

HDL Coder™ 評価ガイド (R2017b)

© Copyright 2016-2018 by The MathWorks, Inc.

内容

| | |
|--|----|
| HDL Coder™ 評価ガイド (R2017b)..... | 1 |
| 1.はじめに | 2 |
| 1.1 HDL コード生成可能なブロック | 2 |
| 1.2 モデルのセットアップ | 3 |
| 1.3 DUT とテストベンチの切り分け | 4 |
| 1.4 HDL 設定へのアクセスと操作..... | 5 |
| 1.5 HDL モデルチェッカー | 6 |
| 2. HDL 設計のための Simulink の利用..... | 6 |
| 2.1 サンプル時間とクロックの概念 | 6 |
| 2.2 効率のよい HDL のためのモデリングベストプラクティス | 7 |
| 2.3 MATLAB Function ブロックの利用 | 12 |
| 2.4 Stateflow チャートの使用 | 14 |
| 3 コード生成・検証機能..... | 17 |
| 3.1 HDL ワークフロー アドバイザー | 17 |
| 3.2 スクリプトによる実行 | 18 |
| 3.3 最適化について | 19 |
| 3.4 コード生成結果の確認 | 22 |
| 3.5 生成されたコードの検証 | 25 |
| 4 FPGA ハードウェアへの実装 | 27 |
| 4.1 基本的な設定 | 27 |
| 4.2 ブロック RAM マッピング | 28 |
| 4.3 DSP マッピング | 29 |
| 4.4 サブシステムへの分散型パイプラインレジスタの挿入 | 34 |

HDL Coder は、FPGA/ASIC 実装を目的として、Simulink®モデルもしくは MATLAB®コードから論理合成可能な VHDL®と Verilog® HDL を生成します。本ドキュメントは、Simulink によるフローに対して、効率的なハードウェアの実装設計を行うための、さらにターゲットデバイスに対して生成する HDL コードを最適化するための、ベストプラクティスを述べた導入ガイドです。本ドキュメントに付属する例は `Examples` のディレクトリの中の `openHdlExampleMenu.m` を MATLAB で実行することで、簡単にアクセスできます。

1.はじめに

Simulink では多様なシステムをモデリングすることが可能です。しかしハードウェア設計に適用する場合には、設計プロセスのできる限り早い段階でハードウェアを意識した設計検討を開始する必要があります。この章では、モデルを作る前のセットアップについて解説します。

1.1 HDL コード生成可能なブロック

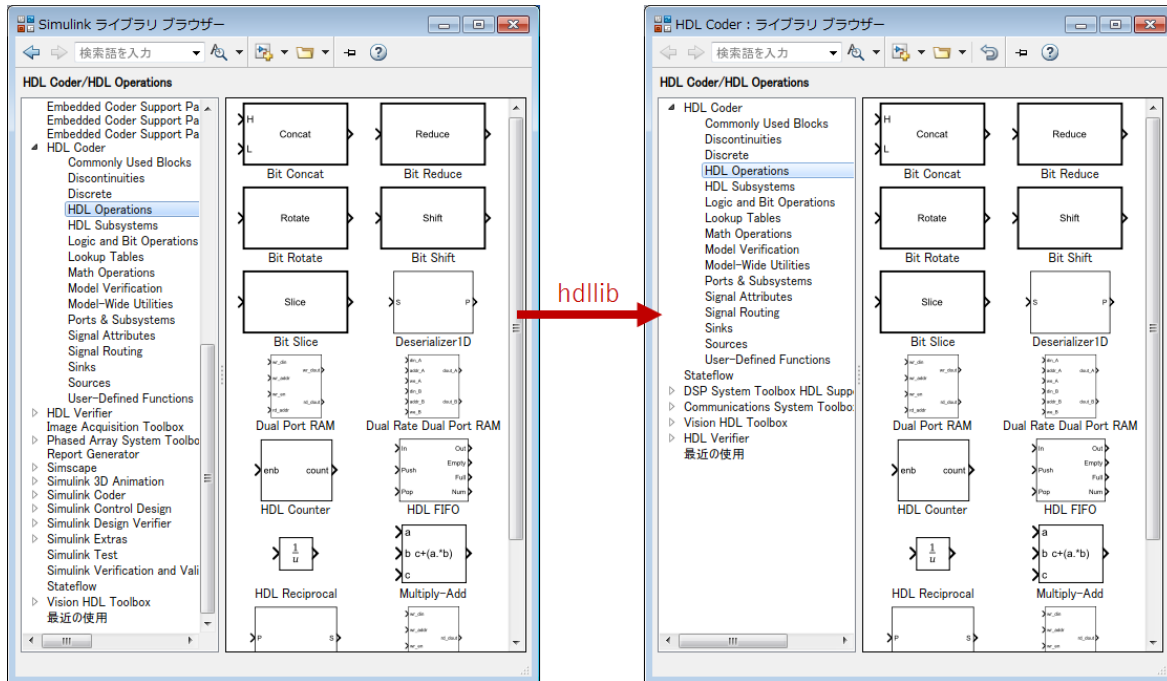
HDL コード生成が可能な Simulink ブロックは、Simulink ライブラリ ブラウザーの HDL Coder ライブラリの中にあります。ほとんどのケースにおいて、このライブラリからブロックを選択することで、標準的な Simulink ライブラリからのブロックを選択するよりも HDL Coder でコード生成可能な設定で利用することができます。

- HDL Coder ライブラリのブロックは、Simulink ユーザも使用することができます。このため、HDL Coder を利用できない Simulink ユーザとの共同作業やモデルの共有も可能です。
- サブライブラリ **HDL Coder > HDL RAMs** と **HDL Subsystems** には同期イネーブル/リセット付きの制御入力を持つ RAM、サブシステムなどの HDL アプリケーション固有のブロックが含まれています。

下記のオプションのツールボックスがインストールされている場合には、ライブラリ ブラウザーに各ツールボックス（信号処理、通信システム、コンピューター ビジョン）に関するブロックが追加され、利用できるようになります。

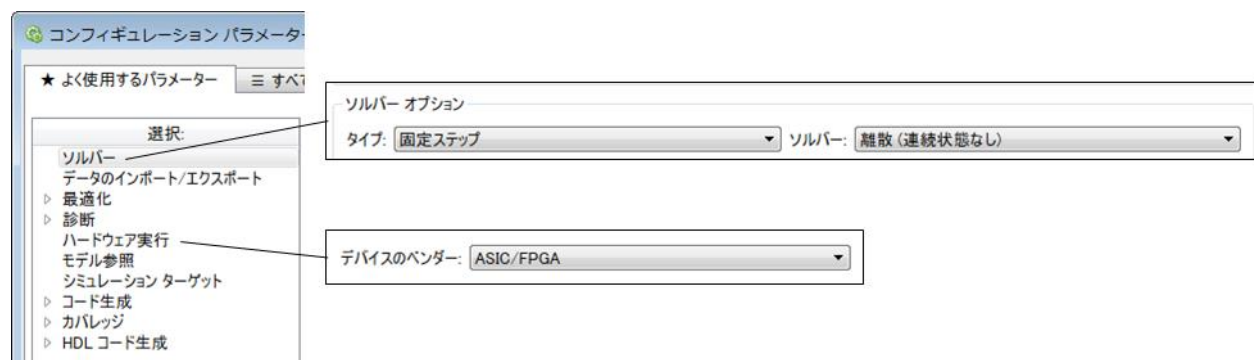
- DSP System Toolbox
- Communications System Toolbox
- Vision HDL Toolbox
- LTE HDL Toolbox (R2017b で新登場)

コマンド `hdl1lib` を実行すると、ライブラリ ブラウザーに HDL でサポートされているブロックのみを表示することができます。



1.2 モデルのセットアップ

新しいモデルを作成するには、コマンド `hdlsetup('モデル名')` で HDL 生成用のコンフィギュレーションパラメーターの設定を行うことができます。この中の多くの設定は、HDL コード生成のために必要です。例えば、マルチレートモデルにおける診断設定([モデル コンフィギュレーションパラメーター]の[診断]、[サンプル時間]、[シングルタスクレート変換]と[マルチタスクレート変換]を[エラー])や、ハードウェアターゲットデバイス設定([デバイスのベンダー]を[ASIC/FPGA])です。後者は、モデルに対して固定小数点演算の伝搬ルールを変更します。それ以外の設定、例えばサンプル時間の色設定や信号のデータ型の表示を有効にするような設定は、推奨(設計やデバッグの際に役立つもの)です。適用されたすべての設定を調べるには、コマンド `edit hdlsetup` を実行します。



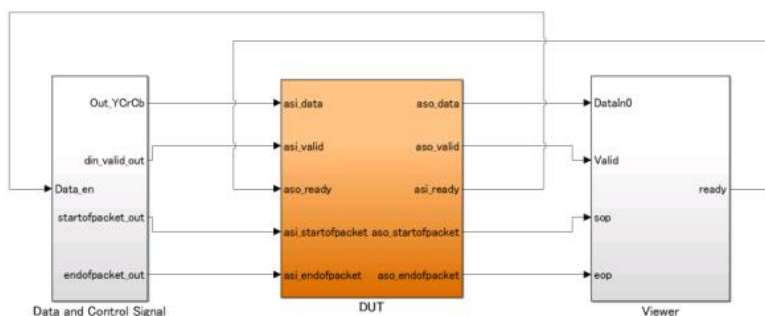
さらに、HDL コード生成用に事前に用意されたテンプレートを使用して Simulink モデルを作成することもできます。Simulink スタートページで、HDL Coder や Vision HDL Toolbox セクションまでスクロールダウンして、さまざまなテンプレートを選択す

ることができます。



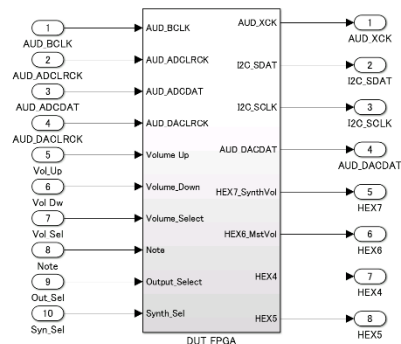
1.3 DUT とテストベンチの切り分け

サブシステム一つを Design-Under-Test (DUT/テスト対象)として定義します。DUT には、コード生成したい処理を含め、その外側にはテストベンチに相当する信号生成や可視化機能を含めます。このサブシステムは、FPGA デザイン上で下位階層に存在しても、通常はモデルの最上位の階層に配置します。



- DUT の外側にある全てのブロックは、テストベンチとして扱われます。テストベンチは実装対象ではないため、HDL コード生成対象外のブロックも含めて任意のブロックが利用できます。
 - テストベンチに対して HDL コードを生成しようとしても、テストベンチ部のブロックに対してはコード生成しません。代わりに DUT の入出力の値のみがロギングされます。

- 最上位階層に信号生成などテストベンチに該当するブロックを含めず、I/O ポートと HDL 生成可能なブロックで構成することで、モデル全体を DUT として構成することができます。この場合には、テストベンチは生成されません。



- DUT 内部ではさらに、機能、サンプルレート、パワーアーキテクチャ、分業作業などに基づいて複数のサブシステムにパーティショニングができます。初期設定では、各サブシステムに対して個別の HDL ファイルが生成されます。

1.4 HDL 設定へのアクセスと操作

Simulink のコンテキストメニューを使用して HDL の多数の設定と操作にアクセスできます。ブロックを右クリックし、[HDL コード] の [HDL ブロックプロパティ] を選択して、ブロックの HDL 固有のプロパティを設定します。このプロパティではブロックごとの回路アーキテクチャやリソース共有などの設定ができます。[HDL コードプロパティ] を選択すると、[コンフィギュレーションパラメーター] ダイアログボックスの[HDL コード生成]セクションに移動し、モデルレベルの HDL プロパティ、最適化、デバイスベンダー、ターゲットおよびテストベンチ生成オプションを設定できます。

● モデルの互換性のチェック

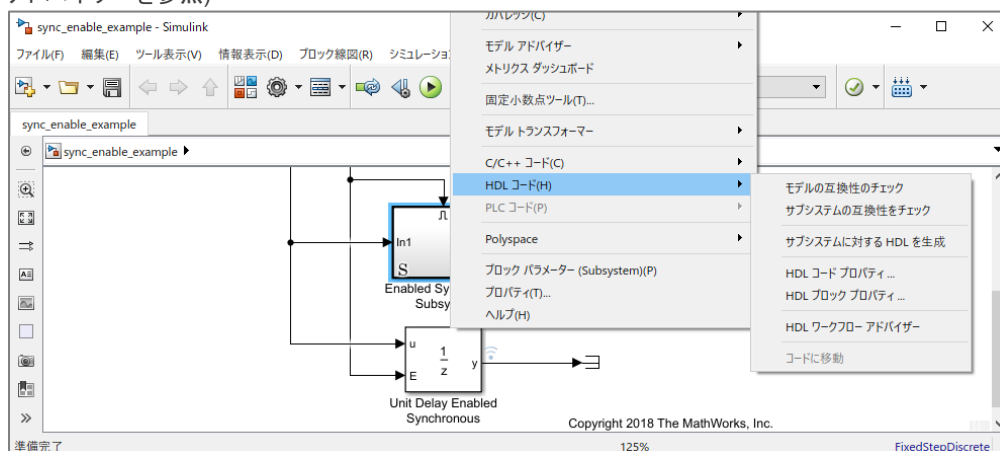
HDL モデルチェッカーを利用して HDL の互換性や効率を解析することができます。(詳細は 1.5 HDL モデルチェッカーを参照)HDL ブロックプロパティと HDL コードプロパティ
HDL コード生成に影響のあるブロックレベルのプロパティとモデルレベルのプロパティを設定します。

● サブシステムの互換性をチェックとサブシステムに対する HDL コードの生成

HDL の互換性を素早く確認し、サブシステムの HDL コードを生成します。

● HDL ワークフローアドバイザー

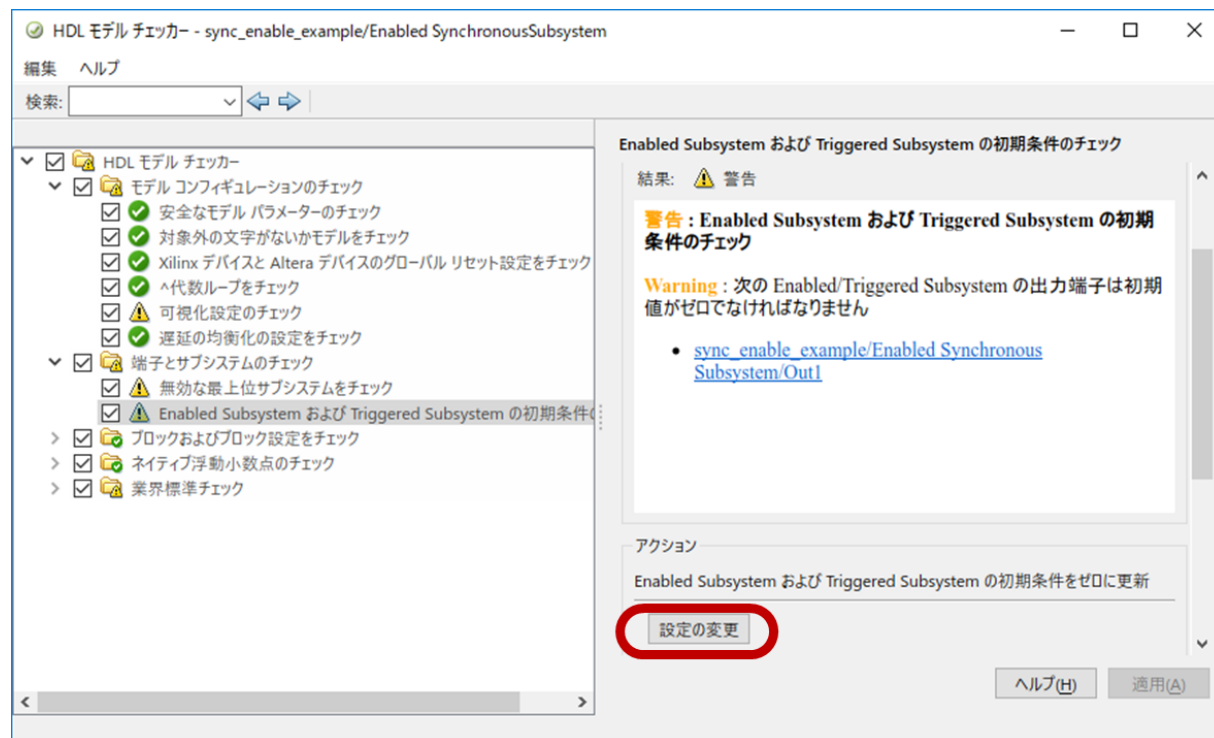
HDL ワークフローアドバイザーを利用して FPGA の論理合成などを行います。(詳細は 3.1 HDL ワークフローアドバイザーを参照)



1.5 HDL モデルチェッカー

R2017b の新機能として、HDL モデルチェッカーを使ってモデルの HDL 互換性を確認できます。また、モデルの記述がドキュメントに記載のベストプラクティスに沿っているか確認することもできます。さらに、モデルチェッカーの UI を通して、モデルが常に推奨設定になるように更新することができます。

HDL モデルチェッカーを起動するには、DUT サブシステム上で右クリックをし、[HDL コード] の [モデルの互換性のチェック] を選択します。詳細はドキュメントの [HDL モデルチェッカー](#) のチェックリストを参照してください。



2. HDL 設計のための Simulink の利用

Simulink で設計する際に、ハードウェア実装を常に意識する必要があります。例えば、ハードウェアは、クロック(単一もしくは複数)を必要とし、固定小数点データで効率的に処理する必要があります。また、どのように設計するかによって、リソース使用率やパフォーマンスに影響するハードウェアのアーキテクチャに重大な影響を与える可能性があります。また、MATLAB Function ブロックや Stateflow ブロックを用いたほうが効率的に設計できる場合があります。この章では、ハードウェア実装を意識した Simulink の適切なモデリング方法について解説します。

2.1 サンプル時間とクロックの概念

「クロック信号はどのようにモデリングするのでしょうか？」これは初めて Simulink を利用するハードウェア設計エンジニアの方からよく聞かれる質問です。クロック信号の考え方は以下のとおりです。

- クロックやリセットなどのグローバル信号は、Simulink のモデル内に明示されません。、コード生成時に生成されます。その代わりに、Simulink 上ではサンプル時間を用いてクロックサイクルを表現します。
- シングルレートモデルでは、HDL の1クロックサイクルが Simulink 上の1サンプルに相当します。自身の好みや設計要求に合わせて、相対値(例えば Simulink 上の 1 秒を 1 HDL クロックとする)、もしくは絶対値(例えば Simulink 上の 10e-9 秒を 10 ns クロックとする)のいずれも利用可能です。
- マルチレートモデルに対しては、1番早いサンプル時間が HDL 1クロックサイクルに相当します。

- マルチレートモデルで、コンフィギュレーションパラメーターの[HDLコード生成]、[グローバル設定]の[クロックの入力]をデフォルトの[単一]に設定している場合、遅いサンプル時間で動作するブロックも HDL の中では同じクロックを使用します。但し、N クロックサイクル毎にアクティブになるクロックイネーブル信号でゲーティングされます。
- マルチレートモデルで、コンフィギュレーションパラメーターの[HDLコード生成]、[グローバル設定]の[クロックの入力]を[複数]に設定している場合、複数の同期クロック信号で生成できます。
- Sharing Factor などのいくつかの最適化設定と、Newton-Raphson 法を用いた Sqrt ブロックなどの HDL コード生成のためのブロック設定は、オリジナルのモデルの中にはない、サンプル時間(クロック)を生成します。この場合には、生成されたうちで最も速いサンプル時間が 1HDL クロック(マスタークロック)にマッピングされます。
- MATLAB 変数でサンプル時間、サンプル比を設定しておく(例 $T_s = 10e-9$, $upsamp = 4$)、全てのサンプル時間を簡単に変更できます。

2.2 効率のよい HDL のためのモデリングベストプラクティス

R2016b から、HDL Coder は単精度浮動小数点演算の合成可能なコードを生成できるようになりました。これは、高ダイナミックレンジの計算や初期のプロトタイプ HDL の生成に役立ちます。この機能の詳細については、[ビデオページ](#)にある「FPGA/ASIC のネイティブ浮動小数点実装」を参照してください。また、「2.2.6 ネイティブ浮動小数点コード生成を使う際のベストプラクティス」で使い方を紹介します。

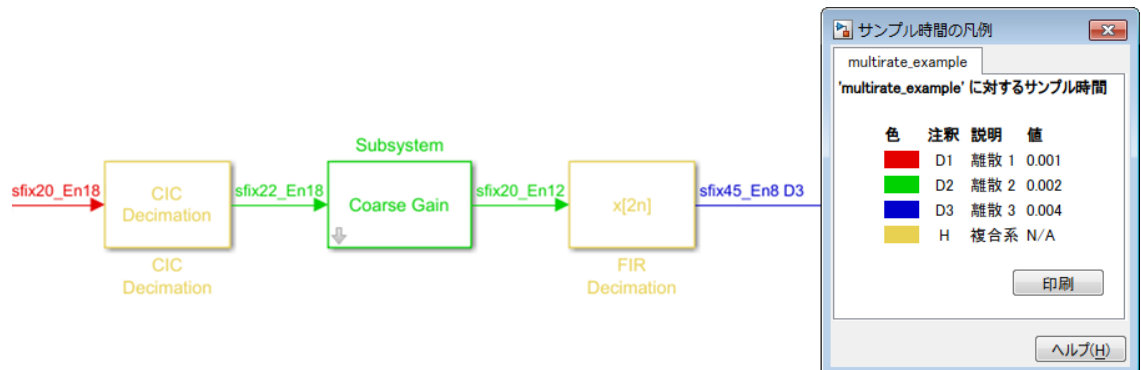
2.2.1 固定小数点設定

ほとんどのデータ型および演算では、必要な演算精度を満たす最小の固定小数点データ型に変換すると、デザインのリソースと回路間のレイテンシが少なくなります。HDL Coder を使って開発を行う際に、以下のポイントは役立ちます。

- モデルウィンドウのメニューバー[情報表示]、[信号と端子]、[端子のデータ型]を有効にすることで、固定小数点設定と伝搬を可視化することができます。
- 2 進小数点スケールのみ HDL コード生成をサポートします。
- 固定小数点は 128 ビットまでサポートしています。
- 1 ビットデータに対して、制御信号などの論理演算に対しては Boolean を使用します。算術演算に対しては `ufix1` を使用します。算術演算を Boolean データで行った場合、予期しない結果につながる場合があります。
- データ型の設定 `Inherit: Inherit via internal rule` は、完全精度の固定小数点演算の語長および小数部の長さを自動的に計算します。計算が HDL 用に最適化されるためには、[コンフィギュレーションパラメーター] [ハードウェア実行] の [デバイスのベンダー] を [ASIC / FPGA] に設定します。
- 丸めや飽和ロジックはハードウェアリソースを消費します。タイミング収束を作り出すためのクリティカルパスになる可能性があります。可能ならば[整数丸めモード] を [負方向] にして[整数オーバーフローで飽和]のチェックボックスを無効にします。
- 実装すべきアルゴリズムが丸めや飽和を必要とする場合は、演算と丸め/飽和ロジックの間にパイプラインレジスタを挿入して、タイミングパフォーマンスを向上させます。次の図のように、乗算を完全精度になるよう設定し、パイプラインレジスタ後に Data Type Conversion ブロックを用いて、明示的に丸め/飽和处理します。

2.2.2 マルチレートモデル

- モデル ウィンドウのメニューバーの[情報表示]、[サンプル時間]で示されるサンプル時間の凡例とサンプル時間に対応した色を用いて、モデル中の異なるサンプルレートを可視化します。最も速いサンプル時間は常に赤色で示されます。



- サンプルレートを変更するには、以下のブロックを以下の設定で使用します。それ以外の設定は HDL コード生成に対応していません。

- Rate Transition ブロック

- ☒ データ転送中の整合性を確保
- ☒ 確定的にデータ転送を確保 (最大遅延)

- Downsample ブロック (DSP System Toolbox)

- Input processing: Elements as channels (sample based)
- Rate options: Allow multirate processing

- Upsample, Repeat ブロック (DSP System Toolbox)

- Input processing: Elements as channels (sample based)

- ダウンサンプリング

例: `downsample_upsample_example.slx`

- Rate Transition ブロック、もしくは[Sample offset (0 to K-1)]が[0]の Downsample ブロックを用いた場合、遅延無しダウンサンプリングが実行されます。以下の例のように、値 2 が入力された場合、遅延なく出力値が 2 になります。この組み合わせパスは、タイミング性能に影響を及ぼすバイパス・ロジックになる可能性があります。

| | | | | | | | | | |
|------------------------|-----|---|-----|-----|-----|-----|-----|-----|-----|
| input | 002 | 0 | 001 | 002 | 003 | 004 | 005 | 006 | 007 |
| downsample output | 002 | 0 | | 002 | | 004 | | 006 | |
| rate transition output | 002 | 0 | | 002 | | 004 | | 006 | |

- Downsample 又は Rate Transition ブロックの後に Delay ブロックを置くことで、より効率の良い HDL を生成することができます。これによりモデルから組み合わせパスが除かれ、HDL Coder が HDL 中のバイパス・ロジックを削除できるようになります。
- [Sample offset (0 to K-1)]が 0 より大きい場合、バイパスは形成されないため、Downsample ブロックの後に Delay ブロックを追加する必要はありません。
- アップサンプリング

例: [downsample_upsample_example.slx](#)

- Upsample ブロックも前記のダウンサンプリングの場合と同様の組合せパスを含みます。より効率の良い HDL を生成するには、Repeat ブロック(HDL 上は wire になります)又は Rate Transition ブロックを使用します。

2.2.3 条件付サブシステム

例: [enabled_subsystem_example.slx](#)

- Enabled Subsystem 及び(立上りか立下りいずれかのトリガーを用いた)Triggered Subsystem は HDL コード生成でサポートされていますが、それらは通常のサブシステムの中に存在する必要があります。
- R2016a 以降のバージョンでは、Enable Subsystem は HDL Coder> HDL Subsystems ライブラリの State Control ブロックまたは、Enabled Synchronous Subsystem ブロックを使用して、[同期モード]でを使用することを推奨します。こちらのほうがコードの生成効率が良く、動作が実際の FPGA の Enable 付き FF と同じであるためです。詳細は次の節の Synchronous サブシステムを参照して下さい。
- Enabled Subsystem 及び Triggered Subsystem の出力ポートは出力の初期値を[0]に設定しなければなりません。(Simulink ライブラリから Enabled Subsystem/Triggered Subsystem ブロックを持ってきた場合の初期出力は[]に設定されています。)
- [同期モード]に設定されていない Enabled Subsystem 及び Triggered Subsystem は、前節に記載のダウンサンプリングの場合と同様の組合せ経路を含みます。より効率の良い HDL を生成するには、これらのサブシステムの出力に Delay ブロックを置きます。HDL Coder は自動的に最適化して HDL 中のバイパス回路を除去します。

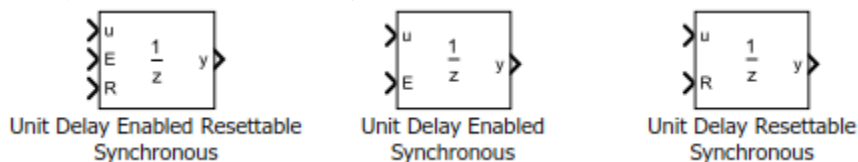
2.2.4 同期イネーブルと同期リセット

例: [sync_enable_example.slx](#)

ローカルの同期イネーブル信号と同期リセット信号は、制御パスロジック、電源管理などのハードウェアデザインで頻繁に使用されます。Simulink ではデザインに応じて複数の方法で同期イネーブルおよび同期リセットをモデル化することができます。



- Delay ブロックと Discrete FIR Filter ブロックはオプションでイネーブルとリセットの端子を提供しています。しかしながら、これらの端子はデフォルトで同期としては振る舞いません。同期イネーブルおよび同期リセットとしてモデル化するには、State Control ブロックをサブシステムに配置し、状態制御を[同期]として設定します。
- あるサブシステム内の複数のブロックを配線なしで同期イネーブル/リセットとしてモデル化したい場合、同期設定にした State Control ブロックと一緒に、Enabled Subsystem および Resettable Subsystem を使ってください。
- 単体のレジスタを表現する場合、HDL Coder > Discrete ライブラリのブロックが使用できます。これらのブロックに対して、State Control ブロックは不要です。



ヒント:: MATLAB の古いバージョンで作成したデザインに Unit Delay Enabled ブロックを含む場合、同期イネーブル付きに設定した Delay ブロックか、Unit Delay Enabled Synchronous ブロックで置き換えることを検討してください

い。古いバージョンの Unity Delay Enabled ブロックは同期サブシステムの中では動作せず、Simulink ライブラリ ブラウザーからもすでに削除されています。

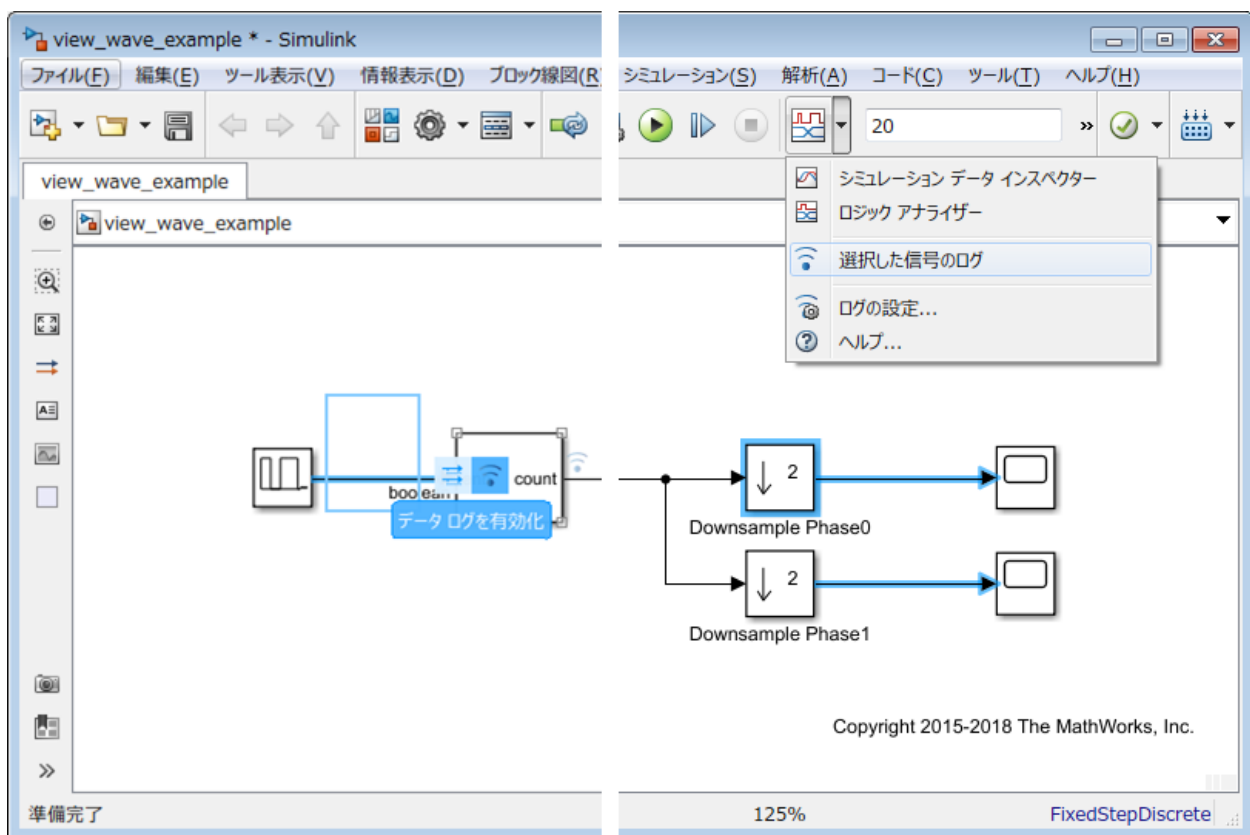
State Control ブロックと Synchronous Subsystem は、Simulink ライブラリ ブラウザー > HDL Coder > HDL Subsystems ライブラリにあります。詳細については、[「State Control ブロックを使用する同期サブシステムの動作」](#)のドキュメントを参照してください。


2.2.5 ロジック アナライザー

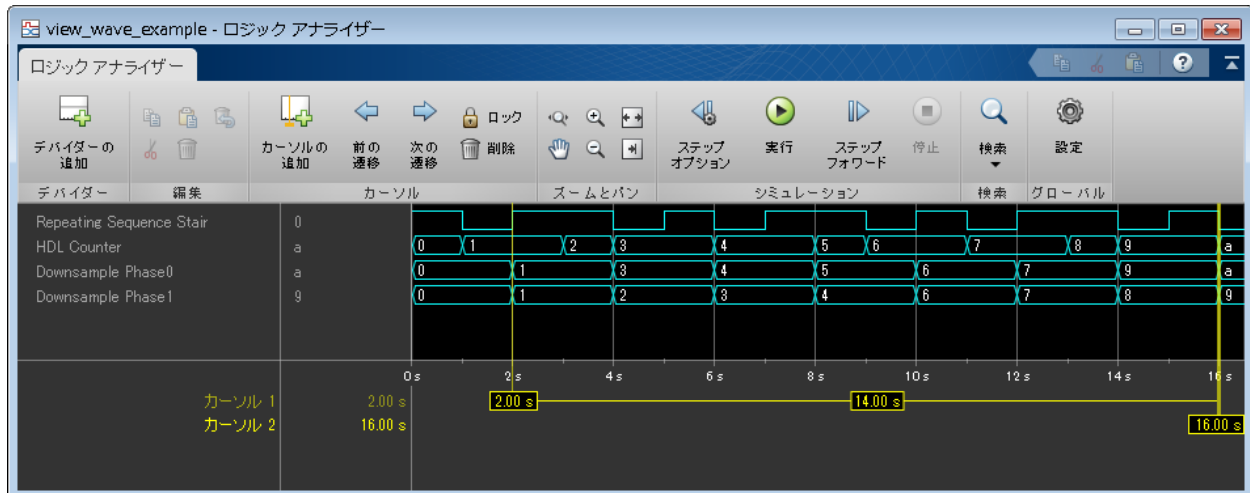
例: view_wave_example.slx

R2016b では、DSP System Toolbox の新しいロジック アナライザーにより、Simulink 信号をデジタル波形として簡単に表示できます。ロジック アナライザーを使用するには:

- モデルツールバーまたは信号線を右クリックすると表示されるポップアップメニューを使用して、1 つまたは複数の信号を選択してデータログを有効化します。



- モデルをシミュレーションし、[ロジック アナライザー]ボタン  をクリックして波形を表示します。



2.2.6 ネイティブ浮動小数点コード生成を使う際のベストプラクティス

- HDL Coder > HDL Floating Point Operations** ライブラリのブロックを使用する
 ネイティブ浮動小数点モードでの HDL コード生成向けに設定された Simulink ブロックで構成されています。
- 固定小数点タイプと浮動小数点タイプを使い分ける
 浮動小数点設計は精度と高ダイナミックレンジの点において、固定小数点よりも優れていますが、ターゲットハードウェアの面積を多く消費する可能性があります。面積の使用率を下げるため、モデルの一部分にのみ浮動小数点を使用することを検討してください。例えば、コントロールパスには固定小数点を使い、高ダイナミックレンジが必要なデータパスには浮動小数点を使います。浮動小数点と固定小数点の区画の間には Data Type Conversion ブロックを使ってください。
- レイテンシをカスタマイズする
 ネイティブ浮動小数点演算はハードウェアのパフォーマンスを最適化するために、生成されるコードにパイプラインレジスタを挿入します。レイテンシとスループットのトレードオフを考慮しながらモデルやブロック単位でレイテンシの挿入ポリシーをカスタマイズできます。詳細はドキュメント「[ネイティブ浮動小数点のレイテンシに関する考慮事項](#)」と「[浮動小数点演算子の最小レイテンシと最大レイテンシ](#)」をご覧ください。
- リソースシェアリングなどの最適化を活用する
 浮動小数点設計の面積とタイミングパフォーマンスを改善するために最適化を有効にすることを考えてください。下記の設定では、浮動小数点の加算器や乗算器、その他のリソースの共有をサブシステムの[リソースシェアリング](#)と同様に設定できます。



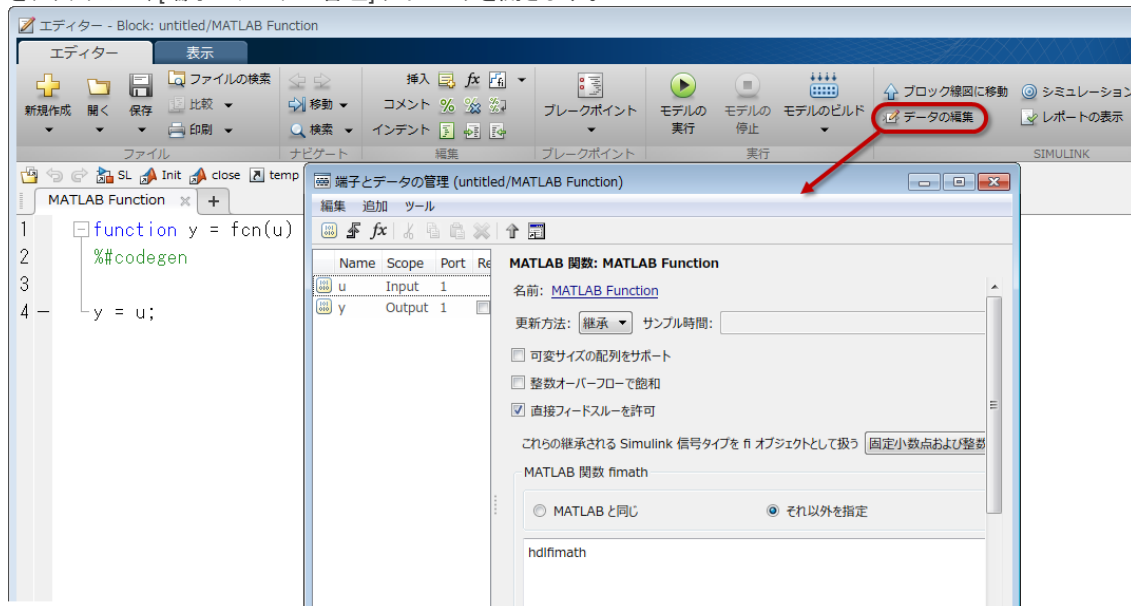
その他のベストプラクティスについては、ドキュメンテーション「[HDL Coder のネイティブの浮動小数点のサポート](#)」をご覧ください。

2.3 MATLAB Function ブロックの利用

例えば if-else のマルチプレクサや 単純な有限状態マシンなど、標準の Simulink ブロックでは利用できない制御回路を作る必要がある場合、Simulink モデルに組み込まれた MATLAB Function ブロックを用いて作成できます。これは専用回路を生成するために MATLAB を用いながら Simulink のアルゴリズムを統合できる簡単な方法です。このフローを開始するには、ライブラリの[HDL Coder]、[User-Defined Functions]にある MATLAB Function ブロックを自身のモデル中に配置し、MATLAB コードを編集するためにダブルクリックします。

2.3.1 ブロック設定

- 関数名の下に `%#codegen` のコンパイル指示を記述します。これによりモデルのコンパイル中にエラーが検知された場合、より良い診断メッセージを得ることができます。HDL Coder ライブラリからブロックを選択した場合は予め付加されています。
- データ型や `fimath` といったブロックプロパティを定義し、関数の入力パラメータを指定するには、[データの編集]をクリックして、[端子とデータの管理]ウィンドウを開きます。

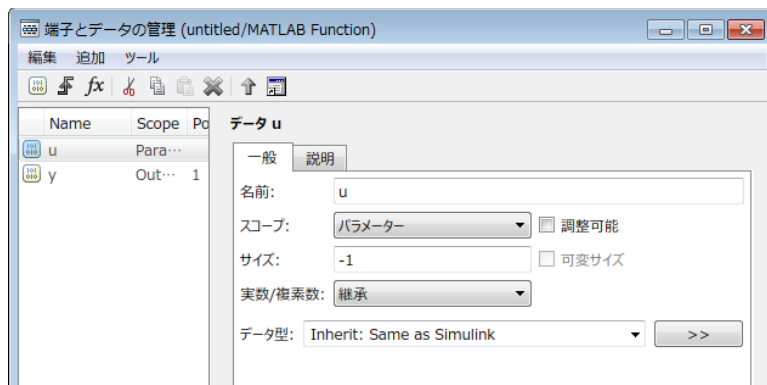


- 固定小数点の節で説明したように、丸めと飽和の回路はハードウェアリソースを使い、タイミング性能に影響するかもしれません。それらが不要でない場合、
 - [整数オーバーフローで飽和]を無効にします。
 - [MATLAB 関数 fimath]を[それ以外を指定]に設定し、エディット画面に[hdlfimath]を入力します。hdlfimath は以下のプロパティを持ちます。

RoundingMethod: Floor
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision

- `uint16` のような組み込みデータ型を `fi` オブジェクトと同様に扱ってビット演算を行うような場合は、[これらの継承される Simulink 信号タイプを `fi` オブジェクトとして扱う]を[固定小数点および整数]に設定することを推奨します。この設定により、16 ビットの `fi` オブジェクトに `int16` データを代入する際に生じるようなエラーを回避することができます。

- MATLAB Function ブロックの処理が全てレジスタでパイプライン化されたロジックの場合、[直接フィードスルーを許可]のチェックを外してください。この設定により、MATLAB Function ブロックに遅延が含まれることが認識されることで、フィードバックループの中でも代数ループが発生せずに、その MATLAB Function ブロックを使うことができます。
- MATLAB Function ブロックの入力引数をブロックへの入力ポートの代わりにパラメータ(例えば MATLAB ワークスペース変数)として設定するには、左側の入力を強調表示して[スコープ]を[パラメータ]に変更します。また、調整可能パラメータは HDL コード生成に未対応のため[調整可能]のチェックボックスは無効化します。



- 最後に、コードの再利用とパイプライン化を容易にするため、大量の MATLAB コードは複数のブロックや独立した MATLAB 関数(.m) に分割することを検討します。

2.3.2 MATLAB コードを用いたレジスタのモデリング

MATLAB では、次の 2 つの方法でレジスタをモデル化することができます。

1. MATLAB コードから適切にレジスタ推定する為に、Persistent 変数を使って以下のルールに従ってください。
 - isempty 関数を使用して、初期値とデータ型を指定します。
 - Persistent 変数に新しい値を代入する前に変数から読みます。それ以外の場合の HDL コードは、レジスタへの入力が出力されるか、またはレジスタ記述がまったく生成されないことがあります。コードに順序ロジックのみが含まれている場合は、このルールを適用するための[直接フィードスルーを許可]をオフにすることができます。

```
function y = fcn(u)
    %#codegen

    persistent u_d;
    if isempty(u_d)
        % 初期値の設定
        u_d = cast(0, 'like', u);
    end

    % 前のサンプルで入力された値を出力
    y = u_d;

    % 新しい値を代入する
    u_d = u;
```

2. `dsp.Delay`: DSP System Toolbox のこの関数は、上述の方法よりもレジスタを簡単に作成することができます。ただし、フィードバックループの遅延をモデル化するために使用することはできません。
3. `coder.hdl.pipeline`: このプラグマを使用すると、式の出力にパイプラインレジスタの追加を指定できます。モデルシミュレーションの動作を変更することなく、生成された HDL にレジスタが追加されます。例えば、算術演算の出力に 2 つのパイプライン状態を追加するには次のように記述します。

```
y = coder.hdl.pipeline(A*D + B*C, 2);
```

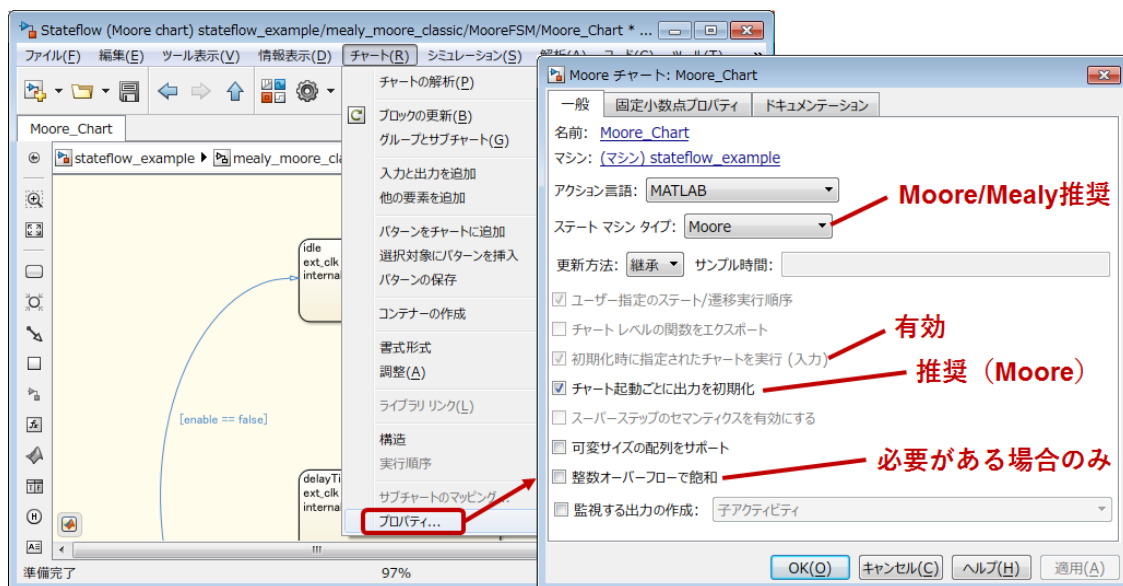
2.4 Stateflow チャートの使用

Stateflow は、複雑な有限状態マシン(FSM)を視覚的に記述することができます。デザインが複雑になるほど、アルゴリズム間のすべてのやりとりを管理するのに役に立ちます。設計資産の再利用のしやすさという観点からも有益です。アプリケーション例としては、以下のようなものがあります。

- ブロックとロジック間のインターフェース (例えば SPI)
- リンク層以上の異なるブロック間の標準接続プロトコル(例えば PCI、独自のプロトコル)
- DO-254/ISO26262/IEC61508 などの要求仕様のトレーサビリティを必要とするプロジェクト

2.4.1 チャートのセットアップ

HDL コード生成用の Stateflow チャートを設定するには、以下のガイドラインに従ってください。チャートのプロパティダイアログボックスを開くには、Stateflow エディタメニューで[チャート]の[プロパティ]を選択します。

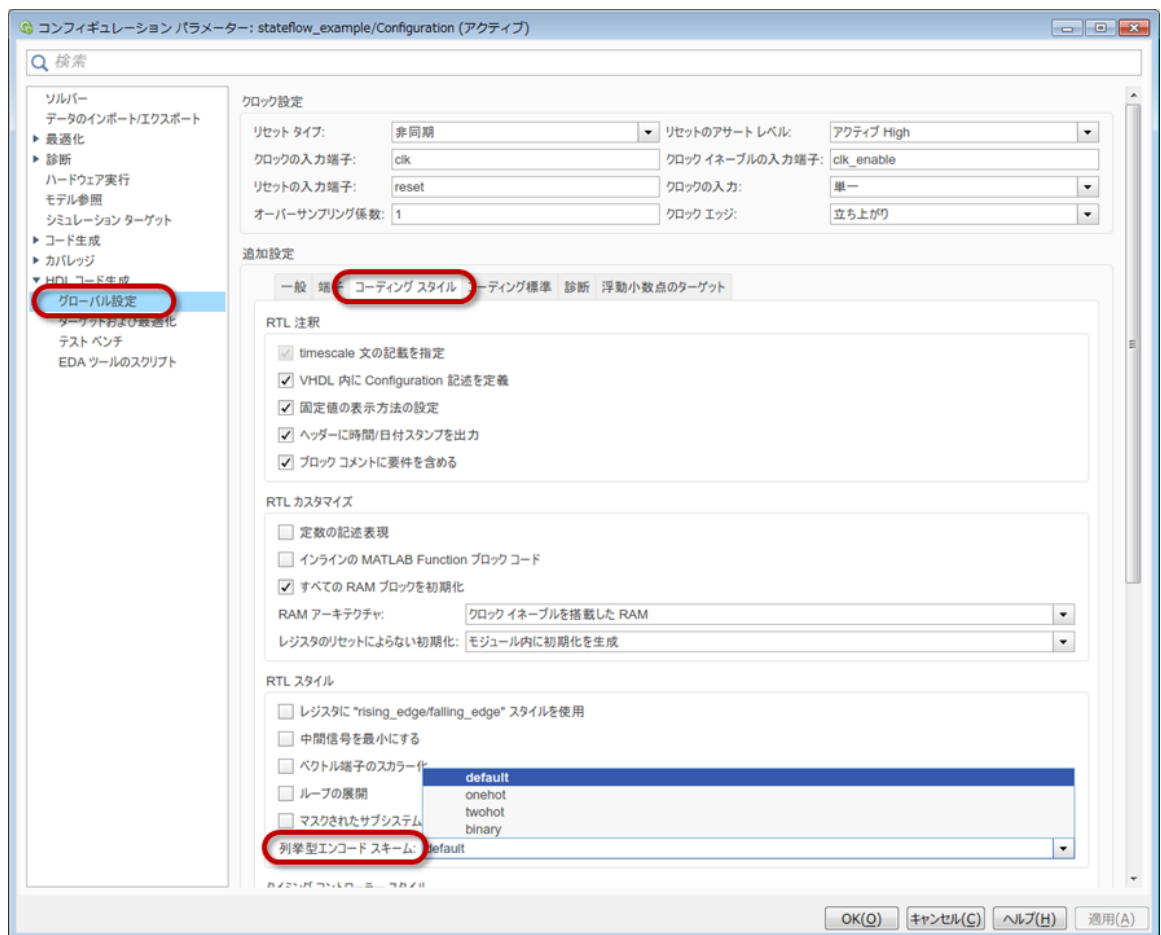


- [アクション言語]を[MATLAB]に設定します(デフォルト)。[C]のアクション言語は設定できますが、推奨されません。
- [初期化時に指定されたチャートを実行 (入力)]オプションを有効にします。
- Moore ステートマシンの場合は、[チャート起動ごとに出力を初期化]オプションを有効にします。
- 同期サブシステム(2.2.4 項を参照)内で使用する場合は、ステートマシンタイプを Moore に設定する必要があります。
- 固定小数点セクションで説明したように、丸めと飽和ロジックはハードウェアリソースを消費し、タイミングパフォーマンスに影響を与える可能性があります。必要がない限り:
 - [整数オーバーフローで飽和]をオフにする
 - MATLAB チャートの `fimath`(固定小数点のプロパティタブの下)を以下のプロパティを持つ `hdlfimath` を設定します。

```
>> hdlfmath
ans =
```

```
RoundingMethod: Floor
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

- uint16 などの組み込みデータ型でビット操作を実行する場合など、[これらの継承される Simulink 信号タイプを fi オブジェクトとして扱う](固定小数点プロパティタブの下)を[固定小数点および整数]に設定する必要がある場合があります。
- one-hot などのステートマシンのエンコーディングスキームを指定するには、グローバルパラメータ**列挙型エンコードスキーム**を使用します。これは、[グローバル設定]、[コーディングスタイル]、または HDL ワークフローアドバイザーの[3.1.2 詳細オプションの設定]にあります。

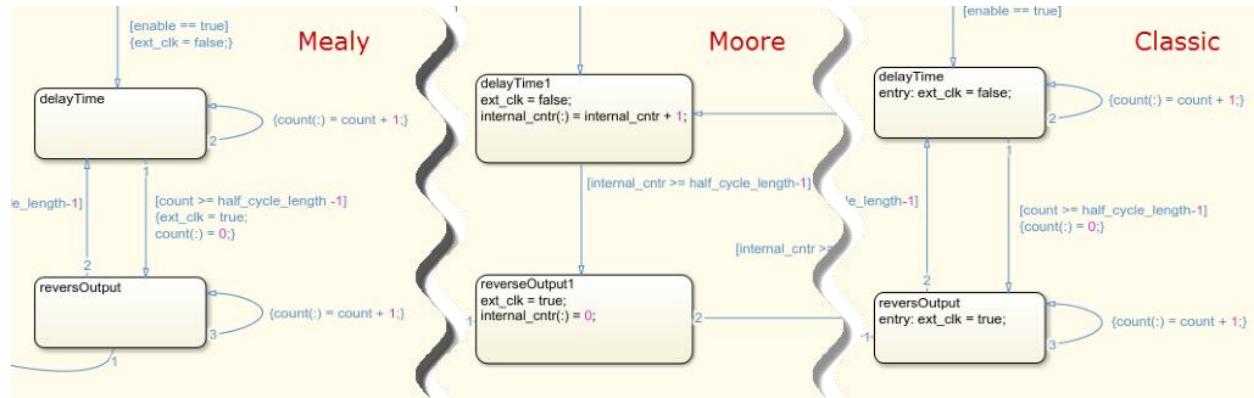


2.4.2 Mealy、Moore および Classic のチャート

例: stateflow_example.slx

Mealy、Moore、または Classic として、チャートの[ステート マシン タイプ]を設定できます。Stateflow は、選択されたステートマシンのタイプに応じてチャートセマンティクスを実行します。

- Mealy ステートマシンは、現在の状態と入力の関数から出力を生成します。出力は、状態遷移に沿ってのみ定義されます。多くの場合、柔軟性が高く、使用するハードウェアリソースが少なく済みます。
- Moore ステートマシンは、現在の状態のみに基づいて出力を生成します。出力は状態でのみ定義されます。通常、読みやすく、特定の操作を実行するために、より多くのハードウェアリソースを消費します。
- Classic ステートマシンは、Mealy マシンと Moore マシンの組み合わせです。Classic チャートでは Mealy と Moore の両方のセマンティクスを使用できます。HDL コード生成時にはこの設定は推奨されません。



2.4.3 ステートマシン設計時のハードウェアの考慮事項

Stateflow チャート内のコード記述はハードウェアリソースを消費する点に注意して下さい。また、回路の安全性や速度の改善とリソース削減のトレードオフを考慮する際には下記を検討して下さい。

- 不要な遷移条件と冗長ロジックを最小限に抑えます。
- Stateflow チャートの外側に大規模コンパレータや算術演算などのリソースが大きなロジックを実装し、シンプルな制御信号を使用してこれらのロジックと接続します。これにより、リソースを簡単に共有することができ、面積と速度の両方のパフォーマンスが向上します。
- デフォルト遷移を使用することで、if/else 文の else 項、case 文の default 項をもつ HDL を生成することを強く推奨します。これにより、未定義の状態遷移のために意図しないラッチの生成を防ぎます。
- Mealy チャートと Classic チャートでは、長い組み合わせデータパスを防ぐために入力にレジスタが入っていることを確認してください。

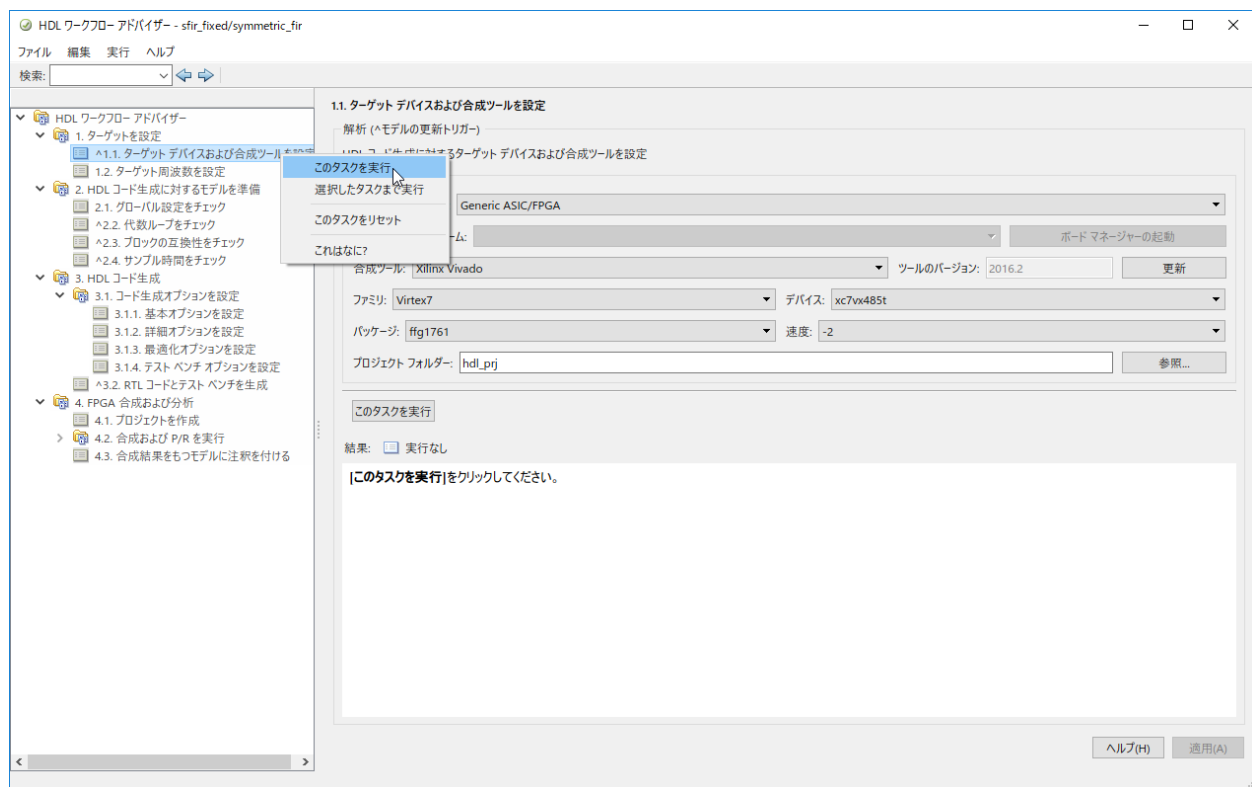
3 コード生成・検証機能

HDL 生成のための準備が一通り完了すると、次はいよいよ HDL 生成のステップになります。HDL を生成し、期待した HDL コードになっているかを解析し、必要があれば調整を行います。そして、コード生成の元になったモデルと機能的に等価であるかどうかを検証します。

3.1 HDL ワークフロー アドバイザー

HDL Coder には、HDL ワークフローアドバイザーと呼ばれるウィザードが用意されており、設計の互換性チェックと HDL コードの生成をサポートします。このガイド付きのワークフローはサードパーティの合成ツールと統合されており、以下のような各種操作を実行できます。

- 合成ツールを手動で起動せずにデザインを合成する
- FPGA-in-the-Loop を使用してハードウェア上で設計検証する
- ポートに AXI4 バスインターフェースを付与する
- SDR (software-defined radio)、ビデオ処理などのリアルタイム処理用に FPGA ボードに実装する



HDL ワークフローアドバイザーの使用を開始するには、次の手順に従います。

- 初めに、hdlsetuptoolpath を使用して、Xilinx® ISE, Vivado® または、Intel® Quartus Prime のパスを MATLAB に追加します。これは MATLAB セッションごとに実行が必要ですので、コマンドをその都度 1 回実行するか、startup.m に追加して MATLAB の起動時に自動的に実行させます。例えば、以下のようにコマンド入力を行います。

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', ...  
    'ToolPath', 'C:\Xilinx\Vivado\2016.4\bin');  
  
hdlsetuptoolpath('ToolName', 'Altera Quartus II', ...  
    'ToolPath', 'C:\altera\16.1\quartus\bin64');
```

- DUT サブシステムを右クリックして HDL ワークフローアドバイザーを起動し、次に[HDL コード] > [HDL ワークフローアドバイザー]を選択します。
- 1.1 ターゲットデバイスおよび合成ツールを設定で、目的のワークフロー、ハードウェアターゲット、合成ツールを設定します。合成による設計の反復では、デフォルトの Generic ASIC / FPGA in Target ワークフローを使用してください。注: 合成ツールを選択すると、適応パイプラインのために生成されたデザインに遅延が発生する可能性があります。詳細は 3.3.4 節を参照してください。
- 後続のステップは、選択したワークフローに従って更新されます。[このタスクを実行]をクリックして手順を順番に実行するか、特定の手順を右クリックして[すべて実行]または[選択したタスクまで実行]をクリックして複数の手順を一度に実行します。

3.2 スクリプトによる実行

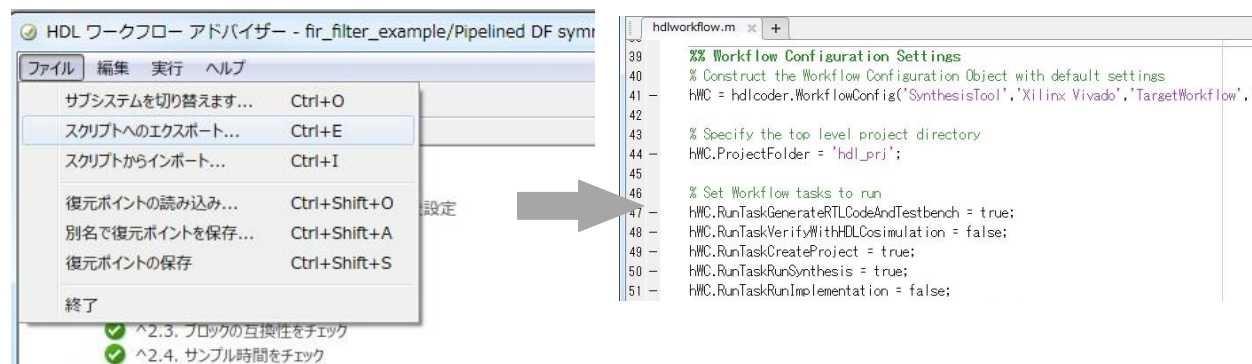
Simulink と HDL Coder は、スクリプトをベースとしたワークフローをサポートするために様々な関数を提供しています。Simulink や HDL Coder に関する各種パラメータを確認・設定したり、コード生成を行ったりすることが可能です。また、Simulink のブロック線図そのものをコマンドを使って自動的に作成することもできます。

- Simulink ブロックやモデルのプロパティを確認・設定するために、`get_param` 及び `set_param` を使用します。HDL プロパティに関しては、`hdlget_param` 及び `hdlset_param` を使用します。
- デフォルト設定とは異なる HDL プロパティ設定をエクスポートするためには、`hdlsaveparams` を使用します。
- HDL 及びテストベンチの生成には `makehdl` と `makehdltb` を使用します。

ヒント: HDL プロパティは `hdlset_param` を使ってモデルに適用することができますが、`makehdl` コマンドの引数としてコード生成時に設定することもできます。一時的に異なる設定でコード生成したい場合は後者が便利です。

より詳しい情報は、次のコマンドを実行することで参照できます。`help <command name>` もしくは `doc <command name>` で確認ください。

HDL ワークフローアドバイザーにおける各種設定や動作もスクリプトにまとめることができます。HDL ワークフローアドバイザー上で[ファイル]、[スクリプトへのエクスポート]を選択すると、現在のワークフローを再実行可能なスクリプトが生成されます。



3.3 最適化について

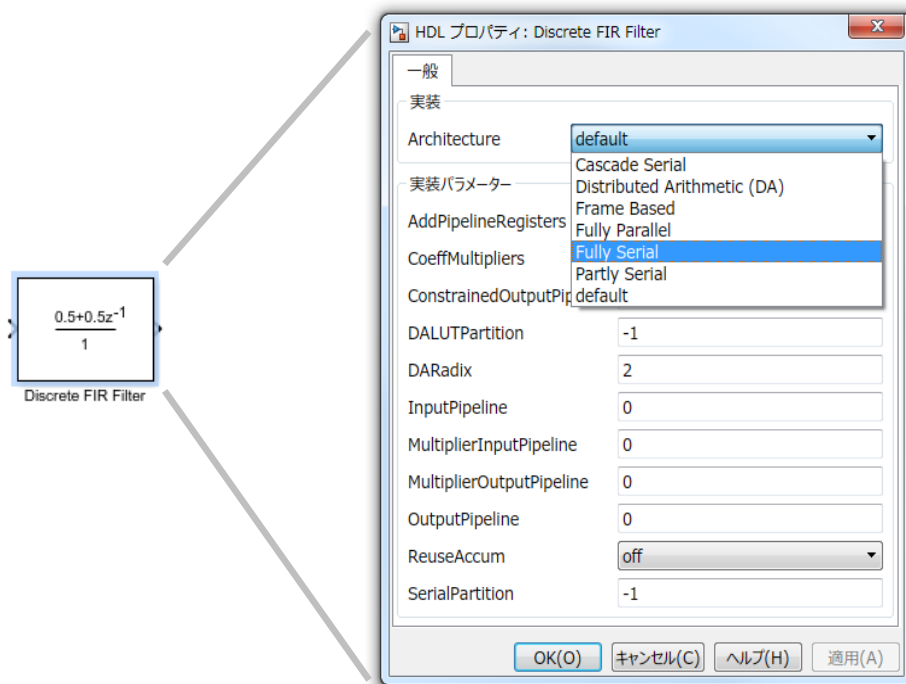
HDL Coder は速度・面積両方に関して自動最適化のオプションを提供しており、アルゴリズムモデルから実装レベルの低抽象度化することが可能です。本セッションでは一般的に良く使用される最適化の機能の概要とメリットについて説明します。(それらの機能の幾つかはデフォルトで有効になっています。)

3.3.1 一般注意事項

- HDL 最適化の設定によって Simulink のシミュレーションの動作が異なることはありません。しかしながら、Simulink モデルと生成された HDL の間にはレイテンシやデータレートの違いが生じることがあります。また、HDL ブロックプロパティでブロックのアーキテクチャを変更した場合は、数値誤差が生じる場合があります。最適化した結果やこのような差分が生じているかどうかの確認を行うための機能については 3.4 章をご覧ください。
- DSP System Toolbox で提供される FFT HDL Optimized のようなブロックは HDL コード生成に特化して作られています。「HDL optimized」が名前に含まれるブロックから生成された HDL は、ビット及びサイクル精度で Simulink 上でのシミュレーションと動作が一致します。
- Constant ブロックのサンプル時間の設定を[-1]に変更してください。デフォルト設定である inf のまま使用すると、最適化が正常に行われない場合があります。

3.3.2 ブロックレベル最適化

Simulink ブロックからデフォルト設定で HDL を生成した場合、その動作はビット及びサイクル精度で Simulink 上での動作と一致します。一方で、設計者は期待するハードウェアパフォーマンスを得るために様々な実装オプションを選択できます。対象のブロックを右クリックし、[HDL コード]、[HDL ブロックプロパティ]を選択します。



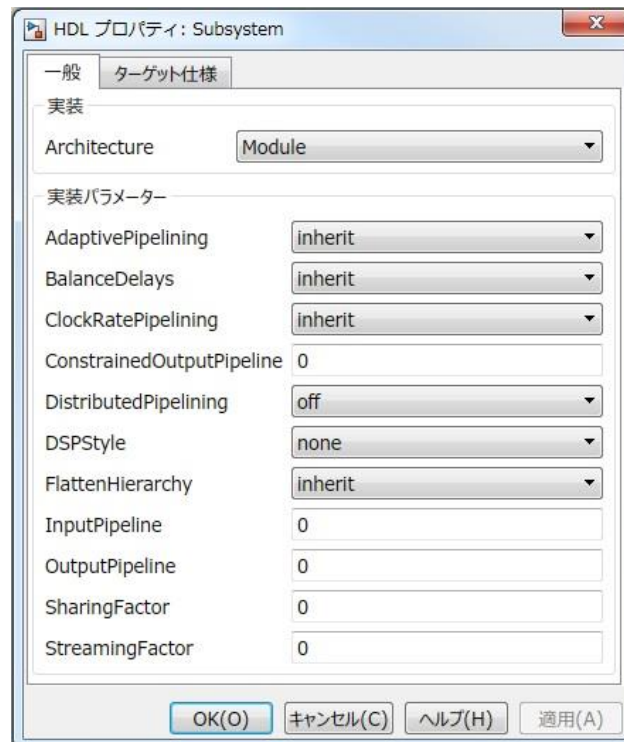
- 速度・面積に対する様々な要求に対応するために、多くのブロックは複数のアーキテクチャが選択可能です。例えば、Discrete FIR ブロックはデフォルトでは Fully Parallel アーキテクチャになっていますが、面積重視の Serial ア

アーキテクチャや、複数の入力サンプルを 1 サイクルで並列処理できる Frame Based アーキテクチャ等を選択できます。

- 幾つかのパラメータはブロックのハードウェア実装に対して大きく影響を与えます。例えば、Gain ブロックの [ConstMultiplierOptimization] 設定を [csd] もしくは [fcsd] に設定することで、定数乗算の場合はシフタと加算器によって実装できます。また、Delay ブロックの [UseRAM] パラメータを on に設定すると、遅延器が RAM にマッピングされます。
- OutputPipeline のようなパイプライン関連パラメータを使うことにより、Simulink のオリジナルのデザインを変更することなく生成される HDL にパイプラインを追加挿入することができます。また、並列のデータパスに対する遅延の調整を自動で行う機能もあります。詳しくは 3.3.4 章をご覧ください。

3.3.3 サブシステムレベル最適化

サブシステムは HDL における 1 階層に相当し、多くの最適化オプションがあります。サブシステムレベルの最適化は、ブロックレベルの最適化で行った操作と同じような手順で実現できます。(サブシステムの HDL ブロックプロパティを利用します)

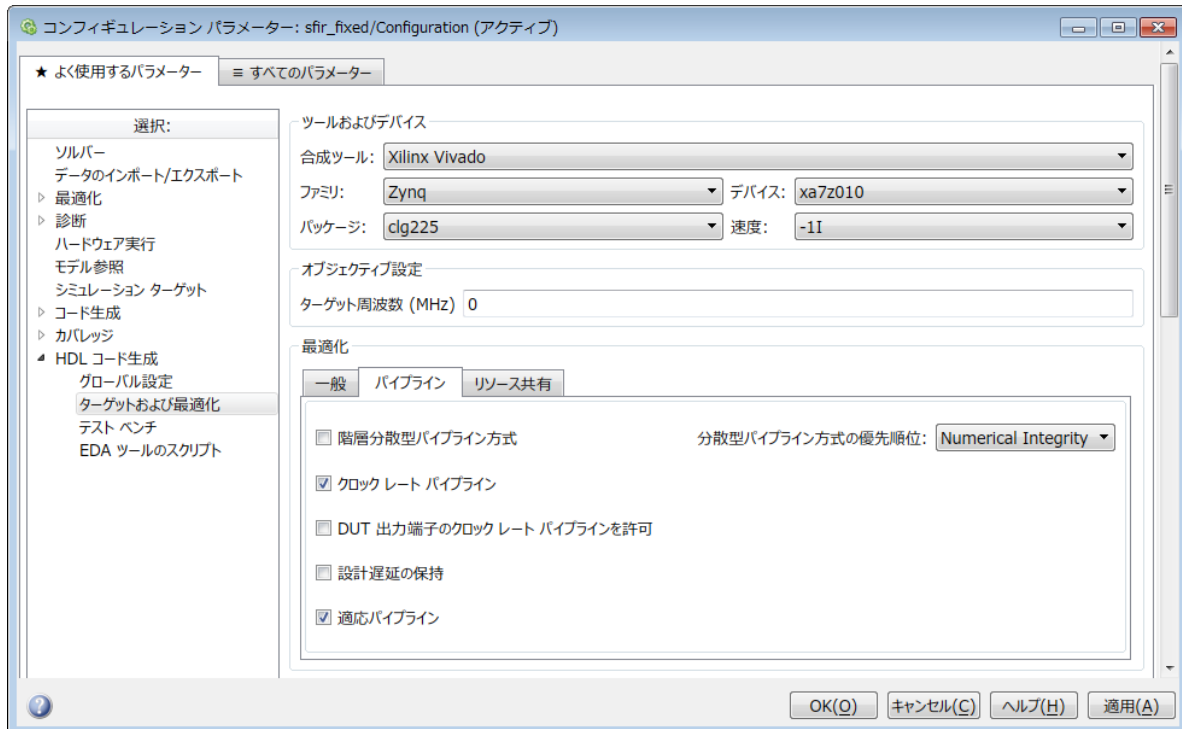


- 面積最適化
オーバーサンプリングが許容される場合、ハードウェアリソースは等価な演算を行っている部分で共有が可能です。例えば、18x18 ビットの乗算器を 100 個含むサブシステムがあり、サンプルレートが 1MHz であった場合、クロックを 20MHz まで上げると 18x18 ビットの乗算器を再利用により 5 個まで減らすことができます。このオーバーサンプリング率は [SharingFactor] か [StreamingFactor] パラメータによって設定できます。(個々の演算子に対しては [SharingFactor]、ベクトル入力に対しては [StreamingFactor] をそれぞれ設定します。)
- 速度最適化
DistributedPipelining は、InputPipeline、OutputPipeline で指定されたレジスタ及びサブシステム内の Delay ブロックを分散して移動または追加することによりクリティカルパスを削減します。FlattenHierarchy はサブシステム配下の階層を取り除き、面積最適化および速度最適化において、複数サブシステムを対象にしたより良い結果を得ることができます。

- グローバル設定や上位階層の最適化設定はデフォルトでは下位の階層に引き継がれますが、特定のサブシステムに対して異なる設定を与えたい場合は該当パラメータを inherit から on、もしくは off に設定します。

3.3.4 モデルレベル最適化

モデルレベルの最適化はコンフィグレーションパラメーターの HDL コード生成セクション、ターゲットおよび最適化ペインを利用して設定することができます。もしくは、HDL ワークフローアドバイザーのステップ 3.1.3 最適化オプション設定を利用します。



- 自動遅延均衡化
 - 最適化の設定によって HDL コード生成時にデータパスに遅延が挿入される場合、遅延均衡化の機能が自動的に並列のデータパスに対しても調整用の遅延を挿入し、生成された HDL とオリジナルの Simulink モデルで同じ結果がえられるようにします。(レイテンシは増加します。)
 - [一般]ペインにある[遅延の均衡化]はデフォルトで有効になっています。R2016 以降のバージョンでは、遅延の均衡化に失敗した場合はエラーとなります。以前のバージョンでは警告としてレポートが生成されます。
- 適応パイプライン
 - 効率的な FPGA 実装を実現するために、いくつかの Simulink ブロックは手動でパイプラインを挿入する必要があります。例えば、Product ブロックと出力の丸め回路の間には手動で Delay ブロックを挿入する必要がありますし(セクション 2.2.1)、Downsample ブロックの後段には Delay ブロックを挿入して余分な組み合わせ回路が生成されるのを防ぐ必要があります(セクション 2.2.2)。R2016b 以降、Xilinx もしくは Intel FPGA デバイスをターゲットにしている場合、適応パイプラインの機能によって HDL Coder が自動的にこれらのパイプライン挿入を行うことができます。R2017b において対応しているデバイスは Cyclone V, Stratix V, Virtex-7 speed grade -1, Zynq speed grade -1 のみです。
 - 適応パイプラインはルックアップテーブル、乗算器やレート変換ブロックに対し、HDL コード生成時に自動的にパイプラインレジスタを挿入します。ルックアップテーブルに対しては、RAM 領域にマッピングさせるためにリセットなしのレジスタが挿入されます。乗算器に対しては、論理合成ツール、ターゲット周波数や語長の設定に従って自動挿入されるレジスタの数や場所が決定されます。また、丸め設定やクリッピングが行われている場

合、レジスタは丸め/クリッピング回路の前に挿入されます。(DSP ブロック内部のパイプラインレジスタが推論されるように考慮されています)

- 適応パイプラインはデフォルトでは有効になっていますが、自動パイプライン挿入は合成ツールが選択されている場合に限り行われます。乗算器の出力レジスタは合成ツールが選択され、ターゲット周波数が 0 より大きい値に設定されている場合に限り挿入されます。適応パイプラインはサブシステムレベル、もしくはモデルレベルで、HDL ブロックプロパティにより有効/無効の設定ができます。
- 適応パイプラインに非対応の Add/Sum などのブロックが含まれていると、非対応ブロックの前後にはパイプラインレジスタが挿入されません。そのため、適応パイプラインの適用は、モデル全体ではなく、対応ブロックで構成されている一部のサブシステムのみにも留めておくのが現実的で、モデル全体に対しては、無効にしておくことを推奨します。

面積・速度最適化の結果を確認するために、最適化レポートを生成することができます。また、これら最適化を行った際には、生成モデル (gm_***) が生成されますので、これを目視して、最適化結果が意図した通りか、またシミュレーション実行して動作が正しいことを確認して下さい。レポートや生成されたモデルについては次のセクションも参照してください。

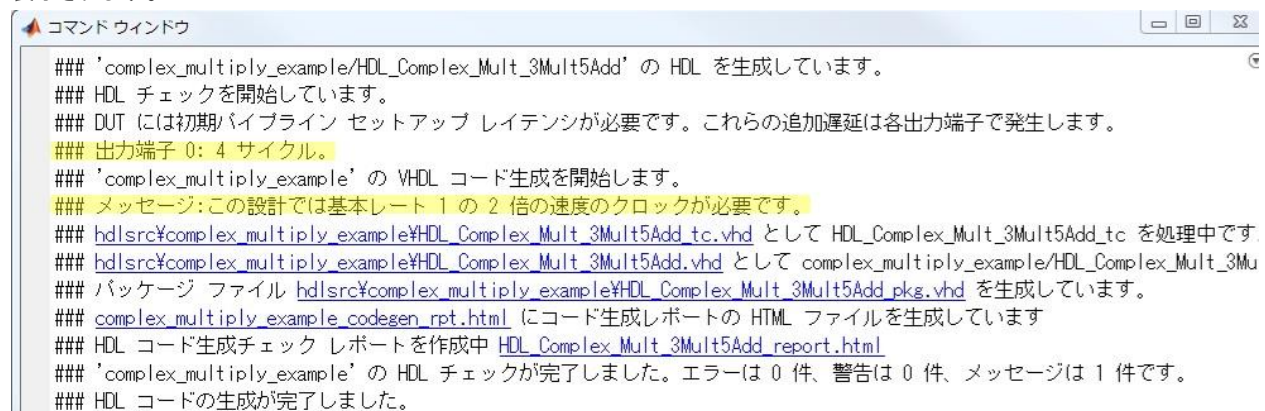
3.4 コード生成結果の確認

RTL コードに加えて、どのようなコードが生成されたのかを理解するための様々な機能が HDL Coder では提供されています。コンフィグレーションパラメーターの HDL コード生成セクションにて、レポート生成機能や、検証モデルの生成機能(3.4.3 章をご確認ください)を有効にすることができます。もしくは、HDL ワークフローアドバイザーのステップ [3.1.1 基本オプションを設定] とステップ [3.2RTL コードとテストベンチを生成] で各機能を有効にできます。



3.4.1 コード生成時メッセージ

コード生成の間、生成状況を示すメッセージが MATLAB のコマンドウィンドウ上もしくは HDL ワークフローアドバイザー上に表示されます。

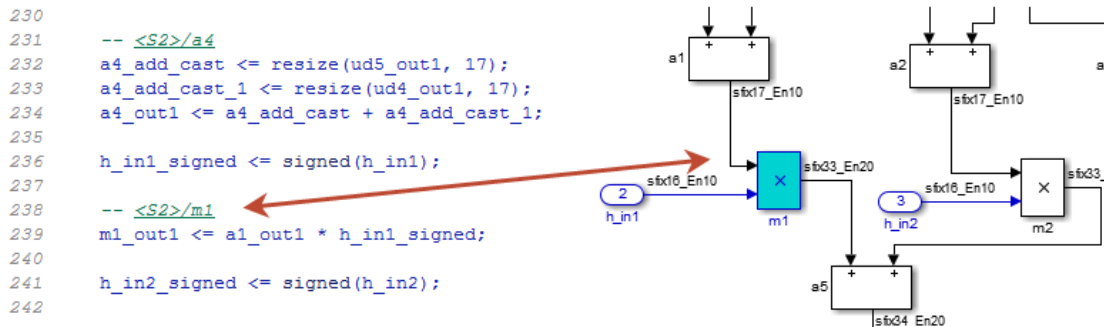


- 前の章で言及したとおり、最適化の設定によっては HDL 生成時にレイテンシが追加されたり、データレートや演算結果に数値誤差が生じたりする場合があります。生成された HDL とオリジナルの Simulink モデルの間に生じたレイテンシやデータレートの差分については、上記のようにメッセージとして MATLAB のコマンドウィンドウ上に表示されます。
- 指定した最適化を適用できなかった場合、問題の箇所が特定できるようにモデルの該当箇所をハイライト表示させるための診断スクリプトが生成されます。スクリプトへのリンクは、最適化に失敗した場合にコマンドウィンドウ上に表示されます。
- コード生成結果のサマリは、詳細が記載された HTML 形式のレポートへのリンクとともに最後に表示されます。このリンクをクリックしてレポートを立ち上げ、警告やエラーがないかどうかを確認してください。コード生成した結果にエラーが含まれる場合は、レポートが自動で立ち上がります。
 ### HDL コード生成チェック レポートを作成中 [HDL_Complex_Mult_3Mult5Add_report.html](#)
 ### 'complex_multiply_example' の HDL チェックが完了しました。エラーは 0 件、警告は 0 件、メッセージは 1 件です。
 ### HDL コードの生成が完了しました。

3.4.2 レポート生成

生成された HDL に関する詳細を確認するために、様々なレポートを生成することができます。デフォルト値から変更されているブロックレベル・モデルレベルの HDL プロパティ設定については、その概要がレポートに含まれます。

- トレーサビリティレポートを使って、Simulink ブロックから生成された RTL の対応箇所へのリンクと、生成された RTL から該当する Simulink ブロックへのリンクを付けることができます。



- リソース利用レポートを使うことで、乗算器やレジスタ、RAM などのリソースの使用率の概算値を確認することができます。生成されるレポートにはターゲットデバイスの情報は反映されませんが(例: 18x30bit の乗算器が 10 個、というように、Xilinx デバイスの DSP48 マクロの個数では表示されません)、論理合成を実行せずに概算値を確認することができますので、デザインを変更した場合のリソースへの影響を確認したい場合などに特に有効です。

complex_multiply_example の汎用リソース レポート

概要

| | |
|-------------|-----|
| 乗算器 | 3 |
| 加算器/減算器 | 5 |
| レジスタ | 19 |
| 1ビットレジスタの合計 | 461 |
| RAM | 0 |
| マルチプレクサー | 0 |
| I/O ビット | 152 |
| 静的シフト演算子 | 0 |
| 動的シフト演算子 | 0 |

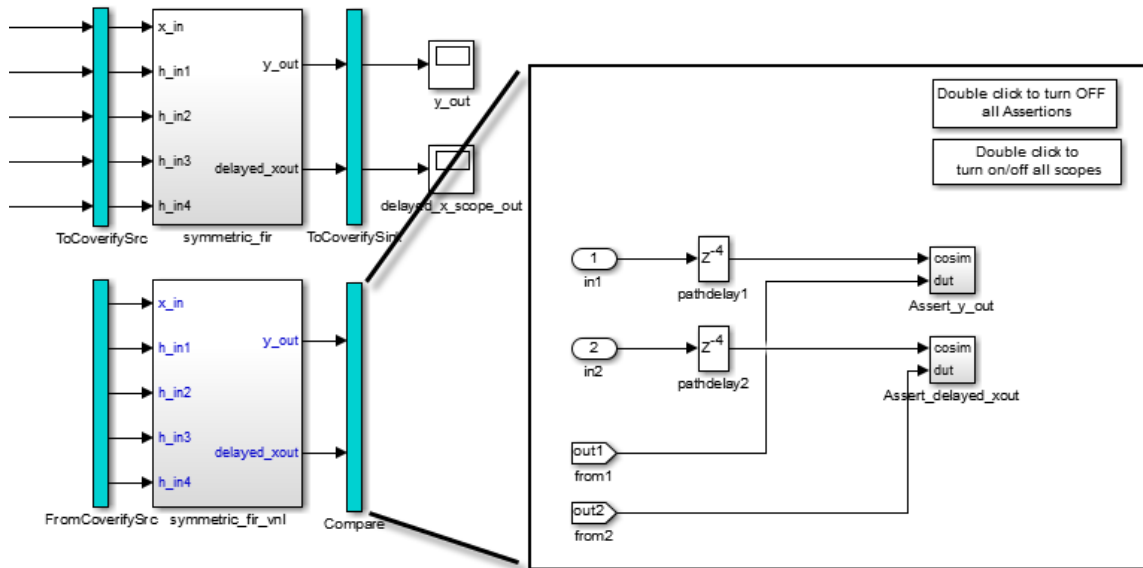
- 最適化レポートを使うことで、リソース共有や分散型パイプライン、遅延の均衡化や適応パイプラインなどの最適化設定の結果を確認できます。
- クリティカルパス推定結果レポートでは、論理合成を行わずに、ターゲットデバイスの情報を基にして推定されたクリティカルパスの情報を確認できます。

3.4.3 生成モデルと検証モデル

生成モデルと検証モデルは、最適化設定によって Simulink モデルと生成された HDL 間に生じた差分を解析するのに有効です。また、これらのモデルを使って、生成された HDL が機能的にはオリジナルの Simulink モデルと等価であることを確認することができます。

- 生成モデル
 - 生成モデルは Simulink モデルから HDL コード生成を行うと自動的に生成され、メモリ上にロードされます。生成モデルは HDL アーキテクチャや最適化設定によって生じたレイテンシの差分や数値精度が反映されており、生成された HDL とビット精度、サイクル精度が同じビヘイビアモデルです。
 - HDL テストベンチを生成する時には、DUT の入出力の値をログに保存する為に生成モデルのシミュレーションが実行されます。これらの値は HDL シミュレーション結果と比較する為に使われます。
 - 生成モデルは元モデルの頭に gm_ を付けた名前になります。例えば、sfir_fixed の生成モデルは gm_sfir_fixed となります。
- 検証モデル
 - 生成モデルは生成された HDL と等価なので、元モデルと比較したり、レイテンシや数値制度の差分を解析したりすることができます。[検証モデルを生成] を有効にすると、これらの比較を行うためのモデルが自動的に生成されます。

- 検証モデルは、元のモデルと生成モデルを同一エディタ上で接続したもので、双方の出力を比較できます。レイテンシの差分は比較前に補正され、数値的な差分があった場合にはアサーションが発火(成立)します。比較ロジックは Compare サブシステムとして以下の例のように実装されます。



- 検証モデルへのリンクはコード生成のステータスを表すメッセージの一部としてコマンドウィンドウ上に表示されます。また、最適化レポートの生成を行っている場合は、レポートの中にもリンクが置かれます。
 ### 'sla_fir_filter/FIR_primitive' の HDL を生成しています。
 ### HDL チェックを開始しています。
 ### 新しい検証モデルを生成しています: [gm_sla_fir_filter_vnl](#).
 ### 検証モデルの生成が完了しました。

3.5 生成されたコードの検証

回路の規模や速度など、目標値を達成するためには結果の解析と最適化設定の調整を繰り返すことになります。それらの作業の結果、ターゲットデバイスに対して満足する結果が得られたら、次は機能検証に進みます。機能検証は主に 4 つの手法があります。

- HDL 生成ワークフローの一部としてテストベンチを生成します。前述のとおり、テストベンチ生成中に生成モデルのシミュレーションが実行され、DUT の入出力の値がログとして保存されます。HDL シミュレータでテストベンチを実行すると、HDL の出力は予め保存された値と比較され、Simulink 上の DUT 出力と HDL 出力が一致したかどうかを示すメッセージ(PASS もしくは FAILED)が表示されます。大規模なテストベンチでは、Simulink Coder 製品と組み合わせて使用し、SystemVerilog DPI テストベンチを生成することを検討してください。
- コシミュレーション: コシミュレーションモデルは生成モデルの DUT 出力と生成された HDL を HDL シミュレータ上で実行した出力を自動的に比較してくれます。生成モデル出力と HDL 出力に不一致が生じた場合、アサーションが発火(成立)します。コシミュレーションモデルは HDL Coder の UI の[テストベンチ]もしくは HDL ワークフローアドバイザ[3.2 RTL コードとテストベンチを生成]から生成することができます。
 HDL Coder から生成したコードのシミュレーションに加えて、コシミュレーションウィザード(cosimWizard)を利用することで手書き HDL を Simulink モデルに組み込むことができます。ただし、コシミュレーションの実行には HDL Verifier とサポートされている HDL シミュレータが必要になります。
- FPGA-in-the-Loop(FIL): HDL Verifier は FPGA ハードウェアとの協調シミュレーションの機能も提供します。FIL モデルは生成モデル出力と FPGA ボード上で動作している論理合成・配置配線後の出力を自動的に比較してくれます。コシミュレーションと同様、生成モデルと FPGA 出力に不一致が生じた場合はアサーションが発火(成立)しま

す。FIL では FPGA が Simulink に同期して動作していることに注意してください。つまり、Simulink モデルがタイムステップを進めるたびにクロック周期のイネーブルで FPGA 上の処理が実行されます。

FIL モデルを生成するためには、HDL ワークフローアドバイザーの[1.1 ターゲットデバイスおよび合成ツールを設定]、[ターゲットワークフロー]を[FPGA-in-the-Loop]に設定してください。

- System Verilog DPI コンポーネント生成: HDL Verifier は、Simulink サブシステムの振る舞いを System Verilog シミュレータ上で再現可能なモジュールとして生成する機能を提供しています。R2016a から HDL コード生成対象サブシステムに対するテストベンチの自動生成機能が利用できるようになりました。HDL コシミュレーションと比べて高速な RTL 検証が可能です。ただし、本機能の利用には HDL Verifier 以外に Simulink Coder とサポートされている HDL シミュレータが必要になります。

4 FPGA ハードウェアへの実装

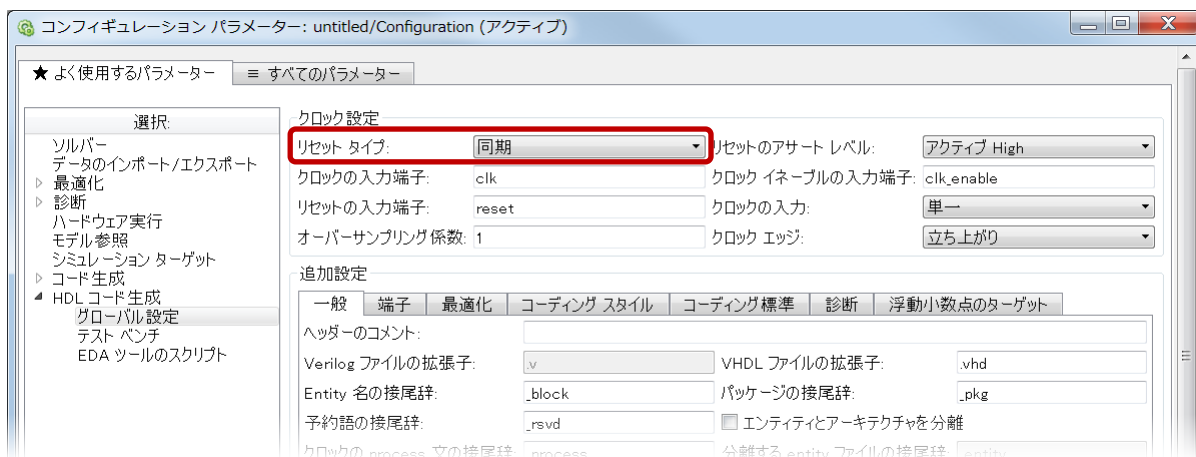
後工程での実装でターゲットとなる性能を達成する上で、デザインがどのようにハードウェア、特にターゲットデバイスにマッピングされたかを理解することが重要です。この章は FPGA ターゲットデバイス一般に適用可能な Tips で構成されています。

4.1 基本的な設定

4.1.1 グローバルリセットタイプ

合成ツールが確実に同期もしくは非同期リセット論理を実装できるようにすることと、リセットタイプを FPGA のアーキテクチャに合わせることで良好なリソース配分とパフォーマンスが得られます。

- Xilinx FPGA デバイス向けには同期グローバルリセットが推奨です。
- Intel (旧 Altera) FPGA デバイス向けには非同期グローバルリセットが推奨です。
- リセットタイプ設定は HDL Coder の UI の[グローバル設定]もしくは HDL ワークフローアドバイザーの[3.1.2 詳細オプションを設定]にあります。
- グローバルリセットが必要ない場合は、[グローバル設定]の[追加設定]の下にある[端子]で[グローバルリセットの最小化]を有効にします。



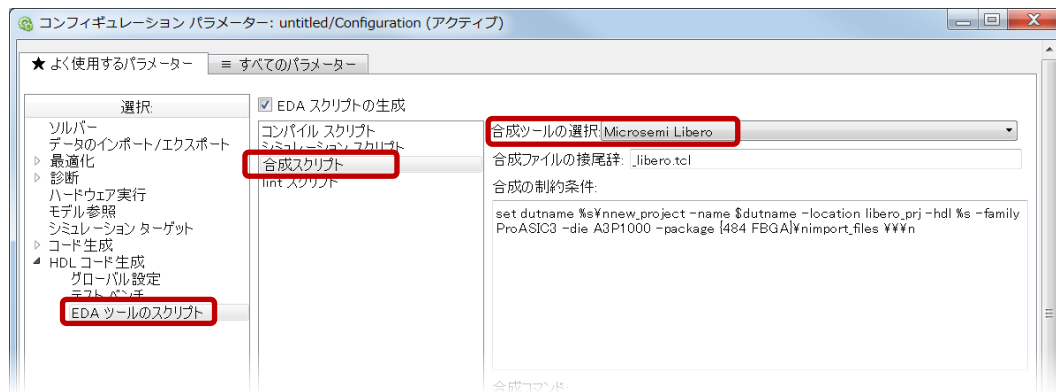
4.1.2 合成ツール連携

Xilinx ISE, Vivado, Intel Quartus Prime を使う場合、FPGA プロジェクトの生成、デザインの合成、タイミング解析を HDL ワークフローアドバイザーから合成ツールを立ち上げることなく実行できます。

タイミング制約や合成時の設定をカスタマイズしたい場合には[4.1 FPGA インザループのオプションの設定]で制約もしくは Tcl ファイルの指定を行なってください。[1.2 ターゲット周波数の設定]に正の値を指定すると、クロック制約が自動的に生成されて、FPGA プロジェクトに適用されます。

注意: 合成ツールとターゲット周波数が指定されている場合、適応パイプライニングによって自動的にレイテンシが挿入される可能性があります。詳細は 3.3.4 モデルレベル最適化 をご覧ください。

Microsemi Libero のような他の合成ツールを使用する場合には合成用の Tcl スクリプトを生成することも可能です。生成された Tcl スクリプトは FPGA プロジェクトを生成する際に使用されます。Tcl スクリプトの生成とコンテンツのカスタマイズを有効にするには、HDL Coder の UI から[EDA ツールのスクリプト]、[合成スクリプト]で任意の合成ツールを選択してください。



4.1.3 タイミング解析のための DUT の入出力端子のレジスタ設定

合成後のタイミング解析を行なう前に DUT 入出力端子にレジスタ(Delay ブロック)を置きます。タイミング解析はレジスタ間の回路パスに対して行われるため、この設定により確実にデザインの全要素がタイミング解析の対象となります。この結果、より精度の高いクリティカルパス推定が可能となります。

4.2 ブロック RAM マッピング

面積の増大のよくある要因の一つとして巨大なレジスタバンクができあがってしまう場合があります。これらは FPGA デバイス上のブロック RAM リソースに割り当ててことでより効率的に実装が可能となります。

4.2.1 RAM

Simulink ライブラリ ブラウザーの[HDL Coder]、[HDL RAMs]には次のような RAM ブロックがあります。

- Single Port RAM (System)
- Simple Dual Port RAM (System)
- Dual Port RAM (System)
- Dual Rate Dual Port RAM

hdl.RAM System Object で構成された RAM ブロックが R2017b のライブラリに追加されました。これらの System Object ベースのブロックは、ベクター入力、ゼロ以外の初期値をサポートし、大きな RAM サイズではシミュレーション速度が向上します。

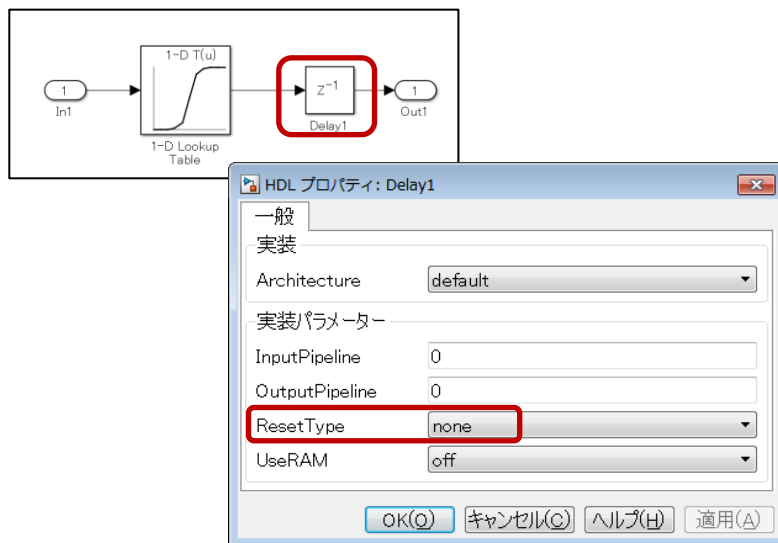
全ての RAM ブロックから生成された HDL は RAM 自身のインスタンスではなく RAM が推論されるような記述になります。そのためこれらの HDL はベンダーの特定のライブラリに依存することなく、FPGA ブロック RAM にマッピングすることが可能です。

Intel FPGA デバイス向けとしては[RAM アーキテクチャ]を[クロック イネーブルを使用しない汎用 RAM]とすることで良好なマッピング結果を得ることができます。このパラメータは HDL Coder の UI の[グローバル設定]、[コーディング スタイル]、もしくは HDL ワークフローアドバイザーの[3.1.2 詳細オプションを設定]にあります。

4.2.2 ROM

ブロック RAM にマッピングされるような ROM を作るには、Direct Lookup Table (n-D)ブロックとその後段に Delay ブロックを置いてください。Delay ブロックを右クリックして、[HDL コード]、[HDL ブロック プロパティ]を選択し、[ResetType]を[none]に設定してください。Delay ブロックへのリセット端子を無くすことで異なる合成ツールや FPGA デバイスでも最適なマッピング結

果を得ることが出来ます。HDL Coder は適応パイプラインングによって遅延挿入を自動化することができます。詳細は 3.3.4 モデルレベル最適化 をご覧ください。



加えて、Direct Lookup Table (n-D)ブロックについては下記のモデリングガイドラインにしたがってください。

- n ビットアドレスの場合、次の例のようにテーブルデータとして 2^n の全てのエントリーを指定してください。

```
x = (0:99)';
pad = 2^nextpow2(length(x)) - length(x);
x_pad = [x; zeros(pad,1)];
```

- [範囲外入力診断]が[エラー]になっていることを確認してください。
- ルックアップテーブルブロックは入力として uint8, uint16 などの組み込みデータタイプを指定する必要があります。ただし、余分なビットは HDL コード生成の際に最適化で除去されます。

4.2.3 補足事項

RAM/ROM のサイズが小さい場合は、合成ツールによってパフォーマンスの良いブロック RAM ではなく、分散 RAM がマッピングされる場合があります。どちらに合成されるかはツールに依存し、オプション設定で優先度を定めることが出来ます。

ブロック RAM を使用する場合 (FFT ブロックのように内部に RAM を含む場合も)、コシミュレーションと FIL において、元モデルとシミュレーション結果が異なる可能性があります。この原因は、RAM の内容が Simulink では実行ごとにリセットされるのに対し、HDL シミュレータと FPGA ではリセットされないことによるものです。

例えば Intel MAX10 デバイスではブロック RAM に推論させるために追加設定が必要です。Single Port RAM ブロックを使用する場合は、[Output data during write]を[Old data]に設定して下さい。Lookup Table ブロックから生成したコードを ROM にするには、Quartus Prime の[Assignments]メニュー、[Device]、[Device and Pin Options]、[Configuration]、[Configuration mode]を[Single (Un)compressed Image with Memory Initialization (512Kbits UFM)]に設定します。

4.3 DSP マッピング

FPGA 中の DSP ブロックは、乗算器だけでなく、加算器、パイプラインレジスタなどのリソースも含んでいます。以下の項目は、DSP ブロックを有効活用するのに役立ちます。

4.3.1 一般的なルール

- リセットタイプ
 - 4.1.1 で推奨されているグローバルリセットタイプを使用するか、乗算/加算器の前後に置く Delay ブロックの HDL ブロックプロパティの[ResetType]を[none]に設定します。間違ったリセットタイプを使用すると、DSP ブロックのリソースが正しくマッピングされないことがあります。
- 乗算器のビット幅
 - 18x18 bit の乗算器であれば、おおよそ正しく DSP ブロックにマッピングされますが、ターゲットデバイスの DSP ブロックのリソースを最大限に活用するためには、デバイスのアーキテクチャに合わせたモデリングを行います。
 - Xilinx Virtex 6, 7 の乗算器は 18x25 bit, UltraScale アーキテクチャの場合、乗算器は 18x27 bit
 - Intel FPGA DSP ブロックは 1 つの大きな乗算器や複数の小さな乗算器としてマッピングすることができます。例えば Arria 10 の DSP ブロックは 1 つの 27x27 bit 乗算器、2 つの 18x19 bit 乗算器、3 つの 9x9 bit 乗算器として利用することができます。実際のビット幅はデバイスファミリーと動作モードに依存します。
 - 乗算処理のビット幅がターゲットデバイスの DSP ブロックのビット幅を超えると、合成ツールは別のロジックを使用するか、複数の DSP ブロックに分割してマッピングします。これは速度低下と面積増大の原因となります。
- 固定小数点設定

乗算/加算器では、フル精度の固定小数点を使用し、出力にパイプラインレジスタを追加します。更に、最後の出力レジスタの後に、Data Type Conversion ブロックを使用して、必要に応じて丸めと飽和処理を入れます。詳細説明は R2016b から、HDL Coder は単精度浮動小数点演算の合成可能なコードを生成できるようになりました。これは、高ダイナミックレンジの計算や初期のプロトタイプ HDL の生成に役立ちます。この機能の詳細については、ビデオページにある「FPGA/ASIC のネイティブ浮動小数点実装」を参照してください。また、「2.2.6 ネイティブ浮動小数点コード生成を使う際のベストプラクティス」で正しい方を紹介します。

- 2.2.1 固定小数点設定を参照してください。
- もしくは、Product/Add ブロックの HDL プロパティで丸めや飽和処理を選択し、適応パイプラインングによって乗算器と出力ロジックの間に自動的にパイプラインレジスタを挿入させる方法もあります。詳細は 3.3.4 モデルレベル最適化を参照してください。
- パイプラインレジスタ
 - DSP ブロック内のパイプラインレジスタを使用することで、レイテンシは増えますがより高速なクロック周波数を達成することが出来ます。デザインで要求される速度に応じてパイプラインレベルを設定して下さい。

4.3.2 乗算

- 単一の乗算処理を行う場合のパイプライン設定
 - Xilinx ISE/Vivado では、DSP48 ブロックに対して入力/出力ともに 2 つまでのレジスタをマッピングすることが出来ます。
 - Intel Quartus Prime では、入力は 1~2、出力は 1 つのレジスタをマッピングできますが、デバイスファミリーに依存します。

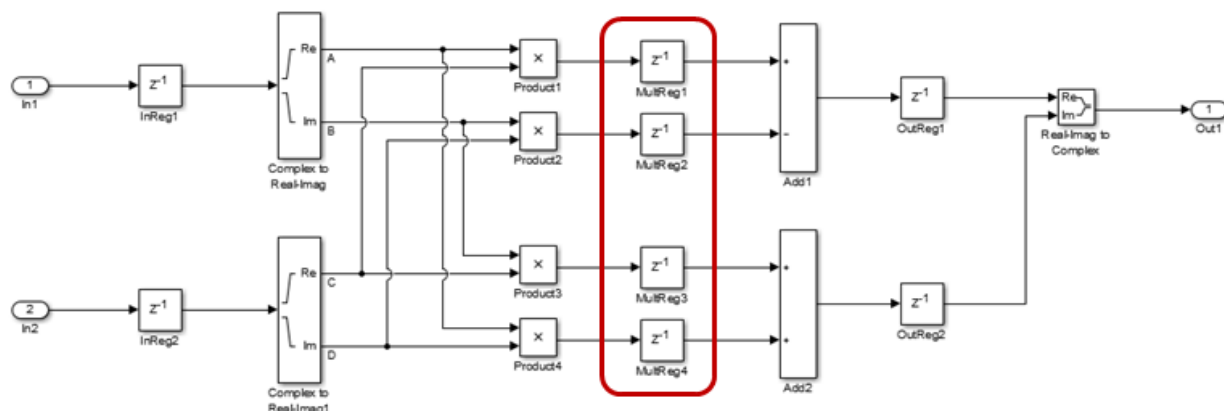
4.3.3 複素乗算

例: [complex_multiply_example.slx](#)

複素乗算は下式のとおりに、乗算と加減算の組み合わせで構成されます。それらは 4 つの乗算器と 2 つの加算器で実装することが可能です。

$$(A + Bj) * (C + Dj) = (A * C - B * D) + (A * D + B * C)j$$

下図に示す通り、加算器のビット幅の増加に合わせて前段の乗算器出力のパイプライン化が必要です。Product ブロックと分散パイプラインを使用して、下図と同等の HDL コードを生成することが可能です。



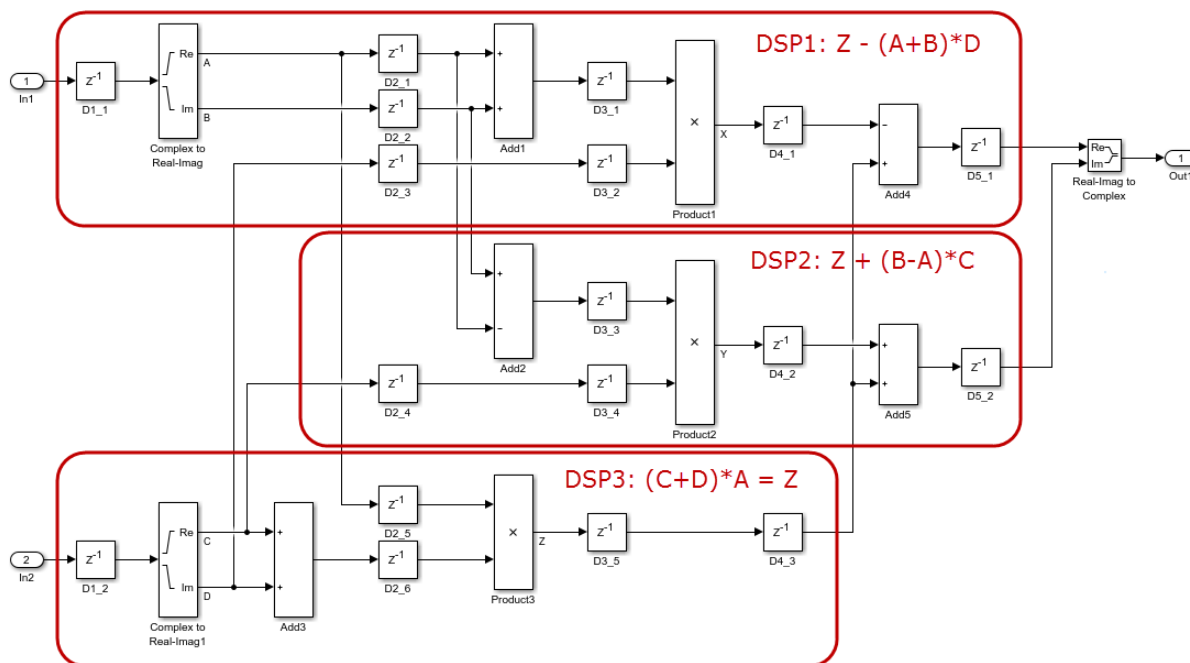
パイプラインレジスタ

- Xilinx ISE/Vivado では、DSP48 ブロックに対して入力 2 つまで、乗算器出力は 1 つ、加算器出力は 1 つのレジスタをマッピングすることができます。両方の出力レジスタを使用するとより良い速度パフォーマンスが得られます。

Intel Quartus Prime では、DSP ブロックに対して入力 1 つまたは 2 つまで、加算器出力は 1 つのレジスタをマッピングすることができます。殆どのデバイスファミリーには乗算器と加算器の間にパイプラインレジスタがありませんので、Quartus Prime は DSP ブロック内に加算器をマッピングするために、加算器出力にそのレジスタを移動させます。

複素乗算器は 3 つの乗算器と 5 つの加算器を使っても実装することもできます。

$$(A + Bj)(C + Dj) = (Z - X) + (Y + Z)j, \text{ where } X = (A + B)*D, Y = (B - A)*C, Z = (C + D)*A$$



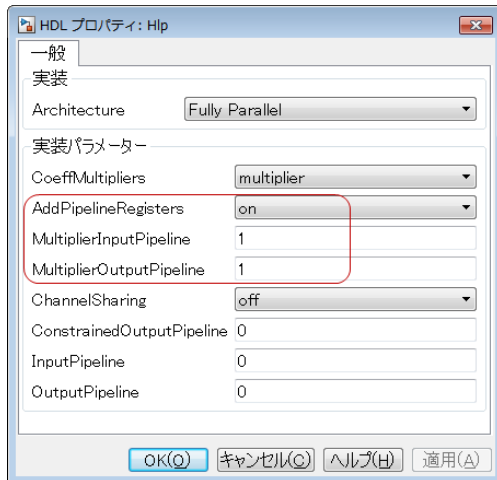
- Xilinx FPGA ターゲットのアーキテクチャでは前述の“4 乗算器, 2 加算器”のアプローチと比べて“3 乗算器, 5 加算器”のアプローチの方が DSP48 ブロックの Pre-Adder が使われるので DSP48 ブロックを 1 個節約できます。その代わり、レジスタリソースを多く消費し、レイテンシが増える可能性があります。最大クロック周波数も低下する可能性があります。
- Intel FPGA ターゲットのアーキテクチャでは“3 乗算器, 5 加算器”のアプローチの方が、特定のワードサイズとデバイスファミリーの組み合わせによっては DSP リソースが節約できる場合もありますが、多くの場合、前述の“4 乗算器, 2 加算器”のアプローチの方が優れています。最適な複素乗算のアーキテクチャについてはデバイスファミリーのドキュメントの DSP の項目を参照してください。
- パイプライン化の方針:
 - Xilinx ISE/Vivado は DSP48 ブロックに 2 つの入力レジスタ、1 つのプリ加算器出力レジスタ、1 つの乗算器出力レジスタ、1 つの加算器出力レジスタを最大割り当てることができます。入力レジスタ (D1 = 2) をさらに増やすことで DSP48 をパイプライン化し、タイミングを改善することもできますが、周辺のレジスタリソースをさらに消費することにつながります。
 - Intel Quartus Prime は 1 つから 2 つの入力レジスタ(D1)、1 つの加算器出力レジスタ (D5) を DSP ブロックに割り当てることができます。

4.3.4 FIR フィルター

例: `fir_filter_example.slx`

Discrete FIR Filter、FIR Decimation、および FIR Interpolation ブロックには、以下のガイドラインが適用されます。アーキテクチャとパイプラインのオプションは、生成されたコードにのみ影響することに注意してください。

- Discrete FIR Filter ブロックの HDL ブロックプロパティの[Architecture]を[Fully Parallel], [Partly Serial], [Fully Serial]に設定すると DSP ブロックに正しくマッピングされます。それに加えてブロックパラメータの[フィルター構造]を[直接型対称]に設定すると最近の FPGA の DSP ブロックでサポートしている Pre-Adder がマッピングされます。
- Discrete FIR Filter のデフォルト設定では[フィルター構造] は [直接型]になります。これはクリティカルパスが長くなり、速度パフォーマンスに悪い結果をもたらします。非対称フィルターに対して速度パフォーマンスを向上させるには、[直接型転置]構造を使用し、次の設定を参考にしてパイプラインレジスタを使用して下さい。
- [Architecture]を [Partly Serial], [Fully Serial] に設定したときの乗算器の共有値=クロック分周比は、乗算器数の公約数の範囲でしか設定できません。(例: 乗算器数が 12 個の時、共有値は 1, 2, 3, 4, 6, 12)クロックの制約からこれ以外の値に設定したい場合は、上位のサブシステムの HDL ブロックプロパティの[SharingFactor]を設定して下さい。これに対しては任意の正の整数値を設定することが可能です。
- Fully Parallel の FIR フィルターのパイプライン設定
「複素乗算の 4 乗算器、2 加算器」のアプローチを参考に、下図に示す HDL ブロックプロパティの [AddPipelineRegisters], [MultiplierInputPipeline], [MultiplierOutputPipeline]を設定してパイプラインレジスタを挿入して下さい。



- フィルター構造が[直接型]で[AddPipelineRegisters]を[on]に設定すると、加算器出力にレジスタが追加されるだけでなく、加算器チェーンがパイプライン化された Tree 構造になります。
- 直接型対称で Fully Parallel の FIR フィルターのパイプライン設定
 - Xilinx ISE/Vivado では、DSP48 ブロックに対して入力は 1 つ、Pre-Adder の出力は 1 つ、乗算器の出力は 1 つ、加算器出力は 1 つ のレジスタをマッピングすることができます。乗算器出力レジスタを 2 つ使用するとより良い速度パフォーマンスが得られ、必要に応じて余分なレジスタは移動されます。
 - Intel Quartus Prime では、デバイスファミリーによりませんが、入力は 1 つ、加算器出力は 1 つのレジスタを DSP ブロックにマッピングすることが出来ます。乗算器と最終段の加算器の間にはパイプラインレジスタはありませんので、そこにレジスタを入れようとすると Quartus Prime は DSP ブロックの出力に移動します。Pre-Adder と乗算器の間にレジスタがあると、Pre-Adder が DSP ブロックにマッピングされません。
 - 直接型対称の FIR フィルターでは、HDL ブロックプロパティの[MultiplierInputPipeline]を設定すると、Pre-Adder の出力と乗算器の入力の間にレジスタが追加されます。入力レジスタを追加したい場合は、[InputPipeline]を使用するか、ブロックの前に Delay ブロックを置いて下さい。
- Serial の FIR フィルターのパイプライン設定
 - Xilinx ISE/Vivado では、入力は 1 つ、乗算器出力は 1 つ、加算器出力は 1 つのレジスタを DSP48 ブロックに割り当てることができます。
 - Intel Quartus Prime では、デバイスファミリーによりませんが、DSP ブロックに対して入力が 1 つまたは 2 つとアキュムレータ出力のレジスタをマッピングすることができます。ほとんどのデバイスファミリーでは、乗算器と加算器の間にはパイプラインレジスタはありませんので、そこにレジスタを入れようとすると加算器が DSP ブロック外のリソースで合成される可能性があります。
 - 入力レジスタと乗算器の出力レジスタを入れたい場合は、必要に応じて[MultiplierInputPipeline], [MultiplierOutputPipeline]を設定します。アキュムレータの出力レジスタはこのアーキテクチャではデフォルトで組み込まれているため設定する必要はありません。
- フレームベース(ベクター)入力の並列処理 FIR フィルターのパイプライン設定
 - [AdderTreePipeline]パラメータは加算ツリーの各レベルに対してのパイプラインレジスタの数を指定することができます。その際、乗算器出力レジスタの最適な数は(MultiplierOutputPipeline + AdderTreePipeline)で決まります。
 - Xilinx FPGA ターゲットの場合、MultiplierOutputPipeline = 0 と AdderTreePipeline = 1 で、初段の加算器を DSP48 ブロックにマッピングすることができ、結果として面積を最小化することができます。一方、AdderTreePipeline = 2 に設定すると、面積を増加させる代わりにより高いパフォーマンスが得られます。
 - Intel FPGA ターゲットの場合、MultiplierOutputPipeline = 1、AdderTreePipeline を 1 か 2 に設定してください。

R2017a 以降では、Data Valid などの制御信号を持つ Discrete FIR Filter HDL Optimized ブロックが追加され、レイテンシとサイクル精度のシミュレーションが可能になりました。パイプライン処理はブロックパラメータのフィルター係数と[Share DSP Resources]、さらにはモデルのターゲットデバイス設定に基づいて自動的に決定されます。最良の結果を得るには、以下のガイドラインに従ってください。

- [HDL コード生成]、[ターゲットおよび最適化]の下[ツールおよびデバイス]設定パラメータ、または HDL ワークフローアドバイザーのステップ[1.1 ターゲットデバイスおよび合成ツールを設定]でターゲットデバイスを設定します。HDL Coder は、ターゲットデバイス設定 (Xilinx/Altera) に応じて、ブロックの内部パイプライン配置を決定します。配置場所についての詳しい情報は、ドキュメントの DSP System Toolbox/ブロック/Discrete FIR Filter HDL Optimized
>> web(fullfile(docroot, 'dsp/ref/discretefirfilterhdloptimized.html'))
をご覧ください。
- セクション 4.3.1「一般的なルール」のリセットタイプと乗数のワードサイズに関するガイドラインに従ってください。ブロック内のパイプラインは、1 つの DSP ブロックで各乗算/加算・アキュムレータを実装するために最適化されているため、語長が長いほど性能が最適ではない可能性があります。

4.4 サブシステムへの分散型パイプラインレジスタの挿入

HDL ブロックプロパティの設定により分散型パイプラインレジスタを挿入することで、クリティカルパスを低減することができます。この利用方法は[製品ドキュメンテーション](#)やモデリングのノウハウについて纏めた HDL Modeling Guideline の 4 章 速度・面積の最適化をご覧ください。

生成される HDL の記述や論理合成結果が意図した通りにならない場合は MathWorks Japan (support@mathworks.co.jp) までお問い合わせ下さい。