Vivado Design Suite ユーザー ガイド

Tcl スクリプト機能の使用

UG894 (v2013.4) 2013 年 12 月 18 日

XILINX®

Notice of Disclaimer
The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products.To the maximum extent permitted by applicable law:(1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same.Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications.You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent.Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx.Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications:http://www.xilinx.com/warranty.htm#critapps.

© Copyright 2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries.All other trademarks are the property of their respective owners.

本資料は英語版 (v2013.4) を翻訳したもので、内容に相違が生じる場合には原文を優先します。
資料によっては英語版の更新に対応していないものがあります。
日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com までお知らせください。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	改訂内容
2012 年 11 月 16 日	2012.3	初版
2012 年 12 月 18 日	2012.4	「Tcl フック スクリプトの定義」を追加
2013 年 4 月 22 日	2013.1	次のセクションをアップデートまたは追加 • 「ヘルプ情報の入手」 • 「プロジェクト フローでのコンパイル」 • 「-hierarchical オプションの使用」 • 「DRC 説明スクリプト」 • 「制約セットでの Tcl スクリプトの使用」 • 「ループの制御」 • 「エラー処理」 • 「環境変数へのアクセス」 • 「その他のリソース」
2013 年 6 月 19 日	2013.2	次のセクションを追加 • 「ローカル変数とグローバル変数」 • 「DRC チェックの作成」 • 「DRC ルール デックの作成」
2013 年 10 月 2 日	2013.3	次のセクションを追加または更新 • 「get_nets コマンド」 • 「Tcl スクリプトの記述」 • 「Tcl プロシージャの定義」 • 「コマンド ライン引数の解析」 • 「プロシージャの名前空間」 • 「テンプレート スクリプト」
2013 年 12 月 18 日	2013.4	技術的なアップデートなし。再リリースのみ。

目次

改訂履歴 3
概要 6
Tcl の概要 7
ヘルプ情報の入手 9
コンパイルおよびレポート生成のスクリプト例 11
非プロジェクト フローでのコンパイル 11
プロジェクト フローでのコンパイル 13
Tcl スクリプトの読み込みと実行 17
Tcl スクリプトの初期化 17
Tcl スクリプトの読み込み 17
制約セットでの Tcl スクリプトの使用 18
Tcl フック スクリプトの定義 18
GUI のカスタマイズ 20
Tcl スクリプトの記述 20
Tcl プロシージャの定義 20
コマンド ライン引数の解析 23
ローカル変数とグローバル変数 26
プロシージャの名前空間 27
テンプレート スクリプト 30
デザイン オブジェクトへのアクセス 32
名前を使用したオブジェクトの取得 33
関連性を使用したオブジェクトの検索 41
get_nets コマンド 43
オブジェクトのリストの処理 46
出力先の指定 47
ファイルへのアクセス 48
文字列の操作 50
ループの制御 52
エラー処理 53
変数の有効性チェック 53
Tcl エラーの処理 54
環境変数へのアクセス 56
カスタム デザイン ルール チェック (DRC) の作成 57
Tcl チェッカー プロシージャの作成 57
DRC チェックの作成 58
DRC ルール デックの作成 59
カスタム DRC のレポート 59
DRC 説明スクリプト 59
Tcl スクリプト記述のヒント 60
ネスト処理を使用したパフォーマンスの向上 60
オブジェクトのキャッシュ 61
オブジェクト名と NAME プロパティ 61
オブジェクトのリストのフォーマット 62
Vivado Tcl コマンドをオプションで検索 62
効率的なコードの記述 63

付録 A：その他のリソース
ザイリンクス リソース 64
ソリューション センター 64
参考資料 64

Vivado での Tcl スクリプト

概要

Tcl (ツール コマンド言語) は、さまざまなデザイン ツールおよびデザイン データにアクセスするための、変数、プロシージャ、制御構造を含むインタープリター型プログラミング言語です。

注記 : 詳細は、『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) [参照 1] を参照するか、「`<command> -help`」と入力してください。

注記 : Vivado® Design Suite の起動方法および使用方法の詳細は、『Vivado Design Suite ユーザー ガイド : 入門』(UG910) [参照 2] を参照してください。

この言語は新しい関数呼び出しで簡単に拡張することができ、1990 年代初期に開発されてから、新しいツールやテクノロジをサポートするため拡張されてきています。多くの EDA ベンダーが標準の API (アプリケーション プログラミング インターフェイス) としてアプリケーションを制御および拡張するために導入しています。

ザイリンクスでは、Vivado Design Suite のネイティブ プログラミング言語として Tcl を導入しているので、この業界標準言語に精通している設計者であれば簡単に取り入れ、理解することができます。Vivado Design Suite の Tcl インタープリターは、アプリケーションの制御、デザイン オブジェクトおよびプロパティへのアクセス、カスタム レポートの作成を実行するための、Tcl 言語の機能と柔軟性を提供しています。Tcl を使用すると、デザインの特定の要件に合わせてデザイン フローを変更できます。

Tcl 言語には、ローカル ファイル システムのファイルに対して読み出しおよび書き込みを実行するビルトイン コマンドが含まれます。これにより、動的にディレクトリを作成し、FPGA デザイン プロジェクトを開始して、プロジェクトにファイルを追加したり、合成およびインプリメンテーションを実行できます。デザイン プロジェクトからデバイス リソースの使用率や QoR (結果の質) に関するカスタマイズ レポートを生成し、企業内で共有できます。

また、Tcl 言語を使用して、新しい設計手法を試したり、既存の問題を回避したり、必要に応じてデザイン オブジェクトの挿入および削除、プロパティの変更を実行できます。デザイン フローの確立された部分を再実行するためのスクリプトを記述し、プロセスを標準化できます。

このガイドで説明する Tcl コマンドおよびスクリプト例のほとんどは、Vivado Design Suite 特定のものです。Vivado 特定の Tcl コマンドの詳細は、『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) [参照 1] を参照するか、または Vivado ツールのヘルプ機能を使用してください。

Vivado IDE では、ザイリンクス デザイン制約 (XDC) を使用してデザイン制約を指定します。XDC は Vivado で使用可能な Tcl コマンドのサブセットに基づいており、Tcl と同様に解釈されます。XDC コマンドには、主にタイミング制約、物理制約、オブジェクト クエリ、およびいくつかの Tcl ビルトイン コマンド (`set`、`list`、および `expr`) があります。XDC コマンドの完全なリストは、『Vivado Design Suite ユーザー ガイド : 制約の使用』(UG903) [参照 3] の付録 B を参照してください。Tcl スクリプトとは異なり、XDC ファイルは Vivado IDE で管理されるので、グラフィカル インターフェイスでの制約の変更は元の XDC ファイルに保存されます。そのため、XDC ファイルでは XDC コマンドのみを使用できます。制約の記述にほかの Tcl コマンドを使用する必要がある場合は、Tcl スクリプトを使用する必要があります。

Vivado ツールでは、`vivado.jou` というジャーナル ファイルが Vivado を起動したディレクトリに作成されます。ジャーナル ファイルにはセッション中に実行された Tcl コマンドが記録されるので、このファイルから新しい Tcl スクリプトを作成できます。

ログ ファイル (`vivado.log`) も作成され、実行されたコマンドの出力が含まれます。ジャーナル ファイルとログ ファイルは、実行されたコマンドおよびその結果を確認するのに有益です。

Vivado Design Suite にビルトインされている Tcl インタープリターにより、追加の Tcl コマンドが提供されています。Tcl ビルトイン コマンドについては、Tcl のオープン ソース ベースおよび資料を管理している Tcl Developer Xchange サイト (http://www.tcl.tk) を参照してください。

Tcl プログラミング言語の入門チュートリアルは、http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html [参照 8] を参照してください。また、Tclers Wiki (http://wiki.tcl.tk) にいくつかのサンプル スクリプトがあります。

このガイドでは、Tcl コマンドおよび Tcl スクリプトの例、および Vivado Design Suite での戻り値を示します。これらのコマンド例とその戻り値は、次の形式で記述されています。

- Tcl コマンドおよびスクリプト例

  `puts $outputDir`

- Tcl コンソールへの出力または Tcl コマンドの結果

  ./Tutorial_Created_Data/cpu_output

Tcl の概要

Tcl スクリプトは、改行またはセミコロンで区切られた一連の Tcl コマンドです。Tcl コマンドは、スペースまたはタブで区切られた単語の文字列です。Tcl インタープリターはコマンド ラインを単語に分割し、必要に応じてコマンドおよび変数置換を実行します。コマンド ラインは左から右に読み込まれ、各単語が完全に評価されてから次の単語が評価されます。コマンドおよび変数置換は、左から右に実行されます。

単語は、1 つの単語または中かっこ ({}) またはダブルクォーテーション (“”) で囲まれた複数の単語です。中かっこまたはダブルクォーテーション内のセミコロン、中かっこ、タブ、スペース、改行は、通常の文字として処理されますが、バックスラッシュ (\) はこの後説明するように、中かっこまたはダブルクォーテーション内でも特殊文字として処理されます。

最初の単語はコマンドとして扱われ、その後の単語は引数としてコマンドに渡されます。

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

この例では、最初の単語は Tcl `set` コマンドで、値を割り当てるために使用します。2 番目の単語は変数名 (`outputDir`)、3 番目の単語は変数値 (`./Tutorial_Created_Data/cpu_output`) として `set` コマンドに渡されます。

単語にバックスラッシュ (\) が使用されている場合、Tcl インタープリターによりバックスラッシュ置換が実行されます。ほとんどの場合、バックスラッシュの次の文字は標準文字として処理されます。これを使用して、ダブルクォーテーション、中かっこ、ドル記号などの特殊文字を文字列に追加できます。Tcl インタープリターでバックスラッシュ文字がどのように処理されるかは、Tcl/Tk のリファレンスを参照してください。

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

中かっことダブルクォーテーション マークの使用法も異なります。中かっこ内の文字に対しては、置換は実行されません。中かっこ内の単語や文字列はそのまま処理され、変数またはコマンド置換のために価されません。次の例に示すように、単語は中かっこに囲まれたそのままの文字列 (中かっこは含まない) となります。ダブルクォーテーションに囲まれた文字列は評価され、変数およびコマンド置換が必要に応じて実行されます。ダブルクォーテーションに囲まれた文字列に対してコマンド置換、変数置換、およびバックスラッシュ置換が実行されます。

```
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
```

上記の例で、ダブルクォーテーションを使用した 2 番目の例では `[version -short]` コマンドが戻り値で置換されていますが、中かっこを使用した 1 番目の例では置換が実行されていないことに注目してください。文字列を囲む場合には、このことに注意してダブルクォーテーションまたは中かっこを選択してください。

変数に値を代入するには、set コマンドを使用します。変数を参照するには、変数名の前にドル記号 ($) を付けて指定します。単語がドル記号で開始している場合、Tcl インタープリターで変数置換が実行され、変数が現在その変数に保存されている値に置換されます。Tcl 言語では、$ は予約語です。

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

角かっこ ([ ]) を使用すると、コマンド内にコマンドをネストできます。ネストされたコマンドは、左から右にボトムアップに評価されます。角かっこ内の文字列が新しい Tcl スクリプトとして反復的に処理されます。ネストされたコマンドに、さらにコマンドをネストさせることもできます。ネストされたコマンドの結果がその上位のコマンドに渡されてから、その上位のコマンドが処理されます。

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

上記の例では、現在のデザインの最上位にあるセル オブジェクトがアルファベット順に並べ替えられ、そのリストが listCells 変数に代入されます。まず get_cells コマンドが実行され、返されたオブジェクトが lsort コマンドで並べ替えられて、並べ替えが終了したリストが変数に代入されます。

ただし Vivado Design Suite では、角かっこの処理は標準の Tcl と多少異なります。角かっこは Verilog および VHDL の名前 (ネット、インスタンスなど) では標準文字として処理され、通常はバスやインスタンスの配列など、ベクターの 1 つまたは複数の要素を示します。Vivado ツールでは、角かっこがネットリスト オブジェクト名の一部である場合はボトムアップに評価されません。

次の 3 つのコマンドは同等です。

```
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\[*\] ]
```

1) では、外側の角かっこは標準の Tcl と同様にコマンドのネスト ([get_pins]) を表しますが、内側の角かっこは Vivado ツールでは指定したオブジェクト名の一部として処理されます (transformLoop[0])。Vivado Design Suite ではこれが自動的に処理されますが、一部の文字に限られます。これらの文字は次のいずれかの形式にする必要があり、それ以外の場合は角かっこは標準の Tcl と同様に評価されます。

- アスタリスク [*] : 任意の数のビットまたはインスタンスを示すワイルドカードです。
- 整数 [12] : 特定のビットまたはインスタンスを指定します。
- ベクター [31:0] : 特定の範囲のビットまたはインスタンスのグループを指定します。

2) では、中かっこを使用して内側の角かっこ内の文字列がコマンド置換されないようにしており、オブジェクト名の一部として処理されます (transformLoop[0])。

3) では、バックスラッシュを使用して角かっこを特殊文字でなく標準文字として評価するよう指定しており、コマンド置換は実行されません。

2) および 3) は角かっこが適切に処理されるようにする方法を示していますが、中かっこまたはバックスラッシュを手動で追加する必要があります。1) は、これが Vivado Design Suite で自動的に処理されることを示しています。

Tcl スクリプトにコメントを追加するには、行を # で開始します。# の後に続く次の改行までの文字は、無視されます。行の最後にコメントを追加するには、次の例に示すように、コマンドの最後にセミコロン (;) を記述し、その後に # を追加してコメントを記述します。

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

ヘルプ情報の入手

Tcl コンソールでヘルプ情報を取得できます。すべての Vivado コマンドで -help オプションがサポートされており、コマンド ラインの任意の位置で使用できます。

次に例を示します。

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

また、help コマンドを使用してもヘルプ情報を表示できます。help コマンドでコマンド名を指定すると (help <*command*>)、<*command*> -help を使用した場合と同じ情報が表示されます。

```
Vivado% help create_clock
```

help コマンドで -args オプションを使用すると、引数の簡単な説明のみを表示できます。

```
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
```

また、-syntax オプションを使用すると、コマンド構文のみを表示できます。

```
Vivado% help create_clock -syntax

create_clock

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

help コマンドを使用すると、特定のコマンドのヘルプ情報だけでなく、コマンドのカテゴリおよびプロジェクトのクラスに関する情報も表示できます。カテゴリのリストを取得するには、help コマンドを引数またはオプションを使用せずに実行します。次に、コマンド カテゴリの一部を示します。

```
Vivado% help

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

各カテゴリのコマンドのリストを取得するには、-category オプションを使用します。たとえば、次のコマンドを実行すると、Tools カテゴリのすべてのコマンドが表示されます。

```
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
```

コンパイルおよびレポート生成のスクリプト例

非プロジェクト フローでのコンパイル

次に、非プロジェクト デザイン フローを定義する Tcl スクリプトの例を示します。

このサンプル スクリプトでは reportCriticalPaths というカスタム コマンドが使用されており、Vivado Design Suite にカスタム コマンドやプロシージャを追加できることを示しています。reportCriticalPaths の内容は、20 ページの「Tcl プロシージャの定義」を参照してください。

```
# 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
```

サンプル スクリプトの詳細

上記のサンプル スクリプトは、次の段階から構成されています。

- **手順 1** : 変数 `$outputDir` を定義して出力ディレクトリを指定し、そのディレクトリを実際に作成します。`$outputDir` 変数は、スクリプトで必要に応じて参照されます。
- **手順 2** : デザインを記述する VHDL および Verilog ファイルと、デザインの物理制約およびタイミング制約を含む XDC ファイルを読み込みます。合成済みネットリスト (EDIF または NGC) を読み込む場合は、`read_edif` コマンドを使用します。

  Vivado Design Suite では、デザイン制約を使用してデザインの物理特性およびタイミング特性を定義します。`read_xdc` コマンドは XDC 制約ファイルを読み込み、読み込まれた制約ファイルが合成およびインプリメンテーションに適用されます。

重要 : Vivado Design Suite では、UCF フォーマットはサポートされません。UCF 制約を XDC コマンドに移行する方法は、『Vivado Design Suite 移行手法ガイド』(UG911) [参照 4] を参照してください。

  `read_*` Tcl コマンドは、非プロジェクト モードで使用し、Vivado Design Suite でディスク上のファイルを読み込んでメモリ内にデザイン データベースを構築します。ファイルがコピーされたり、プロジェクト モードでのようにファイルの依存関係が作成されることはありません。非プロジェクト モードでのすべての操作は、Vivado ツール内のインメモリ データベースに対して実行されます。そのため、非プロジェクト モードは非常に柔軟ですが、ユーザーがソース デザイン ファイルの変更を管理し、それに応じてデザインをアップデートする必要があります。プロジェクト モードまたは非プロジェクト モードを使用した Vivado Design Suite の実行に関する詳細は、『Vivado Design Suite ユーザー ガイド : デザイン フローの概要』(UG892) [参照 5] を参照してください。

- **手順 3** : デザインを指定のターゲット デバイス用に合成します。

  HDL デザイン ファイルをコンパイルし、XDC ファイルに含まれるタイミング制約を適用し、ロジックをザイリンクス プリミティブにマップして、メモリ内にデザイン データベースを作成します。Vivado ツールをバッチ モードで実行している場合でも、Tcl シェル モードで対話的に Tcl コマンドを実行している場合でも、グラフィカル モードでデザイン データを Vivado 統合設計環境 (IDE) で表示している場合でも、メモリ内のデザインは Vivado ツール内に存在します。

  合成が終了したら、チェックポイントを保存します。この時点では、デザインはタイミング制約および物理制約が適用された未配置の合成済みネットリストです。タイミングやリソース使用率など、さまざまなレポートを作成すると、デザインを理解するのに有益です。

  このサンプル スクリプトでは、`reportCriticalPaths` というカスタム コマンドを使用して、TNS、WNS、違反を CSV ファイルにレポートします。これにより、クリティカルなパスをすばやく特定できます。

  合成後に `read_xdc` または `source` コマンドを使用して読み込まれた XDC ファイルは、インプリメンテーションにのみ適用されます。それらのファイルは、その後デザイン チェックポイントを保存した場合にネットリストと共に保存されます。

- **手順 4** : 配置配線の準備としてロジック最適化を実行します。最適化の目的は、ターゲット パーツの物理リソースに配置する前にロジック デザインを簡略化することです。最適化後、タイミング ドリブン配置を実行します。

  各手順の後、reportCriticalPaths コマンドを実行して新しい CSV ファイルを生成します。デザインの異なる段階からの複数の CSV ファイルを使用すると、カスタム タイミング サマリ スプレッドシートを作成でき、インプリメンテーションの各段階でタイミングがどのように向上したかを理解するのに役立ちます。

  配置が完了したら、get_timing_paths コマンドを使用して配置済みデザインのワースト タイミング パスの SLACK プロパティを取得します。report_timing コマンドを使用すると、ワースト スラックを含むタイミング パスの詳細なテキスト形式レポートが生成されますが、get_timing_paths コマンドを使用すると、同じタイミング パスが Tcl オブジェクトとして、パスの主なタイミング特性に対応するプロパティと共に返されます。SLACK プロパティは指定したタイミング パス (この例の場合はワースト パス) のスラックを返します。スラックが負の場合、物理最適化を実行して、配置タイミング違反の解決を試みます。

  手順 4 の最後にチェックポイントを保存し、デザインのタイミング サマリとデバイス使用率をレポートします。これにより、配線前と配線後のタイミングを比較し、配線のタイミングへの影響を評価できます。

- **手順 5** : タイミング ドリブン配線を実行し、チェックポイントを保存します。これでメモリ内のデザインが配線されたので、追加のレポートを生成して、消費電力、デザイン ルール違反、最終的なタイミングに関する重要な情報を入手できます。レポートはファイルに出力するか、Vivado IDE に表示して確認できます。その後、タイミング シミュレーション用に Verilog ネットリストをエクスポートします。
- **手順 6** : デザインをザイリンクス FPGA にプログラムするビットストリームを生成します。

プロジェクト フローでのコンパイル

次に、プロジェクト フローでデザインを合成し、ビットストリーム生成までのインプリメンテーションを実行するスクリプト例を示します。この例では、Vivado インストール ディレクトリにある CPU サンプル デザインを使用しています。

```
#
# 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!"
```

サンプル スクリプトの詳細

- **手順 1**：`create_project` コマンドでプロジェクトを作成します。プロジェクト ディレクトリおよびターゲット デバイスが指定されています。指定したプロジェクト ディレクトリが存在しない場合は、自動的に作成されます。

  この例では、さまざまなレポートを保存する出力ディレクトリは、プロジェクト ディレクトリと同じです。

- **手順 2**：プロジェクトで使用されるすべてのファイルを宣言し、プロジェクトに追加します。これには、`add_files` コマンドを使用します。ファイルをプロジェクトに追加すると、特定のファイルセットに追加されます。ファイルセットは、目的別にファイルをグループ化するコンテナーです。このスクリプト例では、ほとんどのファイルはデフォルトのファイルセット(`sources_1`) に追加されますが、Verilog テストベンチ `cpu_tb.v` のみはデフォルトのシミュレーション ファイルセット `sim_1` に追加されます。

  ファイルは、`import_files` コマンドを使用してプロジェクト ディレクトリにもコピーします。これにより、プロジェクトでソース ファイルのローカル コピーが使用され、元のソース ファイルは参照されません。

- **手順 3**：バックグランドで合成 run を起動し、デザインを合成します (`launch_run synth_1`)。Vivado IDE により必要なスクリプトがすべて自動的に生成され、別の Vivado セッションで合成が実行されます。合成 run は別のプロセスで実行されるので、現在のスクリプトを続行する前に合成 run が完了するのを待つ必要があります。これには、`wait_on_run` コマンドを使用します。

  合成 run が完了したら、`open_run synth_1` コマンドを使用して結果をメモリに読み込むことができます。制約のないチェックポイントが、合成を実行したプロジェクト ディレクトリに保存されます。この例では、チェックポイントは次のディレクトリに保存されます。

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

  注記：合成 run のデフォルト名は `synth_1`、インプリメンテーション run のデフォルト名は `impl_1` です。`create_run` コマンドを使用して、追加の run を作成できます。

- **手順 4**：`launch_run` コマンドを使用してインプリメンテーションを実行します。配置前の最適化からビットストリーム生成までの完全な配置配線フローを、1 つのコマンドで実行できます。このスクリプト例では、ビットストリーム生成までのインプリメンテーションが実行されます (`launch_run impl_1 -to_step write_bitstream`)。

`STEPS.PHYS_OPT_DESIGN.IS_ENABLED` プロパティにより、オプションの `phys_opt_design` がイネーブルになっています。ユーザー定義の条件によりインプリメンテーション コマンドをダイナミックに呼び出すことができる非プロジェクト フローとは異なり、プロジェクト フローの run は実行する前に設定する必要があります。そのため、非プロジェクト フローの例とは異なり、この例では配置後のタイミング スラック値をチェックせずに物理ロジック最適化がイネーブルに設定されています。

run の Tcl フック プロパティ STEPS.<STEPNAME>.TCL.PRE および STEPS.<STEPNAME>.TCL.POST を使用して、各インプリメンテーションの前後にさまざまなレポートを生成しています。これらのプロパティを使用すると、run 構造を使用したときにフローで Tcl スクリプトをいつ実行するかを指定できます。詳細は、18 ページの「Tcl フック スクリプトの定義」を参照してください。

インプリメンテーション run は別の Vivado セッションで実行されるので、Tcl 変数およびプロシージャをスクリプトで使用するには、それらをそのセッションで初期化する必要があります。これには、いくつかの方法があります。

- **方法 1** : Tcl 変数およびプロシージャを init.tcl で定義します (17 ページの「Tcl スクリプトの初期化」を参照)。この方法で変数およびプロシージャを定義すると、すべての プロジェクトおよびセッションに適用されます。
- **方法 2** : 変数およびプロシージャを含む Tcl スクリプトを、run で使用する制約セットに追加します。デザインをメモリに読み込んだときに、制約の一部として読み込まれます。
- **方法 3** : STEPS.OPT_DESIGN.TCL.PRE で変数およびプロシージャを含む Tcl スクリプトを設定します。このスクリプトは、OPT_DESIGN をイネーブルにした場合にのみ読み込まれます。デフォルトでは、true に設定されています。

先ほど示したスクリプト例では、方法 3 を使用しています。変数およびプロシージャを含む Tcl スクリプトは、インプリメンテーション段階で次のように指定されています。

```
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]
```

インプリメンテーション run は、コンパイル Tcl スクリプトが実行されるディレクトリとは異なるプロジェクトのサブディレクトリで実行されるので、Tcl スクリプトは絶対パスで指定する必要があります。

- 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
  ```

  最初の 2 行では、インプリメンテーション run の後の方のいくつかのスクリプトで使用される変数およびプロシージャを初期化します。次の 3 行では、タイミング レポートと使用率レポートを生成します。インプリメンテーションのはじめにタイミング解析を実行し、配置配線で使用されるタイミング制約をチェックし、大きな違反がないことを確認することが推奨されます。reportCriticalPaths レポートは、デザインのワースト パスに関する詳細を示します。この Tcl プロシージャについては、20 ページの「Tcl プロシージャの定義」で詳細に説明します。

- post_opt_design.tcl

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

  outputDir 変数および reportCriticalPaths プロシージャは、同じ Vivado セッションの run で既に読み込まれている pre_opt_design.tcl で定義されているので、このスクリプトでは定義する必要はありません。

  opt_design の後にもタイミング レポートと使用率レポートを生成することが推奨されます。

- post_place_design.tcl

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

配置後、クロック リソースの使用率およびデバイスでの位置を確認できます。フローの後の方では解決できない大きなタイミング違反を検出するため、タイミング解析を実行することが推奨されます。

- 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
```

配置後と同様、フローのこの段階でもタイミング レポートを確認することが重要です。

- 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
```

配線後のタイミング解析では、配線済みの実際のネット遅延が使用されるので、タイミングの最終確認のため確認する必要があります。配線ステータス レポートには、配線問題の数が示されます。配線問題がある場合、DRC レポートを生成するとそれらの問題を特定するのに役立ちます。

注記：上記のスクリプトで配線後に生成される Tcl レポートのほとんどは、run でも自動的に生成されます。また、プロジェクト フローを使用している場合は、フローの各段階の後にデザイン チェックポイントも自動的に生成されるので、スクリプトで write_checkpoint コマンドを呼び出す必要はありません。すべてのチェックポイントとデフォルトのレポートは、インプリメンテーション run ディレクトリにあります。

```
./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
```

インプリメンテーション run が完了したら、open_run impl_1 コマンドを使用してインプリメント済みデザインをメモリに読み込むことができます。

Tcl スクリプトの読み込みと実行

Vivado Design Suite では、デザイン セッション中に Tcl スクリプトを読み込んで実行するのに複数の方法があります。ツールを起動したときにスクリプト ファイルが自動的に読み込まれるようにするか、Tcl コマンド ラインで source コマンドを使用して読み込むか、Vivado IDE のメニューに追加します。

Tcl スクリプトの初期化

Vivado Design Suite で Tcl スクリプトが自動的に読み込まれるようにするには、`init.tcl` ファイルで定義します。この方法は、新しいコマンドを定義する Tcl プロシージャを記述し、Vivado のすべてのセッションで使用できるようにする場合に有益です。

Vivado ツールを起動すると、次の 2 箇所で Tcl 初期化スクリプトが検索されます。

1. ツールのインストール ディレクトリ : `<installdir>/Vivado/version/scripts/init.tcl`
2. ローカルのユーザー ディレクトリ :
   a. Windows 7 : `%APPDATA%/Roaming/Xilinx/Vivado/init.tcl`
   b. Linux : `$HOME/.Xilinx/Vivado/init.tcl`

<installdir> は Vivado Design Suite のインストール ディレクトリです。

`init.tcl` が両方の場所で見つかった場合、まず Vivado ツールのインストール ディレクトリにあるファイルが読み込まれ、次にホーム ディレクトリにあるファイルが読み込まれます。

インストール ディレクトリにある `init.tcl` ファイルを使用すると、企業またはデザイン グループのすべてのユーザーに対して共通の初期化スクリプトをサポートできます。そのインストール ディレクトリから Vivado ツールを起動すると、共通の `init.tcl` スクリプトが使用されます。

ホーム ディレクトリにある `init.tcl` ファイルを使用すると、各ユーザーがそれぞれコマンドを追加したり、デザイン要件を満たすためにツールのインストール ディレクトリに含まれるコマンドを変更できます。

この `init.tcl` スクリプトは標準の Tcl スクリプト ファイルで、Vivado ツールでサポートされるどの Tcl コマンドも含めることができます。`source` コマンドを追加して、`init.tcl` から別の Tcl スクリプト ファイルを読み込むこともできます。

Tcl スクリプトの読み込み

`source` コマンドを使用すると、Tcl スクリプト ファイルを Vivado ツールに手動で読み込むことができます。

```
source <filename>
```

<filename> はファイル名とファイルの相対パスまたは絶対パスを指定します。パスをファイル名の一部として指定しない場合は、現在の作業ディレクトリまたは Vivado Design Suite ツールを起動したディレクトリにファイルが作成されます。

Vivado IDE で Tcl スクリプトを読み込むには、[Tools] → [Run Tcl Script] をクリックします。

デフォルトでは、ファイルの各行が Tcl コンソールに表示されます。表示されないようにするには、`-notrace` オプションを使用します。これは、Vivado Tcl インタープリターに特有のオプションです。

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

制約セットでの Tcl スクリプトの使用

Tcl スクリプトは、通常の XDC ファイルと同様に、プロジェクトの制約セットに追加できます。ただし、XDC ファイルはツールで管理されますが、Tcl スクリプトはツールで管理されません。Tcl スクリプトで定義された制約がツールで変更されても、Tcl スクリプトに自動的には保存されません。変更を保存するには、メモリの制約をすべてファイルにエクスポートし、このファイルを使用してスクリプトを手動でアップデートする必要があります。デザインをメモリで開くと (`open_run`)、XDC ファイルの後に Tcl スクリプトが読み込まれます。これは、非プロジェクト フローで `read_xdc` を使用して XDC ファイルを読み込んだ後に Tcl スクリプトを実行するのと同等です。制約セットでの XDC ファイルおよび Tcl スクリプトの使用については、『Vivado Design Suite ユーザー ガイド : 制約の使用』(UG903) [参照 3] を参照してください。

Tcl フック スクリプトの定義

非プロジェクト フローでは、`synth_design` コマンド実行の前後など、フローのどの時点でも Tcl スクリプトを読み込むことができます。プロジェクト ベース フローでも、Vivado IDE を使用するか、`set_property` コマンドを使用して合成 run またはインプリメンテーション run にプロパティを設定することにより、これを実行できます。Tcl フック スクリプトを使用すると、合成 run またはインプリメンテーション run、あるいはインプリメンテーションの任意の段階の前 (`tcl.pre`) および後 (`tcl.post`) にカスタム Tcl スクリプトを実行できます。

合成 run またはインプリメンテーション run を起動すると、定義済みの Tcl スクリプトが使用され、選択したストラテジに基づいて標準デザイン フローが実行されます。Tcl フック スクリプトによりこの標準フローをカスタマイズできます。任意の段階で Tcl スクリプトを実行できるので、有益です。デザイン フローの各段階の前後でフック スクリプトを実行できます。一般的に、次のような使用法があります。

- カスタム レポート : タイミング、消費電力、リソース使用率、またはユーザー定義の Tcl レポート
- フローの一部でのみタイミング制約を変更
- ネットリスト、制約、またはデバイス プログラムの変更

GUI では、デザイン run を右クリックして [Change Run Settings] をクリックすると、Tcl フック スクリプトを指定できます。詳細は、『Vivado Design Suite ユーザー ガイド : デザイン フローの概要』(UG892) [参照 5] の「run の作成および管理」を参照してください。[Design Run Settings] ダイアログ ボックスに、Tcl フック スクリプトを指定する [`tcl.pre`] と [`tcl.post`] オプションがあります (図 1)。

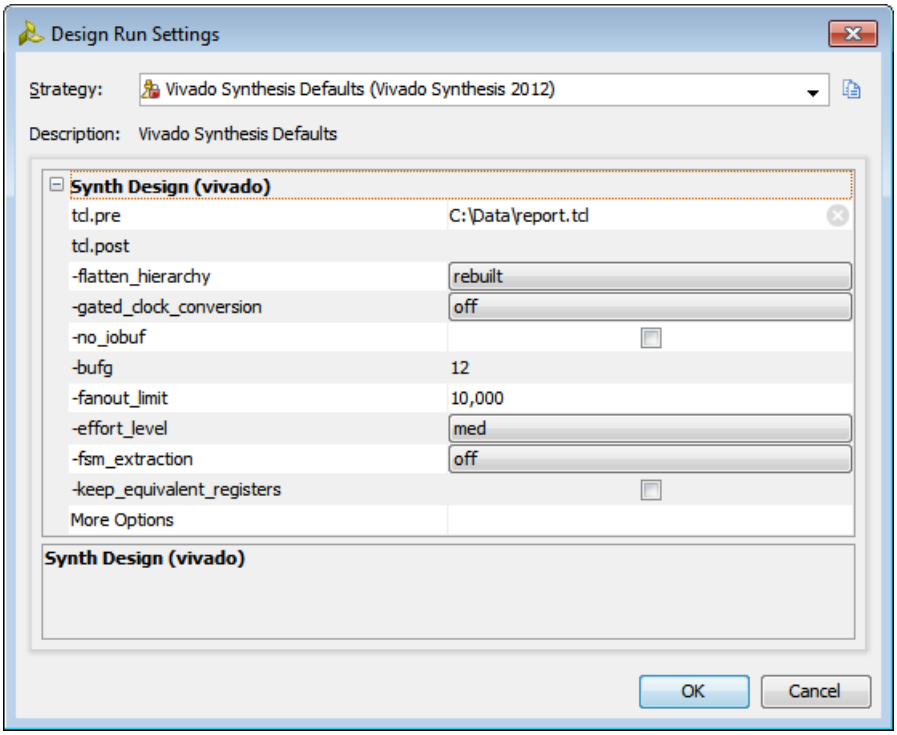

図 1 : Tcl フック スクリプトの指定

合成 run またはインプリメンテーション run にプロパティが設定され、run の前 (`tcl.pre`) または後 (`tcl.post`) に実行するスクリプトが指定されます。Tcl コンソールまたは Tcl スクリプトの一部として、合成 run またはインプリメンテーション run に直接このプロパティを設定することも可能です。

合成 run に設定するプロパティは、次のとおりです。

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

たとえば、合成前に `report.tcl` スクリプトを実行するには、次のように設定します。

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

インプリメンテーション run では、インプリメンテーション プロセスの各段階 (最適化、消費電力最適化、配置、配置後の消費電力最適化、物理最適化、配線、ビットストリーム生成) の前後に Tcl スクリプトを実行できます。これらのプロパティは、次のとおりです。

```
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
```

重要：`tcl.pre` および `tcl.post` スクリプト内のパスは、プロジェクトの関連する run ディレクトリ `<project>/<project.runs>/<run_name>` を基準とします。現在のプロジェクトまたは現在の run の DIRECTORY プロパティを使用して、Tcl フック スクリプト内の相対パスを定義できます。

```
set_property DIRECTORY [current_project]
set_property DIRECTORY [current_run]
```

GUI のカスタマイズ

[Tools] → [Custom Commands] → [Customize Commands] を使用して、Vivado IDE のメイン メニューおよびツールバーにシステムまたはユーザー定義の Tcl コマンドを追加できます。カスタム コマンドをメニューに追加する方法は、『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [参照 6] の「カスタム メニュー コマンドの追加」を参照してください。

Tcl スクリプトの記述

Tcl スクリプトを記述する際は、ユーザーの使いやすさに焦点を置く必要があります。つまり、ヘルプやインタラクティブ コマンド ライン引数を提供するなど、Vivado に組み込まれているコマンドと同じように使用できるようにするのが理想的です。`get_*` コマンドを使用した後に Vivado オブジェクトのが空かどうかなど、まれな状況もすべて考慮する必要があります。Tcl コードを記述する際は、コードで使用される下位プロシージャを作成するのも一般的です。プロシージャとグローバル変数の名前の競合を回避するには、独自の名前空間内でコードを記述し、名前の競合を最小限に抑えるようにすることをお勧めします。

Tcl プロシージャの定義

Vivado Design Suite には、完全な Tcl インタープリターがビルトインされており、新しいカスタム コマンドやプロシージャを簡単に作成できます。Tcl スクリプトを記述して Vivado IDE から読み込んで実行したり、プロシージャを記述して、引数を取り、エラーをチェックして、結果を返す新しい Tcl コマンドとして使用できます。

Tcl プロシージャは `proc` コマンドで指定します。プロシージャ名、引数のリスト、実行するコードの本文を引数として指定します。次に、プロシージャ定義の簡単な例を示します。

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

ヒント：このプロシージャの定義では引数は 1 つなので中かっこで囲む必要はありませんが、中かっこを使用することでプロシージャ定義がわかりやすくなります。引数が複数ある場合は、中かっこは必須です。

通常プロシージャでは、定義済みの引数と、オプションでデフォルト値を指定します。引数にデフォルト値がある場合、その前の必須の引数がすべて指定されていれば、プロシージャを呼び出したときにその引数を指定する必要はありません。プロシージャは、return コマンドを使用して値を返すよう指定していない場合、空の文字列を返します。次の例では、3 つの定義済み引数を持つ reportWorstViolations というプロシージャを定義しています。

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

プロシージャを実行する際、次の例に示すように、すべての引数を指定する必要があります。

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

次の例では、同じプロシージャで 3 つの引数のうち最後の 2 つにデフォルト値があります。corner のデフォルト値は Slow、delayType のデフォルト値は Max です。プロシージャの定義でデフォルト値が設定されているので、プロシージャを呼び出す際は corner および delayType 引数の指定はオプションです。

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

このプロシージャを実行する際は、次のすべての呼び出し方法が有効です。

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

次のプロシージャの例には必須の引数 nbrPath がありますが、それ以外にも追加の引数を指定できます。この場合、プロシージャを定義する際に引数のリストとして Tcl キーワード args を使用します。args キーワードは、任意の数の要素 (0 を含む) を含む Tcl リストを示します。

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

Tcl コマンドを実行する際、Tcl コマンドで使用可能なまたは必須のコマンド ライン引数の代わりに変数置換を使用できます。この場合、Tcl eval コマンドを使用してコマンドの一部として Tcl 変数を含めたコマンド ラインを評価する必要があります。上記の例では、引数のリスト変数 ($args) が report_timing コマンドに変数として渡されるので、eval コマンドが必要です。

プロシージャを実行する際は、次のいずれの形式でも機能します。

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

最初の例では、値 2 が $nbrPaths 引数に渡され、-max_paths に適用されます。2 番目と 3 番目の例では、それぞれ 1 と 10 が -max_paths に適用され、その後の文字列は $args に代入されます。

次の例は、非プロジェクト モードのサンプル スクリプトで使用されていた reportCriticalPaths コマンドを示します。このプロシージャでは 1 つの引数 $filename が使用され、コメントで各セクションを説明しています。

```
#-----------------------------------------------------------------------
# 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
```

コマンド ライン引数の解析

外部パラメーターまたは引数を使用するプロシージャを記述すると、プロシージャの使用範囲が広がり、重複したコードを記述する必要性を軽減できるので、有益な場合や、これが必要な場合があります。1 つのプロシージャで複数のコンテキストを処理できるようにすると、重複したコードを含む複数のプロシージャと同じ範囲のコンテキストを 1 つのプロシージャで網羅でき、使用および管理が簡単になります。

これは、プロシージャをインタラクティブに使用する場合に特に有益です。一部のコマンド ライン オプションをほかの Vivado コマンドと同じように指定できると、ユーザーにとって便利です。

Tcl では、これを `args` 変数を使用して簡単に適用できます。プロシージャの引数リスト内で使用される `args` キーワードは、任意の数の要素 (0 を含む) を示します。`args` 変数は、ほかの Tcl リストと同様に処理および解析可能な Tcl リストです。

コマンド ライン引数を解析する方法は複数あります。次に、その 1 つの例を示します。

```
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 xxxxxxxxxxxxxxx
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  }
```

説明 :

1. 行 1 ～ 6 : リストの最初の要素を削除するプロシージャ `lshift` を定義します。
2. 行 9 : 複数の要素を指定可能な 1 つの引数 `args` を使用する `myproc` を定義します。このコード例では、`myproc` は `-ports` <*string*>、`-verbose`、`-help` の 3 つのコマンド ライン オプションをサポートします。
3. 行 19 ～ 44 : すべてのコマンド ライン引数をループします。すべての引数が処理されると、`args` 変数は空になります。
4. 行 20 : 処理が必要なコマンド ライン引数を `flag` 変数に保存します。`lshift` プロシージャを使用して、`args` 変数から引数を取得および削除します。
5. 行 21 ～ 43 : flag 変数の内容を、有効なすべての引数に対してチェックします。switch 文には `-exact` オプションが使用されており、`flag` の内容が完全なオプション名に対してチェックされます。たとえば、ポートを定義するには、`-p` または `-ports` を指定する必要があります。

   `-p`/`-ports` オプションではコマンド ライン引数が指定され、`lshift` args (行 24) により読み出され、削除されます。

   `-v`/`-verbose` オプションはブール値で、`args` からの引数は必要ありません (行 28)。

   行 31 ～ 33 : `-h`/`-help` オプションをチェックします。

   行 36 ～ 38 : 「-」で開始するすべてのコマンド ライン引数をチェックします。このプロシージャ例では、サポートされません。

   行 39 ～ 40 : 「-」で開始しないすべてのコマンド ライン引数をチェックします。このプロシージャ例では、サポートされません。

6. 行 46 ～ 64 : -h/-help が指定されている場合に、組み込まれているヘルプ情報を表示します。プロシージャにヘルプ情報を組み込む必要がない場合は、これらの行と行 30 ～ 33 は削除できます。
7. 行 68 ～ 70 : エラーが発生していないかをチェックします。通常、引数が有効であるかをチェックする追加コードは、行 68 より前に記述する必要があります。指定されたコマンド ライン オプションが互換していないなど、エラーが発生している場合は、エラー変数がインクリメントされ、行 69 が実行されます。
8. 行 73 以降 : 追加のコードを記述します。

上記のコードはコマンド ライン引数を解析し、サポートされているオプションと完全に一致しているかを調べていますが (行 21)、コマンド ライン引数が完全に一致しているかを調べるよりも、条件式に一致しているかを調べる方が有益な場合もあります。これには、行 21 で -exact オプションの代わりに -glob オプションを使用します。次に例を示します。

```
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  }
```

行 22、26、および 30 では、ワイルドカードとしてアスタリスク (*) が使用されています。この場合、-pfoo など、-p で開始するすべての文字列が、ポートを定義する有効なコマンド ライン オプションとなります。

上記のプロシージャ例 myproc はインタラクティブ コマンドとして機能しますが、引数の解析のためランタイム オーバーヘッドがあります。何回も呼び出される下位プロシージャでは、ランタイム オーバーヘッドが問題となることがあります。プロシージャにコマンド ライン引数を追加するのに別の方法を使用して、ランタイム オーバーヘッドを削減できます。これには、Tcl 配列にコマンド ライン引数のリストを割り当てます。ただし、各コマンド ライン オプションに 1 つの引数しか使用できません。次に例を示します。

```
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  }
```

説明 :

1. 行 1 : 複数の要素を指定可能な 1 つの引数 `args` を使用して `myproc2` を定義します。`args` は後ほど Tcl 配列を設定するのに使用されるので、引数の数は偶数にする必要があります。
2. 行 3 : さまざまなオプションのデフォルト値を指定します。各オプションには、1 つの引数のみ指定できます。
3. リストのフォーマットは、次のとおりです。

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

4. 行 5 : Tcl 配列 `options` をデフォルト値で初期化します。
5. 行 7 : `args` 変数の値でデフォルト値を置き換えます。
6. 行 9 ～ 10 : 各オプションの値を `$options(<option>)` で読み出します。次のコードを使用して、オプションが存在するかをチェックすることも可能です。

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

注記 : フラグとして機能し、本来値のないコマンド ライン オプションは、0 または 1 などの値を渡すことにより簡単にインプリメントできます。上記のプロシージャ例では、フラグ `-v` は「`myproc2 -v 1`」を使用してオンにできます。

ローカル変数とグローバル変数

ローカル変数は、プロシージャ内で作成される変数です。関数のスタック内でランタイムに作成されます。ローカル変数はプロシージャ内でのみアクセス可能で、変数名がプロシージャ外の変数名と競合することはありません。たとえば、プロシージャ内で作成されたローカル変数 `foo` とプロシージャ外で作成された `foo` とは別のもので、これらの変数にはそれぞれ独立したコンテキストがあります。ローカル変数は、ほかの変数と同様に `set` Tcl キーワードで作成します。

プロシージャの引数として定義されたパラメーターは、デフォルトでローカル変数となります。プロシージャが呼び出されると (例 : `reportCriticalPaths $myfilename`)、呼び出し変数 (例 : `$myfilename`) はプロシージャのスタック内にコピーされます。この場合、呼び出し変数が多数の要素を含む Tcl リストであると、ランタイムおよびメモリ使用量が増加します。呼び出し変数の内容を変更することが必要な場合もあります。Tcl では、変数の内容を渡す代わりに、変数名を参照として渡す方法があります。変数が参照として渡されると、プロシージャ内での変数の変更により、呼び出し空間内の呼び出し変数も直接変更されます。参照として渡すパラメーターを定義するには、プロシージャの本文でキーワード `upvar` を使用します。前述の `lshift` ではこの手法を使用しています。

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

サンプル プロシージャ `myproc` では、`$args` の内容ではなく、変数名を渡すことにより `lshift` を呼び出しています。

グローバル変数は、プロシージャ外で作成された変数で、グローバル名前空間に属しています。プロシージャ内でグローバル変数を参照するには、global キーワードを使用し、その後に変数名を記述します。

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

上記の例では、システム環境変数を表示する printEnv というプロシージャを定義しています。Tcl 配列 env は、Vivado ツールの起動時に初期化されるグローバル変数です。printEnv プロシージャでは、global env コマンドを使用して env 変数を参照しています。グローバル変数を定義したら、ローカル変数と同様にアクセスできます。グローバル変数は、読み出しおよび変更できます。

グローバル変数は、名前空間の修飾子を指定してアクセスすることもできます。グローバル名前空間の修飾子は「::」なので、プロシージャでグローバル変数 env を参照するには「::env」を使用します。構文は、どのグローバル変数でも同じです。

次に例を示します。

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

printEnv では env 変数への完全パスを指定しているので、global env を宣言する必要はありません。

注記 : グローバル変数はプロシージャの範囲外で作成された変数名に依存するので、グローバル変数の使用はお勧めしません。プロシージャに大きな Tcl リストを渡す必要性を回避するため、グローバル変数が使用されることがあります。グローバル変数を使用する前に、upvar 手法を考慮してみてください。

プロシージャの名前空間

デフォルトでは、Tcl インタープリター内で作成されたプロシージャはグローバル名前空間内に作成されます。この欠点は、異なるソースからの複数の Tcl スクリプトが使用される場合に、プロシージャ名または変数名の競合が発生する可能性があることです。また、一部のプロシージャのみで使用される、ユーザーが直接アクセスすることを意図しないプロシージャが、グローバル名前空間に含まれることになる可能性もあります。

Tcl では、すべての変数およびプロシージャをグローバル名前空間内で定義する代わりに、範囲を制限した変数およびプロシージャを含む名前空間がサポートされています。名前空間はネストさせることができ、範囲のレベル数の制限なしで、名前空間内に別の名前空間を定義できます。名前空間は、プロシージャ名および変数名に新しい構文を追加します。2 つのコロン (::) で、変数名またはプロシージャ名と名前空間名を区切ります。

次の例に、名前空間の作成方法とその名前空間にプロシージャおよび変数を割り当てる方法を示します。この例では、2 つのパブリック プロシージャ (push および pop) を含む小さなスタックの機能を再生する名前空間 foo を作成しています。

```
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::*
```

説明 :

1. 次のコマンドを使用して名前空間を定義します。

   `namespace eval <name> { … }`

2. 行 1 は名前空間 `foo` を定義し、行 29 は名前空間定義の閉じかっこです。
3. 名前空間内の変数は、variable コマンドで作成します (行 2 ～ 4)。

   `variable <varname> ?<varvalue>?`

   Tcl 配列を variable コマンドで初期化することはできません。まず作成し (行 4)、その後で初期化する必要があります (行 5)。

   注記 : 名前空間内で変数を定義するのに `set` コマンドを使用しないでください。そのようにすると、グローバル名前空間に同じ変数名がある場合に Tcl インタープリターで混同される可能性があります。

4. プロシージャは、名前空間定義の中または外に作成できます。コマンド `namespace eval … { … }` 内でプロシージャを作成した場合、名前に名前空間の修飾子 (この例では `foo::`) は必要ありません。

   行 9 および 16 : 名前空間定義内に `push` および `pop` を作成します。

5. プロシージャを名前空間定義外で作成し、プロシージャ名に完全な名前空間修飾子を付けて名前空間に追加することもできます。上記の例では、プロシージャ dump を名前空間定義外で作成し、名前空間 foo に追加しています (行 31)。
6. 行 10 ～ 11、17 ～ 18、32 ～ 33 : プロシージャで、キーワード `variable` を使用して名前空間内で作成された変数を参照します。

7. 名前空間内で作成されたプロシージャは、完全な名前空間修飾子 (foo::push、foo::pop、foo::dump など) を使用してアクセスできます。同じ名前空間内のプロシージャを参照する場合は、名前空間修飾子は必要ありません。たとえば、dump で push を呼び出す必要がある場合、foo::push ではなく push を指定します。
8. 行 7 : 名前空間では、パブリックおよびプライベート プロシージャという概念がサポートされます。名前空間内のすべてのプロシージャは完全な名前空間修飾子を使用してアクセスできますが、コマンド namespace export… を使用すると、どのプロシージャを名前空間外にエクスポートするかを指定できます。プロシージャ名をエクスポートすると、コマンド namespace import… (行 45) を使用してグローバル名前空間にインポートできます。このようにすると、プロシージャを完全な名前空間修飾子を指定せずに直接呼び出すことができるようになります。

   次に、名前空間 foo の使用例を示します。

```
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"
```

テンプレート スクリプト

次に、先ほど説明した概念に基づくテンプレート スクリプトを示します。このスクリプトでは、次を示します。

1. プライベート名前空間を使用することにより、グローバル名前空間の汚染を回避 (`lshift` は名前空間 `foo` 内のみで使用可能)
2. コマンド ライン引数の処理 (`-help` および `-version` でスクリプトのバージョンを示すなど)
3. `return -error` (または `error`) コマンドを使用して必要に応じて Tcl エラーを生成

```
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: xxxxxxxxxxxxxxxxxxx.
                   xxxxxxxxxxxxxxxxxxx.

      Example:
         %s -port xxxxxxxxxxxxxxx

    } $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 {}
    }
```

デザイン オブジェクトへのアクセス

Vivado Design Suite では、プロジェクト、デザイン、デバイス情報がインメモリ データベースに読み込まれ、合成、インプリメンテーション、タイミング解析、およびビットストリームの生成に使用されます。このデータベースは、プロジェクト モードでも非プロジェクト モードでも同じです。FPGA デザイン フローを実行していくと、それに応じてデータベースがアップデートされます。デザイン フローのどの段階でも、データベースの内容をチェックポイント ファイル (.dcp) に保存できます。Vivado ツールで Tcl コマンドを使用すると、デザイン データベースにアクセスし、Tcl オブジェクトをクエリしたり、プロパティを読み出しまたは設定したりして、その結果を Tcl スクリプトでさまざまな目的で使用できます。データベースの内容を理解し、それに対してスクリプトをいかに効率的に記述できるかを理解しておくと有益です。

Vivado Design Suite の Tcl インタープリターでは、プロジェクト、デバイス、ネット、セル、ピンなど、多数のファースト クラス オブジェクトにアクセスできます。Vivado Design Suite では、プロジェクト モードでも非プロジェクト モードでも、デザインの進行に応じてこれらのデザイン オブジェクトが随時アップデートされ、インメモリ データベースに読み込まれます。

対話的にデザイン オブジェクトのクエリ、プロジェクトの状態の解析、インメモリ デザインにアクセスするスクリプトの記述、カスタム レポートの生成、オプションのデザイン フロー手順などを実行できます。各オブジェクトには複数のプロパティがあり、いつでも読み出すことができ、また一部のプロパティは設定もできます。ほとんどのデザイン オブジェクトはほかのデザイン オブジェクトに関連付けられており、その関連性をたどって関連オブジェクトやその情報を取得できます。

デザイン オブジェクトのクエリには、`get_*` Tcl コマンドを使用します。結果取得されたデザイン オブジェクトは、直接処理するか、Tcl 変数に代入できます。`get_*` コマンドをすべてリストするには、`help get_*` を使用します。オブジェクトを変数に代入すると、デザイン データベースに対するクエリの回数を削減でき、実行時間を短縮できます。ネットやピンのリストのクエリは時間のかかるプロセスであり、結果を保存しておくことで、同じ情報に繰り返しアクセスする必要がある場合にデザイン フローを高速化できます。詳細は、61 ページの「オブジェクトのキャッシュ」を参照してください。

デザイン オブジェクトの各クラス (ネット、ピン、ポートなど) には標準のプロパティがあり、読み出したり、一部のプロパティは値を変更できます。また、RTL ソース ファイルで指定されているデザイン属性、Verilog パラメーター、VHDL ジェネリックも、それらが設定されているネットリスト オブジェクトのプロパティとして保存されます。たとえば、ポート オブジェクトには方向を指定するプロパティがあり、ネット オブジェクトにはファンアウトを指定するプロパティがあります。Vivado ツールでは、これらのプロパティを追加、変更、およびレポートする多数のコマンドがあります。`get_* -filter` オプションを使用すると、デザイン オブジェクトのリストにフィルターを適用し、特定のプロパティ値のオブジェクトのみを取得できます。詳細は、39 ページの「フィルター結果」を参照してください。オブジェクトに設定されているプロパティのリストを取得するには、`list_property` コマンドを使用します。プロパティのタイプが `enum` である場合、`list_property_value` コマンドを使用して有効な値のリストを取得できます。

すべてのオブジェクトには、NAME および CLASS プロパティがあります。オブジェクトを Tcl 変数に代入すると、そのオブジェクトへのポインターが変数に保存されます。オブジェクトを変数により Tcl コマンドや Tcl プロシージャに渡すことができます。オブジェクトが引数として文字列を必要とする Tcl プロシージャまたはコマンドに渡された場合、オブジェクトそのものではなくオブジェクトの NAME プロパティが渡されます。次の例に、セル オブジェクトを変数 `$inst` に代入し、その変数に対して `puts` コマンドおよび `report_property` コマンドを実行した結果を示します。`puts` コマンドでは文字列が処理されるのでオブジェクトの名前が表示され、`report_property` コマンドではオブジェクトのプロパティとその値が返されていることに注目してください。

```
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
```

どのクラスのデザイン オブジェクトに対しても、カスタム プロパティを作成できます。これは、メモリ内のデザイン オブジェクトにスクリプトからの情報を追記する場合に有益です。次の例では、セル オブジェクトに対して SELECTED というプロパティを作成しています。プロパティ値は整数として定義されます。

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

オブジェクトのクラスにプロパティを作成すると、特定のオブジェクト上で `set_property` および `get_property` コマンドを使用して管理し、`list_property` および `report_property` コマンドを使用してレポートできます。次の例では、名前が `*aurora_64b66b*` というパターンに一致するすべてのセルの SELECTED プロパティを 1 に設定しています。

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

名前を使用したオブジェクトの取得

ほとんどのデザインは、階層的に接続されたブロックまたはモジュールで構成されています。ボトムアップ、トップダウン、またはミドルアウトで構築されたデザインのいずれでも、デザイン階層で特定のオブジェクトを検索するのは一般的なタスクです。

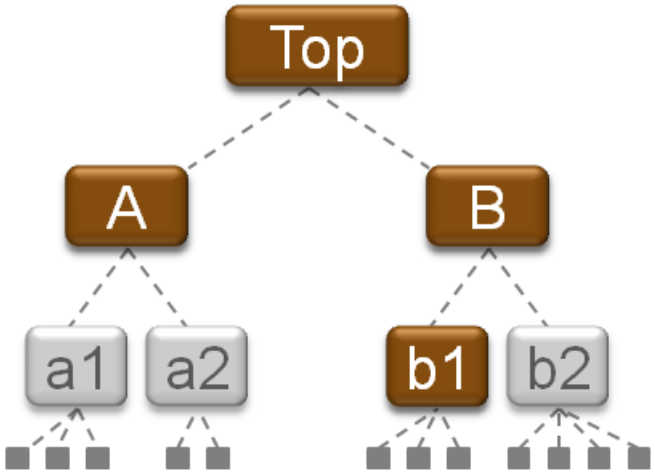

図 2 : デザイン階層の検索

get_* コマンドでは、デフォルトではデザイン階層の最上位のオブジェクトのみが返されます。get_* コマンドを使用する前に current_instance コマンドを使用すると、デザインの特定の階層インスタンスでデザイン オブジェクトを検索できます。検索範囲をデザインの最上位に戻すには、current_instance コマンドを引数を指定せずに実行します。

図 2 に、最上位にモジュール A および B がインスタンシエートされている例を示します。モジュール A には a1 および a2 階層インスタンスが含まれ、モジュール B には b1 および b2 階層インスタンスが含まれます。a1、a2、b1、および b2 には、それぞれ最下位セル (Unisim インスタンス) が含まれます。

```
# 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.
```

get_* コマンドでは最上位または current_instance で指定した現在のインスタンスのレベルでのみ検索が実行されますが、現在のインスタンスに対する階層インスタンス名を含む検索パターンを指定できます。デフォルトでは、現在のインスタンスはデザインの最上位に設定されています。最上位からインスタンス b1 を参照するには、次のように指定します。

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

-hierarchical オプションの使用

get_* コマンドでは、デフォルトでは現在のインスタンスのレベルでのみオブジェクトが検索されますが、-hierarchical オプションを使用すると、現在のインスタンスのレベルから各デザイン階層を検索できます。

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

-hierarchical オプションでは、オブジェクトの完全な階層名に対してではなく、デザイン階層の各レベルで指定された名前のパターンが検索されます。通常、-hierarchical を使用する場合、指定する検索パターンに階層区切り文字を含めないでください。そうでないと、オブジェクトは返されません。ただし、合成中にネットリストが部分的にフラット化されており、フラット化されたネットリスト レベルを示すのにも階層区切り文字が使用されている場合は例外ですこの場合、階層区切り文字は名前の階層レベルを示しており、メモリ内に読み込まれているデザインの階層レベルを示しているわけではないので、階層区切り文字を検索パターンに使用できます。

次の例は 33 ページの図 2 に基づいており、階層ネットリストのみを示します。

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

-hierarchical を使用した検索は、current_instance コマンドを使用して階層インスタンスを指定し、各階層レベルで指定の名前のパターンを手動で検索するのと同じです。次の例では、33 ページの図 2 でこの手動検索を実行しています。

```
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
}
```

重要：-hierarchical を -regexp と共に使用する場合、検索パターンは完全な階層名と比較され、検索パターンとして「B/*」を指定した場合にこのパターンに一致するセル名が返されます。たとえば、図 2 で get_cells -hierarchical -regexp B/.* を実行すると、ブロック B の下のすべてのセルが返されます。-regexp の詳細は、『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) [参照 1] を参照してください。

-filter および -regexp オプションの使用

Vivado Design Suite では、`get_*` コマンドを使用してオブジェクトのサブセットを選択するのに、`-filter` および `-regexp` オプションを `-hierarchical` オプションと共に使用できます。

次の表に、コマンドで指定したパターンに対する `-hierarchical`/`-filter`/`-regexp` オプションの効果を示します。

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

表 1 : -hierarchical/-filter/-regexp オプションの効果

-hierarchical	-filter	-regexp	結果
			pattern は現在の階層レベル (current_instance) にあるオブジェクトのローカル名と比較されます。
使用			*pattern* は現在の階層レベル (current_instance) およびその下位にあるオブジェクトのローカル名と比較されます。
	使用		*pattern* は現在の階層レベル (current_instance) にあるオブジェクトに適用されるフィルター式となります。NAME プロパティがオブジェクトの完全な階層名と比較されます。
使用	使用		*pattern* は現在の階層レベル (current_instance) およびその下位にあるオブジェクトに適用されるフィルター式となります。NAME プロパティがオブジェクトの完全な階層名と比較されます。
		使用	*pattern* は正規表現として現在の階層レベル (current_instance) にあるオブジェクトのローカル名と比較されます。
使用		使用	*pattern* は正規表現として現在の階層レベル (current_instance) およびその下位にあるオブジェクトのローカル名と比較されます。
	使用	使用	*pattern* は現在の階層レベル (current_instance) にあるオブジェクトに適用されるフィルター式となります。NAME プロパティがオブジェクトの完全な階層名と比較されます。フィルター式は、正規表現として適用されます。
使用	使用	使用	*pattern* は現在の階層レベル (current_instance) およびその下位にあるオブジェクトに適用されるフィルター式となります。NAME プロパティがオブジェクトの完全な階層名と比較されます。フィルター式は、正規表現として適用されます。

注記 : オブジェクトのローカル名は、現在の階層レベル (current_instance) からの名前の一部です。親階層から継承された部分の名前は含まれません。

注記 : デザインの一部がフラット化されている場合、フラット化されたレベルに含まれるオブジェクトのローカル名には階層区切り文字が含まれます。この場合、名前のこの部分の階層区切り文字は、区切り文字としてはなく、リテラル文字として扱われます。

注記：-filter オプションはフィルター式を指定し、式内のパターン比較はグローバル条件式のフォーマットに従います。フィルター式ではオブジェクトのプロパティに基づく文字列の比較が実行され、必要に応じて複雑なものにできます。フィルター式に NAME プロパティを使用すると、文字列比較にオブジェクトのローカル名ではなく完全な階層名が使用されます。ただし、現在の階層レベル (current_instance) に含まれるオブジェクトのみが検索されます。-hierarchical を -filter と共に使用すると、現在の階層レベルおよびその下位にあるオブジェクトに対してフィルター処理が実行されます。-regexp を -filter と共に使用すると、フィルター式内のパターン比較は正規表現に従います。

注記：-regexp は、コマンドに指定された検索パターンが正規表現であることを示します。*、.、[ ]、+ などの一部の文字は正規表現では特別な意味を持つので、注意が必要です。正規表現でこれらの文字をリテラルとして使用する場合は、エスケープ処理する必要があります。

文字列比較では大文字と小文字が区別され、常に検索文字列の冒頭および末尾にアンカーされています。検索文字列のサブ文字列を比較する場合は、正規表現を使用するかどうかによって、次の構文を使用します。

- 正規表現を使用する場合 (-regexp のみ)：
  `.*<substring>.*`
- パターンがグローバル条件式のフォーマットに従う場合 (その他のオプション)：
  `*<substring>*`

次に、cpu_hdl プロジェクトに基づく例を示します。このプロジェクトは、Vivado IDE の Getting Started ページで [Open Example Project] リンクをクリックして開くことができます。

- 現在のインスタンスを fftEngine/fftInst/ingressLoop[7].ingressFifo に変更：

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

- 現在のインスタンスの下にあるすべてのセルを取得 (階層セルは 1 つのみ)：

```
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 … (ほかに 154 個のセル)
```

- 現在のインスタンスおよびその下位にあるセルのローカル名には、ingressLoop は含まれません。
  文字列 ingressLoop は親セルから継承されたものであり、完全な階層名の一部です。

```
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*'.
```

- -filter オプションを使用すると、NAME プロパティは完全な階層名と一致します。

```
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 … (ほかに 154 個のセル)
```

- パターン *reg[*]* に一致するセルを検索 :

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

最後のクエリ get_cells -hier -regexp {.*reg[.*].*} で一致するセルがないのは、角かっこ ([]) がエスケープ処理されていないため、セル名のリテラル文字としてではなく、正規表現の特殊文字として扱われているからです。

フィルター式で値の範囲を指定する必要がある場合は、-filter に加えて -regexp オプションを使用する必要があります。たとえば、次のコードでは *reg[0]* から *reg[16]* までのセルのみが取得されます。正規表現では、.*reg\[[0-9]\].* および .*reg\[1[0-6]\].* が使用されます。

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

次の例では、どちらのコマンドも、CLB*X*Y* に一致するタイルで、CLB*X1Y* から CLB*X16Y* (X が 1 ～ 16) を除くものが返されます。

```
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.*"}
```

ピンの検索

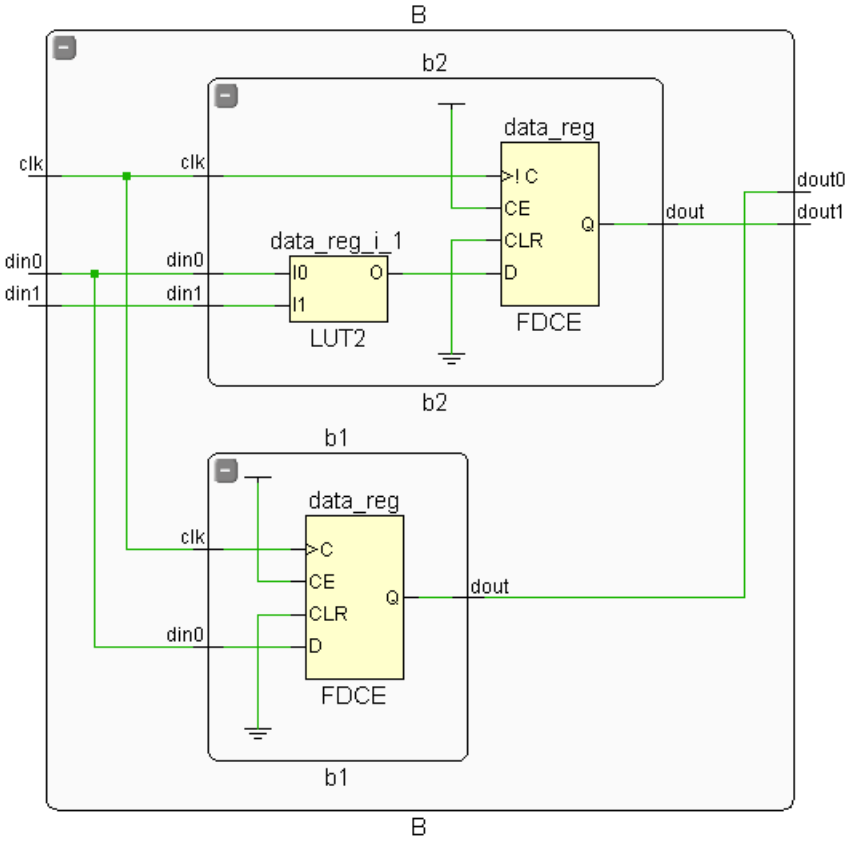

図 3 : ピン名の検索

ピンの名前は、そのピンが属するインスタンスに基づいています。ピンを検索する場合、階層区切り文字を使用し、インスタンス名とピン名を区切る必要があります。次の例は、図 3 に示されています。

```
# 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
```

ピンを検索する際、-hierarchical も使用できます。

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

1. -hierarchical と -hier は同じです。オプションを識別するのに十分な文字数が記述されていれば、Vivado Design Suite Tcl シェルで自動的にオプション名が特定されます。そのため、-of_object と -of も同じオプションとみなされます。

フィルター結果

get_* を使用してデザイン オブジェクトを検索する場合、通常必要なのは一部のオブジェクトのみです。デザインのすべてのネットリスト オブジェクトは必要なく、たとえば特定のタイプのセルや特定の名前のネットのみなどが必要です。要素の一部のみが返されるようにする必要があることもあります。

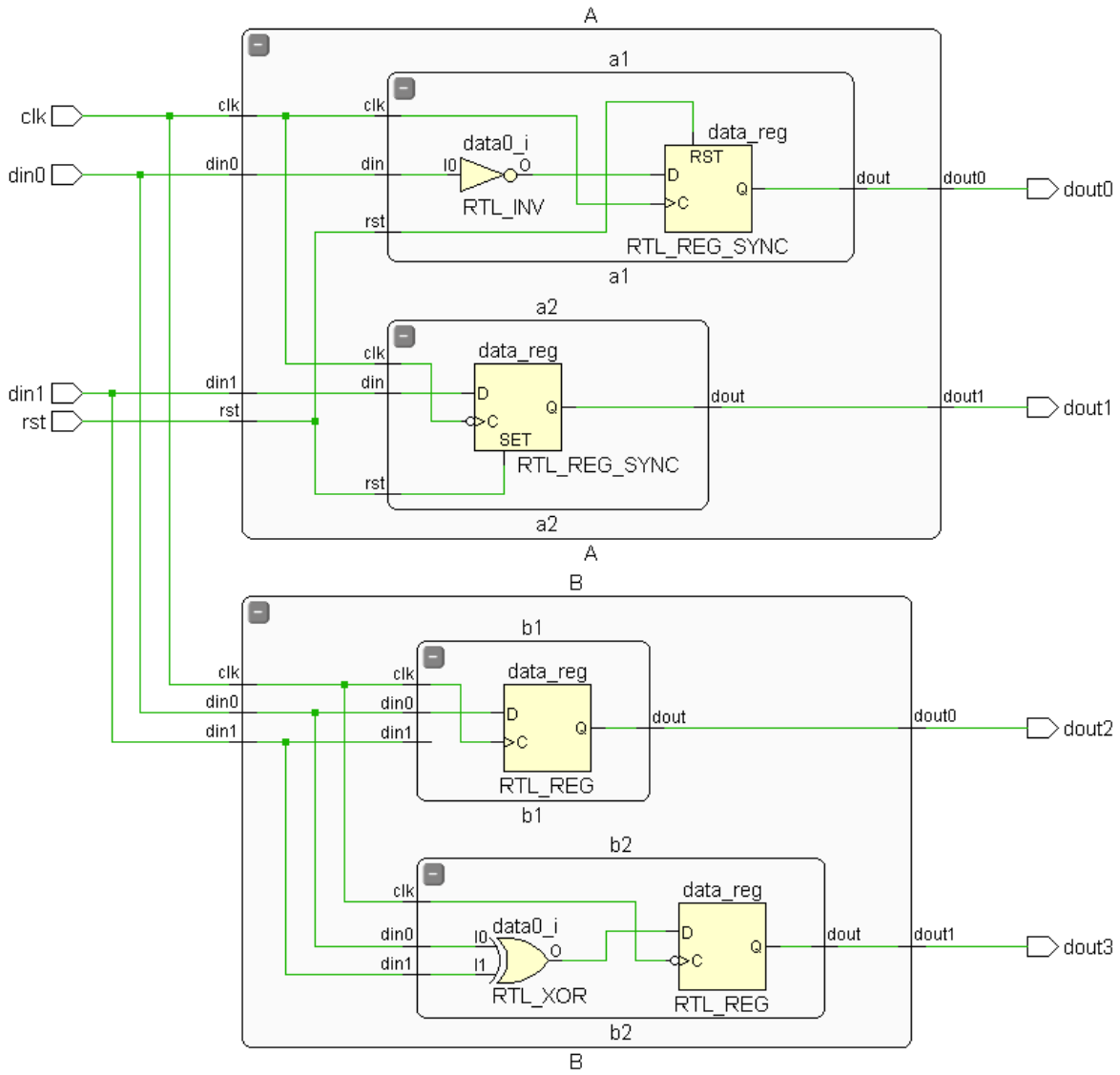

図 4 : 階層デザインの検索

ワイルドカード * および ? を使用したり、-regexp を使用したりして検索パターンを指定し、返される検索結果を制限できます。検索する階層範囲を指定するには、current_instance コマンドまたは -hierarchy オプションを使用します。

次の例では、図 4 に示すデザインに対する異なる結果を示します。

```
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
```

-filter オプションを使用すると、get_* コマンドの結果を特定のプロパティに基づいてフィルターできます。たとえば次のコマンドでは、階層名が「B/b*」で開始するすべてのセルのうち、ユーザーにより配置されていないもの (IS_LOC_FIXED が FALSE または 0) のものが返されます。

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

重要 : NAME プロパティには、オブジェクトの完全な階層名が含まれます。NAME プロパティをフィルターする場合、-hierarchical も含めたコマンドのほかのオプションにかかわらず、検索パターンは完全な NAME 文字列に対して評価されます。

-filter オプションにより、結果がフィルターされてから返されます。ただし、フィルターを適用する前の検索結果を変数に代入している場合は、それがメモリに保存されます。filter コマンドを使用すると、変数として保存されているリストも含め、オブジェクトの任意のリストの内容をフィルターできます。先ほどの例の場合、$unLoced に保存されているリストを次のようにフィルターできます。

```
set unLocedLeaf [filter $unLoced {IS_PRIMITIVE}]
```

この例では、$unLoced に保存されている結果をフィルターし、デザインのプリミティブ インスタンスのみを返しています。

重要 : filter コマンドでは元の Tcl 変数は変更されないので、結果を別の Tcl 変数に保存する必要があります。

ヒント : 上記の例で、ブール型プロパティ !IS_LOC_FIXED および IS_PRIMITIVE が直接使用されていることに注目してください。ブール型 (bool) プロパティでは、フィルター式が True か False かを直接評価できます。

フィルター パターンに使用できる演算子は等価 (==)、不等価 (!=)、含める (=~)、含めない (!~) です。数値比較演算子 <、>、<=、および >= も使用できます。複数のフィルター式を AND (&&) および OR (||) で組み合わせることもできます。

関連性を使用したオブジェクトの検索

デザインのほかのオブジェクトに関連するオブジェクトを検索する必要がある場合があります。たとえば、特定のセルのピンに接続されているすべてのネットや、特定のネットに接続されているすべてのセルを選択する場合などです。Vivado Design Suite では、デザインのエレメントをそれらの関連性を利用して検索できます。これには、`get_*` コマンドで `-of_objects` オプションを使用します。図 5 に、インメモリ デザインのオブジェクト間の関連性を示します。

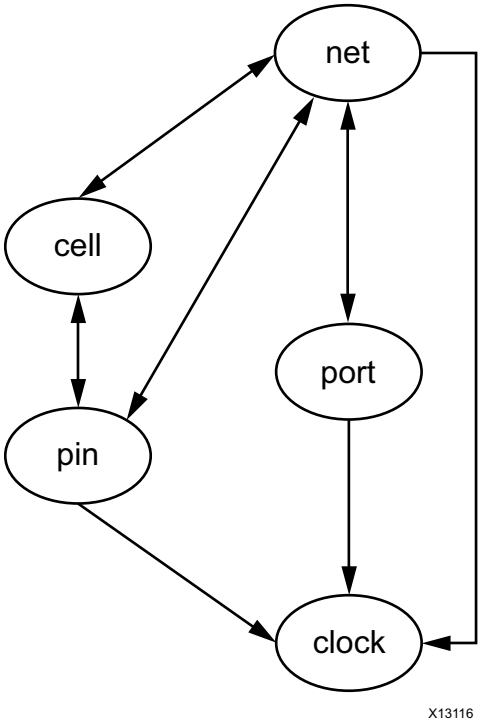

図 5 : Vivado Design Suite でのオブジェクト間の関連性

注記 : これは概念的に図示したものであり、すべてのオブジェクトとその関連性をすべて表すものではありません。

`-of_objects` オプションをサポートする `get_*` コマンドのヘルプに、関連性のあるオブジェクトがリストされます。

```
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}
```

`-of_objects` オプションを使用すると、ネット オブジェクトのリストに接続されているピン オブジェクトのリストを簡単に取得できます。

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

これらのネットのドライバーのリストのみを取得する場合は、`-filter` オプションを使用します。

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

また、セルのリストからピンのリストを取得したり、ネットのリストからセルのリストを取得したりできます。

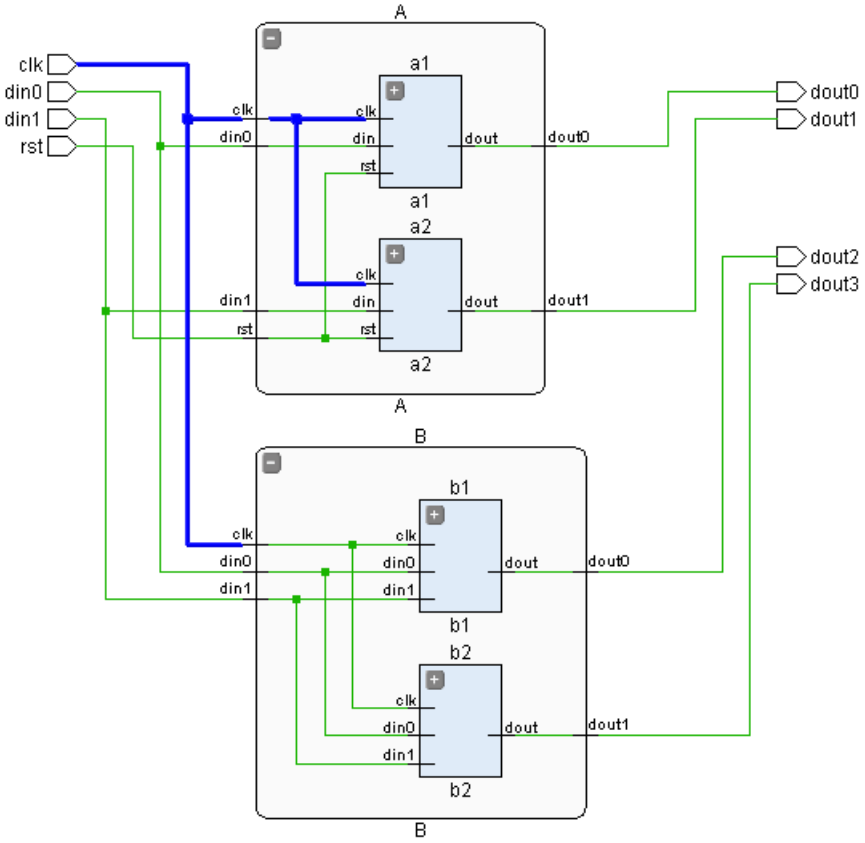

図 6 : 関連性を使用したオブジェクトの検索

次の例では、図 6 に示すように、インスタンス a1 からクロック ピンを取得し、そのクロック ピンに接続されているネットを取得して、そのネットに接続されているピンを取得して、さらにそのピンに接続されているネットを取得して、最後にそれらのネットに接続されているピンを取得しています。

```
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
```

最後の get_pins コマンドにより、それまでに返されたピンに加え、階層モジュール B のクロック ピン B/clk が返されます。階層をまたいでクロック ネット オブジェクトのプリミティブ ピンを取得するには、get_pins コマンドの -leaf オプションを使用できます。次の例では、-leaf を使用した場合の結果を示します。

```
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 コマンド

get_nets コマンドを使用すると、ネットがデザイン階層を通過するため、同じネットが異なる表現で返されることがあります。次に、get_nets コマンドのオプションを使用して、ネットの適切な表現が選択されるようにする例をいくつか示します。

次の例では、次のように定義された LUT2 の最下位ピンを使用して、最下位ピンに接続されているネットの異なるセグメントをクエリします。

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

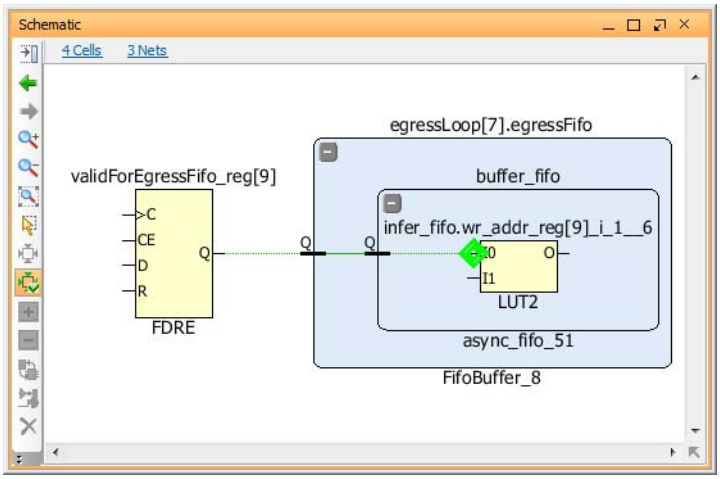

図 7 : set mypin の例

単純に最下位ピン mypin に接続されているネットを取得すると、階層内の最下位ピンに直接接続されているネットセグメントが返されます。

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

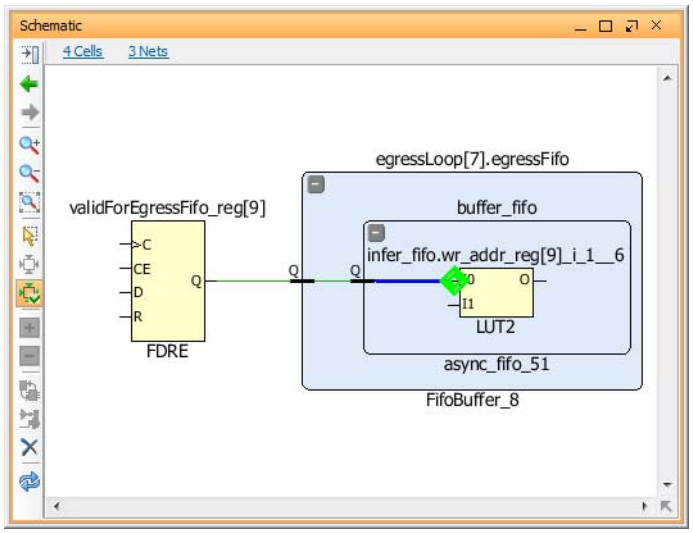

図 8 : get_nets -of の例

最下位ピン mypin に接続されているネットのすべてのセグメントを選択するには、-segments オプションを使用します。

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

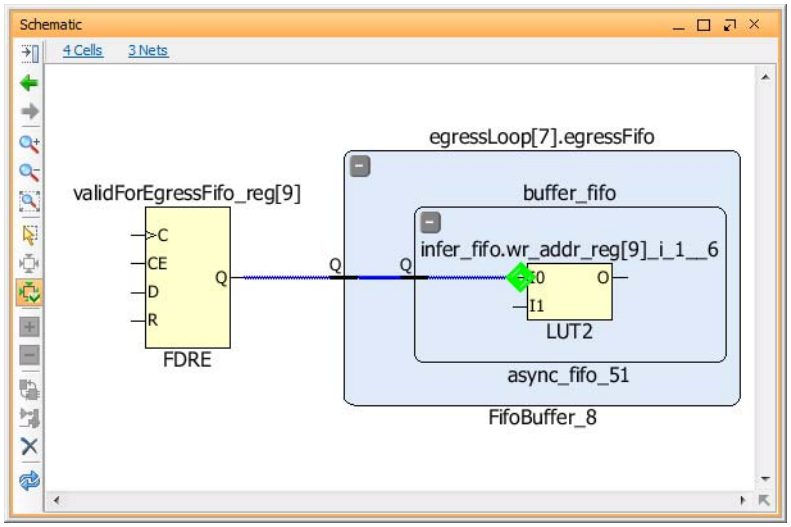

図 9 : get_nets -segments の例

最下位ピン mypin に接続されているネット セグメントで最上位階層のもののみを選択するには、-segments オプションと共に -top_net_of_hierarchical_group を使用します。これは、ネット名を最小文字数で表示するので、デバッグ文を表示する際に有益です。

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

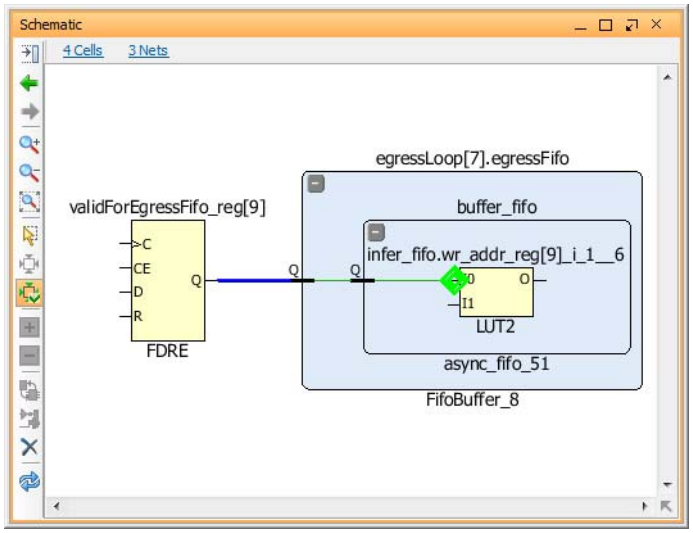

図 10 : get_nets -top_net_of_hierarchical_group の例

オブジェクトのリストの処理

get_* コマンドを使用すると、返されたオブジェクトは標準の Tcl リストと同様で、同じように機能しますが、Vivado Design Suite ではオブジェクトの 1 つのクラスのコンテナー (セル、ネット、ピン、ポートなど) が返され、標準の Tcl リストとは異なります。ただし、このコンテナーは通常、標準の Tcl リストと同様で、同じよう機能します。オブジェクトのコンテナーは Vivado Design Suite で自動的に処理されるので、ユーザーが意識する必要はありません。たとえば、標準 Tcl コマンド llength をオブジェクトのコンテナーに対して使用し (get_cells の結果など)、通常の Tcl リストでの場合と同様に、コンテナーに含まれるエレメントの数を取得できます。

Vivado Design Suite のリストを処理するビルトインの Tcl コマンドは、オブジェクトおよびオブジェクトのコンテナーを完全にサポートするため拡張されています。たとえば、lsort、lappend、lindex、および llength は、オブジェクトの NAME プロパティに基づいてコンテナーを制御するよう拡張されています。これらのコマンドは、オブジェクトのコンテナーが渡された場合、オブジェクトのコンテナーを返します。たとえば、lsort は、get_cells で取得したセルのコンテナーを、オブジェクトの階層名に基づいて並べ替えます。

lappend を使用するなどしてコンテナーにオブジェクトを追加できますが、現在コンテナーに含まれるオブジェクトと同じタイプのオブジェクトしか追加できません。異なるタイプのオブジェクトや文字列を追加しようとすると、Tcl エラーが返されます。

次の例では、Vivado のコンテナーを降順に並べ替え、puts コマンドと foreach ループを使用して、オブジェクトを 1 行に 1 つずつ表示しています。

```
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
...
```

注記 : lappend コマンドは Tcl コンソールでサポートされますが、read_xdc コマンドで読み込む XDC 制約ファイルではサポートされません。ただし、lappend コマンドで取得したリストを read_xdc コマンドと互換性を持たせるように変換できます。次のようなコードがあるとします。

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

Tcl 変数 myClocks を、read_xdc コマンドと互換性を持たせるようにするには、次のようにします。

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

出力先の指定

Vivado Design Suite の多くの Tcl コマンドでは、コマンドから返される情報を、-file オプションを使用して印刷やツール外での処理用にファイルに保存したり、-return_string オプションを使用して Vivado ツールでの処理用に文字列として変数に保存できます。

すべてのレポート コマンドで、-file オプションがサポートされています。大量の情報を出力するレポート コマンドでは、その後の解析、デザイン プロジェクトの文書サポート、またはほかの部署にダウンストリーム処理用に渡す場合などに、ファイルに出力すると有益です。次に、ファイル出力をサポートするコマンドの一部を示します。

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

たとえば、report_timing コマンドの結果をファイルに記述するには、次のコマンドを使用します。

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

ファイル名の一部として、相対パスまたは絶対パスを指定できます。相対パスは、Vivado ツールを起動したディレクトリまたは pwd コマンドで返される現在の作業ディレクトリを基準として指定します。

ヒント：パスをファイル名の一部として指定しない場合は、現在の作業ディレクトリまたは Vivado ツールを起動したディレクトリにファイルが作成されます。

既存のファイルにコマンドの結果を追加するには、-file オプションと共に -append オプションを使用します。次の例では、1 つのファイル all_violations.rpt を作成し、2 つのコマンドの結果を保存しています。

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

ファイルが作成されたら、ファイル システムからファイルを開いて確認したり書き込んだりできます。Vivado Design Suite Tcl コンソールには、ファイルにアクセスするさまざまなコマンドがあります。詳細は、48 ページの「ファイルへのアクセス」を参照してください。

すべての report_* コマンドでは、-return_string オプションもサポートされています。このオプションを使用すると、コマンドの出力が Tcl 変数に代入可能な文字列として返されます。コマンド出力を変数に代入すると、Tcl スクリプトでのその後の処理に有益で、主要な情報を抽出してフロー制御や分岐を可能にしたり、スクリプトで使用するほかの変数を設定したりできます。

次に、-return_string オプションをサポートするコマンドの一部を示します。

```
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
```

レポート コマンドから返された文字列を改行文字 \n で分離して、文字列をリストとして行ごとに処理できます。

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

Vivado Design Suite Tcl コンソールには、文字列を操作するさまざまなコマンドがあります。詳細は、50 ページの「文字列の操作」を参照してください。

ファイルへのアクセス

ファイル システムにファイルを記述すると、ファイルを処理するさまざまな Tcl コマンドを使用できます。ファイル パス、ファイル名、ファイル拡張子など、ファイルの要素を抽出できます。ファイルに関する情報を調べる次のようなコマンドもあります。

- `file exists filename` : *filename* が存在し、その場所の読み取り権限がある場合は 1、それ以外の場合は 0 を返します。ファイルが既に存在しているかどうかを調べるのに使用します。
- `file type filename` : ファイルのタイプを示す文字列を返します。可能な値は `file`、`directory`、`characterSpecial`、`blockSpecial`、`fifo`、`link`、`socket` です。
- `file dirname filename` : *filename* の最後のスラッシュまでのディレクトリ構造を最後のスラッシュを含めずに返します。
- `file rootname filename` : *filename* の最後のピリオドまでの文字を最後のピリオドを含めずに返します。
- `file tail filename` : *filename* の最後のスラッシュより後の文字すべてを返します。
- `file extension filename` : *filename* の最後のピリオド以降の文字を最後のピリオドを含めて返します。

次に、Tcl コマンドのいくつかの例を示します。

```
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
```

`report_*` コマンドまたは `write_*` コマンドでファイルを作成したら、Tcl スクリプトでファイルを開いて、その内容を読み出したり、追加の内容を記述したりできます。ファイルを開いたり閉じたり、ファイルの読み出しまたは書き込みを実行するには、次の Tcl コマンドを使用できます。

- `open <filename> [access] [perms]` : filename を開き、ファイルにアクセスするのに使用したファイル ハンドル (*fileID*) を返します。必要に応じてファイル ハンドルを参照できるようにするため、*fileID* を Tcl 変数に代入するのが一般的です。新しいファイルの権限は、*perms* とプロセス umask の組み合わせで設定します。*access* モードは、開いたファイルの読み取り権および書き込み権を指定します。一般的なアクセス モードは次のとおりです。
  - r : 読み出しモード。ファイルが存在している必要があり、作成されません。アクセス モードを指定しない場合、これがデフォルトです。
  - w : 書き込みモード。ファイルが存在しない場合は、作成されます。データはファイルの冒頭から記述され、既存のファイルの内容は切り捨てられるか、上書きされます。
  - a : 追加モード。ファイルが存在しない場合は、作成されます。データはファイルの末尾に記述され、既存のファイルの内容に追加されます。
- `read [-nonewline] fileId` : *fileId* から残りすべてのバイトを読み出し、オプションで最後の文字が改行 \n の場合はその最後の文字を破棄します。ファイルを開いた直後にこの形式を使用すると、`read` コマンドでファイル全体が一度に読み出されます。
- `read fileId numBytes` : *fileId* から指定したバイト数 *numBytes* を読み出します。この形式は、ファイルのブロックをファイルの最後まで読み出す場合に使用します。
- `eof fileId` : *fileId* で EOF (End Of File) が発生した場合は 1、それ以外の場合は 0 を返します。

- gets *fileId* [*varName*] : *fileId* から次の行を読み出します。改行文字は破棄されます。$varName を指定した場合は行の文字列をその変数に代入し、それ以外の場合は文字列をコマンド シェルに返します。次に、gets コマンドの異なる形式を示します。

```
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.
```

  上記の例では、$fileHandle がファイルを開いたときに返されるファイル ハンドルです。最初の例は gets の単純な形式で、出力を代入する Tcl 変数は指定していません。この場合、出力は stdout に返されます。2 番目の例では出力を $line という変数に代入しており、gets コマンドで読み出された文字数 28 が返されます。

  注記 : gets コマンドの戻り値を Tcl 変数に代入できますが、コマンドの形式によって、ファイルの内容または gets コマンドで読み出された文字数が代入されます。

- puts [-nonewline] [*fileId*] *string* : 指定した *fileId* に文字列を書き込みます。オプションで、改行文字 \n を省くこともできます。puts コマンドのデフォルトの *fileId* は stdout です。
- close *fileId* : 開いているファイル チャネル *fileId* を閉じます。Tcl スクリプトで開いたファイルを閉じるようにすることが重要です。そうしないと、Vivado アプリケーションでメモリ リークやその他の問題が発生する可能性があります。

次の例では、ファイルを読み出しアクセス モードで開き、ファイル ハンドルを $FH に代入して、1 つの操作でファイルの内容を読み出して $content に代入し、その内容を Tcl リストに分割しています。完了したら、ファイルを閉じます。

```
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
```

注記 : パフォーマンスおよびメモリの面から、サイズの大きいファイルを 1 回の操作で読み出すことはお勧めしません。

次の例では、ファイル全体を一度に読み出してから結果を解析するのではなく、ファイルを 1 行ずつ最後まで読み出し、stdout に行数と行の内容を出力しています。完了したら、ファイルを閉じます。

```
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
```

次の例では、デザインのすべての I/O ポートをその方向と共に名前順に並べ替えて、ファイル ports.rpt に書き込んでいます。

```
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
```

上記の例では、ファイルを書き込みモードで開いています。読み出しモードとは異なり、書き込みモードではファイルが存在していない場合は作成され、ファイルが存在している場合は上書きされます。既存のファイルの最後に新しい内容を追加するには、ファイルを追加モードで開く必要があります。

文字列の操作

-return_string オプションを使用すると、report_* コマンドの出力を stdout ではなく Tcl 文字列として返すことができます。文字列は Tcl 変数に代入したり、解析または処理できます。

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

文字列を変数に代入すると、文字列を処理するさまざまな Tcl コマンドを使用できます。

- append *string* [*arg1 arg2* ... *argN*] : 指定した *args* を *string* の最後に追加します。
- format *formatString* [*arg1 arg2* ... *argN*] : 文字列を *formatString* テンプレートで指定したフォーマットの形式にします。テンプレートは、sprintf で使用されるように % 変換指示子を使用して指定する必要があります。追加の引数 *arg* は、フォーマットされた文字列内で置換する値を指定します。
- regexp [switches] *exp string* : 正規表現 *exp* が *string* に一致する場合は 1、それ以外の場合は 0 を返します。-nocase オプションを指定すると、大文字/小文字は区別されません。
- string match *pattern string* : glob *pattern* が *string* に一致する場合は 1、それ以外の場合は 0 を返します。
- scan *string formatString* [*varName1 varName2* ...] : 指定の *string* から値を抽出して変数 *varName* に代入し、sscanf と同じように % 変換指示子を使用して *formatString* を適用します。*varName* を指定しない場合は、値のリストが stdout に出力されます。
- string range *string first last* : *string* から、文字インデックス *first* から *last* までを、それらを含めて返します。
- string compare *string1 string2* : 2 つの文字列に対して辞書式比較を実行し、*string1* が *string2* より前の場合は -1、2 つが同じ場合は 0、*string1* が *string2* より後の場合は 1 を返します。
- string last *string1 string2* : *string2* で *string1* が最初に現れた文字インデックスを返します。*string2* で *string1* が見つからなかった場合は -1 を返します。
- string length *string* : *string* の文字数を返します。

次の例では、-return_string を使用して report_timing コマンドの結果を $report Tcl 変数に代入し、各パスの開始点、終点、パス グループ、およびパス タイプを抽出して、最後にそのパスのサマリを Tcl コンソールに出力しています。

```
# 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]
}
}
```

次は、このコードの出力例です。

```
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]
```

ループの制御

Tcl には、for、foreach、while など、コードのセクションを繰り返すビルトイン コマンドがあります。

これらのコマンドの構文は、次のとおりです。

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

上記のコマンドでは、各反復で Tcl スクリプトの body 全体が実行されますが、Tcl にはループの制御フローを変更する break および continue の 2 つのコマンドがあります。

break 文はループ コマンドを中止するために使用し、continue 文はループの次の反復にジャンプするために使用します。

注記 : プロシージャ内で実行する場合、ループは return コマンドを使用して中止できます。この場合、ループが中止されるだけでなく、制御もループを呼び出したプロシージャに戻されます。

たとえば、1 行に 1 つのインスタンス名を含む、セル名がリストされたファイルがあるとします。次のサンプル コードは、このファイルを読み込み、現在デザインに存在するセル名のみを含む Tcl リストを作成します。先ほど示した get_file_content プロシージャを再利用しています。現在のデザインで見つからないセル名が指定の数を超えた場合、ファイルの処理を停止します。

```
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 "
```

エラー処理

変数の有効性チェック

Tcl スクリプトを記述する際は、コーナー ケース (稀にしか発生しないケース) およびコードでエラーが発生する可能性のある条件を必ずチェックすることをお勧めします。適切なチェックを実行することにより、スクリプトの問題や不正な使用についてユーザーにメッセージを表示できます。チェックを正しく実行しない場合、何が問題なのか、何を直したらよいかなどの情報なしで、スクリプトが停止してしまいます。

例 1 : ファイルを読み出し/書き込みで開いたときのチェック (不適切な例)

```
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"
}
```

上記のスクリプトのアルゴリズムは正しいように見えますが、open コマンドで下位レベルの Tcl エラー (TCL_ERROR) が生成され、ファイルを開けなかった場合にスクリプトが停止します。例 3 では、このスクリプトを改善した例を示します。

例 2 : get_* コマンドを使用した後に Vivado オブジェクトが有効であるかをチェック

```
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
}
```

get_* コマンドを使用した後に Vivado オブジェクトが存在するかどうかを調べることは、これらのオブジェクトをほかのコマンド (filter、get_* など) で使用する場合に特に重要です。

Tcl エラーの処理

一部のビルトイン Tcl コマンド (およびユーザー プロシージャ) は、エラーがプログラムにより検出されない場合、Tcl エラー (TCL_ERROR) を生成してスクリプトを停止することがあります。たとえば file コマンドは、ファイルを開くことができないと、TCL_ERROR を生成します。

プログラムを問題なく実行し、TCL_ERROR 状態を検出するため、Tcl にはビルトインの catch コマンドがあります。このコマンドは、エラーが検出された場合は 1 を返し、検出されなかった場合は 0 を返します。catch コマンドは、1 つのコマンドまたはコマンドのグループに対して使用できます。

catch コマンドの基本的な構文は、次のとおりです。

```
catch script [varname]
```

ここで、script は 1 つの Tcl コマンドまたは Tcl コマンドのグループ、varname は TCL_ERROR を説明するメッセージを保存する変数の名前です。

注記 : 引数 varname は省略できます。

通常、catch コマンドは次のように使用されます。

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

例 1 は、catch コマンドを使用してファイルを開くことができない場合を考慮することにより、より適切なスクリプトとなります。

例 3 : ファイルを読み出し/書き込みで開いたときのチェック (適切な例)

```
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"
}
```

Tcl エラーは、error コマンドを使用しても生成できます。たとえば、catch コマンドで検出した TCL_ERROR を上位に伝搬する場合に使用できます。error コマンドは、スクリプトでコーナー ケースがサポートされていない場合や、コードで予期されない状況が発生した場合に、TCL_ERROR を生成するために使用できます。

たとえば次のプロシージャは、ファイルの内容を返すか、ファイルを開くことができない場合は TCL_ERROR を生成します。

```
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
}
```

プロシージャ get_file_content を catch コマンドで呼び出すと、発生する可能性のあるエラーを検出できます。

```
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 "
}
```

例 2 も、例 4 に示すように、不正な状況が発生した場合に TCL_ERROR を生成するよう改善できます。

例 4 : Vivado オブジェクトが有効であるかをチェックし、有効でない場合は TCL_ERROR を生成 (改善後)

```
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
}
```

環境変数へのアクセス

Tcl グローバル変数 env を使用すると、環境変数に読み出し専用モードでアクセスできます。env 変数は、Tcl インタープリター内でスタートアップ時に自動的に作成および初期化される Tcl 配列です。

注記：初期化後は、env 変数への変更は Tcl インタープリター外の環境には適用されません。同様に、Tcl インタープリターの開始後に環境変数に加えられた変更は、env 変数には反映されません。

env 配列のキーは、Vivado Design Suite の開始時点の環境変数です。キーでは大文字/小文字が区別されます。

次に例を示します。

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

すべての Unix 環境変数のリストを取得するには、次の構文を使用します。

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

環境変数 (env 配列へのキー) が存在するかは、info コマンドを使用して確認できます。たとえば MYVARNAME を確認するには、次の構文を使用します。

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

env 配列はグローバル変数なので、グローバルとして宣言すると、プロシージャ内で参照できます。

次に例を示します。

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

カスタム デザイン ルール チェック (DRC) の作成

Vivado Design Suite では、Tcl でカスタム デザイン ルール チェック (DRC) を定義し、使用できます。カスタム DRC を作成するには、次の手順に従います。

1. デザイン オブジェクトまたはデザイン オブジェクトの属性、およびデザイン ルールを定義するチェック関数を取得する Tcl チェッカー プロシージャを記述します。Tcl チェッカー プロシージャは別の Tcl スクリプトで定義し、report_drc を実行する前に Vivado Design Suite に読み込む必要があります。Tcl チェッカー プロシージャ内では、create_drc_violation コマンドを使用してデザインでルールをチェックした際に検出される違反を指定します。create_drc_violation コマンドは違反オブジェクトをインメモリ デザインに作成するので、そのプロパティをレポートして処理できます。
2. create_drc_check コマンドを使用して、Tcl チェッカー プロシージャをユーザー定義 DRC に関連付けます。report_drc コマンドを実行する際、このルールを名前で呼び出します。
3. create_drc_ruledeck コマンドを使用して、DRC ルール デックを作成します (オプション)。DRC ルール デックは、複数の DRC をグループ化したものです。作成した DRC ルール デックには、add_drc_ruledeck コマンドを使用して、ユーザー定義 DRC およびあらかじめ定義された DRC を追加できます。
4. report_drc コマンドを実行し、デザインをデザイン ルールに対してチェックします。report_drc を実行する際、DRC ルール デック、ユーザー定義デザイン ルール チェック、またはあらかじめ定義された DRC を指定できます。

Tcl チェッカー プロシージャの作成

Tcl チェッカー プロシージャは、チェックするデザイン オブジェクトを選択し、デザイン オブジェクトに対して必要なテストまたは評価を実行して、エラーに関連するオブジェクトを特定する DRC 違反オブジェクトを返します。

次の Tcl スクリプトは、WRITE_B バスの幅をチェックする dataWidthCheck Tck チェッカー プロシージャを定義しています。report_drc を実行する前に、この Tcl スクリプト ファイルを Vivado Design Suite に読み込む必要があります。Tcl チェッカー プロシージャの詳細は、17 ページの「Tcl スクリプトの読み込みと実行」を参照してください。

```
# 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
```

proc 定義からわかるように、dataWidthCheck プロシージャに引数はなく、必要な情報はすべてデザインから取得されます。空のリスト変数 $vios を作成し、create_drc_violation コマンドで返された違反オブジェクトを保存します。

dataWidthCheck プロシージャは、デザインのすべての BRAM セルに対して WRITE_WIDTH_B プロパティの評価を実行します。BRAM セルの WRITE_WIDTH_B が 36 を超える場合、DRC 違反が指定のメッセージ ($msg) で作成されます。メッセージには、セルのプレースホルダー値 %ELG と バス幅 $bwidth が含まれます。dataWidthCheck プロシージャでは、create_drc_violation コマンドで 1 つのオブジェクト $bram のみが返され、メッセージ文字列に定義されている %ELG プレースホルダーに代入されます。create_drc_violation コマンドでは、ネットリスト オブジェクト (%ELG)、クロック領域 (%CRG)、デバイス サイト (%SIG)、およびパッケージ I/O バンク (%PBG) のプレースホルダーがサポートされます。

重要：create_drc_violation コマンドで渡されるオブジェクトの順序とタイプが、create_drc_check コマンドの -msg の指定と一致している必要があります。

WRITE_WIDTH_B プロパティの幅が指定の値を超えている BRAM が検出されるたびに、create_drc_violation で違反オブジェクトが作成されます。違反オブジェクトは、名前は関連付けられている Vivado Design Suite の DRC ルールの名前と同じになり、あらかじめ定義されたメッセージ文字列を含み、ルールに違反するオブジェクトを特定します。デザイン ルール違反で返されるオブジェクトには、セル、ポート、ピン、ネット、クロック領域、デバイス サイト、パッケージ I/O バンクなどがあります。違反からのメッセージ文字列には、特定のプロパティ値などのほかの情報を含めることもでき、DRC レポートに必要な詳細情報を提供できます。

違反が検出された場合、dataWidthCheck プロシージャから report_drc コマンドにチェックの結果を通知するエラー コードが返されます。

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

エラー コードに加え、$vios 変数により、プロシージャで作成された違反オブジェクトのリストが返されます。

DRC チェックの作成

Tcl チェッカー プロシージャを定義したら、Vivado Design Suite 内での DRC レポート システムの一部として DRC を定義する必要があります。

まず create_drc_check コマンドを使用して、新しいデザイン ルールを登録します。このコマンドでは、ユーザー定義ルール チェックに固有の名前を指定する必要があります。この名前は、Tcl チェッカー プロシージャで作成した違反の名前と一致している必要があります。DRC ルール デックにチェックを追加する際や report_drc を実行する際に、この固有名を指定する必要があります。先ほど定義した dataWidthCheck Tcl チェッカー プロシージャでは、create_drc_violation コマンドで RAMW-1 という名前を使用しています。また、create_drc_check コマンドでは、ルールをチェックしたときに実行する Tcl チェッカー プロシージャの名前も指定する必要があります。上記の例では、-rule_body の引数として dataWidthCheck を指定し、report_drc を実行する前に Vivado Design Suite に読み込む必要があります。

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

DRC をカテゴリにグループ化し、レポート用にルールの説明を指定することもできます。

違反が検出されたときに DRC レポートに追加するメッセージを定義できます。デフォルトでは、Tcl チェッカー プロシージャの create_drc_violation で作成されたメッセージが DRC オブジェクトに渡されます。この場合、create_drc_violation で定義されたメッセージが DRC レポートに記述されます。

DRC オブジェクトには is_enabled プロパティがあり、set_property コマンドを使用して TRUE または FALSE に設定できます。新しいルール チェックを作成すると、is_enabled プロパティはデフォルトで TRUE に設定されます。report_drc を実行したときに DRC が使用されないようにするには、is_enabled プロパティを FALSE に設定します。

DRC ルール デックの作成

複数の関連する DRC チェックを一緒に実行できるように、DRC ルール デックにグループ化できます。これには、create_drc_ruledeck コマンドを使用して DRC ルール デックを作成し、add_drc_ruledeck および remove_drc_checks コマンドを使用して、DRC を追加および削除します。1 つの DRC ルール デックにユーザー定義のチェックとあらかじめ定義されたチェックを含めることができます。次に、myrules という DRC ルール デックを作成し、DRC を追加および削除する例を示します。

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

注記 : DRC の is_enabled プロパティが FALSE に設定されている場合、その DRC は report_drc を実行したときに DRC ルール デックの一部として実行されません。DRC ルール デックから DRC を削除するよりも、DRC をディスエーブルにする方が良い場合もあります。

カスタム DRC のレポート

ユーザー定義 DRC は、report_drc コマンドを使用して、個別に、ほかのルールと一緒に、または DRC ルール デックの一部として実行できます。次に、上記の例で定義された RAMW-1 ルールを個別に、ほかのルールと一緒に、または DRC ルール デックの一部として実行する例を示します。

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

report_drc でチェックが実行されるようにするには、ルール チェックの is_enabled プロパティを TRUE に設定しておく必要があります。

DRC 説明スクリプト

DRC ルール名または DRC ルールのパターンを見たときに、そのルールの機能の説明が必要な場合があります。これには、DRC オブジェクトのプロパティをレポートします。

次のスクリプト例は、入力して DRC ルールの検索パターンを指定することにより、各ルールの説明と重要度を表示します。パターンに一致するルール場ない場合は、エラー メッセージが表示されます。

```
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
}
```

Vivado 内に組み込まれている Tcl パッケージも多数あり、このスクリプト例では struct::matric パッケージを使用してサマリ表が見やすく表示されるようにしています。

次に、explain_drc プロシージャの出力例を示します。

```
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 スクリプト記述のヒント

いくつかの規則に従うことで、Tcl スクリプトの実行時間と効率を向上できます。次に、Vivado Design Suite で Tcl スクリプト機能を使用する際に、ランタイムを短縮し、メモリ使用量を抑えるための推奨事項を示します。

ネスト処理を使用したパフォーマンスの向上

Tcl コンソールから Tcl コマンドを実行すると、コマンドはまず Tcl インタープリターのレベルで処理されます。構文エラーがない場合、コマンドはその後 C++ レベルで実行されます。コマンドで値が返される場合、C++ コードにより戻り値がソフトウェアのいくつかの層を介して Tcl インタープリターに送信されます。Tcl インタープリターと下位 C++ コードを行き来することが、実行時間に影響します。同じコマンド内でネスト処理を使用すると、ネストされたコマンドは C++ コードから直接呼び出され、コマンド全体が完了してから C++ コードから Tcl インタープリターに戻ります。

たとえば、次のようなコードがあるとします。

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

次のコードは、上記のコードよりも高速に処理されます。

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

これは、最初のコードでは中間 Tcl 変数 nets が作成されるからです。

ただし、これらの変数の結果がコードのほかの部分で再利用される場合、中間 Tcl 変数を作成する方が有利なこともあります。

オブジェクトのキャッシュ

オブジェクトまたはオブジェクトのリストを Tcl 変数にキャッシュし、再利用します。

たとえば、同じネットのリストをスクリプトで複数回使用する場合は、同じクエリを繰り返し実行するのは効率的ではありません。Vivado ツールの Tcl コマンドは効率的に実行されますが、Tcl クエリを実行するたびに Tcl インタープリターとアプリケーションの 下位 C++ コードの間を行き来することになります。この C++/Tcl のアクセスに時間がかかるので、できる限り避けるようにします。

Vivado ツールの異なるフィルター機能をできる限り利用します。効果的な検索パターン、`-of_objects` オプション、`-filter` オプション、および `filter` コマンドを使用することで実行時間を短縮できます。これらの機能はアプリケーションの下位にインプリメントされており、実行時間およびメモリの点で非常に効率的です。

クエリの結果をキャッシュして、そのオブジェクトのリストに対して `filter` コマンドを実行してオブジェクトのサブリストを作成できます。また、インメモリ デザインにアクセスせずに、標準 Tcl コマンドを使用して Tcl 変数に代入された結果を解析できます。

```
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
```

オブジェクト名と NAME プロパティ

デザイン オブジェクトを必要とする Tcl コマンドと、文字列入力を必要とする Tcl コマンドがあります。Vivado Design Suite では、文字列引数を必要とする Tcl コマンドであっても、デザイン オブジェクトを直接渡すことができます。この場合、デザイン オブジェクトの階層名が文字列として Tcl コマンドに渡されます。オブジェクトの NAME プロパティを取得して Tcl コマンドに渡す必要はありません。

たとえば、次の `regexp` コマンドでは、2 つの if 文は同等で、オブジェクトの名前が渡されます。

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

上記の例では、最初のコードの方が読みやすいだけでなく、オブジェクトのプロパティを取得する必要がないので実行時間も短くなります。2 番目の例では `get_property` コマンドにより、オブジェクト プロパティを取得するため、Tcl インタープリターと下位 C++ アプリケーション コードの間にアクセスが発生します。これを複数のオブジェクトに対してループで実行すると、Tcl スクリプトの実行時間が大幅に増加します。

オブジェクトのリストのフォーマット

get_* コマンドから返されたリストはフォーマットされておらず、stdout にスペースで区切られて 1 行で表示されます。この例を次に示します。

```
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
```

このフォーマットされていないリストでは、Tcl コンソールおよび Vivado IDE で何が返されたかを確認するのが困難です。リストの各アイテムを個別の行に表示するには、次のようにコマンドを join コマンドにネストし、改行文字 \n を追加します。

```
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
```

get_* コマンドで返されるリストは、join コマンドでは変更されません。

Vivado Tcl コマンドをオプションで検索

次のプロシージャ findCmd を使用すると、Vivado Design Suite のすべての Tcl コマンドの構文が検索され、指定のオプションをサポートするコマンドがリストされます。

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

たとえば、-return_string オプションをサポートする Vivado ツール コマンドを検索するには、次のコマンドを使用します。

```
findCmd return_string
```

効率的なコードの記述

実行時間を短縮する方法の 1 つに、コンテナーに含まれる各要素に対してループ内のコマンドを実行するのではなく、コンテナーを作成してコマンドをコンテナー全体に対して実行する方法があります。次に、「カスタム デザイン ルール チェック (DRC) の作成」セクションに類似した例を示します。

非効率なコード：

```
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
```

効率的なコード：

```
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]
```

上記のコードをさらに短く効率的に記述するには、get_cells コマンドにフィルターを適用します。これにより、個別のチェックを含む foreach ループを実行する必要がなくなりますが、多少複雑なフィルターが必要になります。

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

付録 A

その他のリソース

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、次のザイリンクス サポート サイトを参照してください。

http://japan.xilinx.com/support

ザイリンクス資料で使用される用語集は、次を参照してください。

http://japan.xilinx.com/company/terms.htm

ソリューション センター

デバイス、ツール、IP のサポートについては、ザイリンクス ソリューション センターを参照してください。トピックには、デザイン アシスタント、アドバイザリ、トラブルシュート ヒントなどが含まれます。

参考資料

1. 『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835)
2. 『Vivado Design Suite ユーザー ガイド : 入門』(UG910)
3. 『Vivado Design Suite ユーザー ガイド : 制約の使用』(UG903)
4. 『Vivado Design Suite 移行手法ガイド 』(UG911)
5. 『Vivado Design Suite ユーザー ガイド : デザイン フローの概要』(UG892)
6. 『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893)
7. Vivado Design Suite 2013.4 資料ページ : http://japan.xilinx.com/support/documentation/dt_vivado2013-4.htm
8. Tcl Developer Xchange : www.tcl.tk