Vivado Design Suite ユーザー ガイド

Tcl スクリプト機能の使用

UG894 (v2012.4) 2012 年 12 月 18 日

XILINX®

本資料は英語版 (v2012.4) 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	第 1 章の「Tcl フック スクリプトの定義」を追加

目次

XILINX

第 1 章

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。

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。簡単に拡張できることから、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) を参照するか、または Vivado のヘルプ機能を使用してください。

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。

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/tcl/tutorial/tcltutorial.html を参照してください。

このガイドでは、Tcl コマンドの例、Tcl スクリプト、および Vivado Design Suite での戻り値が示されています。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。

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評価されません。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 2012.3
```

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、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 と異なります。角かっこは Verilog および VHDL 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、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
```

サンプル スクリプト : 非プロジェクト コンパイル フロー

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、9 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 post_synth_critpath_report.csv
#
# STEP#4: run logic optimization, placement and physical logic optimization,
# write design checkpoint, report utilization and timing estimates
#
opt_design
reportCriticalPaths post_opt_critpath_report.csv
place_design
report_clock_utilization -file $outputDir/clock_util.rpt
#
# Optionally run optimization if there are timing violations after placement
if {[get_property SLACK [get_timing_paths -max_paths 1 -nworst 1 -setup]] < 0}
{
    puts “Found setup timing violations => running physical optimization”
    phys_opt_design
}
write_checkpoint -force $outputDir/post_place.dcp
report_utilization -file $outputDir/post_place_util.rpt
report_timing_summary -file $outputDir/post_place_timing_summary.rpt
#
# STEP#5: run the router, write the post-route design checkpoint, report the routing
# status, report timing, power, and DRC, and finally save the Verilog netlist.
#
route_design
write_checkpoint -force $outputDir/post_route.dcp
report_route_status -file $outputDir/post_route_status.rpt
report_timing_summary -file $outputDir/post_route_timing_summary.rpt
report_power -file $outputDir/post_route_power.rpt
report_drc -file $outputDir/post_imp_drc.rpt
write_verilog -force $outputDir/cpu_impl_netlist.v -mode timesim -sdf_anno true
#
# STEP#6: generate a bitstream
#
write_bitstream -force $outputDir/cpu.bit
```

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) 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) を参照してください。

- **手順 3** : デザインを特定のパッケージ用に合成します。

  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。

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。読み込む Tcl スクリプトを記述し、Vivado IDE 内から実行したり、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、結果を返すデザイン 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。`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は、空のリストが返されます。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、コマンドを完了するには、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 2 つのデフォルト値を設定しています。`corner` のデフォルト値は `Slow`、`delayType` のデフォルト値は `Max` です。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
}
```

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 10 Fast
%> reportWorstViolations 10 Slow Min
```

次のプロシージャの例には定義済みの引数 `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
```

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。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、29 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、15 ページの「フィルター結果」を参照してください。

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ほかの Tcl コマンドや Tcl プロシージャに渡すことができます。ただし、デザイン オブジェクトを含む変数が文字列を必要とする Tcl コマンドに渡された場合は、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。Vivado Tcl コンソールにデザイン オブジェクトのみを返すコマンドで返されたオブジェクトの名前が表示されるのは、このためです。次の例に、タイミング パス オブジェクトを変数 `$path1` 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` コマンドではオブジェクトの名前のみが表示され、`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 path1 [get_timing_paths -delay_type max]
{usbEngine1/u4/inta_reg/C -->
cpuEngine/or1200_du/tbar_ram/ramb16_s36_s36/DIBDI[12]}

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

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

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、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したら、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*]
```

デザイン階層での検索

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。

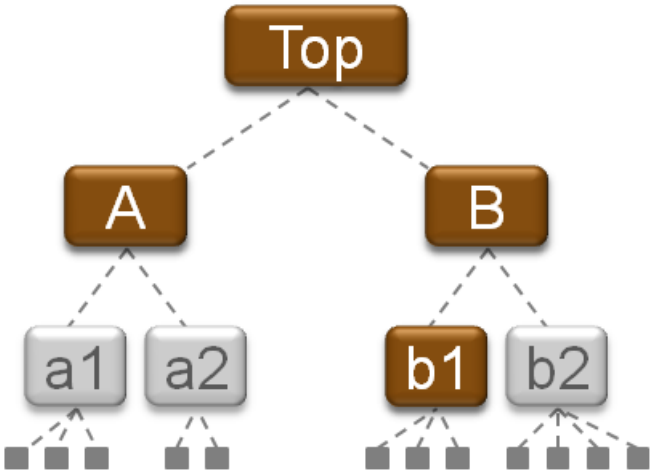

図 1-1 : デザイン階層の検索

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 1-1 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 in the hierarchy.
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。名前の検索パターンを指定する場合は、12 ページの図 1-1 を使用した次の例に示すように、階層区切り文字を含めないでください。

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

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。-regexp の詳細は、『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) を参照してください。

-hierarchical を使用した検索は、current_instance コマンドを使用して各階層レベルを指定し、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、12 ページの図 1-1 を使用して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
}
```

ピンの検索

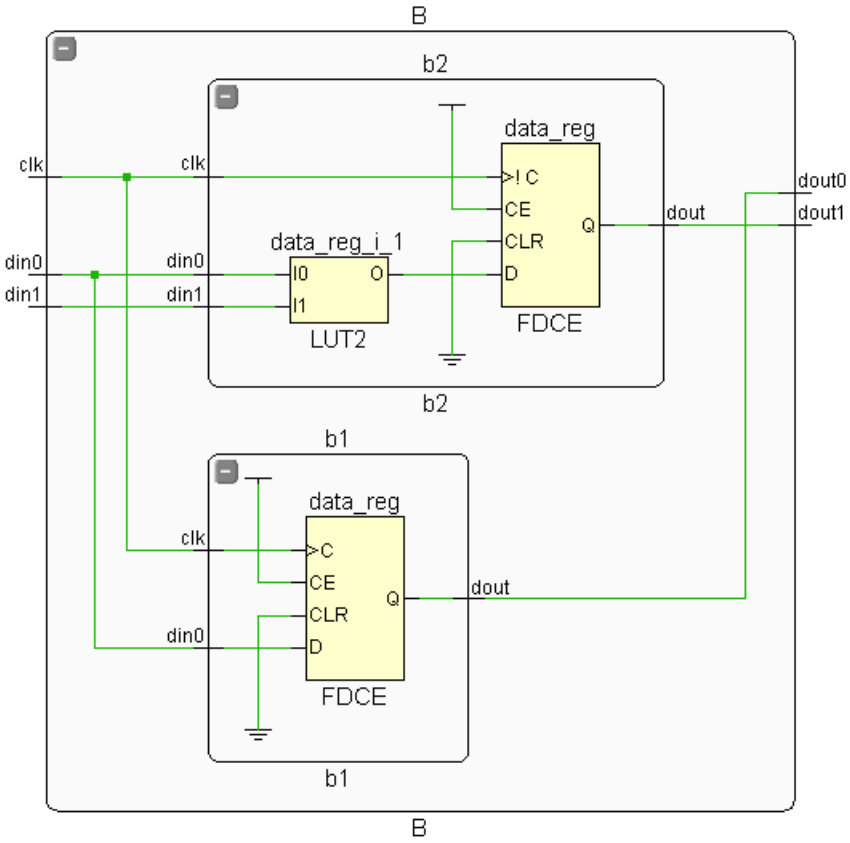

図 1-2 : ピン名の検索

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 1-2 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、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、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。

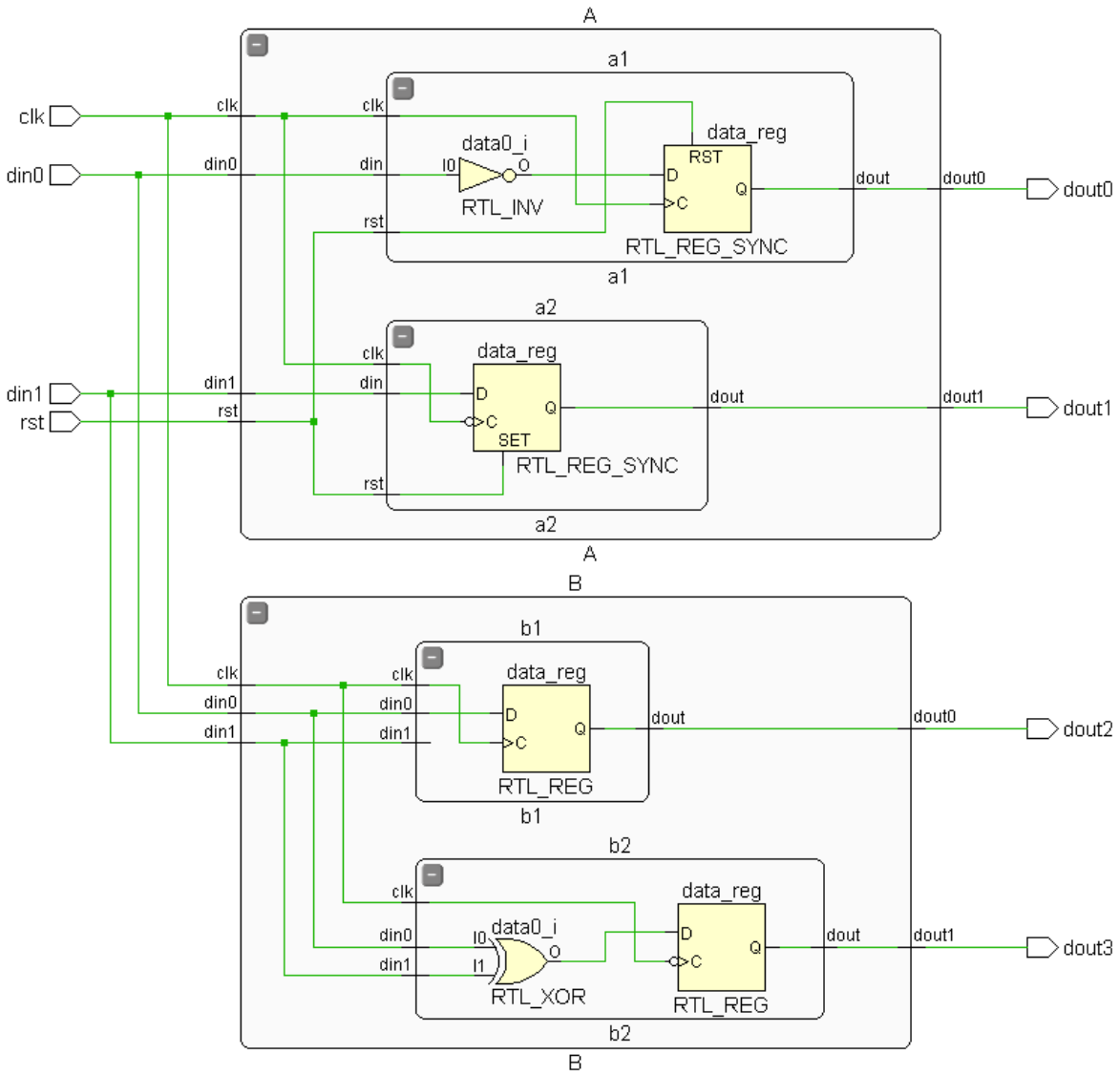

図 1-3 : 階層デザインの検索

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

-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、完全な階層名が「B/b*」と一致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} ]
```

-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。

```
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。

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 1-4 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。

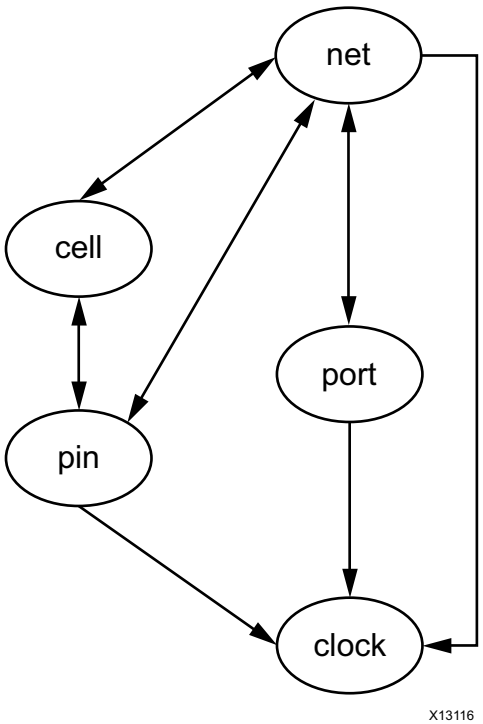

図 1-4 : Vivado Design Suite でのオブジェクト間の関連性

注記 : これは概念的に図示したものであり、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。

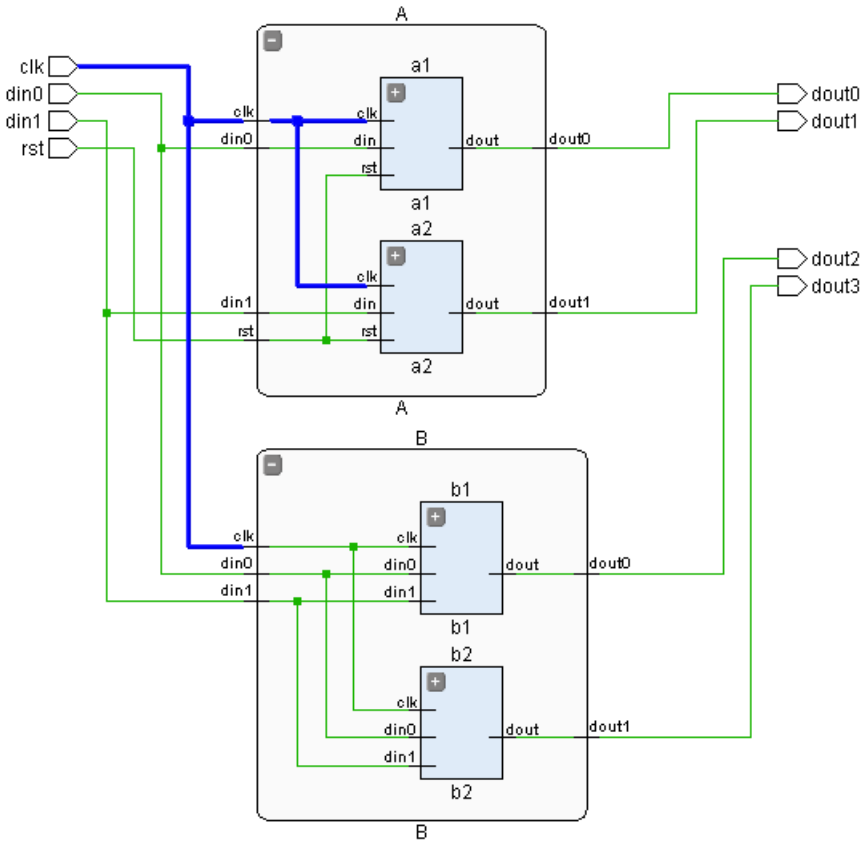

図 1-5 : 関連性を使用したオブジェクトの検索

次の例では、図 1-5 に示すように、instance 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
```

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、同じように機能します。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
...
```

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。Tcl シェルには、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、20 ページの「ファイルへのアクセス」を参照してください。

多くの 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。出力を文字列変数に代入すると、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 ]
```

Tcl シェルには、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、22 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、使用可能な 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、作成はされません。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]
```

カスタム DRC の作成

Vivado Design Suite では、Tcl でカスタム デザイン ルール チェックを定義し、使用できます。プロセスは次のとおりです。

1. デザイン オブジェクトまたは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 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。
2. Tcl チェッカー内で 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。このコマンドにより違反オブジェクトがインメモリ デザインに作成され、そのプロパティをレポートしてその後の処理に使用できます。
3. create_drc_check コマンドを使用して、DRC ルールの名前を -rule_body で指定した Tcl チェッカー プロシージャに関連付けるユーザー定義 DRC ルール チェックを定義します。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。
4. create_drc_ruledeck コマンドを使用してルール デックを作成し、add_drc_checks コマンドを使用してユーザー定義 DRC ルールをルール デックに追加します。
5. ルール デックまたはユーザー定義 DRC ルールを指定して report_drc を実行し、違反がないかどうかをチェックします。

Tcl DRC チェッカーの記述

デザイン ルールを定義する Tcl スクリプトである Tcl チェッカー プロシージャは、まずチェックするデザイン オブジェクトを選択し、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 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 チェッカーの詳細は、26 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ありません。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。

チェックするデザイン ルールに関連するデザイン オブジェクトを選択し (この場合は BRAM)、各セルの WRITE_WIDTH_B プロパティを取得して、値が 36 を超える場合は違反とします。

違反が検出されると、セルのプレースホルダー値 %ELG と バス幅 $bwidth を含むメッセージ $msg を作成します。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。

重要: create_drc_violation コマンドで渡される順序とタイプが、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、dataWidthCheck プロシージャで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。dataWidthCheck プロシージャに示すように、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。

Vivado Tcl DRC コマンド

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 チェック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。このコマンドを使用すると、ユーザー定義ルール チェックに固有の名前または略称を指定できます。これらの名前は、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。先ほど定義した dataWidthCheck 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るので、DRC チェックを作成する際はこの名前を指定する必要があります。

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

DRC チェック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 プロシージャの create_drc_violation で作成されたメッセージが DRC チェック オブジェクトに渡されます。この場合、create_drc_violation の -rule_body で定義されたメッセージが DRC レポートに記述されます。

最後に、-rule_body オプションを使用して、ルールをチェックするときに Vivado Design Suite で実行する Tcl プロシージャの名前を指定します。必要なチェックが完全に定義されるようにするため、report_drc コマンドを実行する前にプロシージャを Vivado Design Suite に読み込んでおく必要があります。

プロシージャが読み込まれていれば、report_drc コマンドを使用して DRC ルール チェックを個別にまたはほかのルールと共に実行できます。create_drc_ruledeck および add_drc_checks コマンドを使用して、一緒に実行する関連のルールをグループ化したルール デックを作成できます。ルール デックから DRC チェックを削除するには、remove_drc_checks コマンドを使用します。

DRC ルール チェック 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 を実行したときにルール チェック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。これにより、新しい DRC チェックを作成し、add_drc_checks を使用してルール デックに追加した場合に、そのチェックをルール デックから削除せずにイネーブルにしたりディスエーブルにしたりできます。

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
```

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 フローが処理されます。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) 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 1-6)。

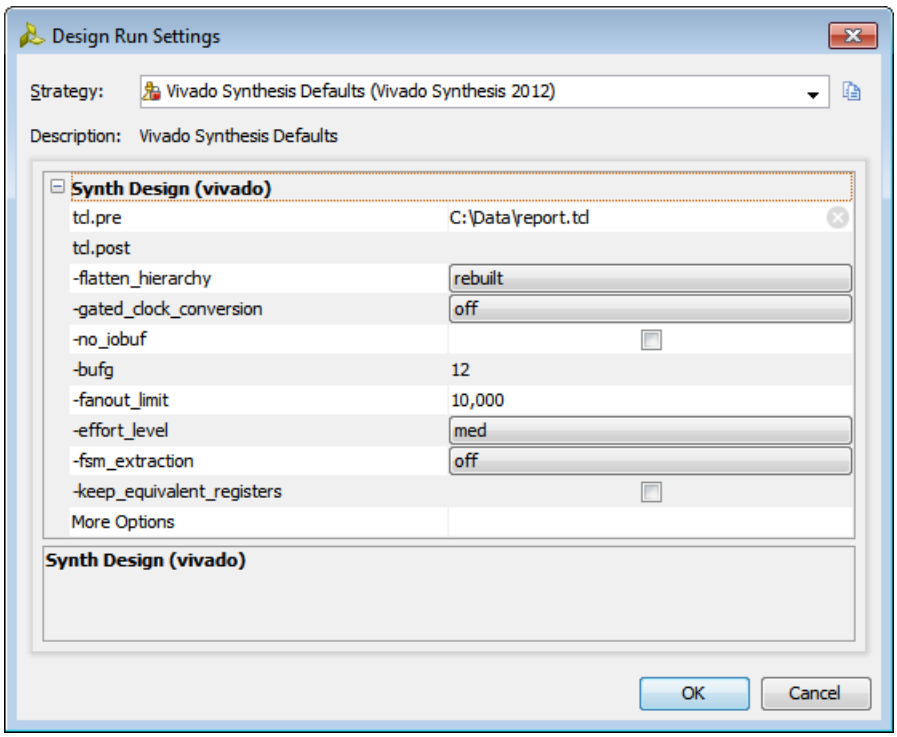

図 1-6 : Tcl フック スクリプトの指定

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。

```
get_property DIRECTORY [current_project]
get_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) 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の推奨事項を示します。

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
```

XILINX

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。

リファレンス

Vivado Design Suite 2012.4 の資料

http://japan.xilinx.com/support/documentation/dt_vivado2012-4.htm

Tcl Developer Xchange

www.tcl.tk