-ports distributed RAM. The write operations are synchronous to the clock RAM but the read operations can be asynchronous when permitted by the design. In this case, it is safe to false paths the timing paths between the write and the read clocks.

There are two ways to do this:

- Define a false path from the write registers before the RAM to the registers after the RAM receiving the read clock:

  ```
  set_false_path -from [get_cells <write_registers>] -to [get_cells <read_registers>]
  ```

  On the Vivado Design Suite example project WAVEGen (HDL):

  ```
  set_false_path -from [get_cells -hier -filter {NAME =~
  *gntv_or_sync_fifo.gl0.wr*reg[*]}] -to [get_cells -hier -filter {NAME=~
  *gntv_or_sync_fifo.mem*gpr1.dout_i_reg[*]}]
  ```

- Define a false path starting from the pin WE of the RAM

  ```
  set_false_path -from [get_cells -hier -filter {REF_NAME =~ RAM* && IS_SEQUENTIAL &&
  NAME =~ <PATTERN_FOR_DISTRIBUTED_RAMS>}]
  ```

  On the Vivado Design Suite example project WAVEGen (HDL):

  ```
  set_false_path -from [get_cells -hier -filter {REF_NAME =~ RAM* && IS_SEQUENTIAL &&
  NAME =~ *char_fifo*}]
  ```

Figure 5-26 illustrates the way the distributed RAM is driven in the WAVE (HDL) example project.

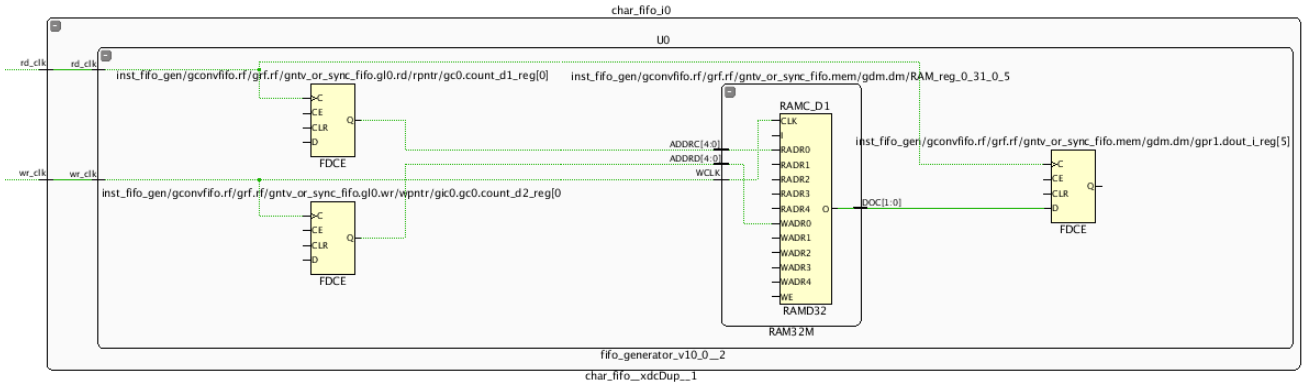


Figure 5-26:

You can override a maximum delay or a minimum delay for a path:

- Use the *Maximum Delay* constraint to override the default setup (or recovery) requirement on a path.
- Use the *Minimum Delay* constraint to override the default hold (or removal) requirement.

VIDEO: *For training on the advanced timing exceptions, including min-man delays, see the* Vivado Design Suite QuickTake Video: Advanced Timing Exceptions - False Path, Min-Max Delay and Set_Case_Analysis.

The Maximum Delay constraint and the Minimum Delay constraint are set by two different XDC commands. These commands accept similar options.

Maximum Delay Constraint Syntax

```
set_max_delay <delay> [-datapath_only] [-from <node_list>]
                      [-to <node_list>] [-through <node_list>]
```

Minimum Delay Constraint Syntax

```
set_min_delay <delay> [-from <node_list>]
                      [-to <node_list>] [-through <node_list>]
```

Additional command options are available to fine tune the path specification. For more information about the supported command line options, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 10].

List of Nodes for the -from Option

- The list of nodes for the `-from` option should preferably be a list of valid startpoints. A valid startpoint is a clock, an input (or inout) port, or the clock pin of a sequential element, such as a register or a RAM.Using a node that is not a valid startpoint results in path segmentation. The path segmentation is covered in the next section.
- Multiple elements can be provided.

List of Nodes for the -to Option

- The list of nodes for the `-to` option should preferably be a list of valid endpoints. A valid endpoint is a clock, an output (or inout) port or the data pin of a sequential cell.
- Using a node that is not a valid endpoint results in path segmentation. For more information, see Path Segmentation, page 133.
- Multiple elements can be provided.

List of Nodes for the -through Option

- The list of nodes for the `-through` option should be a list of valid pins, ports, or nets.
- Multiple elements can be provided.

By default, the timing engine includes the clock skew inside the slack computation.

The `-datapath_only` option can be used to remove the clock skew from the slack computation. The `-datapath_only` option is supported only by the `set_max_delay` command, and requires the `-from` option.

Table 5-4 summarizes the impact of `-datapath_only` in the behavior of `set_max_delay` constraint.

The common behavior for the path delay calculation of `set_max_delay` with or without `-datapath_only` is:

- Input delay is included in the path delay calculation when the path starts on an input port and that a `set_input_delay` has been specified on the port
- Output delay is included in the path delay calculation when the path ends on an output port and that a `set_output_delay` has been specified on the port
- The data pin setup time is included in the path delay calculation when the path ends on the data pin of a sequential element.

Table 5-4:

Path delay calculation	Skew included when the constraint starts on the clock pin of a sequential element or ends on the data pin of a sequential element.	Skew never included.
Hold Requirement	Untouched	False-ed path
`-from` Option	Optional	Mandatory

Consequences of Setting Maximum Delay or Minimum Delay Constraints on a Path

When `-datapath_only` option is not used, setting a Maximum Delay constraint on a path, does not modify the minimum requirement on that path. The hold (or removal) check on that path remains the default one.

Note: Using the `-datapath_only` option with `set_max_delay` results in the hold requirement being ignored on that/those path(s) (some internal `set_false_path -hold` constraints are generated).

Similarly, setting a Minimum Delay constraint on a path does not modify the default setup (or recovery) check.

If a path has only, for example, a max delay requirement, the path can be constrained with a combination of `set_max_delay` and `set_false_path` commands. See the following example:

```
set_max_delay 5 -from [get_pins FD1/C] -to [get_pins FD2/D]
set_false_path -hold -from [get_pins FD1/C] -to [get_pins FD2/D]
```

The above example sets a 5ns setup requirement for the path starting on `FD1/C` and ending on `FD2/D`. There is no minimum requirement due to the `set_false_path` command.

Constraining Input or Output Logic

The set_max_delay command and the set_min_delay command are not typically used to constrain the input or output logic. The input logic between the input ports and the first level of registers is typically constrained with the set_input_delay command. This command provides the option to associate a clock with the input ports.

For the same reason, the output logic between the last level of registers and the output ports is typically constrained with the set_output_delay command. However, the set_max_delay command and the set_min_delay command are typically used to constrain pure combinational path between primary input ports and primary output port (in-to-out I/O paths).

Constraining Asynchronous Signals

The set_max_delay command can also be used to constrain asynchronous signals that do not have a clock relationship, but which require maximum delay.

For example, timing paths between two asynchronous clock domains can be disabled with the set_clock_groups command (recommended) or the set_false_path command (not recommended). This assumes that you have properly designed the inter-clock domains with, for instance, a double registers synchronizer or a FIFO. However, you must still ensure that the path delay between the two clock domains is not unnecessarily high.

In some multi-bit CDC scenarios the skew between the bits must be within certain requirements. Even though the skew can be constrained through the Bus Skew constraint (set_bus_skew), it must be ensured that the path delay between the two clock domains is not unnecessarily high. This can be done by replacing the set_false_path or set_clock_groups constraints inside the source XDC file on the relevant path(s) with set_max_delay -datapath_only. Refer to Chapter 6, CDC Constraints for further information on constraining CDC paths.

Note: There is runtime impact between a False Path constraint and a Max Delay constraint because the paths are timed with Max Delay.

If a maximum delay must be specified for some or for all the paths between two clock domains, then you must use the command set_max_delay -datapath_only to constrain those paths. In this case, set_clock_groups cannot be used to define the two clock domains as asynchronous, as it supersedes the set_max_delay constraint in terms of constraint priority. Other cross clock domains paths must then be constrained with a combination of set_false_path or set_max_delay constraints.

See the following example:

```
set_max_delay <delay> -datapath_only -from <startpoints_source_clock_domain> \
-to <endpoints_destination_clock_domain>
```

Unlike other XDC constraints, the `set_max_delay` command and the `set_min_delay` command can accept, in the case of `-from` and `-to` options, a list of invalid startpoints or endpoints respectively.

When an invalid startpoint is specified, the timing engine stops the propagation of the timing going through the node so that the node becomes a valid startpoint.

In the following example, the only valid startpoint is `FD1/C`:

```
set_max_delay 5 -from [get_pins FD1/C]
```

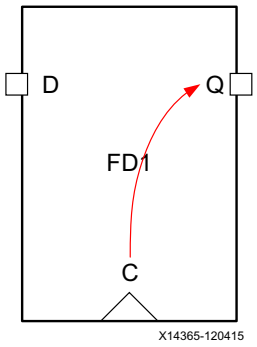


Figure 5-27:

If the constraint is applied to `FD1/Q`, the timing engine stops the propagation through the arc `C->Q` to make the pin `Q` a valid startpoint:

```
set_max_delay 5 -from [get_pins FD1/Q]
```

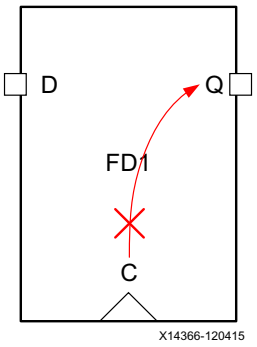


Figure 5-28:

The process of stopping the propagation of the timing to create a valid startpoint is called *path segmentation*. Path segmentation affects both max and min delay analysis. Path segmentation also affects any timing constraint going through those nodes (FD1/C and FD1/Q).

Note: Because of Path Segmentation, no clock insertion delay is used for the launch clock for paths starting from FD1/Q. This can potentially result in large skew because the clock skew of the endpoints is still taken into account. See Figure 5-29.

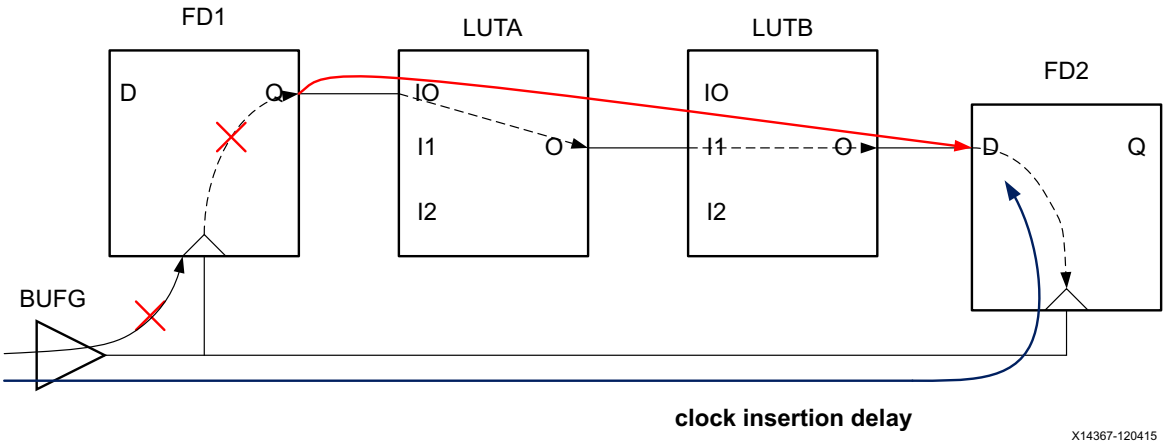


Figure 5-29:

CAUTION! *Path segmentation can have unexpected consequences. Avoid path segmentation altogether, or use it very carefully.*

After path segmentation, there is no default hold requirement on the path. Assuming the `-datapath_only` option has not been specified, use the `set_min_delay` command to set a hold requirement on the path if necessary.

Because of the risks, a critical warning is issued when a path segmentation occurs.

If you targeted the output `FD1/Q` as the startpoint in order to avoid taking the clock skew into account, Xilinx recommends using the `-datapath_only` option. Instead, see the following example:

```
set_max_delay 5 -from [get_pins FD1/C] -datapath_only
```

In the same way, when an invalid endpoint is specified, the timing engine stops the propagation after the node so that the node becomes a valid endpoint.

In the following example, the max delay is specified on `LUTA/O`, which is not a valid endpoint:

```
set_max_delay 5 -from [get_pins LUTA/O]
```

This is shown in the following figure.

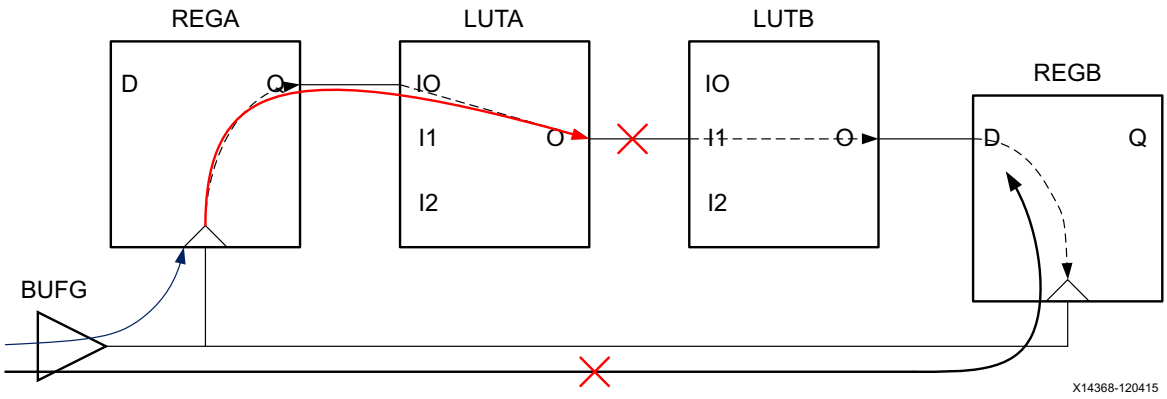


Figure 5-30:

To make `LUTA/O` a valid endpoint, the timing stops propagating after `LUTA/O`. As a result, all timing paths going through `LUTA/O` are impacted for both setup and hold. For the path starting on `REGA/C` and ending on `LUTA/O`, only the insertion delay of the launch clock is taken into account. This can result in very large skew.

Because path segmentation stops the propagation through the timing arcs, it can have unexpected consequences. All the timing paths going through those nodes are impacted.

In the following example, a max delay has been set between `LUTA/O` and `REGB/D`:

```
set_max_delay 6 -from [get_pins LUTA/O] -to [get_pins REGB/D]
```

This is shown in the following figure.

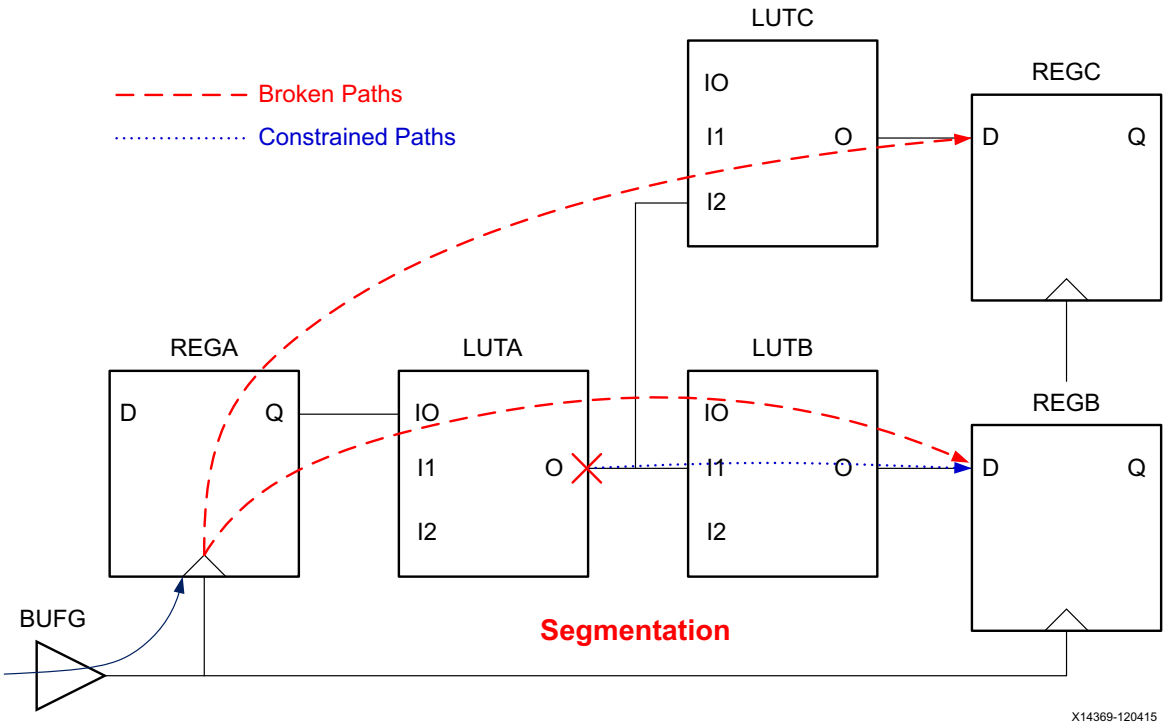


Figure 5-31:

Because the pin `LUTA/O` is not a valid startpoint, a path segmentation occurs and the timing arcs from `LUTA/I*` and `LUTA/O` are broken. Even though the `set_max_delay` constraint was set between `LUTA/O` and `REGB/D` only, other paths such as the path between `REGA/C` and `REGC/D` are also broken.

Path Segmentation and Timing Exception

Path segmentation can result in the perception that the priority between the timing exceptions is altered, which is actually not the case.

There can be a difference on whether a `set_max_delay` constraint is superseded by a `set_clock_groups` constraint. Consider the following two scenarios.

```
set_max_delay <ns> -datapath_only -from <instance> -to <instance>
```

In this scenario, instance names are provided for `-from/-to`. The `set_max_delay` constraint is always overridden by `set_clock_groups -asynchronous`, because Vivado always selects valid startpoints when an instance is provided.

```
set_max_delay <ns> -datapath_only -from <pin> -to <pin | instance>
```

In this scenario, if the pin name provided with `-from` results in path segmentation, then that particular `set_max_delay` constraint is not overriden by `set_clock_groups -asynchronous`. The reason behind is that the path segmentation forces the path starting on the pin name to no longer being considered launched by the first clock domain. As a result, this path is no longer covered by the `set_clock_groups` constraints and the `set_max_delay` constraint get applied.

In some designs, certain signals have a constant value in specific modes. For instance, in functional modes, the test signals do not toggle and are therefore tied either to VDD or VSS depending on their active level. This also applies to signals that do not toggle after the design has been powered up. In the same way, today's designs have multiple functional modes and some signals that are active in some of the functional modes might be inactive in other modes.

To help reduce the analysis space, runtime and memory consumption, it is important to let the Static Timing Engine know about the signals that have a constant value. This is also critical to ensure that non-functional paths and irrelevant paths are not reported.

A signal is declared as inactive to the timing engine with the `set_case_analysis` command. The command applies to pins and/or ports.

Note: After a case analysis is set on a pin, the timing arcs related to that pin are disabled. The timing engine does not report any path going through disabled timing arcs.

VIDEO: *For training on the advanced timing exceptions, including `set_case_analysis`, see the* Vivado Design Suite QuickTake Video: Advanced Timing Exceptions - False Path, Min-Max Delay and Set_Case_Analysis.

The syntax of the `set_case_analysis` command is:

```
set_case_analysis <value> <pins or ports objects>
```

The parameter `<value>` can be any of the following:

```
0, 1, zero, one, rise, rising, fall, or falling
```

When the values `rise`, `rising`, `fall`, or `falling` are specified, this means that the given pins or ports should only be considered for timing analysis with the specified transition. The other transition is disabled.

A case value can be set on a port, a pin of a leaf cell, or a pin of a hierarchical module.

In the example below, two clocks are created on the input pins of the multiplexer clock_sel but only clk_2 is propagated through the output pin after setting the constant value on the selection pin S.

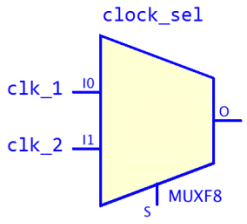


Figure 5-32:

```
create_clock -name clk_1 -period 10.0 [get_pins clock_sel/I0]
create_clock -name clk_2 -period 15.0 [get_pins clock_sel/I1]
set_case_analysis 1 [get_pins clock_sel/S]
```

In the example below, the BUFG_GT has a dynamic clock division as its DIV[2:0] pins driven by some logic instead of being tied to VCC/GND.

Figure 5-33:

In such case, the tool assumes the worst possible scenario for the output clock (divide by 1) and propagates the incoming clock to the buffer output. This worst-case scenario might be pessimistic and over-constrain the design if a clock division of 1 is never exercised. It is possible to control the auto-generated clock on the BUFG_GT output pin by setting the `DIV[2:0]` bus with a `set_case_analysis` constraint.

For example, if the worst-case clock divider is by 3, then the following case analysis should be applied to the BUFG_GT:

```
set_case_analysis 0 [get_pins bufg_gt_pclk/DIV[0] ]
set_case_analysis 1 [get_pins bufg_gt_pclk/DIV[1] ]
set_case_analysis 0 [get_pins bufg_gt_pclk/DIV[2] ]
```

Note: For UltraScale™ and UltraScale+™ devices, the `GT_CHANNEL` has multiple input clocks that propagate to the output of the `GT_CHANNEL` (such as `TXOUTCLK`) through multiple levels of internal muxes. The case analysis can be used in a similar way on the `GT_CHANNEL` clock muxing control signals (such as `TXSYSCLKSEL`, `TXOUTCLKSEL`) to select which of the input or internal clocks should be propagated to the output of the `GT_CHANNEL`.

You can disable timing arcs inside the cell with the `set_disable_timing` command. Only timing arcs going from input to output ports of a cell can be disabled.

Note: The `set_disable_timing` command can also be used to disable a timing arc from a port or a wire. In such cases, the command line options `-from` and `-to` are not used and only the port object(s) or timing arc object(s) are specified.

Some timing arcs are automatically disabled by the timer to handle specific cases. For instance, combinational feedback loops are not recommended and cannot be properly timed. The timer breaks such loops by disabling one of the timing arcs inside the loop.

Another example is a case analysis set on a MUX. By default, all the data inputs of a MUX are propagated to the output port but when a case analysis is set on the select signals, only one data input port gets propagated to the output port. This is done by the timer by breaking timing arcs from the other data input ports to the output port.

The `set_disable_timing` command gives you the ability to manually break cell timing arcs in the design. You can, for example, decide which timing arc(s) of a combinational feedback loop should be disabled to break the loop instead of letting the tool make this determination.

Also, suppose that multiple clocks arrive on a LUT input pins but only one clock should be propagated to the LUT output port. This scenario can be handled by breaking the timing arcs associated to the clocks that should not propagate.

There is also a scenario involving LUTRAM that can be quite frequent. Inside the LUTRAM, there is physical path from WCLK pin to the output O pin between the write and read clocks. However, LUTRAM-base asynchronous FIFO are designed in such way that this CDC path WCLK->O cannot happen by construction. Nevertheless, this timing arc is enabled and can result is the timer reporting paths through this WCLK->O timing arc. This arc can also trigger some TIMING-10 DRC violations. In such case, the user should disable the WCLK->O arc so that those paths are not timed and reported and that they do not trigger invalid DRC violations. This timing arc is automatically disabled in the current implementation of the Xilinx LUTRAM-based FIFO.

Note: After a timing arc is disabled, no timing path will be reported by the timer through this arc. You should be very careful to not disable any valid timing arc. This might result is masking some timing violations and/or timing problems that could result in the design failing in hardware.

The syntax for the set_disable_arc command is:

```
set_disable_timing  [-from <arg>] [-to <arg>] [-quiet] [-verbose] <objects>
```

Only pin names and not Vivado tools objects can be provided to the -from and -to command line options. The pin names should also match pin names from the library cell, not design pin names. For example:

```
set_disable_timing -from WCLK -to O [get_cells inst_fifo_gen/ gdm.dm/gpr1.dout_i_reg[*]]
```

The above command disables the WCLK->O timing arcs for all the LUTRAM-based asynchronous FIFOs inst_fifo_gen/ gdm.dm/gpr1.dout_i_reg[*].

The command line options -from and -to are optional. If -from is not specified, then all the timing arcs ending on the pin specified with -to are being disabled. In the same way if -to is not specified, then all the timing arcs starting on the pin specified with -from are being disabled. If neither -from nor -to are specified, then all the timing arcs of the cells specified in the command are disabled.

You can use the command report_disable_timing to list all the timing arcs that have been automatically disabled by the timer as well as manually disabled by the user. Be careful as the list can be very large. Use the -file command line option to save the result in a file.

Note: report_disable_timing can be scoped to one or more hierarchical module(s) with -cells.

XILINX

Chapter 6

CDC Constraints

Clock Domain Crossing (CDC) constraints apply to timing paths that have a different launch and capture clock. There are synchronous CDC and asynchronous CDC depending on the launch and capture clocks relationship and on the timing exceptions set on the CDC paths. For example, CDC paths between synchronous clocks but covered by false path constraints are not timed, and consequently are treated as asynchronous CDCs.

Asynchronous CDC paths can be safe or unsafe. The terminology of *safe* and *unsafe* for asynchronous CDC paths is different from the terminology used for inter-clock timing analysis (see `report_clock_interaction`). An asynchronous CDC path is considered safe when it uses a synchronization circuitry to prevent metastability of the capture sequential cell.

For more information, refer to this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 4].

The timing analysis of CDC paths can be fully ignored by using `set_false_path` or `set_clock_groups` constraints, or partially analyzed by using `set_max_delay -datapath_only`. In addition, the multi-bit CDC paths capture time spread can be constrained using the `set_bus_skew` constraint.

The bus skew constraint is used to set a maximum skew requirement between several asynchronous CDC paths. The bus skew is not the traditional clock skew associated with a timing path. Instead, it corresponds to the largest capture time difference across all the paths that are covered by a same `set_bus_skew` constraint. The bus skew requirement applies to both Fast and Slow corners, but it is not analyzed across the corners.

The intent of the bus skew constraint is to limit the number of source clock edges that can launch a data and be captured by a single destination clock edge. The tolerance depends on the CDC synchronization scheme used for the constrained paths. The bus skew constraint is typically used for the following CDC topologies:

- Gray-coded bus transfer, such as in asynchronous FIFOs
- Multi-bit CDC implemented with CE, MUX, or MUX Hold circuitry
- Configuration registers

Although the `set_bus_skew` command does not prevent a bus skew constraint to be set on a safely timed synchronous CDC, such a constraint is not needed. The setup and hold checks already ensure a safe transfer between two safely timed synchronous CDC paths.

The CDC scenarios for bus skew constraints are:

- Asynchronous CDC covered with `set_clock_groups`
- Asynchronous CDC entirely covered with `set_false_path` and/or `set_max_delay -datapath_only`
- Synchronous CDC paths covered with `set_false_path` and/or `set_max_delay -datapath_only`

The bus skew constraint is not a timing exception; rather, it is a timing assertion. Therefore, it does not interfere with the timing exceptions (`set_clock_group`, `set_false_path`, `set_max_delay`, `set_max_delay -datapath_only`, and `set_multicycle_path`) and their precedence.

The bus skew constraint is only optimized by the `route_design` command. To report the `set_bus_skew` constraints, use the `report_bus_skew` command from the command line or **Reports** > **Timing** > **Report Bus Skew** from the GUI. The bus skew constraints are not reported inside the Timing Summary report (`report_timing_summary`).

The syntax of the `set_bus_skew` command with the basic options is:

```
set_bus_skew [-from <args>] [-to <args>] [-through <args>] <value>
```

The list of objects for the `-from` option should be a list of valid startpoints. A valid startpoint for `set_bus_skew` is a clock, or the clock pin of a sequential element, such as a register or a RAM. An input (or inout) port is not supported by `set_bus_skew`.

The list of nodes for the `-to` option should be a list of valid endpoints. A valid endpoint for `set_bus_skew` is a clock, or the data pin of a sequential cell. An output (or inout) port is not supported by `set_bus_skew`.

The list of nodes for the `-through` option should be a list of valid pins, or nets.

Although the `-from` and `-to` command line options can refer to clocks, Xilinx recommends that you be more specific and specify a limited list of startpoints and endpoints per constraint. This will ensure that not too many paths get covered by each constraint and that each constraint can be reasonably met.

Note: Both the `-from` and `-to` options must be specified when specifying a bus skew constraint.

Note: Xilinx recommends setting a bus skew constraint on paths with no fanout. Also, each bus skew constraint must cover at least two startpoints and two endpoints.

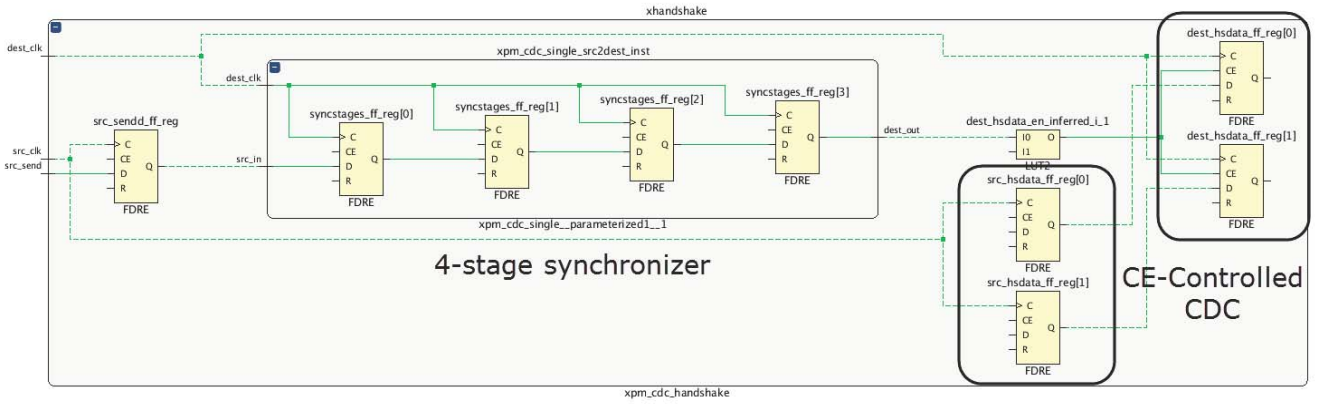
The bus skew value must be realistic and reasonable. Xilinx recommends to use a value larger than half the minimum period of the source and destination clocks. The recommended value for the bus skew also depends on the CDC topology as illustrated by the following examples.

set_bus_skew Example One

In this example, the CDC is part of a handshake mechanism. The source clock domain generates a *send* signal when data is available to be sampled. The destination clock domain uses a 4-stage synchronizer for the *send* signal. After the 4-stage synchronizer, the signal drives the Clock Enable pin of the CDC destination registers. In such Clock-Enabled Control CDC structure, the bus skew must be adjusted to the number of stages on the CE path since it represents the number of destination clock cycles for which the data is valid.

If the source clock period is 5 ns and the destination clock period is 2.5 ns, the bus skew on the CDC path should be set to 10 ns (4×2.5 ns).

```
set_bus_skew -from [get_cells src_hsdata_ff_reg*] -to [get_cells dest_hsdata_ff_reg*] 10.000
```



X18887-031717

Figure 6-1:

Note: For completeness, the CDC needs an additional `set_max_delay` constraint to ensure that the source and destination registers are not placed too far apart:

```
set_max_delay -datapath_only -from [get_cells src_hsdata_ff_reg*] -to [get_cells
dest_hsdata_ff_reg*] 10.000
```

set_bus_skew Example Two

In this example, the CDC is on a gray-coded bus. The system must ensure that only one transition of the gray-coded bus is captured by the destination clock domain at the same time.

If the source clock period is 5 ns and the destination clock period is 2.5 ns, the bus skew on the CDC path should be set to 2.5 ns (destination clock period).

```
set_bus_skew -from [get_cells src_gray_ff_reg*] -to [get_cells {dest_graysync_ff_reg[0]*}]
2.500
```

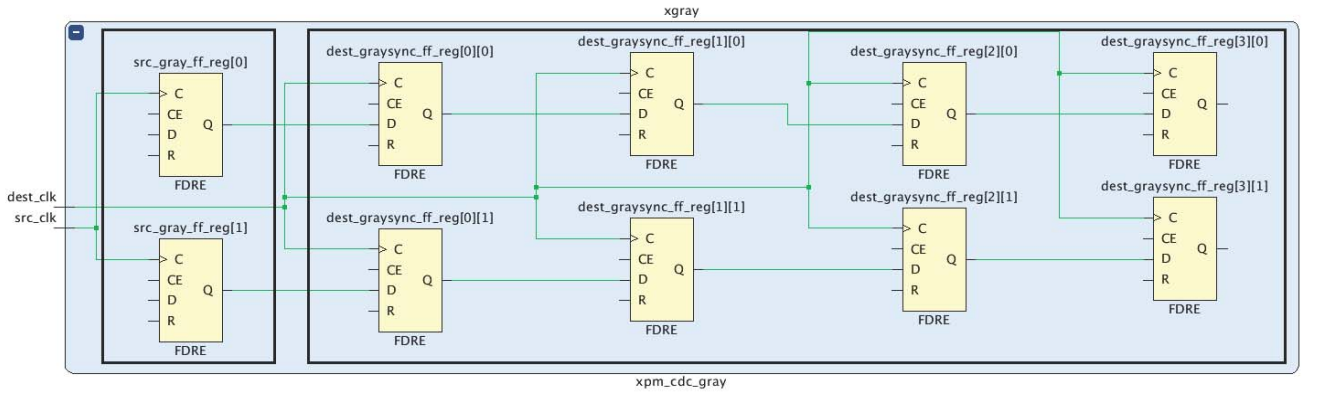


Figure 6-2:

Note: For completeness, the CDC needs an additional `set_max_delay` constraint to ensure that the source and destination registers are not placed too far apart. In this case, the max delay is set to the source clock period as the CDC is between a slower clock to a faster clock and only one transition of the bus should be captured by the destination clock domain:

```
set_max_delay -datapath_only -from [get_cells src_gray_ff_reg*] -to [get_cells
{dest_graysync_ff_reg[0]*}] 5.000
```

In the Vivado IDE, you can set bus skew constraints in multiple ways:

- Through the Timing Constraints Editor. Select **Window > Timing Constraint > Assertion > Set Bus Skew**.

  From the Timing Constraints Editor, you can add, remove, or modify bus skew constraints.

  Note: Locked IP bus skew constraints cannot be edited.

- Through the Report CDC GUI. Select **Reports > Timing > Report CDC**.

Inside the CDC Details tables, you must select one or more rows to include at least two or more startpoints and two or more endpoints. When you right-click and select **Set Bus Skew**, there are two options:

- Startpoint to Endpoint:

  Set a bus skew constraint between the startpoints and endpoints included in the selected row(s).

- Source Clock to Destination Clock:

  Set bus skew constraints between the clock domains of the startpoints and endpoints.

  Note: It is typically not recommended to set a bus skew constraints between clock domains, because it will apply to more paths than needed. This will result in longer implementation runtime and impossible timing closure.

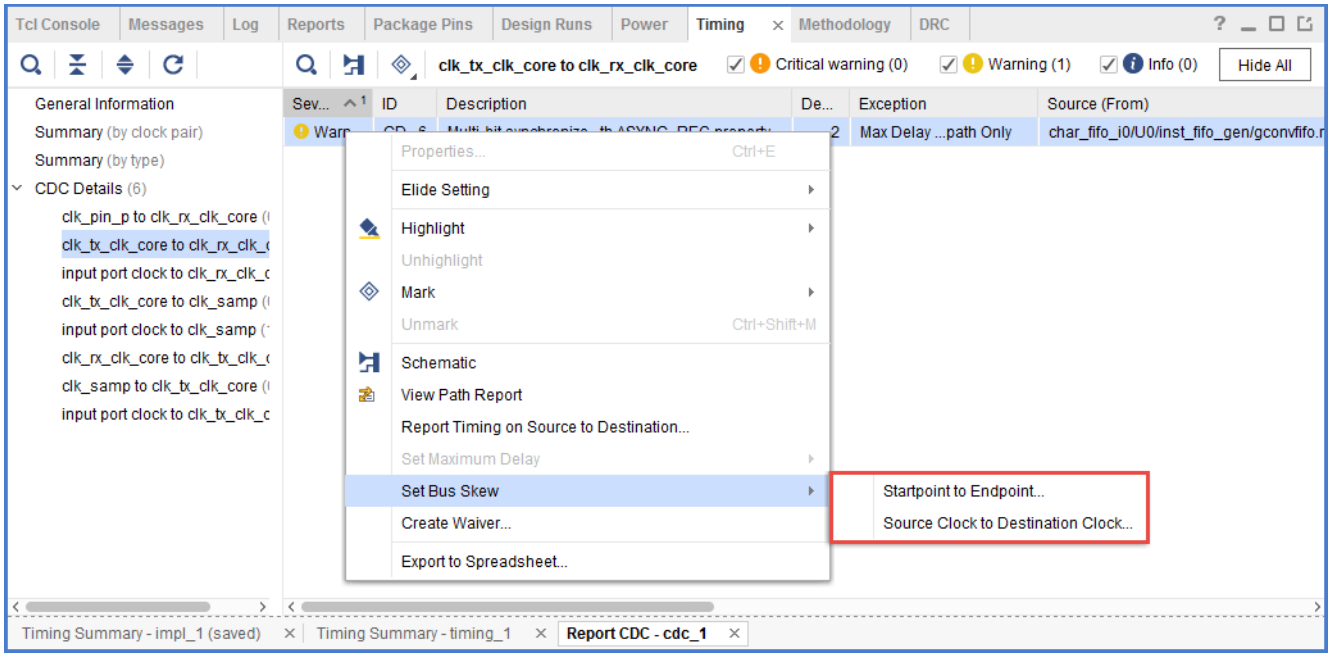


Figure 6-3:

Note: Vivado does not verify the validity of setting a bus skew constraint on the selected objects. You must ensure that a bus skew constraint makes sense with the selected objects.

In the Set Bus Skew dialog box, you can set the bus skew value, the startpoints, and endpoints, as shown in the following figure.

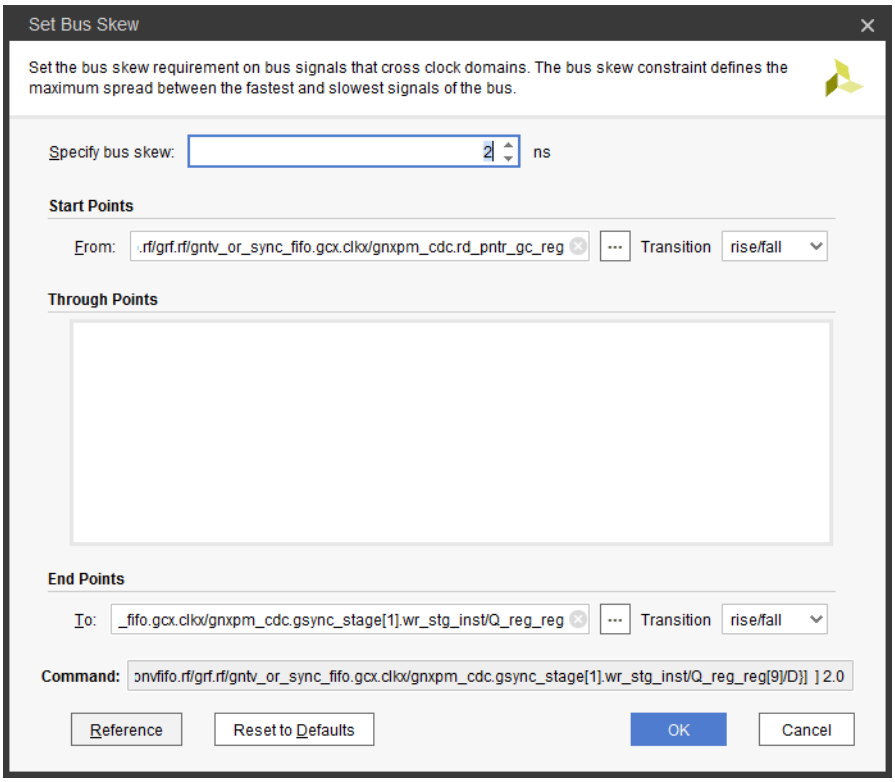


Figure 6-4:

XDC Precedence

The precedence rules for Xilinx® Design Constraints (XDC) are inherited from Synopsys Design Constraints (SDC). This chapter discusses how constraint conflicts or overlaps are resolved.

Constraints Order

XDC constraints are commands interpreted sequentially. For equivalent constraints, the last constraint takes precedence.

Constraints Order Example:

```
> create_clock -name clk1 -period 10 [get_ports clk_in1]
> create_clock -name clk2 -period 11 [get_ports clk_in1]
```

In this example, the second clock definition overrides the first clock definition because:

- They are both attached to the same input port.
- The `create_clock -add` option was not used.

Constraints Priority

If constraints overlap (for example, if several timing exceptions are applied to the same path), the priority from highest to lowest is:

1. Clock Groups (`set_clock_groups`)
2. False Path (`set_false_path`)
3. Maximum Delay Path (`set_max_delay`) and Minimum Delay Path (`set_min_delay`)
4. Multicycle Paths (`set_multicycle_path`)

Note: The `set_bus_skew` constraint does not affect the above constraints precedence. The `set_bus_skew` constraint does not override and is not overridden by clock groups, max delays, false paths, and multicycle paths. The reason is that the bus skew is not a constraint on a particular path, but a constraint between paths.

Note: The priority between the False Path, Maximun/Minimum Delay and Multicycle Path can be altered using the option `-reset_path`. The Clock Group constraint cannot be overriden. A Maximum/Minimum Delay or Multicycle Path constraint can only override a previously defined False Path or Maximum/Minimum Delay constraint when both constraints are defined with the exact same arguments for `-from/-to/-through` and the latest constraint uses `-reset_path`.

In addition, for the same type of exception, the more specific the constraint, the higher the precedence. Depending on the filtering options and the type of objects used in the constraint, you can modify the specificity of a constraint.

The priority rule for the objects is:

1. Ports, pins, and cells

   Pins of a cell are used instead of the cell itself.

2. Clocks

   Clocks always have lower priority than ports, pins, and cells. A timing exception that uses clock object(s) always has a lower priority than another timing exception defined with ports, pins, and cells.

The precedence rule for the filters, from highest to lowest, is:

1. `-from -through -to`
2. `-from -to`
3. `-from -through`
4. `-from`
5. `-through -to`
6. `-to`
7. `-through`

IMPORTANT: *Note that cells used in either the `-from` or `-to`, always have a higher precedence than a clock even if the clock is used in a more specific case of `-from -through -to`.*

```
> set_max_delay 12 -from [get_clocks clk1] -to [get_clocks clk2]
> set_max_delay 15 -from [get_clocks clk1]
```

In this example, the first constraint overrides the second constraint for the paths from clk1 to clk2.

The number of -through options used in an exception does not affect the precedence. The timing engine uses the tightest constraint.

```
> set_max_delay 12 -from [get_cells inst0] -to [get_cells inst1]
> set_max_delay 15 -from [get_clocks clk1] -through [get_pins hier0/p0] -to
[get_cells inst1]
```

In this example, the first constraint only uses cell objects and the second constraint uses a clock object. Although inst0 is clocked by clk1, the first constraint overrides the second constraint for the paths from cell inst0 to cell inst1.

```
> set_max_delay 4 -through [get_pins inst0/I0]
> set_max_delay 5 -through [get_pins inst0/I0] -through [get_pins inst1/I3]
```

Both exceptions are kept by the timing engine. The more challenging constraint is used for timing analysis. In this example, the 4 ns max delay constraint will be used even for paths going through the pin inst1/I3.

```
> set_false_path -from [get_clocks clkA] -to [get_clocks clkB]
> set_max_delay 1 -from [get_clocks clkA] -to [get_clocks clkB] -reset_path
```

The paths between clocks clkA and clkB are covered by the Max Delay with a path requirement of 1ns. The Max Delay is defined with the same arguments for -from/-to and specifies -reset_path, which overrides the False Path.

```
> set_false_path -from [get_clocks clkA] -to [get_clocks clkB]
> set_max_delay 1 -from [get_pins reg0/CLK] -to [get_pins reg1/D] -reset_path
```

The paths between reg0/CLK and reg1/D are covered by the False Path since that constraint has a higher precedence over the Max Delay. The Max Delay doesn't override the False Path despite the -reset_path as it is not defined with the same arguments for -from/-to.

RECOMMENDED: *You must avoid using several timing exceptions on the same paths, so that the timing analysis results are not dependent on priority rules, and it is easier to validate the effect of your constraints.*

It is recommended that you validate the timing exceptions with the report_exceptions command. This command provides insight on which timing exceptions are overriden or ignored. For more information, refer to Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 4].

If a string instead of an object is passed to the constraint, the Tcl interpreter uses the following sequence to determine which object matches the string:

1. **port**
2. **pin**
3. **cell**
4. **net**

The search is not exhaustive. As soon as objects of a certain type match the string pattern, they are returned, even though objects of another type down the list might also match the same pattern.

XILINX

Chapter 8

Physical Constraints

The Xilinx® Vivado® Integrated Design Environment (IDE) enables design objects to be physically constrained by setting values of object properties. Examples include:

- I/O constraints such as location and I/O standard
- Placement constraints such as cell locations
- Routing constraints such as fixed routing
- Configuration constraints such as the configuration mode

Similar to timing constraints, physical constraints must be saved in an Xilinx Design Constraints (XDC) file or a Tcl script so that they can be loaded with the netlist when you open a design. After the design is loaded in memory, you can interactively enter new constraints using the Tcl console, or by using one the Vivado Design Suite IDE editing tools.

Most physical constraints are defined by means of properties on an object:

```
set_property <property> <value> <object list>
```

The exception is for area constraints which use Pblock commands.

Critical Warnings are issued for invalid constraints in XDC files, including those applied to objects that cannot be found in the design.

For property definition and usage, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 11].

RECOMMENDED: *Xilinx highly recommends that you review all Critical Warnings to ensure that the design is properly constrained. Invalid constraints result in errors when applied interactively.*

Netlist constraints are set on netlist objects such as ports, pins, nets or cells, to require synthesis and implementation to handle them in special way.

IMPORTANT: *Be sure that you understand the impact of using these constraints. They might result in increased design area, reduced design performance, or both.*

Netlist constraints include:

- CLOCK_DEDICATED_ROUTE
- MARK_DEBUG
- DONT_TOUCH
- LOCK_PINS

Set `CLOCK_DEDICATED_ROUTE` on a net to indicate how the clock signal is expected to be routed.

The `CLOCK_DEDICATED_ROUTE` property is used on a clock net to override the default routing. This is an advanced control requiring extreme caution as it might affect timing predictability and routability.

For example, `CLOCK_DEDICATED_ROUTE` can be set to `FALSE` when dedicated clock routing is not available. A value of FALSE allows the Vivado tools to route the clock from an input port to a global clocking resource such as a BUFG or MMCM using general routing resources. This should only be used as a last resort when device package pin assignments have been locked down, and the clock input cannot be assigned to an appropriate clock capable input pin (`CCIO`). The routing will be suboptimal and unpredictable unless used in conjunction with `FIXED_ROUTE`.

For more information about this property, see Clock Constraints in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 5].

Set `MARK_DEBUG` on a net in the RTL to preserve it and make it visible in the netlist. This allows it to be connected to the logic debug tools at any point in the compilation flow.

For more information, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 12].

Set DONT_TOUCH on a leaf cell, hierarchical cell, or net object to preserve it during netlist optimizations. DONT_TOUCH is most commonly used to:

- Prevent a net from being optimized away.

  A net with DONT_TOUCH cannot be absorbed by synthesis or implementation. This can be helpful for logic probing or debugging unexpected optimization in designs. To preserve a net with multiple hierarchical segments, place DONT_TOUCH on the net PARENT (`get_property PARENT $net`) which is the net segment closest to its driver.

- Prevent merging of manually replicated logic.

  Sometimes it is best to manually replicate logic, such as a high-fanout driver that spans a wide area. Adding DONT_TOUCH to the manually replicated drivers (as well as the original) prevents synthesis and implementation from optimizing these cells.

Note: Use `reset_property` to reset the DONT_TOUCH property. Setting the DONT_TOUCH property to 0 does not reset the property.

TIP: *Avoid using DONT_TOUCH on hierarchical cells for implementation as Vivado IDE implementation does not flatten logical hierarchy. Use KEEP_HIERARCHY in synthesis to maintain logical hierarchy for applying XDC constraints.*

LOCK_PINS is a cell property used to specify the mapping between logical LUT inputs (I0, I1, I2, ...) and LUT physical input pins (A6, A5, A4, ...).

A common use is to force timing-critical LUT inputs to be mapped to the fastest A6 and A5 physical LUT inputs.

LOCK_PINS Constraint Example One

Map I1 to A6 and I0 to A5 (swap the default mapping).

```
% set myLUT2  [get_cells u0/u1/i_365]
% set_property LOCK_PINS {I0:A5 I1:A6} $myLUT2
# Which you can verify by typing the following line in the Tcl Console:
% get_property LOCK_PINS $myLUT2
```

LOCK_PINS Constraint Example Two

Map I0 to A6 for a LUT6, mapping of I1 through I5 are dont-cares.

```
% set_property LOCK_PINS I0:A6 [get_cell u0/u1/i_768]
```

I/O constraints configure:

- Ports
- Cells connected to ports

Typical constraints include:

- I/O standard
- I/O location

The Vivado Design Suite supports many of the same I/O constraints as the Integrated Software Environment (ISE®) Design Suite. The following list of I/O properties is not exhaustive.

    - For a complete list of I/O properties, more information on I/O port and I/O cell properties, and coding examples with proper syntax, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 11].

      Note: All properties are applied to port objects unless otherwise stated.

    - For more information on the application and methodology behind these properties, see the device SelectIO documents, for example *7 Series FPGAs SelectIO Resources User Guide* (UG471) [Ref 13].

- **DRIVE**

  Sets the output buffer drive strength (in mA), available with certain I/O standards only.

- **IOSTANDARD**

  Sets an I/O Standard,

- **SLEW**

  Sets the slew rate (the rate of transition) behavior of a device output.

- **IN_TERM**

  Sets the configuration of the input termination resistance for an input port

- **DIFF_TERM**

  Turns on or off the 100 ohm differential termination for primitives such as `IBUFDS_DIFF_OUT`.

- **KEEPER**

  Applies a weak driver on an tri-stateable output or bidirectional port to preserve its value when not being driven.

- **PULLTYPE**

  Applies a weak logic low or high level on a tri-stateable output or bidirectional port to prevent it from floating.

- **DCI_CASCADE**

  Defines a set of master and slave banks. The DCI reference voltage is chained from the master bank to the slaves. `DCI_CASACDE` is set on IOBANK objects.

- **INTERNAL_VREF**

  Frees the Vref pins of an I/O Bank and uses an internally generated Vref instead. `INTERNAL_VREF` is set on IOBANK objects

- **IODELAY_GROUP**

  Groups a set of `IDELAY` and `IODELAY` cells with an `IDELAYCTRL` to enable automatic replication and placement of `IDELAYCTRL` in a design.

- **IOB**

  Tells the placer to try to place FFs in I/O Logic instead of the fabric slice. This property must be assigned to the register and not to the port.

IMPORTANT: *There are notable differences between the ISE Design Suite and the Vivado Design Suite in the handling of IOB. The Vivado tools allow IOB to be set on both ports and on register cells connected to ports. If conflicting values are set on a port and its register, the value on the register prevails. The Vivado tools use only the values TRUE and FALSE. The value FORCE is interpreted as TRUE, and the value AUTO is ignored. Unlike ISE, if a setting of IOB true cannot be honored, the Vivado tools generate a critical warning, not an error.*

- **IOB_TRI_REG**

  For HDIO in UltraScale+ devices. Tells the placer to try to place FFs driving Tristate signals on HDIO bank IOBs in the I/O Logic instead of the fabric slice. This property must be assigned to the register and not to the port.

Placement constraints are applied to cells to control their locations within the device. The Vivado Integrated Design Environment (IDE) supports many of the same placement constraints as the Integrated Software Environment (ISE) Design Suite and the PlanAhead™ tool.

- **LUTNM**

  A unique string name applied to two LUTs to control their placement on a single LUT site. Unlike `HLUTNM`, `LUTNM` can be used to combine LUTs that belong to different hierarchical cells.

- **HLUTNM**

  A unique string name applied to two LUTs in the same hierarchy to control their placement on a single LUT site.

  Use `HLUTNM` within a cell that is instantiated multiple times.

- **PROHIBIT**

  Disallows placement to a site.

- **PBLOCK**

  Attached to logical blocks to constrain them to a physical region in the device.

  `PBLOCK` is a read-only cell property that is the name of the Pblock to which the cell is assigned. Cell Pblock membership can be changed only by using the XDC Tcl commands `add_cells_to_pblock` and `remove_cells_from_pblock`.

- **PACKAGE_PIN**

  Specifies the location of a design port on a pin of the target device package.

- **LOC**

  Places a logical element from the netlist to a site on the device.

- **BEL**

  Places a logical element from the netlist to a specific BEL within a slice on the device.

For more information, see:

- Chapter 7, XDC Precedence
- Chapter 9, Defining Relatively Placed Macros

There are two types of placement in the tools:

- Fixed Placement
- Unfixed Placement

Fixed Placement

Fixed placement is placement specified by the user through one of the following:

- Hand placement
- An XDC constraint
- Using either `IS_LOC_FIXED` or `IS_BEL_FIXED` on a cell object of the design loaded in memory.

Unfixed Placement

Unfixed placement is a placement performed by the implementation tools. By setting the placement as fixed, the implementation cannot move the constrained cells during the next iteration or during an incremental run. A fixed placement is saved in the XDC file, where it appears as a simple LOC or BEL constraint.

- `IS_LOC_FIXED`

  Promotes a LOC constraint from unfixed to fixed.

- `IS_BEL_FIXED`

  Promotes a BEL constraint from unfixed to fixed.

Locate a block RAM at `RAMB18_X0Y10` and fix its location.

```
% set_property LOC RAMB18_X0Y10 [get_cells u_ctrl0/ram0]
```

Place a LUT in the C5LUT BEL position within a slice and fix its BEL assignment.

```
% set_property BEL C5LUT [get_cells u_ctrl0/lut0]
```

Locate input bus registers in ILOGIC cells for shorter input delay.

```
% set_property IOB TRUE [get_cells mData_reg*]
```

Combine two small LUTs into a single LUT6_2 that uses both O5 and O6 outputs.

```
% set_property LUTNM L0 [get_cells {u_ctrl0/dmux0 u_ctrl0/dmux1}]
```

Prevent the placer from using the first column of block RAMs.

```
% set_property PROHIBIT TRUE [get_sites {RAMB18_X0Y* RAMB36_X0Y*}]
```

Prevent the placer from using the clock region X0Y0.

```
% set_property PROHIBIT TRUE [get_sites -of [get_clock_regions X0Y0]]
```

Prevent the placer from using SLR0.

```
% set_property PROHIBIT TRUE [get_sites -of [get_slrs SLR0]]
```

IMPORTANT: *When assigning both BEL and LOC properties to a cell, BEL must be assigned before LOC.*

Routing constraints are applied to net objects to control their routing resources.

Fixed Routing is the mechanism for locking down routing, similar to Directed Routing in ISE. Locking down a net routing resources involves three net properties. See Table 8-1.

Table 8-1:

ROUTE	Read-only net property
IS_ROUTE_FIXED	Flag to mark the whole route as fixed
FIXED_ROUTE	The fixed-route portion of a net

To guarantee that a net routing can be fixed, all of its cells must also be fixed in advance.

Following is an example of a fully-fixed route. The example takes the design in Figure 8-1 and creates the constraints to fix the routing of the net `netA` (selected in blue).

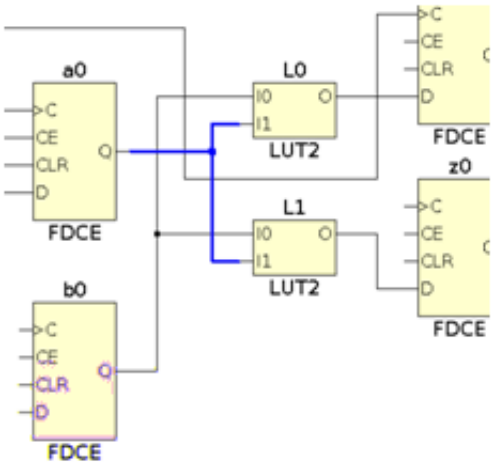


Figure 8-1:

You can query the routing information of any net after loading the implemented design in memory:

```
% set net [get_nets netA]
% get_property ROUTE $net
{ CLBLL_LL_CQ CLBLL_LOGIC_OUTS6 FAN_ALT5 FAN_BOUNCE5  { IMUX_L17 CLBLL_LL_B3 }
IMUX_L11 CLBLL_LL_A4 }
```

The routing is defined as a series of relative routing node names with fanout denoted using embedded curly braces. The routing is fixed by setting the following property on the net:

```
% set_property IS_ROUTE_FIXED TRUE $net
```

To back-annotate the constraints in your XDC file for future runs, the placement of all the cells connected to the fixed net must also be preserved. You can query this information by selecting the cells in the schematics or device view, and look at their LOC/BEL property values in the Properties window. Or, you can query those values directly from the Tcl console:

```
% get_property LOC [get_cells {a0 L0 L1}]
SLICE_X0Y47 SLICE_X0Y47 SLICE_X0Y47
% get_property BEL [get_cells {a0 L0 L1}]
SLICEL.CFF SLICEL.A6LUT SLICEL.B6LUT
```

Because fixed routes are often timing-critical, LUT pins mapping must also be captured in the LOCK_PINS property of the LUT to prevent the router from swapping pins.

Again, you can query the site pin of each logical pin from the Tcl console:

```
% get_site_pins -of [get_pins {L0/I1 L0/I0}]
SLICE_X0Y47/A4 SLICE_X0Y47/A2
% get_site_pins -of [get_pins {L1/I1 L1/I0}]
SLICE_X0Y47/B3 SLICE_X0Y47/B2
```

The complete XDC constraints required to fix the routing of net `netA` are:

```
set_property BEL CFF [get_cells a0]
set_property BEL A6LUT [get_cells L0]
set_property BEL B6LUT [get_cells L1]
set_property LOC SLICE_X0Y47 [get_cells {a0 L0 L1}]
set_property LOCK_PINS {I1:A4 I0:A2} [get_cells L0]
set_property LOCK_PINS {I1:A3 I0:A2} [get_cells L1]
set_property FIXED_ROUTE { CLBLL_LL_CQ CLBLL_LOGIC_OUTS6 FAN_ALT5 FAN_BOUNCE5  {
IMUX_L17 CLBLL_LL_B3 }  IMUX_L11 CLBLL_LL_A4 } [get_nets netA]
```

If you are using interactive Tcl commands instead of XDC, several placement constraints can

XILINX

Chapter 9

Defining Relatively Placed Macros

A Relatively Placed Macro (RPM) is a list of basic logic elements (BELs) grouped into a set. Examples of logic elements include:

- FF
- LUT
- DSP
- RAM

RPMs are primarily used to place small groups of logic close together in order to improve resource efficiency and enable faster interconnections.

Define sets of design elements with U Set (`U_SET`) or HU Set (`HU_SET`) constraints.

- Each element of the set is placed in relation to the other elements of the set by Relative Location (`RLOC`) constraints.
- Logic elements with RLOC constraints and common set names are associated in an RPM.

`U_SET`, `HU_SET`, and `RLOC` constraints:

- Must be defined as properties in the HDL design files.
- Are not supported in Xilinx® Design Constraints format (XDC).

TIP: *You can use the `create_macro` and `update_macro` commands to define macro objects in the Vivado® Design Suite, that act like RPMs within the design. Refer to* *XDC Macros, page 170*.

For more information on `U_SET`, `HU_SET`, and `RLOC` constraints, see the *Vivado Design Suite Properties Reference Guide* (UG912) [Ref 11].

To create an RPM:

1. Group cells into a set.
2. Define relative locations for cells in the RPM set.
3. Specify an RLOC_ORIGIN constraint or a LOC constraint on an RPM cell to fix placement of the RPM on the target device.

   Note: This step is optional.

Design elements in a hierarchical module that are assigned RLOC constraints are automatically grouped into an RPM set.

The grouping occurs by using an H_SET constraint that is implicitly defined by the combination of the design hierarchy and the RLOC constraint.

All design elements with RLOC constraints in a single block of the design hierarchy are considered to be in the same H_SET unless they are tagged with another set constraint, such as U_SET or HU_SET.

While H_SET is implied based on the design hierarchy and the presence of the RLOC constraint, you can also explicitly group design elements into RPM sets using the U_SET and HU_SET constraints.

Explicitly Grouping Design Elements With U_SET

U_SET lets you group cells regardless of hierarchy or where they appear in the design. All cells with the same set_name are members of the same RPM set.

Design elements tagged with a U_SET constraint can be primitive or non-primitive symbols.

When attached to non-primitive symbols, the U_SET constraint propagates downward through the hierarchy to all the primitive symbols below it that are assigned RLOC constraints.

Explicitly Grouping Design Elements With HU_SET

HU_SET has an explicit user-defined and hierarchically qualified name for the set. This lets you create hierarchical RPMs in which RLOC constraints can be placed on cells at different levels of the hierarchy.

All cells with the same hierarchically qualified set_name are members of the same set.

The syntax for defining RPM sets as attributes in VHDL is:

```
attribute U_SET : string;
attribute HU_SET : string;
...
attribute U_SET of my_reg : label  is "uset0";
attribute HU_SET of other_reg : label is "huset0";
```

The syntax for defining RPM sets as attributes in Verilog is as follows.

U_SET Example

```
(* U_SET = "uset0", RLOC = "X0Y0" *) FD my_reg (.C(clk), .D(d0), .Q(q0));
```

HU_SET Example

```
(* HU_SET = "huset0", RLOC = "X0Y0" *) FD other_reg (.C(clk), .D(d1), .Q(q1));
```

RECOMMENDED: *When using H_SET and HU_SET RPMs with Vivado Synthesis, preserve the hierarchical boundary of the module or instance containing the RPMs. This avoids naming collisions between RPMs at the same hierarchical level as a result of hierarchy being dissolved. For further information on hierarchy preservation see the Vivado Design Suite User Guide: Synthesis (UG901) [Ref 8].*

RPM Definition in the Physical Constraints Window

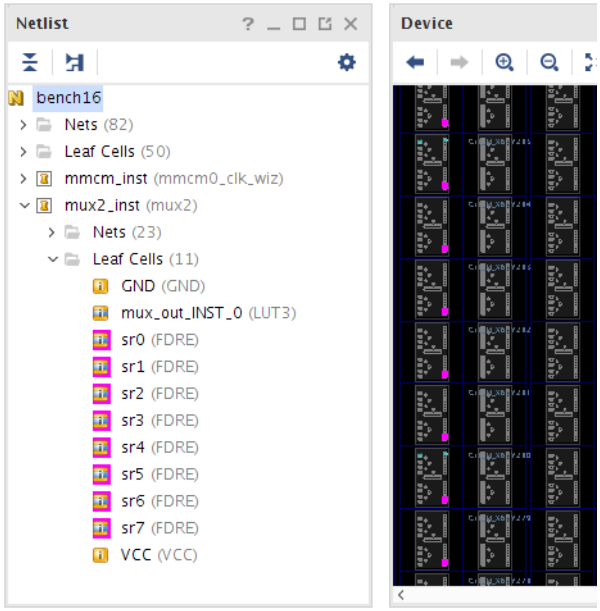


Figure 9-1:

RPM sets must be embedded as properties in HDL source files. After synthesis, RPM related properties appear on netlist objects as read only properties for use by the Xilinx Vivado Integrated Design Environment (IDE) placer.

View RPM definitions in the Physical Constraints window. See Figure 9-1.

To view RPM definitions:

1. Expand the RPM folder to display a list of RPMs.
2. Select an RPM to view its properties or to select related cells.

TIP: *RPMs can be placed and locked down by dragging from the Physical Constraints to the Device window. The RPMs are moved as a single shape instead of cell-by-cell.*

`opt_design` is free to optimize and remove some LUTs that belong to an RPM despite the RLOC constraint. To prevent `opt_design` from optimizing the logic inside an RPM, it is necessary to set the property `DONT_TOUCH` to `TRUE` on all the cells that belong to the RPM. The `DONT_TOUCH` property can be set either through RTL or XDC.

Use the RLOC property to assign relative locations to design objects. The RLOC property specifies relative X-Y coordinates for each cell in the RPM set.

To specify the RLOC property, use either of two different grid coordinate systems:

- Relative Slice-Based Coordinates
- Absolute RPM Grid-Based Coordinates

Use the following syntax:

```
RLOC=XmYn
```

where

- m is an integer representing the relative or absolute X coordinate of the object.
- n is an integer representing the relative or absolute Y coordinate of the object.

The relative grid system:

- Is also known as the standard grid.
- Is sufficient for most RPMs.
- Is used for homogeneous RPMs in which all cells in an RPM belong to the same site type (such as slice, block RAM, and DSP).

  Note: Objects are positioned in relation to other objects in the same RPM set.

The relative grid is a standard rectangular grid in which each grid element is the same size. For example, the following Verilog code example results in an eight-slice-high column with an FD cell in each slice:

```
(* RLOC = "X0Y0" *) FD sr0 (.C(clk), .D(d[0]), .Q(y[0]));
(* RLOC = "X0Y1" *) FD sr1 (.C(clk), .D(d[1]), .Q(y[1]));
(* RLOC = "X0Y2" *) FD sr2 (.C(clk), .D(d[2]), .Q(y[2]));
(* RLOC = "X0Y3" *) FD sr3 (.C(clk), .D(d[3]), .Q(y[3]));
(* RLOC = "X0Y4" *) FD sr4 (.C(clk), .D(d[4]), .Q(y[4]));
(* RLOC = "X0Y5" *) FD sr5 (.C(clk), .D(d[5]), .Q(y[5]));
(* RLOC = "X0Y6" *) FD sr6 (.C(clk), .D(d[6]), .Q(y[6]));
(* RLOC = "X0Y7" *) FD sr7 (.C(clk), .D(d[7]), .Q(y[7]));
```

For complex structures, the BEL or LOC constraints may need to be specified in addition to the RLOC. The BEL constraint must be used to align the cells inside the RPM set, for example, to align the LUTs with the registers. The LOC constraint is uncommon and typically not used because the RPM set is forced on a specific site in the device and cannot be moved by the placer. Whenever some BEL or LOC constraints need to be specified, it is important to not mix the source of those constraints. The BEL/LOC constraints should be entirely specified either through RTL or through XDC, but not a combination of both. Following is an example of BEL constraints specified at the RTL.

Verilog file:

```
(*BEL="H6LUT",RLOC="X0Y0"*) LUT6 S0_LUTH (...);
(*BEL="G6LUT",RLOC="X0Y0"*) LUT6 S0_LUTG (...);
(*BEL="F6LUT",RLOC="X0Y0"*) LUT4 S0_LUTF (...);
(*BEL="E5LUT",RLOC="X0Y0"*) LUT4 S0_LUTE (...);
(*BEL="D6LUT",RLOC="X0Y0"*) LUT6 S0_LUTD (...);
(*BEL="C6LUT",RLOC="X0Y0"*) LUT6 S0_LUTC (...);
(*BEL="B6LUT",RLOC="X0Y0"*) LUT4 S0_LUTB (...);
(*BEL="A5LUT",RLOC="X0Y0"*) LUT4 S0_LUTA (...);

(*BEL="CARRY8",RLOC="X0Y0"*) CARRY8#(.CARRY_TYPE("DUAL_CY4")) S0_CARRY8(...);

(*BEL="HFF2",RLOC="X0Y0"*) FD FD_out5 (...);
(*BEL="GFF2",RLOC="X0Y0"*) FD FD_out4 (...);
(*BEL="FFF2",RLOC="X0Y0"*) FD FD_out3 (...);
(*BEL="DFF2",RLOC="X0Y0"*) FD FD_out2 (...);
(*BEL="CFF2",RLOC="X0Y0"*) FD FD_out1 (...);
(*BEL="BFF2",RLOC="X0Y0"*) FD FD_out0 (...);
```

Note: The INIT string has been omitted for simplification.

In the following example, the RPM is defined at the RTL but the BEL constraints are specified through XDC.

Verilog file:

```
(*RLOC="X0Y0"*) LUT6 S0_LUTH (...);
(*RLOC="X0Y0"*) LUT6 S0_LUTG (...);
(*RLOC="X0Y0"*) LUT4 S0_LUTF (...);
(*RLOC="X0Y0"*) LUT4 S0_LUTE (...);
(*RLOC="X0Y0"*) LUT6 S0_LUTD (...);
(*RLOC="X0Y0"*) LUT6 S0_LUTC (...);
(*RLOC="X0Y0"*) LUT4 S0_LUTB (...);
(*RLOC="X0Y0"*) LUT4 S0_LUTA (...);

(*RLOC="X0Y0"*) CARRY8#(.CARRY_TYPE("DUAL_CY4")) S0_CARRY8(...);

(*RLOC="X0Y0"*) FD FD_out5 (...);
(*RLOC="X0Y0"*) FD FD_out4 (...);
(*RLOC="X0Y0"*) FD FD_out3 (...);
(*RLOC="X0Y0"*) FD FD_out2 (...);
(*RLOC="X0Y0"*) FD FD_out1 (...);
(*RLOC="X0Y0"*) FD FD_out0 (...);
```

Note: The INIT string has been omitted for simplification

XDC file:

```
set_property BEL CARRY8 [get_cells S0_CARRY8]
set_property BEL HFF2 [get_cells FD_out5]
set_property BEL GFF2 [get_cells FD_out4]
set_property BEL FFF2 [get_cells FD_out3]
set_property BEL DFF2 [get_cells FD_out2]
set_property BEL CFF2 [get_cells FD_out1]
set_property BEL BFF2 [get_cells FD_out0]
set_property BEL A5LUT [get_cells S0_LUTA]
set_property BEL B6LUT [get_cells S0_LUTB]
set_property BEL C6LUT [get_cells S0_LUTC]
set_property BEL D6LUT [get_cells S0_LUTD]
set_property BEL E5LUT [get_cells S0_LUTE]
set_property BEL F6LUT [get_cells S0_LUTF]
set_property BEL G6LUT [get_cells S0_LUTG]
set_property BEL H6LUT [get_cells S0_LUTH]
```

The RPM_GRID system is used for heterogeneous RPMs in which cells in an RPM belong to different site types (such as a combination of slice, block RAM, and DSP). This is an absolute coordinate system that is mapped to a specific Xilinx device.

Because the cells can occupy sites of various sizes, the RPM_GRID system uses absolute RPM_GRID coordinates. The RPM_GRID values are visible in the Site Properties window of the Vivado Integrated Design Environment (IDE) when a specific site is selected. The coordinates can also be queried with Tcl commands using the RPM_X and RPM_Y site properties.

RPM_GRID Coordinates VHDL Example

The following VHDL example defines RLOC constraints using RPM_GRID coordinates.

- Two shift registers are placed relative to a block RAM.
- Four stages connect the input.
- Four stages connect the output.

```
attribute RLOC : string;
attribute RPM_GRID : string;
attribute RLOC of di_reg3 : label is "X25Y0";
attribute RLOC of di_reg2 : label is "X27Y0";
attribute RLOC of di_reg1: label is "X29Y0";
attribute RLOC of di_reg0 : label is "X31Y0";
attribute RLOC of ram0 : label is "X34Y0";
attribute RLOC of out_reg3 : label is "X37Y0";
attribute RLOC of out_reg2 : label is "X39Y0";
attribute RLOC of out_reg1 : label is "X41Y0";
attribute RLOC of out_reg0 : label is "X43Y0";
```

Setting a Property to Invoke the RPM_GRID System

To use the RPM_GRID system, set a property on any cell in the RPM set:

```
attribute RPM_GRID of ram0 : label is "GRID";
```

As long as at least one cell has the RPM_GRID property equal to GRID, the RPM_GRID coordinate system is used.

Although the RPM_GRID coordinates are absolute based on the target device, they define the relative placement of the elements of an RPM set.

During implementation, the RPM set can be placed at any suitable location on the device.

RPM_GRID Coordinate Values

The RPM_GRID coordinate values differ significantly from the coordinate values of the SLICEs on the FPGA. These coordinates:

- Are stored as RPM_X and RPM_Y properties on device sites in the Vivado tools.
- Can be queried using get_property.

The following example does the following:

- Gets the RPM coordinates from a selected SLICE.
- Uses join to output both the X and Y coordinates in the required format.

```
join "X[get_property RPM_X [get_selected_objects]]Y[get_property RPM_Y
[get_selected_objects]]"
X25Y394
```

Defining RLOC Properties Directly in the RTL Source File

Because the standard grid is simple and relative, you can define the RLOC properties for an RPM directly in the RTL source file.

Because the RPM_GRID coordinates must be extracted from the target device, you will probably need to:

- Iterate on the design to find the right RPM_GRID values after synthesis.
- Add the coordinates as properties in the RTL source files.
- Resynthesize the netlist before placement.

Optionally use an RLOC_ORIGIN or LOC constraint to place and fix the location of an RPM on the device. In the Vivado IDE, these properties fix the RPM origin, or the lower-left corner of the RPM. Each remaining cell in the RPM set is placed by using the relative location (RLOC) to offset from the origin.

Figure 9-2:

The following example shows a hierarchical RPM that is fixed using RLOC_ORIGIN. RLOC constraints are assigned to the RPM register cells to create a two-up-by-three-across placement pattern.

In Verilog:

```
(* RLOC = "X0Y0" *) FDC sr0...
(* RLOC = "X1Y0" *) FDC sr1...
(* RLOC = "X2Y0" *) FDC sr2...
(* RLOC = "X0Y1" *) FDC sr3...
(* RLOC = "X1Y1" *) FDC sr4...
(* RLOC = "X2Y1" *) FDC sr5...
```

The RPM is instantiated into the design three times with an RLOC on each cell:

```
(* RLOC = "X0Y0" *) ffs u0...
(* RLOC = "X3Y2" *) ffs u1...
(* RLOC = "X6Y4" *) ffs u2...
```

Finally, an RLOC_ORIGIN of X74Y15 is assigned to cell u0 resulting in the placement shown in Figure 9-2. The highlighting in the figure is shown in Table 9-1.

Table 9-1:

u0	yellow
u1	green
u2	red

TIP: *Although RPMs control the relative placement of logic elements, they do not insure that specific routing resources are used to connect the logic from one implementation to the next.*

For more information on controlling the routing used, see Routing Constraints, page 158.

XDC macros enable assignment of relative placement to cells after synthesis. Macros have many characteristics similar to RPMs, but are design objects that can be modified interactively using XDC and Tcl. Macros are created from leaf cells that are grouped together with relative placement constraints.

While RPMs are managed in HDL code, macros are managed using XDC constraints. RPMs cannot be automatically converted to macros. Similarly, macros cannot be automatically annotated to HDL code. Unlike macros, RPMs are not objects, and the XDC macro commands cannot be used on RPMs.

Table 9-2:

	HDL Attributes	XDC constraints
	Read-only	Read-write
	Yes (H_SET/HU_SET)	No
	Non-leaf and leaf cells	Leaf cells only
	Yes, using RPM_GRID attribute	Yes, using update_macro -absolute_grid
	No	Yes
	In netlist	In XDC or Tcl scripts

Use the following XDC Tcl commands to specify macros:

- create_macro
- update_macro
- delete_macros
- get_macros

Each command is supported by `undo` and `redo`.

Following are descriptions of each command.

create_macro

The `create_macro` command creates a new macro object.

Macro names must be unique. Attempting to create a macro with the same name as an existing macro generates an error.

```
create_macro <name>
```

```
create_macro m0
```

Creates a macro object called `m0`.

TIP: *To ensure optimal LUT-FF alignment, specify the BEL location when creating your macro. The BEL location must be set separately as a property on the cell objects. For example:* `set_property BEL AFF [get_cell u2/sr0].`

The `update_macro` command adds leaf cells and relative placements (`RLOC`s) to the macro.

The `RLOC` has identical syntax and functionality as the RPM `RLOC` attribute. All cells must be specified at once. No partial or incremental definition is allowed.

```
update_macro  [-absolute_grid] <macro name>  <cell-RLOC list>
```

where

- `-absolute_grid`: A switch to choose the Absolute Grid for mixing slice and non-slice sites.
  - The X-Y values are the site properties `RPM_X` and `RPM_Y`.
  - The Absolute Grid values are identical to those of `RPM_GRID`.
- `macro name`: The name of the macro to be updated.
- `cell-RLOC list`: A Tcl list of cells and `RLOC` pairs:

  `{cell0 RLOC(cell0) cell1 RLOC(cell1) - cellN RLOC(cellN)}.`
  - All macro cells and RLOCs must be specified at once. It is not possible to build a macro in steps.
  - If you need to update an existing macro, you must re-create it first.

```
update_macro m1 {u2/sr0 X0Y0 u2/sr1 X0Y1}
```

  - Adds `u2/sr0` and `u2/sr1` to macro `m1`
  - Assigns `u2/sr0` an RLOC of `X0Y0`
  - Assigns `u2/sr1` an RLOC of `X0Y1`

The following (`update_macro` Example Two) does the same, with slightly different syntax.

```
set rlocs [list u2/sr0 X0Y0 u2/sr1 X0Y1]
update_macro m1 $rlocs
```

This example uses the absolute grid:

```
set rlocs {ireg X2Y38 q1reg X17Y40 q2reg X17Y40}
update_macro -absolute_grid m2 $rlocs
```

delete_macros

The `delete_macros` command deletes the specified macros.

```
delete_macros <pattern>
```

```
delete_macros m1
```

get_macros

The `get_macros` command returns macro objects in a design.

```
get_macros [pattern]
```

With no arguments, the `get_macros` command returns all macros in the design. When macro names are specified, the command returns the corresponding macro objects.

The `get_macros` command can be used with other object commands. Examples:

```
% create_macro m1
% update_macro m1 {u2/sr0 X0Y0 u2/sr1 X0Y1}
% get_cells -of [get_macros m1]
u2/sr0 u2/sr1
% get_macros -of [get_cells u2]
m1
```

The following command returns all macros that are fully contained within the cells.

```
get_macros -of [get_cells $cells]
```

Using `get_cells`, other indirect combinations are possible such as:

```
get_macros -of [get_cells -of [get_pblocks pb0]]
```

This command returns the macros contained within Pblock `pb0`.

Macros are stored as XDC constraints. By definition, they are Tcl commands. This allows the macros to be used in both XDC constraint files and Tcl scripts, and used interactively.

Macros are written using the `write_xdc` command. Macros are read using the `read_xdc` command. The `-cell` option can be used to limit scope to particular cells.

The `-cell` option is particularly useful for applying a relative placement from one macro to similar instances in different hierarchies.

Managing Macros Example One

Write all XDC constraints in memory, including macros:

```
% write_xdc constrs.xdc
```

Managing Macros Example Two

A design contains three instances of a cell:

```
inst_0, inst_1, and inst_2.
```

A macro is created inside `inst_0`:

```
% create_macro m0
% update_macro m0 {reg0 X0Y0 reg1 X0Y1}
% write_xdc -cell inst_0 inst_0.xdc
```

Managing Macros Example Three

Write all XDC constraints including macro `m0`, for the cell `inst_0`:

```
% write_xdc -cell inst_0.xdc inst_0.xdc
```

Managing Macros Example Four

Read the XDC constraints including the macro `m0` from cell `inst_0`, and apply it to `inst_1` and `inst_2`:

```
% read_xdc inst_0 -cell {inst_1 inst_2}
% get_macros
m0 inst_1_m0 inst_2_m0
```

TIP: *When a macro is read and applied to another cell using the `-cell` option, the new macro name must be unique. The cell name is applied as a prefix to the macro name to create a unique macro name. In Example Four, two new unique macros were created. They are inst_1_m0 and inst_2_m0.*

Macro objects have the following properties:

- `ABSOLUTE_GRID`
- `CLASS`
- `NAME`
- `RLOCS`

Macro Properties Example

```
% report_property [get_macros m1]
Property       Type     Read-only  Visible  Value
ABSOLUTE_GRID  bool     true       true     0
CLASS          string   true       true     macro
NAME           string   true       true     m1
RLOCS          string*  true       true     u2/sr0 X0Y0 u2/sr1 X0Y1
```

Following are descriptions of the properties.

Boolean property that reflects whether or not the RLOCs are using the default grid system or the Absolute Grid system.

The default is false. If update_macro is used with -absolute_grid, then the property is true.

The Absolute Grid uses coordinates that align with site RPM_X and RPM_Y properties to allow creating macros from cells placed at different site types.

Identifies the object as a macro.

Name of the macro object, either the name used by create_macro, or the macro name prefixed by the cell hierarchy when using read_xdc -cell.

String containing the list of macro cells and their RLOC properties in the same format used by the update_macro command.

Macro cells have these additional properties:

- RLOC: The relative location property (RLOC) value of the cell.
- MACRO_NAME: The name of the macro to which the cell belongs.

Using the previous example for macro properties:

```
% get_property RLOC [get_cells {u2/sr0 u2/sr1}]
X0Y0 X0Y1
% get_property MACRO_NAME [get_cells {u2/sr0 u2/sr1}]
m1 m1
```

opt_design is free to optimize and remove LUTs that belong to an XDC macro despite the RLOC constraint. To prevent opt_design from optimizing the logic inside an XDC macro, it is necessary to set the property DONT_TOUCH to TRUE on all the cells that belong to the XDC macro. The DONT_TOUCH property can be set either through RTL or XDC.

This section gives the following advanced XDC macro examples:

- Relative Grid Macro Examples
- Absolute Grid Macro Examples

Relative Grid Macro Examples

By default, the relative grid is used for macro RLOC coordinates because the most common macros are made of cells that belong to the same site type.

The following simple example illustrates the relative placement derived from macro RLOCs. The macro consists of a pair of SRL >FF >FF circuits that are to be arranged in a 2x2 pattern. See Figure 9-3.

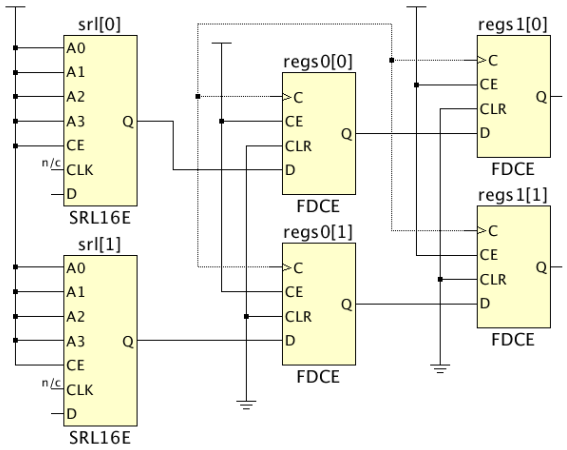


Figure 9-3:

To create the desired relative placement, the cells are assigned RLOCs as follows:

```
srl[0] X0Y0
regs0[0] X0Y0
regs1[0] X1Y0
srl[1] X0Y1
regs0[1] X0Y1
regs1[1] X1Y1
```

The following commands create this macro with a name `m0`:

```
create_macro m0
update_macro m0 {srl[0] X0Y0 regs0[0] X0Y0 regs1[0] X1Y0 srl[1] X0Y1 regs0[1] X0Y1
regs1[1] X1Y1}
```

The macro can be automatically placed by the placer or manually placed as a set. The macro placement appears as shown in Figure 9-4.

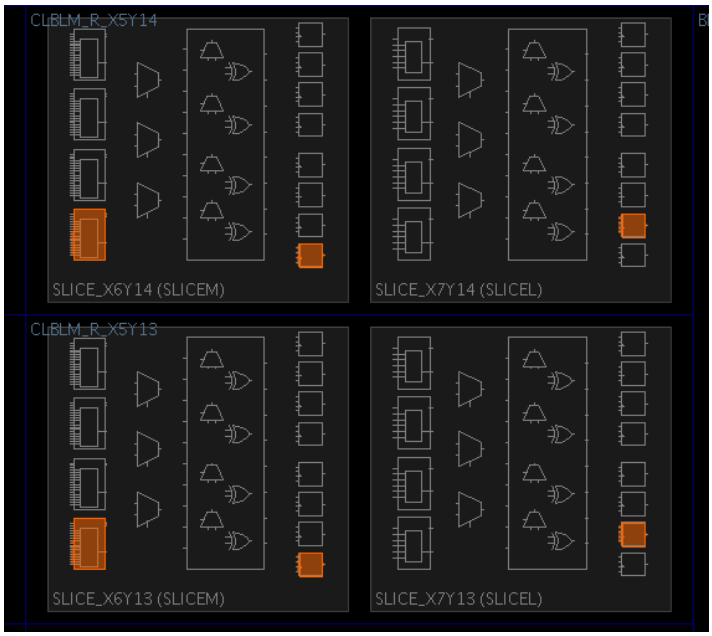


Figure 9-4:

The macro contains SRLs which are based on LUTRAMs, and which can be placed only in SLICEM type slices. This places slight restrictions on the possible locations of the macro. The macro can be located only where a SLICEL column is to the right of a SLICEM column.

CAUTION! *Too many densely packed slices in proximity can cause congestion, which reduces routability and can negatively impact performance.*

Absolute Grid Macro Examples

When combining cells of different site types into a macro, you must use the absolute grid.

The absolute grid (also known as the RPM grid) is an absolute coordinate system that defines the coordinates of a site based on its location within the device. The absolute grid also considers the sizes of sites. RAM and DSP blocks have wider spacing than slices. The absolute grid is illustrated in Figure 9-5.

In this example, there are cells from three different types to group into a macro using the absolute grid. The example consists of an input data path from input ports, through two stages of registers, then block RAMs. This is illustrated in the schematic in Figure 9-5.

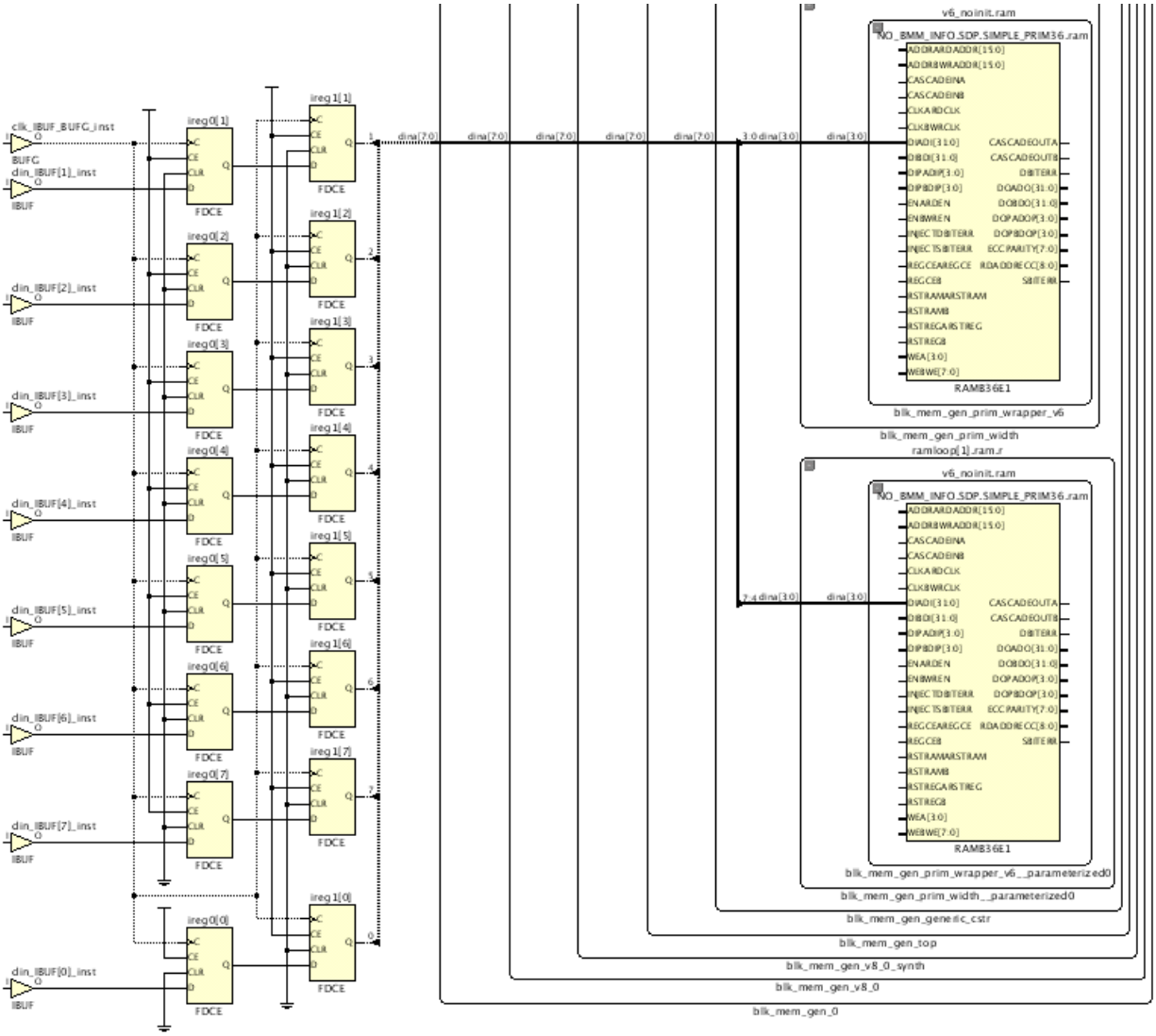


Figure 9-5:

The macro creation requires a list of cells and their relative locations (RLOCs) using the absolute grid. When creating the macro, it might be difficult to visualize the relative placement of absolute grid macros.

RECOMMENDED: *Place the cells temporarily into absolute locations in the device, then derive the absolute grid RLOC values of each cell.*

The cells are first manually placed and arranged in their desired locations as shown in Figure 9-6.

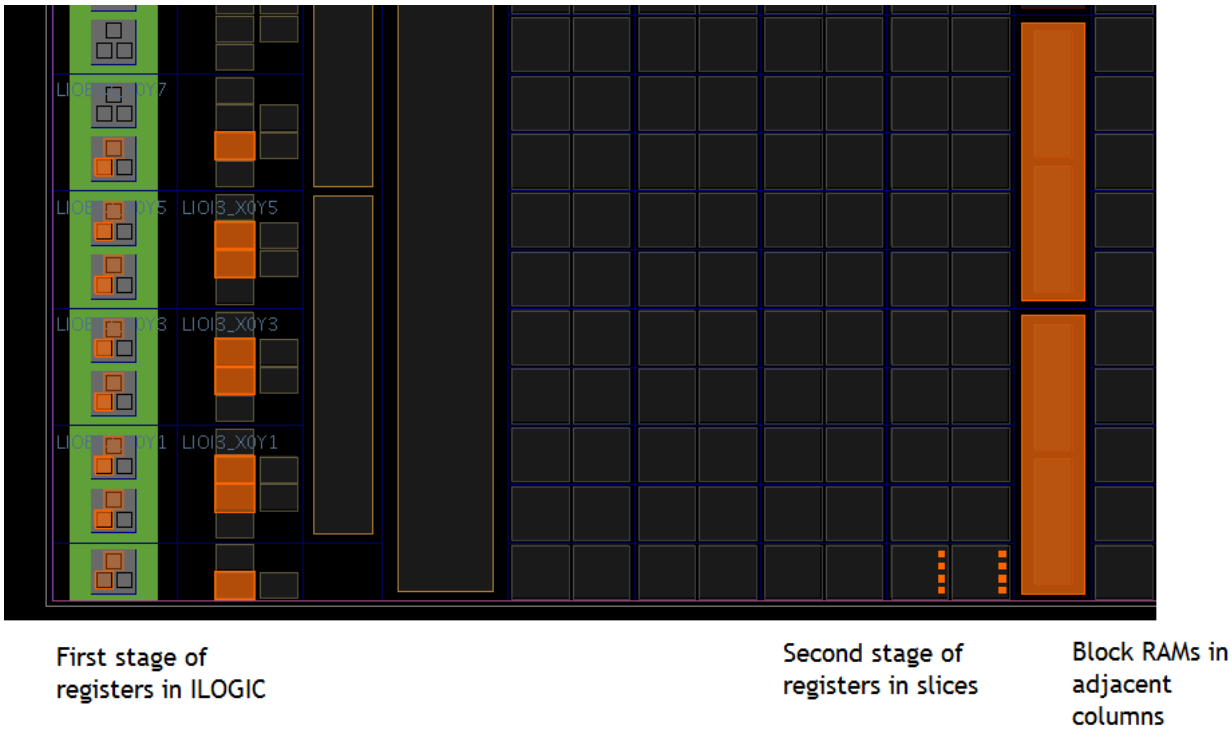


Figure 9-6:

Although the absolute grid specifies absolute locations, the resulting macro can be placed at any location within the device that can accommodate the relative placement of the macro. In this example, the relative locations are specified using the lower-left hand corner as the point of reference.

However, the absolute grid locations specify only relative placement, not absolute placement. That allows the macro to be located anywhere in the device that maintains the relative placement.

Because the example is somewhat complex, consisting of ILOGIC, slices, and block RAM, the macro locations are somewhat restricted but can be placed at any of the three locations highlighted in orange in Figure 9-7.

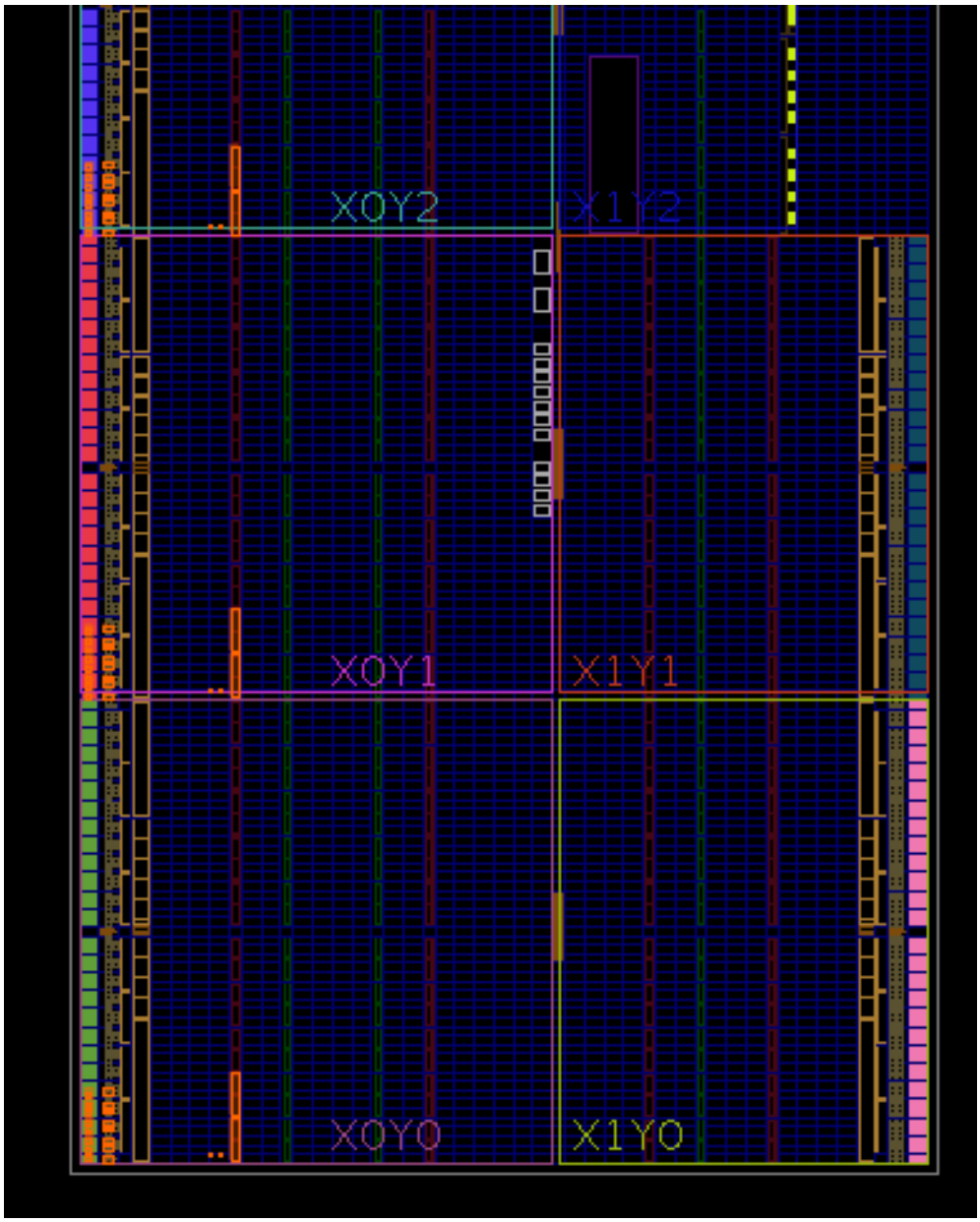


Figure 9-7:

To determine absolute grid `RLOC`s, use the site `RPM_X` and `RPM_Y` properties. For example, the lower block RAM is placed at site RAMB36_X0Y0.

Selecting the site (not the cell) displays the following values of 33 for RPM_X and 0 for RPM_Y (Figure 9-8). These are the absolute grid coordinates. The corresponding RLOC value is X33Y0.

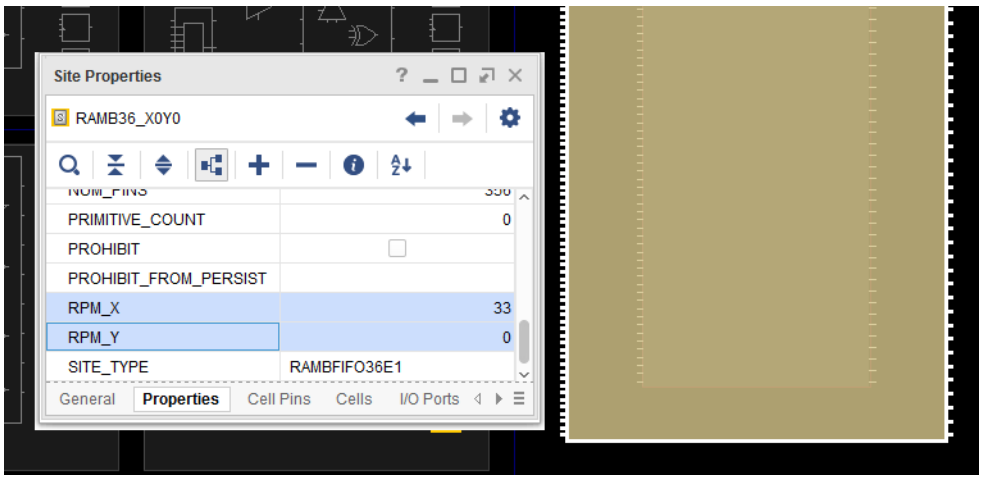


Figure 9-8:

The same method is applied to determine the absolute RLOC of a slice (Figure 9-9). The cells within this slice have an RLOC of X31Y0.

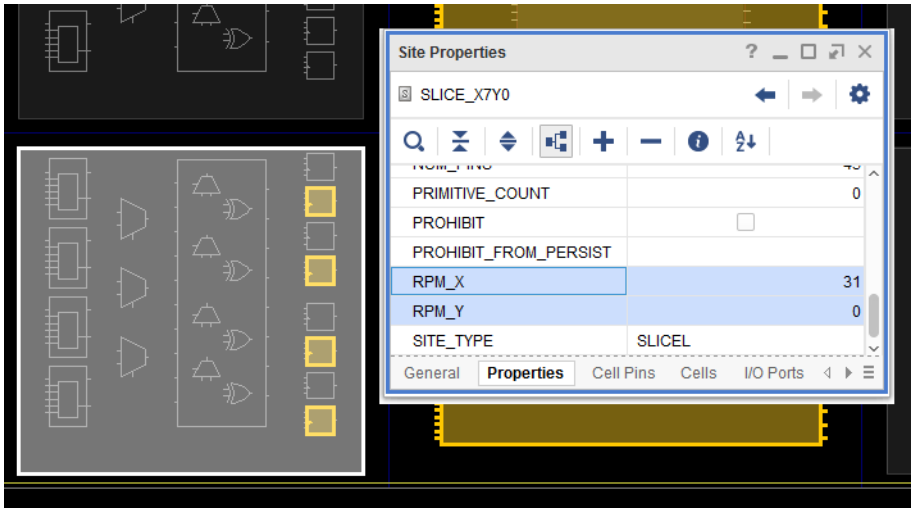


Figure 9-9:

There are two commands used to create the macro, with a name m0:

```
create_macro m0
update_macro m0 -absolute_grid <cell0 rloc0 cell1 rloc1 cell2 rloc2 …  cellN rlocN>
```

If the macro contains many cells as it does in this example, Tcl can be used to simply building and specifying the `cell-rloc` list required by `update_macro`. Given a placed cell, the absolute grid RLOC can be determined using the following Tcl `proc getAbsRLOC`:

```
proc getAbsRLOC {cell} {
  set site [get_sites -of [get_cells $cell]]
  set X [get_property RPM_X $site]
  set Y [get_property RPM_Y $site]
  return "X${X}Y${Y}"
}
```

```
% set rloc [getAbsRLOC $ram0]
X33Y0
```

The Tcl `dict` command can be used to build a dictionary (associative array) of cells and absolute grid RLOCs for the `update_macro` command. A Tcl associative array is a series of key-value pairs. The cells and RLOCs can be arranged as such as series using the `dict` command. The array keys are the macro cell objects. The array values are the cell RLOCs. This helps to automate the process of creating macros with many cells. The following example uses the absolute grid, but the method can be applied to the normal grid as well.

Assuming `$cells` is the list of macro cells, and each cell of `$cells` has been placed to form the desired macro pattern, the following Tcl proc creates a list of cell-RLOC pairs for the `update_macro` command.

```
proc buildRLOCList {cells} {
  set rlocs [dict create] ; # initialize dictionary called rlocs
  foreach cell $cells {
    # dictionary key is cell, value is absolute RLOC
    dict set rlocs $cell [getAbsRLOC $cell]
  }
  return $rlocs
}
```

```
# create macro cell list: input register stage and BRAM cells
set cells [get_cells -hier  [list ireg0* ireg1* *SIMPLE_PRIM36.ram]]
create_macro m0
update_macro m0 -absolute_grid [buildRLOCList $cells]
```

To see the dictionary list created by buildRLOCList:

```
$ puts [buildRLOCList $cells]
{ireg0[6]} X2Y10 {ireg0[5]} X2Y11 {ireg0[4]} X2Y6 {ireg0[3]} X2Y7 . . .
```

If there are many macro cells and macro cells buried in hierarchy, specifying the explicit list of cell-`RLOC` pairs can become complicated and error prone. The creation and management of XDC macros can be made simpler using Tcl.

It is recommended to convert RPMs to XDC macros wherever feasible because XDC macros are the preferred method of implementing relative placement constraints. This process can be done manually by removing the RPM attributes from the HDL sources and creating equivalent XDC macros. Conversion can also be done somewhat automatically by using Tcl to replace RPM attributes with XDC macro constraints.

The automated process consists of the following steps:

1. In all HDL sources, replace each RPM attribute with a similarly named string, for example:
   - Replace hu_set with m_hu_set
   - Replace u_set with m_u_set
   - Replace rloc with m_rloc

   This ensures that the RPMs are not processed however the inactive attributes are passed through to the synthesized netlist as cell properties.
2. Open the synthesized design or run link_design and create XDC macros based on the inactive properties. For example, each HU_SET will have a cell property called m_hu_set that can be used to create the equivalent XDC macro. Each cell within the original HU_SET will have a property m_rloc that can be converted to an RLOC.
3. Save the constraints which now include the XDC macros definitions.

The conversion is best accomplished using Tcl by building XDC macros cell lists based on their unique m_hu_set or m_uset values. Following is a simple VHDL conversion example.

The original VHDL source includes a HU_SET RPM called `set0` with two cells, one with RLOC `X0Y0` and the other with RLOC `X0Y1`.

```
signal r0 : std_logic;
signal r1 : std_logic;

attribute hu_set : string;
attribute rloc : string;

attribute hu_set of r0 : signal is "set0";
attribute hu_set of r1 : signal is "set0";

attribute rloc of r0 : signal is "X0Y0";
attribute rloc of r1 : signal is "X0Y1";
```

Next the VHDL source is modified to replace hu_set and RLOC with similarly named but inactive attributes:

```
signal r0 : std_logic;
signal r1 : std_logic;

attribute m_hu_set : string;
attribute m_rloc : string;

attribute m_hu_set of r0 : signal is "set0";
attribute m_hu_set of r1 : signal is "set0";

attribute m_rloc of r0 : signal is "X0Y0";
attribute m_rloc of r1 : signal is "X0Y1";
```

After synthesis, the cells can be filtered based on these similarly named properties:

```
Vivado% get_cells -filter {m_hu_set == "set0"}
r0_reg r1_reg

Vivado% get_property m_rloc [get_cells {r0_reg r1_reg}]
X0Y0 X0Y1
```

This provides the necessary information to create an XDC macro to replace the RPM:

```
Vivado% create_macro set0
Vivado% update_macro set0 {r0_reg X0Y0 r1_reg X0Y1}
```

These two XDC constraints can be saved as part of the design constraints. Large amounts of RPM conversions are better handled using a Tcl script. Following is an example script to convert HU_SET RPMs to XDC macros.

```
# create a sorted list of all unique RPMs according to m_hu_set values
set RPMs [lsort -uniq [get_property m_hu_set [get_cells -hier -filter
{primitive_level != INTERNAL}]]]

# remove the first element which is empty (no m_hu_set property)
set RPMs [lrange $RPMs 1 end]

# iterate over list of RPMs, convert each to an XDC macro
# get each RPM cell of the RPM with its RLOC
# build a list for the update_macro command
foreach rpm $RPMs {
  create_macro $rpm
  set cells [get_cells -hier -filter "m_hu_set == $rpm"]
  set rlocs [list]
  foreach cell $cells {
    lappend rlocs $cell
    lappend rlocs [get_property m_rloc $cell]
  }
  update_macro $rpm $rlocs
  puts "created XDC macro $rpm, cell list: $rlocs"
}
```

XILINX

Appendix A

Supported XDC and SDC Commands

This Appendix discusses supported Xilinx® Design Constraints (XDC) and Synopsys Design Constraints (SDC) commands in the Xilinx Vivado® Integrated Design Environment (IDE).

Table A-1:

create_clock create_generated_clock group_path set_clock_groups set_clock_latency set_data_check set_disable_timing set_false_path set_input_delay set_output_delay set_max_delay set_min_delay set_multicycle_path set_case_analysis set_clock_sense set_clock_uncertainty set_input_jitter set_max_time_borrow set_propagated_clock set_system_jitter set_external_delay set_bus_skew	add_cells_to_pblock create_pblock delete_pblock remove_cells_from_pblock resize_pblock create_macro delete_macros update_macro set_package_pin_val create_debug_core create_debug_port connect_debug_port set_power_opt set_switching_activity reset_switching_activity set_operating_conditions reset_operating_conditions add_to_power_rail create_power_rail delete_power_rails get_power_rails remove_from_power_rail create_waiver	set expr list filter current_instance get_hierarchy_separator set_hierarchy_separator get_property set_property set_units endgroup startgroup create_property current_design set_load set_logic_dc set_logic_one set_logic_zero set_logic_unconnected make_diff_pair_ports

Table A-1: ***(Cont'd)***

get_iobanks get_package_pins get_sites get_bel_pins get_bels get_nodes get_pips get_site_pins get_site_pips get_slrs get_tiles get_wires get_pkgpin_bytegroups get_pkgpin_nibbles	all_clocks get_path_groups get_clocks get_generated_clocks get_timing_arcs get_speed_models get_pblocks get_macros	all_cpus all_dsps all_fanin all_fanout all_hsios all_inputs all_outputs all_rams all_registers all_ffs all_latches get_cells get_nets get_pins get_ports get_debug_cores get_debug_ports

Note: Because all Xilinx Tcl commands support the `-quiet` and `-verbose` options, the following table does not list them.

Table A-2:

current_instance [*instance_name*]	current_instance [*instance_name*]	The Vivado IDE handles `get_ports` differently when using `read_xdc -cells/-ref` or the `SCOPED_TO_xxx` constraint file property.
expr	expr	
list	list	In the Vivado IDE, a Tcl list is also used as an objects container.
set	set	
set_hierarchy_separator [*separator*]	set_hierarchy_separator [*separator*]	

Table A-2: ***(Cont'd)***

`set_units`
`[-capacitance` *`cap_units`*`]`
`[-resistance` *`res_unit`*`]`
`[-time` *`time_unit`*`]`
`[-voltage` *`voltage_units`*`]`
`[-current` *`current_unit`*`]`
`[-power` *`power_unit`*`]`
	`set_units`
	`[-capacitance` *`arg`*`]`
	`[-resistance` *`arg`*`]`
	`[-time` *`arg`*`]`
	`[-voltage` *`arg`*`]`
	`[-current` *`arg`*`]`
	`[-power` *`arg`*`]`
	`[-suffix` *`arg`*`]`
	`[-digits` *`arg`*`]`
		The `set_units -time` cannot change the timing unit in the Vivado IDE.

`all_clocks`	`all_clocks`	

`all_inputs`
`[-level_sensitive]`
`[-edge_triggered]`
`[-clock` *`clock_name`*`]`
	`all_inputs`	

`all_outputs`
`[-level_sensitive]`
`[-edge_triggered]`
`[-clock` *`clock_name`*`]`
	`all_outputs`	

`all_registers`
`[-no_hierarchy]`
`[-clock` *`clock_name`*`]`
`[-rise_clock` *`clock_name`*`]`
`[-fall_clock` *`clock_name`*`]`
`[-cells]`
`[-data_pins]`
`[-clock_pins]`
`[-slave_clock_pins]`
`[-async_pins]`
`[-output_pins]`
`[-level_sensitive]`
`[-edge_triggered]`
`[-master_slave]`
	`all_registers`
	`[-no_hierarchy]`
	`[-clock` *`args`*`]`
	`[-rise_clock` *`args`*`]`
	`[-fall_clock` *`args`*`]`
	`[-cells]`
	`[-data_pins]`
	`[-clock_pins]`
	`[-async_pins]`
	`[-output_pins]`
	`[-level_sensitive]`
	`[-edge_triggered]`	

`current_design`	`current_design`	In the Vivado IDE, the current design refers to the design loaded in memory, and cannot be changed to another module or entity than the top-level one.

Table A-2: ***(Cont'd)***

get_cells [-hierarchical] [-hsc *separator*] [-regexp] [-nocase] -of_objects *objects* *patterns*	get_cells [-hierarchical] [-hsc *arg*] [-regexp] [-nocase] [-of_objects *args*] [*patterns*] [-filter *arg*] [-match_style *arg*]	
get_clocks [-regexp] [-nocase] *patterns*	get_clocks [-regexp] [-nocase] [*patterns*] [-filter *arg*] [-of_objects *args*] [-match_style *arg*] [-include_generated_clocks]	The Vivado IDE supports the -of_objects option to query the clock object on the clock tree.
get_lib_cells [-hsc *separator*] [-regexp] [-nocase] *patterns*	get_lib_cells [-regexp] [-nocase] *patterns* [-filter *arg*] [-include_unsupported] [-of_objects *args*]	In the Vivado IDE, because only one device library can be loaded for a design, it is not necessary to specify the library name when querying the library cells.
get_lib_pins [-hsc *separator*] [-regexp] [-nocase] *patterns*	get_lib_pins [-regexp] [-nocase] *patterns* [-filter *arg*] [-of_objects *args*]	
get_libs [-regexp] [-nocase] *patterns*	get_libs [-regexp] [-nocase] [*patterns*] [-filter *arg*]	

Table A-2: ***(Cont'd)***

get_nets [-hierarchical] [-hsc *separator*] [-regexp] [-nocase] -of_objects *objects* *patterns*	get_nets [-hierarchical] [-hsc *arg*] [-regexp] [-nocase] [-of_objects *args*] [*patterns*] [-filter *arg*] [-match_style *arg*] [-top_net_of_hierarchical_group] [-segments] [-boundary_type *arg*]	
get_pins [-hierarchical] [-hsc *separator*] [-regexp] [-nocase] -of_objects *objects* *patterns*	get_pins [-hierarchical] [-hsc *arg*] [-regexp] [-nocase] [-of_objects *args*] [*patterns*] [-leaf] [-filter *arg*] [-match_style *arg*]	
get_ports [-regexp] [-nocase] *patterns*	get_ports [-regexp] [-nocase] [*patterns*] [-filter *arg*] [-of_objects *args*] [-match_style *arg*]	
create_clock -period *period_value* [-name *clock_name*] [-waveform *edge_list*] [-add] [*source_objects*]	create_clock -period *arg* [-name *arg*] [-waveform *args*] [-add] [*objects*]	

Table A-2: ***(Cont'd)***

create_generated_clock [-name *clock_name*] -source *master_pin* [-edges *edge_list*] [-divide_by *factor*] [-multiply_by *factor*] [-duty_cycle *percent*] [-invert] [-edge_shift *shift_list*] [-add] [-master_clock *clock*] [-combinational] *source_objects*	create_generated_clock [-name *arg*] [-source *args*] [-edges *args*] [-divide_by *arg*] [-multiply_by *arg*] [-duty_cycle *arg*] [-edge_shift *args*] [-add] [-master_clock *arg*] [-combinational] *objects*	
group_path [-name *group_name*] [-default] [-weight *weight_value*] [-from *from_list*] [-rise_from *from_list*] [-fall_from *from_list*] [-to *to_list*] [-rise_to *to_list*] [-fall_to *to_list*] [-through *through_list*] [-rise_through *through_list*] [-fall_through *through_list*]	group_path [-name *arg*] [-weight 1|2] [-from *args*] [-to *args*] [-through *args*]	
set_clock_groups [-name *name*] [-logically_exclusive] [-physically_exclusive] [-asynchronous] [-allow_paths] -group *clock_list*	set_clock_groups [-name *arg*] [-logically_exclusive] [-physically_exclusive] [-asynchronous] [-group *args*]	

Table A-2: ***(Cont'd)***

set_clock_latency [-rise] [-fall] [-min] [-max] [-source] [-late] [-early] [-clock *clock_list*] *delay* *object_list*	set_clock_latency [-rise] [-fall] [-min] [-max] [-source] [-late] [-early] [-clock *args*] *latency* *objects*	
set_clock_sense [-positive] [-negative] [-pulse *pulse*] [-stop_propagation] [-clock *clock_list*] *pin_list*	set_clock_sense [-positive] [-negative] [-pulse *arg*] [-stop_propagation] [-clocks *args*] *pins*	
set_clock_uncertainty [-from *from_clock*] [-rise_from *rise_from_clock*] [-fall_from *fall_from_clock*] [-to *to_clock*] [-rise_to *rise_to_clock*] [-fall_to *fall_to_clock*] [-rise] [-fall] [-setup] [-hold] *uncertainty* [*object_list*]	set_clock_uncertainty [-from *args*] [-rise_from *args*] [-fall_from *args*] [-to *args*] [-rise_to *args*] [-fall_to *args*] [-setup] [-hold] *uncertainty* [*objects*]	

Table A-2: ***(Cont'd)***

set_data_check [-from *from_object*] [-to *to_object*] [-rise_from *from_object*] [-fall_from *from_object*] [-rise_to *to_object*] [-fall_to *to_object*] [-setup] [-hold] [-clock *clock_object*] *value*	set_data_check [-from *args*] [-to *args*] [-rise_from *args*] [-fall_from *args*] [-rise_to *args*] *[-fall_to args]* [-setup] [-hold] [-clock *args*] *value*	
set_disable_timing [-from *from_pin_name*] [-to *to_pin_name*] *cell_pin_list*	set_disable_timing [-from *arg*] [-to *arg*] *objects*	
set_false_path [-setup] [-hold] [-rise] [-fall] [-from *from_list*] [-to *to_list*] [-through *through_list*] [-rise_from *rise_from_list*] [-rise_to *rise_to_list*] [-rise_through *rise_through_list*] [-fall_from *fall_from_list*] [-fall_to *fall_to_list*] [-fall_through *fall_through_list*]	set_false_path [-setup] [-hold] [-rise] [-fall] [-from *args*] [-to *args*] [-through *args*] [-rise_from *args*] [-rise_to *args*] [-rise_through *args*] [-fall_from *args*] [-fall_to *args*] [-fall_through *args*] [-reset_path]	

Table A-2: ***(Cont'd)***

set_input_delay [-clock *clock_name*] [-clock_fall] [-level_sensitive] [-rise] [-fall] [-max] [-min] [-add_delay] [-network_latency_included] [-source_latency_included] *delay_value* *port_pin_list*	set_input_delay [-clock *args*] [-clock_fall] [-rise] [-fall] [-max] [-min] [-add_delay] [-network_latency_included] [-source_latency_included] *delay* *objects* [-reference_pin *args*]	In the Vivado IDE, input delays are not supported on internal pins.
set_max_delay [-rise] [-fall] [-from *from_list*] [-to *to_list*] [-through *through_list*] [-rise_from *rise_from_list*] [-rise_to *rise_to_list*] [-rise_through *rise_through_list*] [-fall_from *fall_from_list*] [-fall_to *fall_to_list*] [-fall_through *fall_through_list*] *delay_value*	set_max_delay [-rise] [-fall] [-from *args*] [-to *args*] [-through *args*] [-rise_from *args*] [-rise_to *args*] [-rise_through *args*] [-fall_from *args*] [-fall_to *args*] [-fall_through *args*] *delay* [-reset_path] [-datapath_only]	
set_max_time_borrow *delay_value* *object_list*	set_max_time_borrow *delay* *objects*	

Table A-2: ***(Cont'd)***

set_min_delay [-rise] [-fall] [-from *from_list*] [-to *to_list*] [-through *through_list*] [-rise_from *rise_from_list*] [-rise_to *rise_to_list*] [-rise_through *rise_through_list*] [-fall_from *fall_from_list*] [-fall_to *fall_to_list*] [-fall_through *fall_through_list*] *delay_value*	set_min_delay [-rise] [-fall] [-from *args*] [-to *args*] [-through *args*] [-rise_from *args*] [-rise_to *args*] [-rise_through *args*] [-fall_to *args*] [-fall_from *args*] [-fall_through *args*] *delay* [-reset_path]	
set_multicycle_path [-setup] [-hold] [-rise] [-fall] [-start] [-end] [-from *from_list*] [-to *to_list*] [-through *through_list*] [-rise_from *rise_from_list*] [-rise_to *rise_to_list*] [-rise_through *rise_through_list*] [-fall_from *fall_from_list*] [-fall_to *fall_to_list*] [-fall_through *fall_through_list*] *path_multiplier*	set_multicycle_path [-setup] [-hold] [-rise] [-fall] [-start] [-end] [-from *args*] [-to *args*] [-through *args*] [-rise_from *args*] [-rise_to *args*] [-rise_through *args*] [-fall_from *args*] [-fall_to *args*] [-fall_through *args*] *path_multiplier* [-reset_path]	

Table A-2: ***(Cont'd)***

set_output_delay [-clock *clock_name*] [-clock_fall] [-level_sensitive] [-rise] [-fall] [-max] [-min] [-add_delay] [-network_latency_included] [-source_latency_included] *delay_value* *port_pin_list*	set_output_delay [-clock *args*] [-clock_fall] [-rise] [-fall] [-max] [-min] [-add_delay] [-network_latency_included] [-source_latency_included] *delay* *objects* [-reference_pin *args*]	In the Vivado IDE, output delays are not supported on internal pins.
set_propagated_clock *object_list*	set_propagated_clock *object*	In the Vivado IDE, all clocks are propagated clocks by default.
set_case_analysis *value* *port_or_pin_list*	set_case_analysis *value* *objects*	
set_load [-min] [-max] [-subtract_pin_load] [-pin_load] [-wire_load] *value* *objects*	set_load [-max] [-min] *capacitance* *objects* [-rise] [-fall]	In the Vivado IDE, the set_load command is relevant for power analysis only.
set_logic_dc *port_list*	set_logic_dc *objects*	
set_logic_one *port_list*	set_logic_one *objects*	

Table A-2: ***(Cont'd)***

`set_logic_zero` `  port_list`	`set_logic_zero` `  objects`	

`set_operating_conditions`
`  [-library lib_name]`
`  [-analysis_type analysis_type]`
`  [-max max_condition]`
`  [-min min_condition]`
`  [-max_library max_lib]`
`  [-min_library min_lib]`
`  [-object_list objects]`
`  [condition]`
	`set_operating_conditions`
	`  [-voltage args]`
	`  [-grade arg]`
	`  [-process arg]`
	`  [-junction_temp arg]`
	`  [-ambient_temp arg]`
	`  [-thetaja arg]`
	`  [-thetasa arg]`
	`  [-airflow arg]`
	`  [-heatsink arg]`
	`  [-thetajb arg]`
	`  [-board arg]`
	`  [-board_temp arg]`
	`  [-board_layers arg]`
		In the Vivado IDE, the `set_operating_conditions` command: (1) sets the operating conditions for power analysis only; and (2) does not influence the timing reports. The Vivado IDE timing engine is controlled by the `config_timing_analysis` command. For more information on `config_timing_analysis` see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 10].

The following SDC commands are not supported.

- `set_clock_gating_check`
- `set_clock_transition`
- `set_ideal_latency`
- `set_ideal_network`
- `set_ideal_transition`
- `set_max_fanout`

  Note: Maximum fanout is controlled by the `MAX_FANOUT` attribute during synthesis.
- `set_drive`
- `set_driving_cell`
- `set_fanout_load`
- `set_input_transition`
- `set_max_area`
- `set_max_capacitance`
- `set_max_transition`
- `set_min_capacitance`
- `set_port_fanout_number`
- `set_resistance`
- `set_timing_derate`
- `set_voltage`
- `set_wire_load_min_block_size`
- `set_wire_load_mode`
- `set_wire_load_model`
- `set_wire_load_selection_group`
- `create_voltage_area`
- `set_level_shifter_strategy`
- `set_level_shifter_threshold`
- `set_max_dynamic_power`
- `set_max_leakage_power`

XILINX

Appendix B

Additional Resources and Legal Notices

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

The following Vivado® Design Suite guides are referenced in this document.

1. *ISE to Vivado Design Suite Migration Methodology Guide* (UG911)
2. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
3. *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899)
4. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
5. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
6. *AXI Quad SPI v3.2 LogiCORE IP Product Guide* (PG153)
7. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
8. *Vivado Design Suite User Guide: Synthesis* (UG901)
9. *Vivado Design Suite User Guide: Implementation* (UG904)
10. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
11. *Vivado Design Suite Properties Reference Guide* (UG912)
12. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
13. *7 Series FPGAs SelectIO Resources User Guide* (UG471)

The following additional resources are referenced in this document:

14. Xilinx Answer Record 59893

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Designing FPGAs Using the Vivado Design Suite 1 Training Course
2. Designing FPGAs Using the Vivado Design Suite 2 Training Course
3. Designing FPGAs Using the Vivado Design Suite 3 Training Course

4. Designing FPGAs Using the Vivado Design Suite 4 Training Course
5. Vivado Design Suite QuickTake Video Tutorials
6. Vivado Design Suite QuickTake Video: Using the Vivado Timing Constraint Wizard
7. Vivado Design Suite QuickTake Video: Advanced Clock Constraints and Analysis
8. Vivado Design Suite QuickTake Video: Setting Input Delay
9. Vivado Design Suite QuickTake Video: Setting Output Delay
10. Vivado Design Suite QuickTake Video: Migrating UCF Constraints to XDC