

Vivado Design Suite ユーザー ガイド :

合成

UG901 (v2012.2) 2012 年 7 月 25 日



Notice of Disclaimer

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

© Copyright 2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

本資料は英語版 (v2012.2) を翻訳したもので、内容に相違が生じる場合には原文を優先します。

資料によっては英語版の更新に対応していないものがあります。

日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

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

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	改訂内容
2012 年 7 月 25 日	2012.2	初版

目次

改訂履歴	2
Vivado 合成	
概要	4
合成手法	4
プロジェクト モードの使用	5
フロアプランの表示とリソース統計のレポート	19
ロジックの解析	23
XST ストラテジ	26
非プロジェクト モードでの合成	27
付録 A：合成属性	
概要	29
サポートされる属性	29
付録 B：SystemVerilog サポート	
概要	36
特定のファイルで SystemVerilog を使用	36
データ型	37
プロセス	40
手続きプログラム代入文	42
タスクおよび関数	43
モジュールおよび階層	44
インターフェイス	45
付録 C：その他のリソース	
ザイリンクス リソース	48
ソリューション センター	48
Vivado 資料	48

Vivado 合成

概要

合成は、RTL で記述されたデザインをゲート レベル記述に変換するプロセスです。Vivado™ 統合設計環境 (IDE) の合成はタイミングドリブンであり、メモリ使用量およびパフォーマンスで最適化されています。SystemVerilog および VHDL と Verilog の混合もサポートされています。Vivado IDE では、業界標準の Synopsys Design Constraints (SDC) に基づくザイリンクス デザイン制約 (XDC) がサポートされています。



重要 : UCF 制約は、Vivado 合成ではサポートされません。UCF 制約は XDC 制約に変換する必要があります。詳細は、『Vivado Design Suite 移行手法ガイド』(UG912) [\[参照 4\]](#) の「UCF 制約の XDC 制約への移行」を参照してください。

合成を設定して実行するには、次の 2 つの方法があります。

- Vivado IDE のプロジェクト モードを使用
- 非プロジェクト モードを使用 (Tel コマンド `synth_design` を実行し、デザイン ファイルをユーザーが制御)

操作モードの詳細は、『Vivado Design Suite ユーザー ガイド : デザイン フローの概要』(UG892) [\[参照 8\]](#) を参照してください。この章では、両方のモードを使用した合成を個別のセクションで説明します。

合成手法

Vivado IDE では、合成およびインプリメンテーション `run` をボタンをクリックするだけで実行可能な環境が提供されています。`run` のデータは自動的に管理され、さまざまな RTL ソース バージョン、ターゲット デバイス、合成およびインプリメンテーション オプション、物理制約およびタイミング制約を使用して繰り返し実行できます。Vivado IDE では、次の操作を実行できます。

- ストラテジを作成および保存。ストラテジとは、合成またはインプリメンテーションのデザイン `run` に適用されるコマンド オプションの設定です。詳細は、[7 ページの「run ストラテジの作成」](#)を参照してください。
- 複数の合成およびインプリメンテーション `run` を設定し、順次に、またはマルチプロセッサ マシンで同時に実行。詳細は、[13 ページの「合成の実行」](#)を参照してください。
- 合成またはインプリメンテーションの進捗状況を監視、ログ レポートを確認、`run` をキャンセル。詳細は、[17 ページの「合成実行の監視」](#)を参照してください。

プロジェクト モードの使用

このセクションでは、Vivado IDE を使用して Vivado 合成を設定および実行する方法を説明します。

合成設定

デザインの合成オプションを設定するには、次の手順に従います。

1. Flow Navigator で [Synthesis] → [Synthesis Settings] をクリックします (図 1)。

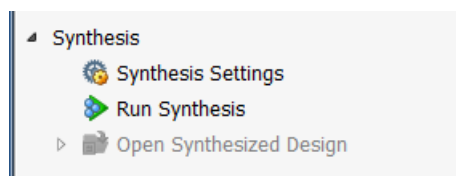


図 1 : Flow Navigator : [Synthesis] セクション

図 2 に示す [Project Settings] ダイアログ ボックスが開きます。

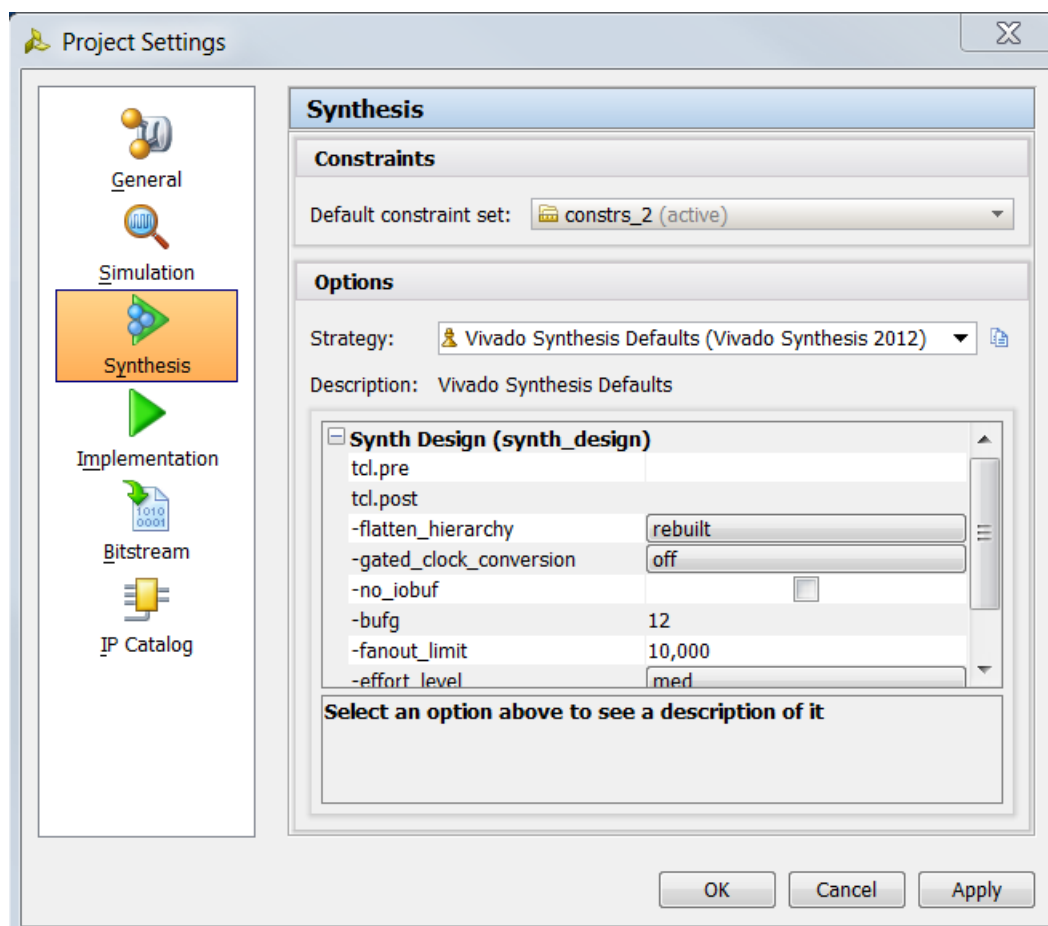


図 2 : [Project Settings] ダイアログ ボックス

2. [Project Settings] ダイアログ ボックスで、次のように設定します。

- a. [Synthesis] ページの [Constraints] で、[Default Constraint Set] にアクティブな制約セットとなる制約セットを選択します。制約セットは、ザイリンクス デザイン制約 (XDC) で記述されたデザイン制約を含む複数の制約ファイルのセットです。デザイン制約には、次の 2 種類があります。
 - 物理制約: ピン配置、ブロック RAM、LUT、フリップフロップなどのセルの絶対配置または相対配置、およびデバイスのコンフィギュレーション設定を定義します。
 - タイミング制約: 業界標準の SDC で記述し、デザインの周波数要件を定義します。タイミング制約を設定しない場合、デザインがワイヤの長さおよび配線の密集度により最適化されます。

選択した制約セットは新しい run に使用され、デザインの変更もこの制約セットに保存されます。

- b. [Options] エリアで、[Strategy] ドロップダウン リストから合成 run に使用する合成ストラテジを選択します。

あらかじめ定義されたストラテジから選択するか、または独自のストラテジを定義できます。合成ストラテジを選択すると、Vivado または XST のコマンド ライン オプションがダイアログ ボックスの下部に表示されます。オプションの値を変更すると、合成ストラテジの設定を変更できます。

図 3 では、[Strategy] ドロップダウン リストで [Vivado Synthesis Defaults] がハイライトされています。

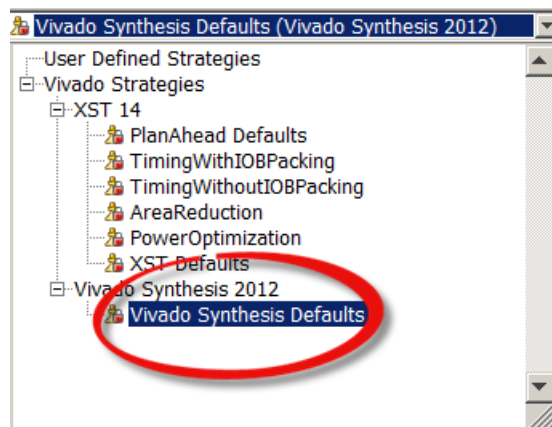


図 3 : [Strategy] ドロップダウン リストの [Vivado Synthesis Defaults] ストラテジ

Vivado 合成 2012 ストラテジを使用することをお勧めします。XST ストラテジについては、26 ページの「XST ストラテジ」を参照してください。

- c. 表示されているオプションを選択します。次のオプションがあります。
 - [tcl.pre] および [tcl.post]: 合成の前後に実行する Tcl ファイルを指定します。
 - [-flatten_hierarchy]: 合成での階層の制御方法を指定します。
 - [none]: 階層をフラット化しません。合成の出力には、元の RTL と同じ階層が含まれます。
 - [full]: 最上位以外の階層をすべてフラット化します。
 - [rebuilt]: これがデフォルトです。階層をフラット化して合成を実行した後に、元の RTL に基づいて階層を再構築します。この設定を使用すると、境界を越えた最適化を実行できるので QoR が向上し、最終的な階層は RTL と似たものになるので解析しやすくなります。
 - [-gated_clock_conversion]: ゲーテッド クロックをイネーブルに変換する機能をオン/オフにします。ゲーテッド クロックの変換を使用するには、RTL 属性も必要です。詳細は、付録 B 「SystemVerilog サポート」を参照してください。

- [-no_iobuf] : 入力または出力バッファが推論されないようにします。これは、ツールの出力がフローの後の方で下位として使用されるボトムアップ フローで有益です。この設定はグローバル設定であり、デザイン全体に適用されます。

この機能をポートごとに設定するには、付録 A 「合成属性」に説明されている BUFFER_TYPE 属性を使用します。

- [-bufg] : デザインで推論可能な BUFG の最大数を指定します。このオプションは、ネットリストのほかの BUFG が合成プロセスで認識されない場合に使用します。

RTL にインスタンス化されている BUFG の数が検出され、指定された数までの BUFG が推論されます。たとえば、-bufg オプションを 12 に設定し、RTL に 3 つの BUFG がインスタンス化されているとすると、あと 9 個の BUFG を推論可能です。

- [-fanout_limit] : 信号で駆動可能なロードの最大数を指定します。ロードの数がこれより大きくなる場合は、ロジックが複製されます。このグローバル設定は一般的なガイドラインであり、ツールで必要と判断された場合は無視されます。制限を強制する必要がある場合は、付録 A 「合成属性」に説明されている MAX_FANOUT を参照してください。
- [-fsm_extraction] : 有限ステート マシンの抽出およびマップ方法を指定します。デフォルトでは [off] に設定されており、ステート マシンはロジックとして合成されます。ステート マシンのエンコード タイプを [one_hot]、[sequential]、[johnson]、[gray]、または [auto] から選択できます。

run ストラテジの作成

ストラテジは、合成ツールおよびインプリメンテーションで実行されるさまざまなユーティリティやプログラムのオプションのあらかじめ定義されたセットです。ストラテジは、ツールおよびバージョン特定です。各メジャー リリースには、そのバージョン専用のストラテジがあります。

フローの現在のストラテジを確認するには、[Tools] → [Options] をクリックし、左側のペインで [Strategies] をクリックします (図 4)。

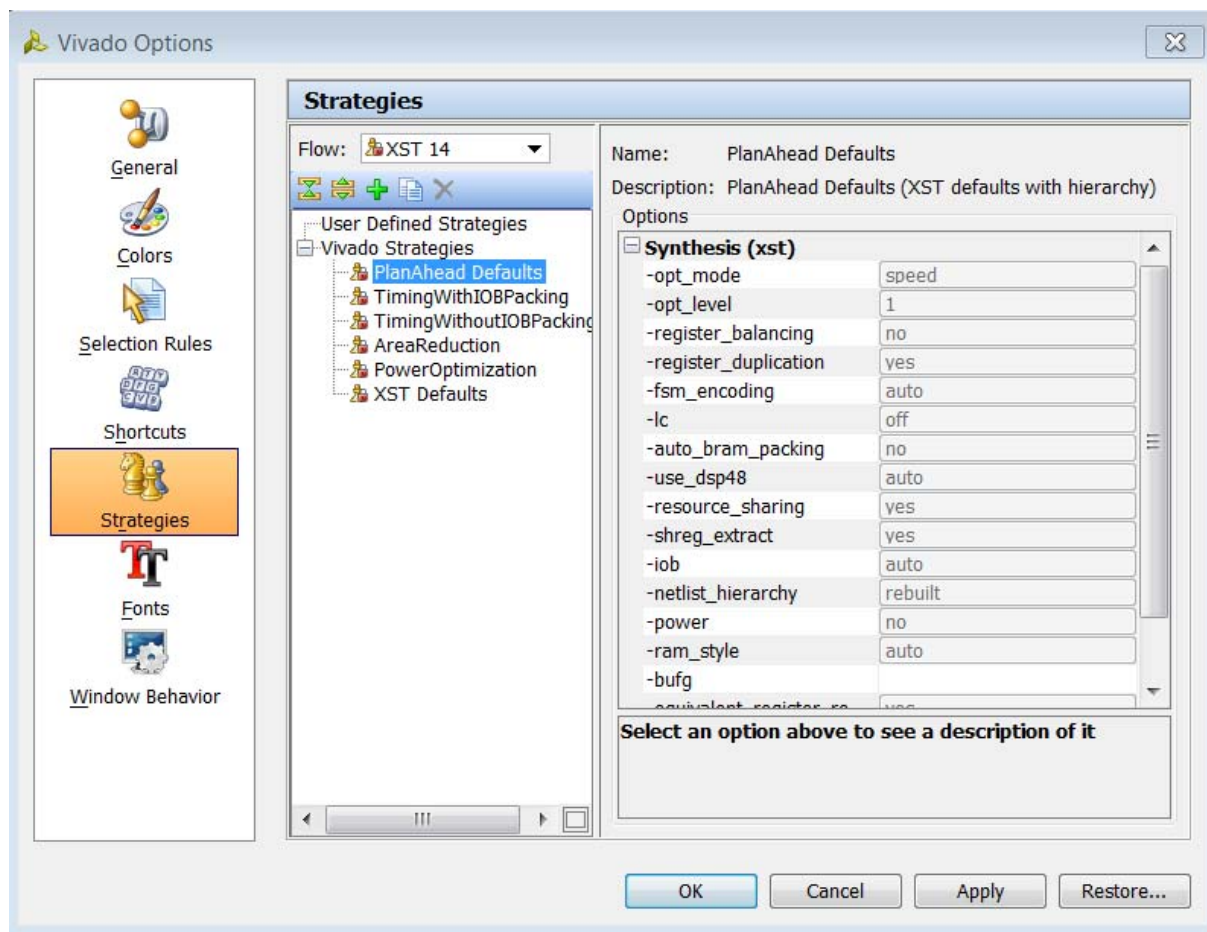



図 4 : 合成ストラテジ

[Flow] ドロップダウン リストから [Vivado Synthesis] を選択します。右側に表示されるオプションは、[Project Settings] ダイアログ ボックスの [Synthesis] ページに表示されるものと同じです。

カスタム ストラテジを作成するには、次のいずれかを実行します。

- [User Defined Strategies] を右クリックし、[Create New Strategy] をクリックします。
- ツールバーの [Create New Strategy] ボタン  をクリックし、[New Strategy] ダイアログ ボックス (図 5) を開きます。

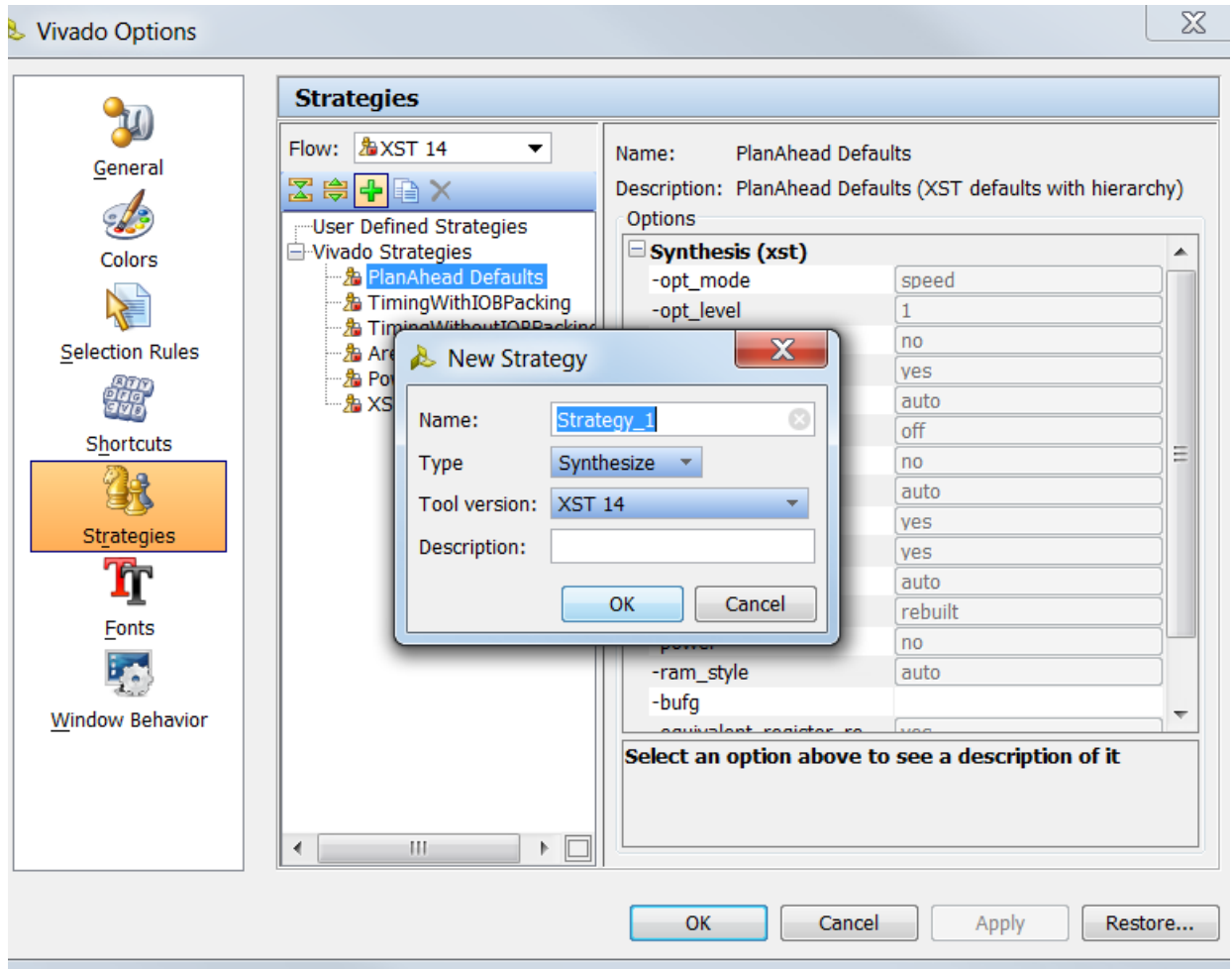


図 5 : [New Strategy] ダイアログ ボックス

[New Strategy] ダイアログ ボックスでストラテジの名前、ストラテジのタイプ、およびツール バージョンを指定します。説明も入力できます。オプションを設定したら [OK] をクリックします。

合成への入力

Vivado 合成には、RTL ソース コードおよびタイミング制約を入力できます。

RTL または制約ファイルを追加するには、Flow Navigator で [Project Manager] → [Add Sources] をクリックし、Add Sources ウィザード (図 6) を開きます。

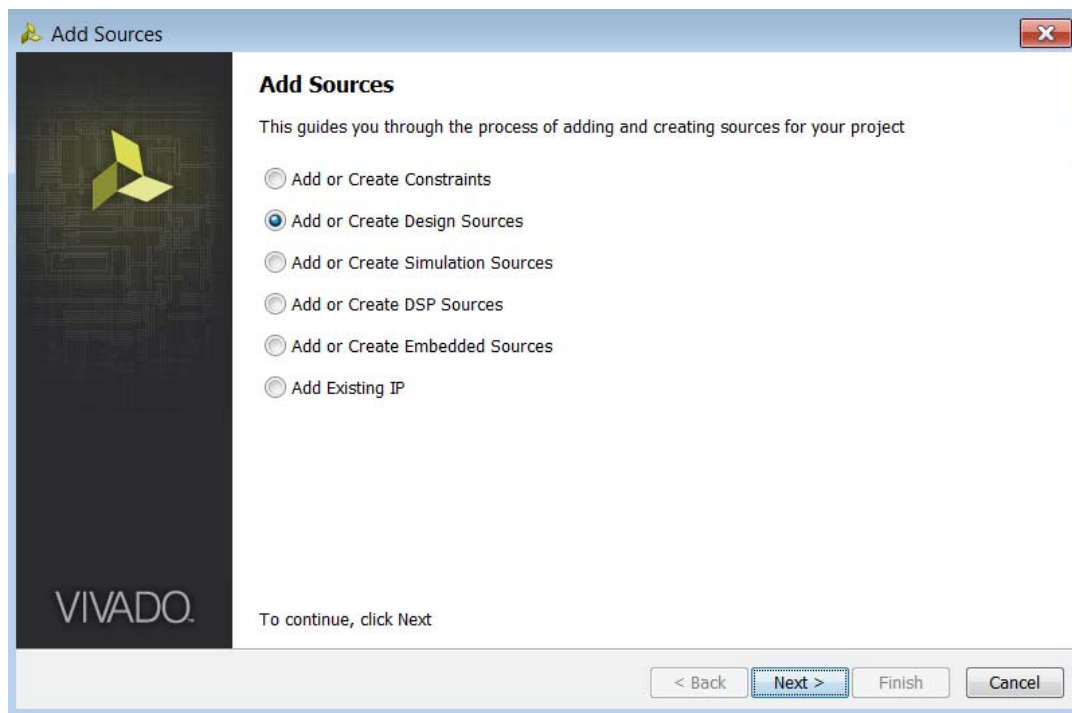


図 6 : Add Sources ウィザード

制約、RTL、またはその他のプロジェクト ファイルを追加します。Add Sources ウィザードの詳細は、『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [参照 5] を参照してください。

Vivado 合成では、ザイリンクス ツールでサポートされる VHDL、Verilog、または SystemVerilog のファイルの合成可能なサブセットを読み込むことができます。付録 B 「SystemVerilog サポート」に、サポートされる SystemVerilog コンストラクトが説明されています。

Vivado 合成では、合成での処理を制御するいくつかの RTL 属性もサポートされています。これらの属性は、付録 A 「合成属性」を参照してください。

Vivado 合成は、タイミング制約に XDC ファイルが使用されます。



重要 : Vivado Design Suite では、UCF フォーマットはサポートされません。UCF から XDC への変換手順は、『Vivado Design Suite 移行手法ガイド』(UG912) [参照 7] を参照してください。

ファイルのコンパイル順

あるファイルに宣言が含まれ、別のファイルがその宣言に依存している場合、特定のコンパイル順が必要になります。Vivado IDE では、RTL ソース ファイルのコンパイル順は、[Sources] ビューの [Compile Order] タブに上から下への順序で表示されます。

Vivado IDE では、最上位モジュールとして適切なモジュールが自動的に特定され、設定されます。コンパイル順も自動的に管理されます。アクティブ階層に含まれる最上位モジュール ファイルおよびすべてのソース ファイルが、合成およびシミュレーションで正しい順序で使用されます。

[Sources] ビューのポップアップ メニューには [Hierarchy Update] コマンドがあり、最上位モジュールへの変更、デザインのソース ファイルへの変更などの処理方法を指定して階層をアップデートできます。

デフォルト設定は [Automatic Update and Compile Order] で、次のように処理されます。

- [Compile Order] タブに示すようにコンパイル順が管理されます。
- [Hierarchy] タブにどのモジュールが使用され、階層ツリーのどこに位置するかが表示されます。

ソース ファイルを変更すると、コンパイル順が自動的に更新されます。

合成の前にコンパイル順を変更するには、次の手順に従います。

1. [Hierarchy Update] → [Automatic Update, Manual Compile Order] をクリックし、デザインに最適な最上位モジュールは自動的に選択され、コンパイル順は手動で指定できるようにします。
2. [Sources] ビューの [Compile Order] タブで、ファイルをドラッグするか、ポップアップ メニューの [Move Up] または [Move Down] コマンドを使用して、コンパイル順を変更します。

[Sources] ビューの詳細は、『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [\[参照 3\]](#) を参照してください。

グローバル インクルード ファイルの定義

Vivado IDE では、1 つまたは複数の Verilog または Verilog ヘッダー ファイルをグローバル インクルード ファイルとして指定できます。グローバル インクルード としてマークされているファイルは、ほかのソースの前に処理されます。

Verilog では通常、別の Verilog ファイルやヘッダー ファイルからの内容を参照する Verilog ソース ファイルの冒頭に ``include` 文を含める必要があります。共通ヘッダー ファイルを使用するデザインでは、複数の Verilog ソースにそれぞれ複数の ``include` 文を含める必要がある場合もあります。

Verilog ファイルまたは Verilog ヘッダー ファイルをグローバル インクルード ファイルとして指定するには、次のいずれかを実行します。

1. [Sources] ビューでファイルを右クリックします。
2. [Set Global Include] をクリックします。または、[Source Node Properties] ビューで [Global Include] チェックボックスをオンにします ([図 7](#))。

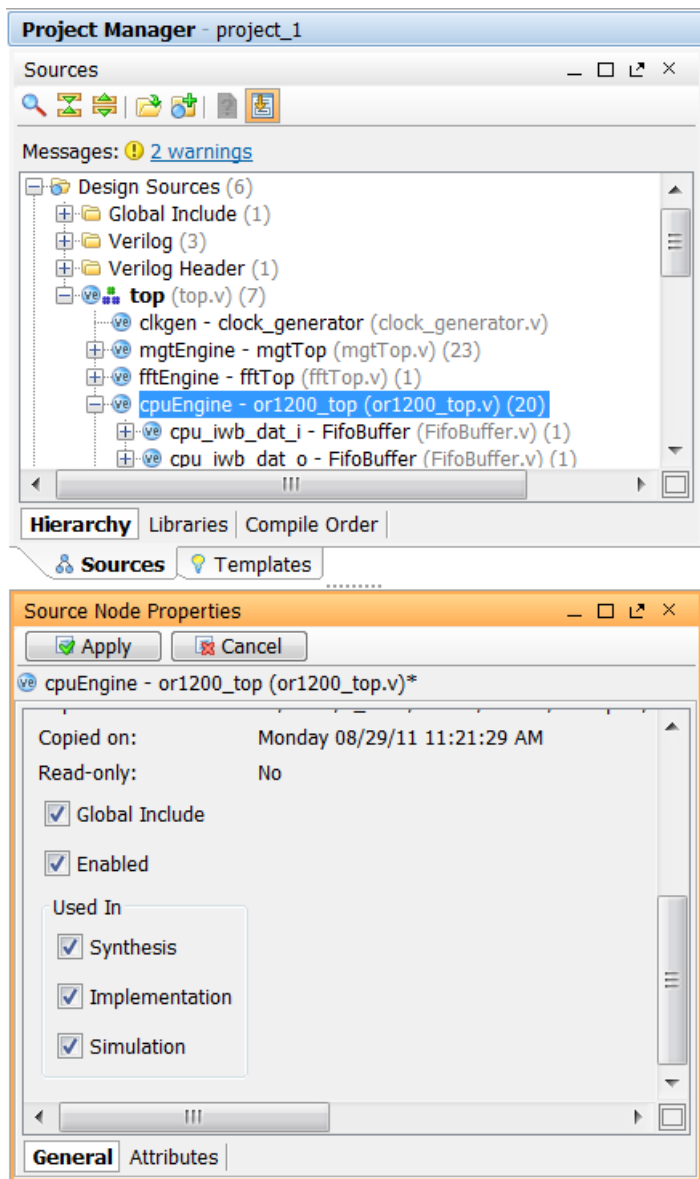


図 7 : [Source Node Properties] ビュー



ヒント : 1 つの Verilog ソースのみに適用する必要がある Verilog ヘッダー ファイル (特定の `define マクロなど) は、グローバル インクルード ファイルとして指定するのではなく `include 文を使用して参照する必要があります。

[Sources] ビューの詳細は、『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [参照 3] を参照してください。

合成の実行

合成 run では、合成中に使用されるデザインの詳細を定義および設定できます。合成 run は次を定義します。

- 合成中にターゲットとするザイリンクス デバイス
- 適用する制約セット
- 1 つまたは複数の合成 run を起動するオプション
- 合成エンジンの結果を制御するオプション

RTL ソース ファイルの run および制約を定義するには、次の手順に従います。

1. 次のいずれかを実行します。
 - [Flow] → [Create Runs] をクリック
 - Flow Navigator で [Run Synthesis] を右クリックして [Create Synthesis Runs] をクリック

Create New Runs ウィザードが開きます。ウィザードの最初のページは、コマンドのサマリです。

2. [Next] をクリックします。

図 8 に示す [Create New Runs] ページが表示されます。

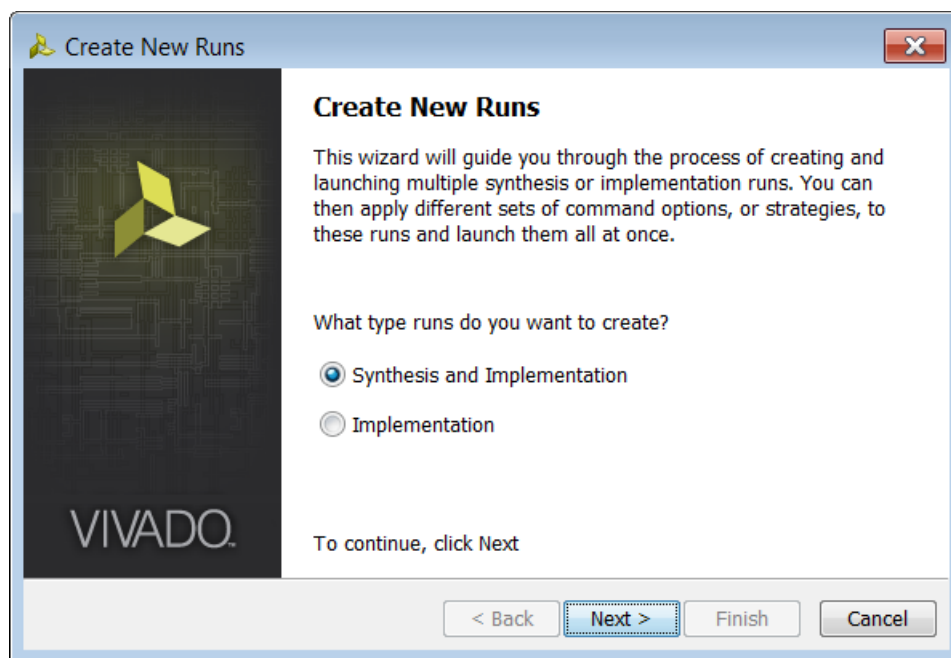


図 8 : [Create New Runs] ページ

3. [Synthesis and Implementation] をオンにし、[Next] をクリックします。

図 9 に示す [Set-Up Synthesis Runs] ページが表示されます。

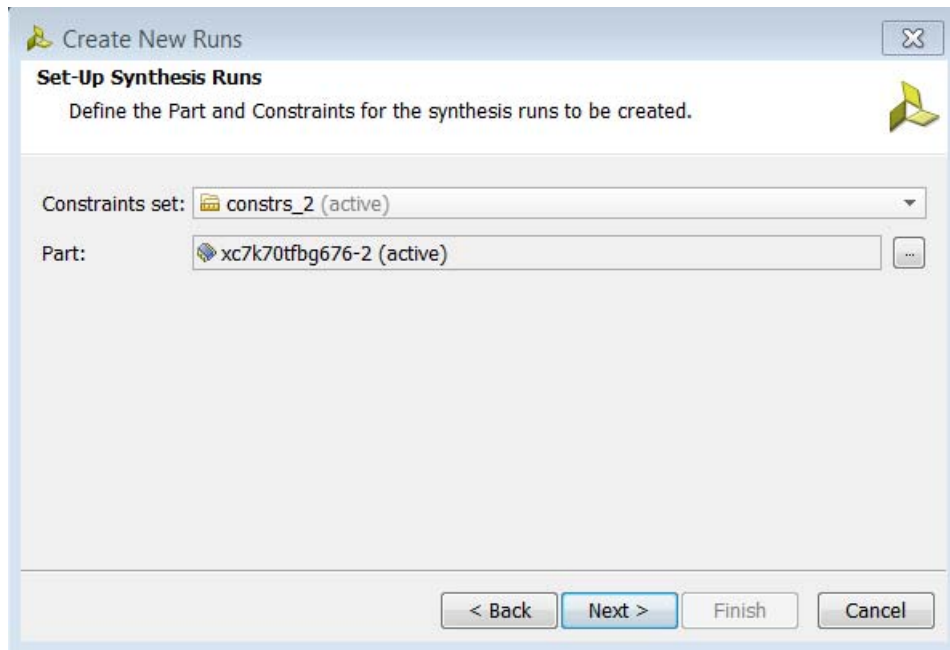


図 9 : [Set-Up Synthesis Runs] ページ

4. 制約セット ([Constraints set]) およびデバイス ([Part]) を選択し、[Next] をクリックします。

図 10 に示す [Choose Synthesis Strategies] ページが表示されます。

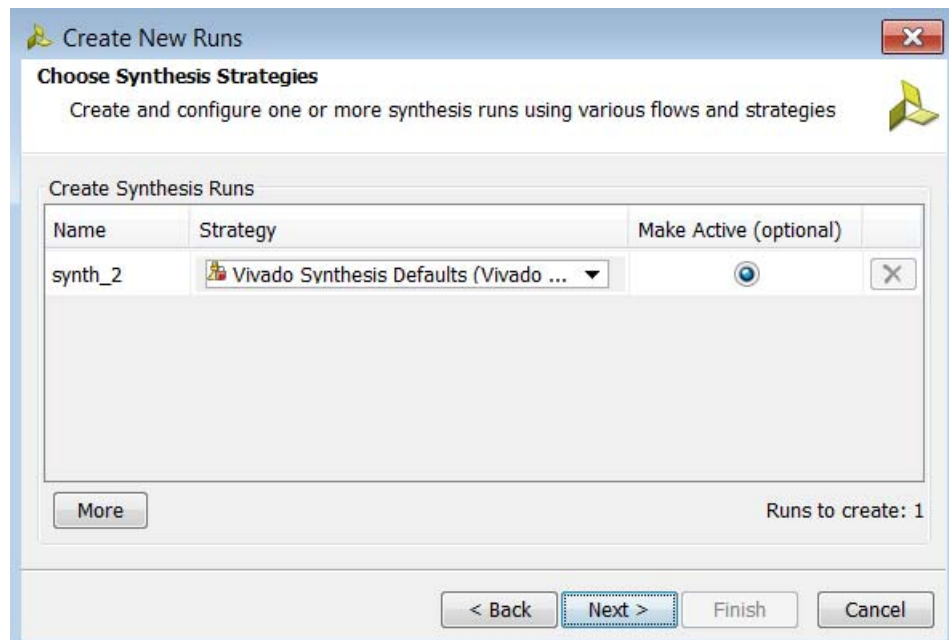


図 10 : [Choose Synthesis Strategies] ページ

5. 定義済みのストラテジを選択します。

Vivado IDE にはデフォルトのストラテジが含まれています。ストラテジ `run` の名前を指定するか、デフォルト名 (`synth_1`、`synth_2` など) を使用します。



ヒント : XST (Xilinx Synthesis Technology) のストラテジも選択可能です。XST のストラテジとレポートについては、[26 ページの「XST ストラテジ」](#)を参照してください。

独自のストラテジの作成方法は、[7 ページの「run ストラテジの作成」](#)を参照してください。

[Next] をクリックします。図 11 に示す [Launch Options] ページが表示されます。

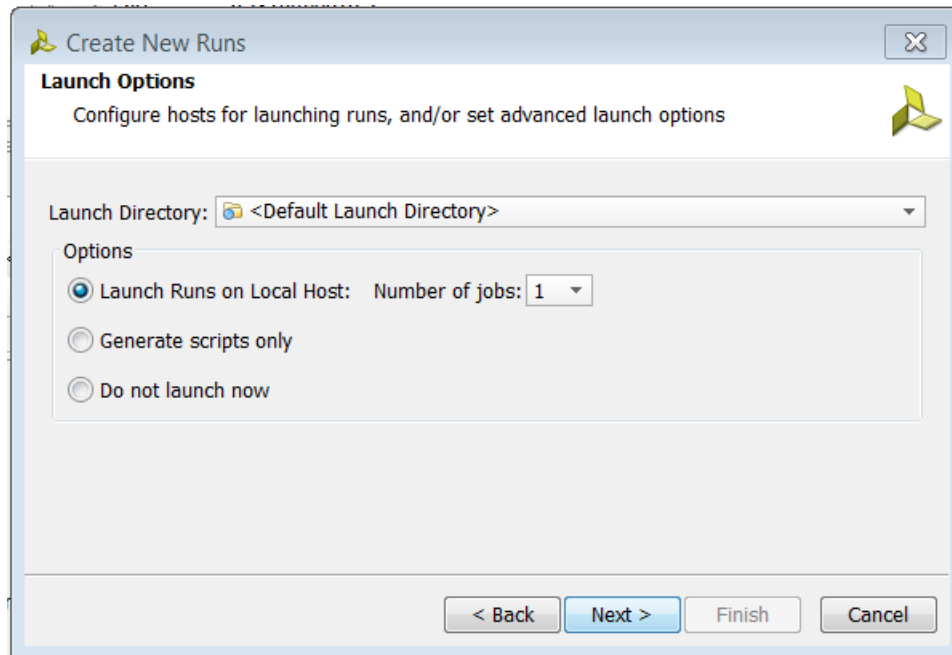


図 11 : [Launch Options] ページ

6. [Launch Options] ページで次のオプションを設定します。

- [Launch Directory] ドロップダウン リストから `run` を実行するディレクトリを選択します。
- [Options] エリアで次のいずれかをオンにします。
 - [Launch Runs on Local Host] : 作業中のマシンで `run` を実行します。[Number of jobs] で起動する `run` の数を指定します。
 - [Launch Runs on Remote Host] : Linux のみのオプションで、`run` をリモート ホストで実行し、そのホストを設定します。Linux のリモート ホストでの `run` の実行については、『Vivado Design Suite ユーザー ガイド : インプリメンテーション』(UG904) [\[参照 6\]](#) の付録 A 「リモート ホストの使用」を参照してください。[Configure Hosts] をクリックすると、ホストを設定するダイアログ ボックスが開きます。
 - [Generate scripts only] : 後で実行するスクリプトを生成します。`runme.bat` (Windows) または `runme.sh` (Linux) を使用して `run` を開始します。
 - [Do not launch now] : 前のページの設定を保存し、`run` を後で実行できるようにします。

Create New Runs ウィザードでオプションを設定して `run` を実行すると、[Design Runs] ビューに結果が表示されます (図 12)。

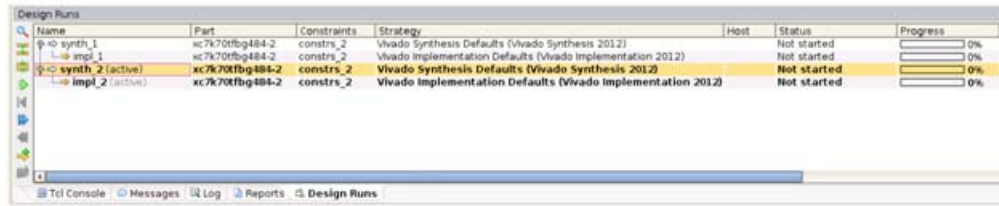


図 12 : [Design Runs] ビュー

[Design Runs] ビューの使用

[Design Runs] ビューには、プロジェクトで作成された合成 run とインプリメンテーション run のすべてが表示され、それらを設定、管理、実行するためのコマンドを実行できます。

[Design Runs] ビューが表示されていない場合は、[Window] → [Design Runs] をクリックします。

1 つの合成 run に、複数のインプリメンテーション run を含めることができます。プラス記号 (+) やマイナス記号 (-) をクリックすると、合成 run のツリー表示を展開したり、閉じたりできます。

[Design Runs] ビューには、run のステータスが実行されていないか、進行中か、完了したか、最新の状態でないかが示されます。

ソース ファイル、制約、またはプロジェクト設定を変更すると、run は最新の状態ではなくなります。特定の run をリセットまたは削除するには、run を右クリックして [Reset Runs] または [Delete] をクリックします。

アクティブ run の設定

Vivado IDE で一度にアクティブにできるには、1 つの合成 run と 1 つのインプリメンテーション run のみです。すべてのレポートおよびビューには、アクティブな run の情報が表示されます。[Project Summary] ビューには、アクティブな run のコンパイル、リソース、およびサマリ情報が表示されます。

別の run をアクティブにするには、[Design Runs] ビューで run を右クリックし、ポップアップ メニューから [Make Active] コマンドをクリックします。

合成 run の起動

合成 run を実行するには、次のいずれかを実行します。

- Flow Navigator で [Run Synthesis] をクリックします。
- メイン メニューから [Flow] → [Run Synthesis] をクリックします。
- [Design Runs] ビューで run を右クリックし、[Launch Runs] をクリックします。

最初の 2 つのオプションでは、アクティブな合成 run が実行されます。3 つ目のオプションでは、[Launch Selected Runs] ダイアログ ボックスが開きます。このダイアログ ボックスで、run をローカル ホストまたはリモート ホストで実行するか、あるいはスクリプトを生成するかを指定できます。

リモート ホストの使用については、『Vivado Design Suite ユーザー ガイド：インプリメンテーション』(UG904) [参照 6] の付録 A 「リモート ホストの使用」を参照してください。



ヒント : run を実行するたびに、別のプロセスが開始されます。メッセージを確認する際は、プロセス特定のものに注意してください。

プロセスのバックグラウンドへの移動

Vivado IDE で合成またはインプリメンテーションを実行すると、ダイアログ ボックスにプロセスをバックグラウンドで実行するオプションが表示されます。run をバックグラウンドに移動すると、レポートを表示するなど、Vivado IDE でほかの機能を実行できるようになります。

合成実行の監視

合成 run のステータスは、[Log] ビュー (図 13) で確認します。合成中にこのビューに表示されるメッセージは、合成ログ ファイルにも含まれます。

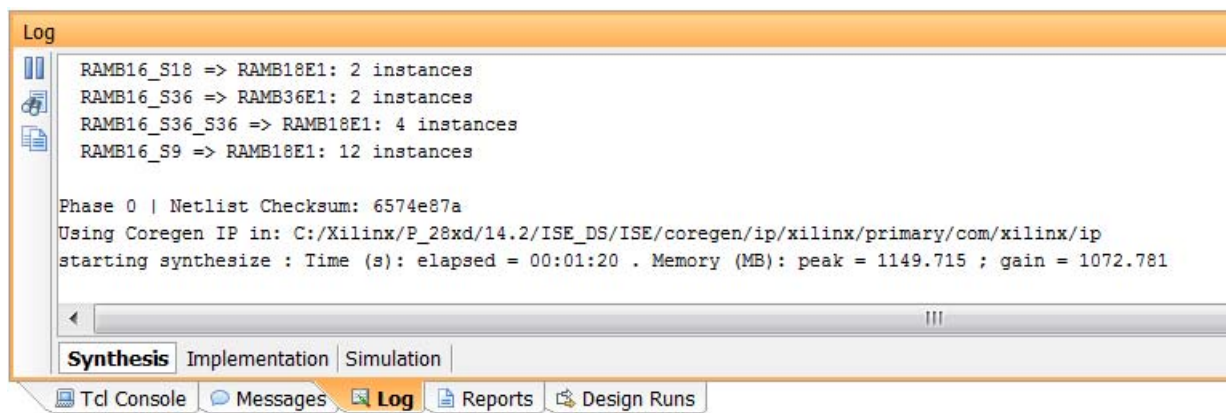


図 13 : [Log] ビュー

合成終了後のフロー

run が完了すると、[Synthesis Completed] ダイアログ ボックスが表示されます (図 14)。

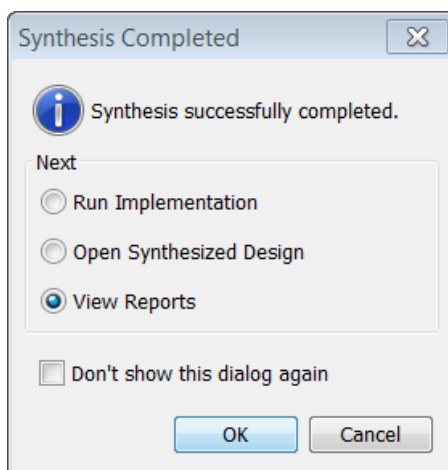


図 14 : [Synthesis Completed] ダイアログ ボックス

次のいずれかをオンにします。

- [Run Implementation] : インプリメンテーションを現在のインプリメンテーション プロジェクト設定を使用して実行します。
- [Open Synthesized Design] : 合成済みネットリスト、アクティブな制約セット、ターゲット デバイスを合成済みデザイン環境で開き、I/O ピン配置、デザイン解析、フロアプランを実行できるようにします。
- [View Reports] : [Reports] ビューを開き、レポートを表示できるようにします。

[Don't show this dialog again] をオンにすると、次回からこのダイアログ ボックスは表示されなくなります。



ヒント : このダイアログ ボックスを再び表示されるようにするには、[Tools] → [Options] をクリックし、左側のペインで [Window Behavior] をクリックします。

合成結果の解析

合成が終了したら、合成レポートを表示し、合成済みデザインを開いて解析できます。[Reports] ビューには、合成およびインプリメンテーションで生成されたレポートのリストが表示されます。

[Reports] ビュー (図 15) を開き、レポートを開いて特定の run の詳細を確認します。

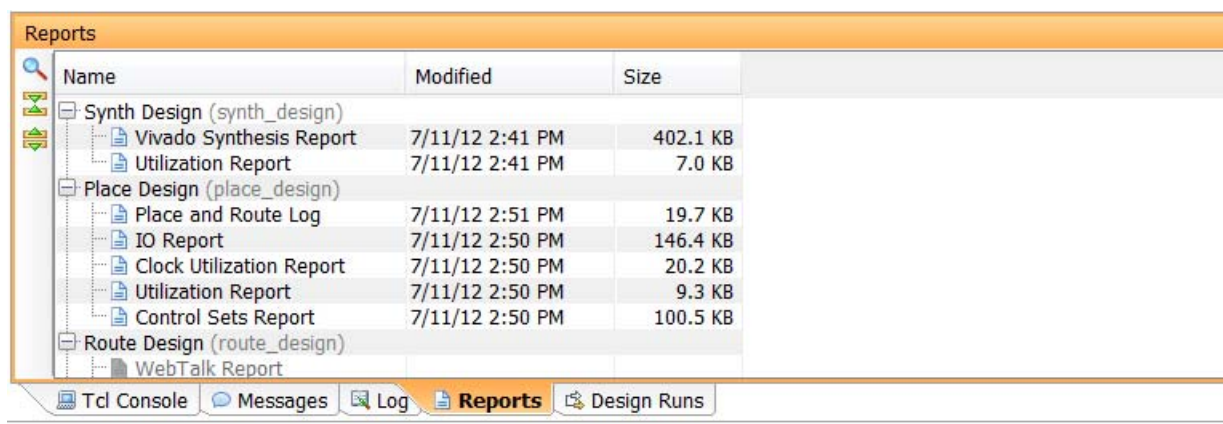


図 15 : [Reports] ビュー

合成済みデザイン環境の使用

Vivado IDE には、デザインをさまざまな面から解析する環境が含まれています。合成済みデザインを開くと、合成済みネットリスト、アクティブな制約セット、およびターゲット デバイスが読み込まれます。

詳細は、『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [参照 3] を参照してください。

合成済みデザインを開くには、次のいずれかを実行します。

- Flow Navigator で [Synthesis] → [Open Synthesized Design] をクリックします。
- メイン メニューから [Flow] → [Open Synthesized Design] をクリックします。

合成済みデザインを開くと、[Device] ビュー (図 16) が表示されます。

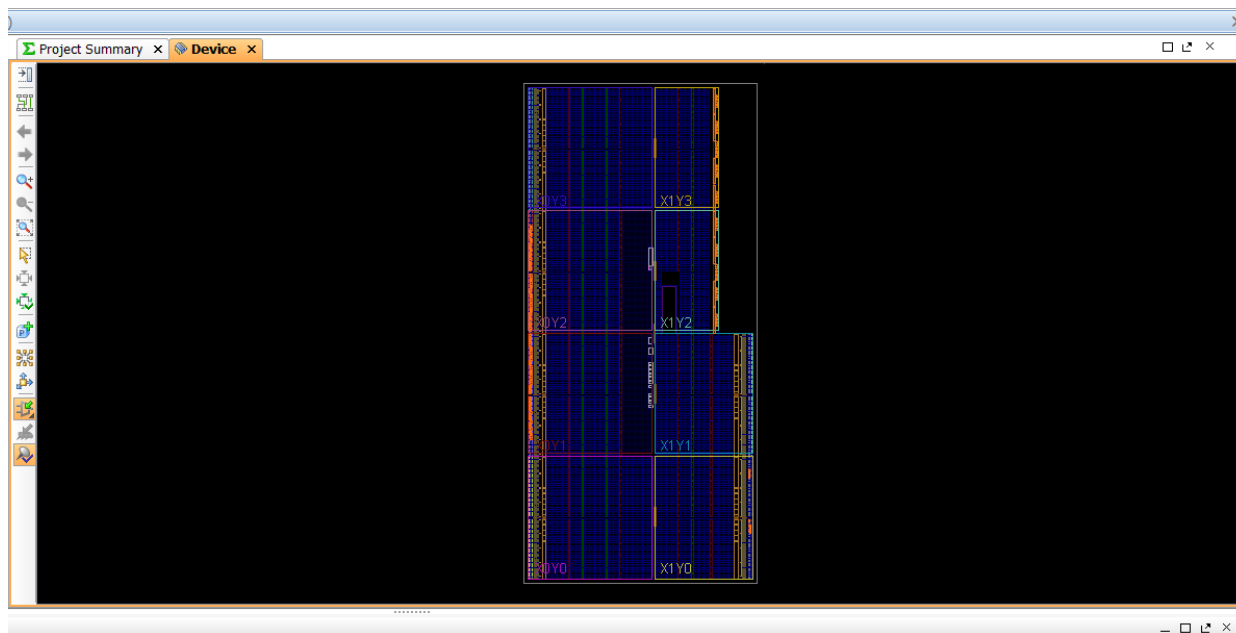


図 16 : [Device] ビュー

この環境から、デザイン ロジックと階層の確認、リソース使用率およびタイミング予測の表示、デザイン ルールチェック (DRC) を実行できます。

フロアプランの表示とリソース統計のレポート

リソース予測は、展開可能な階層ツリーとしてグラフィカルに表示されます。各リソース タイプを展開すると、論理階層の各レベルを表示できます。

デバイス リソース予測をグラフィカルに表示するには、合成済みデザインを開いて次のいずれかをクリックします。

- Flow Navigator → [Report Utilization]
- [Tools] → [Report Utilization]

[Utilization] ビュー (図 17) が開きます。

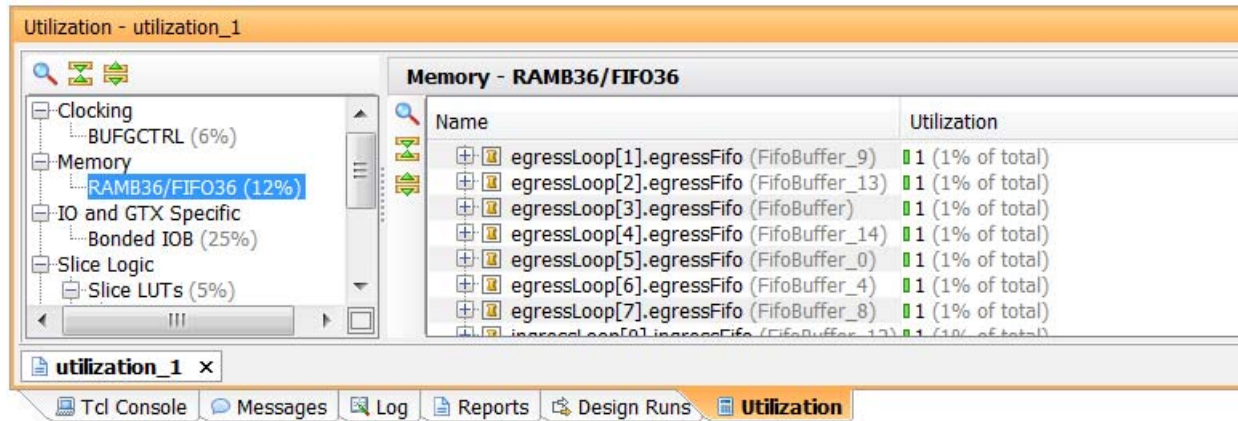


図 17 : [Utilization] ビュー

ロジック インスタンスのリソース統計の表示

Vivado IDE には、デザインに含まれるデバイス リソース数を予測する機能があります。

最上位を含むロジック インスタンスのリソース統計は、[Instance Properties] ビューに表示されます。[Netlist] ビューで最上位モジュールまたは任意のインスタンスを選択します (図 18)。

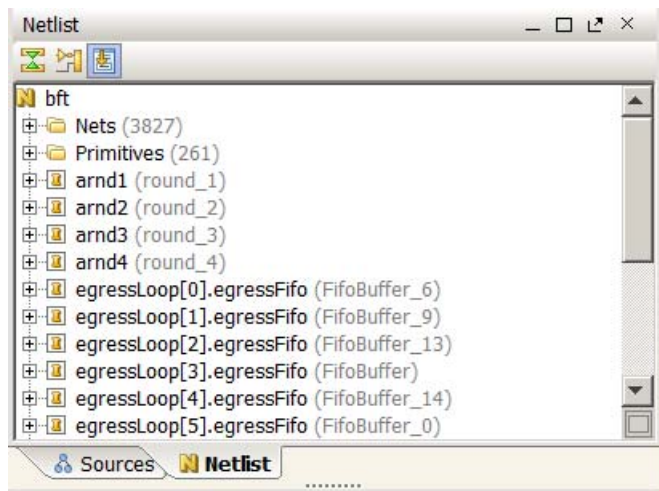


図 18 : [Netlist] ビュー

表示されない場合は、モジュールを右クリックし、ポップアップメニューで [Net Properties] または [Instance Properties] をクリックします。

[Netlist Properties] または [Instance Properties] ビューで [Statistics] タブをクリックします (図 19)。

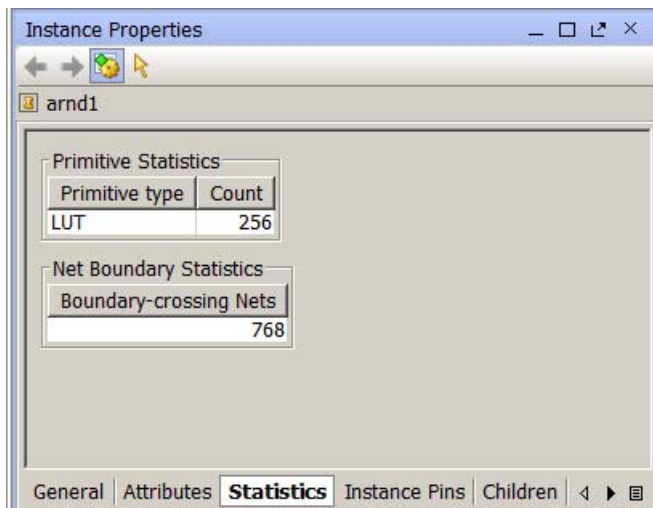


図 19 : [Instance Properties] ビュー

[Statistics] タブには、[Primitive Statistics] (プリミティブ統計) および [Net Boundary Statistics] (ネット境界統計) などの情報が表示されます。

[Instance Properties] ビューには、ほかに次のタブがあります。

- [General] : 選択したインスタンスの名前 ([Name])、セル ([Cell])、およびタイプ ([Type]) を示します。
- [Attributes] : ファイル属性をリストします。
- [Instance Pins] : インスタンス ピンの ID、名前、方向、BEL ピン、およびネットをリストします。
- [Children] : 子インスタンスの ID、名前、セル、およびインスタンス ピン数をリストします。
- [Nets] : ネットの ID、名前、インスタンス ピン、フラット ピン数、およびドライバーが存在するかを示します。
- [Power] : 信号レート ([Signal Rate]) および統計確率 ([Static Probability]) のスピン ボックス、階層のチェック ボックス ([Hierarchy]) を表示します。

リソース統計レポートのエクスポート

表示されているデータをスプレッドシート ファイルに保存できます。Vivado IDE では、階層形式のレポートが生成され、レポートする階層数およびレベル別に各モジュールの予測値を定義できます。

リソース統計レポートをエクスポートするには、[Netlist] ビューで統計を取得する階層を右クリックし、[Export Statistics] をクリックします (図 20)。

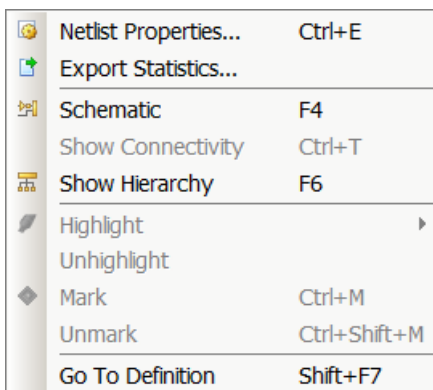


図 20 : [Netlist] ビューのポップアップ メニュー

図 21 に示す [Export Netlist Statistics] ダイアログ ボックスが開きます。

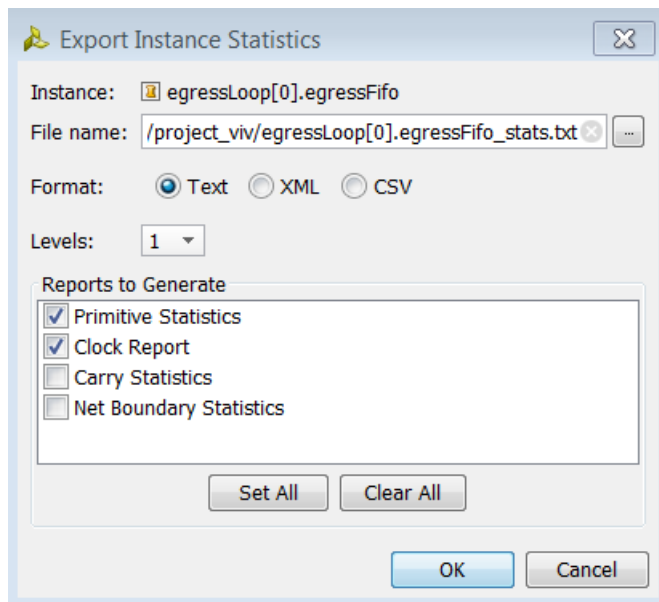


図 21 : [Export Instance Statistics] ダイアログ ボックス

[Export Instance Statistics] ダイアログ ボックスで、次の情報を指定します。

- [File Name]: スプレッドシート ファイルの名前と保存場所を入力します。
- [Format]: 出力ファイルのフォーマットをテキスト、XML、CSV から選択します。
- [Levels]: レポートに含める階層レベル数を設定します。
- [Reports to Generate]: 出力レポート ファイルに含めるネットリスト統計のタイプを定義します。

エクスポートするファイルのオプションを選択し、[OK] をクリックします。

ロジックの解析

Vivado IDE では、ロジックを解析するビューが複数あります。

- [Netlist] および [Hierarchy] ビューには、ナビゲート可能な階層ツリー形式の表示が含まれます。
- [Schematic] ビューでは、選択したロジックを展開したり階層表示にできます。
- [Device] ビューは、デバイス、配置ロジック オブジェクト、および接続をグラフィカルに表示します。

1 つのビューで選択した情報はほかのビューでも選択されるようになっており、必要な情報をすばやく見つけることができます。

ロジック階層の表示

[Netlist] ビューには、合成済みデザインのロジック階層が表示されます。ネットリスト内のロジック インスタンスまたはネットを、展開して選択できます。

別のビューでロジック オブジェクトを選択すると、[Netlist] ビューが自動的に展開されて選択したロジック オブジェクトが表示され、[Instance Properties] または [Net Properties] ビューにインスタンスまたはネットに関する情報が表示されます。

[Netlist] ビューのポップアップ メニューで [Show Hierarchy] をクリックすると表示される [Hierarchy] ビューには、RTL ロジック階層がグラフィカルに表示されます。各モジュールの大きさが、その他のモジュールに相対的な比率で表示されるので、選択したロジック モジュールのサイズや位置を判断できます。

ロジック回路図の解析

[Schematic] ビューでは、選択したロジックを展開して表示できます。[Schematic] ビューを表示するには、少なくとも 1 つのロジック オブジェクトを選択する必要があります。

[Schematic] ビューで、任意のロジックを選択および表示します。タイミング パスのグループを表示して、そのパス上のすべてのインスタンスを表示できます。これにより、タイミング クリティカルなモジュールが含まれる箇所を視覚的に表示できるので、フロアプランしやすくなります。

[Schematic] ビューを開くには、次の手順に従います。

1. 1 つまたは複数のインスタンス、ネット、タイミング パスを選択します。
2. ツールバーまたはポップアップ メニューで [Schematic] をクリックするか、F4 キーを押します。

[Schematic] ビューが開き、選択したモジュールが表示されます (図 22)。

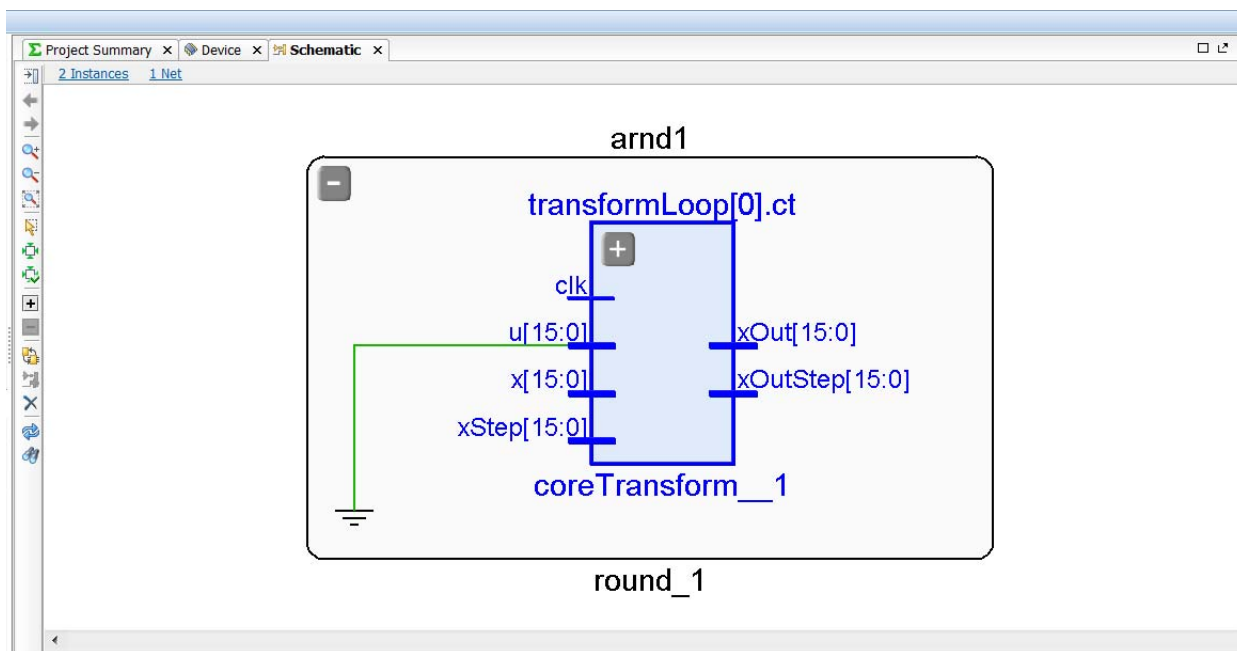


図 22 : [Schematic] ビュー

この後、ピン、インスタンス、階層モジュールを選択して、ロジックを展開できます。

タイミング解析の実行

合成済みデザインのタイミング解析は、パスにインプリメンテーションを効率的に実行するために必要な制約が設定されているかどうかを確認するのに有益です。Vivado 合成はタイミングドリブンであり、設定した制約に基づいて出力が調整されます。

Pblock や LOC 制約のような物理制約をデザインに割り当てていくと、より正確なタイミング解析結果が得られるようになりますが、これらの結果に含まれるのはパス遅延の予測値です。合成済みデザインでは、予測される配線遅延を使用して解析が実行されます。

この時点でタイミング解析を実行すると、パスが正しく制約されているか、およびタイミングパスの全体的な状況を確認できます。



重要：実際の配線遅延が含まれるのは、配置配線後のタイミング解析のみです。合成済みデザインのタイミング解析は、インプリメント済みデザインのタイミング解析ほど正確ではありません。

[Report Timing Summary] コマンドの使用

タイミング解析を実行するには、次のいずれかを実行します。

- [Tools] → [Timing] → [Report Timing Summary] をクリックします。
- Flow Navigator で [Synthesis] → [Synthesized Design] → [Report Timing Summary] をクリックします。

図 23 に示す [Report Timing Summary] ダイアログ ボックスが開きます。

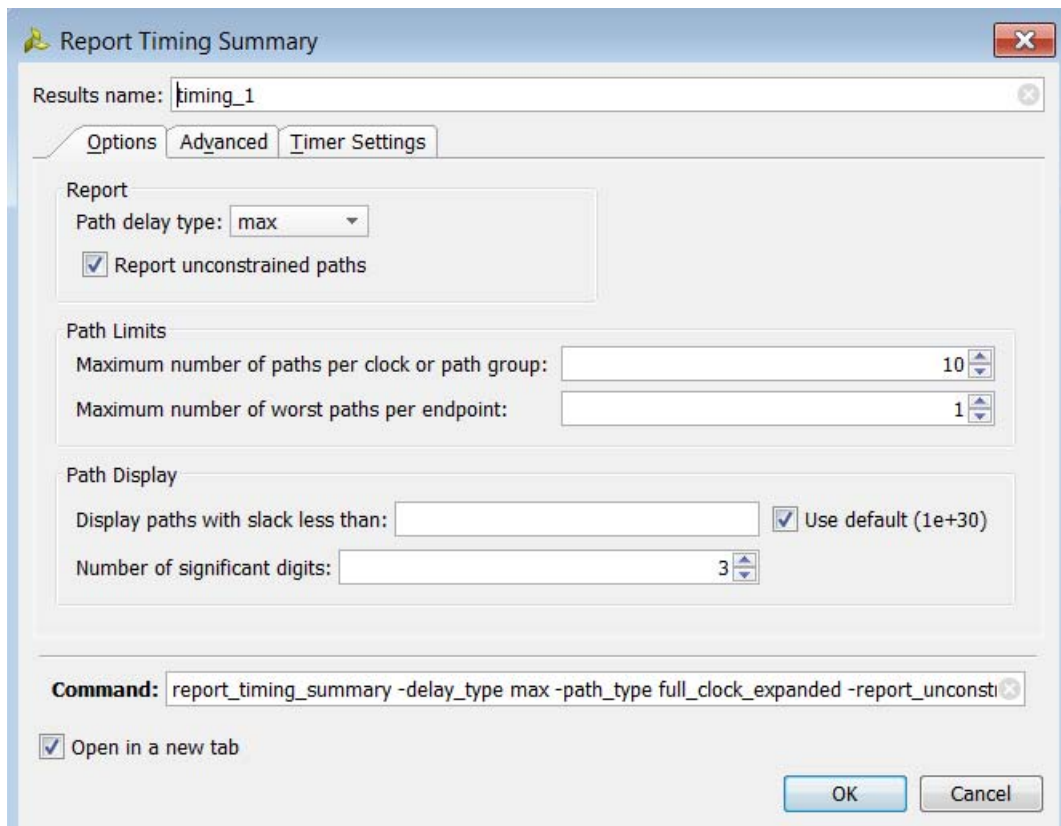


図 23 : [Report Timing Summary] ダイアログ ボックス

次のようにオプションを設定します。

- [Results name] : レポート結果の名前を指定します。
- [Report] : [Path delay type] (パス遅延タイプ) を [Max]、[Min]、または [Min_Max] に設定し、制約されていないパスをレポートするかどうかを [Report unconstrained paths] チェック ボックスで指定します。
- [Path Limits] : このセクションでは、次のオプションを設定します。
 - [Maximum number of paths per clock or path group] : クロックまたはパス グループごとにレポートするパスの最大数を指定します。
 - [Maximum number of worst paths per endpoint] : エンドポイントごとにレポートするパスの最大数を指定します。
- [Command] : 現時点での report_timing コマンドを選択されたオプションも含めて表示します。また、[Open in a new tab] チェック ボックスで別のタブに表示するかを指定します。

XST ストラテジ

Vivado Synthesis Defaults が推奨されるストラテジです。合成に XST を使用する場合は、あらかじめ定義された XST ストラテジを使用できます。主要な XST ストラテジは、次の 2 つです。

- PlanAhead Defaults : 再構築された階層ネットリストを生成します。
- XST Defaults : 階層の再構築をオフにします。

図 24 に、[Strategies] ドロップダウン リストに表示される [XST 14] → [PlanAhead Defaults] および [XST Defaults] を示します。



図 24 : Vivado の XST ストラテジ

XST オプションおよびレポートの詳細は、次の資料を参照してください。

- 『PlanAhead ユーザー ガイド』(UG632) [\[参照 9\]](#)
- 『XST ユーザー ガイド (Virtex-6、Spartan-6、7 シリーズ デバイス)』(UG687) [\[参照 10\]](#)

非プロジェクト モードでの合成

合成を実行する Tcl コマンドは `synth_design` です。通常このコマンドは、次の例のように複数のオプションを使用して実行します。

```
synth_design -part xc7k30tfbg484-2 -top my_top
```

この例では、`synth_design` が `-part` オプションおよび `-top` オプションを使用して実行されます。

Tcl コンソールから、Tcl コマンド オプションを使用して合成オプションを設定して合成を実行できます。[Tcl Console] ビューで「`synth_design -help`」と入力すると、オプションのリストを取得できます。次に、`-help` オプションを使用したときのコマンド出力の一部を示します。

```
synth_design -help
synth_design
Description:
Synthesize a design using Vivado Synthesis and open that design
Syntax:
synth_design  [-name <arg>] [-part <arg>] [-constrset <arg>] [-top <arg>]
               [-include_dirs <args>] [-generic <args>] [-verilog_define <args>]
               [-flatten_hierarchy <arg>] [-gated_clock_conversion <arg>]
               [-effort_level <arg>] [-rtl] [-no_iobuf] [-bufg <arg>]
               [-fanout_limit <arg>] [-fsm_extraction <arg>] [-quiet] [-verbose]
```

Returns:

design object

Usage:

Name	Description
-----	-----
[-name]	Design name
[-part]	Target part
[-constrset]	Constraint fileset to use
[-top]	Specify the top module name
[-include_dirs]	Specify verilog search directories
[-generic]	Specify generic parameters.Syntax: -generic <name>=<value> -generic <name>=<value> ...
[-verilog_define]	Specify verilog defines.Syntax: -verilog_define <macro_name>[=<macro_text>] -verilog_define <macro_name>[=<macro_text>] ...
[-flatten_hierarchy]	Flatten hierarchy during LUT mapping.Values: full, none, rebuilt Default: rebuilt
[-gated_clock_conversion]	Convert clock gating logic to flop enable. Values: off, on, auto Default: off
[-effort_level]	Synthesis effort level.Values: quick, med Default: med
[-rtl]	Elaborate and open an rtl design
[-no_iobuf]	Disable setting of I/O buffers
[-bufg]	Max number of global clock buffers used by synthesis.Default: 12
[-fanout_limit]	Fanout limit Default: 100
[-fsm_extraction]	FSM Extraction Encoding.Values: off, one_hot, sequential, johnson, gray, auto.Default: off
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

コマンドの詳細は、『Rodin Tcl コマンド リファレンス ガイド』(UG835) [\[参照 4\]](#) を参照してください。Vivado IDE での操作に対応する Tcl コマンドを確認するには、Vivado IDE でコマンドを実行し、[Tcl Console] ビューまたはログ ファイルを参照してください。

次に、synth_design の Tcl スクリプト例を示します。

```
# Setup design sources and constraints
read_vhdl -library bftLib [ glob ./Sources/hdl/bftLib/*.vhdl ]
read_vhdl ./Sources/hdl/bft.vhdl
read_verilog [ glob ./Sources/hdl/*.v ]
read_xdc ./Sources/bft_full.xdc
# Run synthesis, report utilization and timing estimates, write design checkpoint
synth_design -top bft -part xc7k70tfbg484-2 -flatten rebuilt
write_checkpoint -force $outputDir/post_synth
```

制約の設定

表 1 に、Vivado タイミング制約にサポートされる Tcl コマンドを示します。

これらのコマンドの詳細は、次の資料を参照してください。

- 『Vivado Design Suite ユーザー ガイド : Vivado IDE の使用』(UG893) [\[参照 3\]](#)
- 『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835) [\[参照 4\]](#)

表 1: サポートされる合成 Tcl コマンド

コマンド タイプ	コマンド			
タイミング制約	create_clock	create_generate_clock	set_false_path	set_input_delay
	set_output_delay	set_max_delay	set_multicycle_path	
	set_clock_latency	set_clock_groups	set_disable_timing	
オブジェクト アクセス	all_clocks	all_inputs	all_outputs	get_cells
	get_clocks	get_nets	get_pins	get_ports

合成属性

概要

Vivado™ 合成では、XST 合成属性の多くを使用できます。ほとんどの場合、これらの属性は同じ構文で、同じ動作になります。

- Vivado 合成で属性がサポートされる場合、その属性が使用され、その属性を反映したロジックが作成されます。
- 指定した属性がツールで認識されない場合、その属性と値は生成されたネットリストに渡されます。

認識されない属性は、フローのダウンストリームで使用されると想定されます。たとえば、LOC 制約は合成では使用されませんが、配置ツールで使用されるので、合成ツールから転送されます。

サポートされる属性

次のセクションに、サポートされる属性をリストします。

BLACK_BOX

BLACK_BOX 属性は、すべての階層レベルをオフにし、合成でそのモジュールまたはエンティティに対してブラックボックスを作成できるようにするデバッグ用の属性です。この属性を指定すると、モジュールまたはエンティティに対して有効なロジックがあったとしても、合成ツールでそのレベルに対してブラックボックスが作成されます。

BLACK_BOX の Verilog 例

```
(* black_box *) module test(in1, in2, clk, out1);
```

BLACK_BOX の VHDL 例

```
attribute black_box : string;  
attribute black_box of beh : architecture is "yes";
```

Verilog では、値は必要ありません。この属性があれば、ブラックボックスが作成されます。

BUFFER_TYPE

BUFFER_TYPE は入力に設定し、使用するバッファのタイプを指定します。

デフォルトでは、クロックには IBUF/BUFG または BUFGP が、入力には IBUF が使用されます。

有効な値は次のとおりです。

- `ibuf`: IBUF/BUFG ペアにクロック ポートが不要な場合に使用します。この場合にのみ、クロックに対して IBUF が推論されます。
- `none`: 入力または出力バッファを使用しないよう指定します。クロック ポートに `none` を指定すると、バッファは使用されません。

注記: XST では、`ibufg`、`bufr`、`bufgp`、および `bufg` などの値もサポートされます。

BUFFER_TYPE の Verilog 例

```
(* buffer_type = "none" *) input in1; //this will result in no buffers
(* buffer_type = "ibuf" *) input clk1; //this will result in a clock with no bufg
```

BUFFER_TYPE の VHDL 例

```
entity test is port(
  in1 : std_logic_vector (8 downto 0);
  clk : std_logic;
  out1 : std_logic_vector(8 downto 0));
attribute buffer_type : string;
attribute buffer_type of in1 : signal is "none";
end test;
```

DONT_TOUCH

DONT_TOUCH 属性は、KEEP または KEEP_HIERARCHY と同じように機能しますが、KEEP および KEEP_HIERARCHY とは異なり配置配線にフォワード アノテートされるので、ロジック最適化は実行されません。

注記: KEEP および KEEP_HIERARCHY 属性を DONT_TOUCH に置換します。

DONT_TOUCH の Verilog 例

```
(* dont_touch = "true" *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

DONT_TOUCH の VHDL 例

```
signal sig1 : std_logic
attribute dont_touch : string;
attribute dont_touch of sig1 : signal is "true";
....
....
sig1 <= in1 and in2;
out1 <= sig1 and in3;
```

FULL_CASE (Verilog のみ)

FULL_CASE は、可能性のあるすべての case 値が case、casex または casez 文で指定されることを示します。case 値が指定されている場合、Vivado 合成で case 値に対して余分なロジックは作成されません。

```
(* full_case *)
case select
3'b100 : sig = val1;
3'b010 : sig = val2;
3'b001 : sig = val3;
endcase
```



重要: この属性は、RTL でのみ制御できます。

GATED_CLOCK

Vivado 合成では、ゲートド クロックの変換が可能です。この変換を実行するには、次の 2 つの方法を使用できます。

- GUI のオプションで変換を試みるように指定します。
- RTL 属性でゲートド ロジックのどの信号がクロックかを指定します。

GUI のオプションを指定するには、次の手順に従います。

1. Flow Navigator で [Synthesis Settings] をクリックします。
2. [Options] フィールドで -gated_clock_conversion オプションを次のいずれかの値に設定します。
 - [off]: ゲートド クロックの変換をディスエーブルにします。
 - [on]: gated_clock 属性が RTL コードで設定されている場合に、ゲートド クロックの変換を実行します。この設定では、結果をより制御できます。
 - [auto]: 次のいずれかの条件が満たされる場合に、ゲートド クロックの変換を実行します。
 - gated_clock 属性が true に設定されている。
 - Vivado 合成でゲートが検出され、有効なクロック制約セットがある。

この設定では、ツールで自動的に判断されます。

GATED_CLOCK の Verilog 例 :

```
(* gated_clk = "true" *) input clk;
```

GATED_CLOCK の VHDL 例

```
entity test is port (
in1, in2 : in std_logic_vector(9 downto 0);
en : in std_logic;
clk : in std_logic;
out1 : out std_logic_vector( 9 downto 0));
attribute gated_clock : string;
attribute gated_clock of clk : signal is "true";
end test;
```

KEEP

KEEP 属性は、信号が削除されたりロジックブロックに吸収されるような最適化が実行されないように指定します。この属性が設定された信号は保持され、ネットリストに含まれます。たとえば、2 ビットの AND ゲートの出力で別の AND ゲートを駆動する信号に KEEP 制約を設定すると、信号は両方の AND ゲートを含むより大きい LUT には統合されません。

KEEP は、タイミング制約ともよく併用されます。通常は最適化される信号にタイミング制約が設定されている場合、KEEP を設定すると最適化されなくなり、正しいタイミング規則が使用されます。

有効な値は、次のとおりです。

- `true`: 信号を保持します。
- `false`: 信号が必要に応じて最適化されるようにします。`false` を使用しても、信号が無条件に削除されることはありません。デフォルトは `false` です。

注記: KEEP 属性を使用しても、配置配線では信号は保持されません。DONT_TOUCH 属性を使用してください。

KEEP の Verilog 例

```
(* keep = "true" *) wire sig1;  
assign sig1 = in1 & in2;  
assign out1 = sig1 & in2;
```

KEEP の VHDL 例

```
signal sig1 : std_logic  
attribute keep : string;  
attribute keep of sig1 : signal is "true";  
....  
....  
sig1 <= in1 and in2;  
out1 <= sig1 and in3;
```

KEEP_HIERARCHY

KEEP_HIERARCHY は、階層レベルが変更されないようにします。Vivado 合成では、RTL で指定されたのと同じ階層が保持されるよう試みられますが、QoR (結果の品質) のために階層がフラットにされたり、変更されることもあります。

インスタンスに KEEP_HIERARCHY を指定すると、合成でその階層レベルは変更されません。これが QoR に影響を与える場合があります。また、トリステート出力および I/O バッファの制御ロジックを記述するモジュールには使用しないでください。KEEP_HIERARCHY は、モジュール、アーキテクチャレベル、またはインスタンスに指定できます。

KEEP_HIERARCHY の Verilog 例

モジュールの場合

```
(* keep_hierarchy = "yes" *) module bottom (in1, in2, in3, in4, out1, out2);
```

インスタンスの場合

```
(* keep_hierarchy = "yes" *)bottom u0 (.in1(in1), .in2(in2), .out1(temp1));
```


KEEP_HIERARCHY の VHDL 例

モジュールの場合

```
attribute keep_hierarchy : string;  
attribute keep_hierarchy of beh : architecture is "yes";
```

インスタンスの場合

```
attribute keep_hierarchy : string;  
attribute keep_hierarchy of u0 : label is "yes";
```

MAX_FANOUT

MAX_FANOUT は、レジスタおよび信号のファンアウトの制限を設定します。信号に KEEP 制約も設定する必要があります。これは、RTL またはプロジェクトへの入力として指定できます。整数値を指定します。

- この属性を RTL で指定した場合はハード リミットになります。
- プロジェクトで指定した場合はソフト リミットになります。ソフト リミットの場合、ツールは制限に従うよう試みますが、無視した方が結果が良くなる場合は無視します。

この属性は、レジスタおよび組み合わせ信号にのみ使用できます。ファンアウトの制限に従うため、レジスタまたは組み合わせ信号を駆動する信号が複製されます。

注記: 入力およびブラック ボックスは、現在のところサポートされません。

MAX_FANOUT の Verilog 例

Vivado の場合

```
(* keep = "true", max_fanout = 50 *) reg sig1;
```

XST の場合

```
(* keep = "true", max_fanout = "50" *) reg sig1;
```

MAX_FANOUT の VHDL 例

```
signal sig1 : std_logic;  
attribute keep : string;  
attribute max_fanout : integer;  
attribute keep of sig1 : signal is "true";  
attribute max_fanout : signal is 50;
```

- max_fanout の値は、VHDL の Vivado 合成では整数です。
- VHDL XST では文字列です。

PARALLEL_CASE (Verilog のみ)

PARALLEL_CASE は、case 文が平行構文で構築される必要のあることを示します。ロジックは if-elsif 構文では作成されません。

```
(* parallel_case *) case select
3'b100 : sig = val1;
3'b010 : sig = val2;
3'b001 : sig = val3;
endcase
```



重要: この制約は、Verilog RTL でのみ制御できます。

RAM_STYLE

RAM_STYLE は、合成でのメモリの推論方法を指定します。有効な値は次のとおりです。

- block : RAMB タイプのコンポーネントが推論されるよう指定します。
- distributed: LUT RAM が推論されるよう指定します。

デフォルトでは、ほとんどのデザインで最適な結果が生成される RAM が推論されます。

RAM_STYLE の Verilog 例

```
(* ram_style = "distributed" *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

RAM_STYLE の VHDL 例

```
attribute ram_style : string;
attribute ram_style of myram : signal is "distributed";
```

ROM_STYLE

ROM_STYLE は、合成での ROM メモリの推論方法を指定します。有効な値は次のとおりです。

- block : RAMB タイプのコンポーネントが推論されるよう指定します。
- distributed : LUT ROM が推論されるよう指定します。デフォルトでは、ほとんどのデザインで最適な結果が生成される ROM が推論されます。

ROM_STYLE の Verilog 例

```
(* rom_style = "distributed" *) reg [data_size-1:0] myrom [2**addr_size-1:0];
```

ROM_STYLE の VHDL 例

```
attribute rom_style : string;
attribute rom_style of myrom : signal is "distributed";
```

TRANSLATE_OFF/TRANSLATE_ON

TRANSLATE_OFF および TRANSLATE_ON は、合成でコードのブロックを無視するよう指定します。これらの属性は、RTL のコメント内で指定します。コメントは、次のいずれかのキーワードで開始します。

- synthesis
- synopsys
- pragma

translate_off で無視が開始され、translate_on で終了します。これらのコマンドはネスト化できません。

TRANSLATE_OFF/TRANSLATE_ON の Verilog 例

```
// synthesis translate_off
Code...
// synthesis translate_on
```

TRANSLATE_OFF/TRANSLATE_ON の VHDL 例

```
-- synthesis translate_off
Code...
-- synthesis translate_on
```



注意 : translate 文の間に含めるコードの種類には、注意が必要です。デザインの動作に影響するコードの場合、シミュレータで使用され、シミュレーションで不一致が発生することがあります。

USE_DSP48

USE_DSP48 は、合成の演算構造をどのように処理するかを指定します。デフォルトでは、乗算器、乗加算器、乗減算器、乗累算器タイプの構造が DSP48 ブロックに含まれます。加算器、減算器、アキュムレータもこれらのブロックに含めることはできますが、デフォルトでは DSP48 ブロックではなくファブリックにインプリメントされます。USE_DSP48 制約を使用すると、このデフォルト動作が変更され、これらの構造が DSP48 ブロックに含まれるようになります。

有効な値は、yes および no です。この属性は、RTL の信号、アーキテクチャおよびコンポーネント、エンティティおよびモジュールに指定できます。優先順位は次のとおりです。

1. 信号
2. アーキテクチャおよびコンポーネント
3. モジュールおよびエンティティ

この属性を指定しない場合は、Vivado 合成で最適な動作が決定されます。

USE_DSP48 の Verilog 例

```
(* use_dsp48 = "yes" *) module test(clk, in1, in2, out1);
```

USE_DSP48 の VHDL 例

```
attribute use_dsp48 : string
attribute use_dsp48 of P_reg : signal is "no"
```

SystemVerilog サポート

概要

Vivado™ 合成では、次のセクションで説明する、合成可能な SystemVerilog RTL の一部がサポートされます。

特定のファイルで SystemVerilog を使用

デフォルトでは、*.v ファイルは Verilog 2001 構文で、*.sv ファイルは SystemVerilog 構文でコンパイルされます。

Vivado IDE で特定の *.v ファイルに SystemVerilog を使用するには、次の手順に従います。

1. ファイルを右クリックし、[Source Node Properties] をクリックします。
2. [Source Node Properties] ビューで、[Type] を [Verilog] から [SystemVerilog] に変更し、[Apply] をクリックします。

または、[Tcl Console] ビューで次の Tcl コマンドを使用します。

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

次のセクションで、Vivado IDE でサポートされる SystemVerilog のデータ型を説明します。

データ型

次のデータ型とその制御方法がサポートされています。

宣言

RTL の変数は次のように宣言します。

```
[var] [DataType] name;
```

説明：

- var はオプションで、宣言文にない場合は自動的に推測されます。
- DataType は次のいずれかになります。
 - integer_vector_type: bit、logic、または reg
 - integer_atom_type: byte、shortint、int、longint、integer、または time
 - non_integer_type: shortreal、real、または realtime
 - struct
 - enum

整数データ型

SystemVerilog では、次の整数型がサポートされます。

- shortint: 2 値の 16 ビット符号付き整数
- int: 2 値の 32 ビット符号付き整数
- longint: 2 値の 64 ビット符号付き整数
- byte: 2 値の 8 ビット符号付き整数
- bit: 2 値のユーザー定義のベクター サイズ
- logic: 4 値のユーザー定義のベクター サイズ
- reg: 4 値のユーザー定義のベクター サイズ
- integer: 4 値の 32 ビット符号付き整数
- time: 4 値の 64 ビット符号なし整数

4 値および 2 値とは、これらのデータ型に割り当てることのできる値を示しています。

- 2 値の場合は 0 および 1 を使用できます。
- 4 値の場合は X と Z も使用できます。

X と Z 値は常に合成できるわけではないので、2 値と 4 値は同じように合成されます。



注意：4 値の変数を使用する場合、RTL とシミュレーションの不一致が発生する可能性があるので、注意してください。

- デフォルトでは、byte、shortint、int、integer、および longint のデータ型は符号付きの値になります。
- bit、reg、および logic は符号なしの値になります。

実数

合成で実数がサポートされますが、ビヘイビアには使用できず、パラメーター値として使用できます。SystemVerilog では、次の実数型がサポートされます。

- `real`
- `shortreal`
- `realtime`

Void データ型

`void` データ型は、戻り値のない関数でのみサポートされます。

ユーザー定義型

Vivado 合成では、`typedef` キーワードを使用してユーザーが定義したデータ型がサポートされます。次の構文を使用します。

```
typedef data_type type_identifier {size};
```

または

```
typedef [enum, struct, union] type_identifier;
```

列挙型

列挙型は、次の構文で宣言できます。

```
enum [type] {enum_name1, enum_name2...enum_namex} identifier
```

データ型を指定しない場合は、デフォルトで `int` になります。次に例を示します。

```
enum {sun, mon, tues, wed, thurs, fri, sat} day_of_week;
```

このコードでは、7つの値を含む `int` の `enum` が生成されます。これらの名前には0から開始する値が割り当てられ、`sun = 0` および `sat = 6` となります。

デフォルト値を変更するには、次の例のようなコードを使用します。

```
enum {sun=1, mon, tues, wed, thurs, fri, sat} day_of_week;
```

この場合、`sun` の値は1で `sat` の値は7になります。

次の例は、デフォルト値を変更する別の方法を示します。

```
enum {sun, mon=3, tues, wed, thurs=10, fri=12, sat} day_of_week;
```

この場合、`sun=0`、`mon=3`、`tues=4`、`wed=5`、`thurs=10`、`fri=12`、および `sat=13` になります。

列挙型は、`typedef` キーワードでも使用できます。

```
typedef enum {sun,mon,tues,wed,thurs,fri,sat} day_of_week;  
day_of_week my_day;
```

この例では、`day_of_week` というデータ型の `my_day` という信号を定義しています。`enum` の範囲を指定することもできます。たとえば、上記の例の場合は次のように指定できます。

```
enum {day[7]} day_of_week;
```

これにより、day0、day1...day6 という N-1 個の要素を含む day_of_week という列挙型が作成されます。

次は、これを別の方法で使用した例です。

```
enum {day[1:7]} day_of_week; // creates day1,day2...day7
enum {day[7] = 5} day_of_week; //creates day0=5, day1=6... day6=11
```

定数

SystemVerilog には、次の 3 種類のエラボレーション時間定数があります。

- parameter: Verilog 規格と同じで、同様に使用できます。
- localparam: parameter と似ていますが、上位モジュールのものよりも優先されます。
- specparam: 遅延とタイミング値を指定するために使用されますが、Vivado 合成ではサポートされません。

const というランタイム定数宣言もあります。

型演算子

型演算子を使用すると、パラメーターをデータ型として指定でき、モジュールの異なるインスタンスに異なるデータ型のパラメーターを設定できます。

キャスト演算子

SystemVerilog では、あるデータ型の値を別のデータ型の値に割り当てることはできませんが、キャスト演算子(')を使用すると可能になります。キャスト演算子を使用すると、データ型を変換できます。次のように使用します。

```
casting_type' (expression)
```

casting_type は次のいずれかになります。

- integer_type
- non_integer_type
- real_type
- 符号なしの定数値
- ユーザーが作成した符号付き値型

複合データ型

複合データ型には、構造体 (struct) と共用体 (union) があります。次にこれらについて説明します。

構造体 (struct)

構造体とは、異なるデータ型の値を 1 つにまとめて格納し、参照できるようにしたものです。各要素はメンバーと呼ばれます。これは、VHDL のレコード型と類似しています。構造体の構文は次のようになります。

```
struct {struct_member1; struct_member2;...struct_memberx;} structure_name;
```

共用体 (union)

共用体は、複数のデータ型を含むデータ型ですが、そのうち 1 つのみが使用されます。これは、データ型が使用方法によって変化する場合などに便利な方法です。次に例を示します。

```
typedef union {int i; logic [7:0] j} my_union;  
my_union sig1;  
my_union sig2;  
sig1.i = 32; //sig1 will get the int format  
sig2.j = 8'b00001111; //sig2 will get the 8bit logic format.
```

パック配列とアンパック配列

Vivado 合成では、パック配列とアンパック配列のどちらもサポートされます。

```
logic [5:0] sig1; //packed array  
logic sig2 [5:0]; //unpacked array
```

幅が決まっているデータ型では、パックされる次元を宣言する必要はありません。

```
integer sig3; //equivalent to logic signed [31:0] sig3
```

プロセス

always プロシージャ

always プロシージャには、次の 4 つがあります。

- always
- always_comb
- always_latch
- always_ff

always_comb プロシージャは、組み合わせロジックを記述します。センシティビティ リストは、always_comb 文を駆動するロジックにより推論されます。

always 文では、ユーザーがセンシティビティ リストを指定する必要があります。次の例では、in1 および in2 センシティビティ リストが使用されます。

```
always@(in1 or in2)  
out1 = in1 & in2;  
always_comb out1 = in1 & in2;
```

always_latch プロシージャでは、ラッチをすばやく作成できます。always_comb と同様、センシティビティ リストは推論されますが、次の例に示すようにラッチ イネーブルの制御信号を指定する必要があります。

```
always_latch  
if(gate_en) q <= d;
```

always_ff プロシージャでは、フリップフロップが作成されます。always と同様、ユーザーがセンシティビティ リストを指定する必要があります。

```
always_ff@(posedge clk)  
out1 <= in1;
```


ブロック文

ブロック文は、複数の文をグループ化します。シーケンシャル ブロックの場合は、文が `begin` と `end` で囲まれます。ブロックでは、そのブロック特有の変数を宣言できます。シーケンシャル ブロックには、そのブロックに関連した名前を付けることもできます。フォーマットは次のとおりです。

```
begin [: block name]
    [declarations]
    [statements]
end [: block name]

begin : my_block
    logic temp;
    temp = in1 & in2;
    out1 = temp;
end : my_block
```

上記の例では、ブロック名が `end` の後にも指定されていますが、これはコードを読みやすくするために、必須ではありません。

注記：パラレルブロック (fork-join ブロック) は Vivado 合成ではサポートされません。

手続きタイミング制御

SystemVerilogでは、次の2種類のタイミング制御がサポートされます。

- 遅延制御: 文とそれが実行されるまでの時間を指定します。これを合成に使用する利点はないので、Vivado 合成では、代入するロジックは作成されますが、タイム文は無視されます。
- イベント制御: `always@(posedge clk)` など、指定したイベントが発生したときに代入が実行されるようにします。これは Verilog の規格ですが、SystemVerilog では機能が追加されています。

論理 or 演算子を使用すると、任意の数のイベントを指定でき、いずれかのイベントで文の実行をトリガーできます。これには、センシティビティ リストでイベントを or またはカンマで区切ります。たとえば、次の2つの文は同じです。

```
always@(a or b or c)
always@(a,b,c)
```

SystemVerilog では、`event_expression *` もサポートされるので、センシティビティ リストの問題によるシミュレーションでの不一致を回避できます。次に例を示します。

```
Logic always* begin
```

演算子

Vivado 合成では、次の SystemVerilog の演算子がサポートされます。

- 代入演算子 (`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`、`<<<=`、`>>>=`)
 - 単項演算子 (`+`、`-`、`!`、`~`、`&`、`~&`、`|`、`~|`、`^`、`~^`、`^~`)
 - インクリメント/デクリメント演算子 (`++`、`--`)
 - 2項演算子 (`+`、`-`、`*`、`/`、`%`、`==`、`~=`、`===`、`~===`、`&&`、`||`、`**`、`<`、`<=`、`>`、`>=`、`&`、`|`、`^`、`^~`、`~^`、`>>`、`<<`、`>>>`、`<<<`)
- 注記：**`A**B` は、A が 2 のべき乗であるか、B が定数の場合にサポートされます。
- 条件演算子 (`? :`)
 - 連結演算子 (`{...}`)

符号付き演算式

Vivado 合成では、符号付き演算と符号なし演算のどちらもサポートされます。信号は、符号ありとなしのどちらにでも宣言できます。次に例を示します。

```
logic [5:0] reg1;  
logic signed [5:0] reg2;
```

手続きプログラム代入文

if-else 条件文

if-else 条件文の構文は、次のようになります。

```
if (expression)  
    command1;  
else  
    command2;
```

else 文はオプションで、クロック文の有無によってラッチまたはフリップフロップが想定されます。次のような複数の if および else 文もサポートされます。

```
If (expression1)  
    Command1;  
else if (expression2)  
    command2;  
else if (expression3)  
    command3;  
else  
    command4;
```

このコードは、priority if 文として合成され、最初の条件文が true である場合、その他の条件文は評価されません。Vivado 合成では、unique if-else 文は parallel_case として、priority if-else 文は full_case として処理されます。

case 文

case 文の構文は、次のようになります。

```
case (expression)  
    value1: statement1;  
    value2: statement2;  
    value3: statement3;  
    default: statement4;  
endcase
```

case 文内の default 文はオプションです。値は順番に評価されるので、value1 と value3 の両方が true の場合、statement1 が実行されます。

case 文のほかに casex 文および casez 文があります。casex ではドントケアを、casez ではトライステート条件を処理できます。

Vivado 合成では、unique case 文は parallel_case として、priority case 文は full_case として処理されます。

ループ文

Vivado 合成および SystemVerilog では、数種のループ文がサポートされます。最もよく使用されるのは for ループ文です。構文は次のとおりです。

```
for (initialization; expression; step)
    statement;
```

for 文では、まず初期化が実行されてから、条件式が評価されます。結果が 0 の場合は停止し、1 の場合は文の実行が続行されます。文の実行が終了したら、ステップ関数が実行されます。

- repeat ループ文では、指定した回数分だけ関数が繰り返し実行されます。構文は次のとおりです。

```
repeat (expression)
    statement;
```

条件式をある値に対して評価し、文をその回数だけ実行します。

- for-each ループ文では、配列内の各エレメントに対して文が実行されます。
- while ループ文では、演算式が false になるまでその文が実行されます。
- do-while ループ文は while ループ文と同じですが、文の後に条件式が評価される点異なります。
- forever ループはずっと実行され続けます。無限ループにならないようにするため、ループを脱出するための break 文と共に使用してください。

タスクおよび関数

タスク

タスク宣言の構文は、次のとおりです。

```
task name (ports);
    [optional declarations];
    statements;
endtask
```

タスクには、次の 2 種類があります。

- **スタティック タスク**: 宣言文に、次にタスクが呼び出されたときに前の値が保持されます。
- **自動タスク**: 宣言文には前の値は保持されません。



注意: Vivado 合成ではすべてのタスクが自動タスクとして処理されるので、タスクを使用する際は注意が必要です。

多くのシミュレータでは、スタティックか自動かを指定しない場合のデフォルトがスタティック タスクなので、シミュレーションで不一致が発生する可能性があります。タスクを自動またはスタティックに指定する方法は、次のとおりです。

```
task automatic my_mult...//or
task static my_mult ...
```

関数 (自動またはスタティック)

関数はタスクと類似していますが、値が返される点が異なります。関数の構文は、次のとおりです。

```
function data_type function_name(inputs);  
    declarations;  
    statements;  
endfunction : function_name
```

最後の `function_name` はオプションですが、この方がコードが読みやすくなります。関数は値を返すので、`return` 文を使用するか、文で特定しておく必要があります。

```
function_name = ....
```

タスクと同様、関数も自動またはスタティックにできます。Vivado 合成ではすべての関数が自動として処理されますが、シミュレータによっては動作が異なります。関数を使用する場合は注意が必要です。

モジュールおよび階層

SystemVerilog のモジュールは Verilog と似ていますが、次のセクションで説明するようにさらに機能が追加されています。

モジュールの接続

モジュールのインスタンス化および接続には、主に 3 つの方法がありますが、最初の 2 つ (順序付きリストおよび名前による方法) は Verilog と同じです。3 つ目はポート名による方法です。

モジュールのポート名がインスタンス化 モジュールの信号と名前およびタイプが同じ場合、下位モジュールを名前前で接続できます。次に例を示します。

```
module lower (  
    output [4:0] myout;  
    input clk;  
    input my_in;  
    input [1:0] my_in2;  
    .....  
endmodule  
//in the instantiating level.  
lower my_inst (.myout, .clk, .my_in, .my_in2);
```

ワイルドカード ポートを使用したモジュールの接続

ワイルドカードを使用してモジュールを接続することもできます。たとえば、上記の例は次のようになります。

```
// in the instantiating module  
lower my_inst (.*);
```

上位モジュールの名前とタイプが正しければ、インスタンス全体が接続されます。

また、これらを混合して使用できます。次に例を示します。

```
lower my_inst (.myout(my_sig), .my_in(din), .*);
```

この例では、`myout` ポートが `my_sig` 信号に、`my_in` ポートが `din` 信号に、`clk` および `my_in2` が `clk` および `my_in2` 信号に接続されます。

インターフェイス

インターフェイスは、ブロック間の通信を指定するために使用します。インターフェイスとは、モジュール間の接続を記述しやすくする目的でまとめられたネットおよび変数のグループです。

基本的なインターフェイスの構文は、次のとおりです。

```
interface interface_name;  
    parameters and ports;  
    items;  
endinterface : interface_name
```

最後の interface_name はオプションで、コードを読みやすくする目的で追加されています。次に例を示します。

```
module bottom1 (  
    input clk,  
    input [9:0] d1,d2,  
    input s1,  
    input [9:0] result,  
    output logic sel,  
    output logic [9:0] data1, data2,  
    output logic equal);  
  
    //logic//  
  
endmodule  
  
module bottom2 (  
    input clk,  
    input sel,  
    input [9:0] data1, data2,  
    output logic [9:0] result);  
  
    //logic//  
  
endmodule  
  
module top (  
    input clk,  
    input s1,  
    input [9:0] d1, d2,  
    output equal);  
  
    logic [9:0] data1, data2, result;  
    logic sel;  
  
    bottom1 u0 (clk, d1, d2, s1, result, sel, data1, data2, equal);  
    bottom2 u1 (clk, sel, data1, data2, result);  
endmodule
```

上記のコードでは、共通する信号を持つ2つの下位モジュールがインスタンス化されています。これらの共通信号はすべてインターフェイスを使用して指定できます。

```
interface my_int  
    logic sel;  
    logic [9:0] data1, data2, result;  
endinterface : my_int
```

これで 2 つの bottom モジュールを次のように変更できます。

```
module bottom1 (  
    my_int int1,  
    input clk,  
    input [9:0] d1, d2,  
    input s1,  
    output logic equal);
```

および

```
module bottom2 (  
    my_int int1,  
    input clk);
```

これらのモジュール内では、`sel`、`data1`、`data2`、`result` へのアクセス方法を変更することもできます。これは、モジュールの視点からは、これらの名前のポートが存在しないからです。その代わりに、`my_int` という名前のポートがあります。これには、次の変更を加える必要があります。

```
if (sel)  
    result <= data1;
```

上記を次のように変更します。

```
if (int1.sel)  
    int1.result <= int1.data1;
```

最後に、最上位にそのインターフェイスをインスタンス化する必要があります。これで、インスタンスでそのインターフェイスが参照されます。

```
module top(  
    input clk,  
    input s1,  
    input [9:0] d1, d2,  
    output equal);  
my_int int3(); //instantiation  
  
bottom1 u0 (int2, clk, d1, d2, s1, equal);  
bottom2 u1 (int2, clk);  
endmodule
```

modport キーワード

上記の例では、インターフェイス内の信号が入力または出力として記述されなくなっています。インターフェイスが追加される前は、sel ポートが bottom1 の出力および bottom2 の入力でした。

インターフェイスが追加されると、それが明確ではなくなります。実際、Vivado 合成エンジンではこれらが双方向ポートと考慮されるようになったことに対して警告メッセージは表示されず、階層を使用して生成されたネットリストではこれらが入出力として定義されます。これは、生成されたロジックの観点からは問題ではありませんが、混乱の原因となります。

方向を指定するには、次のコード例に示すように modport キーワードを使用します。

```
interface my_int;
    logic sel;
    logic [9:0] data1, data2, result;

    modport b1 (input result, output sel, data1, data2);
    modport b2 (input sel, data1, data2, output result);
endinterface : my_int
```

この後、bottom モジュールの宣言に次を使用します。

```
module bottom1 (
    my_int.b1 int1,
```

これで、入力と出力が正しく関連付けられます。

その他のインターフェイス機能

インターフェイスには、信号だけでなく、タスクおよび関数も含めることができ、そのインターフェイスに特有のタスクを作成できます。

インターフェイスのパラメーターも指定できます。前述の例の場合、data1 および data2 は両方とも 10 ビットのベクターですが、パラメーターを設定することにより任意のサイズに変更できます。

package 文

package 文を使用すると、さまざまなコンストラクトを共有できます。これは、VHDL のパッケージ文と同じように動作します。package 文には、関数、タスク、データ型、列挙型などを含めることができます。package 文の構文は、次のとおりです。

```
package package_name;
    items
endpackage : package_name
```

最後の package_name は、コードを読みやすくするために、必須ではありません。

パッケージは、ほかのモジュールで import コマンドを使用して参照されます。構文は次のとおりです。

```
import package_name::item or *;
```

import コマンドでは、インポートするパッケージからのアイテムを指定するか、パッケージ全体を指定する必要があります。

その他のリソース

ザイリンクス リソース

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

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

ザイリンクス資料で使用する用語集は、次を参照してください。

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

ソリューション センター

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

Vivado 資料

Vivado™ Design Suite 2012.2 資料ページ

http://japan.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm

1. 『ザイリンクス デザイン ツール：リリース ノート ガイド』(UG631)
2. 『ザイリンクス デザイン ツール：インストールおよびライセンス ガイド』(UG798)
3. 『Vivado Design Suite ユーザー ガイド：Vivado IDE の使用』(UG893)
4. 『Vivado Design Suite Tcl コマンド リファレンス ガイド』(UG835)
5. 『Vivado Design Suite ユーザー ガイド：Tcl スクリプト機能の使用』(UG894)
6. 『Vivado Design Suite ユーザー ガイド：インプリメンテーション』(UG904)
7. 『Vivado Design Suite 移行手法ガイド』(UG912)
8. 『Vivado Design Suite ユーザー ガイド：デザイン フローの概要』(UG892)
9. 『PlanAhead ユーザー ガイド』(UG632)
10. 『XST ユーザー ガイド (Virtex-6、Spartan-6、7 シリーズ デバイス)』(UG687)