Chapter 5: **Additional Reading Materials**

DC-Tcl



DC-Tcl



Tcl = Tool Command Language (tickle):

- DC-Tcl is the command interface for DC in XG mode
- Built on the "open" industry-standard shell programming language Tcl
- DC-Tcl an interpreted scripting language

- Many Synopsys tools support Tcl for consistency, e.g. Design Compiler, Formality, PrimeTime, Physical Compiler and more.
- Tcl was originally developed by John K. Ousterhout at UC Berkeley.
- There are many books on the topic of Tcl programming, here are
 - Tcl and the Tk Toolkit, John K. Ousterhout
 Practical Programming in Tcl and Tk, Brent B. Welch
- . Some Tcl web sites for reference and further information:
 - www.tcl.tk (documentation and advanced packages for Tcl, same as www.scriptics.com)
 - www.tclforeda.net (many DC script examples and other useful tools for Logic Designers)

Converting from dcsh to DC-Tcl



A program is avilable for users to migrate from "old" dcsh to DC-Tcl.

UNIX% dc-transcript my_script.scr my_script.tcl

- · Will convert most commands in existing scripts to Tcl
- Only goes from DCSH to DC-Tcl
- Called from the UNIX prompt

• The dc-transcript utility accurately translates most existing dcsh mode scripts.

- The dc-transcript does not do the following:
 - Does not check the syntax of your dcsh mode scripts, although serious syntax errors will stop the translation
 - Does not, in general, check the semantics of your commands
 - Does not optimize your scripts
 - Does not, in general, teach you how to write Tcl
 - Does not always update your dcsh mode commands current and preferred Tcl mode commands

Executing DC-Tcl Scripts



- Commands can be typed:
 - Interactively in DC Tcl

dc_shell-xg-t> echo "Running my.tcl..." dc_shell-xg-t> source -echo -verbose my.tcl

Executed in batch mode

UNIX% dc_shell-xg-t -f my.tcl | tee -i my.log

The tee command displays the results on the screen and writes them into the specified log file.

Tcl Basics

- Tcl command =
 - One or more words separated by white space
 - First word is command name, others are arguments
 - Returns string result
- Tcl script =
 - Sequence of commands
 - Commands are separated by newlines and/or semicolons

Examples:

Variable Substitution

- Syntax: \$varName
- Variable name is letters, digits, underscores
- Substitution may occur anywhere within a word:

Sample commands	Results
set b 66	66
set a b	b
set a \$b	66
set a \$b+\$b+\$b	66+66+66
set a \$b.3	66.3
set a \$b4	no such variable

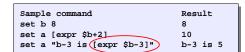
To remove a variable, use the command unset, example:

- unset b
- Variables can be concatenated with strings in many ways, e.g. to get the contents of the variable b concatenated with the string "test", you type:
 - set a \${b}test -> "66test"
- Variables do not need declaration as in languages like C, Pascal, etc., since there is only one "type" of variable – a string. The string may be interpreted in different ways by the command itself, e.g. the expr command (shown later) may interpret the string as an integer or as a floating point number.



Nested Commands

- Syntax: [commands...]
- Evaluate command, return result
- May occur anywhere within a word:





Command substitution produces: set a "b-3 is 5" Then, the command "set" is executed

Note: "expr" is a Tcl function that performs math operations

Defining Words

- Words end or break at white space and semicolons, except:
 - Double-quotes prevent breaks

```
set a "x is $x; y is $y"
```

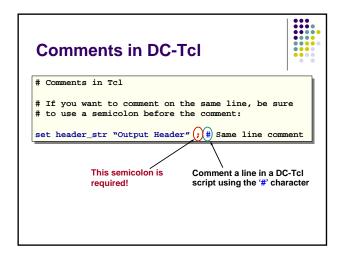
- Curly braces prevent breaks and substitutions set a {[expr \$b*\$c]}
- Backslashes escape special characters

```
set a word\ with\ \$\ and\ space
```

 Backslashes can escape newline (linecontinuation)

```
report_constraint \
    -all_violators
```

```
set x 3
set y 5
set a "x is $x; y is $y"; #Sets the variable a to "x is 3; y is 5"
set a {[expr $b*$c]}; #Sets the variable a to "[expr $b*$c]"
set a word\ with\ \$\ and\ space; #Sets variable a to "word with $ and space"
report_constraint \ all_viclators
    Make sure that there is no space after the backslash. "Line-continuation" means "backslash - newline."
Notice that a \+newline is evaluated as a space. e.g.
set a "1 2\ 3 4"
sets a to "1 2 3 4" - with a space between the 2 and the 3!
```



Using Wildcards

- DC-Tcl supports two wildcard characters:
 - * will match zero to 'n' characters
 - ? matches exactly 1 character

Examples:

Arithmetic Expressions

To evaluate arithmetic expressions use the expr command.

- To have the result of expr represented as a floating point number, at least one of the numbers involved in the calculation has to be a float. The number 7 becomes 7.0 if floating point is required.
- . e.g. the command:
 - expr 5/2
 - will return 2.
- If a floating point answer is required, use:
 - expr 5.0/2
 - This will return 2.5

```
Using Lists in DC-Tcl

Arrange your data as lists, example:

dc_shell-xg-t> set colors {red green blue} red green blue dc_shell-xg-t> echo $colors red green blue dc_shell-xg-t> set Num_of_Elements [llength $colors] 3 dc_shell-xg-t> set colors [lsort $colors] blue green red

dc_shell-xg-t> set link_library {*}

*
dc_shell-xg-t> lappend link_library tc6a.db opcon.db

* tc6a.db opcon.db
dc_shell-xg-t> echo $link_library

* tc6a.db opcon.db
```

```
• To manipulate lists, use Tcl built-in list commands:
                          Concatenates two lists and returns a new list
  concat
                           Joins elements of a list into a string
   join
   lappend
                           Creates a new list by appending elements to a list
   lindex
                           Returns a specific element from a list
                          Creates a new list by inserting elements into a list
Returns a list formed from its argument
  linsert
   list
   llength
                           Returns the number of elements in a list
   lrange
                           Extracts elements from a list
   lreplace
                           Replaces a specific range of elements in a list
                           Searches a list for a regular expression
  lsearch
                           Sorts a list
   lsort
  split
                          Splits a string into a list
```

Iterate through Lists The following example iterates over a list: set all_colors "red green blue" foreach color \$all colors { echo \$color is a nice color... red is a nice color ... green is a nice color... blue is a nice color ...

Objects and Attributes



- Recall that designs consist of objects:
 - Designs, cells, ports, pins, clocks, and nets
- In order to keep track of circuit functionality and timing, DC attaches many attributes to each of these objects:
 - · Ports can have the following attributes direction driving_cell max_capacitance others
 - Designs can have the following attributes

operating_conditions_max area

others... max area

Accessing the Synopsys Database



- Access to DC objects in DC-Tcl is achieved through collections - a DC extension to standard Tcl
- Collections are generally created by get_ or all commands:

Example:

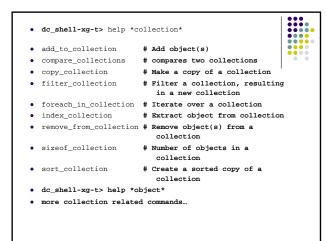
```
get_ports clk*
set myclocks [all_clocks]
set hi_cap_pins [get_pins
busdriver/tristate*]
```

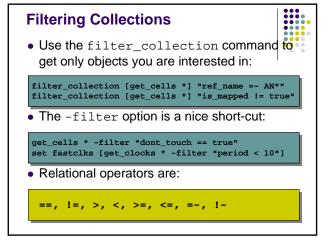
```
    Partial list of get_* and all_* commands:

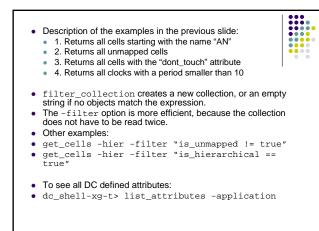
                            # Create a collection of cells
# Create a collection of clocks
   get_cells
get clocks
                              Create a collection of designs
   get_designs
   get libs
                            # Create a collection of libraries
   get_nets
                            # Create a collection of nets
   get pins
                            # Create a collection of pins
   get_ports
                            # Create a collection of ports
   all_clocks
                            # Create a collection of all_designs
# Create a collection of all_inputs
   all designs
   all_inputs
                            # Create a collection of all_outputs
# Create a collection of all_registers
   all outputs
   all_registers
   When these commands are issued, DC internally creates a group of objects, along with all their attributes.
```

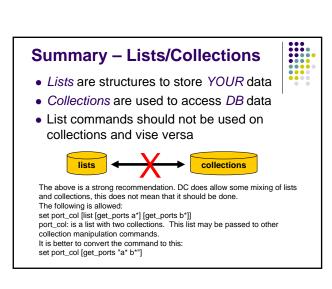
```
Collections Are Referenced by a Handle
          Just like lists, collections have
           special access commands.
 dc shell-xg-t> set foo [get ports p*]
 {"pclk", "pframe_n", "pidsel", "pad[31]"...}
 dc_shell-xg-t> sizeof_collection $foo
 dc_shell-xg-t> query_objects $foo
 {"pclk", "pframe_n", "pidsel", "pad[31]"...}
Collection commands return a collection handle, NOT a list!
A list, containing the names of all the objects returned by the get_ or all_
command is echoed to the screen.
Standard Tcl list commands (concat, llength, etc) will not work with the output of a
```

```
Manipulating Collections
  dc shell-xq-t> help *collection*
  add_to_collection
                       # Add object(s)
  remove_from_collection # Remove object(s) from a
                          collection
 dc_shell-xg-t> set pci_ports [get_ports "DATA*"]
 dc_shell-xg-t> set pci_ports [add_to_collection \
                     $pci_ports [get_ports CTRL*]]
 dc_shell-xg-t> set all_inputs_except_clk \
              [remove_from_collection [all_inputs] \
                [get_ports CLK]]
```









Avoid using aliases and abbreviating command names in scripts Use common extensions: e.g. foo.tcl Use full option names in commands: create_clock -period 5 [get_ports clk] Avoid "snake scripts" "Snake scripts" are scripts that call scripts, that call scripts: Very hard to debug. Avoid sourcing scripts from your synopsys_dc.setup file, since these scripts will be executed automatically every time you start the tool.

