

System Generator for DSP

ユーザー ガイド

UG640 (v11.4) 2009 年 12 月 2 日



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.

© 2009 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, 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 (v11.4) を翻訳したもので、内容に相違が生じる場合には原文を優先します。
資料によっては英語版の更新に対応していないものがあります。
日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

目次

このマニュアルについて

マニュアルの内容	7
System Generator の PDF マニュアル セット	7
その他のリソース	8
表記規則.....	8
書体.....	8
オンライン マニュアル.....	9

第 1 章：System Generator を使用したハードウェア設計

FPGA の概要.....	12
DSP 設計者へのメモ	16
ハードウェア設計者へのメモ.....	16
System Generator を使用したデザイン フロー.....	17
アルゴリズムの解析	17
大型デザインの一部としてインプリメント	17
完全なデザインのインプリメント	17
System Generator でのシステム レベルのモデリング	18
System Generator ブロックセット.....	19
信号型	21
ビット単位およびサイクル単位のモデリング.....	22
タイミングとクロック	22
同期化のメカニズム	34
ブロック マスクとパラメータの伝搬.....	34
リソースの予測	36
自動コード生成	36
System Generator トークンを使用したコンパイルとシミュレーション	37
ISE レポートの表示	42
コンパイル結果	42
HDL テストベンチ	48
MATLAB の FPGA へのコンパイル.....	49
単純なセレクト	50
単純な数値演算	51
レイテンシのある複素乗算器	53
シフト操作	54
MCode ブロックへパラメータを渡す	55
オプションの入力ポート	58
有限状態マシン	60
パラメータ指定アキュムレータ	61
FIR とシステム検証	64
RPN カリキュレータ	67
disp 関数	69
System Generator デザインの大型システムへのインポート	71
HDL ネットリストのコンパイル.....	71
デザインの統合に関する規則	71
System Generator と Project Navigator の統合フロー	72
手順の例	73
コンフィギャブル サブシステムと System Generator	80
コンフィギャブル サブシステムの定義	80
コンフィギャブル サブシステムの使用	82
コンフィギャブル サブシステムからのブロックの削除	83
コンフィギャブル サブシステムへのブロックの追加	83

コンフィギュラブル サブシステムからのハードウェアの生成	84
高パフォーマンス FPGA デザインに関するメモ	86
ブロックのパラメータ ダイアログ ボックスに含まれている「Hardware notes」を読む	86
デザインの入力と出力にレジスタを付ける	86
パイプライン レジスタを挿入する	86
[Saturate] および [Round] オプションは必要な場合以外は使用しない	86
System Generator のタイミング解析ツールおよび消費電力解析ツールを使用する	86
すべての Gateway ブロックでデータ レート オプションを設定する	87
クロック イネーブル (CE) のファンアウトを低減する	87
FPGA 物理デザイン ツールを使用した System Generator デザインの処理	87
HDL シミュレーション	87
FPGA ビットストリームの生成	90
自動生成されたクロック イネーブル ロジックのリセット	93
ce_clr とレート変更ブロック	94
ce_clr の使用に関する推奨事項	95
DSP48 の設計手法	96
DSP48 について	96
標準コンポーネントを使用したデザイン	97
合成可能な Mult、Mux、AddSub ブロックを使用したデザイン	97
DSP48 および DSP48 Macro ブロックを使用したデザイン	98
DSP48 設計手法	103
FDATool を使用したデジタル フィルタ アプリケーション	106
デザインの概要	107
FIR フィルタの係数の生成	107
MAC Based FIR ブロックのパラメータ指定	108
FIR フィルタの係数の生成と割り当て	109
ザイリンクス フィルタ ブロックの理解	110
シミュレーションの実行	111
複数クロックのサイクル単位アイランドの生成	113
複数クロック アプリケーション	114
クロック ドメインの分割	115
クロック ドメインの切り替え	115
複数クロック デザインのネットリスト生成	117
手順の例	118
最上位ラッパの作成	122
ChipScope Pro Analyzer を使用したリアルタイム ハードウェア デバッグ	125
ChipScope Pro の概要	125
チュートリアル : System Generator 内での ChipScope の使用	126
リアルタイム デバッグ	131
ChipScope から MATLAB ワークスペースへのデータのインポート	134

第 2 章：ハードウェア/ソフトウェア協調設計

System Generator でのハードウェア / ソフトウェア協調設計	135
Black Box ブロック	135
PicoBlaze Microcontroller ブロック	135
EDK Processor ブロック	136
カスタム ロジックへのプロセッサの統合	136
メモリ マップの作成	137
ハードウェアの生成	138
ハードウェア協調シミュレーション	139
ソフトウェア ドライバの生成	139
EDK プロセッサのソフトウェア記述	140
EDK プロセッサでの非同期のサポート	141
EDK サポート	142
EDK プロセッサのインポート	142
System Generator へのプロセッサ ポートの追加	144
pcore のエクスポート	145

エンベデッド プロセッサおよびマイクロプロセッサを使用した設計	145
PicoBlaze マイクロコントローラ アプリケーションの設計	145
MicroBlaze プロセッサ ペリフェラルの設計とエクスポート	152
チュートリアル: MicroBlaze プロセッサ システムの設計とシミュレーション	156
XPS の使用	164
Platform Studio SDK の使用	170

第 3 章: ハードウェア協調シミュレーションの使用

概要	181
M コードでのハードウェア協調シミュレーションへのアクセス	181
ハードウェア プラットフォームのインストール	181
イーサネット ベース ハードウェア協調シミュレーション	181
JTAG ベース ハードウェア協調シミュレーション	182
サードパーティ ハードウェア協調シミュレーション	182
ハードウェア協調シミュレーション用のモデルのコンパイル	182
コンパイル ターゲットの選択	182
コードの生成	183
ハードウェア協調シミュレーション ブロック	184
ハードウェア協調シミュレーションのクロック	186
クロック周波数の選択	186
クロック モード	187
クロック モードの選択	187
ボード専用 I/O ポート	188
ハードウェア協調シミュレーションでの I/O ポート	189
イーサネット ハードウェア協調シミュレーション	189
ポイントツーポイント イーサネット ハードウェア協調シミュレーション	190
ネットワーク ベースのイーサネット ハードウェア協調シミュレーション	193
共有メモリのサポート	195
ハードウェア協調シミュレーション用の共有メモリのコンパイル	196
非保護の共有メモリの協調シミュレーション	199
ロック共有メモリの協調シミュレーション	200
共有レジスタの協調シミュレーション	201
共有 FIFO の協調シミュレーション	203
共有メモリの制限	205
ザイリンクス ツール フローの設定	205
ハードウェア協調シミュレーションを使用したフレーム ベースの シミュレーション	207
共有メモリ	208
デザインへのバッファの追加	209
ハードウェア協調シミュレーション用のコンパイル	212
ユーザー ベクタ転送	214
ハードウェア協調シミュレーションを使用したリアルタイム信号処理	219
共有メモリ I/O バッファの例	219
5x5 フィルタ データ パスの挿入	221
5x5 フィルタ カーネル テストベンチ	224
カーネルの再読み込み	228
ハードウェア協調シミュレーション ボードのインストール	229
イーサネット ハードウェア協調シミュレーション用の ML402 プラットフォームの インストール	229
イーサネット ハードウェア協調シミュレーション用の ML506 プラットフォームの インストール	237
イーサネット ハードウェア協調シミュレーション用の ML605 プラットフォームの インストール	246
イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 1800A スタータ プラットフォームのインストール	249

イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 3400A 開発プラットフォームのインストール	253
JTAG ハードウェア協調シミュレーション用の ML402 プラットフォームの インストール	262
JTAG ハードウェア協調シミュレーション用の ML605 プラットフォームの インストール	263
JTAG ハードウェア協調シミュレーション用の SP605 プラットフォームの インストール	266
JTAG ハードウェア協調シミュレーション用の新規プラットフォームのサポート	267
ハードウェア要件	267
新規プラットフォームのサポート	268

第 4 章：HDL モジュールのインポート

ブラック ボックスの HDL の要件と制限	282
ブラック ボックス コンフィギュレーション ウィザード	283
ブラック ボックスのコンフィギュレーション M 関数	284
HDL 協調シミュレーション	296
概要	296
HDL シミュレータの設定	296
複数のブラック ボックスの協調シミュレーション	298
ブラック ボックスの例	298
CORE Generator モジュールのインポート	299
VHDL モジュールのインポート	313
Verilog モジュールのインポート	320
ダイナミック ブラック ボックス	322
複数のブラック ボックスの同時シミュレーション	324
ModelSim を使用したアドバンス ブラック ボックスの例	326
暗号化された VHDL ファイルのインポート、シミュレーション、エクスポート	332

第 5 章：System Generator のコンパイル タイプ

HDL ネットリストへのコンパイル	338
NGC ネットリストへのコンパイル	338
ビットストリームへのコンパイル	339
XFLOW のオプション ファイル	340
追加の設定	341
EDK Processor ブロックに含まれるソフトウェア プログラムの ビットストリームへの再コンパイル	342
EDK Export Tool	343
pcore エクスポート用のカスタム バス インターフェイスの作成	344
pcore として EDK にエクスポート	345
System Generator ポートを EDK の最上位ポートとして使用	346
サポートされるプロセッサと制限	346
関連項目	346
ハードウェア協調シミュレーション用のコンパイル	346
タイミングおよび消費電力解析用のコンパイル	347
タイミング解析の概要	349
タイミング解析ツールの機能	350
コンパイル ターゲットの作成	362
新規コンパイル ターゲットの定義	362

索引	367
----------	-----

このマニュアルについて

このマニュアルには、System Generator を理解し、使用するために重要なトピックが含まれています。また、『System Generator for DSP 入門ガイド』には掲載しきれなかった例やチュートリアルも含まれています。

マニュアルの内容

このマニュアルは、次の章から構成されています。

- 第 1 章「System Generator を使用した ハードウェア設計」
- 第 2 章「ハードウェア/ソフトウェア協調設計」
- 第 3 章「ハードウェア協調シミュレーションの使用」
- 第 4 章「HDL モジュールのインポート」
- 第 5 章「System Generator のコンパイル タイプ」

System Generator の PDF マニュアル セット

このマニュアルは System Generator のヘルプ システム (英語版) として参照でき、また System Generator の PDF マニュアル セット の一部です。この PDF マニュアル セット には、次のマニュアルが含まれています。

- System Generator for DSP 入門ガイド
- System Generator for DSP ユーザー ガイド
- System Generator for DSP リファレンス ガイド

メモ：これらのマニュアル間のハイパーリンクは、PDF ファイルが同じフォルダにある場合にのみ機能します。Adobe Reader でハイパーリンクをクリックした場合、Alt キーと左方向キー (←) を同時に押すと、前に表示していたページに戻ることができます。

その他のリソース

追加の資料は、次の Web サイトから参照できます。

<http://japan.xilinx.com/support/documentation/index.htm>

シリコンやソフトウェア、IP に関するアンサー データベースを検索したり、テクニカル サポートのウェブ ケースを開く場合は、次の 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。	<code>ngdbuild design_name</code>
イタリック フォント	ユーザーが値を入力する必要がある構文内の変数に使用します。	<i><code>ngdbuild design_name</code></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、 <code>bus[7:0]</code> 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。	<code>allow block block_name loc1 loc2 ... locn;</code>

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を示します。	詳細は、「 その他のリソース 」を参照してください。 詳細は、第 1 章「 タイトル フォーマット 」を参照してください。
赤色の文字	ほかのマニュアルへの相互参照を示します。	詳細は、『Virtex-5 FPGA ユーザーガイド』の 図 2-5 を参照してください。
青色の下線付き文字	Web サイト (URL) へのハイパーリンクです。	最新のスピード ファイルは、 http://japan.xilinx.com から入手できます。

System Generator を使用した ハードウェア設計

System Generator は、FPGA ハードウェアを設計するためのシステム レベルのモデリング ツールです。Simulink がさまざまな面で拡張されており、ハードウェア設計に適したモデリング環境を提供します。デザインは高い抽象度で記述され、ボタンをクリックするだけで FPGA にコンパイルされます。低い抽象度により FPGA リソースにもアクセスできるので、効率的な FPGA デザインを構築できます。

FPGA の概要

FPGA の基礎概念を示し、System Generator を使用する場合のコンパイル、プログラム、アーキテクチャに関する考慮事項を説明します。

System Generator を使用したデザインフロー

System Generator でのデザイン設計が有益な状況を示します。

System Generator でのシステムレベルのモデリング

柔軟で高度なシステム モデリング環境から、デバイス特定のハードウェア デザインを直接インプリメントする System Generator の機能を説明します。

自動コード生成

System Generator デザインの自動コード生成について説明します。

MATLAB の FPGA へのコンパイル

MATLAB プログラム言語のサブセットを使用してステート マシンおよび数値演算ファンクションを記述する方法を説明します。このように記述したファンクションを System Generator のブロックに含め、等価の HDL に自動的にコンパイルできます。

System Generator デザインの大型システムへのインポート

System Generator から VHDL ネットリストを取り出して合成し、大型デザインに組み込む方法を説明します。また、System Generator で作成した VHDL をシステム全体のシミュレーション モデルに組み込む方法も説明します。

コンフィギャブル サブシステムと System Generator

System Generator でのコンフィギャブル サブシステムの使用方法を示します。コンフィギャブル サブシステムの定義、ブロックの削除と追加、コンフィギャブル サブシステムを使用したコンパイル結果の System Generator デザインへのインポート 方法について説明します。

高パフォーマンス FPGA デザインに関するメモ

FPGA に効率的で高パフォーマンスのデザインをインプリメントするための、System Generator での設計手法を示します。

FPGA 物理デザイン ツールを使用した System Generator デザインの処理	System Generator で生成した低レベルの HDL を Project Navigator、ModelSim、および Synplify などのツールで使用方法を説明します。
自動生成されたクロック イネーブル ロジックのリセット	System Generator ライブラリ のレート 変更ブロックの、再同期化用に <code>ce_clr</code> を使用した場合の動作を説明します。
DSP48 の設計手法	System Generator で DSP48 (XtremeDSP スライス) をインプリメントし、コンフィギュレーションする方法を説明します。
FDATool を使用したデジタル フィルタ アプリケーション	FDATool ブロックを使用して、FIR フィルタを指定、インプリメント、およびシミュレーションする方法を説明します。
複数クロックのサイクル単位アイランドの生成	System Generator でマルチクロック デザインをインプリメントする方法を説明します。
ChipScope Pro Analyzer を使用したリアルタイム ハードウェア デバッグ	ザイリンクスのデバッグ ツールである ChipScope™ Pro を System Generator 内で接続し、使用方法を説明します。

FPGA の概要

FPGA (Field Programmable Gate Array) は、デバイス製造業者ではなく、設計者がプログラムする汎用集積回路です。ASIC (Application-Specific Integrated Circuit) とは異なり、FPGA はシステムに組み込まれた後でも再プログラム可能です。

FPGA は、ビットストリームと呼ばれるコンフィギュレーション プログラムをスタティック オンチップ RAM にダウンロードすることによりプログラムします。このビットストリームは、マイクロプロセッサのオブジェクト コードと同様に、コンパイル ツールで設計者が作成した抽象度の高い記述を低レベルの実行可能なコードに変換することにより生成されます。ザイリンクス System Generator は、高レベルの Simulink モデルから FPGA プログラムをコンパイルできるようにしたツールです。

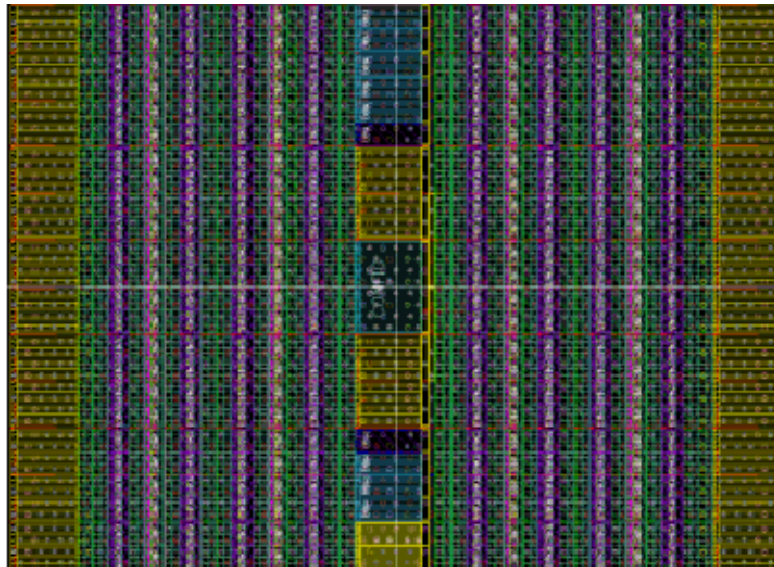
FPGA は、さまざまな演算ファンクションおよびロジック ファンクションをインプリメント可能なコンフィギュラブル リソースの 2 次元のアレイで構成されています。これらのリソースには、DSP ブロック、乗算器、デュアルポート メモリ、ルックアップ テーブル (LUT)、レジスタ、トライステート バッファ、マルチプレクサ、デジタルクロック マネージャ (DCM) などがあります。ザイリンクス FPGA にはさらに、広範囲のバンド幅および電圧要件に対応できる高度な I/O 機構が含まれています。Virtex®-4 FPGA には、エンベデッド マイクロコントローラ (IBM PowerPC® 405) およびマルチギガビット シリアルトランシーバが組み込まれています。演算および I/O リソースはプログラマブルインターコネクトで接続されており、ビットストリームに従ってシステム内で配線されます。

FPGA は、高パフォーマンスのデータ処理デバイスです。FPGA ではデータ処理にパラレル構造を使用できるので、高パフォーマンスの DSP を達成できます。パフォーマンスがプロセッサで実行可能なクロック レートに依存するマイクロプロセッサまたは DSP プロセッサとは異なり、FPGA のパフォーマンスは、信号処理システムを構成するアルゴリズムに導入可能なパラレル構造の量に依存します。高システム クロック レート (現在では 100 ~ 200MHz のシステム周波数が一般的) および高度に分散されたメモリ構造により、データ ストリームで動作する DSP やその他のアプリケーションでパラレル構造を利用できます。たとえば、150MHz のクロック レートで動作する大型 FPGA のメモリ バンド幅は、毎秒数百テラ バイトにも達します。

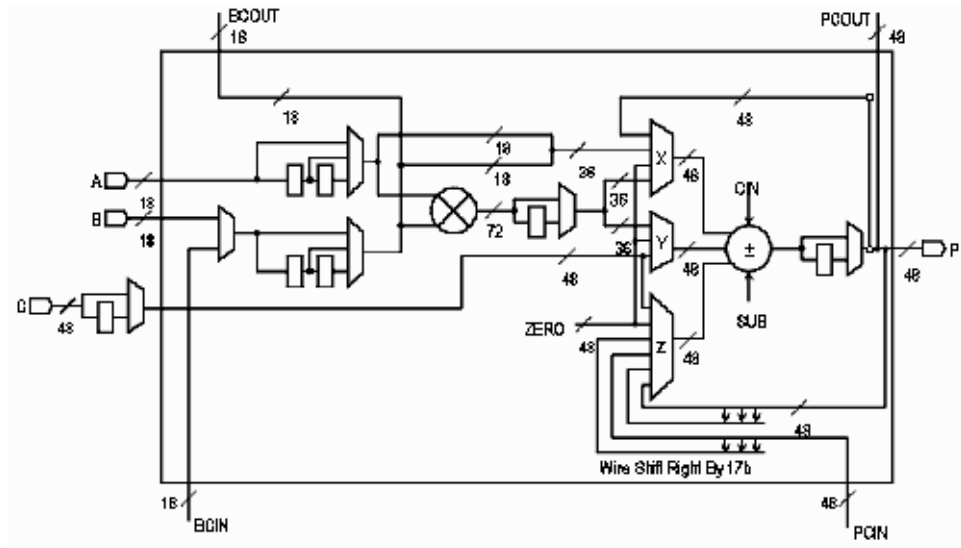
デジタル アップ/ダウン コンバータなど、カスタム集積回路 (IC) のみまたは 1 つの FPGA にインプリメントできる DSP アプリケーションもありますが、フォン ノイマン プロセッサでは、演算機能およびメモリのバンド幅の両方において不十分です。FPGA を使用すると、カスタム IC と比べて NRE (Non-Recurring Engineering) コストを大幅に削減し (FPGA は市販の既製デバイス)、タイムトゥーマーケットを短縮でき、またコンフィギャブルであることからエンド アプリケーションに設置した後にでもデザインに変更を加えることができます。

System Generator を使用する際は、FPGA には信号処理ファンクションをインプリメントするのにさまざまなレベルでの柔軟性があることを頭に入れておくことが重要です。たとえば、システム内で異なるデータ パス幅を定義し、システム要件に応じて MAC エンジンなどの個別のデータ プロセッサを使用できます。System Generator では、インプリメントするアルゴリズムを考慮するだけで FPGA 用のデザインを設計できますが、FPGA に関する理解を深めることにより、高パフォーマンスを達成するための FPGA 特有の機能を利用できるようになります。

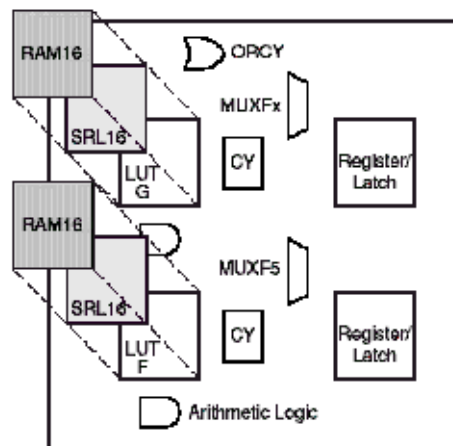
このセクションの残りの部分では、FPGA で利用可能な一部のロジック リソースについて簡単に説明します。



上図は、Virtex-4 FPGA の物理的な構造を表しています。FPGA は、コンフィギャブル インターコネクトの網に埋め込まれたロジック スライスの二次元配列とハード マクロ ブロック (ブロック メモリおよび演算ブロック) の列で構成されていると考えることができ、DSP ファンクションのインプリメンテーションに適しています。Virtex-4 FPGA の DSP ブロック (次の図を参照) は 450MHz 以上で動作可能であり、さまざまなワード数 (BRAM ごとの合計 18Kb) にコンフィギュレーション可能なデュアル ポート メモリ ブロック (BRAM) と周期が一致しています。Virtex-4 SX55 デバイスには、512 個の DSP ブロックと BRAM が含まれています。System Generator では、演算およびロジックの抽象記述を使用してこれらのリソースすべてにアクセス可能で、高パフォーマンスのデジタルフィルタ、FFT、およびその他の演算ファンクションや信号処理ファンクションを構築できます。

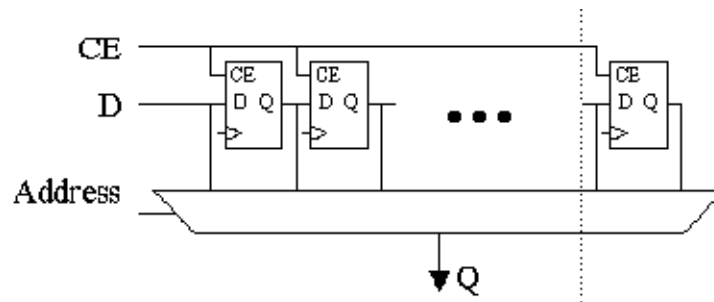


DSP 設計者は、Virtex-4 DSP ブロックでサポートされている積和演算 (MAC) ファンクションに精通しているかも知れませんが、Virtex FPGA ファミリの基本ユニットであるロジック スライス (下図を参照) を理解しておくことは有益です。



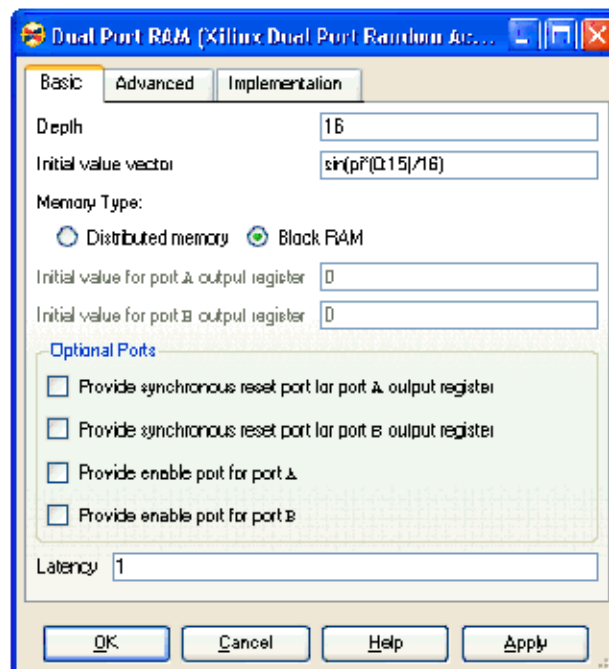
各ロジック スライスには、2つの4入力ルックアップ テーブル (LUT)、2つのコンフィギュラブル D フリップフロップ、マルチプレクサ、専用キャリー ロジック、およびスライス ベースの乗算器を作成するためのゲートが含まれています。1つの LUT には、任意の4入力ブールファンクションをインプリメントできます。高速キャリー回路をインプリメントする専用ロジックと LUT を組み合わせることにより、任意のワード数の高速加減算器や乗算器を構築できます。ブールファンクションのインプリメンテーションに加え、各 LUT は 16X1 ビット RAM またはシフトレジスタ (SRL16) としてもコンフィギュレーションできます。SRL16 シフトレジスタは 16X1 ビットの同期遅延ラインで、動的にタップポイントを指定可能です。

System Generator では、これらの異なるメモリ オプションが高度な抽象記述で表現されています。D フリップフロップ プリミティブの代わりに、任意のサイズのレジスタが提供されています。任意の幅、ワード数の遅延ラインをインプリメントするブロックが 2 つあり、SRL16 に直接マップされます。遅延ブロックは、パイプライン構造のバランスを取るため、または時分割多重 (TDM) データストリームに使用できます。次の図に示す Addressable Shift Register (ASR) ブロックは、任意の幅、ワード数のタップ遅延ラインをインプリメントします。このブロックは、タップ遅延ラインをインプリメントするだけでなく、TDM データ ストリームをスライプするために使用できるので、DSP 設計で特に有益です。



RAM は BRAM または LUT (RAM16X1) プリミティブを使用して構築できますが、大型システムにプリミティブを正しく組み込むには、効率的にマップされるようにするなど、通常詳細な注意が必要です。System Generator ではそのような配慮は不要です。

たとえば、次の図に示す Dual Port RAM (DPRAM) ブロックでは、必要なメモリをインプリメントするため、デバイスに任意の数の RAM または RAM16X1 コンポーネントを効率的にマップできます。このブロックのパラメータ ダイアログ ボックスで、メモリのタイプ (BRAM または分散 RAM)、ワード数 (データ幅は入力ポートを駆動する Simulink 信号から推論)、メモリの初期内容などを設定できます。



通常 System Generator で抽象記述がデバイスのプリミティブに適切にマップされるので、プリミティブ間の接続を考慮する必要はありません。また、ブロックライブラリのファンクションは必要に応じて IP ライブラリを利用して効率的にインプリメントされるので、FPGA の詳細な知識は必要ありません。ただし、FPGA の知識を活用して、加算器、レジスタ、メモリ などの基本ファンクションを使用したアルゴリズムを、すべての信号を明示的に制御せずにインプリメントすることも可能です。

この後のセクションで、System Generator ブロックと Simulink からハードウェアへのマップについて詳細に説明します。FPGA の詳細情報は、<http://japan.xilinx.com/support> からデータシート、アプリケーション ノート、ホワイト ペーパー、その他の技術資料を参照してください。

DSP 設計者へのメモ

System Generator は、Simulink を拡張してハードウェア設計を可能にしたもので、FPGA に自動的にコンパイル可能な高度な抽象記述を提供します。演算の抽象記述は Simulink に適していますが (離散時間/空間ダイナミック システム シミュレーション)、System Generator では FPGA の機能にもアクセスできます。

並列処理やパイプライン処理などをハードウェアでどのように実現しているかを理解することにより、より適切なインプリメンテーションが得られます。IP コアを使用すると、FFT などの複雑なファンクションを含む FPGA デザインを効率的に作成できます。また、より厳密にアプリケーションにフィットするようモデルを調整することも可能です。

System Generator のマニュアル全体をとおして、システム パラメータを使用してハードウェアの機能を設定する方法を示します。

ハードウェア設計者へのメモ

System Generator は、ハードウェア記述言語 (HDL) ベースのデザインに置き換わるものではありませんが、重要な部分にだけ集中することを可能にします。ほとんどの DSP プログラムは、アセンブラでのみプログラムするのではなく、C 言語のような高級言語でプログラムを開始し、パフォーマンス要件を満たすために必要な場合だけアセンブリ コードを記述しています。

経験的に、デザインの内部ハードウェア クロックを制御する必要のある部分 (DDR、位相クロックを使用するなど) は、HDL を使用してインプリメントするのが適切です。比較的重要度の低い部分は System Generator でインプリメントし、その後 HDL 部分と System Generator 部分を接続します。通常、信号処理システムのほとんどの部分では、外部インターフェイスを除き、このレベルの制御は不要です。System Generator には、HDL 設計者に特に関係する HDL コードの部分をデザインにインポートする機能 (「[HDL モジュールのインポート](#)」を参照) が含まれています。

HDL を使用する設計者に关系する System Generator の機能として、テスト ベクタを含む HDL テスト ベンチの自動生成があります。この機能については、「[HDL テストベンチ](#)」を参照してください。

「[ハードウェア協調シミュレーションの使用](#)」で説明されているハードウェア協調シミュレーション インターフェイスを使用すると、Simulink の制御下でデザインをハードウェアで実行でき、MATLAB と Simulink のデータ解析および可視化の機能を最大限に活用できます。

System Generator を使用したデザイン フロー

System Generator は、さまざまな状況で有益です。デザインをハードウェアに変換せずにアルゴリズムを解析する場合、System Generator デザインを大型デザインの一部として使用する場合、System Generator デザインをそのみで完成させ、FPGA ハードウェアで使用する場合があります。これらの 3 つの状況について説明します。

アルゴリズムの解析

System Generator は、特にアルゴリズムの解析、デザインのプロトタイプ作成、モデル解析で有益です。これらが目的である場合は、ツールを使用してアルゴリズムを具体化し、デザインで発生する可能性のある問題を調べたり、ハードウェアへのインプリメンテーションにおけるコストやパフォーマンスを予測できます。これらの作業は準備のためであり、デザインをハードウェアに変換する必要はほとんどありません。

この場合、詳細なインプリメンテーションを考慮することなく、デザインの主な部分を組み立てます。Simulink ブロックと MATLAB の M コードにより、シミュレーションおよび結果の解析用にステミュラスを供給します。リソースの予測では、ハードウェアにインプリメントしたデザインのコストを概算します。ハードウェア生成を使用したテストにより、可能なハードウェア スピードが示されます。

有効な方法が決まったら、デザインを具体化します。System Generator では、調整を段階的に行うことができるので、デザインの一部はハードウェアにインプリメントする準備が完了していても、その他の部分を抽象記述のままにしておくことができます。System Generator のハードウェア協調シミュレーション機能は、デザインを部分的に調整している場合に特に有益です。

大型デザインの一部としてインプリメント

System Generator は、大型デザインの一部をインプリメントするのによく使用されます。たとえば、System Generator はデータパスおよび制御をインプリメントするには適していますが、厳密なタイミング要件を持つ高度な外部インターフェイスにはそれほど適していません。この場合、System Generator を使用してデザインの一部をインプリメントし、ほかの部分を他のツールでインプリメントして、これらを組み合わせて 1 つの大型デザインを構成できます。

このフローでは、デザイン全体を表す HDL ラッパを作成し、System Generator の部分をコンポーネントとして使用するのが一般的な方法です。ほかのツールで作成した部分も、コンポーネントとしてラッパに含めるか、ラッパに直接インスタンスエートできます。

完全なデザインのインプリメント

デザインに必要なものがすべて System Generator に含まれている場合があります。この場合、System Generator トークンのパラメータ ダイアログ ボックスで [Generate] ボタンをクリックするだけで、デザインを HDL に変換し、ダウンストリーム ツールを使用して HDL の処理に必要なファイルを生成できます。生成されるファイルは、次のとおりです。

- デザインをインプリメントする HDL ファイル。
- デザインを含むクロック ラッパ。このクロック ラッパは、デザインに必要なクロックおよびクロック イネーブル信号を生成します。
- クロック ラッパを含む HDL テストベンチ。このテストベンチを使用すると、Simulink シミュレーションの結果をロジック シミュレータの結果と比較できます。

- XST や Synplify Pro などの合成ツールで System Generator HDL を処理できるようにするプロジェクト ファイルとスクリプト。
- System Generator HDL を Project Navigator のプロジェクトとして使用できるようにするファイル。

System Generator で生成されるファイルの詳細は、「[コンパイル結果](#)」を参照してください。

System Generator でのシステム レベルのモデリング

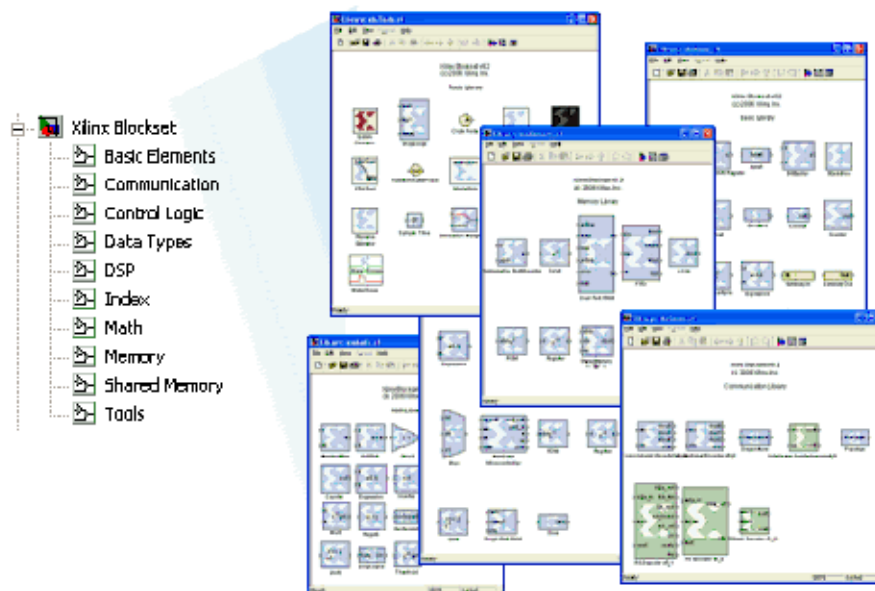
System Generator では、デバイス特有のハードウェア デザインを柔軟な高レベルのシステム モデリング環境で構築できます。System Generator デザインでは、信号は単なるビットではなく、符号付きまたは符号なしの固定小数点値にすることができ、デザインに変更を加えると、信号型も自動的に適切な型に変換されます。ブロックは単にハードウェアの代わりではなく、その周辺に応じて自動的に生成結果および最終的なハードウェアが調整されます。

System Generator では、さまざまな要素からデザインを構築できます。データ フロー モデル、ハードウェア設計言語 (VHDL、Verilog、EDIF)、および MATLAB プログラム言語による関数を同時に使用、シミュレーション、およびハードウェアに合成できます。System Generator のシミュレーション結果は、ビット精度およびサイクル精度であり、ハードウェアでの結果と厳密に一致します。System Generator のシミュレーションは従来の HDL シミュレータでのシミュレーションよりも高速で、結果も解析しやすくなっています。

System Generator ブロックセット	System Generator ブロックのライブラリへの分類方法、ブロックのパラメータ設定方法および使用方法を示します。
信号型	System Generator で使用されるデータ型と、ツールで自動的にデータ型を割り当てる方法を説明します。
ビット単位およびサイクル単位のモデリング	System Generator モデルの Simulink ベースのシミュレーションと、ハードウェアでの動作との関係を説明します。
タイミングとクロック	クロックのハードウェアへのインプリメント方法、および System Generator でクロックのインプリメンテーションを制御する方法を示します。System Generator でマルチレート Simulink モデルがどのようにクロック同期ハードウェアに変換されるかを説明します。
同期化のメカニズム	高レベル System Generator デザインにおいてデータ パス エlement間でデータ フローを同期化する方法と、制御パス ファンクションをインプリメントする方法を説明します。
ブロック マスクとパラメータの伝搬	Simulink でパラメータ指定システムおよびサブシステムを作成する方法を説明します。
リソースの予測	System Generator デザインをインプリメントするのに必要なハードウェアの予測方法を説明します。

System Generator ブロックセット

Simulink ブロックセットは、Simulink ブロック エディタで接続し、ダイナミックなシステムのファンクション モデルを作成するためのブロックのライブラリです。システムのモデリングでは、System Generator ブロックセットをその他の Simulink ブロックセットと同様に使用できます。ブロックは、数値演算、ロジック、メモリ、DSP ファンクションなどの抽象表現であり、高度な信号処理システムやその他のシステムを構築するために使用できます。また、System Generator コード生成ソフトウェアだけでなく、FDATool、ModelSim などその他のソフトウェア ツールへのインターフェイスとなるブロックもあります。



System Generator ブロックは、ビット精度およびサイクル精度です。ビット精度ブロックは、ハードウェアで生成される対応する値に一致した値を Simulink で生成し、サイクル精度ブロックは対応する時間に対応する値を生成します。

ザイリンクス ブロックセット

ザイリンクス ブロックセット (Xilinx Blockset) は、基本的な System Generator ブロックを含むライブラリを集めたものです。デバイス特有のハードウェアにアクセスする低レベルのブロックと、信号処理や高度な通信アルゴリズムなどをインプリメントする高レベルのブロックがあります。Gateway In/Gateway Out ブロックなど幅広く応用可能なブロックは、複数のライブラリに含まれています。Index ライブラリには、すべてのブロックが含まれています。次に、これらのライブラリについて説明します。

ライブラリ	説明
Index	ザイリンクス ブロックセットのすべてのブロック
Basic Elements	デジタル ロジックの標準基本ブロック
Communication	デジタル通信システムでよく使用される順方向誤り訂正ブロックおよびモジュレータ ブロック
Control Logic	制御回路およびステート マシン用のブロック
Data Types	データ型を変換するブロック (Gateway ブロックを含む)
DSP	DSP (デジタル信号処理) ブロック
Math	演算ファンクションをインプリメントするブロック
Memory	メモリをインプリメントするブロックおよびメモリにアクセスするブロック
Shared Memory	ザイリンクス共有メモリをインプリメントするブロックおよび共有メモリにアクセスするブロック
Tools	コード生成 (System Generator トークン)、リソース予測、HDL 協調シミュレーションなどを実行するユーティリティ ブロック

メモ：ブロックに関する詳細は、「[ザイリンクス ブロックセット](#)」を参照してください。

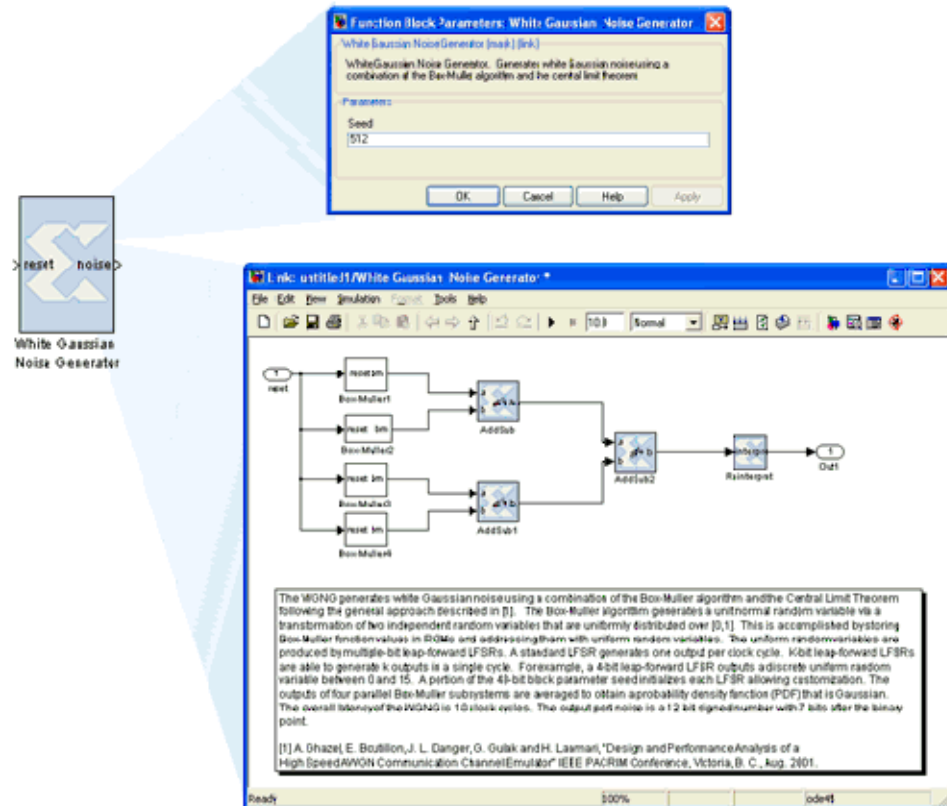
ザイリンクス リファレンス ブロックセット

ザイリンクス リファレンス ブロックセット (Xilinx Reference Blockset) には、さまざまなファンクションをインプリメントする複合 System Generator ブロックが含まれます。このブロックセットのブロックは、ファンクション別にライブラリに分類されています。次に、これらのライブラリについて説明します。

ライブラリ	説明
Communication	デジタル通信システムでよく使用されるブロック
Control Logic	制御回路およびステート マシン用のブロック
DSP	DSP (デジタル信号処理) ブロック
Imaging	イメージ処理ブロック
Math	演算ファンクションをインプリメントするブロック

このブロックセットの各ブロックは複合ブロックであり、ブロックをコンフィギュレーションするパラメータでマスク サブシステムとしてインプリメントされます。

ライブラリに含まれるブロックをそのまま使用するか、類似した特性を持つデザインを構築する際に開始点として使用できます。各リファレンス ブロックには、インプリメンテーションの説明およびハードウェア リソース要件が含まれています。各ブロックの詳細は、「[ザイリンクス リファレンス ブロックセット](#)」にも含まれています。



信号型

ハードウェアのビット精度シミュレーションを実行するには、System Generator ブロックがブール値および任意の精度の固定小数点値で動作する必要がありますが、Simulink の基本的なスカラ信号型は、倍精度浮動小数点です。ザイリンクス ブロックとザイリンクス以外のブロックの間の接続には、Gateway ブロックを使用します。Gateway In ブロックは倍精度の信号をザイリンクス信号に変換し、Gateway Out ブロックはザイリンクス信号を倍精度の信号に変換します。Simulink の連続タイミング信号は、Gateway In ブロックでサンプリングします。

ほとんどのザイリンクス ブロックは多様型であり、入力型に応じて適切な出力型を推測できます。ブロックのパラメータ ダイアログ ボックスで完全精度が指定されている場合は、精度が失われないように出力型が選択されます。必要に応じて、符号拡張および 0 のパディングが自動的に行われます。ユーザー指定の精度も使用できます。この設定では、ブロックの出力型と、量子化およびオーバーフローの処理方法を指定できます。量子化の処理方法としては、正または負の無限大への不偏丸め (符号によって異なる) または切り捨てがあります。オーバーフローの処理方法には、正または負の最大値を使用するか、切り捨てるか、オーバーフローをエラーとしてレポートするかのオプションがあります。

メモ : System Generator のデータ型は、Simulink で [書式] → [ポート/信号の表示] → [端子のデータタイプ] をクリックすると表示されます。データ型を表示すると、モデルの精度を簡単に判断できます。たとえば、ポートのデータ型が Fix_11_9 の場合は、信号は小数点以下のビットが 9 桁の 2 の補数符号付き 11 ビット値であり、Ufix_5_3 の場合は小数点以下のビットが 3 桁の符号なし 5 ビット値です。

Simulink モデルの System Generator 部分では、すべての信号をサンプリングする必要があります。サンプリング時間は、Simulink の伝搬ルールにより自動的に設定されるようにするか、ブロックのパラメータ ダイアログ ボックスで明示的に設定できます。フィードバック ループがあると、System Generator でサンプリング周期および信号型を推論できない場合もあり、その場合はエラーメッセージが表示されます。フィードバック ループには Assert ブロックを挿入して、この問題を回避する必要があります。ループ内のすべてのポイントに Assert ブロックを追加する必要はありません。通常、ループの 1 箇所に追加するだけで十分です。

メモ : Simulink では、ブロックと信号を実行レートに応じて色分けして表示できます (Simulink メニューから [書式] → [ポート/信号の表示] → [サンプル時間の色分け表示] をクリック)。これは、マルチレート デザインを解析するのに有益です。

ビット単位およびサイクル単位のモデリング

System Generator でのシミュレーションは、ビット単位またはサイクル単位で実行されます。ビット単位のシミュレーションとは、System Generator ブロックと System Generator 以外のブロックの境界で、シミュレーションで生成された値がハードウェアで生成された対応する値とビット単位で同一であるということです。サイクル単位のシミュレーションとは、System Generator ブロックと System Generator 以外のブロックの境界で、対応する値が対応する時間に生成されるということです。デザインの境界は、System Generator の Gateway ブロックが配置されている部分です。デザインがハードウェアに変換されると、Gateway In ブロックは最上位入力ポート、Gateway Out ブロックは最上位出力ポートになります。

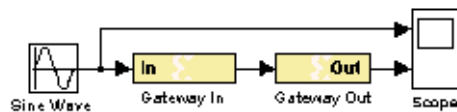
タイミングとクロック

離散時間システム

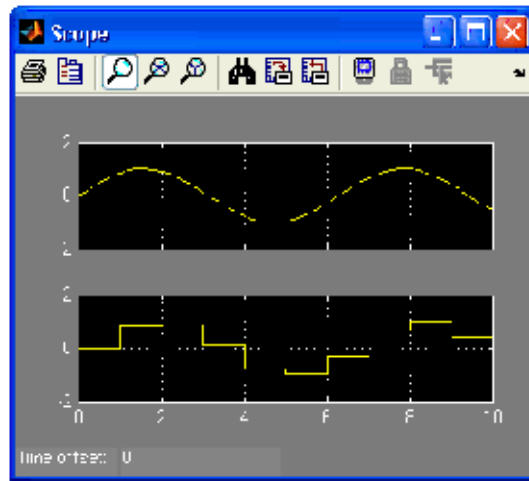
System Generator のデザインは離散時間システムであるので、信号とその信号を生成するブロックにはサンプリング レートがあります。ブロックのサンプリング レートは、ブロックのステートがアップデートされる頻度を決定します。System Generator では、ほとんどのサンプリング レートが自動的に設定されますが、サンプリング レートを明示的または暗示的に設定する必要のあるブロックもあります。

メモ : Simulink の離散時間システムとサンプリング時間の詳細は、MathWorks 社のマニュアル『Using Simulink』を参照してください。

単純な System Generator モデルで、離散時間システムの動作を示します。次の図に示すモデルがあるとします。Simulink ソース (Sine Wave) により Gateway In ブロックが駆動され、Gateway Out ブロックで Simulink シンク (Scope) が駆動されています。



Gateway In ブロックは、1 秒のサンプリング周期でコンフィギュレーションされています。Gateway Out ブロックは、ザイリンクス固定小数点信号を Simulink の Scope で解析できるように倍精度に変換しますが、サンプリングレートは変更しません。Scope の出力は、サンプリングレートが変更されていない、サンプリングされたサイン波となります。



マルチレート モデル

System Generator では、信号が複数のサンプリングレートで動作するマルチレート デザインがサポートされています。マルチレート モデルは、System Generator で自動的にハードウェアにコンパイルされます。マルチレート デザインは、Simulink に適した直接的な方法でインプリメントされます。

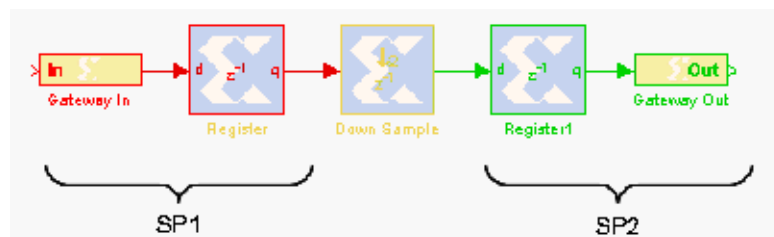
レート変換ブロック

System Generator には、サンプリングレートを変換するブロックが含まれています。最も基本的なレート変換ブロックは、Up Sample と Down Sample ブロックです。これらのブロックは、次の図に示すようにパラメータ ダイアログ ボックスで指定した固定の値を乗算することにより、レートを変換します。



Parallel To Serial や Serial To Parallel などのその他のブロックは、ブロックのパラメータ指定に応じて非間接的にレートを変換します。

次のような単純なマルチレート システムがあるとします。このモデルでは、SP1 と SP2 の 2 つのサンプリング周期が使用されます。サンプリング周期 SP1 は、Gateway In のパラメータ ダイアログ ボックスで指定します。Down Sample ブロックによりモデルのレートが変更され、SP1 の 1/2 である新しいレート SP2 が作成されます。



ハードウェア オーバーサンプリング

一部の System Generator ブロックは、そのブロックのデータ レートより高速のレートで内部処理が行われます (オーバーサンプリング)。ハードウェアでは、これはデータ サンプルを処理するのに複数のクロック サイクルが必要であることを意味します。Simulink では、これらのブロックのサンプリング レートに計測される変化はありません。

オーバーサンプリングされるブロックの 1 つに DAFIR FIR フィルタがあります。サンプルはシリアルに処理されるので、高速レートで動作しますが、使用されるハードウェアは少なく済みます。

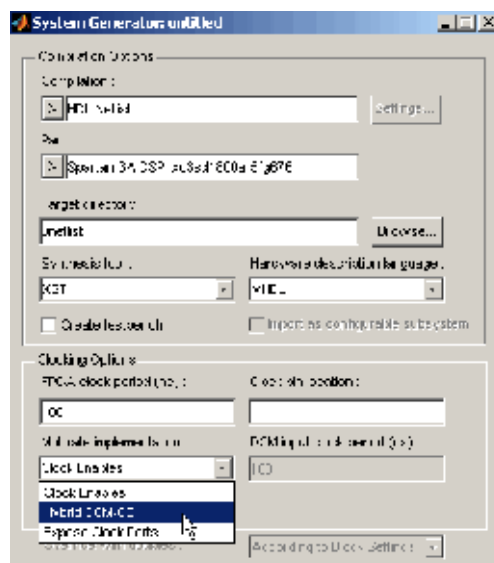
Simulink ではオーバーサンプリングされるブロックでサンプリング レート が変化することはありませんが、System Generator では、ハードウェア インプリメンテーション用のクロック ロジックを生成する際に、サンプリング レート と共に内部ブロック レート も考慮されるので、System Generator トークンのパラメータ ダイアログ ボックスで Simulink のシステム周期を指定する際に、オーバーサンプリングされるブロック の内部処理時間も 考慮する必要があります。

非同期クロック

System Generator は、1 つのクロックに同期するハードウェアの設計に適していますが、場合によっては、複数のクロックを使用するシステムの設計にも使用できます。この場合、デザインをクロック ドメインに分割し、ドメイン間での情報転送をデュアル ポート メモリおよび FIFO で制御します。System Generator では、Simulink でシミュレーションし、完全なハードウェア記述を生成することも含め、このようなマルチ クロック デザインがサポートされています。詳細は、「[複数クロックのサイクル単位アイランドの生成](#)」を参照してください。このセクションの残りの部分で、System Generator のクロック同期について説明します。この内容は、1 つのクロックのデザインおよび複数クロックのデザインの両方に関係します。

同期クロック

System Generator トークンを使用してデザインをハードウェアにコンパイルする場合、次の図に示すように、マルチレート インプリメンテーション用に [Clock Enables] (デフォルト)、[Hybrid DCM-CE]、および [Expose Clock Ports] の 3 つのクロック供給オプションがあります。



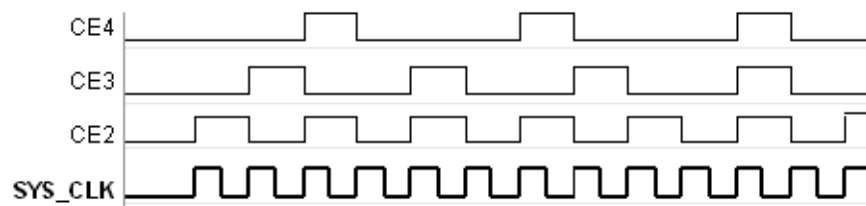
[Clock Enables] オプション

System Generator で [Clock Enables] オプションを選択してモデルをハードウェアにコンパイルすると、ハードウェアの対応する部分が適切なレートで動作するようにデザインのサンプリング レート情報が保持されます。System Generator は、クロックとクロック イネーブル (1 つのレートに 1 つのイネーブル) を組み合わせて、関連するレートをハードウェアで生成します。各クロック イネーブルの周期は、システム クロックの周期の整数倍です。

Simulink 内では、System Generator デザインの信号としてクロックおよびクロック イネーブルを明示的に追加する必要はありません。System Generator でデザインをハードウェアにコンパイルする際、デザインのサンプルレート (具体的には System Generator トークンの 2 つのユーザー指定値である Simulink システム周期と FPGA クロック周期) から必要なクロック イネーブルが推論されます。これらの値は、Simulink シミュレーションの時間と実際のハードウェア インプリメンテーションでの時間のスケール係数を定義します。Simulink システム周期は、モデルに含まれるサンプリング周期の最大公約数 (gcd) にする必要があり、FPGA のクロック周期 (ns) はシステム クロックの周期です。Simulink システム周期を p 、FPGA システム クロック周期を c とすると、Simulink で kp かかる処理は、ハードウェアではシステム クロックの k サイクル分 (kc ナノ秒) になります。

たとえば、3 つの Simulink サンプリング周期 2、3、4 を含むモデルがあるとします。これらのサンプリング周期の gcd は 1 です。FPGA クロック周期は 10ns に設定されているとします。これらの情報から、ハードウェアでの対応するクロック イネーブルの周期を決定できます。

Simulink のサンプリング周期 2、3、4 に対応するハードウェアでのクロック イネーブルを CE2、CE3、CE4 とします。各クロック イネーブルの周期とシステム クロック周期の関係は、対応する Simulink サンプリング周期を Simulink システム周期で割ることにより求められます。この結果、CE2、CE3、CE4 の周期はそれぞれ 2、3、4 システム クロック周期になります。次の図に、これらのクロック イネーブル信号のタイミングを示します。



[Hybrid DCM-CE] オプション

インプリメンテーション ターゲットがデジタル クロック マネージャ (DCM) を含む FPGA の場合、クロック ツリーを DCM で駆動できます。DCM を使用すると、クロック イネーブル ネットのファンアウトが大きく、タイミング クロージャの達成が困難な場合に有益です。

System Generator では DCM が最上位 HDL クロック ラッパにインスタンス化され、異なるレートのクロック ポートを Virtex-4 および Virtex-5 では 3 つまで、Spartan®-3A DSP では 2 つまで供給できます。デザインに DCM でサポート可能な数以上のクロック ポートが含まれる場合は、残りのクロックは CE (クロック イネーブル) コンフィギュレーションでサポートされます。レートは、 $CLK0 > CLK2x > CLKdv > CLKfx$ という優先順で DCM 出力にマップされます。クロック レートの高いものが DCM でサポートされます。

dcm_reset 入力ポートは最上位ラッパに含まれ、ビットストリーム コンフィギュレーション後に外部デザインで DCM をリセットできるようになっています。dcm_locked 出力も最上位ラッパに含まれており、外部デザインで入力データを 1 つの clk 入力ポートに同期させるのに役立ちます。

既知の制限 : 次の System Generator ブロックでは、[Hybrid DCM-CE] オプションはサポートされていません。

- Clock Enable Probe
- Clock Probe
- DAFIR
- Down Sample ([First value of frame] がオンの場合)
- FIR Compiler (コア レートが入力サンプリング レートと異なる場合)
- Parallel to Serial ([Latency] が 0 に設定されている場合)
- Time Division Demultiplexer
- Time Division Multiplexer
- Up Sample ([Copy samples (otherwise zeros are inserted)] がオフの場合)

[Expose Clock Ports] オプション

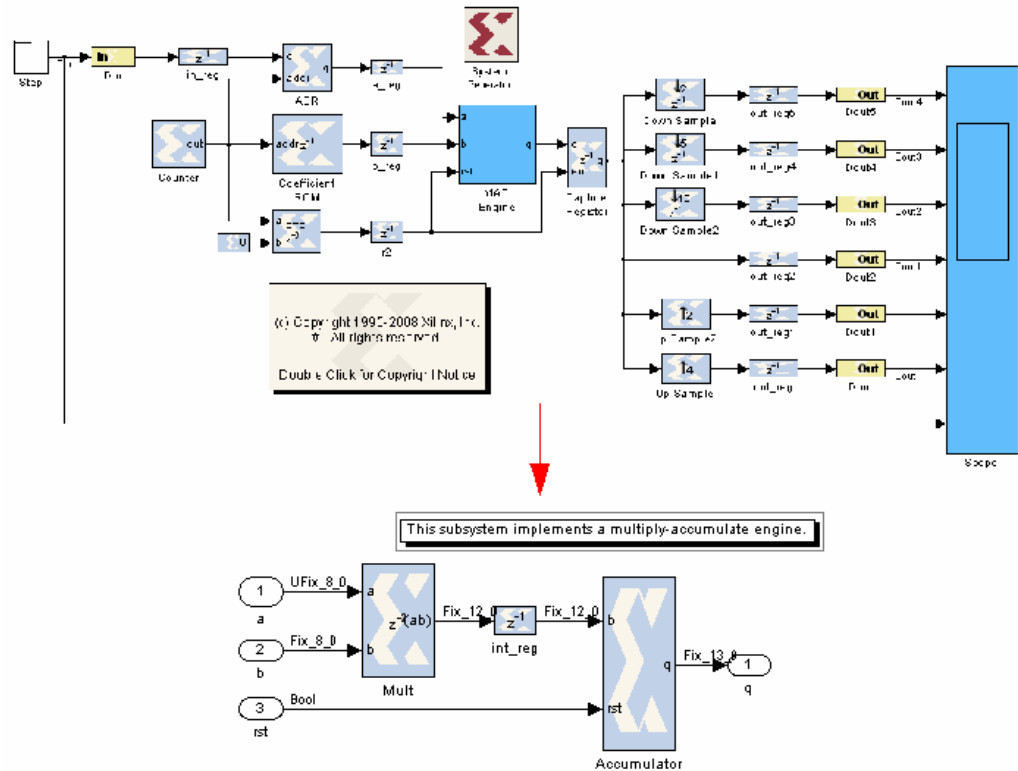
このオプションを選択すると、各レートのクロック ポートを含む最上位ラップが作成されます。これにより、デザインの外部にクロック ジェネレータを手動でインスタンス化して、クロック ポートを駆動できます。

チュートリアル : [Hybrid DCM-CE] オプションの使用

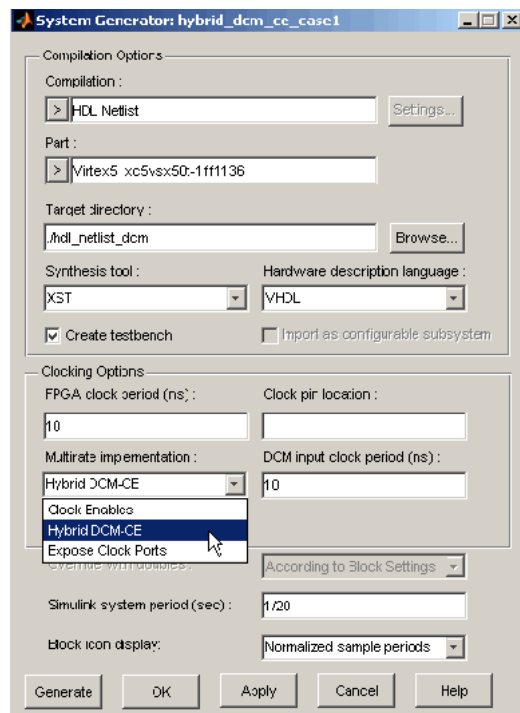
次の例では、[Hybrid DCM-CE] オプションを選択し、HDL デザインのネットリストを作成して ISE® でデザインをインプリメントし、デザインをシミュレーションして、DCM が適切にインスタンス化され、コンフィギュレーションされていることをファイルおよびレポートを参照して検証します。

hybrid_dcm_ce_case1.mdl デザイン例は、<path_to_sysgen>\examples\clocking_options\hybrid_dcm_ce_case1\hybrid_dcm_ce_case1.mdl にあります。

1. MATLAB でモデルを開き、次のブロックを確認します。
 - Addressable Shift Register (ASR) : 入力遅延バッファをインプリメントするために使用されます。アドレス ポートはデータ ポートの n 倍の速度で動作します。 n はフィルタ タップの数で、この例では 5 です。
 - Coefficient ROM : フィルタ係数を保存するのに使用されます。
 - Counter : ROM および ASR のアドレスを生成するのに使用されます。
 - Comparator : リセット信号およびイネーブル信号を生成するのに使用されます。
 - MAC Engine : フィルタの積和演算子として使用されます。



2. System Generator トークンをダブルクリックしてパラメータ ダイアログ ボックスを開きます。



[Hybrid DCM-CE] を選択し、[Generate] をクリックします。現在の作業ディレクトリに hdl_netlist_dcm というサブディレクトリが作成され、生成されたファイルが保存されます。

3. MATLAB の [Current Directory] ウィンドウで、hybrid_dcm_ce_case1_sysgen.log ファイルをダブルクリックします。次の図に示すように、DCM クロックが最初にリストされ (最高レートから順にリスト)、次に CE で駆動されるクロックがリストされます。

----- DCM Clock Outputs -----		
Normalized Period	DCM Output Used	Frequency
1	CLKO	100.000C
2	CLKFX	50.000C
4	CLKDV	25.000C
8	(CLKDV,ce_8)	12.500C
20	(CLKDV,ce_20)	5.000C
40	(CLKDV,ce_40)	2.500C
----- ** -----		

4. ISE を起動し、<path_to_sysgen>\examples\clocking_options\hybrid_dcm_ce_case1\hdl_netlist\hybrid_dcm_ce_case1_dcm_mcw.ise という ISE プロジェクトを開きます。
5. Project Navigator の [Design] パネルの [Processes] ペインで、[Implement Design] をダブルクリックします。
6. Project Navigator の [Design] パネルの [Hierarchy] ペインで次の操作を実行します。
- hybrid_dcm_ce_case1_dcm_mcw.vhd をダブルクリックして開き、次の VHDL コードに示す DCM コンポーネント宣言の部分にスクロールします。

```

580  component DCM
581  generic (
582      CLKDV_DIVIDE: real := 4.0;
583      CLKFX_MULTIPLY: integer := 2;
584      CLKFX_DIVIDE: integer := 4;
585      FFS_FREQUENCY_MODE: string := "LOW";
586      DLL_FREQUENCY_MODE: string := "LOW";
587      CLKIN_PERIOD: real := 10.0;
588      CLKIN_DIVIDE_BY_2: boolean := false;
589      CLKCUT_PHASE_SHIFT: string := "NONE";
590      CLK_FEEDBACK: string := "1X";
591      PHASE_SHIFT: integer := 0
592  );

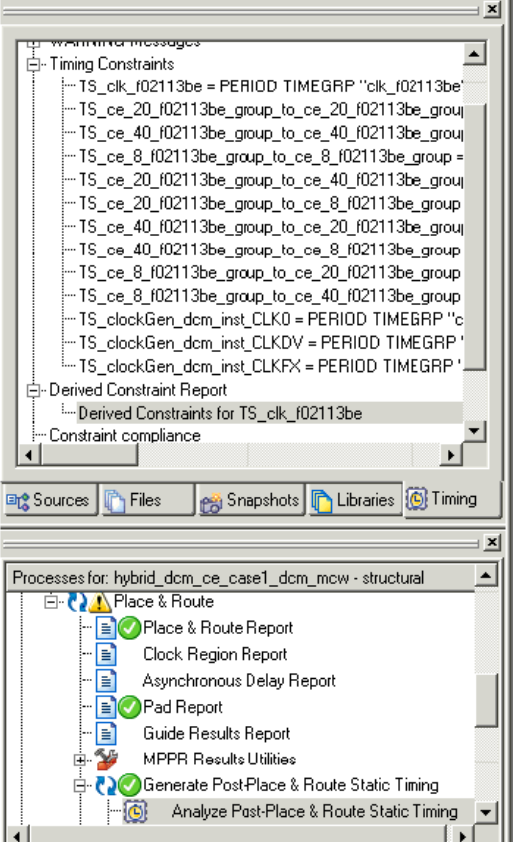
```

- System Generator により DCM インスタンスが自動的にインスタンス化され、クロック出力の要件に応じてパラメータが設定されていることを確認します。
- VHDL ファイルを閉じます。

次に、ISE タイミング レポート を参照してクロックの伝搬を調べます。まず、レポートを生成します。

7. [Processes] ペインで [Implement Design] → [Place & Route] → [Generate Post-Place & Route Static Timing] を展開表示します。

8. [Analyze Post-Place & Route Static Timing] をダブルクリックします。次のような情報が表示されます。



The screenshot shows the 'Derived Constraint Report' window in the Xilinx System Generator. The report is titled 'Derived Constraints for TS_clk_f02113be' and contains a table with two columns: 'Constraint' and 'Period Requirement'.

Constraint	Period Requirement
TS_clk_f02113be	10.000ns
TS_clockGen_dcm_inst_CLK0	10.000ns
TS_clockGen_dcm_inst_CLKDV	40.000ns
TS_clockGen_dcm_inst_CLKFX	20.000ns

The background window shows the 'Processes for: hybrid_dcm_ce_case1_dcm_mcw - structural' pane with the 'Analyze Post-Place & Route Static Timing' process selected.

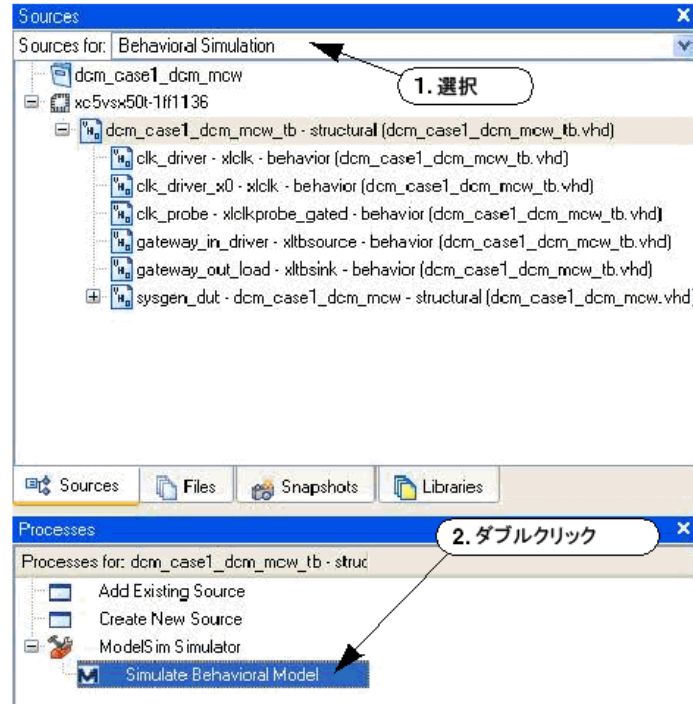
このデザインでは、10ns のグローバル クロック制約を基準とした 6 つのクロック レート (1、2、4、8、20、40) が使用されています。タイミング レポートでは、System Generator により次のクロックが正しく生成され、伝搬されていることが示されます。

- ◆ DCM ベースのクロック : clk_1 (CLK0 -> 10ns)、clk_2 (CLKFX -> 20ns)、clk_4 (CLKDIV -> 40ns) を 10ns のグローバル クロック入力を基準に DCM を使用して生成
- ◆ クロック イネーブル ベースのクロック : ce_8 (80ns)、ce_20 (200ns)、ce_40 (400ns) を clk_4 クロック入力を基準にクロック イネーブルを使用して生成

次に、ModelSim を使用してビヘイビア シミュレーションを実行します。

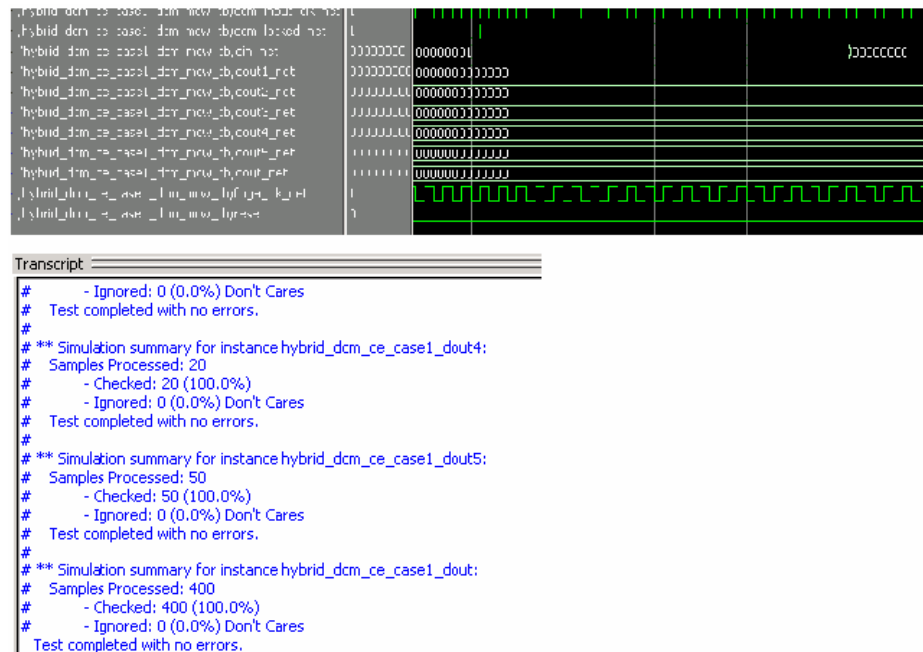
9. 次の図に示すように、[Hierarchy] ペインの [Sources for] ドロップダウンリストから [Behavioral Simulation] を選択します。

メモ : System Generator では、最上位ラップ VHDL テストベンチ、スクリプト ファイル、および入力/出力ステイミュラス データ ファイルが自動的に生成されます。[Processes] ペインに表示されるプロセスは、[Hierarchy] ペインで選択したソース タイプによって異なります。



10. [Processes] ペインで [Simulate Behavioral Model] をダブルクリックし、シミュレーションを実行します。

11. シミュレーションが終了すると、次の図に示すような波形が表示されます。



すべての DCM クロック (clk_1、clk_2、および clk_4) は、最上位ラップ テストベンチ ファイル (hybrid_dcm_ce_case1_dcm_mcw_tb.vhd) に含まれています。

まとめ

[Hybrid DCM-CE] オプションを選択すると、System Generator により自動的に DCM がインスタンス化され、手動で調整する必要はありません。また、異なるクロックレートが DCM および CE を使用して適切に生成され、最適な QoR および低消費電力を達成できます。属性を設定したり DCM クロック出力を指定したりする必要はありません。[Hybrid DCM-CE] オプションを選択すると、[Clock Enables] オプションを選択した場合と比較して、クロック スキューを最小限に抑えることができます。

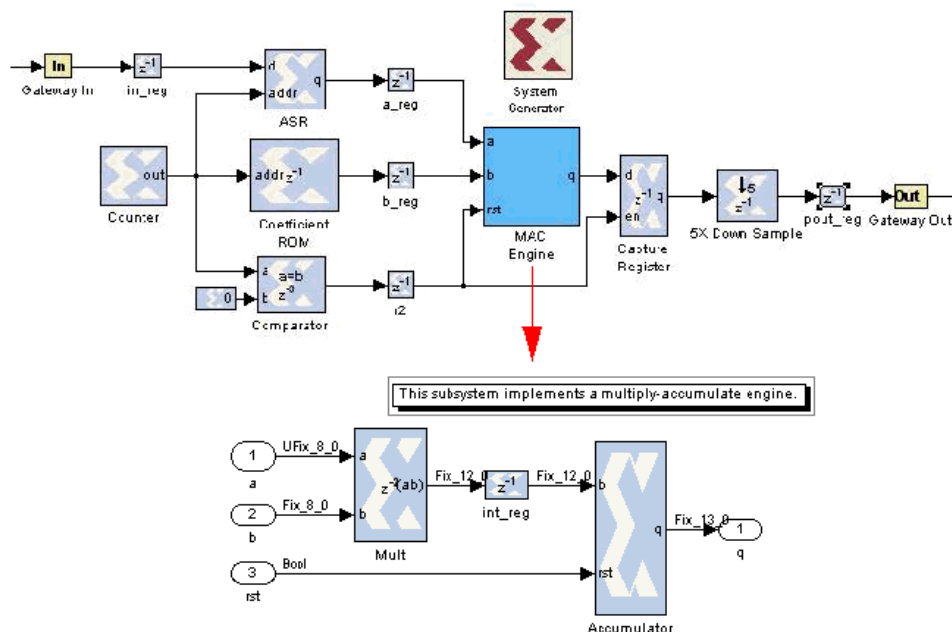
チュートリアル : [Expose Clock Ports] オプションの使用

次の例では、[Expose Clock Ports] オプションを選択し、HDL デザインのネットリストを作成して ISE でデザインをインプリメントして、デザインをシミュレーションして、ファイルおよびレポートを参照してデザインを検証します。

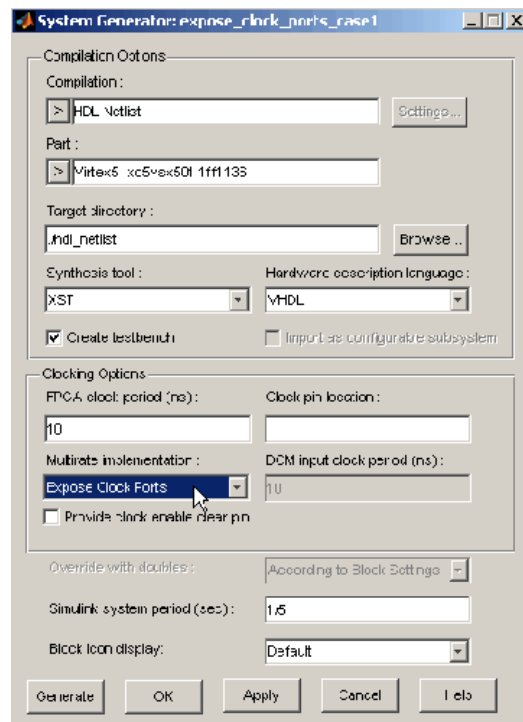
expose_clock_ports_case1 デザイン例は、<path_to_sysgen>\examples\clocking_options\expose_clock_ports_case1\expose_clock_ports_case1.mdl にあります。

1. MATLAB でモデルを開き、次のブロックを確認します。

- **Addressable Shift Register (ASR)** : 入力遅延バッファをインプリメントするために使用されます。アドレスポートはデータポートの n 倍の速度で動作します。 n はフィルタ タップの数で、この例では 5 です。
- **Coefficient ROM** : フィルタ係数を保存するのに使用されます。
- **Counter** : ROM および ASR のアドレスを生成するのに使用されます。
- **Comparator** : リセット信号およびイネーブル信号を生成するのに使用されます。
- **MAC Engine** : フィルタの積和演算子として使用されます。



2. System Generator トークンをダブルクリックして次のダイアログ ボックスを開きます。



[Expose Clock Ports] を選択し、[Generate] をクリックします。現在の作業ディレクトリに hdl_netlist というサブディレクトリが作成され、生成されたファイルが保存されます。

3. ISE を起動し、<path_to_sysgen>\examples\clocking_options\expose_clock_ports_case1\hdl_netlist\expose_clock_ports_case1_mcw.isc という ISE プロジェクトを開きます。
4. Project Navigator の [Design] パネルの [Processes] ペインで [Implement Design] をダブルクリックします。
5. Project Navigator の [Design] パネルの [Hierarchy] ペインで次の操作を実行します。
 - a. expost_clock_ports_case1_mcw.vhd をダブルクリックして開き、次の VHDL コードに示す expose_clock_ports_case1_mcw というエンティティの部分にスクロールします。

```

37 library IEEE;
38 use IEEE.std_logic_1164.all;
39 use work.conv_pkg.all;
40
41 entity expose_clock_ports_case1_mcw is
42   port (
43     clk_1: in std_logic; -- clock period = 10.0 ns (100.0 Mhz)
44     clk_5: in std_logic; -- clock period = 50.0 ns (20.0 Mhz)
45     gateway_in: in std_logic_vector(7 downto 0);
46     gateway_out: out std_logic_vector(12 downto 0)
47   );
48 end expose_clock_ports_case1_mcw;

```

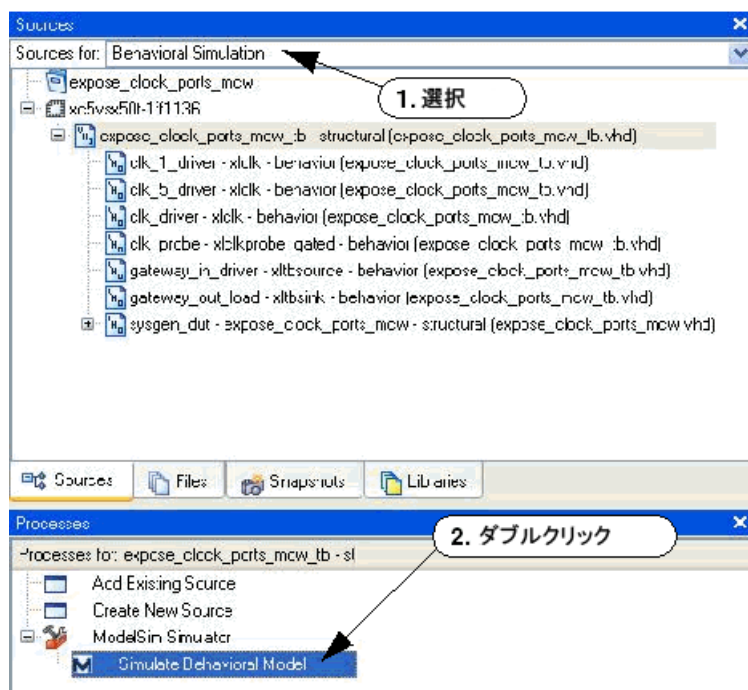
- b. System Generator により異なるレートに基づくクロックが挿入され、最上位ラッパにクロックポートが含まれていることを確認します。このデザインでは 2 つのクロック レート が使用されるので、clk_1 と clk_5 の 2 つのクロック が最上位ラッパに含まれています。これにより、複数の同期クロックを System Generator デザインの外から直接駆動できます。

c. VHDL ファイルを閉じます。

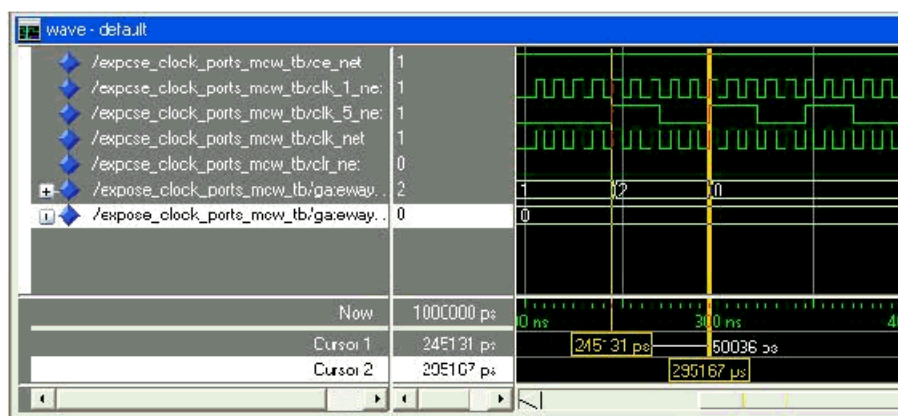
次に、ModelSim を使用してビヘイビア シミュレーションを実行します。

6. 次の図に示すように、[Hierarchy] ペインの [Sources for] ドロップダウン リストから [Behavioral Simulation] を選択します。

メモ：System Generator では、最上位ラッパ VHDL テストベンチ、スクリプト ファイル、および入力/出力ステミュラス データ ファイルが自動的に生成されます。[Processes] ペインに表示されるプロセスは、[Hierarchy] ペインで選択したソース タイプによって異なります。



7. [Processes] ペインで [Simulate Behavioral Model] をダブルクリックし、シミュレーションを実行します。
8. シミュレーションが終了すると、次の図に示すようなシミュレーション波形が表示されます。



まとめ

[Expose Clock Ports] オプションを選択すると、System Generator により自動的にデザイン レートに適切なクロックが挿入され、クロック ポートが最上位ラップに含まれます。クロック レートは、[Clock Enables] オプションを選択した場合と同じ方法で判断されます。これにより、最上位ラップに含まれるクロック ポートを、外部同期クロック ソースで駆動できます。

同期化のメカニズム

System Generator では、同期化のメカニズムは自動的に作成されません。設計者が明示的に作成する必要があります。

有効なポート

System Generator には、同期化に使用できるブロック (特に FIFO) が複数含まれています。これらのブロックには、入力 (または出力) サンプルが有効になると指定される入力 (または出力) ポートがあります。これらのポートはチェーン接続でき、プリミティブ形式のフロー制御が可能です。FFT、FIR、Viterbi などのブロックにこれらのポートが含まれます。

不定データ

多くのハードウェア シミュレーション環境では、不定値があるのが一般的です。これらは、「ドントケア」または「X」と示されます。特に System Generator シミュレーションでの値は、不定値である可能性があります。たとえば、デュアル ポート メモリ ブロックでは、メモリの両方のポートで同じアドレスに同時にアクセスしようとする、値が不定になります。ハードウェアでの実際の動作は、どちらのポートのクロック エッジが先に到着するかなどを決定するインプリメンテーションの詳細によって異なります。値が不定になることを許容すると、柔軟性が増します。先ほどの例で、メモリで値が不定になっても、その後の処理がその値に依存していなければ、問題ありません。

HDL 協調シミュレーションによりシミュレーションに含まれる HDL モジュールは、一般的にデータ サンプルが不定になる原因となります。System Generator で HDL 協調シミュレーション モジュールの入力に入力される不定値は、標準ロジック ベクタ XXX...XX で表されます。

Gateway Out を駆動する不定値は、NaN (Not a Number) という値になります。Simulink の Scope では、NaN 値は表されません。Gateway In を駆動する NaN も不定値になります。System Generator には、不定値を検出する Indeterminate Probe ブロックが含まれています。このブロックは、ハードウェアには変換されません。

System Generator では、演算信号が不定値になってもかまいませんが、ブール信号を不定値にすることはできません。シミュレーションでブール信号が不定値になる状況が発生した場合は、シミュレーションは中断され、エラー メッセージが表示されます。ザイリンクス ブロックには、ブール信号のみを入力として使用可能な制御ポートが含まれているものが多数あります。これらのブロックでは、制御ポートのブール信号を不定値にすることはできません。

UFix_1_0 は、ブール信号と同等のデータ型ですが、不定値に関する上記の制限はありません。

ブロック マスクとパラメータの伝搬

通常の Simulink ブロックに適用されるスコーピング ルールとパラメータ伝搬ルールは、System Generator ブロックにも適用されます。つまり、ザイリンクス ブロックセット内のブロックは、MATLAB の変数および論理式を使用してパラメータ指定できます。この機能により、MATLAB 言語の表現能力および計算能力を活用した高度なパラメータ指定デザインを作成できます。

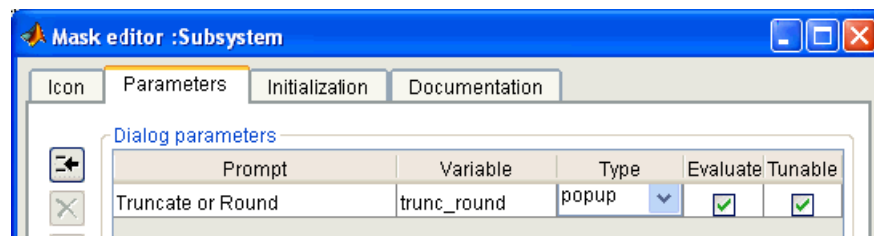
ブロック マスク

Simulink では、マスクと呼ばれるメカニズムでブロックのパラメータを指定します。実際には、ブロックにマスク変数を割り当て、この変数の値をダイアログ ボックスで指定するか、マスク初期化コマンドで算出します。この変数は、マスク ワークスペースに保存されます。マスク ワークスペースは、マスクが適用されるブロックでのみ使用され、外部ブロックからアクセスすることはできません。

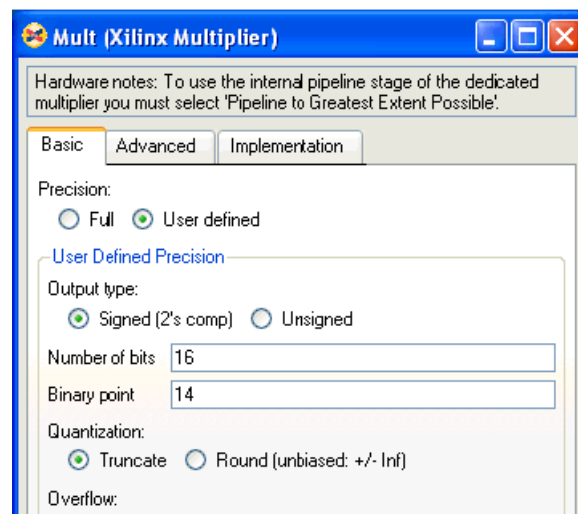
メモ： マスクでグローバル変数および基本ワークスペースの変数にアクセスすることは可能です。基本ワークスペースの変数にアクセスするには、MATLAB の `evalin` 関数を使用します。MATLAB と Simulink のスコーピング ルールの詳細は、The MathWorks 社のマニュアル『Using MATLAB』および『Using Simulink』を参照してください。

パラメータの伝搬

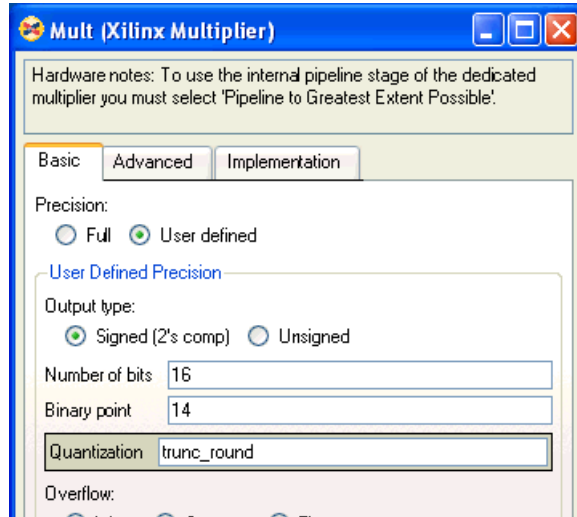
マスク サブシステム内のブロックに変数を渡す必要があることがよくあります。変数を渡すことにより、サブシステム内のパラメータによりブロックのコンフィギュレーションを判断させることができます。この手法は、ザイリンクス ブロックセット内のブロックのパラメータにも適用されます。たとえば、**Mult** ブロックと **Accumulator** ブロックで構成されるサブシステムを構築する場合、結果を切り捨てるか丸めるかを指定するパラメータをサブシステム内に作成できます。次の図では、このパラメータに `trunc_round` という名前が付けられています。



Mult ブロックと **Accumulator** ブロックのパラメータ ダイアログ ボックスには、[Truncate] (切り捨て) または [Round] (丸め) を選択するラジオ ボタンがあります。



ラジオ ボタンではなくパラメータを使用して選択する場合は、ラジオ ボタンを右クリックして [Define With Expression] をクリックすると、MATLAB の論理式をパラメータの設定に使用できます。次の例では、サブシステム マスクからの `trunc_round` パラメータを **Mult** ブロックと **Accumulator** ブロックの両方で使用し、サブシステムのマスク変数からの同じ設定が適用されるようにしています。



リソースの予測

System Generator では、デザインをインプリメントするのに必要な FPGA ハードウェア リソースを予測するツールが提供されています。スライス、ルックアップ テーブル、フリップフロップ、ブロック メモリ、エンベデッド乗算器、I/O ブロック、トライステート バッファの数が予測されます。これらの予測により、デザインの選択がハードウェア要件に与える影響を調べることができます。サブシステムに必要なリソースを予測するには、サブシステムに **Resource Estimator** ブロックをドラッグし、このブロックをダブルクリックして [Estimate] ボタンをクリックします。

自動コード生成

System Generator は、デザインを自動的に低レベル表現にコンパイルします。モデルのコンパイル方法は、System Generator トークンでの設定によって異なります。ハードウェアの HDL 記述に加え、補助ファイルも生成されます。プロジェクト ファイルや制約ファイルなどのファイルはダウンロード ツールで使用され、VHDL テスト ベンチなどのファイルはデザインの検証に使用されます。

System Generator トークンを使用したコンパイルとシミュレーション

コンパイル結果

HDL テストベンチ

System Generator トークンを使用してデザインを低レベル HDL にコンパイルする方法を説明します。

System Generator トークンで [HDL Netlist] を選択して [Generate] ボタンをクリックしたときに生成される低レベルファイルについて説明します。

System Generator で生成される VHDL テストベンチについて説明します。

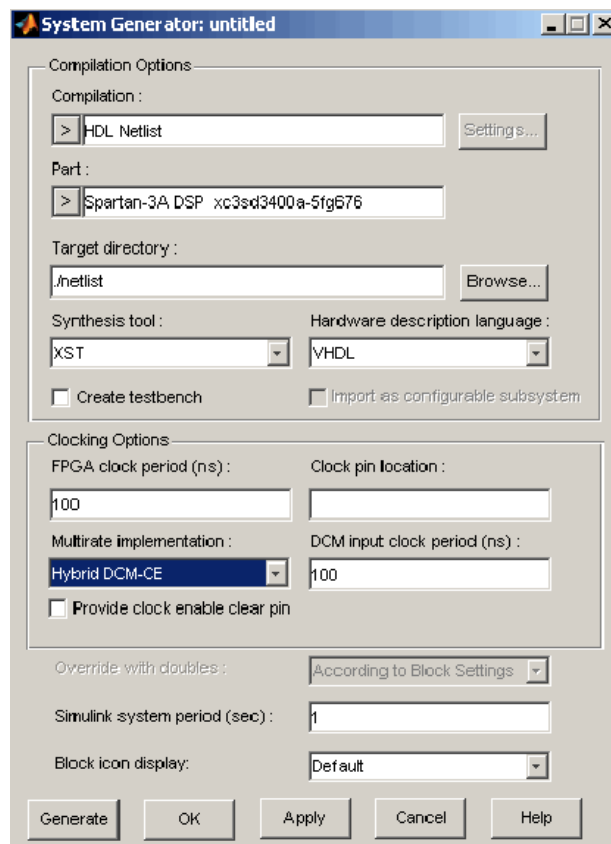
System Generator トークンを使用したコンパイルとシミュレーション

System Generator は、デザインを自動的に低レベル表現にコンパイルします。デザインをコンパイルまたはシミュレーションするには、**System Generator** トークンを使用します。このセクションでは、このブロックの使用法を説明します。

System Generator デザインをシミュレーションまたはハードウェアに変換するには、デザインに **System Generator** トークンを含める必要があります。新しいデザインを作成する際には、**System Generator** トークンをすぐに追加するようにすることをお勧めします。**System Generator** トークンは、Xilinx Blockset の Basic Element および Tools ライブラリに含まれています。また、その他のザイリンクスブロックと同様 Index ライブラリにも含まれています。

デザインには、**System Generator** トークンを少なくとも 1 つ含める必要があり、複数の **System Generator** トークンを異なるレベルに含めることが可能です (レベルごとに 1 つ)。上位に別の **System Generator** トークンがあるものはスレーブ、上位に別の **System Generator** トークンがないものはマスタとなります。1 つの **System Generator** トークンの適用範囲は、トークンが挿入されているレベルと、そのレベルの下にあるすべてのサブシステムです。[Simulink system period] などの一部のパラメータは、マスタでしか指定できません。

System Generator トークンを追加したら、コードの生成およびシミュレーションの処理方法を指定できます。次の図に、**System Generator** トークンのパラメータ ダイアログ ボックスを示します。



コンパイル タイプと [Generate] ボタン

[Generate] ボタンをクリックすると、System Generator でデザインの一部が低レベルにコンパイルされます。コンパイルされる部分は、ブロックを含むサブシステムがルートとなっている部分です。デザイン全体をコンパイルするには、System Generator トークンをデザインの最上位に配置します。コンパイル タイプ ([Compilation]) は、生成する結果のタイプを指定します。次のオプションがあります。

- [HDL Netlist]、[NGC Netlist] : HDL ネットリスト、NGC ネットリストを生成します。
- [Bitstream] : ハードウェア FPGA プラットフォームで実行可能な FPGA コンフィギュレーション ビットストリームを生成します。
- [EDK Export Tool] : ザイリンクス エンベデッド開発キット (EDK) にエクスポートするファイルを生成します。
- [Hardware Co-Simulation] : ハードウェア協調シミュレーション用のファイルを生成します。
- [Timing and Power Analysis] : デザインのタイミングおよび消費電力に関するレポートを生成します。

[HDL Netlist] が最も頻繁に使用されます。この場合、HDL ファイルおよび EDIF ファイルと、ダウンストリームの処理を簡略化するための補助ファイルが生成されます。HDL ファイルおよび EDIF ファイルは XST などの合成ツールで処理し、ザイリンクス インプリメンテーション ツール (NGDBuild、MAP、PAR、BitGen) に入力して、ザイリンクス FPGA 用のコンフィギュレーション ビットストリームを生成できます。生成されるファイルの詳細は、「[コンパイル結果](#)」を参照してください。

[NGC Netlist] は [HDL Netlist] と似ていますが、HDL ファイルの代わりに NGC ファイルが生成されます。

[Hardware Co-Simulation] のいずれかを選択している場合は、選択したハードウェア FPGA プラットフォームで実行可能なコンフィギュレーション ビットストリームが生成されます。たとえば、[Hardware Co-Simulation] → [XtremeDSP Development Kit] → [PCI and USB] を選択すると、XtremeDSP ボード (ザイリンクスから別途購入可能) 用のビットストリームが生成されます。ハードウェア協調シミュレーション ブロックも生成され、ビットストリームに関連付けられます。このブロックは、Simulink シミュレーションにも使用されます。機能的には、このブロックが生成されたデザインの部分と同等ですが、ビットストリームでインプリメントされます。シミュレーションでこのブロックが生成する結果は、その部分が生成するシミュレーション結果と同じになりますが、結果は動作中のハードウェアから算出されます。

メモ : [Compilation] のオプションのリストをカスタマイズできます。詳細は、「[Linux OS への System Generator のインストール](#)」を参照してください。

その他のコンパイル パラメータを、次の表に示します。一部のパラメータは、[Compilation] に [HDL Netlist] を選択した場合にのみ設定可能です。たとえば、クロック ピンのロケーションは各ハードウェア FPGA プラットフォームで固定されているので、[Hardware Co-Simulation] の 1 つを選択している場合は [Clock pin location] は設定できません。

オプション	説明
Part	使用する FPGA デバイスを指定します。
Target directory	コンパイル結果を保存するディレクトリを指定します。 System Generator および FPGA インプリメンテーション ツール では多数のファイルが生成されるので、個別のディレクトリ (Simulink モデルファイルが含まれるディレクトリとは別のディレクトリ) を作成することをお勧めします。ディレクトリは、絶対パス (c:\netlist など) またはモデルを含むディレクトリを基準とした相対パス (netlist など) で指定できます。
Synthesis tool	デザインの合成に使用するツールを指定します。[XST]、[Synplify]、または [Synplify Pro] を選択できます。
Hardware description language	デザインの HDL ネットリストで使用する言語を指定します。[VHDL] または [Verilog] を選択できます。
Create testbench	HDL テストベンチを作成するよう指定します。HDL シミュレータでテストベンチをシミュレーションし、コンパイルされたデザインのシミュレーション結果を Simulink シミュレーション結果と比較します。 System Generator では、デザインを Simulink でシミュレーションし、 Gateway ブロックで検出される値を保存することにより、テスト ベクタを作成します。テストベンチの最上位 HDL ファイルの名前は、<name>_testbench.vhd/.v となります (<name> はテストするデザインの部分から導出された名前、拡張子はハードウェア記述言語により異なる)。
Import as configurable subsystem	コンパイル結果を関連付けるブロックを作成し、ブロックとこのブロックが作成された元のサブシステムを含むコンフィギャブル サブシステムを作成します。詳細は、「 コンフィギャブル サブシステムと System Generator 」を参照してください。
FPGA clock period	システム クロックの周期を ns で指定します。値は整数である必要はありません。ここで指定した周期は、制約ファイルでグローバル PERIOD 制約として設定され、ザイリンクス インプリメンテーション ツールに渡されます。複数サイクルパスは、この値の整数倍で制約されます。
Clock pin location	ハードウェア クロックのピン ロケーションを指定します。この情報は、制約ファイルを介してザイリンクス インプリメンテーション ツールに渡されます。

オプション	説明
Multirate implementation	<p>[Clock Enables] (デフォルト): クロック イネーブル ジェネレータ回路を作成してマルチレート デザインを駆動します。</p> <p>[Hybrid DCM-CE]: DCM を使用したクロック ラップを作成します。DCM は、異なるレートのクロック ポートを Virtex-4 および Virtex-5 では 3 つまで、Spartan-3A DSP では 2 つまで駆動できます。レートは、CLK0 > CLK2x > CLKdv > CLKfx という 順で DCM 出力ポートにマップされます。クロック レートの高いものが DCM でサポートされます。デザインに DCM でサポート可能な数以上のクロック ポートが含まれる場合は、残りのクロックはクロック イネーブル コンフィギュレーションでサポートされます。</p> <p>リセット入力ポートは DCM クロック ラップに含まれるので DCM を外部からリセットできるようになっています。また、ロック出力もラップに含まれており、外部デザインで入力データを 1 つの clk 入力ポートに同期させるのに役立ちます。</p> <p>[Expose Clock Ports]: 複数のクロック ポートを System Generator デザインの最上位に含め、デザインの外部から複数の同期クロック入力を供給できるようにします。</p>
DCM input clock period (ns)	[FPGA clock period (ns)] オプション (システム クロック) と異なる場合に指定します。FPGA クロック周期 (システム クロック) はこのハードウェア定義入力から生成されます。
Provide clock enable clear pin	<p>最上位クロック ラップに ce_clr ポートを含めるよう指定します。</p> <p>ce_clr 信号は、クロック イネーブル生成ロジックをリセットするのに使用されます。クロック イネーブル生成ロジックをリセットできるようにすると、データ パスのサンプリングの開始を動的に指定できます。詳細は、「自動生成されたクロック イネーブル ロジックのリセット」を参照してください。</p>

[Simulink system period]

System Generator トークンのパラメータ ダイアログ ボックスで、**Simulink システム周期** ([Simulink system period]) を指定する必要があります。この値は、デザインのシミュレーションを実行する基準レートを秒単位で指定します。**Simulink システム周期**は、デザインで使用されるすべてのサンプリング周期の公約数にする必要があります。たとえば、サンプリング周期 2、6、8 を使用するブロックを含むデザインでは、使用可能な最大の **Simulink システム周期**は 2 ですが、1 および 0.5 も使用可能です。サンプリング周期は、明示的に指定するか、自動的に算出されるか、内部レート変換が行われるブロック内で導出されます。システム周期のハードウェア クロックへの影響については、「[タイミングとクロック](#)」を参照してください。

デザインをシミュレーションまたはコンパイルする前に、**System Generator** でシステム周期がデザインのすべてのサンプリング周期の公約数になっているかが検証されます。問題が検出された場合は、適切な値を推奨するダイアログ ボックスが表示されます。**[Update]** ボタンをクリックすると、推奨された値が使用されます。システム周期競合のサマリを表示するには、**[View Conflict Summary]** ボタンをクリックします。**[Update]** をクリックした場合は、シミュレーションまたはコンパイルを再実行する必要があります。

周期を調整できないために、**System Generator** モデルが矛盾したものになる可能性もあります。たとえば、システム レートで動作する必要のあるブロックで **Up Sample** ブロックを駆動すると、モデルが矛盾したものになります。システム周期をアップデートしても競合がレポートされる場合は、モデルに矛盾があるということであり、修正が必要です。

周期は階層で制御されます。詳細は、「[階層制御](#)」を参照してください。

[Block icon display]

モデルに表示するブロックのアイコンの表示を制御します。モデルをコンパイルした後（生成またはシミュレーションを実行、あるいは **Ctrl + D** を押す）、ブロックの情報がこのオプションでの選択に応じて表示されます。

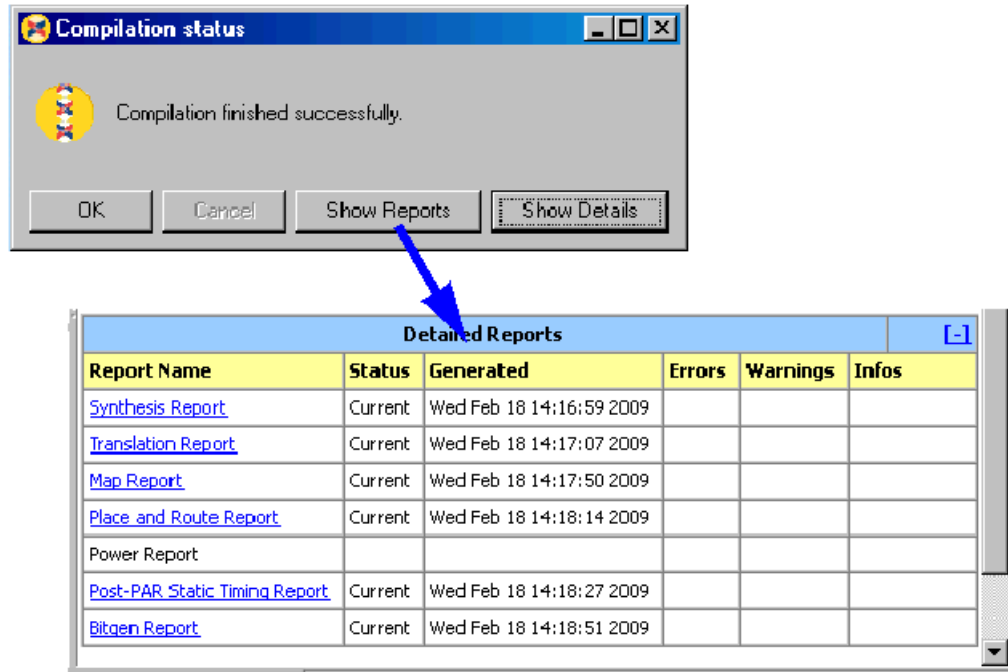
- [Default] : ポートの方向に関する基本的な情報が表示されます。
- [Sample rates] : 各ポートのサンプリング レートが表示されます。
- [Pipeline stages] : パイプライン段数が表示されます。
- [HDL port names] : ポート名が表示されます。
- [Input data types] : 各ポートの入力データ型が表示されます。
- [Output data types] : 各ポートの出力データ型が表示されます。

階層制御

System Generator トークンの [Simulink system period] オプション（「[\[Simulink system period\]](#)」を参照）は、階層で制御されます。設定は、**System Generator** トークンが適用される範囲にのみ適用され、下位にある **System Generator** トークンで変更されます。たとえば、デザインの最上位にある **System Generator** トークンで [Simulink system period] が設定されており、サブシステムに配置されている **System Generator** トークンでは別の値が設定されているとします。この場合、サブシステムでは 2 番目のシステム周期が使用されますが、デザインのその他の部分では最上位の **System Generator** トークンで設定されたシステム周期が使用されます。

ISE レポートの表示

コンパイルが終了すると、次に示す [Compilation status] ダイアログ ボックスが表示されます。コンパイル ターゲットを [Bitstream] または [Timing and Power Analysis] に設定した場合、[Show Reports] ボタンをクリックすると関連する ISE レポートが表示されます。



コンパイル結果

このセクションでは、System Generator トークンで [HDL Netlist] を選択して [Generate] ボタンをクリックしたときに生成される低レベル ファイルについて説明します。このコンパイル タイプでは、デザインのインプリメンテーションに使用される HDL、NGC、EDIF ファイルと、ダウンストリーム処理を簡略化する補助ファイルとして、Project Navigator にデザインを組み込むためのファイル、HDL シミュレータでシミュレーションするためのファイル、さまざまな合成ツールを使用して合成するためのファイルなどが生成されます。すべてのファイルは、System Generator トークンで指定したディレクトリに生成されます。テストベンチを作成するよう指定していない場合 ([Create testbench] をオフ)、生成されるファイルは次のとおりです。

ファイル名/タイプ	説明
<design>.vhd/.v	デザインのほとんどの HDL 記述が含まれます。
<design>_cw.vhd/.v	<design>_files.vhd/.v の HDL ラップ。クロックおよびクロック イネーブルを駆動します。
EDN および NGC ファイル	System Generator では、CORE Generator™ を使用してデザインの一部をインプリメントします。CORE Generator では、multiplier_virtex2_6_0_83438798287b830b.edn というような名前の EDIF ファイルが生成されます。その他の必要なファイルが、NGC ファイルとして生成される場合もあります。

ファイル名/タイプ	説明
globals	デザインを記述するキー/値のペアが含まれます。このファイルは Perl ハッシュ テーブルとして構成されており、Perl の eval 関数を使用することにより Perl スクリプト でキーと 値を使用できます。
<design>_cw.xcf (または .ncf)	タイミング制約およびポート ロケーション制約が含まれます。これらの制約は、ザイリンクスの合成ツール XST およびザイリンクス インプリメンテーション ツールで使用されます。XST 以外の合成ツールを指定した場合は、拡張子は .ncf となります。
<design>_cw.ise	ザイリンクスのプロジェクト管理ツール Project Navigator で HDL および EDIF ファイルを処理するためのプロジェクト ファイル。
hdlFiles	System Generator で生成される HDL ファイルのリストが含まれます。ファイルは、通常の HDL 依存順でリストされています。
synplify_<design>.prj または xst_<design>.pr	指定した合成ツールでデザインをコンパイルできるようにするファイル。
vcom.do	ModelSim でデザインのビヘイビア シミュレーションを実行するために HDL をコンパイルするのに使用できるスクリプト ファイル。

テストベンチを作成するよう指定している場合は ([Create testbench] をオン)、上記のファイルに加え、シミュレーション結果を比較するためのファイルが生成されます。Simulink シミュレーション結果と ModelSim のシミュレーション結果が比較されます。生成される追加ファイルは、次のとおりです。

ファイル名/タイプ	説明
DAT ファイル	Simulink でのシミュレーション結果が含まれます。
<design>_tb.vhd/.v	デザインをラップするテストベンチ。 ModelSim でシミュレーションを実行すると、このテストベンチにより Simulink のシミュレーション結果と ModelSim のシミュレーション結果が比較されます。
vsim.do	ModelSim でテストベンチ シミュレーションを実行するために使用するスクリプト。
pn_behavioral.do、 pn_postmap.do、 pn_postpar.do、 pn_posttranslate.do	Project Navigator から ModelSim シミュレーションを開始できるようにするファイル。

Sysmte Generator 制約ファイルの使用

デザインをコンパイルすると、ダウンストリーム ツールでのデザインの処理方法を指示する制約ファイルが生成されます。この制約ファイルにより、高質のインプリメンテーションが短時間で生成されます。制約は、次の情報を供給します。

- システム クロックの周期
- システム クロックに対するデザインのさまざまな部分のスピード要件
- ポートを配置する必要があるピン ロケーション
- ポートの動作スピード

ファイルのフォーマットは、**System Generator** トークンで指定した合成ツールによって異なります。**XST** を選択した場合は **XCF** フォーマットで記述され、**Synplify** または **Synplify Pro** を選択した場合は **NCF** フォーマットで記述されます。フォーマットに応じて、ファイルの拡張子は **.xcf** または **.ncf** になります。

システム クロック周期

システム クロック 周期 (デザインで最速のハードウェア クロックの周期) は、**System Generator** トークンで指定できます。**System Generator** で指定した周期は、制約ファイルに記述されます。ダウンストリーム ツールでデザインがインプリメントされる際、この周期が目標として使用されます。

複数サイクル パス制約

多くのデザインは、異なるクロック レートで動作する複数の部分で構成されています。最速の部分ではシステム クロックが使用され、その他の部分のクロック周期は、システム クロック周期の整数倍になります。ダウンストリーム ツールに、デザインの各部分で達成する必要があるスピードを伝達する必要があります。この情報により、ツールの効率が大幅に向上し、短いコンパイル時間で高質のハードウェアを実現できます。デザインの分割、それぞれの部分のスピードは、制約ファイルで複数サイクル パス制約を使用して指定されます。

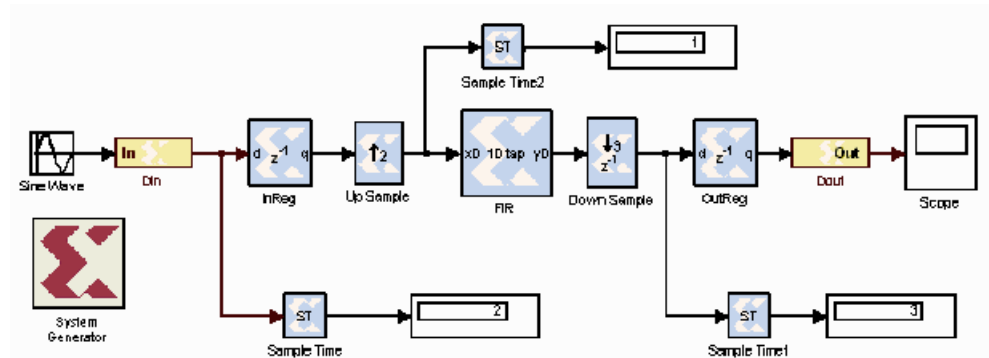
IOB タイミング制約と配置制約

System Generator の **Gateway In** と **Gateway Out** ブロックは、ハードウェアに変換されると入力ポートと出力ポートになります。これらのポートの位置とスピードは、**Gateway In** および **Gateway Out** ブロックのパラメータ ダイアログ ボックスで入力します。

詳細は、「**Gateway In**」および「**Gateway Out**」を参照してください。ポートのロケーションとスピードは、制約ファイルで **IOB** タイミングごとに指定されます。

制約の例

次の図に、小型のマルチレート デザインと、このデザインに対して System Generator で生成される制約を示します。



Up Sample ブロックでレートを 2 倍にし、Down Sample でレートを 1/3 にしています。システム クロック周期は 10ns であるとして、FIR ブロックのクロック周期は 10ns、入力レジスタ InReg ブロックのクロック周期は 20ns、出力レジスタ OutReg ブロックのクロック周期は 30ns です。次に、この情報を表す制約を説明します。

システム クロック周期が 10ns であることを示す行は、次のとおりです。

```
# Global period constraint
NET "clk" TNM_NET = "clk_392b7670";
TIMESPEC "TS_clk_392b7670" = PERIOD "clk_392b7670" 10.0 ns HIGH 50 %;
```

タイミング制約を作成するため、デザインのブロックをタイミング グループに分割します。2 つのブロックは、同じサンプリング レートで動作する場合のみ、同じタイミング グループに含めます。このデザインには、3 つのレートに対応した 3 つのタイミング グループがあります。最速のグループには、名前はありません。残りの 2 つのタイミング グループは、周期が 20ns のものは ce_2_392b7670_group、30ns のものは ce_3_392b7670_group とします。

FIR はシステム レート (最速) で動作するので、前述のグローバル周期制約を使用して制約します。クロックを生成するのに使用されるロジックは、常にシステム レートで動作し、システム レートに制約されます。

ce_2_392b7670_group には、システム レートの 1/2 で動作する InReg ブロックと Up Sample ブロックが含まれ、ce2_sysgen というクロック イネーブル ネットで駆動されます。このグループを定義する制約は、次のとおりです。

```
# ce_2_392b7670_group and inner group constraint
Net "ce_2_sg_x0*" TNM_NET = "ce_2_392b7670_group";
TIMESPEC "TS_ce_2_392b7670_group_to_ce_2_392b7670_group" = FROM
"ce_2_392b7670_group" TO "ce_2_392b7670_group" 20.0 ns;
```

メモ：ネット名にワイルドカードを使用することにより、クロック イネーブル ロジックが複製されたときに生成されるコピーにも、制約が適用されるようにしています。クロック イネーブル ネットの最大ファンアウト数は、合成ツールで制御できます。

ce_3_392b7670_group は、システム レートの 1/3 で動作します。Down Sample ブロックと OutReg ブロックが含まれ、ce_2_392b7670_group と同様に定義されます。

```
# ce_3_392b7670_group and inner group constraint
Net "ce_3_sg_x0*" TNM_NET = "ce_3_392b7670_group";
TIMESPEC "TS_ce_3_392b7670_group_to_ce_3_392b7670_group" = FROM
"ce_3_392b7670_group" TO "ce_3_392b7670_group" 30.0 ns;
```

グループ間制約は、グループ同士の相対的なスピード関係を定義します。次の制約は、`ce_2_392b7670_group` と `ce_3_392b7670_group` のスピード関係を確立します。

```
# Group-to-group constraints
TIMESPEC "TS_ce_2_392b7670_group_to_ce_3_392b7670_group" = FROM
"ce_2_392b7670_group" TO "ce_3_392b7670_group" 20.0 ns;
TIMESPEC "TS_ce_3_392b7670_group_to_ce_2_392b7670_group" = FROM
"ce_3_392b7670_group" TO "ce_2_392b7670_group" 20.0 ns;
```

ポートのタイミング要件は、**Gateway** ブロックのパラメータ ダイアログ ボックスで設定できます。これらの要件は、次に示すようなポート制約に変換されます。この例では、3 ビットの `din` 入力を **Gateway** のサンプリング レート (周期 20ns) で動作するように制約されます。**FAST** 属性は、遅延を削減するハードウェアを使用してポートをインプリメントする必要があることを示します。遅延を削減すると、ノイズや消費電力が増加することがあります。

```
# Offset in constraints
NET "din(0)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(0)" FAST;
NET "din(1)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(1)" FAST;
NET "din(2)" OFFSET = IN : 20.0 : BEFORE "clk";
NET "din(2)" FAST;
```

Gateway ブロックで **[Specify IOB location constraints]** をオンにすると、ポート ロケーションを指定できます。ポート ロケーションは、**[IOB pad locations]** ボックスに文字列のセル配列として入力します。ロケーションは、パッケージによって異なります。この例では、FG680 パッケージで **Virtex-E 2000** を使用しています。`din` バスのロケーション制約は、**Gateway** ブロックのパラメータ ダイアログ ボックスでは `{'D35', 'B36', 'C35'}` と入力します。これは、XCF (または NCF) ファイルでは次のような制約に変換されます。

```
# Loc constraints
NET "din(2)" LOC = "D35";
NET "din(1)" LOC = "B36";
NET "din(0)" LOC = "C35";
Clock Handling in HDL
```

HDL でのクロックの処理

このセクションでは、**System Generator** で生成される HDL でハードウェアクロックがどのように処理されるかを説明します。`<design>` という名前のデザインがあり、この名前は HDL 識別子として有効であるとしてします。**System Generator** でデザインをコンパイルすると、複数の HDL エンティティまたはモジュールが記述され、最上位のものに `<design>` という名前が付けられ、`<design>.vhd/.v` というファイルに保存されます。

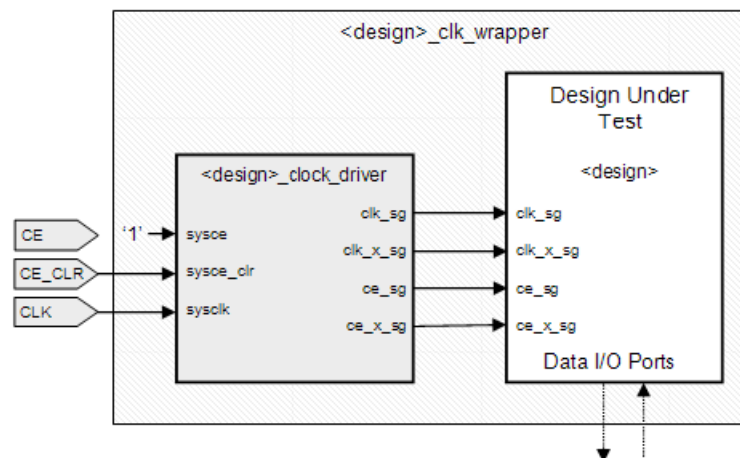
[Clock Enable] オプションを使用したマルチレート インプリメンテーション

クロックとクロック イネーブルは、ペアで HDL に配置されます。典型的なクロック名は `clk_1`、`clk_2`、`clk_3` で、そのペアであるクロック イネーブルの名前はそれぞれ `ce_1`、`ce_2`、および `ce_3` です。名前からクロック/クロック イネーブルのペアの動作レートがわかります。`clk_1` および `ce_1` で駆動されるロジックはシステム レート (最速) で動作し、`clk_2` および `ce_2` で駆動されるロジックはシステム レートの 1/2 で動作します。クロックおよびクロック イネーブルは、`<design>` エンティティまたはモジュール内では駆動されず、最上位入力ポートとなります。

このクロックとクロック イネーブルを生成するため、**System Generator** でクロック ラッパ (<design>_cw.vhd/.v) が記述されます。このラッパは、前述のファイルの外部となります。これは、HDL に柔軟性を持たせるのが目的です。アプリケーションによっては、前述のファイルが大型デザインに追加され、クロック ラッパが除外されることがあります。この場合、クロックとクロック イネーブルが生成されるようにする必要がありますが、詳細な制御が可能です。クロック ラッパがアプリケーションに適している場合は、含めます。

System Generator で生成された HDL に含まれるクロックとクロック イネーブルの名前から、クロック供給は完全に汎用であるように見えますが、そうではありません。たとえば、デザインに clk_1 と clk_2 というクロックが含まれており、対応するクロック イネーブルが ce_1 および ce_2 であるとしします。この場合、ハードウェアで ce_1 および ce_2 信号を High にし、clk_2 を clk_1 の 1/2 のレートであるクロック信号で駆動すればいいと思うかも知れませんが、ほとんどの **System Generator** デザインではこれではうまくいきません。その代わりに、clk_1 と clk_2 を同じクロックで駆動し、ce_1 を High に固定して、ce_2 を clk_1 と clk_2 の 1/2 のレートにします。

クロック ラッパには、デザインそのものと、クロックおよびクロック イネーブルを生成するクロック ドライバ コンポーネントの 2 つのコンポーネントで構成されます。クロック ドライバは、<design>_cw.vhd/.v というファイルに含まれます。<design>_cw に含まれるロジックは、ce_x 信号を生成します。**System Generator** トークンのパラメータダイアログボックスで [Provide clock enable clear pin] をオンにすると、オプションの ce_clr ポートも生成されます。クロックまたはクロック イネーブルでないポートは、クロック ラッパの外部に送られます。クロック ラッパを回路図で示すと、次のようになります。



メモ：クロック ラッパには ce というポートもありますが、このポートはラッパ上の clk ポートに付随しているだけで機能はありません。このポートは、**System Generator** デザインでクロック ラッパをブラック ボックスとして使用できるようにするために追加されています。

[Hybrid DCM-CE] オプションを使用したマルチレート インプリメンテーション

デジタル クロック マネージャ (DCM) を含む FPGA をターゲットとしている場合、クロック ツリーを DCM で駆動できます。DCM の使用は、クロック イネーブル ネットのファンアウトが大きくタイミング クロージャを達成するのが困難な場合に適しています。

System Generator では DCM が最上位 HDL クロック ラップ (接尾辞 `_dcm_mcw` が付いたファイル) にインスタンス化され、Virtex-4 および Virtex-5 では 3 つまでのレートが異なるクロック ポート、Spartan-3A DSP では 2 つまでのレートが異なるクロック ポートを使用できるよう DCM がコンフィギュレーションされます。デザインに DCM でサポートされる以上のクロック ポートがある場合は、残りのクロックは前のセクションで説明した CE (クロック イネーブル) コンフィギュレーションを使用して作成されます。

このオプションを使用して作成されたファイルの詳細は、「チュートリアル : [\[Hybrid DCM-CE\] オプションの使用](#)」を参照してください。

[Expose Clock Ports] オプションを使用したマルチレート インプリメンテーション

このオプションを選択すると、各レートのクロック ポートを含むラップが生成され、デザインの外部にクロック ジェネレータを手動でインスタンス化することによりこれらのクロック ポートを駆動できます。

このオプションを使用して作成されたファイルの詳細は、「チュートリアル : [\[Expose Clock Ports\] オプションの使用](#)」を参照してください。

コアのキャッシュ

System Generator では、CORE Generator で生成されたコアを使用してデザインの一部をインプリメントします。コアの生成には時間がかかることがあるので、System Generator ではあらかじめ生成されたコアをキャッシュします。CORE Generator を呼び出す前にキャッシュ内を検索して、コアが既に生成されている場合は、それを再利用します。

デフォルトでは、キャッシュのディレクトリは `$TEMP/sg_core_cache` で、2,000 個までのコアが保存されます。この制限に達すると、新しいコアを保存するためにキャッシュにあるコアが削除されます。

メモ：キャッシュの場所およびサイズを変更するには、環境変数を使用します。次に、これらの変数を示します。

環境変数	説明
SGCORECACHE	キャッシュ ファイルを保存する場所を指定します。この変数にブランク文字列を指定すると、コアはキャッシュされません。
SGCORECACHELIMIT	キャッシュするコアの最大数を指定します。

HDL テストベンチ

通常、System Generator デザインはビット 精度およびサイクル精度であり、Simulink でのシミュレーション結果はハードウェアでの動作と完全に一致します。場合によっては、Simulink シミュレーションの結果と HDL シミュレータでのシミュレーション結果を比較すると有益であることがあります。デザインにブラック ボックスが含まれている場合は、特に意味があります。System Generator トークンのパラメータ ダイアログ ボックスで **[Create testbench]** をオンにすると、これが可能です。

デザインの名前が `<design>` で、最上位に System Generator トークンが配置されているとします。このブロックのパラメータ ダイアログ ボックスでは、**[Compilation]** が **[HDL Netlist]** に設定されており、**[Create testbench]** がオンになっています。**[Generate]** をクリックすると、通常デザインに対して生成されるファイルに加え、次のファイルが生成されます。

1. テストベンチ HDL エンティティを含む <design>_tb.vhd/.v
2. HDL テストベンチ シミュレーションで使用するベスト ベクタを含む DAT ファイル
3. ModelSim でテストベンチをコンパイルおよびシミュレーションし、Simulink のテスト ベクタ と HDL で生成されたテスト ベクタを比較するために使用する vcom.do および vsim.do スクリプト ファイル

DAT ファイルには、Gateway ブロックを通過する値が保存されます。HDL シミュレーションでは、DAT ファイルからの入力値がスティミュラス、出力値が予測結果となります。テストベンチは、デザインの HDL にスティミュラスを供給し、HDL の結果と予測結果を比較するためのラップです。

MATLAB の FPGA へのコンパイル

System Generator では、MCode ブロックにより MATLAB が直接的にサポートされています。MCode ブロックは、入力値を M 関数に適用し、ザイリンクスの固定小数点データ型を評価します。評価は、サンプリング周期ごとに行われます。MCode ブロックでは、持続型変数を使用することにより、内部ステートを保持できます。入力ポートは指定の M 関数の入力引数、出力ポートは M 関数の出力引数により決定されます。MCode ブロックは、有限ステート マシン、制御ロジック、演算処理の多いシステムを構築するのに便利です。

MCode ブロックを使用するには、M 関数を記述する必要があります。M ファイルは、M ファイルを使用するモデルのディレクトリまたは MATLAB パスに配置します。

このセクションでは、MCode ブロックを使用する例を示します。

- 例 1: **単純なセレクト** - 入力の最大値を返すファンクションをインプリメントする方法を示します。
- 例 2: **単純な数値演算** - 単純な演算処理をインプリメントする方法を示します。
- 例 3: **レイテンシのある複素乗算器** - レイテンシを持つ複素乗算器を構築する方法を示します。
- 例 4: **シフト操作** - シフト操作をインプリメントする方法を示します。
- 例 5: **MCode ブロックへパラメータを渡す** - MCode ブロックにパラメータを渡す方法を示します。
- 例 6: **オプションの入力ポート** - MCode ブロックにオプションの入力ポートをインプリメントする方法を示します。
- 例 7: **有限ステートマシン** - 有限ステート マシンのインプリメント方法を示します。
- 例 8: **パラメータ指定アキュムレータ** - パラメータ指定アキュムレータを構築する方法を示します。
- 例 9: **FIR とシステム検証** - FIR ブロックをモデリングし、システム検証を実行する方法を示します。
- 例 10: **RPN カリキュレータ** - RPN カリキュレータ (スタック マシン) をモデリングする方法を示します。
- 例 11: **disp 関数** - disp 関数を使用して変数値をプリントする方法を示します。

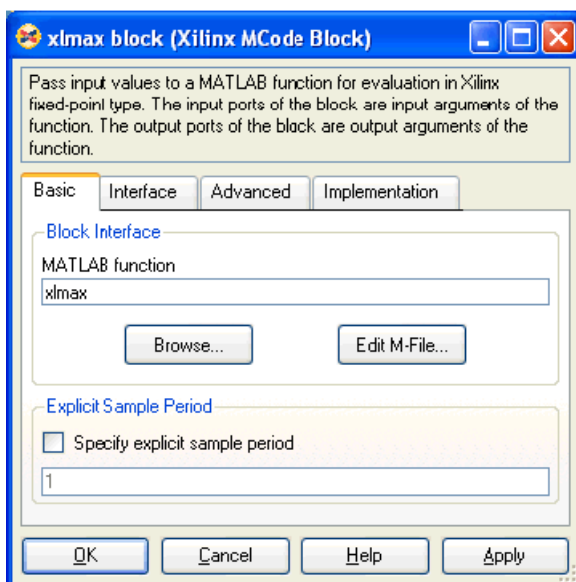
例 1 と 2 は、System Generator のインストールディレクトリの examples\mcode_block ディレクトリにある mcode_block_tutorial.mdl ファイルに、例 3 と 4 は mcode_block_tutorial2.mdl ファイルに、例 5 と 6 は mcode_block_tutorial3.mdl ファイルに、例 7 と 8 は mcode_block_tutorial4.mdl ファイルに、例 9 は mcode_block_verify_fir.mdl ファイルに、例 10 は mcode_block_rpn_calculator.mdl に含まれています。

単純なセレクト

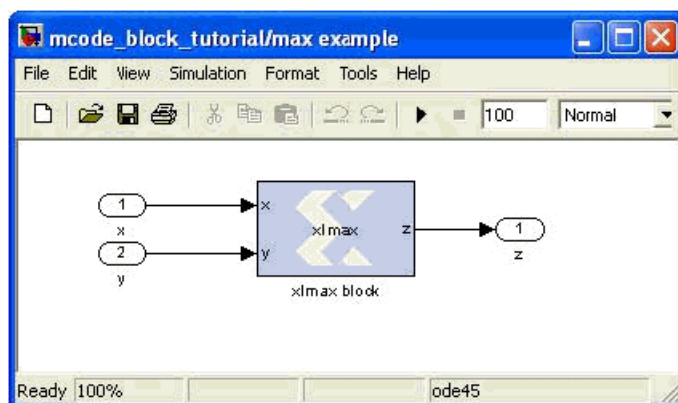
この例では、2 つの入力の最大値を出力に割り当てる、単純なデータ パス コントローラを示します。M 関数は次のように定義され、xlmax.m という M ファイルに保存されます。

```
function z = xlmax(x, y)
    if x > y
        z = x;
    else
        z = y;
    end
```

xlmax.m ファイルは、モデル ファイルと同じディレクトリまたは MATLAB パスに保存する必要があります。xlmax.m を適切な場所に保存したら、MCode ブロックをモデルにドラッグしてパラメータ ダイアログ ボックスを開き、[MATLAB Function] に「xlmax」と入力します。[OK] をクリックすると、ブロックに入力ポート x と y、出力ポート z が表示されます。



次の図に、モデルをコンパイルした後のブロックを示します。ブロックで計算が実行され、出力ポートに必要な固定小数点データ型が設定されます。



単純な数値演算

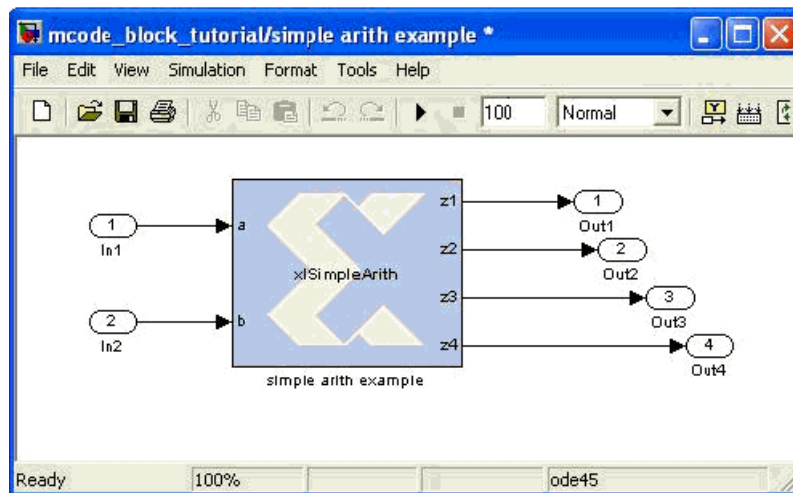
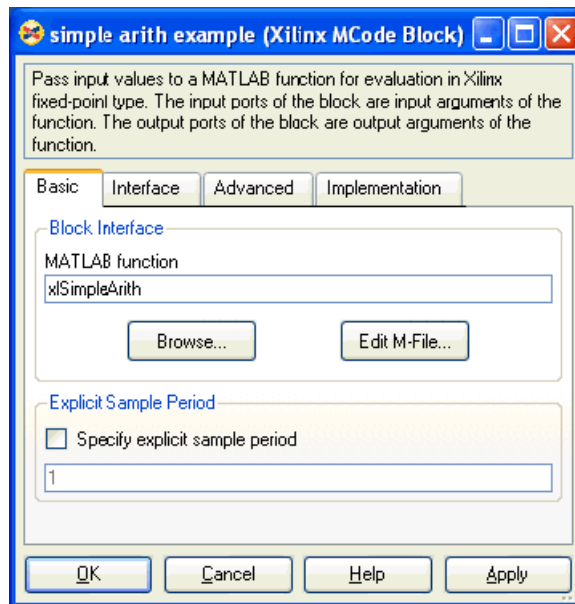
この例では、単純な数値演算と型変換を示します。xlSimpleArith という M 関数を定義する xlSimpleArith.m ファイルの内容は、次のとおりです。

```
function [z1, z2, z3, z4] = xlSimpleArith(a, b)
% xlSimpleArith demonstrates some of the arithmetic operations
% supported by the Xilinx MCode block. The function uses xfix()
% to create Xilinx fixed-point numbers with appropriate
% container types.%
% You must use a xfix() to specify type, number of bits, and
% binary point position to convert floating point values to
% Xilinx fixed-point constants or variables.
% By default, the xfix call uses xlTruncate
% and xlWrap for quantization and overflow modes.
% const1 is Ufix_8_3
const1 = xfix({xlUnsigned, 8, 3}, 1.53);
% const2 is Fix_10_4
const2 = xfix({xlSigned, 10, 4, xlRound, xlWrap}, 5.687);
z1 = a + const1;
z2 = -b - const2;
z3 = z1 - z2;
% convert z3 to Fix_12_8 with saturation for overflow
z3 = xfix({xlSigned, 12, 8, xlTruncate, xlSaturate}, z3);
% z4 is true if both inputs are positive
z4 = a>const1 & b>-1;
```

この M 関数では、加算および減算演算子を使用しています。MCode ブロックは、これらの演算子を完全精度で計算します。つまり、出力の精度は、これらの演算を情報を失わずに実行するのに十分であるということです。

ここで、xfix という関数の呼び出しに注目してください。この関数では、固定小数点データ型の精度と値の 2 つの引数が必要です。精度は、セル配列で指定します。精度のセル配列の最初の要素はデータ型で、xlUnsigned、xlSigned、または xlBoolean を指定できます。2 番目の要素は固定小数点値のビット数、3 番目の要素は 2 進小数点の位置を指定します。データ型が xlBoolean である場合は、ビット数および 2 進小数点の位置を指定する必要はありません。ビット数と 2 進小数点の位置は、対にして指定する必要があります。4 番目の要素は量子化モード、5 番目の要素はオーバーフローモードを指定します。量子化モードは、xlTruncate、xlRound、または xlRoundBanker のいずれかに指定します。オーバーフローモードは、xlWrap、xlSaturate、または xlThrowOverflow のいずれかに指定します。量子化モードとオーバーフローモードは、対にして指定する必要があります。量子化モードとオーバーフローモードを指定しない場合は、符号付きおよび符号なしの値に対して xlTruncate と xlWrap が使用されます。xfix 関数の 2 番目の引数は、倍精度またはザイリンクス固定小数点値で指定します。定数が整数値である場合は、xfix 関数を使用する必要はありません。MCode ブロックで、自動的に適切な固定小数点値に変換されます。

MCode ブロックのパラメータ ダイアログ ボックスで [MATLAB Function] に「xlSimpleArith」を入力した場合は、ブロックに 2 つの入力ポート a および b と、4 つの出力ポート z1、z2、z3、および z4 が表示されます。



ザイリンクスのデータ型および関数を使用する M 関数は、MATLAB の [Command Window] でテストできます。たとえば、MATLAB の [Command Window] に「[z1, z2, z3, z4] = xlSimpleArith(2, 3)」と入力すると、次の行が表示されます。

```
UFix(9, 3): 3.500000
Fix(12, 4): -8.687500
Fix(12, 8): 7.996094
Bool: true
```

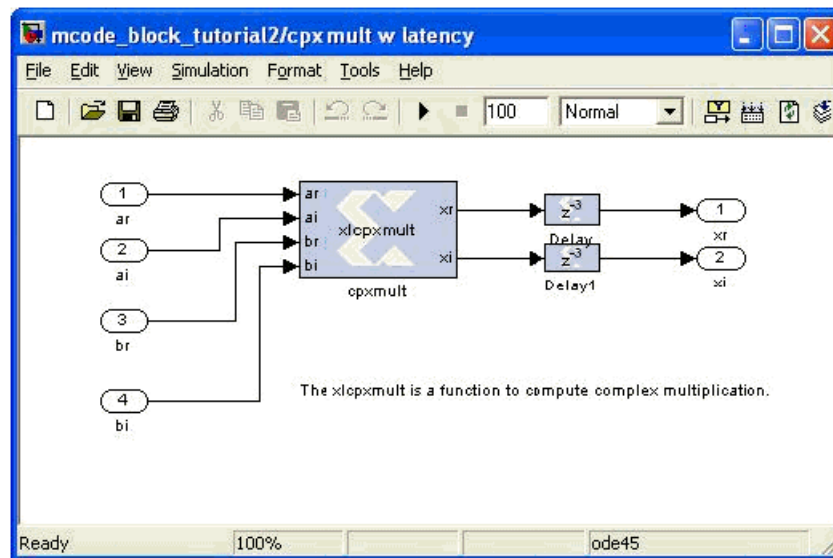
2 つの引数 2 および 3 は、自動的に固定小数点値に変換されています。引数として浮動小数点値を使用する場合は、xfix 関数呼び出しが必要です。

レイテンシのある複素乗算器

この例では、複素乗算器の作成方法を示します。xlcpxmult 関数を定義する xlcpxmult.m ファイルの内容は、次のとおりです。

```
function [xr, xi] = xlcpxmult(ar, ai, br, bi)
    xr = ar * br - ai * bi;
    xi = ar * bi + ai * br;
```

次の図に、サブシステムを示します。



MCode ブロックの後に、2 つの Delay ブロックが追加されています。Delay ブロックのパラメータダイアログボックスの [Implementation] タブで [Implement using behavioral HDL] をオンにすると、ダウンストリームのロジック合成ツールで、高パフォーマンスを達成するために適切な最適化を実行できます。

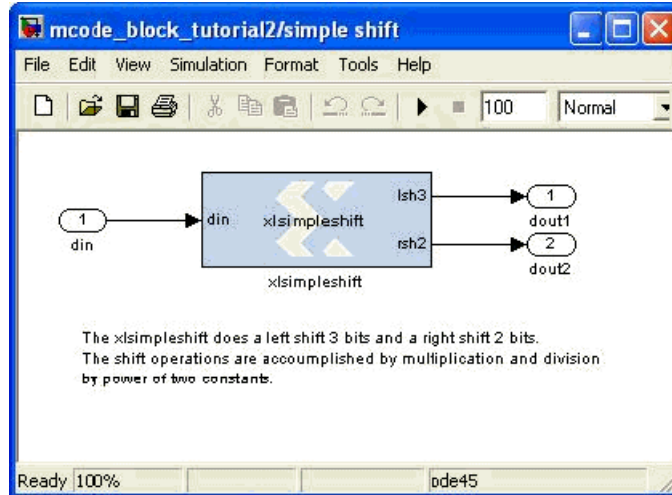
シフト操作

この例では、MCodeブロックを使用したビット シフト 操作のインプリメント 方法を示します。シフト 操作は、2 のべき乗での乗算および除算で達成されます。たとえば、4 で乗算することは 2 ビットのレフト シフトと同じであり、8 で割ることは 3 ビットのライト シフトと同じです。シフト 操作は、2 進小数点の位置の移動および必要に応じてビット 幅の拡張によりインプリメント されます。Fix_8_4 値を 4 で乗算すると Fix_8_2 値になり、Fix_8_4 を 64 で乗算すると Fix_10_0 値になります。

次に、1 レフト シフトと 1 ライト シフトを定義する xlsimpleshift.m ファイルの内容を示します。

```
function [lsh3, rsh2] = xlsimpleshift(din)
% [lsh3, rsh2] = xlsimpleshift(din) does a left shift
% 3 bits and a right shift 2 bits.
% The shift operation is accomplished by
% multiplication and division of power
% of two constant.
lsh3 = din * 8;
rsh2 = din / 4;
```

次の図に、コンパイル後のサブシステムを示します。



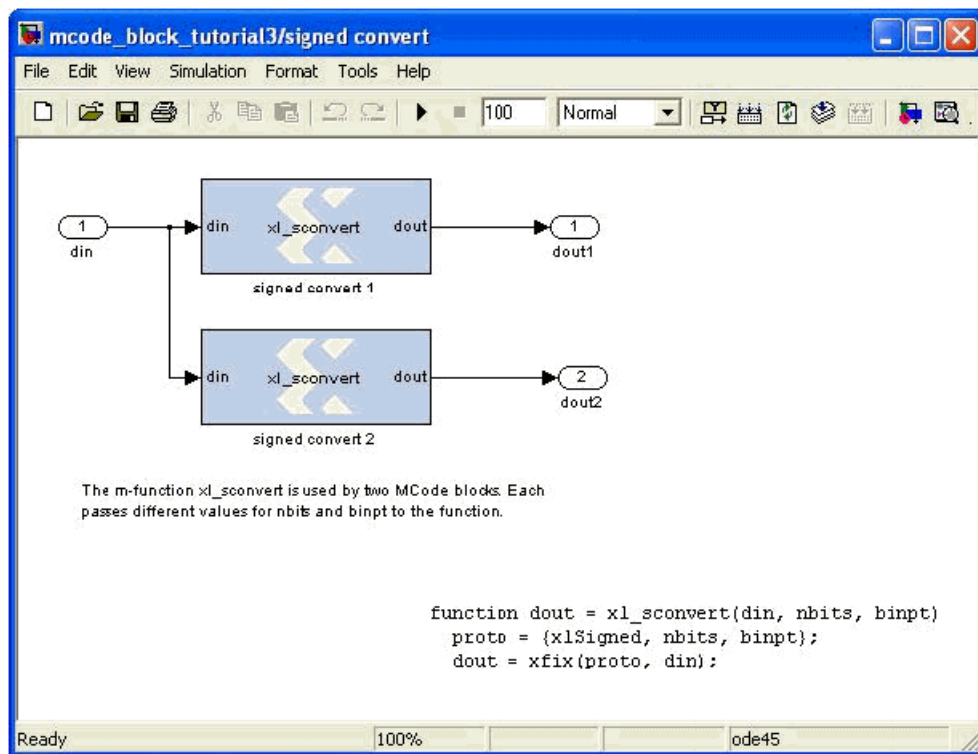
MCode ブロックへパラメータを渡す

この例では、MCode ブロックにパラメータを渡す方法を示します。M 関数への入力引数は、MCode ブロックの入力ポートとして、またはブロックの内部パラメータとして解釈されます。

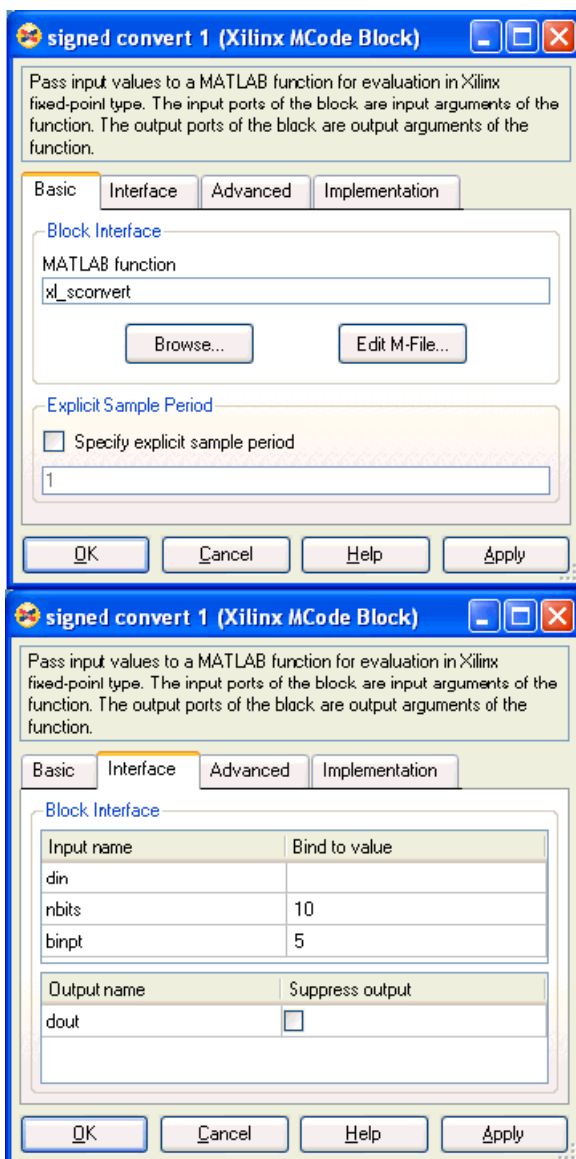
次の M コードは、xl_sconvert.m に含まれる M 関数 xl_sconvert を定義します。

```
function dout = xl_sconvert(din, nbits, binpt)
    proto = {xlSigned, nbits, binpt};
    dout = xfix(proto, din);
```

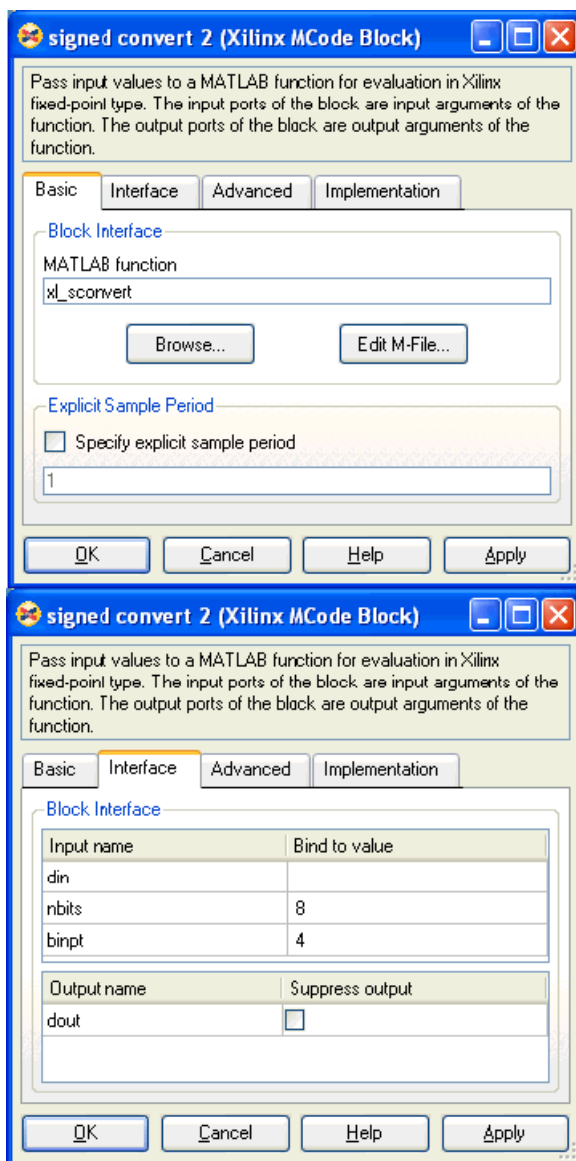
次の図に、M 関数 xl_sconvert を使用する 2 つの MCode ブロックを含むサブシステムを示します。M 関数の引数 nbits と binpt には、各 MCode ブロックに異なるパラメータを渡すことにより、異なる値が指定されます。signed convert 1 という MCode ブロックでは、渡されるパラメータにより入力データが Fix_16_8 から Fix_10_5 に変換されます。signed convert 2 という MCode ブロックでは、渡されるパラメータにより入力データが Fix_16_8 から Fix_8_4 に変換されます。



各 MCode ブロックにパラメータを渡すには、MCode ブロックのパラメータ ダイアログ ボックスで [Interface] タブをクリックし、M 関数の引数を設定します。次に、MCode ブロック signed convert 1 の設定を示します。



上図の [Interface] タブでは、M 関数の引数 `nbits` を 10 に、`binpt` を 5 に設定しています。次に、MCode ブロック `signed convert 2` の設定を示します。



上図の [Interface] タブでは、M 関数の引数 `nbits` を 8 に、`binpt` を 4 に設定しています。

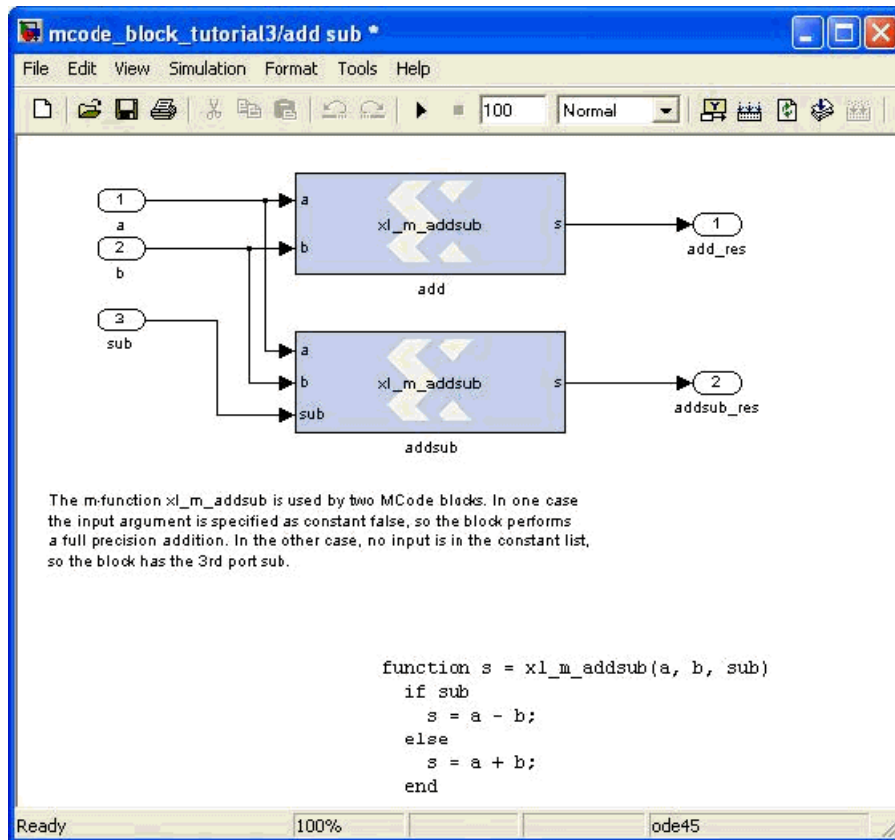
オプションの入力ポート

この例では、MCode ブロックにパラメータを渡す機能を使用して、MCode ブロックでオプションの入力ポートを使用するかどうかを指定する方法を示します。

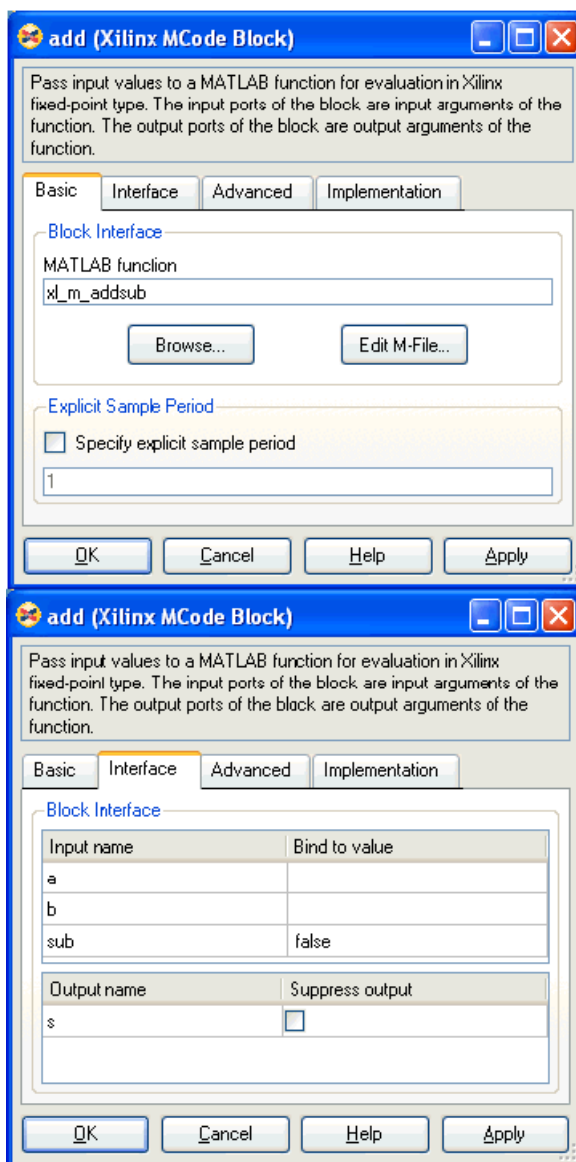
次の M コードは、xl_m_addsub.m に含まれる M 関数 xl_m_addsub を定義します。

```
function s = xl_m_addsub(a, b, sub)
    if sub
        s = a - b;
    else
        s = a + b;
    end
```

次の図に、M 関数 xl_m_addsub を使用する 2 つの MCode ブロックを含むサブシステムを示します。



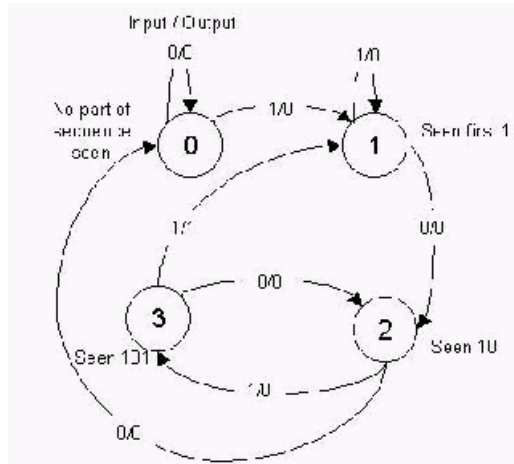
次に、add という名前の MCode ブロックのパラメータ ダイアログ ボックスを示します。



この設定により、入力ポート **a** と **b** が使用され、完全精度の加算が実行されます。addsub という MCode ブロックでは、入力パラメータ **sub** の [Bind to value] に何も指定されていないので、addsub ブロックでは入力ポート **a**、**b**、および **sub** が使用され、入力ポート **sub** の値に応じて完全精度の加算または減算が実行されます。

有限ステートマシン

この例では、MCode ブロックと内部ステート変数を使用して有限ステート マシンを作成する方法を示します。次の図に示すステート マシンは、入力データの 1011 というパターンを検出します。



MCode ブロックで使用する M 関数には、現在の入力のステートに基づいて次のステートを算出する遷移関数が含まれています。例 3 とは異なり、この例の M 関数では持続型ステート変数を定義して、MCode ブロックに有限ステート マシンのステートを保存します。次の M コードは、detect1011_w_state.m に含まれる M 関数 detect1011_w_state を定義します。

```
function matched = detect1011_w_state(din)
% This is the detect1011 function with states for detecting a
% pattern of 1011.

seen_none = 0; % initial state, if input is 1, switch to seen_1
seen_1 = 1;    % first 1 has been seen, if input is 0, switch
               % seen_10
seen_10 = 2;   % 10 has been detected, if input is 1, switch to
               % seen_101
seen_101 = 3; % now 101 is detected, if input is 1, 1011 is
               % detected and the FSM switches to seen_1

% the state is a 2-bit register
persistent state, state = xl_state(seen_none, {xlUnsigned, 2, 0});

% the default value of matched is false
matched = false;

switch state
case seen_none
    if din==1
        state = seen_1;
    else
        state = seen_none;
    end
case seen_1 % seen first 1
    if din==1
        state = seen_1;
    else
        state = seen_10;
    end
end
```

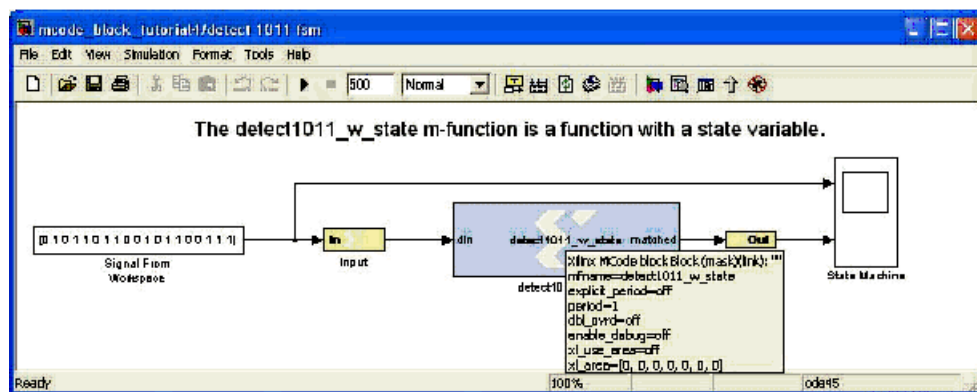


```

case seen_10 % seen 10
    if din==1
        state = seen_101;
    else
        % no part of sequence seen, go to seen_none
        state = seen_none;
    end
case seen_101
    if din==1
        state = seen_1;
        matched = true;
    else
        state = seen_10;
        matched = false;
    end
end
end

```

次の図に、M 関数 detect1011_w_state を使用する MCode ブロックを含むステート マシン サブシステムを示します。



パラメータ指定アキュムレータ

この例では、MCode ブロックを使用して、柔軟なインプリメンテーションを可能にするため持続性ステート変数とパラメータを定義したアキュムレータを作成する方法を示します。次の M コードは、xl_accum.m に含まれる M 関数 xl_accum を定義します。

```

function q = xl_accum(b, rst, load, en, nbits, ov, op,
    feed_back_down_scale)
% q = xl_accum(b, rst, nbits, ov, op, feed_back_down_scale) is
% equivalent to our Accumulator block.
    binpt = xl_binpt(b);
    init = 0;
    precision = {xSigned, nbits, binpt, xlTruncate, ov};
    persistent s, s = xl_state(init, precision);
    q = s;
    if rst
        if load
            % reset from the input port
            s = b;
        else
            % reset from zero
            s = init;
        end
    else

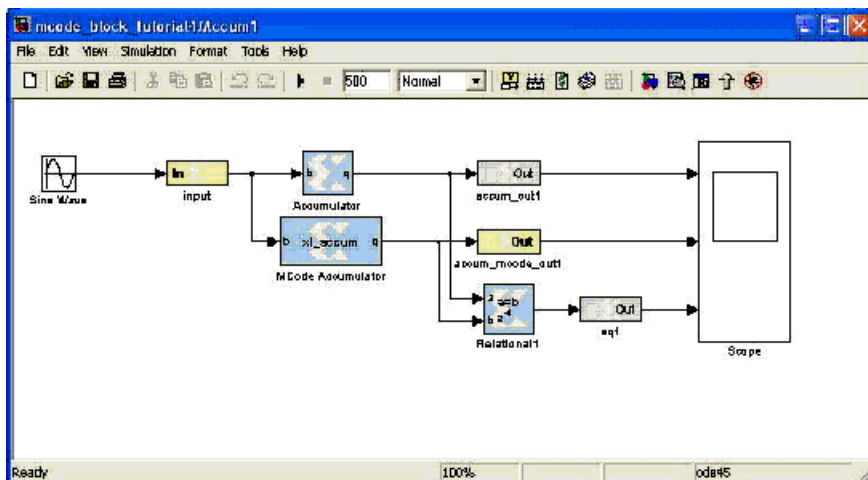
```

```

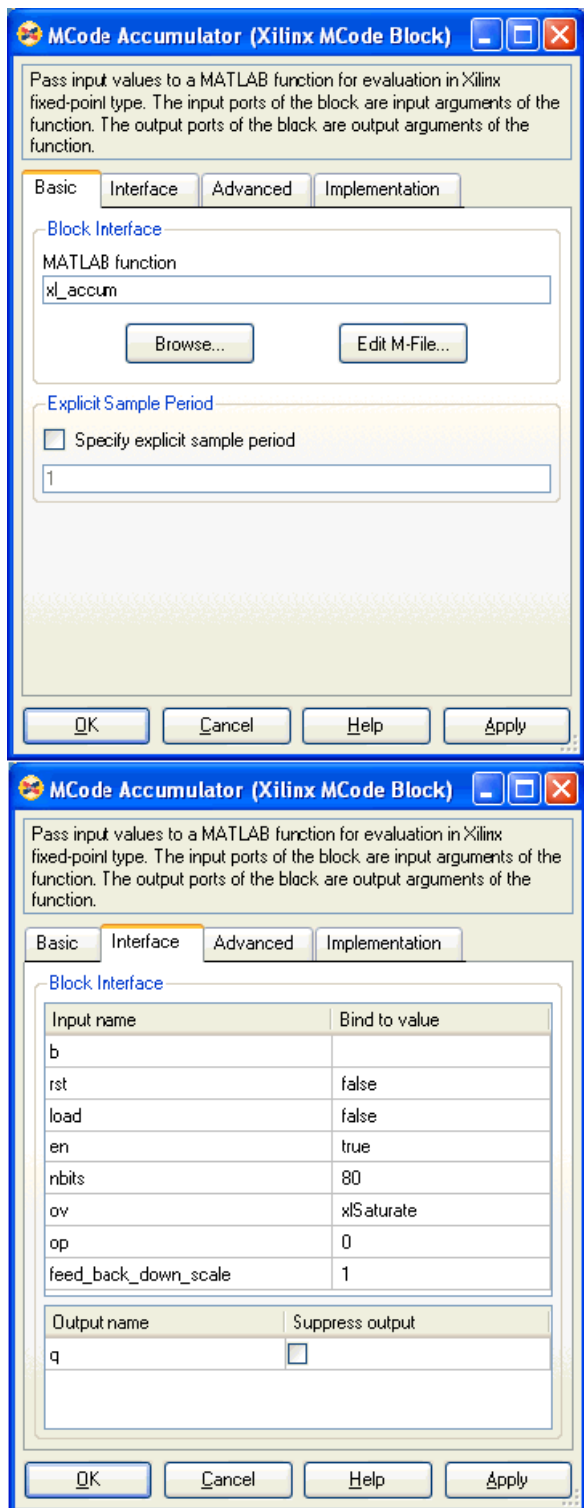
if ~en
else
    % if enabled, update the state
    if op==0
        s = s/feed_back_down_scale + b;
    else
        s = s/feed_back_down_scale - b;
    end
end
end
end

```

次の図に、M 関数 `xl_accum` を使用するアキュムレータ **MCode** ブロックを含むサブシステムを示します。**MCode** ブロックには、**MCode Accumulator** という名前が付いています。サブシステムには、比較のため、**Accumulator** という名前のザイリンクスアキュムレータブロックも含まれています。**MCode** ブロックは、ザイリンクス アキュムレータブロックと同じ機能を持ちますが、**MCode** ブロックのパラメータをマスク インターフェイス (パラメータ ダイアログ ボックスの [Interface] タブ) で指定している点が異なります。



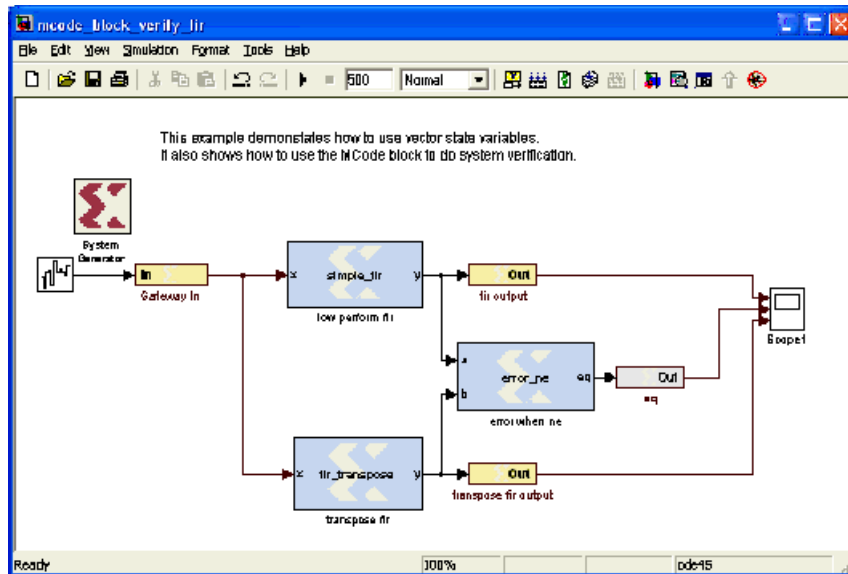
MCode Accumulator のオプションの入力 rst および load は、[Interface] タブでディスエーブルに設定されています。次に、MCode Accumulator ブロックのパラメータ ダイアログ ボックスを示します。



この例には、同じ M 関数を使用した MCode ブロックのアキュムレータ サブシステムがさらに 2 つ含まれていますが、異なるパラメータ設定を使用して、異なるアキュムレータを作成しています。

FIR とシステム検証

この例では、MCode ブロックを使用して FIR をモデリングする方法と、MCode ブロックでシステム検証を実行する方法を示します。



この例には、FIR ブロックが 2 つ含まれています。これらのブロックは MCode ブロックで定義されており、どちらも合成可能です。次のコードは、これら 2 つのブロックを定義する 2 つの関数です。

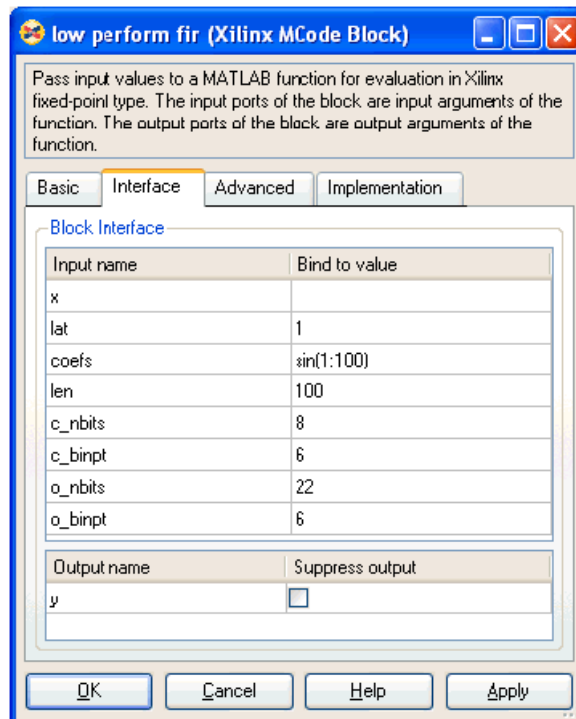
```
function y = simple_fir(x, lat, coefs, len, c_nbits, c_binpt, o_nbits,
o_binpt)
    coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbits, o_binpt};

    coefs_xfix = xfix(coef_prec, coefs);
    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent x_line, x_line = xl_state(zeros(1, len-1), x);
    persistent p, p = xl_state(zeros(1, lat), out_prec, lat);

    sum = x * coef_vec(0);
    for idx = 1:len-1
        sum = sum + x_line(idx-1) * coef_vec(idx);
        sum = xfix(out_prec, sum);
    end
    y = p.back;
    p.push_front_pop_back(sum);
    x_line.push_front_pop_back(x);
function y = fir_transpose(x, lat, coefs, len, c_nbits, c_binpt,
o_nbits, o_binpt)
    coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
    out_prec = {xlSigned, o_nbits, o_binpt};
    coefs_xfix = xfix(coef_prec, coefs);
    persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
    persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);
    if lat <= 0
        error('latency must be at least 1');
    end
    lat = lat - 1;
    persistent dly,
```

```
if lat <= 0
    y = reg_line.back;
else
    dly = xl_state(zeros(1, lat), out_prec, lat);
    y = dly.back;
    dly.push_front_pop_back(reg_line.back);
end
for idx = len-1:-1:1
    reg_line(idx) = reg_line(idx - 1) + coef_vec(len - idx - 1) * x;
end
reg_line(0) = coef_vec(len - 1) * x;
```

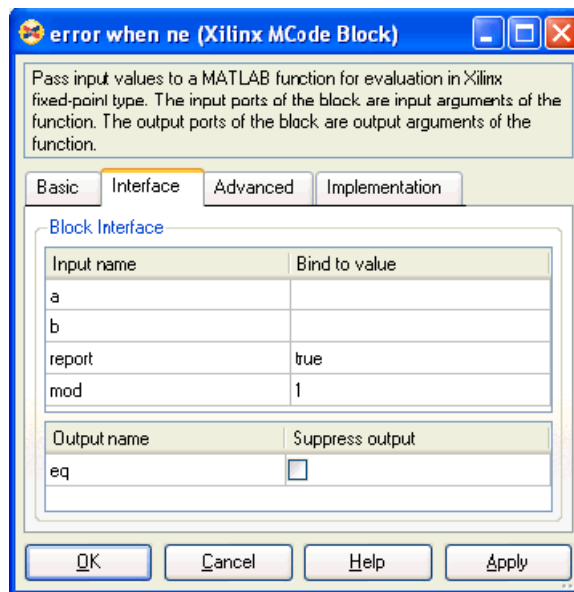
パラメータは、次のように設定されます。



2つのブロックの機能が同一であることを検証するため、MCode ブロックをもう 1 つ使用して 2 つのブロックの出力を比較します。2 つの出力が等しくない場合は、エラー チェック ブロックによりエラーがレポートされます。エラー チェックは、次の関数により行われます。

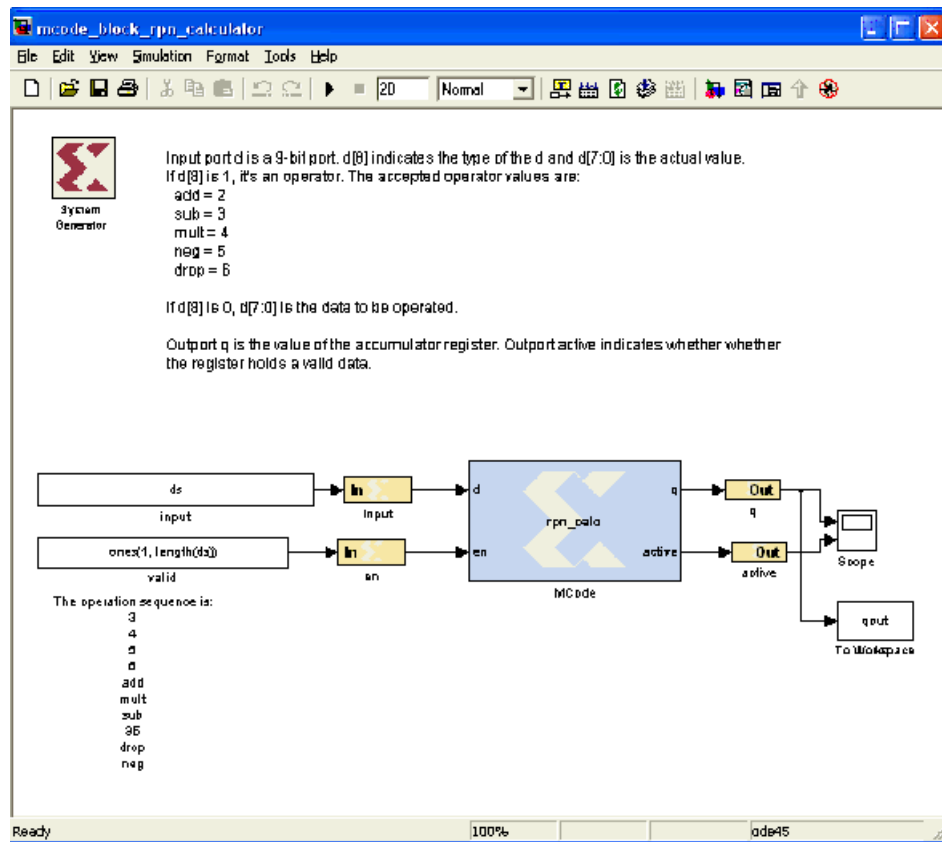
```
function eq = error_ne(a, b, report, mod)
persistent cnt, cnt = xl_state(0, {xlUnsigned, 16, 0});
switch mod
case 1
    eq = a==b;
case 2
    eq = isnan(a) || isnan(b) || a == b;
case 3
    eq = ~isnan(a) && ~isnan(b) && a == b;
otherwise
    eq = false;
    error(['wrong value of mode ', num2str(mod)]);
end
if report
    if ~eq
        error(['two inputs are not equal at time ', num2str(cnt)]);
    end
end
cnt = cnt + 1;
```

このブロックは、次のように設定されます。



RPN カリキュレータ

この例では、MCode ブロックを使用して、スタック マシンである RPN カリキュレータを構築する方法を示します。このブロックは、合成可能です。



次の関数は、RPN カリキュレータを定義します。

```
function [q, active] = rpn_calc(d, rst, en)
    d_nbits = xl_nbits(d);
    % the first bit indicates whether it's a data or operator
    is_oper = xl_slice(d, d_nbits-1, d_nbits-1)==1;
    din = xl_force(xl_slice(d, d_nbits-2, 0), xlSigned, 0);
    % the lower 3 bits are operator
    op = xl_slice(d, 2, 0);
    % acc the the A register
    persistent acc, acc = xl_state(0, din);
    % the stack is implemented with a RAM and
    % an up-down counter
    persistent mem, mem = xl_state(zeros(1, 64), din);
    persistent acc_active, acc_active = xl_state(false, {xlBoolean});
    persistent stack_active, stack_active = xl_state(false, ...
                                                    {xlBoolean});

    stack_pt_prec = {xlUnsigned, 5, 0};
    persistent stack_pt, stack_pt = xl_state(0, {xlUnsigned, 5, 0});
    % when en is true, it's action
    OP_ADD = 2;
    OP_SUB = 3;
    OP_MULT = 4;
    OP_NEG = 5;
```



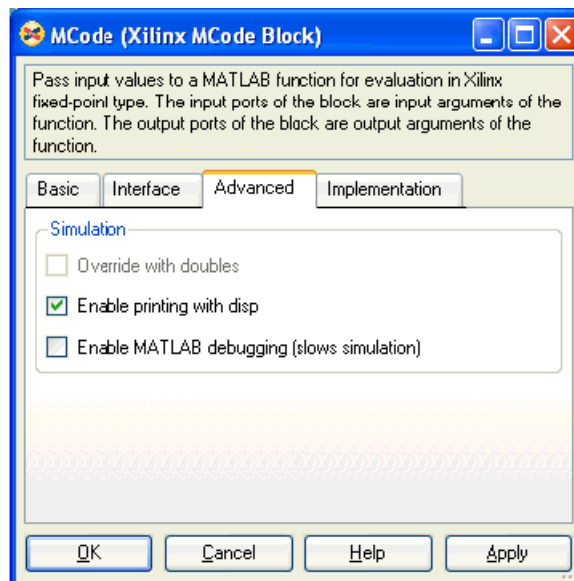
```
OP_DROP = 6;
q = acc;
active = acc_active;
if rst
    acc = 0;
    acc_active = false;
    stack_pt = 0;
elseif en
    if ~is_oper
        % enter data, push
        if acc_active
            stack_pt = xfix(stack_pt_prec, stack_pt + 1);
            mem(stack_pt) = acc;
            stack_active = true;
        else
            acc_active = true;
        end
        acc = din;
    else
        if op == OP_NEG
            % unary op, no stack op
            acc = -acc;
        elseif stack_active
            b = mem(stack_pt);
            switch double(op)
                case OP_ADD
                    acc = acc + b;
                case OP_SUB
                    acc = b - acc ;
                case OP_MULT
                    acc = acc * b;
                case OP_DROP
                    acc = b;
            end
            stack_pt = stack_pt - 1;
        elseif acc_active
            acc_active = false;
            acc = 0;
        end
    end
end
stack_active = stack_pt ~= 0;
```

disp 関数

次の MCode 関数は、disp 関数を使用して変数値を指定する方法を示します。

```
function x = testdisp(a, b)
persistent dly, dly = xl_state(zeros(1, 8), a);
persistent rom, rom = xl_state([3, 2, 1, 0], a);
disp('Hello World!');
disp(['num2str(dly) is ', num2str(dly)]);
disp('disp(dly) is ');
disp(dly);
disp('disp(rom) is ');
disp(rom);
a2 = dly.back;
dly.push_front_pop_back(a);
x = a + b;
disp(['a = ', num2str(a), ', ', ...
      'b = ', num2str(b), ', ', ...
      'x = ', num2str(x)]);
disp(num2str(true));
disp('disp(10) is');
disp(10);
disp('disp(-10) is');
disp(-10);
disp('disp(a) is ');
disp(a);
disp('disp(a == b)');
disp(a==b);
```

MCode ブロックのパラメータ ダイアログ ボックスの [Advanced] タブで、[Enable printing with disp] をオンにする必要があります。



次に、最初のシミュレーション ステップで MATLAB の [Command Window] に表示される行を示します。

```
mcode_block_disp/MCode (Simulink time: 0.000000, FPGA clock: 0)
Hello World!
num2str(dly) is [0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000]
disp(dly) is
    type: Fix_11_7,
    maxlen: 8,
    length: 8,
    0: binary 0000.0000000, double 0.000000,
    1: binary 0000.0000000, double 0.000000,
    2: binary 0000.0000000, double 0.000000,
    3: binary 0000.0000000, double 0.000000,
    4: binary 0000.0000000, double 0.000000,
    5: binary 0000.0000000, double 0.000000,
    6: binary 0000.0000000, double 0.000000,
    7: binary 0000.0000000, double 0.000000,
disp(rom) is
    type: Fix_11_7,
    maxlen: 4,
    length: 4,
    0: binary 0011.0000000, double 3.0,
    1: binary 0010.0000000, double 2.0,
    2: binary 0001.0000000, double 1.0,
    3: binary 0000.0000000, double 0.0,
a = 0.000000, b = 0.000000, x = 0.000000
1
disp(10) is
    type: UFix_4_0, binary: 1010, double: 10.0
disp(-10) is
    type: Fix_5_0, binary: 10110, double: -10.0
disp(a) is
    type: Fix_11_7, binary: 0000.0000000, double: 0.000000
disp(a == b)
    type: Bool, binary: 1, double: 1
```

System Generator デザインの大型システムへのインポート

System Generator デザインは、多くの場合サブシステムとして大型の HDL デザインに組み込まれます。このセクションでは、2 つの System Generator デザインを大型デザインに組み込む方法、System Generator で作成した VHDL をシステム全体のシミュレーション モデルに組み込む方法を示します。

System Generator 10.1 から、System Generator (Sysgen) と Project Navigator (ProjNav) の統合フローが導入されています。現時点では、次の操作を実行できます。

- System Generator デザインをサブレベルとして大型デザインに追加する。
- System Generator の制約を最上位デザインに統合し、関連付ける。
- Project Navigator から System Generator デザインの一部の操作を実行可能。

HDL ネットリストのコンパイル

System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] に [HDL Netlist] を選択すると、HDL とデザインのインプリメンテーションに必要な関連のファイル (NGC ファイル、EDIF ファイルなど) が生成されます。また、ダウンストリーム処理を簡略化する補助ファイルとして、Project Navigator にデザインを組み込むためのファイル、HDL シミュレータでシミュレーションするためのファイル、さまざまな合成ツールを使用して合成するためのファイルなどが生成されます。詳細は、「[System Generator のコンパイル タイプ](#)」を参照してください。

10.1 から、System Generator プロジェクトの情報は、選択したクロック供給オプションに応じて <design_name>_cw.sgp または <design_name>_mcw.sgp というファイルに含まれるようになりました。このセクションでは、複数の System Generator デザインをサブモジュールとして大型デザインに組み込む方法を説明します。

デザインの統合に関する規則

System Generator モデルを大型デザインに組み込む場合、次の 2 つの規則に従う必要があります。

規則 1 : Gateway ブロックまたは System Generator トークンで IOB/CLK ロケーション制約を指定しないでください。これらのブロックで IOB/CLK ロケーション制約を指定すると、NGDBuild で次のような警告メッセージが表示されます。

```
WARNING:NgdBuild:483 - Attribute "LOC" on "clk" is on the wrong type of
object. Please see the Constraints Guide for more information on this
attribute.
```

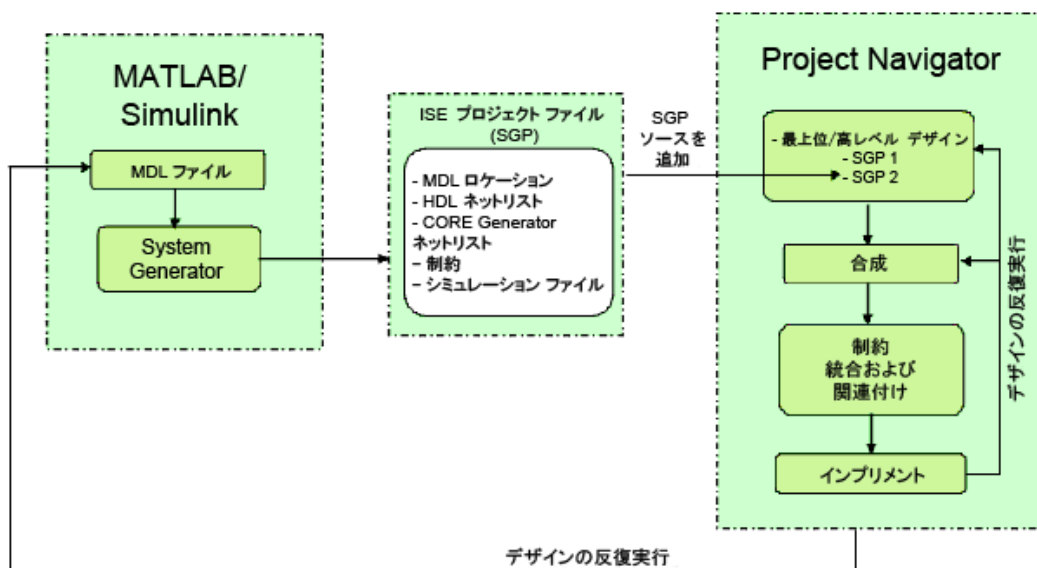
この場合、次の NGDBuild error を回避するため、IOB タイミング制約を none に設定する必要があります。

```
NgdBuild:756 -Could not find net(s) 'gateway_out(1)' in the design. To
suppress this error, specify the correct net name or remove the
constraint.
```

規則 2 : System Generator デザインから I/O ポートを最上位デザインに含める必要がある場合は、最上位 HDL コードに適切なバッファをインスタンスエートする必要があります。

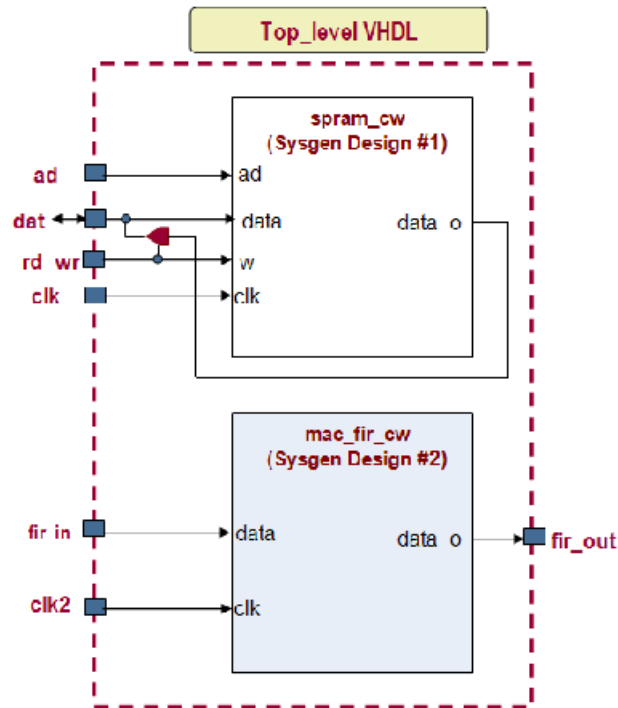
System Generator と Project Navigator の統合フロー

次の図は、複数の System Generator デザインを下位デザインとして Project Navigator に組み込むフローを示します。System Generator により拡張子が .sgp のプロジェクト ファイルが生成され、このプロジェクト ファイルを System Generator ソース タイプとして Project Navigator に追加します。このファイルには、ファイルの場所、制約ファイルなど、System Generator デザインに関する必要な情報がすべて含まれています。10.1 より 前の Project Navigator では、UCF ファイルの制約を手動で最上位デザインに統合し、関連付ける必要がありましたが、現在のリリースでは Project Navigator でインプリメンテーションを実行する際に自動的に実行されます。



手順の例

この例では、System Generator で生成した 2 つの HDL ネットリストを 1 つの大型 VHDL デザインに統合します。デザイン 1 は **SPRAM**、デザイン 2 は **MAC_FIR** という名前です。最上位 VHDL エンティティでは、**SPRAM** デザインの 2 つのデータポートと制御信号を使用して双方向バスを作成します。また、最上位 VHDL には **MAC_FIR** デザインをインスタンスエートし、別のクロック **clk2** を供給します。次に、このデザインのブロック図を示します。



この例のファイルは、<path_to_sysgen>\examples\projnav\mult_diff_designs に含まれています。次のファイルが含まれています。

- spram.mdl : System Generator デザイン 1
- mac_fir.mdl : System Generator デザイン 2

top_level というサブディレクトリに含まれるファイル：

- top_level.ise : top_level デザインをコンパイルするための Project Navigator プロジェクト
- top_level.vhd : 最上位 VHDL ファイル
- top_level_testbench.do : カスタム ModelSim DO ファイル
- top_level_testbench.vhd : 最上位 VHDL テストベンチ ファイル
- wave.do : 波形を表示するために top_level_testbench.do により呼び出される ModelSim DO ファイル

System Generator デザインの HDL ファイルの生成

HDL ファイルを生成するには、次の手順に従います。

1. MATLAB でデザイン 1 である `spram.mdl` を開きます。これはマルチレート デザインであり、Single Port RAM ブロックの出力の後に Down Sample ブロックが配置されています。
2. System Generator トークンをダブルクリックし、[Compilation] で [HDL Netlist] を選択して、[Generate] ボタンをクリックします。[Generate] ボタンをクリックすると、このデザインの HDL ファイルが `<path_to_sysgen>\examples\projnav\mult_diff_designs\hdl_netlist1` ディレクトリに作成されます。
3. `mac_fir.mdl` に対して、手順 1 ~ 2 を実行します。このデザインの HDL ファイルが、`<path_to_sysgen>\examples\projnav\mult_diff_designs\hdl_netlist2` ディレクトリに生成されます。

これで、System Generator により HDL ネットリストが生成されました。

HDL ライブラリ名の変更

複数の System Generator デザインを大型デザインに組み込む場合、シミュレーションの名前の競合やその他の不正な動作を回避するため、HDL ライブラリ名を変更する必要があります。System Generator には、System Generator デザインのすべてのファイルでライブラリ名を変更するユーティリティが含まれています。また、元のライブラリ名に戻す必要がある場合を考慮して、バックアップ コピーも作成されます。このユーティリティの構文は、次のとおりです。

構文：

```
xlSwitchLibrary(<target_dir_pathname>, <from_lib_name>, <to_lib_name>)
```

<target_dir_pathname> : デザインの場所

<from_lib_name> : 元の HDL ライブラリ名

<to_lib_name> : 新しい HDL ライブラリ名

1. MATLAB の [Command Window] に次のコマンドを入力します。

```
xlSwitchLibrary('hdl_netlist1','work','design1_lib')
```

2. 次のコマンドを入力します。

```
xlSwitchLibrary('hdl_netlist2','work','design2_lib')
```

次のようなメッセージが表示されます。

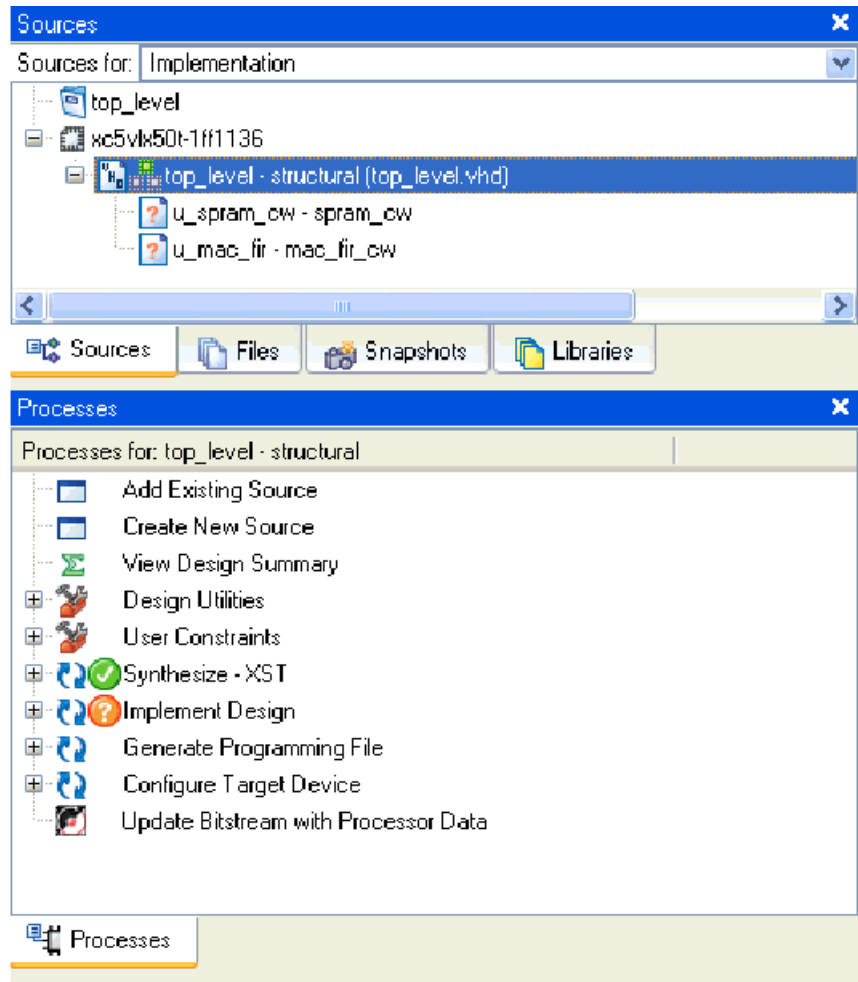
```
>> xlSwitchLibrary('hdl_netlist1','work','design1_lib')
INFO: Switching HDL library references in design spram.mdl ...
INFO: A backup of the original files can be found at 'C:\D_0_10.1\Examples\ProjNav_SysGen_Inst\hdl_netlist1\switch_lib_backup'
INFO: Processing file 'spram.vhd' ...
INFO: Processing file 'spram_cw.vhd' ...
INFO: Processing file 'xst_spram.prj' ...
INFO: Processing file 'vcom.dc' ...
INFO: Processing file 'vsim.dc' ...
INFO: Processing file 'fn_behav:occl.de' ...
INFO: Processing file 'fn_posttranslate.dc' ...
INFO: Processing file 'fn_postmap.dc' ...
INFO: Processing file 'fn_postpa.dc' ...
INFO: Processing file 'spram_cw.ice' ...
>> xlSwitchLibrary('hdl_netlist2','work','design2_lib')
INFO: Switching HDL library references in design mac_fir_cw ...
INFO: A backup of the original files can be found at 'C:\D_0_10.1\Examples\ProjNav_SysGen_Inst\hdl_netlist2\switch_lib_backup'
INFO: Processing file 'mac_fir.vhd' ...
INFO: Processing file 'mac_fir_cw.vhd' ...
INFO: Processing file 'xst_mac_fir.prj' ...
INFO: Processing file 'vcom.dc' ...
INFO: Processing file 'vsim.dc' ...
INFO: Processing file 'fn_behav:occl.de' ...
INFO: Processing file 'fn_posttranslate.dc' ...
INFO: Processing file 'fn_postmap.dc' ...
INFO: Processing file 'fn_postpa.dc' ...
INFO: Processing file 'mac_fir_cw.ice' ...
>>
```

System Generator ソースの最上位デザインへの追加

top_level デザインを合成するには、次の手順に従います。

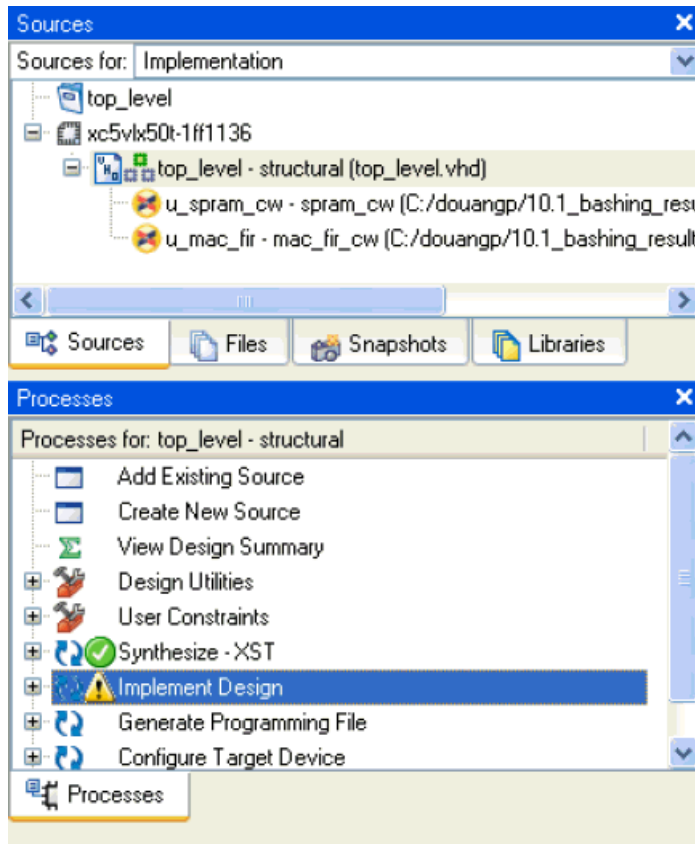
1. ISE を起動し、ISE プロジェクト <path_to_sysgen>\examples\projnav\
mult_diff_designs\top_level\top_level.ise のを開きます。

メモ：この時点では、この Project Navigator プロジェクトは次の図に示すようになっています。spram_cw および mac_fir_cw インスタンスは最上位デザインにインスタンス化されていますが、最上位デザインと同じディレクトリに含まれていないので、これらのインスタンス/モジュールが見つからないことを示すクエスチョン マーク (?) アイコンが表示されています。



2. [Hierarchy] ペインで u_spram_cw を右クリックして [Add Source] をクリックし、<path_to_sysgen>\examples\projnav\mult_diff_designs\hdl_netlist1\spram_cw.sgp ファイルを選択して [開く] をクリックします。
3. u_mac_fir に対して同じ操作を実行します。選択するファイルは <path_to_sysgen>\examples\projnav\mult_diff_designs\hdl_netlist2\mac_fir_cw.sgp です。

4. [Hierarchy] ペインで top_level を選択し、[Processes] ペインで [Implement Design] をダブルクリックします。インプリメンテーションが終了すると、Project Navigator の表示は次の図のようになります。



5. ワークスペースに表示されている [Design Summary] ウィンドウの左上のペインで [Detailed Reports] → [Place and Route Report] をクリックし、PAR レポート ファイルでタイミング制約を確認します。

PAR レポートを見ると、マルチレート制約が満たされていることが示されています。

	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
①	T5_clk_f488215c2 = PERIOD TIMEGRP "clk_f488215c2" 100 ns HIGH 50%	SETUP	96.339ns	3.666ns	0	0
②	T5_clk_c4b7e2441 = PERIOD TIMEGRP "clk_c4b7e2441" 100 ns HIGH 50%	HOLD	96.366ns	3.634ns	0	0
③	T5_ce_16_c4b7e244_group1 = MAXDELAY FROM TIMEGRP "ce_16_c4b7e244_group1" TO TIMEGRP "ce_16_c4b7e244_group1" 1600 ns	SETUP	1597.868ns	2.132ns	0	0
④	T5_ce_2_f488215c_group2 = MAXDELAY FROM TIMEGRP "ce_2_f488215c_group2" TO TIMEGRP "ce_2_f488215c_group2" 200 ns	N/A	N/A	N/A	N/A	N/A

各 System Generator の制約は、System Generator で作成され、UCF (ユーザー制約ファイル) に変換されています。これらの UCF 制約ファイルは、ISE インプリメンテーション (NGDBuild) の段階で統合され、関連付けられます。これらの制約は、次のとおりです。

両方のデザインの System Generator トークンで、システム サンプリング周期 100ns が設定されています。

- TS_clk_f488215c2 制約は、SRAM デザインのもので (1)。
- TS_clk_c4b7e2441 制約は、FIR デザインのもので (2)。
- TS_ce_16_c4b7e244_group_to_ce_16_cb47e244_group1 制約は、Down Sample ブロック以降の同期エレメントすべてに適用されます。システム サンプリング周期の 16 倍に設定されています (3)。
- SRAM デザインの Down Sample ブロックは、サンプリング レートを 1/2 にします。TS_ce_2_f488215c_group_to_ce_2_f488215c_group2 制約は、Down Sample ブロック以降の同期エレメントすべてに適用されます。システム サンプリング周期の 2 倍に設定されています (4)。

System Generator と Project Navigator の統合フローでは、Project Navigator によりこれらの制約が最上位デザインに統合され、関連付けられます。このフローは、10.1 以降でのみサポートされます。

デザイン全体のシミュレーション

top_level デザインのビヘイビア シミュレーションを実行するには、次の手順に従います。

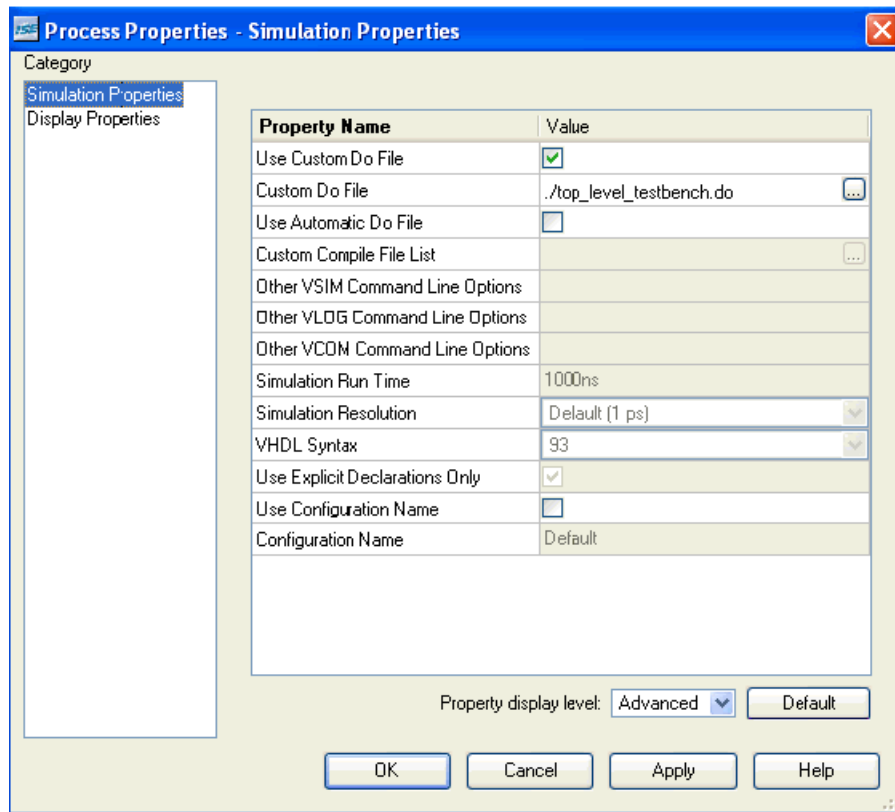
1. System Generator は、VHDL ファイルを作成し、選択されたロジック 合成ツールを起動して HDL ネットリスト を生成します。最上位デザインをシミュレーションするには、これらの VHDL ファイルを使用します。生成されるデザインの VHDL ファイルの名前は、<design>_cw.vhd または <design>.vhd です。top_level_testbench.do という ModelSim DO ファイルを開き、VHDL ファイルがどのように参照されているかを確認します。

シミュレーションに使用するメモリ初期化ファイル (MIF) と係数ファイル (COE) は、最上位 VHDL ファイルと同じディレクトリに配置する必要があります。この例では、ModelSim DO ファイル (top_level_testbench.do) の次の記述により、hdl_netlist1 および hdl_netlist2 サブフォルダから MIF ファイルがコピーされます。

```
foreach i [glob ../hdl_netlist1/*.mif] {  
  file copy -force $i .  
}
```

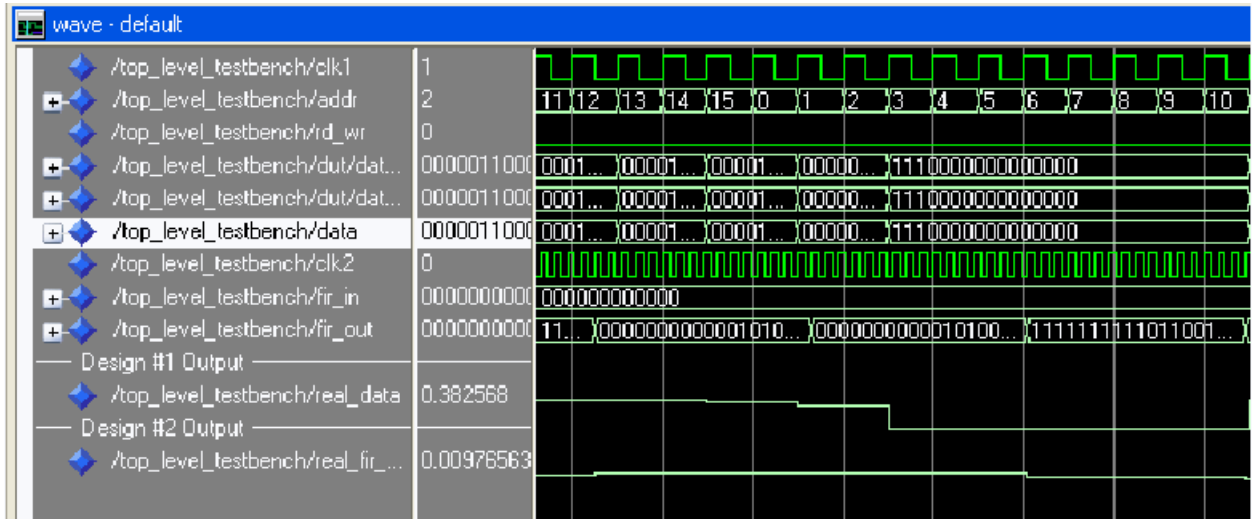
係数ファイルも存在する場合は、DO ファイルに同様の記述を追加すると、ファイルを最上位 VHDL ファイルのディレクトリにコピーできます。

2. Project Navigator で、[Hierarchy] ペインの [Sources for] で [Behavioral Simulation] を選択し、[top_level_testbench - structural (top_level.vhd)] を選択します。このファイルはプロジェクトにテストベンチ ファイルとしてインポートされており、シミュレータを使用してデザインをシミュレーションできるようにします。
3. [Processes] ペインで [Simulate Behavioral Model] を右クリックし、[Properties] をクリックします。[Process Properties] ダイアログ ボックスが開きます。次の図に示すように、[Custom Do File] が指定されています。



```
## NOTE: customer.do file
##
vlib design1_lib
vcom -explicit -93 -work design1_lib "../hdl_netlist1/spram.vhd"
vcom -explicit -93 -work design1_lib "../hdl_netlist1/spram_cw.vhd"
vlib design2_lib
vcom -explicit -93 -work design2_lib "../hdl_netlist2/mac_fir.vhd"
vcom -explicit -93 -work design2_lib "../hdl_netlist2/mac_fir_cw.vhd"
vlib work
vcom -explicit -93 top_level.vhd
vcom -explicit -93 top_level_testbench.vhd
foreach i [glob ../hdl_netlist1/*.mif] {
    file copy -force $i .
}
foreach i [glob ../hdl_netlist2/*.mif] {
    file copy -force $i .
}
vsim -t 1ps -lib work top_level_testbench
do wave.do
run 1000ns
```

上図は、System Generator で生成された VHDL コードをコンパイルする ModelSim コマンドを示します。top_level デザインをシミュレーションするには、[Simulate Behavioral Model] をダブルクリックします。ModelSim DO ファイルにより VHDL コードがコンパイルされ、シミュレーションが 1000ns 間実行されます。シミュレーション結果の波形を次に示します。



まとめ

このセクションでは、System Generator デザインを大型システムにインポートする方法を示しました。プロセスの各段階で、注意すべき事項がいくつかあります。

System Generator デザインの作成：

- Gateway In ブロックおよび System Generator トークンで IOB 制約およびクロック ピン ロケーションを指定しないでください。
- System Generator トークンのパラメータ ダイアログ ボックスの [Compilation] で [HDL Netlist] を選択します。System Generator で生成される HDL ネットリストには、デザインの RTL、EDIF、および制約に関する情報が含まれています。

最上位シミュレーション：

- System Generator で生成された VHDL ファイルをコンパイルするため、カスタム ModelSim DO ファイルを使用します。Project Navigator のプロパティを、カスタム DO ファイルを使用するよう変更します。

新機能：

- System Generator プロジェクト ファイル (SGP) をサブモジュールとして Project Navigator に追加可能
- System Generator 制約を自動的に最上位デザインに統合および関連付け
- MATLAB および System Generator MDL を Project Navigator から起動し、デザインの一部のプロセスを実行

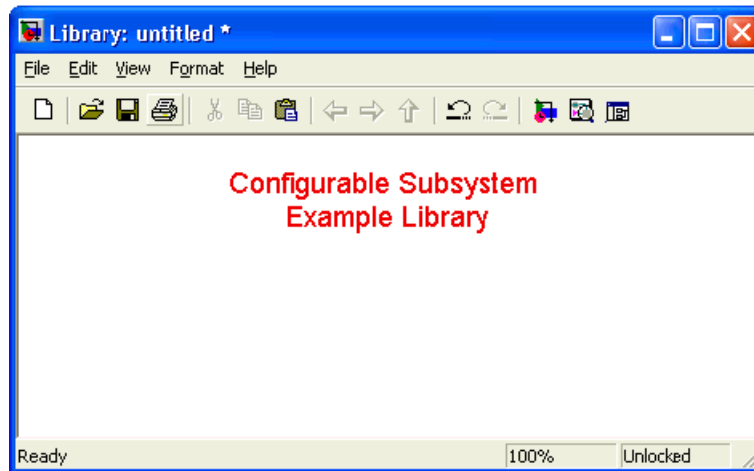
コンフィギャブル サブシステムと System Generator

コンフィギャブル サブシステムは、Simulink の基本パーツとして使用可能なブロックであり、基になるブロックを複数指定できるブロックです。各ブロックはそれぞれ可能なインプリメンテーションであり、どのインプリメンテーションを使用するかを自由に選択できます。たとえば、**System Generator** で汎用 FIR フィルタをコンフィギャブル サブシステムとして指定し、そのサブシステムの基になるブロックとして特定の FIR フィルタを複数指定します。高速だがハードウェア リソースを多く必要とするフィルタ、比較的低速だが必要なハードウェア リソースが少ないフィルタなどを指定できます。フィルタの選択を切り替えることにより、ハードウェア コストまたはスピードを優先した場合の動作を調べることができます。

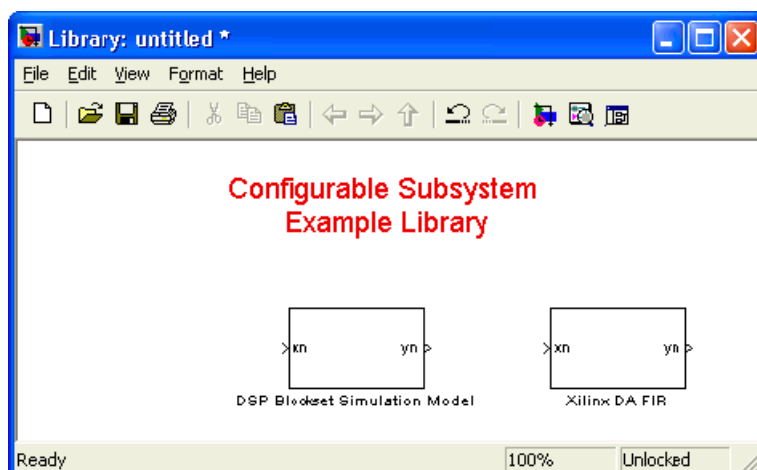
コンフィギャブル サブシステムの定義

コンフィギャブル サブシステムを定義するには、**Simulink** ライブラリを作成します。コンフィギャブル サブシステムの基になるブロックは、このライブラリで管理されます。ライブラリを作成するには、次の手順に従います。

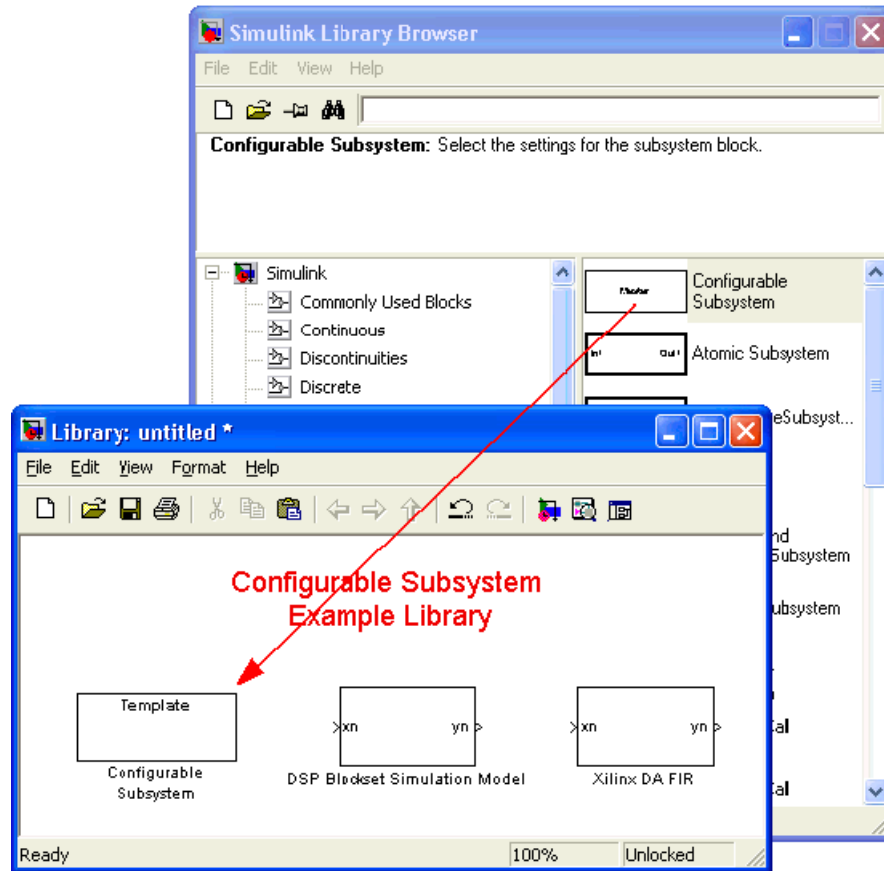
- 空のライブラリを作成します。



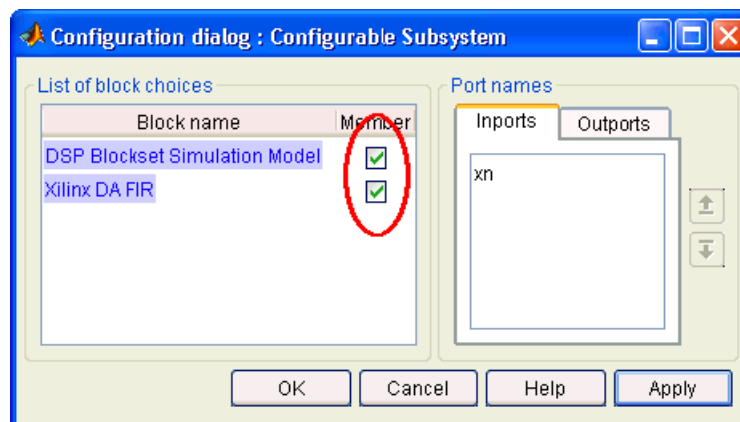
- 作成したライブラリに基になるブロックを追加します。



- [Simulink] ライブラリ → [Ports & Subsystems] にある Configurable Subsystem テンプレートをライブラリに追加します。



- 必要に応じて、テンプレート ブロックの名前を変更します。
- ライブラリを保存します。
- テンプレートをダブルクリックして開きます。
- [設定ダイアログ] ダイアログ ボックスで、必要なブロックのチェック ボックスをオンにします。

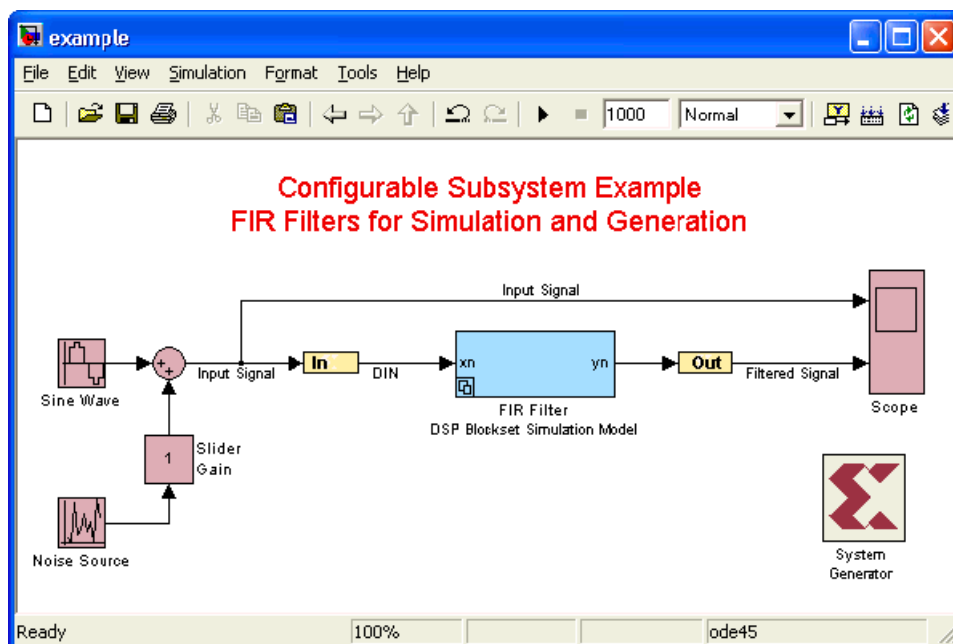


- [OK] をクリックし、その後ライブラリを保存します。

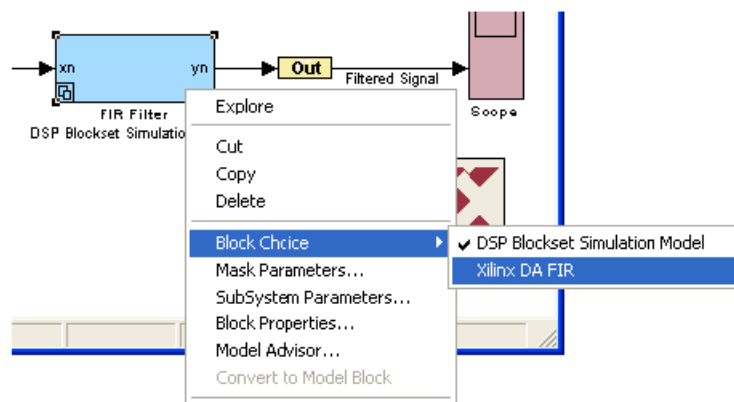
コンフィギャブル サブシステムの使用

デザインでコンフィギャブル サブシステムを使用するには、次の手順に従います。

- コンフィギャブル サブシステムを定義するライブラリを作成します。
- ライブラリを開きます。
- ライブラリからテンプレートをデザインの適切な位置にドラッグします。
- ドラッグしたテンプレートが、コンフィギャブル サブシステムのインスタンスになります。



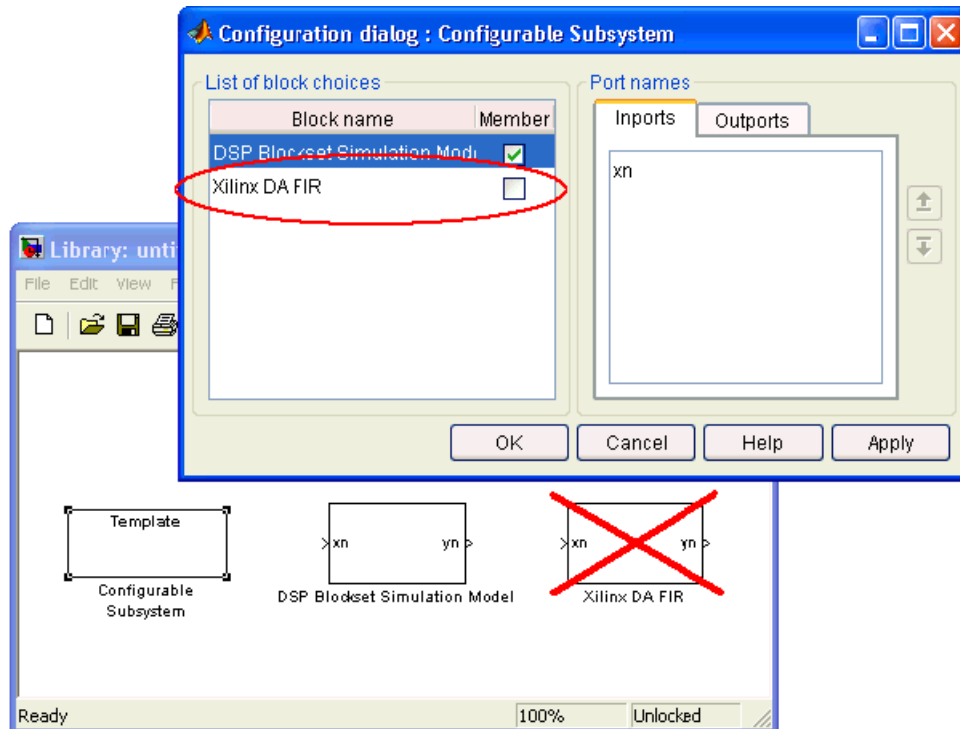
- インスタンスを右クリックし、[ブロックの選択] をクリックして、使用するブロックを選択します。



コンフィギャブル サブシステムからのブロックの削除

コンフィギャブル サブシステムからブロックを削除するには、次の手順に従います。

- コンフィギャブル サブシステムのライブラリを開き、ライブラリのロックを解除します ([編集] → [ライブラリのロックを解除] をクリック)。
- テンプレートをダブルクリックし、削除するブロックのチェック ボックスをオフにします。
- [OK] をクリックし、その後ブロックを削除します。



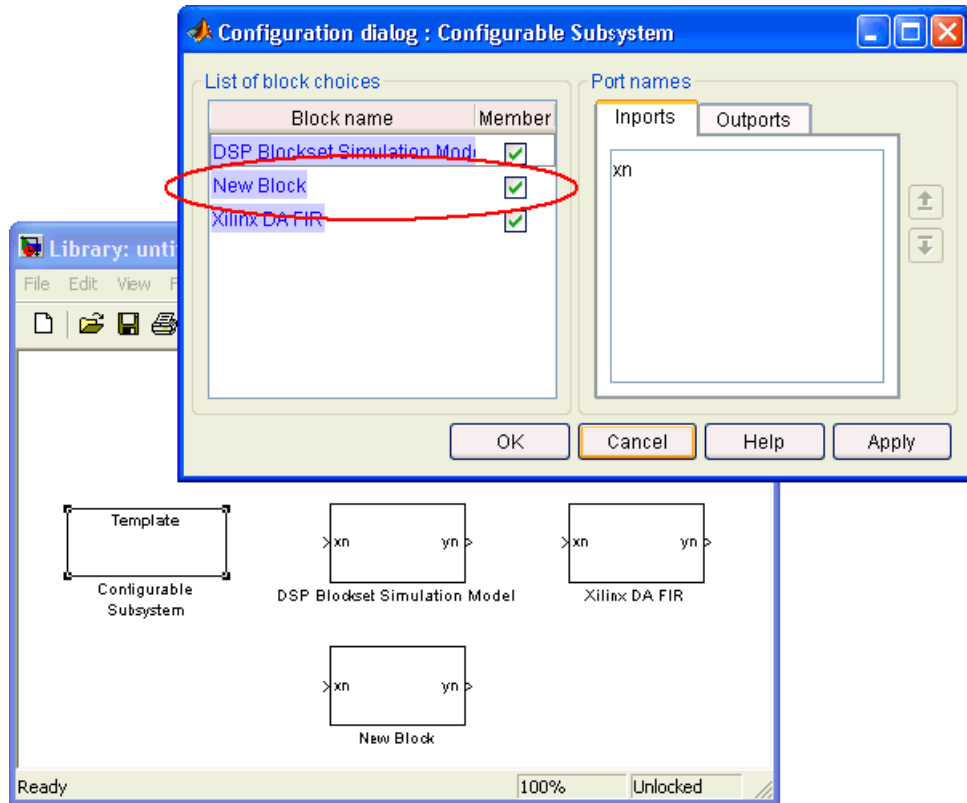
- ライブラリを保存します。
- Ctrl + D を押し、デザインをコンパイルします。
- 必要に応じて、コンフィギャブル サブシステムの各インスタンスの選択をアップデートします。

コンフィギャブル サブシステムへのブロックの追加

コンフィギャブル サブシステムにブロックを追加するには、次の手順に従います。

- コンフィギャブル サブシステムのライブラリを開き、ライブラリのロックを解除します ([編集] → [ライブラリのロックを解除] をクリック)。
- ブロックをライブラリにドラッグします。

- テンプレートをダブルクリックし、追加したブロックのチェック ボックスをオンにします。



- [OK] をクリックし、その後ライブラリを保存します。
- Ctrl + D を押し、デザインをコンパイルします。
- 必要に応じて、コンフィギャブルサブシステムの各インスタンスの選択をアップデートします。

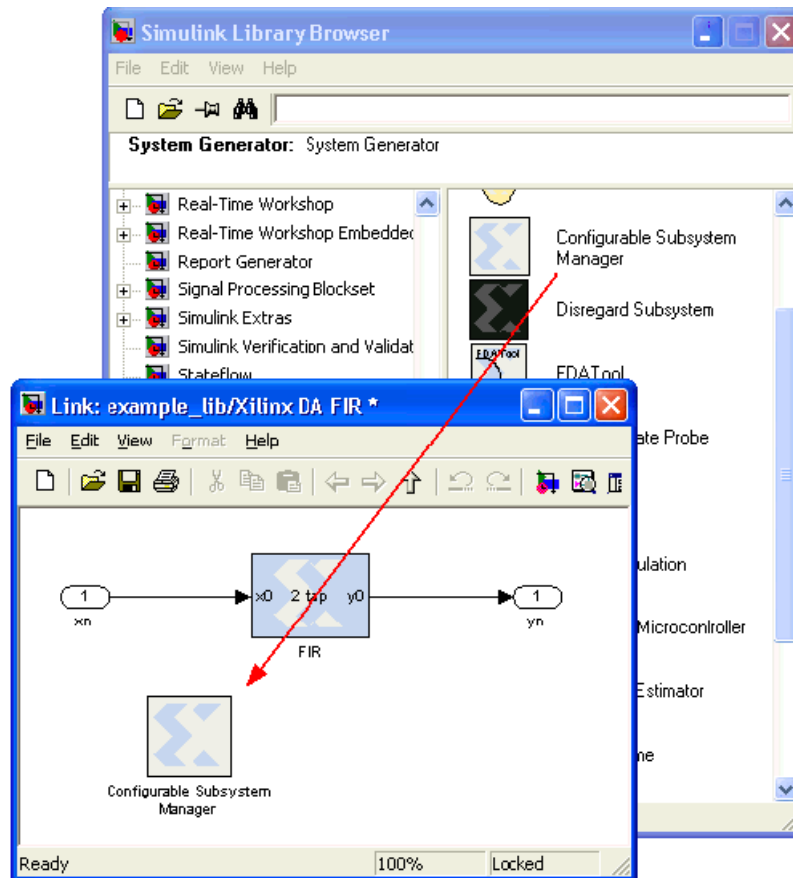
コンフィギャブルサブシステムからのハードウェアの生成

System Generator では、ブロックはシミュレーションとハードウェア生成の両方に使用されます。コンフィギャブルサブシステムが含まれる場合、シミュレーションで 1 つのブロックを使用し、ハードウェア生成で別のブロックを使用することがあります。たとえば、シミュレーションを実行するのに通常の System Generator ブロックを使用し、対応する HDL を生成するのにブラックボックスを使用する場合などです。System Generator の Configurable Subsystem Manager ブロックを使用すると、シミュレーションで通常のコンフィギャブルサブシステムを使用し、ハードウェア生成では別のブロックを使用することが可能です。

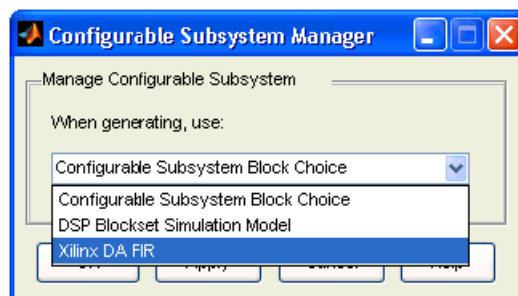
Configurable Subsystem Manager ブロックを使用するには、次の手順に従います。

- コンフィギャブルサブシステムのライブラリを開き、ライブラリのロックを解除します ([編集] → [ライブラリのロックを解除] をクリック)。
- ライブラリのブロックの 1 つをダブルクリックして開きます。テンプレート以外のブロックで、サブシステムであるものを選択してください。ライブラリにサブシステムがない場合は、Configurable Subsystem Manager は使用できません。

- 開いたサブシステムに、Configurable Subsystem Manager ブロックをドラッグします。
Configurable Subsystem Manager ブロックは、[Xilinx Blockset] → [Tools] にあります。



- Configurable Subsystem Manager ブロックをダブルクリックしてダイアログ ボックスを開き、ハードウェア生成に使用するブロックを選択します。



- [OK] をクリックし、その後サブシステムおよびライブラリを保存します。

コンフィギャブル サブシステムに関する The MathWorks 社の説明は、次のサイトに掲載されています。

<http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/configurable subsystem.shtml>

高パフォーマンス FPGA デザインに関するメモ

次の推奨事項に従うと、System Generator で効率的な高パフォーマンスのハードウェアを実現できます。

ブロックのパラメータ ダイアログ ボックスに含まれている「Hardware notes」を読む

ブロックのパラメータ ダイアログ ボックスに含まれている「Hardware notes」を読むようにしてください。Xilinx Blockset ライブラリの多くのブロックでは、最も効率の良いハードウェアインプリメンテーションを達成する方法が記載されています。たとえば、Scale ブロックにはこのブロックにハードウェアコストがかからないことが記述されていますが、同じ目的で使用するもののある Shift ブロックでは場合によってハードウェアが使用されることが記述されています。

デザインの入力と出力にレジスタを付ける

デザインの入力および出力にレジスタを付けてください。レジスタを付けるには、Gateway In ブロックの後および Gateway Out ブロックの前に、レイテンシ 1 の Delay ブロックまたは Register ブロックを配置します。Register ブロックの機能のいずれかを使用すると、追加のハードウェア リソースが必要になります。

I/O に 2 つのレジスタを付けると、有益な場合があります。この場合、Register ブロックを 2 つインスタンス化するか、レイテンシが 1 の Delay ブロックを 2 つインスタンス化します。このようにすると、1 つのレジスタが IOB 内に配置され、もう 1 つのレジスタが FPGA のロジックの横に配置されます。Delay ブロックのレイテンシを 2 にすると、SRL16 を使用してインプリメントされるので、IOB 内に配置されません。

パイプライン レジスタを挿入する

可能な限り、パイプライン レジスタを挿入してください。パイプラインは、Delay ブロックを使用して効率的にインプリメントできます (SRL16 プリミティブが使用される)。レジスタに初期値を指定する必要がある場合は、Register ブロックを使用してください。

[Saturate] および [Round] オプションは必要な場合以外は使用しない

これらのオプションを使用すると、リソースが多く使用され、パフォーマンスが低下します。必要な場合にのみ使用してください。

System Generator のタイミング解析ツールおよび消費電力解析ツールを使用する

System Generator のタイミング解析ツールおよび消費電力解析ツールを使用できます。System Generator にはタイミング問題を解決するのに役立つタイミング解析ツールが含まれており、このツールで最も遅いパス、タイミング要件を満たしていないパスを表示できます。消費電力解析ツール XPower では、簡単な概算解析または HDL シミュレーションを使用した完全な解析を実行できます。詳細は、「[タイミングおよび消費電力解析用のコンパイル](#)」を参照してください。

すべての Gateway ブロックでデータ レート オプションを設定する

Gateway In ブロックと Gateway Out ブロックのパラメータ ダイアログ ボックスで、[IOB timing constraint] に [Data rate] を選択します。このオプションを選択すると、IOB が動作するデータ レートで制約されます。このレートは、System Generator トークンの [Simulink system period (sec)] の値、およびデザイン内のその他のサンプリング周期に対する Gateway ブロックのサンプリング レートによって決定されます。

クロック イネーブル (CE) のファンアウトを低減する

ISE の MAP ツールで使用するアルゴリズムでは、ファンアウトの大きいネットでロードを繰り返し分割することによりレジスタの複製および配置が行われるので、CE のファンアウトが大きい System Generator デザインで FMAX が向上しています。

System Generator ではこの機能がデフォルトでイネーブルになっていますが、ファンアウトの低減は ISE でのマップ プロセス時に実行されるので、次のマップ オプションをオンにする必要があります。

- [Perform Timing-Driven Packing and Placement] : オン
- [Map Effort Level] : [High]
- [Register Duplication] : オン

ISE Project Navigator フローを使用している場合は、これらのマップ オプションはデフォルトでオンになっていますが、ビットストリーム生成などの System Generator フローを使用する場合は、bitstream.opt ファイルを変更するか独自の OPT ファイルを使用して、これらのマップ オプションをオンにする必要があります。詳細は、「[XFLOW のオプション ファイル](#)」を参照してください。

FPGA 物理デザイン ツールを使用した System Generator デザインの処理

HDL シミュレーション

System Generator では、生成されたプロジェクトと ModelSim シミュレータで使用するカスタム DO ファイルが生成されます。これらの DO ファイルを使用するには、ModelSim が必要です。シミュレーションは、ModelSim をスタンドアロンで起動して実行するか、ISE Project Navigator で ModelSim を指定し、インプリメンテーション フローの一部として実行できます。

IP のコンパイル

デザインをシミュレーションする前に、IP (コア) ライブラリを ModelSim 用にコンパイルする必要があります。

ModelSim SE

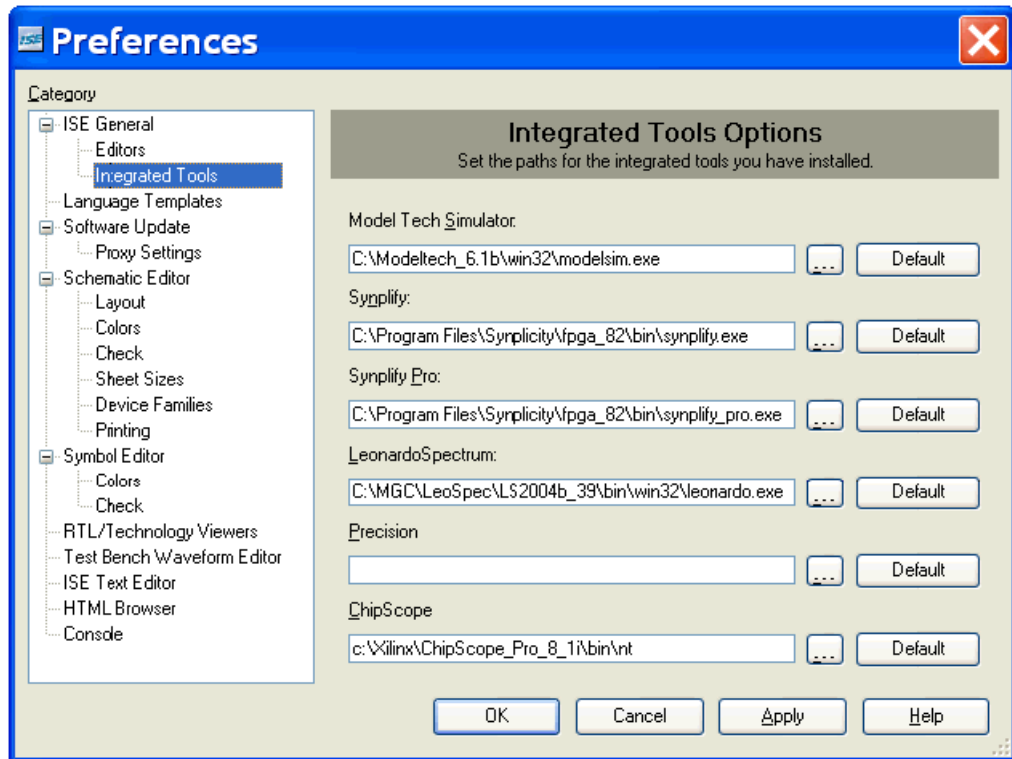
IP ライブラリをコンパイルする方法は、複数あります。COMPXLIB の実行方法は、『コマンド ライン ツール ユーザー ガイド』の「COMPXLIB」の章を参照してください。

Windows のコマンド プロンプトから、COMPXLIB を使用して必要な HDL ライブラリをコンパイルできます。たとえば、すべての HDL ライブラリを ModelSim SE 用にコンパイルするには、次のコマンドを使用します。

```
compplib -s mti_se -f all -l all
```

Project Navigator からの ModelSim を使用したシミュレーションの実行

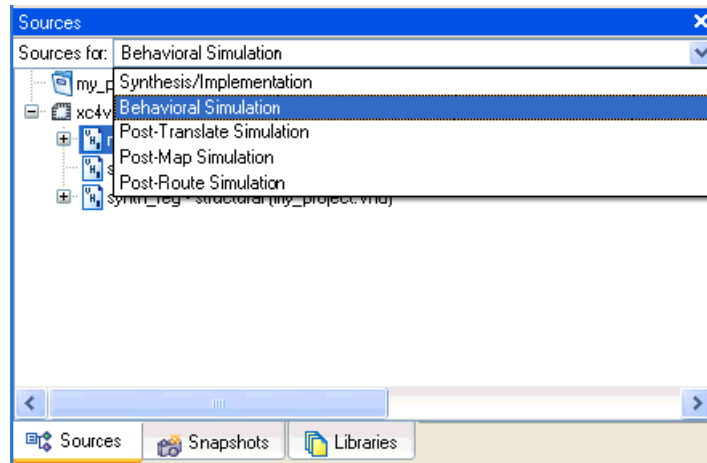
Project Navigator から ModelSim を起動するには、ModelSim のインストール場所を指定する必要があります。ModelSim のインストール場所を指定するには、Project Navigator で [Edit] → [Preferences] をクリックし、[Preferences] ダイアログ ボックスの [Category] で [ISE General] → [Integrated Tools] をクリックし、[Model Tech Simulator] に ModelSim への完全なパスを入力します。このパスには、実行ファイルの名前も含める必要があります。



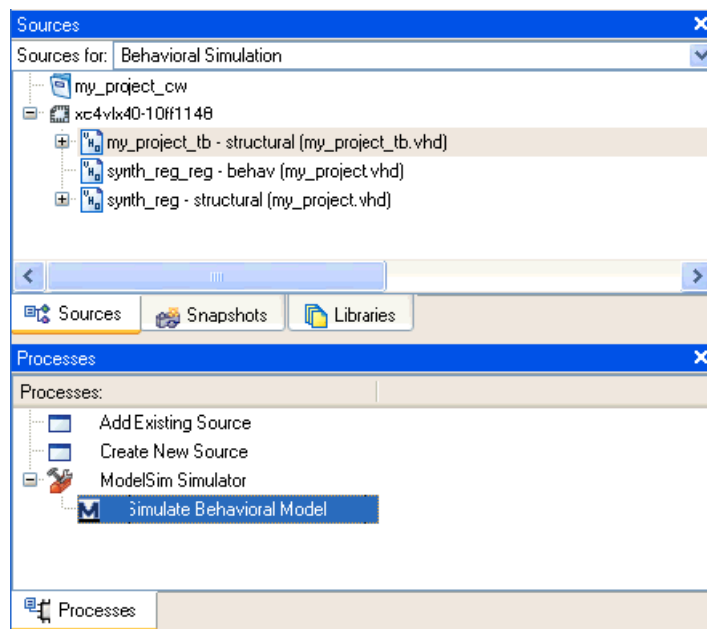
Project Navigator プロジェクトは、インプリメンテーションの 4 つの段階でシミュレーションを実行できるよう設定されています。System Generator トークンのパラメータ ダイアログ ボックスで [Create testbench] をオンにすると、次の 4 つの DO ファイルが生成されます。

- `pn_behavioral.do`: HDL ファイルに対するビヘイビア (HDL) シミュレーション用 (合成およびインプリメンテーション前)。
- `pn_posttranslate.do`: 変換 (NGDBuild) 後のシミュレーション用。
- `pn_postmap.do`: マップ後のシミュレーション用。このファイルには、バックアノテートされたシミュレーションも含まれます。
- `pn_postpar.do`: 配置配線後のシミュレーション用。このファイルには、バックアノテートされたシミュレーションも含まれます。

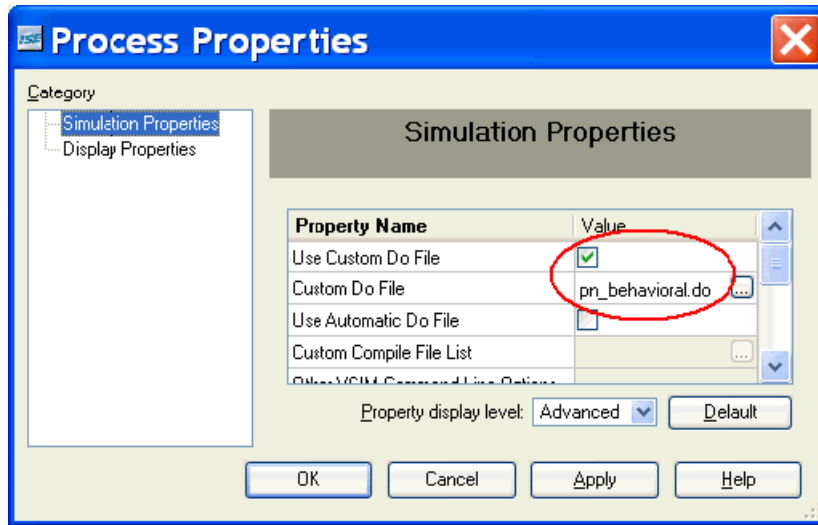
Project Navigator の [Hierarchy] ペインで、[Sources for] ドロップダウン メニューから [Behavioral Simulation] (pn_behavioral.do に対応)、[Post-Translate Simulation] (pn_posttranslate.do に対応)、[Post-Map Simulation] (pn_postmap.do に対応)、または [Post-Route Simulation] (pn_postpar.do に対応) を選択します。



Project Navigator の [Hierarchy] ペインで <your design>_tb.vhd/.v を選択すると、[Processes] ペインに ModelSim のプロセスを含む [ModelSim Simulator] が表示されます。[ModelSim Simulator] の左側にあるプラス記号 (+) をクリックすると、ModelSim のプロセスが表示されます。



[Process Properties] ダイアログ ボックスでは、既に System Generator で生成された DO ファイルがカスタム DO ファイルとして指定されています。



シミュレーション プロセスをダブルクリックすると、ModelSim コンソールが開き、カスタム DO ファイルが使用されて、System Generator テストベンチがコンパイルおよび実行されます。テストベンチでは Simulink で生成された入力スティミュラスと同じものが使用され、HDL シミュレーションの結果が Simulink シミュレーションの結果と比較されます。デザインにエラーがなければ、ModelSim によりシミュレーションが正常に終了したことが示されます。

FPGA ビットストリームの生成

ザイリンクス ISE Project Navigator

System Generator では、コード生成の際にザイリンクス ツールおよびパートナー ツールで使われるプロジェクト ファイルが生成されます。これらのプロジェクト ファイルの 1 つは、ザイリンクス ISE Project Navigator 用です。このプロジェクト ファイルを開くと、System Generator デザインを Project Navigator にインポートし、Project Navigator からデザインを合成、シミュレーション、インプリメントできます。このプロジェクト ファイルは <design_name>_cw.isc という名前で、System Generator トークンで指定したターゲット ディレクトリに保存されます。

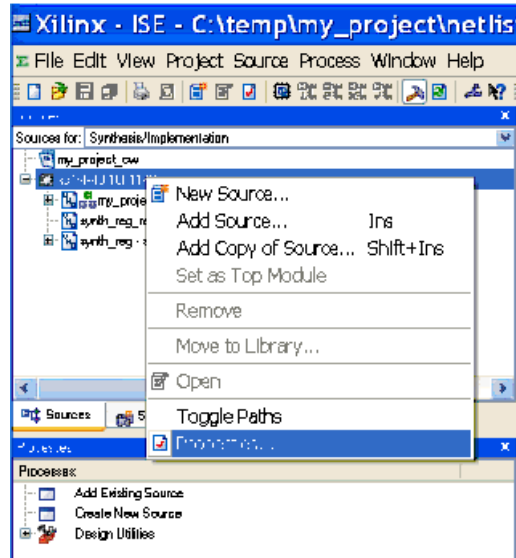
メモ：この後の説明では、my_project_cw.isc という名前を使用します。

System Generator プロジェクトを開く

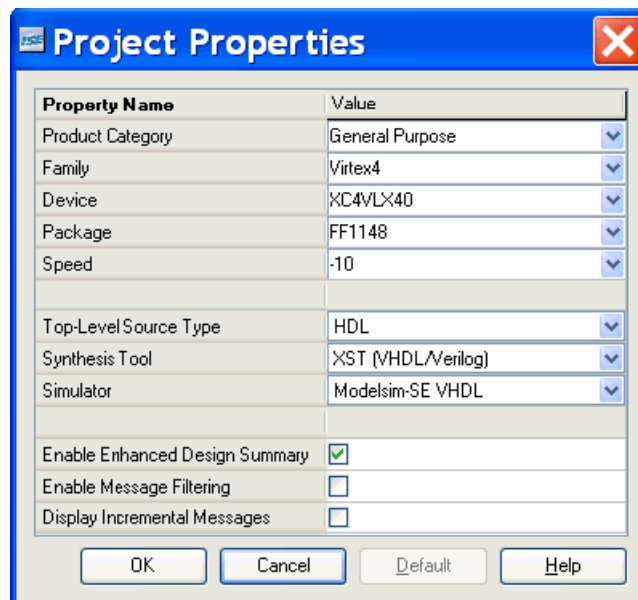
Windows エクスプローラから ISE ファイルをダブルクリックします。Project Navigator が起動し、my_project_cw.isc が開きます。また、Project Navigator を起動してから、[File] → [Open Project] をクリックし、my_project_cw.isc を選択して開くこともできます。

System Generator プロジェクトのカスタマイズ

System Generator プロジェクトを開くと、System Generator トークンで指定した合成ツール、デバイス、パッケージ、およびスピード グレードが既に設定されています。これらの設定を変更するには、[Hierarchy] ペインでデバイス名を右クリックし、[Design Properties] をクリックします。

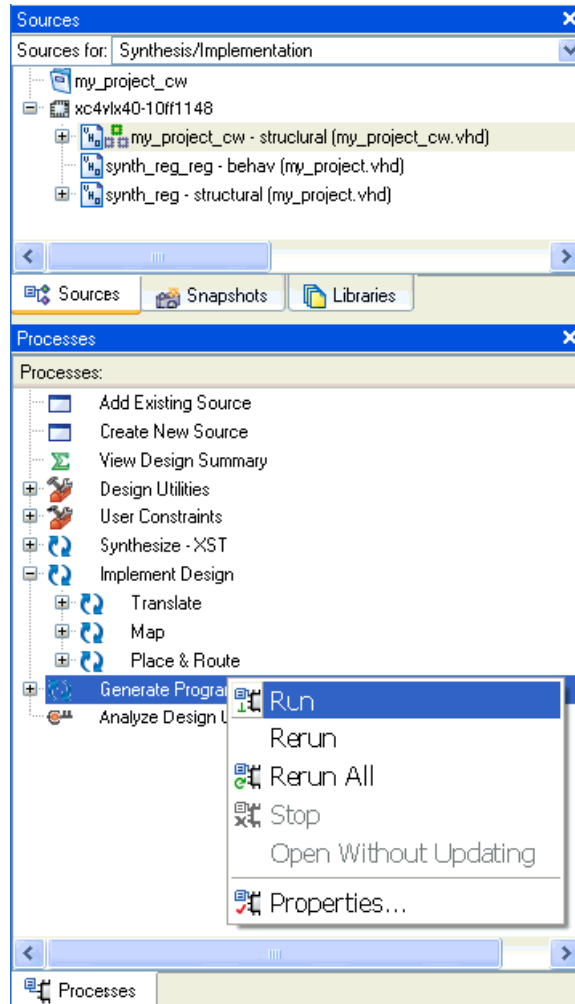


[Design Properties] ダイアログ ボックスが開きます。このダイアログ ボックスで、デバイス、パッケージ、スピード グレード、合成ツールなどを変更できます。デバイス ファミリを変更した場合は、System Generator で生成した IP コアを再生成する必要があります。この場合、System Generator に戻ってプロジェクトを再生成することをお勧めします。



デザインのインプリメンテーション

Project Navigator では、プロジェクトを作業するのにさまざまなオプションが提供されています。Constraints Editor、レポート ビューアなどのツールを起動できます。デザインをインプリメントするには、合成からビットストリーム生成までのプロセスを実行するよう指示するだけです。[Hierarchy] ペインで、最上位 HDL モジュールを選択します。この例では、最上位 HDL モジュールの名前は my_project_cw - structural です。[Processes] ペインに最上位 HDL モジュールに対して実行可能なプロセスが表示されます。

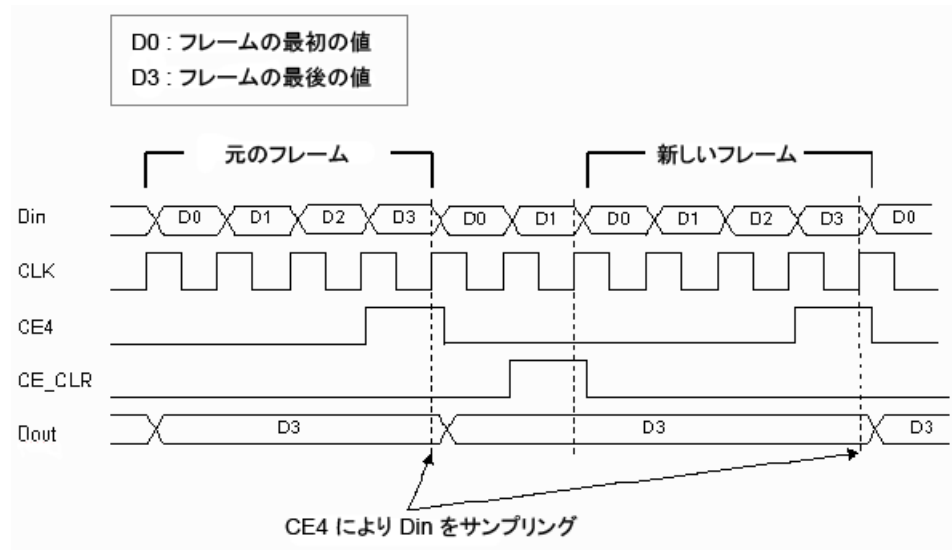


[Processes] ペインで [Generate Programming File] を右クリックして [Run] をクリックすると、選択した HDL ソースからプログラム ファイル (FPGA ビットストリーム) を生成するのに必要なプロセスが実行されます。[Console] パネルに、合成、変換、マップ、配置配線、ビットストリームの生成などが実行されていることが表示されます。

デザインのビットストリームを生成したら、それまでに生成されたすべてのファイルにアクセスできます。

自動生成されたクロック イネーブル ロジックのリセット

System Generator では、Simulink 環境で FPGA ハードウェアのビット精度およびサイクル精度のモデリングを提供します。デフォルトの [Clock Enables] オプションに加え、複数のクロック供給オプションがあります。[Clock Enables] オプションを選択すると、System Generator では 1 つのクロックとクロック イネーブル (ce) を使用して、さまざまなサンプリング ドメインが同期化されます。マルチレート クロック 供給については、「[コンパイル結果](#)」を参照してください。多くの場合、System Generator モデルは大型システム デザインの一部として組み込まれますが、大型システム デザインでは、データ パス サンプリングの開始を指定するために動的な制御が必要です。大型システム デザインで動的な制御を可能にするため、最上位 HDL クロック ラップに、クロック イネーブル 生成ロジックをリセットするためのオプションの ce_clr ポートを含めることができます。次の図は、ce_clr 信号がディアサートされた後に CE4 信号生成ロジックがリセットされる様子を示します。



ce_clr 信号を使用した場合の動作は、元の System Generator デザインを使用してシミュレーションすることはできません。Simulink 内でこの動作をモデリングするには、次の手順に従います。

1. System Generator トークンのパラメータ ダイアログ ボックスで、[NGC Netlist] (「[NGC ネットリストへのコンパイル](#)」を参照) を選択し、[Provide clock enable clear pin] をオンにします。
2. [Generate] ボタンをクリックします。
3. MATLAB の [Command Window] に次のコマンドを入力し、変換後の VHDL ネットリストを生成します。Verilog ネットリストを生成する場合は、「-ofmt verilog」を使用してください。

```
>> !netgen -ofmt vhd1 ./<target_directory>/<design_name>_cw.ngc
```
4. 変換後の VHDL/Verilog ファイルをブラック ボックスとして Simulink に取り込み、HDL 協調シミュレーションを使用して、ce_clr 信号をアサートした場合のデザインへの影響をモデリングします。

ce_clr とレート変更ブロック

ce_clr 信号は、すべてのサンプリング データ信号のサンプリング位相を変更します。これにより、ce 信号を使用して周期的な動作を達成しているレート変更ブロックの機能が変更される可能性があります。次の表に、ce_clr 信号をディアサートした場合の各レート変更ブロックの動作を示します。これらのブロックは、変換後の HDL モデルをブラック ボックスとしてインポートし、シミュレーションすることにより、特性化されています。

表 1-1 :

ブロック名	ce_clr に同期	ce_clr がディアサートされると ce に同期 (1 サンプリング サイクル遅延)	ce_clr がディアサートされた後 次の ce パルスまでの動作
Down Sample ([Last value of frame] をオン)	する	なし	新しい ce 信号が到着するまで最後のサンプル値が保持されます。
Down Sample ([First value of frame] をオン)	しない	しない	ce_clr 信号がディアサートされた後、再同期は行われません。
Up Sample ([Copy samples] をオン)	する	なし	ハードウェアでは、ワイヤとしてインプリメントされます。
Up Sample ([Copy samples] をオフ、ゼロを挿入)	しない	する	次の ce 信号が到着するまで、最後の値 (ゼロまたはサンプル値) が保持されます。
Time Division Multiplexer	しない	する	残りの入力チャンネルがすべてサンプリングされた後、次の ce が到着するまで出力が 0 に設定されます。ce 信号が到着すると、出力が新しいフレーム定義に再同期化されます。
Time Division Demultiplexer	しない	する	次の ce 信号が到着するまで出力チャンネルが同じ値に保持されます。ce 信号が到着すると、出力が新しいフレーム定義に再同期化されます。
Parallel to Serial	しない	する	残りのデータ ワードがすべてサンプリングされた後、次の ce 信号が到着するまで最後のサンプル ワードが保持されます。ce 信号が到着すると、パラレル データからシリアル データへの変換が開始します。

表 1-1 :

ブロック名	ce_clr に同期	ce_clr がディアサートされると ce に同期 (1 サンプルング サイクル遅延)	ce_clr がディアサートされた後の ce パルスまでの動作
Serial to Parallel	しない	する	ce_clr 信号がアサートされると、出力が保持されます。ce_clr がディアサートされると、入力サンプル フレームの最後の値がサンプルングされ、次の ce 信号で出力レートに対応して出力が現れます。
Addressable Shift Register (ASR)	しない	する	ce_clr 信号がアサートされると、シフトレジスタの値が保持されます。ce_clr がディアサートされると、保存されていた値がシフトアウトされ、新しい値がシフトレジスタに入力されます。
Polyphase FIR	しない	しない	オプションのリセット ポートを使用しない場合、ce_clr 信号は機能しません。オプションのリセット ポートを使用すると、ce_clr 信号がディアサートされた後に FIR がリセットされます。

ce_clr の使用に関する推奨事項

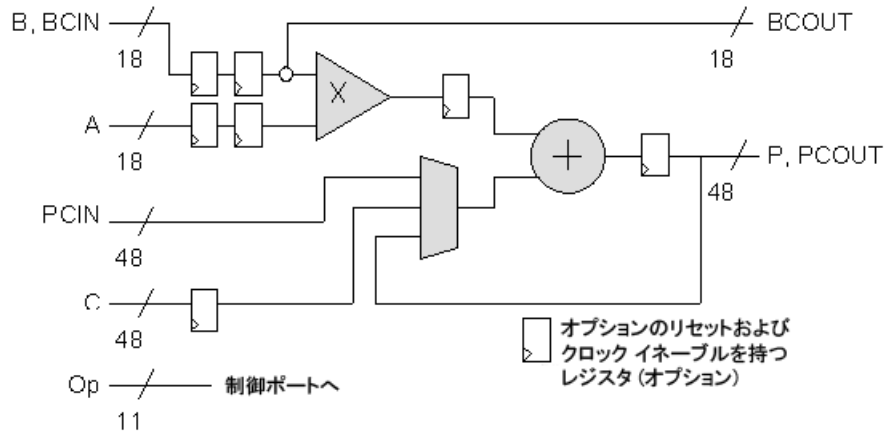
上記の結果から、ce_clr 信号を使用する場合は次の推奨事項に従ってください。

- Down Sample ブロックで [First value of frame] をオンにしている場合は、[Last value of frame] をオンにした Down Sample ブロックを使用した等価回路に置き換えてください。
- ce_clr がディアサートされた後に、N クロック サイクル間無効なデータが出力されることを考慮してください。N は、ブロックの最も遅い ce です。
- Down Sample ブロックでは [Last value of frame] をオンにし、Up Sample ブロックでは [Copy samples] をオンにします。
- N サイクル間無効なデータが出力されるのが好ましくない場合は、Parallel to Serial、Serial to Parallel、Time Division Multiplexer、Time Division Demultiplexer ブロックの代わりに、Counter、Mux、Up Sample、Down Sample ブロックを使用して等価回路を作成してください。等価回路では、リセット ポートを最上位に含め、ce_clr ポートを駆動する信号に接続します。
- 積和演算などの演算を実行するのに使用されるカウンタをリセットするには、ce_clr 信号に接続されたユーザー リセットと、Clock Enable Probe ブロックからの ce 信号を組み合わせで使用します。
- ce_clr 信号のデザインへの影響を、変換後の HDL モデルをブラック ボックスとしてインポートし、シミュレーションして確認します。

DSP48 の設計手法

DSP48 について

ザイリンクス Virtex および Spartan デバイスには、DSP48 (XtremeDSP スライスとも呼ばれる) という DSP アプリケーション用の効率的な機能ブロックが含まれています。DSP48 は、System Generator のブロックとして提供されています。このブロックは、DSP48 UNISIM プリミティブのラッパです。このプリミティブのアーキテクチャおよび使用方法は、ご使用のデバイスの DSP48 ユーザー ガイドを参照してください。



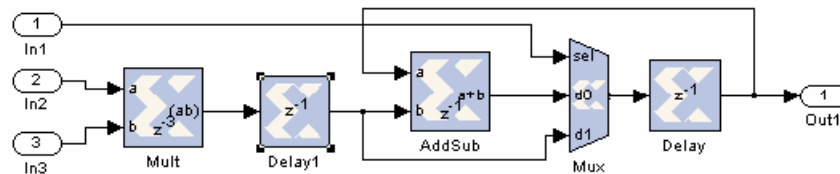
DSP48 では、48ビット 加算器を含む 18X18ビット の符号付き乗算器とプログラマブル マルチプレクサを組み合わせ、加算器の入力を選択しています。基本演算 $p = a * b + (c + cin)$ をインプリメントしますが、ほかの演算も動的に選択できます。オプションの入力および乗算器パイプラインレジスタも含まれており、最高スピードを達成するために使用できます。また、隣接する DSP48 ブロックとの間に高パフォーマンス ローカル インターコネクトもあります (BCIN, BCOUT, PCIN-PCOUT)。DSP48 では、対称丸めもサポート されます。これらの機能により、500MHz 以上で動作可能な高速 DSP システムを構築することが可能です。

System Generator で DSP48 をプログラムするには、次の 3 つの方法があります。

- 標準コンポーネントを使用する : Mult や AddSub ブロックを使用するか、MACFIR フィルタなどの IP コア を使用します。この方法は、低速クロックを使用しており DSP48 にマップする必要がない場合に有益です。
- 合成可能なブロックを使用する : デザインが DSP48 の内部アーキテクチャにマップされるよう構成し、合成可能な Mult、AddSub、Mux、Delay ブロックを使用してデザインを作成します。この方法では、ロジック合成により適切な場所に DSP48 が推論されるので、柔軟性が増し、多くの場合高パフォーマンスを達成できます。
- DSP48 ブロックを使用する : System Generator の DSP48 および DSP48 Macro ブロックを使用して、DSP48 ベースのデザインを直接インプリメントします。この方法では最高のパフォーマンスを達成できますが、DSP48 を使用して最高パフォーマンスおよび最小エリアを達成するには、ターゲット アルゴリズムを DSP の内部アーキテクチャに注意深くマップする必要があります。またデザインの物理設計も必要です。

標準コンポーネントを使用したデザイン

Spartan-3 などのザイリンクス FPGA 用のデザインは、Virtex-4 デバイスにコンパイルできます。乗算器は DSP48 ブロックにマップされますが、加算器およびマルチプレクサはコアとして供給され、合成でロジックを最適化できないため、加算器およびマルチプレクサを DSP48 ブロックに含めることはできません。配置配線では MULT18X18S および MULT18X18 は DSP48 ブロックに配置されますが、加算器およびマルチプレクサは DSP48 ブロックに配置されません (マルチプレクサは LUT ベースの加算器に配置される)。

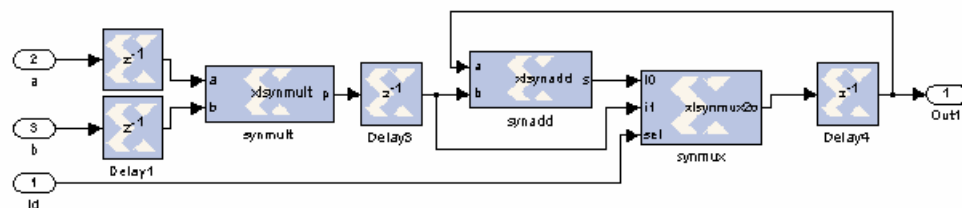


最高のパフォーマンスを得るには、乗算器のレイテンシを 3 に設定し、入力レジスタを追加して、DSP48 の出力から加算器への遅延に対処するようにします。Virtex-4 デバイスでは、Spartan-3 デバイスと異なり、乗算速度はビット幅に依存しません。中程度の速度のデザインは、この方法でうまくいきます。

CORE Generator で提供されている MACFIR ブロックなど、DSP48 用に最適化された IP ブロックを使用する方法や、Architecture Wizard を使用して DSP48 をカスタマイズする方法もあります。これらの方法では、DSP48 を含むロジックをブラックボックスとして System Generator にインポートする必要があります。シミュレーションは、ModelSim の HDL 協調シミュレーションで実行する必要があります。

合成可能な Mult、Mux、AddSub ブロックを使用したデザイン

合成ツールで DSP48 ロジックを推論できます。合成ツールで推論させると、加算器、乗算器、マルチプレクサを DSP48 ブロックに配置でき、リタイミングやレジスタの複製などの合成手法も可能になります。



合成可能なブロックでデザインを作成すると、Synplify Pro および XST の両方で DSP48 が推論され、DSP48 のローカル インターコネクト バス (PCOUT、PCIN、BCOUT、BCIN) が利用されます。上記の例では、3 つのブロックが次の M 関数で定義された MCode ブロックを使用して作成されています。

```
function o = xlsynmux2(i0,i1,sel)
if (sel==0) o=i0; else o=i1; end
```

```
function p = xlsynmult(a,b)
p=a*b;
```

```
function s = xlsynadd(a,b)
s=a+b;
```

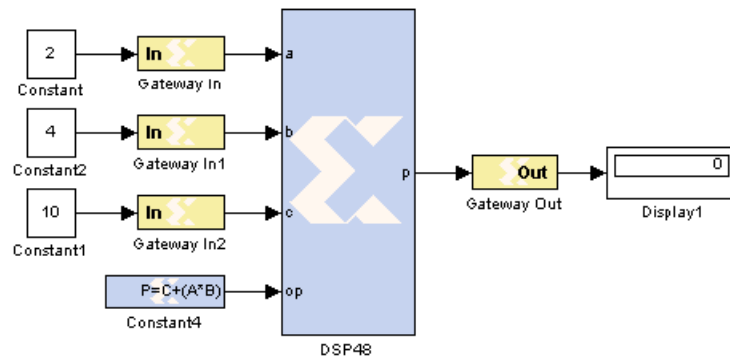
正しく合成されるようにするには、DSP48 にマップ可能な回路を作成し、信号のビット幅を DSP48 の対応するバス幅以下にする必要があります。

ロジック合成ツールは急速に向上しており、DSP48 の推論が必ずしも公式どおりに行われない場合があります。たとえば、マップ可能なデザインであってもマップが効率的でなかったり、マップ結果が一貫しないことがあります。Synplify Pro のゲート レベル テクノロジ ビューアなどのツールを使用して、合成後のネットリストでデザインが正しくマップされているかを確認する必要があります。正しくマップされていない場合は、修正します。完全に合成可能な FIR フィルタのモデル例が、System Generator のインストール ディレクトリの次のパスに含まれています。

```
<path_to_sysgen>\examples\dsp48\synth_fir\synth_fir_tb.mdl
```

DSP48 および DSP48 Macro ブロックを使用したデザイン

DSP48 ブロック

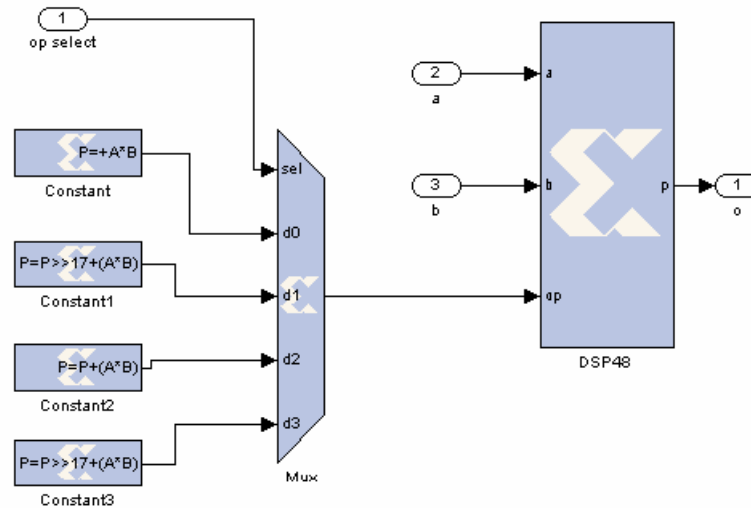


DSP48 ブロックは、DSP48 UNISIM プリミティブのラップであるので、可能な DSP48 デザインであればどんなデザインでもインプリメントできます。ただし、この低レベル インプリメンテーションでは、11 ビットのバイナリ `opmode` を DSP48 の制御ポートに配線する必要があります。Constant ブロックには、DSP 制御フィールドの生成を可能にする特別なモードが含まれています。DSP48 のパイプライン モードおよびローカル インターコネクト バス (PCOUT、PCIN、BCOUT、BCIN) の使用を指定するには、DSP48 ブロックのパラメータ ダイアログ ボックスを使用します。DSP48 ブロックを使用した Simulink モデルの例が、System Generator のインストール ディレクトリの次のパスに含まれています。

```
<path_to_sysgen>\examples\dsp48\dsp48_primitive.mdl
```

DSP48 のダイナミック制御

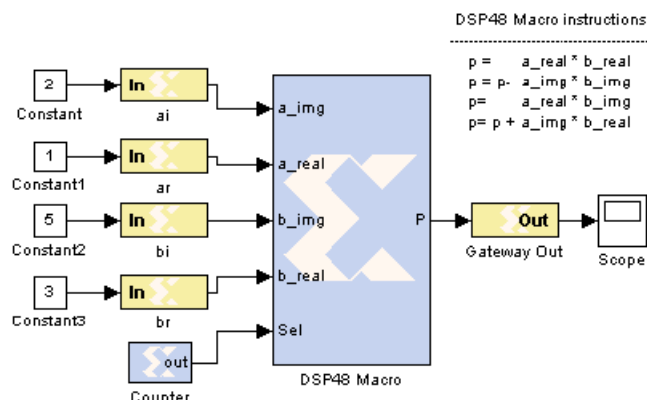
DSP48 には、サイクルごとに操作を変更する機能があります。この機能は、複数のタップが 1 つの乗算器でインプリメントされている FIR フィルタなどのように、DSP48 をリソース共有モードで使用するアプリケーションで有益です。このような制御パターンを生成するには、クロックごとに DSP48 命令を選択するマルチプレクサを使用するのが簡単な方法です。



上図の例では、DSP48 ブロックと Constant ブロックを使用して、35X35 ビット乗算器 (4 クロックサイクル) をインプリメントしています。合成のロジック最適化により、Mux および Constant ロジックが縮小されます。上図の例では、DSP48 ブロックと 4:1 Mux ブロックが 2 つの 4 入力 LUT にインプリメントされます。パラレルおよびシーケンシャル 35X35 ビット乗算器 (シーケンシャルモードにダイナミック操作を使用) をインプリメントする方法を示す Simulink モデルの例が、System Generator のインストール ディレクトリの次のパスに含まれています。

<path_to_sysgen>\examples\dsp48\mult35x35\mult35x35_tb.mdl

DSP48 Macro ブロック



DSP48 Macro ブロックは、DSP48 命令 (ダイナミック命令) のシーケンスを簡単にインプリメントできるようにした DSP48 ブロックのラップです。このブロックでは、入力および出力のデータ型を指定できます。上図の例では、DSP48 Macro ブロックが、4 つの異なる命令のシーケンスを使用した複素乗算器をインプリメントするよう設定されています。命令は、DSP48 Macro ブロックのパラメータ ダイアログ ボックスで入力します。DSP48 Macro ブロックを使用したモデル例が、System Generator のインストール ディレクトリの次のパスに含まれています。

```
<path_to_sysgen>\examples\dsp48\dsp48_macro.mdl
```

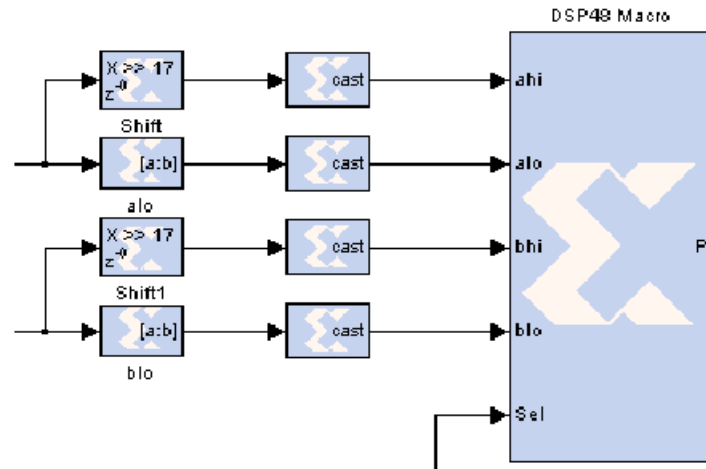
DSP48 Macro ブロックの DSP48 Macro 2.0 ブロックへの置き換え

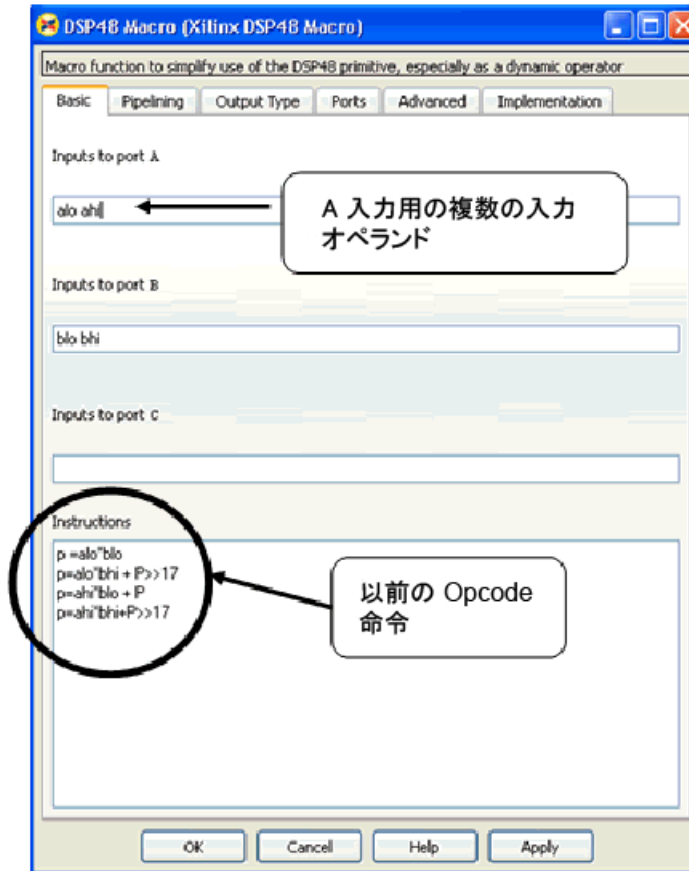
11.4 で、DSP Macro ブロック 2.0 がリリースされました。次に、既存の DSP Macro ブロックを DSP Macro 2.0 ブロックに置き換える方法を示します。

これまでの DSP Macro ブロックと新しい DSP48 Macro 2.0 ブロックとの基本的な違いは、IP のロジックの効率を向上し、サイズを縮小するため、内部入力マルチプレクサ回路が削除されていることです。そのため、DSP48 Macro を含むデザインを DSP48 Macro 2.0 を含むデザインに変換する際にいくつかの注意点があります。たとえば、複数の入力オペランド (A1、A2、B1、B2 など) を指定することはできなくなっています。そのため、DSP48 Macro 2.0 を使用するデザインで次の例に示すように固有の入力オペランドが複数ある場合、単純なマルチプレクサ回路を追加する必要があります。

DSP48 Macro を使用した符号付き 35X35 乗算器

次の DSP48 Macro には、ポート A への入力に 18 ビット入力オペランド alo と ahi、ポート B の入力に blo と bhi があります。入力オペランドおよび Opcode 命令は、次のように指定されます。複数の入力オペランドは、DSP48 Macro ブロック内で処理されます。





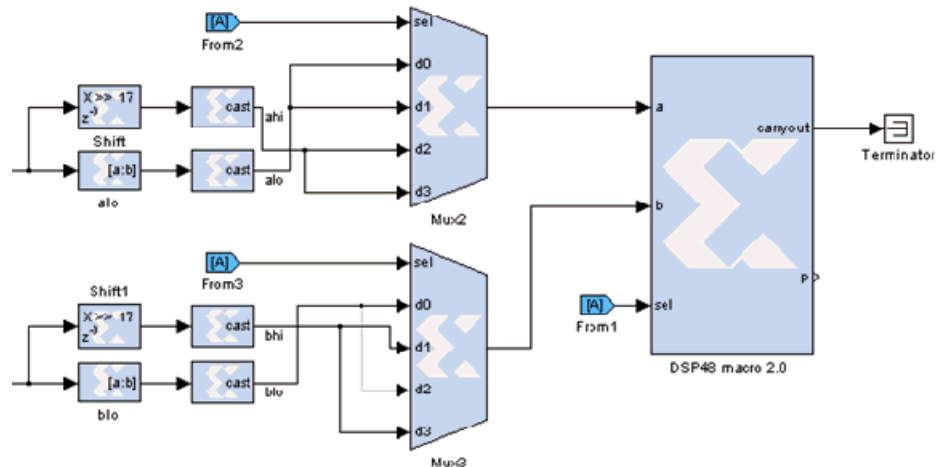
DSP48 Macro 2.0 を使用した符号付き 35X35 乗算器

上記のモデルを DSP48 Macro 2.0 ブロックに移行できます。デザインをアップデートするには、次の手順およびデザイン ガイドラインに従う必要があります。

1. 既存のブロックと新しいブロックで、入力および出力パイプラインレジスタの選択を同一にします。これには、[Pipeline Options] タブの設定を比較します。
2. 複数の固有入力オペランドが必要な場合は、次の図に示すようにマルチプレクサ回路を追加する必要があります。
3. 新しいデザインの機能と質が以前のものと同一であることを確認します。これには、Simulink シミュレーションおよびデザインのインプリメンテーションを実行します。
4. System Generator で DSP48 Macro 2.0 ブロックを使用して前置加算器モードをコンフィギュレーションおよび指定する場合、データ幅入力オペランドなどの一部のデザイン パラメータはデバイスによって異なります。この LogicCORE IP のパラメータの詳細は、LogicCORE IP DSP48 Macro v2.0 のデータシート (DS754) を参照してください。

4 つの入力および 2 つの出力を持つマルチプレクサ回路は、次のように表すことができます。

sel	A 入力	B 入力	Opcode
0	alo	blo	$A*B$
1	alo	bhi	$A*B+P>>17$
2	ahi	blo	$A*B+P$
3	ahi	bhi	$A*B+P>>17$



完成した上記のモデルは、次の場所にあります。

<path_to_sysgen>\examples\dsp48\mult35x35\dsp48macro_mult35x35.mdl

DSP48 設計手法

DSP48 を使用したフィルタの設計

DSP48 ブロックは、FIR フィルタをインプリメントするのに適しています。タイプ 1 およびタイプ 2 の FIR フィルタに DSP48 ブロックを使用する方法を示したモデル例が、System Generator のインストール ディレクトリの次のパスに含まれています。

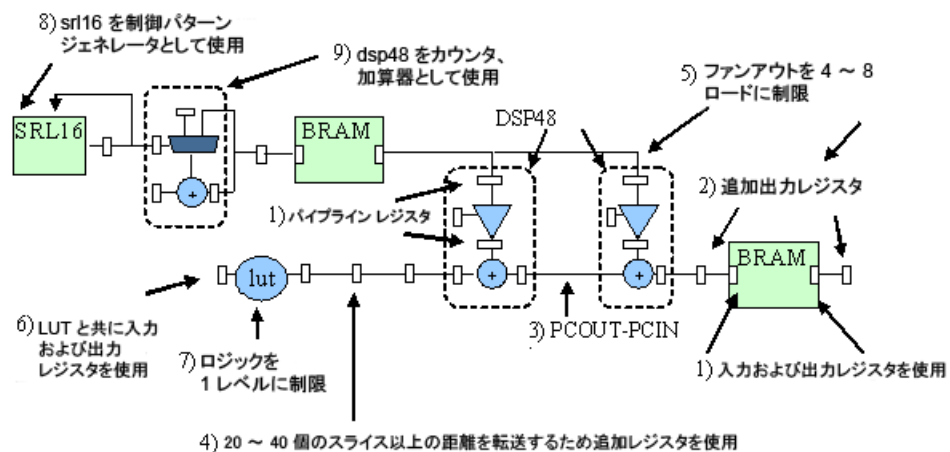
```
<path_to_sysgen>\examples\firs\dsp48_firs_tb.mdl
```

高パフォーマンスデザインの設計手法

DSP48 ベースのデザインには、通常 I/O、BRAM、およびスライス ロジックが必要です。スライス ロジックは、遅延レジスタ、SRL16、マルチプレクサ、カウンタ、および制御ロジックをインプリメントするのに使用されます。DSP48 ブロックは、500MHz 以上のスピードで動作すると予測されるので、ほかのコンポーネントも同じスピードで動作する必要があります。そのため、DSP48 以外のロジックに対して特別な設計手法が必要になります。

500MHz では、各クロック サイクルは 2ns です。Virtex-4 -11 デバイスでは、レジスタの clock-to-out に約 300ps、セットアップに約 300ps が必要です。LUT 遅延は 166ps です。クロック イネーブルや DSP48 および BRAM の信号など特殊な入力/出力では、clock-to-out タイムはほぼ 500ps です。クロック スキューおよびジッタを含めると、ネット遅延に許容されるのは約 1ns です。この制限により、各パスに 1 ネットしか使用できず、かなり短い必要があります。

DSP48 のスピードで確実に動作するようにするためのガイドラインがあります。その一部を次に示します。

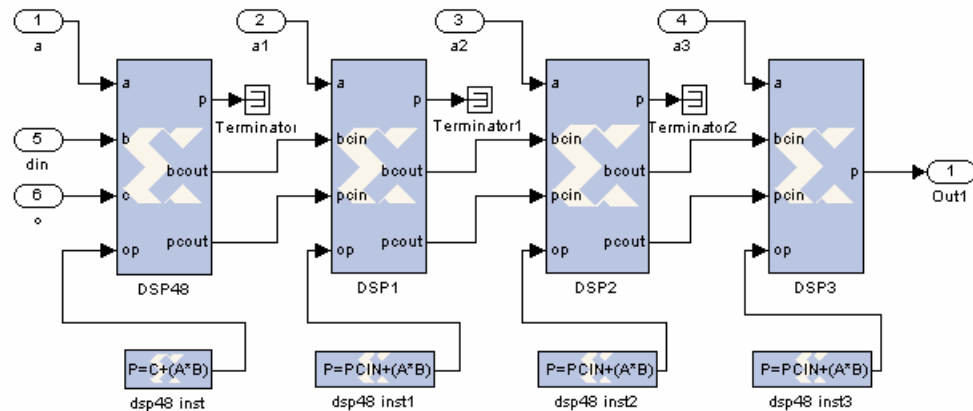


1. DSP48、BRAM16、FIFO16 には、常に入力レジスタ、乗算器、および出力レジスタを使用する。
2. 必要に応じて、DSP48 および BRAM の出力にフリップフロップを追加する。
3. PCOUT と PCIN バスを使用し、DSP48 をチェーン接続できるようにする。
4. 20 ～ 40 個以上のスライスが連続する場合は、パスにレジスタを追加する。
5. 20 個のスライス内にあるファンアウトを 32 個のロードに制限する。
6. LUT ベースのロジックに出力レジスタを追加する。
7. LUT を 1 レベルまたは 4:1 マルチプレクサに制限し、入力または出力でローカルレジスタが使用されるようにする。
8. 制御パターンを生成するには、ステート マシンではなく RAM または SRL16 を使用する。

9. 8 ～ 16 ビット以上のカウンタおよび加算器インプリメントするには DSP48 を使用する。
10. エリア制約「INST ff1* LOC = SLICE_X0Y8:SLICE_X1Y23;」を使用する。

DSP48 ベースのデザインの物理設計

DSP48 を使用して高集積度、高パフォーマンスのデザインを達成するには、正しい配置が必要です。配置配線ツールで適切な配置を達成することは可能ですが、最適な結果を得るには手動で DSP48 および RAM ブロックを配置する必要がある場合があります。DSP48 を使用する際には、追加の考慮事項がいくつかあります。



カスケード接続バス

隣接する DSP48 ブロックは、PCOUT および BCOUT という 2 つのローカル バスで接続されます。PCOUT バスは、1 つの DSP48 から次の DSP48 に累積データを送信します。BCOUT バスは、遅延された B 入力のデータを次の DSP48 に送信します。DSP48 および DSP48 Macro ブロックは、両方とも PCOUT と BCOUT バスをサポートします。上の図はパイプライン 4 タップ タイプ 1 FIR フィルタであり、これらのバスの使用法を示します。

C 入力の共有

DSP48 の各ペアでは、1 つの C 入力に共有されます。リソースの使用を計画する際、このことを考慮する必要があります。配置ツールで C 入力を共有するための最適な配置を見つけられるとは限らないので、可能な限り C 入力は使用しないようにします。

加算器ツリー

ツリー ベースのフィルタ トポロジは、DSP48 の効率的なインプリメンテーションの妨げとなります。加算器ツリーには、分離された 2 入力加算器が必要です。2 入力の 36 ビット加算器は、1 つの DSP48 を使用してインプリメントできますが、これには C 入力が必要であり、乗算器が使用できなくなります。また、DSP48 間の長い信号に追加のパイプライン段が必要な場合もあります。ツリー構造をパイプライン カスケード接続に変換することをお勧めします。

配置

ほとんどのデザインは、DSP48 および BRAM を一部配線することで向上します。エリア制約を使用して LUT ロジックの配置を制約すると、有益な場合があります。

信号の長さ

500MHz では、信号の長さを 20 スライス程度に制限する必要があります。長い信号には、複数のパイプライン段を使用してください。

クロック イネーブル

[Clock Enables] オプションを使用すると、高周波数ではクロック イネーブルが制限パスになることが良くあります。これは、**System Generator** で **LUT** を使用して、デスティネーションでクロックをゲート処理しているからです。クリティカルパスでクロック イネーブルを回避するには、**Up Sample** および **Down Sample** を介したクロックドメインを使用しないようにしてください。これには、システム クロック レート 未満で動作するロジックにクロック イネーブルを手動で追加する必要があります。

配置配線フロー

- **map -timing** コマンドをエフォート レベル **High** で使用し、マップと配置を実行します。
- **trce -v 100** を使用してタイミングを満たしていないネットを確認し、**xflow\design.twr** ファイルでデザインのタイミングを理解します。
- **bitstream_v4.opt** ファイルは、**examples\dsp48** ディレクトリにあります。このファイルは、**System Generator** トークンのパラメータ ダイアログ ボックスで **[Compilation]** を **[Bitstream]** を選択した場合に、**PAR** オプションを設定するために使用できます。

合成フロー

- すべての **LUT** と信号を手動でパイプライン化するのを回避するため、**Synplify Pro** を使用する際はリタイミングおよびパイプライン処理をイネーブルにします。
- **Synplify Pro** では、ファンアウトを 32 に制限し、ネット遅延が長くなるのを回避します。
- **Synplify Pro** でコンパイルしたプロジェクトを開き、**RTL** レベルおよびゲート レベルの表示を使用して生成されたロジックを確認します。
- **syn.pl** ファイルは、**examples\dsp48** ディレクトリにあります。このファイルを **<path_to_sysgen>\scripts** ディレクトリに配置し、**System Generator** の合成オプションを変更します。

ロジックのレベル数

次のガイドラインに従うと、**Virtex-4-11** デバイスを使用した場合に **LUT** を **450MHz** で動作させることが可能です。

- **450MHz** のクリティカルパスでは 1 つのネットしか許容されません。4:1 マルチプレクサからレジスタ、4 入力 **LUT** からレジスタ、または **LUT** を介したネットを直接 **DSP48** に接続できます。
- 16 ビットまでのカウンタを使用できますが、カウント制限のあるカウンタを使用する場合は、パイプラインを追加します。
- アキュムレータまたはカウンタを使用する場合は、イネーブル ラインを反転してアクティブ **Low** にし、クリティカルパスに余分な **LUT** が挿入されるのを回避します。
- 加算器には、ローカル入力レジスタを付ける必要があります。スピードを達成するため、**DSP48** に制御カウンタを配置することが必要な場合があります。

ファンアウト

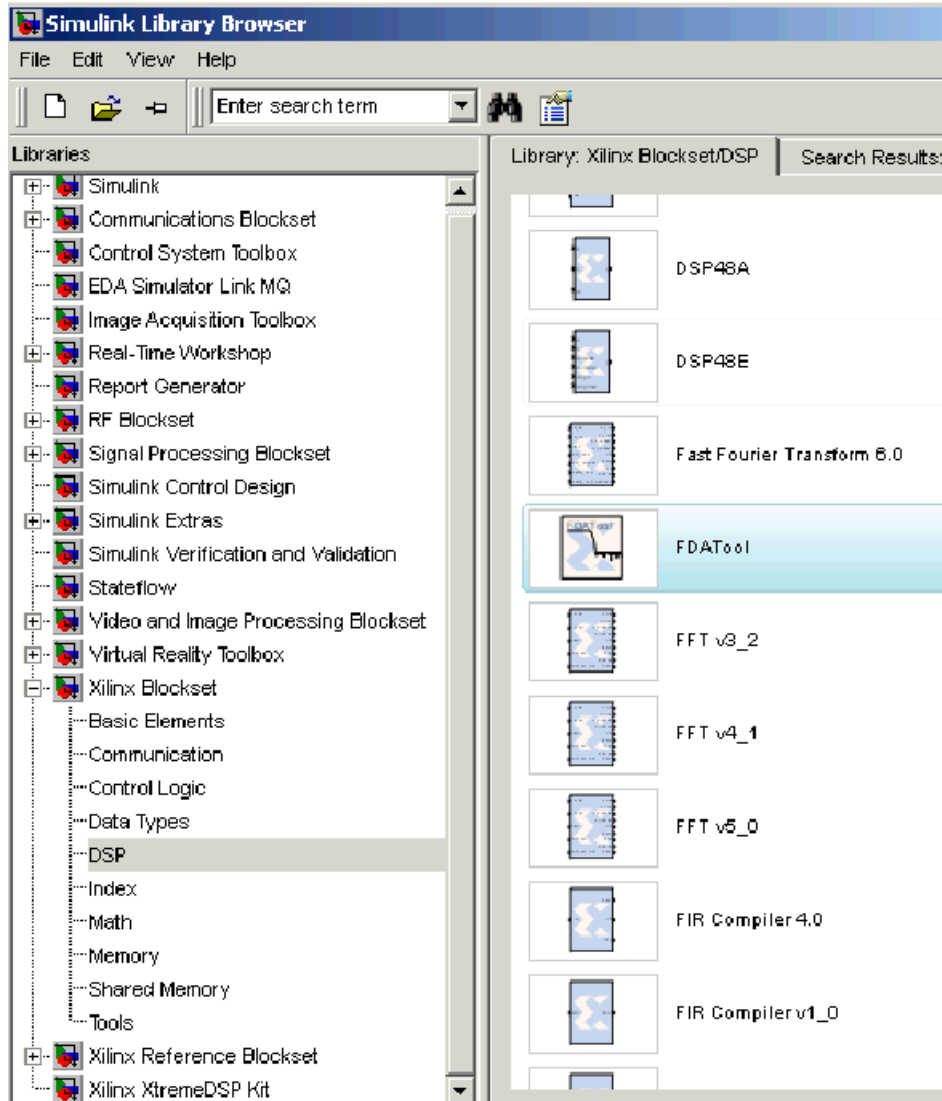
ファンアウトを 32 個の **LUT**、8 個の **DSP48**、または 8 個の **BRAM** 以内にします。これらの信号パスにパイプライン レジスタを追加すると、ファンアウトがこれ以上になるのを回避できます。

レジスタのリタイミング

Delay ブロックのリタイミングを確認し、パイプライン用のレジスタとして使用できるようにします。**Synplify Pro** または **XST** でリタイミングをイネーブルにし、レジスタが最適な位置に移動できるようにします。

FDATool を使用したデジタル フィルタ アプリケーション

次の例では、FDATool ブロックを使用して FIR フィルタを指定、インプリメント、シミュレーションする 1 つの方法を示します。FDATool ブロックをフィルタの次数および係数を定義するために使用し、ザイリンクスブロックセットを使用して MAC Based FIR フィルタを 1 つの MAC エンジンでインプリメントします。その後、周波数応答の質を倍精度の Simulink フィルタ モデルと比較することにより検証します。



この例では 1 つの MAC エンジンの FIR フィルタが使用されていますが、ザイリンクスリファレンスブロックセットの一部として提供されている DSP ライブラリのブロックを見てみることをお勧めします。DSP ライブラリには、複数の MACを含む例や、メモリのタイプが異なるマルチチャネルインプリメンテーションの例が含まれています。

System Generator デモ ライブラリにも、MAC ベースの補間フィルタを効率的にインプリメントする例が示されています。デモを参照するには、MATLAB の [Command Window] に次のように入力します。

```
>> demo blockset xilinx
```

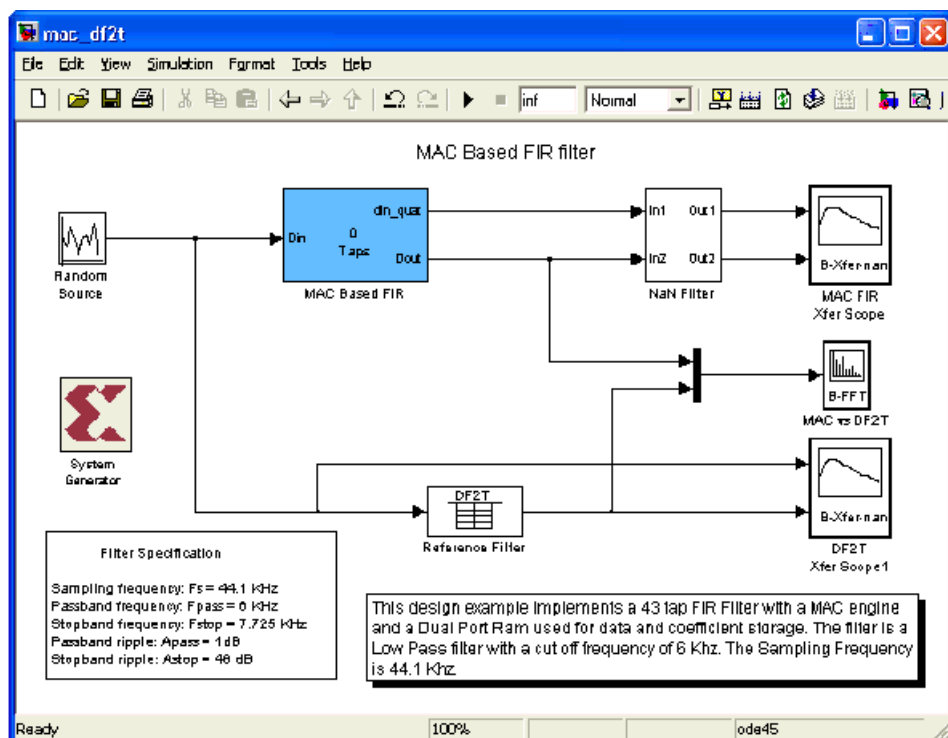
表示されたウィンドウで、デモ デザインのリストから [Polyphase 1:8 filter using SRL16Es] をクリックします。

デザインの概要

このデザインでは、[Signal Processing Blockset] → [Signal Processing Source] にある Random Source ブロックを使用し、FIR フィルタの 2 つのインプリメンテーションを駆動します。

- 最初のフィルタは、ザイリンクス デバイスにインプリメントします。このフィルタは、デュアルポートブロックメモリと MAC を使用してインプリメントされた、固定小数点 FIR フィルタです。
- 2 つ目のフィルタは Reference Filter で、倍精度の直接型 II 転置フィルタです。

各フィルタの周波数応答は、伝達関数スコープで表示されます。



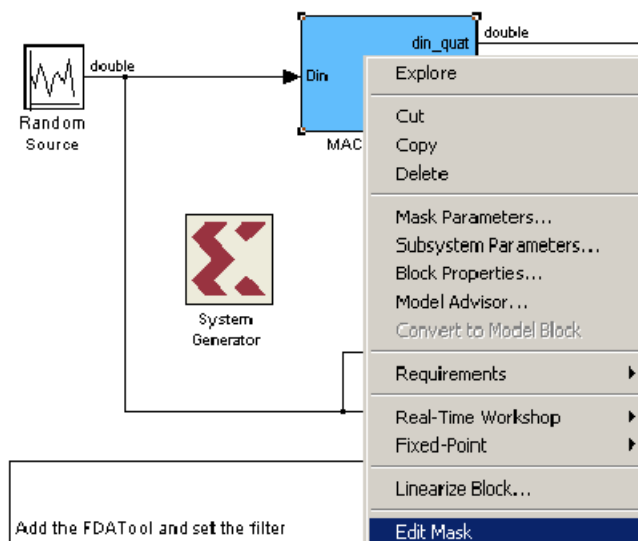
FIR フィルタの係数の生成

- MATLAB の [Command Window] で、cd コマンドを使用して <path_to_sysgen>\examples\mac_fir ディレクトリに移動します。
- [Command Window] に「mac_df2t」と入力し、デザイン モデルを開きます。

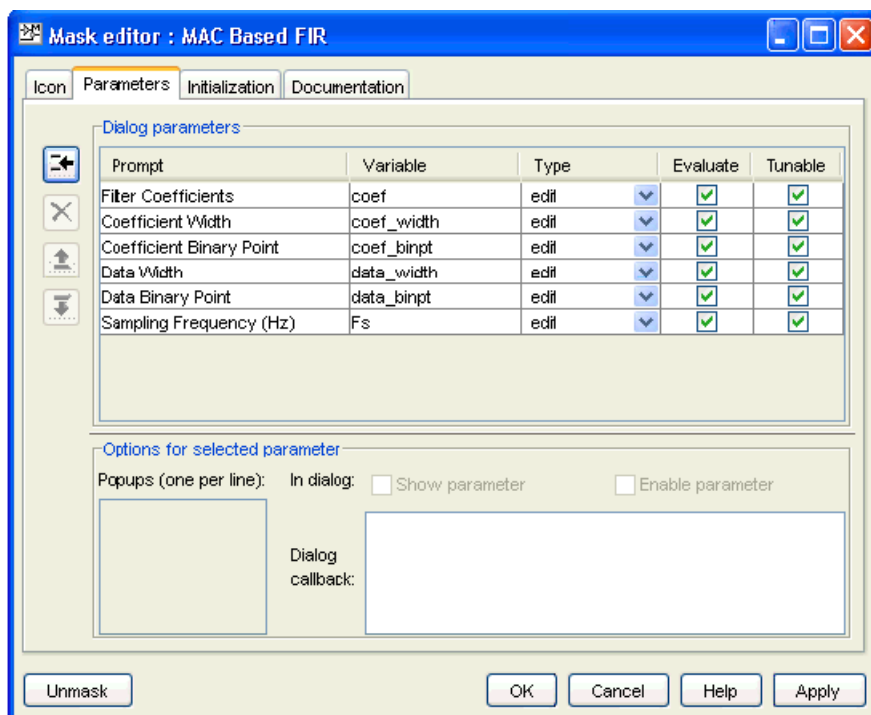
このチュートリアル の目的のため、変数 coef、coef_width、coef_binpt、data_width、data_binpt、および Fs は定義されていません。これらの変数は、MAC Based FIR ブロックのマスク パラメータとして使用し、FDATool を使用してフィルタを設計し、割り当てます。完全に機能するモデルが、同じディレクトリに mac_df2t_soln.mdl という名前で含まれています。

MAC Based FIR ブロックのパラメータ指定

1. MAC Based FIR ブロックを右クリックし、[マスクの編集] をクリックします。

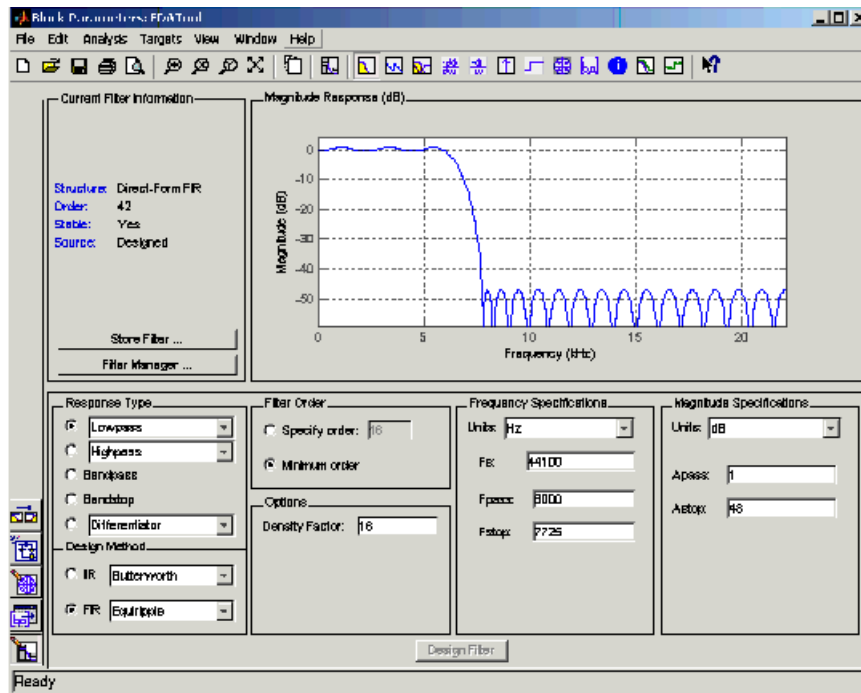


2. [マスクエディタ] ダイアログ ボックスの [パラメータ] タブで、次の図に示すように coef、data_width、および data_binpt を追加します。



FIR フィルタの係数の生成と割り当て

1. [Xilinx Blockset] → [DSP] にある FDATool ブロックをドラッグしてモデルに追加します。
2. FDATool ブロックをダブルクリックし、オーディオ システムで高周波数ノイズを除去するローパス フィルタの次の仕様を入力します。
 - ◆ [応答タイプ]: [ローパス]
 - ◆ [フィルタ次数]: [最小次数]
 - ◆ [周波数仕様]
 - [単位]: [Hz]
 - [Fs]: 44100
 - [Fpass]: 6000
 - [Fstop]: 7725
 - ◆ [振幅仕様]
 - [単位]: [dB]
 - [Apass]: 1
 - [Astop]: 48



3. [フィルタの設計] をクリックし、フィルタ次数と振幅応答を確認します。
 ツールバー ボタンをクリックして、位相応答、インパルス応答、フィルタ係数なども表示できます。上記の仕様を満たすには、43 タップ FIR フィルタ (次数 0 ~ 42) が必要であることが示されます。

フィルタ係数を表示するには、MATLAB の [Command Window] で次のように入力します。

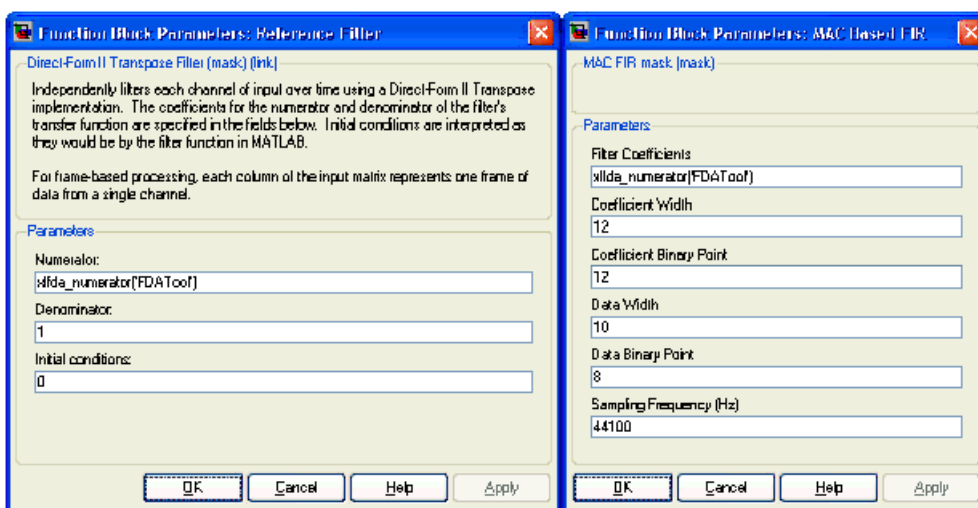
```
>> xlfda_numerator('FDATool')
```

次のコマンドを使用すると、係数幅と 2 進小数点を正しく指定するための最大係数値および最小係数値を知ることができます。

```
>> max(xlfsda_numerator('FDATool'))
>> min(xlfsda_numerator('FDATool'))
```

このチュートリアルでは、係数タイプは Fix_12_12 (2 進小数点が 12 番目のビットの左側にある 12 ビット 値) に設定されています。max() 関数の結果、最大係数は 0.3022 であり、2 進小数点を最上位ビットの左側に配置できます。これは、Fix_12_12 の値の範囲が -0.5 ~ 0.4998 であり、2 進小数点を最上位ビットの左側に配置することで、ダイナミック範囲を最大限にできるからです。2 進小数点を右に移動すると (Fix_12_11 値を使用)、Fix_12_11 値の範囲が -1 ~ 0.9995 で係数を表現するのに必要以上あるため、ダイナミック範囲が 1 ビット失われます。

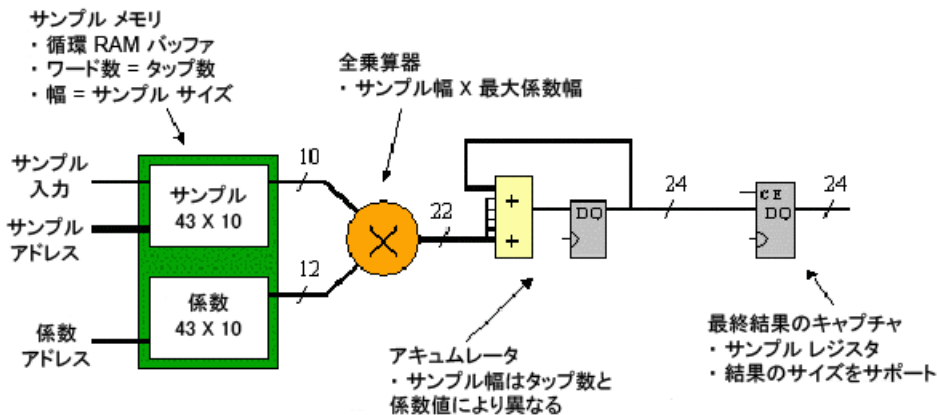
4. Reference Filter ブロックと MAC Based FIR ブロックのパラメータ ダイアログ ボックスで、coef、coef_width、coef_binpt、data_width、data_binpt、Fs のパラメータ値が次のようになっていることを確認します。



それぞれのダイアログ ボックスで [OK] をクリックします。

ザイリンクス フィルタ ブロックの理解

次に、このチュートリアルの MAC Based FIR をインプリメントしたブロック図を示します。



この時点では、MAC Based FIR は 10 ビットの符号付き入力データ (Fix_10_8)、12 ビットの符号付き係数 (Fix_12_12)、43 タップに設定されています。これらのパラメータは、MAC Based FIR ブロックのパラメータ ダイアログ ボックスで直接変更できます。係数とデータは、メモリ システムに保存する必要があります。このチュートリアルでは、データと係数の保存にデュアル ポート メモリを使用します。データのキャプチャおよび読み出しには、循環 RAM バッファを使用します。RAM は混合モード コンフィギュレーションで使用され、ポート A で値の書き込みおよび読み出しが行われ (RAM モード)、ポート B で係数の読み出しのみが行われます (ROM モード)。

乗算器は、最速のパフォーマンスを達成するため、ザイリンクス Virtex デバイスに含まれているエンベデッド乗算器リソースを使用し、レイテンシが 3 になるよう設定されています。乗算器およびアキュムレータに必要な精度は、フィルタ タップ (係数) とタップ数の関数です。これらの数値は設計時に固定されるので、フィルタ仕様を満たすようハードウェア リソースを調整できます。アキュムレータの精度は、フィルタ タップに対して最大入力を累積するのに十分であればよく、次の式で算出されます。

$$\text{acc_nbits} = \text{ceil}(\log_2(\text{sum}(\text{abs}(\text{coef} * 2^{\text{coef_width_bp}}))) + \text{data_width} + 1;$$

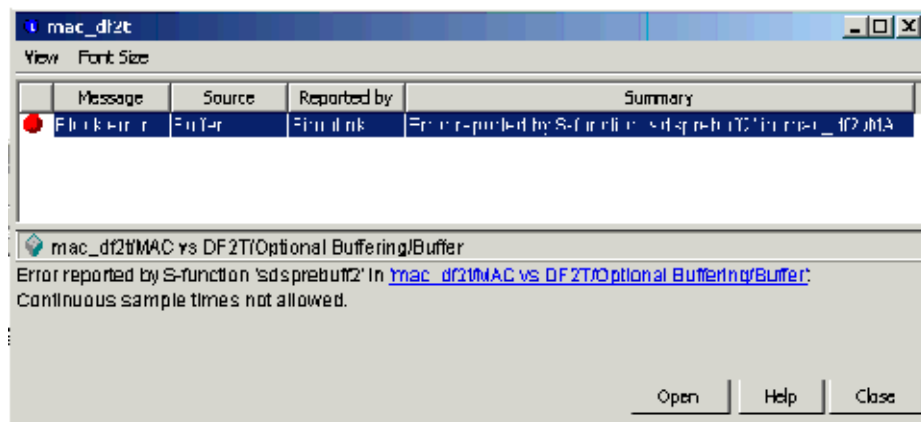
アキュムレータをリセットすると、0 ではなく現在の入力値に初期化されるので、MAC エンジンでデータを停止することなく送信し続けることができます。MAC エンジンは、出力サンプルの最後の部分積を算出した後アキュムレータに入力サンプルを読み込むので、ストリーミング操作ではキャプチャレジスタが必要です。

最後に、Down Sample ブロックでキャプチャレジスタのサンプリング周期を出力サンプリング周期に変換します。効率的なハードウェア インプリメンテーションを達成するため、このブロックはレイテンシを持つようコンフィギュレーションされます。サンプリング レートは、係数アレイの長さと同じです。

シミュレーションの実行

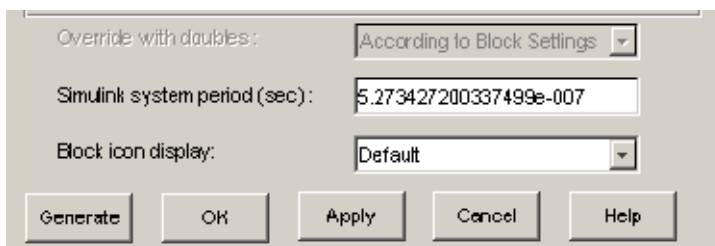
1. シミュレーション時間を 0.05 に変更し、シミュレーションを実行します。

次の図に示すようなメッセージが表示されます。



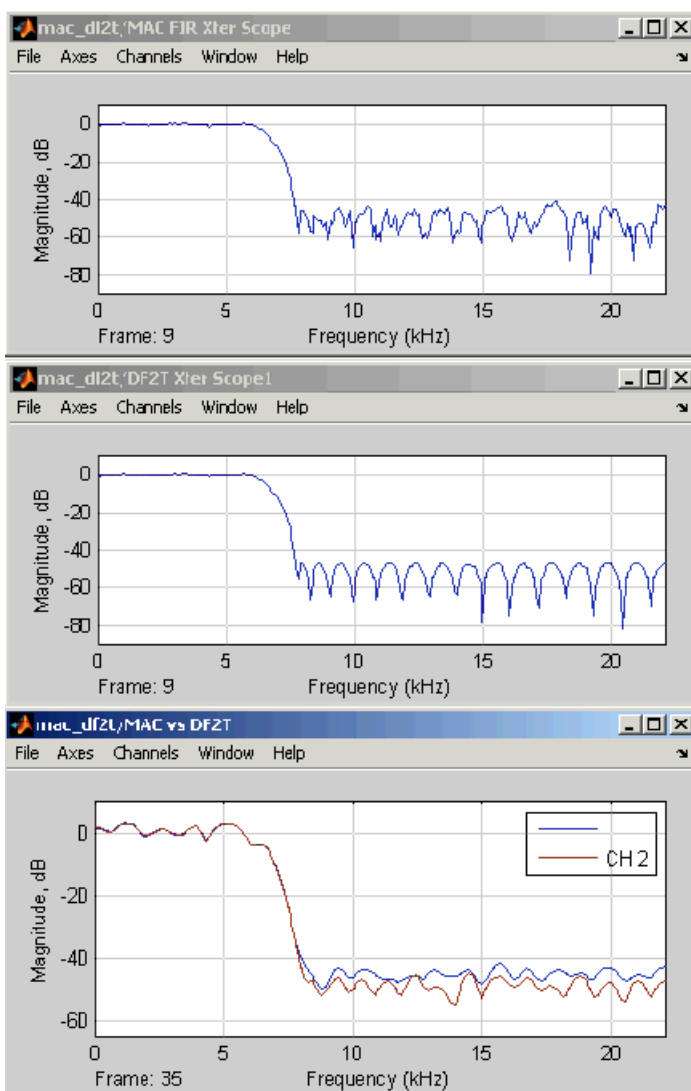
System Generator では、din という Gateway In ブロックから入力サンプリング周期を取得しますが、この周期は 1/Fs に設定されています。MAC Based FIR フィルタは、タップ数に応じてオーバーサンプリングされるので、システム クロック周期は常に 1 (フィルタ タップ数 × Fs) です。

2. System Generator トークンをダブルクリックし、[Simulink system period] を $5.273427\text{e-}007 = 1/(43 * 44100)$ に設定します。



3. ミュレーションをもう 1 度実行します。MAC Based FIR フィルタのザイリンクス インプリメンテーションは元のフィルタの仕様を満たしており、周波数応答は倍精度の Simulink モデルとほぼ同一です。

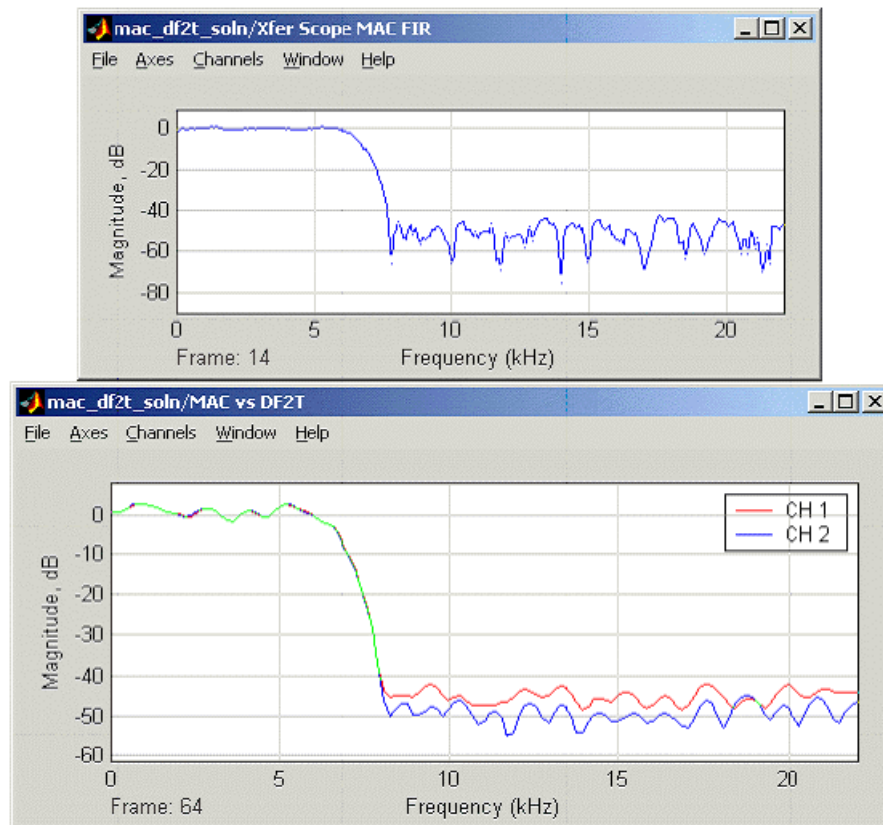
フィルタのパスバンド応答と 0 が明確に示されています。次の図のような周波数応答が得られるはずです。



フィルタの精度を上げたり下げたりすることにより、デザインの仕様に必要な範囲、パフォーマンス、質のバランスを取ることができます。

シミュレーションを停止し、ブロックのパラメータ ダイアログ ボックスで係数幅を `FIX_10_10`、データ幅を `FIX_8_6` に変更します。Ctrl + D を押してモデルをアップデートし、MAC engine ブロックの階層を表示します。データパスが、乗算器の出力では 18 ビットに、アキュムレータでは 20 ビットにアップデートされています。

シミュレーションを再開し、周波数応答がどのように変化したかを観察します。固定ワード長により、減衰が悪化しています (40dB 未満)。



複数クロックのサイクル単位アイランドの生成

System Generator の共有メモリ インターフェイスを使用すると、複数のクロック ソースで駆動するデザインをインプリメントできます。複数クロック デザインでは、異なるクロックおよび派生クロックを使用して、1 つのデザイン環境で高度なクロック供給方法をインプリメントできます。このセクションでは、System Generator で複数クロック デザインをインプリメントする方法を、次の内容とおして説明します。

- 複数クロックを使用することが利点となるデザイン
- 階層を使用して System Generator モデルを複数のクロック ドメインに分割
- 共有メモリを使用したクロック ドメインの切り替え
- 複数クロック デザインのシミュレーションおよびネットリスト作成
- ザイリンクス [Multiple Subsystem Generator](#) ブロックを使用した複数のクロック ドメインの配線

これらの内容を、例を使用して説明します。この例では 2 つのクロックを使用していますが、ここに示す概念は、クロック ソースがいくつかのデザインにも応用できます。

まず、System Generator での標準のクロック供給手法およびインプリメンテーション手法を理解しておくことが有益です。これらの情報は、「[タイミングとクロック](#)」で詳細に説明されています。通常 System Generator デザインは 1 つのシステム クロック ソースで駆動し、複数のレートはシステム クロック ソースにクロック イネーブルを使用することにより生成しますが、複数のクロック ソースで駆動するデザインをインプリメントすることも可能です。

大まかに言うと、次の手法が使用されます。

デザインを複数のサブシステムに分割し、それぞれを異なるクロックで駆動します。この例では、これらのサブシステムを「非同期クロック アイランド」と呼びます。ザイリンクスの共有メモリ ブロックを、クロック アイランド間のブリッジとして使用します。デザインを分割したら、ザイリンクス Multiple Subsystem Generator ブロックを使用して、デザインを複数のクロック ソースを使用するハードウェアに変換できます。

複数クロック アプリケーション

複数のクロック ドメインは、異なるクロック レートで動作する複数の外部ハードウェアのインターフェイスによく使用されます。たとえば、マイクロプロセッサに複数の I/O レジスタを接続し、個別のクロックに同期するこれらのレジスタに対してマイクロプロセッサで読み出しおよび書き込みを実行する場合や、クロック/データ リカバリ ユニットのデータを取得し、このデータをローカル クロック ドメインに再同期化する場合がある場合、システム クロックとは異なるサンプリング レートで動作する必要がある DA コンバータにデータを供給する場合があります。

複数クロック ドメインのその他の重要なアプリケーションとして、高速処理ユニットがあります。たとえば、補間 FIR フィルタを考えてみます。このフィルタは、外部ユニットからシンボル データを取得し、4X の補間を実行して各入力シンボルに 4 つの出力サンプルを作成します。出力サンプルは、サンプリング レートで動作する DA コンバータ (DAC) に供給されます。

FIR フィルタは、複数のレートのいずれかで動作させることができます。シンボル レートで動作させる場合、各サイクルで 4 つのサンプルを作成し、これらのサンプルをサンプリング レートで DAC に供給します。このパラレル インプリメンテーションでは、ハードウェア リソースが多く必要となり、サンプリング レートが非常に高速な場合にのみ採用されます。別の方法としては、FIR フィルタをサンプリング レートで動作させ、サイクルごとに 1 つのサンプルを作成します。この場合、ハードウェア リソースの使用量は中程度であり、中速のサンプリング レートで使用されます。サンプリング レートが低速である場合は、FIR フィルタをサンプリング レートの数倍で動作させることもできます (DCM を使用してサンプリング レート クロックを逡倍)。このようにすると、FIR フィルタの MAC ユニットの各サンプル出力の計算で複数回使用でき、ハードウェア リソースの量を最小限に抑えることができます。この最後の方法では、シンボル レート クロック ドメイン、高速処理クロック ドメイン、およびサンプリング レート クロック ドメインを使用します。

FPGA の設計では、FPGA デバイスの各リソースをできるだけ高速に動作させ、ハードウェアの使用量を最適化するようにすることが推奨されます。通常は、可能な限り 1 つのクロック ドメインを使用し、クロック イネーブルを使用して低速の回路へのクロック供給を調整し、複数サイクル パスを作成します。この方法の欠点は、消費電力が増加し、高速クロック イネーブルを配線するのが困難な場合があることです。そのため、場合によっては、高速処理用のドメインを別に作成する方が適切です。また、非同期データ入力および出力を使用する場合は、複数のクロック ドメインを使用せざるを得ない場合があります。

クロック ドメインの分割

FPGA デザインでは、複数クロック デザインを複数のドメインに分割することが重要です。System Generator では、デザイン階層を使用してクロック ドメインの分割をサポートしています。具体的には、デザインで複数のクロック ドメインを使用する場合、各クロック ドメインに関連するロジックを Simulink サブシステムとしてグループ化します。

サブシステム (この場合は同期アイランド) はサイクル単位であり、アイランドに生成されるハードウェアはアイランド モデルの Simulink の動作に忠実です。ビット精度およびサイクル精度は、各同期アイランド内でのみ保持されます。同期アイランドを含む最終的なデザインは、アイランドを非同期クロックで駆動するので、サイクル単位とは限りません。System Generator および Simulink では、理想的なクロック ソースを使用してデザインをシミュレーションできますが、非同期クロック供給システムは複雑であるため、ソフトウェア シミュレーションとハードウェア シミュレーションの結果が異なるものになる場合があります。

サブシステムを使用したデザインの分割には、次のような利点があります。

- 物理クロック ラインがブロック図から省略されます。
- クロック ドメインが切り替わる転送が明確に定義され、ザイリンクス ブロックセットに含まれるメタステーブルにならないブロックで処理できます。
- ドメインが明確に定義されるので、System Generator で同期アイランドのタイミング制約を正確に生成できます。

System Generator の抽象度により、一般的なデザイン エラーを犯す可能性が減少します。これらのデザイン エラーには、次のようなものがあります。

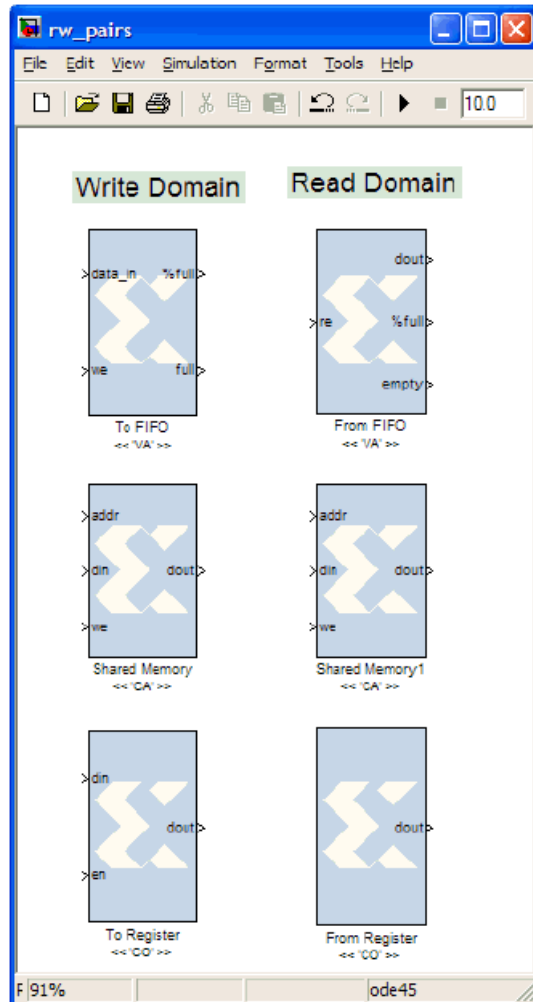
- ゲーティッド クロック : System Generator のクロックは、ハードウェア生成中に推論されるので、クロック以外のラインをクロック入力に接続する (ゲーティッド クロック) のは不可能です。
- 非同期クリア : System Generator の非同期リセットはハードウェア生成中に推論されるので、非同期リセットを使用して同期ロジックをクリアするよう指定することはできません。このように指定すると、多くの場合タイミングの問題が発生します。
- ラッチの推論 : System Generator デザインからラッチは生成されません。

クロック ドメインの切り替え

クロック ドメインが切り替わる場合は、System Generator の共有メモリブロックを使用する必要があります。クロック ドメインが切り替わるデータ転送用にいくつかのブロックが用意されており、[Xilinx Blockset] → [Shared Memory] に含まれています。

- [Shared Memory](#)
- [To FIFO / From FIFO](#)
- [To Register / From Register](#)

クロック ドメインの切り替えにこれらの共有メモリ ブロックを使用する際は、ペアになっているブロックと対にして使用します。



To FIFO ブロックはこのブロックに書き込みを行うドメインに配置し、From FIFO ブロックはこのブロックから読み出しを行うドメインに配置します。同じ名前の 2 つのブロックは、リンクされます。FIFO は、ザイリンクス FIFO Generator コアを使用してハードウェアにインプリメントされます。クロック ドメインの切り替えには、FIFO ブロックを使用するのが最も確実で簡単であり、バンド幅の広いシーケンシャル データ転送に最適です。

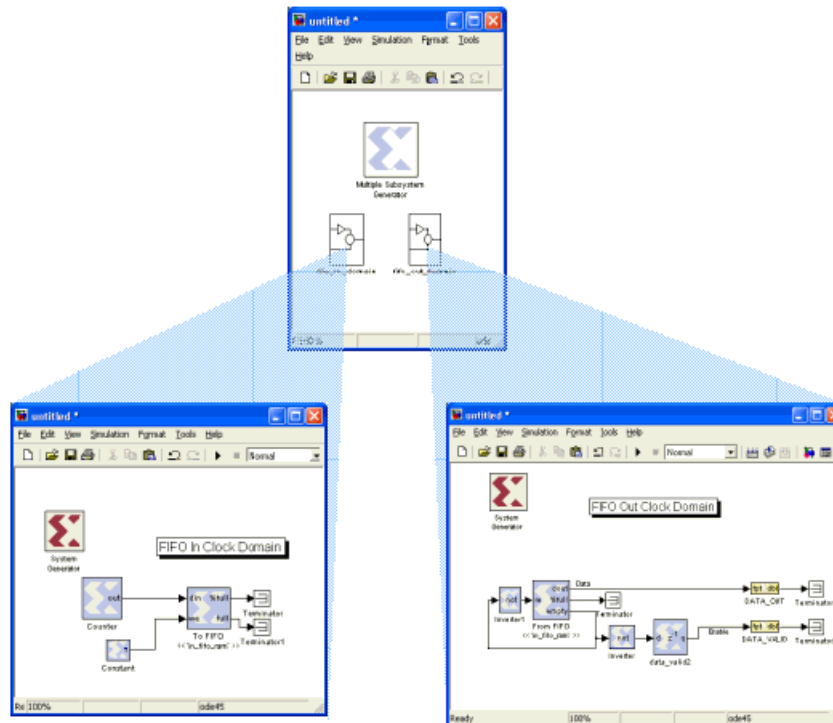
Shared Memory ブロックのペアは、ザイリンクス Dual-Port Block RAM コアとしてインプリメントされます。同じ名前の Shared Memory ブロックはリンクされます。ペアとなっている 2 つのブロックは、異なるドメインに配置されます。RAM はデュアル ポートであるので、各ドメインで RAM に書き込みを実行できます。セマフォやロジックを使用して、2 つの書き込みまたは読み出しと書き込みが、同じアドレスに対して同時に行われなくする必要があります。たとえば、ドメイン B が読み出しを行っているメモリ ロケーションに対して同時にドメイン A で書き込みを行うと、データ読み出しが無効になる可能性があります。共有メモリはザイリンクス Dual Port Block Memory コアを使用してインプリメントされ、大型メモリが複数の BRAM に効率的にマップされます。

To Register ブロックはこのブロックに書き込みを行うドメインに配置し、**From Register** ブロックはこのブロックから読み出しを行うドメインに配置します。同じ名前のブロックはリンクされます。**To Register** ブロックは、そのドメインから同時に読み出すことができます。レジスタの幅は可変にすることができ、フリップフロップとして合成されます。1 ビットの **To Register/From Register** ペアは、1 つのフリップフロップとして合成されます。

メモ：この方法でドメインを切り替えるのは安全でない可能性があり、複数ビットの転送にはメタステーブル状態を削減する同期化フリップフロップおよびセマフォを使用する必要があります。この手法は、ハードウェアで犯しやすいエラーをよく理解した上で使用してください。

複数クロック デザインのネットリスト生成

各クロックドメインは、**System Generator** デザイン内の 1 つのサブシステムとする必要があります。次の図に、2 つのドメインを含むデザインを示します。最上位ブロックには、**Multiple Subsystem Generator** ブロックと、それぞれクロックドメインを構成する 2 つのサブシステムが含まれます。各サブシステムには、そのクロックドメインのシステムクロック周期を設定する **System Generator** トークンが含まれます。

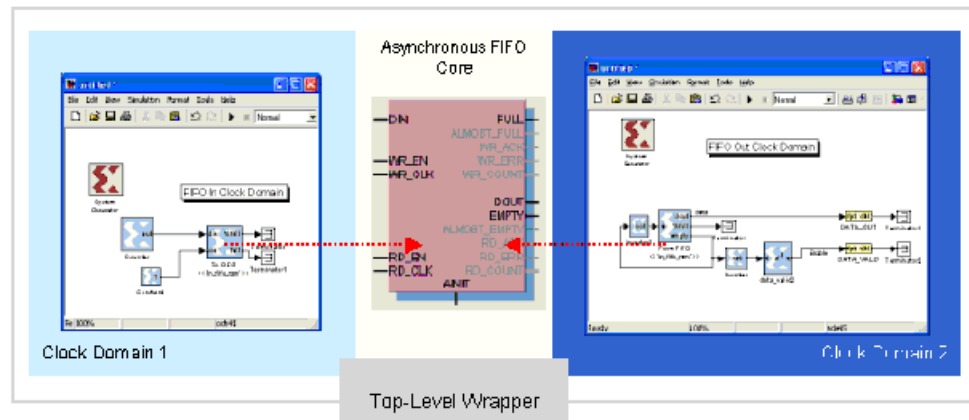


メモ： **Multiple Subsystem Generator** ブロックでは、**EDK Processor** ブロックを含むデザインはサポートされません。

次の図は、サブシステムにドメインを切り替えるブロックを配置する概念を示します。複数のドメインを含むデザインのネットリストを生成すると、**System Generator** で次の処理が実行されます。

- **To FIFO** ブロックを除外したドメイン 0 (左側) の HDL ファイルを作成し、ネットリストを呼び出してブラックボックス ネットリスト (NGC ファイル) を生成します。
- **From FIFO** ブロックを除外したドメイン 1 (右側) の HDL ファイルを作成し、ネットリストを呼び出してブラックボックス ネットリスト (NGC ファイル) を生成します。
- **CORE Generator** を起動し、**FIFO** ブロック (中央) のコアを生成します。

- これら 3 つのブロック コンポーネントをインスタンス化する最上位 HDL ラップを作成します。



手順の例

この例では、デザイン階層を使用して System Generator デザインを複数の非同期クロック アイランドに分割する方法、Shared Memory ブロックをアイランド間の通信に使用する方法、および Multiple Subsystem Generator ブロックを使用して複数クロック デザインのネットリストを生成する方法を示します。

1. MATLAB ウィンドウで、次のディレクトリに移動します。
`<path_to_sysgen>\examples\multiple_clocks\`
2. two_async_clks フォルダにある two_async_clks.mdl を開き、一時的なディレクトリに保存します。

この例ではサブシステム階層を使用してデザインを 2 つの同期クロック ドメイン (A と B) に分割しています。これらのクロック ドメインはそれぞれ 1 つのクロックに同期していますが、ドメイン同士は非同期です。デザインには ss_clk_domainA および ss_clk_domainB という 2 つのサブシステムがあり、それぞれクロック ドメイン (A と B) に関連するロジックが含まれています。ss_clk_domainA サブシステムに含まれるブロックはクロック ドメイン A で動作し、ss_clk_domainB サブシステムに含まれるブロックはクロック ドメイン B で動作します。

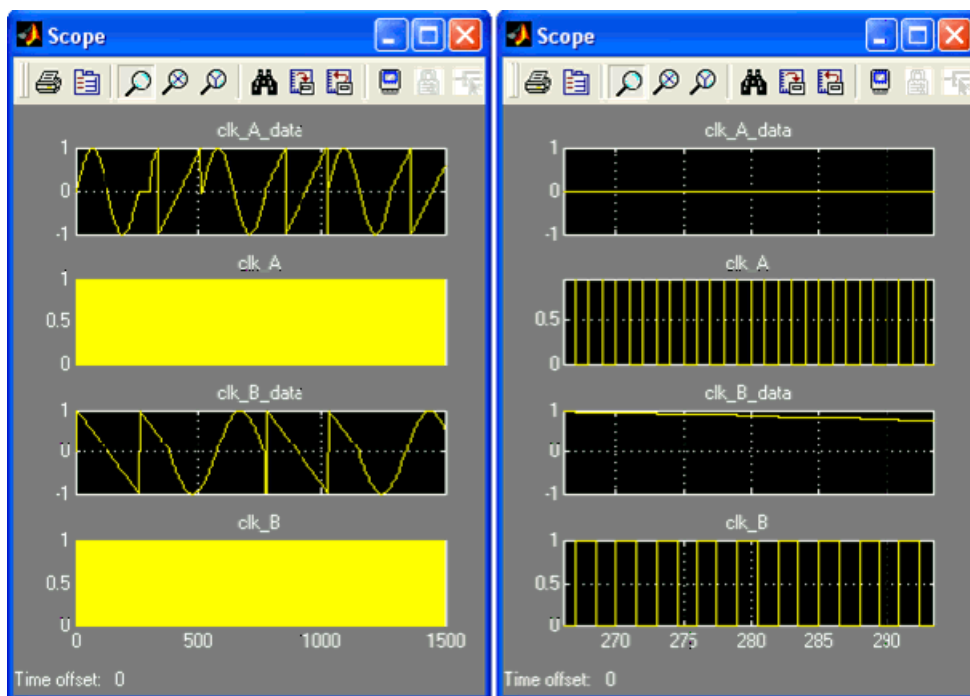
これら 2 つの非同期サブシステムは、Shared Memory ブロックのペアを使用してインプリメントされた共有メモリ インターフェイスを介して通信します。Shared Memory ブロックの 1 つは ss_clk_domainA 内に、もう 1 つは ss_clk_domainB 内に配置されています。この 2 つのブロックには、bram_iface という同じ名前が付けられています。このようにすることで、シミュレーション中に両方の Shared Memory ブロックで同じアドレス空間にアクセスできます。ダイアグラムでは、2 つのブロックの間に物理的な接続は示されていません。これは、2 つの Shared Memory ブロックの接続が、同じオブジェクト名を指定することにより、アドレス空間を共有するように暗示的に定義されているからです。2 つのサブシステムがワイヤで接続され、ハードウェアに変換されると、2 つの Shared Memory ブロックはサブシステムから移動され、1 つのブロック RAM に統合されます。詳細は、「Multiple Subsystem Generator」を参照してください。

同期アイランドは、異なる入力ソースをサンプリングします。サブシステム `ss_clk_domainA` では `sinusoid` 入力がサンプリングされ、`ss_clk_domainB` ではのこぎり波がサンプリングされます。各サブシステムは、サンプルをもう 1 方の共有メモリに書き込みます。メモリがフルになると、もう 1 方のメモリからサンプルを読み出します。デザインをシミュレーションすると、モデルの動作を可視化できます。

3. **Simulink** モデルのツールバーで [シミュレーションの開始] をクリックし、デザインのシミュレーションを実行します。

4. **Scope** ブロックを開いて、出力信号を表示します。

Scope ダイアログボックスには、2 つのクロック `clk_A` および `clk_B` が表示されます。デフォルトの尺度では、2 つのクロックを区別するのは困難です。拡大表示して、詳細が確認できるようにします。

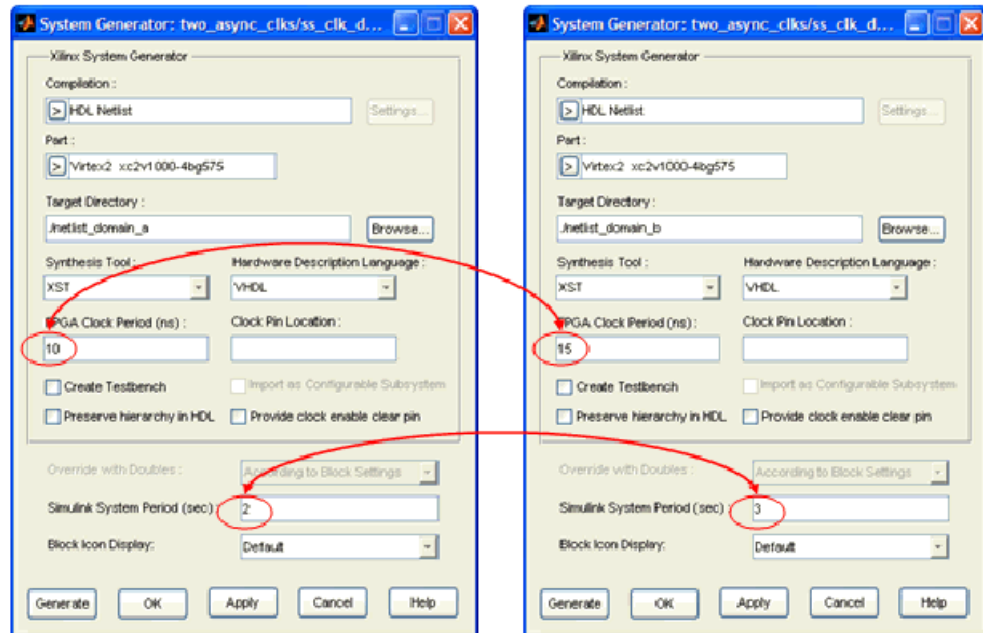


`clk_A` と `clk_B` は、周期が異なり、位相が揃っていません。**System Generator** では 1 つのデザインに 1 つのクロック ソースが使用されると説明しましたが、ここでは 2 つの異なるクロックが使用されています。

これは、デザインが階層構造になっているからです。すべてのブロックは、サブシステムを使用した階層レベルに配置されています。最上位に **System Generator** トークンが含まれていないため、各サブシステムは完全に個別の **System Generator** デザインであると考えることができます。このモデルでは、`ss_clk_domainA` と `ss_clk_domainB` サブシステムに異なる **Simulink** システム周期を設定して、2 つのクロック ドメインを定義しています。サブシステムを異なる **System Generator** デザインとして扱っているため、これが可能です。`ss_clk_domainA` と `ss_clk_domainB` サブシステム内のクロック プローブは、それぞれの **Simulink** システム周期を使用して出力が決定されるので、異なるシステム周期により異なるシステム クロックが得られます。

ここで、`ss_clk_domainA` および `ss_clk_domainB` サブシステムの **System Generator** トークンで定義されているクロックを確認します。

5. ss_clk_domainA および ss_clk_domainB サブシステムに含まれる System Generator トークンのパラメータ ダイアログ ボックスを開きます。



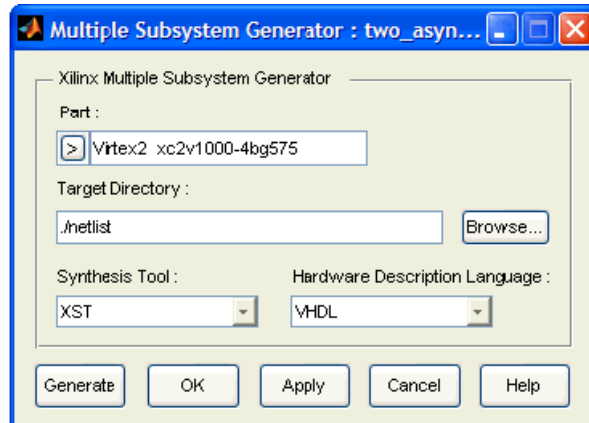
ss_clk_domainA サブシステムの FPGA クロック周期は 10ns (周波数 100MHz) です。モデル内のサンプリング周期の値を単純にするため、10ns クロックは Simulink システム周期の値 2s に正規化されています。ss_clk_domainB サブシステムの FPGA クロック周期は 15ns (周波数 66.7MHz) です。このクロック周期を正規化すると、Simulink システム周期 3s になります。

この例の 2 つのサブシステムでは複数の同期した System Generator ドメインをインプリメントするので、Multiple Subsystem Generator ブロックを使用してサブシステムを接続し、2 つのクロックポートを持つ 1 つの HDL 最上位コンポーネントを作成します。Multiple Subsystem Generator ブロックがデザインをハードウェアに変換する際、各サブシステムに対してそれぞれ NGC ネットリストファイルが生成されます。サブシステム ネットリスト ファイルをブラック ボックスとしてインスタンス化して最上位 VHDL コンポーネントまたは Verilog モジュールも生成し、共有メモリコアをクロック ドメインブリッジとして使用して 2 つを接続します。

Multiple Subsystem Generator ブロックを使用して、ss_clk_domainA および ss_clk_domainB サブシステムのネットリストを生成します。

6. two_async_clks モデルの最上位に含まれている Multiple Subsystem Generator ブロックをダブルクリックして、[Multiple Subsystem Generator] ダイアログ ボックスを開きます。
7. [Target Directory] でターゲット ディレクトリを指定します。デフォルトのディレクトリは、netlist です。

8. [Generate] ボタンをクリックします。[Part]、[Synthesis Tool]、[Hardware Description Language] は、そのままにします。



Multiple Subsystem Generator の実行が終了すると、生成が完了したことを示すダイアログ ボックスが表示されます。生成された結果を確認します。

9. cd コマンドを使用してデザインのターゲット ディレクトリ netlist に移動します。

このディレクトリには、2つの NGC ファイル `ss_clk_domaina_cw.ngc` と `ss_clk_domainb_cw.ngc` があります。これらのファイルは、それぞれ `ss_clk_domainA` および `ss_clk_domainB` サブシステムのネットリストおよび制約情報を含みます。これらの NGC ファイルには、そのサブシステムに関連するクロック ラップレイヤ ロジックが含まれています。マルチレート デザインに必要なクロック イネーブル ロジックが含まれていることを確認する必要があります。デザインのクロック ラップレイヤを使用することにより、対応するクロック ドライバ コンポーネントが自動的にネットリストに含まれます。

このディレクトリには、`dual_port_block_memory_virtex2_6_1_ef64ec122427b7be.edn` という名前のデュアル ポート コアのネットリスト ファイルも含まれます。このコアは、元のデザインで使用されている Shared Memory ブロックのハードウェア インプリメンテーションです。このメモリの幅とワード数は、Shared Memory ブロックの設定値に基づきます。

ここで、Multiple Subsystem Generator で生成された最上位 HDL コンポーネントを確認します。

10. テキスト エディタで `two_async_clks.vhd` ファイルを開きます。

このファイルでは、`two_async_clks` モデルの最上位 HDL が定義されています。

```
entity two_async_clks is
  port (
    din_a: in std_logic_vector(7 downto 0);
    din_b: in std_logic_vector(7 downto 0);
    ss_clk_domaina_cw_ce: in std_logic := '1';
    ss_clk_domaina_cw_clk: in std_logic;
    ss_clk_domainb_cw_ce: in std_logic := '1';
    ss_clk_domainb_cw_clk: in std_logic;
    dout_a: out std_logic_vector(7 downto 0);
    dout_b: out std_logic_vector(7 downto 0)
  );
end two_async_clks;
```

ポート インターフェイスに関して、特出すべきことがいくつかあります。まず、2つのクロック ポートがあります。これらのクロック ポートは、サブシステム名に基づいて名前が付けられており (ss_clk_domaina など)、対応するサブシステム NGC ネットリスト ファイルに接続されています。また、各サブシステムの最上位ポート (din_a、dout_a など) がポート インターフェイスに含まれています。

Multiple Subsystem Generator は、複数のクロック ソースを生成する回路 (DCM など) は生成しません。最上位 HDL を変更してクロック回路を含めるか、最上位 HDL をコンポーネントとしてクロック回路を含む別のラッパにインスタンス化します。

最上位ラッパの作成

複数クロック System Generator デザインの最上位ラッパを作成する場合は、最低でも次のタスクを実行する必要があります。

- System Generator 最上位コンポーネントをその他のラッパ ロジック (DCM など) と共にインスタンス化します。
- System Generator コンポーネントをその他のロジックに接続します。
- System Generator コンポーネントの最上位ポート マップの代わりとなる新しい最上位ポート マップを作成します。

次に、クロック回路をインスタンス化する最上位 HDL の例を示します。この例では、前のセクションで Multiple Subsystem Generator ブロックで生成した出力を使用します。System Generator デザインは two_async_clks、最上位 HDL は top_wrapper です (VHDL 合成の場合)。

クロック ラインおよびクロック イネーブルは推論されるので、クロックとクロック イネーブルの名前はそのクロックが推論されたサブシステムの名前に接頭辞 (_clk と _ce) を付けたものになっています。dout_a などその他のポート名は、System Generator デザインの Gateway ブロックで指定された名前が使用されます。

次に、エンティティ two_async_clks をインスタンス化する VHDL 最上位ラッパの例を示します (明確にするため削除されている部分もあり)。このラッパでは、DCM コンポーネントを使用して System Generator デザインに必要な 2 つのクロックを生成しています。

```
-----
-- top_wrapper.vhd
-- Example Top Level Wrapper
--
-- This is an example top-level wrapper for instantiating a System
Generator
-- design along with a DCM. In this example, the DCM connects the two
clock
-- inputs of the System Generator block ('two_async_clks') to two
buffered
-- outputs of the DCM, namely, CLK0 and CLKFX. CLK0 is the same
frequency
-- and phase as the input clock, and CLKFX is configured to be twice the
-- frequency of the input clock.
-----

library IEEE;
library unisim;
use IEEE.std_logic_1164.all;
use unisim.vcomponents.all;
entity top_wrapper is
```



```

port (
    clk : in std_logic;
    din_a : in std_logic_vector(7 downto 0);
    din_b : in std_logic_vector(7 downto 0);
    dout_a : out std_logic_vector(7 downto 0);
    dout_b : out std_logic_vector(7 downto 0)
);
end top_wrapper;

architecture structural of top_wrapper is
    -----
    -- SysGen Model Component Declaration
    -----
    component two_async_clks
    port (
        din_a: in std_logic_vector(7 downto 0);
        din_b: in std_logic_vector(7 downto 0);
        ss_clk_domaina_cw_ce: in std_logic := '1';
        ss_clk_domaina_cw_clk: in std_logic;
        ss_clk_domainb_cw_ce: in std_logic := '1';
        ss_clk_domainb_cw_clk: in std_logic;
        dout_a: out std_logic_vector(7 downto 0);
        dout_b: out std_logic_vector(7 downto 0)
    );
end component;
component bufg
port(i: in std_logic;
    o: out std_logic);
end component;
    -----
    -- DCM Component Declaration
    -----
    component dcm
    -- synopsys translate_off
    generic (clkout_phase_shift : string := "fixed";
        dll_frequency_mode : string := "low";
        duty_cycle_correction : boolean := true;
        clkdv_divide : real := 3;
        clkfx_multiply : integer := 2;
        clkfx_divide : integer := 1);
    -- synopsys translate_on
    port (clkin : in std_logic;
        clkfb : in std_logic;
        dssen : in std_logic;
        psincdec : in std_logic;
        psen : in std_logic;
        pscclk : in std_logic;
        rst : in std_logic;
        clk0 : out std_logic;
        clk90 : out std_logic;
        clk180 : out std_logic;
        clk270 : out std_logic;
        clk2x : out std_logic;
        clk2x180 : out std_logic;
        clkdv : out std_logic;
        clkfx : out std_logic;
        clkfx180 : out std_logic;
        locked : out std_logic;
        psdone : out std_logic;
    );
end component;

```



```

        status : out std_ulogic_vector(7 downto 0));
end component;
-----
-- DCM Attributes
-----
attribute dll_frequency_mode : string;
attribute duty_cycle_correction : string;
attribute startup_wait : string;
attribute clkdv_divide : string;
attribute clkfx_multiply : string;
attribute clkfx_divide : string;
attribute clkin_period : string;

attribute duty_cycle_correction of dcm0 : label is "true";
attribute startup_wait of dcm0 : label is "false";
attribute dll_frequency_mode of dcm0 : label is "low";
attribute clkdv_divide of dcm0 : label is "3";
attribute clkfx_multiply of dcm0 : label is "2";
attribute clkfx_divide of dcm0 : label is "1";
attribute clkin_period of dcm0 : label is "10";

signal clk0unbuf : std_logic;
signal clk0buf : std_logic;
signal clkfxbuf : std_logic;
signal clk2xunbuf : std_logic;
signal clkfxunbuf : std_logic;
signal clkdvunbuf : std_logic;
signal clkdvbuf : std_logic;
signal ff1,ff2,ff3,ff4 : std_logic;
signal dcm_rst : std_logic;
signal intlock : std_logic;

-----
-----
-- The top level instantiates the SysGen design, a DCM, and two BUFs.
-- The DCM generates two clocks of different frequencies.
-- These two clocks are used to drive the two different clock domains
-- in the SysGen block.
-----
-----
begin
  dcm0: dcm
    -- synopsys translate_off
    generic map (dll_frequency_mode => frequency_mode,
                 clkdv_divide => clkdv_divide_generic,
                 clkfx_multiply => clkfx_multiply_generic,
                 clkfx_divide => clkfx_divide_generic)
    -- synopsys translate_on
    port map (clkin => clk,
              clkfb => clk0buf,
              dssen => '0',
              psincdec => '0',
              psen => '0',
              psclock => '0',
              rst => dcm_rst,
              clk0 => clk0unbuf,
              clk2x => clk2xunbuf,
              clkfx => clkfxunbuf,
              clkdv => clkdvunbuf,

```

```

        locked => intlock);
bufg_clk0: bufg
port map (i => clk0unbuf,
          o => clk0buf);
bufg_clkfx: bufg
port map (i => clkfxunbuf,
          o => clkfxbuf);

-----
-- This is the DCM reset. It is a four-cycle shift register used to
-- hold the DCM in reset for a few cycles after programming.
-----

flop1: FDS port map (D => '0', C => clk, Q => ff1, S => '0');
flop2: FD port map (D => ff1, C => clk, Q => ff2);
flop3: FD port map (D => ff2, C => clk, Q => ff3);
flop4: FD port map (D => ff3, C => clk, Q => ff4);
dcm_rst <= ff2 or ff3 or ff4;

-----
-- SysGen Component Port Mapping
-- One clock input is being connected to clk0 of the DCM,
-- and the other clock is being connected to clkfx.
-----

two_async_clks: two_async_clks
port map (
    din_a => din_a,
    din_b => din_b,
    ss_clk_domaina_cw_ce => '1',
    ss_clk_domaina_cw_clk => clk0buf,
    ss_clk_domainb_cw_ce => '1',
    ss_clk_domainb_cw_clk => clkfxbuf,
    dout_b => dout_b);
end structural;

```

ChipScope Pro Analyzer を使用したリアルタイム ハードウェア デバッグ

System Generator フローに ChipScope Pro を統合すると、システム スピードでリアルタイムのデバッグが可能です。ChipScope ブロックを System Generator デザインに挿入することにより、FPGA のすべての内部信号およびノードをデバッグおよび検証できます。

ChipScope Pro の概要

FPGA デバイスの集積度の向上に伴い、テスト プローブをデバイスに取り付けるのは実用的ではなくなりました。ChipScope Pro は、ロジック解析ハードウェア コンポーネントをザイリンクス デバイス内に統合します。システム動作中にこれらのコンポーネントと通信し、ザイリンクス FPGA 内のノードのロジック解析を可能にします。ChipScope では、大容量トレース メモリ、高速クロック、さまざまなトリガ オプションが提供されます。このツールを使用すると、大きなロジック スペースを使用したり、複雑な手法を探したり、追加の I/O ピンを割り当てたりすることなく、FPGA 内の信号の動作を簡単にキャプチャし、表示できます。データ サンプルは、ユーザー定義のトリガ条件に基づいてキャプチャされ、内部ブロックメモリに保存されます。制御信号およびデータ信号はすべて JTAG ポートを介して転送され、I/O ピンを使用してデータをオフチップに駆動する必要はありません。

ChipScope Pro の詳細は、次の Web サイトを参照してください。

http://japan.xilinx.com/ise/optional_prod/cspro.htm

チュートリアル：System Generator 内での ChipScope の使用

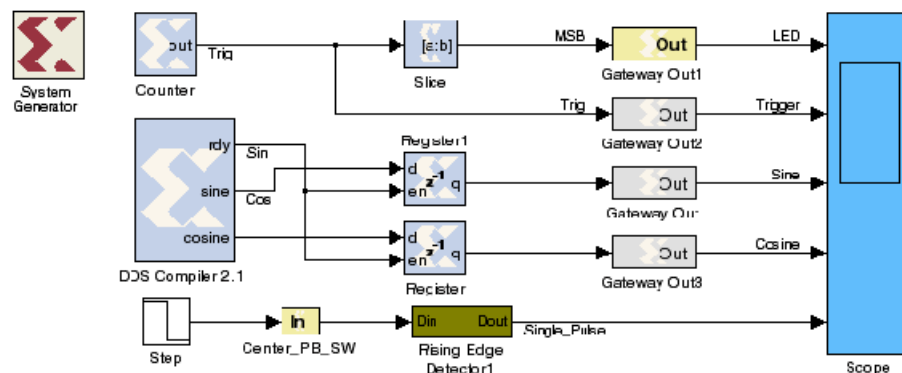
メモ: このチュートリアルを実行するには、ML506 プラットフォームを実行するのに必要なハードウェアとソフトウェアをインストールおよび設定する必要があります。インストールおよび設定に関する情報は、次の Web サイトから ML506 の資料を参照してください。


<http://japan.xilinx.com/products/boards/ml506/docs.htm>

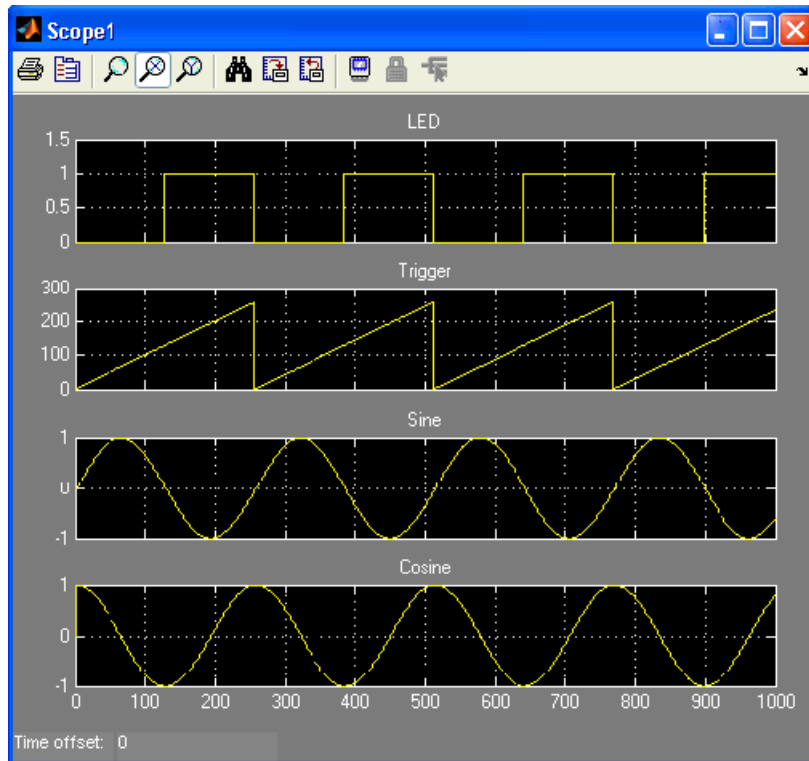
このチュートリアルでは、**Simulink** モデルを変更して **ChipScope** ブロックを統合し、デバッグ用にキャプチャおよび表示するデータの選択方法を説明します。手順は次のとおりです。

1. MATLAB ウィンドウで、`<path_to_sysgen>\examples\chipscope` ディレクトリに移動します。このディレクトリには、次のファイルが含まれています。
 - ◆ `chip.mdl` : 作業モデル
 - ◆ `chip_soln.mdl` : ChipScope ブロックを含む完成したモデル
2. `chip.mdl` を開きます。このモデルは、サイン/コサイン出力波形を生成する DDS Compiler ブロックの単純な使用モデルです。サインおよびコサインの出力波形は、後ほど ChipScope ブロックに接続し、波形をプローブおよび表示することにより System Generator ブロックをデバッグおよび検証します。
3. 8 ビット カウンタを使用して ChipScope をトリガします。Slice ブロックで最上位ビットを取り出し、さまざまな用途に使用できます。このチュートリアルでは、ML506 プラットフォームの LED を駆動します。

ChipScope Pro Tutorial Example



4. Simulink モデルのツールバーで [シミュレーションの開始]  をクリックし、モデルのシミュレーションを開始します。モデルを変更していないこの時点では、次の波形が表示されます。



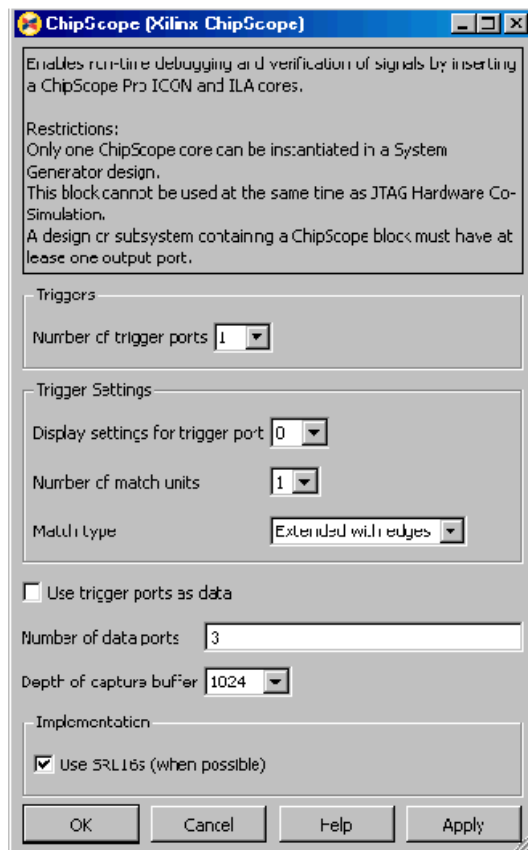
- ◆ 1 番上の波形は、8 ビット カウンタの最上位ビットを示します。カウンタ出力が 128 ～ 255 になると、MSB が 1 になります。
 - ◆ 2 番目の波形は、カウンタの完全な出力です。
 - ◆ 3 番目と 4 番目の波形は、サイン波およびコサイン波を示します。
5. ChipScope を Simulink モデルに組み込みます。ChipScope ブロックは、Simulink Library Browser の [Xilinx Blockset] → [Tools] にあります。ChipScope ブロックを Simulink モデルの右下のエリアにドラッグします。
6. ChipScope ブロックをダブルクリックし、次のパラメータを設定します。
- ◆ [Number of trigger ports] : 複数のトリガ ポートを使用すると、検出されるイベントの範囲が拡大し、保存する必要がある値の数を削減できます。16 個までのトリガ ポートを選択できます。この例で使用するトリガ ポートは 1 個のみです。
 - ◆ [Display settings for trigger port] : 各トリガ ポートに対し、一致ユニットの数と、一致タイプを設定する必要があります。このオプションでは、表示オプションを設定するトリガ ポートをドロップダウン リストから指定します。N 個のポートに対し、トリガ ポート 0 ～ N-1 のトリガ ポートの表示オプションが表示されます。この例では、Trig0 という名前のトリガ ポートが 1 個使用されるので、このオプションは 0 に設定します。
 - ◆ [Number of match units] : トリガ ポートごとに複数の一致ユニットを使用すると、イベント検出の柔軟性が向上します。トリガ イベントをテストするため、1 ～ 4 つの一致ユニットを使用できます。この例では、1 つの条件 (8 ビット カウンタの値) のみをチェックするので、このオプションは 1 に設定します。トリガ値は、ChipScope Pro Analyzer の実行時に設定します。

- ◆ [Match type] : このオプションは、次の 6 つのいずれかに設定できます。
 1. [Basic] : = または <> 比較を実行します。
 2. [Basic with edges] : [Basic] の操作に加え、High から Low、Low から High への遷移も検出します。
 3. [Extended] : = または <>、>、<、<=、>= 比較を実行します。
 4. [Extended with edges] : [Extended] の操作に加え、High から Low、Low から High への遷移も検出します。
 5. [Rnage] : =、<>、>、>=、<、<=、範囲内、範囲外比較を実行します。
 6. [Range with edges] : [Range] の操作に加え、High から Low、Low から High への遷移も検出します。

この例では、[Basic with edges] に設定します。

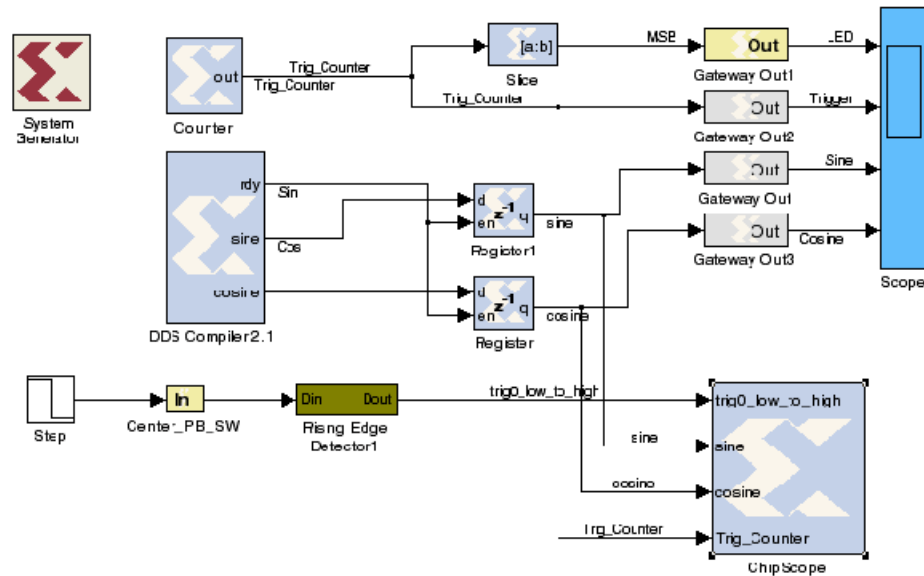
- ◆ [Number of data ports] : サンプルごとに 256 ビットまでをキャプチャできます。すべてのポートで使用されるビット数の合計を 256 ビット以下にする必要があります。System Generator ではデータ幅が自動的に伝搬されるので、データ ポートの数のみを指定します。この例では、サイン波、コサイン波、および trig_counter を表示するので、3 に設定します。
- ◆ [Depth of capture buffer] : このオプションは 2 のべき乗で設定し、16384 までの値を設定できます。この例では、1024 (Virtex-5 の最小値) に設定します。

これらのオプションを設定すると、ChipScope ブロックのパラメータ ダイアログ ボックスは次のようになります。



7. ChipScope ブロックの接続

ChipScope をトリガするのに使用する信号は、カウンタの出力です。Sine/Cosine ブロックからのサイン波とコサイン波の 2 つのバスをプローブします。次の図のように、信号を接続します。



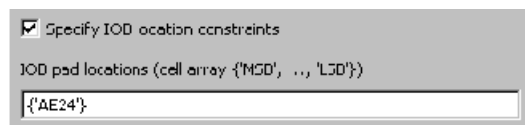
ChipScope ブロック上のポート名は、ブロックに接続した信号の名前 (Sine、Cosine など) になります。

8. ロケーション制約

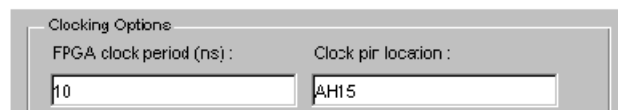
デザインをインプリメントし、シミュレーションしたら、ハードウェア ターゲットに接続する準備をします。デザインはどのハードウェア プラットフォームでも動作しますが、ここでは ML506 に接続する場合の手順を説明します。

このデザインでは、LED ピンとクロック ピンを固定する必要があります。

- ◆ LED ピン : Gateway Out1 ブロックをダブルクリックし、[Specify IOB location constraints] をオンにして、[IOB pad locations] に「{'AE24'}」(一重引用符が必要) と入力します。



- ◆ クロック ピン : System Generator トークンをダブルクリックし、[FPGA clock period] を 10ns に、[Clock pin location] を AH15 に設定します。

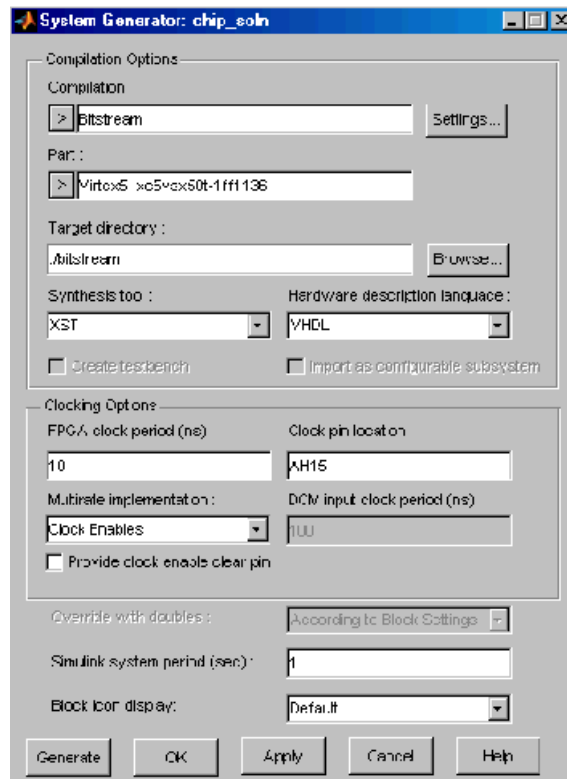


別のボードを使用している場合は、ピン ロケーションはボード に応じて変更する必要があります。

9. System Generator トークンの設定

ビットストリームを生成する前に、ターゲット デバイスとコンパイル ターゲットを設定する必要があります。

- ◆ System Generator トークンをダブルクリックし、パラメータ設定が次のようになっていることを確認します。



10. ビットストリームの生成

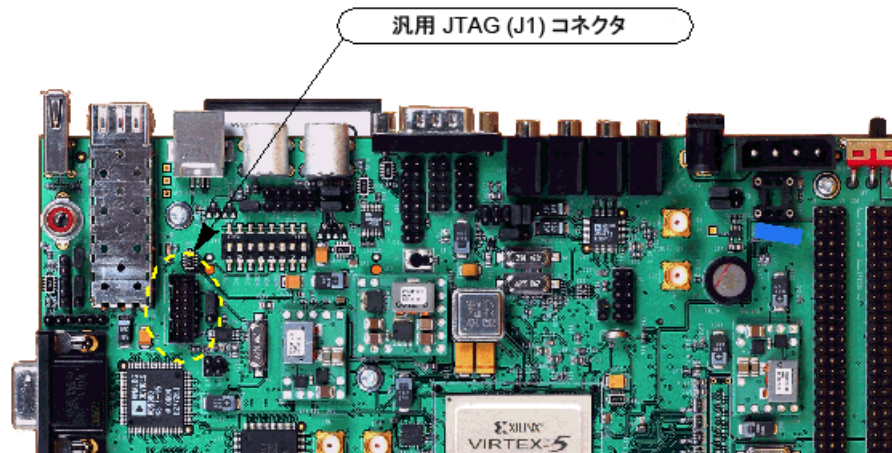
System Generator は、CORE Generator および ChipScope Core Generator を自動的に呼び出し、ネットリストおよびコアを生成します。[Compilation] に [Bitstream] を設定している場合は、コンフィギュレーション ビットストリームも生成されます。

- ◆ [Generate] ボタンをクリックしてビットストリームを生成します。
- ◆ CORE Generator が呼び出され、Sine/Cosine ブロックと Counter ブロックのネットリストが生成されます。ChipScope Core Generator が呼び出され、JTAG ポートを介して ChipScope Pro と通信する ILA (Integrated Logic Analyzer) コアと ICON コアが生成されます。

リアルタイム デバッグ

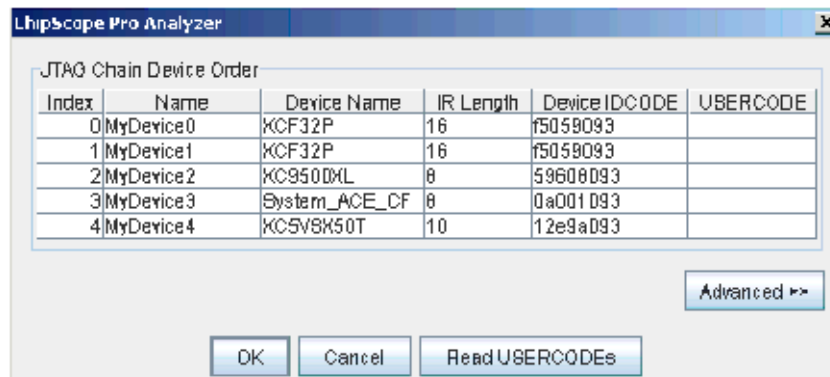
次に、ML506 プラットフォーム上でデザインを実行し、ChipScope Pro Analyzer でプローブ出力を表示します。

1. パラレル ケーブル IV またはプラットフォーム ケーブル USB の一端を ML506 ボード上の汎用 JTAG コネクタ (J1) に接続し、もう一端をコンピュータに接続します。



2. ChipScope Pro Analyzer を起動します。

- ◆ アイコンをクリックするか、[JTAG Chain] → [Xilinx Platform USB Cable] をクリックして、JTAG チェーンを開きます。JTAG チェーンのデバイス順を示す次の表が表示されます。デバイス順を確認し、[OK] をクリックします。



3. FPGA をコンフィギュレーションします。

- ◆ [New Project] ウィンドウで [Device 4] を右クリックし、[Configure] → [Select New File] をクリックします。前のセクションの手順 10 で生成したビットストリーム (./bitstream/chip_cw.bit) を指定します。コンフィギュレーションが終了すると、ChipScope Analyzer のウィンドウの下部に「Found 1 Core Unit in the JTAG chain」というメッセージが表示されます。

4. ChipScope プロジェクト ファイルをインポートします。

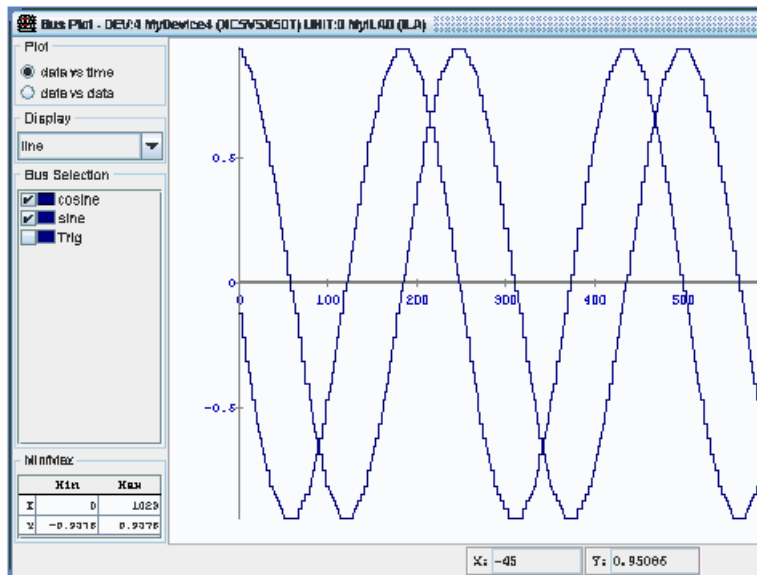
System Generator では、データ信号をバスにグループ化するために ChipScope のプロジェクト ファイルを作成します。バスは各データ ポートに対して作成され、Simulink 環境でと同様 (符号および精度) に表示できるようにしています。

[File] → [Import] → [Select New File] をクリックし、<path_to_sysgen>\examples\chipscope\netlist\temp\chip_chipscope.cdc を選択して、このプロジェクト ファイルを読み込みます。

5. サイン波を表示します。

- ◆ [New Project] ウィンドウの [Device 4] の下にある [Trigger Setup] をダブルクリックし、セットアップ ウィンドウを表示します。ここでは表示するだけで設定はしません。
- ◆ [New Project] ウィンドウの [Device \$] → [Unit 0 MyILA0 (ILA)] の下にある [Bus Plot] をダブルクリックします。

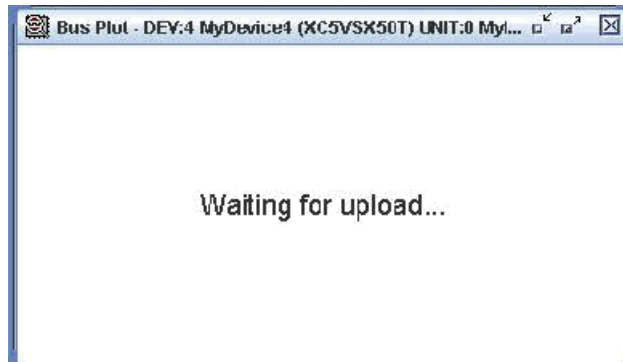
[Bus Plot] ウィンドウが開きます。[Bus Selection] で cosine および sine を選択し、▶ ボタンをクリックします。トリガ条件を設定していないので、値はすぐにキャプチャされます。サイン波とコサイン波が次のように表示されます。波形を点、ライン、またはその両方で表すよう表示オプションを選択できます。



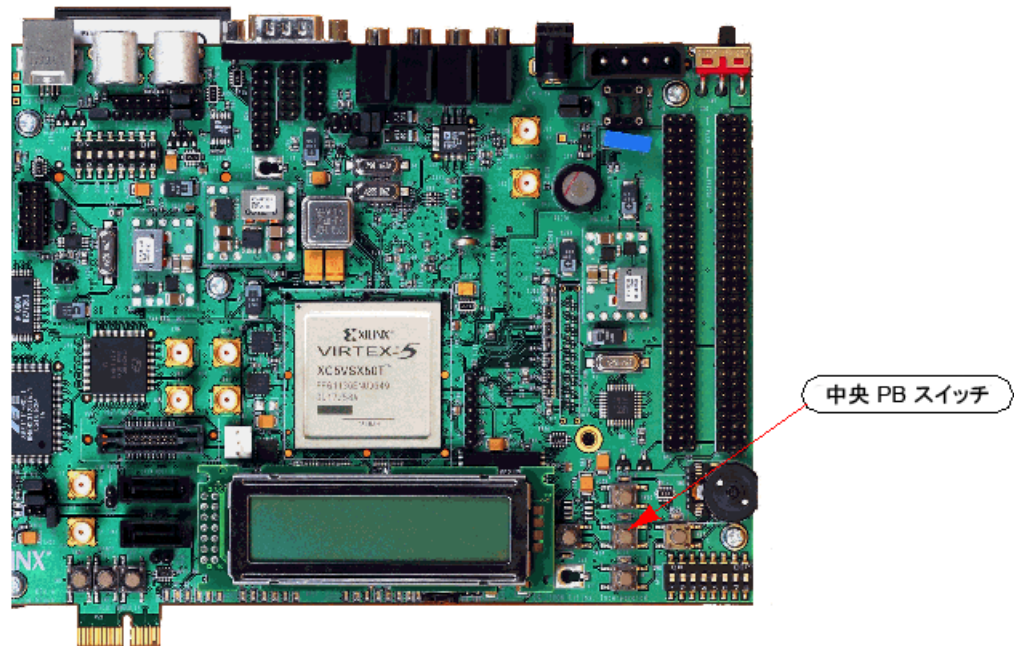
6. トリガを設定します。

[Trigger Setup] ウィンドウで、X 値をすべて 1 に変更します。このトリガには Low から High に遷移するパルスが使用されますが、次の図に示す中央の PB スイッチを押すことで手動でトリガできます。Low から High に遷移するパルスが検出されると、ChipScope でデータのキャプチャが開始します。バッファを 1024 に設定したので、1024 個までのデータ ポイントがキャプチャされ、表示されます。

▶ ボタンをクリックしてデータを再度キャプチャすると、次のような画面が表示され、データをキャプチャおよび表示するためのトリガ信号を待っていることが示されます。



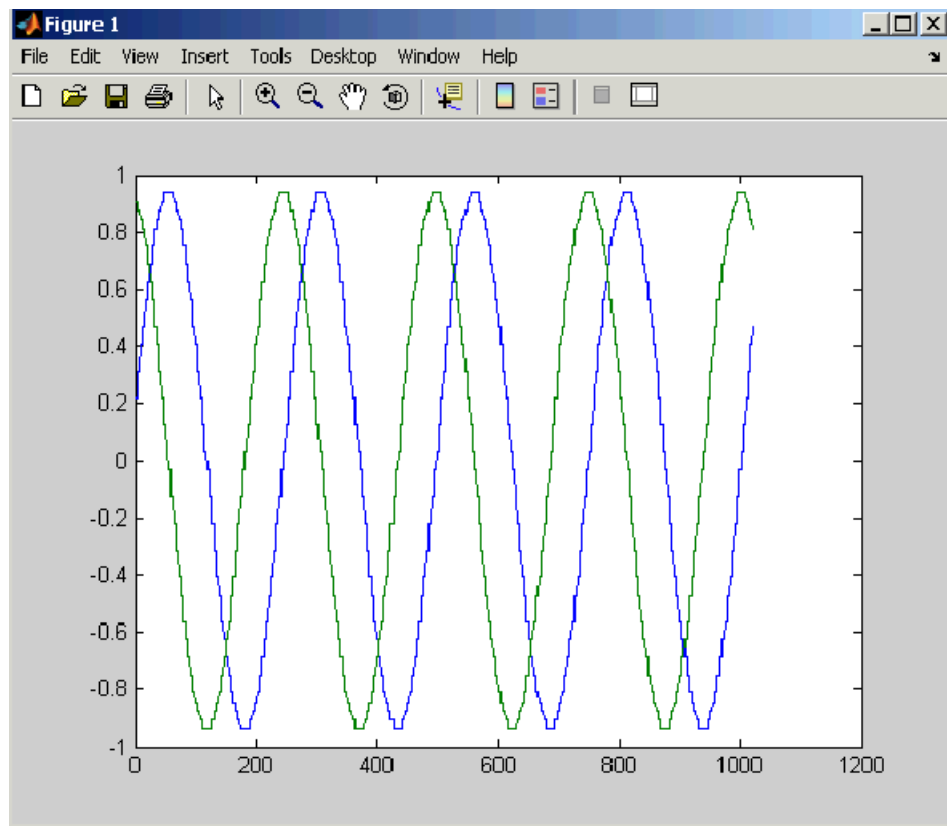
この方法は、データをいつキャプチャするかを完全に制御するのに有益です。この方法を使用するには、PB スイッチの 1 つをシングル ショット (立ち上がりエッジ検出器) 回路に接続します。このチュートリアルでは、中央の PB スイッチ (AJ6、SW14) を使用します。



ChipScope から MATLAB ワークスペースへのデータのインポート

ChipScope でキャプチャしたデータを MATLAB ワークスペースにインポートします。

1. ChipScope Pro Analyzer からデータをエクスポートします。
 - ◆ ChipScope Pro Analyzer で [File] → [Export] をクリックします。[Format] で [ASCII] をオンにし、[Signals to Export] で [Bus Plot Buses] を選択します。[Export] をクリックし、ファイルを `sinecos.prn` という名前で保存します。
2. MATLAB を起動し、`sinecos.prn` を保存したディレクトリに移動します。
 - ◆ 「`xlLoadChipScopeData('sinecos.prn');`」と入力します。このコマンドを実行すると、PRN ファイルからのデータが MATLAB ワークスペースに読み込まれます。ワークスペースに、`Sin` および `Cos` という 2 つの新しいアレイが表示されます。
3. これらの値を、MATLAB の `plot` 関数を使用して波形として表示できます。
 - ◆ 「`plot(1:1024, sine, 1:1024, cosine)`」と入力すると、次の曲線が生成されます。



ハードウェア/ソフトウェア協調設計

この章では、System Generator を使用したソフトウェアとハードウェアの設計について説明します。

System Generator でのハードウェア/ソフトウェア協調設計	System Generator を使用したハードウェア/ソフトウェア協調設計の概要を示します。
カスタム ロジックへのプロセッサの統合	プロセッサをカスタム ロジックに統合する方法を説明します。
EDK サポート	ザイリンクス エンベデッド開発キットのサポートについて説明します。
エンベデッド プロセッサおよびマイクロプロセッサを使用した設計	エンベデッド プロセッサを使用したデザインの設計について説明します。

System Generator でのハードウェア/ソフトウェア協調設計

System Generator では、**Black Box** ブロック、**PicoBlaze Microcontroller** ブロック、または **EDK Processor** ブロックをインポートすることにより、プロセッサをモデルに含めることができます。

Black Box ブロック

Black Box ブロックを使用すると、最も柔軟な設計が可能ですが、デザインの複雑性が増します。この方法では、どんなプロセッサ HDL でも System Generator デザインに組み込むことができます。プロセッサのすべてのポートとバスを System Generator のブロック図に追加し、その他の System Generator ブロックと接続します。ソフトウェアのコンパイルの問題も、完全に制御できます。詳細は、「[HDL モジュールのインポート](#)」を参照してください。

PicoBlaze Microcontroller ブロック

PicoBlaze™ Microcontroller ブロックを使用すると、柔軟性は低くなりますが、使用は最も簡単です。PicoBlaze Microcontroller ブロックは、PicoBlaze マクロを使用して 8 ビットのエンベデッド マイクロコントローラをインプリメントし、System Generator への固定のインターフェイスを含みます。通常は、1024 X 8 ビットのブロック ROM 1 つにプログラムを保存できます。PicoBlaze は、PicoBlaze アセンブリ言語を使用してプログラムできます。このフローの詳細は、「[PicoBlaze マイクロコントローラ アプリケーションの設計](#)」を参照してください。

EDK Processor ブロック

EDK Processor ブロックは、Xilinx Platform Studio (XPS) で作成した MicroBlaze™ プロセッサへのインターフェイスを提供します。このブロックを使用すると、System Generator の共有メモリブロック (From Register、To Register、From FIFO、To FIFO、Shared Memory ブロック) を自動生成されたメモリ マップ インターフェイスを介してプロセッサに接続できます。接続したメモリは、MicroBlaze プロセッサで実行しているソフトウェアで読み出しおよび書き込みできます。このフローの詳細は、「[カスタム ロジックへのプロセッサの統合](#)」を参照してください。

EDK Processor ブロックを使用すると、Xilinx Platform Studio (XPS) および Base System Builder を使用して作成した EDK プロジェクトで指定した MicroBlaze プロセッサをインポートできます。また、EDK Processor ブロックを含む System Generator デザインを EDK プロジェクトにエクスポートすることも可能です。

エクスポートを実行すると PLB ベースまたは FSL ベースの pcore が作成され、これを XPS プロジェクトに追加して MicroBlaze または PowerPC® と通信させることができます。

カスタム ロジックへのプロセッサの統合

プロセッサをユーザー定義ロジックと統合するのは、通常かなり複雑です。プロセッサとカスタムハードウェア間の通信は共有バスを介して行われ、通信される情報には、処理用データ、ハードウェアの状態を表すデータ、処理モードに関連するデータなど、異なるタイプのデータが含まれます。これらのデータをプロセッサとカスタム ロジック間でどのように転送するかを計画するのは、時間がかかり、エラーの原因ともなるので、自動化すると有益です。接続だけではなく、カスタム ロジックとの通信を実行するソフトウェアを記述するのも簡単ではありません。

EDK Processor ブロックを使用すると、これらの作業を自動的に実行できます。このブロックでは、プロセッサとカスタム ロジック間のインターフェイスを共有メモリで指定することが推奨されます。共有メモリは、名前で参照可能な格納場所として使用され、これによりメモリ マップとソフトウェアドライバの生成が可能になります。

ブロックの使用方法については、[EDK Processor](#) ブロックに関する記述を参照してください。次に、メモリ マップと生成されるドライバの機能を説明します。

メモリ マップの作成

プロセッサに共有メモリを追加した場合に生成されるメモリ マップについて説明します。

ハードウェアの生成

ハードウェア生成オプションについて説明します。

ハードウェア協調シミュレーション

EDK Processor ブロックのハードウェア協調シミュレーション モデルの作成方法を説明します。

ソフトウェアドライバの生成

ソフトウェアドライバの作成方法を説明します。

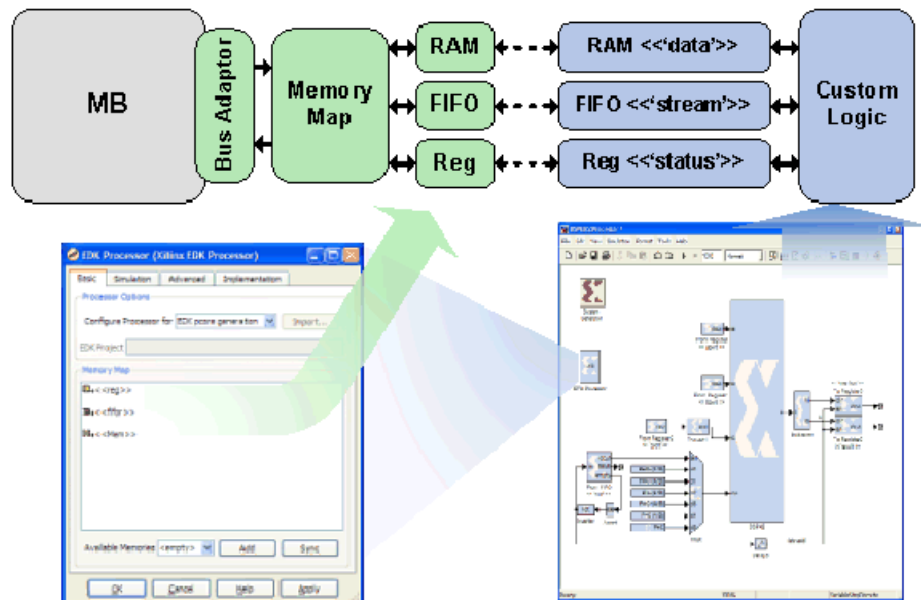
EDK プロセッサのソフトウェア記述

System Generator で作成したハードウェアを制御するソフトウェアを記述するプロセスを説明します。

EDK プロセッサでの非同期のサポート

プロセッサと System Generator デザインを異なるクロックで動作させることができるようにする System Generator の機能についてインポート モードおよびエクスポート モードの両方で説明します。

メモリ マップの作成



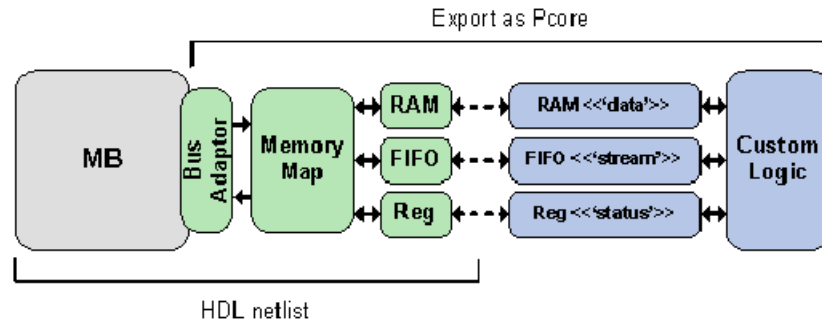
上図の右下に、System Generator モデルが示されています。この System Generator モデルが、MicroBlaze プロセッサに統合するカスタム ロジックです。モデルの作成では、ソフトウェア アクセスが必要な部分に共有メモリを使用します。たとえば、ハードウェアのステータスはレジスタに保存できますが、この情報をプロセッサに認識させるには、レジスタの代わりに名前の付いた共有レジスタを使用します。この共有レジスタに **status** という名前を付けることにより、メモリの内容に名前を付けることができ、ソフトウェア開発の際に有益です。

EDK Processor ブロックのパラメータ ダイアログ ボックスでは、プロセッサのメモリ マップに共有メモリを追加できます (上図の左下)。上図の上部は、データ フローを示します。共有メモリをプロセッサのメモリ マップに追加すると、対応する共有メモリが作成され、EDK Processor ブロック用に生成されるメモリ マップに添付されます。このメモリ マップは、バス アダプタを使用して MicroBlaze プロセッサに接続されます。

ハードウェアを生成すると、各共有メモリのペアが 1 つの物理メモリとしてインプリメントされます。共有メモリの各クラスのインプリメンテーションについては、「ハードウェア協調シミュレーションの使用」の章の「共有メモリのサポート」を参照してください。

ハードウェアの生成

EDK Processor ブロックでは、EDK pcore 生成 ([EDK pcore generation]) と HDL ネットリスト生成 ([HDL netlisting]) の 2 つのモードがサポートされています。次の図に、これらのモードを示します。モードは、EDK Processor ブロックのパラメータ ダイアログ ボックスで選択できます。



EDK pcore 生成モード

ザイリンクスのエンベデッド開発キット (EDK) では、作成したプロセッサにペリフェラルを接続できます。これらのペリフェラルは、pcore としてまとめられます。各 pcore には、ペリフェラルのハードウェア記述、ソフトウェアドライバおよびバス接続を記述するファイル、ペリフェラルに関する情報が含まれます。

EDK pcore 生成モードに設定し、System Generator トークンで [EDK Export Tool](#) を指定すると、System Generator モデルから pcore が生成されます。上図は、pcore として生成したモデルの一部を示しています。このモードに設定すると、モデルに追加される MicroBlaze プロセッサはプレースホルダであると認識されます。実際のインプリメンテーションは、ペリフェラルが EDK に追加されたときに EDK により供給されます。pcore 自体は、カスタム ロジック、生成されたメモリ マップ、カスタム ロジックへの仮想接続、およびバス アダプタで構成されています。

HDL ネットリスト生成モード

HDL ネットリスト 生成モードを選択すると、EDK プロセッサを System Generator モデルに組み込むことができます。EDK Processor ブロックで HDL ネットリスト 生成モードを設定するのは、EDK プロジェクトをブロックに供給する場合のみです。HDL ネットリスト 生成モードでは、EDK プロジェクトで記述されたプロセッサを System Generator にブラック ボックスとしてインポートします。インポートされた EDK プロジェクトには、System Generator メモリ マップをプロセッサに接続するのに必要なバス インターフェイスも追加されます。ネットリストを生成すると、MicroBlaze プロセッサとメモリ マップ ハードウェアがハードウェアに組み込まれます。

ハードウェア協調シミュレーション

EDK Processor ブロックでは、ハードウェア協調シミュレーションにより、ハードウェア ベースのシミュレーションがサポートされています。ハードウェア協調シミュレーションブロックは、「[ハードウェア協調シミュレーションの使用](#)」に説明されている標準の協調シミュレーション フローで作成できます。違いは、インポートした XPS プロジェクトの最上位ポートの処理のみです。

System Generator に XPS プロジェクトをインポートすると、インポート ウィザードではハードウェア協調シミュレーション ブロックを作成したときにすべてのポートが適切に制約されていると想定されます。たとえば、XPS システムの最上位エンティティに FPGA のパッドに接続されているポートが含まれている場合、ハードウェア協調シミュレーション ブロックをコンパイルする際これらのポートは FPGA 上のパッドに接続されたままとなり、ハードウェア協調シミュレーションブロックにポートとして含まれません。同様に、インポートした XPS システムの最上位ポートにビットストリーム フロー制約が設定されている場合、この制約は保持されます。

パッドに接続されていない最上位ポートがある場合や、ポートが制約されていない場合は、EDK Processor ブロックの [Advanced] タブにある [Processor Port Interface] 表を使用してポートが System Generator で認識されるようにすることができます。詳細は、「[System Generator へのプロセッサ ポートの追加](#)」を参照してください。

ソフトウェアの記述およびコンパイルには EDK の XPS を使用できますが、シミュレーションの前に協調シミュレーションブロックの [Software] タブで [Compile and update bitstream] ボタンをクリックして、コンパイルした C コードをビットストリームに組み込む必要があります。

ネットワーク ベースのハードウェア協調シミュレーションでサポートされているハードウェア ボードを使用すると、FPGA 上の JTAG ポートを XMD でのソフトウェア デバッグに使用できます。

ソフトウェア ドライバの生成

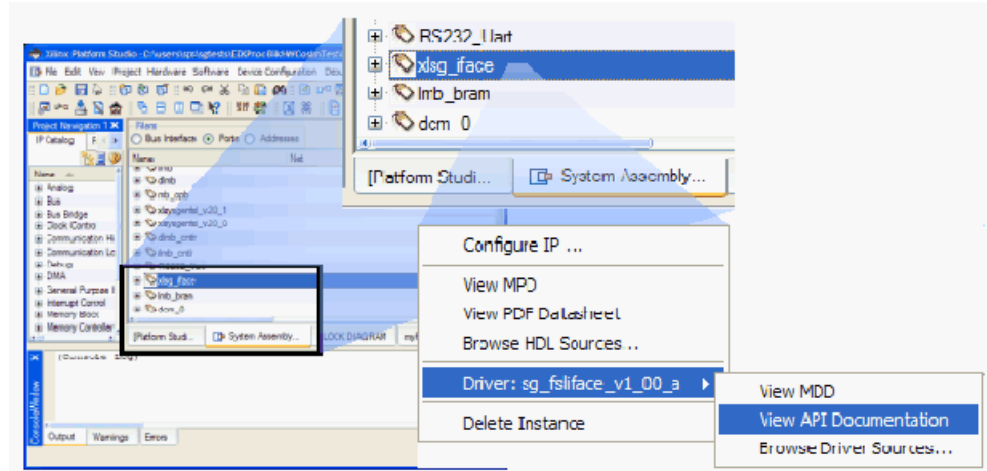
どちらの動作モードでも、メモリ マップを生成すると、ソフトウェア ドライバテンプレートが自動的に作成されます。ドライバテンプレートは、EDK でソフトウェア ライブラリがコンパイルされる際に生成されます。これは、EDK から実行可能です。ソフトウェア ライブラリがコンパイルされると、ドライバを参照できるようになり、ドライバの情報にもアクセスできます。

EDK Processor ブロックを EDK pc core 生成モードにすると、ソフトウェア ドライバの生成は pc core のインスタンス名に依存します。たとえば、System Generator で作成した pc core の名前が sysgen_fft_sm だとすると、これがペリフェラル名になります。EDK プロセッサには複数のペリフェラルを追加できるので、各インスタンス名は固有のものにする必要があります。EDK では、ペリフェラル名に自動的に接尾番号が追加されます。1 つ目のペリフェラルを追加すると、sysgen_fft_sm_0 というような名前が付けられ、これがペリフェラルのインスタンス名になります。ペリフェラルのインスタンス名は変更できます。詳細は、EDK XPS の資料を参照してください。

EDK Processor ブロックを HDL ネットリスト生成モードにし、EDK プロジェクトを System Generator モデルにインポートすると、System Generator により特別なペリフェラルが EDK プロジェクトに追加されます。このペリフェラルは xls_g_iface という名前で、MicroBlaze プロセッサと System Generator モデル間の接続を指定します。このペリフェラルは、1 つのインスタンスしか追加できません。EDK プロジェクトに関連付けることのできる EDK Processor ブロックは 1 つのみです。

EDK プロセッサのソフトウェア記述

ペリフェラルのソフトウェアドライバおよびその情報は、すべてのソフトウェアライブラリが EDK でコンパイルされた後に生成されます。コンパイル後、EDK からソフトウェアドライバの情報にアクセスできます。



上図は、EDK XPS の GUI を示しています。[System Assembly View] タブで System Generator ペリフェラルを右クリックし、[Driver] → [View API Documentation] をクリックすると、ペリフェラルの情報が表示されます。このオプションがない場合は、ドライバをコンパイルする必要があります。ドライバをコンパイルするには、XPS で [Software] → [Generate Libraries and BSPs] をクリックします。

ヘッダファイルの情報、ドライバの呼び出し、メモリ マップ、サンプル コードは、生成された情報を参照してください。

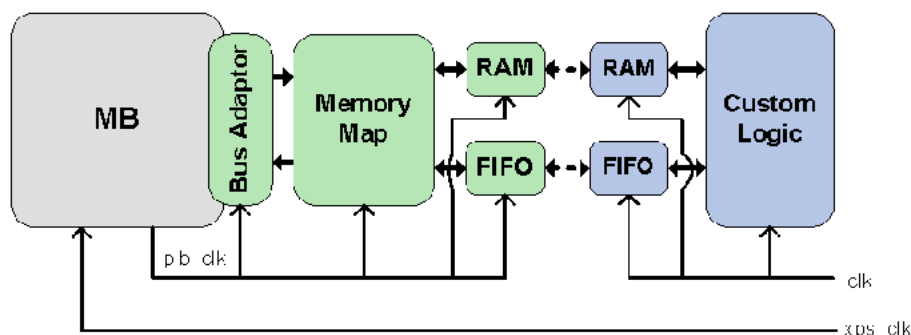
生成されたソフトウェアドライバには、共有メモリにアクセスする 4 つの基本関数があります。次の表で、<inst> はペリフェラルのインスタンス名を表します。

```
int <inst>_Read (unsigned int memName,
               unsigned int addr,
               unsigned int* val);
int <inst>_ArrayRead (unsigned int memName,
                    unsigned int startAddr,
                    unsigned int transferLength,
                    unsigned int** valBuf);
int <inst>_Write (unsigned int memName,
                unsigned int addr,
                unsigned int val);
int <inst>_ArrayWrite (unsigned int memName,
                    unsigned int startAddr,
                    unsigned int transferLength,
                    const unsigned int* valBuf);
```

EDK プロセッサでの非同期のサポート

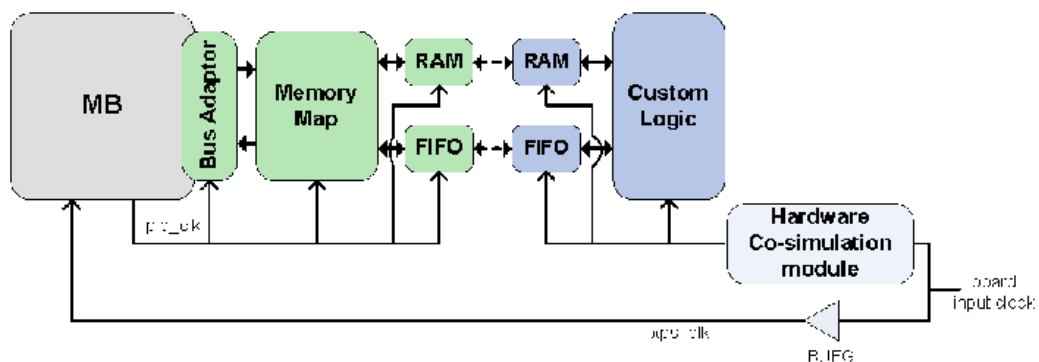
プロセッサでの非同期のサポートにより、プロセッサとプロセッサに接続されているハードウェア アクセラレータに異なるクロックを使用できます。これにより、ハードウェア アクセラレータを外部パブリックと接続する必要がある場合に、最速のクロック レートまたは正しい動作に必要なクロック レートで動作させることができます。

この機能は、EDK Processor ブロックのパラメータ ダイアログ ボックスの [Implementation] タブで [Dual Clock] をオンにするとイネーブルになります。次の図は、インポート およびエクスポート フローでクロックがどのように接続されるかを示します。エクスポート フローには、MB (MicroBlaze) ブロックはありません。基本的に、System Generator のカスタム ロジック デザインは clk クロックで駆動され、プロセッサシステムは xps_clk クロックで駆動されます。MicroBlaze プロセッサシステムの PLB バスを駆動するクロック ソースが抽出され、バス アダプタ、メモリ マップ、共有メモリのペアを駆動します。共有メモリはこれら 2 つのクロックドメイン (clk ドメインと plb_clock ドメイン) にまたがっており、これら両方のクロックで駆動されます。このフローでは、共有レジスタはサポートされません。XPS プロジェクトを System Generator にインポート するインポート フローでは、プロセッサ上の PLB バスを xps_clk 信号と同じクロックで駆動する必要があります。



[Dual Clock] をイネーブルにし、ハードウェア協調シミュレーション用のネットリストを作成すると、多少異なるクロック接続方法が使用されます。これを次の図に示します。ボードからのクロック ソースは 2 つに分岐されます。1 つの分岐は clk に接続される前にハードウェア協調シミュレーション モジュールに供給され、もう 1 つの分岐はクロック バッファを介して xps_clk 信号に接続されます。

この接続方法では、MicroBlaze プロセッサをフリー ランニング モードで動作させながら System Generator で設計したカスタム ロジックをシングルステップ クロック モードで実行できます。これにより、ハードウェア協調シミュレーション ブロックをシングル ステップ クロック モードに設定した場合に、RS232 UARTS などのクロックの影響を受けやすいペリフェラルを機能させることができます。



ハードウェア協調シミュレーションでは、プロセッサ サブシステムはボードのクロックで直接駆動されます。これは、プロセッサ サブシステムがこのクロックで設定された要件を満たす必要があるということです。ハードウェア協調シミュレーションでは、入力ボード周波数に基づいて異なるクロック比のクロック周波数を選択できます。このハードウェア協調シミュレーション クロックはハードウェア協調シミュレーション ブロック内で生成され、プロセッサ サブシステムでは使用できません。

たとえば、入力ボード周波数が 125MHz で、ハードウェア協調シミュレーションの周波数が 33MHz に設定されている場合、MicroBlaze プロセッサは 125MHz で動作する必要があります。MicroBlaze プロセッサがこの速度でタイミングを満たさない場合、XPS にクロック生成ペリフェラルをインスタンスエートしてクロックの速度を下げる必要があります。

EDK サポート

EDK プロセッサのインポート

EDK プロジェクトを EDK Import Wizard を使用して System Generator にインポートする方法を説明します。

System Generator へのプロセッサポートの追加

EDK の最上位ポートを System Generator に配線する方法を説明します。

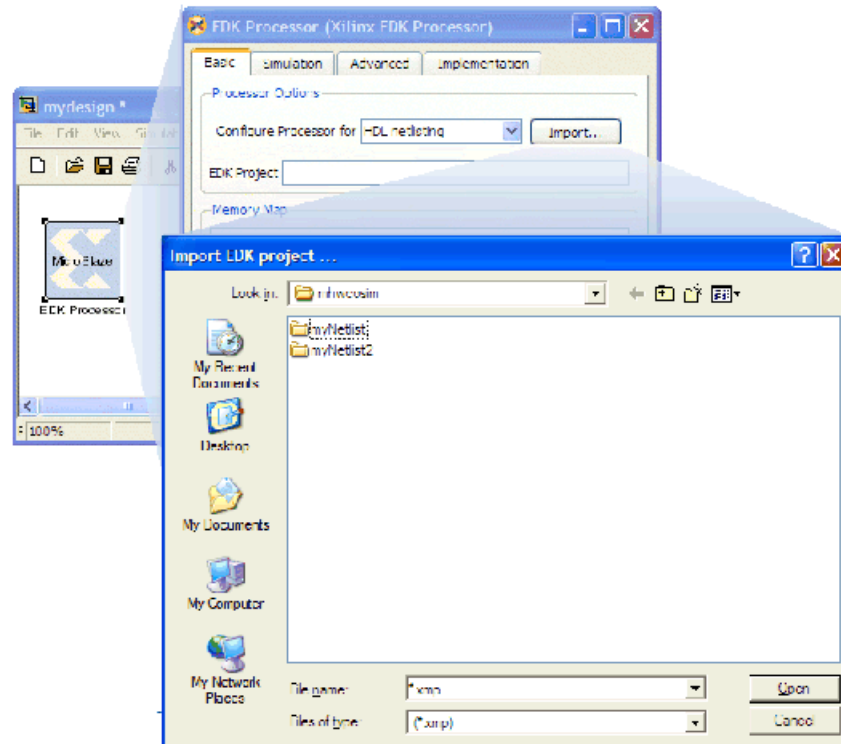
pcore のエクスポート

System Generator デザインを pcore として EDK にエクスポートする方法を説明します。

EDK プロセッサのインポート

メモ：11.3 リリース以降、System Generator での FSL のサポートに関する開発は中止されます。FSL は ISE Design Suite 11 では引き続き使用できますが、ISE Design Suite 12 ではサポートされません。

EDK に含まれる Xilinx Platform Studio (XPS) を使用して作成したプロセッサは、EDK Import Wizard を使用して System Generator にインポートできます。



EDK Processor ブロックから EDK Import Wizard を起動するには、[Import] ボタンをクリックするか、[EDK project] を空にして [HDL netlisting] を選択します。

メモ：EDK プロジェクトを System Generator にインポートすると、EDK プロジェクトが変更されます。これら変更について、次に説明します。

EDK Import Wizard

EDK Import Wizard を起動すると、EDK プロジェクト ファイル (XMP ファイル) を指定するダイアログ ボックスが表示されます。

[Import] ボタンをクリックすると、インポートが開始します。

メモ：インポートを実行すると、EDK プロジェクトが System Generator 内で機能するよう変更されます。変更されていない元の EDK プロジェクトを保持する場合は、インポートを実行する前にコピーを作成してください。System Generator では、EDK プロジェクトのハードウェア プラットフォーム仕様 (MHS ファイル) およびソフトウェア プラットフォーム仕様 (MSS ファイル) を拡張子が .bak のファイルに自動的にバックアップします。

EDK プロジェクトを System Generator にインポートすると、EDK Processor ブロックで設定したオプションによって、FSL のペアまたは PLB46 インターフェイスも追加されます。インターフェイスのソフトウェア ドライバとして、pcore (FSL の場合は xls_g_iface、PLB の場合は xls_g_plbiface) が追加されます。EDK プロジェクトに含まれる MHS および MSS ファイルが変更され、プロセッサを記述する HDL ファイルが生成されて System Generator プロジェクトにリンクされます。

メモ：11.3 リリース以降、System Generator での FSL のサポートに関する開発は中止されます。FSL は ISE Design Suite 11 では引き続き使用できますが、ISE Design Suite 12 ではサポートされません。

インポート後のプロセッサ ハードウェアの変更

EDK プロジェクトをインポートした後に EDK 内でハードウェアに加えた変更は、System Generator 内には反映されません。ハードウェアの構成は、インポートした時点で固定されます。プロセッサハードウェアに変更を加えた場合は、EDK プロジェクトを EDK Import Wizard を使用してインポートし直す必要があります。

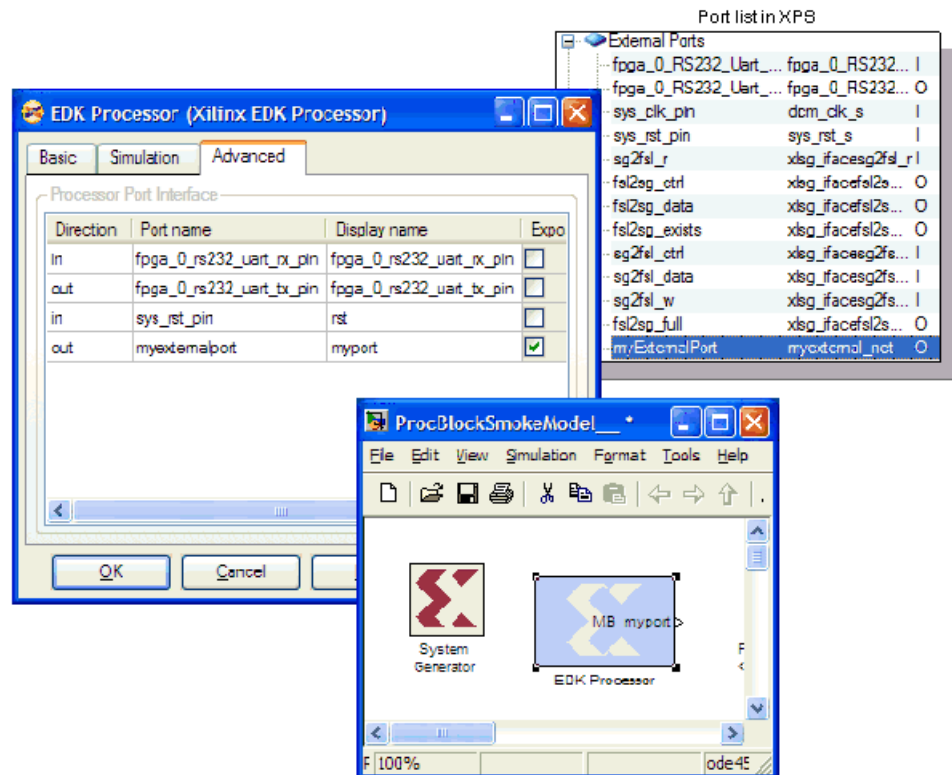
EDK プロジェクトを変更した場合は、インポートし直すことをお勧めします。System Generator での以前のインポートで自動的に追加された PLB または FSL インターフェイスおよび関連する pcore は EDK Processor ブロックで認識されるので、XPS プロジェクトにハードウェアまたはソフトウェアが重複して追加されることはありません。

制限

現在のところ、EDK Import Wizard でインポートできるのはプロセッサが 1 つのプロジェクトのみで、MicroBlaze プロセッサのみがサポートされます。プロセッサに追加されているペリフェラルは、その他の System Generator サービスで利用されるリソースと競合させることはできません。たとえば、ネットワークベースのハードウェア協調シミュレーションを使用する場合、EDK プロジェクトでイーサネット MAC を使用するペリフェラルは使用できません。

System Generator へのプロセッサ ポートの追加

プロセッサと System Generator 間でデータを転送するには、共有メモリを介する方法が推奨されますが、System Generator にプロセッサの最上位ポートを追加することも可能です。



上図の右上は、XPS での EDK プロジェクトを示します。外部ポート リストには、myExternalPort というユーザー定義のポートが含まれます。

EDK プロジェクトをインポートした後、**System Generator** でプロセッサのパラメータ ダイアログ ボックスを開きます。**[Advanced]** タブをクリックし、プロセッサ ポート インターフェイスのリストを表示します。

ポート リストには、プロセッサで使用可能な最上位ポートが表示されます。クロック ポートおよびメモリ マップ インターフェイスをインプリメントするために **System Generator** で使用される信号は表示されません。上図の例では、RS232 のポート、sys_rst_pin、myexternalport ポートをブロックの最上位に追加できます。**[Expose]** チェック ボックスをオンにすると、**EDK Processor** ブロックにポートが表示されます。元の名前が長すぎる場合は、上図に示すように、ポートの表示名を変更できます。

この方法では、**System Generator** で生成されたメモリ マップを介することなく、プロセッサのポートを **System Generator** デザインに直接接続できます。プロセッサのリセット ポート、割り込みポートを **System Generator** ダイアグラムに直接接続する場合などにこの方法を使用できます。

pcore のエクスポート

EDK Processor ブロックを含む **System Generator** デザインは、**System Generator** トークンの**[Compilation]** を**[EDK Export Tool]** に設定することにより、EDK pcore としてエクスポートできます。

EDK に pcore としてエクスポートするには、EDK Processor ブロックを EDK pcore 生成モードでコンフィギュレーションする必要があります。これには、EDK Processor ブロックのパラメータ ダイアログ ボックスで**[Configure processor for]** を**[EDK pcore generation]** に設定します。

詳細は、「[EDK Export Tool](#)」を参照してください。

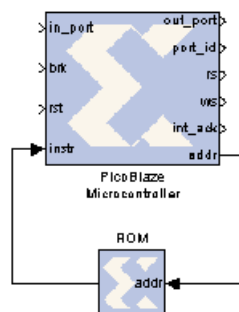
エンベデッド プロセッサおよびマイクロプロセッサを使用した設計

PicoBlaze マイクロコントローラ アプリケーションの設計

System Generator の PicoBlaze Microcontroller ブロックは、8 ビット マイクロコントローラをインプリメントします。このブロックは、複雑だがタイミングがクリティカルでないステート マシンを必要とするアプリケーションや、データ処理アプリケーションなどで使用できます。マイクロコントローラは完全にデバイスに組み込まれており、外部サポートは必要ありません。追加のロジックはデバイス内のマイクロコントローラに接続できるので、柔軟な設計が可能です。

PicoBlaze の概要

次の例では、リソース使用量を少なくするよう最適化された PicoBlaze 3 (以下単に PicoBlaze) を使用します。メモリ ブロックは、1024 個までの命令を保存するために使用されます。



PicoBlaze 命令セット アーキテクチャ

PicoBlaze は、ハードウェア中心のマイクロコントローラで、アセンブリ コードを使用してプログラムできます。命令数が 1024 個までのプログラムがサポートされます。これ以上のプログラムが必要な場合は、通常複数のマイクロコントローラを使用します。

16 個の汎用レジスタ

s0 ~ sF に指定された 16 個の 8 ビット汎用レジスタがあります。

ALU

論理演算ユニット (ALU) は、add、sub、load、and、or、xor、shift、rotate、compare、test などの演算を実行します。各命令の最初のオペランドは、結果を保存するレジスタです。2 つ目のオペランドが必要な演算では、2 つ目のレジスタまたは 8 ビットの定数値を指定できます。

フラグおよびプログラム制御

ALU 演算の結果は、ゼロ フラグおよびキャリー フラグのステータスにより判断されます。ゼロ フラグは結果が 0 の場合に 1 になり、キャリー フラグは演算でオーバーフローが発生した場合に 1 になります。これらのフラグのステータスから、jump や call などの条件プログラム フロー制御命令を使用してプログラムの実行シーケンスを決定します。

入力/出力

256 個の入力ポートと 256 個の出力ポートがあります。アクセスするポートは、port_id の 8 ビット アドレス値で指定されます。ポート アドレスは、プログラム内で絶対値または間接的にレジスタの値として指定できます。入力操作では、in_port の値が 16 個のレジスタのいずれかに転送され、出力操作ではデータがレジスタから out_port に転送されます。

割り込み

プロセッサには、brk という割り込み入力ポートが 1 つあります。割り込みがイネーブルの場合、brk を 1 にすると、プログラム カウンタが割り込みサービス ルーチンへのジャンプ ベクタが保存されているメモリ ロケーション 0x3FF に設定されます。この時点で ack ポートでパルスが生成され (brk がアサートされた 2 クロック サイクル後)、制御フラグが保持されて、それ以外の割り込みがディスエーブルになります。return 命令により、割り込みルーチンが終了したときに制御フラグのステータスが復元され、その後の割り込みをイネーブルにするかどうか指定されます。

機能および命令セットの詳細は、ザイリンクス Web サイトの [PicoBlaze User Resources のページ](#)を参照してください。

チュートリアル : System Generator での PicoBlaze の使用

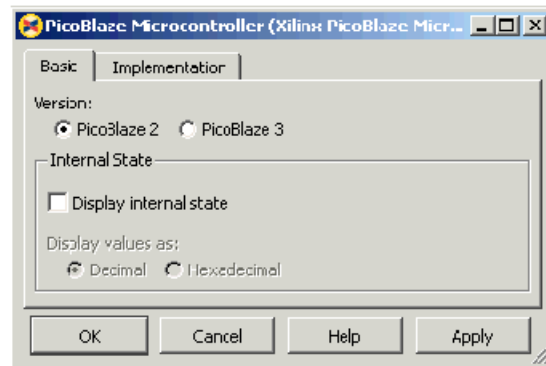
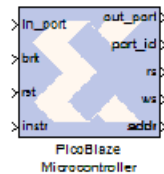
次のチュートリアルでは、割り込み中の DDS (Direct Digital Synthesizer) の出力周波数を変更するよう PicoBlaze プログラムを変更します。

Simulink モデルと PicoBlaze アセンブラ コードが提供されていますが、変更が必要です。

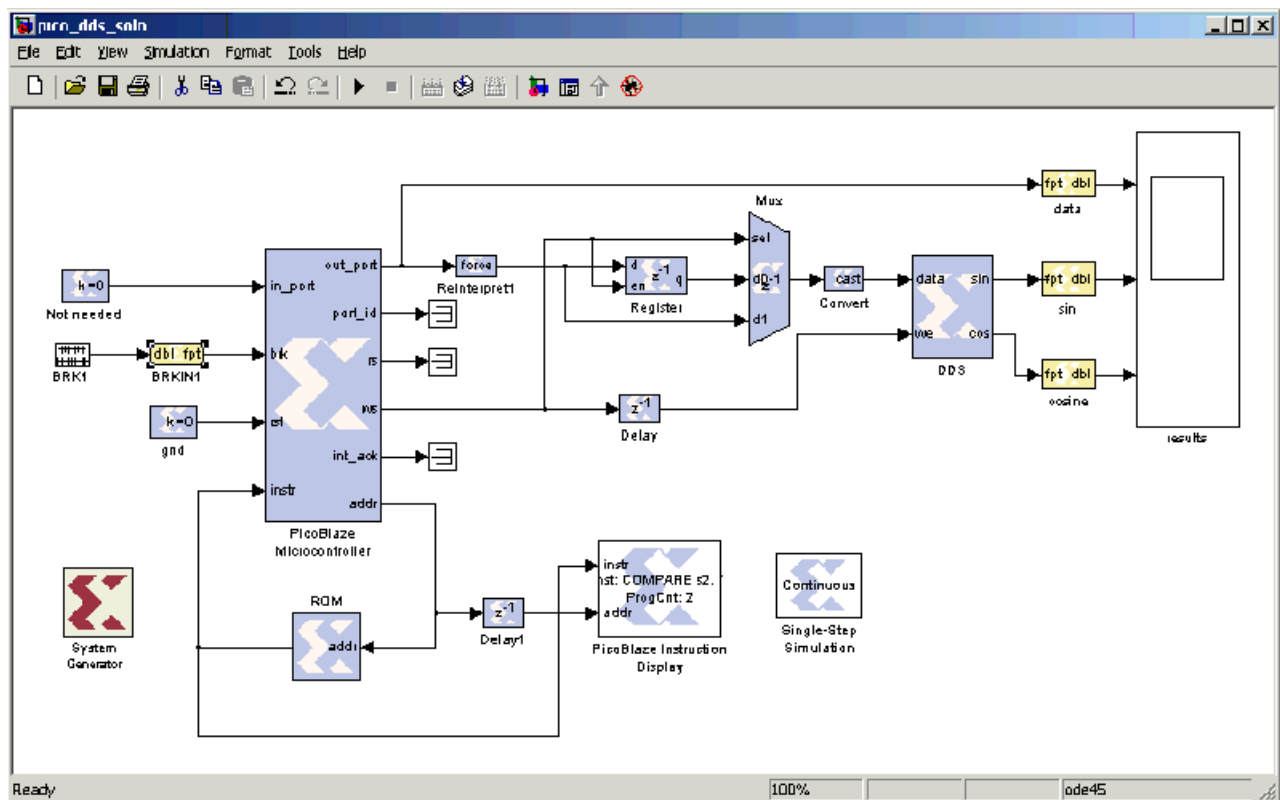
1. MATLAB ウィンドウで、<path_to_sysgen>\examples\picoblaze ディレクトリに移動します。このディレクトリには、次のファイルが含まれています。
 - ◆ Pico_dds.mdl : 未完成の Simulink モデル
 - ◆ Pico_code.psm : 未完成の PicoBlaze コード
2. Pico_dds.mdl を開きます。

3. デザインを変更します。

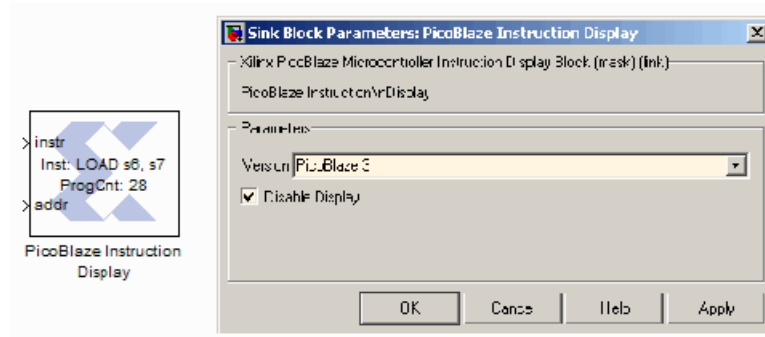
- a. [Xilinx Blockset Library] → [Index] または [Control Logic] にある PicoBlaze Microcontroller ブロックを指定の場所に追加します。デフォルト の設定では、ブロックのポート 数はモデルで必要なポート 数と異なります。これを次の手順に従って変更します。ブロックに割り 当てられたスペースに収まるよう、ブロックのサイズを変更してください。
- b. PicoBlaze Microcontroller ブロックをダブルクリックし、[Version] で [PicoBlaze 3] をオンにします。[Display internal state] はオフにします。ポートをモデルに描かれているラインに接続します。



- c. [Xilinx Blockset Library] → [Index] ライブラリ にある PicoBlaze Instruction Display ブロックを指定の場所に追加します。次の図に示すように接続されていることを確認します。



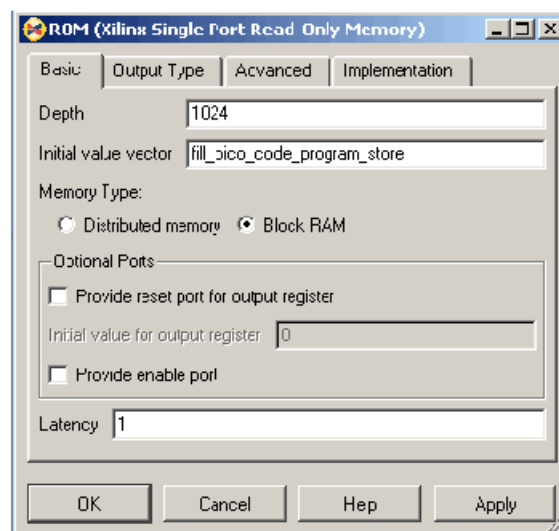
- d. PicoBlaze Instruction Display ブロックをダブルクリックし、[Version] を [PicoBlaze 3] に設定します。[Disable Display] をオンにします。このオプションをオンにすると、シミュレーションをブロック表示をアップデートせずに実行できます。



- e. [Xilinx Blockset Library] → [Index] または [Memory] ライブラリにある ROM ブロックを指定の場所に追加します。ROM ブロックを右クリックし、[書式] → [ブロックの反転] をクリックしてブロックを反転します。ポート をモデルに描かれているラインに接続します。
 - f. Single-Step Simulation ブロックをダブルクリックして、[Continuous] モードにします。
4. ROM をダブルクリックして次のように設定し、プログラムの保存をコンフィギュレーションします。

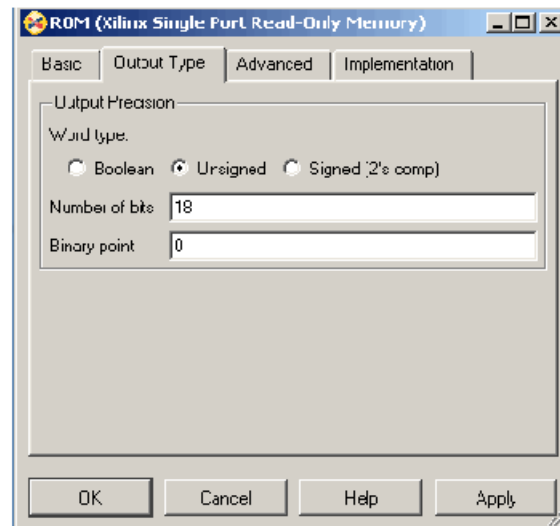
[Basic] タブで、次のように設定します。

- a. この ROM ブロックには、PicoBlaze の命令を保存します。[Depth] は 1024 に設定する必要があります。これは、プログラムで割り込みが使用され、brk を 1 にしたときにプログラム カウンタが 0x3FF に設定されるからです。
- b. 手順 5 でコードを編集し、fill_pico_code_program_store.m というメモリの初期化ファイルを生成するので、[Initial value vector] は fill_pico_code_program_store に設定します。
- c. 同期デザインのパフォーマンスを向上させるため、[Latency] は 1 に設定します。



[Output Type] タブをクリックし、次のように設定します。

- a. [Word type] で [Unsigned] をオンにし、[Number of bits] を 18、[Binary Point] を 0 に設定します。



5. PicoBlaze アセンブリ プログラムを編集します。
 - a. Pico_code.psm を開きます。
 - b. このファイルの説明に従って命令を追加します。PicoBlaze の命令セットに関する詳細情報は、アプリケーション ノート XAPP627 (http://japan.xilinx.com/support/documentation/application_notes/xapp627.pdf) を参照してください。
 - c. ファイルを保存します。
6. アセンブラを実行して、メモリ初期化ファイルを生成します。

メモ：ザイリンクス PicoBlaze アセンブラは、Windows オペレーティング システムでのみ使用可能です。サードパーティから Linux 用の PicoBlaze アセンブラが提供されていますが、ザイリンクスソフトウェアには含まれていません。

MATLAB の [Command Window] に、次のように入力します。

```
>> xlpb_as -p pico_code.psm
```

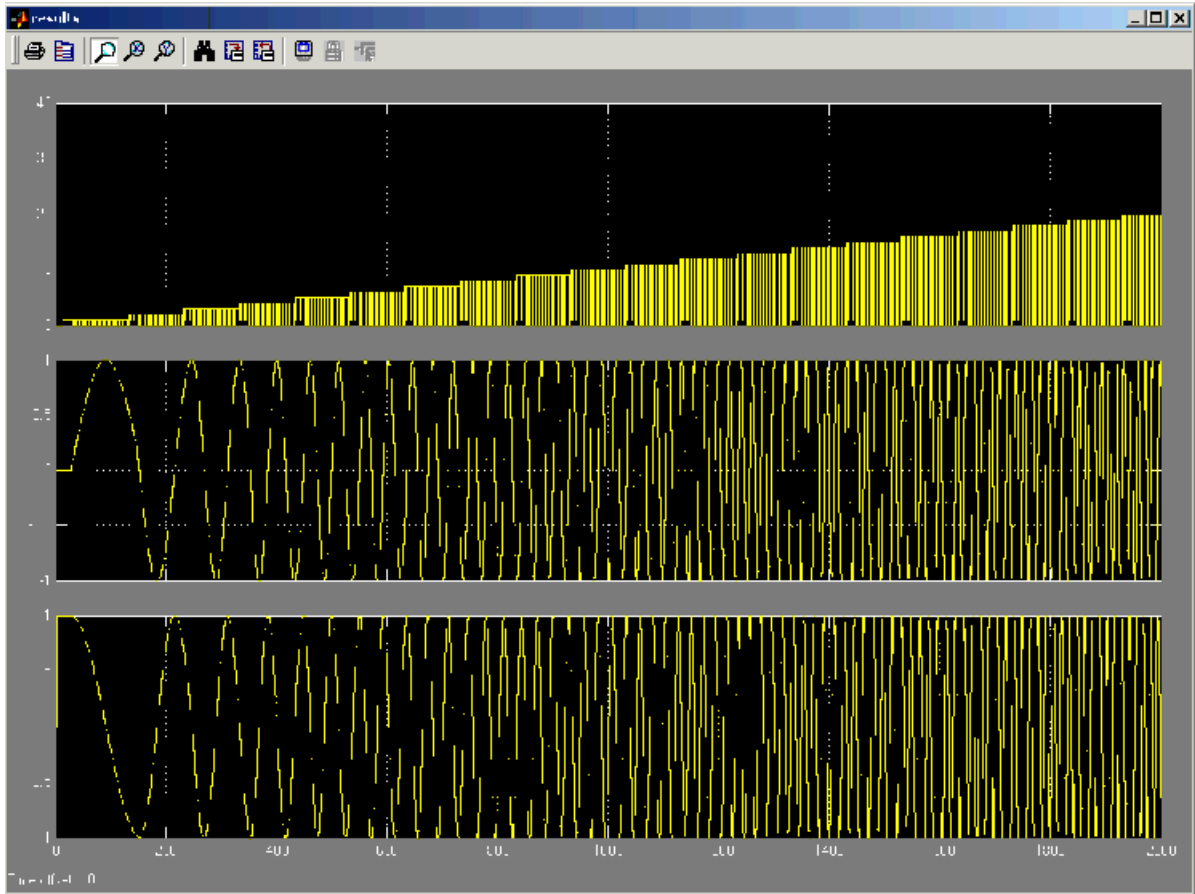
xlpb_as は、ザイリンクスの PicoBlaze アセンブラです。

fill_pico_code_program_store.m というファイルが生成されます。

7. Simulink モデルをシミュレーションします。

Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、シミュレーションを開始します。

次のような出力が表示されます。



位相の増加に比例して、波形の周波数が増加しています。

8. デバッグ ツールを使用します。

プログラムが正常に機能しない場合は、デバッグにいくつかのツールを利用できます。**PicoBlaze Instruction Display** ブロックで **[Disable Display]** をオフにすると、各クロック サイクルでプログラム カウンタと命令がアップデートされます。また、**PicoBlaze Microcontroller** ブロックで **[Display internal state]** をオンにすると、レジスタおよび制御フラグの値を表示できます。**Single-Step Simulation** ブロックをダブルクリックして **[Single-Step]** モードにすると、シミュレーションを 1 行ずつ実行してデバッグできます。

MicroBlaze プロセッサ ペリフェラルの設計とエクスポート

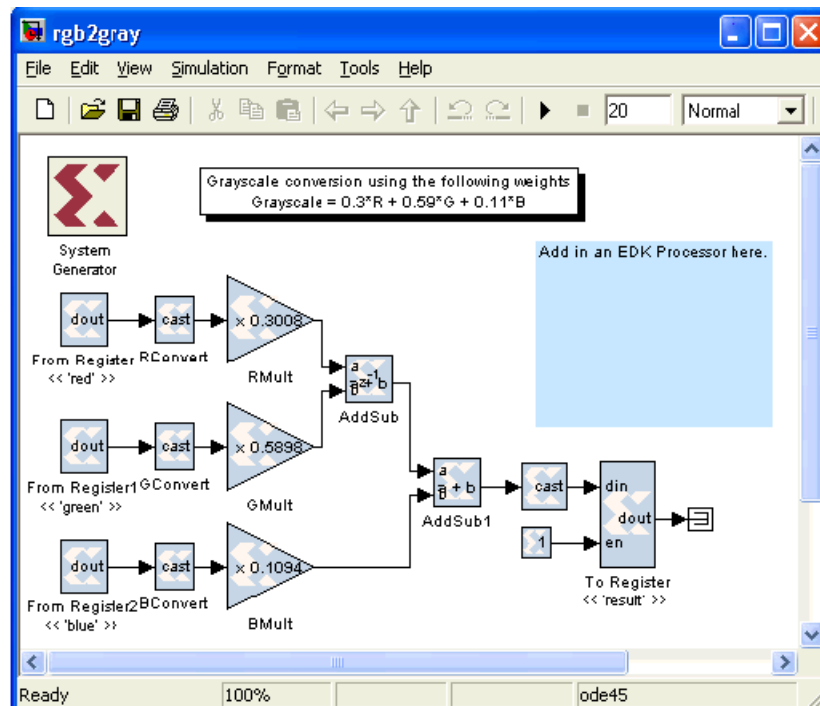
Xilinx Platform Studio (XPS) を使用すると、カスタマイズされた MicroBlaze および PowerPC プロセッサ システムを設計できます。プロセッサ システムのハードウェア ペリフェラルは **pcore** と呼ばれ、特定のディレクトリ構造で配置されたデザイン ファイルで構成されています。これらのデザイン ファイルには、XPS pcore のハードウェア インプリメンテーション、接続インターフェイス、ソフトウェア ドライバなどが記述されています。

EDK Processor ブロックで **EDK Export Tool** を設定すると、System Generator でカスタマイズされたプロセッサ ハードウェア ペリフェラルを設計できます。System Generator デザインは XPS pcore としてエクスポートし、XPS プロジェクトに含めて使用できます。

次のチュートリアルで、System Generator で XPS pcore を作成する方法を示します。このチュートリアルで使用するファイルは、<path_to_sysgen>\examples\EDK\rgb2gray (<path_to_sysgen> は System Generator のインストール ディレクトリ) に含まれています。

チュートリアル：System Generator での MicroBlaze ペリフェラルの作成

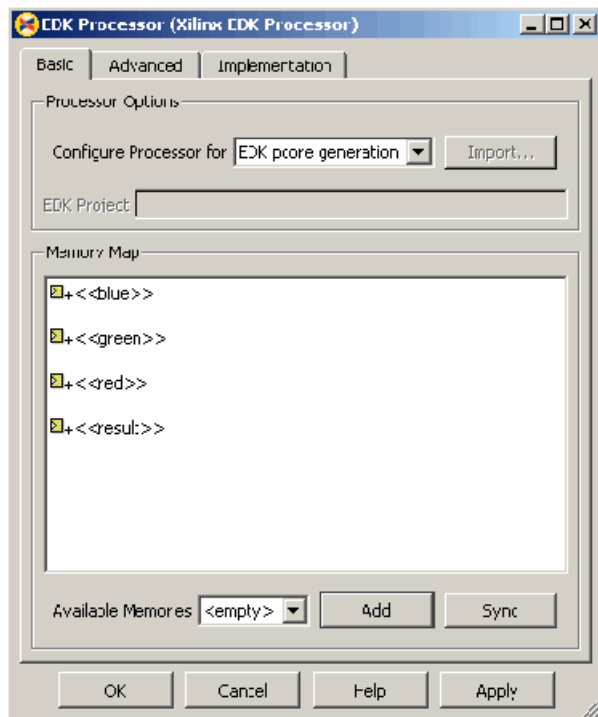
メモ：このチュートリアルを完了するには、EDK がインストールされている必要があります。



1. <path_to_sysgen>\examples\EDK\rgb2gray から rgb2gray.mdl を開きます。

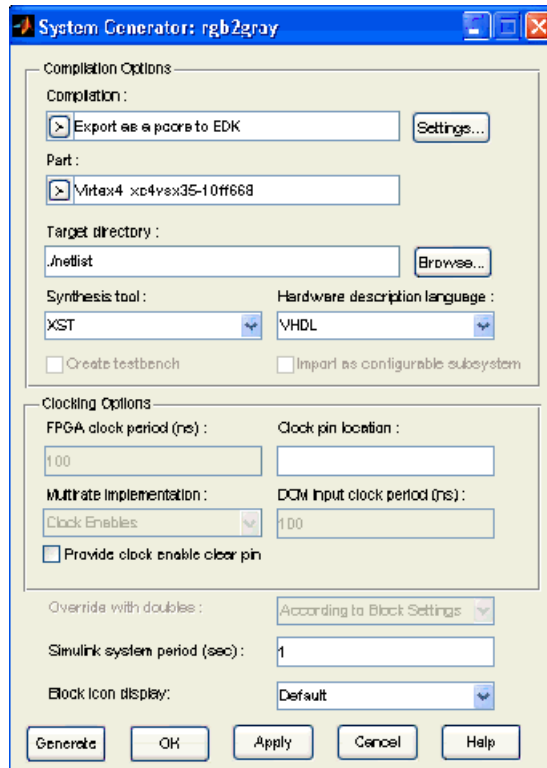
このペリフェラルには、赤、緑、青のピクセル値を表す 32 ビットの入力が 3 つあります。これらの値が 32 ビット グレースケール値に変換されます。赤、緑、青の値は、red、green、blue という名前の 3 つの共有レジスタから供給され、変換結果が result という 共有レジスタに書き込まれます。

2. pcore をエクスポートできるように準備します。EDK Processor ブロックをモデルにドラッグします。EDK Processor ブロックをダブルクリックして、次のように設定します。



[Available Memories] を [<all>] に設定し、[Add] をクリックしてモデルに含まれるすべての共有メモリを EDK Processor ブロックに追加します。[Configure processor for] を [EDK pcore generation] に設定します。[OK] をクリックします。これで、EDK Processor ブロックにより共有メモリのメモリマップが生成されます。

3. pcore を見てみます。System Generator トークンをダブルクリックして、パラメータ ダイアログ ボックスを開きます。この例では、EDK Export Tool を使用して pcore を作成します。EDK Export Tool のオプションの詳細は、「System Generator のコンパイル タイプ」を参照してください。



[Compilation] を [Export as a pcore to EDK] に設定します。[Settings] をクリックし、[EDK export settings] ダイアログ ボックスを開きます。[Export pcore to] で [System Generator target directory] をオンにし (デフォルト)、pcore がモデルのターゲット ディレクトリにエクスポートされるようにします。

[Generate] ボタンをクリックし、エクスポートを開始します。

4. エクスポートした pcore を XPS に統合します。

XPS プロジェクトを作成し、そのプロジェクトに pcore を統合します。「XPS の使用」の手順に従って、XPS プロジェクトを作成してください。

XPS プロジェクトを作成したら、System Generator で生成した pcore をローカル pcore リポジトリにコピーします。前の手順で pcore を System Generator のターゲット ディレクトリに保存するよう指定したので、ターゲット ディレクトリ内に pcore という名前のディレクトリがあるはずです。ディレクトリの内容を XPS プロジェクトの対応する pcore ディレクトリにコピーします。XPS プロジェクトに pcore ディレクトリがない場合は、作成してください。

XPS で、[Project] → [Rescan User Repositories] をクリックします。System Generator でエクスポートした rgb2gray_plbw という pcore が EDK ペリフェラルのリストに表示されます。

「XPS の使用」の手順に従い、EDK で pcore を MicroBlaze プロセッサに接続します。

pcore を接続したら、[Hardware] → [Generate Netlist] をクリックしてネットリストをコンパイルします。

ソフトウェアの記述

XPS プロジェクトでソフトウェア アプリケーションを作成します。ソフトウェア アプリケーションの作成方法の詳細は、「[XPS の使用](#)」を参照してください。アプリケーションに次のコードを追加して、ソフトウェアをコンパイルします。

```
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"

// header file of System Generator Pcore
#include "rgb2gray_plbw.h"

int main (void) {
    int i;
    uint32_t gray, red, green, blue;

    print("-- Entering main() --\n\r");

    xc_iface_t *iface;
    xc_from_reg_t *fromreg_gray;
    xc_to_reg_t *toreg_red, *toreg_green, *toreg_blue;

    // initialize the software driver
    xc_create(&iface, &RGB2GRAY_PLBW_ConfigTable[0]);

    // obtain the memory locations
    xc_get_shmem(iface, "result", (void **) &fromreg_gray);
    xc_get_shmem(iface, "red", (void **) &toreg_red);
    xc_get_shmem(iface, "green", (void **) &toreg_green);
    xc_get_shmem(iface, "blue", (void **) &toreg_blue);

    for (i=15; i<30; i++){
        red = i;
        green = i + 10;
        blue = i + 20;

        // Write RGB value to peripheral
        xc_write(iface, toreg_red->din, red);
        xc_write(iface, toreg_green->din, green);
        xc_write(iface, toreg_blue->din, blue);

        xil_printf("R = 0x%x, G = 0x%x, B = 0x%x -- ",
            red, green, blue);

        xc_read(iface, fromreg_gray->dout, &gray);

        xil_printf("Gray = %x \n\r", gray);
    }

    print("-- Exiting main() --\n\r");
    return 0;
}
```


XPS プロジェクトには System Generator pcore の複数のインスタンスを含めることができます。各インスタンスにはデバイス ID が関連付けられており、xparameter.h に記述されています。xparameter.h の次の情報から、この例のインスタンスのデバイス ID が 0 であることがわかります。

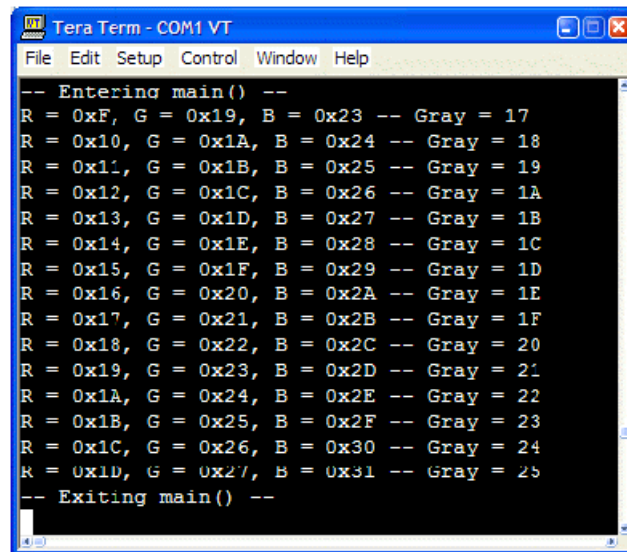
```
/* Definitions for driver SG_PLBIFACE */
#define XPAR_SG_PLBIFACE_NUM_INSTANCES 1

/* Definitions for peripheral SG_PLBIFACE_0 */
#define XPAR_SG_PLBIFACE_0_DEVICE_ID 0
```

System Generator pcore インスタンスのデバイス ID を使用して、RGB2GRAY_PLBW_ConfigTable の対応する項目を選択します。この項目は xc_create に供給され、特定の System Generator pcore インスタンスの設定情報が取得されます。

ハードウェアの接続方法、その他のソフトウェアの問題に関する詳細は、「[カスタム ロジックへのプロセッサの統合](#)」を参照してください。

コードを実行すると、RS232 ターミナルに次のように表示されます。



```
Tera Term - COM1 VT
File Edit Setup Control Window Help

-- Entering main() --
R = 0xF, G = 0x19, B = 0x23 -- Gray = 17
R = 0x10, G = 0x1A, B = 0x24 -- Gray = 18
R = 0x11, G = 0x1B, B = 0x25 -- Gray = 19
R = 0x12, G = 0x1C, B = 0x26 -- Gray = 1A
R = 0x13, G = 0x1D, B = 0x27 -- Gray = 1B
R = 0x14, G = 0x1E, B = 0x28 -- Gray = 1C
R = 0x15, G = 0x1F, B = 0x29 -- Gray = 1D
R = 0x16, G = 0x20, B = 0x2A -- Gray = 1E
R = 0x17, G = 0x21, B = 0x2B -- Gray = 1F
R = 0x18, G = 0x22, B = 0x2C -- Gray = 20
R = 0x19, G = 0x23, B = 0x2D -- Gray = 21
R = 0x1A, G = 0x24, B = 0x2E -- Gray = 22
R = 0x1B, G = 0x25, B = 0x2F -- Gray = 23
R = 0x1C, G = 0x26, B = 0x30 -- Gray = 24
R = 0x1D, G = 0x27, B = 0x31 -- Gray = 25
-- Exiting main() --
```

チュートリアル：MicroBlaze プロセッサ システムの設計とシミュレーション

このチュートリアルでは、MicroBlaze プロセッサを含む System Generator モデルの設計方法とシミュレーション方法を示します。DSP48 コプロセッサを System Generator を使用して設計します。EDK Processor ブロックを使用し、XPS でカスタマイズした MicroBlaze プロセッサを System Generator モデルにインポートします。その後、EDK Processor ブロックの自動メモリ マップ機能を使用して、インポートした MicroBlaze プロセッサに DSP48 コプロセッサを接続します。

このチュートリアルでは、ハードウェア協調シミュレーションを使用してデザインをシミュレーションおよび検証します。MicroBlaze プロセッサをハードウェアにコンパイルし、DSP48 コプロセッサ モデルは System Generator ダイアグラムに残してソフトウェア シミュレーションを実行します。このチュートリアルでは、ポイントツーポイント イーサネット協調シミュレーション手法を使用して、ハードウェア シミュレーションとソフトウェア シミュレーションを通信させます。

このチュートリアルは、次のセクションから構成されています。

- XPS プロジェクトの作成
- DSP48 コプロセッサ モデルの作成
- XPS プロジェクトのインポート
- メモリ マップ インターフェイスのコンフィギュレーション
- ソフトウェア プログラムの記述
- ハードウェア協調シミュレーション ブロックの作成
- テストベンチ モデルの作成
- コンパイルされたソフトウェアで協調シミュレーション ブロックをアップデート
- シミュレーションの実行

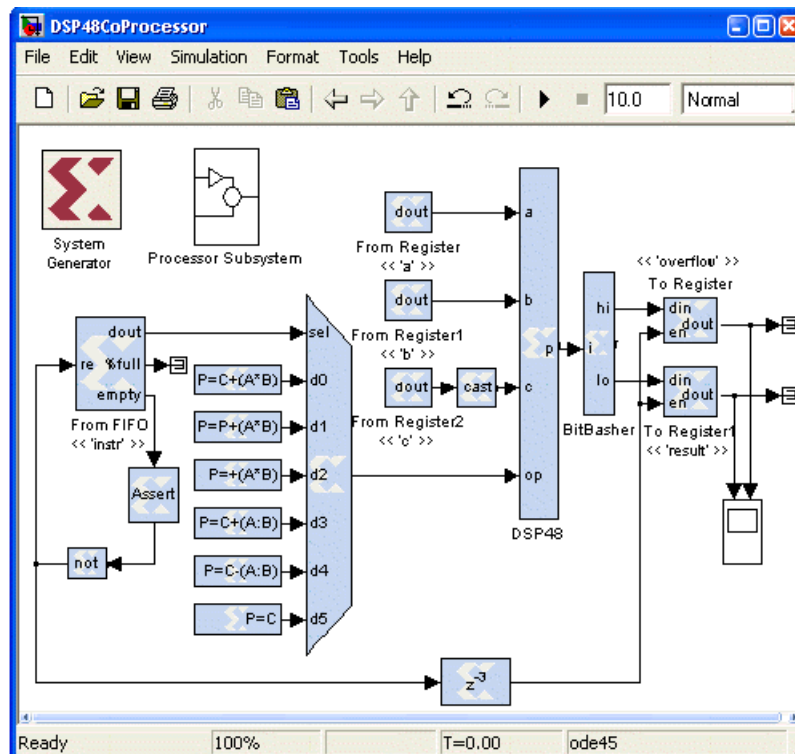
このチュートリアルでは、ザイリンクス Virtex®-4 ML402 評価プラットフォームを使用します。

このチュートリアルで使用するファイルは、<path_to_sysgen>\examples\EDK\ DSP48CoProcessor (<path_to_sysgen> は System Generator のインストール ディレクトリ) に含まれています。

XPS プロジェクトの作成

まず、PLB ベースの UART ペリフェラルを含む XPS プロジェクトを作成する必要があります。
XPS プロジェクトの作成方法は、「[XPS の使用](#)」を参照してください。

DSP48 コプロセッサ モデルの作成



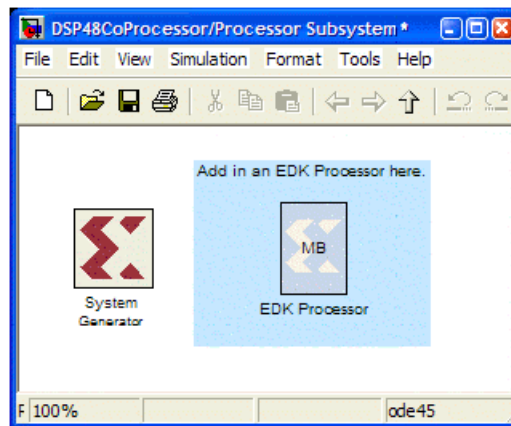
<path_to_sysgen>\examples\EDK\ DSP48CoProcessor フォルダにある
DSP48CoProcessor.mdl を一時作業ファイルにコピーし、開きます。

このモデルには DSP48 ブロックが含まれており、a、b、c ポートに対応する名前の From Register ブロックが接続されています。op ポートにはマルチプレクサから信号が供給されており、このマルチプレクサのセレクトラインには instr という名前の From FIFO ブロックから信号が共有されます。

DSP48 ブロックの出力ポート p は、BitBasher ブロックを介して 2 つに分割され、上位 16 ビットは overflow という To Register ブロック、下位 32 ビットは result という To Register に入力されます。

XPS プロジェクトのインポート

MicroBlaze プロセッサを含む XPS プロジェクトを、DSP48 コプロセッサモデルにインポートします。Processor Subsystem というサブシステムをダブルクリックして表示します。このサブシステムには、System Generator トークンと、EDK Processor ブロックのプレースホルダである青いボックスが含まれています。[Xilinx Blockset] → [Index] ライブラリから EDK Processor ブロックをサブシステムにドラッグします。サブシステムは、次の図のようになります。



EDK Processor ブロックをコンフィギュレーションして XPS プロジェクトをインポートします。インポートを実行すると、XPS プロジェクトが変更されます。インポートを開始する前に、XPS プロジェクトが XPS で開いていないことを確認してください。

EDK Processor ブロックをダブルクリックし、パラメータダイアログボックスを開きます。[Configure processor for] を [HDL netlisting] に設定します。この設定により、[Import] ボタンが有効になります。

[EDK pcore generation] を設定した場合、[Import] ボタンは無効になります。[EDK pcore generation] モードは、System Generator で pcore を作成してエクスポートし、別の XPS プロジェクトで使用する場合に使用します。この場合、EDK Processor ブロック内にプロセッサインスタンスは含まれません。[HDL netlisting] モードは、XPS プロジェクトを System Generator モデルにインポートし、その他の System Generator ブロックと共にネットリストを作成する場合に使用します。

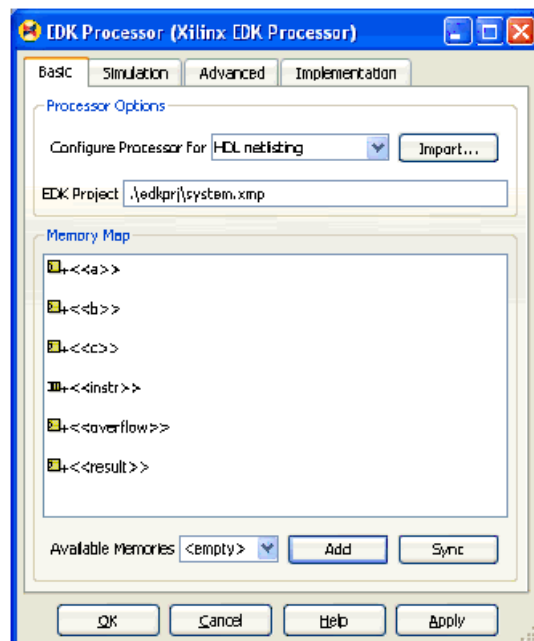
XPS プロジェクトをインポートしない場合、[HDL netlisting] に設定することにより EDK Import Wizard が自動的に起動されます。[Import] をクリックすると、EDK Import Wizard を手動で起動できます。

[Import EDK project] ダイアログ ボックスで、前の手順で作成した XPS プロジェクトを選択します。XPS プロジェクト (XMP ファイル) を選択すると、インポート プロセスが開始します。インポート プロセスでは、必要なファイルが XPS プロジェクトにコピーされ、MicroBlaze プロセッサが System Generator モデルと通信できるようにプロジェクトが変更されます。

インポートされた XPS プロジェクトにソフトウェア アプリケーションが含まれている場合、これらのアプリケーションはインポート中にコンパイルされません。

メモリ マップ インターフェイスのコンフィギュレーション

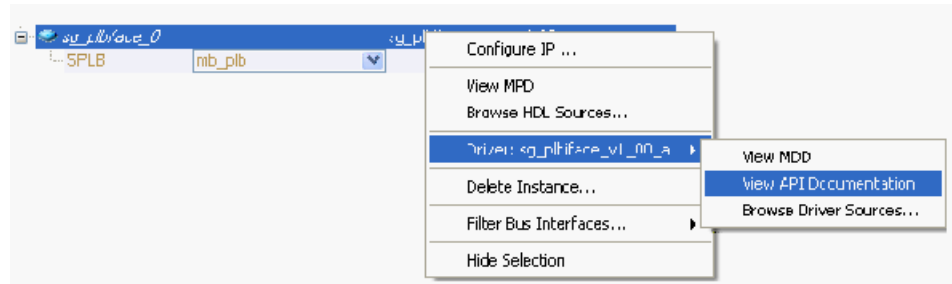
EDK Processor ブロックのパラメータ ダイアログ ボックスを開きます。[Available memories] で [<all>] を選択し、[Add] をクリックして、モデル内のすべての共有メモリをプロセッサのメモリ マップに追加します。EDK Processor ブロックのパラメータ ダイアログ ボックスは、次のようになります。[OK] をクリックします。



ソフトウェア プログラムの記述

MicroBlaze プロセッサで実行するソフトウェア プログラムを記述します。このソフトウェア プログラムは、共有メモリから読み出し、共有メモリに書き込みます。XPS で XPS プロジェクトを開きます。MyProject というソフトウェア アプリケーションを作成します。[Mark to Initialize BRAMs] オプションを、MyProject ではオンにし、その他のソフトウェア アプリケーションではオフにします。EDK プロジェクトにソフトウェア アプリケーションを追加する方法は、「[XPS の使用](#)」を参照してください。

MyProject のソース コード ファイル MyProject.c を作成し、XPS コード エディタで開きます。



上図は、XPS で開いた XPS プロジェクトの [System Assembly View] タブの一部を示しています。System Generator のインポート が正常に完了すると、sg_plbiface ペリ フェラルが XPS プロジェクト に自動的に追加されます。sg_plbiface ペリ フェラルは、インポートされた MicroBlaze プロセッサに接続されている PLB バスをメモリ マップ インターフェイスを介して System Generator モデルに接続し、対応するデバイス ソフトウェアドライバを生成するための情報を取得します。[System Assembly View] タブで sg_plbiface を右クリックし、[Driver] → [View API Documentation] をクリックして API 資料を開きます。

API 資料を参照して MyProject.c に次のヘッダ ファイルを追加し、ソフトウェアドライバを初期化します。

```
#include "sg_plbiface.h"

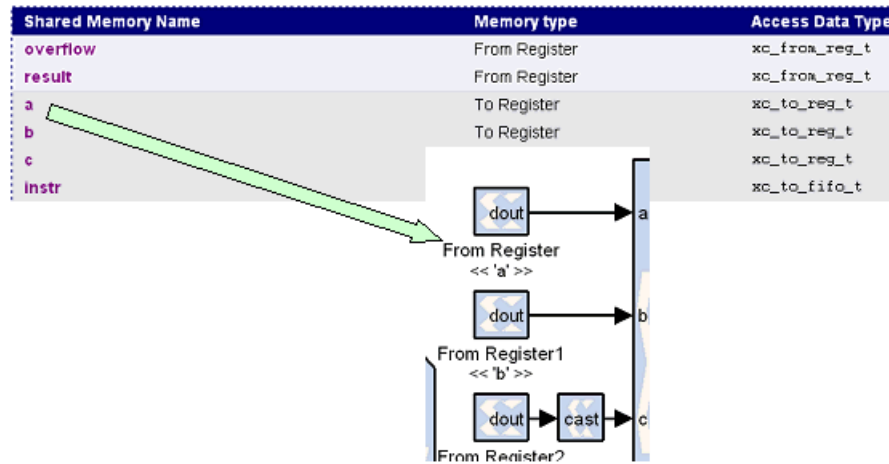
xc_iface_t *iface;

// initialize the software driver
xc_create(&iface, &SG_PLBIFACE_ConfigTable[0]);
```

プロセッサで実行するコードを確認する前に、モデルの a レジスタにデータを書き込む方法を考えます。DSP48 コプロセッサ モデルを見ると、DSP48 ブロックの a ポートは、同じ名前の共有レジスタの出力で駆動されています。この共有レジスタに、MicroBlaze プロセッサ コード内から値を書き込みます。ドライバ API を参照すると、a という共有メモリは xc_to_reg_t アクセス データ型を使用する To Register で、次のデータ フィールドを含むことがわかります。

```
typedef struct {
    xc_w_addr_t din;
    uint32_t n_bits;
    uint32_t bin_pt;
} xc_to_reg_t;
```

ソフトウェア ドライバが初期化されると、din に共有メモリ a の din ポートのメモリ マップ アドレスが保存され、n_bits および bin_pt にビット数と 2 進小数点情報が保存されます。



共有レジスタ a に値を書き込むには、まず xc_get_shmem を使用してその設定を取得する必要があります。

```
xc_to_reg_t *toreg_a;
xc_get_shmem(iface, "a", (void **) &toreg_a);
```

メモ : xc_get_shmem は、時間がかかります。返される toreg_a を後で使えるようキャッシュし、プログラムで xc_get_shmem を複数回呼び出す必要がないようにしてください。

共有レジスタ a に値を書き込むには、次の 1 ワード アクセス関数を使用します。

```
// -- Set the a port register to 2
xc_write(iface, toreg_a->din, 2);
```

MyProject.c の完全なコードは、次の場所にあります。

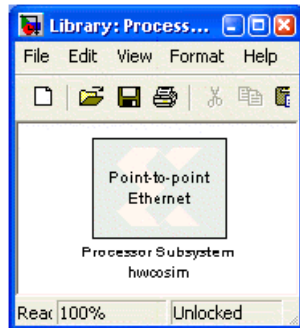
```
<path_to_sysgen>\examples\EDK\DSP48CoProcessor\MyProject.c
```

ハードウェア協調シミュレーション ブロックの作成

完全な Simulink モデルは、ハードウェア協調シミュレーションを使用してシミュレーションできます。EDK Processor ブロックのパラメータ ダイアログ ボックスの [Basic] タブで [Memory Map] に共有メモリを追加し、[Configure processor for] で [HDL netlisting] を選択していることを確認してください。ハードウェア協調シミュレーションでは、これらの設定が必要です。

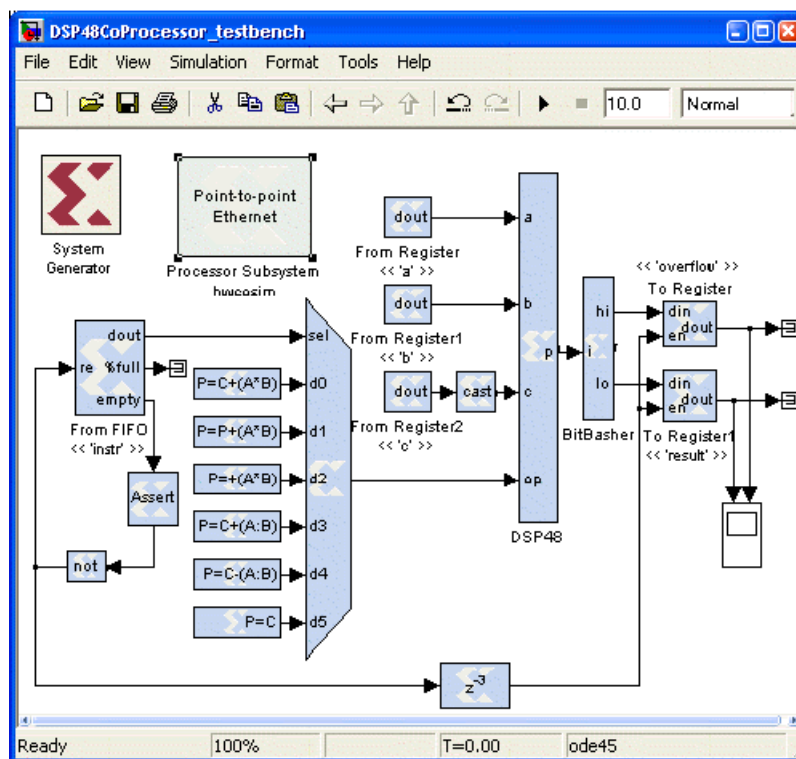
EDK Processor ブロックを含むサブシステムにある System Generator トークンのパラメータ ダイアログ ボックスを開きます。ハードウェア協調シミュレーション ブロックはモデルのこのレベルで作成し、インポートされた MicroBlaze プロセッサのみがハードウェアで実行され、その他の部分は Simulink に保持してソフトウェア シミュレーションが実行されるようにします。

[Compilation] で [Hardware Co-Simulation] → [ML402] → [Ethernet] → [Point-to-point] を選択します。[Generate] ボタンをクリックし、コンパイルを開始します。この処理には、多少時間がかかる場合があります。コンパイルが完了すると、MicroBlaze プロセッサを含むハードウェア協調シミュレーションブロックが作成されます。

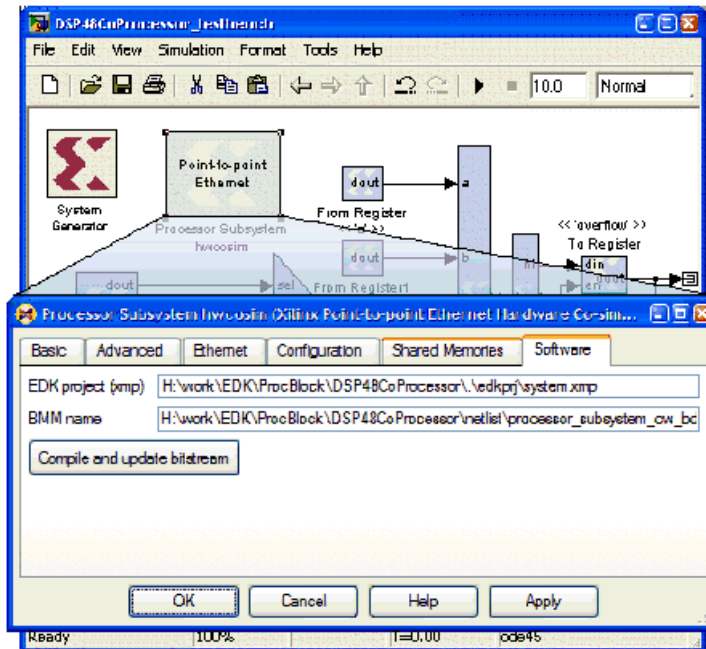


テストベンチ モデルの作成

作成した協調シミュレーションブロックを使用するためのテストベンチ モデルを作成します。DSP48CoProcessor.mdl を開き、Processor Subsystem を削除して、先ほど作成した Processor Subsystem hwcosim ブロックを追加します。モデルを DSP48CoProcessor_testbench.mdl という名前で保存します。



コンパイルされたソフトウェアで協調シミュレーション ブロックをアップデート



テストベンチ モデルを開きます。Processor Subsystem hwcosim ブロックをダブルクリックし、パラメータ ダイアログ ボックスを開きます。[Software] タブで [Compile and update bitstream] をクリックし、リストされている XPS プロジェクトに含まれるソフトウェアをコンパイルして、ハードウェア協調シミュレーション ビットストリームに読み込みます。

ポイントツーポイント イーサネット協調シミュレーションを選択しているため、イーサネット インターフェイスと Processor Subsystem hwcosim ブロックのコンフィギュレーション インターフェイスをコンフィギュレーションする必要があります。イーサネット通信に有効なホスト インターフェイスを選択し、コンフィギュレーション インターフェイスをポイントツーポイント イーサネットに設定します。ハードウェア協調シミュレーションブロックの使用の詳細は、「[ハードウェア協調シミュレーションの使用](#)」を参照してください。

シミュレーションの実行

シミュレーションを開始する前に、コンピュータの COM ポートに接続するターミナルを設定する必要があります。これにより、テキスト入力および出力を RS232 ポートを介して MicroBlaze プロセッサで読み出しおよび書き込みできるようになります。

使用するターミナル プログラムを開きます。Windows には、ハイパーターミナルというアプリケーションが含まれています。このアプリケーションを開くには、[スタート] → [すべてのプログラム] → [アクセサリ] → [通信] → [ハイパーターミナル] をクリックします。ターミナル プログラムを、RS232 に接続した COM ポートと通信するよう設定します。

ターミナルを次のように設定します。

- ボー レート = 115200
- データ = 8 ビット
- パリティ = なし
- 停止 = 1 ビット
- フロー制御 = なし

テストベンチ モデルのシミュレーション時間を `inf` に設定し、シミュレーション時間が MicroBlaze プロセッサが動作を開始して応答するのに十分であるようにします。

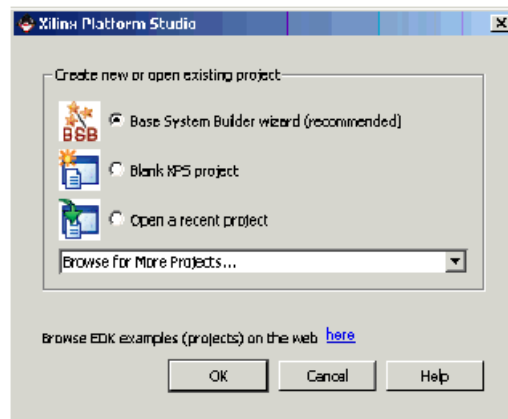
XPS の使用

このセクションでは、[ザイリンクスのエンベデッド開発キット \(EDK\)](#) のいくつかの機能について簡単に説明します。詳細な説明およびチュートリアルは、EDK の資料を参照してください。

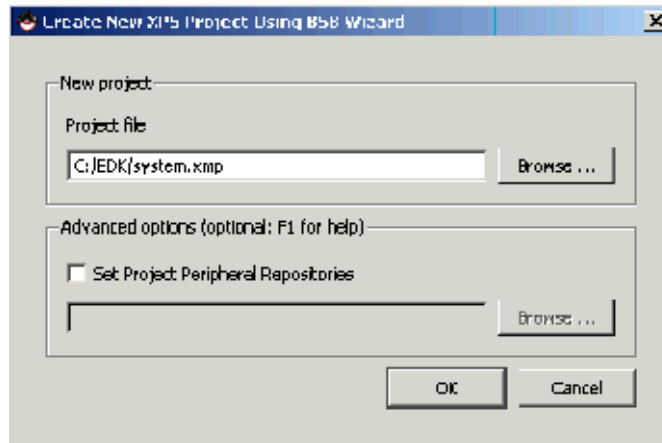
チュートリアル：新規 XPS プロジェクトの作成

EDK の Base System Builder を使用すると、完全にコンフィギュレーションされた EDK プロジェクトを簡単に作成できます。このチュートリアルでは、ザイリンクス ML402 ハードウェア開発ボードで実行する MicroBlaze プロセッサを含む EDK プロジェクトを作成します。

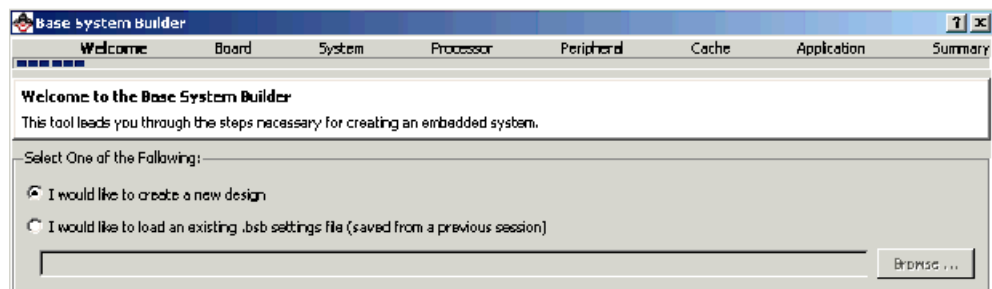
1. Windows の [スタート] メニューから XPS を起動します。
2. XPS を起動すると、次のダイアログ ボックスが表示されます。[Base System Builder wizard (recommended)] をオンにし、[OK] をクリックします。



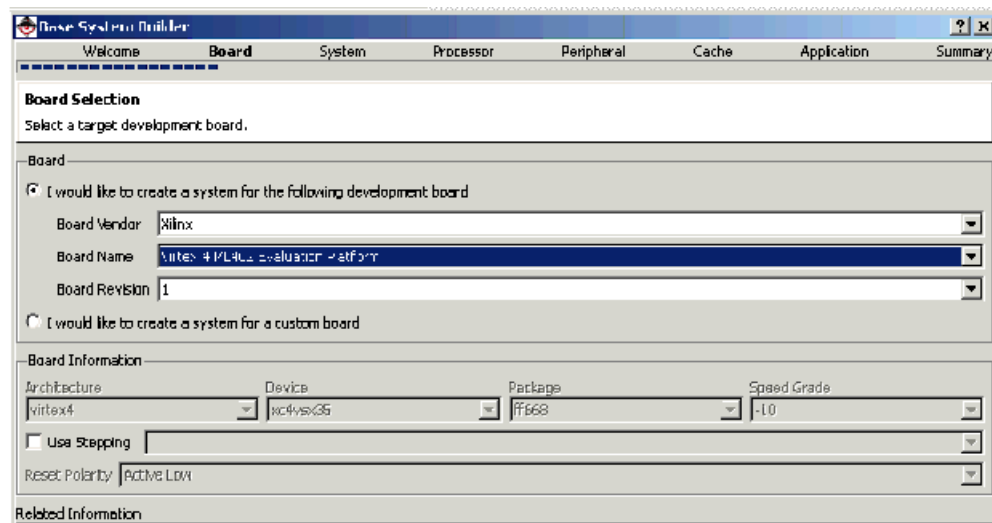
3. プロジェクト ファイル名を `system.xmp`、保存場所を次の図に示すように指定し、[OK] をクリックします。



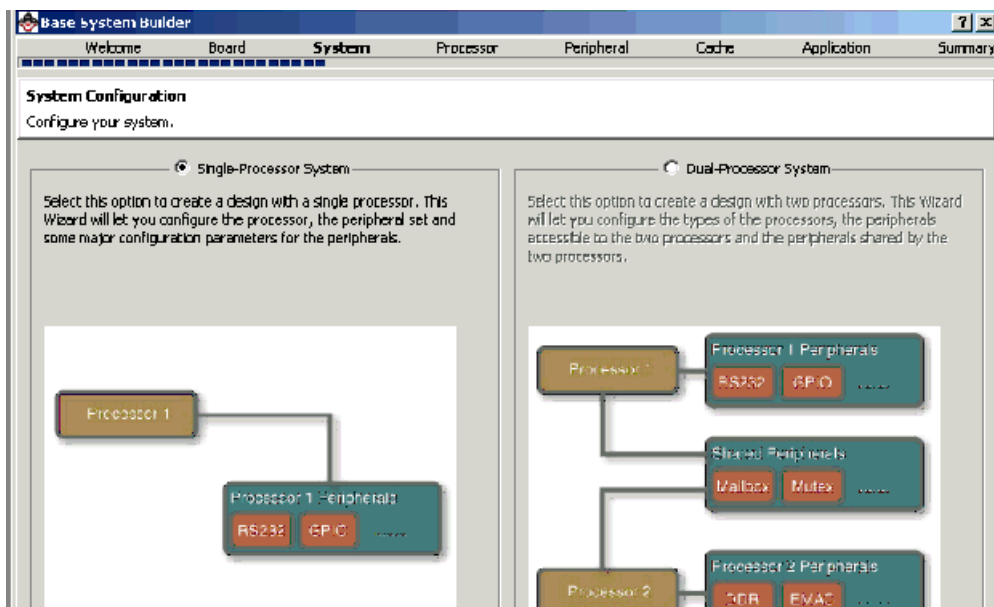
4. 次の画面で [I would like to create a new design] をオンにして新規デザインを作成することを指定し、[Next] をクリックします。



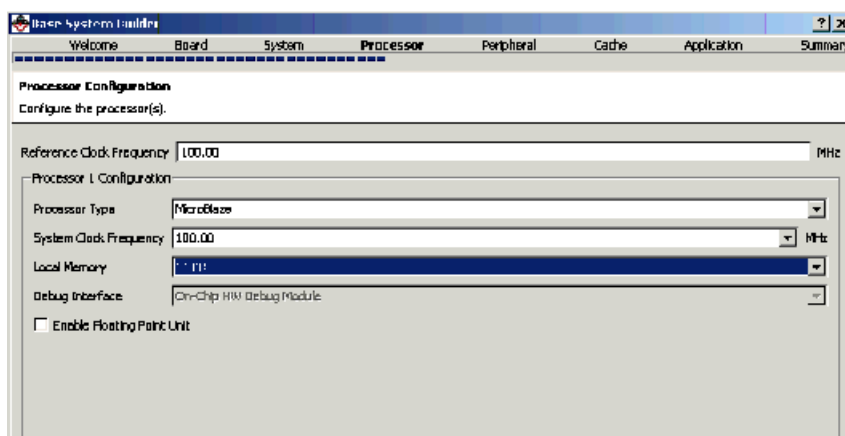
5. [Board Selection] ページで次の図に示すようにボード ベンダー ([Board Vendor]) およびボード名 ([Board Name]) を選択し、[Next] をクリックします。



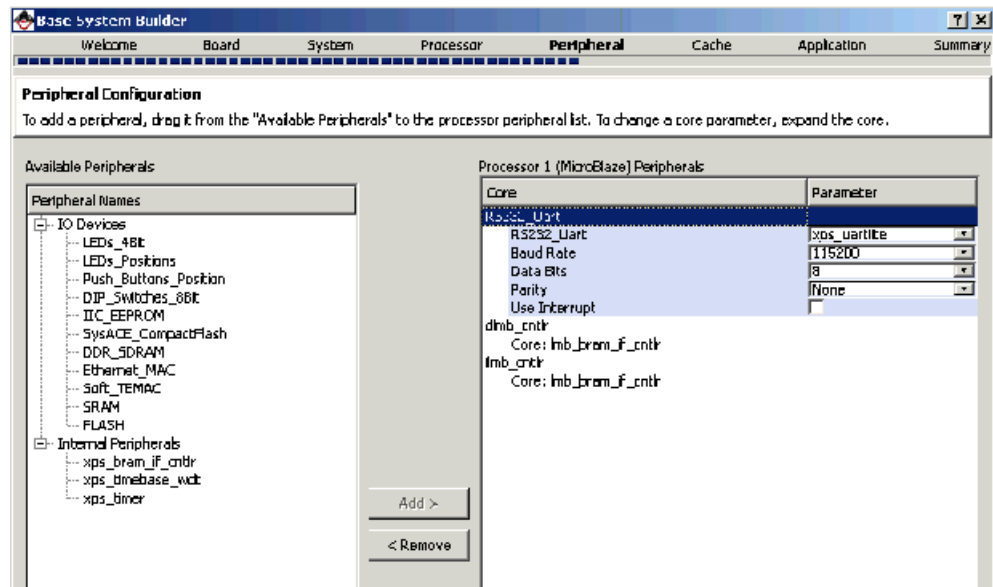
6. [System Configuration] ページで [Single-Processor System] をオンにし、[Next] をクリックします。



7. [Processor Configuration] ページで次の図に示すようにリファレンス クロックの周波数 ([Reference Clock Frequency]) とローカル メモリ ([Local Memory]) を設定し、[Next] をクリックします。



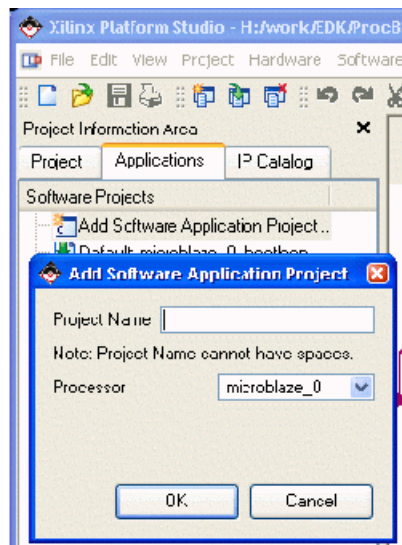
8. [Peripheral Configuration] ページで RS232_Uart、dlmb_cntlr、および ilmb_cntlr を選択します。これ以外のペリフェラルは、[Remove] をクリックして削除します。完了したら、[Next] をクリックします。



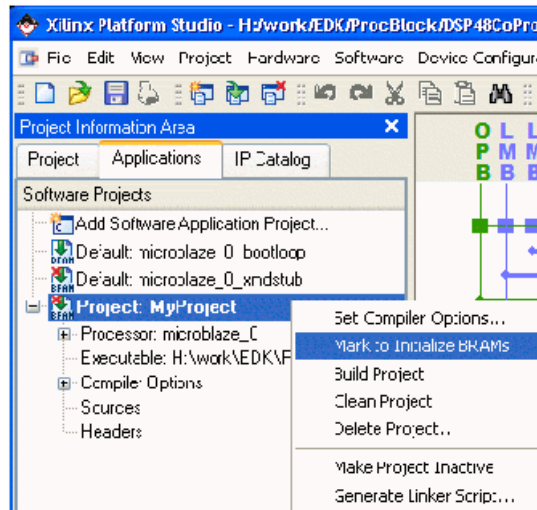
9. [Cache Configuration] ページで [Next] をクリックします。
 10. [Application Configuration] ページで [Next] をクリックします。
 11. [Summary] ページで [Finish] をクリックします。XPS プロジェクトが作成されます。

ソフトウェア アプリケーションの追加

1. EDK プロジェクトに新しいソフトウェア アプリケーションを追加するには、EDK でプロジェクトを開きます。
 2. [Project Information Area] で、[Applications] タブをクリックします。



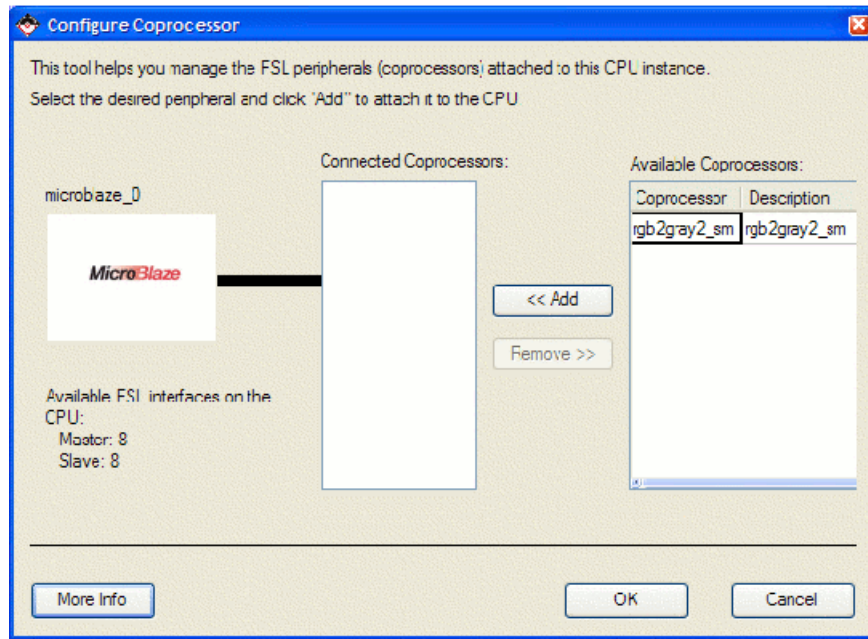
3. [Add Software Application Project] をダブルクリックし、[Add Software Application Project] ダイアログ ボックスを開きます。プロジェクト名を入力し、[OK] をクリックします。
4. デフォルトでは、BRAM に初期化されるよう設定されていません。[Mark to Initialize BRAMs] をオンにして、BRAM にプロジェクトが初期化されるようにしてください。このオプションをオフのままにすると、ソフトウェア コードはコンパイルされず、ビットストリームに追加されません。複数のアプリケーションがある場合は、その他のアプリケーションでは [Mark to Initialize BRAMs] をオフにしてください。



5. ソース ファイルまたはヘッダ ファイルを作成します。[Project] ツリーの下にある [Source] をダブルクリックし、[Select Source/Header File to Add to Project] ダイアログ ボックスを開きます。EDK プロジェクトのディレクトリが表示されています。プロジェクトを表す名前のディレクトリ (この場合は MyProject) を作成し、ソース ファイルおよびヘッダ ファイルをそのディレクトリに保存します。XMP ファイルと同じディレクトリにディレクトリを作成します。

EDK プロジェクトへの pcore の追加

1. EDK プロジェクトの pcore は、ユーザー レポジトリ、または EDK プロジェクト ファイルと同じディレクトリ レベルにある pcores ディレクトリに配置する必要があります。
2. pcore が確実に読み込まれるようにするには、XPS で [Project] → [Rescan User Repositories] をクリックします。
3. System Generator の pcore は FSL ベースであるので、Configure Coprocessor ツールを使用できます。Configure Coprocessor を起動するには、[Hardware] → [Configure Coprocessor] をクリックします。



使用可能な FSL ベースの pcore が、右側のボックスに表示されます。使用する pcore を選択し、[Add] をクリックします。Configure Coprocessor ツールでは、FSL バスのクロック信号とリセット信号が接続されますが、ユーザー信号は手動で接続する必要があります。

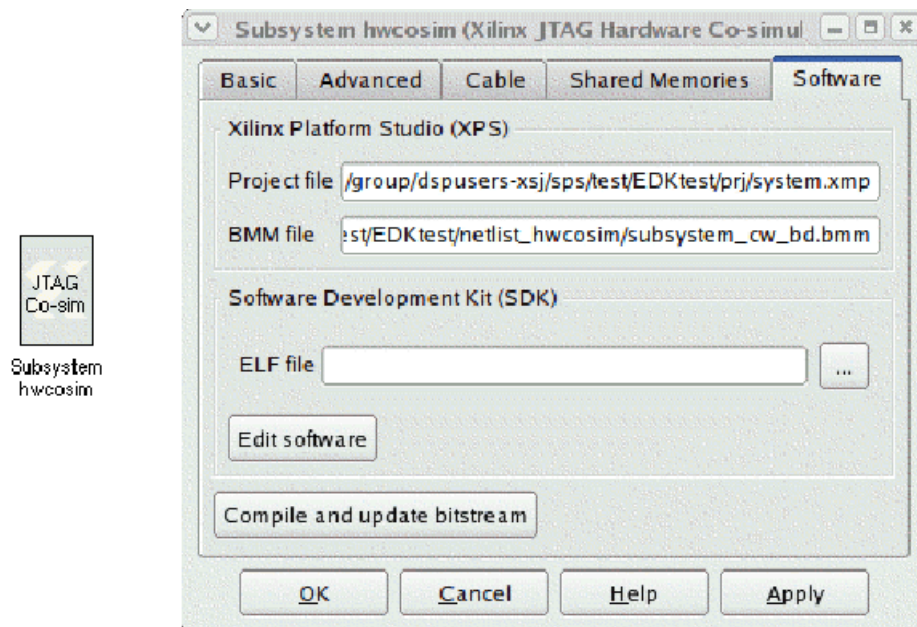
Platform Studio SDK の使用

概要

ザイリンクス ソフトウェア開発キット (SDK) は、ソフトウェアプラットフォーム デザインを作成するための Eclipse ベースの統合開発環境で、ザイリンクス エンベデッド プロセッサ用に高質の C/C++ コードを記述できます。System Generator から SDK に直接アクセスでき、SDK ワークスペースが自動的に生成されて、サンプルコードを含む Hello World プログラム テンプレートが表示されます。このサンプルコードを利用して、機能するコードを短時間に記述できます。

SDK の起動

System Generator では、ハードウェア協調シミュレーション用のコンパイルを実行した際に EDK Processor ブロックが含まれていると、自動的に SDK ワークスペースが生成されます。また、ハードウェア協調シミュレーションブロックのパラメータ ダイアログ ボックスには [Software] というタブがあり、ここから SDK にアクセスできます。



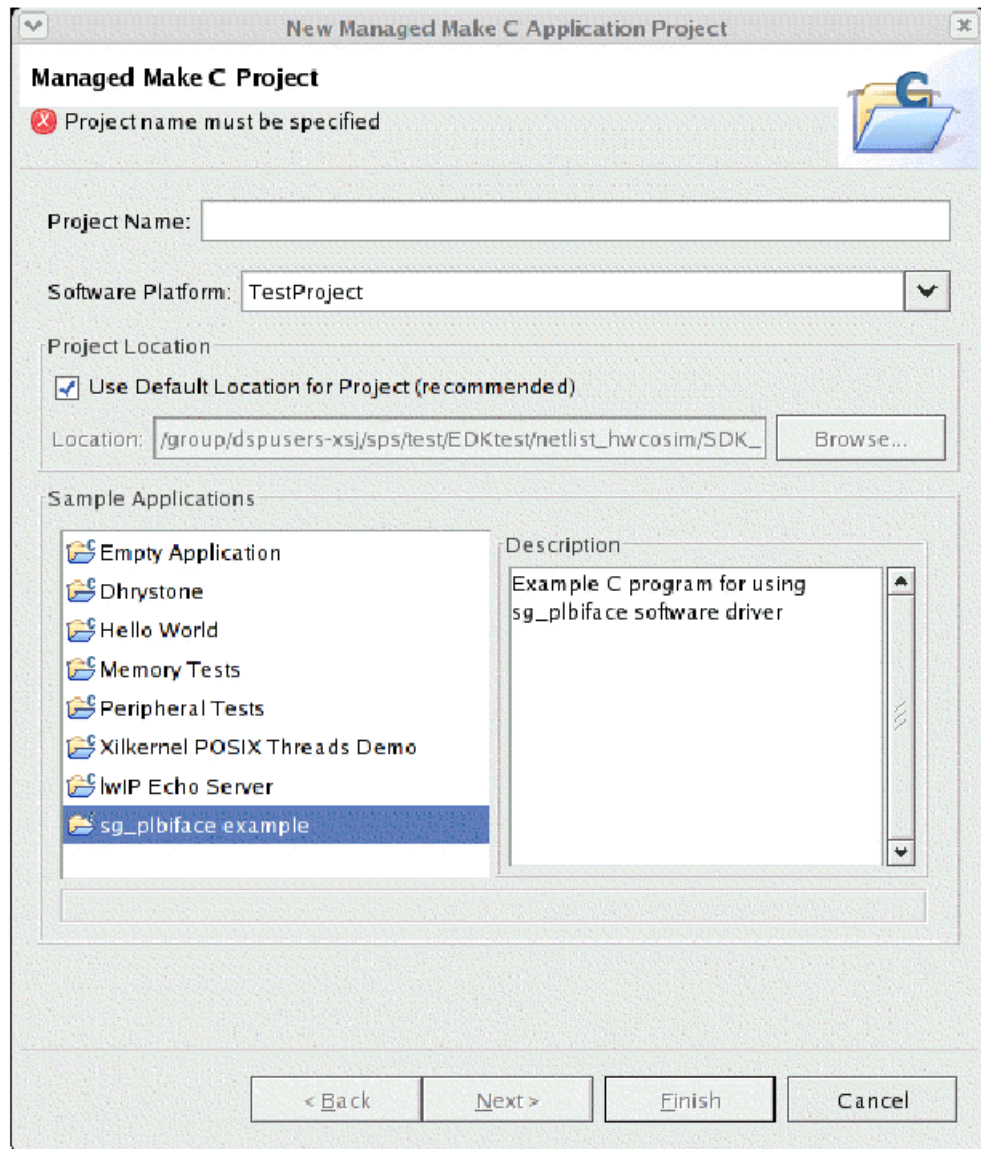
[Edit software] ボタンをクリックすると SDK が起動します。SDK ワークスペースは、デザインの ネットリスト ディレクトリの下にもあります。[ELF file] では、ハードウェア協調シミュレーション ブロックでシミュレーション中に使用する実行バイナリ ファイルを指定します。[Compile and update bitstream] をクリックすると、[ELF file] で指定したバイナリ ファイルがビットストリームに組み込まれます。

SDK でのソフトウェアプラットフォームの作成および Managed C または C++ プロジェクトの作成については、SDK の資料を参照してください。

SDK での Hello World アプリケーションの作成

System Generator で作成されたワークスペースで SDK を起動したら、System Generator のメモリに対する読み出しおよび書き込みの実行方法を示すサンプル コードを含む C アプリケーション プロジェクトを作成できます。

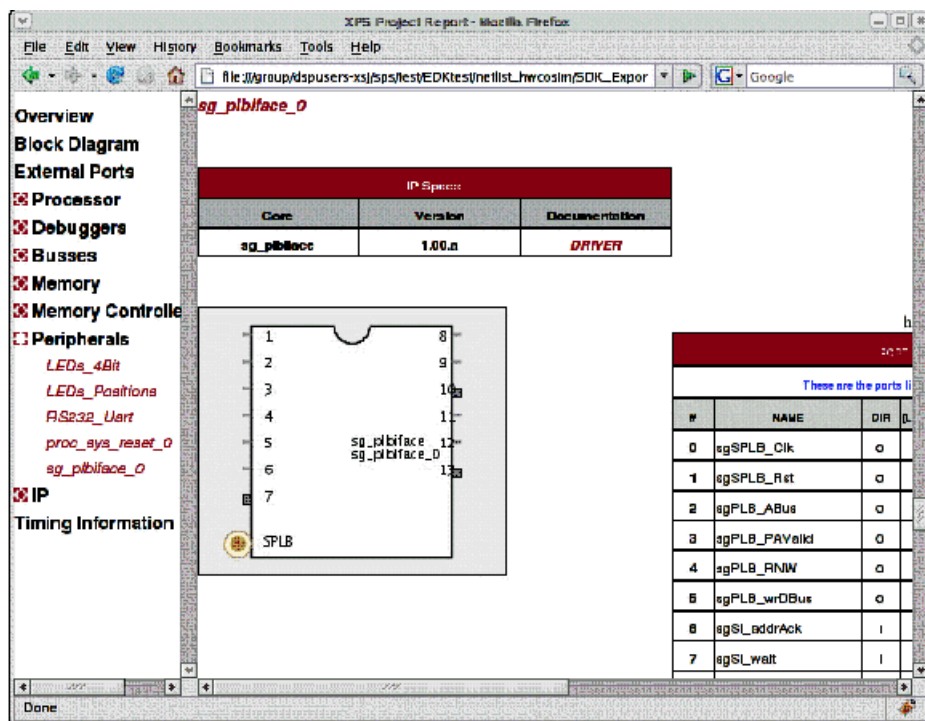
[C/C++ Project] ビューでソフトウェア プラットフォームを右クリックし、[New] → [Managed Make C Application Project] をクリックします。



[Sample Applications] から [sg_plbiface example] を選択し、[Finish] をクリックします。これにより、「Hello World」と表示する main() ルーチンを含むソフトウェアプロジェクトが作成されます。このファイルには、System Generator デザインのメモリにアクセスする方法を示すサンプル関数も含まれています。

System Generator で生成されたソフトウェア ドライバに関するヘルプの表示

SDK メニューで [Hardware Design] → [View Design Report] をクリックします。ソフトウェアプラットフォームで使用可能なリソースを含む Web ページが表示されます。左側のナビゲーションパネルで、[Peripherals] → [sg_plbiface_0] をクリックします。



レポートの System Generator ペリフェラルに関するセクションが表示されます。[IP Specs] 表で [DRIVER] リンクをクリックすると、System Generator でペリフェラル用に生成された文書が表示されます。

XPS からスタンドアロン SDK へのソフトウェア プロジェクトの移行

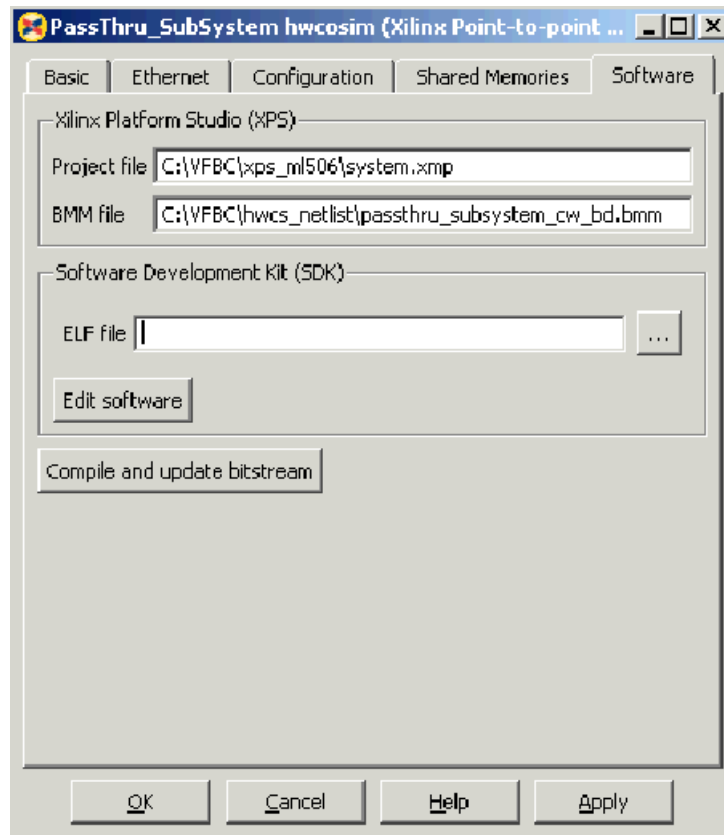
XPS でのソフトウェア アプリケーションの追加および管理は廃止予定であり、12.1 リリース以降はサポートされなくなります。11 リリースでは既存の XPS プロジェクトを引き続き使用できますが、ソフトウェア部分をスタンドアロン SDK フローに移行し、SDK ソフトウェアの追加および管理のアドバンス機能を活用することをお勧めします。既存の XPS フローを SDK デザインフローに移行する手順を次に示します。その他の System Generator/EDK デザインでも、同じ手順を使用して XPS とスタンドアロン SDK の間でデザインをやり取りできます。

スタンドアロン SDK フローへの移行方法

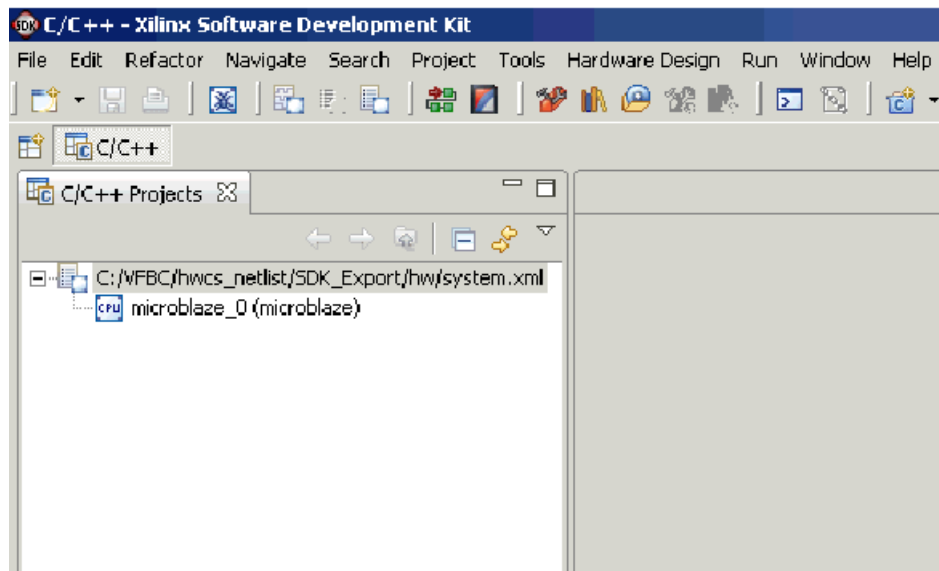
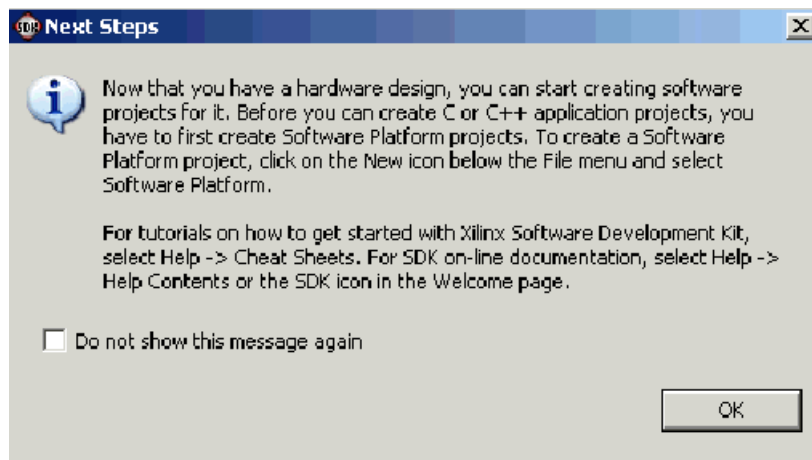
次の説明では、VFBC というデザインを使用します。

1. ハードウェア協調シミュレーション ブロックを含む System Generator モデル (vfbc_hwcs.mdl) を開きます。ハードウェア協調シミュレーション ブロックをダブルクリックし、[Edit software] をクリックして SDK を起動します。

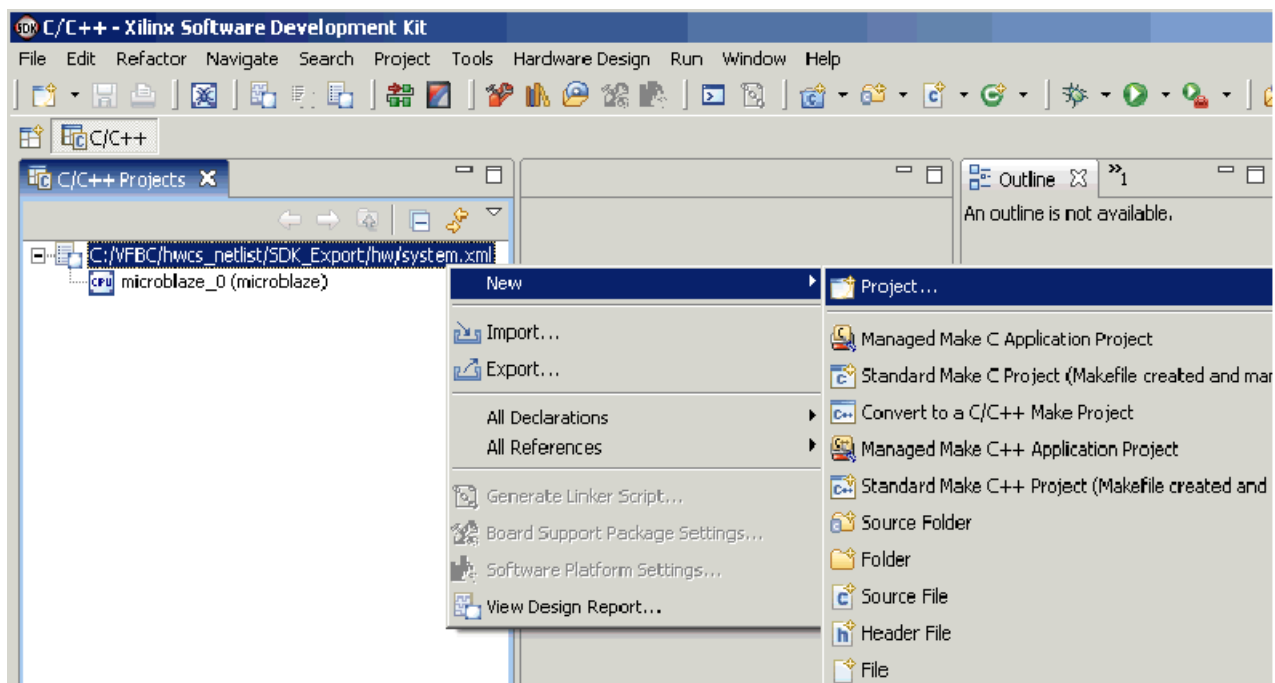
メモ：この時点では、ELF ファイルを入力する必要はありません。



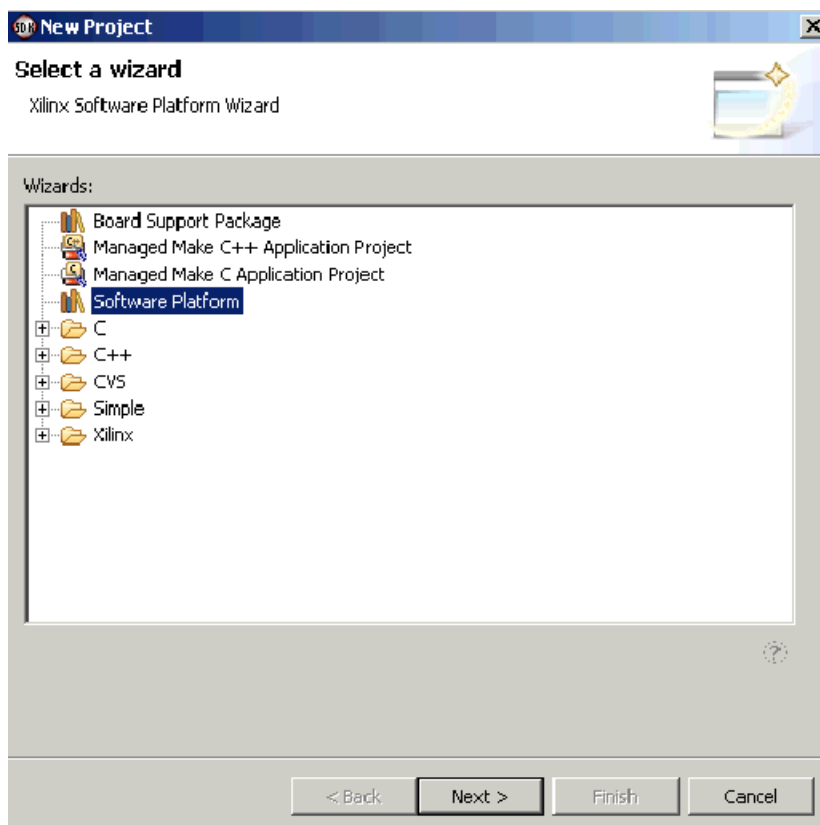
2. [OK] をクリックして、ソフトウェア プロジェクトの作成を開始します。



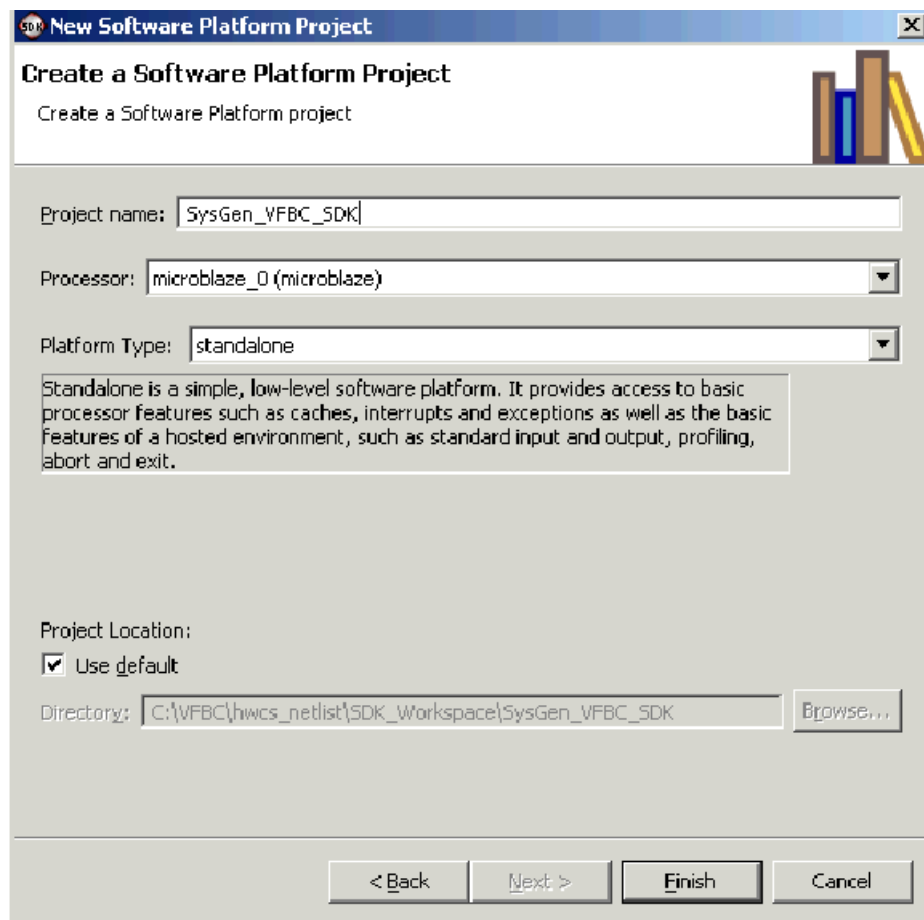
3. system.xml ファイルを右クリックし、[New] → [Project] をクリックします。



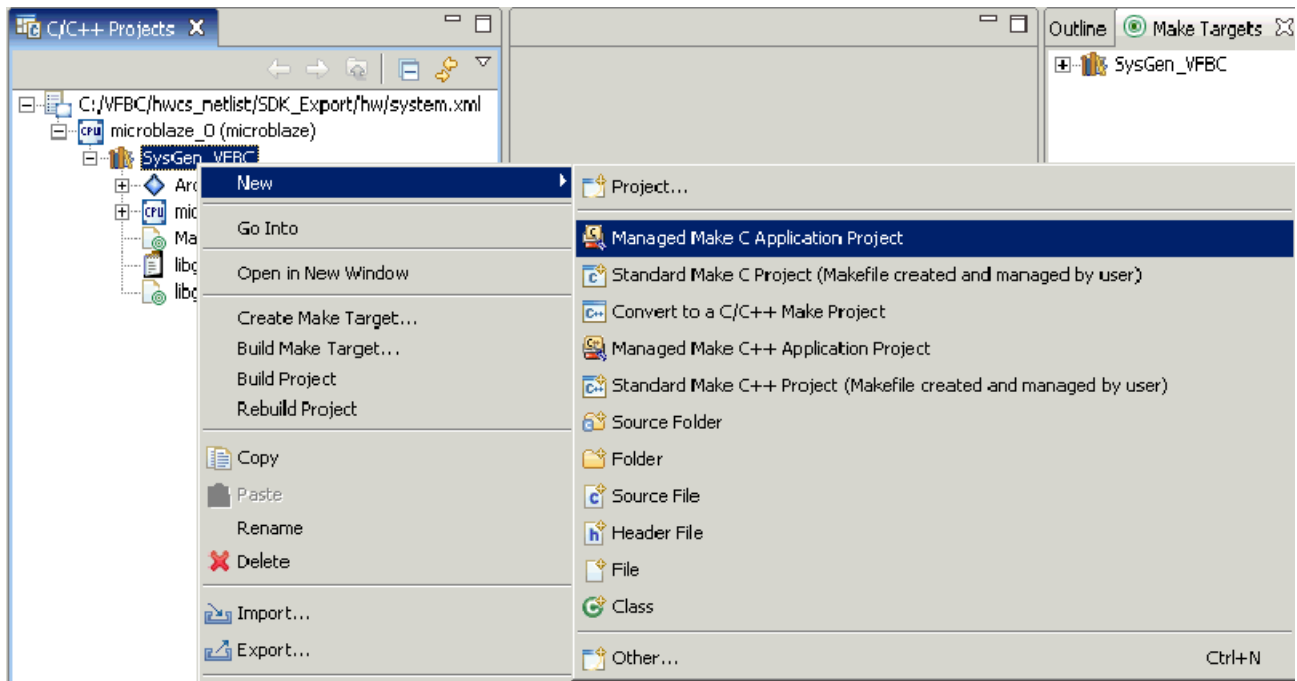
4. [Software Platform] を選択し、[Next] をクリックします。



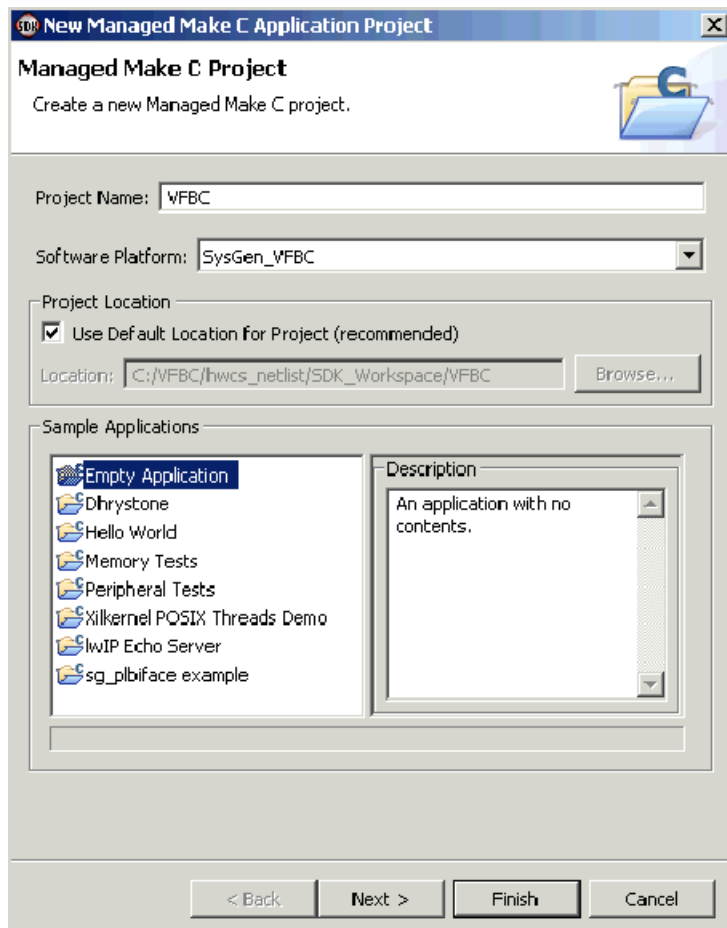
5. プロジェクト名を入力し、[Finish] をクリックします。



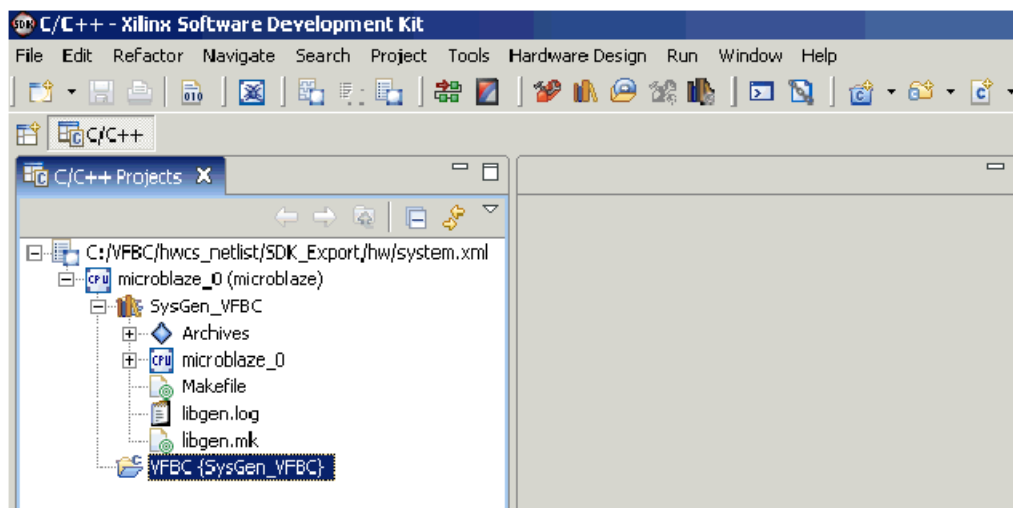
6. [SysGen_VFBC] を右クリックし、[New] → [Managed Make C Application Project] をクリックします。



7. ソフトウェア プラットフォーム プロジェクト名を入力し、[Empty Application] を選択して、[Finish] をクリックします。



SDK デザインの表示は、次のようになります



8. 新規 C コード ソース ファイルを作成するか、既存のファイルをプロジェクトに追加します。この例では、C:\VFBC\C-code\vfbc.c を追加します。

VFBC {SysGen_VFBC} アプリケーション プロジェクトに C コード ソース ファイルを追加するには、プロジェクトにファイルをコピーして張り付けるか、ドラッグ アンド ドロップするのが最も簡単な方法です。ファイルを追加すると、プロジェクトが構築され、自動的にコンパイルされます。

System Generator と SDK の間でのデザインのやり取り

デフォルトでは、ELF ファイルにはアプリケーション プロジェクト名に基づく名前が付けられます (この例では vfbc.elf)、プロジェクト ディレクトリ内のフォルダに配置されます。

SDK でのデザインの作業は XPS での作業と同様ですが、SDK にはより高度な機能が含まれています。C コードを変更し、保存すると、ソフトウェア アプリケーションが自動的に再構築および再コンパイルされます。これにより新しい ELF ファイルが生成され、これを **System Generator** で最コンパイルし、ターゲット プラットフォーム用のビットストリームをアップデートします。

ソフトウェア

1. SDK : C コードを変更し、ソフトウェア プロジェクトが正しく再コンパイルされたことを確認します。
2. System Generator : [Compile and Update bitstream] ボタンをクリックします。
3. System Generator : デザインをシミュレーションします。

ハードウェア

1. SDK : 新規ペリフェラルを追加するか、既存のペリフェラルを変更します。
2. System Generator : XPS プロジェクトを System Generator デザインに再インポートします。
3. System Generator : ハードウェア協調シミュレーション ブロックを再生成します。
4. SDK : C コードを必要に応じて変更します。
5. System Generator : デザインを再度シミュレーションします。

メモ : ハードウェアに変更を加えた場合、デザインを配置配線までインプリメンテーションする必要があります。

ハードウェア協調シミュレーションの使用

概要

System Generator ではハードウェア協調シミュレーションがサポートされており、FPGA で実行するデザインを直接 Simulink シミュレーションに組み込むことができます。コンパイルのターゲットとしてハードウェア協調シミュレーションを選択すると、ビットストリームが作成され、ブロックに関連付けられます。デザインを Simulink でシミュレーションすると、コンパイルされた部分の結果はハードウェアで算出されます。コンパイルされた部分は実際のハードウェアでテストされるので、シミュレーションを大幅に高速化できます。

M コードでのハードウェア協調シミュレーションへのアクセス

System Generator ハードウェア協調シミュレーションフローで作成したハードウェアを、MATLAB M コード (M-Hwcosim) を使用してプログラムで制御できます。M-Hwcosim インターフェイスにより、Simulink フレームワークから独立したハードウェアに対応する MATLAB オブジェクトを M コードのみで作成することができます。これらのオブジェクトを使用して、ハードウェアに対して読み出しおよび書き込みを実行できます。この機能はハードウェア協調シミュレーションのスクリプト インターフェイスとして機能し、ハードウェアをスクリプトで記述されたテストベンチで使用できるようにしたり、M コードにハードウェアアクセラレーションとして導入できます。

詳細は、『System Generator for DSP リファレンスガイド』の「プログラムを使用したアクセス」の章で「ハードウェア協調シミュレーションへの M コードアクセス」を参照してください。

ハードウェア プラットフォームのインストール

ハードウェア協調シミュレーションを実行するには、ハードウェアプラットフォームをインストールし、セットアップする必要があります。次のトピックは、ザイリンクスでサポートされるプラットフォームのインストールおよびセットアップ手順を示しています。

イーサネット ベース ハードウェア協調シミュレーション

[イーサネット ハードウェア協調シミュレーション用の ML402 プラットフォームのインストール](#)

[イーサネット ハードウェア協調シミュレーション用の ML506 プラットフォームのインストール](#)

[イーサネット ハードウェア協調シミュレーション用の ML605 プラットフォームのインストール](#)

[イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 1800A スタータ プラットフォームのインストール](#)

[イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 3400A 開発プラットフォームのインストール](#)

メモ：上記のプラットフォーム以外のインストール手順は、ご使用のプラットフォーム キットに付属のインストール ガイドを参照してください。

JTAG ベース ハードウェア協調シミュレーション

JTAG ハードウェア協調シミュレーション用の ML402 プラットフォームの インストール

JTAG ハードウェア協調シミュレーション用の ML605 プラットフォームの インストール

JTAG ハードウェア協調シミュレーション用の SP605 プラットフォームの インストール

サードパーティ ハードウェア協調シミュレーション

ザイリンクスでは、XtremeDSP™ ソリューションの一環として、多数の代理店および OEM と協力して、さまざまな DSP プロトタイプの作成およびプラットフォームの開発を行っています。使用可能なプラットフォームの詳細は、ザイリンクス Web サイトの次のページを参照してください。

http://japan.xilinx.com/products/boards_kits/index.htm

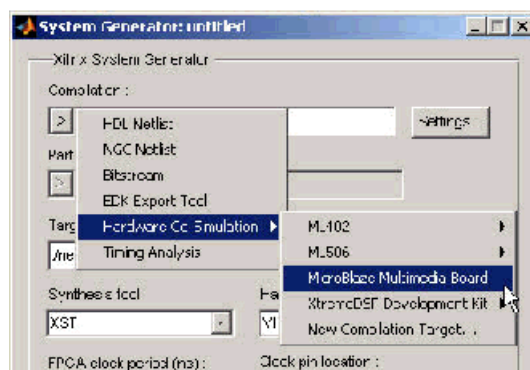
ハードウェア協調シミュレーション用のモデルのコンパイル

ハードウェア プラットフォームをインストールしたら、ハードウェア協調シミュレーションはハードウェアで実行する System Generator モデルまたはサブシステムから開始します。モデルは、ハードウェア プラットフォームの要件を満たしていれば、協調シミュレーションできます。このモデルには、System Generator トークンを含める必要があります。System Generator トークンでは、モデルをハードウェアにコンパイルする方法を定義します。まず System Generator トークンのパラメータ ダイアログ ボックスを開き、[Compilation] でコンパイル ターゲットを選択します。

System Generator トークンの使用方法は、「[System Generator トークンを使用したコンパイルとシミュレーション](#)」を参照してください。

コンパイル ターゲットの選択

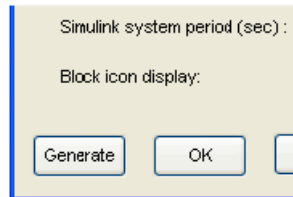
System Generator トークンのパラメータ ダイアログ ボックスで、ハードウェア協調シミュレーション プラットフォームを選択できます。ハードウェア協調シミュレーション ターゲットは、[Compilation] の [Hardware Co-Simulation] メニューから選択できます。



コンパイルターゲットを選択すると、System Generator トークンのパラメータ ダイアログ ボックスで、選択したターゲットに適したオプションが自動的に設定されます。System Generator では、各コンパイルターゲットの設定が保存されます。これらの設定は新しいターゲットを選択したときに保存され、ターゲットを再び設定すると呼び出されます。

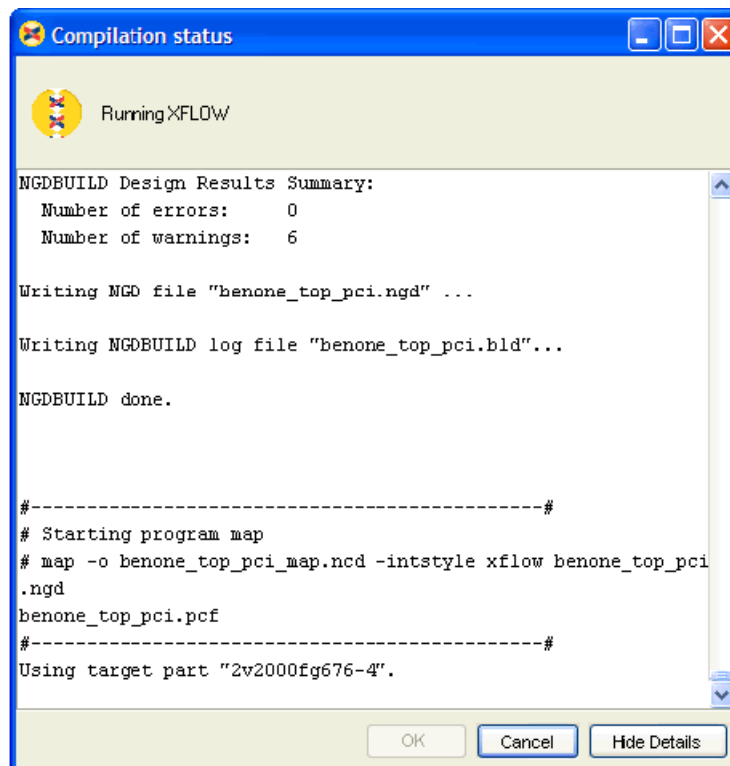
コードの生成

System Generator トークンのパラメータ ダイアログ ボックスで [Generate] ボタンをクリックすると、コードが生成されます。



ハードウェア協調シミュレーションに適したデザインの FPGA コンフィギュレーション ビットストリームが生成されます。コンパイル プロセスでは、HDL ファイルおよびネットリスト ファイルが生成されるだけでなく、FPGA コンフィギュレーション ファイルを生成するのに必要なダウンストリーム ツールも実行されます。

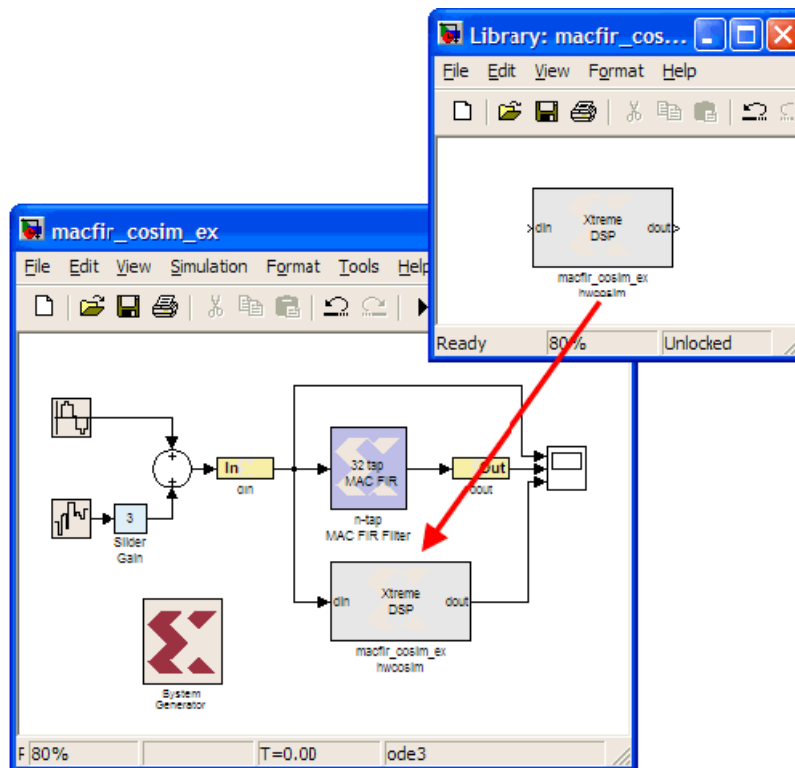
メモ： [Generate] をクリックすると、次のようなステータス ダイアログ ボックスが表示されます。コンパイル中にこのダイアログ ボックスの [Cancel] をクリックするとコンパイルが停止し、[Show Details] をクリックするとコンパイルの各フェーズの詳細が表示されます。詳細を表示している場合、[Hide Details] をクリックすると詳細が非表示になります。



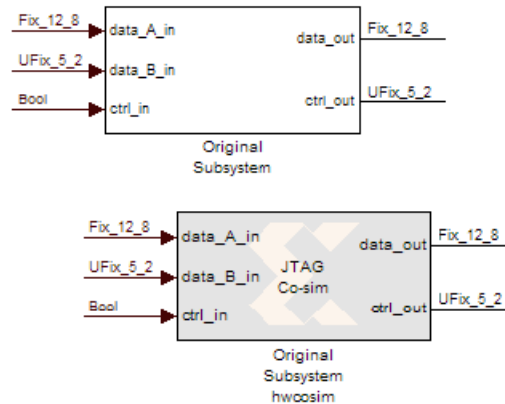
コンフィギュレーション ビットストリームには、モデルに関連するハードウェアに加え、プラットフォームと PC 間の物理インターフェイスを使用した **System Generator** とデザインを通信させるためインターフェイス ロジックが含まれています。このロジックには、**System Generator** でデザインの入力ポートと出力ポートに対して値の読み出しおよび書き込みを実行するメモリ マップ インターフェイス、ターゲット **FPGA** プラットフォームを正しく機能させるために必要なプラットフォーム専用の回路 (DCM、外部コンポーネント配線など) が含まれます。

ハードウェア協調シミュレーション ブロック

System Generator では、デザインの **FPGA** ビットストリームへのコンパイルが終了すると、ハードウェア協調シミュレーション ブロックと、ハードウェア協調シミュレーション ブロックを保存する **Simulink** ライブラリが作成されます。作成されたハードウェア協調シミュレーション ブロックは、ほかの **Simulink** および **System Generator** のブロックと同様に、ライブラリから **System Generator** デザインにコピーして使用できます。



ハードウェア協調シミュレーション ブロックは、そのブロックの基になるモデルまたはサブシステムの外部インターフェイスとして機能します。ハードウェア協調シミュレーション ブロックのポート名、ポート タイプ、レートは、元のサブシステムのものと一致します。



ハードウェア協調シミュレーションブロックは、**Simulink** デザインでその他のブロックと同様に使用されます。シミュレーションでは、**FPGA** プラットフォームと通信し、デバイス コンフィギュレーション、データ転送、クロック供給などを自動化します。ハードウェア協調シミュレーションブロックでは、ほかの **System Generator** ブロックと同じ型の信号を使用します。ブロックの入力ポートに値が書き込まれると、対応するデータがハードウェアの適切な場所に送信され、出力ポートでイベントが発生すると、ハードウェアからデータが取得されます。

ハードウェア協調シミュレーションブロックは、ザイリンクスの固定小数点信号、**Simulink** の固定小数点信号、または **Simulink** の倍精度信号で駆動できます。出力ポートでは、駆動するブロックに適切な信号型が使用されます。出力ポートを **System Generator** ブロックに接続すると、ザイリンクスの固定小数点信号が生成され、ポートが **Simulink** ブロックを駆動する場合は **Simulink** データ型が使用されます。

メモ : ブロック信号型として **Simulink** データ型が使用される場合、入力データの量子化は **[Round]**、オーバーフローは **[Saturate]** に設定されます。

ハードウェア協調シミュレーションブロックでは、その他の **System Generator** ブロックと同様に、パラメータ ダイアログ ボックスでパラメータを設定できます。ハードウェア協調シミュレーションブロックのパラメータは、ブロックがインプリメントされる **FPGA** プラットフォームによって異なります (各 **FPGA** プラットフォームで独自のカスタマイズされたハードウェア協調ブロックが提供される)。

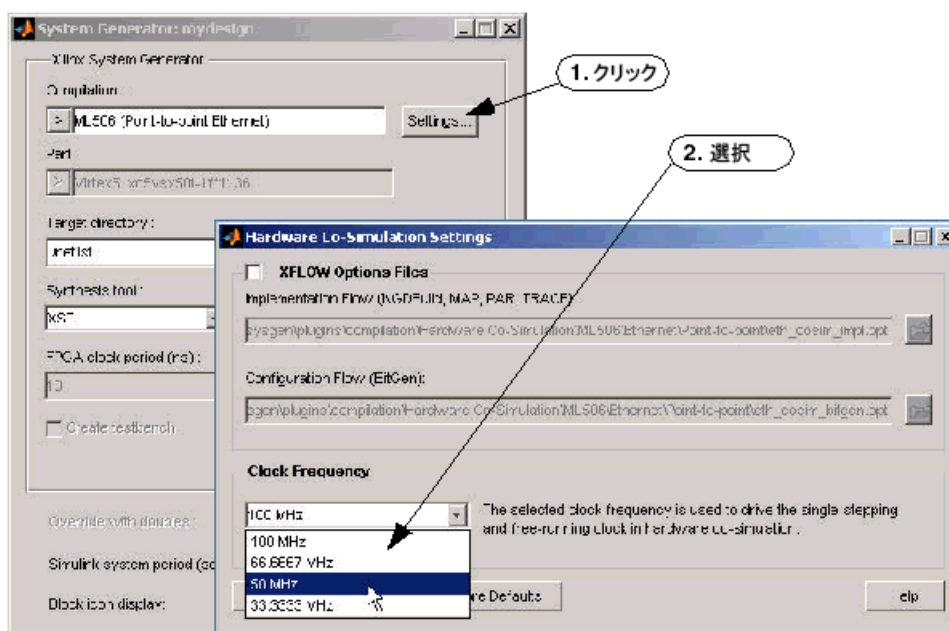
ハードウェア協調シミュレーションのクロック

クロック周波数の選択

ザイリンクス ML402 または ML506 プラットフォームを使用する場合、ターゲット デザインのクロック周波数 (システム クロック周波数以下) を選択できます。次の表に、選択可能な周波数を示します。

プラットフォーム	インターフェイス	システム クロック周波数	選択可能な周波数
ザイリンクス ML402	JTAG ポイントツーポイント イーサネット ネットワーク ベースのイーサネット	100MHz	100MHz
			66.7MHz
			50MHz
			33.3MHz
ザイリンクス ML506	ポイントツーポイント イーサネット ネットワーク ベースのイーサネット	200MHz	100MHz
			66.7MHz
			50MHz
			33.3MHz

クロック周波数を設定するには、次の図に示すように、System Generator トークンのパラメータ ダイアログ ボックスで [Settings] ボタンをクリックし、プルダウン メニューから周波数を選択します。



クロック モード

System Generator ハードウェア協調シミュレーション ブロックを **FPGA** ハードウェアと同期化するには、いくつかの方法があります。シングル ステップ クロック モードでは、**FPGA** には **Simulink** からクロックが供給されます。フリー ランニング クロック モードでは、**FPGA** は内部クロックで動作し、**Simulink** がハードウェア協調シミュレーション ブロックを起動すると、非同期にサンプリングされます。

シングル ステップ クロック

シングル ステップ クロック モードでは、ハードウェアはソフトウェア シミュレーションと同じステップで実行されます。これは、各シミュレーション サイクルごとにハードウェアに 1 クロック パルス (入力/出力レートに対して **FPGA** の周波数を高くしている場合は複数のクロック パルス) を供給することで達成します。このモードでは、ハードウェア協調シミュレーションは元のモデルに対してビット単位およびサイクル単位です。

ハードウェア協調シミュレーション ブロックは、**Simulink** により起動された場合にのみ **FPGA** ハードウェアに対してクロック信号を生成するので、**Simulink** モデルの残りの部分のシミュレーションに関するオーバーヘッドおよび **Simulink** と **FPGA** プラットフォーム間の通信オーバーヘッド (バス レイテンシなど) によりハードウェアで達成可能なパフォーマンスが大幅に制限されます。一般的には、**FPGA** 内での処理量が通信オーバーヘッドよりもかなり大きい場合は (ロジックが大きい、ハードウェアの周波数を大幅に高くしている場合など)、ハードウェアでのシミュレーションは高速になります。

フリーランニング クロック

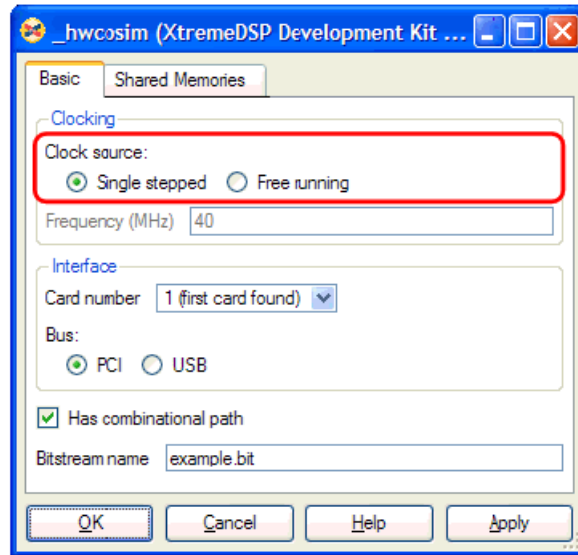
フリーランニング クロック モードでは、ハードウェアはソフトウェア シミュレーションとは非同期に実行されます。**Simulink** で **FPGA** クロックが生成されるシングル ステップ クロック モードとは異なり、ハードウェア クロックは **FPGA** 内で連続的に動作します。

このモードでは、シミュレーションは元のモデルに対してビット単位およびサイクル単位ではなく、**Simulink** がハードウェア協調シミュレーションを起動したときにのみハードウェアの内部ステータスがサンプリングされます。**FPGA** ポート I/O は、**Simulink** のイベントと同期しません。**Simulink** のポートでイベントが発生すると、その時点でハードウェアの対応するポートに対して読み出しまたは書き込みが実行されます。ポート イベント間でのクロック サイクル数は不明なので、ハードウェアの現在のステータスを元の **System Generator** モデルに調整することはできません。このモードでは、**FPGA** をフル スピードで動作させ、必要に応じて **Simulink** と同期化するので、多くのストリーミング アプリケーションに適しています。

フリーランニング モードでは、**System Generator** モデルに同期化機構を構築する必要があります。簡単な例としては、ステータス レジスタがあります。このステータス レジスタを出力ポートとしてハードウェア協調シミュレーション ブロックに追加し、条件が満たされたときにハードウェアでセットされるようにします。**System Generator** モデルの残りの部分では、このステータス レジスタをポーリングしてハードウェアのステータスを判断できます。

クロック モードの選択

すべてのハードウェア プラットフォームでフリーランニング クロックをサポートしているわけではありませんが、サポートしているプラットフォームでは、ハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスでクロック モードを選択できます。[Basic] タブの [Clocking] で、[Single stepped] または [Free running] をオンにします。



メモ：ハードウェア協調シミュレーションブロックのクロック オプションは、使用する FPGA プラットフォームにより異なります。フリーランニング クロックがサポートされないプラットフォームでは、ダイアログ ボックスで選択できません。

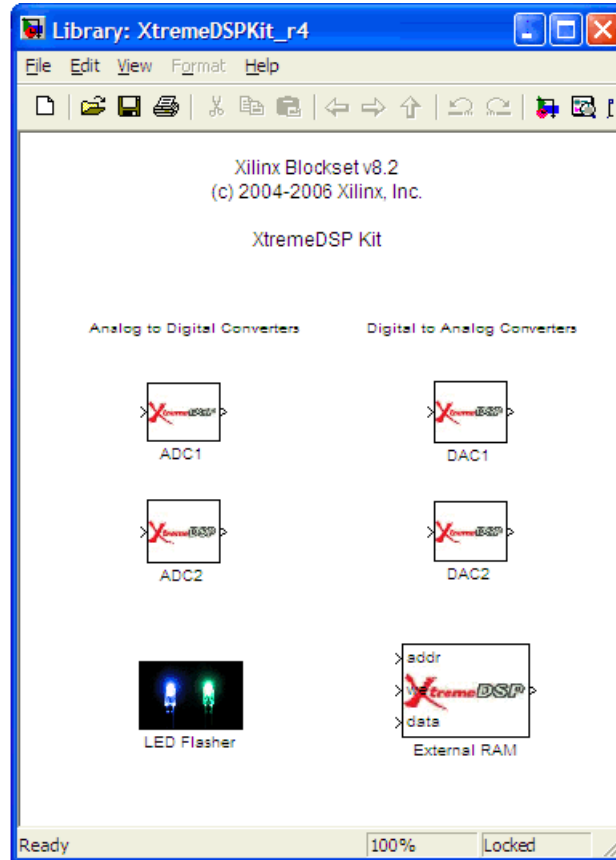
ボード専用 I/O ポート

FPGA プラットフォームには、FPGA と通信可能なさまざまなオンボード デバイス (外部メモリ、AD コンバータなど) が含まれます。これらのコンポーネントを **System Generator** モデル内で接続し、ハードウェア協調シミュレーションで使用すると有益な場合があります。たとえば、ボードに外部メモリが含まれる場合、**System Generator** デザインでこのメモリに制御ロジックおよびインターフェイスロジックを定義し、ハードウェア協調シミュレーションで物理メモリを使用できます。

これらのコンポーネントを接続するには、**System Generator** モデルにボード専用 I/O ポートを含めます。ボード専用ポートは、モデルをハードウェア協調シミュレーション用にコンパイルしたときに FPGA パッドに配線されます。これらのポートは、ハードウェア協調シミュレーションブロックのポートで制御される標準の協調シミュレーションポートとは異なります。

ボード専用 I/O ポートは、メモリにマップされない特殊な **Gateway** ブロックを使用してインプリメントします。このブロックにより、モデルをハードウェアにコンパイルしたときに信号が適切な FPGA ピンに配線されます。**System Generator** 信号をボード専用ポートに接続するには、ワイヤをこの特殊な **Gateway** ブロックに接続します。

多くの場合、メモリにマップされない **Gateway** ブロックは、**Simulink** サブシステムまたはライブラリに含まれています。たとえば、**XtremeDSP** 開発キットでは、AD コンバータ、DA コンバータ、LED、外部メモリを含む外部デバイス インターフェイス サブシステムのライブラリが提供されています。インターフェイス サブシステムは、ボード専用ポート接続を指定する **Gateway** ブロックを使用して構築されています。シミュレーション中、これらのサブシステムはその他の **System Generator** サブシステムと同様に処理されます (倍精度からザイリンクス固定小数点型への変換を実行)。**System Generator** でデザインがハードウェアにコンパイルされると、**Gateway** ブロックに関連する信号がハードウェアで指定した外部デバイスに接続されます。



ハードウェア協調シミュレーションでの I/O ポート

ハードウェア協調シミュレーション ブロックには、外部インターフェイスにボード専用ポートは含まれません。これは、モデルにボード専用ポートに対応する **Gateway** ブロックがある場合、デザインをハードウェア協調シミュレーション用にコンパイルしたときに、対応するポートが実際のハードウェアではなくシミュレーション モデルに接続されます。ポートを実際のポートに接続するには、メモリにマップされない **Gateway** ブロックを使用する必要があります。メモリにマップされないポートについては、「[新規プラットフォームのサポート](#)」を参照してください。

イーサネット ハードウェア協調シミュレーション

System Generator では、イーサネット接続を介した FPGA プラットフォームとの高スループット通信を可能にするハードウェア協調シミュレーション インターフェイスが提供されています。これらのインターフェイスでは、プログラム ケーブルを使用した場合の通信範囲の制限をなくし、リアルタイム アプリケーションでのバンド幅も広がります。デバイス コンフィギュレーションをイーサネットをサポートされるので、プログラム ケーブルは必要ありません。

2 種類のイーサネット ハードウェア協調シミュレーションがサポートされています。ポイントツーポイント イーサネット協調シミュレーションでは、PC と FPGA プラットフォーム間で直接的なポイントツーポイント イーサネット接続を使用して、高パフォーマンス協調シミュレーション環境が提供されます。ネットワーク ベースのイーサネット協調シミュレーションでは、IPv4 ネットワーク インフラストラクチャを介してリモート FPGA との通信が可能です。

ポイントツーポイント イーサネット ハードウェア協調シミュレーション

ポイントツーポイント イーサネット ハードウェア協調シミュレーションでは、イーサネット 接続をそのまま使用した協調シミュレーション インターフェイスが提供されます。このイーサネット 接続はレイヤ 2 (データリンク層) イーサネット 接続であり、FPGA 開発プラットフォームとホスト PC をネットワーク ルート 装置なしで接続します。広く 普及した高度なイーサネット 技術を活用することにより、外部 FPGA デバイスに対してバンド 幅の広い協調シミュレーションを実行可能です。

インターフェイスの機能

インターフェイスでは、10/100/1000Mbps の半二重/全二重モードがサポートされています。ギガビット イーサネット では、元の接続でイネーブルになっていれば、ジャンボ フレームもサポートされます。FPGA デバイス コンフィギュレーションでは、ザイリンクス パラレル ケーブル IV またはザイリンクス プラットフォーム ケーブル USB を介した JTAG ベースのコンフィギュレーション、あるいは協調シミュレーションと同じポイントツーポイント イーサネット 接続を介したイーサネット ベースのコンフィギュレーションがサポートされます。

メモ：この協調シミュレーション インターフェイスでは、評価版のイーサネット MAC コアを使用しています。評価版であるため、ターゲット FPGA で連続的に長時間 (約 7 時間) 使用すると、動作しなくなります。シミュレーションを再開すると、コアは再び動作するようになります。コアの正規版の入手方法は、次の Web サイトを参照してください。

<http://japan.xilinx.com/products/ipcenter/TEMAC.htm>

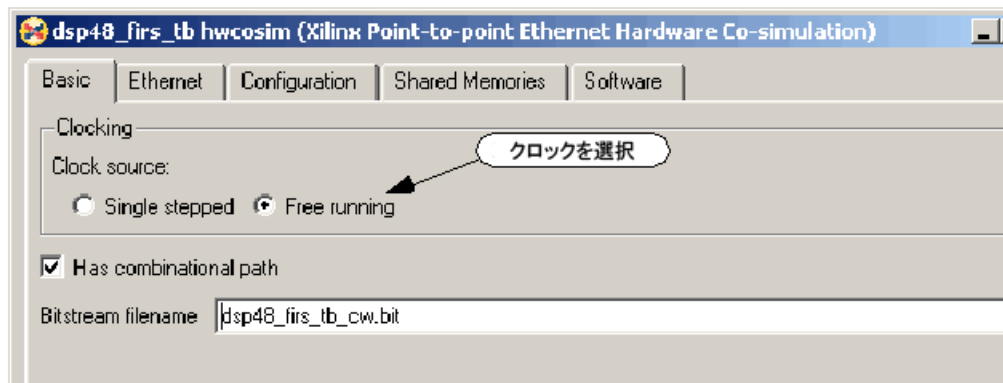
サポートされる FPGA 開発プラットフォーム

「イーサネット ベース ハードウェア協調シミュレーション」に、ポイントツーポイント イーサネット ハードウェア協調シミュレーションをサポート する開発プラットフォームおよびそのインストール手順へのリンクがリストされています。

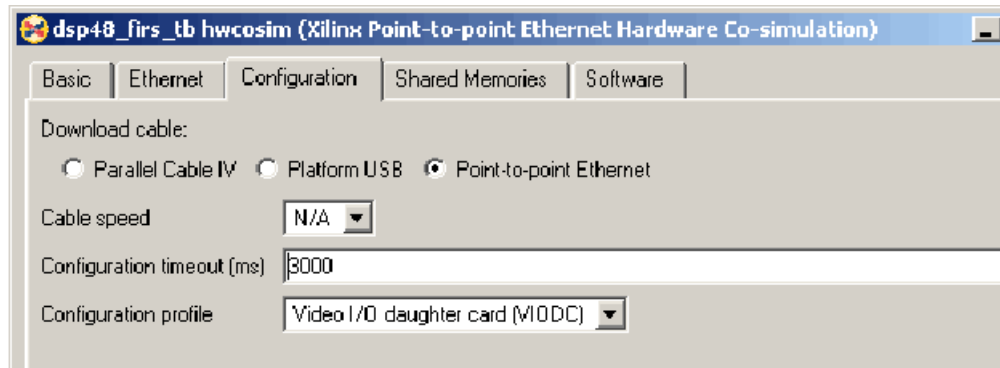
協調シミュレーション ブロックのパラメータ設定

ポイントツーポイント イーサネット協調シミュレーション インターフェイスに特有のパラメータがいくつかあります。このセクションでは、これらのパラメータの設定方法を説明します。すべてのブロック パラメータの詳細は、「[Point-to-point Ethernet Co-Simulation](#)」を参照してください。

1. [Basic] タブで、協調シミュレーションのクロック ソースを選択します。

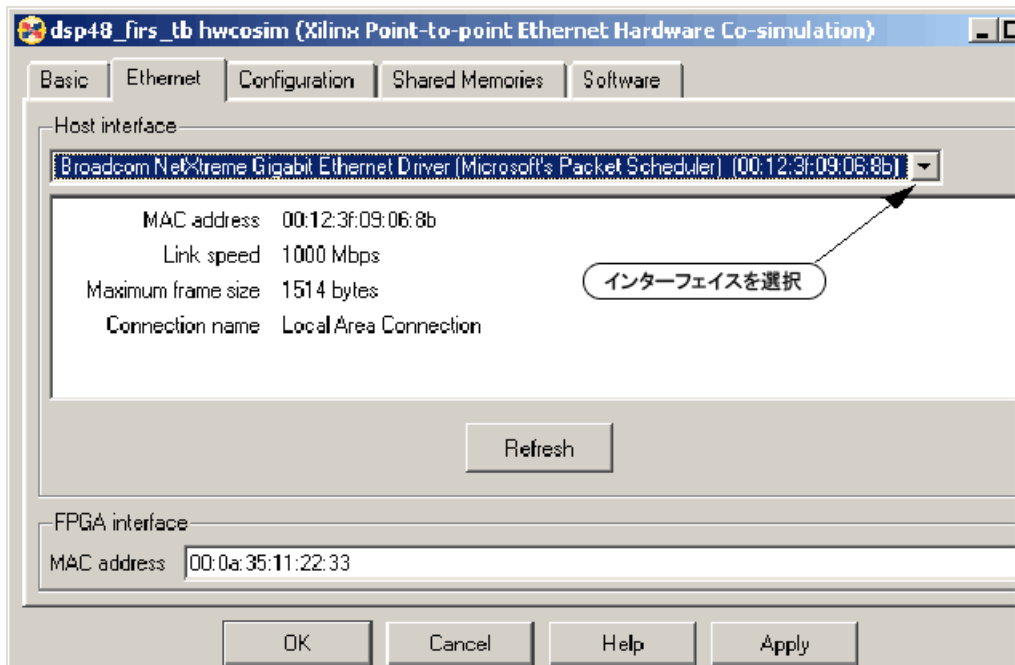


2. [Configuration] タブで、コンフィギュレーション方法を選択します。



- ◆ [Download cable] で、[Point-to-point Ethernet] をオンにします。
- ◆ JTAG ベースのダウンロード ケーブル ([Parallel Cable IV] または [Platform USB]) を選択した場合は、ケーブル スピードを適切な値に変更します。
- ◆ [Configuration timeout (ms)] は、必要な場合にのみ変更します。ほとんどの場合は、デフォルト値で十分です。デバイスのコンフィギュレーションが完了した後に FPGA プラットフォームとのネットワーク接続を再確立するのに時間がかかる場合は、この値を大きくする必要があります。
- ◆ ML402 プラットフォームにビデオ I/O ドータ カードが接続されている場合は、[Configuration profile] で [Video I/O daughter card (VIODC)] を選択します。

3. [Ethernet] タブで、イーサネット インターフェイス設定を選択します。



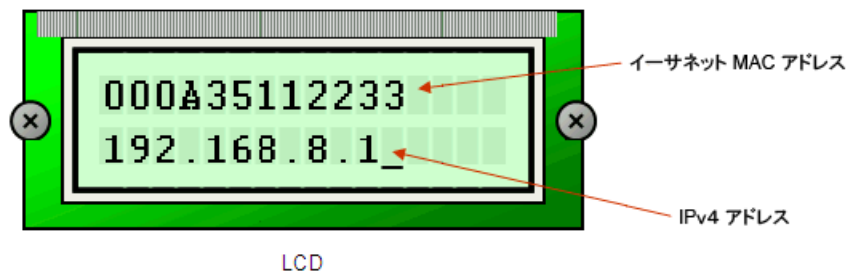
- ◆ [Host interface] で、協調シミュレーション用のネットワーク インターフェイスを選択します。

メモ：ドロップダウン リストには、ホストにインストールされているイーサネット互換のネットワーク インターフェイスで、10/100/1000Mbps をサポートし、現在イネーブルでアクティブ イーサネット セグメントに接続されているもののみが表示されます。予測されるターゲット インターフェイスが表示されない場合は、接続を確認して [Refresh] をクリックし、リストをアップデートしてください。

- ◆ ドロップダウン リストの下ボックスに、選択したインターフェイスに関する情報が表示されます。情報を参照して適切なインターフェイスが選択されていることを確認し、必要に応じて OS のネットワーク設定を変更します。
4. 選択したコンフィギュレーション方法によって、[FPGA interface] の [MAC address] の変更が必要な場合があります。

a. ポイントツーポイント イーサネット ベースのコンフィギュレーション

コンフィギュレーション ブートローダを実行しているときに、ターゲット プラットフォームの LCD に表示される MAC アドレスを確認します。協調シミュレーション ブロックのデフォルト値が一致していない場合は、[MAC address] を変更します。ML402 プラットフォームの MAC アドレスの設定については、「イーサネット MAC アドレスと IPv4 アドレスの設定 (オプション)」を参照してください。

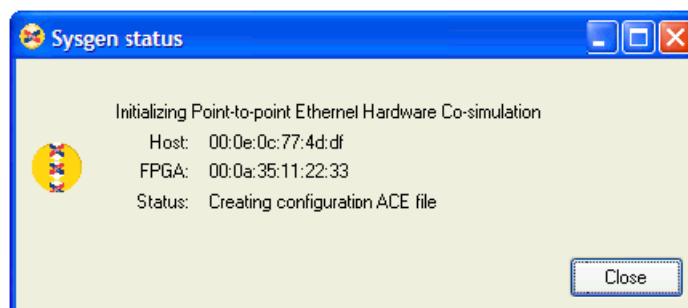


メモ：MAC アドレスは、2 桁の 16 進数 6 個をコロンで区切って指定します (00:0a:35:11:22:33 など)。

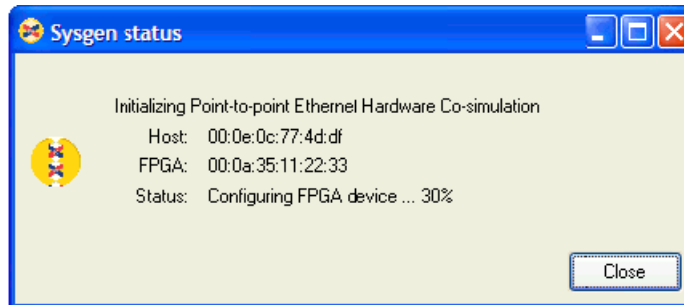
デザインの協調シミュレーション

ブロック パラメータを設定したら、Simulink モデルのツールバーで [シミュレーションの開始] をクリックして協調シミュレーションを開始します。System Generator で自動的にデバイスのコンフィギュレーションが実行され、協調シミュレーション用にテスト中のデザイン (DUT) が FPGA デバイスに転送されます。プロセスのステータスを示すダイアログ ボックスが表示されます。

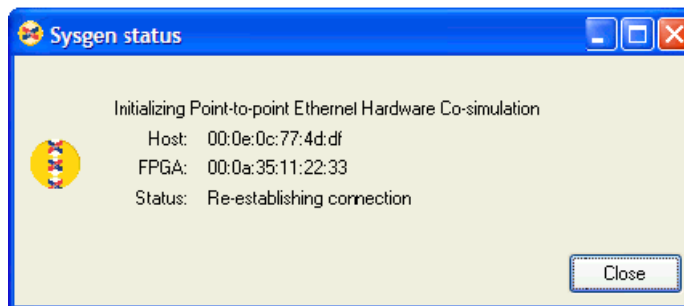
1. ブロック パラメータで指定した入力ビットストリームに基づいて、最終的なコンフィギュレーション ファイルが生成されます。



- 最終的なコンフィギュレーション ファイルが選択したダウンロード ケーブルを使用してターゲット プラットフォームに転送され、FPGA デバイスのコンフィギュレーションに使用されます。ポイントツーポイント イーサネット接続を介したコンフィギュレーションの進行状況がダイアログ ボックスに表示されます。



- デバイスのコンフィギュレーションが完了すると、協調シミュレーション エンジンでターゲット プラットフォームへの接続が再確立され、デザインの協調がシミュレーションが開始します。



既知の問題

- ポイントツーポイント イーサネット接続でのデータ転送で問題が発生した場合や、安定性の問題が発生した場合は、Intel プラットフォームのハイパースレッディングをディスエーブルしてください。
- IP フラグメンテーションは、ネットワーク ベースのイーサネット コンフィギュレーションでサポートされていません。ネットワーク管理者に連絡するかイーサネット インターフェイスカードのユーザー マニュアルを参照し、ホストとターゲット FPGA プラットフォーム間の接続で、1300 バイト以上の MTU (Maximum Transmission Unit) をフラグメンテーションなしで処理できることを確認してください。MTU サイズ (または最大転送サイズ、ジャンボ フレーム サイズなど同様の最大フレーム サイズ) は、イーサネット インターフェイス設定から変更できません。

ネットワーク ベースのイーサネット ハードウェア協調シミュレーション

インターフェイスの機能

インターフェイスでは、10/100/1000Mbps の半二重/全二重モードがサポートされています。FPGA デバイスのコンフィギュレーションでは、協調シミュレーションと同じネットワーク接続を介したイーサネット ベースのコンフィギュレーションがサポートされており、プログラム ケーブル (パラレル ケーブル IV など) は必要ありません。

メモ：この協調シミュレーション インターフェイスでは、評価版のイーサネット MAC コアを使用しています。評価版であるため、ターゲット FPGA で連続的に長時間 (約 7 時間) 使用すると、動作しなくなります。シミュレーションを再開すると、コアは再び動作するようになります。コアの正規版の入手方法は、次の Web サイトを参照してください。

<http://japan.xilinx.com/products/ipcenter/TEMAC.htm>

サポートされる FPGA 開発プラットフォーム

ネットワーク ベースのイーサネット 協調シミュレーションでは、ザイリンクス ML402 および ML506 開発プラットフォームがサポートされています。

セットアップ手順

1. ネットワーク ベースのイーサネット協調シミュレーションでは、ネットワーク コンフィギュレーションを介してデバイスがコンフィギュレーションされます。ネットワーク コンフィギュレーションを使用する前に、IP アドレス、MAC アドレス、コンフィギュレーション サーバーが System ACE™ CompactFlash で適切に設定する必要があります。設定方法の詳細は、「[イーサネット MAC アドレスと IPv4 アドレスの設定 \(オプション\)](#)」を参照してください。
2. ターゲット FPGA は、UDP ポート 9999 から信号を受信します。ネットワークで関連する通信が遮断されていないことを確認してください。

既知の問題

- IP フラグメンテーションは、ネットワーク ベースのイーサネット コンフィギュレーションでサポートされていません。ネットワーク管理者に連絡するかイーサネット インターフェイスカードのユーザー マニュアルを参照し、ホストとターゲット FPGA プラットフォーム間の接続で、1300 バイト以上の MTU (Maximum Transmission Unit) をフラグメンテーションなしで処理できることを確認してください。MTU サイズ (または最大転送サイズ、ジャンボ フレーム サイズなど同様の最大フレーム サイズ) は、イーサネット インターフェイス設定から変更できます。

共有メモリのサポート

System Generator のハードウェア協調シミュレーション インターフェイスでは、共有メモリ ブロックおよび共有メモリ ブロックから作成されたブロック（共有 FIFO、共有レジスタなど）を FPGA ハードウェアにコンパイルし、協調シミュレーションできます。これらのインターフェイスにより、ハードウェア ベースの共有メモリ リソースを、ホスト PC の共通アドレス空間にマップすることが可能です。共有メモリを System Generator 協調シミュレーション ハードウェアに適用すると、PC と FPGA 間でデータを高速に転送でき、リアルタイム ハードウェア協調シミュレーション機能が強化されます。このセクションでは、System Generator のハードウェア協調シミュレーションで共有メモリを使用する方法を説明します。

- | | |
|-------------------------------|---|
| ハードウェア協調シミュレーション用の共有メモリのコンパイル | System Generator デザインに共有メモリ ブロックが含まれている際に、デザインをハードウェア協調シミュレーション用にコンパイルする方法を説明します。 |
| 非保護の共有メモリの協調シミュレーション | 非保護のアクセス モードに設定された共有メモリ ブロックが、ハードウェア協調シミュレーション中にどのように動作するかを説明します。 |
| ロック共有メモリの協調シミュレーション | ロック アクセス モードに設定された共有メモリ ブロックが、ハードウェア協調シミュレーション中にどのように動作するかを説明します。 |
| 共有レジスタの協調シミュレーション | System Generator デザインに共有レジスタが含まれている際に、デザインをハードウェア協調シミュレーション用にコンパイルする方法を説明します。 |
| 共有 FIFO の協調シミュレーション | System Generator デザインに共有 FIFO が含まれている際に、デザインをハードウェア協調シミュレーション用にコンパイルする方法を説明します。 |
| 共有メモリの制限 | ハードウェア協調シミュレーションで共有メモリ ブロックを使用する際の制限を示します。 |

ハードウェア協調シミュレーション用の共有メモリのコンパイル

System Generator では、共有メモリ ブロックおよび共有メモリ ブロックから作成されたブロック (共有 FIFO、共有レジスタなど) をハードウェア協調シミュレーション用にコンパイルできます。共有メモリを含むデザインをハードウェア協調シミュレーション用にコンパイルするには、通常の System Generator デザインと同様に、System Generator トークンのパラメータ ダイアログ ボックスでコンパイル ターゲットを設定し、[Generate] ボタンをクリックします。共有メモリ ブロックを含むデザインは、「共有メモリの制限」に示す要件を満たしていれば、ハードウェア協調シミュレーション用にコンパイルできます。

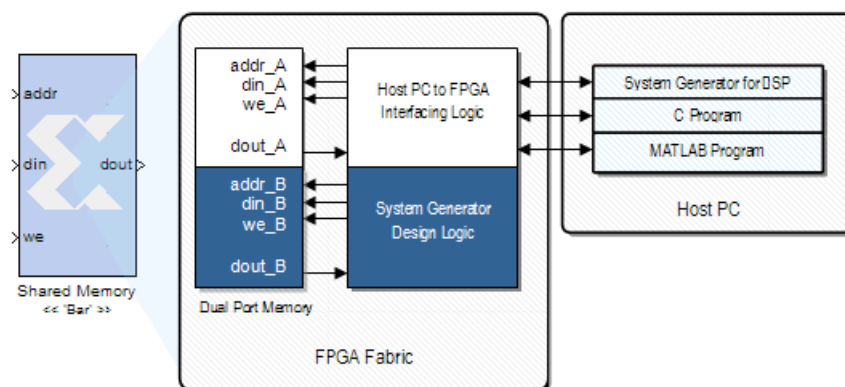
共有メモリをハードウェア協調シミュレーション用にコンパイルすると、コアまたは HDL コンポーネントによりハードウェアにインプリメントされます。次の表に、共有メモリ ブロックがハードウェア インプリメンテーションにどのようにマップされるかを示します。

To ブロック	From ブロック	ハードウェア インプリメンテーション
Shared Memory	Shared Memory	Dual Port Block Memory 6.1
To FIFO	To FIFO	FIFO Generator 2.1
To Register	To Register	synth_reg_w_init.(vhd,v)

共有メモリをハードウェア協調シミュレーション用にコンパイルするには、2 つの方法があります。コンパイル方法は、共有メモリの名前がデザイン内で固有であるか、同じ名前のパートナーがあるかによって異なります。次に、これらのコンパイル方法について説明します。

単一の共有メモリのコンパイル

共有メモリ ブロックの名前がデザイン内で固有であれば、単一であると判断されます。単一の共有メモリをハードウェア協調シミュレーション用にコンパイルすると、System Generator でペアになるメモリが作成され、ネットリストに組み込まれます。メモリに PC と通信するためのインターフェイス ロジックが追加されます。共有メモリで協調シミュレーションを実行すると、一方のメモリは System Generator デザインのロジックで使用され、もう一方のメモリは PC インターフェイス ロジックと通信するために使用されます (下図を参照)。このようになっていることで、シミュレーションを実行中に FPGA 内に組み込まれた共有メモリとの通信が可能になります。



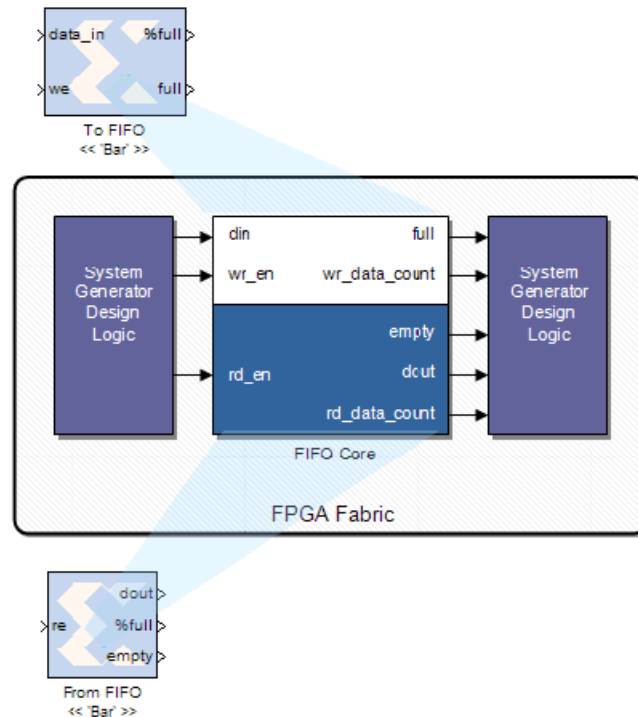
共有メモリ ハードウェアとインターフェイス ロジックは、デザインに生成されたハードウェア協調シミュレーション ブロックに完全に含まれます。共有メモリを含むハードウェア協調シミュレーション ブロックで協調シミュレーションを実行すると、デザインのロジックとホスト PC ソフトウェアで FPGA 上の共通アドレス空間を共有できます。

メモ：ハードウェア共有メモリの名前は、元の共有メモリ ブロックで使用された共有メモリの名前と同じです。たとえば、共有メモリ ブロックの名前が **my_memory** である場合、このブロックのハードウェア インプリメンテーションには **my_memory** という名前を使用してアクセスします。

FPGA 内に組み込まれたすべての共有メモリは、それぞれの協調シミュレーション ブロックによりシミュレーションの前に自動的に作成され、初期化されます。これは、ハードウェア共有メモリにアクセスするその他の共有メモリ オブジェクトでは、[Ownership and initialization] パラメータを [Owned and initialized elsewhere] に設定する必要があります。このように設定すると、ソフトウェア ベースの共有メモリが FPGA 内の共有メモリに関連付けられます。

共有メモリ ペアのコンパイル

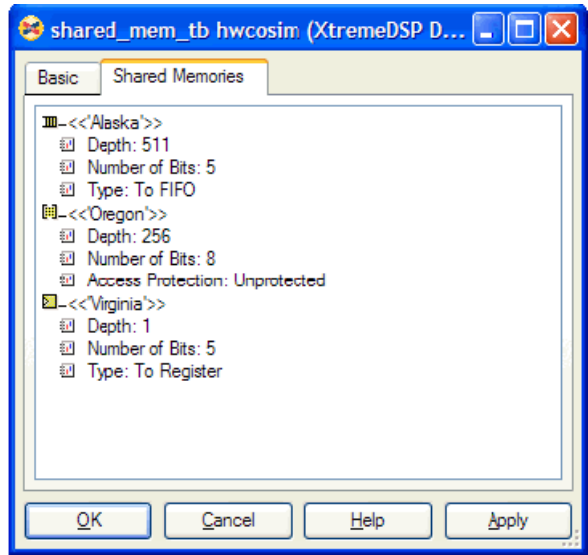
共有メモリのペア (同じ名前の 2 つの共有メモリ) をハードウェア協調シミュレーション用にコンパイルすることも可能です。この場合、これらの 2 つの共有メモリはコンパイル中に 1 つのハードウェア インプリメンテーションに統合されます。単一共有メモリとは異なり、共有メモリ ペアの両側が System Generator ユーザー デザイン ロジックに接続されます。たとえば次の図では、To FIFO と From FIFO の共有メモリ ペアのハードウェア インプリメンテーションを示します。



共有メモリの両側がユーザー デザイン ロジックに接続されるので、これらの共有メモリにホスト PC から直接通信することは不可能です。

共有メモリ情報の表示

ハードウェア協調シミュレーション ブロックでは、デザインの一部としてコンパイルされている共有メモリに関する情報を表示できます。共有メモリを含むハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスには、次に示すように [Shared Memories] タブがあります。このタブをクリックすると、デザインに含まれる各共有メモリの情報が表示されます。



共有メモリの情報には、デザインに含まれる各共有メモリのタイプ、ビット幅、ワード数が示されています。Shared Memory ブロックの場合は、[Access Protection] モードも示されます。共有メモリアイコンの横にあるプラス記号 (+) またはマイナス記号 (-) をクリックすると、そのメモリの情報を表示または非表示にできます。

次の表に、各共有メモリのアイコンを示します。

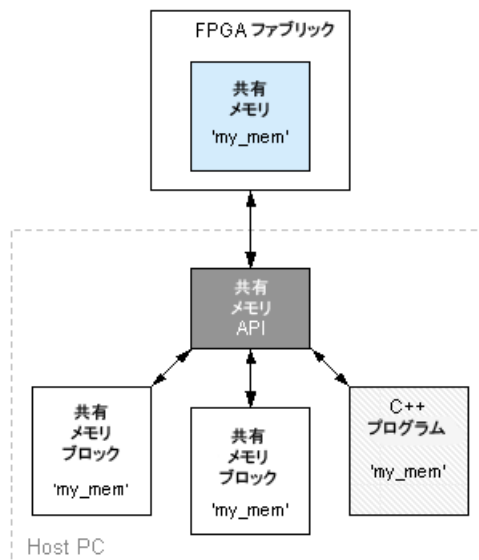
メモリ タイプ	アイコン
Shared Memory	
From FIFO/ To FIFO	
From Register/ To Register	

非保護の共有メモリの協調シミュレーション

非保護の共有メモリブロックは、いつでも書き込みまたは読み出しを実行できます。このタイプのメモリでは、アクセスは排他的ではありません。非保護のハードウェア共有メモリに対するデータ転送は、ロック共有メモリで使用する高速データ転送とは異なり、ワードごとに行われます。ソフトウェアとハードウェアの間でデータに一貫性を持たせるため、ハードウェアとソフトウェアで共有メモリデータの1つのイメージが共有されます。このイメージは、デュアルポートメモリを使用してFPGAに保存されます。System Generatorでは、ハードウェアデザインロジックとホストPC上のその他のソフトウェアベースの共有メモリオブジェクトで、共有メモリデータに同時にアクセスすることが可能です。ソフトウェア共有メモリオブジェクトで共有メモリに対してデータの読み出しまたは書き込みが実行されると、プロキシによりハードウェアメモリリソースとの通信が処理されます。

次の図に、ホストPCで実行される3つの共有メモリと通信する、FPGAにインプリメントされた非保護の共有メモリの例を示します。この例では、ソフトウェア共有メモリオブジェクトは、同じ共有メモリ名 `my_mem` を指定することにより、ハードウェア共有メモリにアクセスします。ソフトウェア共有メモリの視点からは、共有メモリリソースのインプリメンテーションは関係なく、ハードウェア共有メモリはほかの共有メモリオブジェクトと同様に扱われます。共有メモリに対する読み出しおよび書き込みは、共有メモリAPIで実行されます。

メモ：すべての共有メモリオブジェクトをSimulink環境で作成し、実行する必要はありません。次の図のC++アプリケーションは、共有メモリAPIを使用してハードウェア共有メモリと通信する外部アプリケーションの単なる1例です。

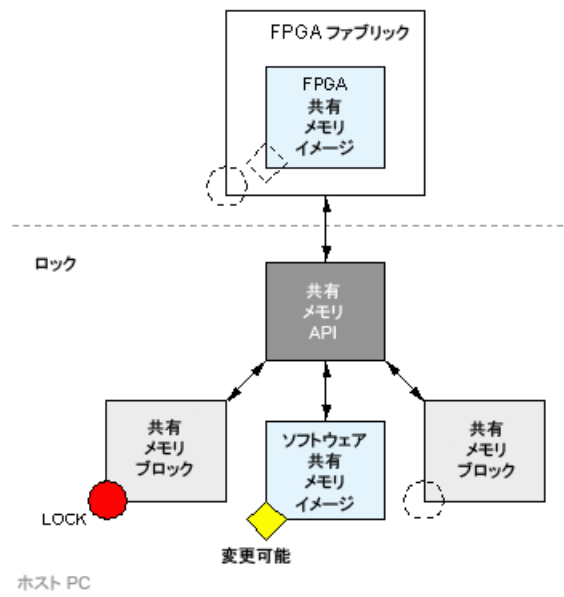


ロック共有メモリの協調シミュレーション

ロック アクセス モードでは、System Generator 協調シミュレーション ハードウェアで共有メモリオブジェクトの内容にアクセスする際、ロックを取得する必要があります。ハードウェアで共有メモリのロックが取得されると、メモリの内容が高速データ転送を使用して FPGA に転送されます。この方法では、メモリ バンド幅の広い System Generator ハードウェア協調シミュレーション デザインをインプリメントできます。この方法の詳細は、「ハードウェア協調シミュレーションを使用したリアルタイム信号処理」を参照してください。

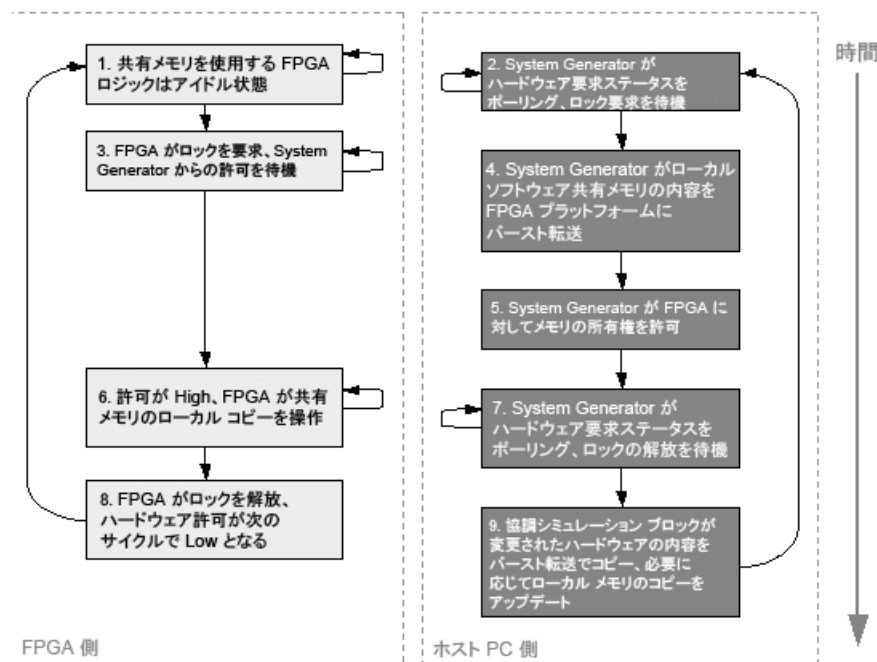
ロック共有メモリで協調シミュレーションを実行する際は、非保護の共有メモリとは異なり、共有メモリデータの 2 つのイメージが使用されます。1 つのイメージは、FPGA のデュアルポートメモリを使用して保存されます。このイメージは、System Generator ハードウェア協調シミュレーション デザインと協調シミュレーション インターフェイス ロジックによりアクセスされます。もう 1 つのイメージは、ホスト PC の共有メモリ オブジェクトとしてインプリメントされます。このソフトウェア共有メモリ イメージは、デザインで使用するソフトウェア共有メモリ オブジェクトによりアクセスされます。

ロック アクセス モードでは、共有メモリにアクセスするソフトウェア プロセスまたはハードウェア回路は、まずロックを取得する必要があります。ハードウェアがメモリのロックを保持している場合は、ソフトウェア オブジェクトでメモリの内容にアクセスできません。同様に、ソフトウェア オブジェクトがメモリに対して操作を実行している場合は、ハードウェアでメモリに対して読み出しまたは書き込みを実行することはできません。ロック ハードウェア共有メモリには、排他的なアクセスを制御するための追加ロジックが含まれます。ハードウェアおよびソフトウェアのロック共有メモリのやり取りを、次の図に示します。



赤色の丸は、ロック トークンを示します。このトークンは、ハードウェアまたはソフトウェアのどちらかにインプリメントされているかにかかわらず、どの共有メモリ オブジェクトにも渡すことができます。破線の丸はロックのプレースホルダを示し、そのブロックにロックを渡すことができることを示します。黄色のひし形は、変更可能なトークンを示します。このトークンは、ハードウェアがメモリのロックを保持している場合、ハードウェア共有メモリ イメージが変更可能であることを示します。同様に、ソフトウェア共有メモリ オブジェクトがロックを保持している場合、ソフトウェア共有メモリ イメージが変更可能です。

2つの共有メモリ イメージがあるため、イメージが一貫するようソフトウェアとハードウェアの間で同期化が必要になります。この同期化は、ロック転送の際にソフトウェアとハードウェアの間でメモリ イメージを転送することにより行われます。**System Generator** は、ホスト PC と FPGA 間で高速データ転送を実行します。次の図に、これらの転送に関するセマンティクスを示します。



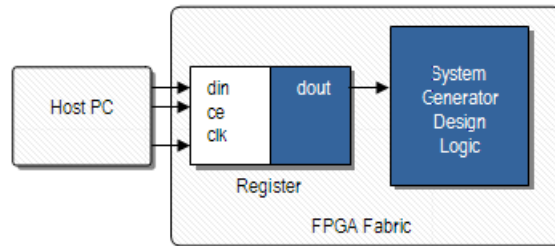
共有レジスタの協調シミュレーション

To Register と **From Register** の共有レジスタのペアを生成し、FPGA ハードウェアで協調シミュレーションを実行できます。共有レジスタ ペアとして認識させるには、**To Register** ブロックと **From Register** ブロックに同じ名前を付けます。ハードウェアでは、共有レジスタは合成可能なレジスタ コンポーネント (VHDL) またはモジュール (Verilog) でインプリメントされます。このセクションでは、単一の共有レジスタと共有レジスタのペアが、ハードウェア協調シミュレーションでどのように動作するかを説明します。

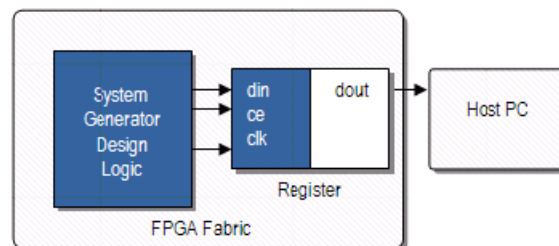
共有レジスタのペアを含むデザインをハードウェア協調シミュレーション用にコンパイルすると、ペアは 1 つのレジスタ インスタンスに置き換えられ、レジスタの両側が元の **System Generator** ロジックからのユーザー デザイン ロジックに接続されます。**Multiple Subsystem Generator** ブロックを使用してコンパイルしたデザインとは異なり、ハードウェア レジスタのすべてのポートが同じクロック ドメインの信号に接続されます。この場合、すべてのレジスタ ポートがユーザー デザイン ロジックに接続されているので、レジスタの制御は PC と FPGA ハードウェアの間で共有されません。共有レジスタのペアをハードウェアにコンパイルするのは、**System Generator** の **Register** または **DDS Compiler 4.0** ブロックをコンパイルするのと同様です。

単一の **To Register** または **From Register** ブロックをハードウェア協調シミュレーション用にコンパイルした場合、インプリメンテーションは異なります。**To Register** または **From Register** ブロックは 1 つのレジスタに置き換えられますが、レジスタは PC インターフェイスと FPGA ロジックの両方に接続されます。元のモデルのレジスタ側はユーザー デザイン ロジックに接続され、もう一方は PC と通信するデータ ポートおよび制御ポートに接続されます。

たとえば次の図では、**From Register** ブロックをハードウェア協調シミュレーション用にコンパイルすると、**dout** レジスタポートはユーザーデザインに接続され、**din**、**ce**、**clk** レジスタポートは PC と通信する制御ポートおよびデータポートに接続されます。この方法では、PC で **System Generator** のハードウェア協調シミュレーションインターフェイスを使用してレジスタに書き込みを実行することが可能です。



To Register ブロックをハードウェア協調シミュレーション用にコンパイルすると、次の図に示すように、入力ポートがユーザーロジックに接続され、出力ポートが PC インターフェイスロジックに接続されます。ハードウェア協調シミュレーション中、もう一方の共有レジスタ (**To Register** または **From Register** ブロック)、C プログラムまたは実行ファイル (**System Generator API**)、あるいは **MATLAB** プログラムを使用して共有レジスタにアクセスできます。



ハードウェア協調シミュレーションを使用するデザインでは、通常共有レジスタのペアはソフトウェアと FPGA ハードウェアに分散されます。つまり、ペアの一方は FPGA にインプリメントされ、もう一方は **To Register** または **From Register** を使用してソフトウェアでシミュレーションされます。ソフトウェアの **To Register** ブロックにデータが書き込まれると、ハードウェアレジスタに同じデータが書き込まれ、データがハードウェアレジスタに書き込まれると、同じデータが **From Register** ブロックにより読み出されます。ソフトウェア共有レジスタは、ハードウェア協調シミュレーション用にコンパイルされた共有レジスタの名前を指定することにより、ハードウェア共有レジスタと接続できます。

メモ： FPGA 協調シミュレーション デザインに組み込まれたすべての共有メモリの名前は、ハードウェア協調シミュレーションブロックのパラメータダイアログボックスの **[Shared Memories]** タブに示されます。

ソフトウェア/ハードウェア共有メモリペアで協調シミュレーションを実行すると、**System Generator** で PC と FPGA ハードウェア間のトランザクションが制御されます。ソフトウェアでシミュレーションされる共有レジスタペアの動作は、PC と FPGA ハードウェアに分散された共有レジスタペアと同じになるはずです。

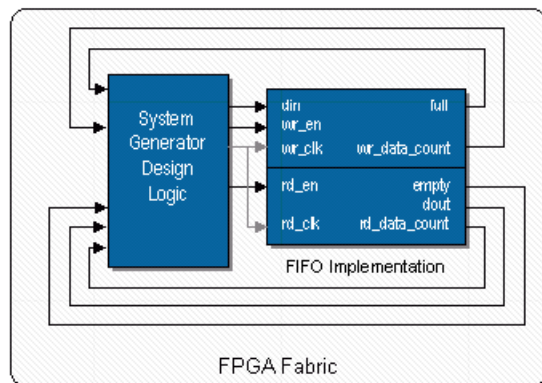
共有 FIFO の協調シミュレーション

To FIFO と **From FIFO** の共有 FIFO のペアを生成し、ハードウェアで協調シミュレーションを実行できます。共有 FIFO ペアとして認識させるには、**To FIFO** ブロックと **From FIFO** ブロックに同じ名前を付けます。共有 FIFO は、**FIFO Generator** コアを使用してハードウェアにインプリメントされます。このコアは、独立したクロック (非同期) とデータ保存用にブロック メモリを使用してコンフィギュレーションされます。このセクションでは、共有 FIFO の協調シミュレーションが有益な理由と、これらのブロックのハードウェアでの動作を説明します。

非同期 FIFO は、一般的に複数クロック アプリケーションでクロック ドメインを切り替えるために使用されます。ハードウェア協調シミュレーションで**フリーランニング クロック** モードを使用している場合、FPGA は **Simulink** シミュレーションとは非同期に動作します。フリーランニング クロック モードを使用すると、**Simulink** シミュレーション クロック ドメインと **FPGA** フリーランニング クロック ドメインの 2 つのクロック ドメインができることになります。これらのデザインで共有 FIFO を使用すると、ホスト PC と FPGA プラットフォームの間で安全に確実にデータを転送できます。

共有 FIFO は、協調シミュレーション中のバースト転送もサポートするので、データのベクタまたはフレームを作成し、1 つのトランザクションで **FPGA** に転送することが可能です。これらのインターフェイスを使用して、通常のハードウェア協調シミュレーションで可能なシミュレーション スピード以上のスピードを達成できます。詳細は、「[ハードウェア協調シミュレーションを使用したフレーム ベースの シミュレーション](#)」を参照してください。

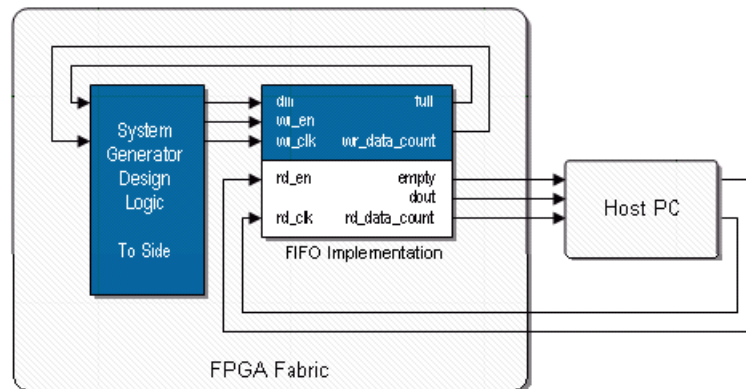
共有 FIFO ペアを協調シミュレーション用に生成すると、2 つの共有 FIFO ブロックが 1 つの非同期 FIFO コアに置き換えられます。次の図に示すように、FIFO の読み出し/書き込み側は、**From FIFO** および **To FIFO** ブロックに接続されたユーザー デザイン ロジック (元の **System Generator** モデルから作成されたロジック) に接続されます。FIFO の両側がハードウェアのユーザー ロジックに接続されるので、PC とデザインで FIFO の制御は共有されず、FIFO は通常の FIFO ブロックを含む **System Generator** デザインと同様に動作します。



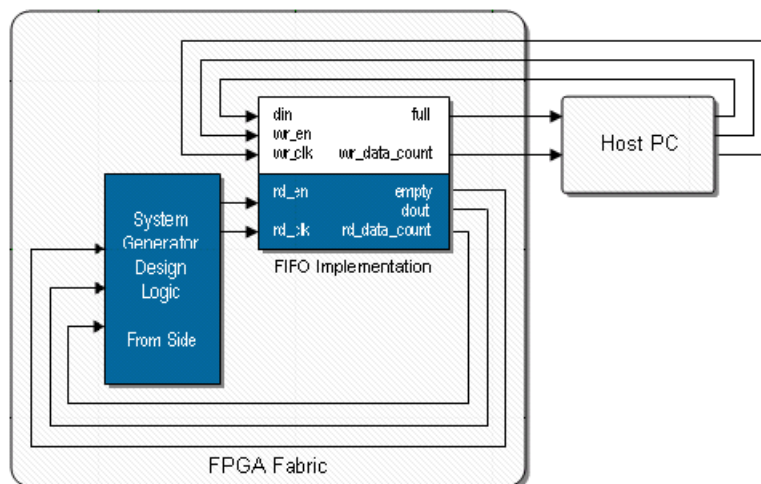
FIFO にはそれぞれ独立したクロック ポートがありますが、FIFO ペアをコンパイルすると、両方のクロック ポートが同じ協調シミュレーション クロックで駆動されます。これは、共有 FIFO ペアを **Multiple Subsystem Generator** ブロックを使用してコンパイルする場合と異なります。この場合、クロックは異なるクロック ドメインから供給されます。

1 つの共有 FIFO ブロックは、共有 FIFO ペアとは処理が異なります。1 つの To FIFO または From FIFO ブロックをハードウェア協調シミュレーション用にコンパイルすると、1 つの非同期 FIFO コアに置き換えられ、FIFO の一方 (System Generator で使用されていない共有 FIFO 側) は PC インターフェイス ロジックに接続され、もう一方は元の To FIFO または From FIFO ブロックに接続されたユーザー ロジック デザインに接続されます。この方法では、FIFO は PC と FPGA デザインの両方で制御されます。

次の図に示すように、To Register ブロックをハードウェア協調シミュレーション用にコンパイルすると、FIFO の書き込み側はユーザー デザインの To FIFO に接続された同じロジックに接続されます。読み出し側は PC インターフェイスに接続され、シミュレーション中に PC で FIFO からデータを読み出すことができます。



次の図は、From FIFO ブロックがハードウェア協調シミュレーション用にコンパイルした場合を示しており、接続は逆になっています。FIFO の書き込み側が PC インターフェイス ロジックに接続され、読み出し側がユーザー デザイン ロジックに接続されます。ホスト PC により FIFO にデータが書き込まれ、デザイン ロジックで FIFO からのデータを読み出すことができます。



ハードウェア協調シミュレーションを使用するデザインでは、通常共有 FIFO のペアはソフトウェアと FPGA ハードウェアに分散されます。つまり、ペアの一方は FPGA にインプリメントされ、もう一方は To FIFO または From FIFO を使用してソフトウェアでシミュレーションされます。ソフトウェア部分とハードウェア部分で、1 つの完全な非同期 FIFO が構成されます。ソフトウェア/ハードウェア共有 FIFO ペアで協調シミュレーションを実行すると、System Generator で PC と FPGA ハードウェア間のトランザクションが制御されます。

シミュレーション中にソフトウェアの To FIFO ブロックにデータが書き込まれると、ハードウェア FIFO に同じデータが書き込まれ、FIFO を読み出すことによりハードウェアのデザインでこのデータを取得できます。同様に、データがハードウェア FIFO に書き込まれると、同じデータを From Register ブロックにより読み出すことができます。共有 FIFO ブロックの empty、full、rd_data_count、wr_data_count により、対応するハードウェア FIFO のステートを判断できます。ソフトウェア共有 FIFO は、ハードウェア協調シミュレーション用にコンパイルされた共有 FIFO の名前を指定することにより、ハードウェア共有 FIFO と接続できます。

メモ: FPGA 協調シミュレーション デザインに組み込まれたすべての共有メモリの名前は、ハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスの [Shared Memories] タブに示されます。

共有メモリの制限

共有メモリ、共有レジスタ、共有 FIFO ブロックをハードウェア協調シミュレーションで使用する System Generator デザインには、次の制限が適用されます。

- 共有メモリのアクセス保護モードは、ハードウェア協調シミュレーション用にコンパイルした後は変更できません。
- 共有メモリのアドレス ポートの幅は 24 ビット以下で、16,777,216 ワードのアドレス空間にアクセスできます。
- 共有メモリ、レジスタ、FIFO のデータ ポート幅は、現在のところ 32 ビット以下に制限されています。
- 共有メモリおよび FIFO は、ハードウェアではブロック メモリを使用してインプリメントされます。分散メモリまたは外部メモリでのインプリメンテーションはサポートされていません。
- 同じ名前の共有メモリを 3 つ以上ハードウェア協調シミュレーション用にコンパイルすることはできません。
- 同じ共有メモリ名を持つハードウェア協調シミュレーション ブロックを、同じデザインで同時に 2 つ以上使用することはできません。

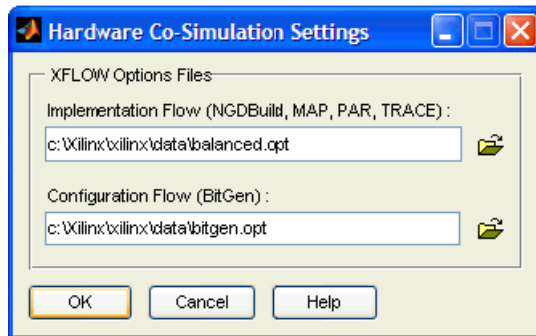
ザイリンクス ツール フローの設定

デザインを System Generator ハードウェア協調シミュレーション用にコンパイルすると、XFLOW というコマンド ライン ツールを使用して、デザインが選択した FPGA プラットフォーム用にインプリメントおよびコンフィギュレーションされます。XFLOW では、コンパイルでデザインに実行する必要のあるプログラムのシーケンスを指定するさまざまなフローが定義されています。必要な出力 (ハードウェア協調シミュレーションの場合はコンフィギュレーション ビットストリーム) を得るためには、通常複数のフローを実行する必要があります。

コンフィギュレーション ビットストリームの生成には、インプリメンテーションとコンフィギュレーションの 2 つのフローが使用されます。インプリメンテーション フローでは、合成ツールのネットリスト出力 (EDIF または NGC) が配置配線された NCD ファイルにコンパイルされます。これには、ザイリンクス ツール NGDBuild、MAP、および PAR が使用されます。インプリメンテーション フローではタイミング解析のために TRACE も実行できますが、通常のコンパイルプロセスでは実行されません。コンフィギュレーション フローでは、配置配線された NCD ファイルを入力として使用して、FPGA ビットストリームを作成するのに必要なツールが実行されます。

インプリメンテーション フローとコンフィギュレーション フローには、それぞれ XFLOW オプション ファイルが関連付けられています。XFLOW オプション ファイルでは、そのフローで実行する必要のあるプログラムとコマンド ライン オプションが定義されます。ハードウェア協調シミュレーションの各コンパイル ターゲットに対して、デフォルトのコンフィギュレーション オプションを定義するオプション ファイルが提供されています。場合によって、デフォルトのオプション ファイルとは異なる設定を使用することがあります (PAR の配置エフォート レベルを上げるなど)。そのような場合は、独自のオプション ファイルを作成するか、デフォルトのオプション ファイルを変更します。

次の図に示す [Hardware Co-Simulation Settings] ダイアログ ボックスで、使用するオプション ファイルを指定できます。



[Hardware Co-Simulation Settings] ダイアログ ボックスのパラメータは、次のとおりです。

- **[Implementation Flow]** : インプリメンテーション フローで使用するオプション ファイルを指定します。デフォルトでは、コンパイル ターゲットに提供されているインプリメンテーション オプション ファイルが使用されます。
- **[Configuration Flow]** : コンフィギュレーション フローで使用するオプション ファイルを指定します。デフォルトでは、コンパイル ターゲットに提供されているコンフィギュレーション オプション ファイルが使用されます。

ザイリンクス ISE® ソフトウェアに、XFLOW オプション ファイルの例が含まれています。これらのファイルは、ISE のインストール ディレクトリの `xilinx\data` ディレクトリに含まれています。よく使用されるインプリメンテーション オプション ファイルは、次のとおりです。

- `balanced.opt`
- `fast_runtime.opt`
- `high_effort.opt`

メモ : 指定したオプション ファイルにより、System Generator のハードウェア協調シミュレーション フローでエラーが発生することがあるので、デフォルトのオプション ファイルを変更する場合はバックアップ コピーを作成してください。また、ほとんどの FPGA ハードウェア プラットフォームには特定のコンフィギュレーション パラメータ要件があるので、コンフィギュレーション オプション ファイルを変更する際は注意が必要です。

ハードウェア協調シミュレーションを使用したフレーム ベースのシミュレーション

プログラマブル デバイスの大型化および処理能力の向上に伴い、シミュレーション時間が長くなっています。デザインのサイズおよび複雑さによっては、シミュレーションに数日かかることもあります。ほとんどのシステムでは、デザインが正常に機能し、使用できるようにするためには、シミュレーションを何回も実行する必要があるため、これは深刻な問題です。この問題に対処するため、**System Generator** では、**FPGA** デザインのシミュレーションを大幅に高速化するハードウェア協調シミュレーション インターフェイスが提供されています。

ハードウェア協調シミュレーションを使用してどの程度の高速化を達成できるかには、デザインのサイズ、モデルのポート数、ハードウェアのオーバーサンプリング レートなど、いくつかの要素が影響します。通常の操作では、**Simulink** の各シミュレーション サイクルごとに **PC** が **FPGA** と通信します。このソフトウェアとハードウェア間のトランザクションは大きなオーバーヘッドとなり、シミュレーション パフォーマンスを制限します。使用する協調シミュレーション インターフェイスも影響を及ぼします。**PCI** などのインターフェイスは、**JTAG** などのほかのインターフェイスよりも高速です。ある程度大型のデザインでは、通常ハードウェア協調シミュレーションによりシミュレーションが 1 桁高速化されます。

上記の点を考慮した上で、シミュレーション パフォーマンスをさらに向上させる方法がいくつかあります。**PC** がハードウェアと通信するたびに、オーバーヘッドが発生します。**FPGA** トランザクションの数を削減するには、**Simulink** ベクタとフレーム信号を使用するのが 1 つの方法です。このセクションでは、**Simulink** ベクタとフレーム信号を使用する **FPGA** トランザクションを「ベクタ転送」と呼びます。この方法では、できるだけ多くの入力データ サンプルを 1 つにまとめ、**FPGA** でこれらを 1 つのトランザクションで処理させます。**FPGA** でのトランザクションの数が減れば、シミュレーション パフォーマンスが向上します。

このチュートリアルでは、**Simulink** ベクタとフレーム信号を使用して、通常ハードウェア協調シミュレーション以上のシミュレーション パフォーマンスを達成します。この方法を、フィルタ デザイン例を使用して説明します。

詳細を説明する前に、どのようにこれを達成するかを概念を説明します。要約すると、**Simulink** のシミュレーションで次を実行することになります。

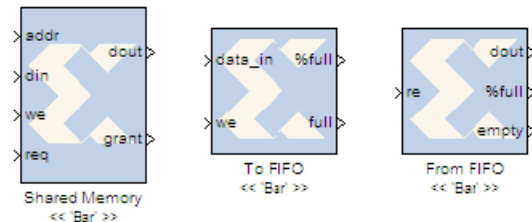
- 複数のスカラー入力データ値をバッファを介して **Simulink** ベクタに格納します。
- バースト転送を使用して、ベクタ データを **FPGA** 上のバッファに転送します。
- **FPGA** をフリーランニング クロック モードを使用して、入力バッファ全体を順に処理します。
- **FPGA** でデータを出力バッファに書き込みます。
- 出力バッファの内容を **Simulink** に転送し、**Simulink** ベクタのデータを構築します。
- ベクタを複数の出力スカラー値に変換します。

共有メモリ

System Generator でベクタ転送を使用できるようにするには、入力バッファと出力バッファを追加する必要があります。ハードウェアでは、これらのバッファは内部メモリ (BRAM など) を使用してインプリメントされ、PC により FPGA に書き込まれるシミュレーション データおよび FPGA から読み出されるシミュレーション データのベクタを格納するのに使用されます。バッファの最大サイズは、ターゲット デバイス上で使用可能な内部メモリの容量によって制限されます。System Generator では、これらのバッファをインプリメントするインターフェイスとして共有メモリ ブロックが提供されています。

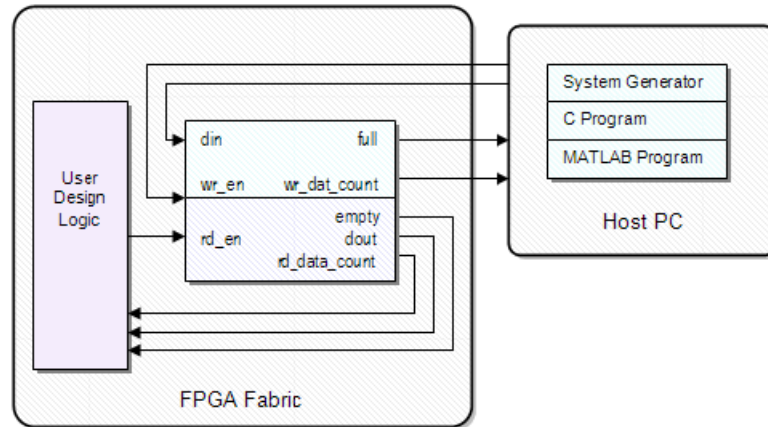
標準の FIFO またはメモリ ブロックを使用しないのは、ハードウェア協調シミュレーションで使用されるバッファは、標準の FIFO およびメモリとは異なり、PC および FPGA ユーザー デザイン ロジックの両方で制御する必要があるからです。System Generator で提供されている標準の FIFO およびメモリ ブロックは、ユーザー デザイン ロジックにしか接続できません。

この制御が可能な共有メモリには、ロック共有メモリと共有 FIFO があります。これらのブロックのデータ格納方法、および FPGA とのバースト転送のタイミングと方法を指定するハンドシェイク プロトコルは異なります。このチュートリアルでは、主に共有 FIFO バッファに焦点を置きます。ロック共有メモリの使用方法の例は、「[ハードウェア協調シミュレーションを使用したリアルタイム信号処理](#)」を参照してください。ロック共有メモリと FIFO ブロックは、[Xilinx Blockset] → [Shared Memory] ライブラリにあります。



共有 FIFO はベクタ転送を可能にする上で重要な役割を果たすので、まず共有 FIFO の動作について簡単に説明します。共有 FIFO のペアは、同じ名前 (上図では Bar) を指定した To FIFO ブロックと From FIFO ブロックで構成されます。To FIFO ブロックは書き込み側の制御信号を供給し、From FIFO ブロックは書き込み側の制御信号を供給します。共有 FIFO のペアは概念的には 1 つの FIFO と同じで、両側の制御信号はグラフィック上では接続されていませんが、同じ FIFO メモリ空間を共有します。たとえば、To FIFO ブロックにデータを書き込むと、同じデータを From FIFO ブロックから読み出すことができます。ペアの 2 つのブロック間の接続は、Simulink ワイヤで明示的に指定されているわけではなく、名前によって関連付けられます。

共有 FIFO および共有メモリは、ハードウェア協調シミュレーション用にコンパイルできます。このチュートリアルでも共有 FIFO の協調シミュレーション方法については簡単に説明しますが、詳細は「[共有 FIFO の協調シミュレーション](#)」を参照してください。共有ブロックの一方がハードウェア協調シミュレーション用にコンパイルされると、FIFO ブロック全体が FIFO Generator コアを使用して FPGA に組み込まれます。FIFO の一方はユーザー デザイン ロジック (共有 FIFO ロジックに接続されている System Generator ロジック) に接続され、もう一方は PC と通信するためのインターフェイス ロジックと接続されます。FIFO のインターフェイス ロジックと接続された側は、その他の System Generator ソフトウェア モデル ロジック (共有 FIFO のもう一方など)、C プログラムまたはソフトウェア実行ファイル、あるいは MATLAB プログラムで制御できます。共有 FIFO をハードウェア協調シミュレーション用にコンパイルすると、PC で直接制御可能なエンベデッド FIFO スタイルのバッファが FPGA に作成されます。

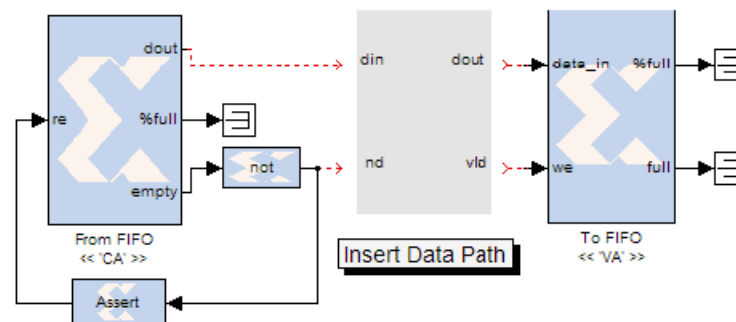


FPGA 内に組み込まれた共有 FIFO と通信するには、いくつかの方法があります。最も一般的なのは、共有 FIFO のもう一方を System Generator デザインに含める方法です。また、C プログラムまたは MATLAB プログラムを使用して共有 FIFO と通信することも可能です。System Generator では、FIFO とのベクタ転送をサポートするブロックが提供されています。これらのブロックについては、このチュートリアルの後の方で説明します。これらのブロックは、FPGA とのバースト転送をサポートするのに重要な役割を果たします。

デザインへのバッファの追加

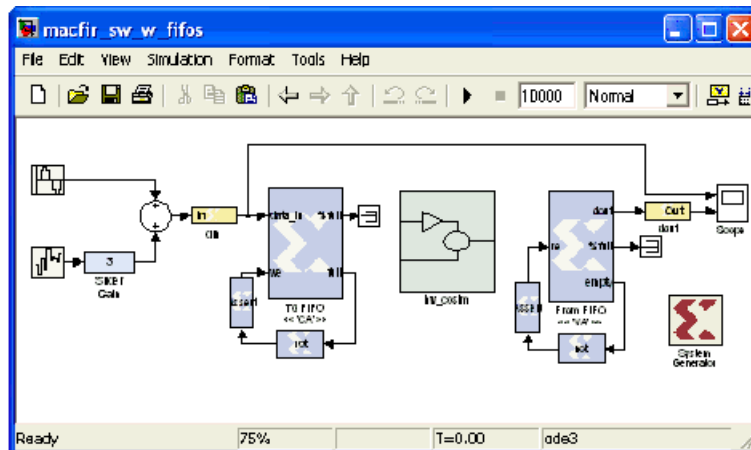
これらのバッファを FPGA で的高速ベクタ処理に使用するデザインを構築します。

FPGA データパスをベクタ転送を使用して高速化するとします。この場合、PC により書き込まれるデータ入力サンプルを保存する入力バッファを FPGA に含める必要があります。また、PC でデータが読み出されるまで FPGA で処理済のデータを格納する出力バッファも必要です。入力データバッファのインプリメントには From FIFO ブロックを使用し、出力データバッファのインプリメントには To FIFO ブロックを使用します。次の図に示すモデルでは、データが入力 FIFO に示されるとすぐにデータパスに書き込まれます。データパスブロックには、新規データ用の制御ポート (nd) とデータバリッド制御ポート (vld) があります。これらのポートは、新規データが入力され、有効なデータが出力されることを判断するために使用されます。入力 FIFO にデータが示されると nd 信号がアサートされ、データパスに有効なデータが存在すると、出力 FIFO にデータが書き込まれます。

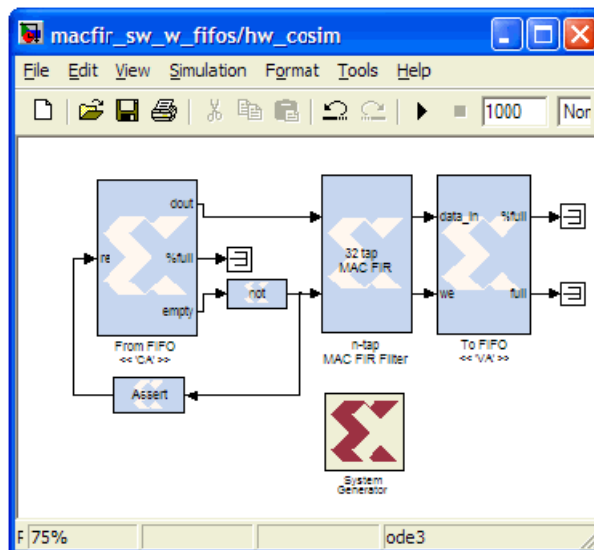


共有 FIFO がどのように使用されるかを理解するため、MAC フィルタ デザインの高速化にベクタ転送を使用する例を示します。

1. MATLAB ウィンドウで、<path_to_sysgen>\examples\shared_memory\hardware_cosim\frame_acc ディレクトリに移動します。
2. macfir_sw_w_fifos.mdl を開きます。



このデザイン例は、sinusoid 入力ソースからホワイト ノイズを除去する 32 タップ MAC FIR フィルタをインプリメントします。シミュレーション前またはシミュレーション中に Slider Gain 制御バーを動かすことにより、ホワイト ノイズの量を動的に変更できます。出力スコープは、フィルタされた出力データとフィルタ前の入力データを比較します。MAC フィルタ自体は、hw_cosim というサブシステムに含まれています。このサブシステムには、ハードウェア協調シミュレーション用に FPGA にコンパイルされるロジックがすべて含まれています。デザインのその他のコンポーネント (最上位のすべてのブロック) は、デザイン テストベンチであると考えられます。

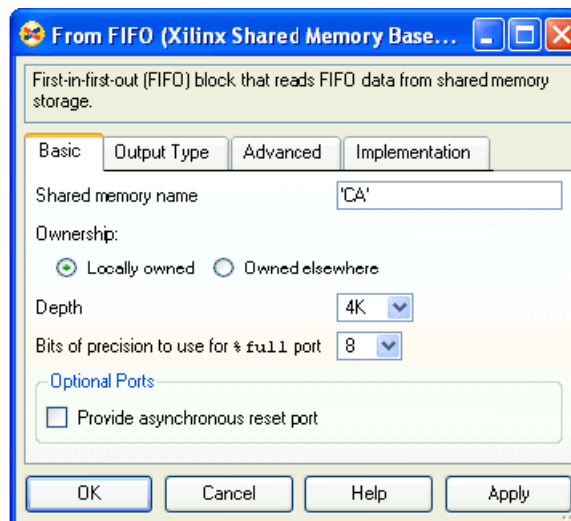


hw_cosim サブシステムを開くと、デザイン データ パスをインプリメントする n-tap MAC FIR Filter ブロックがあります。フィルタの前後には、入力バッファとなる From FIFO と出力バッファとなる To FIFO ブロックがあります。このデザインの MAC フィルタは、[Xilinx Reference Blockset] → [DSP] ライブラリに含まれる n-tap MAC FIR Filter ブロックを変更したもので、FIFO フロー制御をサポートするためにバリッド入力ポートとバリッド出力ポートが追加されています。

このデザインには、CA および VA 共有 FIFO ペアを定義する 4 つの共有メモリ ブロックがあります。このデザインをハードウェア協調シミュレーション用にコンパイルするのに必要なのは、hw_cosim サブシステムに含まれる共有 FIFO ブロックのみですが、FPGA ハードウェアを含む完全なデザインをシミュレーションするので、テストベンチ ロジックには To FIFO ブロック CA と From FIFO ブロック VA が含まれています。これらの共有 FIFO ブロックは、hw_cosim サブシステム内の共有 FIFO に対してテスト データの書き込みおよび読み出しを実行します。

Gateway In ブロック din から入力されたフィルタ前のデータは、To FIFO ブロック CA に書き込まれます。この時点で、hw_cosim サブシステム内の From FIFO ブロック CA が FIFO からデータを読み出し、MAC フィルタに書き込みます。MAC フィルタはこのデータを処理し、出力バッファである To FIFO ブロック VA に書き込みます。最後に、最上位の From FIFO ブロック VA がデータを読み出し、Scope ブロックに送信します。

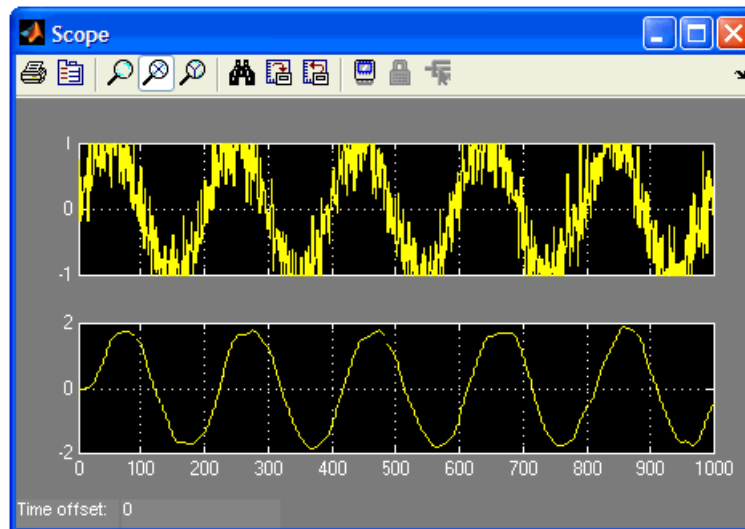
この例では、最大バッファ サイズ 4K が選択されています。このパラメータを設定するには、From FIFO ブロック CA と To FIFO ブロック VA のパラメータ ダイアログ ボックスを開き、[Depth] を [4K] に設定します。共有 FIFO は非同期 FIFO Generator コアを使用してインプリメントされるので、ハードウェア FIFO の実際のワード数は n-1 (n はダイアログ ボックスで指定したワード数) になります。



デザインのシミュレーションを実行して、ソフトウェアでの速度を調べます。

3. Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、ソフトウェアでデザインのシミュレーションを開始します。
4. デザインを 10000 サイクル間シミュレーションするのにかかる時間を記録します。正確に測定するため、Scope ブロックを閉じておくことをお勧めします。グラフィックのアップデートがシミュレーション パフォーマンスに影響する可能性があります。

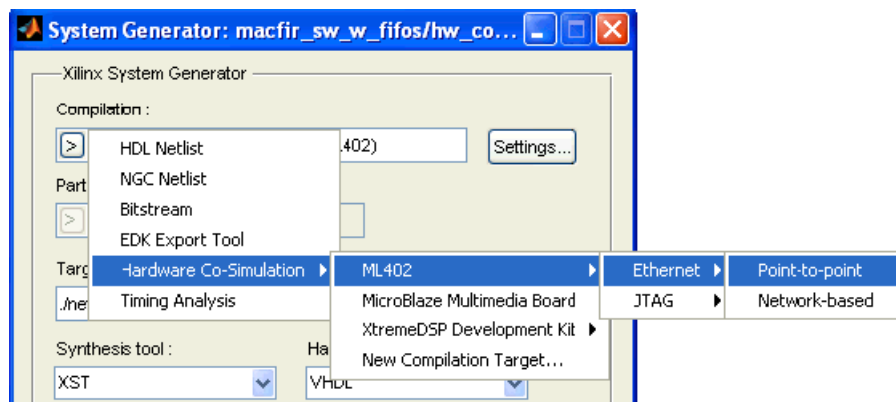
シミュレーション中に **Slider Gain** バーを動かし、ノイズが追加された場合のフィルタ パフォーマンスへの影響を確認します。フィルタ前のデータとフィルタ後のデータは、出力 **Scope** ブロックで表示できます。フィルタ前の入力データが上部に、フィルタ後のデータが下部に表示されます。



ハードウェア協調シミュレーション用のコンパイル

デザインをハードウェア協調シミュレーション用にコンパイルします。次の手順を実行する前に、**System Generator** に適切なハードウェア協調シミュレーション プラットフォームがインストールされており、PC に接続されていることを確認してください。この例では、デザインの **hw_cosim** サブシステムに含まれる部分のみをコンパイルします。これは、**To FIFO** ブロック **CA** と **From FIFO** ブロック **VA** をデザイン テストベンチの一部としてソフトウェアに保持するからです (パートナーの共有 **FIFO** ブロックは **FPGA** ロジックにコンパイル)。

5. **hw_cosim** サブシステム内の **System Generator** トークンをダブルクリックして、パラメータ ダイアログ ボックスを開きます。
6. **[Compilation]** で、適切なハードウェア協調シミュレーション ターゲットを選択します。この例ではポイントツーポイント イーサネット ハードウェア協調シミュレーション インターフェイスを使用しますが、インストールされているハードウェア協調シミュレーション プラットフォーム (JTAG 協調シミュレーションをサポートするプラットフォーム) であればどれも使用できます。



7. [Generate] ボタンをクリックします。

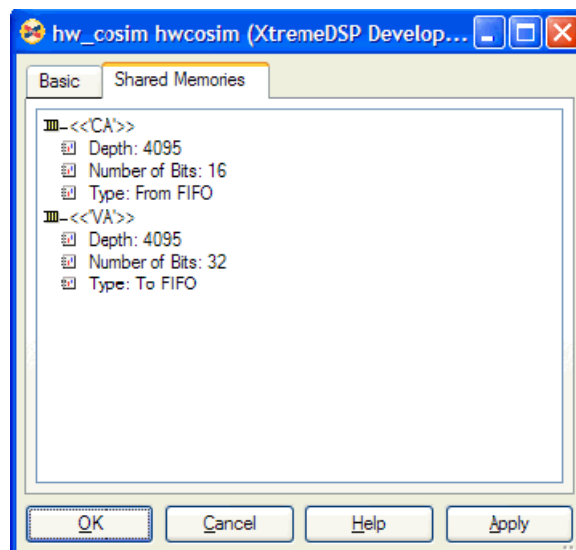
デザインのコンパイルが完了すると、新しいハードウェア協調シミュレーション ライブラリとブロックが作成されます。新しいハードウェア協調シミュレーション ブロックには、入力ポートおよび出力ポートはありません。これは、サブシステムに **Gateway** ブロックまたは **Simulink** ポートが含まれていないからです。ほかの **Simulink** ブロックとの接続は、FPGA にコンパイルされた共有メモリを介して処理されます。To FIFO ブロックと From FIFO ブロックをソフトウェア テストベンチの一部として保持しているため、シミュレーションの開始時にソフトウェア FIFO が自動的にハードウェア FIFO に接続されます。

通常、ハードウェア協調シミュレーション用にコンパイルされた共有メモリのタイプとコンフィギュレーションを確認する必要があります。各共有メモリに関する情報は、ハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスの [Shared Memories] タブに示されます。このタブには、デザインに含まれる各共有メモリに対して情報がツリー表示で示されます。

8. ハードウェア協調シミュレーション ブロックをダブルクリックして、パラメータ ダイアログ ボックスを開きます。

9. [Shared Memories] タブをクリックします。

共有 FIFO ブロック CA と VA の情報が表示されます。協調シミュレーション デザインにほかの共有メモリ ブロックが含まれる場合は、それらのブロックの情報もここに表示されます。共有メモリの横にあるプラス記号 (+) またはマイナス記号 (-) をクリックすると、共有メモリの情報を表示または非表示にできます。

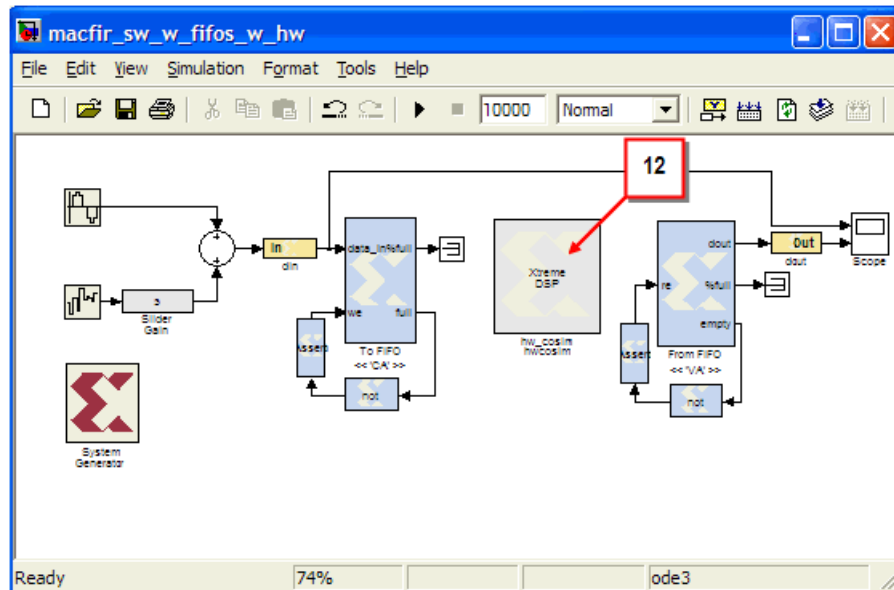


10. パラメータ ダイアログ ボックスを閉じます。

ハードウェア協調シミュレーション ブロックを元のデザインに挿入します。元のデザインに変更を加えるので、次の手順を実行する前に、デザインの名前を変更するか、バックアップ コピーを作成することをお勧めします。

11. デザインから hw_cosim サブシステムを削除します。

12. hw_cosim サブシステムのあった場所に、ハードウェア協調シミュレーション ブロックを追加します。



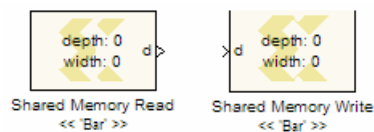
13. ハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスで、[Single stepped] をオンにし、その他の必要なパラメータを設定します。
14. Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、デザインのシミュレーションを開始します。
15. デザインを 10000 サイクル間シミュレーションするのにかかる時間を記録します。

16. デザインを閉じます。ハードウェア協調シミュレーション ライブラリは、次の手順で使用する
ので開いたままにしてください。

上記のシミュレーションでは単一ワード転送を使用しているので、読み出すシミュレーション値がある場合、ハードウェア協調シミュレーションに書き込む値がある場合ごとに、PC により FPGA とのトランザクションが実行されます。次のセクションでは、シミュレーション速度を向上するため、ハードウェア協調シミュレーションのバンド幅をより効率的に使用してベクタ転送を実行する方法を示します。

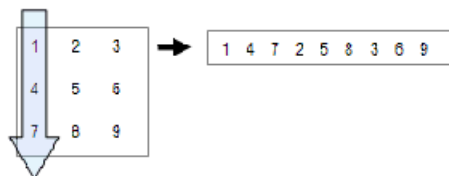
ユーザー ベクタ転送

System Generator の Shared Memory Read および Shared Memory Write ブロックを使用すると、ハードウェア協調シミュレーションでベクタ転送を使用できます。これらのブロックは、[Xilinx Blockset] → [Shared Memory] ライブラリに含まれています。



Shared Memory Write ブロックは、Simulink のスカラ、ベクタ、マトリックス、またはフレーム データを受信し、データを順に共有メモリに書き込みます。Simulink 信号のすべての内容が、1 シミュレーション サイクルで共有メモリに書き込まれます。ほかの共有メモリ ブロックと同様に、Shared Memory Read ブロックまたは Shared Memory Write ブロックとほかの共有メモリは、同じ名前を指定することにより関連付けられます。

マトリックス型では列優先順が使用され、データを共有メモリに順次書き込む場合、まず列のすべてのエレメントが書き込まれてから次の列が書き込まれます。たとえば、次のようなマトリックス データがあるとして、シミュレーション中、このマトリックス データは次のように FIFO (または共有メモリ) に書き込まれます。



これらのブロックを使用し、次の条件を満たしていれば、共有メモリに対して完全なベクタ、フレーム、またはマトリックス信号の読み出しまたは書き込みを実行できます。

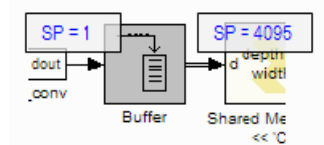
- Shared Memory Write ブロックには、8 ビット、16 ビット、32 ビットの符号付きまたは符号なしの整数型の信号を入力します。
- ベクタまたはマトリックスのエレメント 数は、共有メモリまたは FIFO のワード 数以内にします。
- Shared Memory Read または Shared Memory Write ブロックのデータ幅 (スカラのビット 幅、ベクタまたはマトリックスのエレメント) は、共有メモリまたは FIFO のデータ幅と 同じにします。

これらのブロックをデザイン例で使用して、ソフトウェア/ハードウェアの 1 トランザクションで MAC フィルタに対してデータ サンプルのベクタの読み出しおよび書き込みを実行できます。

17. macfir_hw_w_frames_tb.mdl を開きます。

このデザインは前のセクションのデザインと似ていますが、Shared Memory Read および Shared Memory Write ブロックをサポートするための変更が加えられています。デザインをシミュレーションする前に、これらの変更について説明します。まず、To FIFO および From FIFO ブロックの代わりに Shared Memory Write および Shared Memory Read ブロックが使用されています。Shared Memory Write ブロックの名前が CA、Shared Memory Read ブロックの名前が VA になっているので、シミュレーション中 FPGA ハードウェアに含まれる入力 FIFO バッファと出力 FIFO バッファに関連付けられます。

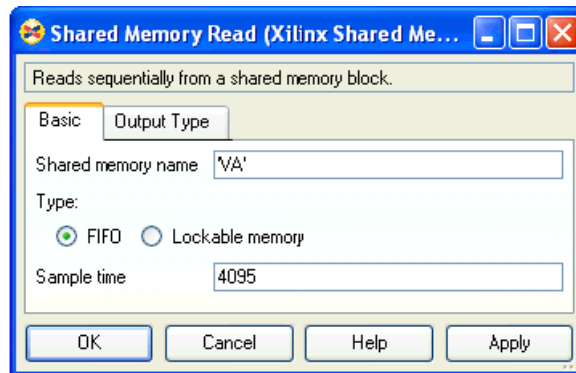
Simulink の Buffer ブロックは、フィルタ前の入力データを順に格納することにより、スカラ入力サンプルのフレームを作成します。この Buffer ブロックは、シリアルデータからパラレルデータへの変換を実行すると考えることができます。FIFO バッファはワード数 4K でコンパイルしたので、フレーム サイズは 4095 にします。



Buffer ブロックにより、デザインのサンプリング データが変更されます。4095 個の入力ごとに出力が 1 つ生成されるので、データ入力のサンプリング周期が 1 の場合、データ出力のサンプリング周期は 4095 になります。つまり、Shared Memory Write ブロックからの FPGA へのデータ フレーム送信は、4095 シミュレーション サイクルごとに実行されることになります。これは、各シミュレーション サイクルでハードウェア トランザクションを実行するのに比べてかなり効率的です。

Buffer ブロックでサンプリング レートが変更されるので、この遅いサンプリング レートに合わせてダウンストリーム ブロックを調整する必要があります。まず、Shared Memory Read ブロックでデータ フレームが 4095 シミュレーション サイクルごとに読み出されるようにします。

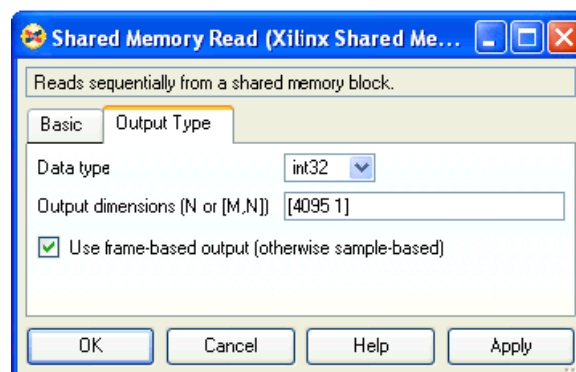
18. Shared Memory Read ブロックをダブルクリックして、パラメータ ダイアログ ボックスを開きます。



[Basic] タブの [Type] で [FIFO] をオンにします。フレームが適切なタイミングで読み出されるようにするため、[Sample time] を 4095 に設定します。

Shared Memory Read ブロックでは、出力データの型と次元を設定します。

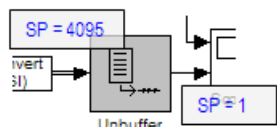
19. パラメータ ダイアログ ボックスで、[Output Type] タブをクリックします。



このタブで、いくつかのパラメータを設定します。まず、[Data type] を [int32] に設定して、フィルタのデータパスの出力幅 32 ビットに一致するようにします。データ幅が一致していないと、デザインをシミュレーションできません。次に、[Output dimensions] を 4095 ワードに設定します。最後に、ダウンストリームの Unbuffer ブロックでフレームベースの出力が必要なので、[Use frame-based output] をオンにします。

20. パラメータ ダイアログ ボックスを閉じます。

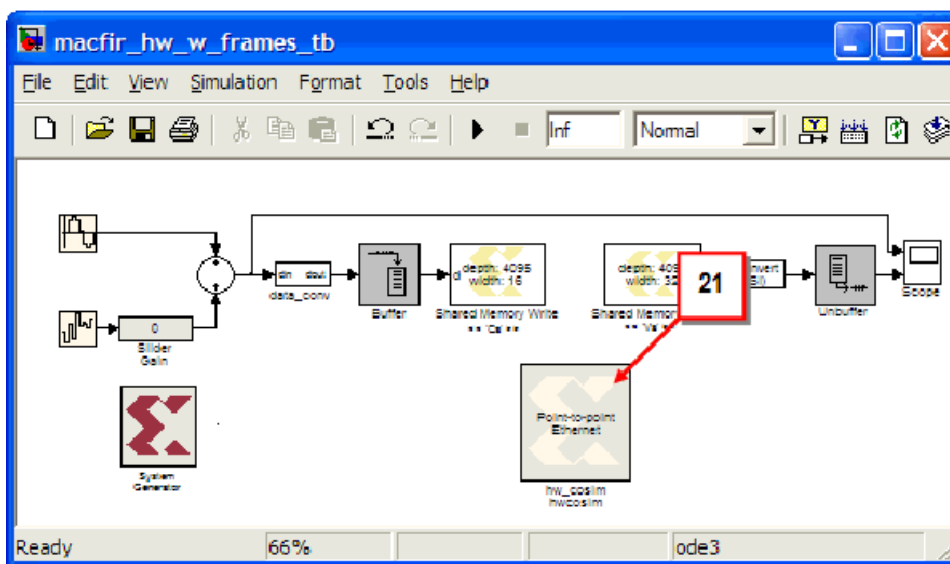
Simulink の Unbuffer ブロックは、Shared Memory Read ブロックからのフレーム データをシーケンシャルなスカラ値に変換します。Unbuffer ブロックでも、デザインのサンプリング データが変更されます。Unbuffer ブロックへの入力サンプリング周期は 4095 であり、フレーム サイズが 4095 ワードであるので、出力サンプリング周期は 1 になります。これにより、システム全体の有効サンプリング周期は 1 になります。



Shared Memory Write および Shared Memory Read ブロックは整数値を処理するので、Simulink の型変換ブロックを追加して、モデルのさまざまな部分でデータが正しく処理されるようにする必要があります。in_data_conv サブシステムは、Simulink の倍精度値を、FPGA ハードウェアで適切に処理可能な 16 ビットの整数値に変換します。出力側の out_data_conv サブシステムは、32 ビットの整数値を 32 ビットの Simulink 固定小数点精度値に変換します。

デザインをシミュレーションする前に、前のセクションで作成したハードウェア協調シミュレーションブロックを追加します。

21. 次の図に示すように、ハードウェア協調シミュレーション ブロックを追加します。

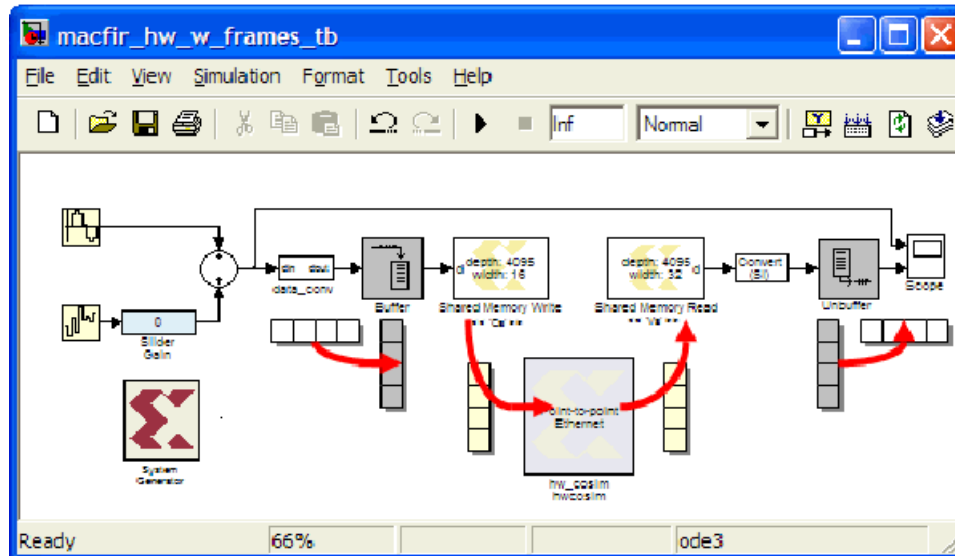


前述のとおり、Shared Memory Write ブロックは 4095 ワードの入力フレームを 4095 クロック サイクルごとに書き込み、Shared Memory Read ブロックは 4095 ワードの出力フレームを 4095 クロック サイクルごとに読み出します。そのため、FPGA でフレーム全体を 1 サイクルで処理する必要があります。

まず、FPGA をフリーランニング クロック モードにコンフィギュレーションします。このモードに設定すると、Simulink シミュレーションと同じステップで実行する場合に比べて、データがかなり高速に処理されます。シングルステップ モードでは、Simulink の 1 サイクルごとに FPGA で 1 つのデータしか処理されませんが、フリーランニング クロック モードでは、FPGA の処理速度を制限するのはシステム クロックの周波数のみです。それでも、バッファのサイズが大きいと、次のブロックが起動するまでに、FPGA でバッファのデータすべてを処理できない可能性があります。FPGA でバッファのデータ全体を処理し終えるまで、シミュレーションを待機させる方法が必要です。

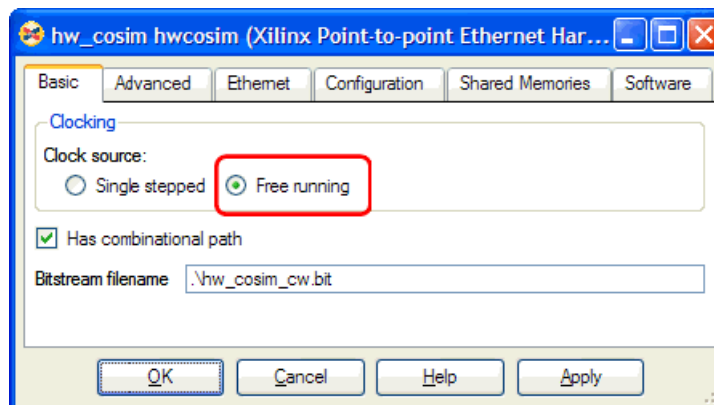
Shared Memory Read ブロックは、フレームの読み出しを実行する前に、出力バッファにある FIFO のワード数をチェックします。バッファ内のワード数が不十分な場合は、短時間待機し、ワード数が十分になったかどうかを再びチェックします。出力バッファにすべてのワードが揃うまで（この例の場合は 4095 ワード）、フレームの読み出しは実行されません。Shared Memory Read ブロックは、このようにして、FPGA でフレームが完全に処理されるまでシミュレーションを停止します。

次の図に、シミュレーションでのデータフローを示します。



Simulink フレーム信号を使用してシミュレーションを実行するには、次の手順に従います。

22. ハードウェア協調シミュレーション ブロックをダブルクリックして、パラメータ ダイアログ ボックスを開きます。
23. 次の図に示すように、[Free running] をオンにします。



24. ハードウェア協調シミュレーション ブロックのパラメータ ダイアログ ボックスで、使用する協調シミュレーション プラットフォームに必要なその他のパラメータを設定します。
25. Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、デザインのシミュレーションを開始します。

26. デザインを 10000 サイクル間シミュレーションするのにかかる時間を記録します。

27. 手順 15 で記録した時間内にシミュレーションスピードがどのくらい増加したかを記録します。

ハードウェア協調シミュレーションを使用したリアルタイム信号処理

System Generator で使用可能な共有メモリ インターフェイスでは、バンド幅が広く、大型メモリが必要な信号処理デザインを、FPGA ハードウェアを使用して協調シミュレーションできます。このインターフェイスを Shared Memory Read および Shared Memory Write ブロックと共に使用すると、ハードウェア協調シミュレーション デザインで Simulink ベクタおよびマトリックス信号を 1 シミュレーション サイクルで処理できます。Simulink と FPGA 間の大型データ トランザクションは、バースト転送を使用して実現され、協調シミュレーション インターフェイスによってはリアルタイム信号処理アプリケーションに十分なスループットが得られます。

FPGA ハードウェアにコンパイルしたときにバースト転送をサポートする System Generator インターフェイスは、2 種類あります。これらのインターフェイスには、ロック共有メモリと共有 FIFO ブロックが含まれます。これらのブロックでは、FPGA とホスト PC 間のトランザクションの方法とタイミングを定義するのに異なるハンドシェイク プロトコルが使用されます。これらのブロックを使用する前に、ハードウェア協調シミュレーションでどのように機能するかを理解しておくとは有益です。詳細は、次のセクションを参照してください。

- [ロック共有メモリの協調シミュレーション](#)
- [共有 FIFO の協調シミュレーション](#)

このマニュアルでは、System Generator モデルとしてインプリメントされた高速協調シミュレーション バッファ インターフェイスについて説明します。このインターフェイス例では、ロック共有メモリを使用して必要なバッファをインプリメントします。フロー制御ロジックで共有メモリの代わりに共有 FIFO を使用するよう変更するのは、比較的簡単です。

まず高速バッファ インターフェイスについて説明し、5x5 フィルタ カーネルを使用したビデオ ストリームのリアルタイム処理をサポートするインターフェイスの例を示します。最後に、非保護共有メモリをシステムに追加して、協調シミュレーション中にイメージ カーネルの動的な再読み込みをサポートする方法を示します。

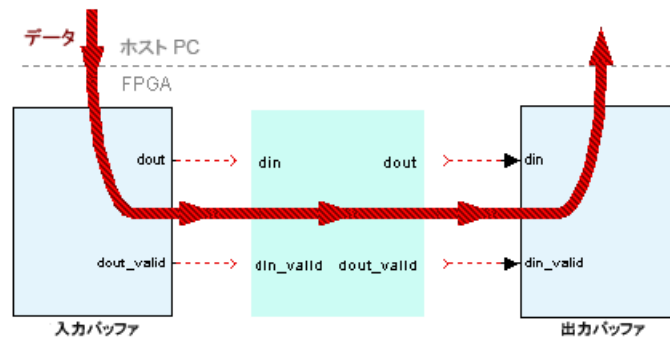
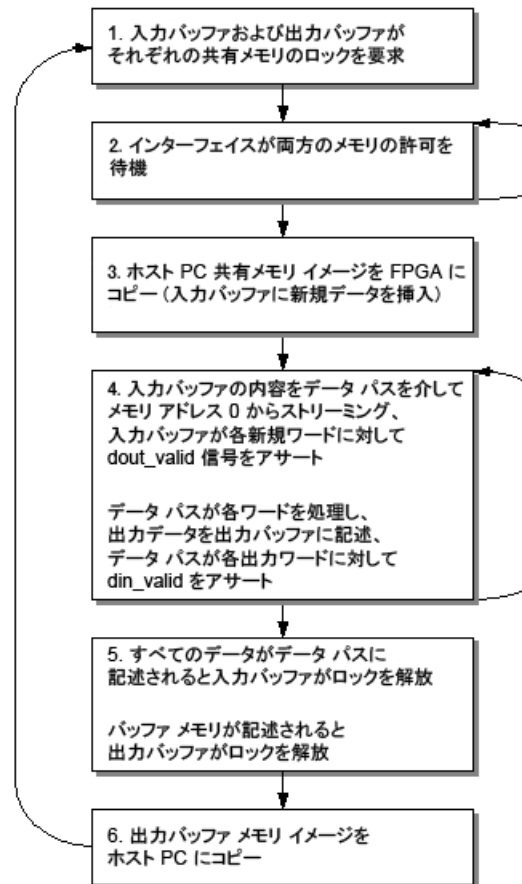
共有メモリ I/O バッファの例

ロック共有メモリをハードウェア協調シミュレーション用にコンパイルすると、排他的処理が行われるようにするため FPGA に回路が追加されます。この回路には、FPGA がメモリのロックを取得または解除したときに、メモリ イメージの高速転送を可能にするロジックが含まれています。ロック共有メモリの排他的な処理を利用して、ハードウェア協調シミュレーション用の高速 I/O バッファ インターフェイスをインプリメントします。このセクションでは、System Generator のサンプル モデルとして含まれているこのインターフェイスについて説明します。

1. MATLAB ウィンドウで、<path_to_sysgen>\examples\shared_memory\hardware_cosim\io_buffering ディレクトリに移動します。
2. highspeed_iobuf_ex.mdl を開きます。

I/O バッファ インターフェイスを使用すると、ハードウェア協調シミュレーションで System Generator 信号処理データ パスを介してデータを格納およびストリーミングできます。このデザイン例は、入力バッファをインプリメントする Input Buffer サブシステムと出力バッファをインプリメントする Output Buffer サブシステムで構成されています。中央の水色の四角は、このチュートリアルで挿入する信号処理データ パスのプレースホルダです。

各バッファ サブシステムでは、バッファとしてロック共有メモリ ブロックを使用しています。共有メモリの周辺には、ホスト PC からのデータ フローを制御し、ホスト PC に送信するロジックがあります。I/O バッファ インターフェイスの動作を次に示します。



バッファ インターフェイス デザインには、複数のデータ バリッド ポートが含まれています。これらのポートをデータ フロー制御に使用します。新しいデータがデータ パスで処理可能になると、**Input Buffer** サブシステムの **dout_valid** 出力が **High** になります。同様に、データ パスでデータ処理が終了すると、**Output Buffer** サブシステムの **din_valid** ポートが **High** になり、出力データが有効であることを示します。**din_valid** ポートは、ライト イネーブル制御信号と似ています。

デザイン例に含まれているプレースホルダは、**System Generator** のデータ パスに置き換える必要があります。バッファ インターフェイスには、前述の有効な信号の条件を満たしていれば、どのデータ パスでも挿入できます。

メモ：出力バッファ共有メモリでは、出力バッファがフルになるまでロックは解除されません。デッドロック状態になるのを回避するため、データ パスによるバリッド信号のアサートの数が、処理サイクルの出力メモリ バッファのサイズと同じになるようにします。

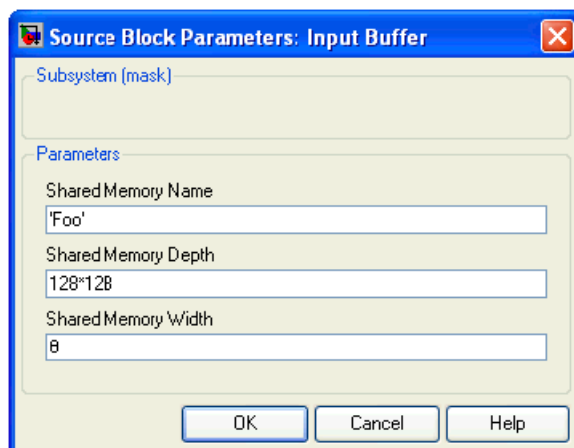
5x5 フィルタ データ パスの挿入

I/O バッファ インターフェイスにデータ パスを挿入し、**128X128** の 8 ビット グレースケール ビデオ ストリームをリアルタイムで処理できるシステムにします。高速バッファ インターフェイスのデータ パス部分のインプリメントには、**5x5** イメージ処理カーネルを使用します。フィルタ カーネルの詳細は、**sysgenConv5x5** という **System Generator** のデモを参照してください。まず、デザイン インプリメンテーションのさまざまな面を考慮します。

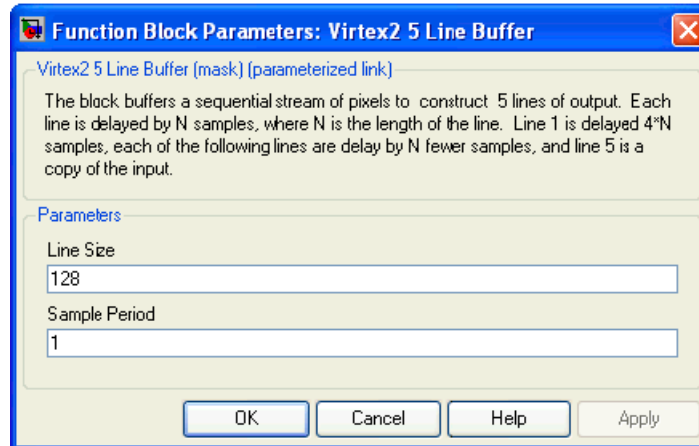
3. **MATLAB** ウィンドウで、`<path_to_sysgen>\examples\shared_memory\hardware_cosim\conv5x5_video` ディレクトリに移動します。
4. `conv5x5_video_ex.mdl` を開きます。

バッファとデータ パスのコンフィギュレーション

フレームとピクセルの制限を考慮して、**Input Buffer** と **Output Buffer** のパラメータ ダイアログボックスでは、ワード数が **128x128 (16K)**、ワード幅が 8 ビットに設定されています。このワード数の設定により、インターフェイスで 1 つのフレームを 1 つのシミュレーション サイクルで処理することが可能になります。これらのコンフィギュレーション パラメータは、バッファにインプリメントされる自動的にロック共有メモリにも自動的に適用されます。

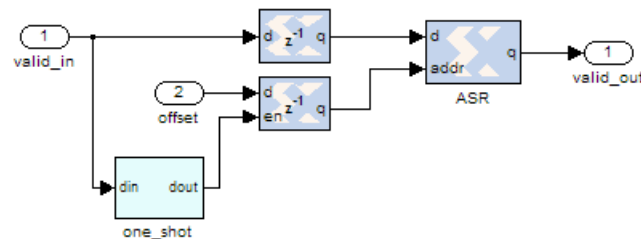


データパスではラインバッファを使用し、データサンプルがフィルタカーネルで適切に揃うようにします。これらのラインバッファのサイズは、異なるフレームサイズに対応するようパラメータ指定されています。この例では、ラインバッファは `conv5x5_video_ex` の `5x5_filter` サブシステムに含まれる **Virtex2 5 Line Buffer** にインプリメントされ、ラインサイズが 128 に設定されています。異なるサイズのフレームを処理する場合は、このブロックのパラメータダイアログボックスで [Line Size] を変更します。



バリッド ビットの生成

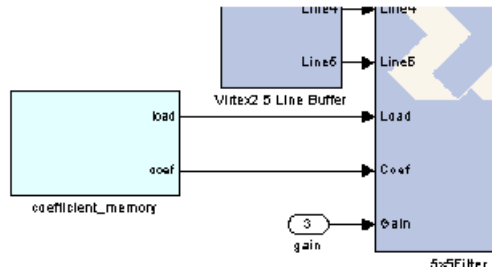
データパスには、Output Buffer ブロックの `din_valid` ポートを駆動する `valid_generator` というサブシステムが含まれます。このサブシステムには、`valid_in` と `offset` の 2 つの入力があります。`valid_in` ポートは、Input Buffer ブロックの `dout_valid` 信号で駆動され、可変なサイクル数遅延した後 `valid_out` ポートを駆動します。次の図に、`valid_generator` サブシステムに含まれるロジックを示します。



ASR (Addressable Shift Register) ブロックは、バリッド ビットを遅延させるために使用します。`offset` ポートは、ASR ブロックのアドレスを制御するために使用され、ASR ブロックでバリッドビットで発生させるレイテンシの量を制御します。Input Buffer ブロックで生成されるバリッドビットを単に遅延させることにより、Output Buffer に書き込まれるワード数が常にバッファサイズと等しくなるようにします。デザインをハードウェアで実行すると、`offset` の値の変化により、フィルタされたイメージの垂直アライメントが変更されます。

係数の再読み込み

カーネル データ パスには、ランタイムに係数を動的に再読み込みできるという特徴があります。5x5 Filter ブロックには Load と Coef 制御ポートがあり、coefficient_memory サブシステムで駆動されます。



coefficient_memory サブシステムには、最も最近読み込まれたフィルタ係数のコピーが含まれており、coef_buffer という非保護の共有メモリに保存されています。このサブシステムは、ランタイムに共有メモリの内容をモニタし、変化が検出されると再読み込みシーケンスを開始します。非保護の共有メモリで協調シミュレーションを実行すると、ホスト PC の任意のプロセスで共有メモリ coef_buffer に新しいカーネル係数を書き込むことができます。FPGA ハードウェアとの通信は共有メモリ API を介して完全に抽象化されるので、このインターフェイスは便利です。

ハードウェア協調シミュレーション用のコンパイル

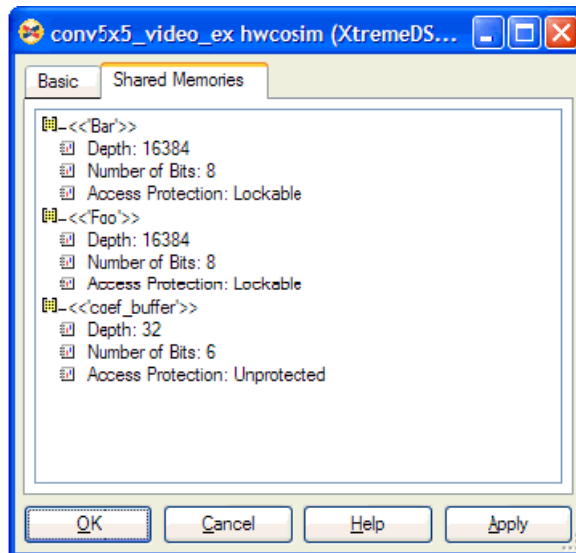
完全なフィルタ カーネル デザインをシミュレーションするには、ハードウェア協調シミュレーション用にコンパイルする必要があります。

- conv5x5_video_ex.mdl に含まれる System Generator トークンをダブルクリックします。
- [Compilation] で、適切なハードウェア協調シミュレーション ターゲットを選択します。
- [Generate] をクリックし、デザインをハードウェア協調シミュレーション用にコンパイルします。

デザインのコンパイルが完了すると、新しいハードウェア協調シミュレーション ブロックが作成されます。



ハードウェア協調シミュレーション ブロックでは、デザインの一部としてコンパイルされている共有メモリ、レジスタ、FIFO に関する情報を表示できます。この情報を表示するには、ハードウェア協調シミュレーション ブロックをダブルクリックしてパラメータ ダイアログ ボックスを開き、[Shared Memories] タブをクリックします。



ハードウェア協調シミュレーション ライブラリは開いたままにします。次のセクションで、ハードウェア協調シミュレーション ブロックをビデオ処理テストベンチ デザインに追加します。

5x5 フィルタ カーネル テストベンチ

デザイン例には、ハードウェア協調シミュレーション ブロックを使用してループ ビデオ シーケンスをフィルタする Simulink テストベンチ モデルが含まれています。

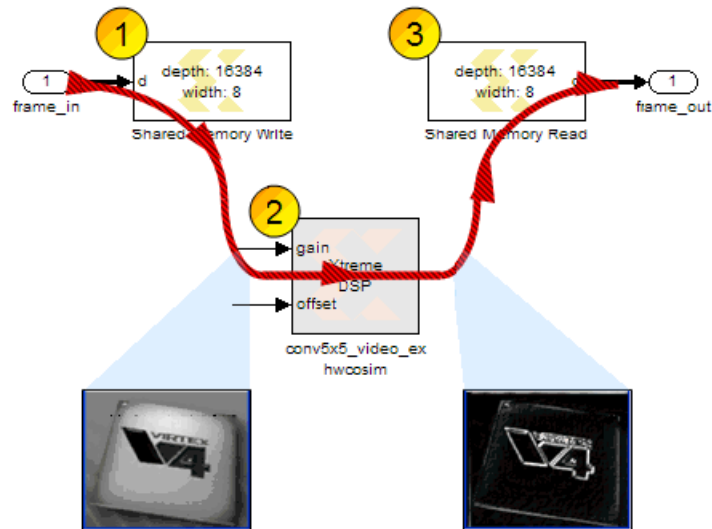
8. MATLAB ウィンドウで、<path_to_sysgen>\examples\shared_memory\hardware_cossim\conv5x5_video ディレクトリに移動します。

このテストベンチ モデルは、From Workshop ブロックを使用してループ ビデオ シーケンスを生成します。ビデオ シーケンスの各フレームは 128X128 の uint8 Simulink マトリックスで表現されます。モデルを開くと、preload 関数によりビデオ シーケンスが読み込まれ、初期化されます。ビデオ フレームは FPGA Processing サブシステムに書き込まれ、各シミュレーション サイクルごとに 1 フレーム フィルタされます。フィルタされた出力は Matrix Viewer ブロックに書き込まれ、解析されます。

FPGA Processing サブシステムには、ハードウェア協調シミュレーション ブロックのプレースホルダ、Shared Memory Read ブロック、および Shared Memory Write ブロックが含まれています。この例では、Shared Memory Read ブロックおよび Shared Memory Write ブロックにより FPGA 内の共有メモリとのビデオ フレーム I/O の送受信を制御します。これらのブロックの動作を次に説明します。

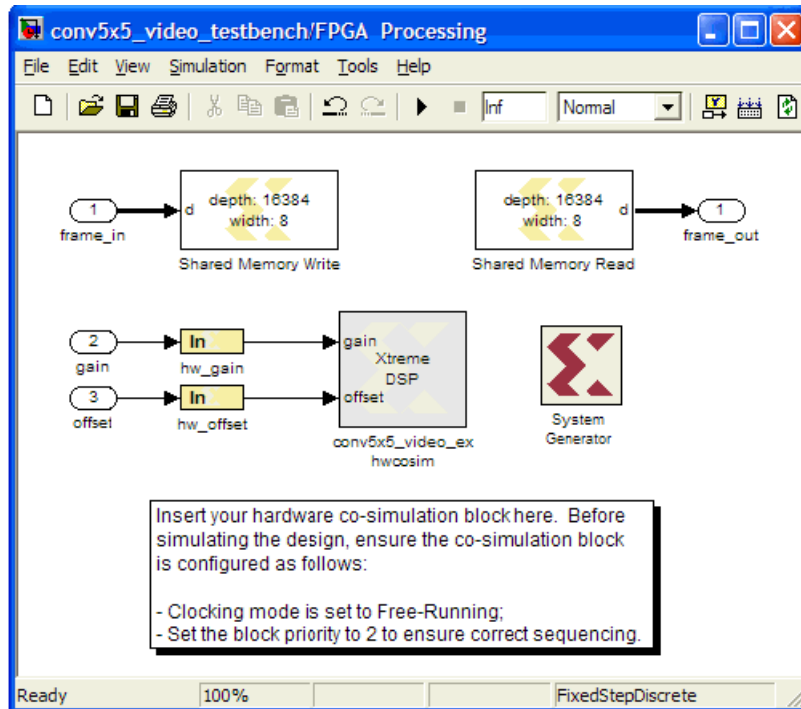
- a. Shared Memory Write ブロックが起動し、入力バッファのロック共有メモリ Foo のロックを要求します。ロックが許可されると、ビデオ フレーム データ入力をロック共有メモリに書き込み、ロックを解除します。

- b. ハードウェア協調シミュレーション ブロックが起動し、入力および出力バッファのロック共有メモリ **Foo** および **Bar** のロックを要求します。ホスト PC の共有メモリ イメージが FPGA に転送され、ロックが許可されます。FPGA が入力バッファ データを処理し、出力を出力バッファに書き込みます。FPGA での **Foo** と **Bar** のロックが解除され、FPGA の共有メモリ イメージがホスト PC に転送されます。
- c. Shared Memory Read ブロックが起動し、出力バッファのロック共有メモリ **Bar** のロックを要求します。出力バッファからビデオを読み出し、処理済のビデオ フレームを出力ポートに送信します。



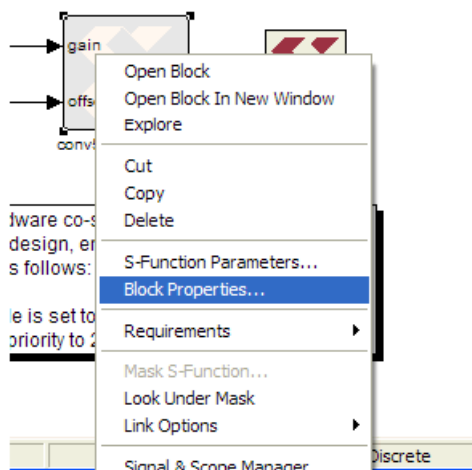
上記の手順では、ハードウェア協調シミュレーション、Shared Memory Read ブロック、Shared Memory Write ブロックが特定のシーケンスで動作することが必要です。これらのブロックが正しいシーケンスで動作するようにするには、ブロックの優先度を設定できます。シミュレーションでは、優先度の低いブロックから起動します。

9. FPGA Processing サブシステム内の水色のプレースホルダに、ハードウェア協調シミュレーションブロックを追加します。



テストベンチ内の Shared Memory Write ブロックは優先度 1、Shared Memory Read ブロックは優先度 3 でコンフィギュレーションされています。ハードウェア協調シミュレーションブロックをシミュレーション シーケンスで 2 番目に起動させる必要があるため、ハードウェア協調シミュレーションブロックの優先度は 2 に設定します。

10. ハードウェア協調シミュレーションブロックを右クリックし、[ブロックプロパティ] をクリックします。

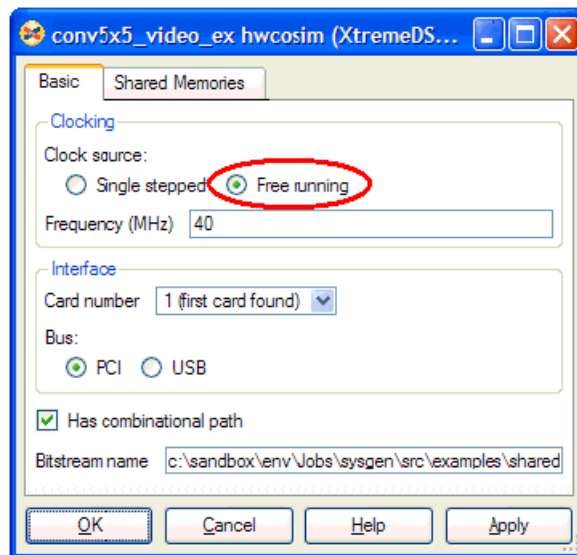


11. [モデルプロパティ] ダイアログ ボックスで、[優先度] を 2 に設定します。



高速処理アプリケーションでは、ハードウェア協調シミュレーション ブロックをフリーランニング クロック モードに設定する必要があります。このモードを使用すると、FPGA と Simulink 間の同期は、ロック共有メモリで行われます。FPGA をフリーランニング モードで実行すると、FPGA が 1 Simulink サイクルで 1 つのビデオ フレームを処理するのに十分な速度で動作します。ハードウェア協調シミュレーションブロックは、ロックを取得してからデータ処理を開始します。ハードウェア協調シミュレーションブロックが起動するまでロックは与えられないので、入力バッファに新しいデータが示されるまで FPGA はアイドル状態になります。

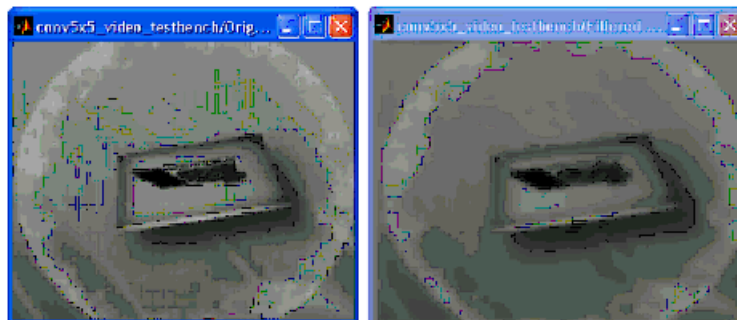
12. ハードウェア協調シミュレーション ブロックをダブルクリックして、パラメータ ダイアログ ボックスの [Basic] タブで [Free running] をオンにします。



これで、デバイスをシミュレーションする準備ができました。

13. Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、デザインのシミュレーションを開始します。

2 つのウィンドウが開き、元のビデオ ストリームとフィルタ後のビデオ ストリームが表示されます。



左側は元のビデオ フレームで、右側は同じフレームを **smooth** フィルタ カーネルを使用して処理したものです。平滑化フィルタは、ビデオ ソースに適用可能なフィルタの 1 つです。

カーネルの再読み込み

フィルタ データ パスは、ハードウェア協調シミュレーションを実行中に、フィルタ カーネルを動的に再読み込みできるように設計されています。シミュレーションの実行中は、**xlReloadFilterCoef** 関数を使用して新しいカーネルを読み込みます。この関数は、文字列カーネル識別子 (**sobelxy** など) を入力パラメータとして使用します。使用可能なフィルタ カーネルのリストは、**MATLAB** の [Command Window] に「**xlReloadFilterCoef**」と入力すると表示されます。この関数は **MATLAB** ソース ファイルとして提供されており、`<path_to_sysgen>\examples\shared_memory\hardware_cosim\conv5x5_video` ディレクトリにあります。

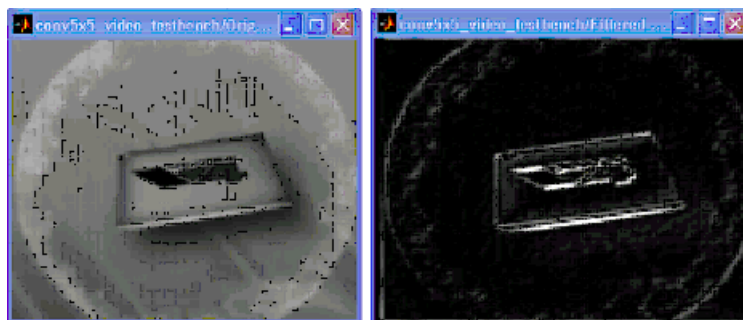
メモ：フィルタを再読み込みしたら、係数利得を調整できます。この利得は、テストベンチ モデルの最上位にある **Coefficient Adjust** ブロックを使用して調整できます。これは、**System Generator** のポート ベースのハードウェア協調シミュレーション インターフェイスを共有メモリのハードウェア協調シミュレーション インターフェイスと共に使用する方法も示しています。

System Generator では、共有メモリ オブジェクトへのアクセスに **MATLAB** オブジェクト インターフェイスを使用します。**xlReloadFilterCoef** 関数は、このオブジェクト インターフェイスを使用して、**FPGA** 上の非保護共有メモリ **coef_buffer** に新しい係数値を書き込みます。この関数には、共有メモリオブジェクトの作成方法、書き込み方法、操作が完了した際の解除方法を示す説明が付属しています。

メモ：**MATLAB** オブジェクト インターフェイスのソース コードは、`<path_to_sysgen>\examples\shared_memory\mex_function` ディレクトリにあります。このディレクトリには、**mex** 関数ソース コードの構築方法を示す **MATLAB** の **M** コードも含まれています。

14. テスト ベンチ デザインが実行されていることを確認したら、**MATLAB** の [Command Window] に「**xlReloadFilterCoef('sobelxy')**」と入力し、**SobelXY** フィルタ カーネルを読み込みます。

SobelXY カーネルを使用したビデオ出力が表示されます。



ハードウェア協調シミュレーション ボードのインストール

メモ：ご使用のプラットフォームのインストール手順がない場合は、プラットフォーム キットに付属のインストール ガイドを参照してください。

イーサネット ハードウェア協調シミュレーション用の ML402 プラットフォームのインストール

次に、ML402 ボードでイーサネット ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

1. ザイリンクス Virtex®-4 SX ML402 プラットフォーム。次のものが含まれています。
 - a. Virtex-4 ML402 プラットフォーム
 - b. ML402 キットに含まれる 5V 電源
 - c. コンパクト フラッシュ カード
2. 次のものも必要です。
 - a. ホスト PC 用のイーサネット ネットワーク インターフェイス カード (NIC)
 - b. イーサネット RJ45 オス/オス ケーブル (ネットワーク ケーブルまたはクロスオーバー ケーブルを使用可)
 - c. PC 用のコンパクト フラッシュ リーダ

ホスト PC へのソフトウェアのインストール

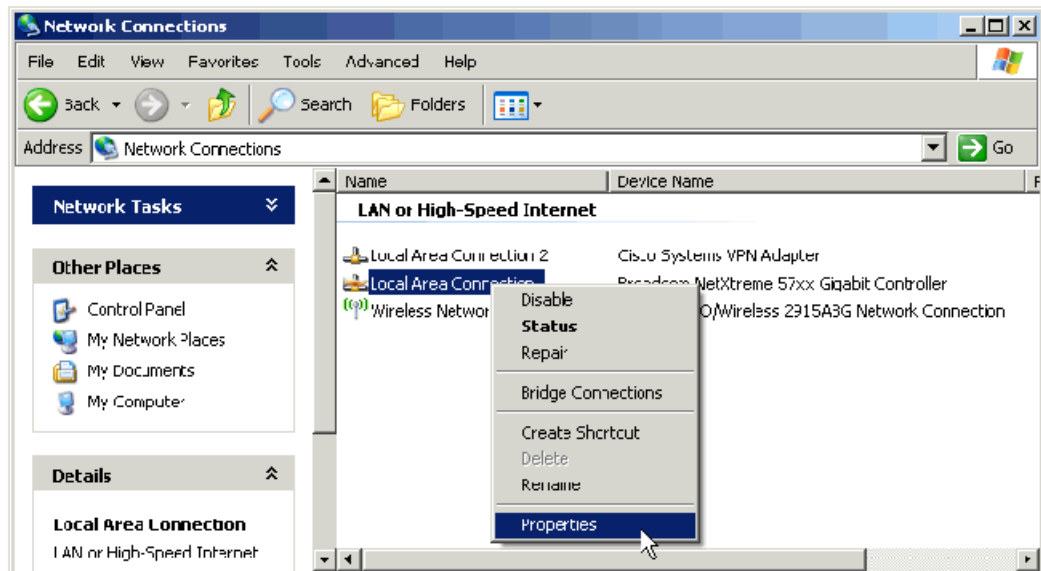
次のソフトウェアが PC にインストールされていることを確認してください。

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン
- WinPcap バージョン 4.0 (System Generator のインストーラでインストールするか、<http://www.winpcap.org> から入手)

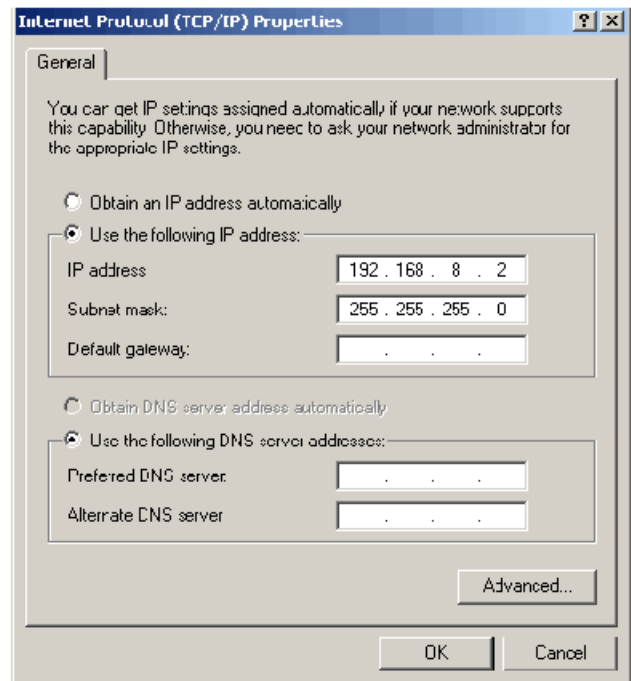
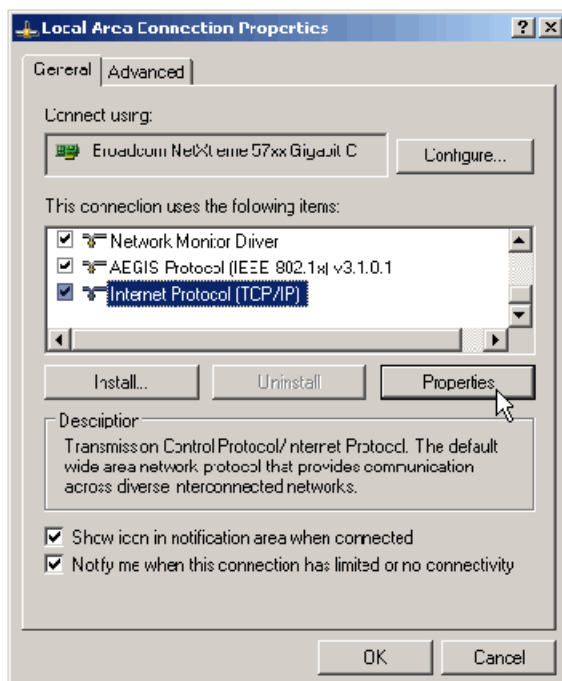
PC 上の LAN (ローカル エリア ネットワーク) の設定

PC 上に、10/100 高速イーサネットまたはギガビット イーサネット アダプタが必要です。次の手順に従って設定します。

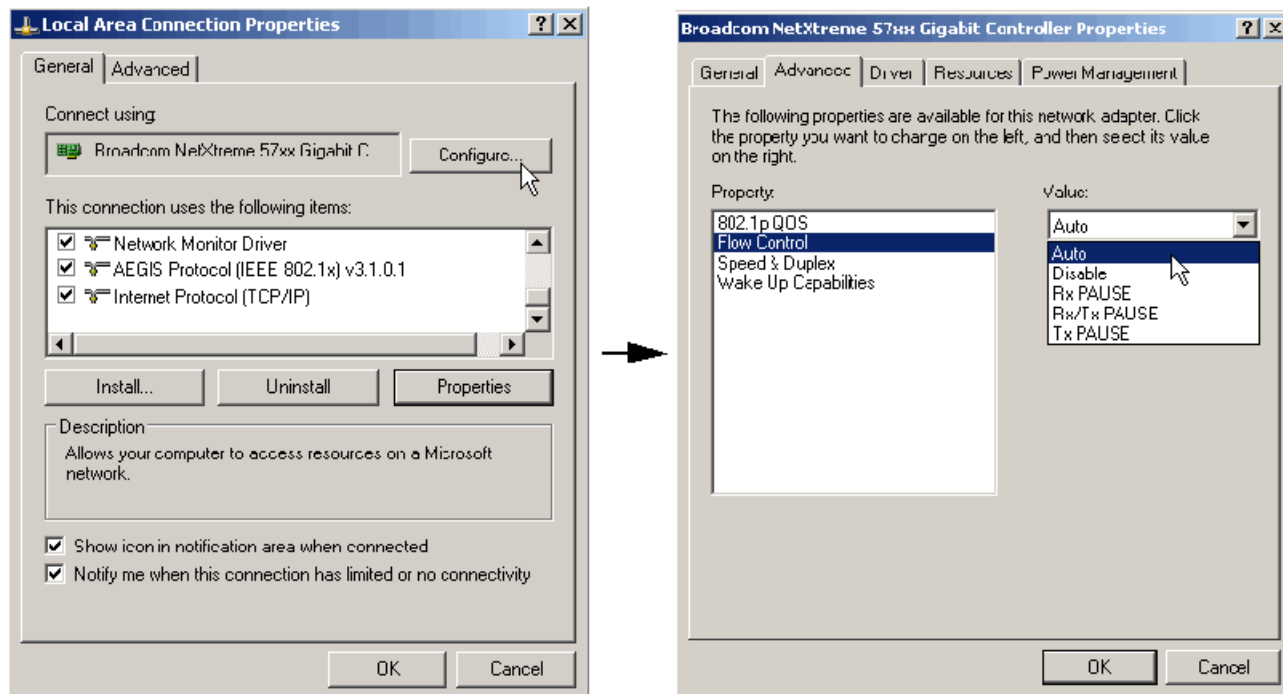
1. [スタート] → [コントロール パネル] をクリックして [ネットワーク接続] をダブルクリックし、[ローカル エリア接続] を右クリックして [プロパティ] をクリックします。



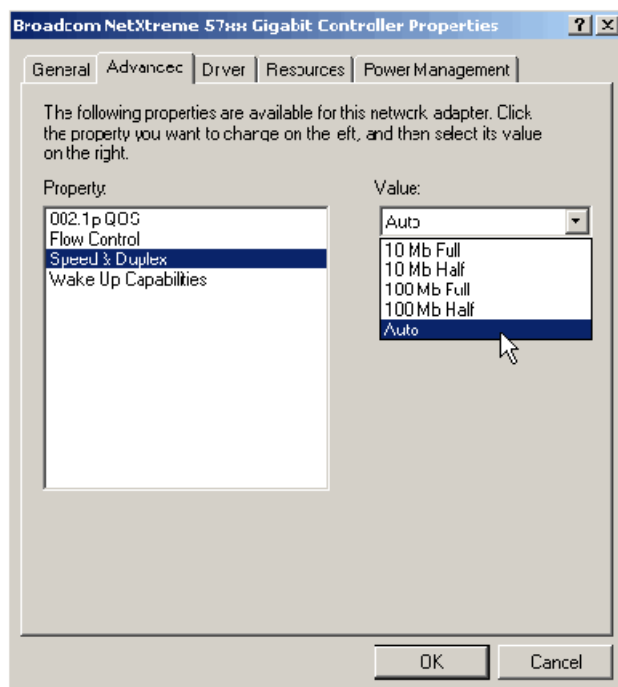
2. [インターネット プロトコル (TCP/IP)] を選択し、[プロパティ] ボタンをクリックします。[インターネット プロトコル (TCP/IP) のプロパティ] ダイアログ ボックスで次の IP アドレスを使う] をオンにし、[IP アドレス] を 192.168.8.2 に、[サブネット マスク] を 255.255.255.0 に設定します。ML402 のデフォルトの IP アドレスは 192.168.8.1 であるので、IP アドレスの最後の桁は 1 以外にする必要があります。詳細は、「[System Generator ML402 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み](#)」を参照してください。



3. [構成] ボタンをクリックし、[詳細設定] タブをクリックして [Flow Control] を [Auto] に設定します。



4. [Speed & Duplex] を [Auto] に設定し、[OK] をクリックします。



System Generator ML402 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み

System Generator ではハードウェア協調シミュレーション コンフィギュレーション ファイルが提供されており、コンパクト フラッシュ リーダで ML402 コンパクト フラッシュ カードに読み込む必要があります。

1. ML402 デモ ファイルのバックアップを作成します (オプション)。

ML402 コンパクト フラッシュ カードには、複数のデモ ファイルが含まれており、読み込んで使用できます。

- a. コンパクト フラッシュ リーダを PC に接続します。これには、通常 USB ポートを使用します。
- b. コンパクト フラッシュ カードをコンパクト フラッシュ リーダに挿入します。
- c. [マイ コンピュータ] をダブルクリックし、コンパクト フラッシュ リーダを表す [リムーバブル ディスク] を選択します。
- d. PC でバックアップ フォルダを作成または開き、コンパクト フラッシュ カードの内容をコピーします。

メモ：この後の手順では、E: をコンパクト フラッシュ リーダのドライブ名とします。

2. コンパクト フラッシュ カードを再フォーマットします。

System Generator ファイルを転送できるようにするには、カードを FAT16 ファイル システムに再フォーマットする必要があります。カードをフォーマットするには、mkdosfs ユーティリティを使用します。

- a. 次のザイリンクス Web サイトから mkdosfs ユーティリティをダウンロードします。
<http://japan.xilinx.com/products/boards/ml310/current/utilities/mkdosfs.zip>
- b. ダウンロードした ZIP ファイルを C:\mkdosfs に解凍します。
- c. [スタート] → [ファイル名を指定して実行] をクリックし、「cmd」と入力して [OK] をクリックします。
- d. Windows コマンド プロンプトで、次のように入力して mkdosfs フォルダに移動します。

```
cd C:\mkdosfs
```

注意：次のステップで、ドライブ名がコンパクト フラッシュのリムーバブル ディスク (この例では E:) に指定されていることを確認してください。ドライブ名が正しく指定されていないと、間違って指定したドライブが消去され、再フォーマットされます。

- e. 次の mkdosfs コマンドを入力します。

```
mkdosfs -v -F 16 e:
```

コンパクト フラッシュ カードの内容が消去され、再フォーマットされます。

3. System Generator コンフィギュレーション ファイルをコンパクト フラッシュ カードにコピーします。

メモ：System Generator コンフィギュレーション ファイルは、次の場所にあります。

```
...<path_to_sysgen>\plugins\bin\ML402_sysace_cf.zip
```

MATLAB を起動し、[Command Window] に次のコマンドを入力します。

```
unzip(fullfile(xlFindSysgenRoot,'plugins/bin/ML402_sysace_cf.zip'),'e:/')
```

コンパクト フラッシュのドライブに、次のファイルとフォルダが表示されます。



イーサネット MAC アドレスと IPv4 アドレスの設定 (オプション)

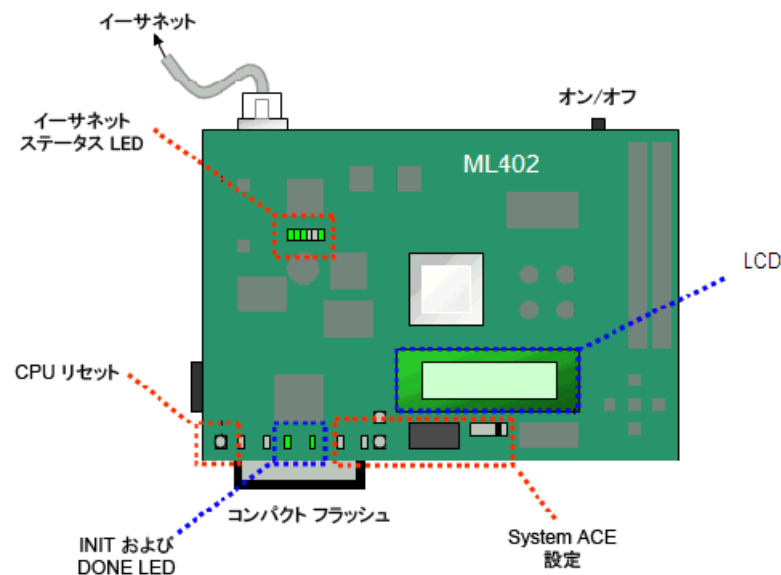
メモ：デフォルトの MAC アドレスと IP アドレスがデフォルトのネットワーク設定と競合する場合、または複数の ML402 ボードを同時に協調シミュレーションする場合は、次の手順が必要です。競合していない場合は、次のセクションに進んでください。

データをカードに書き込むと、カードのルート ディレクトリに **mac.dat** および **ip.dat** ファイルがあります。これらのファイルは、プラットフォームに関連付けるイーサネット **MAC** アドレスと **IPv4** アドレスを指定します。これらのアドレスは、イーサネット ハードウェア協調シミュレーション中にターゲット プラットフォームを識別するために使用されます。

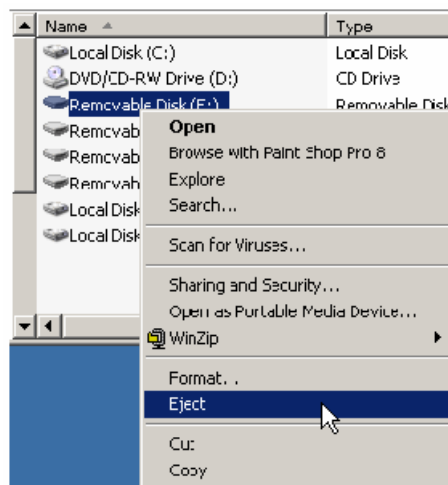
- a. **mac.dat** ファイルをテキスト エディタで開き、イーサネット **MAC** アドレスを変更します。**MAC** アドレスは、2 桁の 16 進数 6 個をコロンで区切って指定します (00:0a:35:11:22:33 など)。すべて 0、ブロードキャスト、マルチキャスト **MAC** アドレスはサポートされていません。
- b. **ip.dat** ファイルをテキスト エディタで開き、**IP** アドレスを変更します。**IP** アドレスは、10 進数をピリオドで区切った表記方法 (192.168.8.1 など) で指定します。すべて 0、ブロードキャスト、マルチキャスト、ループバック **IP** アドレスはサポートされていません。**ML402** プラットフォームの **IP** アドレスを変更したら、「[PC 上の LAN \(ローカル エリア ネットワーク\) の設定](#)」の手順に従って **PC** のネットワーク接続の **IP** アドレスも変更します。直接接続するには、**ML402** と **PC** を同じサブネットにする必要があります。同じサブネットにしない場合は、**ML402** の **IP** アドレスに **PC** から、**PC** の **IP** アドレスに **ML402** からアクセスできるようにしてください。

ML402 プラットフォームの設定

次の図に、この設定手順に関連する ML402 コンポーネントを示します。



1. ML402 プラットフォームを、Virtex-4 とザイリンクスのロゴがボードの上部になるように配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。
3. 次の図に示すように、コンパクト フラッシュ リーダのリムーバブル ディスクを右クリックして [取り出し] をクリックします。



4. コンパクト フラッシュ カードをコンパクト フラッシュ リーダから取り出します。
5. コンパクト フラッシュ カードのスロット (ML402 プラットフォームの裏側) に、コンパクト フラッシュ カードをラベルがプラットフォームの反対側になるように挿入します。次の図に、プラットフォームの裏側にコンパクト フラッシュ カードを正しく挿入した様子を示します。

メモ：プラットフォームに含まれるコンパクト フラッシュ カードが、写真のものと異なる場合があります。

注意：コンパクト フラッシュ カードは無理に挿入したり取り出したりせず、取り扱いに注意してください。

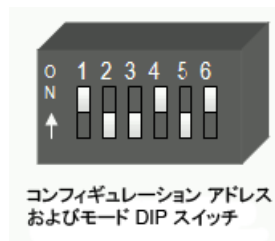


6. AC 電源コードを電源装置に接続し、電源アダプタ ケーブルを ML402 ボードに接続し、電源を AC 電源に接続します。

注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。

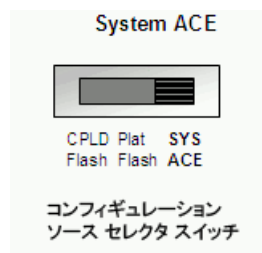
7. RJ45 オス/オス イーサネット ケーブルを使用して、ML402 プラットフォーム上のイーサネット コネクタをホスト PC 上のイーサネット コネクタに接続します。
8. コンフィギュレーション アドレス DIP スイッチを設定します。

次の図に示すように、1 をオン、2 をオフ、3 をオフ、4 をオン、5 をオフ、6 をオンにします。



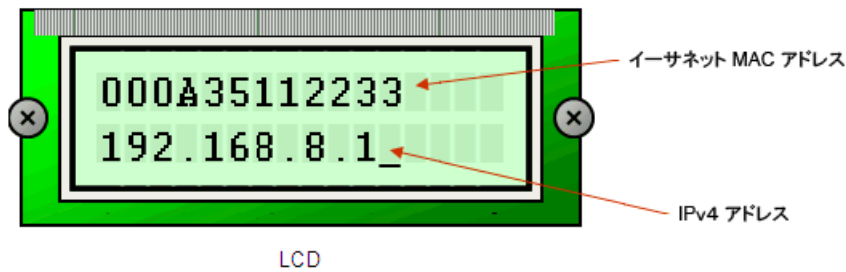
9. コンフィギュレーション ソース セレクタ スイッチを設定します。

次の図に示すように、SYS ACE に設定します。



10. コンフィギュレーション設定を確認します。

- a. プラットフォームの電源をオンにします。
- b. プラットフォーム上のステータス LED で、FPGA がコンフィギュレーションされていることを確認します。コンフィギュレーションが正常に完了した場合は、DONE LED がオン、エラー LED がオフになっているはずです。
- c. ボード上の 16 文字、2 行の LCD に表示されている情報を確認します (次の図を参照)。エラーが発生していない場合は、1 行目にイーサネット MAC アドレス (コロンなし)、2 行目に IPv4 アドレスが表示されます。



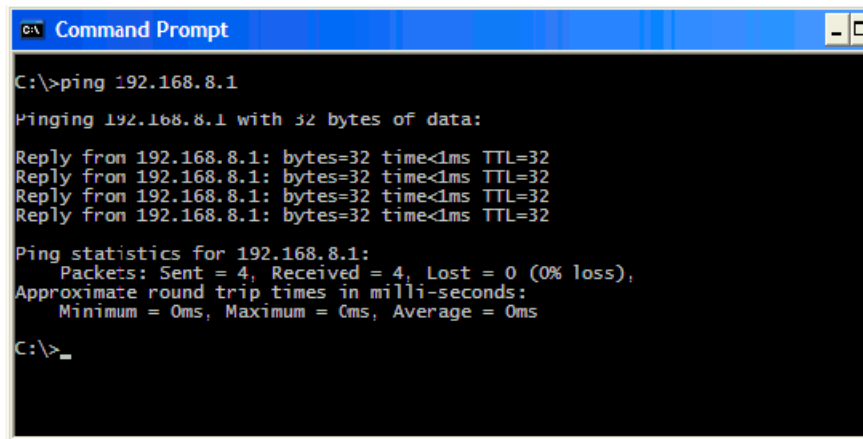
- d. LCD にこれらの情報が表示されない場合は、System ACE リセット ボタンを押して、FPGA をリコンフィギュレーションします。
 - e. ステータス LED で、コンフィギュレーション シーケンスが正しく完了したことを確認します。
11. イーサネット インターフェイスと接続ステータスを確認します。

- a. ボードのイーサネット インターフェイスをネットワークまたは直接ホストに接続します。
- b. オンボード イーサネットのステータス LED で、イーサネット インターフェイスがアクティブ イーサネット セグメントに割り当てられていることを確認します。LED は、インターフェイスが動作しているリンク スピードと二重モードを示しているはずです。ネットワークトラフィックに応じて、TX および RX LED がオン/オフになります。LED がオンにならない場合は、CPU リセット ボタンを押して FPGA をリセットし、イーサネット セグメントがアクティブであるかどうかを確認します。



イーサネット ステータス LED

- c. ホストから ICMP ping を発行して、プラットフォームがホストからアクセス可能であることを確認します。たとえば、コンソールで「ping 192.168.8.1」と入力すると、IP アドレス 192.168.8.1 でのプラットフォームへの接続をテストできます。



```
C:\> Command Prompt
C:\>ping 192.168.8.1
Pinging 192.168.8.1 with 32 bytes of data:
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32

Ping statistics for 192.168.8.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
C:\>
```

- d. ターゲット FPGA は、UDP ポート 9999 から信号を受信します。ネットワーク ベースのイーサネット コンフィギュレーションを使用する場合は、ネットワークに関連する通信が遮断されていないことを確認してください。これは、ポイントツーポイント イーサネット コンフィギュレーションには影響しません。

イーサネット ハードウェア協調シミュレーション用の ML506 プラットフォームのインストール

次に、ML506 プラットフォームでポイントツーポイント イーサネット ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

1. ザイリンクス Virtex-5 SX ML506 プラットフォーム。次のものが含まれています。
 - a. Virtex-5 ML506 プラットフォーム
 - b. ML506 キットに含まれる 5V 電源
 - c. コンパクト フラッシュ カード
2. 次のものも必要です。
 - a. ホスト PC 用のイーサネット ネットワーク インターフェイス カード (NIC)
 - b. イーサネット RJ45 オス/オス ケーブル (ネットワーク ケーブルまたはクロスオーバー ケーブルを使用可)
 - c. PC 用のコンパクト フラッシュ リーダ

ホスト PC へのソフトウェアのインストール

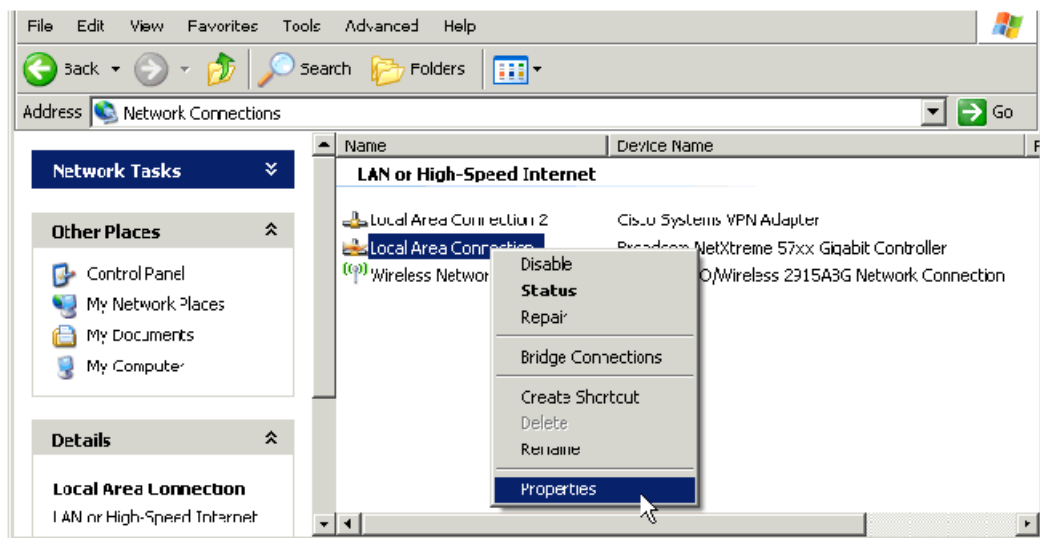
次のソフトウェアが PC にインストールされていることを確認してください。

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン
- WinPcap バージョン 4.0 (System Generator のインストーラでインストールするか、<http://www.winpcap.org> から入手)

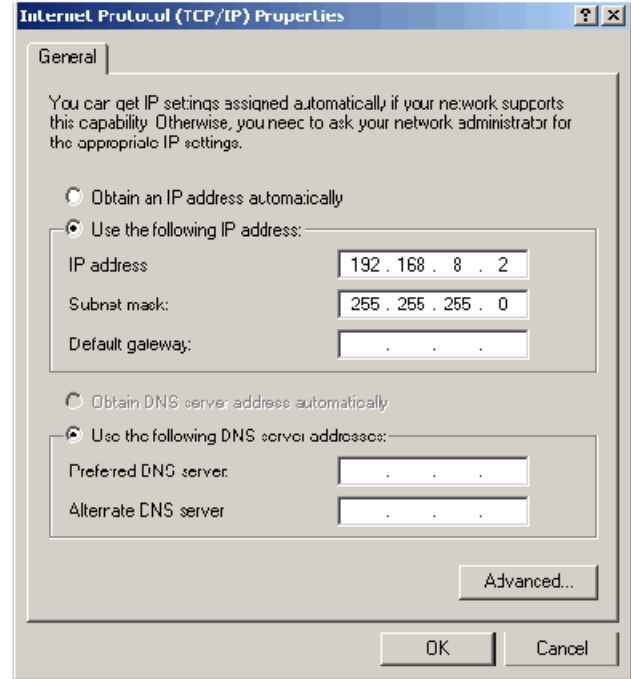
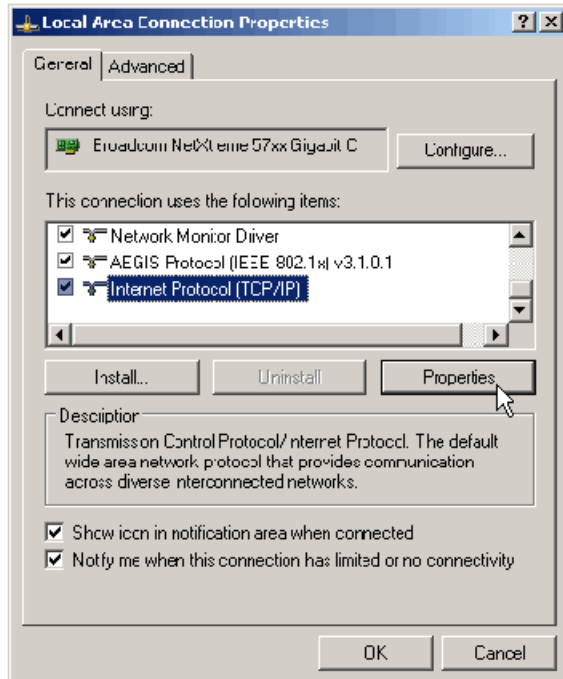
PC 上の LAN (ローカル エリア ネットワーク) の設定

PC 上に、10/100 高速イーサネットまたはギガビット イーサネット アダプタが必要です。次の手順に従って設定します。

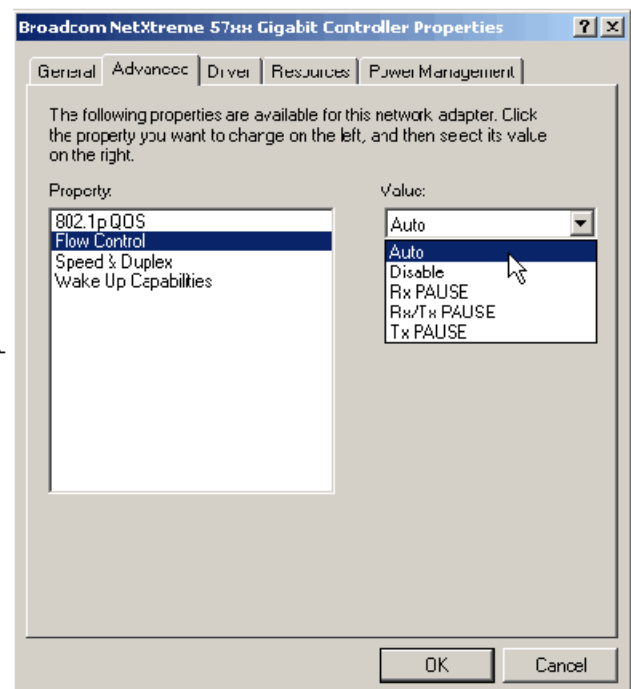
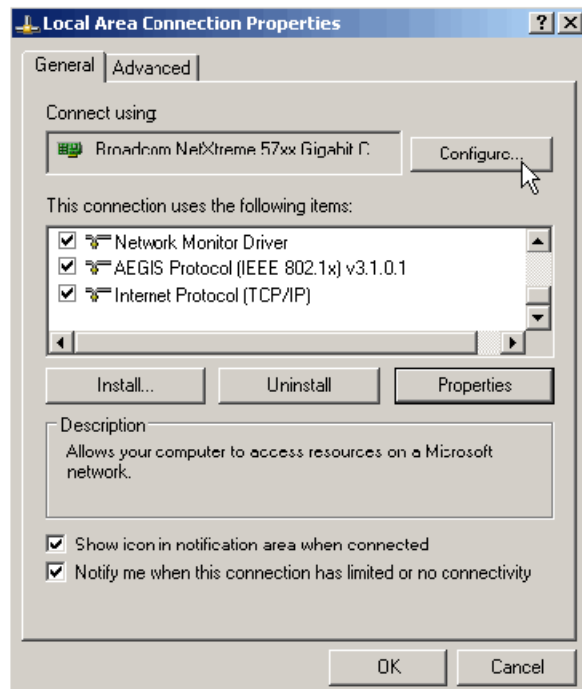
1. [スタート] → [コントロール パネル] をクリックして [ネットワーク接続] をダブルクリックし、[ローカル エリア接続] を右クリックして [プロパティ] をクリックします。



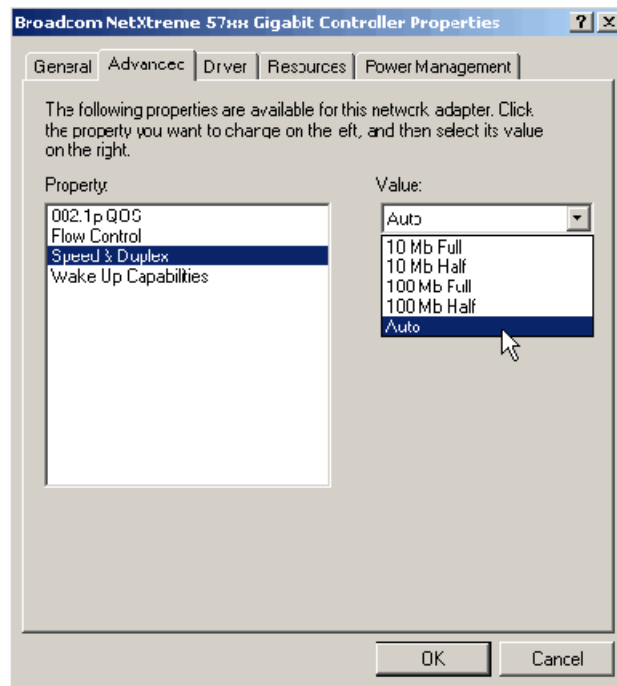
2. [インターネット プロトコル (TCP/IP)] を選択し、[プロパティ] ボタンをクリックします。
[インターネット プロトコル (TCP/IP) のプロパティ] ダイアログ ボックスで [次の IP アドレスを使う] をオンにし、[IP アドレス] を 192.168.8.2 に、[サブネット マスク] を 255.255.255.0 に設定します。ML506 のデフォルトの IP アドレスは 192.168.8.1 であるので、IP アドレスの最後の桁は 1 以外にする必要があります。詳細は、「[System Generator ML506 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み](#)」を参照してください。



3. [構成] ボタンをクリックし、[詳細設定] タブをクリックして [Flow Control] を [Auto] に設定します。



4. [Speed & Duplex] を [Auto] に設定し、[OK] をクリックします。



System Generator ML506 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み

System Generator ではハードウェア協調シミュレーション コンフィギュレーション ファイルが提供されており、コンパクト フラッシュ リーダで ML506 コンパクト フラッシュ カードに読み込む必要があります。

1. ML506 デモ ファイルのバックアップを作成します (オプション)。

ML506 コンパクト フラッシュ カードには、複数のデモ ファイルが含まれており、読み込んで使用できます。

- a. コンパクト フラッシュ リーダを PC に接続します。これには、通常 USB ポートを使用します。
- b. コンパクト フラッシュ カードをコンパクト フラッシュ リーダに挿入します。
- c. [マイ コンピュータ] をダブルクリックし、コンパクト フラッシュ リーダを表す [リムーバブル ディスク] を選択します。
- d. PC でバックアップ フォルダを作成または開き、コンパクト フラッシュ カードの内容をコピーします。

メモ：この後の手順では、E: をコンパクト フラッシュ リーダのドライブ名とします。

2. コンパクト フラッシュ カードを再フォーマットします。

System Generator ファイルを転送できるようにするには、カードを FAT16 ファイル システムに再フォーマットする必要があります。カードをフォーマットするには、mkdosfs ユーティリティを使用します。

- a. 次のザイリンクス Web サイトから mkdosfs ユーティリティをダウンロードします。

<http://japan.xilinx.com/products/boards/ml310/current/utilities/mkdosfs.zip>

- b. ダウンロードした ZIP ファイルを C:\mkdosfs に解凍します。
- c. [スタート] → [ファイル名を指定して実行] をクリックし、「cmd」と入力して [OK] をクリックします。
- d. Windows コマンド プロンプトで、次のように入力して mkdosfs フォルダに移動します。

```
cd C:\mkdosfs
```

注意：次のステップで、ドライブ名がコンパクト フラッシュのリムーバブルディスク (この例では E:) に指定されていることを確認してください。ドライブ名が正しく指定されていないと、間違って指定したドライブが消去され、再フォーマットされます。

- e. 次の mkdosfs コマンドを入力します。

```
mkdosfs -v -F 16 e:
```

コンパクト フラッシュ カードの内容が消去され、再フォーマットされます。

3. System Generator コンフィギュレーション ファイルをコンパクト フラッシュ カードにコピーします。

メモ：System Generator コンフィギュレーション ファイルは、次の場所にあります。

```
...<path_to_sysgen>\plugins\bin\ML506_sysace_cf.zip
```

MATLAB を起動し、[Command Window] に次のコマンドを入力します。

```
unzip(fullfile(xlFindSysgenRoot,'plugins/bin/ML506_sysace_cf.zip'),'e:/')
```

コンパクト フラッシュのドライブに、次のファイルとフォルダが表示されます。



イーサネット MAC アドレスと IPv4 アドレスの設定 (オプション)

メモ：デフォルトの MAC アドレスと IP アドレスがデフォルトのネットワーク設定と競合する場合、または複数の ML506 プラットフォームを同時に協調シミュレーションする場合は、次の手順が必要です。競合していない場合は、次のセクションに進んでください。

データをカードに書き込むと、カードのルート ディレクトリに mac.dat および ip.dat ファイルが配置されます。これらのファイルは、プラットフォームに関連付けるイーサネット MAC アドレスと IPv4 アドレスを指定します。これらのアドレスは、イーサネット ハードウェア協調シミュレーション中にターゲット プラットフォームを識別するために使用されます。

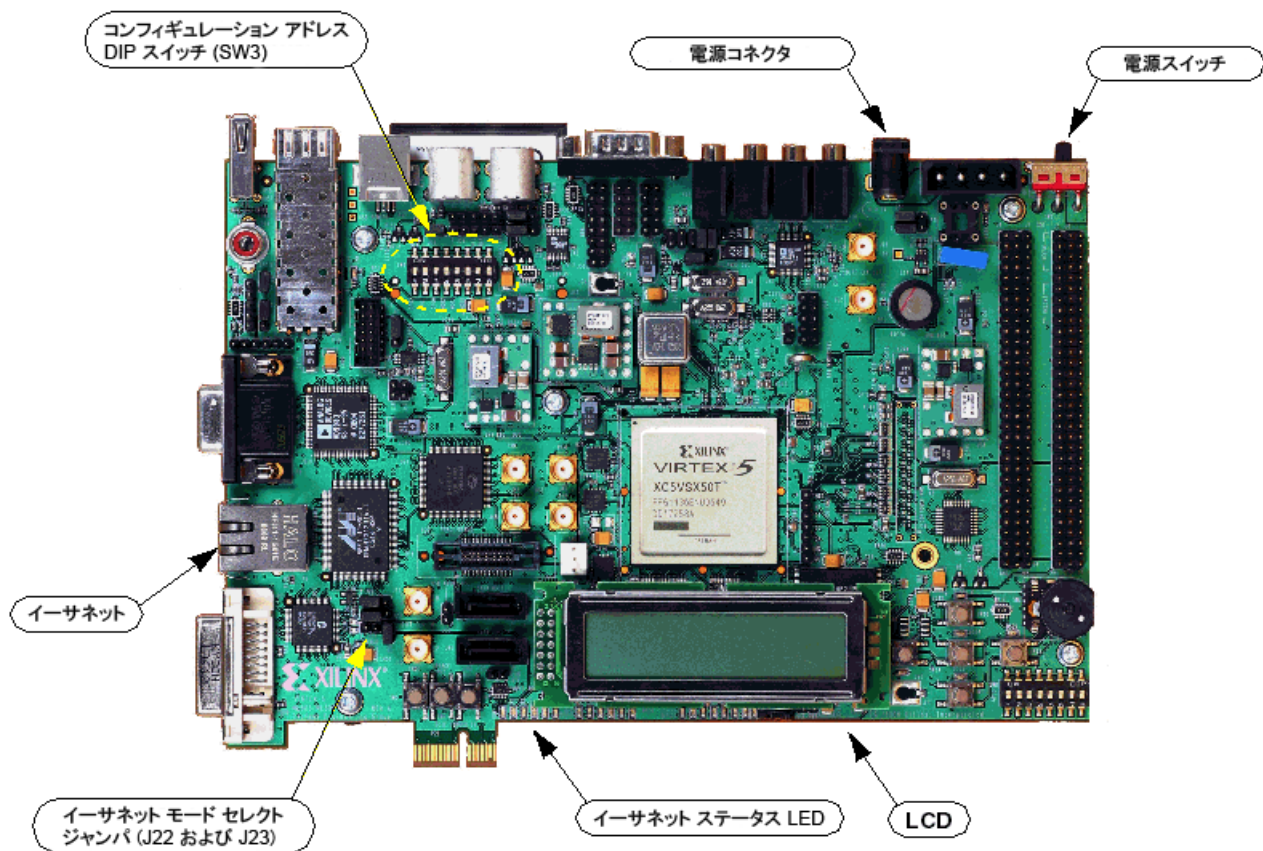
- a. mac.dat ファイルをテキスト エディタで開き、イーサネット MAC アドレスを変更します。MAC アドレスは、2 桁の 16 進数 6 個をコロンで区切って指定します (00:0a:35:11:22:33 など)。すべて 0、ブロードキャスト、マルチキャスト MAC アドレスはサポートされていません。

- b. ip.dat ファイルをテキスト エディタで開き、IP アドレスを変更します。IP アドレスは、10 進数をピリオドで区切った表記方法 (192.168.8.1 など) で指定します。すべて 0、ブロードキャスト、マルチキャスト、ループバック IP アドレスはサポートされていません。

ML506 プラットフォームの IP アドレスを変更したら、「PC 上の LAN (ローカル エリア ネットワーク) の設定」の手順に従って PC のネットワーク接続の IP アドレスも変更します。直接接続するには、ML506 と PC を同じサブネットにする必要があります。同じサブネットにしない場合は、ML506 の IP アドレスに PC から、PC の IP アドレスに ML506 からアクセスできるようにしてください。

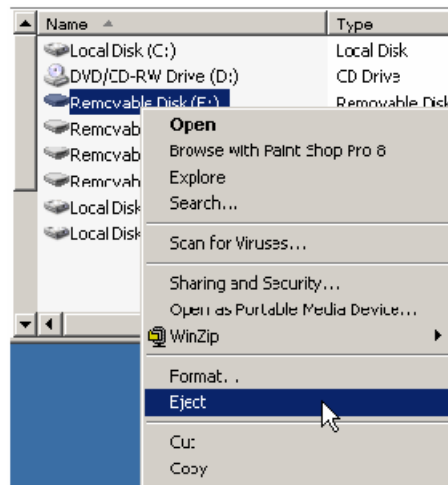
ML506 プラットフォームの設定

次の図に、この設定手順に関連する ML506 コンポーネントを示します。



1. ML506 プラットフォームを、ザイリンクスのロゴがボードの左下になるように配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。

- 次の図に示すように、コンパクト フラッシュ リーダのリムーバブル ディスクを右クリックして [取り出し] をクリックします。



- コンパクト フラッシュ カードをコンパクト フラッシュ リーダから取り出します。
- コンパクト フラッシュ カードのスロット (ML506 プラットフォームの裏側) に、コンパクト フラッシュ カードをラベルがプラットフォームの反対側になるように挿入します。次の図に、プラットフォームの裏側にコンパクト フラッシュ カードを正しく挿入した様子を示します。

メモ：プラットフォームに含まれるコンパクト フラッシュ カードが、写真のものと異なる場合があります。

注意：コンパクト フラッシュ カードは無理に挿入したり取り出したりせず、取り扱いに注意してください。

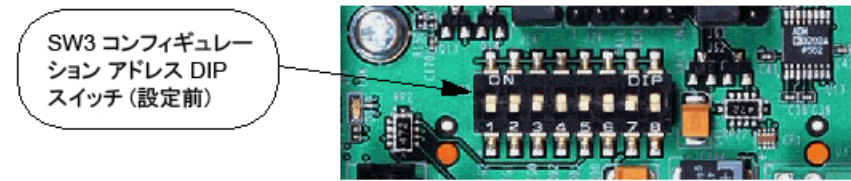


- AC 電源コードを電源装置に接続し、5V 電源アダプタ ケーブルを ML506 プラットフォームに接続し、電源を AC 電源に接続します。

注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。

- RJ45 オス/オス イーサネット ケーブルを使用して、ML506 ボード上のイーサネット コネクタをホスト PC 上のイーサネット コネクタに接続します。

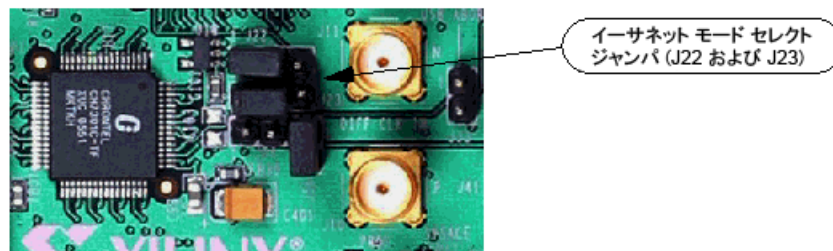
8. コンフィギュレーション アドレス DIP スイッチ SW3 を設定します。



1 をオン、2 をオフ、3 をオフ、4 をオン、5 をオフ、6 をオン、7 をオフ、8 をオンにします。

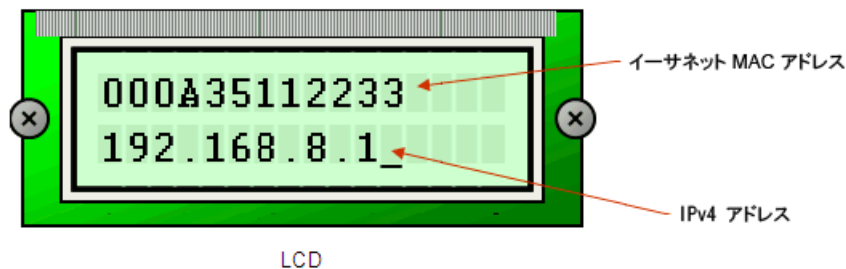
9. イーサネット モード セレクト ジャンパを設定します。

次の図に示すように、両方のイーサネット モード セレクト ジャンパ (J22 と J23) で、ピン 1 と 2 を接続します。

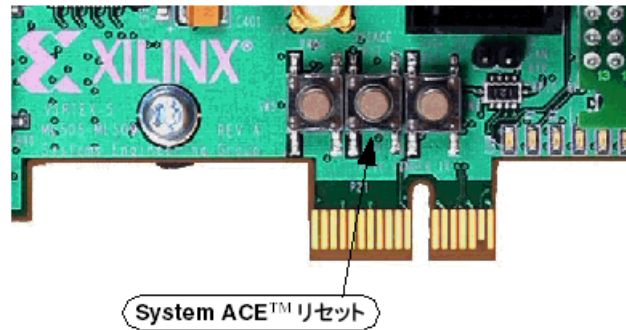


10. コンフィギュレーション設定を確認します。

- プラットフォームの電源をオンにします。
- プラットフォーム上のステータス LED で、FPGA がコンフィギュレーションされていることを確認します。コンフィギュレーションが正常に完了した場合は、DONE LED がオン、エラー LED がオフになっているはずです。
- ボード上の 16 文字、2 行の LCD に表示されている情報を確認します (次の図を参照)。エラーが発生していない場合は、1 行目にイーサネット MAC アドレス (コロンなし)、2 行目に IPv4 アドレスが表示されます。



- d. LCD にこれらの情報が表示されない場合は、System ACE リセット ボタンを押して、FPGA をリコンフィギュレーションします。



- e. ステータス LED で、コンフィギュレーション シーケンスが正しく完了したことを確認します。
11. イーサネット インターフェイスと接続ステータスを確認します。
- プラットフォームのイーサネット インターフェイスをネットワークまたは直接ホストに接続します。
 - オンボード イーサネットのステータス LED で、イーサネット インターフェイスがアクティブ イーサネット セグメントに割り当てられていることを確認します。LED は、インターフェイスが動作しているリンク スピードと二重モードを示しているはずです。ネットワークトラフィックに応じて、TX および RX LED がオン/オフになります。LED がオンにならない場合は、CPU リセット ボタンを押して FPGA をリセットし、イーサネット セグメントがアクティブであるかどうかを確認します。



イーサネット ステータス LED

- c. ホストから ICMP ping を発行して、プラットフォームがホストからアクセス可能であることを確認します。たとえば、コンソールで「ping 192.168.8.1」と入力すると、IP アドレス 192.168.8.1 でのプラットフォームへの接続をテストできます。

```

C:\> Command Prompt

C:\>ping 192.168.8.1

Pinging 192.168.8.1 with 32 bytes of data:

Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32

Ping statistics for 192.168.8.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>

```

- d. ターゲット FPGA は、UDP ポート 9999 から信号を受信します。ネットワーク ベースのイーサネット コンフィギュレーションを使用する場合は、ネットワークで関連する通信が遮断されていないことを確認してください。これは、ポイントツーポイント イーサネット コンフィギュレーションには影響しません。

イーサネット ハードウェア協調シミュレーション用の ML605 プラットフォームのインストール

次に、ML605 プラットフォームでポイントツーポイント イーサネット ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

1. ザイリンクス Virtex-6 LX ML605 プラットフォーム。次のものが含まれています。
 - a. Virtex-6 ML605 プラットフォーム
 - b. ML605 キットに含まれる 12V 電源
2. 次のものも必要です。
 - a. ホスト PC 用のイーサネット ネットワーク インターフェイス カード (NIC)
 - b. イーサネット RJ45 オス/オス ケーブル (ネットワーク ケーブルまたはクロスオーバー ケーブルを使用可)

ホスト PC へのソフトウェアのインストール

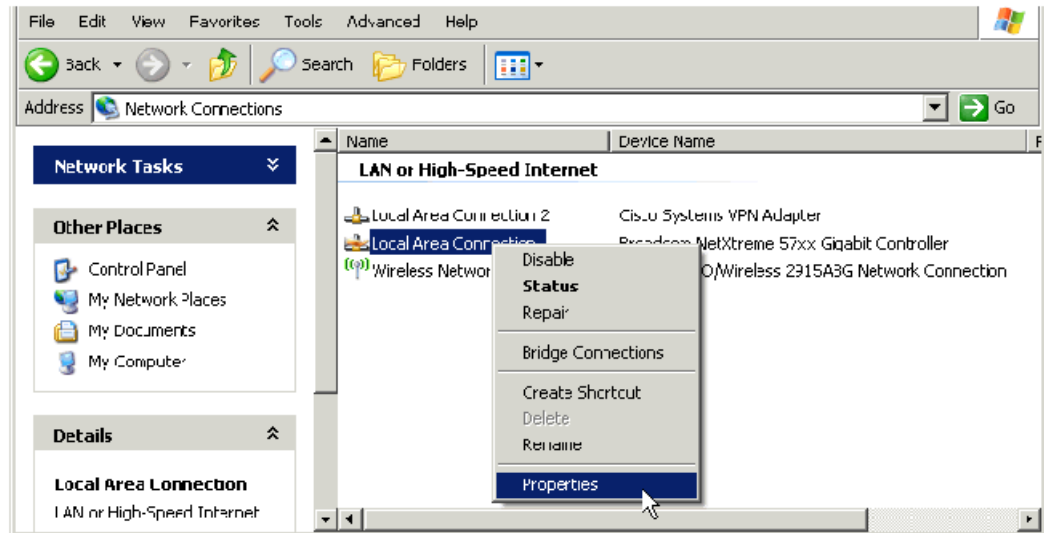
次のソフトウェアが PC にインストールされていることを確認してください。

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン
- WinPcap バージョン 4.0 (System Generator のインストーラでインストールするか、<http://www.winpcap.org> から入手)

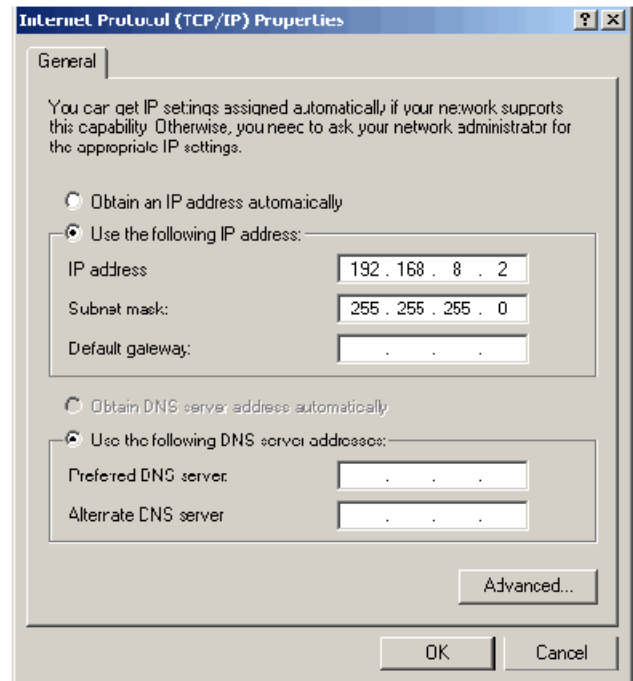
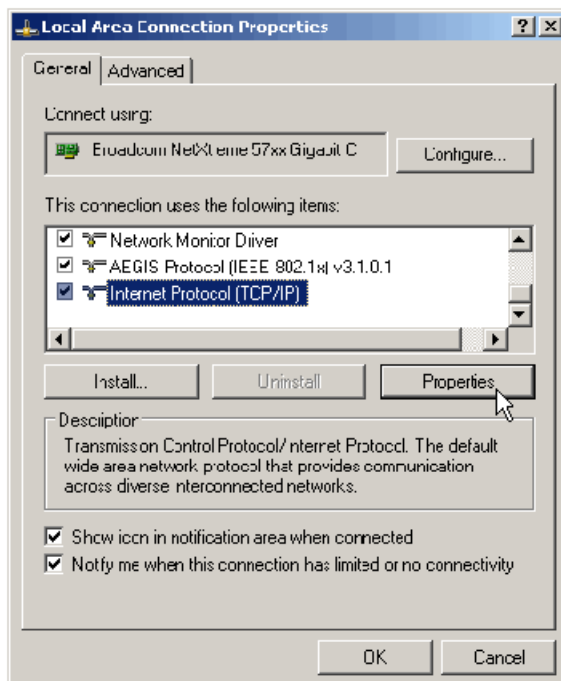
PC 上の LAN (ローカル エリア ネットワーク) の設定

PC 上に、10/100 高速イーサネットまたはギガビット イーサネット アダプタが必要です。次の手順に従って設定します。

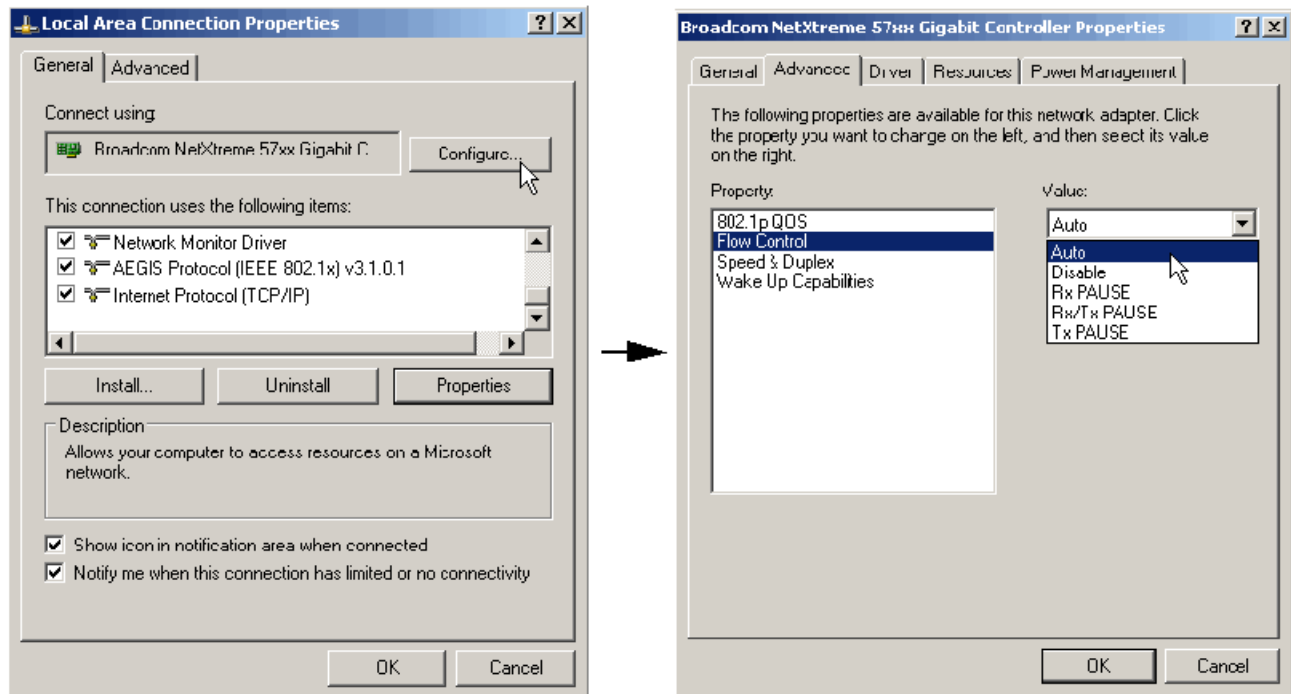
1. [スタート] → [コントロール パネル] をクリックして [ネットワーク接続] をダブルクリックし、[ローカル エリア接続] を右クリックして [プロパティ] をクリックします。



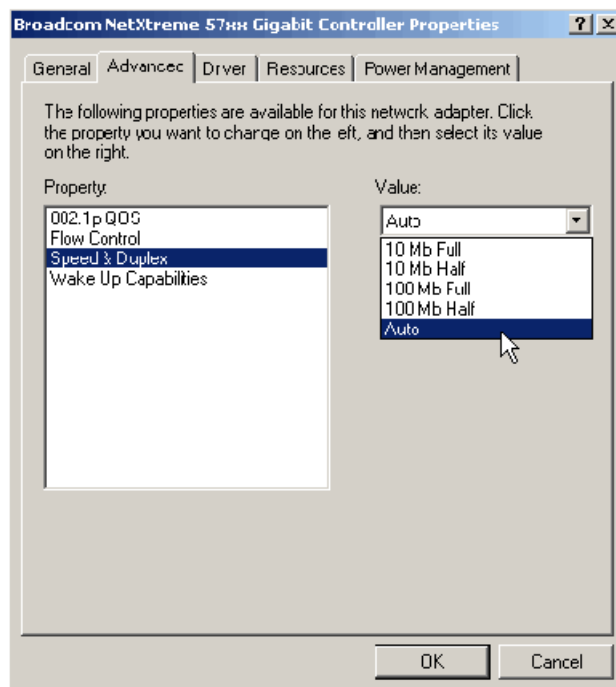
2. [インターネット プロトコル (TCP/IP)] を選択し、[プロパティ] ボタンをクリックします。
[インターネット プロトコル (TCP/IP) のプロパティ] ダイアログ ボックスで [次の IP アドレスを使う] をオンにし、[IP アドレス] を 192.168.8.2 に、[サブネット マスク] を 255.255.255.0 に設定します。ML506 のデフォルトの IP アドレスは 192.168.8.1 であるので、IP アドレスの最後の桁は 1 以外にする必要があります。詳細は、「[System Generator ML506 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み](#)」を参照してください。



3. [構成] ボタンをクリックし、[詳細設定] タブをクリックして [Flow Control] を [Auto] に設定します。

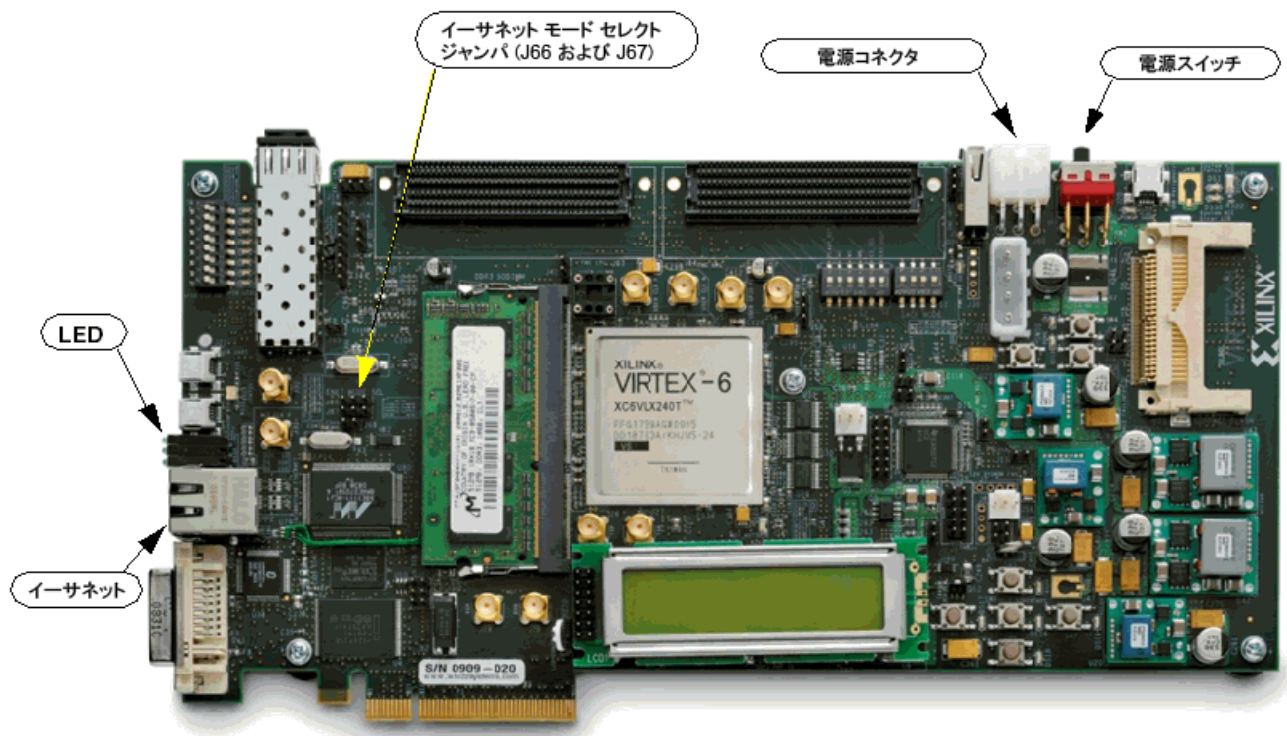


4. [Speed & Duplex] を [Auto] に設定し、[OK] をクリックします。



ML605 プラットフォームの設定

次の図に、この設定手順に関連する ML605 コンポーネントを示します。



1. ML605 プラットフォームを、ザイリンクスのロゴがボードの左下になるように配置します。
 2. 電源スイッチが右上にあり、オフになっていることを確認します。
 3. AC 電源コードを電源装置に接続し、12V 電源アダプタ ケーブルを ML605 プラットフォームに接続し、電源を AC 電源に接続します。
- 注意：**正しい電圧および電力定格の適切な電源を使用していることを確認してください。
4. RJ45 オス/オス イーサネット ケーブルを使用して、ML605 ボード上のイーサネット コネクタをホスト PC 上のイーサネット コネクタに接続します。
 5. イーサネット モード セレクト ジャンパを設定します。

両方のイーサネット モード セレクト ジャンパ (J66 と J67) で、ピン 1 と 2 を接続します。

イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 1800A スタータ プラットフォームのインストール

次に、Spartan®-3A DSP 1800A スタータ プラットフォームでハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。このプラットフォームは、コンフィギュレーションビットストリームのダウンロードに System ACE ではなく JTAG ケーブルを使用します。

必要なハードウェア

1. ザイリンクス Spartan-3A DSP 1800A スタータ プラットフォーム。次のものが含まれています。
 - a. Spartan-3A DSP 1800A スタータ プラットフォーム
 - b. 開発キットに含まれる 5V 電源
2. 次のものも必要です。
 - a. ホスト PC 用のイーサネット ネットワーク インターフェイス カード (NIC)
 - b. イーサネット RJ45 オス/オス ケーブル (ネットワーク ケーブルまたはクロスオーバー ケーブルを使用可)
 - c. ザイリンクス パラレル ケーブル IV と Power Jack スプリッタ ケーブル、またはザイリンクス プラットフォーム USB ケーブルと 14 ピン リボン ケーブル

ホスト PC へのソフトウェアのインストール

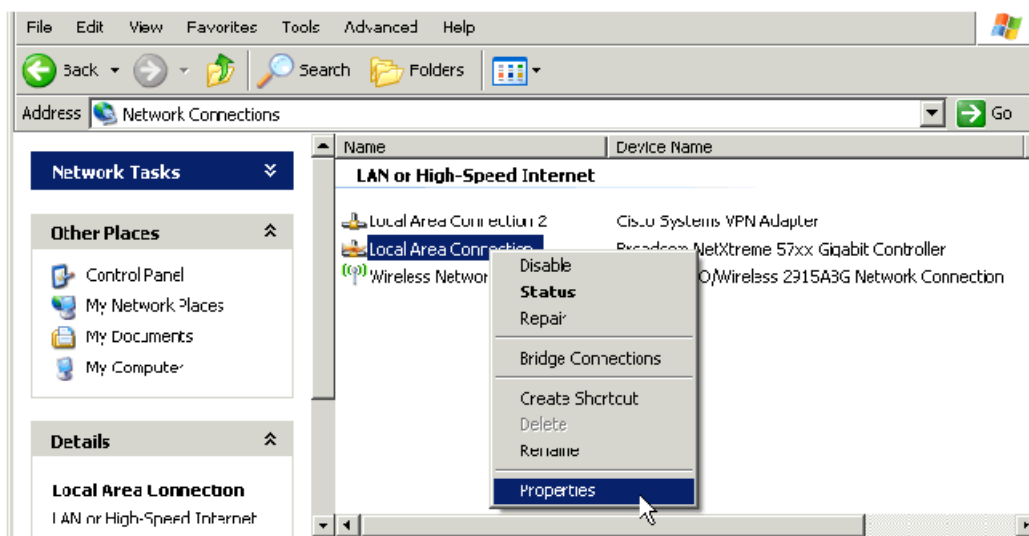
次のソフトウェアが PC にインストールされていることを確認してください。

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン
- WinPcap バージョン 4.0 (System Generator のインストーラでインストールするか、<http://www.winpcap.org> から入手)

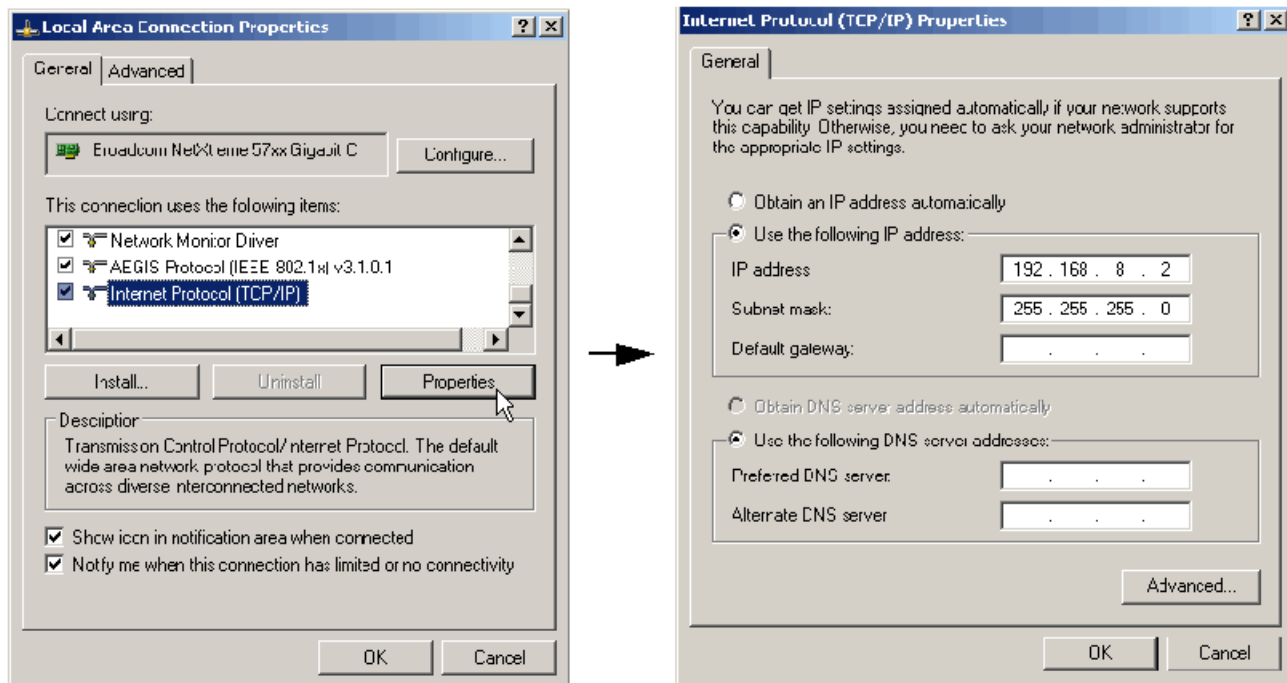
PC 上の LAN (ローカル エリア ネットワーク) の設定

PC 上に、10/100 高速イーサネットまたはギガビット イーサネット アダプタが必要です。次の手順に従って設定します。

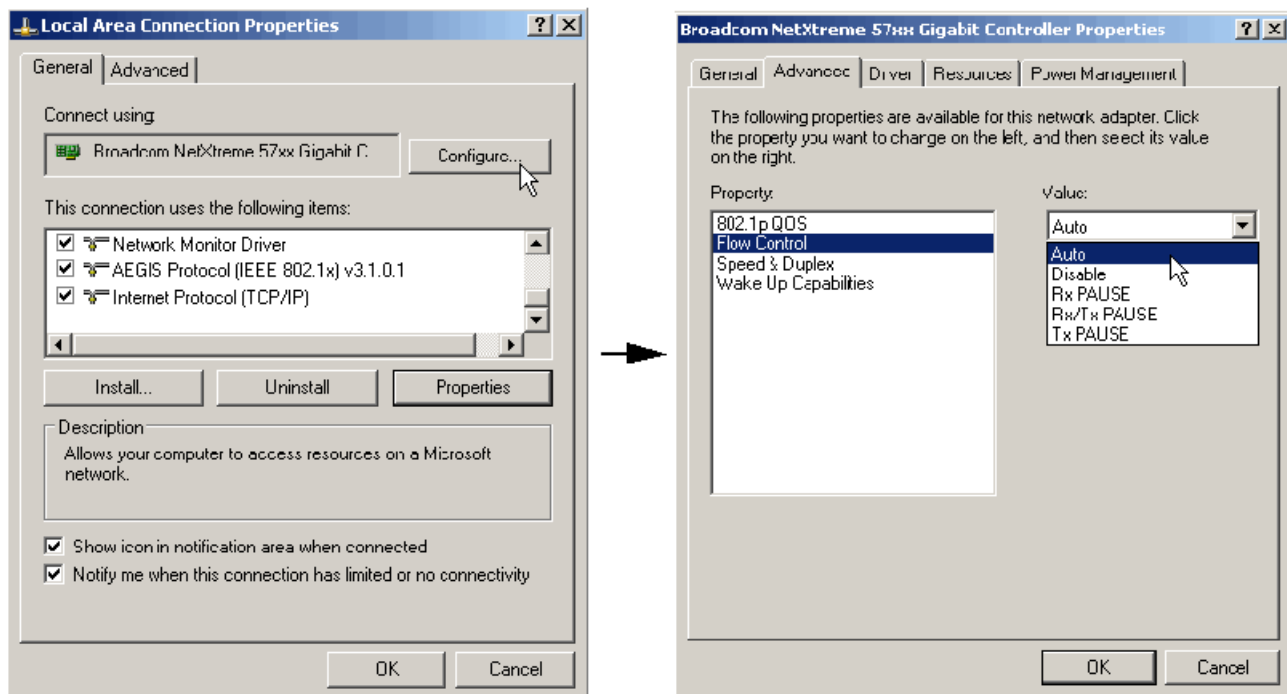
1. [スタート] → [コントロール パネル] をクリックして [ネットワーク接続] をダブルクリックし、[ローカル エリア接続] を右クリックして [プロパティ] をクリックします。



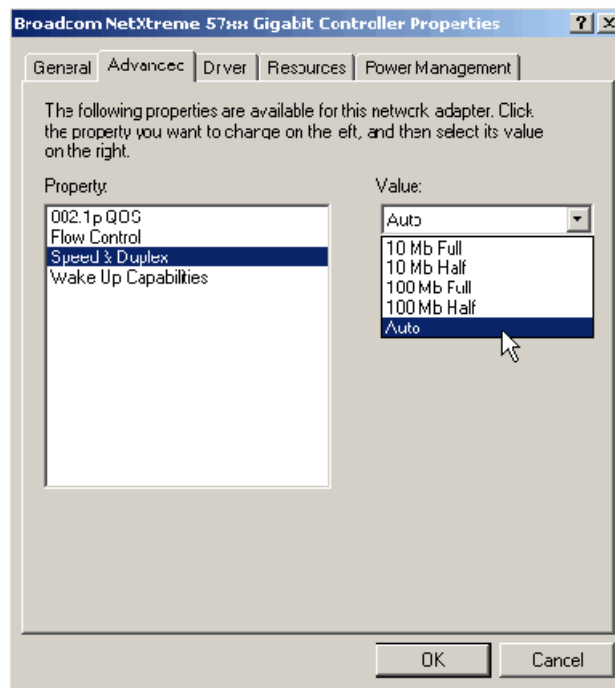
2. [インターネット プロトコル (TCP/IP)] を選択し、[プロパティ] ボタンをクリックします。[インターネット プロトコル (TCP/IP) のプロパティ] ダイアログ ボックスで次の IP アドレスを使う] をオンにし、[IP アドレス] を 192.168.8.2 に、[サブネット マスク] を 255.255.255.0 に設定します。スタータ プラットフォームのデフォルトの IP アドレスは 192.168.8.1 であるので、IP アドレスの最後の桁は 1 以外にする必要があります。



3. [構成] ボタンをクリックし、[詳細設定] タブをクリックして [Flow Control] を [Auto] に設定します。



4. [Speed & Duplex] を [Auto] に設定し、[OK] をクリックします。



Spartan-3A DSP 1800A スタータ プラットフォームの設定

1. Spartan-3A DSP 1800A スタータ プラットフォームを、ザイリンクスのロゴの右側が上になるようにし、プラットフォームの右下の 1/4 区画に配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。
3. ザイリンクス パラレル ケーブル IV を使用する場合は、次の手順 a ~ d に従います。
 - a. ザイリンクス パラレル ケーブル IV の DB25 プラグ コネクタを IEEE-1284 準拠の PC パラレル (プリンタ) ポート コネクタに接続します。
 - b. 14 ピンの 6 インチ高パフォーマンス リボン ケーブルを使用して、ザイリンクス パラレル ケーブル IV のポッド エンドをスタータ プラットフォームの JTAG ポート (J2) に接続します。
 - c. Power Jack ケーブルを PC のキーボード/マウス コネクタに接続します。
 - d. 必要に応じて、キーボード/マウス ケーブルのオス側をザイリンクス Power Jack ケーブル (スプリッタ ケーブル) のメス コネクタに接続します。
4. ザイリンクス プラットフォーム ケーブル USB を使用する場合は、次の手順 a ~ b に従います。
 - a. ザイリンクス プラットフォーム USB を PC 上の USB ポートに接続します。
 - b. 14 ピンの 6 インチ高パフォーマンス リボン ケーブルを使用して、ザイリンクス プラットフォーム ケーブル USB のポッド エンドをスタータ プラットフォームの JTAG ポート (J2) に接続します。
5. AC 電源コードを電源装置に接続し、5V 電源アダプタ ケーブルをスタータ ボードの 5V DC ONLY コネクタ (J5) に接続し、電源コードを AC 電源に接続します。
注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。
6. Spartan-3A DSP 1800A スタータ プラットフォームの電源をオンにします。

イーサネット ハードウェア協調シミュレーション用の Spartan-3A DSP 3400A 開発プラットフォームのインストール

次に、Spartan-3A DSP 3400A 開発プラットフォームでポイントツーポイント イーサネット ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

1. ザイリンクス Spartan-3A DSP 3400A 開発プラットフォーム。次のものが含まれています。
 - a. Spartan-3A DSP 3400A 開発プラットフォーム
 - b. ボード LYR178-101C (Rev C) に含まれる +12V 電源、またはボード LYR178-101D (Rev D) に含まれる +5V 電源
 - c. コンパクト フラッシュ カード
2. 次のものも必要です。
 - a. ホスト PC 用のイーサネット ネットワーク インターフェイス カード (NIC)
 - b. イーサネット RJ45 オス/オス ケーブル (ネットワーク ケーブルまたはクロスオーバー ケーブルを使用可)
 - c. PC 用のコンパクト フラッシュ リーダ

ホスト PC へのソフトウェアのインストール

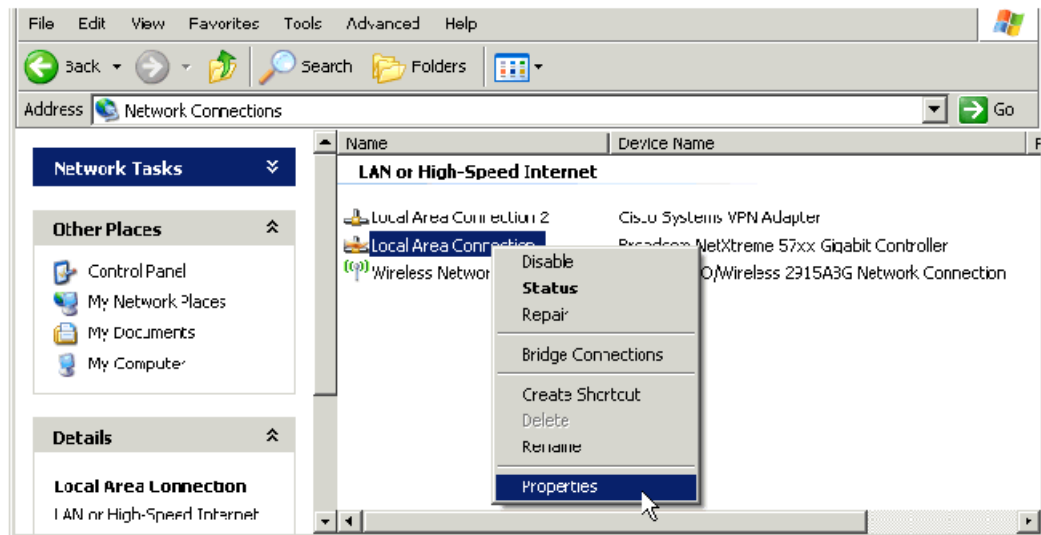
次のソフトウェアが PC にインストールされていることを確認してください。

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン
- WinPcap バージョン 4.0 (System Generator のインストーラでインストールするか、<http://www.winpcap.org> から入手)

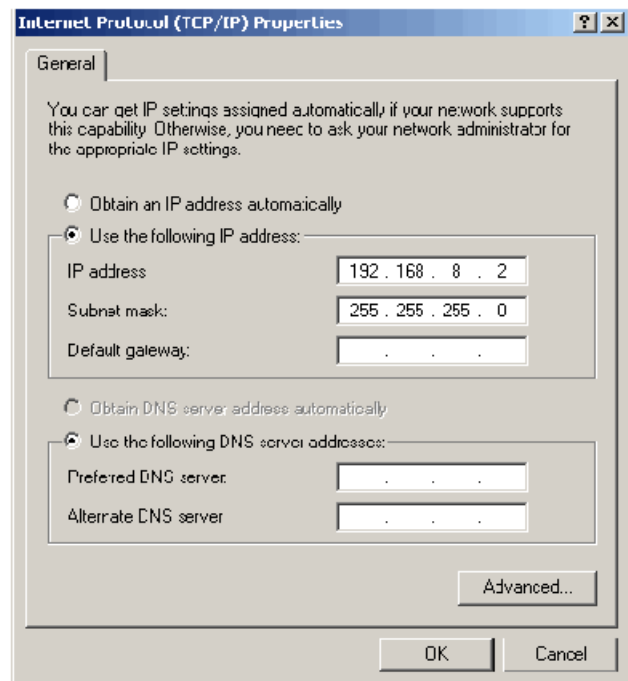
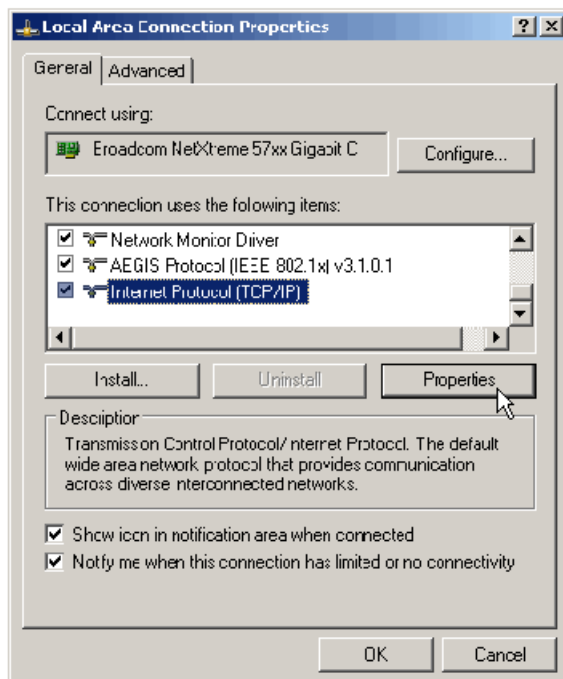
PC 上の LAN (ローカル エリア ネットワーク) の設定

PC 上に、10/100 高速イーサネットまたはギガビット イーサネット アダプタが必要です。次の手順に従って設定します。

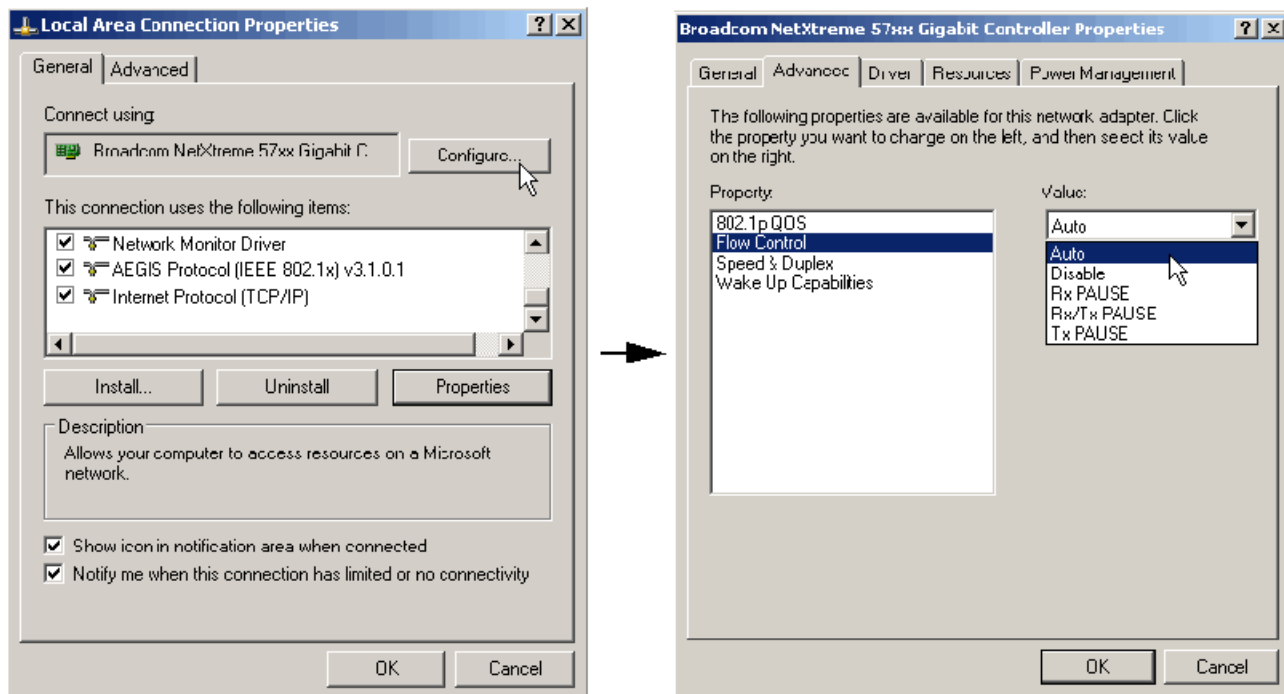
1. [スタート] → [コントロール パネル] をクリックして [ネットワーク接続] をダブルクリックし、[ローカル エリア接続] を右クリックして [プロパティ] をクリックします。



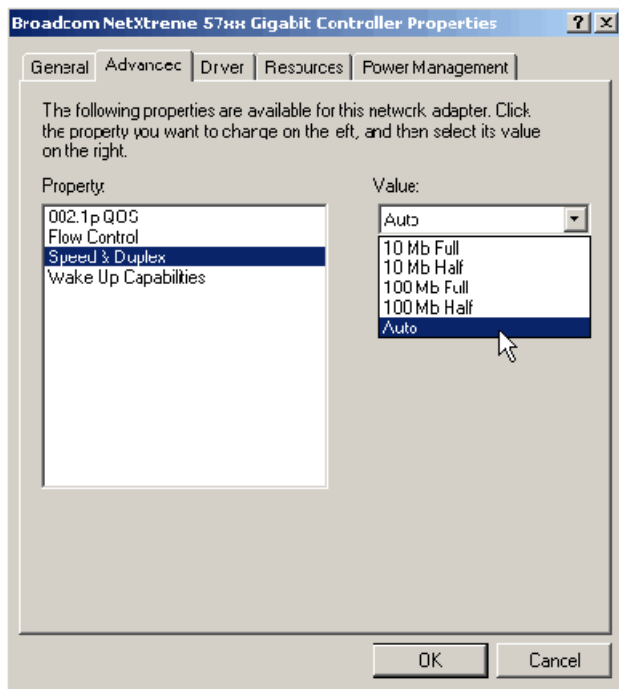
2. [インターネット プロトコル (TCP/IP)] を選択し、[プロパティ] ボタンをクリックします。[インターネット プロトコル (TCP/IP) のプロパティ] ダイアログ ボックスで [次の IP アドレスを使う] をオンにし、[IP アドレス] を 192.168.8.2 に、[サブネット マスク] を 255.255.255.0 に設定します。ML506 のデフォルトの IP アドレスは 192.168.8.1 であるので、IP アドレスの最後の桁は 1 以外にする必要があります。詳細は、「[System Generator ML506 ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み](#)」を参照してください。



3. [構成] ボタンをクリックし、[詳細設定] タブをクリックして [Flow Control] を [Auto] に設定します。



4. [Speed & Duplex] を [Auto] に設定し、[OK] をクリックします。



System Generator Spartan-3A DSP 3400A ハードウェア協調シミュレーション コンフィギュレーション ファイルの読み込み

System Generator ではハードウェア協調シミュレーション コンフィギュレーション ファイルが提供されており、コンパクト フラッシュ リーダで Spartan-3A DSP 3400A コンパクト フラッシュ カードに読み込む必要があります。

1. Spartan-3A DSP 3400A デモ ファイルのバックアップを作成します (オプション)。

Spartan-3A DSP 3400A コンパクト フラッシュ カードには、複数のデモ ファイルが含まれており、読み込んで使用できます。

- a. コンパクト フラッシュ リーダを PC に接続します。これには、通常 USB ポートを使用します。
- b. コンパクト フラッシュ カードをコンパクト フラッシュ リーダに挿入します。
- c. [マイ コンピュータ] をダブルクリックし、コンパクト フラッシュ リーダを表す [リムーバブル ディスク] を選択します。
- d. PC でバックアップ フォルダを作成または開き、コンパクト フラッシュ カードの内容をコピーします。

メモ：この後の手順では、E: をコンパクト フラッシュ リーダのドライブ名とします。

2. コンパクト フラッシュ カードを再フォーマットします。

System Generator ファイルを転送できるようにするには、カードを FAT16 ファイル システムに再フォーマットする必要があります。カードをフォーマットするには、mkdosfs ユーティリティを使用します。

- a. 次のザイリンクス Web サイトから mkdosfs ユーティリティをダウンロードします。
<http://japan.xilinx.com/products/boards/ml310/current/utilities/mkdosfs.zip>
- b. ダウンロードした ZIP ファイルを C:\mkdosfs に解凍します。
- c. [スタート] → [ファイル名を指定して実行] をクリックし、「cmd」と入力して [OK] をクリックします。
- d. Windows コマンド プロンプトで、次のように入力して mkdosfs フォルダに移動します。

```
cd C:\mkdosfs
```

注意：次のステップで、ドライブ名がコンパクト フラッシュのリムーバブル ディスク (この例では E:) に指定されていることを確認してください。ドライブ名が正しく指定されていないと、間違って指定したドライブが消去され、再フォーマットされます。

- e. 次の mkdosfs コマンドを入力します。

```
mkdosfs -v -F 16 e:
```

コンパクト フラッシュ カードの内容が消去され、再フォーマットされます。

3. System Generator コンフィギュレーション ファイルをコンパクト フラッシュ カードにコピーします。

メモ：System Generator コンフィギュレーション ファイルは、次の場所にあります。

```
...<path_to_sysgen>\plugins\bin\S3ADSP_DB_sysace_cf.zip
```

MATLAB を起動し、[Command Window] に次のコマンドを入力します。

```
unzip(fullfile(xlFindSysgenRoot, 'plugins/bin/S3ADSP_DB_sysace_cf.zip'), 'e:/')
```

イーサネット MAC アドレスと IPv4 アドレスの設定 (オプション)

メモ：デフォルトの MAC アドレスと IP アドレスがデフォルトのネットワーク設定と競合する場合、または複数の ML506 プラットフォームを同時に協調シミュレーションする場合は、次の手順が必要です。競合していない場合は、次のセクションに進んでください。

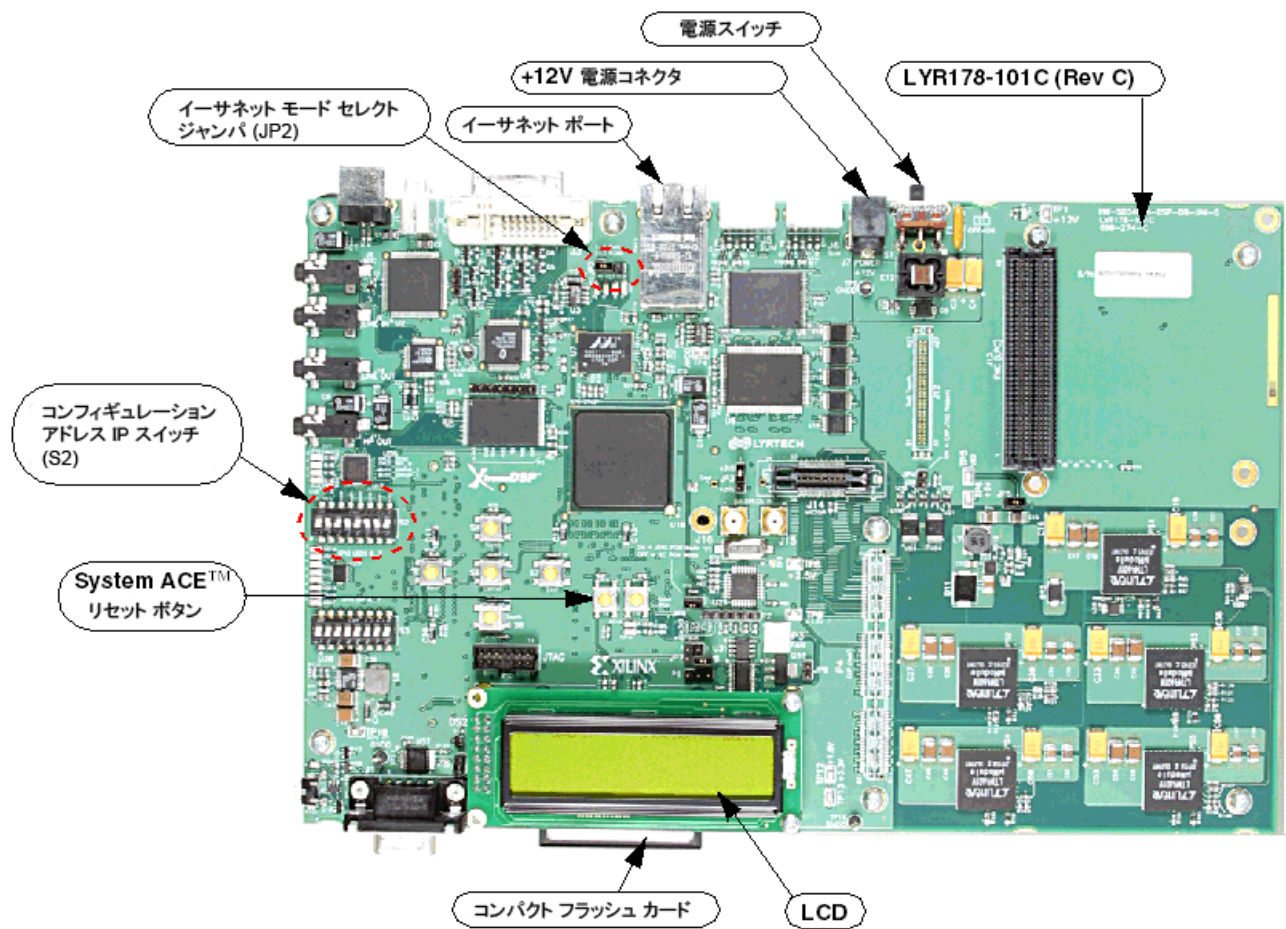
データをカードに書き込むと、カードのルート ディレクトリに `mac.dat` および `ip.dat` ファイルがあります。これらのファイルは、プラットフォームに関連付けるイーサネット MAC アドレスと IPv4 アドレスを指定します。これらのアドレスは、イーサネット ハードウェア協調シミュレーション中にターゲット プラットフォームを識別するために使用されます。

- a. `mac.dat` ファイルをテキスト エディタで開き、イーサネット MAC アドレスを変更します。MAC アドレスは、2 桁の 16 進数 6 個をコロンで区切って指定します (00:0a:35:11:22:33 など)。すべて 0、ブロードキャスト、マルチキャスト MAC アドレスはサポートされていません。
- b. `ip.dat` ファイルをテキスト エディタで開き、IP アドレスを変更します。IP アドレスは、10 進数をピリオドで区切った表記方法 (192.168.8.1 など) で指定します。すべて 0、ブロードキャスト、マルチキャスト、ループバック IP アドレスはサポートされていません。

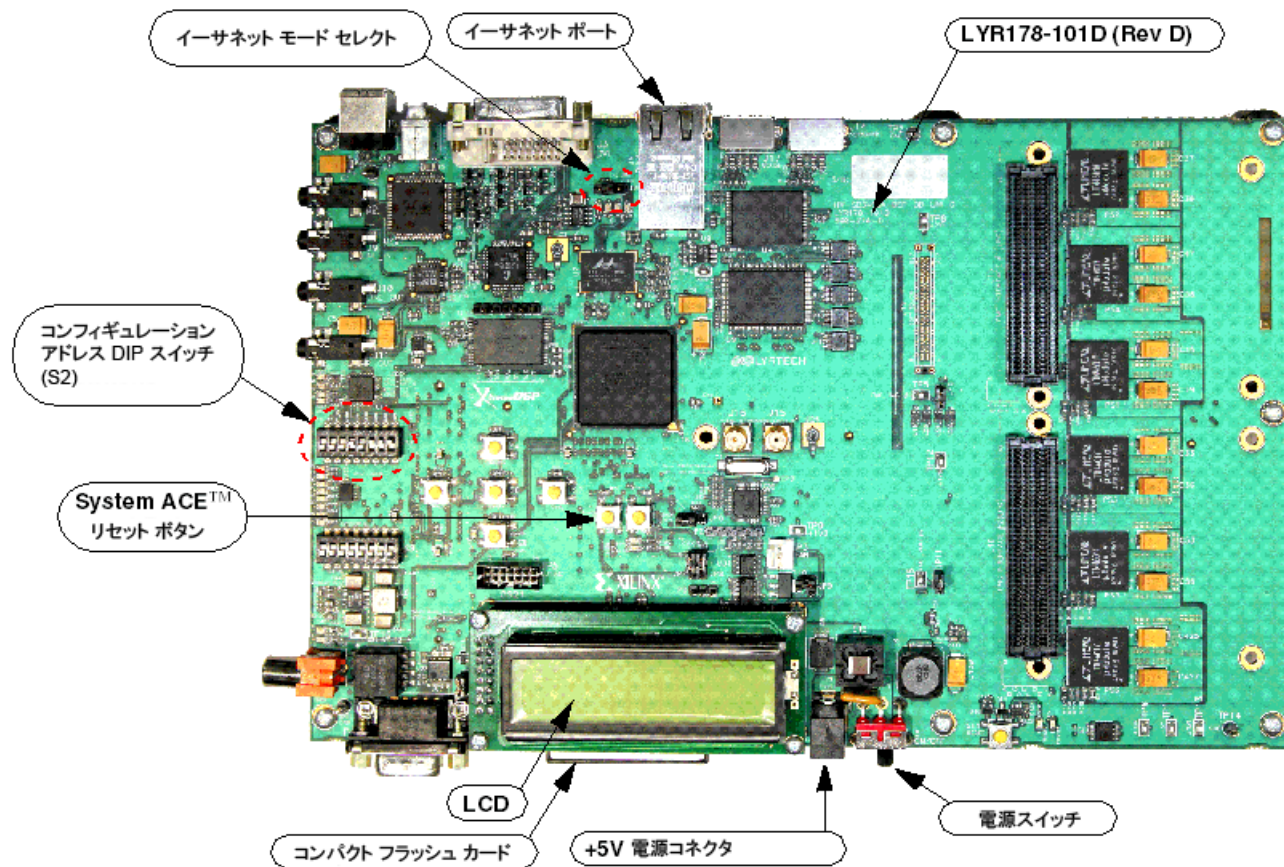
ML506 プラットフォームの IP アドレスを変更したら、「[PC 上の LAN \(ローカル エリア ネットワーク\) の設定](#)」の手順に従って PC のネットワーク接続の IP アドレスも変更します。直接接続するには、ML506 と PC を同じサブネットにする必要があります。同じサブネットにしない場合は、ML506 の IP アドレスに PC から、PC の IP アドレスに ML506 からアクセスできるようにしてください。

Spartan-3A DSP 3400A 開発プラットフォームの設定

次の図に、この設定手順に関連する Spartan-3A DSP 3400A プラットフォーム (Rev C) コンポーネントを示します。

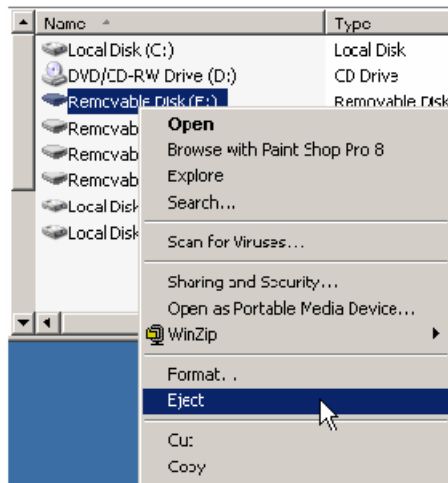


次の図に、この設定手順に関連する Spartan-3A DSP 3400A プラットフォーム (Rev D) コンポーネントを示します。



1. Spartan-3A 3400A 開発プラットフォームを、上図に示すように LCD ディスプレイが下になるように配置します。
2. 電源スイッチがオフになっていることを確認します。

3. 次の図に示すように、コンパクト フラッシュ リーダのリムーバブル ディスクを右クリックして [取り出し] をクリックします。



4. コンパクト フラッシュ カードをコンパクト フラッシュ リーダから取り出します。
5. コンパクト フラッシュ カードのスロット (Spartan-3A DSP 3400A プラットフォームの裏側) に、コンパクト フラッシュ カードをラベルがプラットフォームの反対側になるように挿入します。次の図に、プラットフォームの裏側にコンパクト フラッシュ カードを正しく挿入した様子を示します。

メモ：プラットフォームに含まれるコンパクト フラッシュ カードが、写真のものと異なる場合があります。

注意：コンパクト フラッシュ カード は無理に挿入したり 取り 出したり せず、取り扱いに注意してください。



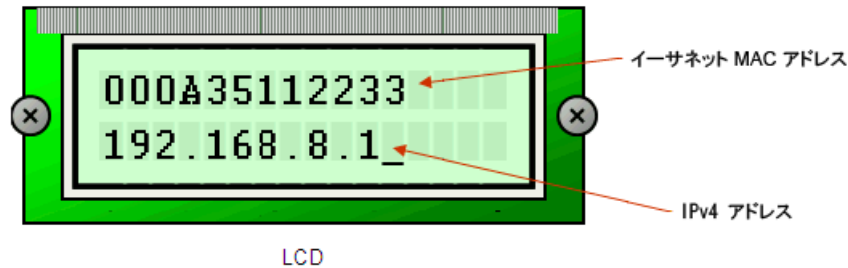
6. Rev C の 3400A 開発プラットフォームを使用している場合は、+12V 電源アダプタ ケーブルを電源コネクタに接続し、電源を AC 電源に接続します。

Rev D の 3400A 開発プラットフォームを使用している場合は、+5V 電源アダプタ ケーブルを電源コネクタに接続し、電源を AC 電源に接続します。

注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。

7. RJ45 オス/オス イーサネット ケーブルを使用して、Spartan-3A DSP 3400A プラットフォーム上のイーサネット コネクタをホスト PC 上のイーサネット コネクタに接続します。

8. コンフィギュレーションアドレス DIP スイッチ S2 を、1 をオフ、2 をオン、3 をオフ、4 をオン、5 をオフ、6 をオン、7 をオフ、8 をオフにします。
9. イーサネット モード セレクト ジャンパ JP2 で、ピン 1 と 2 を接続します (デフォルト GMII)。
10. コンフィギュレーション設定を確認します。
 - a. ターゲット プラットフォームの電源をオンにします。
 - b. プラットフォーム上の 16 文字、2 行の LCD に表示されている情報を確認します (次の図を参照)。エラーが発生していない場合は、1 行目にイーサネット MAC アドレス (コロンなし)、2 行目に IPv4 アドレスが表示されます。



- c. LCD にこれらの情報が表示されない場合は、System ACE リセット ボタンを押して、FPGA をリコンフィギュレーションします。
11. イーサネット インターフェイスと接続ステータスを確認します。
 - a. ホストから ICMP ping を発行して、プラットフォームがホストからアクセス可能であることを確認します。たとえば、コンソールで「ping 192.168.8.1」と入力すると、IP アドレス 192.168.8.1 でのプラットフォームへの接続をテストできます。

```
Command Prompt
C:\>ping 192.168.8.1
Pinging 192.168.8.1 with 32 bytes of data:
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Reply from 192.168.8.1: bytes=32 time<1ms TTL=32
Ping statistics for 192.168.8.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
C:\>
```

- b. ターゲット FPGA は、UDP ポート 9999 から信号を受信します。ネットワーク ベースのイーサネット コンフィギュレーションを使用する場合は、ネットワークに関連する通信が遮断されていないことを確認してください。これは、ポイントツーポイント イーサネット コンフィギュレーションには影響しません。

Spartan-3A DSP 3400A 開発プラットフォームの詳細情報は、次のサイトから『XtremeDSP Development Platform: Spartan-3A DSP 3400A Edition』を参照してください。

http://japan.xilinx.com/support/documentation/boards_and_kits/ug498_s3a_3400_board.pdf

JTAG ハードウェア協調シミュレーション用の ML402 プラットフォームのインストール

次に、ML402 プラットフォームで JTAG ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

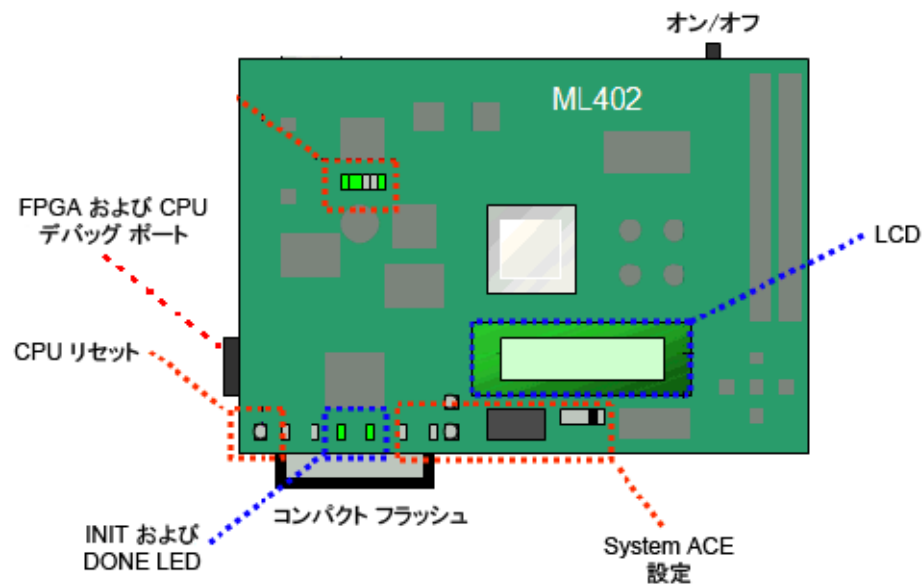
1. ザイリンクス Virtex-4 SX ML402 プラットフォーム。次のものが含まれています。
 - a. Virtex-4 ML402 プラットフォーム
 - b. ML402 キットに含まれる 5V 電源
 - c. コンパクト フラッシュ カード
2. 次のものも必要です。
 - a. ザイリンクス パラレル ケーブル IV と Power Jack スプリッタ ケーブル、またはザイリンクス プラットフォーム USB ケーブルと 14 ピン リボン ケーブル
 - b. PC 用のコンパクト フラッシュ リーダ

ホスト PC へのソフトウェアのインストール

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン

ML402 プラットフォームの設定

次の図に、この JTAG 設定手順に関連する ML402 コンポーネントを示します。



1. ML402 プラットフォームを、Virtex-4 とザイリンクスのロゴがプラットフォームの上部になるように配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。

3. ザイリンクス パラレル ケーブル IV を使用する場合は、次の手順 a ~ d に従います。
 - a. ザイリンクス パラレル ケーブル IV の DB25 プラグ コネクタを IEEE-1284 準拠の PC パラレル (プリンタ) ポート コネクタに接続します。
 - b. 14 ピンの 6 インチ高パフォーマンス リボン ケーブルを使用して、ザイリンクス パラレル ケーブル IV のポッド エンドを ML402 プラットフォームの FPGA および CPU デバッグ ポート (上図を参照) に接続します。
 - c. Power Jack ケーブルを PC のキーボード/マウス コネクタに接続します。
 - d. 必要に応じて、キーボード/マウス ケーブルのオス側をザイリンクス Power Jack ケーブル (スプリッタ ケーブル) のメス コネクタに接続します。
4. ザイリンクス プラットフォーム ケーブル USB を使用する場合は、次の手順 a ~ b に従います。
 - a. ザイリンクス プラットフォーム ケーブル USB を PC 上の USB ポートに接続します。
 - b. 14 ピンの 6 インチ高パフォーマンス リボン ケーブルを使用して、ザイリンクス プラットフォーム ケーブル USB のポッド エンドを ML402 プラットフォームの FPGA および CPU デバッグ ポート (上図を参照) に接続します。
5. AC 電源コードを電源装置に接続し、電源アダプタ ケーブルを ML402 プラットフォームに接続し、電源を AC 電源に接続します。

注意: 正しい電圧および電力定格の適切な電源を使用していることを確認してください。
6. ML402 プラットフォームの電源をオンにします。

JTAG ハードウェア協調シミュレーション用の ML605 プラットフォームのインストール

次に、ML605 プラットフォームで JTAG ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

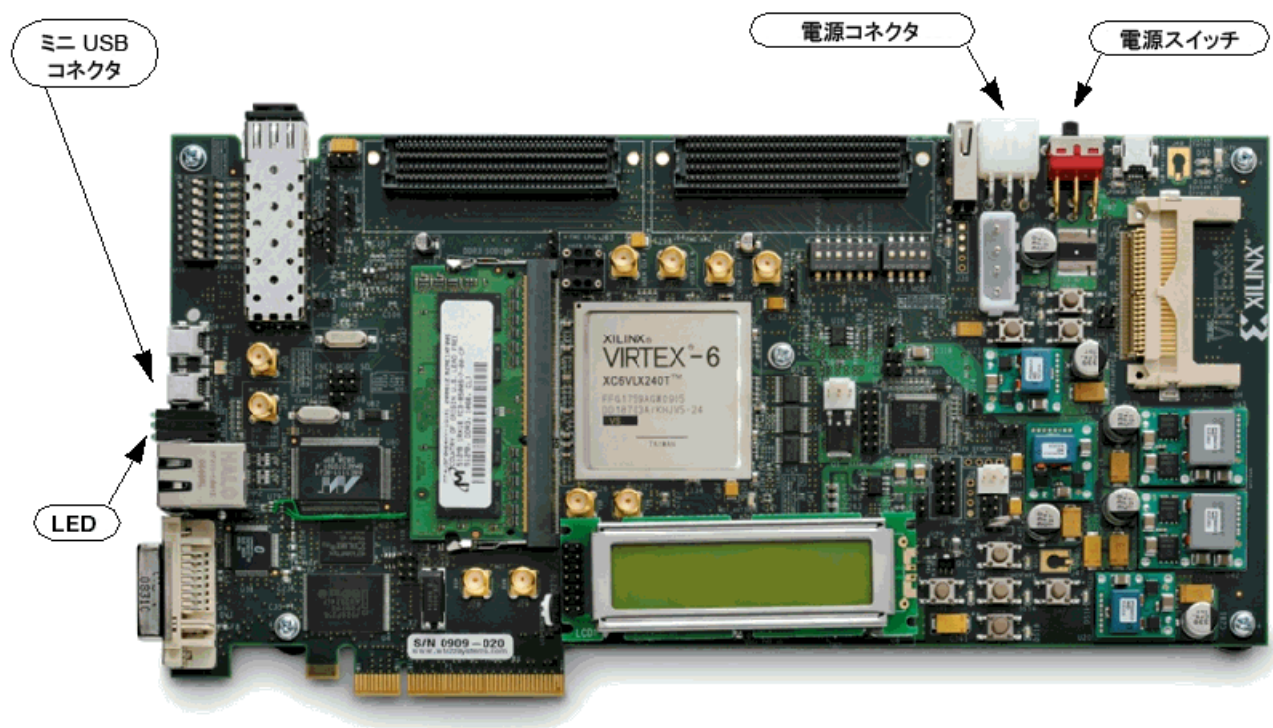
1. ザイリンクス Virtex-6 SX ML605 プラットフォーム。次のものが含まれています。
 - a. Virtex-6 ML605 プラットフォーム
 - b. ML605 キットに含まれる 12V 電源
 - c. ミニ USB ケーブル

ホスト PC へのソフトウェアのインストール

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン

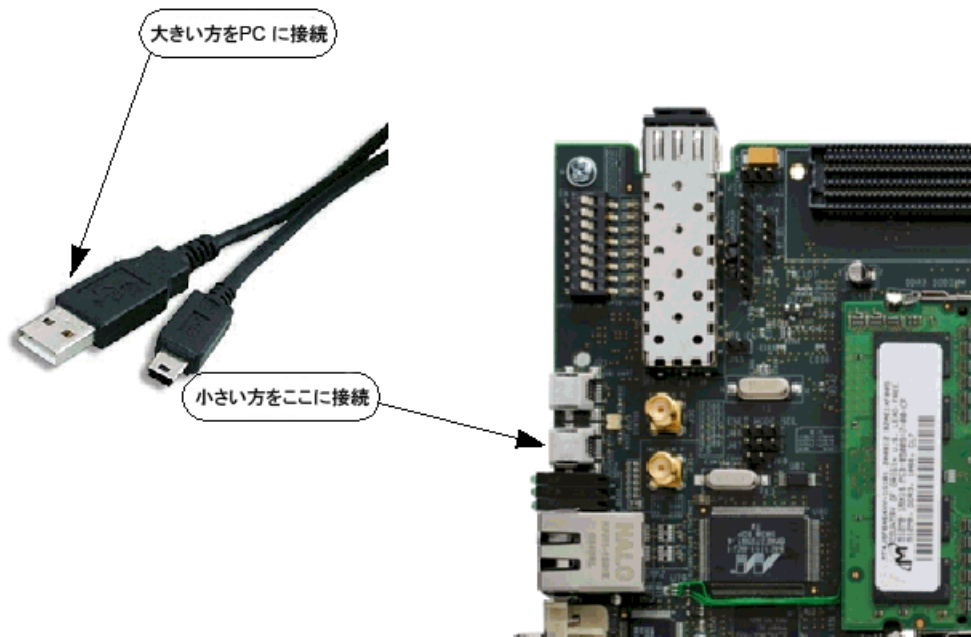
ML605 プラットフォームの設定

次の図に、この JTAG 設定手順に関連する ML605 コンポーネントを示します。

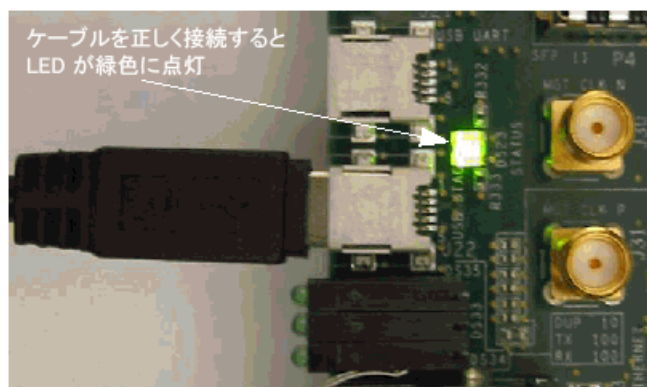


1. ML605 プラットフォームを上図のように配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。

3. 次の図に示すように、ミニ USB ケーブルの小さい方の端を LED に最も近いコネクタ USB ソケットに接続します。



4. ミニ USB ケーブルの大きい方の端を PC の USB ソケットに接続します。
ケーブルを正しく接続すると、次の図に示すように、ミニ USB コネクタの横にある LED が緑色に点灯します。



5. AC 電源コードを電源装置に接続し、電源アダプタ ケーブルを ML605 プラットフォームに接続し、電源を AC 電源に接続します。

注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。

6. ML605 プラットフォームの電源をオンにします。

JTAG ハードウェア協調シミュレーション用の SP605 プラットフォームのインストール

次に、SP605 プラットフォームで JTAG ハードウェア協調シミュレーションを実行するために必要なハードウェアおよびソフトウェアのインストール方法と設定方法を説明します。

必要なハードウェア

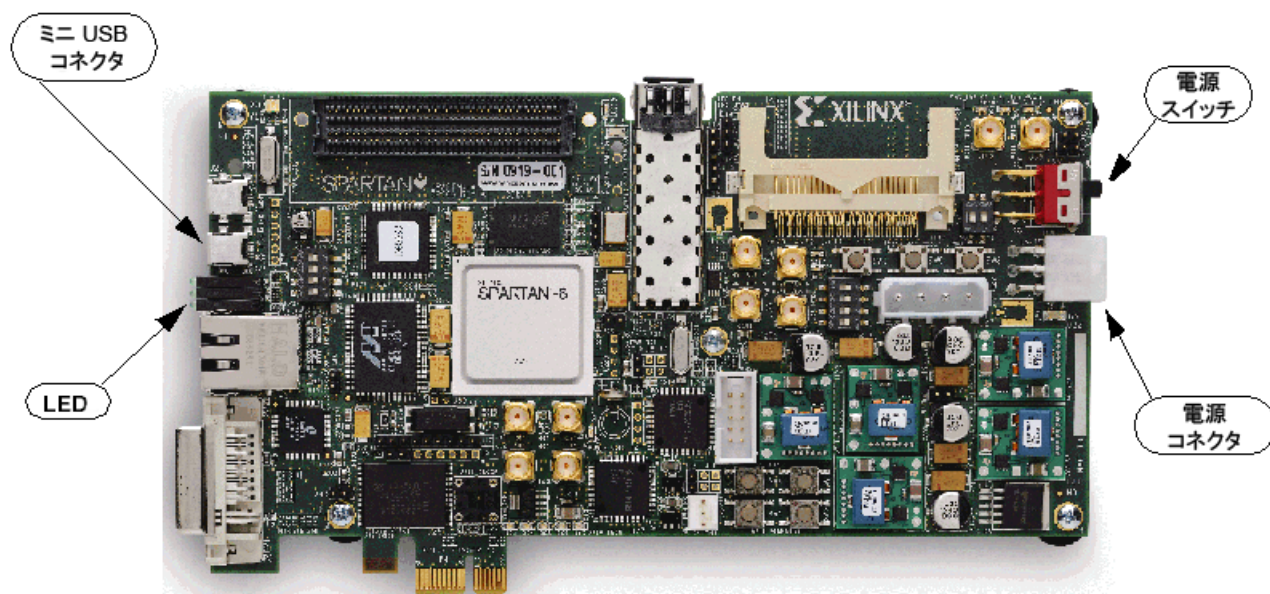
1. ザイリンクス Spartan-6 SP605 キット。次のものが含まれています。
 - a. Spartan-6 LXT SP605 プラットフォーム
 - b. 12V 電源
 - c. ミニ USB ケーブル

ホスト PC へのソフトウェアのインストール

- 現在の System Generator リリース ノートで指定されている System Generator のバージョン
- 現在の System Generator リリース ノートで指定されているザイリンクス ISE のバージョン

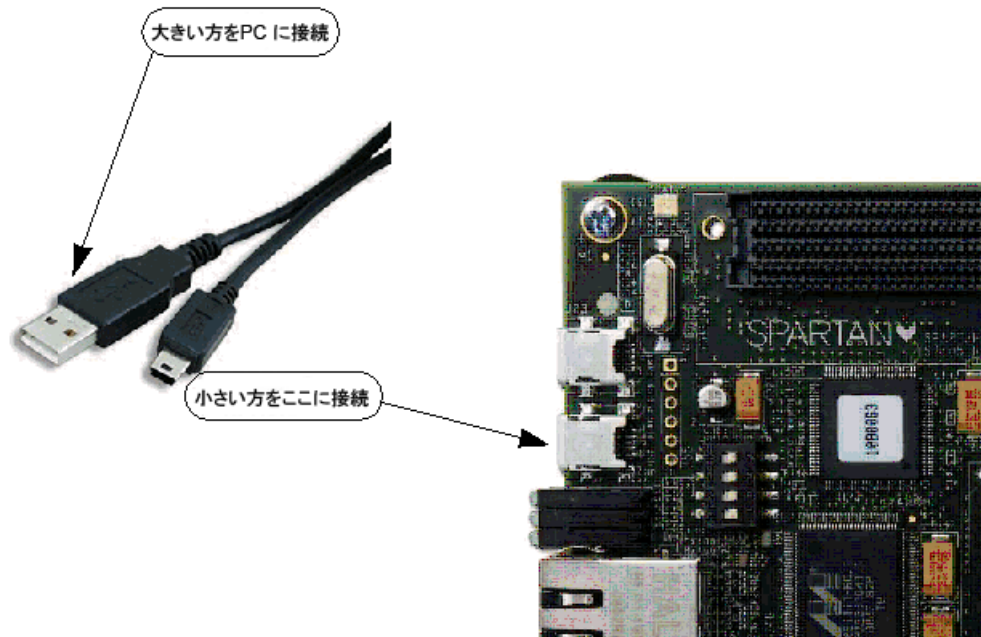
SP605 プラットフォームの設定

次の図に、この JTAG 設定手順に関連する SP605 コンポーネントを示します。



1. SP605 プラットフォームを上図のように配置します。
2. 電源スイッチが右上にあり、オフになっていることを確認します。

3. 次の図に示すように、ミニ USB ケーブルの小さい方の端を LED に最も近いコネクタ USB ソケットに接続します。



4. ミニ USB ケーブルの大きい方の端を PC の USB ソケットに接続します。
5. AC 電源コードを電源装置に接続し、電源アダプタ ケーブルを SP605 プラットフォームに接続し、電源を AC 電源に接続します。
注意：正しい電圧および電力定格の適切な電源を使用していることを確認してください。
6. SP605 プラットフォームの電源をオンにします。

JTAG ハードウェア協調シミュレーション用の新規プラットフォームのサポート

System Generator では、JTAG とザイリンクス プログラム ケーブル (パラレル ケーブル IV、プラットフォーム ケーブル USB など) を使用して FPGA ハードウェアと通信する汎用インターフェイスが提供されています。このインターフェイスは、JTAG の機能を活用して System Generator の Hardware in the Loop シミュレーションをほかのさまざまな FPGA プラットフォームに拡張しています。

ハードウェア要件

FPGA プラットフォームでは、次のハードウェア コンポーネントが含まれていれば、JTAG ハードウェア協調シミュレーション インターフェイスをサポートできます。

- System Generator でサポートされているザイリンクス FPGA デバイス (System Generator トークンのパラメータ ダイアログ ボックスの [Part] で選択可能なデバイス)
- FPGA にフリーランニング クロック ソースを共有するオンボード オシレータ
- FPGA へのアクセスを提供する JTAG ヘッド

新規プラットフォームのサポート

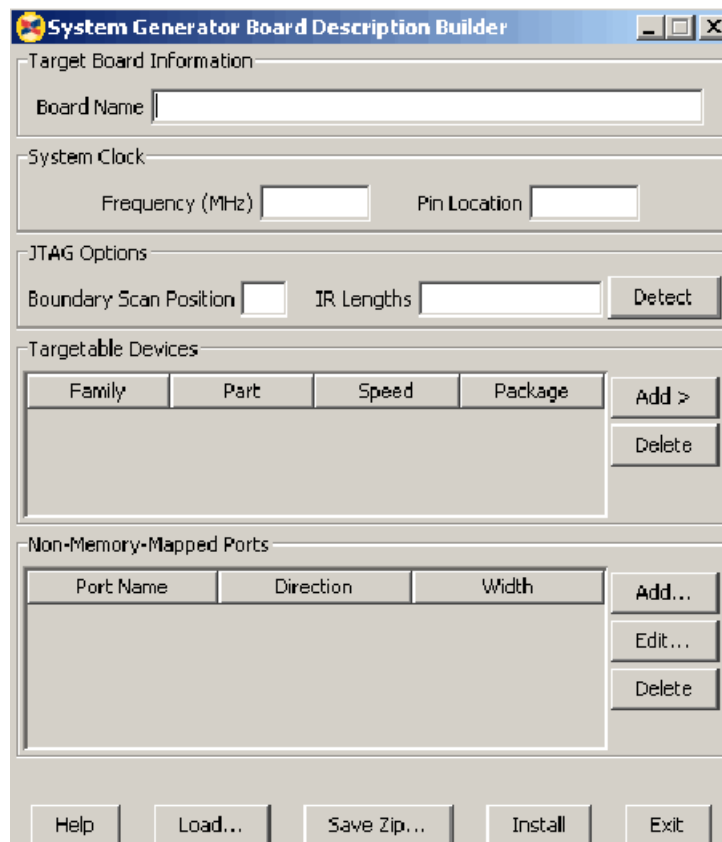
JTAG ハードウェア協調シミュレーション インターフェイスは汎用ですが、FPGA プラットフォームを System Generator でサポートさせるためには、独自のボード サポート パッケージが必要です。ボード サポート パッケージは、ボード (プラットフォーム) の情報を含む 4 つのファイルで構成されます。多くの FPGA プラットフォームには、既にボード サポート パッケージが用意されています。これらのファイルをダウンロードする方法は、「[Linux OS への System Generator のインストール](#)」を参照してください。

ハードウェア協調シミュレーション ボード サポート パッケージの提供されていない FPGA プラットフォームを使用する場合は、プラットフォームが[ハードウェア要件](#)を満たしていれば、独自のボード サポート パッケージを作成できます。プラットフォームのボード サポート パッケージの作成は簡単です。System Generator では、System Generator Board Description Builder (SBDBuilder) というユーティリティが提供されており、新規ボード サポート パッケージをグラフィカル インターフェイスを使用して作成できます。System Generator に含まれているテンプレート ファイルを手動で編集し、ボード サポート パッケージを作成することも可能です。

SBDBuilder を起動するには、MATLAB の [Command Window] に「xlSBDBuilder」と入力するか、System Generator トークンの [Compilation] で [Hardware Co-Simulation] → [New Compilation Target] をクリックします。

SBDBuilder のダイアログ ボックス

SBDBuilder を起動すると、次のようなダイアログ ボックスが開きます。



Family	Part	Speed	Package
--------	------	-------	---------

Port Name	Direction	Width
-----------	-----------	-------

このダイアログ ボックスで次に説明するようにパラメータを指定して、ボード サポート パッケージを作成します。

[Board Name]: ボード名を指定します。この名前が、System Generator トークンの [Compilation] で JTAG ハードウェア協調シミュレーション プラットフォームとしてリストされます。

[System Clock]: JTAG ハードウェア協調シミュレーションでは、System Generator デザインを駆動する オンボード クロックが必要です。ボードのシステムクロックについて、次の情報を指定します。

- [Frequency (MHz)]: オンボードのシステムクロックの周波数を MHz で指定します。

メモ: クロック周波数は 10MHz ~ 100MHz にする必要があります。ターゲット FPGA デバイスおよびデザインによっては、ハードウェア協調シミュレーション ロジックを追加すると、ハードウェア協調シミュレーション用にコンパイルしたデザインが高周波数で制約を満たさない可能性があります。

- [Pin Location]: システムクロックを接続する FPGA 入力ピンを指定します。

[JTAG Options]: System Generator でハードウェア協調シミュレーション用に FPGA をプログラムするには、FPGA ボードの JTAG チェーンに関する情報が必要です。これらの情報を見つける方法は、「プラットフォームの情報の取得」を参照してください。ボードの仕様がはっきりわからない場合は、製造業者のマニュアルを参照してください。[JTAG Options] では、次の情報を入力します。

- [Boundary Scan Position]: JTAG チェーンでのターゲット FPGA の位置を指定します。チェーンのデバイスには 1 から番号が付けられています (最初のデバイスは 1、2 番目のデバイスは 2 など)。
- [IR Lengths]: JTAG チェーン上のすべてのデバイスに対して命令レジスタの幅を指定します。値の間は、スペース、カンマ、またはセミコロンで区切ります。
- [Detect]: このボタンをクリックすると、FPGA ボードにアクセスして自動的に IR 幅が検出されます。この機能を使用するには、ボードの電源をオンにし、パラレル ケーブル IV に接続しておく必要があります。JTAG チェーン上の不明のデバイスの IR 幅はクエスチョン マーク (?) で示されるので、手動で指定する必要があります。

[Targetable Devices]: この表には、ボード上の FPGA でプログラム可能なものがリストされます。JTAG チェーンのすべてのデバイスがリストされるわけではなく、指定したバウンダリ スキャン位置に存在する可能なデバイスがリストされます。ほとんどのボードでは、指定する必要のあるデバイスは 1 のみですが、ボードによっては複数のデバイスから選択可能です (同じソケットにある XCV1000 または XCV2000 など)。次のように [Add] および [Delete] ボタンを使用して、デバイスリストを作成します。

- [Add]: ボード上のデバイスを選択するメニューが表示されます。次の図に示すように、デバイスはファミリー、パーツ名、スピード、パッケージタイプで分類されています。
- [Delete]: リストから選択したデバイスを削除します。

spartan2 ▶			
spartan2e ▶	xc2s50e ▶		
spartan3 ▶	xc2s100e ▶		
virtex ▶	xc2s150e ▶	-7 ▶	
virtexe ▶	xc2s200e ▶	-6 ▶	ft256
virtex2 ▶	xc2s300e ▶		fg456
virtex2p ▶	xc2s400e ▶		pq208
virtex4 ▶	xc2s600e ▶		

[Non-Memory-Mapped Ports]：ボード専用ポートのサポートを追加できます。ボード専用ポートは、ハードウェア協調シミュレーション中に FPGA でアクセスする必要があるオンボード コンポーネント (外部メモリ、DAC、ADC など) がある場合に便利です。ボード専用ポートは、ハードウェア協調シミュレーション用にデザインをコンパイルしたときに、Simulink ポートが作成されるのではなく物理的な場所にマップされるので、[Non-Memory-Mapped Ports] (メモリ マップされないポート) と示されています。詳細は、「[メモリ マップされないポートの指定](#)」を参照してください。[Add]、[Edit]、および [Delete] ボタンを使用して、ボード専用ポートを設定します。

- [Add]：ポートの情報を入力するダイアログ ボックスが表示されます。
- [Edit]：選択したポートに変更を加えます。
- [Delete]：リストから選択したポートを削除します。

[Help]：このマニュアルを表示します。

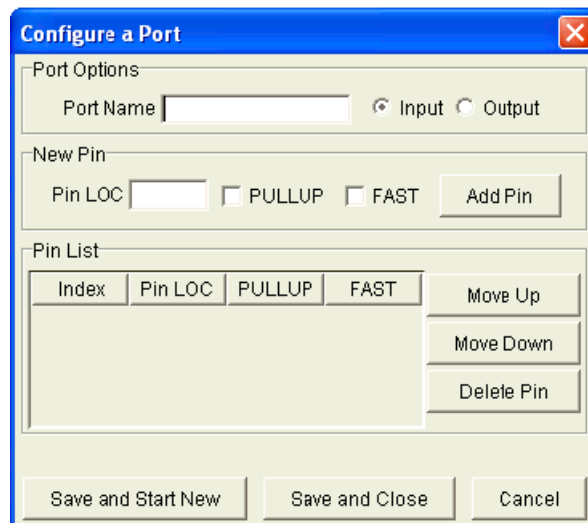
[Load]：読み込む SBDBuilder Saved Description (保存された記述) XML ファイルを選択します。このファイルは、プラグインを作成するたびに保存されます。以前のプラグイン ファイルを読み込んで、編集する場合に便利です。

[Save Zip]：System Generator のプラグイン ファイルすべてを含む ZIP ファイルを作成します。開いたダイアログ ボックスで、ファイル名とパス名を指定します。作成される ZIP ファイルは、System Generator の [xlInstallPlugin](#) 関数に渡すのに適切なフォーマットです。

[Exit]：アプリケーションを終了します。

メモリ マップされないポートの指定

SBDBuilder を使用して、FPGA プラットフォームのメモリ マップされないポートを指定できます。SBDBuilder のメインのダイアログ ボックスで、[Non-Memory-Mapped Ports] の [Add] または [Edit] をクリックすると、次の図に示す [Configure a Port] ダイアログ ボックスが表示されます。



The "Configure a Port" dialog box is shown with the following sections:

- Port Options**: A text field for "Port Name" and two radio buttons, "Input" (selected) and "Output".
- New Pin**: A text field for "Pin LOC", two checkboxes "PULLUP" and "FAST", and an "Add Pin" button.
- Pin List**: A table with columns "Index", "Pin LOC", "PULLUP", and "FAST". To the right of the table are three buttons: "Move Up", "Move Down", and "Delete Pin".
- Buttons**: At the bottom are three buttons: "Save and Start New", "Save and Close", and "Cancel".

このダイアログ ボックスで、ポートに関する次の情報を設定します。

[Port Options] : ポート全体に影響するオプションを指定します。

- [Port Name] : System Generator で表示するポート 名を指定します。MATLAB に準拠した名前を付ける必要があります (最初の文字はアルファベット、その後の文字は英数字またはアンダースコア)。
- [Input]/[Output] : ポートの方向を指定します。

[New Pin] : ポートに追加するピンの情報を入力します。ポートは、ブール値の単一ピン、またはベクタまたはバスの複数ピンで構成されます。

- [Pin LOC] : FPGA 上のピンの絶対位置をロケーション制約により定義します。FPGA プログラムを実際のハードウェア接続に正しく対応させるため、各ピンに対して定義する必要があります。
- [PULLUP] : ピンに PULLUP 制約を設定します。この制約を設定するとロジック レベルが High になり、トライステート ネットが駆動されていない場合でもフロート状態になりません。
- [FAST] : ピンに FAST 制約を設定します。この制約を設定すると、IOB 出力の速度が上がりますが、ノイズおよび消費電力が増加する場合があります。
- [Add Pin] : ピンをポートに追加します。このボタンをクリックしないと、ピンがポートに追加されません。

メモ : [Pin LOC] にカーソルがあるときに Enter キーを押しても、ピンがポート に追加されます。

[Pin List] :

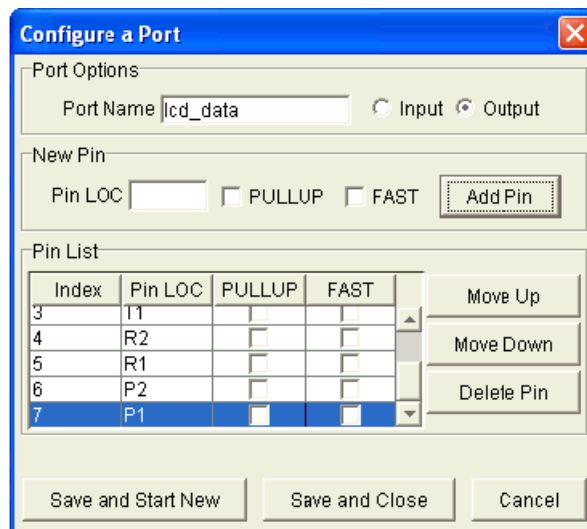
- [Index] : 直接変更できません。ポートは複数ビットにできるので、ピンのベクタで表されます。[Index] は、ポートにおける特定のピンのビット位置を示します。0 が最下位ビットです。
- [Move Up]/[Move Down] : ピン リストで選択したピンを上または下に移動します。ポートのベクタ ビット順を修正するのに便利です。
- [Delete Pin] : リストから選択したピンを削除します。

[Save and Start New] : ポートをボード サポート パッケージに保存します。入力した情報がクリアされ、新しいポートの情報を入力できるようになります。

[Save and Close] : ポートをボード サポート パッケージに保存し、メインのダイアログ ボックスに戻ります。

[Cancel] : 現在のポートの情報を破棄し、メインのダイアログ ボックスに戻ります。

ポートの入力が終了すると、ダイアログ ボックスは次のようになります。

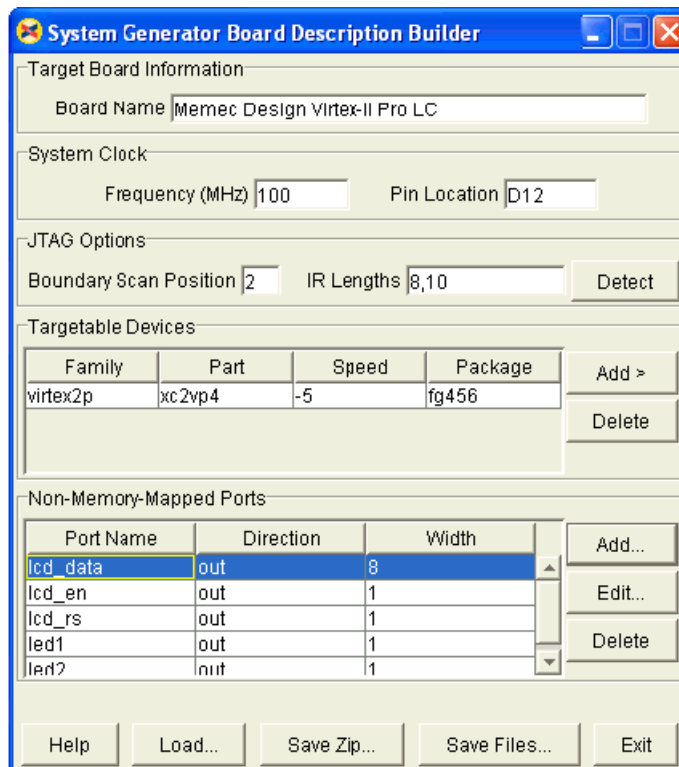


The "Configure a Port" dialog box is shown. It has a title bar with a close button. The "Port Options" section contains a "Port Name" field with "lcd_data" and radio buttons for "Input" and "Output", with "Output" selected. The "New Pin" section has a "Pin LOC" field, checkboxes for "PULLUP" and "FAST", and an "Add Pin" button. The "Pin List" section contains a table with columns "Index", "Pin LOC", "PULLUP", and "FAST". The table has 5 rows, with the last row (Index 7, Pin LOC P1) selected. To the right of the table are buttons "Move Up", "Move Down", and "Delete Pin". At the bottom are buttons "Save and Start New", "Save and Close", and "Cancel".

Index	Pin LOC	PULLUP	FAST
3	T1	<input type="checkbox"/>	<input type="checkbox"/>
4	R2	<input type="checkbox"/>	<input type="checkbox"/>
5	R1	<input type="checkbox"/>	<input type="checkbox"/>
6	P2	<input type="checkbox"/>	<input type="checkbox"/>
7	P1	<input type="checkbox"/>	<input type="checkbox"/>

プラグイン ファイルの保存

プラットフォームに関する情報をダイアログ ボックスに入力すると、ダイアログ ボックスは次のようになります。



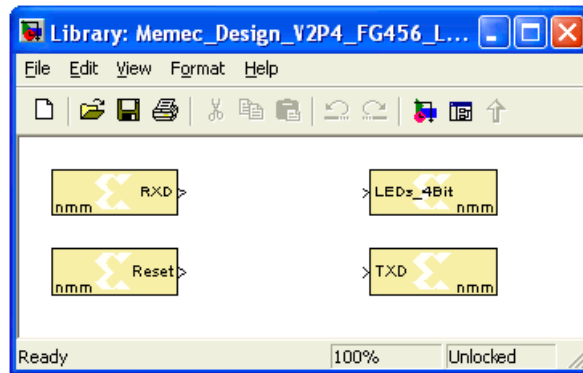
The "System Generator Board Description Builder" dialog box is shown. It has a title bar with standard window controls. The "Target Board Information" section has a "Board Name" field with "Memec Design Virtex-II Pro LC". The "System Clock" section has a "Frequency (MHz)" field with "100" and a "Pin Location" field with "D12". The "JTAG Options" section has a "Boundary Scan Position" field with "2", an "IR Lengths" field with "8,10", and a "Detect" button. The "Targetable Devices" section has a table with columns "Family", "Part", "Speed", and "Package". The table has 1 row with values "virtex2p", "xc2vp4", "-5", and "fg456". To the right of the table are buttons "Add >" and "Delete". The "Non-Memory-Mapped Ports" section has a table with columns "Port Name", "Direction", and "Width". The table has 5 rows, with the first row (Port Name lcd_data, Direction out, Width 8) selected. To the right of the table are buttons "Add...", "Edit...", and "Delete". At the bottom are buttons "Help", "Load...", "Save Zip...", "Save Files...", and "Exit".

Family	Part	Speed	Package
virtex2p	xc2vp4	-5	fg456

Port Name	Direction	Width
lcd_data	out	8
lcd_en	out	1
lcd_rs	out	1
led1	out	1
led2	inut	1

ここで、ボード サポート パッケージを System Generator のプラグイン ZIP ファイルに保存するか、「ボード サポート パッケージ ファイル」に説明されているように、ボード サポート パッケージ ファイルをそのまま保存します。次の SBDBuilder ファイルも作成されます。

- `yourboard.xml` : SBDBuilder Saved Description ファイル。以前に作成したプラグインを再読み込みする際に使用します。このファイルに指定した名前 (`yourboard`) がほかのファイルにも使用されます。
- `yourboard_libgen.m` : デバイス上のメモリ マップされないポート用にゲートウェイを自動的に作成するスクリプト。このスクリプトを実行すると、次の図のようなライブラリが作成されます。



ボード サポート パッケージ ファイル

JTAG ハードウェア協調シミュレーションをサポートする FPGA プラットフォームは、**System Generator** でボード サポート パッケージにより定義されます。このボード サポート パッケージには、デバイス設定や、プラットフォームで提供される JTAG およびバウンダリ スキャン インターフェイスに関する情報など、プラットフォームに関する有益な情報が含まれます。ボード サポート パッケージは、次のファイルで構成されます。

メモ : このマニュアルでは、3 つのファイル名の頭に `yourboard` が付いています。この部分は、ボードに適切な名前 (`xtremedspkit`、`mblazedemo` など) に置き換えてください。

1. `xltarget.m` : FPGA プラットフォームがコンパイル ターゲットであることを **System Generator** に示します。各コンパイル ターゲットにそれぞれ `xltarget.m` ファイルがあります。この関数は、コンパイル ターゲットの名前 (**System Generator** トークンのパラメータ ダイアログ ボックスに表示される名前)、ボードに関する情報を検索する関数の名前をツールに示します。
2. `yourboard_target.m` : デバイス情報、クロック周波数、クロック ピンのロケーションなど、FPGA プラットフォームに関する情報を **System Generator** トークンのパラメータ ダイアログ ボックスで設定します。
3. `yourboard_postgeneration.m` : HDL ネットリスト生成後にプラットフォーム用に FPGA コンフィギュレーション ファイルを生成するために実行するスクリプトを示します。プラットフォームのバウンダリ スキャン チェーンでのデバイス位置、各デバイスの命令レジスタ幅など、**System Generator** トークンのパラメータ ダイアログ ボックスに含まれない情報も指定します。この関数は、`post-generation` 関数として参照されます。
4. `yourboard.ucf` : FPGA プラットフォームのユーザー制約ファイル (UCF)。クロック ピン ロケーションおよび周波数、ボード専用ポートの制約を指定します。

System Generator のインストール ディレクトリには、上記のファイルのテンプレートがあります。4 つのテンプレート ファイルをプラットフォームに関する情報に変更すると、新規ボードのボード サポート パッケージを作成できます。`yourboard` を適切な名前に変更してください。

各テンプレートには、変更する必要があるフィールド、それらのフィールドの値を示す手順が含まれています。変更が必要なフィールドは、~~~ の下線で示されています。テンプレート ファイルは、`<path_to_sysgen>\hwcosim\jtag\templates` にあります。

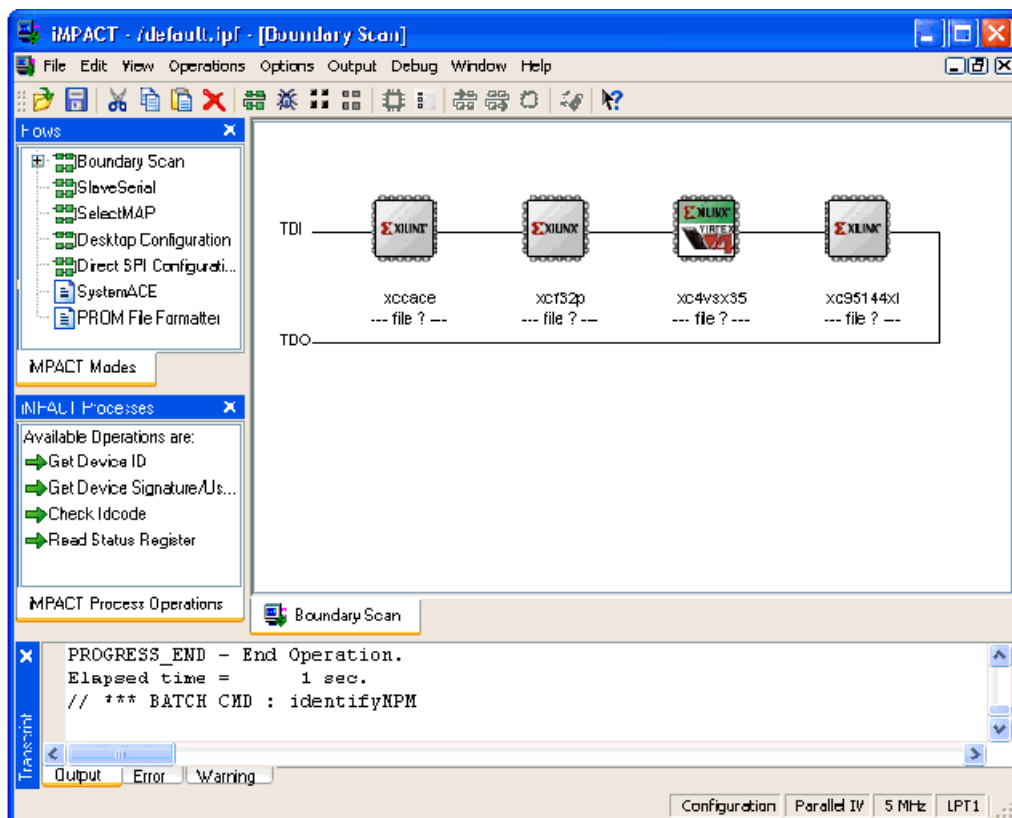
プラットフォームの情報の取得

SBDBuilder (またはボード サポート パッケージのテンプレート ファイル) では、FPGA プラットフォームに関する情報が必要です。次の表に、必要な情報を示します。

情報	説明
クロック ピン ロケーション	FPGA システム クロック ソースのピン ロケーション制約。
クロック周期	FPGA システム クロック ソースの PERIOD 制約。
バウンダリ スキャン チェーンでのデバイスの 位置	プラットフォームのバウンダリ スキャン チェーンでのターゲット FPGA の位置を指定します。チェーンのデバイスには 1 から番号が付 けられており、最初のデバイスは 1 になります。
命令レジスタ幅	バウンダリ スキャン チェーンの各デバイスの命令レジスタ幅。

クロック ピン ロケーションと周期は、ベンダーのマニュアル、既存の制約ファイル、ベンダーのオンライン資料/サポートなど、さまざまなソースから取得できます。

プラットフォームのバウンダリ スキャン チェーンにどのデバイスがあるのかわからない場合は、iMPACT を使用して調べることができます。iMPACT はザイリックス ISE に含まれているツールで、デバイスのコンフィギュレーションおよびファイルの生成を実行します。iMPACT を起動すると、プラットフォームのバウンダリ スキャン チェーンが自動的に検出され、次のようにグラフィックで示されます。



バウンダリ スキャン チェーンデバイスを特定したら、各デバイスの命令レジスタ幅を判断する必要があります。次の表に、ザイリンクス ファミリの命令レジスタ幅を示します。命令レジスタ幅は、**SBDBuilder** の自動検出機能を使用して調べることもできます。この機能で命令レジスタ幅を判断できない場合は、次の表を参照してください。

デバイス ファミリ	IR 幅
XC9500 / XC9500XL / XC9500XV	8
XC1800 / XC18V00	8
XC4000XL/XLA	3
Spartan-XL	3
System_ACE-CF	8
Virtex / Virtex-E(EM)	5
Spartan-II / Spartan-IIE	5
Virtex-II	6
Spartan-3	6
Spartan-3E	6
Spartan-3A/Spartan-3AN	6
Spartan-3A DSP	6
Virtex-II Pro 2、4、7	10
Virtex-II Pro 20、30、40、50、70、100	14
Virtex-II Pro 125	16
Virtex-4 LX	10
Virtex-4 SX	10
Virtex-4 FX 12、20	10
Virtex-4 FX 40、60、100、140	14
Virtex-5 LX	10
XCR3000XL	5
XCR3000A / XCR3128	4
XCR3320 / XCR3960	5
XCR5128 / XCR5032C / XCR5064C / XCR5128C	4
CoolRunner™-II	8
Platform Flash XCFxxS	8
Platform Flash XCFxxP	16

ボード専用ポートの手動指定

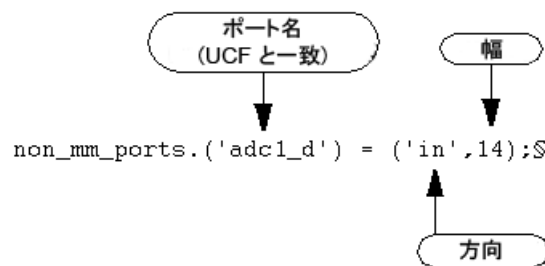
ボード サポート パッケージを作成する際に、ボード専用ポートを手動で指定できます。FPGA プラットフォームのボード専用ポートを定義するには、次の操作を実行します。

- `yourboard.ucf` テンプレート ファイルにボード専用ポートを追加します。 各制約には、「<port> contingent」(<port> はボード専用ポートの名前) というコメントをつける必要があります。System Generator でハードウェアのモデルをコンパイルすると、カスタム UCF ファイルが作成されます。モデルで使用されない信号に関連付けられている制約は、カスタム UCF ファイルから削除されます。

次に、ポート `adc1_d(0)` と `adc1_d(1)` の例を示します。

```
net adc1_d(0) loc = af20; # adc1_d contingent
net adc1_d(1) loc = ad18; # adc1_d contingent
```

- `yourboard_postgeneration.m` 関数で、すべてのボード専用ポートを宣言します。

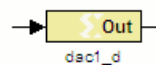


メモ：双方向ポートは、現在サポートされていません。

`yourboard_postgeneration.m` 関数に次の行を含めます。

```
params('non_memory_mapped_ports') = non_mm_ports;
```

- ボード専用ポートの情報で Gateway ブロックをカスタマイズします。
 - ◆ ライブラリを作成し、Gateway ブロックを作成します。
 - ◆ Gateway にボード専用ポートの名前を付けます。この名前は、`post-generation` 関数および UCF ファイルで使用したポート名と一致している必要があります。

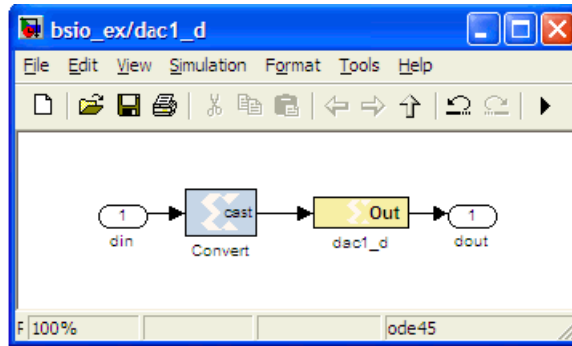


- ◆ Gateway ブロックをクリックします。
- ◆ MATLAB の [Command Window] に、次のように入力します。


```
> xlSetNonMemMap(gcb, 'Xilinx', 'jtaghwcosim')
```
- ◆ ライブラリを保存します。

これで、System Generator でボード専用の Gateway を使用できます。モデルにこの Gateway ブロックを含める場合は、Gateway を駆動する信号または Gateway で駆動される信号の幅が、ハードウェアのポートの幅と一致するようにしてください。Gateway Out ブロックを駆動する信号の幅は、Convert ブロックを使用して強制できます。

メモ：次の図に示すように、サブシステムを使用し Gateway Out ブロックと Convert ブロックのペアを保存すると便利です。



最上位コンポーネントの作成

モデルを JTAG ハードウェア協調シミュレーション用にコンパイルすると、**System Generator** でデザインの最上位 HDL エンティティが生成されます。このエンティティには、モデルに必要なロジックと、JTAG ハードウェア協調シミュレーションに必要なインターフェイス ロジックがインスタンス化されています。

プラットフォームの特定の要件により、この生成された最上位 HDL エンティティを使用できない場合があります。たとえば、プラットフォーム FPGA の DCM で生成したクロックを使用するコンポーネントがプラットフォームにある場合などです。この場合、モデルをハードウェアにコンパイルするときに独自の最上位コンポーネントを作成できます。

メモ：独自の最上位コンポーネントを使用する場合は、**System Generator** に合成済みのネットリスト (NGC、EDF、EDN) を供給する必要があります。

メモ：独自の最上位コンポーネントには、生成された汎用の JTAG ハードウェア協調シミュレーション最上位コンポーネントをインスタンス化する必要があります。コンポーネントのインスタンス化には、必要なクロック信号、ボード専用 I/O ポートを含める必要があります。次に、コンポーネントのインスタンス化の例を示します。

```
component jtagcosim_top port (
    -- required clocking ports
    sys_clk   : in std_logic;
    cosim_clk : out std_logic;
    sys_clk_buf : out std_logic;
    -- board specific ports
    adc1_d   : in std_logic_vector(13 downto 0);
    dac1_d   : out std_logic_vector(13 downto 0);
    dac1_div0 : out std_logic;
    dac1_div1 : out std_logic;
    dac1_mod0 : out std_logic;
    dac1_mod1 : out std_logic;
    dac1_reset : out std_logic
);
end component;
```

最上位ネットリストは、yourboard_postgeneration.m で次のように指定します。

```
params.vendor_toplevel = 'yourboard_toplevel';
```

ここで yourboard_toplevel は、System Generator で使用するコンパイル済みの最上位ネットリスト コンポーネントの名前です。最上位コンポーネントに関連付けられているネットリスト ファイルの名前も指定する必要があります。これらのファイルは、yourboard_postgeneration.m で次のように指定します。

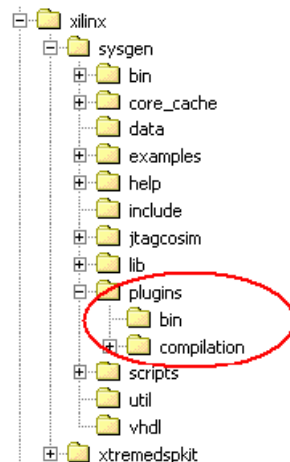
```
params.vendor_netlists = {'yourboard_toplevel.ngc', 'foo.edf'};
```

ボード サポート パッケージのインストール

SBDBuilder では、System Generator で提供される [xlInstallPlugin](#) ユーティリティを使用して自動的にインストール可能なボード サポート パッケージのプラグイン ZIP ファイルを作成できます。適切なプラグイン ZIP ファイルがない場合は、ボード サポート パッケージを手動でインストールできます。次に、System Generator のインストールディレクトリにボード サポート パッケージをインストールする方法を示します。

プラグイン ディレクトリ

System Generator のインストール ディレクトリには、新しいコンパイルターゲットのボード サポート パッケージのファイルを保存するディレクトリがあります。このディレクトリ (plugins\compilation) は System Generator コンパイル ターゲット プラグインのレポジトリとして使用され、独特の特性があります。System Generator のインストール ディレクトリは、次のようになっているはずです。

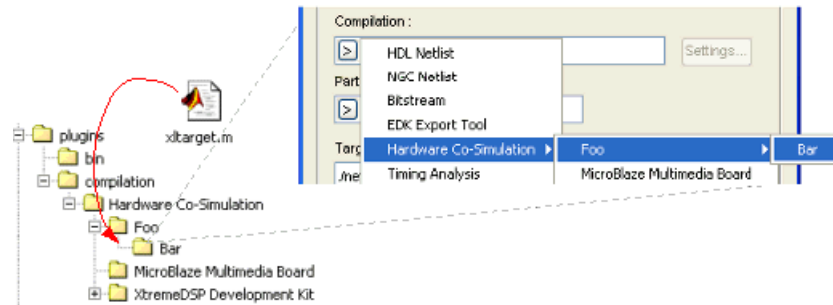


プラットフォームのボード サポート パッケージ ファイルは、plugins\compilation ディレクトリ内にサブディレクトリを作成して保存します。

メモ： ボード サポート パッケージに関連付けられているコンフィギュレーション ファイルはすべて、同じディレクトリに保存します。

System Generator では、コンパイルターゲットに対してこのディレクトリ (およびサブディレクトリ) を検索します。前述のとおり、xltarget.m ファイルは FPGA プラットフォームがコンパイルターゲットであることを System Generator に示します。System Generator は、plugins\compilation ディレクトリを検索し、各 xltarget.m ファイルに対して System Generator トークンのパラメータダイアログボックスにある [Compilation] のリストに追加します。

System Generator トークンのパラメータ ダイアログ ボックスにある [Compilation] のサブメニューは、plugins\compilation ディレクトリのディレクトリ構造を反映しています。ボード サポート パッケージの新しいディレクトリを作成すると、その名前が [Compilation] のサブメニューに表示されます。



新規パッケージの検出

System Generator で新規ターゲットが認識されるようにするには、MATLAB の [Command Window] に新規ターゲットを検索する次のコマンドを入力します。

```
xlrehash_xltarget_cache
```

これで、新規 FPGA プラットフォームが System Generator トークンのパラメータ ダイアログ ボックスで選択できるようになります。

メモ：このコマンドを実行したときに System Generator トークンのパラメータ ダイアログ ボックスが開いている場合、ダイアログ ボックスを一度閉じて開き直さないと、新規ターゲットは表示されません。

HDL モジュールのインポート

既存の HDL モジュールを System Generator デザインに追加する必要がある場合があります。System Generator の Black Box ブロックを使用すると、VHDL、Verilog、EDIF をデザインに追加できます。Black Box ブロックは、ほかのブロックと同様に、デザインに接続し、シミュレーションに含め、ハードウェアにコンパイルします。System Generator で Black Box ブロックをコンパイルすると、インポートされたモジュールと関連のファイルが周囲のネットリストに接続されます。

表 4-1:

Black Box インターフェイス	
ブラック ボックスの HDL の要件と制限	ブラック ボックスに関連する VHDL、Verilog、EDIF の要件と制限を説明します。
ブラック ボックス コンフィギュレーション ウィザード	ブラック ボックス コンフィギュレーション ウィザードの使用法を説明します。
ブラック ボックスのコンフィギュレーション M 関数	ブラック ボックス コンフィギュレーション M 関数の作成方法を説明します。

HDL 協調シミュレーション	
HDL シミュレータの設定	Black Box ブロックの HDL に対して協調シミュレーションを実行するために、ISE® ソフトウェアまたは ModelSim を設定する方法を説明します。
複数のブラック ボックスの協調シミュレーション	1 つの HDL シミュレータ セッションで複数の Black Box ブロックに対して協調シミュレーションを実行する方法を説明します。

ブラック ボックスの例 1: ブラック ボックスの HDL 要件を満たした CORE Generator モジュールのインポート	System Generator のブラック ボックス コンフィギュレーション ウィザードを使用した例を示します。
ブラック ボックスの例 2: ブラック ボックスの HDL 要件を満たす VHDL ラップが必要な CORE Generator モジュールのインポート	VHDL コア ラップが必要な例を示します。シミュレーション問題の解決方法も示します。

ブラック ボックスの例 3 : VHDL モジュールのインポート	Black Box ブロックを使用して VHDL を System Generator デザインにインポートし、ModelSim を使用して協調シミュレーションを実行する方法を示します。
ブラック ボックスの例 4 : Verilog モジュールのインポート	Verilog ブラック ボックスを System Generator で使用し、ModelSim を使用して協調シミュレーションする方法を示します。
ブラック ボックスの例 5 : ダイナミックブラック ボックス	入力幅の変化に応じて動的に調整される転置型 FIR フィルタ ブラック ボックスを使用したダイナミック ブラック ボックスを示します。
ブラック ボックスの例 6 : 複数の ブラック ボックスの同時シミュレーション	1 つの ModelSim ライセンスを使用し、複数の Black Box ブロックに対して同時に協調シミュレーションを実行する方法を示します。
ブラック ボックスの例 7 : ModelSim を使用したアドバンス ブラック ボックス	ダイナミック ポート インターフェイスを含む Black Box ブロックを設計し、マスク パラメータを使用してブラック ボックスをコンフィギュレーションする方法を示します。入力ポートのデータ型に基づいてジェネリック値を割り当て、後で再利用するために Black Box ブロックを Simulink ライブラリに保存する方法も示します。ModelSim HDL 協調シミュレーション用のカスタム スクリプトの指定方法についても説明します。
ブラック ボックスの例 8 : 暗号化 された VHDL ファイルのイン ポート、シミュレーション、エ クスポート	デザインを暗号化された VHDL ファイルとして Black Box ブロックにインポートし、デザインをシミュレーションして、VHDL を個別の暗号化されたファイルとしてエクスポートする方法を示します。

ブラック ボックスの HDL の要件と制限

ブラック ボックスに関連する HDL コンポーネントは、すべての次の System Generator の要件および制限を満たしている必要があります。

- デザインのほかのエンティティ名とは異なる名前を付けます。
- HDL ブラック ボックスで双方向ポートはサポートされますが、System Generator にポートとしては表示されません。生成された HDL にのみ含まれます。
- Verilog のブラック ボックスの場合、モジュール名やポート 名は必ず小文字にし、標準的な VHDL の命名規則に従うようにします。
- クロック ポート またはクロック イネーブル ポートのタイプは必ず `std_logic` にします。Verilog のブラック ボックスの場合、ポート はベクタではない入力にします (例 : `input clk`)。
- ブラック ボックスの HDL のクロック ポートとクロック イネーブル ポートは、次のように記述します。クロックとクロック イネーブルはペアで記述する必要があります。つまり、クロックごとに対応するクロック イネーブルが必要です。ブラック ボックスにクロック ポートが 1 つ以上含まれることもありますが、各クロック ポートを駆動するために使用されるクロック ソースは 1 つだけです。異なるのは、クロック イネーブルのレートだけです。
- クロック 名には `clk` (`my_clk_1` など)、クロック イネーブル 名には `ce` (`my_ce_1` など) を含めます。

- クロック イネーブルの名前は、対応するクロックと同じにする必要がありますが、clk の部分は ce にします。たとえば、クロック名が src_clk_1 の場合、クロック イネーブルは src_ce_1 にする必要があります。
- 立ち下がりエッジでトリガされる出力データは使用できません。

ブラック ボックス コンフィギュレーション ウィザード

System Generator には、Verilog または VHDL モジュールを Black Box ブロックに関連付けるためのコンフィギュレーション ウィザードが含まれています。このウィザードは、インポートする VHDL または Verilog モジュールを解析してコンフィギュレーション M 関数を作成し、この M 関数をモデルの Black Box ブロックに関連付けます。コンフィギュレーション M 関数を使用できるかどうかは、インポートする HDL の複雑度によって決まります。ウィザードで検出されなかった詳細を設定するため、コンフィギュレーション M 関数を手動でカスタマイズする必要がある場合があります。コンフィギュレーション M 関数の詳細は、「[ブラック ボックスのコンフィギュレーション M 関数](#)」を参照してください。

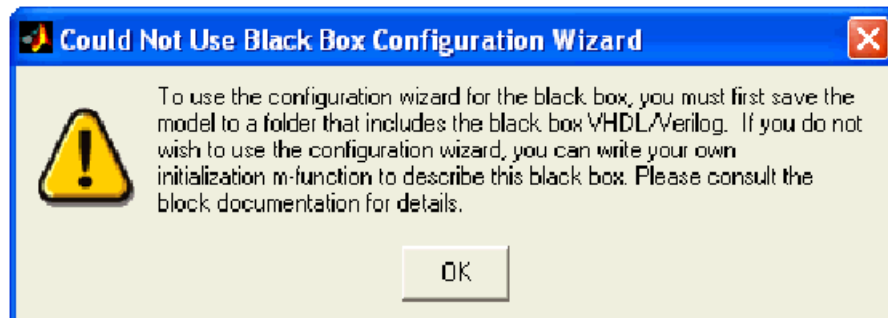
ブラック ボックス コンフィギュレーション ウィザードの使用

Black Box ブロックをモデルに追加すると、ブラック ボックス コンフィギュレーション ウィザードが起動します。

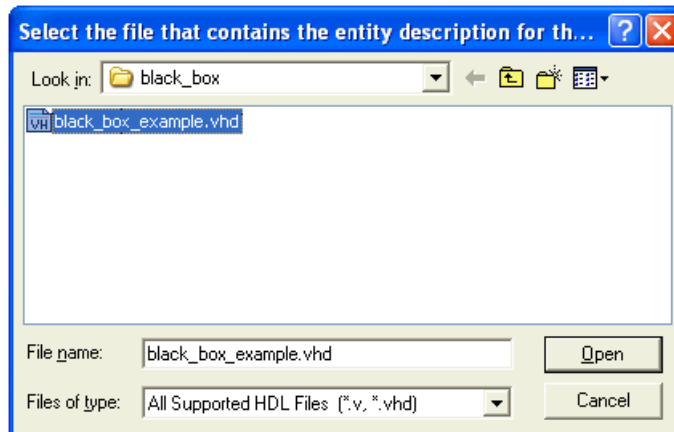
メモ： ブラック ボックス コンフィギュレーション ウィザードを起動する前に、インポートする VHDL または Verilog が [ブラック ボックスの HDL の要件と制限](#) を満たしていることを確認してください。

ウィザードでモジュールが検索されるようにするには、インポートするモジュールと同じディレクトリにモデルを保存する必要があります。

メモ： ウィザードは、ファイルとして保存されているモデルに Black Box ブロックを追加した場合にのみ実行されます。モデルが保存されていない場合は、ウィザードでファイルの検索場所が判断できず、次のような警告メッセージが表示されます。



ウィザードはモデルのディレクトリで VHD および V ファイルを検索し、インポート可能なファイルをダイアログ ボックスに表示します。このダイアログ ボックスの例を、次に示します。



このダイアログ ボックスでインポートするファイルを選択し、[開く] をクリックします。コンフィギュレーション M 関数が作成され、Black Box ブロックに関連付けられます。

メモ：コンフィギュレーション M 関数は、モデルのディレクトリに <module>_config.m (<module> はインポートするモジュールの名前) という名前で保存されます。

ブラック ボックス コンフィギュレーション ウィザードの詳細

ブラック ボックス コンフィギュレーション ウィザードを実行すると、一部の情報はインポートされるモジュールから抽出されますが、手動で設定する必要がある情報もあります。これらの情報は、次のとおりです。

メモ：コンフィギュレーション M 関数には、変更が必要な箇所を示すコメントが追加されています。

- モデルに組み合わせパスがある場合、ブロックの SysgenBlockDescriptor オブジェクトの tagAsCombinational メソッドを呼び出す必要があります。
- ウィザードでは、インポートする最上位エンティティの情報しか抽出できません。通常エンティティにはその他のファイルも付随していますが、これらの各ファイルに対して addFile メソッドを起動してコンフィギュレーション M 関数に追加する必要があります。
- ウィザードではシングル レートのブラック ボックスが作成されるので、ブラック ボックスの各ポートは同じレートで動作します。ほとんどの場合これで問題ありませんが、ポート レートを明示的に設定すると、シミュレーション時間が短縮されます。

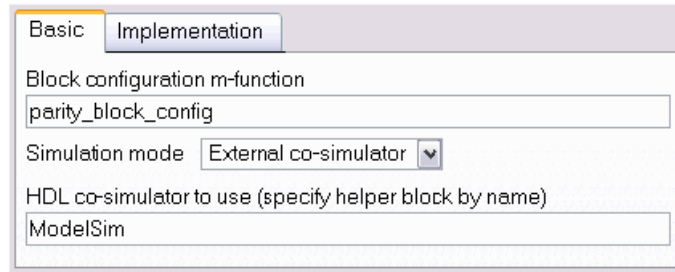
ブラック ボックスのコンフィギュレーション M 関数

インポートしたモジュールは、System Generator で Black Box ブロックとして表されます。インポートしたモジュールの情報は、コンフィギュレーション M 関数によって Black Box ブロックに適用されます。このコンフィギュレーション M 関数は、ブラック ボックスのインターフェイス、インプリメンテーション、シミュレーション動作を定義します。具体的には、コンフィギュレーション M 関数は次の情報を定義します。

- モジュールの最上位エンティティの名前
- VHDL または Verilog の選択
- ポートの説明
- モジュールに必要なジェネリック

- クロック レートおよびサンプリング レート
- モジュールに関連したファイル
- モジュールに組み合わせパスが含まれるかどうか

ブラックボックスに関連付けられるコンフィギュレーション M 関数の名前は、**Black Box** ブロックのパラメータ ダイアログ ボックスでパラメータとして指定されます。次の例では、`parity_block_config.m` という名前です。



コンフィギュレーション M 関数は、オブジェクト ベースのインターフェイスを使用してブラックボックスの情報を指定します。このインターフェイスは、**SysgenBlockDescriptor** および **SysgenPortDescriptor** という 2 つのオブジェクトを定義します。**System Generator** でコンフィギュレーション M 関数が起動されると、**function** にブロック ディスクリプタが渡されます。

```
function sample_block_config(this_block)
```

SysgenBlockDescriptor オブジェクトには、ブラックボックスに関する情報を指定するメソッドが含まれています。ブロック ディスクリプタのポートは、同様にポート ディスクリプタを使用して定義されます。

言語選択

ブラックボックスには、**VHDL** および **Verilog** モジュールをインポートできます。

SysgenBlockDescriptor には、インポートするモジュールのタイプを指定する **setTopLevelLanguage** というメソッドが含まれています。コンフィギュレーション M 関数内では、このメソッドを起動する必要があります。次に、**VHDL** または **Verilog** を選択するコードを示します。

VHDL モジュール:

```
this_block.setTopLevelLanguage('VHDL');
```

Verilog モジュール:

```
this_block.setTopLevelLanguage('Verilog');
```

メモ: ブラックボックス コンフィギュレーション ウィザードでは、コンフィギュレーション M 関数を生成する際に適切な言語が自動的に選択されます。

最上位エンティティの指定

ブラックボックスに関連付ける最上位エンティティの名前を指定する必要があります。

SysgenBlockDescriptor には、最上位エンティティの名前を指定する **setEntityName** メソッドが含まれています。

メモ: エンティティ名は、小文字を使用して指定します。

たとえば次のコードでは、foo という最上位エンティティを指定しています。

```
this_block.setEntityName('foo');
```

メモ : ブラック ボックス コンフィギュレーション ウィザードでは、コンフィギュレーション M 関数を生成する際に最上位エンティティの名前が自動的に設定されます。

ブロック ポートの定義

ブラック ボックスのポート インターフェイスは、ブロックのコンフィギュレーション M 関数で定義されます。ブラック ボックスのポートは、ポート ディスクリプタを使用して定義されます。ポート ディスクリプタを使用すると、ポート幅、データ型、2 進小数点、サンプリング レートなど、ポートのさまざまな属性を設定できます。

新規ポートの追加

ブラック ボックスのポート インターフェイスを定義する際、ブロック ディスクリプタに入力ポートと出力ポートを追加する必要があります。これらのポートは、インポートするモジュール上のポートに対応します。モデルでは、**Black Box** ブロックのポート インターフェイスはブロック ディスクリプタ オブジェクトで宣言されたポート名で決まります。**SysgenBlockDescriptor** には、入力ポートおよび出力ポートを追加するメソッドが含まれています。

入力ポートの追加 :

```
this_block.addSimulinkInport('din');
```

出力ポートの追加 :

```
this_block.addSimulinkOutport('dout');
```

addSimulinkInport および **addSimulinkOutport** メソッドに渡される文字列パラメータは、ポート名を指定します。これらの名前には、インポートしたモジュールのポート名と一致している必要があります。

メモ : ポート名は、小文字を使用して指定します。

双方向ポートの追加 :

```
config_phase = this_block.getConfigPhaseString;  
if (strcmpi(config_phase, 'config_netlist_interface'))  
    this_block.addInoutport('bidi');  
    % Rate and type info should be added here as well  
end
```

双方向ポートはデザインのネットリスト生成でのみサポートされ、**System Generator** のモデルには表示されません。生成された HDL にのみ含まれます。双方向ポートは、HDL を生成する際にのみ追加するようにしてください。双方向ポートを追加するコードには、**if-end** 文を使用して条件を付けるようにしてください。

1 つのメソッド呼び出しで入力ポートと出力ポートの両方を定義することも可能です。

setSimulinkPorts メソッドでは、2 つのパラメータを指定できます。1 つ目のパラメータはブロックの入力ポート名を定義する文字列のセル配列、2 つ目のパラメータはブロックの出力ポート名を定義する文字列のセル配列です。

メモ : ブラック ボックス コンフィギュレーション ウィザードでは、コンフィギュレーション M 関数を生成する際にポート名が自動的に設定されます。

ポート オブジェクトの取得

ブロック ディスクリプタにポートを追加したら、ポートの属性を設定する必要があります。ポートをコンフィギュレーションする前に、ポートのディスクリプタを取得する必要があります。**SysgenBlockDescriptor** には、ポート オブジェクトにアクセスするメソッドが含まれています。たとえば次のメソッドでは、**this_block** ディスクリプタ上の **din** というポートを取得します。

SysgenPortDescriptor オブジェクトにアクセス：

```
din = this_block.port('din');
```

上記のコードでは、オブジェクト **din** が作成され、**port** 関数呼び出しで戻されたディスクリプタに割り当てられます。

SysgenBlockDescriptor には、ポート インデックスを戻す **inport** および **outport** というメソッドも含まれています。ポート インデックスは、ブロック インターフェイスに示された順にポートに付けられたインデックス番号で、1 からブロック上の入力/出力の数までのいずれかです。これらのメソッドは、エラー チェックなどでブロックのポートを確認する際に便利です。

ポート タイプの設定

SysgenPortDescriptor には、個々のポートを設定するメソッドが含まれています。たとえば、**dout** ポートが符号なしの 12 ビットで、2 進小数点の位置が 8 であるとします。このポート タイプを定義するには、次のコードを使用します。

```
dout = this_block.port('dout');  
dout.setWidth(12);  
dout.setBinPt(8);  
dout.makeUnsigned();
```

次のコードも可能です。

```
dout = this_block.port('dout');  
dout.setType('Ufix_12_8');
```

最初のコードでは個々のメソッド呼び出しを使用してポート属性を設定しており、2 番目のコードでは信号タイプを文字列として定義しています。どちらのコードも、機能的には同じです。

ブラックボックスでは、1 ビット ポート (**std_logic** など) またはベクタ (**std_logic_vector(0 downto 0)** など) を使用して宣言した 1 ビット ポートを含む HDL モジュールがサポートされます。**System Generator** では、デフォルトでポートがベクタで宣言されていると想定されます。ディスクリプタの **useHDLVector** を使用すると、このデフォルトを変更できます。このメソッドを **true** に設定すると、**System Generator** でポートがベクタと解釈され、**false** に設定すると、ポートが 1 ビットであると解釈されます。

```
dout.useHDLVector(true); % std_logic_vector  
dout.useHDLVector(false); % std_logic
```

メモ：ブラックボックス コンフィギュレーション ウィザードでは、コンフィギュレーション M 関数を生成する際にポート タイプが自動的に設定されます。

シミュレーション用の双方向ポートの設定

双方向 (**inout**) ポートは、HDL ネットリストの生成でのみサポートされ、**System Generator** のモデルには表示されません。シミュレーションでは、双方向ポートはデフォルトで **X** により駆動されます。ポートにデータ ファイルを関連付けると、この動作を変更できます。双方向ポートを追加するコードが **config_netlist_interface** フェーズでのみ追加されるように、条件文を使用してください。

```

if
  (strcmpi(this_block.getConfigPhaseString,'config_netlist_interface'))
  bidi_port = this_block.port('bidi');
  bidi_port.setGatewayFileName('bidi.dat');
end

```

上記の例では、テキスト ファイル `bidi.dat` がシミュレーション中に使用され、ポートにステイミューラスを供給します。データ ファイルはテキスト形式で、各行に各シミュレーション サイクルでポートを駆動する信号を記述します。たとえば、3 ビットの双方向ポートを 4 サイクル間シミュレーションする場合、データ ファイルは次のようになります。

```

ZZZ
110
011
XXX

```

指定したデータ ファイルが存在しない場合、シミュレーションがエラーとなります。

ポートのサンプリング レートの設定

Black Box ブロックでは、ポートに異なるサンプリング レートを設定できます。デフォルトでは、出力ポートのサンプリング レートは入力ポート (同じサンプリング レートで動作している場合は複数の入力ポート) のサンプリング レートになります。出力ポートのサンプリング レートが入力ポートのサンプリング レートと異なる場合など、ポートのサンプリング レートを明示的に設定する必要があります。

メモ : ブラック ボックスの複数の入力で異なるサンプリング レートが使用されている場合は、各出力ポートのサンプリング レートを指定する必要があります。

SysgenPortDescriptor には、ポートのサンプリング レートを設定する `setRate` メソッドが含まれています。

メモ : `setRate` メソッドに渡されるサンプリング レート パラメータは、そのポートが動作する **Simulink** のサンプリング レートではなく、必要なポートのサンプリング周期と **Simulink** システムクロック周期 (**System Generator** トークンのパラメータ ダイアログ ボックスで指定) との比を指定する正の整数です。

Simulink のシステム周期が 2s と定義されたモデルがあり、次のように `setRate` メソッドを使用して `dout` ポートにレート 3 を設定するとします。

```

dout.setRate(3);

```

このレート 3 は、**Simulink** の 3 システム周期ごとに `dout` ポートにサンプルが生成されることを意味します。**Simulink** のシステム周期は 2s なので、ポートのサンプリング レートは $3 \times 2 = 6\text{s}$ となります。

メモ : ポートがサンプリングされない定数である場合は、コンフィギュレーション **M** 関数で **SysgenPortDescriptor** の `setConstant` を使用して定義できます。`setRate` メソッドに `Inf` を渡すことにより定数を定義することも可能です。

ダイナミック出力ポート

ブラック ボックスでは、動的に出力ポートのタイプおよびレートを変更できます。たとえば、入力ポートの幅に応じて出力ポートの幅を設定する必要があります。 **SysgenPortDescriptor** には、ポート設定を判断できるメンバー変数が含まれています。出力ポートのタイプまたはレートを、ブロックの入力ポートのメンバー変数を調べることで設定できます。

たとえば次の例のように、ポート `din` の幅とレートを取得できます。

```
input_width = this_block.port('din').width;  
input_rate  = this_block.port('din').rate;
```

メモ: ブラックボックスのコンフィギュレーション M 関数は、モデルがコンパイルされたときに何度か実行されます。データ型およびレートがブラックボックスに伝搬される前に実行されることがあります。

`SysgenBlockDescriptor` には、ポートタイプおよびレートがブロックに伝搬されたかどうかを調べる `inputTypesKnown` および `inputRatesKnown` というブールメンバー変数が含まれています。入力ポートに応じてダイナミック出力ポートタイプまたはレートを設定する場合は、`inputTypesKnown` と `inputRatesKnown` の値をチェックする条件文内にコンフィギュレーション呼び出しをネストする必要があります。

次のコードは、ダイナミック出力ポート `dout` の幅を入力ポート `din` の幅と同じに設定しています。

```
if (this_block.inputTypesKnown)  
    dout.setWidth(this_block.port('din').width);  
end
```

ダイナミックレートも、同様に設定できます。次のコードは、出力ポート `dout` のサンプリングレートを入力ポート `din` のサンプリングレートの 2 倍に設定しています。

```
if (this_block.inputRatesKnown)  
    dout.setRate(this_block.port('din').rate*2);  
end
```

ブラックボックスのクロック

マルチレート モジュールをインポートするには、コンフィギュレーション M 関数でモジュールのクロックに関する情報を指定する必要があります。**System Generator** では、クロックとクロックイネーブルをほかのポートとは異なる方法で処理します。インポートしたモジュールのクロックポートには、クロックイネーブルポートが必要です。つまり、クロックとクロックイネーブルはペアとして定義し、インポートしたモジュール内でペアとして存在する必要があります。これは、シングルレート デザインおよびマルチレート デザインの両方に適用されます。

メモ: クロックおよびクロックイネーブルはペアで存在する必要がありますが、**System Generator** ではインポートしたモジュールのすべてのクロックポートが **FPGA システムクロック** で駆動され、クロックイネーブルポートは **FPGA システムクロック** から生成されたクロックイネーブル信号で駆動されます。

`SysgenBlockDescriptor` には、ブラックボックスのクロックおよびクロックイネーブルの情報を定義する `addClkCEPair` メソッドが含まれています。このメソッドでは、3 つのパラメータを指定できます。1 つ目のパラメータはクロックポートの名前 (モジュールで使われる名前)、2 つ目のパラメータはクロックイネーブルポートの名前 (モジュールで使われる名前) を定義します。

クロックとクロックイネーブルのペアのポート名は、次の命名規則に従って付ける必要があります。

- クロックポート名には、`clk` を含めます。
- クロックイネーブルポート名には、`ce` を含めます。
- クロックとクロックイネーブルのペアでは、文字列の `clk` および `ce` 以外の部分を同じにする必要があります (`my_clk_1` と `my_ce_1` など)。

3 つ目のパラメータは、クロックとクロックイネーブルのレート関係を定義します。このレートパラメータでは、**Simulink** のサンプリングレートではなく、クロックサンプリング周期とクロックイネーブルサンプリング周期の比を整数値で指定します。

たとえば、`ce_3` というクロック イネーブル ポートのサンプリング周期をシステム クロック周期の 3 倍に設定する場合、次のコードを使用します。

```
addClkCEPair('clk_3','ce_3',3);
```

System Generator でブラック ボックスをハードウェアにコンパイルすると、適切なクロック イネーブル信号が生成され、適切なクロック イネーブル ポートに配線されます。

組み合わせパス

インポートするモジュールに組み合わせパスがある場合 (入力の変化がクロック イベントなしで出力ポートに影響する)、コンフィギュレーション **M** 関数で指定する必要があります。

SysgenBlockDescriptor オブジェクトには、モジュールに組み合わせパスがあることを示す `tagAsCombinational` メソッドが含まれています。コンフィギュレーション **M** 関数内では、このメソッドを次のように起動する必要があります。

```
this_block.tagAsCombinational;
```

VHDL ジェネリックおよび Verilog パラメータの指定

System Generator でモデルを HDL にコンパイルする際に、モジュールに渡すジェネリックのリストを指定できます。これらのジェネリックに割り当てる値は、マスク パラメータおよび伝搬されたポート情報 (ポート 幅、タイプ、レート など) から抽出されます。このように柔軟にジェネリックを割り当てることのできるため、ブラック ボックスの周辺環境に基づいてカスタマイズされるモジュールがサポートされます。

`addGeneric` メソッドを使用すると、デザインをハードウェアにコンパイルしたときにモジュールに渡す必要のあるジェネリックを定義できます。次のコードでは、VHDL の **Integer** ジェネリック `dout_width` を 12 に設定しています。

```
addGeneric('dout_width','Integer','12');
```

ジェネリック値は、伝搬された入力ポートの情報 (ダイナミック出力ポートの幅を指定するジェネリックなど) に基づいて設定することも可能です。

ブラック ボックスのコンフィギュレーション **M** 関数はモデルがコンパイルされる際に複数回起動されますが、データ型 (またはレート) がブラック ボックスに伝搬される前に起動されることがあります。入力ポート のタイプまたはレート に基づいてジェネリック値を設定する場合は、`inputTypesKnown` または `inputRatesKnown` の値をチェックする条件文内に `addGeneric` 呼び出しをネストする必要があります。たとえば次のように、`dout` ポートの幅を `din` の値に基づいて設定できます。

```
if (this_block.inputTypesKnown)
    % set generics that depend on input port types
    this_block.addGeneric('dout_width', ...
        this_block.port('din').width);
end
```

ジェネリック値は、ブラック ボックスに関連付けられたマスク パラメータに基づいて設定できます。**SysgenBlockDescriptor** には、**Simulink** でのブラック ボックス名を表す文字列である `blockName` メンバー変数が含まれます。この変数を使用して、特定のコンフィギュレーション **M** 関数に関連付けられたブラック ボックスにアクセスできます。たとえば、ブラック ボックスで `init_value` というパラメータを定義する場合、次のコードを使用できます。

```
simulink_block = this_block.blockName;
init_value = get_param(simulink_block,'init_value');
this_block.addGeneric('init_value', 'String', init_value);
```


メモ：ブラック ボックスに独自のパラメータを追加するには (ジェネリック値を指定する値など)、次の手順に従います。

- Black Box ブロックを Simulink ライブラリまたはモデルにコピーします。
- Black Box ブロックのリンクを解除します。
- Black Box ブロックのダイアログ ボックスにパラメータを追加します。

エラーのチェック

通常、ブラック ボックスのポート タイプ、レート、マスク パラメータでエラー チェックを実行する必要があります。SysgenBlockDescriptor には、表示されるエラー メッセージを指定する setError メソッドが含まれています。setError に渡される文字列パラメータが、ユーザーに表示されるエラー メッセージです。

ブラック ボックスの API

SysgenBlockDescriptor メンバー変数

データ型	メンバー	説明
文字列	entityName	エンティティまたはモジュールの名前
文字列	blockName	Black Box ブロックの名前
整数	numSimulinkInports	ブラック ボックス上の入力ポートの数
整数	numSimulinkOutputs	ブラック ボックス上の出力ポートの数
ブール代数	inputTypesKnown	すべての入力タイプが定義されている場合は true、されていない場合は false
ブール代数	inputRatesKnown	すべての入力レートが定義されている場合は true、されていない場合は false
倍精度値の配列	inputRates	入力ポート (inport(indx) としてインデックス) のサンプリング周期の配列。サンプリング周期は、System Generator トークンのパラメータ ダイアログ ボックスで指定された Simulink のシステム周期の倍数を整数で指定します。
ブール代数	error	エラーが検出された場合は true、されなかった場合は false
文字列のセル配列	errorMessages	ブロックのすべてのエラー メッセージ

SysgenBlockDescriptor のメソッド

メソッド	説明
setTopLevelLanguage(language)	ブラック ボックスの最上位エンティティ (またはモジュール) の言語を指定します。VHDL または Verilog を指定できます。
setEntityName(name)	エンティティまたはモジュールの名前を設定します。
addSimulinkInport(pname)	ブラック ボックスに入力ポートを追加します。pname はポートの名前です。
addSimulinkOutport(pname)	ブラック ボックスに出力ポートを追加します。pname はポート名を指定します。
setSimulinkPorts(in,out)	ブラック ボックスに入力ポートと出力ポートを追加します。in は入力ポート名を、out は出力ポート名をセル配列で指定します。
addInoutport(pname)	ブラック ボックスに双方向ポートを追加します。pname はポート名を指定します。双方向ポートは、コンフィギュレーションの config_netlist_interface フェーズでのみ追加可能です。
tagAsCombinational()	ブロックに組み合わせパスがあることを示します。
addClkCEPair(clkPname, cePname, rate)	ブロックのクロック ポートとクロック イネーブルポートのペアを定義します。clkPname はクロック名、cePname はクロック イネーブル名、rate (倍精度値) はポート ペアの動作レートを指定します。rate は正の整数で指定する必要があります。クロック名には clk を、クロック イネーブル名には ce を含める必要があります。クロック イネーブルの名前は、対応するクロックと同じにする必要がありますが、clk の部分は ce にします。
port(name)	指定の名前に一致する SysgenPortDescriptor を返します。
inport(indx)	指定の入力ポートを説明する SysgenPortDescriptor を返します。index はポートのインデックス (1 ~ numInputPorts) を指定します。
outport(indx)	指定の出力ポートを説明する SysgenPortDescriptor を返します。index はポートのインデックス (1 ~ numOutputPorts) を指定します。
addGeneric(identifier, value)	ブロックのジェネリック (Verilog の場合はパラメータ) を指定します。identifier はジェネリック名を指定し、value はジェネリックの値を倍精度値または文字列で指定します。ジェネリックのデータ型は、value のデータ型から判断されます。value が倍精度の整数 (40 など) の場合は integer に、倍精度の非整数の場合は real に設定されます。value が 0 と 1 のみを含む文字列 ('0101' など) の場合は bit_vector に、その他の文字列の場合は string に設定されます。

メソッド	説明
<code>addGeneric(identifier, type, value)</code>	ブロックのジェネリック (Verilog の場合はパラメータ) の名前、データ型、値を指定します。3 つの引数はすべて文字列で、 identifier は名前、 type はデータ型、 value は値を指定します。
<code>addFile(fn)</code>	ブラック ボックスに関連付けるファイル リストにファイルを追加します。 fn はファイル名を指定します。通常ブラック ボックスには HDL ファイルが関連付けられますが、どんなファイルでも追加できます。VHDL ファイルの拡張子は .vhd 、Verilog ファイルの拡張子は .v である必要があります。ファイルの追加順は保持され、この順序で HDL ファイルがコンパイルされます。相対パス名または絶対パス名のどちらでも使用できます。相対パス名は、デザインの MDL ファイルまたはライブラリ MDL ファイルの場所を基準に解釈されます。
<code>getDeviceFamilyName()</code>	ブラック ボックスに対応する FPGA デバイスの名前を取得します。
<code>getConfigPhaseString</code>	現在のコンフィギュレーション フェーズを文字列として返します。有効な戻り値には、 config_interface 、 config_rate_and_type 、 config_post_rate_and_type 、 config_simulation 、 config_netlist_interface 、 config_netlist などがあります。
<code>setSimulatorCompilationScript (script)</code>	ブラック ボックスで生成されたデフォルトの HDL 協調シミュレーション コンパイル スクリプトの代わりに、 script で指定したスクリプトを使用します。このメソッドを使用すると、たとえばブラック ボックスの HDL が変更されていない場合に、シミュレーションの再実行でコンパイルをスキップできます。
<code>setError(message)</code>	エラーが発生したときに表示するエラー メッセージを設定します。 message は表示するエラー メッセージを指定します。

SysgenPortDescriptor メンバー変数

データ型	メンバー	説明
文字列	name	ポート名
整数	simulinkPortNumber	Simulink でのポートのインデックス。インデックスは 1 から開始します。
ブール代数	typeKnown	ポートのデータ型がわかっている場合は true、わかっていない場合は false
文字列	type	ポートのデータ型 (UFix_<n>_、Fix_<n>_、または Bool)
ブール代数	isBool	ポートのデータ型が Bool の場合は true、Bool でない場合は false
ブール代数	isSigned	ポートのデータ型が signed の場合は true、signed でない場合は false
ブール代数	isConstant	ポートのデータ型が constant の場合は true、constant でない場合は false
整数	width	ポート幅
整数	binpt	2 進小数点の位置 (0 ~ width の整数値)
ブール代数	rateKnown	レートがわかっている場合は true、わかっていない場合は false
倍精度値	rate	ポートのサンプリング レート。MATLAB 倍精度値で表現した正の整数です。Inf の場合は、ポート出力が定数であることを示します。

SysgenPortDescriptor のメソッド

メソッド	説明
setName(name)	ポートの HDL 名を指定します。
setSimulinkPortNumber(num)	Simulink でポートに関連付けるインデックスを設定します。num はインデックスを指定します。インデックスは 1 から開始します。
setType(typeName)	ポートのデータ型を設定します。Bool、UFix_<n>_、Fix_<n>_、signed、または unsigned のいずれかに設定します。signed および unsigned では、幅と 2 進小数点位置は変更されません。
setWidth(w)	ポート幅を w に設定します。
setBinpt(bp)	ポートの 2 進小数点位置を bp に設定します。
makeBool()	ポートのデータ型を Bool にします。
makeSigned()	ポートのデータ型を signed にします。
makeUnsigned()	ポートのデータ型を unsigned にします。
setConstant()	ポートのデータ型を constant にします。
setGatewayFileName(filename)	ポートのシミュレーションおよびテストベンチ生成に使用するデータ ファイル名を指定します。この関数は、双方向ポートのシミュレーション用に手動で作成したデータ ファイルを使用するために使用します。入力ポートまたは出力ポートにこのパラメータを設定するのは無効であり、無視されます。
setRate(rate)	ポート のレート を設定します。rate は MATLAB 倍精度の正の整数で指定するか、定数の場合は Inf にします。
useHDLVector(s)	1 ビット ポートが 1 ビット (std_logic など) で表されているか、ベクタ (std_logic_vector(0 downto 0) など) で表されているかを示します。
HDLTypeIsVector()	1 ビット ポートの表現を std_logic_vector(0 downto 0) に設定します。

HDL 協調シミュレーション

概要

このセクションでは、ザイリンクス ブロック、HDL モジュール、Simulink ブロック ダイアグラムを含むデザインに対して、混合言語フローを実行する方法を説明します。

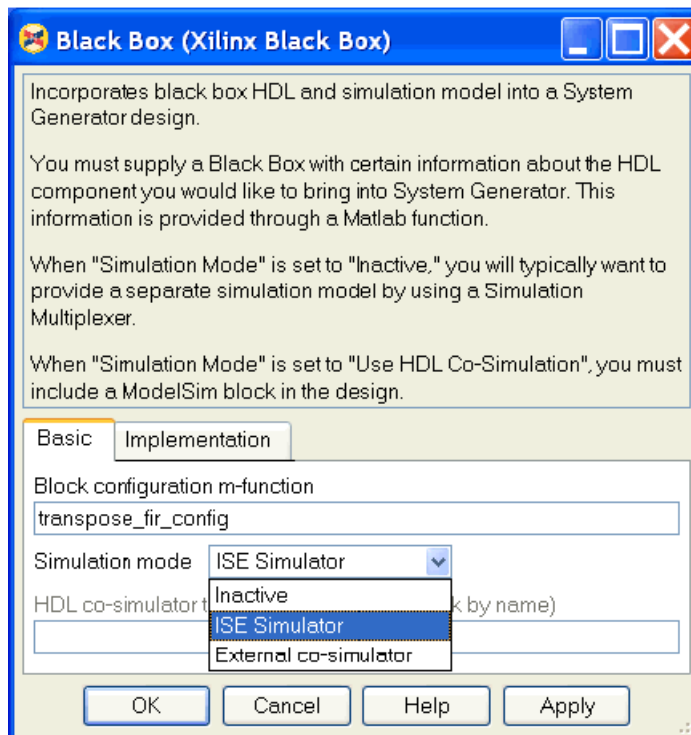
System Generator は、ブラック ボックスをシミュレーションする際、自動的に HDL シミュレータを起動して必要な追加の HDL (HDL テストベンチ) を生成し、HDL をコンパイルし、シミュレーション イベントをスケジュールし、Simulink と HDL シミュレータの間のデータ交換を制御します。これを HDL 協調シミュレーションと呼びます。

HDL シミュレータの設定

ブラック ボックスの HDL は、Simulink で System Generator インターフェイスを使用して ISim または ModelSim にアクセスすることにより協調シミュレーションできます。

ISim

協調シミュレーションに ISim を使用するには、次の図に示すように、Black Box ブロックのパラメータ ダイアログ ボックスで [Simulation mode] を [ISE Simulator] に設定します。これで、HDL 協調シミュレーションを実行できます。



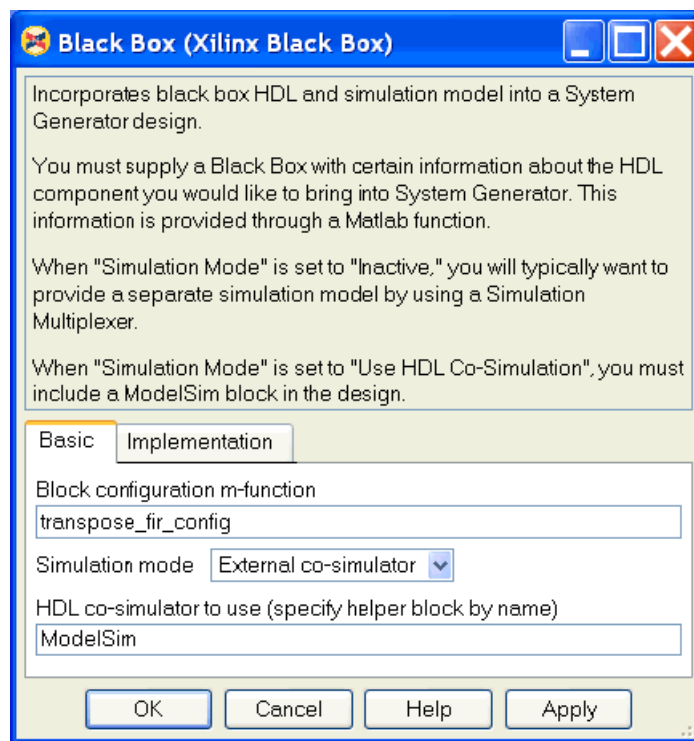
ModelSim シミュレータ

ModelSim シミュレータを使用するには、[Xilinx Blockset] → [Tools] ライブラリにある ModelSim ブロックを Simulink ダイアグラムに追加する必要があります。



ModelSim で協調シミュレーションを実行する各 Black Box ブロックのパラメータ ダイアログボックスで、ModelSim を使用するよう設定します。次の 2 つのパラメータを設定します。

1. [Simulation mode] を [External co-simulator] に設定します。
2. [HDL Co-Simulator to use] に ModelSim ブロックの名前 (ModelSim など) を入力します。



ModelSim ブロックのパラメータ ダイアログボックスには、ModelSim でのシミュレーションセッションを制御するオプションが含まれています。詳細は、[ModelSim](#) ブロックのヘルプを参照してください。これで、HDL 協調シミュレーションを実行できます。

複数のブラック ボックスの協調シミュレーション

System Generator では、同じ ModelSim 協調シミュレーション セッションで複数の Black Box ブロックをシミュレーションできます。つまり、複数の Black Box ブロックで同じ ModelSim ブロックを使用するよう設定できます。この場合、すべての Black Box ブロックの HDL コンポーネントが 1 つの最上位協調シミュレーション コンポーネントに統合されます。これは自動的に行われます。このようなシミュレーションでは、1 つの ModelSim シミュレーション ライセンスで複数の Black Box ブロックに対して協調シミュレーションを実行できます。

複数の Black Box ブロックに対して同時に協調シミュレーションを実行する方法は、「[複数のブラック ボックスの同時シミュレーション](#)」を参照してください。

ISim でも、複数の Black Box ブロックに対して協調シミュレーションを実行できます。これには、各 Black Box ブロックで [Simulation mode] を [ISE Simulator] に設定します。

ブラック ボックスの例

[ブラック ボックスの例 1 : ブラック ボックスの HDL 要件を満たした CORE Generator モジュールのインポート](#)

System Generator のブラック ボックス コンフィギュレーション ウィザードを使用した例を示します。

[ブラック ボックスの例 2 : ブラック ボックスの HDL 要件を満たす VHDL ラップが必要な CORE Generator モジュールのインポート](#)

VHDL コア ラップが必要な例を示します。シミュレーション問題の解決方法も示します。

[ブラック ボックスの例 3 : VHDL モジュールのインポート](#)

Black Box ブロックを使用して VHDL を System Generator デザインにインポートし、ModelSim を使用して協調シミュレーションを実行する方法を示します。

[ブラック ボックスの例 4 : Verilog モジュールのインポート](#)

Verilog ブラック ボックスを System Generator で使用し、ModelSim を使用して協調シミュレーションする方法を示します。

[ブラック ボックスの例 5 : ダイナミックブラック ボックス](#)

入力幅の変化に応じて動的に調整される転置型 FIR フィルタ ブラック ボックスを使用したダイナミック ブラック ボックスを示します。

[ブラック ボックスの例 6 : 複数のブラック ボックスの同時シミュレーション](#)

1 つの ModelSim ライセンスを使用し、複数の Black Box ブロックに対して同時に協調シミュレーションを実行する方法を示します。

[ブラック ボックスの例 7 : ModelSim を使用したアドバンス ブラック ボックス](#)

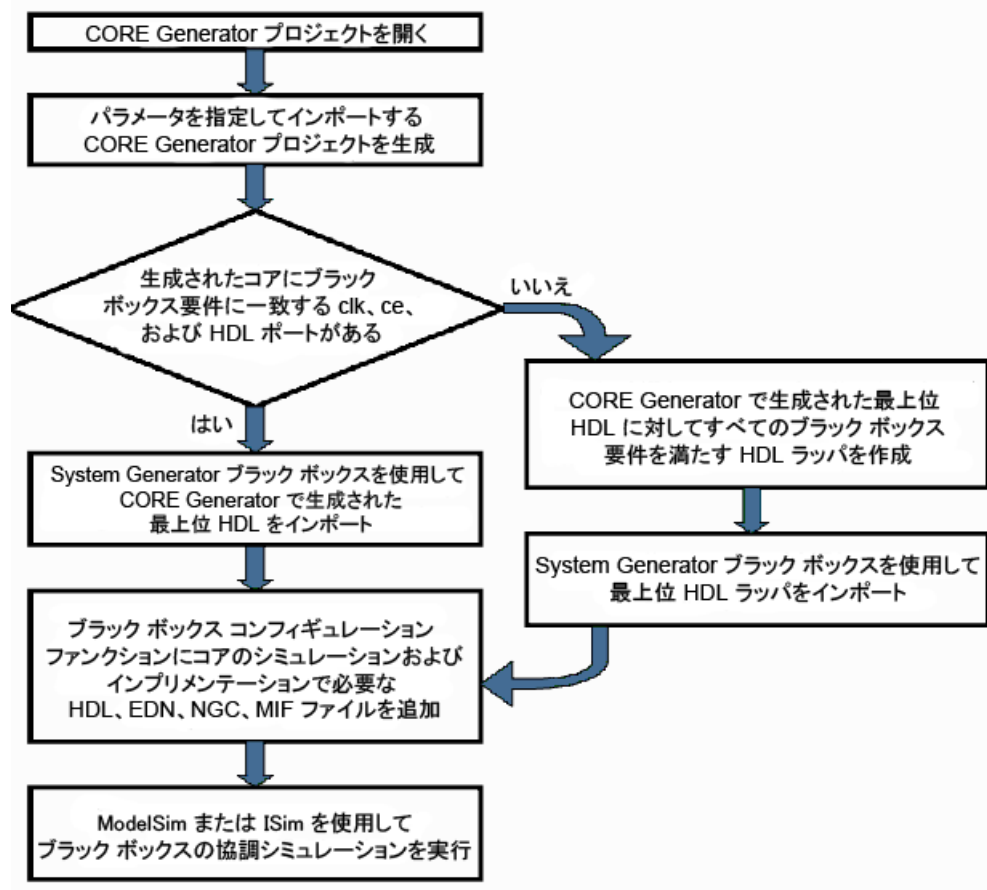
ダイナミック ポート インターフェイスを含む Black Box ブロックを設計し、マスク パラメータを使用してブラック ボックスをコンフィギュレーションする方法を示します。入力ポートのデータ型に基づいてジェネリック値を割り当て、後で再利用するために Black Box ブロックを Simulink ライブラリに保存する方法も示します。ModelSim HDL 協調シミュレーション用のカスタム スクリプトの指定方法についても説明します。

ブラック ボックスの例 8: 暗号化された VHDL ファイルのインポート、シミュレーション、エクスポート

デザインを暗号化された VHDL ファイルとして Black Box ブロックにインポートし、デザインをシミュレーションして、VHDL を個別の暗号化されたファイルとしてエクスポートする方法を示します。

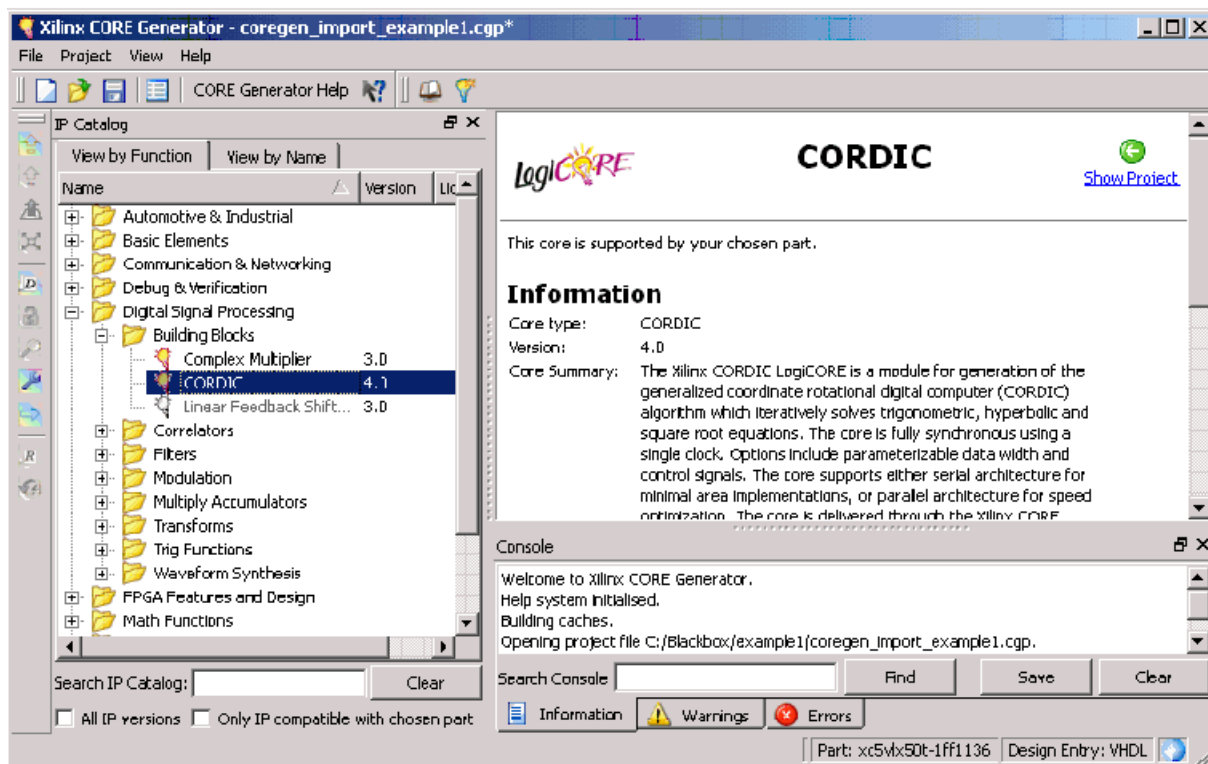
CORE Generator モジュールのインポート

このセクションでは、CORE Generator™ モジュールをブラック ボックスとして System Generator にインポートする 2 つの方法を示します。1 つ目の方法では、[ブラック ボックスの HDL の要件と制限](#)を満たしているブラック ボックスをインポートする方法を示します。2 つ目の方法では、VHDL ラッパを記述して CORE Generator モジュールをブラック ボックスとしてインポートする方法を示します。次に、CORE Generator モジュールのインポート フローを示します。

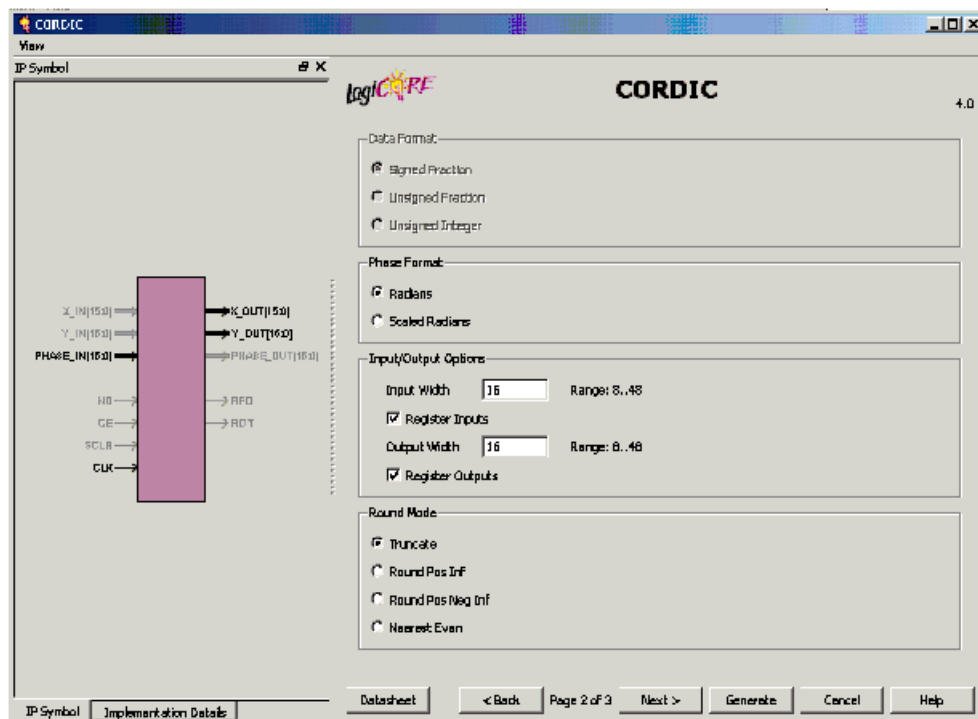
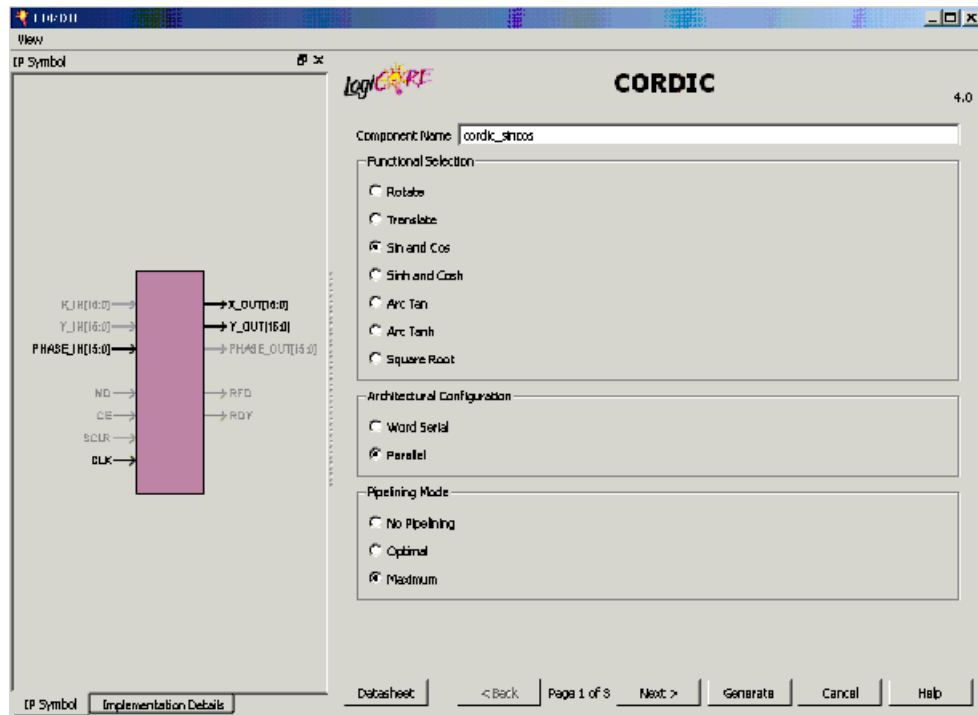


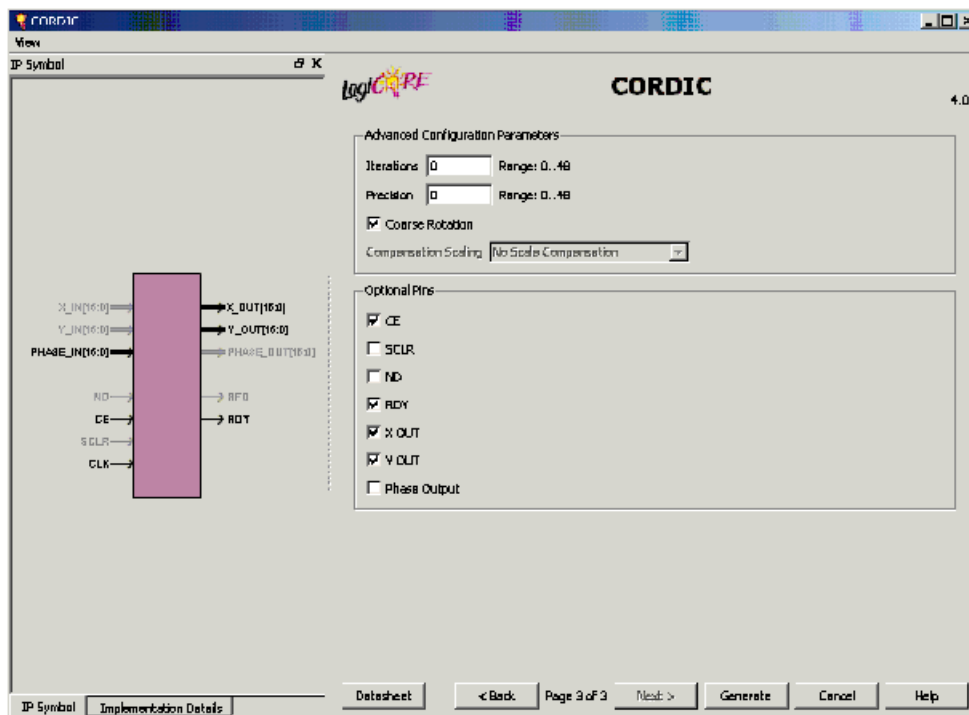
ブラック ボックスの例 1 : ブラック ボックスの HDL 要件を満たした CORE Generator モジュールのインポート

1. CORE Generator を起動し、次の CORE Generator プロジェクト ファイルを開きます。
`<path_to_sysgen>\examples\coregen_import\example1\coregen_import_example1.cgp`
2. [View by Function] タブで [Digital Signal Processing] → [Building Blocks] の下にある [CORDIC 4.0] をダブルクリックし、カスタマイズ ウィンドウを開きます。



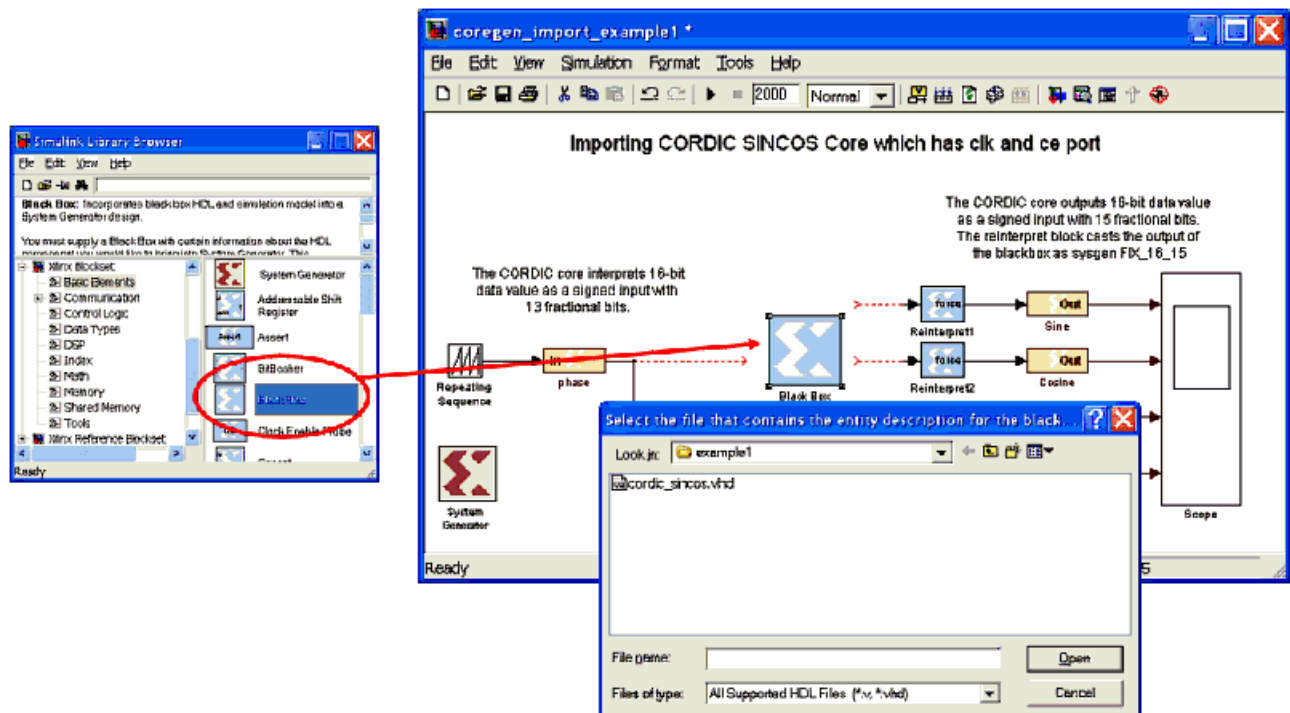
3. 次の図に示すように、[Component Name] を cordic_sincos、[Functional Selection] で [Sin and Cos] をオン、残りのオプションはデフォルト値に設定します。



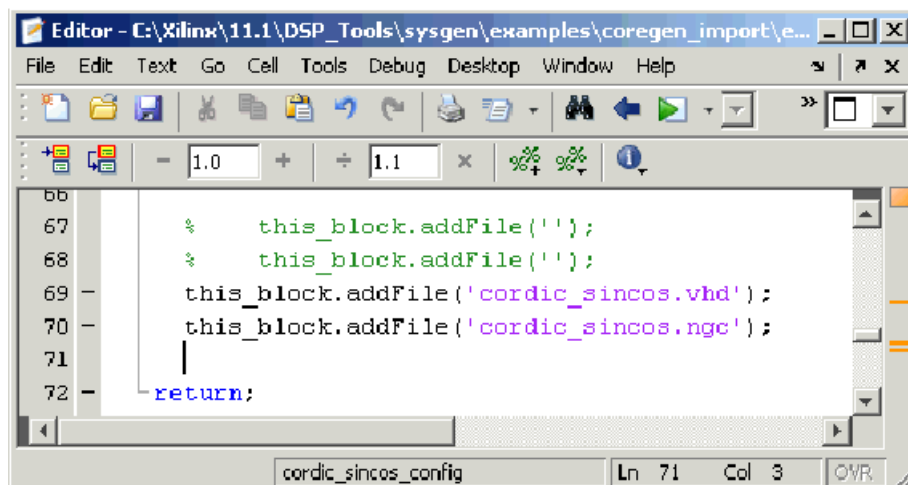


4. [Generate] をクリックします。CORE Generator で次のファイルが生成されます。
 - ◆ cordic_sincos.ngc: インプリメンテーション ネットリスト
 - ◆ cordic_sincos.vhd: ビヘイビアシミュレーション用の VHDL ラップ
 - ◆ cordic_sincos.vho: コア インスタンス化テンプレート
 - ◆ cordic_sincos.xco: コアの生成に選択されたパラメータ
5. Simulink を起動し、デザイン ファイル <path_to_sysgen>\examples\coregen_import\example1\coregen_import_example1.mdl を開きます。

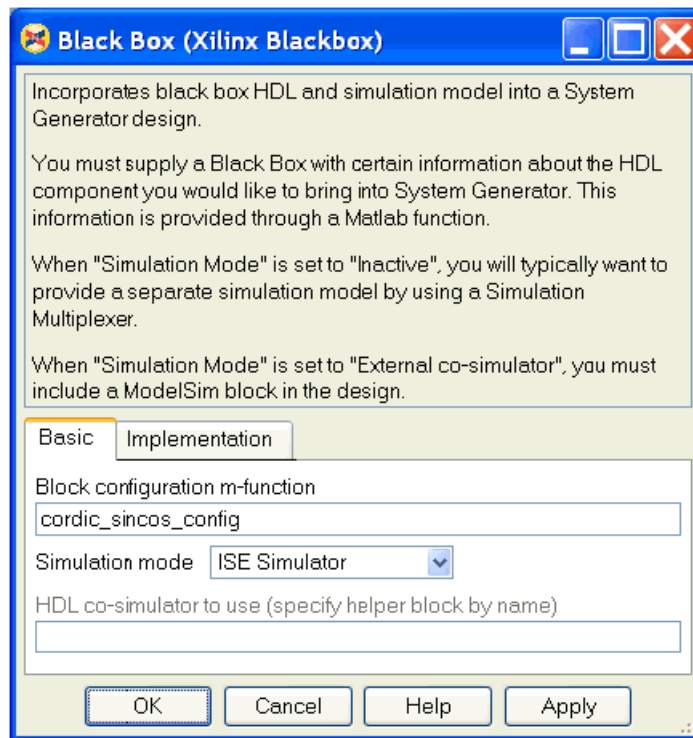
6. [Xilinx Blockset] → [Basic Elements] ライブラリから Black Box ブロックを coregen_import_example1.mdl にドラッグします。最上位 HDL ファイルとして cordic_sincos.vhd を選択し、[Open] をクリックします。



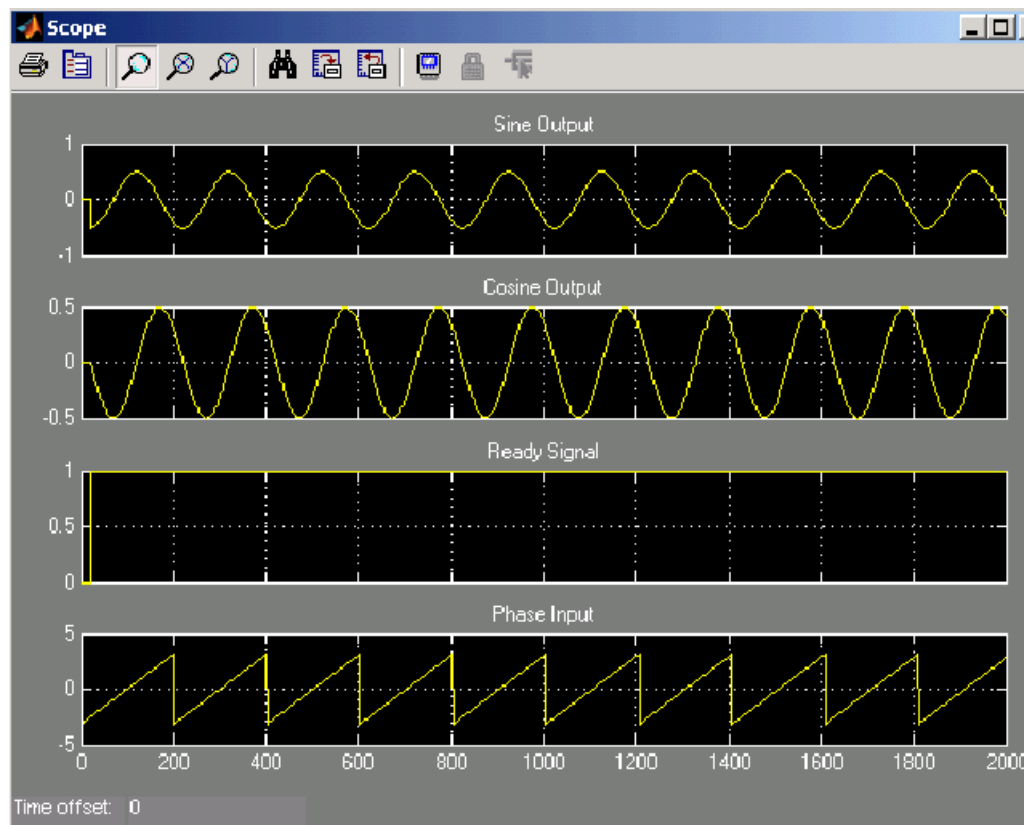
7. Black Box ブロックの入力ポートおよび出力ポートをワイヤに接続します。
8. cordic_sincos_config.m を開き、ファイルのリストに次の図に示すように EDIF ネットリストを追加します。デザインの System Generator ネットリストが作成される際に、このファイルが含まれます。



9. Black Box ブロックのパラメータ ダイアログ ボックスを開き、[Simulation mode] を [ISE Simulator] に設定します。

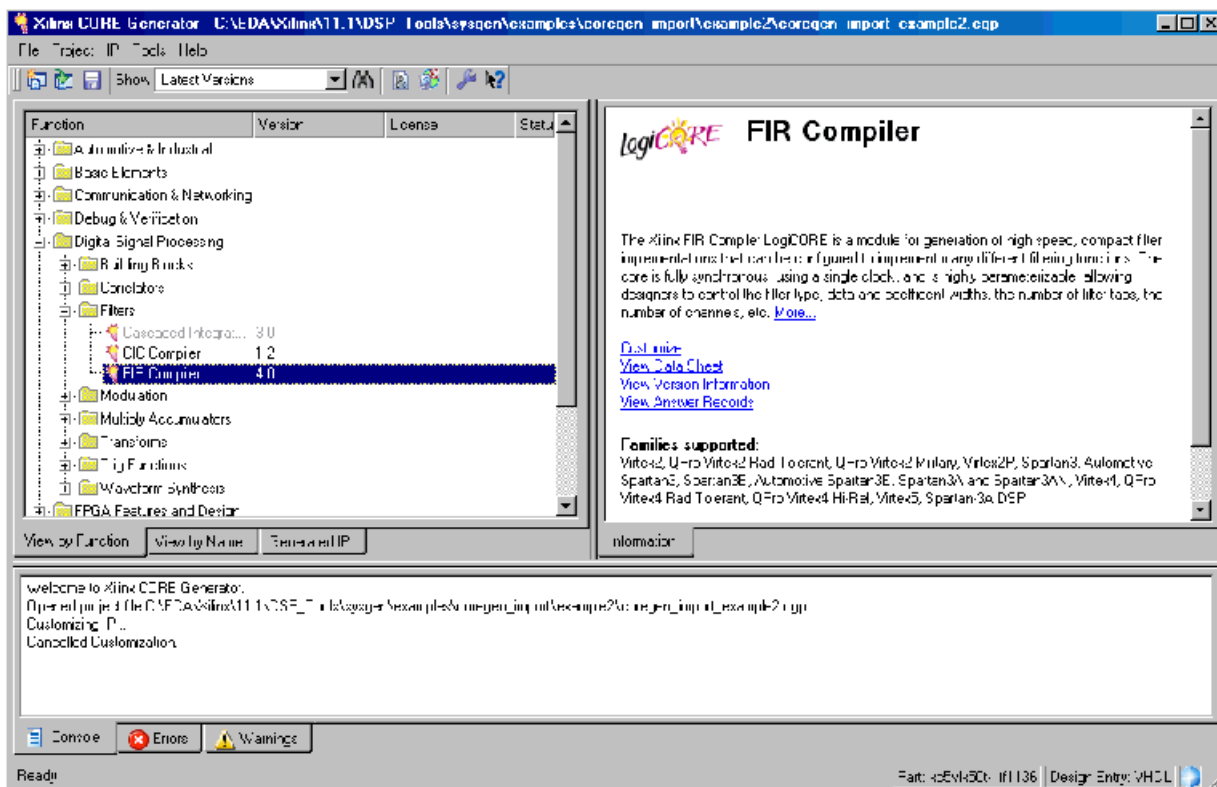


10. Simulink モデルで [シミュレーションの開始] をクリックし、**Simulator** を使用して CORDIC コアに対して協調シミュレーションを実行します。次のようなシミュレーション結果が表示されます。



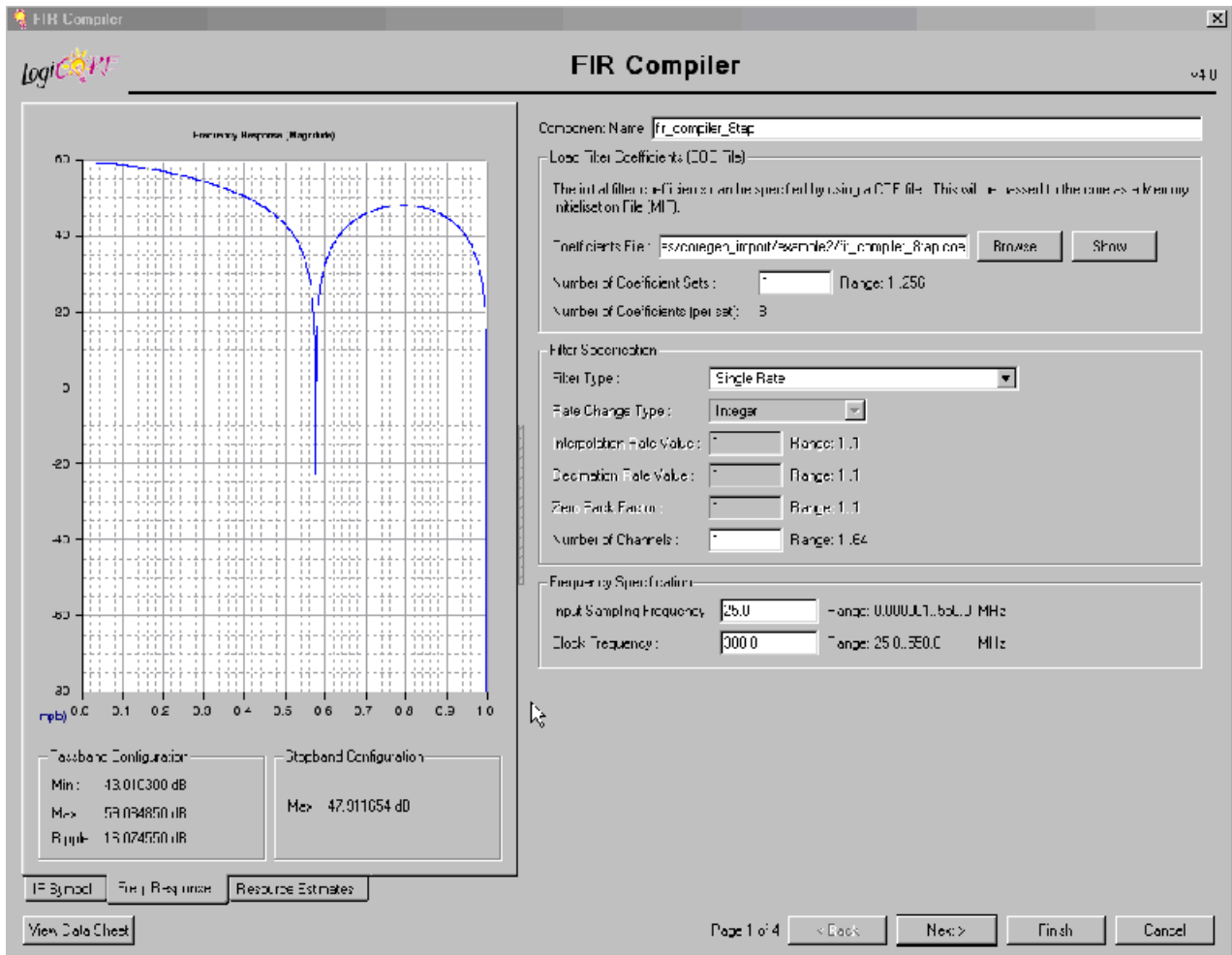
ブラック ボックスの例 2 : ブラック ボックスの HDL 要件を満たす VHDL ラップが必要な CORE Generator モジュールのインポート

1. CORE Generator を起動し、次の CORE Generator プロジェクト ファイルを開きます。
`<path_to_sysgen>\examples\coregen_import\example2\coregen_import_example2.cgp`
2. [View by Function] タブで [Digital Signal Processing] → [Filters] の下にある [FIR Compiler] をダブルクリックし、カスタマイズ ウィンドウを開きます。



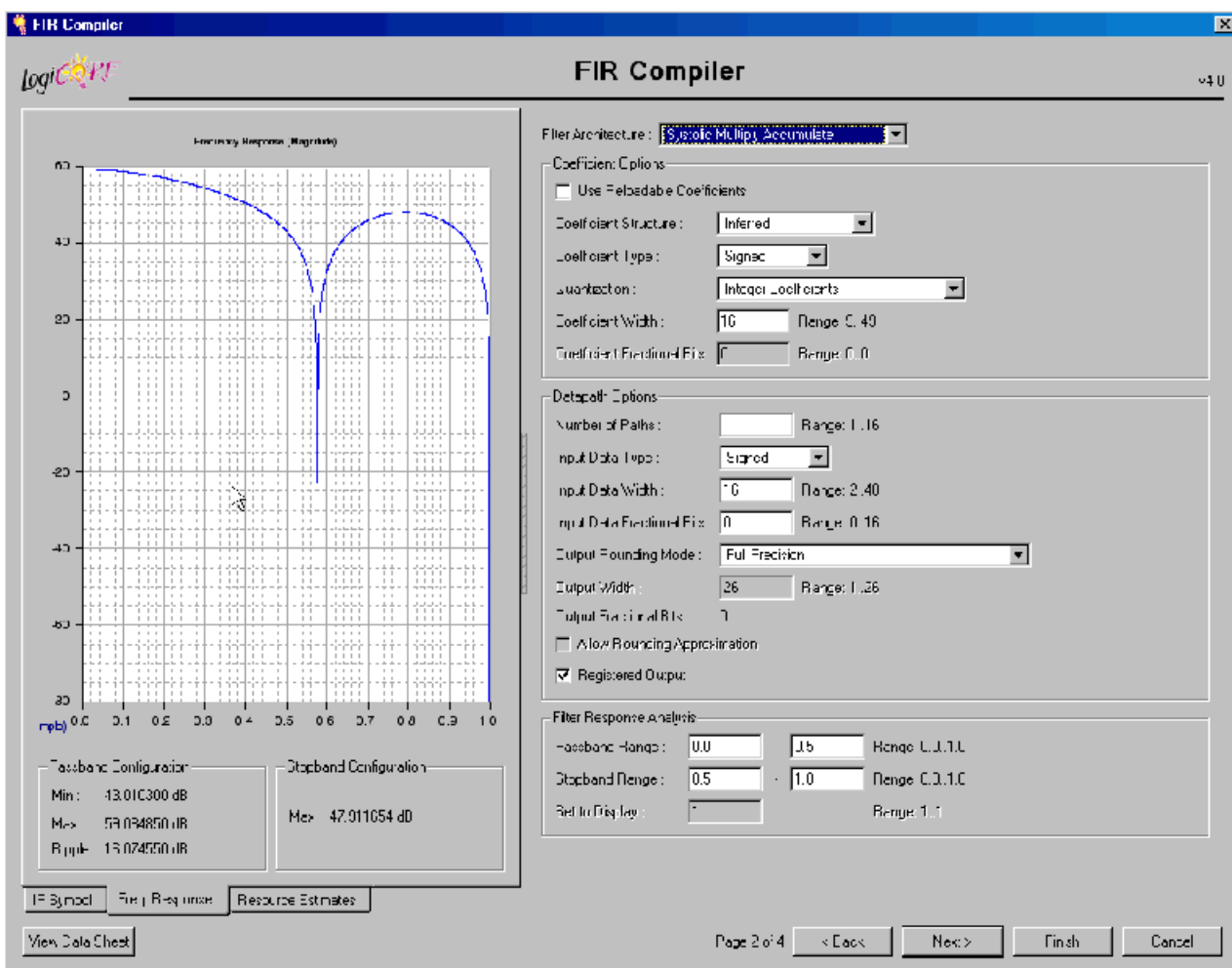
3. 次のオプションを設定し、[Generate] をクリックして FIR Compiler 4.0 コアを生成します。

- ◆ [Component Name] : fir_compiler_8tap
- ◆ [Coefficient File] : <path_to_sysgen>\examples\coregen_import\example2
ディレクトリにある fir_compiler_8tap.coe
- ◆ [Input Sampling Frequency] : 25
- ◆ [Clock Frequency] : 300



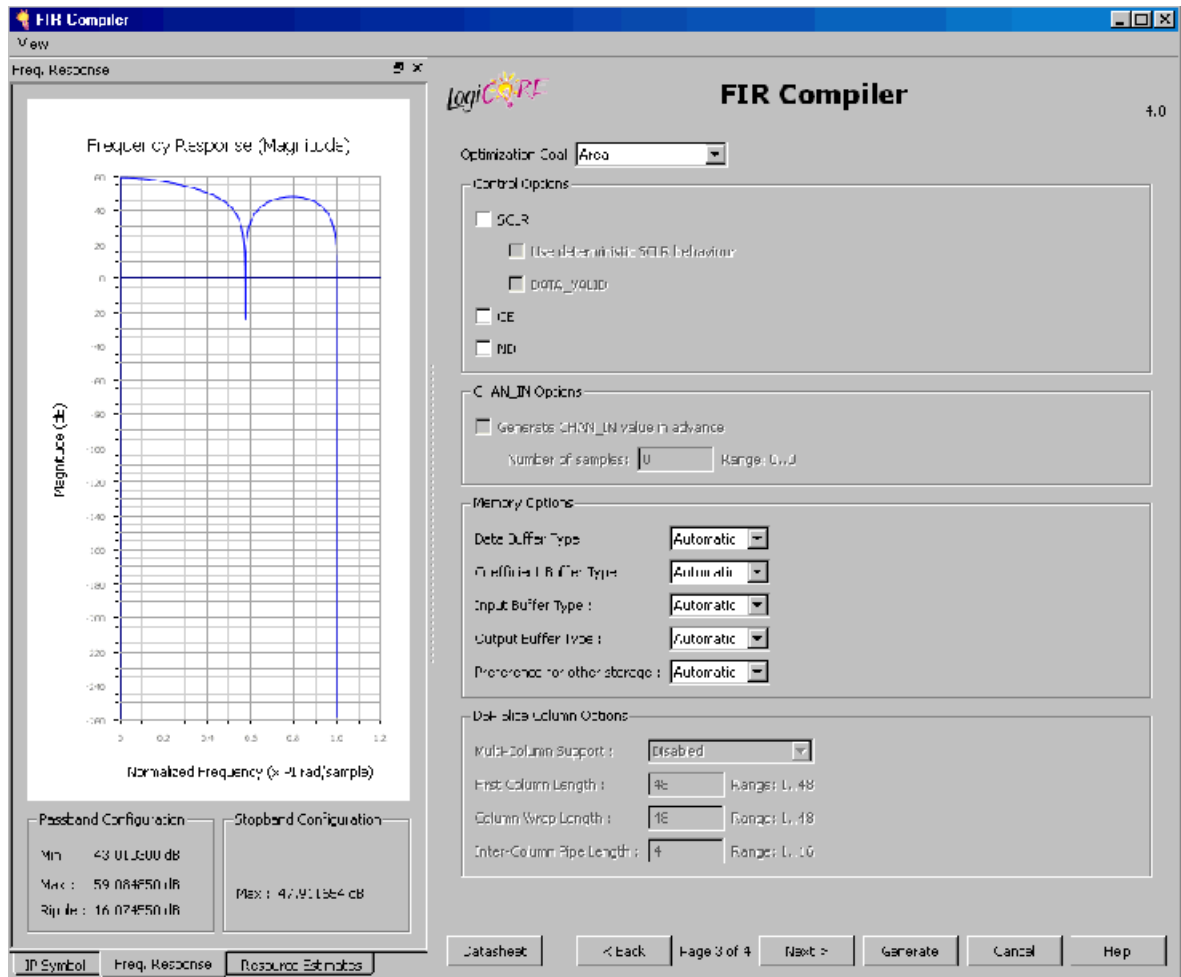
- ◆ その他のオプションは、デフォルト値を使用します。
- ◆ [Next] をクリックします。

- ◆ 次のページで、すべてのオプションをデフォルト値のままにします。



- ◆ [Next] をクリックします。

- ◆ この例では、CE (クロック イネーブル) ポートのないコアをインポートします。次の図に示すように [CE] がオフになっていることを確認し、[Generate] をクリックします。



- CORE Generator で次のファイルが生成されます。
 - ◆ fir_compiler_8tap.ngc: インプリメンテーション ネットリスト
 - ◆ fir_compiler_8tap.vhd: ビヘイビア シミュレーション用の VHDL ラッパ
 - ◆ fir_compiler_8tap.vho: コア インスタンス化テンプレート
 - ◆ fir_compiler_8tap.xco: コアの生成に選択されたパラメータ
 - ◆ 複数の MIF ファイル: 論理シミュレーション用のメモリ初期化ファイル
- System Generator のブラック ボックスにはクロックとクロック イネーブルのペアが必要ですが、このコアにはクロック イネーブルがないので、コア ラッパを指定して最上位にクロック イネーブル ポートを追加する必要があります。
- 次の空のテンプレート ラッパ ファイルを開きます。
 <path_to_sysgen>\examples\coregen_import\example2\
 fir_compiler_8tap_wrapper.vhd
 このファイルには、空のエンティティ宣言が含まれています。

7. 次の手順に従って、テンプレート ラップアを変更します。

- ◆ fir_compiler_8tap.vho ファイルを開きます。
- ◆ fir_compiler_8tap.vho からコンポーネント宣言部分をコピーし、fir_compiler_8tap_wrapper.vhd のコンポーネント宣言エリア (「-- Add Component Declaration from VHO file ----」の後) に貼り付けます。
- ◆ fir_compiler_8tap.vho からコア インスタネーションテンプレートをコピーし、fir_compiler_8tap_wrapper.vhd のアーキテクチャ本体 (「---- ADD INSTANTIATION Template ----」の後) に貼り付けます。
- ◆ fir_compiler_8tap のポート宣言をコピーし、fir_compiler_8tap のエンティティ宣言 (「---- Add Port declaration for entity ----」の後) に貼り付けます。
- ◆ 最上位エンティティ宣言に ce ポートを追加し、「CLK」を「clk」に変更します。

```

LIBRARY std, ieee;
USE std.standard.ALL;
USE ieee.std_logic_1164.ALL;
-- Remember to modify the CLK port declaration
-- of the entity below to be lower case
entity fir_compiler_8tap_wrapper is
---- Add Port declaration for entity ----
port (
  clk: IN std_logic;
  ce: IN std_logic;
  rfd: CUT std_logic;
  rdy: CUT std_logic;
  din: IN std_logic_VECTOR(15 downto 0);
  dout: OUT std_logic_VECTOR(25 downto 0));
---- End Port declaration for entity ----
end fir_compiler_8tap_wrapper;

architecture test of fir_compiler_8tap_wrapper is
-- Add Component Declaration from VHO file ----
component fir_compiler_8tap
port (
  clk: IN std_logic;
  rfd: CUT std_logic;
  rdy: CUT std_logic;
  din: IN std_logic_VECTOR(15 downto 0);
  dout: OUT std_logic_VECTOR(25 downto 0));
end component;

---- End COMPONENT Declaration ----
begin
---- ADD INSTANTIATION Template ----
J0 : fir_compiler_8tap
port map (
  clk => clk,
  rfd => rfd,
  rdy => rdy,
  din => din,
  dout => dout);
---- End INSTANTIATION Template ----
end test;

```

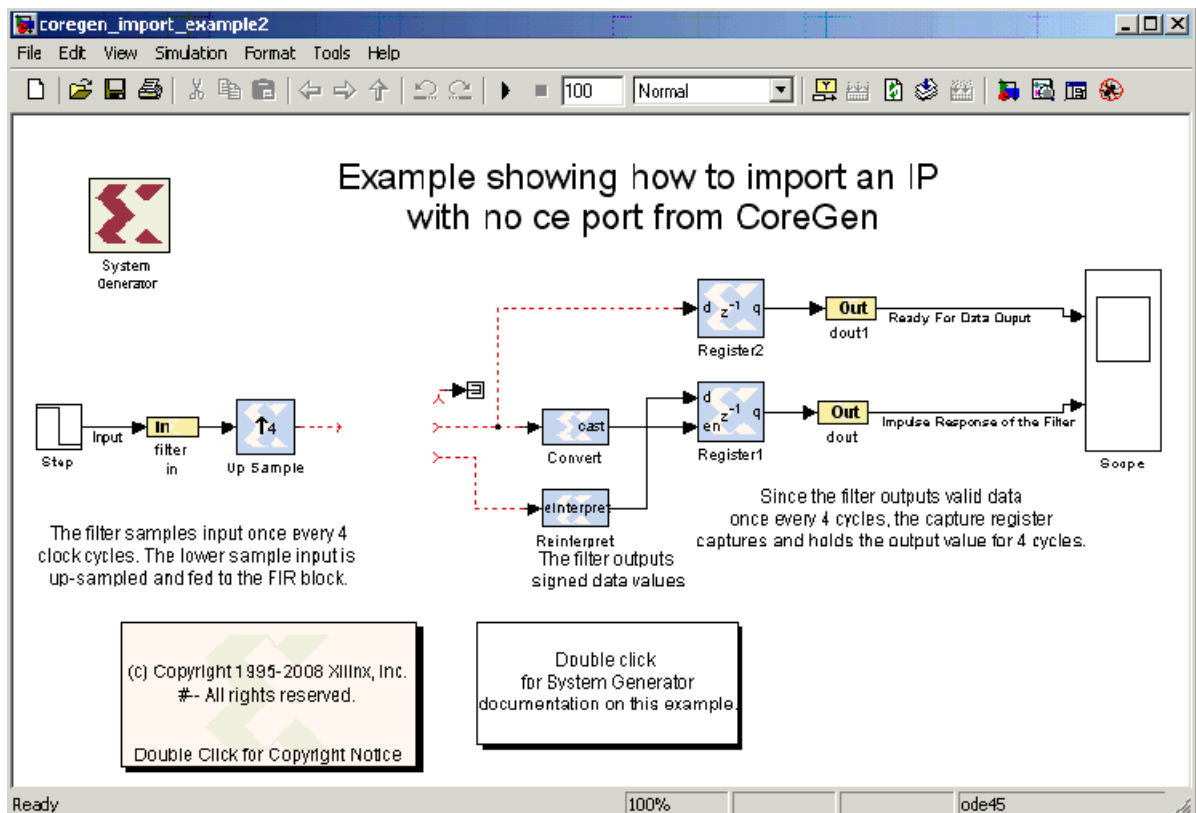
8. Simulink を起動し、次のデザイン ファイルを開きます。

```

<path_to_sysgen>\examples\coregen_import\example2\
coregen_import_example2.mdl

```

9. [Xilinx Blockset] → [Basic Elements] ライブラリから Black Box ブロックを coregen_import_example2.mdl にドラッグします。最上位 HDL ファイルとして fir_compiler_8tap_wrapper.vhd を選択します。



10. Black Box ブロックのポートを未使用のワイヤに接続します。
11. fir_compiler_8tap_wrapper_config.m を開き、ファイルのリストに次の図に示すように VHDL ファイル、EDIF ネットリスト、MIF ファイルを追加します。デザインの System Generator ネットリストが作成される際に、これらのファイルが含まれます。

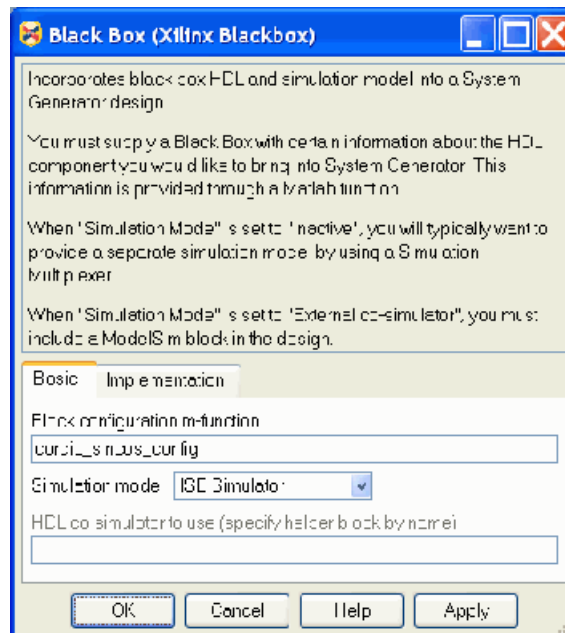
```

67
68 %     this_block.addFile('');
69 %     this_block.addFile('');
70 -     this_block.addFile('fir_compiler_8tap.vhd');
71 -     this_block.addFile('fir_compiler_8tap_wrapper.vhd');
72 -     this_block.addFile('fir_compiler_8tap.mif');
73 -     this_block.addFile('fir_compiler_8tap.ngc');
74
75 - return;

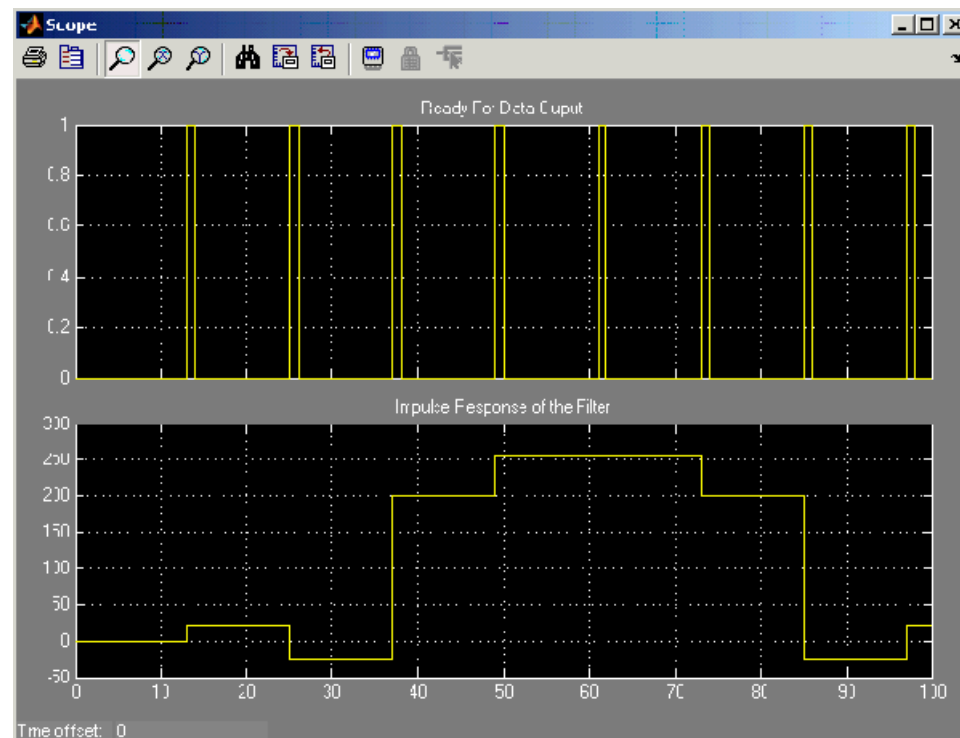
```

メモ：合成およびシミュレーションでは、ファイルはコンフィギュレーション関数に追加した順序でコンパイルされます。

12. Black Box ブロックのパラメータ ダイアログ ボックスを開き、[Simulation mode] を [ISE Simulator] に設定します。



13. Simulink モデルで [シミュレーションの開始] をクリックし、ISim を使用して FIR Compiler コアに対して協調シミュレーションを実行します。次のようなシミュレーション結果が表示されます。



VHDL モジュールのインポート

ブラック ボックスの例 3 : VHDL モジュールのインポート

このセクションでは、Black Box ブロックを使用して VHDL を System Generator デザインにインポートし、ModelSim を使用して協調シミュレーションを実行する方法を示します。

1. MATLAB ウィンドウで、<path_to_sysgen>\examples\black_box\intro ディレクトリに移動します。

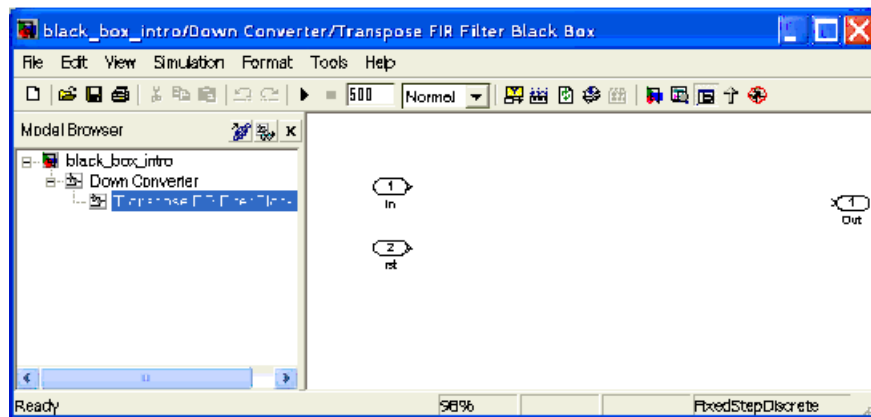
このディレクトリには、次のファイルが含まれています。

- ◆ black_box_intro.mdl : ブラック ボックスの例を含む Simulink モデル
- ◆ transpose_fir.vhd : 転置型 FIR フィルタの最上位 VHDL。ブラック ボックスに関連付ける VHDL です。
- ◆ mac.vhd : 転置型 FIR フィルタを構築する際に使用する MAC コンポーネント

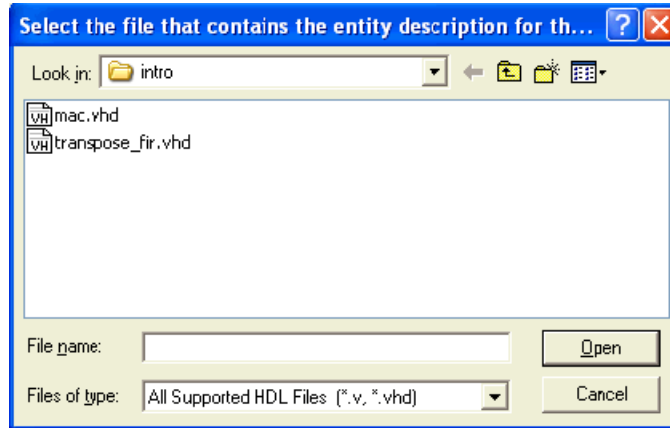
2. MATLAB の [Command Window] に次のように入力し、black_box_intro.mdl を開きます。

```
>> black_box_intro
```

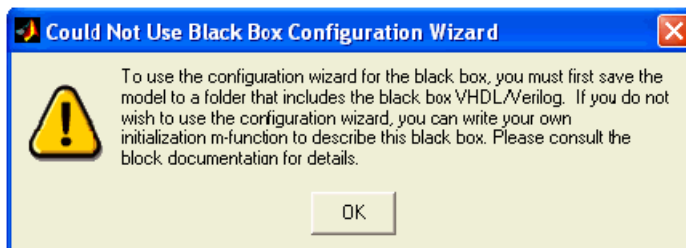
3. Down Converter サブシステムに含まれる Transpose FIR Filter Black Box というサブシステムを開きます。このサブシステムには、2 つの入力ポートと 1 つの出力ポートが含まれています。次の図に、Transpose FIR Filter Black Box サブシステムを示します。



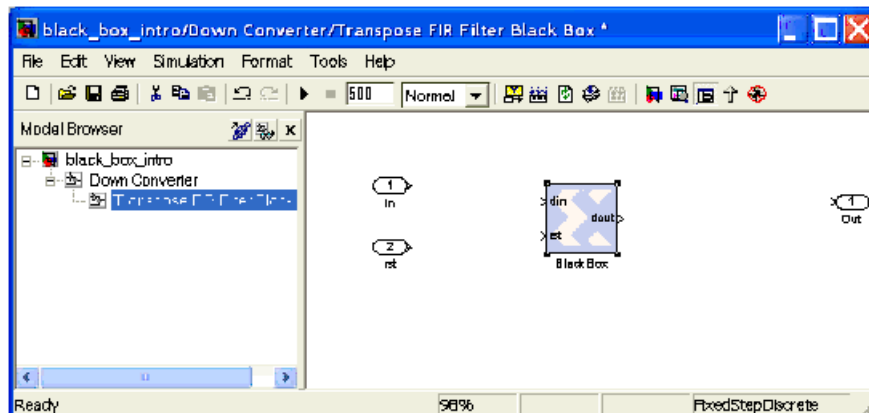
4. Transpose FIR Filter Black Box サブシステムに、Simulink Library Browser から Black Box ブロックを追加します。Black Box ブロックは、[Xilinx Blockset] → [Basic Elements] ライブラリに含まれています。Black Box ブロックをサブシステムに追加すると、ブラック ボックス コンフィギュレーション ウィザードが起動します。最上位 VHDL ファイル transpose_fir.vhd を選択します。これを次の図に示します。



メモ：ウィザードは、ファイルとして保存されているモデルに Black Box ブロックを追加した場合にのみ実行されます。モデルが保存されていない場合は、ウィザードでファイルの検索場所が判断できず、次のような警告メッセージが表示されます。



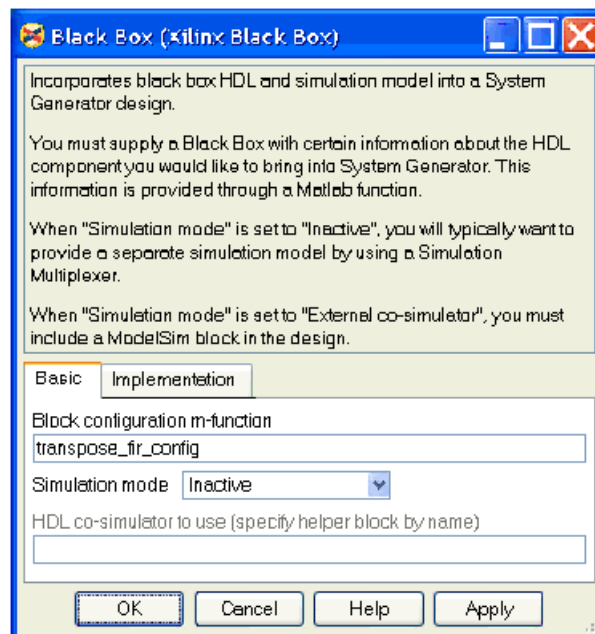
5. ウィザードで VHDL が解析され、Black Box ブロックのコンフィギュレーション M 関数が生成されます。これは MATLAB スクリプトであり、Black Box ブロックを VHDL に関連付け、ポートを作成します。関数を実行すると、Black Box ブロック上のポートが最上位 VHDL エンティティと一致します (クロック ポートおよびクロック イネーブル ポートを除く)。これを次に示します。



この例で作業する際は、次の規則を考慮する必要があります。

- ◆ **Black Box** に関連付けられている同期 HDL デザインに、1 つ以上のクロック ポートおよびクロック イネーブル ポートが含まれていることが必要です。これらのポートは、ペアで指定する必要があります (1 つのクロックに 1 つのクロック イネーブル)。各ポートのデータ型は、`std_logic` に設定します。クロック ポート名には、`clk` を含めます。クロック イネーブルのポート名は、対応するクロックと同じにする必要がありますが、`clk` の部分は `ce` にします。
- ◆ **System Generator** のクロック イネーブル ポートは特殊で、汎用のユーザー イネーブルとは異なります。詳細は、「[ブラック ボックスの HDL の要件と制限](#)」を参照してください。

6. **Black Box** ブロックをダブルクリックします。次のようなダイアログ ボックスが表示されます。



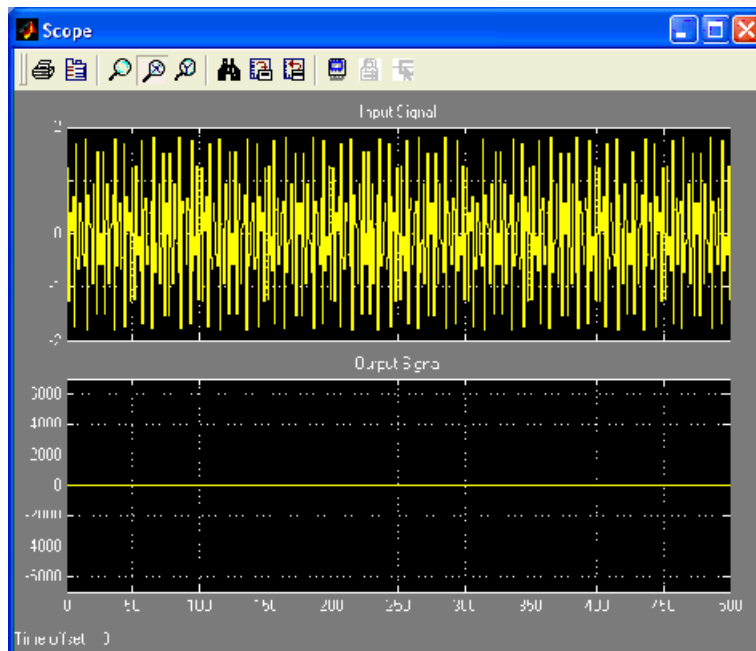
このダイアログ ボックスには、次のフィールドが含まれます。

- ◆ **[Block configuration M-function]** : ブラック ボックスのコンフィギュレーション M 関数の名前を指定します。この例では、ブラック ボックス コンフィギュレーション ウィザードで生成された関数の名前が入力されています。デフォルトでは、ウィザードで生成された関数を使用されます。独自の関数を指定することも可能です。コンフィギュレーション M 関数の詳細は、「[ブラック ボックスのコンフィギュレーション M 関数](#)」を参照してください。
- ◆ **[Simulation mode]** : 次のいずれかのシミュレーション モードを選択します。
 - **[Inactive]** : シミュレーションの際に入力が無視され、0 が生成されます。この設定は、通常ブラック ボックスに別のシミュレーション モデルがある場合に使用されます。このシミュレーション モデルは、シミュレーション マルチプレクサを使用して、**Black Box** ブロックと並列に接続されます。この方法は、「[ブラック ボックスの例 1 : ブラック ボックスの HDL 要件を満たした CORE Generator モジュールのインポート](#)」に示されています。
 - **[ISE Simulator]** : ブラック ボックスのシミュレーション結果が、ブラック ボックスに関連付けられた HDL で協調シミュレーションを使用して作成されます。

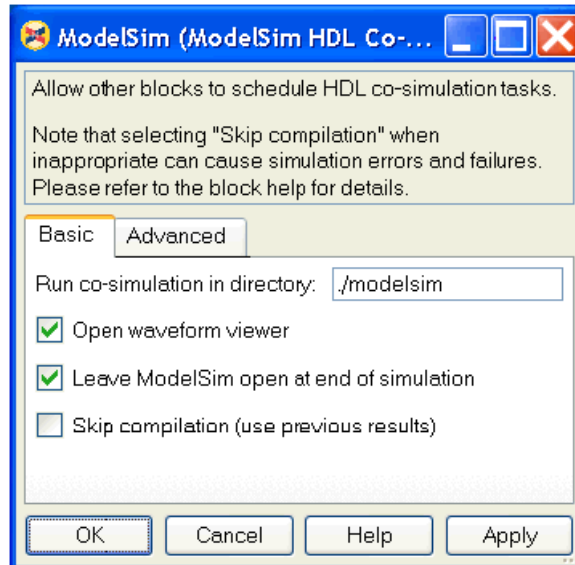
- [External co-simulator] : このオプションを設定した場合は、デザインに ModelSim ブロックを追加し、[HDL co-simulator to use] に ModelSim ブロックの名前を入力する必要があります。ブラック ボックスは、HDL 協調シミュレーションを使用してシミュレーションされます。
- ◆ [FPGA Area Estimation] : ブラック ボックスの HDL で使用される FPGA リソースの予測量を入力します。これらの値は、手動で入力する必要があります。これらの数値は、System Generator で提供されているリソース予測ユーティリティを使用する場合にのみ使用します。詳細は、「リソースの予測」を参照してください。

チュートリアルを続行するには、パラメータ設定をそのままにします。

7. Black Box ブロックのポートを、対応するサブシステムのポートに接続します。
8. Simulink モデルのツールバーで [シミュレーションの開始] をクリックし、Scope ブロックをダブルクリックします。[Output Signal] が 0 であることに注意してください。シミュレーション中 Black Box ブロックが非アクティブになるように設定されているので、この結果は予測とおりです。

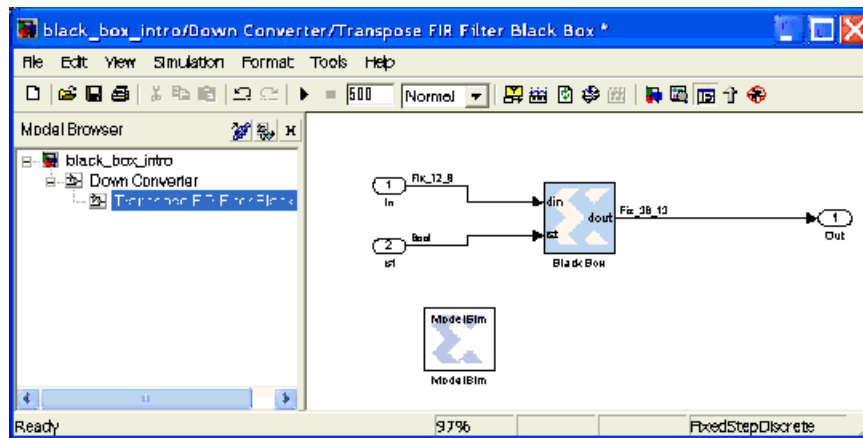


9. Transpose FIR Filter Black Box サブシステムに、Simulink Library Browser から ModelSim ブロックを追加します。ModelSim ブロックは、[Xilinx Blockset] → [Tools] ライブラリに含まれています。このブロックは、Black Box ブロックを ModelSim シミュレータと通信できるようにします。ModelSim ブロックをダブルクリックして、パラメータ ダイアログ ボックスを開きます。

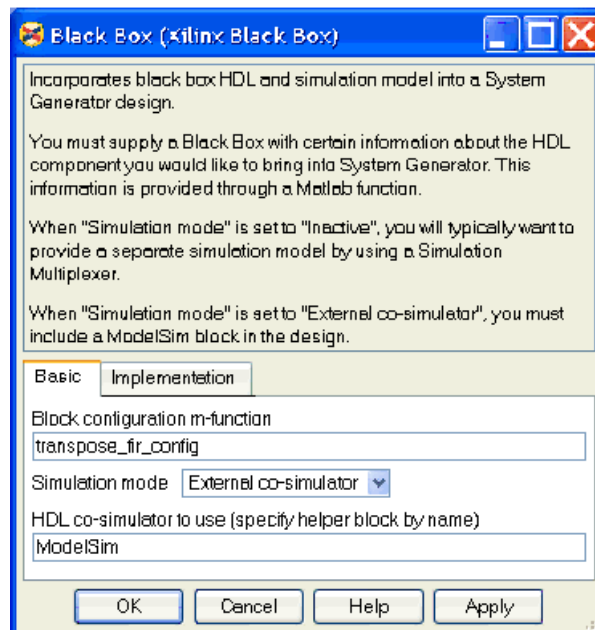


10. パラメータ設定が上図と同じになるようにします。ダイアログ ボックスを閉じます。
11. Simulink メニューで、[書式] → [ポート/信号の表示] → [端子のデータタイプ] をクリックして Black Box ブロックのポートのデータ型を表示します。Ctrl + D キーを押してモデルをコンパイルし、最新のポートのデータ型が表示されるようにします。出力ポートのデータ型は UFix_26_0 で、これは 26 ビット幅の符号なし信号であり、2 進小数点が 0 位置で最下位ビットの左側にあることを示します。
12. コンフィギュレーション M 関数 transpose_fir_config.m を開き、出力のデータ型を UFix_26_0 から Fix_26_12 に変更します。変更後の行は、次のようになります。
- ```
dout_port.setType('Fix_26_12');
```
13. コンフィギュレーション M 関数に、Black Box ブロックに関連付ける HDL ファイルを追加します。次の行を検索します。
- ```
this_block.addFile('transpose_fir.vhd');
```
- この行の上に、次の行を追加します。
- ```
this_block.addFile('mac.vhd');
```

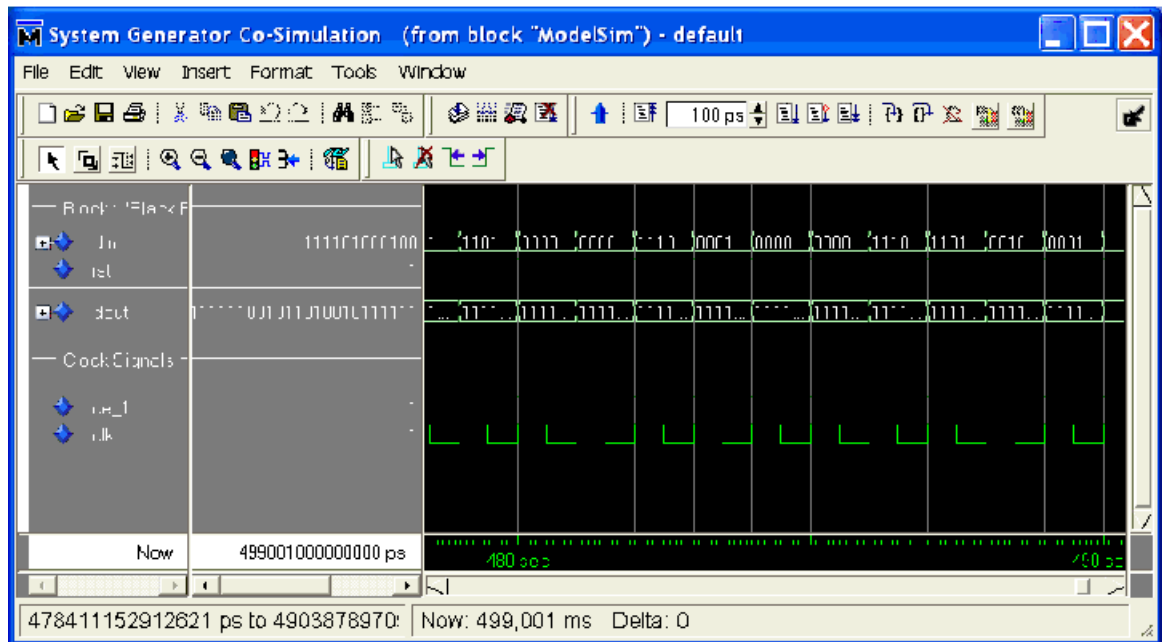
14. コンフィギュレーション M 関数に加えた変更を保存し、Ctrl + D キーを押してモデルをコンパイルします。サブシステムは、次のようになります。



15. Black Box ブロックのパラメータ ダイアログ ボックスで、[Simulation mode] を [External co-simulator] に変更し、[HDL co-simulator to use] に「ModelSim」と入力します。ここには、モデルに追加した ModelSim ブロックの名前を入力します。Black Box ブロックのパラメータ ダイアログ ボックスは、次のようになります。



16. シミュレーションを実行します。ModelSim のコマンド ウィンドウと波形ビューアが開きます。VHDL は ModelSim でシミュレーションされ、シミュレーション全体は Simulink で制御されます。出力波形は次のようになります。

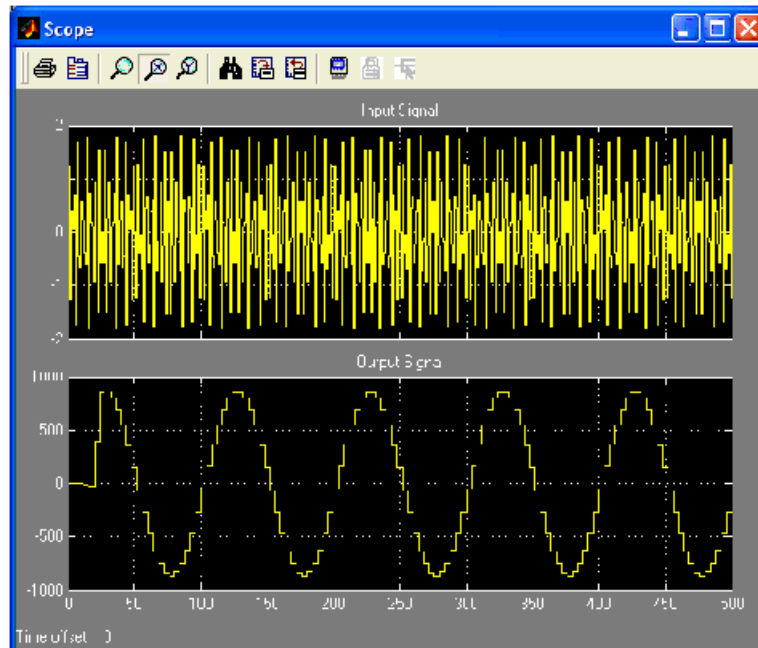


次の警告メッセージは、無視しても問題ありません。

```
** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand,
the result will be 'X'(es).
Time: 0 ps Iteration: 0 Instance: /
xlcossim_black_box_ex1_down_converter_transpose_fir_filter_b1
ack_box_modelsim/
black_box_ex1_down_converter_transpose_fir_filter_black_box_
black_box/g0__22/g_last/m2
```

このメッセージは、Black Box ブロックの VHDL でシミュレーション開始時の初期値が指定されていないために表示されます。

17. シミュレーションが完了したら、Scope ブロックの出力を確認します。[Simulation mode] を [Inactive] に設定したときは [Output Signal] が 0 でしたが、[External co-simulator] に変更した結果、0 ではなく、ModelSim シミュレーションの結果が表示されるようになりました。



## Verilog モジュールのインポート

このセクションでは、Verilog ブラック ボックスを System Generator で使用し、ModelSim を使用して協調シミュレーションする方法を示します。Verilog モジュールは、VHDL モジュールと同様にインポートできます。モジュールのインポート方法の詳細は、「[ブラック ボックス コンフィギュレーション ウィザード](#)」および「[ブラック ボックスのコンフィギュレーション M 関数](#)」を参照してください。System Generator では、Verilog ブラック ボックスをハードウェアおよび協調シミュレーション HDL に組み込むのに必要なコードがすべて提供されています。また、Verilog ブラック ボックスのパラメータも設定できます。この例では、これらの機能を示します。この例のファイルは、次のディレクトリに含まれています。

`<path_to_sysgen>\examples\black_box\example4`

このディレクトリには、次のファイルが含まれます。

- `black_box_ex4.mdl` : VHDL を使用するブラック ボックスと Verilog を使用するブラック ボックスの 2 つのブラック ボックスを含む Simulink モデル。
- `word_parity_block.vhd` : ワード パリティ例のステート マシンの組み合わせ部分を表す VHDL。この部分は完全に組み合わせブロックで、各入力ワードのパリティを算出し、パリティビットを出力します。ジェネリックを使用してパラメータ指定されており、どの入力型でも処理できます。ジェネリックについては、ダイナミック ブラック ボックスの説明を参照してください。
- `word_parity_block_config.m` : VHDL ブラック ボックスのコンフィギュレーション M 関数 (ジェネリック設定を含む)。この関数では、Simulink で正しくシミュレーションされるよう、ブロックが組み合わせであることが示されています。
- `shutter.v` : 単純な動機ラッチの Verilog。入力ポート `din` に任意の幅を使用できるよう設定されています。

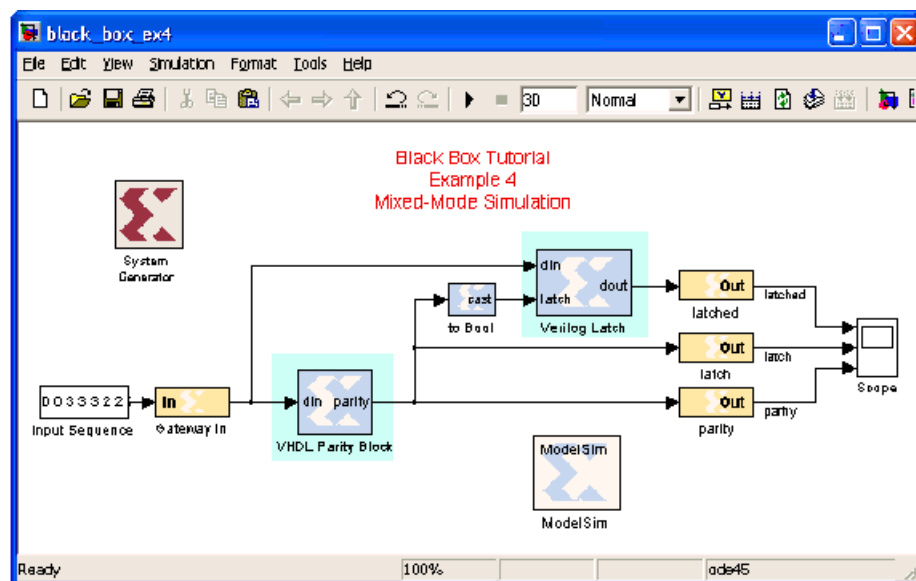
- shutter\_config.m : Verilog ブラック ボックスのコンフィギュレーション M 関数 (パラメータ設定を含む)。Verilog ブラック ボックスをコンフィギュレーションするため、VHDL 構文を参照するメソッドが含まれます。次の行が含まれています。

```
this_block.setEntityName('shutter');
this_block.addGeneric('din_width', dwidth);
```

## ブラック ボックスの例 4 : Verilog モジュールのインポート

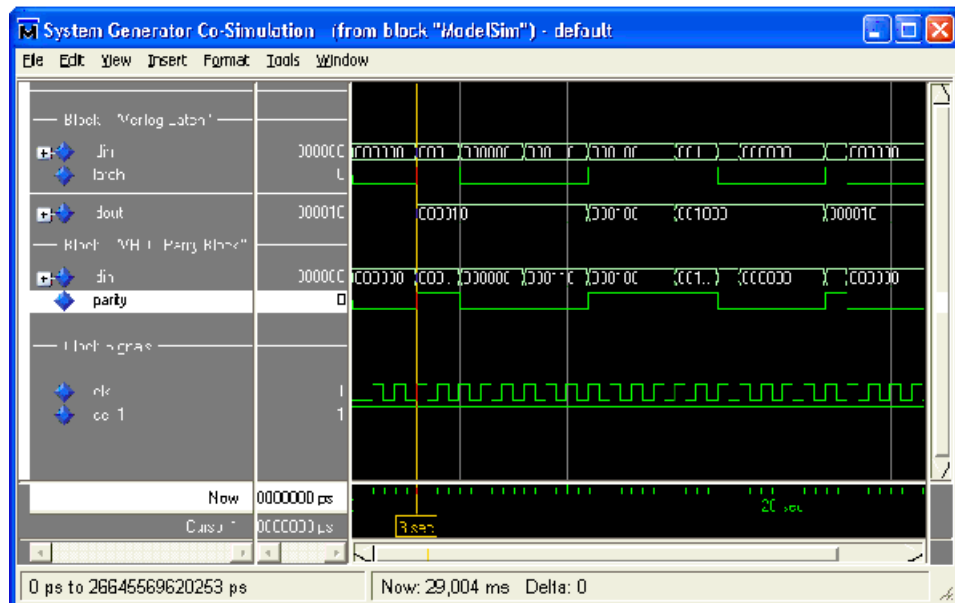
1. example4 に移動し、black\_box\_ex4.mdl を開きます。

これは、VHDL ブラック ボックスと Verilog ブラック ボックスの 2 つのブラック ボックスを含む単純なデザインです。VHDL ブラック ボックスは各入力ワードのパリティを算出し、Verilog ブラック ボックスは奇数のパリティを持つワードを格納します。ブラック ボックスの動作を算出するのに Simulink モデルは使用せず、HDL 協調シミュレーションを使用します。次の図に、このデザイン例を示します。



この例でシミュレーションを実行するには、ModelSim の混合モード シミュレーションのライセンスが必要です。ライセンスがある場合、シミュレーションを実行すると、次のような ModelSim の波形ビューアが表示され、両方のブラック ボックスの動作が示されます。ModelSim でデザインの構造を調べると、System Generator で 2 つのブラック ボックスがどのように統合されているかがわかります。

2. 入力のデータ型を **arbitrary** に変更し、シミュレーションを再実行します。変更に応じてブラックボックスが調整されます。



## ダイナミック ブラック ボックス

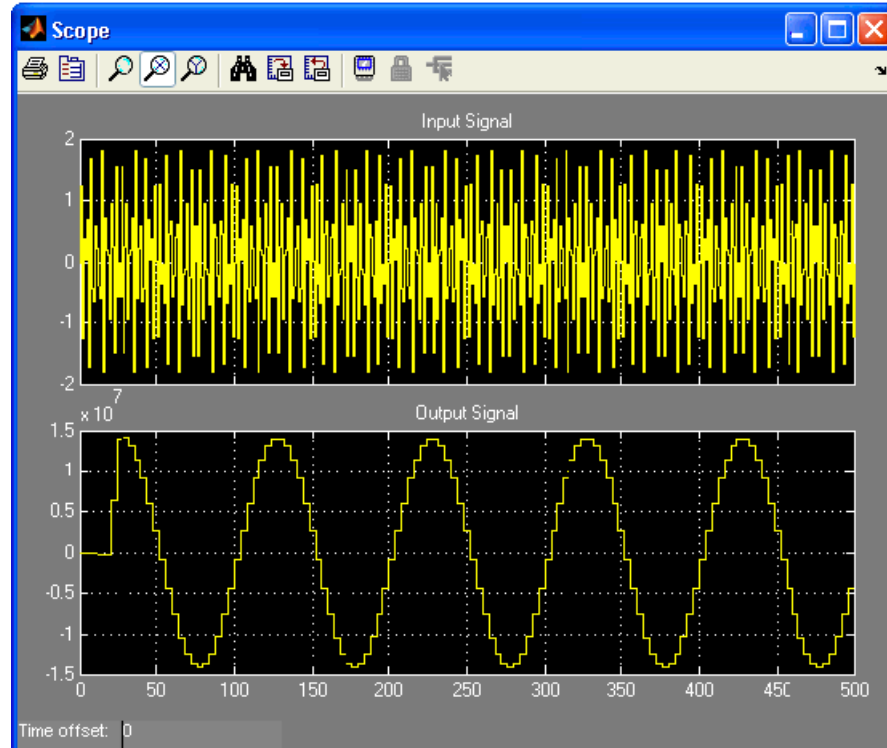
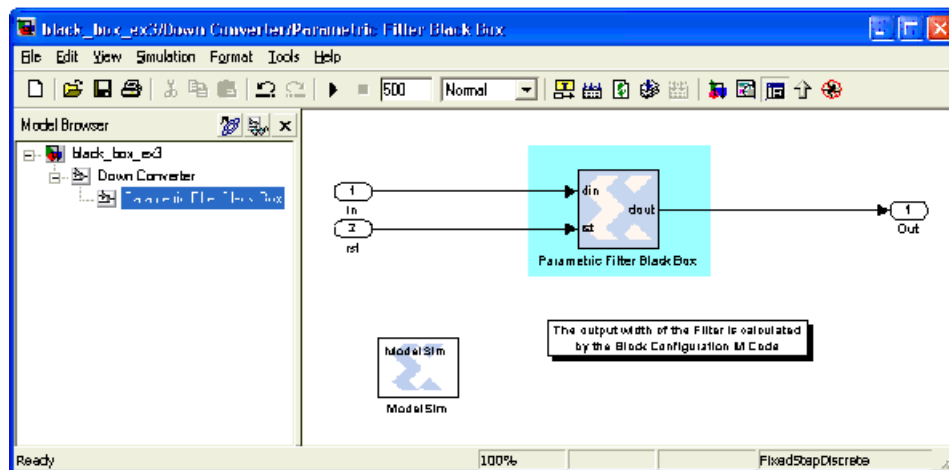
この例では、転置型 FIR フィルタ ブラック ボックスを拡張し、入力幅の変化に応じて動的に調整されるようにします。この例のファイルは、`<path_to_sysgen>\examples\black_box\example3` に含まれています。この例を正しく実行するには、MATLAB の [Command Window] で `cd` コマンドを使用してこのディレクトリに移動する必要があります。

このディレクトリには、次のファイルが含まれています。

- `black_box_ex3.mdl` : ダイナミック ブラック ボックスを含む Simulink モデル
- `transpose_fir_parametric.vhd` : 転置型 FIR フィルタの VHDL
- `mac.vhd` : 転置型 FIR フィルタを構築する際に使用する MAC コンポーネント
- `transpose_fir_parametric_config.m` : ブラック ボックスのコンフィギュレーション M 関数

## ブラックボックスの例 5: ダイナミック ブラックボックス

1. MATLAB の [Command Window] に「black\_box\_ex3」と入力し、モデルを開きます。
2. 最上位モデルからシミュレーションを実行し、Scope ブロックで結果を表示します。
3. Gateway In ブロック Din のパラメータダイアログボックスで、[Number of bits] を 16 から 12 に、[Binary point] を 14 から 10 に変更して、シミュレーションを再実行します。ブラックボックスの入力幅と出力幅は、自動的に調整されます。Parametric Filter Black Box サブシステムとシミュレーション結果は、次のようになります。





4. コンフィギュレーション M 関数での定義により、入力幅の変更に応じてブラック ボックスが調整されます。これを機能させるため、M 関数を変更する必要があります。コンフィギュレーション M 関数 `transpose_fir_parametric.m` を開きます。重要な点は、次のとおりです。

- 入力データ幅の取得 :  

```
input_bitwidth = this_block.port('din').width;
```
- 出力幅の算出 :  

```
output_bitwidth = ceil(log2(2^(input_bitwidth-1)*2^(coef_bitwidth-1) *
number_of_coef));
```
- 出力のデータ型の設定 :  

```
dout_port.makeSigned;
dout_port.width = output_bitwidth;
dout_port.binsize = 12;
```
- 入力と出力のビット幅を VHDL にジェネリックとして渡す :  

```
this_block.addGeneric('input_bitwidth',this_block.port('din').width);
this_block.addGeneric('output_bitwidth',output_bitwidth);
```

ブラック ボックスのコンフィギュレーション M 関数の詳細は、「[ブラック ボックスのコンフィギュレーション M 関数](#)」を参照してください。

ブラック ボックスの VHDL ファイル `transpose_fir_parametric.vhd` を見ると、ジェネリック `input_bitwidth` で入力幅、`output_bitwidth` で出力幅が指定されています。これらのジェネリックは、下位の VHDL コンポーネントに伝搬されます。

## 複数のブラック ボックスの同時シミュレーション

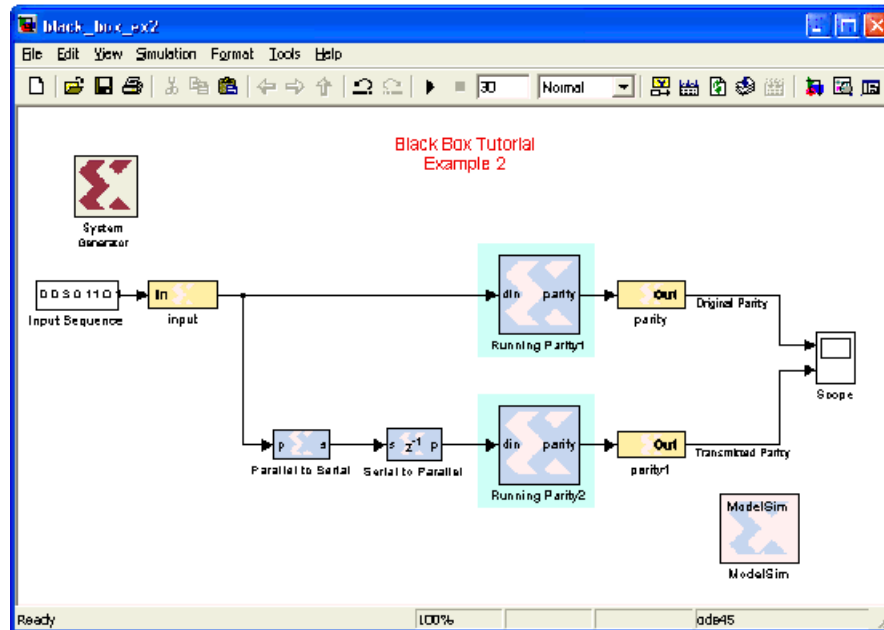
1 つの ModelSim ライセンスを使用し、複数の Black Box ブロックに対して同時に協調シミュレーションを実行できます。次の例では、この方法を示します。この例のファイルは、`<path_to_sysgen>\examples\black_box\example2` ディレクトリに含まれています。

このディレクトリには、次のファイルが含まれています。

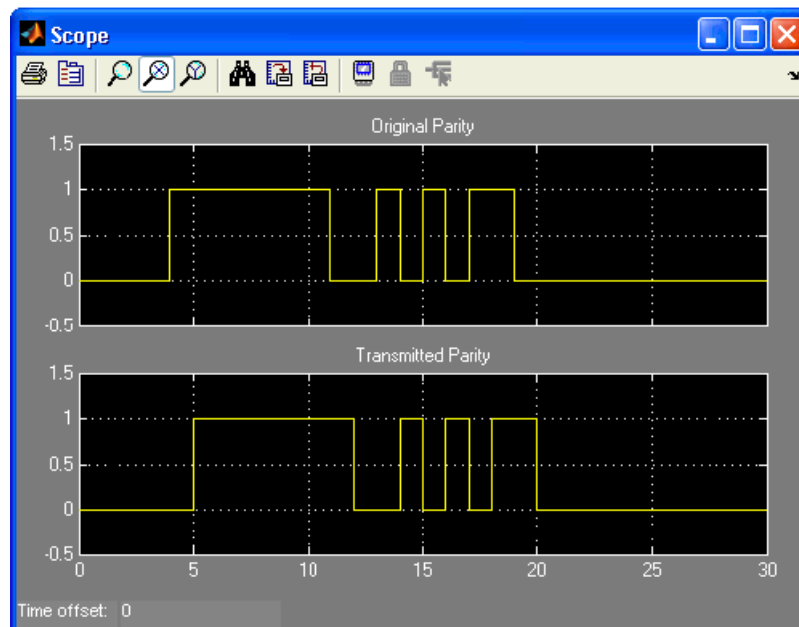
- `black_box_ex2.mdl` : 2 つのブラック ボックスを含む Simulink モデル
- `parity_block.vhd` : 8 ビットの入力ワードのランニング パリティをモニタする単純なステート マシンの VHDL
- `parity_block_config.m` : ブラック ボックスのコンフィギュレーション M 関数。ブラック ボックス コンフィギュレーション ウィザードで生成されたコンフィギュレーション M 関数から、組み合わせパスが含まれることを示すメソッド (`this_block.tagAsCombinational`) を削除したものです。

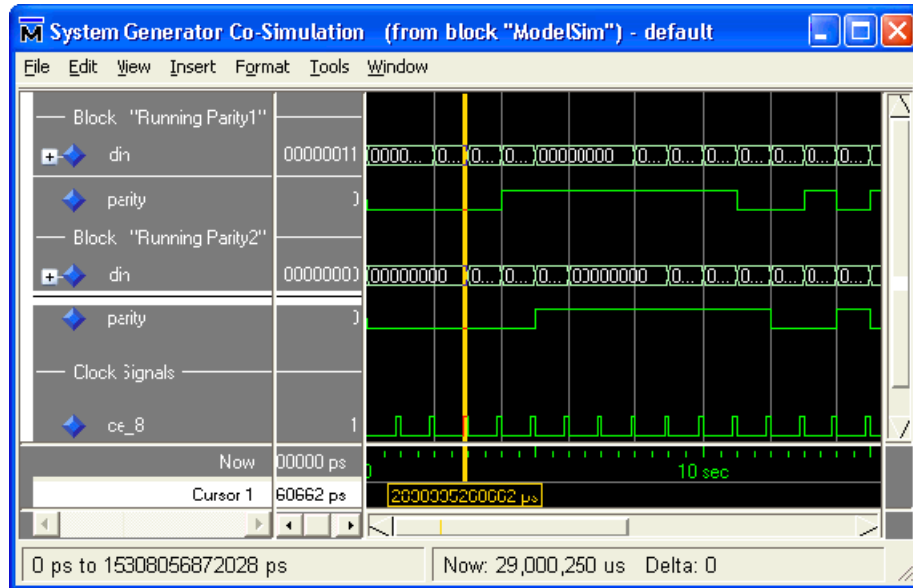
### ブラック ボックスの例 6 : 複数のブラック ボックスの同時シミュレーション

`example2` に移動し、`black_box_ex2.mdl` を開きます。これは、ステート マシンをインプリメントする 2 つのブラック ボックスを含む単純なモデルです。このステート マシンは、入力のランニング パリティを算出します。1 つのブラック ボックスにはモデルの入力ストリームを入力し、もう 1 つのブラック ボックスには入力ストリームをシリアルライズおよびデシリアルライズしたものを入力します。どちらのステート マシンにも、シミュレーション モデルではなく、HDL 協調シミュレーションを使用してシミュレーション結果を生成します。ModelSim ブロックは、ModelSim でブラック ボックスにアクセスできるようにします。次の図に、このデザイン例を示します。



シミュレーションを実行すると、Simulink の Scope ブロックと ModelSim の波形ビューアは次のようになります。Scope ブロックでは、両方のブラック ボックスで同じパリティ結果が生成されることを示していますが、1 クロック サイクルのずれがあります。ModelSim の波形ビューアでも同じ結果が表示されますが、2 進数で表示されます。System Generator で波形ビューアが自動的に設定されており、各ブラック ボックスの入力信号と出力信号が表示されます。ModelSim でデザインの構造を調べると、System Generator で 2 つのブラック ボックスがどのように統合されているかがわかります。





## ModelSim を使用したアドバンス ブラック ボックスの例

この例では、次の内容について説明します。

- ダイナミック ポート インターフェイスを使用したブラック ボックスの設計
- マスク パラメータを使用してブラック ボックスを設定する方法
- 入力ポートのデータ型に基づいてジェネリック値を割り当てる方法
- ブラック ボックスを再利用できるように **Simulink** ライブラリに保存する方法
- **ModelSim** HDL 協調シミュレーション用のカスタム スクリプトの指定方法

この例では、ブラック ボックスからの信号を表示する方法も示します。**Simulink** では、波形は通常 **Scope** ブロックで表示します。バージョン 8.1 以降の **System Generator** では、**WaveScope** ブロックも使用できます。**ModelSim** の波形ビューアを使用して波形を表示することもできます。この例では、ブラック ボックスをザイリンクスの固定小数点信号を表示する特別な **ModelSim** 波形スコープとしてコンフィギュレーションしています。このブラック ボックス スコープを使用するモデルをシミュレーションすると、ブラック ボックスを駆動する信号が **ModelSim** に表示されます。

この例のファイルは、次のディレクトリに含まれています。

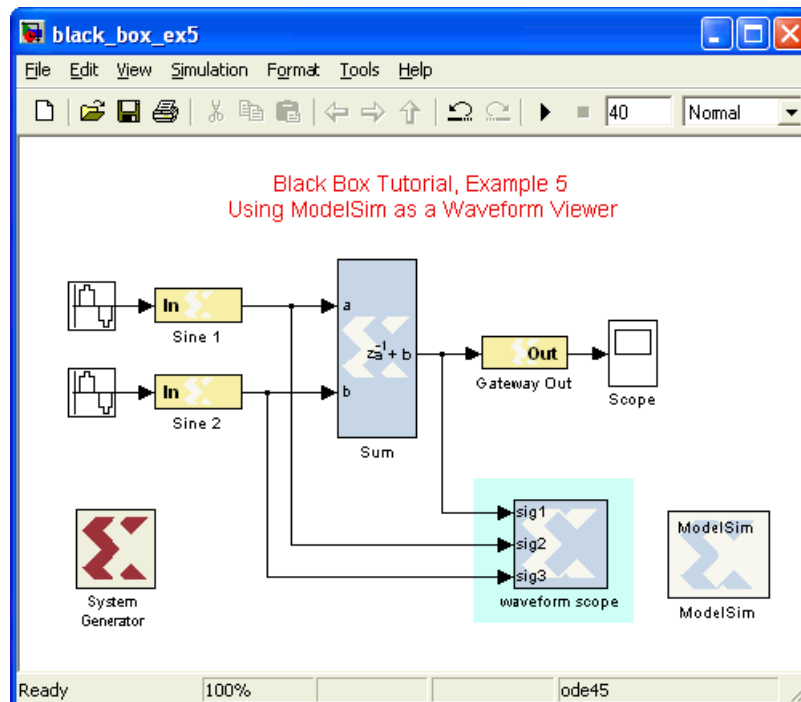
<path\_to\_sysgen>\examples\black\_box\example5

このディレクトリには、次のファイルが含まれています。

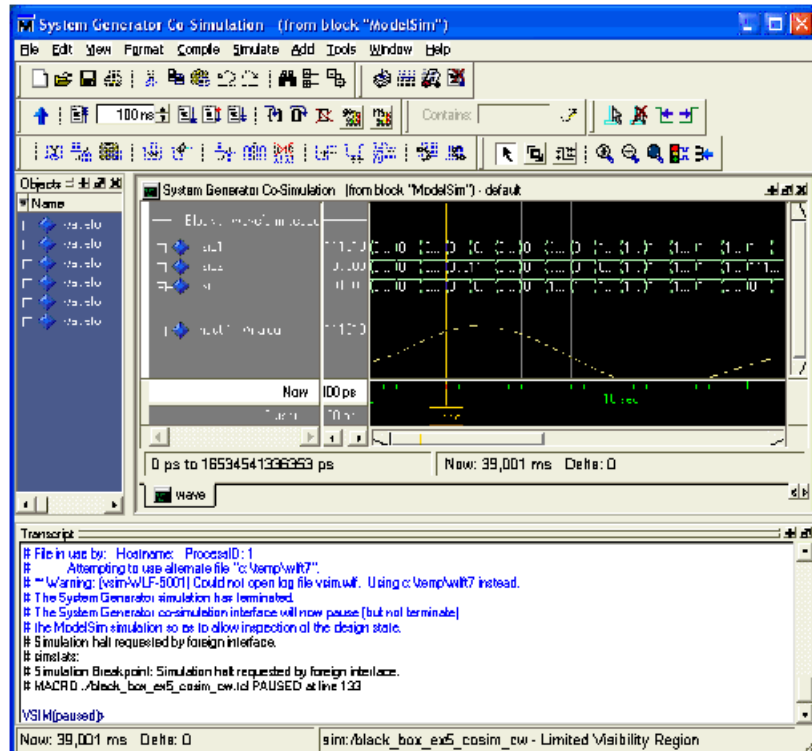
- **black\_box\_ex5.mdl**: ブラック ボックス スコープを含む **Simulink** モデル
- **scope\_lib.mdl**: ブラック ボックス スコープを含む **Simulink** ライブラリ
- **scope\_config.m**: ブラック ボックス スコープのコンフィギュレーション M 関数
- **scope1.vhd**, **scope2.vhd**, **scope3.vhd**, **scope4.vhd**: 入力信号を 1 つ、2 つ、3 つ、4 つ受信できるブラック ボックス スコープの **VHDL**
- **waveform.do**: シミュレーション中に **ModelSim** でどのように信号を表示するかを指定するスクリプト

## ブラック ボックスの例 7 : ModelSim を使用したアドバンス ブラック ボックス

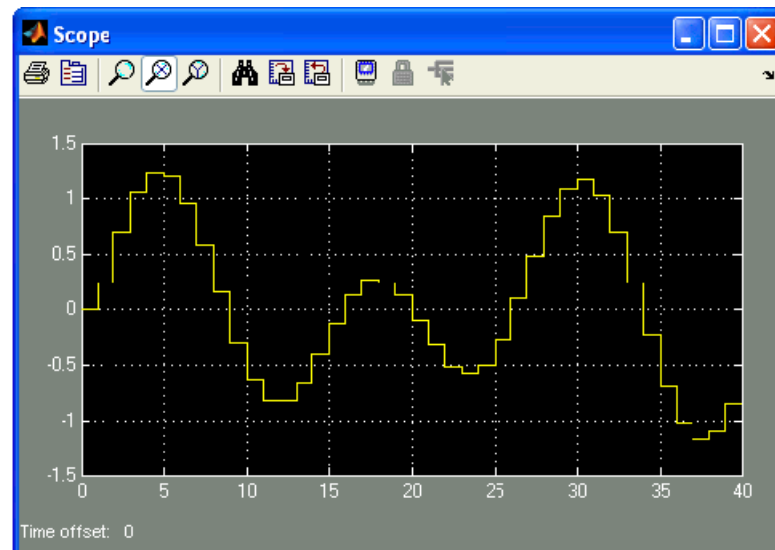
1. example5 に移動し、black\_box\_ex5.mdl を開きます。このモデルには、2 つの Gateway In ブロックで駆動される加算器が含まれています。Gateway In ブロックは、2 進小数点の右側に 6 ビットある符号付きの 8 ビット値を生成するように設定されています。サイン波ジェネレータからの出力で駆動されます。このモデルには、waveform scope というブラックも含まれます。このブロックは、1 つの入力は加算器の出力で駆動され、残りの 2 つの入力は加算器への入力で駆動されます。ModelSim ブロックは、HDL 協調シミュレーションを実行できるようにします。このモデルを次に示します。



2. black\_box\_ex5 モデルをシミュレーションします。ModelSim ウィンドウが開き、シミュレーションに必要なファイルがコンパイルされます。コンパイルが終了すると、MATLAB と ModelSim の両方でシミュレーションが開始します。ModelSim の波形ビューアが開き、4 つの信号が表示されます。最初の入力 sig1 は、加算器の出力で駆動されます。この信号は、ModelSim の波形ビューアでは 2 進数とアナログで表示されます。次の図に、black\_box\_ex5 のシミュレーションの ModelSim 波形を示します。

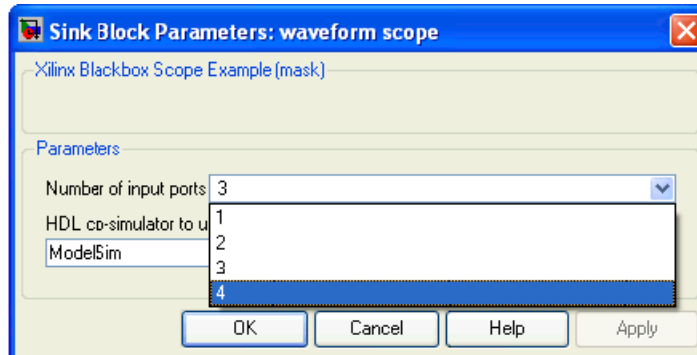


3. モデルで Simulink の Scope ブロックをダブルクリックします。表示される出力は、ModelSim の波形ビューアのアナログ信号に類似しています。

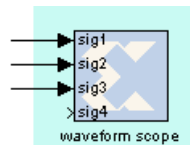


この例のブラック ボックスは、マスク パラメータを使用してコンフィギュレーションされています。この方法は、さまざまな状況で有益です。この場合は、マスク パラメータを使用してブラック ボックスの入力ポートの数 (waveform scope の入力数) を指定しています。

4. **waveform scope** ブラック ボックスをダブルクリックします。パラメータ ダイアログ ボックスに入力ポートの数を指定する **[Number of input ports]** が含まれており、ブラック ボックスのインスタンスごとに個別の値を指定できます。次の図に、**waveform scope** のパラメータ ダイアログ ボックスを示します。



5. **[Number of input ports]** を 3 から 4 に変更し、**[適用]** ボタンをクリックして変更を適用します。ブラック ボックスに **sig4** という入力ポートが追加され、次の図のようになります。



各ブラック ボックスには、マスク パラメータの標準リストがあります。この例のブラック ボックスでは、ユーザーが選択した入力ポートの数を保存する **nports** という追加のマスク パラメータがあります。ブラック ボックスのマスク パラメータを変更するには、ライブラリへのリンクを解除する必要があります。このようにブラック ボックスを変更する場合は、ライブラリのブラック ボックスを保存することをお勧めします。ライブラリの詳細は、**Simulink** のマニュアルを参照してください。scope\_lib.mdl ライブラリには、この例で使用する変更されたブラック ボックスが含まれています。ブラック ボックスのコンフィギュレーション **M** 関数に **HDL** ファイルを追加する場合、このファイルへのパスはライブラリが保存されているディレクトリを基準とした相対パスで指定できます。相対パスを使用すると、**HDL** をモデルと同じディレクトリにコピーする必要がなくなります。

ブラック ボックスのパラメータ ダイアログ ボックスに変更が加えられるたびに、コンフィギュレーション **M** 関数が実行されます。これにより、コンフィギュレーション **M** 関数でマスク パラメータがチェックされ、それに応じてブラック ボックスがコンフィギュレーションされます。この例では、**M** 関数により、マスクで指定された **nports** パラメータに基づいてブロックの入力ポートの数が調整されます。

6. ブラック ボックスのコンフィギュレーション **M** 関数を定義する **scope\_config.m** を開きます。次の行を検索します。

```
simulink_block = this_block.blockName;
```

この行は、ブラック ボックスの名前を取得し、**simulink\_block** 変数に割り当てます。この名前は、**MATLAB** 関数でブロックを変更する際に必要なハンドルです。

7. 次の行を検索します。

```
nports = eval(get_param(simulink_block,'nports'));
```

マスク パラメータ `nports` の値は、`get_param` コマンドを使用して取得します。このコマンドは、ポート数を含む文字列を返します。`get_param` を含む `eval` は、文字列を `nports` 変数に割り当てる整数に変換します。

8. 入力ポートが取得されると、ブラック ボックスに入力ポートが追加されます。これを実行するコードを次に示します。

```
for i=1:nports
 this_block.addSimulinkInport(sprintf('sig%d',i));
end
```

`scope1.vhd`、`scope2.vhd`、`scope3.vhd`、`scope4.vhd` という、この例のブラック ボックスで使用可能な VHDL ファイルが 4 つあります。ブラック ボックスは、適切なポート数を宣言するファイルに関連付けられます。

9. コンフィギュレーション `M` 関数で、ブラック ボックスに適切な VHDL ファイルが選択されます。`scope_config.m` で、次の行を検索します。

```
entityName = sprintf('scope%d',nports);
```

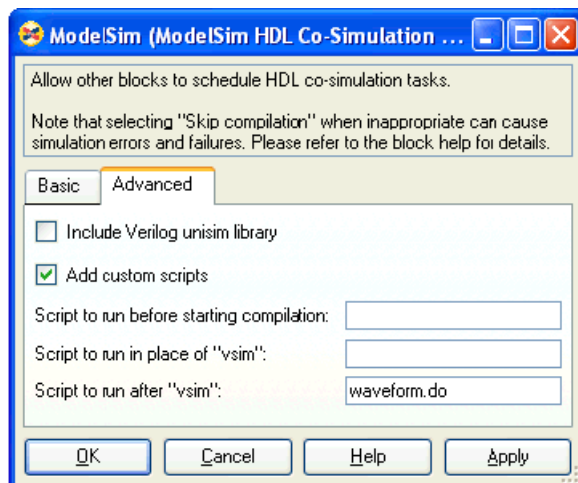
ブラック ボックスの HDL エンティティ名は、`scope` に `nports` の値を追加したものです。ブラック ボックスに VHDL を関連付ける行は、次のとおりです。

```
this_block.addFile(['vhdl/' entityName '.vhd']);
```

10. 各 VHDL エンティティの入力ポート幅は、ジェネリックを使用して割り当てられます。ジェネリック名から、幅を割り当てる入力ポートを特定できます。たとえば `width3` ジェネリックは、3 番目の入力の幅を指定します。`scope_config.m` で、ジェネリック名とその値は次のように設定されます。

```
% -----
if (this_block.inputTypesKnown)
 for i=1:nports
 width = this_block.inport(i).width;
 this_block.addGeneric(sprintf('width%d',i),width);
 end
end % if(inputTypesKnown)
% -----
```

11. シミュレーション中に ModelSim での信号波形の表示方法は、ModelSim ブロックのカスタム tcl スクリプトを使用して変更できます。black\_box\_ex5 モデルで ModelSim ブロックをダブルクリックします。次のようなダイアログ ボックスが表示されます。



カスタム スクリプトを指定するには、[Advanced] タブで [Add custom scripts] をオンにします。この例では、[Script to run after "vsim"] で waveform.do というスクリプトが指定されています。このスクリプトには、加算器の出力をアナログ波形で表示するための ModelSim コマンドが含まれます。



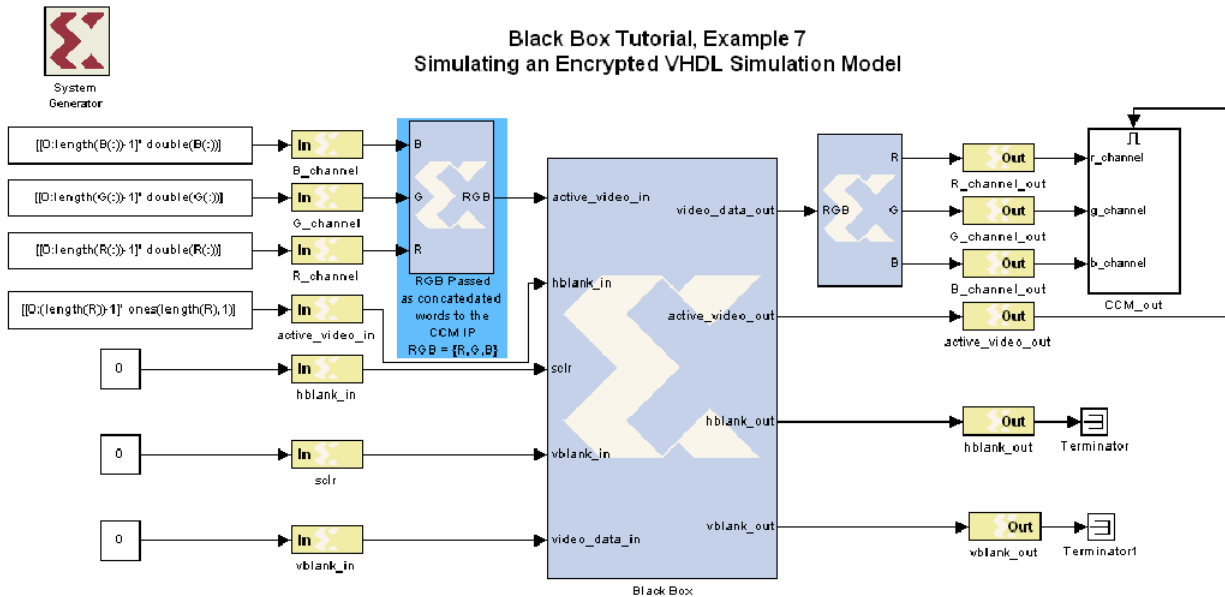
## 暗号化された VHDL ファイルのインポート、シミュレーション、エクスポート

この例では、暗号化された VHDL ファイルを Black Box ブロックにインポートし、デザインをシミュレーションして、VHDL をネットリストのほかの部分とは個別の暗号化されたファイルとしてエクスポートする方法を示します。

### ブラック ボックスの例 8 : 暗号化された VHDL ファイルのインポート、シミュレーション、エクスポート

1. MATLAB で次の MDL ファイルを開きます。

<sysgen\_tree>/examples/black\_box/example7/black\_box\_ex7.mdl



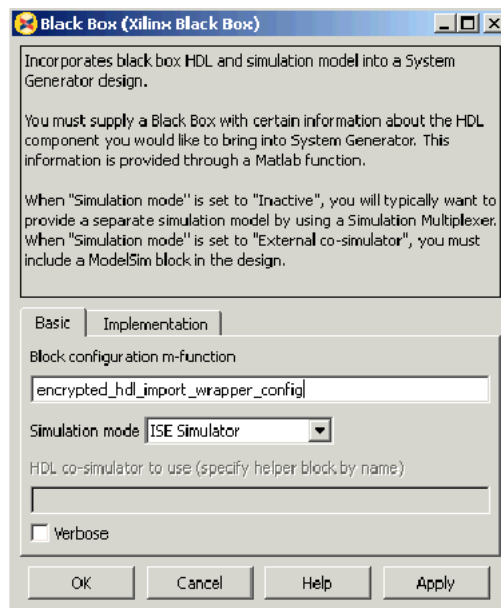
このデザインは、ライセンス付きのコア Color Correction Matrix v1.0 から生成された暗号化された VHDL ファイルをインポートします。このコアの入力は 24 ビットの RGB 信号 {R, G, B} で、出力は次のように色変換された 24 ビット信号 {Rt, Gt, Bt} です。

$$\begin{bmatrix} R_t \\ G_t \\ B_t \end{bmatrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.5 & 1.0 & 0.0 \\ 0.5 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

active\_video\_in 信号は、各 video\_data\_in サンプルが有効であることを示します。hblank\_in および vblank\_in 信号は、このデザイン例では無視されます。このコアの詳細は、Color Correction Matrix v1.0 LogiCORE のデータシートを参照してください。

2. encrypted\_hdl\_import.vhd というファイルは、CORE Generator で生成された暗号化されたシミュレーション モデルです。この暗号化されたシミュレーション モデルをインポートするには、暗号化された VHDL モデルをインスタンス化して VHDL ラッパ ファイルを作成し、ブラック ボックス コンフィギュレーション ウィザードを使用してこのラッパ ファイルをインポートします。この手順は「ブラック ボックスの例 2 : ブラック ボックスの HDL 要件を満たす VHDL ラッパが必要な CORE Generator モジュールのインポート」で説明されており、この例では既に実行されています。

ブラック ボックスを生成すると、encrypted\_hdl\_import\_wrapper\_config.m というコンフィギュレーション ファイルが生成されます。デザイン例で **Black Box** ブロックをダブルクリックすると、このコンフィギュレーション ファイルが指定されていることがわかります。

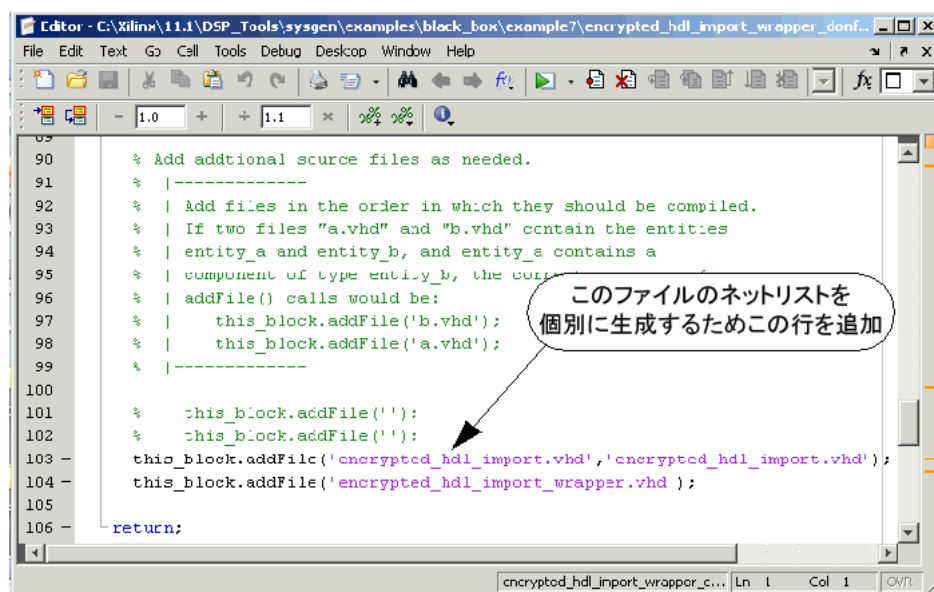


使用するシミュレータとして、[ISE Simulator] が選択されています。

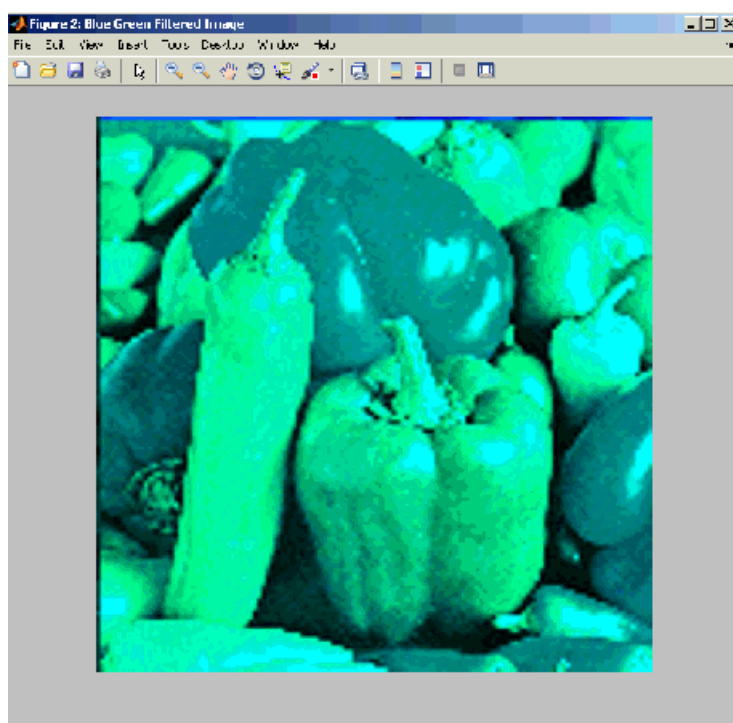
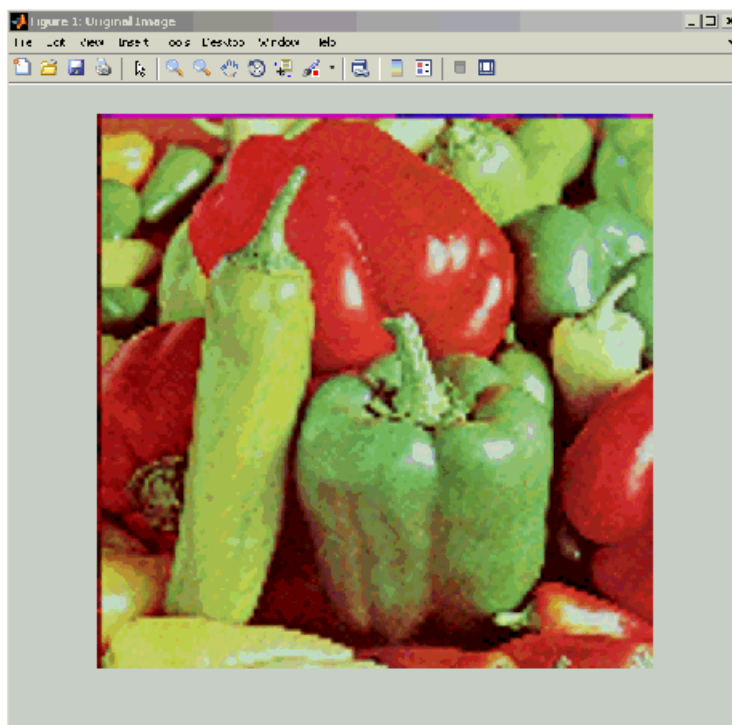
3. System Generator で暗号化された VHDL ファイルに対して個別にネットリストが生成されるようにするには、encrypted\_hdl\_import\_wrapper\_config.m ファイルを開き、次の行を追加します。

```
this_block.addFile('encrypted_hdl_import.vhd','encrypted_hdl_import.vhd');
```

この行で、addFile 関数の 2 番目のパラメータにより 暗号化されたファイルに対して個別のファイルを生じ、統合 VHDL ネットリストには含めないように指定しています。次の図に、この行を追加した後の encrypted\_hdl\_import\_wrapper\_config.m ファイルを示します。



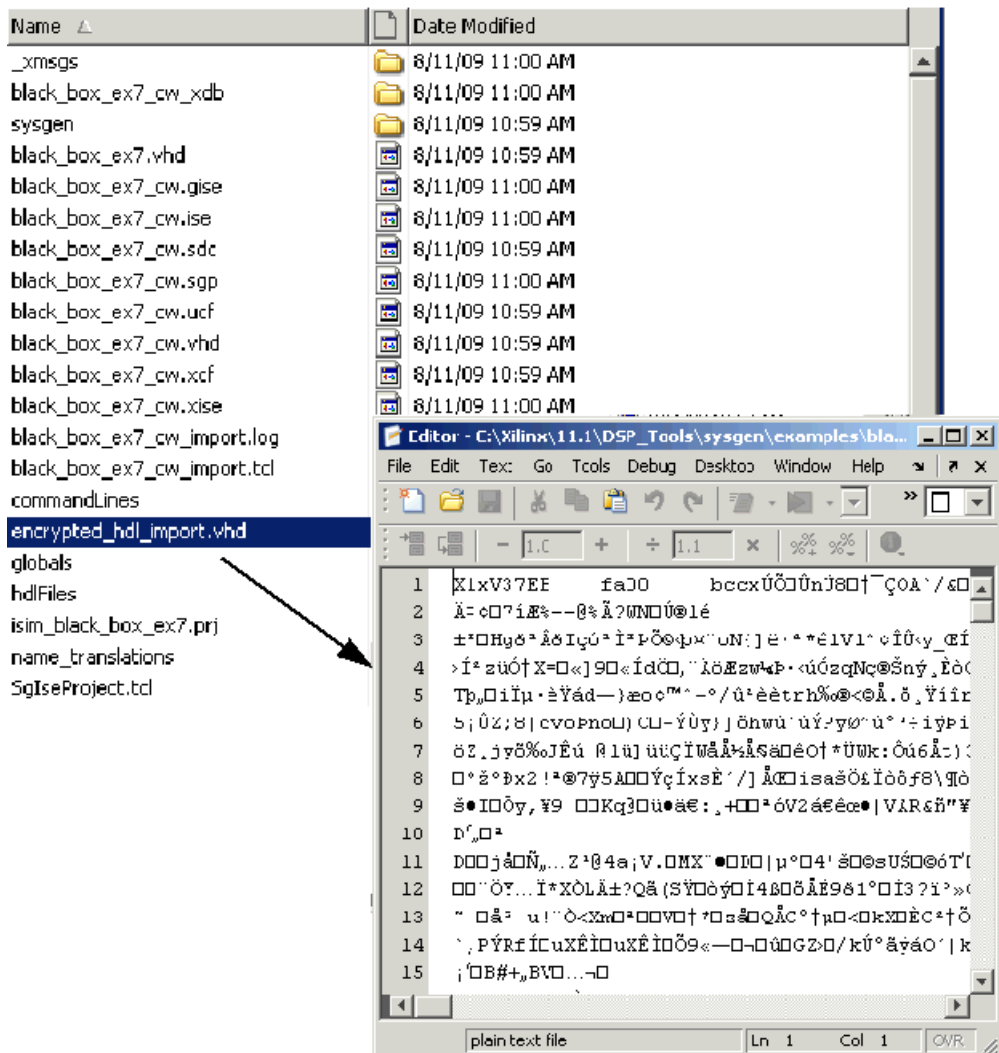
4. [シミュレーションの開始] をクリックしてデザインをシミュレーションします。  
次のようなシミュレーション結果が表示されます。



5. System Generator トークンをダブルクリックし、[Compilation] が [HDL Netlist] に設定されていることを確認して [Generate] をクリックします。

example7フォルダ内に hdl フォルダが作成されます。

6. hdl フォルダを開き、encrypted\_hdl\_import.vhd というファイルがあることを確認します。ファイルを開き、個別に作成された暗号化されたファイルであることを確認します。



**メモ：** encrypted\_hdl\_import.vhd はシミュレーション用です。このデザインのインプリメンテーション用のネットリストを生成するには、コンフィギュレーション ファイルに別の addFile 行を追加し、CORE Generator で作成された NGC ファイルを指定する必要があります。これを実行する例は、「ブラック ボックスの例 2: ブラック ボックスの HDL 要件を満たす VHDL ラップが必要な CORE Generator モジュールのインポート」を参照してください。



# System Generator のコンパイル タイプ

---

System Generator では、複数の方法でデザインを等価の下位表現にコンパイルできます。コンパイル方法は、System Generator トークンのパラメータ ダイアログ ボックスで指定します。複数のコンパイル方法が提供されているので、デザインの環境に応じて適切な表現を選択できます。たとえば、デザインをコンポーネントとして大型のシステムで使用する場合は、HDL または NGC ネットリストが適していますが、システム全体を System Generator でモデル化する場合は、FPGA コンフィギュレーション ビットストリームにコンパイルします。また、System Generator 外部のアプリケーションで特定の機能を実行する等価の高レベル モジュールにデザインをコンパイルする場合があります (ModelSim ハードウェア 協調シミュレーションなど)。

**HDL ネットリストへのコンパイル** System Generator では、デフォルトで [Compilation] が [HDL Netlist] に設定されています。HDL ネットリストコンパイル フローの詳細は、「[コンパイル結果](#)」を参照してください。

**NGC ネットリストへのコンパイル** デザインをスタンドアロン NGC ファイルにコンパイルする方法を説明します。

**ビットストリームへのコンパイル** デザインを FPGA コンフィギュレーション ビットストリームにコンパイルする方法を説明します。

**EDK Export Tool** デザインを選択したデバイスに適した FPGA コンフィギュレーション ビットストリームにコンパイルする方法を説明します。

**ハードウェア協調シミュレーション用のコンパイル** デザインを Simulink および ModelSim で使用可能な FPGA ハードウェアにコンパイルする方法を説明します。

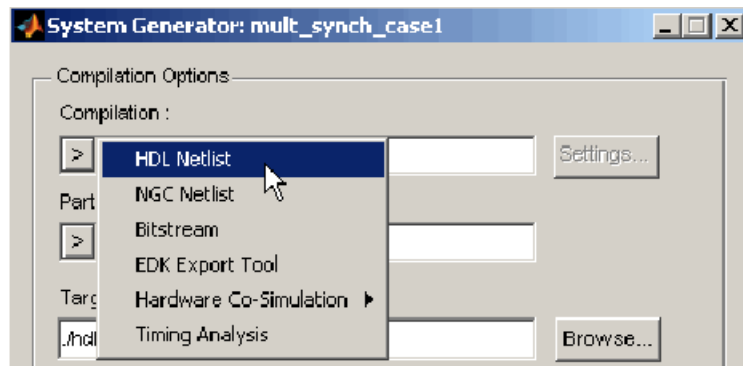
**タイミングおよび消費電力解析用のコンパイル** System Generator のタイミング解析ツールおよび消費電力解析ツールを使用する方法を説明します。

**コンパイル ターゲットの作成** System Generator トークンにカスタム コンパイル ターゲットを追加する方法を説明します。

## HDL ネットリストへのコンパイル

System Generator では、デフォルトで [Compilation] が [HDL Netlist] に設定されています。HDL ネットリスト コンパイル フローの詳細は、「[コンパイル結果](#)」を参照してください。

次の図に示すように、System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウン メニューから [HDL Netlist] を選択します。



## NGC ネットリストへのコンパイル

[Compilation] を [NGC Netlist] に設定すると、デザインをスタンドアロンのザイリンクス NGC バイナリ ネットリスト ファイルにコンパイルできます。System Generator で生成される NGC ネットリストには、デザインのロジックと制約に関する情報が含まれています。つまり、System Generator デザインに関する HDL、コア、制約ファイルが、1 つのファイルに含まれています。

デザインにクロック ラップ ロジックを含める場合、ネットリスト ファイルは <design>\_cw.ngc という名前で保存されます。クロック ラップ ロジックを含めない場合は、ファイル名は <design>.ngc になります。ここで、<design> はコンパイルするデザイン部分から得られた名前です。このファイルは、モジュールとして大型のデザインで使用するか、完全なデザインである場合は NGDBuild の入力として使用できます System Generator デザインを大型デザインのコンポーネントとして使用する例は、「[System Generator デザインの大型システムへのインポート](#)」を参照してください。

コンパイル ターゲットを NGC ネットリストにした場合、HDL コンポーネント インスタンス化 ション テンプレートが作成され、System Generator デザインをコンポーネントとして大型デザインに簡単に組み込むことができます。VHDL のテンプレートは、クロック ラップを含める場合は <design>\_cw.vho に保存され、クロック ラップを含めない場合は <design>.vho に保存されます。Verilog のテンプレートでは、拡張子は .veo です。インスタンス化 テンプレートは、デザインのターゲット ディレクトリに保存されます。

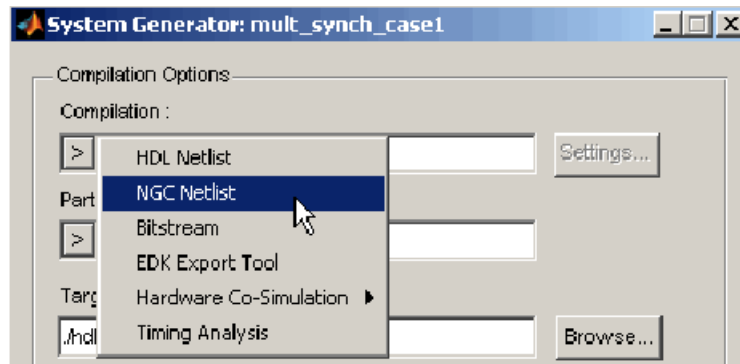
System Generator では、コンパイルで次の処理を実行して NGC ネットリスト ファイルを生成します。

1. 選択された合成ツールを実行して、下位ネットリストを生成します。ネットリストのタイプは、選択された合成ツールによって異なります (Synplify または Synplify Pro の場合は EDIF、XST の場合は NGC)。

**メモ:** 合成中、デザインに I/O バッファは挿入されません。

2. 合成結果、コアのネットリスト、ブラック ボックスのネットリスト、およびオプションで制約 ファイルを統合して、1 つの NGC ファイルを生成します。

次の図に示すように、System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウン メニューから [NGC Netlist] を選択します。



[Compilation] を [NGC Netlist] に設定した場合、[Settings] ボタンをクリックして NGC ネットリスト専用の追加のパラメータを設定できます。[Settings] ボタンをクリックして開く [NGC Netlist Settings] ダイアログ ボックスには、次のパラメータが含まれています。

- **[Include Clock Wrapper]** : オンにすると、NGC ネットリスト ファイルにデザインのクロックラップが含まれます。クロック ラップの詳細は、「[コンパイル結果](#)」を参照してください。  
**メモ** : マルチレート デザインにクロック ラップを含めない場合は、最上位デザインの適切な信号でクロック イネーブル ポートを駆動する必要があります。
- **[Include Constraints File]** : オンにすると、NGC ネットリスト ファイルにデザインに関連付けられた制約ファイルが含まれます。  
**メモ** : 制約ファイルを含めない場合、System Generator デザインの複数サイクル パスが適切に制約されるよう、独自の制約を設定する必要があります。

## ビットストリームへのコンパイル

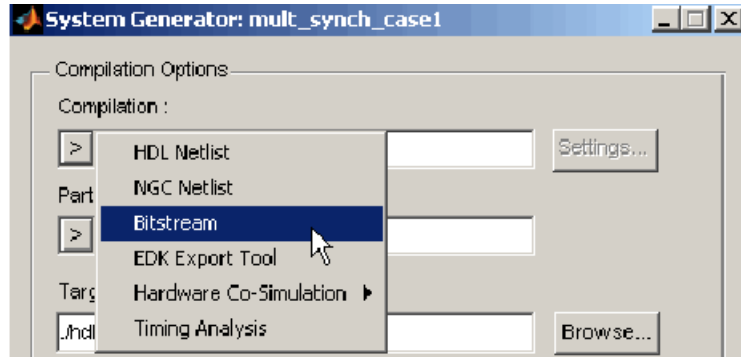
[Compilation] を [Bitstream] に設定すると、System Generator トークンのパラメータ ダイアログ ボックスで選択した FPGA デバイス用のザイリンクス コンフィギュレーション ビットストリーム ファイルにデザインがコンパイルされます。ビットストリーム ファイルは、<design>\_cw.bit (<design> はコンパイルされたデザインから取得された名前) という名前で、デザインのターゲット ディレクトリに保存されます。

System Generator では、コンパイルで次の処理を実行してビットストリーム ファイルを生成します。

1. デザインの HDL ネットリストを生成します。
2. 選択された合成ツールを実行して、下位ネットリストを生成します。ネットリストのタイプは、選択された合成ツールによって異なります (Synplify または Synplify Pro の場合は EDIF、XST の場合は NGC)。
3. XFLOW を実行して、コンフィギュレーション ビットストリームを生成します。



次の図に示すように、System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウン メニューから [Bitstream] を選択します。



System Generator では、XFLOW を実行してコンフィギュレーション ビットストリームの生成に必要なツールを実行します。XFLOW では、インプリメンテーションとコンフィギュレーションの 2 つのフローが実行されます。

インプリメンテーション フローでは、合成ツールのネットリスト出力 (EDIF または NGC) が配置配線された NCD ファイルにコンパイルされます。次の処理が実行されます。

1. NGDBuild を使用して、合成結果、コアのネットリスト、ブラック ボックスのネットリスト、および制約ファイルを統合します。
2. MAP、PAR、TRACE をこの順に実行します。

コンフィギュレーション フローでは、配置配線された NCD ファイルを入力として使用して、FPGA ビットストリームを作成するのに必要なツール (BitGen など) が実行されます。

## XFLOW のオプション ファイル

インプリメンテーション フローとコンフィギュレーション フローには、それぞれ XFLOW オプション ファイルが関連付けられています。XFLOW オプション ファイルでは、そのフローで実行する必要があるプログラムとコマンドライン オプションが定義されます。ザイリックス ISE® ソフトウェアに、XFLOW オプション ファイルの例が含まれています。これらの ファイルは、ISE のインストール ディレクトリの xilinx\data ディレクトリに含まれています。よく使用されるインプリメンテーション オプション ファイルは、次のとおりです。

- balanced.opt
- fast\_runtime.opt
- high\_effort.opt

**メモ** : デフォルトでは、インプリメンテーション フローに balanced.opt ファイルが使用され、コンフィギュレーション フローに bitgen.opt ファイルが使用されます。

場合によって、デフォルトのオプション ファイルとは異なる設定を使用することがあります (PAR の配置エフォート レベルを上げるなど)。そのような場合は、独自のオプション ファイルを作成するか、デフォルトのオプション ファイルを変更します。System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] を [Bitstream] に設定して [Settings] をクリックすると、オプション ファイルを指定できます。

## 追加の設定

[Compilation] を [Bitstream] に設定した場合、[Settings] ボタンをクリックしてビットストリーム専用の追加のパラメータを設定できます。[Settings] ボタンをクリックして開く [Compilation Target Settings] ダイアログ ボックスには、次のパラメータが含まれています。

- [Import Top-level Netlist] : オンにすると、デザインの System Generator 部分をモジュールとして含める最上位ネットリストを指定できます。System Generator クロック ラップをコンポーネントとしてインスタンス化して大型デザインがある場合は、その最上位ネットリストを選択できます。クロック ラップの詳細は、「[コンパイル結果](#)」を参照してください。この最上位ネットリストは、コンパイル時に生成されるビットストリーム ファイルに含まれます。このチェック ボックスをオンにすると、[Top-level Netlist File (EDIF or NGC)] と [Search Path for Additional Netlist and Constraint Files] が有効になります。
  - ◆ [Top-level Netlist File (EDIF or NGC)] : コンパイルで含める最上位ネットリスト ファイルの名前と場所を指定します。最上位ネットリストで使用される HDL コンポーネント (最上位自身も含む) がネットリスト ファイルに合成されていることが必要です。
  - ◆ [Search Path for Additional Netlist and Constraint Files] : 最上位ネットリスト ファイルに関連した追加のネットリスト ファイルおよび制約ファイルを検索するディレクトリを指定します。このディレクトリを指定すると、すべてのネットリスト ファイル (EDN、EDF、NGC など) および制約ファイル (UCF、XCF、NCF など) が implementation ディレクトリにコピーされます。ディレクトリを指定しない場合は、[Top-level Netlist File (EDIF or NGC)] で指定されたネットリスト ファイルのみがコピーされます。
- [Specify Alternate Clock Wrapper] : オンにすると、System Generator で生成されたクロック ラップの代わりに独自のクロック ラップを指定できます。クロック ラップは System Generator デザイン用に作成された最上位 HDL ファイルのレベルであり、デザインのクロック 信号およびクロック イネーブル信号を駆動します。デザインに複数のクロック 信号がある場合や、デザインでインターフェイスとして使用するボード 専用のハードウェアがある場合など、独自のクロック ラップを使用する必要がある場合があります。

メモ : 独自のクロック ラップ ファイルを使用する場合は、ファイル名を <design>\_cw.vhd または <design>\_cw.v にしないと、ビットストリームの生成で使用されません。
- [XFLOW Option Files] : デザインを System Generator ハードウェア協調シミュレーション用にコンパイルすると、XFLOW というコマンド ライン ツールを使用して、デザインが選択した FPGA プラットフォーム用にインプリメント およびコンフィギュレーションされます。XFLOW では、コンパイルでデザインに実行する必要があるプログラムのシーケンスを指定するさまざまなフローが定義されています。必要な出力 (ハードウェア協調シミュレーションの場合はコンフィギュレーションビットストリーム) を得るためには、通常複数のフローを実行する必要があります。
  - ◆ [Implementation Phase (NBDBuild, MAP, PAR, TRACE)] : インプリメンテーション フローで使用するオプション ファイルを指定します。デフォルトでは、コンパイル ターゲットに提供されているインプリメンテーション オプション ファイルが使用されます。
  - ◆ [Configuration Phase (BitGen)] : コンフィギュレーション フローで使用するオプション ファイルを指定します。デフォルトでは、コンパイル ターゲットに提供されているコンフィギュレーション オプション ファイルが使用されます。

## EDK Processor ブロックに含まれるソフトウェア プログラムのビットストリームへの再コンパイル

EDK Processor ブロックを含む System Generator デザインのビットストリームをコンパイルすると、インポートされた EDK プロジェクトと、System Generator デザインと MicroBlaze™ プロセッサの間にある共有メモリがネットリスト処理され、ビットストリームに含まれます。

System Generator では、インポートされた EDK プロジェクトに含まれるアクティブ ソフトウェア プログラムもコンパイルされます。アクティブ ソフトウェア プログラムのコンパイルが正常に完了すると、data2bram ユーティリティが起動され、コンパイルされたソフトウェア プログラムがビットストリームに含まれます。

**メモ :** ソフトウェア プログラムのコンパイル中またはビットストリームをコンパイルされたソフトウェア プログラムでアップデートする際にエラーが発生しても、エラー メッセージまたは警告メッセージは表示されません。

インポートされた EDK プロジェクトでソフトウェア プログラムを変更した場合、次のコマンドを使用してソフトウェア プログラムをコンパイルし、System Generator ビットストリームをコンパイルされたソフトウェア プログラムでアップデートできます。

```
xlProcBlockCallbacks('updatebitstream', [], xmp_file, bit_file,
bmm_file);
```

**xmp\_file :** インポートされた EDK プロジェクト ファイルへのパス名

**bit\_file :** System Generator のビット ファイルへのパス名

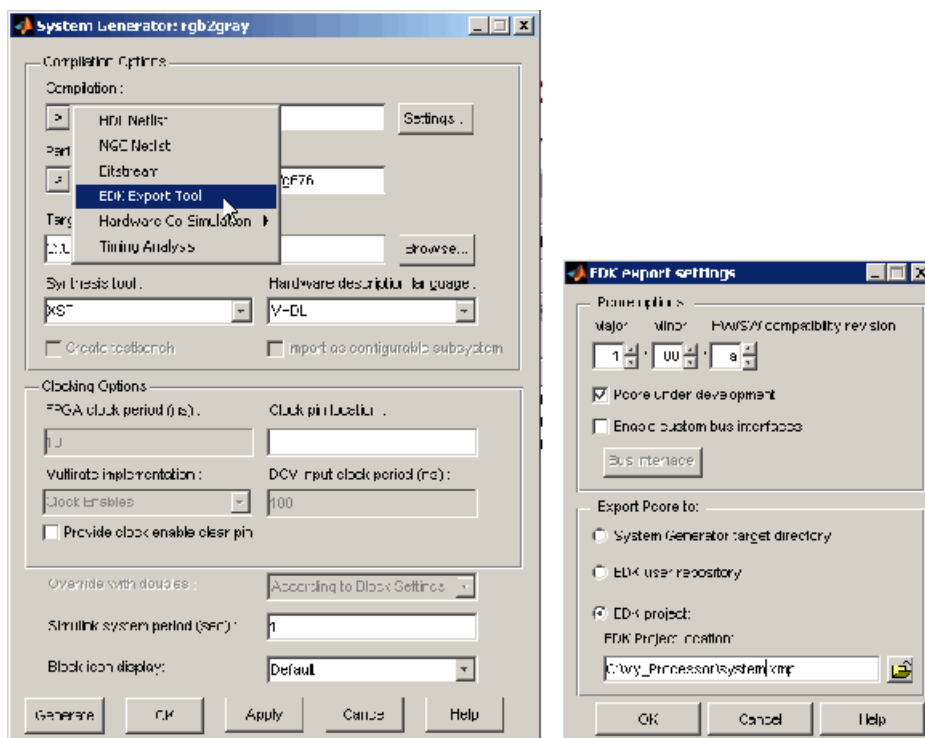
**bmm\_file :** ビットストリームのコンパイル時に System Generator で生成されるバックアノテートされた BMM ファイルへのパス名

インポートされた EDK プロジェクトに `imported_edk_project.bmm` という 名前の BMM ファイルが含まれている場合、System Generator により `imported_edk_project_bd.bmm` という 名前のバックアノテートされた BMM ファイルが生成されます。ビットストリームを正しくアップデートするには、上記のコマンドで最新のバックアノテートされた BMM ファイルを指定する必要があります。

## EDK Export Tool

EDK Export Tool を使用すると、System Generator デザインを[ザイリンクス エンベデッド開発キット EDK](#) プロジェクトにエクスポートできます。このツールは、EDK に必要なファイルを自動的に生成することにより、ペリフェラルの作成プロセスを簡略化します。

次の図に示すように、System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウンメニューから [EDK Export Tool] を選択します。[EDK Export Tool] を選択すると、[Settings] ボタンが有効になります。



[Settings] をクリックすると、[EDK export settings] ダイアログ ボックスが開きます。

このダイアログ ボックスの [pcore options] では、次を設定できます。

- pcore のバージョン番号
- [Pcore under development] オプション

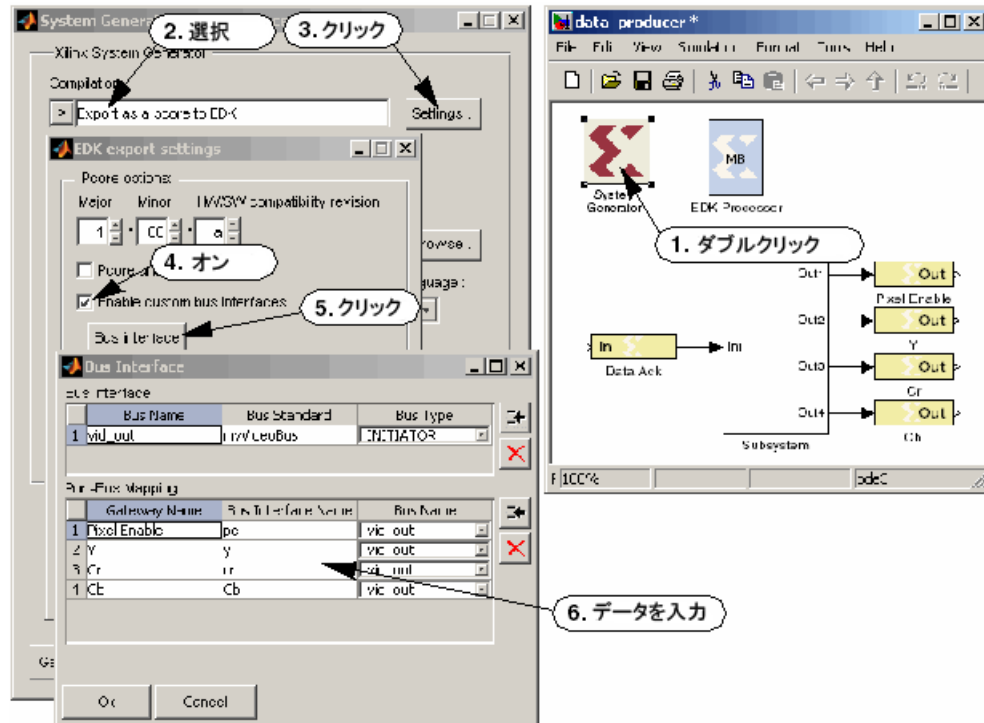
このオプションは、FSL ベースおよび PLB ベースの pcore のエクスポートで設定できます。このオプションをオンにすると、この pcore に対して生成される HDL はキャッシュされません。この機能は、System Generator で pcore を開発中に XPS でテストする場合に有益です。このチェック ボックスをオンにし、System Generator で変更を加え、XPS でコンパイルすると、常に生成された pcore がコンパイルされ、XPS キャッシュを空にする必要はありません (キャッシュにはほかのペリフェラルも含まれているので、空にしてしまうと最終的なビットストリームのコンパイル時間が長くなる)。

- [Enable custom bus interfaces] オプション

このオプションは、FSL ベースおよび PLB ベースの pcore のエクスポートで設定できます。XPS で認識可能なカスタム バス インターフェイスを使用できるようにします。

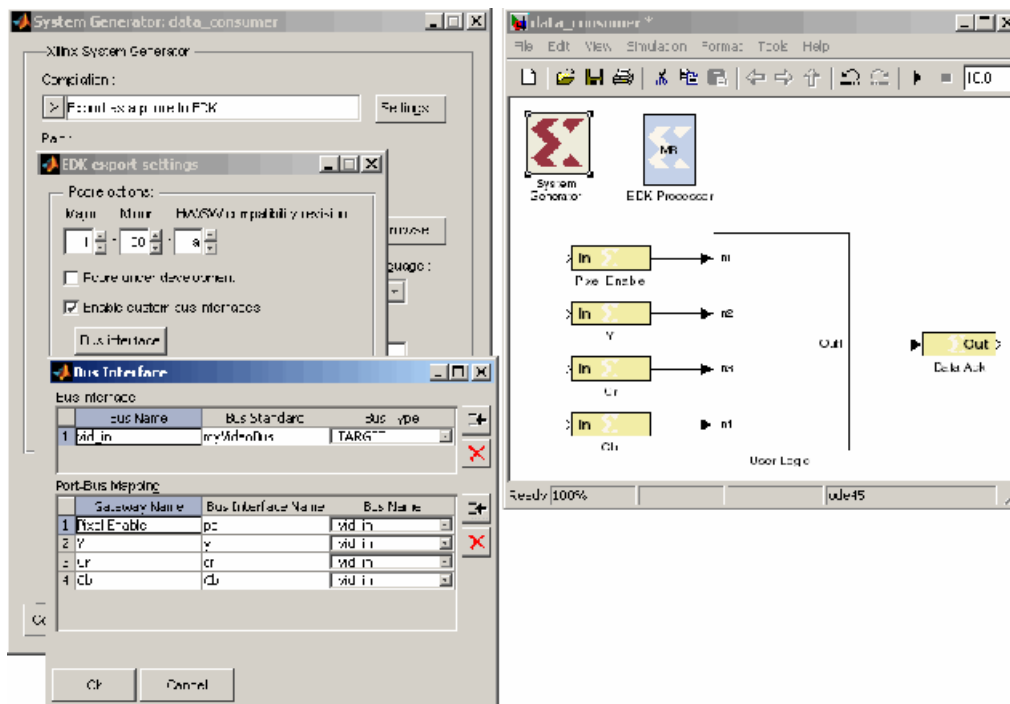
## pcore エクスポート用のカスタム バス インターフェイスの作成

次の図のモデルに示すように、XPS に pcore としてエクスポートするデザインが 1 つあるとします。このデザインには、Pixel Enable、Y、Cr、Cb という出力ポートがあります。これらの信号をバスにまとめ、XPS への接続を簡略化します。



上記の手順に従い、[Bus Interface] ダイアログ ボックスを開きます。このダイアログ ボックスで、名前が vid\_out、バス規格が myVideoBus、バス タイプが INITIATOR の新しいバス インターフェイスを定義します。バス タイプには、INITIATOR 以外に TARGET、MASTER、SLAVE、MASTER\_SLAVE、MONITOR を指定できます。次に、[Port-Bus Mapping] 表でバスに必要なすべてのゲートウェイを入力し、それぞれバス インターフェイス名を指定します。バス インターフェイスの定義を完了したら、デザインを pcore としてエクスポートします。この pcore バスには出力が含まれているので、INITIATOR と定義されていることに注意してください。

次の図に示す別のモデルで、対応する入力ゲートウェイを作成します。このバスには、バス規格を同じく myVideoBus、バス タイプを TARGET に設定します。XPS では、バス規格名を使用して異なるバス インターフェイスを一致させ、バス インターフェイス名が同じ入力と出力を接続します。



この pcore を XPS プロジェクトとしてエクスポートします。これら 2 つの pcore を同じ XPS プロジェクトで使用すると、XPS でバスに互換性があることが認識されるので、必要に応じて接続できます。

## pcore として EDK にエクスポート

System Generator デザインを EDK にエクスポートすると、PLC v6.4 バスを指定した場合、pcore (プロセッサ コア) 名はモデル名に「\_plbw」を追加したものになります。たとえば、mul\_accumulate というモデルを EDK にエクスポートすると、EDK での名前は mul\_accumulate\_plbw となります。高速シンプレックス リンクを指定した場合は、pcore 名はモデル名に「\_sm」を追加したものになります。

次の表に、System Generator で生成される pcore のサブディレクトリ構造を示します。

| ディレクトリ  | 説明                                                                                                                                                                                                                                                                                                                                                                                            |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| data    | <p>BBD、PAO、MPD、TCL の 4 つのファイルが含まれます。</p> <ul style="list-style-type: none"> <li>• BBD (Black Box Definition) ファイル : デザインで使用する EDN または NGC ファイルを定義します。</li> <li>• PAO (Peripheral Analyze Order) ファイル : HDL ファイルの解析順を定義します。</li> <li>• MPD (Microprocessor Peripheral Description) ファイル : ペリフェラルのプロセッサへの接続方法を定義します。</li> <li>• TCL ファイル : ペリフェラルのソフトウェア ドライバを生成する際に Libgen で使用されます。</li> </ul> |
| doc     | HTML 形式の文書ファイルが含まれます。                                                                                                                                                                                                                                                                                                                                                                         |
| hdl     | System Generator で生成された HDL ファイルが含まれます。                                                                                                                                                                                                                                                                                                                                                       |
| netlist | BBD ファイルにリストされている EDN ファイルおよび NGC ファイルが含まれます。                                                                                                                                                                                                                                                                                                                                                 |
| src     | ソフトウェア ドライバのソース ファイルが含まれます。                                                                                                                                                                                                                                                                                                                                                                   |

## System Generator ポートを EDK の最上位ポートとして使用

System Generator で作成された入力ポートと出力ポートは、EDK でペリフェラルのポートとして使用できます。これらのポートは、EDK デザインの最上位ポートとして使用できます。これは、System Generator デザインに FPGA デバイス上の入力/出力パッドに接続されるポートがある場合に便利です。

## サポートされるプロセッサと制限

現在のところ、EDK Export Tool を使用してエクスポート できるのは MicroBlaze プロセッサへの PLB v4.6 メモリ マッピングおよび FSL メモリ マップで、EDK Processor ブロックのインスタンスを 1 つのみ含むことができます。

## 関連項目

[EDK Processor](#)

## ハードウェア協調シミュレーション用のコンパイル

System Generator では、デザインを Simulink シミュレーションとの協調シミュレーションで使用可能な FPGA ハードウェアにコンパイルできます。この機能については、「[ハードウェア協調シミュレーションの使用](#)」を参照してください。

System Generator トークンのパラメータ ダイアログ ボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウン メニューで [Hardware Co-Simulation] の下にあるリストからハードウェア協調シミュレーション プラットフォーム を選択します。選択可能なプラットフォームは、システムにインストールされているハードウェア協調シミュレーション プラグインによって異なります。

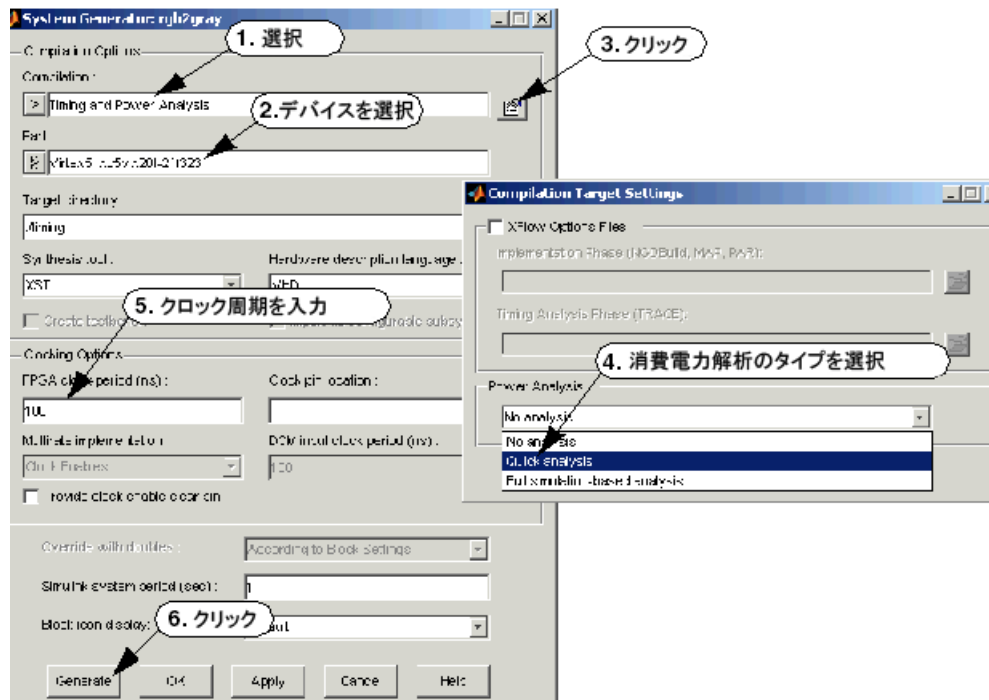


メモ：コンパイルターゲットとしてリストされていない FPGA プラットフォームがある場合は、FPGA ハードウェアと JTAG を介して通信する新しいコンパイルターゲットを作成できます。この方法については、「[新規プラットフォームのサポート](#)」を参照してください。

## タイミングおよび消費電力解析用のコンパイル

System Generator で生成されたハードウェアがタイミング要件を満たさない場合があります。System Generator では、タイミングおよび消費電力の問題を解決するのに役立つタイミングおよび消費電力解析ツールフローが提供されています。このツールを使用すると、グラフィックおよびテキスト形式の両方で、最も遅いパス、タイミング要件を満たしていないパスが表示されるので、これらの情報を利用してデザインを調整できます。このセクションでは、この方法を説明します。System Generator のタイミング解析ツールは、ISE ソフトウェアでタイミング解析に使用される TRACE というソフトウェアアプリケーションに基づいています。

次の図に示すように、System Generator トークンのパラメータダイアログボックスで [Compilation] のボックスの左側にある [>] ボタンをクリックし、プルダウンメニューから [Timing and Power Analysis] を選択します。オプションの消費電力解析オプションと使用するデバイスを指定します。デバイスのサイズおよびスピードがパス遅延に影響するので、正しいデバイスを選択する必要があります。解析結果のファイルは、[Target Directory] で指定されたディレクトリに保存されます。[FPGA Clock Period] の値が、配置配線中に使用されます。



パラメータダイアログボックスでパラメータを指定して [Generate] ボタンをクリックすると、System Generator が次の処理を実行します。

1. Simulink を使用してデザインをコンパイルし、HDL ソースを生成します。
2. 消費電力解析オプションとして [Full simulation-based analysis] を選択した場合、ISim シミュレータを呼び出して HDL デザインをシミュレーションします。その後 [Synthesis tool] で指定した合成ツールを呼び出し、HDL を EDIF (Synplify/Synplify Pro) または NGC (XST) ネットリストに変換します。



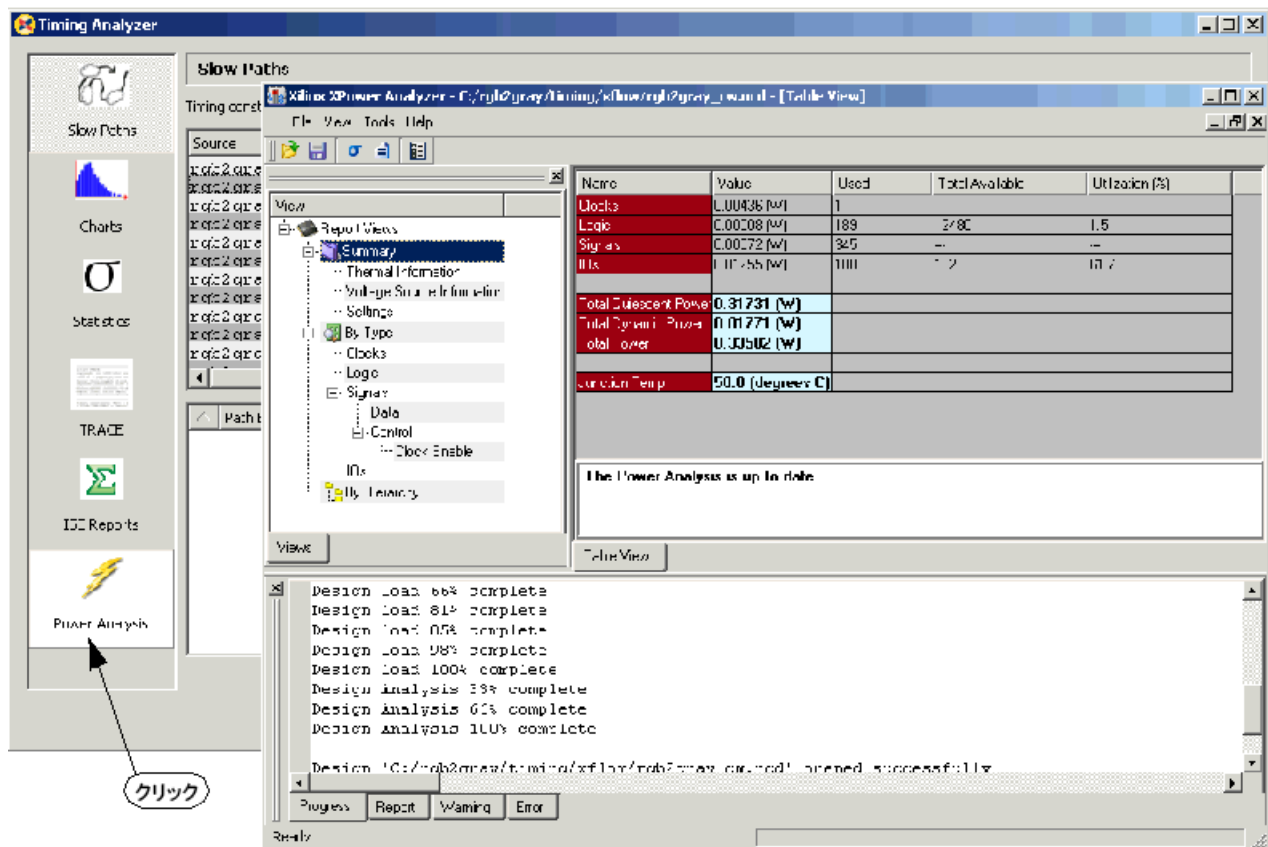
3. NGDBuild を呼び出し、ネットリストを NGD ファイルに変換します。その後 ISE の MAP を呼び出し、ロジックのエレメントをスライスにマップし、NCD ファイルを生成します。
4. ISE の PAR を呼び出し、スライスおよびエレメントをザイリンクス デバイスに配置し、スライス間を配線します。これらの情報は、別の NCD ファイルに保存されます。
5. ISE の TRACE を呼び出し、PAR で生成された NCD を解析してワースト スラックのパスを見つけ、TRACE レポートを作成します。System Generator のタイミング解析ツールを表示し、TRACE レポートのデータを表示します。

メモ：この方法を使用してタイミング データを生成し、後でこのデータを表示する場合は、MATLAB の [Command Window] に次のコマンドを入力します。

```
>>xlTimingAnalysis('timing')
```

ここで、timing は以前のタイミング解析結果が保存されているディレクトリです。

6. [Timing Analyzer] ウィンドウで [Power Analysis] ボタンをクリックすると、ザイリンクス XPower 解析ツールでレポートが表示されます。

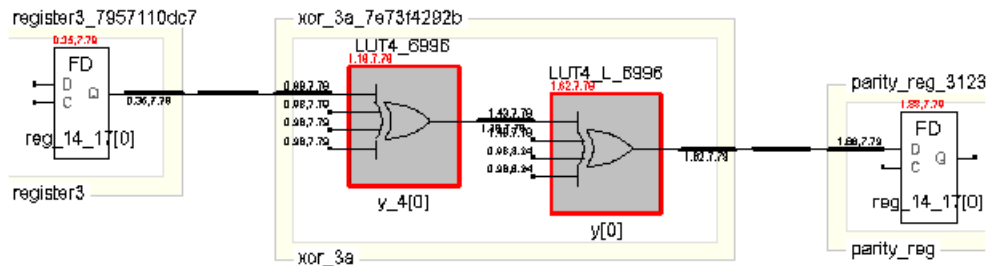


## タイミング解析の概要

このセクションでは、ロジック パス解析の概要を説明します。

### 周期とスラック

タイミングが満たされないということは、通常デザインにセットアップ タイム違反があることを意味します。セットアップ タイム違反は、ある信号が 1 つの同期エレメント の出力から別の同期エレメント の入力に要求されたクロック 周期内に到達することができず、2 番目の同期エレメント のセットアップ タイム要件が満たされない状況です。次の図に、典型的なパスを示します。



この図では、左側のレジスタ (register3) の Q 出力から右側のレジスタ (parity\_reg) の D 入力までのパスを示しています。このパスは、4 入力 XOR ゲートとしてコンフィギュレーションされている 2 つの LUT を通過します。ロジック レベル数は 2 で、2 つの組み合わせエレメント (LUT) を通過することを意味します。

このパスに要求される周期は 10ns で、タイミングを簡単に満たすことができます。各ロジック エレメントの上に表示されているカンマで区切られた赤色の 2 つの数値のうち 2 番目の値は、パスのスラックを示します。スラックは、パスがタイミングをどの程度満たしているかを示します。上図の例のスラックは 7.79ns で、パスが 7.79ns 遅くても 10ns の周期要件が満たされることを意味します。負のスラック値は、パスがタイミングを満たしておらず、セットアップ (またはホールド) タイム違反が発生していることを示します。

### パス解析の例

ここで、このパスを詳細に調べてみます。register3 の上に示されている最初の値は 0.35ns です。これは、このレジスタの clock-to-out タイムが 0.35ns であることを示しており、クロック信号の立ち上がりエッジの 0.35ns 後に Q にデータが出力されます。クロック信号は、この図には示されていませんが、両方のレジスタの C 入力を駆動します。

LUT y\_4[0] の各入力には、2 つの値が表示されています。最初の値 0.98ns は信号の到着時間です。これは、クロックの立ち上がりエッジの 0.98ns 後に信号が入力に到着するということなので、ネット遅延は  $0.98\text{ns} - 0.35\text{ns} = 0.63\text{ns}$  です。パス遅延は、ネット遅延とロジック遅延に分けることができます。FPGA では、通常ネット遅延がパス遅延の大部分を占めます。これは、FPGA のコンフィギュラブル配線構造のため、ネットがデスティネーションに到達するために多数のスイッチ ボックスを通過する必要があるからです。

y\_4[0] からのパスは、y[0] への別のネットを通過します。y[0] の出力に示されている 2 つの数値のうち最初の値 1.62ns は、この LUT の出力に信号が到着する時間を示しています。この後信号は最後のネットを 0.26ns のネット遅延で通過し、parity\_reg の D 入力にクロック エッジの 1.88ns 後に到着します。このレジスタのセットアップ タイム要件は 0.33ns です。これは、次のクロックの立ち上がりエッジの 0.33ns 前に、信号が D 入力に到着する必要があるということです。そのため、このパスに必要な合計時間は  $1.88\text{ns} + 0.33\text{ns} = 2.21\text{ns}$  です。10ns からこの値を引くと、7.79ns のスラック値が得られます。

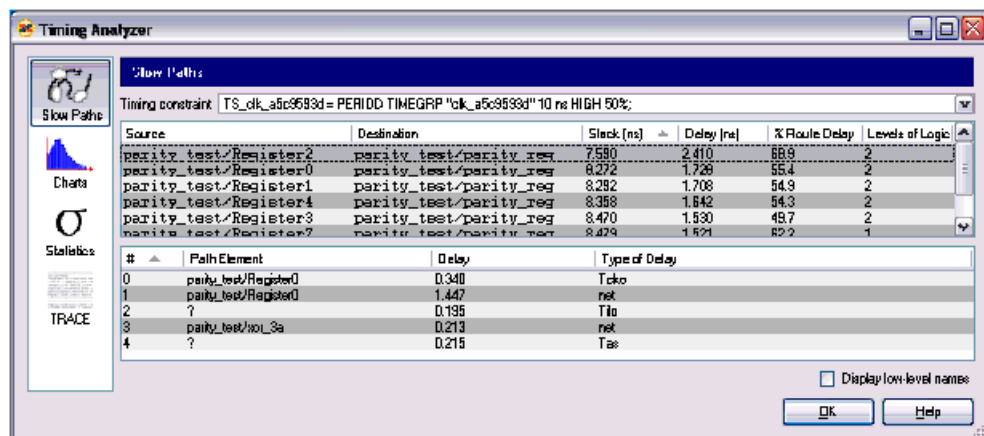
## クロック スキューとジッタ

ここに示されているネット遅延は、Synplify による予測値です。実際のネット遅延は配置配線後でないと確定できないので、合成ツールでは実際の遅延は判断できません。実際のパスでは、クロック スキューやクロック ジッタなど、その他の要素も考慮する必要があります。クロック スキューは、1 つのクロック信号が複数のデスティネーション同期エレメントに到着する時間の差です。クロック ジッタは、サイクル間のクロック周期の変動を示します。ジッタは、DCM (デジタル クロック マネージャ) やその他の要因で発生します。タイミング解析は、デバイスのワースト ケースの遅延値、ジッタ、スキュー、および温度を使用して行われます。

## タイミング解析ツールの機能

### 遅いパスの表示

[Slow Paths] をクリックすると、各タイミング制約に対してスラックが最小のパスが表示されます。次の図は、その例です。



このウィンドウの上部には遅いパスのリストが表示され、下部には選択したパスの詳細が表示されます。次に、このウィンドウの各項目を説明します。

- [Timing constraint] : すべてのタイミング制約のパスを表示するか、1 つの制約のパスを表示するかを選択します。典型的な System Generator デザインでは、システム クロックの周期を定義するタイミング制約が 1 つ定義されており、この例ではその制約のパスが表示されています。TS\_clk\_a5c9593d は制約名で、複数の System Generator デザインをコンポーネントとして大型のデザインで使用する際に固有の名前になるようにするため、最後にハッシュ値が追加されています。clk\_a5c9593 は同期ロジックのタイミング グループで、同様にハッシュ値が追加されています。この例でのタイミング グループには、デザインのすべての同期エレメントが含まれます。クロック周期は 10ns で、デューティ サイクルは 50% です。
- [Source] : パスを駆動する System Generator ブロックを示します。
- [Destination] : パスをデスティネーションとなる System Generator ブロックを示します。
- [Slack] : パスのスラックを示します。詳細は、「周期とスラック」を参照してください。
- [Delay] (パス) : セットアップ タイム要件を含むパス全体の遅延を示します。
- [% Route Delay] : 遅延全体に対する配線 (ネット) 遅延の割合を示します。残りの遅延はロジック遅延です。
- [Levels of Logic] : パスの組み合わせロジックのレベル数を示します。組み合わせロジックには、LUT、F5MUX、キャリー チェーン マルチプレクサなどがあります。

- **[Path Element]** : 選択されたパスのロジックとネット エレメントを示します。
- **[Delay] (エレメント)** : 選択されたパスでロジックおよびネット エレメント通過する際に発生する遅延を示します。
- **[Type of Delay]** : パス エレメントで発生する遅延のタイプを示します。これらの値は、ザイリンクス デバイスのデータシートで定義されています。上図の例では、**Tcko** はフリップフロップの **clock-to-out** タイム、**net** はネット遅延、**Tilo** は LUT を通過する際の遅延、**Tas** はフリップフロップのセットアップ タイムです。

列ヘッダをクリックすると、データをその列を基準に並べ替えることができます。タイミングを満たしていないパスは、赤/ピンクでハイライトされます。

#### 名前の関連付けと下位レベル名の表示

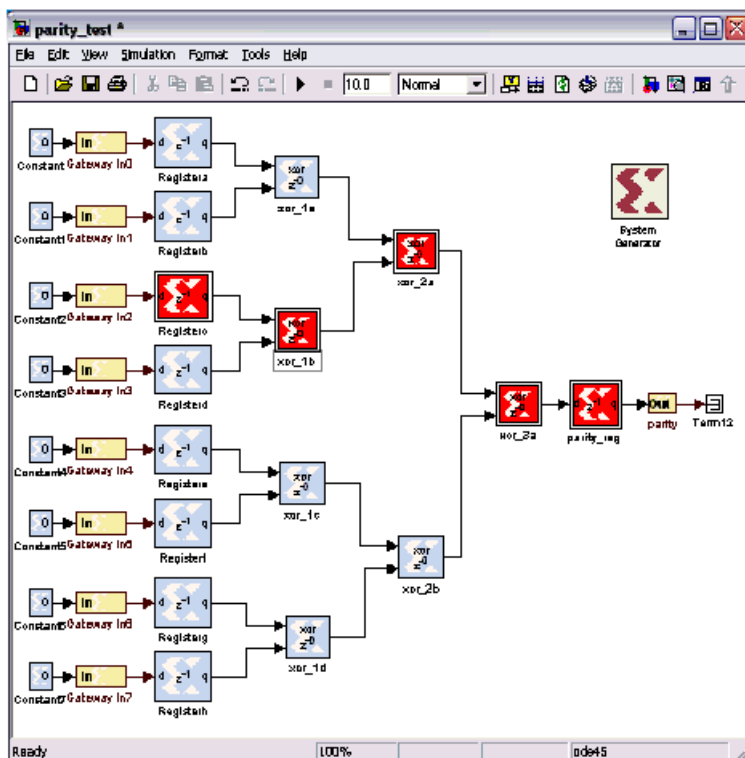
System Generator のタイミング解析ツールには、System Generator コンポーネントをザイリンクスインプリメンテーション ツールで生成された下位コンポーネント名と自動的に関連付ける非常に有益な機能があります。これらのコンポーネント名は通常大きく異なっており、System Generator モデルに表示されるロジック ブロックとワイヤと、合成で生成される実際にロジックとには多少の関係がある程度です。System Generator のタイミング解析ツールでは、TRACE レポートに表示されるロジック エレメントおよびネットの名前と System Generator モデルのブロックとワイヤを関連付ける必要があります。

このプロセスで、関連付けが判断できない場合もあります。先ほどの図で、パス エレメント #2 と #4 の **[Path Element]** にはクエスション マークが表示されています。これは、タイミング解析ツールで TRACE レポートのエレメント名と System Generator のブロックを関連付けることができなかったということです。

TRACE レポートの実際の名前を表示するには、**[Display low-level names]** をオンにします。TRACE レポートの名前が表示されます。これにより、TRACE レポートの名前と System Generator エレメントを関連付けることができる場合があります。

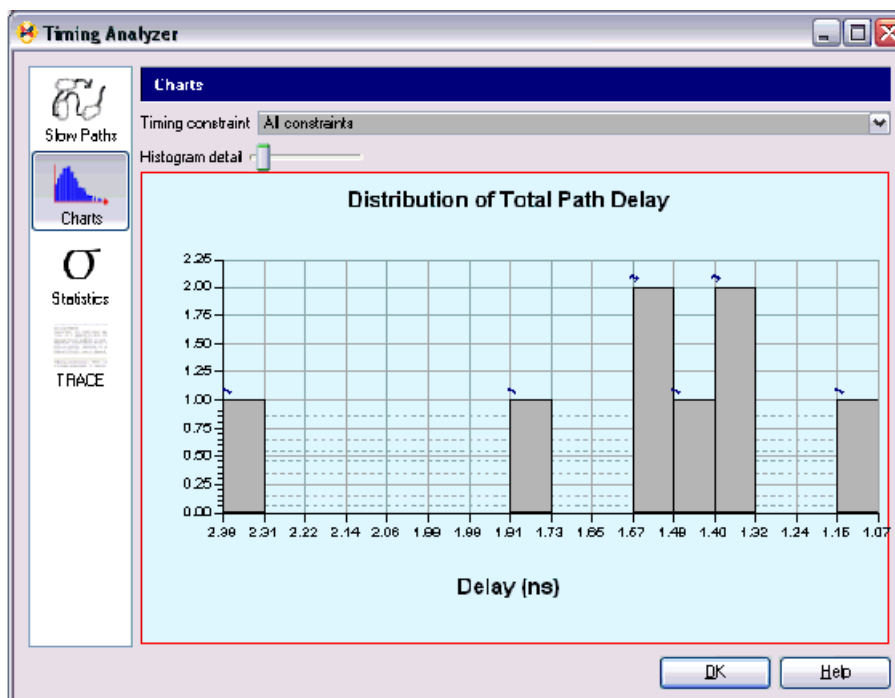
## クロスプロローブ

[Slow Paths] ビューでパスを選択すると、System Generator モデルでパス上のブロックがハイライトされます。パスのソース ブロック、デスティネーション ブロック、パスが通過する組み合わせブロックが赤でハイライトされます。次の図に、ソースが Registerc でデスティネーションが parity\_reg のパスを選択したときに、モデルがどのように表示されるかを示します。パスの一部である xor\_1b、xor\_2a、xor\_3a もハイライトされます。



## ヒストグラム

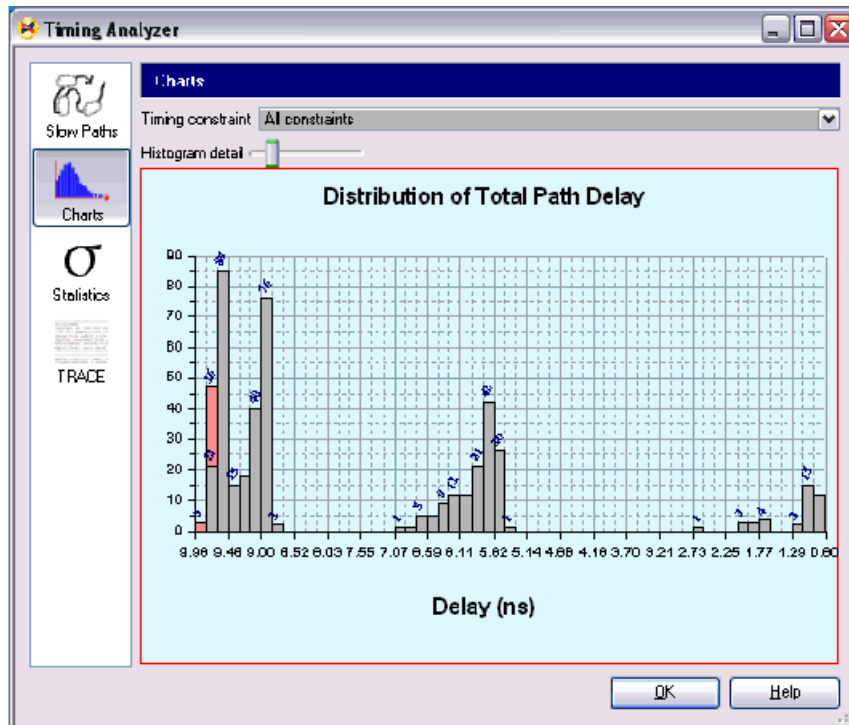
[Charts] をクリックすると、遅いパスのヒストグラムが表示されます。このヒストグラムは、デザインの解析に便利です。たとえば、100MHz で動作する必要があるデザインが 99MHz で動作しているとします。タイミング要件にどれだけ近いのか、要件を満たすためにはどの程度の作業が必要かを、ヒストグラムから予測できます。たとえば、単純なデザインに対して次のようなヒストグラムが示されたとします。



このヒストグラムから、遅いパスのほとんどが 1.5ns 周辺に集中しており、最も遅いパスは 2.35ns であることがわかります。ヒストグラムの縦軸は、遅延範囲内のパスの数を示します。遅延が 2.31ns ~ 2.39ns の範囲内のパスは 1 つだけで、その右側の範囲 (1.81ns ~ 2.31ns) にはパスはありません。つまり、最も遅いパスのみが離れていて、たとえばタイミング要件が周期 2ns であるとする、このパスの遅延を削減するだけでタイミング要件を満たすことができます。

## ヒストグラムの詳細

[Histogram Detail] スライダー バーを使用すると、ヒストグラムの表示幅を調整でき、必要に応じて詳細を表示できます。次の図に、遅延の範囲が広い別のデザインのヒストグラムを示します。



このヒストグラムでは、パスが3つの領域にグループ化されており、それぞれおおよそ釣鐘型の曲線となっています。これらのグループは回路の異なる部分からのものか、異なるクロック領域からの異なるタイミング制約のものであると考えられます。1つのタイミング制約のパスを解析する場合は、[Timing constraint] ドロップダウンメニューから制約を選択します。

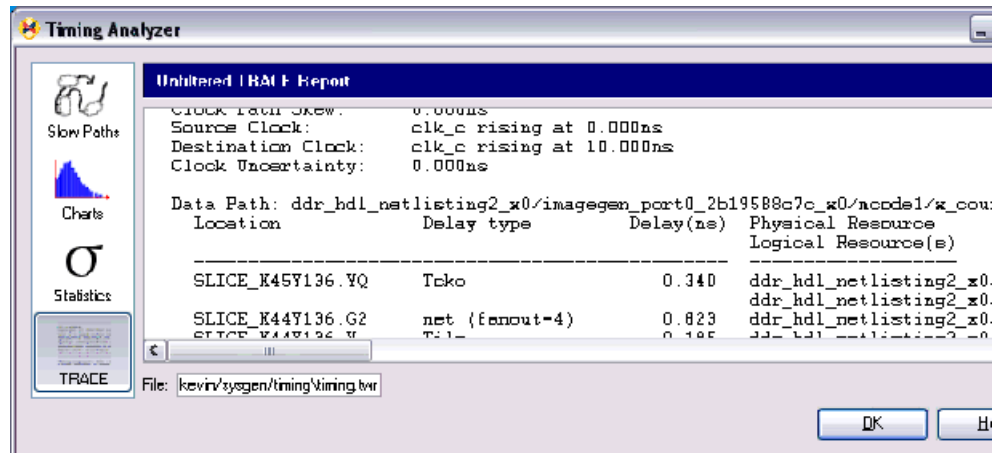
赤で示された部分に注目してください。これらのパスはスラックが負であり、タイミング制約を満たしていません。この例では、いくつかのパスで制約が満たされていませんが、それほど大きく離れているわけではなく、デザインをある程度変更すればタイミングが満たされると考えることができます。

## 統計

[Statistics] をクリックすると、制約の数、解析されたパスの数、デザインの最大周波数など、デザインに関する統計情報が表示されます。

## TRACE レポート

[TRACE] をクリックすると、TRACE レポートが表示されます。このファイルには、解析されたパスの詳細が含まれています。解析された各パスに対して、ネットおよびロジック遅延、クロックスキュー、クロック誤差に関する情報が示されます。このウィンドウの左下には、TRACE レポートのパス名が示されます。



## タイミングを満たしていないパスの向上

タイミングを満たしていないパスがある場合にどうすれば向上できるかは簡単に答えることができない質問ではなく、FPGA デザインの下位レベルを検証する必要がある場合があります。

通常タイミングを満たすための手順は、次のとおりです。

1. ソース デザインの変更: どんなタイミング問題もソース デザインを変更することで解決でき、これが回路のスピードを向上させる最も簡単な方法です。ただし、多くの場合、高速のデバイスに変更するなど、手取り早い解決策が使用されているのが現実です。ソース デザインは、いくつかの方法で変更できます。
  - a. パイプライン: これはスピードを確実に向上させる方法ですが、注意が必要です。パイプラインレジスタを追加すると、レイテンシが増加します。フィードバックを使用するデザインでは、デザインの一部でパイプラインのバランスを取る必要がある場合があります。パイプラインの詳細は、後に示す例を参照してください。
  - b. 並列化: これは、スピードを向上させるためにおそらく 2 番目に重要な変更です。要求されるスピードで動作していない FIR フィルタがある場合、半分のレートで動作する 2 つの FIR フィルタを並列に使用し、出力をインターリーブします。ただし、この方法を使用すると、エリアが増加します。
  - c. リタイミング: 組み合わせロジック内でレジスタの位置を移動することにより、パスの遅延を均等に分配します。この方法は、スラックが足りない部分と余っている部分がある場合に効果的です。合成ツールによっては、ある程度のリタイミングを自動的に実行できます。
  - d. 複製: レジスタまたはバッファを複製すると、ロジックの使用量は増加しますが、複製したオブジェクトのファンアウトを減少させることができます。これにより、ネットのキャパシタンスが減少し、ネット遅延も減少します。複製したレジスタは、フロアプランしてこれらのレジスタが駆動するロジック グループの近くに配置できます。System Generator のような高レベルのデザイン環境では、複製は通常ツールにより自動的に実行され、手動で実行することは一般的ではありません。



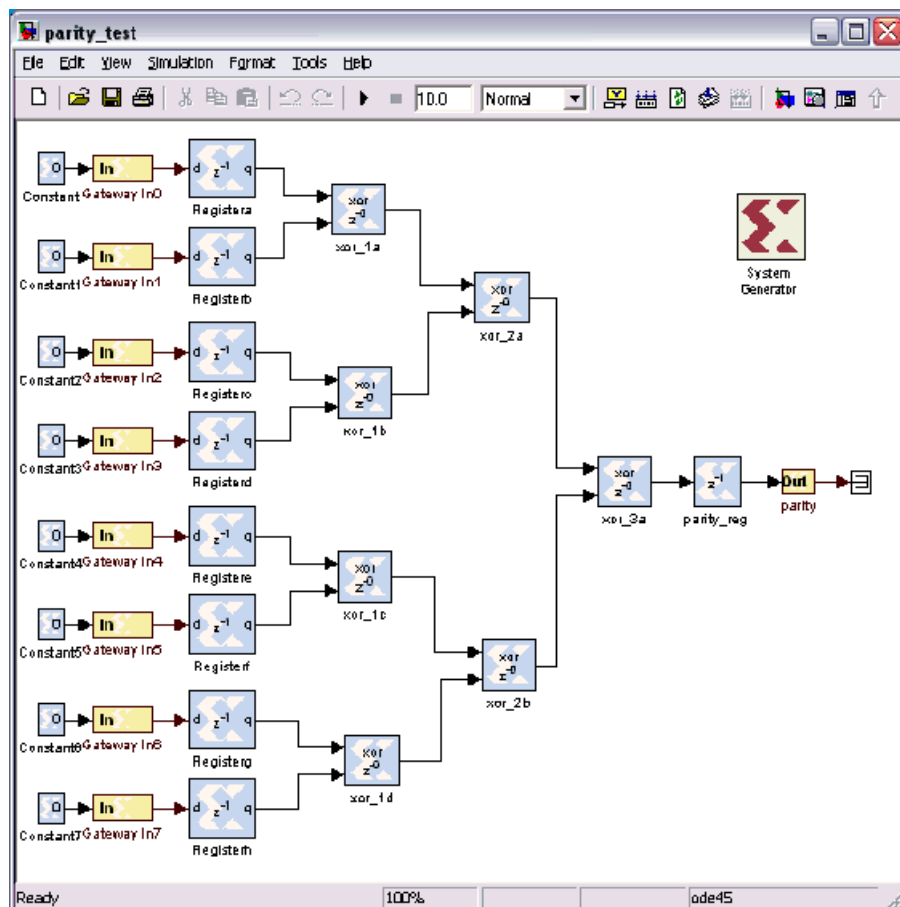
- e. シヤノン展開 : この方法では、低速のロジックへの依存を回避するため、クリティカルパスの高速ロジックを複製します。この操作は、合成ツールで自動的に実行される場合があります。
  - f. ハード コアの使用 : ROM のインプリメントに分散 RAM の代わりにブロック メモリハード コアを使用したり、多入力加算器を 500MHz で動作可能な DSP48 ブロックにインプリメントするなど、エンベデッド ハード コアを利用するとデザインを高速にできます。
  - g. 新しい考え方 : 大きな遅延を作成する必要がある場合、長いキャリー チェーンを持つカウンタを使用する代わりに、SRL16 を使用してカスケード接続したジョンソン リングを構成したり、LFSR を使用して遅延を作成できます。どちらの方法でもキャリー チェーンは不要で、スピードが大幅に向上します。デザイン エレメントによっては、その設計を 1 から考え直すことが必要な場合があります。
2. 過度の制約を削除 : デザインのエレメントを低いサンプリング レートで動作させればよい場合には、System Generator で Down Sample および Up Sample ブロックを使用してください。これらのブロックを使用しないと、System Generator でデザインのその部分のサンプリング レートが低くてよいことが認識されず、デザインに過度の制約が適用される結果となります。
  3. 制約の変更 : デザインを低いクロック速度で実行可能な場合は、要件を満たすのにこれが最も簡単な方法です。ただし、通常デザイン要件は厳しいので、この方法はほとんどの場合適用できません。
  4. PAR のエフォート レベルの増加 : ISE の MAP および PAR ツールでは、エフォート レベルを引数として指定できます。ISE を使用する際、MAP の -timing オプションを使用してみてください。また、PAR のエフォート レベルを増加すると、PAR の実行時間は長くなりますが、より高速なデザインが得られる場合があります。
  5. SmartXplorer を使用した PAR の複数回実行 : PAR の実行では、初期条件が最終的な結果に大きく影響します。Project Navigator から SmartXplorer を実行すると、デザイン パフォーマンスを最適化する異なるインプリメンテーション プロパティの組み合わせを使用してインプリメンテーション フローが複数回実行されます。
  6. フロアプラン : この方法はできるだけ使用しないことをお勧めしますが、デザインが大きく向上する可能性があります。PAR による配置を変更し、クリティカル エレメントをデバイス上で近い位置に配置して、ネット遅延を削減します。CPLD デバイスでは、これに ISE の PACE ツールを使用します。FPGA デバイスでは、より高度なツール PlanAhead™ ソフトウェアを使用します。
  7. 高速のデバイスを使用 : この方法が最初に適用されることがよくありますが、コストのかかる解決策でもあります。以前のデバイスの代わりに新しい高速デバイスを使用すると、新しいデバイスは以前のデバイスに比べて安価である場合があるので、コストを削減できる可能性があります。同じファミリの高速デバイスに移行すると、通常コストが増加し、1 ~ 6 の方法を適用すればほとんどの場合必要ありません。

## チュートリアル：タイミング解析の使用

System Generator で生成されたハードウェアがタイミング要件を満たさない場合があります。これは、通常デザインにセットアップ タイム違反があることを意味します。セットアップ タイム違反は、ある信号が 1 つの同期エレメントの出力から別の同期エレメントの入力に要求されたクロック周期内に到達することができず、2 番目の同期エレメントのセットアップ タイム要件が満たされない状況です。

このセクションでは、タイミング解析を使用して回路のパフォーマンスを向上させる例を示します。この例で使用するデザインは、8 入力 XOR を使用してバイトのパリティを算出するパリティ カリキュレータです。このデザインは、次の場所にあります。

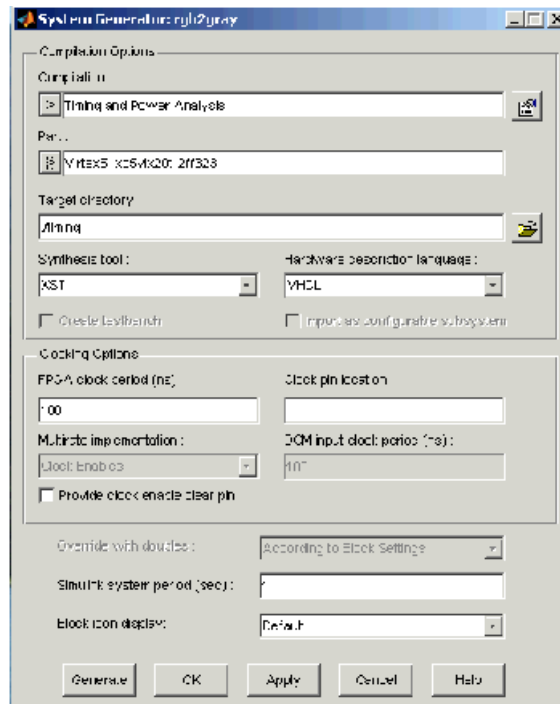
`<path_to_sysgen>/examples/timing_analysis/parity_test.mdl`



このデザインには、8 つの 1 ビット Gateway In ブロックがあり、その入力がそれぞれ 1 ビットレジスタに取り込まれ、7 つの 2 入力 XOR ブロックで処理されます。これらの XOR ブロックのレイテンシは 0 で、純粋な組み合わせです。最後のレジスタ parity\_reg には最終的な結果 (パリティ) が取り込まれ、Gateway Out ブロックに接続されています。各パスは 3 つの XOR ブロックを介して parity\_reg に到達するので、このデザインのロジック レベル数は 3 であると考えられます。

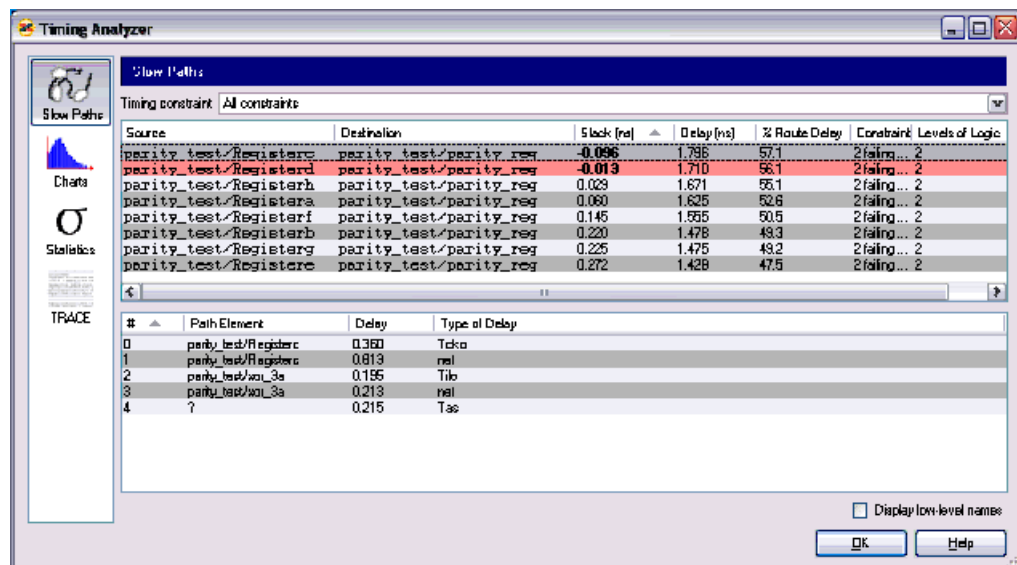
## デザイン例の生成

このデザインを、[Compilation] を [Timing Analysis] に、周期を 1.4ns (714MHz) に設定して生成します。周波数を非常に高く設定するのは、タイミングを向上させる手順を示すため、タイミングを満たさないパスを作ることが目的です。これらのパラメータは、System Generator トークンで設定します。



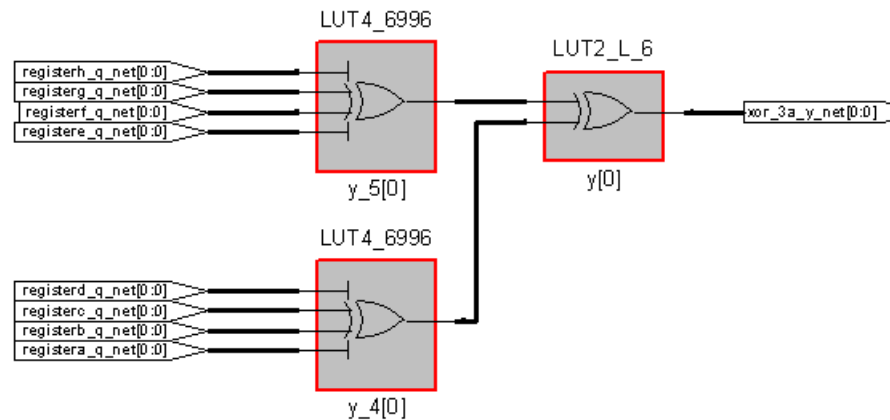
## 遅いパスの検証

[Generate] をクリックすると、次のような [Timing Analyzer] ウィンドウが表示されます。



タイミングを満たしていないパスは2つあり、赤/ピンクでハイライトされています。上図では、一番上のパスは選択しているのでグレーになっています。負のスラック値は、太字で示されています。最も遅いパスは、96ps の差でタイミング要件を満たしていません。

System Generator モデルではすべてのパスのロジック レベル数 3 であるように見えますが、タイミング解析の結果ではロジック レベル数が 2 になっています。これは、インプリメントされたデザインが System Generator モデルとは正確に一致していないからです。この例では、合成ツールで一部の 2 ビット XOR ブロックが 4 入力 LUT に配置され、次の Synplify Pro の回路図に示すように、8 ビット XOR が 2 段のロジック レベルで作成されているからです。



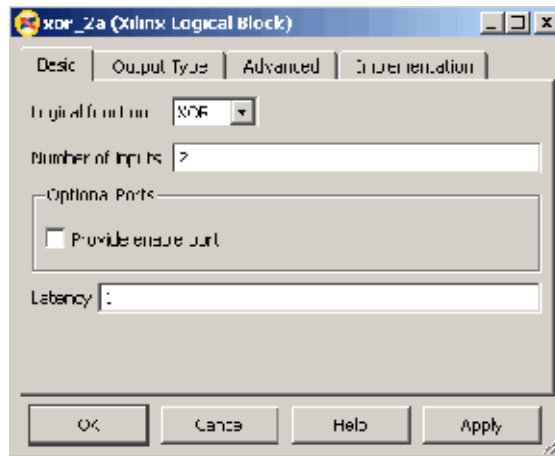
ネット およびブロックの下位レベル名を表示してみてください ([Display low-level names] をオン)。タイミング解析ツールの名前の関連付け機能が非常に有益であることがわかります。

選択されたパスの詳細を確認してください。ロジック遅延は削減できません。ネット遅延の 1 つが 813ps です。これは、フロアプラン、配置配線の複数回実行、または PAR のエフォート レベルを上げることで削減できる可能性があります。

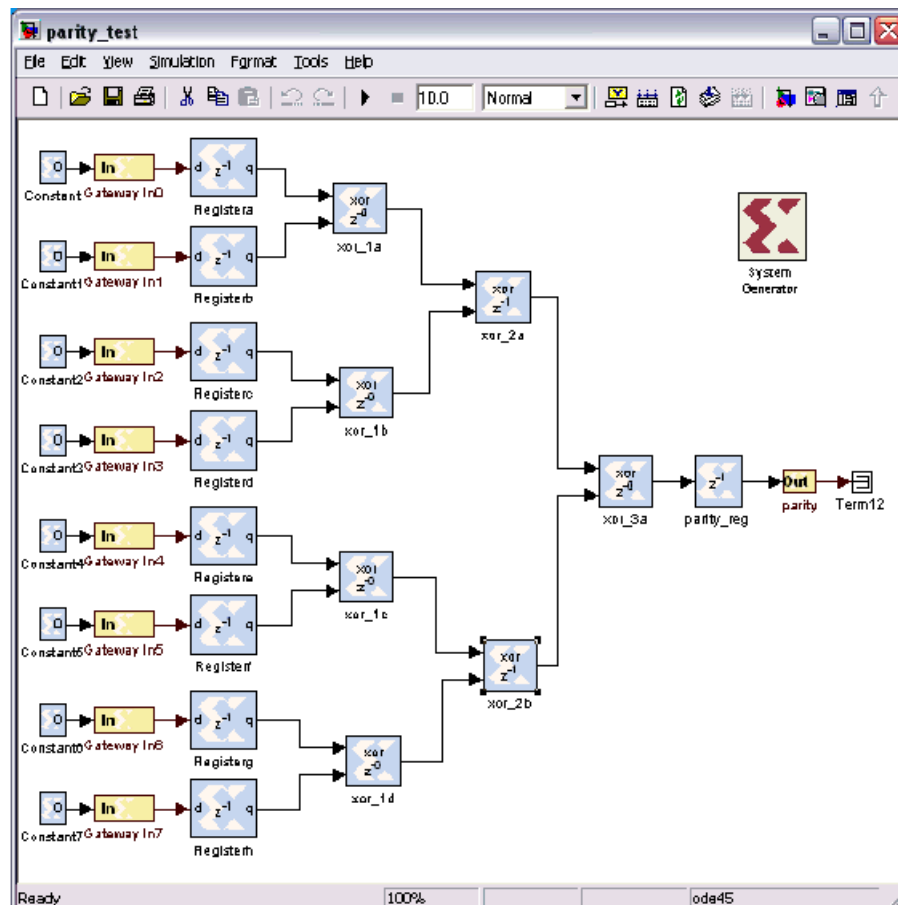
### デザインの向上

ここでは、ソース デザインを変更することで、より 確実な修正を試みます。このデザインにはフィードバック パスがなく、デザインに 1 サイクルのレイテンシを追加してパイプライン化することができますと想定します。タイミングを満たしていないパスのロジック レベル数は 2 です。論理的には、どんなデザインも 1 段のロジック レベルでインプリメント できます。ここで、それを実行します。

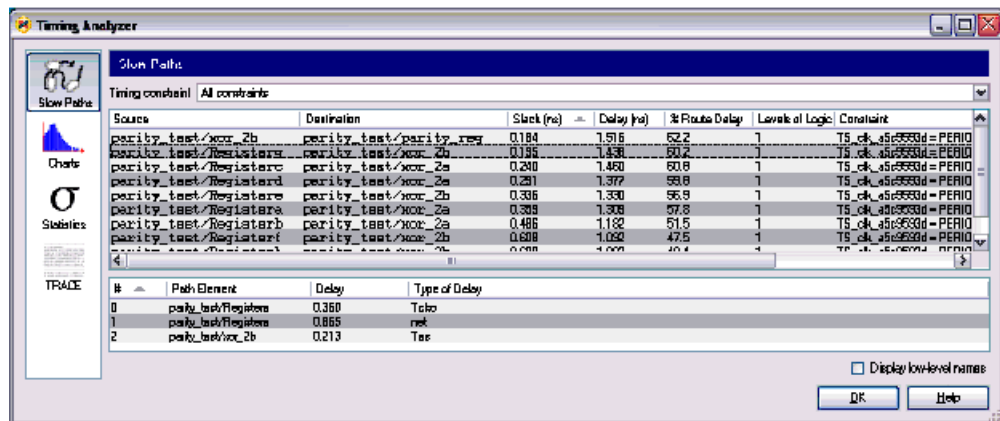
パイプライン段を追加するには、一部の XOR ブロックにレイテンシを追加します。XOR ブロックをダブルクリックし、パラメータ ダイアログ ボックスで [Latency] を 1 に設定します。



レイテンシを追加することにより、XOR ゲートの後にレジスタが追加されます。ここでは、xor\_2a と xor\_2b のレイテンシを変更します。Synplify Pro の回路図から、合成済みのデザインではこれらのブロックの出力が最初のロジック レベルの出力であることがわかっています。変更後の System Generator モデルでは、2 つの XOR ブロックに Z<sup>-1</sup> と表示されており、新しいレイテンシが示されています。

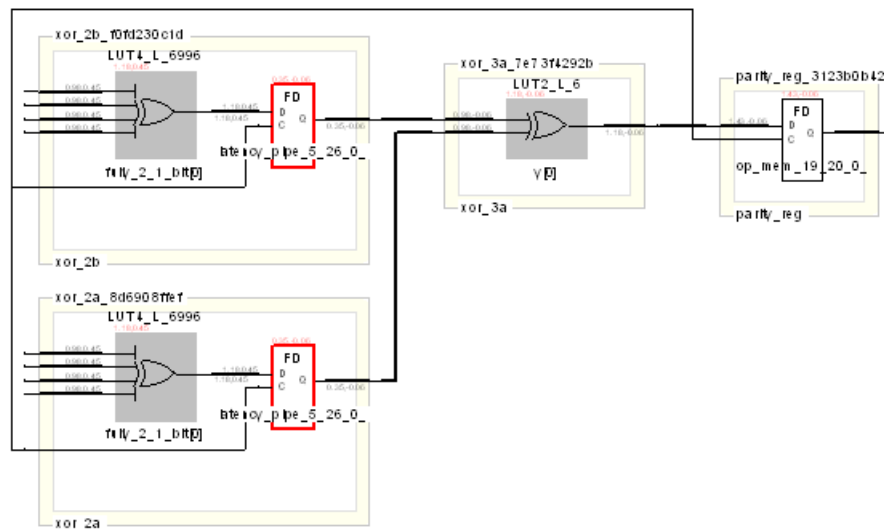


このデザインを先ほどと同様に生成し、遅いパスを検証します。



すべてのパスでタイミングが満たされました。デザインは、コストのかかるデバイスに移行することなく、短時間で修正できました。

すべてのパスのロジック レベル数が 1 になっています。どういう構造になっているかを、Synplify Pro の回路図で確認してみます。



2つのロジックレベルの間に、レジスタ(赤色)が追加されています。回路は修正前と同様に機能しますが、1サイクルのレイテンシが追加されています。

### リタイミングを使用したデザインの向上

レイテンシを元のデザインと同じにする必要がある場合は、最後の出力レジスタまたは入力レジスタを削除することでこれを達成できる可能性があります。これは、ザイリンクス チップ外のパス(PCBの銅線パス)の制約を増加させることとなりますが、ボードレベルにパス遅延によっては実現可能です。これはリタイミングの例であり、レイテンシは同じままレジスタを移動します。

## コンパイル ターゲットの作成

System Generator でデザインをハードウェアにコンパイルする際に生成される HDL ファイルとネットリスト ファイルから、使用する FPGA に適したコンフィギュレーション ビットストリーム ファイルを生成するには、追加のツールを実行する必要があります。通常は、FPGA コンフィギュレーション ファイルを生成するのに Project Navigator を使用しますが、ほかの方法もあります。たとえば、デザインをコンパイルする際にコンフィギュレーション ファイルを生成するのに必要なツールが自動的に実行されるように、System Generator を設定できます。このようにすると、System Generator からビットストリームの生成までを実行できるので便利です。また、コンフィギュレーション ファイルが作成された後、別のツール (ChipScope™ Pro Analyzer、iMPACT など) を実行するよう設定することもできます。

System Generator でモデルをハードウェアにコンパイルする方法は、選択したコンパイル ターゲットによって異なります。最も頻繁に使用されるのは [HDL Netlist] で、デザインの HDL ネットリストとコアが生成されます。この設定を拡張して、生成された HDL ネットリスト ファイルに追加のツールを実行する新しいコンパイル ターゲットを作成できます。

このセクションでは、[HDL Netlist] ターゲットを拡張して、FPGA ハードウェアの生成およびコンフィギュレーションを実行する新規コンパイル ターゲットを作成する方法を示します。具体的には、モデルのビットストリームを生成し、そのビットストリームに対してさまざまなツールを実行するよう System Generator を設定します。

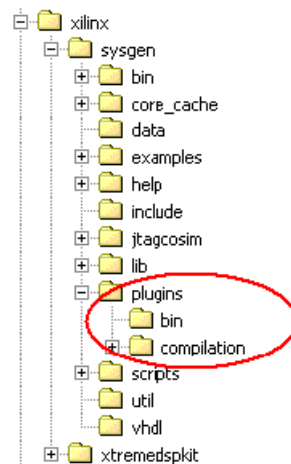
### 新規コンパイル ターゲットの定義

[HDL Netlist] 設定で生成される出力ファイルに対してツールを実行する新規コンパイル ターゲットを作成します。コンパイル ターゲットは、2 つ以上の MATLAB 関数で定義されます。最初の関数 `xltarget.m` は、ターゲットを System Generator トークンのパラメータ ダイアログ ボックスで選択できるようにし、ターゲットに関する詳細情報が含まれる MATLAB 関数を指定します。この関数は、`target info` 関数と呼ばれます。ターゲットに関する情報を定義するこの `target info` 関数には、任意の名前を付けることができますが、`xltarget.m` 関数で正しく指定する必要があります。`target info` 関数で、`post-generation` 関数が定義されている場合があります。この関数は、通常の HDL ネットリスト コンパイルが終了した後にツールまたはスクリプトを実行します。次に、これらの関数の詳細を説明します。

#### xltarget 関数

`xltarget` 関数は、System Generator でサポートする必要のある 1 つ以上のコンパイル ターゲットを指定します。ターゲットに関する詳細情報の場所も指定します。

メモ : System Generator では、System Generator のインストール ディレクトリの `plugins\compilation` およびそのサブディレクトリから `xltarget.m` ファイルを検索して、サポートされているコンパイル ターゲットを判断します。



`xltarget` 関数では複数のターゲットを指定できますが、各コンパイル ターゲットにそれぞれ `xltarget` 関数を指定するのが一般的です。これらの関数が保存されるディレクトリにより、ターゲットが区別されます。これは、各 `xltarget.m` ファイルを `plugins\compilation` ディレクトリの下に配置した独自のサブディレクトリに保存する必要があります。

`xltarget` 関数は、ターゲット情報のセル配列を返します。このセル配列の各要素は、異なるコンパイル ターゲットを定義します。これらの要素は、次の 2 つのパラメータを定義する MATLAB struct です。

1. System Generator トークンのパラメータ ダイアログ ボックスの [Compilation] に表示されるコンパイル ターゲットの名前
2. ターゲットに関する詳細情報 (System Generator トークンのパラメータ、使用する post-generation 関数など) を見つけるために起動する MATLAB 関数の名前

次のコードは、Standalone Bitstream、MPACT、ChipScope Pro Analyzer という 3 つのコンパイル ターゲットを定義する例を示します。

```
function s = xltarget
s = {};
target_1('name') = 'Standalone Bitstream';
target_1('target_info') = 'xltools_target';
target_2('name') = 'iMPACT';
target_2('target_info') = 'xltools_target';
target_3('name') = 'ChipScope Pro Analyzer';
target_3('target_info') = 'xltools_target';
s = {target_1, target_2, target_3};
```

上記のコードの `name` フィールドは、System Generator トークンのパラメータ ダイアログ ボックスの [Compilation] に表示されるコンパイル ターゲットの名前を指定します。

```
target_1('name') = 'Standalone Bitstream';
```

`target_info` フィールドは、ターゲットに関する情報を見つけるために呼び出す `target info` 関数を指定します。この関数には、任意の名前を付けることができますが、対応する `xltarget.m` ファイルと同じディレクトリか、MATLAB パスに保存する必要があります。

```
target_1('target_info') = 'xltools_target';
```



**メモ** : `xltarget` 関数のサンプル ファイルが、`<path_to_sysgen>\examples\comp_targets` ディレクトリにあります。この関数を変更して、ビットストリームに関するコンパイル ターゲットを定義できます。

## target info 関数

上記のコードで `target_info` フィールドで指定される `target info` 関数には、次の 2 つの役割があります。

- System Generator トークンのデフォルト設定と選択可能な設定を定義します。
- 標準のコード生成プロセスの前後で呼び出す必要のある関数を指定します。

**メモ** : `target info` 関数、`xltools_target.m` ファイルのサンプル ファイルが、`<path_to_sysgen>\examples\comp_targets` ディレクトリにあります。

これらの関数の 1 つに、`post-generation` 関数があります。`post-generation` 関数は、標準のコード生成の後に実行されます。次のコードは、`target info` 関数で `post-generation` 関数を指定する方法を示します。

```
settings('postgeneration_fcn') = 'xltools_postgeneration';
```

## post-generation 関数

System Generator でのコンパイルを拡張するには、`post-generation` 関数を指定するコンパイルを定義します。`post-generation` 関数は、HDL ファイルおよびネットリスト ファイルを生成した後にこれらのファイル进行处理する方法を指定する MATLAB 関数です。この関数は、[Compilation] を [HDL Netlist] に設定したときに System Generator で実行される通常のコード生成プロセス (デザインの HDL 記述の生成、CORE Generator™ の実行など) が終了した後に実行されます。たとえば、ハードウェア協調シミュレーション ターゲットでは、Simulink シミュレーションで使用可能なハードウェアを生成するために必要なツールを実行する `post-generation` 関数を定義します。

**メモ** : `xlBitstreamPostGeneration.m` および `xltools_postgeneration.m` という 2 つの `post-generation` 関数が、`<path_to_sysgen>examples/comp_targets` ディレクトリにあります。

### xlBitstreamPostGeneration.m

この `post-generation` 関数の例は、モデルを System Generator トークンのパラメータ ダイアログボックスで指定した設定 (FPGA デバイス、クロック周波数、クロック ピン ロケーションなど) に適したコンフィギュレーション ビットストリームにコンパイルします。

この関数では、XFLOW を使用して FPGA コンフィギュレーション ビットストリームを生成するのに必要なザイリンクス ツールを実行します。

XFLOW で実行する各ツールの設定を指定できます。この方法の詳細は、「[XFLOW の使用](#)」を参照してください。

### xltools\_postgeneration.m

コンフィギュレーション ビットストリームを生成した後、FPGA をコンフィギュレーションし、実行するツール (iMPACT、ChipScope Pro Analyzer など) を実行する場合があります。

`xltools_postgeneration` 関数は、まず `xlBitstreamGeneration` 関数を呼び出してビットストリームを生成し、選択したコンパイル ターゲットに応じて適切なツールを実行します。

たとえば、ビットストリームの生成後に **iMPACT** を実行するコンパイル ターゲットを作成する場合、次のコードを使用します (**iMPACT** がシステム パスにある場合)。

```
if (strcmp(params.compilation, 'iMPACT'))
 dos('impact');
end;
```

最初の行はコンパイル ターゲットの名前をチェックし、2 番目の行は **iMPACT** を起動する DOS コマンドを指定します。**ChipScope Pro Analyzer** も、同様に起動できます。

```
if (strcmp(params.compilation, 'ChipScope Pro Analyzer'))
 xlCallChipScopeAnalyzer;
end;
```

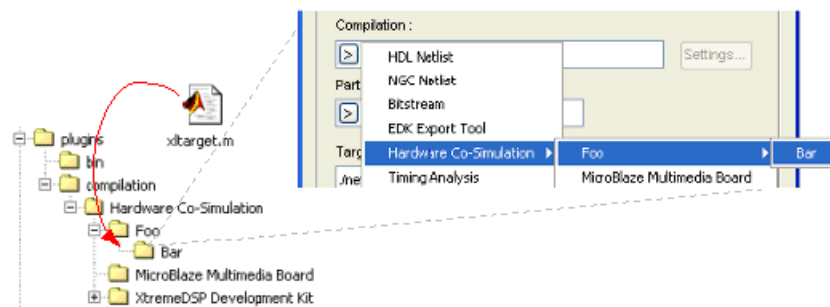
**メモ** : `xlCallChipScopeAnalyzer` は、**ChipScope** を起動するために **System Generator** で提供されている **MATLAB** 関数です。

## コンパイル ターゲットの設定とインストール

次に、ビットストリームを生成する新規コンパイル ターゲットを設定し、インストールする手順を示します。

1. `examples\comp_targets` ディレクトリから `xltarget.m`、`xltools_postgeneration.m`、`xltools_target.m` ファイルを一時ディレクトリにコピーします。
2. 上記のファイルの権限を変更して、編集できるようにします。
3. `xltarget.m` ファイルにコンパイル ターゲット (**iMPACT**、**ChipScope Analyzer Pro** など) を追加します。
4. `xltools_postgeneration.m` ファイルに、ツールの起動コードを追加します。
5. `plugins\compilation` ディレクトリに新規ディレクトリ (**Bitstream** など) を作成します。このディレクトリに、`xltarget.m`、`xltools_postgeneration.m`、`xltools_target.m` ファイルをコピーします。

**メモ** : **System Generator** トークンのパラメータ ダイアログ ボックスにある **[Compilation]** のサブメニューは、`plugins\compilation` ディレクトリのディレクトリ構造を反映しています。コンパイル ターゲットの新しいディレクトリを作成すると、その名前が **[Compilation]** のサブメニューに表示されます。



6. `examples\comp_targets` ディレクトリから `xlBitstreamPostGeneration.m`、`xlToolsMakebit.pl`、`balanced_xltools.opt`、および `bitgen_xltools.opt` ファイルを **MATLAB** パスのディレクトリにコピーします。これらのファイルは、共通のディレクトリに配置する必要があります。

7. MATLAB の [Command Window] に、次のように入力します。

```
>> rehash toolboxcache
>> xlrehash_xltarget_cache
```

8. これで、System Generator トークンのパラメータ ダイアログ ボックスの [Compilation] で新しいコンパイル ターゲットを選択できます。

## XFLOW の使用

この例に含まれている **post-generation** 関数では、FPGA のコンフィギュレーション ファイルを生成するのに **XFLOW** を使用します。**XFLOW** は、デザインの合成、インプリメンテーション、シミュレーションをコマンド ラインを使用して自動的に実行します。実行するツールおよびその設定は、コマンド ファイルで指定されます。

この例には、**balanced\_xltools.opt** および **bitgen\_xltools.opt** という 2 つの **XFLOW** オプションファイルが含まれています。これらのファイルは、それぞれ **XFLOW** のインプリメンテーションフローとコンフィギュレーションフローに関連付けられています。**balanced\_xltools.opt** は、**NGDBuild**、**MAP**、および **PAR** のオプションを指定し、**bitgen\_xltools.opt** は **BitGen** のオプションを指定します。これらのファイルは、必要に応じて変更できます。

# 索引

## A

Addressable Shift Register ブロック 15  
ADR ブロック 15

## C

ChipScope Pro Analyzer 125

## D

DCM リセット ピン 40  
DCM ロック ピン 40  
DSP48  
    DSP48ブロックを使用したマップ 98  
    設計手法 96, 103  
    標準コンポーネントにマップ 97  
    物理設計 104  
    ロジック合成ツールを使用したマップ 97  
DSP48 Macro ブロック 99

## E

EDK  
    System Generator でのサポート 142  
    ソフトウェアドライバの記述 140  
    ソフトウェアドライバの生成 139  
EDK Export Tool 343  
    pcore のエクスポート 145  
EDK Import Wizard 143, 158  
EDK プロセッサ  
    インポート 142  
    プロセッサ ポートの追加 144  
[Expose Clock Ports] オプション  
    チュートリアル 31

## F

FDATool  
    デジタル フィルタ アプリケーションでの使用 106  
FPGA  
    概要 12

高パフォーマンス FPGA に関するメモ 86  
ビットストリームの生成 90  
FPGA ビットストリームの生成 90  
FSL ベース pcore 136

## H

HDL 協調シミュレーション  
    HDL シミュレータの設定 296  
    概要 296  
    複数のブラック ボックスの協調シミュレーション 298  
HDL テストベンチ 48  
HDL ネットリストへのコンパイル 338  
[Hybrid DCM-CE] オプション  
    チュートリアル 26  
    リセット ピン 25  
    ロック ピン 25

## J

JTAG ハードウェア協調シミュレーション  
    最上位コンポーネントの作成 278  
    新規プラットフォームのサポート 267, 268  
    新規ボード パッケージの検出 280  
    プラットフォームの情報の取得 274  
    ボード サポート パッケージのインストール 279  
    ボード サポート パッケージ ファイル 273  
    ボード専用ポートの手動指定 277  
JTAG ベースのハードウェア協調シミュレーション 262, 263, 266

## M

M 関数  
    ブラック ボックスのコンフィギュレーション 284  
MATLAB  
    disp 関数 69  
    FIR の例 64  
    FPGA へのコンパイル 49

MCode ブロックへパラメータを渡す 55  
RPN カリキュレータ 67  
オプションの入力ポート 58  
単純なシフト操作 54  
単純な数値演算 51  
単純なセクタ 50  
パラメータ指定アキュムレータ 61  
有限ステート マシン 60  
レイテンシのある複素乗算器 53

## MicroBlaze

System Generator でのチュートリアル 152  
システム設計とシミュレーション 156

## ML402 ボード

JTAG ハードウェア協調シミュレーション用のインストール 262

## ML605 プラットフォーム

JTAG ハードウェア協調シミュレーション用のインストール 263

## N

NGC ネットリストへのコンパイル 338

## P

### pcore

System Generator モデルをペリフェラルとしてエクスポート 138  
エクスポート 145  
開発中としてエクスポート 343

### pcore のエクスポート

カスタム バス インターフェイスのイネーブル 344

## PicoBlaze

System Generator でのチュートリアル 147  
System Generator 内での設計 145  
概要 145

PLB ベース pcore 136  
Project Navigator

System Generator との統合フロー 72

## S

SBDBuilder  
 プラグイン ファイルの保存 272  
 ボード専用 I/O ポートの指定 270

SDK スタンドアロン  
 XPS からのソフトウェア プロジェクトの移行 173

[Simulink system period] オプション 40

SP605 プラットフォーム  
 JTAG ハードウェア協調シミュレーション用のインストール 266

Spartan-3A DSP 1800A スタータ プラットフォーム  
 イーサネット ハードウェア協調シミュレーション用にインストール 249

System Generator  
 コンフィギャブル サブシステム 80  
 コンフィギャブル サブシステムからのハードウェアの生成 84  
 コンフィギャブル サブシステムからのブロックの削除 83  
 コンフィギャブル サブシステムの使用 82  
 コンフィギャブル サブシステムの定義 80  
 コンフィギャブル サブシステムへのブロックの追加 83  
 システム レベルのモデリング 18  
 自動生成されたクロック イネーブル ロジックのリセット 93  
 出力ファイル 42  
 物理デザイン ツールでのデザインの処理 87  
 ブロックセット 19

System Generator 制約  
 IOB タイミングと配置 44  
 システム クロック周期 44  
 制約ファイル 44  
 複数サイクル パス 44  
 例 45

System Generator デザインのインポート  
 Project Navigator との統合フロー 72  
 デザインの統合に関する規則 71  
 手順の例 73

System Generator デザイン フロー  
 アルゴリズムの解析 17  
 大型デザインの一部としてインプリメント 17  
 完全なデザインのインプリメント 17

System Generator トークン  
 コンパイルとシミュレーション 37

## T

TDM データ ストリーム 15

TRACE レポート  
 タイミング解析 355

## X

XFLOW の使用 366

xlCallChipScopeAnalyzer 365

xlmax 50

xlSimpleArith 51

xltarget  
 新規コンパイルターゲットの定義 362

xlTimingAnalysis 348

xltools\_postgeneration 364

xltools\_target 364

XPower  
 消費電力解析 347

## あ

アルゴリズムの解析 17

暗号化された VHDL ファイル  
 ブラック ボックスとしてインポート 332

## い

イーサネット ベースのハードウェア協調シミュレーション 249

色分け表示  
 ブロックを信号レートに応じて 22

インストール  
 JTAG ハードウェア協調シミュレーション用の ML402 ボードのインストール 262  
 JTAG ハードウェア協調シミュレーション用の ML605 プラットフォームのインストール 263

JTAG ハードウェア協調シミュレーション用の SP605 プラットフォームのインストール 266

ハードウェア協調シミュレーション用の Spartan-3A DSP 1800A スタータプラットフォーム 249

インプリメント  
 完全なデザイン 17  
 デザインの一部 17

インポート  
 EDK プロジェクト 138  
 EDK プロセッサ 142  
 System Generator デザイン 71

## う

ウィザード  
 Base System Builder 164  
 EDK Import 158  
 EDK プロジェクトのインポート 143  
 ブラック ボックスのコンフィギュレーション 283, 314

## え

エクスポート  
 pc core 145  
 System Generator モデルを pc core として 138

## お

オーバーサンプリング 24

## か

階層制御 41

開発中  
 pc core を開発中としてエクスポート 343

概要  
 FPGA 12

カスタム バス インターフェイス  
 pc core エクスポート用 344

可変クロック周波数  
 ハードウェア協調シミュレーション用に選択 186

完全精度信号型 21

## き

- 共有メモリのコンパイル
  - ハードウェア協調シミュレーション 196
- 共有メモリのサポート
  - ハードウェア協調シミュレーション 195

## く

- クロック
  - タイミング 22
  - 同期 24
  - 非同期 24
- クロック イネーブル
  - ファンアウトの低減 87
- クロック供給オプション
  - [Clock Enables] 25
  - [Expose Clock Ports] 26
  - [Hybrid DCM-CE] 25, 40
- クロック周波数
  - ハードウェア協調シミュレーション用に選択 186
- クロック ドメインの切り替え 115
- クロック ドメインの分割 115

## こ

- コード生成
  - 自動 36
- コンパイル
  - HDL ネットリスト生成 338
  - EDK Export Tool 343
  - NGC ネットリスト生成 338
  - タイミングおよび消費電力解析 347
  - ハードウェア協調シミュレーション 346
  - ビットストリームの生成 339
- コンパイル、MATLAB
  - disp 関数 69
  - FIR の例 64
  - FPGA へのコンパイル 49
  - MCode ブロックへパラメータを渡す 55
  - RPN カリキュレータ 67
  - オプションの入力ポート 58
  - シフト操作 54
  - 単純な数値演算 51
  - 単純なセレクタ 50

- パラメータ指定アキュムレータ 61
- 有限ステート マシン 60
- レイテンシのある複素乗算器 53
- コンパイル ターゲットの作成 362
- コンパイル ターゲットの設定とインストール 365
- コンパイル タイプ
  - EDK Export Tool 343
  - HDL ネットリストへのコンパイル 338
  - NGC ネットリストへのコンパイル 338
  - XFLOW の使用 366
  - 新規コンパイルターゲットの作成 362
  - 設定とインストール 365
  - ハードウェア協調シミュレーション用のコンパイル 346
  - ビットストリームへのコンパイル 339
- コンフィギュラブル サブシステムと System Generator 80

## さ

- サイクル精度 19
- サイクル単位クロック アイランド 113
- サイクル単位のモデリング 22
- ザイリンクス
  - ブロックセット 20
  - リファレンス ブロックセット 20
- ザイリンクス ツール フローの設定
  - ハードウェア協調シミュレーション 205

## し

- システム レベルのモデリング 18
- 自動コード生成 36
- 自動生成されたクロック イネーブル ロジック
  - リセット 93
- 時分割多重 15
- 出力ファイル
  - System Generator で生成 42
- 消費電力解析
  - XPower を使用 347
- 新規コンパイル ターゲットの定義 362
  - target info 関数
    - xltools\_target 364
  - xltarget 関数 362

- 信号型 21
  - Gateway ブロック 21
  - 完全精度 21
  - データ型の表示 22
  - ユーザー指定の精度 21

## せ

- 制御
  - 階層 41
- 生成
  - EDK ソフトウェア ドライバ 139
  - FPGA ビットストリーム 90
- 制約ファイル
  - System Generator 44

## そ

- ソフトウェア プロジェクト
  - XPS から SDK への移行 173

## た

- タイミングおよび消費電力解析
  - コンパイル タイプ 347
- タイミング解析
  - TRACE レポート 355
  - 以前に生成されたデータの表示 348
  - 遅いパスの表示 350
  - 概要 349
  - 下位レベル名の表示 351
  - クロスプローブ 352
  - クロック スキューとジッタ 350
  - 周期とスラック 349
  - タイミングを満たしていないパスの向上 355
  - チュートリアル 357
  - 統計 354
  - パス解析の例 349
  - ヒストグラム 353, 355
- タイミングとクロック 22
- タップ遅延ライン 15

## ち

- チュートリアル
  - ChipScope
    - System Generator 内での使用 126

クロック供給  
 [Expose Clock Ports] オプションの使用 31  
 デジタル クロック マネージャ (DCM) を使用 26  
 タイミング解析 357  
 ハードウェア/ソフトウェア協調設計  
 MicroBlaze プロセッサ システムの設計とシミュレーション 156  
 System Generator で MicroBlaze ペリフェラルを作成 152  
 System Generator で PicoBlaze を使用 147  
 新規 XPS プロジェクトの作成 164  
 ブラック ボックス  
 CORE Generator モジュールのインポート 300  
 Verilog モジュールのインポート 321  
 VHDL モジュールのインポート 313  
 暗号化された VHDL モジュールのインポート、シミュレーション、エクスポート 332  
 ダイナミック ブラック ボックス 323  
 複数のブラック ボックスの同時シミュレーション 324  
 ブラック ボックスの HDL 要件を満たす VHDL ラップが必要な CORE Generator モジュールのインポート 306

## て

低減  
 クロック イネーブルのファンアウト 87  
 テストベンチ  
 HDL 48  
 デバッグ  
 ChipScope Pro を使用 125

## と

同期化のメカニズム  
 不定データ 34  
 有効なポート 34  
 同期クロック 24

[Clock Enables] オプション 25  
 [Expose Clock Ports] オプション 26  
 [Hybrid DCM-CE] オプション 25, 40

## ね

ネットリスト生成  
 複数クロック デザイン 117  
 ネットワーク ベースのイーサネット  
 ハードウェア協調シミュレーション 193

## は

ハードウェア  
 オーバーサンプリング 24  
 ハードウェア協調シミュレーション 181  
 JTAG ハードウェア要件 267  
 共有 FIFO の協調シミュレーション 203  
 共有メモリのコンパイル 196  
 共有メモリのサポート 195  
 共有メモリの制限 205  
 共有レジスタの協調シミュレーション 201  
 クロック周波数の選択 186  
 コードの生成 183  
 コンパイル ターゲットの選択 182  
 ザイリンクス ツール フローの設定 205  
 ネットワーク ベースのイーサネット 193  
 非保護の共有メモリの協調シミュレーション 199  
 フレーム ベースのシミュレーション 207  
 プロセッサの統合 139  
 ブロック 184  
 ポイントツーポイント イーサネット 190  
 リアルタイム信号処理に使用 219  
 ロック共有メモリの協調シミュレーション 200  
 ハードウェア協調シミュレーション用のコンパイル 346  
 ハードウェア生成モード  
 EDK pc96 138  
 HDL ネットリスト 138  
 ハードウェア/ソフトウェア協調設計 135

## 例

EDK を使用 164  
 MicroBlaze プロセッサ システムの設計とシミュレーション 156  
 System Generator で MicroBlaze ペリフェラルを作成 152  
 System Generator で PicoBlaze を使用 147  
 ハードウェア デバッグ  
 ChipScope Pro を使用 125  
 ハードウェアの生成 138  
 パラメータの伝搬 35

## ひ

ヒストグラム  
 タイミング解析ツール 353, 355  
 ビットストリームへのコンパイル 339  
 ビット精度 19  
 ビット単位のモデリング 22  
 非同期クロック 24

## ふ

ファンアウトの低減  
 クロック イネーブル 87  
 複数クロック アプリケーション 114  
 複数のクロック  
 複数のサイクル単位アイランドの生成 113  
 ブラック ボックス  
 HDL 協調シミュレーション  
 HDL シミュレータの設定 296  
 概要 296  
 複数のブラック ボックスの協調シミュレーション 298  
 コンフィギュレーション M 関数  
 SysgenBlockDescriptor のメソッド 292  
 SysgenBlockDescriptor メンバー変数 291  
 SysgenPortDescriptor のメソッド 295  
 SysgenPortDescriptor メンバー変数 294  
 Verilog パラメータの指定 290  
 VHDL ジェネリックの指定 290  
 エラーのチェック 291



組み合わせパス 290  
 言語選択 285  
 最上位エンティティの指定 285  
 新規ポートの追加 286  
 ダイナミック出力ポート 288  
 ブラック ボックスの API 291  
 ブラックボックスのクロック 289  
 ポート オブジェクトの取得 287  
 ポート タイプの設定 287  
 ポートのサンプリング レートの設定 288  
 ブロック ポートの定義 286

例 298

CORE Generator モジュールのインポート 299, 300  
 ModelSim を使用したアドバンス ブラック ボックス 326  
 Verilog モジュールのインポート 320  
 VHDL モジュールのインポート 313  
 暗号化された VHDL ファイルのインポート 332  
 暗号化された VHDL モジュールのインポート、シミュレーション、エクスポート 332  
 ダイナミック ブラック ボックス 322, 323  
 複数のブラック ボックスの同時シミュレーション 324  
 ブラック ボックスの HDL 要件を満たす VHDL ラッパが必要な CORE Generator モジュールのインポート 306

ブラック ボックス コンフィギュレーション ウィザード 283  
 ブラック ボックスのコンフィギュレーション

M 関数 284

フレーム ベースのシミュレーション  
 ハードウェア協調シミュレーション 207

プロセッサの統合

カスタム ロジック 136  
 ハードウェア協調シミュレーション 139

ハードウェアの生成 138

メモリ マップの作成 137

ブロックセット

ザイリンクス 20  
 ブロック マスク 35

## ほ

ポイントツーポイント イーサネット  
 ハードウェア協調シミュレーション 190

## ま

マルチレート デザイン  
 信号レートに応じた色分け表示 22  
 マルチレート モデル 23

## め

メモ  
 高パフォーマンス FPGA デザイン 86  
 メモリ マップの作成  
 プロセッサの統合 137

## も

モデリング  
 ビット単位およびサイクル単位 22

## り

リアルタイム信号処理  
 ハードウェア協調シミュレーション 219  
 離散時間システム 22  
 リセット ピン  
 [Hybrid DCM-CE] オプション 25  
 リソースの予測 36  
 リファレンス ブロックセット  
 ザイリンクス 20

## れ

レート変換ブロック 23

## ろ

ロック ピン  
 [Hybrid DCM-CE] オプション 25