

合成/シミュレーション デザイン ガイド

UG626 (v12.3) 2010 年 9 月 21 日

目次

1: このマニュアルについて	9
このマニュアルの概要	9
デザイン例	10
マニュアルの内容	10
その他のリソース	10
表記規則	11
書体	11
オンライン マニュアル	12
2: ハードウェア記述言語 (HDL)	13
FPGA デバイスの設計で HDL を使用する利点	13
大規模なプロジェクトでトップダウン設計が可能	14
デザインフローの初期段階で論理シミュレーションが可能	14
HDL コードをゲートに合成可能	14
初期段階でさまざまなデザイン インプリメンテーションをテスト可能	14
RTL コードを再利用可能	14
HDL を使用した FPGA デバイスの設計	15
HDL を使用した FPGA デバイスの設計	15
VHDL を使用した FPGA デバイスの設計	15
Verilog を使用した FPGA デバイスの設計	15
合成ツールの使用	16
FPGA システム機能を使用したデバイス パフォーマンスの向上	16
デザイン階層	16
スピード要件の指定	17
3: FPGA デザイン フロー	19
デザイン フロー	20
デザイン入力での推奨事項	20
RTL コードの使用	20
デザイン階層の正しい選択	21
Architecture Wizard	21
Architecture Wizard の起動	21
Architecture Wizard のコンポーネント	21
CORE Generator ソフトウェア	23
EDN および NGC ファイル	24
VHO ファイル	24
VEO ファイル	24
V および VHD ラップ ファイル	24
ASY (ASCII シンボル) ファイル	24
デザインフローの初期段階で論理シミュレーションが可能	24
合成および最適化	25
コンパイル実行スクリプトの作成	25
デザインを効果的に合成するためのコード変更	27
コアの読み込み	27
制約の設定	28

ユーザー制約ファイル (UCF) での制約の設定	28
ISE Design Suite での制約の設定	28
デザイン サイズおよびパフォーマンスの評価	29
デバイス使用率およびパフォーマンスの予測	29
実際のデバイスの使用率および配線前のパフォーマンスの確認	29
コーディング スタイルおよびシステム機能の評価	31
デザイン パフォーマンスを向上するためのコード変更	31
FPGA システム機能を使用したリソース使用率の向上	32
合成ツールによるザイリンクス特有の機能の設定	32
配置配線	32
タイミング シミュレーション	33
4: コーディングに関する推奨事項	35
HDL を使用した設計	35
名前、ラベル、一般的なコーディング スタイル	36
一般的なコーディング スタイル	36
ザイリンクス命名規則	36
予約名	37
信号およびインスタンスの命名	37
エンティティ名およびモジュール名とファイル名の一致	38
識別子の命名	38
サブモジュールのインスタンス化	38
行の長さ	39
共通のファイル ヘッダ	40
インデントおよびスペース	40
定数の指定	41
ジェネリックおよびパラメータを使用したダイナミック バスおよび配列幅の指定	42
TRANSLATE_OFF および TRANSLATE_ON	43
5: FPGA フローでのコーディング	45
VHDL および Verilog の制限	46
非同期 FIFO (First-In-First-Out) の使用	46
階層デザインの利点と欠点	47
階層デザインでの合成ツールの使用	48
共有リソースを同じ階層レベルに制限	48
複数インスタンスを一緒にコンパイル	48
関連する組み合わせロジックを同じ階層レベルに制限	48
スピードがクリティカルなパスとクリティカルではないパスを分離	48
レジスタを駆動する組み合わせロジックを同じ階層レベルに制限	49
モジュール サイズを制限	49
出力すべてにレジスタを付ける	49
各モジュールまたはデザイン全体のクロックを 1 個に制限	49
データ型の選択	49
std_logic (IEEE 1164) の使用	50
ポート宣言	50
ポート宣言での配列	51
バッファとして宣言されるポートの低減	51
‘timescale の使用	52
混合言語デザイン	53

if 文と case 文	53
if 文を使用した 4 : 1 マルチプレクサのコード例	54
case 文を使用した 4 : 1 マルチプレクサのコード例	55
process 文および always 文のセンシティビティリスト	56
合成コードでの遅延	57
FPGA デザインのレジスタ	58
IOB レジスタ	60
デュアルデータレート (DDR) レジスタ	60
FPGA デザインのラッチ	62
シフトレジスタのインプリメンテーション	62
シフトレジスタの記述	63
制御信号	66
セット、リセットの使用と合成の最適化	66
ゲーティッドクロックの代わりにクロックイネーブルピンを使用	70
ゲーティッドクロックからクロックイネーブルに変更する例	71
レジスタおよびラッチの初期ステート	72
シフトレジスタの初期ステート	73
RAM の初期ステート	73
マルチプレクサ	74
有限ステートマシン (FSM) コンポーネント	76
FSM の記述スタイル	76
1 つのプロセスを使用した FSM	77
2 つまたは 3 つのプロセスを使用した FSM	80
FSM の認識と最適化	80
その他の FSM の機能	81
メモリのインプリメンテーション	81
ブロック RAM の推論	82
書き込みポートが 2 つある READ_FIRST モードのデュアルポート RAM	90
分散 RAM の推論	93
数値演算	95
演算関数の順序およびグループ化	105
リソース共有	105
合成ツールの命名規則	108
FPGA プリミティブのインスタンス化	108
CORE Generator ソフトウェア モジュールのインスタンス化	109
属性および制約	110
属性	110
合成制約	110
インプリメンテーション制約	110
属性の使用	111
合成制約の使用	111
パイプライン	113
パイプライン処理前	114
パイプライン処理後	114
リタイミング	114

6: デザインのシミュレーション	115
業界標準規格への準拠	116
シミュレーション フロー	116
ザイリンクス シミュレーション フローでサポートされる標準規格	116
サポートされるシミュレータおよび OS	117
ザイリンクス ライブラリ	117
HDL デザイン フローのシミュレーション ポイント	117
HDL デザイン フローのシミュレーション ポイント	118
シミュレーション フロー ライブラリ	119
VHDL の標準遅延フォーマット (SDF) ファイル	119
Verilog の標準遅延フォーマット (SDF) ファイル	119
レジスタトランスファレベル (RTL)	119
合成後 (NGDBuild 前) のゲートレベル シミュレーション	120
NGDBuild 後 (マップ前) のゲートレベル シミュレーション	120
部分的なタイミング (ブロック遅延) を含むマップ後のシミュレーション	121
配置配線後 (ブロック遅延およびネット遅延) のタイミング シミュレーション	121
テストベンチを使用したスティミュラスの指定	122
テストベンチの作成	123
テストベンチでの推奨事項	123
VHDL および Verilog のライブラリとモデル	123
シミュレーション ポイントに必要なライブラリ	124
シミュレーションで使用されるライブラリ	125
ライブラリ ソース ファイルとコンパイル順	125
シミュレーション ライブラリ	129
シミュレーション ランタイムの短縮	133
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	134
JTAG シミュレーション	134
SelectMAP シミュレーション	135
Spartan-3AN インシステム フラッシュ シミュレーション	138
シミュレーションでのブロック RAM 競合チェックのディスエーブル	143
SIM_COLLISION_CHECK の文字列	143
シミュレーションでのグローバル リセットおよびトライステート	143
FPGA デバイスでのグローバル トライステート (GTS) とグローバル セット/リセット (GSR) 信号	144
Verilog でのグローバル セット/リセット (GSR) とグローバル トライステート (GTS)	144
デザイン階層とシミュレーション	145
デザインの使用率およびパフォーマンスの向上	145
デザインのガイドライン	145
階層の維持	146
ザイリンクス ライブラリを使用した RTL シミュレーション	148
デルタ サイクルとレース状態	148
シミュレーションの精度	149
SecureIP モデルの暗号化手法	150
ゲートレベル ネットリストの生成 (NetGen の実行)	150
同期エレメントでの X 伝搬のディスエーブル	150
ASYNC_REG 制約の使用	151
MIN/TYP/MAX シミュレーション	152
最小 (MIN)	152
標準 (TYP)	152
最大 (MAX)	152

正確なタイミング シミュレーション結果	152
絶対最小遅延値を使用したシミュレーション	153
VOLTAGE および TEMPERATURE 制約の使用	154
DCM、DLL、および MMCM に関する注意事項	155
DLL/DCM クロックでスキューが調整されないように見える	155
DCM/DLL における TRACE とシミュレーション モデルの違い	156
LVTTL 以外の入力ドライバ	157
波形ビューアに関する注意事項	157
シミュレーションおよびインプリメンテーションの属性	157
タイミング シミュレーションの理解	157
タイミング シミュレーションの重要性	158
デザインでのグリッチ	158
タイミング問題のデバッグ	159
タイミング違反の原因	160
デバッグのヒント	162
セットアップおよびホールド違反	162
ザイリンクスでサポートされる EDA シミュレーション ツールを使用したシミュレーション	164
7: 設計に関する考慮事項	165
アーキテクチャの理解	165
スライスの構造	166
ハード IP ブロック	166
クロック リソース	167
クロックのインプリメンテーションの評価	168
クロック レポート	168
タイミング要件の定義	170
厳しすぎる制約	170
制約の適用範囲	170
合成に関する推奨事項	171
適切なコーディング手法を使用する	171
ロジックの推論を解析する	171
デザインの完全な情報を入力する	171
最適なツール設定を使用する	171
便利な合成属性	172
その他のタイミング オプション	172
インプリメンテーション オプションの選択	173
パフォーマンス評価モード	173
タイミングドリブン パックと配置オプション	173
物理合成オプション	173
SmartXplorer	174
クリティカル パスの評価	174
ロジック レベル数が多い	174
ロジック レベル数が少ない	175
SmartGuide テクノロジ	175
SmartGuide テクノロジを使用する状況	176
SmartGuide テクノロジ	176
付録 A ModelSim でのザイリンクス デザインのシミュレーション	179
ModelSim でのザイリンクス デザインのシミュレーション	179

ISE Design Suite からのシミュレーションの実行 (VHDL/Verilog).....	179
ModelSim (スタンドアロン) での論理シミュレーション.....	179
ModelSim (スタンドアロン) でのバックアノテートされたシミュレーション	180
ModelSim および Questa を使用した SecureIP のシミュレーション	182
付録 B IES でのザイリンクス デザインのシミュレーション	185
ISE Design Suite からのシミュレーションの実行	185
NC-Verilog でのシミュレーション	185
NC-Verilog でのシミュレーション (方法 1).....	185
NC-Verilog でのシミュレーション (方法 2).....	186
NC-Verilog を使用した SecureIP のシミュレーション	187
NC-VHDL でのシミュレーション	187
NC-VHDL でのビヘイビア シミュレーション	188
NC-VHDL でのタイミング シミュレーション	188
付録 C VCS および VCS MX でのザイリンクス デザインのシミュレーション	189
ISE® Design Suite からの VCS および VCS MX の実行	189
VCS および VCS MX でのザイリンクス デザインのシミュレーション (スタンドアロン)	189
コンパイル時間のオプションを含むライブラリ ソース ファイルの使用	189
共有のコンパイル済みライブラリの使用	190
ユニファイド使用モデルの使用 (3 段階プロセス).....	191
VCS での SDF ファイルの使用	191
VCS を使用した SecureIP のシミュレーション	192
VCS を使用した SecureIP のシミュレーションについて	192
コンパイル時間のオプションを含むライブラリ ソース ファイルの使用	192
タイミング シミュレーションでの SIMPRIM ライブラリの使用	192



Xilinx is disclosing this user guide, manual, release note, and/or specification (the “Documentation”) to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU “AS-IS” WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

© Copyright 2002–2010 Xilinx Inc. All Rights Reserved. XILINX, the Xilinx logo, the Brand Window and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners. The PowerPC name and logo are registered trademarks of IBM Corp., and used under license. All other trademarks are the property of their respective owners.

本資料は英語版 (v.12.3) を翻訳したもので、内容に相違が生じる場合には原文を優先します。
資料によっては英語版の更新に対応していないものがあります。
日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

このマニュアルについて

この章では、『合成/シミュレーション デザイン ガイド』に関する一般的な情報を示します。次のセクションが含まれています。

- ・ [このマニュアルの概要](#)
- ・ [デザイン例](#)
- ・ [マニュアルの内容](#)
- ・ [その他のリソース](#)
- ・ [表記規則](#)

このマニュアルの概要

『合成/シミュレーション デザイン ガイド』では、ハードウェア記述言語 (HDL) を使用した FPGA デバイスの一般的な設計方法について説明します。HDL 初心者向けの設計ヒントに加え、FPGA デバイスを初めて設計する HDL 経験者向けのヒントも含まれています。このマニュアルは、ザイリンクス ソフトウェア ツールすべてに共通する操作に精通していることを前提として説明しています。

このマニュアルには、HDL デザインの設計で重要となる次のようなトピックは含まれていません。

- ・ デザイン環境
- ・ 検証手法
- ・ 合成ツールでの制約の設定
- ・ テストでの注意事項
- ・ システム検証

詳細は、合成ツールのマニュアルを参照してください。

デザイン例

このマニュアルに含まれるデザイン例は、次のように作成されています。

- ・ VHDL および Verilog で記述されています。

ザイリンクスでは、Verilog および VHDL の両方を同等に推奨します。VHDL は、Verilog に比べると難度が高く、詳細な情報が必要となります。
- ・ さまざまな合成ツールでコンパイルされています。
- ・ 次のデバイスをターゲットとしています。
 - Spartan®-3
 - Spartan-3E
 - Spartan-3A
 - Spartan-6
 - Virtex®-4
 - Virtex-5
 - Virtex-6

マニュアルの内容

このマニュアルには、次の章が含まれています。

- ・ 第 1 章「[このマニュアルについて](#)」：このマニュアルの一般的な情報を示します。
- ・ 第 2 章「[ハードウェア記述言語 \(HDL\)](#)」：HDL を使用した FPGA デバイスの設計方法について説明します。
- ・ 第 3 章「[FPGA デザイン フロー](#)」：典型的な FPGA デザイン フローを説明します。
- ・ 第 4 章「[コーディングに関する推奨事項](#)」：効率のよいコードを記述するために役立つ HDL コーディングに関する一般的な情報とデザイン例を示します。
- ・ 第 5 章「[FPGA フローでのコーディング](#)」：FPGA デバイス用のデザインのコーディングに関する情報を示します。
- ・ 第 6 章「[デザインのシミュレーション](#)」：ザイリンクスおよびサードパーティのソフトウェアを使用した基本的な HDL シミュレーション フローについて説明します。
- ・ 第 7 章「[設計に関する考慮事項](#)」：アーキテクチャの理解、クロックリソースの理解、タイミング要件の定義、合成の実行、インプリメンテーション オプションの選択、クリティカルパスの評価に役立つ情報を示します。
- ・ 付録 A「[ModelSim でのザイリンクス デザインのシミュレーション](#)」
- ・ 付録 B「[IES でのザイリンクス デザインのシミュレーション](#)」
- ・ 付録 C「[VCS および VCS MX でのザイリンクス デザインのシミュレーション](#)」。

その他のリソース

その他の資料については、次の Web サイトから参照してください。

<http://japan.xilinx.com/literature>

シリコン、ソフトウェア、IP に関する質問および解答をアンサー データベースで検索したり、テクニカル サポートのウェブ ケースを開くには、次のザイリンクス Web サイトにアクセスしてください。

<http://japan.xilinx.com/support>

表記規則

このマニュアルでは、次の表記規則を使用しています。各規則について、例を挙げて説明します。

書体

次の規則は、すべてのマニュアルで使用されています。

表記規則	使用箇所	例
Courier フォント	システムが表示するメッセージ、プロンプト、プログラム ファイルを表示します。	<code>speed grade: - 100</code>
Courier フォント (太字)	構文内で入力するコマンドを示します。	ngdbuild <i>design_name</i>
イタリック フォント	ユーザーが値を入力する必要のある構文内の変数に使用します。	<i>ngdbuild design_name</i>
二重/一重かぎカッコ『』、『』、「」	『』はマニュアル名を、「」はセクション名を示します。	詳細については、『コマンドライン ツール ユーザー ガイド』の「PAR」を参照してください。
角カッコ []	オプションの入力またはパラメータを示しますが、 bus[7:0] のようなバス仕様では必ず使用します。また、GUI 表記にも使用します。	<code>ngdbuild [option_name] design_name</code> [File] → [Open] をクリックします。
中カッコ { }	1 つ以上の項目を選択するためのリストを示します。	<code>lowpwr = {on off}</code>
縦棒	選択するリストの項目を分離します。	<code>lowpwr = {on off}</code>
縦の省略記号	繰り返し項目が省略されていることを示します。	IOB #1: Name = QOUT IOB #2: Name = CLKIN . . .
横の省略記号 . . .	繰り返し項目が省略されていることを示します。	allow block . . . <i>block_name loc1 loc2 ... locn;</i>

オンライン マニュアル

このマニュアルでは、次の規則が使用されています。

表記規則	使用箇所	例
青色の文字	マニュアル内の相互参照、その他の文書へのリンクを示します。	詳細については、「 その他のリソース 」を参照してください。 詳細については、第 1 章「 タイトル フォーマット 」を参照してください。 詳細は、『 Virtex-6 ハンドブック 』の図 25 を参照してください。

ハードウェア記述言語 (HDL)

この章では、Hardware Description Language (HDL) について説明します。次のセクションが含まれています。

- ・ [FPGA デバイスの設計で HDL を使用する利点](#)
- ・ [HDL を使用した FPGA デバイスの設計](#)

HDL は、システムおよび回路デザインのビヘイビアおよび構造を記述するのに使用されます。FPGA アーキテクチャを理解すると、FPGA のシステム機能を効率的に使用する HDL コードを作成できるようになります。次に、HDL を使用した FPGA 設計の理解を深めるための推奨事項を示します。

- ・ ザイリンクスおよび合成ツールのベンダーが提供するトレーニング クラスを受講する。
- ・ このマニュアルに含まれる HDL デザイン例を参照する。
- ・ [ザイリンクス サポート Web ページ](#)からデザイン例をダウンロードする。
- ・ ザイリンクスが提供する次のリソースを利用する。
 - 資料
 - チュートリアル
 - サービス パック
 - ホットライン
 - アンサー データベース

詳細は、[「その他のリソース」](#)を参照してください。

FPGA デバイスの設計で HDL を使用する利点

高集積の FPGA を HDL を使用して設計すると、次のような利点があります。

- ・ [大規模なプロジェクトでトップダウン設計が可能](#)
- ・ [デザインフローの初期段階で論理シミュレーションが可能](#)
- ・ [HDL コードをゲートに合成可能](#)
- ・ [初期段階でさまざまなデザイン インプリメンテーションをテスト可能](#)
- ・ [RTL コードを再利用可能](#)

大規模なプロジェクトでトップダウン設計が可能

HDL は、複雑なデザインを作成する場合に使用されます。システム デザインを使用したトップダウン手法の設計は、多くの設計者が同時に作業する必要がある大規模な HDL プロジェクトで有益です。全体的な設計プランを決定した後、各設計者がコードの別々のセクションを個別に作業できます。

デザインフローの初期段階で論理シミュレーションが可能

HDL 記述をシミュレーションすることで、デザイン フローの初期段階でデザインの論理を検証できます。デザインを RTL (レジスタトランスファレベル) またはゲートレベルにインプリメントする前にデザインをテストすることで、設計プロセスの初期段階で必要な変更を加えることができます。

HDL コードをゲートに合成可能

ハードウェア記述を FPGA デバイスのインプリメンテーションに合成すると、次のような利点があります。

- ・ FPGA デバイスの基本エレメントからデザインを定義する代わりに、上位レベルでデザインを設計できるため、設計時間を短縮できる。
- ・ ハードウェア記述を回路図デザインに手動で変換する際に発生する可能性があるエラーを削減できる。
- ・ オリジナルの HDL コードに対して最適化する際に、ステート マシンのエンコード方式や I/O の自動挿入などの合成ツールの自動処理機能を適用することで、効率を向上させることができる。

初期段階でさまざまなデザイン インプリメンテーションをテスト可能

HDL を使用すると、デザイン フローの初期段階でデザインのさまざまなインプリメンテーションをテストできます。合成ツールを使用して、論理合成およびゲートへの最適化を実行します。

ザイリンクス FPGA を使用すると、デザインをコンピュータ上でインプリメントできます。合成に要する時間は短いため、RTL (レジスタトランスファレベル) でさまざまなアーキテクチャを試すことができます。ザイリンクスの FPGA は、さまざまなデザイン インプリメンテーションをテストするために再プログラム可能です。

RTL コードを再利用可能

RTL コードを少し変更するだけで、別の FPGA デバイスに移行できます。

HDL を使用した FPGA デバイスの設計

このセクションでは、HDL を使用した FPGA デバイスの設計について説明します。次の内容が含まれます。

- ・ [HDL を使用した FPGA デバイスの設計](#)
- ・ [VHDL を使用した FPGA デバイスの設計](#)
- ・ [Verilog を使用した FPGA デバイスの設計](#)
- ・ [合成ツールの使用](#)
- ・ [FPGA システム機能を使用したデバイス パフォーマンスの向上](#)
- ・ [デザイン階層](#)
- ・ [スピード要件の指定](#)

HDL を使用した FPGA デバイスの設計

回路図デザイン入力に精通している場合、ブロック図、ステート マシン、フロー図、真理表といったグラフィカルな概念からデザイン コンポーネントの抽象的表現へ移行する必要があるため、HDL デザインの作成が最初のうちは困難だと思われるかもしれません。HDL でコードを記述する際は、全体的な設計プランを見失わないようにすることが重要です。

HDL を効果的に使用するために、次を理解する必要があります。

- ・ 言語の構文
- ・ 合成ツールおよびシミュレータ
- ・ 使用デバイスのアーキテクチャ
- ・ インプリメンテーション ツール

VHDL を使用した FPGA デバイスの設計

VHDL は、IC (集積回路) 設計用のハードウェア記述言語です。VHDL は合成の入力として開発された言語ではないため、構文の多くは合成ツールでサポートされていません。ただし、VHDL は抽象性が高いため、合成されないシステム レベルのコンポーネントおよびテストベンチを簡単に記述できます。また、合成ツールにより VHDL 言語の異なるサブセットが使用されます。

このマニュアルの例は、一般的に使用される FPGA 合成ツールで正しく処理されます。このマニュアルのこの後のセクションで示されているコーディング手法を使用すると、合成可能な HDL 記述を作成できます。

Verilog を使用した FPGA デバイスの設計

Verilog は、次の理由で合成デザインによく使用されます。

- ・ 従来の VHDL に比べ簡潔である。
- ・ IEEE-STD-1364-95 および IEEE-STD-1364-2001 として規格化されている。

Verilog は合成の入力として開発された言語ではないため、構文の多くは合成ツールでサポートされていません。このマニュアルの Verilog コードの例は、現在よく使用されている FPGA 合成ツールでテストおよび合成されています。このマニュアルのこの後のセクションで示されているコーディング手法を使用すると、合成可能な HDL 記述を作成できます。

SystemVerilog は、合成およびシミュレーションの両方で新たな標準規格となってきました。この規格が将来的に一般的なデザイン ツールで広く採用され、サポートされるかどうかは、現在のところ不明です。

この新しい規格を使用しない場合でも、次を実行することをお勧めします。

- ・ この新しい規格が普及したときに現在使用している Verilog コードをそのまま移行できるように、この規格を評価しておく。
- ・ この規格で定められた新しいキーワードを確認する。
- ・ 現在の Verilog コードで新しいキーワードを使用しないようにする。

合成ツールの使用

ほとんどの合成ツールには、ザイリンクス FPGA デバイス用の特別な最適化アルゴリズムが含まれています。制約およびコンパイル オプションは、使用するデバイスによって異なります。ASIC の合成ツールには FPGA では使用されないコマンドおよび制約があり、これらを使用すると結果に悪影響を与える可能性があります。

FPGA デザインを作成する前に合成ツールでデザインがどのように処理されるかを理解する必要があります。FPGA 合成ベンダーのほとんどは、それらの情報を含むザイリンクス FPGA デバイス用のマニュアルを提供しています。

FPGA システム機能を使用したデバイス パフォーマンスの向上

DCM、乗算器、シフトレジスタ、およびメモリなどの FPGA のシステム機能を使用した HDL コードを作成することで、デバイスのパフォーマンス、エリア使用率、および電気特性を向上させることができます。詳細は、デバイスの [データシート](#) および [ユーザー ガイド](#) を参照してください。

デバイスのサイズ（データ幅およびワード数）およびファンクションの特性を考慮する必要があります。このためには、使用する FPGA のリソースを理解し、アーキテクチャに最適なシステムを選択する必要があります。

デザイン階層

HDL を使用するとデザインを柔軟に記述できますが、すべてのコードが同様に最適化されるわけではありません。ファンクションの記述方法および記述位置によって、最終的な最適化の結果は大きく異なります。

- ・ 使用する手法によって、パフォーマンスが低下したり、デザインのサイズおよび消費電力が不必要に大きくなることがあります。
- ・ 別の手法を使用することで、同じマトリックスでも最適なデザイン結果を達成することが可能になります。

このマニュアルでは、FPGA 設計での手法を示します。

デザイン階層は、FPGA をインプリメントする場合、デザインを段階的に変更する場合の両方で重要です。合成ツールによっては、モジュールをグループ化しない限り、階層の境界が保持されるものもあります。階層の境界が最適化の障害にならないように、モジュールの出力にはレジスタを付ける必要があります。出力にレジスタを付けない場合は、モジュールを合成ツールで許容される最大のサイズにする必要があります。

モジュール 1 つにゲート 5,000 個という規則は現在では無効で、最適化の障害となる可能性があります。合成ベンダーで推奨されるモジュール サイズを確認してください。合成ツールにグループ化コマンドがある場合は、最終手段としてこのコマンドも使用できます。モジュールのサイズおよび内容は、合成結果およびデザインのインプリメンテーションに影響します。このマニュアルでは、デザイン階層を効果的に作成する方法について説明します。

スピード要件の指定

タイミング要件を満たすため、合成ツールおよび配置配線ツールの両方でタイミング制約を設定する必要があります。設計開始時にタイミング要件を指定すると、パフォーマンスに加えてエリア、消費電力、およびツールのランタイムも最適にすることができます。

これにより、次のようなデザインが得られます。

- ・ パフォーマンス要件を満たす
- ・ 小型
- ・ 消費電力が少ない
- ・ 処理時間が短い

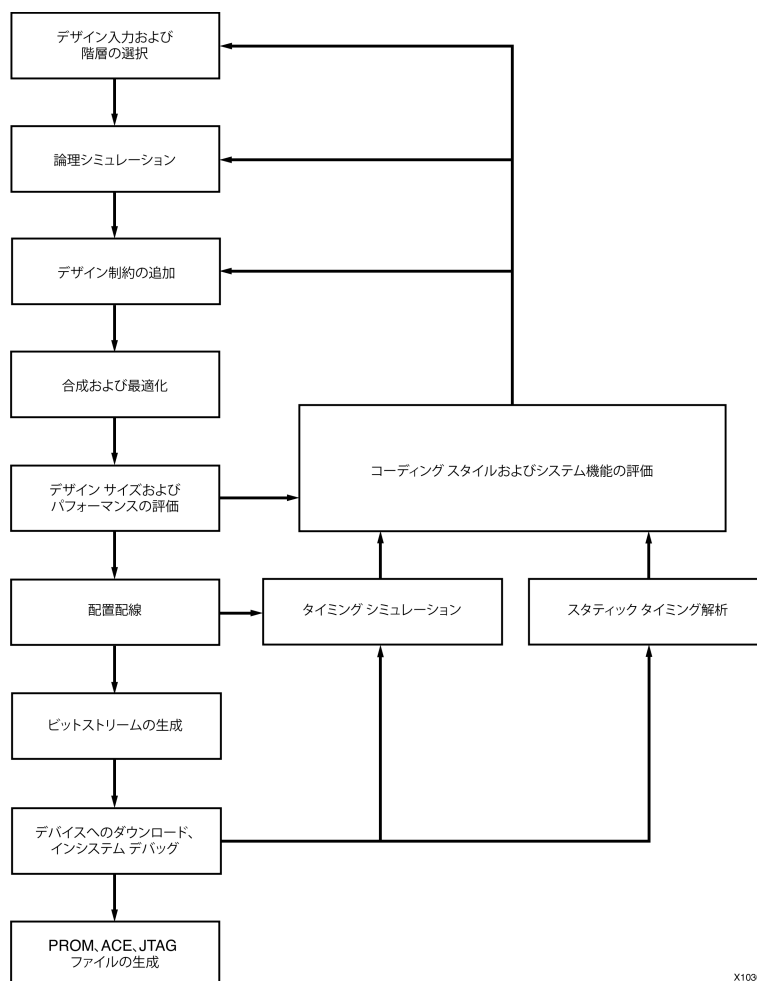
詳細は、「[制約の設定](#)」を参照してください。

FPGA デザイン フロー

この章では、典型的な HDL デザイン フローの各段階を説明します。次のセクションが含まれています。

- ・ デザイン フロー
- ・ デザイン入力での推奨事項
- ・ Architecture Wizard
- ・ CORE Generator™ ソフトウェア
- ・ 論理シミュレーション
- ・ 合成および最適化
- ・ 制約の設定
- ・ デザイン サイズおよびパフォーマンスの評価
- ・ コーディング スタイルおよびシステム機能の評価
- ・ 配置配線
- ・ タイミング シミュレーション

デザイン フロー



X10303

デザイン入力での推奨事項

ザイリンクスでは、次のデザイン入力で次を推奨します。

- ・ [RTL コードの使用](#)
- ・ [デザイン階層の正しい選択](#)

RTL コードの使用

レジスタトランスファレベル (RTL) を使用し、可能な限り特定のコンポーネントをインスタンス化しないようにすることで、次の特徴があるデザインを作成できます。

- ・ コードが解読しやすい
- ・ 合成とシミュレーションで同じコードを使用可能
- ・ シミュレーションが高速で簡単になる
- ・ ほかのデバイス ファミリに移植可能
- ・ 将来のデザインで再利用可能

場合によっては、最適化された CORE Generator モジュールをインスタンス化することが RTL で有益な場合があります。

デザイン階層の正しい選択

デザイン階層を正しく選択すると、次のような利点があります。

- ・ シミュレーションおよび合成の結果が向上
- ・ デバッグが向上
- ・ 複数のエンジニアでデザインの異なる部分を同時に作業可能
- ・ 配線密度を下げ、タイミングを向上することでデザインの配置配線が向上
- ・ 現在および将来のデザインでコードを簡単に再利用可能

Architecture Wizard

ザイリンクス デバイスのアドバンス機能をコンフィギュレーションするには、Architecture Wizard を使用します。Architecture Wizard は、デバイスの特定の機能をコンフィギュレーションする複数のコンポーネントから構成されており、個別のウィザードとして表示されます。詳細は、「[Architecture Wizard のコンポーネント](#)」を参照してください。

Architecture Wizard では、フローに応じて VHDL、Verilog、EDIF (Electronic Data Interchange Format) ファイルが生成されます。生成された HDL ファイルはプリミティブおよびそのプロパティで構成されたモジュールであり、単なるコードの一部ではないため、HDL エディタから参照することが可能です。必須の属性は HDL ファイルに記述されるので、ユーザー制約ファイル (UCF) は出力されません。

Architecture Wizard の起動

Architecture Wizard は、次の方法で起動できます。

- ・ ISE® Design Suite
詳細は、ISE ヘルプの「Architecture Wizard IP の使用」を参照してください。
- ・ CORE Generator ソフトウェア
CORE Generator ソフトウェアの IP のリストから Architecture Wizard IP を選択します。
- ・ コマンド ライン
「**arwz**」と入力します。

Architecture Wizard のコンポーネント

Architecture Wizard は、次のウィザードから構成されています。

- ・ Clocking Wizard
- ・ RocketIO™ ウィザード
- ・ ChipSync Wizard
- ・ XtremeDSP™ Slice Wizard

Clocking Wizard

Clocking Wizard では、次の操作が可能です。

- ・ デジタル クロックの設定
- ・ DCM およびクロック バッファの表示
- ・ DRC チェック

Clocking Wizard の機能 :

- ・ DCM コンポーネントの表示
- ・ 属性の指定
- ・ 対応するコンポーネントおよび信号の生成
- ・ DRC チェックの実行
- ・ 最大 8 個のクロック バッファを表示可能
- ・ フィードバック パスの情報を設定可能
- ・ クロックの周波数生成情報を設定可能
- ・ コンポーネントの属性を表示、変更可能
- ・ コンポーネントの制約を表示、変更可能
- ・ Virtex®-4 デバイスで、1 つまたは 2 つの位相一致クロック分周器 (PMCD) を表示およびコンフィギュレーション可能
- ・ Virtex-5 デバイスで、位相ロック ループ (PLL) を表示およびコンフィギュレーション可能
- ・ XAW ファイルにコンポーネント 1 個を自動的に配置
- ・ VHDL ファイルにコンポーネントの設定を保存
- ・ Verilog ファイルにコンポーネントの設定を保存

RocketIO ウィザード

RocketIO Wizard では、デバイス、バックプレーン、およびサブシステム間のシリアル接続を作成できます。

RocketIO Wizard の機能 :

- ・ RocketIO の種類を指定可能
- ・ チャネル ボンディング オプションを定義可能
- ・ エンコード、CRC、およびクロックを含む一般的なトランスミッタの設定を指定可能
- ・ エンコード、CRC、およびクロックを含む一般的なレシーバの設定を指定可能
- ・ 同期化を指定可能
- ・ 均等化、シグナル インテグリティ (レジスタ、終端モードなど) を指定可能
- ・ コンポーネントの属性を表示、変更可能
- ・ コンポーネントの制約を表示、変更可能
- ・ XAW ファイルにコンポーネント 1 個を自動的に配置
- ・ VHDL または Verilog ファイルにコンポーネントの設定を保存

ChipSync Wizard

ChipSync Wizard は、Virtex-4 および Virtex-5 デバイスでのみ使用可能です。

ChipSync Wizard の機能 :

- ・ 高速のソース同期アプリケーションをインプリメント可能
- ・ メモリ、ネットワーク、またはその他のバス インターフェイスでできるように、I/O ブロックをまとめて 1 つのインターフェイスにコンフィギュレーション可能
- ・ 入力に応じて次の機能を設定する HDL コードを生成可能
 - インターフェイスのデータ幅および I/O 規格、アドレス、およびクロック
 - リファレンス クロックおよび制御ピンなどの追加のピン
 - データ ピンおよびクロック ピンでの調整可能な入力遅延
 - 入力クロックのクロック バッファ (BUFIO)
 - データ幅、クロック イネーブル、およびトライステート信号を制御する ISERDES/OSERDES または IDDR/ODDR ブロック

XtremeDSP Slice Wizard

XtremeDSP Slice Wizard は、Virtex-4 および Virtex-5 デバイスでのみ使用可能です。

XtremeDSP Slice Wizard を使用すると、XtremeDSP Slice Wizard スライスを簡単にインプリメントできます。詳細は、次の資料を参照してください。

- ・ Virtex-4 および Virtex-5 デバイスの[データシート](#)
- ・ [『XtremeDSP for Virtex-4 FPGAs User Guide』](#)
- ・ [『Virtex-5 FPGA XtremeDSP ユーザー ガイド』](#)

CORE Generator ソフトウェア

CORE Generator は、パラメータを指定してザイリンクス FPGA デバイス用に最適化した IP を提供するデザイン ツールです。CORE Generator では、FIFO やメモリから高度なシステムまで、さまざまな既製のファンクションが提供されます。高度なシステムには、次のようなものがあります。

- ・ リード ソロモン デコーダおよびエンコーダ
- ・ FIR フィルタ
- ・ DSP アプリケーション用の FFT
- ・ PCI™ や PCI-X™ などの標準バス インターフェイス
- ・ 接続およびネットワーク インターフェイス (イーサネット、SPI-4.2、PCI EXPRESS® マイクロプロセッサ インターフェイス など)

CORE Generator でコアを生成すると、次のファイルが生成されます。

- ・ EDN および NGC ファイル
- ・ VHO ファイル
- ・ VEO ファイル
- ・ V および VHD ラッパ ファイル
- ・ ASY ファイル

EDN および NGC ファイル

EDIF (Electronic Data Interchange Format) ネットリスト (EDN) ファイルおよび NGC ファイルには、ザイリンクス FPGA にモジュールをインプリメントする際に必要な情報が含まれています。NGC ファイルはバイナリ形式のため、このファイルのリソースおよびタイミング情報をサードパーティの合成ツールに渡す ASCII 形式の NDF ファイルも生成できます。NDF ファイルは合成ツールでのみ読み込まれ、インプリメンテーションでは使用されません。

VHO ファイル

VHDL テンプレート ファイル (VHO) には、CORE Generator モジュールを VHDL デザインにインスタンス化して使用するモデルとして使用可能なコードが含まれています。VHO ファイルを生成すると、VHDL (VHD) ラップ ファイルも生成されます。

VEO ファイル

Verilog テンプレート ファイル (VEO) には、CORE Generator モジュールを Verilog デザインにインスタンス化して使用するモデルとして使用可能なコードが含まれています。VEO ファイルを生成すると、Verilog (V) ラップ ファイルも生成されます。

V および VHD ラップ ファイル

V (Verilog) および VHD (VHDL) ラップ ファイルでは、論理シミュレーションがサポートされています。これらのファイルには、シミュレーション モデルのカスタマイズ データが含まれており、このデータは、パラメータ指定したコアのシミュレーション モデルに渡されます。Verilog デザインの V ラップ ファイルには、合成時に Verilog デザインにコアを統合する際に必要となるポート情報も含まれます。

コアによっては、クロック リソースおよび IOB インスタンスを含む実際のソース コードまたは最上位 HDL ラップ ファイルが作成され、これを独自のクロック供給スキームに適合するよう変更できます。詳細は、コアに関する資料を参照してください。

V および VHD ラップ ファイルは主にシミュレーション用であり、合成できません。

ASY (ASCII シンボル) ファイル

ASY (ASCII シンボル) 情報ファイルを使用すると、ModelSim または ISE® Design Suite ツールの回路図デザインに CORE Generator ソフトウェア モジュールを組み込むことができます。

デザインフローの初期段階で論理シミュレーションが可能

デザインの構文および機能を検証するには、論理 (RTL) シミュレーションを実行します。

デザインのシミュレーション時には、次の推奨事項を考慮してください。

- ・ シミュレーションの個別実行
大型の階層 HDL デザインでは、デザイン全体をテストする前に各モジュールで個別にシミュレーションを実行してください。これにより、コードのデバックが容易になります。
- ・ テストベンチの作成
各モジュールの動作の確認後、デザイン全体が予測どおりに機能するかを検証するテストベンチを作成します。同じテストベンチを最終的なタイミング シミュレーションでも使用して、デザインにワーストケースの遅延がある場合でも予測どおりに機能することを確認します。

ModelSim シミュレータは、ISE と共に使用できます。次のいずれかがインストールされている場合、ISE で ModelSim をシミュレータに指定すると、ModelSim 用のプロセスが [Hierarchy] パネルの [Processes] ペインに表示されます。

- ・ ModelSim Xilinx Edition III
- ・ ModelSim SE、ModelSim PE、または ModelSim DE

これらのシミュレータは、ISE でサードパーティの合成ツールと共に使用できます。

合成および最適化

結果を向上させ、ランタイムを削減するには、次の推奨事項を参照してください。

- ・ [コンパイル実行スクリプトの作成](#)
- ・ [デザインを効果的に合成するためのコード変更](#)
- ・ [コアの読み込み](#)

詳細は、合成ツールのマニュアルを参照してください。

コンパイル実行スクリプトの作成

Tcl スクリプトを使用すると、デザインのコンパイルを容易にすばやく実行でき、コンパイル時間も短縮できます。高度なスクリプトでは、次の操作が可能です。

- ・ さまざまなオプションを使用してコンパイルを複数回実行
- ・ 異なるディレクトリに保存
- ・ 別のコマンド ライン ツールを実行

Tcl スクリプトの実行 (Precision RTL Synthesis)

Precision RTL Synthesis で Tcl スクリプトを実行するには、次のいずれかを実行します。

1. Precision でプロジェクトを設定します。
2. プロジェクトを合成します。
3. Precision RTL Synthesis で次のコマンドを実行し、Tcl スクリプトを保存および実行します。

Precision RTL Synthesis のコマンド

機能	コマンド
Tcl スクリプトの保存	[File] → [Save Command File]
Tcl スクリプトの実行	[File] → [Run Script]
Tcl スクリプトをコマンドラインから実行	c:\precision -shell -file project.tcl
合成の実行	add_input_file top.vhdl setup_design -manufacturer xilinx -family vertex-ii -part 2v40cs144 -speed 6 compile synthesize

Tcl スクリプトの実行 (Synplify)

Synplify で Tcl スクリプトを実行するには、次のいずれかを実行します。

[File] → [Run TCL Script] をクリックします。

または

コマンド プロンプトで「**Synplify -batch script_file.tcl**」と入力します。Synplify では、次の Tcl コマンドを使用できます。

Synplify のコマンド

機能	コマンド
新規プロジェクトの作成	project -new
デバイス オプションの設定	set_option -technology vertex set_option -part XCV50E set_option -package CS144 set_option -speed_grade -8
ファイル オプションの追加	add_file -constraint watch.sdc add_file -vhdl -lib work macro1.vhd add_file -vhdl -lib work macro2.vhd add_file -vhdl -lib work top_level.vhd
コンパイルおよびマップ オプションの設定	set_option -default_enum_encoding onehot set_option -symbolic_fsm_compiler true set_option -resource_sharing true
シミュレーション オプションの設定	set_option -write_verilog false set_option -write_vhdl false

機能	コマンド
自動配置配線 (ベンダー) オプションの設定	<pre>set_option -write_apr_cnstrnt true set_option -part XCV50E set_option -package CS144 set_option -speed_grade -8</pre>
最終フォーマットおよびファイル オプションの設定	<pre>project -result_format edif project -result_file top_level.edf project -run project -save "watch.prj"</pre>
Exit	<code>exit</code>

Tcl スクリプトの実行 (XST)

XST (Xilinx Synthesis Technology) で使用されるオプションについては、『[XST ユーザー ガイド](#)』を参照してください。

デザインを効果的に合成するためのコード変更

デザインを正しく合成するために、コードを変更する必要がある場合があります。シミュレーションでは効果的であったデザインの構文が、合成では効果的でない可能性があります。合成の構文およびコードは、シミュレーションの構文およびコードと多少異なる場合があります。

コアの読み込み

このセクションで説明する合成ツールでは、タイミングおよびエリア解析の際に CORE Generator で生成された NDF ファイルの情報を読み込むことができます。

デザインを解析する際に IP コアの NDF ファイルを読み込むと、周辺のロジックのタイミングおよびリソースの最適化が向上します。NDF ファイルは、IP コアに関連するロジック エLEMENT の遅延を予測するために使用されます。合成ツールでは、IP コアそのものが最適化されることはなく、合成されたデザイン出力ネットリストに IP コアのネットリストは含まれません。

XST でのコアの読み込み

read_cores オプションを使用して XST を起動します。このオプションはデフォルトでオンになっており、EDIF および NGC ネットリストが読み込まれます。詳細は、次を参照してください。

- ・ [『XST ユーザー ガイド』](#)
- ・ ISE ヘルプ

Synplify Pro でのコアの読み込み

Synplify Pro で EDIF ファイルを読み込むと、ソースフォーマットの 1 つとして処理されるので、EDIF を読み込む際は、プロジェクトに最上位 VHDL または Verilog を指定する必要があります。

Precision RTL Synthesis でのコアの読み込み

Precision RTL Synthesis では、EDIF および NGC ファイルをソース ファイルとしてプロジェクトに追加できます。詳細は、Precision RTL Synthesis ヘルプを参照してください。

制約の設定

制約を設定すると、次のような利点があります。

- ・ タイミングの最適化を制御可能
- ・ 合成ツールおよびインプリメンテーション ツールをより効率的に使用可能
- ・ ランタイムの短縮およびデザイン要件の達成に有効

Precision RTL Synthesis および Synplify 合成ツールには、HDL デザインに制約を適用するための 制約エディタが含まれています。

詳細は、合成ツールのマニュアルを参照してください。

次の制約を追加できます。

- ・ クロック周波数、クロック サイクル、オフセット
- ・ 入力および出力のタイミング
- ・ 信号の保持
- ・ モジュールの制約
- ・ バッファ ポート
- ・ パスのタイミング
- ・ グローバル タイミング

ユーザー制約ファイル (UCF) での制約の設定

合成で定義した制約は、ネットリスト制約ファイル (NCF) または出力 EDIF ファイルでインプリメンテーションに渡すこともできますが、これらの制約をインプリメンテーションに渡すのではなく、ユーザー制約ファイル (UCF) で設定することをお勧めします。UCF ファイルを使用すると、次のことが可能になり、全体的な仕様を細かく制御できます。

- ・ より多くの種類の制約を使用可能
- ・ 正確なタイミング パスの定義
- ・ 信号制約に優先順位を付ける

合成およびインプリメンテーションでの制約の設定に関する推奨事項は、「[設計に関する考慮事項](#)」の章を参照してください。各タイミング制約の詳細と構文例は、『[制約ガイド](#)』を参照してください。

ISE Design Suite での制約の設定

ISE® Design Suite では、次のツールを使用して制約を設定できます。

- ・ Constraints Editor
- ・ PACE (CPLD デバイスのみ)
- ・ PlanAhead™

詳細は、ISE ヘルプを参照してください。

デザイン サイズおよびパフォーマンスの評価

デザインは、次の要件を満たしている必要があります。

- ・ 指定のスピードで動作する。
- ・ 指定のデバイスに収まる。

デザインのコンパイル後に、合成ツールのレポート オプションを使用してデバイスの使用率およびパフォーマンスを予測できます。実際のデバイス使用率は、デザインをマップした後に確認できます。

デザイン フローのこの段階では、次のことを確認する必要があります。

- ・ 選択したデバイスがこの後の変更および追加を組み込むのに十分な大きさである。
- ・ デザインが指定どおりに機能する。

デバイス使用率およびパフォーマンスの予測

合成ツールのエリアおよびタイミング レポート オプションを使用すると、デバイスの使用率およびパフォーマンスを予測できます。コンパイル後に、デバイスの使用率を表示するコマンドを使用して確認します。合成ツールによっては、自動的にレポートが表示されるものもあります。レポート表示のコマンド構文については、合成ツールのマニュアルを参照してください。

合成ツールでは、コードからロジックを作成し FPGA デバイスにデザインをマップするので、これらのレポートは通常正確です。レポートは合成ツールによって異なり、最低限必要な CLB 数を示すレポートもあれば、パックされていない配線可能な CLB 数を示すレポートもあります。正しく比較するには、インプリメンテーション後にマップ レポートと比較してください。

CORE Generator モジュール、EDIF ファイルなど、コンパイル中に合成ツールで認識されないインスタンス化されたコンポーネントは、このレポート ファイルには含まれません。デザインにこのようなコンポーネントが含まれる場合は、デザイン サイズを概算する際にこれらのコンポーネントで使用するロジック エリアも考慮する必要があります。また、デザインの一部分がマップ中に削除され、デザイン サイズが小さくなる可能性もあります。

合成ツールのタイミング レポート コマンドを使用すると、データ パス遅延の概算値を示すレポートを表示できます。

詳細は、合成ツールのマニュアルを参照してください。

タイミング レポートは、セル ライブラリのロジック レベル遅延およびデザインのワイヤ ロード概算モデルに基づいています。このレポートは、目標のタイミングにどれだけ近いかを予測したものであり、実際のタイミングではありません。正確なタイミング レポートは、デザインの配置配線後にのみ作成可能です。

実際のデバイスの使用率および配線前のパフォーマンスの確認

デザインが指定のデバイスに収まるかを確認するには、ザイリンクスの MAP プログラムでデザインをマップする必要があります。生成されるレポート ファイル `design_name.mrp` には、インプリメントされたデバイスの使用率が示されます。このレポート ファイルは、ISE® のデザイン サマリの左上のペインで [Map Report] をクリックすると表示されます。MAP プログラムは、ISE またはコマンド ラインから実行できます。

ISE を使用したデザインのマップ

ISE でデザインをマップするには、次の手順に従います。

1. [Design] パネルの [Processes] ペインにアクセスします。
2. [Implement Design] の横にあるプラス記号 (+) をクリックして展開します。
3. [Map] をダブルクリックします。
4. マップ レポートを表示するには、デザイン サマリの左上のペインで [Map Report] をクリックします。

レポートがない場合は生成されます。レポート名の横に緑のチェック マークが表示されている場合はレポートは最新で、プロセスを実行する必要はありません。

5. レポートが最新ではない場合は、次の手順に従います。

- a. レポート名をクリックします。
- b. [Process] → [Rerun] をクリックして、レポートを更新します。

このプロセスでは、レポートを更新するために必要となるプロセスのみが実行されます。

また、[Process] → [Rerun All] をクリックして全プロセスを再実行することも可能です。この場合、プロセスに最新のものがあっても、そのレポートを作成する段階までのすべてのプロセスが再実行されます。

6. レポート ブラウザでロジック レベルのタイミング レポートを表示します。このレポートには、ロジック レベルおよびベストケースの配線遅延に基づいたデザインのパフォーマンスが示されます。
7. Timing Analyzer を起動して、デザイン パスのより詳細なレポートを作成します (オプション)。
8. ロジック レベルのタイミング レポートと Timing Analyzer またはマップ プログラムで生成したレポートを使用して、目標のパフォーマンスおよび使用率にどこまで近づいているかを評価します。

これらのレポートを使用して、インプリメンテーションの配置配線プロセスへ進むか、デザインまたはインプリメンテーション オプションを変更するかを決定します。配置配線でデザインが正しく実行されるようにするため、配線遅延に多少の余裕が必要です。Timing Analyzer で verbose オプションを使用してブロックごとの遅延を確認してください。マップ済みデザイン (配置配線前) のタイミング レポートには、ブロック遅延および最小配線遅延が示されます。

典型的な Virtex®-4 デバイスまたは Virtex-5 デバイスのデザインでは、ロジック遅延 40%、配線遅延 60% が一般的です。ロジック遅延が大部分を占める場合は、デザインが配置配線後にタイミングを満たす可能性はほとんどありません。

コマンド ラインを使用したデザインのマップ

コマンド ラインで引数を使用せずに **trce** コマンドのみを入力すると、使用可能なオプションが表示されます。

コマンドラインを使用してデザインをマップするには、次の手順に従います。

1. 次のコマンドを実行し、デザインを変換します。

```
ngdbuild -p target_device design_name.edf (または ngc)
```

2. 次のコマンドを実行し、デザインをマップします。

```
map design_name.ngd
```

3. テキストエディタを使用して、マップレポート <design_name>.mrp の「Device Summary」のセクションを表示します。

このセクションには、デバイス使用率の情報が示されています。

4. 次のコマンドを使用して、マップ済みデザインのロジックレベル遅延のタイミング解析を実行します。

```
trce [options] design_name.ncd
```

TRACE レポートでは、次を確認できます。

- ・ デザインが目標のパフォーマンスにどれだけ近づいているかを評価
- ・ インプリメンテーションの配置配線へ進むか、デザインまたはインプリメンテーションオプションを変更するかを決定

配置配線でデザインが正しく実行されるようにするため、配線遅延に多少の余裕が必要です。

コーディング スタイルおよびシステム機能の評価

デザインのパフォーマンスが満たされていない場合、コードを再評価します。コードを変更し、異なるコンパイラ オプションを指定することで、デバイス使用率およびスピードを大幅に向上できます。

デザイン パフォーマンスを向上するためのコード変更

次のようにデザインを変更すると、デザインのパフォーマンスを向上できます。

1. 次の方法を使用して、ロジックレベルを削減する。
 - a. パイプラインおよびリタイミング手法を使用する。
 - b. HDL を記述し直す。
 - c. リソース共有をイネーブル/ディスエーブルにする。
2. ロジックを再構築して階層の境界を定義し直し、コンパイラでデザインのロジックを最適化しやすくする。
3. ロジックを複製してクリティカル ネットのファンアウトを削減し、配線密度を下げる。
4. CORE Generator モジュールを使用することにより、デバイス リソースを利用する。

FPGA システム機能を使用したリソース使用率の向上

コーディングの問題を修正した後、次の FPGA システム機能をデザインで使用して、リソースの使用率およびクリティカル パスのスピードを向上させます。

- ・ クロック イネーブルを使用する。
- ・ 大型で複雑なステート マシンにワンホット エンコーディングを使用する。
- ・ I/O レジスタを使用する (適切な場合)。
- ・ 専用シフト レジスタを使用する。
- ・ Virtex®-4 および Virtex-5 デバイスで、専用 DSP ブロックを使用する。

各デバイス ファミリには、それぞれ特有のシステム機能があります。指定のデバイスで使用可能なシステム機能については、[データシート](#)を参照してください。

合成ツールによるザイリンクス特有の機能の設定

合成ツールのザイリンクス特有の機能を使用すると、次を制御できます。

- ・ 生成されるロジック
- ・ ロジック レベル数
- ・ 使用するアーキテクチャ エlement
- ・ ファンアウト

デザイン パフォーマンスが目標まで数パーセントという場合、配置配線ツール (PAR) のアルゴリズムで合成ツールを使用して効率的にデザイン パフォーマンスを達成できます。ほとんどの合成ツールには、ザイリンクス特定の機能を制御するオプションがあります。

詳細は、合成ツールのマニュアルを参照してください。

配置配線

デザインの配置配線では、短いランタイムで高いパフォーマンスを得ることが全般的な目標です。ただし、この目標を達成できない場合もあります。

- ・ デザイン開発の初期段階ではランタイムがパフォーマンスより重視され、後半ではパフォーマンスがランタイムより重視されます。
- ・ ターゲット デバイスの使用率が高いと配線密度が高くなり、デザインの配線が困難になる場合があります。このような場合、配置配線プログラムでタイミング要件を満たすのに要する時間が長くなる可能性があります。
- ・ デザイン制約が厳しい場合、正確にデザインを配置配線し、指定のタイミングを満たすのに要する時間が長くなる可能性があります。

詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

タイミング シミュレーション

タイミング シミュレーションでは、ワースト ケースの配置配線 (PAR) 遅延を算出した後の回路の動作を確認します。ほとんどの場合、論理シミュレーションで使ったテストベンチを使用してより正確なシミュレーションを実行できます。この 2 つのシミュレーション結果を比較して、デザインが指定したとおりに機能しているかを確認します。ザイリンクス ツールでは、配置配線済みのデザインの VHDL または Verilog のシミュレーション ネットリストが生成され、一般的な HDL シミュレータの多くで動作するライブラリが提供されます。詳細は、「[デザインのシミュレーション](#)」の章を参照してください。

タイミングドリブンの PAR は、ザイリンクス タイミング解析ツール TRACE に基づいています。TRACE は、統合スタティック タイミング解析ツールで、回路への入力ステイムラスには基づいていません。配置配線は、設計プロセスの初期段階で指定したタイミング制約に従って実行されます。TRACE は PAR と連動し、設定したタイミング制約が満たされているかを確認します。

タイミング制約がある場合、TRACE でこれらの制約に基づいたレポートが生成されます。タイミング制約がない場合は、オプションを使用して次を含むタイミング レポートを生成できます。

- ・ すべてのクロックと各クロックに必要な OFFSET の解析結果
- ・ 組み合わせロジックのみを含むパスの解析結果 (遅延順)

TRACE の詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。タイミング解析の詳細は、ISE ヘルプの Timing Analyzer のセクションを参照してください。

コーディングに関する推奨事項

この章では、効率のよいコードを記述するために役立つ HDL コーディングに関する一般的な情報およびデザイン例を示します。FPGA デバイスのコーディングに関する情報は、「[FPGA フローでのコーディング](#)」の章を参照してください。次のセクションが含まれています。

- ・ [HDL を使用した設計](#)
- ・ [名前、ラベル、一般的なコーディング スタイル](#)
- ・ [定数の指定](#)
- ・ [ジェネリックおよびパラメータを使用したダイナミック バスおよび配列幅の指定](#)
- ・ [TRANSLATE_OFF および TRANSLATE_ON](#)

HDL を使用した設計

HDL には、複雑な構文が多数含まれています。HDL のマニュアルに含まれる手法および例は、必ずしも FPGA デザインに適用できるとは限りません。現段階で ASIC の設計に HDL を使用している場合、そのコーディング スタイルを FPGA デザインに適用すると、ロジックレベル数を不必要に増やしてしまう可能性があります。

HDL 合成ツールは、デザインのコーディング スタイルに基づいてロジックをインプリメントします。次の方法を使用すると、HDL コードを効率よく記述する方法を学ぶことができます。

- ・ トレーニング クラスを受講する。
- ・ リファレンスおよび手法ヒントを参照する。
- ・ 合成ガイドラインおよびザイリンクスや合成ツール ベンダーから入手可能なテンプレートを参照する。

デザインのコードを記述する際には、HDL がハードウェアの記述言語であることを念頭に置いてください。ハードウェアの最終的なパフォーマンスとシミュレーションの速度とのバランスを見つける必要があります。

このマニュアルで VHDL または Verilog のすべてを示すことはできませんが、効率のよいコードを記述するのに役立つ情報を提供します。

名前、ラベル、一般的なコーディング スタイル

ザイリンクスでは、次の命名規則および一般的なコーディング スタイルに従うことをお勧めします。

- ・ [一般的なコーディング スタイル](#)
- ・ [ザイリンクス命名規則](#)
- ・ [予約名](#)
- ・ [信号およびインスタンスの命名](#)
- ・ [エンティティ名およびモジュール名とファイル名の一致](#)
- ・ [識別子の命名](#)
- ・ [サブモジュールのインスタンス化](#)
- ・ [行の長さ](#)
- ・ [共通のファイル ヘッダ](#)
- ・ [インデントおよびスペース](#)

一般的なコーディング スタイル

HDL デザインを設計者のチームで作成する場合は、プロジェクト開始時にコーディング スタイルを決めておくことをお勧めします。1 つの確立したコーディング スタイルを使用していれば、チーム メンバーが記述したコードを理解できます。効率の悪いコーディング スタイルは合成およびシミュレーションに悪影響を与える可能性があり、回路が遅くなる原因となります。また、既存の HDL デザインは部分的に新しいデザインに使用されることが多いので、ほかの HDL 設計者が理解できるようコーディング スタイルに従う必要があります。このセクションでは、設計開始前に確立する必要のある推奨コーディング スタイルを示します。

ザイリンクス命名規則

ネット、バス、シンボルに変換される信号、変数、インスタンスに対しては、ザイリンクス命名規則に従ってください。

- ・ VHDL キーワード (**entity**、**architecture**、**signal**、**component** など) は、Verilog コードを記述する場合でも、使用を避ける。
- ・ Verilog キーワード (**module**、**reg**、**wire** など) は、VHDL コードを記述する場合でも、使用を避ける (System Verilog 仕様バージョン 3.1a の Annex B を参照)。
- ・ スラッシュ (/) は、階層の区切り文字として使用されるので、使用しない。
- ・ 数字以外の文字を最低 1 個含める。
- ・ ドル記号 (\$) を使用しない。
- ・ 大なり記号 (<) および小なり記号 (>) は、バス インデックスに使用されることがあるので、名前には使用しない。

予約名

次の FPGA リソース名は予約されているので、ネットおよびコンポーネントの名前には使用しないでください。

- ・ デバイス アーキテクチャ名 (**CLB**、**IOB**、**PAD**、**Slice** など)
- ・ 専用ピン (**CLK**、**INIT** など)
- ・ **GND** および **VCC**
- ・ **BUFG**、**DCM**、**RAMB16** などの UNISIM のプリミティブ名
- ・ **P1**、**A4** などのピン名

言語特有の命名規則に関しては、Verilog または VHDL のリファレンス マニュアルを参照してください。不正文字に対しエスケープ シーケンスを使用しないでください。回路図をデザインにインポートする場合や混合言語の合成や検証を実行する場合は、最も限定的な文字セットを使用してください。

信号およびインスタンスの命名

命名規則に従うと、次の目標を達成できます。

- ・ 最大の行の長さ
- ・ 一貫性のある読みやすいコード
- ・ VHDL および Verilog デザインの混合
- ・ 一貫性のある HDL コード

信号およびインスタンスの一般的な命名規則

ザイリンクスでは、次の一般的な命名規則に従うことをお勧めします。

- ・ 信号およびインスタンス名に予約語を使用しない。
- ・ 信号およびインスタンス名の文字数は、できるだけ 16 文字以内にする。
- ・ 信号およびインスタンスに、その接続または用途を反映する名前を付ける。
- ・ 名前およびキーワードで大文字および小文字を混ぜないようにし、すべて大文字にするかまたは小文字にする。

VHDL および Verilog での大文字/小文字の使用

ザイリンクスでは、次に示す大文字/小文字のガイドラインに従って、VHDL および Verilog に含める信号およびインスタンスに名前を付けることをお勧めします。

小文字	大文字	混合
ライブラリ名	ユーザー ポート	コメント
キーワード	インスタンス名	—
モジュール名	UNISIM コンポーネント名	—
エンティティ名	パラメータ	—
ユーザー コンポーネント名	ジェネリック	—
内部信号	—	—

Verilog では大文字と小文字が区別されるので、同じ名前でも大文字/小文字が異なればモジュール名またはインスタンス名は別の名前として認識されますが、ファイル名の互換性、混合言語のサポート、ほかのツールとの互換性を考慮し、インスタンスに大文字と小文字の異なる同じ名前を使用することは避けてください。

エンティティ名およびモジュール名とファイル名の一致

HDL ファイルに名前を付ける際には、次の事項を考慮してください。

- ・ VHDL または Verilog ソースコードのファイル名が、デザイン ファイルで指定したエンティティ (VHDL) またはモジュール (Verilog) の名前と一致していることを確認してください。名前を一致させると、デザインをコンパイルするスクリプト ファイルの作成が容易になります。
- ・ デザインに複数のエンティティまたはモジュールが含まれている場合は、それぞれを個別のファイルに含めます。VHDL デザインでは、エンティティと関連アーキテクチャを同じファイルにまとめることをお勧めします。
- ・ 合成スクリプト ファイルには、最上位のデザイン ファイル名と同じ名前に拡張子 `.do`、`.scr`、`.script`、または使用している合成ツールのデフォルトの拡張子を付けた名前を使用してください。

識別子の命名

デザイン コードをデバッグおよび再利用しやすくするには、次のガイドラインに従います。

- ・ 簡潔で意味のある名前を使用する。
- ・ ワイヤ、レジスタ、信号、変数、タイプ、およびその他の識別子には、わかりやすい名前を付ける。

例： **CONTROL_reg**

- ・ 読みやすくなるようにアンダースコア (`_`) を使用する。

サブモジュールのインスタンス化

次に、サブモジュールをインスタンス化するための推奨事項を示します。

- ・ 名前による関連付けを使用する。名前による関連付けを使用すると、インスタンス化されたコンポーネントのポートが不正に接続されるのを防ぐことができます。
- ・ 同じ文内で名前による関連付けと位置による関連付けを組み合わせない。
- ・ 1 行につき 1 個のポートをマップするようにする。このようにすると、次のようなコードを記述できます。
 - 解読しやすい
 - コメントを追加可能
 - 変更しやすい

VHDL および Verilog の正しいコード例と不正なコード例

	VHDL	Verilog
不正なコード例	<pre>CLK_1: BUFG port map (I=>CLOCK_IN, CLOCK_OUT);</pre>	<pre>BUFG CLK_1 (.I(CLOCK_IN), CLOCK_OUT);</pre>
正しいコード例	<pre>CLK_1: BUFG port map(I=>CLOCK_IN, O=>CLOCK_OUT);</pre>	<pre>BUFG CLK_1 (.I(CLOCK_IN), .O(CLOCK_OUT));</pre>

サブモジュール インスタンスーションの VHDL コード例

```
-- FDCPE: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and
-- Clock Enable (posedge clk). All families.
-- Xilinx HDL Language Template
```

```
FDCPE_inst : FDCPE
generic map (
  INIT => '0') -- Initial value of register ('0' or '1')
port map (
  Q => Q,    -- Data output
  C => C,    -- Clock input
  CE => CE,  -- Clock enable input
  CLR => CLR, -- Asynchronous clear input
  D => D,    -- Data input
  PRE => PRE -- Asynchronous set input
);

-- End of FDCPE_inst instantiation
```

サブモジュール インスタンスーションの Verilog コード例

```
// FDCPE: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and
// Clock Enable (posedge clk). All families.
// Xilinx HDL Language Template

FDCPE #(
  .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)
) FDCPE_inst (
  .Q(Q),    // Data output
  .C(C),    // Clock input
  .CE(CE),  // Clock enable input
  .CLR(CLR), // Asynchronous clear input
  .D(D),    // Data input
  .PRE(PRE) // Asynchronous set input
);

// End of FDCPE_inst instantiation
```

行の長さ

VHDL または Verilog コードの各行は、80 文字以内にしてください。信号名およびインスタンス名も、この制限を超えないように注意して付けます。

80 文字を超える場合は、継続記号で行を分割し、直前の行と後続の行を揃えます。

ネストされた **if** 文や **case** 文など、コードにネストを多用しないようにしてください。ネストを多用すると、行が長くなり、最適化が困難になります。ネスト文を制限すると、コードの解説および移植が容易になり、また印刷用にフォーマットしやすくなります。

共通のファイル ヘッダ

各ファイルの始めには、コメントを使用した共通のファイル ヘッダを使用してください。共通のファイル ヘッダを使用すると、次が可能になります。

- ・ デザインおよびコードに関する説明を記述できます。
- ・ コードのリビジョンを確認しやすくなります。
- ・ 再利用性が向上します。

ヘッダの内容は、個人および会社の標準に基づいています。

VHDL ファイル ヘッダの例

```
-----  
-- Copyright (c) 1996-2010 Xilinx, Inc.  
-- All Rights Reserved  
--  
-- /____/ \ / \ Company: Xilinx  
-- /____/ \ / \ Design Name: MY_CPU  
-- \ \ \ / \ Filename: my_cpu.vhd  
-- \ \ \ Version: 1.1.1  
-- / / \ Date Last Modified: Fri Sep 24 2009  
-- /____/ / \ Date Created: Tue Sep 21 2009  
-- \ \ \ / \  
-- \____/ \____/ \  
--  
--Device: XC3S1000-5FG676  
--Software Used: ISE 11.1  
--Libraries used: UNISIM  
--Purpose: CPU design  
--Reference:  
-- CPU specification found at: http://www.mycpu.com/docs  
--Revision History:  
-- Rev 1.1.0 - First created, joe_engineer, Tue Sep 21 2009.  
-- Rev 1.1.1 - Ran changed architecture name from CPU_FINAL  
-- john_engineer, Fri Sep 24 2009.
```

インデントおよびスペース

コードでインデントを正しく使用すると、次のような利点があります。

- ・ 1 つのインデントレベルにグループの構文をまとめると、コードが解読、理解しやすくなります。
- ・ コードの記述ミスが減ります。
- ・ デバッグが容易になります。

VHDL コードでのインデントの例

```
entity AND_OR is
  port (
    AND_OUT : out std_logic;
    OR_OUT  : out std_logic;
    I0       : in  std_logic;
    I1       : in  std_logic;
    CLK      : in  std_logic;
    CE       : in  std_logic;
    RST      : in  std_logic);
end AND_OR;
architecture BEHAVIORAL_ARCHITECTURE of AND_OR is
  signal and_int : std_logic;
  signal or_int  : std_logic;
begin
  AND_OUT <= and_int;
  OR_OUT  <= or_int;
  process (CLK)
  begin
    if (CLK'event and CLK='1') then
      if (RST='1') then
        and_int <= '0';
        or_int  <= '0';
      elsif (CE='1') then
        and_int <= I0 and I1;
        or_int  <= I0 or I1;
      end if;
    end if;
  end process;
end AND_OR;
```

Verilog コードでのインデントの例

```
module AND_OR (AND_OUT, OR_OUT, I0, I1, CLK, CE, RST);
  output reg AND_OUT, OR_OUT;
  input  I0, I1;
  input  CLK, CE, RST;
  always @(posedge CLK)
    if (RST) begin
      AND_OUT <= 1'b0;
      OR_OUT  <= 1'b0;
    end else (CE) begin
      AND_OUT <= I0 and I1;
      OR_OUT  <= I0 or I1;
    end
endmodule
```

定数の指定

名前をわかりやすいものにするため、デザインで数値の代わりに定数を使用します。定数を使用すると、デザインの解釈が容易になり、移植性が向上します。

定数を使用すると、コードのファンクションを理解しやすくなります。

- ・ VHDL では、コードの定数に変数を使用しないことをお勧めします。コード内の定数値は定数として定義し、それらの値に名前を付けて使用します。
- ・ Verilog では、パラメータを定数として使用できます。これにより、同じリテラル値がある場合にそれらが同じ意味であるかを簡単に確認できます。

次のコード例では、**OPCODE** 値が定数またはパラメータとして宣言されており、その名前でファンクションが参照されます。これにより、コードが読みやすくなり、変更が容易になります。

定数およびパラメータの使用の VHDL コード例

```
constant ZERO : STD_LOGIC_VECTOR (1 downto 0):=00; constant A_B: STD_LOGIC_VECTOR (1 downto 0):=01;
constant A_B : STD_LOGIC_VECTOR (1 downto 0):=10;
constant ONE : STD_LOGIC_VECTOR (1 downto 0):=11;
process (OPCODE, A, B)
begin
    if (OPCODE = A_B)then OP_OUT <= A and B;
    elsif (OPCODE = A_B) then
        OP_OUT <= A or B;
    elsif (OPCODE = ONE) then
        OP_OUT <= '1';
    else
        OP_OUT <= '0';
    end if;
end process;
```

定数およびパラメータの使用の Verilog コード例

```
//Using parameters for OPCODE functions
parameter ZERO = 2'b00;
parameter A_B = 2'b01;
parameter A_B = 2'b10;
parameter ONE = 2'b11;
always @ (*)
begin
    if (OPCODE == ZERO)
        OP_OUT = 1'b0;
    else if (OPCODE == A_B)
        OP_OUT=A&B;
    else if (OPCODE == A_B)
        OP_OUT = A|B;
    else
        OP_OUT = 1'b1;
end
```

ジェネリックおよびパラメータを使用したダイナミック バスおよび配列幅の指定

VHDL および Verilog デザイン モジュールで変更可能なバス幅を指定するには、次の手順に従います。

- ・ ジェネリック (VHDL) またはパラメータ (Verilog) を定義します。
- ・ ジェネリック (VHDL) またはパラメータ (Verilog) を使用して、ポートまたは信号のバス幅を定義します。

ジェネリックまたはパラメータには、モジュールをインスタンス化することで上書き可能なデフォルト値を含むことができます。これにより、コードが再利用しやすくなり、また解読が容易になります。

VHDL ジェネリックを使用したコード例

```
-- FIFO_WIDTH data width (number of bits)
-- FIFO_DEPTH by number of address bits
-- for the FIFO RAM i.e. 9 -> 2**9 -> 512 words
-- FIFO_RAM_TYPE: BLOCKRAM or DISTRIBUTED_RAM
-- Note: DISTRIBUTED_RAM suggested for FIFO_DEPTH
-- of 5 or less
entity async_fifo is
  generic (FIFO_WIDTH: integer := 16;)
    FIFO_DEPTH: integer := 9; FIFO_RAM_TYPE: string := "BLOCKRAM");
  rd_clk : in std_logic;
  rd_en : in std_logic;
  ainit : in std_logic;
  wr_clk : in std_logic;
  wr_en : in std_logic;
  dout : out std_logic_vector(FIFO_WIDTH-1 downto 0) := (others=> '0');
  empty : out std_logic := '1';
  full : out std_logic := '0';
  almost_empty : out std_logic := '1';
  almost_full : out std_logic := '0');
end async_fifo;
architecture BEHAVIORAL of async_fifo is
  type ram_type is array ((2**FIFO_DEPTH)-1 downto 0) of std_logic_vector (FIFO_WIDTH-1 downto 0);
```

Verilog パラメータを使用したコード例

```
-- FIFO_WIDTH data width(number of bits)
-- FIFO_DEPTH by number of address bits
-- for the FIFO RAM i.e. 9 -> 2**9 -> 512 words
-- FIFO_RAM_TYPE: BLOCKRAM or DISTRIBUTED_RAM
-- Note: DISTRIBUTED_RAM suggested for FIFO_DEPTH
-- of 5 or less
module async_fifo (din, rd_clk, rd_en, ainit, wr_clk,
  wr_en, dout, empty, full, almost_empty, almost_full, wr_ack);
  parameter FIFO_WIDTH = 16;
  parameter FIFO_DEPTH = 9;
  parameter FIFO_RAM_TYPE = "BLOCKRAM";
  input [FIFO_WIDTH-1:0] din;
  input rd_clk;
  input rd_en;
  input ainit;
  input wr_clk;
  input wr_en;
  output reg [FIFO_WIDTH-1:0] dout;
  output empty;
  output full;
  output almost_empty;
  output almost_full;
  output reg wr_ack;
  reg [FIFO_WIDTH-1:0] fifo_ram [(2**FIFO_DEPTH)-1:0];
```

TRANSLATE_OFF および TRANSLATE_ON

合成の指示子 **TRANSLATE_OFF** および **TRANSLATE_ON** は、合成ツールにジェネリックおよびパラメータを渡すために使用されていました。これは、ほとんどの合成ツールでジェネリックおよびパラメータを読み込むことができなかったためです。また、合成ツールではライブラリも認識されなかったため、「library UNISIM」などのライブラリ宣言にもこれらの指示子が使用されていました。

現在ではほとんどの合成ツールでジェネリックおよびパラメータの読み出しが可能になり、また UNISIM ライブラリも認識できるようになったため、これらの指示子を合成ツールで使用する必要はなくなりました。**TRANSLATE_OFF** および **TRANSLATE_ON** 指示子は、合成可能なファイルにシミュレーションのみのコードを組み込む場合に使用できますが、シミュレーションのみの構文は、シミュレーションのみのファイルまたはテストベンチに含めるようにしてください。

TRANSLATE_OFF および **TRANSLATE_ON** の詳細は、『[制約ガイド](#)』を参照してください。

FPGA フローでのコーディング

この章では、FPGA デバイス用のデザインのコーディングに関する情報を示します。HDL に関する一般的な情報は、「[コーディングに関する推奨事項](#)」の章を参照してください。次のセクションが含まれています。

- ・ [VHDL および Verilog の制限](#)
- ・ [非同期 FIFO \(First-In-First-Out\) の使用](#)
- ・ [階層デザインの利点と欠点](#)
- ・ [階層デザインでの合成ツールの使用](#)
- ・ [データ型の選択](#)
- ・ [`timescale の使用](#)
- ・ [混合言語デザイン](#)
- ・ [if 文と case 文](#)
- ・ [process 文および always 文のセンシティビティリスト](#)
- ・ [合成コードでの遅延](#)
- ・ [FPGA デザインのレジスタ](#)
- ・ [IOB レジスタ](#)
- ・ [FPGA デザインのラッチ](#)
- ・ [シフトレジスタのインプリメンテーション](#)
- ・ [シフトレジスタの記述](#)
- ・ [制御信号](#)
- ・ [レジスタおよびラッチの初期ステート](#)
- ・ [シフトレジスタの初期ステート](#)
- ・ [RAM の初期ステート](#)
- ・ [有限ステート マシン \(FSM\) コンポーネント](#)
- ・ [メモリのインプリメンテーション](#)
- ・ [ブロック RAM の推論](#)
- ・ [分散 RAM の推論](#)
- ・ [数値演算](#)
- ・ [合成ツールの命名規則](#)
- ・ [FPGA プリミティブのインスタンス化](#)
- ・ [CORE Generator™ ソフトウェア モジュールのインスタンス化](#)

- ・ 属性および制約
- ・ パイプライン
- ・ リタイミング

VHDL および Verilog の制限

VHDL および Verilog は、合成の入力として設計されたものではありません。そのため、ハードウェア記述およびシミュレーション構文の多くは、合成ツールでサポートされていません。また、合成ツールによって VHDL および Verilog の異なるサブセットが使用されます。VHDL および Verilog のセマンティクスは、デザインのシミュレーション用に明確に定義されています。合成ツールでは、デザインが合成前と合成後で同様にシミュレーションされるよう、これらのセマンティクスに従う必要があります。次のセクションのガイドラインに従って、ザイリンクス デザイン フローに適したコードを作成してください。

非同期 FIFO (First-In-First-Out) の使用

データを 1 つのクロックドメインから別のクロックドメインに転送するため、非同期 FIFO (First-In-First-Out) バッファがよく使用されます。FIFO のステータスを正しく判断し、データを確実に転送するには、ステータス フラグ (**EMPTY** および **FULL** 信号) を監視する必要があります。

これらのフラグは位相および周期に関連性のない 2 つのクロックドメインに基づいているので、フラグのタイミングとステータスが容易に判断できない場合があります。そのため、非同期 FIFO を使用する際は注意が必要です。

ほとんどの非同期 FIFO のインプリメンテーションでは、フラグのアサートおよびディアサートはサイクルに基づいていません。論理シミュレーションまたはタイミング シミュレーションでステータス フラグがあるクロック サイクルで変化したとしても、FPGA デバイスではステータス フラグがその前または後のクロック サイクルで変化している場合があります。このような状況は、シミュレータでのイベントのタイミングと順序が FPGA デバイスでのイベントのタイミングと順序とは異なっている場合に発生します。

FPGA デバイスの最終的なタイミングは、プロセス、電圧、および温度 (PVT) で決定されます。そのため、チップによって、または同じチップでも環境によって、サイクルの違いが発生する可能性があります。回路の設計では、これらの違いを考慮する必要があります。

EMPTY および FULL フラグを直接監視せずに、特定のクロック サイクル中またはクロック サイクル後にデータが有効であると想定すると、問題が発生します。ほとんどの FIFO のインプリメンテーションでは、メモリに容量があっても、EMPTY フラグがアサートされているときの FIFO からの読み出し、FULL がアサートされているときの FIFO への書き込みは無効です。このような読み出しまたは書き込みにより予測されない結果が得られ、重大なデバッグ問題となることがあります。非同期 FIFO インプリメンテーションのシミュレーションで問題が検出されなかった場合でも、ステータス フラグを必ず監視するようにしてください。

ほとんどの非同期 FIFO のインプリメンテーションでは、EMPTY および FULL ステータスフラグが変化するときまたはその付近で読み出しまたは書き込みが実行されると、ステータスフラグはデフォルトでセーフ状態になります。FIFO が実際にはフルでないのに FULL フラグがアサートされたり、FIFO が実際には空でないのに EMPTY フラグがアサートされることがあります。これにより、FIFO が空またはフルになったときにフラグがアサートされないという問題が回避されます。FIFO がフルでないのに FULL フラグがアサートされたり FIFO が空でないのに EMPTY フラグがアサートされるのは、2 つのクロックドメインのタイミング、FIFO が空またはフルに近い状態で読み出しまたは書き込みが実行されるなどの状況によります。これは、すべての状況における FIFO の動作を確実にするため考慮する必要があります。

FIFO 回路を実現するため、多くの場合 CORE Generator™ ソフトウェアが使用されるか、FIFO プリミティブ (**FIFO18** など) がインスタンス化されます。より移行性を高くし、柔軟にカスタマイズ、効率的にインプリメントするため、独自の FIFO ロジックが使用される場合もあります。

さまざまな合成およびシミュレーション指示子を使用することにより、非同期状況をテストする際に非同期 FIFO が予測どおりに動作するようにできます。

- ・ 多くの場合、FIFO フラグ ロジックを設計する際タイミング違反を回避することはできません。タイミング シミュレーションでタイミング違反が発生した場合、シミュレータで不定のステートを示す X 出力が生成されます。そのため、ロジックを既知の非同期ソースで駆動し、違反が発生した場合でも正しく動作するように処理されている場合は、**ASYNC_REG=TRUE** 属性を関連するフラグレジスタに追加することをお勧めします。このようにすると、レジスタで問題なく非同期入力を受信でき、レジスタでタイミング違反が発生しても X は生成されず、以前の値が保持されます。また、レジスタの動作に悪影響を与える可能性のあるレジスタの複製やその他の最適化も回避されます。詳細は、「[同期エレメントでの X 伝搬のディスエーブル](#)」を参照してください。
- ・ 同じメモリ ロケーションに対して読み出しと書き込みが同時に実行されると、メモリの競合が発生することがあります。メモリの競合が発生すると、読み出しデータが破損することがあるので、回避するようにしてください。ロジックまたはデザインで読み出しデータが無視される場合は、メモリの競合は問題にはなりません。この場合は、RAM モデルに **SIM_COLLISION_CHECK** 属性を設定して競合チェックをディスエーブルにすることもできます。詳細は、「[シミュレーションでのブロック RAM 競合チェックのディスエーブル](#)」を参照してください。

階層デザインの利点と欠点

HDL デザインは、フラットなモジュールまたは複数の小型モジュールとして合成できます。どちらの手法にも利点と欠点がありますが、集積度の高い FPGA では階層デザインの方が有利です。

階層デザインには、次のような利点があります。

- ・ 検証/シミュレーションの簡略化および高速化
- ・ 1 つのデザインを複数のエンジニアが同時に作業可能
- ・ デザインのコンパイル時間の短縮
- ・ わかりやすいデザインを作成可能
- ・ デザイン フローを効率的に管理可能

階層デザインには、次のような欠点があります。

- ・ FPGA へのデザインのマップが階層の境界を越えて最適化されず、リソースの使用率が低くなり、デザインのパフォーマンスが低下する可能性がある。ただし、この点に対処すれば、影響を最小限に抑えることができます。
- ・ デザイン ファイルのリビジョン管理が困難になる。
- ・ デザインが冗長になる。

上記の欠点のほとんどは、デザイン階層を選択するときに注意を払うことで回避できます。

階層デザインでの合成ツールの使用

デザインを効果的に分割することで、コンパイル時間を大幅に短縮でき、合成結果を向上できます。デザインを効果的に分割するには、次の点を考慮します。

- ・ 共有リソースを同じ階層レベルに制限
- ・ 複数インスタンスを一緒にコンパイル
- ・ 関連する組み合わせロジックを同じ階層レベルに制限
- ・ スピードがクリティカルなパスとクリティカルではないパスを分離
- ・ レジスタを駆動する組み合わせロジックを同じ階層レベルに制限
- ・ モジュール サイズを制限
- ・ 出力すべてにレジスタを付ける
- ・ 各モジュールまたはデザイン全体のクロックを 1 個に制限

共有リソースを同じ階層レベルに制限

共有可能なリソースは、同じ階層レベルに配置します。これらのリソースが別の階層レベルにあると、合成ツールで共有されることが認識されません。

複数インスタンスを一緒にコンパイル

複数の同じインスタンスを一緒にコンパイルして、ゲート数を減らします。ただし、デザイン スピードを向上させる場合は、クリティカル パスに含まれるモジュールをほかのインスタンスと一緒にコンパイルしないでください。

関連する組み合わせロジックを同じ階層レベルに制限

関連する組み合わせロジックを同じ階層レベルに配置すると、合成ツールでクリティカル パス全体を 1 回の操作で最適化できます。ブール最適化は階層の境界を越えては行われないので、クリティカル パスが複数の階層に分割されると、ロジックの最適化が制限されます。また、組み合わせロジックが同じ階層レベルにない場合、モジュールの制約が困難になります。

スピードがクリティカルなパスとクリティカルではないパスを分離

最適な合成結果を得るには、ファンクションの異なるデザイン モジュールを異なる階層レベルに配置します。最適化アルゴリズムでは、デザイン スピードが最も優先されます。デザイン エリアを効率よく使用するには、デザイン モジュールからタイミング制約を削除します。

レジスタを駆動する組み合わせロジックを同じ階層レベルに制限

使用する CLB 数を減らすには、レジスタを駆動する組み合わせロジックを同じ階層レベルに制限します。

モジュール サイズを制限

モジュールのサイズを 100 ～ 200 個の CLB に制限してください。この値の範囲は、次の要素によって異なります。

- ・ コンピュータのコンフィギュレーション
- ・ デザインをチームで作業しているかどうか
- ・ ターゲット FPGA デバイスの配線リソース

小型のブロックは比較的制御しやすいですが、必ずしも効率の良いデザインになるとは限りません。デザインの最終的なコンパイルは、トップダウンで行うことをお勧めします。

詳細は、合成ツールのマニュアルを参照してください。

出力すべてにレジスタを付ける

各階層ブロックのモジュールの出力を、レジスタで駆動するようにしてください。出力にレジスタを付けると、クロック周期とその前のモジュールの clock-to-setup タイムのみを制約するだけで済むので、デザインの制約が簡単になります。階層の異なるレベルに複数の組み合わせブロックがある場合は、各モジュールの遅延を手動で計算する必要があります。また、デザイン階層の出力にレジスタを付けると、階層の境界を越えたロジックの最適化で発生する可能性がある問題を回避できます。

各モジュールまたはデザイン全体のクロックを 1 個に制限

各モジュールのクロックを 1 個に制限すると、デザイン階層の最上位クロックと各モジュールのクロックとの関係を記述するだけで済みます。

デザイン全体のクロックを 1 個に制限すると、デザイン階層の最上位にクロックを記述するだけで済みます。

階層の境界を越えた最適化および階層デザインのコンパイルについては、合成ツールのマニュアルを参照してください。

データ型の選択

メモ： このセクションは、VHDL のみに適用されます。

このセクションでは、データ型の選択について説明します。次の内容が含まれます。

- ・ [std_logic \(IEEE 1164\) の使用](#)
- ・ [ポート宣言](#)
- ・ [ポート宣言での配列](#)
- ・ [バッファとして宣言されるポートの低減](#)

std_logic (IEEE 1164) の使用

デザインのコードを記述する際には、ハードウェア記述の標準規格である **std_logic** (IEEE 1164) を使用してください。この標準規格は、次の理由から推奨されています。

1. 多数のステート値を使用可能
デジタル回路で使用されるステートのほとんどを表現できる 9 個の値が含まれています。
2. FPGA で使用可能なロジック ステートをすべて指定可能
 - a. ロジック High (**1**) およびロジック Low (**0**) に加え、プルアップ (**H**) またはプルダウン (**L**) を使用するか、出力がハイインピーダンス (**Z**) であるかを指定できます。
 - b. 競合、タイミング違反などに対する不明の値 (**X**)、入力または信号が未接続であるか (**U**) も指定できます。
 - c. 合成およびシミュレーションに対する FPGA のロジック表現がより正確になります。
3. ボードレベルのシミュレーションを簡単に実行可能

たとえば、1 つの回路のポートに整数型を、別の回路のポートに **std_logic** 型を使用してもデザインを合成できます。ただし、ボードレベルのシミュレーションで時間のかかる型変換を実行する必要があります。

ザイリンクスのインプリメンテーションで出力されるバックアノテーション済みのネットリストは、**std_logic** 型です。テストベンチで最上位エンティティの駆動に **std_logic** 型を使用しない場合、タイミング シミュレーションで論理シミュレーションのテストベンチを再利用できません。合成ツールによっては、2 つの最上位エンティティ間に型変換のラップを作成できるものもありますが、ザイリンクスでは推奨しません。

ポート宣言

すべてのエンティティ ポート宣言に **std_logic** 型を使用してください。 **std_logic** 型を使用すると、合成されたネットリストをポートの変換ファンクションを使用せずにデザイン階層に戻すことができます。次に、ポート宣言に **std_logic** 型を使用した VHDL のコード例を示します。

```
Entity alu is
  port (
    A   : in STD_LOGIC_VECTOR(3 downto 0);
    B   : in STD_LOGIC_VECTOR(3 downto 0);
    CLK : in STD_LOGIC;
    C   : out STD_LOGIC_VECTOR(3 downto 0)
  );
end alu;
```

最上位のポートを **std_logic** 以外の型で指定すると、ソフトウェアで生成されるシミュレーション モデル (タイミング シミュレーションなど) がテストベンチに対応しない場合があります。これは、次の原因です。

- ・ 特定のデザイン ポートの型情報は格納できない。
- ・ FPGA ハードウェアのシミュレーションでは、動作を正しく表示するために、ハイインピーダンス (トライステート) や不明な値 X のような **std_logic** の値を指定する必要がある。

次の事項に従うことをお勧めします。

- ・ 配列をポートとして宣言しない。配列をポートとして宣言すると、正しく表現または再生成できません。
- ・ すべての最上位ポート宣言に **stg_logic** および **STD_LOGIC_VECTOR** を使用する。

ポート宣言での配列

VHDL では、ポートを配列型として宣言できますが、ザイリンクスでは次の理由から推奨しません。

- ・ Verilog との互換性がない
- ・ 元の配列宣言を格納または再生成できない
- ・ ソフトウェアのピン名と一致しない

Verilog との互換性がない

Verilog では、ポートを配列型として宣言できないため、言語間での移植性が制限されています。また、混合言語プロジェクトでコードを使用することもできなくなります。

元の配列宣言を格納または再生成できない

ポートを配列型として宣言すると、元の配列宣言を格納および再生成できません。EDIF ネットリスト フォーマットでは、ザイリンクスのデータベース同様、配列の元の型宣言を格納できません。

そのため、NetGen または別のネットリスタでデザインを再生成しようとするときに、元のポート宣言に関する情報が見つからず、出力されるネットリストでポート宣言と信号名に不一致が発生します。ネット名を保持するために **KEEP_HIERARCHY** 属性を使用できるため、この問題は最上位のポート宣言だけでなく下位のポート宣言でも発生します。

ソフトウェアのピン名と一致しない

ポートを配列として宣言すると、ソフトウェアのピン名がソース コードのピン名と異なるものになります。ソフトウェアでは、各 I/O を個別のラベルとして処理する必要があるため、配列として宣言されたポートに対応する名前が予測されるものと異なる場合があります。これが原因で、デザイン制約の渡し、デザイン解析、およびデザイン レポートの理解が困難になります。

バッファとして宣言されるポートの低減

信号が内部で出力ポートとして使用される場合は、バッファを使用しないでください。次の VHDL コードの例を参照してください。

信号 C を内部で外部ポートとして使用した VHDL コード例

次に、信号 **c** を内部で外部ポートとして使用した場合の VHDL コード例を示します。

```
Entity alu is
    port(
        A : in STD_LOGIC_VECTOR(3 downto 0);
        B : in STD_LOGIC_VECTOR(3 downto 0);
        CLK : in STD_LOGIC;
        C : buffer STD_LOGIC_VECTOR(3 downto 0) );
end alu;
architecture BEHAVIORAL of alu is
begin
    process begin
        if (CLK'event and CLK='1') then
            C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
        end if;
    end process;
end BEHAVIORAL;
```

この例では、信号 **c** は内部で出力ポートとして使用されるため、ポート **c** に接続されているデザイン内のすべての階層をバッファとして宣言する必要があります。ただし、バッファタイプは合成中にエラーが発生する原因となる可能性があるため、通常 VHDL デザインでは使用されません。

ダミー信号を挿入してポート C を出力として宣言した VHDL コード例

階層デザインでバッファを削減するには、次の VHDL コード例に示すように、ダミー信号を挿入して、ポート **c** を出力として宣言します。

```
Entity alu is
    port(
        A : in STD_LOGIC_VECTOR(3 downto 0);
        B : in STD_LOGIC_VECTOR(3 downto 0);
        CLK : in STD_LOGIC;
        C : out STD_LOGIC_VECTOR(3 downto 0) );
end alu;
architecture BEHAVIORAL of alu is
    -- dummy signal
    signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
    C <= C_INT;
    process begin
        if (CLK'event and CLK='1') then
            C_INT <= A and B and C_INT;
        end if;
    end process;
end BEHAVIORAL;
```

‘timescale の使用

メモ : このセクションは、Verilog のみに適用されます。

すべての Verilog テストベンチおよびソース ファイルに **‘timescale** 指示子を含めるか、または **‘timescale** 指示子を含む include ファイルを参照する必要があります。**‘timescale** 指示子または参照をソース ファイルの冒頭、モジュールまたはその他のデザイン ユニットの定義の前に含めてください。

``timescale` に 1ps 精度を使用することをお勧めします。DCM などの一部のザイリンクス プリミティブ コンポーネントでは、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適切に実行するため、精度を 1ps にする必要があります。精度を 1ps にしても、これより低い精度を使用した場合と比較して、シミュレーション スピードにほとんどまたはまったく差はありません。

次に、典型的なデフォルトの ``timescale` 指示子を示します。

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

混合言語デザイン

ほとんどの FPGA 合成ツールでは、VHDL および Verilog ファイルの両方を含むプロジェクトを作成できます。VHDL と Verilog の混合は、デザイン ユニット (セル) のインスタンス化のみに制限されています。VHDL デザインへの Verilog モジュールのインスタンス化および Verilog デザインへの VHDL エンティティのインスタンス化が可能です。

VHDL と Verilog の特性は異なるので、混合言語プロジェクト作成の際は次の事項に注意する必要があります。

- ・ 大文字/小文字の区別
- ・ Verilog デザインへの VHDL デザイン ユニットのインスタンス化
- ・ VHDL デザインへの Verilog モジュールのインスタンス化
- ・ 使用可能なデータ型
- ・ ジェネリックおよびパラメータの使用

合成ツールによって、混合言語のサポートは異なります。

詳細は、合成ツールのマニュアルを参照してください。

if 文と case 文

ほとんどの合成ツールでは、**if-elsif** 条件が相互排他的であるかを確認でき、プライオリティ ツリーを構築するためにロジックは追加されません。

次に **if** 文を記述する際の注意点を示します。

- ・ **if** 文の分岐で全出力が定義されていることを確認してください。全出力が定義されていない場合、ラッチが生成されるか、CE 信号で長い論理式が生成されます。**if** 文の前に全出力のデフォルト値を記述すると、このような問題を回避できます。
- ・ 1 つの **if** 文に含める入力信号数を制限すると、ロジック レベル数を削減できます。入力信号数が多数の場合、**if** 文の前で一部の信号をデコードすることが可能か、またはレジスタを介することが可能であるかを確認してください。
- ・ データフローを複雑な **if** 文に含めないようにしてください。複雑な **if-else** 文で生成するのは制御信号のみにしてください。

if 文と case 文の比較

if 文	case 文
プライオリティ エンコード ロジックを生成	バランスのとれたロジックを生成
複数の式を含むことが可能	共通の制御式 1 つに対して評価
スピード クリティカル パスで使用	複雑なデコードで使用

if 文を使用した 4 : 1 マルチプレクサのコード例

次に、**if** 文を使用して 4 : 1 マルチプレクサを記述したコード例を示します。

if 文を使用した 4 : 1 マルチプレクサの VHDL コード例

```
-- IF_EX.VHD
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity if_ex is
    port (
        SEL: in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end if_ex;
architecture BEHAV of if_ex is
begin
    IF_PRO: process (SEL,A,B,C,D)
    begin
        if (SEL="00") then MUX_OUT <= A;
        elsif (SEL="01") then
            MUX_OUT <= B;
        elsif (SEL="10") then
            MUX_OUT <= C;
        elsif (SEL="11") then
            MUX_OUT <= D;
        else
            MUX_OUT <= '0';
        end if;
    end process; --END IF_PRO
end BEHAV;
```

if 文を使用した 4 : 1 マルチプレクサの Verilog コード例

```
////////////////////////////////////////
// IF_EX.V                               //
// Example of a if statement showing a    //
// mux created using priority encoded logic //
// HDL Synthesis Design Guide for FPGA devices //
////////////////////////////////////////
module if_ex (
    input A, B, C, D,
    input [1:0] SEL,
    output reg MUX_OUT);
always @ (*)
begin
    if (SEL == 2'b00)
        MUX_OUT = A;
    else if (SEL == 2'b01)
        MUX_OUT = B;
    else if (SEL == 2'b10)
        MUX_OUT = C;
    else if (SEL == 2'b11)
        MUX_OUT = D;
    else
        MUX_OUT = 0;
    end
endmodule
```

case 文を使用した 4 : 1 マルチプレクサのコード例

次に、同じマルチプレクサを **case** 文を使用して設計したコード例を示します。

これらの例では、**if** 文では合成ツールによってスライスが 2 個必要になる場合がありますが、**case** 文で必要なスライスは 1 個のみです。このような場合は、**case** 文を使用すると使用されるリソースが少なく、遅延パスが短くなります。**case** 文を記述する場合、分岐で全出力が定義されていることを確認してください。

下の Case_Ex のインプリメンテーションの図に、これらのデザインのインプリメンテーションを示します。

case 文を使用した 4:1 マルチプレクサの VHDL コード例

```
-- CASE_EX.VHD
-- May 2009
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity case_ex is
    port (
        SEL : in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end case_ex;
architecture BEHAV of case_ex is
begin
    CASE_PRO: process (SEL,A,B,C,D)
    begin
        case SEL is
            when "00" => MUX_OUT <= A;
            when "01" => MUX_OUT <= B;
            when "10" => MUX_OUT <= C;
            when "11" => MUX_OUT <= D;
            when others => MUX_OUT <= '0';
        end case;
    end process; --End CASE_PRO
end BEHAV;
```

case 文を使用した 4:1 マルチプレクサの Verilog コード例

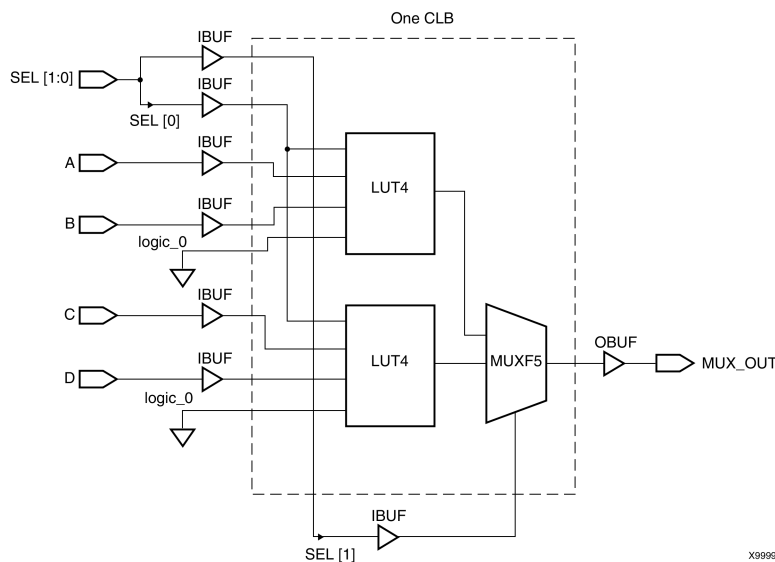
```

////////////////////////////////////
// CASE_EX.V                               //
// Example of a Case statement showing      //
// A mux created using parallel logic       //
// HDL Synthesis Design Guide for FPGA devices //
////////////////////////////////////
module case_ex (
  input A, B, C, D,
  input [1:0] SEL,
  output reg MUX_OUT);

always @ (*)
  begin
    case (SEL)
      2'b00: MUX_OUT = A;
      2'b01: MUX_OUT = B;
      2'b10: MUX_OUT = C;
      2'b11: MUX_OUT = D;
      default: MUX_OUT = 0;
    endcase
  end
endmodule

```

Case_Ex のインプリメンテーション



process 文および always 文のセンシティブティ リスト

process 文 (VHDL) および always 文 (Verilog) のセンシティブティリストは、process ブロック (VHDL) または always ブロック (Verilog) の動作を制御する信号のリストです。センシティブティリストに含まれるいずれかの信号の値が変化すると、process ブロックまたは always ブロックに含まれる文が実行されます。センシティブティリストおよびブロックに含まれる文により、フリップフロップやラッチなどの順次エレメント、組み合わせエレメント、またはこれら 2 つを組み合わせたものを記述できます。

センシティビティリストを使用する場合は、必要な信号がすべて含まれていることを確認してください。含まれていない信号があると、HDL コードから生成されるハードウェアの動作が RTL 記述と異なるものになる可能性があります。この問題は、次の理由から合成ツールで発生します。

- ・ RTL 記述を既存のハードウェアを使用して表すのが不可能な場合がある。
- ・ RTL 記述を HDL コードで正確に表すため、最終的なインプリメンテーションに追加ロジックが必要。

これらの問題を回避するため、合成ツールで HDL コードに明示的に記述されていない信号がセンシティビティリストに含まれていると想定して処理される場合があります、その結果必要なハードウェアが得られたとしても、RTL シミュレーション結果と合成後のシミュレーション結果が異なるものになります。一部の合成ツールでは、センシティビティリストが不完全であることを示す警告メッセージが表示されます。このような警告メッセージが表示された場合は、合成のログ ファイルを確認して、RTL コードを修正してください。

次に、process ブロックおよび always ブロックを使用した簡単な AND ファンクションの記述例を示します。センシティビティリストには必要な信号がすべて含まれています。このブロックから、1 つの LUT が生成されます。

VHDL の process ブロックのコード記述例 1

```
process (a,b)
begin
    c <= a and b;
end process;
```

Verilog の always ブロックのコード記述例 1

```
always @(a or b)
    c <= a & b;
```

次の例は、上記の例のセンシティビティリストから信号 **b** を削除したものです。この場合でも、合成ツールによりセンシティビティリストに信号 **b** が含まれていると想定され、組み合わせロジック (**AND** ファンクション) が生成されます。

VHDL の process ブロックのコード記述例 2

```
process (a)
begin
    c <= a and b;
end process;
```

Verilog の always ブロックのコード記述例 2

```
always @(a)
    c <= a & b;
```

合成コードでの遅延

コードに **Wait for XX ns** 文 (VHDL) または **#XX** (Verilog) 文を使用しないでください。**XX** には、ある条件が実行されるまで待機する時間を ns で指定します。この文はコンポーネントに合成されないため、この文を含むとシミュレーションされるデザインの機能と合成済みのデザインの機能が一致しない場合があります。

Wait for XX ns 文の VHDL コード例

```
wait for XX ns;
```

Wait for XX ns 文の Verilog コード例

```
#XX;
```

VHDL コードの **...after XX ns** 文および Verilog コードの遅延代入は使用しないでください。

after XX ns 文の VHDL コード例

```
(Q <=0 after XX ns)
```

遅延代入を使用した Verilog コード例

```
assign #XX Q=0;
```

XX には、ある条件が実行されるまで待機する時間を ns で指定します。この文は、通常合成ツールで無視されるので、シミュレーションされたデザインの機能と合成済みのデザインの機能が一致しません。

FPGA デザインのレジスタ

ザイリンクス FPGA デバイスには、フリップフロップが多く含まれています。FPGA アーキテクチャでは、次の制御信号を含むフリップフロップがサポートされています。

- ・ クロック イネーブル
- ・ 非同期セット/リセット
- ・ 同期セット/リセット

ザイリンクス FPGA デバイスをターゲットとする合成ツールでは、上記の制御信号を含むレジスタが推論されます。FPGA デザインでの制御信号の使用については、「[制御信号](#)」を参照してください。

デバイスのスタートアップ時のフリップフロップの値は、**0** または **1** に指定できます。この値は、初期化ステートまたは **INIT** と呼ばれます。

立ち上がりエッジで動作するフリップフロップの VHDL コード例

```
process (C)
begin
    if (C'event and C='1') then
        Q <= D;
    end if;
end process;
```

立ち上がりエッジで動作するフリップフロップの Verilog コード例

```
always @(posedge C)
begin
    Q <= D;
end
```

クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップの VHDL コード例

```
process (C)
begin
    if (C'event and C='1') then
        if (CE='1') then
            Q <= D;
        end if;
    end if;
end process;
```

クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップの Verilog コード例

```
always @(posedge C)
begin
    if (CE)
        Q <= D;
end
```

クロックの立ち下がりエッジで動作する非同期リセット付きフリップフロップの VHDL コード例

```
process (C, CLR)
begin
    if (CLR = '1')then
        Q <= '0';
    elsif (C'event and C='0')then
        Q <= D;
    end if;
end process;
```

クロックの立ち下がりエッジで動作する非同期リセット付きフリップフロップの Verilog コード例

```
always @(negedge C or posedge CLR)
begin
    if (CLR)
        Q <= 1'b0;
    else
        Q <= D;
    end
end
```

クロックの立ち上がりエッジで動作する同期セット付きフリップフロップの VHDL コード例

```
process (C)
begin
    if (C'event and C='1') then
        if (S='1') then
            Q <= '1';
        else
            Q <= D;
        end if;
    end if;
end process;
```

クロックの立ち上がりエッジで動作する同期セット付きフリップフロップの Verilog コード例

```
always @(posedge C)
begin
    if (S)
        Q <= 1'b1;
    else
        Q <= D;
end
```

IOB レジスタ

入出力ブロック (IOB) には、通常のフリップフロップまたはデバイスでサポートされている場合はデュアル データレート (DDR) レジスタとしてコンフィギュレーション可能な記憶エレメントが複数含まれています。

IOB 内に配置されるフリップフロップのファンアウトは、すべて 1 です。これは、出力レジスタおよびトリステート イネーブル レジスタでも同様です。たとえば 32 ビットの双方向バスの場合、トリステート イネーブル信号を、ファンアウトが 1 になるように元のデザインで複製する必要があります。

フリップフロップを IOB 内に配置するには、次の方法があります。

- ・ 合成制約を使用する。
- ・ UCF ファイルに **IOB=TRUE** 制約を追加する。
- ・ コマンド ラインで **map -pr** オプションを使用する。

合成ツールにより、フリップフロップが自動的に IOB に配置される場合があります。

詳細は、合成ツールのマニュアルを参照してください。

デュアルデータレート (DDR) レジスタ

DDR レジスタを使用するには、対応する UNISIM プリミティブをインスタンス化する必要があります。合成ツールによっては、HDL コードから DDR を推論できるものもあります。

詳細は、合成ツールのマニュアルを参照してください。

デュアル データ レート IOB レジスタの VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
entity ddr_input is
port (  clk : in std_logic;
        d : in std_logic;
        rst : in std_logic;
        q1 : out std_logic;
        q2 : out std_logic
    );
end ddr_input;

architecture behavioral of ddr_input is
begin
    qlreg : process (clk, rst)
    begin
        if rst = '1' then
            q1 <= '0';
        elsif clk'event and clk='1' then
            q1 <= d;
        end if;
    end process;

    q2reg : process (clk, rst)
    begin
        if rst = '1' then
            q2 <= '0';
        elsif clk'event and clk='0' then
            q2 <= d;
        end if;
    end process;
end behavioral;
```

デュアル データ レート IOB レジスタの Verilog コード例

```
module ddr_input (
    input data_in, clk, rst,
    output data_out);

    reg q1, q2;

    always @ (posedge clk, posedge rst)
    begin
        if (rst)
            q1 <= 1'b0;
        else
            q1 <= data_in;
        end

    always @ (negedge clk, posedge rst)
    begin
        if (rst)
            q2 <= 1'b0;
        else
            q2 <= data_in;
        end
    assign data_out = q1 & q2;
end module
```

FPGA デザインのラッチ

次のような不完全な条件式では、ラッチが推論されます。

- ・ **else** 節のない **if** 文
- ・ 立ち上がりまたは立ち下がりエッジの構文がないレジスタ

else 節のない if 文の VHDL コード例

```
process (G, D)
begin
    if (G='1') then
        Q <= D;
    end if;
end process;
```

else 節のない if 文の Verilog コード例

```
always @(G or D)
begin
    if (G)
        Q = D;
end
```

このような条件式が誤って作成されても、シミュレーションでは正しく機能しているように見える場合がありますが、ラッチを含むパスのタイミング解析は困難であるため、これが FPGA デザインで問題となる可能性があります。ラッチが推論されると、通常合成ツールによりログ ファイルでレポートされます。

FPGA デザインでは、ラッチを使用しないようにしてください。ラッチを使用すると、タイミング解析が困難になります。

合成ツールによっては、デザインにインプリメントするラッチの数を指定できるものもあります。

詳細は、合成ツールのマニュアルを参照してください。

else 節またはクロック エッジのない **if** 文は、すべてレジスタまたはロジック ゲートに変換する必要があります。変換する際は、合成ツールのマニュアルで推奨されているレジスタのコーディング スタイルを使用してください。

シフト レジスタのインプリメンテーション

シフト レジスタには、通常次の制御信号およびデータ信号があります。

- ・ クロック
- ・ シリアル入力
- ・ 非同期セット/リセット
- ・ 同期セット/リセット
- ・ 同期/非同期パラレル ロード
- ・ クロック イネーブル
- ・ シリアル/パラレル出力

シフトレジスタの出力モードは、次のとおりです。

- ・ シリアル

最後のフリップフロップのデータのみを出力

- ・ パラレル

最後以外の 1 つまたは複数のフリップフロップのデータを、シフト モード (レフト、ライトなど) で指定して出力

ザイリンクス FPGA デバイスには専用の SRL16 および SRL32 リソース (LUT に組み込まれている) が含まれており、フリップフロップ リソースを使用せずにシフトレジスタを効率的に生成できます。ただし、これらのエレメントではレフト シフトしかサポートされておらず、使用可能な I/O 信号の数も制限されています。

- ・ クロック
- ・ クロック イネーブル
- ・ シリアル データ入力
- ・ シリアルデータ出力

SRL には、上記の信号に加え、シフトレジスタの長さを決定するアドレス入力 (SRL16 では LUT の **A3**、**A2**、**A1**、**A0** 入力) があります。シフトレジスタの長さは、固定することも変動させることもできます。可変長モードでは、新しいアドレスが 4 ビットの入力アドレスピンに読み込まれると、LUT にアクセスする時間だけ遅れて、Q に新しいビット位置の値が出力されます。

SRL プリミティブでは、同期および非同期セット/リセット制御信号は使用できませんが、一部の合成ツールでは専用 SRL リソースを使用してエリアを接続したインプリメンテーションが可能です。

詳細は、合成ツールのマニュアルを参照してください。

シフト レジスタの記述

VHDL でシフトレジスタを記述するには、複数の方法があります。

- ・ 連結演算子

```
shreg <= shreg (6 downto 0) & SI;
```

- ・ **for loop**

```
for i in 0 to 6 loop  
  shreg(i+1) <= shreg(i);  
end loop;  
shreg(0) <= SI;
```

- ・ あらかじめ定義されたシフト演算子

例： **SLL**、**SRL**

シリアル入力およびシリアル出力のある 8 ビット シフト レフト レジスタの VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;

entity shift_regs_1 is
    port(C, SI : in std_logic;
         SO : out std_logic);
end shift_regs_1;

architecture archi of shift_regs_1 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            tmp <= tmp(6 downto 0) & SI;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

シリアル入力およびシリアル出力のある 8 ビット シフト レフト レジスタの Verilog コード例

```
module v_shift_regs_1 (C, SI, SO);
    input C,SI;
    output SO;
    reg [7:0] tmp;

    always @(posedge C)
    begin
        tmp = {tmp[6:0], SI};
    end

    assign SO = tmp[7];

endmodule
```


シリアル入力およびシリアル出力のある 16 ビット シフト レジスタの VHDL コード例 (可変長モード)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity dynamic_shift_regs_1 is
    port(CLK : in std_logic;
         DATA : in std_logic;
         CE : in std_logic;
         A : in std_logic_vector(3 downto 0);
         Q : out std_logic);
end dynamic_shift_regs_1;

architecture rtl of dynamic_shift_regs_1 is
    constant DEPTH_WIDTH : integer := 16;

    type SRL_ARRAY is array (0 to DEPTH_WIDTH-1) of std_logic;
    -- The type SRL_ARRAY can be array
    -- (0 to DEPTH_WIDTH-1) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- or array (DEPTH_WIDTH-1 downto 0) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- (the subtype is forward (see below))
    signal SRL_SIG : SRL_ARRAY;

begin
    PROC_SRL16 : process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (CE = '1') then
                SRL_SIG <= DATA & SRL_SIG(0 to DEPTH_WIDTH-2);
            end if;
        end if;
    end process;

    Q <= SRL_SIG(conv_integer(A));

end rtl;
```

シリアル入力およびシリアル出力のある 16 ビット シフト レジスタの Verilog コード例 (可変長モード)

```
module v_dynamic_shift_regs_1 (Q,CE,CLK,D,A);
    input CLK, D, CE;
    input [3:0] A;
    output Q;
    reg [15:0] data;

    assign Q = data[A];

    always @(posedge CLK)
    begin
        if (CE == 1'b1)
            data <= {data[14:0], D};
        end
    end

endmodule
```

制御信号

このセクションでは、制御信号について説明します。次の内容が含まれます。

- ・ セット、リセットの使用と合成の最適化
- ・ ゲーティッド クロックの代わりにクロック イネーブル ピンを使用
- ・ ゲーティッド クロックからクロック イネーブルに変更する例

セット、リセットの使用と合成の最適化

ザイリンクス FPGA デバイスには、フリップフロップが多く含まれています。すべてのアーキテクチャでこれらのレジスタおよびラッチに非同期リセットを使用できますが、ザイリンクスでは推奨しません。非同期リセットを使用すると、次の状況が発生する可能性があります。

- ・ タイミング解析が困難になる。
- ・ 合成ツールによる最適化で最適な結果が得られない。

同期システムで非同期リセットを使用することによってタイミングの問題が発生することはよく知られていますが、最適化に影響することはあまり知られていません。

グローバル セット/リセット (GSR)

すべてのザイリンクス FPGA デバイスには、グローバル セットリセット (GSR) という専用の非同期リセットが含まれています。GSR は、デザインに関わらず、FPGA コンフィギュレーションの最後で自動的にアサートされます。ゲートレベルのシミュレーションでもこの GSR 信号を挿入し、初期化されたデザインのシリコンでの動作を正確にシミュレーションできます。非同期リセットをもう 1 個コードに追加しても、この専用機能が複製されるのみで、デバイスの初期化またはシミュレーションの初期化では不要です。

シフト レジスタ LUT (SRL)

ザイリンクスのすべての FPGA デバイスには LUT が含まれており、シフトレジスタ LUT (SRL) と呼ばれる 16 ビット シフト レジスタとしてコンフィギュレーションできます。シフトレジスタを推論するときにリセットを使用すると、シフトレジスタ LUT コンポーネントを推論できません。

SRL は、固定長および可変長のシフトレジスタを構築するのに効率的な構造ですが、同期リセットまたは非同期リセットを使用すると、このコンポーネントを使用できなくなり、レジスタまたはロジックが組み合わされた効率の低い構造になってしまいます。

同期リセットと非同期リセット

同期リセットを使用するかまたは非同期リセットを使用するかによって、大型の IP ブロック内でのレジスタの使用方法が異なります。たとえば、Virtex®-4 および Virtex-5 デバイスの DSP48 にはレジスタが数個含まれており、これらのレジスタを使用すると大幅にエリアを節約でき、また回路全体のパフォーマンスを向上できます。

DSP48 には、同期リセットが 1 個のみ含まれています。DSP48 にパックできるロジック付近のレジスタに同期リセットが推論されると、レジスタもコンポーネント内にパックされ、デザインのサイズが小さくなり、スピードが向上します。非同期リセットが使用されると、レジスタはブロック外に配置される必要があるため、最適な結果が得られません。ブロック RAM レジスタおよび FPGA デバイスに含まれるその他のコンポーネントでも同様の最適化が適用されます。

FPGA デバイスに含まれるフリップフロップは、非同期セット/リセットまたは同期セット/リセットにコンフィギュレーションできます。非同期リセットがコードに記述された場合、フリップフロップを非同期セット/リセットを使用するようにコンフィギュレーションする必要がありますが、これが原因でこのリソースを使用するその他の信号が使用できなくなります。

フリップフロップに同期リセットが記述される場合やリセットがない場合は、セット/リセットを同期動作としてコンフィギュレーションできるので、このリソースをセット/リセットとして使用できます。また、このリソースを使用してデータパスを分割することも可能です。これにより、リソース数が減り、レジスタへのデータパスが短くなります。最適化の内容は、コード記述および合成ツールによって異なります。

非同期リセットのコード例

このセクションでは、非同期リセットのコード例を示します。同じコードを同期リセットで記述したコード例は、「[同期リセットのコード例](#)」を参照してください。

非同期リセットの VHDL コード例

```
process (CLK, RST)
begin
    if (RST = '1') then
        Q <= '0';
    elsif (CLK'event and CLK = '1') then
        Q <= A or (B and C and D and E);
    end if;
end process;
```

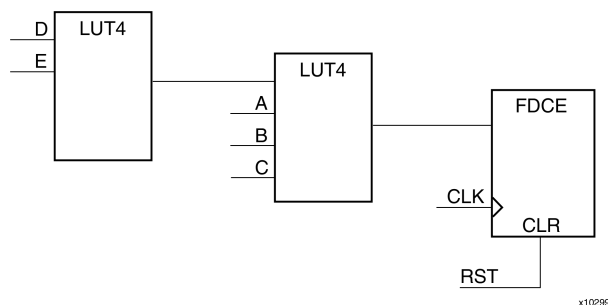
非同期リセットの Verilog コード例

次のコードをインプリメントするには、このロジックを作成するのに信号が 5 個使用されるので、データパスに対し 2 個の LUT が推論されます。

```
always @(posedge CLK or posedge RST)
    if (RST)
        Q <= 1'b0;
    else
        Q <= A | (B & C & D & E);
```

このコード例のインプリメンテーションは、次の図を参照してください。

非同期リセットの Verilog コード例のインプリメンテーション



同期リセットのコード例

「[非同期リセットのコード例](#)」で示したコードを同期リセットを使用して記述し直した例を示します。

同期リセットの VHDL コード例 1

```

process (CLK)
begin
    if (CLK'event and CLK = '1') then
        if (RST = '1') then
            Q <= '0';
        else
            Q <= A or (B and C and D and E);
        end if;
    end if;
end process;

```

同期リセットの Verilog コード例 1

```

always @(posedge CLK)
    if (RST)
        Q <= 1'b0;
    else
        Q <= A | (B & C & D & E);

```

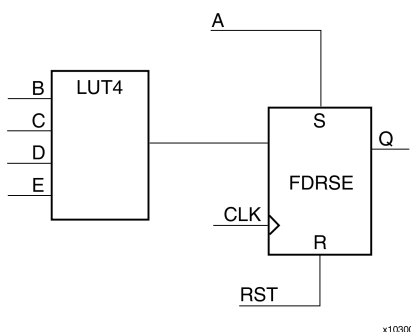
同期リセットを使用すると、合成ツールでのファンクション表現の柔軟性が向上します。このコード例のインプリメンテーションは、次の図を参照してください。

このインプリメンテーションでは、A が High のときに常に Q がロジック 1 になることが合成ツールで認識されます。レジスタは同期動作としてセット/リセットと共にコンフィギュレーションされているため、セットは同期データパスの一部として自由に使用できます。これにより、次のものが削減されます。

- ・ ファンクションをインプリメントするのに必要なロジック数
- ・ D および E 信号のデータパス遅延

ロジックをリセット側にシフトした方が効率的なインプリメンテーションが得られるようコードが記述されている場合は、ロジックがリセット側にシフトされる可能性もあります。

同期リセットの Verilog コード例 1 のインプリメンテーション



同期リセットの VHDL コード例 2

次の例は、「同期リセットの VHDL コード例 1」を変更したものです。

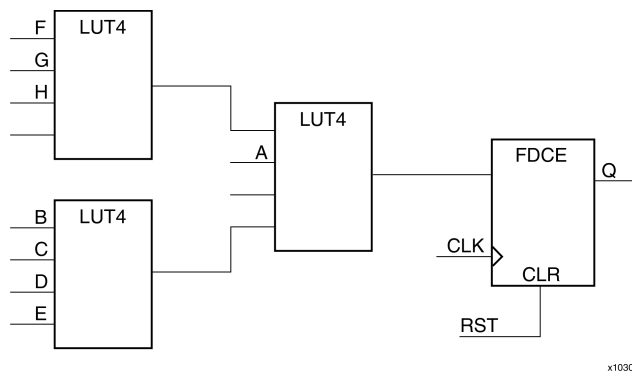
```
process (CLK, RST)
begin
    if (RST = '1') then
        Q <= '0';
    elsif (CLK'event and CLK = '1') then
        Q <= (F or G or H) and (A or (B and C and D and E));
    end if;
end process;
```

同期リセットの Verilog コード例 2

```
always @(posedge CLK or posedge RST)
    if (RST)
        Q <= 1'b0;
    else
        Q <= (F | G | H) & (A | (B & C & D & E));
```

このロジックに使用される信号は 8 個となり、インプリメンテーションには最低 3 個の LUT が必要となります。このコード例のインプリメンテーションは、次の図を参照してください。

同期リセットの Verilog コード例 2 のインプリメンテーション



同期リセットの VHDL コード例 3

次に、同じコードを同期リセットを使用して記述し直した例を示します。

```
process (CLK)
begin
    if (CLK'event and CLK = '1') then
        if (RST = '1') then
            Q <= '0';
        else
            Q <= (F or G or H) and (A or (B and C and D and E));
        end if;
    end if;
end process;
```

同期リセットの Verilog コード例 3

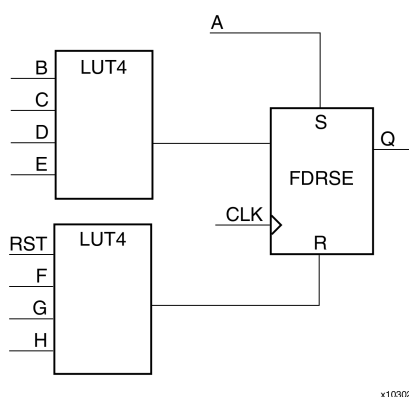
```
always @(posedge CLK)
  if (RST)
    Q <= 1'b0;
  else
    Q <= (F | G | H) & (A | (B & C & D & E));
```

このコード例のインプリメンテーションは、次の図を参照してください。

この例では、同じロジック ファンクションをインプリメントするのに必要な LUT 数が減るだけでなく、このファンクションに使用される各信号のロジック レベルが削減されるため、デザインのスピードが向上する可能性があります。上記の例は単純なものです。同期リセットを使用すると、データ入力のすべての同期データ信号がレジスタを介するようになり、インプリメンテーションが最適にならないことを示しています。

通常、ロジック ファンクションに入力される信号が多いほど、同期セット/リセットを使用して (またはリセットなしで) 効果的にロジック リソースを最小限に抑え、デザインのパフォーマンスを向上できます。

同期リセットの Verilog コード例 3 のインプリメンテーション



x10302

ゲートッド クロックの代わりにクロック イネーブル ピンを使用

ザイリンクスでは、ゲートッド クロックの代わりに CLB のクロック イネーブル ピンを使用することをお勧めします。ゲートッド クロックではグリッチの発生、クロック遅延の増加、クロック スキューなどの問題が発生する可能性があります。クロック イネーブル ピンを使用すると、クロック リソースを節約でき、タイミング特性およびデザイン解析が向上します。

消費電力を削減するためにゲートッド クロックを使用する場合は、ほとんどの FPGA デバイスに含まれる **BUFGCE** と呼ばれるクロック イネーブルが付いたグローバル バッファリソースを使用できますが、デザインの一部に対してクロックを分周したり停止するには、クロック イネーブル ピンを使用する方法の方が適切です。

ゲートイッド クロックを使用した VHDL コード例

```
-- The following code is for demonstration purposes only
-- Xilinx does not suggest using the following coding style in FPGAs
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity gate_clock is
    port (DATA, IN1, IN2, LOAD, CLOCK: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end gate_clock;
architecture BEHAVIORAL of gate_clock is
    signal GATECLK: STD_LOGIC;
begin
    GATECLK <= (IN1 and IN2 and LOAD and CLOCK);
    GATE_PR: process (GATECLK)
    begin
        if (GATECLK'event and GATECLK='1') then
            OUT1 <= DATA;
        end if;
    end process; -- End GATE_PR
end BEHAVIORAL;
```

ゲートイッド クロックを使用した Verilog コード例

```
// The following code is for demonstration purposes only
// Xilinx does not suggest using the following coding style in FPGAs
module gate_clock(
    input DATA, IN1, IN2, LOAD, CLOCK,
    output reg OUT1
);
    wire GATECLK;
    assign GATECLK = (IN1 & IN2 & LOAD & CLOCK);
    always @(posedge GATECLK)
        OUT1 <= DATA;
endmodule
```

ゲートイッド クロックからクロック イネーブルに変更する例

このセクションでは、ゲートイッド クロックからクロック イネーブルに変更する VHDL および Verilog コード例を示します。

クロック イネーブルを使用した VHDL コード例

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity clock_enable is
    port (DATA, IN1, IN2, LOAD, CLOCK: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end clock_enable;
architecture BEHAVIORAL of clock_enable is
    signal ENABLE: std_logic;
begin
    ENABLE <= IN1 and IN2 and LOAD;
    EN_PR: process (CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
                OUT1 <= DATA;
            end if;
        end if;
    end process;
end BEHAVIORAL;

```

クロック イネーブルを使用した Verilog コード例

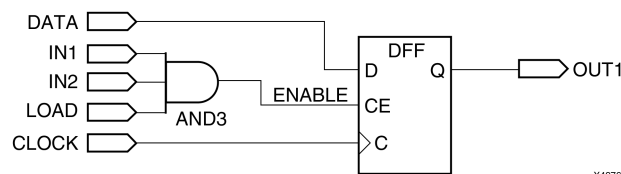
```

module clock_enable (
    input DATA, IN1, IN2, LOAD, CLOCK,
    output reg OUT1
);
    wire ENABLE;

    assign ENABLE = (IN1 & IN2 & LOAD);
    always @(posedge CLOCK)
        if (ENABLE)
            OUT1 <= DATA;
endmodule

```

クロック イネーブルのインプリメンテーション



X4976

レジスタおよびラッチの初期ステート

FPGA のフリップフロップは、スタートアップ中にプリセット (非同期セット) またはクリア (非同期リセット) のいずれかにコンフィギュレーションされます。この値は、初期化ステートまたは **INIT** と呼ばれます。レジスタの初期ステートは、次のように指定できます。

- ・ レジスタをインスタンス化する場合は、**INIT** ジェネリック/パラメータの値を **1** または **0** に設定します。詳細は、ライブラリガイドを参照してください。
- ・ レジスタが推論される場合は、次のコード例に示すように VHDL の信号宣言または Verilog の reg 宣言を初期化します。

レジスタおよびラッチの初期状態を指定する VHDL コード例 1

```
signal register1 : std_logic := '0'; -- specifying register1 to start as a zero
signal register2 : std_logic := '1'; -- specifying register2 to start as a one
signal register3 : std_logic_vector(3 downto 0):="1011"; -- specifying INIT value for 4-bit register
```

レジスタおよびラッチの初期状態を指定する Verilog コード例 1

```
reg register1 = 1'b0; // specifying register1 to start as a zero
reg register2 = 1'b1; // specifying register2 to start as a one
reg [3:0] register3 = 4'b1011; //specifying INIT value for 4-bit register
```

レジスタおよびラッチの初期状態を指定する Verilog コード例 2

Verilog では、initial 文も使用できます。

```
reg [3:0] register3;
initial begin
    register3= 4'b1011;
end
```

合成ツールによっては、このような初期化がサポートされないものもあります。サポートの有無は、合成ツールのマニュアルを参照してください。初期化がサポートされていない場合、またはコードに記述されていない場合、初期値はコードに非同期プリセットが含まれているかいないかによって決定します。非同期プリセットが含まれている場合はレジスタは 1 に初期化され、含まれていない場合は 0 に初期化されます。

シフト レジスタの初期状態

シフト レジスタの初期値の定義方法は、レジスタおよびラッチの場合と同じです。詳細は、「[レジスタおよびラッチの初期状態](#)」を参照してください。

RAM の初期状態

RAM (ブロック RAM および分散 RAM) の初期値の定義方法は、レジスタおよびラッチの場合と同様です。RAM の初期状態は、次のように指定できます。

- ・ RAM をインスタンス化する場合、**INIT_00**、**INIT_01** などの ジェネリック/パラメータの値を設定します。詳細は、ライブラリガイドを参照してください。
- ・ RAM が推論される場合は、次のコード例に示すように VHDL の信号宣言を初期化するか、または Verilog の initial 文を使用します。初期値は、HDL コードで直接指定するか、初期化データを含む外部ファイルで指定できます。

RAM の初期状態を指定する VHDL コード例

```
type ram_type is array (0 to 63) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=(
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
    X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
    X"08201", X"00500", ... );
```

RAM の初期ステートを指定する Verilog コード例

```
reg [19:0] ram [63:0];
initial begin
    ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
    ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
    ...
    ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
end
```

合成ツールによっては、このような初期化がサポートされないものもあります。サポートの有無は、合成ツールのマニュアルを参照してください。

マルチプレクサ

ザイリンクス FPGA デバイスにマルチプレクサをインプリメントするには、次のリソースを使用できます。

- ・ MUXF5、MUXF6 などの専用リソース
- ・ キャリー チェーン
- ・ LUT のみ

インプリメンテーション方法は、デザインでスピードを優先するかエリアを優先するかに応じて、合成ツールにより自動的に選択されます。合成ツールによっては、マルチプレクサのインプリメンテーション方法をユーザーが指定できるものもあります。

詳細は、合成ツールのマニュアルを参照してください。

マルチプレクサの記述方法は、**if-then-else** 文や **case** 文など複数あります。マルチプレクサを記述する際は、よくある間違いに注意してください。たとえば、**case** 文を使用してマルチプレクサを記述する場合に、セレクトのすべての値を指定しないと、マルチプレクサではなくラッチが推論されてしまいます。

Verilog の **case** 文は、次のように指定できます。

- ・ フル
- ・ フルでない

case 文に可能なすべての分岐が指定されている場合はフルです。

Verilog の **case** 文は、さらに次のように指定できます。

- ・ パラレル
- ・ パラレルでない

case 文に同時に実行可能な分岐が含まれていない場合はパラレルです。

合成ツールは、**case** 文の特性を自動的に判断し、対応するロジックを生成します。また、指示子を使用して **case** 文の解釈方法を指定することも可能です。

詳細は、合成ツールのマニュアルを参照してください。

case 文を使用した 4:1 の 1 ビット MUX の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_2 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_2;

architecture archi of multiplexers_2 is
begin
    process (a, b, c, d, s)
    begin
        case s is
            when "00" => o <= a;
            when "01" => o <= b;
            when "10" => o <= c;
            when others => o <= d;
        end case;
    end process;
end archi;
```

case 文を使用した 4:1 の 1 ビット MUX の Verilog コード例

```
module v_mults_2 (a, b, c, d, s, o);
    input a,b,c,d;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or d or s)
    begin
        case (s)
            2'b00 : o = a;
            2'b01 : o = b;
            2'b10 : o = c;
            default : o = d;
        endcase
    end
endmodule
```

if 文を使用した 4:1 の 1 ビット MUX の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_1 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_1;

architecture archi of multiplexers_1 is
begin
    process (a, b, c, d, s)
    begin
        if (s = "00") then o <= a;
        elsif (s = "01") then o <= b;
        elsif (s = "10") then o <= c;
        else o <= d;
        end if;
    end process;
end archi;
```

if 文を使用した 4:1 の 1 ビット MUX の Verilog コード例

```
module v_mults_1 (a, b, c, d, s, o);
    input a,b,c,d;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or d or s)
    begin
        if (s == 2'b00) o = a;
        else if (s == 2'b01) o = b;
        else if (s == 2'b10) o = c;
        else o = d;
    end
endmodule
```

有限ステート マシン (FSM) コンポーネント

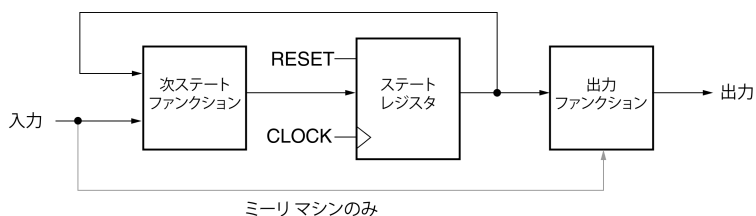
このセクションには、次の項目が含まれます。

- ・ [FSM の記述スタイル](#)
- ・ [1 つのプロセスを使用した FSM](#)
- ・ [2 つまたは 3 つのプロセスを使用した FSM](#)
- ・ [FSM の認識と最適化](#)
- ・ [その他の FSM の機能](#)

FSM の記述スタイル

ほとんどの FPGA 合成ツールでは、有限ステート マシン (FSM) を記述するテンプレートが多数提供されています。FSM コンポーネントを記述する方法は多数あります。従来からの方法では、次の図に示すように、ミラー マシンまたはムーア マシンが使用されます。

ミーリ マシンおよびムーア マシン



X10899

HDL では、FSM の記述に process ブロック (VHDL) および always ブロック (Verilog) を使用する のが最適です。ここでの説明では、「プロセス」という言葉で VHDL の process ブロックと Verilog の always ブロックの両方を示します。

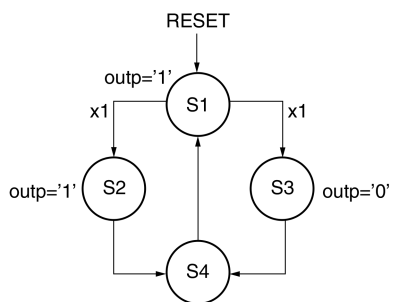
モデルの異なる部分をどのように分割するかによって、1 つの記述に複数のプロセス (1、2、または 3 個) を含めることができます。

次は、非同期リセット (RESET) 付きのムーア マシンの例です。

- ・ 4 つのステート：s1、s2、s3、s4
- ・ 5 つの遷移
- ・ 1 つの入力：x1
- ・ 1 つの出力：outp

このモデルは、次のステート ダイアグラムで表すことができます。

ステート ダイアグラム



X10900

1 つのプロセスを使用した FSM

次のコード例で、出力信号 **outp** はレジスタです。

1 つのプロセス ブロックを使用した FSM の VHDL コード例

```
----- State Machine with a single process.
--
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
           outp : OUT std_logic);
end entity;

architecture beh1 of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type ;

begin
    process (clk,reset)
    begin
        if (reset ='1') then
            state <=s1;
            outp<='1';
        elsif (clk='1' and clk'event) then
            case state is
                when s1 => if x1='1' then
                            state <= s2;
                            outp <= '1';
                        else
                            state <= s3;
                            outp <= '0';
                        end if;
                when s2 => state <= s4; outp <= '0';
                when s3 => state <= s4; outp <= '0';
                when s4 => state <= s1; outp <= '1';
            end case;
        end if;
    end process;
end beh1;
```

1 つの always ブロックを使用した FSM の Verilog コード例

```
//  
// State Machine with a single always block.  
//  
module v_fsm_1 (clk, reset, x1, outp);  
    input clk, reset, x1;  
    output outp;  
    reg outp;  
    reg [1:0] state;  
  
    parameter s1 = 2'b00; parameter s2 = 2'b01;  
    parameter s3 = 2'b10; parameter s4 = 2'b11;  
  
    initial begin  
        state = 2'b00;  
    end  
  
    always@(posedge clk or posedge reset)  
    begin  
        if (reset)  
            begin  
                state <= s1; outp <= 1'b1;  
            end  
        else  
            begin  
                case (state)  
                    s1: begin  
                        if (x1==1'b1)  
                            begin  
                                state <= s2;  
                                outp <= 1'b1;  
                            end  
                        else  
                            begin  
                                state <= s3;  
                                outp <= 1'b0;  
                            end  
                        end  
                    s2: begin  
                        state <= s4; outp <= 1'b1;  
                    end  
                    s3: begin  
                        state <= s4; outp <= 1'b0;  
                    end  
                    s4: begin  
                        state <= s1; outp <= 1'b0;  
                    end  
                endcase  
            end  
        end  
    end  
endmodule
```

VHDL の場合、ステートレジスタは次のタイプにできます。

- ・ **integer**
- ・ **bit_vector**
- ・ **std_logic_vector**

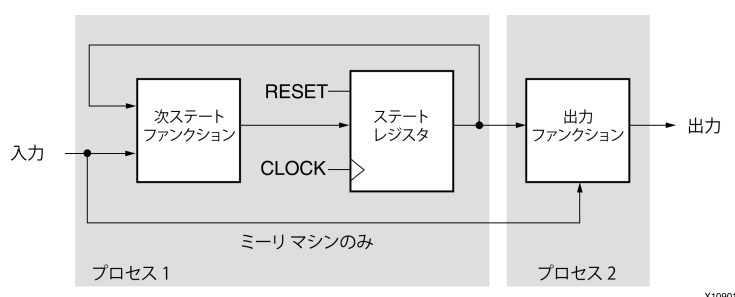
ザイリンクスでは、可能なすべてのステート値を含む列挙型を定義し、そのタイプでステートレジスタを宣言することをお勧めします。上記の VHDL コード例で使用されているのは、この方法です。

Verilog では、ステートレジスタに整数型または定義されたパラメータを使用できますが、定義されたパラメータを使用することをお勧めします。上記の Verilog コード例で使用されているのは、この方法です。

2 つまたは 3 つのプロセスを使用した FSM

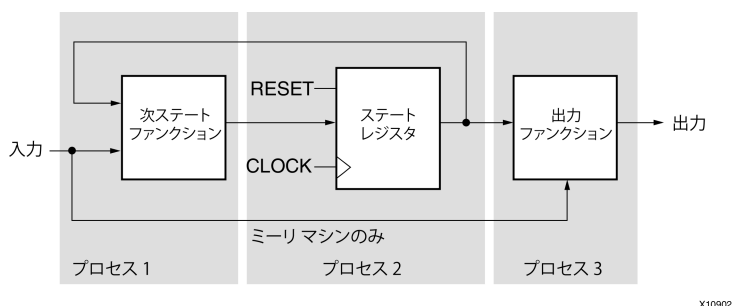
1 つのプロセスを使用した FSM は、次の図に示すように、2 つのプロセスを使用して記述できます。

2 つのプロセスを使用した FSM



1 つのプロセスを使用した FSM は、次の図に示すように、3 つのプロセスを使用して記述できます。

3 つのプロセスを使用した FSM 記述



FSM の認識と最適化

FPGA の合成ツールは、HDL コードから FSM コンポーネントを自動的に認識し、FSM 専用の最適化を実行できます。FSM を認識させるため、ステートレジスタの初期化が必要など、ご使用の合成ツールによって特定の条件がある場合があります。

詳細は、合成ツールのマニュアルを参照してください。

通常デフォルト モードでは、最高のスピードまたは最小のエリアを達成するために最適なエンコード方法が選択されます。ワンホット、シーケンシャル、グレーなど、多数のエンコード方法がサポートされています。通常、ワンホット エンコードを使用すると、FPGA アーキテクチャで効率的にステート マシンをインプリメントできます。

自動的に選択されるエンコード方式が好ましくない場合は、特定のエンコード方式を使用するよう設定できます。別の方法として、合成制約を使用して各ステートに適用するバイナリコードを直接指定することも可能です。

その他の FSM の機能

合成ツールによっては、セーフ ステート マシンや BRAM への FSM コンポーネントのインプリメンテーションなど、FSM に関するその他の機能をサポートしているものもあります。

詳細は、合成ツールのマニュアルを参照してください。

メモリのインプリメンテーション

ザイリンクス FPGA デバイスには、次のメモリが含まれています。

- ・ 分散 RAM (SelectRAM)
- ・ ブロック RAM (ブロック SelectRAM)

RAM は、次の 3 つの方法でデザインに組み込むことができます。

- ・ 合成ツールによる自動推論
- ・ CORE Generator™ ソフトウェアを使用
- ・ UNISIM または UniMacro ライブラリから専用エレメントをインスタンス化

それぞれの方法には、次の表に示すように利点と欠点があります。

RAM をデザインに組み込む 3 つの方法の利点と欠点

方法	利点	欠点
推論	<ul style="list-style-type: none"> ・ デザインに RAM を組み込む最も汎用な方法、FPGA ファミリー間で簡単に自動で移行可能 ・ シミュレーションが高速 	<ul style="list-style-type: none"> ・ 特定のコーディングスタイルが必要 ・ サポートされない RAM モードがある ・ インプリメンテーションでの制御が最小
CORE Generator ソフトウェア	RAM の生成をより制御可能	<ul style="list-style-type: none"> ・ 異なる FPGA ファミリーへの移行が複雑になる可能性がある ・ 推論に比べてシミュレーションが低速
インスタンス化	インプリメンテーションでの制御が最大	<ul style="list-style-type: none"> ・ 異なる FPGA ファミリーへの移行が制限され、複雑になる ・ 正しい RAM コンフィギュレーションを作成するのに複数のインスタンス化が必要

ブロック RAM と分散 RAM の両方で同期書き込みがサポートされています。ブロック RAM は同期読み出し、分散 RAM は非同期または同期読み出しにコンフィギュレーションできます。

分散 RAM およびブロック RAM のどちらを使用するかは、通常 RAM のサイズによって決まります。RAM のワード数が比較的小さい場合は分散 RAM を使用する方が有益で、ワード数が大きい場合はブロック メモリを使用する方が有益です。

メモリ記述がブロック RAM または分散 RAM のどちらを使用してもインプリメントできる場合は、RAM のサイズ、デザインのスピード要件およびエリア要件に応じてツールによりインプリメント方法が選択されます。合成ツールによる自動選択で要件が満たされない場合は、専用の制約を使用して RAM タイプを指定できます。

詳細は、合成ツールのマニュアルを参照してください。

ザイリンクス RAM はすべて初期化できるので、ROM または内容が定義済みの RAM としてもコンフィギュレーションできます。RAM は、HDL コードで直接初期化できます。

合成ツールによっては、パイプライン化、ブロック RAM の自動パック、自動ブロック RAM リソース管理など、RAM の推論および最適化をより詳細に制御できます。

詳細は、合成ツールのマニュアルを参照してください。

メモリのインプリメンテーションに関する追加情報は、次のセクションを参照してください。

- ・ [ブロック RAM の推論](#)
- ・ [分散 RAM の推論](#)

ブロック RAM の推論

ザイリンクスブロック RAM は、完全なデュアル ポート ブロック RAM リソースです。各ポートは完全に独立しており、異なるワード数および幅にコンフィギュレーションできます。読み出しおよび書き込みは、同期操作です。ブロック RAM リソースでは、次の読み出し/書き込み同期モードがサポートされます。

- ・ READ_FIRST (書き込み前に読み出し)
- ・ WRITE_FIRST (透過)
- ・ NO_CHANGE (変化なし)

Virtex®-5 などの最新の FPGA デバイス ファミリには、次のような改善点も含まれます。

- ・ カスケード接続可能なブロック RAM
- ・ パイプライン化された出力レジスタ
- ・ バイト幅書き込みイネーブル

ブロック RAM の推論機能は、合成ツールによって異なります。

詳細は、合成ツールのマニュアルを参照してください。

このセクションに示すコード例は、頻繁に使用されるブロック RAM コンフィギュレーションのコーディング スタイルであり、ほとんどの合成ツールでサポートされています。

READ_FIRST モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

READ_FIRST モードのシングル ポート RAM の VHDL コード例

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_01 is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM: ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                end if;
                do <= RAM(conv_integer(addr)) ;
            end if;
        end if;
    end process;

end syn;

```

READ_FIRST モードのシングル ポート RAM の Verilog コード例

```

module v_rams_01 (clk, en, we, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
                RAM[addr]<=di;
            do <= RAM[addr];
        end
    end
end
endmodule

```

WRITE_FIRST モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

WRITE_FIRST モードのシングル ポート RAM の VHDL コード例 1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02a is
    port (clk  : in std_logic;
          we   : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_02a;

architecture syn of rams_02a is
    type ram_type is array (63 downto 0)
        of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                    do <= di;
                else
                    do <= RAM( conv_integer(addr));
                end if;
            end if;
        end if;
    end process;

end syn;
```

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 1

```

module v_rams_02a (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
            begin
                RAM[addr] <= di;
                do <= di;
            end
            else
                do <= RAM[addr];
            end
        end
    end
endmodule

```

WRITE_FIRST モードのシングル ポート RAM の VHDL コード例 2

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02b is
port (clk : in std_logic;
      we : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      di : in std_logic_vector(15 downto 0);
      do : out std_logic_vector(15 downto 0));
end rams_02b;

architecture syn of rams_02b is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addr: std_logic_vector(5 downto 0);
begin
    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    ram(conv_integer(addr)) <= di;
                end if;
                read_addr <= addr;
            end if;
        end if;
    end process;

    do <= ram(conv_integer(read_addr));
end syn;

```

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 2

```

module v_rams_02b (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [5:0] read_addr;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
                RAM[addr] <= di;
            read_addr <= addr;
        end
    end

    assign do = RAM[read_addr];

endmodule

```

NO_CHANGE モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

NO_CHANGE モードのシングル ポート RAM の VHDL コード例

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_03 is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_03;

architecture syn of rams_03 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin
    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                else
                    do <= RAM( conv_integer(addr));
                end if;
            end if;
        end if;
    end process;
end syn;

```

NO_CHANGE モードのシングル ポート RAM の VHDL コード例

```

module v_rams_03 (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
                RAM[addr] <= di;
            else
                do <= RAM[addr];
            end
        end
    end

endmodule

```


書き込みポートが 1 つある READ_FIRST モードのデュアル ポート RAM のピンの説明

I/O ピン	説明
CLKA、CLKB	クロック (立ち上がりエッジ)
ENA	プライマリ グローバル イネーブル (アクティブ High)
ENB	デュアル グローバル イネーブル (アクティブ High)
WEA	プライマリ同期書き込み
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRB	デュアル読み出しアドレス
DIA	プライマリ データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

書き込みポートが 1 つある READ_FIRST モードのデュアル ポート RAM の VHDL コード例

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_01_1 is
  port (clka, clkb      : in std_logic;
        wea             : in std_logic;
        ena, enb        : in std_logic;
        addra, addrb    : in std_logic_vector(5 downto 0);
        dia             : in std_logic_vector(15 downto 0);
        doa, dob        : out std_logic_vector(15 downto 0));
end rams_01_1;

architecture syn of rams_01_1 is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM: ram_type;
begin

  process (clka)
  begin
    if clka'event and clka = '1' then
      if ena = '1' then
        if wea = '1' then
          RAM(conv_integer(addra)) <= dia;
        end if;
        doa <= RAM(conv_integer(addra)) ;
      end if;
    end if;
  end process;

  process (clkb)
  begin
    if clkb'event and clkb = '1' then
      if enb = '1' then
        dob <= RAM(conv_integer(addrb)) ;
      end if;
    end if;
  end process;

end syn;

```

書き込みポートが 1 つある READ_FIRST モードのデュアル ポート RAM の Verilog コード例

```
module v_rams_01_1 (clk_a, clk_b, ena, enb, wea, addra, addrb, dia, doa, dob);

    input  clk_a, clk_b;
    input  wea;
    input  ena, enb;
    input  [5:0] addra, addrb;
    input  [15:0] dia;
    output [15:0] doa, dob;
    reg    [15:0] RAM [63:0];
    reg    [15:0] doa, dob;

    always @(posedge clk_a)
    begin
        if (ena)
            begin
                if (wea)
                    RAM[addra] <= dia;
                doa <= RAM[addra];
            end
        end

    always @(posedge clk_b)
    begin
        if (enb)
            begin
                dob <= RAM[addrb];
            end
        end
    end

endmodule
```

書き込みポートが 2 つある READ_FIRST モードのデュアル ポート RAM

一部の合成ツールでは、書き込みポートが 2 つあるデュアル ポート ブロック RAM が VHDL と Verilog の両方でサポートされます。デュアル書き込みポートでは、データポートが 2 つあるだけでなく、各ポートに個別の書き込みクロックおよび書き込みイネーブルを使用できます。デュアル ポート ブロック RAM には 2 つのクロックがあり、1 つはプライマリの読み出しと書き込みポートで共有され、もう 1 つはセカンダリの読み出しと書き込みポートで共有されるので、各ポートに個別の書き込みクロックを使用する場合、読み出しクロックも個別になることに注意してください。このタイプのブロック RAM は、VHDL では SHARED 変数を使用して記述されています。

SHARED 変数があるため、各ポートに対する同期読み出し/書き込みの記述がシングル書き込みポートがある RAM で推奨されるコード例とは異なる場合があります。コードを記述する順序が重要なので注意してください。READ_FIRST の同期を記述する次の VHDL コード例では、書き込みポートの前に読み出しポートを記述する必要があります。

書き込みポートが 2 つある READ_FIRST モードのデュアル ポート RAM のピンの説明

I/O ピン	説明
CLKA、CLKB	クロック（立ち上がりエッジ）
ENA	プライマリ グローバル イネーブル（アクティブ High）
ENB	デュアル グローバル イネーブル（アクティブ High）
WEA、WEB	プライマリ同期書き込みイネーブル（アクティブ High）
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRB	デュアル読み出しアドレス
DIA	プライマリ データ入力
DIB:	デュアル データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

書き込みポートが 2 つある READ_FIRST モードのデュアル ポート RAM の VHDL
コード例

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_16 is
    port(clka : in std_logic;
         clkb : in std_logic;
         ena : in std_logic;
         enb : in std_logic;
         wea : in std_logic;
         web : in std_logic;
         addra : in std_logic_vector(5 downto 0);
         addrb : in std_logic_vector(5 downto 0);
         dia : in std_logic_vector(15 downto 0);
         dib : in std_logic_vector(15 downto 0);
         doa : out std_logic_vector(15 downto 0);
         dob : out std_logic_vector(15 downto 0));
end rams_16;

architecture syn of rams_16 is
    type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                DOA <= RAM(conv_integer(ADDRA));
                if WEA = '1' then
                    RAM(conv_integer(ADDRA)) := DIA;
                end if;
            end if;
        end if;
    end process;

    process (CLKB)
    begin
        if CLKB'event and CLKB = '1' then
            if ENB = '1' then
                DOB <= RAM(conv_integer(ADDRB));
                if WEB = '1' then
                    RAM(conv_integer(ADDRB)) := DIB;
                end if;
            end if;
        end if;
    end process;

end syn;
```

書き込みポートが 2 つある READ_FIRST モードのデュアル ポート RAM の Verilog コード例

```
module v_rams_16 (clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

    input  clka,clkb,ena,enb,wea,web;
    input  [5:0]  addra,addrb;
    input  [15:0] dia,dib;
    output [15:0] doa,dob;
    reg    [15:0] ram [63:0];
    reg    [15:0] doa,dob;

    always @(posedge clka) begin
        if (ena)
            begin
                if (wea)
                    ram[addra] <= dia;
                doa <= ram[addra];
            end
        end

    always @(posedge clkb) begin
        if (enb)
            begin
                if (web)
                    ram[addrb] <= dib;
                dob <= ram[addrb];
            end
        end
    end

endmodule
```

分散 RAM の推論

次に示すコード例は、頻繁に使用される分散 RAM コンフィギュレーションのコーディング スタイルであり、ほとんどの合成ツールでサポートされています。

シングル ポート分散 RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	読み出し/書き込みアドレス
di	データ入力
do	データ出力

シングル ポート分散 RAM の VHDL コード例

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_04 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_04;

architecture syn of rams_04 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
        end if;
    end process;

    do <= RAM(conv_integer(a));

end syn;

```

シングル ポート分散 RAM の Verilog コード例

```

module v_rams_04 (clk, we, a, di, do);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        end

    assign do = ram[a];

endmodule

```

デュアル ポート分散 RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	書き込みアドレス/プライマリ読み出しアドレス
DPRA	デュアル読み出しアドレス
di	データ入力
SPO	プライマリ出力ポート
DPO	デュアル出力ポート

デュアル ポート分散 RAM の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_09 is
    port (clk : in std_logic;
          we : in std_logic;
          a : in std_logic_vector(5 downto 0);
          dpra : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          spo : out std_logic_vector(15 downto 0);
          dpo : out std_logic_vector(15 downto 0));
end rams_09;

architecture syn of rams_09 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
        end if;
    end process;

    spo <= RAM(conv_integer(a));
    dpo <= RAM(conv_integer(dpra));

end syn;
```

デュアル ポート分散 RAM の Verilog コード例

```
module v_rams_09 (clk, we, a, dpra, di, spo, dpo);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        end

        assign spo = ram[a];
        assign dpo = ram[dpra];
    end

endmodule
```

数値演算

ザイリンクス FPGA デバイスには、従来から LUT やキャリー チェーンなどのハードウェア リソースが含まれています。これらのハードウェア リソースは、加算器、減算器、カウンタ、アキュムレータ、コンパレータなどの演算を効率的にインプリメントします。

Virtex®-4 デバイスから、DSP48 という新しいプリミティブが導入されました。このブロックは、Virtex-5 および Spartan®-3A DSP などの新規デバイスでさらに向上しています。DSP48 を使用すると、乗算器、加算器、カウンタ、バレル シフタ、コンパレータ、アキュムレータ、積和演算器、複素乗算器など、さまざまなファンクションを作成できます。

現在のところ、合成ツールで乗算器、加減算器、乗加算/減算器、積和演算器などの DSP アプリケーション用に DSP48 モードがサポートされ、頻繁に使用されます。また、DSP48 に含まれる内部レジスタおよびダイナミック OPMODE ポートも使用できます。

DSP48 の高速接続により、の高速 DSP48 チェーンをフィルタとして効率的に作成できます。この高速接続は、現在の合成ツールでサポートされています。

DSP48 のサポート レベルは合成ツールによって異なります。

詳細は、合成ツールのマニュアルを参照してください。

ターゲット デバイスにある数値演算操作をインプリメントする方法は複数あり、操作のタイプ、サイズ、使用される状況、タイミング要件などに応じて、合成ツールにより自動的に選択されます。合成ツールによる自動選択で要件が満たされない場合は、XST の **use_dsp48** や Synplicity の **syn_dspstyle** など、インプリメンテーション プロセスを制御する制約が用意されています。

詳細は、合成ツールのマニュアルを参照してください。

デザインを DSP48 ブロックを含む FPGA デバイス ファミリに移行し、DSP48 ブロックの機能を利用する場合は、最適なパフォーマンスを得るため次の事項に注意してください。

- ・ DSP48 は、完全にパイプライン化すると最高のパフォーマンスを得られます。最高のパフォーマンスを達成するには、パイプライン段を追加してください。
- ・ 内部 DSP48 レジスタでは、同期セットおよびリセット信号がサポートされます。非同期セットおよびリセット信号はサポートされません。非同期の初期化信号は、同期信号に置き換える必要があります。合成ツールによっては、この置換が自動的に実行されます。この処理により、生成されたネットリストが元の RTL 記述と異なるものになります。

詳細は、合成ツールのマニュアルを参照してください。

- ・ DSP アプリケーションで DSP48 の機能を最大限に利用するには、RTL 記述にツリー構造ではなくチェーン構造を使用してください。

DSP48 ブロックおよび DSP アプリケーション特定のコーディング スタイルについては、ターゲット ファミリ用の [XtremeDSP™ ユーザー ガイド](#)を参照してください。

符号なし 8 ビット加算器の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_01 is
    port(A,B : in std_logic_vector(7 downto 0);
         SUM : out std_logic_vector(7 downto 0));
end arith_01;

architecture archi of arith_01 is
begin

    SUM <= A + B;

end archi;
```


符号なし 8 ビット加算器の Verilog コード例

```
module v_arith_01(A, B, SUM);
    input [7:0] A;
    input [7:0] B;
    output [7:0] SUM;

    assign SUM = A + B;

Endmodule
```

符号付き 8 ビット加算器の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity arith_02 is
    port(A,B : in std_logic_vector(7 downto 0);
         SUM : out std_logic_vector(7 downto 0));
end arith_02;

architecture archi of arith_02 is
begin

    SUM <= A + B;

end archi;
```

符号付き 8 ビット加算器の Verilog コード例

```
module v_arith_02 (A,B,SUM);
    input signed [7:0] A;
    input signed [7:0] B;
    output signed [7:0] SUM;
    wire signed [7:0] SUM;

    assign SUM = A + B;

Endmodule
```

レジスタ付き入力/出力を持つ符号なし 8 ビット加算器の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_03 is
    port(clk : in std_logic;
          A,B : in std_logic_vector(7 downto 0);
          SUM : out std_logic_vector(7 downto 0));
end arith_03;

architecture archi of arith_03 is
    signal reg_a, reg_b: std_logic_vector(7 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            reg_a <= A;
            reg_b <= B;
            SUM <= reg_a + reg_b;
        end if;
    end process;

end archi;
```

レジスタ付き入力/出力を持つ符号なし 8 ビット加算器の Verilog コード例

```
module v_arith_03 (clk, A, B, SUM);
    input      clk;
    input [7:0] A;
    input [7:0] B;
    output [7:0] SUM;

    reg [7:0] reg_a, reg_b, SUM;

    always @(posedge clk)
    begin
        reg_a <= A;
        reg_b <= B;
        SUM    <= reg_a + reg_b;
    end

endmodule
```

符号なし 8 ビット加減算器の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_04 is
    port(A,B : in std_logic_vector(7 downto 0);
          OPER: in std_logic;
          RES : out std_logic_vector(7 downto 0));
end arith_04;

architecture archi of arith_04 is
begin

    RES <= A + B when OPER='0'
           else A - B;

end archi;
```

符号なし 8 ビット加減算器の Verilog コード例

```
module v_arith_04 (A, B, OPER, RES);
    input OPER;
    input [7:0] A;
    input [7:0] B;
    output [7:0] RES;
    reg [7:0] RES;

    always @(A or B or OPER)
    begin
        if (OPER==1'b0) RES = A + B;
        else RES = A - B;
    end

endmodule
```

符号なし 8 ビットコンパレータの VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_05 is
    port(A,B : in std_logic_vector(7 downto 0);
          CMP : out std_logic);
end arith_05;

architecture archi of arith_05 is
begin

    CMP <= '1' when A >= B else '0';

end archi;
```

符号なし 8 ビット コンパレータの Verilog コード例

```
module v_arith_05 (A, B, CMP);
    input  [7:0] A;
    input  [7:0] B;
    output CMP;

    assign CMP = (A >= B) ? 1'b1 : 1'b0;

endmodule
```

レジスタ付き入力/出力を持つ符号なし 17x17 ビット乗算器の VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arith_06 is
    port(clk : in std_logic;
         A : in unsigned (16 downto 0);
         B : in unsigned (16 downto 0);
         MULT : out unsigned (33 downto 0));
end arith_06;

architecture beh of arith_06 is
    signal reg_a, reg_b : unsigned (16 downto 0);

begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            reg_a <= A; reg_b <= B;
            MULT <= reg_a * reg_b;
        end if;
    end process;

end beh;
```

レジスタ付き入力/出力を持つ符号なし 17x17 ビット乗算器の Verilog コード例

```
module v_arith_06(clk, A, B, MULT);

    input      clk;
    input  [16:0] A;
    input  [16:0] B;
    output [33:0] MULT;

    reg [33:0] MULT;
    reg [16:0] reg_a, reg_b;

    always @(posedge clk)
    begin
        reg_a <= A;
        reg_b <= B;
        MULT <= reg_a * reg_b;
    end
endmodule
```

同期リセットを持つ符号なし 8 ビット アップ カウンタの VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_07 is
    port(clk, reset : in std_logic;
          Res : out std_logic_vector(7 downto 0));
end arith_07;

architecture archi of arith_07 is
    signal cnt: std_logic_vector(7 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                cnt <= "00000000";
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end process;

    Res <= cnt;

end archi;
```

同期リセットを持つ符号なし 8 ビット アップ カウンタの Verilog コード例

```
module v_arith_07 (clk, reset, Res);
    input      clk, reset;
    output [7:0] Res;

    reg [7:0] cnt;

    always @(posedge clk)
    begin
        if (reset)
            cnt <= 8'b00000000;
        else
            cnt <= cnt + 1'b1;
        end

        assign Res = cnt;
    endmodule
```

同期リセットを持つ符号なし 8 ビット アップ アキュムレータの VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arith_08 is
    port(clk, reset : in std_logic;
          din : in std_logic_vector(7 downto 0);
          Res : out std_logic_vector(7 downto 0));
end arith_08;

architecture archi of arith_08 is
    signal accu: std_logic_vector(7 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                accu <= "00000000";
            else
                accu <= accu + din;
            end if;
        end if;
    end process;

    Res <= accu;

end archi;
```

同期リセットを持つ符号なし 8 ビット アップ アキュムレータの Verilog コード例

```
module v_arith_08 (clk, reset, din, Res);
    input      clk, reset;
    input  [7:0] din;
    output [7:0] Res;

    reg [7:0] accu;

    always @(posedge clk)
    begin
        if (reset)
            accu <= 8'b00000000;
        else
            accu <= accu + din;
        end

        assign Res = accu;
    endmodule
```

乗算器の入力に 2 段のレジスタ、乗算器の後に 1 段のレジスタ、加算器の後に 1 段のレジスタが付いた乗加算器の VHDL コード例

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity arith_09 is
    generic (p_width: integer:=8);
    port (clk : in std_logic;
          A, B : in std_logic_vector(7 downto 0);
          C : in std_logic_vector(15 downto 0);
          RES : out std_logic_vector(15 downto 0));
end arith_09;

architecture beh of arith_09 is
    signal reg1_A, reg2_A,
           reg1_B, reg2_B : std_logic_vector(7 downto 0);
    signal reg_C, reg_mult : std_logic_vector(15 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            reg1_A <= A; reg2_A <= reg1_A;
            reg1_B <= B; reg2_B <= reg1_B;
            reg_C <= C;
            reg_mult <= reg2_A * reg2_B;
            RES <= reg_mult + reg_C;
        end if;
    end process;

end beh;
```

乗算器の入力に 2 段のレジスタ、乗算器の後に 1 段のレジスタ、加算器の後に 1 段のレジスタが付いた乗加算器の Verilog コード例

```
module v_arith_09 (clk, A, B, C, RES);

    input      clk;
    input [7:0] A;
    input [7:0] B;
    input [15:0] C;
    output [15:0] RES;
    reg [7:0] reg1_A, reg2_A, reg1_B, reg2_B;
    reg [15:0] reg_C, reg_mult, RES;

    always @(posedge clk)
    begin
        reg1_A <= A; reg2_A <= reg1_A;
        reg1_B <= B; reg2_B <= reg1_B;
        reg_C <= C;
        reg_mult <= reg2_A * reg2_B;
        RES <= reg_mult + reg_C;
    end

endmodule
```

乗算器の入力に 2 段のレジスタ、乗算器の後に 1 段のレジスタ、アキュムレータの後に 1 段のレジスタが付いた乗算アップ アキュムレータの VHDL コード例

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity arith_10 is
    port (clk : in std_logic;
          A, B : in std_logic_vector(7 downto 0);
          RES : out std_logic_vector(15 downto 0));
end arith_10;

architecture beh of arith_10 is
    signal reg1_A, reg2_A,
           reg1_B, reg2_B : std_logic_vector(7 downto 0);
    signal reg_mult, reg_accu : std_logic_vector(15 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            reg1_A <= A; reg2_A <= reg1_A;
            reg1_B <= B; reg2_B <= reg1_B;
            reg_mult <= reg2_A * reg2_B;
            reg_accu <= reg_accu + reg_mult;
        end if;
    end process;

    RES <= reg_accu;

end beh;
```

乗算器の入力に 2 段のレジスタ、乗算器の後に 1 段のレジスタ、アキュムレータの後に 1 段のレジスタが付いた乗算アップ アキュムレータの Verilog コード例

```
module v_arith_10 (clk, A, B, RES);

    input      clk;
    input [7:0] A;
    input [7:0] B;
    output [15:0] RES;
    reg [7:0] reg1_A, reg2_A, reg1_B, reg2_B;
    reg [15:0] reg_mult, reg_accu;
    wire [15:0] RES;

    always @(posedge clk)
    begin
        reg1_A <= A; reg2_A <= reg1_A;
        reg1_B <= B; reg2_B <= reg1_B;
        reg_mult <= reg2_A * reg2_B;
        reg_accu <= reg_accu + reg_mult;
    end

    assign RES = reg_accu;

endmodule
```


演算関数の順序およびグループ化

演算関数の順序およびグループ化は、デザインのパフォーマンスに影響します。たとえば、次の 2 つの VHDL 文は必ずしも同等ではありません。

```
ADD <= A1 + A2 + A3 + A4;  
ADD <= (A1 + A2) + (A3 + A4);
```

Verilog でも、次の 2 つの文は必ずしも同等ではありません。

```
ADD = A1 + A2 + A3 + A4;  
ADD = (A1 + A2) + (A3 + A4);
```

最初の文では、加算器が 3 個カスケード接続されます。2 番目の文では、**A1 + A2** と **A3 + A4** の 2 個の加算が並列して実行され、その結果が 3 個目の加算器で加算されます。RTL シミュレーションの結果は、両方の文で同じになりますが、合成後には 2 番目の文の結果の方が、入力信号のビット幅によっては回路が高速になります。

通常は、2 番目の文の方が回路が高速になりますが、場合によっては最初の文を使用した方が良い場合もあります。たとえば、**A4** 信号がほかの信号よりも遅れて加算器に到達する場合、最初の文の方が **A4** のロジックレベルが少なくなるため、高速になります。この構造では、**A4** がほかの信号に追いつくことが可能です。この場合、信号の速さは **A1**、**A2**、**A3**、**A4** の順になります。

ほとんどの合成ツールでは、タイミング制約を指定すれば、演算ツリーのバランスを取ったり構造を変更したりすることが可能ですが、選択した構造でデザインのコードを記述することをお勧めします。

リソース共有

リソースの共有では、1 個のファンクション ブロック (加算器やコンパレータなど) を使用して HDL コードの複数の演算子をインプリメントします。リソースの共有を使用してゲート数および配線密度を削減することで、デザインのパフォーマンスを向上できます。リソースの共有を使用しない場合、各 HDL 演算はそれぞれ個別の回路で構築されます。デザインのスピードクリティカル パスには、リソースの共有を使用しないでください。

次の演算子は、同じ演算子のインスタンスまたは同じ行の演算子と共有できます。

- ・ *
- ・ + -
- ・ > >= < <=

たとえば + 演算子は、ほかの + 演算子のインスタンスまたは - 演算子と共有できます。* 演算子は、ほかの * 演算子としか共有できません。

次の演算ファンクションは、ゲートでインプリメントするか、または合成ツールのモジュール ライブラリでインプリメントできます。

- ・ +
- ・ -
- ・ マグニチュード コンパレータ

ライブラリ ファンクションでは、FPGA デバイスのキャリー ロジックを使用したモジュールを使用します。キャリー ロジックおよびその専用配線を使用すると、4 ビット以上の演算ファンクションを高速に処理できます。デザインに 4 ビット以上の演算ファンクションが含まれる場合や、演算ファンクションがデザインに 1 つしか含まれない場合は、モジュール ライブラリを使用するとスピードが向上します。演算ファンクションが同じプロセスに存在する場合、ほとんどの合成ツールでモジュール ライブラリのリソースが自動的に共有されます。

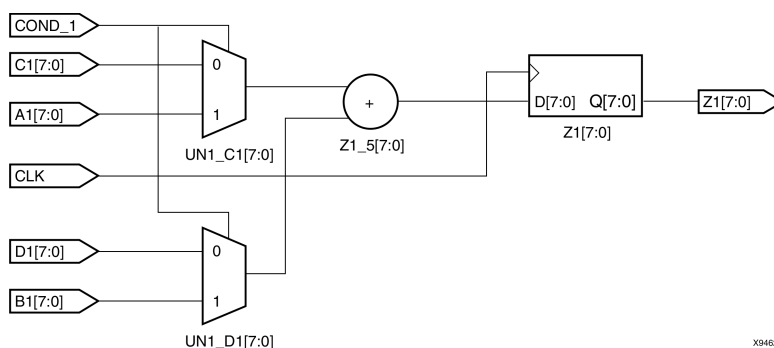
リソースの共有では、入力がマルチプレクサを介するようにして複数のファンクションをインプリメントするため、ロジックレベルが追加されます。このため、クリティカル パスの一部に演算ファンクションが含まれている場合は、リソースの共有を使用しないでください。

リソースを共有すると、デザインのリソース数を減らすことができるので、デザインで必要となるデバイスの使用エリアも減少します。共有リソースで使用されるエリアは、共有される演算のタイプおよびビット幅によって異なります。最大ビット幅に対応したすべての演算を実行する共有リソースを作成する必要があります。

デザインでリソースの共有を使用する場合、各ソースの値をマルチプレクサを介して 1 個の共有リソース入力に転送することをお勧めします。共有される演算で出力ターゲットが同じ場合、次の VHDL および Verilog のコード例に示すように、マルチプレクサ数は削減されます。

次に、VHDL の例をゲートを使用してインプリメントした図を示します。

リソースの共有のインプリメンテーション



X9462

リソース共有の VHDL コード例

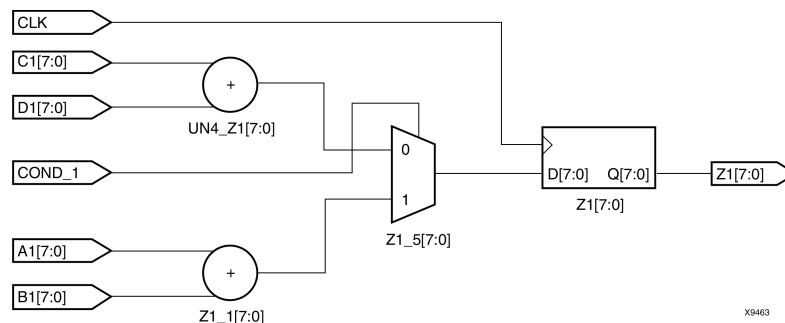
```
-- RES_SHARING.VHD
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity res_sharing is
    port (
        A1,B1,C1,D1 : in STD_LOGIC_VECTOR (7 downto 0);
        COND_1 : in STD_LOGIC;
        Z1 : out STD_LOGIC_VECTOR (7 downto 0));
end res_sharing;
architecture BEHAV of res_sharing is
begin
    P1: process (A1,B1,C1,D1,COND_1)
    begin
        if (COND_1='1') then
            Z1 <= A1 + B1;
        else
            Z1 <= C1 + D1;
        end if;
    end process; -- end P1
end BEHAV;
```

リソース共有の Verilog コード例

```
/* Resource Sharing Example
 * RES_SHARING.V
 */
module res_sharing (
    input [7:0] A1, B1, C1, D1,
    input COND_1,
    output reg [7:0] Z1);
always @(*)
begin
    if (COND_1)
        Z1 <= A1 + B1;
    else
        Z1 <= C1 + D1;
    end
endmodule
```

リソースの共有を使用しない場合、または加算器を別のプロセスで記述する場合は、次の図に示すように、デザインは 2 個のモジュールを使用してインプリメントされます。

リソースの共有を使用しないインプリメンテーション



X9463

詳細は、合成ツールのマニュアルを参照してください。

合成ツールの命名規則

ネット名およびロジック名は、合成ツールにより保持されるものと変更されるものがあります。そのため、解読しにくい、元のコードと対応させにくいネットリストが生成される可能性があります。合成ツールによって、VHDL または Verilog コードから名前を生成する方法は異なります。

合成ツールでネットリストの生成に使用される命名規則を理解しておく、次のような利点があります。

- ・ 最終的なネットリストに含まれるネット名およびコンポーネント名がどのように元の入力デザインと関連しているかを理解できます。
- ・ 合成後のデザインに含まれるネットおよび名前を元の入力デザインのものと対応させるのに役立ちます。
- ・ 生成されたネットリストでオブジェクトを検索し、ユーザー制約ファイル (UCF) でインプリメンテーション制約を適用するのに役立ちます。

詳細は、合成ツールのマニュアルを参照してください。

FPGA プリミティブのインスタンス化

ザイリンクスでは、コンポーネントとしてデザインにインスタンス化可能な、アーキテクチャ固有のカスタマイズされたコンポーネントを含むライブラリを提供しています。

インプリメンテーション ツールのライブラリに含まれているアーキテクチャ固有のコンポーネントは、定義を指定しなくてもインスタンス化できます。このようなコンポーネントは、ライブラリ ガイドに「プリミティブ」と示されています。ライブラリ ガイドに「マクロ」と示されているコンポーネントは、インプリメンテーション ツールのライブラリに含まれていないため、インスタンス化できません。マクロ コンポーネントは、回路図シンボルを定義します。マクロを使用すると、回路図ツールでネットリストが生成されるときに、プリミティブ エlement に分解されます。FPGA のプリミティブは、VHDL および Verilog でインスタンス化できます。すべての FPGA プリミティブは、UNISIM ライブラリに含まれています。

コンポーネントおよびポート マップ宣言の VHDL コード例

```
library IEEE;
use IEEE.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;
entity flops is port(
    di    : in std_logic;
    ce    : in std_logic;
    clk   : in std_logic;
    qo    : out std_logic;
    rst   : in std_logic);

end flops;

architecture inst of flops is
begin
    U0 : FDCE port map(
        D    => di,
        CE   => ce,
        C    => clk,
        CLR  => rst,
        Q    => qo);
end inst;
```

コンポーネントおよびポート マップ宣言の Verilog コード例

```
module flops (
    input d1, ce, clk, rst,
    output q1);
    FDCE u1 (
        .D (d1),
        .CE (ce),
        .C (clk),
        .CLR (rst),
        .Q (q1));
endmodule
```

合成ツールによっては、UNISIM ライブラリを明示的にプロジェクトに含める必要があります。

詳細は、合成ツールのマニュアルを参照してください。

ザイリンクス プリミティブの多くには、プロパティがあります。これらのプロパティ (制約) は、次の方法でプリミティブに追加できます。

- ・ VHDL での属性の追加
- ・ Verilog での属性の追加
- ・ VHDL でのジェネリックの追加
- ・ Verilog でのパラメータの追加
- ・ User Constraints File (UCF)

これらのプロパティの使用法は、「[属性および制約](#)」を参照してください。

CORE Generator ソフトウェア モジュールのインスタンス化

CORE Generator™ では、次のものが生成されます。

- ・ 機能を記述した EDIF または NGC ネットリスト、あるいはその両方
- ・ HDL インスタンス化用のコンポーネント インスタンス化 テンプレート

ISE で CORE Generator ソフトウェア モジュールをインスタンスエートする方法については、ISE ヘルプの「CORE Generator IP の使用」を参照してください。CORE Generator ソフトウェアの詳細は、CORE Generator ヘルプを参照してください。

属性および制約

属性および制約は、同じ意味で使用される場合と、別の意味で使用される場合があります。また、構文で属性と指示子が使用される場合も、意味は類似していますが異なります。ザイリンクスの資料では、属性および制約という用語をこのセクションで定義しているように使用します。

属性

属性は、デバイス アーキテクチャのプリミティブ コンポーネントに関連付けるプロパティで、インスタンスエートされるコンポーネントのファンクションおよびインプリメンテーションに影響します。属性は、次の方法で設定します。

- ・ VHDL : ジェネリック マップ
- ・ Verilog : defparam またはインライン パラメータ

属性の例 :

- ・ **LUT4** コンポーネント上の **INIT** プロパティ
- ・ **DCM** 上の **CLKFX_DIVIDE** プロパティ

属性はすべて、ライブラリ ガイドのプリミティブ コンポーネントの説明に含まれています。

合成制約

合成制約を使用すると、合成ツールによる特定のデザインまたは HDL コードの一部に対する最適化手法を制御できます。合成制約は、VHDL または Verilog コードに組み込むか、または別の合成制約ファイルに含めます。

次に、合成制約の例を示します。

- ・ **USE_DSP48** (XST)
- ・ **RAM_STYLE** (XST)

詳細は、合成ツールのマニュアルを参照してください。

XST 制約の詳細は、『[XST ユーザー ガイド](#)』を参照してください。

インプリメンテーション制約

インプリメンテーション制約は、FPGA インプリメンテーション ツールで FPGA デザインを処理する際に従うマップ、配置、タイミング、またはその他のガイドラインを指定します。インプリメンテーション制約は、通常 UCF ファイルに含められますが、HDL コードまたは合成制約ファイルにも含めることができます。

次に、インプリメンテーション制約の例を示します。

- ・ LOC (配置) 制約
- ・ PERIOD (タイミング) 制約

詳細は、『[制約ガイド](#)』を参照してください。

属性の使用

属性は、ザイリンクス プリミティブの動作を指定するためにインスタンス化に設定するプロパティです。ジェネリック (VHDL) またはパラメータ (Verilog) を使用し、合成およびシミュレーションの両方に正しく属性が渡されるようにします。

プリミティブ属性を設定する VHDL コード例

次に、インスタンス化される **RAM16XS** に対して **INIT** プリミティブ属性を設定する VHDL コード例を示します。この属性は、**RAM** シンボルの初期内容を 16 進数値 **A1B2** に設定します。

```
small_ram_inst : RAM16X1S
generic map (
  INIT => X"A1B2")
port map (
  O => ram_out,    -- RAM output
  A0 => addr(0),    -- RAM address[0] input
  A1 => addr(1),    -- RAM address[1] input
  A2 => addr(2),    -- RAM address[2] input
  A3 => addr(3),    -- RAM address[3] input
  D => data_in,     -- RAM data input
  WCLK => clock,    -- Write clock input
  WE => we          -- Write enable input
);
```

プリミティブ属性を設定する Verilog コード例

次に、インスタンス化される **IBUFDS** シンボルで **DIFF_TERM** を **FALSE**、**IOSTANDARD** を **LVDS_25** に指定する Verilog コードの例を示します。

```
IBUFDS #(
  .CAPACITANCE("DONT_CARE"), // "LOW", "NORMAL", "DONT_CARE" (Virtex-4/5 only)
  .DIFF_TERM("FALSE"),       // Differential Termination (Virtex-4/5, Spartan-3E/3A)
  .IBUF_DELAY_VALUE("0"),    // Specify the amount of added input delay for
                              // the buffer, "0"-"16" (Spartan-3E/3A only)
  .IFD_DELAY_VALUE("AUTO"),  // Specify the amount of added delay for input
                              // register, "AUTO", "0"-"8" (Spartan-3E/3A only)
  .IOSTANDARD("DEFAULT")    // Specify the input I/O standard
) IBUFDS_inst (
  .O(0), // Buffer output
  .I(I), // Diff_p buffer input (connect directly to top-level port)
  .IB(IB) // Diff_n buffer input (connect directly to top-level port)
);
```

合成制約の使用

このセクションでは、合成制約の使用について説明します。次の内容が含まれます。

- ・ [合成制約の使用について](#)
- ・ [VHDL 合成属性を渡す](#)
- ・ [Verilog 合成属性を渡す](#)

合成制約の使用について

制約は、デザインに含まれる HDL オブジェクトに設定するか、または制約ファイルで指定します。制約を HDL オブジェクトに渡すには、次の 2 つの方法があります。

- ・ オブジェクトを記述したデータをあらかじめ定義する。
- ・ 属性を直接 HDL オブジェクトに設定する。

あらかじめ定義された属性は、合成ツールに含まれる COMMAND ファイルもしくは制約ファイルで渡すか、または直接 HDL コードで設定できます。

このセクションでは、属性を HDL コードで設定する方法のみを説明します。コマンド ファイルを使用して属性を渡す方法は、合成ツールのマニュアルを参照してください。

VHDL 合成属性を渡す

次に、合成属性を VHDL コードで設定する例を示します。

- ・ 属性宣言
- ・ ポートまたは信号での属性の使用
- ・ インスタンスでの属性の使用
- ・ コンポーネントでの属性の使用

属性宣言

```
attribute attribute_name : attribute_type;
```

ポートまたは信号での属性の使用

```
attribute attribute_name of object_name : signal is attribute_value
```

次に例を示します。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d_reg is
    port (
        CLK, DATA: in STD_LOGIC;
        Q: out STD_LOGIC);
    attribute FAST : string;
    attribute FAST of Q : signal is "true";
end d_reg;
```

インスタンスでの属性の使用

```
attribute attribute_name of object_name : label is attribute_value
```

次に例を示します。

```
architecture struct of spblkrams is
    attribute LOC: string;
    attribute LOC of SDRAM_CLK_IBUFG: label is "AA27";
Begin
    -- IBUFG: Single-ended global clock input buffer
    -- All FPGA
    -- Xilinx HDL Language Template
    SDRAM_CLK_IBUFG : IBUFG
    generic map (
        IOSTANDARD => "DEFAULT")
    port map (
        O => SDRAM_CLK_o, -- Clock buffer output
        I => SDRAM_CLK_i -- Clock buffer input
    );
    -- End of IBUFG_inst instantiation
```


コンポーネントでの属性の使用

```
attribute attribute_name of object_name : component  
is attribute_value
```

次に例を示します。

```
architecture xilinx of tenths_ex is  
attribute black_box : boolean;  
component tenths  
    port (  
        CLOCK : in STD_LOGIC;  
        CLK_EN : in STD_LOGIC;  
        Q_OUT : out STD_LOGIC_VECTOR(9 downto 0)  
    );  
end component;  
attribute black_box of tenths : component is true;  
begin
```

Verilog 合成属性を渡す

ほとんどのベンダーで VHDL で属性を設定するために採用されている構文は同じですが、Verilog では異なる構文が採用されています。Verilog では、メタコメントと呼ばれる方法を使用して属性を渡すのが一般的ですが、合成ツールによって異なる構文が使用されています。メタコメントの構文は、合成ツールのマニュアルを参照してください。

Verilog 2001 では、属性を設定する一定の構文が提供されています。属性はオブジェクト宣言の直前に宣言されるので、オブジェクト名は属性の宣言には含まれません。

```
(* attribute_name = "attribute_value" *)  
Verilog_object;
```

次に例を示します。

```
(* RLOC = "R1C0.S0" *) FDCE #(  
    .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)  
) U2 (  
    .Q(q1), // Data output  
    .C(clk), // Clock input  
    .CE(ce), // Clock enable input  
    .CLR(rst), // Asynchronous clear input  
    .D(q0) // Data input  
);
```

この属性の設定方法は、合成ツールによってはサポートされない場合があります。

詳細は、合成ツールのマニュアルを参照してください。

パイプライン

パイプラインを使用すると、次のような利点があります。

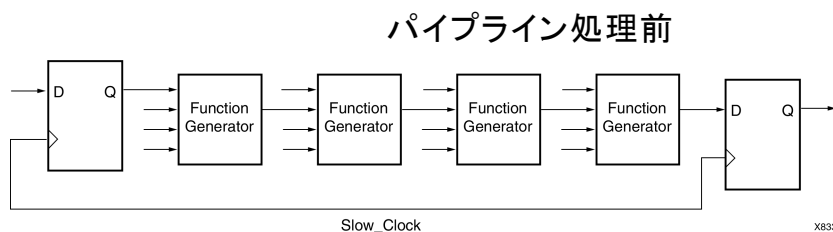
- ・ レイテンシが増えてデータを処理するクロック サイクル数が増えますが、デバイスのパフォーマンスを大幅に向上できます。
- ・ 長いデータ パスを数個のロジックレベルで再構築し、複数クロック サイクルに分割することにより、パフォーマンスを向上できます。
- ・ クロック サイクルが高速になり、レイテンシは増加しますが、スループットを向上できます。

ザイリンクス FPGA はレジスタが豊富なので、パイプラインを作成するのにデバイスリソースの点でコストがかかりません。パイプラインを使用すると、データが複数サイクル パス上になるため、デザインの残りの部分で追加されたパスのレイテンシを考慮する必要があります。これらのパスのタイミング仕様を定義するときにも、注意が必要です。

パイプライン処理前

次のパイプライン処理前の図では、クロック スピードが次のものによって制限されます。

- ・ ソース フリップフロップの clock-to-out タイム
- ・ ロジック レベル 4 段を介したロジック遅延
- ・ 4 個のファンクション ジェネレータに関連した配線
- ・ デスティネーション レジスタのセット アップ タイム

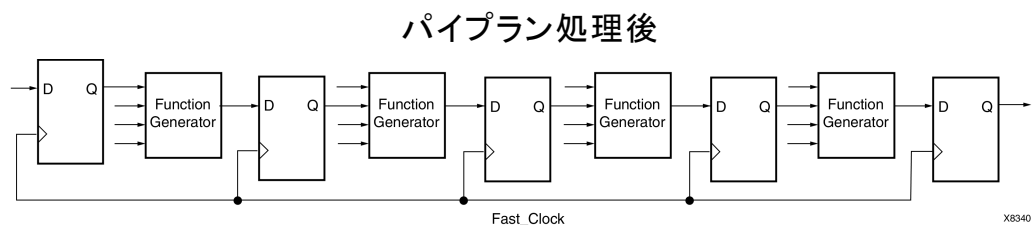


パイプライン処理後

次に示すパイプライン処理後の図は、「パイプライン処理前」に示すデータ パスをパイプライン処理した例を示します。フリップフロップはファンクション ジェネレータと同じ CLB に含まれているので、クロック スピードは次のもので制限されます。

- ・ ソース フリップフロップの clock-to-out タイム
- ・ ロジック レベル 1 段を介したロジック遅延、1 段の配線遅延
- ・ デスティネーション レジスタのセット アップ タイム

この例では、パイプライン処理前よりシステム クロックが高速になります。



リタイミング

合成ツールによっては、デザインのスピードを向上するため、ロジックの前後でレジスタを順方向または逆方向に自動的に移動する機能があります。この機能の特徴は、次のとおりです。

- ・ 合成ツールによってリタイミングまたはレジスタの自動調整と呼ばれます。
- ・ デザインを変更せずにデザインのスピードを増加させることができます。
- ・ フリップフロップの数が大幅に増加する場合があります。

詳細は、合成ツールのマニュアルを参照してください。

デザインのシミュレーション

この章では、ザイリンクスおよびサードパーティのソフトウェアを使用した基本的な HDL シミュレーション フローについて説明します。次のセクションが含まれています。

- ・ 業界標準規格への準拠
- ・ HDL デザイン フローのシミュレーション ポイント
- ・ テストベンチを使用したスティミュラスの指定
- ・ VHDL および Verilog のライブラリとモデル
- ・ 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
- ・ シミュレーションでのブロック RAM 競合チェックのディスエーブル
- ・ シミュレーションでのグローバル リセットおよびトライステート
- ・ デザイン階層とシミュレーション
- ・ ザイリンクス ライブラリを使用した RTL シミュレーション
- ・ ゲートレベル ネットリストの生成 (NetGen の実行)
- ・ 同期エレメントでの X 伝搬のディスエーブル
- ・ ASYNC_REG 制約の使用
- ・ MIN/TYP/MAX シミュレーション
- ・ CLKDLL、DCM、および DCM_ADV に関する注意事項
- ・ タイミング シミュレーションの理解
- ・ ザイリンクスでサポートされる EDA シミュレーション ツールを使用したシミュレーション

デザイン サイズが大きくなり、複雑性が増したことに加え、合成およびシミュレーション ツールが向上したことにより、HDL が集積回路設計で最もよく使用されるようになりました。その筆頭となる HDL 合成およびシミュレーションの言語が Verilog と VHDL です。これらの 2 つの言語は、IEEE 規格として採用されています。

ISE® Design Suite は、さまざまな HDL 合成ツールおよびシミュレーション ツール と共に使用できるように設計されているので、この 1 つのソリューションでプログラマブル ロジックを最初から最後まで設計できます。ISE Design Suite では、ライブラリ、ネットリストリーダ、ネットリストライタと共に配置配線ツールが提供されており、Windows および Linux 上の HDL 設計環境に統合して使用できます。

業界標準規格への準拠

ザイリンクスは、関連する業界の標準規格に準拠しています。

- ・ [シミュレーション フロー](#)
- ・ [ザイリンクス シミュレーション フローでサポートされる標準規格](#)
- ・ [サポートされるシミュレータおよび OS](#)
- ・ [ザイリンクス ライブラリ](#)

シミュレーション フロー

ソース ファイルをコンパイルする際は、次の表に示す規則に従ってください。

コンパイル順の規則

HDL 言語	依存性	コンパイル順
Verilog	依存しない	任意の順序
VHDL	依存する	下位から上位

次の事項に従うことをお勧めします。

- ・ HDL ネットリストの前にテストベンチを指定する。
- ・ テストベンチ ファイルで、メイン モジュール名を **testbench** とする。

この名前は、ISE で使用されるデフォルト名です。この名前を使用すると、シミュレーションを ISE から実行するためにオプションで名前を変更する必要はありません。

ザイリンクス シミュレーション フローでサポートされる標準規格

説明	バージョン
VHDL	IEEE-STD-1076-2000
VITAL モデリング規格	IEEE-STD-1076.4-2000
Verilog	IEEE-STD-1364-2001
標準遅延フォーマット (SDF)	OVI 3.0

ザイリンクス HDL ネットリスタでは、IEEE-STD-1076-2000 VHDL コードまたは IEEE-STD-1364-2001 Verilog コードが生成されますが、テスト ベンチまたはその他のシミュレーション ファイルの生成には、新しい規格または以前の規格のどちらも使用できます。シミュレータで新しい規格と以前の規格の両方がサポートされる場合は、シミュレーション ファイルで両方の規格を使用できます。コードをコンパイルするときには、シミュレータでファイル生成に使用した規格を必ず指定してください。

ザイリンクスでは、SystemVerilog はサポートしていません。System Verilog のサポート予定については、次の付録にリストされているザイリンクス EDA パートナーにお問い合わせください。

- ・ [ModelSim でのザイリンクス デザインのシミュレーション](#)
- ・ [IES でのザイリンクス デザインのシミュレーション](#)
- ・ [VCS および VCS MX でのザイリンクス デザインのシミュレーション](#)

サポートされるシミュレータおよび OS

シミュレータ	RH Linux	RH Linux-64	SuSe Linux	SuSe Linux-64	Windows XP	Windows XP-64	Windows Vista	Windows Vista-64
ISim	○	○	○	○	○	×	○	×
ModelSim Xilinx Edition III (6.5c)	×	×	×	×	○	○	○	○
ModelSim SE (6.5c)	○	○	○	○	○	○	○	○
ModelSim PE (6.5c)	×	×	×	○	○	○	○	○
ModelSim DE (6.5c)	×	×	×	○	○	○	○	○
Cadence 社 Incisive Enterprise Simulator (IES) (9.2)	○	○	○	○	×	×	×	×
Synopsys 社 VCS および VCS MX (D2009.12)	○	○	○	○	×	×	×	×

ザイリンクスでは、UNIX OS はサポートしていません。

通常は、最新バージョンのシミュレータを使用してください。

ザイリンクスでは、IEEE 標準規格を使用してライブラリおよびシミュレータ ネットリストを開発しているので、ほとんどの VHDL および Verilog シミュレータを使用できます。シミュレータで標準規格がサポートされていることを確認し、シミュレータの正しい設定をシミュレータベンダーにお問い合わせください。

ザイリンクス ライブラリ

ザイリンクスの VHDL ライブラリは、シミュレーションを高速化するため、IEEE-STD-1076 4-2000 VITAL 標準規格に基づいています。VITAL2000 は、IEEE-STD-1076-93 VHDL 言語に基づいています。このため、ザイリンクス ライブラリを 1076-93 としてコンパイルする必要があります。

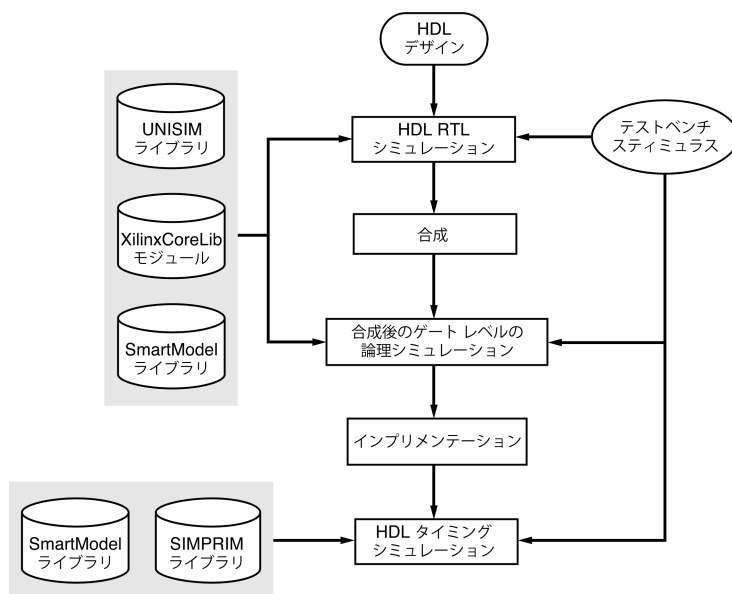
VITAL ライブラリには、タイミング チェックおよびバック アノテーション スタイルの処理が追加で含まれています。UNISIM ライブラリの場合、ユニット遅延の論理シミュレーションではタイミング チェックがオフになります。SIMPRIM のバックアノテーション ライブラリの場合、正確なタイミング シミュレーションを実行するため、デフォルトでこのチェックがオンになります。

HDL デザイン フローのシミュレーション ポイント

ザイリンクスでは、下の「HDL デザイン フローの 5 つのシミュレーション ポイント」の表に示す HDL デザイン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をサポートしています。

次の図に、デザイン フローのシミュレーション ポイントを示します。

HDL デザインの主要なシミュレーション ポイント



X10018

合成およびマップでの問題のデバッグには、NGDBuild 後およびマップ後のシミュレーションを使用できます。

HDL デザイン フローのシミュレーション ポイント

	UNISIM	UniMacro	XilinxCoreLib モデル	SecureIP	SIMPRIM	Standard Delay Format (SDF)
1. レジスタトランスファレベル (RTL)	○	○	○	○	×	×
2. 合成後 (NGDBuild 前) のゲートレベルシミュレーション (オプション)	○	×	×	○	×	×
3. NGDBuild 後 (マップ前) のゲートレベルシミュレーション (オプション)	×	×	×	○	○	×
4. 部分的なタイミング (ブロック遅延) を含むマップ後のシミュレーション (オプション)	×	×	×	○	○	○
5. 配置配線後 (ブロック遅延およびネット遅延) のタイミングシミュレーション	×	×	×	○	○	○

SecureIP の詳細は、「[SecureIP モデルの暗号化手法](#)」を参照してください。

シミュレーション フロー ライブラリ

シミュレーション フローをサポートする際に必要となるライブラリの詳細は、「[VHDL および Verilog のライブラリとモデル](#)」を参照してください。フローおよびライブラリにより、論理シミュレーションとタイミング シミュレーションでの初期化のビヘイビアが機能的に同等になります。

NGDDBuild 前のシミュレーションと NGDDBuild 後のシミュレーションでは、異なるシミュレーション ライブラリが使用されます。

- ・ NGDDBuild 前のシミュレーションでは、デザインはユニファイド ライブラリ コンポーネントを含む UNISIM ネットリストで論理デザインとして表現されます。
- ・ NGDDBuild 後のシミュレーションでは、デザインは SIMPRIM を含むネットリストで物理デザインとして表現されます。

ここでは、次の 2 点に注意する必要があります。

- ・ インプリメンテーションの前後のシミュレーションで異なるシミュレーション ライブラリを指定する必要がある。
- ・ インプリメンテーションの前後のネットリストに含まれるゲートレベルのセル数は異なる。

VHDL の標準遅延フォーマット (SDF) ファイル

VHDL では、次を指定する必要があります。

- ・ SDF ファイルのディレクトリ
- ・ タイミング シミュレーション中にアノテートするインスタンス

この指定方法は、シミュレータによって異なります。通常は、コマンドラインまたはプログラム オプションを使用して SDF ファイルを読み込みます。SDF ファイルのアノテート方法は、シミュレータのマニュアルを参照してください。

Verilog の標準遅延フォーマット (SDF) ファイル

Verilog では、シミュレーションのネットリストに Verilog のシステム タスク `$sdf_annotate` が含まれており、読み込む SDF ファイル名が示されています。

- ・ シミュレータで `$sdf_annotate` システム タスクがサポートされる場合は、Verilog のシミュレーション ネットリストをコンパイルするときに、SDF ファイルが自動的に読み込まれます。
- ・ シミュレータで `$sdf_annotate` システム タスクがサポートされない場合は、ゲートレベルのネットリストにタイミング値が適用されるようするため、SDF ファイルがアノテートされるように手動で指定する必要があります。

レジスタトランスファ レベル (RTL)

レジスタトランスファ レベル (RTL) には、次を含めることができます。

- ・ RTL コード
- ・ インスタンス化された UNISIM ライブラリ コンポーネント
- ・ インスタンス化された UniMacro コンポーネント
- ・ XilinxCoreLib および UNISIM ゲートレベル モデル (CORE Generator™ ソフトウェア)
- ・ SecureIP

RTL レベル (ビヘイビア) のシミュレーションでは、システム レベルまたはチップ レベルの記述を検証、シミュレーションできます。このシミュレーションでは通常、コードの構文を確認し、コードが設計どおりに機能しているかを検証します。この段階ではタイミング情報はないので、レース コンディションを回避するため、シミュレーションをユニット遅延モードで実行する必要があります。

デザインに UNISIM または CORE Generator ソフトウェアのコンポーネントがインスタンス化されていない場合、RTL シミュレーションはアーキテクチャ特有ではありません。これらのインスタンス化をサポートするため、ザイリンクスでは UNISIM および XilinxCoreLib ライブラリを提供しています。次のような場合は、CORE Generator ソフトウェアのコンポーネントを使用できます。

- ・ 合成ツールのモジュール生成機能を使用しない場合
- ・ デザインで大型の構造が必要な場合

設計の初期段階では、コードをビヘイビアにしておくことをお勧めします。また、必要のない限り、特定のコンポーネントをインスタンス化しないようにしてください。これにより、次が可能になります。

- ・ コードが読みやすくなる。
- ・ シミュレーションが高速で簡単になる
- ・ コードの移植 (別のデバイス ファミリーへの移行) が可能になる。
- ・ コードの再利用 (将来のデザインで同じコードを使用) が可能になる。

コンポーネントが推論不可能な場合には、コンポーネントをインスタンス化する必要があります。

合成後 (NGDBuild 前) のゲート レベル シミュレーション

合成後 (NGDBuild 前) のゲート レベル シミュレーションでは、オプションで次のいずれか 1 つを含めることができます。

- ・ UNISIM 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

ほとんどの合成ツールでは、合成後の HDL ネットリストを書き出すことができます。VHDL または Verilog ネットリストが UNISIM ライブラリ コンポーネントを使用して記述されている場合、このネットリストを使用してデザインをシミュレーションし、合成結果を評価できます。

ただし、ネットリストがベンダー特有のシミュレーション モデルで表現されている場合は、ザイリンクスのツールではこの方法はサポートされません。

NGDBuild 後 (マップ前) のゲート レベル シミュレーション

NGDBuild 後 (マップ前) のゲート レベル シミュレーションには、次をオプションで含めることができます。

- ・ 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

NGDBuild 後 (マップ前) のゲートレベルの論理シミュレーションは、合成ツールの出力をシミュレーションできない場合に使用します。合成ツールで UNISIM 対応の VHDL または Verilog ネットリストを書き出すことができない場合などです。この場合、NGDBuild で生成された NGD ファイルをザイリンクス シミュレーション ネットリスト (NetGen) に入力します。NetGen では、SIMPRIM モデルに基づいた構造シミュレーション ネットリストが生成されます。

NGDBuild 後のシミュレーションでは、合成後のシミュレーションと同様、デザインが正しく合成されたかを検証できます。また、抽象度が低いので、合成前のデザインとの違いを特定できます。合成後 (NGDBuild 前) のシミュレーションと異なり、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) 信号を初期化する必要があります。NGDBuild 後のシミュレーションでの **GSR** および **GTS** 信号に関する詳細は、「[シミュレーションでのグローバル リセットおよびトライステート](#)」を参照してください。

部分的なタイミング (ブロック遅延) を含むマップ後のシミュレーション

部分的なタイミング (ブロック遅延) を含むマップ後のシミュレーションには、オプションで次を含めることができます。

- ・ 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
- ・ 標準遅延フォーマット (SDF) ファイル
- ・ SecureIP

デザインのマップ後にシミュレーションを実行することも可能です。マップ後のシミュレーションは、配置配線前に実行します。このシミュレーションには、デザインのブロック遅延は含まれますが、配線遅延は含まれていないので、シミュレーションの結果は正確ではありません。このシミュレーションは、配置配線後のシミュレーションでエラーが発生した場合のデバッグ ステップとして使用してください。

NGDBuild 後のシミュレーションと同様に、NetGen を使用して構造シミュレーション ネットリストが生成されます。Netlister では、SDF ファイルも生成されます。デザインの遅延は、すべて標準遅延フォーマット (SDF) ファイルに格納されます。ただし、この時点ではデザインで配置配線が実行されていないため、配線遅延は含まれません。NetGen で生成されるほかのネットリストと同様に、グローバル セット/リセット (GSR) とグローバル トライステート (GTS) 信号を考慮する必要があります。NGDBuild 後のシミュレーションでの GSR および GTS 信号に関する詳細は、「[シミュレーションでのグローバル リセットおよびトライステート](#)」を参照してください。

配置配線後 (ブロック遅延およびネット遅延) のタイミング シミュレーション

タイミング (ブロック遅延およびネット遅延) すべてを含む配置配線後のタイミング シミュレーションには、次を含めることができます。

- ・ 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
- ・ 標準遅延フォーマット (SDF) ファイル
- ・ SecureIP

デザインの配置配線プロセスの終了後には、タイミング シミュレーション ネットリストを生成できます。この段階では、実際の回路でのデザインの動作を確認できます。デザイン全体の機能は初期段階で定義されますが、デザインが配置配線されるまではデザインのタイミング情報を正しく計算できません。

NetGen を使用したこれ以前のシミュレーションでは、SIMPRIM モデルに基づく構造ネットリストが生成されますが、配置配線後のネットリストは配置配線された NCD (Native Circuit Description) ファイルから生成されます。このネットリストには、初期化する必要があるグローバル セット/リセット (GSR) およびグローバル トライステート (GTS) ネットが含まれています。GSR および GTS ネットの初期化については、「[シミュレーションでのグローバル リセットおよびトライステート](#)」を参照してください。

タイミング シミュレーションを実行すると、マップ後のシミュレーションと同様に標準遅延フォーマット (SDF) ファイルが生成されます。この段階で生成される SDF ファイルには、デザインのブロック遅延および配線遅延のすべてが含まれます。

ザイリンクスでは、このフローを実行することを強くお勧めします。詳細は、「[タイミング シミュレーションの重要性](#)」を参照してください。

テストベンチを使用したスティミュラスの指定

シミュレーションを実行する前に、デザインにスティミュラスを適用するためのテストベンチを作成する必要があります。

テストベンチは、シミュレーション用に記述された HDL コードで、次を実行します。

- ・ デザインのネットリストをインスタンス化。
- ・ デザインを初期化する。
- ・ スティミュラスを適用してデザインの機能を検証する。

また、シミュレーションの出力をファイルとして保存するか、波形として表示するか、画面に表示するようテストベンチを設定できます。

特定の入力に連続的にスティミュラスを供給する単純なテストベンチから、次を含む複雑な構造のテストベンチまでを作成できます。

- ・ サブルーチンの呼び出し
- ・ 外部ファイルからのスティミュラスの読み出し
- ・ 条件スティミュラス
- ・ その他の複雑な構造

テストベンチには、対話型シミュレーションと比べて次のような利点があります。

- ・ デザイン プロセスでシミュレーションを繰り返し実行できる。
- ・ テスト条件のドキュメントが提供される。

詳細は、「[アプリケーション ノート XAPP199 『Writing Efficient Test Benches』](#)」を参照してください。

テストベンチの作成

次のいずれかの方法を使用して、テストベンチを作成し、デザインをシミュレーションします。

- ISE® Design Suite

ISE Design Suite では、デザイン ファイルに基づいて、適切な構造、ライブラリ リファレンス、およびデザインのインスタンス化を含むテンプレートのテストベンチが作成されます。これにより、デザインの初期段階でのテストベンチの開発が大幅に簡略化されます。

- NetGen

NetGen の **-tb** オプションを使用してテストベンチ ファイルを生成することもできます。

NetGen で作成されるテストベンチ ファイル

HDL 言語の指定	ファイル	ファイル拡張子
VHDL	テストベンチ	.tvhd
Verilog	テスト フィクスチャ	.tv

テストベンチでの推奨事項

次に、テストベンチを作成して実行する際の推奨事項を示します。

- テストベンチ ファイルでは、メインのモジュールまたはエンティティの名前を **testbench** にする。Verilog テストベンチ ファイルでは、常に **timescale** を指定する。
- インスタンス化されたデザインの最上位のインスタンス名を **UUT** にする。
これらの名前は、ISE でシミュレータの起動時にテストベンチを呼び出し、SDF ファイルをアノテートする際にデフォルトで使用する名前と一致しています。
- シミュレーションを既知の値で正しく開始するため、シミュレーション時間 0 でデザインの入力をすべて初期化する。
- SIMPRIM 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) のパルスを考慮して、100ns 後からステイムラスを適用する。ただし、クロックのソースは GSR が解放される前に開始する必要があります。詳細は、「[シミュレーションでのグローバル リセットおよびトライステート](#)」を参照してください。

VHDL および Verilog のライブラリとモデル

このセクションでは、VHDL および Verilog のライブラリとモデルについて次の事項を説明します。

- [シミュレーション ポイントで必要なライブラリ](#)
- [シミュレーションで使用されるライブラリ](#)
- [ライブラリ ソース ファイルとコンパイル順](#)

シミュレーション ポイントで必要なライブラリ

上記の 5 つのシミュレーション ポイントでは、次のライブラリが必要です。

- ・ UNISIM
- ・ UniMacro
- ・ CORE Generator™ ソフトウェア (XilinxCoreLib)
- ・ SecureIP
- ・ SIMPRIM

シミュレーション ポイント 1：レジスタトランスファ レベル (RTL)

最初のシミュレーション ポイントであるレジスタトランスファ レベル (RTL) は、レジスタトランスファ レベルでのデザインのビヘイビア記述です。デザインに UNISIM または CORE Generator ソフトウェアのコンポーネントがインスタンシエートされていない場合、RTL シミュレーションはアーキテクチャ特有ではありません。

これらのインスタンシエーションをサポートするため、ザイリンクスでは次のライブラリを提供しています。

- ・ UNISIM
- ・ UniMacro
- ・ CORE Generator テクノロジ ビヘイビア XilinxCoreLib
- ・ SecureIP

シミュレーション ポイント 2：合成後 (NGCBuild 前) のゲート レベル シミュレーション

2 つ目のシミュレーション ポイントは、合成後 (NGCBuild 前) のゲート レベル シミュレーションです。

HDL ネットリストを書き出す際、合成ツールで UNISIM プリミティブが使用されます。それ以外の場合、合成ベンダーにより独自の合成後のシミュレーション ライブラリが提供されます。デザインに合成ツールでブラックボックスとして処理される IP が含まれる場合は、NetGen の前に NGCBuild を実行する必要があります。NGCBuild は、NGC ファイルおよび EDIF ファイルをすべて 1 つの NGC ファイルに統合します。このファイルを使用して、NetGen を実行します。NGCBuild の実行方法については、『[コマンドライン ツール ユーザー ガイド](#)』の「NGCBuild」の章を参照してください。

ザイリンクスでは次のライブラリを提供しています。

- ・ UNISIM
- ・ UniMacro
- ・ SecureIP

シミュレーション ポイント 3：NGCBuild 後 (マップ前) のゲート レベル シミュレーション

3 つ目のシミュレーション ポイントは、NGCBuild 後 (マップ 前) のゲート レベル シミュレーションです。このシミュレーション ポイントでは、SIMPRIM ライブラリおよび SecureIP ライブラリが使用されます。

シミュレーション ポイント 4：部分的なタイミング（ブロック遅延）を含むマップ後のシミュレーション

4 つ目のシミュレーション ポイントは、部分的なタイミング（ブロック遅延）を含むマップ後のシミュレーションです。このシミュレーション ポイントでは、SIMPRIM ライブラリおよび SecureIP ライブラリが使用されます。

シミュレーション ポイント 5：配置配線後（ブロック遅延およびネット遅延）のタイミング シミュレーション

5 つ目のシミュレーション ポイントは、配置配線後（ブロック遅延およびネット遅延）のタイミング シミュレーションです。このシミュレーション ポイントでは、SIMPRIM ライブラリおよび SecureIP ライブラリが使用されます。

シミュレーションで使用するライブラリ

各シミュレーション ポイントに必要なライブラリの一覧

シミュレーション ポイント	必要なライブラリのコンパイル順
シミュレーション ポイント 1 レジスタトランスファ レベル (RTL)	UNISIM UniMacro XilinxCoreLib SecureIP
シミュレーション ポイント 2 合成後 (NGDBuild 前) のゲートレベル シミュレーション	UNISIM UniMacro SecureIP
シミュレーション ポイント 3 NGDBuild 後 (マップ前) のゲートレベル シミュレーション	SIMPRIM
シミュレーション ポイント 4 部分的なタイミング（ブロック遅延）を含むマップ後のシミュレーション	SIMPRIM SecureIP
シミュレーション ポイント 5 配置配線後（ブロック遅延およびネット遅延）のタイミング シミュレーション	SIMPRIM SecureIP

ライブラリ ソース ファイルとコンパイル順

ライブラリのコンパイルには、Compplib を使用することをお勧めします。

VITAL VHDL ソース ファイルではコンパイル順が必要です。

シミュレーション ライブラリ VITAL VHDL ソース ファイルの
ディレクトリ (Linux)

ライブラリ	ソース ファイルのディレクトリ
UNISIM Spartan®-3 Spartan-3E Virtex®-4	\$XILINX/vhdl/src/unisims \$XILINX/vhdl/src/unimacro
UNISIM 9500 CoolRunner™ XPLA3 CoolRunner-II	\$XILINX/vhdl/src/unisims
XilinxCoreLib FPGA ファミリの み	\$XILINX/vhdl/src/XilinxCoreLib
SecureIP Virtex-4 Virtex-5 Virtex-6 Spartan-6	\$XILINX/secureip/<simulator> /
SIMPRIM (全ザイリンクス テク ノロジー)	\$XILINX/vhdl/src/simprims

シミュレーション ライブラリ VITAL VHDL ソース ファイルの
ディレクトリ (Windows)

ライブラリ	ソース ファイルのディレクトリ
UNISIM Spartan-3 Spartan-3E Virtex-4	%XILINX%\vhdl\src\unisims %%XILINX%\vhdl\src\unimacro
UNISIM 9500 CoolRunner XPLA3 CoolRunner-II	%XILINX%\vhdl\src\unisims
XilinxCoreLib FPGA ファミリの み	%XILINX%\vhdl\src\XilinxCoreLib
SecureIP Virtex-4 Virtex-6 Spartan-6	%XILINX%\secureip\<simulator> \

ライブラリ	ソース ファイルのディレクトリ
Virtex-5	
SIMPRIM (全ザイリンクス テクノロジ)	%XILINX%\vhdl\src\simprims

シミュレーション ライブラリ VITAL VHDL のコンパイル順

ライブラリ	コンパイル順
UNISIM	<ul style="list-style-type: none"> · unisim_VCOMP.vhd · unisim_VPKG.vhd · primitive/vhdl_analyze_order · unimacro_VCOMP.vhd · UniMacro ディレクトリにあるすべてのファイル
<ul style="list-style-type: none"> · UNISIM 9500 · CoolRunner XPLA3 · CoolRunner-II 	<ul style="list-style-type: none"> · unisim_VCOMP.vhd · unisim_VPKG.vhd · primitive/vhdl_analyze_order
XilinxCoreLib FPGA ファミリのみのみ	UNISIM ライブラリ ソース ファイルのディレクトリにある vhdl_analyze_order を参照
SecureIP	<p>論理シミュレーション</p> <ul style="list-style-type: none"> · UNISIM ライブラリ · <simulator>_secureip_cell.list.f · \$XILINX/vhdl/src/unisims/secureip/other/vhdl_analyze_order <p>タイミング シミュレーション</p> <ul style="list-style-type: none"> · SIMPRIM ライブラリ · <simulator>_secureip_cell.list.f · \$XILINX/vhdl/src/simprims/secureip/other/vhdl_analyze_order または \$XILINX/vhdl/src/simprims/secureip/modelsim/vhdl_analyze_order (ModelSim のみ)
SIMPRIM (全ザイリンクス テクノロジ)	<ul style="list-style-type: none"> · simprim_Vcomponents.vhd または simprim_Vcomponents_ModelSim.vhd (ModelSim only) · simprim_Vcomponents.vhd または simprim_Vpackage_mti.vhd (ModelSim のみ) · primitive/other/vhdl_analyze_order · primitive/modelsim/vhdl_analyze_order

シミュレーション ライブラリ Verilog ソース ファイル (Linux)

ライブラリ	ソース ファイルのディレクトリ
<ul style="list-style-type: none"> UNISIM Spartan-3 Spartan-3E Virtex-4 	\$XILINX/verilog/src/unisims \$XILINX/verilog/src/unimacro
<ul style="list-style-type: none"> UNISIM 9500 CoolRunner XPLA3 CoolRunner-II 	\$XILINX/verilog/src/uni9000
XilinxCoreLib FPGA ファミリのみ	UNISIM ライブラリ \$XILINX/verilog /src/XilinxCoreLib
<ul style="list-style-type: none"> SecureIP Virtex-4 Virtex-5 Virtex-6 Spartan-6 	UNISIM ライブラリ <simulator>_secureip_cell.list.f
SIMPRIM (全ザイリンクス テクノロジー)	\$XILINX/verilog/src/simprims

シミュレーション ライブラリ Verilog ソース ファイル (Windows)

ライブラリ	ソース ファイルのディレクトリ
<ul style="list-style-type: none"> UNISIM Spartan-3 Spartan-3E Virtex-4 	%XILINX%\verilog\src\unisims %XILINX%\verilog\src\unimacro
<ul style="list-style-type: none"> UNISIM 9500 CoolRunner XPLA3 CoolRunner-II 	%XILINX%\verilog\src\uni9000
XilinxCoreLib FPGA ファミリのみ	UNISIM ライブラリ %XILINX%\verilog\src\XilinxCoreLib
<ul style="list-style-type: none"> SecureIP Virtex-4 Virtex-5 Virtex-6 Spartan-6 	UNISIM ライブラリ <simulator>_secureip_cell.list.f

ライブラリ	ソース ファイルのディレクトリ
SIMPRIM (全ザイリンクス テクノロジー)	%XILINX%\verilog\src\simprims

Verilog ライブラリでは、特定のコンパイル順はありません。

シミュレーション ライブラリ

XST では、次のシミュレーション ライブラリがサポートされています。

- ・ UNISIM ライブラリ
- ・ VHDL UNISIM ライブラリ
- ・ Verilog UNISIM ライブラリ
- ・ UniMacro ライブラリ
- ・ VHDL UniMacro ライブラリ
- ・ Verilog UniMacro ライブラリ
- ・ CORE Generator™ ソフトウェア XilinxCoreLib ライブラリ
- ・ SIMPRIM ライブラリ
- ・ SecureIP ライブラリ
- ・ VHDL SecureIP ライブラリ
- ・ Verilog SecureIP ライブラリ
- ・ ザイリンクス シミュレーション ライブラリ (Compplib)

UNISIM ライブラリ

UNISIM ライブラリは、論理シミュレーションおよび合成で使用されます。このライブラリには、次が含まれています。

- ・ ほとんどの合成ツールで推論されるザイリンクス ユニファイド ライブラリのプリミティブ
- ・ 次のようなよくインスタンスシートされるプリミティブ
 - **DCM**
 - **BUFG**
 - **MGT**

デザインのファンクションは、次の場合を除き、ビヘイビア RTL コードを使用して推論させることをお勧めします。

- ・ 合成ツールでコンポーネントが推論されない場合
- ・ ファンクションのマッピングおよび配置を手動で制御する場合

VHDL UNISIM ライブラリ

VHDL の UNISIM ライブラリは、次の 4 個のファイルに分けられています。

- ・ コンポーネント宣言 (unisim_VCOMP.vhd)
- ・ パッケージ ファイル (unisim_VPKG.vhd)
- ・ エンティティおよびアーキテクチャ宣言 (unisim_VITAL.vhd)

全ザイリンクス デバイス ファミリのプリミティブは、すべてこれらのファイルで指定されています。これらのプリミティブを使用するには、次の 2 行を各ファイルの最初に追加します。

```
Library UNISIM;  
use UNISIM.vcomponents.all;
```

Verilog UNISIM ライブラリ

Verilog では、各ライブラリ コンポーネントが個別のファイルで指定されます。これは、**-y** ライブラリ仕様オプションを使用して自動的にライブラリを拡張することを可能にするためです。Verilog のモジュール名およびファイル名は、すべて大文字です。たとえば、モジュール **BUFG** は **BUFG.v**、モジュール **IBUF** は **IBUF.v** となります。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もすべて大文字で記述する必要があります。

コンパイル済みのライブラリを使用する場合は、適切な指示子を使用してコンパイル済みライブラリを指定します。たとえば、ModelSim では次のように指定します。

```
-L unisims_ver
```

UniMacro ライブラリ

UniMacro ライブラリには、次のような特徴があります。

- ・ 論理シミュレーションおよび合成のみで使用されます。
- ・ 複雑なザイリンクス プリミティブのインスタンス化を支援します。
- ・ UNISIM ライブラリのプリミティブを抽象化したものです。合成ツールで自動的に基になるプリミティブに展開されます。

詳細は、[ライブラリガイド](#)を参照してください。

VHDL UniMacro ライブラリ

これらのマクロを使用するには、UNISIM 宣言に加え、次の 2 行を各ファイルの最初に追加します。

```
Library UNIMACRO;  
use UNIMACRO.vcomponents.all
```

Verilog UniMacro ライブラリ

Verilog では、各マクロ コンポーネントが個別のファイルで指定されます。これは、**-y** ライブラリ仕様オプションを使用して自動的にライブラリを拡張することを可能にするためです。Verilog のモジュール名およびファイル名は、すべて大文字です。Verilog では大文字/小文字が区別されるので、UniMacro のインスタンス化もすべて大文字で記述する必要があります。

コンパイル済みのライブラリを使用する場合は、適切な指示子を使用してコンパイル済みライブラリを指定します。たとえば、ModelSim では次のように指定します。

```
-L unimacro_ver
```

CORE Generator ソフトウェア XilinxCoreLib ライブラリ

ザイリンクスの CORE Generator ソフトウェアは、次のような高度なモジュール (IP) を生成するためのグラフィカル デザイン ツールです。

- ・ FIR フィルタ
- ・ FIFO
- ・ CAM
- ・ その他の高度な IP

モジュールをカスタマイズおよび最適化することにより、次のようなザイリンクス FPGA デバイスのアーキテクチャ機能を最大限に活用できます。

- ・ ブロック乗算器
- ・ SRL
- ・ 高速キャリー ロジック
- ・ オンチップのシングル ポート RAM
- ・ オンチップのデュアル ポート RAM

また、適切な HDL モデルを出力として選択することにより、HDL デザインに統合することもできます。

CORE Generator ソフトウェアの HDL ライブラリ モデルは、RTL シミュレーションで使用されます。

SIMPRIM ライブラリ

SIMPRIM ライブラリは、次のシミュレーションで使用されます。

- ・ NGDBuild 後のシミュレーション (ゲートレベルの論理シミュレーション)
- ・ マップ後のシミュレーション (部分的なタイミング シミュレーション)
- ・ 配置配線後のシミュレーション (完全なタイミング シミュレーション)

SIMPRIM ライブラリは、アーキテクチャに依存しません。

SecureIP ライブラリ

ハード IP ブロックは ISim で完全にサポートされており、特別な設定は必要ありません。詳細は、『ISim ユーザー ガイド』を参照してください。ザイリンクスでは、Verilog LRM - IEEE 標準規格 1364-2005 で規定されている最新の暗号化手法を利用しています。PowerPC プロセッサ、MGT、PCIe® などのハード IP の Virtex®-4 および Virtex-5 デバイス シミュレーション モデルでこの手法が使用されます。コンピュータに適切なバージョンのシミュレータが存在すれば、すべて Complib で自動的に処理されます。Verilog でこの手法を使用してシミュレーションを実行する場合、SecureIP ライブラリを参照する必要があります。ほとんどのシミュレータでは、このライブラリを **-L** オプションを使用して使用できます (**-L secureip** など)。シミュレータで使用するオプションについては、シミュレータのマニュアルを参照してください。

メモ： SecureIP を使用する場合は、**-Ica** オプションを使用してください。

次の表に、これらのライブラリをシミュレータで使用する際の要件を示します。

SecureIP ライブラリを使用する際の考慮事項

シミュレータ	ベンダー	要件
ModelSim SE ModelSim PE ModelSim DE Questa	Mentor Graphics	デザイン入力に VHDL を使用する場合、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。
IUS	Cadence	輸出規制法ライセンスが必要です。
VCS	Synopsys	SecureIP を含むデザインをシミュレーションする場合は、VCS コマンドに <code>-lca</code> オプションを使用する必要があります。

VHDL SecureIP ライブラリ

デザイン入力に VHDL を使用している場合は、ハード IP をシミュレーションするのに混合言語ライセンス（または Mentor Graphics 社の製品では SecureIP OP）が必要です。混合言語シミュレーション オプションの価格などについては、ベンダーにお問い合わせください。

SecureIP を使用するには、次の 2 行を各ファイルの最初に追加します。

```
Library UNISIM;
use UNISIM.vcomponents.all;
```

Verilog SecureIP ライブラリ

シミュレータで `-f` オプションを使用すると、コンパイル時にこれらのライブラリを使用できます。たとえば、VCS では次のように指定します。

```
vcs -lca -f $XILINX/secureip/vcs/vcs_secureip_cell.list.f \
-y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/xilinxcorelib \
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \
-Mupdate -R <testfixture> .v <design> .y
```

コンパイル済みのライブラリを使用する場合は、適切な指示子を使用してコンパイル済みライブラリを指定します。たとえば、ModelSim では次のように指定します。

```
-L secureip
```

ザイリンクス シミュレーション ライブラリ (Compplib)

ModelSim XE (Xilinx Edition) または ISim では使用しないでください。

論理シミュレーションを実行する前に、Compplib を使用してザイリンクス シミュレーション ライブラリを使用するシミュレータ用にコンパイルする必要があります。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

シミュレーション ランタイムの短縮

ザイリンクス シミュレーション モデルには、シミュレーションのランタイムを短縮するオプションのジェネリック/パラメータ **SIM_MODE** があります。**SIM_MODE** には、次の 2 つの設定があります。

- ・ **SIM_MODE = "SAFE"**
- ・ **SIM_MODE = "FAST"**

この設定は、プリミティブの一部の機能に対するシミュレーション サポートに影響します。次の UNISIM プリミティブでサポートされています。

- ・ Virtex®-5 デバイスのブロック RAM
- ・ Virtex-5 デバイスの FIFO
- ・ Virtex-5 デバイスの DSP ブロック

次の表に、FAST モードを使用した場合にサポートされない機能を示します。

FAST モードでサポートされない Virtex-5 デバイスのブロック RAM の機能

機能	説明
パラメータの有効性のチェック	ジェネリック/パラメータが使用されているプリミティブで有効であるかどうかのチェック
カスケード機能	複数のブロック RAM のカスケード接続
ECC 機能	エラーのチェックおよび修正
メモリ競合のチェック	同じアドレスに対して同時にデータの書き込みおよび読み出しを実行していないかどうかをチェック

FAST モードでサポートされない Virtex-5 デバイスの FIFO の機能

機能	説明
パラメータ チェック	ジェネリック/パラメータが使用されているプリミティブで有効であるかどうかのチェック
リセットのデザイン ルール チェック	リセット時、正しい数のリセット パルスが適用されているかどうかはモデルでチェックされません。
ECC 機能	エラーのチェックおよび修正

FAST モードでサポートされない Virtex-5 デバイスの DSP ブロックの機能

機能	説明
DRC チェック (opmode および alumode)	削除された opmode および alumode 設定に対するさまざまなチェック

シミュレーションを完了し、シミュレーション モデルがハードウェアで予測どおりに機能することを確実にするには、**SAFE** モードを使用してください。

SIM_MODE は、UNISIM の RTL シミュレーション モデルにのみ適用されます。SIMPRIM のゲートシミュレーション モデルではサポートされません。SIMPRIM ベースのシミュレーションでは、すべてのチェックが実行され、シミュレーション ランタイムが長くなります。

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、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。次の内容が含まれています。

- ・ [JTAG シミュレーション](#)
- ・ [SelectMAP シミュレーション](#)
- ・ [Spartan®-3AN インシステム フラッシュ シミュレーション](#)

JTAG シミュレーション

次のデバイスでは、**BSCAN** コンポーネントのシミュレーションがサポートされています。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Virtex-6
- ・ Spartan®-3A
- ・ Spartan®-6

このシミュレーションでは、JTAG ポートと一部の JTAG 操作コマンドの相互作用がサポートされています。スキャン チェーンへのインターフェイスを含む JTAG インターフェイスは、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、次を実行します。

1. **BSCAN_VIRTEX4**、**BSCAN_VIRTEX5**、**BSCAN_VIRTEX6**、**BSCAN_SPARTAN3A**、または **BSCAN_SPARTAN6** コンポーネントをインスタンスシートし、デザインに接続します。
2. **JTAG_SIM_VIRTEX4**、**JTAG_SIM_VIRTEX5**、**JTAG_SIM_VIRTEX6**、**JTAG_SIM_SPARTAN3A**、または **JTAG_SIM_SPARTAN6** コンポーネントを、デザインではなくテストベンチにインスタンスシートします。

これが次のものになります。

- ・ **TDI**、**TDO**、および **TCK** などの外部 JTAG 信号へのインターフェイス
- ・ **BSCAN** コンポーネントへの通信チャネル

コンポーネント間の通信は、VHDL の VPKG パッケージ ファイルまたは Verilog の **glbl** グローバル モジュールで発生するため、**JTAG_SIM_<device>** コンポーネントとデザイン間、または **JTAG_SIM_<device>** コンポーネントと **BSCAN_<device>** シンボル間には、間接的な接続は不要です。

テストベンチの **JTAG_SIM_<device>** コンポーネントからステイミュラスを駆動および表示すると、JTAG/BSCAN ファンクションの動作を確認できます。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、ISE の言語テンプレートおよびデバイスのライブラリ ガイドを参照してください。

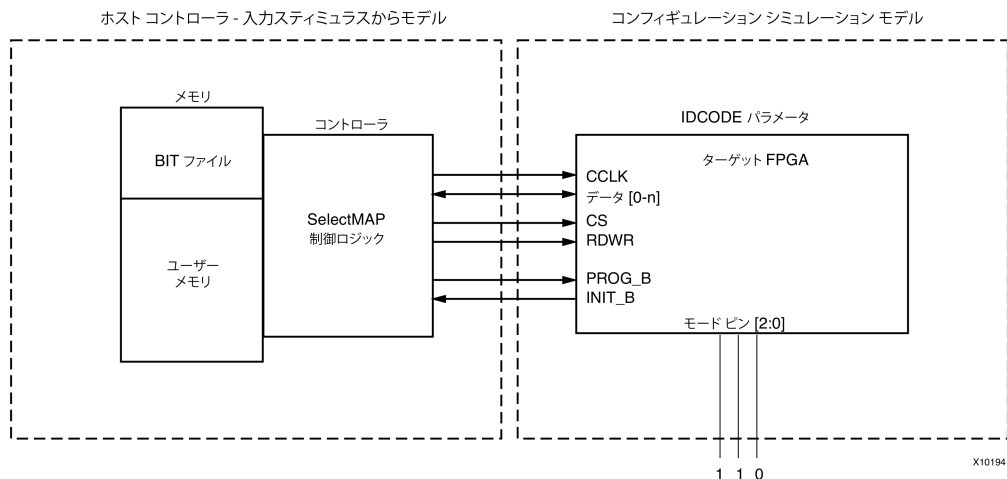
SelectMAP シミュレーション

コンフィギュレーション シミュレーション モデル (SIM_CONFIG_xx) をインスタンス化してテンプレート共に使用すると、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でき、**DONE** ピンが High になることを確認できます。このモデルは、コンフィギュレーション インターフェイスにおけるデバイスのスティミュラスに対する動作を示します。サポートされるインターフェイスおよびデバイスのリストは、次の表を参照してください。モデルは、制御信号の動作および BIT ファイルのダウンロードを処理するように設定されています。また、CRC、IDCODE、ステータスレジスタなどの内部レジスタ設定も含まれます。同期ワードの入力状況、スタートアップシーケンスの進行状況をモニタできます。下に示す図に、ハードウェアとシミュレーション環境の構成を示します。コンフィギュレーション プロセスは、各デバイス ファミリ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 インターフェイス、コンフィギュレーション シーケンスなどに関する情報が含まれます。

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
Virtex®-6	あり	あり	なし	なし
Virtex®-5	あり	なし	なし	なし
Spartan®-6	あり	あり	なし	なし
Spartan®-3A	あり	なし	なし	なし

モデル間の通信



システム レベルの記述

このモデルはデバイス全体をシミュレーションするので、システムレベルで使用できます。プロセッサを使用してコンフィギュレーション ロジックを制御するアプリケーションでは、このモデルを利用して適切な配線、制御信号の処理、データ入力のアライメントを確実にできます。**CS** (SelectMAP のチップ セレクト) または **CLK** 信号でデータの読み込みを制御するアプリケーションでは、データが正しく揃えられているかどうかをテストできます。SelectMAP ABORT またはリードバックを実行する必要があるシステムでも、このモデルを利用できます。

このモデルに関する ZIP ファイルを ftp://ftp.xilinx.com/pub/documentation/misc/config_test_bench.zip からダウンロードできます。この 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あります。フル コンフィギュレーション、ABORT、IDCODE およびステータスレジスタのリードバックなどの機能も含まれます。シミュレーションするホストシステムに、ファイル供給方法および制御信号の制御方法が必要です。これらの制御システムは、[コンフィギュレーション ユーザー ガイド](#)に示されているように設計する必要があります。この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、BIT ファイルにデバイスを読み込むコンフィギュレーション プロセスにおけるデバイス内の変化も示します。BIT ファイルのダウンロード中、各コマンドが処理され、レジスタ設定を変更して、ハードウェアの変化を反映します。CRC 値を蓄積する CRC レジスタもモニターでき、またコンフィギュレーションの異なる段階でデバイスの進行状況を示すステータスレジスタも示されます。

モデルのデバッグ

このモデルでは、正しいコンフィギュレーション例が提供されています。このコンフィギュレーション例を利用すると、問題が発生した場合のデバッグ処理に役立ちます。ステータスレジスタにはデバイスの現在のステートに関する情報が含まれるので、デバッグで有益です。このレジスタの値は、iMPACT を使用して JTAG を介してデバイスから読み出すことができます。ボード上で問題が発生した場合は、まずステータスレジスタの値を確認してください。

ステータスレジスタの値を確認したら、シミュレーションに対応させ、エラー発生時のコンフィギュレーション段階を判断します。たとえば、GHIGH ビットはデータ読み込み後に High になりますが、このビットが Low の場合はデータの読み込みが完了していないことを示します。BitGen で設定される **GTS**、**GWE**、および **DONE** 信号は、スタートアップ シーケンスで解放されますが、これらをモニターできます。

エラーを発生させることも可能です。データの読み込みを停止し、再開したときに問題が発生した場合、CRC ロジックで検出されます。BIT ファイルに手動で挿入したビットの反転も検出され、エラーがデバイスで同様に処理されます。

サポートされる機能

各デバイスの[コンフィギュレーション ユーザー ガイド](#)に、各コンフィギュレーション インターフェイスでサポートされる通信方法が説明されています。このガイドでは、「すべてのデバイスファミリのモデルでサポートされるスレーブ SelectMAP およびシリアル機能」の表および「Virtex-5 デバイスのスレーブ SelectMAP 機能のモデルによるサポート」の表に、[コンフィギュレーション ユーザー ガイド](#)で説明されている項目がモデルでサポートされているかどうかを示します。

モデルでは、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ていません。また、CRC 値は算出されますが、コンフィギュレーション データは保存されません。リードバックは、デバイスに有効なコマンド シーケンスが供給され、信号が適切に処理されることを確実にするために、特定のレジスタ上でのみ実行可能です。このモデルでは、リードバック データ ファイルは生成できません。

すべてのデバイス ファミリのモデルでサポートされるスレーブ SelectMAP およびシリアル機能

機能	サポートあり
マスタ モード	なし
デイジー チェーン接続 - Spartan-3E デバイス および Spartan-3A デバイスのスレーブ パラレル デイジー チェーン	あり
デイジー チェーン接続 - 任意のザイリンクス FPGA ファミリを使用したスレーブ パラレル デイ ジー チェーン	なし
SelectMAP データの読み込み	あり
継続的名 SelectMAP データの読み込み	あり
断続的な SelectMAP データの読み出し	あり
SelectMAP の ABORT	あり
SelectMAP のリコンフィギュレーション	なし
SelectMAP のデータ順	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 ション中の CRC チェック	なし
BitGen による DONE_cycle 、 GTS_cycle 、 GWE_cycle の変更	あり
BitGen によるその他のオプションのデフォルト 値からの変更	DONE 、 GTS 、 GWE の解放位置の変更は、タイ ミングのみに影響
DCM でシミュレーションを実行できるようにな るまで待機	あり (Spartan®-6 のみ)

Virtex-5 デバイスのスレーブ SelectMAP 機能のモデルによるサポート

機能	サポートあり
マスタ モード	なし
単一デバイスの SelectMAP コンフィギュレー ション	あり
複数デバイスの SelectMAP コンフィギュレー ション	あり
パラレル デイジー チェーン	あり
ギャング SelectMAP	あり
SelectMAP データの読み込み	あり

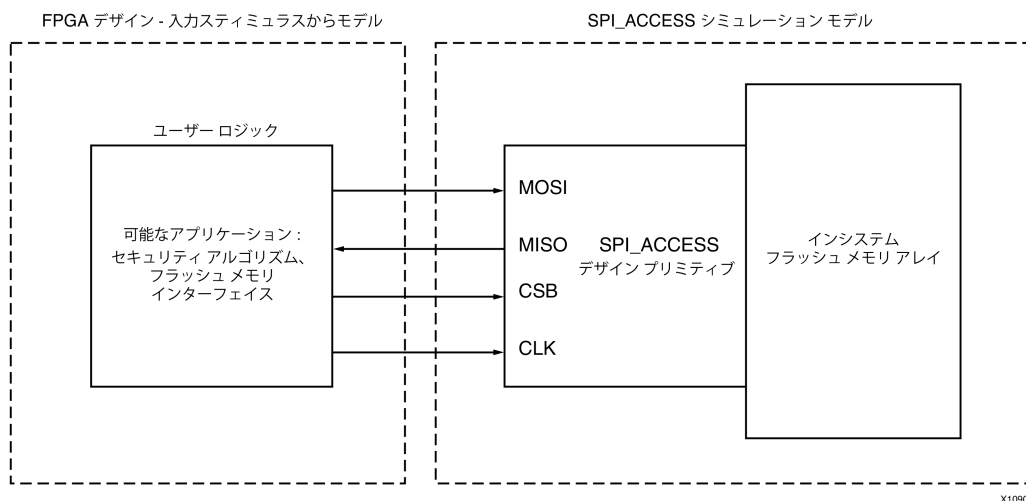
機能	サポートあり
SelectMAP の ABORT	あり
SelectMAP のリコンフィギュレーション	なし
SelectMAP のデータ順	あり
リードバックおよびコンフィギュレーションの検証	IDCODE とステータス レジスタのみリードバック可能
リコンフィギュレーションおよびマルチブート	なし
リードバック CRC	なし
BitGen による DONE_cycle 、 GTS_cycle 、 GWE_cycle の変更	DONE 、 GTS 、 GWE の解放位置の変更は、タイミングのみに影響
BitGen によるその他のオプションのデフォルト値からの変更	なし

Spartan-3AN インシステム フラッシュ シミュレーション

Spartan-3AN デバイスには、初期コンフィギュレーション、マルチブート、ユーザー メモリ、またはこれらの組み合わせに使用可能な内部メモリ機能があります。デバイスのコンフィギュレーション後にこのメモリにアクセスするには、FPGA デバイスに読み込まれているアプリケーションで **SPI_ACCESS** という特別なデザイン プリミティブを使用する必要があります。ISF (インシステム フラッシュ) メモリに対するデータ アクセスは、SPI (シリアル ペリフェラル インターフェイス) プロトコルを使用して実行されます。Spartan-3AN デバイスにも、**SPI_ACCESS** プリミティブにも、専用 SPI マスタ コントローラは含まれていません。制御ロジックは、FPGA デバイスのプログラマブル ロジック リソースを使用してインプリメントします。**SPI_ACCESS** プリミティブは、FPGA デバイス アプリケーションをインシステム フラッシュ メモリ アレイに接続します。シミュレーション モデルを使用すると、このインターフェイスの動作をシミュレーションでテストできます。このインターフェイスは、4 つの標準 SPI 接続で構成されています。

- ・ **MOSI** (マスタ出力スレーブ入力)
- ・ **MISO** (マスタ入力スレーブ出力)
- ・ **CLK** (クロック)
- ・ **CSB** (アクティブ Low のチップ セレクト)

Spartan-3AN SPI_ACCESS の ISF メモリへの接続



SPI_ACCESS でサポートされるコマンド

SPI_ACCESS シミュレーション モデルでは、ハードウェアで実行可能なコマンドの一部のみがサポートされます。次の表に、モデルでサポートされるコマンドを示します。これらのコマンドは、モデルおよびシリコン上で機能することがテストおよび検証されています。ここにリストされている以外のコマンドは、シミュレーション モデルではサポートされていませんが、ハードウェアでは予測どおりに機能するので、このガイドで説明しています。すべてのコマンドの詳細な説明は、[『Spartan-3AN FPGA In-System Flash User Guide』](#)を参照してください。

SPI_ACCESS でサポートされるコマンド

コマンド	説明	16 進コマンド コード
Fast Read (高速読み出し)	CLK の周波数が 33MHz より高い場合、連続データを大きなブロックで読み出します。	0x0B
Random Read (ランダム読み出し)	ランダムに指定したロケーションからバイトを読み出します。すべての読み出しは 33MHz 以下で実行されます。	0x03
Status Register Read (ステータスレジスタの読み出し)	プログラム コマンド、比較の結果、保護、アドレス指定モードなどの Ready/Busy をチェックします。	0xD7
Information Read (情報の読み出し)	JEDEC の製造業者およびデバイス ID を読み出します。	0x9F
Security Register Read (セキュリティレジスタの読み出し)	セキュリティレジスタの内容を読み出します。	0x77
Security Register Program (セキュリティレジスタのプログラム)	セキュリティレジスタのユーザー定義フィールドをプログラム	0x9B
Buffer Write (バッファの書き込み)	SRAM ページ バッファにデータを書き込み、完了したら Buffer to Page Program コマンドを使用して ISF メモリに転送	Buffer1 - 0x84 Buffer2 - 0x87
Buffer to Page Program with Built-in Erase (ビルトインの消去を含むバッファからのページのプログラム)	まず選択したメモリ ページを消去し、指定のバッファからのデータでメモリをプログラムします。	Buffer1 - 0x83 Buffer2 - 0x86
Buffer to Page Program without Built-in Erase (ビルトインの消去なしのバッファからのページのプログラム)	前に消去したページを指定のバッファからのデータでプログラムします。	Buffer1 - 0x88 Buffer2 - 0x89
Page Program Through Buffer with Erase (消去を含むバッファを介したページのプログラム)	Buffer Write と Buffer to Page Program with Built-in Erase を組み合わせたものです。	Buffer1 - 0x82 Buffer2 - 0x85
Page to Buffer Compare (ページとバッファの比較)	ISF メモリ アレイが正しくプログラムされたかどうかを検証します。	Buffer1 - 0x60 Buffer2 - 0x61
Page to Buffer Transfer (ページからバッファへの転送)	選択した ISF メモリ ページの内容全体を指定した SRAM ページ バッファに転送します。	Buffer1 - 0x53 Buffer2 - 0x55
Sector Erase (セクタの消去)	メイン メモリで保護されていないロックされていないセクタを消去します。	0x7C
Page Erase (ページの消去)	ISE メモリ アレイの 1 ページを消去します。	0x81

SPI_ACCESS メモリの初期化

ISF の初期化に使用するメモリ ファイルは、16 進バイトを ASCII 形式でリストして作成します。行に 1 バイトずつリストしてください。行数は、メモリのサイズにより異なります。このファイルは、ISF メモリ空間を初期化します。

初期化ファイルのメモリ サイズとデバイス上のメモリ サイズが一致しない場合は、ファイルが大きすぎるか小さすぎることを示す警告メッセージが表示されます。

- ・ 初期化ファイルの方が小さい場合は、メモリの残りの部分に 0xFF が挿入されます。
- ・ 初期化ファイルの方が大きい場合は、余ったバイトは使用されません。

次の表に、各デバイスで使用可能なメモリのサイズを示します。

ISF のメモリ サイズ

デバイス	ISF メモリのビット数	使用可能なユーザーメモリ (バイト)	初期化ファイルの行数
3S50AN	1M +	135,168	135,168
3S200AN	4M +	540,672	540,672
3S400AN	4M +	540,672	540,672
3S700AN	8M +	1,081,344	1,081,344
3S1400AN	16M +	2,162,688	2,162,688

SPI_ACCESS の属性

SPI_ACCESS コンポーネントでは、5 つの属性を設定できます。

- ・ **SIM_DEVICE**
- ・ **SIM_USER_ID**
- ・ **SIM_MEM_FILE**
- ・ **SIM_FACTORY_ID**
- ・ **SIM_DELAY_TYPE**

SPI_ACCESS の SIM_DEVICE 属性

使用する Spartan-3AN デバイスを指定します。これにより、SPI フラッシュのサイズが正しく設定されます。**SIM_DEVICE** 属性は必須です。

SPI_ACCESS の SIM_USER_ID 属性

シミュレーションで使用し、セキュリティレジスタのユーザー 定義フィールドを初期化します。ハードウェアでは、いつでも任意の値にプログラムできます。このフィールドは、1 回しかプログラムできません (OTP)。出荷時のデフォルト状態は消去され、すべてのロケーションが 0xFF になります。**SIM_USER_ID** は、Verilog では 512 ビットの reg、VHDL では 512 ビットの bit_vector を使用して、シミュレーションに使用する 16 進数に指定します。ビット 511 がセキュリティレジスタのユーザー部分の最初のビットで、ビット 0 が最後のビットです。

SPI_ACCESS の SIM_MEM_FILE 属性

メモリ初期化ファイルのディレクトリとファイル名を指定します。詳細は、「**SPI_ACCESS** メモリの初期化」を参照してください。

SPI_ACCESS の SIM_FACTORY_ID 属性

シミュレーションでのみ使用し、セキュリティレジスタの一意識別子 (Unique Identifier) 部分の値を設定します。この値は、Information Read コマンドを送信することによりリードバックします。デフォルトの **FACTORY_ID** はすべて 1 です。

シミュレーションでは、**FACTORY_ID** は 1 度しか書き込むことができません。1 以外の値が検出されると、書き込みができなくなります。

ハードウェアでは、このフィールドに各デバイスにファクトリでプログラムされた固有の ID が含まれており、再プログラムまたは消去できません。

SPI_ACCESS の SIM_DELAY_TYPE 属性

チップの遅延をシミュレーションに適切な値に変更するかどうかを指定します。**ACCURATE** に設定した場合は、実際のタイミング仕様 (セクタの消去に 5 秒など) が使用されます。**SCALED** に設定すると、シミュレーションランタイムを短縮するため、小さい遅延値が使用されます。デバイスの動作には影響しません。

SPI_ACCESS の属性

属性	データ型	値	デフォルト	説明
SIM_DEVICE	文字列	3S50AN 3S200AN 3S400AN 3S700AN 3S1400AN	3S1400AN	適切なサイズの SPI メモリが使用されるようにターゲット デバイスを指定します。テスト中のデバイスに合わせてこの属性に値を設定します。
SIM_USER_ID	64 バイトの 16 進数	任意の 64 ビット 16 進値	すべてのロケーションが 0xFF	SPI メモリのセキュリティレジスタの USER ID を指定します。
SIM_MEM_FILE	文字列	ファイル名およびディレクトリ名	なし	SPI メモリの初期化値を含む HEX ファイルを指定します (オプション)。
SIM_FACTORY_ID	64 バイトの 16 進数	任意の 64 ビット 16 進値	すべてのロケーションが 0xFF	シミュレーション用に、セキュリティレジスタの一意識別子 (Unique Identifier) を指定します (実際の値はデバイス特有の値)。
SIM_DELAY_TYPE	文字列	ACCURATE SCALED	SCALED	シミュレーションを高速に実行するため、一部の遅延値を小さくします。ACCURATE に指定すると、データシート仕様のタイミングと遅延が使用されます。SCALED に設定すると、小さい遅延値が使用されます。デバイスの動作には影響しません。

SPI_ACCESS プリミティブの詳細は、ライブラリ ガイドを参照してください。

シミュレーションでのブロック RAM 競合チェックのディスエーブル

ザイリンクス ブロック RAM メモリは、2 つのポートがいつでも任意のメモリ位置にアクセスできる完全なデュアル ポート RAM です。ただし、同じアドレス空間に対して、同時に読み出しと書き込みを行うことはできません。同時にアクセスすると、ブロック RAM アドレスで競合が発生してしまいます。リード ポートで読み出されるデータは無効なため、競合と言えます。ハードウェアでは、読み出されるデータが、直前のデータ、新しいデータ、または直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、読み出される値は不明なため、出力が X になります。ブロック RAM の競合については、[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、この状況を回避できないものもあります。このような場合には、ブロック RAM が競合を検出しないようにコンフィギュレーションすることができます。ジェネリック (VHDL) またはパラメータ (Verilog) でブロック RAM プリミティブに **SIM_COLLISION_CHECK** を設定します。

SIM_COLLISION_CHECK の文字列

次の表に示す文字列を使用して、競合が発生したときの動作を制御します。

SIM_COLLISION_CHECK の文字列

文字列	競合の発生を示すメッセージを表示	X を出力
ALL	はい	はい
WARN_ONLY	はい	いいえ 競合が発生した場合にのみ適用されます。同じアドレス空間に対する次の読み出しで、X が出力される場合があります。
GENERATE_X_ONLY	いいえ	はい
なし	いいえ	いいえ 競合が発生した場合にのみ適用されます。同じアドレス空間に対する次の読み出しで、X が出力される場合があります。

SIM_COLLISION_CHECK はインスタンスレベルで設定できるので、ブロック RAM のインスタンスごとに制御できます。

シミュレーションでのグローバル リセットおよびトリステート

ザイリンクス FPGA には、デバイス内のレジスタすべてに接続されている専用の配線と回路があります。専用のグローバル セット/リセット (GSR) 信号は、コンフィギュレーション中にアサートされ、コンフィギュレーションが完了すると解放されます。すべてのフリップフロップおよびラッチにこのリセットが送信され、レジスタの定義に従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 信号を使用することも可能ですが、GSR 回路を手動リセットの代わりに使用しないでください。これは、FPGA ではシステムリセットのようなファンアウトが大きい信号に対し、高速バックボーン配線が使用されるためです。バックボーン配線は、専用 GSR 回路よりも高速で、GSR 信号を伝送する専用グローバル配線に比べて解析が簡単です。

バックエンドのシミュレーションでは、コンフィギュレーション後に発生するリセットをシミュレーションするために、GSR 信号のパルスが最初の 100ns 間自動的に発生します。GSR のパルスは、オプションでフロントエンドの論理シミュレーションでも使用可能ですが、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ある場合は不要です。テストベンチを生成する際は、GSR パルスがバックエンドシミュレーションで自動的に発生し、すべてのレジスタがシミュレーションの最初の 100ns 間リセットになることを考慮することが重要です。

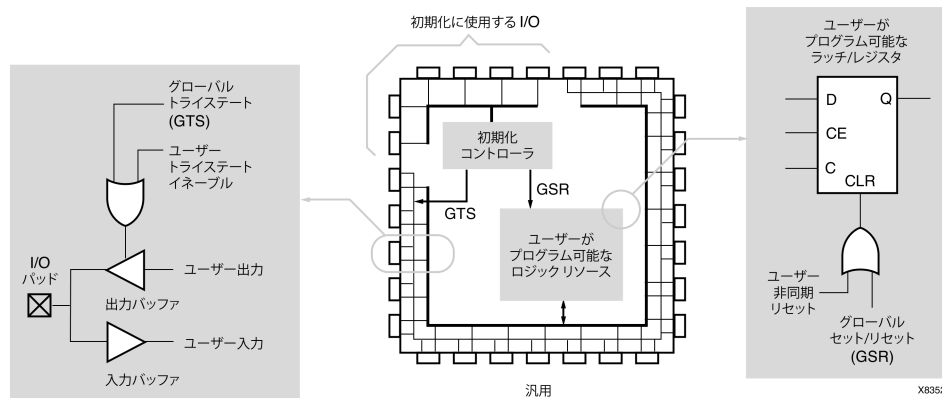
すべての出力バッファは、コンフィギュレーションモード中に専用グローバルトライステート (GTS) ネットにより、ハイインピーダンスに設定されます。一方向、トライステート、または双方向のいずれであるかにかかわらず、汎用出力すべてが影響を受けます。これにより、FPGA デバイスのコンフィギュレーション時に出力が別のデバイスを駆動しないようにできます。

シミュレーションでは、通常 GTS 信号は駆動されません。GTS 信号を駆動する回路は、バックエンドのシミュレーションで使用可能で、フロントエンドのシミュレーションでもオプションで追加可能ですが、GTS パルスの幅はデフォルトで 0 に設定されています。

FPGA デバイスでのグローバルトライステート (GTS) とグローバル セット/リセット (GSR) 信号

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) 信号がどのように FPGA デバイスで使用されるかを示します。

ビルトイン FPGA 初期化回路



Verilog でのグローバル セット/リセット (GSR) とグローバル トライステート (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、`$XILINX/verilog/src/glbl.v` モジュールで定義されています。

`glb.v` モジュールは、グローバル信号をデザインに接続するので、このモジュールをほかのデザインファイルと共にコンパイルし、シミュレーションで `design.v` ファイルおよび `testfixture.v` ファイルと共に読み込む必要があります。

ほとんどの場合、GSR および GTS 信号をテストベンチで定義する必要はありません。glb.v ファイルでは、グローバル GSR および GTS 信号が宣言され、自動的に 100ns 間 GSR 信号にパルスを送ります。バックエンドのシミュレーションでは、このファイルのみが必要で、論理シミュレーションでも通常このファイルのみが必要です。

デザイン階層とシミュレーション

階層を使用すると、次のような利点があります。

- ・ デザインが解読しやすくなる。
- ・ 再利用が容易になる。
- ・ 複数のエンジニアに設計を割り当てることが可能になる。
- ・ 検証が向上する。

デザインの使用率およびパフォーマンスの向上

デザインの使用率およびパフォーマンスを向上させるために、合成ツールまたはザイリンクスのインプリメンテーション ツールでデザイン階層がフラット化されたり、階層が変更されることがありますが、このような場合、階層を再構築することが困難になります。

その結果、RTL 検証で元の階層デザインを使用したときの利点がバックエンドの検証で失われてしまいます。バックエンドのシミュレータでのデザインの表示を向上させるため、ザイリンクス デザイン フローでは元のデザイン階層を保持できます。

インプリメンテーション プロセスでパフォーマンスを低下させたりデザイン リソースを増加させずにデザイン階層を保持するには、次のようにします。

- ・ より厳密なデザイン ルールに従う。
- ・ デザイン階層の境界を越えた最適化が必要にならないよう、デザイン階層を注意して選択する。

デザインのガイドライン

次に、ガイドラインのいくつかを示します。

- ・ 保持するエンティティまたはモジュールの出力すべてにレジスタを付ける。
- ・ クリティカル タイミング パスが複数のエンティティまたはモジュールにまたがらないようにする。
- ・ 関連するロジックまたは共有されるロジックを、同じエンティティまたはモジュール内に配置する。
- ・ I/O (IOB レジスタ、トライステート バッファ、インスタンス化された I/O バッファなど) に配置するロジックをすべて最上位モジュールまたはエンティティに配置する。これには、I/O で使用するダブル データレートのレジスタも含まれます。
- ・ タイミングを向上する必要がある場合、ファンアウトの大きいレジスタを階層の境界に手動で複製する。

階層の維持

全階層または階層の一部を合成中に保持するには、それを合成ツールで設定する必要があります。設定には、次を使用します。

- ・ グローバル オプション
- ・ ソース ファイルのコンパイラ指示子
- ・ 合成コマンド

階層を保持する方法については、合成ツールのマニュアルを参照してください。

階層を保持するために必要な手順に従いデザインを正しく合成すると、階層が保持されたインプリメンテーション ファイル (EDIF または NGC) が作成されます。

ザイリンクス ソフトウェアでデザインをインプリメントする前に、デザインの各インスタンスに KEEP_HIERARCHY 制約を配置して、階層が保持されるようにします。この制約により、フラット化したり変更を加えたりしない部分を指定でき、階層の境界を正しく保持できます。

この制約は、ソース コードで属性として渡すか、NCF または UCF ファイルでインスタンスの制約として渡すか、または合成ツールで自動的に生成させることができます。

詳細は、合成ツールのマニュアルを参照してください。

KEEP_HIERARCHY 制約の詳細は、『[制約ガイド](#)』を参照してください。

デザインのマップ、配置、配線後に、次のパラメータを使用して NetGen を実行すると、デザインの階層が正しくバックアノテートされます。

```
netgen -sim -ofmt {vhdl|verilog} design_name.ncd netlist_name
```

これは、ISE または XFLOW を使用してシミュレーション ファイルを生成する場合の NetGen のデフォルト設定です。この設定は、ISE または XFLOW を使用せずに NetGen を実行する場合や、ISE または XFLOW でデフォルトのオプションを変更した場合にのみ使用する必要があります。NetGen を上記の設定で実行すると、出力される VHDL または Verilog ネットリストで KEEP_HIERARCHY 制約を指定した階層がすべて再構築されます。

NetGen では、保持された各階層レベルに対しネットリスト ファイルと SDF ファイルを生成できます。この機能を使用すると、デザインの一部に対して完全なタイミング シミュレーションを実行できるので、次が可能になります。

- ・ テストベンチの再利用
- ・ チームでの検証手法
- ・ 検証時間の短縮

KEEP_HIERARCHY 制約が設定された各インスタンスに対して個別のファイルを生成するには、**-mhf** オプションを使用します。このオプションを **-dir** オプションと共に使用すると、関連するファイルすべてを別のディレクトリに保存できます。

```
netgen -sim -ofmt {vhdl|verilog} -mhf -dir directory_name  
design_name.ncd
```

NetGen を **-mhf** オプションを使用して実行する場合、`design_mhf_info.txt` という名前のテキストファイルも生成されます。このファイルには、生成されたモジュールおよびエンティティの名前、関連インスタンス名、SDF ファイル、サブ モジュールがすべて記述されます。このファイルは、適切なコンパイル順および SDF アノテーション オプションを決定するときに有益です。

mhf_info.txt ファイルの例

次は、VHDL で生成したネットリストの mhf_info.txt ファイルの例です。

```
// Xilinx design hierarchy information file produced by netgen
// The information in this file is useful for
//   - Design hierarchy relationship between modules
//   - Bottom up compilation order (VHDL simulation)
//   - SDF file annotation (VHDL simulation)
//
// Design Name : stopwatch
//
// Module      : The name of the hierarchical design module.
// Instance    : The instance name used in the parent module.
// Design File : The name of the file that contains the module.
// SDF File    : The SDF file associated with the module.
// SubModule   : The sub module(s) contained within a given module.
//      Module, Instance : The sub module and instance names.

Module      : hex2led_1
Instance    : msbled
Design File : hex2led_1_sim.vhd
SDF File    : hex2led_1_sim.sdf
SubModule   : NONE

Module      : hex2led
Instance    : lsbled
Design File : hex2led_sim.vhd
SDF File    : hex2led_sim.sdf
SubModule   : NONE

Module      : smallcntr_1
Instance    : lsbcnt
Design File : smallcntr_1_sim.vhd
SDF File    : smallcntr_1_sim.sdf
SubModule   : NONE

Module      : smallcntr
Instance    : msbcnt
Design File : smallcntr_sim.vhd
SDF File    : smallcntr_sim.sdf
SubModule   : NONE

Module      : cnt60
Instance    : sixty
Design File : cnt60_sim.vhd
SDF File    : cnt60_sim.sdf
SubModule   : smallcntr, smallcntr_1
              Module : smallcntr, Instance : msbcnt
              Module : smallcntr_1, Instance : lsbcnt

Module      : decode
Instance    : decoder
Design File : decode_sim.vhd
SDF File    : decode_sim.sdf
SubModule   : NONE

Module      : dcm1
Instance    : Inst_dcm1
```

```
Design File : dcml_sim.vhd
SDF File    : dcml_sim.sdf
SubModule   : NONE

Module      : statmach
Instance    : MACHINE
Design File : statmach_sim.vhd
SDF File    : statmach_sim.sdf
SubModule   : NONE

Module      : stopwatch
Design File : stopwatch_timesim.vhd
SDF File    : stopwatch_timesim.sdf
SubModule   : statmach, dcml, decode, cnt60, hex2led, hex2led_1
              Module : statmach, Instance : MACHINE
              Module : dcml, Instance : Inst_dcml
              Module : decode, Instance : decoder
              Module : cnt60, Instance : sixty
              Module : hex2led, Instance : lsbled
              Module : hex2led_1, Instance : msbled
```

生成されたインスタンスの命名規則がシミュレータと合成ツールで異なるため、generate 文で生成された階層が、元のシミュレーションと一致しない場合があります。

ザイリンクス ライブラリを使用した RTL シミュレーション

ザイリンクスのシミュレーション ライブラリは、VHDL-93 および Verilog-2001 言語規格をサポートするどのシミュレータでもシミュレーションできます。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一部の遅延およびモデリング情報は、ライブラリに組み込まれています。

論理シミュレーションであってもクロックのエッジでデータ信号を切り替えないようにしてください。シミュレータでは、同じシミュレーション時間で切り替わる信号間にユニット遅延が追加されます。データがクロックと同時に変化すると、シミュレータでデータがクロック エッジの後で入力される可能性があり、最初のクロック エッジの前にデータを入力するつもりであっても、データが次のクロック エッジまで入力されません。このような、シミュレーション結果が予測と一致しない状況を回避するため、データ信号とクロック信号を同時に切り替えないようにしてください。

デルタ サイクルとレース状態

ザイリンクスでは、イベント ベースのシミュレータをサポートしています。イベント ベースのシミュレータでは、指定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。これらのイベントを処理中は、シミュレーション時間を進めることはできません。この時間は、デルタ サイクルと呼ばれます。指定のシミュレーション時間内に、デルタ サイクルが複数発生することがあります。処理するトランザクションがなくなると、シミュレーション時間が進みません。この理由から、シミュレータで予測されない結果が得られることがあります。次の VHDL コード例は、予測されない結果がどのように発生するかを示します。

予測されない結果が得られる 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;
```

上記の例には、次の 2 つのクロックそれぞれに対して同期処理があります。

- ・ **clk**
- ・ **clk_b**

シミュレータでシミュレーション時間が進む前に **clk = clk_b** の代入が実行されるため、2 つのクロック エッジで実行するはずの処理が 1 つのクロック エッジで実行され、レース状態が発生します。

次に、このような場合の推奨方法の一部を示します。

- ・ クロックとデータを同時に変更しないようにします。各出力に遅延を挿入してください。
- ・ 同じクロックを使用するようにします。
- ・ 次に示すように一時信号を使用して、デルタ遅延を挿入します。

```
clk_b <= clk;
clk_prcs : process (clk)
begin
    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;
```

ほとんどすべてのイベント ベース シミュレータで、デルタ サイクルを表示できます。シミュレーションの問題をデバッグする際に、この機能を活用してください。

シミュレーションの精度

シミュレーションの精度は 1ps にすることをお勧めします。**DCM** などの一部のザイリンクス プリミティブ コンポーネントでは、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適切に実行するため、精度を 1ps にする必要があります。

ザイリンクス シミュレーション モデルでは、シミュレーションの精度を低くしても、シミュレータのパフォーマンスは向上しません。シミュレーション時間の大部分はデルタ サイクルで占められますが、デルタ サイクルはシミュレーションの精度には影響されないため、シミュレーションのパフォーマンスにはそれほど効果はありません。

ザイリンクスでは、fs を使用するなど、精度をこれ以上高くしないことをお勧めします。シミュレータによって、値が切り上げられたり切り下げられたりします。

テスト装置で測定可能なタイミングは ps の単位なので、1ps の精度が最低の精度です。すべての HDL シミュレーションで精度を 1ps にすることをお勧めします。

SecureIP モデルの暗号化手法

ザイリンクスでは、Verilog LRM – IEEE 標準規格 1364-2005 で規定されている最新の暗号化手法を利用しています。PowerPC® プロセッサ、MGT、PCIe® などのハード IP のデバイス シミュレーション モデルでこの手法が使用されます。

コンピュータに適切なバージョンのシミュレータが存在すれば、すべて Compxlib で自動的に処理されます。Verilog でこの手法を使用してシミュレーションを実行する場合、SecureIP ライブラリを参照する必要があります。

ほとんどのシミュレータでは、このライブラリを **-L secureip** オプションを使用して使用できます (**-L secureip** など)。詳細は、「[SecureIP ライブラリ](#)」を参照してください。

シミュレータで使用するオプションについては、シミュレータのマニュアルを参照してください。

デザイン入力に VHDL を使用している場合は、新しい IP 暗号化手法を使用してハード IP をシミュレーションするのに混合言語ライセンスが必要です。

ゲート レベル ネットリストの生成 (NetGen の実行)

NetGen を使用すると、デザイン ファイルから検証 ネットリスト ファイルを生成できます。タイミング シミュレーション ネットリストを作成するには、次の方法で NetGen を実行します。

- ISE® Design Suite
ISE でバックアノテートされたシミュレーション ネットリストを生成する方法については、ISE ヘルプを参照してください。
- XFLOW
XFLOW のオプションおよび XFLOW オプション ファイルの一覧を表示するには、引数なしで「**xflow**」とコマンド プロンプトに入力します。XFLOW のオプションおよびオプション ファイルの詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。
- コマンド ラインまたはスクリプト ファイル
コマンド ラインまたはスクリプト ファイルからシミュレーション ネットリストを生成するには、『[コマンドライン ツール ユーザー ガイド](#)』の「NetGen」の章を参照してください。

同期エレメントでの 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違反が発生すると、ラッチ、レジスタ、RAM、またはその他の同期エレメントからデフォルトで X が出力されます。

これは、タイミング違反が原因で実際の出力値が不明になるためです。実際のレジスタの出力は、次のいずれかになります。

- ・ 直前の値を保持
- ・ 新しい値に更新
- ・ 同期エレメントにクロックを入力後しばらくしてから確実な値が決定するメタステーブル状態

値を決定できないので正しいシミュレーション結果が得られず、エレメントで値が不明であることを示す X が出力されます。別の違反が発生しなければ、値 X は次のクロック サイクルで新しい値に更新されます。

このような状態は、シミュレーションに大きな影響を与える可能性があります。たとえば、レジスタで X が 1 つ出力されると、その後のクロック サイクルでほかのエレメントに X が伝搬され、デザインの大部分が不明になってしまう可能性があります。このような状態が発生した場合は、次のように修正します。

- ・ 同期パスの場合、パスを解析してこのパスおよびその他のパスでのタイミングの問題を修正することにより、回路の動作を確実にする。
- ・ 非同期パスでこの問題が発生し、タイミング違反を回避できない場合は、同期エレメントでの X の伝搬をディスエーブルにする。

X の伝搬をディスエーブルにすると、レジスタの出力でその直前の値が保持されます。実際のシリコンでは、新しい値に更新される可能性があるため、X の伝搬をディスエーブルにすると、シリコンの動作とシミュレーション結果が一致しない場合があることに注意してください。

注意： このオプションは、タイミング違反を回避できない場合にのみ使用してください。

ASYNC_REG 制約の使用

ASYNC_REG 制約には、次のような機能があります。

- ・ デザインの非同期レジスタを識別する。
- ・ これらのレジスタの X の伝搬をディスエーブルにする。

ASYNC_REG は、次の方法でフロントエンド デザインのレジスタに設定します。

- ・ HDL コードの属性
- ・ UCF の制約

レジスタに ASYNC_REG を設定すると、タイミング シミュレーション中にこれらのレジスタで直前の値が保持され、シミュレーションで X は出力されません。

タイミング違反はそれでも発生する可能性があります。新しい値が供給される場合もあるので、注意してください。

ASYNC_REG 制約は、CLB と IOB のレジスタおよびラッチにのみ適用されます。非同期信号の供給を回避できない場合、IOB または CLB レジスタのみに限定してください。RAM、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不定になります。

RAM、SRL、またはその他の同期エレメントに書き込む前に、レジスタ、ラッチまたは FIFO の非同期信号を正しく同期化してください。

詳細は、『[制約ガイド](#)』を参照してください。

MIN/TYP/MAX シミュレーション

標準遅延フォーマット (SDF) ファイルを使用すると、シミュレーションで次の 3 種類の遅延値を指定できます。

- ・ 最小 (**MIN**)
- ・ 標準 (**TYP**)
- ・ 最大 (**MAX**)

ザイリンクスのツールでは、これらの値を使用してターゲット アーキテクチャをさまざまな動作条件でシミュレーションできます。さまざまな動作条件でシミュレーションを実行することで、より正確なセットアップ タイムおよびホールド タイムのタイミング検証を実行できます。

最小 (MIN)

ベスト ケースの動作条件での遅延を表します。ベスト ケースの動作条件は、最小動作温度、最大電圧、およびベスト ケースのプロセス変化で定義されます。この条件下では、デバイスのデータ パス遅延が最小値となり、クロック パス遅延は最大値となります。この状況は、デバイスのホールド タイムを検証するときに最適です。

標準 (TYP)

通常の動作条件での遅延を表します。この状況では、クロック パスおよびデータ パスの遅延値が両方とも最大値となります。この点が、データ パスの遅延値が最大になるのに対しクロック パスの遅延値が最小になる最大 (**MAX**) の値と異なります。ザイリンクスのツールで生成される SDF ファイルでは、このフィールドは使用されません。

最大 (MAX)

ワースト ケースの動作条件での遅延を表します。ワースト ケースの動作条件は、最大動作温度、最小電圧、およびワースト ケースのプロセス変化で定義されます。この条件下では、デバイスのデータ パス遅延が最大値となり、クロック パス遅延は最小値となります。この状況は、デバイスのセットアップ タイムを検証するときに最適です。

正確なタイミング シミュレーション結果

正確なセットアップおよびホールド タイムのタイミング シミュレーションを実行するためには、次のステップを実行します。

- ・ NetGen
- ・ セットアップ シミュレーション
- ・ ホールド シミュレーション

NetGen の実行

正確な標準遅延フォーマット (SDF) 値を得るには、NetGen を **-pcf** オプションで有効な PCF ファイルを指定して実行します。**-pcf** オプションを使用する必要があるのは、新しいザイリンクス デバイスではタイミング情報に相対最小遅延を利用しているからです。NetGen を **-pcf** オプションを使用して実行すると、SDF ファイル内の最小 (**MIN**) と最大 (**MAX**) の値が異なるものになります。

正しい SDF ファイルを作成したら、次の 2 つのシミュレーションを実行してタイミング クロージャを達成します。

- ・ セットアップ シミュレーション
- ・ ホールド シミュレーション

これらのシミュレーションを実行するには、シミュレータを適切なオプションを使用して呼び出す必要があります。

セットアップ シミュレーションの実行

セットアップ シミュレーションを実行するには、**-SDFMAX** コマンドライン オプションを使用して最大 (**MAX**) の値を指定します。

ホールド シミュレーションの実行

ホールド タイムのシミュレーションを実行するには、**-SDFMIN** コマンドライン オプションを使用して最小 (**MIN**) の値を指定します。

SDF オプションをシミュレータに渡す方法の詳細は、シミュレータのマニュアルを参照してください。

絶対最小遅延値を使用したシミュレーション

NetGen の **-s min** オプションを使用すると、シミュレーション用に絶対最小遅延値を生成できます。生成された標準遅延フォーマット (SDF) ファイルでは、この絶対最小遅延値が次の 3 つの遅延値フィールドすべてに記述されます。

- ・ 最小 (**MIN**)
- ・ 標準 (**TYP**)
- ・ 最大 (**MAX**)

最小 (**MIN**) は絶対高速遅延値で、次に示すアーキテクチャの最適な動作条件でパスがターゲット アーキテクチャ内を伝搬するときの遅延です。

- ・ 最低温度
- ・ 最大電圧
- ・ 高品質のシリコン

絶対最小遅延値は通常、ベスト ケースおよびワーストケースでの高速データ パスに対し、ボードレベルおよびチップ間のタイミングを確認する場合にのみ有益です。

デフォルトでは、ターゲット アーキテクチャのワーストケースの温度、電圧、シリコンからワーストケースの遅延値が求められます。回路の動作中に温度および電圧の特性が比較的良好なことがわかっている場合、結果を向上させるためにシミュレータで比例配分したワーストケースの値を使用できます。

デフォルトでは、デバイスの推奨動作条件の範囲内で指定した **TEMPERATURE** および **VOLTAGE** でのワーストケースのタイミング値が適用されます。**TEMPERATURE** および **VOLTAGE** 制約の詳細は、『[制約ガイド](#)』を参照してください。

最小 (**MIN**) 値を含む SDF ファイルは、絶対最小遅延値をサポートするデバイスにのみ生成されます。

VOLTAGE および TEMPERATURE 制約の使用

次の内容が含まれています。比例配分は、既存のスピードファイルの遅延に対して行われ、すべての遅延に対してグローバルに適用されます。比例配分制約である **VOLTAGE** および **TEMPERATURE** では、既知の環境パラメータに基づいてタイミング遅延の特性を定義できます。

VOLTAGE および **TEMPERATURE** 制約の詳細は、『[制約ガイド](#)』を参照してください。

VOLTAGE 制約の使用

VOLTAGE 制約では、デバイスに供給される電圧に基づいて遅延特性を比例配分します。UCF 構文は次のとおりです。

VOLTAGE=value [units]

説明：

- ・ *value*：電圧を指定する整数または実数です。
- ・ *units*：測定単位（上記の構文では V）を指定します（オプション）。

TEMPERATURE 制約の使用

TEMPERATURE 制約では、ジャンクション温度に基づいて遅延特性を比例配分します。UCF 構文は次のとおりです。

TEMPERATURE=value [C|F|K]

説明：

- ・ *value*
温度を指定する整数または実数です。
- ・ **C**、**F**、**K**：温度の単位です。
 - C = 摂氏（デフォルト）
 - F = 華氏
 - K = ケルビン温度

VOLTAGE および **TEMPERATURE** の値を使用すると、SDF ファイルには比例配分されたワーストケースの値が記述されます。

有効な動作温度と電圧の範囲

ターゲット デバイスの有効な動作温度および電圧の範囲については、デバイスの[データシート](#)を参照してください。制約で指定した温度および電圧の値が指定範囲にない場合、制約は無視されてデバイスのデフォルト値が使用されます。

すべてのアーキテクチャで比例配分のタイミング値がサポートされているわけではありません。シミュレーションでは、**VOLTAGE** および **TEMPERATURE** 制約は UCF ファイルから PCF ファイルに処理されるので、動作条件を遅延アノテーションで使用するには、NetGen を実行するときに PCF ファイルを参照する必要があります。

VHDL で比例配分を使用したシミュレーション ネットリストを生成するには、次のように入力します。

```
netgen -sim -ofmt vhdl [options] -pcf design.pcf design.ncd
```

Verilog で比例配分を使用したシミュレーション ネットリストを生成するには、次のように入力します。

```
netgen -sim -ofmt verilog [options] -pcf design.pcf design.ncd
```

絶対最小遅延と比例配分の両方を使用すると、絶対最小遅延の値のみが SDF ファイルの **MIN** フィールドに表示されます。

比例配分は、特定の FPGA ファミリーでのみ使用でき、ミリタリおよびインダストリアル グレードの製品には使用できません。コマーシャル グレード製品の動作範囲内でのみ使用できます。

異なる遅延値の NetGen オプション

NetGen のオプション	NetGen -sim で生成される SDF ファイルの MIN:TYP:MAX フィールド
-pcf <pcf_file>	MIN:MIN (ホールド タイム) TYP:TYP (無視) MAX:MAX (セットアップ タイム)
デフォルト	MAX:MAX:MAX
-s min	Process MIN: Process MIN: Process MIN
UCF または PCF ファイルの比例配分された電圧または温度	Prorated MAX: Prorated MAX: Prorated MAX

DCM、DLL、および MMCM に関する注意事項

CLKDLL、**DCM**、および **DCM_ADV** に関する注意事項は、次のとおりです。

- ・ [DLL/DCM クロックでスキューが調整されないように見える](#)
- ・ [DCM/DLL における TRACE とシミュレーション モデルの違い](#)
- ・ [LVTTL 以外の入力ドライバ](#)
- ・ [波形ビューアに関する注意事項](#)
- ・ [シミュレーションおよびインプリメンテーションの属性](#)
- ・ [タイミング シミュレーションの理解](#)

DLL/DCM クロックでスキューが調整されないように見える

DLL および **DCM** コンポーネントは、チップに入力されるクロックからのクロック遅延を取り除きます。この結果、発生するスキューは入力クロックとデバイスのレジスタに供給されるクロックの間にデバイスのデータシートで示されている範囲内になります。ただし、タイミング シミュレーションでは、指定範囲内にスキューが調整されていないように見える場合があります。これは、シミュレータによる SDF ファイルの遅延の処理方法が原因です。

SDF ファイルでは、**X_FF** コンポーネントの **CLOCK PORT** 遅延がアノテートされます。ただし、シミュレータによっては、この遅延を考慮する前にクロック信号が波形ビューアに表示されます。シミュレータでクロックのスキューが正しく調整されていないように見える場合、波形ビューアで入力ノードに入力ポートの遅延が表示されるかどうかをシミュレータのマニュアルで確認してください。この遅延が表示されない場合、**X_FF** の **CLOCK PORT** 遅延を内部クロック信号に追加すると、波形ビューアで入力ポートのクロックとデバイスの仕様範囲で揃うはずですが、シミュレーションは正しく実行されており、単に波形ビューアで予期されるノードで信号が表示されていないだけです。**DLL/DCM** が正しく機能していることを確認するには、SDF ファイルの遅延を考慮して入力と内部クロック間の実際のスキューを計算します。

DCM/DLL における TRACE とシミュレーション モデルの違い

シミュレーション モデルを理解するには、次のような違いがあることを知っておく必要があります。

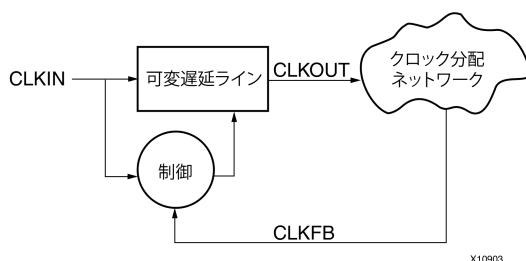
- ・ **DLL/DCM** がシリコンに組み込まれる方法
- ・ TRACE でのタイミングのレポート方法
- ・ シミュレーションでのモデリング方法

DLL/DCM のシミュレーション モデルは、ザイリンクス シリコンでの **DLL/DCM** のファンクションをできるだけ正確にモデル化していますが、シリコンにインプリメントされるファンクションとまったく同じになるとは限りません。シリコンでは、**DLL/DCM** はタップ遅延ラインを使用してクロック信号を遅延します。これには、クロック位相が正しく調整されるように、フィードバックへの入力遅延パスおよびグローバル バッファの遅延パスが含まれます。TRACE または Timing Analyzer では、スタティック タイミング解析でクロックのタイミングを調整できるように、位相の調整を単純な遅延 (通常は負) としてレポートします。

シミュレーションでは、**DLL/DCM** シミュレーション モデル自体で入力クロックをフィードバック入力に戻ってくるクロックと一致させようとします。**DLL** または **DCM** 自体に遅延を付けるのではなく、遅延の一部をまとめてクロック バッファ (コンポーネント) および クロック ネット (ポート) のクロック遅延としてフィードバック パスに追加することによって処理されます。残りの遅延は、**CLKFB** ピンのポート遅延に追加されます。TRACE または Timing Analyzer での遅延のレポート方法やシリコンでの遅延のインプリメント方法と異なりますが、最終的な機能およびタイミングは同じになります。TRACE およびシミュレーションでは、調整可能な遅延タップラインではなく、単純な遅延モデルが使用されます。

DLL および **DCM** の主要機能は、次の図に示すように内部クロック回路からクロック遅延を削除することです。

遅延ロック ループのブロック図



この機能とクロックのスキュー調整を混同しないでください。クロック スキューは、通常クロック ツリー内で遅延が一定していないことと関連しており、この機能とはまったく別のものです。クロック遅延を削除すると、クロック信号が各レジスタに到達したときに、デバイスピンへの入力クロックとクロック信号の位相が正しく揃います。つまり、通常は **DLL/DCM** ピンの信号をモニタしても、クロック遅延が適切に削除されているかは判断できません。

DCM が正しく動作しているかを確認するには、デザインの入力ポートの入力クロックと、そのクロックが供給されるレジスタのクロック ピンを比較します。これらの位相が揃っているか、または指定の量だけシフトされていれば、**DLL/DCM** は正しく機能しています。

LVTTL 以外の入カドライバ

LVTTL 以外の入力バッファドライバでクロックを駆動する場合、DCM では入力バッファの種類に応じてクロックを調整するのではなく、すべての I/O 規格に対して最適なクロック遅延を与える遅延値を 1 つ使用します。データに同じ入力規格を使用する場合、遅延値は特定されるので問題は発生しません。

入力規格が異なる場合でも、遅延のばらつきは入力遅延量に比べると小さいので、通常はホールド タイム エラーは発生しません。遅延のばらつきは、スタティック タイミング解析およびシミュレーションの両方で計算されるので、セットアップ タイム値が正しくなるはずですが、

波形ビューアに関する注意事項

シミュレータによっては、波形ビューアで予測どおりの遅延タイミングが表示されない場合があります。ModelSim など一部のシミュレータでは、インターコネクト遅延およびポート遅延と入力ピンのコンポーネント遅延がまとめられます。シミュレーションの結果は正しいものですが、波形ビューアの表示は予測とは異なる可能性があります。

インターコネクト遅延はまとめられているので、ModelSim のビューアではピンで発生している遷移が表示されません。シミュレーションは正しく行われていますが、クロック遅延を計算する際にクロック ピンの前のインターコネクト遅延を考慮する必要があります。

シミュレーションおよびインプリメンテーションの属性

シミュレーションとインプリメンテーションに同じ属性が渡されることを確認してください。インプリメンテーションには、**DLL** および **DCM** の属性を次の方法で渡します。

- ・ 合成ツール (ジェネリックまたはインライン パラメータ宣言)
- ・ ユーザー制約ファイル (UCF)

UNISIM モデルの RTL シミュレーションでは、次の方法でシミュレーション属性を渡す必要があります。

- ・ ジェネリック (VHDL)
- ・ インライン パラメータ (Verilog)

DLL/DCM のデフォルト設定を使用しない場合は、RTL シミュレーションの属性とインプリメンテーションの属性が同じであることを確認してください。属性が異なる場合、RTL シミュレーションと実際のデバイスのインプリメンテーションが異なるものになります。

合成ツールでジェネリック マップ手法 (VHDL) およびインライン パラメータ (Verilog) がサポートされている場合は、これらを使用すると、インプリメンテーションとシミュレーションで確実に同じ属性が使用されます。

タイミング シミュレーションの理解

バックアノテートされた (タイミング) シミュレーションでは、遅延が原因で動作が予期したものと異なることがあります。このような問題は、ほとんどの場合デザインでのタイミング違反が原因であり、シミュレータでレポートされます。ほかにもこのセクションで示す状況が発生する場合があります。

タイミング シミュレーションの重要性

FPGA デバイスでデザインを正しく機能させるためには、論理シミュレーションとタイミング シミュレーションの両方が必要です。FPGA デザインはより複雑になっており、従来の検証方法では十分ではありません。シミュレーションは、以前は FPGA デザイン フローでそれほど重要ではありませんでしたが、現在では最も重要な段階の 1 つとなっています。タイミング シミュレーションは、高度な FPGA デバイスを設計する場合に特に重要です。

論理シミュレーション

論理シミュレーションは重要な検証プロセスですが、それだけでは不十分です。論理シミュレーションで検証できるのは RTL デザインの論理機能のみで、タイミング情報は含まれておらず、インプリメンテーションおよび最適化での変更も考慮されません。

スタティック タイミング解析と等価性チェック

デザインがタイミングを満たしているかを検証するのにスタティック タイミング解析と等価性チェックのみを実行する場合、不利な点が多数あります。スタティック タイミング解析では、デザイン全体がセットアップおよびホールド要件を満たしているかを示すのみで、デザインを実行した場合の問題は検出されません。通常、適用されたタイミング制約に関する結果しか得られません。

実際のシステムでは、ブロック RAM の競合などの動的な要因により、FPGA デバイス上でタイミング違反が発生することがあります。デュアル ポートブロック RAM を使用する場合は、同じロケーションで同時に読み出しと書き込みを実行すると、不正なデータが読み出されることがあるので、注意が必要です。スタティック タイミング解析では、このような問題は検出されません。また、タイムスペックが誤って解釈される問題も検出されません。

インシステム テスト

インシステム テストが究極のテストであり、デザインがボード上で機能し、テストで問題が検出されなければ、デバイスは製品化の準備ができたと考えられがちです。インシステム テストは目的によっては効果的な方法ですが、すべての問題がすぐに検出できるわけではありません。デザインを長時間実行して初めて検出されるような問題もあります。たとえば、タイミング違反などの問題は、すべてのデバイスで同じように発生するわけではありません。製品が顧客に発送されてからこのような問題が発見された場合、コストがかさみ、問題の解決にも時間がかかります。インシステム テストを適切に完了するには、SSO、クロストーク、その他のボードに関する問題を解決する必要があります。インシステム テストを実行するには、外部インターフェイスに接続する必要もあり、タイムトゥ マーケットが長くなります。

システムを完全に検証するには、従来の方法では不十分です。さまざまな理由から、ダイナミック タイミング解析の実行が必須であると言えます。

デザインでのグリッチ

グリッチ (小さなパルス) が FPGA 回路または集積回路で発生すると、回路に含まれるトランジスタおよびインターコネクトによってグリッチが伝搬される場合と、グリッチがフィルタされて FPGA デバイスの次のリソースに伝搬されない場合があります。これは、グリッチの幅およびグリッチが伝搬されるリソースの種類によって異なります。シリコンで信号がどのように伝搬されるかを正しくシミュレーションするため、ザイリンクスではこれらの動作がタイミング シミュレーションのネットリストでモデル化されます。

VHDL シミュレーション

VHDL のシミュレーションでは、ライブラリ コンポーネントが NetGen によりインスタンス化され、シミュレーション ネットリストにパルス拒否に対する正しい値がアノテートされます。シミュレーション ネットリストでこれらの文を使用すると、より正確なシミュレーションを実行できます。

Verilog シミュレーション

Verilog シミュレーションでは、**PATHPULSE** 文によってこの情報が標準遅延フォーマット (SDF) ファイルに渡されます。この文を使用して、ネットリストのコンポーネントで拒否またはフィルタされるパルスのサイズを指定します。

タイミング問題のデバッグ

バックアノテートされた (タイミング) シミュレーションでは、標準遅延フォーマット (SDF) ファイルに含まれているタイミング情報が処理されます。そのため、回路が高速な場合やデザインに非同期回路が含まれる場合に、シミュレータでタイミング違反がレポートされることがあります。

このセクションでは、よく発生するタイミング違反を説明し、そのデバッグおよび修正方法を示します。

タイミング シミュレーションの実行後に、シミュレータで生成された警告およびエラー メッセージを確認します。

次は、ModelSim で表示される Verilog デザインでの典型的なセットアップ違反のメッセージの例です。メッセージのフォーマットはシミュレータによって異なりますが、どのメッセージにも同じ基本情報が含まれています。詳細は、合成ツール ベンダーにお問い合わせください。

```
# ** Error:/path/to/xilinx/verilog/src/simprims/X_RAMD16.v(96):  
$setup(negedge WE:29138 ps, posedge CLK:29151 ps, 373 ps);  
# Time:29151 ps Iteration:0 Instance: /test_bench/u1/\U1/X_RAMD16\
```

セットアップ違反のメッセージ：1 行目

```
# ** Error:/path/to/xilinx/verilog/src/simprims/X_RAMD16.v(96):
```

シミュレータ モデルのエラーが発生した行を示します。上記の例では、エラーが発生した行は Verilog ファイル **X_RAMD16** の 96 行目です。

セットアップ違反のメッセージ：2 行目

```
$setup(negedge WE:29138 ps, posedge CLK:29151 ps, 373 ps);
```

エラーの原因となる 2 つの信号に関する次の情報を示します。

- ・ 違反のタイプ (**\$setup**、**\$hold**、**\$recovery** など)。上記の例では、**\$setup** 違反です。
- ・ 違反に関連する各信号の名前およびその信号の値が最後に変化したシミュレーション時間。上記の例では、エラーが発生したのは WE 信号の立ち下がりエッジ (最後に値が変化したのは 29138ps) および CLK 信号の立ち上がりエッジ (最後に値が変化したのは 29151ps) です。
- ・ セットアップに割り当てられた時間。上記の例では、クロックが遷移する前に WE 信号が 373ps 間安定している必要がありますが、WE がクロックの 13ps 前に遷移しています。

セットアップ違反のメッセージ：3 行目

```
# Time:29151 ps Iteration:0 Instance: /test_bench/u1/\U1/X_RAMD16\
```

エラーがレポートされたシミュレーション時間および違反が発生した構造デザイン (**time_sim**) のインスタンスを示します。

タイミング違反の原因

\$setuphold のようなタイミング違反は、セットアップ タイムまたはホールド タイム内にレジスタの入力データ (データまたはクロック イネーブル) が変化すると発生します。最も一般的なタイミング違反は、次のとおりです。

- ・ シミュレーション クロックとタイムスペックの不一致
- ・ クロック スキューが考慮されていない
- ・ 非同期入力、非同期クロックドメイン、位相のずれ

シミュレーション クロックとタイムスペックの不一致

シミュレーションで指定されたクロックの周波数が、タイミング制約で指定されたものよりも大きい場合、タイミング違反が発生する場合があります。たとえば、シミュレーション クロックでの周波数が 5ns で、**PERIOD** 制約が 10ns に指定されている場合、タイミング エラーが発生する可能性があります。この状況は、クロック パスに DLL または DCM が存在する場合は、さらに複雑になります。

通常、この問題はテストベンチまたは制約の指定のいずれかに含まれるエラーが原因で発生します。制約がテストベンチの条件と一致していることを確認し、一致していない場合は修正してください。制約を修正した場合、デザインで配置配線を再実行してすべての制約が満たされるかどうかを確認してください。

クロック スキューが考慮されていない

クロック スキューは、クロック信号がデスティネーション レジスタに到達するのにかかる時間とクロック信号がソース レジスタに到達するのにかかる時間の差異です。データは、1 クロック周期にクロック スキューを追加した時間または 1 クロック周期からクロック スキューを差し引いた時間内にデスティネーション レジスタに到達する必要があります。グローバル バッファを使用する場合は、クロック スキューは通常問題になりませんが、ローカル配線ネットワークをクロック信号に使用する場合は、問題になる可能性があります。

クロック スキューが問題となるかどうかは、TRACE でセットアップ テストを実行してレポートを確認します。セットアップ テストの実行方法および TRACE のレポートの読み方については、『[コマンドライン ツール ユーザー ガイド](#)』の「TRACE」の章を参照してください。また、Timing Analyzer を使用してもクロック スキューを判断できます。この方法については、ISE ヘルプから Timing Analyzer のセクションを参照してください。

非同期入力、非同期クロックドメイン、位相のずれ

タイミング違反は、次のようなデータパスが原因で発生します。

- ・ シミュレーション クロックで制御されないデータパス
- ・ クロックで制御されないデータパス
- ・ 非同期クロックドメインの境界を通過するデータパス
- ・ 非同期入力があるデータパス
- ・ 位相がずれたクロックドメインを通過するデータパス

非同期クロック

デザインでクロックドメインが複数ある場合にその境界を通過するパスがあると、タイミングエラーが発生する可能性があります。ドメインの境界を通過するデータパスが必ずしも非同期とは限りませんが、その可能性を考慮することが重要です。

次のような場合は、常に非同期として処理してください。

- ・ 2 つのクロックの周波数に関連がない場合
- ・ オフチップからクロック信号が供給される場合
- ・ レジスタのクロックにゲートが付いている場合（細心の注意を払って使用する場合を除く）

ソースコードおよびタイミング解析レポートで、問題となるパスが非同期のクロックドメインの境界を通過していないかを確認してください。1 つのドメインから次のドメインに正しくデータが供給されるための時間がデザインで十分許容されない場合、クロック供給方法の再設計が必要となることもあります。1 つのクロックドメインから次のクロックドメインにデータを渡すのに、非同期 FIFO を使用することを検討してください。

非同期入力

クロックが供給されるエレメントで制御されないデータパスは、非同期入力です。このようなデータパスはクロックで制御されないため、セットアップおよびホールドタイムの違反が発生しやすくなります。

問題となるパスが入力レジスタに同期しているかをソースコードで確認してください。同期させることができない場合は、ASYNC_REG 制約を使用するとこの問題を回避できます。詳細は、「[ASYNC_REG 制約の使用](#)」を参照してください。

位相のずれたデータパス

データパスは同じ周波数のクロックで制御できますが、クロックの位相がずれているためにセットアップまたはホールド違反が発生する可能性があります。クロックの周波数に関連したものであっても、位相のずれが原因でセットアップ違反が発生する場合があります。

ソースコードおよびタイミング解析レポートで、問題となるパスがクロックの位相がずれているクロックドメインの境界を通過していないかを確認してください。

デバッグのヒント

タイミング違反が発生した場合は、次の事項を確認してください。

- ・ TRACE または Timing Analyzer でクロック パスが解析されたか。
- ・ TRACE または Timing Analyzer でデータ パスがシミュレーションのクロック スピードで動作可能であるとレポートされたか。
- ・ パス遅延にクロック スキューが考慮されているか。
- ・ データ パスの遅延からクロック パスの遅延を差し引いてもクロック スピードが許容されるか。
- ・ クロック スピードを下げることでセットアップ タイムおよびホールド タイムの違反をなくすことが可能か。
- ・ データ パスが複数のクロックドメインの境界を通過しているか。クロックは同期しているか。これらのクロック間にクロック スキューまたは位相のずれはあるか。
- ・ パスがデバイスへの入力パスの場合、入力スティミュラスを入力する時間を変更することで、セットアップ/ホールド タイムの違反がなくなるか。

これらの質問の答えによって、デザインまたはテストベンチを変更する必要があります。詳細は、「[設計に関する考慮事項](#)」の章を参照してください。

セットアップおよびホールド違反

このセクションでは、セットアップおよびホールド違反について説明します。次の内容が含まれています。

- ・ [ゼロ ホールド タイム](#)
- ・ [負のホールド タイムに関する考慮事項](#)
- ・ [RAM に関する考慮事項](#)

ゼロ ホールド タイム

ザイリンクスの[データシート](#)には、グローバル クロック バッファを使用した場合、デフォルトの遅延値で内部レジスタと I/O レジスタでのホールド タイムがゼロになると記載されていますが、シミュレーションでホールド タイム違反が発生する可能性はあります。このホールド タイム違反は、実際にはレジスタでのセットアップ違反ですが、CLB 遅延を正しく表現するために、セットアップタイムの一部がホールド タイムとして処理されています。

負のホールド タイムに関する考慮事項

以前のザイリンクス シミュレーション モデルでは、負のホールド タイムがゼロ ホールド タイムとして指定されています。これが原因でシミュレーションが不正になるわけではありませんが、実際の FPGA で可能なタイミングよりも結果が悪くなり、厳しいタイミング要件を満たすことが困難になることがあります。

タイミングをより正確に表現するため、現在ではタイミング モデルで負のホールド タイムが指定されています。同期モデルのセットアップおよびホールドのパラメータは、1 つのセットアップ/ホールド パラメータにまとめられています。これにより、シミュレーション方法が変更されるわけではありません。

Cadence 社の NC-Verilog では、セットアップとホールドのそれぞれに違反メッセージが表示されるのではなく、1 つのセットアップ/ホールド違反が表示されます。

RAM に関する考慮事項

このセクションでは、セットアップおよびホールド違反の RAM に関する考慮事項を示します。次の内容が含まれています。

- ・ [タイミング違反](#)
- ・ [競合チェック](#)
- ・ [階層に関する考慮事項](#)

タイミング違反

ザイリンクス デバイスには、次の 2 種類のメモリが含まれています。

- ・ ブロック RAM
- ・ 分散 RAM

これらのメモリでは同期エレメントなので、タイミング違反を回避する必要があります。データが正しく保存されるようにするには、クロック信号が到達する前に、データ入力、アドレス入力、イネーブルがすべて安定している必要があります。

競合チェック

ブロック RAM では、同期読み出しが可能です。読み出しサイクルでは、クロック信号が到達する前にアドレス入力とイネーブルが安定している必要があります。安定していないと、タイミング違反が発生します。

ブロック RAM をデュアル ポート モードで使用する場合、メモリの競合が発生しないように注意する必要があります。メモリの競合は、次の 2 つの状況が重なったときに発生します。

1. 1 つのポートでデータの書き込みを実行している。
2. 同時または短時間内に、もう 1 つのポートで同じアドレスに対しデータの読み出しまたは書き込みを実行する。

競合が発生すると、警告メッセージが表示されます。

RAM が 1 つのポートで読み出され、もう一方で書き込まれる場合、モデルでは不明の値を示す X 値が出力されます。2 つのポートが同時にデータを同じアドレスに書き込んでいる場合、モデルでは不明のデータがメモリに書き込まれます。このような状況を回避するため、細心の注意を払う必要があります。競合チェックの詳細は、デバイスの[ユーザー ガイド](#)で「Design Considerations」(デザインに関する注意事項)の章の「Using Block SelectRAM™ Memory」(ブロック SelectRAM メモリの使用)を参照してください。

競合チェックをディスエーブルにするには、ジェネリック (VHDL) またはパラメータ (Verilog) に対して「[シミュレーションでのブロック RAM 競合チェックのディスエーブル](#)」に記述されている方法を使用します。

階層に関する考慮事項

最上位の信号がセットアップ タイムおよびホールド タイムを満たして正しく切り替わっても、最下位プリミティブでエラーが発生する可能性があります。これは、信号が下位階層のプリミティブに到達したときには、信号間の遅延差が削減され、セットアップ タイム違反の原因となることがあるためです。

この問題を修正するには、次の手順に従います。

1. デザイン階層を表示し、エラーが発生したインスタンスの信号を最上位の波形に追加します。最下位でセットアップ タイム違反が発生しているかどうかを確認します。
2. エラーが発生したこのインスタンスに対応する RTL (合成前) のデザイン パスを見つけます。
3. RTL パスにタイミング制約を設定し、タイミング違反が発生しないようにします。インプリメントされたデザインには、タイミング制約の適用されないパスが多少ありますが、ほとんどの場合、これらのパスがセットアップおよびホールド違反の原因となります。

ホールド違反とセットアップ違反のデバッグの手順は同じです。

ザイリンクスでサポートされる EDA シミュレーション ツールを使用したシミュレーション

ザイリンクスでサポートされる EDA シミュレーション ツールを使用したシミュレーションについては、次の付録を参照してください。

- ・ [ModelSim でのザイリンクス デザインのシミュレーション](#)
- ・ [IES でのザイリンクス デザインのシミュレーション](#)
- ・ [VCS および VCS MX でのザイリンクス デザインのシミュレーション](#)

設計に関する考慮事項

この章では、設計に関する考慮事項を示します。次のセクションが含まれています。

- ・ [アーキテクチャの理解](#)
- ・ [クロック リソース](#)
- ・ [タイミング要件の定義](#)
- ・ [合成に関する推奨事項](#)
- ・ [インプリメンテーション オプションの選択](#)
- ・ [クリティカル パスの評価](#)
- ・ [SmartGuide テクノロジ](#)

アーキテクチャの理解

新しい FPGA アーキテクチャを評価する際は、そのアーキテクチャのハードウェア機能とそのトレードオフを考慮する必要があります。ほとんどの FPGA デザインは、VHDL または Verilog などのハードウェア記述言語で記述され、合成ツールでアーキテクチャにマップします。

HDL コードを記述する際は、使用するアーキテクチャのハードウェア機能を考慮し、合成ツールでハードウェアへのマップが効率的に行われ、最適なパフォーマンスを達成できるようにすることが重要です。デザインの設計を開始する前に、ターゲット デバイスの[ユーザー ガイド](#)および[データシート](#)を参照することをお勧めします。

スライスの構造

スライスには、順序回路および組み合わせ回路を FPGA デバイスにインプリメントするための基本エレメントが含まれています。エリアを最小限に抑え、最高のデザイン パフォーマンスを達成するには、デザインでスライスの機能が効果的に使用されているかを知ることが重要です。この際、次の点を考慮します。

- ・ スライスに含まれている基本エレメント、およびこれらの基本エレメントの異なるコンフィギュレーション方法。たとえば、ルックアップ テーブル (LUT) は、分散 RAM またはシフトレジスタでコンフィギュレーションできます。
- ・ これらの基本エレメント間の専用インターコネクト。たとえば、LUT から複数のレジスタへのファンアウトが原因で、スライスの最適なパックが妨げられていないかなどを考慮します。
- ・ 制御信号およびクロックなど、スライスのエレメントに共通の入力がないか、それによりパックが制限されていないか。セット/リセット、クロック イネーブル、およびクロックが共通のレジスタを使用すると、デザインのパックが向上します。ロジックを複製すると、同じリセット ネットに複数の名前が付けられ、スライスへのレジスタのパックが最適になりません。合成でリセット ネットとクロック イネーブルの複製をオフにすることを考慮してみてください。
- ・ LUT のサイズ、およびデザインの特定の組み合わせファンクションをインプリメントするのに必要な LUT の数。

ハード IP ブロック

BRAM、DSP ブロックなどのハード IP ブロックがクリティカル パスのソースまたはデスティネーションとして繰り返しリストされる場合は、次を試してください。

- ・ ブロックの機能を最大限に活用する
- ・ BRAM または DSP ブロックの使用率を確認する
- ・ ブロックの配置を固定する
- ・ ハード IP ブロックとスライス ロジックを比較する
- ・ SelectRAM™ メモリを使用する
- ・ ロジック ファンクションのスライス ロジックへの配置と DSP ブロックへの配置を比較する

ブロックの機能を最大限に活用する

ブロックの機能を最大限に活用しているかどうかを確認してください。FPGA アーキテクチャによっては、これらのブロックにセットアップ タイムおよび clock-to-out タイムを削減するパイプライン レジスタが含まれています。これらの内部レジスタには、通常同期セットおよびリセットがあります。HDL にこの動作が記述されているかどうかを確認してください。ISE および Synplify Pro の HDL Analyst から使用可能なゲートレベルの回路図ビューアを使用すると、合成ツールでハード IP ブロックおよびその機能がどのように推論されているかを確認できます。

BRAM または DSP ブロックの使用率を確認する

BRAM または DSP ブロックの使用率を確認してください。これらのブロックは、FPGA の限られた列にのみ配置されているので、特に使用率が高い場合は、配置が限定されます。また、これらのブロックに接続される I/O およびロジックに対する配置制約によって、さらに制限される場合があります。

ブロックの配置を固定する

BRAM または DSP ブロックの使用率が高く、パフォーマンスが制限される原因となっている場合は、ロケーション制約を使用して配置を固定することを検討してみてください。詳細は、『[制約ガイド](#)』を参照してください。

ハード IP ブロックとスライス ロジックを比較する

ハード IP ブロックの使用とスライス ロジックの使用のトレードオフを考慮します。ハード IP ブロックの代わりにスライス ロジックを使用するかどうかは、ハード IP ブロックがクリティカル パスのソースまたはデスティネーションとして繰り返しリストされ、ハード IP ブロックの機能が最大限に活用されている場合に検討します。

SelectRAM メモリを使用する

デザインにさまざまなメモリ要件がある場合は、BRAM に加えて LUT で構成される分散 SelectRAM を使用することを考慮してください。分散 SelectRAM は LUT で構成されるので、柔軟に配置できます。DSP ブロックの場合、専用パイプライン レジスタの 1 つをスライス レジスタに移動すると、DSP ブロックに接続されているロジックが配置しやすくなる場合があります。

ロジック ファンクションのスライス ロジックへの配置と DSP ブロックへの配置を比較する

加算器などのロジック ファンクションを、スライス ロジックに配置するか DSP ブロックに配置するかを検討します。多くの合成ツールでは、複雑な DSP ファンクションに対して推論される DSP ブロックの数がターゲット デバイスに含まれるブロックの数以下である場合、加算器やカウンタに対して DSP ブロックを推論できます。合成レポートを参照して、これらのブロックが推論されているかどうかを確認してください。

Synplify Pro では、**syn_allowed_resources** 属性を使用すると、ツールで推論するブロックの数を制御できます。詳細は、Synplify Pro のマニュアルを参照してください。DSP ブロックの使用率が高いためにデザインのパフォーマンスが低下しており、ブロックの配置が困難な場合、この属性が有益です。

クロック リソース

ターゲット アーキテクチャのクロック リソースがデザイン要件を満たしているかどうかを評価する必要があります。次の点を考慮します。

- ・ クロック配線リソースの数とタイプ
- ・ 各クロック配線リソースに許容される最大周波数
- ・ 専用クロック入力ピンの数
- ・ DCM や PLL など、クロックを制御するリソースの数とタイプ
- ・ DCM および PLL の周波数、ジッタなどに関する機能と制限、クロック制御の柔軟性

ほとんどのザイリンクス FPGA アーキテクチャでは、デバイスは複数のクロック領域に分割されており、各領域で使用可能なクロック配線リソース数は限られています。通常、クロック配線リソースの合計数は、1 つのクロック領域で使用可能なクロック数よりも多いので、クロックの数がそのクロック領域で使用可能な数を超えてしまうことがよくあります。その場合、クロックを複数の領域に分散する必要があります。これは、同期エレメントがクロック領域の制限を超える配置になってしまう制約がある場合は不可能です。

クロックのインプリメンテーションの評価

デザインのクロックのインプリメンテーション方法を評価するには、ボード レイアウトの前に次の点を解析します。

- ・ DCM または PLL を使用したデザインに必要なクロック周波数と位相シフト。
- ・ 複数のクロックを必要とするハード IP ブロックがあるか。その場合、どのようなタイプのリソースが必要か。デバイスのクロック領域に対してどのように配置する必要があるか。

たとえば、Virtex®-4 では 1 つのクロック領域で 8 個までのグローバル クロック リソースを使用できますが、トライモード イーサネット MAC は 5 個以上のグローバル クロック リソースを使用します。この場合、このクロック領域に関連付けられた I/O バンクに、異なるクロック リソースを必要とする追加の I/O ピンをロックするのを最小限に抑えることをお勧めします。

- ・ デザインに必要なクロック数、これらのクロックドメインの負荷、およびクロック配線リソースのタイプと使用されるクロック バッファ。

FPGA アーキテクチャによっては、複数のタイプのクロック配線リソースを使用できます。たとえば Virtex-5 では、I/O、リージョナル、およびグローバル クロック配線リソースがあります。クロック数の多いデザインでは特に、これらのクロック配線リソースの使用方法を理解し、アーキテクチャのクロック領域の規則に違反しないようにすることが重要です。

- ・ クロックを配置する I/O ピンと、BUFG/DCM/PLL の配置への影響。

ほとんどのアーキテクチャでは、クロックを I/O から入力し、直接 BUFG、DCM、または PLL に接続する場合、BUFG、DCM、または PLL を I/O と同じデバイスの側（上または下、左または右）に配置する必要があります。また、DCM または PLL の出力を BUFG に接続する場合、これらの BUFG を同じデバイスのエッジに配置する必要があります。そのため、デバイスの 1 つのエッジにすべてのクロック I/O を配置すると、そのエッジのリソースが足りなくなり、別のエッジのリソースを使用せざるを得なくなります。この場合、効率の高い専用配線リソースは使用できず、ローカル配線を使用することになりますが、クロックの質が低下し、不要な配線遅延が発生する原因となります。

- ・ 配線リソースを選択し、ハード IP ブロックを特定し、ピン ロケーションの制約を考慮した場合、クロック リソースはクロック領域にどのように分配されるか。

クロック レポート

配置配線レポート(<design_name>.par) には、デザインに含まれるクロックの詳細を示すクロック レポートが含まれています。各クロックに対し、次の情報がレポートされます。

- ・ 使用されているリソースのタイプ（グローバル、リージョナル、ローカル）
- ・ クロック バッファにロケーション制約が設定されているかどうか
- ・ ファンアウト
- ・ 最大スキュー
- ・ 最大遅延

配置配線レポートの参照

配置配線レポートを参照して、クロックに適切なリソースが使用されているか、ネット スキューが適切であるかどうかを確認してください。Spartan®-3 など一部のアーキテクチャでは、レポートでローカル配線と記述される汎用インターコネクトも、注意を払えばクロックに使用できます。

配置配線レポートでクロックにローカル配線リソースが使用されていることが記述されていて、それが計画されたものでない場合やアーキテクチャでサポートされていない場合は、専用クロック配線リソースに配置できるかどうかを確認する必要があります。クロックにグローバルまたはリージョナル クロック リソースを使用できるのにもかかわらず、クロック入力以外のものに接続された場合は、専用クロック配線リソースではなく汎用インターコネクトが使用されます。

クロックには、ゲートを使用するのではなく、クロック イネーブルを使用するか、BUFGMUX を使用して必要なクロックを選択するようにすることをお勧めします。

Virtex®-4 および Virtex-5 デバイスでは、シングル エンド クロックをグローバル クロック入力差動ペアの N 側に配置した場合、クロックリソースには直接配線されず、ローカル配線リソースが使用されます。ローカルリソースが使用されると遅延が増加し、クロックの質が低下します。

クロック領域レポート

ISE® Design Suite では、次の 2 つのレポートが生成されます。

- ・ グローバル クロック領域レポート
- ・ セカンダリ クロック領域レポート

これらのレポートでは、次の点を確認できます。

- ・ グローバルまたはリージョナル クロック リソースの数を超過しているクロック領域
- ・ クロック領域の各クロックで駆動されるリソースの数
- ・ 使用されていないクロック領域、使用されているクロックリソース数が少ないクロック領域
- ・ クロック領域エラーの解決方法、複数のクロック領域間でクロックのバランスを取る方法

MAP をタイミングドリブン パックと配置 (**-timing**) を使用して実行すると、これらの情報は MAP のログ ファイル (`<design_name>.map`) に表示されます。タイミングドリブン パックと配置を使用しない場合は、PAR レポート (`<design_name>.par`) に表示されます。

グローバル クロック領域レポート

グローバル クロック領域レポートは、1 つの領域で使用可能な最大クロックリソース数を超過している場合にのみ作成されます。たとえば、Virtex-5 デバイスでは 1 つのクロック領域で 10 個のグローバル クロックリソースを使用できるので、11 個以上のグローバル クロックリソースが使用されている場合にのみ、グローバル クロック領域レポートが作成されます。

グローバル クロック領域レポートには、次の情報が記述されます。

- ・ クロック領域で使用されているグローバル クロックと、各クロックで駆動されるリソースの数
- ・ DCM、PLL、および BUFG に適用されているロケーション制約
- ・ 各グローバル クロックのロードを適切なクロック領域にロックするエリア グループ制約

セカンダリ クロック領域レポート

セカンダリ クロック領域レポートには、次の情報が記述されます。

- ・ 各クロック領域での BUFIO、BUFR、およびリージョナル クロック スパインの使用
- ・ クロック領域で使用されている I/O およびリージョナル クロック ネットと、各クロックで駆動されるリソースの数
- ・ BUFIO および BUFR に適用されているロケーション制約
- ・ 各リージョナル クロックのロードを適切なクロック領域にロックするエリア グループ制約

ロケーション制約とエリア グループ制約は、レポートが作成された時点の初期配置に基づいて定義されています。この配置は、フローのこの後の段階で実行される最適化により変更される場合があります。これらの制約は、開始点として使用します。まずクロック領域へのクロックの分配を解析してから、クロック領域の規則に従うよう制約を調整するようにしてください。調整した制約をユーザー制約ファイル (UCF) (<design_name> .ucf) に追加すると、この後のインプリメンテーションで使用できます。

タイミング要件の定義

ISE® Design Suite の合成ツールおよびインプリメンテーション ツールは、タイミング制約で指定したパフォーマンス要件に基づいて処理を実行します。次を達成するには、制約を適切に定義する必要があります。

- ・ 合成での正確な最適化
- ・ インプリメンテーションでの最適なパック、配置、配線

デザインでは、内部クロックドメイン、入力および出力 (IO) パス、マルチサイクル パス、False パスの制約を適切に設定する必要があります。詳細は、『[制約ガイド](#)』を参照してください。

厳しすぎる制約

制約を厳しくすることは、デザインの最大パフォーマンスを調べるのに役立ちますが、合成でロジックが過剰に複製される可能性があるため、注意が必要です。

PAR の制約を自動的に緩和する機能を使用すると、ツールで制約を満たすことができないと判断された場合に自動的に制約が緩和されます。これによりランタイムが短縮され、すべての制約に対して最高のパフォーマンスが得られるように処理されます。

合成で指定するタイミング制約は、インプリメンテーションで指定する制約と一致するようにする必要があります。多くの合成ツールでインプリメンテーション用にタイミング制約を書き出すことができますが、このオプションの使用はお勧めしません。インプリメンテーション制約は、ユーザー制約ファイル (UCF) (<design_name> .ucf) で指定してください。サポートされるタイミング制約の説明と構文例は、『[制約ガイド](#)』を参照してください。

制約の適用範囲

合成レポートで複製されたレジスタがあるかどうかを調べ、インプリメンテーションで元のレジスタに適用されたタイミング制約が複製されたレジスタにも適用されるようにしてください。インプリメンテーションのランタイムを短縮し、メモリの使用量を最小限に抑えるには、まず同じタイミング要件を持つパスをグループ化して制約を設定してから、特定のタイミング制約を設定するようにしてください。

グループ化されていない制約の例

```
TIMESPEC "TS_firsttimespec" = FROM "flopa" TO "flop" 10ns;  
TIMESPEC "TS_secondtimespec" = FROM "flopc" TO "flop" 10ns;  
TIMESPEC "TS_thirddtimespec" = FROM "flop" TO "flop" 10ns;
```

グループを使用した制約の例

```
INST "flopa" TNM = "flopgroup";  
INST "flopc" TNM = "flopgroup";  
INST "flop" TNM = "flopgroup";  
TIMESPEC "TS_consolidated" = FROM "flopgroup" TO "flop" 10ns;
```

合成に関する推奨事項

高パフォーマンスの回路を作成するための推奨事項は、次のとおりです。

- ・ 適切なコーディング手法を使用する
- ・ ロジックの推論を解析する
- ・ デザインの完全な情報を入力する
- ・ 最適なツール設定を使用する

適切なコーディング手法を使用する

適切なコーディング手法を使用することにより、合成ツールによる HDL コードのビヘイビア記述からの推論で、デバイスのアーキテクチャが最大限に使用されます。ISE の言語テンプレートには、VHDL と Verilog の両方のコード例が含まれています。

ロジックの推論を解析する

ブロックの機能が最大限に活用されているか、HDL コードから予測される機能が適切に推論されているかを確認してください。これには、Synplify Pro の HDL Analyst などのゲートレベルの回路図ビューアが役立ちます。BRAM を使用する場合は、可能な限り専用出力パイプライン レジスタを使用して、RAM から出力されるデータの clock-to-out 遅延が削減されるようにします。DSP ブロックにも、セットアップ タイムおよび clock-to-out タイムを削減するパイプライン レジスタが含まれています。

デザインの完全な情報を入力する

合成ツールにデザインの完全な情報を入力するようにしてください。

- ・ デザインに CORE Generator™ ソフトウェアで生成された IP、サードパーティ IP、下位レベルのブラック ボックス ネットリストが含まれている場合は、それらのネットリストを合成プロジェクトに含めます。合成ツールでネットリスト内のロジックを最適化することはできませんが、これらのネットリストに接続される HDL コードの最適化が向上します。
- ・ タイミング制約を使用して、デザインのパフォーマンス要件を合成ツールに入力します。インプリメンテーションのクリティカル パスが合成ツールでクリティカル パスと認識されない場合は、Synplify Pro の -route 制約を使用してそのパスに焦点が置かれるようにします。

最適なツール設定を使用する

合成ツールのさまざまな設定を使用して、最適なデザインが得られるようにします。まず、基本的なオプションから開始して、徐々にオプションを追加してその効果を調べます。また、属性の設定もロジックの推論および合成の最適化に影響します。これらの属性を変更する場合、コードを記述し直す必要はありません。「[便利な合成属性](#)」を参照してください。

便利な合成属性

	Xilinx Synthesis Technology (XST)	Synplify Pro
ファンアウト制御	MAX_FANOUT	syn_maxfan
RAM に BRAM または分散 SelectRAM™ を推論するよう指定	RAM_STYLE	syn_ramstyle
DSP48 を使用するよう指定	USE_DSP48	syn_multstylesyn_dspstyle
SRL16 を使用するよう指定	SHREG_EXTRACT	syn_srlstyle
ブロック RAM の使用率を制御	なし	syn_allowed_resources
最適化でレジスタ インスタンスを保持	KEEP	syn_preserve
ワイヤを保持	KEEP	syn_keep
未使用の出力があるブラックボックスを保持	KEEP	syn_noprune
フリップフロップのクロック イネーブルの使用を制御	USE_CLOCK_ENABLE	なし
同期セットの使用を制御	USE_SYNC_SET	なし
同期リセットの使用を制御	USE_SYNC_RESET	なし

すべての属性とその機能の詳細は、合成ツールのマニュアルを参照してください。XST 制約の詳細は、『[XST ユーザー ガイド](#)』を参照してください。

その他のタイミング オプション

Synplify Pro のリタイミングや XST のレジスタの調整など、ロジックを複製する可能性のあるオプションを使用すると、タイミングは向上しますが、エリアが増加する可能性があります。

特定のネットのファンアウトを削減するには、ファンアウトの制限をグローバルに設定するのではなく、そのネットにファンアウトを制御する属性を指定してください。

階層を保持する場合は、階層の境界のポートがレジスタを介するようにしてください。クリティカルパスが階層の境界を通過する場合、合成ツールで一部の最適化を実行できません。また、階層を保持すると、インプリメンテーション ツールで使用される物理合成オプションによる最適化も制限され、パフォーマンスが低下し、エリアが大きくなる原因となります。

別の方法として、次の目的で **KEEP HIERARCHY** を **soft** に設定することもできます。

- ・ 合成で階層を保持する。
- ・ 合成後のシミュレーションを実行しやすくする。
- ・ MAP の物理合成オプションによる階層の境界を越えた最適化を可能にする。

KEEP_HIERARCHY の詳細は、『[制約ガイド](#)』を参照してください。

インプリメンテーションを開始する前に、次の操作を実行してください。

- ・ 合成レポートの警告メッセージを確認します。
- ・ RTL 回路図ビューで、HDL コードが合成ツールでどのように解釈されているかを確認します。テクノロジ回路図を使用して、HDL コードがターゲット アーキテクチャにどのようにマップされているかを確認します。

インプリメンテーション オプションの選択

最高のパフォーマンスを達成するオプションの組み合わせは、次の要素によって異なります。

- ・ パフォーマンス要件
- ・ 合成フロー
- ・ 構造

パフォーマンス評価モード

タイミング制約を指定しない場合、ISE でパフォーマンス評価モードを使用して、デザインのパフォーマンスを簡単に評価できます。このモードでは、インプリメンテーション用に各内部ブロックに対してタイミング制約が自動的に生成されます。このモードを使用する場合は、ユーザー制約ファイル (UCF) を指定しないでください。パフォーマンス評価モードを使用すると、タイミング要件を指定せずに、インプリメンテーションで高パフォーマンスを達成できます。

タイミングドリブン パックと配置オプション

MAP のタイミングドリブン パックと配置オプション (**map -timing**) を使用します (アーキテクチャでサポートされている場合)。このオプションをイネーブルにすると、MAP でパックと配置が実行され、PAR では配線のみが実行されます。パックと配置を統合し、両方のプロセスでタイミング情報を使用することにより、ハードウェアの機能がよりよく活用されるようになり、パフォーマンスが向上します。

Virtex®-5 デバイスでは、MAP は常にタイミングドリブン パックと配置を使用して実行されます。Virtex-5 デバイスのスライス構造は複雑なため、このオプションを使用しないと効率的なパックが実行されません。最高のパフォーマンスを達成するには、MAP と PAR のエフォートレベルを **High** に設定することをお勧めします。ランタイムはエフォートレベルを **std** に設定した場合より長くなりますが、初期の結果が向上します。

物理合成オプション

インプリメンテーションで物理合成オプションを使用すると、デザインのクリティカル パスの情報に基づいて、ロジックの最適化およびパックを再実行できます。物理合成オプションは、MAP で適用されます。次のようなオプションがあります。

- ・ グローバル ネットリスト最適化
- ・ ローカル ロジックの最適化
- ・ リタイミング
- ・ レジスタの複製
- ・ 等価レジスタの削除

詳細は、[ホワイト ペーパー](#) WP230『Physical Synthesis and Optimization with ISE®』を参照してください。これらの物理合成オプションは、前述の合成に関する推奨事項に従っていない場合に最も効果があります。物理合成を使用すると、ロジックの複製によりエリアが増加する場合があるので、注意してください。

SmartXplorer

SmartXplorer を使用すると、ISE® Design Suite で最高のデザイン パフォーマンスを達成するインプリメンテーション オプションを調べることができます。次のモードがあります。

タイミング クロージャ モード

SmartXplorer では MAP および PAR が複数回実行されるので、週末に実行するのが一般的です。SmartXplorer で最適な設定が選択されたら、その設定を使用してデザインを実行します。最初に SmartXplorer を実行してから多数の変更を加えており、SmartXplorer で得られた設定でタイミングが満たされなくなっている場合は、SmartXplorer をもう一度実行することを検討してください。

タイミング クロージャ モード

タイミング クロージャ モードは、ISE またはコマンド ラインから設定できます。このモードでは、タイミング制約を評価し、タイミング要件を満たすようにインプリメンテーション オプションを異なる組み合わせで使用します。インプリメンテーションを複数回実行するためランタイムは長くなりますが、最適なオプションの組み合わせが得られれば、タイミング クロージャを達成するために必要なデザインの実行回数は削減できます。

クリティカル パスの評価

クリティカル パスを理解することは、次のデザイン実行でより適切な選択をするために重要です。データ パスは、ロジック遅延とインターコネクト遅延で構成されています。ロジック遅延を構成する個々のコンポーネント遅延は固定されているので、ロジックのレベル数を削減するか、ロジックの構造を変更しないとロジック遅延は削減されません。それに対し、インターコネクト遅延は変更しやすく、ロジックの配置によって決まります。

ロジック レベル数が多い

デザインのロジックレベル数が多いために、配線インターコネクトが多くなっている場合は、次を実行します。

- ・ MAP の物理合成オプションの使用を評価します。
- ・ インプリメンテーションでレポートされているクリティカル パスが合成でレポートされているものと一致しているかどうかを調べます。一致していない場合は、Synplify Pro の `-route` などの制約を使用して合成ツールでそのパスに焦点が置かれるようにします。
- ・ HDL コードでハードウェアが最大限に活用されているかどうかを確認します。
- ・ 推論が、特にハードウェア IP ブロックに対して適切に実行されているかどうかを確認します。

ロジック レベル数が少ない

ロジックのレベル数が少ないのに一部のデータパスがパフォーマンス要件を満たしていない場合は、次を実行します。

- ・ 遅延の大きい配線のファンアウトを調べます。
- ・ クリティカルパスのデスティネーションがフリップフロップのクロックイネーブルまたは同期セット/リセット入力の場合は、ソースとなっている LUT を使用して SR/CE ロジックをインプリメントしてみます。

XST には、同期セット/リセットまたはクロックイネーブル付きのレジスタの推論をグローバルまたは個別にディスエーブルにする属性があります。これらの属性をディスエーブルにすると、同期セット/リセットまたはクロックイネーブルの機能がフリップフロップのデータ入力に推論されます。このようにすると、LUT とフリップフロップが同じスライスにパックされるようになります。Virtex®-5 デバイスでは、各スライスに 3 ～ 4 個のレジスタがあり、すべてに同じ制御ロジックピンを使用する必要があるため、これが特に有益です。
- ・ クリティカルパスにブロック RAM や DSP48 などのハード IP ブロックが含まれている場合は、デザインでエンベデッドレジスタが有効に活用されているかどうかを確認します。また、ハード IP ブロックを使用した場合とスライスロジックを使用した場合のトレードオフを評価します。
- ・ 配置を解析します。ロジック間の距離が長い場合は、クリティカルブロックをフロアプランする必要があります。タイミング要件を満たす必要があるロジックのみをフロアプランしてみてください。フロアプランを過剰に実行すると、パフォーマンスが低下する可能性があります。
- ・ クロックパススキューを評価します。クロックスキューが予測よりも大きい場合は、FPGA Editor ですべてのクロックリソースが専用クロックリソースを使用して配線されているかどうかを確認します。専用クロックリソースが使用されていない場合、クロックスキューが大きくなります。

SmartGuide テクノロジ

デザインの変更されていない部分を保持するには、SmartGuide™ テクノロジを使用します。

SmartGuide テクノロジ

機能	SmartGuide テクノロジ
以前のインプリメンテーションを再利用	あり
変更のないモジュールのインプリメンテーションは完全に同じ	なし
計画的な設計が必要	なし
ランタイムが短縮	配置および配線の両方
使いやすさ	簡単

SmartGuide テクノロジを使用する状況

SmartGuide™ テクノロジは、次のような場合に使用します。

- ・ デザインは完了し、タイミングが満たされているが、多少の変更を加える必要があり、ランタイムを削減する場合
- ・ デザインは完了し、タイミングが満たされているが、属性を変更したり、ピン ロケーションを変更する必要がある場合

SmartGuide テクノロジ

SmartGuide™ テクノロジは、以前のインプリメンテーションの結果（配置配線済みの NCD ファイル）を現在のインプリメンテーション実行のガイドとして使用するよう指定します。SmartGuide テクノロジを使用すると、一貫性のある結果が得られ、ランタイムも向上します。

SmartGuide テクノロジは、次の環境で使用できます。

- ・ ISE® Design Suite
- ・ TCL
- ・ コマンド ライン

SmartGuide テクノロジの使用方法的詳細は、次を参照してください。

- ・ ISE Design Suite ヘルプ
- ・ [『コマンド ライン ツール ユーザー ガイド』](#)

SmartGuide テクノロジの使用に適した変更

SmartGuide テクノロジは、ロジックの論理方程式を変更するなど、小さいな変更を加える場合に最も有益です。新規モジュールや新規インスタンスを追加するなどの大きな変更はデザイン階層に影響するので、以前のインプリメンテーションとの一致率が下がります。

SmartGuide テクノロジの使用に適した変更は、次のとおりです。

- ・ 1 つまたは 2 つのモジュールでの小さなロジックの変更 (10% 未満)
- ・ ピン ロケーションの移動
- ・ コンポーネント上の属性の変更
- ・ タイミング制約の変更

MAP および PAR で指定するオプションは、ガイド ファイルを生成したときの設定と一致させる必要があります。このようにすると、同じアルゴリズムおよび最適化が使用され、一致率が上がります。

タイミング制約と配置制約を両方変更すると、SmartGuide テクノロジの結果に影響します。多数の制約を変更する場合は、まず SmartGuide テクノロジを使用せずにツールを実行して、その結果をガイド ファイルとして制約変更後のインプリメンテーションを実行します。

SmartGuide テクノロジに影響する制約の変更

次のような制約の変更は、SmartGuide テクノロジの結果に影響します。

- ・ ピン ロケーションの移動

ピン ロケーションの移動は、通常うまくいきます。変更したピンおよびネットのみが再配線されます。ただし、ピンを密集した領域に移動し、移動したピンに接続されているネットを配線するのにネットを移動する必要がある場合は、困難になります。この場合、デザインを配線してタイミングを満たすために、連鎖的な影響が発生することがあります。

- ・ コンポーネントの移動

コンポーネントの移動は、ピン ロケーションの移動と同様です。コンポーネントの移動は、タイミングが向上する場合は有益ですが、コンポーネントを密集した領域に移動すると結果が悪化する場合があります。

- ・ タイミング制約の緩和

タイミング制約を緩和すると、タイミングが満たされていなかったパスのタイミングが満たされることがあるので、非常に有益です。ロジックが変更されたかどうかにかかわらず、SmartGuide テクノロジは常にタイミングを満たすことを試みます。そのため、SmartGuide テクノロジはタイミングを満たしているデザインで使用することをお勧めします。

- ・ タイミング制約を厳しくする

タイミング制約を厳しくすることはお勧めしません。制約の変更によりパスのタイミングが満たされなくなった場合、配線を完了しタイミングを満たすため、そのパスおよびほかのロジックが再インプリメントされます。

SmartGuide テクノロジを使用しない再インプリメンテーション

SmartGuide テクノロジを使用して 10 回程度インプリメンテーションを実行した場合は、デザイン全体を最適化するために SmartGuide テクノロジを使用せずに再インプリメントすることをお勧めします。SmartGuide テクノロジを使用せずに再インプリメントすると、SmartGuide テクノロジでガイドされていたロジックと変更されたロジック間で最適化が実行されます。

ModelSim でのザイリンクス デザインのシミュレーション

この付録では、ModelSim を使用したザイリンクス デザインのシミュレーションについて説明します。次のセクションが含まれています。

- ・ [ModelSim でのザイリンクス デザインのシミュレーション](#)
- ・ [ModelSim および Questa を使用した SecureIP のシミュレーション](#)

ModelSim でのザイリンクス デザインのシミュレーション

論理シミュレーションを実行する前に、Compplib を使用してザイリンクス シミュレーション ライブラリを使用するシミュレータ用にコンパイルする必要があります。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

ISE Design Suite からのシミュレーションの実行 (VHDL/Verilog)

ISE® Design Suite では、シミュレーションに必要なコマンドが自動的に生成されます。

1. [Design] パネルの [View] ペインで [Simulation] をオンにし、ドロップダウンリストから実行するシミュレーションを選択します。
2. [Hierarchy] ペインでテストベンチ ファイルを選択します。
3. [Processes] ペインで [Simulate <実行するシミュレーション> Model] プロセスをダブルクリックします。

ModelSim (スタンドアロン) での論理シミュレーション

このセクションには、次の項目が含まれます。

- ・ [ModelSim \(スタンドアロン\) での論理シミュレーション \(Verilog\)](#)
- ・ [ModelSim \(スタンドアロン\) での論理シミュレーション \(VHDL\)](#)

ModelSim (スタンドアロン) での論理シミュレーション (VHDL)

ModelSim (スタンドアロン) を使用して VHDL の論理シミュレーションを実行するには、次の手順に従います。

1. 次のものをコンパイルします。

- a. ソース ファイル
- b. テストベンチ

次に例を示します。

```
vcom -93 <source1>.vhd <source2>.vhd ... testbench.vhd
```

2. デザインを読み込みます。

```
vsim -t lps work.<testbench>
```

ModelSim (スタンドアロン) での論理シミュレーション (Verilog)

ModelSim (スタンドアロン) を使用して Verilog の論理シミュレーションを実行するには、次の手順に従います。

1. 次のものをコンパイルします。

- a. **glbl.v** モジュール
- b. ソース ファイル
- c. テストベンチ

次に例を示します。

```
vlog $env(XILINX)/verilog/src/glbl.v <source1>.v <source2>.v ... <testbench>.v
```

glbl.v モジュールの詳細は、「[シミュレーションでのグローバル リセットおよびトライステート](#)」を参照してください。

2. ModelSim にデザインを読み込みます。

- a. **-L** を使用してデザインで使用するライブラリを指定します。
- b. **glb.v** モジュールを読み込みます。

次に例を示します。

```
vsim -t ps -L unisims_ver -L xilinxcorelib_ver work.<testbench> work.glbl
```

glbl.v により、シミュレーションの開始後 100ns 間グローバル セット/リセット (GSR) パルスが生成されます。

ModelSim (スタンドアロン) でのバックアノテートされたシミュレーション

このセクションでは、ModelSim を使用したバックアノテートされたシミュレーションの実行方法を説明します。次の内容が含まれます。

- ・ [ModelSim \(スタンドアロン\) でのバックアノテートされたシミュレーション \(VHDL\)](#)
- ・ [ModelSim \(スタンドアロン\) でのバックアノテートされたシミュレーション \(Verilog\)](#)

ModelSim (スタンドアロン) でのバックアノテートされたシミュレーション (VHDL)

ModelSim (スタンドアロン) を使用して VHDL のバックアノテートされたシミュレーションを実行するには、次の手順に従います。

1. シミュレーション モデルを作成します。

- a. ISE Design Suite でのシミュレーション モデルの作成

[Implement Design] プロセスの各段階の下に、シミュレーション モデルを生成するプロセスがあります。たとえば、[Place & Route] プロセスの下には [Generate Post-Place & Route Simulation Model] プロセスがあります。このプロセスを実行すると NetGen が実行され、シミュレーション モデルおよびタイミング情報を含む SDF ファイルが生成されます。モデルおよび SDF ファイルのデフォルト名は、**<design_name>_timesim.vhd** および **<design_name>_timesim.sdf** です。モデル生成のプロパティを変更するには、プロセスを右クリックして [Properties] をクリックします。各プロパティの詳細は、[Help] をクリックしてください。

- b. コマンド ラインでのシミュレーション モデルの作成

シミュレーション モデルを作成するには、NetGen を使用します。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

2. 次のものをコンパイルします。

- a. 生成したシミュレーション モデル

- b. テストベンチ

次に例を示します。

```
vcom -93 <design_name>_timesim.vhd testbench.vhd
```

3. デザインと Standard Delay Format (SDF) ファイルを読み込みます。

次に例を示します。

```
vsim -t ps -sdfmax /UUT=<design_name>_timesim.sdf work.testbench
```

次の情報が必要です。

- ・ SDF ファイルを適用する領域。この領域は、生成されたタイミング シミュレーション ネットリストをインスタンス化する場所を指定します。テストベンチ内のエンティティ名を **TESTBENCH**、テストベンチにインスタンス化されるシミュレーション ネットリストのインスタンス名を **UUT** とすると、領域は **/TESTBENCH/UUT** です。
- ・ SDF ファイルのディレクトリ。SDF ファイルがシミュレーション ネットリストと同じディレクトリにある場合は、SDF ファイル名のみを指定します。別のディレクトリにある場合は、パス全体を指定する必要があります。

次に、VSIM のコマンドライン例を示します。

```
vsim -t ps -sdfmax /testbench/uut=c:/project/sim/time_sim.sdf work.testbench
```

ModelSim (スタンドアロン) でのバックアノテートされたシミュレーション (Verilog)

ModelSim (スタンドアロン) を使用して Verilog のバックアノテートされたシミュレーションを実行するには、次の手順に従います。

1. シミュレーション モデルを作成します。
 - a. ISE® Design Suite でのシミュレーション モデルの作成
[Implement Design] プロセスの各段階の下に、シミュレーション モデルを生成するプロセスがあります。たとえば、[Place & Route] プロセスの下には [Generate Post-Place & Route Simulation Model] プロセスがあります。このプロセスを実行すると NetGen が実行され、シミュレーション モデルおよびタイミング情報を含む SDF ファイルが生成されます。モデルおよび SDF ファイルのデフォルト名は、<design_name>_timesim.v および <design_name>_timesim.sdf です。モデル生成のプロパティを変更するには、プロセスを右クリックして [Properties] をクリックします。各プロパティの詳細は、[Help] をクリックしてください。
 - b. コマンド ラインでのシミュレーション モデルの作成
シミュレーション モデルを作成するには、NetGen を使用します。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。
2. ModelSim にデザインを読み込みます。
 - a. **-L** オプションを使用して、シミュレーション モデル内のコンポーネントの動作を定義する Verilog SIMPRIM モデルを指定します。
 - b. **glbl** モジュールを読み込みます。
次に例を示します。

```
vsim -t ps -L simprims_ver work.<testbench> work.glbl
```

Verilog では、タイミング シミュレーション ネットリストに SDF ファイルを呼び出す **\$sdf_annotate** 文が含まれているので、シミュレーションを実行すると SDF ファイルが自動的に読み込まれます。**glbl.v** により、シミュレーションの開始後 100ns 間グローバル セット/リセット (GSR) パルスが生成されます。

ModelSim および Questa を使用した SecureIP のシミュレーション

このセクションでは、ModelSim および Questa を使用した SecureIP のシミュレーションについて説明します。SecureIP の詳細は、『[SecureIP モデルの暗号化手法](#)』を参照してください。

SecureIP は Verilog 規格なので、ModelSim および Questa で Verilog ライセンスが必要です。Verilog ライセンスがない場合は、ModelSim で Verilog ライセンスなしで SecureIP のシミュレーションを実行する方法を説明した[アンサー 33118](#) を参照してください。

SecureIP ライブラリを含むザイリンクス ライブラリをコンパイルするには、Compplib を使用してください。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

Compplib ではライブラリが自動的に設定され、modelsim.ini ファイルがそれに応じて変更されます。Compplib を使用してライブラリをコンパイルしたら、それ以上 modelsim.ini を変更する必要はありません。

ModelSim でシミュレーションを実行する際、**-L** オプションを使用して SecureIP ライブラリを指定する必要があります。

```
vsim -t ps -L secureip -L simprims_ver work.<testbench>  
work.glbl
```

SecureIP のシミュレーションに関してヘルプが必要な場合は、<http://japan.xilinx.com/support> からウェブケースを開いてください。

IES でのザイリンクス デザインのシミュレーション

この付録では、Incisive Enterprise Simulator (IES) を使用したザイリンクス デザインのシミュレーションについて説明します。次のセクションが含まれています。

- ・ [ISE® Design Suite からのシミュレーションの実行](#)
- ・ [NC-Verilog でのシミュレーション](#)
- ・ [NC-VHDL でのシミュレーション](#)

ISE Design Suite からのシミュレーションの実行

IES は ISE® Design Suite に統合されていません。

NC-Verilog でのシミュレーション

このセクションでは、Cadence 社の NC-Verilog を使用したシミュレーションの実行方法を説明します。次の内容が含まれます。

- ・ [NC-Verilog でのシミュレーション \(方法 1\)](#)
- ・ [NC-Verilog でのシミュレーション \(方法 2\)](#)
- ・ [NC-Verilog を使用した SecureIP のシミュレーション](#)

NC-Verilog でのシミュレーション (方法 1)

この方法では、コンパイル時間のオプションを含むライブラリ ソース ファイルを使用します (Verilog-XL と同様)。RTL シミュレーションでは、デザインの構成 (インスタンス化されたザイリンクス プリミティブ、CORE Generator™ コンポーネントなど) によって、コマンドラインに次のように入力します。

```
irun -y $XILINX/verilog/src/unisims -y  
$XILINX/verilog/src/XilinxCoreLib \  
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \  
<testfixture>.v <design>.v
```

\$XILINX/verilog/src/unisims には、RTL シミュレーション用のユニファイド ライブラリ コンポーネントが含まれています。\$XILINX/verilog/src/simprims には、汎用シミュレーション プリミティブが含まれています。

タイミング シミュレーション、マップ後のシミュレーション、および変換後のシミュレーションには、SIMPRIM ベースのライブラリが使用されます。コマンドラインに次のように入力します。

```
irun -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v
\+libext+.v <testfixture>.v <design>.v
```

SDF ファイルのアノテーションについては、「[SDF ファイルからバックアノテートされた遅延値](#)」を参照してください。

NC-Verilog でのシミュレーション (方法 2)

この方法では、共有のコンパイル済みライブラリを使用します。この方法でシミュレーションを実行する前に、ザイリンクス シミュレーション ライブラリを使用するシミュレータ用にコンパイルする必要があります。ザイリンクスでは、コンパイル ツールとして Compplib を提供しています。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

RTL シミュレーションでは、デザインの構成 (インスタンシエートされたザイリンクス プリミティブ、CORE Generator™ ソフトウェア コンポーネントなど) によって、次の例のように **hdl.var** および **cds.lib** ファイルでライブラリ マップを指定します。

cds.lib ファイルの例

```
# cds.lib DEFINE worklib worklib
```

hdl.var ファイルの例

```
# hdl.var DEFINE LIB_MAP ($LIB_MAP, + => worklib)
```

ライブラリを設定したら、デザインをコンパイルしてシミュレーションします。

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v <testfixture>.v <design>.v
ncelab -messages <testfixture_name> glbl
ies -messages <testfixture_name>
```

ncvlog の **-update** オプションを使用すると、インクリメンタルなコンパイルをイネーブルにできます。

SDF ファイルからバックアノテートされた遅延値

NC-Verilog では、コンパイル済みの SDF ファイルしか読み込まれません。SDF ファイルは、NetGen の **\$sdf_annotate** タスクの引数として指定します。NetGen の詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

SDF ファイルのタイミング情報をアノテートするには、NCSDFC を使用します。

```
ncsdfc sdf_filename.sdf
```

NCSDFC を実行すると、sdf_filename.sdf.X というファイルが作成されます。コンパイル済みのファイルが存在する場合は、コンパイル済みのファイルの日付けがソース ファイルの日付けより後で、コンパイル済みのファイルのバージョンが NCSDFC のバージョンと一致していることを確認してください。コンパイル済みのファイルの日付けが古い場合またはバージョンが一致していない場合は、SDF ファイルが再コンパイルされます。

バックアノテートされたシミュレーションには、SIMPRIM ベースのライブラリが使用されます (合成後のシミュレーションを除く)。コマンドラインに次のように入力します。

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v <testfixture>.v time_sim.v
ncelab -messages -autosdf <testfixture_name> glbl
ies -messages <testfixture_name>
```

NC-Verilog を使用した SecureIP のシミュレーション

このセクションでは、NC-Verilog を使用した SecureIP のシミュレーションについて説明します。SecureIP の詳細は、「[SecureIP モデルの暗号化手法](#)」を参照してください。

11.1 リリースから、すべてのハード IP は SecureIP を使用して暗号化されています。サポートされる IES のバージョンは、「[サポートされるシミュレータおよび OS](#)」を参照してください。

コンパイル済みライブラリを使用する方法 (複数段階プロセス)

1. Compxlib を実行して SecureIP ライブラリを含むザイリンクス ライブラリをコンパイルします。
Compxlib はすべてのライブラリをコンパイルし、ライブラリのマップ情報で CDS.lib および HDL.var ファイルをアップデートします。
Compxlib の詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。
2. **ncvlog**、**ncelab**、および **IES** を実行します。
CDS.lib および HDL.var ファイルのマップ情報に基づいて、SecureIP ライブラリが自動的に参照されます。コマンドライン オプションや環境設定は必要ありません。

1 段階で実行する方法

1 段階で実行する場合、Compxlib を実行してザイリンクス ライブラリをコンパイルする必要はありません。ただし、次のコマンドライン オプションが必要になります。

```
-f $XILINX/secureip/ies/ies_secureip_cell.list.f
```

Example

```
irun \  
design>.v testbench>.v \  
${Xilinx}/verilog/src/glbl.v \  
-f $XILINX/secureip/ies/ies_secureip_cell.list.f \ \b>  
-y ${Xilinx}/verilog/src/unisims +libext+.v \  
-y ${Xilinx}/verilog/src/simprims +libext+.v \  
+access+r+w
```

SecureIP のシミュレーションに関してヘルプが必要な場合は、<http://japan.xilinx.com/support> からウェブケースを開いてください。

NC-VHDL でのシミュレーション

シミュレーションを実行する前に、Compxlib を使用してザイリンクス シミュレーション ライブラリを使用するシミュレータ用にコンパイルする必要があります。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

デザインの構成 (たとえば RTL シミュレーションではインスタンス化されたザイリンクス プリミティブ、CORE Generator™ ソフトウェア コンポーネント) によって、次の例のように **hdl.var** および **cds.lib** ファイルでライブラリ マップを指定します。

cds.lib ファイルの例

```
# cds.lib DEFINE worklib worklib
```

hdl.var ファイルの例

```
# hdl.var DEFINE LIB_MAP ($LIB_MAP, + => worklib)
```

NC-VHDL でのビヘイビア シミュレーション

ライブラリを設定したら、デザインをコンパイルしてシミュレーションします。

```
ncvhd1 <testbench>.vhd <design_name>.vhd ncelab -lib_binding -vhd1_time_precision lps -work worklib  
-cdslib cds.lib -access +wc worklib.testbench:behavior ies -extassertmsg -gui -cdslib cds.lib  
worklib.<testbench>:<architecture_name>
```

NC-VHDL でのタイミング シミュレーション

タイミング シミュレーションでは、SDF ファイルをコンパイルし、**ncelab** 行に追加する必要があります。

SDF ファイルをコンパイルするには、次のコマンドを実行します。

```
ncsdfc <name_sdf_file>
```

このコマンドを実行すると、コンパイル済みの SDF ファイル **<name_of_sdf_file>.x** が生成されます。コンパイル済みのファイルが存在する場合は、コンパイル済みのファイルの日付けがソース ファイルの日付けより後で、コンパイル済みのファイルのバージョンが NCSDFC のバージョンと一致していることを確認してください。

ncelab には、SDF ファイルを指定する **-SDF_CMD_FILE** *<file_name>* というオプションがあります。

```
// SDF command file sdf_cmd1 COMPILED_SDF_FILE = "dcmt_timesim_vhd.sdf.X", SCOPE = :uut,  
MTM_CONTROL = "MAXIMUM", SCALE_FACTORS = "1.0:1.0:1.0", SCALE_TYPE = "FROM_MTM"; // END OF FILE: sdf_cmd
```

SDF ファイルの情報が正しくアノートされたら、**ncelab** を次のように変更します。

```
ncelab -vhd1_time_precision lps -work worklib -cdslib cds.lib -SDF_CMD_FILE <file_name> -access  
+wc worklib.<testbench>:<architecture_name>
```

IUS5.5 以降を使用している場合は、次のコマンドを使用します。

```
ncelab -lib_binding -vhd1_time_precision lps -work worklib -cdslib cds.lib  
-SDF_CMD_FILE <file_name> -access +wc worklib.<testbench>:<architecture_name>
```

VCS および VCS MX でのザイリンクス デザインのシミュレーション

この付録では、Synopsys 社の VCS および VCS MX を使用したザイリンクス デザインのシミュレーションについて説明します。次のセクションが含まれています。

- ・ ISE® Design Suite からの VCS および VCS MX の実行
- ・ VCS および VCS MX でのザイリンクス デザインのシミュレーション (スタンドアロン)
- ・ VCS を使用した SecureIP のシミュレーション

ISE® Design Suite からの VCS および VCS MX の実行

Synopsys 社の VCS および VCS MX は、ISE® Design Suite には統合されていません。

VCS および VCS MX でのザイリンクス デザインのシミュレーション (スタンドアロン)

このセクションでは、VCS および VCS MX をスタンドアロンで使用したザイリンクス デザインのシミュレーションについて説明します。次の内容が含まれます。

- ・ コンパイル時間のオプションを含むライブラリ ソース ファイルの使用
- ・ 共有のコンパイル済みライブラリの使用
- ・ ユニファイド使用モデルの使用 (3 段階プロセス)

コンパイル時間のオプションを含むライブラリ ソース ファイルの使用

RTL シミュレーションでは、デザインの構成 (インスタンス化されたプリミティブ、CORE Generator™ コンポーネントが含まれるかどうか) によって、コマンドラインで次を入力します。

```
vcs -y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/xilinxcorelib \
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \
-Mupdate -R <testfixture>.v <design>.v
```

タイミング シミュレーションでは、SIMPRIM ベースのライブラリが使用されます。コマンドラインに次のように入力します。

```
vcs +compsdf -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
+libext+.v -Mupdate -R <testfixture>.v time_sim.v
```

タイミング シミュレーションでの SDF ファイルのバックアノテーションについては、「VCS での SDF ファイルの使用」を参照してください。

-R オプションを使用すると、コンパイル後に実行ファイルが自動的にシミュレーションされます。

-Mupdate オプションを使用すると、インクリメンタルなコンパイルが可能になります。モジュールは、次のいずれかの場合に再コンパイルされます。

- ・ 階層の参照ターゲットが変更された。
- ・ パラメータなどのコンパイル時間の定数が変更された。
- ・ モジュールにインスタンス化されているモジュールのポートが変更された。
- ・ モジュールがインライン処理されている。たとえば、VCS 内で複数のモジュール定義を 1 つのモジュール定義に結合すると、シミュレーションが速くなります。これらのモジュールは、再コンパイルされます。この操作は、1 度のみ実行されます。

共有のコンパイル済みライブラリの使用

論理シミュレーションを実行する前に、Compplib を使用してザイリンクス シミュレーション ライブラリを使用するシミュレータ用にコンパイルする必要があります。詳細は、『[コマンドライン ツール ユーザー ガイド](#)』を参照してください。

RTL シミュレーションでは、デザインの構成 (インスタンス化されたプリミティブ、CORE Generator™ コンポーネントが含まれるかどうか) によって、コマンドラインで次を入力します。

```
vcs -Mupdate -Mlib=<compiled_dir>/unisims_ver -y $XILINX/verilog/src/unisims \
-Mlib=<compiled_dir>/xilinxcorelib_ver - +incdir+$XILINX/verilog/src \
+libext+.v $XILINX/verilog/src/glbl.v -R <testfixture>.v <design>.v
```

タイミング シミュレーションまたは NGD2VER 後のシミュレーションには、SIMPRIM ベースのライブラリを使用します。コマンドラインに次のように入力します。

```
vcs +compsdf -Mupdate -Mlib=<compiled_lib_dir>/simprims_ver \
-y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v +libext+.v \
-R <testfixture>.v time_sim.v
```

タイミング シミュレーションでの SDF ファイルのバックアノテーションについては、『[VCS での SDF ファイルの使用](#)』を参照してください。

-R オプションを使用すると、コンパイル後に実行ファイルが自動的にシミュレーションされます。

-Mlib=<compiled_lib_dir> オプションを使用すると、VCS でモジュールをコンパイルする前にディスクリプタ情報を検索したり、実行ファイルをリンクさせる際にオブジェクト ファイルを取得するためのディレクトリを指定できます。

-Mupdate オプションを使用すると、インクリメンタルなコンパイルが可能になります。モジュールは、次のいずれかの場合に再コンパイルされます。

- ・ 階層の参照ターゲットが変更された。
- ・ パラメータなどのコンパイル時間の定数が変更された。
- ・ モジュールにインスタンス化されているモジュールのポートが変更された。
- ・ モジュールがインライン処理されている。たとえば、VCS 内で複数のモジュール定義を 1 つのモジュール定義に結合すると、シミュレーションが速くなります。これらのモジュールは、再コンパイルされます。この操作は、1 度のみ実行されます。

ユニファイド使用モデルの使用 (3 段階プロセス)

3 段階プロセスには、次のフェーズがあります。

- ・ 解析
- ・ エラボレーション
- ・ シミュレーション

3 段階プロセスの解析フェーズ

3 段階プロセスの解析フェーズでは、次が実行されます。

- ・ **vlogan [vlogan_options] file2.v file3.v file4.v**
最上位以外のすべての Verilog ファイルを解析します。
- ・ **vhdlan [vhdlan_options] file5.vhd file6.vhd**
VHDL の最下位エンティティから順に解析します。

3 段階プロセスのエラボレーション フェーズ

3 段階プロセスのエラボレーション フェーズでは、次が実行されます。

vcs [vcs_options] entity

3 段階プロセスのシミュレーション フェーズ

3 段階プロセスのシミュレーション フェーズでは、次が実行されます。

simv [simv_options]

詳細は、VCS のインストール ディレクトリにある VCS ユーザー ガイド (VCS_HOME/doc/UserGuide/vcsmx_ug_uum.pdf) を参照してください。

VCS での SDF ファイルの使用

SDF ファイルから遅延値をバックアノテートするには、次の 2 つの方法があります。

- ・ コンパイル時に SDF ファイルをコンパイル
- ・ ランタイムに ASCII SDF ファイルを読み込む

コンパイル時に SDF ファイルをコンパイル

コンパイル時に SDF ファイルをコンパイルするには、次のように **+compsdf** オプションを使用します。

vcsci -R -f options.f +compsdf

デフォルトでは、最上位シミュレーション ネットリストと同じ名前の SDF ファイルが使用されます。別の SDF ファイルを使用するには、**+compsdf** オプションの後に SDF ファイルを指定してください。コマンドラインにテーブル ファイルは必要ありません。VCS では、必要な機能が自動的に判断されます。

ランタイムに ASCII SDF ファイルを読み込む

ランタイムに ASCII SDF ファイルを読み込むには、次の手順に従い、**-P** オプションを使用してテーブル ファイルを指定します。

1. **\$sdf_annotate** システム タスクを C 関数 **sdf_annotate_call** にマップする PLI テーブル ファイル (sdf.tab) を作成します。
2. 次のように、**-P** オプションを使用して作成したテーブル ファイルを指定します。

```
vcs -P sdf.tab -y $XILINX/verilog/src/simprims +libext+.v  
time_sim.v
```

次に、sdf.tab ファイルの入力例を示します。

```
$sdf_annotate call=sdf_annotate_call acc+=tchk, mp, mipb:%CELL+
```

VCS を使用した SecureIP のシミュレーション

このセクションでは、VCS を使用した SecureIP のシミュレーションについて説明します。次の内容が含まれています。

- ・ [VCS を使用した SecureIP のシミュレーションについて](#)
- ・ [コンパイル時間のオプションを含むライブラリ ソース ファイルの使用](#)
- ・ [タイミング シミュレーションでの SIMPRIM ライブラリの使用](#)

VCS を使用した SecureIP のシミュレーションについて

ISE® Design Suite 11.1 リリースから、すべてのハード IP は SecureIP を使用して暗号化されています。サポートされる VCS のバージョンは、「[サポートされるシミュレータおよび OS](#)」を参照してください。

SecureIP の詳細は、「[SecureIP モデルの暗号化手法](#)」を参照してください。

コンパイル時間のオプションを含むライブラリ ソース ファイルの使用

RTL シミュレーションでは、デザインの構成 (インスタンス化されたプリミティブ、CORE Generator™ コンポーネントが含まれるかどうか) によって、コマンドラインで次を入力します。

```
vcs -f $XILINX/secureip/vcs/vcs_secureip_cell.list.f \  
-y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/xilinxcorelib \  
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \  
-Mupdate -R <testfixture>.v <design>.v
```

シミュレータで **-f** オプションを使用すると、コンパイル時に SecureIP ライブラリを使用できます。

タイミング シミュレーションでの SIMPRIM ライブラリの使用

SIMPRIM ライブラリは、タイミング シミュレーションで使用するライブラリです。コマンドラインに次のように入力します。

```
vcs +compsdf -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \  
-f $XILINX/secureip/vcs/vcs_secureip_cell.list.f \  
+libext+.v -Mupdate -R <testfixture>.v time_sim.v
```


SecureIP に SystemVerilog オプションを使用する場合は、[アンサー 32821](#) を参照してください。

SecureIP のシミュレーションに関してヘルプが必要な場合は、<http://japan.xilinx.com/support> からウェブケースを開いてください。