Vivado Design Suite User Guide

Using Tcl Scripting

UG894 (v2013.3) October 2, 2013

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.
04/22/2013	2013.1	Updated or added the following sections: • Getting Help • Compilation with a Project Flow • Using the -hierarchical option • DRC Explanation Script • Using Tcl Scripts in a Constraints Set • Controlling Loops • Error Handling • Accessing Environment Variables • Additional Resources
06/19/2013	2013.2	Added the following sections: • Local and Global Variables • Creating a DRC Check • Creating a DRC Rule Deck
10/02/2013	2013.3	Added or revised the following section: • get_nets Command • Writing a Tcl Script • Defining Tcl Procedures • Parsing Command Line Arguments • Namespaces for Procedures • Template Script

Table of Contents

XILINX®

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.

Note: For more information, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 1], or type `<command> -help`.

Note: For information on launching and using the Vivado® Design suite, see *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 2]

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. 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, as it is easily adopted and mastered by designers familiar with this 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 can find detailed information regarding Vivado specific Tcl commands in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 1], or in the help system of the Vivado tools.

The Vivado IDE uses Xilinx Design Constraints (XDC) to specify the design constraints. XDC is based on a subset of all the Tcl commands available in Vivado and is interpreted exactly like Tcl. The XDC commands are primarily timing constraints, physical constraints, object

queries and a few Tcl built-in commands: `set`, `list`, and `expr`. For a complete description of the XDC commands, see Appendix B of the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 3]. Unlike Tcl scripts, XDC files are managed by the Vivado IDE so that any constraint edited through the graphical interface or the Timing Constraints Editor can be saved back to its original XDC file. For this reason, only XDC commands can be used in a XDC file. If you need to use other Tcl commands to write your constraints, you must use a Tcl script.

The Vivado tools write 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.

A log file, `vivado.log` is also created by the tool and includes the output of the commands that are executed. Both the journal and log file are useful to verify which commands were run and what result they produced.

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/tcl8.5/tutorial/tcltutorial.html [Ref 8]for an introductory tutorial to the Tcl programming language. Also see the Tclers Wiki located at http://wiki.tcl.tk for some example scripts.

In this document you will see some examples of Tcl commands and Tcl scripts, and the results that are returned by the Vivado Design Suite when these commands are run. The commands and their return values appear 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 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.

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

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 consists 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 2013.1
```

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 from left to right 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 existing 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 slightly differently than standard Tcl. Square brackets are treated as standard characters in Verilog and VHDL names (net, instances, etc.), and usually identify one or more elements of vectors, such as busses or arrays of instances. In the Vivado tools 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 tools as part of the specified object name `transformLoop[0]`. This is handled automatically by the Vivado Design Suite, but is limited to certain characters. These characters must be in one of the following forms, or the brackets will be interpreted as they would be in normal Tcl syntax:

- 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 hash character 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 hash character as shown below:

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

Getting Help

You can get help directly from the Tcl console. Every Vivado command supports the `-help` command line argument that can be used anywhere in the line.

For example:

```
Vivado% create_clock -help
Vivado% create_clock -name CLK1 -period 10 -help
```

In addition, there is a `help` command that provides additional information. Providing a command name to the `help` command (i.e `help` <*command*>) reports the same help information as <*command*> `-help`:

```
Vivado% help create_clock
```

The `help` command can also just return a short description of the arguments with the `-args` option:

```
Vivado% help create_clock -args

create_clock

Description:
Create a clock object

Syntax:
create_clock  -period <arg> [-name <arg>] [-waveform <args>] [-add] [-quiet]
[-verbose] [<objects>]

Returns:
new clock object
```

```
Usage:
  Name          Description
  ------------------------
  -period       Clock period: Value > 0
  [-name]       Clock name
  [-waveform]   Clock edge specification
  [-add]        Add to the existing clock in source_objects
  [-quiet]      Ignore command errors
  [-verbose]    Suspend message limits during command execution
  [<objects>]   List of clock source ports, pins or nets
```

A short summary of the syntax of a command is also available with the `-syntax` option:

```
Vivado% help create_clock -syntax

create_clock

Syntax:
create_clock  -period <arg> [-name <arg>] [-waveform <args>] [-add]
[-quiet][-verbose] [<objects>]
```

In addition to providing help for the specific commands, the help command can also provide information on categories of commands or classes of objects. A list of categories can be obtained by executing the help command without any argument or option. A non-exhaustive list of categories is:

```
Vivado% help

ChipScope
DRC
FileIO
Floorplan
GUIControl
IPFlow
Object
PinPlanning
Power
Project
PropertyAndParameter
Report
SDC
Simulation
TclBuiltIn
Timing
ToolLaunch
Tools
XDC
```

The list of commands available under each category can be also reported with the `-category` option. For example, the following command reports all the commands under the Tools category:

```
Vivado% help -category tools

Topic            Description
link_design      Open a netlist design
list_features    List available features.
load_features    Load Tcl commands for a specified feature.
opt_design       Optimize the current netlist. This will perform the retarget,
                  propconst, and sweep optimizations by default.
phys_opt_design  Optimize the current placed netlist.
place_design     Automatically place ports and leaf-level instances
route_design     Route the current design
synth_design     Synthesize a design using Vivado Synthesis and open that design
```

Compilation and Reporting Example Scripts

Compilation with a Non-Project 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 22.

```
# 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 $outputDir/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 $outputDir/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 (EDIF or NGC) 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 Guide (UG911) [Ref 4] 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 the 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) [Ref 5].

- **Step 3:** synthesizes the design on the specified target device.

  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 tools, 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.

Compilation with a Project Flow

The following script illustrates a Project flow that synthesizes the design and performs a complete implementation, including bitstream generation. It is based on the CPU example design provided in the Vivado installation tree.

```
#
# STEP#1: define the output directory area.
#
set outputDir ./Tutorial_Created_Data/cpu_project
file mkdir $outputDir
create_project project_cpu_project ./Tutorial_Created_Data/cpu_project \
    -part xc7k70tfbg676-2 -force
#
# STEP#2: setup design sources and constraints
#
add_files -fileset sim_1 ./Sources/hdl/cpu_tb.v
add_files [ glob ./Sources/hdl/bftLib/*.vhdl ]
add_files ./Sources/hdl/bft.vhdl
add_files [ glob ./Sources/hdl/*.v ]
add_files [ glob ./Sources/hdl/mgt/*.v ]
add_files [ glob ./Sources/hdl/or1200/*.v ]
add_files [ glob ./Sources/hdl/usbf/*.v ]
add_files [ glob ./Sources/hdl/wb_conmax/*.v ]
add_files -fileset constrs_1 ./Sources/top_full.xdc
set_property library bftLib [ get_files [ glob ./Sources/hdl/bftLib/*.vhdl ]]
#
# Physically import the files under project_cpu.srcs/sources_1/imports directory
import_files -force -norecurse
```

```
#
# Physically import bft_full.xdc under project_cpu.srcs/constrs_1/imports directory
import_files -fileset constrs_1 -force -norecurse ./Sources/top_full.xdc
# Update compile order for the fileset 'sources_1'
set_property top top [current_fileset]
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
#
# STEP#3: run synthesis and the default utilization report.
#
launch_runs synth_1
wait_on_run synth_1
#
# STEP#4: run logic optimization, placement, physical logic optimization, route and
#         bitstream generation. Generates design checkpoints, utilization and timing
#         reports, plus custom reports.
set_property STEPS.PHYS_OPT_DESIGN.IS_ENABLED true [get_runs impl_1]
set_property STEPS.OPT_DESIGN.TCL.PRE [pwd]/pre_opt_design.tcl [get_runs impl_1]
set_property STEPS.OPT_DESIGN.TCL.POST [pwd]/post_opt_design.tcl [get_runs impl_1]
set_property STEPS.PLACE_DESIGN.TCL.POST [pwd]/post_place_design.tcl [get_runs impl_1]
set_property STEPS.PHYS_OPT_DESIGN.TCL.POST [pwd]/post_phys_opt_design.tcl [get_runs impl_1]
set_property STEPS.ROUTE_DESIGN.TCL.POST [pwd]/post_route_design.tcl [get_runs impl_1]
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
puts "Implementation done!"
```

Details of the Sample Script

- **Step 1:** The project is created with the `create_project` command. The project directory and the target device are specified. The project directory is automatically created if it does not already exist.

  In this example, the output directory where the various reports are saved is the same as the project directory.

- **Step 2:** All the files that are used in a project need to be explicitly declared and added to the project. This is done with the `add_files` command. When a file is added to the project, it is added to a specific fileset. A fileset is a container that groups files together for a purpose. In this example script, most of the files are added to the default fileset (`sources_1`). Only the Verilog testbench `cpu_tb.v` is added to the default simulation fileset `sim_1`.

  The files are also copied inside the project directories with the `import_files` command. By doing this, the project points to the local copy of the source files and does not track the original source files anymore.

- **Step 3:** The design is synthesized by launching a synthesis run in the background (`launch_run synth_1`). The Vivado IDE automatically generates all the necessary scripts to run synthesis in a separate Vivado session. Because synthesis runs in a different process, it is necessary to wait for its completion before continuing the current script. This is done by using the `wait_on_run` command.

Once the synthesis run is completed, the results can be loaded in memory with the `open_run synth_1` command. A checkpoint without constraints is saved in the project directories, where synthesis was run. In this example, it can be found under:

```
./Tutorial_Created_Data/cpu_project/project_cpu.runs/synth_1/top.dcp
```

Note: The names `synth_1` and `impl_1` are default names for the synthesis and implementation runs. Additional runs can be created with `create_run` command.

- **Step 4:** The implementation is done by using the `launch_run` command. The complete P&R flow from pre-place optimization to writing the bitstream can be performed in a single command. In this example script, the implementation is done up to the bitstream generation (`launch_run impl_1 -to_step write_bitstream`).

The optional step `phys_opt_design` is enabled in the script through the property `STEPS.PHYS_OPT_DESIGN.IS_ENABLED`. Unlike with the non-project flow which allows dynamically calling the implementation commands based on conditions defined by the user, the run of a project flow must be configured statically before it is launched. This is why, in this example, the physical logic optimization step is enabled without checking the timing slack value after placement, unlike in the Compilation with a Non-Project Flow example.

The various reports are generated before or after each implementation step by using the run Tcl hook properties `STEPS.<STEPNAME>.TCL.PRE` and `STEPS.<STEPNAME>.TCL.POST`. These properties allow the user to specify where a Tcl script is executed in the flow when using the run infrastructure. See Defining Tcl Hook Scripts, page 20 for additional information.

Because the implementation run is executed in a separate Vivado session, all the Tcl variables and procs need to be initialized in that session in order to be used by the scripts. This can be done in several ways:

- **Method 1:** Define the Tcl variables and procs in your init.tcl (see Initializing Tcl Scripts, page 18). This is sticky to all your Vivado projects and sessions.
- **Method 2:** Add a Tcl script which contains the variables and procs to the constraints set used by the run. It will always be sourced as part of your constraints when opening the design in memory.
- **Method 3:** Set `STEPS.OPT_DESIGN.TCL.PRE` to a Tcl script which contains the variables and proc. This script will only be sourced if the `OPT_DESIGN` step is enabled, which is true by default.

The current example uses the Method 3. The Tcl scripts are associated with the implementation steps as follow:

```
set_property STEPS.OPT_DESIGN.TCL.PRE       [pwd]/pre_opt_design.tcl       [get_runs impl_1]
set_property STEPS.OPT_DESIGN.TCL.POST      [pwd]/post_opt_design.tcl      [get_runs impl_1]
set_property STEPS.PLACE_DESIGN.TCL.POST    [pwd]/post_place_design.tcl    [get_runs impl_1]
set_property STEPS.PHYS_OPT_DESIGN.TCL.POST [pwd]/post_phys_opt_design.tcl [get_runs impl_1]
set_property STEPS.ROUTE_DESIGN.TCL.POST    [pwd]/post_route_design.tcl    [get_runs impl_1]
```

The absolute Tcl script path must be specified because the implementation run is executed in a sub-directory of the project tree, which is different from the one where the full compilation Tcl script is executed.

- `pre_opt_design.tcl`

```
############## pre_opt_design.tcl ##################
set outputDir [file dirname [info script]]/Tutorial_Created_Data/cpu_project
source [file dirname [info script]]/reportCriticalPaths.tcl
#
report_timing_summary -file $outputDir/post_synth_timing_summary.rpt
report_utilization -file $outputDir/post_synth_util.rpt
reportCriticalPaths $outputDir/post_synth_critpath_report.csv
```

  The two first lines correspond to the initialization of the variable and proc used by several scripts later in the implementation run. The three next lines run some utilization and timing reports. It is generally recommended to run timing analysis at the beginning of implementation to validate the timing constraints used during place and route, and insure there is no large violation. The `reportCriticalPaths` report provides more info on the worst paths of the design. This Tcl proc is described further in Defining Tcl Procedures, page 22.

- `post_opt_design.tcl`

```
############## post_opt_design.tcl ##################
# Run custom script to report critical timing paths
reportCriticalPaths $outputDir/post_opt_critpath_report.csv
```

  This script does not need to define the `outputDir` variable and `reportCriticalPaths` proc because they are already defined in `pre_opt_design.tcl` which is sourced earlier in the run in the same Vivado session.

  It is recommended to also run utilization and timing analysis after `opt_design`.

- `post_place_design.tcl`

```
############## post_place_design.tcl ##################
report_clock_utilization -file $outputDir/clock_util.rpt
```

  After placement, you can review the utilization of the clock resources and where they are located in the device. It is recommended to run timing analysis to identify large timing violations that cannot be resolved later in the flow.

- `post_phys_opt_design.tcl`

```
############## post_phys_opt_design.tcl ##################
report_utilization -file $outputDir/post_phys_opt_util.rpt
report_timing_summary -file $outputDir/post_phys_opt_timing_summary.rpt
```

  Like after placement, it is important to review the timing report at this point of the flow.

- `post_route_design.tcl`

```
############## post_route_design.tcl ##################
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
```

After route, the timing analysis uses actual routed net delays and must be reviewed for timing signoff. The route status report summarizes the number of unresolved routing issues. If any, the DRC report often helps identify what the routing issues are.

Note: Most of the Tcl reports generated during post-route above are also automatically created by the run infrastructure. Similarly, a design checkpoint is generated after each step of the flow, so there is usually no need to call the `write_checkpoint` command in your scripts when using a project flow. You can find all the checkpoints and default reports in the implementation run directory:

```
./Tutorial_Created_Data/cpu_project/project_cpu.runs/impl_1/
  top_opt.dcp
  top_placed.dcp
  top_physopt.dcp
  top_routed.dcp

  top_clock_utilization_placed.rpt
  top_control_sets_placed.rpt
  top_utilization_placed.rpt
  top_io_placed.rpt
  top_drc_routed.rpt
  top_power_routed.rpt
  top_route_status.rpt
  top_timing_summary_routed.rpt
```

Once the implementation run is complete, the implemented design can be loaded in memory with the `open_run impl_1` command.

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 tools, 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 tools 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 tools. 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 tools:

```
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 tools look 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
```

Using Tcl Scripts in a Constraints Set

Tcl scripts can be added to project constraint sets like any regular XDC file, except that the XDC files are managed by the tool, and not Tcl scripts. Any constraint defined by a Tcl script and edited by the tool cannot be saved back to the Tcl script automatically. If you need to save your edits, you must export all the constraints in memory to a file and use this file to update your script manually. When opening a design in memory (`open_run`), the Tcl scripts are sourced after the XDC files. In a non-project flow, this is equivalent to explicitly sourcing the Tcl scripts after loading the XDC files with `read_xdc`. For more information on using XDC files and Tcl scripts in a constraints set, see *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 3].

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 tools use 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-hook 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) [Ref 5]. There are

`tcl.pre` and `tcl.post` options which you can use to specify a Tcl hook script as shown in the following figure.

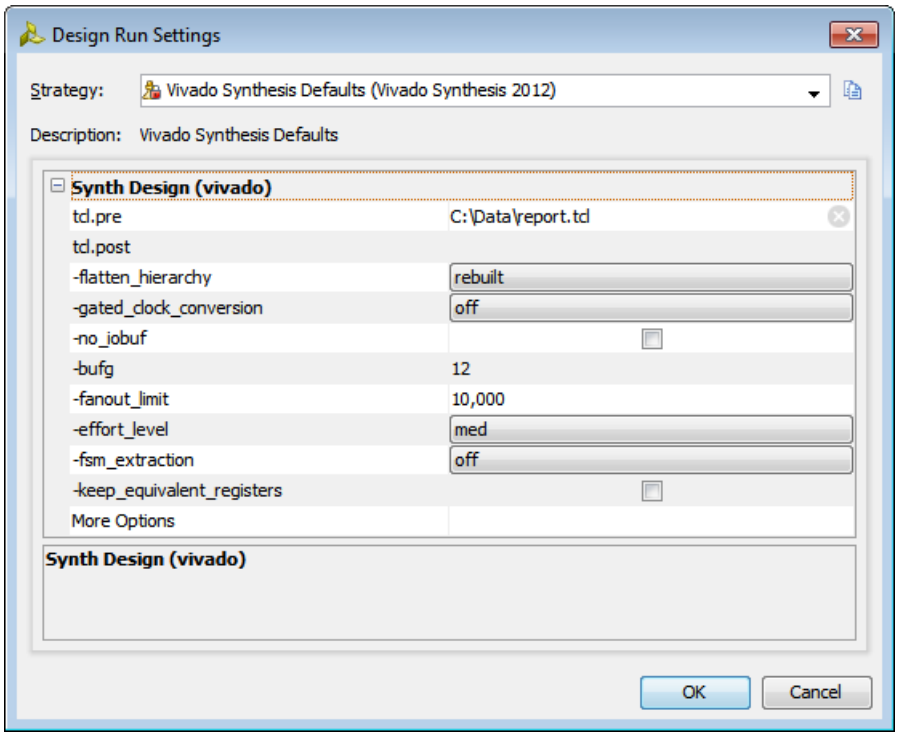

Figure 1: **Defining Pre and Post Run Tcl Scripts**

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:*

```
set_property DIRECTORY [current_project]
set_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) [Ref 6] for more information on adding custom commands to the menu.

Writing a Tcl Script

When writing a Tcl script, the emphasis should be, whenever possible, on providing code that can enhance the user experience. This means writing scripts that provide the same type of user experience as the embedded Vivado commands such as providing some type of embedded help and interactive command line argument. It is also critical to consider all the corner cases that could happen, whether Vivado objects are empty or not after using the `get_*` commands and so forth. It is also common when writing Tcl code, to create some lower-level procedures that are used in the code. To avoid name collision of procedures and global variables, Xilinx recommends that you develop the code within its own namespace so that name collisions are minimized.

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 that can be

loaded and run from the Vivado IDE, or you can write procedures (or procs), to act like new Tcl 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 be executed. 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 argument definition of this procedure, since helloProc has only one argument, it makes the procedure definition easier to read by enclosing the arguments in braces. The braces are required when the procedure accepts multiple arguments.*

A procedure usually has predefined arguments. Each of them can optionally have a default value. When an argument has a default value, it does not need to be specified when calling the procedure if all the mandatory preceding arguments are specified. A procedure returns an empty string unless the `return` command is used to return a different value. The following example defines a procedure, `reportWorstViolations`, with three arguments:

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

When running the procedure, all the arguments must be specified as shown:

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

The next example is a different form of the same procedure, where the last 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 specified in the definition of the procedure, 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, all of the following calls of the command are valid:

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

The following example is an illustration of a procedure that has one mandatory argument, `nbrPath`, but that can also accept any number of additional arguments. This uses the Tcl keyword `args` in the list of arguments when defining the procedure. The `args` keyword is 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 a Tcl command, 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
```

Parsing Command Line Arguments

It can be useful and sometimes necessary to write procedures that use external parameters or arguments as they can broaden the spectrum of usage of the procedure without having to write too much redundant code. A single procedure that can handle multiple contexts is easier to use and maintain that multiple procedures that cover the same range of contexts with duplicated code.

This is especially useful when the procedure is being used interactively. It is a lot friendlier for the user to be able to specify some command line options like with any Vivado commands.

Tcl provides an easy way to do this through the `args` variable. The keyword `args` used inside the list of arguments of a procedure can match any number of elements, including none. The `args` variable is a regular Tcl list that can be processed and analyzed like any Tcl list.

There are multiple techniques to parse the command line arguments, and the example code below shows just one of them:

```
01  proc lshift listVar {
02      upvar 1 $listVar l
03      set r [lindex $l 0]
04      set l [lreplace $l [set l 0] 0]
05      return $r
06  }
07
08
09  proc myproc { args } {
10
11    #-------------------------------------------------------
12    # Process command line arguments
13    #-------------------------------------------------------
14    set error 0
15    set help 0
16    set verbose 0
17    set ports {}
```

```
18    # if {[llength $args] == 0} { incr help }; # Uncomment if necessary
19    while {[llength $args]} {
20      set flag [lshift args]
21      switch -exact -- $flag {
22        -p -
23        -ports {
24             set ports [lshift args]
25        }
26        -v -
27        -verbose {
28             set verbose 1
29        }
30        -h -
31        -help {
32             incr help
33        }
34        default {
35             if {[string match "-*" $flag]} {
36               puts " ERROR - option '$flag' is not a valid option."
37               incr error
38             } else {
39               puts "ERROR - option '$flag' is not a valid option."
40               incr error
41             }
42        }
43      }
44    }
45
46    if {$help} {
47      set callerflag [lindex [info level [expr [info level] -1]] 0]
48      # <-- HELP
49      puts [format {
50    Usage: %s
51                [-ports|-p <listOfPorts>]
52                [-verbose|-v]
53                [-help|-h]
54
55    Description: xxxxxxxxxxxxxxxxxx.
56                 xxxxxxxxxxxxxxxxxx.
57
58    Example:
59       %s -port xxxxxxxxxxxxxx
60
61  } $callerflag $callerflag ]
62      # HELP -->
63      return -code ok {}
64    }
65
66    # Check validity of arguments. Increment $error to generate an error
67
68    if {$error} {
69      return -code error {Oops, something is not correct}
70    }
71
72    # Do something
73
74    return -code ok {}
75  }
```

Explanations:

1. Lines 1-6: Definition for the `lshift` procedure that removes the first element of a list.
2. Line 9: `myproc` is defined with a single argument, `args`, that can take any number of elements. In this example code, `myproc` supports three command line options: `-ports` <*string*> / `-verbose` / `-help`.
3. Lines 19-44: Loop through all the command line arguments. When all the arguments have been processed, the `args` variable is empty.
4. Line 20: The command line argument that needs to be processed is saved inside the `flag` variable. The `lshift` proc is used to get the argument and remove it from the `args` variable.
5. Lines 21-43: Check the content of the flag variable against all the valid arguments. The switch statement uses the `-exact` option so that the full option name is checked against the content of `flag`. For example, to define the ports, the user needs to specify `-p` or `-ports`.

   The `-p`/`-ports` option takes a command line argument that is being read and removed from the list args with `lshift` args (line 24).

   The `-v`/`-verbose` option is just a boolean and therefore does not need any additional argument from `args` (line 28).

   Lines 31-33: Check for the `-h`/`-help` options.

   Lines 36-38: Check for any command line argument starting with "-" (without quotes). In this sample proc, they are not supported.

   Lines 39-40: Check for a command line argument that does not start with "-" (without quotes). In this example proc, they are not supported.
6. Lines 46-64: Display the embedded help if `-h`/`-help` has been specified. Those lines as well as lines 30-33 can be removed if the proc does not need to provide any embedded help.
7. Lines 68-70: Check if any error has occurred. Typically, some additional code to check the validity of the arguments should happen before line 68. If there would be any error or, for example, incompatibility between the command line options provided by the user, then the error variable could be incremented which would then trigger line 69.
8. Line 73 and beyond: Add your code here

The above code parses the command line arguments and searches for an exact match with the supported options (line 21). However, there are some cases when it might be better to match the command line arguments against some expressions instead of searching for an exact match. This is done by using the `-glob` switch instead of the `-exact` switch on line 21. See the following example.

```
21  switch -glob -- $flag {
22        -p* -
23        -ports {
24             set ports [lshift args]
25        }
26        -v* -
27        -verbose {
28             set verbose 1
29        }
30        -h* -
31        -help {
32             incr help
33        }
34        default {
35             if {[string match "-*" $flag]} {
36               puts " ERROR - option '$flag' is not a valid option."
37               incr error
38             } else {
39               puts "ERROR - option '$flag' is not a valid option."
40               incr error
41             }
42        }
43  }
```

Lines 22, 26 and 30 illustrate some expressions using the "`*`" as a wildcard. The above code matches any string starting with `-p` as a valid command line option to define the ports, for example `-pfoo`.

Although the example procedure, `myproc`, is acceptable for an interactive command, it has some runtime overhead due to the parsing of the arguments. The runtime overhead might not be acceptable for a low-level procedure that is called many times. A different technique can be used to add some command line arguments to a procedure that needs very little runtime overhead. This is done by assigning the list of commands line arguments to a Tcl array. However, this implies that each command line option has one and only one argument. See the following example.

```
01  proc myproc2 { args } {
02    # Default values
03    set defaults [list -p 123 -v 0]
04    # First, assign default values
05    array set options $defaults
06    # Then, override with user choice
07    array set options $args
08
09    set ports $options(-p)
10    set verbose $options(-v)
11    set error 0
12
13    # Check validity of arguments. Increment $error to generate an error
14
15    if {$error} {
16      return -code error {Oops, something is not correct}
17    }
18
19    # Do something
20
21    return -code ok {}
22  }
```

Explanations:

1. Line 1: `myproc2` is defined with a single argument, `args`, that can take any number of elements. However, since `args` is used later on to set a Tcl array, it must have an even number of arguments.
2. Line 3: Default values for the various options. Each option has one and only one value.
3. The format of the list is:

<option1> <valueForOption1> <option2> <valueForOption2> … <optionN> <valueForOptionN>

4. Line 5: The Tcl array `options` is initialized with the default values.
5. Line 7: The `args` variable overrides the default values
6. Line 9-10: The value of each option is being read with `$options(<option>)`. It is also possible to check that an option exist with the following.

```
if [info exists options(<option>)] { … }
```

Note: The command line options that are working as a flag and have no intrinsic value are easily implemented by passing, for example, a value of 0 or 1 with the option. In the previous example procedure, the flag `-v` is turned on with: `myproc2 -v 1`

Local and Global Variables

A local variable is a variable created inside a procedure. It is created at runtime inside the stack of the function. The variable is only accessible within the procedure and the variable name is not subject to name collision with variable names outside of the procedure. This means that, for example, a local variable `foo` created inside a procedure is different from a variable `foo` created outside of the procedure and both variables have independent content. A local variable is created with the `set` Tcl keyword like any other variable.

The parameters defined as the arguments of a procedure are, by default, local variables. Whenever a procedure is called (for example `reportCriticalPaths $myfilename`), the content of the calling variables (for example `$myfilename`) are copied inside the stack of the procedure. If the calling variables are Tcl lists with a large number of elements, this mechanism has a runtime and memory penalty. There are also some scenarios when it is necessary to modify the content of the caller variables. Tcl provides a way to do that by passing a variable name as a reference instead of passing the content of the variable. Once a variable is passed as a reference, any modification of the variable inside the procedure directly modifies the caller's variable inside the caller's space. The keyword `upvar` is used inside the body of the procedure to define a parameter that is passed as reference. The procedure `lshift` that has been introduced earlier uses this technique:

```
proc lshift {listVar} {
   upvar 1 $listVar l
   set r [lindex $l 0]
   set l [lreplace $l [set l 0] 0]
   return $r
 }
```

In the example proc `myproc`, `lshift` is called by passing the variable name args instead of the content `$args`.

A global variable is a variable created outside of a procedure and that belongs to the global namespace. To refer to a global variable inside a procedure, the keyword `global` is used followed by the variable name:

```
proc printEnv {} {
  global env
  foreach var [lsort [array names env]] { puts "  $var = $env($var)" }
}
```

The above example defines a procedure `printEnv` that prints the system environment variables. The Tcl array, `env` is a global variable initialized when the Vivado tools start. The `printEnv` procedure refers to the global `env` variable through the `global env` command. After the global variable is declared, it is accessed like any local variable. The global variable can be read and modified.

Another way to access a global variable is to specify the namespace qualifier with it. The namespace qualifier for the global namespace is "`::`" (without any quotes) and therefore a procedure can refer to the global variable `env` with `::env`. The syntax is the same for any global variable.

For example:

```
proc printEnv {} {
  foreach var [lsort [array names ::env]] { puts "  $var = $::env($var)" }
}
```

Since `printEnv` specifies the full path to the `env` variable, the procedure does not need to declare `global env`.

Note: Xilinx does not recommend that you use global variables as it relies on variable names created outside the scope of the procedure. Global variables are sometimes used to avoid having to pass large Tcl lists to a procedure. The upvar technique should always be considered before using a global variable.

Namespaces for Procedures

By default, every procedure created inside the Tcl interpreter is created inside the global namespace. A disadvantage of this is potential conflicts with the procedure or variable names when multiple Tcl scripts from different sources are being used. In addition, the global namespace is also being polluted by procedure names that might be only be used by some other procedures and that are not meant to be directly accessed by the user.

Instead of defining all the variables and procedures in the global namespace, Tcl supports the concept of namespace that encompasses variables and procedures inside a more private scope. Namespaces can also be nested, so a namespace can be defined inside another namespace without restriction on the number of levels of scoping. Namespaces add a new syntax to procedure and variable names. A double-colon, ::, separates the namespace name from the variable or procedure name.

Below is an example that illustrates how a namespace is created and how procedures and variables are assigned to the namespace. This example creates a namespace, `foo` that reproduces the functionality of a small stack with 2 public procedures (`push` and `pop`):

```
01  namespace eval foo {
02    variable stack [list]
03    variable count 0
04    variable params
05    array set params [list var1 value1 var2 value2 var3 value3]
06
07    namespace export push pop
08
09    proc push { args } {
10      variable stack
11      variable count
12      lappend stack $args
13      incr count
14    }
15
16    proc pop {} {
17      variable stack
18      variable count
19      if {[llength $stack] > 0} {
20        set value [lindex $stack end]
21        set stack [lrange $stack 0 end-1]
22        incr count -1
23        return $value
24      } else {
25        error " no more element in the stack"
26      }
```

```
27    }
28
29  }
30
31  proc foo::dump {} {
32    variable stack
33    variable count
34    if {[llength $stack] > 0} {
35      puts " There are $count element(s) in the stack:"
36      foreach element $stack {
37        puts "    $element"
38      }
39      return 0
40    } else {
41      error " no element in the stack"
42    }
43  }
44
45  namespace import foo::*
```

Explanations:

1. The namespace is defined with the command:

   `namespace eval <name> { … }`

2. Line 1 declares the namespace, `foo` and line 29 is the closing curly bracket of the namespace definition.

3. Variables inside the namespace are created with the command variable (lines 2-4):

   `variable <varname> ?<varvalue>?`

   A Tcl array cannot be initialized with the variable command. It needs to be created first (line 4) and initialized afterward (line 5).

   Note: Do not use the `set` command to declare variables inside a namespace as it will confuse the Tcl interpreter in the case the same variable name exists in the global namespace.

4. Procedures can be created directly inside the namespace definition or outside. When a procedure is created within the command, `namespace eval … { … }`, it does not need to have the namespace qualifier in the name (in this example `foo::`).

   Lines 9 and 16: `push` and `pop` are created inside the namespace definition

5. Procedures can also be created outside of the namespace definition and added to the namespace by using the full namespace qualifier prepended to the procedure name. In the above example, the procedure dump (line 31) is created output of the namespace definition but added to the namespace foo.

6. Lines 10-11, 17-18, 32-33: Procedures refer to variables created inside the namespace using the keyword `variable`.

7. A procedure created inside a namespace can be accessed with the full namespace qualifier, for example `foo::push`, `foo::pop` and `foo::dump`. From within the namespace itself, the namespace qualifier is not needed when referring to procedures

from the same namespace. For instance, if the procedure `dump` needs to call `push`, it does not need to specify `foo::push`, but just `push`.

8. Line 7: The namespace supports the concept of public and private procedures. Although all the procedures within a namespace can be accessed with the full namespace qualifier, a namespace can specify which of the procedures can be exported outside of the namespace with the command, `namespace export...` . Once a procedure name is exported, it can be imported into the global namespace with the command, `namespace import...` (line 45). Doing that enables the procedure to be directly called without having to specify the full namespace qualifier.

   Here is an example usage of the namespace `foo`:

```
vivado% foo::push This is a test
1
vivado% foo::push {This is another line}
2
vivado% push This is the third line
3
vivado% foo::dump
 There are 3 element(s) in the stack:
    This is a test
    {This is another line}
    This is the third line
0
vivado% puts "The last element stacked is: [foo::pop]"
The last element stacked is: This is the third line
vivado% puts "The previous element stacked is: [pop]"
The previous element stacked is: {This is another line}
vivado% foo::dump
 There are 1 element(s) in the stack:
    This is a test
0
vivado% dump
invalid command name "dump"
```

Template Script

Below is a template script based on the notions that been introduced earlier. It illustrates:

1. Usage of a private namespace to avoid polluting the global namespace (`lshift` is only available inside the namespace `foo`).
2. Handling of command line arguments (including `-help` and `-version` to provide a version of the script).
3. Usage of return `-error` (or `error`) command to generate Tcl errors when it is needed

```
namespace eval foo {
  namespace export myproc
  variable version 1.0
}


proc foo::lshift listVar {
    upvar 1 $listVar l
    set r [lindex $l 0]
    set l [lreplace $l [set l 0] 0]
    return $r
}


proc foo::myproc { args } {

  #-------------------------------------------------------
  # Process command line arguments
  #-------------------------------------------------------
  set error 0
  set help 0
  set verbose 0
  set ports {}
  # if {[llength $args] == 0} { incr help }; # Uncomment if necessary
  while {[llength $args]} {
    set flag [lshift args]
    switch -exact -- $flag {
      -p -
      -ports {
           set ports [lshift args]
      }
      -v -
      -verbose {
           set verbose 1
      }
      -h -
      -help {
           incr help
      }
      -version {
           variable version
           return $version
      }
      default {
           if {[string match "-*" $flag]} {
             puts " ERROR - option '$flag' is not a valid option."
             incr error
           } else {
             puts "ERROR - option '$flag' is not a valid option."
             incr error
           }
      }
    }
  }
```

```
  if {$help} {
set callerflag [lindex [info level [expr [info level] -1]] 0]
    # <-- HELP
    puts [format {
  Usage: %s
              [-ports|-p <listOfPorts>]
              [-verbose|-v]
              [-version]
              [-help|-h]

  Description: xxxxxxxxxxxxxxxxxx.
               xxxxxxxxxxxxxxxxxx.

  Example:
     %s -port xxxxxxxxxxxxxx

} $callerflag $callerflag ]
    # HELP -->
    return -code ok {}
  }

  # Check validity of arguments. Increment $error to generate an error

  if {$error} {
    return -code error {Oops, something is not correct}
  }

  # Do something

  return -code ok {}
}
```

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 tools 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. The complete list of `get_*` commands can be returned with `help get_*`. 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 67 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 tools 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 44. We can get the list of properties on an object by using the `list_property` command. When a property type is `enum`, it is possible to get the list of all the valid values by using the `list_property_value` command.

There are two properties that are common to all objects: NAME and CLASS. When an object is assigned to a Tcl variable, a pointer to the object is stored in the variable. Objects can be

passed by variable to Tcl commands or Tcl procs. When an object is passed as an argument to a Tcl proc or command which expects a string, the object's NAME property value is passed instead of the object itself. The example below shows a cell object assigned to the variable, `$inst`, and the results of the `puts` command and the `report_property` command on the `$inst` variable. Notice that the `puts` command just prints the object NAME because it only works with strings, while the `report_property` command returns all of the object properties and their values:

```
set inst [get_cells cpuEngine]
cpuEngine

puts $inst
cpuEngine

report_property $inst
Property          Type    Read-only  Value
CLASS             string  true       cell
FILE_NAME         string  true
C:/2013.1/cpu/project_1.srcs/sources_1/imports/netlist/top.edf
IS_BLACKBOX       bool    true       0
IS_PRIMITIVE      bool    true       0
IS_SEQUENTIAL     bool    true       0
LINE_NUMBER       int     true       812044
NAME              string  true       cpuEngine
PRIMITIVE_COUNT   int     true       11720
REF_NAME          string  true       or1200_top
```

You can also create custom properties for any class of design objects in the Vivado Design Suite. This can be useful when you want to annotate some information from a script onto the in-memory design objects. The following example creates a property, SELECTED, for a cell object. 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 on a specific object 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 Objects By Name

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.

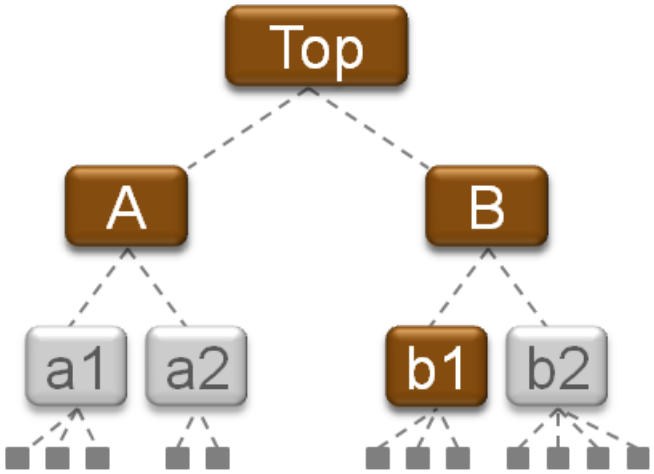

Figure 2: **Searching the Design Hierarchy**

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 2 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 (unisim instances) 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 of the design.
get_nets -hier *nt* ; # Returns all hierarchical nets that match *nt*.
```

However, one important feature of the `-hierarchical` option is that the Vivado tools try to match the specified name pattern at each level of the design hierarchy, and not against the full hierarchical name of an object. In general, when `-hierarchical` is used, the specified search pattern must not include the hierarchical separator otherwise no object will be returned. There is an exception to this rule when the netlist has been partially flattened during synthesis. That is when the hierarchy separator is also used to mark the flattened netlist level. In this case, it is possible to use it in the search pattern as it only represents a level of hierarchy in the name and not in the design loaded in memory.

The following example is based on Figure 2, page 38 which shows a purely hierarchical netlist.

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

The `-hierarchical` search is equivalent to manually performing a search at each level of the hierarchy, using the `current_instance` command to set the search scope to a particular hierarchical instance, and return all the objects that match the specified name pattern at that level. The following Tcl code, based on Figure 2, page 38, 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
}
```

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. For example, based on* Figure 2, *`get_cells -hierarchical -regexp B/.*` returns all the cells below the block B. See the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 1] for more information on -regexp.*

Using the -filter and -regexp Options

The Vivado Design Suite offers multiple ways to select a subset of objects with the `get_*` commands, using the `-filter` and/or `-regexp` options in conjunction with the `-hierarchical` option.

The table below summarizes the effect of the `-hierarchical/-filter/-regexp` options on the pattern that is provided to the command:

```
get_* [-hierarchical] [-filter] [-regexp] pattern
```

Table 1: **Effects of -hierarchical/-filter/-regexp options**

-hierarchical	-filter	-regexp	Result
			Pattern matches the local name of objects that belong to the current hierarchical level (current_instance).
Yes			*Pattern* matches the local name of objects that belong to the current hierarchical level (current_instance) and below.
	Yes		*Pattern* is a filtering expression applied to objects that belong to the current hierarchical level (current_instance). The NAME property matches against the full hierarchical name of the object.
Yes	Yes		*Pattern* is a filtering expression applied to objects that belong to the current hierarchical level (current_instance) and below. The NAME property matches against the full hierarchical name of the object.
		Yes	*Pattern* is a regular expression that matches the local name of objects that belong to the current hierarchical level (current_instance).
Yes		Yes	*Pattern* is a regular expression that matches the local name of objects that belong to the current hierarchical level (current_instance) and below.

Table 1: **Effects of -hierarchical/-filter/-regexp options**

-hierarchical	-filter	-regexp	Result
	Yes	Yes	Pattern is a filtering expression applied to objects that belong to the current hierarchical level (current_instance). The Name property matches against the full hierarchical name of the object. The matching expressions are based on the regular expression format.
Yes	Yes	Yes	Pattern is a filtering expression applied to objects that belong to the current hierarchical level (current_instance) and below. The Name property matches against the full hierarchical name of the object. The matching expressions are based on the regular expression format.

Note: The local name of an object refers to the part of the name that comes from the current hierarchical level (current_instance). It does not include the part of the name inherited from the parent.

Note: When some parts of the design have been flattened, the local name of objects that belonged to the flattened levels includes the hierarchical separator. In such case, the hierarchy separator in this part of the name is not seen any more as a separator, but just as a literal character.

Note: The `-filter` option specifies a filtering expression where the pattern matching inside the expression follows the global expression format. A filtering expression involves string matching based on object properties and can be as complex as required. When the `NAME` property is used inside a filtering expression, the full hierarchical name of the object is used for the string matching and not its local name. However, only objects that belong to the current hierarchical level (current_instance) are being considered. If `-hierarchical` is used with `-filter`, then all the objects that belong to the current hierarchical level and below are considered for the filtering. If `-regexp` is used with `-filter`, the pattern matching inside the expression follows the regular expression format.

Note: The `-regexp` option implies that the pattern provided to the command is a regular expression. Be careful as some characters, such as `* . [ ] +`, have a special meaning inside a regular expression. Those characters need to be escaped when used as literal and not as special characters in the context of the regular expression.

The string matching is case sensitive and always anchored to the start and end of the search string. To match a sub-string of a search string, use the following syntax depending on whether or not a regular expression is used:

- The pattern follows the format of a regular expression (`-regexp` only):
  `.*<substring>.*`
- The pattern follows the format of a global expression (other options):
  `*<substring>*`

Following are some examples based on the `cpu_hdl` project, which can be found under the Open Example Project link on the Getting Started page of the Vivado IDE, that illustrate the differences between the options.

- Change the current instance to `fftEngine/fftInst/ingressLoop[7].ingressFifo:`

```
vivado% current_instance fftEngine/fftInst/ingressLoop[7].ingressFifo
fftEngine/fftInst/ingressLoop[7].ingressFifo
```

- Get all the cells under the current instance: there is only one (hierarchical) cell:

```
vivado% get_cells
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo
vivado% get_cells -hier
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo/infer_fifo.two_rd_addr_reg
[8]_i_1__29 … (154 other cells)
```

- The local name of the cells under the current instance and below does not include `ingressLoop`. The string `ingressLoop` is inherited from the parent cell and is part of the full hierarchical name:

```
vivado% get_cells *ingressLoop*
WARNING: [Vivado 12-180] No cells matched '*ingressLoop*'.
vivado% get_cells *ingressLoop* -hier
WARNING: [Vivado 12-180] No cells matched '*ingressLoop*'.
```

- With the `-filter` option, the `NAME` property matches the full hierarchical names:

```
vivado% get_cells -filter {NAME =~ *ingressLoop*}
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo
vivado% get_cells -filter {NAME =~ *ingressLoop*} -hier
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo/infer_fifo.two_rd_addr_reg
[8]_i_1__29 … (154 other cells)
```

- Search for cells matching the pattern `*reg[*]*:`

```
vivado% get_cells *reg[*]*
WARNING: [Vivado 12-180] No cells matched '*reg[*]*'.
vivado% get_cells *reg[*]* -hier
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo/infer_fifo.wr_addr_reg[9]_
i_1__15 … (109 other cells)
vivado% get_cells -hier -regexp {.*reg\[.*\].*}
fftEngine/fftInst/ingressLoop[7].ingressFifo/buffer_fifo/infer_fifo.wr_addr_reg[9]_
i_1__15 … (109 other cells)
vivado% get_cells -hier -regexp {.*reg[.*].*}
WARNING: [Vivado 12-180] No cells matched '.*reg[.*].*'.
```

The last query, `get_cells -hier -regexp {.*reg[.*].*}` does not match any cell since the square brackets, `[]` have not been escaped. As a result, they are processed as special characters for the regular expression and not as literal square brackets in the cell name.

When a range of values needs to be specified in the filtering expression, the `-regexp` option should be used in addition to the `-filter` option. For example, the code below only gets the cells from `*reg[0]*` to `*reg[16]*`. The regular expression is built around matching `.*reg\[[0-9]\].*` or matching `.*reg\[1[0-6]\].*` :

```
vivado% get_cells -hierarchical -regexp -filter {NAME =~ ".*reg\[([0-9]||1[0-6])\].*"}
```

In another example, both commands below return all the tiles matching `CLB*X*Y*`, excluding the tiles from `CLB*X1Y*` to `CLB*X16Y*` with `X` between 1 and 16:

```
vivado% get_tiles -regexp -filter {NAME !~ "CLB.*X([1][0-6]|[0-9])Y.*" && TYPE=~ "CLB.*"}
vivado% get_tiles -regexp -filter {NAME !~ "CLB.*X1[0-6]Y.*" && NAME !~ "CLB.*X[1-9]Y.*" &&
TYPE=~ "CLB.*"}
```

Searching for Pins

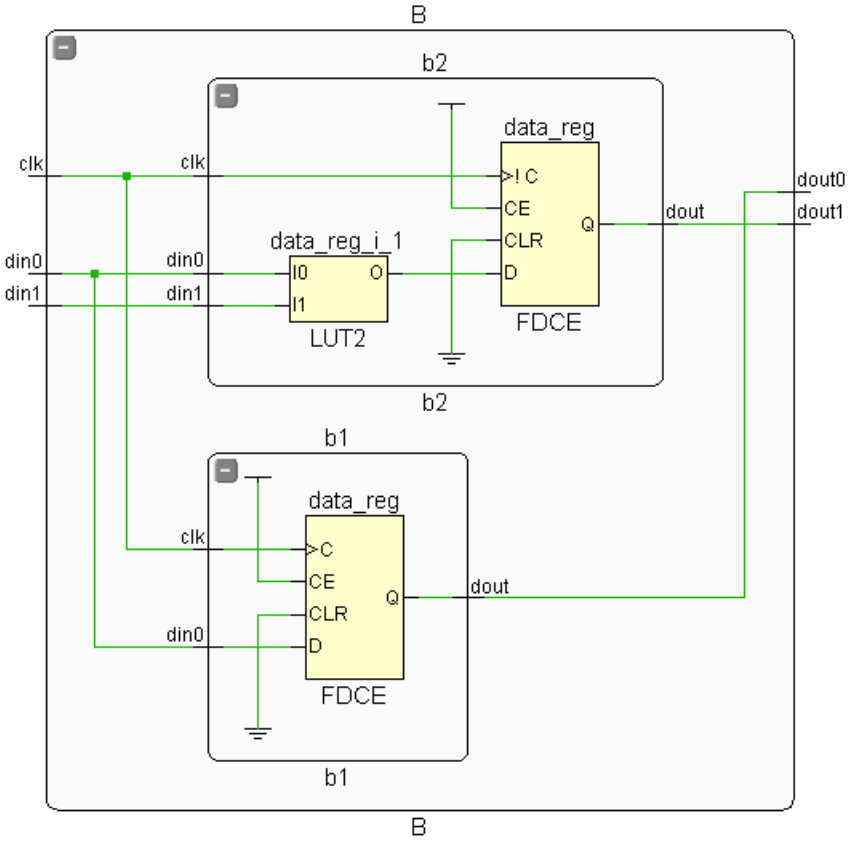

Figure 3: **Searching for Pin Names**

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 3.

```
# 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)
```

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 for example, 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.

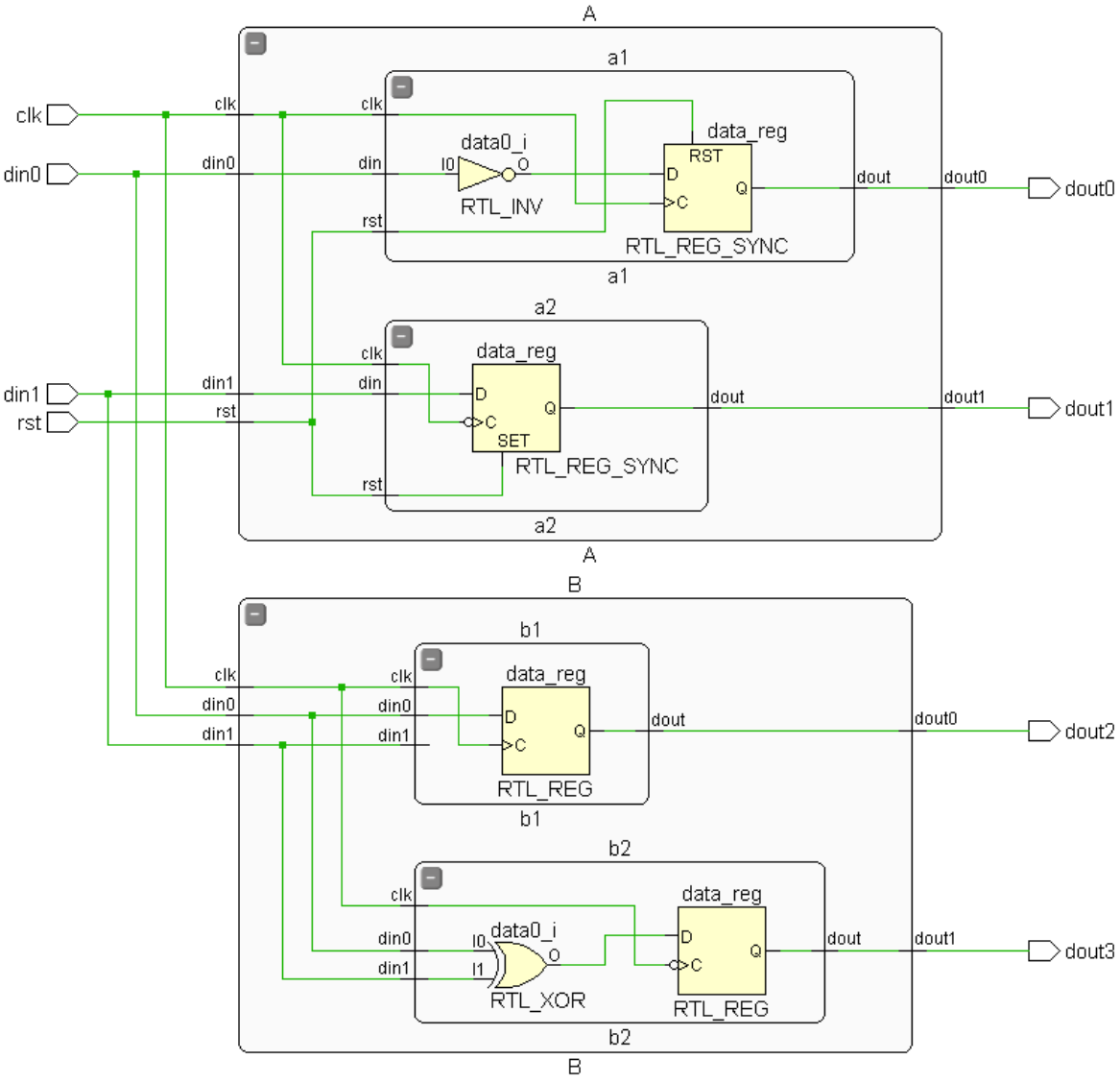

Figure 4: **Searching Hierarchical Designs**

1. `-hierarchical` and `-hier` refer to the same option. The Vivado Design Suite Tcl shell completes 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.

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 4, page 44, the following expressions return different results:

```
get_cells * ; # Returns 2 cells: A,B
get_cells -hier * ; # Returns all cells of the design (leaf and hierarchical)
get_cells -hier * -filter {NAME =~ */?1/*} ; # Returns 3 cells: A/a1/data0_i,
                                               # A/a1/data_reg, B/b1/data_reg
```

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 with a hierarchical name that begins with B/b*, 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} ]
```

IMPORTANT: *The NAME property contains the full hierarchical name of the object. When filtering on the NAME property, the pattern is evaluated against the complete NAME string, regardless of the other options used in the command, including `-hierarchical`.*

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:

```
set unLocedLeaf [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.

IMPORTANT: *The `filter` command does not modify the original Tcl variable and therefore the result must be saved inside another Tcl variable*

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 5, page 46 illustrates the relationship of objects in the in-memory design.

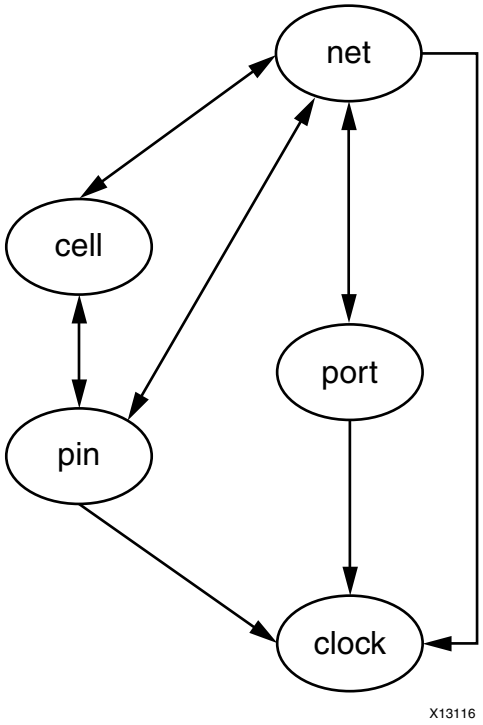

Figure 5: **Vivado Design Suite - Object Relationships**

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

The help message for each of the `get_*` commands that supports the `-of_objects` option 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.

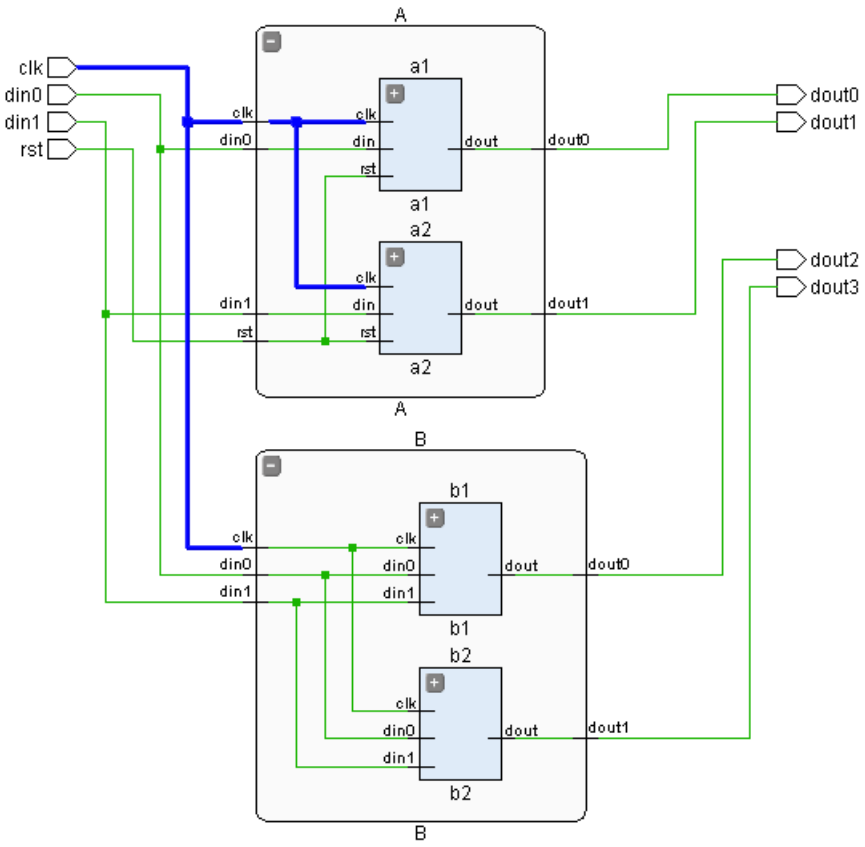

Figure 6: **Traversing Objects by Relationship**

Using Figure 6 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
```

get_nets Command

The `get_nets` command can return multiple representations of the same net as the net traverses through the design hierarchy. Below are some examples using the options provided with `get_nets` to ensure that the proper representation of the net is selected.

The following examples use the leaf pin of a LUT2 as defined below to query different segments of the net connected to the leaf pin.

```
set mypin [get_pins{egressLoop[7].egressFifo/buffer_fifo/infer_fifo.wr_addr_reg[9]_i_1__6/I0}]
mark_objects -color green $mypin
```

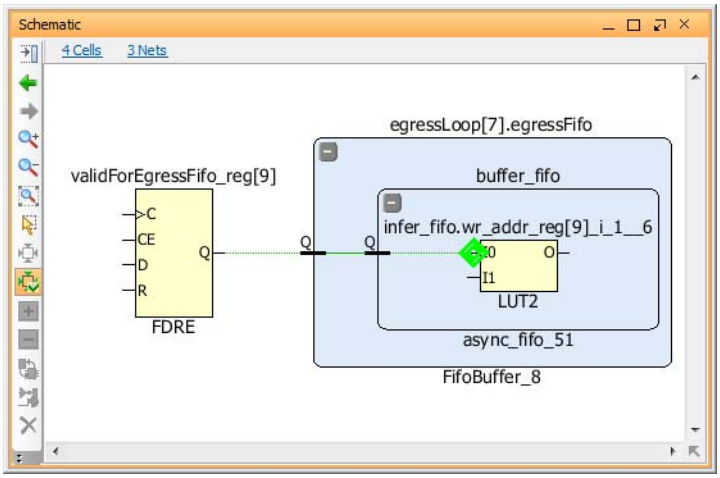

Figure 7: **set mypin example**

Simply getting the net connected to the leaf pin `mypin` returns the net segment within the hierarchy directly connected to the leaf pin.

```
select_objects [get_nets -of $mypin]
```

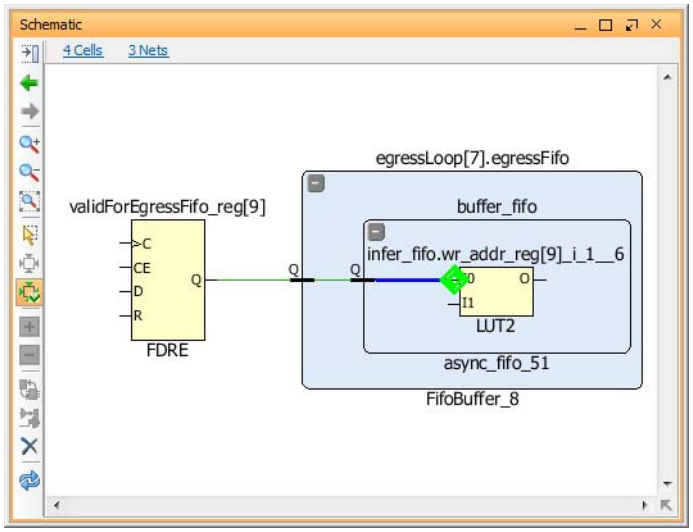

Figure 8: **get_nets of pin example**

In order to select all segments of the net connected to leaf pin `mypin`, use the `-segments` option.

```
select_objects [get_nets -segments -of $mypin]
```

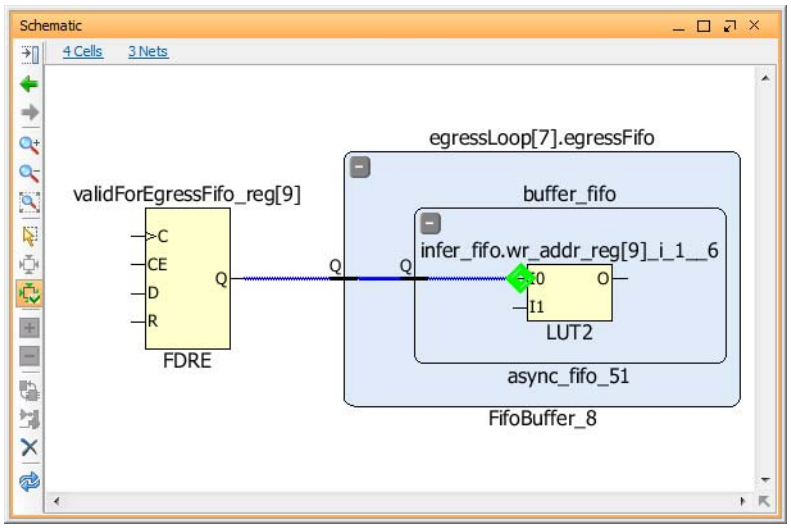

Figure 9: **get_nets segments example**

In order to only get the net segment at the highest level of the hierarchy connected to the leaf pin mypin, use the -top_net_of_hierarchical_group along with the -segments option. This is often useful when printing debug statements as this will provide the name of the net with the shortest number of characters.

```
select_objects [get_nets -top_net_of_hierarchical_group -segments -of $mypin]
```

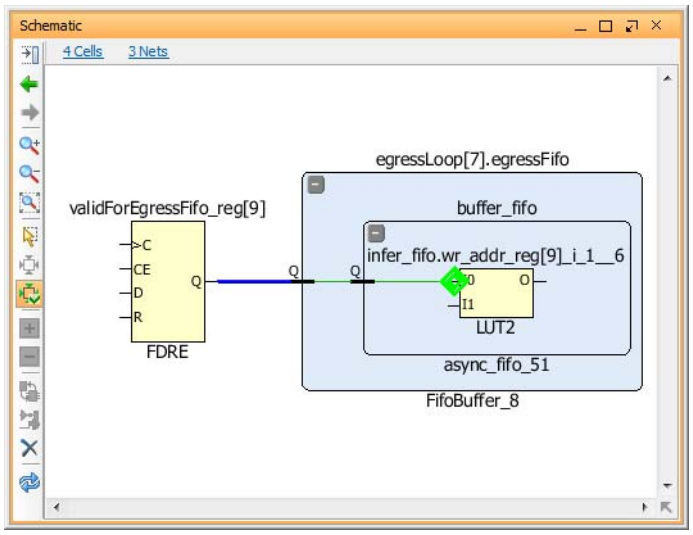

Figure 10: **get_nets top segments example**

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 (for example 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 (for example 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
...
```

Note: Although the `lappend` command is supported by the Tcl Console, it is not supported in a XDC constraint file with the `read_xdc` command. However, it is possible to convert a list built with the `lappend` command in a different way that is compatible with the `read_xdc` command. In the following example:

```
set myClocks {}
lappend myClocks [get_clocks CLK1]
lappend myClocks [get_clocks CLK2]
lappend myClocks [get_clocks CLK3]
```

The Tcl variable `myClocks` can be built differently to be compatible with the `read_xdc` command:

```
set myClocks [list [get_clocks CLK1] \
                   [get_clocks CLK2] \
                   [get_clocks CLK3] \
                ]
```

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 tools.

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 tools have 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 is written into the current working directory, or the directory from which the Vivado tools were 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 Vivado Design Suite Tcl console offers a number of commands for accessing files. See Accessing Files, page 53 for more information.

All `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 the command output to a variable 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 Vivado Design Suite Tcl console also offers many tool for working with strings. See Working with Strings, page 55 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 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 tools have 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]
```

Controlling Loops

Tcl has few built-in commands such as `for`, `foreach` and `while` that are used to loop or iterate through a section of code.

Their syntax is:

```
for start testCondition next body
foreach varname list body
while testCondition body
```

With all the above commands, the entire Tcl script body is executed at each iteration. However, Tcl provides two commands to change the control flow of the loop: `break` and `continue`.

The `break` statement is used to abort the looping command. The `continue` statement is used to jump to the next iteration of the loop.

Note: When running inside a proc, the loop can also be aborted using the `return` command. In this case, not only the loop is aborted but the control goes back to the caller of the proc.

For example, let us suppose that we have a file that contains a list of cell names with the format of one instance name per line. The sample code below reads this file and build a Tcl list that only includes the cell names that currently exist in the design. The code reuses the procedure `get_file_content` that was introduced earlier. If too many cell names are not found inside the current design then the code stops processing the content of the file:

```
set valid_cell_names [list]
set error 0
set max_errors 1000
foreach line [split [get_file_content ./all_cell_names.lst] \n] {
  if {[get_cells $line] == {}} {
    # cell name not found
    puts " Error - cell $line not found "
    incr error
    if {$error > $max_errors} {
      puts " Too many errors occured. Cannot continue "
      break
    } else {
      # Go to next cell name
      continue
    }
  }
  lappend valid_cell_names $line
}

puts " [llength $valid_cell_names] valid cells have been found "
```

Error Handling

Checking Validity of Variables

When developing Tcl scripts, it is recommended to always check for corner cases and for conditions where the code could fail. By doing the proper checks, it is possible to inform the user of issues and/or incorrect usage of the script. If the checks are not correctly performed, then the script could stop without informing the user about what went wrong and therefore what should be corrected.

Example 1: Check when a file is opened for read/write (unsafe version).

```
if {[file exists $filename]} {
  set FH [open $filename r]
  if {$FH != {}} {
    # The file is opened, do something
    # …
    close $FH
  } else {
    puts " File $filename could not be opened"
  }
} else {
  puts " File $filename does not exist"
}
```

Although the above script seems algorithmically correct, it would not work properly as the `open` command generates a low-level Tcl error (TCL_ERROR) that would stop the execution of the script in the event the file could not be opened. Later, in Example 3 we will see how this script can be improved.

Example 2: Check that the Vivado objects are valid after using the `get_*` commands.

```
proc get_pin_dir { pinName } {
 if {$pinName == {}} {
   puts " Error - no pin name provided"
   return {}
 }
 set pin [get_pins $pinName]
 if {$pin == {}} {
   puts " Error - pin $pinName does not exist"
   return {}
 }
 set direction [get_property DIRECTION $pin]
 return $direction
}
```

It is especially important to check that Vivado objects do exist after using the `get_*` commands when those objects are used inside other commands (`filter`, `get_*`, ...).

Handling Tcl Errors

Some built-in Tcl commands (but also user procs) can generate a Tcl error (TCL_ERROR) that can stop the execution of the script if the error is not caught by the program. An example is the `file` command that generates a TCL_ERROR when the file cannot be opened.

To do safe programming and catch the TCL_ERROR condition, Tcl has a built-in `catch` command that returns 1 when an error is caught and otherwise returns 0. The `catch` command can be used on a single command or a set of commands.

The basic syntax of the catch command is:

```
catch script [varname]
```

Where the `script` is a single or a set of Tcl commands and `varname` is a variable name in which the message explaining the TCL_ERROR is saved.

Note: The `varname` argument is optional

The catch command is often used as illustrated below:

```
If {[catch script errorstring]} {
  # A low-level TCL_ERROR happened
  puts " Error - $errorstring "
} else {
  # No TCL_ERROR was generated
  puts " The code completed successfully "
}
```

Example 1 can be made safer using the `catch` command to cover the case where the file cannot be opened:

Example 3: Check when a file is opened for read/write (safe version).

```
if {[file exists $filename]} {
  if {[catch { set FH [open $filename r] } errorstring]} {
    puts " File $filename could not be opened : $errorstring"
  } else {
    # The file is opened, do something
    # …
    close $FH
  }
} else {
  puts " File $filename does not exist"
}
```

A Tcl error can also be user generated using the `error` command. This can be used, for example, to propagate a TCL_ERROR that is caught with the `catch` command to the upper level (bubbling). However, the error command can be used as well to generate a TCL_ERROR when, for example, a corner case is not supported by the script or an unexpected condition has happened in the code.

For example, the proc below returns the file content or generates a TCL_ERROR when the file cannot be opened:

```
proc get_file_content { filename } {
  if {[catch {
    set FH [open $filename r]
    set content [read $FH]
    close $FH
  } errorstring]} {
    error " File $filename could not be opened : $errorstring "
  }
  return $content
}
```

The proc `get_file_content` can be called through a `catch` command to catch the potential error:

```
if {[catch { set content [get_file_content ./myreport.rpt] } errorstring]} {
  puts " The following error was generated: $errorstring "
} else {
  puts " The content of the file is: $content "
}
```

Example 2 can also be improved to generate a TCL_ERROR when a wrong condition occurs as shown in Example 4:

Example 4: Check that the Vivado objects are valid. Generate a TCL_ERROR otherwise (revised).

```
proc get_pin_dir { pinName } {
  if {$pinName == {}} {
    error " Error - no pin name provided"
  }
  set pin [get_pins $pinName]
  if {$pin == {}} {
    error " Error - pin $pinName does not exist"
  }
  set direction [get_property DIRECTION $pin]
  return $direction
}
```

Accessing Environment Variables

Tcl provides a convenient way to access the environment variables in a read-only mode through the Tcl global variable `env`. The variable `env` is a Tcl array that is automatically created and initialized at startup inside the Tcl interpreter.

Note: After initialization, any change to the `env` variable is not applied to the environment outside of the Tcl interpreter. Similarly, any change to your environment variables done after starting the Tcl interpreter will not be reflected by the `env` variable.

The keys of the `env` array are the environment variables at the time Vivado Design Suite starts. The keys are case sensitive.

For example:

```
Vivado% puts "The PATH variable is $env(PATH) "
```

To get the list of all the Unix environment variables:

```
Vivado%: set all_env_var [array names env]
```

It is possible to check if an environment variable exists (i.e a key to `env` array exists) by using the `info` command. For example to check for `MYVARNAME`:

```
Vivado% if {[info exists env(MYVARNAME)]} { … }
```

The `env` array is a global variable and can therefore be referenced inside a proc after being declared as global.

For example:

```
proc print_env {} {
  global env
  puts " UNIX Environment:"
  foreach var [lsort [array names env]] {
    puts "   $var : $env($var)"
  }
}
```

Creating Custom Design Rules Checks (DRCs)

The Vivado Design Suite lets you define and use custom design rule checks (DRCs) written in Tcl. To create custom DRCs use the following steps:

1. The basis of creating a custom DRC is a Tcl checker procedure that gets design objects of interest, or attributes of those design objects, and a 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`. Inside of the Tcl checker procedure, the `create_drc_violation` command is used to identify and flag violations when checking the rule against a design. The `create_drc_violation` command creates a violation object within the in-memory design, with properties that can be reported and further processed in the Vivado Design Suite.
2. The Tcl checker procedure is associated to a user-defined DRC that is created using the `create_drc_check` command. Call this rule by name when you run the `report_drc` command.
3. Optionally, you can elect to create a DRC rule deck by using the `create_drc_ruledeck` command. A DRC rule deck is a collection of DRCs. A mix of both user-created and predefined DRCs can be added to the user-created DRC rule deck by using the `add_drc_checks` command.
4. Checking the design against the design rules is performed by running the `report_drc` command. When running report_drc, you can specify to run either a DRC rule deck, user-defined design rule checks, or predefined DRCs.

Creating a Tcl Checker Procedure

The TCL checker procedure selects the design objects of interest to be checked. It then performs the necessary tests or evaluations of the design objects, 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` Tcl checker procedure which checks the width of the `WRITE_B` bus. This Tcl script file must be loaded into the Vivado tools prior

to running the report_drc command. Refer to Loading and Running Tcl Scripts, page 18 for more information on loading the Tcl checker procedure.

```
# 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
```

As you can see from the proc definition, the dataWidthCheck procedure accepts no arguments and can find everything it needs from the design. It creates an empty list variable, $vios, to store the violation objects returned by the create_drc_violation command.

The dataWidthCheck procedure iterates through all of the BRAMs in the design, and performs an evaluation of the WRITE_WIDTH_B property on each of those cells. If the WRITE_WIDTH_B of the BRAM cell exceeds a width of 36, a DRC violation is created with a specific message, $msg. The message contains a placeholder for the cell %ELG and 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. The create_drc_violation command supports messaging placeholders for netlist objects, clock regions, device sites, and package I/O banks by using their respective keys %ELG, %CRG, %SIG, and %PBG.

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

Creating a DRC Check

Once the Tcl checker procedure is defined, you must now define the DRC 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 requires you to provide a unique name for the user-defined rule check. This name that must match the name given to the violation created by the Tcl checker procedure. You will need to specify this unique name when adding the check to DRC rule decks or when running `report_drc`. In the `dataWidthCheck` Tcl checker procedure above, the `create_drc_violation` command uses the name `RAMW-1`. In addition, the `create_drc_check` command requires you to provide the procedure name of the Tcl checker procedure to be run when the rule is checked. In this case above, the Tcl checker procedure `dataWidthCheck` is provided as the `-rule_body` argument and must be loaded into the Vivado Design Suite prior to running the `report_drc` command.

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

You can optionally group the DRC 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 checker procedure is passed upward to the DRC object. In this case, any message defined by `create_drc_violation` is simply passed through to the DRC report.

The DRC 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 DRC from being used when `report_drc` is run.

Creating a DRC Rule Deck

You can optionally group multiple related DRC checks that can be run together into a DRC rule deck. To do this, you must first create the DRC rule deck by using the `create_drc_ruledeck` command. Once the DRC rule deck is created, DRCs can be added and removed from the DRC rule deck by using the `add_drc_checks` and `remove_drc_checks` commands. Mixing user-defined checks and predefined checks into a single DRC rule deck is allowed in the Vivado Design Suite. Below is an example of

creating a DRC rule deck called `myrules` along with the addition and removal of DRCs from the DRC rule deck.

```
create_drc_ruledeck myrules
add_drc_checks -ruledeck myrules {RAMW-1 RAMW-2 RAMW-3}
remove_drc_checks {RAMW-2} -ruledeck myrules
```

Note: If the `is_enabled` property of the DRC is set to FALSE, then the DRC will not be run as part of the DRC rule deck when running `report_drc`. In some cases, it might be more desirable to disable the DRC than to remove it from the DRC rule deck.

Reporting Custom DRCs

A user-defined DRC can be run individually, with other rules, or as part of a DRC rule deck using the `report_drc` command. Below are examples of running the previously defined `RAMW-1` rule individually, with other rules, and as part of the previously created DRC rule deck.

```
report_drc -rules {RAMW-1}
report_drc -rules {RAMW-1 RAMW-2}
report_drc -ruledeck myrules
```

Remember that the `is_enabled` property of the rule check must be set to TRUE in order for `report_drc` to run the check.

DRC Explanation Script

There are times when the designer has a DRC rule name or a pattern of DRC rules and wants to get an explanation about what these rules are doing. This can be done by reporting properties on the DRC objects.

The example script below takes as input a pattern matching a set of DRC rule(s) and prints some explanation (severity and description) for each rule. If the pattern does not match any rule then an error message is issued.

```
proc explain_drc { drcs } {
  package require struct::matrix
  set loop_drcs [get_drc_checks $drcs]
  if {$loop_drcs == {}} {
    puts "  Error: $drcs does not match any existing DRC rule"
    return
  }
 struct::matrix drcsm
  drcsm add columns 3
  drcsm add row {DRC_ID SEVERITY DESCRIPTION}
  foreach drc $loop_drcs {
   set description "\{[get_property NAME [get_drc_checks $drc]]\}"
   set severity "\{[get_property SEVERITY [get_drc_checks $drc]]\}"
   set key "\{[get_property KEY [get_drc_checks $drc]]\}"
   drcsm add row "$key $severity $description"
  }
```

```
  puts "[drcsm format 2chan]";
  drcsm destroy
}
```

There are a number of Tcl packages embedded inside Vivado and this example script uses the `struct::matric` package to format the summary table.

The following are some sample outputs from the explain_drc proc:

```
Vivado% explain_drc CFGBVS-1
DRC_ID   SEVERITY DESCRIPTION
CFGBVS-1 Warning  Missing CFGBVS and CONFIG_VOLTAGE Design Properties

Vivado% explain_drc CFGBVS-*
DRC_ID   SEVERITY         DESCRIPTION
CFGBVS-1 Warning          Missing CFGBVS and CONFIG_VOLTAGE Design Properties
CFGBVS-2 Critical Warning CFGBVS Design Property
CFGBVS-3 Warning          CONFIG_VOLTAGE Design Property
CFGBVS-4 Critical Warning CFGBVS and CONFIG_VOLTAGE Design Properties
CFGBVS-5 Critical Warning CONFIG_VOLTAGE Design Property
CFGBVS-6 Critical Warning CONFIG_VOLTAGE with HP Config Banks
CFGBVS-7 Warning          CONFIG_VOLTAGE with Config Bank VCCO
Vivado% explain_drc foo
Error: foo does not match any existing DRC rule
```

Tcl Scripting Tips

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

Performance via Nesting

When a Tcl command is executed from the Tcl console, the command is first being processed at the level of the Tcl interpreter. If there is no syntax error, the command is then executed at the C++ level. If the command returns a value, the C++ code sends the returned value to the Tcl interpreter through some layers of software. This layering back and forth between the Tcl interpreter and the low-level C++ code has some runtime penalty. However, when nesting is used within the same command, nested commands are directly called from the C++ code. The C++ code only returns to the Tcl interpreter once the whole command has been completed.

For example, the this code:

```
set nets [get_nets -hier]
set pins [get_pins -of_objects $nets]
```

is slower than this code:

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

This is because the first code sample creates an intermediate Tcl variable, `nets`.

However, it might sometimes be preferable to create intermediate Tcl variables if the results of these variables can be reused in other parts of the code.

Caching Objects

Objects or lists of objects should be cached in Tcl variables so they can be reused later.

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 tools 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 tools 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 tools 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 tools commands that support the `-return_string` option use:

```
findCmd return_string
```

Writing Efficient Code

One way to improve runtime is to write code efficiently such that a container is built and a command runs on the entire container versus running the command within a loop on each item that would be part of the container. Below is an example similar to that seen in the Creating Custom Design Rules Checks (DRCs) section to illustrate this.

Inefficient Code:

```
foreach bram [get_cells -hier -filter {PRIMITIVE_SUBGROUP == bram}] {
   set bwidth [get_property WRITE_WIDTH_B $bram]
   if { $bwidth > 36} {
      highlight_object -color red [get_cells $bram]
   }; # End IF
}; # End FOR
```

Efficient Code:

```
foreach bram [get_cells -hier -filter {PRIMITIVE_SUBGROUP == bram}] {
   set bwidth [get_property WRITE_WIDTH_B $bram]
   if { $bwidth > 36} {
      lappend bram_list $bram
   }; # End IF
}; # End FOR
highlight_object -color red [get_cells $bram_list]
```

An even more compact and efficient way to code this is to apply the filter as part of the `get_cells` command. This removes the need to perform a `foreach` loop with individualized checking at the expense of a slightly more complicated filter.

```
highlight_object -color red [get_cells -hier -filter {PRIMITIVE_SUBGROUP == bram &&
WRITE_WIDTH_B > 36}]
```

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

1. *Vivado Design Suite Tcl Command Reference Guide (UG835)*
2. *Vivado Design Suite User Guide: Getting Started* (UG910)
3. *Vivado Design Suite User Guide: Using Constraints* (UG903)
4. *Vivado Design Suite Migration Guide (UG911)*
5. *Vivado Design Suite User Guide: Design Flows Overview (UG892)*
6. *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)*
7. Vivado Design Suite 2013.3 Documentation (www.xilinx.com/support/documentation/dt_vivado_vivado2013-3.htm)
8. Tcl Developer Xchange www.tcl.tk