

Vivado Design Suite ユーザーガイド: ロジック シミュレーション

UG900 (v2020.1) 2020 年 6 月 3 日

この資料は表記のバージョンの英語版を翻訳したもので、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。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2020 年 6 月 3 日 バージョン 2020.1	
第 3 章: サードパーティ シミュレータを使用したシミュレーション	表 8 および表 9 を追加
資料全体	第 2 章の「シミュレーション設定」をアップデート
資料全体	第 4 章の「[Objects] ウィンドウ」をアップデート

目次

改訂履歴.....	2
第 1 章: ロジック シミュレーションの概要	7
サポートされるシミュレータ.....	7
シミュレーション フロー.....	8
言語および暗号化サポート.....	10
第 2 章: シミュレーションの準備	11
テストベンチおよびスティミュラス ファイルの使用.....	11
シミュレータのインストール場所の指定.....	12
シミュレーション ライブラリのコンパイル.....	13
ザイリンクス シミュレーション ライブラリの使用.....	18
シミュレーション設定.....	26
シミュレーション ソース ファイルの追加または作成.....	30
ネットリストの生成.....	32
第 3 章: サードパーティ シミュレータを使用したシミュレーション	34
Vivado IDE でのサードパーティ シミュレータを使用したシミュレーションの実行.....	36
消費電力解析用の SAIF の出力.....	39
VCD の出力.....	40
IP のシミュレーション.....	41
統合されたシミュレーションの実行中のカスタム DO ファイルの使用.....	42
バッチ モードでのサードパーティ シミュレータの実行.....	44
第 4 章: Vivado シミュレータを使用したシミュレーション	45
Vivado シミュレータの実行.....	45
論理およびタイミング シミュレーションの実行.....	62
シミュレーション結果の保存.....	65
複数のシミュレーション run の区別.....	65
シミュレーションを閉じる.....	65
シミュレーション起動スクリプト ファイルの追加.....	66
シミュレーション メッセージの表示.....	67
launch_simulation コマンドの使用.....	68
デザイン変更後のシミュレーションの再実行.....	69
保存されたシミュレータのユーザー インターフェイス設定の使用.....	70
第 5 章: Vivado シミュレータを使用したシミュレーション波形の解析	72
波形設定と波形ウィンドウの使用.....	72
前に保存したシミュレーション run を開く.....	73

波形設定の HDL オブジェクト.....	74
波形のカスタマイズ.....	77
波形表示の制御.....	81
波形の分類.....	85
波形の解析.....	87
AXI インターフェイス トランザクションの解析.....	91
第 6 章: Vivado シミュレータを使用したデザインのデバッグ	104
ソース レベルでのデバッグ.....	104
波形オブジェクトを特定値に設定.....	108
Vivado シミュレータを使用した消費電力解析.....	115
Tcl コマンド report_drivers の使用.....	117
VCD 機能の使用.....	118
Tcl コマンド log_wave の使用.....	118
[Objects] ウィンドウ、波形ウィンドウ、テキスト エディターでの信号のクロスプローブ.....	119
第 7 章: Vivado シミュレータでのバッチまたはスクリプト モードを使用したシミュレーション	126
シミュレーション ファイルとスクリプトのエクスポート.....	126
バッチ モードでの Vivado シミュレータの実行.....	131
デザイン スナップショットのエラボレートおよび生成 (xelab).....	133
デザイン スナップショットのシミュレーション (xsim).....	144
スタンドアロン モードでの Vivado シミュレータの実行例.....	147
プロジェクト ファイル (.prj) の構文.....	148
定義済みマクロ.....	149
ライブラリ マップ ファイル (xsim.ini).....	149
シミュレーション モードの実行.....	150
Tcl コマンドとスクリプトの使用.....	152
export_simulation.....	153
export_ip_user_files.....	156
付録 A: コンパイル、エラボレーション、シミュレーション、ネットリスト、アドバンス オプション	159
コンパイル オプション.....	159
エラボレーション オプション.....	161
シミュレーション オプション.....	163
ネットリスト オプション.....	165
アドバンス シミュレーション オプション.....	166
付録 B: Vivado シミュレータでの SystemVerilog のサポート	167
特定のファイルで SystemVerilog を使用.....	167
テストベンチの機能.....	174
付録 C: UVM のサポート	183
付録 D: Vivado シミュレータの VHDL 2008 サポート	184

概要.....	184
コンパイルおよびシミュレーション.....	184
サポートされる機能.....	186
付録 E: Vivado シミュレータのダイレクト プログラミング インターフェイス (DPI).....	
概要.....	188
C コードのコンパイル.....	188
xsc コンパイラ.....	188
xelab を使用したコンパイル済み C コードの SystemVerilog への統合.....	190
C と SystemVerilog の境界で使用可能なデータ型.....	191
ユーザー定義型のマップ.....	192
svdpi.h 関数のサポート.....	193
Vivado Design Suite に含まれる DPI 例.....	200
付録 F: 特殊なケースの処理.....	
グローバル リセットとトライステートの使用.....	201
デルタ サイクルとレース コンディション.....	203
ASYNC_REG 制約の使用.....	204
コンフィギュレーション インターフェイスのシミュレーション.....	205
シミュレーションでのブロック RAM の競合チェックのディスエーブル.....	208
消費電力解析のための SAIF ファイルの出力.....	209
コンパイルまたはシミュレーションのスキップ.....	209
付録 G: Vivado シミュレータ Tcl コマンドの値の規則.....	
文字列の値の解釈.....	210
Vivado Design Suite シミュレーション ログック.....	210
付録 H: Vivado シミュレータの混合言語サポートおよび例外.....	
混合言語シミュレーションの使用.....	212
VHDL 言語サポートの例外.....	217
Verilog 言語サポートの例外.....	219
付録 I: Vivado シミュレータ クイック リファレンス ガイド.....	
付録 J: ザイリンクス シミュレータ インターフェイスの使用.....	
ダイナミック リンク用に XSI 関数を準備.....	225
テストベンチ コードの記述.....	227
C/C++ プログラムのコンパイル.....	227
デザイン共有ライブラリの準備.....	228
XSI 関数参照.....	228
Vivado シミュレータの VHDL データ形式.....	233
Vivado シミュレータの Verilog データ形式.....	236
付録 K: その他のリソースおよび法的通知.....	
ザイリンクス リソース.....	238

Documentation Navigator およびデザイン ハブ.....	238
参考資料.....	238
サードパーティ シミュレータに関する情報へのリンク.....	239
トレーニング リソース.....	239
お読みください: 重要な法的通知.....	240

ロジック シミュレーションの概要

シミュレーションは、実際のデザインの動作をソフトウェア環境でエミュレートするプロセスです。シミュレーションを実行すると、スティミュラスを挿入してデザイン出力を観察することにより、デザインの機能を検証できます。

この章では、シミュレーション プロセスの概要および Vivado® Design Suite のシミュレーション オプションについて説明します。

シミュレーション プロセスには次の操作が含まれます。

- テストベンチの作成、ライブラリの設定、およびシミュレーション設定の指定。
- ネットリスの生成 (合成後またはインプリメンテーション後のシミュレーションを実行する場合)。
- Vivado シミュレータまたはサードパーティ シミュレータを使用したシミュレーションの実行。サポートされるシミュレータの詳細は、[サポートされるシミュレータ](#)を参照してください。

サポートされるシミュレータ

次の表に、Vivado Design Suite でサポートされるシミュレータを示します。

表 1: サポートされるシミュレータ

シミュレータ	バージョン	Vivado IDE との統合
Vivado® シミュレータ	2020.1	Vivado IDE に統合されており、Vivado IDE 内の各シミュレーションが表示されるウィンドウに含まれます。
Mentor Graphics 社 Questa Advanced Simulator	2019.4	○
Mentor Graphics 社 ModelSim Simulator	2019.4	○
Cadence 社 Incisive Enterprise Simulator (IES)	15.20.079	○
Synopsys 社 Verilog Compiler Simulator (VCS)	P-2019.06-SP1-1	○
Aldec Rivera-PRO Simulator	2019.10	○
Aldec Active-HDL	11.1a	×
Cadence 社 Xcelium Parallel Simulator	19.09.004	○

サードパーティ シミュレータのサポートされるバージョンについては、『Vivado Design Suite ユーザー ガイド: リリース ノート、インストール、およびライセンス』 ([UG973](#)) を参照してください。

Vivado IDE および Vivado Design Suite フローの詳細は、次を参照してください。

- 『Vivado Design Suite ユーザー ガイド: Vivado IDE の使用』 (UG893)
- 『Vivado Design Suite ユーザー ガイド: デザイン フローの概要』 (UG892)

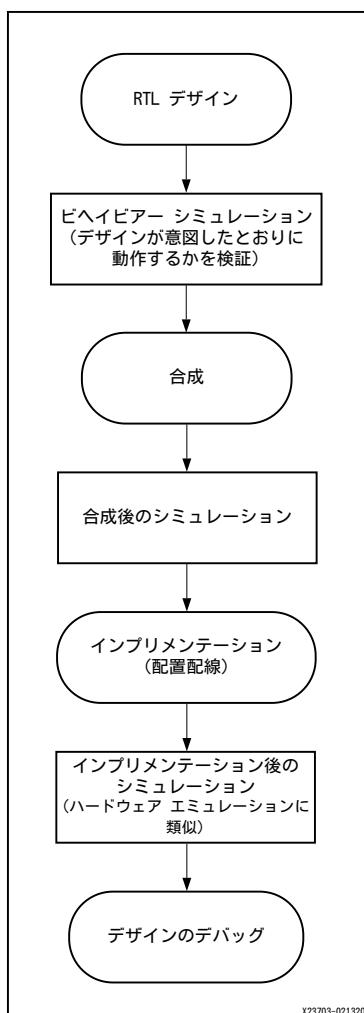
シミュレーション フロー

シミュレーションは、デザイン フローのさまざまな段階で実行できます。デザイン入力後の最初の段階の 1 つであり、最終的な機能とデザイン パフォーマンスを検証する、インプリメンテーション後の最後の段階の 1 つでもあります。

シミュレーションは、通常はデザイン機能とタイミングの両方の条件が満たされるまで繰り返す必要があります。

次の図は、典型的なデザインのシミュレーション フローを示しています。

図 1: シミュレーション フロー



レジスタ トランスファー レベルのビヘイビアー シミュレーション

レジスタ トランスファー レベル (RTL) のビヘイビアー シミュレーションには、次が含まれます。

- RTL コード
- インスタンスシート済み UNISIM ライブラリ コンポーネント
- インスタンスシート済み UNIMACRO コンポーネント
- UNISIM ゲートレベル モデル (Vivado 論理解析用)
- SECUREIP ライブラリ

RTL シミュレーションでは、合成またはインプリメンテーション ツールを実行する前にデザインをシミュレーション および検証できます。デザインは、モジュールまたはエンティティ、ブロック、デバイス、またはシステムとして検証できます。

RTL シミュレーションは、通常コード構文を検証し、コードが意図したとおりに機能するかどうかを確認するために実行されます。この段階では、デザインは主に RTL で記述されるので、タイミング情報は必要ありません。

デザインにデバイス ライブラリ コンポーネントがインスタンスシートされていなければ、RTL シミュレーションはアーキテクチャによって異なることはありません。ザイリンクスでは、インスタンス化をサポートするため、UNISIM ライブラリを提供しています。

デザインをビヘイビアー RTL で検証すると、早期に問題を解決できるので、デザイン サイクルを短縮できます。

最初のデザイン作成をビヘイビアー コードのみにすると、次が可能になります。

- より読みやすいコード
- より高速でシンプルなシミュレーション
- コードの移植 (別のデバイス ファミリに移行可能)
- コードの再利用 (今後別のデザインに同じコードを使用可能)

合成後のシミュレーション

合成済みのネットリストをシミュレーションすると、合成後のデザインが論理要件を満たしているか、意図したとおりに動作するかどうかを検証できます。このシミュレーション段階では、一般的ではありませんが、見積もられたタイミング値を使用してタイミング シミュレーションを実行できます。

論理シミュレーション ネットリストは、階層状になっており、プリミティブ モジュールおよびエンティティ レベルに展開されます。階層の最下位にはプリミティブおよびマクロ プリミティブが含まれます。

これらのプリミティブは、Verilog の場合は UNISIMS_VER ライブラリに、VHDL の場合は UNISIM ライブラリに含まれます。

関連情報

[UNISIM ライブラリ](#)

インプリメンテーション後のシミュレーション

インプリメンテーション後は論理シミュレーションまたはタイミング シミュレーションを実行できます。タイミング シミュレーションは、実際にデザインをデバイスにダウンロードするのに最も近く、インプリメント済みデザインが論理要件およびタイミング要件を満たし、デバイスで意図したとおりに動作することを確認します。



重要: 詳細なタイミング シミュレーションを実行することにより、次のようなエラーが見逃されるのを防ぐことができます。

- 次が原因で合成後およびインプリメンテーション後の機能が変更されている:
 - 不一致の原因となる合成プロパティまたは制約 (`full_case` および `parallel_case` など)
 - ザイリンクス デザイン制約 (XDC) ファイルでの UNISIM プロパティの適用
 - 異なるシミュレータによるシミュレーション中の言語の解釈
- デュアル ポート RAM の競合
- タイミング制約が適用されていないか、不適切に適用
- 非同期バスの操作
 - 最適化手法による機能の問題

言語および暗号化サポート

Vivado シミュレータでは、次がサポートされます。

- VHDL ([『IEEE Standard VHDL Language Reference Manual』](#) (IEEE-STD-1076-1993) を参照)
- Verilog ([『IEEE Standard Verilog Hardware Description Language』](#) (IEEE-STD-1364-2001) を参照)
- SystemVerilog ([『IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language』](#) (IEEE-STD-1800-2009) を参照)
- IEEE P1735 暗号化 ([『Recommended Practice for Encryption and Management of Electronic Design Intellectual Property \(IP\)』](#) (IEEE-STD-P1735) を参照)

シミュレーションの準備

この章では、ザイリンクス デバイスを Vivado® 統合設計環境 (IDE) でシミュレーションする際に必要なコンポーネントについて説明します。

シミュレーションを実行する前に次を設定します。

- 実行するシミュレーション アクションを反映するテストベンチを作成します。
- Vivado IDE のインストール ディレクトリを設定します (Vivado シミュレータを使用しない場合)。
- ライブラリをコンパイルします (Vivado シミュレータを使用しない場合)。
- 使用する必要のあるライブラリを選択して宣言します。
- ターゲット シミュレータ、シミュレーション最上位モジュール名、最上位モジュール (テスト中のデザイン) などのシミュレーション設定を指定し、シミュレーション セットを表示して、コンパイル、エラボレーション、シミュレーション、ネットリスト、およびアドバンス オプションを定義します。
- ネットリスを生成します (合成後またはインプリメンテーション後のシミュレーションを実行する場合)。

テストベンチおよびスティミュラス ファイルの使用

テストベンチは、次を実行するシミュレータ用に記述されるハードウェア記述言語 (HDL) コードです。

- デザインをインスタンス化および初期化。
- スティミュラスを生成および適用。
- デザイン出力結果を監視および論理精度を確認 (オプション)。

テストベンチは、ファイル、波形または表示画面へのシミュレーション出力を表示するためにも設定できます。テストベンチは、シンプルな構造で、特定の入力にスティミュラスを順次適用できます。

テストベンチは複雑にすることもできます。この場合、次を含めることができます。

- サブルーチン呼び出し
- 外部ファイルから読み出されるスティミュラス
- 条件付きスティミュラス
- その他の複雑な構造

テストベンチを使用すると、対話型シミュレーションと比較して次のような利点があります。

- デザイン プロセスでの反復シミュレーションが可能になります。
- テスト条件の資料が提供されます。

次に、効果的なテストベンチを作成するための推奨事項を示します。

- Verilog テストベンチ ファイルで `timescale を常に指定します。次に例を示します。

```
`timescale 1ns/1ps
```

- テストベンチ内のデザインへの入力すべてをシミュレーション 時間 0 に初期化し、シミュレーションが既知の値で正しく開始するようにします。
- 論理およびタイミング ベースのシミュレーションで使用されるデフォルトのグローバル セット/リセット (GSR) パルスを考慮し、ステイミュラス データを 100 ns 後に適用します。
- グローバル セット/リセット (GSR) が解除される前にクロック ソースを開始します。

テストベンチの詳細は、『効率的なテストベンチの記述』(XAPP199) を参照してください。



ヒント: テストベンチを作成する際は、合成後およびインプリメンテーション後のタイミング シミュレーションでは GSR パルスが自動的に発生することを考慮してください。このパルスにより、すべてのレジスタがシミュレーションの最初の 100 ns 間リセット状態になります。

関連情報

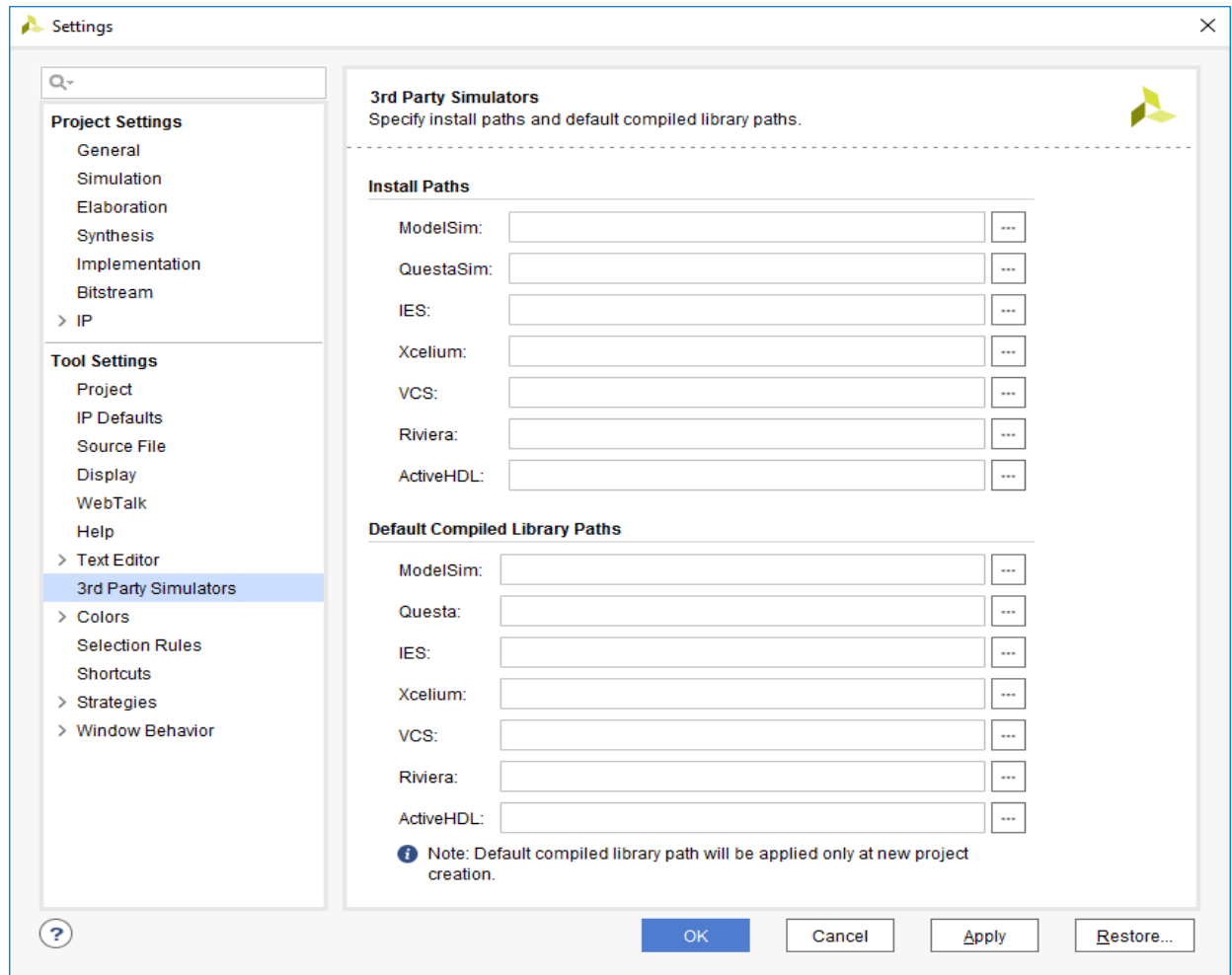
[グローバル リセットとトライステートの使用](#)

シミュレータのインストール場所の指定

インストール パスを定義します。

1. [Tools] → [Settings] → [Tool Settings] → [3rd Party Simulators] をクリックします。
2. [Settings] ダイアログ ボックスの [3rd Party Simulators] ページの [Install Paths] セクションにシミュレータのインストール パスを入力します。
3. [Default Compiled Library Paths] セクションでシミュレータのコンパイル済みライブラリのパスを指定します。このライブラリ パスは、後で指定することもできます。シミュレータ用にライブラリをコンパイルする方法については、[シミュレーション ライブラリのコンパイル](#)を参照してください。

注記: Vivado シミュレータは Vivado IDE の一部としてインストールされるので、Vivado シミュレータのインストール ディレクトリを設定する必要はありません。



シミュレーション ライブラリのコンパイル



重要: Vivado シミュレータを使用する場合は、シミュレーション ライブラリをコンパイルする必要はありませんが、サードパーティ シミュレータを使用する場合は、ライブラリをコンパイルする必要があります。

Vivado Design Suite では、シミュレーション モデルのファイルセットおよびライブラリを提供しています。デザインをシミュレーションする前に、これらのファイルをシミュレーション ツールでコンパイルしておく必要があります。シミュレーション ライブラリには、デバイスおよび IP のビヘイビアおよびタイミング モデルが含まれます。コンパイルされたライブラリは、複数のデザイン プロジェクトで使用できます。

コンパイル プロセス中、Vivado によりシミュレータでコンパイル済みライブラリを参照するために使用されるデフォルトの初期化ファイルが作成されます。compile_simlib コマンドは、ライブラリをコンパイルし、指定されたライブラリ出力ディレクトリにファイルを作成します。デフォルトの初期化ファイルには、参照するライブラリのパスを指定する変数、最適化、コンパイラ、シミュレータの設定が含まれています。正しい初期化ファイルがパスにならない場合は、ラインクス プリミティブを含むデザインをシミュレーションできません。

初期化ファイルの名前は、次のように、使用しているシミュレータによって異なります。

- Questa Advanced Simulator/ModelSim: `modelsim.ini`
- IES および Xcelium: `cds.lib`
- VCS: `synopsys_sim.setup`
- Riviera/Active-HDL: `library.cfg`

シミュレータ用のコンパイルされたライブラリ ファイルについては、サードパーティ シミュレーション ツールの資料を参照してください。



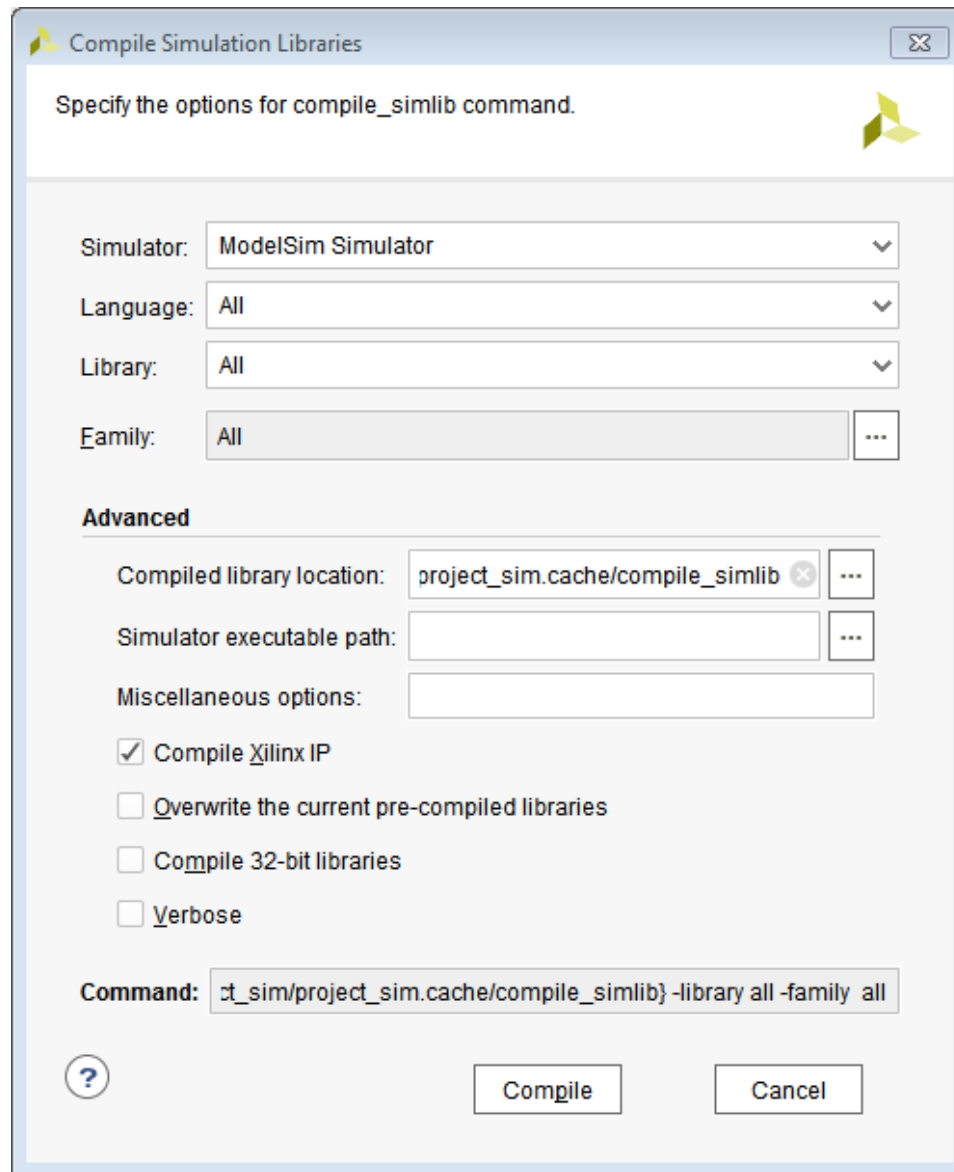
重要: ライブラリのコンパイルは、同じバージョンのツールでは通常は 1 回実行するだけです。Vivado ツールまたはシミュレータのバージョンを変更した場合は、ライブラリをコンパイルし直す必要があります。

ライブラリは、次に説明するように、Vivado IDE または Tcl コマンドを使用してコンパイルできます。

Vivado IDE を使用したシミュレーション ライブラリのコンパイル

[Tools] → [Compile Simulation Libraries] をクリックし、次の図に示すダイアログ ボックスを開きます。

図 2: [Compile Simulation Libraries] ダイアログ ボックス



次のオプションを設定します。

- [Simulator]: ドロップダウン リストからシミュレータを選択します。
- [Language]: ライブラリをコンパイルする言語を指定します。このオプションを指定しない場合、選択したシミュレータに対応する言語に設定されます。混合言語シミュレータの場合は、Verilog および VHDL ライブラリの両方がコンパイルされます。
- [Library]: コンパイルするシミュレーション ライブラリを指定します。デフォルトでは、`compile_simlib` コマンドですべてのシミュレーション ライブラリがコンパイルされます。
- [Family]: 選択したライブラリを指定したデバイス ファミリー用にコンパイルします。デフォルトでは、すべてのデバイス ファミリーが生成されます。

- [Compiled library location]: コンパイルされたライブラリを保存するディレクトリ パスを指定します。デフォルトでは、非プロジェクトモードではライブラリは現在の作業ディレクトリに保存され、プロジェクト モードでは `<project>/<project>.cache/compile_simlib` ディレクトリに保存されます。プロジェクト モードおよび非プロジェクト モードの詳細は、『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。



ヒント: Vivado シミュレータにプリコンパイル ライブラリが含まれるので、ライブラリ ディレクトリを指定する必要はありません。

- [Simulator executable path]: シミュレータ実行ファイルのディレクトリを指定します。このオプションは、ターゲット シミュレータが `$PATH` または `%PATH%` 環境変数で指定されていない場合や、`$PATH` または `%PATH%` 環境変数で指定されているパスとは別のパスを指定する場合に使用します。
- [Miscellaneous Options]: `compile_simlib` Tcl コマンドに追加のオプションを指定します。
- [Compile Xilinx IP]: ザイリンクス IP のシミュレーション ライブラリのコンパイルを有効または無効にします。
- [Overwrite current pre-compiled libraries]: 現在のコンパイル済みライブラリを上書きします。
- [Compile 32-bit libraries]: ライブラリをデフォルトの 64 ビット モードではなく、32 ビット モードでコンパイルします。
- [Verbose]: メッセージの非表示設定を一時的に解除し、コマンドからのすべてのメッセージを返します。
- [Command]: ダイアログ ボックスに入力したオプションと同等の Tcl コマンドを表示します。



ヒント: [Command] フィールドの値を使用すると、Tcl または非プロジェクト モードでシミュレーション ライブラリを生成できます。

Tcl コマンドを使用したシミュレーション ライブラリのコンパイル

シミュレーション ライブラリは、`compile_simlib` Tcl コマンドを使用してコンパイルすることもできます。詳細は、『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) の `compile_simlib` を参照するか、「`compile_simlib -help`」と入力してください。

次に、各サードパーティ シミュレータのコマンド例を示します。

- Questa Advanced Simulator: すべての言語、すべてのライブラリ、すべてのファミリに対して Questa 用のシミュレーション ライブラリを現在のディレクトリに生成します。

```
compile_simlib -language all -simulator questa -library all -family all
```

- ModelSim: ModelSim のシミュレーション ライブラリを `/a/b/c` に生成します。
`<simulator_installation_path>` は ModelSim の実行ファイルのパスです。

```
compile_simlib -language all -dir {/a/b/c} -simulator modelsim -
simulator_exec_path
{<simulator_installation_path>} -library all -family all
```

- IES: IES 用の Verilog 言語の UNISIM シミュレーション ライブラリを `/a/b/c` に生成します。

```
compile_simlib -language verilog -dir {/a/b/c} -simulator ies -library
unisim
-family all
```


- VCS: VCS 用の Verilog 言語の UNISIM シミュレーション ライブラリを /a/b/c に生成します。

```
compile_simlib -language verilog -dir {/a/b/c} -simulator vcs_mx -library
unisim
-family all
```

- Xcelium: Xcelium 用の Verilog 言語の UNISIM シミュレーション ライブラリを /a/b/c に生成します。

```
compile_simlib -language verilog -dir {/a/b/c} -simulator xcelium -
library unisim
-family all
```

compile_simlib のデフォルトの変更

config_compile_simlib Tcl コマンドを使用すると、compile_simlib コマンドで使用されるサードパーティ シミュレータのオプションを設定できます。

Tcl コマンド

```
config_compile_simlib [-cfgopt <arg>] [-simulator <arg>] [-reset] [-quiet]
[-verbose]
```

説明:

- -cfgopt <arg>: シミュレータの設定オプションを <simulator>:<language>:<library>:<options> の形式で設定します。
- -simulator: 設定を表示するシミュレータを指定します。
- -reset: 指定したシミュレータの設定をすべてリセットします。
- -quiet: [Tcl Console] ウィンドウにコマンド出力を表示せずにコマンドを実行します。
- -verbose: [Tcl Console] ウィンドウにコマンド出力をすべて表示します。

たとえば、UNISIM VHDL ライブラリのコンパイルに使用するオプションを変更するには、次のコマンドを入力します。

```
config_compile_simlib {cxl.modelsim.vhdl.unisim:-source -93 -novopt}
```



重要: compile_simlib コマンドは、ザイリンクス プリミティブおよびザイリンクス Vivado IP のシミュレーション モデルをコンパイルします。ザイリンクス Vivado IP コアは、IP を生成したときに出力ファイルとして提供されないで、compile_simlib コマンドを使用して作成されるコンパイル済みライブラリに含まれます。

MYVIVADO を使用してパッチされた IP リポジトリを新規出力ディレクトリにコンパイルする方法

パッチされた IP リポジトリが次のディレクトリにあるとします。

```
'/test/patched_ip_repo/data/ip/xilinx'
```

デフォルトでインストールされた IP リポジトリと MYVIVADO で指定されたりポジトリを新規出力ディレクトリにコンパイルするには、MYVIVADO 環境変数をこのパッチされた IP リポジトリに指定し、`compile_simlib` を実行します。`compile_simlib` は、デフォルトのインストール リポジトリと MYVIVADO で設定されたりポジトリからの IP ライブラリ ソースを処理します。

```
% setenv MYVIVADO /test/patched_ip_repo
% compile_simlib -simulator <simulator> -directory <new_clibs_dir>
```

MYVIVADO を使用してパッチされた IP リポジトリを既存の出力ディレクトリにコンパイルする方法

パッチされた IP リポジトリが次のディレクトリにあるとします。

```
'/test/patched_ip_repo/data/ip/xilinx'
```

MYVIVADO で指定されたりポジトリを既存の出力ディレクトリ (ライブラリは既にデフォルトでインストールされた IP リポジトリ用にコンパイル済み) にコンパイルするには、MYVIVADO 環境変数をこのパッチされた IP リポジトリに指定し、`compile_simlib` を実行します。`compile_simlib` は、MYVIVADO で設定されたりポジトリからの IP ライブラリ ソースを処理して既存の出力ディレクトリに配置します。

```
% setenv MYVIVADO /test/patched_ip_repo
% compile_simlib -simulator <simulator> -directory <existing_clibs_dir>
```

ザイリンクス シミュレーション ライブラリの使用

ザイリンクス シミュレーション ライブラリは、VHDL-93 および Verilog-2001 言語規格をサポートするシミュレータで使用できます。ライブラリには、ザイリンクス ハードウェア デバイスを正しくシミュレーションするのに必要な特定の遅延およびモデル情報が組み込まれています。

クロック エッジ内のブロックにはノンブロッキング割り当てを使用します。これ以外の場合、Verilog のブロッキング割り当てを使用してコードを記述します。同様に、プロセス内のローカル計算には変数割り当てを、プロセス間のデータフローが必要な場合は信号割り当てを使用します。

データがクロックと同時に変化する場合は、データ入力クロック エッジ後に発生するようシミュレータでスケジューリングされる可能性があります。この場合、データを最初のクロック エッジより前に供給するつもりであっても、そのデータは次のクロック エッジまで供給されません。



推奨: このような意図しないシミュレーション結果にならないようにするため、データ信号とクロック信号は同時に切り替えないようにしてください。

デザインにコンポーネントをインスタンス化する場合、シミュレータでコンポーネントの機能を記述したライブラリが参照されていないと、シミュレーションが正しく実行されません。ザイリンクス ライブラリは、モデルの機能に基づいてカテゴリに分類されます。

次の表に、ザイリンクスの提供するシミュレーション ライブラリを示します。

表 2: シミュレーション ライブラリ

ライブラリ名	説明	VHDL ライブラリ名	VHDL ライブラリ名
UNISIM	ザイリンクス プリミティブの論理シミュレーション。	UNISIM	UNISIMS_VER
UNIMACRO	ザイリンクス マクロの論理シミュレーション。	UNIMACRO	UNIMACRO_VER

表 2: シミュレーション ライブラリ (続き)

ライブラリ名	説明	VHDL ライブラリ名	VHDL ライブラリ名
UNIFAST	高速シミュレーション ライブラリ。	UNIFAST	UNIFAST_VER
SIMPRIM	ザイリンクス プリミティブのタイミング シミュレーション。	なし	SIMPRIMS_VER ¹
SECUREIP	PCIe IP、ギガビット トランシーバーなど、ザイリンクス デバイス機能の論理シミュレーションおよびタイミング シミュレーション用のシミュレーション ライブラリです。 IP、次のディレクトリの SECUREIP の下にリストされます。 <Vivado_Install_Dir>/data/secureip	SECUREIP	SECUREIP
XPM	ザイリンクス プリミティブの論理シミュレーション。	XPM	XPM ²

注記:

- SIMPRIMS_VER は、Verilog SIMPRIM 物理ライブラリがマップされている論理ライブラリ名です。
- XPM はコンパイル済み IP としてサポートされています。プロジェクトにソース ファイルを追加する必要はありません。サードパーティ シミュレータを使用している場合は、Vivado により `compile_simlib` で生成されたコンパイル済み IP にマップされます。



重要: シミュレーション ポイントに従って、それぞれのシミュレーション ライブラリを指定する必要があります。インプリメンテーション前とインプリメンテーション後のネットリストのゲート レベル セルは異なります。

次の表に、各シミュレーション ポイントに必要なシミュレーション ライブラリをリストします。

表 3: シミュレーション ポイントと関連ライブラリ

シミュレーション ポイント	UNISIM	UNIFAST	UNIMACRO	SECUREIP	SIMPRIM (Verilog のみ)	SDF
1. レジスタ トランスファー レベル (RTL) (ビヘイビア)	○	○	○	○	なし	×
2. 合成後シミュレーション (論理)	○	○	なし	○	なし	なし
3. 合成後シミュレーション (タイミング)	なし	なし	なし	○	○	○
4. インプリメンテーション後シミュレーション (論理)	○	○	なし	○	なし	なし
5. インプリメンテーション後シミュレーション (タイミング)	なし	なし	なし	○	○	○



重要: Vivado シミュレータでは、コンパイル済みのシミュレーション デバイス ライブラリが使用されます。アップデートをインストールすると自動的にこれらのライブラリがアップデートされます。

注記: Verilog SIMPRIMS_VER では、UNISIM と同じソースに加え、タイミング アノテーション用に特別なブロックも使用されます。SIMPRIMS_VER は Verilog SIMPRIM がマップされている論理ライブラリ名です。

次の表に、ライブラリのディレクトリを示します。

表 4: シミュレーション ライブラリのディレクトリ

ライブラリ	HDL タイプ	ディレクトリ
UNISIM	Verilog	<Vivado_Install_Dir>/data/verilog/src/unisims
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unisims
UNIFAST	Verilog	<Vivado_Install_Dir>/data/verilog/src/unifast
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unifast
UNIMACRO	Verilog	<Vivado_Install_Dir>/data/verilog/src/unimacro
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unimacro
SECUREIP	Verilog	<Vivado_Install_Dir>/data/secureip/

次のセクションでは、これらのライブラリについて詳細に説明します。

UNISIM ライブラリ

論理シミュレーションでは、UNISIM ライブラリが使用されます。このライブラリには、デバイス プリミティブまたは最下位構築ブロックの記述が含まれます。



重要: デフォルトでは、`compile_simlib` コマンドで IP カタログに含まれるすべての IP のスタティック シミュレーション ファイルがコンパイルされます。

暗号化されたコンポーネント ファイル

次の表に、IP をデザインに含めたときにコンパイル済みの暗号化されたライブラリ ファイルを呼び出す UNISIM ライブラリをリストします。ライブラリ検索パスに必要なパスを含めてください。

表 5: コンポーネント ファイル

コンポーネント ファイル	説明
<Vivado_Install_Dir>/data/verilog/src/unisim-retarget-comp.vp	暗号化された Verilog ファイル
<Vivado_Install_Dir>/data/vhdl/src/unisims/unisim-retarget_VCOMP.vhdp	暗号化された VHDL ファイル



重要: Verilog モジュール名とファイル名は大文字です。たとえば、モジュール BUFG は `BUFG.v`、モジュール IBUF は `IBUF.v` と指定します。UNISIM プリミティブ インスタンス化が大文字の命名規則に従うようにしてください。

VHDL UNISIM ライブラリ

VHDL UNISIM ライブラリは、ザイリンクス デバイス ファミリのプリミティブを指定する次のファイルに分割されています。

- コンポーネント宣言 (`unisim_VCOMP.vhd`)
- パッケージ ファイル (`unisim_VPKG.vhd`)

これらのプリミティブを使用するには、各ファイルの最初に次の 2 行を追加する必要があります。

```
library UNISIM;
use UNISIM.VCOMPONENTS.all;
```



重要: ライブラリをコンパイルし、シミュレータにマップする必要もあります。この方法はシミュレータによって異なります。

注記: Vivado シミュレータでは、ライブラリのコンパイルとマップは統合されているので、ユーザーがコンパイルまたはマップする必要はありません。

Verilog UNISIM ライブラリ

Verilog では、各ライブラリ モジュールは個別の HDL ファイルで指定されます。これにより、`-y` オプションでライブラリを指定することで、指定のディレクトリですべてのコンポーネントを検索してライブラリを自動的に展開できます。

Verilog UNISIM ライブラリは、モジュール使用前に HDL ファイルで指定することはできません。ライブラリ モジュールを使用するには、すべて大文字を使用してモジュール名を指定してください。

次の例は、インスタンス化されたモジュール名とそのモジュールに関連するファイル名を示しています。

- BUFG モジュールは `BUFG.v`
- IBUF モジュールは `IBUF.v`

Verilog では大文字/小文字が区別されるので、UNISIM プリミティブのインスタンス化ではすべて大文字の命名規則に従うようにしてください。

コンパイル済みライブラリを使用する場合は、正しいシミュレータ コマンド ライン オプションを使用して、コンパイル済みライブラリを指定してください。次に、Vivado シミュレータの例を示します。

```
-L unisims_ver
```

説明:

`-L` はライブラリ指定コマンドです。

UNIMACRO ライブラリ

UNIMACRO ライブラリは、論理シミュレーション中に使用されます。このライブラリには、選択したデバイス プリミティブのマクロの記述が含まれます。



重要: 『Vivado Design Suite 7 シリーズ FPGA および Zynq-7000 SoC ライブラリ ガイド』(UG953) にリストされているデバイス マクロを含めるには、UNIMACRO ライブラリを指定する必要があります。

VHDL UNIMACRO ライブラリ

これらのプリミティブを使用するには、各ファイルの最初に次の 2 行を追加する必要があります。

```
library UNIMACRO;  
use UNIMACRO.Vcomponents.all;
```

Verilog UNIMACRO ライブラリ

Verilog では、各ライブラリ モジュールは個別の HDL ファイルで指定されます。これにより、`-y` オプションでライブラリを指定することで、指定のディレクトリですべてのコンポーネントを検索してライブラリを自動的に展開できます。

Verilog UNIMACRO ライブラリは、VHDL のように、モジュール使用前に HDL ファイルで指定する必要はありません。ライブラリ モジュールを使用するには、すべて大文字を使用してモジュール名を指定してください。ライブラリをコンパイルおよびマップする必要もあります。この方法は、選択したシミュレータによって異なります。



重要: Verilog モジュール名とファイル名は大文字です。たとえば、BUFG モジュールの名前は `BUFG.v` です。UNIMACRO プリミティブ インスタンスエーションが大文字の命名規則に従うようにしてください。

SIMPRIM ライブラリ

SIMPRIM ライブラリは、合成またはインプリメンテーション後に生成されるタイミング シミュレーション ネットリストをシミュレーションする際に使用します。



重要: タイミング シミュレーションは Verilog でのみサポートされます。VHDL バージョンの SIMPRIM ライブラリはありません。



ヒント: VHDL を使用する場合、合成後およびインプリメンテーション後の論理シミュレーションを実行できます。この場合、標準のデフォルト フォーマット (SDF) アノテーションは必要なく、シミュレーション ネットリストで UNISIM ライブラリが使用されます。ネットリストは `write_vhdl` Tcl コマンドを使用して作成できます。使用情報は、『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。

次に、Vivado シミュレータ用にライブラリを指定する例を示します。

```
-L SIMPRIMS_VER
```

説明:

- `-L` はライブラリ指定コマンドです。
- `SIMPRIMS_VER` は Verilog SIMPRIM がマップされている論理ライブラリ名です。

SECUREIP シミュレーション ライブラリ

SECUREIP ライブラリは、GT などの複雑なデバイス コンポーネントの論理およびタイミング シミュレーションに使用します。

注記: Secure IP ブロックは Vivado シミュレータで完全にサポートされています。追加の設定は必要ありません。

ザイリンクスでは、IEEE 規格『Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)』(IEEE-STD-P1735) で指定された暗号化手法を使用しています。暗号化は、ライブラリ コンパイル プロセスで自動的に処理されます。

注記: シミュレータでライブラリを指定するコマンド ライン オプションについては、それぞれのシミュレータの資料を参照してください。

次の表に、これらのライブラリを使用する際のシミュレータ ベンダーごとの注意点をリストします。

表 6: SECUREIP ライブラリを使用する際の注意事項

シミュレータ名	ベンダー	要件
ModelSim SE	Mentor Graphics	デザイン入力が VHDL の場合は、混合言語ライセンスまたは SECUREIP OP が必要です。詳細はベンダーにお問い合わせください。
Questa Advanced Simulator		
VCS および VCS MX	Synopsys	

表 6: SECUREIP ライブラリを使用する際の注意事項 (続き)

シミュレータ名	ベンダー	要件
Active-HDL	Aldec	デザイン入力が VHDL のみの場合、言語中立の SECUREIP ライセンスが必要です。詳細はベンダーにお問い合わせください。
Riviera-PRO*		



重要: サードパーティ シミュレータのサポートされるバージョンは、『Vivado Design Suite ユーザー ガイド: リリースノート、インストール、およびライセンス』(UG973) を参照してください。

VHDL SECUREIP ライブラリ

UNISIM ライブラリには、VHDL SECUREIP のラッパーが含まれます。シミュレータがエンティティにバインドできるようにするためには、次の 2 行を各ファイルの最初に追加します。

```
Library UNISIM;
UNISIM.VCOMPONENTS.all;
```

Verilog SECUREIP ライブラリ

Verilog コードを使用してシミュレーションを実行する場合、ほとんどのシミュレータで SECUREIP ライブラリを参照する必要があります。

コンパイル済みライブラリを使用する場合は、正しい指示子を使用して、コンパイル済みライブラリを指定してください。次に、Vivado シミュレータの例を示します。

```
-L SECUREIP
```



重要: Verilog SECUREIP ライブラリを使用するには、コンパイル時に -f オプションを指定します。ファイルリストは、<Vivado_Install_Dir>/data/secureip/secureip_cell.list.f から入手できます。

UNIFAST ライブラリ

UNIFAST ライブラリは RTL ビヘイビアー シミュレーションで使用可能なオプションのライブラリで、シミュレーションの実行時間を短縮します。



重要: UNIFAST ライブラリは論理シミュレーションで使用するオプションのライブラリで、シミュレーションを高速化します。UNIFAST ライブラリは 7 シリーズ デバイスでのみサポートされています。UltraScale 以降のデバイス アーキテクチャでは、すべての最適化が UNISIM ライブラリにデフォルトで組み込まれているので、UNIFAST はサポートされません。UNIFAST ライブラリには、フル モデルで使用可能なチェック/機能がすべて含まれているわけではないので、サインオフ シミュレーションには使用できません。



推奨: デザインの最初の検証には UNIFAST ライブラリを使用し、完全な検証には UNISIM ライブラリを実行してください。

シミュレーション時間の短縮は、シミュレーション モードのプリミティブ機能のサブセットをサポートすることで達成されます。

注記: このシミュレーション モデルでは、サポートされない属性値のみがチェックされます。

MMCM2

シミュレーションの実行時間を短縮するため、高速の MMCM2 シミュレーション モデルではフル モデルと比較して次の点が変更されています。

1. 基本的なクロック生成機能のみが含まれます。DRP、ファイン位相シフト、クロック停止およびクロック カスケードなどのその他の機能はサポートされません。
2. 入力クロックは、周波数および位相が変更されておらず、安定していると想定されます。入力クロック周波数のサンプリングは、LOCKED 信号が High にアサートされると停止します。
3. 出力クロック周波数、位相、デューティ サイクルおよびその他の機能は、入力クロック周波数とパラメーター設定から直接計算されます。

注記: 出力クロック周波数は、input-to-VCO クロックからは生成されません。

4. 標準の MMCM2 シミュレーション モデルと高速の MMCM2 シミュレーション モデルの LOCKED 信号のアサート時間は異なります。
 - 標準モデルの LOCKED アサート時間は、M および D 設定によって異なります。M および D の値が大きいと、標準 MMCM2 シミュレーション モデルのロック時間が比較的長くなります。
 - 高速シミュレーション モデルの LOCKED アサート時間は短くなっています。

DSP48E1

シミュレーション実行時間を短縮するため、高速 DSP48E1 シミュレーション モデルでは次の機能がフル モデルから削除されています。

- パターン検出
- Overflow/Underflow
- DRP インターフェイスのサポート

GTHE2_CHANNEL/GTHE2_COMMON

シミュレーション実行時間を短縮するため、高速 GTHE2 シミュレーション モデルでは次の機能が異なります。

- GTH リンクは、近端と遠端のリンク パートナー間で PPM (Parts Per Million) レートの差なしで同期する必要があります。
- GTH を介したレイテンシのハードウェア操作とのサイクル精度は高くありません。
- DRP プロダクションリセット シーケンスはシミュレーションできません。UNIFAST モデルを使用する場合は、これをバイパスしてください。

Verilog UNIFAST ライブラリの使用

シミュレーション実行時間を短縮するため、高速 GTXE2 シミュレーション モデルでは次の機能が異なります。

- GTX リンクは、近端と遠端のリンク パートナー間で PPM (Parts Per Million) レートの差なしで同期する必要があります。
- GTX を介したレイテンシのハードウェア操作とのサイクル精度は高くありません。

方法 1: フル Verilog UNIFAST ライブラリ (推奨)

方法 1 では、シミュレーションにすべての UNIFAST モデルが使用されます。これが推奨される方法です。

Vivado プロジェクト環境で Vivado シミュレータ、ModelSim、IES、または VCS に対して UNIFAST サポート (高速シミュレーション モデル) をイネーブルにするには、Tcl コンソールで次の Tcl コマンドを実行します。

```
set_property unifast true [current_fileset -simset]
```

コンポーネント ファイルに関する詳細は、[UNISIM ライブラリ](#)を参照してください。

詳細は、サードパーティ シミュレータのユーザー ガイドを参照してください。

方法 2: 特定の UNIFAST モジュールを使用

この方法は上級ユーザー用で、UNIFAST モデルを使用してシミュレーションするモジュールを指定します。

個別のライブラリ コンポーネントを指定するには、Verilog コンフィギュレーション文を使用します。config.v ファイルで次を指定してください。

- 最上位モジュールまたはコンフィギュレーションの名前 (例: config cfg_xilinx;)
- デザイン設定を適用する名前 (例: design test bench;)
- 明示的に呼び出されないセルまたはインスタンスのライブラリ検索順 (例: default liblist unisims_ver unifast_ver;)
- 特定の CELL または INSTANCE を特定のライブラリにマップ (例: instance testbench.inst.O1 use unifast_ver.MMCME2;)

注記: ModelSim (vsim) の場合は、階層名に -genblk を追加します (例: instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.MMCME2; - VSIM)。

config.v の例

```
config cfg_xilinx;
design testbench;
default liblist unisims_ver unifast_ver;
//Use fast MMCM for all MMCM blocks in design
cell MMCME2 use unifast_ver.MMCME2;
//use fast dSO48E1for only this specific instance in the design
instance testbench.inst.O1 use unifast_ver.DSP48E1;
//If using ModelSim or Questa, add in the genblk to the name
(instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.DSP48E1)
endconfig
```

VHDL UNIFAST ライブラリの使用

VHDL UNIFAST ライブラリの基本構造は Verilog と同じであり、アーキテクチャまたはライブラリと共に使用できます。ライブラリはテスト ベンチに含めることができます。

次の例では、for 呼び出しのドリルダウン階層を使用しています。

```
library unisim;
library unifast;
configuration cfg_xilinx of testbench
is for xilinx
.. for inst:netlist
. . . use entity work.netlist(inst);
.....for inst
.....for all:MMCME2
.....use entity unifast.MMCME2;
.....end for;
```

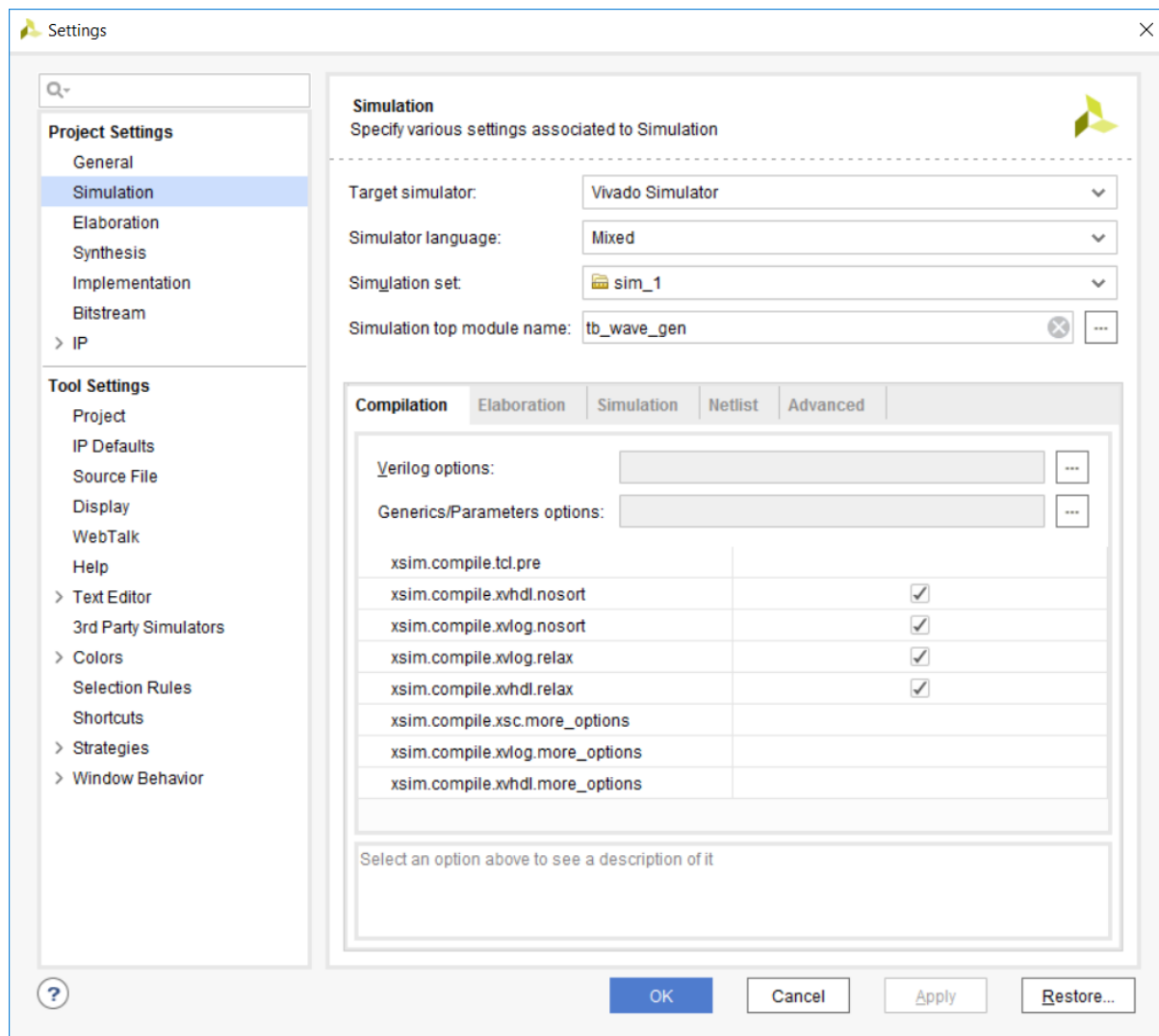
```
.....for O1 inst:DSP48E1;
.....use entity unifast.DSP48E1;
.....end for;
...end for;
..end for;
end for;
end cfg_xilinx;
```

注記: VHDL の UNIFAST モデルを使用する必要がある場合は、エラボレーション中に UNIFAST ライブラリのバインド設定を使用してください。

シミュレーション設定

[Project Settings] ダイアログ ボックスの [Simulation] ページでは、ターゲット シミュレータ、シミュレーション セット、シミュレーション最上位モジュール名、最上位モジュール (テスト中のデザイン) を指定し、コンパイル、エラボレーション、シミュレーション、ネットリスト、アドバンス オプションを設定できます。Vivado IDE の Flow Navigator で [Simulation] を右クリックし、[Simulation Settings] をクリックすると、次の図に示す [Settings] ダイアログ ボックスの [Simulation] ページが開きます。

図 3: [Settings] ダイアログ ボックス



[Settings] ダイアログ ボックスの [Simulation] ページには、次のシミュレーション設定があります。

- [Target simulator]: ドロップダウン リストからシミュレータを選択します。Vivado® シミュレータがデフォルトのシミュレータです。サードパーティ シミュレータも多数サポートされています。
- [Simulator language]: シミュレータの言語モードを選択します。デザインのさまざまな IP に使用されるシミュレーション モデルは、IP でサポートされる言語によって異なります。
- [Simulation set]: シミュレーション コマンドでデフォルトで使用するシミュレーション セットを指定します。



重要: 前に定義されたシミュレーション セットのコンパイルおよびシミュレーション設定は、新しく定義されたシミュレーション セットには適用されません。

- [Simulation top module name]: シミュレーションで使用する別の最上位モジュールを入力します。

- [Compiled library location]: サードパーティ シミュレータを選択した場合に表示され、コンパイルしたライブラリを保存するディレクトリ パスを指定します。デフォルトでは、ライブラリは非プロジェクト モードでは現在の作業ディレクトリに保存され、プロジェクト モードでは `<project>/<project>.cache/compile_simlib` ディレクトリに保存されます。
- [Compilation] タブ: コンパイラ指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして格納され、Verilog および VHDL ソース ファイルをシミュレーション用にコンパイルするために `xvlog` および `xvhdl` ユーティリティで使用されます。

注記: `xvlog` および `xvhdl` は、Vivado シミュレータ特定のコマンドです。指定可能なユーティリティは、ターゲット シミュレータによって異なります。

- [Elaboration] タブ: エラボレーション指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして格納され、`xelab` ユーティリティでシミュレーション スナップショットをエラボレートおよび生成する際に使用されます。表のプロパティを選択すると、そのプロパティの詳細が表示され、値を変更できます。

注記: `xelab` は、Vivado シミュレータ特定のコマンドです。指定可能なユーティリティは、ターゲット シミュレータによって異なります。

- [Simulation] タブ: シミュレーション指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして格納され、`xsim` アプリケーションで現在のプロジェクトをシミュレーションする際に使用されます。表のプロパティを選択すると、そのプロパティの詳細が表示され、値を変更できます。
- [Netlist] タブ: Verilog ネットリストの SDF アノテーションおよび SDF 遅延で取り込まれるプロセス コーナーに関するネットリスト コンフィギュレーション オプションにアクセスできます。これらのオプションはシミュレーション ファイルセットのプロパティとして保存され、シミュレーション用ネットリストを生成する際に使用されます。
- [Advanced] タブ: 次の 2 つのオプションが含まれます。
 - [Enable incremental compilation]: インクリメンタル コンパイルをイネーブルにし、次の run でシミュレーション ファイルを保持します。
 - [Include all design sources for simulation]: デフォルトでオンになっています。このオプションをオンにすると、デザイン ソースからのファイルと現在のシミュレーション セットからのファイルすべてがシミュレーションで使用されます。デザイン ソースを変更した場合でも、ビヘイビア シミュレーションを実行するときにその変更が反映されます。



注意: [Advanced] タブの設定は、必要な場合にのみ変更してください。[Include all design sources for simulation] チェックボックスはデフォルトでオンになっています。オフにすると、予期しない結果になる可能性があります。チェックボックスがオンになっていれば、シミュレーション セットにアウト オブ コンテキスト (OOC) の IP、IP インテグレーター ファイル、および DCP が含まれます。

注記: [Compilation]、[Elaboration]、[Simulation]、[Netlist]、[Advanced] タブのプロパティの詳細は、[付録 A: コンパイル、エラボレーション、シミュレーション、ネットリスト、アドバンス オプション](#)を参照してください。

シミュレータ言語オプションの理解

ほとんどのザイリンクス IP には 1 つの言語に対するビヘイビア シミュレーション モデルしか提供されていないので、該当する言語のライセンスがない場合、その言語にロックされたシミュレータでのシミュレーションはディスエーブルになります。`simulator_language` プロパティを使用すると、IP で指定した言語のシミュレーション モデルが提供されます。シミュレーション言語は、[Settings] ダイアログ ボックスで設定できます。たとえば、1 つの言語シミュレータを使用する場合は、`simulator_language` プロパティをシミュレータの言語と一致するように設定します。

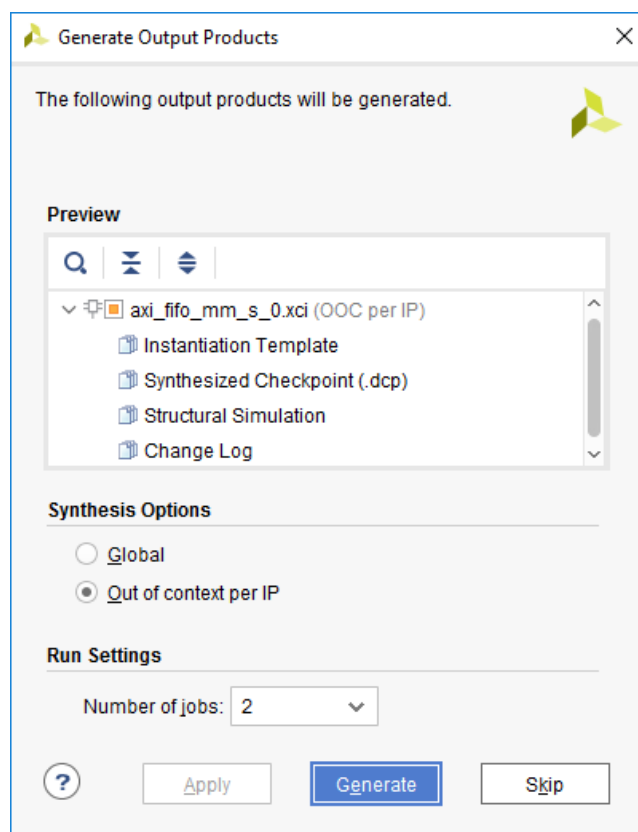
Vivado Design Suite では、IP の合成ファイルを使用して言語専用の構造型シミュレーション モデルをオンデマンドで生成することにより、シミュレーション モデルを提供します。ビヘイビア モデルがない場合や、言語がライセンス付与されたシミュレーション言語と異なる場合は、Vivado ツールで構造型シミュレーション モデルが自動的に生成され、シミュレーションが可能になります。これ以外の場合は、IP には既存のビヘイビア シミュレーション モデルが使用されます。合成またはシミュレーション ファイルが存在しない場合、シミュレーションはサポートされません。

注記: [Generate Synthesized Checkpoint (.dcp)] がオフになっていると、`simulator_language` プロパティで言語専用のシミュレーション ネットリスト ファイルを提供できません。

1. Flow Navigator で [IP Catalog] をクリックし、IP カタログを開きます。
2. 該当する IP を右クリックし、[Customize IP] をクリックします。
3. [Customize IP] ダイアログ ボックスで [OK] をクリックします。

次の図に示す [Generate Output Products] ダイアログ ボックスが開きます。

図 4: [Generate Output Products] ダイアログ ボックス



次の表に、`simulator_language` プロパティの機能を説明します。

表 7: simulator_language プロパティの機能

IP に含まれるシミュレーション モデル	simulator_language の値	使用されるシミュレーション モデル
IP で VHDL および Verilog のビヘイビア モデルを提供	Mixed	ビヘイビア モデル (target_language)
	Verilog	Verilog ビヘイビア モデル
	VHDL	VHDL ビヘイビア モデル
IP で Verilog ビヘイビア モデルのみを提供	Mixed	Verilog ビヘイビア モデル
	Verilog	Verilog ビヘイビア モデル
	VHDL	DCP から生成された VHDL シミュレーション ネットリスト
IP で VHDL ビヘイビア モデルのみを提供	Mixed	VHDL ビヘイビア モデル
	Verilog	DCP から生成された Verilog シミュレーション ネットリスト
	VHDL	VHDL ビヘイビア モデル
IP でビヘイビア モデルが提供されない	Mixed/Verilog/VHDL	DCP から生成されたネットリスト (target_language)

注記:

1. 可能であれば、ビヘイビア シミュレーション モデルが構造型シミュレーション モデルよりも優先されます。Vivado ツールでは、使用可能なモデルに基づいて、ビヘイビアまたは構造型モデルが自動的に選択されます。この自動選択は変更できません。
2. どちらの言語もシミュレーションに使用できる場合は target_language プロパティを使用し、set_property target_language VHDL [current-project] のように指定します。

シミュレーション ランタイム精度の設定

テストベンチで 'timescale を使用してシミュレーション ランタイム精度を設定します。ザイリンクス シミュレーション モデルでは、これより粗い精度を使用してもシミュレータ パフォーマンスが良くなることはありません。ザイリンクス シミュレーション モデルでは、シミュレーション時間のほとんどがデルタ サイクルで費やされますが、デルタ サイクルはシミュレーション精度の影響を受けません。



重要: 時間精度 1 fs を使用してシミュレーションを実行します。GT などのザイリンクスのプリミティブ コンポーネントの中には、論理またはタイミング シミュレーションのいずれかで正しく動作するためには 1 fs の精度が必要なものがあります。

[Settings] ダイアログ ボックスのシミュレーション オプションの詳細は、[シミュレーション オプション](#)を参照してください。






重要: テスト ツールはタイミングをすべて一番近いピコ秒 (ps) に対してのみ計測するので、最小精度としてピコ秒が使用されます。

シミュレーション ソース ファイルの追加または作成

シミュレーション ソースを Vivado Design Suite プロジェクトに追加するには、次の手順に従います。

1. [File]→[Add Sources] をクリックするか、Flow Navigator で [Add Sources] をクリックします。
Add Sources ウィザードが表示されます。
2. [Add or Create Simulation Sources] をオンにし、[Next] をクリックします。

[Add or Create Simulation Sources] ページが開きます。次のオプションがあります。

- [Add Files]: プロジェクトに追加するシミュレーション ソース ファイルを選択します。
- [Add Directories]: 選択したディレクトリに含まれるすべてのシミュレーション ソース ファイルを追加します。指定したディレクトリにある有効なソース ファイルがすべてプロジェクトに追加されます。
- [Create File]: 新規シミュレーション ファイルを作成する [Create Source File] ダイアログ ボックスが開きます。プロジェクト ソース ファイルの詳細は、『Vivado Design Suite ユーザー ガイド: システム レベル デザイン入力』 (UG895) の[このセクション](#)を参照してください。
- ウィザードには、次のボタンがあります。
 - [Remove] : 追加するファイルのリストから選択したソース ファイルを削除します。
 - [Move Up] : ファイルをリストの上方向に移動します。
 - [Move Down] : ファイルをリストの下方向に移動します。
- ウィザードには、次のチェック ボックスがあります。
 - [Scan and add RTL include files into project]: 追加された RTL ファイルをスキャンし、参照されているインクルード ファイルを追加します。
 - [Copy sources into project]: ソース ファイルをプロジェクト ディレクトリにコピーします。プロジェクトではローカルにコピーされたバージョンが使用されます。

[Add Directories] ボタンをクリックしてソース ファイルのディレクトリを追加した場合は、ファイルがローカルプロジェクトにコピーされる際にディレクトリ構造もそのまま保持されます。

 - [Add sources from subdirectories]: 指定したディレクトリのサブディレクトリに含まれるソース ファイルをすべて追加します。
 - [Include all design sources for simulation]: シミュレーションにデザイン ソースをすべて含めます。



ビデオ: この機能の詳細は、[Vivado Design Suite QuickTake ビデオ: ロジック シミュレーション](#)をご視聴ください。

シミュレーション セットの操作

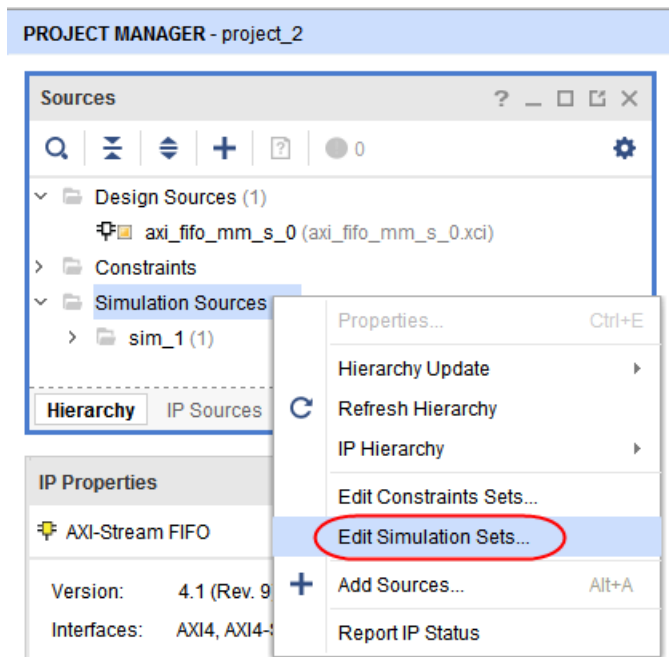
Vivado IDE では、シミュレーション ソース ファイルは [Sources] ウィンドウにフォルダーとして表示されるシミュレーション セットに保存されます。リモートのファイルを参照するか、ローカル プロジェクト ディレクトリに保存されているファイルを使用できます。

シミュレーション セットにより、デザインの異なる段階に異なるソースを使用できます。たとえば、エラボレート済みデザインまたはデザインのモジュールのビヘイビア シミュレーション用にスティミュラスを供給するテストベンチソースと、インプリメント済みデザインのタイミング シミュレーション用にスティミュラスを供給するテストベンチを別にできます。

シミュレーション ソースをプロジェクトに追加する際、使用するシミュレーション ソース セットを指定できます。

シミュレーション セットを変更するには、次を実行します。

1. [Sources] ウィンドウで [Simulation Sources] を右クリックし、[Edit Simulation Sets] をクリックします。



[Add or Create Simulation Sources] ページが表示されます。

2. [Add Sources] をクリックしてファイルを選択します。
これで、プロジェクトに関するソースが新しく作成されたシミュレーション セットに追加されます。
3. 必要に応じてほかのファイルも追加します。
選択したシミュレーション セットがアクティブなデザイン run に使用されます。

ネットリストの生成

合成済みまたはインプリメント済みデザインのシミュレーションを実行するには、ネットリスト生成プロセスを実行する必要があります。Tcl のネットリスト生成コマンドは合成済みまたはインプリメント済みデザイン データベースを使用し、デザイン全体に対して 1 つのネットリストを生成します。

Vivado Design Suite では、IDE または `launch_simulation` コマンドを使用してシミュレータを実行すると、ネットリストが自動的に生成されます。

ネットリスト生成コマンドでは、SDF およびデザイン ネットリストを生成できます。Vivado Design Suite では、次の Tcl コマンドが提供されています。

- `write_verilog`: Verilog ネットリスト
- `write_vhdl`: VHDL ネットリスト
- `write_sdf`: SDF 生成



ヒント: SDF 値は、デザイン プロセスの早期 (合成中など) では見積りにすぎません。デザイン プロセスが進むと、データベースにより多くの情報が含まれるようになり、タイミング値の精度も上がります。

論理ネットリストの生成

Vivado Design Suite では、論理シミュレーション用に Verilog または VHDL 構造ネットリストを生成できます。このネットリストは、シミュレーションをタイミング情報なしで実行し、構造ネットリストのビヘイビアが予測されるビヘイビア モデル (RTL) シミュレーションと一致するかどうかをチェックするのに使用されます。

論理シミュレーション ネットリストは階層構造で、モジュールまたはエンティティ レベル (プリミティブおよびマクロ プリミティブを含む下位階層) に展開可能なネットリストです。

これらのプリミティブは、次のライブラリに含まれます。

- Verilog シミュレーションの場合は UNISIMS_VER シミュレーション ライブラリ
- VHDL シミュレーションの場合は UNISIM シミュレーション ライブラリ

多くの場合、ビヘイビア シミュレーションで使ったのと同じテストベンチを使用して、さらに精度の高いシミュレーションを実行できます。

Verilog および VHDL の論理シミュレーション ネットリストを生成するには、次の Tcl コマンドを使用します。

```
write_verilog -mode funcsim <Verilog_Netlist_Name.v>
write_vhdl -mode funcsim <VHDL_Netlist_Name.vhd>
```

タイミング ネットリストの生成

Verilog タイミング シミュレーションを使用すると、Vivado ツールでのワースト ケースの配置配線遅延の計算後に回路動作を検証できます。

多くの場合、論理シミュレーションで使ったのと同じテストベンチを使用して、さらに精度の高いシミュレーションを実行できます。

2 つのシミュレーションからの結果を比較し、デザインが最初に指定したとおりに実行されているかどうかを検証します。

タイミング シミュレーション ネットリストを生成する手順は、次のとおりです。

1. デザインのシミュレーション ネットリスト ファイルを生成します。
2. タイミング遅延すべてをアノテートした SDF 遅延ファイルを生成します。



重要: Vivado IDE では、Verilog タイミング シミュレーションのみがサポートされます。



ヒント: VHDL を使用する場合、合成後およびインプリメンテーション後の論理シミュレーションを実行できます。この場合、標準のデフォルト フォーマット (SDF) アノテーションは必要なく、シミュレーション ネットリストで UNISIM ライブラリが使用されます。ネットリストは [write_vhdl](#) Tcl コマンドを使用して作成できます。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。

タイミング シミュレーション ネットリストを生成するには、次の Tcl 構文を使用します。

```
write_verilog -mode timesim -sdf_anno true <Verilog_Netlist_Name>
```

サードパーティ シミュレータを使用したシミュレーション

Vivado® Design Suite では、サードパーティ ツールを使用したシミュレーションがサポートされています。サードパーティ ツールを使用したシミュレーションは、Vivado 統合設計環境 (IDE) 内から直接実行するか、またはカスタムの外部シミュレーション環境を使用して実行できます。

表 8: サポートされるサードパーティ シミュレータ

サードパーティ シミュレータ	Red Hat Linux	Red Hat 64 ビット Linux	SUSE Linux	Windows 10 64 ビット
Mentor Graphics 社 ModelSim SE	なし	○	なし	○
Mentor Graphics 社 Questa Advanced Simulator	なし	○	なし	○
Cadence 社 Incisive Enterprise シミュレータ	なし	○	なし	なし
Cadence 社 Xcelium Parallel Simulator	なし	○	なし	なし
Synopsys VCS	なし	○	なし	なし
Aldec Active HDL	なし	なし	なし	○
Aldec Riviera PRO	なし	○	なし	○

Vivado IDE の使用方法は、『Vivado Design Suite ユーザー ガイド: Vivado IDE の使用』(UG893) を参照してください。

Vivado IDE でシミュレーションを実行する前に、次の環境変数を設定してください。

表 9: サードパーティ シミュレータの環境変数設定

シミュレータ	Linux	Windows
Modelsim	<pre>setenv MODEL_TECH <tool installation path> setenv LM_LICENSE_FILE <license file> setenv PATH \${MODEL_TECH}/bin:\$PATH</pre>	<pre>set MODEL_TECH=<tool installation path> set LM_LICENSE_FILE=<license file> set Path= %MODEL_TECH% \win32;%Path%</pre>

表 9: サードパーティ シミュレータの環境変数設定 (続き)

シミュレータ	Linux	Windows
Questa	<pre>setenv MODEL_TECH <tool installation path> setenv LM_LICENSE_FILE <license file> setenv PATH \${MODEL_TECH}/bin:\$PATH</pre>	<pre>set MODEL_TECH=<tool installation path> set LM_LICENSE_FILE=<license file> set Path= %MODEL_TECH% \win32;%Path%</pre>
Riviera	<pre>setenv ALDEC_PATH <tool installation path> setenv VSIMSACFG <tool installation path> setenv ALDEC_GCCDIR gcc setenv PATH \${ALDEC_PATH}:\${ALDEC_PATH}/bin/Linux64:\${ALDEC_PATH}/\${ALDEC_GCCDIR}/bin:\$PATH setenv LD_LIBRARY_PATH \${ALDEC_PATH}/bin/Linux64:\$LD_LIBRARY_PATH set LM_LICENSE_FILE <license file></pre>	<pre>set RIVIERA_BIN=<tool installation path> set Path= %<Riviera install dir>% \bin;%Path% set LM_LICENSE_FILE=<license file></pre>
Active-HDL	なし	<pre>set ACTIVE_BIN=<tool installation path> set Path= %<Active-hdl install dir>% \BIN;%Path% set LM_LICENSE_FILE=<license file></pre>
Xcelium	<pre>setenv GCC_HOME <path to gcc installation> setenv CDS_INST_DIR <path to installation dir> setenv LD_LIBRARY_PATH \$* {CDS_INST_DIR}/tools/xcelium/ lib:\$LD_LIBRARY_PATH* setenv PATH \${GCC_HOME}/bin:\${CDS_INST_DIR}/bin:\${CDS_INST_DIR}/tools/ xcelium/bin:\$PATH set LM_LICENSE_FILE <license file> NA IUS setenv CDS_INST_DIR <path to installation dir> setenv LD_LIBRARY_PATH \${CDS_INST_DIR} /tools/lib:\$LD_LIBRARY_PATH* setenv PATH \${CDS_INST_DIR}/tools/bin:\$PATH setenv PATH \${CDS_INST_DIR}/tools/verilog/bin:\${CDS_INST_DIR}/bin:\$PATH set LM_LICENSE_FILE <license file></pre>	なし

表 9: サードパーティ シミュレータの環境変数設定 (続き)

シミュレータ	Linux	Windows
VCS	<pre>setenv VCS_HOME <path to installation dir> setenv GCC_HOME <path to installation dir>/amd64/gcc-6.2.0 setenv PATH \$* {GCC_HOME} /bin:\${VCS_HOME}/bin:\$PATH* setenv LD_LIBRARY_PATH /usr/lib64:\$LD_LIBRARY_PATH set LM_LICENSE_FILE <license file></pre>	なし

注記:

- OS にかかわらず、環境変数 `PATH` にツールのインストール パスを追加する必要があります。サポートされているシミュレータで SystemC ベースのデザインをシミュレーションする場合は、`GCC_HOME` に必要な g++ バージョンのインストール パスを指定してください。`LD_LIBRARY_PATH` にシミュレータのライブラリ パスを含める必要もあります。

サードパーティ シミュレータの詳細は、[サードパーティ シミュレータに関する情報へのリンク](#) を参照してください。



重要: サポートされているバージョンのサードパーティ シミュレータのみを使用してください。サポートされるシミュレータおよび OS の詳細は、『Vivado Design Suite ユーザー ガイド: リリース ノート、インストール、およびライセンス』(UG973) の「互換性のあるサードパーティ ツール」の表を参照してください。

Vivado IDE でのサードパーティ シミュレータを使用したシミュレーションの実行



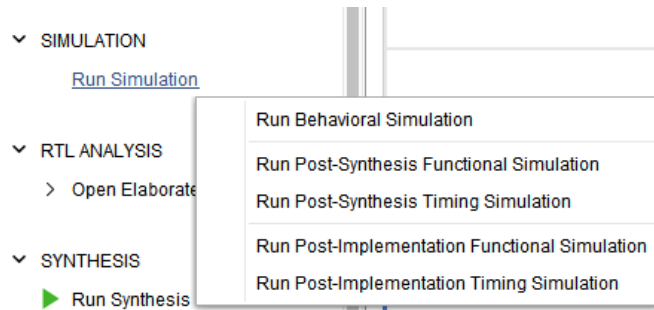
重要: サードパーティ シミュレータを実行する前に、コンパイルされたライブラリのディレクトリ (`compile_simlib` が実行されたパス、または `-directory` オプションで指定されたディレクトリ) を確認してください。

Vivado IDE では、シミュレーション設定に基づいてデザインをコンパイル、エラボレート、シミュレーションできます。シミュレータは別ウィンドウで起動します。

デザインの合成前にシミュレーションを実行すると、ビヘイビアー シミュレーションが実行されます。各デザイン手順 (合成およびインプリメンテーション) が終了したら、論理またはタイミングシミュレーションを実行するオプションが選択できるようになります。シミュレーション run は、Flow Navigator または Tcl コマンドから開始できます。

Flow Navigator で [Run Simulation] をクリックし、次の図に示すリストから実行するシミュレーションをクリックします。

図 5: シミュレーションの種類



対応する Tcl コマンドは `launch_simulation` です。



ヒント: このコマンドには、DO ファイルまたは SH ファイルを出力するための `-scripts_only` オプションがあります。どちらのファイルが出力されるかは、使用しているシミュレータによります。IDE 環境外でシミュレーションを実行するには、DO または SH ファイルを使用してください。

注記: Vivado 外で VCS シミュレータを実行する場合は、`-full64` オプションを使用してください。そうしないと、デザインにザイリンクス IP が含まれている場合にシミュレータが実行されません。



重要: 32 ビットのシミュレータを実行するには、`set_property 32bit 1 [current_fileset -simset]` コマンドを使用します。

注記: ザイリンクス Verification IP (VIP) では、SystemVerilog コンストラクトが使用されます。VIP をインスタンス化して使用する場合は、シミュレータで SystemVerilog がサポートされている必要があります。

サードパーティ ツールを使用したタイミング シミュレーションの実行



ヒント: 合成後のタイミング シミュレーションでは、合成されたネットリストから見積もられたタイミング遅延が使用されます。インプリメンテーション後のタイミング シミュレーションでは実際のタイミング遅延が使用されます。

合成後およびインプリメンテーション後のタイミング シミュレーションを実行する際、シミュレータには次のものが含まれます。

- SIMPRIM ライブラリ コンポーネントを含むゲートレベルのネットリスト
- SECUREIP
- 標準遅延フォーマット (SDF) ファイル

最初にデザイン全体の機能を定義します。デザインがインプリメントされると、正確なタイミング情報が利用できるようになります。

ネットリストおよび SDF を作成するため、Vivado Design Suite は次の処理を実行します。

- `write_verilog` オプションを使用してネットリスト ライターの `-mode timesim`、および `write_sdf` (SDF アノテーター) を呼び出します。
- 生成されたネットリストをターゲット シミュレータに送ります。

これらのオプションを変更するには、[シミュレーション設定](#)で説明されているように、[Settings] ダイアログ ボックスの [Simulation] ページを使用します。



重要: 合成後とインプリメンテーション後のタイミング シミュレーションは、Verilog でのみサポートされます。VHDL のタイミング シミュレーションはサポートされません。VHDL を使用する場合、合成後およびインプリメンテーション後の論理シミュレーションを実行できます。この場合、SDF アノテーションは必要なく、シミュレーション ネットリストで UNISIM ライブラリが使用されます。ネットリストは `write_vhdl Tcl` コマンドを使用して作成できます。使用情報は、『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。

合成後のタイミング シミュレーション

合成を実行すると、[Run Simulation] → [Post-Synthesis Timing Simulation] を実行できるようになります。

合成後のタイミング シミュレーションを選択すると、タイミング ネットリストと SDF ファイルが生成されます。ネットリスト ファイルには `$sdf_annotate` コマンドが含まれるので、生成された SDF ファイルが自動的に指定されます。

インプリメンテーション後のタイミング シミュレーション

インプリメンテーションを実行すると、[Run Simulation] → [Post-Implementation Timing Simulation] を実行できるようになります。

インプリメンテーション後のタイミング シミュレーションを選択すると、タイミング ネットリストと SDF ファイルが生成されます。ネットリスト ファイルには `$sdf_annotate` コマンドが含まれるので、生成された SDF ファイルが自動的に指定されます。

タイミング シミュレーション用の SDF ファイルのアノテート

シミュレーション設定を指定したときに、SDF ファイルを作成するかどうかと、プロセス コーナーをファーストまたはスローに設定しました。



ヒント: SDF ファイルのオプションの設定を確認するには、Vivado IDE の Flow Navigator で [Simulation] を右クリックし、[Simulation Settings] をクリックします。[Settings] ダイアログ ボックスの [Simulation] ページで [Netlist] タブをクリックします。

SDF ファイルには、指定したプロセス コーナーに基づいて異なる `min` および `max` 値が含まれます。



推奨: セットアップ チェックを実行するには、`[-process_corner]` を `[slow]` に設定して SDF を作成し、SDF ファイルの `max` 列を使用します。

ホールド チェックを実行するには、`[-process_corner]` を `[fast]` に設定して SDF ファイルを作成し、SDF ファイルの `min` 列を使用します。使用する SDF 遅延フィールドの指定方法は、使用するシミュレーション ツールによって異なります。このオプションの設定方法は、ご使用のシミュレーション ツールの資料を参照してください。

4 つのタイミング シミュレーションすべてを実行するには、次のように指定します。

1. スロー コーナー: SDFMIN および SDFMAX
2. ファースト コーナー: SDFMIN および SDFMAX

スタンドアロンのタイミング シミュレーションの実行

Vivado IDE からタイミング シミュレーションを実行する場合は、タイミング シミュレーション関連のオプションがシミュレータに追加されますが、スタンドアロン タイミング シミュレーションを実行する場合は、エラボーレーション中に次のオプションをシミュレータに渡す必要があります。

IUS の場合:

```
-PULSE_R/0 -PULSE_E/0
```

エラボレーション中 (ncelab を使用)

VCS の場合:

```
+pulse_e/<number> and +pulse_r/<number> +transport_int_delays
```

エラボレーション中 (VCS を使用)

ModelSim/Questa Advanced Simulator の場合:

```
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0
```

エラボレーション中 (vsim を使用)



重要: Vivado シミュレータ モデルではインターコネクト遅延が使用されるので、タイミング シミュレーションが正しく実行されるようにするには、`-transport_int_delays -pulse_r 0 -pulse_int_r 0` オプションを追加する必要があります。これらのコマンドについては、[表 16: xelab、xvhdl、xvlog のコマンド オプション](#) で説明しています。

消費電力解析用の SAIF の出力

SAIF (Switching Activity Interchange Format) は、シミュレータ ツールで生成されたスイッチング アクティビティを抽出して保存する ASCII 形式のレポートです。このスイッチング アクティビティは、サイリンクスの消費電力解析および最適化ツールにバックアノテートして消費電力の測定および見積りに使用できます。

Questa Advanced Simulator/ModelSim での SAIF の出力

Questa Advanced Simulator/ModelSim で SAIF ファイルを出力するには、次の手順に従います。

1. 次のように入力し、出力する範囲または信号を指定します。

```
power add <hdl_objects>
```

2. シミュレーションを指定時間または `run -all` を使用して実行します。
3. 次のように入力し、消費電力レポートを出力します。

```
power report -all filename.saif
```

各コマンドの使用方法および詳細は、ModelSim の資料を参照してください。

DO ファイルの例

```
power add tb/fpga/*
run 500us
power report -all -bsaif routed.saif
quit
```

IES での SAIF の出力

IES で SAIF ファイルを出力するには、次の手順に従います。

1. 次のように入力し、出力する範囲と出力 SAIF ファイル名を指定します。

```
dumpsaif -scope hdl_objects -output filename.saif
```

2. シミュレーションを実行します。
3. 次のように入力して SAIF の出力を終了します。

```
dumpsaif -end
```

IES コマンドの使用方法および詳細は、[Cadence 社 IES の資料](#)を参照してください。

VCS での SAIF の出力

VCS で SAIF ファイルを出力するには、次の手順に従います。

1. 次のように入力し、出力する範囲および信号を指定します。

```
power <hdl_objects>
```

2. SAIF の出力をイネーブルにします。シミュレータ ワークスペースでコマンド ラインを使用できます。

```
power -enable
```

3. シミュレーションを指定した時間実行します。
4. 次のように入力し、消費電力の出力をディスエーブルにし、SAIF をレポートします。

```
power -disable  
power -report filename.saif
```

各コマンドの使用方法および詳細は、Synopsys 社 VCS の資料を参照してください。

SAIF (Switching Activity Interchange Format) の詳細は、[Vivado シミュレータを使用した消費電力解析](#)を参照してください。

VCD の出力

シミュレーション出力を記録するには、VCD (Value Change Dump) ファイルを使用できます。Tcl コマンドは、出力される値に関連する Verilog システム タスクに基づきます。

Questa Advanced Simulator/ModelSim での VCD の出力

Questa Advanced Simulator/ModelSim で VCD ファイルを出力するには、次の手順に従います。

1. VCD ファイルを開きます。

```
vcd file my_vcdfile.vcd
```


- 出力する範囲または信号を指定します。

```
vcd add <hdl_objects>
```

- シミュレーションを指定時間または `run -all` を使用して実行します。

各コマンドの使用方法および詳細は、[ModelSim の資料](#)を参照してください。

DO ファイルの例:

```
vcd file my_vcdfile.vcd
vcd add -r tb/fpga/*
run 500us
quit
```

IES での VCD の出力

- 次のコマンドを実行して、`vcddb` という VCD データベースを開きます。ファイル名は `verilog.dump` です。-`timescale` オプションは、VCD ファイルの `$timescale` 値を `1 ns` に設定します。VCD ファイルの値が `1 ns` にスケーリングされます。

```
database -open -vcd vcddb -into verilog.dump -default -timescale ns
```

- 次の `probe` コマンドを使用して、スコープ `top.counter` のすべてのポートにプローブを作成します。データはデフォルトの VCD データベースに送信されます。

```
probe -create -vcd top.counter -ports
```

- シミュレーションを実行します。

VCS での VCD の出力

VCS で VCD ファイルを生成するには、`dumpvar` コマンドを使用します。ファイル名およびインスタンス名を指定します (デフォルトでは階層全体になります)。

```
vcs +vcs+dumpvars+test.vcd
```

IP のシミュレーション

次の例では、`accum_0.xci` ファイルは Vivado® IP カタログから生成された IP です。次のコマンドを使用して、この IP を VCS でシミュレーションします。

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>[current_project]
launch_simulation -noclean_dir -of_objects [get_files accum_0.xci]
```

統合されたシミュレーションの実行中のカスタム DO ファイルの使用

シミュレーションの実行前に起動しておく 特定のコマンド セット (カスタム DO ファイル) がある場合は、これらのコマンドをファイルに追加して、次に示すコマンドを使用して渡します。

Questa Advanced Simulator

```
expanse="page">set_property -name {questa.simulate.tcl.post} -value  
{<AbsolutePathOfFileLocation>}  
-objects [get_filesets sim_1]
```

ModelSim

```
expanse="page">set_property -name {modelsim.simulate.tcl.post} -value  
{<AbsolutePathOfFileLocation>}  
-objects [get_filesets sim_1]
```

IES

```
expanse="page">set_property -name {ies.simulate.tcl.post} -value  
{<AbsolutePathOfFileLocation>} -objects  
[get_filesets sim_1]
```

VCS

```
expanse="page">set_property -name {vcs.simulate.tcl.post} -value  
{<AbsolutePathOfFileLocation>} -objects  
[get_filesets sim_1]
```

Xcelium

```
expanse="page">set_property -name {xcelium.simulate.tcl.post} -value  
{<AbsolutePathOfFileLocation>}  
-objects
```

```
expanse="page">[get_filesets sim_1]
```

ModelSim および Questa Advanced Simulator でのシミュレーション ステップ制御コンストラクト

次の表に、do ファイル フォーマットに基づくステップ実行の制御に使用されるコンストラクトを示します。

- ネイティブ do ファイル: デフォルトの do ファイル フォーマットです。このフォーマットでは、コンパイルおよびエラボレート シェル スクリプトは `source <tb>_compile/elaborate.do` を呼び出します。次に例を示します。

```
source bft_tb_compile.do 2>&1 | tee -a compile.log
```

シミュレーション スクリプトは `"vsim -64 -c -do "do {<tb>_simulate.do}"` を呼び出します。次に例を示します。

```
$bin_path/vsim -64 -c -do "do {bft_tb_simulate.do}" -l simulate.log
```

- クラシック do ファイル: コンパイルおよびエラボレート シェル スクリプトでは、クラシック do ファイル フォーマットはネイティブ do ファイルと異なります。シミュレーション スクリプトでは変更はありません。このフォーマットでは、コンパイルおよびエラボレート シェル スクリプトは `"vsim -c -do "do {<tb>_compile/elaborate.do}"` を呼び出します。次に例を示します。

```
$bin_path/vsim -64 -c -do "do {bft_tb_compile.do}" -l compile.log
```

これには、Tcl コンソールで `'set_param project.writeNativeScriptForUnifiedSimulation 0'` コマンドを実行して `'project.writeNativeScriptForUnifiedSimulation'` を 0 に設定する必要があります。

シェル スクリプトでは Questa Advanced Simulator/ModelSim ユーティリティがハードコード化されているので、このファイル フォーマットは共有プロジェクトに便利です。

表 10: シミュレーション ステップ制御コンストラクトのパラメーター

パラメーター	説明	デフォルト
<code>project.writeNativeScriptForUnifiedSimulation</code>	シミュレータ コマンドのみを含む (Tcl またはシェル コンストラクトなし) 純粋な .do ファイルを記述します。	0 (false)
<code>simulator.quitOnSimulationComplete</code>	ModelSim/Questa Advanced Simulator のシミュレーションで、シミュレータ完了時にシミュレータを終了します。終了をディセーブルにするには、このパラメーターを <code>false</code> に設定します。	1 (true)
<code>simulator.modelsimNoQuitOnError</code>	ModelSim/Questa Advanced Simulator のシミュレーションでは、エラーまたはブレークが発生したときにデフォルトではシミュレータは終了しません。エラーまたはブレークが発生したときにシミュレータを終了するには、このパラメーターを <code>false</code> に設定します。	1 (true)
<code>project.enable2StepFlowForModelSim</code>	ユニファイドシミュレーション用に、ModelSim-PE/DE/SE エディションの 2 段階シミュレーションフローを実行します。	1 (true)

説明

- `simulator.quitOnSimulationComplete`: デフォルトでは、生成された `simulate.do` に `quit -force` が含まれます。シミュレーションが指定した時間実行されると、シミュレータは終了します。シミュレータが終了しないようにするには、`set_peram simulator.quitOnSimulationComplete 0` を実行して `simulator.quitOnSimulationComplete` を 0 に設定します。
- `simulator.modelsimNoQuitOnError`: デフォルトでは、エラーまたはブレークが発生してもシミュレータは終了しません。シミュレータが終了するようにするには、次のパラメーターを設定します。

```
set_param simulator.modelsimNoQuitOnError 0
```

これにより、<tb>_simulate.do に次の2行が追加されます。

```
onbreak {quit -f}  
onerror {quit -f}
```

- `project.enable2StepFlowForModelSim`: デフォルトでは、3段階フローのスク립トが生成されます。2段階フローを生成するには、このパラメーターを0に設定します。このパラメーターは、ModelSim PE/DE/SEでのみ使用可能です。

バッチ モードでのサードパーティ シミュレータの実行

Vivado Design Suite では、サードパーティ 検証用にバッチまたはスク립ト モードのシミュレーションがサポートされます。すべてのデザイン ファイルを準備し、ターゲット シミュレータをサポートするスク립トを生成したら、スク립トを確認して、それらを検証環境に組み込むことができます。カスタム API を構築するよりも、`export_simulation` スクリプトをシミュレーション フローの開始点として使用して、スク립トを生成することをお勧めします。シミュレーション スクリプトのエクスポートの詳細は、[シミュレーション ファイルとスク립トのエクスポート](#)を参照してください。

スク립トを実行する前に環境がそのシミュレータ用に設定されているかどうかを確認してください。シミュレータの設定の詳細は、[シミュレーション設定](#)を参照してください。バッチ モードまたはスク립ト モードの実行方法の詳細は、そのシミュレータのユーザー ガイドを参照してください。

Vivado シミュレータを使用したシミュレーション

Vivado シミュレータは、HDL イベント ドリブン シミュレータで、VHDL、Verilog、SystemVerilog (SV)、VHDL/Verilog または VHDL/SV 混合言語のデザインの論理およびタイミング シミュレーションをサポートします。

Vivado シミュレータでは、次の機能がサポートされます。

- ソース コード デバッグ (ステップ、ブレークポイント、現在値の表示)
- タイミング シミュレーション用の SDF アノテーション
- VCD 出力
- SAIF 出力 (電力解析および最適化用)
- HardIP ブロック (シリアル トランシーバーおよび PCIe[®] など) のネイティブ サポート
- マルチスレッド コンパイル
- 混合言語 (VHDL、Verilog、または SystemVerilog デザイン コンストラクト)
- 1 クリックでシミュレーションを再コンパイルおよび再起動
- 1 クリックでコンパイルおよびシミュレーション
- ザイリンクス シミュレーション ライブラリのビルトイン サポート
- リアルタイムの波形アップデート

Vivado シミュレーションを実行する詳細な手順は、『Vivado Design Suite チュートリアル: ロジック シミュレーション』 ([UG937](#)) を参照してください。

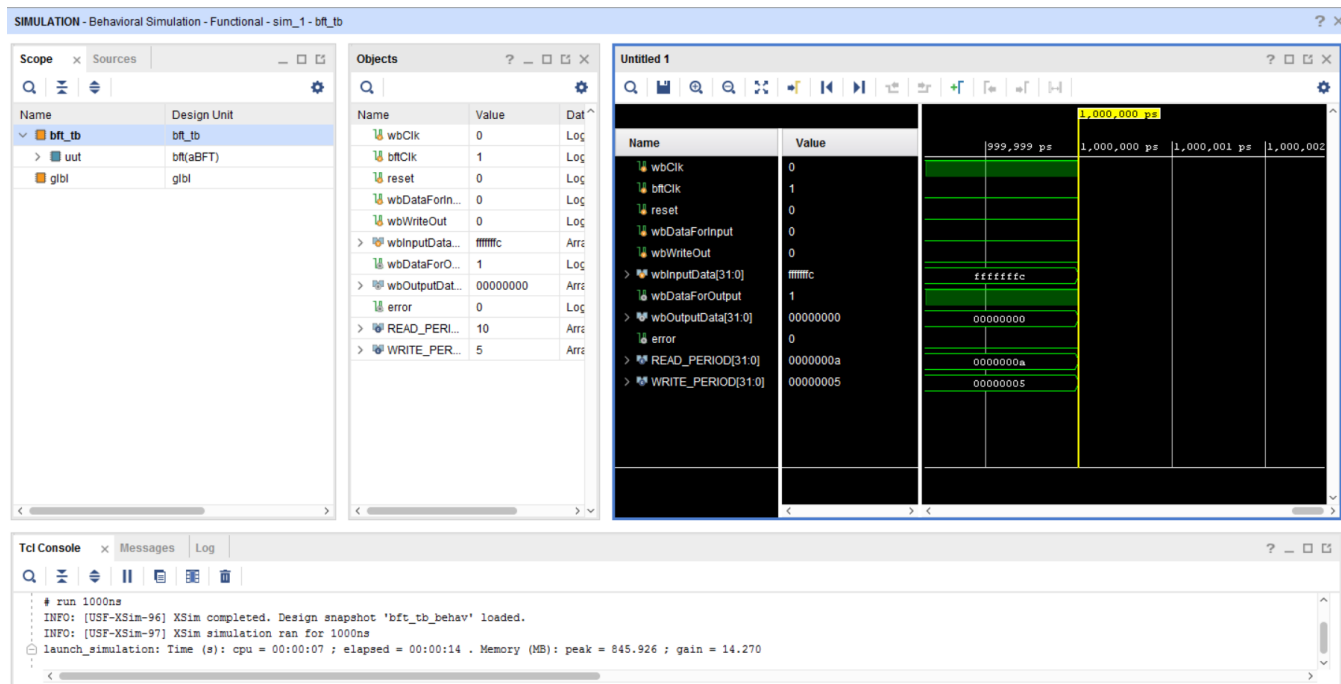
Vivado シミュレータの実行



重要: Vivado シミュレータを使用している場合は、シミュレーションを実行する前に、デザインに適切なプロジェクト設定を指定していることを確認してください。サポートされているサードパーティ シミュレータを使用している場合は、[第 3 章: サードパーティ シミュレータを使用したシミュレーション](#)を参照してください。

Flow Navigator で [Run Simulation] をクリックし、シミュレーション タイプを選択して、次の図に示す Vivado シミュレータのワークスペースを起動します。

図 6: Vivado シミュレータのワークスペース



メイン ツールバー

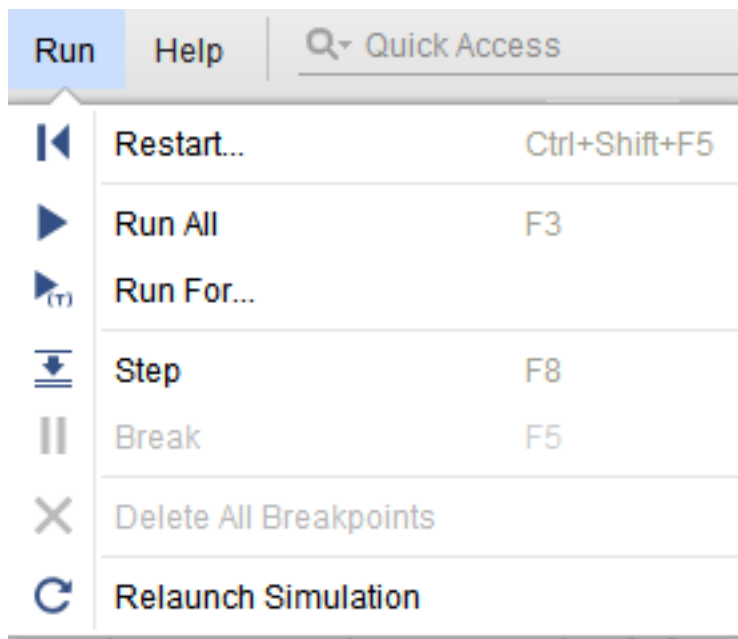
Vivado IDE でよく使用するコマンドを 1 クリックで実行できます。オプションの上にカーソルを置くと、詳細を示すツール ヒントが表示されます。

[Run] メニュー

シミュレーションを実行すると、Vivado IDE のメイン メニューに [Run] メニューが追加されます。

次の図に、シミュレーションの [Run] メニューを示します。

図 7: シミュレーションの [Run] メニュー コマンド



Vivado シミュレータの [Run] メニュー コマンドは次のとおりです。

- [Restart]: 既存のシミュレーションを時間 0 から再開します。Tcl コマンド: `restart`
- [Run All]: 開いているシミュレーションを最後まで実行します。Tcl コマンド: `run -all`
- [Run For]: 実行するシミュレーション時間を指定します。Tcl コマンド: `run <time>`



ヒント: 「`run 100 ns`」のように `run` コマンドで時間の単位を指定できますが、単位は省略することもできます。時間の単位を指定しない場合、Vivado シミュレータで `TIME_UNIT` Tcl プロパティで指定された単位が使用されます。`TIME_UNIT` プロパティを確認するには、Tcl コマンド `get_property time_unit [current_sim]` を使用します。`TIME_UNIT` プロパティを変更するには、Tcl コマンド `set_property time_unit <unit> [current_sim]` を使用します。`<unit>` には、`fs`、`ps`、`us`、`ms`、`s` のいずれかを指定します。

- [Step]: シミュレーションを次の HDL ソース行まで実行します。
- [Break]: 実行中のシミュレーションを停止します。
- [Delete All Breakpoints]: ブレークポイントをすべて削除します。
- [Relaunch Simulation]: シミュレーション ファイルを再コンパイルしてシミュレーションを再実行します。

関連情報

[デザイン変更後のシミュレーションの再実行](#)

[Simulation] ツールバー

Vivado シミュレータを実行すると、シミュレーション用のツールバー (次の図を参照) が、メインのツールバーの右側に開きます。

図 8: [Simulation] ツールバー



これらのボタンは [\[Run\] メニュー](#) のコマンドと同じで ([Delete All Breakpoints] オプションはなし)、使いやすくするために提供されています。

[Simulation] ツールバーのボタンの説明

ツールバーのボタンの上にカーソルを置くと、その説明がツール ヒントとして表示されます。

- [Restart]: シミュレーション時間を 0 にリセットします。
- [Run all]: すべてのイベントが完了するまで、またはシミュレーションを停止する HDL 文に到達するまで、シミュレーションを実行します。
- [Run For]: シミュレーションを指定した時間実行します。
- [Step]: 次の HDL 文までシミュレーションを実行します。
- [Break]: 現在のシミュレーションを停止します。
- [Relaunch]: コード変更した後などに、シミュレーション ソースを再コンパイルしてシミュレーションを再実行します。

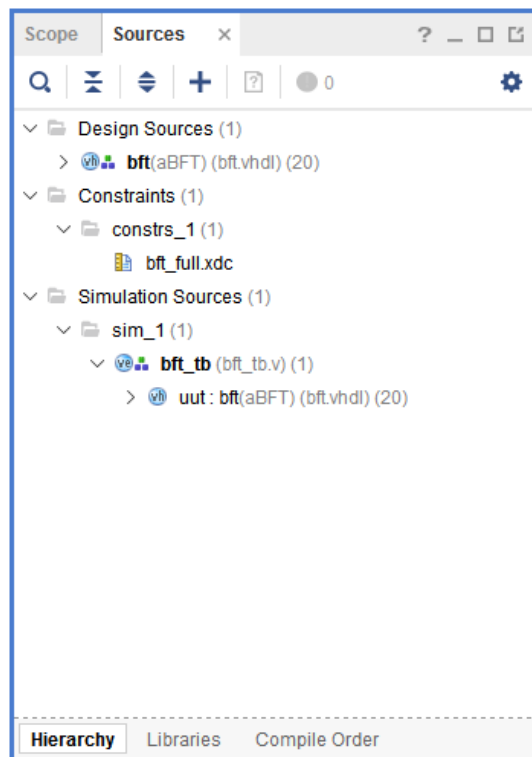
関連情報

[デザイン変更後のシミュレーションの再実行](#)

[Sources] ウィンドウ

[Sources] ウィンドウには、次の図に示すように、シミュレーション ソースが階層ツリーで表示され、[Hierarchy]、[IP Sources]、[Libraries]、[Compile Order] ビューがあります。

図 9: [Sources] ウィンドウ



[Sources] ウィンドウのボタンの上にカーソルを置くと、その説明がツール ヒントとして表示されます。ボタンを使用すると、ファイルの検証、展開/非展開、追加、開く、フィルターおよびスクロールなどを実行できます。

ソース オブジェクトを右クリックして [Open File] をクリックするとソース ファイルが開き、[Add Sources] をクリックするとソース ファイルを追加できます。

[Scopes] ウィンドウ

スコープとは HDL デザインを階層に分けたものです。デザイン ユニットをインスタンスエートするたび、またはプロセス、ブロック、パッケージ、サブプログラムを定義するたびに、スコープが作成されます。

次の図に示すように、[Scopes] ウィンドウにはデザイン階層が表示されます。この階層でスコープを選択すると、[Objects] ウィンドウにそのスコープの HDL オブジェクトがすべて表示されます。[Objects] ウィンドウで HDL オブジェクトを選択し、それを波形ビューアーに追加できます。

図 10: [Scopes] ウィンドウ


Name	Design Unit	Block Type
bft_tb	bft_tb	Verilog M...
vut	bft(aBFT)	VHDL Entity
> amd1	round_1(aR...	VHDL Entity
> amd2	round_2(aR...	VHDL Entity
> amd3	round_3(aR...	VHDL Entity
> amd4	round_4(aR...	VHDL Entity
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
> ingressLo...	bft(aBFT)	VHDL Block
ingressloop	bft(aBFT)	VHDL Block
> egressLo...	bft(aBFT)	VHDL Block
> egressLo...	bft(aBFT)	VHDL Block
> egressLo...	bft(aBFT)	VHDL Block

スコープのフィルター

- [Scope] ウィンドウの右上にある [Settings] ボタンをクリックすると、スコープ タイプの表示/非表示を切り替えることができます。



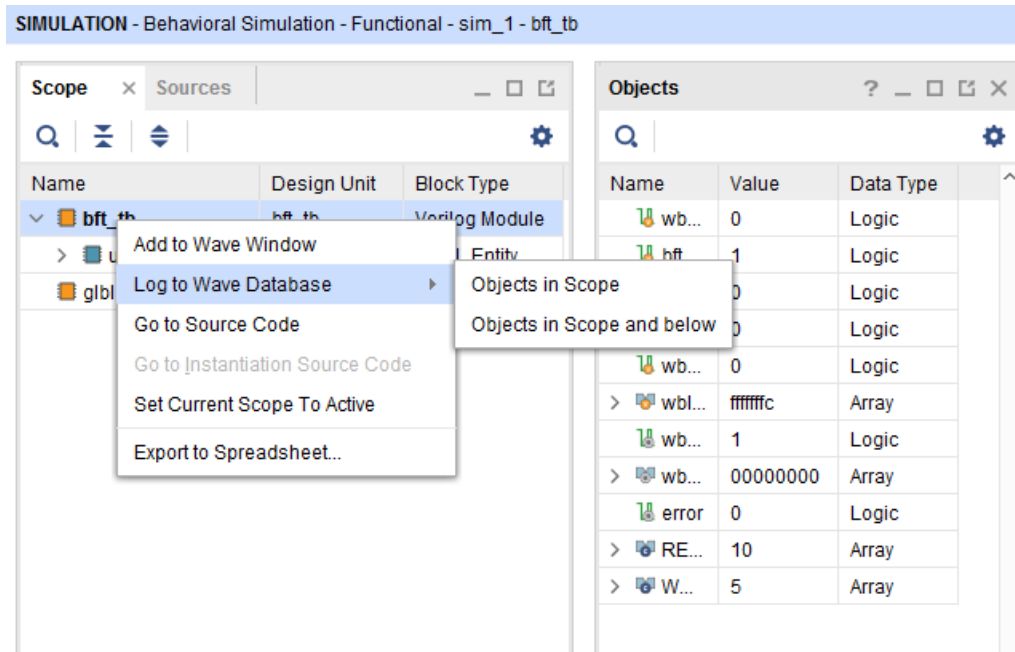
ヒント: [Settings] ボタンをクリックして表示されるオプションを使用してスコープを非表示にすると、スコープの種類に関係なく、そのスコープ内のすべてのスコープが非表示になります。たとえば、上の図で、すべての Verilog モジュール スコープを非表示にするため Verilog モジュールのボタンをクリックすると、`bft_tb` スコープだけでなく、`uut` は VHDL エンティティ スコープですが、`uut` も非表示になります。

- 特定の文字列を含むスコープに表示を限定するには、[Search] ボタン  をクリックし、テキスト ボックスに文字列を入力します。

[Objects] ウィンドウに表示されるオブジェクトは、選択しているスコープによって異なります。スコープを変更すると、異なるオブジェクトが表示されます。

スコープを右クリックすると、次の図に示すメニューが表示されます。

図 11: スコープのコンテキスト メニュー



- [Add to Wave Window]: 選択したスコープの表示可能な HDL オブジェクトすべてを波形設定に追加します。



ヒント: ビット幅の大きな HDL オブジェクトを波形ビューアーに追加すると、表示に時間がかかります。[Add to Wave Window] コマンドを実行する前に、波形設定で表示制限を設定すると、そのようなオブジェクトが表示されないようになります。それには、`set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]` という Tcl コマンドを使用します。

[Add to Wave Window] コマンドを使用すると、[Objects] ウィンドウに表示されている HDL オブジェクトとは異なる HDL オブジェクトが追加されることがあります。[Scopes] ウィンドウでスコープを選択すると、選択されたスコープで直接定義されているオブジェクトだけでなく、そのスコープに含まれる HDL オブジェクトも表示されることがあります。[Add to Wave Window] コマンドの場合は、選択したスコープのオブジェクトのみが追加されます。

または、オブジェクトを [Objects] ウィンドウから波形ウィンドウの [Name] 列にドラッグアンドドロップする方法もあります。



重要: 波形ウィンドウには、オブジェクトが追加されたシミュレーション時間から、そのオブジェクトの値の変化が表示されます。



ヒント: オブジェクト挿入以前のオブジェクト値を表示するには、シミュレーションを実行し直す必要があります。シミュレーション run を開始するときに、`log_wave -r /` Tcl コマンドを実行してデザインの表示可能な HDL オブジェクトすべての値を取り込んでおけば、値を表示するためにシミュレーションを実行し直す必要がなくなります。詳細は、[Tcl コマンド log_wave の使用](#) を参照してください。

波形設定の作成や HDL オブジェクトの追加などの波形設定への変更は、WCFG ファイルを保存しないと失われます。

- [Go To Source Code]: ソース コードの選択したスコープ定義の部分を開きます。
- [Go To Instantiation Source Code]: Verilog モジュールおよび VHDL エンティティ インスタンスに対し、インスタンス化時点のソース コードを開きます。

- [Set Current Scope to Active]: 現在のスコープを選択したスコープに設定します。選択したスコープがアクティブシミュレーション スコープになります (例: `get_property active_scope [current_sim]`)。アクティブシミュレーション スコープはその HDL プロセスのスコープであり、シミュレーションは現時点で一時停止されています。設定で [follow active scope] をオフにすると、Vivado シミュレータでシミュレーションが進んだ場合でも、最後の `current_scope` の選択が記録されます。ブレークポイントに到達すると、`current_scope` が最後のスコープをポイントしたままになり、アクティブ スコープとして設定されます。
- [Log to Wave Database]: 次のいずれかをログに記録します。
 - 。 現在のスコープのオブジェクト。
 - 。 現在のスコープおよびその下にあるすべてのスコープのオブジェクト。



ヒント: デフォルトでは、Vivado シミュレータのログに大型の HDL オブジェクトは記録されません。ログに記録されるオブジェクトのサイズ制限を変更するには、`set_property trace_limit <size> [current_sim] Tcl` コマンドを使用してください。<size> には、HDL オブジェクトのスカラ エLEMENT の数を指定します。

ソース コード テキスト エディターでコードの識別子にカーソルを置くと値が表示されます ([Scopes] ウィンドウ)。



重要: この機能を使用するには、[Scopes] ウィンドウで選択したソース コードにスコープが関連付けられていることを確認してください。

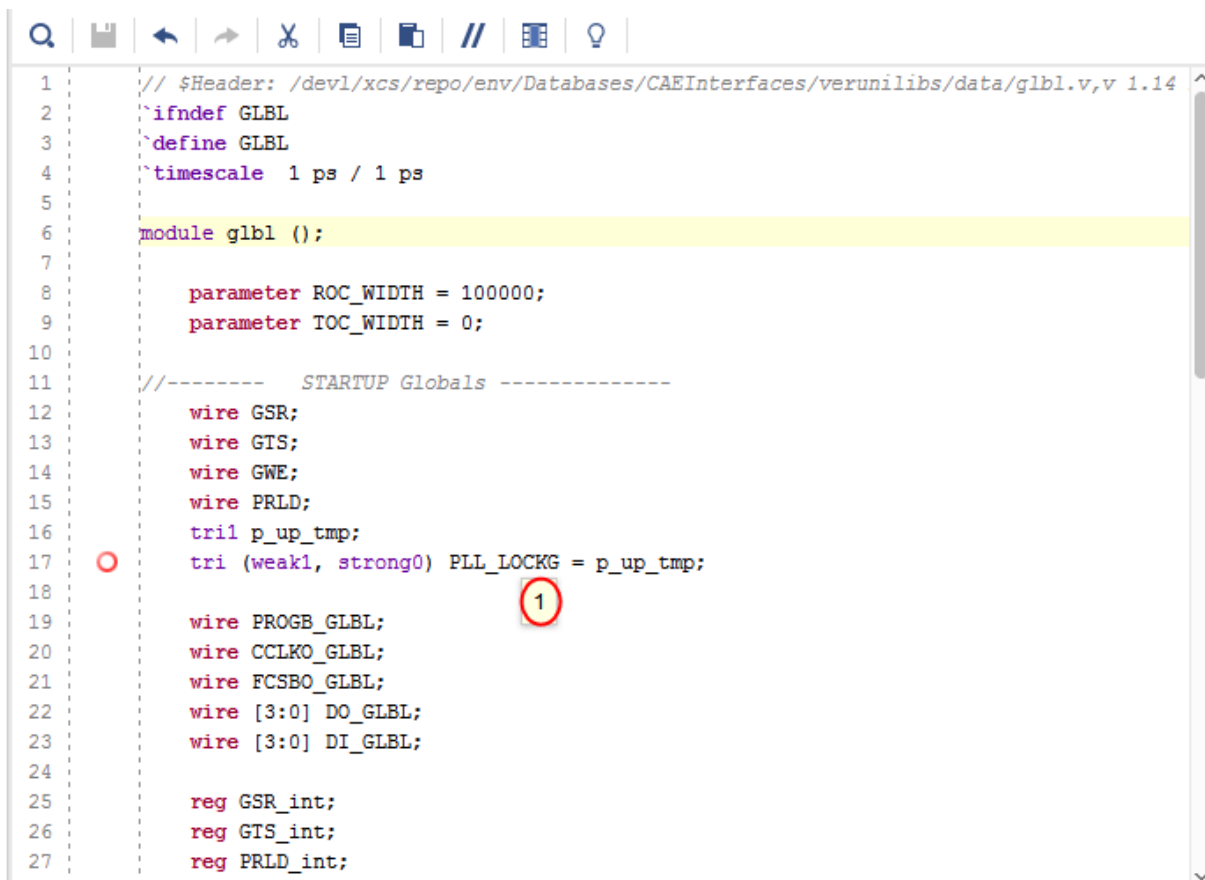


ヒント: 上の図では、最上位モジュールがインスタンス化されていないため、最上位モジュールを右クリックしても、[Go to Instantiation Source Code] は淡色表示になっています。



ヒント: 現在のスコープとそれ以下のスコープのオブジェクトをログに記録するには、`log_wave` を使用します。シミュレーション後は、波形に任意のオブジェクトを追加して、時間 0 から現在のシミュレーション時間までの値を確認できます。

図 12: 識別子の値が表示されたソース コード




```

1 // $Header: /dev1/xcs/repo/env/Databases/CAEInterfaces/verunilibs/data/glbl.v,v 1.14
2 `ifndef GLBL
3 `define GLBL
4 `timescale 1 ps / 1 ps
5
6 module glbl ();
7
8     parameter ROC_WIDTH = 100000;
9     parameter TOC_WIDTH = 0;
10
11     //----- STARTUP Globals -----
12     wire GSR;
13     wire GTS;
14     wire GWE;
15     wire PRLD;
16     tri1 p_up_tmp;
17     tri (weak1, strong0) PLL_LOCKG = p_up_tmp;
18
19     wire PROGB_GLBL;
20     wire CCLKO_GLBL;
21     wire FCSBO_GLBL;
22     wire [3:0] DO_GLBL;
23     wire [3:0] DI_GLBL;
24
25     reg GSR_int;
26     reg GTS_int;
27     reg PRLD_int;

```

その他のスコープおよびソース オプション

[Scopes] ウィンドウおよび [Sources] ウィンドウで [Show Search] ボタン  をクリックすると、検索フィールドが表示されます。

[Scopes] および [Objects] ウィンドウを使用する代わりに、Tcl コンソールに次を入力して HDL デザインを表示することもできます。

```

get_scopes
current_scope
report_scopes

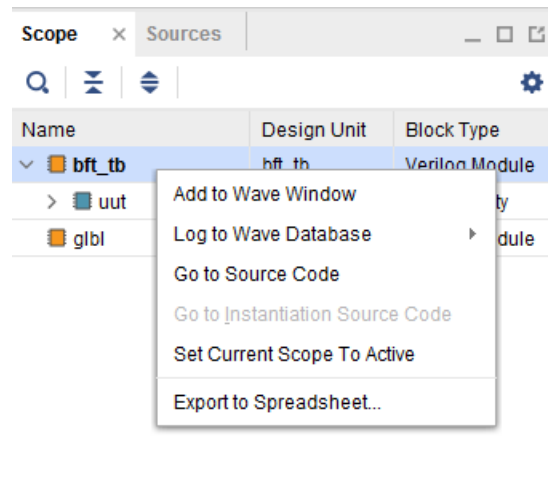
```


```
id="ai516872">report_values
```



ヒント: ソース ファイルを編集するには、[Scopes] ウィンドウ に示すように、[Scopes] ウィンドウまたは [Objects] ウィンドウでオブジェクトを右クリックし、[Go to Source Code] をクリックしてファイルを開きます。

図 13: [Scopes] ウィンドウのコンテキスト メニュー

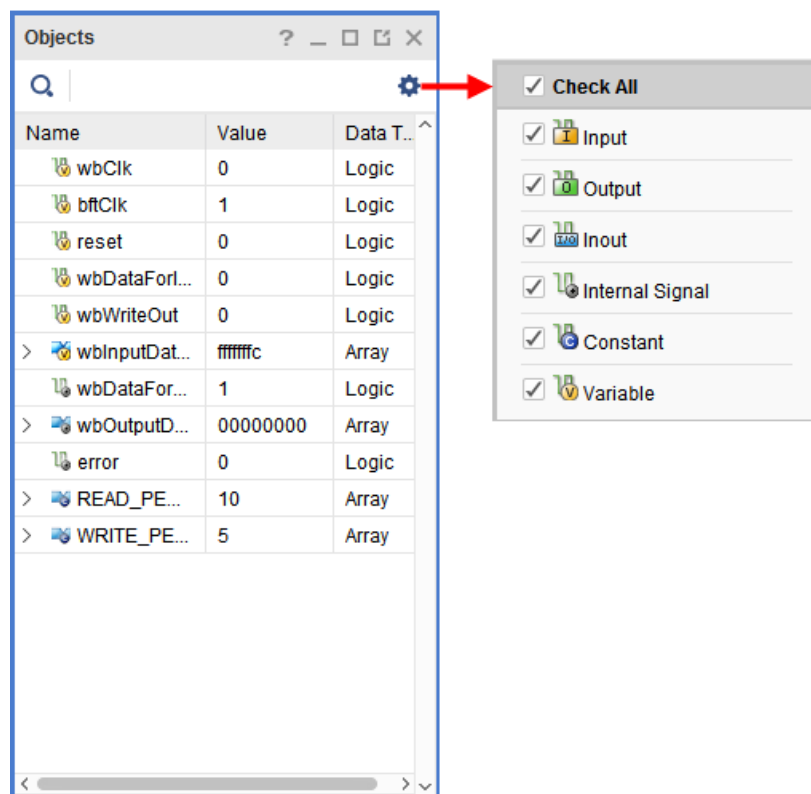


ヒント: ソース コードを編集してファイルを保存したら、[Relaunch] ボタン  をクリックすると、シミュレーションを閉じて再び開かずに、シミュレーションを再コンパイルおよび再実行できます。

[Objects] ウィンドウ

次の図に示すように、[Objects] ウィンドウには、[Scopes] ウィンドウで選択したスコープに関連付けられている HDL シミュレーション オブジェクトが表示されます。

図 14: [Objects] ウィンドウ



HDL オブジェクトの横にあるアイコンは、各オブジェクトのタイプまたはポート モードを示します。このウィンドウにはシミュレーション オブジェクトの名前、値、データ型などがリストされます。

次のコマンドを Tcl コンソールに入力すると、オブジェクトの現在の値を取得できます。

```
get_value <hdl_object>
```



ヒント: ベクターの表示桁数を制限するには、`set_property array_display_limit <bits>`
[current_sim] コマンドを使用してください。<bits> には、表示するビット数を指定します。

次の表に、[Objects] ウィンドウで使用可能なオプションを示します。[Settings] ボタンをクリックすると、[Objects] ウィンドウに表示するオブジェクトを選択して、[Objects] ウィンドウのリストを絞り込むことができます。

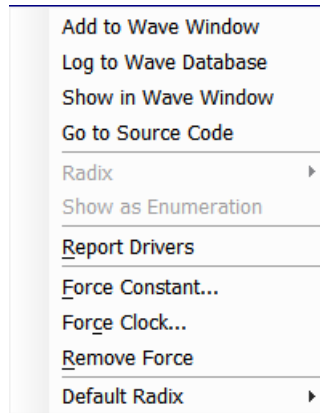
表 11: [Objects] ウィンドウのボタン

ボタン	説明
[Search]	検索するオブジェクト名を入力する検索フィールドを開きます。
[Settings]	ウィンドウでの HDL オブジェクトのタイプの表示/非表示を切り替えます。

[Objects] ウィンドウのコンテキスト メニュー

[Objects] ウィンドウでオブジェクトを右クリックすると、コンテキスト メニューが表示されます ([Objects] ウィンドウ)。次に、これらのコンテキスト メニューを説明します。

図 15: [Objects] ウィンドウのコンテキスト メニュー



- [Add to Wave Window]: 選択したオブジェクトを波形設定に追加します。または、オブジェクトを [Objects] ウィンドウから波形ウィンドウの [Name] 列にドラッグ アンド ドロップします。
- [Log to Wave Database]: 選択したオブジェクトのイベントを、後で波形ウィンドウに表示するために波形データベース (WDB) に記録します。



ヒント: デフォルトでは、Vivado シミュレータのログに大型の HDL オブジェクトは記録されません。ログに記録されるオブジェクトのサイズ制限を変更するには、`set_property trace_limit <size> [current_sim]` Tcl コマンドを使用してください。<size> には、HDL オブジェクトのスカラー エLEMENT の数を指定します。

- [Show in Wave Window]: 波形ウィンドウで選択されているオブジェクトをハイライトします。
- [Default Radix]: [Objects] ウィンドウおよびテキスト エディターすべてのオブジェクトのデフォルト 基数を設定します。デフォルト基数は、[Hexadecimal] (16 進数) です。このオプションはコンテキスト メニューから変更できます。

Tcl コマンド:

```
set_property radix <new radix> [current_sim]
```

<new radix> には次のいずれかを指定します。

- bin
- unsigned (符号なし 10 進数値)
- hex
- dec (符号付き 10 進数値)
- ascii
- oct
- smag (符号付き大きさ)

注記: 各信号の基数を変更する必要がある場合は、コンテキスト メニューから基数オプションを使用してください。

- [Radix]: [Objects] ウィンドウおよびソース コード ウィンドウで選択したオブジェクトの値を表示する際に使用する数値フォーマットを選択します。

個々のオブジェクトの基数を次のように変更できます。

1. [Objects] ウィンドウでオブジェクトを右クリックします。

2. コンテキスト メニューから [Radix] をクリックして、フォーマットを選択します。

- ・ [Default] (デフォルト)
- ・ [Binary] (2 進数)
- ・ [Hexadecimal] (16 進数)
- ・ [Octal] (8 進数)
- ・ [ASCII]
- ・ [Unsigned Decimal] (符号なし 10 進数)
- ・ [Signed Decimal] (符号付き 10 進数)
- ・ [Signed Magnitude] (符号付き大きさ)



ヒント: [Objects] ウィンドウで基数を変更しても、波形ウィンドウにはその変更は反映されません。

- ・ [Show as Enumeration]: SystemVerilog の列挙信号または変数の値を列挙ラベルを使用して表示します。

注記: このメニュー項目は、SystemVerilog の列挙型の場合にのみ選択可能になります。オフにすると、列挙オブジェクトの値すべてがそのオブジェクトの基数セットに従って数字で表示されます。オンにすると、エニユメレーション宣言で定義したラベルの値がそのラベル テキストで表示され、その他の値は数字で表示されます。

- ・ [Report Drivers]: 選択したオブジェクトに値を割り当てる HDL プロセスのレポートを [Tcl Console] ウィンドウに表示します。
- ・ [Go To Source Code]: ソース コードの選択したオブジェクトの定義を開きます。
- ・ [Force Constant]: 選択したオブジェクトを定数に割り当てます。
- ・ [Force Clock]: 選択したオブジェクトを定期的に切り替わる値に割り当てます。
- ・ [Remove Force]: 選択したオブジェクトに割り当てた値を削除します。



ヒント: 波形ビューアーに表示されていない HDL オブジェクトがある場合は、そのオブジェクトの波形トレースが Vivado シミュレータでサポートされていないことを示します。Verilog の名前付きイベントやローカル変数などの波形トレースはサポートされていません。

関連情報

[force コマンドの使用](#)

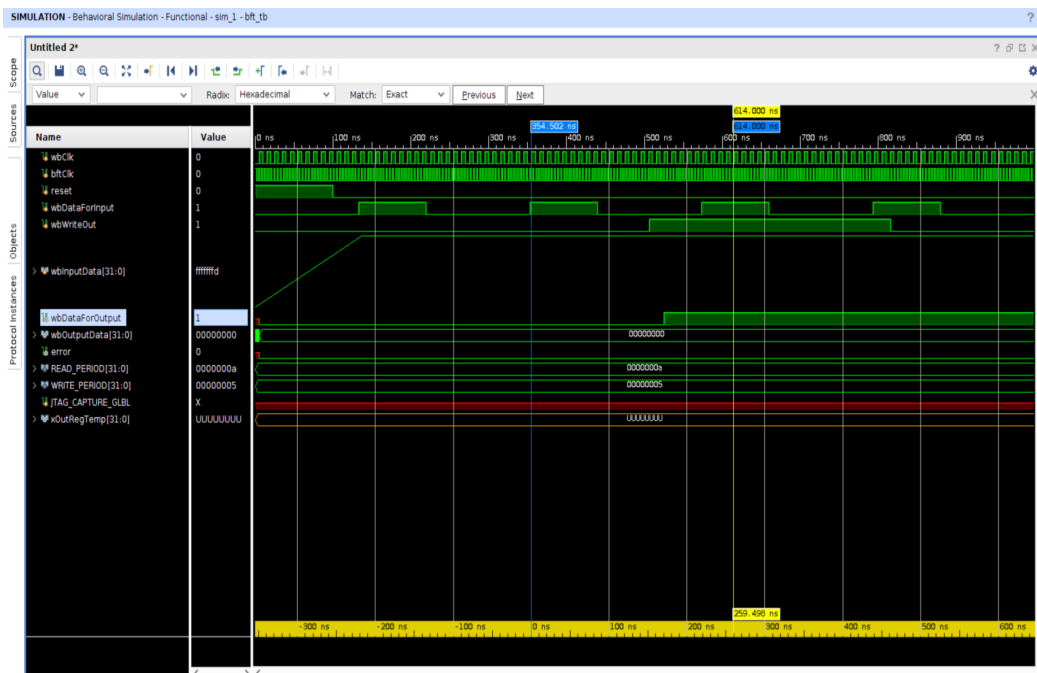
波形ウィンドウ

Vivado シミュレータを起動すると、デフォルトで波形ウィンドウが開きます。このウィンドウには、[波形ウィンドウ](#) に示すように、シミュレーションの最上位モジュールからトレース可能な HDL オブジェクトを含む波形設定が表示されます。



ヒント: プロジェクトを閉じて開き直したら、波形ウィンドウを表示するにはシミュレーションを再実行する必要があります。シミュレーション中にデフォルトの波形ウィンドウを誤って閉じてしまった場合は、メイン メニューから [Window] → [Waveform] をクリックすると波形ウィンドウが再び表示されます。

図 16: 波形ウィンドウ



波形ウィンドウに個々の HDL オブジェクト、またはオブジェクトのセットを追加するには、そのオブジェクトを右クリックして [Add to Wave Window] をクリックします ([Objects] ウィンドウ)。

Tcl コマンドを使用してオブジェクトを追加するには、`add_wave <HDL-objects>` を使用します。

`add_wave` コマンドを使用すると、HDL オブジェクトへの絶対パスまたは相対パスを指定できます。

たとえば、現在のスコープが `/bft_tb/uut` の場合、`uut` の下のリセット レジスタへの絶対パスは `/bft_tb/uut/reset`、相対パスは `reset` になります。

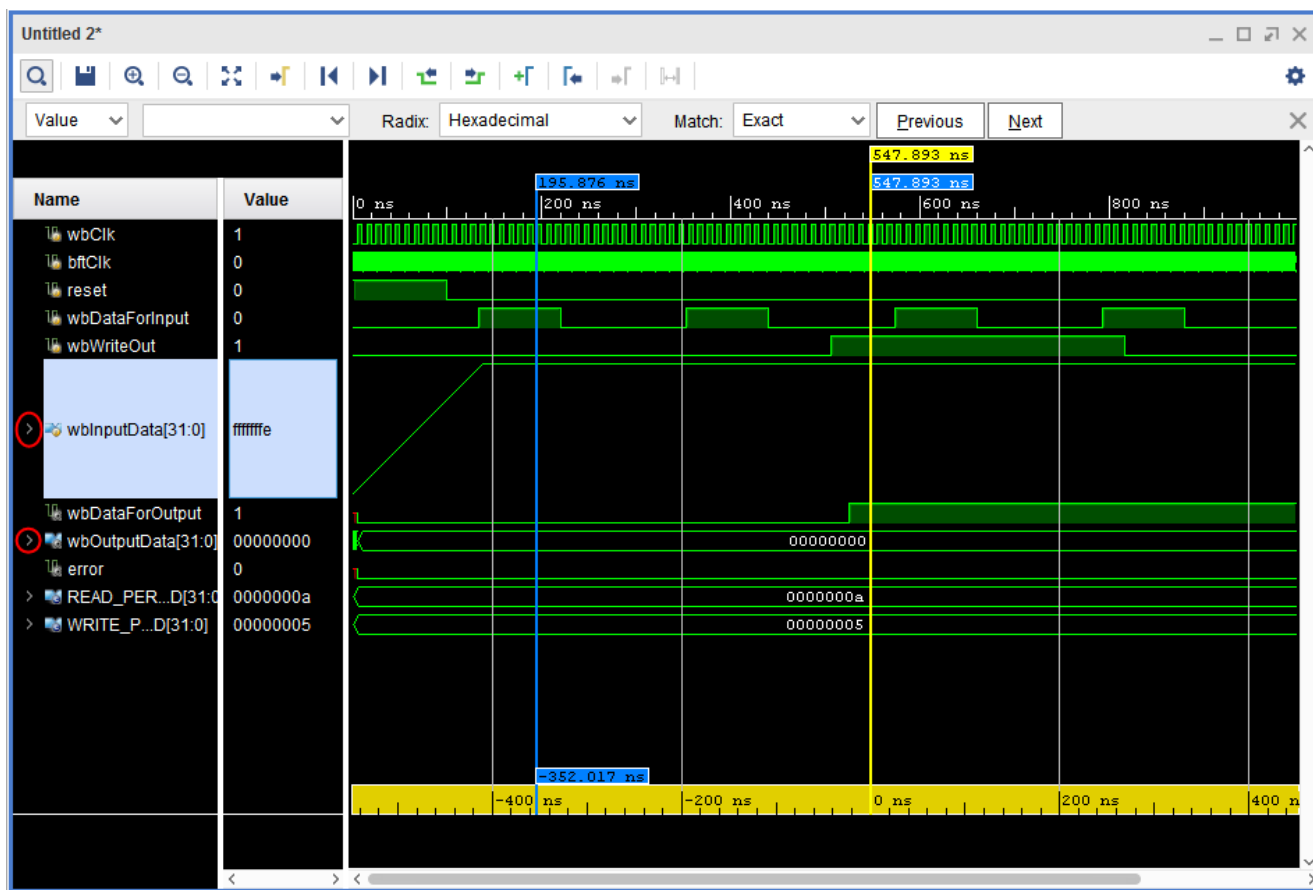


ヒント: `add_wave` コマンドには、HDL スコープおよび HDL オブジェクトを指定できます。`add_wave` でスコープを指定した場合、[Sources] ウィンドウの [Add To Wave Window] コマンドと同じ動作になります。ビット幅の大きな HDL オブジェクトを波形ビューアーに追加すると、表示に時間がかかります。[Add to Wave Window] コマンドを実行する前に、波形設定で表示制限を設定すると、そのようなオブジェクトが表示されないようになります。それには、`set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]` という Tcl コマンドを使用します。

波形オブジェクト

Vivado IDE の波形ウィンドウは、多くの Vivado Design Suite ツールで共通して使用されます。次の図に、波形設定に含まれる波形オブジェクトの例を示します。

図 17: 波形での HDL オブジェクト



波形ウィンドウには、HDL オブジェクトとその値および波形に加え、グループ、仕切り、仮想バスなどの HDL オブジェクトを見やすくするためのアイテムも表示されます。

HDL オブジェクトおよびそれを見やすくするためのアイテムは、まとめて「波形設定」と呼ばれます。波形ウィンドウの波形部分には、カーソル、マーカー、時間軸などの時間計測用のアイテムも表示されます。

Vivado IDE ではシミュレーション中の HDL オブジェクトの値変化が波形ウィンドウに表示されるので、ユーザーは波形設定を使用してシミュレーション結果を検証します。

デザイン階層およびシミュレーションの波形は波形設定の一部ではなく、別の波形データベース ファイル (WDB) に保存されます。

波形ウィンドウのコンテキスト メニュー

波形ウィンドウでオブジェクトを右クリックすると、次の図に示すコンテキスト メニューが表示されます。波形ウィンドウの HDL オブジェクトの詳細は、[波形設定の HDL オブジェクト](#)を参照してください。次に、これらのコンテキスト メニュー コマンドを説明します。

- [Go To Source Code]: ソース コードの選択したデザイン波形オブジェクトの定義部分を開きます。
- [Show in Object Window]: デザイン波形オブジェクトの HDL オブジェクトを [Objects] ウィンドウでハイライトします。

- [Report Drivers]: 選択した波形オブジェクトに値を割り当てる HDL プロセスのレポートを [Tcl Console] ウィンドウに表示します。
- [Force Constant]: 選択したオブジェクトを定数に割り当てます。
- [Force Clock]: 選択したオブジェクトを定期的に切り替わる値に割り当てます。
- [Remove Force]: 選択したオブジェクトに割り当てた値を削除します。
- [Find]: 波形ウィンドウの上部に [Find] ツールバーを表示し、波形オブジェクトを名前で検索します。
- [Find Value]: 波形ウィンドウの上部に [Find] ツールバーを表示し、波形を値で検索します。
- [Select All]: 波形ウィンドウの波形オブジェクトをすべて選択します。
- [Expand]: 選択した波形オブジェクトのサブオブジェクトを表示します。
- [Collapse]: 選択した波形オブジェクトのサブオブジェクトを非表示にします。
- [Ungroup]: 選択したグループまたは仮想バスを解除します。
- [Rename]: 選択した波形オブジェクトの表示名を変更します。
- [Name]: 選択した波形オブジェクトの名前を完全な階層名 (long name)、信号名またはバス名のみ (short name)、あるいはカスタム名で表示します。
- [Waveform Style]: 選択したデザイン波形オブジェクトの波形フォーマットをデジタルまたはアナログにします。
- [Signal Color]: 選択したデザイン波形オブジェクトの色を設定します。
- [Divider Color]: 選択した仕切りの色を設定します。
- [Radix]: 選択したデザイン波形オブジェクトの表示値の基数を設定します。
- [Show as Enumeration]: 選択した SystemVerilog 列挙波形オブジェクトの値を数値の代わりに列挙ラベルで表示します (可能な場合)。
- [Reverse Bit Order]: 選択した配列波形オブジェクトに表示されている値のビット順を逆にします。
- [New Group]: 選択した波形オブジェクトをフォルダーのようなグループ波形オブジェクトにまとめます。
- [New Divider]: 波形ウィンドウの波形オブジェクトのリストに水平の仕切りを作成します。
- [New Virtual Bus]: 選択したデザイン波形オブジェクトのビットで構成される新しいロジック ベクター波形オブジェクトを作成します。
- [Cut]: [Waveform] ウィンドウで信号を切り取ります。
- [Copy]: [Waveform] ウィンドウで信号をコピーします。
- [Paste]: [Waveform] ウィンドウで信号を貼り付けます。
- [Delete]: [Waveform] ウィンドウで信号を削除します。

図 18: 波形ウィンドウのコンテキスト メニュー

Go To Source Code	
Show in Object Window	
Report Drivers	
Force Constant...	
Force Clock...	
Remove Force	
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Delete
Find...	Ctrl+F
Find Value...	Ctrl+Shift+F
Select All	Ctrl+A
Expand	
Collapse	
Ungroup	
Rename	F2
Name	▶
Waveform Style	▶
Signal Color	▶
Divider Color	▶
Radix	▶
Show as Enumeration	
Reverse Bit Order	
New Group	
New Divider	
New Virtual Bus	

波形ウィンドウの使用に関する詳細は、[第 5 章: Vivado シミュレータを使用したシミュレーション波形の解析](#)を参照してください。

波形設定の保存

新規波形設定はディスクに自動的に保存されません。[File]→[Simulation Waveform]→[Save Configuration As] をクリックし、ファイル名を指定して WCFG ファイルを保存します。

波形設定を WCFG ファイルに保存するには、Tcl コンソールに「`save_wave_config <filename.wcfg>`」と入力します。

指定したコマンド引数の名前で WCFG ファイルが保存されます。



重要: ズーム設定は波形設定には保存されません。

関連情報[アナログ波形の使用](#)[SystemVerilog の列挙フォーマットの変更](#)[波形の分類](#)[波形オブジェクトの命名](#)[force コマンドの使用](#)[波形設定での値の検索](#)[信号およびオブジェクトのグループ](#)[バス ビット順の反転](#)

複数の波形設定の作成と使用

1 つのシミュレーションセッションで、独自の波形ウィンドウを使用する複数の波形設定を作成および使用できます。複数の波形ウィンドウが表示されている場合は、一番最近作成したウィンドウまたは最近使用したウィンドウがアクティブ ウィンドウとなります。アクティブ ウィンドウは、現在表示されているウィンドウと、[HDL Objects]→[Add to Wave Window] などの外部コマンドが適用される波形ウィンドウです。

別の波形ウィンドウをアクティブにするには、そのウィンドウのタイトルをクリックします。

関連情報[複数のシミュレーション run の区別](#)[波形設定の新規作成](#)

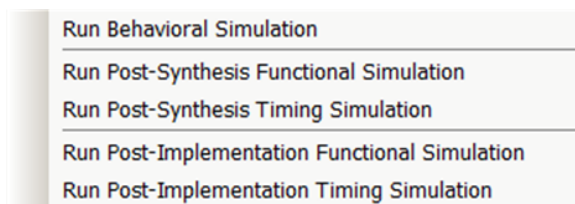
論理およびタイミング シミュレーションの実行

Vivado Design Suite でプロジェクトを作成したら、すぐにビヘイビア シミュレーションを実行できます。合成またはインプリメンテーションを実行した後は、論理およびタイミング シミュレーションを実行できます。シミュレーションを実行するには、Flow Navigator で [Run Simulation] をクリックし、次の図に示すように、ポップアップメニューから実行するシミュレーションを選択します。



ヒント: ポップアップメニューでどのオプションを選択できるかは、デザイン開発段階によって異なります。たとえば、合成は実行したがインプリメンテーションはまだである場合は、ポップアップメニューでインプリメンテーションに関するシミュレーションは淡色表示になります。

図 19: [Run Simulation] のポップアップメニュー



論理シミュレーションの実行

合成後の論理シミュレーション

合成 run が正常に完了すると、[Run Simulation]→[Post-Synthesis Functional Simulation] が表示されるようになります。

合成後は、汎用ロジック デザインがデバイス特定のプリミティブに合成されています。合成後に論理シミュレーションを実行することにより、合成の最適化によりデザイン機能が変わっていないことを確認できます。合成後の論理シミュレーションを選択すると、論理ネットリストが生成され、その UNISIM ライブラリがシミュレーションに使用されます。

インプリメンテーション後の論理シミュレーション

インプリメンテーション run が完了すると、[Run Simulation]→[Post-Implementation Functional Simulation] を実行できるようになります。

インプリメンテーション後は、デザインはハードウェアに配置配線されています。この段階で論理検証をしておき、インプリメンテーション中に実行された物理的最適化によりデザインの機能が変更されていないかを確認できます。

インプリメンテーション後の論理シミュレーションを選択すると、論理ネットリストが生成され、その UNISIM ライブラリがシミュレーションに使用されます。

タイミング シミュレーションの実行



ヒント: 合成後のタイミング シミュレーションでは、デバイス モデルから見積もられたタイミング遅延が使用され、インターコネクト遅延は含まれません。インプリメンテーション後のタイミング シミュレーションでは実際のタイミング遅延が使用されます。

合成後およびインプリメンテーション後のタイミング シミュレーションを実行するとき、シミュレータには次のものが含まれます。

- SIMPRIM ライブラリ コンポーネントを含むゲートレベルのネットリスト
- SECUREIP
- 標準遅延フォーマット (SDF) ファイル

デザインの全体的な機能は最初に定義しています。デザインがインプリメントされると、正確なタイミング情報が利用できるようになります。

ネットリストおよび SDF を作成するため、Vivado Design Suite は次の処理を実行します。

- `write_verilog` オプションを使用してネットリスト ライターの `-mode timesim`、および `write_sdf` (SDF アノテーター) を呼び出します。
- 生成されたネットリストをターゲット シミュレータに送ります。

これらのオプションを変更するには、[シミュレーション設定](#)で説明されているように、[Settings] ダイアログ ボックスの [Simulation] ページを使用します。



重要: 合成後とインプリメンテーション後のタイミング シミュレーションは、Verilog でのみサポートされます。VHDL のタイミング シミュレーションはサポートされません。VHDL を使用する場合は、合成後およびインプリメンテーション後の論理シミュレーションを実行できます。この場合、SDF アノテーションは必要なく、シミュレーション ネットリストで UNISIM ライブラリが使用されます。ネットリストは `write_vhdl` Tcl コマンドを使用して作成できます。使用情報は、『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。



重要: Vivado シミュレータ モデルではインターコネクト遅延が使用されるので、タイミング シミュレーションが正しく実行されるようにするには、`-transport_int_delays -pulse_r 0 -pulse_int_r 0` オプションを追加する必要があります。

合成後のタイミング シミュレーション

合成 run が正常に完了すると、[Run Simulation]→[Post-Synthesis Timing Simulation] を実行できるようになります。

合成後は、汎用ロジック デザインがデバイス特定のプリミティブに合成され、配線遅延およびコンポーネント遅延の見積もり値が利用できます。合成後のタイミング シミュレーションを実行することにより、インプリメンテーション段階で時間を費やす前に、タイミング クリティカル パスがないかを確認できます。合成後のタイミング シミュレーションを選択すると、タイミング ネットリストと SDF ファイルの見積もり遅延が生成されます。シミュレータで生成された SDF ファイルが含まれるよう、ネットリスト ファイルには `$sdf_annotate` コマンドが含まれます。

インプリメンテーション後のタイミング シミュレーション

インプリメンテーション run が完了すると、[Run Simulation]→[Post-Implementation Timing Simulation] を実行できるようになります。

インプリメンテーション後は、デザインはハードウェアにインプリメントおよび配線されています。この段階でタイミング シミュレーションを実行すると、正確なタイミング遅延を使用して、指定速度でデザインが機能しているかどうかを確認できます。このシミュレーションは、制約が設定されていないパスや、非同期パスのタイミング エラー(リセットで発生するエラーなど)を検出するのに便利です。インプリメンテーション後のタイミング シミュレーションを選択すると、タイミング ネットリストと SDF ファイルが生成されます。ネットリスト ファイルには `$sdf_annotate` コマンドが含まれるので、生成された SDF ファイルが自動的に指定されます。

タイミング シミュレーション用の SDF ファイルのアノテート

シミュレーション設定を指定したときに、SDF ファイルを作成するかどうかと、プロセス コーナーをファーストまたはスローに設定しました。



ヒント: SDF ファイルのオプションの設定を確認するには、Vivado IDE の Flow Navigator で [Simulation] を右クリックし、[Simulation Settings] をクリックします。[Settings] ダイアログ ボックスの [Simulation] ページで、[Netlist] タブをクリックします。

SDF ファイルには、指定したプロセス コーナーに基づいて異なる `min` および `max` 値が含まれます。

2 つの異なるシミュレーションを実行して、セットアップおよびホールド違反をチェックをしてください。

セットアップ チェックを実行するには、`[-process_corner]` を `[slow]` に設定して SDF を作成し、SDF ファイルの `max` 列を使用します。

ホールド チェックを実行するには、`[-process_corner]` を `[fast]` に設定して SDF ファイルを作成し、SDF ファイルの `min` 列を使用します。使用する SDF 遅延フィールドの指定方法は、使用するシミュレーション ツールによって異なります。このオプションの設定方法は、ご使用のシミュレーション ツールの資料を参照してください。

4 つのタイミング シミュレーションすべてを実行するには、次のように指定します。

- スロー コーナー: SDFMIN および SDFMAX

- ファースト コーナー: SDFMIN および SDFMAX

シミュレーション結果の保存

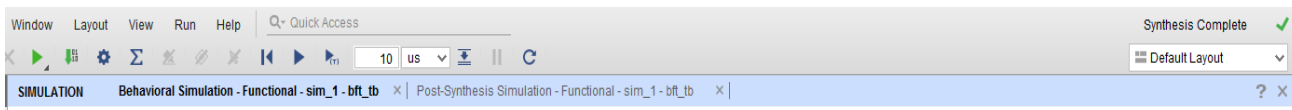
Vivado シミュレータでは、<project>.sim/<simset> ディレクトリの波形データベース (WDB) ファイル (<filename>.wdb) にオブジェクト (VHDL 信号、Verilog レジスタまたはワイヤなど) のシミュレーション結果が保存されます。

オブジェクトを波形ウィンドウに追加してシミュレーションを実行すると、デザインの完全な階層と追加したオブジェクトのトランザクションが自動的に WDB ファイルに保存されます。また、log_wave コマンドを使用して、波形ウィンドウには表示されないオブジェクトを波形データベースに追加することもできます。log_wave コマンドの詳細は、[Tcl コマンド log_wave の使用](#) を参照してください。

複数のシミュレーション run の区別

デザインに対して複数のシミュレーションを実行すると、次の図に示すように、Vivado シミュレータのワークスペース上部に各シミュレーションの名前を示すタブが表示され、現在ウィンドウに表示されているシミュレーション タイプがハイライトされます。

図 20: アクティブなシミュレーション タイプ



シミュレーションを閉じる

シミュレーションを閉じるには、Vivado IDE で次を実行します。

[File]→[Exit] をクリックするか、プロジェクト ウィンドウの右上の [X] マークをクリックします。



注意: 複数のシミュレーションを実行している場合は、青いタイトル バーの [X] マークをクリックすると、すべてのシミュレーションが閉じます。1 つのシミュレーションを閉じるには、青いタイトル バーの下小さなグレーまたは白いタブの [X] マークをクリックします。

Tcl コンソールからシミュレーションを閉じるには、次を入力します。

```
close_sim
```

このコマンドは、まず保存されていない波形設定がないかどうかをチェックします。あった場合は、エラー メッセージが表示されます。close_sim コマンドを実行する前に未保存の波形設定を閉じるか、または保存します。または -force オプションを Tcl コマンドに追加します。

注記: `close_sim` コマンドを使用して現在のプロジェクトを閉じる前に、`close_project` コマンドを使用してシミュレーションを完全に閉じることをお勧めします。

シミュレーション起動スクリプト ファイルの追加

バッチ ファイルのカスタム Tcl コマンドをプロジェクトに追加して、それらの Tcl コマンドがシミュレーションで実行されるようにできます。これらのコマンドはシミュレーション開始後に実行されます。このプロセスは次のようになります。

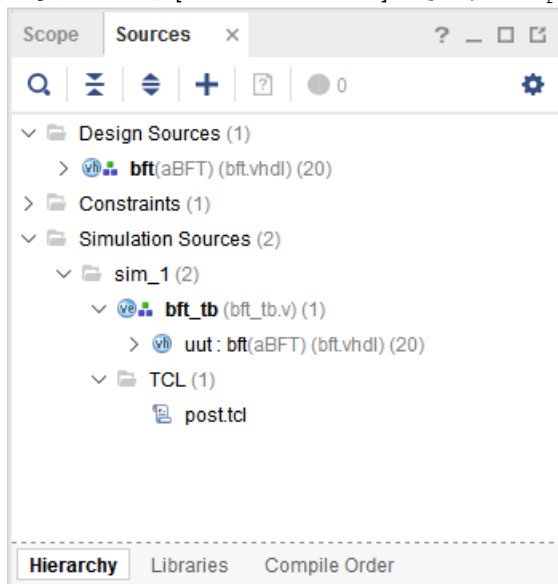
1. シミュレーション ソース ファイルに追加するシミュレーション コマンドを含む Tcl スクリプトを作成します。たとえば、1,000 ns 間実行するシミュレーションがあり、実行時間を長くする場合、次のコマンドを含むファイルを作成します。

```
run 5us
```

デフォルトでは最上位の信号のみが波形に追加されますが、最上位にない信号を監視する場合は、`post.tcl` スクリプトにこれらの信号を追加します。次に例を示します。

```
add_wave/top/I1/<signalName>
```

2. このファイルを「`post.tcl`」という名前を付けて保存します。
3. Flow Navigator で [Add Sources] をクリックして Add Sources ウィザードを起動し、[Add or Create Simulation Sources] をオンにします。
4. `post.tcl` ファイルをシミュレーション ソースとして Vivado Design Suite プロジェクトに追加します。次の図に示すように、[Simulation Sources] フォルダに `post.tcl` ファイルが表示されます。



5. [Simulation] ツールバーの [Relaunch] ボタン  をクリックします。

元の時間に `post.tcl` ファイルで指定された時間が追加された時間シミュレーションが実行されます。Vivado シミュレータは、そのコマンドをすべて実行した後に、`post.tcl` ファイルを自動的に呼び出します。

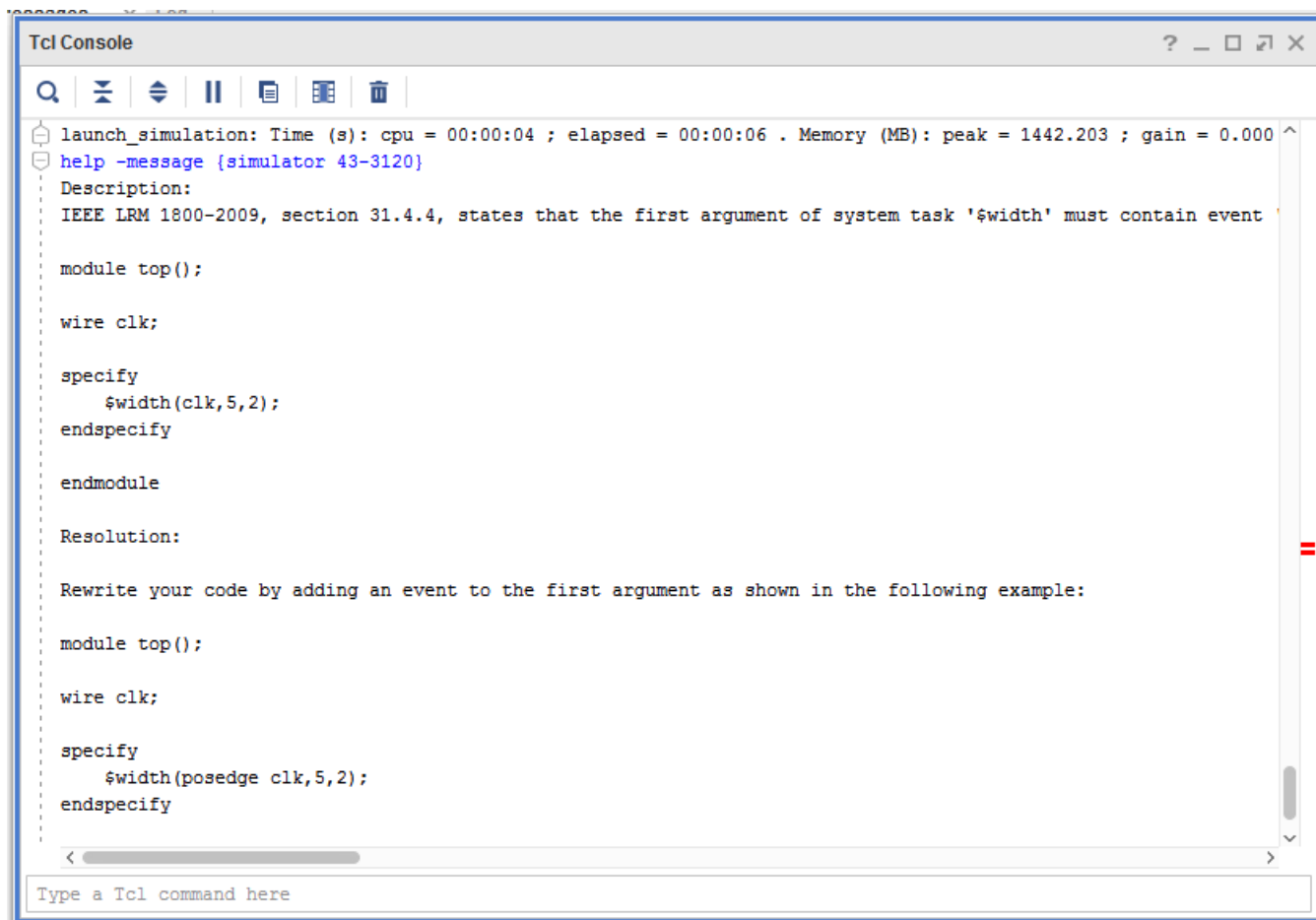
シミュレーション メッセージの表示

Vivado IDE には、情報、警告、エラーなどのメッセージが表示されるメッセージ フィールドが含まれます。Vivado シミュレータからの一部のメッセージには、次の図に示すように、問題の詳細と推奨回避策が記載されます。

同じ詳細を Tcl コンソールに表示するには、次を構文を使用します。

```
help -message {message_number}
```

図 21: シミュレータ メッセージの説明と回避策情報



次に例を示します。

```
help -message {simulator 43-3120}
```

メッセージ出力の管理

HDL デザインで生成されるメッセージ数が非常に多い場合 (\$display Verilog システム タスクや、report VHDL 文により生成されるものなど)、Tcl コンソールおよびログ ファイルに出力されるメッセージ文の量を制限できます。これにより、コンピューターのメモリやディスク容量を節約できます。これには、`-maxlogsize` オプションを使用します。

1. Flow Navigator で [SIMULATION] を右クリックし、[Simulation Settings] をクリックします。
2. [Settings] ダイアログ ボックスで、`xsim.simulate.xsim.more_options` の横に `-maxlogsize <size>` (`<size>` はテキスト出力の最大量 (MB)) を追加します。

launch_simulation コマンドの使用

`launch_simulation` コマンドを使用すると、サポートされているシミュレータをスクリプト モードで実行できます。

`launch_simulation` の構文は、次のとおりです。

```
launch_simulation [-step <arg>] [-simset <arg>] [-mode <arg>] [-type <arg>]
                  [-scripts_only] [-of_objects <args>] [-absolute_path]
                  [-install_path <arg>] [-noclean_dir] [-quiet] [-
verbose]
```

次の表に、`launch_simulation` のオプションを説明します。

表 12: `launch_simulation` のオプション

オプション	説明
<code>[-step]</code>	実行するシミュレーション手順を指定します。有効な値は <code>all</code> 、 <code>compile</code> 、 <code>elaborate</code> 、 <code>simulate</code> です。デフォルトは <code>all</code> (すべてのステップを実行) です。
<code>[-simset]</code>	シミュレーション ファイルセットの名前を指定します。
<code>[-mode]</code>	シミュレーション モードを指定します。有効な値は <code>behavioral</code> 、 <code>post-synthesis</code> 、 <code>post-implementation</code> で、デフォルトは <code>behavioral</code> です。
<code>[-type]</code>	ネットリストのタイプを指定します。有効な値は <code>functional</code> 、 <code>timing</code> です。このオプションは、 <code>-mode</code> を <code>post-synthesis</code> または <code>post-implementation</code> に設定している場合にのみ有効です。
<code>[-scripts_only]</code>	スクリプトの生成のみを実行します。
<code>[-of_objects]</code>	指定したオブジェクトのコンパイル順ファイルを生成します。 <code>-scripts_only</code> オプションを使用している場合にのみ有効です。
<code>[-absolute_path]</code>	すべてのファイルパスを参照ディレクトリに対する絶対パスにします。
<code>[-install_path]</code>	インストール ディレクトリ パスを指定します。
<code>[-noclean_dir]</code>	シミュレーション実行ディレクトリ ファイルを削除しません。
<code>[-quiet]</code>	コマンド エラーを表示しません。
<code>[-verbose]</code>	メッセージの非表示設定を解除し、すべてのメッセージを表示します。

例

- Vivado シミュレータを使用してビヘイビア シミュレーションを実行します。

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
launch_simulation
```

- Questa Advanced Simulator を使用してビヘイビア シミュレーション用のスクリプトを生成します。

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
set_property target_simulator Questa [current_project]
set_property compxlib.questa_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -scripts_only
```

- Synopsys VCS を使用して合成後の論理シミュレーションを起動します。

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -mode post-synthesis -type functional
```

- Cadence IES を使用してインプリメンテーション後のタイミング シミュレーションを実行します。

```
set_property target_simulator IES [current_project]
set_property compxlib.ies_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -mode post-implementation -type timing
```

デザイン変更後のシミュレーションの再実行

Vivado シミュレータで HDL デザインをデバッグ中に HDL ソース コードを修正する必要があるかどうかを判断できます。

デザインを変更してシミュレーションを実行し直す手順は、次のとおりです。


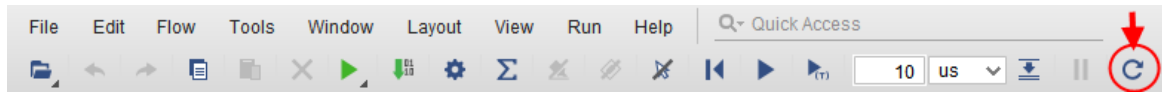
1. Vivado コード エディターまたはその他のテキスト エディターを使用して、ソース コードに必要な変更を加えて保存します。
2. Vivado IDE ツールバーの [Relaunch Simulation] ボタン  をクリックしてシミュレーションを再実行します。または、Tcl コマンドの `relaunch_sim` を使用して、シミュレーションをコンパイルし直して再起動します。

図 22: [Relaunch Simulation] ボタン



- 変更したデザインがコンパイルできない場合、そのエラーの原因を示すエラー メッセージが表示されます。Vivado IDE には、ディスエーブル ステートの前のシミュレーション run の結果が引き続き表示されます。手順 1 に戻り、エラーを修正して、シミュレーションを再実行します。


デザインが問題なく再コンパイルされたら、再びシミュレーションが開始します。



重要: ファイル システム エラーなど、コンパイル エラー以外の理由でシミュレーションを再実行できないこともあります。[Simulation] ツールバーの [Run] ボタンが再実行後に淡色表示になっている場合は、シミュレーションがディスエーブルになっていることを意味します。Tcl コンソールの内容から、再実行できなかった理由を確認してください。



注意: シミュレーションは、Flow Navigator の [Run Simulation] をクリックするか、Tcl コマンドの `launch_simulation` を使用しても再実行できますが、これらのコマンドを使用するとシミュレーションが閉じ、波形の変更や基数のカスタマイズなどのシミュレーション設定が破棄されることがあります。

注記: [Relaunch Simulation] ボタン  は、`launch_simulation` を使用して Vivado シミュレータが一回問題なく実行された後にのみ使用可能になります。シミュレーションがバッチ/スクリプト モードで実行されると、[Relaunch Simulation] ボタンは淡色表示されます。

保存されたシミュレータのユーザー インターフェイス設定の使用

デフォルトでは、Vivado シミュレータのユーザー インターフェイスおよび Tcl コマンドを使用した設定の変更は、Vivado シミュレータによりシミュレーションの作業ディレクトリにあるファイルに保存されます。保存される設定は、次のとおりです。

- [Scopes] および [Objects] ウィンドウのフィルター ボタンの状態および列の幅。
- run コマンドの配列表示制限、デフォルト基数、デフォルト時間単位、トレース制限など、シミュレーションの Tcl プロパティ。
- [Objects] ウィンドウで HDL オブジェクトに設定した基数および [Show as Enumeration] の設定。

シミュレーションを終了した後、Vivado シミュレータをもう一度開いて実行すると、Vivado シミュレータで前回の設定が復元されます。



重要: シミュレーションを再起動したときに設定ファイルが消去されないようにするには、Vivado の [Project Settings] ダイアログ ボックスの [Simulation] ページで [Clean Up Simulation Files] チェック ボックスをオフにします。



ヒント: 設定をデフォルト値に戻すには、設定ファイルを削除してください。設定ファイルは、Vivado プロジェクトディレクトリの `<project>.sim/<simset>/<simtype>/xsim.dir/<snapshot>/xsimSettings.ini` にあります。たとえば、BFT サンプル デザインのデフォルト ビヘイビア シミュレーション run の設定ファイルは、`bft.sim/sim_1/behav/xsim.dir/bft_tb_behav/xsimSettings.ini` にあります。ファイルが消去されてもかわまない場合は、[Clean Up Simulation Files] をオンにします。

デフォルト設定

Vivado® プロジェクトの Tcl オブジェクトでは、クリーンアップされたシミュレーションまたは新しく作成されたシミュレーションのデフォルト設定を指定するためのプロパティがいくつかサポートされています。これらのシミュレーションには設定ファイルはまだありません。次に、プロジェクトのデフォルト設定プロパティをリストします。

- XSIM.ARRAY_DISPLAY_LIMIT
- XSIM.RADIX
- XSIM.TIME_UNIT
- XSIM.TRACE_LIMIT

プロパティの現在の値を確認するには `report_property [current_project]` Tcl コマンドを使用し、プロパティ値を設定するには `set_property <property name> <property value> [current_project]` Tcl コマンドを使用します。たとえば、配列表示を 16 に制限するには、次のコマンドを使用します。

```
set_property xsim.array_display_limit 16 [current_project]
```

新規作成したシミュレーションまたはクリーンアップされたシミュレーションを起動すると、シミュレーション Tcl オブジェクトにはプロジェクトのプロパティが使用されます。プロパティを確認するには、次の Tcl コマンドを使用します。

```
report_property [current_sim]
```



重要: プロジェクト プロパティは、クリーンアップされた、または新しく作成されたシミュレーションにしか適用されません。特定の run タイプおよび `sim_1/behav` などのシミュレーション セットのシミュレーションを実行すると、別の設定ファイルが作成され、その後の実行ではそのファイルの設定が使用されます。プロジェクト プロパティへの変更は、そのシミュレーションには反映されません。プロジェクト プロパティは、シミュレーションをクリーンアップした場合、または設定ファイルを削除した場合にのみ適用されます。

Vivado シミュレータを使用したシミュレーション波形の解析

Vivado® シミュレータでは、波形を使用してデザインを解析し、コードをデバッグできます。シミュレータにより、[Objects] ウィンドウや [Scopes] ウィンドウなど、ワークスペースのほかのエリアにデザイン 信号データが自動的に入力されます。

シミュレーションは、通常シミュレーションをする HDL オブジェクトを定義するテストベンチで設定します。テストベンチの詳細は、『効率的なテストベンチの記述』(XAPP199)を参照してください。

Vivado シミュレータを起動すると、最上位 HDL オブジェクトを含む波形設定が表示されます。[Objects] ウィンドウおよび [Scopes] ウィンドウなどのワークスペースのほかのエリアにデザイン データが Vivado シミュレータにより自動的に入力されます。この後、別の HDL オブジェクトを追加したり、シミュレーションを実行したりできます。[波形設定と波形ウィンドウの使用](#)を参照してください。

波形設定と波形ウィンドウの使用

Vivado シミュレータでは、波形ウィンドウをカスタマイズできます。表示の現在の状態は、波形設定と呼ばれます。この設定を WCFG ファイルに保存して、後で使用できます。

波形設定には名前を付けることができます。デフォルト名は `untitled` です。名前は波形設定ウィンドウのタイトルバーに表示されます。波形設定を一度もファイルに保存していない場合は、名前は「untitled」になっています。

波形設定の新規作成

波形を表示するための新しい波形設定を作成するには、次の手順に従います。

1. [File] → [Simulation Waveform] → [New Configuration] をクリックします。

新しい波形ウィンドウが開き、「untitled」という名前の新しい波形設定が表示されます。Tcl コマンド:
`create_wave_config <waveform_name>`

2. [波形設定の HDL オブジェクト](#)に示す方法で HDL オブジェクトを波形設定に追加します。

波形の新規作成の詳細は、[第 4 章: Vivado シミュレータを使用したシミュレーション](#)を参照してください。複数波形に関する詳細は、[複数の波形設定の作成と使用](#)も参照してください。

WCFG ファイルを開く

シミュレーションで使用する WCFG ファイルを開くには、次の手順に従います。

1. [File] → [Simulation Waveform] → [Open Configuration] をクリックします。

[Open Waveform Configuration] ダイアログ ボックスが開きます。

2. WCFG ファイルを選択します。

注記: WCFG ファイルにスタティック シミュレーションの HDL デザイン階層にはない HDL オブジェクトへの参照が含まれる場合、Vivado シミュレータではこれらの HDL オブジェクトが無視され、読み込まれた波形設定から削除されます。

波形ウィンドウが開き、WCFG ファイルにリストされた波形オブジェクトの波形データが表示されます。

Tcl コマンド: `open_wave_config <waveform_name>`

波形設定の保存

波形設定を変更して WCFG ファイルに保存するには、[File]→[Simulation Waveform]→[Save Configuration As] をクリックし、波形設定の名前を指定します。

Tcl コマンド: `save_wave_config <waveform_name>`

前に保存したシミュレーション run を開く

前に保存したシミュレーションを Vivado Design Suite で開くには、スタンドアロン、インタラクティブ、プログラム手法の 3 つの方法があります。

スタンドアロン モード

次のコマンドを使用すると、Vivado の外部から WDB ファイルを開くことができます。

```
xsim <name>.wdb -gui
```



ヒント: `-view <WCFG file>` コマンドに `xsim` を追加すると、WDB ファイルと WCFG ファイルを一緒に開くことができます。

インタラクティブ

1. Vivado Design Suite プロジェクトを読み込んだら、[Flow > Open Static Simulation] をクリックし、実行済みシミュレーションの波形を含む WDB ファイルを選択します。



ヒント: スタティック シミュレーションは Vivado シミュレータのモードで、シミュレーションの実行からのデータの代わりに、WDB ファイルからのデータがウィンドウに表示されます。

1. Tcl コンソールに「`open_wave_database <name>.wdb`」と入力します。

プログラム手法

次の内容を含む Tcl ファイル (`design.tcl` など) を作成します。

```
current_fileset
open_wave_database <name>.wdb
```

このファイルを次のように実行します。

```
vivado -source design.tcl
```



重要: Vivado シミュレータでは、サポートされている OS で作成された WDB ファイルを開くことができます。Vivado Design Suite 2014.3 以降のバージョンで作成された WDB ファイルも開くことができます。Vivado Design Suite 2014.3 よりも前のバージョンで作成された WDB ファイルを Vivado シミュレータで開くことはできません。

シミュレーションを実行して HDL オブジェクトを波形ウィンドウで表示する場合、シミュレーションを実行すると、表示されている HDL オブジェクトの波形アクティビティを含む波形データベース (WDB) ファイルが作成されます。WDB ファイルには、シミュレーションされたデザインの HDL スコープとオブジェクトすべてに関する情報も含まれます。このモードでは、下位に制御するライブシミュレーション モデルがないので、run コマンドのようなシミュレーションを制御または監視するコマンドは使用できません。

ただし、波形および HDL デザイン階層を表示することはできます。



波形設定の HDL オブジェクト

HDL オブジェクトを波形設定に追加すると、波形ビューアーに HDL オブジェクトの波形オブジェクトが作成されます。波形オブジェクトは、関連付けられている HDL オブジェクトにリンクされますが、それぞれ別のものです。

同じ HDL オブジェクトから複数の波形オブジェクトを作成でき、各波形オブジェクトの表示プロパティを別々に設定できます。

たとえば、myBus という HDL オブジェクトから作成されたある波形オブジェクトを 16 進数で表示し、同じ myBus の別の波形オブジェクトを 10 進数で表示させることができます。

仕切り、グループ、仮想バスなどの波形オブジェクトも波形設定に表示できます。

HDL オブジェクトから作成された波形オブジェクトは、「デザイン波形オブジェクト」と呼ばれます。これらのオブジェクトは、対応するアイコン付きで表示されます。デザイン波形オブジェクトの場合、アイコンを見ると、そのオブジェクトがスカラー  なのか、Verilog ベクターや VHDL レコードなどの複合型  なのかがわかります。

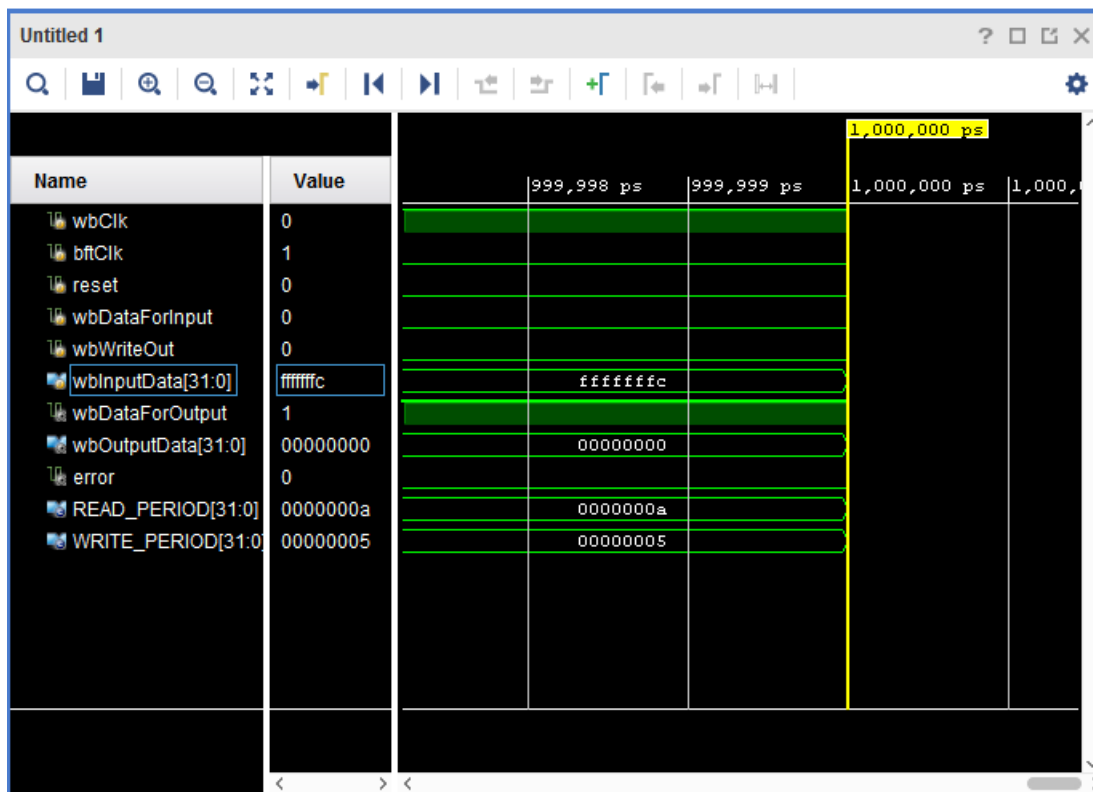


ヒント: [Objects] ウィンドウにデザイン波形オブジェクトの HDL オブジェクトを表示するには、デザイン波形オブジェクトの名前を右クリックし、[Show in Object Window] をクリックします。

次の図に、波形設定ウィンドウの HDL オブジェクトの例を示します。デザイン オブジェクトには、名前と値が表示されます。

- [Name]: デフォルトでは、HDL オブジェクトの名前が表示されます。表示されるのは名前だけで、オブジェクトの階層パスは表示されません。階層パスを含めた名前を表示したり、カスタム名を指定して表示することもできます。
- [Value]: 波形ウィンドウのメインカーソルに示されている時間におけるオブジェクトの値を表示します。値のフォーマットまたは基数は、同じ HDL オブジェクトにリンクされているほかのデザイン波形オブジェクトのフォーマットや、[Objects] ウィンドウおよびソースコードウィンドウに表示される値のフォーマットに関係なく変更できます。

図 23: 波形 HDL オブジェクト



[Scopes] ウィンドウには、選択したスコープに対して表示可能なすべての HDL オブジェクトを、波形ウィンドウに追加する機能があります。[Scopes] ウィンドウの使用方法是、[\[Scopes\] ウィンドウ](#)を参照してください。

基数

バスのデータ型を理解することは重要です。デジタルおよびアナログの波形を効果的に使用するには、基数設定とデータ型の関係を認識する必要があります。




重要: 基数を変更するウィンドウで基数設定を変更してください。[Objects] ウィンドウで基数を変更しても、波形ウィンドウまたは Tcl コンソールにはその変更は反映されません。たとえば、[Objects] ウィンドウで wbOutputData[31:0] というアイテムを [Signed Decimal] に変更しても、波形ウィンドウでは [Binary] のままです。

デフォルト基数の変更

デフォルトの波形基数は、基数が明示的に設定されていない波形オブジェクトすべての値の数値形式を制御します。デフォルトの波形基数は、[Hexadecimal] (16 進数) です。

デフォルトの波形基数を変更するには、次の手順に従います。

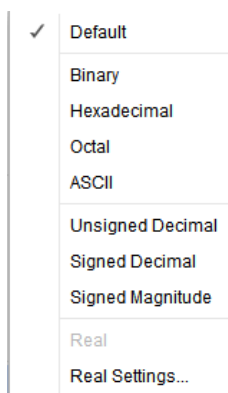
1. 波形ウィンドウの [Settings] ボタン  をクリックし、[Waveform Settings] を開きます。
2. [General] ビューで [Default Radix] ドロップダウン リストをクリックします。
3. ドロップダウン リストから基数を選択します。

オブジェクトごとの基数の変更

波形ウィンドウで波形オブジェクトの基数を変更するには、次の手順に従います。

1. 波形オブジェクトの名前を右クリックします。
2. [Radix] をクリックし、ドロップダウン リストからフォーマットを選択します。

- [Default] (デフォルト)
- [Binary] (2 進数)
- [Hexadecimal] (16 進数)
- [Octal] (8 進数)
- [ASCII]
- [Unsigned Decimal] (符号なし 10 進数)
- [Signed Decimal] (符号付き 10 進数)
- [Signed Magnitude] (符号付き大きさ)
- [Real] (実数)



注記: [Real] および [Real Settings] の使用方法は、[アナログ波形の使用](#)を参照してください。

3. Tcl コンソールから表示された値を数値の形式に変更するには、次を Tcl コマンドを入力します。

```
set_property radix <radix> <hdl_object>
```

<radix> には、bin、unsigned、hex、dec、ascii、または oct を指定します。<wave_object> には、add_wave コマンドで返されるオブジェクトを指定します。



ヒント: 波形ウィンドウで基数を変更しても、[Objects] ウィンドウにはその変更は反映されません。

波形のカスタマイズ

アナログ波形の使用

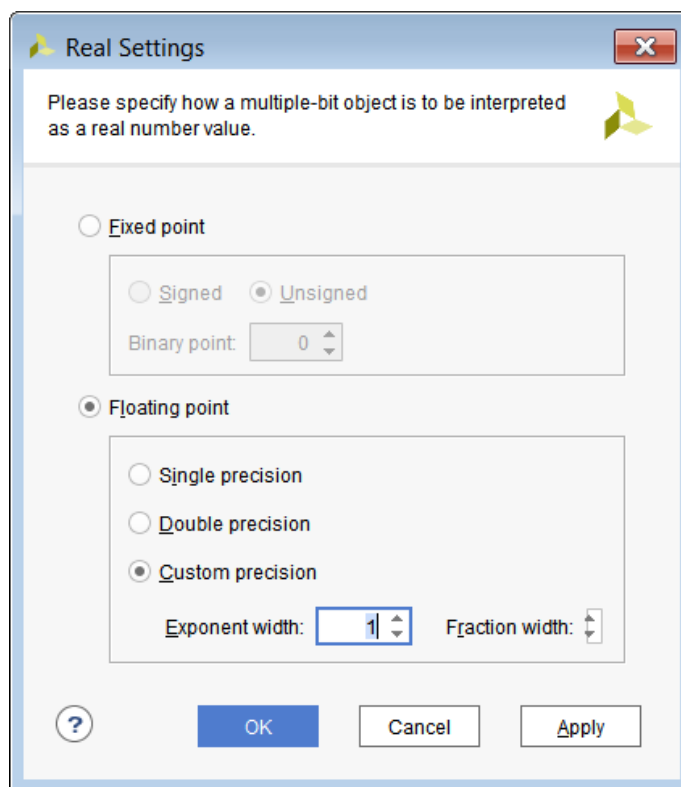
基数およびアナログ波形の使用

バスの値が数値として処理される方法は、バス波形オブジェクトの基数設定によって決まります。

- 2 進数、8 進数、16 進数、ASCII、および符号なしの 10 進数の基数を使用すると、バスの値が符号なしの整数として処理されます。
- バス内のビットのいずれかが 0 でも 1 でもない場合、バス全体の値は 0 と解釈されます。
- 符号付きの 10 進数および符号付き大きさ基数を使用すると、バスの値が符号付き整数として処理されます。
- 基数を実数に設定すると、[Real Settings] ダイアログ ボックスの設定に基づいて、バスの値が固定小数点または浮動小数点の実数として処理されます。

波形オブジェクトの基数を実数に設定するには、次の手順に従います。

1. 波形ウィンドウで HDL オブジェクトを右クリックし、ポップアップ メニューを開きます。
2. [Radix]→[Real Settings] をクリックして、次の図に示す [Real Settings] ダイアログ ボックスを開きます。



波形の基数を [Real] に設定して、オブジェクトの値を実数で表示できます。この基数を選択する前に、波形ビューアーでの値のビットの処理方法を指定する必要があります。

[Real Setting] ダイアログ ボックスのオプションは、次のとおりです。

- [Fixed Point]: 選択したバス波形オブジェクトのビットが固定小数点の符号付きまたは符号なし実数として処理されます。
- [Binary Point]: 2 進数小数点の右側のビット数を指定します。[Binary point] で指定した値が波形オブジェクトのビット幅よりも大きい場合、波形オブジェクトの値は固定小数点として処理できず、波形オブジェクトがデジタル波形で表示される場合はすべての値が <Bad Radix> と表示されます。アナログ波形として表示される場合、すべての値は 0 として処理されます。
- [Floating Point]: 選択したバス波形オブジェクトのビットが IEEE 浮動小数点の実数として処理されます。
注記: 単精度および倍精度 (および単/倍精度に設定されている値のカスタム精度) のみがサポートされています。
 その他の値は、[Fixed point] を使用した場合と同様 <Bad Radix> と表示されます。[Exponent Width] および [Fraction Width] を加算した値が波形オブジェクトのビット幅と一致する必要がある、一致しない場合は <Bad Radix> と表示されます。



ヒント: 行を区切る線が表示されていない場合、[Waveform Settings] スライドアウトでオンにできます。

波形のアナログ表示



重要: HDL バス オブジェクトをアナログ波形で表示して予測される波形を出力する場合、HDL オブジェクトのデータの性質と一致する基数を選択してください。次に例を示します。

- バスでエンコードされるデータが 2 の補数の符号付き整数の場合は、符号付きの基数を選択する必要があります。
- データが IEEE フォーマットでエンコードされる浮動小数点の場合は、基数に実数を選択する必要があります。

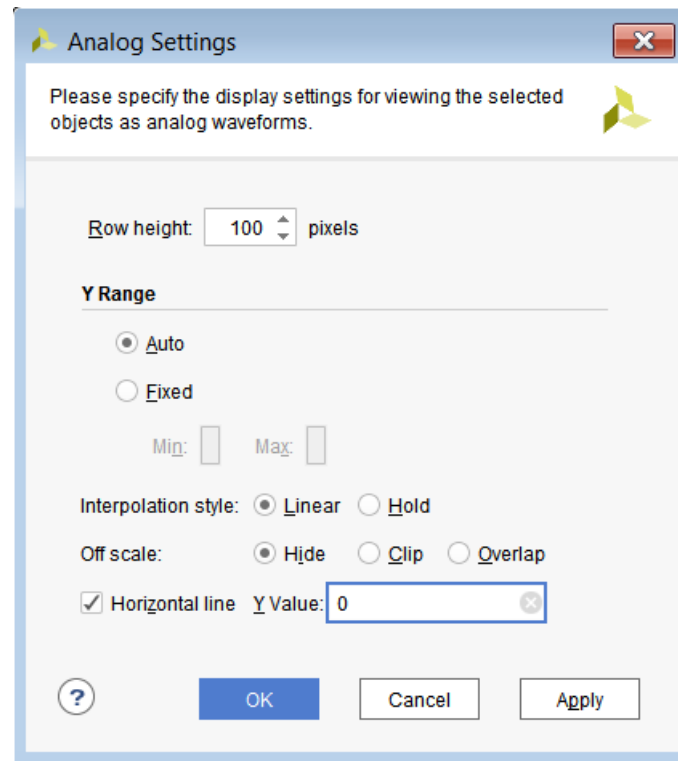
アナログ波形表示のカスタマイズ

アナログ波形の表示をカスタマイズするには、波形設定ウィンドウの [Name] 列で HDL オブジェクトを右クリックし、[Waveform Style] をクリックします。ポップアップメニューに次のオプションが表示されます。

- [Analog]: 波形をアナログに設定します。
- [Digital]: 波形オブジェクトをデジタルに設定します。
- [Analog Settings]: アナログ波形表示に関するオプションを設定する [Analog Settings] ダイアログ ボックス (次の図) が開きます。

波形ウィンドウでは、幅が 64 ビット以下のバスのアナログ波形のみを表示できます。

図 24: [Analog Settings] ダイアログ ボックス



[Analog Settings] ダイアログ ボックスのオプションの説明

- [Row Height]: 選択した波形オブジェクトの高さをピクセル数で指定します。行の高さを変更しても波形の垂直方向の表示域は変わりませんが、波形の高さの伸縮が変わります。

アナログとデジタルを切り替えるとき、行の高さはそれぞれに合った適切なデフォルトの高さに設定されます (デジタルの場合は 20、アナログの場合は 100)。



ヒント: 行インデックスが表示されていない場合は、波形ウィンドウの [Waveform Settings] スライドアウトでオンにしてください。オプション設定の変更方法については、[\[Waveform Settings\] スライドアウト](#)を参照してください。行を区切る線を波形名の左および下にドラッグして、行の高さを変更することもできます。

- [Y Range]: 波形エリアに表示される数値の範囲を指定します。
 - [Auto]: 表示されている時間の範囲の値が現在の範囲を超えたときに、表示範囲が拡大されます。
 - [Fixed]: 時間範囲を一定にします。
 - [Min]: 波形エリアの一番下に表示される値を指定します。
 - [Max]: 波形エリアの一番上に表示される値を指定します。

どちらの値も浮動小数点として指定できますが、波形オブジェクトの基数が整数の場合、値は整数に切り捨てられます。
- [Interpolation Style]: データ ポイントを接続するラインの描画方法を指定します。
 - [Linear]: 2 つのデータ ポイント間のラインを直線にします。
 - [Hold]: 2 つのデータ ポイントのうち、左のポイントから右のポイントの X 軸に向かって水平ラインを描画し、そのラインから右のポイントに向かって別のラインを L 字型に描画します。

- [Off Scale]: 波形エリアの Y 軸を超えた値をどのように描画するかを指定します。
 - [Hide]: 範囲外にある値を非表示にします。波形が波形エリアの上下の範囲外になると、値が範囲内に戻るまで非表示になります。
 - [Clip]: 値が範囲を超えると、範囲内に戻るまで波形エリアの上下境界線上に水平線として表示されます。
 - [Overlap]: 波形が波形エリアの範囲を超えてほかの波形と重なったとしても、波形ウィンドウの境界に達するまで値の位置に波形が描画されます。
 - [Horizontal Line]: 指定した値で水平方向のラインを描画するかどうか指定します。このチェック ボックスをオンにすると、Y 軸の [Y Value] で指定した値の位置に水平線が描画されます (指定した値が波形の Y 範囲内である場合)。
- [Min] および [Max] の場合と同様、Y 軸の値には浮動小数点値を指定できますが、選択した波形オブジェクトの基数が整数の場合は、整数値に切り捨てられます。

波形オブジェクトの命名

オブジェクト名を変更および表示するオプションや、名前の表示を変更するオプションがあります。

オブジェクト名の変更

デザイン波形オブジェクト、仕切り、グループ、仮想バスなどの波形設定に含まれる波形オブジェクトの名前は変更できます。

1. [Name] 列でオブジェクト名を選択します。
 2. 右クリックして [Rename] をクリックします。
[Rename] ダイアログ ボックスが開きます。
 3. [Rename] ダイアログ ボックスに新しい名前を入力し、[OK] をクリックします。
- 注記:** 波形設定のデザイン波形オブジェクトの名前を変更しても、その下位の HDL オブジェクトの名前は変わりません。

オブジェクト名表示の変更

完全な階層名 (long name)、信号またはバス名のみ (short name)、あるいは各デザイン波形オブジェクトのカスタム名を表示できます。オブジェクト名は、波形設定の [Name] 列に表示されます。名前が非表示になっている場合は、次の操作を実行します。

1. 名前全体が表示されるように [Name] 列の幅を調整します。
2. [Name] 列でスクロール バーを使用して名前を表示します。

表示名を変更するには、次の手順に従います。

1. 信号名またはバス名を 1 つ以上選択します。複数の信号を選択する場合は、Shift キーまたは Ctrl キーを押しながらクリックします。
2. 右クリックして [Name] をクリックします。ポップアップ メニューに次のオプションが表示されます。
 - [Long]: デザイン オブジェクトのフル階層名を表示します。
 - [Short]: 信号名またはバス名のみを表示します。
 - [Custom]: オブジェクトのカスタム名を表示します。 [オブジェクト名表示の変更](#) を参照してください。



ヒント: 波形オブジェクトの名前を変更すると、名前表示モードが [Custom] に変わります。元の表示モードに戻すには、表示モードを [Long] または [Short] に戻す必要があります。[Long] および [Short] の名前はデザイン波形オブジェクトに対してのみ意味があります。その他のオブジェクト (仕切り、グループ、仮想バス) はデフォルトで [Custom] 名で表示され、[Long] および [Short] 名に対する ID 文字列が表示されます。

バス ビット順の反転

波形設定でバス ビット順を反転すると、バス値の表示を MSB から表示するか (ビッグ エンディアン)、LSB から表示するか (リトル エンディアン) を切り替えることができます。

ビット順序を逆にするには、次の手順に従います。

1. バスを選択します。
2. 右クリックし、[Reverse Bit Order] をクリックします。

これでバス ビットの順序が逆になります。[Reverse Bit Order] コマンドの横にチェック マークが表示され、適用されていることが示されます。



重要: [Reverse Bit Order] コマンドはバスに表示されている値に対してのみ実行できます。このコマンドを実行しても、バス波形オブジェクトを拡張するときバスの下に表示されるバス エLEMENT のリストの順序は逆になりません。



ヒント: バスの [Long] 名および [Short] 名に表示されるインデックス範囲は、バス エLEMENT のビット順を示します。たとえば、bus[0:7] バスに [Reverse Bit Order] を適用すると、bus[7:0] と表示されます。

SystemVerilog の列挙フォーマットの変更

SystemVerilog 列挙型は数値を持つ HDL オブジェクトで、特定の値を表すためにテキスト ラベルが定義されます。たとえば、値 1 を表すために LABEL1、値 5 を表すために LABEL2 と定義するなどです。コンテキスト メニューの [Show As Enumeration] を使用すると、列挙値を指定したラベルまたは数値のどちらで表示するかを指定できます。上の例の場合、[Show As Enumeration] をオンにすると、値 5 は LABEL2 と表示されます。オフにすると、[Radix] メニューで列挙型にどんな基数を設定したとしても、値 5 が表示されます。

ラベルを使用して列挙型を表示する手順は、次のとおりです。

1. 列挙型を選択します。
2. 右クリックして [Display As Enumeration] をオンにします。

列挙型を数値で表示する手順は、次のとおりです。

1. 列挙型を選択します。
2. 右クリックして [Display As Enumeration] をオフにします。

注記: 定義されたラベルがない列挙値の場合は、[Display As Enumeration] 設定に関係なく、常に数字で表示されます。[Display As Enumeration] は、SystemVerilog 列挙オブジェクトに対してのみイネーブルになります。

波形表示の制御

波形表示は、次を使用して制御できます。

- 波形ウィンドウの [Name]、[Value]、波形列のサイズ調整ハンドル
- マウス ホイールを使用したスクロールの組み合わせ
- 波形ウィンドウのサイドバーにあるズーム ボタン
- マウス ホイールを使用したズームの組み合わせ
- Vivado IDE の Y 軸のズーム機能
- Vivado シミュレーションの X 軸のズーム機能。マウス ボタンを使用したズーム表示については、『Vivado Design Suite ユーザー ガイド: Vivado IDE の使用』 (UG893) を参照してください。

注記: ほかの Vivado Design Suite のグラフィック ウィンドウとは異なり、波形ウィンドウの拡大/縮小は X (時間) 軸に適用されます。このため、ウィンドウを拡大/縮小する範囲を指定する [Zoom Range X] が、ほかの Vivado Design Suite ウィンドウの [Zoom to Area] の代わりに使用されます。



ヒント: WCFG ファイルを保存すると、波形ウィンドウの設定と、波形オブジェクトおよびマーカーが保存されます。波形ウィンドウの設定は、名前、値の列の幅、ズーム レベル、スクロール位置、グループやバスの拡張、メインカーソルの位置などです。

列サイズ調節ハンドルの使用

名前または値の列幅を変更するには、列の右側の縦線にマウスを置き、カーソルの形が変わったら、マウスを左右に動かします。

注記: [Name] 列の幅を広げるには、[Value] 列の幅が既に最小である場合は、まず [Value] 列の幅を広げる必要があります。

マウス ホイールを使用したスクロール

波形ウィンドウ内でクリックすると、マウス ホイールを使用して上下にスクロールできます。Shift キーを押しながらマウス ホイールを使用すると、波形を左右にスクロールできます。

ズーム ボタンの使用

波形ウィンドウには、拡大、縮小、全体表示のズーム ボタンがあり、これらを使用して波形設定を拡大および縮小できます。



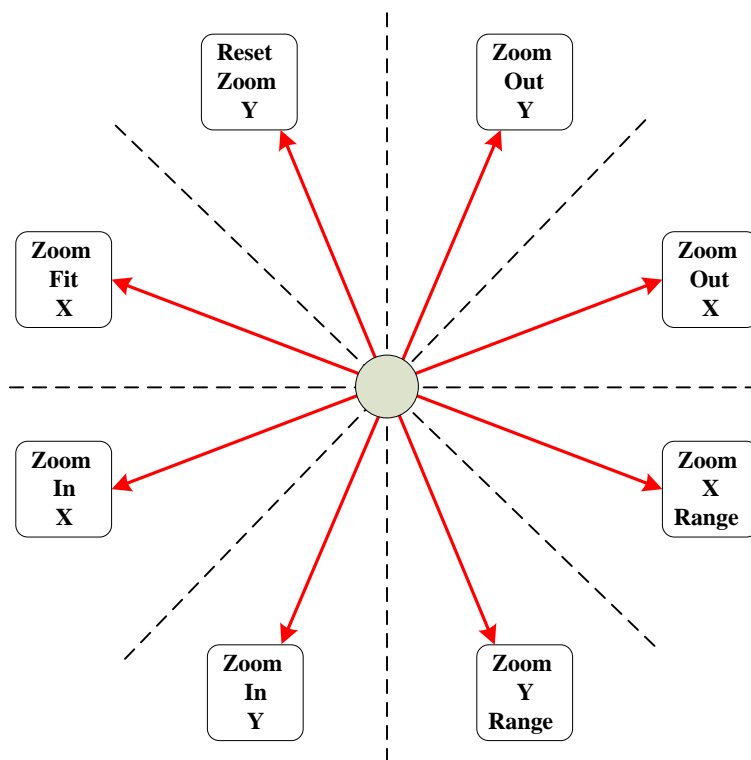
マウス ホイールを使用したズーム

波形エリアをクリックして Ctrl キーを押しながらマウス ホイールを使用すると、オシロスコープのダイヤルを操作するように、表示を拡大/縮小できます。

アナログ波形の Y 軸方向のズーム

X 軸方向のズームでサポートされている機能に加え、アナログ波形の場合は、次の図に示す追加のズーム機能があります。

図 25: アナログ ズームのオプション



ズーム機能を使用するには、マウスの左ボタンを押したまま、図で示されている方向にマウスをドラッグします。この図の中央がマウスの位置です。

次の追加ズーム機能があります。

- [Zoom Out Y]: 始点からマウス ボタンを放した位置までの距離により決定される 2 のべき乗分 Y 軸方向に縮小します。始点のマウス位置の Y 値をそのまま維持してズームが実行されます。
- [Zoom Y Range]: 垂直方向に線を描き、マウス ボタンを離れた位置までの Y 軸の範囲を表示します。
- [Zoom In Y]: 始点からマウス ボタンを放した位置までの距離により決定される 2 のべき乗分 Y 軸方向に拡大します。始点のマウス位置の Y 値をそのまま維持してズームが実行されます。
- [Reset Zoom Y]: Y の範囲を波形ウィンドウに現在表示されている値にリセットし、Y の範囲モードを [Auto] に設定します。

Y 軸の方向のズーム機能はすべて Y の範囲のアナログ値を設定します。[Reset Zoom Y] は Y の範囲を [Auto] に設定しますが、ほかのズーム機能は [Fixed] に設定します。

[Waveform Settings] スライドアウト


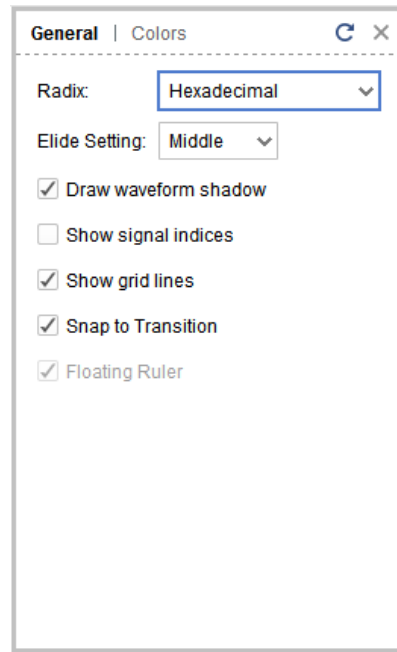
[Settings] ボタン  をクリックすると、次の図に示す [Waveforms Settings] スライドアウトが開きます。

図 26: 波形設定



[General] タブでは、次を設定できます。

- [Radix]: 新しく作成した波形オブジェクトに使用する数値形式を設定します。
- [Elide Setting]: 波形ウィンドウで長い信号名の省略方法を指定します。
 - [Left]: 長い信号名の左端を切り捨てます。
 - [Right]: 長い信号名の右端を切り捨てます。
 - [Middle]: 長い信号名の真ん中を切り捨て、左右両端を残します。
- [Draw Waveform Shadow]: 影付きの波形を作成します。
- [Show signal indices]: 各波形オブジェクト名の左に行番号を表示します。行番号を区切る線をドラッグすると、波形オブジェクトの高さを変更できます。
- [Show grid lines]: グリッド線を表示します。
- [Snap to Transition]: カーソルおよびマーカーをドラッグしたときに、カーソルまたはマーカーがマウスに一番近い遷移部分に配置されるよう指定します。詳細は、[カーソルの使用](#) を参照してください。
- [Floating Ruler]: 2 番目のカーソルが表示されたとき、またはマーカーが選択されたときに、フロート ルーラーを表示します。詳細は、[フロート ルーラーの使用](#) を参照してください。



ヒント: [Floating Ruler] オプションがオフの場合、波形ウィンドウで Shift キーを押しながらクリックすると、2 番目のカーソルが表示され、[Floating Ruler] オプションがオンになります。

- [Colors] タブ: 波形ウィンドウ内のアイテムの色を設定します。

時間スケールの表示の変更

ルーラーを右クリックして、時間スケールを表示します。このメニューから、時間スケールの時間値の表示方法を選択できます。

時間スケールには次のオプションがあります。

- [Auto]: 波形ウィンドウのズーム レベルに適切な時間単位が自動的に選択されます。
- [Default]: HDL デザインがコンパイルされたときに決定されたシミュレーションの精度に対応する時間単位を表示します。
- [Samples]: 時間を秒の分数ではなく、個別のサンプル値で表示します (HDL シミュレーションでは使用不可)。
- [User]: ユーザー指定の時間単位を表示します (HDL シミュレーションでは使用不可)。
- [fs]: 時間単位をフェムト秒で表示します。
- [ps]: 時間単位をマイクロ秒で表示します。
- [ns]: 時間単位をナノ秒で表示します。
- [us]: 時間単位をマイクロ秒で表示します。
- [ms]: 時間単位をミリ秒で表示します。
- [s]: 時間単位を秒で表示します。

波形の分類

ここでは、波形内の情報を分類するオプションについて説明します。

信号およびオブジェクトのグループ

関連している波形オブジェクトを見やすくするため、グループにまとめることができます。グループに含まれているオブジェクトは展開または非展開にできます。グループ自体は波形データを表示しませんが、その内容の表示/非表示を切り替えることができます。グループは追加、変更、削除できます。

グループを追加するには、次の手順に従います。


1. 波形ウィンドウで、グループに追加する波形オブジェクトを 1 つまたは複数選択します。

注記: グループには、仕切り、仮想バス、ほかのグループを含めることができます。

2. 右クリックして [New Group] をクリックします。

これにより、選択した波形オブジェクトを含むグループが波形設定に追加されます。

Tcl コンソールで `add_wave_group` を入力すると、新しいグループが追加されます。

グループは、[Group] アイコン  で表されます。HDL オブジェクトをドラッグ アンド ドロップして、グループに信号やバスを追加することもできます。

波形設定ファイルを保存すると、新しいグループとそれに含まれる波形オブジェクトも保存されます。

グループは、次の方法で移動または削除できます。

- グループを移動するには、[Name] 列のグループ名を別の位置にドラッグ アンド ドロップします。
- グループを削除するには、グループを選択して右クリックして [Ungroup] をクリックします。グループに含まれていた波形オブジェクトは、波形設定の一番上に配置されます。

グループ名も変更できます。詳細は、[オブジェクト名表示の変更](#)を参照してください。



注意: Delete キーを押すと、選択されているグループと、それに含まれている波形オブジェクトが波形設定から削除されます。

仕切りの使用

信号やオブジェクトを見やすくするため、HDL オブジェクトの間に仕切りを作成できます。波形設定に仕切りを追加するには、次の手順に従います。

1. 波形ウィンドウの [Name] 列で信号をクリックします。仕切りはその信号の下に追加されます。
2. 右クリックし、[New Divider] をクリックします。

新しい区切りマークは、波形コンフィギュレーション ファイルを保存したときに保存されます。

Tcl コマンド: `add_wave_divider`

仕切りは、次の方法で移動または削除できます。

- 仕切りを移動するには、名前を別の位置にドラッグ アンド ドロップします。
- 仕切りを削除するには、Delete キーを押すか、または右クリックして [Delete] をクリックします。

仕切りの名前も変更できます。詳細は、[オブジェクト名表示の変更](#)を参照してください。


仮想バスの定義

波形設定に仮想バスを定義できます。仮想バスは、論理スカラーおよびベクターをグループにまとめたものです。

仮想バスはバス波形を表示します。バス波形の値は、仮想バスの下に表示される順番で追加されたスカラーおよび配列から対応する値を取り出して、1 次元のベクターに平坦化することにより作成されます。

仮想バスを追加するには、次の手順に従います。

1. 波形設定で、仮想バスに追加する波形オブジェクトを 1 つまたは複数選択します。
2. 右クリックして [New Virtual Bus] をクリックします。

仮想バスは、[Virtual Bus] アイコン  で表されます。

Tcl コマンド: `add_wave_virtual_bus`

信号名やバス名をドラッグ アンド ドロップすると、仮想バスに論理スカラーおよび配列を移動できます。

波形設定ファイルを保存すると、新しい仮想バスとそれに含まれるものも保存されます。また、仮想バスの名前を波形の別位置にドラッグ アンド ドロップして移動できます。

仮想バスの名前を変更するには、[オブジェクト名表示の変更](#)を参照してください。

仮想バスを削除し、その内容をグループ解除するには、仮想バスを選択して右クリックし、[Ungroup] をクリックします。



注意: Delete キーを押すと、仮想バスと、それに含まれている HDL オブジェクトが波形設定から削除されます。

波形の解析

このセクションでは、波形内のデータを解析する機能について説明します。

カーソルの使用

カーソルは一時的なタイム マーカーで、2 つの波形エッジ間の時間を計測するため、頻繁に移動させることができます。

メイン カーソルと 2 番目のカーソル

波形ウィンドウを 1 回クリックすると、メイン カーソルが配置されます。

2 番目のカーソルを配置するには、Ctrl キーを押しながらクリックし、マウスを左または右にドラッグします。カーソルの上に位置を示すフラグが表示されます。または、Shift キーを押したまま波形のどこかをクリックします。

2 番目のカーソルがオンになっていない場合、この操作により 2 番目のカーソルがメイン カーソルの現在の位置に設定され、メイン カーソルがクリックした位置に配置されます。

注記: 2 番目のカーソルの位置を保持しながらメイン カーソルの位置を変更するには、Shift キーを押しながらクリックします。2 つ目のカーソルをドラッグして配置する場合、最小間隔以上ドラッグしないと 2 つ目のカーソルは表示されません。

カーソルの移動

カーソルを移動するには、ポインターが手のひらのマークになるまでマウスを動かし、クリックして次の位置までカーソルをドラッグします。

カーソルをドラッグするとき、[Snap to Transition] がオンになっていると (デフォルト)、白抜きの丸、または中が塗りつぶされた丸が表示されます。

- 白抜きの丸 ○ は、選択した信号の波形の遷移間にあることを示します。
- 中が塗りつぶされた丸 ● は、カーソルがマウスの下またはマーカーの波形の遷移にロックされていることを示します。

カーソル、マーカー、フロートしているルーラー以外の場所をクリックすると、2 番目のカーソルは非表示になります。



次または前の遷移の検索

波形ウィンドウのバーにあるボタンを使用して、メイン カーソルを、選択されている波形の次の遷移または前の遷移に移動できます。

メイン カーソルを波形の次または前の遷移に移動するには、次を実行します。

1. 波形の波形オブジェクト名をクリックしてアクティブにします。

波形オブジェクトが選択され、オブジェクトの波形が通常よりも太い線が表示されます。

2. ツールバーの [Next Transition] または [Previous Transition] ボタン   をクリックするか、キーボードの右方向キーまたは左方向キーを使用して、次または前の遷移に移動します。




ヒント: 複数の波形オブジェクトを一緒に選択すると、それらの波形の一番近い遷移に移動します。

マーカーの使用

波形内の重要イベントを恒久的にマークする必要がある場合はマーカーを使用します。マーカーを使用すると、マーカーが付けられたイベントに関連した時間を計測できます。

マーカーは、次のように追加、移動、削除できます。

- 波形設定のメイン カーソルの位置にマーカーを追加します。
 1. 波形ウィンドウの時間または遷移をクリックして、マーカーを追加する時間の箇所にメイン カーソルを置きます。
 2. 右クリックして [Markers]→[ Add Marker] をクリックします。

カーソル位置にマーカーが配置されます。マーカーがその位置に既にある場合は、若干オフセットされます。マーカーの時間が上部に表示されます。

新しい波形マーカーを作成するには、次の Tcl コマンドを使用します。

```
add_wave_marker <time> <timeunit> -name <name of the marker> -into
<wcfg file>
```

- マーカーを波形ウィンドウの別の位置に移動するには、ドラッグ アンド ドロップします。マーカー ラベル (マーカー上部またはマーカー線) をクリックしてドラッグします。
 - 波形ウィンドウでマーカーをドラッグする際、[Waveform Settings] スライドアウトで [Snap to Transition] がオンになっていると (デフォルト)、白抜きの丸または中が塗りつぶされた丸が表示されます。
 - 中が塗りつぶされている丸 ● は、選択した信号の波形の遷移点、または別のマーカー上であることを示します。
 - マーカーの場合は、丸は白く塗りつぶされています。
 - 白抜きの丸 ○ は、マーカーがマウスの下またはマーカーの波形の遷移にロックされていることを示します。

新しい位置にマーカーをドロップするには、マウスのボタンを放します。

- 1 つのコマンドでマーカーを 1 つ、またはすべて削除できます。マーカーを右クリックして、次のいずれかの操作を実行します。
 - マーカーを 1 つ削除するには、ポップアップ メニューから [Delete Marker] をクリックします。
 - マーカーをすべて削除するには、ポップアップ メニューから [Delete All Markers] をクリックします。

または、Delete キーを使用して選択したマーカーを削除することもできます。

コマンドの使用方法は、Vivado Design Suite のヘルプ (-help) または『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。

フロート ルーラーの使用

フロート ルーラーを使用すると、波形ウィンドウの上部にある標準ルーラーに表示される絶対シミュレーション時間ではなく、別の時間基準を使用して時間を計測できます。

フロート ルーラーの表示/非表示は切り替えることができ、波形ウィンドウで垂直方向にドラッグして位置を変更できます。このルーラーの時間基準 (時間 0) は、2 番目のカーソルか、2 番目のカーソルがない場合は選択されたマーカーです。

フロート ルーラーは、2 番目のカーソルまたはマーカーがある場合にのみ表示されます。

1. ルーラーの表示/非表示を切り替えるには、次のいずれかを実行します。

- 2 番目のカーソルを配置します。
- マーカーを選択します。

2. [Waveform Settings] スライドアウトで [Floating Ruler] をオンにします。

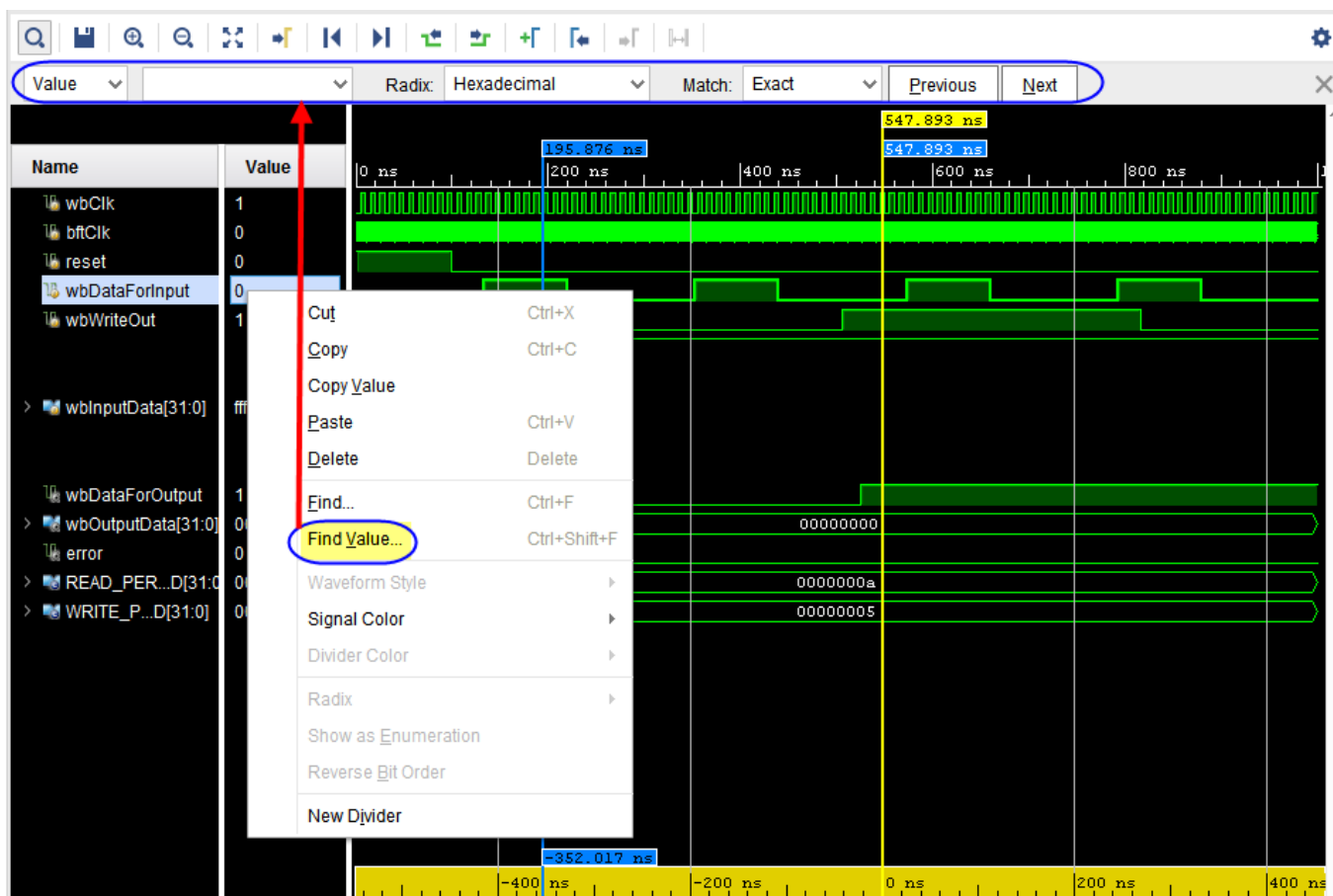
この操作は初回のみ必要です。それ以降は、フロート ルーラーは 2 番目のカーソルを置くか、またはマーカーを選択すると表示されます。

フロート ルーラーを非表示にするには、[Floating Ruler] をオフにします。

波形設定での値の検索

[Find] ツールバーを使用すると、1 つまたは複数の波形で特定の値を検索できます。完全に一致する値 (23FF など) またはあるパターンに一致する値 (最初の 2 桁が 23 で 4 桁目が F など) を検索できます。

図 27: [Find] ツールバーの [Value] オプション





重要: この検索機能では、logic 型のスカラーおよびベクター (1-D) 波形オブジェクトのみがサポートされます。logic 型には、Verilog/SystemVerilog の 2 値および 4 値型、VHDL の bit および std_logic 型が含まれます。

検索を実行するには、次の手順に従います。

1. [Name] 列で、デザイン波形オブジェクト (波形を持つ波形オブジェクト) を 1 つまたは複数選択します。
2. [Name] 列または [Value] 列で選択した波形オブジェクトを右クリックして [Find Value] をクリックし、[Find] ツールバーを表示します。
3. [Find] ツールバーで、[Radix] ドロップダウン リストから検索する値の基数を選択します。検索機能では、次の基数がサポートされます。
 - [Binary] (2 進数)
 - [Hexadecimal] (16 進数)
 - [Octal] (8 進数)
 - [Unsigned Decimal] (符号なし 10 進数)
 - [Signed Decimal] (符号付き 10 進数)
4. [Find] ツールバーの空のテキスト ボックスに、選択した基数で有効な値のパターンを入力します。有効な値には、数値、VHDL MVL 9 リテラル (U、X、0、1、Z、W、L、H、-)、および Verilog リテラル (0、1、x、z) があります。

注記: 無効な値を入力すると、テキスト ボックスが赤くなり、ツール ヒントにエラー メッセージが表示されます。有効な値は、基数によって異なります。たとえば基数として [Octal] を選択した場合、有効な値は 0 ~ 7 です。基数として [Hexadecimal] を選択した場合、有効な値は 0 ~ 9 および A ~ F (または a ~ f) です。任意の値を指定するため、ピリオド (.) を使用できます。たとえば、8 進数パターン「12.4」は 1234、1204 などを検索します。

5. [Match] ドロップダウン リストから次のいずれかの一致スタイルを選択します。
 - [Exact]: テキスト ボックスに入力した値と桁数も含め完全に一致する値のみを検索します。たとえば、値パターン「1234」は波形内の 1234 を検索しますが、123 や 12345 などは検索しません。



ヒント: [Exact] 一致スタイルでは、冒頭の 0 は省略できます。たとえば、波形で値 0023 を検索するには、0023 と入力することもできますが、単に 23 とも入力できます。

- [Beginning]: テキスト ボックスに入力した値と冒頭部分が一致する値を検索します。たとえば、値パターン「1234」は波形内の 1234 および 12345 を検索しますが、1235 や 123 など検索しません。このオプションは、基数が [Binary]、[Octal]、および [Hexadecimal] の場合にのみ設定可能です。
- [End]: テキスト ボックスに入力した値と末尾部分が一致する値を検索します。たとえば、値パターン「1234」は波形内の 1234 および 91234 を検索しますが、1235 や 234 など検索しません。このオプションは、基数が [Binary]、[Octal]、および [Hexadecimal] の場合にのみ設定可能です。
- [Next] ボタンをクリックするか Enter キーを押してメイン カーソルを次の一致に移動するか、[Previous] ボタンをクリックしてメイン カーソルを前の一致に移動します。複数の波形オブジェクトを選択している場合、カーソルは選択している波形オブジェクトのいずれかの一番近い一致に移動します。



ヒント: 指定した方向に一致がない場合、カーソルは移動せず、ツールバーの右の方に値が見つからないことを示すメッセージが表示されます。

AXI インターフェイス トランザクションの解析

Vivado IP インテグレーターを使用してデザインをブロック デザインとして作成した場合、Vivado シミュレータを起動したときに、Vivado によりデザインから AMBA® AXI インターフェイスが Vivado シミュレータに自動的にインポートされ、プロトコル インスタンスとして波形ウィンドウに表示されます。AXI インターフェイスのプロトコル インスタンスが波形ウィンドウに追加されると、シミュレーション中にそのインターフェイス上で発生するデータ トランザクションが表示されます。

プロトコル インスタンスの理解

AXI インターフェイスは、『AMBA® AXI and ACE Protocol Specification』および『AMBA® 4 AXI4-Stream Protocol Specification』で Arm® 社により定義されている論理信号の標準セットで構成されています。これらの信号は、仕様で説明されている論理イベントとしてコード記述されたデータ トランザクションを伝搬します。Vivado シミュレータでは、これらの信号を波形ビューアーで直接表示できますが、信号からだけではどのようなトランザクションが発生しているのかを判断するのが困難な場合があります。

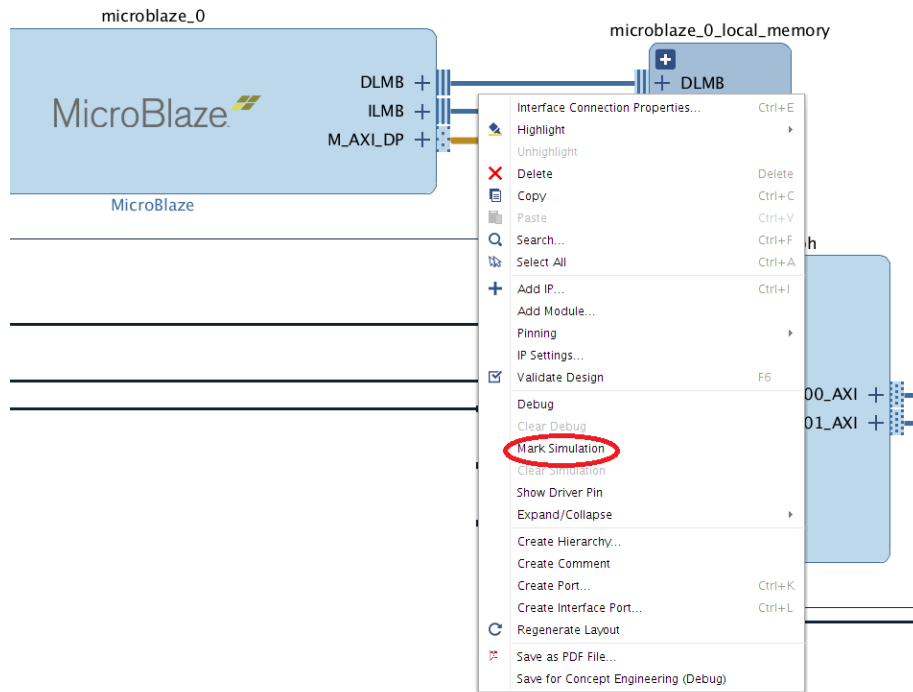
トランザクションを表示しやすくするため、Vivado シミュレータには、信号アクティビティを解析してアクティビティをトランザクション レベルで示す新しい信号を生成する機能があります。このプロセスは、プロトコル解析のと呼ばれます。Vivado シミュレータは、各 AXI インターフェイスに対し、AXI インターフェイスとプロトコル解析の入力および出力を表す「プロトコル インスタンス」という新しいデザイン オブジェクトを作成します。プロトコル インスタンスは通常、入力信号と同じスコープに配置されます。

AXI インターフェイスを Vivado シミュレータで表示するよう IP インテグレーターでマーク

Vivado IP インテグレーターには、AXI インターフェイスを特定して、ブロック デザイン ウィンドウから Vivado シミュレータの波形ビューアーで直接表示する機能があります。AXI インターフェイスを Vivado シミュレータで表示するようマークするには、次の手順に従います。

1. 表示する AXI インターフェイスを見つけます。
2. 対応するネット接続 (次の図にオレンジ色で示されている線) を右クリックします。
3. [Mark Simulation] をクリックします。
注記: [Mark Simulation] オプションは、AXI インターフェイスにのみ適用できます。
4. 手順 1 ~ 3 を繰り返し、表示するインターフェイスをすべてマークします。
5. [Clear Simulation] オプションをクリックし、AXI インターフェイスのマークをクリアします。
注記: [Clear Simulation] オプションは、AXI インターフェイスにのみ適用できます。
6. Vivado シミュレータを起動します。ブロック デザインを保存するかどうかを尋ねるメッセージが表示されたら、保存します。

Vivado シミュレータが起動すると、波形ウィンドウにマークしたインターフェイスが表示されます。Vivado プロジェクトを波形設定を自動的に開くようカスタマイズしている場合は、波形設定にマークしたインターフェイスが追加されます。Vivado プロジェクトを波形設定を開くようカスタマイズしていない場合は、Vivado シミュレータによりデフォルトの波形設定が作成され、通常の最上位 HDL 信号リストの代わりにマークしたインターフェイスが追加されます。



注意: AXI インターコネクト内部の AXI インターフェイスには、インターコネクトの設定によって、波形ビューアーに正しく表示されないものがあります。インターコネクトの境界にあるインターフェイスのみをマークすることをお勧めします。

Vivado シミュレータでのプロトコル インスタンスの検索

Vivado シミュレータが起動すると、デザインとその入力ファイルがスキャンされ、プロトコル インスタンスが検索されます。スキャンの結果 (次の図を参照) は、シミュレータ出力の上部にある Tcl コンソールに表示されます。Tcl コンソールからプロトコル インスタンス パスをコピーし、Tcl コマンドに貼り付けることができます。

図 28: Tcl コンソールに表示されたプロトコル インスタンス

```
time resolution is 1 ps
INFO: [Wavedata 42-565] Reading protoinst file protoinst_files/base_mb.protoinst
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//axi_gpio_0/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//axi_uartlite_0/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0/M_AXI_DP
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/M00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/M01_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/S00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m00_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m00_couplers/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m01_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m01_couplers/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/s00_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/s00_couplers/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/M00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/M01_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/S00_AXI
source system_th.tcl
```

[Objects] ウィンドウでのプロトコル インスタンスの検索

プロトコル インスタンス オブジェクトは、[Objects] ウィンドウの対応する AXI インターフェイス信号が含まれるスコープに表示されます。[Scope] ウィンドウでプロトコル インスタンスを検索するには、次の手順を実行します。

1. [Scope] ウィンドウで AXI インターフェイス信号を含むスコープを選択します。

注記: スコープ階層は、ブロック デザインとおおよそ一致しています。

2. ブロック デザインのブロックの AXI インターフェイスに対応するプロトコル インスタンスを見つけます。ブロックのインスタンスと名前が同じスコープを選択します。

[Objects] ウィンドウでプロトコル インスタンスを検索するには、次の手順を実行します。

1. [Objects] ウィンドウをスクロールしてリストの一番下に移動します。
2. ブロック デザインの AXI インターフェイスと名前が同じプロトコル インスタンスを見つけます。通常、AXI ポート名には接尾辞として `_AXI` が付いており、`M_AXI` または `S_AXI` という名前であることが多いです。



ヒント: プロトコル インスタンスには、内部信号のポート モードがあります。[Objects] ウィンドウでプロトコル インスタンスを検索しやすくするには、内部信号オブジェクト以外のすべてのオブジェクトを非表示にします。[Settings] ボタンをクリックし、[Select All] をオフにして、[Internal Signal] をオンにします。[Objects] ウィンドウを復元するには、[Select All] をオンにします。

Tcl コマンドを使用したプロトコル インスタンスの検索

プロトコル インスタンス オブジェクトでは、Tcl `type` フィールドが `proto_inst` に設定されています。指定のスコープ内またはその下にあるプロトコル インスタンスを検索するには、`get_objects` Tcl コマンドを使用できます。

デザイン内のすべてのプロトコル インスタンスを検索するコマンド:

```
get_objects /* -r -filter {type==proto_inst}
```

スコープ内のすべてのプロトコル インスタンスを検索するコマンド:

```
get_objects <Design scope hierarchy>/* -filter {type==proto_inst}
```

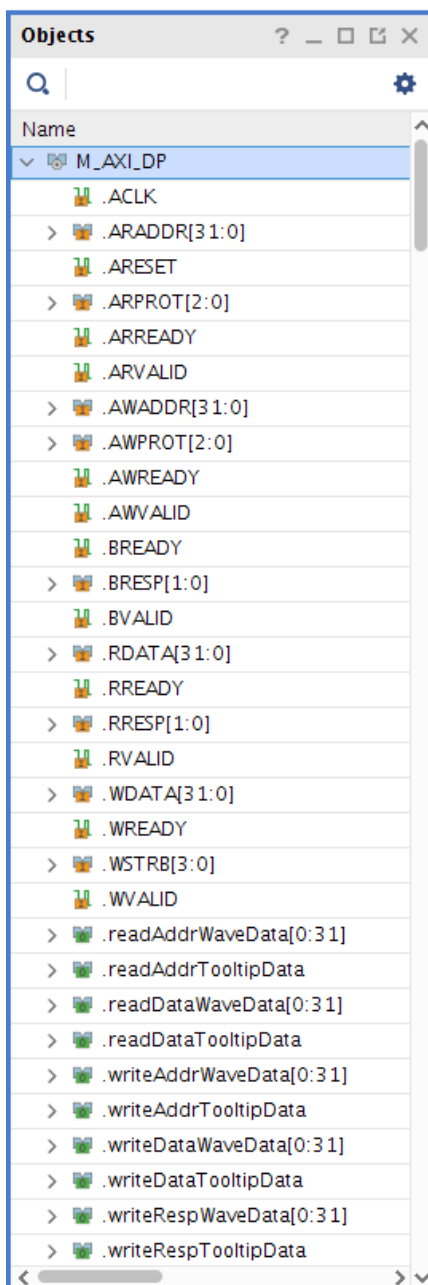
スコープ内またはその下にあるすべてのプロトコル インスタンスを検索するコマンド:

```
get_objects /system_tb/base_mb_wrapper/base_mb_i/* -r -filter {type==proto_inst}
```

[Objects] ウィンドウのプロトコル インスタンス

[Objects] ウィンドウでは、プロトコル インスタンスは、ブロック デザインに含まれる AXI インターフェイスのポート名と同じ名前の 1 つの集合デザイン オブジェクトとして表示されます。左側の矢印をクリックして、プロトコル インスタンスのプロトコル解析の入力および出力を表示します。次の図に、プロトコル インスタンス `M_AXI_DP` の例を示します。

図 29: [Objects] ウィンドウのプロトコル インスタンス

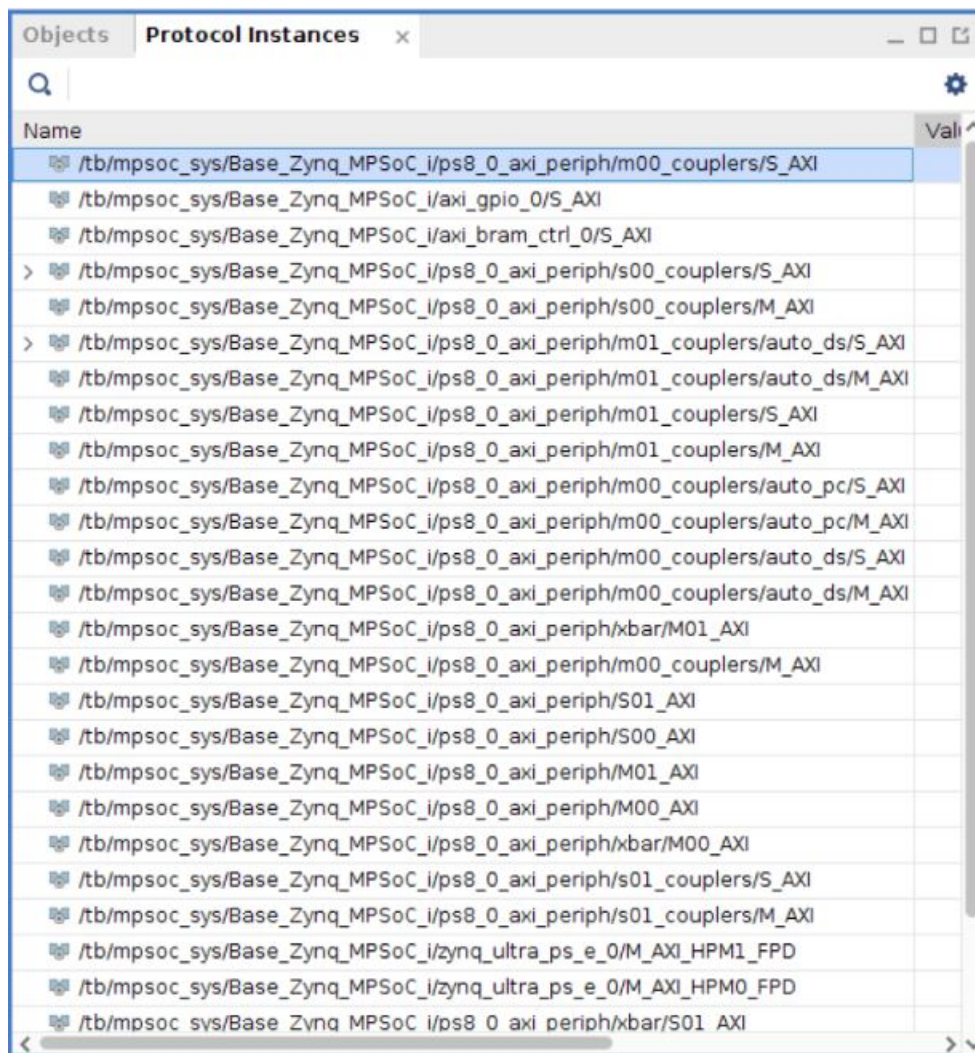


注記: コンピューター リソースを最適化するため、プロトコル インスタンスが波形ウィンドウに追加されるまでプロトコル インスタンスに子オブジェクトは表示されません。

- [Protocol Instances] ウィンドウ:

[Protocol Instances] ウィンドウには、デザインに含まれるすべてのプロトコル インスタンスがリストされます。プロトコル インスタンスへの絶対パスが示されるので、同じ名前のインスタンスもパスによって区別できます。

図 30: [Protocol Instances] ウィンドウ



- プロトコル インスタンス入力:

プロトコル インスタンスの入力信号は、オレンジ色のアイコンで示されており、HDL デザインからの AXI 信号のエイリアスです。入力の上にカーソルを置くと、ツール ヒントにエイリアスの完全パスと実際の信号の完全パスが表示されます。[Scope] および [Objects] ウィンドウのプロトコル インスタンス入力から実際の信号にジャンプすることもできます。プロトコル インスタンス入力の実際の信号に移動するには、次の手順を実行します。

1. [Objects] ウィンドウで [Settings] ボタンをクリックし、[Check All] をオンにしてすべての信号を表示します。
2. プロトコル インスタンス入力を右クリックします。
3. [Go To Actual] をクリックします。



ヒント: プロトコル インスタンス入力信号を波形設定 (WCFG) ファイルに保存するには、入力信号の実際の信号を追加します。プロトコル インスタンスの入力および出力は、WCFG ファイルが読み込まれた後に作成されるので、保存した入力は波形設定に含まれません。プロトコル インスタンス出力は、WCFG ファイルに保存できません。

- プロトコル インスタンス出力:

プロトコル インスタンス出力は、緑色の O アイコンで示されます。これらはプロトコル インスタンスに特殊の信号で、HDL デザインに対応するものではありません。これらの出力信号は、トランザクションを表示するため、波形ビューアーにのみ関係するイベントを生成します。



注意: プロトコル インスタンス出力信号とその内容は、Vivado のリリース間で異なります。Tcl スクリプトを記述する際は、プロトコル インスタンス出力信号の特定の動作を利用しないようにすることをお勧めします。

波形ウィンドウへのプロトコル インスタンスの追加

デザインに存在するプロトコル インスタンスを波形ウィンドウに追加できます。プロトコル インスタンスを波形ウィンドウに追加すると、シミュレーション時間がどれだけ経過していても、Vivado シミュレータでシミュレーション時間 0 からプロトコル インスタンス入力のプロトコル解析が実行されます。プロトコル解析では波形データベース (WDB) が利用されるので、プロトコル インスタンス入力をトレースするよう設定したりプロトコル インスタンスを波形ウィンドウに追加したりしなくても、すべてのプロトコル インスタンスの入力が波形データベースで常にトレースされます。

AXI インターフェイスを Vivado シミュレータで表示するよう IP インテグレーターでマークで説明されている IP インテグレーター ブロック デザインで AXI インターフェイスをマークする方法に加え、[Objects] ウィンドウまたは Tcl コマンドを使用してプロトコル インスタンスを波形ウィンドウに追加できます。



重要: プロトコル インスタンスはコンピューター リソースを多量に使用することがあるので、必要なプロトコル インスタンスのみを追加することをお勧めします。シミュレーション中にプロトコル インスタンスを追加できるので、データを逃すことはありません。

[Objects] ウィンドウを使用してプロトコル インスタンスを波形ウィンドウに追加するには、次の手順を実行します。

1. [\[Objects\] ウィンドウでのプロトコル インスタンスの検索](#)を参照して [Objects] ウィンドウでプロトコル インスタンスを検索します。
2. 次のいずれかの方法を使用してプロトコル インスタンスを波形ウィンドウに追加します。
 - a. プロトコル インスタンスを右クリックし、[Add to Wave] をクリックします。
 - b. プロトコル インスタンスを波形ウィンドウの [Name] 列にドラッグ アンド ドロップします。

Tcl コマンドを使用してプロトコル インスタンスを波形ウィンドウに追加するには、次の手順を実行します。

1. [Vivado シミュレータでのプロトコル インスタンスの検索](#)を参照して [Objects] ウィンドウでプロトコル インスタンスを検索します。
2. プロトコル インスタンス パスをクリップボードにコピーします。
 - a. [Objects] ウィンドウでプロトコル インスタンスを検索した場合は、プロトコル インスタンスをクリックして選択し、プロトコル インスタンス パスをコピーします。
 - b. Tcl コンソールでプロトコル インスタンスを検索した場合は、マウスを使用してプロトコル インスタンス パスを選択してコピーします。
 - c. `get_objects` Tcl コンソールを使用してプロトコル インスタンスを検索した場合は、Tcl コンソールでマウスを使用してプロトコル インスタンス パスのテキストを選択してコピーします。または、次のセクションで説明されているようにオブジェクトを取得できます。
3. 「`add_wave`」と入力してその後にプロトコル インスタンス名をコピーします。



ヒント: プロトコル インスタンス パスに特殊文字が含まれている場合は、パスを二重波かっこで囲みます (例: `add_wave {{path}}`)。

get_objects の使用

Tcl コマンドを使用したプロトコル インスタンスの検索 に説明されているように get_objects Tcl コマンドを使用すると、プロトコル インスタンスが Tcl リストとして返されます。返されたリストは Tcl 変数に保存できます。

```
set p [get_objects -r /* -filter {type==proto_inst}]
```

この Tcl 変数を add_wave Tcl コマンドで指定して、リストに含まれるすべてのプロトコル インスタンスを追加できます。

```
add_wave $p
```

または、ビルトインの lindex コマンドを使用して特定のプロトコル インスタンスを追加できます。次の例では、リストの最初のプロトコル インスタンスを追加しています。

```
add_wave [lindex $p 0]
```

波形ウィンドウでのプロトコル インスタンスの解析

このセクションでは、すべてのインターフェイス タイプに共通の波形の機能を説明します。特定のインターフェイス タイプ (AXI メモリ マップド、AXI4-Stream など) に関する情報は、そのインターフェイス タイプのセクションを参照してください。

波形ウィンドウでのプロトコル インスタンスの理解

プロトコル インスタンスを波形ウィンドウに追加すると、Vivado シミュレータによりプロトコル インスタンスを表す波形オブジェクトの階層が作成されます。この階層の構造は変更できません。AXI インターフェイスのタイプにより階層は異なります。

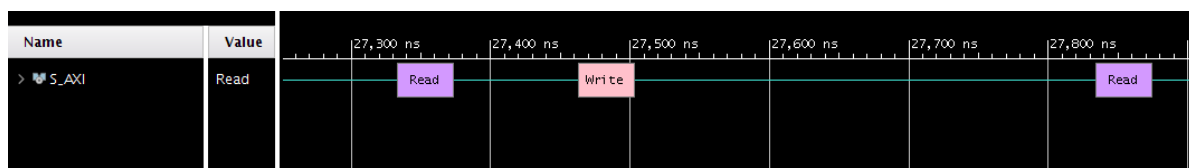


ヒント: 波形オブジェクト階層に含まれていないプロトコル インスタンス入力信号を表示する必要がある場合があります。信号を階層に追加することはできませんが、階層の前後に追加することは可能です。

トランザクション波形の理解

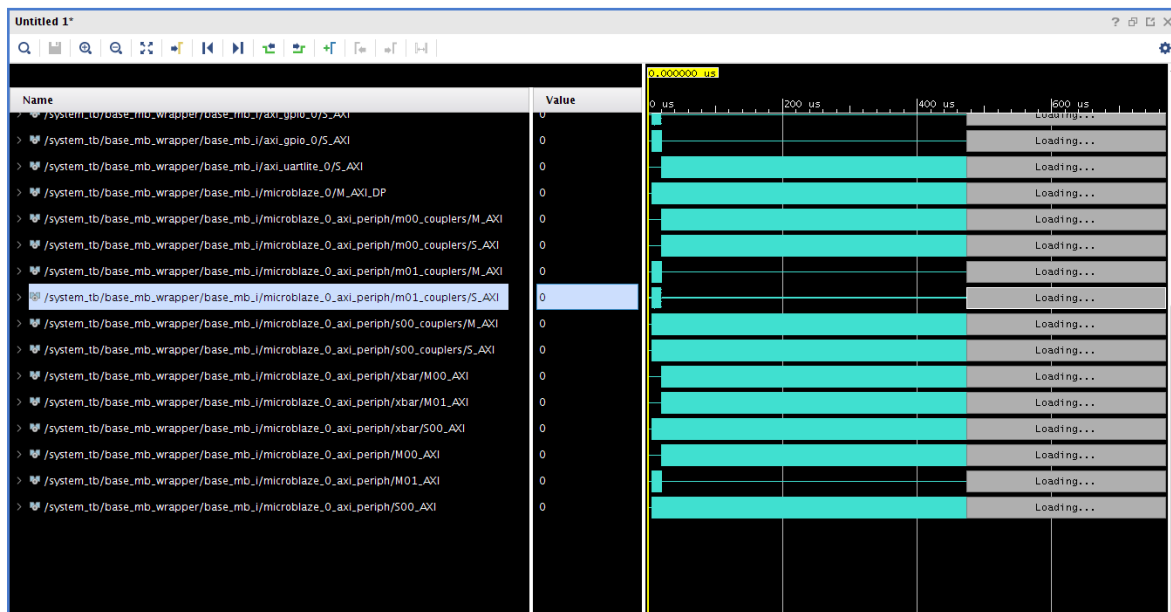
トランザクション波形は、ほかのタイプの波形とは異なります。トランザクション波形は、時間の経過に対する信号の値の変化を表示するのではなく、シミュレーションしているデザインの動作に関するアクティビティの期間を表示します。次の図に、トランザクション波形の例を示します。細い線はアクティビティのない期間を表し、矩形 (トランザクション バーと呼ばれる) はアクティビティの期間を表します。次の図の例には、3 つのトランザクション バーが示されています。

図 31: トランザクション波形の表示



プロトコル インスタンス入力に対してプロトコル解析が実行されている間は、次の図に示すように、トランザクション波形に「Loading」というテキストを含む灰色のバーが表示されます。プロトコル解析が進行すると、灰色のバーがトランザクション データに置き換わります。

図 32: プロトコル解析中のトランザクション波形

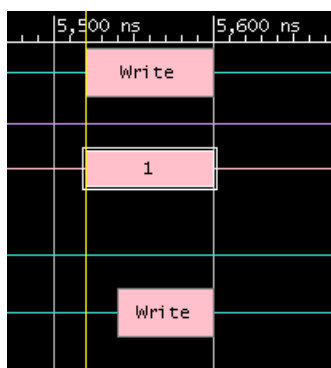


トランザクション バーの使用

トランザクション バーの選択

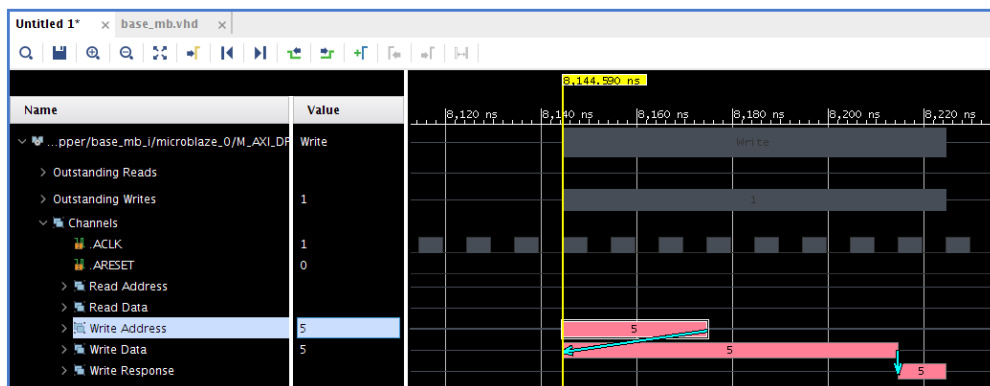
トランザクション バーをクリックして選択すると、次の図に示すように、二重境界線で囲まれます。

図 33: 選択されたトランザクション バー



選択したトランザクションが関連するトランザクション バーのグループのメンバーである場合は、関連のトランザクション バーの関連性を示す矢印が表示されます。波形ウィンドウのその他のオブジェクトは灰色で表示され、選択されたトランザクション バーがハイライトされます。次の図に、選択したトランザクション バーと、その関連性矢印で接続されたトランザクション バーの例を示します。

図 34: トランザクション バーと関連性矢印



トランザクション バーの選択を解除するには、Esc キーを押します。



ヒント: トランザクション波形内でメイン カーソルの位置を変更するには、Ctrl キーを押しながらカーソル時間をクリックします。

関連性矢印を使用したトランザクションのナビゲート

関連性矢印の一端をクリックすると、トランザクション バーの関連性矢印のもう一端に選択が移動し、波形ウィンドウがスクロールしてもう一端が表示されます。

注記: もう一端が既に表示されている場合は、波形ウィンドウはスクロールしません。

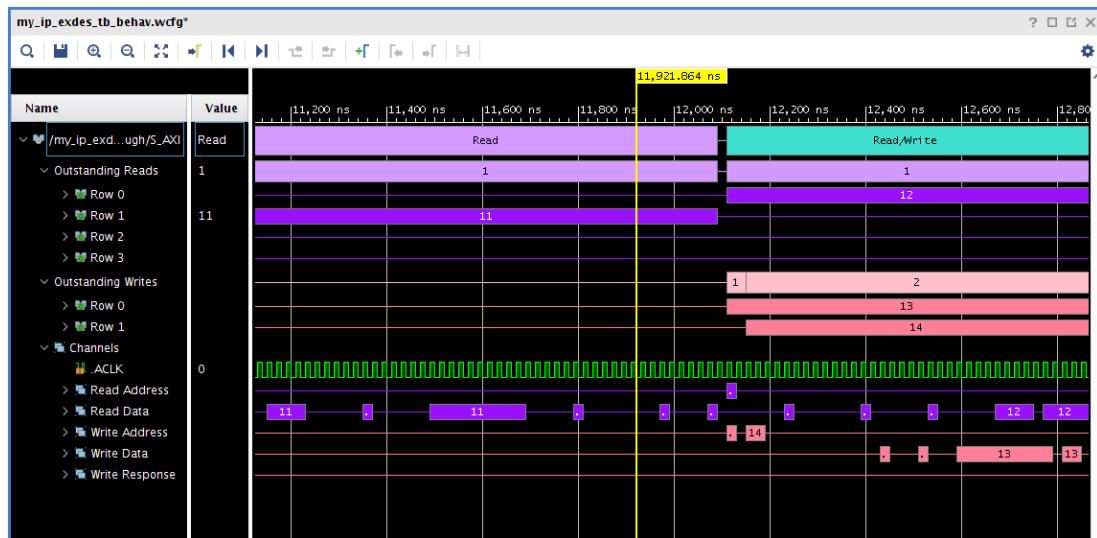
ツール ヒント

トランザクション バーまたは関連性矢印の上にマウスを置くと、プロトコル インスタンス インターフェイスのタイプによって、ツール ヒントにそのトランザクション バーまたは関連性矢印に関する追加情報が表示されます。

AXI メモリ マップド (AXI-MM) インターフェイス

このセクションでは、AXI-MM プロトコル インスタンスに特定のトランザクション表示機能を説明します。AXI-MM インターフェイスのプロトコル インスタンスは、次の図に示すように、波形ウィンドウに波形オブジェクト階層と共に表示されます。

図 35: AXI-MM インターフェイス



最上位サマリ行の理解

AXI-MM プロトコル インスタンスの波形オブジェクト階層の最上位は、最上位サマリ行です。このトランザクション波形は、次の規則に基づいて、AXI インターフェイスの読み出しおよび書き込みアクティビティを示します。

- 1 つまたは複数の AXI 読み出しトランザクションが進行中の場合は、最上位サマリ行に読み出しトランザクションバーが紫色で表示されます。
- 1 つまたは複数の AXI 書き込みトランザクションが進行中の場合は、最上位サマリ行に書き込みトランザクションバーがピンク色で表示されます。
- 1 つまたは複数の AXI 読み出しおよび書き込みトランザクションが進行中の場合は、最上位サマリ行に読み出し/書き込みトランザクションバーが青緑色で表示されます。

AXI トランザクションは抽象的な概念であり、図で示されているトランザクションバーと混同しないようにしてください。AXI トランザクションは、アドレス、データ、およびオプションで応答フェーズを含む、AXI 信号を使用した完全なデータ交換です。



ヒント: パフォーマンスの理由から、波形ビューアーを縮小すると、トランザクションバーは異なる色では表示されなくなり、すべてのトランザクションが青緑色になります。読み出しトランザクションと書き込みトランザクションを区別するには、拡大表示する必要があります。

未処理の読み出し行と未処理の書き込み行の理解

AXI-MM プロトコル インスタンスの波形オブジェクト階層の最上位の下に、未処理の AXI 読み出しトランザクションのグループと書き込みトランザクションのグループがあります。インターフェイス マスターで `A*VALID` または `WVALID` がアサートされ、最後のデータ フェーズまたはオプションの応答フェーズがまだ完了していない場合、AXI トランザクションは未処理となります。未処理の読み出し行には、未処理の AXI 読み出しトランザクションの現在の数が表示されるか、未処理の AXI 読み出しトランザクションがない場合は細い線でアクティビティがないことが示されます。未処理の書き込み行には、未処理の AXI 書き込みトランザクションの現在の数が表示されるか、未処理の AXI 書き込みトランザクションがない場合は細い線でアクティビティがないことが示されます。

トランザクション サマリ行の理解

未処理の読み出し行と未処理の書き込み行の下には、[Row <n>] (<n> は整数) というラベルの付いたトランザクション サマリ行があります。トランザクション サマリは、AXI トランザクションの最初のフェーズで開始し、最後のフェーズで終了する 1 つの AXI トランザクションを表すトランザクション バーです。1 つのトランザクション サマリが特定の数の行に割り当てられていることには特別な意味はなく、複数の未処理の AXI トランザクションが同じ行で一オーバーラップしないようにすることが目的です。



ヒント: シミュレーションが進行するにつれ、トランザクション サマリ行の数が増えることがあります。パフォーマンスの理由から、波形ウィンドウの行はプロトコル解析が完了するまでアップデートされません。シミュレーション全体が完了するまで待たずにシミュレーション中に行の最新状態を表示するには、シミュレーションを一時停止し、[Loading] バーが消えるのを待ちます。

各トランザクション サマリには、シーケンス番号が表示されます。最初の AXI トランザクションはシーケンス番号 1、2 番目の AXI トランザクションにはシーケンス番号 2、のようになります。シーケンス番号の増加は読み出しと書き込みで別であり、ほかのプロトコル インスタンスの AXI トランザクションとも別です。たとえば、あるプロトコル インスタンスの AXI 読み出しトランザクションのシーケンス番号と、別の AXI 書き込みトランザクションのシーケンス番号がどちらも 16 であることもあります。

チャンネル行の理解

チャンネル波形オブジェクト グループは、デフォルトで展開が閉じられています。グループを展開すると、AXI インターフェイス クロックおよびリセット (存在する場合) の論理信号と、インターフェイスの各 AXI チャンネルに対して 1 つのトランザクション行が表示されます。

注記: AXI インターフェイスに 5 つのチャンネルすべてがあるとは限りません。読み出しのみのインターフェイスには、書き込みチャンネルはありません。書き込みのみのインターフェイスには、読み出しチャンネルはありません。書き込みチャンネルを使用する AXI インターフェイスには、AXI マスターに応答情報が必要ないため、応答チャンネルが含まれないものもあります。

各チャンネル行には、その AXI チャンネルの VALID から READY へのハンドシェイクを示すトランザクション バーが表示されます。ただし、複数の連続するデータ ビートは、1 つのトランザクション バーで示されます。AXI トランザクションのすべてのチャンネル トランザクション バーを表示上でまとめるには、各チャンネル トランザクション バーに対応するトランザクション サマリと同じシーケンス番号のタグを付けます。チャンネル行を展開すると、そのチャンネルの主要な AXI 信号が表示されます。



ヒント: 波形オブジェクト階層に含まれていないプロトコル インスタンス入力信号を表示する必要がある場合があります。信号を階層に追加することはできませんが、階層の前後に追加することは可能です。



ヒント: チャンネル トランザクション バーは、対応する AXI 信号イベントの 1 サイクル後まで表示されます。AXI プロトコル解析では、クロックの立ち上がりエッジまたはその後に発生した AXI 信号イベントは、クロックの次の立ち上がりエッジで効果が現れると考慮されます。

チャンネル トランザクション バー、関連性矢印、またはトランザクションの上にマウスを置くと、AXI トランザクションのアドレス フェーズからの情報 AXI アドレス チャンネル信号の値がツール ヒントに表示されます。

注記: インターフェイスに AXI アドレス チャンネル信号がない場合は、ツール ヒントに表示されません。

チャンネル トランザクション バーを選択すると、そのトランザクション バーと同じ AXI トランザクションに関連するすべてのチャンネル トランザクションの関連性矢印が表示されます。関連性矢印の後尾をクリックして、AXI トランザクションの進行状況をアドレスフェーズから応答フェーズまでたどることができます。関連性矢印のチェーンは、アドレス フェーズの前にデータ フェーズがくる場合でも、常にアドレス フェーズ トランザクションから開始します。

エラー状況

インターフェイスにハンドシェイク エラーがある場合、チャネル トランザクションのシーケンス番号がすべて 9 の文字列になることがあります。このシーケンス番号は、データまたは応答フェーズとアドレスまたはデータ フェーズが一致しないことを示します。このエラーの一般的な原因は、読み出し/書き込み ID タグが一致していないこと、および AXI フェーズが実行中にプロトコル アナライザーがリセットに保持される (ARESET または ARESETn 信号がアクティブ) ことです。

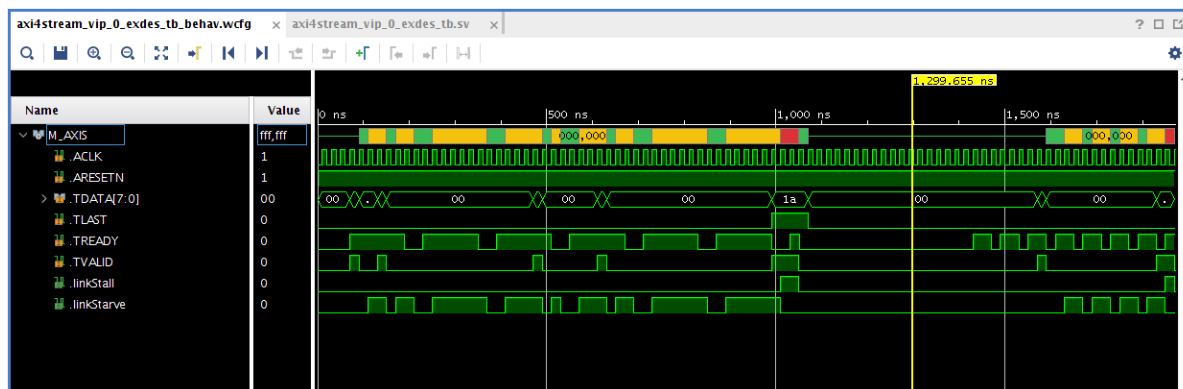


注意: AXI インターコネクトはトランザクションのデバッグ用ではなくパフォーマンスを最適化するように設定されていることがあるので、AXI インターコネクト内部の AXI インターフェイスにインターフェイスに接続されているリセット信号ではなくほかのリセット信号が適用され、波形ビューアーでトランザクション エラーとなることがあります。インターフェイスでトランザクション エラーが発生した場合は、インターコネクト外にあるインターフェイスを監視することをお勧めします。

AXI4-Stream (AXI-S) インターフェイスの解析

このセクションでは、AXI4-Stream プロトコル インスタンスに特定のトランザクション表示機能を説明します。AXI-S インターフェイスのプロトコル インスタンスは、次の図に示すように、波形ウィンドウに波形オブジェクト階層と共に表示されます。

図 36: AXI4-Stream (AXI-S) インターフェイス



AXI-S インターフェイスの波形オブジェクト階層には、トランザクション行が 1 つのみ含まれ、階層の最上位に表示されます。行の各トランザクション バーは、1 つの完全な AXI トランザクションに対応します。トランザクションのテキストは、AXI 信号 TID からのストリーム識別子と AXI 信号 TDEST からの大まかな配線情報から構成されます。

トランザクション バーには、次の表に説明するように、AXI トランザクションのステータスが色分けして示されます。

表 13: AXI トランザクションのステータス

色	ステータス	説明
緑	Normal	データが通常どおりにストリーミングされています。
黄色	Starve	スレーブがマスターからのデータを待っています。
赤	Stall	マスターがスレーブで消費できる以上の速度でデータを生成しています。

最上位トランザクション行の下に、主な AXI 信号と、stall および starve ステータス信号を含む波形オブジェクト階層が表示されます。ステータス信号は、トランザクション行の色と同じ情報を示します。



ヒント: チャンネル トランザクション バーは、対応する AXI 信号イベントの 1 サイクル後まで表示されます。AXI プロトコル解析では、クロックの立ち上がりエッジまたはその後に発生した AXI 信号イベントは、クロックの次の立ち上がりエッジで効果が現れると考慮されます。

エラー状況

ハンドシェイク エラーが発生すると、複数の F を含むテキストを示すエラー トランザクションが生成されます。

Vivado シミュレータを使用したデザインのデバッグ

Vivado® Design Suite シミュレータには次の機能があります。

- ソース コードを確認
- ブレークポイントを設定し、ブレークポイントに達するまでシミュレーションを実行
- コード セクションをステップオーバー
- 波形オブジェクトを特定値に設定

この章では、デバッグ方法のほか、デバッグ プロセスで有効な Tcl コマンドについても説明します。また、サードパーティ シミュレータを使用したデバッグのフローについても説明します。

ソース レベルでのデバッグ

HDL ソース コードをデバッグすると、デザインの予期しない動作を検出できます。デバッグでは、ソース コードの実行を制御することにより、問題の原因を見つけます。デバッグで使用可能なストラテジは、次のとおりです。

- 1 行ずつステップ実行: 任意の開発段階で、`step` コマンドを使用して HDL ソース コードを 1 行ずつデバッグすることにより、デザインが意図どおりに動作することを確認します。コードの 1 つの行の実行が終了したら、再び `step` コマンドを実行して解析を続けます。詳細は、[シミュレーションのステップ実行](#) を参照してください。
- HDL コードの特定の行にブレークポイントを設定し、ブレークポイントまでシミュレーションを実行: 大型のデザインでは、HDL ソース コードの各行を実行するたびに停止すると時間がかかりすぎるので、HDL ソース コードの特定の位置にブレークポイントを設定し、テストベンチの最初またはデザインの現在の地点からシミュレーションを実行したときに各ブレークポイントで停止するようにします。停止後にシミュレーションを続行するには、`[Step]`、`[Run All]`、または `[Run For]` コマンドを使用します。詳細は、[ブレークポイントの使用](#) を参照してください。
- 条件を設定します。ツールが各条件を評価し、その条件が当てはまる場合は、Tcl コマンドを実行します。次のコマンドを使用します。

```
add_condition <condition> <instruction>
```

詳細は、[条件の追加](#) を参照してください。


シミュレーションのステップ実行

デザインが意図したとおりに動作することを確認するため、`step` コマンドを使用して HDL ソース コードを 1 行ずつ実行できます。

このコマンドが実行されているコード行はハイライトされ、矢印が表示されます。

ブレークポイントを作成して、ステップ実行中に停止する箇所を増やすこともできます。シミュレータでのデバッグストラテジについては、[ブレークポイントの使用](#)を参照してください。

1. シミュレーションでステップ実行するには、次の手順に従います。

- 現在の実行時間から [Run > Step] をクリックするか、[Step] ボタン  をクリックします。

最上位デザイン ユニットに関連付けられている HDL が波形ウィンドウに新しく表示されます。

- 開始時間 (0 ns) からシミュレーションを再開します。[Restart] コマンドを使用すると、テストベンチの冒頭に時間をリセットできます。[第 4 章: Vivado シミュレータを使用したシミュレーション](#)を参照してください。
2. 波形ウィンドウまたは HDL ファイルのタブを右クリックして [Tile Horizontally] をクリックし、波形と HDL コードを同時に表示します。
 3. デバッグが終了するまで、[Step] を繰り返します。

コマンドが 1 行ずつ実行され、それによって矢印がコードの下の方に移動していきます。シミュレータが別のファイルの行を実行する場合は、そのファイルが開き、同様に実行されている行に矢印が表示されます。ほとんどのシミュレーションでは、[Step] コマンドを実行しているときに複数のファイルが開くのが通常です。Tcl コンソールにも、step コマンドの実行が HDL コードのどこまで進んだかが示されます。

ブレークポイントの使用



ブレークポイントは、ユーザーが指定するソース コード内の停止点で、デザインをデバッグする際に使用できます。



ヒント: ブレークポイントは、[Step] コマンドで 1 行ずつ停止すると時間がかかりすぎてしまうような大規模デザインをデバッグする場合に使用すると、特に便利です。

シミュレータで HDL ソース コードにブレークポイントを設定し、そのブレークポイントに達するまでコードを連続して実行できます。

注記: ブレークポイントは実行可能なコードにのみ設定できます。実行不可能なコードの行にブレークポイントを設定しても、ブレークポイントは追加されません。

1. シミュレーションを実行します。
2. ソース ファイルを表示して、該当する行の左にある白抜き丸  をクリックします。ブレークポイントが正しく設定されると、赤く塗りつぶされた丸  になります。

この手順が終了すると、シミュレーション ブレークポイント アイコンがコード行の横に表示されます。

Tcl コマンド `add_bp <file_name> <line_number>` を入力します。

このコマンドにより、`<line_number>` の `<file_name>` にブレークポイントが追加されます。コマンドの使用方法は、Vivado Design Suite のヘルプ (-help) または『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。

HDL ソース ファイルを開きます。

3. HDL ソース ファイルの実行可能な行にブレークポイントを設定します。
4. すべてのブレークポイントが設定されるまで手順 1 および 2 を繰り返します。
5. 次の実行オプションを使用してシミュレーションを実行します。
 - 最初から実行する場合は、[Run]→[Restart] をクリックします。

- [Run]→[Run All] または [Run]→[Run For] コマンドを使用します。

シミュレーションがブレークポイントに到達するまで実行され、停止します。


HDL ソース ファイルに矢印が表示され、ブレークポイントの停止ポイントが示されます。


6. 手順 4 を繰り返して、結果に満足するまでシミュレーションをブレークポイントごとに進めていきます。

シミュレーションは、HDL ソース ファイルに設定した各ブレークポイントで停止します。

デザインのデバッグ中には、[Run]→[Step] コマンドを実行して 1 行ごとにコードを検証することで、さらに詳細にデザインを検証することもできます。

HDL ソース コードから 1 つのブレークポイントまたはすべてのブレークポイントを削除できます。

1 つのブレークポイントを削除するには、[Breakpoint] ボタン  をクリックします。

すべてのブレークポイントを削除するには、[Run]→[Delete All Breakpoints] をクリックするか、[Delete All Breakpoints] ボタン  をクリックします。

すべてのブレークポイントを削除するには、次を入力します。

- `remove_bps -all`

ブレークポイント オブジェクトの指定したリストに関するブレークポイント情報を取得するには、次を入力します。

- `report_bps`

条件の追加

ブレークポイントを条件に基づいて追加し、診断メッセージを表示するには、次のコマンドを使用します。

```
add_condition <condition> <message>
```

たとえば、Vivado IDE の BFT サンプル デザインで、`wbClk` 信号と `reset` の両方が 1 になったときにシミュレーションを停止して診断メッセージを出力するには、シミュレーション開始時に次のコマンドを実行します。

```
add_condition {reset == 1 && wbClk == 1} {puts "Reset went to high"; stop}
```

BFT サンプル デザインは、追加された条件により、条件が満たされたときにシミュレーションが 5 ns 停止し、「Reset went to high」というメッセージがコンソールに表示されます。シミュレータは、次の `step` または `run` コマンドでシミュレーションが再開されるまで待機します。

-notrace オプション

通常、`add_condition` コマンドを実行すると、指定した `Tcl` コマンドがコンソール、ログ ファイル、ジャーナル ファイルにも含まれますが、`-notrace` オプションを使用すると、出力は記録されますが、コマンド自体が含まれなくなります。

たとえば、次のコマンドを実行したとします。

```
puts 'Hello'
```

上記のコマンドは、コンソール、ログ ファイル、ジャーナル ファイルに次のように表示されます。


```
# puts 'Hello'
Hello
```

これに `-notrace` オプションを付けて実行すると、次のように出力のみが表示されるようになります。

```
Hello
```

シミュレーションの一時停止

シミュレーションを実行中に一時停止するには、[Break] コマンドを使用します。これにより、シミュレーション セッションは開いたままになります。

シミュレーションを一時停止するには、[Simulation]→[Break] をクリックするか、[Break] ボタン  をクリックします。

シミュレーションが次の実行可能な HDL 行で停止します。シミュレーションの停止した行がテキスト エディターで表示されます。

注記: この動作は、`-debug <kind>` オプションでコンパイルしたデザインにも適用されます。

[Run All]、[Run]、[Step] コマンドを使用すると、シミュレーションを再開できます。詳細は、[シミュレーションのステップ実行](#)を参照してください。

シミュレーション実行のトレース

シミュレーション実行中の各ソース行に関する注記を Tcl コンソールに表示できます。このように各行を表示することを「行トレース」と呼びます。

行トレースをオンにするには、次のいずれかの Tcl コマンドを使用します。

```
ltrace on
set_property line_tracing true [current_sim]
```

行トレースをオフにするには、次のいずれかの Tcl コマンドを使用します。

```
ltrace off
set_property line_tracing false [current_sim]
```

シミュレーション実行中の各プロセスに関する注記を Tcl コンソールに表示できます。このように各プロセスを表示することを「プロセストレース」と呼びます。

プロセストレースをオンにするには、次のいずれかの Tcl コマンドを使用します。

```
ptrace on
set_property process_tracing true [current_sim]
```

プロセストレースをオフにするには、次のいずれかの Tcl コマンドを使用します。

```
ptrace off
set_property process_tracing false [current_sim]
```

波形オブジェクトを特定値に設定

force コマンドの使用

Vivado シミュレータでは、信号、ワイヤ、レジスタを、指定の時間に、または指定の期間中、特定の値に設定できます。また、ある時間が経過した後にオブジェクトに適用する強制値も設定できます。



ヒント: force は、ある信号に対し HDL で定義された動作を無効にするコマンドであると同時に、Tcl のファーストクラス オブジェクトであり、Tcl 変数として保存できます。

HDL 信号に対して force コマンドを使用すると、HDL デザインで定義されている信号の動作を変更できます。たとえば、信号の動作を次の目的で変更できます。

- HDL テストベンチにより駆動されていないテストベンチ信号にスティミュラスを供給するため
- デバッグ中に無効な値を一時的に修正するため (問題の解析を継続できるようにするため)

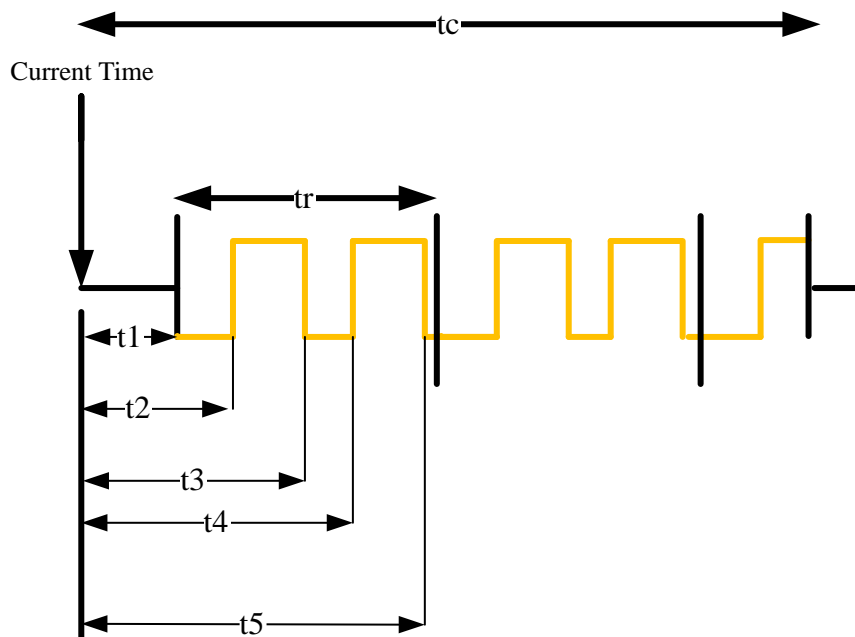
使用可能な force コマンドは、次のとおりです。

- [Force Constant] コマンド
- [Force Clock] コマンド
- [Remove Force] コマンド

次の図に、次の `add_force` コマンドを実行したときに、その機能がどのように適用されるかを示します。

```
add_force mySig {0 t1} {1 t2} {0 t3} {1 t4} {0 t5} -repeat_every tr -  
cancel_after tc
```

図 37: -add_force オプションを使用した結果



このコマンドの詳細を表示するには、Tcl コンソールに次を入力します。

```
add_force -help
```

[Force Constant] コマンド

[Force Constant] オプションを使用すると、信号を定数値に固定し、HDL コード内で割り当てられた値や前回適用された別の定数、または [Force Clock] を変更できます。

[Force Constant] および [Force Clock] は、[Objects] ウィンドウまたは波形ウィンドウで右クリックして表示されるオプションです (次の図を参照)。または、テキスト エディターでソース コードを開いて使用できます。



ヒント: [Objects]、[Sources]、[Scopes] ウィンドウでアイテムをダブルクリックすると、テキスト エディターが開きます。テキスト エディターの追加情報は、『Vivado Design Suite ユーザー ガイド: Vivado IDE の使用』 ([UG893](#)) を参照してください。

図 38: force 関連のコマンド

Go To Source Code	
Show in Object Window	
Report Drivers	
Force Constant...	
Force Clock...	
Remove Force	
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Delete
Find...	Ctrl+F
Find Value...	Ctrl+Shift+F
Select All	Ctrl+A

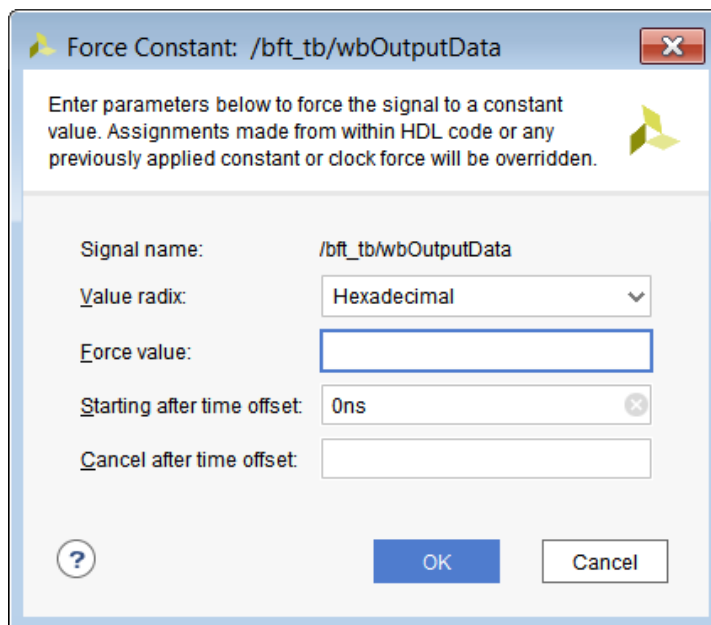
force オプションは、Vivado シミュレータで force がサポートされていないオブジェクトに対しては無効になります。オブジェクトのタイプやこういったオブジェクトに対する Vivado シミュレータの制限によって、このようなオブジェクトがサポートされないことがあります。



ヒント: force オプションが無効になったモジュールまたはエンティティ ポートに force を実行するには、その接続された実際の信号の 1 レベル上のスコープで force を実行してみます。add_force Tcl コマンド (例: add_force myObj 0) を使用すると、オプションが無効になる理由が表示されます。

[Force Constant] をクリックすると、次の図に示す [Force Constant] ダイアログ ボックスが開き、関連する値を入力できます。

図 39: [Force Constant] ダイアログ ボックス



[Force Constant] ダイアログ ボックスには、次のオプションがあります。

- [Signal name]: デフォルトの信号名 (選択したオブジェクトの完全パス名) を表示します。
- [Value radix]: 選択した信号の基数を指定します。サポートされている基数タイプ (2 進数、16 進数、符号なしの 10 進数、符号付きの 10 進数、符号付き大きさ、8 進数、ASCII) のいずれかを選択できます。GUI には、基数設定に基づいた値の入力のみが可能になります。たとえば、[Binary] を選択すると、0 と 1 以外の数値は入力できません。
- [Force value]: 選択された基数に基づいて強制する定数値を指定します。基数の詳細は、[デフォルト基数の変更](#) および [アナログ波形の使用](#) を参照してください。
- [Starting after time offset]: 指定した時間後に開始します。デフォルトの開始時間は 0 です。時間は、10 や 10 ns のように、文字列で指定できます。単位を付けずに数値だけ入力すると、Vivado シミュレータではデフォルト単位の ns が使用されます。
- [Cancel after time offset]: 指定した時間後にキャンセルします。時間は、10 や 10 ns のように、文字列で指定できます。単位を付けずに数値だけ入力すると、デフォルトのシミュレーション時間単位が使用されます。

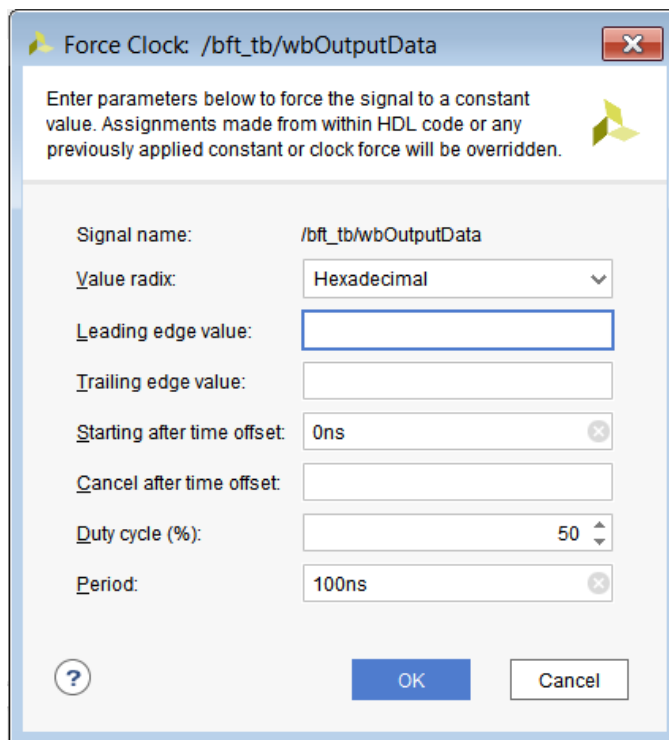
Tcl コマンド:

```
add_force /testbench/TENSOUT 1 200 -cancel_after 500
```

[Force Clock] コマンド

[Force Clock] コマンドを使用すると、クロック信号のように指定期間 2 つのステート間を指定レートでトグルする値を、信号に割り当てることができます。[Objects] ウィンドウで [Force Clock] コマンドをクリックすると、次の図に示す [Force Clock] ダイアログ ボックスが開きます。

図 40: [Force Clock] ダイアログ ボックス



[Force Clock] ダイアログ ボックスには、次のオプションがあります。

- [Signal name]: デフォルトの信号名 ([Objects] ウィンドウまたは波形で選択したアイテムの完全パス名) を表示します。



ヒント: [Force Clock] コマンドは、クロック信号だけでなく、任意の信号に適用して切り替わる値を定義できます。

- [Value radix]: 選択した信号の基数を指定します。ドロップダウン リストから [Binary]、[Hexadecimal]、[Unsigned Decimal]、[Signed Decimal]、[Signed Magnitude]、[Octal]、[ASCII] のいずれかの基数を選択します。
- [Leading edge value]: クロック パターンの最初のエッジを指定します。[Value radix] で定義した基数が使用されます。
- [Trailing edge value]: クロック パターンの 2 番目のエッジを指定します。[Value radix] で定義した基数が使用されます。
- [Starting after time offset]: 現在のシミュレーションから指定した時間後に force コマンドを開始します。デフォルトの開始時間は 0 です。時間は、10 や 10 ns のように、文字列で指定できます。単位を付けずに数値だけ入力すると、Vivado シミュレータでデフォルトのユーザー単位が使用されます。
- [Cancel after time offset]: 現在のシミュレーションから指定した時間後に force コマンドをキャンセルします。時間は、10 や 10 ns のように、文字列で指定できます。単位を付けずに数値だけ入力すると、Vivado シミュレータでデフォルトのシミュレーション時間単位が使用されます。
- [Duty cycle (%): クロック パルスがアクティブ ステートである時間の割合を % で指定します。使用できる値は 0 ~ 100 で、デフォルト値は 50 です。
- [Period]: クロック パルスの長さを時間で指定します。時間は、10 や 10 ns のように、文字列で指定できます。

注記: 基数の詳細は、[デフォルト基数の変更](#) および [アナログ波形の使用](#) を参照してください。

Tcl コマンド例:

```
add_force /testbench/TENSOUT -radix bin {0} {1} -repeat_every 10ns -
cancel_after 3us
```

[Remove Force] コマンド

オブジェクトから特定の force を削除するには、次の Tcl コマンドを使用します。

```
remove_forces <force object>
remove_forces <HDL object>
```

バッチ モードでの Force コマンドの使用

次のコード例に、`add_force` コマンドを使用して信号を指定の値に強制する方法を示します。簡単な Verilog 回路が提供されています。最初の例は、`add_force` コマンドをインタラクティブに使用する例を、2 番目はスクリプトを使用した例を示しています。

例 1: force の追加

次のコード例は Verilog 回路です。

```
module bot(input in1, in2,output out1);
  reg sel;
  assign out1 = sel? in1: in2;
endmodule
module top;
  reg in1, in2;
  wire out1;
  bot I1(in1, in2, out1);
  initial
  begin
    #10 in1 = 1'b1; in2 = 1'b0;
    #10 in1 = 1'b0; in2 = 1'b1;
  end
  initial
    $monitor("out1 = %b\n", out1);
endmodule
```

`add_force` の効果を観察するには、次のコマンドを入力します。

```
xelab -vlog tmp.v -debug all
xsim work.top
```

コマンド プロンプトに次を入力します。

```
add_force /top/I1/sel 1
run 10
add_force /top/I1/sel 0
run all
```

`add_force` コマンドを使用すると、信号、ワイヤ、またはレジスタを指定値に設定できます。

```
add_force [-radix <arg>] [-repeat_every <arg>] [-cancel_after <arg>] [-
quiet]
[-verbose] <hdl_object> <values>...
```

この Tcl コマンドおよびその他の 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) を参照してください。

例 2: スクリプトでの add_force と remove_forces の使用

次は、カウンタをインスタンス化して Verilog ファイル top.v の例です。このファイルは、次のコマンド例で使用できます。

```
module counter(input clk,reset,updown,output [4:0] out1);
reg [4:0] r1;
always@(posedge clk)
begin
    if(reset)
        r1 <= 0;
    else
        if(updown)
            r1 <= r1 + 1;
        else
            r1 <= r1 - 1;
end
assign out1 = r1;
endmodule
module top;
reg clk;
reg reset;
reg updown;
wire [4:0] out1;
counter I1(clk, reset, updown, out1);
initial
begin
    reset = 1;
    #20 reset = 0;
end
initial
begin
    updown = 1; clk = 0;
end
initial
    #500 $finish;
initial
    $monitor("out1 = %b\n", out1);
endmodule
```

コマンド例

1. 次のコマンドを含む add_force.tcl というファイルを作成します。

```
create_project add_force -force
add_files top.v
set_property top top [get_filesets sim_1]
set_property -name xelab.more_options -value {-debug all} -objects
[get_filesets
sim_1]
set_property runtime {0} [get_filesets sim_1]
launch_simulation -simset sim_1 -mode behavioral
add_wave /top/*
```

2. Vivado Design Suite を Tcl モードで起動して、add_force.tcl ファイルを読み込みます。

3. [Tcl Console] ウィンドウに次を入力します。

```
set force1 [add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500]
set force2 [add_force updown {0 10} {1 20} -repeat_every 30]
run 100
```

out1 の値の増減を波形ウィンドウで確認します。start_gui コマンドを使用して、Vivado IDE で波形を確認します。

updown 信号の値を波形ウィンドウで確認します。

4. [Tcl Console] ウィンドウに次を入力します。

```
remove_forces $force2
run 100
```

out1 の値だけが増加したことを確認します。

5. [Tcl Console] ウィンドウに次を入力します。

```
remove_forces $force1
run 100
```

out1 信号はトグルしないので、clk の値が変化していないことを確認します。

Vivado シミュレータを使用した消費電力解析

SAIF (Switching Activity Interchange Format) は、シミュレータ ツールで生成されたスイッチング アクティビティを抽出して保存する ASCII 形式のレポートです。このスイッチング アクティビティは、サイリンクスの消費電力解析および最適化ツールにバックアノテートして消費電力の測定および見積りに使用できます。

SAIF (Switching Activity Interchange Format) 出力は、サイリンクス消費電力ツールおよび report_power Tcl コマンド用に最適化されています。Vivado シミュレータでは、SAIF ファイルに次の HDL タイプを記述できます。追加情報は、『Vivado Design Suite ユーザー ガイド: 消費電力解析および最適化』(UG907) のこのセクションを参照してください。

- Verilog:
 - 入力、出力、入出力ポート
 - 内部ワイヤ宣言
- VHDL:
 - std_logic、std_ulogic、および bit (スカラー、ベクター、配列) 型の入力、出力、入出力ポート。

注記: Vivado Design Suite ではタイミング シミュレーション用の VHDL ネットリストは生成されないで、VHDL ソースは RTL レベルのコード専用で、ネットリスト シミュレーションには使用できません。

RTL レベルのシミュレーションの場合、ブロック レベルのポートのみが生成され、内部信号は生成されません。

サードパーティ シミュレータを使用した消費電力解析については、[第 3 章: サードパーティ シミュレータを使用したシミュレーションの消費電力解析用の SAIF の出力](#)、[IES での SAIF の出力](#)、および [VCS での SAIF の出力](#)を参照してください。

SAIF 出力の生成

log_saif コマンドを使用する前に、open_saif を呼び出す必要があります。log_saif はオブジェクトまたは値を返しません。

1. RTL コードを -debug typical オプションを使用してコンパイルし、SAIF 出力をイネーブルにします。

```
xelab -debug typical top -s mysim
```

2. 次の Tcl コマンドを使用して SAIF 出力を開始します。

```
open_saif <saif_file_name>
```

3. 次の Tcl コマンドのいずれかを入力して生成する範囲および信号を追加します。

```
log_saif [get_objects]
```

すべてのインスタンスを繰り返しログに記録するには、次の Tcl コマンドを入力します。

```
log_saif [get_objects -r *]
```

4. run コマンドのいずれかを使用してシミュレーションを実行します。
5. シミュレーション データを SAIF 形式にインポートするには、次を入力します。

```
close_saif
```

SAIF コマンドの例

SAIF を記録するには、次のコマンドを使用します。

- スコープ内のすべての信号: /tb: log_saif /tb/*
- スコープ内のすべてのポート: /tb/UUT
- a で開始して b で終了し、その間に数字が含まれる名前のオブジェクト:

```
log_saif [get_objects -regexp {^a[0-9]+b$}]
```

- current_scope および children_scope のオブジェクト:

```
log_saif [get_objects -r *]
```

- current_scope のオブジェクト:

```
log_saif * or log_saif [get_objects]
```

- スコープ /tb/UUT のポートのみ:

```
id="ah453025">log_saif [get_objects -filter {type == in_port || type == out_port || type == inout_port || type == port } /tb/UUT/* ]
```

- スコープ scope /tb/UUT の内部信号のみ:

```
log_saif [get_objects -filter { type == signal } /tb/UUT/* ]
```



ヒント: HDL オブジェクトを必要とする Tcl コマンドすべてに、このフィルター機能を使用できます。

Tcl シミュレーション バッチ ファイルを使用した SAIF の出力

```
sim.tcl:  
open_saif xsim_dump.saif  
log_saif /tb/dut/*  
run all  
close_saif  
quit
```

Tcl コマンド report_drivers の使用

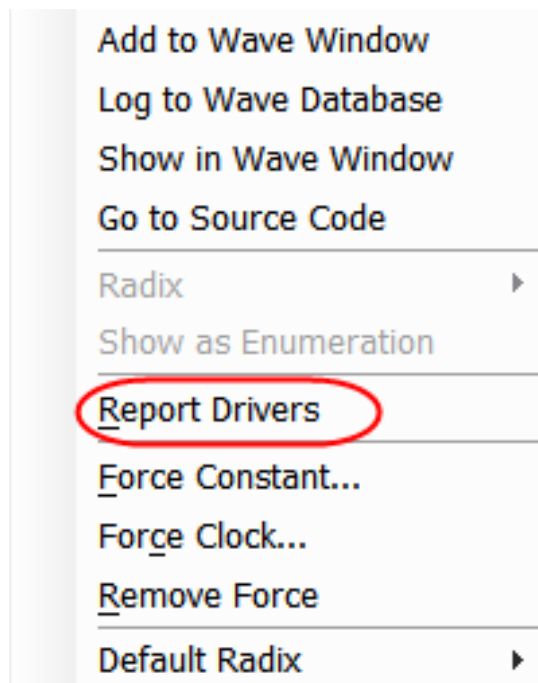
`report_drivers` Tcl コマンドを使用すると、HDL オブジェクトの値を駆動している信号がレポートされます。構文は次のとおりです。

```
report_drivers <hdl_object>
```

Tcl コンソールにドライバーが表示され、wire または signal 型の HDL オブジェクトへの代入の右側に現在の駆動値が表示されます。

[Objects] ウィンドウまたは波形ウィンドウのコンテキスト メニュー、あるいはテキスト エディターから `report_drivers` コマンドを呼び出すこともできます。コンテキスト メニューを開くには、信号を右クリックして、[Report Drivers] を選択します (次の図を参照)。結果は Tcl コンソールに表示されます。

図 41: コンテキスト メニューの [Report Drivers] コマンド オプション



VCD 機能の使用

シミュレーション出力を記録するには、VCD (Value Change Dump) ファイルを使用できます。Tcl コマンドは、出力される値に関連する Verilog システム タスクに基づきます。

VCD 機能では、次の表にリストされる Tcl コマンドで Verilog システム タスクが記述されます。

表 14: VCD の Tcl コマンド

Tcl コマンド	説明
open_vcd	シミュレーション出力を取り込むための VCD ファイルを開きます。\$dumpfile Verilog システム タスクと同様の操作を実行します。
checkpoint_vcd	\$dumpall Verilog システム タスクと同様の操作を実行します。
start_vcd	\$dumpon Verilog システム タスクと同様の操作を実行します。
log_vcd	指定した HDL オブジェクトの VCD ファイルを記録します。\$dumpvars Verilog システム タスクと同様の操作を実行します。
flush_vcd	\$dumpflush Verilog システム タスクと同様の操作を実行します。
limit_vcd	\$dumplimit Verilog システム タスクと同様の操作を実行します。
stop_vcd	\$dumpoff Verilog システム タスクと同様の操作を実行します。
close_vcd	VCD の生成を閉じます。

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、Tcl コンソールに次を入力してください。

```
<command> -help
```

例:

```
open_vcd xsim_dump.vcd
log_vcd /tb/dut/*
run all
close_vcd
quit
```

詳細は、[Verilog 言語サポートの例外](#) を参照してください。

VCD データを使用して、シミュレーション エラーをデバッグするためにシミュレータの出力を検証できます。

関連情報

[Vivado シミュレータの混合言語サポートおよび例外](#)

Tcl コマンド log_wave の使用

log_wave コマンドを使用すると、Vivado シミュレータの波形ビューアーで指定した HDL オブジェクトを表示するため、シミュレーション出力を記録できます。add_wave とは異なり、log_wave コマンドでは波形ビューアー (波形設定) に波形オブジェクトは追加されず、Vivado シミュレータの波形データベース (wdb) への出力が記録されるようになるだけです。



ヒント: オブジェクト挿入以前のオブジェクト値を表示するには、シミュレーションを実行し直す必要があります。値がないためにシミュレーションを再実行するのを避けるには、シミュレーション run を開始するときに、log_wave -r / Tcl コマンドを実行してデザインの表示可能な HDL オブジェクトすべての値を取り込みます。

構文

```
log_wave [-recursive] [-r] [-quiet] [-verbose] <hdl_objects>...
```

Tcl コマンド log_wave の使用例

波形出力を記録するには、次を使用します。

- デザイン内のすべての信号 (代替の最上位モジュールのものを除く):

```
log_wave -r /
```

- スコープ /tb 内のすべての信号:

```
log_wave /tb/*
```

- a で始まって b で終わり、ab 間に数字が含まれる名前のオブジェクト:

```
log_wave [get_objects -regexp {^a[0-9]+b$}]
```

- 現在のスコープおよびすべての下位スコープ内のすべてのオブジェクト:

```
log_wave -r *
```

- メッセージの制限を一時的に解除し、コマンドからのすべての出力を返します。

```
log_wave -v
```

- 現在のスコープ内のオブジェクト:

```
log_wave *
```

- スコープ /tb/UUT のポートのみ:

```
log_wave [get_objects -filter {type == in_port || type == out_port ||  
type ==  
inout_port || type == port} /tb/UUT/*]
```

- スコープ /tb/UUT の内部信号のみ:

```
log_wave [get_objects -filter {type == signal} /tb/UUT/*]
```

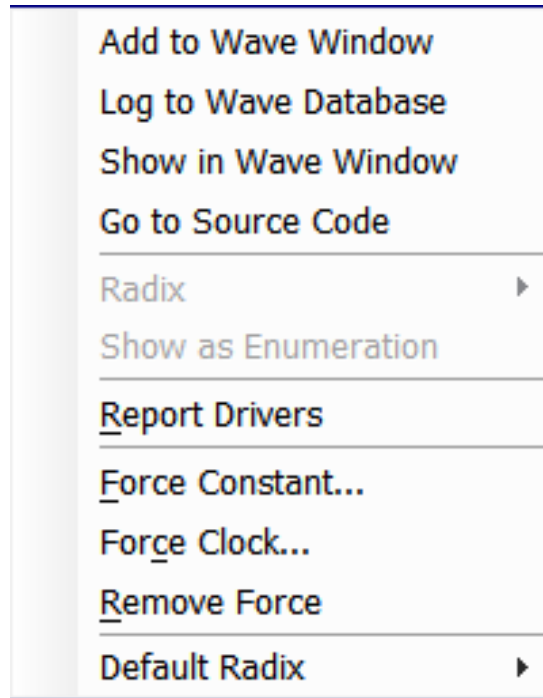
信号順、名前形式、基数、色などの波形設定は、必要に応じて波形設定 (WCFG) ファイルに保存されます。第 5 章: [Vivado シミュレータを使用したシミュレーション波形の解析](#) を参照してください。

[Objects] ウィンドウ、波形ウィンドウ、テキスト エディターでの信号のクロスプローブ

Vivado シミュレータでは、[Objects] ウィンドウ、波形ウィンドウ、テキスト エディターにある信号に対し、クロスプローブを実行できます。










[Objects] ウィンドウから波形ウィンドウに信号が表示されているかどうかを確認でき、またはその逆も可能です。信号を右クリックして次の図に示すコンテキスト メニューを開き、[Show in Wave Window] をクリックするか、信号がまだ波形ウィンドウに表示されていない場合は [Add to Wave Window] をクリックします。

図 42: [Objects] ウィンドウのコンテキスト メニュー



テキスト エディターで信号をクロスプローブすることもできます。信号を右クリックして次の図に示すコンテキスト メニューを開き、[Add to Wave Window]、[Show in Waveform]、または [Show in Objects] をクリックします。波形ウィンドウまたは [Objects] ウィンドウで信号がハイライトされます。

図 43: テキスト エディターのコンテキスト メニュー

	Save File	Ctrl+S
	Save File As...	
	Save All Files...	
	Find Usages...	Ctrl+Alt+U
	Go to Definition	Ctrl+Shift+D
	Undo	Ctrl+Z
	Redo	Ctrl+Shift+Z
	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+V
	Duplicate Selection	Ctrl+D
	Select All	Ctrl+A
	Toggle column selection mode	Ctrl+Back Slash
	Find...	Ctrl+F
	Replace...	Ctrl+R
	Find in Files...	Ctrl+Shift+F
	Replace in Files...	Ctrl+Shift+R
	Indent Selection	Tab
	Unindent Selection	Shift+Tab
	Toggle Line Comments	Ctrl+Slash
	Toggle Block Comments	Ctrl+Shift+Slash
	Blank Operations	▶
	Diff with	▶
	Folding	▶
	Add to Wave Window	
	Show in Wave Window	
	Show in Objects Window	
	Report Drivers	
	Force Constant...	
	Force Clock...	
	Remove Force	
	Insert Template	

Vivado シミュレータの init.tcl

Vivado シミュレータでは、シミュレーション実行中に次のディレクトリにある init ファイルが読み込まれます。

```
$HOME/.xilinx/xsim/xsim_init.tcl
```

この init ファイルは、複数 run に対してプロパティを設定する場合に便利です。これらは Tcl ファイル内に記述でき、Vivado シミュレータでタイム 0 よりも前に読み込まれます。

サブプログラムの呼び出しスタックのサポート

この段階で、サブプログラム呼び出しをステップスルーして、`get_value/set_value` オプションを使用してサブプログラム内の自動変数およびスタティック変数にアクセスできるようになりました。

現在のところ、これらの変数にはサブプログラムが呼び出しスタックの一番上にある場合にのみアクセスできます。

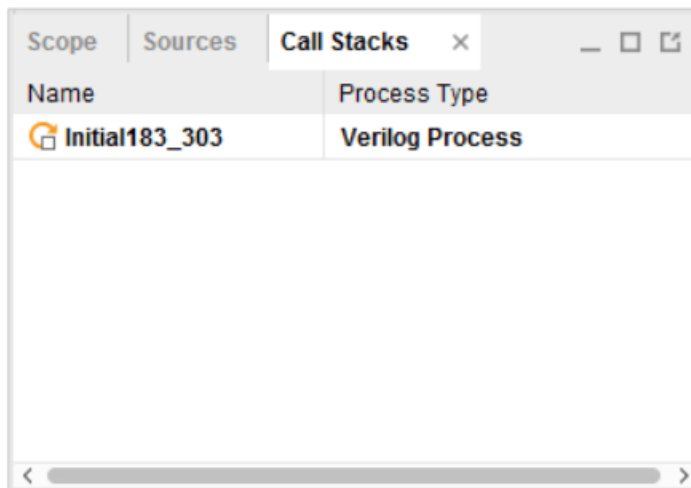
次のオプションを使用すると、呼び出しスタックのどのレベルの変数にもアクセスできます。

[Call Stacks] ウィンドウ

[Call Stacks] ウィンドウには、現在のシミュレーション時間でサブプログラム内で待機中の VHDL/Verilog プロセスすべての HDL スコープが表示されます。これは `get_stacks` Tcl コマンドに類似しています。

デフォルトでは、[Call Stacks] ウィンドウでシミュレーションが停止する箇所 (サブプログラム内) のプロセスが選択されますが、サブプログラムで待機中のどのプロセスでも選択できます。[Call Stacks] ウィンドウでプロセスを選択するのは、[Scope] ウィンドウからプロセス スコープを選択したり、`current_scope` Tcl コマンドを使用したりするのと同じことです。[Call Stacks] ウィンドウでプロセスを選択すると、アップデートされたプロセスが [Scope] ウィンドウ、[Objects] ウィンドウ、[Stack Frames] ウィンドウ、および [Locals] タブに表示されます。[Call Stacks] ウィンドウには、プロセス名と選択したプロセスの絶対パスとそのタイプが表示されます。

図 44: [Call Stacks] ウィンドウ



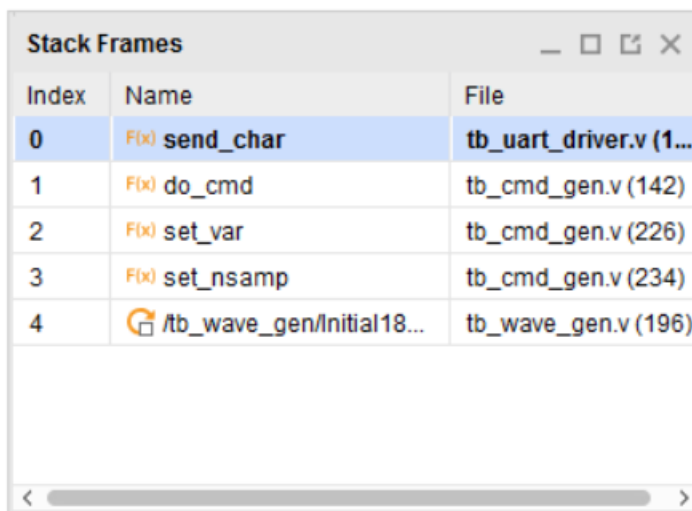
[Stack Frames] ウィンドウ


[Stack Frames] ウィンドウには、サブプログラム内で待機中の現在の HDL プロセスとその呼び出しスタック内のサブプログラムが表示されます。これは `report_frames` および `current_frame Tcl` コマンドに類似しています。

[Stack Frames] ウィンドウには、現在の階層の最も最近のサブプログラムが一番上に、次に呼び出し元のサブプログラムが表示されます。呼び出し元の HDL プロセスは一番下に表示されます。その他のフレームを現在のフレームとして選択することもできます。これは、`current_frame -set <selected_frame_index> Tcl` コマンドでも設定できます。

[Objects] ウィンドウの [Locals] タブには、サブプログラムのフレーム選択に従い、選択したサブプログラムフレームに対してローカルのスタティックおよび自動変数が表示されます。[Stack Frames] ウィンドウには、フレーム数、サブプログラム/プロセス名、ソース ファイル、選択した HDL プロセスの現在の行が表示されます。

図 45: [Stack Frames] ウィンドウ

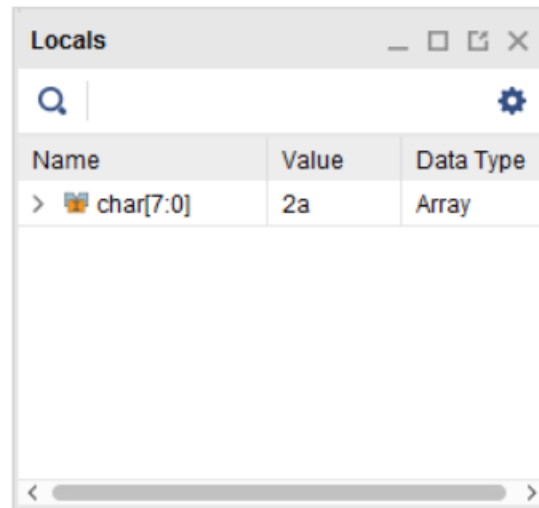


Index	Name	File
0	F(x) send_char	tb_uart_driver.v (1...
1	F(x) do_cmd	tb_cmd_gen.v (142)
2	F(x) set_var	tb_cmd_gen.v (226)
3	F(x) set_nsamp	tb_cmd_gen.v (234)
4	 /tb_wave_gen/Initial18...	tb_wave_gen.v (196)

[Objects] ウィンドウの [Locals] タブ

[Objects] ウィンドウの [Locals] タブには、現在実行中 (または選択された) サブプログラムのローカルのスタティックおよび自動変数の名前、値、およびタイプが表示されます。これは `get_objects -local Tcl` コマンドに類似しています。このウィンドウは、[Stack Frames] ウィンドウで選択したフレームに従って表示されます。各変数/引数に対して、[Locals] タブに名前、値、タイプが表示されます。

図 46: [Objects] ウィンドウの [Locals] タブ



ダイナミック型を使用したデバッグ

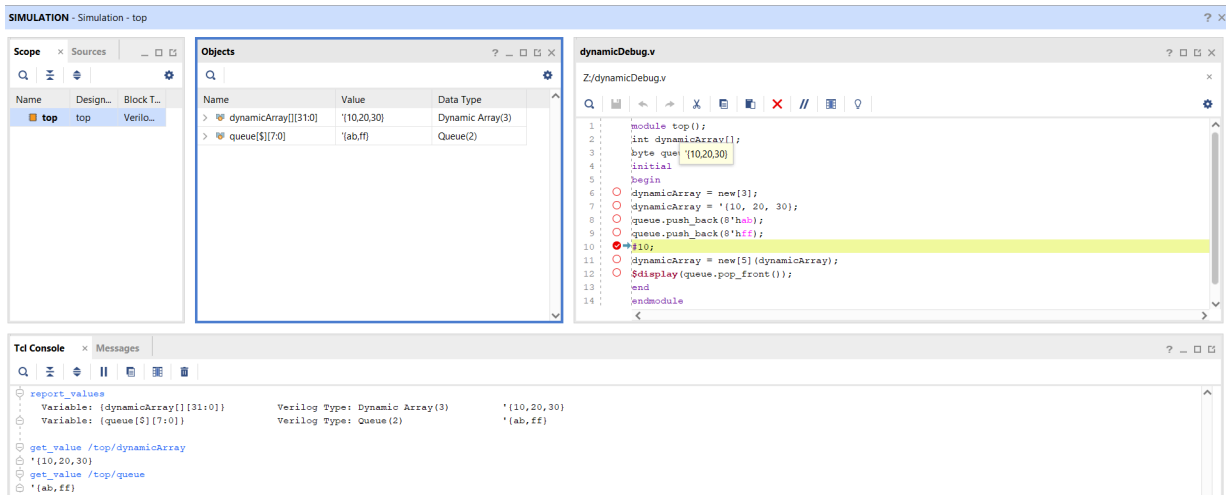
SystemVerilog には、クラス、動的配列、キュー、連想配列などのダイナミック型があり、Vivado シミュレータ (XSim) でサポートされています。Vivado では、ダイナミック型変数をプローブできます。次に例を示します。

```
module top();
  int dynamicArray[];
  byte queue[$];
  initial
  begin
    dynamicArray = new[3];
    dynamicArray = '{10, 20, 30};
    queue.push_back(8'h0a);
    queue.push_back(8'hff);
    #10;
    dynamicArray = new[5](dynamicArray);
    $display(queue.pop_front());
  end
endmodule
```

ダイナミック型変数は、次を使用してプローブできます (次の図を参照)。

- [Objects] ウィンドウ
- [Tcl Console] ウィンドウで `get_value` および `report_value` コマンドを使用。
- [Sources] ウィンドウのツール ヒント

図 47: ダイナミック型のプロープ



注記: ダイナミック型は、波形のトレース (add_wave) または波形データベースの作成 (log_wave) ではサポートされません。

Vivado シミュレータでのバッチまたはスクリプト モードを使用したシミュレーション

この章では、コマンド ラインでのコンパイルおよびシミュレーション プロセスについて説明します。

Vivado では統合シミュレーション フローがサポートされており、IDE から Vivado シミュレータまたはサードパーティ シミュレータを起動できますが、システム レベルのシミュレーションや UVM のような高度な検証など、検証環境でシミュレーションをバッチ モードまたはスクリプト モードを実行する場合も多くあります。Vivado Design Suite では、Vivado シミュレータでのバッチまたはスクリプト モードのシミュレーションがサポートされます。

この章では、必要なデザイン ファイルを集めてターゲット シミュレータ用のシミュレーション スクリプトを生成し、バッチ モードでシミュレーションを実行するプロセスについて説明します。シミュレーション スクリプトは、最上位 HDL デザインに対して、または階層モジュール、Manage IP プロジェクト、Vivado IP からのブロック デザイン用に生成できます。バッチ シミュレーションは、プロジェクト フローと非プロジェクト スクリプト ベース フローの両方でサポートされます。

シミュレーション ファイルとスクリプトのエクスポート

ビヘイビアー シミュレーションまたはタイミング シミュレーションをコマンド ラインから実行するには、次の手順を実行する必要があります。

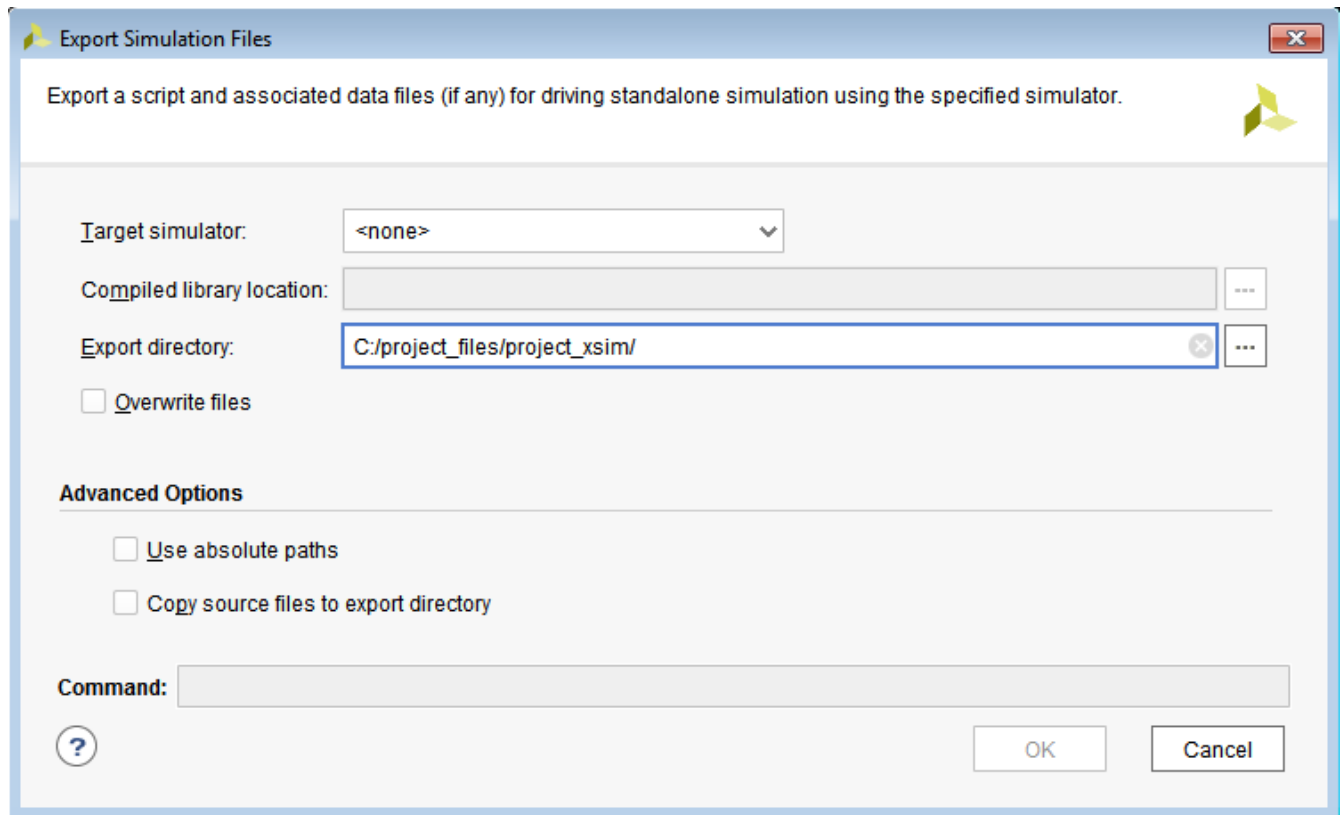
1. デザイン ファイルの識別と解析。
2. 実行可能なシミュレーション スナップショットのエラボレートと生成。
3. 実行可能なスナップショットを使用したシミュレーションの実行。

Vivado Design Suite の [Export Simulation] コマンドを使用すると、シミュレーションに必要なデザイン ファイルをすばやく集めて、最上位 RTL デザインまたは下位デザインのシミュレーション スクリプトを生成できます。

`export_simulation` コマンドでは、サポートされるサードパーティ シミュレータすべて、またはユーザーの選択したターゲット シミュレータのスクリプトが生成されます。

Vivado IDE で [File] → [Export] → [Export Simulation] をクリックし、次の図に示す [Export Simulation] ダイアログ ボックスを開きます。

図 48: [Export Simulation] ダイアログ ボックス



[Export Simulation] コマンドは、サポートされるすべてのシミュレータまたは指定したターゲット シミュレータに対してシミュレーション スクリプト ファイルを生成します。生成されるスクリプトには、デザインをコンパイル、エラボレート、およびシミュレーションするシミュレータ コマンドが含まれます。

[Export Simulation] ダイアログ ボックスには、次のオプションが含まれます。

- [Target simulator]: コマンド ライン スクリプトを生成するシミュレータを指定します。Vivado シミュレータまたはサポートされているサードパーティ シミュレータを選択できます。詳細は、[第 3 章: サードパーティ シミュレータを使用したシミュレーション](#) を参照してください。


注記: Windows OS の場合は、Windows で実行されるシミュレータに対してのみスクリプトが生成されます。

- [Compiled library location]: [Export Simulation] コマンドで生成されるスクリプトを使用してシミュレーションを実行するには、[compile_simlib](#) Tcl コマンドを使用してシミュレーション ライブラリをコンパイルする必要があります。生成されるスクリプトでは、コンパイル ライブラリ ディレクトリからターゲット シミュレータに必要なセットアップ ファイルが自動的に含まれます。詳細は、[シミュレーション ライブラリのコンパイル](#) を参照してください。



ヒント: [Export Simulation] を実行する際は、常に [Compile library location] にパスを指定しておくことをお勧めします。これにより、スクリプトが常に正しいシミュレーション ライブラリをポイントするようになります。

- [Export directory]: [Export Simulation] で生成されるスクリプトの出力ディレクトリを指定します。デフォルトでは、シミュレーション スクリプトが現在のプロジェクトのローカル ディレクトリに出力されます。
- [Overwrite files]: `export` ディレクトリに同じ名前のファイルが既に存在している場合に、上書きします。
- [Use absolute paths]: デフォルトでは、生成されたスクリプトのソース ファイルとディレクトリ パスは、スクリプトで定義された基準ディレクトリに対する相対パスになります。このオプションを使用すると、スクリプト内のファイル パスは相対パスではなく絶対パスになります。

- [Copy source files to export directory]: デザイン ファイルを出力ディレクトリにコピーします。これにより、シミュレーション ソース ファイルと生成されたスクリプトがコピーされ、シミュレーション フォルダ全体が移植しやすくなります。
- [Command]: [Export Simulation] ダイアログ ボックスで指定したさまざまなオプションおよび設定により実行される `export_simulation` コマンドの Tcl コマンド構文を表示します。
- [Help]:  ボタンで [Export Simulation Files] ダイアログ ボックスのさまざまなオプションの説明を表示します。

[Export Simulation] コマンドでは、プロジェクトと非プロジェクトの両方のデザインがサポートされます。Verilog の `'defines` および `'include` ディレクトリのクエリ以外は現在のプロジェクトからのプロパティは読み込まれず、ダイアログ ボックスまたは `export_simulation` コマンド オプションから指示子が取得されます。必要な結果を取得するには、適切なオプションを指定する必要があります。また、最上位デザインで使用されるすべての IP およびブロック デザインの出力ファイルが必要です。



重要: IP およびブロック デザインの出力ファイルがない場合、`export_simulation` コマンドでは自動的に生成されず、エラー メッセージが表示され、実行が停止します。

[Export Simulation] ダイアログ ボックスで [OK] をクリックすると、指定したデザイン オブジェクト (最上位デザイン、階層モジュール、IP コア、Vivado IP インテグレーターからのブロック デザイン、または複数 IP を含む Manage IP プロジェクト) をシミュレーションするのに必要なデザイン ファイルすべてのシミュレーション コンパイル順が取得されます。必要なデザイン ファイルのシミュレーション コンパイル順は、ターゲット シミュレータのコンパイラ コマンドおよびオプションを含めてシェル スクリプトにエクスポートされます。

シミュレーション スクリプトは [Export Simulation] ダイアログ ボックスで指定したエクスポート ディレクトリの個別フォルダーに生成されます。指定したシミュレータごとに個別のフォルダーが作成され、そのシミュレータ用の `compile`、`elaborate`、および `simulate` スクリプトが記述されます。

[Export Simulation] コマンドで生成されたスクリプトでは、3 段階のプロセス (解析/コンパイル、エラボレート、シミュレーション) が使用されます。これは、Vivado シミュレータを含め、多くのシミュレータに共通しています。ModelSim では、生成されたスクリプトでツールが必要とする 2 段階のプロセス (コンパイルとシミュレーション) が使用されます。



ヒント: Questa または Aldec シミュレータで 2 段階のプロセスを使用するには、ModelSim 用に生成されたスクリプトを必要に応じて変更してください。

[Export Simulation] コマンドでは、ファイルセットまたは IP からのデータ ファイルが指定したエクスポート ディレクトリにコピーされます。デザインに Verilog ソースが含まれる場合は、生成されるスクリプトにより `glbl.v` も Vivado ソフトウェア インストール パスから出力ディレクトリにコピーされます。

```
export_ip_user_files -no_script -force
export_simulation -directory "C:/Data/project_wave1" -simulator all
```

ダイアログ ボックスから [Export Simulation] コマンドを実行すると、Vivado IDE でエクスポートされたスクリプトのベース ディレクトリを定義するコマンド シーケンスが実際に実行され、IP ユーザー ファイルがエクスポートされてから、`export_simulation` コマンドが実行されます。

Vivado IDE で `export_ip_user_files` コマンドが自動的に実行され、コア コンテナおよびコア コンテナ以外の IP 両方のシミュレーションをサポートするのに必要なすべてのファイルとブロック デザインが使用可能になります。詳細は、『Vivado Design Suite ユーザー ガイド: IP を使用した設計』(UG896) のこのセクションを参照してください。`export_ip_user_files` は [Export Simulation] ダイアログ ボックスを使用すると自動的に実行されますが、`export_simulation` コマンドを使用する場合は実行前に必ず手動で実行するようにしてください。



ヒント: Vivado IDE で `export_ip_user_files` コマンドが自動的に実行される際、`-no_script` オプションが指定されます。これは、最上位デザインで使用される個別の IP およびブロック デザインに対してシミュレーション スクリプトが生成されるとコマンドの実行時間がかなり長くなるので、生成されないようにするためです。個別の IP およびブロック デザインのシミュレーション スクリプトを生成する場合は、特定のオブジェクト (`export_ip_user_files`) に対して `-of_objects` を実行するか、`-no_script` オプションを使用しないようにします。

`export_ip_user_files` コマンドでシミュレーションおよび合成に必要な IP およびブロック デザインのユーザー ファイル環境が設定され、インスタンス化テンプレート、サードパーティ合成ツールで使用するスタブ ファイル、ラッパー ファイル、メモリ初期化ファイル、およびシミュレーション スクリプトを含む `ip_user_files` というフォルダーが作成されます。

`export_ip_user_files` コマンドでは、プロジェクト内のすべての IP およびブロック デザインで共有されるスタティック シミュレーション ファイルも統合され、`ipstatic` フォルダーにコピーされます。プロジェクト内の複数の IP およびブロック デザイン間で共有される IP ファイルの多くは、特定のカスタマイズ IP 用に変更されることはありません。これらのスタティック ファイルは `ipstatic` ディレクトリにコピーされます。シミュレーション用に作成されたスクリプトは、このディレクトリの共有ファイルを必要に応じて参照します。カスタマイズ IP に特有のダイナミック シミュレーション ファイルは、IP フォルダーにコピーされます。詳細は、『Vivado Design Suite ユーザー ガイド: IP を使用した設計』 (UG896) の [このセクション](#) を参照してください。



重要: `export_simulation` コマンドで生成されるスクリプトおよびファイルでは `ip_user_files` ディレクトリのファイルが指定されます。`export_ip_user_files` コマンドよりも前に `export_simulation` を実行しないと、シミュレーション エラーが発生する可能性があります。

最上位デザインのエクスポート

最上位 RTL デザインのシミュレーション スクリプトを作成するには、`export_simulation` コマンドを使用し、シミュレーション ファイル オブジェクトを指定します。次の例では、シミュレーション ファイルセットは `sim_1` で、`export_simulation` によりデザイン内のすべての RTL エンティティ、IP、およびブロック デザイン オブジェクトに対してシミュレーション スクリプトが作成されます。

```
export_ip_user_files -no_script
export_simulation -of_objects [get_filesets sim_1] -directory C:/test_sim \
-simulator questa
```

ザイリンクス カタログおよびブロック デザインからの IP のエクスポート

IP または Vivado IP インテグレーター ブロック デザインのスクリプトを生成するには、IP またはブロック デザイン オブジェクトに対してコマンドを実行します。

```
export_ip_user_files -of_objects [get_ips blk_mem_gen_0] -no_script -force
export_simulation -simulator ies -directory ./export_script \
-of_objects [get_ips blk_mem_gen_0]
```

または、デザイン内のすべての IP およびブロック デザインの `ip_user_files` をエクスポートします。

```
export_ip_user_files -no_script -force
export_simulation -simulator ies -directory ./export_script
```

ブロック デザイン オブジェクトのシミュレーション スクリプトを生成することもできます。

```
export_ip_user_files -of_objects [get_files base_microblaze_design.bd] \
-no_script -force
export_simulation -of_objects [get_files base_microblaze_design.bd] \
-directory ./sim_scripts
```

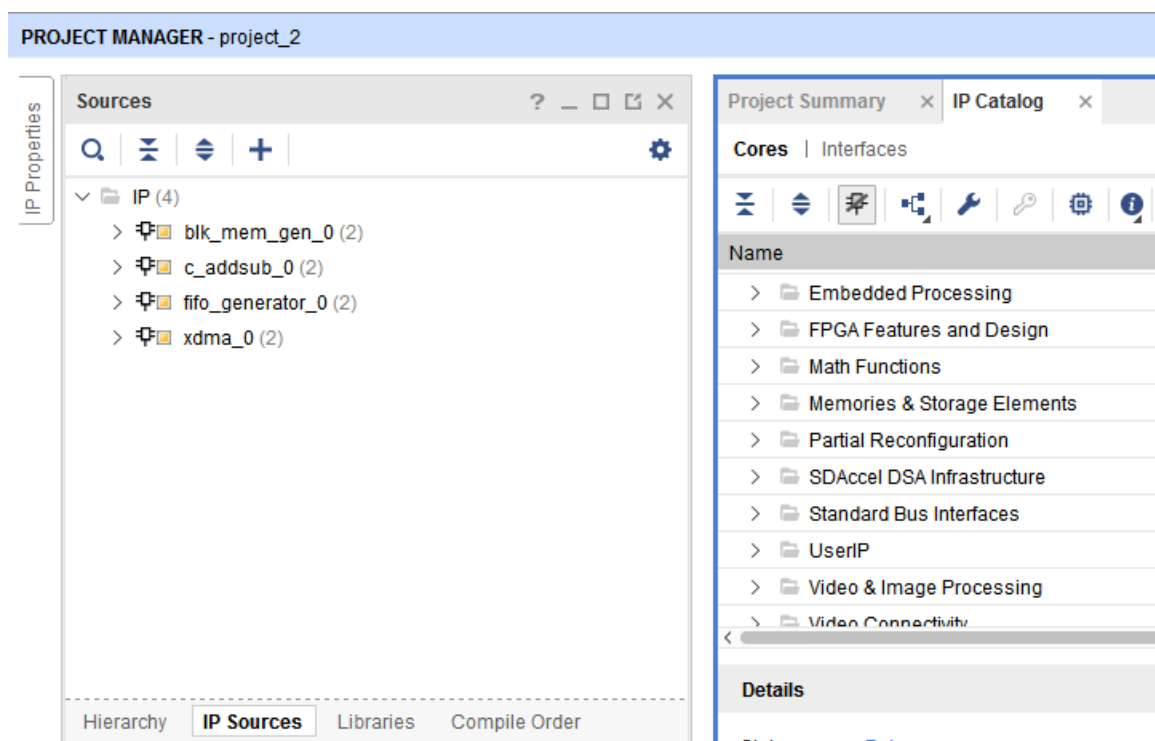


重要: 最上位デザインで使用されるすべての IP およびブロック デザインの出力ファイルが必要です。IP およびブロック デザインの出力ファイルがない場合、`export_simulation` コマンドでは自動的に生成されず、エラー メッセージが表示されるか、実行が停止します。

Manage IP プロジェクトのエクスポート

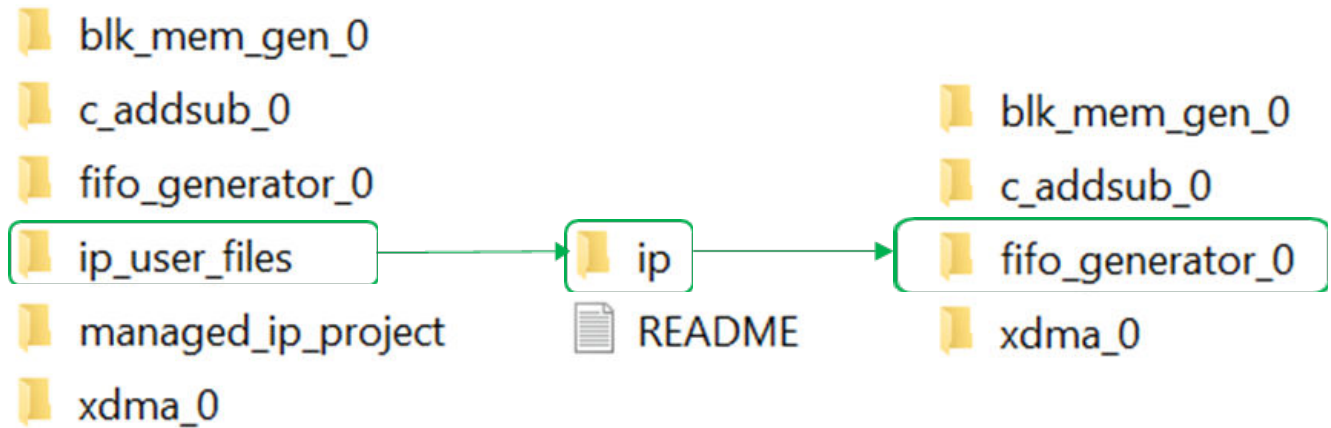
Manage IP プロジェクトには、カスタマイズした IP を集めたりリポジトリを作成して管理する機能があります。Manage IP プロジェクトの詳細は、『Vivado Design Suite ユーザー ガイド: IP を使用した設計』(UG896)の[このセクション](#)を参照してください。Manage IP プロジェクトの IP 出力ファイルを生成すると、前述のように、Vivado ツールで `export_ip_user_files` コマンドを使用して各 IP のシミュレーション スクリプトも生成されます。

図 49: Manage IP プロジェクト



上記の Manage IP プロジェクトには、`blk_mem_gen_0`、`c_addsub_0`、`fifo_generator_0`、`xdma_0` という 4 つのカスタマイズ IP が含まれています。このプロジェクトの場合、Vivado Design Suite で次の図に示すように `ip_user_files` フォルダが作成されます。

図 50: Manage IP のディレクトリ構造



`ip_user_files` フォルダは、前述のように `export_ip_user_files` コマンドを実行すると生成されます。Manage IP プロジェクトでこのコマンドを実行すると、プロジェクト内のすべての IP が処理され、IP の合成およびシミュレーションに必要なスクリプトおよびその他のファイルが生成されます。`ip_user_files` フォルダには、バッチシミュレーションに使用されるスクリプトのほか、シミュレーションのサポートに必要なダイナミックおよびスタティック IP ファイルが含まれます。

ユーザーのターゲットシミュレータまたはすべてのサポートされるシミュレータ用のシミュレーションスクリプトは、[Manage IP プロジェクトのエクスポート](#) に示すように `./sim_scripts` に含まれます。ターゲットシミュレータのフォルダに移動し、`compile`、`elaborate`、および `simulate` スクリプトをシミュレーションフローに組み込むことができます。

Vivado ツールでは、デザイン内の複数の IP およびブロックデザインで使用されるすべての共有シミュレーションファイルが `./ipstatic` という 1 つのフォルダにまとめられます。カスタマイズ IP の特性によって異なるダイナミックファイルは、`./ip` フォルダに含まれます。



ヒント: Manage IP プロジェクトのすべての IP をエクスポートするだけでなく、[ザイリンクス カタログおよびプロジェクトデザインからの IP のエクスポート](#) の手順を使用すると、プロジェクトで個別 IP のスクリプトを生成できます。

バッチ モードでの Vivado シミュレータの実行

バッチ モードまたはスクリプト モードで実行するため、Vivado シミュレータは `export_simulation` コマンドで生成されるファイルでサポートされる 3 つのプロセスに依存します。

- デザイン ファイル、`xvhdl` および `xvlog` の解析
- デザイン スナップショットのエラボレートおよび生成 (`xelab`)
- デザイン スナップショットのシミュレーション (`xsim`)

タイミングシミュレーションの場合は、シミュレーションを終了するのに次の追加手順およびデータが必要となります。

- タイミング ネットリストの生成
- 合成後およびインプリメンテーション後のシミュレーション

デザイン ファイル、xvhdl および xvlog の解析

xvhdl および xvlog コマンドは、それぞれ VHDL および Verilog ファイルを解析します。各オプションの詳細は、[表 16: xelab、xvhdl、xvlog のコマンド オプション](#) を参照してください。

xvhdl

xvhdl コマンドは VHDL アナライザー (パーサー) です。

xvhdl の構文

```
xvhdl
[-encryptdumps]
[-f [-file] <filename>]
[-h [-help]]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-nolog]
[-prj <filename>]
[-relax]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=<library_dir>]]
[-incr]
[-2008]
[-93_mode]
[-nosignalhandlers]
```

このコマンドは VHDL ソース ファイルを解析し、解析済みデータをディスクの HDL ライブラリに保存します。

xvhdl の例

```
xvhdl file1.vhd file2.vhd
xvhdl -work worklib file1.vhd file2.vhd
xvhdl -prj files.prj
```

xvlog

xvlog コマンドは Verilog パーサーです。xvlog Verilog ソース ファイルを解析し、解析済みデータをディスクの HDL ライブラリに保存します。

xvlog の構文

```
xvlog
[-d [define] <name>[=<val>]]
[-encryptdumps]
[-f [-file] <filename>]
[-h [-help]]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-nolog]
[-noname_unamed_generate]
[-relax]
[-prj <filename>]
```

```
[ -sourcelibdir <sourcelib_dirname>]
[ -sourcelibext <file_extension>]
[ -sourcelibfile <filename>]
[ -sv]
[ -v [verbose] [0|1|2]]
[ -version]
[ -work <library_name> [=<library_dir>]
[ -incr]
[ -nosignalhandlers]
[ -uvm_version arg]
```

xvlog の例

```
xvlog file1.v file2.v
xvlog -work worklib file1.v file2.v
xvlog -prj files.prj
```

注記: xelab、xvlog、xvhdl は Tcl コマンドではありません。xvlog、xvhdl、xelab は Vivado に依存しないコンパイラ実行ファイルなので、対応する Tcl コマンドはありません。

シミュレーションの開始は Vivado に依存しているので、Tcl コマンドの `xsim` で実行します。

シミュレーションを Vivado 外で実行するため、`xsim` と同じ名前の実行ファイルが提供されます。`xsim` 実行ファイルは、Vivado を非プロジェクト モードで起動し、`xsim` Tcl コマンドを実行してシミュレーションを開始します。このため、Vivado IDE 内から `xvlog`、`xvhdl`、`xelab` フォームのヘルプを表示する場合は、コマンドの前に `exec` を付けてください。

例: `exec xvlog -help`

`xsim` のヘルプを表示するには、`xsim -help` と入力します。

デザイン スナップショットのエラボレートおよび生成 (xelab)

Vivado シミュレータを使用するシミュレーションは、次の 2 段階で実行されます。

- 最初の段階では、シミュレータ コンパイラ `xelab` により、HDL モデルがスナップショットにコンパイルされます。スナップショットは、シミュレータが実行できる形式でモデルを表記したものです。
- 次に、そのモデルをシミュレーションするため、シミュレータでそのスナップショットが読み込まれ、実行されます (`xsim` コマンドを使用)。非プロジェクト モードでは、最初の段階を飛ばして 2 番目の段階のみを繰り返すと、スナップショットを再利用できます。

シミュレータでスナップショットが作成されると、モデルの最上位モジュールの名前に基づいてスナップショット名が付けられますが、コンパイラに対するオプションとしてスナップショット名を指定することもできます。スナップショット名は、ディレクトリまたは SIMSET 内で重複しないようにします。デフォルトでもカスタムでも、同じスナップショット名を使用すると、前に生成されたスナップショットが上書きされます。



重要: 同じディレクトリまたは SIMSET 内で同じスナップショット名を使用して、2 つのシミュレーションを実行することはできません。

xelab

xelab コマンドは、指定した最上位ユニットに対して次を実行します。

- 言語結合ルールおよび `-L <library>` コマンド ラインで指定した HDL ライブラリのいずれかを使用して、下位 デザイン ユニットを読み込みます。
- デザインのスタティック エラボレーション (パラメーターの設定、ジェネリックの処理、generate 文の実行など) を実行します。
- 実行可能なコードを生成します。
- 生成された実行可能コードをシミュレーション カーネル ライブラリにリンクして、実行可能なシミュレーション スナップショットを作成します。

この後、`xsim` コマンドのほかのオプションと共に、出力された実行可能なシミュレーションのスナップショット名をオプションとして指定します。



ヒント: `xelab` は解析コマンドの `xvlog` および `xvhdl` を暗示的に呼び出すことができます。解析手順を組み込むには、`xelab -prj` オプションを使用します。プロジェクト ファイルの詳細は、[プロジェクト ファイル \(.prj\) の構文](#)を参照してください。

注記: `xelab`、`xvlog`、`xvhdl` は Tcl コマンドではありません。`xvlog`、`xvhdl`、`xelab` は Vivado に依存しないコンパイラ実行ファイルなので、対応する Tcl コマンドはありません。

xelab コマンド構文オプション

各オプションの詳細は、次のコードを参照してください。

```
xelab
[-d [define] <name>[=<val>]
[-debug <kind>]
[-f [-file] <filename>]
[-generic_top <value>]
[-h [-help]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-log <filename>]
[-L [-lib] <library_name> [=<library_dir>]
[-maxdesigndepth arg]
[-mindelay]
[-typdelay]
[-maxarraysize arg]
[-maxdelay]
[-mt arg]
[-nolog]
[-noname_unnamed_generate]
[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
[-O arg]
[-override_timeunit]
[-override_timeprecision]
[-prj <filename>]
[-pulse_e arg]
[-pulse_r arg]
[-pulse_int_e arg]
[-pulse_int_r arg]
[-pulse_e_style arg]
[-r [-run]]
```

```

[-R [-runall]]
[-rangecheck]
[-relax]
[-s [-snapshot] arg]
[-sdfnowarn]
[-sdfnoerror]
[-sdfroot <root_path>]
[-sdfmin arg]
[-sdfmax arg]
[-sdfmax arg]
[-sourceelibdir <sourceelib_dirname>]
[-sourceelibext <file_extension>]
[-sourceelibfile <filename>]
[-stats]
[-timescale]
[-timeprecision_vhdl arg]
[-transport_int_delays]
[-v [verbose] [0|1|2]]
[-version]
[-sv_root arg]
[-sv_lib arg]
[-sv_liblist arg]
[-dpiheader arg]
[-driver_display_limit arg]
[-dpi_absolute]
[-incr]
[-93_mode]
[-nosignalhandlers]
[-dpi_stacksize arg]
[-transform_timing_checkers]
[-a[ --standalone]
[-ignore_assertions]
[-ignore_coverage]
[-cov_db_dir arg]
[-cov_db_name arg]
[-uvm_version arg]
[-report_assertion_pass]
[-dup_entity_as_module]

```

xelab の例

```

xelab work.top1 work.top2 -s cpusim
xelab lib1.top1 lib2.top2 -s fftsim
xelab work.top1 work.top2 -prj files.prj -s pciesim
xelab lib1.top1 lib2.top2 -prj files.prj -s ethernetstim

```

Verilog 検索順序

xelab コマンドは、インスタンス化された Verilog デザイン ユニットを検索および結合する際、次の検索順を使用します。

1. Verilog コードの 'uselib 指示子で指定されたライブラリ。次に例を示します。

```

module
full_adder(c_in, c_out, a, b, sum)
input c_in,a,b;
output c_out,sum;
wire carry1,carry2,sum1;

```



```
`uselib lib = adder_lib
half_adder adder1(.a(a),.b(b),.c(carry1),.s(sum1));
half_adder adder1(.a(sum1),.b(c_in),.c(carry2),.s(sum));
c_out = carry1 | carry2;
endmodule
```

2. -lib|-L オプションを使用してコマンド ラインで指定されたライブラリ。
3. 親デザイン ユニットのライブラリ。
4. work ライブラリ。

Verilog インスタンスレーション ユニット

Verilog デザインにコンポーネントがインスタンスレーションされると、`xelab` コマンドによりコンポーネント名が Verilog ユニットとして処理され、ユニファイド論理ライブラリのユーザー指定のリストでユーザーの指定順に Verilog モジュールが検索されます。

- ユニットが見つかったら、`xelab` により結合され、検索が停止します。
- 大文字/小文字を区別した検索で何も見つからない場合、`xelab` により、ユニファイド論理ライブラリのユーザー指定のリストでユーザーの指定順に、拡張識別子として構築された VHDL デザイン ユニット名が大文字/小文字を区別せずに検索され、最初に一致したものが選択されて検索が停止します。
- `xelab` が 1 つのライブラリ特有の結合を見つけた場合は、その名前が選択されて検索が停止します。

注記: 混合言語デザインの場合、Verilog モジュールでインスタンスレーションされる VHDL エンティティへの関連付けに使用されるポート名では、常に大文字/小文字が区別されます。また、VHDL ジェネリックを変更するのに `defparam` 文は使用できないことにも注意してください。詳細は、[混合言語シミュレーションの使用](#)を参照してください。



重要: VHDL レコード オブジェクト全体の Verilog オブジェクトへの接続はサポートされていません。

VHDL インスタンスレーション ユニット

VHDL デザインにコンポーネントがインスタンスレーションされると、`xelab` コマンドによりそのコンポーネント名が VHDL ユニットとして処理され、論理 `work` ライブラリで検索されます。

- VHDL ユニットが見つかったら `xelab` 結合され、検索が停止します。
- `xelab` VHDL ユニットが見つからない場合、大文字/小文字の保持されたコンポーネント名が Verilog モジュール名として処理され、ユーザーの指定したユニファイド論理ライブラリのリスト (ユーザーの指定順) で大文字/小文字を区別した検索が実行されます。最初に一致したものが選択されて検索が停止します。
- 大文字/小文字を区別した検索で何も見つからない場合、`xelab` ユーザーの指定したユニファイド論理ライブラリのリスト (ユーザーの指定順) で、Verilog モジュールが大文字/小文字を区別せずに検索されます。1 つのライブラリ特有の結合が見つかった場合は、その名前が選択されて、検索が停止します。

Verilog の `uselib 指示子

ライブラリ検索順を設定する Verilog の ``uselib` 指示子がサポートされています。

`uselib の構文

```
<uselib compiler directive> ::= `uselib [<Verilog-XL uselib directives>|
<lib
directive>]
<Verilog-XL uselib directives> ::= dir = <library_directory> | file =
<library_file>
| libext = <file_extension>
<lib directive> ::= <library reference> {<library reference>}
<library reference> ::= lib = <logical library name>
```

`uselib lib 指示子

`uselib lib 指示子は、Verilog-XL の `uselib 指示子とは一緒に使用できません。たとえば、次のコードは無効です。

```
`uselib dir=./ file=f.v lib=newlib
```

1 つの `uselib 指示子で複数のライブラリを指定できます。

ライブラリを指定した順が検索順になります。次に例を示します。

```
`uselib lib=mylib lib=yourlib
```

この場合、インスタンスエートされたモジュールの検索がまず mylib で実行され、次に yourlib で実行されます。

`uselib dir、`uselib file、`uselib libext などの指示子と同様、解析の 1 つのセッションでは `uselib lib 指示子が HDL ファイルすべてに適用されます。別の `uselib 指示子があるまでは、HDL ソースの `uselib (Verilog XL `uselib を含む) 指示子を使用されます。引数なしで `uselib を使用すると、現在アクティブな `uselib <lib|file|dir|libext> が無効になります。

Verific の Verilog エラレーション アルゴリズムでのインスタンスエート済みモジュールまたは UDP の解決には、次のモジュール検索メカニズムが使用されます。

- まず、現在アクティブな `uselib lib の論理ライブラリの順序リストでインスタンスエート済みモジュールが検索されます。
- 見つからない場合は、xelab コマンドで検索ライブラリとして指定されているライブラリの順序リストでインスタンスエート済みモジュールが検索されます。
- 見つからない場合は、親モジュールのライブラリでインスタンスエート済みモジュールが検索されます。たとえば、work ライブラリのモジュール A に mylib ライブラリのモジュール B がインスタンスエートされ、モジュール B にモジュール C がインスタンスエートされている場合、/mylib ライブラリ (C の親モジュールである B のライブラリ) でモジュール C が検索されます。
- 見つからない場合は、次のいずれかの work ライブラリでインスタンスエートされたモジュールを検索します。
 - HDL ソースがコンパイルされるライブラリ
 - work ライブラリとして明示的に設定されるライブラリ
 - work という名前のデフォルトの作業ライブラリ

`uselib の例

表 15: `uselib の例

adder_lib という名前の論理ライブラリにコンパイルされる half_adder.v ファイル	work という名前の論理ライブラリにコンパイルされる full_adder.v ファイル
<pre>module half_adder(a,b,c,s); input a,b; output c,s; s = a ^ b; c = a & b; endmodule</pre>	<pre>module full_adder(c_in, c_out, a, b, sum) input c_in,a,b; output c_out,sum; wire carry1,carry2,sum1; `uselib lib = adder_lib half_adder adder1(.a(a)..b(b)..c(carry1)..s(sum1)); half_adder adder1(.a(sum1)..b(c_in)..c(carry2)..s(sum)); c_out = carry1 carry2; endmodule</pre>

xelab、xvhdl、xvlog xsim コマンド オプション

次の表に、xelab、xvhdl、および xvlog xsim コマンドのコマンド オプションを示します。

表 16: xelab、xvhd、xvlog のコマンド オプション

コマンド オプション	説明	使用するコマンド
-d [define] <name>[=<val>]	Verilog マクロを定義します。各 Verilog マクロに -d --define を使用します。マクロのフォーマットは <name>[=<val>] で、<name> はマクロの名前、<value> はオプションのマクロの値です。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-debug <kind>	指定したデバッグ機能をオンにしてコンパイルします。<kind> に有効な値は、次のとおりです。 <ul style="list-style-type: none"> typical: 最も一般的な機能で、line および wave が含まれます。 line: HDL ブレークポイント。 wave: 波形の生成、条件付き実行、強制値。 xlibs: ザイリンクスのコンパイル済みライブラリの可視性。このオプションは、コマンド ラインでのみ使用できます。 off: すべてのデバッグ機能をオフにします (デフォルト)。 all: すべてのデバッグ オプションを使用します。 	xelab
-encryptdumps	コンパイルされるデザイン ユニットの解析済みダンプを暗号化します。	デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析

表 16: xelab、xvhdl、xvlog のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
-f [-file] <filename>	指定したファイルから追加オプションを読み出します。	xelab xsim のオプション デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-generic_top <value>	最上位デザイン ユニットのジェネリックまたはパラメーターを指定した値に置き換えます。例: -generic_top "P1=10"	xelab
-h [-help]	ヘルプ メッセージを表示します。	xelab xsim のオプション デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-i [include] <directory_name>	Verilog の <code>`include</code> を使用して含めたファイルを検索するディレクトリを指定します。各検索ディレクトリに <code>-i --include</code> を使用します。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-initfile <init_filename>	デフォルトの <code>xsim.ini</code> ファイルで提供される設定に追加、またはそれらの設定の代わりに使用する設定を含むユーザー定義のシミュレータ初期化ファイルを指定します。	xelab デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-L [-lib] <library_name> [=<library_dir>]	インスタンス化された VHDL 以外のデザイン ユニット (Verilog デザイン ユニットなど) の検索ライブラリを指定します。 各検索ライブラリに <code>-L --lib</code> を使用します。引数のフォーマットは <code><name>[=<dir>]</code> で、 <code><name></code> はライブラリの論理名、 <code><library_dir></code> はライブラリのオプションの物理ディレクトリです。	xelab デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-log <filename>	ログ ファイルの名前を指定します。デフォルトは <code><xvlog xvhdl xelab xsim>.log</code> です。	xelab xsim のオプション デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-maxarraysize <arg>	VHDL 配列の最大サイズを 2^{*n} に設定します。デフォルトは $n=28$ 、つまり 2^{*28} です。	xelab
-maxdelay	Verilog デザイン ユニットを最大遅延でコンパイルします。	xelab
-maxdesigndepth <arg>	エラボーレーターで許容されるデザイン階層深さの最大値を変更します。デフォルトは 5000 です。	xelab
-maxlogsize <arg> (=-1)	ログ ファイルの最大サイズを MB で指定します。デフォルトは無制限です。	xsim のオプション

表 16: xelab、xvhdl、xvlog のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
-mindelay	Verilog デザイン ユニットを最小遅延でコンパイルします。	xelab
-mt <arg>	<p>並列実行可能なサブコンパイル ジョブの数を指定します。有効な値は、auto、off、または 1 より大きい integer です。</p> <p>auto に設定すると、xelab でホスト マシンの CPU 数に基づいて並列ジョブの数が選択されます。デフォルトは auto です。</p> <p>-mt オプションをさらに詳細に制御するには、次のように Tcl プロパティを設定します。</p> <pre>set_property XELAB.MT_LEVEL off N [get_filesets sim_1]</pre>	xelab
-nolog	ログ ファイルが生成されないようにします。	xelab xsim の構文 デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-noieeewarnings	VHDL IEEE 機能からの警告をオフにします。	xelab
-noname_unnamed_generate	名前の付いていない生成ブロックに対して名前を付けないようにします。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-notimingchecks	Verilog 指定ブロックでタイミング チェック コンストラクトを無視します。	xelab
-nosdfinterconnectdelays	SDF の SDF ポートおよびインターコネクト遅延コンストラクトを無視します。	xelab
-nospecify	Verilog パス遅延とタイミング チェックを無視します。	xelab
-O <arg>	<p>最適化をイネーブルまたはディスエーブルにします。</p> <ul style="list-style-type: none"> -O 0: 最適化をディスエーブル -O 1: 基本的な最適化をイネーブル -O 2: 最もよく使用される最適化をイネーブル (デフォルト) -O 3: アドバンス最適化をイネーブル <p>注記: 値が小さいほどコンパイル時間は短くなりますが、シミュレーションは低速になります。値が大きいほどコンパイル時間は長くなりますが、シミュレーションは高速になります。</p>	xelab
-override_timeunit	すべての Verilog モジュールの時間単位を -timescale オプションで指定した値に置き換えます。	xelab
-override_timeprecision	すべての Verilog モジュールの時間精度を -timescale オプションで指定した時間精度に置き換えます。	xelab
-pulse_e <arg>	パス パルス エラー制限をパス遅延のパーセントで指定します。有効な値は 0 ~ 100 です (デフォルトは 100)。	xelab

表 16: xelab、xvhdl、xvlog のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
-pulse_r <arg>	パス パルス拒否制限をパス遅延のパーセントで指定します。有効な値は 0 ~ 100 です (デフォルトは 100)。	xelab
-pulse_int_e arg	インターコネクト パルス拒否制限を遅延のパーセントで指定します。有効な値は 0 ~ 100 です (デフォルトは 100)。	xelab
-pulse_int_r <arg>	インターコネクト パルス拒否制限を遅延のパーセントで指定します。有効な値は 0 ~ 100 です (デフォルトは 100)。	xelab
-pulse_e_style <arg>	パルスがモジュール パス遅延よりも短いことを示すエラーをいつレポートするかを指定します。有効な値は次のとおりです。 <ul style="list-style-type: none"> ondetect: 違反が検出されたときにエラーをレポートします。 onevent: モジュール パス遅延後にエラーをレポートします。 デフォルトは onevent です。	xelab
-prj <filename>	vhdl verilog <work lib> <HDL file name> の入力 1 つまたは複数含まれる Vivado シミュレータ プロジェクト ファイルを指定します。	xelab デザイン ファイル、xvhdl および xvlog の解析 デザイン ファイル、xvhdl および xvlog の解析
-r [-run]	生成された実行可能スナップショットをコマンド ラインの対話型モードで実行します。	xelab
-rangecheck	ランタイム値の範囲のチェックをイネーブルにします (VHDL)。	xelab
-R [-runall]	生成された実行可能スナップショット ファイルをシミュレーションの最後まで実行します。	xelab xsim の構文
-relax	厳密な言語規則を緩和します。	xelab デザイン ファイル、xvhdl および xvlog の解析 デザイン ファイル、xvhdl および xvlog の解析
-s [-snapshot] <arg>	出力されるシミュレーション スナップショットの名前を指定します。デフォルトは <worklib>.<unit> です (例: work.top)。追加のユニット名は # で連結します (例: work.t1#work.t2)。	xelab
-sdfnowarn	SDF 警告を出力しません。	xelab
-sdfnoerror	SDF ファイルで見つかったエラーを警告として処理します。	xelab
-sdfmin <arg>	<root=file> という形式で指定し、<file> という SDF ファイルの最小遅延を <root> にアノテートします。	xelab
-sdftyp <arg>	<root=file> という形式で指定し、<file> という SDF ファイルの標準遅延を <root> にアノテートします。	xelab
-sdfmax <arg>	<root=file> という形式で指定し、<file> という SDF ファイルの最大遅延を <root> にアノテートします。	xelab

表 16: xelab、xvhd、xvlog のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
-sdfroot <root_path>	SDF アノテーションを適用するデフォルトのデザイン階層を指定します。	xelab
-sourcelibdir <sourcelib_dirname>	コンパイルされていないモジュールの Verilog ソース ファイルのディレクトリを指定します。 各ソース ディレクトリに -sourcelibdir <sourcelib_dirname> を使用します。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-sourcelibext <file_extension>	コンパイルされていないモジュールの Verilog ソース ファイルのファイル拡張子を指定します。 ソース ファイル拡張子に -sourcelibext <file extension> を使用します。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-sourcelibfile <filename>	コンパイルされていないモジュールを含む Verilog ソース ファイルのファイル名を指定します。	xelab デザイン ファイル、 xvhdl および xvlog の 解析
-stat	CPU、メモリ使用量、デザイン統計を表示します。	xelab
-sv	入力ファイルを SystemVerilog モードでコンパイルします。	デザイン ファイル、 xvhdl および xvlog の 解析
-timescale	Verilog モジュールのデフォルトの時間単位を指定します。デフォルトは 1ns/1ps です。	xelab
-timeprecision_vhdl <arg>	VHDL デザインの時間精度を指定します。デフォルトは 1ps です。	xelab
-transport_int_delays	インターコネクト遅延に転送モデルを使用します。	xelab
-typdelay	Verilog デザイン ユニットを標準遅延 (デフォルト) でコンパイルします。	xelab
-v [verbose] [0 1 2]	表示メッセージの詳細レベルを指定します。デフォルトは 0 です。	xelab デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-version	コンパイラ バージョンを表示します。	xelab xsim のオプション デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-work <library_name> [=<library_dir>]	work ライブラリを指定します。この引数のフォーマットは <name>[=<dir>] です。 <ul style="list-style-type: none"><name>: ライブラリの論理名。<library_dir>: ライブラリの物理ディレクトリ (オプション)。	デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析
-sv_root <arg>	DPI ライブラリを含むルート ディレクトリを指定します。デフォルトは <current_directory>/xsim.dir/xsc> です。	xelab

表 16: xelab、xvhd、xvlog のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
-sv_lib <arg>	DPI インポートされた関数の共有ライブラリ名 (.dll/.so) をファイル拡張子なしで指定します。	xelab
-sv_liblist <arg>	DPI 共有ライブラリをポイントするブートストラップ ファイルを指定します。	xelab
-dpiheader <arg>	エクスポートおよびインポートされた関数のヘッダー ファイル名を指定します。	xelab
-driver_display_limit <arg>	指定した最大サイズまでの信号のドライバー デバッグをイネーブルにします (デフォルト: n = 65536)。	xelab
-dpi_absolute	Linux で lib<libname>.so という形式の DPI ライブラリに LD_LIBRARY_PATH ではなく絶対パスを使用します。	xelab
-incr	シミュレーションでインクリメンタル解析/エラボレーションをイネーブルにします。	デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析 xelab
-93_mode	VHDL を 93 モードでコンパイルします。	デザイン ファイル、 xvhdl および xvlog の 解析 xelab
-2008	VHDL を 2008 モードでコンパイルします。	デザイン ファイル、 xvhdl および xvlog の 解析
-nosignalhandlers	コンパイラでアンチウィルス、ファイアウォール信号を受信しないようにします。	デザイン ファイル、 xvhdl および xvlog の 解析 デザイン ファイル、 xvhdl および xvlog の 解析 xelab
-dpi_stacksize <arg>	DPI タスクのユーザー定義のスタック サイズ。	xelab
-transform_timing_checkers	タイミング チェッカーを Verilog プロセスに変換します。	xelab
-a	run-all を実行するスタンドアロンのインタラクティブではない実行ファイルを生成します。 常に -R と共に使用してください。 デバッグ機能なしでシミュレーションをより高速に実行するには、-standalone と -R を一緒に使用します。これにより、Vivado IDE を起動せずにシミュレーションがスタンドアロンで起動されます。このオプションを使用すると、ライセンスの読み込み時間が短縮されます。	xelab
-ignore_assertions	SystemVerilog の同時処理アサートを無視します。	xelab
-ignore_coverage	SystemVerilog の機能カバレッジを無視します。	xelab
-cov_db_dir <arg>	機能カバレッジ データベースのダンプを保存するディレクトリを指定します。カバレッジ データは、<arg>/xsim.covdb/<cov_db_name> ディレクトリに保存されます。デフォルトは ./ です。	xelab
-cov_db_name <arg>	機能カバレッジ データベースの名前を指定します。カバレッジ データは、<cov_db_dir>/xsim.covdb/<arg> ディレクトリに保存されます。デフォルトはスナップショット名です。	xelab

表 16: **xelab**、**xvhdl**、**xvlog** のコマンド オプション (続き)

コマンド オプション	説明	使用するコマンド
<code>-uvm_version <arg></code>	UVM バージョンを指定します。デフォルトは 1.2 です。	デザイン ファイル、 xvhdl および xvlog の解析 xelab
<code>-report_assertion_pass</code>	パス アクション ブロックがない場合でも、SystemVerilog 並列アサーション パスをレポートします。	xelab
<code>-dup_entity_as_module</code>	混合言語デザインの Verilog 階層内で階層参照のレポートをイネーブルにします。これにより、コンパイル速度が大幅に低下する可能性があるため、注意が必要です。	xelab

デザイン スナップショットのシミュレーション (xsim)

xsim コマンドは、シミュレーション スナップショットを読み込んで、バッチ モードのシミュレーションを実行するか、GUI または Tcl ベースの対話型シミュレーション環境を提供します。

xsim の構文

コマンド構文は、次のとおりです。

```
xsim <options> <snapshot>
```

説明:

- **xsim**: コマンド。
- **<options>**: 次の表に示すオプションを指定します。
- **<snapshot>**: シミュレーション スナップショット。

xsim のオプション

表 17: **xsim** コマンドのオプション

xsim オプション	説明
<code>-f [-file] <filename></code>	ファイルからコマンド ライン オプションを読み込みます。
<code>-g [-gui]</code>	対話型 GUI で実行します。
<code>-h [-help]</code>	ヘルプ メッセージを表示します。
<code>-log <filename></code>	ログ ファイルの名前を指定します。
<code>-maxdeltaid arg (=-1)</code>	最大差異を指定します。この値と最大シミュレーション ループを同時に超える場合にエラーをレポートします。
<code>-maxlogsize arg (=-1)</code>	ログ ファイルの最大サイズを MB で指定します。デフォルトは無制限です。
<code>-ieeewarnings</code>	VHDL IEEE 関数からの警告をイネーブルにします。
<code>-nolog</code>	ログ ファイルが生成されないようにします。

表 17: xsim コマンドのオプション (続き)

xsim オプション	説明
-nosignalhandlers	<p>シミュレーションでの OS レベル信号ハンドラーのインストールをディスエーブルにします。シミュレータでは、パフォーマンスの理由から、整数の 0 での除算など、OS レベルの致命的ランタイム エラーになる可能性のある状況は明示的にはチェックされません。その代わりに信号ハンドラーがインストールされ、このようなエラーを検出してレポートを生成します。</p> <p>信号ハンドラーをディスエーブルにすると、セキュリティ ソフトウェアが存在していてもシミュレーションを実行できますが、OS レベルの致命的エラーにより、エラーの原因がはっきりと示されないまま突然シミュレーションが停止してしまうことがあります。</p> <p> 注意: このオプションは、セキュリティ ソフトウェアによりシミュレータがうまく動作しない場合にのみ使用してください。</p>
-onfinish <quit stop>	シミュレーション終了時の動作を指定します。
-onerror <quit stop>	シミュレーション ランタイム エラーの発生した場合の動作を指定します。
-R [-runall]	シミュレーションを最後まで実行します (例: do 'run all;quit')。
-stats	終了するときにメモリおよび CPU の統計を表示します。
-testplusarg <arg>	plusargs が \$test\$plusargs および \$value\$plusargs システム関数で 사용되는ように指定します。
-t [-tclbatch] <filename>	バッチ モード実行用の Tcl ファイルを指定します。
-tp	実行中のプロセスの階層名を表示します。
-tl	実行中の文のファイル名および行番号を表示します。
-vcdfile <vcd_filename>	VCD 出力ファイルを指定します。
-vcdunit <vcd_unit>	VCD 出力単位を指定します。デフォルトはエンジンの精度単位になります。
-wdb <filename.wdb>	波形データベース出力ファイルを指定します。
-version	コンパイラ バージョンを表示します。
-view <wavefile.wcfg>	波形設定ファイルを開きます。-gui オプションと共に使用します。
-protoinst	プロトコル解析の .protoinst ファイルを指定します。
-sv_seed	SystemVerilog 制約乱数のシードを指定します。
-cov_db_dir	機能カバレッジ データベースのダンプを保存するディレクトリを指定します。カバレッジ データは、<arg>/xsim.covdb/<cov_db_name> ディレクトリに保存されます。デフォルトは ./ または xelab の値セットが継承されます。
-cov_db_name	機能カバレッジ データベースの名前を指定します。カバレッジ データは、<cov_db_dir>/xsim.covdb/<arg> ディレクトリに保存されます。デフォルトはスナップショット名または xelab の値セットが継承されます。
-downgrade_error2info	HDL メッセージの重要度をエラーから情報に下げます。
-downgrade_error2warning	HDL メッセージの重要度をエラーから警告に下げます。
-downgrade_fatal2info	HDL メッセージの重要度を致命的から情報に下げます。
-downgrade_fatal2warning	HDL メッセージの重要度を致命的から警告に下げます。
-downgrade_severity	<p>HDL メッセージの重要度を下げます。選択肢は次のとおりです。</p> <ul style="list-style-type: none"> error2warning error2info fatal2warning fatal2info
-ignore_assertions	SystemVerilog の同時処理アサートを無視します。
-ignore_coverage	SystemVerilog の機能カバレッジを無視します。

表 17: xsim コマンドのオプション (続き)

xsim オプション	説明
-ignore_feature	特定の HDL の機能またはコンストラクトの効果を無視します。選択肢は次のとおりです。 <ul style="list-style-type: none"> assertion coverage
-tempDir	一時ディレクトリの名前を指定します。



ヒント: バッチ ファイルまたはスクリプトで xelab、xsc、xsim、xvhdl、xcrg、または xvlog コマンドを実行する場合、XILINX_VIVADO 環境変数で Vivado Design Suite のインストール ディレクトリを指定する必要がある場合があります。XILINX_VIVADO 変数を設定するには、次のいずれかをスクリプトまたはバッチ ファイルに追加します。

- Windows: set XILINX_VIVADO=<vivado_install_area>/Vivado/<version>
- Linux: setenv XILINX_VIVADO vivado_install_area>/Vivado/<version>
- <version> には、2014.3、2014.4、2015.1 など、使用している Vivado ツールのバージョンを指定します。

機能カバレッジ レポートの生成

Vivado シミュレータには、機能カバレッジ レポートをテキストまたは HTML 形式で生成するユーティリティが含まれています。XCRG (Xilinx Coverage Report Generator) を使用すると、複数のカバレッジ データベースを 1 つのデータベースに統合できます。

表 18: xcrg コマンドのオプションと説明

xcrg のオプション	説明
-db_name arg	xsim.covdb に含まれるデータベースの名前を指定します。指定しない場合、ディレクトリに含まれるすべてのデータベースが使用されます。
-dir arg	xsim.covdb データベース ディレクトリへのパスを指定します。デフォルトは ./xsim.covdb です。
-file arg	カバレッジ データベースを復元するファイルとその場所指定します。
-h	ヘルプ メッセージを表示します。
-help	ヘルプ メッセージを表示します。
-merge_db_name arg	統合したデータベースの名前を指定します。デフォルトは xcrg_mdb です。
-merge_dir arg	統合したデータベースを保存するディレクトリを指定します。デフォルトは ./xsim.covdb です。
-nolog	ログ ファイルを生成しません。
-report_dir arg	カバレッジ データベースとレポートを保存するディレクトリを指定します。デフォルトは ./xcrg_report です。
-report_format arg	カバレッジ レポートのフォーマットを html、text、または all に指定します。デフォルトは html です。
-log arg	保存するログ ファイルの名前を指定します。デフォルトは xcrg.log です。

xcrg の構文例

```
xcrg -h
xcrg -file /path/to/file
xcrg -file /path/to/file -db_name work.top
xcrg -dir /path/to/abc
xcrg -dir ./abc -report_dir def -report_format html
xcrg -dir ./abc -db_name work.top -report_dir def -report_format html
xcrg -dir /path/to/abc -db_name work.top -report_dir def -report_format text
xcrg -merge_dir m
xcrg -merge_db_name xyz -report_dir def
xcrg -report_format html -nolog
xcrg -report_format html -log xcrgOutput.log
```

スタンドアロン モードでの Vivado シミュレータの実行例

Vivado シミュレータをスタンドアロン モードで実行する場合、コマンドを使用して次を実行できます。

- デザイン ファイルを解析
- デザインをエラボレートし、スナップショットを作成
- Vivado シミュレータのワークスペースと 波形設定ファイルを開いてシミュレーションを実行

手順 1: デザイン ファイルの解析

次の表に示すように、HDL ソース ファイルをタイプ別に解析します。各コマンドで複数ファイルを処理できます。

表 19: デザイン ファイル解析に使用するファイル タイプおよび関連コマンド

ファイル タイプ	コマンド
Verilog	xvlog <VerilogFileName(s)>
SystemVerilog	xvlog -sv <SystemVerilogFileName(s)>
VHDL	xvhdl <VhdlFileName(s)>

手順 2: デザインのエラボレートとスナップショットの作成

解析を実行したら、xelab コマンドを使用してデザインをエラボレートし、シミュレーションのスナップショットを作成します。

```
xelab <topDesignUnitName> -debug typical
```



重要: xelab を使用すると、複数の最上位デザイン ユニット名を指定できます。launch_simulation で使用されるのと同様の目的で Vivado シミュレータのワークスペースを使用するには、シミュレータにデバッグ レベルを typical に設定する必要があります。

手順 3: シミュレーションの実行

xelab が問題なく完了すると、Vivado シミュレータでシミュレーションの実行に使用するスナップショットが作成されます。

Vivado シミュレータのワークスペースを起動するには、次のコマンドを使用します。

```
xsim <SnapShotName> -gui
```

波形設定ファイルを開くには、次のように入力します。

```
xsim <SnapShotName> -view <wcfg FileName> -gui
```

複数の wcfg ファイルを指定するには、-view オプションを複数使用します。次に例を示します。

```
xsim <SnapShotName> -view <wcfg FileName> -view <wcfg FileName>
```

プロジェクト ファイル (.prj) の構文

注記: ここで説明するプロジェクト ファイルは、Vivado シミュレータのテキスト ベースのプロジェクト ファイルです。Vivado Design Suite で作成されるプロジェクト ファイル (.xpr) とは別のものです。

プロジェクト ファイルを使用してデザイン ファイルを解析するには、<proj_name>.prj というファイル名のテキスト ファイルを作成し、そのプロジェクト ファイル内で次の構文を使用します。

```
verilog <work_library> <file_names>... [-d <macro>]... [-i  
<include_path>]...  
vhdl <work_library> <file_name>  
sv <work_library> <file_name>  
vhdl2008 <work_library> <file_name>
```

説明:

<work_library>: 指定した行の HDL ファイルをコンパイルするライブラリ。

<file_names>: Verilog ソース ファイル。各行に複数の Verilog ファイルを指定できます。

<file_name>: VHDL ソース ファイル。1 行に 1 つの VHDL ファイルを指定します。

1. Verilog または SystemVerilog の場合: [-d <macro>] で 1 つまたは複数のマクロを定義できます。
2. Verilog または SystemVerilog の場合: [-i <include_path>] で 1 つまたは複数の <include_path> ディレクトリを定義できます。

定義済みマクロ

XILINX_SIMULATOR は Verilog の定義済みマクロです。このマクロの値は 1 です。定義済みマクロでは、ツール専用の関数が実行されるか、デザイン フローで使用するツールが特定されます。次に使用例を示します。

```
`ifdef VCS
    // VCS specific code
`endif
`ifdef INCA
    // NCSIM specific code
`endif
`ifdef MODEL_TECH
    // MODELSIM specific code
`endif
`ifdef XILINX_ISIM
    // ISE Simulator (ISim) specific code
`endif
`ifdef XILINX_SIMULATOR
    // Vivado Simulator (XSim) specific code
`endif
```

ライブラリ マップ ファイル (xsim.ini)

HDL コンパイラ プログラムの xvhdl、xvlog、xelab では、xsim.ini 設定ファイルから VHDL および Verilog 論理ライブラリの定義および物理的な位置が検出されます。

コンパイラでは、次の場所からリスト順で xsim.ini が読み込まれます。

1. 現在の作業ディレクトリの xsim.ini
2. -initfile オプションで指定されたユーザー ファイル。-initfile を指定しない場合、現在の作業ディレクトリに含まれる xsim.ini が検索されます。
3. <Vivado_Install_Dir>/data/xsim

xsim.ini ファイルの構文は、次のとおりです。

```
<logical_library1> = <physical_dir_path1>
<logical_library2> = <physical_dir_path2>
```

次は xsim.ini ファイルの例です。

```
std=<Vivado_Install_Area>/xsim/vhdl/std
ieee=<Vivado_Install_Area>/xsim/vhdl/ieee
vl=<Vivado_Install_Area>/xsim/vhdl/vl
ieee_proposed=$RDI_DATADIR/xsim/vhdl/ieee_proposed
synopsys=<Vivado_Install_Area>/xsim/vhdl/synopsys
uvm=<Vivado_Install_Area>/xsim/system-verilog/uvm
unisim=<Vivado_Install_Area>/xsim/vhdl/unisim
unimacro=<Vivado_Install_Area>/xsim/vhdl/unimacro
unifast=<Vivado_Install_Area>/xsim/vhdl/unifast
simprims_ver=<Vivado_Install_Area>/xsim/verilog/simprims_ver
```

```
unisims_ver=<Vivado_Install_Area>/xsim/verilog/unisims_ver
unimacro_ver=<Vivado_Install_Area>/xsim/verilog/unimacro_ver
unifast_ver=<Vivado_Install_Area>/xsim/verilog/unifast_ver
secureip=<Vivado_Install_Area>/xsim/verilog/secureip
work=./work
```

xsim.ini ファイルの機能および制限は、次のとおりです。

- xsim.ini ファイルでは、1 つの行で指定できるライブラリ パスは 1 つのみです。
- 物理パスに対応するディレクトリがない場合、コンパイラでそのパスに最初書き込むときに、xvhd または xvlog により作成されます。
- 物理パスは環境変数を使用して記述できます。環境変数は \$ 文字で開始する必要があります。
- 論理ライブラリのデフォルトの物理ディレクトリは、xsim/<language>/<logical_library_name> です。次は論理ライブラリ名の例です。

```
<Vivado_Install_Area>/xsim/vhdl/unisim
```

- ファイル コメントは -- で始める必要があります。

注記: 2018.2 リリース以降は、ザイリンクスでは xsim.ini および xsim_legacy.ini という 2 つの INIT ファイルを提供しています。xsim_legacy.ini は、以前のバージョンの xsim.ini と類似したファイルで、UNISIM ライブラリのマッピングが含まれます。新しい xsim.ini ファイルには UNISIM ライブラリの全ファイルのマッピングだけでなく、コンパイル済みの IP のマッピングも含まれます。

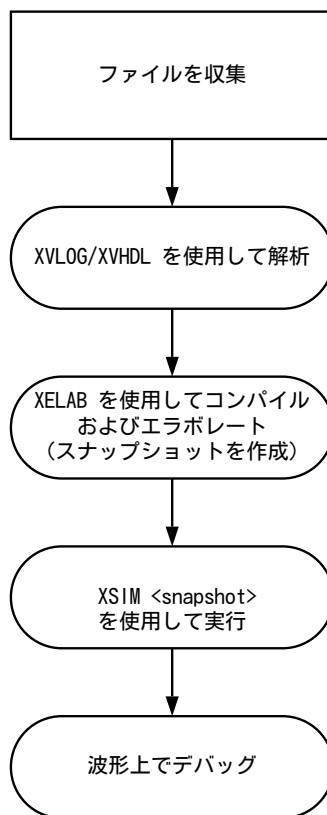
シミュレーション モードの実行

どのシミュレーション モードでもコマンド ラインから実行できます。次のセクションでは、コマンド ラインから実行する場合のシミュレーション モードについて説明します。

ビヘイビアー シミュレーション

次の図に、ビヘイビアー シミュレーションのプロセスを示します。

図 51: ビヘイビアー シミュレーション プロセス



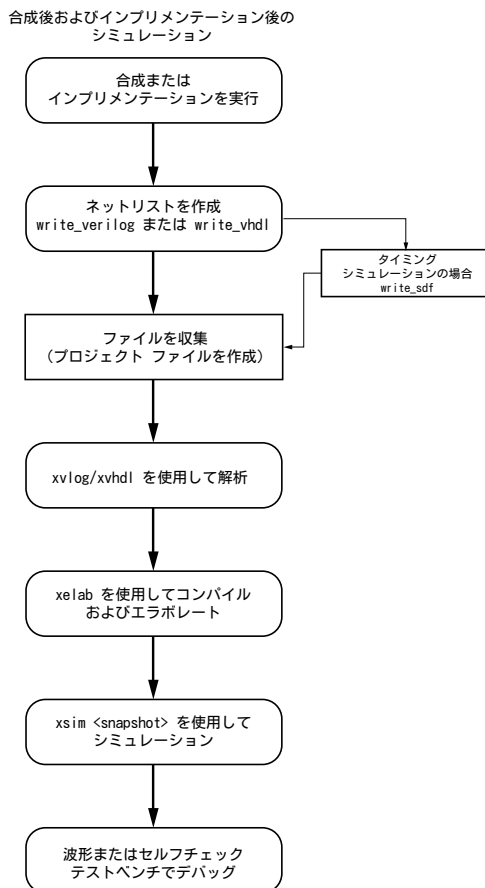
X23705-021420

Vivado Design Suite からビヘイビアー シミュレーションを実行するには、Tcl コマンド `launch_simulation -mode behavioral` を使用します。

合成後およびインプリメンテーション後のシミュレーション

合成後およびインプリメンテーション後のシミュレーションでは、論理シミュレーションまたは Verilog タイミングシミュレーションを実行できます。次の図に、合成後およびインプリメンテーション後のシミュレーション プロセスを示します。

図 52: 合成後およびインプリメンテーション後のシミュレーション



X12985

次に、コマンド ラインから合成後の論理シミュレーションを実行する例を示します。

```
synth_design -top top -part xc7k70tfbg676-2
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
launch_simulation
```



ヒント: 合成後およびインプリメンテーション後のタイミングシミュレーションを実行する場合、`write_sdf` の後に `write_verilog` コマンドを実行する必要がある、エラーレションおよびシミュレーションに適切なアノテーション コマンドが必要です。

Tcl コマンドとスクリプトの使用

シミュレーションを実行するには、Tcl コンソールで Tcl コマンドを個別に実行するか、コマンドを Tcl スクリプトに含めます。

-tclbatch オプションの使用

シミュレーション コマンドを Tcl ファイルに記述し、その Tcl ファイルを `-tclbatch <filename>` コマンドを使用して参照できます。

`-tclbatch` オプションを使用してコマンドを含むファイルを指定し、シミュレーションを開始したときにそれらのコマンドが実行されるようにします。たとえば、次を含む `run.tcl` というファイルがあるとします。

```
run 20ns
```

```
id="ag415279">current_time
quit
```

シミュレーションを実行するには、次のコマンドを使用します。

```
xsim <snapshot> -tclbatch run.tcl
```

シミュレーション コマンドを表す変数を設定しておく、よく使用されるシミュレーション コマンドをすばやく実行できます。

Tcl コンソールからの Vivado シミュレータの起動

次に、プロジェクトを作成し、ソース ファイルを読み込み、Vivado シミュレータを起動し、配置配線を実行し、SDF ファイルを書き出して、シミュレーションを再実行する Tcl コマンドの例を示します。

```
Vivado -mode Tcl
Vivado% create_project prj1
Vivado% read_verilog dut.v
Vivado% synth_design -top dut
Vivado% launch_simulation -simset sim_1 -mode post-synthesis -type
functional
Vivado% place_design
Vivado% route_design
Vivado% write_verilog -mode timesim -sdf_anno true -sdf_file postRoute.sdf
postRoute_netlist.v
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulation -simset sim_1 -mode post-implementation -type
timing
Vivado% close_project
```

export_simulation

ターゲット シミュレータのシミュレーション スクリプトをエクスポートします。生成されたスクリプトには、デザインをコンパイル、エラボレート、シミュレーションするためのシミュレータ コマンドが含まれます。

このコマンドは、指定したオブジェクトのシミュレーション コンパイル順を取得して、この情報をコンパイラ コマンドとターゲット シミュレータのデフォルト オプションを使用してシェル スクリプトにエクスポートします。オブジェクトとしては、シミュレーション ファイルセットまたは IP を指定できます。Vivado IDE 外でシミュレーションを実行する場合は、スクリプト ファイルの生成に `launch_simulation -scripts_only` ではなく `export_simulation` を使用してください。

```
export_simulation [-simulator <arg>] [-of-objects <arg>] [-lib_map_path
<arg>]
[-script_name <arg>] [-directory <arg>] [-runtime <arg>] [-absolute_path]
[-export_source_files] [-32bit] [-force] [-quiet] [-verbose]
[-ip_user_files_dir <arg>] [-ipstatic_source_dir <arg>] [-define <arg>] [-
generic
<arg>] [-include <arg>] [-use_ip_compiled_libs]
```

用途:

表 20: `export_simulation` のオプション

名前	説明
<code>[-simulator]</code>	指定したシミュレータ用のシミュレーション スクリプトを作成します。有効な値は、all、xsim、modelsim、questa、ies、vcs、xcelium、riviera、および activehdl です。デフォルトは all です。
<code>[-of-objects]</code>	指定のオブジェクトのシミュレーション スクリプトをエクスポートします。デフォルトはありません。
<code>[-lib_map_path]</code>	コンパイル済みのシミュレーション ライブラリのディレクトリ パスを指定します。指定しない場合は、生成されたスクリプトのヘッダーの手順に従い、シミュレーション ライブラリのマップ情報を手動で供給してください。デフォルトはありません。
<code>[-script_name]</code>	出力シェル スクリプトのファイル名を指定します。指定しない場合は、デフォルトという名前のファイルが生成されます。デフォルトは <code>top_module.sh</code> です。
<code>[-directory]</code>	生成したシミュレーション スクリプトの保存先ディレクトリを指定します。デフォルトは <code>export_sim</code> です。
<code>[-runtime]</code>	シミュレーションを指定した時間実行します。デフォルトでは、シミュレーションが完全に実行されるか、論理的なブレイクまたは終了条件に達するまで実行されます。
<code>[-absolute_path]</code>	すべてのファイル パスを絶対パスにします。
<code>[-export_source_files]</code>	IP/BD デザイン ファイルを出力ディレクトリにコピーします。
<code>[-32bit]</code>	32 ビットのコンパイルを実行します。
<code>[-force]</code>	既存のファイルを上書きします。
<code>[-quiet]</code>	コマンド エラーを表示しません。
<code>[-verbose]</code>	メッセージの非表示設定を解除し、すべてのメッセージを表示します。
<code>[-ip_user_files_dir]</code>	エクスポートされた IP/BD ユーザー ファイル (スタティック、ダイナミックおよびデータ ファイル) のディレクトリ パスを指定します。デフォルトはありません。
<code>[-ip_static_source_dir]</code>	エクスポートされた IP/BD スタティック ファイルへのディレクトリ パスを指定します。デフォルトはありません。
<code>[-define]</code>	このオプションで指定したリストから Verilog 定義を読み出します。デフォルトはありません。

表 20: `export_simulation` のオプション (続き)

名前	説明
<code>[-generic]</code>	このオプションで指定したリストから VHDL ジェネリックを読み出します。デフォルトはありません。
<code>[-include]</code>	このオプションで指定したリストからインクルード ディレクトリ パスを読み出します。デフォルトはありません。
<code>[-use_ip_compiled_libs]</code>	コンパイル中にコンパイル済みの IP スタティック ライブラリを参照します。このオプションを使用するには、コンパイル済み IP ライブラリを使用してスクリプトを生成するため、 <code>-ip_user_files_dir</code> および <code>-ipstatic_source_dir</code> オプションも必要です。

カテゴリ

```
simulation, xilinx Tcl store, user-written
```

ターゲット シミュレータのシミュレーション スクリプト ファイルをエクスポートします。サポートされるシミュレータは、下のリストを参照してください。生成されたスクリプトには、デザインをコンパイル、エラボレート、シミュレーションするためのシミュレータ コマンドが含まれます。

このコマンドは、指定したオブジェクトのシミュレーション コンパイル順を取得して、この情報をコンパイラ コマンドとターゲット シミュレータのデフォルト オプションを使用してシェル スクリプトにエクスポートします。オブジェクトとしては、シミュレーション ファイルセットまたは IP を指定できます。

オブジェクトを指定しない場合、アクティブなシミュレーション `top` のスクリプトが生成されます。Verilog のインクルード ディレクトリまたは Verilog の `define` 文を含むファイルへのパスは、コンパイラ コマンド ラインに追加されます。

デフォルトでは、コンパイラ コマンド ラインのデザイン ソース ファイルおよびインクルード ディレクトリ パスは、生成されるスクリプトで定義される `reference_dir` 変数に相対するパスとして設定されます。これらのパスを絶対パスにするには、`-absolute_path` オプションを使用します。

このコマンドでは、ファイルセットからのデータ ファイル (存在する場合) も出力ディレクトリにコピーされます。デザインに Verilog ソースが含まれる場合は、生成されるスクリプトにより `glbl.v` もソフトウェア インストール パスから出力ディレクトリにコピーされます。

ターゲット シミュレータ用のシミュレーション スクリプトのコンパイラ コマンドで使用されるデフォルト `.do` ファイルが出力ディレクトリに作成されます。

注記: 生成されたスクリプトでシミュレーションを実行するには、このスクリプトを生成するときに、コンパイル済みライブラリ ディレクトリ パスを指定して `compile_simlib Tcl` コマンドを実行し、シミュレーション ライブラリをコンパイルする必要があります。生成されるスクリプトでは、コンパイル ライブラリ ディレクトリからターゲット シミュレータのセットアップ ファイルが自動的に含まれます。[Project Settings] の [Simulation] ページでオプションを指定しても `export_simulation` スクリプトには影響しません。

サポートされるシミュレータ

- Vivado シミュレータ (xsim)
- ModelSim シミュレータ (modelsim)
- Questa Advanced Simulator (questa)
- Incisive Enterprise Simulator (ies)
- Verilog Compiler Simulator (vcs)

- Riviera-PRO Simulator (riviera)
- Active-HDL Simulator (activehdl)
- Cadence 社 Xcelium Parallel Simulator (xcelium)

引数

- `-of_objects`: (オプション) シミュレーション スクリプト ファイルを生成する必要のあるターゲット オブジェクトを指定します。ターゲット オブジェクトとしては、シミュレーション ファイルセット (simset) または IP を指定できます。このオプションを使用しない場合、現在のシミュレーション ファイルセットに対するファイルが生成されます。
- `-lib_map_path`: (オプション) 選択したシミュレータ用のザイリンクスコンパイル済みシミュレーション ライブラリへのパスを指定します。シミュレーション ライブラリは、`compile_simlib` を使用してコンパイルします。詳細は、生成されたスクリプトのヘッダー セクションを参照してください。このオプションを指定しない場合、生成されたスクリプトでコンパイル済みシミュレーション ライブラリが参照されず、スタティック IP ファイルがローカルでコンパイルされます。
- `-script_name`:

(オプション) シェル スクリプトの名前を指定します。このオプションを指定しない場合、`of_objects` オプションで指定されたオブジェクト タイプに基づいて、次の形式の名前が使用されます。

```
<simulation_top_name>_sim_<simulator>.sh
<ip_name>_sim_<simulator>.sh
```

- `-absolute_path`: (オプション) スクリプトで使用されるソースおよびインクルード ディレクトリ パスを絶対パスにします。デフォルトでは、すべてのパスは `-directory` オプションで指定されたディレクトリを基準とする相対パスで記述されます。スクリプトで、`reference_dir` 変数が `-directory` オプションで指定されたディレクトリ パスに設定されます。
- `-32bit`: (オプション) 32 ビット シミュレーションを実行するよう指定します。このオプションを指定しない場合、デフォルトの 64 ビット オプションがシミュレーション コマンド ラインに追加されます。
- `-force`: (オプション) 指定したスクリプト ファイルが存在する場合に上書きします。スクリプト ファイルが存在する場合に `-force` が指定されていないと、エラー メッセージが表示されます。
- `-directory`: (必須) スクリプト ファイルをエクスポートするディレクトリ パスを指定します。
- `-simulator`: (必須) シミュレーション スクリプトのターゲット シミュレータを指定します。有効なシミュレータ名は `xsim`、`modelsim`、`questa`、`ies`、および `vcs` (または `vcs_mx`) です。
- `-quiet`: (オプション) コマンドをメッセージを表示せずに実行します。コマンド ライン エラーは無視され、エラー メッセージは表示されません。実行中にエラーが発生しても、`TCL_OK` が返されます。
- `-verbose`: (オプション) メッセージの非表示設定を一時的に解除し、コマンドからのすべてのメッセージを返します。

export_ip_user_files

プロジェクトから IP/IP インテグレーター ファイルを生成およびエクスポートします。1 つまたは複数の IP で実行されるよう適用範囲を設定できます。

構文

```
export_ip_user_files [-of_objects <arg>] [-ip_user_files_dir <arg>]
                    [-ipstatic_source_dir <arg>] [-lib_map_path <arg>]
                    [-no_script] [-sync] [-reset] [-force] [-quiet]
                    [-verbose]
```

戻り値: エクスポートされたファイルのリスト。

使用法

表 21: **export_ip_user_files**

名前	説明
[-of_objects]	IP、IP インテグレーター、またはファイルセットを指定します。 デフォルトはありません。
[-ip_user_files_dir]	スタティク、ダイナミック、ラッパー、ネットリスト、スクリプト、および MEM ファイルのシミュレーション ベース ディレクトリへのパスを指定します。 デフォルトはありません。
[-ipstatic_source_dir]	スタティク IP ファイルへのディレクトリ パスを指定します。 デフォルトはありません。
[-lib_map_path]	コンパイル済みのシミュレーション ライブラリのディレクトリ パスを指定します。 デフォルトはありません。
[-no_script]	シミュレーション スクリプトをエクスポートしません。 デフォルトは 1 です。
[-sync]	IP/IP インテグレーターのダイナミック ファイルおよびシミュレーション スクリプト ファイルを削除します。
[-reset]	IP/IP インテグレーターのスタティク、ダイナミック、およびシミュレーション スクリプト ファイルをすべて削除します。
[-force]	既存のファイルを上書きします。
[-quiet]	コマンド エラーを表示しません。
[-verbose]	メッセージの非表示設定を解除し、すべてのメッセージを表示します。

説明

スタティク、ダイナミック、ネットリスト、Verilog/VHDL スタブ、メモリ初期化ファイルを使用して IP ユーザー ファイルのリポジトリをエクスポートします。

引数

- **-of_objects:**(オプション) IP のスタティク ファイルおよびダイナミック ファイルをエクスポートする必要のあるターゲット オブジェクトを指定します。
- **-ip_user_files_dir:**(オプション) ダイナミックおよび IP のその他のスタティクでないファイルの IP ユーザー ファイルのベース ディレクトリ パスを指定します。このオプションを使用しない場合、デフォルトでは IP.USER_FILES_DIR プロジェクト プロパティで指定されているパスが使用されます。
- **-ipstatic_source_dir:**(オプション) スタティク IP ファイルへのディレクトリ パスを指定します。このオプションを使用しない場合、デフォルトでは SIM.IPSTATIC_SOURCE_DIR プロジェクト プロパティで指定されているパスが使用されます。

注記: `-ip_user_files_dir` オプションを指定すると、デフォルトで IP スタティック ファイルが `ipstatic` という名前の下位ディレクトリにエクスポートされます。このオプションを `-ipstatic_source_dir` と共に指定すると、IP スタティック ファイルが `-ipstatic_source_dir` オプションで指定したパスにエクスポートされます。

- `-clean_dir:` (オプション) 中央ディレクトリからスタティック、ダイナミック、およびその他のファイルを含むすべてのファイルを削除します。

例

次の例では、`char_fifo` IP のダイナミック ファイルを `<project>/<project>.ip_user_files/ip/char_fifo` ディレクトリに、`char_fifo` IP のスタティック ファイルを `<project>/<project>.ip_user_files/ipstatic` ディレクトリにエクスポートしています。

```
% export_ip_user_files -of_objects [get_ips char_fifo]
```

コンパイル、エラボレーション、シミュレーション、ネットリスト、アドバンスオプション

Vivado IDE の Flow Navigator で [Simulation] を右クリックして [Simulation Settings] をクリックすると、[Settings] ダイアログ ボックスの [Simulation] ページが開き、さまざまなコンパイル、エラボレーション、シミュレーション、ネットリスト、アドバンス オプションを設定できます。

コンパイル オプション

[Compilation] タブでは、コンパイラ指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして保存され、Verilog および VHDL ソース ファイルをシミュレーション用にコンパイルするため xvlog および xvhdl ユーティリティで使用されます。

Vivado シミュレータのコンパイル オプション

表 22: Vivado シミュレータのコンパイル オプション

オプション	説明
Verilog options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
xsim.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル。
xsim.compile.xvlog.nosort	コンパイル中に Verilog ファイルを並べ替えません。
xsim.compile.xvhdl.nosort	コンパイル中に VHDL ファイルを並べ替えません。
xsim.compile.xvlog.relax	厳しい HDL 言語チェック ルールを緩和します。
xsim.compile.xvhdl.relax	厳しい HDL 言語チェック ルールを緩和します。
xsim.compile.incremental	インクリメンタル コンパイルを実行します。
xsim.compile.xvlog.more-options	追加の XVLOG コンパイル オプションを指定します。
xsim.compile.xvhdl.more-options	追加の XVHDL コンパイル オプションを指定します。

Questa Advanced Simulator のコンパイル オプション

表 23: Questa Advanced Simulator のコンパイル オプション

オプション	説明
Verilog options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
questasim.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル。
questasim.compile.vhdl_syntax	VHDL 構文を指定します。
questasim.compile.use_explicit_decl	すべての信号を記録します。
questasim.compile.load_glbl	GLBL モジュールを読み込みます。
questasim.compile.incremental	インクリメンタル コンパイルを実行します。
questasim.compile.vlog.more_options	追加の VLOG コンパイル オプションを指定します。
questasim.compile.vcom.more_options	追加の VCOM コンパイル オプションを指定します。

ModelSim シミュレータのコンパイル オプション

表 24: ModelSim シミュレータのコンパイル オプション

オプション	説明
Verilog options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
modelsim.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル。
modelsim.compile.vhdl_syntax	VHDL 構文を指定します。
modelsim.compile.use_explicit_decl	すべての信号を記録します。
modelsim.compile.load_glbl	GLBL モジュールを読み込みます。
modelsim.compile.vlog.more_options	追加の VLOG コンパイル オプションを指定します。
modelsim.compile.vcom.more_options	追加の VCOM コンパイル オプションを指定します。

IES シミュレータのコンパイル オプション

表 25: IES シミュレータのコンパイル オプション

オプション	説明
Verilog options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
ies.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル。
ies.compile.v93	VHDL 93 機能をイネーブルにします。
ies.compile.relax	VHDL の解釈を緩和します。
ies.compile.load_glbl	GLBL モジュールを読み込みます。
ies.compile.ncvhdlmore_options	追加の NCVHDL コンパイル オプションを指定します。
ies.compile.ncvlog.more_options	追加の NCVLOG コンパイル オプションを指定します。

VCS シミュレータのコンパイル オプション

表 26: VCS シミュレータのコンパイル オプション

オプション	説明
Verilog options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
vcs.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル。
vcs.compile.load_glbl	GLBL モジュールを読み込みます。
vcs.compile.vhdlan.more_options	追加の VHDLAN コンパイル オプションを指定します。
vcs.compile.vlogan.more_options	追加の VLOGAN コンパイル オプションを指定します。

Xcelium シミュレータのコンパイル オプション

表 27: Xcelium のコンパイル オプション

オプション	説明
Verilog Options	Verilog インクルード パスを設定し、マクロを定義します。
Generics/Parameters options	ジェネリック/パラメーター値を指定します。
xcelium.compile.tcl.pre	コンパイルを実行する前に実行する必要があるコマンドを含む Tcl ファイル
xcelium.compile.v93	VHDL 93 機能をイネーブルにします。
xcelium.compile.relax	VHDL の解釈を緩和します。
xcelium.compile.load_glbl	GLBL モジュールを読み込みます。
xcelium.compile.xmvhdl.more_options	追加の XMVHDL コンパイル オプションを指定します。
xcelium.compile.xmvlog.more_options	追加の XMVLOG コンパイル オプションを指定します。
xcelium.compile.xmsc.more_option	追加の XMSC コンパイル オプションを指定します。
xcelium.compile.g++.more_option	追加の G++ コンパイル オプションを指定します。

エラボレーション オプション

[Elaboration] タブでは、エラボレーション指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして保存され、xelab ユーティリティでシミュレーション スナップショットをエラボレートおよび生成する際に使用されます。表のプロパティを選択すると、そのプロパティの詳細が表示され、値を変更できます。

Vivado シミュレータのエラボレーション オプション

表 28: Vivado シミュレータのエラボレーション オプション

オプション	説明
xsim.elaborate.snapshot	シミュレーション スナップショット名を指定します。

表 28: Vivado シミュレータのエラボレーション オプション (続き)

オプション	説明
xsim.elaborate.debug_level	シミュレーション デバッグの可視化レベルを選択します。デフォルトは typical です。
xsim.elaborate.relax	厳しい HDL 言語チェック ルールを緩和します。
xsim.elaborate.mt_level	並列実行するサブコンパイル ジョブ数を指定します。
xsim.elaborate.load_glbl	GLBL モジュールを読み込みます。
xsim.elaborate.rangecheck	VHDL のランタイム値の範囲チェックをイネーブルにします。
xsim.elaborate.sdf_delay	タイミング シミュレーションで使用する SDF タイミング遅延タイプを指定します。
xsim.elaborate.xelab.more_option	追加の XELAB エラボレーション オプションを指定します
xsim.elaborate.xsc.more_option	追加の XSC のオプションをエラボレーション中に指定します。

Questa Advanced Simulator のエラボレーション オプション

表 29: Questa Advanced Simulator のエラボレーション オプション

オプション	説明
questasim.elaborate.acc	デフォルトで最適化される可能性のあるシミュレーション オブジェクトへのアクセスをイネーブルにします。デフォルトは npr です。
questasim.elaborate.vopt.more_options	追加の VOPT エラボレーション オプションを指定します。
questasim.elaborate.sccom.more_options	追加の sccom のオプションをエラボレーション中に指定します。

ModelSim シミュレータのエラボレーション オプション

表 30: ModelSim のエラボレーション オプション

オプション	説明
modelsim.elaborate.acc	デフォルトで最適化される可能性のあるシミュレーション オブジェクトへのアクセスをイネーブルにします。
modelsim.elaborate.vopt.more_options	追加の VOPT エラボレーション オプションを指定します。

IES シミュレータのエラボレーション オプション

表 31: IES シミュレータのエラボレーション オプション

オプション	説明
ies.elaborate.update	書き込みの前にユニットが最新であるかどうかをチェックします。
ies.elaborate.ncelab.more_options	追加の ncelab エラボレーション オプションを指定します。

VCS シミュレータのエラボレーション オプション

表 32: VCS シミュレータのエラボレーション オプション

オプション	説明
vcs.elaborate.debug_pp	プロセス後のデバッグ アクセスをイネーブルにします。
vcs.elaborate.vcs.more_options	追加の VCS エラボレーション オプションを指定します。

Xcelium シミュレータのエラボレーション オプション

表 33: Xcelium のエラボレーション オプション

オプション	説明
xcelium.elaborate.update	書き込みの前にユニットが最新であるかどうかをチェックします。
xcelium.elaborate.xmelab.more_options	追加の xmelab エラボレーション オプションを指定します。

シミュレーション オプション

[Simulation] タブでは、シミュレーション指示子を定義および管理します。これらの指示子はシミュレーション ファイルセットのプロパティとして保存され、xsim アプリケーションで現在のプロジェクトをシミュレーションする際に使用されます。表のプロパティを選択すると、そのプロパティの詳細が表示され、値を変更できます。

Vivado シミュレータのシミュレーション オプション

表 34: Vivado シミュレータのオプション

オプション	説明
xsim.simulate.runtime	Vivado シミュレータのシミュレーション実行時間を指定します。シミュレーション スナップショットを読み込む際は空白にし、ユーザー入力を待つようにします。
xsim.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
xsim.simulate.log_all_signals	すべてのオブジェクト 信号を記録します。
xsim.simulate.wdb	シミュレーション波形データベース ファイルを指定します。
xsim.simulate.saif	SAIF ファイル名を指定します。
xsim.simulate.saif_scope	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
xsim.simulate.saif_all_signals	SAIF ファイルを生成するため、テスト中のデザインのオブジェクト信号をすべて記録します。
xsim.simulate.xsim.more_option	Vivado シミュレータの追加のシミュレーション オプションを指定します。
xsim.simulate.custom_tcl	Vivado で生成される通常の Tcl ファイルの代わりにシミュレーション中に読み込まれるカスタムの Tcl ファイルの名前を指定します。
xsim.simulate.add_positional	XSIM に渡す追加の位置パラメーターを指定します。

Questa Advanced Simulator のシミュレーション オプション

表 35: Questa Advanced Simulator のシミュレーション オプション

オプション	説明
questasim.simulate.runtime	シミュレーション実行時間を指定します。
questasim.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
questasim.simulate.log_all_signals	すべての信号を記録します。
questasim.simulate.custom_do	カスタムの do ファイルの名前を指定します。
questasim.simulate.custom_udo	カスタム ユーザー do ファイルの名前を指定します。
questa.simulate.ieee_warning	IEEE 警告を表示しません。
questasim.simulate.sdf_delay	sdf アノテーションの遅延タイプを指定します。
questasim.simulate.saif	SAIF ファイルを指定します。
questasim.simulate.saif_scope	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
questasim.simulate.vsim.more_option	追加の VSIM シミュレーション オプションを指定します。
questa.simulate.custom_wave_do	通常 Vivado で生成される wave.do ファイルの代わりに使用されるカスタムの wave.do ファイルの名前を指定します。

ModelSim シミュレータのシミュレーション オプション

表 36: ModelSim シミュレータのシミュレーション オプション

オプション	説明
modelsim.simulate.runtime	シミュレーション実行時間を指定します。
modelsim.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
modelsim.simulate.log_all_signals	すべての信号を記録します。
modelsim.simulate.custom_do	カスタムの do ファイルの名前を指定します。
modelsim.simulate.custom_udo	カスタム ユーザー do ファイルの名前を指定します。
modelsim.simulate.sdf_delay	sdf アノテーションの遅延タイプを指定します。
modelsim.simulate.ieee_warning	IEEE 警告を表示しません。
modelsim.simulate.saif	SAIF ファイルを指定します。
modelsim.simulate.saif_scope	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
modelsim.simulate.vsim.more_option	追加の VSIM シミュレーション オプションを指定します。
modelsim.simulate.custom_wave_do	通常 Vivado で生成される wave.do ファイルの代わりに使用されるカスタムの wave.do ファイルの名前を指定します。

IES シミュレータのシミュレーション オプション

表 37: IES シミュレータのシミュレーション オプション

オプション	説明
ies.simulate.runtime	シミュレーション実行時間を指定します。
ies.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
ies.simulate.log_all_signals	すべての信号を記録します。

表 37: IES シミュレータのシミュレーション オプション (続き)

オプション	説明
ies.simulate.update	書き込みの前にユニットが最新であるかどうかをチェックします。
ies.simulate.ieee_warning	IEEE 警告を表示しません。
ies.simulate.saif	SAIF ファイル名を指定します。
ies.simulate.saif_scope	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
ies.simulate.ncsim.more_option	追加の NCSIM シミュレーション オプションを指定します。

VCS シミュレータのシミュレーション オプション

表 38: VCS シミュレータのシミュレーション オプション

オプション	説明
vcs.simulate.runtime	シミュレーション実行時間を指定します。
vcs.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
vcs.simulate.log_all_signals	すべての信号を記録します。
vcs.simulate.saif	SAIF ファイル名を指定します。
vcs.simulate.saif_scope	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
vcs.simulate.vcs.more_option	追加の VCS シミュレーション オプションを指定します。

Xcelium シミュレータのシミュレーション オプション

表 39: Xcelium シミュレータのシミュレーション オプション

オプション	説明
xcelium.simulate.tcl.post	シミュレーションを実行する前に実行するコマンドを含む Tcl ファイル。
xcelium.simulate.runtime	シミュレーション実行時間を指定します。
xcelium.simulate.log_all_signals	すべての信号を記録します。
xcelium.simulate.update	書き込みの前にユニットが最新であるかどうかをチェックします。
xcelium.simulate.ieee_warnings	IEEE 警告を表示しません。
xcelium.simulate.saif_scope	SAIF ファイル名を指定します。
xcelium.simulate.saif	消費電力見積もりが必要なデザイン階層インスタンス名を指定します。
xcelium.simulate.xmsim.more_options	追加の XMSIM シミュレーション オプションを指定します。

ネットリスト オプション

[Netlist] タブでは、Verilog ネットリストの SDF アノテーションおよび SDF 遅延で取り込まれるプロセス コーナーに関するネットリスト コンフィギュレーション オプションを指定します。これらのオプションはシミュレーション ファイルセットのプロパティとして保存され、シミュレーション用ネットリストを生成する際に使用されます。

Vivado シミュレータのネットリスト オプション

表 40: Vivado シミュレータのネットリスト オプション

オプション	説明
-sdf_anno	-sdf_anno オプションを使用します。このオプションはデフォルトでオンになっています。
-process_corner	-process_corner を fast または slow に設定します。

注記: すべてのサードパーティ シミュレータ (Questa Advanced Simulator、ModelSim シミュレータ、IES、VCS、および Xcelium シミュレータ) のネットリスト オプションは、Vivado シミュレータのネットリスト オプションと同様です。

アドバンス シミュレーション オプション

[Advanced] タブには 2 つのオプションが含まれます。

- [Enable incremental compilation] オプション: インクリメンタル コンパイルをイネーブルにし、次回の run でシミュレーション ファイルを保持します。
- [Include all design sources for simulation] オプション: デフォルトでオンです。このオプションをオンにすると、デザイン ソースからのファイルと現在のシミュレーション セットからのファイルすべてがシミュレーションで使用されます。デザイン ソースを変更した場合でも、ビヘイビアー シミュレーションを実行するときにその変更が反映されます。



重要: これは、上級ユーザー用の機能です。オフにすると、予期しない結果になる可能性があります。[Include all design sources for simulation] チェック ボックスはデフォルトでオンになっています。チェック ボックスがオンになっていれば、シミュレーション セットにアウト オブ コンテキスト (OOC) の IP、IP インテグレーター ファイル、および DCP が含まれます。

このチェック ボックスをオフにすると、シミュレーションする必要のあるファイルのみを含めることができますが、先に述べたように、予期せぬ結果となる可能性があります。

注記: アドバンス シミュレーション オプションはどのシミュレータでも同じです。

Vivado シミュレータでの SystemVerilog のサポート

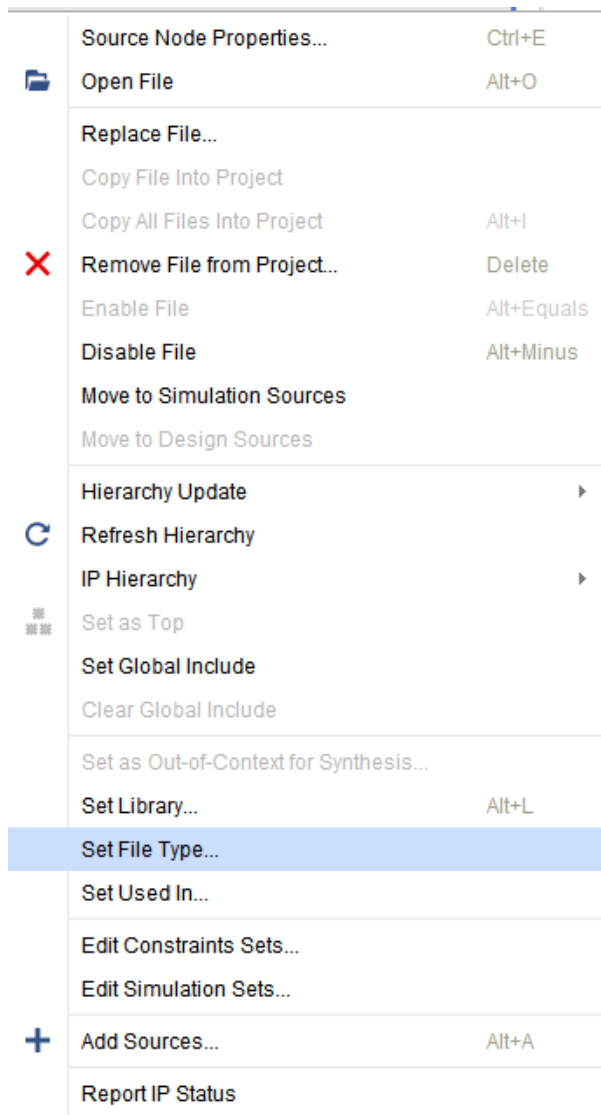
Vivado シミュレータでは、SystemVerilog のサブセットがサポートされます。次の表に、SystemVerilog の合成可能なセットを示します。表 42: サポートされるダイナミック型コンストラクト に、サポートされるテストベンチを示します。

特定のファイルで SystemVerilog を使用

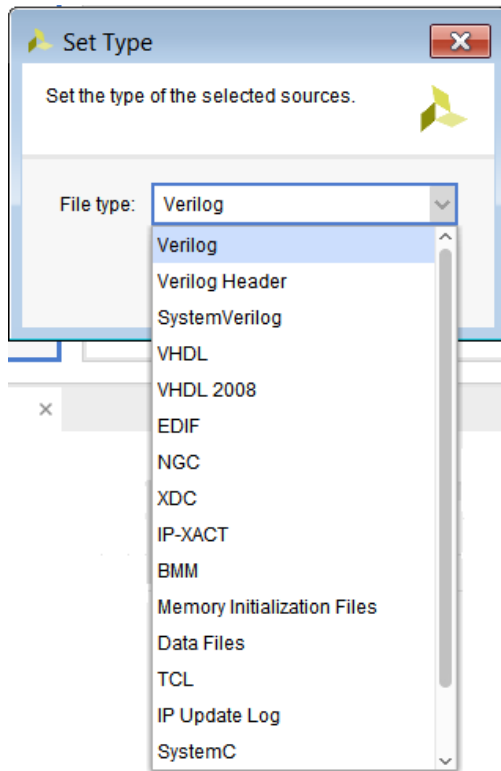
Vivado IDE のデフォルトでは、.v ファイルは Verilog 2001 構文で、.sv ファイルは SystemVerilog 構文でコンパイルされます。

Vivado IDE で特定の .v ファイルに SystemVerilog を使用するには、次の手順に従います。

1. ファイルを右クリックして [Set file type] をクリックします (次の図を参照)。



2. [Set Type] ダイアログ ボックスで、[File type] を [Verilog] から [SystemVerilog] に変更し、[OK] をクリックします。



または、Tcl コンソールで次の Tcl コマンドを使用します。

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

スタンドアロンまたは PRJ モードでの SystemVerilog の実行

スタンドアロン モード

SystemVerilog ファイルを読み込むには、次のように `xvlog` に `-sv` オプションを使用します。

```
xvlog -sv <Design file list>
xvlog -sv -work <LibraryName> <Design File List>
xvlog -sv -f <FileName> [Where FileName contain path of test cases]
```

PRJ モード

Vivado シミュレータを `prj` ベースのフローで実行する場合は、`verilog` または `vhdl` の場合と同様に、ファイル タイプを `sv` に指定します。

```
xvlog -prj <prj File>
xelab -prj <prj File> <topModuleName> <other options>
```

prj ファイルのエンティティは次のように表示されます。

```
verilog      library1 <FileName>
sv           library1 <FileName> [File parsed in SystemVerilog mode]
vhdl        library2 <FileName>
sv           library3 <FileName> [File parsed in SystemVerilog mode]
```

表 41: SystemVerilog 1800-2012 の合成可能なセット

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
データ型		6	
	単体型および集合体型	6.4	サポートあり
	ネットおよび変数	6.5	サポートあり
	変数宣言	6.8	サポートあり
	ベクター宣言	6.9	サポートあり
	2 ステート (2 値) および 4 ステート (4 値) データ型	6.11.2	サポートあり
	符号付きおよび符号なし整数データ型	6.11.3	サポートあり
	real、shortreal、および realtime データ型	6.12	サポートあり
	ユーザー定義型	6.18	サポートあり
	列挙型	6.19	サポートあり
	新規データ型を列挙型として定義	6.19.1	サポートあり
	列挙型の範囲	6.19.2	サポートあり
	型チェック	6.19.3	サポートあり
	数式の列挙型	6.19.4	サポートあり
	列挙型メソッド	6.19.5	サポートあり
	型パラメーター	6.20.3	サポートあり
	const 定数	6.20.6	サポートあり
	型演算子	6.23	サポートあり
	キャスト演算子	6.24.1	サポートあり
	\$cast ダイナミック キャスト	6.24.2	サポートあり
	ビットストリーム キャスト	6.24.3	サポートあり
集合体データ型		7	
	構造体	7.2	サポートあり
	パック型/アンパック型構造体	7.2.1	サポートあり
	構造体への代入	7.2.2	サポートあり
	共用体	7.3	サポートあり
	パック型/アンパック型共用体	7.3.1	サポートあり
	タグ付き共用体	7.3.2	サポートなし
	パック型配列	7.4.1	サポートあり
	アンパック型配列	7.4.2	サポートあり
	配列の演算	7.4.3	サポートあり
	多次元配列	7.4.5	サポートあり
	配列のインデックスおよびスライス	7.4.6	サポートあり

表 41: SystemVerilog 1800-2012 の合成可能なセット (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	配列代入	7.6	サポートあり
	サブルーチンへの引数としての配列	7.7	サポートあり
	配列クエリ関数	7.11	サポートあり
	配列ソース メソッド	7.12	サポートあり
プロセス		9	
	組み合わせロジックの <code>always_comb</code> プロシージャ	9.2.2	サポートあり
	暗示的な <code>always_comb</code> のセンシティビティリスト	9.2.2.1	サポートあり
	ラッチ ロジックの <code>always_latch</code> プロシージャ	9.2.2.3	サポートあり
	順序ロジックの <code>always_ff</code> プロシージャ	9.2.2.4	サポートあり
	順次ブロック	9.3.1	サポートあり
	並列ブロック	9.3.2	サポートあり
	手続的タイミング制御	9.4	サポートあり
	条件付きイベント制御	9.4.2.3	サポートあり
	順次イベント	9.4.2.4	サポートなし
代入文		10	
	連続代入文	10.3.2	サポートあり
	変数宣言代入 (変数初期化)	10.5	サポートあり
	代入のようなコンテキスト	10.8	サポートあり
	配列代入パターン	10.9.1	サポートあり
	構造代入パターン	10.9.2	サポートあり
	アンパック型配列連結	10.10	サポートあり
	ネット エイリアス	10.11	サポートなし
演算子および演算式		11	
	定数式	11.2.1	サポートあり
	集合体式	11.2.2	サポートあり
	実数オペランドを使用する演算子	11.3.1	サポートあり
	ロジック (4 値) およびビット (2 値) 型の演算	11.3.4	サポートあり
	演算式内の代入	11.3.6	サポートあり
	代入演算子	11.4.1	サポートあり
	インクリメントおよびデクリメント演算子	11.4.2	サポートあり
	符号なしおよび符号付き型を使用した演算式	11.4.3.1	サポートあり
	ワイルドカード等価演算子	11.4.6	サポートあり
	連結演算子	11.4.12	サポートあり
	メンバーシップ演算子の設定	11.4.13	サポートあり
	<code>stream_expressions</code> の連結	11.4.14.1	サポートあり
	ジェネリック ストリームの並べ替え	11.4.14.2	サポートあり
	代入ターゲットとしてのストリーミング連結 (アンパック型)	11.4.14.3	サポートあり

表 41: SystemVerilog 1800-2012 の合成可能なセット (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	動的にサイズが決定されるデータのストリーミング	11.4.14.4	サポートあり
手続きプログラム文		12	
	unique-if、unique0-if、および priority-if	12.4.2	サポートあり
	unique-if、unique0-if、および priority-if コンストラクトで生成された違反レポート	12.4.2.1	サポートあり
	if 文違反レポートおよび複数プロセス文	12.4.2.2	サポートあり
	unique-case、unique0-case、および priority-case	12.5.3	サポートあり
	unique-case、unique0-case、および priority-case コンストラクトで生成された違反レポート	12.5.3.1	サポートあり
	case 文違反レポートおよび複数プロセス文	12.5.3.2	サポートあり
	メンバーシップ case 文の設定	12.5.4	サポートあり
	パターン一致条件文	12.6	サポートなし
	ループ文	12.7	サポートあり
	ジャンプ文	12.8	サポートあり
タスク		13.3	
	スタティックおよび自動タスク	13.3.1	サポートあり
	タスクのメモリ使用量および同時アクティベーション	13.3.2	サポートあり
関数		13.4	
	戻り値と void 関数	13.4.1	サポートあり
	スタティックおよび自動関数	13.4.2	サポートあり
	定数関数	13.4.3	サポートあり
	関数呼び出しで起動されるバックグラウンドプロセス	13.4.4	サポートあり
サブルーチン呼び出しおよび引数渡し		13.5	
	値渡し	13.5.1	サポートあり
	参照渡し	13.5.2	サポートあり
	デフォルトの引数値	13.5.3	サポートあり
	名前による引数バインド	13.5.4	サポートあり
	オプションの引数リスト	13.5.5	サポートあり
	インポートおよびエクスポート関数	13.6	サポートあり
	タスクおよび関数名	13.7	サポートあり
ユーティリティ システム タスクおよびシステム関数 (合成可能セットのみ)		20	サポートあり
I/O システム タスクおよびシステム関数 (合成可能セットのみ)		21	サポートあり
コンパイラ指示子		22	サポートあり

表 41: SystemVerilog 1800-2012 の合成可能なセット (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
モジュールおよび階層		23	
	デフォルトのポート値	23.2.2.4	サポートあり
	最上位モジュールおよび \$root	23.3.1	サポートあり
	モジュール インスタンス化構文	23.3.2	サポートあり
	入れ子のモジュール	23.4	サポートあり
	外部モジュール	23.5	サポートあり
	階層名	23.6	サポートあり
	メンバー選択および階層名	23.7	サポートあり
	上方向の名前参照	23.8	サポートあり
	モジュール パラメーターのオーバーライド	23.10	サポートあり
	スコープまたはインスタンスへの補助コードのバインド	23.11	サポートなし
インターフェイス		25	
	インターフェイス 構文	25.3	サポートあり
	入れ子のインターフェイス	25.3	サポートあり
	インターフェイスのポート	25.4	サポートあり
	名前指定ポート バンドルの例	25.5.1	サポートあり
	ポート バンドルの接続例	25.5.2	サポートあり
	ポート バンドルのジェネリック インターフェイスへの接続例	25.5.3	サポートあり
	modport 式	25.5.4	サポートあり
	クロッキング ブロックおよび modport	25.5.5	サポートあり
	インターフェイスおよび指定ブロック	25.6	サポートあり
	インターフェイスでのタスクの使用例	25.7.1	サポートあり
	modport でのタスクの使用例	25.7.2	サポートあり
	タスクおよび関数のエクスポート例	25.7.3	サポートあり
	複数タスクのエクスポート例	25.7.4	サポートあり
	パラメーター指定されたインターフェイス	25.8	サポートあり
	仮想インターフェイス	25.9	サポートあり
パッケージ		26	
	パッケージ宣言	26.2	サポートあり
	パッケージ内のデータの参照	26.3	サポートあり
	モジュール ヘッダーでのパッケージの使用	26.4	サポートあり
	パッケージからのインポートされた名前のエクスポート	26.6	サポートあり
	std ビルトイン パッケージ	26.7	サポートあり
generate コンストラクト		27	サポートあり

テストベンチの機能

Vivado シミュレータでは、次の表に示すように、よく使用されるテストベンチ機能の一部がサポートされています。

表 42: サポートされるダイナミック型コンストラクト

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
文字列データ型		6.16	サポートあり
	文字列演算子 (IEEE 1800-2012 の表 6-9)	6.16	サポートあり
	len()	6.16.1	サポートあり
	putc()	6.16.2	サポートあり
	getc()	6.16.3	サポートあり
	toupper()	6.16.4	サポートあり
	tolower()	6.16.5	サポートあり
	compare	6.16.6	サポートあり
	icompare()	6.16.7	サポートあり
	substr()	6.16.8	サポートあり
	atoi()、atohex()、atooct()、atobin()	6.16.9	サポートあり
	atoreal()	6.16.10	サポートあり
	itoa()	6.16.11	サポートあり
	hextoa()	6.16.12	サポートあり
	octtoa()	6.16.13	サポートあり
	bintoa()	6.16.14	サポートあり
	realtoa()	6.16.15	サポートあり
動的配列		7.5	サポートあり
	動的配列 (新規)	7.5.1	サポートあり
	サイズ	7.5.2	サポートあり
	削除	7.5.3	サポートあり
連想配列		7.8	サポートあり
	ワイルドカード インデックス	7.8.1	サポートあり
	文字列インデックス	7.8.2	サポートあり
	クラス インデックス	7.8.3	サポートあり
	積分インデックス	7.8.4	サポートあり
	その他のユーザー定義型	7.8.5	サポートあり
	無効インデックスへのアクセス	7.8.6	サポートあり
	連想配列メソッド	7.9	サポートあり
	num() および size()	7.9.1	サポートあり
	delete()	7.9.2	サポートあり
	exists()	7.9.3	サポートあり
	first()	7.9.4	サポートあり
	last()	7.9.5	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	next()	7.9.6	サポートあり
	prev()	7.9.7	サポートあり
	トラバース メソッドへの引数	7.9.8	サポートあり
	連想配列代入	7.9.9	サポートあり
	連想配列引数	7.9.10	サポートあり
	連想配列リテラル	7.9.11	サポートあり
[Queue]		7.10	サポートあり
	キュー演算子	7.10.1	サポートあり
	キュー メソッド	7.10.2	サポートあり
	size()	7.10.2.1	サポートあり
	insert()	7.10.2.2	サポートあり
	delete()	7.10.2.3	サポートあり
	pop_front()	7.10.2.4	サポートあり
	pop_back()	7.10.2.5	サポートあり
	push_front()	7.10.2.6	サポートあり
	push_back()	7.10.2.7	サポートあり
	キューの要素への参照の持続性	7.10.3	サポートあり
	代入およびアンパック型配列連結を使用したキューのアップデート	7.10.4	サポートあり
	境界付きキュー	7.10.5	サポートあり
クラス		8	サポートあり
	クラス ジェネラル	8.1	サポートあり
	概要	8.2	サポートあり
	構文	8.3	サポートあり
	オブジェクト (クラス インスタンス)	8.4	サポートあり
	オブジェクト プロパティおよびオブジェクト パラメーター データ	8.5	サポートあり
	オブジェクト メソッド	8.6	サポートあり
	コンストラクター	8.7	サポートあり
	スタティック クラス プロパティ	8.8	サポートあり
	スタティック メソッド	8.9	サポートあり
	this	8.10	サポートあり
	割り当て、名前変更、およびコピー	8.11	サポートあり
	継承およびサブクラス	8.12	サポートあり
	オーバーライドされたメンバー	8.13	サポートあり
	スーパー	8.14	サポートあり
	キャスト演算子	8.15	サポートあり
	チェーン コンストラクター	8.16	サポートあり
	データ非表示およびカプセル化	8.17	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	定数クラス プロパティ	8.18	サポートあり
	仮想メソッド	8.19	サポートあり
	抽象クラスおよび純仮想メソッド	8.20	サポートあり
	多様性: ダイナミック メソッド ルックアップ	8.21	サポートあり
	クラス スコープ解決演算子 ::	8.22	サポートあり
	アウト オブ ブロック宣言	8.23	サポートあり
	パラメーター化されたクラス	8.24	サポートあり
	パラメーター化されたクラスのクラス解決演算子	8.24.1	サポートあり
	Typedef クラス	8.25	サポートあり
	クラスおよびストラクチャ	8.26	サポートあり
	メモリ管理	8.27	サポートあり
プロセス		9	サポートあり
	並列処理 - fork join_any および fork join_none	9.3	サポートあり
	wait fork	9.6.1	サポートあり
	disable fork	9.6.3	サポートあり
	詳細なプロセス制御	9.7	サポートあり
クロッキング ブロック		14	サポートあり
	資料全体	14.1	サポートあり
	概要	14.2	サポートあり
	クロッキング ブロック宣言	14.3	サポートあり
	入力および出力キュー	14.4	サポートあり
	階層式	14.5	サポートなし
	複数クロッキング ブロックの信号	14.6	サポートあり
	クロッキング ブロックのスコープ およびライフタイム	14.7	サポートあり
	複数クロッキング ブロックの例	14.8	サポートあり
	インターフェイスおよびクロッキング ブロック	14.9	サポートあり
	クロッキング ブロック イベント	14.10	サポートあり
	サイクル遅延	14.11	サポートあり
	デフォルト クロッキング	14.12	サポートあり
	入力サンプリング	14.13	サポートあり
	グローバル クロック	14.14	サポートなし
	同期イベント	14.15	サポートあり
	同期ドライブ	14.16	サポートあり
	ドライブおよびノンブロッキング代入	14.16.1	サポートあり
	クロッキング出力信号の駆動	14.16.2	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
セマフォ		15.3	サポートあり
	セマフォ メソッド new()	15.3.1	サポートあり
	セマフォ メソッド put()	15.3.2	サポートあり
	セマフォ メソッド get()	15.3.3	サポートあり
	セマフォ メソッド try_get()	15.3.4	サポートあり
メールボックス		15.4	サポートあり
	メールボックス メソッド new()	15.4.1	サポートあり
	メールボックス メソッド num()	15.4.2	サポートあり
	メールボックス メソッド put()	15.4.3	サポートあり
	メールボックス メソッド try_put()	15.4.4	サポートあり
	メールボックス メソッド get()	15.4.5	サポートあり
	メールボックス メソッド try_get()	15.4.6	サポートあり
	メールボックス メソッド peek()	15.4.7	サポートあり
	メールボックス メソッド try_peek()	15.4.8	サポートあり
	パラメーター指定されたメールボックス	15.4.9	サポートあり
名前付きイベント		15.5	サポートあり
	イベントのトリガー	15.5.1	サポートあり
	イベントを待機	15.5.2	サポートあり
	持続トリガー	15.5.3	サポートなし
	イベント シーケンス	15.5.4	サポートなし
	名前付きイベント変数に対する演算	15.5.5	サポートあり
	結合イベント	15.5.5.1	サポートあり
	イベントの再要求	15.5.5.2	サポートあり
	イベント比較	15.5.5.3	サポートあり
アサーション		16	サポートあり
	資料全体	16.1	サポートあり
	概要	16.2	サポートあり
	assert	16.2	サポートあり
	assume	16.2	サポートあり
	cover	16.2	サポートなし
	restrict	16.2	サポートなし
	即時アサーション	16.3	サポートあり
	遅延 (deferred) アサーション	16.4	サポートなし
	並列アサーションの概要	16.5	サポートあり
	サンプリング	16.5.1	サポートあり
	アサーション クロック	16.5.2	サポートあり
	ブール式	16.6	サポートあり
	シーケンス	16.7	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	シーケンスの宣言	16.8	サポートあり
	シーケンス宣言での型が指定された仮引数	16.8.1	サポートあり
	シーケンス宣言でのローカル変数仮引数	16.8.2	サポートあり
	シーケンス 演算	16.9	サポートあり
	演算子の優先順位	16.9.1	サポートあり
	シーケンスでの繰り返し	16.9.2	サポートあり
	サンプリングされた値の関数	16.9.3	サポートあり
	グローバル クロックで過去または将来にサンプリングされる値の関数	16.9.4	サポートなし
	AND 演算	16.9.5	サポートあり
	交差 (長さ制限付き AND)	16.9.6	サポートあり
	OR 演算	16.9.7	サポートあり
	first_match 演算	16.9.8	サポートあり
	シーケンスの条件	16.9.9	サポートあり
	シーケンス内のシーケンス	16.9.10	サポートあり
	単純なサブシーケンスからのシーケンスの作成	16.9.11	サポートあり
	ローカル変数	16.10	サポートあり
	シーケンスの一致に対するサブルーチンの呼び出し	16.11	サポートあり
	プロパティの宣言	16.12	サポートあり
	シーケンス プロパティ	16.12.1	サポートあり
	否定プロパティ	16.12.2	サポートあり
	論理和プロパティ	16.12.3	サポートあり
	論理積プロパティ	16.12.4	サポートあり
	if-else プロパティ	16.12.5	サポートあり
	含意	16.12.6	サポートあり
	含意および iff プロパティ	16.12.7	サポートあり
	プロパティ インスタンス化	16.12.8	サポートあり
	followed-by プロパティ	16.12.9	サポートなし
	nexttime プロパティ	16.12.10	サポートなし
	always プロパティ	16.12.11	サポートなし
	until プロパティ	16.12.12	サポートなし
	eventually プロパティ	16.12.13	サポートなし
	中止プロパティ	16.12.14	サポートなし
	強演算子および弱演算子	16.12.15	サポートなし
	ケース	16.12.16	サポートなし
	再帰プロパティ	16.12.17	サポートなし

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	プロパティ 宣言での型が指定された仮引数	16.12.18	サポートあり
	プロパティ 宣言でのローカル変数仮引数	16.12.19	サポートあり
	プロパティの例	16.12.20	サポートあり
	有限長さと無限長さの動作	16.12.21	サポートあり
	非縮退	16.12.22	サポートあり
	複数クロックのサポート	16.13	サポートなし
	並列アサーション	16.14	サポートあり
	assert 文	16.14.1	サポートあり
	assume 文	16.14.2	サポートあり
	cover 文	16.14.3	サポートなし
	restrict 文	16.14.4	サポートなし
	手続きコード外での並列アサーション文の使用	16.14.5	サポートあり
	手続きコードへの並列アサーション文の埋め込み	16.14.6	サポートなし
	推論された値の関数	16.14.7	サポートなし
	有意な評価	16.14.8	サポートなし
	disable iff の解決	16.15	サポートあり
	クロックの解決	16.16	サポートあり
	複数クロック シーケンスおよびプロパティのセマンティック先頭クロック	16.16.1	サポートあり
	expect 文	16.17	サポートなし
	クロッキング ブロックおよび並列アサーション	16.18	サポートあり
ランダム制約		18	サポートあり
	コンセプトおよび使用法	18.3	サポートあり
	ランダム変数	18.4	サポートあり
	rand 修飾子	18.4.1	サポートあり
	randc 修飾子	18.4.2	サポートあり
	制約ブロック	18.5	サポートあり
	外部制約ブロック	18.5.1	サポートあり
	制約継承	18.5.2	サポートあり
	セット メンバーシップ	18.5.3	サポートあり
	分配	18.5.4	サポートあり
	含意	18.5.6	サポートあり
	if-else 制約	18.5.7	サポートあり
	反復制約	18.5.8	サポートあり
	foreach 反復制約	18.5.8.1	サポートあり
	配列削減反復制約	18.5.8.2	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	グローバル制約	18.5.9	サポートあり
	変数の順序付け	18.5.10	サポートあり
	スタティック制約ブロック	18.5.11	サポートあり
	制約内の関数	18.5.12	サポートあり
	制約ガード	18.5.13	サポートあり
	ソフト制約	18.5.14	サポートあり
	randomize メソッド	18.6.1	サポートあり
	pre_randomize および post_randomize	18.6.2	サポートあり
	ランダム化メソッドの動作	18.6.3	サポートあり
	インライン制約	18.7	サポートあり
	ローカル スコープ解決	18.7.1	サポートあり
	rand_mode を使用したランダム変 数の無効化	18.8	サポートあり
	constraint_mode を使用した制約の 制御	18.9	サポートあり
	動的な制約変更	18.10	サポートあり
	インライン ランダム変数制御	18.11	サポートあり
	インライン制約チェッカー	18.11.1	サポートあり
	スコープ変数 std::randomize のラ ンダム化	18.12	サポートあり
	スコープ変数 std::randomize への 制約の追加	18.12.1	サポートあり
	乱数システム関数およびメソッド	18.13	サポートあり
	\$urandom	18.13.1	サポートあり
	\$urandom_range	18.13.2	サポートあり
	srandom	18.13.3	サポートあり
	get_randstate	18.13.4	サポートあり
	set_randstate	18.13.5	サポートあり
	ランダム安定性	18.14	サポートあり
	ランダム化の手動シード設定	18.15	サポートあり
	randcase	18.16	サポートあり
	randsequence	18.17	サポートなし
プログラム		24	サポートあり
	プログラム コンストラクト	24.3	サポートあり
	プログラム コンストラクトにおけ るコードのセマンティックのスケジ ューリング	24.3.1	サポートあり
	プログラム ポート接続	24.3.2	サポートあり
	テストベンチ レースの除去	24.4	サポートあり
	サイクル/イベント モードのプロッ キング タスク	24.5	サポートあり

表 42: サポートされるダイナミック型コンストラクト (続き)

プライマリ コンストラクト	セカンダリ コンストラクト	LRM セクション	ステータス
	匿名プログラム	24.6	サポートなし
	プログラム制御タスク	24.7	サポートあり
機能カバレッジ		19	サポートあり
	資料全体	19.1	サポートあり
	概要	19.2	サポートあり
	カバレッジ モデルの定義: covergroup	19.3	サポートあり
	クラスでの covergroup の使用	19.4	サポートあり
	カバレッジ ポイントの定義	19.5	サポートあり
	値のピンの指定	19.5.1	サポートあり
	カバーグループを使用したカバーポ イント ピン	19.5.1.1	サポートあり
	カバーポイント ピン設定カバーグ ループ式	19.5.1.2	サポートなし
	遷移のピンの指定	19.5.2	サポートあり
	カバレッジ ポイントの自動ピン作 成	19.5.3	サポートあり
	カバレッジ ポイント ピンのワイル ドカード指定	19.5.4	サポートあり
	カバレッジ ポイント値または遷移 の除外	19.5.5	サポートあり
	無効なカバレッジ ポイント値また は遷移の指定	19.5.6	サポートあり
	値の解決	19.5.7	サポートあり
	クロス カバレッジの定義	19.6	サポートあり
	クロス カバレッジ ピンの定義	19.6.1	サポートあり
	ユーザー定義のクロス カバレッジ および選択式の例	19.6.1.1	サポートあり
	カバーグループ式を使用したクロス ピン	19.6.1.2	サポートあり
	クロス ピン自動定義型	19.6.1.3	サポートあり
	クロス ピン設定式	19.6.1.4	サポートあり
	クロス積の除外	19.6.2	サポートあり
	無効なクロス積の指定	19.6.3	サポートあり
	カバレッジ オプションの指定	19.7	サポートあり
	カバーグループ型のオプション	19.7.1	サポートあり
	定義済みのカバレッジ メソッド	19.8	サポートあり
	ビルトイン サンプル メソッドのオ ーバーライド	19.8.1	サポートあり
	定義済みのカバレッジ システム タ スクおよびシステム関数	19.9	サポートあり
	option および type_option メンバ ーの構成	19.10	サポートあり

注記: キュー、動的配列、連想配列、クラスなどのダイナミック型のセンシティビティはサポートされていません。そのため、ダイナミック型のアップデートを待つブロックは正しく機能しない場合があります。次に例を示します。

```
module top();
  int c[$];
  event e1;
  initial
  begin
    c[0] = 10;
    for(int i = 0; i <= 10; i++)
    begin
      c = {i, c};
      -> e1;
      #5;
    end
  end
  always@(*) $display($time, " trying to read sensitivity on dynamic type :
%d", c[0]);
  // this won't work as sensitivity on dynamic type is not supported
  always @(e1) $display($time, " coming from event sensitivity : %d",
c[0]); // this we
can do as WA
  always_comb if(c.size() > 0) $display($time, " Coming from size
sensitivity : %d",
c[0]); // sensitivity on size works
```

UVM のサポート

Vivado® 統合設計環境では、Vivado シミュレータ (XSim) で UVM (Universal Verification Methodology) がサポートされています。UVM バージョン 1.2 のライブラリがコンパイル済みであり、Vivado で使用可能です。このコンパイル済み UVM ライブラリを使用するには、`xvlog` および `xelab` コマンドに `-L uvm` を渡します。Vivado 統合設計環境で使用する場合は、次のプロパティを設定します。

```
set_property -name {xsim.compile.xvlog.more_options} -value {-L uvm} -  
objects [get_filesets sim_1]  
set_property -name {xsim.elaborate.xelab.more_options} -value {-L uvm} -  
objects [get_filesets sim_1]
```

デフォルトでは、Vivado シミュレータで UVM バージョン 1.2 がサポートされます。UVM バージョン 1.1 を使用するには、`xvlog` および `xelab` コマンドに `-uvm_version 1.1` を渡します。Vivado 統合設計環境で使用する場合は、次のプロパティを設定します。

```
set_property -name {xsim.compile.xvlog.more_options} -value {-uvm_version  
1.1} -objects [get_filesets sim_1]  
set_property -name {xsim.elaborate.xelab.more_options} -value {-uvm_version  
1.1} -objects [get_filesets sim_1]
```

これらのプロパティを Vivado GUI から設定するには、[Settings] ダイアログ ボックスの [Simulation] ページで [Compilation] および [Elaboration] タブを使用します。詳細は、[シミュレーション設定](#) を参照してください。

Vivado シミュレータの VHDL 2008 サポート

概要

Vivado® シミュレータでは、VHDL 2008(IEEE 1076-2008) のサブセットがサポートされます。サポートされる機能は、「サポートされる機能」セクションにリストされています。

コンパイルおよびシミュレーション

Vivado シミュレータ実行ファイルの `xvhdl` は、VHDL のデザイン ユニットをパーサー ダンプ (`.vdb`) に変換するために使用します。Vivado シミュレータではデフォルトで 93 および 2008 規格 (STD) と IEEE パッケージが使用されるので、93 および 2008 機能を自由に混合させることができます。VHDL-93 規格 (STD) のみと IEEE パッケージを使用する場合は、`xvhdl` に `-93_mode` を渡します。VHDL 2008 モードのみを使用したファイルをコンパイルするには、`xvhdl` に `-2008` オプションを渡す必要があります。

たとえば、`top.vhdl` というデザインを VHDL 2008 でコンパイルするには、次のコマンド ラインを使用します。

```
xvhdl -2008 -work mywork top.vhdl
```

Vivado シミュレータ実行ファイル `xelab` は、デザインをエラボレートし、シミュレーション用の実行イメージを作成するために使用します。

`xelab` では、次のいずれかを実行できます。

- `xvhdl` で作成されたパーサー ダンプのエラボレート
- VHDL ソース ファイルを直接読み込み。

`xvhdl` で作成されたパーサー ダンプをエラボレートするのに必要なオプションはありません。`xelab` で `-vhdl2008` を指定すると、VHDL ソース ファイルが直接使用できます。

例 1:

```
xelab top -s mysim; xsim mysim -R
```

例 2:

```
xelab -vhdl2008 top.vhdl top -s mysim; xsim mysim -R
```


コマンドラインで `xvhdl` および `xelab` に VHDL ファイルを指定する代わりに、PRJ ファイルを使用できます。デザインに `top.vhdl` (2008 モード) および `bot.vhdl` (93 モード) という 2 つのファイルがある場合、次のように `example.prj` という名前のプロジェクト ファイルを作成できます。

<code>vhdl xil_defaultlib bot.vhdl</code>
<code>vhdl2008 xil_defaultlib top.vhdl</code>

プロジェクト ファイルでは、各行をファイルの言語タイプで開始し、`xil_defaultlib` などのライブラリ名をその後記述します。ファイルが複数ある場合は、名前をスペースで区切ります。VHDL 93 の場合、言語タイプとして `vhdl` を使用する必要があります。VHDL 2008 の場合は、`vhdl2008` を使用してください。

PRJ ファイルは、次の例のように使用できます。

```
xelab -prj example.prj xil_defaultlib.top -s mysim; xsim mysim -R
```

また、VHDL 93 および VHDL 2008 デザイン ユニットを混合するには、`xvhdl` で正しい言語モードを指定してファイルを別々にコンパイルし、デザインの最上位をエラボレートします。たとえば、`bot.vhdl` ファイルに `bot` という VHDL 93 モジュールが含まれ、`top.vhdl` ファイルに `top` という VHDL 2008 モジュールが含まれる場合、次の例に示すようにコンパイルします。

```
xvhdl bot.vhdl
xvhdl -2008 top.vhdl
xelab -debug typical top -s mysim
```

実行ファイルを `xelab` で生成したら、シミュレーションを通常どおりに実行できます。

例 1:

```
xsim mysim -gui
```

例 2:

```
xsim mysim -R
```

固定小数点および浮動小数点パッケージ

Vivado シミュレータで使用される固定小数点および浮動小数点パッケージは、VHDL-2008 で導入された新しい改善された IEEE 規格パッケージです。VHDL-93 規格の固定小数点または浮動小数点パッケージを使用している場合、Vivado 合成では機能するかも知れませんが、シミュレーション用に VHDL ソース ファイルを変更する必要があります。

たとえば、Vivado 合成で固定小数点パッケージの次の構文を使用しているとします。

```
library ieee;
use ieee.fixed_pkg.all;
```

この場合、Vivado シミュレータでは次の VHDL-2008 の構文に変更します。

```
library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
```

Vivado 合成での固定小数点および浮動小数点パッケージの詳細は、『Vivado Design Suite ユーザー ガイド: 合成』(UG901) の [このセクション](#) を参照してください。

浮動小数点パッケージの場合も同様に変更します。

サポートされる機能

表 43: VHDL 2008 (IEEE1076-2008) でサポートされる機能

機能	例/コメント
固定および浮動小数点パッケージ、符号なしビットなどを含む、コンパイル済み VHDL 2008 STD および IEEE パッケージ。	XSIM がまだサポートされていない一般的なパッケージなど、その他の言語機能によって制限されます。新しく追加されたすべての std 関数がサポートされているわけではありません。 stop および finish はサポートされています。
単純化されたセンシテビティ リスト	process(all)
比較演算子	?=、?/=、?>、?>=、?<、?<= x ?= y
単項簡約論理演算子	<pre>signal x: std_logic_vector(0 to 31); signal x_and : std::logic; ... x_and <= and x;</pre>
単純 case 文	<pre>case x and y is when '1' => report "1"; when '0' => report "0"; end case;</pre> 中間変数または信号の代わりに、論理式を直接 case 文で使用できます。
配列/ビット論理演算子	<pre>signal s : std_logic; signal v, r : std_logic_vector(0 to 7); ... r <= s and v;</pre>
配列/ビット加算演算子	ライブラリ関数
拡張ビット文字列リテラル	16SX"FF" = "1111_1111_1111_1111" 16UX"FF" = "0000_1111_1111_1111"
条件および選択した逐次文	<pre>process(clk) ... with x select y := "111" when "110", "000" when others; a := '1' when b = '1' else '0' when b = '0'; ...</pre>

表 43: VHDL 2008 (IEEE1076-2008) でサポートされる機能 (続き)

機能	例/コメント
保護付き型	<pre>type areaOfSquare is protected procedure setx(newx : real); impure function area return real; end protected; type areaOfSquare is protected body variable x : real = 0.0; ...</pre> <p>保護付き型の共有変数は、HDL シミュレーションでサポートされていますが、Tcl および GUI ではその値を検証できません。</p>
プロシージャ宣言でのキーワード parameter	<pre>procedure proc parameter (a : in std_logic)</pre>
サブタイプ定義の配列要素分解関数	<pre>type bit_word is array (natural range <>) of bit; function resolve_array (s : bit_word) return bit; subtype resolved_array is (resolve_array) bit_word;</pre>
ブロック コメント	<pre>/* X <= 1; process(all) ... */</pre>
定義済み配列型	boolean_vector、integer_vector など。
ジェネリックとして渡される型	<pre>Sentity test is generic (type data_type); port (x : in data_type; s : out data_type); end entity test;</pre>
信号への階層参照	<pre><<signal .top.dut_inst.sig1 : std_logic_vector(3 downto 0)>></pre>
ポート マップの論理式	
出力ポートの読み出し	
最大および最小演算子	
マッチング case 文	
この表に記述されていない機能は、Vivado シミュレータではサポートされていません。	

Vivado シミュレータのダイレクト プログラミング インターフェイス (DPI)

概要

SystemVerilog ダイレクト プログラミング インターフェイス (DPI) を使用すると、C コードと SystemVerilog コードを結合でき、SystemVerilog コードで C 関数を呼び出したり、C 関数から SystemVerilog タスクまたは関数をコールバックしたりできます。Vivado® シミュレータでは、次に説明するように、DPI タスク/関数としてすべてのコンストラクトがサポートされます。

C コードのコンパイル

C コードをオブジェクト コードに変換し、複数のオブジェクト コード ファイルを共有ライブラリ (Windows の場合は .a、Linux の場合は .so) にリンクするため、xsc という新しいコンパイラ実行ファイルが提供されています。xsc コンパイラは、<Vivado installation>/bin ディレクトリに含まれます。-sv_lib を使用すると、C コードを含む共有ライブラリを Vivado シミュレータ/エラボーレーター実行ファイルに変換できます。xsc コンパイラは、gcc などの C コンパイラと同じように機能します。xsc コンパイラでは、次が呼び出されます。

- LLVM clang コンパイラ: C コードをオブジェクト コードに変換
- GNU リンカー: C ファイルに対応する 1 つまたは複数のオブジェクト ファイルから共有ライブラリ (Windows の場合は .a、Linux の場合は .so) を作成

xsc コンパイラで生成された共有ライブラリは、次に示すように xelab に新しく追加されたオプションを 1 つまたは複数使用すると、Vivado シミュレータ カーネルにリンクされます。xelab で作成されたシミュレーション スナップショットには、コンパイルされた C コードとコンパイルされた SystemVerilog コードを接続して、C と SystemVerilog 間の効率的な通信を可能にする機能があります。

xsc コンパイラ

xsc コンパイラを使用すると、1 つまたは複数の C ファイルから共有ライブラリ (Windows の場合は .a、Linux の場合は .so) を作成できます。この xsc で生成された共有ライブラリを残りのデザインに含めるには、xelab を使用します。共有ライブラリを作成するには、次のプロセスを使用します。

- 1 段階プロセス: xsc や -compile オプションを付けずに、すべての C ファイルを -link に渡します。

- 2 段階プロセス::

```
xsc -compile <C files>
xsc -link <object files>
```

使用法

```
xsc [options] <files...>
```

オプション

オプションには、ダブル ダッシュ (--) かダッシュ (-) を使用できます。

表 44: XSC コンパイラ オプション

オプション	説明
-compile	オブジェクト ファイルをソース C ファイルからのみ生成します。リンク ステージは実行されません。
-f [-file] <arg>	指定したファイルから追加オプションを読み出します。
-h [-help]	ヘルプ メッセージを表示します。
-i [-input_file] <arg>	コンパイルまたはリンク用の入力ファイルをリストします (1 つのオプションに 1 つのファイル)。
-link	リンク ステージのみを実行して、オブジェクト ファイルから共有ライブラリ (.a または .so) を生成します。
-mt <arg> (=auto)	並列実行可能なサブコンパイル ジョブの数を指定します。有効な値は次のとおりです。 <ul style="list-style-type: none"> • auto: 自動 • n: 1 より大きい整数 • off: マルチスレッドをオフ デフォルトは auto です。
-o [-output] <arg>	出力される共有ライブラリの名前を指定します。-link オプションとのみ使用可能です。
-work <arg>	出力される作業ディレクトリを指定します。デフォルトは <current_directory>/xsim.dir/xsc です。
-v [-verbose] <arg>	表示メッセージの詳細レベルを指定します。有効な値: 0、1 デフォルトは 0 です。
-additional_option <arg>	コンパイラに追加オプションを指定します。複数の -additional_option オプションを使用できます。
-gcc_compile_options <arg>	コンパイラに追加オプションを供給します。複数の -gcc_compile_options オプションを使用できます。
-gcc_link_options <arg>	リンカーに追加オプションを供給します。複数の -gcc_link_options オプションを使用できます。
-shared	リンク ステージのみを実行して、オブジェクト ファイルから共有ライブラリ (.so) を生成します。
-gcc_version	内部で使用する C コンパイラのバージョンを表示します。
-gcc_path	内部で使用する C コンパイラのパスを表示します。

表 44: XSC コンパイラ オプション (続き)

オプション	説明
-lib <arg>	読み込む論理ライブラリのディレクトリを指定します。デフォルトは <current_directory>/xsim.dir/xs です。
-cppversion <arg>	C++ バージョンを設定します。現在のところ、C++ 11 および 14 がサポートされています。デフォルトは 11 です。

例

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
xsc -compile function1.c function2.c -work abc
xsc -link abc/function1.lnx64.o abc/function2.lnx64.o -work abc
```

注記: Linux では、DPI ライブラリの検索にデフォルトで LD_LIBRARY_PATH が使用されます。このため、ライブラリ名が lib* で開始している場合は、xelab に -dpi_absolute オプションを使用してください。

注記: コンパイラに -additional_option オプションを使用すると、追加のオプションを渡すことができます。

- 例:

```
xsc t1.c --additional_option "-I<path>"
```

- 複数のパスを渡す例:

```
xsc t1.c --additional_option "-I<path>" --additional_option "-I<path>"
```

xelab を使用したコンパイル済み C コードの SystemVerilog への統合

コンパイルした C コードを SystemVerilog に統合する xelab の DPI に関するオプションは、次のとおりです。

表 45: xelab の DPI 関連のオプション

オプション	説明
-sv_root arg	検索する必要がある DPI 共有ライブラリに関連するルート ディレクトリを指定します。デフォルトは <current_directory>/xsim.dir/xsc です。
-sv_lib arg	SystemVerilog にインポートされる C 関数を定義する DPI 共有ライブラリの名前 (ファイル拡張子なし) を指定します。
-sv_liblist arg	DPI 共有ライブラリをポイントするブートストラップ ファイルを指定します。
-dpiheader arg	インポートおよびエクスポートされた関数の C 宣言を含む DPI ヘッダー ファイルを生成します。
-dpi_absolute	Linux で lib<libname>.so という形式の DPI ライブラリに LD_LIBRARY_PATH ではなく絶対パスを使用します。
-dpi_stacksize arg	DPI タスクのユーザー定義のスタック サイズ。

`r-sv_liblist arg` の詳細は、『IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language』の 1228 ページの付録 J.4.1 を参照してください。

C と SystemVerilog の境界で使用可能なデータ型

SystemVerilog の IEEE 規格では、C と SystemVerilog の境界で使用できるのは C および SystemVerilog データ型の一部のみです。次に、Vivado シミュレータでサポートされるデータ型と、C と SystemVerilog データ型間のマップを示します。

サポートされるデータ型

次の表に、C と SystemVerilog の境界で使用可能なデータ型と、C から SystemVerilog および SystemVerilog から C へのデータ型のマップを示します。

表 46: C と SystemVerilog の境界で使用可能なデータ型

SystemVerilog	C	サポート	コメント
<code>byte</code>	<code>char</code>	あり	なし
<code>shortint</code>	<code>short int</code>	あり	なし
<code>int</code>	<code>int</code>	あり	なし
<code>longint</code>	<code>long long</code>	あり	なし
<code>real</code>	<code>double</code>	あり	なし
<code>shortreal</code>	<code>float</code>	あり	なし
<code>chandle</code>	<code>void *</code>	あり	なし
<code>string</code>	<code>const char*</code>	あり	なし
<code>bit</code>	<code>unsigned char</code>	あり	<code>sv_0</code> 、 <code>sv_1</code>
svdpi.h を使用した場合 C でのみ使用可能			
<code>logic</code> 、 <code>reg</code>	<code>unsigned char</code>	あり	<code>sv_0</code> 、 <code>sv_1</code> 、 <code>sv_z</code> 、 <code>sv_x</code> :
bit の配列 (パック型)	<code>svBitVecVal</code>	あり	svdpi.h で定義
logic/reg の配列 (パック型)	<code>svLogicVecVal</code>	あり	svdpi.h で定義
<code>enum</code>	基になる <code>enum</code> 型	あり	なし
パック型 <code>struct</code> 、 <code>union</code>	配列として渡される	あり	なし
bit、 <code>logic</code> のアンパック型配列	配列として渡される	あり	C は SystemVerilog を呼び出し可能
アンパック型 <code>struct</code>	<code>struct</code> として渡される	あり	なし
アンパック型 <code>union</code>	<code>struct</code> として渡される	なし	なし
配列を開く	<code>svOpenArrayHandle</code>	あり	なし

SystemVerilog データ型の C データ型へのマップ方法を示す C ヘッダー ファイルを生成するには、`-dpiheader <file name>` パラメーターを `xelab` に渡します。データ型マップに関するその他の詳細は、SystemVerilog の IEEE 規格を参照してください。

ユーザー定義型のマップ

列挙型 (enum)

同等の SystemVerilog 型 (enum の基本型によって `svLogicVecVal` または `svBitVecVal`) への変換には、列挙型 (enum) を定義できます。列挙型の配列の場合は、同等の SystemVerilog 配列が作成されます。

例

- SystemVerilog 型:

```
typedef enum reg [3:0] { a = 0, b = 1, c } eType;
eType e;
eType e1[4:3];
typedef enum bit { a = 0, b = 1 } eTypeBit;
eTypeBit e3;
eTypeBit e4[3:1];
```

- C 型:

```
svLogicVecVal e[SV_PACKED_DATA_NELEMS(4)];
svLogicVecVal e1[2][SV_PACKED_DATA_NELEMS(4)];
svBit e3;
svBit e4[3];
```



ヒント: C 引数型は、enum の基本型と方向によって異なります。

パック型構造体/共用体

パック型構造体または共用体を使用する場合、同等の SystemVerilog 型である `svLogicVecVal` または `svBitVecVal` が DPI の C 側に作成されます。

例

- SystemVerilog 型:

```
typedef struct packed {
    int i;
    bit b;
    reg [3:0] r;
    logic [2:0] [4:8] [9:1] l;
} sType;
sType c_obj;
sType [3:2] c_obj1[5];
```

- C 型:

```
svLogicVecVal c_obj[SV_PACKED_DATA_NELEMS(172)];
svLogicVecVal c_obj1[5][SV_PACKED_DATA_NELEMS(344)];
```

配列は、パック型またはアンパック型のどちらでも、`svLogicVecVal` または `svBitVecVal` の配列として表記されます。

アンパック型構造体

同等のアンパック型が C 側で作成され、すべてのメンバーが同等の C 表記に変換されます。

例

- SystemVerilog 型:

```
typedef struct {
    int i;
    bit b;
    reg r[3:0];
    logic [2:0] l[4:8][9:1];
} sType;
```

- C 型:

```
typedef struct {
    int i;
    svBit b;
    svLogic r[4];
    svLogicVecVal l[5][9][SV_PACKED_DATA_NELEMS(3)];
} sType;
```

svdpi.h 関数のサポート

svdpi.h ヘッダー ファイルは、<vivado installation>/data/xsim/include ディレクトリに含まれます。

次の svdpi.h 関数がサポートされます。

```
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);
void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);
const char* svDpiVersion();
svScope svGetScope();
svScope svSetScope(const svScope scope);
const char* svGetNameFromScope(const svScope);
int svPutUserData(const svScope scope, void*userKey, void* userData);
void* svGetUserData(const svScope scope, void* userKey);
```

DPI でのオープン配列

SystemVerilog でインポート関数を宣言する際には、オープン配列として仮引数を指定できます。仮配列引数の特定の次元を空白 (オープン) に指定すると、さまざまなサイズの実際の引数を渡すことができるので、より一般的な C コードが使用できるようになります。C 言語では、オープン配列は `SVOpenArrayHandle` と記述します。このハンドルを該当する関数に渡すと、オープン配列の情報 (オープンになった次元のサイズなど) をクエリして、実際のデータにアクセスできます。

宣言

オープン配列は、SystemVerilog コードのインポート関数/タスク宣言にのみ表示されることがあります。次元をオープンにしておく場合は、オープン配列を指定する必要があります。この空白の次元のサイズは、実際の引数に対して決定されます。

例

SystemVerilog 関数の宣言:

```
import "DPI-C" function int myFunction1(input bit[] v);
import "DPI-C" function void myFunction2(input int v1[], input int v2[],
output int
v3[]);
```

C 言語では、オープン配列はハンドルと指定された API でのみアクセスできることがあります。

```
int myFunction1(const SVOpenArrayHandle v);
void myFunction2(const SVOpenArrayHandle v1, const SVOpenArrayHandle v2,
const
SVOpenArrayHandle v3);
```

svdpi.h のサポート

svdpi.h では、次のオープン配列に関連する関数がサポートされます。

```
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svSize(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);
void *svGetArrayPtr(const svOpenArrayHandle);
int svSizeOfArray(const svOpenArrayHandle);
void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2,
int indx3);
void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, ...);
void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1);
void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, int indx2);
void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, int indx2, int indx3);
void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal*
```

```

s, int indx1, ...);
void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1);
void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1, int indx2);
void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1, int indx2, int indx3);
void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, ...);
void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1);
void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2);
void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2, int indx3);
void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, ...);
void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int
indx1);
void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2);
void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2, int indx3);
svBit svGetBitArrElem(const svOpenArrayHandle s, int indx1, ...);
svBit svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int
indx3);
svLogic svGetLogicArrElem(const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
int
indx3);
void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value, int
indx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int
indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int
indx1, int
indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
int indx2, int indx3);
void svPutBitArrElem(const svOpenArrayHandle d, svBit value, int
indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1,
int indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1,
int indx2, int indx3);

```

SystemVerilog コード例

```

module m();
import "DPI-C" function void myFunction1(input int v[]);
int arr[4];
int dynArr[];
initial begin

```

```
arr = '{4, 5, 6, 7}';
myFunction1(arr);
dynArr = new[6];
dynArr = '{8, 9, 10, 11, 12, 13}';
myFunction1(dynArr);
end
endmodule
C code:
#include "svdpi.h"
void myFunction1(const svOpenArrayHandle v)
{
    int l1 = svLow(v, 1);
    int h1 = svHigh(v, 1);
    for(int i = l1; i <= h1; i++) {
        printf("\t%d", *((char*)svGetArrElemPtr1(v, i)));
    }
    printf("\n");
}
```

例

注記: 次のすべての例では、問題なく実行されると PASSED と表示されます。

次に例を示します。

- **-sv_lib、-sv_liblist、および -sv_root を使用したインポート例:** sv_lib、-sv_liblist および -sv_root オプションを使用するインポート例。
- **出力を含む関数:** 出力引数を含む関数。
- **単純なインポート/エクスポート フロー (xelab -dpiheader フロー):** 単純なインポート/エクスポート フロー (xelab -dpiheader <filename> フロー) を示す例。

-sv_lib、-sv_liblist、および -sv_root を使用したインポート例

コード

次のファイルがあるとしてします。

- C 関数を含むファイル 2 つ
- 次の関数を使用する SystemVerilog ファイル:
 - function1.c
 - function2.c
 - file.sv

function1.c

```
#include "svdpi.h"
DPI_DLLESPEC
int myFunction1()
{
    return 5;
}
```

function2.c

```
#include <svdpi.h>
DPI_DLLESPEC
int myFunction2()
{
    return 10;
}
```

file.sv

```
module m();
import "DPI-C" pure function int myFunction1 ();
import "DPI-C" pure function int myFunction2 ();
integer i, j;
initial
begin
#1;
    i = myFunction1();
    j = myFunction2();
    $display(i, j);
    if( i == 5 && j == 10)
        $display("PASSED");
    else
        $display("FAILED");
end
endmodule
```

使用法

次に、C ファイルをコンパイルして Vivado シミュレータにリンクする方法を示します。

1 段階フロー (最も簡単なフロー)

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
```

フローの説明:

xsc コンパイラで C コードをコンパイルしてリンクし、共有ライブラリ (xsim.dir/xsc/dpi.so) を作成して、xelab でこの共有ライブラリを -sv_lib オプションを使用して参照します。

2 段階フロー

```
xsc -compile function1.c function2.c -work abc
xsc -link abc/function1.lnx64.o abc/function2.lnx64.o -work abc
xelab -svlog file.sv -sv_root abc -sv_lib dpi -R
```

フローの説明:

- 2 つの C ファイルを作業ディレクトリ (abc) で該当するオブジェクト コードにコンパイルします。
- これらの 2 つのファイルをリンクして共有ライブラリ (dpi.so) を作成します。
- -sv_root オプションを使用して、このライブラリが作業ディレクトリ (abc) から検索されるようにします。



ヒント: `-sv_root` には、`-sv_lib` オプションで指定された共有ライブラリを検索するディレクトリを指定します。Linux では、`-sv_root` が指定されておらず、DPI ライブラリの名前に接頭辞 `lib` と接尾辞 `.so` が付いている場合、共有ライブラリのディレクトリ パスには `LD_LIBRARY_PATH` 環境変数が使用されます。

2 段階フロー (上記よりもさらにオプションを追加)

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -link "abc/function1.lnx64.o" "abc/function2.lnx64.o" -work "abc" -o
final -v 1
xelab -svlog file.sv -sv_root "abc" -sv_lib final -R
```

フローの説明:

ユーザーがコンパイルおよびリンクを実行する場合は、`-verbose` オプションを使用すると、コンパイラが起動されたパスおよびオプションを確認できます。これらはこの後必要に合わせて調整できます。上記の例の場合、`final` という共有ライブラリが作成されます。この例では、ファイル パスのスペースがどのように処理されるかも示しています。

出力を含む関数

コード

file.sv

```
/*- - - -*/
package pack1;
import "DPI-C" function int myFunction1(input int v, output int o);
import "DPI-C" function void myFunction2 (input int v1, input int v2,
output int o);
endpackage
/*-- ---*/
module m();
int i, j;
int o1, o2, o3;
initial
begin
#1;
j = 10;
o3 = pack1::myFunction1(j, o1); // should be 10/2 = 5
pack1::myFunction2(j, 2+3, o2); // 5 += 10 + 2+3
$display(o1, o2);
if( o1 == 5 && o2 == 15)
$display("PASSED");
else
$display("FAILED");
end
endmodule
```

function.c

```
#include "svdpi.h"
DPI_DLLESPEC
int myFunction1(int j, int* o)
{
*o = j / 2;
return 0;
}
```

```

}
DPI_DLLESPEC
void myFunction2(int i, int j, int* o)
{
  *o = i+j;
  return;
}

```

run.ksh

```

xsc function.c
xelab -vlog file.sv -sv -sv_lib dpi -R

```

単純なインポート/エクスポート フロー (xelab -dpiheader フロー)

このフローでは、次を実行します。

1. xelab を dpiheader オプションを使用して実行し、file.h ヘッダー ファイルを作成します。
2. file.c のコードを xelab で生成されたヘッダー ファイル (file.h) に含めます。これは最後にリストされています。
3. 前と同様 file.c および test.sv のコードをコンパイルして、シミュレーション実行ファイルを生成します。

file.c

```

#include "file.h"
/* NOTE: This file is generated by xelab -dpiheader <filename> flow */
int cfunc (int a, int b) {
  //Call the function exported from SV.
  return c_exported_func (a,b);
}

```

test.sv

```

module m();
export "DPI-C" c_exported_func = function func;
import "DPI-C" pure function int cfunc (input int a ,b);
/*This function can be called from both SV or C side. */
function int func(input int x, y);
begin
  func = x + y;
end
endfunction
int z;
initial
begin
  #5;
  z = cfunc(2, 3);
  if(z == 5)
    $display("PASSED");
  else
    $display("FAILED");
  end
endmodule

```

run.ksh

```
xelab -dpiheader file.h -svlog test.sv
xsc file.c
xelab -svlog test.sv -sv_lib dpi -R
file.h
/*****
/* ----- */
/* / \ / / */
/* /---/ \ / */
/* \ \ \ / */
/* \ \ Copyright (c) 2003-2013 Xilinx, Inc. */
/* / / All Right Reserved. */
/* /---/ \ / */
/* \ \ / \ */
/* \---\ /---\ */
*****/
/* NOTE: DO NOT EDIT. AUTOMATICALLY GENERATED FILE. CHANGES WILL BE LOST. */
#ifndef DPI_H
#define DPI_H
#ifdef __cplusplus
#define DPI_LINKER_DECL extern "C"
#else
#define DPI_LINKER_DECL
#endif
#include "svdpi.h"
/* Exported (from SV) function */
DPI_LINKER_DECL DPI_DLLISPEC
int c_exported_func(
int x, int y);
/* Imported (by SV) function */
DPI_LINKER_DECL DPI_DLLESPEC
int cfunc(
int a, int b);
#endif
```

Vivado Design Suite に含まれる DPI 例

Vivado Design Suite には、Vivado シミュレータでの DPI の使用方法を理解するのに役立つ 2 つの例が含まれています。これらの例は、<vivado installation dir>/examples/xsim/systemverilog/dpi にあります。まず、このディレクトリに含まれる README ファイルを参照することをお勧めします。含まれる例は、次のとおりです。

- simple_import: 純粋な関数の単純なインポート
- simple_export: 純粋な関数の単純なエクスポート



ヒント: 「純粋な関数」とは、関数の戻り値がその入力値にのみに基づいて計算される関数のことです。

特殊なケースの処理

グローバル リセットとトライステートの使用

ザイリンクス デバイスには、デバイスのすべてのレジスタに接続されている専用の配線および回路があります。

グローバル セットおよびリセット ネット

コンフィギュレーション中には、専用グローバル セット/リセット (GSR) 信号がアサートされます。GSR 信号はデバイスのコンフィギュレーションが終了するとディアサートされます。すべてのフリップフロップおよびラッチはこのリセットを受信し、レジスタの定義方法によって、セットまたはリセットされます。

コンフィギュレーション後に GSR ネットにアクセスすることはできますが、手動リセットの代わりに GSR 回路を使用しないでください。これは、FPGA デバイスではシステム リセットなどのファンアウトの大きい信号に対して、高速バックボーン配線が提供されているからです。このバックボーン配線は専用 GSR 回路よりも高速で、GSR 信号を転送する専用グローバル配線よりも簡単に解析できます。

合成後およびインプリメンテーション後のシミュレーションでは、GSR 信号は自動的に最初の 100 ns 間アサートされ、コンフィギュレーション後に発生するリセットがシミュレーションされます。

オプションで、GSR パルスを合成前の論理シミュレーションで供給できますが、すべてのレジスタをリセットするローカル リセットがデザインに含まれている場合は必要ありません。



ヒント: テストベンチを作成する際は、合成後およびインプリメンテーション後のシミュレーションでは GSR パルスが自動的に発生することを考慮してください。このパルスにより、すべてのレジスタがシミュレーションの最初の 100 ns 間リセット状態になります。

注記: デザインで ICAP プリミティブが使用される場合、GSR はその時点で 1.281 us 間続きます。

グローバル トライステート ネット

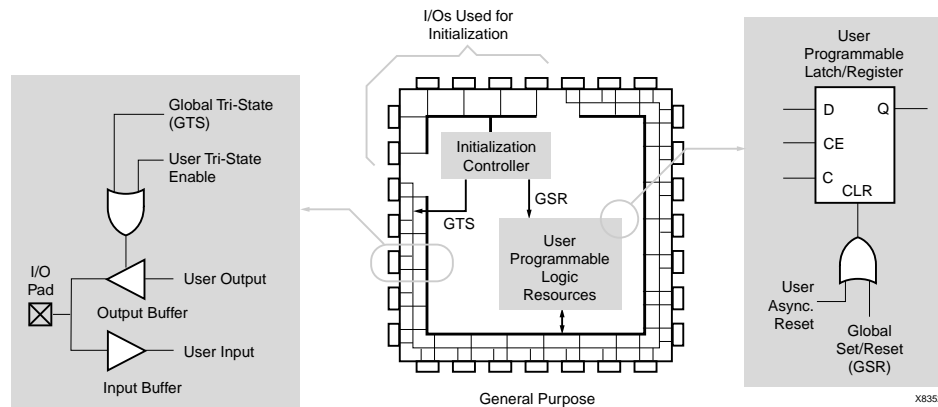
専用グローバル GSR だけでなく、専用グローバル トライステート (GTS) ネットを使用してコンフィギュレーションモード中に出力バッファをハイ インピーダンス ステートに設定することもできます。通常の操作中、汎用出力が標準、トライステート、双方向出力であるにかかわらず、すべての汎用出力に適用されます。このため、FPGA がコンフィギュレーションされたときに、出力が誤ってほかのデバイスを駆動することはありません。

シミュレーションでは、GTS 信号は通常駆動されません。GTS を駆動する回路は、合成後およびインプリメンテーション後のシミュレーションで使用可能であり、早期 (合成前) 論理シミュレーションではオプションで追加できますが、GTS パルス幅はデフォルトで 0 に設定されます。

グローバル トライステートおよびグローバル セット/リセット信号の使用

次の図に、グローバル トライステート (GTS) およびグローバル セット/リセット (GSR) 信号が FPGA でどのように使用されるかを示します。

図 53: ビルトイン FPGA 初期化回路図



Verilog の GSR および GTS

グローバル セット/リセット (GSR) およびグローバル トライステート (GTS) 信号は `<Vivado_Install_Dir>/data/verilog/src/glbl.v` モジュールで定義されています。

ほとんどの場合、GSR および GTS をテストベンチで定義する必要はありません。

`glbl.v` ファイルはグローバル GSR および GTS 信号を宣言し、自動的に GSR に 100 ns 間パルスを供給します。

VHDL の GSR および GTS

グローバル セット/リセット (GSR) およびグローバル トライステート (GTS) 信号は `<Vivado_Install_Dir>/data/vhdl/src/unisims/primitive/GLBL_VHD.vhd` モジュールで定義されています。

GLBL_VHD コンポーネントを使用するには、テストベンチにインスタンス化する必要があります。

GLBL_VHD ファイルはグローバル GSR および GTS 信号を宣言し、自動的に GSR に 100 ns 間パルスを供給します。

次のコードは、テストベンチに GLBL_VHD をインスタンス化し、ROC (Reset on Configuration) のアサート パルス幅を 90 ns に変更する例を示しています。

```
GLBL_VHD inst:GLBL_VHD generic map (ROC_WIDTH => 90000);
```

デルタ サイクルとレース コンディション

この資料では、イベント ベースのシミュレータについて説明します。イベント シミュレータでは、任意のシミュレーション時間に複数のイベントを処理できます。これらのイベントが処理中のとき、シミュレータはシミュレーション時間を進めることはできません。このイベント処理時間は、一般的に「デルタ サイクル」と呼ばれます。任意のシミュレーション タイム ステップには複数のデルタ サイクルを含めることができます。

シミュレーション時間は、そのシミュレーション時間で処理するトランザクションがない場合にのみ進みます。このため、イベントが 1 つのタイム ステップ内でスケジュールされる場合、シミュレータで予期しない結果となる可能性があります。次の VHDL コードは、その例を示しています。

予期しない結果になる VHDL コード例

```
clk_b <= clk;
clk_prcs : process (clk)
begin
    if (clk'event and clk='1') then
        result <= data;
    end if;
end process;
clk_b_prcs : process (clk_b)
begin
    if (clk_b'event and clk_b='1') then
        result1 <= result;
    end if;
end process;
```

この例には、2 つの同期プロセスがあります。

- clk_prcs
- clk_b_prcs

シミュレータは、シミュレーション時間を進める前に `clk_b <= clk` 代入を実行します。この結果、2 クロック エッジ内で発生するはずのイベントが 1 クロック エッジ内で発生し、レース コンディションになります。

このような状況を回避するため、次を推奨します。

- クロックとデータは同時に変更しないでください。出力ごとに遅延を挿入します。
- 同じクロックを使用します。
- 次の例に示すように、一時的な信号を使用してデルタ遅延を強制します。

```
clk_b <= clk;
clk_prcs : process (clk)
begin
    if (clk'event and clk='1') then
        result <= data;
    end if;
end process;
result_temp <= result;
clk_b_prcs : process (clk_b)
```

```
begin
  if (clk_b'event and clk_b='1') then
    result1 <= result_temp;
  end if;
end process;
```

ほとんどのイベント ベース シミュレータでデルタ サイクルを表示できます。シミュレーション問題をデバッグする際は、この表示を利用してください。

ASYNC_REG 制約の使用

ASYNC_REG 制約には、次のような特徴があります。

- デザインの非同期レジスタを特定します。
- これらのレジスタの X 伝搬をディスエーブルにします。

ASYNC_REG 制約は、次のいずれかを使用してフロントエンド デザインのレジスタに適用できます。

- HDL コードの属性
- ザイリンクス デザイン制約 (XDC)

ASYNC_REG が適用されているレジスタでは、タイミング シミュレーション中に前の値が保持され、シミュレーションで X は出力されません。新しい値が供給されている可能性もあるので、注意してください。

ASYNC_REG 制約は、CLB および入出力ブロック (IOB) レジスタおよびラッチにのみ適用できます。ASYNC_REG 制約の詳細は、『Vivado Design Suite プロパティ リファレンス ガイド』(UG912) の[このセクション](#)を参照してください。

非同期データを供給するのを回避できない場合は、IOB または CLB レジスタにのみに供給してください。RAM、シフト レジスタ LUT (SRL)、またはその他の同期エレメントに非同期信号を供給すると、結果が決定的なものにならないので、避けてください。ザイリンクスでは、非同期信号をまずレジスタ、ラッチ、または FIFO で同期化してから、RAM、シフト レジスタ LUT (SRL) などの同期エレメントに書き込むようにすることを強くお勧めします。詳細は、『Vivado Design Suite ユーザー ガイド: 制約の使用』(UG903) を参照してください。

同期エレメントの X 伝搬のディスエーブル

タイミング シミュレーション中にタイミング エラーが発生すると、デフォルトではラッチ、レジスタ、RAM またはその他の同期エレメントから X が出力されます。これは、実際の出力値が不明であるからです。この場合、レジスタの出力は次のいずれかになります。

- 前の値を保持
- 新しい値にアップデート
- メタステーブル状態(同期エレメントにクロックが供給された後しばらく経たないと値が確定しない)

この値は決定できず、正確なシミュレーション結果は得られないので、エレメントから不明の値を示す X が出力されます。X 出力は次のクロック サイクルまで保持され、ほかに違反がなければ、次のクロック サイクルで供給された値により出力がアップデートされます。

X が出力されると、シミュレーションに大きく影響することがあります。たとえば、1 つのレジスタで X が生成されると、続くクロック サイクルで、その X がほかに伝搬され、テスト中のデザインの大部分が不明のステートになる可能性があります。

X が生成されないようにするには、次を実行します。

- 同期パスを解析し、このパスまたはほかのパスに関連したタイミング問題を修正して、回路が正しく動作するようにします。
- 非同期パスでタイミング違反を回避できない場合は、ASYNC_REG プロパティを使用して、タイミング違反が発生したときの同期エレメントによる X 伝搬をディスエーブルにします。

X 伝搬をディスエーブルにすると、レジスタの出力には前の値が保持されます。実際のシリコンでは、レジスタの値が新しい値に変更されることもあります。X 伝搬をディスエーブルにすると、シミュレーション結果がシリコンの動作とは一致しない可能性があります。



注意: このオプションを使用する場合は注意が必要です。タイミング違反をこれ以外の方法で回避できない場合にのみ使用してください。

コンフィギュレーション インターフェイスのシミュレーション

このセクションでは、次のコンフィギュレーション インターフェイスのシミュレーションについて説明します。

- JTAG シミュレーション
- SelectMAP シミュレーション

JTAG シミュレーション

BSCAN コンポーネントのシミュレーションはすべてのデバイスでサポートされます。

シミュレーションでは、JTAG ポートと一部の JTAG 操作コマンドがサポートされます。JTAG インターフェイスは、スキャン チェーンへのインターフェイスを含め、完全にはサポートされていません。このインターフェイスをシミュレーションするには、次の手順に従います。

1. BSCANE2 コンポーネントをインスタンス化し、デザインに接続します。
2. JTAG_SIME2 コンポーネントを、デザインにではなくテストベンチにインスタンス化します。

このインターフェイスは次のものになります。

- 外部 JTAG 信号 (TDI、TDO、TCK など) へのインターフェイス
- BSCAN コンポーネントへの通信チャンネル

コンポーネント間の通信は、VPKG VHDL パッケージ ファイルまたは g1b1 Verilog グローバル モジュールで実行されます。そのため、特定の JTAG_SIME2 コンポーネントとデザイン間、または特定の BSCANE2 シンボル間に暗示的接続は必要ありません。

JTAG/BSCAN の動作を理解するため、スティミュラスをテストベンチ内の特定の JTAG_SIME2 コンポーネントから駆動および表示できます。これらのコンポーネントのインスタンス化テンプレートは、Vivado® Design Suite テンプレートおよびそのデバイスのライブラリ ガイドから入手できます。

SelectMAP シミュレーション

インスタンシエーション テンプレートを含むコンフィギュレーション シミュレーション モデル (SIM_CONFIG2 および SIM_CONFIG3) を使用すると、サポートされているコンフィギュレーション インターフェイスをシミュレーションして DONE ピンが High になるのを確認できます。このモデルは、サポート デバイスがサポートされているコンフィギュレーション インターフェイスのスティミュラスにどのように応答するかを示します。

次の表に、サポートされるインターフェイスとデバイスをリストします。

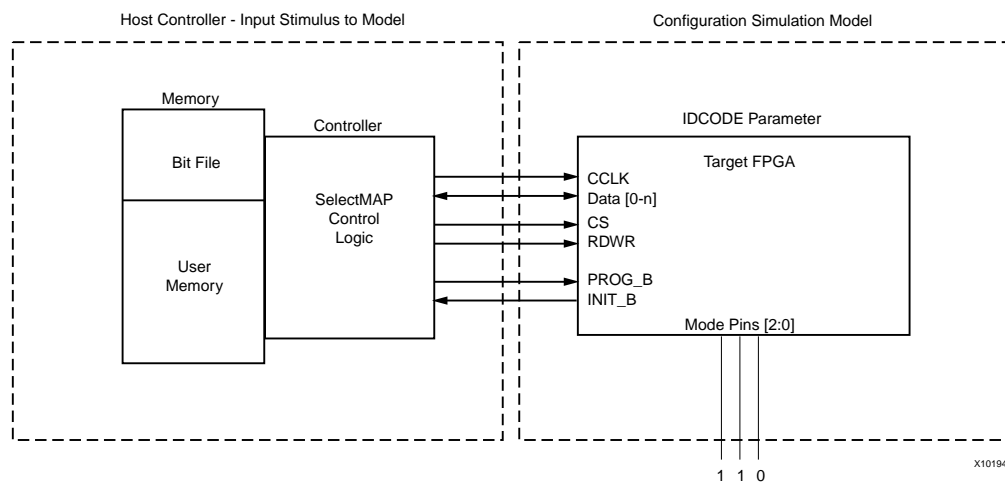
表 47: サポートされるコンフィギュレーション デバイスおよびモード

デバイス	SelectMAP	シリアル	SPI	BPI
7 シリーズおよび Zynq®-7000 SoC デバイス	○	○	×	×
UltraScale デバイス	○	○	×	×
UltraScale+™ デバイス	○	○	×	×

これらのモデルは、制御信号アクティビティを処理し、ビット ファイルをダウンロードします。CRC、IDCODE などの内部レジスタ設定とステータス レジスタが含まれます。同期ワードがデバイスに入力され、スタートアップシーケンスで処理されるところを監視できます。次の図に、システムをハードウェアからシミュレーション環境にマップする方法を示します。

コンフィギュレーション プロセスは、各デバイスのコンフィギュレーション ユーザー ガイドで説明されています。これらのガイドには、コンフィギュレーション シーケンスおよびコンフィギュレーション インターフェイスに関する情報が記載されています。

図 54: モデルの通信を示すブロック図



システム レベルの記述

コンフィギュレーション モデルを使用すると、ハードウェアが使用可能になる前に、コンフィギュレーション インターフェイスの制御ロジックをテストできます。このモデルはデバイス全体をシミュレーションし、次のシステム レベルで使用されます。

- 適切なワイヤ、制御信号処理、データ入力アライメントを確実にするため、コンフィギュレーション ロジックを制御するためにプロセッサを使用するアプリケーション。

- データ アライメントが正しく実行されるよう CS (SelectMAP チップ セレクト) または CLK 信号を使用してデータ読み込みプロセスを制御するアプリケーション。
- SelectMAP ABORT または Readback を実行する必要があるシステム。

config_test_bench.zip ファイルには、SelectMAP ロジックを実行するプロセッサをシミュレーションするサンプル テストベンチが含まれています。これらのテストベンチには、SelectMAP インターフェイスを制御するプロセッサをエミュレートする制御ロジックが含まれ、フル コンフィギュレーション、ABORT、および IDCODE とステータス レジスタの Readback などの機能があります。

このモデルに関連する ZIP ファイルは、ザイリンクス アンサー [53632](#) から入手できます。

シミュレーションされるホスト システムには、ファイルの配布および制御信号管理などの機能が必要です。これらの制御システムは、デバイスのコンフィギュレーション ユーザー ガイドの説明に従って設計する必要があります。

コンフィギュレーション モデルは、BIT ファイルがデバイスに読み込まれる際、コンフィギュレーション プロシージャ中にデバイス内で起きていることも示します。

BIT ファイルのダウンロード中、モデルにより各コマンドが実行され、ハードウェアの変更を反映してレジスタ設定が変更されます。

CRC レジスタで CRC 値が累積されるのを監視できます。また、デバイスがさまざまなコンフィギュレーション ステートを遷移していくときに設定されるステータス レジスタ ビットも示されます。

モデルを使用したデバッグ

各コンフィギュレーション モデルでは、正しいコンフィギュレーションの例が提供されます。デバイス プログラムの問題が発生したときに、デバッグでこの例を参考にすると有益です。

Vivado デバイス プログラマを使用すると、JTAG を介してステータス レジスタを読み出すことができます。レジスタには、デバイスの現在のステータスに関連した情報が含まれるので、デバッグのときに利用できます。ボード上で問題が発生した場合は、Vivado デバイス プログラマでステータス レジスタを読み込むのがデバッグの最初の手順の 1 つです。

ステータス レジスタを読み込んだら、それをシミュレーションにマップし、デバイスのコンフィギュレーション段階を特定できます。

たとえば、データの読み込みが正しく完了すると GHIGH ビットが High になります。このビットが High になっていない場合、データの読み込みは完了していません。BitGen でセットされ、スタートアップ シーケンスで解放される GTW、GWE および DONE 信号も監視できます。

コンフィギュレーション モデルでは、エラー挿入も可能です。問題が発生してデータの読み込みが一時停止されて再開した場合、アクティブな CRC ロジックで問題が検出されます。BIT ファイルに手動で挿入されたビット フリップも検出され、このエラーがデバイスで処理されるのと同様に処理されます。

サポートされる機能

デバイス別のコンフィギュレーション ユーザー ガイドには、各コンフィギュレーション インターフェイスとの通信にサポートされる方法が示されています。次の表に、コンフィギュレーション ユーザー ガイドで説明される機能の中で、サポートされているものを示します。

SIM_CONFIG2 モデルには、次のような特徴があります。

- コンフィギュレーション データのリードバックはサポートされません。
- CRC 値は計算されますが、供給されるコンフィギュレーション データは格納されません。

- デバイスに対して有効なコマンド シーケンスおよび信号処理が提供されていることを確認するため、特定のレジスタのみでのリードバックが可能です。
- リードバック データ ファイルを生成するためのモデルではありません。

表 48: モデルをサポートするスレーブ SelectMAP およびシリアル機能

スレーブ SelectMAP およびシリアル機能	サポートあり
マスター モード	なし
デジター チェーン - スレーブ パラレル デジター チェーン	なし
SelectMAP データ読み込み	あり
連続 SelectMAP データ読み込み	あり
非連続 SelectMAP データ読み込み	あり
SelectMAP ABORT	あり
SelectMAP リコンフィギュレーション	なし
SelectMAP データ順序付け	あり
リコンフィギュレーションおよびマルチブート	なし
コンフィギュレーション CRC - コンフィギュレーション中の CRC チェック	あり
コンフィギュレーション CRC - コンフィギュレーション後の CRC	なし

シミュレーションでのブロック RAM の競合チェックのデイスエーブル

ザイリンクス ブロック RAM メモリは完全なデュアル ポート RAM で、両方のポートでいつでもどのメモリ ロケーションにもアクセスできますが、同じアドレス空間に対して同時に読み出しおよび書き込みが実行されると、ブロック RAM アドレスの競合が発生します。これは、読み出しポートから読み出されるデータが有効ではないために発生します。

ハードウェアでは、読み出される値は、以前のデータ、新しいデータ、または以前のデータと新しいデータの組み合わせとなります。

シミュレーションでは、読み出された値が不明となるため X が出力されます。ブロック RAM の競合の詳細は、デバイスのユーザー ガイドを参照してください。

アプリケーションによっては、この状態を回避したり、設計を変更したりできないことがあります。その場合、ブロック RAM でこれらの違反が検出されないように設定できます。これは、ブロック RAM プリミティブのジェネリック (VHDL) またはパラメーター (Verilog) の `SIM_COLLISION_CHECK` 文字列で制御します。

次の表に、競合が発生したときのシミュレーション ビヘイビアーを制御する `SIM_COLLISION_CHECK` に使用可能な文字列オプションを示します。

表 49: `SIM_COLLISION_CHECK` 文字列

文字列	書き込み競合メッセージ	出力への X の書き込み
ALL	あり	あり
WARNING_ONLY	あり	なし。競合発生時にのみ適用され、同じアドレス空間への後続の読み出しで、出力に X が示される可能性があります。

表 49: SIM_COLLISION_CHECK 文字列 (続き)

文字列	書き込み競合メッセージ	出力への X の書き込み
GENERATE_X_ONLY	なし	あり
None	なし	なし。競合発生時にのみ適用され、同じアドレス空間への後続の読み出しで、出力に X が示される可能性があります。

インスタンス レベルで SIM_COLLISION_CHECK を使用すると、各ブロック RAM インターフェイスの設定を変更できます。

消費電力解析のための SAIF ファイルの出力

- Vivado シミュレータ: [Vivado シミュレータを使用した消費電力解析](#)
- [第 3 章: サードパーティ シミュレータを使用したシミュレーションの消費電力解析用の SAIF の出力](#)、IES での SAIF の出力、および VCS での SAIF の出力

コンパイルまたはシミュレーションのスキップ

コンパイルのスキップ

既存のスナップショットにシミュレーションを実行し、シミュレーション ファイルセットに SKIP_COMPILATION プロパティを設定して、デザインのコンパイル (または再コンパイル) をスキップできます。

```
set_property SKIP_COMPILATION 1 [get_filesets sim_1]
```

注記: このプロパティを設定すると、前回のコンパイル後に適用されたデザイン ファイルへの変更はシミュレーションに反映されません。

シミュレーションのスキップ

シミュレーションを実行せずに、シミュレーション スナップショットをエラボレートおよびコンパイルして、デザイン HDL ファイルに対してセマンティック チェックを実行するには、シミュレーション ファイルセットに SKIP_SIMULATION プロパティを設定します。

```
set_property SKIP_SIMULATION true [get_filesets sim_1]
```



重要: 上記のいずれかのプロパティを使用する場合は、[Project Settings] ダイアログ ボックスの [Simulation] ページで [Clean up simulation files] チェック ボックスをオフにします。バッチ/Tcl モードで実行している場合は、`launch_simulation` を `-noclean_dir` を使用して実行します。

Vivado シミュレータ Tcl コマンドの値の規則

この付録では、Tcl コマンドの `add_force` および `set_value` の両方に適用される値の規則を示します。

文字列の値の解釈

文字列の値は、HDL オブジェクトの宣言型と `-radix` コマンド ライン オプションに基づいて解釈されます。`-radix` の指定が、HDL オブジェクト型で定義されたデフォルトの基数よりも優先されます。

- `logic` 型の HDL オブジェクトの場合、値は `logic` 型の 1 次元配列または指定した基数の桁数の文字列です。
 - 文字列のビット数が型のビット数よりも少ない場合、文字列がその型のビット数に一致するようにゼロ拡張 (符号拡張ではない) されます。
 - 文字列のビット数が型のビット数よりも大きい場合、上位の余分なビットが 0 でないと、サイズ不一致エラーになります。

たとえば、基数が 16 進数で 6 ビットの `logic` 配列を使用する 8 ビット (16 進数の 1 桁ごとに 4 ビット) の場合、値 3F は 2 進数で 0011 1111 になります。ただし、上位 2 ビットは 0 なので、値は HDL オブジェクトに代入できます。値が 7F の場合は、上位 2 ビットが 0 ではないので、エラーになります。

- スカラー (配列またはレコードではない) の `logic` 型 HDL オブジェクトの長さは 1 ビットです。
- `a [left:right]` (Verilog) または `a(left TO/DOWNTO right)` と宣言される `logic` 配列の場合、拡張/切り捨て後の一番左の値ビットが `a[left]` に代入され、一番右の値ビットが `a[right]` に代入されます。

Vivado Design Suite シミュレーション ロジック

ロジックは、HDL で定義される概念ではなく、Vivado® シミュレータによるヒューリスティックです。

- Verilog オブジェクトは、Verilog の `bit` 型であることが暗示される場合、`logic` 型であるとみなされます。これには `wire` および `reg` オブジェクトや整数、時間が含まれます。
- VHDL オブジェクトは、オブジェクト型が `bit`、`logic`、または列挙型 (列挙子が `std_logic` の列挙型のサブセットで少なくとも 0 および 1 が含まれる) の場合、またはオブジェクトの型がこのような型の 1 次元配列である場合、`std_logic` 型であるとみなされます。
- VHDL 列挙型の HDL オブジェクトの場合、値は列挙子リテラルの 1 つになります。列挙子が文字リテラルではない場合シングル クォーテーション (') は含めません。基数は無視されます。
- 整数型の VHDL オブジェクトの場合、値をその型の範囲内の符号付き 10 進数整数にできます。基数は無視されます。

- VHDL および Verilog の浮動小数点型の場合、値は浮動小数点になります。基数は無視されます。
- すべての型の HDL オブジェクトで、Tcl コマンド set による値の設定はサポートされません。

Vivado シミュレータの混合言語サポート および例外

Vivado 統合設計環境 (IDE) では、次の言語がサポートされます。

- VHDL ([IEEE Standard VHDL Language Reference Manual](#)) (IEEE-STD-1076-1993) を参照)
- Verilog ([IEEE Standard Verilog Hardware Description Language](#)) (IEEE-STD-1364-2001) を参照)
- SystemVerilog の合成可能なサブセット ([IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language](#)) (IEEE-STD-1800-2009) を参照)
- IEEE P1735 暗号化 ([Recommended Practice for Encryption and Management of Electronic Design Intellectual Property \(IP\)](#)) (IEEE-STD-P1735) を参照)

この付録では、Vivado シミュレータの混合言語アプリケーションおよび、Verilog、System Verilog、および VHDL サポートの例外をリストします。

混合言語シミュレーションの使用

Vivado シミュレータでは、混合言語プロジェクト ファイルおよび混合言語シミュレーションがサポートされるので、VHDL デザインに Verilog/SystemVerilog (SV) モジュールを含めることができ、またその逆も可能です。

シミュレーションでの混合言語の制限

- VHDL デザインに Verilog/SystemVerilog (SV) モジュールをインスタンス化でき、Verilog/SV デザインに VHDL コンポーネントをインスタンス化できます。Verilog/SV モジュールと VHDL コンポーネントの結合には、コンポーネント インスタンス化ベースのデフォルトの結合が使用されます。VHDL プロセス文で Verilog 関数を呼び出すというような VHDL と Verilog の混合使用はサポートされません。
- Verilog/SV モジュールへの境界には、VHDL 型、ジェネリック、およびポートのサブセットを使用できます。同様に、VHDL コンポーネントへの境界には Verilog/SV 型、パラメーター、およびポートのサブセットを使用できます。[表 51: サポートされる VHDL および Verilog データ型](#) を参照してください。



重要: VHDL レコード オブジェクトの Verilog オブジェクトへの接続はサポートされていませんが、サポートされる型の VHDL レコード エLEMENT は互換性のある Verilog ポートに接続できます。

- Verilog/SV 階層参照では VHDL ユニットは参照できず、VHDL 拡張/選択名では Verilog/SV ユニットは参照できません。

混合言語シミュレーションでの重要な手順

1. 混合言語プロジェクトのデザイン ライブラリ内で VHDL コンポーネントまたは Verilog/SV モジュールの検索順を指定できます (オプション)。

2. `xelab -L` を使用すると、混合言語プロジェクトのデザイン ライブラリ内で VHDL コンポーネントまたは Verilog/SV モジュールの結合順を指定できます。

注記: Verilog モジュールと別の Verilog モジュールの結合にも、`-L` で指定したライブラリ検索順が使用されます。

混合言語の結合と検索

VHDL コンポーネントを Verilog/SV モジュールに、または Verilog/SV モジュールを VHDL アーキテクチャにインスタンス化するには、`xelab` コマンドを使用します。

- まず、インスタンス化するデザイン ユニットと同じ言語のユニットが検索されます。
- 同じ言語のユニットが見つからない場合は、`xelab` により `-L` オプションで指定されたライブラリで両方の言語のデザイン ユニットが検索されます。

この検索順は、`xelab` コマンド ラインでライブラリが表示される順序と同じです。

注記: Vivado IDE を使用する場合、ライブラリ検索順は自動的に指定されます。ユーザーによる設定は必要であり、また不可能です。

関連情報

[Verilog 検索順序](#)

混合言語コンポーネントのインスタンス化

混合言語デザインでは、次のセクションで説明するように、Verilog/SV モジュールを VHDL アーキテクチャに、または VHDL コンポーネントを Verilog/SV モジュールにインスタンス化できます。

ポート タイプが一致しているかどうかは、[ポート マップおよびサポートされるポート型](#)を参照してください。

VHDL デザイン ユニットへの Verilog モジュールのインスタンス化

1. VHDL コンポーネントを、インスタンス化する Verilog モジュールと同じ名前 (大文字と小文字の区別も含む) で宣言します。次に例を示します。

```
COMPONENT MY_VHDL_UNIT PORT (
  Q : out STD_ULOGIC;
  D : in  STD_ULOGIC;
  C : in  STD_ULOGIC );
END COMPONENT;
```

2. 名前または位置の関連付けを使用し、Verilog モジュールをインスタンス化します。次に例を示します。

```
UUT : MY_VHDL_UNIT PORT MAP (
  Q => O,
  D => I,
  C => CLK);
```

Verilog/SV デザイン ユニットへの VHDL コンポーネントのインスタンス化

Verilog/SV デザイン ユニットに VHDL コンポーネントをインスタンス化するには、VHDL コンポーネントを Verilog/SV モジュールのようにインスタンス化する必要があります。

次に例を示します。

```
module testbench ;
wire in, clk;
wire out;
FD FD1(
.Q(Q_OUT),
.C(CLK);
.D(A);
);
```

ポート マップおよびサポートされるポート型

次の表に、サポートされるポート型をリストします。

表 50: サポートされるポート型

VHDL ¹	Verilog/SV ²
IN	INPUT
OUT	OUTPUT
INOUT	INOUT

注記:

- VHDL のバッファおよびリンケージ ポートはサポートされません。
- Verilog の双方向パス オプションへの接続はサポートされません。名前の付いていない Verilog ポートは混合デザインの境界では使用できません。

次の表に、混言言語デザインの境界でサポートされるポートの VHDL および Verilog データ型を示します。

表 51: サポートされる VHDL および Verilog データ型

VHDL ポート	Verilog ポート
bit	ネット
std_logic	ネット
bit_vector	vector net
signed	vector net
unsigned	vector net
std_ulogic_vector	vector net
std_logic_vector	vector net

注記: Verilog 出力ポートの `reg` 型は、混言言語の境界でサポートされます。境界上では、出力ポート `reg` が出力ネット (ワイヤ) ポートのように処理されます。これ以外の型が混言言語の境界で使用されると、エラーとなります。

注記: Vivado シミュレータでは、レコード エレメントは混合ドメインにインスタンス化される Verilog モジュールのポート マップのアクチュアルとしてサポートされます。VHDL ポートとしてサポートされる型 (表 51: サポートされる VHDL および Verilog データ型 にリスト) も、すべてレコード エレメントとしてサポートされます。

表 52: サポートされる SV および VHDL データ型

SV データ型	VHDL データ型
Int	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
byte	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
shortint	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
longint	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
integer	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
bit(1D) のベクター	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
logic(1D) のベクター	
	bit_vector
	std_logic_Vector

表 52: サポートされる SV および VHDL データ型 (続き)

SV データ型	VHDL データ型
	std_ulogic_vector
	signed
	unsigned
reg(1D) のベクター	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
logic/bit	
	bit
	std_logic
	std_ulogic
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned

注記: VHDL エンティティに real ポートを持つ Verilog モジュールをインスタンス化することはサポートされません。

ジェネリック (パラメーター) のマップ

Vivado シミュレータでは、次の VHDL ジェネリック型 (および同等の Verilog/SV) がサポートされます。

- 整数
- real
- 文字列
- boolean

注記: これ以外のジェネリック型が混合言語の境界で使用されると、エラーとなります。

VHDL および Verilog 値のマップ

次の表に、std_logic および bit への Verilog ステートのマップをリストします。

表 53: std_logic および bit にマップされる Verilog ステート

Verilog	std_logic	bit
Z	Z	0
0	0	0

表 53: std_logic および bit にマップされる Verilog ステート (続き)

Verilog	std_logic	bit
1	1	1
X	X	0

注記: Verilog の駆動電流は無視されます。VHDL には、これに対応するマップがありません。

次の表に、Verilog ステートにマップされる VHDL 型 `bit` をリストします。

表 54: Verilog ステートにマップされる VHDL 型 bit

bit	Verilog
0	0
1	1

次の表に、Verilog ステートにマップされる VHDL 型 `std_logic` をリストします。

表 55: Verilog ステートにマップされる VHDL 型 std_logic

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0
H	1
-	X

Verilog では大文字/小文字が区別されるので、コンポーネント宣言で使用する関連付けおよびローカル ポート名は Verilog ポート名の大文字/小文字と一致している必要があります。

VHDL 言語サポートの例外

言語のコンストラクトの中には、Vivado シミュレータでサポートされないものもあります。次の表に、VHDL 言語サポートの例外をリストします。

表 56: VHDL 言語サポートの例外

サポートされる VHDL コンストラクト	例外
<code>abstract_literal</code>	基底付きリテラルとして表現される浮動小数点はサポートされません。

表 56: VHDL 言語サポートの例外 (続き)

サポートされる VHDL コンストラクト	例外
alias_declaration	オブジェクト以外へのエイリアスは通常、特に次の場合サポートされません。 <ul style="list-style-type: none"> エイリアスのエイリアス subtype_indication なしのエイリアス宣言 エイリアス宣言のシグネチャ alias_designator としての演算子シンボル 演算子シンボルのエイリアス エイリアス宣言としての文字リテラル
alias_designator	alias_designator としての operator_symbol alias_designator としての character_literal
association_element	結合エレメントのアクチュアルのスライスには、グローバル、ローカルなスタティック範囲を使用できます。
attribute_name	接頭語の後のシグネチャはサポートされません。
binding_indication	entity_aspect を使用しない binding_indication はサポートされません。
bit_string_literal	空の bit_string_literal (" ") はサポートされません。
block_statement	guard_expression はサポートされません。たとえば、保護付きブロック、保護付き信号、保護付きターゲット、および保護付き代入はサポートされません。
choice	case 文で集合体を選択として使用することはサポートされません。
concurrent_assertion_statement	実行延期はサポートされていません。
concurrent_signal_assignment_statement	実行延期はサポートされていません。
concurrent_statement	wait 文を含む並列手続き呼び出しはサポートされません。
conditional_signal_assignment	オプションの一部であるキーワード guarded は、保護付き信号代入はサポートされないため、サポートされません。
configuration_declaration	コンフィギュレーションで使用する generate インデックスのローカルではないスタティックはサポートされません。
entity_class	エンティティ クラスとしてのリテラル、ユニット、ファイル、グループはサポートされません。
entity_class_entry	グループ テンプレートと共に使用するためのオプションの <> はサポートされません。
file_logical_name	file_logical_name では、ワイルドカードを使用して文字列値を評価できますが、ファイル名として使用できるのは文字列リテラルおよび識別子のみです。
function_call	function_call 内の名前によるパラメーター結合では、スライス、インデックス、およびフォーマルの選択はサポートされません。
instantiated_unit	ダイレクト コンフィギュレーション インスタンス化はサポートされません。
mode	リンケージおよびバッファー ポートは完全にはサポートされません。
options	guarded はサポートされません。
primary	primary が使用された場所で、アロケーターが拡張されます。

表 56: VHDL 言語サポートの例外 (続き)

サポートされる VHDL コンストラクト	例外
<code>procedure_call</code>	<code>procedure_call</code> 内の名前によるパラメーター結合では、スライス、インデックス、およびフォーマルの選択はサポートされません。
<code>process_statement</code>	実行延期プロセスはサポートされません。
<code>selected_signal_assignment</code>	オプションの一部であるキーワード <code>guarded</code> は、保護付き信号代入はサポートされないの、サポートされません。
<code>signal_declaration</code>	<code>signal_kind</code> はサポートされません。 <code>signal_kind</code> はサポートされない保護付き信号を宣言するために使用されます。
<code>subtype_indication</code>	分解された複合体 (配列およびレコード) のサブタイプはサポートされません。
<code>waveform</code>	<code>unaffected</code> はサポートされていません。
<code>waveform_element</code>	空の波形エレメントは、保護付き信号にのみ関連するためサポートされません。

Verilog 言語サポートの例外

次の表に、Verilog の言語サポートの例外をリストします。

表 57: Verilog 言語サポートの例外

Verilog コンストラクト	例外
コンパイラ指示子のコンストラクト	
<code>`unconnected_drive</code>	サポートなし
<code>`nounconnected_drive</code>	サポートなし
属性	
<code>attribute_instance</code>	サポートなし
<code>attr_spec</code>	サポートなし
<code>attr_name</code>	サポートなし
プリミティブ ゲートおよびスイッチ タイプ	
<code>cmos_switchtype</code>	サポートなし
<code>mos_switchtype</code>	サポートなし
<code>pass_en_switchtype</code>	サポートなし
生成されたインスタンスエーション	

表 57: Verilog 言語サポートの例外 (続き)

Verilog コンストラクト	例外
generated_instantiation	<p>module_or_generate_item 代替はサポートされません。</p> <p>IEEE 規格からの出力 (『IEEE Standard Verilog Hardware Description Language』 (IEEE-IEEE -1364-2001) セクション 13.2 を参照):</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_block module_or_generate_item</pre> <p>シミュレータでサポートされる出力</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_blockgenerate_condition</pre>
genvar_assignment	<p>部分的にサポート。</p> <p>すべての generate ブロックに名前を付ける必要があります。</p> <p>規格からの出力 (『IEEE Standard Verilog Hardware Description Language』 (IEEE-IEEE -1364-2001) セクション 13.2 を参照):</p> <pre>generate_block ::= begin [: generate_block_identifier] { generate_item } end</pre> <p>シミュレータでサポートされる出力</p> <pre>generate_block ::= begin: generate_block_identifier { generate_item } end</pre>
ソース テキスト コンストラクト	
ライブラリ ソース テキスト	
library_text	サポートなし
library_descriptions	サポートなし
library_declaration	サポートなし
include_statement	ライブラリ マップ ファイル内の include 文を参照します (『IEEE Standard Verilog Hardware Description Language』 (IEEE-STD-1364-2001) セクション 13.2 を参照)。`include コンパイラ指示子は参照しません。
システム タイミング チェック コマンド	
\$skew_timing_check	サポートなし
\$timeskew_timing_check	サポートなし
\$fullskew_timing_check	サポートなし
\$nochange_timing_check	サポートなし
システム タイミング チェック コマンドの引数	
checktime_condition	サポートなし

表 57: Verilog 言語サポートの例外 (続き)

Verilog コンストラクト	例外
PLA モデリング タスク	
\$async\$nand\$array	サポートなし
\$async\$nor\$array	サポートなし
\$async\$or\$array	サポートなし
\$sync\$and\$array	サポートなし
\$sync\$nand\$array	サポートなし
\$sync\$nor\$array	サポートなし
\$sync\$or\$array	サポートなし
\$async\$and\$plane	サポートなし
\$async\$nand\$plane	サポートなし
\$async\$nor\$plane	サポートなし
\$async\$or\$plane	サポートなし
\$sync\$and\$plane	サポートなし
\$sync\$nand\$plane	サポートなし
\$sync\$nor\$plane	サポートなし
\$sync\$or\$plane	サポートなし
VCD (Value Change Dump) ファイル	
\$dumpportson \$dumpports \$dumpportsoff \$dumpportsflush \$dumpportslimit \$vcdplus	サポートなし

Vivado シミュレータ クイック リファレンス ガイド

次の表に、よく使用される Vivado® シミュレータ コマンドのクイック リファレンスおよび例を示します。

HDL ファイルの解析		
Vivado シミュレータでは、Verilog、SystemVerilog および VHDL の 3 つのタイプの HDL ファイルがサポートされています。これらのサポートされるファイルは、XVHDL および XVLOG コマンドを使用して解析できます。		
VHDL ファイルの解析	xvhdl file1.vhd file2.vhd xvhdl -work worklib file1.vhd file2.vhd xvhdl -prj files.prj	
Verilog ファイルの解析	xvlog file1.v file2.v xvlog -work worklib file1.v file2.v xvlog -prj files.prj	
SystemVerilog ファイルの解析	xvlog -sv file1.v file2.v xvlog -work worklib -sv file1.v file2.v xvlog -prj files.prj PRJ ファイル形式に関する詳細は、 プロジェクト ファイル (.prj) の構文 を参照してください。	
その他の xvlog および xvhdl オプション		
xvlog および xvhdl の主なオプション	コマンド オプションの全リストは、 表 16: xelab、xvhdl、xvlog のコマンド オプション を参照してください。 次は、xvlog および xvhdl の主なオプションです。	
	主なオプション	適用先
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	xelab、xvhdl、xvlog xsim コマンド オプション	xvhdl、vlog
	xelab、xvhdl、xvlog xsim コマンド オプション	xvlog、xvhdl
	実行可能なスナップショットのエラポレートおよび生成	

解析後は、XELAB コマンドを使用して、デザインを Vivado シミュレータでエラボレートできます。XELAB では、実行可能なスナップショットが生成されます。

解析段階を飛ばして直接 XELAB コマンドを実行し、PRJ ファイルを渡すこともできます。XELAB でファイルを解析するため XVLOG および XVHDL が呼び出されます。

使用法	xelab top1 top2	2つの最上位デザインユニット (top1 および top2) を含むデザインをエラボレートします。この例では、デザインユニットが work ライブラリにコンパイルされます。
	xelab lib1.top1 lib2.top2	2つの最上位デザインユニット (top1 および top2) を含むデザインをエラボレートします。この例では、デザインユニットがそれぞれ lib1 および lib2 にコンパイルされます。
	xelab top1 top2 -prj files.prj	2つの最上位デザインユニット (top1 および top2) を含むデザインをエラボレートします。この例では、デザインユニットが work ライブラリにコンパイルされます。files.prj ファイルには、次のようなコードが含まれます。
	xelab top1 top2 -s top	2つの最上位デザインユニット (top1 および top2) を含むデザインをエラボレートします。この例では、デザインユニットが work ライブラリにコンパイルされます。コンパイル後、xelab により top という名前の実行可能なスナップショットが生成されます。-s top オプションを指定しない場合、xelab ですべてのユニット名を連結してスナップショットが作成されます。

コマンドラインヘルプおよび xelab オプション	xelab -help 表 16: xelab、xvhd、xvlog のコマンド オプション
---------------------------	---------------------------------------------------

シミュレーションの実行

解析、エラボレート、コンパイル段階が問題なく終了すると、xsim によりシミュレーションを実行する実行可能なスナップショットが生成されます。

使用法	xsim top -R	デザインを最後までシミュレーションします。
	xsim top -gui	Vivado シミュレータ ワークスペース (GUI) を開きます。
	xsim top	Vivado Design Suite コマンド プロンプトを Tcl モードで開きます。ここから次のオプションを実行できます。

重要なショートカット

解析、エラボレーション、実行ファイル生成、シミュレーションを 1～3 段階で実行できます。

	3 段階	xvlog bot.v xvhdl top.vhd xelab work.top -s top xsim top -R
	2 段階	xelab -prj my_prj.prj work.top -s top xsim top -R my_prj.prj ファイルには、次が含まれます。 verilog work bot.v vhd work top.vhd
	1 段階	xelab -prj my_prj.prj work.top -s top -R my_prj.prj ファイルには、次が含まれます。 verilog work bot.v vhd work top.vhd

注記: デザインに UVM コンストラクトが含まれている場合は、xvlog および xelab コマンドに `-L uvm` を渡します。

Vivado シミュレーションの Tcl コマンド

次に、よく使用される Tcl コマンドを示します。すべてのコマンドのリストは、Tcl コンソールに次のコマンドを入力してください。

```
load_features simulator
```

```
help -category simulation
```

Tcl コマンドの詳細は、「`-help <Tcl_command>`」と入力してください。

よく使用される Vivado シミュレータの Tcl コマンド	<code>add_bp</code>	HDL ソースの行にブレークポイントを追加します。
	<code>add_force</code>	信号、ワイヤ、またはレジスタを特定の値に強制的に設定します。Tcl コマンド例は、 force コマンドの使用 を参照してください。
	<code>current_time</code> <code>now</code>	現在のシミュレーション時間をレポートします。Tcl スクリプトでのこのコマンドの例は、 -tclbatch オプションの使用 を参照してください。
	<code>current_scope</code>	現在の作業中の HDL スコープをレポートまたは設定します。詳細は、 [Scopes] ウィンドウ を参照してください。
	<code>get_objects</code>	1 つまたは複数の HDL スコープで、指定したパターンごとに HDL オブジェクトのリストを取得します。コマンドの使用例は、 SAIF コマンドの例 を参照してください。
	<code>get_scopes</code>	子 HDL スコープのリストを取得します。詳細は、 [Scopes] ウィンドウ を参照してください。
	<code>get_value</code>	選択した HDL オブジェクト (変数、信号、ワイヤ、レジスタ) の現在の値を取得します。詳細は、Tcl コンソールに <code>get_value -help</code> と入力すると取得できます。
	<code>launch_simulation</code>	Vivado シミュレータを使用してシミュレーションを実行します。
	<code>remove_bps</code>	シミュレーションからブレークポイントを削除します。
	<code>report_drivers</code>	HDL ワイヤまたは信号オブジェクトのドライバーと現在の駆動値を表示します。詳細は、 Tcl コマンド report_drivers の使用 を参照してください。
	<code>report_values</code>	指定した HDL オブジェクト (変数、信号、ワイヤ、またはレジスタ) の現在のシミュレーション値を表示します。Tcl コマンド例は、 [Scopes] ウィンドウ を参照してください。
	<code>restart</code>	シミュレーションをデザインを読み込んだ後の状態に戻し、時間を 0 に設定します。
	<code>set_value</code>	HDL オブジェクト (変数、信号、ワイヤ、またはレジスタ) を特定の値に設定します。詳細は、 付録 G: Vivado シミュレータ Tcl コマンドの値の規則 を参照してください。
	<code>step</code>	シミュレーションを次の文に進めます。 シミュレーションのステップ実行 を参照してください。

ザイリンクス シミュレータ インターフェイスの使用

XSI (ザイリンクス Simulator Interface) は、C/C++ プログラムが HDL デザインのテストベンチとして機能するようにするザイリンクス Vivado シミュレータ (xsim) への C/C++ アプリケーションプログラミング インターフェイス (API) です。C/C++ プログラムは、XSI を使用し、HDL デザインをホストする Vivado シミュレータのアクティビティを制御します。

この C/C++ プログラムは、次の方法でシミュレーションを制御します。

- HDL デザインの最上位入力ポートの値の設定
- 一定時間シミュレーションを実行するよう Vivado シミュレータに指示

HDL デザインの最上位出力ポートの値を読み出すことも可能です。

C/C++ プログラムで XSI を使用するには、次の手順に従います。

1. ダイナミック リンクを使用して XSI API 関数を呼び出すよう準備します。
2. C/C++ テストベンチ コードを API 関数を使用して記述します。
3. C/C++ プログラムをコンパイルおよびリンクします。
4. Vivado シミュレータと HDL デザインを 1 つの共有ライブラリにパッケージします。

ダイナミック リンク用に XSI 関数を準備

ザイリンクスでは、XSI 関数を間接的に呼び出すには、ダイナミック リンクを使用すること推奨します。この方法は XSI 関数を直接呼び出すよりも複雑ですが、ダイナミック リンクを使用すると、HDL デザインのコンパイルを C/C++ プログラムのコンパイルとは切り離して実行できます。C/C++ プログラムが実行中でも、HDL デザインをいつでもコンパイルし、読み込むことができます。

ダイナミック リンクを介して関数を呼び出すには、次の手順を実行する必要があります。

1. 関数を含む共有ライブラリを開きます。
2. 関数を名前で検索し、そのポインターを取得します。
3. 関数ポインターを使用して関数を呼び出します。
4. 共有ライブラリを閉じます (オプション)。

手順 1、2、4 では、次の表に示すように、OS 用のライブラリ呼び出しを使用する必要があります。これらの関数の詳細は、ご使用の OS の資料を参照してください。

表 59: OS 別ライブラリ コール

機能	Linux	Windows
ライブラリを開く	<pre>void *dlopen(const char *filename, int flag);</pre>	<pre>HMODULE WINAPI LoadLibrary(_In_ LPCTSTR lpFileName);</pre>
関数を名前で検索	<pre>void *dlsym(void *handle, const char *symbol);</pre>	<pre>FARPROC WINAPI GetProcAddress(_In_ HMODULE hModule, _In_ LPCSTR lpProcName);</pre>
共有ライブラリを閉じる	<pre>int dlclose(void *handle);</pre>	<pre>BOOL WINAPI FreeLibrary(_In_ HMODULE hModule);</pre>

XSI では、カーネル共有ライブラリとデザイン共有ライブラリの 2 つの共有ライブラリから関数を呼び出す必要があります。カーネル共有ライブラリは Vivado シミュレータに含まれており、`librdsimulator_kernel.so` (Linux) または `librdsimulator_kernel.dll` (Windows) という名前です。このライブラリは次のディレクトリにあります。

```
<Vivado Installation Root>/lib/<platform>
```

<platform> は、`lnx64.o` または `win64.o` です。プログラムの実行中は、ライブラリパスにこのディレクトリを必ず含めてください。Linux では `LD_LIBRARY_PATH` 環境変数に、Windows では `PATH` 環境変数に含めます。

デザイン共有ライブラリは、Vivado シミュレータにより HDL デザインのコンパイル中に作成され ([デザイン共有ライブラリの準備](#) を参照)、`xsimk.so` (Linux) または `xsimk.dll` (Windows) と呼ばれ、通常は次のディレクトリにあります。

```
<HDL design directory>/xsim.dir/<snapshot name>
```

<HDL design directory> はデザイン共有ライブラリが作成されたディレクトリ、<snapshot name> はライブラリ作成中に指定するスナップショット名です。

C/C++ プログラムは、デザイン共有ライブラリにある `xsi_open()` という XSI 関数と、カーネル共有ライブラリのその他すべての XSI 関数を呼び出します。

Vivado シミュレータに含まれている XSI コード例では、XSI 関数が `Xsi::Loader` と呼ばれる C/C++ クラスにまとめられています。このクラスは、2 つの共有ライブラリの名前を受け入れて、必要なダイナミックリンク手順を内部で実行し、すべての XSI 関数をクラスのメンバー関数として使用できるようにします。このように XSI 関数をラップすると、ダイナミックリンクの OS 関数を直接呼び出す必要がなくなります。このクラスのソースコードは、Vivado インストールディレクトリの次のディレクトリにあり、ユーザーのプログラムにコピーできます。

```
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.h
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.cpp
```

`Xsi::Loader` を使用するには、次の例に示すように、2 つの共有ライブラリの名前を渡してインスタンス化します。

```
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so",
"librdi_simulator_kernel.so");
```

テストベンチ コードの記述

XSI を使用する C/C++ テストベンチでは、通常次の手順を使用します。

1. デザインを開きます。
2. 各最上位ポートの ID をフェッチします。
3. シミュレーションが終了するまで、次の手順を繰り返します。
 - a. 最上位入力ポートに値を設定します。
 - b. シミュレーションを特定の時間実行します。
 - c. 最上位出力ポートの値をフェッチします。
4. デザインを閉じます。

次の表に、各手順で使用する XSI 関数と、等価の `Xsi::Loader` メンバー関数をリストします。各 XSI 関数の使用方は、[XSI 関数参照](#) を参照してください。

表 60: `Xsi::Loader` メンバー関数

機能	XSI 関数	<code>Xsi::Loader</code> メンバー関数
デザインを開く	<code>xsi_open</code>	<code>open</code>
ポート ID をフェッチ	<code>xsi_get_port_number</code>	<code>get_port_number</code>
入力ポート値を設定	<code>xsi_put_value</code>	<code>put_value</code>
シミュレーションを実行	<code>xsi_run</code>	<code>run</code>
出力ポート値をフェッチ	<code>xsi_get_value</code>	<code>get_value</code>
デザインを閉じる	<code>xsi_close</code>	<code>close</code>

Vivado シミュレータで XSI を使用する C++ プログラムのサンプルは、次のディレクトリにあります。

```
<Vivado Installation Root>/examples/xsim/<HDL language>/xsi
```

C/C++ プログラムのコンパイル

XSI のサンプル プログラムをガイドラインとして使用できます。各例で、サンプル コードをコンパイルして実行するためのスクリプトが 1 または 2 つ提供されています。プログラムのコンパイルについては、コンパイラの資料を参照してください。Linux の場合、コンパイルおよび実行は 2 段階プロセスになります。

1. C シェルで、set_env.csh を実行します。
2. run.csh を起動します。

Windows の場合、run.bat というバッチ ファイルを実行します。

スクリプトに関して、次の点に注意してください。

1. コンパイル行で -I を使用して xsi.h というインクルード ファイルを含むディレクトリを含めるよう指定します。
2. C++ プログラムのコンパイル中は、デザイン共有ライブラリまたはカーネル共有ライブラリは指定しません。

XSI インクルード ファイルは次のディレクトリにあります。

```
<Vivado Installation Root>/data/xsim/include/xsi.h
```

デザイン共有ライブラリの準備

XSI ベースの C/C++ プログラムを作成する最後の手順では、HDL デザインをコンパイルし、それを Vivado シミュレータと共にパッケージして、デザイン共有ライブラリにします。HDL デザインのソース コードを変更するたびに、この手順を繰り返します。



注意: C/C++ プログラムの実行中にそのプログラム用のデザイン共有ライブラリを再構築する場合は、この手順を実行する前にプログラムでデザインを閉じてください。

Vivado シミュレータのユーザー インターフェイスで使用する通常のスナップショットの代わりに、HDL デザインに対して xelab を -dll オプションを使用して共有ライブラリを作成するよう指定して実行し、デザイン共有ライブラリを作成します。

次に例を示します。

Linux のコマンド ラインに次のコマンドを入力し、./xsim.dir/design/xsimk.so にデザイン共有ライブラリを作成します。

```
xelab work.top1 work.top2 -dll -s design
```

work.top1 および work.top2 は最上位モジュール名、design はスナップショット名です。

HDL デザインのコンパイルの詳細は、[xelab](#)、[xvhdl](#)、[xvlog xsim コマンド オプション](#)を参照してください。

XSI 関数参照

このセクションでは、各 XSI API 関数を標準形式 (直接 C 呼び出し) および Xsi::Loader メンバー関数形式で示します。標準形式の関数には xsiHandle という引数を指定できますが、メンバー関数には指定できません。xsiHandle には、開いている HDL デザインのステート情報が含まれます。標準形式の xsi_open は xsiHandle を生成します。Xsi::Loader 内には xsiHandle が含まれています。

xsi_close

```
void xsi_close(xsiHandle design_handle);
void Xsi::Loader::close();
```

この関数は HDL デザインを閉じ、デザインに関連付けられているメモリを解放します。シミュレーションを終了するためにこの関数を呼び出します。

xsi_get_error_info

```
const char* xsi_get_error_info(xsiHandle design_handle);
const char* Xsi::Loader::get_error_info();
```

この関数は、最後に発生したエラーの説明文を返します。

xsi_get_port_number

```
XSI_INT32 xsi_get_port_number(xsiHandle design_handle, const char*
port_name);
int Xsi::Loader::get_port_number(const char* port_name);
```

この関数は、HDL デザインの要求された最上位ポートの整数 ID を返します。この ID は、`xsi_get_value` および `xsi_put_value` 呼び出しでポートを指定するのに使用できます。`port_name` はポート名です。Verilog では大文字/小文字が区別され、VHDL では大文字/小文字は区別されません。指定した名前のポートが存在しない場合は、-1 が返されます。

コード例

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
```

xsi_get_status

```
XSI_INT32 xsi_get_status(xsiHandle design_handle);
int Xsi::Loader::get_status();
```

この関数は、シミュレーションのステータスを返します。返されるステータスは次の識別子のいずれかです。

表 61: Xsi シミュレーション ステータス識別子

ステータス コード識別子	説明
<code>xsiNormal</code>	エラーなし。
<code>xsiError</code>	シミュレーションで HDL ランタイム エラーが発生しました。
<code>xsiFatalError</code>	シミュレーションでエラー コンディションが発生し、Vivado シミュレータが続行できません。

コード例

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
if (loader.get_status() == xsiError)
    printf("HDL run-time error encountered.\n");
```

xsi_get_value

```
void xsi_get_value(xsiHandle design_handle, XSI_INT32 port_number, void*
value);
int Xsi::Loader::get_value(int port_number, void* value);
```

この関数は、ポート ID `port_number` で示されるポートの値をフェッチします。値は値が示すメモリ バッファに配置されます。ポートの ID 取得については、[xsi_get_port_number](#) を参照してください。



重要: 関数を呼び出すため、プログラムでバッファに十分なメモリを割り当てておく必要があります。必要なバッファ サイズを判断するには、[Vivado シミュレータの VHDL データ形式](#)および [Vivado シミュレータの Verilog データ形式](#)を参照してください。

コード例

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Buffer for value of port "count"
s_xsi_vlog_logicval count_val = {0X00000000, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
loader.get_value(count, &count_val);
```

xsi_open

```
typedef struct t_xsi_setup_info {
    char* logFileName;
    char* wdbFileName;
} s_xsi_setup_info, *p_xsi_setup_info;
xsiHandle xsi_open(p_xsi_setup_info setup_info);
void Xsi::Loader::open(p_xsi_setup_info setup_info);
bool Xsi::Loader::isopen() const;
```

この関数は、シミュレーション用に HDL デザインを開きます。この関数を使用するには、まず関数に渡す `s_xsi_setup_info` 構造体を初期化する必要があります。シミュレーション ログ ファイルの名前には `logFileName` を使用し、ログをディisableにするには、`NULL` を使用します。波形トレース機能がオンになっている場合 ([xsi_trace_all](#) を参照)、`wdbFileName` は出力 WDB (波形データベース) ファイルの名前になります。`NULL` のデフォルト名には `xsim.wdb` を使用します。波形トレース機能がオフの場合は、Vivado シミュレータにより `wdbFileName` フィールドは無視されます。



ヒント: ザイリンクスでは、XSI API が今後変更されないようにプログラムを保護するには、[xsi_open](#) に示すように、フィールドに挿入する前に `s_xsi_setup_info` 構造体を 0 にすることをお勧めします。

この関数の標準形式は、デザインのプロセス ステート情報を含む C オブジェクト `xsiHandle` を返します。これは、ほかのすべての標準形式 XSI 関数で使用できます。ローダー形式のこの関数は値を返ませんが、`isopen` メンバー関数をクエリすることにより、デザインを開いたかどうかをチェックできます。このメンバー関数は、`open` メンバー関数が呼び出された場合に `true` を返します。

例

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
info.logFileName = NULL;
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
```

xsi_put_value

```
void xsi_put_value(xsiHandle design_handle, XSI_INT32 port_number, void*
value);
void Xsi::Loader::put_value(int port_number, const void* value);
```

この関数は、`value` に格納されている値を、ポート ID `port_number` で指定されているポートに出力します。ポートの ID 取得については、[xsi_get_port_number](#) を参照してください。`value` は、プログラムが割り当てて値を挿入する必要があるメモリ バッファへのポインターです。値のフォーマットについては、[Vivado シミュレータの VHDL データ形式](#)および [Vivado シミュレータの Verilog データ形式](#)を参照してください。



注意: パフォーマンスを上げるため、Vivado シミュレータでは `xsi_put_value` に渡す値のサイズまたは型はチェックされません。`xsi_put_value` にポートのサイズまたは型に一致しない値を渡すと、Vivado シミュレータおよびプログラムで予期しない動作が見られることがあります。

コード例

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Hard-coded Buffer for a 1-bit "1" Verilog 4-state value
const s_xsi_vlog_logicval one_val = {0X00000001, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

xsi_restart

```
void xsi_restart(xsiHandle design_handle);
void Xsi::Loader:: restart();
```

この関数は、シミュレーション時間を 0 にリセットします。

xsi_run

```
void xsi_run(xsiHandle design_handle, XSI_UINT64 time_ticks);
void Xsi::Loader::run(XSI_INT64 step);
```

この関数は、カーネル精度単位で指定された時間、シミュレーションを実行します。カーネル精度ユニットはデザインの HDL ソース ファイルの中で最小の時間単位です。たとえば、デザインにソース ファイルが 2 つあり、1 つは精度が 1 ns で、もう 1 つは 1 ps である場合、カーネル精度単位は小さい方、つまり 1 ps になります。

Verilog のソース ファイルでは、`timescale を使用して時間精度を指定できます。

例:

```
`timescale 1ns/1ps
```

この例では、/ の後の時間単位 (1 ps) が時間精度です。VHDL には `timescale に該当するものではありません。

xelab コマンドに --timescale、--override_timeprecision、--timeprecision_vhdl オプションを使用して、カーネル精度単位をさらに調整できます。これらのコマンド ライン オプションの使用法は、[xelab](#)、[xvhdl](#)、[xvlog xsim コマンド オプション](#)を参照してください。

注記: シミュレーションが指定された時間実行されるまで、xsi_run はブロックされます。プログラムと Vivado シミュレータは、1 スレッドの実行を共有します。

xsi_trace_all

```
void xsi_trace_all(xsiHandle design_handle);
void Xsi::Loader:: trace_all();
```

HDL デザインの信号すべてに対して波形トレース機能をオンにするため、xsi_open の後にこの関数を呼び出します。波形トレースをオンにした状態でシミュレーションを実行すると、Vivado シミュレータで、デザインのすべての信号のすべてのイベントを含む波形データベース (WDB) ファイルが生成されます。WDB ファイルのデフォルト名は xsim.wdb です。別の名前を指定するには、次のコード例に示すように、wdbFileName を呼び出すときに s_xsi_setup_info 構造体の xsi_open フィールドを設定します。

コード例

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
```



```
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
loader.trace_all();
```

シミュレーションが完了したら、Vivado で WDB ファイルを開いて、信号の波形を確認できます。Vivado での WDB ファイルの表示方法は、[前に保存したシミュレーション run を開く](#)を参照してください。



重要: HDL デザインをコンパイルするときは、xelab に `-debug all` または `-debug typical` を指定する必要があります。`-debug` を指定しないと、波形データは Vivado シミュレータで保存されません。

Vivado シミュレータの VHDL データ形式

このセクションでは、VHDL 値の変換方法と、XSI 関数の `ksi_get_value` および `ksi_put_value` で使用するメモリ バッファのフォーマットについて説明します。

IEEE の std_logic 型

VHDL の `std_logic` および `std_ulogic` の 1 ビットは、C/C++ では 1 バイトとして表現されます (char または符号なしの char)。次の表に、`std_logic/std_ulogic` の値およびそれと等価の C/C++ 型を示します。

表 62: `std_logic/std_ulogic` 値およびそれに対応する C/C++ 値

std_logic 値	C/C++ バイト値 (10 進数)
'U'	0
'X'	1
'0'	2
'1'	3
'Z'	4
'W'	5
'L'	6
'H'	7
'_'	8

コード例

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : std_logic;
const char one_val = 3; // C encoding for std_logic '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL の bit 型

VHDL の `bit` の 1 ビットは、C/C++ では 1 バイトとして表現されます。次の表に、`bit` の値およびそれと等価の C/C++ 型を示します。

表 63: bit およびそれと等価の C/C++ 型

bit 値	C/C++ バイト値 (10 進数)
'0'	0
'1'	1

コード例

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : bit;
const char one_val = 1; // C encoding for bit '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL の character 型

VHDL の `character` 値は、C/C++ では 1 バイトとして表現されます。VHDL の `character` 値は C/C++ の `char` リテラルとまったく同じで、その ASCII の数値とも等しくなります。たとえば、VHDL の `character` 値 `m` は、C/C++ の `char` リテラル `m` または 10 進数値 109 と同じです。

コード例

```
// Put a 'T' on signal "myChar," where "myChar" is defined as
// signal myChar : character;
const char tVal = 'T';
int myChar = loader.get_port_number("myChar");
loader.put_value(myChar, &tVal);
```

VHDL の整数型

VHDL の `integer` 値は、C/C++ では `int` として表現されます。

コード例

```
// Put 1234 (decimal) on signal "myInt," where "myInt" is defined as
// signal myInt : integer;
const int intVal = 1234;
int myInt = loader.get_port_number("myInt");
loader.put_value(myInt, &intVal);
```

VHDL の real 型

VHDL の `real` 値は、C/C++ では `double` として表現されます。

コード例

```
// Put 3.14 on signal "myReal," where "myReal" is defined as
// signal myReal : real;
const double doubleVal = 3.14;
int myReal = loader.get_port_number("myReal");
loader.put_value(myReal, &doubleVal);
```

VHDL の配列型

VHDL 配列は、C/C++ では VHDL 配列のエレメント タイプを表す C/C++ 型の配列として表現されます。次の表に、VHDL 配列およびそれと等価の C/C++ 型の例を示します。

表 64: VHDL 配列およびそれと等価の C/C++ 型

VHDL の配列型	C/C++ の配列型
<code>std_logic_vector</code> (<code>std_logic</code> の配列)	<code>char []</code>
<code>bit_vector</code> (<code>bit</code> の配列)	<code>char []</code>
<code>string</code> (<code>character</code> の配列)	<code>char []</code>
<code>integer</code> の配列	<code>int []</code>
<code>real</code> の配列	<code>double []</code>

C/C++ では、VHDL 配列の左インデックスは C/C++ 配列要素 0 にマップされ、右インデックスは C/C++ 要素 `<array size> - 1` にマップされます。

C/C++ の配列インデックス	0	1	2		<code><array size> - 1</code>
VHDL 配列 (left TO right) インデックス	left	left + 1	left + 2		right
VHDL 配列 (left DOWNTO right) インデックス	left	left - 1	left - 2		right

コード例

```
// For the following VHDL definitions
// signal slv : std_logic_vector(7 downto 0);
// signal bv : bit_vector(3 downto 0);
// signal s : string(1 to 11);
// type IntArray is array(natural range <>) of integer;
// signal iv : IntArray(0 to 3);
// do the following assignments
//
// slv <= "11001010";
// bv <= B"1000";
// s <= "Hello world";
// iv <= (33, 44, 55, 66);
const unsigned char slvVal[] = {3, 3, 2, 2, 3, 2, 3, 2}; // 3 = '1', 2 = '0'
loader.put_value(slv, slvVal);
const unsigned char bvVal[] = {1, 0, 0, 0};
loader.put_value(bv, bvVal);
const char sVal[] = "Hello world"; // ends with extra '\0' that XSI ignores
loader.put_value(s, sVal);
const int ivVal[] = {33, 44, 55, 66};
loader.put_value(iv, ivVal);
```

Vivado シミュレータの Verilog データ形式

Verilog ロジック データは、`xsi.h` で定義されている次の構造体を使用した C/C++ でエンコードされています。

```
typedef struct t_xsi_vlog_logicval {
    XSI_UINT32 aVal;
    XSI_UINT32 bVal;
} s_xsi_vlog_logicval, *p_xsi_vlog_logicval;
```

Verilog 値の各 4 ステート ビットは、`aVal` の 1 ビット位置および `bVal` のそれに対応するビット位置を占めます。

表 66: Verilog 値マッピング

Verilog 値	aVal ビット値	bVal ビット値
0	0	0
1	1	0
X	1	1
Z	0	1

2 ステートの SystemVerilog ビット値の場合、`aVal` ビットはビット値を保持し、対応する `bVal` ビットは未使用です。ザイリンクスでは、`xsi_put_value` の 2 ステート値を形成するときは、`bVal` を 0 にすることをお勧めします。

Verilog ベクターは、C/C++ では Verilog ベクターの右インデックスが `aVal/bVal` ビット位置 0 にマップされ、左インデックスが `aVal/bVal` ビット位置 `<vector size> - 1` にマップされます。

aVal/bVal ビット位置	<vector size> ~ 31	<vector size> - 1	<vector size> - 2	...	1	0
インデックス wire [left:right] vec (left > right)	未使用	left	left - 1	...	right + 1	right
インデックス wire [left:right] vec (left < right)	未使用	left	left + 1	...	right - 1	right

次の表に、次の Verilog ベクターと等価の Verilog および C/C++ を示します。

```
wire [7:4] w = 4'bXX01;
```

Verilog ビット インデックス				7	6	5	4
Verilog ビット値				X	X	0	1
C/C++ ビット位置	31	...	4	3	2	1	0
aVal ビット値	未使用	...	未使用	1	1	0	1
bVal ビット値	未使用	...	未使用	1	1	0	0

33 個以上の要素がある Verilog ベクターの C/C++ 表現は `s_xsi_vlog_logicval` の配列で、Verilog ベクターの右から 32 ビットが C/C++ 配列の要素 0 に、次の 32 ビットが C/C++ 配列の要素 1 に、というようにマップされます。次の表に、Verilog ベクターのマップ例を示します。

```
wire [2:69] vec;
```

これは、次の C/C++ 配列にマップされます。

```
s_xsi_vlog_logicval val[3];
```

表 69: Verilog インデックス範囲

Verilog インデックス範囲	C/C++ の配列要素
vec[38:69]	val[0]
vec[6:37]	val[1]
vec[2:5]	val[3]

つまり、vec[2] は val[3] ビット位置 3 にマップされ、vec[69] は val[0] ビット位置 0 にマップされます。

多次元 Verilog 配列は、Verilog 配列が C/C++ へのマップ前に行優先順でフラット化されるように、`s_xsi_vlog_logicval` または `s_xsi_vlog_logicval` 配列のビットにマップされます。

たとえば、次の 2 次元配列があるとして。

```
reg [7:0] mem[0:1];
```

C/C++ にマップする前にベクターにコピーされたように扱われます。

```
reg [15:0] vec;
vec[7:0] = mem[1];
vec[8:15] = mem[0];
```

その他のリソースおよび法的通知

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、[ザイリンクス サポート](#) サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hub View] タブをクリックします。
- ザイリンクス ウェブサイトで[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

このガイドの補足情報は、次の資料を参照してください。

1. 『Vivado Design Suite ユーザー ガイド: リリース ノート、インストール、およびライセンス』 ([UG973](#))
2. 『Vivado Design Suite ユーザー ガイド: システム レベル デザイン入力』 ([UG895](#))
3. 『Vivado Design Suite ユーザー ガイド: IP を使用した設計』 ([UG896](#))

4. 『Vivado Design Suite ユーザー ガイド: Vivado IDE の使用』 (UG893)
5. 『Vivado Design Suite ユーザー ガイド: Tcl スクリプト機能の使用』 (UG894)
6. 『Vivado Design Suite 7 シリーズ FPGA および Zynq-7000 SoC ライブラリ ガイド』 (UG953)
7. 『Vivado Design Suite Tcl コマンド リファレンス ガイド』 (UG835)
8. 『Vivado Design Suite ユーザー ガイド: 消費電力解析および最適化』 (UG907)
9. 『Vivado Design Suite ユーザー ガイド: 制約の使用』 (UG903)
10. 『Vivado Design Suite チュートリアル: ロジック シミュレーション』 (UG937)
11. 『Vivado Design Suite ユーザー ガイド: デザイン フローの概要』 (UG892)
12. 『Vivado Design Suite プロパティ リファレンス ガイド』 (UG912)
13. 『Vivado Design Suite ユーザー ガイド: 合成』 (UG901)
14. 『効率的なテストベンチの記述』 (XAPP199)
15. 『IEEE Standard VHDL Language Reference Manual』 (IEEE-STD-1076-1993)
16. 『IEEE Standard Verilog Hardware Description Language』 (IEEE-STD-1364-2001)
17. 『IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language』 (IEEE-STD-1800-2009)
18. 『Standard Delay Format Specification (SDF)』 (IEEE-STD-1497-2004)
19. Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (IEEE-STD-1735)

サードパーティ シミュレータに関する情報へのリンク

詳細は次の各章を参照してください。

1. Questa Advanced Simulator/ModelSim シミュレータ:
 - <http://www.mentor.com/products/fv/questa/>
 - <http://www.mentor.com/products/fv/modelsim/>
2. Cadence IES シミュレータ: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx
3. Synopsys VCS シミュレータ: <http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx>
4. Active-HDL シミュレータ: <https://www.aldec.com/en/support/resources/documentation/articles/1579>
5. Riviera PRO シミュレータ: <https://www.aldec.com/en/support/resources/documentation/articles/1525>

トレーニング リソース

ザイリンクスでは、この資料に含まれるコンセプトを説明するさまざまなトレーニング コースおよび QuickTake ビデオを提供しています。次のリンクから関連するトレーニング リソースを参照してください。

1. トレーニング コース: Vivado Design Suite を使用した FPGA の設計 1
2. トレーニング コース: Vivado Design Suite を使用した FPGA の設計 2
3. トレーニング コース: Vivado Design Suite を使用した FPGA の設計 3
4. Vivado Design Suite QuickTake ビデオ: Zynq-7000 Verification IP を使用してシミュレーションでデザインを検証/デバッグする方法
5. Vivado Design Suite QuickTake ビデオ: ロジック シミュレーション
6. Vivado Design Suite QuickTake ビデオ チュートリアル

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとし、また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえば当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ 設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとし、セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品責任の制限を規定する適用法令および規則にのみ従うものとし、また、

商標

© Copyright 2012-2020 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリンクス社の商標です。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、および MPCore は、EU およびその他の各国の Arm Limited の商標です。PCI、PCIe、および PCI Express は PCI-SIG の商標であり、ライセンスに基づいて使用されています。すべてのその他の商標は、それぞれの保有者に帰属します。

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、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。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。