Vivado Design Suite User Guide

Using Constraints

UG903 (v2012.2) September 4, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/25/2012	2012.2	Initial Xilinx release.
09/04/2012	2012.2	Minor updates, including: • Minor grammatical edits • Updated figures • Edits to Setup Path Requirement Example, page 33 • Edits to Hold Path Requirement Example, page 35

Chapter 1: Introduction

Chapter 2: Constraints Methodology

Chapter 3: Timing Analysis

Chapter 4: Defining Clocks

Chapter 5: Timing Exceptions

Chapter 6: XDC Precedence

Chapter 7: Physical Constraints

Appendix A: Additional Resources

Introduction

The Vivado™ Integrated Design Environment (IDE) uses Xilinx® Design Constraints (XDC).

Migrating From UCF Constraints to XDC Constraints

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. 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 *Vivado Design Suite Methodology Guide (UG911)*. 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 conversion utility from UCF to XDC is not a replacement for properly understanding and creating XDC constraints. Each XDC constraint is described in this User Guide.*

About XDC Constraints

XDC constraints are a combination of:

- Industry standard Synopsys Design Constraints (SDC), 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.

Entering XDC Constraints

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

- Store the constraints in one or more files that can be added to a project constraints set.
- Read the files in using the **`read_xdc`** command.
- Type the constraints directly in the Tcl console once a design has been loaded in memory.

  This is particularly powerful for entering, validating, and debugging new constraints individually and interactively.

Constraints Methodology

This chapter discusses the recommended constraints entry flow.

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 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.

Organizing Your Constraints

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 IPs or large blocks developed by different teams.

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.*

Figure 2-1, Single or Multi XDC, shows two constraint sets in a project:

- The first constraint set includes two XDC files.
- The second constraint set uses only one XDC file containing all the constraints.

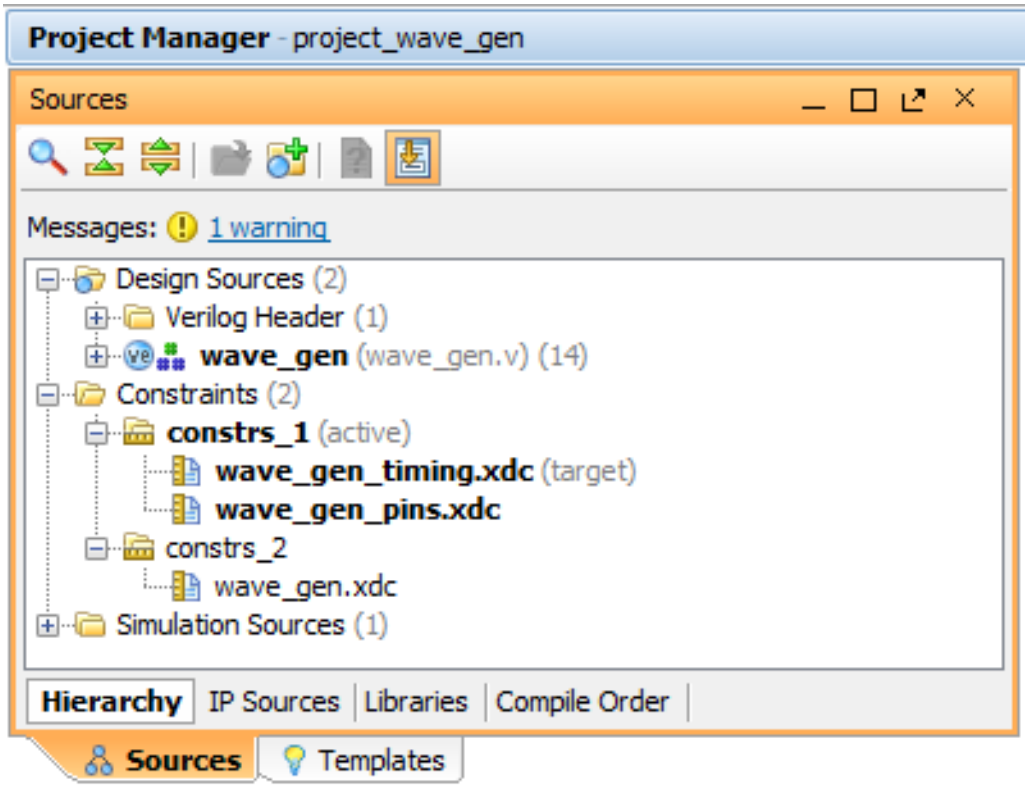

Figure 2-1: Single or Multi XDC

Non-Project Flows

To get the same result in a non-project flow, 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
opt_design
place_design
route_design
```

Using a Constraint File for Synthesis or Implementation

You can use a constraint file for:

- Synthesis only
- Implementation only
- Both synthesis and implementation

Edit the constraint file properties to specify whether the file will be used for synthesis only, implementation only, or both.

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**.
3. Select **Apply**.

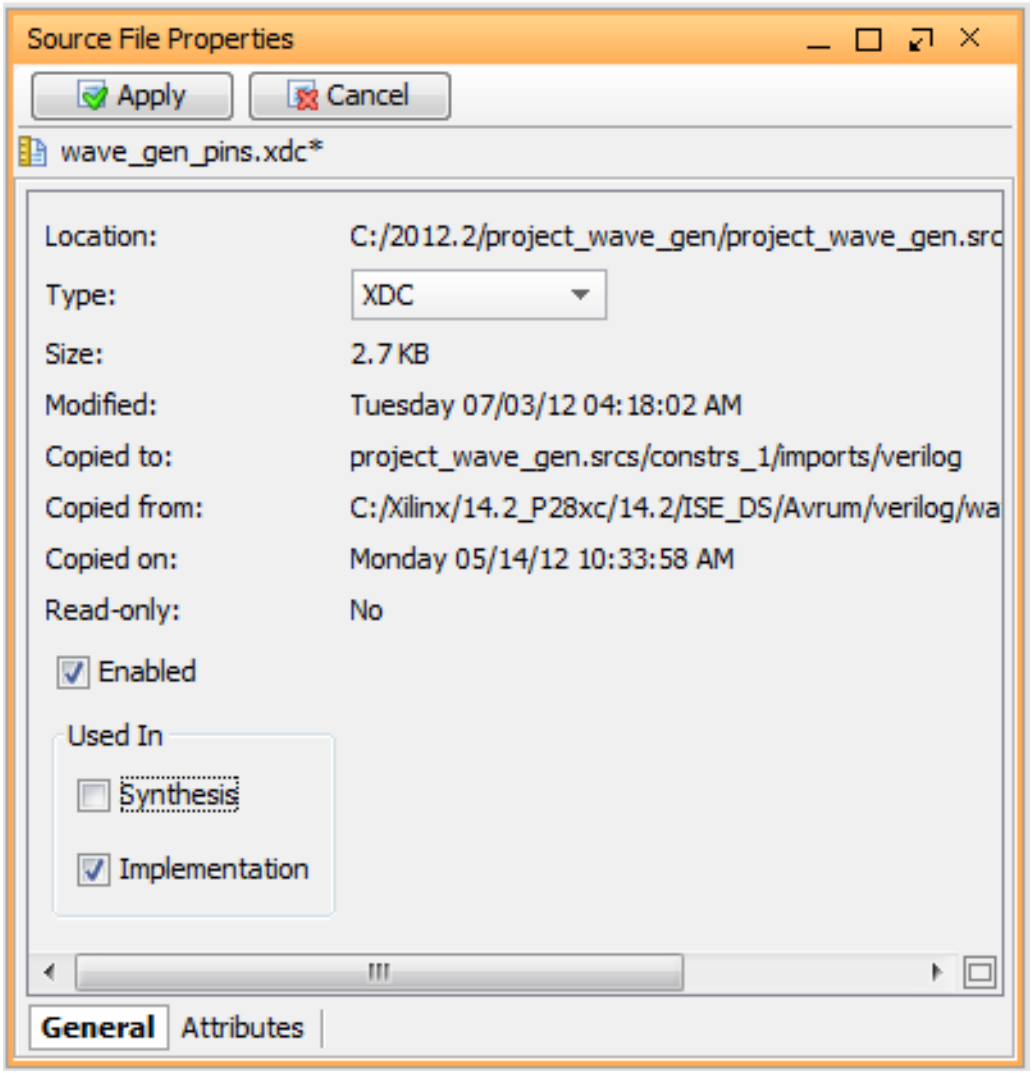

Figure 2-2: **Source File Properties Window**

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 IDE without a project, you can read in the constraints directly between any steps of the flow.

The following Tcl script shows how to read two XDC files:

```
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: **Reading Two XDC Files**

File Name	File Placement	Used For
wave_gen_timing.xdc	Before synthesis	• Synthesis • Implementation
wave_gen_pins.xdc	After synthesis	• Implementation

Ordering Your Constraints

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 6, XDC 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 Constraints Sequence

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
# 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 XDC file
```

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 XDC file, as well as across all the XDC files in your design.

XDC File Order

The order of the XDC 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.

Constraint Files Order

In a project flow, the constraints are located in a constraints set. When opening the elaborated or post-synthesis netlist, the constraint files are loaded in the same sequence as the way they are listed in the constraints set, that is, from top to bottom as displayed in the Vivado IDE.

For example, Figure 2-1, Single or Multi XDC, page 8, shows that the constraints set **constr_1** contains two XDC files.

Table 2-2: **Constraint Files Order**

FIle Order	File Name	Read In
First	wave_gen_timing.xdc	First
Second	wave_gen_pins.xdc	Second

Changing Read Order

To change the read order:

1. Select the XDC file you want to move.
2. Drag and drop the XDC file to the desired place in the list.

For the example shown in Figure 2-1, Single or Multi XDC, page 8, the equivalent Tcl command is:

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

In a non-project flow, the sequence of the **read_xdc** calls determines the order of the constraint files.

If you use the native IPs that come with a constraint file, the IP XDC files are loaded after your files, in the same sequence as the IPs are listed in the IP Sources window. For example, Figure 2-3, XDC Files in the IP Sources, shows that one of the project IPs comes with an XDC file.

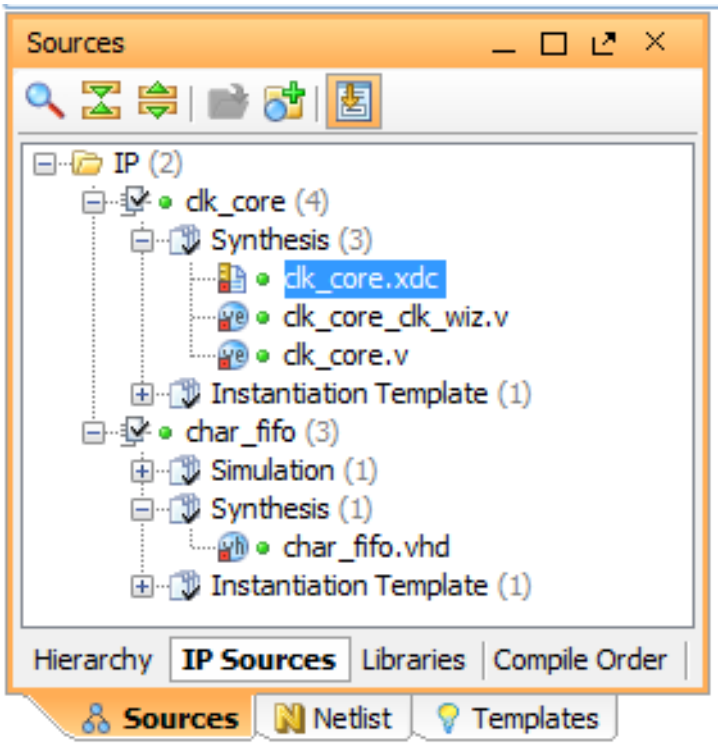

Figure 2-3: **XDC Files in the IP Sources**

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

```
Parsing XDC File [C:/project_wave_gen.srcs/constrs_2/wave_gen_all.xdc]
INFO: [Timing 38-35] Done setting XDC timing constraints.
[C:/project_wave_gen.srcs/constrs_2/wave_gen_all.xdc:9]
INFO: [Timing 38-2] Deriving generated clocks
[C:/project_wave_gen.srcs/constrs_2/wave_gen_all.xdc:9]
Finished Parsing XDC File [C:/project_wave_gen.srcs/constrs_2/wave_gen_all.xdc]
Parsing XDC File [c:/project_wave_gen.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.srcs/sources_1/ip/clk_core/clk_core.xdc] for cell
'clk_gen_i0/clk_core_i0/inst'
```

You cannot change the IP XDC files order. If you must modify the order, do the following:

1. Disable the corresponding IP XDC files (see Properties).
2. Copy their content.
3. Paste the content into one of the XDC files included in your constraints set.
4. Update the names with complete hierarchical path in the copied IP XDC constraints.

Constraints Sequence Editing

The Vivado IDE constraints manager saves any edited constraint back to its original location in the XDC files. Any new constraint is saved at the end of the constraint file marked as *target*. In most 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 on disk 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 XDC files. This is especially important for timing constraints.*

Entering Constraints

The Vivado IDE provides several ways to enter your 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.

Saving Constraints in Memory

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 (except for the Timing Constraints editor).

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.

To manually save your constraints:

- Click the **Save** button, or
- Select **File > Save Constraints**.

Constraints Editing Flow Options

Figure 2-5, Constraints Editing Flow, shows the recommended flow options. Do not use these options at the same time. Mixing these options may 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 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 between:

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

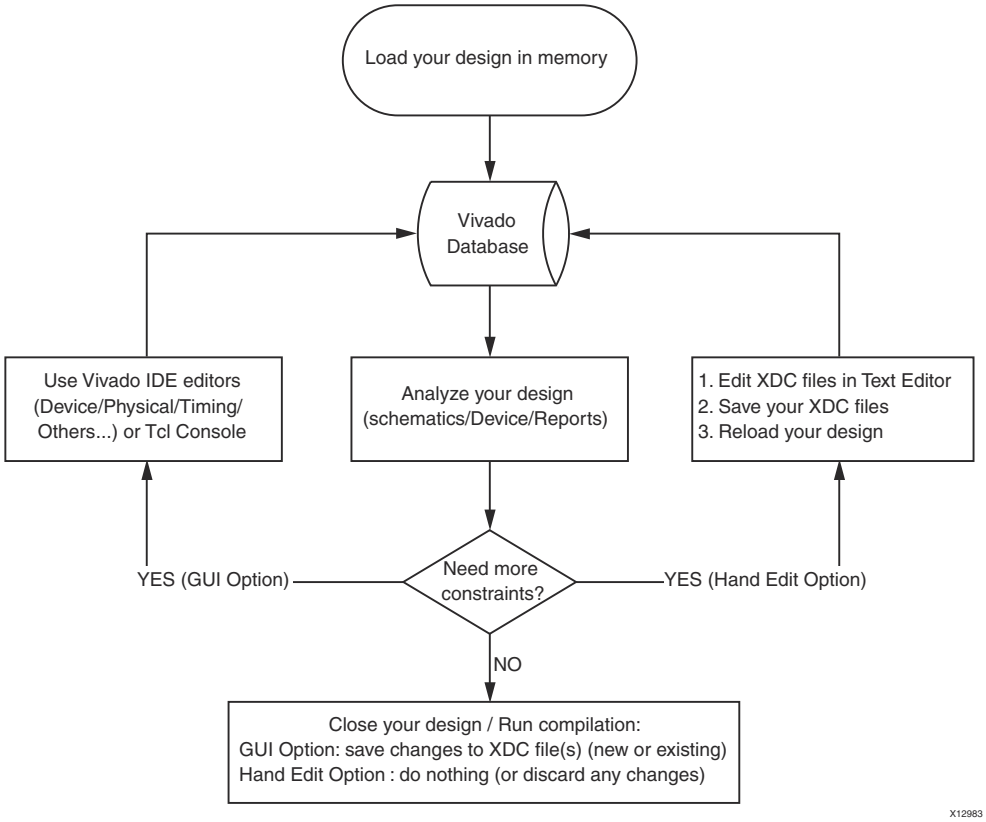

Figure 2-5: **Constraints Editing Flow**

Constraints creation is iterative. You can use interface features in some cases, and hand edit the constraint file in others.

Within each iteration described on Figure 2-5, Constraints Editing Flow, 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.

Pin Assignment

To create and edit existing top-level ports placement:

1. Select the I/O Planning pre-configured layout.
2. Open the windows shown in the following table.

Table 2-3: **Creating and Editing Existing Top-Level Ports Placement**

Window	Function
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 the *Vivado Design Suite User Guide: IO and Clock Planning (UG899)*.

Clock Resources Assignment

To view and edit the placement of your clock trees:

1. Select the Clock Planning pre-configured layout.
2. Open the windows shown in the following table.

Table 2-4: **Viewing and Editing the Placement of Clock Trees**

Window	Function
Clock Resources	• View the connectivity between the clock resources in the architecture. • View where your clock tree cells are currently located.
Netlist	• Drag and drop the clock resources from your netlist to a specific location in the Clock Resources window or Device window.

For more information on Clock Resources Assignment, see the *Vivado Design Suite User Guide: IO and Clock Planning (UG899)*.

Floorplanning

To create and edit Pblocks:

1. Select the Floorplanning pre-configured layout.
2. Open the windows shown in the following table.

Table 2-5: **Creating and Editing Pblocks**

Window	Function
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 the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)*.

Timing Constraints

The Timing Constraints window is available for Synthesized and Implemented designs only. For elaborated design constraints, see Creating Synthesis Constraints.

You can open the Timing Constraints window using one of the following three options, displayed in Figure 2-6:

- 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**.

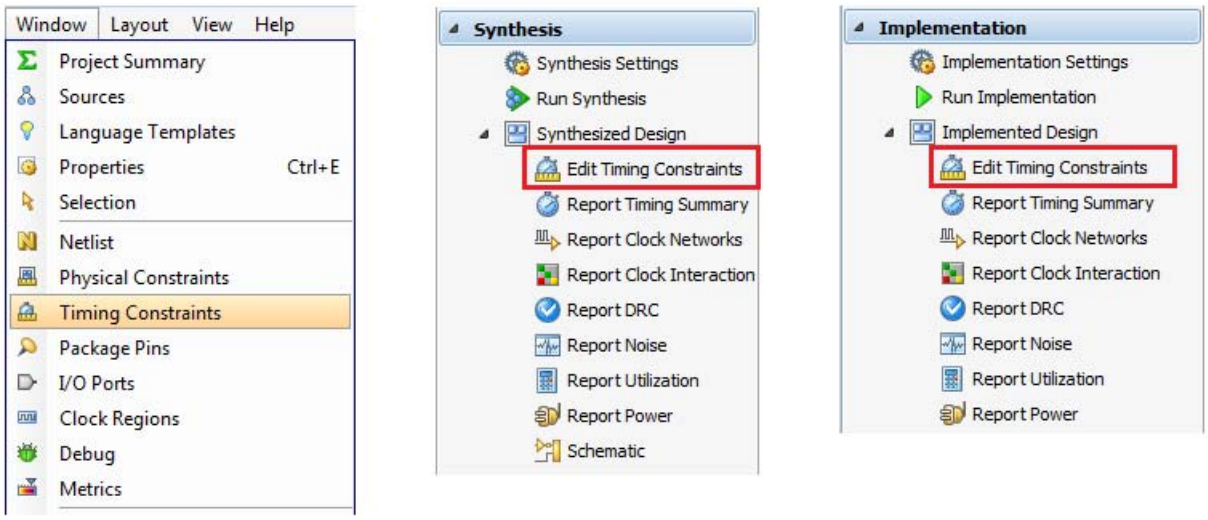

Figure 2-6: **Multiple Methods for Opening the Timing Constraints Window**

The Timing Constraints editor displays the timing constraints in memory, in either:

- The same sequence as in the XDC file, or
- The same sequence in which you entered them in the Tcl Console.

Constraints Spreadsheet

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

Although Xilinx recommends that you use the constraints creation wizards, you can create an additional constraint of the same type by clicking the **+** button. A new line in the spreadsheet is added, allowing you to provide the value for each option.

To verify that the new constraint has been added at the end the constraints, look at the number displayed in the Position column.

This new constraint is not validated or applied in memory until you click **Apply**. Clicking **Apply**:

- Resets the timing constraints in memory before applying them all.
- Does not save the constraints to the XDC file.

Constraints Creation, Grouped by Category

When you select a constraint, the corresponding spreadsheet appears on the right sub-window. To create a new constraint, double click the name. A wizard allows you to specify the value for each option. When you click **OK**, the wizard:

1. Validates the syntax.
2. Applies it 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.

- To delete a constraint, select it and click **X**.
- Use the spreadsheet view to edit constraint options and re-apply it.

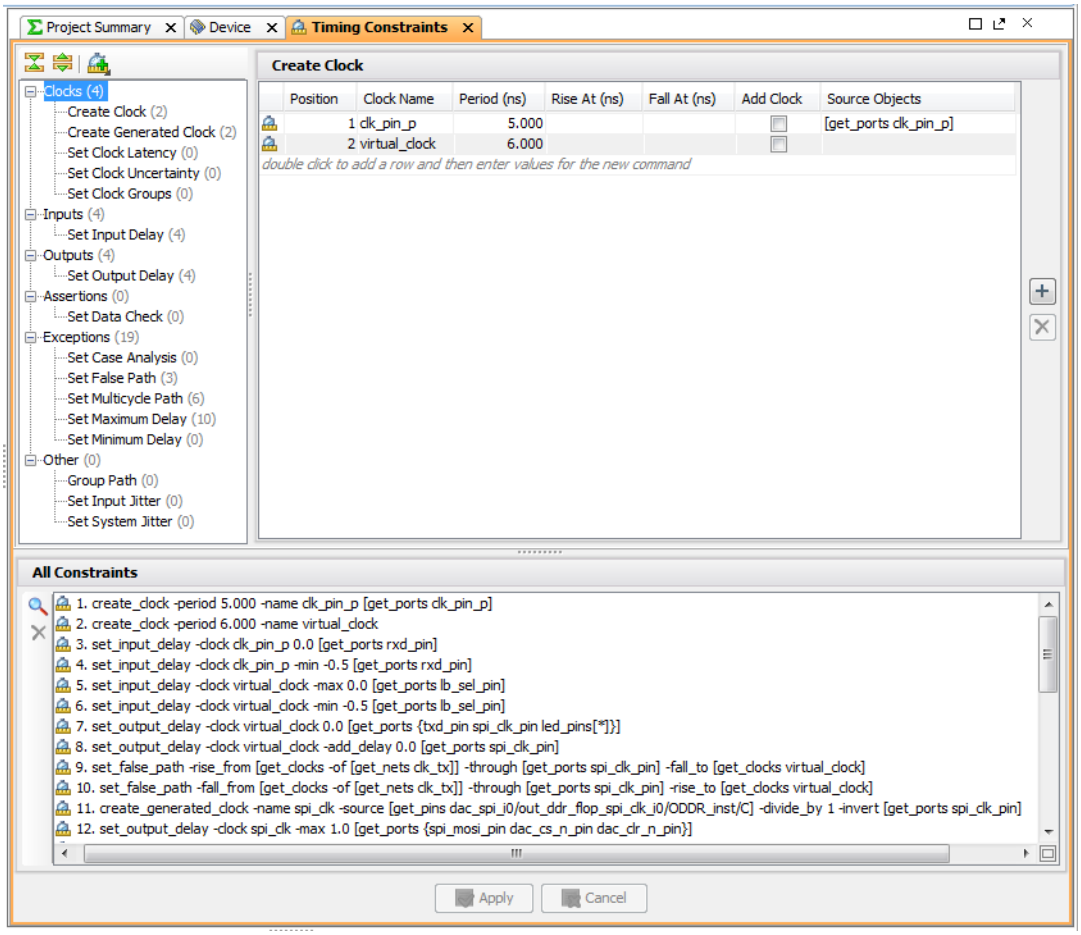

Figure 2-7: **Timing Constraints Editor**

If you type a new valid timing constraint in the Tcl Console, it appears immediately at the end of the list of existing constraints in the Timing Constraints window

When you create a new constraint, or edit an existing constraint in the spreadsheet view in the Timing Constraints window, the constraint is not applied in memory until you click **Apply**.

CAUTION! *Do not enter new constraints in the Tcl Console if any constraints in the Timing Constraints editor 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 add a new constraint or edit an existing one.*

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

XDC Templates

Access XDC templates from the Language Templates window.

XDC Template Contents

The XDC templates include:

- The most common timing constraints such as:
  - Clock definitions
  - Jitter
  - Input/output delay
  - 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 use Tcl variables to capture the design requirements and use them 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.

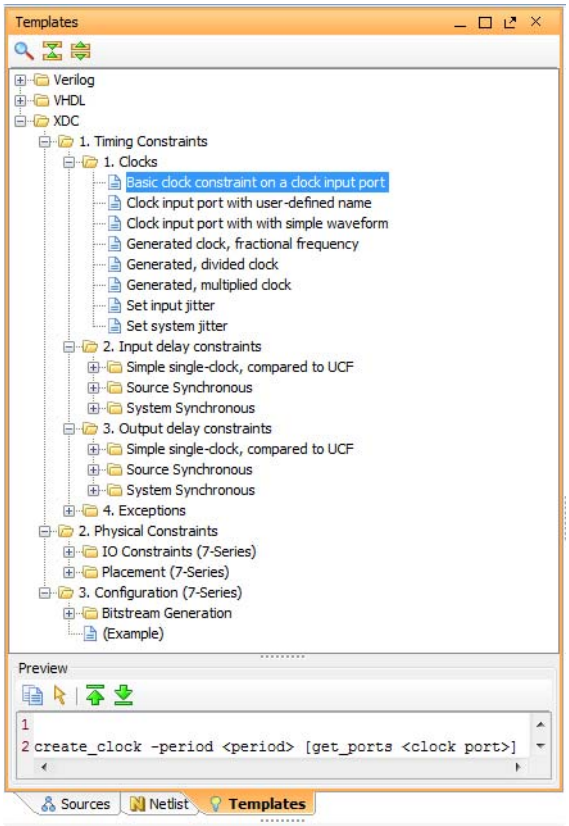

Figure 2-8: **XDC Templates**

Creating Synthesis Constraints

The Vivado IDE synthesis engine 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 FPGA devices 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 three categories of constraints for the Vivado IDE synthesis:

- RTL Attributes
- Timing Constraints
- Physical and Configuration Constraints

RTL Attributes

RTL attributes must be written in the RTL files. They usually correspond to directives related to 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 the *Vivado Design Suite User Guide: Synthesis (UG901)*.

Only the DONT_TOUCH attribute can be set from the XDC file as a property on a netlist object.

DONT_TOUCH Attribute Example

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

Timing Constraints

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

Physical and configuration constraints are ignored by the synthesis algorithms.

RTL-Based XDC Iterations

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 2-9, Creating Constraints with the Elaborated Design. 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. You will not be able to run any timing report as this is not supported on elaborated netlist.

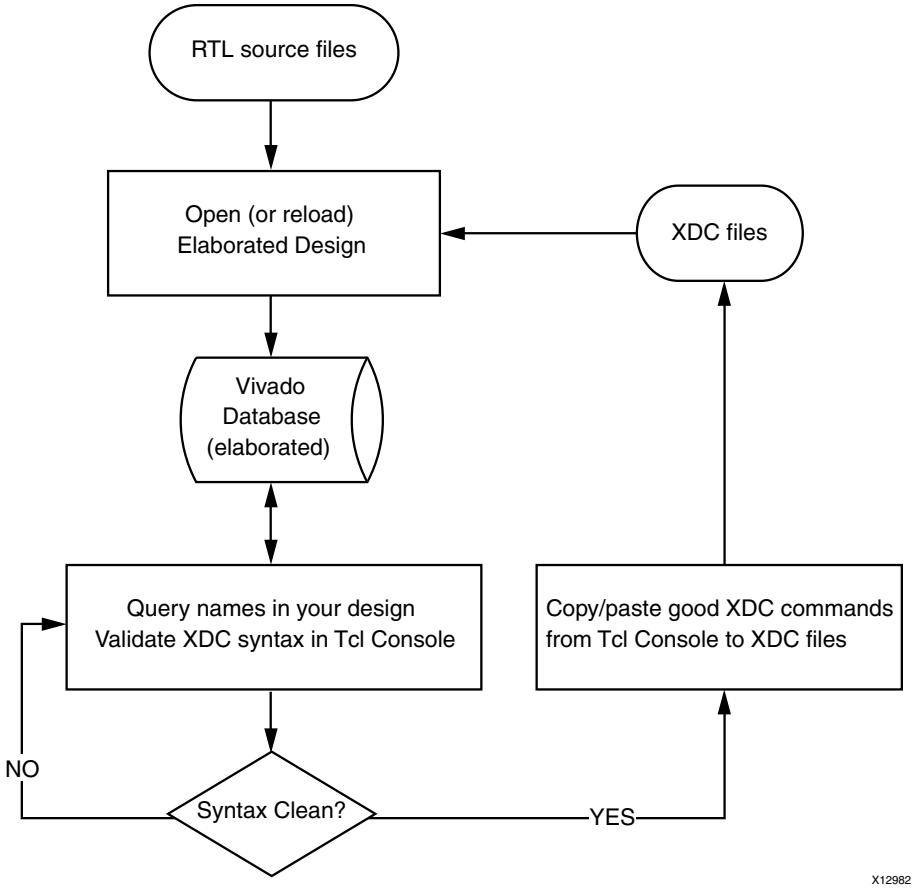

Figure 2-9: **Creating Constraints with the Elaborated Design**

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

Single-Bit Register Names

By default, the name is based on the RTL name, plus the **_reg** suffix.

Single-Bit Register Name VHDL Example

```
signal wbDataForInputReg : std_logic;
```

Single-Bit Register Name Verilog Example

```
reg wbDataForInputReg;
```

Single-Bit Register Name Elaborated Design Example

```
wbDataForInputReg_reg
```

Figure 2-10, Single-Bit Register in Elaborated Design, shows the schematic of the register, and more particularly its pins. It is also possible to refer to its pins in the XDC command if necessary.

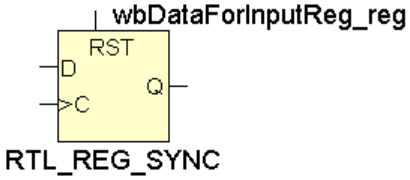

Figure 2-10: **Single-Bit Register in Elaborated Design**

Multi-Bit Register Names

By default, the name is based on the RTL name, plus the **_reg** suffix. You can refer to individual bits in your XDC constraints, even if they cannot be queried in the elaborated design.

Multi-Bit Register Name VHDL Example

```
signal validForEgressFifo : std_logic_vector(13 downto 0);
```

Multi-Bit Register Name Verilog Example

```
reg [13:0] validForEgressFifo;
```

Multi-Bit Register Name Elaborated Design Example

```
validForEgressFifo_reg
```

Figure 2-11, Multi-Bit Register in Elaborated Design, shows the schematic of the register. The pins appear as vectors, which is just for simplifying the visual representation.

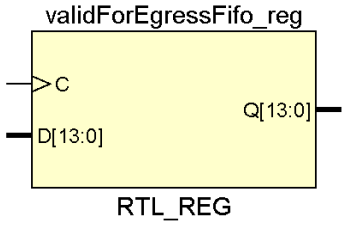

Figure 2-11: **Multi-Bit Register in Elaborated Design**

You can still constrain each register individually or as a group by using the following names:

- Register bit **0** only

  ```
  validForEgressFifo[0]
  ```

- All register bits

  ```
  validForEgressFifo[*]
  ```

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.

Absorbed Registers and Nets

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

Hierarchical Names

Unless you plan to force Vivado synthesis to keep the complete hierarchy of your design, some or all levels of the hierarchy will be flattened during synthesis. For more information, see the *Vivado Design Suite User Guide: Synthesis (UG901)*.

RECOMMENDED: *Use fully resolved hierarchical names in your synthesis constraints. 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.

RTL 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 based on the tool options Flat Netlist and Hierarchical Netlist.

Flat Netlist Example

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

Hierarchical Netlist Example

```
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:

- Use **get_*** commands without the **-hierarchical** option
- Mark explicitly with the **/** character all the levels of hierarchy as they show in the RTL 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
```

CAUTION! *Do not (a) attach constraints to hierarchical pins during synthesis for the same reason as explained above for hierarchical cells; or (b) 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.*

For more information, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)*.

Some pre-synthesis constraints may no longer apply properly because of the transformations performed by synthesis on the design. To resolve these problem:

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.

Creating Implementation Constraints

Once you have a synthesized netlist, you can load it into memory together with the XDC files enabled for implementation. You can run timing analysis in order to:

- Correct the timing constraints based on the netlist names and save them to an implementation-only XDC file.
- Add missing constraints, such as asynchronous and exclusive clock groups.
- Add timing exceptions, such as multicycle paths and 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.

The netlist-based XDC iteration is shown in Figure 2-12, page 29.

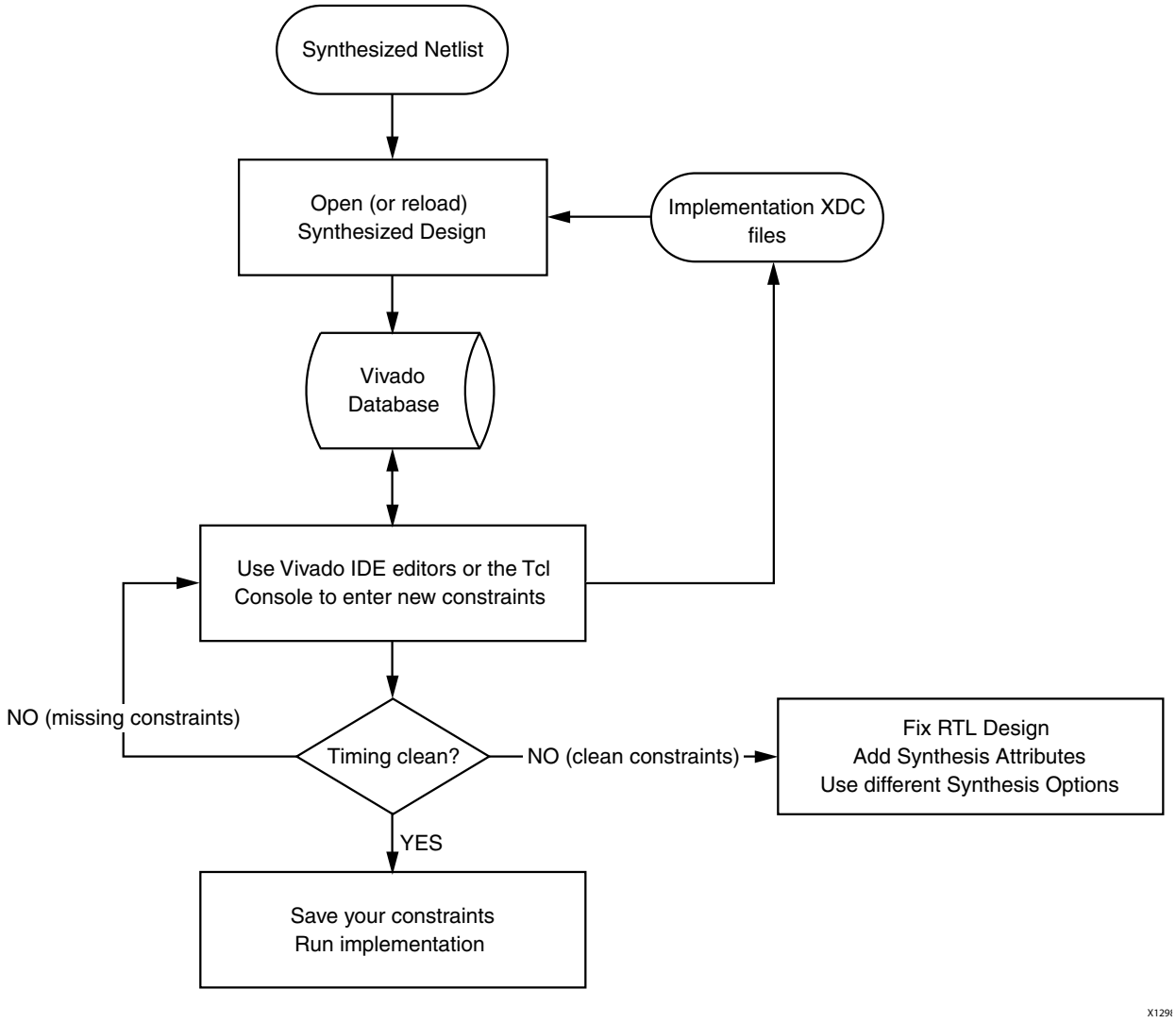

Figure 2-12: **Creating Constraints with the Synthesized Design**

Before proceeding to implementation, you must verify that your design does not show any major timing violation. While the implementation tool will place the cells of the most critical paths close to each other, and use the fastest routing resources between them, the tool is unable to resolve large violations.

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 the *Vivado Design Suite User Guide: Synthesis (UG901)*.

Timing Analysis

Before adding timing constraints to your design, you must understand the fundamentals of timing analysis, and the terminology associated with it. This chapter discusses some of key concepts used by the Vivado™ Integrated Design Environment (IDE) timing engine.

Timing Paths

Timing paths are defined by the connectivity between the instances of the design. In digital designs, timing paths are formed by a pair of sequential elements controlled by the same clock, or by two different clocks.

Common Timing Paths

The most common paths in any design are:

- Input Port to Internal Sequential Cell Path
- Internal Path from Sequential Cell to Sequential Cell
- Internal Sequential Cell to Output Port Path
- Input Port to Output Port Path

Input Port to Internal Sequential Cell Path

In an *input port to internal sequential cell path*, the data:

- Is launched outside the device by a port clock.
- Reaches the device port after a delay called the input delay (SDC definition).
- Propagates through the device internal logic before reaching a sequential cell clocked by the destination clock.

Internal Path from Sequential Cell to Sequential Cell

In an *internal path from sequential cell to sequential cell*, the data:

- Is launched inside the device by a sequential cell, which is clocked by the source clock.
- Propagates through some internal logic before reaching a sequential cell clocked by the destination clock.

Internal Sequential Cell to Output Port Path

In an *internal sequential cell to output port path,* the data:

- Is launched inside the device by a sequential cell, which is clocked by the source clock.
- Propagates through some internal logic before reaching the output port.
- Is captured by a port clock after an additional delay called output delay (SDC definition).

Input Port to Output Port Path

In an *input port to output port path*, the data :

- Propagates directly from an input port to an output port without being latched inside the device. They are commonly called in-to-out data paths.

A port clock can be a virtual clock or a design clock.

Path Example

Figure 3-1 shows the paths described above. In this example, the design clock `CLK0` can be used as the port clock for both `DIN` and `DOUT` delay constraints.

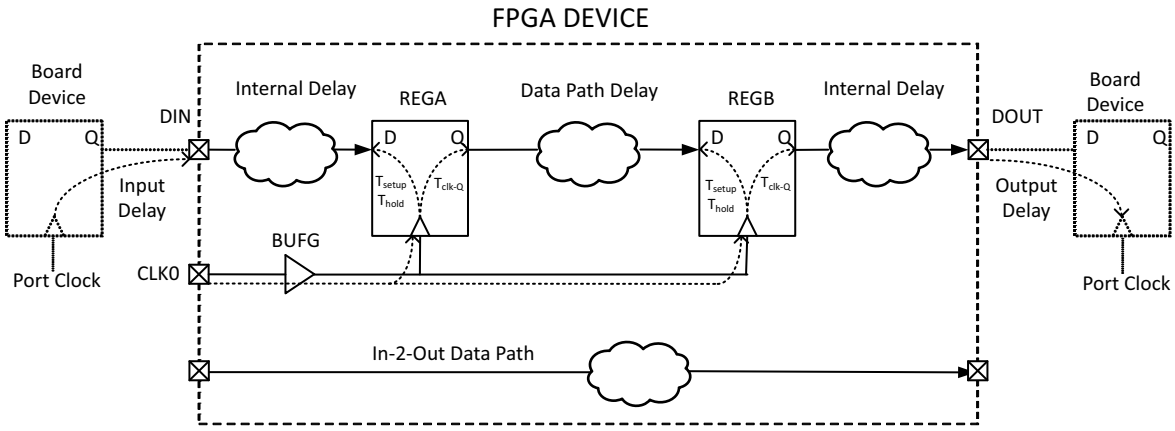

Figure 3-1: **Path Example**

Timing Path Sections

Each timing path is composed of three sections:

- Source Clock Path
- Data Path
- Destination Clock Path

Source Clock Path

This source clock path is the path followed by the source clock from its source point (typically an input port) to the clock pin of the launching sequential cell. For a timing path starting from an input port, there is no source clock path.

Data Path

For internal circuitry, the data path is the path between the launching and capturing sequential cells.

The active clock pin of the launching sequential cell is called the *path startpoint*.

The data input pin of the capturing sequential cell is called the *path endpoint*.

For an input port path, the data path starts at the input port. The input port is the path startpoint.

For an output port path, the data path ends at the output port. The output port is the path endpoint.

Destination Clock Path

The destination clock path is the path followed by the destination clock from its source point, typically an input port, to the clock pin of the capturing sequential cell.

For a timing path ending at an output port, there is no destination clock path.

Figure 3-2 shows the three sections of a typical timing path.

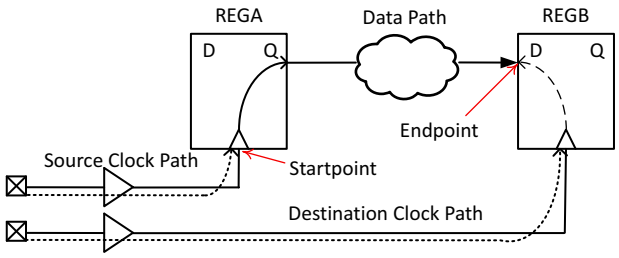

Figure 3-2: **Typical Timing Path**

Setup and Hold Analysis

The Vivado IDE analyzes and reports slack at the timing path endpoints. The slack is the difference between the data required time and the data arrival timing at the path endpoint. If the slack is positive, the path is considered functional from a timing point of view.

Setup Check

To calculate the data required time for setup analysis, the timing engine:

1. Determines the common period between the source and destination clock. If none can be found, up to 1,000 clock cycles are considered for the analysis.
2. Examines all rising and falling edges of the startpoint and endpoint clocks over their common period.
3. Determines the smallest positive delta between any two active edges. This delta is called the timing path requirement for setup analysis.

Setup Path Requirement Example

Consider a path between two registers which are sensitive to the rising edge of their respective clock. The active clock edges of this path are the rising edges only. The clocks are defined as follows:

- **clk0** has a period of 6ns
- **clk1** has a period of 4ns

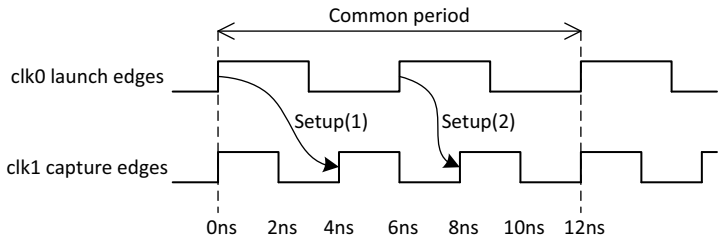

Figure 3-3: **Setup Path Requirement Example**

Figure 3-3 shows that there are two unique source and destination clock edges that qualify for setup analysis: setup(1) and setup(2).

The smallest positive delta from **clk0** to **clk1** is 2ns, which corresponds to setup(2):

```
Source Clock Launch Edge Time: 0ns + 1 * T(clk0) = 6ns
Destination Clock Capture Edge Time: 0ns + 2 * T(clk1) = 8ns
Setup Path Requirement = capture edge time – launch edge time = 2ns
```

When computing the path requirement, two important considerations are made:

1. The clock edges are ideal, that is, the clock tree insertion delay is not considered yet.
2. The clocks are phase-aligned at time zero by default, unless their waveform definition introduces a phase-shift. Asynchronous clocks do not have a known phase relationship. The timing engine applies the default assumption when analyzing paths between them. For more information on asynchronous clocks, see the following sections.

Data Required Time For Setup Analysis

The data required time for setup analysis is the time before which the data must be stable in order for the destination cell to latch it safely. Its value is based on:

- Destination clock capture edge time
- Destination clock delay
- Source and destination clock uncertainty
- Destination cell setup time

Data Arrival Time For Setup Analysis

The data arrival time for setup analysis is the time it takes for the data to be stable at the path endpoint after being launched by the source clock. Its value is based:

- Source clock launch edge time
- Source clock delay
- Datapath delay

The datapath delay includes all the cell and net delays, from the startpoint to the endpoint.

In the timing reports, the Vivado IDE considers the setup time as being part of the datapath. Accordingly, the equation for data arrival and required time are:

```
Data Required Time (setup) = destination clock capture edge time
                           + destination clock path delay
                           - clock uncertainty

Data Arrival Time (setup) = source clock launch edge time
                          + source clock path delay
                          + datapath delay
                          + setup time
```

The setup slack is the difference between the required time and the arrival time:

```
Slack (setup) = Data Required Time – Data Arrival Time
```

A negative setup slack on a register input data pin means that the register can potentially latch an unknown value and go to a metastable state.

Hold Check

The hold slack computation is directly connected to the setup slack computation. While the setup analysis validates that data can safely be captured under the most pessimistic scenario, the hold analysis ensures that:

- The same data cannot be wrongly captured by the previous destination clock edge.
- The data launched by the next source clock edge cannot be captured by the destination clock edge used for setup analysis.

Consequently, in order to find the timing path requirement for hold analysis, the timing engine considers all possible combinations of source and destination clock edges for setup analysis.

For each possible combination, the timing engine:

- Examines the time difference between the launch edge and the capture edge minus 1 destination clock period.
- Examines the time difference between the launch edge plus 1 source clock period and the capture edge.
- Keeps only the launch and capture edges associated with the greatest time difference.

Hold Path Requirement Example

Consider the clocks used in Setup Path Requirement Example, page 33. There are only two possible edge combinations for setup analysis:

```
Setup Path Requirement (S1) = 1*T(clk1) - 0*T(clk0) = 4ns
Setup Path Requirement (S2) = 2*T(clk1) - 1*T(clk0) = 2ns
```

The corresponding hold requirements are as follows.

```
For setup S1:
  Hold Path Requirement (H1a) = (1*T(clk1) - 1*T(clk1)) - 0*T(clk0) = 0ns
  Hold Path Requirement (H1b) = 1*T(clk1) - (0*T(clk0) + 1*T(clk0)) = -2ns
For setup S2:
  Hold Path Requirement (H2a) = (2*T(clk1) - 1*T(clk1)) - 1*T(clk0) = -2ns
  Hold Path Requirement (H2b) = 2*T(clk1) - (1*T(clk0) + 1*T(clk0)) = -4ns
```

The greatest hold requirement is 0ns, which corresponds to the first rising edge of both source and destination clocks.

Hold Path Requirement Example, page 36 shows the setup check edges and their associated hold checks.

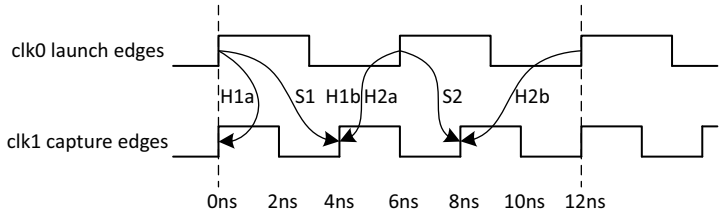

Figure 3-4: **Hold Path Requirement Example**

In this example, the final hold requirement is not derived from the tightest setup requirement. This is because all possible setup edges were considered in order to find the most challenging hold requirement.

As in setup analysis, the data required time and the data arrival time are calculated based on:

- Source clock launch edge time
- Destination clock capture edge time
- Source and destination clock delays
- Clock uncertainty
- Datapath delay
- Destination register hold time

```
Data Required Time (hold) = destination clock capture edge time
                          + destination clock path delay
                          + clock uncertainty
Data Arrival Time (hold) = source clock launch edge time
                          + source clock path delay
                          + datapath delay
                          - hold time
```

The hold slack is the difference between the required time and the arrival time:

```
Slack (hold) = Data Arrival Time - Data Required Time
```

A positive hold slack means that the data cannot be captured by the wrong clock edges under the most pessimistic conditions. A negative hold slack means that the wrong data is captured, and the register can potentially go to a metastable state.

Recovery and Removal Analysis

The recovery and removal timing checks are similar to setup and hold checks, except that they apply to asynchronous data pin such as *set* or *clear*.

For a register with an asynchronous reset:

- The recovery time is the minimum time before the next active clock edge after the asynchronous reset signal has toggled to its inactive state in order to properly latch a new data.
- The removal time is the minimum time after an active clock edge before the asynchronous reset signal can be safely toggled to its inactive state.

The following equations describe how the slack is computed for each check.

Recovery Check

The equations below describe how the following are computed:

```
Data Required Time (recovery) = destination clock edge start time
                                + destination clock path delay
                                - clock uncertainty

Data Arrival Time (recovery) = source clock edge start time
                               + source clock path delay
                               + datapath delay
                               + recovery time

Slack (recovery) = Data Required Time – Data Arrival Time
```

Removal Check

The equations below describe how the following are computed:

```
Data Required Time (removal) = destination clock edge start time
                               + destination clock path delay
                               + clock uncertainty

Data Arrival Time (removal) = source clock edge start time
                              + source clock path delay
                              + datapath delay
                              - removal time

Slack (removal) = Data Arrival Time – Data Required Time
```

As with setup and hold checks, a negative recovery slack or removal slack means that the register can go to a metastable state, and propagate an unknown electrical level through the design.

Defining Clocks

About Clocks

In digital designs, clocks represent the time reference for reliably transferring data from register to register. The Vivado™ Integrated Design Environment (IDE) timing engine uses the clock characteristics to:

- Compute timing paths requirements.
- Report the design timing margin by means of the slack computation.

For more information, see Chapter 3, Timing Analysis.

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

- A clock is defined on the driver pin or port of its tree root, which is called the source point.
- The clock 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 4-1, page 39, the clock **Clk0** has a 10ns period, a 50% duty cycle and 0ns phase. The clock **Clk1** has 8ns period, 75% duty cycle and a 2ns phase shift.

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

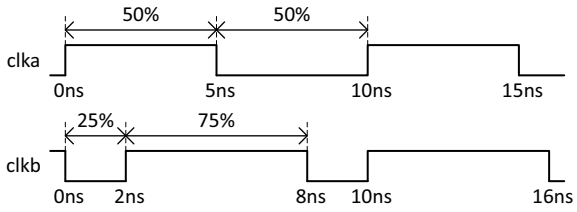

Figure 4-1: **Clock Waveforms Example**

Propagated Clocks

The period and waveform properties represent the ideal characteristics of a clock. When entering the FPGA device 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
- Phase error
- Any additional uncertainty that you have specified

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.

Dedicated Hardware Resources

The dedicated hardware resources of Xilinx® FPGA devices efficiently support a large number of design clocks. These clocks are usually generated by an external component on the board. They are fed to the design by means of 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.

Primary Clocks

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.

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, since the skew, and consequently the slack, value between the clocks can be inaccurate.*

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

Primary Clocks Examples

As shown in Figure 4-2, 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 10ns.
- Its duty cycle is 50%
- Its phase is not shifted.

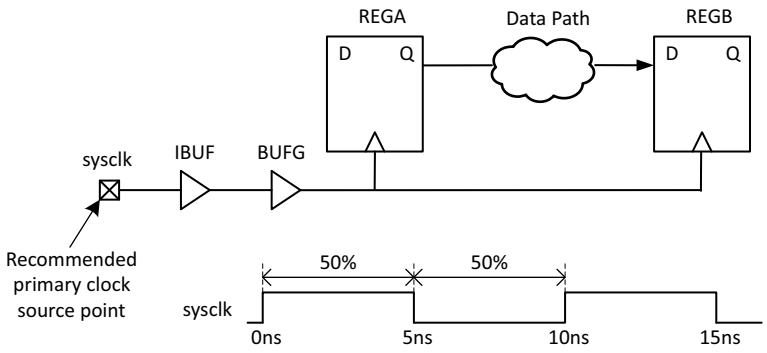

Figure 4-2: **Primary Clock Example**

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

Corresponding 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 10ns.
- 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 4-3 shows a transceiver gt0, which recovers the clock **rxclk** from a high speed link on the board. The clock **rxclk** has a 3.33ns 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]
```

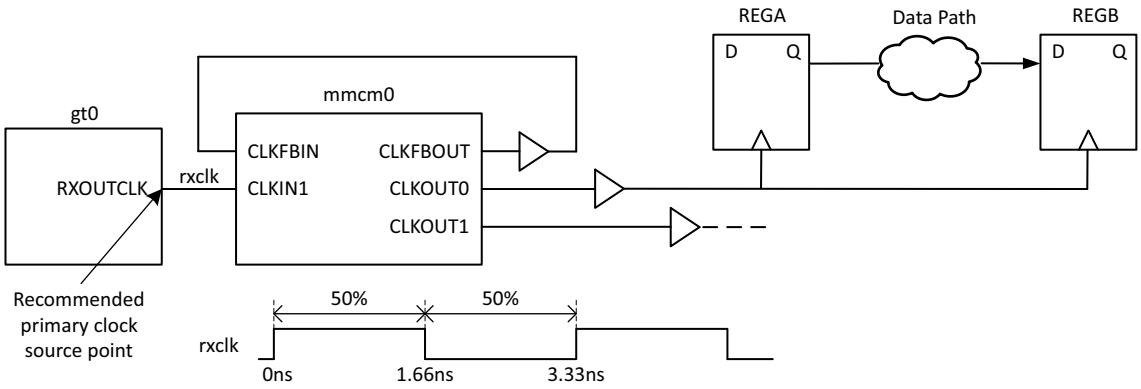

Figure 4-3: **GT Primary Clock Example**

Virtual Clocks

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 10ns 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.

Generated Clocks

There are two kinds of generated clocks:

- User Defined Generated Clocks
- Automatically Derived Clocks

About Generated 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 to a master clock. The master clock can be:

- A primary clock
- 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 generated clock can be:

- A simple period division, or
- A simple period multiplication, or
- Both a simple period division and a simple period multiplication in the case of an MMCM or PLL.

A generated clock can also be a simple copy of its master clock, with a phase shift or a waveform inversion.

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

User Defined Generated Clocks

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.

Example One: Simple Division by 2

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

Three 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]
```

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.

```
# Option 3: 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 an absolute 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 10ns period and a 50% duty cycle. It reaches the cell CMB which generates a clock with a 25% duty cycle, shifted by 90 degrees.

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

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 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 CMB instance which multiplies by 4/3 the master clock clkin:

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

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

Automatically Derived Clocks

Automatically derived clocks are also called auto-generated clocks. Their constraint is automatically created by the Vivado IDE on the output pins of the Clock Modifying Blocks (CMB), provided the associated master clock has already been defined. The CMBs are MMCMx, PLLx or BUFR primitives, including the PHASER_x ones in the MIG IP.

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 source point. The name of the auto-generated clock is based on the name of the net directly connected to the source point.

Automatically Derived Clock Example

Clock generated by a MMCM

The master clock **clkin** drives the input **CLKIN0** the MMCME2 instance **mmcm_i**. The name of the auto-generated clock is **cpuClk** and its definition point is **mmcm_i/CLKOUT0**

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 **aa/bb/usrclk**:

- The parent cell name is **aa/bb**.
- The generated clock name is **usrclk**.

Name Conflicts

In case of name conflict between two 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 explicitly define the generated clock constraints.

Clock Groups

The Vivado IDE assumes that all clocks are related by default unless you specify otherwise by adding clock groups constraints. The **set_clock_groups** command disables timing analysis between groups of clocks that you identify.

Use the schematic viewer or the Report Clock Networks to visualize the topology of the clock trees, and determine which clocks must remain related.

CAUTION! *Disabling 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.*

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.

For example, a generated clock and its master clock are *synchronous* because they propagate through the same netlist resources up to the generated clock source point.

Asynchronous Clock Groups

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 device 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.

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** is a recovered clock defined on the output of a GTP instance and reaches a second MMCM which generated the clocks **gtclkrx** and **gtclktx**.

Creating Asynchronous Clock Groups

Use the **–asynchronous** option to create asynchronous groups.

```
set_clock_groups -name async_clk0_clk1 -asynchronous -group {clk0 usrclk itfclk} \
  -group {clk1 gtclkrx gtclktx}
```

Dynamically Retrieving Generated Clock Names

If the name of the generated clocks cannot be predicted in advance, use **get_clocks –include_generated_clocks** to dynamically retrieve them. The **–include_generated_clocks** option is an SDC extension.

The previous example can also be written as:

```
set_clock_groups -name async_clk0_clk1 -asynchronous \
  -group [get_clocks -include_generated_clocks clk0] \
  -group [get_clocks -include_generated_clocks clk1]
```

Exclusive Clock Groups

Some designs have several operation modes that require the use of different clocks. The selection between the clocks is usually done with:

- A clock multiplexer such as BUFGMUX and BUFGCTRL, or
- A LUT

RECOMMENDED: *Avoid using LUTs in clock trees as much as possible.*

Because these cells are combinatorial cells, the Vivado DS propagates all incoming clocks to the output. With the Vivado IDE, several timing clocks can exist on a clock tree at the same time, which is convenient for reporting on all the operation modes at once, but is not possible in hardware.

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

- -logically_exclusive, or
- –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 FPGA devices:

- –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 FPGA devices.

Clock Latency, Jitter, and Uncertainty

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

Clock Latency

After propagating on the board and inside the FPGA device, 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 network latency (also called insertion delay) is either:

- Automatically estimated (pre-route design), or
- Accurately computed (post-route design)

The **set_propagated_clock** command triggers the propagation delay computation in standard SDC tools. This command is not needed with the Vivado IDE because by default:

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

For Xilinx FPGA devices, 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 and Clock Uncertainty

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

Input Jitter

Input jitter is the difference between successive clock edges with respect to variation from the nominal or ideal clock arrival times.

Use the **`set_input_jitter`** command to specify input jitter for each clock individually. The input jitter is not transmitted from a master clock to its derived clocks. Consequently, you must manually specify the input jitter, even for derived clocks.

System Jitter

System jitter is the overall jitter due to:

- Power supply noise
- Board noise
- 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.

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.

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 Vivado IDE recognizes timing only within the boundaries of the FPGA device, you must use the following commands to specify delay values that exist beyond these boundaries:

- set_input_delay
- set_output_delay

Input 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. When considering the application board, this 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 device, 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.

Using Input Delay Options

Although the **-clock** option is optional in the 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 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) output delay constraint.

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.

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

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.

Output Delay

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.

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

a. The data propagating from the output package pin of the FPGA device, 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 device.

Using Output Delay Options

Although the **-clock** option is *optional* in the SDC standard, it is *required* by the Vivado IDE.

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 different 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 input 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

The **-add_delay** option must be used if:

1. A max output delay constraint exists, and
2. You want to specify a second max output delay constraint.

The same is true for a min output delay constraint. This option is commonly used for to constrain an input port relative to more than one clock edge, for example, rising and falling edges for DDR interfaces or different board clocks of the path ends to different devices.

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.*

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_dd**r clock outside the device to the data input of the internal flip-flop sensitive to both rising and falling clock edges.

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 Vivado™ Integrated Design Environment (IDE) supports the timing exceptions commands shown in Table 5-1, Timing Exceptions Commands.

Table 5-1: **Timing Exceptions Commands**

Command	Function
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.
set_case_analysis	Performs timing analysis using logic constants or logic transitions on ports or pins to restrict the signals propagated through the design.

Min/Max Delay Constraints

The **`set_min_delay`** and **`set_max_delay`** commands can be used to:

- Constrain special paths, such as `in-to-out` I/O paths
- Override the default path requirement on selected paths usually defined by the clocks.

Table 5-2: **Min/Max Delay Constraints**

Constraint	Function
Min delay	Corresponds to the path requirement used for hold and removal analysis.
Max delay	Corresponds to the path requirement used for setup and recovery analysis.

The slack computation includes the clock skew of the path by default, unless:

- Path segmentation occurs when using improper startpoints in the constraint, in which case only the destination clock is considered.
- The **-datapath_only** option is used (**set_max_delay** only), in which case both source and destination clocks are ignored. For more information, see the following sections.

Command Options

The **-datapath_only** option is supported by the **set_max_delay** command only.

In general, use the options as shown in Table 5-3, Command Options.

Table 5-3: **Command Options**

Options	Attach To
• -from • -rise from • -fall_from	Valid startpoints such as input or bidir ports, and clock pins of sequential cells or a clock.
• -to • -rise_to • -fall_to	Valid endpoints such as output or bidir ports, and input data pins of sequential cells or a clock.
• -through • -rise_through • -fall_through	Any net or pin

Applying **set_max_delay** to a path does not affect the min delay analysis of the same path. The same behavior is true for **set_min_delay** with respect to max delay analysis. You must use false path constraints to explicitly remove the min or max delay requirement on the same path.

For example, the following constraints set the path requirement to 5ns for max delay analysis (setup check) and disable the same path for min delay analysis (hold check):

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

Datapath Only Options

With the **-datapath_only** option, the use model of **set_max_delay** is more restrictive. The **–from** option must be used.

Datapath Only Example One

data1_0_reg/Q is directly connected to **data12_0_reg**. This is a valid scenario.

```
> set_max_delay –from data1_0_reg/C –to data12_0_reg/D 5 –datapath_only
```

Path Segmentation

When using other pins than valid startpoints with the **–from** option of **set_max_delay** or **set_min_delay**, the Vivado IDE breaks the timing paths going through the specified pins and uses them as startpoints.

Using invalid endpoints with those commands also breaks the timing path through these pins as they become endpoints. This can be a problem. Path segmentation breaks the propagation of clocks of segmented paths, allowing large skew to appear on these paths. This should be accounted for in the **min** and **max** delay values specified in the constraint. In addition, any path segmentation created by improper use of the **set_max_delay** or **set_min_delay** commands affects both min and max delay analysis.

RECOMMENDED: *To avoid path segmentation, carefully choose valid startpoints and endpoints.*

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.

XDC 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.

Exceptions 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 and Minimum Delay Path (**set_max_delay** and **set_min_delay**)

4. Multicycle Paths (**set_multicycle_path**)

Note: For the same exception, the more specific the constraint, the higher the precedence.

Exceptions Priority Example

```
> 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 type of object matters, as well as the filter option used to specify a timing exception. Following is the sorted list from highest precedence to lowest:

1. **–from pin**
2. **–to pin**
3. **–through pin**
4. **–from clock**
5. **–to clock**

Physical Constraints

The Vivado™ Integrated Design Environment (IDE) enables design objects to be physically constrained by setting values of object properties. Examples include:

- IO constraints such as location and IO standard
- Placement constraints such as cell locations
- Routing constraints such as fixed routing
- Configuration constraints such as the configuration mode

Applying Constraints

Constraints can be applied by:

- Applying Constraints from an XDC Constraint File
- Applying Constraints Interactively Using Tcl Commands

Applying Constraints from an XDC Constraint File

When you use an XDC constraint file, the constraints are applied as the netlist is processed, in the order they appear in the file.

Applying Constraints Interactively Using Tcl Commands

When you use Tcl commands or a Tcl script, constraints are:

- Applied immediately to design objects in memory.
- Applied using the XDC **`set_property`** command.

Use the following syntax:

```
set_property <property> <value> <object list>
```

Location Constraint using LOC Property Example

```
set_property LOC SLICE_X32Y49 [get_cells XCOUNTER/BU5]
```

Critical Warnings

Critical Warnings are issued for invalid constraints in XDC files, including those applied to objects that cannot be found in the design.

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.*

For constraint definition and usage, see the *Vivado Design Suite Constraints Reference Guide (UG912)*.

Netlist Constraints

Netlist constraints are set on netlist objects such as ports, pins, nets or cells, to require the compilation tool to handle them in special way.

- CLOCK_DEDICATED_ROUTE

  Set CLOCK_DEDICATED_ROUTE on a net or a pin to indicate how the clock signal is expected to be routed.

- MARK_DEBUG

  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.

- DONT_TOUCH

  Set DONT_TOUCH on a cell or a hierarchical instance to preserve it during netlist optimizations.

IO Constraints

IO constraints configure:

- Ports
- Cells connected to ports

Typical constraints include:

- IO standard
- IO location

The Vivado™ Integrated Design Environment (IDE) supports many of the same IO constraints as the Integrated Software Environment (ISE™) Design Suite:

- DRIVE

  DRIVE sets the output buffer drive strength (in mA), available with certain IO standards only

- IOSTANDARD

  IOSTANDARD sets an IO Standard to an I/O buffer instance.

- SLEW

  SLEW sets the slew rate (rate of transition) behavior of a device output.

- IN_TERM

  IN_TERM sets the configuration of the input termination resistance for an input port

- OUT_TERM

  OUT_TERM sets the configuration of the output termination resistance for an output port

- DIFF_TERM

  DIFF_TERM turns on or off the 100 ohm differential termination for primitives such as IBUFDS_DIFF_OUT.

- KEEPER

  KEEPER applies a weak driver on an tri-stateable output or bidirectional port to preserve its value when not being driven.

- PULLDOWN

  PULLDOWN applies a weak logic low level on a tri-stateable output or bidirectional port to prevent it from floating.

- PULLUP

  PULLUP applies a weak logic high level on a tri-stateable output or bidirection port to prevent it from floating.

- DCI_VALUE

  DCI_VALUE determines which buffer behavioral models are associated with the IOB elements when generating the IBIS file.

- DCI_CASCADE

  DCI_CASCADE defines a set of master and slave banks. The DCI reference voltage is chained from the master bank to the slaves.

- INTERNAL_VREF

  INTERNAL_VREF frees the Vref pins of an I/O Bank and uses an internally generated Vref instead.

- IODELAY_GROUP

  IODELAY_GROUP groups a set of IDELAY and IODELAY cells with an IDELAYCTRL to enable automatic replication and placement of IDELAYCTRL in a design.

- IOBDELAY

  IOBDELAY sets the tap delay value of an IDELAY or IODELAY delay line cell.

- IOB

  IOB tells the placer to try to place FFs and Latches in I/O Logic instead of the slice fabric.

For more information on constraints, see the *Vivado Design Suite Properties Reference Guide (UG912)*.

Examples

Properties can be set on single objects or multiple objects using Tcl lists.

The following sets IOSTANDARD on two ports **mode0** and **mode1**:

```
% set_property IOSTANDARD LVCMOS18 [get_ports {mode0 mode1}]
```

Placement Constraints

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.

- HLUTNM

  A unique string name applied to two LUTs in the same hierarchy to control their placement on a single LUT site.

- PROHIBIT

  Disallows placement to a site.

- PBLOCK

  Attached to logical blocks to constrain them to a physical region in the FPGA.

- 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 specific BEL within a slice on the device.

Placement Types

There are two types of placement in the tools:

- Fixed Placement
- Unfixed Placement

Fixed Placement

Fixed placement is placement specified by the user through:

- Hand placement, or
- An XDC constraint
- Using either of the following on a cell object of the design loaded in memory:
  - IS_LOC_FIXED
  - IS_BEL_FIXED

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.

Placement Constraint Example One

Locate a block RAM at RAMB18_X0Y10 and fix its location.

```
% set_property LOC RAMB18_X0Y10 [get_cells u_ctrl0/ram0]
```

Placement Constraint Example Two

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]
```

Placement Constraint Example Three

Locate input bus registers in ILOGIC cells for shorter input delay.

```
% set_property IOB TRUE [get_cells mData_reg*]
```

Placement Constraint Example Four

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}]
```

Placement Constraint Example Five

Prevent the placer from using the first column of block RAMs.

```
% set_property PROHIBIT TRUE [get_sites {RAMB18_X0Y* RAMB36_X0Y*}]
```

Routing Constraints

Routing constraints are applied to net objects to control their routing resources.

Pin Locking

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_cell 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]
```

Fixed Routing

Fixed Routing is the mechanism for locking down routing, similar to Directed Routing in ISE. Locking down a net's routing resources involves three net properties. See the following table.

Table 7-1: **Net Properties**

Property	Function
ROUTE	Read-only net property
IS_ROUTE_FIXED	Flag to mark the whole route as fixed
FIXED_ROUTE	A net's fixed route portion

To guarantee that a net's 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 7-1, Simple Design to Illustrate Routing Constraints, and creates the constraints to fix the routing of the net **a** (selected in blue).

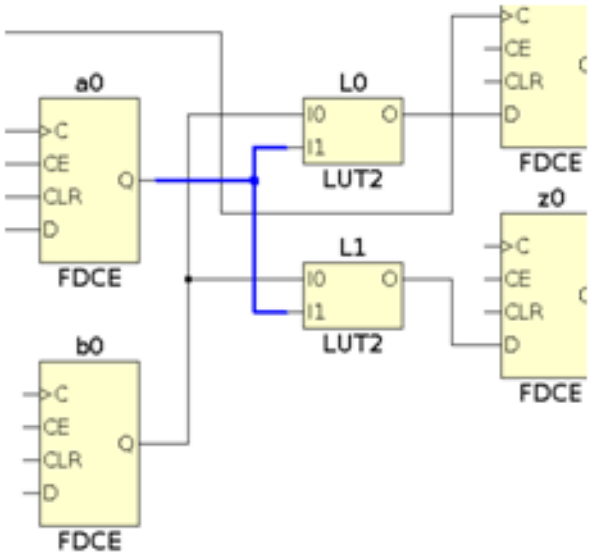

Figure 7-1: **Simple Design to Illustrate Routing Constraints**

You can query the routing information of any net after loading the implemented design in memory:

```
% set net [get_nets a]
% 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 doing:

```
% set_property IS_ROUTE_FIXED $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 pin mappings must also be captured as LOCK_PINS constraints to prevent the router from swapping pins. This would result in misalignment of the fixed route and the net's pins.

```
% 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 **a** are:

```
set_property LOC SLICE_X0Y47 [get_cells {a0 L0 L1}]
set_property BEL CFF [get_cells a0]
set_property BEL A6LUT [get_cells L0]
set_property BEL B6LUT [get_cells 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 a]
```

If you are using interactive Tcl commands instead of XDC, the following constraints can be set with the single **`place_cell`** command shown below:

- LOC
- BEL
- IS_LOC_FIXED
- IS_BEL_FIXED

```
place_cell a0 SLICE_X0Y47/CFF L0 SLICE_X0Y47/A6LUT L1 SLICE_X0Y47/B6LUT
```

For more information on **place_cell**, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)*.

Configuration Constraints

Configuration constraints are global constraints for bitstream generation that are applied to the current design. This includes constraints such as the configuration mode.

Configuration Constraint Example One

Set the CONFIG_MODE to M_SELECTMAP.

```
% set_property CONFIG_MODE M_SELECTMAP [current_design]
```

Configuration Constraint Example Two

Set device pins E11 and F11 to be voltage reference pins.

```
% set_property VREF {E11 F11} [current_design]
```

Configuration Constraint Example Three

Disable CRC checking.

```
% set_property BITSTREAM.GENERAL.CRC Disable [current_design]
```

For a list of bitstream generation properties and definitions, see the *Vivado Design Suite Tcl Command Reference Guide (UG835)*.

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

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.

References

These documents provide supplemental material useful with this guide:

- *Vivado Design Suite 2012.2 Documentation* (www.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm