Vivado Design Suite User Guide

Using Tcl Scripting

UG894 (v2012.4) December 18, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/16/2012	2012.3	Initial Xilinx release.
12/18/2012	2012.4	Documented Defining Tcl Hook Scripts in Chapter 1.

XILINX®

Table of Contents

Tcl Scripting in Vivado

Introduction

The Tool Command Language, or Tcl, is an interpreted programming language with variables, procedures, and control structures, to interface to a variety of design tools and to the design data.

The language is easily extended with new function calls, so that it has been expanded to support new tools and technology since its inception and adoption in the early 1990s. Because it is easily extended, it has been adopted as the standard application programming interface, or API, among most EDA vendors to control and extend their applications.

Xilinx® has adopted Tcl as the native programming language for the Vivado™ Design Suite, so that it is easily adopted and mastered by designers familiar with the industry standard language. The Tcl interpreter inside the Vivado Design Suite provides the full power and flexibility of the Tcl language to control the application, access design objects and their properties, and create custom reports. Using Tcl, you can adapt your design flow to meet specific design challenges.

The Tcl language provides built-in commands to read and write files to the local file system. This enables you to dynamically create directories, start FPGA design projects, add files to the projects, run synthesis and implementation. You can customize the reports generated from design projects, on device utilization and quality of results, to share across the organization.

You can also use the Tcl language to implement new design approaches, or work around existing problems, inserting and deleting design objects, or modifying properties as needed. You can write scripts to replay established portions of your design flow to standardize the process.

Many of the Tcl commands discussed in the following text and script examples are specific to the Vivado Design Suite. You will find detailed information regarding Vivado specific Tcl commands in the *Vivado Design Suite Tcl Command Reference Guide (UG835)*, or in the help system of the Vivado tool.

The Vivado tool writes a journal file called `vivado.jou` into the directory from which Vivado was launched. The journal is a record of the Tcl commands run during the session that can be used as a starting point to create new Tcl scripts.

Additional Tcl commands are provided by the Tcl interpreter that is built into the Vivado Design Suite. For Tcl built-in commands, Tcl reference material is provided by the Tcl Developer Xchange website, which maintains the open source code base and documentation for Tcl, and is located at http://www.tcl.tk

See http://www.tcl.tk/man/tcl/tutorial/tcltutorial.html for an introductory tutorial to the Tcl programming language.

In the text that follows you will see examples of Tcl commands, and Tcl scripts, and the results that are returned by the Vivado Design Suite when these commands are run. These example commands and their return values appear in this text with the following formats:

- Tcl commands and example scripts:

```
puts $outputDir
```

- Output to Tcl Console or results of Tcl commands:

```
./Tutorial_Created_Data/cpu_output
```

A Brief Overview of Tcl

A Tcl script is a series of Tcl commands, separated by new-lines or semicolons. A Tcl command is a string of words, separated by blanks or tabs. The Tcl interpreter breaks the command line into words and performs command and variable substitutions as needed. The Tcl interpreter reads the line from left to right, evaluates each word completely before attempting to evaluate the next. Command and variable substitutions are performed from left to right as the line is read.

A word is a string that can be a single word, or multiple words within braces, {}, or multiple words within quotation marks, "". Semicolons, brackets, tabs, spaces, and new-lines, within quotation marks or braces are treated as ordinary characters. However, the backslash, \, is treated as a special character even within braces and quotation marks, as discussed below. The first word identifies the command, and all subsequent words are passed to the command as arguments.

```
set outputDir ./Tutorial_Created_Data/cpu_output
```

In the preceding example, the first word is the Tcl `set` command, which is used to assign variables. The second and third words are passed to the `set` command as the variable name (`outputDir`), and the variable value (`./Tutorial_Created_Data/cpu_output`).

When a backslash, '\', is used in a word, then the Tcl interpreter performs backslash substitution. In most cases, this simply means that the character following the backslash is treated as a standard character in the word. This is used to add quotes, braces, dollar signs, or other special characters to a string. Refer to a Tcl/Tk reference source for more information on how the Tcl interpreter handles the backslash character.

There is a also difference between the use of braces and quotation marks. No substitutions are performed on the characters between the braces. Words, or strings, within braces are taken verbatim, and are not evaluated for variable or command substitution by the Tcl interpreter. The word will consist of exactly the characters between the outer braces, not including the braces themselves, as shown in the example below. Strings within quotation marks are evaluated, and variable and command substitutions are performed as needed. Command substitution, variable substitution, and backslash substitution are performed on the characters between quotes.

```
puts {The version of Vivado Design Suite is [version -short]}
The version of Vivado Design Suite is [version -short]

puts "The version of Vivado Design Suite is [version -short]"
The version of Vivado Design Suite is 2012.3
```

Notice in the example above, that the `[version -short]` command is substituted for the returned value when enclosed within quotation marks, but is not substituted when enclosed within braces. Keep substitution in mind when choosing to use either "" or {} to enclose a string of words.

Variable assignment is performed using the `set` command. You can access a previously assigned variable by specifying the name of the variable with a leading dollar sign, '$'. If a word starts with a dollar sign the Tcl interpreter performs variable substitution, replacing the variable with the value currently stored by the variable. The '$' is a reserved character in the Tcl language.

```
set outputDir ./Tutorial_Created_Data/cpu_output
puts $outputDir
./Tutorial_Created_Data/cpu_output
```

Commands can also be nested inside other commands within brackets, [], which are evaluated in a bottom-up manner. The Tcl interpreter recursively processes the strings within the brackets as a new Tcl script. A nested command can also contain other nested commands. The result of a nested command is passed up into the higher-level command, which is then processed.

```
set listCells [lsort [get_cells]]
```

The preceding example assigns the sorted list of cell objects at the top-level of the current design to the `$listCells` variable. The `get_cells` command is executed first, the returned objects are sorted by the `lsort` command, and the sorted list is assigned to the specified variable.

However, the Vivado Design Suite handles square brackets differently than standard Tcl. Square brackets are treated as standard characters in Verilog and VHDL, and usually identify one or more elements of vectors, such as busses or arrays of instances. In the Vivado tool the square brackets are not evaluated in a bottom-up manner when they are expected to be part of a netlist object name.

The following three commands are equivalent:

```
1.) set list_of_pins [get_pins transformLoop[0].ct/xOutReg_reg/CARRYOUT[*] ]
2.) set list_of_pins [get_pins {transformLoop[0].ct/xOutReg_reg/CARRYOUT[*] } ]
3.) set list_of_pins [get_pins transformLoop\[0\].ct/xOutReg_reg/CARRYOUT\[*\] ]
```

In line 1, the outer pair of brackets indicate a nested command, `[get_pins]`, as is standard in Tcl. However, the subsequent square brackets are interpreted by the Vivado tool as part of the specified object name `transformLoop[0]`. This is handled automatically by the Vivado Design Suite, but is limited to certain characters, or else the brackets will be interpreted as is standard in Tcl:

- star: [*] - The wildcard indicates any of a number of bits or instances.
- integer: [12] - The integer indicates a specific bit or instance.
- vector: [31:0] - The vector indicates a specific range of bits, or group of instances.

In line 2 the use of the braces, {}, will prevent command substitution of the string inside the braces. In this case the square brackets would be evaluated as part of the object name, `transformLoop[0]`.

In line 3, the backslash indicates that the bracket should be interpreted as a standard character rather than a special character, and this will prevent nested command substitution.

While lines 2 and 3 prevent the square brackets from being misinterpreted, those lines require you to manually apply the braces or backslash as needed by standard Tcl. Line 1 shows how the Vivado Design Suite automatically handles this for you.

Finally, to add comments to a Tcl script, simply start a new-line with the number sign, or hash character, '#'. Characters that follow a number sign are ignored, up to the next new-line. To add a comment to the end of a line, simply end the command with a semicolon, ';', and then begin the comment with a number sign as shown below:

```
# This is a comment
puts "This is a command"; # followed by a comment
```

Example Script: Non-Project Compilation Flow

The following is an example Tcl script that defines a Non-Project design flow.

The example script uses a custom command `reportCriticalPaths`. This is an illustration on how the Vivado Design Suite can be augmented with custom commands and procedures. The content of `reportCriticalPaths` is provided and explained in the section Defining Tcl Procedures, page 11.

```
# STEP#1: define the output directory area.
#
set outputDir ./Tutorial_Created_Data/cpu_output
file mkdir $outputDir
#
# STEP#2: setup design sources and constraints
#
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhdl
read_verilog  [ glob ./Sources/hdl/*.v ]
read_verilog  [ glob ./Sources/hdl/mgt/*.v ]
read_verilog  [ glob ./Sources/hdl/or1200/*.v ]
read_verilog  [ glob ./Sources/hdl/usbf/*.v ]
read_verilog  [ glob ./Sources/hdl/wb_conmax/*.v ]
read_xdc ./Sources/top_full.xdc
#
# STEP#3: run synthesis, write design checkpoint, report timing,
# and utilization estimates
#
synth_design -top top -part xc7k70tfbg676-2
write_checkpoint -force $outputDir/post_synth.dcp
report_timing_summary -file $outputDir/post_synth_timing_summary.rpt
report_utilization -file $outputDir/post_synth_util.rpt
#
# Run custom script to report critical timing paths
reportCriticalPaths post_synth_critpath_report.csv
#
# STEP#4: run logic optimization, placement and physical logic optimization,
# write design checkpoint, report utilization and timing estimates
#
opt_design
reportCriticalPaths post_opt_critpath_report.csv
place_design
report_clock_utilization -file $outputDir/clock_util.rpt
#
# Optionally run optimization if there are timing violations after placement
if {[get_property SLACK [get_timing_paths -max_paths 1 -nworst 1 -setup]] < 0}
{
    puts "Found setup timing violations => running physical optimization"
    phys_opt_design
}
write_checkpoint -force $outputDir/post_place.dcp
report_utilization -file $outputDir/post_place_util.rpt
report_timing_summary -file $outputDir/post_place_timing_summary.rpt
#
# STEP#5: run the router, write the post-route design checkpoint, report the routing
# status, report timing, power, and DRC, and finally save the Verilog netlist.
#
route_design
write_checkpoint -force $outputDir/post_route.dcp
report_route_status -file $outputDir/post_route_status.rpt
report_timing_summary -file $outputDir/post_route_timing_summary.rpt
report_power -file $outputDir/post_route_power.rpt
report_drc -file $outputDir/post_imp_drc.rpt
write_verilog -force $outputDir/cpu_impl_netlist.v -mode timesim -sdf_anno true
#
# STEP#6: generate a bitstream
#
write_bitstream -force $outputDir/cpu.bit
```

Details of the Sample Script

The key steps of the preceding script can be broken down as follows:

- **Step 1:** defines a variable, `$outputDir`, that points to an output directory and also physically creates the directory. The `$outputDir` variable is referenced as needed at other points in the script.
- **Step 2:** reads the VHDL and Verilog files that contain the design description, and the XDC file that contains the physical and/or timing constraints for the design. You can also read synthesized netlists using the read_edif command.

  The Vivado Design Suite uses design constraints to define requirements for both the physical and timing characteristics of the design. The `read_xdc` command reads an XDC constraints file which will be used during synthesis and implementation.

IMPORTANT: *The Vivado Design Suite does not support the UCF format. For information on migrating UCF constraints to XDC commands refer to the Vivado Design Suite Migration Methodology Guide (UG911) for more information.*

  The `read_*` Tcl commands are designed for use with the Non-Project Mode, as it allows a file on the disk to be read by the Vivado Design Suite to build an in-memory design database, without copying the file or creating a dependency on the file in any way, as it would in Project Mode. All actions taken in the Non-Project Mode are directed at the in-memory database within the Vivado tools. The advantages of this approach make the Non-Project Mode extremely flexible with regard to design. However, a limitation of the Non-Project Mode is that you must monitor any changes to the source design files, and update the design as needed. For more information on running the Vivado Design Suite using either Project Mode or Non-Project Mode, refer to the *Vivado Design Suite User Guide: Design Flows Overview (UG892)*.

- **Step 3:** synthesizes the design on a specific package.

  This step compiles the HDL design files, applies the timing constraints located in the XDC file, and maps the logic onto Xilinx primitives to create a design database in memory. The in-memory design resides in the Vivado tool, whether running in batch mode, Tcl shell mode for interactive Tcl commands, or in the Vivado Integrated Design Environment (IDE) for interaction with the design data in a graphical form.

  Once synthesis is done, a checkpoint is saved for reference. At this point the design is an unplaced synthesized netlist with timing and physical constraints. Various reports like timing and utilization can provide a useful resource to better understand the challenges of the design.

  This sample script uses a custom command, `reportCriticalPaths`, to report the TNS/WNS/Violators into a CSV file. This provides the ability for you to quickly identify which paths are critical.

Any additional XDC file read in after synthesis by the `read_xdc` or `source` commands will be used during the implementation steps only. They will be stored in any subsequent design checkpoints, along with the netlist.

- **Step 4:** performs pre-placement logic optimization, in preparation for placement and routing. The objective of optimization is to simplify the logic design before committing to physical resources on the target part. Optimization is followed by timing-driven placement with the Vivado placer.

  After each of those steps, the `reportCriticalPaths` command is run to generate a new CSV file. Having multiple CSV files from different stages of the design lets you create a custom timing summary spreadsheet that can help visualizing how timing improves during each implementation step.

  Once the placement is done, the script uses the `get_timing_paths` command to examine the SLACK property of the worst timing path in the placed design. While the `report_timing` command returns a detailed text report of the timing path with the worst slack, the `get_timing_paths` command returns the same timing path as a Tcl object with properties that correspond to the main timing characteristics of the path. The SLACK property returns the slack of the specified timing path, or worst path in this case. If the slack is negative then the script runs physical optimization to resolve the placement timing violations whenever possible.

  At the very end of Step 4, another checkpoint is saved and the device utilization is reported along with a timing summary of the design. This will let you compare pre-routed and post-routed timing to assess the impact that routing has on the design timing.

- **Step 5:** The Vivado router performs timing-driven routing, and a checkpoint is saved for reference. Now that the in-memory design is routed, additional reports provide critical information regarding power consumption, design rule violations, and final timing. You can output reports to files, for later review, or you can direct the reports to the Vivado IDE for more interactive examination. A Verilog netlist is exported, for use in timing simulation.
- **Step 6:** writes a bitstream to test and program the design onto the Xilinx FPGA.

Defining Tcl Procedures

Because the Vivado Design Suite provides a full Tcl interpreter built into the tool, creating new custom commands and procedures is a simple task. You can write Tcl scripts to be loaded and run from within the Vivado IDE, or you can write procedures (or procs), to act as design commands, taking arguments, checking for errors, and returning results.

A Tcl procedure is defined with the `proc` command which takes three arguments: the procedure name, the list of arguments, and the body of code to execute. The following code provides a simple example of a procedure definition:

```
proc helloProc { arg1 } {
  # This is a comment inside the body of the procedure
  puts "Hello World! Arg1 is $arg1"
}
```

TIP: *Although the curly braces are optional for the definition of this procedure, since helloProc has only one argument, it makes the procedure definition easier to read by enclosing the arguments in braces, and is required when the procedure accepts multiple arguments.*

A procedure usually has predefined arguments with optional default values. It returns an empty list unless the `return` command is used to return a different value. The following example defines a procedure, `reportWorstViolations`, with three predefined arguments:

```
proc reportWorstViolations { nbrPaths corner delayType } {
  report_timing -max_paths $nbrPaths -corner $corner -delay_type $delayType -nworst 1
}
```

When running the procedure, all of the arguments are required to complete the command, as shown:

```
%> reportWorstViolations 2 Slow max
%> reportWorstViolations 10 Fast min
```

The following is a different form of the same procedure, where two of the three arguments have a default value. The default value for `corner` is Slow, and the default value for `delayType` is Max. With default values provided, the corner and delayType arguments are optional when calling the procedure.

```
proc reportWorstViolations { nbrPaths { corner Slow } { delayType Max } } {
  report_timing -max_paths $nbrPaths -corner $corner -delay_type $delayType -nworst 1
}
```

When running this procedure, any of the following forms of the command will work:

```
%> reportWorstViolations 2
%> reportWorstViolations 10 Fast
%> reportWorstViolations 10 Slow Min
```

The following is an example of the procedure that has one predefined argument, `nbrPath`, but also accepts any number of additional arguments. This uses the Tcl keyword `args` in the list of arguments when defining the procedure. The `args` keyword indicates a Tcl list that can have any number of elements, including none.

```
proc reportWorstViolations { nbrPaths args } {
  eval report_timing -max_paths $nbrPaths $args
}
```

When executing Tcl commands, you can use variable substitution to replace some of the command line arguments accepted or required by the Tcl command. In this case, you must use the Tcl `eval` command to evaluate the command line with the Tcl variable as part of the command. In the preceding example, the variable list of arguments (`$args`) is passed to the encapsulated `report_timing` command as a variable, and so requires the use of the `eval` command.

When running this procedure, any of the following forms of the command will work:

```
%> reportWorstViolations 2
%> reportWorstViolations 1 -to [get_ports]
%> reportWorstViolations 10 -delay_type min_max -nworst 2
```

In the first example, the number 2 is passed to the `$nbrPaths` argument, and applied to `-max_paths`. In the second and third examples, the numbers 1 and 10 respectively are applied to `-max_paths`, and all the subsequent characters are assigned to `$args`.

The following example provides the procedure definition for the `reportCriticalPaths` command that was previously used in the Non-Project Mode example script. The procedure takes a single argument, `$filename`, and has been commented to explain each section:

```
#-------------------------------------------------------------------------
# reportCriticalPaths
#-------------------------------------------------------------------------
# This function generates a CSV file that provides a summary of the first
# 50 violations for both Setup and Hold analysis. So a maximum number of
# 100 paths are reported.
#-------------------------------------------------------------------------
proc reportCriticalPaths { fileName } {
  # Open the specified output file in write mode
  set FH [open $fileName w]

  # Write the current date and CSV format to a file header
  puts $FH "#\n# File created on [clock format [clock seconds]]\n#\n"
  puts $FH "Startpoint,Endpoint,DelayType,Slack,#Levels,#LUTs"

  # Iterate through both Min and Max delay types
  foreach delayType {max min} {
    # Collect details from the 50 worst timing paths for the current analysis
    # (max = setup/recovery, min = hold/removal)
    # The $path variable contains a Timing Path object.
    foreach path [get_timing_paths -delay_type $delayType -max_paths 50 -nworst 1] {
      # Get the LUT cells of the timing paths
      set luts [get_cells -filter {REF_NAME =~ LUT*} -of_object $path]
```

```
      # Get the startpoint of the Timing Path object
      set startpoint [get_property STARTPOINT_PIN $path]
      # Get the endpoint of the Timing Path object
      set endpoint [get_property ENDPOINT_PIN $path]
      # Get the slack on the Timing Path object
      set slack [get_property SLACK $path]
      # Get the number of logic levels between startpoint and endpoint
      set levels [get_property LOGIC_LEVELS $path]

      # Save the collected path details to the CSV file
      puts $FH "$startpoint,$endpoint,$delayType,$slack,$levels,[llength $luts]"
    }
  }
  # Close the output file
  close $FH
  puts "CSV file $fileName has been created.\n"
  return 0
}; # End PROC
```

Accessing Design Objects

The Vivado Design Suite loads the project, design, and device information into an in-memory database, which is used by synthesis, implementation, timing analysis, and to generate a bitstream. The database is the same for project and non-project flows. The database is updated as you step through the FPGA design flow. You can write the database contents out to disk as a checkpoint file (.dcp) at any point of the design flow. Using Tcl commands in the Vivado tool lets you interact with the design database, query Tcl objects, read or set their properties, and use them in Tcl scripts for various purposes. It is very helpful to understand the content of the database, to understand how efficient scripts can be written around it.

The Vivado Design Suite Tcl interpreter provides access to many first class objects such as project, device, nets, cells, and pins. The Vivado Design Suite updates these design objects dynamically, as the design progresses, and loads them into the in-memory database in both Project and Non-Project modes.

You can interactively query design objects, analyze the state of your project, write a script to access the in-memory design, and run custom reports or execute optional design flow steps. Each object comes with a number of properties that can always be read and sometimes written. Most design objects are related to other design objects, allowing you to traverse the design to find related objects or information.

You can query design objects using the `get_*` Tcl commands which return list of design objects, that can be directly manipulated, or assigned to a Tcl variable. Caching objects in variables can save runtime by reducing the number of queries to the design database. Querying the list of nets or pins can be a time consuming process, so saving the results can speed the design flow when accessing the information repeatedly. See Caching Objects, page 34 for more on this topic.

Each class of design object (net, pin, port, ...) has a unique set of standard properties that can be read and sometimes written to modify their value in the database. In addition, the design attributes specified in the RTL source files, the Verilog parameters and VHDL generics are stored with the associated netlist object as properties. For example, a port object has a property that indicates its direction, while a net object has a property that defines its fanout. The Vivado tool provides a number of commands for adding, changing, and reporting these properties. Using the `get_* -filter` option lets you get a list of design objects that is filtered, or reduced, to match specific property values, as described in Filtering Results, page 18.

There are two properties that are common to all objects: NAME and CLASS. When an object is assigned to a variable, a pointer to the object is stored in the variable. Objects can be passed by variable to other Tcl commands or Tcl procs. However, when a variable containing a design object is passed to a Tcl command requiring a string, the object's NAME property is passed instead of the object itself. This is why the Vivado Tcl console displays the names of objects returned by commands that only return design objects. The example below shows a timing path object assigned to the variable, `$path1`, and the results of the `puts` command and the `report_property` command on the specified variable. Notice that the `puts` command just prints the object NAME, while the `report_property` command returns all of the object properties and their values:

```
set path1 [get_timing_paths -delay_type max]
{usbEngine1/u4/inta_reg/C -->
cpuEngine/or1200_du/tbar_ram/ramb16_s36_s36/DIBDI[12]}

puts $path1
{usbEngine1/u4/inta_reg/C -->
cpuEngine/or1200_du/tbar_ram/ramb16_s36_s36/DIBDI[12]}

report_property -all $path1
Property           Type    Read-only  Visible  Value
CLASS              string  true       true     timing_path
DATAPATH_DELAY     double  true       true     7.972
DELAY_TYPE         string  true       true     max
ENDPOINT_CLOCK     clock   true       true     cpuClk
ENDPOINT_PIN       pin     true       true
cpuEngine/or1200_du/tbar_ram/ramb16_s36_s36/DIBDI[12]
GROUP              string  true       true     cpuClk
LOGIC_LEVELS       int     true       true     11
NAME               string  true       true     {usbEngine1/u4/inta_reg/C -->
cpuEngine/or1200_du/tbar_ram/ramb16_s36_s36/DIBDI[12]}
REQUIREMENT        double  true       true     10.000
SKEW               double  true       true     -0.010
SLACK              double  true       true     1.816
STARTPOINT_CLOCK   clock   true       true     usbClk
STARTPOINT_PIN     pin     true       true     usbEngine1/u4/inta_reg/C
UNCERTAINTY        double  true       true     -0.202
```

You can also create custom properties for any class of design objects in the Vivado Design Suite. This can be useful when you would like to annotate some information from a script onto design objects. The following example creates a property, SELECTED, on a cell objects. The value of the property is defined as an integer.

```
create_property SELECTED cell -type int
```

Once a property has been created on a class of objects, it can be managed with `set_property` and `get_property` commands, and reported with `list_property` and `report_property` commands. The following example sets the SELECTED property to a value of `1` on all the cells that match the specified name pattern, `*aurora_64b66b*`:

```
set_property SELECTED 1 [get_cells -hier *aurora_64b66b*]
```

Getting By Name – Traversing the Design Hierarchy

Most designs are made up of a series of blocks or modules connected in some hierarchical fashion. Whether the design is crafted from the bottom-up, or the top-down, or from the middle out, searching the design hierarchy to locate a specific object is a common task.

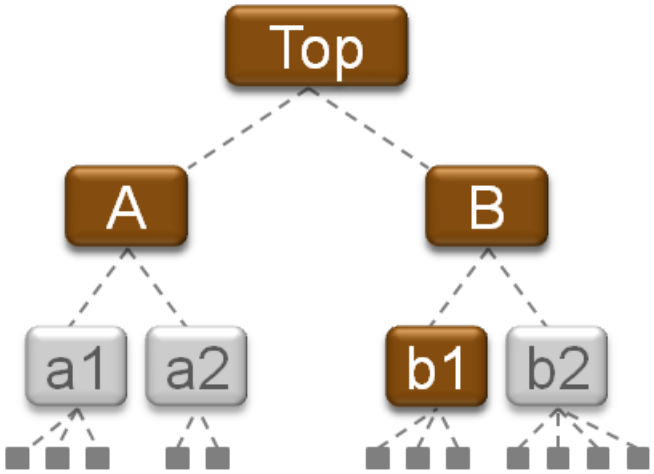

Figure 1-1: **Searching the Design Hierarchy**

By default, the `get_*` commands only return objects from the top-level of the design. You can use the `current_instance` command before using the `get_*` commands to scope the design object queries to a specific hierarchical instance of the design. To move the scope back to the top-level of the design, you simply have to use the `current_instance` command with no argument.

As an example, Figure 1-1 shows a hierarchical design where the modules A and B are instantiated at the top-level. Module A includes the a1 and a2 hierarchical instances, and module B includes the b1 and b2 hierarchical instances. Each of a1, a2, b1, and b2 has leaf cells inside, as indicated in the figure.

```
# Set the current instance of the design to module B.
current_instance B
get_cells * ; # Returns b1 and b2, cells found in the level of the current instance.
get_nets * ; # Returns nets from module B, the current instance.
# Reset the current instance to the top-level of the design.
current_instance
get_cells * ; # Returns A and B, located at the top-level of the design.
```

Although the `get_*` commands only search the top-level, or the level of the current instance, you can specify a search pattern that includes a hierarchical instance name relative to the current instance. By default, the current instance is set to the top-level of the design. To query the instance b1 from the top-level, you can specify the following name pattern:

```
get_cells B/b1 ; # Search the top-level for an instance with a hierarchical name.
```

Using the -hierarchical option

While the default behavior is to search for objects only at the level of the current instance, many of the `get_*` commands have a `-hierarchical` option to enable searching the design hierarchy level by level, starting from the level of the current instance.

```
get_cells -hierarchical * ; # Returns all cells in the hierarchy.
get_nets -hier *nt* ; # Returns all hierarchical nets that match *nt*.
```

However, one feature of the `-hierarchical` option is that the Vivado tool tries to match the specified name pattern at each level of the design hierarchy, and not against the full hierarchical name of an object. The name pattern **should not** include the hierarchical separator in this case, as shown in the following example, based on Figure 1-1, page 15:

```
get_cells -hierarchical B/* ; # No cell is returned.
get_cells -hierarchical b* ; # B/b1 and B/b2 are returned.
```

IMPORTANT: *When -hierarchical is used with -regexp, the specified search string is matched against the full hierarchical name, and B/.* will return all cell names that match this pattern. See the Vivado Design Suite Tcl Command Reference Guide (UG835) for more information on -regexp.*

The `-hierarchical` search is equivalent to manually performing a search at each level of the hierarchy, using the `current_instance` command to set the level of hierarchy, and returning all the objects on the current level that match the specified name pattern. The following Tcl code, based on Figure 1-1, page 15, illustrates this manual process:

```
set result {}
foreach hcell [list "" A B A/a1 A/a2 B/b1 B/b2] {
  current_instance $hcell ;# Move scope to $hcell
  set result [concat $result [get_cells <pattern>]]
  current_instance ;# Return scope to design top-level
}
```

Searching for Pins

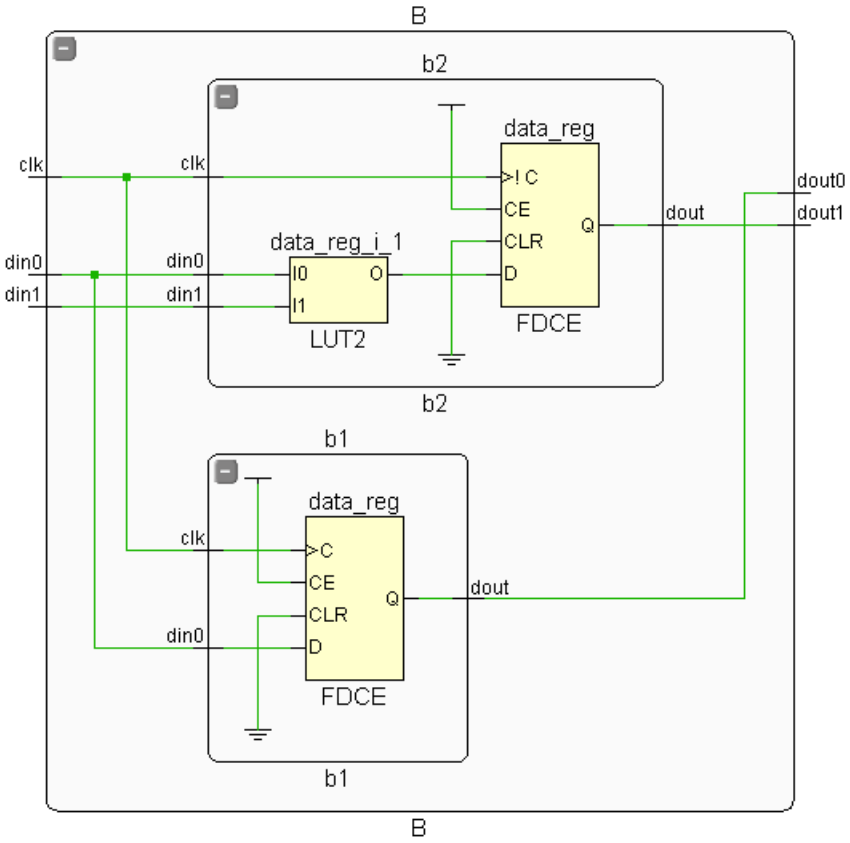

Figure 1-2: **Searching for Pin Names**

The names of pins are based on the instance they belong to. When searching for pins, you must use the hierarchy separator character to separate the instance name and the pin name patterns. The following examples are illustrated by Figure 1-2.

```
# Current instance is set to design top-level
get_pins B/* ; # Returns B/clk B/din0 B/din1 B/dout0 B/dout1
get_pins B/b2/*/O ; # Returns B/b2/data_reg_i_1/O
current_instance B/b2 ; # Change scope to B/b2
get_pins *_reg/D ; # Returns B/b2/data_reg/D
```

You can also use the `-hierarchical` option when searching for pins:

```
current_instance ; # Reset to the top-level of the hierarchy
get_pins -hier */D # Returns pin objects for all D pins in the design(1)
```

1. `-hierarchical` and `-hier` refer to the same option. The Tcl shell will complete option names automatically if enough characters are provided to identify a unique option. Therefore `-of_object` and `-of` can also refer to the same option.

Filtering Results

More often than not, when you using `get_*` to search for design objects, you are only interested in a subset of the objects returned. You might not want all of the netlist objects in the design, but only cells of a certain type, or nets of a certain name. In some cases, only a subset of elements are of interest and need to be returned.

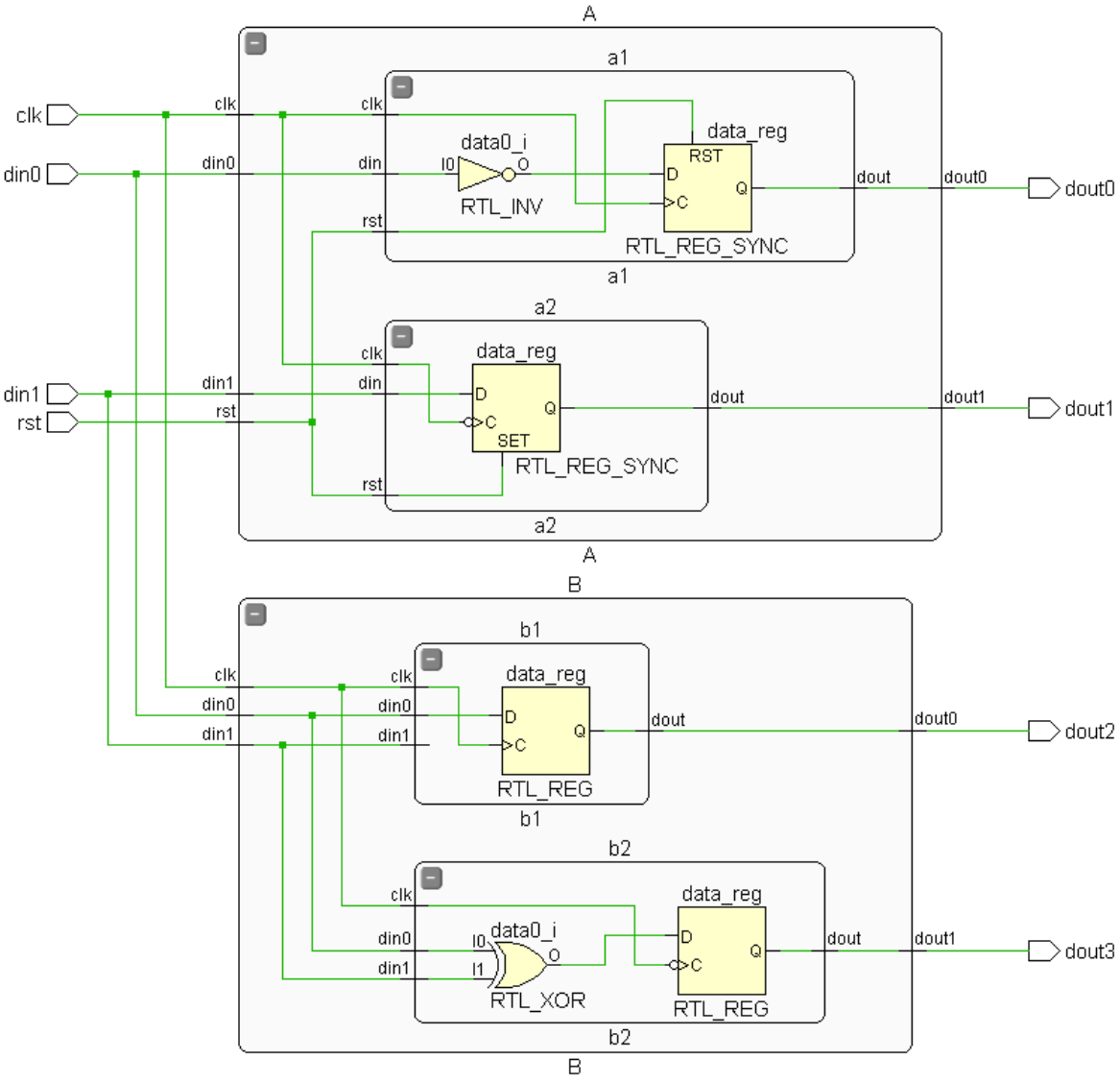

Figure 1-3: **Searching Hierarchical Designs**

You can limit the search results, by narrowly defining your search pattern, using wildcards '*' and '?', or using -regexp to build complex search patterns to evaluate. You can limit the scope of hierarchy being searched, by setting the `current_instance` or by specifying `-hierarchy`.

For example, using the design shown in Figure 1-3, page 18, the following expressions return different results:

```
get_cells * ; # Returns 2 cells
get_cells -hier * ; # Returns 12 cells
get_cells -hier * -filter {NAME =~ */?1/*} ; # Returns 3 cells
```

The `-filter` option lets you filter the results of the `get_*` commands, based on specific properties of the objects being searched. For example, the following command returns all the cells that have B/b* in their full hierarchical name, and that have not been placed by the user, so that IS_LOC_FIXED is FALSE, or 0:

```
set unLoced [ get_cells -hier -filter {NAME =~ B/b* && !IS_LOC_FIXED} ]
```

The `-filter` option causes Vivado to filter the results of a query before it is returned. However, in some cases you may have assigned the results of a prior search to a variable, that is now stored in memory. The `filter` command lets you filter the content of any list of objects, including lists stored in a variable. Using the results of the prior example, stored in `$unLoced`, you can further filter the list of objects as follows:

```
filter $unLoced {IS_PRIMITIVE}
```

The preceding example filters the stored results of the prior search, filtering the list to return only the objects that are primitive instances in the design.

TIP: *Note the direct use of the boolean properties !IS_LOC_FIXED and IS_PRIMITIVE in the example above. Boolean (bool) type properties can be directly evaluated in filter expressions as true or false.*

The specific operators that can be used in filter expressions are "equals" and "not-equals" (== and !=), and "contains" and "not-contains" (=~ and !~). Numeric comparison operators <, >, <=, and >= can also be used. Multiple filter expressions can be joined by AND and OR (&& and ||).

Getting Objects by Relationship

There are times when you will need to find objects that are related to other objects in the design. For instance, selecting all of the nets connected to the pins of a specific cell, or all of the cells connected to a specific net. The Vivado Design Suite provides the ability to traverse the elements of the design through their various relationships to one another. This is accomplished through the use of the `-of_objects` option supported by many of the `get_*` commands. Figure 1-4, page 20 illustrates the relationship of objects in the in-memory design.

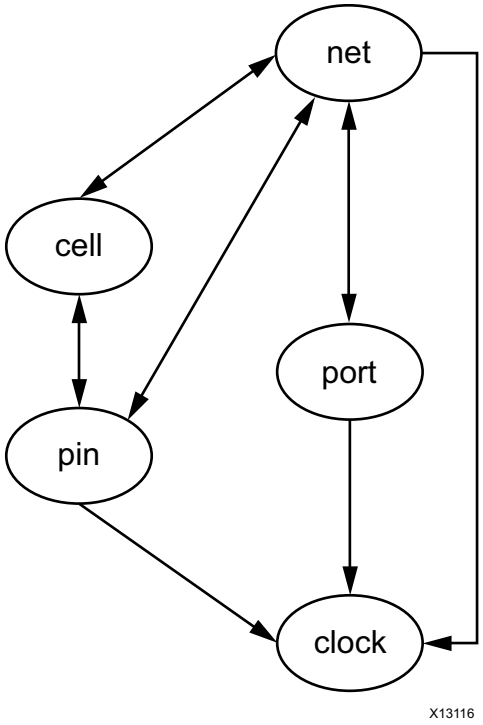

Figure 1-4: **Vivado Design Suite - Object Relationships**

Note: This image is intended to be representative, and not a complete map of objects and relationships in the Vivado Design Suite database.

The help text for each of the `get_*` commands that supports the `-of_objects` command lists the related objects that can be traversed:

```
get_cells -of_objects {pins, timing paths, nets, bels or sites}
get_clocks -of_objects {nets, ports, or pins}
get_nets -of_objects {pins, ports, cells, timing paths or clocks}
get_pins -of_objects {cells, nets, bel pins, timing paths or clocks}
get_ports -of_objects {nets, instances, sites, clocks, timing paths, io standards, io
banks, package pins}
```

With the `-of_objects` option, getting the list of all pin objects attached to a list of net objects becomes very simple:

```
get_pins -of_objects [get_nets -hier]
```

To only get the list of drivers for those nets you just need to use the `-filter` option:

```
get_pins -of [get_nets -hier] -filter {DIRECTION == OUT}
```

You can also get the list of pins from a list of cells, or a list of cells from a list of nets and so on.

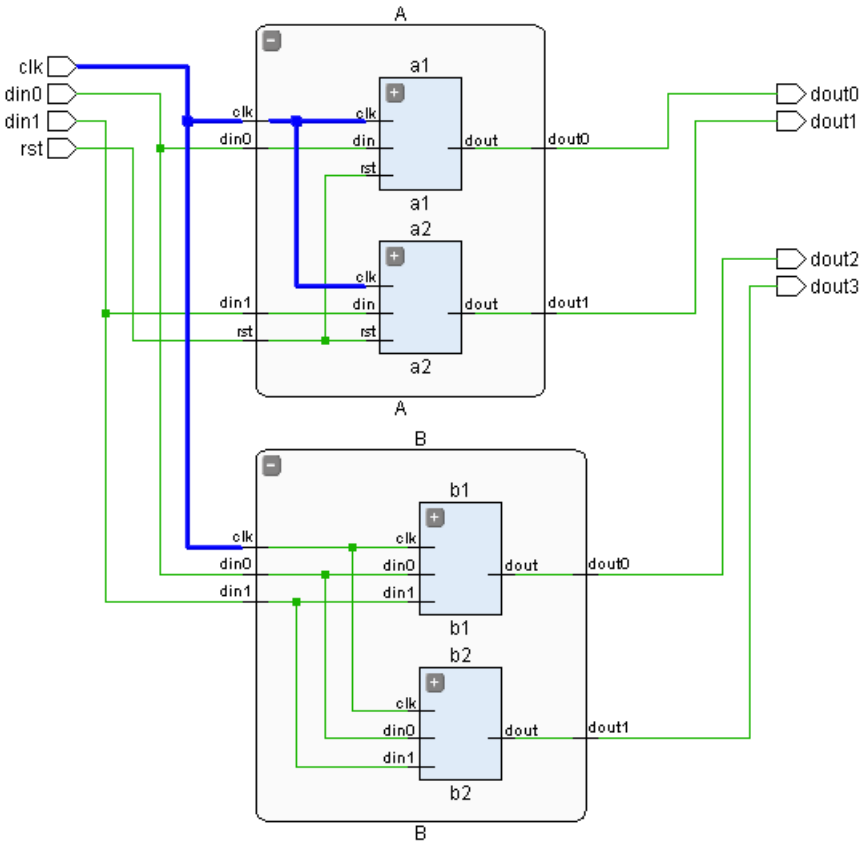

Figure 1-5: **Traversing Objects by Relationship**

Using Figure 1-5 as an illustration, the following example gets the clock pin from instance a1, then works its way outward and upward through the hierarchy. It gets the net connected to that pin, and gets the pins connected to that net, then gets the nets connected to those pins, and finally gets the pins connected to those nets.

```
get_pins -of [get_nets -of [get_pins -of [get_nets -of [get_pins A/a1/clk]]]]
A/a2/clk A/clk A/a1/clk B/clk
```

Notice that the last `get_pins` command returns the clock pin of hierarchical module B, `B/clk`, along with the other pins that have already been returned. However, to cross the hierarchy, and return the primitive pins on the clock net object, you can use the `-leaf` option of the `get_pins` command. The following example shows what is returned when `-leaf` is used:

```
get_pins -leaf -of [get_nets -of [get_pins -of [get_nets -of [get_pins A/a1/clk]]]]
B/b1/data_reg/C A/a2/data_reg/C A/a1/data_reg/C B/b2/data_reg/C
```

Handling Lists of Objects

When using the `get_*` commands, the returned object looks and acts like a standard Tcl list. However, the Vivado Design Suite is returning a container of a single class of objects (e.g. cells, nets, pins, or ports), which is not a generic Tcl list. However, for most of your purposes, the container of objects looks and acts just like any Tcl list. The container of objects is handled automatically by the Vivado Design Suite, and is totally transparent to the user. For example, the standard Tcl `llength` command can be used on a container of objects (e.g. from `get_cells`) and returns the number of elements in the container, like it would for any standard Tcl list.

The built-in Tcl commands that handle lists in the Vivado Design Suite have been overloaded and enhanced to fully support objects and containers of objects. For example, `lsort`, `lappend`, `lindex`, and `llength`, have been enhanced to manage the container based on the NAME property of the object. The result of these commands, when passed a container of objects, returns a container of objects.

For example, `lsort` will sort a container of cells from the `get_cells` command based on the hierarchical names of the objects.

You can add new objects to the container (using `lappend` for instance), but you can only add the same type of object that is currently in the container. Adding a different type of object, or string, to the list is not permitted and will result in a Tcl error.

The following example shows a Vivado container being sorted in a descending order, and each object put onto a separate line, by using the `puts` command and a `foreach` loop:

```
foreach X [lsort -decreasing [get_cells]] {puts $X}
wbArbEngine
usb_vbus_pad_1_i_IBUF_inst
usb_vbus_pad_0_i_IBUF_inst
usbEngine1
usbEngine0
...
```

Redirecting Output

Many of the Vivado Design Suite Tcl commands allow you to redirect the information returned by the command to a file with the `-file` option, for printing or processing outside of the tool; or as a string that can be saved in a variable with the `-return_string` option for further processing within the Vivado tool.

All of the report commands support the `-file` option. File output is useful for report commands that output a great deal of information, that may require further review, or support the documentation of a design project, or to pass to downstream processes such as other design disciplines, or other departments. Some of the commands supporting file output include:

```
report_datasheet
report_drc
report_power
report_timing
report_timing_summary
report_utilization
```

For example, to save the results of the `report_timing` command to a file:

```
report_timing -delay_type max -file setup_violations.rpt
report_timing -delay_type min -file hold_violations.rpt
```

A relative or absolute path can be specified as part of the file name. A relative path is relative to the directory from which the Vivado tool has been started, or to the current working directory which can be retrieved with the `pwd` command.

TIP: *If the path is not specified as part of the file name, the file will be written into the current working directory, or the directory from which the Vivado tool was launched.*

To append the content from a command to an existing file, use the `-append` option in addition to `-file`. For example, the code below creates one file, `all_violations.rpt`, that combines the output of two separate commands:

```
report_timing -delay_type max -file all_violations.rpt
report_timing -delay_type min -file -append all_violations.rpt
```

After the file has been created, you may have need to access the file from the file system, opening the file to read from it, or to write to it. The Tcl shell offers a number of commands for accessing files. See Accessing Files, page 24 for more information.

Many `report_*` commands also support the `-return_string` option. This option directs the command to return its output as a string that can be assigned to a Tcl variable. Assigning output to string variables is useful for further processing within the Tcl script, to allow extraction of key information to enable flow control, branching, and to set other variables for use in the script.

Some of the commands that support `-return_string` are:

```
report_clocks
report_clock_interaction
report_disable_timing
report_environment
report_high_fanout_nets
report_operating_conditions
report_power
report_property
report_pulse_width
report_route_status
report_utilization
```

You can split the returned string from the report on the newline character, \n, to process the string line by line as a list:

```
set timeLines [split [report_timing -return_string -max_paths 10] \n ]
```

The Tcl shell also offers many tool for working with strings. See Working with Strings, page 27 for more information.

Accessing Files

After a file has been written to the file system, the Tcl language provides many useful commands for working with the files. You can extract elements of a file, such as the file path, file name and file extension. Some of the Tcl commands available to examine information about file include:

- `file exists` *`filename`* - A boolean test that returns 1 if *`filename`* exists, and you have permission to read its location; returns 0 otherwise. Use this to determine if the file you are looking for already exists.
- `file type` *`filename`* - Returns the type of filename as a string with one of the following values: file, directory, characterSpecial, blockSpecial, fifo, link, or socket.
- `file dirname` *`filename`* - This returns the directory structure of the *`fileName`*, up to but not including the last slash.
- `file rootname` *`filename`* - Returns all the characters in *`filename`* up to but not including last dot.
- `file tail` *`filename`* - Return all characters in *`filename`* after the last slash.
- `file extension` *`filename`* - Returns all characters in *`filename`* after, and including, the last dot.

The following examples illustrates some of the available Tcl commands:

```
set filePath {C:/Data/carry_chain.txt}
file dirname $filePath ; # Returns C:/Data
file tail $filePath ; # Returns carry_chain.txt
file extension $filePath ; # Returns .txt
```

After the Vivado tool has created a file, through a `report_*` command, or `write_*` command, you can open the file from within a Tcl script to read its contents, or to write additional content. To open, read from, write to, and close a file, you can use some of the following Tcl commands:

- `open <filename> [access] [perms]` - Opens the `filename` and returns the file handle, or *fileID*, used to access the file. It is a standard practice to capture the *fileID* in a Tcl variable, so that you can refer to the file handle when needed. The file permissions of a new file are set to the conjunction of *perms* and the process umask. The *access* mode determines whether you can read to or write to the open file. Some common access modes are:
  - r - Read only. The file must exist; it will not be created. This is the default access mode if one is not specified.
  - w - Write only. This will create the specified file if it does not already exist. The data is written to the front of the file, truncating or overwriting the content of an existing file.
  - a - Append only. This will create the specified file if it does not already exist. The data is written at the end of the file, appending to any existing file content.
- `read [-nonewline] fileId` - Read all remaining bytes from *fileId*, optionally discarding the last character if it is a newline, `\n`. When used in this form right after opening the file, the `read` command will read the entire file at once.
- `read fileId numBytes` - Read the specified number of bytes, *numBytes*, from *fileId*. Use this form of the command to read blocks of the file, up to the end of the file.
- `eof fileId` - Returns 1 if an end-of-file (EOF) has occurred on *fileId*, 0 otherwise.
- `gets fileId [varName]` - Read the next line from *fileId*, discarding the newline character. Places the characters of line in `$varName` if given, otherwise returns them to the command shell. The following are different forms of the gets commands:

```
gets $fileHandle
Append line 4 of file.
gets $fileHandle line
28
puts $line
Append line 5 of file.
set line [gets $fileHandle]
Append line 6 of file.
puts $line
Append line 6 of file.
```

  In the preceding example, `$fileHandle` is the file handle returned when the file was opened. The first line uses the simple form of `gets`, without specifying a Tcl variable to capture the output. In this case the output is returned to `stdout`. The second form of the command assigns the output to a variable called `$line`, and the `gets` command returns the number of characters it has read, 28.

Note: You can assign the return of the `gets` command to a Tcl variable, but depending on which form you are using you may capture the contents of the file, or the number of characters the gets command has read.

- `puts [-nonewline] [fileId] string` - Write a string to the specified *`fileId`*, optionally omitting the newline character, `\n`. The default *`fileId`* for the `puts` command is `stdout`.
- `close fileId` - Close the open file channel *`fileId`*. It is very important to close any files your Tcl scripts have opened, or you may develop memory leaks in the Vivado application, or encounter other undesirable effects.

The following example opens a file in read access mode, storing the file handle as `$FH`, reads the contents of the file in a single operation, assigning it to `$content`, and splits the contents into a Tcl list. The file is closed upon completion.

```
set FH [open C:/Data/carry_chains.txt r]
set content [read $FH]; # The entire file content is saved to $content
foreach line [split $content \n] {
  # The current line is saved inside $line variable
  puts $line
}
close $FH
```

Note: It is not recommended to read large files in a single operation due to performance and memory considerations.

Rather than reading the entire file at once, and then parsing the results, the following example reads the file line-by-line until the end of the file has been reached, and writes the line number and line content to `stdout`. The file is closed upon completion:

```
set FH [open C:/Data/carry_chains.txt r]
set i 1
while {![eof $FH]} {
  # Read a line from the file, and assign it to the $line variable
  set line [gets $FH]
    puts "Line $i: $line"
    incr i
}
close $FH
```

The example below writes to a file, `ports.rpt`, saving all the I/O ports from the design, with the port direction, sorted by name:

```
set FH [open C:/Data/ports.rpt w]
foreach port [lsort [get_ports *]] {
  puts $FH [format "%-18s %-4s" $port [get_property DIRECTION $port]]
}
close $FH
```

In the example above, the file is opened in write mode. Unlike read mode, the write mode will create the file if it does not exist, or overwrite the file if it does exist. To write new content to the end of an existing file, you should open the file in append mode instead.

Working with Strings

The `-return_string` argument directs the output of a `report_*` command to a Tcl string rather than to `stdout`. The string can be assigned to a Tcl variable, and parsed or otherwise processed.

```
set rpt [report_timing -return_string]
```

After the string has been assigned to a variable, the Tcl language provides many useful commands for processing the string in a number of ways:

- `append` *`string`* `[`*`arg1 arg2 ... argN`*`]` - Appends the specified *`args`* to the end of *`string`*.
- `format` *`formatString`* `[`*`arg1 arg2 ... argN`*`]` - Returns a formatted string generated to match the format specified by the *`formatString`* template. The template must be specified using % conversion specifiers as used in `sprintf`. The additional arguments, *`args`*, provide values to be substituted into the formatted string.
- `regexp [switches]` *`exp string`* - Returns 1 if the regular expression, *`exp`*, matches all or part of *`string`*, 0 otherwise. The `-nocase` switch can be specified to ignore character case when matching.
- `string match` *`pattern string`* - Returns 1 if the glob *`pattern`* matches *`string`*, 0 otherwise.
- `scan` *`string formatString`* `[`*`varName1 varName2 ...`*`]` - Extracts values from the specified `string` into variables, `varName`, applying the `formatString` using % conversion specifiers as in `sscanf` behavior. If no *`varNames`* are specified, `scan` returns the list of values to `stdout`.
- `string range` *`string first last`* - Returns the range of characters from *`string`* specified by character indices *`first`* through *`last`* inclusive.
- `string compare` *`string1 string2`* - Performs a lexicographical comparison of two strings, and returns -1 if *`string1`* comes before *`string2`*, 0 if they are the same, and 1 if *`string1`* comes after *`string2`*.
- `string last` *`string1 string2`* - Return the character index in *`string2`* of the last occurrence of *`string1`*. Returns -1 if *`string1`* is not found in *`string2`*.
- `string length` *`string`* - Returns the number of characters in *`string`*.

The following example assigns the results of the `report_timing` command to the `$report` Tcl variable, using `-return_string`. The string is processed to extract the start point, end point, path group and path type of each path. After the path information is extracted, a summary of that path is printed to the Tcl console.

```
# Capture return string of timing report, and assign variables
set report [report_timing -return_string -max_paths 10]
set startPoint {}
set endPoint {}
set pathGroup {}
set pathType {}

# Write the header for string output
puts [format "  %-12s %-12s %-20s -> %-20s" "Path Type" "Path Group" "Start Point" "End Point"]
puts [format "  %-12s %-12s %-20s -> %-20s" "---------" "----------" "-----------" "---------"]

# Split the return string into multiple lines to allow line by line processing
foreach line [split $report \n] {
if {[regexp -nocase -- {^\s*Source:\s*([^[:blank:]]+)((\s+\(?)|$)} $line - startPoint]} {
} elseif {[regexp -nocase -- {^\s*Destination:\s*([^[:blank:]]+)((\s+\(?)|$)} $line - endPoint]} {
} elseif {[regexp -nocase -- {^\s*Path Group:\s*([^[:blank:]]+)\s*$} $line - pathGroup]} {
} elseif {[regexp -nocase -- {^\s*Path Type:\s*([^[:blank:]]+)((\s+\(?)|$)} $line - pathType]} {
puts [format "  %-12s %-12s %-20s -> %-20s" $pathType $pathGroup $startPoint $endPoint]
}
}
```

An example output from the code is:

```
Path Type    Path Group   Start Point          -> End Point
---------    ----------   -----------          -> ---------
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[0]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[10]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[11]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[12]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[13]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[14]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[15]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[16]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[17]
Setup        bftClk       ingressLoop[0]/ram/CLKBWRCLK -> transformLoop[0].ct/xOutReg_reg/A[18]
```

Creating Custom DRCs

The Vivado Design Suite lets you define and use custom design rule checks written in Tcl. The process in brief is:

1. Write a Tcl procedure to get the design objects of interest, or attributes of those objects, and the checking function that defines the design rule. The Tcl checker procedure is defined in a separate Tcl script that must be loaded into the Vivado Design Suite prior to running `report_drc`.
2. Inside the Tcl checker, you will use the `create_drc_violation` command to identify and flag violations found when checking the rule against a design. This command creates a violation object within the in-memory design, with properties that can be reported and further processed in the Vivado Design Suite.
3. Define a user-defined DRC rule check using the `create_drc_check` command that associates the name of the DRC rule with the Tcl checker procedure specified by the `-rule_body` argument. You will call this rule by name when you run the `report_drc` command.

4. Create a rule deck using the `create_drc_ruledeck` command, and add the user-defined DRC rule to the rule deck using the `add_drc_checks` command.
5. Run `report_drc`, and specify either the rule deck, or the user-defined DRC rule to check for violations.

Writing a Tcl DRC Checker

The flow of the Tcl checker procedure, or the Tcl script that defines the design rule, first selects the specific design objects to be checked, and then performs the necessary tests or evaluations, and finally returns the results in the form of DRC violation objects that identify the objects associated with the specific error.

The following Tcl script defines the `dataWidthCheck` procedure which checks the width of the WRITE_B bus. This Tcl script file must be loaded into the Vivado tool prior to running the `report_drc` command. Refer to Loading and Running Tcl Scripts, page 31 for more information on loading the Tcl checker.

```
# This is a simplistic check -- report BRAM cells with WRITE_WIDTH_B wider than 36.
proc dataWidthCheck {} {
  # list to hold violations
  set vios {}
  # iterate through the objects to be checked
  foreach bram [get_cells -hier -filter {PRIMITIVE_SUBGROUP == bram}] {
    set bwidth [get_property WRITE_WIDTH_B $bram]
    if { $bwidth > 36} {
      # define the message to report when violations are found
      set msg "On cell %ELG, WRITE_WIDTH_B is $bwidth"
      set vio [ create_drc_violation -name {RAMW-1} -msg $msg $bram ]
      lappend vios $vio
      }; # End IF
    }; # End FOR
    if {[llength $vios] > 0} {
      return -code error $vios
    } else {
      return {}
  }; # End IF
} ; # End PROC
```

The `dataWidthCheck` procedure accepts no arguments, as you can see from the `proc` definition. Everything the procedure needs it will find in the design.

It creates an empty list variable, `$vios`, to store the violation objects returned by the `create_drc_violation` command.

It selects the design objects of interest to the design rule being checked, in this case the BRAMs in the design, and it iterates through each of the cells to get the WRITE_WIDTH_B property. It checks the property for a specific value, which triggers a violation if the property exceeds a width of 36.

If a violation is found, the procedure creates a message, `$msg`, containing placeholder values for the cell, `%ELG`, with the width of the bus found, `$bwidth`. In the `dataWidthCheck` procedure, the `create_drc_violation` command only returns one object, `$bram`, that maps to the `%ELG` placeholder defined in the message string.

IMPORTANT: Both the order and the type of objects passed by `create_drc_violation` must match the `-msg` specification in the `create_drc_check` command, or the expected substitution will not occur.

A violation object is created, using `create_drc_violation`, each time the tested BRAM exceeds the allowable width for the `WRITE_WIDTH_B` property. The violation object is given the same name as the associated DRC rule in the Vivado Design Suite, includes a message string defined by the `dataWidthCheck` procedure, and identifies the specific object or objects that are in violation of the rule. The standard objects that the design rule violation can return include cells, ports, pins, nets, clock regions, device sites, and package I/O banks. However, as shown in the `dataWidthCheck` procedure, the message string from the violation can pass other information, such as the value of a specific property, to provide as much detail in the DRC report as may be needed.

If any violations are found, the `dataWidthCheck` proc returns an error code to inform the `report_drc` command of the results of that specific check:

```
return -code error $vios
```

In addition to the error code, the violation objects are returned with the `$vios` variable, which stores a list of the violation objects created by the procedure.

Vivado Tcl DRC commands

With the Tcl checker procedure defined, you must now define the DRC check as part of the DRC reporting system within the Vivado Design Suite.

First, you must register the new design rule using the `create_drc_check` command. This command allows you to define a unique name or abbreviation for the user-defined rule check that must match the name given to the violation created by the Tcl checker procedure. In the `dataWidthCheck` proc defined above, the `create_drc_violation` command uses the name "`RAMW-1`". This is the name you will need to specify when creating the DRC check inside the Vivado tool:

```
create_drc_check -name {RAMW-1} -category {RAMB Checks} \
  -desc {Block RAM Data Width Check} -rule_body dataWidthCheck
```

You can optionally group the DRC check into a special category, and provide a description of the rule, for reporting purposes.

You can define a message to add to the DRC report when violations are encountered. By default, the message created by the `create_drc_violation` command in the Tcl

procedure is passed upward to the DRC check object. In this case, any message defined by create_drc_violation in the -rule_body is simply passed through to the DRC report.

Finally, you must use the `-rule_body` option to specify the name of the Tcl procedure to be run by the Vivado Design Suite when the rule is checked. The procedure must be loaded into the Vivado Design Suite prior to running the `report_drc` command, so that the necessary checks are fully defined.

The DRC rule check can then be run directly, individually or with other rules, when using the `report_drc` command. You can create a rule deck, which is a related collection of rules that can be run together, by using the `create_drc_ruledeck` and the `add_drc_checks` commands. You can also remove DRC checks from rule decks using the `remove_drc_checks` command.

The DRC rule check object features the `is_enabled` property that can be set to TRUE or FALSE using the `set_property` command. When a new rule check is created, the is_enabled property is set to TRUE as a default. Set the `is_enabled` property to FALSE to disable the rule check from being used when `report_drc` is run. This lets you create new DRC checks, add them to rule decks using `add_drc_checks`, and then enable them or disable them as needed without having to remove them from the rule deck.

Loading and Running Tcl Scripts

The Vivado Design Suite offers several different ways to load and run a Tcl script during a design session. You can have script files loaded automatically when the tool is launched, source scripts from the Tcl command line, or add them to the menus in the Vivado IDE.

Initializing Tcl Scripts

The Vivado Design Suite can automatically load Tcl scripts defined in an `init.tcl` file. This approach is useful when you have written Tcl procedures that define new commands that you want to make available in all your Vivado sessions.

When you start the Vivado tool, it looks for a Tcl initialization script in two different locations:

1. In the software installation: `<installdir>/Vivado/version/scripts/init.tcl`
2. In the local user directory:
   a. For Windows 7: `%APPDATA%/Roaming/Xilinx/Vivado/init.tcl`
   b. For Linux: `$HOME/.Xilinx/Vivado/init.tcl`

Where <*installdir*> is the installation directory where the Vivado Design Suite is installed.

If `init.tcl` exists in both of these locations, the Vivado tool sources the file from the installation directory first, and then from your home directory.

The `init.tcl` file in the installation directory allows a company or design group to support a common initialization script for all users. Anyone starting the Vivado tool from that software installation sources the enterprise `init.tcl` script.

The `init.tcl` file in the home directory allows each user to specify additional commands, or to override commands from the software installation to meet their specific design requirements.

The `init.tcl` file is a standard Tcl script file that can contain any valid Tcl command supported by the Vivado tool. You can even source another Tcl script file from within `init.tcl` by adding the `source` command.

Sourcing Tcl Scripts

The source command lets you manually load Tcl script files into the Vivado tool:

```
source <filename>
```

Where <*filename*> specifies both the name of the file, as well as the relative or absolute path to the file. If no path is specified as part of the file name, then the Vivado tool looks for the file in the working directory, or the directory from which the Vivado Design Suite was launched.

Within the Vivado IDE you can also source a Tcl script from the **Tools >Run Tcl Script** menu command.

By default, the source command echoes each line of the file to the Tcl console. This can be prevented by using the `-notrace` option, which is specific to the Vivado Tcl interpreter:

```
source <filename> -notrace
```

Defining Tcl Hook Scripts

In a non-project flow you have the ability to source a Tcl script at any point in the flow, such as before or after running the `synth_design` command. You can also do this in a project-based flow, using the Vivado IDE, or by using the `set_property` command to set a property on either a synthesis or implementation run. Tcl hook scripts allow you to run custom Tcl scripts prior to (`tcl.pre`) and after (`tcl.post`) synthesis and implementation design runs, or any of the implementation steps.

Whenever you launch a synthesis or implementation run, the Vivado tool uses a predefined Tcl script to process a standard design flow based on the selected strategy. Tcl hook scripts let you customize the standard flow, with pre-processors or post-processors. Being able to add Tcl script processing anywhere in a run can be useful. Every step in the design flow has a pre- and post-hook capability. Common uses are:

- Custom reports: timing, power, utilization, or any user-defined tcl report.
- Modifying the timing constraints for portions of the flow only.
- Modifications to netlist, constraint, or device programming.

In the GUI you can specify Tcl hook scripts to be sourced by using the **Change Run Settings** command for the design run. For more information refer to "Creating and Managing Runs", in the *Vivado Design Suite User Guide: Design Flows Overview (UG892)*. There are `tcl.pre` and `tcl.post` options which you can use to specify a Tcl hook script as shown in Figure Figure 1-6.

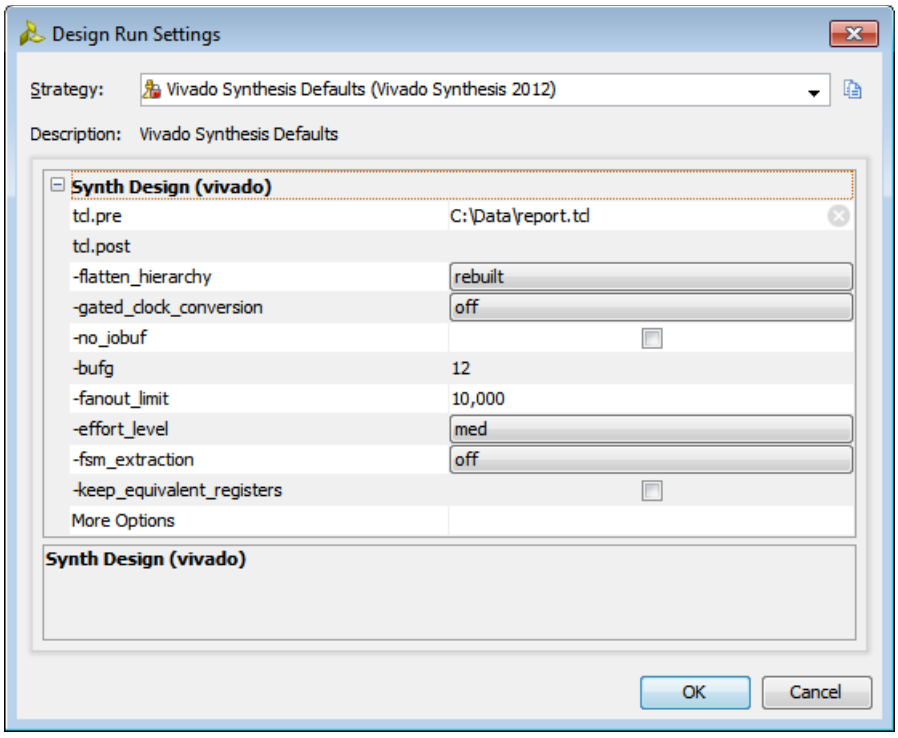

Figure 1-6: **Defining Pre and Post Run Tcl Scripts**

The Vivado IDE sets a property on the synthesis or implementation run to specify the tcl.pre or tcl.post script to apply before or after the run. You can also set this property directly on a synthesis or implementation run, either in the Tcl Console or as part of a Tcl script.

The properties to set on a synthesis run are:

```
STEPS.SYNTH_DESIGN.TCL.PRE
STEPS.SYNTH_DESIGN.TCL.POST
```

For instance, set the following property to have the `report.tcl` script launched before synthesis is complete:

```
set_property STEPS.SYNTH_DESIGN.TCL.PRE {C:/Data/report.tcl} [get_runs synth_1]
```

For an implementation run you can define Tcl scripts before and after each step of the implementation process: Opt Design, Power Opt Design, Place Design, Post-Place Power Opt Design, Phys Opt Design, Route Design and Bitstream generation. The properties for each of these are:

```
STEPS.OPT_DESIGN.TCL.PRE
STEPS.OPT_DESIGN.TCL.POST
STEPS.POWER_OPT_DESIGN.TCL.PRE
STEPS.POWER_OPT_DESIGN.TCL.POST
STEPS.PLACE_DESIGN.TCL.PRE
STEPS.PLACE_DESIGN.TCL.POST
STEPS.POST_PLACE_POWER_OPT_DESIGN.TCL.PRE
STEPS.POST_PLACE_POWER_OPT_DESIGN.TCL.POST
STEPS.PHYS_OPT_DESIGN.TCL.PRE
STEPS.PHYS_OPT_DESIGN.TCL.POST
STEPS.ROUTE_DESIGN.TCL.PRE
STEPS.ROUTE_DESIGN.TCL.POST
STEPS.WRITE_BITSTREAM.TCL.PRE
STEPS.WRITE_BITSTREAM.TCL.POST
```

IMPORTANT: *Relative paths within the `tcl.pre` and `tcl.post` scripts are relative to the appropriate run directory of the project they are applied to: `<project>/<project.runs>/<run_name>`. You can use the DIRECTORY property of the current project or current run to define the relative paths in your Tcl hook scripts:*

```
get_property DIRECTORY [current_project]
get_property DIRECTORY [current_run]
```

Customizing the GUI

You can use the **Tools > Custom Commands > Customize Commands** menu item to add system or user-defined Tcl commands to the Vivado IDE main menu and toolbar menu. Refer to "Adding Custom Menu Commands" in the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* for more information on adding custom commands to the menu.

Tcl Scripting Tips

The runtime and efficiency of Tcl scripts can be improved by following few rules. The following are a few suggestions for ways to improve using Tcl scripting in the Vivado Design Suite.

Caching Objects

Cache objects or lists of objects in Tcl variables so that they can be reused.

For example, if the same list of nets is going to be reused multiple times in the script, it does not make sense to do the same query over and over. Although the Tcl commands in the

Vivado tool have been implemented efficiently, every Tcl query goes back and forth between the Tcl interpreter and the lower level C++ code of the application. This C++/Tcl interface consumes runtime that should be avoided when possible.

Use the different filtering capabilities of the Vivado tool as often as possible. Tools such as effective search pattern definition, `-of_objects` option, the `-filter` option, and the `filter` command can reduce run-time. Those features have been implemented at a very low level in the application, and are very efficient in terms of runtime and memory.

By caching the results of a general query, a list of objects can be post-processed using the `filter` command to create a sub-list of objects. You can also use standard Tcl list commands to parse the results assigned to the Tcl variable without accessing the in-memory design unless necessary.

```
set allCells [get_cells * -hier]
lsort $allCells ; # Returns a sort ordered list of all cells
filter $allCells {IS_PRIMITIVE} ; # Returns only the primitive cells
filter $allCells {!IS_PRIMITIVE} ; # Returns non-primitive cells
```

Object Names and the NAME Property

While some Tcl commands expect a design object, other commands may expect a string input. The Vivado Design Suite has been implemented to allow design objects to be passed directly to Tcl commands, even those expecting a string argument. In this case, the hierarchical name of the design object is passed to the Tcl command as a string. There is no need to access the NAME property of the object in order to pass it to the Tcl command.

For example in the following `regexp` command, both IF statements are equivalent, since in both cases the Tcl interpreter is passed the name of the object:

```
if {[regexp {.*enable.*} $MyObject]} { ... }
if {[regexp {.*enable.*} [get_property NAME $MyObject]]} { ... }
```

In this example, the first expression is not only easier to read than the second expression, it will also run much faster than the second, since it does not have to access and return the properties on the object. The `get_property` command in the second statement will cause the Vivado tool to iterate between the Tcl interpreter and the underlying C++ application code to access and return the object properties. If this is done in a looping construct, for multiple objects, it can significantly increase the run time for your Tcl script.

Formatting Lists of Objects

When a list is returned from the `get_*` commands, the list is un-formatted and returned to `stdout` in a single line delimited by a space. This is shown in the following example:

```
get_cells
A B clk_IBUF_inst rst_IBUF_inst din0_IBUF_inst din1_IBUF_inst dout0_OBUF_inst
dout1_OBUF_inst dout2_OBUF_inst dout3_OBUF_inst clk_IBUF_BUFG_inst
```

This un-formatted return makes it difficult to see what has been returned in the Tcl Console and the Vivado IDE. To have each item in the list returned on a separate line, simply execute the command nested in a `join` command, with the newline character, '`\n`', as follows:

```
join [get_cells] \n
A
B
clk_IBUF_inst
rst_IBUF_inst
din0_IBUF_inst
din1_IBUF_inst
dout0_OBUF_inst
dout1_OBUF_inst
dout2_OBUF_inst
dout3_OBUF_inst
clk_IBUF_BUFG_inst
```

The list returned by the `get_*` command is unaffected by the `join` command.

Finding Vivado Tcl Commands by Options

The following procedure, `findCmd`, searches through the syntax of all the Tcl commands in the Vivado Design Suite, and displays a list of commands that support the specified option:

```
proc findCmd {option} {
  foreach cmd [lsort [info commands *]] {
    catch {
      if {[regexp "$option" [help -syntax $cmd]]} {
        puts $cmd
      }
    }
  }
} ; # End proc
```

To find the Vivado tool commands that support the `-return_string` option use:

```
findCmd return_string
```

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

Vivado Design Suite 2012.4 Documentation:

(www.xilinx.com/support/documentation/dt_vivado_vivado2012-4.htm)

Tcl Developer Xchange:

www.tcl.tk