

Vivado Design Suite

ユーザー ガイド

高位合成

UG902 (v2012.2) 2012 年 7 月 25 日



Notice of Disclaimer

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

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

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

資料によっては英語版の更新に対応していないものがあります。

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

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

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	改訂内容
2012年7月25日	1.0	初版

目次

改訂履歴	2
第 1 章：高位合成の概要	
高位合成の概要	11
第 2 章：高位合成ユーザー ガイド	
はじめに	12
関数抽象レベル	12
高位合成 (HLS)	13
高位合成の概要	18
高位合成の概要	18
高位合成の使用	20
チュートリアル	46
C の検証とコーディング スタイル	47
合成前の検証	47
サポートされていない C 言語コンストラクト	49
任意精度データ型	51
浮動小数点型	56
マルチアクセス ポインター インターフェイス	57
インターフェイスの管理	62
インターフェイス合成	62
バスインターフェイスの指定	79
SystemC のインターフェイス合成	96
手動のインターフェイス仕様	97
デザイン最適化	99
チェックリストおよびガイドライン	99
クロック、タイミング、および RTL 出力	102
関数の最適化	107
関数の再利用、インライン化、およびインスタンシエーション	108
関数のパイプライン処理	112
レイテンシ制約	115
関数のインターフェイスプロトコル	115
ループの最適化	115
ループの展開	117
ループの結合	120
ネストされたループのフラット化	122
ループのデータフロー パイプライン処理	122
ループのパイプライン処理	124
ループ キャリー依存性	126
ループ 反復の制御	127
ループのレイテンシ	128
配列の最適化	128
配列の初期化とリセット	130
メモリリソースの選択	131
配列のマップ	132
配列の分割	137

配列の変更	138
配列のストリーミング	139
ロジック構造の最適化	140
演算子選択	140
ハードウェアリソースの制御	141
構造体パッキング	141
演算調整	142
エラボレーションエフォート	143
検証	144
RTLの自動検証	145
RTLデザインのエクスポート	150
バスインターフェイス	150
RTL合成	150
パッケージの識別	151
Tclコマンド	152
IP-XACTフォーマットでのエクスポート	152
IP-XACTパッケージのVivadoへのインポート	152
Pcoreフォーマットでのエクスポート	153
PcoreパッケージのEDK環境へのインポート	154
System Generatorへのエクスポート	154

第3章：高位合成演算子およびコアガイド

合成の概要	157
スケジューリングとは	157
バインディングとは	158
演算子、コア、指示子について	159
演算子およびコアの制御	160
演算子の制限	160
リソースの制御	160
スケジュールの制御	162
バインディングの制御	162
高位合成の演算子	163
高位合成のコア	164
ファンクションユニットコア	165
ストレージコア	165
コネクター コア	166
アダプター コア	166
浮動小数点コア	167

第4章：高位合成コーディングスタイルガイド

はじめに	169
表記規則	169
はじめに	169
コード例	170
Cの合成	170
最上位デザイン	170
テストベンチ	172
デザインファイルとテストベンチファイル	174
最上位の引数: RTLインターフェイスポート	177
データ型	192
Cライブラリ	202
hls_mathライブラリ	203
HLSビデオライブラリ	208
ハードウェアのコード記述方法	214
C++でのユーザー定義レジスター	214
SRLリソースへの直接マップ	215

ストリーミング データを使用した設計	216
関数	226
ループ	227
配列	234
サポートされない C コンストラクト	237
C++ の合成	240
C++ クラス	241
テンプレート	246
データ型	247
サポートされない C++ コンストラクト	252
SystemC の合成	253
デザインの記述	253
最上位 SystemC ポート	261
サポートされない SystemC コンストラクト	265
C の任意精度型	267
[u]int#W 型のコンパイル	267
[u]int#W 変数の宣言/定義	267
定数(リテラル)からの初期化および代入	268
コンソール I/O(出力)のサポート	269
[u]int#W 型を使用した式	270
ビット レベル演算: サポートされる関数	274
C++ の任意精度型	277
ap_[u]<> 型のコンパイル	277
ap_[u] 変数の宣言/定義	277
定数(リテラル)からの初期化および代入	278
コンソール I/O(出力)のサポート	279
ap_[u]<> 型を使用した式	279
クラス演算子およびメソッド	280
その他のクラス メソッドおよび演算子	285
C++ の任意精度(AP) 固定小数点型	289
ap_[u]fixed 表現	289
量子化モード	290
オーバーフロー モード	291
ap_[u]fixed<> 型のコンパイル	293
ap_[u]fixed<> 変数の宣言/定義	293
定数(リテラル)からの初期化および代入	293
コンソール I/O(出力)のサポート	294
ap_[u]fixed<> 型を使用した式	294
クラス演算子およびメソッド	295

第5章：高位合成コマンド リファレンス ガイド

高位合成コマンドの使用	305
プロジェクトの管理	305
高位合成の最適化のロケーション	306
コマンドおよびプラグマ	308
高位合成コマンド	309
add_file	309
構文	309
説明	309
オプション	309
Pragma	310
例	310
autoimpl	310
構文	310
説明	310
オプション	311
pragma	311

例	311
cosim_design	312
構文	312
説明	312
オプション	312
pragma	313
例	313
autosyn	313
構文	313
説明	313
pragma	313
例	313
close_project	314
構文	314
説明	314
pragma	314
例	314
close_solution	314
構文	314
説明	314
pragma	314
例	314
config_array_partition	315
構文	315
説明	315
オプション	315
pragma	315
例	315
config_bind	316
構文	316
説明	316
オプション	316
pragma	316
例	317
config_dataflow	317
構文	317
説明	317
オプション	317
pragma	317
例	317
config_interface	318
構文	318
説明	318
オプション	319
pragma	319
例	319
config_rtl	320
構文	320
説明	320
オプション	320
pragma	320
例	321
config_schedule	321
構文	321
説明	321
オプション	321
pragma	321

例	321
create_clock	321
構文	321
説明	322
オプション	322
pragma	322
例	322
delete_project	322
構文	322
説明	322
pragma	323
例	323
delete_solution	323
構文	323
説明	323
pragma	323
例	323
elaborate	324
構文	324
説明	324
オプション	324
pragma	324
例	324
help	324
構文	324
説明	324
オプション	325
pragma	325
例	325
list_core	325
構文	325
説明	325
オプション	326
pragma	326
例	326
list_part	327
構文	327
説明	327
pragma	327
例	327
open_project	327
構文	327
説明	327
オプション	328
pragma	328
例	328
open_solution	328
構文	328
説明	328
オプション	329
pragma	329
例	329
set_clock_uncertainty	329
構文	329
説明	329
pragma	329
例	330

set_directive_allocation	330
構文	330
説明	330
オプション	331
pragma	331
例	331
set_directive_array_map	331
構文	331
説明	332
オプション	332
pragma	332
例	332
set_directive_array_partition	333
構文	333
説明	333
オプション	333
pragma	333
例	334
set_directive_array_reshape	334
構文	334
説明	334
オプション	334
pragma	335
例	335
set_directive_array_stream	335
構文	335
オプション	336
pragma	336
例	336
set_directive_clock	337
構文	337
説明	337
pragma	337
例	337
set_directive_dataflow	338
構文	338
説明	338
オプション	338
pragma	338
例	338
set_directive_data_pack	339
構文	339
説明	339
オプション	339
pragma	339
例	339
set_directive_dependence	340
構文	340
説明	340
オプション	340
pragma	341
例	341
set_directive_expression_balance	341
構文	341
説明	341
オプション	342
pragma	342

例	342
set_directive_function_instantiate	342
構文	342
説明	342
pragma	343
例	343
set_directive_inline	343
構文	343
説明	343
オプション	343
pragma	344
例	344
set_directive_interface	344
構文	344
説明	344
オプション	345
pragma	346
例	346
set_directive_latency	347
構文	347
説明	347
オプション	347
pragma	347
例	348
set_directive_loop_flatten	348
構文	348
説明	348
オプション	348
pragma	348
例	349
set_directive_loop_merge	349
構文	349
説明	349
オプション	349
pragma	349
例	350
set_directive_loop_tripcount	350
構文	350
説明	350
オプション	350
pragma	350
例	351
set_directive_loop_unroll	351
構文	351
説明	351
オプション	351
pragma	352
例	352
set_directive_occurrence	352
構文	352
説明	353
オプション	353
pragma	353
例	353
set_directive_pipeline	353
構文	353
説明	353

オプション	354
pragma	354
例	354
set_directive_protocol	355
構文	355
説明	355
オプション	355
pragma	355
例	355
set_directive_resource	356
構文	356
説明	356
オプション	356
pragma	356
例	356
set_directive_top	357
構文	357
説明	357
オプション	357
pragma	357
例	357
set_directive_unroll	358
構文	358
説明	358
オプション	358
pragma	359
例	359
set_part	359
構文	359
説明	359
pragma	360
例	360
set_top	360
構文	360
説明	360
pragma	360
例	360

付録 A: その他のリソース

ザイリンクス リソース	361
ソリューション センター	361
リファレンス	361

高位合成の概要

高位合成の概要

本書では、高位合成 (HLS) に関するさまざまなコンセプトと、高位合成およびザイリンクスの高位合成ツールの基本的な概要について説明します。

- 高位合成は、C、C++、または SystemC デザイン仕様をレジスタ トランസファー レベル (RTL) インプリメンテーションに変換し、それをザイリンクス フィールド プログラマブル ゲート アレイ (FPGA) に合成
- ザイリンクス FPGA デバイスのインプリメンテーション用に C コード (C++ および SystemC も含む) を記述する方法を示したコード スタイル
- 高位合成のリファレンス情報

高位合成ユーザー ガイド

はじめに

本章では、高位合成 (HLS) に関するさまざまなコンセプトと、高位合成およびザイリンクスの Vivado HLS ツールの基本的な概要について説明します。

関数抽象レベル

定義

FPGA 設計業界は、デザインの複雑性を管理するため、抽象レベルをいくつか移動させました。新しい抽象レベルに移動するたびに、デザインインプリメンテーション手順の複雑さが隠され、低い抽象レベルに関連する課題が見えにくくなる代わりに、生産性が向上します。

- トランジスタ レイアウト データベースでは、マスク製造とウエア工程の複雑さが隠されます。レイアウト抽象化レイヤーでは、基本的なレイアウトをモデリングするデザインルールチェック (DRC) に従うことに焦点が置かれます。
- FPGA デザインでは、ネットリストでは詳細なレイアウトの実行を避け、あらかじめ構築されたライブラリから生成されます。ネットリスト抽象化レイヤーでは、適切なエリア、パフォーマンス、および消費電力のデザインのプール機能を定義します。
- レジスタトランスマート (RTL) 記述は、レジスタの境界間のデータバスおよびロジックを定義することにより、必要な機能をキャプチャします。RTL 合成により、デザインをインプリメントするプール機能のネットリストが作成されます。RTL 抽象化レイヤーでは、機能的に正しいハードウェアのモデルを定義します。
- 機能仕様により、必要なアルゴリズムをインプリメントするのにレジスタ境界とその間に必要なロジックを定義する必要がなくなります。設計者は機能を定義するだけです。

上記の抽象レベルの移動と同様に、Vivado HLS による機能仕様を使用して RTL デザインを自動的に作成することにより、検証およびデザイン最適化において生産性が向上します。

- C 言語を基にした機能仕様を使用することによりシミュレーション時間を大幅に短縮することができ、デザインエラーを早期に発見することができます。
- 高位合成により手動で TRL を作成するプロセスを短縮でき、機能仕様から RTL を自動的に作成することにより変換エラーを回避できます。
- 高位合成では RTL アーキテクチャの最適化が自動的に実行され、どれか 1 つに絞る前に複数のアーキテクチャをすばやく簡単に評価できます。

C ベースの仕様

機能仕様を作成するには、C ベース入力が最もよく使用されます。現在のところ、FPGA にインプリメントするシステムの機能を定義するには ANSI-C (C99)、C++、および SystemC が標準的に使用されています。

Vivado HLS では、C、C++、および SystemC がサポートされています。SystemC は、ハードウェアのモデリングおよび同時シミュレーションに使用される IEEE 規格 (IEEE-1666) です。エラボレーション時にボンディングされていない合成不可能なコンストラクトや、有限のサイズの記述を判断できないコンストラクトは、合成できません。

ネイティブ C データ型は、通常の 8 ビット、16 ビット、32 ビット、および 64 ビット ワードです (char、short、int、long、long long)。ANSI-C および C++ には、ビット精度計算に対応したビットインのデータ型はありません。ビット精度計算では、データ型と完全に一致するビット幅が使用され、最適なサイズのハードウェアが得られます。Vivado HLS では、C および C++ の両方で任意精度のデータ型がサポートされます。また、SystemC の任意精度のデータ型もサポートされます。

高位合成 (HLS)

C の RTL への合成では、デザインエリアとパフォーマンスのさまざまな面を考慮した高度な変換が多数実行されます。Vivado HLS では、C、C++、および SystemC の 3 つの C 入力すべてがサポートされ、C コードを最小限の変更で合成できます。

Vivado HLS は、デザインに対して主に 2 種類の合成を実行します。

- アルゴリズム合成 : 関数の内容を読み込み、機能文のあるクロック サイクル数での RTL 文に合成します。
- インターフェイス合成 : 関数引数 (またはパラメーター) を特定のタイミングプロトコルを使用して RTL ポートに変換し、デザインがシステム上のほかのデザインと通信できるようにします。
 - インターフェイス合成は、グローバル変数、最上位関数引数、および最上位関数の戻り値に対して実行できます。
 - 使用可能なインターフェイスのタイプは、次のとおりです。
 - ワイヤ
 - レジスタ
 - 一方向および双方向ハンドシェイク
 - バス
 - FIFO
 - RAM
 - また、関数レベル プロトコルを最上位関数に合成できます。関数レベル プロトコルには、関数がいつ演算を開始するかを制御する信号、終了したことを示す信号が含まれます。

Vivado HLS 合成は、複数の段階で実行されます。インターフェイス合成の結果とアルゴリズム合成の結果は、お互いに影響を与えます。手動の RTL デザインと同様に、可能なインプリメンテーションおよび最適化は数多くあり、その組み合わせも非常に多数です。Vivado HLS を使用すると、ユーザーがこのような詳細を考慮する必要はなく、短時間で最適なデザインを得ることができます。

Vivado HLS によりこのような詳細がどのように処理されるかを理解するため、このセクションの残りの部分で HLS の基本的な概念と Vivado HLS で実行される最適化について説明します。

- 制御およびデータパスの抽出
- スケジューリングおよびバインディング
- 任意精度データ型
- 最適化
- デザイン制約

制御およびデータパスの抽出

Vivado HLS でまず最初に実行されるのは、コードにより推論される制御およびデータパスの抽出です。図 2-1 に、これがどのように実行されるかの例を示します。

制御機能は、コードではループおよび条件分岐で示されます。図 2-1 は、コードから制御動作をどのように抽出できるかを示しています。関数がループに入りするのは、RTL 有限ステート マシン (FSM) 2 のいずれかのステートに出入りするのに対応します。

図 2-1 では、すべての演算が 1 サイクル (ステート) で実行されると想定しています。実際には、タイミング遅延およびクロック周波数により演算を完了するのにこれ以上のサイクルが必要な場合もあります。たとえば、ステート 1 がステート 11、12、および 13 に拡張され、制御ロジックはインターフェイス合成で推論される I/O プロトコルに影響され、Vivado HLS によりより複雑な最適化されたステート マシンが生成される場合があります。

データパスの抽出はもう少し簡単で、ループを展開し、デザインの条件文を評価することにより決定できます。

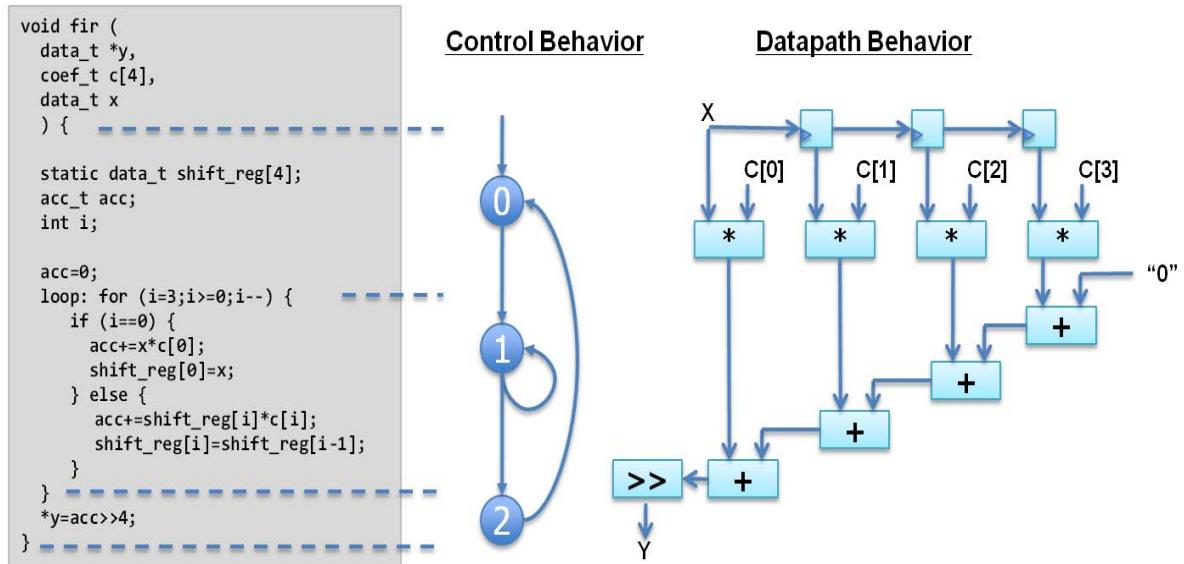


図 2-1: 制御およびデータの抽出

RTL の最終的なデータパス インプリメンテーションは、通常図 2-1 に示すような単純なものではありません。Vivado HLS では、最後のシフト操作は 2 のべき乗であり、ハードウェアを必要としないので、最初の加算器は不要であると簡単に判断できます。デザインがスケジューリングされる場合、より複雑な最適化および判断が必要になります。

スケジューリングおよびバインディング

スケジューリングおよびバインディングは、高位合成の中心となるプロセスです。Vivado HLS は、スケジューリングプロセス中に、どのサイクルで演算を実行するかを決定します。スケジューリング中の決定には、クロック周波数、クロックのばらつき、デバイス テクノロジ ライブリから のタイミング情報、エリア、レイテンシ、スループットなどが考慮されます。

図 2-1 に示すコード例の場合、複数の RTL インプリメンテーションが可能です。図 2-2 に、そのうちの 3 つを示します。

1. Vivado HLS では複数のクロック サイクルで加算器と乗算器を共有できるので、4 クロック サイクルを使用するということは、加算器を 1 つと乗算器 1 つを使用できるということです。加算器 1 つ、乗算器 1 つを使用して、4 クロック サイクルで完了します。
2. ターゲット テクノロジのタイミングの解析により加算器のチェーンを 1 クロックで完了できると判断された場合、加算器 3 つと乗算器 4 つが使用されますが、演算を 1 クロック サイクルで完了できます。つまり、オプション 1 よりも高速ですが、大型になります。
3. 加算器 2 つと乗算器 2 つを使用して、2 クロック サイクルで完了します。つまり、オプション 2 よりも小型で、オプション 1 より高速です。

Vivado HLS では、ツールのデフォルトとユーザーが指定した制約や指示子に基づいて、最適なインプリメンテーションを短時間で作成できます。この後の章で、特定の要件において最も理想的なソリューションを短時間で達成できるよう制約および指示子を設定する方法を示します。

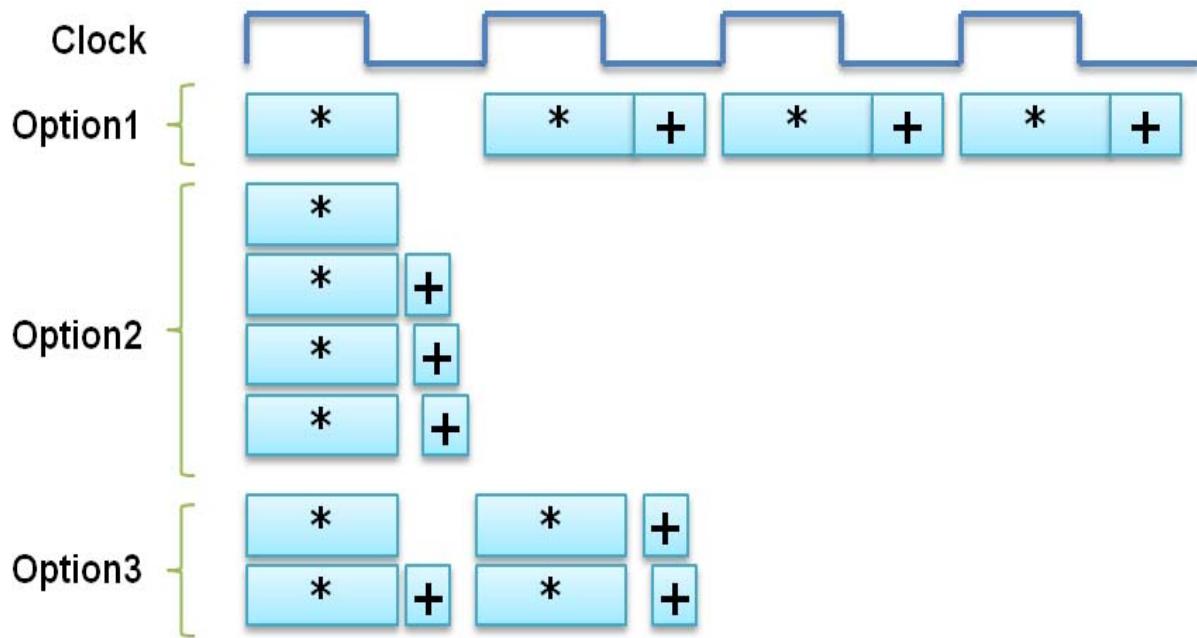


図 2-2: スケジューリング

バインディングは、スケジューリングされた各演算に使用するハードウェアリソースまたはコアを決定するプロセスです。たとえば、加算器と減算器を 1 つずつ使用するか、加減算器 1 つを両方の演算に使用するかを Vivado HLS で自動的に判断できます。

通常の組み合わせ乗算器の代わりにパイプライン乗算期を使用するなどの、バインディングプロセスでの決定が演算のスケジューリングに影響するので、バインディングに関する決定はスケジューリング中に考慮されます。

任意精度データ型

ネイティブ C データ型は、8 ビット境界 (8、16、32、64 ビット) にあります。RTL 演算 (ハードウェアに対応) では、任意の幅がサポートされます。HLS には、任意精度のビット幅を使用できるようにしたり、RTL デザインで 17 ビット乗算器のみが必要なときに 32 ビット乗算器を使用できるようにする (C プログラムでは問題ないが RTL デザインでは重大な問題) などのメカニズムが必要です。

Vivado HLS では、任意精度の整数および固定小数点のデータ型を使用できます (表 2-1)。

表 2-1: 整数データ型

言語	整数データ型	必要なヘッダー
C	[u]int<precision> (1024 ビット)	#include "ap_cint.h"
C++	ap_[u]int<W> (1024 ビット) ap_[u]fixed<W,I,Q,O,N>	#include "ap_int.h" #include "ap_fixed.h"
System C	sc_[u]int<W> (64 ビット) sc_[u]bigint<W> (512 ビット) sc_[u]fixed<W,I,Q,O,N>	#include "systemc.h" #include "sc_fixed.h"

これらの任意型は、ビットスライス、連結、範囲選択など、ハードウェアのような動作を提供する関数によりサポートされます。このユーザー ガイドの「任意精度データ型」を参照してください。

最適化

任意精度データ型 HLS では、デザインに多数の最適化を実行し、パフォーマンスおよびエリアの要件を満たす質の高い RTL を生成します。このセクションでは、最適化手法のいくつかを紹介し、その機能の概要を示します。

パイプライン処理は、ハードウェアの主なパフォーマンスの利点である同時実行(並列実行)を RTL デザインに自動的にインプリメントする最適化です。

C プログラムは、順次に実行されます。図 2-3 の左側の関数 top では、各サブ関数 func_A、func_B、および func_C は、func_A が 2 回目に実行される前に完了する必要があります。

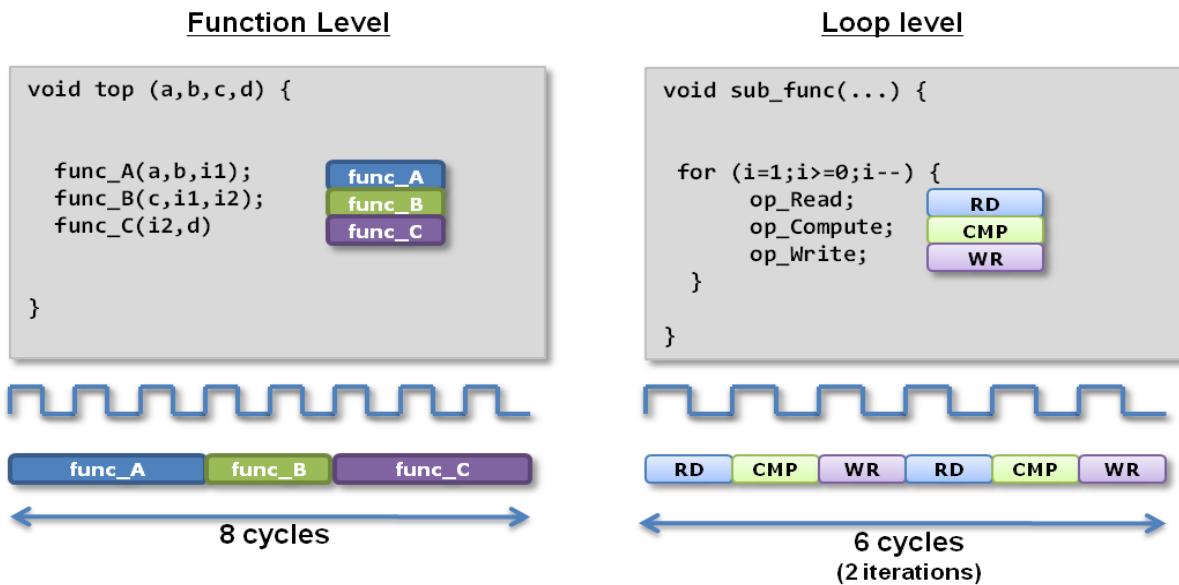


図 2-3: パイプライン処理を使用しない関数およびループ

func_A が完了すると func_A は次の演算を実行可能になりますが、func_A がもう 1 度実行される前に、関数 func_B と func_C が完了する必要があります。

図 2-3 の右側に示す関数 sub_func は、演算子レベルでは関数 top と同様で、最初の演算が 2 回目に実行される前に最後の演算が完了する必要があります。

C 言語は順次に実行され、同時実行は不可能なため、各演算は実行の順番を待たなければなりません。Vivado HLS では、関数およびループを自動的にパイプライン処理でき、RTL デザインが同様に制限されないようにします。

デフォルトでは、Vivado HLS でこれらの演算を並列実行する方法が検索され、デザイン全体のレイテンシが削減されます。Vivado HLS では、これに加え、これらの演算をパイプライン処理することによりスループットを向上でき、関数の異なる実行やループの異なる反復が時間的にオーバーラップすることができます。

図 2-4 に、Vivado HLS を使用してサブ関数およびループ内の演算をパイプライン処理した場合の結果を示します。

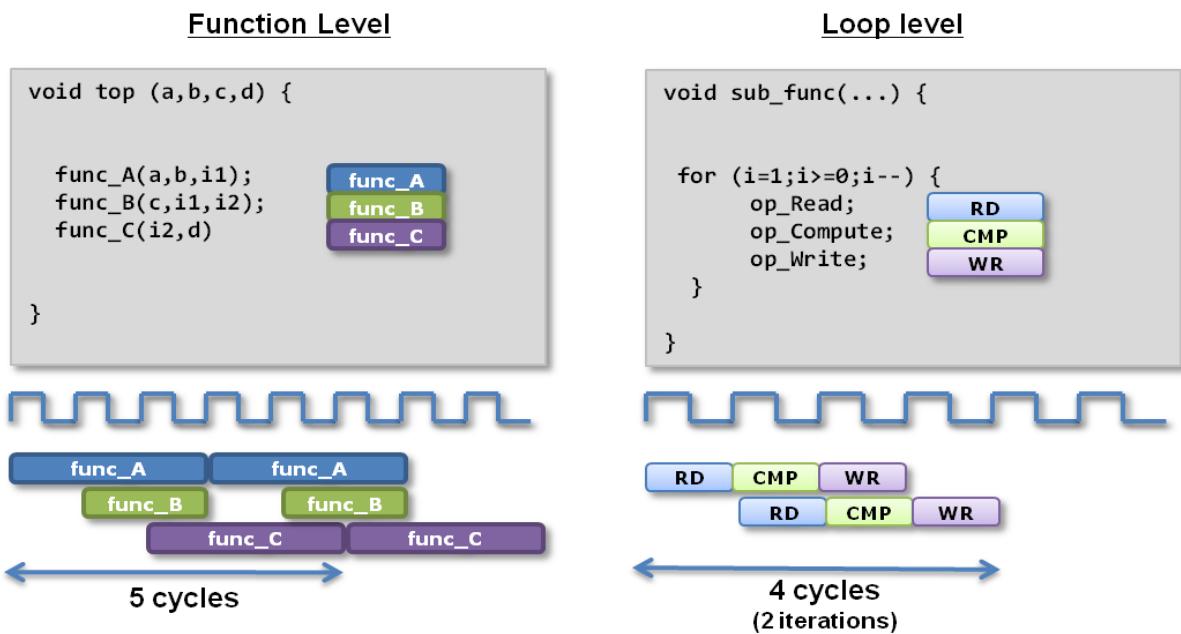


図 2-4: パイプライン処理を使用した関数およびループ

- 関数レベルでは、データフロー最適化によりデータが到着するとすぐにサブ関数 (`func_A`、`func_B`、および `func_C`) が実行されます。
 - 関数 `func_A` の 2 回目の実行は、`func_C` の 1 回目の実行が完了する前に開始します。
 - 図 2-3 に示したインプリメンテーションでは関数の実行に 8 クロック サイクルかかっていましたが、それが 5 サイクルで実行できるようになり、`func_A` は 8 クロック サイクルごとではなく 3 クロック サイクルごとに開始します。
- ループをパイプライン処理すると、ループ内の演算を同時に実行できます。
 - 図 2-4 に、ループのパイプライン処理により 図 2-3 と比較してパフォーマンスが向上することを示します。ループは 4 クロック サイクルで完了しており、新しい入力 (RD 演算) は 3 クロック サイクルごとではなく各クロック サイクルで処理されます。

Vivado HLS で自動的にインプリメントされるデサイン最適化のもう 1 つの例として、配列の分割があります。

C 言語の記述では、配列は類似した要素をグループ化する便利な方法として使用されます。配列要素が記憶要素として合成されると (すべてのクロック サイクルで値が保持される)、これらの配列要素は RTL で RAM 内にグループ化されるか、各構成要素に分割されて個別のレジスタとしてインプリメントされます。

- 配列要素が一度に 1 つずつアクセスされる場合、要素をグループ化して 1 つの RAM にマップするのがハードウェアでは効率的なインプリメンテーションです。
- 複数の配列要素が同時に必要な場合は、個別のレジスタにインプリメントしてデータに同時にアクセスできるようにした方が、パフォーマンスの面では有利かもしれません。

記憶要素の配列を個別のレジスタにインプリメントするとパフォーマンスが向上するかもしれません、RAM の利点は活用できません。RAM はエリアが効率的で、デバイス上に BRAM として LUT およびレジスタとは別に用意されています。

Vivado HLS には、配列を最適な方法でインプリメントするためのさまざまな機能が含まれています。

- 大型の配列を複数の小型の配列に分割し、それぞれを RAM のインスタンスにマップし、複数の読み出しありは書き込みを同じクロック サイクルで実行できるようにします。

- 複数の小型の配列を同じ RAM リソースにインプリメントできるようにします。

いくつかの簡単な指示子を適用することにより、パイプライン処理から配列の調整まで、多数のインプリメンテーションが提供され、デザインに最適なインプリメンテーションをすばやく簡単に見つけられるようにします。

デザイン制約

最後に、Vivado HLS では、クロック周期とクロックのばらつきに加え、次のような制約を多数使用できます。

- 関数、ループ、および領域のレイテンシを指定
- 使用されるリソースの数を制限
- コードに固有のまたはコードで暗示される依存性を変更した演算を可能にする（メモリに書き込む前に読み出すなど）

これらの制約を Vivado HLS 指示子を使用して適用し、必要な特性を持つデザインを作成できます。

Vivado HLS を使用した設計は HLS フローであり、コード内の依存性および Vivado HLS のデフォルトの C 言語コンストラクトの解析を使用して初期アーキテクチャをすばやくインプリメントし、その後指示子を使用して目標の高パフォーマンス インプリメンテーションに近づけていくことができます。

高位合成の概要

高位合成の概要

図 2-5 に示すように、Vivado HLS には C 言語ベースのデザイン記述を、グラフィカル ユーザー インターフェイス (GUI) または Tcl バッチ スクリプトを使用して指定した指示子および制約を入力します。サポートされるすべてのザイリンクス デバイスのタイミングおよびエリアの詳細が指定されたテクノロジ ライブラリはあらかじめ組み込まれており、入力する必要はありません。

出力は Verilog、VHDL、および SystemC 形式の RTL デザイン ファイルです。RTL 検証および RTL 合成を自動化する検証スクリプトおよびインプリメンテーション スクリプトも作成されます。

このセクションでは、これらの入力および出力の概要を説明します。

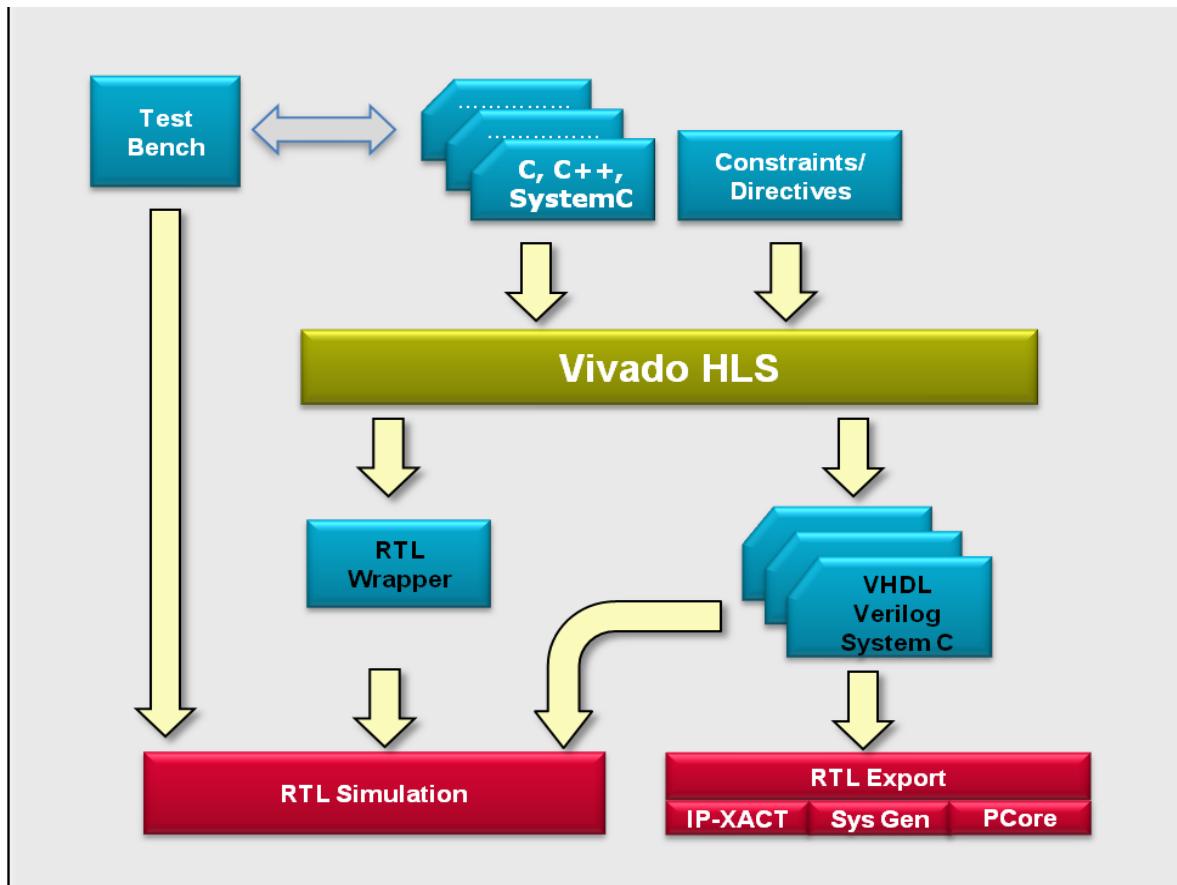


図 2-5 : 高位合成の使用モデル

デザイン ファイル

C または C ベース デザインと言った場合、Vivado HLS では次の 3 つの規格すべてを意味します。

- ANSI-C (任意の整数精度に対応するデータ型で拡張されたもの)
- C++ (任意の整数精度および固定小数点精度に対応するクラスで拡張されたもの)
- SystemC (IEEE-1666)

このガイドでは、入力仕様のシミュレーション方法を、任意精度の拡張も含めて説明します。

C ベース入力には、テストベンチを含めることができます。C テストベンチを入力した場合、出力 RTL の検証にも使用でき、RTL 検証用に RTL テストベンチを別に作成する必要はありません。Vivado HLS では複数の入力ファイルがサポートされているので、テストベンチを合成するデザインとは別のファイルに分離することをお勧めします。ただし、これは必須ではありません。

デバイス テクノロジ ライブラリ

デバイス テクノロジ ライブラリは、サポートされる各ザイリンクス デバイスのエリアおよびタイミングをモデルリングしたもので、最適化エンジンによりトレードオフを考慮した適切な選択が実行されるようになっています。デバイス テクノロジ ライブラリは Vivado HLS に組み込まれており、供給する必要はありません。

指示子および制約

指示子および制約は Vivado HLS の GUI または Tcl ベースのコマンドで指定し、目標のパフォーマンスおよび RTL アーキテクチャを達成できるよう最適化エンジンが実行されるようにします。

RTL 出力

合成が正常に完了すると、自動的に RTL 出力が記述されます。Vivado HLS では、次の 3 つのハードウェア記述言語 (HDL) がサポートされています。

- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- SystemC (IEEE 1666-2006 -Version 2.2-)

注記 : Vivado HLS からの SystemC 出力は、レジスタ トランスファー レベル (RTL) でのデザイン インプリメンテーションです。

シミュレーション出力 (RTL 協調シミュレーション)

Vivado HLS では、RTL を検証するのに必要なスクリプトが作成されます。この検証は、元のテストベンチと RTL シミュレータを使用した協調シミュレーションにより実行します。次の RTL シミュレータがサポートされています。

- ModelSim
- VCS
- OSCI SystemC

SystemC 出力はビルドインの SystemC カーネルを使用して検証できるので、サードパーティの RTL シミュレータやライセンスは必要ありません。サポートされている HDL シミュレータを使用するには、そのシミュレータのベンダーからライセンスを取得する必要があります。

インプリメンテーション出力

デザインを RTL 合成し、FPGA 上に配置配線するために必要なスクリプトおよび制約ファイルが提供されます。これらのスクリプトにより、RTL 合成プロセスを Vivado HLS の GUI からボタンをクリックするだけで実行できます。

高位合成の使用

このセクションでは、Vivado HLS の起動、プロジェクトの作成、RTL インプリメンテーションの管理、最適化用の指示子の適用など、Vivado HLS の使用方法の概要を説明します。その後、詳細を例を使用しながら示します。

Vivado HLS は、グラフィカル ユーザー インターフェイス (GUI) または Tcl コマンドを対話型またはバッチ モードで使用可能なコマンド ライン インターフェイス (CLI) で起動できます。

高位合成の GUI

Windows

PC Windows プラットフォームで Vivado HLS を起動するには、デスクトップで [Vivado HLS] アイコン (図 2-6) をクリックします。



図 2-6 : Vivado HLS の GUI アイコン

Linux

Linux プラットフォームで Vivado HLS を起動するには、Linux コマンドプロンプトで次のコマンドを入力します。

```
$ vivado_hls
```

図 2-7 に示す Vivado HLS の GUI が開きます。

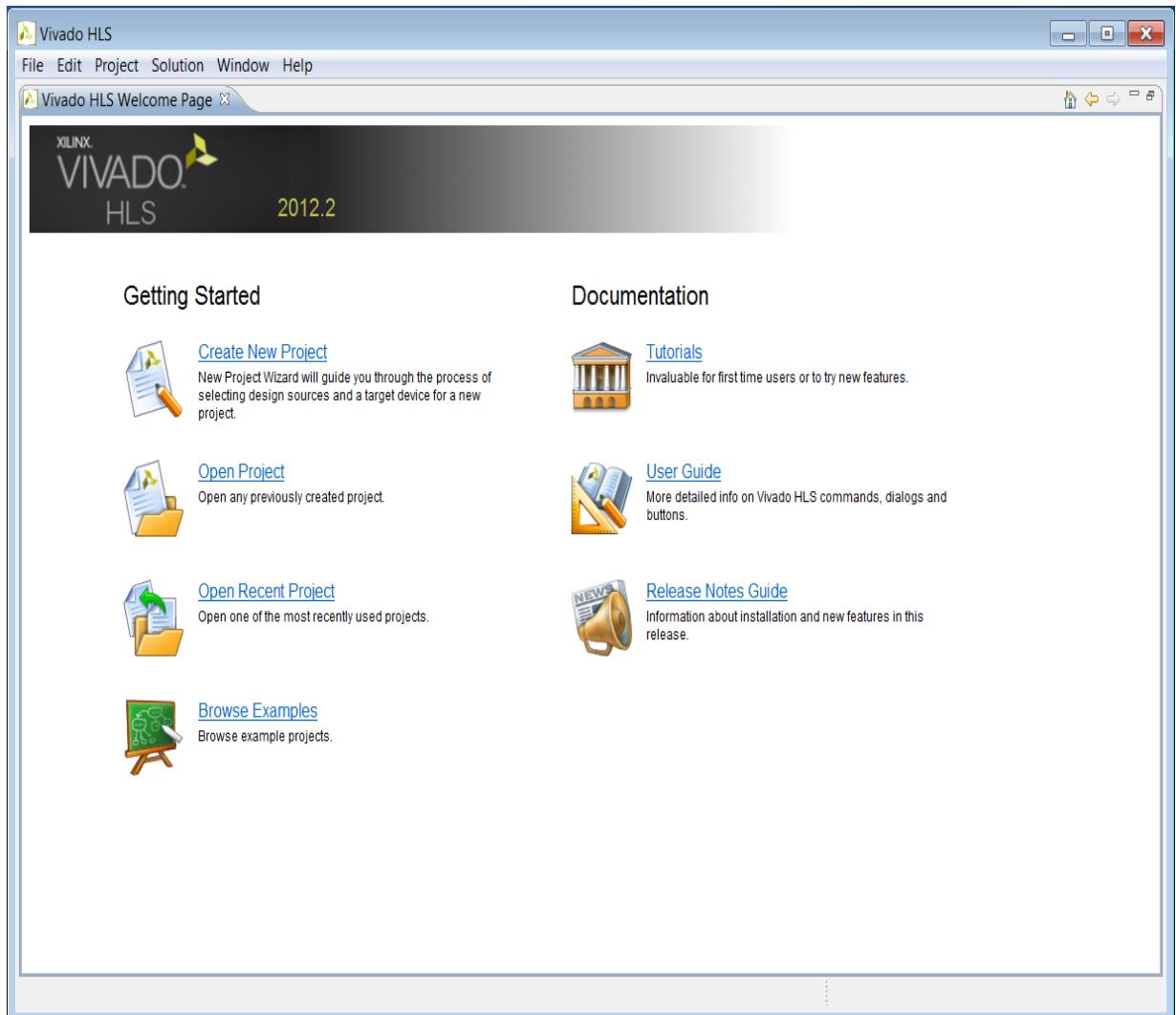


図 2-7 : GUI モード

図 2-7 の [Getting Started] の下には、次のタスクがリストされています。

- [Create New Project]
 - プロジェクト設定ウィザードが起動します。
- [Open project]
 - 既存のプロジェクトを開けます。
- [Open Recent Project]
 - 最近のプロジェクトをリストから選択します。

[Documentation] の下には、ウェルカム ページ (図 2-7) から参照可能な資料がリストされています。

- [Browse Examples]
 - 高位合成例を開けます。例は、Vivado HLS のインストールディレクトリ内の examples ディレクトリから開くこともできます。
- [Release Note Guide]
 - このツールバージョンのリリース ノートを開けます。
- [User Guide]
 - 『Vivado Design Suite ユーザー ガイド』を集めたウェブサイトを開けます。
- [Tutorials]
 - 『Vivado Design Suite チュートリアル』を集めたウェブサイトを開けます。

高位合成のコマンド ライン インターフェイス

Windows の場合、高位合成のコマンド ライン インターフェイス (CLI) は、[スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [Vivado 2012.2] → [Vivado HLS Command Prompt] をクリックすると開けます。



図 2-8 : Vivado HLS CLI アイコン

Windows および Linux の両方で、vivado_hls コマンドを -i オプションを使用して実行すると、Vivado HLS が対話型モードで開けます。Vivado HLS コマンド プロンプトが表示され、Tcl コマンドを入力できるようになります。

```
$ vivado_hls -i [-l <log_file>]

vivado_hls>
```

デフォルトでは、現在のディレクトリに vivado_hls.log ファイルが作成されます。別のファイルを指定するには、-l <log_file> オプションを使用します。

Vivado HLS ではオートコンプリージョンがサポートされており、コマンドの最初の数文字を入力して Tab キーを押すと、その文字で開始するコマンドがリストされます。

```
vivado_hls> open<TAB>
open
open_project
```

```
open_solution
```

Vivado HLS コマンドにはビルトインのヘルプがあり、`help` コマンドを使用して表示できます。コマンド名の入力には、オートコンプリエーション機能を使用できます。

```
vivado_hls> help <command>
```

`exit` コマンドを入力して対話型モードを終了し、シェルプロンプトに戻ります。

```
vivado_hls> exit
$
```

コマンドを Tcl スクリプトに記述し、`-f <script_file>` オプションを使用してバッチモードで実行することもできます。

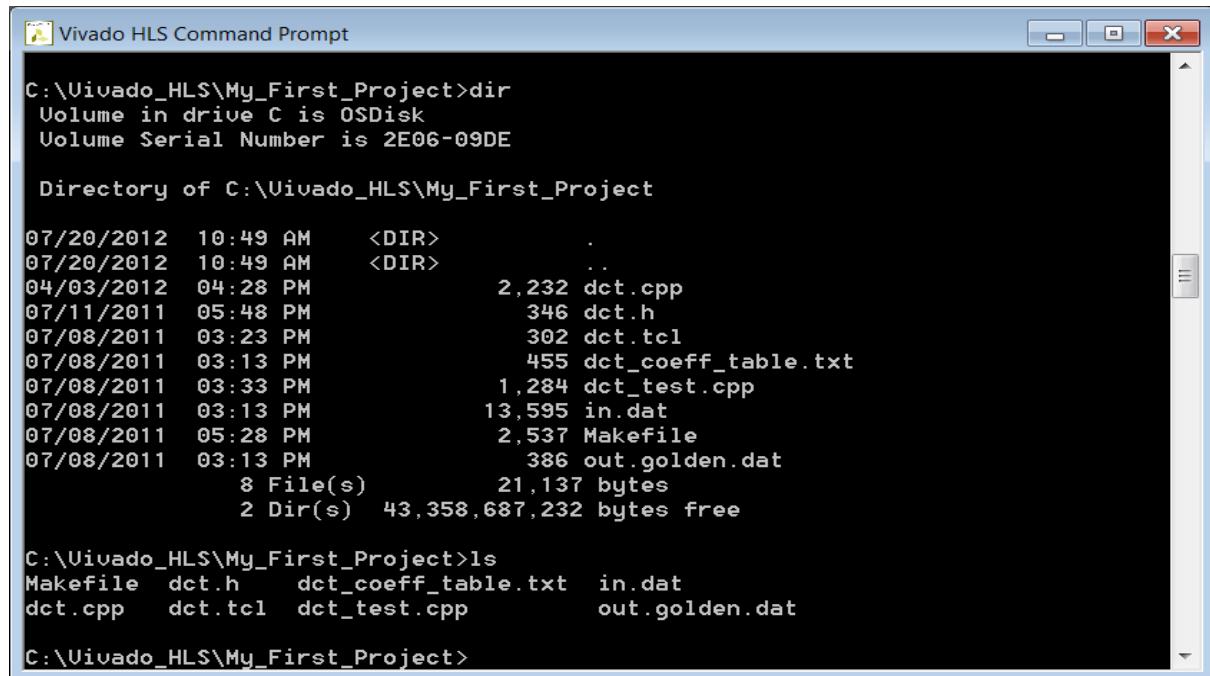
```
$ vivado_hls -f script.tcl
```

スクリプトによる自動化を支援するため、Vivado HLS には実行されている環境の詳細を返すオプションがあります。`-version` オプションでは Vivado HLS のバージョン番号、`-system` オプションでは Vivado HLS が実行されている OS、`-machine` オプションでは現在のマシンアーキテクチャ、`-root_dir` オプションでは Vivado HLS がインストールされているディレクトリ名が返されます。

Windows での CLI シェルの使用

Windows OS では、CLI シェルは Minimalist GNU for Windows (minGW) 環境を使用してインプリメントされています。この環境では、標準 Windows DOS コマンドと Linux コマンドのサブセットの両方を使用できます。

図 2-9 では、Linux の `ls` コマンドと DOS の `dir` コマンドのどちらでもディレクトリの内容をリストできることを示しています。



```
Vivado HLS Command Prompt

C:\Uivado_HLS\My_First_Project>dir
Volume in drive C is OSDisk
Volume Serial Number is 2E06-09DE

Directory of C:\Uivado_HLS\My_First_Project

07/20/2012  10:49 AM    <DIR>    .
07/20/2012  10:49 AM    <DIR>    ..
04/03/2012  04:28 PM          2,232  dct.cpp
07/11/2011  05:48 PM          346  dct.h
07/08/2011  03:23 PM          302  dct.tcl
07/08/2011  03:13 PM          455  dct_coeff_table.txt
07/08/2011  03:33 PM          1,284  dct_test.cpp
07/08/2011  03:13 PM          13,595  in.dat
07/08/2011  05:28 PM          2,537  Makefile
07/08/2011  03:13 PM          386  out.golden.dat
                  8 File(s)    21,137 bytes
                  2 Dir(s)  43,358,687,232 bytes free

C:\Uivado_HLS\My_First_Project>ls
Makefile  dct.h    dct_coeff_table.txt  in.dat
dct.cpp   dct.tcl  dct_test.cpp       out.golden.dat

C:\Uivado_HLS\My_First_Project>
```

図 2-9: 高位合成 CLI アイコン

minGW 環境では、すべての Linux コマンドとその動作がサポートされているわけではないので注意してください。次に、サポートの違いのいくつかを示します。

- Linux の which コマンドはサポートされていません。
- makefile の Linux パスは、自動的に minGW パスに変換されます。すべての makefile に含まれる Linux 形式のパス名の代入 (FOO := :/ など) は、パスが置換されないように、パス名にクオーテーションが付いたもの (FOO := ":/") に変換されます。

高位合成プロジェクトの作成

Vivado HLS を使用するには、まず新規プロジェクトを作成するか、既存のプロジェクトを開きます。[図 2-7](#) に示す Vivado HLS の GUI のウェルカムページからこれらの操作を実行するには、[Create New Project] または [Open Project] をクリックします。

[Create New Project] をクリックすると、高位合成プロジェクト ウィザードが開きます。ウィザードの 1 ページ目では、プロジェクトの仕様を指定します([図 2-10](#))。

次を指定します。

- [Project name] : プロジェクト名を指定します。この名前は、プロジェクトが保存されるディレクトリ名としても使用されます。[図 2-10](#) の例のように .prj などの拡張子を指定しておくと、ディレクトリをプロジェクトのものと判断しやすくなりますが、これは必須ではありません。
- [Location] : プロジェクトの保存先を指定します。
- [Top Level] : 最上位モジュールが SystemC SC_MODULE である場合は、[SystemC] をオンにします。それ以外の場合は、[C/C++] をオンにします(デフォルト)。ソースファイルで SystemC タイプが使用されているが最上位が SC_MODULE でない場合は、[C/C++] をオンにします。

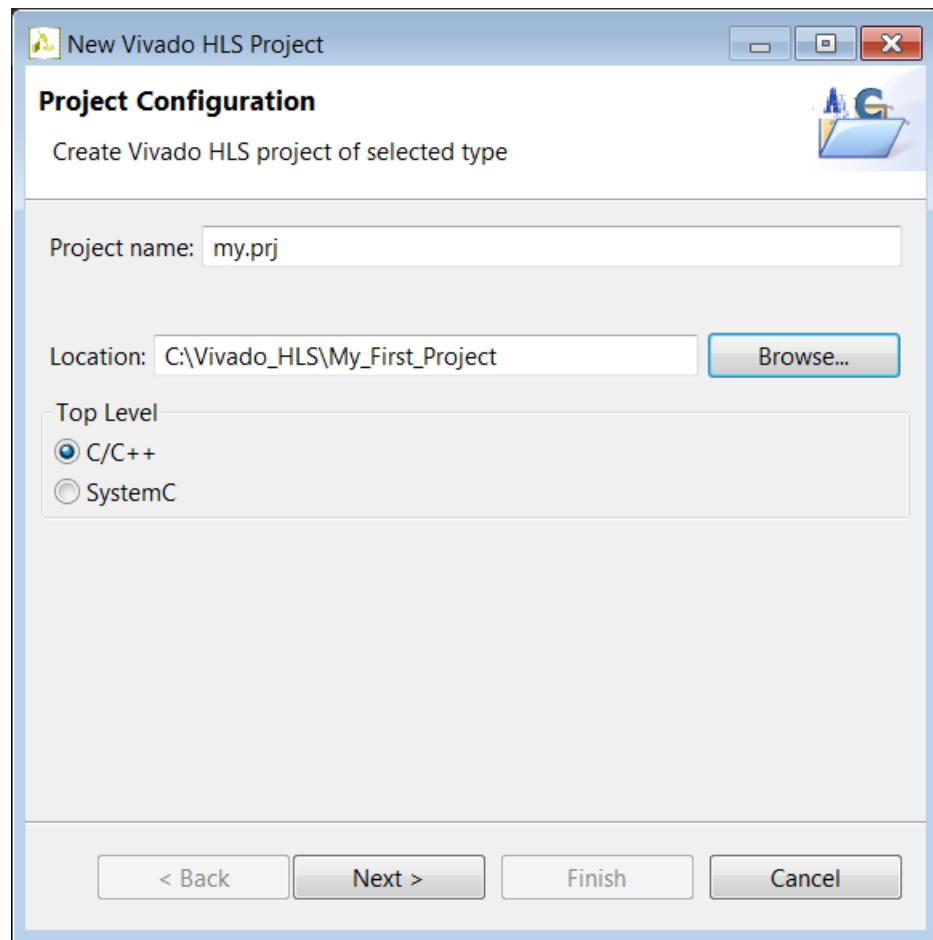


図 2-10: プロジェクト仕様

[Next] をクリックして次のページに進むと、プロジェクトの C ソースを追加するページ (図 2-11) が表示されます。

[Top Function] には合成する最上位関数の名前を指定します。

注記 : プロジェクトを SystemC として指定する場合はこれは不要です。

[Add Files] をクリックし、ソースコードファイルをプロジェクトに追加します。

[Edit CFLAGS] をクリックすると、ソースファイルを正しくコンパイルするために必要な C コンパイラ フラグをプロジェクトに追加できます。

C コンパイラ フラグには、-DMACRO_1 (コンパイル中に MACRO_1 を定義) などのマクロ仕様、ネストされた関数を含むデザインに必要な -fnested-functions、関連するヘッダー ファイルの検索パスを指定する -I/project/source/headers などがあります。ローカル ディレクトリ (図 2-10 の [Location] で指定したディレクトリ) にあるヘッダーは、自動的に検索されて含まれます。

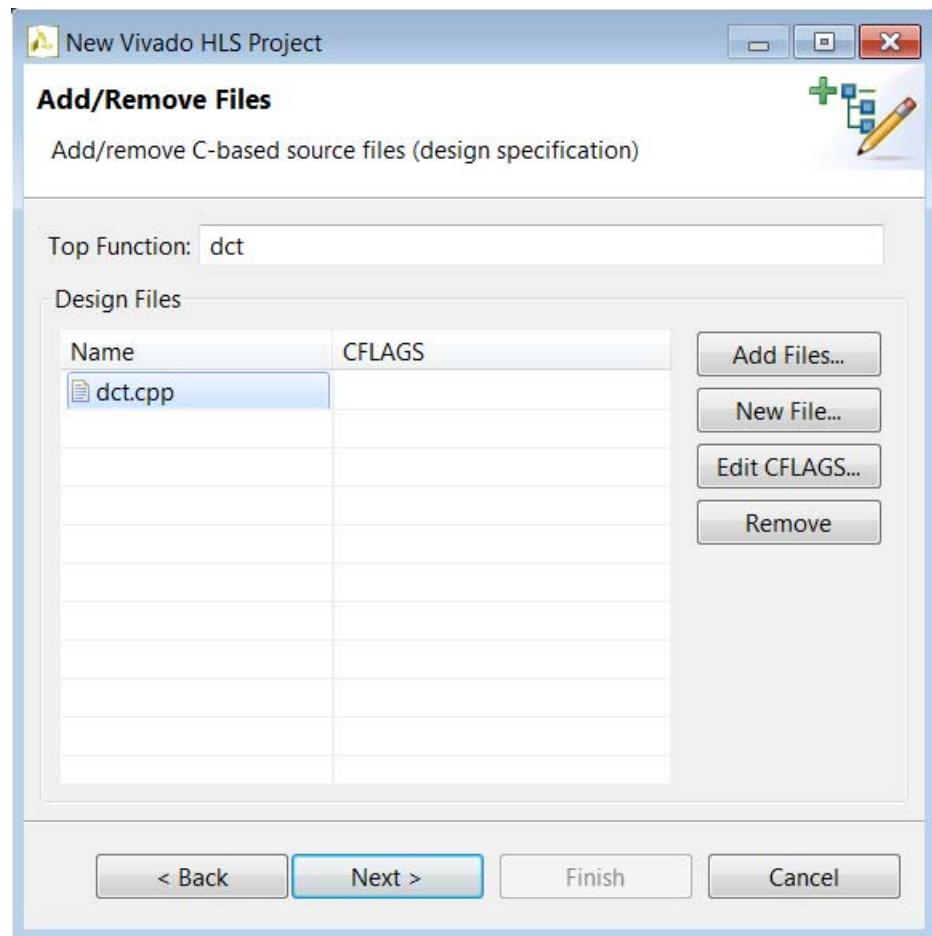


図 2-11: プロジェクトのソース ファイル

ウィザードの次のページでは、テストベンチに関連するファイルをプロジェクトに追加します。

C アルゴリズムを検証するのに使用した C テストベンチを、出力 RTL を検証するのに使用できます。Vivado HLS では、RTL デザインを C テストベンチにインスタンシエートするアダプターとラッパーが自動的に生成され、C と HDL の協調シミュレーションにより RTL が検証されます。RTL テストベンチを作成する必要はありません。

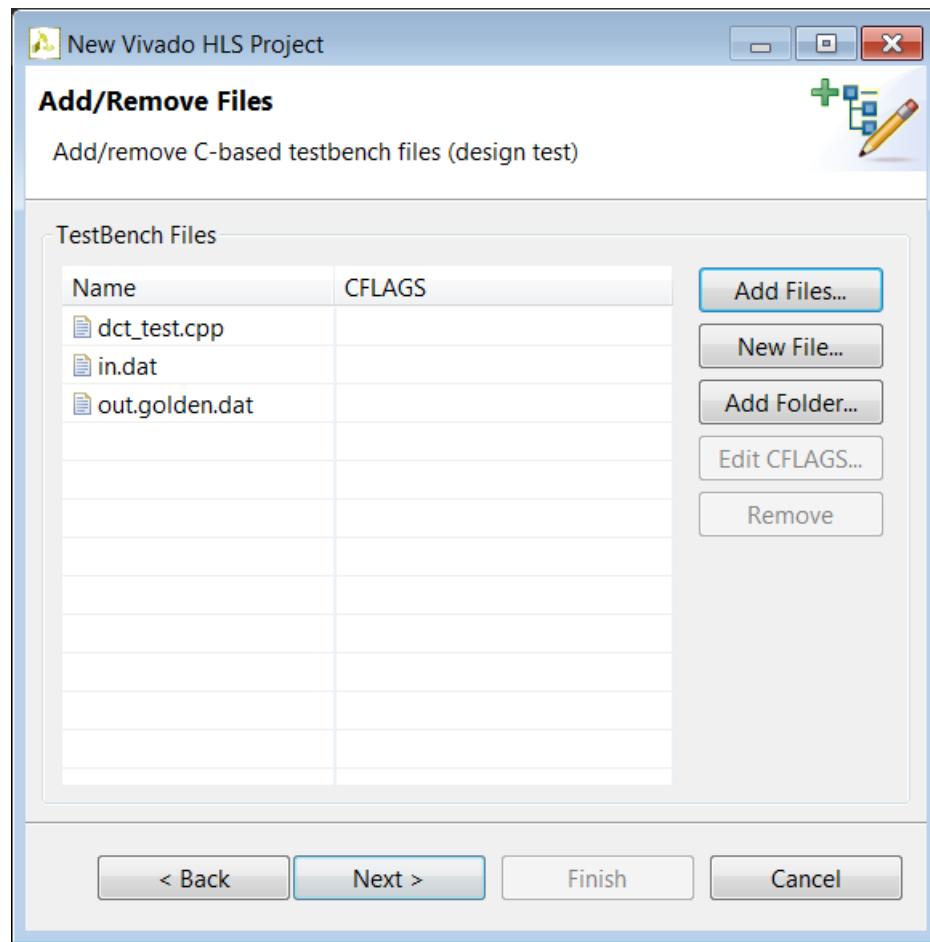


図 2-12: プロジェクトのテストベンチ ファイル

C ソース ファイルと同様に、[Add Files] をクリックして C テストベンチを追加し、[Edit CFLAGS] をクリックして C コンパイラ オプションを含めます。

C ソース ファイルと共に、テストベンチで読み込まれるすべてのファイルをプロジェクトに追加する必要があります。図 2-12 の例では、テストベンチで入力ステイミュラスを供給する *in.dat* ファイルと、予測される結果を読み込む *out.golden.dat* ファイルが使用されます。テストベンチがこれらのファイルにアクセスするので、これらもプロジェクトに追加する必要があります。

テストベンチ ファイルがディレクトリにある場合は、個々のファイルではなくディレクトリ全体を追加できます。

C テストベンチがない場合は、ここで何も入力する必要はありません。[Next] をクリックすると、最初のソリューションの詳細を入力するウィザードの最後のページが表示されます(図 2-13)。

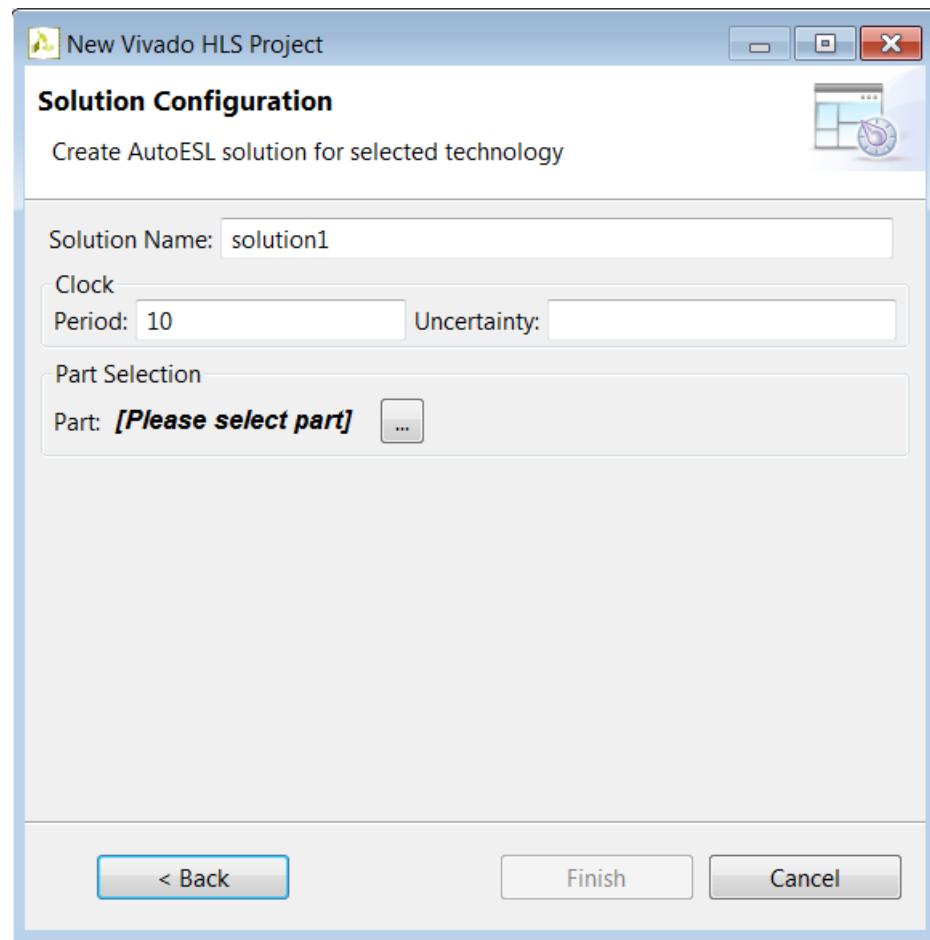


図 2-13 : 最初のソリューションの設定

図 2-13 では、次を指定します。

- [Solution Name] : ソリューション名を指定します。デフォルトでは、「solution1」が指定されています。
- [Clock] : クロック周期を ns で指定します。合成で使用されるクロック周期は、クロック周期からクロックのばらつきを引いた値になります。Vivado HLS でテクノロジライブラリのタイミング情報が使用され、RTL デザインが作成されます。[Uncertainty] に入力するクロックのばらつきの値は、RTL 合成、配置配線によるネット遅延の増加を考慮するためのマージンを ns で指定できます。指定しない場合、クロックのばらつきはクロック周期の 12.5% に設定されます。
- [Part] : [...] ボタンをクリックし、適切なパートを選択します (図 2-14)。

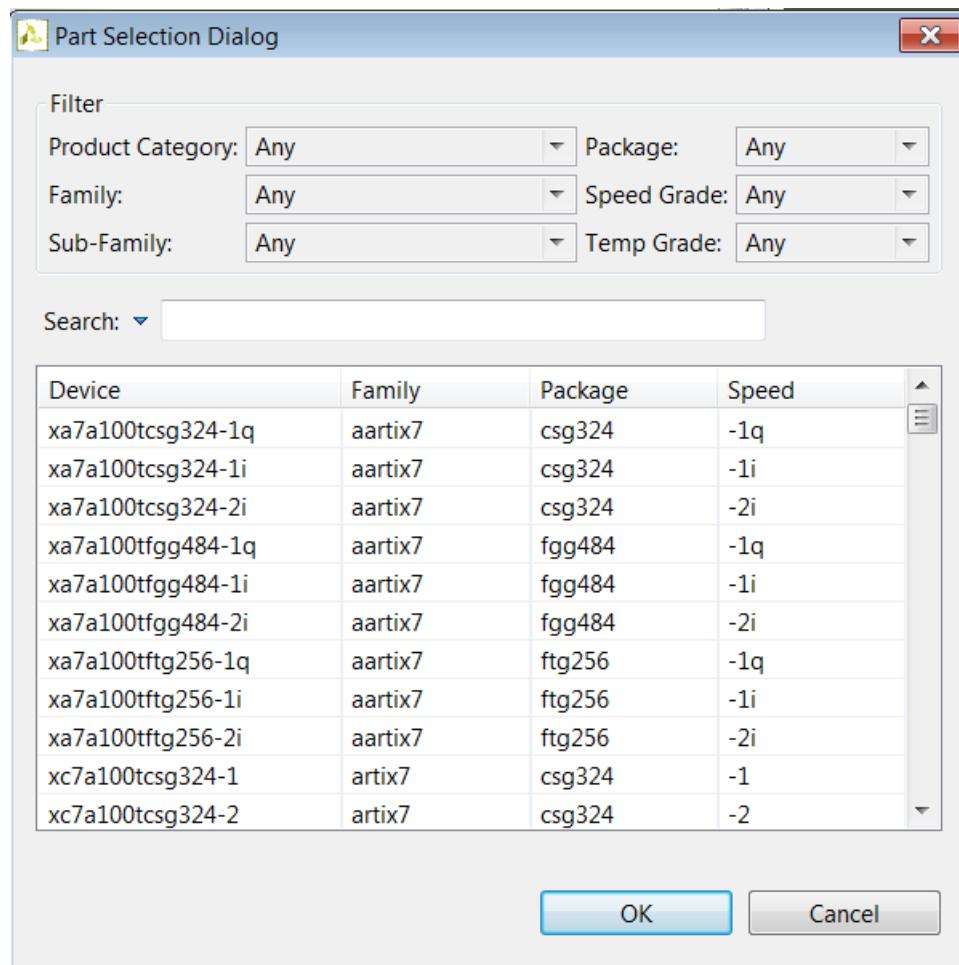


図 2-14 : パーツ選択

[Finish] をクリックすると、プロジェクトが開きます(図 2-15)。

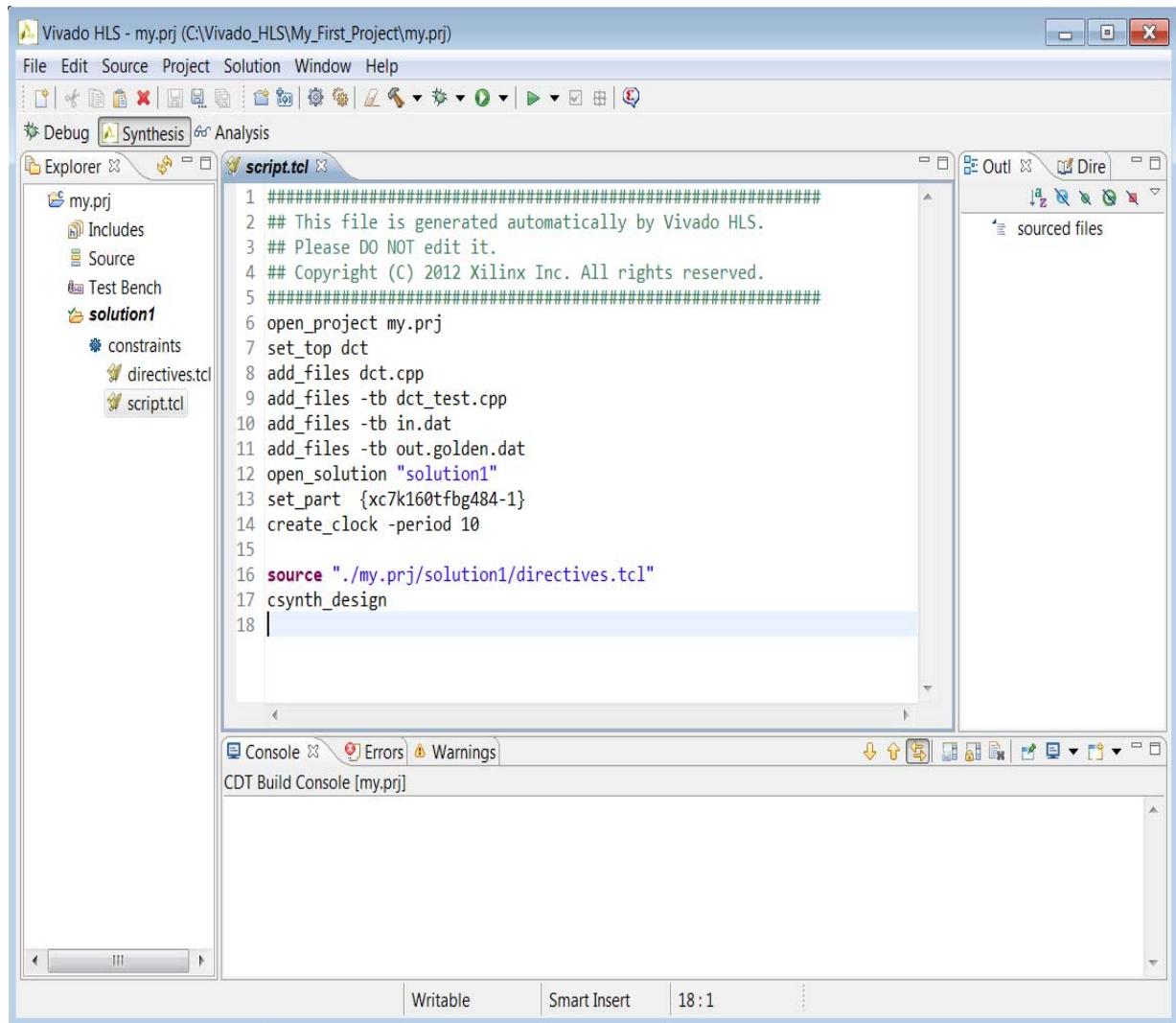


図 2-15: 新規プロジェクト

プロジェクトを作成した Tcl コマンドは、solution ディレクトリの script.tcl ファイルに保存されています。[Explorer] タブ (図 2-15 の左側) で script.tcl ファイルをダブルクリックすると、ファイルが情報エリアに開きます (図 2-15)。

Tcl バッチ スクリプトを作成する場合、script.tcl ファイルを開始点として使用すると便利です。[Explorer] タブに表示されるコンテナーは、プロジェクト ディレクトリにあります。solution ディレクトリからファイルをコピーしてください。

Vivado HLS の主要なコマンドは、ツールバーから実行できます (図 2-16)。現在実行可能なコマンド以外は、淡色表示されます。たとえば、RTL シミュレーションを実行する前に合成を実行する必要があるので、合成が完了するまで [RTL Simulation] ボタンは淡色表示になります。

図 2-16 で赤い四角で囲まれているコマンドは、次のとおりです(左から右)。

- [New Project] : 新規プロジェクトを作成します。
- [New Solution] : 新規ソリューションを作成します。
- [Project Settings] : 既存のプロジェクト設定を変更します。
- [Solution Settings] : 既存のソリューションの設定を変更します。
- [Compare Reports] : ソリューション レポートを比較します。
- [Clean C/C++ Project] : C シミュレーション環境をクリーンアップします。
- [Build C/C++/SystemC Project] : C/C++/SystemC 実行ファイルを作成します。
- [Run C/C++/SystemC Project] : C/C++/SystemC 実行ファイルを実行します。
- [Debug C/C++/SystemC Project] : C/C++/SystemC 実行ファイルをデバッグ モードで実行します。
- [Run Synthesize] : 合成を実行します。
- [RTL Simulation] : RTL シミュレーションを実行します。
- [Export RTL] : RTL デザインをエクスポートします。



図 2-16: ツールバー

ツールバーの各ボタンには、それに対応するメニュー コマンドがあります。

プロジェクトを作成した後の最初の手順は、C 関数の確認です。[Build...] ツールバー ボタンをクリックすると、図 2-17 に示すように、C デザインがコンパイルされます(デバッグまたは最適化されたリリース バージョン)。

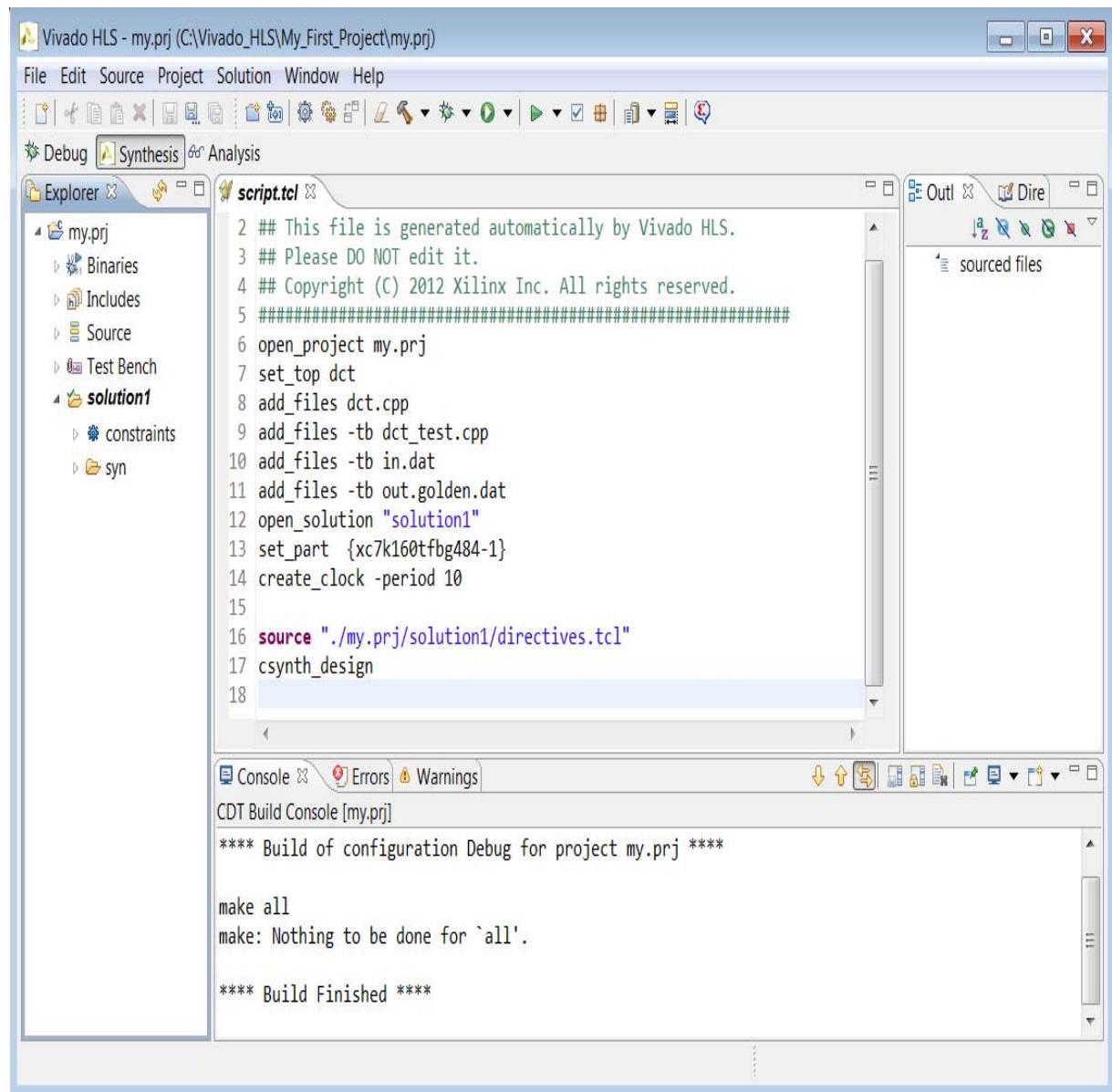


図 2-17 : C デザインをコンパイル

その後、ビルドを実行するか、オプションでデバッグ環境を表示します。[Debug...] ツールバー ボタンをクリックすると、デバッグ環境 (図 2-18) が開き、[Step..] ボタン (図 2-18 の赤で囲まれたボタン) を使用してコードを 1 行ずつ実行してその動作を解析できます。

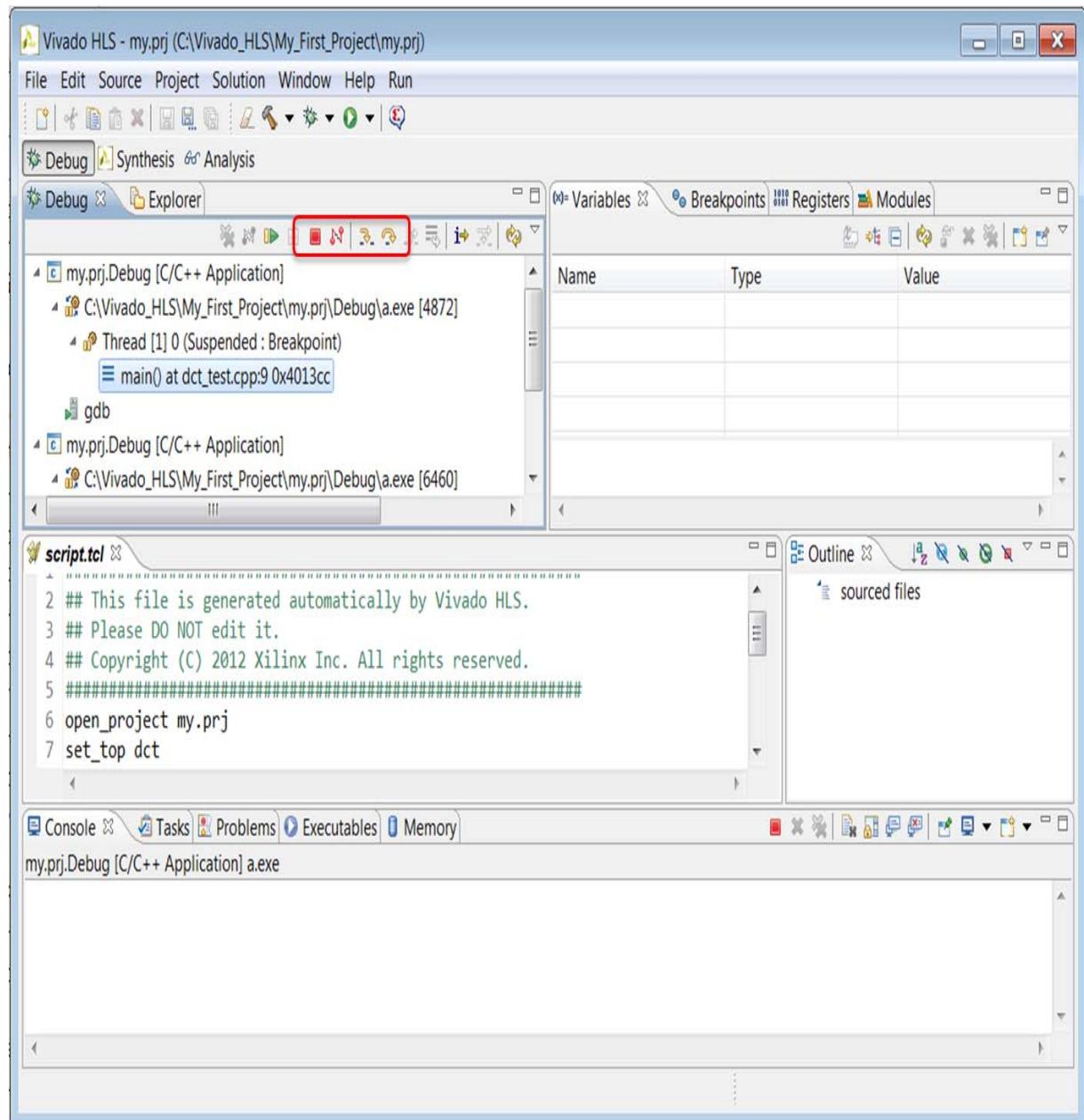


図 2-18 : C デバッグ環境

次に、合成を合成します。合成が完了すると、情報エリアに合成レポートが表示され、結果を解析できます(図 2-19)。

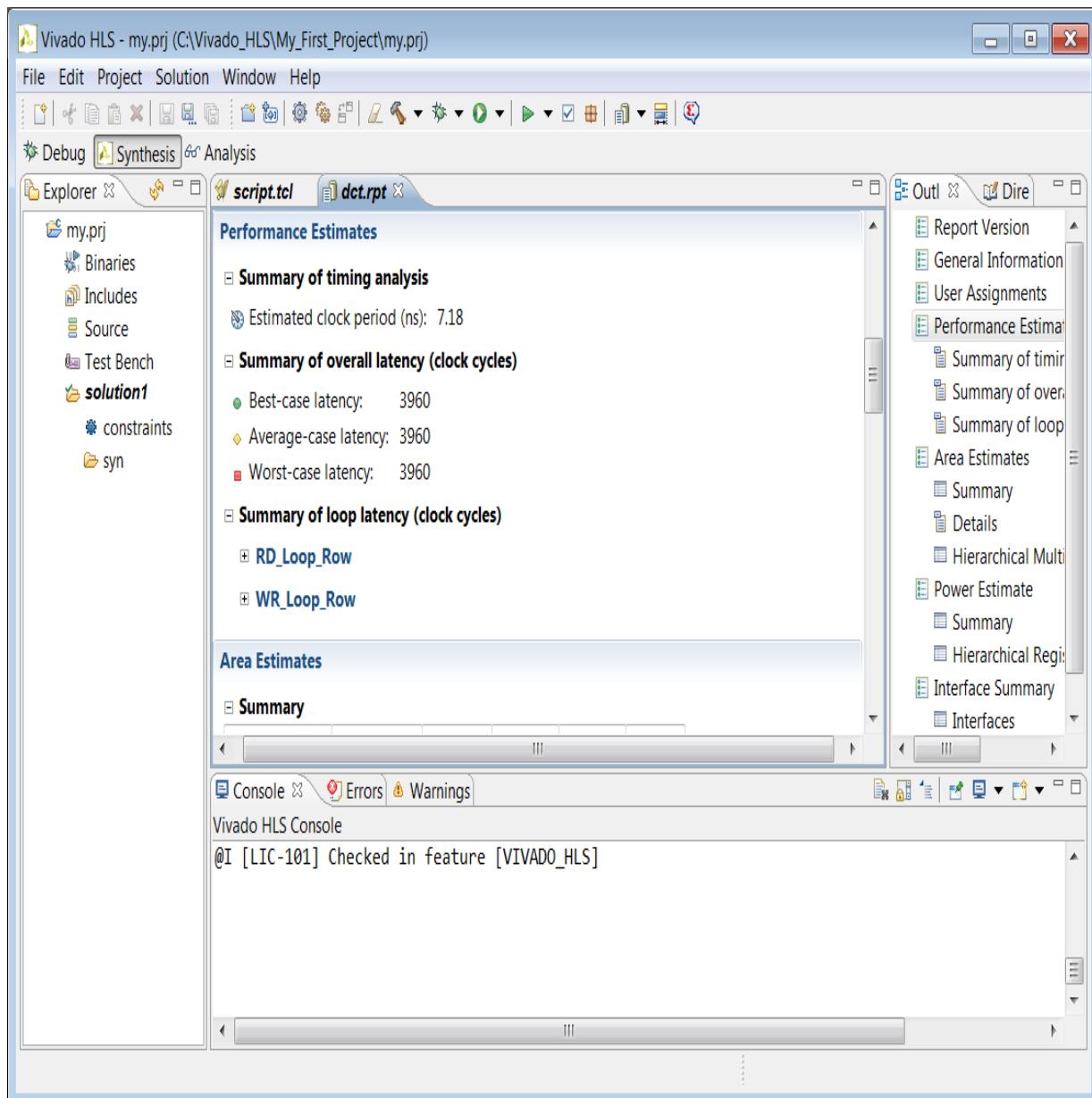


図 2-19 : 合成レポート

レポートには、RTL デザインのパフォーマンスとエリアに関する詳細が記述されています。[Outline] タブを使用すると、レポートを簡単にナビゲートできます。レポートは、関数がインライン化されていなければ、階層の各関数に対して作成されます(最適化の 1 つであるインライン化については後の章で説明)。最上位関数のレポートには、デザイン全体の詳細が表示されます。

表 2-2 は、合成レポートのカテゴリについて説明しています。

表 2-2 : 合成レポートのカテゴリ

カテゴリ	サブカテゴリ	説明
Report Version	---	結果を作成するのに使用された Vivado HLS のバージョンに関する詳細を示します。
General Information	---	プロジェクト名、ソリューション名、およびソリューションが実行された日時を示します。
User Assignments	---	ターゲット デバイス、ターゲット デバイスの属性、ターゲット クロック周期の詳細を示します。
Performance Estimates	Summary of timing analysis	予測される達成可能な最速クロック周波数。論理合成および配置配線はまだ実行されていないので、概算です。
	Summary of overall latency	デザインのレイテンシ。デザインのレイテンシとは、実行が開始されてから最終出力が書き込まれるまでのクロック サイクル数です。 ループのレイテンシにはらつきがある場合、ベスト ケース、平均、ワースト ケースのレイテンシは異なります。 デザインがパイプライン処理されている場合は、スループットが示されます。パイプライン処理されていない場合、スループットはレイテンシと同じになり、最終出力が書き込まれてから次の入力が読み出されます。
	Summary of loop latency	デザインに含まれる個々のループのレイテンシを示します。 「Trip count」は、ループの繰り返し回数を示します。 ループ レイテンシは、ループのすべての繰り返しを終了するためのレイテンシです。
Area Estimates	Summary	デザインをインプリメントするために使用されるリソース (LUT、フリップフロップ、DSP48 など) を示します。 サブカテゴリは、この表の「Details」に示します。
	Utilization	一部のデバイスに対して、リソースの使用率を示します。
	Details : Component	最上位デザイン内のコンポーネント (サブブロック) で使用されるリソースを示します。コンポーネントはデザイン内のサブ関数で作成されます。インライン化しない限り、関数はそれぞれが階層になります。 この例では、サブブロックがないので、デザイン階層は 1 つだけです。

表 2-2: 合成レポートのカテゴリ

カテゴリ	サブカテゴリ	説明
	Details : Expression	現在の階層レベルで乗算器、加算器、コンパレータなどの演算で使用されるエリアを示します。
	Details : FIFO	ここにリストされるリソースは、この階層レベルでの FIFO のインプリメンテーションに使用されるものです。
	Details : Memory	この階層レベルでメモリのインプリメンテーションに使用されるリソースをリストします。
	Details : Multiplexors	この階層レベルでマルチプレクサーのインプリメントに使用されるリソースをリストします。
	Details : Registers	この階層レベルで使用されるレジスタリソースを示します。
	Hierarchical Multiplexor Count	階層中のマルチプレクサーのサマリを示します。
Power Estimate	Summary	デバイスで使用される予測消費電力を示します。 この抽象レベルでは消費電力は概算であり、さまざまなソリューションの効率を比較するために使用します。
	Hierarchical Register Count	デザイン階層のレジスタで使用される予測消費電力を示します。
Interface Summary	Interface	関数およびポート (ポート名、方向、ビット幅など) で使用されるインターフェイスのタイプを示します。

Vivado HLS の最も一般的な使用方法は、まず初期デザインを作成し、その後エリアおよびパフォーマンスを満たすために最適化を実行する方法です。ソリューションにより、初期の合成を保持し、比較できます。

ソリューションの使用

新しいソリューションを作成するには、[New Solution] ツールバー ボタン (図 2-16) をクリックするか、[Project] → [New Solution] をクリックします。[Solution Wizard] ダイアログ ボックスが開きます (図 2-20)。

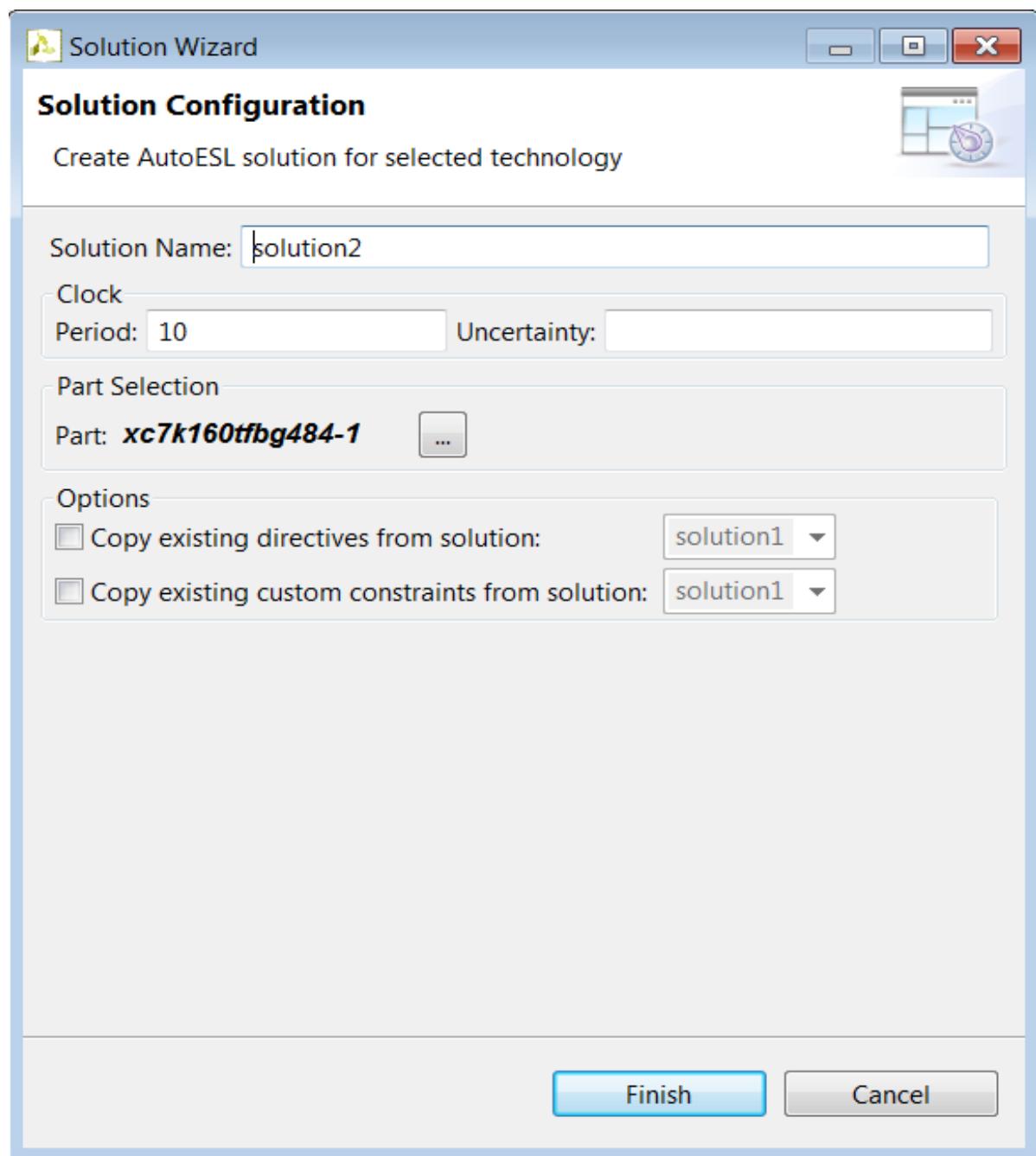


図 2-20 : [Solution Wizard] ダイアログ ボックス

このダイアログ ボックスには、New Vivado HLS Project ウィザードの最後のページ (図 2-15) と同じオプションと、既存の指示子やカスタム コマンドをコピーするオプションがあります。次のセクションで、ソリューションに指示子を追加する方法を説明します。

高位合成 GUI の使用

最適化の実行方法を説明する前に、Vivado HLS の GUI に情報がどのように表示されるか、GUI をどうやってカスタマイズするかを説明します。

Vivado HLS の GUI のデフォルト設定では、一部の情報が表示されないようになっています。次のような情報です。

- ヘッダーファイルに定義されている情報
- 英語以外の言語で記述されたソースのコメント

ヘッダーファイル情報の解決

デフォルトの Vivado HLS の GUI では、ヘッダーファイルを解析してすべてのコーディング コンストラクトは解決されません。このため、次のような状況が発生することがあります。

- コード ビューアーに変数または値が不明または定義できないと表示される。
- コードの変数が指示子 ウィンドウに表示されない。

どちらの場合も、不明の値および表示されていない変数はヘッダーファイル (拡張子が .h または .hpp) で定義されます。欠けている情報を解決するには、[Project] → [Project Settings] をクリックして [Parse all header files] をオンにします (図 2-21)。

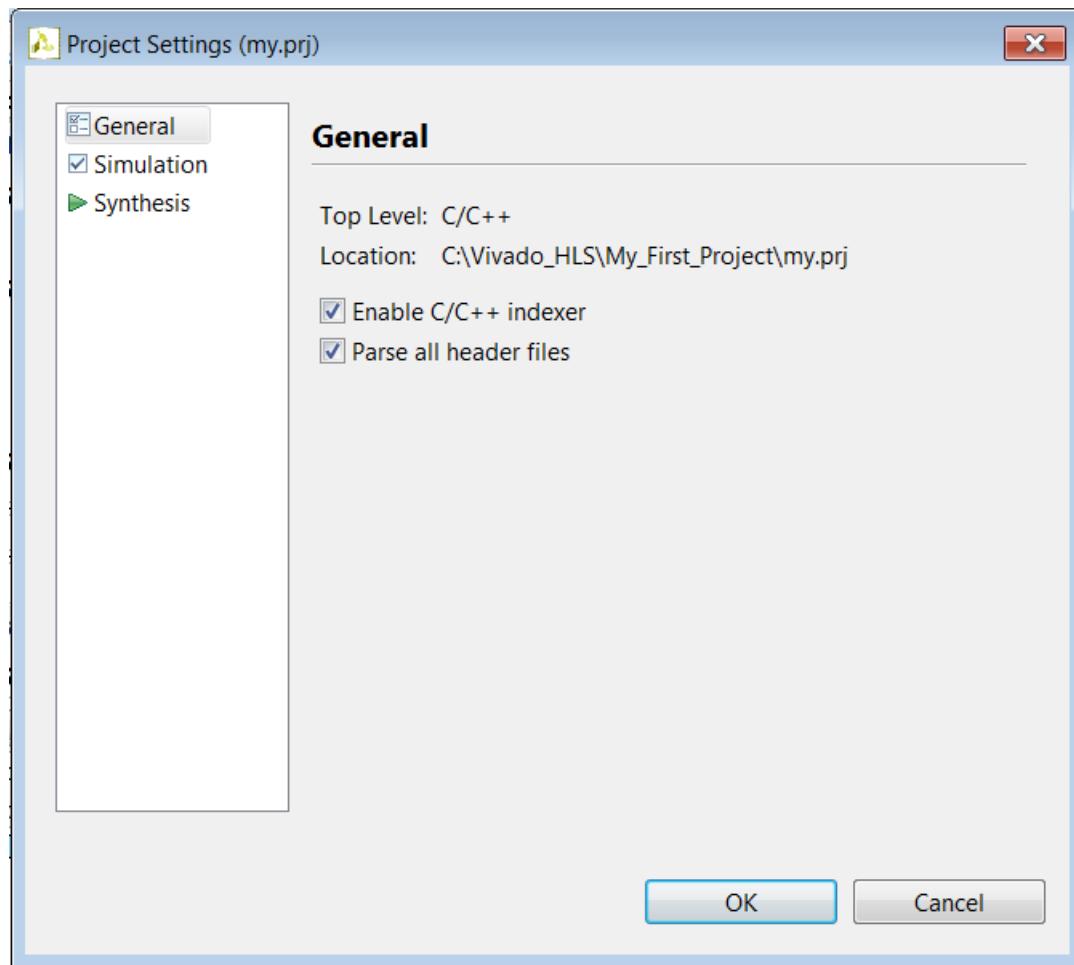


図 2-21: ヘッダーファイルの解析をイネーブル

注記 : [Parse all header files] をオンにすると、すべてのヘッダーファイルが継続的にポーリングされ、変更がないかどうかがチェックされます。CPU サイクルではヘッダーファイルのチェックは通常の動作なので、これにより GUI の応答時間が削減される場合があります。

ソースコードのコメントの解決

言語環境によっては、英語以外の言語のコメントが文字化けすることがあります。これを修正するには、次の手順に従います。

1. [Explorer] タブでプロジェクトを選択します。
2. 右クリックして [Properties] をクリックし、[Resource] ページで適切な言語エンコードを選択します。[Text file encoding] で [Other] をオンにし、ドロップダウンリストから適切なエンコーディングを選択します。

GUI のカスタマイズ

Vivado HLS の GUI の動作をカスタマイズするには、[Windows] → [Preferences] をクリックしてオプションを設定します。

[Preferences] ダイアログ ボックスではさまざまな詳細な設定が可能です。たとえば、キーの組み合わせ **Ctrl + Tab** は、デフォルトでは情報エリアのアクティブ タブをソース コードとヘッダーファイルで切り替えますが、各タブが順にアクティブになるように変更できます。

- [Preferences] ダイアログ ボックスの左側のボックスで [General] → [Keys] をクリックし、表の [Command] 列で [Toggle Source/Header] を選択して、[Unbind Command] をクリックして **Ctrl + Tab** の組み合わせを削除します。
- [Command] 列で [Next Tab] を選択し、下の [Binding] ボックスをクリックして **Ctrl** キーを押してから **Tab** キーを押すと、**Ctrl + Tab** キーを押したときに次のタブがアクティブになるよう設定できます。

[Preferences] ダイアログ ボックスの左側のボックスでさまざまな項目を選択することにより、Vivado HLS の GUI 環境を詳細にカスタマイズでき、生産性を向上することができます。

最適化のための指示子の使用

指示子は、デザインのさまざまな最適化を実行するために使用できます。このセクションでは、ソリューションに最適化を追加する方法を説明します。さまざまな最適化の詳細は、このユーザー ガイドの後の方の章で説明します。

最適化指示子を追加するには、まず情報エリアにソース コードを開きます。

[Explorer] タブで [Source] を展開し、ソース コードをダブルクリックして情報エリアを開きます (図 2-22)。

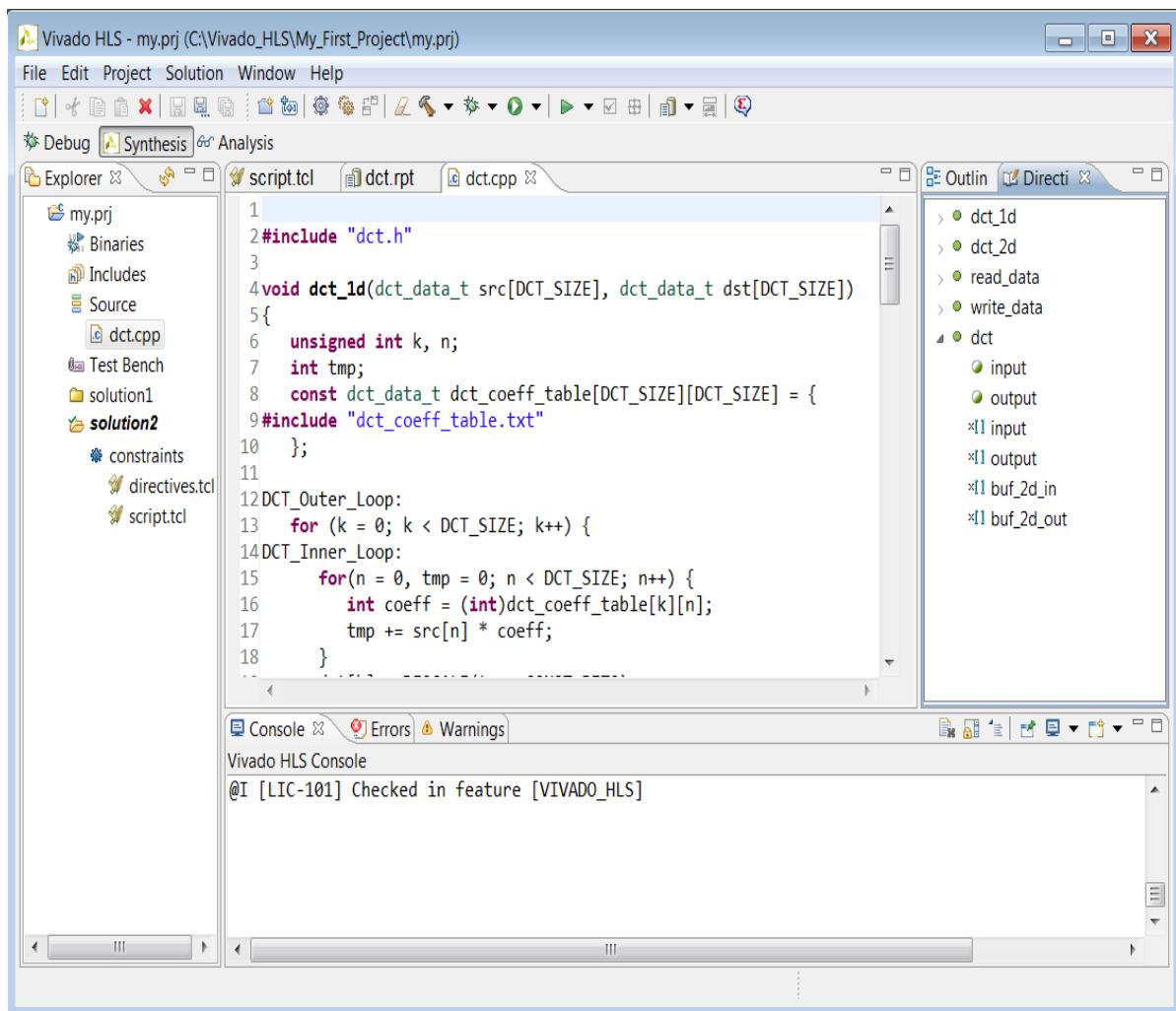


図 2-22 : ソースと指示子

情報エリアでソース コードをアクティブにすると、右側で [Directive] タブがアクティブになります。

[Directive] タブ

[Directive] タブには、開いているソース コードの指示子が適用されているオブジェクトが表示されます。

- 関数
- インターフェイス文
 - インターフェイスは、最上位関数への引数で、RTL デザインのポートとなります。指示子を使用して、インターフェイスに I/O プロトコル ポートを指定できます。
- 配列
- ループ
- 領域
 - 領域とは、中かっこ {} で囲まれている名前が付けられた領域です。

注記 : [Directive] タブに表示されるのは、現在情報エリアに表示されているファイル(現在アクティブなファイル)のオブジェクトのみです。デザインに含まれるすべてのファイルからのオブジェクトではありません。

次の例では、あるソースコードのアウトラインを示し、指示子を指定して最適化を実行可能な各オブジェクトをハイライトしています。

```
int foo_sub_A (int mem_1[64], ...){  
    for_A: for (int n = 0; n < 3; ++n) {  
        ...  
    }  
    ...  
}  
int foo_sub_B (int mem_1[64], int i) {  
    for_B:for (int n = 0; n < 4; ++n) {  
        ...  
    }  
    ...  
}  
void foo_top (int mem_1[64], int mem_2[64]) {  
    ...  
    for_top: for (int i = 0; i < 64; ++i) {  
        my_label:{  
            ...  
        }  
    }  
}
```

図 2-23 に、このコード例がどのように [Directive] タブに表示されるかを示します。

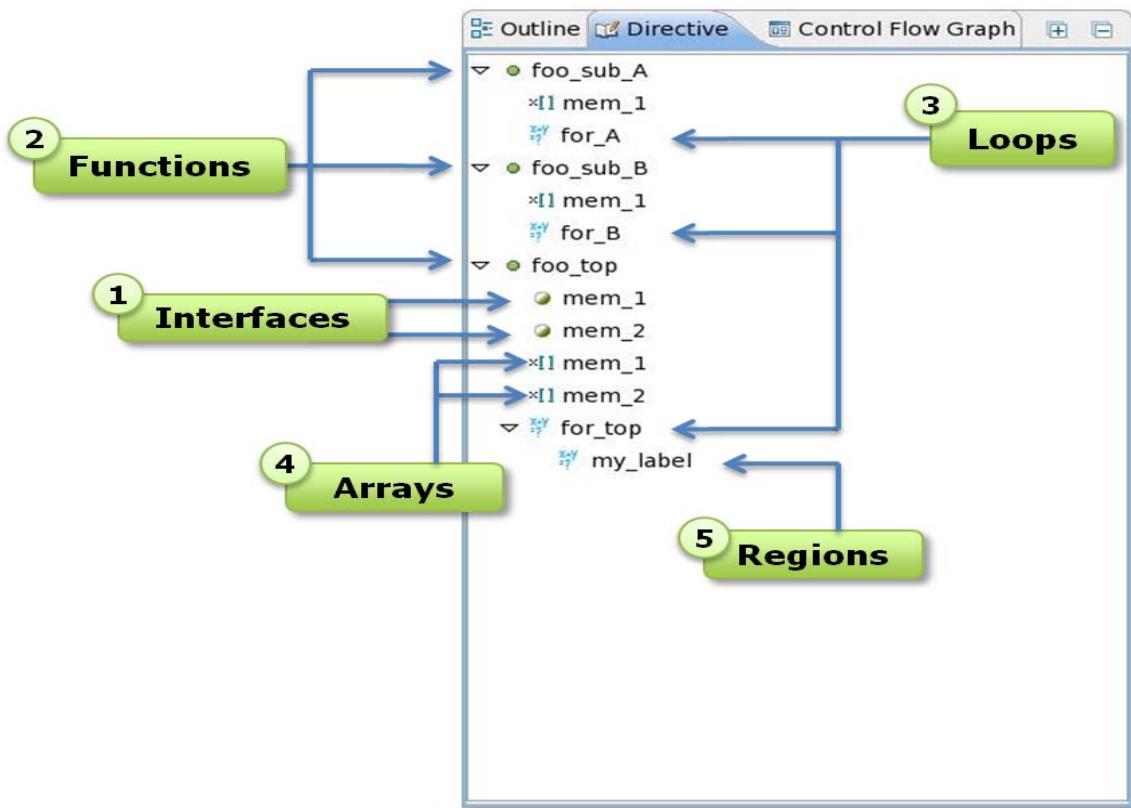


図 2-23 : [Directive] タブのオブジェクト

指示子の適用

指示子を適用するには、[Directive] タブでオブジェクトを右クリックし、[Insert Directive] をクリックして、[Vivado HLS Directive Editor] ダイアログ ボックス (図 2-24) を開きます。

[Directive] ドロップダウン リストで、追加する指示子を選択します。図 2-24 の例では、[DATAFLOW] が選択されており、データフロー指示子が追加されます。指示子のオプション (後の章で詳細に説明) のほかに、指示子を Tcl コマンドとして指示子ファイルに挿入するか、プラグマとしてコードに直接挿入するかを選択できます。

注記 : 指示子をヘッダーファイルのクラスなどのオブジェクトに追加するには、次のようにします。

指示子をクラス メンバーまたはグローバル変数に追加するには、その変数を使用する関数に対して [Vivado HLS Directive Editor] ダイアログ ボックスを開き、変数名を直接入力します。

指示子をローカル スカラーに追加するには、その変数を含む関数に対して [Vivado HLS Directive Editor] ダイアログ ボックスを開き、変数名を入力します。

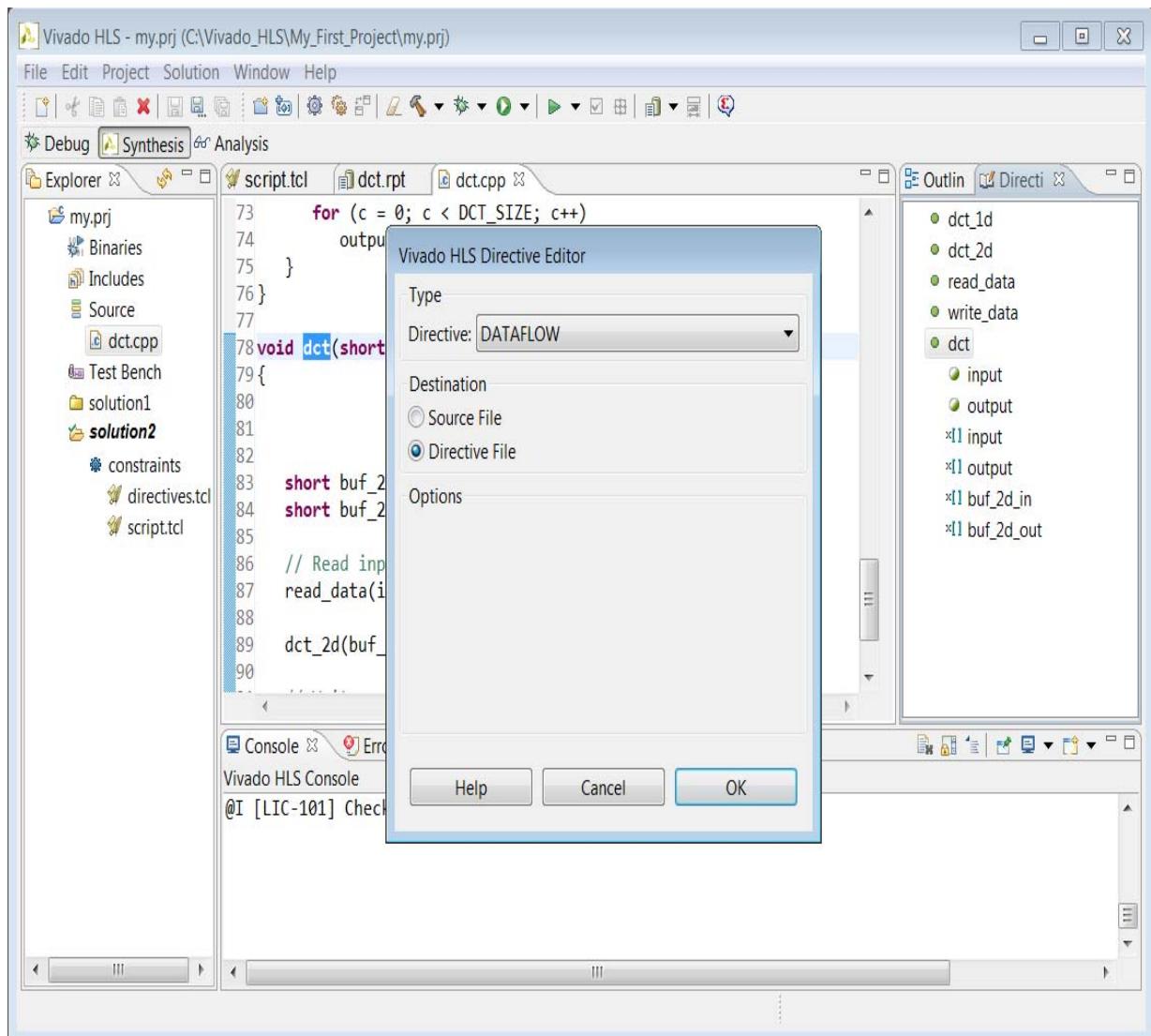


図 2-24 : 指示子の追加

[Vivado HLS Directive Editor] ダイアログ ボックスで [Directive File] をオンにすると、指示子が solution ディレクトリの directives.tcl ファイルに記述されます。指示子子ファイルに指示子を記述すると、次のような利点があります。

- 各ソリューションにそれぞれ異なる指示子を指定できます。
- Tcl パッチ ファイルを作成する場合、directives.tcl ファイルから指示子をコピーできます。

図 2-25 に、指示子を directives.tcl ファイルに記述するように設定し、その結果の directives.tcl ファイルを情報エリアに開いたところを示します。

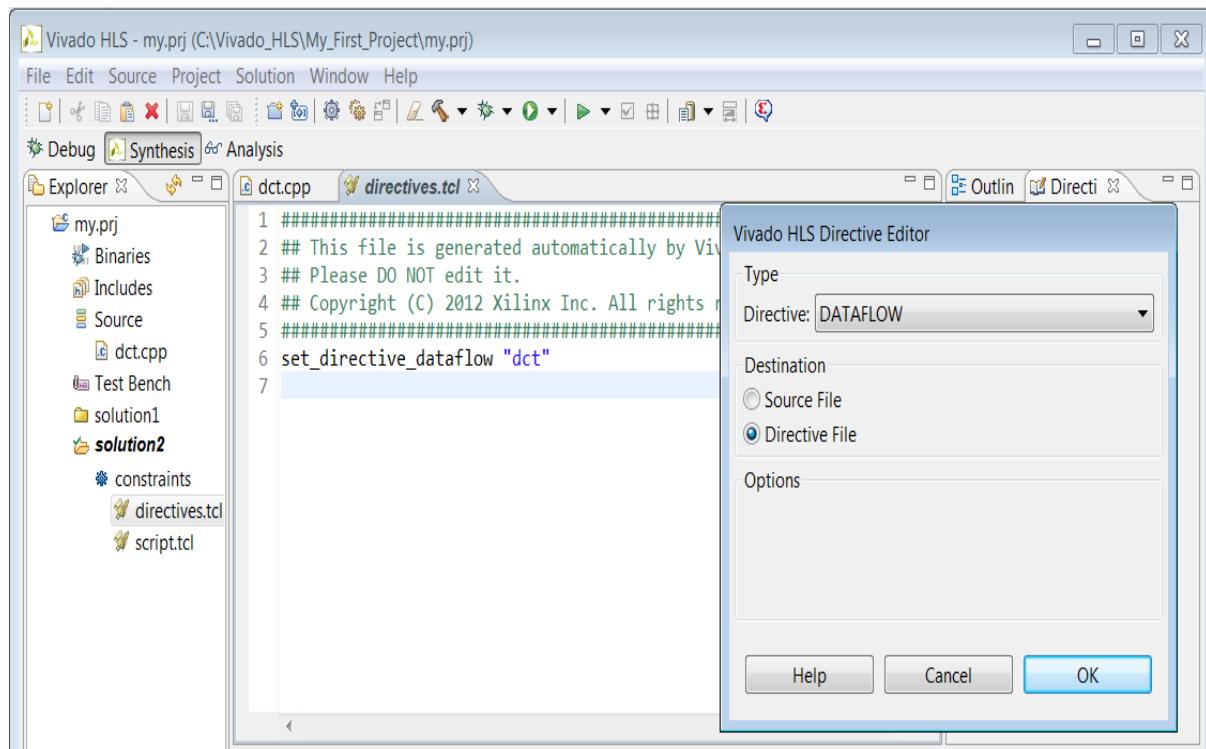


図 2-25 : Tcl 指示子の追加

もう 1 つのオプションは、プログラマをコードに追加する方法です。この方法の利点は、指示子が恒久的にコードに適用され、ほかのファイルは必要ありません。これは、IP をリリースする場合や、TRIPCOUNT 指示子などテクノロジターゲットによって変化しない指示子に適しています。

図 2-26 に、[Vivado HLS Directive Editor] ダイアログ ボックスの [Destination] で [Source File] をオンにして前述の例の指示子をプログラマとして適用し、その結果のソース コードを情報エリアに表示したところを示します。

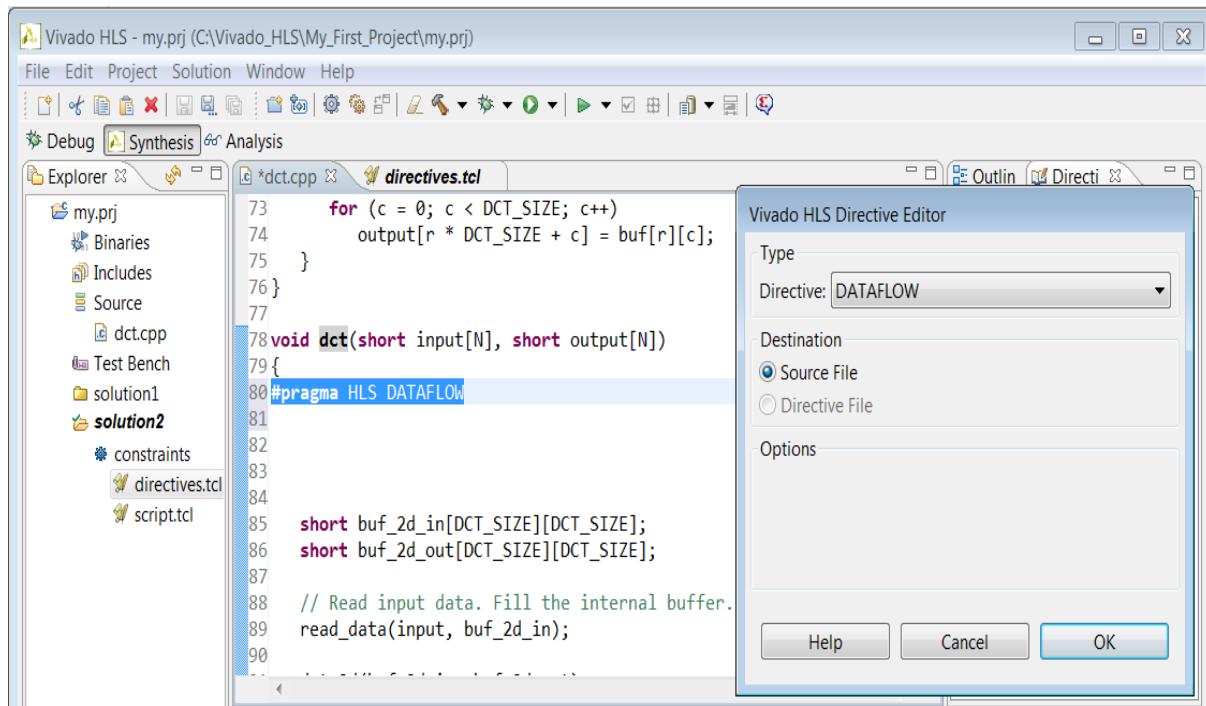


図 2-26: プラグマ指示子の追加

どちらの場合も、指示子が適用され、合成を実行すると最適化が実行されます。

プラグマを使用する唯一の欠点は、指示子が恒久的にソースコードに組み込まれ、すべてのソリューションに使用されることです。

チュートリアル

『Vivado Design Suite チュートリアル：デザインフローの概要』に、[Help] メニューからアクセスできます。

このチュートリアルは、Vivado HLS の使用に関する次のトピックを理解するのに役立ちます。

- C デザインの検証
- 高位合成プロジェクトの作成
- 合成およびデザイン解析
- アドレスビット精度デザイン
- デザイン最適化
- RTL 検証およびエクスポートの実行方法の理解
- Tcl スクリプトを使用した Vivado HLS ツールの使用

このチュートリアルで使用するサンプルデザインは、Vivado HLS のインストールディレクトリ内の examples ディレクトリにあります。

Cの検証とコーディングスタイル

HLS フローでの検証には、2 つのプロセスがあります。合成前の検証では C プログラムで必要な機能が正しくインプリメントされるかどうかを検証し、合成後の検証では RTL が正しいかどうかを検証します。これらのプロセスは、それぞれ C シミュレーションおよび RTL シミュレーションと呼ばれることもあります。

合成前の検証

合成前に、合成する関数をテストベンチを使用して検証する必要があります。理想的なテストベンチには、次のような特徴があります。

- セルフチェック機能がある。
- デザインとは別のファイルに記述されている（これは必須ではありませんが、推奨されます）。

テストベンチと合成する関数を別ファイルにしておくと、シミュレーションと合成のプロセスを分離できます。テストベンチが合成する関数と同じファイルに含まれている場合、Vivado HLS フローを少し変更する必要があります。テストベンチと合成する関数が含まれるファイルを、Vivado HLS プロジェクトにソースファイルとして、かつテストベンチとして追加する必要があります。

同様に、合成する関数を含むファイルにテストベンチ ファイルにない関数が含まれている場合、最上位関数より上位の関数をプロジェクトにテストベンチ ファイルとして追加する必要があります。

合成前の検証用に C デザインをコンパイルするプロセスは、Vivado HLS で図 2-17 および図 2-18 に示すように実行できます。C 検証は、Vivado HLS コマンド プロンプトから実行することもできます。

高位合成コマンド プロンプトでの C 検証

最上位デザインファイルが foo_top.c、テストベンチ ファイルが tb_foo_top.c である場合、次のコマンドを使用してテストベンチをコンパイルおよび実行できます。

```
$ gcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

高位合成コマンド プロンプトでの C++ 検証

最上位 C++ デザインファイルが foo_top.cpp、テストベンチ ファイルが tb_foo_top.cpp である場合、次のコマンドを使用してテストベンチをコンパイルおよび実行できます。

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
$ ./foo_top
```

高位合成コマンド プロンプトでの SystemC 検証

最上位デザインファイルが foo_top.cpp、テストベンチ ファイルが tb_foo_top.cpp である場合、次のコマンドを使用してテストベンチをコンパイルおよび実行できます。gcc コマンドのコマンド オプションをわかりやすいように複数行に分離していますが、実際には 1 行に記述します。

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
      -I$ vivado_hls_ROOT/Linux_x86_64/tools/systemc/include/
      -lsystemc
      -L$ vivado_hls_ROOT/Linux_x86_64/tools/systemc/lib-linux64
$ ./foo_top
```

SystemC が使用されているので、systemc.h ヘッダー ファイルをすべてのコンパイルに含める必要があります。

サポートされるバージョン以外の GCC の使用

合成前の C コードのコンパイルには、GCC バージョンを使用してください。Vivado HLS では、このバージョンの GCC の機能に一致する RTL が生成され、RTL と C テストベンチの協調シミュレーションを実行する場合にこのバージョンが使用されます。

Vivado HLS を起動する前に AP_SIM_GCC 環境変数を設定すると、RTL シミュレーションに異なるバージョンの GCC を使用できます。この変数は、ローカルバージョンの GCC を含むディレクトリへのパスと共に定義する必要があります。

Visual Studio コンパイラ

Vivado HLS を使用する前に、Microsoft Visual Studio コンパイラ (MVSC) を使用してコードをコンパイルできます。

任意精度整数を使用する場合など、関数を Vivado HLS ヘッダー ファイルと共にコンパイルする場合、特別な考慮事項があります (任意精度整数については「任意精度データ型」を参照)。

C のコンパイル

Vivado HLS ヘッダー ファイル ap_cint.h で定義される任意精度整数を使用する C 関数は、「C 言語での任意精度データ型」に説明するように APCC を使用してコンパイルする必要があります。Vivado HLS の任意精度型を使用する C デザインには、MVSC は使用できません。

C++ のコンパイル

Vivado HLS ヘッダー ファイルを含む C++ 関数では、MVSC でヘッダー ファイルの場所を指定する必要があります。

MVSC で Vivado HLS ヘッダー ファイルの場所を指定するには、次の手順に従います。

1. [Project] をクリックします。
2. [Properties] をクリックします。
3. 開いたパネルで、[C/C++] を選択します。
4. [General] を選択します。
5. [Additional Include Directories] をクリックし、図 2-27 に示すようにパスを追加します。

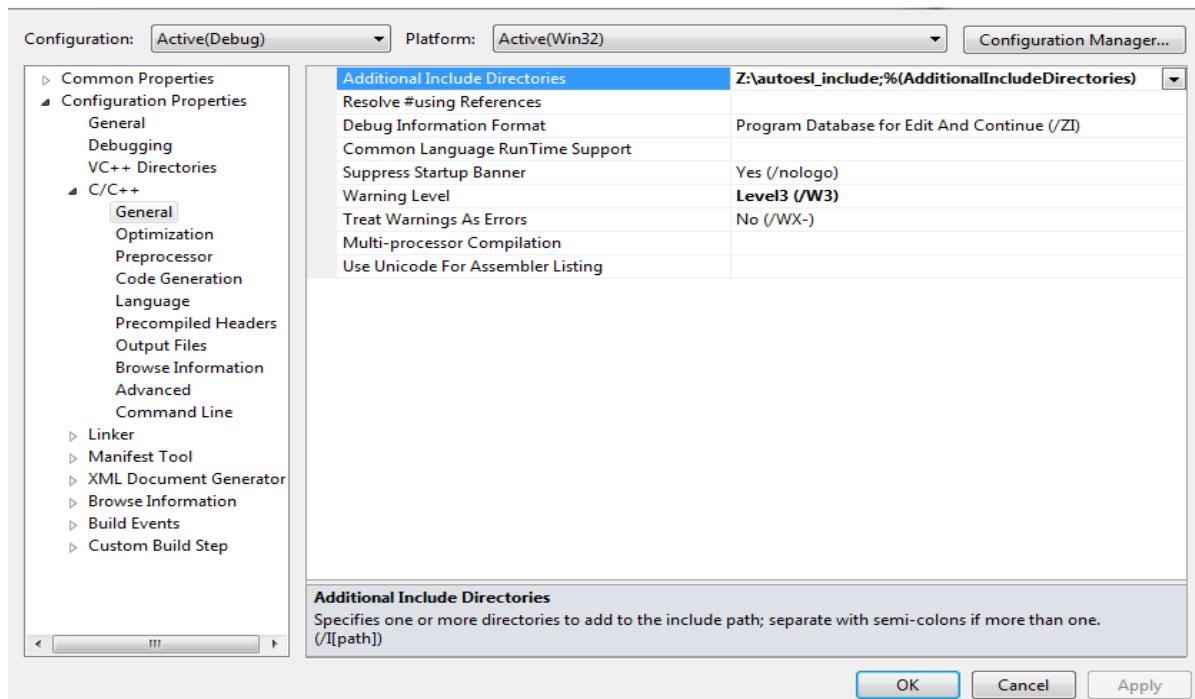


図 2-27 : Visual Studio での Vivado HLS インクルード パスの設定

サポートされていない C 言語コンストラクト

Vivado HLS では、C、C++、および SystemC の 3 つの C モデリング規格の大部分を合成できますが、合成できないコンストラクトもいくつかあります。このセクションでは、合成できないコンストラクトについて説明します。

C 関数を合成できるようにするには、デザインのすべての機能を含め (OS へのシステム コールで機能を実行することは不可)、C コンストラクトが固定のサイズであり、これらのコンストラクトのインプリメンテーションが明確である必要があります。次のコンストラクトは、これらの要件を満たしません。

システム コール

システム コールは、C プログラムを実行している OS で一部のタスクが実行されるので、合成できません。次に、合成できないシステム コールの例を示します。

- `printf()` は、情報をシステム コンソールに表示します。これは C シミュレーションでは有益ですが、合成できないので、最終的なハードウェア デザインの機能として含めることはできません。
- `fprintf()` は、プログラムを実行しているシステムのファイルにアクセスします。これは、最終的なハードウェア では実行できません。外部データにアクセスするには、最上位関数の引数またはグローバル変数を使用する必要があります。

`getc()`、`time()`、`sleep()` などのシステム コールも、OS に呼び出しを実行するので合成できません。

ほとんどのシステム コールは合成できません。よく使用されるシステム コールの一部 (`printf`、`cout` など) は Vivado HLS で無視されますが、通常は `_SYNTHESIS_` マクロを使用して合成で無視されるようにする必要があります。

プログラムがアクセスするメモリを割り当ているようなシステム コールがデザインに含まれている場合は、削除して同じ機能が別の方法で実行されるようにする必要があります。

ダイナミックメモリと関数

システム内のメモリ割り当てを制御するシステムコール (malloc()、alloc()、free() など) は、合成前にデザインコードから削除する必要があります。

これは、これらのシステムコールはランタイムに作成され、解放されるリソースにアクセスするからです。ハードウェアインプリメンテーションを合成可能にするためには、デザインが自己完結しており、必要なすべてのリソースが指定されている必要があります。

これらのシステムコールはデザインの機能を定義するのに使用されるので、同等の範囲が制限された表現に変換する必要があります。次に、ダイナミックメモリの割り当てを同等の範囲が制限された表現に変換する方法の例を示します。

```
#ifndef __SYNTHESIS__
// If synthesis is not required, use this code
long long x = malloc (sizeof(long long));
int* arr = malloc (64 * sizeof(int));
#else
// For synthesis, use this code
static long long x;
int arr[64];
#endif
```

推奨される方法は、上記の変更を加え、Cシミュレーションを再実行してコードを検証し、元のコードと動作が同じであることを確認します (gcc または g++ コンパイルに「-D__SYNTHESIS__」を追加)。

同様に、ダイナミック仮想関数呼び出しも合成できません。次のコードは、ランタイムで新しい関数を作成するので、合成できません。

```
Class A {
public:
    virtual void bar() {...};
};

void fun(A* a) {
    a->bar();
}
A* a = 0;
if (base)
    A= new A();
else
    A = new B();

foo(a);
```

ポインターの型変換

ポインターの型変換は通常サポートされませんが、ネイティブ C 型の間ではサポートされます。次のコードは合成できないので、例に示すように、元の型を使用して値を割り当てるよう変換する必要があります。

```
struct {
    short first;
    short second;
} pair;
#ifndef __SYNTHESIS__
// If synthesis is not required, use this code
*(unsigned*)pair = -1U;
#else
// For synthesis, use this code
```

```

pair.first = -1U;
pair.second = -1U;
#endif

```

再帰関数

ハードウェアインプリメンテーション用のC関数を作成するには、その機能をインプリメントするのに必要なリソースを Vivado HLS で判断できることが必要です。再帰関数は、再帰が恒久的に繰り返され、制限がない可能性があるので、合成できません。

```

unsigned foo (unsigned n)

{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}

```

末尾再帰は合成できます。この例では、再帰は最大制限に達するので、合成可能です。

```

unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
    if (n == 0) return m;
    return foo(n, m%n);
}

```

標準テンプレートライブラリ

C++ 標準テンプレートライブラリ (STL) には、再帰関数が含まれており、ダイナミックメモリ割り当てが使用されます。そのため、STL は合成できません。

この STL での問題を回避するには、再帰関数やダイナミックメモリ割り当てを含まない、同一の機能のローカル関数を作成します。

任意精度データ型

ネイティブ C データ型は、8 ビット境界 (8、16、32、64 ビット) にあります。RTL バス (ハードウェアに対応) では、任意の幅がサポートされます。HLS では、任意精度のビット幅の仕様を使用できるようにし、ネイティブ C データ型の制限に依存ないようにする機構が必要です。たとえば、17 ビット乗算器が必要な場合、32 ビット乗算器にインプリメントしなければならないような制限はあるべきではありません。

Vivado HLS では、C および C++ の任意精度データ型が提供されており、SystemC の一部である任意精度データ型がサポートされます。

任意精度データ型の利点は、C コードをビット幅の狭い変数を使用するようアップデートし、C シミュレーションを再実行して機能が同一であることを検証できる点です。

`__SYNTHESIS__` マクロを使用して、元のデータ型を参照として残し、任意精度データ型をソース コードに追加できます。

```
void foo {
    ifdef __SYNTHESIS__
        // use bit accurate type
        int8 a,
    #else
        // Original Source
        int a,
    #endif
    ...
};
```

Vivado HLS では、整数および固定小数点の両方のデータ型を使用できます。

整数データ型

Vivado HLS では、任意精度の整数データ型が提供されています(表 2-3)。これらの整数データ型により、指定の幅の制限内の整数値を制御できます。

表 2-3: 整数データ型

言語	整数データ型	必要なヘッダー
C	[u]int<precision> (1024 ビット)	apcc none gcc #include "ap_cin.h"
C++	ap_[u]int<W> (1024 ビット)	#include "ap_int.h"
System C	sc_[u]int<W> (64 ビット) sc_[u]bigint<W> (512 ビット)	#include "systemc.h"

C 言語での任意精度データ型

C 言語では、ヘッダーファイル `ap_cint.h` により任意精度の整数データ型 `[u]int` が定義されます。`ap_cint.h` ファイルは、`$VIVADO_HLS_ROOT/include` (where `$VIVADO_HLS_ROOT` は Vivado HLS のインストールディレクトリ) にあります。

C 関数で任意精度の整数データ型を使用するには、次の手順に従います。

- ソースコードにヘッダーファイル `ap_cint.h` を追加します。
- ビット型を `intN` または `uintN` (N はビットサイズを表す 1 ~ 1024 の値) に変更します。
- apcc コンパイラを使用してコンパイルします。
 - GUI で [Use APCC for Compiling C Files] オプションをオンにします。
 - コマンドプロンプトで `gcc` の代わりに `apcc` を使用します。

次の例に、ヘッダーファイルの追加方法と、2つの変数を 9 ビット整数型および 10 ビットの符号なし整数型を使用してインプリメントする方法を示します。

```
#include ap_cint.h

void foo_top (...) {
    int9 var1;           // 9-bit
    uint10 var2;          // 10-bit unsigned
```

注記: 標準 C コンパイラでは、C 任意精度がたを正しくシミュレーションできません。Vivado HLS の apcc ユーティティを使用する必要があります。

gcc などの標準 C コンパイラでは、ヘッダー ファイルで使用されている属性がコンパイルされてビット サイズが定義されますが、その意味を解釈できません。標準 C コンパイラで最終的な実行ファイルが作成されるときに、次のようなメッセージが表示されます。

```
$VIVADO_HLS_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute
directive ignored
```

そして、シミュレーションにネイティブ C データ型が使用され、コードのビット精度動作が結果に反映されません。

Vivado HLS には apcc というコンパイラが含まれており、この制限を克服し、関数をビット精度でコンパイルおよび検証できます。

apcc コンパイラをイネーブルにするには、[Project] → [Project Settings] をクリックし、左側のボックスで [Simulation] を選択して、[Use APCC for Compiling C Files] をオンにします (図 2-28)。

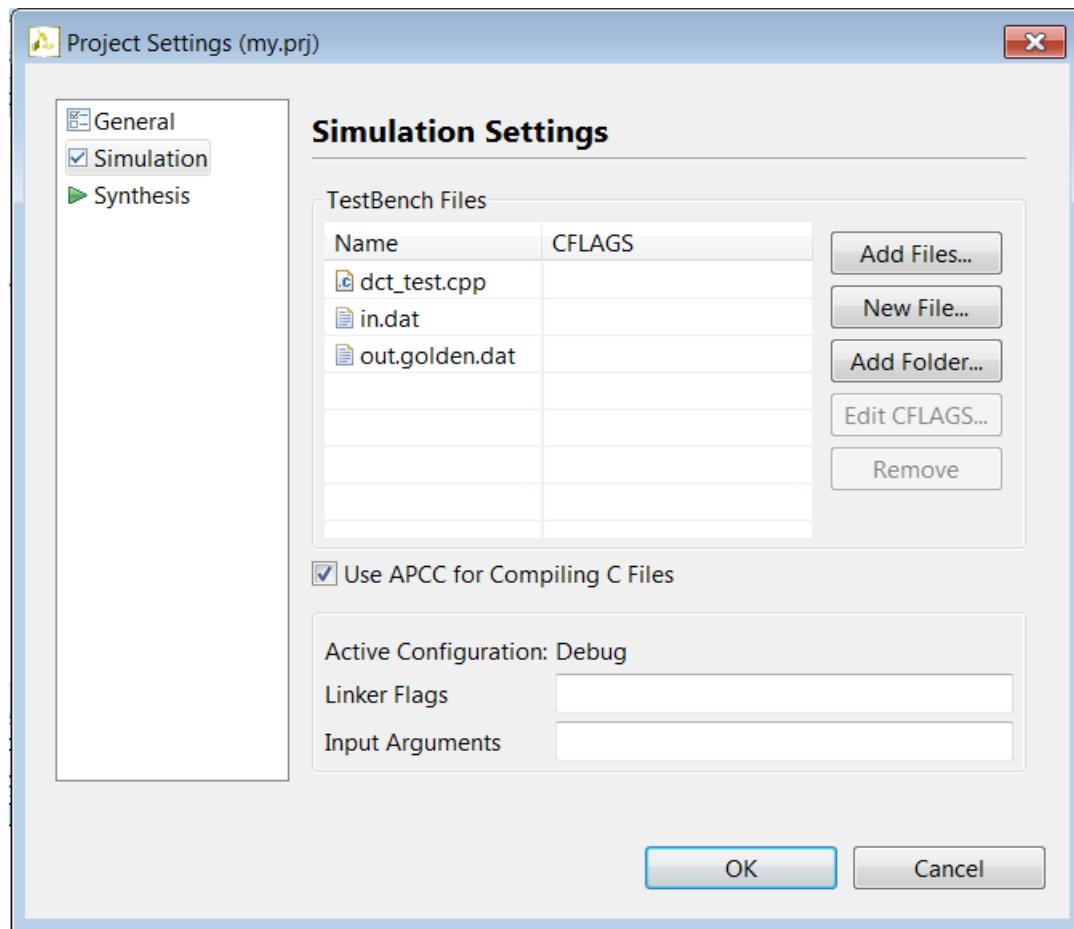


図 2-28 : APCC コンパイラのイネーブル

注記 : [Use APCC for Compiling C Files] をオンにすると、デザインをデバッガーで解析することはできなくなります。これが、C コードで任意精度型を使用する欠点です。

C++ または SystemC で指定された関数では、任意精度型を使用した場合にそのような制限はありません。この制限は C を使用する場合にのみ存在します。

C++ または SystemC 関数のコンパイルに APCC を使用しないでください。APCC を選択した場合、無視されます。

コマンドプロンプトでコンパイルする場合は、シェルプロンプトで apcc コンパイラを使用してください。これは gcc に互換しており、任意精度演算を正しく処理します(ビット幅情報で指定された制限を保持)。

apcc を使用すると Vivado HLS ヘッダー ファイルが自動的に含まれ (-I\$VIVADO_HLS_ROOT/include は不要)、デザインをシミュレーションしたときに正しいビット精度の動作が得られます。

```
$ apcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

C++ 言語での任意精度データ型

C++ 言語では、ヘッダー ファイル ap_int.h により任意精度の整数データ型 ap_[u]int が定義されます。ap_int.h ファイルは、\$VIVADO_HLS_ROOT/include (\$VIVADO_HLS_ROOT は Vivado HLS のインストール ディレクトリ) にあります。

C++ 関数で任意精度の整数データ型を使用するには、次の手順に従います。

- ソースコードにヘッダー ファイル ap_int.h を追加します。
- ビット型を ap_int<N> または ap_uint<N> (N はビット サイズを表す 1 ~ 1024 の値) に変更します。

次の例に、ヘッダー ファイルの追加方法と、2 つの変数を 9 ビット整数型および 10 ビットの符号なし整数型を使用してインプリメントする方法を示します。

```
#include ap_int.h

void foo_top (...) {

    ap_int<9> var1;           // 9-bit
    ap_uint<10> var2;         // 10-bit unsigned
```

任意精度整数を使用する場合、シミュレーションにヘッダー ファイル ap_int.h へのパスを含める必要があります。

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp -I$VIVADO_HLS_ROOT/include
```

SystemC 言語での任意精度データ型

SystemC で使用される任意精度型は、ヘッダー ファイル systemc.h で定義されています。このヘッダー ファイルは、すべての SystemC デザインに含める必要があります。SystemC sc_int<>、sc_uint<>、sc_bigint<>、および sc_bignum<> 型が含まれます。

SystemC デザインをシミュレーションする際、SystemC ヘッダー ファイルへのパスを含める必要があります。Linux では、最上位デザイン ファイルが foo_top.cpp、テストベンチ ファイルが tb_foo_top.cpp である場合、次のコマンドを使用してテストベンチをコンパイルおよび実行できます。gcc コマンドのコマンド オプションをわかりやすいように複数行に分離していますが、実際には 1 行に記述します。

```
$ g++ -o foo_top foo_top.cpp tb_foo_top.cpp
      -I$VIVADO_HLS_ROOT]\Win_x86\tools\systemc\include
      -lsystemc
      -L$VIVADO_HLS_ROOT\Win_x86\tools\systemc\lib
$ ./foo_top
```

固定小数点データ型

固定小数点データ型を使用して実行された C++/SystemC シミュレーションの動作は、合成で作成されたハードウェアの結果と一致し、ビット精度、量子化、およびオーバーフローの影響を高速の C レベル シミュレーションで解析できるので、HLS を使用する場合、固定小数点データ型の使用は特に重要です。

Vivado HLS では、C++ および SystemC に任意精度の固定小数点データ型を使用できます (表 2-4)。

表 2-4 : 固定小数点データ型

言語	固定小数点データ型	必要なヘッダー
C	-- なし --	-- なし --
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
System C	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h"

これらのデータ型は、指定の合計幅および整数幅の制限内での浮動小数点値を制御します (図 2-29)。

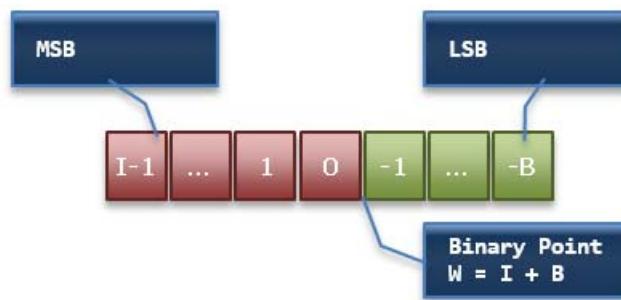


図 2-29 : 固定小数点データ型

表 2-5 に、固定小数点でサポートされる演算の概要を示します。

表 2-5 : 固定小数点の識別子

識別子	説明		
W	ワード長をビット数で指定		
I	整数値をビット数で指定 (整数部のビット数)		
Q	量子化モードを指定。量子化モードは、結果の保存に使用される変数の最小の小数ビットで定義できるよりも大きい精度が生成された場合の動作を指定します。 SystemC 型 SC_RND SC_RND_ZERO SC_RND_MIN_INF AP_RND_INF AP_RND_CONV AP_TRN AP_TRN_ZERO	AP_Fixed 型 AP_RND AP_RND_ZERO AP_RND_MIN_INF AP_RND_INF AP_RND_CONV AP_TRN AP_TRN_ZERO	説明 正の無限大への丸め 0への丸め 負の無限大への丸め 無限大への丸め 収束丸め 負の無限大への切り捨て 0への切り捨て (デフォルト)

表 2-5: 固定小数点の識別子

識別子	説明		
O	折り返しモードでの飽和ビット数 SystemC 型 SC_SAT SC_SAT_ZERO SC_SAT_SYM SC_WRAP SC_WRAP_SM	AP_SAT AP_SAT_ZERO AP_SAT_SYM AP_WRAP AP_WRAP_SM	飽和 0への飽和 対称飽和 折り返し (デフォルト) 符号絶対値の折り返し
N	折り返しモードでの飽和ビット数		

ap_fixed

次の例では、Vivado HLS の `ap_fixed` 型を使用して、18 ビット変数 (6 ビットが整数部、12 ビットが小数部を表す) を定義しています。変数は符号付き、量子化モードは丸め、オーバーフロー モードはデフォルトの折り返しに指定されています。

```
#include <ap_fixed.h>
...
ap_fixed<18,6,AP_RND> my_type;
...
```

sc_fixed

次の `sc_fixed` 型の例では、22 ビット変数が整数部 21 ビットで定義されており、最小精度は 0.5 です。0 への丸めが使用されているので 0.5 未満の結果はすべて 0 に丸められ、飽和が指定されています。

```
#define SC_INCLUDE_FX
#define SC_FX_EXCLUDE_OTHER
#include <systemc.h>
...
sc_fixed<22,21,SC_RND_ZERO,SC_SAT> my_type;
...
```

浮動小数点型

Vivado HLS でデザインを合成すると、デザインの演算が演算子に変換され、それらがテクノロジ ライブラリのコアにマップされます。浮動小数点デザインでは、すべての演算子に対応する浮動小数点コアがライブラリに存在するわけではありません。

テクノロジ ライブラリにマップできるコアがない場合、Vivado HLS での合成は停止され、マップするライブラリ コアがないことを示すメッセージが表示されます。Vivado HLS ライブラリに含まれるすべてのコアのリストは、第 3 章「高位合成演算子およびコア ガイド」を参照してください。

浮動小数点演算

ライブラリからの浮動小数点コアを使用するには、演算のすべての引数が浮動小数点引数である必要があります。次のようなコードがあるとします。

```
A = B/2;
```

これは、次のように変更する必要があります。

```
A = B * 0.5;
```

このように変更すると、乗算に浮動小数点演算子が使用されます。Vivado HLS では、定数データ型は自動的に変換されません。

標準の算術演算子 (+、-、*、/、および %) は、テクノロジライブラリの浮動小数点コアでサポートされています。変数を浮動小数点として定義し、標準の算術演算子で使用すれば、インプリメンテーションで浮動小数点コアが使用されます。

浮動小数点の数学関数

C/C++ の数学関数では、関数を宣言する必要があります。たとえば、sqrtf() 関数を使用するには、コードに次を追加する必要があります。

```
#include <math.h>
extern "C" float sqrtf(float);
```

これで、浮動小数点変数に sqrtf() 関数を使用できるようになります。

マルチアクセス ポインター インターフェイス

最上位関数の引数リストにポインターを使用するデザインでは、ポインターを使用して複数アクセスを実行する際に特別な注意事項があります。複数アクセスは、同じ関数でポインターが複数回読み出されたり書き込まれたりすることです。

次の例では、入力ポインター d_i が 4 回読み出され、出力ポインター d_o が 2 回書き込まれます。

```
#include "fifo.h"

void fifo ( int *d_o,  int *d_i) {
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

マルチアクセス ポインターを使用する場合、次を実行する必要があります。

- volatile 修飾子を使用する必要があります。
- cosim_design で RTL を検証する場合は、ポート インターフェイスでのアクセス数を指定します。
- 合成の前に C を検証し、目的と C モデルが一致していることを確認してください。

揮発性インターフェイスの理解

上記のコードは、入力ポインター d_i および出力ポインター d_o を、RTL でハンドシェイク付きのインターフェイス (FIFO またはハンドシェイク ポート) としてインプリメントすることを意図して記述されています。これにより、次のことが確実になります。

- アップストリームの送信ブロックは、RTL ポート d_i で読み出しが実行されるたびに新しいデータを供給します。
- ダウンストリームの受信ブロックは、RTL ポート d_o で書き込みが実行されるたびに新しいデータを受信します。

ただし、このコードを C コンパイラでコンパイルすると、各ポインターへの複数アクセスが 1 つのアクセスに削減されます。コンパイラには、d_i 上のデータが関数の実行中に変化することは示されておらず、d_o への最終の書き込みのみが必要であると解釈されます（ほかの書き込みは、関数が完了するまでに上書きされる）。

Vivado HLS での処理は、C コンパイラの動作と一致しており、複数の読み出しおよび書き込みは 1 つの読み出し操作および 1 つの書き込み操作に最適化されます。

このデザインが RTL で意図したとおりに動作するようにするには、次の例に示すように、コードに volatile キーワードを使用します。

```
#include "fifo.h"

void fifo ( volatile int *d_o,  volatile int *d_i) {
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

}
```

volatile キーワードを使用すると、C コンパイラおよび Vivado HLS でデータは揮発性であり変化する可能性があると解釈され、複数アクセスが 1 つのアクセスに最適化されることになります。

この例は、意図したとおり、入力ポート d_i で読み出しが 4 回実行され、出力ポート d_o で書き込みが 2 回実行されます。

ただし、ポインターに複数回アクセスするコードでは、volatile キーワードを使用しても、テストベンチと検証に関連する問題が 2 つあります。

次のような結果が発生する可能性があります。

- cosim_design シミュレーションでのエラー
- モデリングとシミュレーションが一致しない。

RTL の cosim_design シミュレーションでのエラー

次のテストベンチを使用して上記のアルゴリズムを検証できます。このテストベンチは、コードの機能をハイライトするため、関数の 4 回の実行 (4 つのトランザクション) をモデリングしています。

注記：このテストベンチには、わかりやすくするため、セルフチェック機能は含めていません。

```
#include <stdio.h>

#include "foo.h"

int main () {
    int d_o, d_i;

    for (d_i=0;d_i<4;d_i++) {
        foo(&d_o,&d_i);
        printf("%d %d\n", d_i, d_o);
    }

    return 0;
}
```

この例で使用されているヘッダー ファイル foo.h は次の内容で、単に関数を宣言しており、関数 foo を別のファイルで定義できるようにしています。

```
#ifndef FOO_H_
#define FOO_H_
void foo ( volatile int *d_o,  volatile int *d_i);

#endif
```

このテストベンチの問題は、各トランザクションで関数に 1 つの値のみが供給されることです。

各トランザクションで、テストベンチにより 1 つの値のみが供給されますが、volatile キーワードが使用されているため、関数 foo は読み出しを 4 回実行し、同じ値が 4 回読み出されます。

このテストベンチにより、次のような結果が得られます。出力は、4 回の入力読み出しの累積と、前回のトランザクションが加算された値になります。

```
Di Do
0 0
1 4
2 12
3 24
```

RTL 検証を実行すると、volatile キーワードにより、読み出しを 4 回実行する RTL が作成され、ハンドシェイク付きインターフェイスに合成された場合、ポート d_i が読み出されるたびに新しいデータが送信されるようになります。

cosim_design で RTL を検証するため、Vivado HLS で RTL にアダプターを含む SystemC ラッパーが作成され、この(C コード)ラッパーが既存の C テストベンチにインスタンシエートされます(図 2-30)。

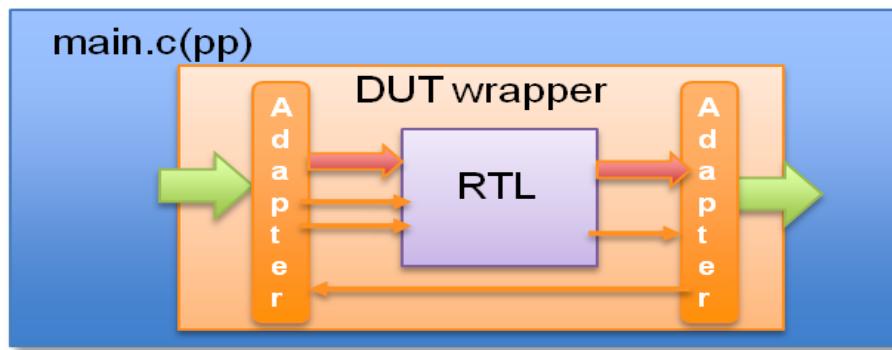


図 2-30 : cosim_design ラッパーの概要

Vivado HLS で作成されるラッパーは、RTL インターフェイスにハンドシェイクが必要な場合はそれがモデリングされ、テストベンチから供給される DUT への入力値が RTL デザインで必要なときに準備されていることを確実にする必要があります。これには、ストレージが必要です。

Vivado HLS では、このようなポインターを使用する関数インターフェイスで読み出しありは書き込みが何回実行されるかを判断できません。関数インターフェイスの引数では、値がいくつ読み出されるか、または書き込まれるかは Vivado HLS に示されません。

```
void foo ( volatile int *d_o,  volatile int *d_i)
```

配列の最大サイズなど、インターフェイスに値がいくつ必要かを示すようなものがない限り、Vivado HLS では 1 つの値であると想定され、1 つの入力および 1 つの出力用のシミュレーション ラッパーが作成されます。

RTL ポートが実際には複数の値を読み出しました書き込みする場合、これにより RTL cosim_design シミュレーションが停止する可能性があります。ラッパーは RTL デザインに接続される送信ブロックと受信ブロックがモデリングされるため、RTL デザインで次の値の読み出しました書き込みが試みられますが、読み出す値がないか書き込むスペースがないため、ハンドシェイクインターフェイスによりデザインに待機するよう指示されます。

インターフェイスでマルチアクセスポインターが使用される場合、インターフェイスでの読み出しました書き込みの最大回数を Vivado HLS に示す必要があります。インターフェイスを指定する場合、[Vivado HLS Directive Editor] ダイアログ ボックスの [Directive] で [INTERFACE] を選択し、[depth] オプションを設定します (図 2-33)。

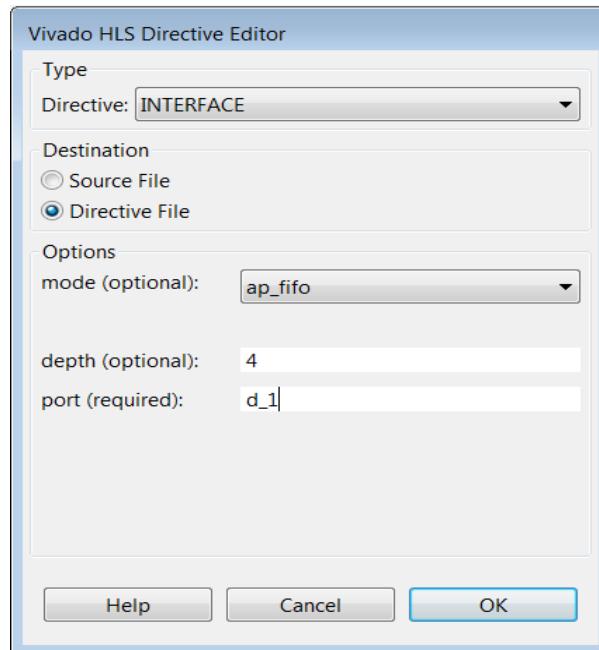


図 2-31 : [Vivado HLS Directive Editor] ダイアログ ボックス :[depth] オプション

上記の例では、引数/ポート `d_i` の (FIFO インターフェイスの) 深さが 4 に設定されており、cosim_design で RTL を正しく検証するのに十分な値が供給されます。

シミュレーションの不一致と C モデリング

ポートの最大深さが定義され、cosim_design で関数から作成された RTL を検証できるようになったら、次の問題はシミュレーションの不一致です。

RTL 入力インターフェイスでは、テストベンチのように、各トランザクションで同じ値が 4 回供給されるわけではありません。RT レベルでは、デザインが最終的に接続される RTL 送信ブロックをモデリングするラッパーにより、ハンドシェイクインターフェイスで要求されるたびに新しい値が供給されます。この例では、最初のトランザクションで 4 つの値が供給されます。テストベンチで供給される 0 と、テストベンチで RTL ラッパーに供給されるのは 1 つの値のみであるため、未定義の値 `x`、`y`、および `z` です。

C と RTL の間でシミュレーションが一致しません。

注記 : 問題は、C コードおよびテストベンチで複数の読み出しました書き込みが必要な状況を正しくモデリングできないことです。

この例の冒頭で、テストベンチからの同じ値を 4 回読み出して関数を検証しました。これは、RT レベルではデータはアップデートされるが何とかうまくいくという想定の下に実行されました、それは間違います。

この例は非現実的なものではなく、C 関数をハードウェアに合成する場合によく見られます。コードを記述して合成しても、意図した動作が得られず、RTL のデバッグに時間を費やす結果になることがあります。

注記 : C コードを合成する前に、必ず C シミュレーションで検証してください。

このコーディングスタイルの制限は、コードを記述しなおすことで克服できます。

次の例は、テストベンチから 4 つの異なる値が読み出されるようにアップデートしたコードです。これは、テストベンチで明示的に定義されている値にアクセスすることにより達成しています。ポインターのアクセスはシーケンシャルでロケーション 0 から開始するので、合成ではストリーミング インターフェイスが使用されます。

```
#include "foo.h"

void foo ( volatile int *d_o,  volatile int *d_i) {
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *(d_i+1);
    *d_o = acc;
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;

}
```

関数が各トランザクションで 4 つの固有の値を読み出すことをモデリングするように、テストベンチがアップデートされます。このテストベンチは、わかりやすく短くするため、1 つのトランザクションのみをモデリングし、セルフチェックコードは省かれています。

```
#include "foo.h"

int main () {
    int d_o[4], d_i[4];
    int i;

    for (i=0;i<4;i++) {
        d_i[i]=i;
    }

    foo(d_o,d_i);

    printf("Di Do\n");
    for (i=0;i<4;i++) {
        if (i<2)
            printf("%d %d\n", d_i[i], d_o[i]);
        else
            printf("%d\n", d_i[i]);
    }

    return 0;
}
```

このテストベンチにより、次のような結果が得られます。1 つのトランザクションから 2 つの出力が生成され、1 つの出力は最初の 2 つの入力読み出しを加算したもの、2 つ目の出力は次の 2 つの入力読み出しと最初の出力を加算したものになります。

```
Di Do
0 1
1 6
2
3
```

このデザインを合成し、ポートインターフェイスのポート `d_i` の深さを 4、ポート `d_o` の深さを 2 に設定して `cosim_design` で RTL を検証できますが、

このアップデートされた例には制限があります。関数が呼び出されるたびに外部配列の位置 0 (テストベンチの `d_i[0]`) からデータの読み出しが開始します。この関数では、関数が呼び出されたときにすべてのデータが準備されている必要があります。

この制限を回避するには、ストリーミングを使用できます。AP_STREAM は Vivado HLS により提供されているコードコンストラクトで、ストリーミングデータおよびストリーミングデータを使用するアプリケーションをより簡単に C で記述できるようにします。

この例で示すように、ストリーミングデータは C では簡単にモデリングできませんがハードウェアではよく使用されるので(ビデオ、通信など)、C でハードウェアを記述するには AP_STREAM を使用するのが最もわかりやすい方法です。これについては、「ストリーミングでのコード記述」で説明しています。

インターフェイスの管理

Cベースデザインでは、すべての入力および出力操作が、0 時間で、正式な関数引数により実行されます。RTL デザインでは、同じ入力および出力操作をデザインインターフェイスのポートを介して特定の I/O(入出力)プロトコルを使用して実行する必要があります。

Vivado HLS では、使用する I/O プロトコルを指定するのに 2 つの方法がサポートされます。

- インターフェイス合成：効率的で安全な標準インターフェイスに基づいてポートインターフェイスが自動的に作成されます。
- 手動インターフェイス指定：インターフェイスの動作を入力ソースコードで明示的に指定します。この方法では、任意の I/O プロトコルを使用できるので、関数を任意のハードウェアリソースとインターフェイスさせることができます。
 - SystemC デザインでは、I/O 制御信号がインターフェイス宣言で指定され、その動作がコードで指定されるので、この方法を使用するのが一般的です。
 - Vivado HLS では、このインターフェイス指定方法を C および C++ デザインでもサポートしています。これについては、この章の後の方で説明します。

インターフェイス合成

C プログラムを RTL デザインに合成すると、C 引数は RTL データポートに合成されます。インターフェイス合成では、インターフェイスプロトコルを自動的に RTL データポートに自動的に追加できます。インターフェイスプロトコルは、出力の準備ができたことを示す出力 Valid 信号などの単純なものから、BRAM に対して読み出しちゃは書き込みを実行するために必要なすべてのポートを含めることができます。

インターフェイス合成で作成可能なインターフェイスのタイプは、C 引数によって異なります。たとえば、インターフェイス合成で出力 Valid 信号を作成する場合、値渡しスカラーは入力にしかできないので、C 引数はポインターまたは C++ 参照にする必要があります。図 2-32 に、各 C 関数引数でサポートされるインターフェイスのタイプを示します。

注記：通常 SystemC デザインではインターフェイス合成はサポートされません。

SystemC デザインでは、I/O プロトコル信号はインターフェイス宣言で宣言され、その動作がコードで完全に指定さ

れます。

ただし、「SystemC のインターフェイス合成」に説明するメモリインターフェイスでは例外です。

ポートにインターフェイスタイプを指定しない場合、デフォルトのインターフェイスでインプリメントされます(図 2-32)。サポートされないインターフェイスタイプを指定した場合、Vivado HLS で警告メッセージが表示され、デフォルトインターフェイスタイプが使用されます(図 2-32)。

Argument	Variable			Pointer Variable			Array			Reference Variable		
	Type			Pass-by-value			Pass-by-reference			Pass-by-reference		
Interface Type	I ¹	IO ²	O ²	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld ³					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none ⁴												
ap_ctrl_hs ⁴			D									

Key:

I : input	Supported Interface
IO : inout	
O : output	Unsupported Interface
D : Default Interface	

図 2-32 : データ型とインターフェイス合成のサポート

図 2-32 の注記は、次のとおりです。

1. 入力および出力の概念は、C 関数と RTL ブロックでは若干異なります。インターフェイス合成を説明するため、ここでは次の規則が使用されます。
 - RTL 入力ポートのように、読み出しが実行されるが書き込みは実行されない関数引数は、入力(I)と示します。
 - RTL 入出力ポートのように、読み出しと書き込みの両方が実行される関数引数は、入出力(IO)と示します。
 - RTL 出力ポートのように、書き込みが実行されるが読み出しが実行されない関数引数は、出力(O)と示します。
2. 標準の値渡し引数は、呼び出し関数に値を出力するのに使用できません。このような引数の値は、関数の return 文で返す(または関数から出力する)ことのみが可能です。

- RTL 出力ポートのように、書き込みが実行されるが読み出しは実行されない値渡し関数引数は、ファンアウトのない RTL 入力ポートに合成されます。
- RTL 入出力ポートのように、書き込みと読み出しの両方が実行される値渡し関数引数は、RTL 入力ポートのみに合成されます。
- インターフェイス タイプ `ap_ovld` は、出力ポートのみに使用可能です。
- インターフェイス タイプ `ap_ctrl_none` および `ap_ctrl_hs` は、関数レベルのインターフェイス プロトコルの合成を制御するために使用します。これらのインターフェイス タイプは、関数自体に指定します。ほかのインターフェイス タイプは、関数引数に指定します。

インターフェイス タイプ

このセクションでは、Vivado HLS でサポートされるインターフェイス タイプについて詳細に説明します。インターフェイスの指定方法の詳細は、このセクションの後に説明します。まず、各インターフェイスについて説明します。

インターフェイス合成には、C 関数引数に実行されるインターフェイス合成と、関数またはブロック レベルで実行されるインターフェイス合成があります。

ブロック レベルのインターフェイス合成では、ブロック全体に I/O プロトコルが適用され、ブロックによる操作の開始を制御する制御信号と、新しいデータの準備ができ操作が完了したことを示す信号が追加されます。ブロック レベルの合成は、インターフェイス タイプ `ap_ctrl_hs` および `ap_ctrl_none` により制御されます。これらのインターフェイス タイプは、関数または関数リターン ポートに適用されます。

標準のポート レベル インターフェイス合成は、関数引数に適切なインターフェイス タイプを適用することにより指定します。RTL 入出力ポートのように、読み出しと書き込みの両方が実行される関数引数は、次のように合成されます。

- インターフェイス タイプ `ap_none`、`ap_stable`、`ap_ack`、`ap_vld`、`ap_ovld`、および `ap_hs` の場合：個別の入力ポートおよび出力ポートとして合成されます。たとえば、関数引数 `arg1` で読み出しと書き込みの両方が実行される場合、RTL 入力データ ポート `arg1_i` と出力データ ポート `arg1_o` として合成され、指定のまたはデフォルトの I/O プロトコルが各ポートに個別に適用されます。
- インターフェイス タイプ `ap_memory` および `ap_bus` の場合：1 つのインターフェイスが作成されます。これらの RTL インターフェイスでは、読み出しと書き込みがサポートされます。
- インターフェイス タイプ `ap_fifo` の場合：読み出しおよび書き込みはサポートされません。

インターフェイス上の構造体はフラット化され、ポートとしてインプリメントする前に階層が削除されます。構造体階層の最初の引数はポートの LSB にインプリメントされ、最後の引数はポートの MSB にインプリメントされます。構造体内の配列のインプリメンテーションは、構造体が値渡し引数であるかポインター引数であるかによって異なります。

- 値渡し構造体の場合、配列は完全にスカラーに分割され、すべての要素がインライン化されて、シングル ワイド バスが作成されます。
- ポインター構造体では、配列ポートが保持され、ほかの配列と同様にインプリメントできます(下の説明を参照)。

`cosim_design` を使用してデザインを検証する場合は、次の条件を満たしている必要があります。

- `ap_ctrl_hs` で指定されているように、デザインでブロック レベルのハンドシェイクが使用されている。
- 各出力ポートには、書き込み操作がいつ実行されたかが示されるインターフェイス タイプ `ap_vld`、`ap_ovld`、`ap_hs`、`ap_memory`、`ap_fifo`、または `ap_bus` を使用する必要があります。

デフォルトのインターフェイス タイプを使用した場合、デザインは `cosim_design` 機能で検証できます。SystemC デザインではインターフェイス合成が使用され、SystemC デザインを `cosim_design` で検証するための上記のような要件はありません。

次の説明では、送信ブロックは現在のブロック入力にデータを供給する RTL ブロックで、受信ブロックは現在のブロックの出力データを使用する RTL ブロックです。

ap_ctrl_none および ap_ctrl_hs

インターフェイス タイプ ap_ctrl_none および ap_ctrl_hs は、RTL をブロック レベル ハンドシェイク信号と共にインプリメンテーションするかどうかを指定します。ブロック レベルのハンドシェイク信号は、デザインで標準の操作を開始できる状態になったことと、操作が終了したこととを示します。これらのインターフェイス タイプは、関数または関数リターンに指定します。

図 2-33 に、ap_ctrl_hs を関数に指定した場合に生成される RTL ポートと動作を示します。これがデフォルトです。この例では、関数で return 文を使用して値が返されるので、RTL デザインに出力ポート ap_return が作成されます。関数に return 文がなければ、このポートは作成されません。

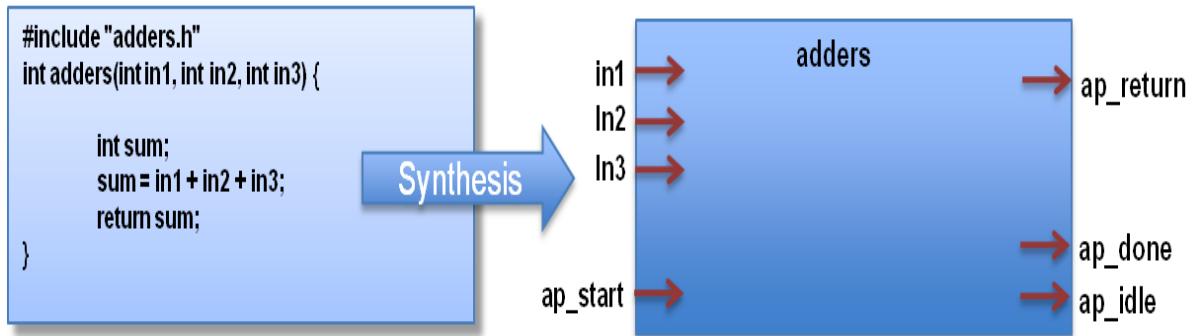


図 2-33 : ap_ctrl_hs インターフェイスの例

ap_ctrl_none を指定すると、図 2-33 に示すハンドシェイク信号ポート (ap_start、ap_idle、および ap_done) は作成されず、ブロックを cosim_design 機能で検証することはできません。

図 2-34 に、インターフェイス タイプ ap_ctrl_hs で作成されたブロック レベルのハンドシェイク信号の動作を示し、その後にそれらの動作を解説します。

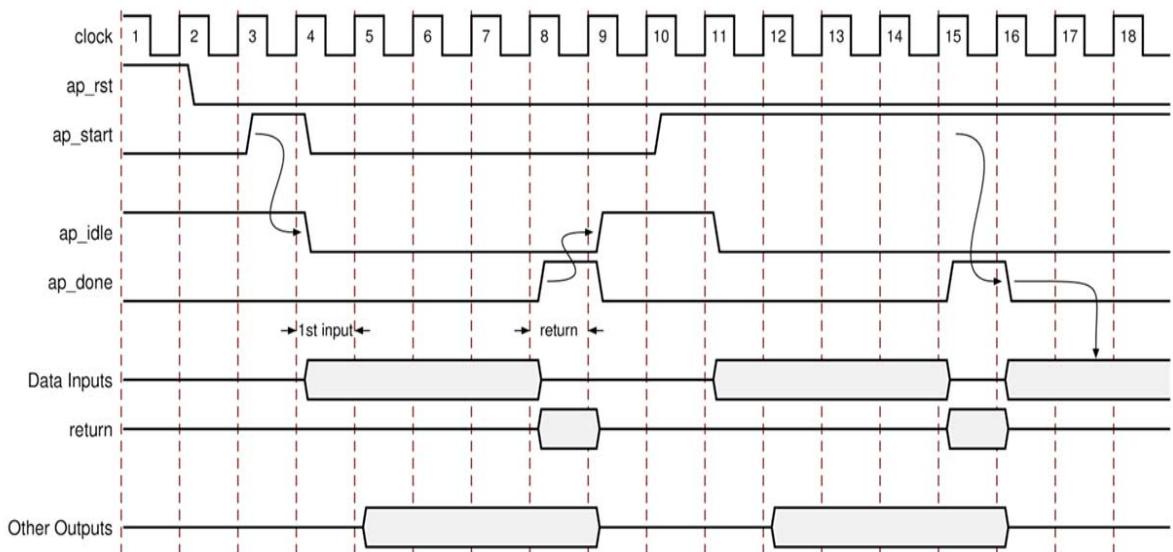


図 2-34 : ap_ctrl_hs インターフェイスの動作

リセット後、動作は次のようにになります。

- ap_start が High になるとブロックが操作を開始します。
- ap_start 出力が High になると、ap_idle が Low になります。
- これで、入力ポートからデータを読み出せるようになります。
 - 最初の入力データは、ap_idle が Low になった後の最初のクロック エッジでサンプリングされます。
- ブロックですべての操作が完了すると、戻り値が ap_return ポートに書き込まれます。
 - 関数リターンがない場合は、RTL ブロックに no ap_return ポートはありません。
 - その他の出力は、ブロックが完了するまで任意に書き出され、この I/O プロトコルからは独立しています。
- ブロックの操作が完了すると、ap_done 出力が High になります。
 - ap_return ポートがある場合、このポートの値は ap_done が High なったときに有効になります。
 - ap_done 信号は、関数の戻り値 (ap_return ポートの出力) が有効であることを示すために使用できます。
- ap_idle 信号は、ap_done が High になった 1 サイクル後に High になり、次に ap_start が High になり、ブロックが操作を開始することが示されるまで、High のままになります。

ap_done が High になったときに ap_start 信号が High の場合は、次のようにになります。

- ap_idle 信号は Low のままになります。
- ブロックは次の実行 (次のトランザクション) を即座に開始します。
- 次の入力は、次のクロック エッジで読み出されます。

Vivado HLS ではパイプライン処理がサポートされており、現在のトランザクションが終了する前に次のトランザクションを開始できます。この場合、ブロックで最初のトランザクションが完了する前に新しい入力を受信でき、ap_done は High になります。

関数がパイプライン処理されている場合、または最上位ループが -rewind オプションを使用してパイプライン処理されている場合、ap_ready 出力ポートが作成され、新しい入力が適用できることを示します。図 2-35 に、パイプライン処理されている場合のブロック レベル インターフェイスの動作を示します。

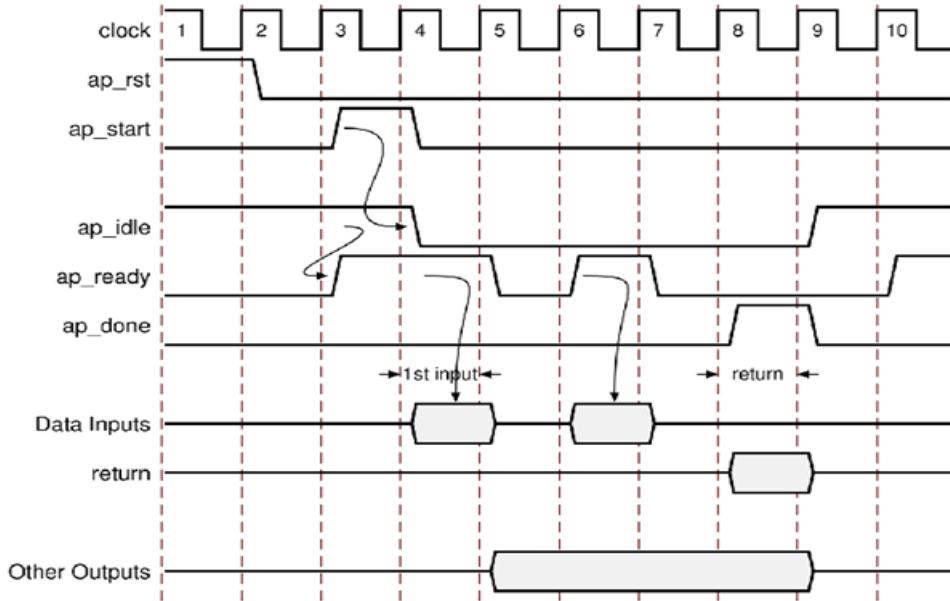


図 2-35: パイプライン処理された ap_ctrl_hs インターフェイスの動作

リセット後、動作は次のようにになります。

- ap_idle が High の場合、ap_ready 信号が High になります。
- ap_start が High になるとブロックが操作を開始します。
- ap_start 出力が High になると、ap_idle が Low になります。
- ap_ready が High になると、入力ポートからデータが読み出されます。

注記 : ap_idle が Low になる前に ap_ready が High になる可能性があるので、送信ブロックでこれら 2 つの信号両方を使用して新しいデータを適用する必要があります。

残りの動作は、図 2-34 で説明したのと同じです。

注記 : ap_done が High になったときに確実に有効なデータは、ap_return ポートの値です。このポートは、関数で return 文を使用して値を返す場合にのみ存在します。

トランザクションが終了して ap_done が High になったときに、デザインのその他の出力も有効である可能性がありますが、必ずしも有効であるとは限りません。出力ポートに対応する Valid 信号が必要な場合は、この後説明するポートレベルの I/O プロトコルで指定する必要があります。

ap_none

ap_none インターフェイス タイプは最も単純なインターフェイスで、ほかの信号はありません。入力信号にも出力信号にも、データの読み出しありまたは書き込みがいつ実行されているかを示す制御ポートはありません。RTL デザインに含まれるポートは、ソースコードで指定されているものだけです。

ap_none インターフェイスでは追加のハードウェアオーバーヘッドはありませんが、送信ブロックで適切な時間に入力ポートにデータを供給し、トランザクション中(デザインが完了するまで)保持する必要があり、受信ブロックで適切な時間に出力ポートを読み出す必要があります。そのため、インターフェイス タイプ ap_none が出力に指定されたデザインは、cosim_design 機能を使用して自動的に検証することはできません。

ap_none インターフェイスは、図 2-32 に示すように、配列引数では使用できません。

ap_stable

ap_stable インターフェイス タイプは、ap_none と同様に、デザインにインターフェイス制御ポートは追加されません。ap_stable タイプは、ポートが通常の操作中は安定しているが、最適化される可能性のある低数値ではなく、ポートにレジスタを付ける必要がないことを指定します。

ap_stable タイプは、通常コンフィギュレーションデータを供給するポートに使用されます。コンフィギュレーションデータは変化する必要がありますが、通常の操作では変化しません。コンフィギュレーションデータは、通常リセット中またはリセット前にのみ変化します。

ap_stable タイプは、入力ポートのみに適用できます。入出力ポートに適用すると、ポートの入力部分のみが安定していると想定されます。

ap_hs (ap_ack、ap_vld、および ap_ovld)

ap_hs インターフェイスでは、データが使用されたことを示す ACK 信号とデータが読み出されたことを示す Valid 信号が使用されます。このインターフェイスは、ap_ack、ap_vld、および ap_ovld のスーパーセットです。

- インターフェイス タイプ ap_ack では、ACK 信号のみが提供されます。
- インターフェイス タイプ ap_vld では、Valid 信号のみが提供されます。
- インターフェイス タイプ ap_ovld では Valid 信号のみが提供され、出力ポートまたは入出力ポートの出力部分のみに適用されます。

図 2-36 に、入力ポートと出力ポートで ap_hs インターフェイスがどのように動作するかを示します。この例では、入力ポート名は in、出力ポート名は out です。制御信号には、元のポート名に基づいて自動的に名前が付けられます。たとえば、入力ポート in の Valid ポートは in_vld です。

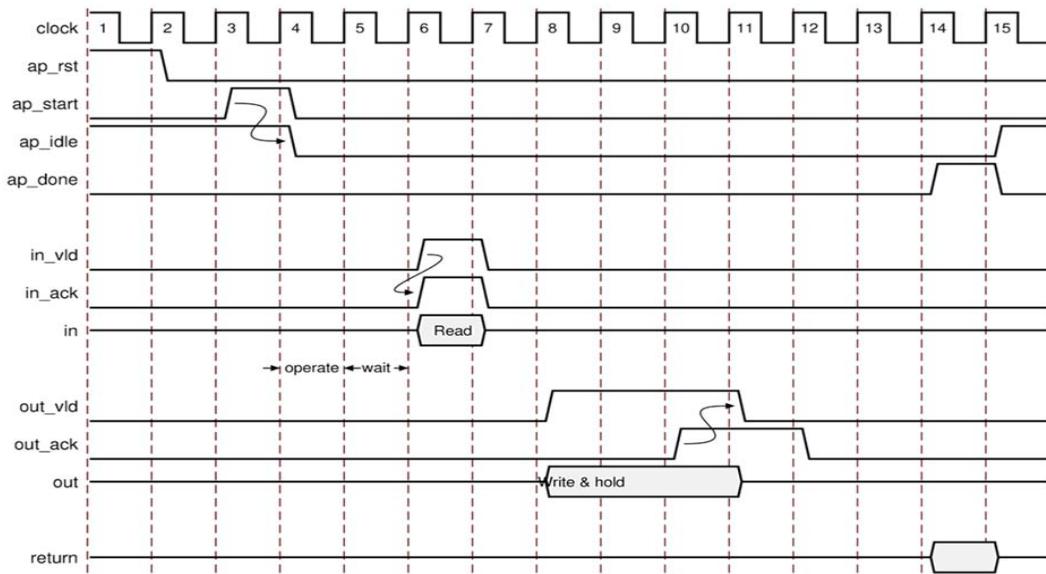


図 2-36 : ap_hs インターフェイスの動作

入力は、次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。

- 入力ポートが読み出されるときに `in_vld` 入力が `Low` の場合、デザインは停止し、`in_vld` 入力が `High` になって新しい入力値が有効であることが示されるまで待機します。
- `in_vld` 入力が `High` になるとすぐに、`in_ack` 出力が `High` になり、データが読み出されたことが示されます。

出力は、次のようにになります。

- リセット後 `ap_start` が `High` になると、ブロックが通常の操作を開始します。
- 出力ポートへの書き込みが実行されると、同時に `out_vld` 出力信号が `High` になり、ポートに有効なデータが存在することが示されます。
- `out_ack` 入力が `Low` の場合、デザインが停止し、`out_ack` 入力が `High` になるまで待機します。
- `out_ack` 入力が `High` になると、次のクロック エッジで `out_vld` 出力が `Low` になります。

`ap_hs` インターフェイスを使用するデザインは `cosim_design` で検証できるので、柔軟な開発プロセスが可能となり、ボトムアップおよびトップダウンの設計フローの両方が使用できます。ブロック間の通信は双方のハンドシェイクにより安全に実行でき、正しい操作のために手動の操作や前提は必要ありません。

`ap_hs` インターフェイスは安全なインターフェイスプロトコルですが、2つのポートと関連の制御ロジックが必要となります。

`ap_ack` インターフェイスでは、ACK ポートのみが合成されます。

- 入力引数に指定すると、出力 ACK ポートが生成され、入力が読み出されるサイクルで `High` になります。
- 出力引数に指定すると、入力 ACK ポートとなります。
 - 書き込み操作後、デザインは停止し、ACK 入力が `High` になって出力が受信ブロックにより読み出されたことが示されるまで待機します。
 - ただし、データが使用可能であることを示す出力ポートはありません。

出力ポートで `ap_ack` インターフェイスタイプを指定する場合は、注意が必要です。出力ポートに `ap_ack` を使用するデザインは、`cosim_design` で検証できません。

インターフェイスタイプ `ap_vld` を指定すると、RTL デザインに関連の Valid ポートが追加されます。

- 出力引数に指定すると、出力 Valid ポートが生成され、出力ポートのデータが有効になったことを示します。
- 注記：入力引数に指定した場合、この Valid ポートは `ap_hs` でインプリメントされる Valid ポートとは動作が異なります。
- `ap_vld` を入力ポートに使用すると（出力 ACK 信号はなし）、Valid がアクティブになると同時に入力ポートが読み出されます。これはデザインでポートを読み出す準備ができていない場合も同様で、その場合はデータポートがサンプリングされ、必要になるまで内部で保持されます。
 - Valid 入力がアクティブな各サイクルで、入力データが読み出されます。

`ap_ovld` インターフェイスタイプは `ap_vld` と同じですが、出力ポートのみに指定できます。これは、読み出しと書き込みの両方が実行されるポインターを、Valid 出力ポートのみを付けてインプリメントし、入力側はデフォルトの `ap_none` でインプリメントする場合に有益です。

ap_memory

配列引数は、通常 `ap_memory` インターフェイスを使用してインプリメントされます。このポートインターフェイスは、インプリメンテーションでメモリのアドレスロケーションにランダムアクセスが必要な場合に、メモリ エレメント（RAM、ROM）と通信するために使用されます。配列引数は、ランダム アクセス メモリインターフェイスをサポートする唯一の引数です。

メモリ エレメントへのシーケンシャルアクセスのみが必要な場合は、次に説明する `ap_fifo` インターフェイスを使用して、ハードウェアのオーバーヘッドを削減できます。`ap_fifo` インターフェイスでは、アドレス生成は実行されません。

ap_memory インターフェイスを使用する場合、「メモリ リソースの選択」に示すように、配列ターゲットを set_directive_resource コマンドを使用して指定する必要があります。配列にターゲットが指定されていない場合、Vivado HLS により自動的にシングル ポートまたはデュアル ポート RAM インターフェイスを使用するかが判断されます。



ヒント: 合成の前に、RESOURCE 指示子を使用して、配列引数のターゲットとして正しいメモリ タイプを指定してください。新しい正しいメモリで再合成すると、スケジューリングおよび RTL が異なるものになる可能性があります。

図 2-37 に、d という配列のターゲットがシングル ポート BRAM として指定されたリソースである例を示します。

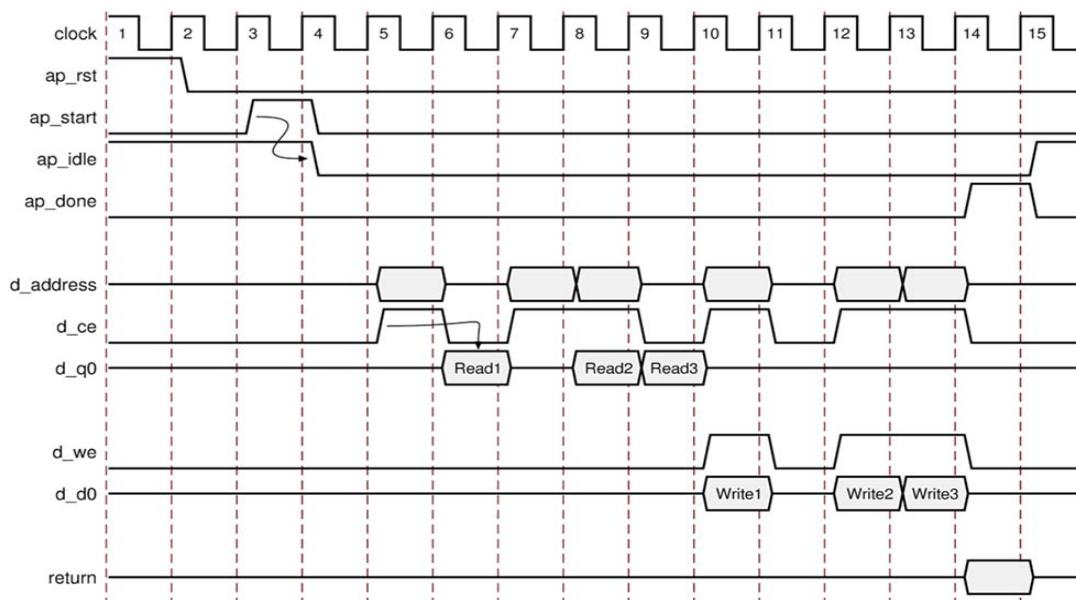


図 2-37 : ap_memory インターフェイスの動作

リセット後、動作は次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。
- 読み出しは、出力アドレス ポートにアドレスを供給し、出力信号 d_ce をアサートすると実行されます。この BRAM ターゲットでは、入力データが次のクロック サイクルで有効になるとデザインで判断されます。
- 書き込み操作は、出力ポート d_ce と d_we をアサートし、同時にアドレスとデータを供給すると実行されます。

メモリ インターフェイスは外部信号によって停止できず、出力データが有効であるかどうかが示されるので、cosim_design を使用して検証できます。

ap_fifo

メモリ エレメントにアクセスする必要があり、アクセスがシーケンシャルでランダム アクセスが不要な場合は、ap_fifo インターフェイスを使用するのがハードウェアの点では最も効率的です。ap_fifo インターフェイスでは、ポートを FIFO に接続でき、完全な双方向の Empty/Full の通信がサポートされ、配列、ポインター、および参照渡し引数で指定できます。

ap_fifo インターフェイスを使用可能な関数では、ポインターがよく使用され、同じ変数に複数回アクセスする可能性があります。この場合、「マルチアクセス ポインター インターフェイス」を参照して volatile 修飾子の重要性を理解してください。

注記 : ap_fifo インターフェイスでは、すべての読み出しおよび書き込みがシーケンシャルであると想定されます。

Vivado HLS でそうではないと判断された場合、エラー メッセージが表示されて処理が停止します。

Vivado HLS でアクセスが常にシーケンシャルかどうかを判断できない場合は、警告メッセージが表示され、処理が継続されます。

次の例では、in1 は現在のアドレスにアクセスし、その後現在のアドレスの上 2 つのアドレスにアクセスして、最後に 1 つ下のアドレスにアクセスするポインターです。

```
void foo(int* in1, ...){
    int data1, data2, data3;
    ...
    data1= *in1;
    data2= *(in1+2);
    data3= *(in1-1);
    ...
}
```

in1 を ap_fifo インターフェイスとして指定すると、Vivado HLS でアクセスがチェックされ、アクセスがシーケンシャルでない場合はエラー メッセージが表示され、停止します。シーケンシャルでないアドレス ロケーションから読み出すには、ap_memory インターフェイスをランダム アクセスとして使用するか、ap_bus インターフェイスを使用します。

ap_fifo インターフェイスは、読み出しおよび書き込みの両方に使用される引数(入出力ポート)には指定できません。入力引数または出力引数のみで指定できます。入力引数 in、出力引数 out を ap_fifo インターフェイスとして指定したインターフェイスは、次のように動作します。

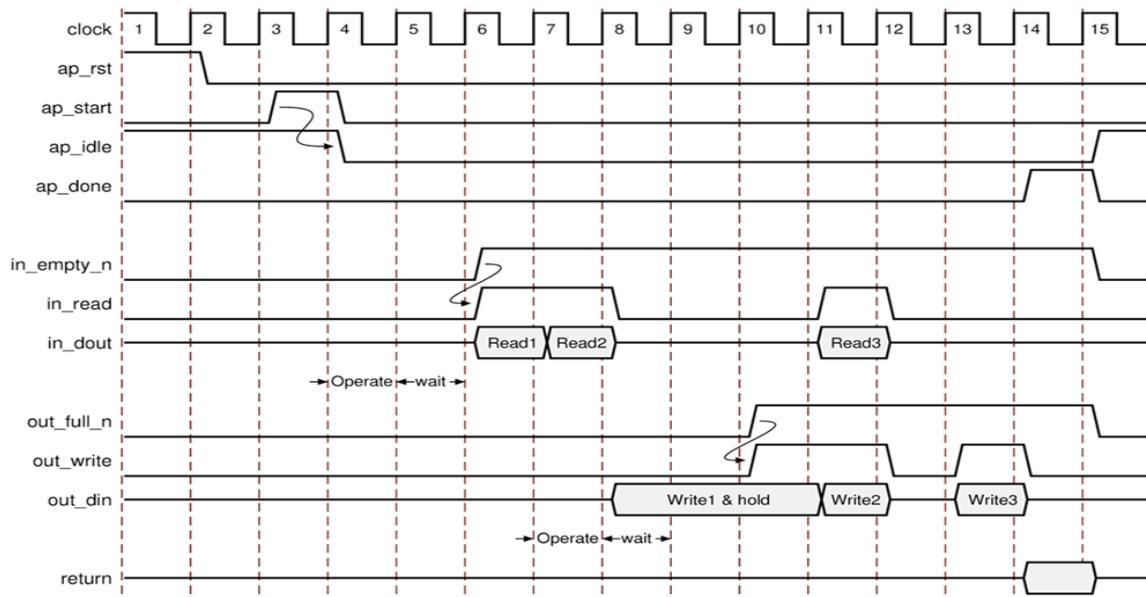


図 2-38 : ap_fifo インターフェイスの動作

リセット後 ap_start が High になると、ブロックが通常の操作を開始します。

読み出しだけでは、次のようになります。

- 入力ポートで読み出しが実行されるときに FIFO が空の場合 (in_empty_n 入力ポートが Low)、デザインは停止し、データを読み出せるようになるまで待機します。
- in_empty_n 入力が High になって FIFO にデータがあることが示されるとすぐに、ACK 信号である in_read 出力が High になり、このサイクルでデータが読み出されたことが示されます。
- ポートで読み出しが必要なときに FIFO にデータがあると、出力 ACK 信号が High になり、このサイクルでデータが読み出されたことが示されます。

出力は、次のようにになります。

- 出力ポートで書き込みが実行されるときに FIFO がフルの場合 (out_full_n 入力ポートが Low)、データは出力ポートに配置されますが、デザインは停止し、FIFO に書き込むスペースができるまで待機します。
- FIFO に書き込むスペースができると (out_full_n 入力が High)、out_write 出力が High になり、出力データが有効であることが示されます。
- 出力を書き込む必要があるときに FIFO にスペースがあると、出力 Valid 信号が High になり、このサイクルでデータが有効であることが示されます。

ap_fifo インターフェイスでは、出力データ ポートに関連の書き込み信号ポートがあるので、cosim_design を使用して検証できます。

ap_bus

ap_bus インターフェイスを使用すると、バス ブリッジと通信できます。このインターフェイスは特定のバス規格には従っていませんが、汎用であるためバス ブリッジと共に使用でき、バス ブリッジでシステムバスをアービトレイションできます。バス ブリッジでは、バースト書き込みをキャッシュが必要です。

ap_bus インターフェイスを使用可能な関数では、ポインターがよく使用され、同じ変数に複数回アクセスする可能性があります。この場合、「マルチアクセスポインター インターフェイス」を参照して volatile 修飾子の重要性を理解してください。

ap_bus インターフェイスは、次の 2 つの方法で使用できます。

- 標準モード：読み出しおよび書き込みを、それぞれにアドレスを指定して個別に実行します。
- バースト モード：C ソース コードで C 関数 memcpy を使用すると、データ転送にバースト モードが使用されます。バースト モードでは、インターフェイスにより転送のベース アドレスとサイズが示され、データ サンプルが連続するサイクルで高速に転送されます。

図 2-39 および図 2-40 に、標準モードでの読み出しと書き込みの動作の例を示します。この例では、ap_bus インターフェイスが引数 d に適用されています。

```
void foo (int *d) {
    static int acc = 0;
    int i;

    for (i=0;i<4;i++) {
        acc += d[i+1];
        d[i] = acc;
    }
}
```

図 2-41 および図 2-42 に、C 関数 memcpy とバースト モードを使用する例を示します。この例のコードは、次のとおりです。

```
void bus (int *d) {
    int buf1[4], buf2[4];
    int i;

    memcpy(buf1,d,4*sizeof(int));
```

```

for (i=0;i<4;i++) {
    buf2[i] = buf1[3-i];
}

memcpy(d,buf2,4*sizeof(int));
}

```

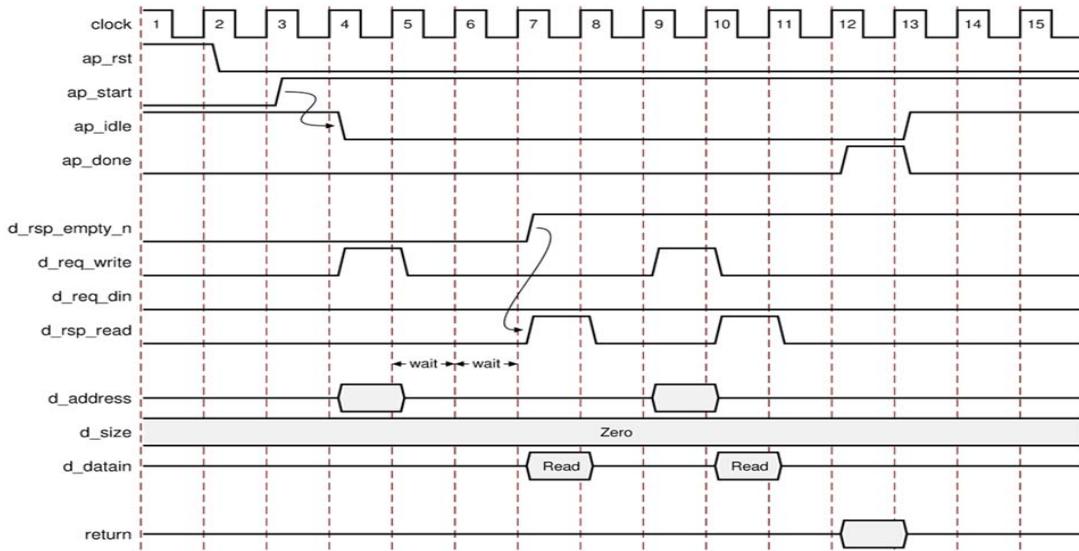


図 2-39 : ap_bus インターフェイスの動作標準読み出し

リセット後、動作は次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。
- 読み出しを実行するときにバス ブリッジ FIFO にデータがない場合は (d_rsp_empty_n が Low)、次のようになります。
 - 出力ポート d_req_write がアサートされ、d_req_din ポートがディアサートされて、読み出し操作が示されます。
 - アドレスが出力です。
 - デザインが停止し、データが読み出せるようになるまで待機します。
- データが読み出せるようになると、出力信号 d_rsp_read がアサートされ、次のクロック エッジでデータが読み出されます。
- 読み出しを実行するときにバス ブリッジ FIFO にデータがある場合は (d_rsp_empty_n が High)、次のようになります。
 - 出力ポート d_req_write がアサートされ、d_req_din ポートがディアサートされて、読み出し操作が示されます。
 - アドレスが出力です。
 - 次のクロック サイクルで d_rsp_read がアサートされ、次のクロック エッジでデータが読み出されます。

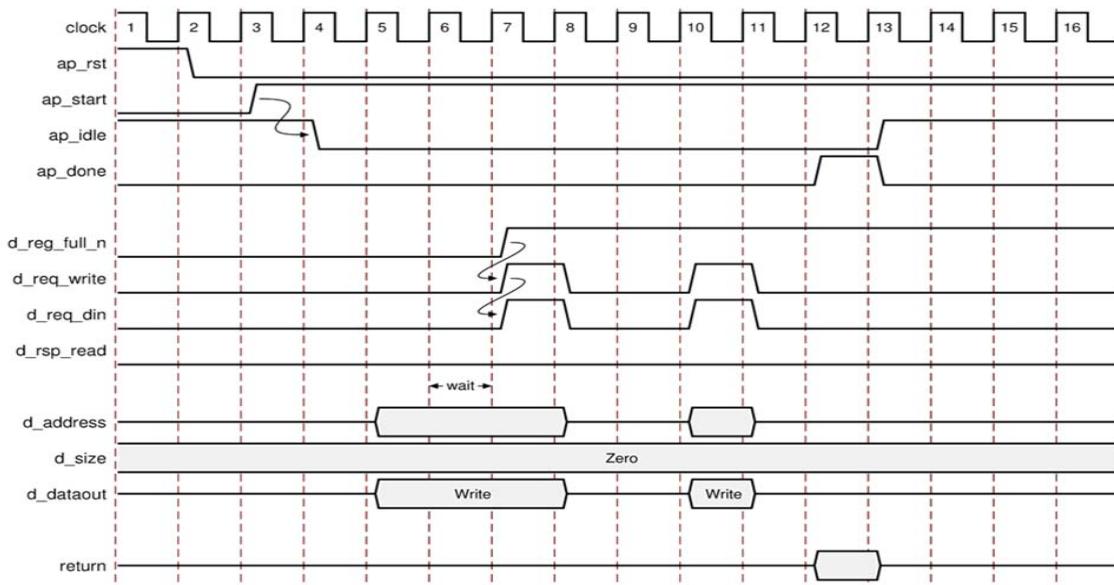


図 2-40 : ap_bus インターフェイスの動作標準書き込み

リセット後、動作は次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。
- 書き込みを実行するときにバス ブリッジ FIFO にスペースがない場合は (d_req_full_n が Low)、次のようになります。
 - アドレスおよびデータが出力です。
 - デザインが停止し、スペースが使用可能になるまで待機します。
- 書き込むスペースができると、次のようになります。
 - 出力ポート d_req_write と d_req_din がアサートされて、書き込み操作が示されます。
 - 出力信号 d_req_din がアサートされ、次のクロック エッジでデータが有効であることが示されます。
- 書き込みを実行するときにバス ブリッジ FIFO にスペースがある場合は (d_req_full_n が High)、次のようになります。
 - 出力ポート d_req_write と d_req_din がアサートされて、書き込み操作が示されます。
 - アドレスおよびデータが出力です。
 - d_req_din がアサートされ、次のクロック エッジでデータが有効であることが示されます。

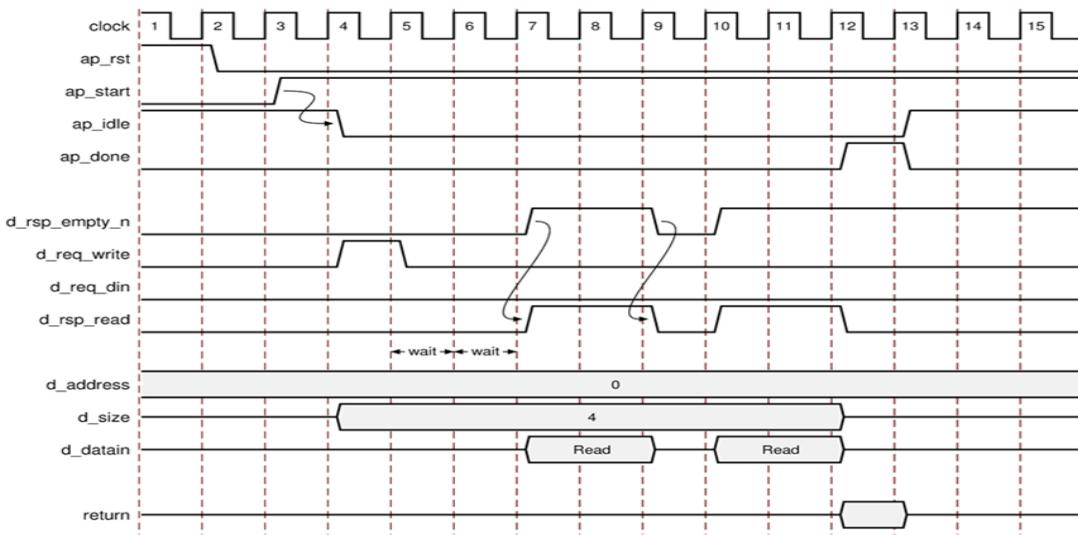


図 2-41 : ap_bus インターフェイスの動作バースト読み出し

リセット後、動作は次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。
- 読み出しを実行するときにバス ブリッジ FIFO にデータがない場合は (d_rsp_empty_n が Low)、次のようになります。
 - 出力ポート d_req_write がアサートされ、d_req_din ポートがディアサートされて、読み出し操作が示されます。
 - 転送のベース アドレスとサイズが出力されます。
 - デザインが停止し、データが読み出せるようになるまで待機します。
- データが読み出せるようになると、出力信号 d_rsp_read がアサートされ、次の N クロック エッジ (N は d_size 出力ポートの値) でデータが読み出されます。
- バス ブリッジ FIFO が途中で空になると、データ転送は即座に停止し、データが読み出されるようになるまで待機して、停止した地点から再開します。
- 読み出しを実行するときにバス ブリッジ FIFO にデータがある場合は、次のようになります。
 - 転送は上記のように開始し、FIFO が途中で空になると、デザインが停止してデータが再び読み出せるようになるまで待機します。

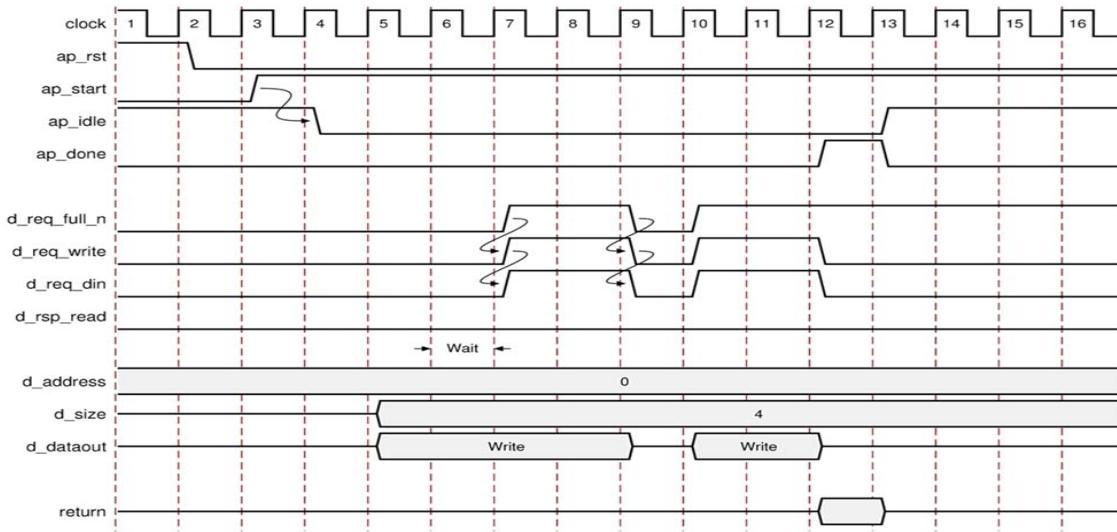


図 2-42 : ap_bus インターフェイスの動作バースト書き込み

リセット後、動作は次のようにになります。

- リセット後 ap_start が High になると、ブロックが通常の操作を開始します。
- 書き込みを実行するときにバス ブリッジ FIFO にスペースがない場合は (d_req_full_n が Low)、次のようになります。
 - ベース アドレス、転送サイズ、およびデータが出力です。
 - デザインが停止し、スペースが使用可能になるまで待機します。
- 書き込むスペースができると、次のようになります。
 - 出力ポート d_req_write と d_req_din がアサートされて、書き込み操作が示されます。
 - 出力信号 d_req_din がアサートされ、次のクロック エッジでデータが有効であることが示されます。
 - FIFO がフルになると即座に d_req_din 出力信号がディアサートされ、スペースができると再びアサートされます。
 - N 個のデータ値が転送されると転送が停止します (N は d_size の値)。
- 書き込みを実行するときにバス ブリッジ FIFO にスペースがある場合は (d_req_full_n が High)、次のようになります。
 - 転送は上記のように開始し、FIFO が途中でフルになると、デザインが停止してデータが再びスペースができるまで待機します。

cosim_design インターフェイスは、autosim 機能で検証できます。

インターフェイス合成の制御

インターフェイス合成は、INTERFACE 指示子を使用するか、コンフィギュレーション設定を使用して制御します。

コンフィギュレーション設定は、RTL ポートおよびインターフェイスを作成する際のデフォルト操作を指定します。INTERFACE 指示子は、特定のポートのインターフェイス タイプを明示的に指定し、デフォルトまたはグローバル設定を変更します。

デフォルトのポート インターフェイス タイプの設定

コンフィギュレーション設定を使用すると、次を実行できます。

- RTL デザインにグローバル クロック イネーブルを追加します。
- すべてのグローバル変数に RTL ポートを作成します。
- 指定のタイプのポートすべてにデフォルトのインターフェイス ポートを設定します。

コンフィギュレーションを設定するには、GUI で [Solution] → [Solution Settings] をクリックし、左側のボックスで [General] を選択して [Add] をクリックし、[Command] ドロップダウン リストから [config_interface] を選択します (図 2-43)。

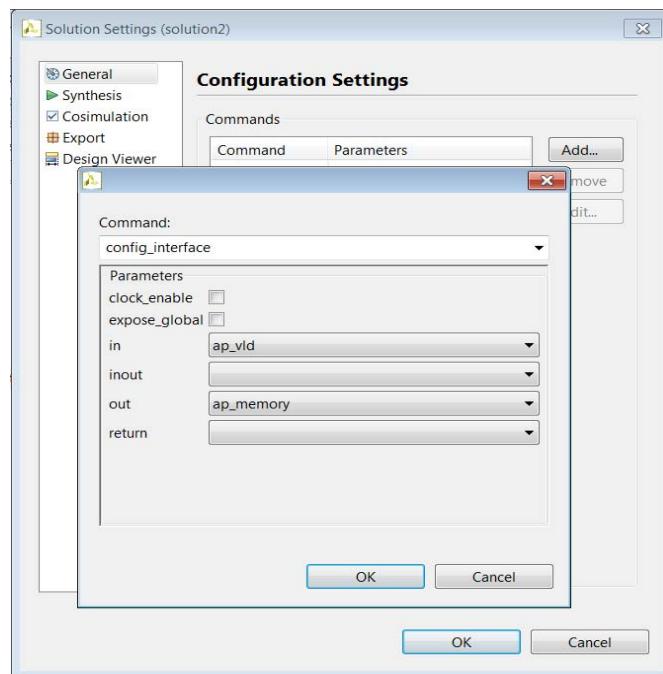


図 2-43: コンフィギュレーション設定

[clock_enable] をオンにすると、RTL デザインにグローバル クロック イネーブル (CE) が追加されます。これにより最上位ポートに ap_ce ポートが追加されます。ap_ce が Low の場合、デザイン内のすべてのレジスタでクロックがオフになります。

どの C 関数でもグローバル変数を使用できます。グローバル変数は、関数外で定義されます。デフォルトでは、グローバル変数により RTL ポートは作成されません。Vivado HLS では、グローバル変数は最終デザイン内にあると想定されます。[expose_global] をオンにすると、グローバル変数が関数/デザイン外にインプリメントされ、RTL ポートが作成されます。

すべてのポートのデフォルト インターフェイス タイプは、図 2-32 に示されていますが、インターフェイス コンフィギュレーションでこれらのデフォルト インターフェイスをユーザーが設定できます。図 2-43 および次の Tcl コマンドは、すべての入力ポートのインターフェイス タイプを ap_vld に、すべての出力ポートのインターフェイス タイプを ap_memory に設定しています。

```
config_interface -mode in ap_vld
config_interface -mode out ap_memory
```

注記: ポートにサポートされていないインターフェイス タイプを指定した場合、コンフィギュレーションは無視され、インターフェイスはそのポートのデフォルト タイプに設定されます。サポートされていないポート タイプとデフォルトのポート タイプは、[図 2-32](#) を参照してください。

たとえば、配列ポートに `ap_ack` などのサポートされていない I/O プロトコルを指定すると、無視されます。

ポートインターフェイス タイプの指定

インターフェイス タイプは、個々のポートに設定できます。GUI の [Directive] タブでポートを右クリックして [Insert Directive] をクリックして、[Vivado HLS Directive Editor] ダイアログ ボックスを開きます ([図 2-44](#))。

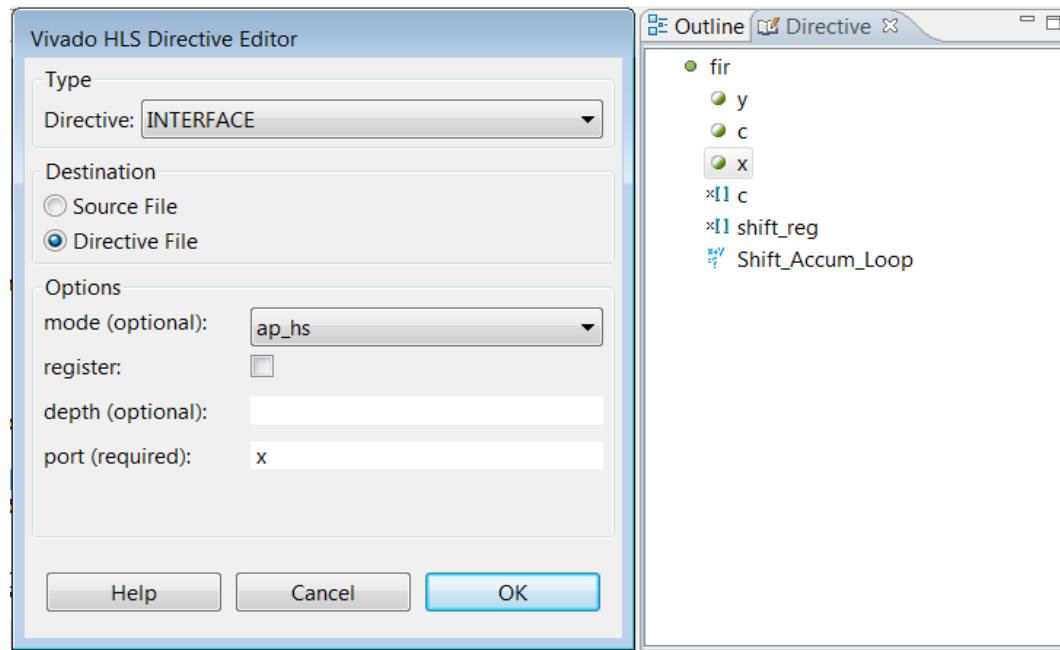


図 2-44: ポートインターフェイスの指定

インターフェイスは Tcl コマンド ファイルでも指定できます。次の例では、[図 2-44](#) と同様に関数 `foo` の `x` ポートを `ap_hs` に設定しています。

```
set_directive_interface -mode ap_hs fir x
```

サポートされていないポートインターフェイスを選択した場合、メッセージが表示され、[図 2-32](#) に示すデフォルトのポート タイプがインプリメントされます。

ポートオプションの指定

INTERFACE 指示子には 2 つのオプションがあります。ポートにレジスタを付けるか付けないかを指定するオプションと、ポートから読み出す値またはポートに書き込む値の数を指定するオプションです。

デフォルトでは、ポートにレジスタは付けられません。データが必要なときに入力ポートが読み出され、レジスタによるエリアの増加が回避されます。レジスタを付けるオプションを選択すると、操作の最初のサイクルの値渡し読み出しすべてにレジスタが付けられます。ポイントアンドクリック読み出しでは読み出しにレジスタが付けられますが、データが必要なサイクルより 1 サイクル早く読み出しが実行されます。出力ポートでは、レジスタを付けるオプションを選択すると、出力に必ずレジスタが付きます。メモリ、FIFO、およびバスインターフェイスでは、レジスタオプションを設定しても変化はありません。

1つのトランザクションでポインタに対して読み出しありは書き込みが複数回実行される場合は、[depth] オプションを使用して読み出しありは書き込みの値の最大数を指定する必要があります。このオプションを正しく指定しないと、cosim_design 機能を使用して RTL を検証したときに検証エラーが発生することがあります。

バスインターフェイスの指定

Vivado HLS では「インターフェイス合成」セクションで説明される標準インターフェイスだけでなく、バスインターフェイスを自動的に RTL デザインに追加することもできます。

バスインターフェイスポートとインターフェイス合成 (ap_none、ap_hs など) で作成された RTL ポートの主な違いは、バスインターフェイスが [Export RTL] プロセス中にデザインに追加される点です。RTL エクスポートプロセスの詳細は、「RTL デザインのエクスポート」セクションを参照してください。

バスインターフェイスには、次のように処理されます。

- バスインターフェイスは合成レポートにはレポートされません。
- バスインターフェイスは、合成後に記述された RTL には含まれません。

次のバスインターフェイスを使用できます。

- AXI4 Lite Slave
- AXI4 Master
- AXI4 Stream
- PLB 4.6 Slave
- PLB 4.6 Master
- FSL
- NPI

バスインターフェイスの別の重要な事項として、使用可能なバスインターフェイスのタイプが RTL ポートのプロトコル (ap_none、ap_hs、ap_bus など) によって異なることがあります。RTLインターフェイスのタイプは、それぞれ特定のバスインターフェイスにしか接続できません。

表 2-6 は、Vivado HLS で作成される RTLインターフェイスポートとそれらに接続できるバスインターフェイスをリストしています。たとえば、AXI4 Stream バスインターフェイスは ap_fifo ポートタイプにしか追加できません。

表 2-6: RTL ポートからバスインターフェイスへのマップ

バスインターフェイスプロトコル							
RTLインターフェイスプロトコル	AXI4 Lite Slave	AXI4 Master	AXI4 Stream	PLB 4.6 Slave	PLB 4.6 Master	FSL	NPI
ap_bus	-	X	-	-	X	-	-
ap_fifo	-	-	X	-	-	X	-
ap_ctrl_ap_vld	X	-	-	X	-	-	-
ap_ack							
ap_hs							
ap_ovld	-	-	-	-	-	-	-
ap_memory							

たとえば、RTL ポートプロトコルのタイプが ap_fifo の場合、これは AXI4 Stream または FSL バスインターフェイスに接続できますが、RTL ポートのタイプが ap_ovld または ap_memory の場合は、どのバスインターフェイスにも接続できません。

- ap_memoryインターフェイスには、インターフェイスは必要なく、直接 EDK のメモリ (BRAM) に接続できます。

- ap_ovld インターフェイスを使用したポートはすべてサポートされるタイプの 1 つ(たとえば、ap_hs)に変更されるか、インターフェイスには接続できないかになります。

バス インターフェイスの追加

バス インターフェイスを既存の RTL インターフェイスに追加するには、RESOURCE 指示子を使用してポートにリソースを指定します(同じプロセスおよび指示子でどのメモリリソース タイプを配列ポートに接続するかを指定できます)。

RESOURCE 指示子はコアを指定しないと使用できません。すべてのバス インターフェイスとそれに対応するコアのリストについては第 3 章「高位合成演算子およびコア ガイド」および表 2-7 を参照してください。

表 2-7: バス インターフェイス

バス インターフェイス	コア	説明
AXI4 Lite スレーブ	AXI4LiteS	AXI4 スレーブのインターフェイス
AXI4 マスター	AXI4M	AXI4 マスターのインターフェイス
AXI4 Stream	AXI4Stream	AXI4 ストリームのインターフェイス
PLB 4.6 スレーブ	PLB46S	標準バス スレーブ インターフェイス
PLB 4.6 マスター	PLB46M	標準バス マスター インターフェイス
FSL	FSL	標準ザイリンクス FSL インターフェイス
NPI	NPI64M	ネイティブ マルチポート メモリ コントローラー インターフェイス

バス インターフェイスを作成するプロセスは、次の 2 つの手順で成り立っています。

- RTL インターフェイスの作成
- バス インターフェイス リソースの指定

次は、ポートの出力 (out) を AXI4 Master バス インターフェイスへ接続する方法を示しています。

RTL インターフェイスの選択

バス インターフェイスを作成するには、まず必要なバス インターフェイスをサポートする RTL ポートを作成します。AXI4 Master バス インターフェイスが必要なので、表 2-7 に示すように、RTL ポートは ap_bus プロトコルを使用してインプリメントする必要があります。

図 2-45 では、関数引数の out が選択され、指示子が右クリックで適用されています。ドロップダウン メニューから [INTERFACE] を選択すると、インターフェイス モードが ap_bus と指定されます。

[Vivado HLS Directive Editor] ダイアログ ボックスのその他のオプションを使用すると、オプションでポートにレジスタを付けたり、RTL 検証用に必要な深さを指定できます(ポートが何度もアクセスされるポインターの場合は、アクセス数をここで指定する必要があります)。

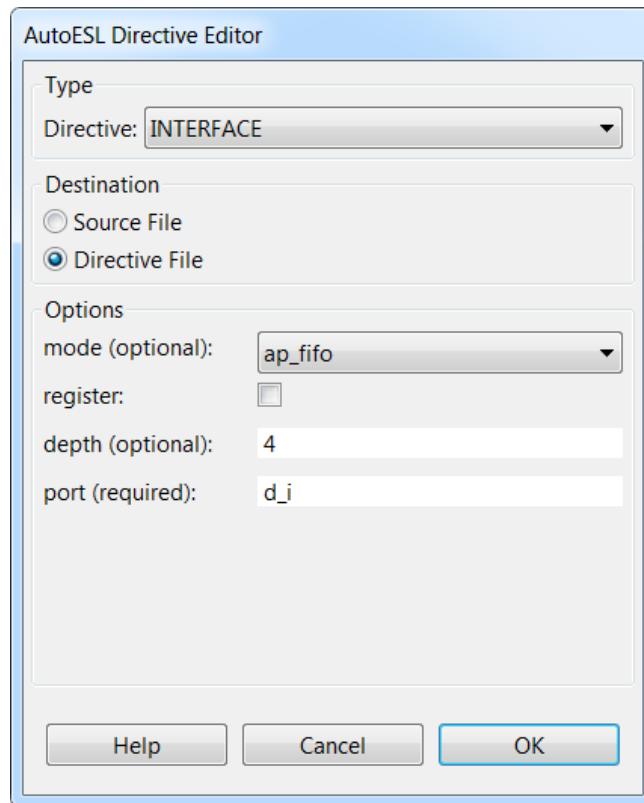


図 2-45 : RTL インターフェイス指示子

バス インターフェイスの選択

AXI4 Master バス インターフェイスを指定するには、図 2-46 のように RESOURCE 指示子を使用して、AXI4M リソースをポート/引数 out へ追加します。

RESOURCE 指示子に関連するオプションを使用すると、特定のバス インターフェイスに複数ポートを含めたり、AXI4 インターフェイスのサイドチャネルを使用したりできます。

- AXI4 Lite Slave または PLB 4.6 Slave インターフェイスを追加すると、port map オプションで複数のポートが 1 つのスレーブ ポートにまとめられるようにできます。このオプションの使用方法は、これらのバス インターフェイスの説明に含まれています。
- metadata オプションは、C 関数で特定の変数を AXI4 インターフェイス規格の特定の信号にマップするために使用します。この例は、「AXI4 Stream インターフェイス」および「AXI4 Master インターフェイス」を参照してください。

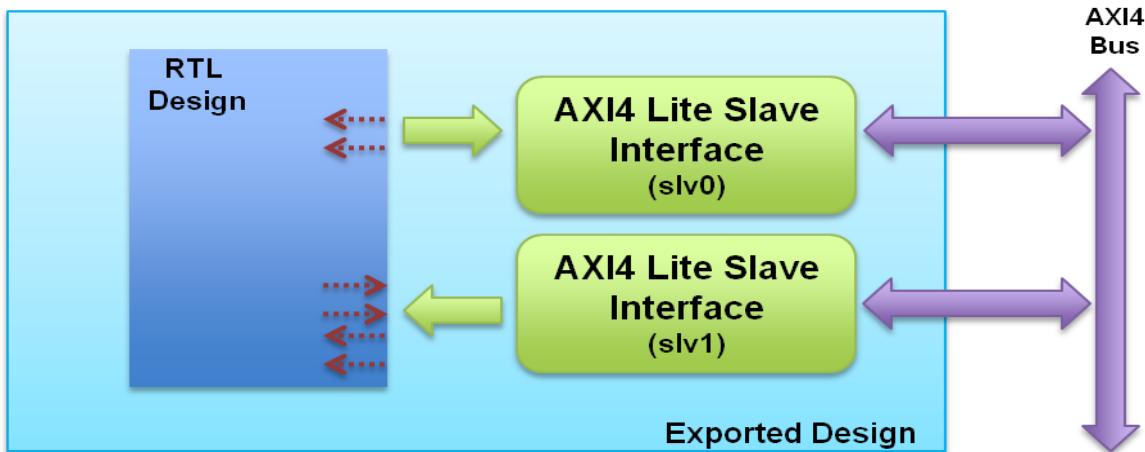


図 2-46: バス インターフェイスのリソース指示子

AXI4 Lite スレーブ インターフェイス

AXI4 のスレーブ インターフェイスは、デザインを CPU やマイクロプロセッサーなどのフォームで制御できるようにするために使用され、スレーブ ポートで使用可能な機能のみを提供します。

- 複数の RTL ポートを同じバス インターフェイスにまとめることができます。
- デザインが EDK 環境用の PCore としてエクスポートされると、出力には C 関数とヘッダー ファイルが含まれ、プロセッサで実行されるコードで使用されます。

次の例は、複数の RTL ポートが共通の AXI4 スレーブ インターフェイスにまとめらるところを示しています。複数の RTL ポートが 1 つのバス インターフェイスを介してアクセスされるようになります。コード例の後には、作成された C ファイルを確認します。

```
int foo_top (int *a, int *b, int *c, int *d) {
    // Define the RTL interfaces
    #pragma AP interface ap_hs port=a
    #pragma AP interface ap_none port=b
    #pragma AP interface ap_vld port=c
    #pragma AP interface ap_ack port=d
    #pragma AP interface ap_ctrl_hs port=return register

    // Define the pc当地 interfaces and group into AXI4 slave "slv0"
    #pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a
    #pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=b

    // Define the pc当地 interfaces and group into AXI4 slave "slv1"
    #pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=return
    #pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=c
    #pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=d

    *a += *b;
    return (*c + *d);
}
```

上記の例でポート a および b は共通の AXI4 Lite スレーブ インターフェイスにまとめられています(図 2-47)。

ブロック レベルの IO プロトコルは、ap_ctrl_hs インターフェイス モードを return 関数に適用すると設定されます。これは、デフォルトですが、この例では明確に指定しています。return ポートにポート c と d をまとめることで、すべてのブロック レベルの IO プロトコル信号(ap_start、ap_done など)が AXI4 Lite スレーブ インターフェイスの slv1 に割り当てられます。

RESOURCE 指示子が GUI を使用して適用される場合は、そのグループを作成するために使用されたバス バンドル ストリング全体(例: "-bus_bundle slv1")を クオーテーションマークも含めて metadata オプション ボックスに入力する必要があります。slv1 のように同じ bus_bundle 名を使用し、各 RTL ポートがまとめられるようにします。

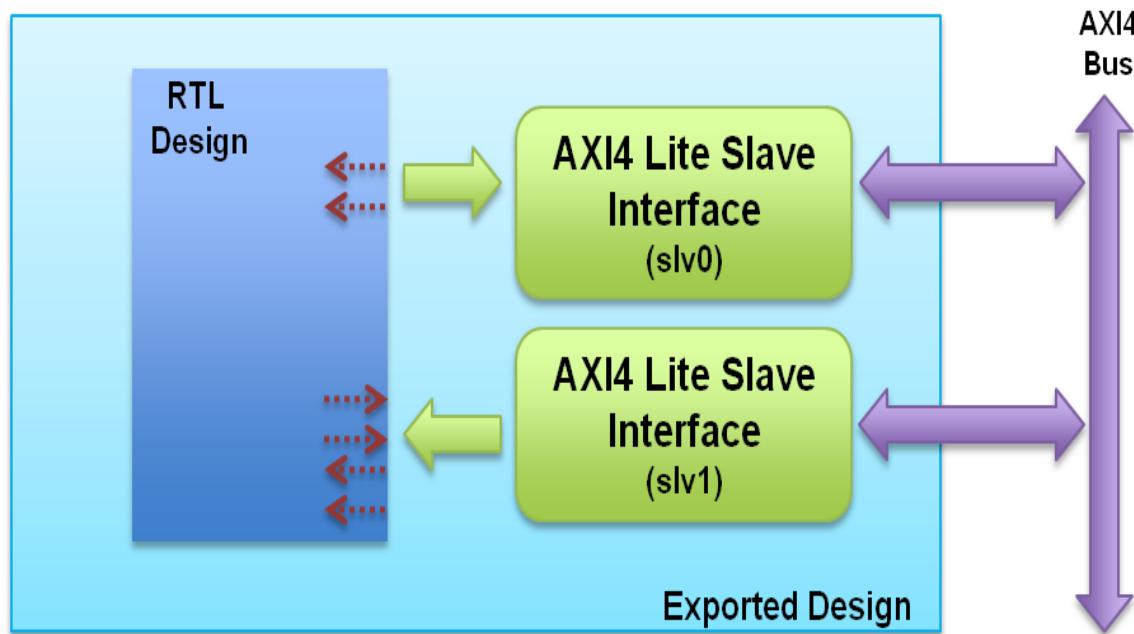


図 2-47 : RTL ポートがまとめられた AXI4 Lite スレーブ インターフェイス

デザインが Pcore としてエクスポートされると、RTL デザインに加えて次の C ファイルが output されます。「RTL デザインのエクスポート」セクションを参照してください(表示されるファイルおよびファイル名はこの例用です)。

- xfoo_top_slv0.h
- xfoo_top_slv1.h
- xfoo_top.h
- xfoo_top.c

xfoo_top_slv0.h ファイルには、slv0 インターフェイスにまとめられた RTL ポートのメモリ マップ ロケーションが含まれます。このファイルのコメントには、RTL ポート データと制御信号がどのように AXI4 バス インターフェイスにマップされたかが示されています。

- ポート a は読み出し/書き込み信号なので、別々の入力および出力ポートとしてインプリメントされます。
- ポート a には、Valid 信号と ACK (acknowledge) 信号が接続されます。
- ポート b は ap_none タイプで、接続される制御信号はありません。

```

// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Control signal of a_i
//           bit 0 - a_i_ap_vld (Read/Write/COH)

```

```

//      bit 1 - a_i_ap_ack (Read)
//      others - reserved
// 0x14 :Data signal of a_i
//      bit 31~0 - a_i[31:0] (Read/Write)
// 0x18 :Control signal of a_o
//      bit 0 - a_o_ap_vld (Read)
//      bit 1 - a_o_ap_ack (Read/Write/COH)
//      others - reserved
// 0x1c :Data signal of a_o
//      bit 31~0 - a_o[31:0] (Read)
// 0x20 : reserved
// 0x24 :Data signal of b
//      bit 31~0 - b[31:0] (Read/Write)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
// Handshake)

#define XFOO_TOP_SLV0_ADDR_A_I_CTRL 0x10
#define XFOO_TOP_SLV0_ADDR_A_I_DATA 0x14
#define XFOO_TOP_SLV0_BITS_A_I_DATA 32
#define XFOO_TOP_SLV0_ADDR_A_O_CTRL 0x18
#define XFOO_TOP_SLV0_ADDR_A_O_DATA 0x1c
#define XFOO_TOP_SLV0_BITS_A_O_DATA 32
#define XFOO_TOP_SLV0_ADDR_B_DATA 0x24
#define XFOO_TOP_SLV0_BITS_B_DATA 32

```

xfoo_top_slv1.h にも同じような情報が含まれますが、slv1 インターフェイスにはブロック レベルの IO プロトコル信号 (return 関数に関連) が含まれるので、ブロック割り込みの設定および制御の詳細も含まれます。

```

// 0x00 :Control signals
//      bit 0 - ap_start (Read/Write/SC)
//      bit 1 - ap_done (Read/COR)
//      bit 2 - ap_idle (Read)
//      others - reserved
// 0x04 :Global Interrupt Enable Register
//      bit 0 - Global Interrupt Enable (Read/Write)
//      others - reserved
// 0x08 :IP Interrupt Enable Register (Read/Write)
//      bit 0 - Channel 0 (ap_done)
//      others - reserved
// 0x0c :IP Interrupt Status Register (Read/TOW)
//      bit 0 - Channel 0 (ap_done)
//      others - reserved
// 0x10 :Control signal of c
//      bit 0 - c_ap_vld (Read/Write/SC)
//      others - reserved
// 0x14 :Data signal of c
//      bit 31~0 - c[31:0] (Read/Write)
// 0x18 :Control signal of d
//      bit 1 - d_ap_ack (Read/COR)
//      others - reserved
// 0x1c :Data signal of d
//      bit 31~0 - d[31:0] (Read/Write)
// 0x20 :Data signal of ap_return
//      bit 31~0 - ap_return[31:0] (Read)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
// Handshake)

#define XFOO_TOP_SLV1_ADDR_AP_CTRL 0x00
#define XFOO_TOP_SLV1_ADDR_GIE 0x04

```

```

#define XFOO_TOP_SLV1_ADDR_IER      0x08
#define XFOO_TOP_SLV1_ADDR_ISR      0x0c
#define XFOO_TOP_SLV1_ADDR_C_CTRL   0x10
#define XFOO_TOP_SLV1_ADDR_C_DATA   0x14
#define XFOO_TOP_SLV1_BITS_C_DATA   32
#define XFOO_TOP_SLV1_ADDR_D_CTRL   0x18
#define XFOO_TOP_SLV1_ADDR_D_DATA   0x1c
#define XFOO_TOP_SLV1_BITS_D_DATA   32
#define XFOO_TOP_SLV1_ADDR_AP_RETURN 0x20
#define XFOO_TOP_SLV1_BITS_AP_RETURN 32

```

上記のインターフェイス ファイルに記述されるアドレス ロケーションには、アクセスするためにそれに関連する C 関数が含まれます。これらの関数は `xfoo_top.c` ファイルで提供され、次に示す `xfoo_top.h` で定義されます。

```

int XFoo_top_Initialize(XFoo_top *InstancePtr, XFoo_top_Config *ConfigPtr);

void XFoo_top_Start(XFoo_top *InstancePtr);
u32 XFoo_top_IsDone(XFoo_top *InstancePtr);
u32 XFoo_top_IsIdle(XFoo_top *InstancePtr);
u32 XFoo_top_GetReturn(XFoo_top *InstancePtr);

void XFoo_top_SetC(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetC(XFoo_top *InstancePtr);
void XFoo_top_SetCVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetCVld(XFoo_top *InstancePtr);
void XFoo_top_SetD(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetD(XFoo_top *InstancePtr);
u32 XFoo_top_GetDAck(XFoo_top *InstancePtr);
void XFoo_top_SetA_i(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetA_i(XFoo_top *InstancePtr);
void XFoo_top_SetA_ivld(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_ivld(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_iAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_o(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_oVld(XFoo_top *InstancePtr);
void XFoo_top_SetA_oAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetA_oAck(XFoo_top *InstancePtr);
void XFoo_top_SetB(XFoo_top *InstancePtr, u32 Data);
u32 XFoo_top_GetB(XFoo_top *InstancePtr);

void XFoo_top_InterruptGlobalEnable(XFoo_top *InstancePtr);
void XFoo_top_InterruptGlobalDisable(XFoo_top *InstancePtr);
void XFoo_top_InterruptEnable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptDisable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptClear(XFoo_top *InstancePtr, u32 Mask);
u32 XFoo_top_InterruptGetEnabled(XFoo_top *InstancePtr);
u32 XFoo_top_InterruptGetStatus(XFoo_top *InstancePtr);

```

これらの関数名はデザイン名およびデザインのポート タイプによって異なります。この例には、一般的な例を説明するため、すべてのタイプのポートが含まれています。次に、この例のために作成される関数について説明します。関数は、AXI4 Lite スレーブ インターフェイスを介した制御、およびブロックへのアクセスに使用される C プログラムで通常呼び出される順序で説明されています。

たとえば、ブロックはまずプログラム スペースにインスタンシエーションされます。次に割り込みが設定されます。ブロックが開始されると、ポート値がアクセスされ、最後に結果が読み出されます。

まず、この関数を使用してプログラム スペースに `XFoo_Top` のインスタンスをインスタンシエートします。

```
int XFoo_top_Initialize(XFoo_top *InstancePtr, XFoo_top_Config *ConfigPtr);
```

割り込みが使用される場合は、まずそれらをプロセッサでイネーブルにする必要があります。このタスクの実行方法については、プロセッサの資料を参照してください。割り込みがプロセッサでイネーブルにならないと、ここで示す割り込み関数は動作しません。

後に示すように、割り込みサービス ルーチンを使用するのではなく、デバイスをポーリングすることも可能です。ただし、次の関数では、割り込みサービス ルーチンを使用しています。AXI Lite スレーブ インターフェイスをサポートする割り込みには、次の 2 つがあります。

- 1 つ目は、`ap_done` 信号のステータスで、ロケーション 1 で使用できます。
- 2 つ目は、タスク レベルの割り込みで、ロケーション 2 で使用できます。タスク レベルの割り込みでは、新規タスクがいつ開始できるかが示され、ポートの動作がパイプライン処理できるようにされます。新規タスクは `ap_done` 信号/割り込みが最初のタスクの終了を示すよりも前に開始できます。

これらの関数をこの順序で使用すると、次の割り込みサービス タスクが C コードで実行できるようになります。

- すべての割り込みソースのステータスを戻す
- どの割り込みがイネーブルかを表示
- クリアされる個々の割り込みを許容

```
u32 XFoo_top_InterruptGetEnabled(XFoo_top *InstancePtr);
u32 XFoo_top_InterruptGetStatus(XFoo_top *InstancePtr);
void XFoo_top_InterruptClear(XFoo_top *InstancePtr, u32 Mask);
```

次のタスクは、割り込みをイネーブルにします。グローバル割り込み関数を使用すると、ハードウェア ブロックからのすべての割り込みがイネーブルになります。

```
void XFoo_top_InterruptGlobalEnable(XFoo_top *InstancePtr);
```

個々の割り込みは、次の関数を使用するとイネーブルにできます。マスク値 1 で `ap_done` 信号からの割り込みが、マスク値 2 でタスク レベルの割り込みがイネーブルになり、マスク値 3 でどちらのタイプの割り込みもイネーブルになります。

```
void XFoo_top_InterruptEnable(XFoo_top *InstancePtr, u32 Mask);
```

この段階では、ブロックの開始前に、入力ハンドシェイクを使用しないポートすべて(例: `ap_none` または `ap_ack`)がコンフィギュレーションされている必要があります(コンフィギュレーションされていないと、ブロックの開始直後にインターフェイス レジスタにデフォルト値が読み込まれます)。この例では、ポート `b` および `d` にこのようなプロトコルが含まれます。これらのポートに値を設定するには、次の関数を使用します。

```
void XFoo_top_SetB(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetD(XFoo_top *InstancePtr, u32 Data);
```

必須ではありませんが、次の関数を使用すると、ポート `b` および `d` からデータを読み戻し、正しい値が書き込まれたどうかを検証できます。

```
u32 XFoo_top_GetB(XFoo_top *InstancePtr);
u32 XFoo_top_GetD(XFoo_top *InstancePtr);
```

ブロックは開始関数を呼び出すと開始できます(まずアイドリング信号のステータスを確認し、最後のプロセスが本当に終了したかどうかを確認することをお勧めします)。

```
u32 XFoo_top_IsIdle(XFoo_top *InstancePtr);
void XFoo_top_Start(XFoo_top *InstancePtr);
```

ハンドシェイクを使用する入力ポートには、ブロックの開始前または開始後に値を設定できます。ブロックが既に開始されたのに入力に有効な値がない場合、有効なデータが入ってくるまでブロックは停止します。

これらの書き込みはハードウェア プロトコル(データ値を書き込んでから有効ビットを設定してデータが有効かどうかを指定)と同じように実行する必要があります。データがブロックで読み出されると、アダプターの有効ビットが自動的にクリアされます。

この例の場合、ポート a(入力側) と c はブロックの開始後に設定されています。これらの値を設定するには、次の関数を使用します。

```
void XFoo_top_SetA_i(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetA_iVld(XFoo_top *InstancePtr);

void XFoo_top_SetC(XFoo_top *InstancePtr, u32 Data);
void XFoo_top_SetCVld(XFoo_top *InstancePtr);
```

ポート a および c の値は、ポート b および d と同じように、次の関数を使用して読み戻すと、値が正しいかどうかを確認できます。

```
u32 XFoo_top_GetA_i(XFoo_top *InstancePtr);
u32 XFoo_top_GetC(XFoo_top *InstancePtr);
```

また、有効フラグのステータスを読み出すことができます。ブロックが既に開始されている場合は、有効フラグはデータがハードウェアで読み出されるとすぐに自動的にクリアされます。

```
u32 XFoo_top_GetA_iVld(XFoo_top *InstancePtr);
u32 XFoo_top_GetCVld(XFoo_top *InstancePtr);
```

この例では、ポート b および d にその IO プロトコルに関連する出力 ACK (acknowledge) 信号が含まれます。この出力 ACK 信号のステータスも読み出すことができます。これが High の場合、読み出しだはハードウェアで認識されています。

次の関数は、ACK ポートのステータスを取得するために使用でき、これらのレジスタが関数呼び出しによって読み込まれると、すぐにレジスタビットがクリアされ、ハードウェアでのみ設定できるようになります。

```
u32 XFoo_top_GetA_iAck(XFoo_top *InstancePtr);
u32 XFoo_top_GetDAck(XFoo_top *InstancePtr);
```

この例では、ブロックがポート a(出力側) で実行されている間は、リターンポート (ap_return) だけがデータを戻すポートになります。次の関数では、ポート a に有効な出力データがあるかどうかをチェックできます。

```
u32 XFoo_top_GetA_oVld(XFoo_top *InstancePtr);
```

有効なデータが出力ポートにあると、それが読み出されます。

```
u32 XFoo_top_GetA_o(XFoo_top *InstancePtr);
```

問題なく読み出しされれば、次の関数を使用して読み出しを認識 (acknowledge) させることができます。ハードウェアは書き込み出力が認識されるまで停止されます。

```
void XFoo_top_SetA_oAck(XFoo_top *InstancePtr);
```

ハードウェアが AXI Lite スレーブ インターフェイスの ACK レジスタを読み出すと、その ACK レジスタが自動的にクリアされます。これは、次の関数を使用すると確認できます。

```
u32 XFoo_top_GetA_oAck(XFoo_top *InstancePtr);
```

最後に、その他すべてのポートが使用されれば、関数からの戻り値を読み出すことができます。割り込みが使用されない場合は、終了したかどうかをチェックするためにデバイスをポーリングできます。

```
u32 XFoo_top_IsDone(XFoo_top *InstancePtr);
```

ap_done 信号をポーリングするにしても、割り込みサービス ルーチンを使用するにしても、次の関数を使用すると戻り値を読み出すことができます。

```
u32 XFoo_top_GetReturn(XFoo_top *InstancePtr);
```

最後に、割り込みが使用される場合は、ブロックの割り込みを個別またはグローバルにディスエーブルにできます。

```
void XFoo_top_InterruptDisable(XFoo_top *InstancePtr, u32 Mask);
void XFoo_top_InterruptGlobalDisable(XFoo_top *InstancePtr);
```

AXI4 Master インターフェイス

AXI4 Master インターフェイスを作成するには、RTL ポートに ap_bus インターフェイスが必要です(表 2-7)。この例では、ポート m が ap_bus として設定され、それが AXI4M リソースに接続されるように指定されています。

ロック レベルのインターフェイス プロトコルは IO プロトコルを ap_ctrl_none に設定することで削除されています。これにより、この例にはその他のインターフェイスがなくなります(ロック レベルのハンドシェイク信号なし、リターン ポートなし)。

```
#include "ap_cint.h"

#define N 256
typedef uint32 DT;

void foo_top (volatile DT *m) {

    // Define the RTL interfaces
    #pragma AP interface ap_ctrl_none port=return
    #pragma AP interface ap_bus port=m

    // Define the pcore interface as an AXI4 master
    #pragma AP resource core=AXI4M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
    for (i = 0, j = N - 1; i < j; i++, j--) {
        tmp = buff[i];
        buff[i] = buff[j];
        buff[j] = tmp;
    }
    memcpy(m, buff, N * sizeof(DT));
}
```

pcore が生成されると、AXI4 マスター インターフェイスがポート m に接続されます。これは AXI4 システム バスに接続できます(図 2-48)。

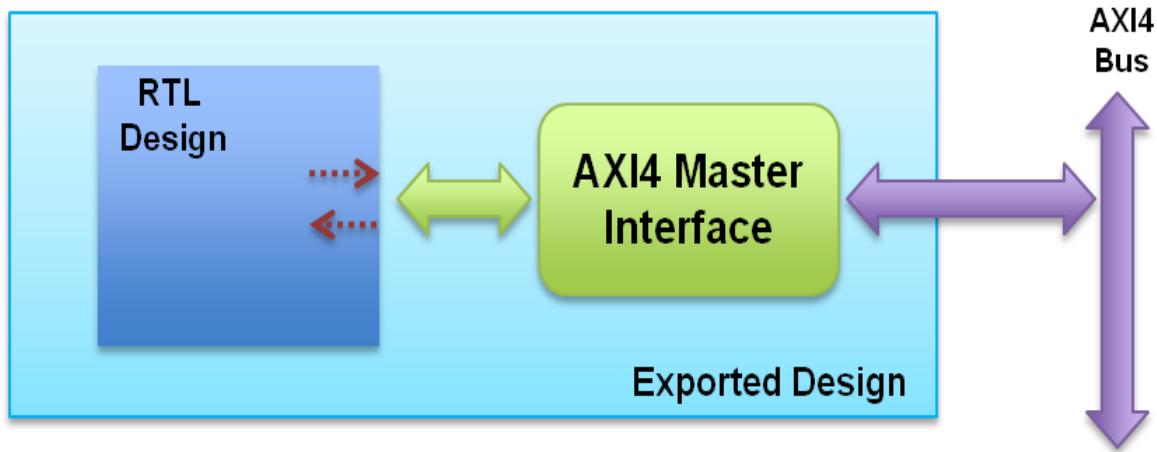


図 2-48: AXI4 マスター インターフェイス

AXI4 Stream インターフェイス

AXI4 Stream インターフェイスは、AXI4-Stream のマスター/スレーブ インターフェイスです。このインターフェイスは、すべての ap_fifo RTL ポートに適用できます。

ストリーム インターフェイスを適用する場合、通常は構造体 (struct) を定義します。この構造体を構成するメンバーはそれぞれ AXI4 Stream バス信号に対応しています。この例では、DATA 構造体が定義され、data と strb の 2 つのメンバーが含まれ、ポート マップとグループ オプションの両方を使用する必要があります。データ変数が 1 つしかない場合は、その他のオプションは必要ありません。

Vivado HLS に含まれる AXI4 Stream インターフェイスは、バス規格の柔軟性に合うように設計されています。AXI4 Stream 規格のほとんどの信号(例:TREADY、TKEEP、TSTRB)はオプションです。必須なのは TVALID および TREADY の 2 つの信号のみです(TREADY はバス規格ではオプションですが、Vivado HLS の AXI4 Stream インターフェイスで使用されます)。

この例では、構造体メンバーの data を TDATA 信号に、strb を TSTRB 信号にマップします。

```
port_map={{data_i_data TDATA} {data_i_strb TSTRB}}
port_map={{data_o_data TDATA} {data_o_strb TSTRB}}
```

構造体を使用すると、複数の変数を 1 つの AXI4 Stream にマップできますが、Vivado HLS のデフォルト動作では、構造体の各メンバーが別々の RTL インターフェイス ポートを作成します。すべてのメンバーを同じ AXI4 グループにマップするには、バスバンドル オプションを使用する必要があります。

複数の RTL ポートは AXI4 スレーブ インターフェイスと同じように 1 つの AXI4 Stream インターフェイスにまとめるすることができます。ただし、AXI4 Stream インターフェイスにまとめられる RTL ポートは、すべて入力ポートかすべて出力ポートである必要があります。双方向インターフェイスは、グループ化ではサポートされていません。

出力インターフェイスは AXI4 Stream マスター インターフェイスを、入力インターフェイスは AXI4 Stream スレーブ インターフェイスをインプリメントします。

この例では、IO プロトコルを ap_ctrl_none に設定することで、ブロック レベルのインターフェイス プロトコルが削除されています。これにより、この例にはその他のインターフェイスがなくなります(ブロック レベルのハンドシェイク信号なし、リターン ポートなし)。

```
#include "ap_cint.h"
```

```
#define N 256
```

```
typedef struct {
    uint32 data;
    uint4 strb;
} DATA;

void foo_top (DATA data_i[N], DATA data_o[N]) {

    // Define the RTL interfaces
    #pragma AP interface ap_ctrl_none port=return
    #pragma AP interface ap_fifo port=data_i
    #pragma AP interface ap_fifo port=data_o

    // Define the pcore interfaces AXI4 slave
    #pragma AP resource core=AXI4Stream variable=data_i metadata="-bus_bundle
    AXI4Stream_S" port_map={{data_i_data TDATA} {data_i_strb TSTRB} }

    // Define the pcore interfaces AXI4 master
    #pragma AP resource core=AXI4Stream variable=data_o metadata="-bus_bundle
    AXI4Stream_M" port_map={{data_o_data TDATA} {data_o_strb TSTRB} }

    int i;
    DATA buf[N];

    for (i = 0; i < N; i++) {
        buf[i] = data_i[i];
    }

    for (i = 0; i < N; i++) {
        data_o[i] = buf[N-1-i];
    }
}
```

ポートマップを GUI で適用する場合、各ポートマップは波括弧で囲み、すべてのマップを波括弧でさらに囲む必要があります。文字列全体を `port_map` オプションボックスに入力する必要があります。次に例を示します。

```
 {{data_o_data TDATA} {data_o_strb TSTRB}}
```

図 2-49 は、上記の例から作成されたインターフェイスを示しており、RTL 入力および出力が別々のインターフェイスにまとめられています。

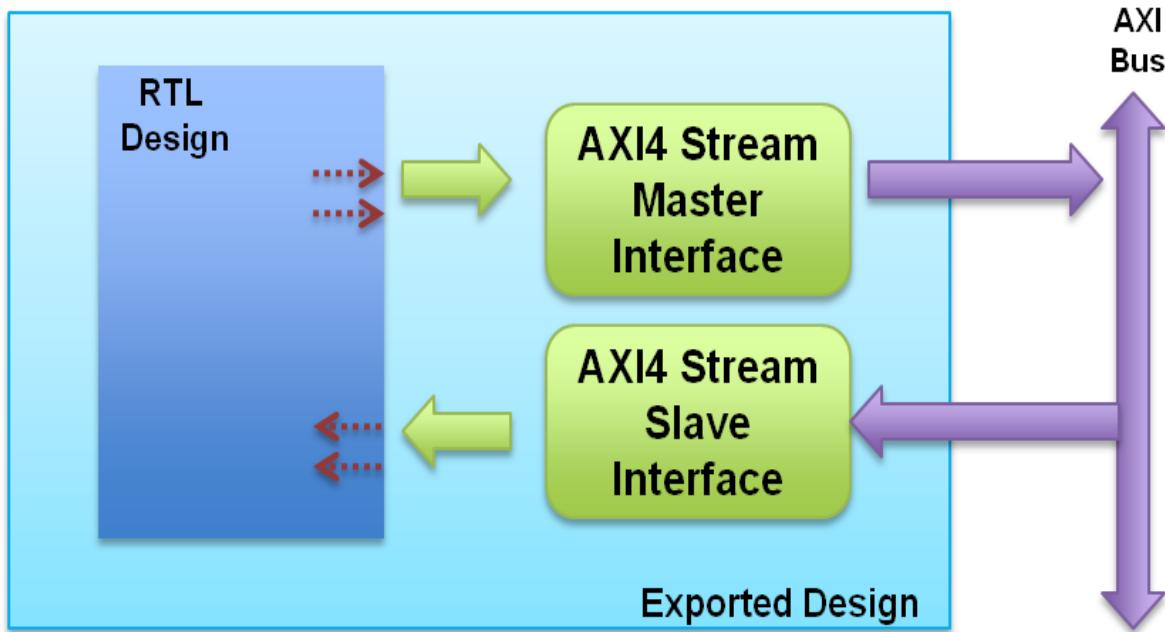


図 2-49 : Pcore : AXI4 Stream インターフェイス

この例では、独自の構造体を使用しています。Vivado HLS インストールディレクトリのインクルードディレクトリにあるヘッダーファイル `ap_axi_sdata.h` では、AXI4 インターフェイス規格のすべての可能性のある信号に対して符号付きおよび符号なしの構造体が定義されています。これらを使用すれば、ユーザー自身が構造体を作成する必要はありません。

- `ap_axi_s` : AXI Stream の符号付き構造
- `ap_axi_s` : AXI Stream の符号なし構造

```
template<int D,int U,int TI,int TD>
struct ap_axis{
    ap_int<D>    data;
    ap_uint<D/8>  keep;
    ap_uint<D/8>  strb;
    ap_uint<U>    user;
    ap_uint<1>    last;
    ap_uint<TI>   id;
    ap_uint<TD>   dest;
};

template<int D,int U,int TI,int TD>
struct ap_axiu{
    ap_uint<D>    data;
    ap_uint<D/8>  keep;
    ap_uint<D/8>  strb;
    ap_uint<U>    user;
    ap_uint<1>    last;
    ap_uint<TI>   id;
    ap_uint<TD>   dest;
};
```

PLB Slave インターフェイス

次の例は、複数ポートがどのように共有の PLB スレーブ インターフェイスにまとめられるかを示しています。これにより、複数の RTL ポートが 1 つのバス インターフェイスを介してアクセスできるようになります。

```
int foo_top (int *a, int *b, int *c, int *d) {  
  
    // Define the RTL interfaces  
    #pragma AP interface ap_hs port=a  
    #pragma AP interface ap_hs port=b  
    #pragma AP interface ap_hs port=c  
    #pragma AP interface ap_hs port=d  
    #pragma AP interface ap_ctrl_hs port=return register  
  
    // Define the pcore interfaces and group into PLB slave "slv0"  
    #pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=a  
    #pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv0" variable=b  
  
    // Define the pcore interfaces and group into PLB slave "slv1"  
    #pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=c  
    #pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=d  
    #pragma AP resource core=PLB_SLAVE metadata="-bus_bundle slv1" variable=return  
  
    *a += *b;  
    return (*c + *d);  
}
```

上記の例でポート a および b は共通の PLB スレーブ インターフェイスにまとめられています (図 2-50)。

ブロック レベルの IO プロトコルは、ap_ctrl_hs インターフェイス モードを return 関数に適用すると設定されます (これがデフォルトですが、この例では明確に指定しています)。すべてのブロック レベルの IO インターフェイス ポート (ap_start, ap_done、など) は関数出力 (ap_return ポート) と共に、1 つのグループとして PLB スレーブ インターフェイスに割り当てられます。

ポート c および d はブロック レベルの IO プロトコル信号と return 関数と一緒に slv1 グループにまとめられます (図 2-50)。

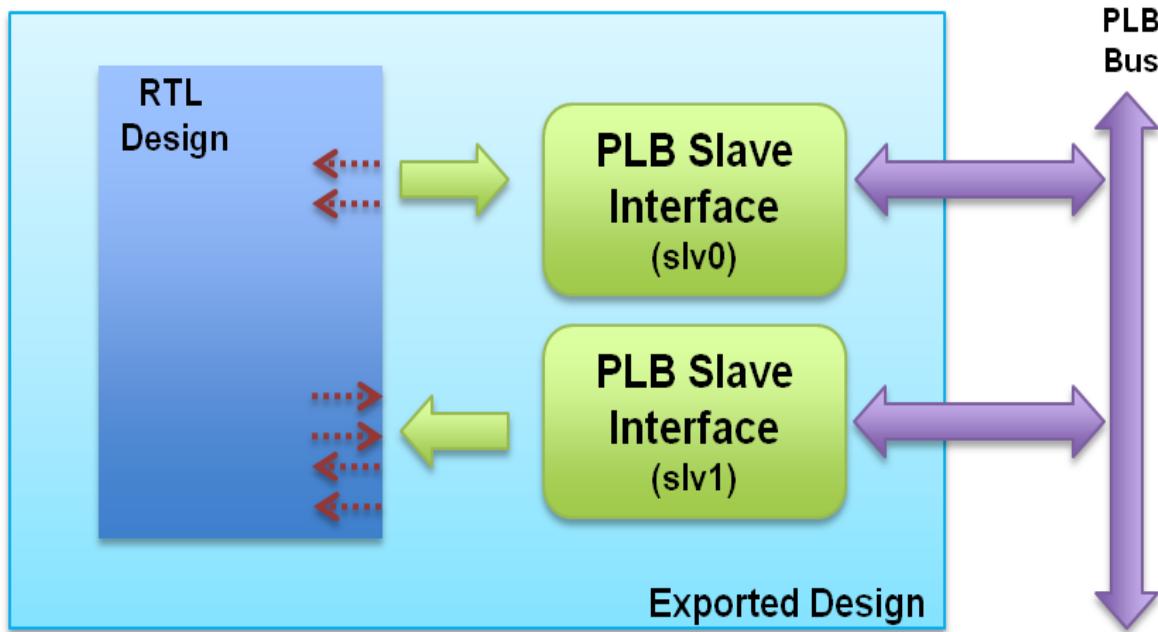


図 2-50 : Pcore : バンドルポートを含む PLB スレーブインターフェイス

RESOURCE 指示子が GUI を使用して適用される場合は、バスバンドルストリング全体(例：“-bus_bundle slv1”)を クオーテーションマークも含めて metadata オプションボックスに入力する必要があります。

Pcore としてエクスポートされると、PLB スレーブインターフェイスが C ファイルのセットを使用して作成され、スレーブインターフェイスにアクセスするプロセッサと一緒に使用されます。これらの C ファイルは AXI4 Lite スレーブインターフェイスに含まれるものと類似しています。C ファイルの説明は、前述の「AXI4 Lite スレーブ」セクションを参照してください。

PLB マスターインターフェイス

PLB マスターインターフェイスを作成するには、RTL ポートに ap_busインターフェイスが必要です(表 2-7)。この例では、ポート m が an_ap_bus として設定され、そのポートリソースが PLB46M リソースであることが指定されています。

ロックレベルのインターフェイスプロトコルは IO プロトコルを ap_ctrl_none に設定することで削除されています。これにより、このデザインにはその他のインターフェイスがなくなります(ロックレベルのハンドシェイク信号なし、リターンポートなし)。

```
#include "ap_cint.h"

#define N 256

typedef uint32 DT;
void foo_top (volatile DT *m) {

    // Define the RTL interfaces
    #pragma AP interface ap_ctrl_none port=return
    #pragma AP interface ap_bus port=m

    // Define the pcore interface as a PLB master
    #pragma AP resource core=PLB46M variable=m
```

```

DT buff[N], tmp;
int i, j;
memcpy(buff, m, N * sizeof(DT));
for (i = 0, j = N - 1; i < j; i++, j--) {
    tmp = buff[i];
    buff[i] = buff[j];
    buff[j] = tmp;
}
memcpy(m, buff, N * sizeof(DT));
}

```

これは PLB v4.6 マスター インターフェイスになります。PLB v4.6 マスター インターフェイスは PLB バスに接続でき、マスター インターフェイスとして動作します。

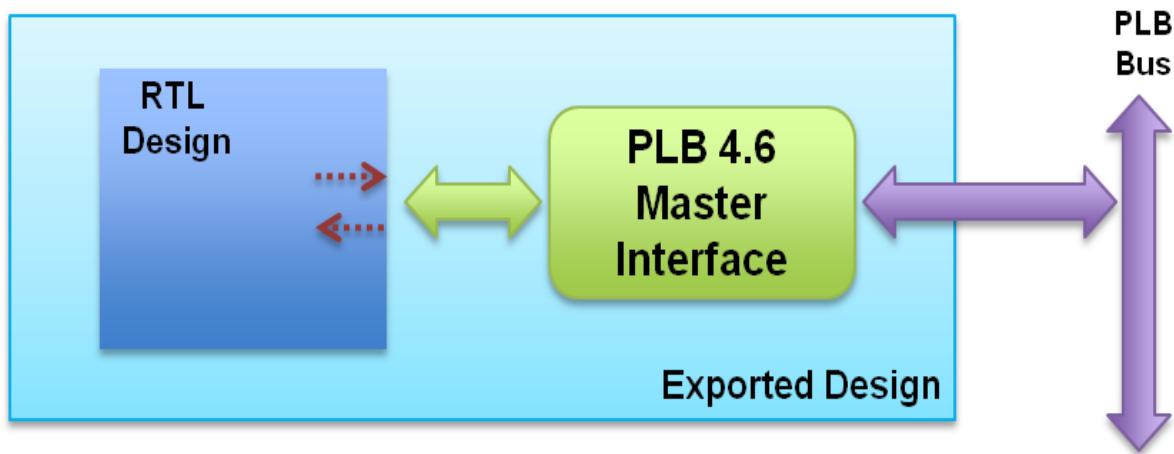


図 2-51 : Pcore : PLB マスター インターフェイス

NPI インターフェイス

NPI インターフェイスは、最初に RTL インターフェイスで `ap_bus` を作成すると作成されます。このインターフェイスには、その後次の例に示すように NPI64M リソースを割り当てます。

```

#include "ap_cint.h"

#define N 256
typedef uint32 DT;

void foo_top (volatile DT *m) {

    // Define the RTL interfaces
    #pragma AP interface ap_ctrl_none port=return
    #pragma AP interface ap_bus port=m

    // Define the pcore interface as an NPI master
    #pragma AP resource core=NPI64M variable=m

    DT buff[N], tmp;
    int i, j;
    memcpy(buff, m, N * sizeof(DT));
}

```

```

for (i = 0, j = N - 1; i < j; i++, j--) {
    tmp = buff[i];
    buff[i] = buff[j];
    buff[j] = tmp;
}
memcpy(m, buff, N * sizeof(DT));
}

```

この例では、IO プロトコルを `ap_ctrl_none` に設定することで、ブロック レベルのインターフェイス プロトコルが削除されています。これにより、このデザインにはその他のインターフェイスがなくなります(ブロック レベルのハンドシェイク信号なし、リターン ポートなし)。

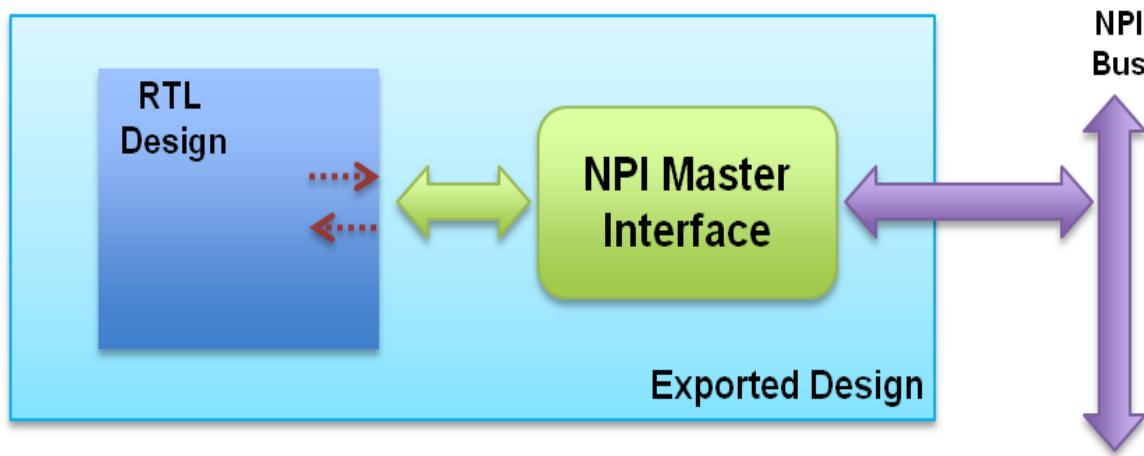


図 2-52 : Pcore : NPI Master インターフェイス

FSL インターフェイス

FSL インターフェイスは、RTL の `ap_fifo` インターフェイスに接続できます。FSL インターフェイスはマスター/スレーブ インターフェイスです。

- インターフェイスが入力の場合、FSL スレーブ インターフェイスが生成されます。
- インターフェイスが出力の場合は、FSL マスター インターフェイスが生成されます。
- FSL インターフェイスは双方向ポートには接続できません (`ap_fifo` インターフェイスはこれらのポートには適用できません)。

```

#define N 256

void foo_top (int data_i[N], int data_o[N]) {

    // Define the RTL interfaces
    #pragma AP interface ap_ctrl_none port=return
    #pragma AP interface ap_fifo port=data_i
    #pragma AP interface ap_fifo port=data_o

    // Define the pcore interfaces as FSL types
    #pragma AP resource core=FSL variable=data_i
    #pragma AP resource core=FSL variable=data_o

    int buff[N], i;
}

```

```

for (i = 0; i < N; i++) {
    buff[i] = data_i[i];
}
for (i = 0; i < N; i++) {
    data_o[i] = buff[N-1-i];
}
}

```

この例には、2つの ap_fifo インターフェイスがあります。1つは入力で、もう1つは出力です。この例では、ブロック レベルのプロトコルを ap_ctrl_none に設定して、その他のポートがないようにしています (return 関数なし、ap_ctrl_none によりブロック レベルのハンドシェイク信号なし)。

図 2-53 は、入力および出力 FSL インターフェイスを示しています。

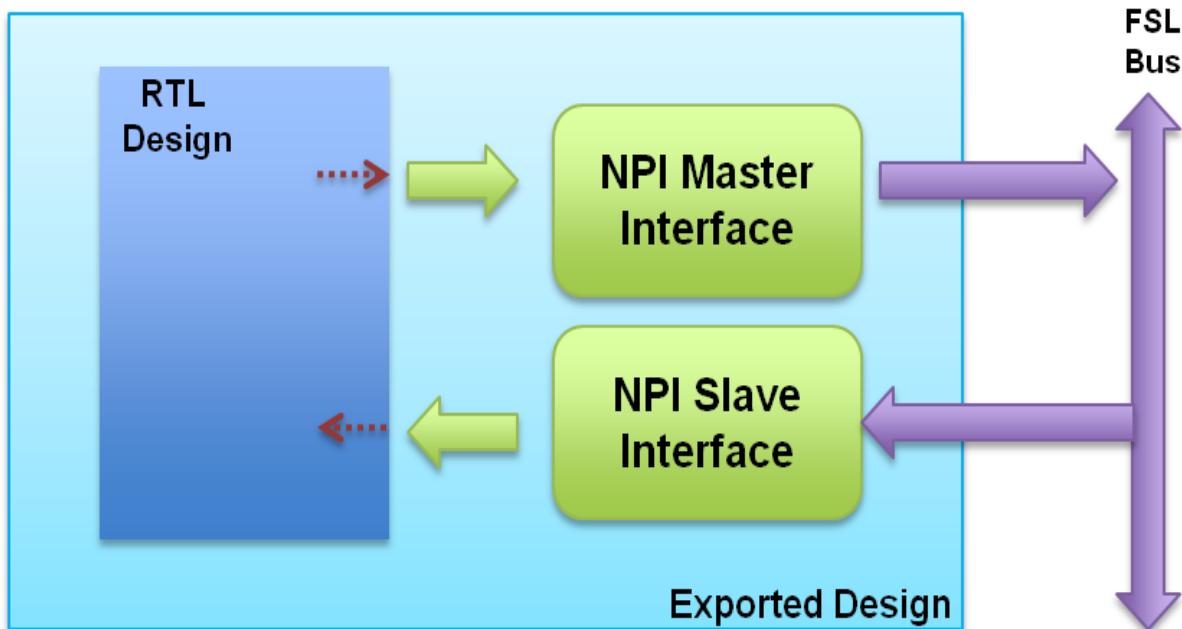


図 2-53 : Pcore : FSL マスター & スレーブ インターフェイス

SystemC のインターフェイス合成

通常 SystemC デザインでは、インターフェイス合成はサポートされません。SystemC デザインの I/O ポートを SC_MODULE インターフェイスに記述し、ポートの動作をソースコードに完全に記述する必要があります。

ただし、メモリポートは例外です。たとえば、標準 SystemC 配列ポートを含む次のようなデザインがあるとします。

```

SC_MODULE(my_design) {
    // "RAM" Port
    sc_uint<20> my_array[256];
    ...
}

```

my_array ポートは、内部 RAM に合成されます。

Vivado HLS ヘッダーファイル ap_mem_if.h を含めると、ポートを ap_mem_port<data_width, address_bits> ポートとして指定できます。ap_mem_port は、指定のデータおよびアドレスバス幅の標準 RAM インターフェイスポートに合成されます。

上記のコード例は、次のように変換できます。

```
#include "ap_mem_if.h"

SC_MODULE(my_design) {
    // "RAM" Port
    ap_mem_port<sc_uint<20>, sc_uint<8>, 256> my_array;
    ...
}
```

これにより、インターフェイスポート `my_array` が確実に RAM インターフェイスとしてインプリメントされます。

SystemC デザインに `ap_mem_port` を追加した場合、SystemC テストベンチに `ap_mem_port` を駆動する `ap_mem_chn` を追加する必要があります。

テストベンチでは、次のように `ap_mem_chn` を定義してインスタンスに接続します。

```
#include "ap_mem_if.h"
...
ap_mem_chn<int, int, 68> bus_mem;
...

// Instantiate the top-level module
my_design U_dut ("U_dut")
U_dut.my_array.bind(bus_mem);
...
```

ヘッダーファイル `ap_mem_if.h` は、\$VIVADO_HLS_ROOT\include\ap_systc\ap_mem_if.h にあり、SystemC のシミュレーションに含める必要があります。

手動のインターフェイス仕様

Vivado HLS には、特定の I/O プロトコルを定義するコード ブロックを特定する機能があります。この機能を使用すると、インターフェイス合成または SystemC を使用せずに I/O プロトコルを指定できます (SystemC では、下に説明するプロトコル指示子を使用して I/O をより厳密に制御可能)。

次の例は、次のようなコードです。

- 入力 `response[0]` が読み出されます。
- 出力 `request` が書き込まれます。
- 入力 `response[1]` が読み出されます。
- 最終的なデザインで、この順序で I/O アクセスを実行する必要があります。

```
void test (
    int *z1,
    int a,
    int b,
    int *mode,
    volatile int *request,
    volatile int response[2],
    int *z2
) {

    int read1, read2;
    int opcode;
    int i;

    P1: {
```

```

    read1      = response[0];
    opcode     = 5;
    *request   = opcode;
    read2      = response[1];
}
C1:{{
    *z1      = a + b;
    *z2      = read1 + read2;
}
}

```

Vivado HLS でこのコードをインプリメントした場合に、request への書き込みを response に対する 2 つの読み出しの間にする必要はありません。コードはこのように I/O が動作するよう記述されていますが、この依存性を強制するものはコードにありません。Vivado HLS では、I/O アクセスがコード記述と同様にスケジュールされる場合と、ほかの順序が使用される場合があります。

この場合、特定の I/O プロトコル動作を強制するためにプロトコルブロックを使用できます。アクセスはブロック P1 で定義された範囲内で実行されるので、プロトコルを次のように適用できます。

- ap_wait() を定義する ap_utils.h ヘッダー ファイルを含めます。
- request への書き込みの後、response[1] の読み出しの前に、ap_wait() 文を追加します。
 - ap_wait() 文によりシミュレーションの動作が変わることはできませんが、合成ではここにクロックが挿入されます。
- ブロック P1 をプロトコル領域に指定します。
 - このようにすると、この領域内のコードがそのままスケジューリングされます。I/O と ap_wait() 文の順序が変更されることはありません。

次のように指示子を適用します。

```
set_directive_protocol test P1 -mode floating
```

コードを次のように変更します。

```
#include "ap_utils.h"// Added include file
```

```

void test (
    int      *z1,
    int      a,
    int      b,
    int      *mode,
    volatile int *request,
    volatile int response[2],
    int      *z2
) {

    int    read1, read2;
    int    opcode;
    int    i;

P1:{{
    read1      = response[0];
    opcode     = 5;
    ap_wait(); // Added ap_wait statement
    *request   = opcode;
    read2      = response[1];
}
C1:{{

```

```
*z1    = a + b;  
*z2    = read1 + read2;  
}  
}
```

これにより、コードに指定されたとおりの I/O 動作が得られます。

- 入力 `response[0]` が読み出されます。
- 出力 `request` が書き込まれます。
- 入力 `response[1]` が読み出されます。

`-mode floating` オプションを使用すると、データの依存性で許容される場合は、ほかのコードをこのブロックと並列実行できます。このオプションを `fixed` に設定すると、並列実行されません。

デザイン最適化

この章では、Vivado HLS でパフォーマンス、エリア、消費電力の目標を満たすマイクロ アーキテクチャが生成されるようにするための、さまざまな最適化および手法を説明します。

この章のトピックは、適用する正しい最適化をすばやく見つけられるように構成されており、生産性を向上します。トップを次の順に参照し、デザイン最適化を実行することをお勧めします。

- チェックリストおよびガイドライン
- クロック、タイミング、および RTL 出力
- 任意精度データ型
- 最適化の実行

「チェックリストおよびガイドライン」では、この章で説明する最適化を適用するためのストラテジを示します。エリア、スループット、レイテンシ、消費電力など、さまざまなデザインの目標の最適化ストラテジが提供されています。デザインの全体的な目標がこれら的一般的な目標とは異なる場合でも、このセクションを参照することにより、デザインの目標を満たすために Vivado HLS の機能を適用するための基礎を学ぶことができます。

論理合成などの高位合成では、入力ソース コードや制約をほんの少し変更しただけでも、出力デザインが異なるものになります。最初からやり直さなければならないような状況を避けるため、「クロック、タイミング、および RTL 出力」および「任意精度データ型」を参照して、複雑な最適化を適用する前に、基本的なデザインパラメーターおよびデザイン記述が正しく、できるだけ理想に近いことを確認してください。

複雑で高度な最適化を適用する段階になったら、トップダウンおよびボトムアップ手法を使用できます。

- デザインでレイテンシまたはスループットが目標からかなり離れている場合は、トップダウン手法を使用します。抽象度の高いレベルで最適化を適応した方が、大きく向上します。デザインを機能レベルから開始してロジック レベルに向かって見直します。このユーザー ガイドでも、この順序で情報を提供しています。
- デザインが目標をほぼ達成している場合は、クロック サイクルまたはリソースをいくつか削減することに焦点を置きます。この場合、ロジック構造のレベルから開始し、別のコンポーネントを選択することにより 1 サイクルで実行される操作が増加するか、配列アクセスが障害となっていないかなどを確認して、機能レベルに向かって作業します。

チェックリストおよびガイドライン

このセクションでは、最適化の目標を短時間で達成するための基本的なストラテジの概要を示します。

デザインの基礎

「クロック、タイミング、および RTL 出力」に説明されている基本的なデザインパラメーターを見直し、RTL が次のもので作成されていることを確認します。

- 正しいクロックとクロックのばらつき
- 正しいリセットスタイル
- ステート エンコード

インターフェイス合成

アルゴリズムとインターフェイス合成の両方に注目することが重要です。インターフェイスはデータを供給するので、必要なスループット レートを達成するの障害となっている可能性があります。

デザイン最適化を実行する前に、正しいインターフェイスが使用されていることを確認してください。

- デザイン上の現在のインターフェイスが、通信するデザインと互換していることを確認します。
- 「インターフェイスの管理」のセクションを参照し、正しいインターフェイスを選択して定義します。
- 選択したインターフェイスを合成後の検証方法で使用できることを確認します。インターフェイスを後で変更すると、スケジュールがことなるものになる可能性があります。検証フローの詳細は、「検証」のセクションを参照してください。

最後に、パフォーマンス要件を満たすためには、インターフェイスのインプリメンテーション方法またはインターフェイスのタイプを変更する必要があるということを認識することが必要です。たとえば、配列インターフェイスを次のように変更すると有益な場合があります。

- シングルポートからデュアルポートの RAM インプリメンテーションに変更する。
- RAM からストリーミング FIFO インプリメンテーションに変更する。
- 配列から DMA のようなバスインターフェイスをサポートするポインターに変更する。

このような決定および変更は、手動で最適化を実行する場合と同じです。デザインの上のレベルを考慮し、インターフェイスプロトコルを変更するのが、デザインを向上させる唯一の方法である場合があります。Vivado HLS では、このような変更により有益なアーキテクチャが得られるかどうかを簡単に判断できます。

データ型とビット幅

Vivado HLS では、演算特性に基づいてデータ型が伝搬されますが、明示的に指定する方が確実です。

C 関数での変数のビット幅は、RTL インプリメンテーションで使用されるストレージエレメントのサイズと演算子に直接影響します。変数に 8 ビットのみが必要なのに整数型(32 ビット)で指定されていると、大型で低速の 32 ビット演算子が使用され、1 クロック サイクルで実行できる演算の数が削減し、レイテンシおよびスループットが増加する可能性があります。

- データ型には、適切な精度を使用します。この章の「任意精度」を参照してください。
- RAM またはレジスタとしてインプリメントされる配列のサイズを確認します。各配列には複数の要素があるので、配列に必要以上に大きい要素があると、そのエリアへの影響は拡大します。
- 乗算、除算、対数演算、その他の複雑な算術演算に特に注目します。これらの変数が必要以上に大きい場合、エリアおよびパフォーマンスの両方に悪影響を与えます。
- ANSI C を使用する場合は、ビット幅を変更した後に apcc を使用して、既存のシミュレーションと同じ結果が得られるかどうかを確認します。apcc が必要な理由は、「任意精度」を参照してください。

エリアを最小限に抑えたデザイン

デザインのエリアを最小限に抑えるには、関数およびループを再利用するようにします。関数およびループは、実行されるたびに同じハードウェアリソースで繰り返されます。これにより、演算よりも上のレベルで共有を最大化できます。

- エリアを縮小する作業を始める前に、デザインがパフォーマンス要件を満たしているか、ほぼ満たしていることを確認します。
- 関数が複数回呼び出されている場合は、同じハードウェアが供給されます。これは、コード記述方法によりエリアを節約する最良の方法です。関数インライン化により、より多くの関数が共有されるかどうかを確認してください（「関数の再利用、インライン化」を参照）。
- ループも関数と同様に同じハードウェアで繰り返されます。これによりループ機能が小さいエリアにインプリメントされますが、レイテンシが大きくなります。forループが展開されていないことを確認してください。デフォルトでは展開されません。詳細は、「ループの最適化」を参照してください。
- デザインがパイプライン処理されている場合は、開始間隔を変えた場合でも、コンポーネントを再利用しながらスループット要件が満たされることを確認します。詳細は、「関数の最適化」および「ループの最適化」を参照してください。
- 配列が既存の RAM に最適にインプリメントされているか、配列を分割したり変更することで使用可能な RAM リソースがより効率的に使用され、並列アクセスが実行できるようになるかを確認します。詳細は、「配列の最適化」を参照してください。

スループットを最大にするデザイン

スループットを最大にするデザインでは、最小のサイクル数でできるだけ多くのデータを処理するようにします。

- 関数レベルから開始し、特に C および C++ デザインの場合は「関数のデータフロー パイプライン処理」を参照してください。SystemC では、並列処理がサポートされます。
- 「ループのパイプライン処理」を参照して、パイプライン処理をループに適用できるかどうかを判断します。

可能なパイプライン処理がすべて適用されている場合や、パイプライン処理が不可能な場合は、各関数およびループでレイテンシを最小にします。演算を完了するのに必要なサイクル数が少なければ、次の入力を早く読み込むことができます。レイテンシを削減する手法は次のセクションで説明しますが、デザインスループットを考慮する場合、次の点が重要です。

- レポートのサマリ セクションを確認します。階層の各関数で最も影響するクリティカルループを調べ、「レイテンシを最小限に抑えたデザイン」の手法を使用してクリティカルループのレイテンシを削減します。
 - 25 クロックで実行可能なループが 2 回実行される場合、影響は 50 サイクルです。
 - 4 クロックで実行可能なループが 256 回実行される場合、影響は 1024 サイクルであり、よりクリティカルであると考える必要があります。
- デザインポートが制限となっている場合は、インターフェイスタイプを変更することを考慮します。この場合、インターフェイスタイプの変更がシステムのほかのデザインでサポートされるかを確認する必要があります。

レイテンシを最小限に抑えたデザイン

デザインのレイテンシを削減する方法は、次のとおりです。

- 関数およびループ レベルから開始し、特に C および C++ デザインの場合は「関数のデータフロー パイプライン処理」および「ループのデータフロー パイプライン処理」を参照して並列処理を向上します。SystemC では、並列処理がサポートされます。
- for ループがある場合は、「ループの展開」に説明されているように一部またはすべてを展開し、レイテンシが削減されるかどうかを確認します。展開すると、より多くの演算を少ないサイクル数で並列実行できるようになりますが、リソースおよびエリアが増加します。

- 複数のループがある場合は、1つのループ本体から別のループに移動するのに1クロックサイクル必要であることを考慮します。「ループの結合」および「ネストされたループのフラット化」を参照してループのオーバーヘッドを削減してください。
- 配列には注意します。配列は通常メモリにマップされますが、アクセスは制限されます（読み出しポートおよび書き込みポート）。これにより、レイテンシを増加するハードウェアの依存性が発生する可能性があります。たとえば、デュアルポート RAM または「配列の最適化」で説明されているリコンフィギュレーションされた RAM では、同じクロックサイクルで読み出しおよび書き込みをもっと実行できる場合があります。

消費電力を最小限に抑えたデザイン

Vivado HLS では、自動的に多数の消費電力最適化が実行されます。これらの最適化には、スケジューリング中に実行される演算子のゲーティングや、パイプラインの開始へのパイプラインイネーブルの追加などがあります。すべての消費電力最適化は、パフォーマンス制限に悪影響を与えなければ、自動的に適用されます。

Vivado HLS では、パフォーマンスを犠牲にして消費電力が削減されることはありませんが、さまざまな手法を使用してさらに消費電力を削減できます。

- エリアを削減すると消費電力に大きく影響するので、「エリアを最小限に抑えたデザイン」を参照します。
- Vivado HLS のアーキテクチャ探索機能を活用してみてください。クロック周期を変更し、新しいマイクロアーキテクチャをすばやく再生成できます。

クロック、タイミング、および RTL 出力

C および C++ デザインでは、クロック 1 つのみがサポートされます。デザイン内のすべての関数に、同じクロックが適用されます。SystemC デザインでは、各 SC_MODULE に異なるクロックを指定できます。

SystemC デザインで複数のクロックを指定するには、create_clock コマンドの -name オプションを使用して名前を指定してクロックを作成し、set_directive_clock コマンドまたはプラグマを使用して、指定のクロックで合成する SC_MODULE を含む関数を指定します。各 SC_MODULE の合成には、1 つのクロックのみを使用できます。複数のクロックを最上位ポートから個々のブロックに接続している場合など、複数のクロックを関数を介して分配できますが、各 SC_MODULE は 1 つのクロックにのみ応答します。

クロック周期は、GUI の [Solution Settings] ダイアログ ボックスで ns で指定します。クロックのばらつきを指定してタイミング マージンを定義することも可能です。

Vivado HLS ではデザインの操作のタイミングを予測できますが、最終的なコンポーネントの配置およびネット配線はわかっていないので、正確な遅延は算出できません。クロックのばらつきは、クロック周期から差し引いて、使用可能なクロック周期を求めます（図 2-54）。Vivado HLS は使用可能なクロック周期を使用して、デザインの操作をスケジューリングします。

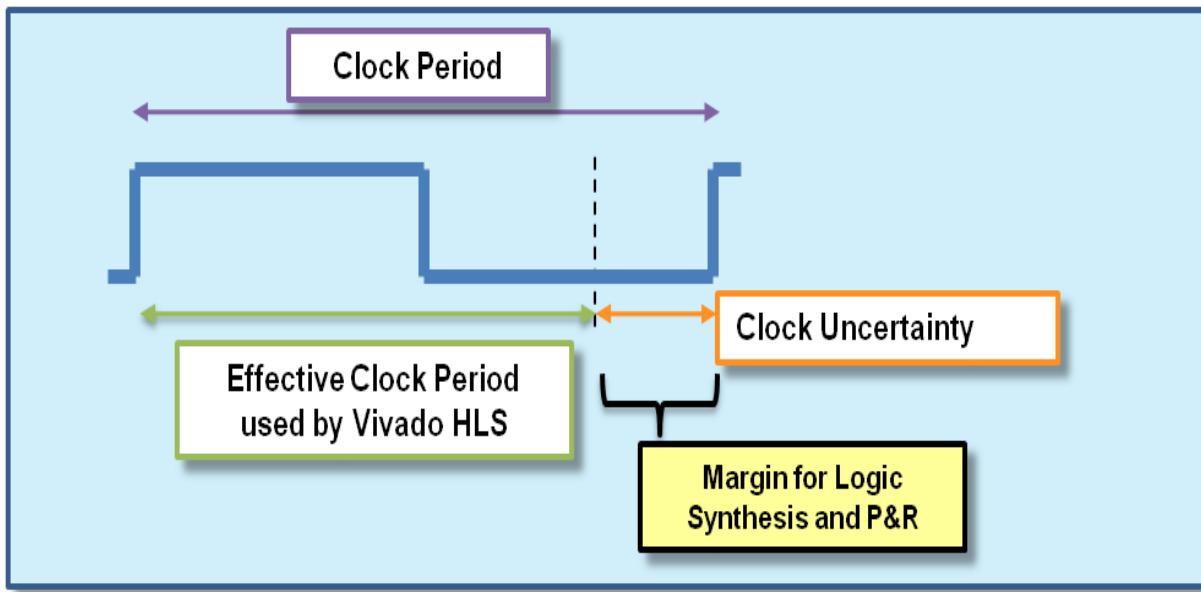


図 2-54: クロック周期とマージン

これにより、ユーザー指定のスラックが使用され、論理合成や配置配線などのダウンストリーム プロセスで操作を完了するために十分なタイミング マージンが供給され、タイミング クロージャを達成するためのデザインの反復回数を削減できます。

デフォルトではクロックのばらつきはサイクル時間の 12.5% に設定されますが、[Solutions Settings] ダイアログ ボックスの [Synthesis] ページでユーザーが指定できます。自動設定を使用すると、クロック周期を変更した場合にクロックのばらつきもアップデートされます。クロックのばらつきを手動で設定した場合は、クロック周期を変更したときにクロックのばらつきも手動でアップデートする必要があります。

タイミング

RTL 演算子およびレジスタに使用されるタイミング情報は、ライブラリで定義されています。ライブラリはすべて事前に特性化されており、Vivado HLS に格納されています。

Vivado HLS では、レイテンシ、スループット (開始間隔)、およびタイミング制約を満たすことを目的として処理が実行されますが、制約が満たされない場合でも、RTL デザインが出力されます。

- データの依存性によりスループット制約を満たすことができない場合は (スループット要件は 1 であるがメモリから値を読み出すのに 2 サイクル必要など)、タイミングが満たされたるまでスループットが増加されます。
- クロック周期により推論されるタイミング制約を満たすことができない場合は、Vivado HLS で次のような SCED-644 というメッセージが表示され、達成可能な最良のパフォーマンスのデザインが output されます。
`@W [SCED-644] Max operation delay (<operation_name> 2.39ns) exceeds the effective cycle time`

特定のパスでタイミング要件が満たされない場合でも、ほかのすべてのパスではタイミングが満たされたるように処理が実行されます。これにより、ダウンストリーム プロセスにより最適化を実行したりタイミングが満たされないパスを特別処理することにより、タイミングを満たすことができるかをユーザーが評価できます。



重要：合成後に制約レポートを確認して、すべての制約が満たされたかどうか（高位合成で生成された出力デザインがすべてのパフォーマンス制約を満たすわけではないという事実）を確認することが重要です。デザインレポートの「Performance Estimates」セクションを確認してください。

デザインレポートは、合成が完了したときに階層の各関数に対して自動的に生成され、[Explorer] タブの [syn] → [report] フォルダーから参照でき、solution ディレクトリの `./syn/report/<function name>.rpt` に保存されます。

デザイン全体のワースト ケースタイミングは、各関数レポートにレポートされます。階層の各レポートをすべて参照する必要はありません。タイミング違反が最適化やダウンストリームプロセスで修正できないほど大きい場合は、より高速のテクノロジをターゲットする前に、この章のこの後に説明する手法を試してみてください。

RTL 出力

Vivado HLS の RTL 出力のさまざま特性は、`config_rtl` コマンドを使用して制御できます。次のような操作が可能です。

- RTL ステートマシンで使用される FSM エンコードを指定
- `-header` オプションを使用して、すべての RTL ファイルに著作権情報などのコメント文字列を追加
- `-prefix` オプションを使用して、すべての出力ファイルに固有の接頭辞を追加し、同じ最上位関数から生成された RTL ファイル（デフォルトでは同じ名前）を簡単に同じディレクトリにまとめる

RTL コンフィギュレーションを定義するには、[Solution] → [Solution Settings] をクリックし、左側のボックスで [General] を選択して [Add] をクリックし、[Command] ドロップダウン リストから [config_interface] を選択します（[図 2-55](#)）。

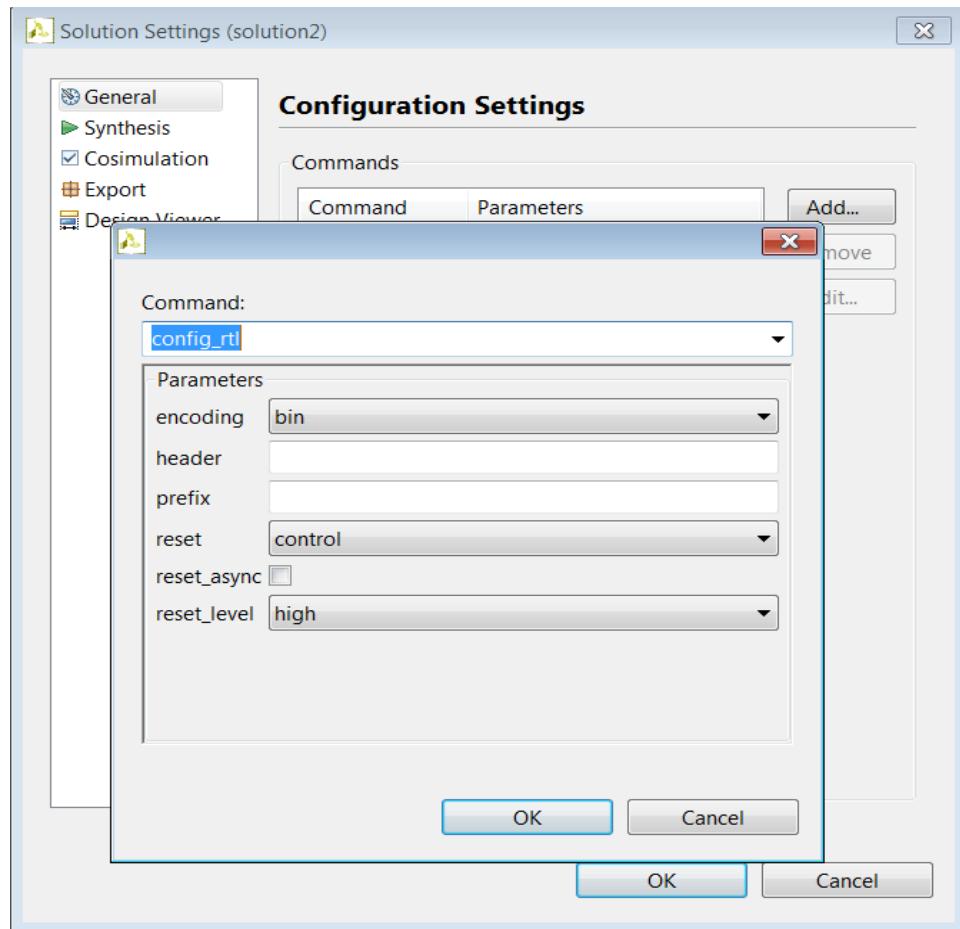


図 2-55 : RTL コンフィギュレーション

RTL コンフィギュレーション設定で最も重要なのは、通常リセット動作の選択です。

RTL リセット動作の選択

リセット動作について説明する前に、初期化とリセットの違いを理解しておくことが重要です。

C では、変数は `static` 修飾子で定義され、グローバルに定義された変数はデフォルトで 0 に初期化されます。これらの変数に初期値を指定することも可能です。変数の初期値を指定した場合、C コードの初期値はコンパイル時(時間 0)に割り当てられ、その後再び割り当てられることはできません。RTL では、どちらの場合も、同じ初期値が合成後にインプリメントされ、FPGA ビットストリームによりデバイスが同じ初期値に初期化されます。

スタティック変数またはグローバル変数でない変数に初期値が設定されている場合、関数が実行されるたびに初期値が割り当てられます。つまり、このような変数の初期化は RTL デザインが開始するたびに実行され、通常の動作の一部となります。FIR フィルターのアキュムレータはその一例です。新しいデータサンプルが到着すると、アキュムレータが 0 に設定され、計算および累積が実行されます。結果が次のトランザクションに繰り越されることはありません。次のサンプルが到着すると、アキュムレータが再び 0 に設定されます。

スタティック変数およびグローバル変数の初期値が RTL でも割り当てられることに加え、オプションで RTL にリセットを適用することもできます。デフォルトではリセットが追加されます。

注記 : リセットは初期化とは別であり、初期化に加えて設定されます。

リセットのコンフィギュレーションによっては、リセット信号が適用されたときに、変数が電源投入時の初期ステートに初期化されない場合もあります。その場合、初期値は電源投入時にのみ適用されます。

現在のところ、デフォルトではスタティック変数およびグローバル変数はリセット時に再初期化されません。

RTL のリセット動作は、図 2-55 に示す RTL コンフィギュレーション設定ダイアログ ボックスで指定します。

リセットの極性、リセットが同期または非同期のどちらか(ザイリンクス テクノロジでは同期リセットの使用を推奨)、および [reset] オプションでリセット信号を適用したときにリセットするレジスタを指定できます。

[reset] オプションには、次の 4 つの設定があります。

- [none] : デザインにリセットは追加されません。
- [control] : ステート マシンに使用されるレジスタや I/O プロトコル信号を生成するために使用されるレジスタなどのレジスタをリセットにより制御します。
- [state] : C コードのスタティック変数またはグローバル変数から生成されたレジスタおよびレジスタ/メモリをリセットにより制御します。C コードで初期化されるスタティック変数またはグローバル変数は、初期値にリセットされます。
- [all] : デザイン内のレジスタおよびメモリをすべてリセットします。C コードで初期化されるスタティック変数またはグローバル変数は、初期値にリセットされます。

デフォルト設定は [control] です。

注記 : [state] に設定すると、RAM でインプリメントされる配列がリセット時に初期化されます。RAM としてインプリメントされる多くの配列はスタティックとして定義されるので、要素が明示的に初期化されなくても、すべての要素が 0 に初期化されます。

大型メモリの場合、リセットに何クロック サイクルもかかることがあります。インプリメントするのによりエリアおよびリソースが必要です。

次に、このセクションで説明したすべての属性を指定する Tcl コマンドの例を示します。このほかのオプションは、config_rtl の man ページを参照してください。このコマンドは、次を実行します。

- クロック周期を 10ns に設定
 - テクノロジ ライブラリを追加
 - クロックのばらつきを 2ns に設定(設定しない場合はクロック周期の 12.5%)
 - 出力 RTL にすべてのレジスタをリセットするアクティブ Low の非同期リセットを作成
 - すべてのステート マシンにワンホット エンコードを使用
- ```
open_solution solution2

set_part {xc6vlx365tff1759-3}

create_clock -period 10ns
set_clock_uncertainty 2

config_rtl my_func -encoding onehot -reset all -reset_level low -reset_async
```

これらの基本的な仕様を設定したら、この章の残りのセクションで説明する最適化を実行できます。

# 関数の最適化

関数に指定可能な最適化は、GUI にリストされます (図 2-56)。

1. [Explorer] タブでソース コードをダブルクリックして開きます。
2. 補足エリアで [Directive] タブをクリックします。
3. 関数を右クリックして [Insert Directives] をクリックします。
4. [Directive] ドロップダウンリストから適切な指示子を選択し、オプションを設定します。

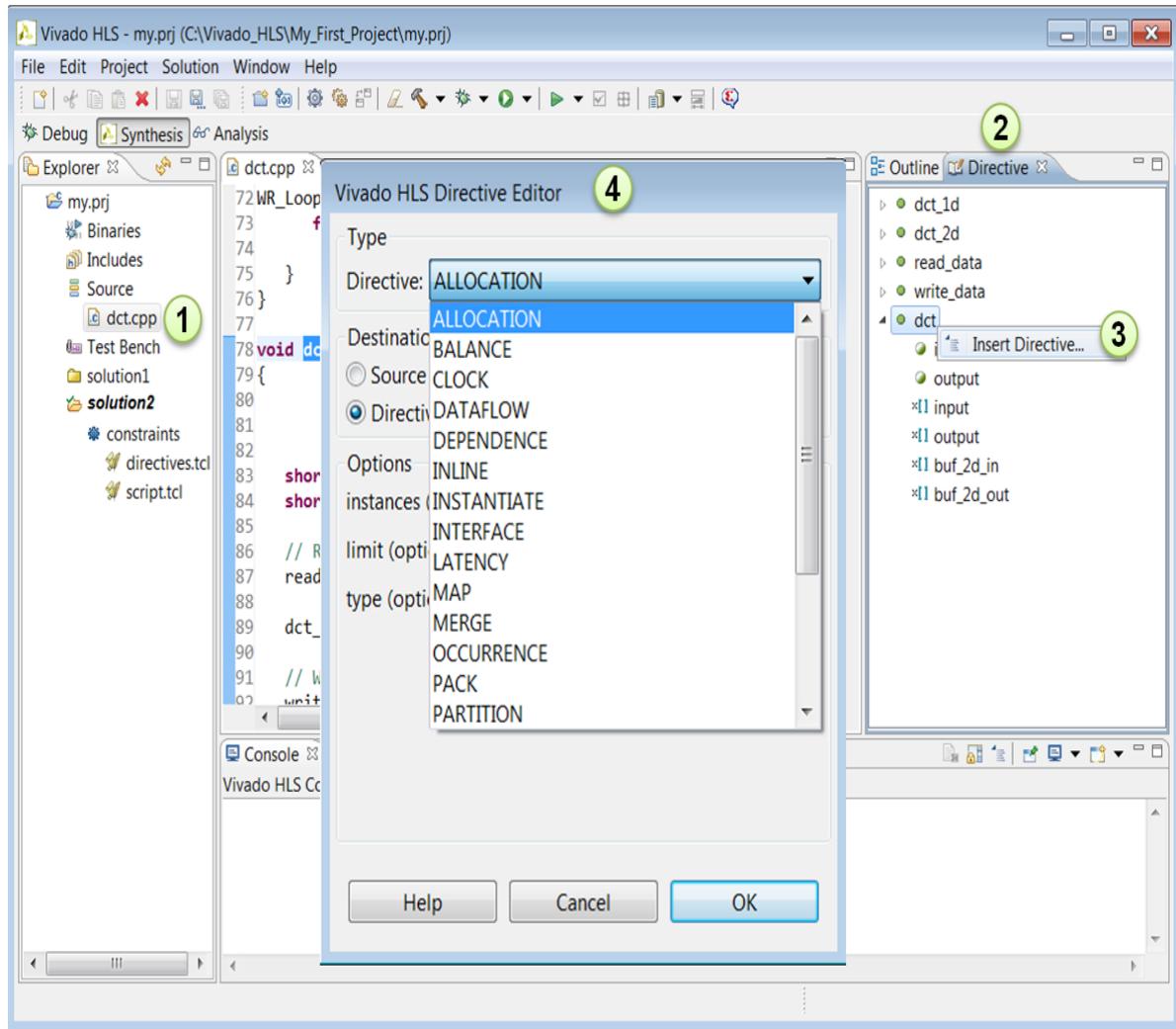


図 2-56 : 関数の指示子

**ヒント :** 関数レベルで適用できる指示子すべてが関数の最適化に関連しているわけではありません。



たとえば、RESHAPE 指示子を関数に適用すると、関数内のすべての配列に適用され、複数の配列に同時に指示子を適用できます。配列の最適化については、別の章で説明します。

表 2-8 に、関数自体の動作および最適化に影響する指示子を、この章で説明する順に示します。

表 2-8: 関数の最適化

| GUI 指示子     | 説明                                                                                                         |
|-------------|------------------------------------------------------------------------------------------------------------|
| INLINE      | 関数をインライン化し、関数の階層をすべて削除します。関数呼び出しのオーバーヘッドを削減することにより、レイテンシおよびスループットを向上します。                                   |
| INSTANTIATE | 関数をローカルで最適化できるようにします。                                                                                      |
| DATAFLOW    | 関数レベルでの同時実行をイネーブルにし、スループットおよび」レイテンシを向上します。                                                                 |
| PIPELINE    | 関数内での演算を同時実行できるようにし、関数のスループットを向上します。                                                                       |
| LATENCY     | 関数に最小および最大レイテンシ制約を指定します。                                                                                   |
| INTERFACE   | 関数レベルのハンドシェイクを適用します。合成された制御ポート start、done、および idle により、テストベンチを再利用できます。このオプションについては、「インターフェイスの管理」で詳細に説明します。 |

## 関数の再利用、インライン化、およびインスタンシエーション

Vivado HLS では、関数呼び出しの階層の合成がサポートされます。デフォルトでは、各関数が特定のハードウェアインプリメンテーションにマップされます。関数内で同じ関数を複数回呼び出すと、RTL デザインにブロックをインスタンシエートするように、同じハードウェアが再利用されます。

関数の再利用では、関数内のすべての演算が共有されるので、リソースを共有し、デザインを小型にするのに効果的です。関数の階層を削除すると、個々の演算が複数の場所で作成され、同様のリソース共有は実行されません(異なる場所でのローカル最適化により共有が制限される)。

### 関数のインライン化

関数に入ったり出たりするには 1 サイクルかかるので、関数の階層を削除すると、レイテンシおよびスループットが向上することがあります。

関数をインライン化すると、関数の階層を削除できますが、通常エリアが増加します。ただし、関数が数回呼び出されるだけであったり小型の場合は、関数をインライン化することにより関数内のコンポーネントを共有しやすくなり、エリアが向上することがあります。このほかにも、関数をインライン化することが有益な場合があります。

- 関数の共有を向上
- アーキテクチャ探索機能を使用可能

関数のすべてのインスタンスに同じインプリメンテーションが使用されますが、関数が確実に共有されるようにするには、同じ関数内から同じ階層レベルで呼び出す必要があります。これには、ほかの関数をインライン化する必要がある場合もあります。

次のコード例では、関数呼び出し foo\_1 と foo\_2 は共有できますが、foo\_3 は別の階層レベルにあるので共有できません。

```
foo {x,y} {
 ...
}
foo_sub (p, q) {
 int q1 = q + 10;
```

```

 foo(p1, q); // foo_3
 ...
}
void foo_top { a, b, c, d} {
 ...
 foo(a, b); // foo_1
 foo(a, c); // foo_2
 foo_sub(a, d);
 ...
}

```

上記の例では、インライン化を適用して関数 foo\_sub により作成された階層を削除することにより、共有を向上できます。これには、GUI では 図 2-56 に示す [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [INLINE] を選択するか、set\_directive\_inline コマンドを使用します。

```
set_directive_inline foo_sub
```

インライン化指示子は、-recursive オプションを使用して指定の関数の下位にあるすべての関数もインライン化するよう指定できます。上記の例でこのオプションを使用すると、関数 foo\_sub と関数呼び出し foo\_3 がインライン化されます。-recursive オプションを最上位関数に使用すると、デザインのすべての関数階層が削除されます。

-off オプションを使用すると、関数がインライン化されなくなります。上記の例に次のコマンドを適用するとします。

```

set_directive_inline -region -recursive foo_top
set_directive_inline -off foo_sub

```

foo\_top の下位にあるすべての関数がインライン化され、関数呼び出し foo\_1 および foo\_2 はインライン化されますが、foo\_sub に -off オプションが使用されているので、foo\_3 はインライン化されません。

関数のインライン化を使用すると、ソースに変更を加えずにコードの構造を大きく変更できるので、アキテクチャ探索の効果的な手法として使用できます。

## 関数のインスタンシエーション

関数のインスタンシエーションは、関数階層を保持しながら、関数の特定のインスタンスに対してローカル最適化を実行する最適化手法です。これにより関数呼び出し周辺の制御ロジックが簡略化され、レイテンシおよびスループットが向上することがあります。

関数インスタンシエーションでは、関数が呼び出されたときに関数の一部の入力が定数であることがあるという事実を利用して、周辺の制御構造(通常定数を生成)を簡略化し、最適化されたより小型の関数ブロックを作成します。これを例を使用して説明します。

次のようなコードがあるとします。

```

void A() {
 B(true);
 B(false);
 B(true);
 B(false);
}

void B(bool mode) {
 if (mode) {
 // code segment 1
 } else {
 // code segment 2
 }
}

```

関数 B は複数回実行されますが、mode が true であるか false であるかによって、排他的である場合とない場合があります。関数 B の各インスタンスは同じようにインプリメントされます。これは関数の再利用およびエリア最適化の面では有益ですが、関数内の制御ロジックが必要以上に複雑になります。

関数のインスタンシエーションを実行するには、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスの [Directive] ドロップダウン リストから [INSTANTIATE] を選択するか、コードにプラグマを挿入するか、次の Tcl コマンドを使用します。

```
set_directive_function_instantiate -variable mode=true B
```

関数のインスタンシエーションを実行すると、次に示すように、コードが 2 つの関数を含むものに変更され、それがモード値用に最適化されます。

```
void A() {
 B1();
 B2();
 B1();
 B2();
}

void B1() {
 // code segment 1
}

void B2() {
 // code segment 2
}
```

関数 B から作成された 2 つの新しい関数 B1 と B2 では、制御構造が簡略化されています。

関数が異なる階層レベルで呼び出され、インライン化をいくつも適用したりコードを変更したりしないと関数共有が困難な場合、関数のインスタンシエーションによりエリアが向上することがあります。ローカルで最適化された小型の関数の方が、共有できない大型の関数が多数あるよりも効率的です。

## 関数のデータフロー パイプライン処理

データフロー パイプライン処理は、順次に記述された関数(図 2-57)から、並列処理アーキテクチャ(図 2-58)を作成します。データフロー パイプライン処理は、デザインのスループットを向上するのに優れた方法です。



図 2-57 : 順次関数記述



図 2-58 : 並列処理アーキテクチャ

図 2-58 に示すチャネルにより、前の関数がすべての演算を完了するまで待機するようになっています。図 2-59 に、データフロー パイプライン処理により関数がどのように並列実行され、デザイン全体のスループットを向上してレイテンシを削減するかを示します。

データフロー パイプライン処理が使用されていない図 2-59 の (A) では、func\_A が開始してから再び新しい入力を処理できるようになるのに 8 サイクルかかり、func\_C により出力が書き込まれるまでに 8 サイクルかかります (出力は func\_C の最後に書き込まれる)。

図 2-59 (B) では、func\_A が開始してから 3 クロック サイクル後に新しい入力を処理でき (スループットが向上)、最終的な値が出力に書き込まれるまでに 5 サイクルかかります (レイテンシが短縮)。

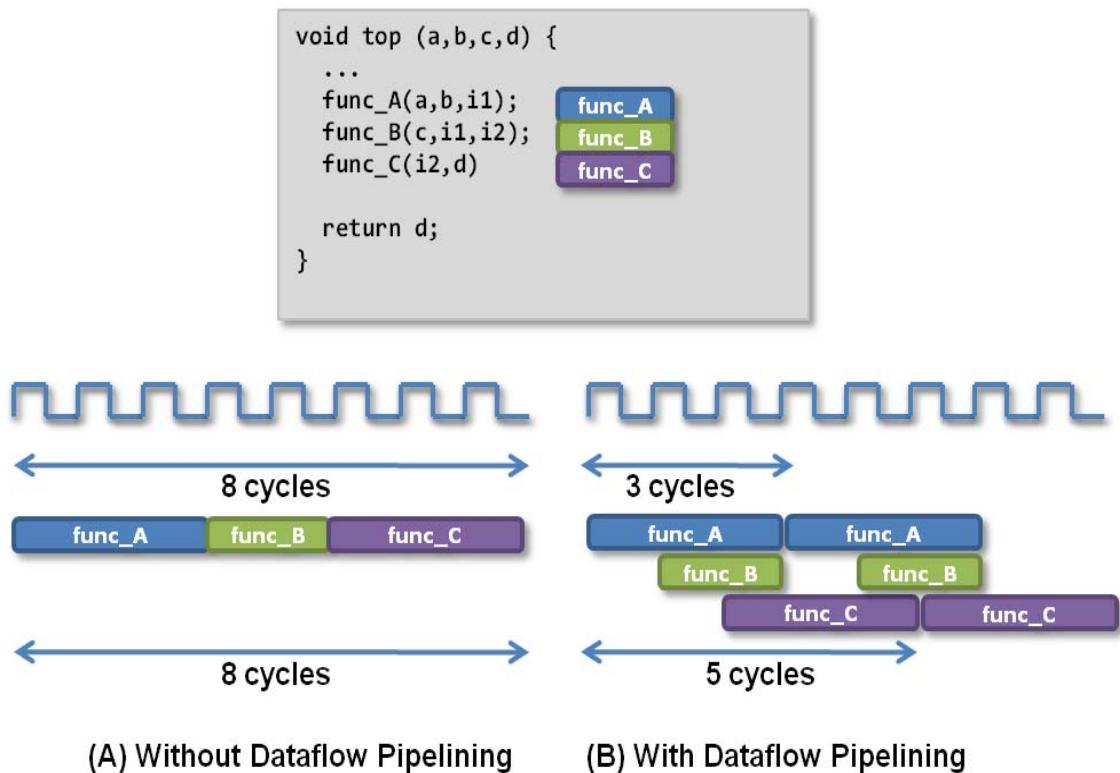


図 2-59: データフロー パイプライン処理

処理間のチャネルは、データの送信側と受信側のアクセスパターンによって、ピンポン バッファーまたは FIFO でインプリメントされます。

- 関数パラメーター (送信側または受信側) が配列の場合、チャネルはピンポン バッファーとして、標準メモリアクセスを使用してアドレス信号と制御信号と共にインプリメントされます。
- スカラー、ポインター、参照パラメーター、および関数リターンの場合は、チャネルは FIFO としてインプリメントされます。この場合、使用されるハードウェアリソースは少なくなりますが (アドレスは生成されない)、データにシーケンシャルにアクセスする必要があります。



**重要:** データフロー パイプライン処理を使用するには、引数を各関数で 2 回しか使用できません。1 回は送信側で 1 つの関数呼び出しから (戻り引数を含む)、もう 1 回は受信側で別の関数引数で使用します。

データフロー パイプライン処理は、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [DATAFLOW] を選択するか、`set_directive_dataflow` コマンドを使用します。関数に対してこの指示子を設定すると、Vivado HLS でその関数内の関数の並列処理を向上するよう処理されます。図 2-59 に示す例では、

次のコマンドを使用すると関数 func\_A、func\_B、および func\_C に対してデータフロー パイプライン処理が実行されます。

```
set_directive_dataflow top
```

データフロー パイプライン処理を使用すると、デフォルトでは Vivado HLS で各 RTL ブロックを同じクロック エッジで開始できるかどうかが試みられます。データの依存性により前の関数からデータが供給されるまで次の関数を実行できない場合(図 2-59 の (B) では func\_B は func\_A が i1 を生成するまで 1 クロック サイクル待つ必要がある)、1 つのブロックの開始と次のブロックの開始間隔が可能な最小サイクル数に自動的に調整されます。

-interval オプションを使用すると、1 つの RTL ブロックが実行を開始してから次の RTL ブロックが実行を開始するまでのサイクル数を指定できます。たとえば、-interval を 3 に設定すると、図 2-59 の (B) で各関数の開始間隔は 3 サイクルになります。

スカラー値の場合、最大チャネル サイズは 1 で、1 つの関数から別の関数に渡すことのできる値は 1 つのみです。関数引数として配列を使用すると、チャネル(メモリ)の要素数は受信側配列または送信側配列の最大サイズで定義されます。Vivado HLS では、デフォルトのチャネル ワード数を指定できます(「データフロー メモリの設定」を参照)。

関数にデータフロー パイプライン処理を適用すると、現在の階層レベルのサブ関数のみがパイプライン処理されます。サブ関数に別の関数が含まれており、データフロー パイプライン処理が有益である場合、サブ関数をインライン化して、すべての関数が同じ階層レベルになるようにします。

## データフロー メモリの設定

関数インターフェイス間で使用されるデフォルトのチャネルは、config\_dataflow コマンドを使用して指定できます。このコマンドを使用すると、ソリューションのデフォルト操作を設定できます。デザインのすべてのチャネルに対するデフォルトのチャネル サイズおよびインプリメンテーションを指定可能です。

```
config_dataflow -default_channel (fifo | *pingpong*) -fifo_depth < FIFO size >
```

チャネルのサイズは、受信側配列または送信側配列の最大サイズで定義されます。場合によっては、これは控えめすぎる設定があります。-fifo\_depth オプションを使用すると、このデフォルトを変更できます。



**重要:** 配列パラメーターが FIFO チャネルを使用するよう指定されている場合、配列をストリーミング型配列に設定する必要があります。ストリーミングについては、「配列のストリーミング」を参照してください。

デフォルトのチャネル タイプが FIFO で、set\_directive\_array\_stream コマンドを使用して配列がストリーミング以外に設定されている場合、その配列のチャネルインプリメンテーションはデフォルトでピンポン チャネルになります(明示的なディレクトリによりコンフィギュレーションが変更される)。

## 関数のパイプライン処理

データフロー パイプライン処理では関数間の通信が最適化されてスループットが向上しますが、関数のパイプライン処理では関数内の操作が最適化され、スループットが向上されます。

図 2-60 に、関数のパイプライン処理によるスループットの向上を示します。関数のパイプライン処理では、操作が同時に実行されます。関数が次の操作を開始する前にすべての操作を完了する必要はありません。

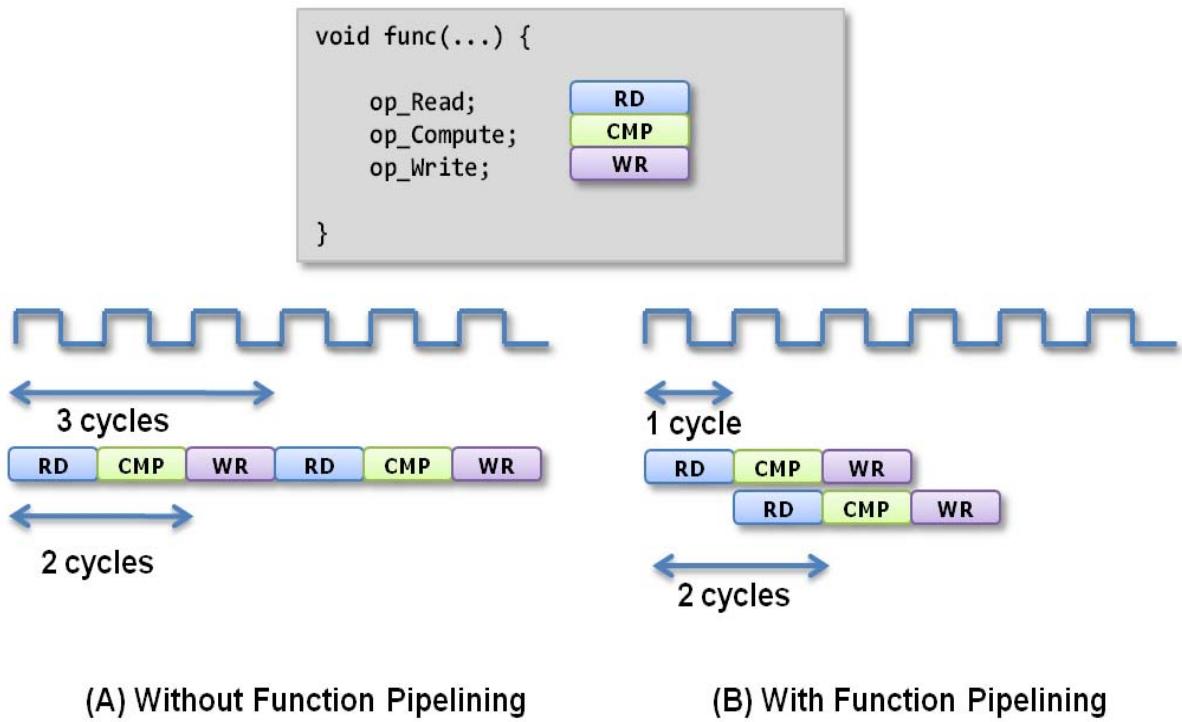


図 2-60 : 関数のパイプライン処理

パイプライン処理されていない場合、関数で入力が 3 クロック サイクルごとに読み出され、値が 2 サイクルごとに出力されます。パイプライン処理を実行すると、新しい入力は各サイクルで読み出され、出力レイテンシまたは使用されるリソースに変化はありません。

関数のパイプライン処理は、パイプライン処理の妨げとなるリソースの競合やデータの依存性がない場合にのみ可能です。たとえば、図 2-61 で配列  $m[2]$  がシングルポート RAM にインプリメントされているとすると、図 2-61 の (A) に示すように、入力  $m[2]$  に対する 2 つの読み出し ( $op\_Read\_m[0]$  および  $op\_Read\_m[1]$ ) を同じクロック サイクルで実行することはできないので、関数はパイプライン処理できません。

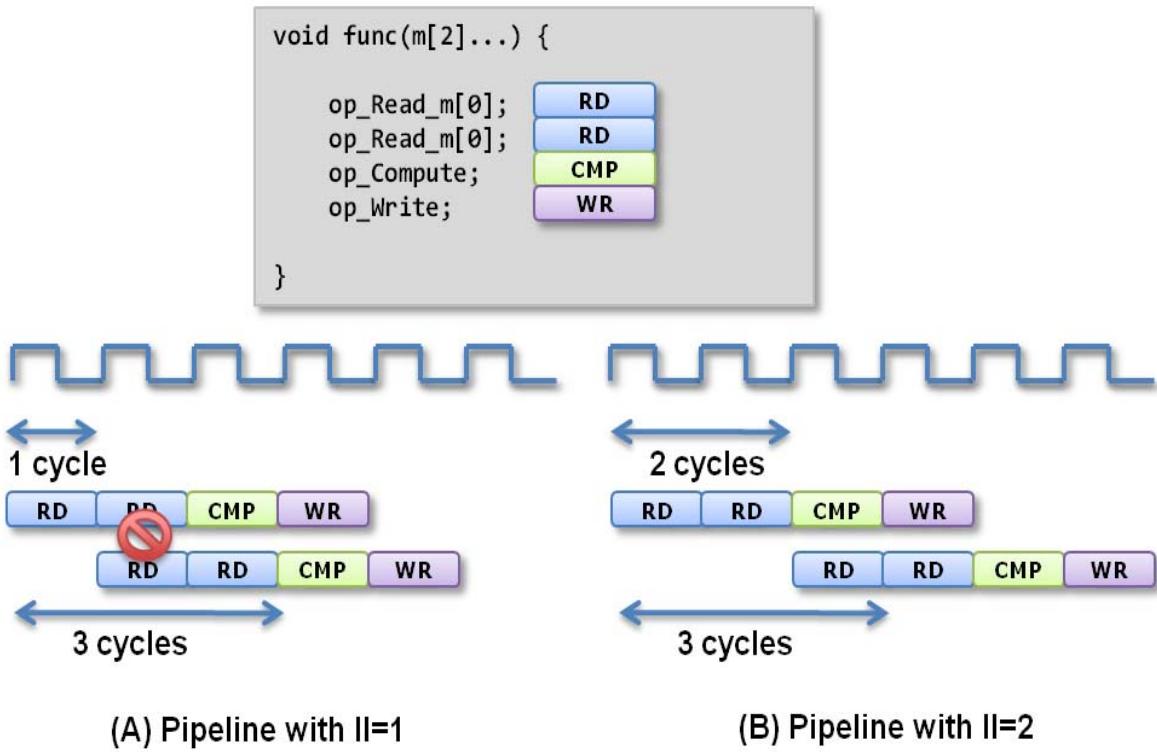


図 2-61: パイプライン処理によるリソースの競合

この関数は、図 2-62 の (B) に示すようにパイプラインの開始間隔を増加すると、パイプライン処理できます。開始間隔は、入力の読み出し間のサイクル数です。

図 2-61 の (A) では新しい操作が各クロックサイクルで実行されるので、開始間隔は 1 です。図 2-61 の (B) では開始間隔 2 が使用されており、入力ポートでリソースの競合が発生することはなくなり、関数が正しくパイプライン処理されています。

**注記:** リソースの競合は、ポートに対する読み出しおよび書き込み、使用可能な乗算器が 1 つのみなど限られたリソースへのアクセス、または RAM または FIFO にマップされた配列の読み出しおよび書き込みにより発生することができます。

パイプライン処理で適切な開始間隔が達成できなかった場合のリソース競合については、「配列の最適化」および「ハードウェアリソースの制御」を参照してください。図 2-61 のリソース競合は、配列 `m[2]` にデュアルポート RAM を使用して、同じクロックサイクルで 2 つの読み出しを実行できるようにしても解決できます。

関数のパイプライン処理は、次の Tcl コマンドを使用して適用できます。

```
set_directive_pipeline function_foo
```

または、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [PIPELINE] を選択します。パイプライン処理は、指定の関数の階層に適用されます。その階層の下にあるすべてのサブ関数はすべてインライン化され、階層内のループは展開されます。

デフォルトでは、関数のパイプライン処理により恒久的に実行されるパイプラインが作成されます。-flush オプションを使用すると、パイプラインをフラッシュして空にすることができます。パイプラインをフラッシュすると、パイプラインの開始時にデータ Valid 信号によりデータがないことが示されたときに、新しい入力の読み出しが停止されますが、処理は継続され、最後の入力が処理されて出力されるまで、1 段ずつパイプライン段を閉鎖していきます。

## レイテンシ制約

Vivado HLS では、関数にレイテンシ制約を使用できます。関数に最大制約または最小制約を設定すると、Vivado HLS で関数のすべての操作が指定のクロック サイクル内に完了するよう処理されます。

レイテンシ制約は、[図 2-56](#) に示す [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [LATENCY] を選択するか、`set_directive_latency` コマンドを使用します。

```
set_directive_latency -min 3 -max 5 function_foo
```

Vivado HLS でレイテンシ制約を満たすことができない場合、「クロック、タイミング、および RTL 出力」に説明するように、クロック サイクルの 1 つのタイミング違反が許容されます。Vivado HLS では、最小限のタイミング違反でデザインが生成され、ダウンストリームの論理合成を使用してタイミングを満たすことができるようになります。タイミング違反が論理合成で満たすには大きすぎる場合は、「ロジック構造の最適化」の手法を使用してロジック遅延を削減してください。

すべてのレイテンシ制約が満たされているかどうかを確認するには、制約レポートを参照してください。

## 関数のインターフェイスプロトコル

Vivado HLS には、関数のインターフェイスプロトコルを自動的に作成する機能があります。関数を合成すると、各関数パラメーター、関数の戻り値、および関数によりアクセスされるグローバル変数が、最終的な RTL デザインでは入力ポートまたは出力ポートとしてインプリメントされます。これらのポートに加え、Vivado HLS では関数の制御ポートを合成でき、RTL インプリメンテーションを周辺のシステムに統合しやすくなります。

インターフェイスプロトコルでは、1 に設定することにより関数の実行を開始する入力信号 (`ap_start`)、関数がすべての操作を完了したことを示す出力信号 (`ap_done`)、および関数で操作が実行されていないことを示すアイドル出力信号 (`ap_idle`) が供給されます。

関数レベルのインターフェイスプロトコルが自動的に追加され、プロトコルのインプリメンテーションの詳細をソース コードに記述する必要はなく、アルゴリズムの高位仕様を保つことができます。

インターフェイスプロトコルは階層の関数に適用できますが、最上位関数にのみ適用し、Vivado HLS によりサブ関数間の最適な通信がスケジューリングされるようにすることをお勧めします。

関数のインターフェイスプロトコルについては、「インターフェイスの管理」で説明しており、プロトコルの詳細な波形を[図 2-33](#) に示しています。

## ループの最適化

関数内の C 言語記述は、一般的に複数のループにより構成されます。HLS でループがどのようにインプリメントおよび最適化されるか、ループ階層がどのように影響するかを理解することは、には、RT レベルで最良のパフォーマンスを達成するために重要です。

ループに指定可能な指示子は、GUI にリストされます ([図 2-62](#))。

1. [Explorer] タブでソース コードをダブルクリックして開きます。
2. 補足エリアで [Directive] タブをクリックします。
3. ループを右クリックして [Insert Directives] をクリックします。
4. [Directive] ドロップダウン リストから適切な指示子を選択し、オプションを設定します。

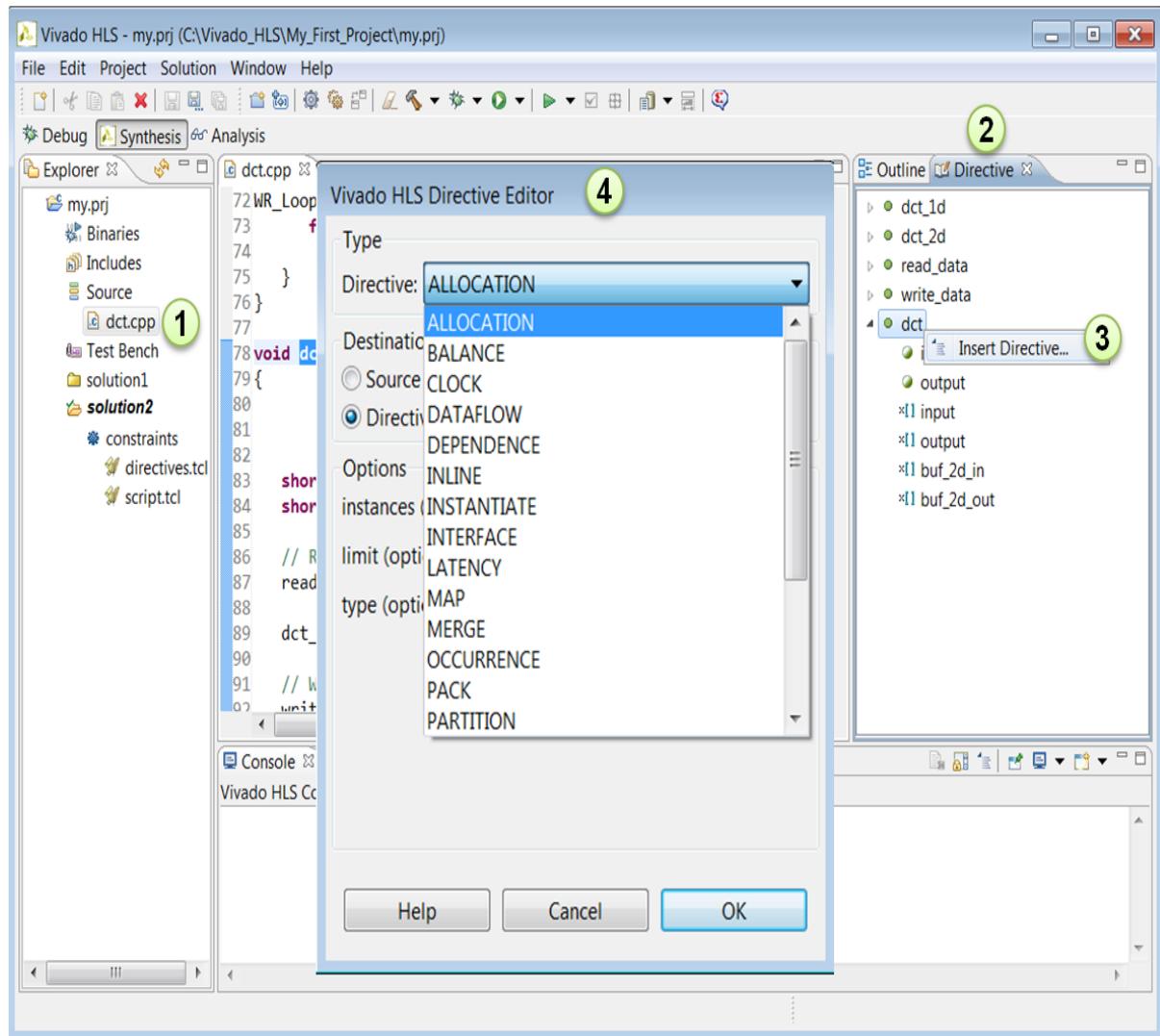


図 2-62 : 関数の指示子

重要 : ループに適用できる指示子すべてがループの最適化に関連しているわけではありません。



たとえば、BALANCE 指示子をループに適用すると、ループで作成されたロジック構造に適用されます。ロジック最適化については、別の章で説明します。

表 2-9 に、ループに適用可能な最適化を、この章で説明する順に示します。

表 2-9: ループ レベルの最適化

| GUI 指示子    | 説明                                                |
|------------|---------------------------------------------------|
| UNROLL     | for ループを展開し、複数の操作を 1 つにまとめたものではなく、複数の個別の操作を作成します。 |
| MERGE      | 連続するループを結合して、全体的なレイテンシを削減し、共有および最適化を向上します。        |
| FLATTEN    | ネストされたループを 1 つのループにし、レイテンシおよびロジック最適化を向上します。       |
| DATAFLOW   | シーケンシャルループを同時に実行できるようにします。                        |
| PIPELINE   | 同時操作を実行することによりスループットを向上します。                       |
| DEPENDENCE | ループ キャリー依存性を解決するための追加情報を供給します。                    |
| TRIPCOUNT  | 反復解析を設定します。                                       |
| LATENCY    | ループ操作のサイクルレイテンシを指定します。                            |

## ループの展開

Vivado HLS のデフォルトでは、ループは展開されません。つまり、ループは 1 つのエンティティとして処理され、ループのすべての操作は同じハードウェアリソースを使用してインプリメントされます。

Vivado HLS には、for ループの一部または全体を展開する機能があります。

図 2-63 に、ループ展開の利点と、ループを展開する際の考慮事項を示します。図 2-63 の例では、配列  $a[i]$ 、 $b[i]$ 、および  $c[i]$  は RAM にマップされると想定されています。配列が順次エレメントにマップされない場合、サイクル数は乗算器の組み合わせ遅延により決まります。

図 2-63 からまず最初にわかることは、ループの展開を適用しただけで多数のインプリメンテーションを作成できるということです。

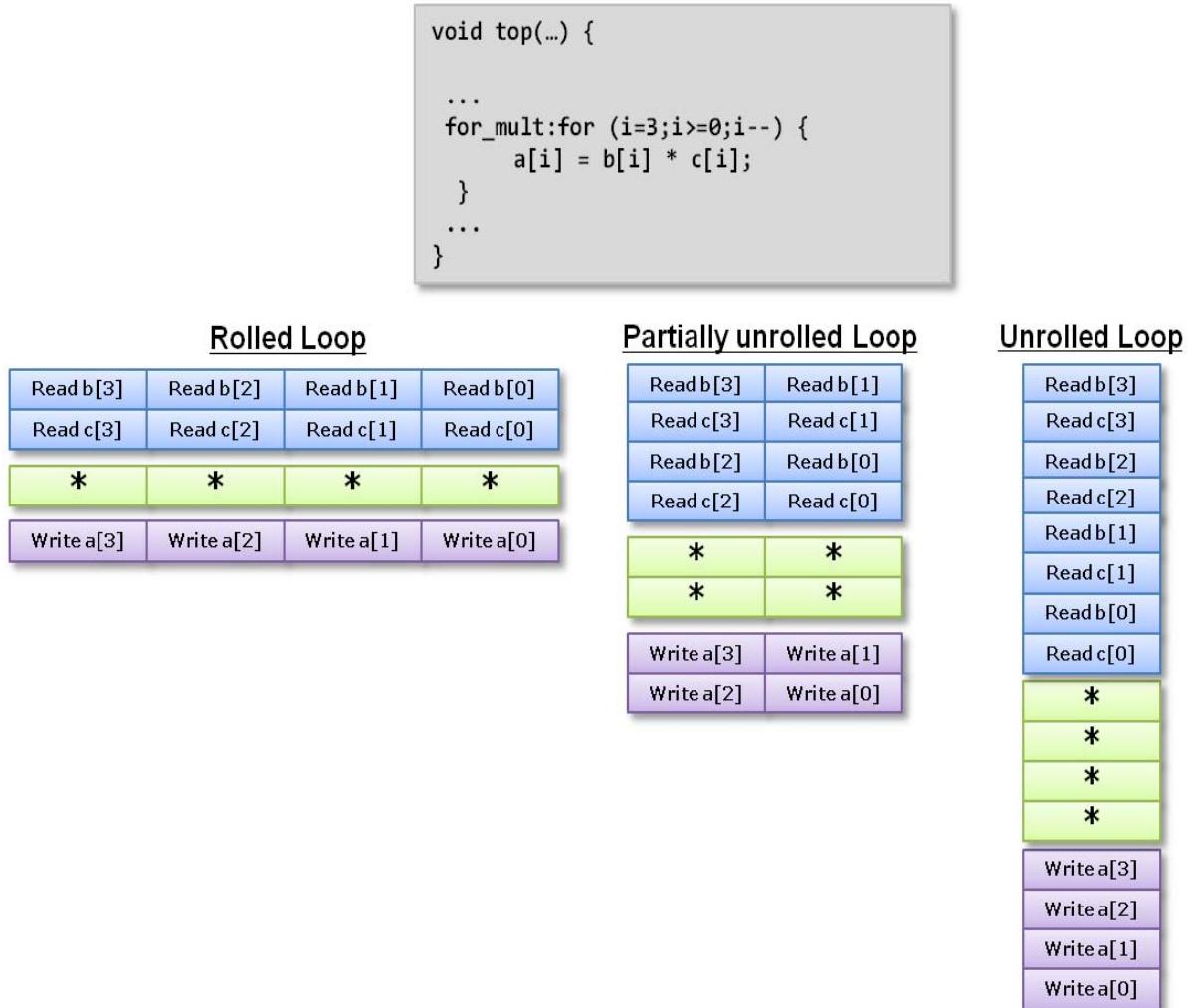


図 2-63: ループの展開

- 未展開のループ：ループが展開されていない場合、各反復は異なるクロックサイクルで実行されます。このインプリメンテーションは4クロックサイクルかかり、1つの乗算器のみが必要で、各RAMはシングルポートRAMにできます。
- 部分的に展開されたループ：この例では、ループが係数2で部分的に展開されています。このインプリメンテーションには、各RAMに対して2つの読み出しありは書き込みを同じクロックサイクルで実行するため乗算器2つとデュアルポートRAMが必要ですが、2クロックサイクルで完了し、未展開のループと比較するとスループットは2倍、レイテンシは1/2になります。
- 完全に展開されたループ：ループ操作全体を完全に展開した例は、1クロックサイクルで実行できますが、乗算器が4つ必要で、4つの読み出し操作と4つの書き込み操作を同じクロックサイクルで実行する必要があります。クロックポートRAMは一般的ではないので、配列をRAMではなくレジスタの配列でインプリメントするか、配列の分割または配列の構成の変更が必要になります。

配列のインプリメンテーション方法（一部またはすべてをRAMにマップ）、乗算器の遅延によっては、この単純な例にはほかにも多数のインプリメンテーションが可能です。

ループの展開は、GUIの図2-56に示す[Vivado HLS Directive Editor]ダイアログボックスで[Directive]ドロップダウンリストから[UNROLL]を選択して、個々のループに指示子を適用します。または、関数自体にUNROLL指示子を

設定し、関数内のすべての for ループに適用できます(図 2-56)。特定のループを展開するには、次の Tcl コマンドを使用します。

```
set_directive_unroll -skip_exit_check -factor 2 top/for_mult
```

set\_directive\_unroll コマンドは、指示子をプログラマとしてソースコードに挿入している場合以外は、ラベルの付けられたループにのみ適用可能です(図 2-63)。指示子をプログラマとしてソースコードに挿入すると、コードのすべてのバージョンに適用されます。

ループを部分的に展開する場合、展開係数を最大反復回数の整数倍にする必要はありません。Vivado HLS では、部分的に展開されたループが元のループと同じように動作することを確認するチェックが追加されます。たとえば、次のようなコードがあるとします。

```
for(int i = 0; i < N; i++) {
 a[i] = b[i] + c[i];
}
```

ループを係数 2 で展開すると、RTL にインプリメントしたときに機能が同じになるように、コードが break コンストラクトを使用した次のようなものになります。

```
for(int i = 0; i < N; i += 2) {
 a[i] = b[i] + c[i];
 if (i+1 >= N) break;
 a[i+1] = b[i+1] + c[i+1];
}
```

N は変数なので、Vivado HLS で最大値を特定できない場合があります(入力ポートで駆動されている可能性あり)。展開係数(この例の場合は 2)が最大反復数 N の整数倍であることがわかっている場合は、-skip\_exit\_check オプションを使用して終了時のチェックを削除できます。展開の結果は、次のようにになります。

```
for(int i = 0; i < N; i += 2) {
 a[i] = b[i] + c[i];
 a[i+1] = b[i+1] + c[i+1];
}
```

これによりエリアが最小限に抑えられ、制御ロジックが簡略化されます。

## C++ クラスのループの展開

ループを C++ クラスで使用する場合、ループ帰納変数がクラスのデータ メンバーにならないように注意する必要があります。ループ帰納変数がクラスのデータ メンバーになると、ループを展開できなくなります。

この例では、ループ帰納変数 k がクラス foo\_class のメンバーです。

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
 pe_mac<T0, T1, T2> mac;
public:
 T0 areg;
 T0 breg;
 T2 mreg;
 T1 preg;
 T0 shift[N];
 int k; // Class Member
 T0 shift_output;
 void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
 {
 Function_label0:;
 #pragma AP inline off
```

```

SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll// Loop will fail UNROLL
 if (k > 0)
 shift[k] = shift[k-1];
 else
 shift[k] = data;
}

*dataOut = shift_output;
shift_output = shift[N-1];
}

*pcout = mac.exec1(shift[4*col], coeff, pcin);
};

```

UNROLL プラグマ指示子で指定したように Vivado HLS でループを展開できるようにするには、コードを記述し直して k をクラス メンバーからはずす必要があります。

```

template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
 pe_mac<T0, T1, T2> mac;
public:
 T0 areg;
 T0 breg;
 T2 mreg;
 T1 preg;
 T0 shift[N];
 T0 shift_output;
 void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
 {
 Function_label0:;
 int k; // Local variable
#pragma AP inline off
 SRL:for (k = N-1; k >= 0; --k) {
#pragma AP unroll// Loop will unroll
 if (k > 0)
 shift[k] = shift[k-1];
 else
 shift[k] = data;
 }

 *dataOut = shift_output;
 shift_output = shift[N-1];
 }

 *pcout = mac.exec1(shift[4*col], coeff, pcin);
};

```

## ループの結合

展開されたループからは、少なくとも 1 つのステートを持つ有限ステート マシン (FSM) が作成されます。複数のシーケンシャルループがある場合、これにより余分なクロック サイクルが追加され、これ以上の最適化ができなくなります。

図 2-64 に、簡潔に見えるコード記述が RTL デザインのパフォーマンスに悪影響を与える例を示します。

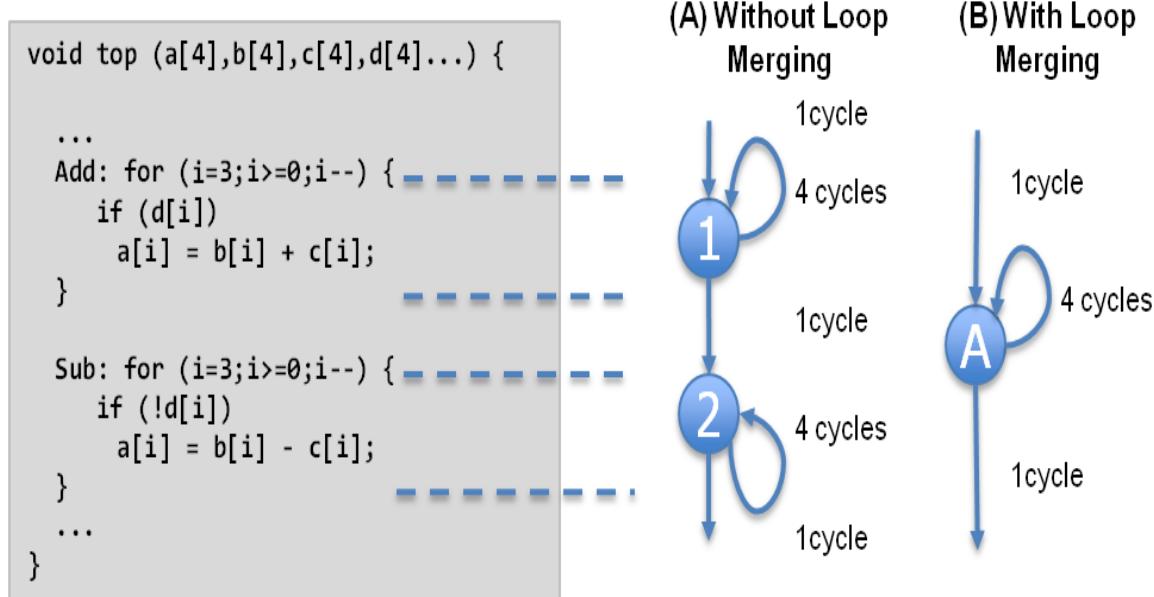


図 2-64: ループの結合

図 2-64 (A) は、デフォルトでは展開された各ループから、FSM3 に最低 1 つ(実行される操作の回数によりこれ以上)のステートが作成されます。これらのステート間の遷移により、クロック サイクルが浪費されます。各ループの反復に 1 クロック サイクルかかるとすると、両方のループを実行するのに 11 サイクルかかります。

- Add ループに入るのに 1 クロック サイクル
- Add ループを実行するのに 4 クロック サイクル
- Add ループを抜けて Sub ループに入るのに 1 クロック サイクル
- Sub ループを実行するのに 4 クロック サイクル
- Sub ループを抜けるのに 1 クロック サイクル
- 合計 11 クロック サイクル

この単純な例では、Add ループに else 分岐を追加することにより問題を解決できることは簡単にわかりますが、より複雑なコードではそう簡単にはわからない可能性もあり、同じデザインチームのほかの設計者が複雑なアルゴリズムを簡単に理解できるように、よりわかりやすいコード記述を使用した方が有益かもしれません。同じ結合操作を実行するのに、複数の if-else 文を追加するなど、文を追加すると、コードが理解しにくくなる可能性があります。

Vivado HLS には、ループを自動的に結合する機能があります。関数(または両方のループを含む領域)に対して、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [MERGE] を選択すると、Vivado HLS でループが結合され、図 2-64 (B) に示す 6 クロック サイクルで完了するものと同様の制御構造が作成されます。

現在のところ、Vivado HLS のループの結合には次の制限があります。

- ループの範囲がすべて変数である場合は、同じ値である必要があります。
- ループの範囲が定数である場合は、最大定数値が結合されたループの範囲として使用されます。
- 変数範囲と定数範囲の両方を使用するループは結合できません。
- 結合されるループの間のコードによる影響がないことを確認します。つまり、このコードを複数回実行したときに、同じ結果が得られるようにします。たとえば、 $a=b$  は使用できますが、 $a=a+1$  は使用できません。

- ループに FIFO の読み出しが含まれている場合は結合できません。FIFO からの読み出しありは FIFO インターフェイスは常に正しい順序である必要がありますが、FIFO の読み出しが含まれているループを結合すると、読み出しの順序が変更されます。

ループの結合は、コマンド ラインで `set_directive_loop_merge` コマンドを使用するか、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [MERGE] を選択します。

```
set_directive_loop_merge top
```

## ネストされたループのフラット化

前のセクションで説明した連続するループと同じように、展開されていないネストされたループ間を移動するのにも追加のクロック サイクルが必要です。外側のループから内側のループに、内側のループから外側のループに移動するのに 1 クロック サイクルかかります。

次に示す例では、ループ Outer を実行するのに 200 クロック サイクル余分にかかることがあります。

```
void foo_top { a, b, c, d} {
 ...
 Outer: while(j<100)
 Inner: while(i<6)// 1 cycle to enter inner
 ...
 LOOP_BODY
 ...
 } // 1 cycle to exit inner
}
...
}
```

また、ループがネストされていると、「ループのデータフロー パイプライン処理」に説明するように、外側のループをパイプライン処理できません。

Vivado HLS では `set_directive_loop_flatten` コマンドが提供されており、ラベル付きの完全なネスト ループおよびほぼ完全なネスト ループを自動的にフラット化でき、最高のハードウェアパフォーマンスを得るためにコードを記述し直す必要はなく、ループ内の操作を実行するのにかかるサイクル数を削減します。

- 完全なループ ネスト：最内側のループのみにループ本体があり、ループ文の間にロジックは指定されておらず、すべてのループ範囲が定数
- ほぼ完全なループ ネスト：最内側のループのみにループ本体があり、ループ文の間にロジックは指定されていないが、最外側ループの範囲が変数

内側のループの範囲が変数であったり、ループ本体が最内側のループにのみ含まれているとは限らない不完全ループ ネストは、コードの構造を変更するか、ループ本体の中のループを展開して、完全ループ ネストを作成してみてください。

ネストされた ループに指示子を適用する際は、ループ本体を含む最内側のループに適用する必要があります。

```
set_directive_loop_flatten top/Inner
```

ループのフラット化は、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスを使用しても指定できます。個々のループに、または関数レベルで指示子を適用して関数に含まれるすべてのループに設定できます。

## ループのデータフロー パイプライン処理

ループにも、関数と同様にデータフロー パイプライン処理を適用できます。データフロー パイプライン処理を適用すると、通常順次に実行されるループを RTL で同時実行できます。データフロー パイプライン処理は、関数、ループ、あるいは関数またはループのみを含む領域に適用する必要があります。ループと関数が混合している領域には適用しないでください。

図 2-65 に、ループにデータフロー パイプライン処理を適用した場合の利点を示します。

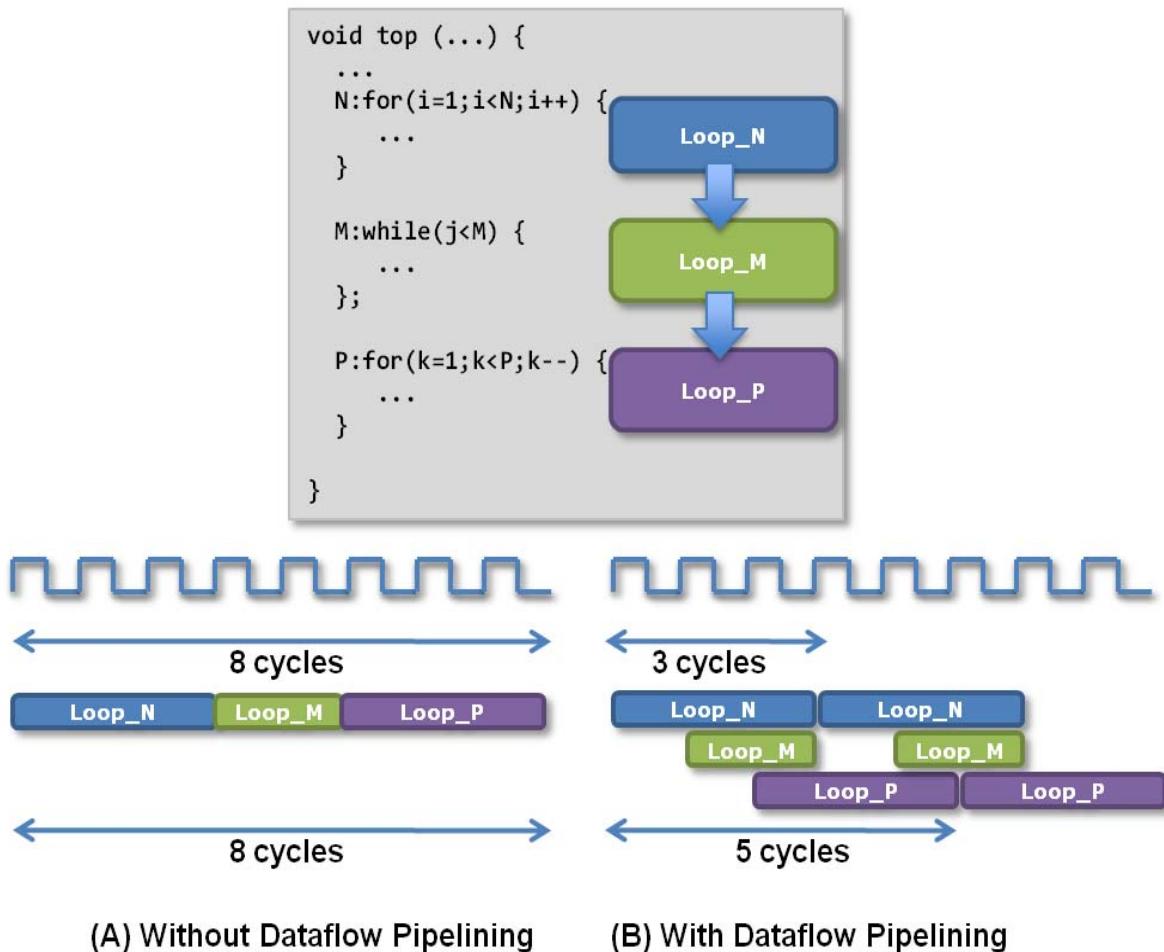


図 2-65 : ループのデータフロー パイプライン処理

データフロー パイプライン処理を適用しない場合、ループ N のすべての反復が実行されて完了するまで、ループ M は開始できません。同様に、ループ P もループ M が完了するまで実行できません。この例では、ループ N が次の値の処理を開始するまでに 8 サイクル、出力が書き込まれるまでに 8 サイクルかかります (出力はループ P が完了してから書き込まれる)。

データフロー パイプライン処理を適用すると、これらのループを並列実行できるようになり、同じハードウェアリソースを使用して、新しい値を 3 サイクルごとに処理でき、出力は 5 サイクルごとに書き込まれます。Vivado HLS では、ループの間に自動的にチャネルが挿入され、データが 1 つのループから次のループに非同期に流れるようにします。

ループ間のチャネルは、ピンポン バッファーまたは FIFO としてインプリメントされます。

- 変数が配列の場合、チャネルはピンポン バッファーとして、標準メモリ アクセスを使用してアドレス信号と制御信号と共にインプリメントされます。
- その他すべての変数型およびストリーミング配列の場合は、チャネルは FIFO としてインプリメントされます。この場合、使用されるハードウェアリソースは少なくなりますが (アドレスは生成されない)、データにシーケンシャルにアクセスする必要があります。

**重要:** データフロー パイプライン処理を使用するには、変数が 1 つのループで生成され、別の 1 つのループでのみ使用されるようにする必要があります。



データフロー パイプライン処理は、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [DATAFLOW] を選択するか、`set_directive_dataflow` コマンドを使用します。領域に対してこの指示子を設定すると、Vivado HLS でその領域内のループの並列処理を向上するよう処理されます。[図 2-65](#) に示す例では、次のコマンドを使用すると、ループ `Loop_N`、`Loop_M`、および `Loop_P` に対してデータフロー パイプライン処理が実行されます。

```
set_directive_dataflow -interval 2 top
```

`-interval` オプションを使用すると、1 つのループ インプリメンテーションが開始してから次のループ インプリメンテーションが開始するまでのサイクル数を指定できます。デフォルトでは、この時間を最小限に抑えるように試みられます。すべてのループを同じクロック エッジで開始してすべてを並列に実行するのが理想的ですが、データの依存性によりこれは通常不可能です。たとえば、`-interval` を 3 に設定すると、[図 2-65](#) の (B) で各ループの開始間隔は 3 サイクルになります。

チャネル(メモリ)の要素数は受信側配列または送信側配列の最大サイズで定義されますが、Vivado HLS ではデフォルトのチャネル ワード数を指定できます(「デフォルトのチャネルの設定」を参照)。

### デフォルトのチャネルの設定

ループ間で使用されるデフォルトのチャネルは、`config_dataflow` コマンドを使用して指定できます。このコマンドを使用すると、ソリューションのデフォルト操作を設定できます。デザインのすべてのチャネルに対するデフォルトのチャネル サイズおよびインプリメンテーションを指定可能です。

```
config_dataflow -default_channel (fifo | *pingpong*) -size <FIFO size>
```

チャネルのサイズは、受信側配列または送信側配列の最大サイズで定義されます。場合によっては、これは控えめすぎる設定があります。`-size` オプションを使用すると、このデフォルトを変更できます。

配列パラメーターが FIFO チャネルを使用するよう指定されている場合、配列は自動的にストリーミング型配列に変換されます。ストリーミングについては、「配列のストリーミング」を参照してください。デフォルトのチャネル タイプが FIFO で、`set_directive_array_stream` コマンドを使用して配列がストリーミング以外に設定されている場合、その配列のチャネル インプリメンテーションはデフォルトでピンポン チャネルになります(明示的なディレクトリによりコンフィギュレーションが変更される)。

## ループのパイプライン処理

C 言語記述では、ループ内の操作は順次実行され、ループ内の最後の操作が完了するまで次の反復を実行できません。RTL デザインでは複数の操作を同時に実行でき、RTL をそのようにインプリメントすることが望されます。

ループのパイプライン処理では、[図 2-66](#) に示すように、ループが同時処理されるようにインプリメントされます。[図 2-66](#) (A) には、デフォルトの順次操作を示します。各入力は 3 クロック サイクルごとに処理され、最後の出力が書き込まれるまでに 8 クロック サイクルかかります。

[図 2-66](#) (B) に示すパイプライン処理されたループでは、入力サンプルが各クロック サイクルで読み出され、最終的な出力は 4 クロック サイクル後に書き込まれます。デザインの変更は制御ロジックのみに加えられるので、同じハードウェア リソースでスループットとレイテンシの両方を向上できます。

入力読み出し間のクロック サイクル数はパイプライン開始間隔と呼ばれ、ユーザーが指定できます。デフォルトでは 1 に設定されます。

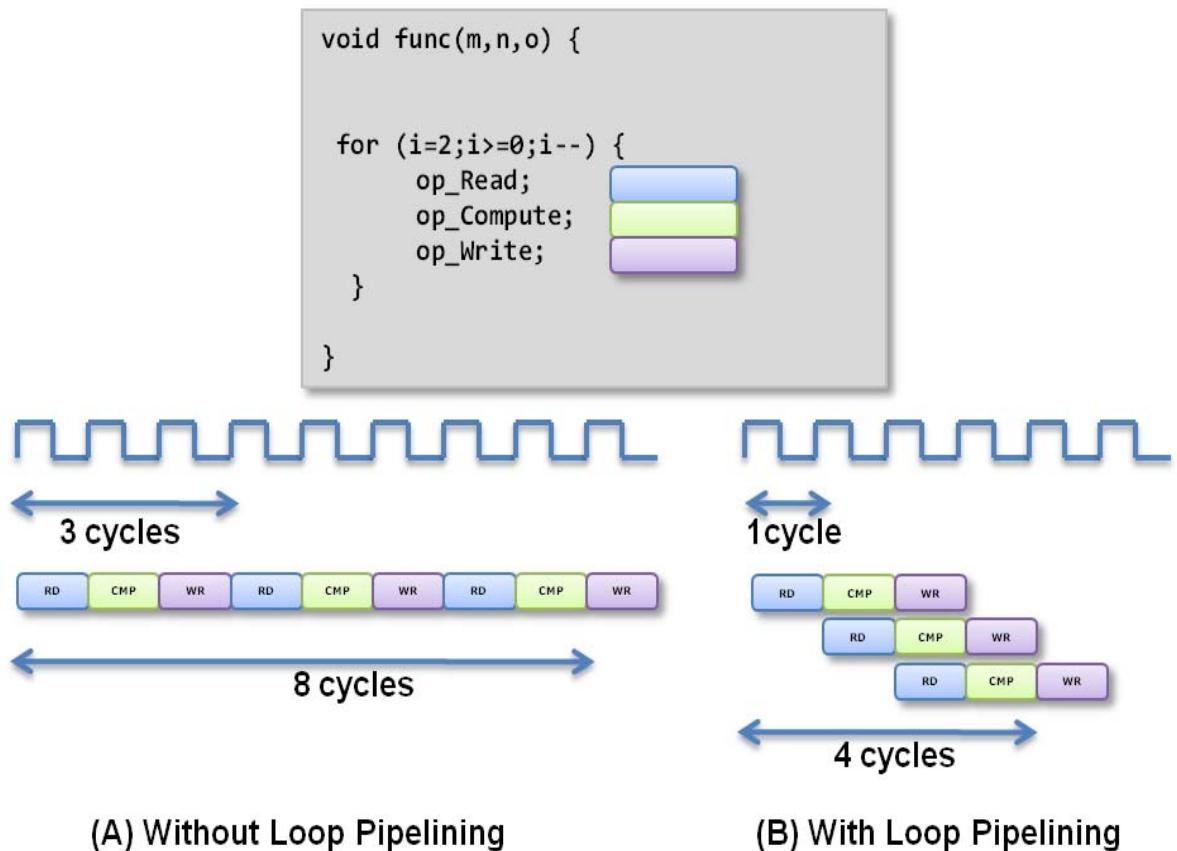


図 2-66: ループのパイプライン処理

ループのパイプライン処理が、図 2-61 に示したように、リソースの競合およびデータの依存性のため、実行できない場合があります。

ネストされたループでは、最内側のループのみをパイプライン処理できます。ただし、パイプライン処理されると、パイプライン処理される領域の下位の階層にあるループはすべて自動的にフラット化されます。

データの依存性は解消するのがより困難で、通常ソース コードの変更が必要になります。スカラー データの依存性は、次のようにになります。

```

while (a != b) {
 if (a > b) a -= b;
 else b -= a;
}

```

このループの次の反復は、現在の反復が完了して `a` と `b` の値がアップデートされるまで開始できません (図 2-67)。



図 2-67: スカラー データの依存性

データの依存性は、メモリアクセスでもよく見られます。

```
for (i = 1; i < N; i++)
 mem[i] = mem[i-1] + i;
```

この場合、ループの次の反復は、現在の反復が完了して配列の内容がアップデートされるまで開始できません（[図 2-68](#)）。



図 2-68: メモリ アクセスの依存性

ループの反復を始めるのに前のループ反復の結果が必要な場合、ループのパイプライン処理は不可能です。Vivado HLS では、指定の開始間隔でパイプライン処理できない場合、自動的に開始間隔が増加され、結果的に、次の反復が現在の反復に引き込まれて依存性が削除されます。[図 2-67](#) および [図 2-68](#) に示すようにパイプライン処理がまったく不可能な場合は、パイプライン処理が停止され、スループットの低いパイプライン処理されていないデザインが 출력されます。

ループのパイプライン処理を指定するには、指定したループに対して GUI の [Vivado HLS Directive Editor] ダイアログボックスで [Directive] ドロップダウンリストから [PIPELINE] を選択するか、ラベル付きのループに対して次の Tcl コマンドを使用します。

```
set_directive_pipeline -II 5 -enable_flush foo/sum_loop
```

この例では、パイプライン処理の開始間隔が 5 に設定されており、新しい入力が 5 クロック サイクルごとに読み出され、フラッシュがイネーブルになっています。フラッシュをイネーブルにすると、パイプラインが閉鎖して空にできるように構成されます。デフォルトでは、フラッシュはディスエーブルになっており、デザインがリセットされるまで継続的に実行されるパイプラインが作成されます。

## ループ キャリー依存性

「ループのパイプライン処理」に説明するように、ループ キャリー依存性によりループをパイプライン処理できない場合があります。ただし、状況によっては、自動依存性解析で、依存性があるように見えるが実際には依存性がないようなものを除去できない場合があります。

たとえば、次のように入力します。

```
void foo(int A[3*N], int x)
{
 LF: for (i = 0; i < N; i++)
 A[i+x] = A[i] + i; // User knows that 2*N > x >= N
}
```

この例では、Vivado HLS で入力パラメーター  $x$  の範囲を判断できず、 $A[i+x]$  への書き込みと  $A[i]$  からの読み出しに常に依存性があると判断され、ループ反復を順次スケジューリングします。

この依存性を解消するには、DEPENDENCE 指示子を指定して、1つまたは複数の変数のループ キャリー依存性に関する情報を入力します。上記の例では、 $x$  が  $N$  以上  $2*N$  以下であることがわかっているので、ループ キャリー依存性はないということをツールに指示できます。

```
set_directive_dependence -variable x -type inter -dependent false foo/LF
```

依存性を指定する場合、主に二種類あります。

- 反復間：同じループの異なる反復の間の依存性を指定します。このタイプの依存性を `false` に設定すると、ループが展開されていない場合または部分的に展開されていない場合に並列実行が可能になり、`true` に設定すると並列実行は不可能になります。
- 反復内：ループの同じ反復内での依存性を指定します。たとえば、同じ反復の最初と最後に配列にアクセスするような場合です。このタイプの依存性を `false` に設定すると、Vivado HLS によりループ内で操作を自由に移動でき、パフォーマンスまたはエリアを向上できる可能性が高くなります。`true` に設定すると、操作は指定の順序で実行する必要があります。

## ループ反復の制御

Vivado HLS では、自動的に解析が実行され、ループの可能な最大反復回数が予測されますが、実際の最大反復回数を判断することはできません。

次の例では、`for` ループの最大反復回数は `num_samples` 入力によって決まります。Vivado HLS では、この変数の最大値は 15 であると判断されますが(この変数は `uint4` 型)、`num_samples` の実際の値が、たとえばほかの要因により 8 以下であったとしても、それを判断することはできません。

```
void foo (uint4 num_samples, ...);

void foo (num_samples, ...){
 int i;
 ...
 loop_1: for(i=0;i< num_samples;i++) {
 ...
 result = a + b;
 }
}
```

デザインのレイテンシまたはスループットが変数インデックスを含むループによって決まる場合、Vivado HLS で実際の正しいスループットまたはレイテンシは算出できず、ループのレイテンシは不明(クエスチョンマーク(?))であるとレポートされます。

ループの反復回数(TRIPCOUNT)を指定すると、レポートに有効な数値が含まれます。ループの反復回数を指定するには、GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで [Directive] ドロップダウン リストから [TRIPCOUNT] を選択するか、ラベル付きのループに対して次の Tcl コマンドを使用します。

```
set_directive_loop_tripcount -min 3 -max 8 -avg 5 foo/loop_1
```

`-max` オプションはループの最大反復回数、`-min` オプションは最小反復回数、`-avg` オプションは平均反復回数を指定します。これにより、Vivado HLS でデザインの最大/最小スループットが正しくレポートされます。

`TRIPCOUNT` 指示子は、合成結果には影響しません。`TRIPCOUNT` の値はレポート目的のみに使用され、Vivado HLS でレポートに意味のあるレイテンシおよびスループットが示されるようにします。

## ループのレイテンシ

ループの最大および最小レイテンシは、制約として指定できます。これによりパフォーマンス要件が満たされることを確実にし、ループ内のリソースがどのように使用されるかを制御できます。

デフォルトでは、Vivado HLS でタイミングを満たしてからレイテンシを削減するよう処理されます。タイミングが満たされない場合、タイミングが満たされるまでレイテンシが増加されます。レイテンシ制約が設定されていてレイテンシを増加できない場合は、ローカル レイテンシ違反が許容されます（「ロック、タイミング、および RTL 出力」を参照）。デフォルト設定で達成されたレイテンシがあるとすると、次のようにになります。

- 最大レイテンシをこの値より小さい値に設定すると、Vivado HLS で低いレイテンシ値を満たすために演算子の共有やチェーン化を向上するよう試みられます。
- 最小レイテンシをこの値より大きい値に設定すると、Vivado HLS で操作を完了するクロック サイクル数が増加され、共有を増加でき、現時点で満たされていないパスのタイミングが満たされる可能性があります。

レイテンシ制約は、GUI で各ループに個別に設定するか、`set_directive_latency` コマンドを使用してループ ラベルに設定します。

---

## 配列の最適化

デザインのメモリ コンフィギュレーションは、デザイン全体のパフォーマンスおよびエリアに大きく影響します。C 言語記述の配列は通常メモリにマップされるので、配列に最適化を実行すると、エリアとパフォーマンスに大きく影響します。

配列に指定可能な指示子は、GUI にリストされます（図 2-69）。

- [Explorer] タブでソース コードをダブルクリックして開きます。
- 補足エリアで [Directive] タブをクリックします。
- 配列を右クリックして [Insert Directives] をクリックします。
- [Directive] ドロップダウン リストから適切な指示子を選択し、オプションを設定します。

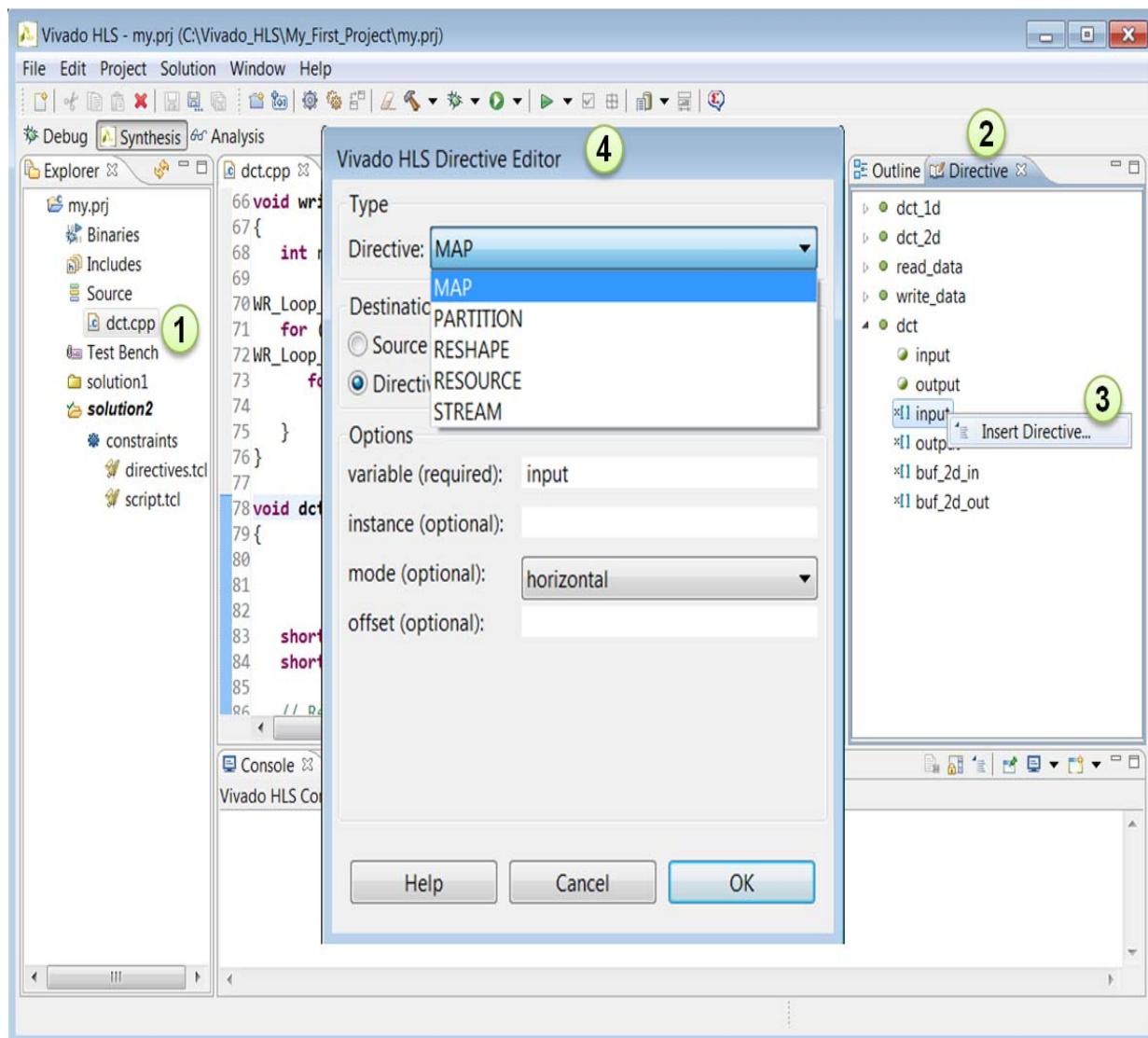


図 2-69 : 配列の指示子

表 2-10 に、配列に適用可能な最適化を、この章で説明する順に示します。

表 2-10 : 配列の最適化

| GUI 指示子   | 説明                                                                            |
|-----------|-------------------------------------------------------------------------------|
| RESOURCE  | 配列をマップするハードウェア リソース (RAM コンポーネント) を指定します。                                     |
| MAP       | 複数の小型の配列を 1 つの大型の配列に結合して配列のサイズを変更し、RAM リソースおよびエリアを削減します。                      |
| PARTITION | 大型の配列を複数の小型の配列に分割し、RAM アクセスの障害を取り除きます。配列を RAM ではなくレジスタとしてインプリメントするためにも使用されます。 |

表 2-10 : 配列の最適化

| GUI 指示子 | 説明                                                                 |
|---------|--------------------------------------------------------------------|
| RESHAPE | 配列を多数の要素を含むものからワード幅の広いものに変更します。多数の RAM を使用せずに RAM アクセスを向上するのに有益です。 |
| STREAM  | 配列を RAM ではなく FIFO としてインプリメントします。                                   |

C 言語記述の配列は、RTL では通常メモリを使用してインプリメントされます。この章では、配列を特定の RAM または ROM にインプリメントする方法、使用可能なメモリリソースに効率的にマップするため変換する方法(水平に分割、垂直に分割、これらの操作の組み合わせ)、個々のレジスタに分割する方法を説明します。

最上位関数に引数として指定されている配列は、多少異なる方法で合成されます。この場合、メモリリソースはデザインの外にあると想定され、そのリソースにアクセスするポートとして合成されます。そのような配列にも、同様の変換を実行できます。変換された配列にアクセスするポートに合成されます。

## 配列の初期化とリセット

Vivado HLS では、ソースコードでの配列の初期化がサポートされています。配列の初期値は RTL および FPGA でも複製され、ビットストリームによりデバイスに電源を投入したときに RAM(配列が分割されている場合はレジスタ)が同じ値で初期化されるように指定されます。

以下のコード例は、次のようなものです。

- 配列が `fir_coef.h` ファイルから初期化されます。
- `const` 修飾子により、配列が ROM としてインプリメントされるよう指定されています。

`const` 修飾子を使用しない場合、配列が読み出されるだけの場合は Vivado HLS で ROM にインプリメントするべきであることが自動的に判断されますが、配列をインプリメントするのにどのメモリリソースを使用するべきであるかを明示的に指定するのが理想的です。リソースの選択については、次のセクションで説明します。

```

typedef intcoef_t;
typedef intdata_t;
typedef intacc_t;

data_t fir (
data_t x
)
{
 static data_t shift_reg[N];
 acc_t acc;
 int i;

 const coef_t c[N+1]={
#include "fir_coef.h"
};

 acc=0;
 mac_loop: for (i=N-1;i>=0;i--) {
 if (i==0) {
 acc+=x*c[0];
 shift_reg[0]=x;
 } else {
 shift_reg[i]=shift_reg[i-1];
 acc+=shift_reg[i]*c[i];
 }
 }
}

```

```
 return=acc;
}
```

配列は、電源投入時にソースコードで指定された値に初期化されますが、その後デバイスにリセットを適用しても、電源等乳児のステートには初期化されません。Vivado HLS で追加されるリセットを使用した配列のリセットは、サポートされません。

配列を初期(リセット)ステートに戻す必要がある場合は、外部信号を供給する必要があります。これは、コードに明示的に記述します。このコード例は、次のとおりです。

```
...
 const coef_t c_temp[N+1] ={
#include "fir_coef.h"
 };
 coef_t c[N+1];

 if (rst_array==1) {
 for (i=0;i<N;i++) {
 c[i] = c_temp[i];
 }
 }
...
```

このコードにより、RTL デザインが次のようになってしまう可能性があります。

- 電源投入時の初期化とは異なり、このような明示的なリセットでは RTL デザインで配列/RAM 内の各アドレスに反復的に値を設定する必要があります、N が大きい場合は多数のクロックサイクルがかかります。
- 最上位関数インターフェイスに追加の変数 `rst_array` が必要です。このポートは、合成中に追加されるリセット信号のほかに追加されます。

## メモリリソースの選択

配列にメモリリソースが指定されていない場合、Vivado HLS により使用するメモリリソース(シングルポート、デュアルポートなど)が自動的に判断されます。これは、最上位関数の関数引数に指定された配列でも同じです。Vivado HLS で、スループットが大きくなるという理由でデュアルポートメモリへのインターフェイスが作成されることがあります。これが必ずしも最適であるとはかぎらないので、各配列をマップするメモリリソースを明示的に指定することをお勧めします。

配列を特定の RAM リソースにマップするには、次の例のように `set_directive_resource` コマンドを使用するか、GUI の [Vivado HLS Directive Editor] ダイアログボックスの [Directive] ドロップダウンリストから [RESOURCE] を選択するか(図 2-69)、コードにプログラマを挿入します。

次の例には 3 つの配列が含まれており、1 つは関数パラメーターとして、2 つは関数内で定義されています。

```
void foo (in[16], ...){
 int8 array1[16];
 int12 array2[48];
 ...
loop_1: for(i=0;i<8;i++) {
 array1[i] = in[i];
 ...
}
}
```

## インターフェイスリソース

次を使用すると、データをポート `in[16]` に供給する RAM のタイプが指定されます。

```
set_directive_resource -core RAM_1P foo in
```

-core オプションは、RAM コアを指定します。使用可能な RAM コアのリストは、「高位合成演算子およびコア ガイド」の章にリストされており、[Vivado HLS Directive Editor] ダイアログ ボックスで選択できます。

パラメーター in[16] のインターフェイスが ap\_memory に指定されている場合、作成されるポートは RAM\_1P 上のポートに一致します。RAM\_1P にチップ イネーブル (CE) ポートがある場合、CE ポートを持つインターフェイスが作成されます。RAM\_1P に読み出し用と書き込み用に別のアドレス ポートがある場合は、RAM を読み出すアドレス ポートと RAM に書き込むアドレス ポートが作成されます。読み出しと書き込みの両方にアドレス ポートが 1 つ使用される場合は、読み出しと書き込みの両方に使用されるアドレス ポートが 1 つ作成されます。

ポート in[16] が ap\_fifo に指定されている場合、ap\_fifo は常に同じなので(読み出しおよび書き込み用のデータ ポート、Empty ポート、Full ポート)、メモリ リソース タイプはそれほど重要ではありません。

インターフェイスの選択の詳細は、「インターフェイスの管理」の章を参照してください。

## デザイン リソース

上記の同じコード例を使用した場合、次のコマンドにより配列 array1 および array2 をインプリメントするのに使用される RAM のタイプが指定されます。

```
set_directive_resource -core RAM_1P foo array1
set_directive_resource -core RAM_2P foo array2
```

この場合、array1 は RAM\_1P コアに、array2 は RAM\_2P コアにマップされます。この時点で、次を満たす必要があります。

- RAM\_1P では、int8 は 8 ビット データ型なので、要素(アドレス)が 17 個以上、各要素が 9 ビット以上である必要があります。
- RAM\_2P では、int12 は 12 ビット データ型なので、要素(アドレス)が 28 個以上、各要素が 13 ビット以上である必要があります。

ポート in1 が関数外の RAM\_1P コンポーネントと通信するよう指定されていたとしても、array1 を同じタイプの RAM コンポーネントにマップするという要件はありません。Vivado HLS により、あるタイプの RAM または RAM ポートから読み出し、別のタイプの RAM または RAM ポートに書き出すために必要な変換が実行されます。

## 配列のマップ

ほとんどのテクノロジ ライブラリでは、RAM のサイズはあらかじめ定義されています(2 のべき乗のワード数、1、8、16 ビット ワードなど)。元の仕様に小型の配列が多数ある場合は、ターゲット リソースを指定する前にこれらの配列を 1 つの大型の配列にまとめると、ストレージのオーバーヘッドを削減できます。小型の配列それが個別のメモリにマップされると、多数のメモリ空間が無駄になり、デザインが必要以上に大きくなる可能性があります。

Vivado HLS の set\_directive\_array\_map コマンドでは、複数の小型の配列を 1 つの大型の配列にマップする 2 つの方法がサポートされています。

- **水平マップ** : 元の配列を連結して、新しい配列を作成します。物理的には、要素数の多い 1 つの配列にインプリメントされます。
- **垂直マップ** : 元の配列のワードを連結して、新しい配列を作成します。物理的には、ビット幅の広い 1 つの配列にインプリメントされます。

### 水平マップ

次のコード例には 2 つの配列が含まれており、2 つの RAM コンポーネントにマップされます。

```
void foo (...) {
 int8 array1[M];
 int12 array2[N];
```

```

...
loop_1: for(i=0;i<M;i++) {
 array1[i] = ...;
 array2[i] = ...;
 ...
}
...
}

```

配列 array1 および array2 は、次の例のように array3 という配列に結合できます。

```

set_directive_array_map -instance array3 -mode horizontal foo array1
set_directive_array_map -instance array3 -mode horizontal foo array2

```

MAP 指示子は、個々の配列に対して GUI の [Vivado HLS Directive Editor] ダイアログ ボックスで指定することもできます。これにより、配列が 図 2-70 に示すように変換されます。

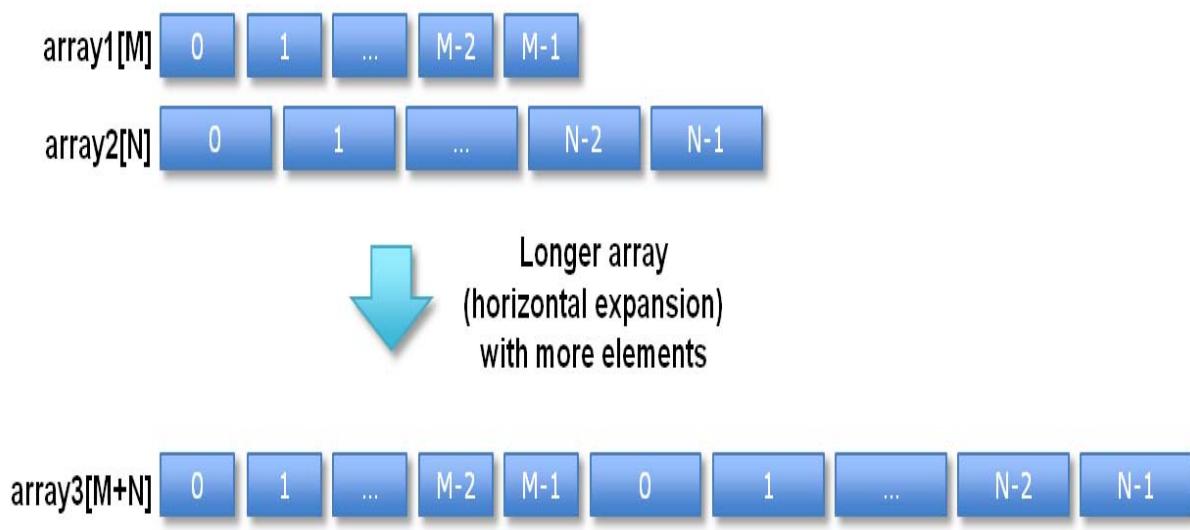


図 2-70: 水平マップ

水平マップを使用すると、複数の小型の配列を 1 つの大型の配列にロケーション 0 からコマンドの順序でマップされます (GUI では、配列を指定した順序)。-offset オプションを使用すると、配列間に要素を追加できます。

上記の例で `array2` を追加してから `array1` を追加し、オフセットを追加する場合は、次のコマンドを使用します。

```

set_directive_array_map -instance array3 -mode horizontal foo array2
set_directive_array_map -instance array3 -mode horizontal -offset 2 foo array1

```

結果は図 2-71 に示すようになります。

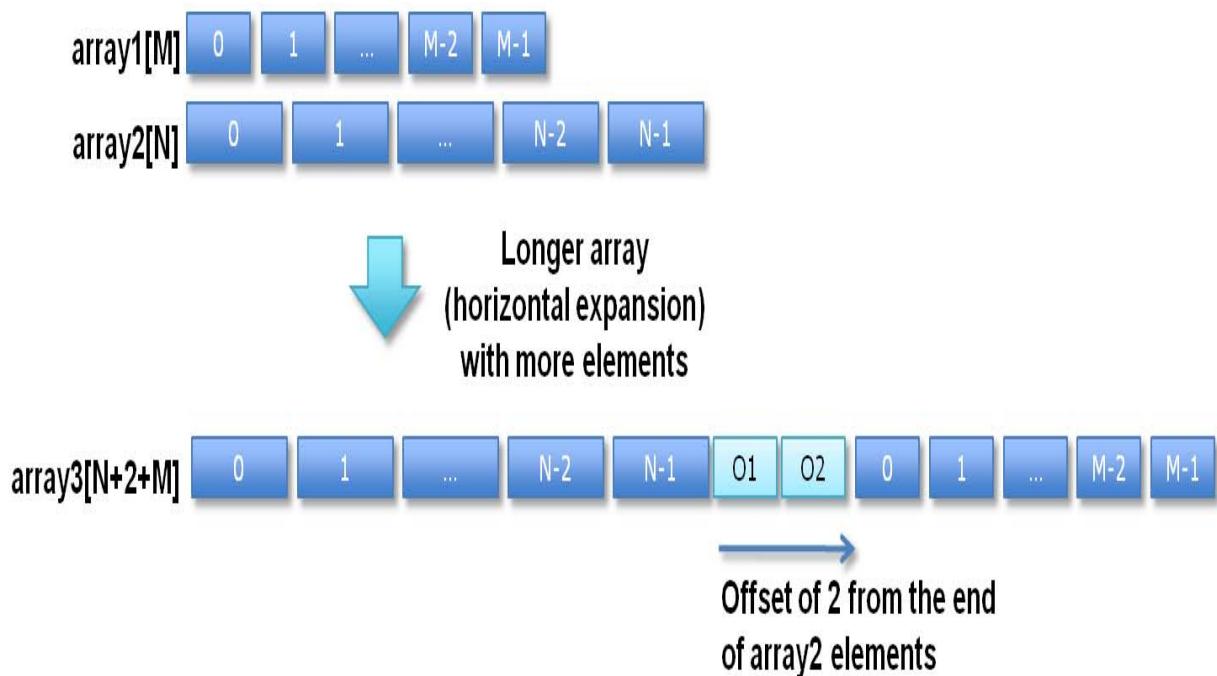


図 2-71: オフセットを使用した水平マップ

上記の例で新しい配列  $\text{array3}$  がマップされたら、1 つの RAM コンポーネントにマップできます。

```
set_directive_resource -core RAM_1P foo array3
```

図 2-72 に示す RAM インプリメンテーションは、オフセットを使用しない図 2-70 のマップに対応します。

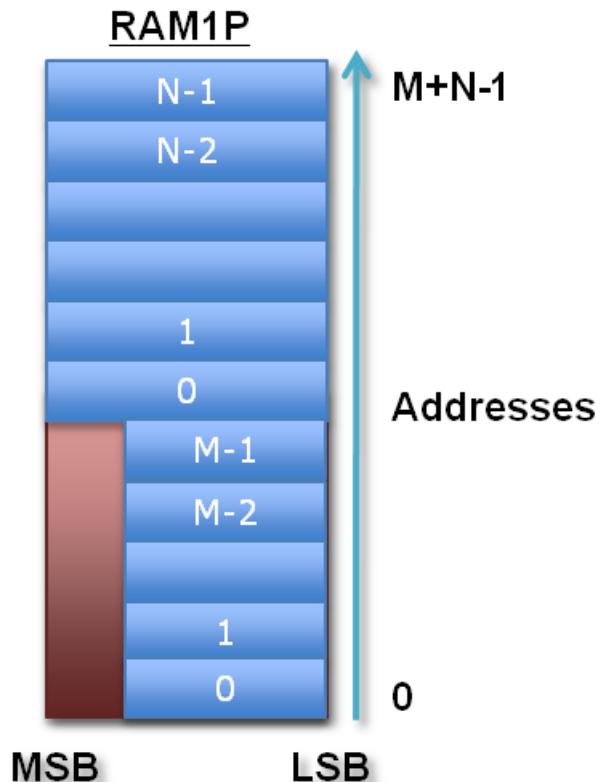


図 2-72 : 水平マップのメモリ

水平マップにより RAM コンポーネントの数を削減でき、エリアが向上しますが、スループットおよびパフォーマンスに影響する可能性があります。上記の例では、loop\_1 の array1 および array2 へのアクセスを同じクロック サイクルで実行できますが、両方の配列が同じ RAM にマップされると、それぞれの読み出しに個別のアクセスとクロック サイクルが必要になります。

この制限を解消するため、Vivado HLS では垂直マップがサポートされています。

### 垂直マップ

垂直マップでは、ビット幅が広い配列が生成されるよう配列が連結されます。図 2-73 に、水平マップの説明で使用した例に垂直マップを適用した結果を示します。

```
set_directive_array_map -instance array3 -mode vertical foo array2
set_directive_array_map -instance array3 -mode vertical foo array1
```

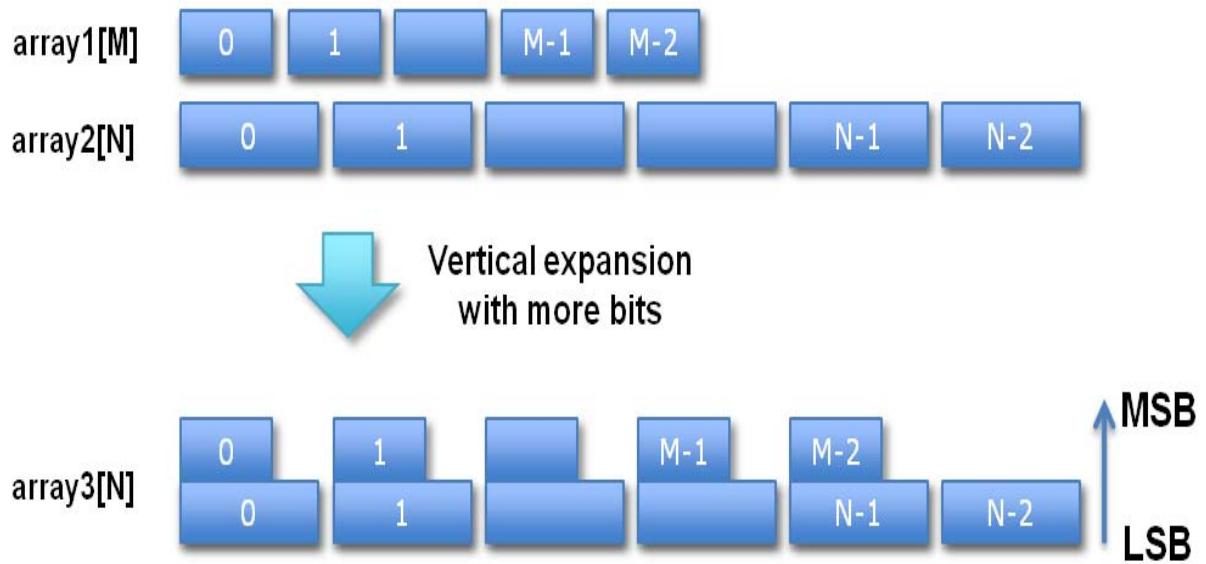


図 2-73: 垂直マップ

垂直マップでは、コマンドで指定された順に配列がマップされ、最初の配列が LSB から開始し、最後の配列が MSB で終了します。上記の `set_directive_array_map` コマンドでは、array2 が先に指定されています。Vivado HLS の GUI では、配列を指定した順序になります。

上記の例で新しい配列 array3 がマップされたら、1 つの RAM コンポーネントにマップできます (図 2-74)。

```
set_directive_resource -core RAM_1P foo array3
```

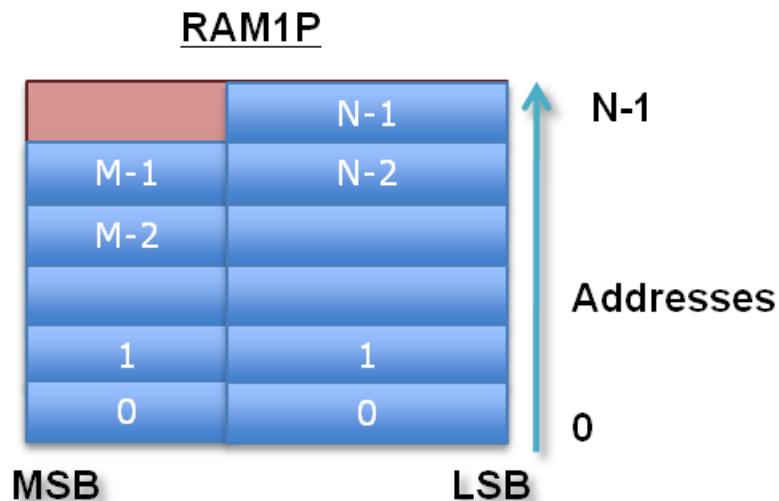


図 2-74: 垂直マップのメモリ

## マップおよびグローバル配列

グローバル配列もマップできますが、結果の配列インスタンスはグローバルになり、同じ配列インスタンスにマップされたローカル配列もすべてグローバルになります。

異なる関数のローカル配列が同じターゲット配列にマップされると、そのターゲット配列インスタンスはグローバルになります。

配列関数パラメーターをマップする場合、同じ関数のパラメーターである必要があります。

## 配列の分割

配列を小型の配列に分割できます。メモリの読み出しポートと書き込みポートの量は限られているので、ロード/ストアが多数実行されるアルゴリズムでは、スループットが制限されます。元の配列(1つのメモリリソース)を複数の小型の配列(複数のメモリ)に分割することにより、ポート数が増加し、バンド幅を向上することができます。

そのため、`set_directive_array_partition` コマンドを使用して1つの大型の配列を複数の小型の配列に分割すると、スループットを向上できます。

Vivado HLS には、3 タイプの配列分割があります(図 2-75)。さまざまなタイプを `-type` オプションで指定できます。

- **block** : 元の配列の連続したエレメントが同じサイズのブロックに分割されます。
- **cyclic** : 元の配列の交互のエレメントが同じサイズのブロックに分割されます。
- **complete** : デフォルトでは配列が個別エレメントに分割されます。1つのメモリは複数のレジスタに分解されます。

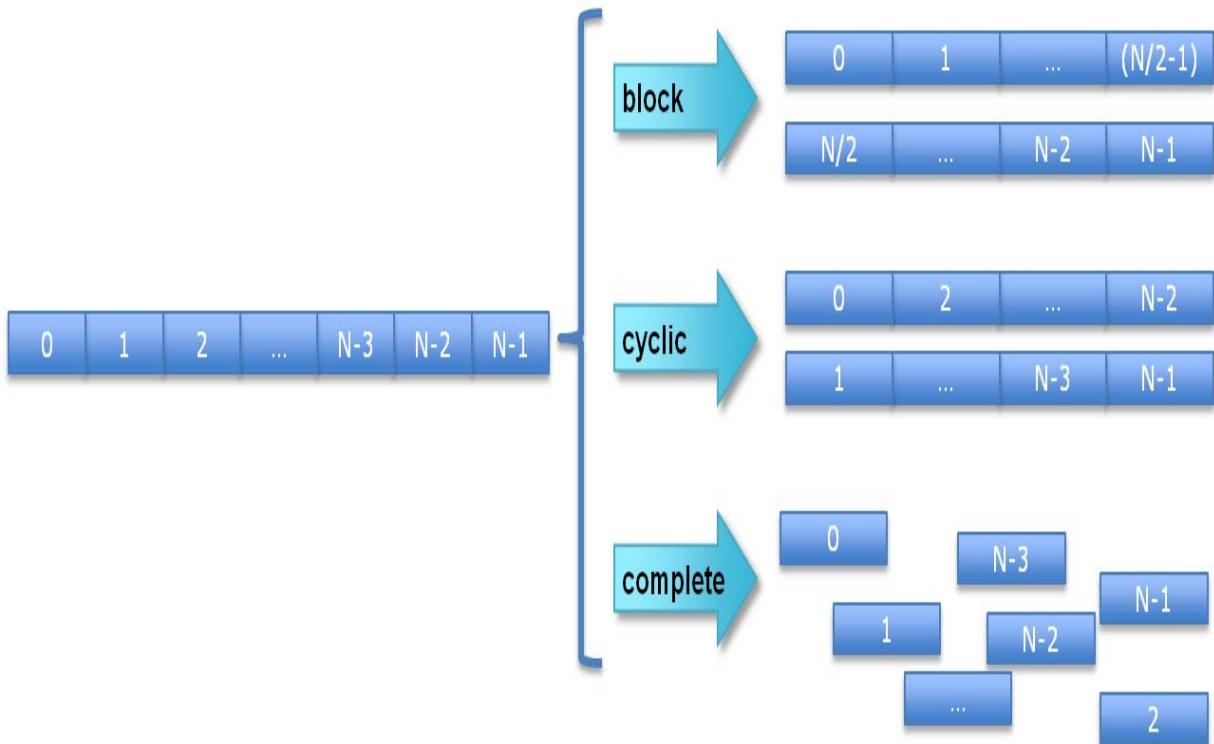


図 2-75: 配列の分割

block および cyclic 分割の場合は、-factor を使用すると作成される配列数を指定できます。図 2-75 では、-factor オプション 2 が使用され、配列が 2 つの小さい配列に分割されています。配列のエレメント数がこの係数の整数倍ではない場合、最終的な配列に含まれるエレメントは少なくなります。

多次元配列を分割する際は、-dim オプションを使用すると、どの次元を分割するか指定できます。

```
void foo (...){
 int array1[L][M][N];
 ...
}
```

array1 はそれぞれが 1 次元の L/3 サイズの 3 つの配列に分けられます。

```
set_directive_array_partition -type block -dim 1 -factor 3 foo array1
```

次元に 0 が指定されると (-dim 0)、すべての次元が分割されます。

## 配列の変更

set\_directive\_array\_reshape コマンドを使用すると、垂直マップで配列分割をまとめることができます。これにより元の配列の 1 つの次元から異なるエレメントを取り出し、1 つのエレメントにまとめて変更した配列に入れることができます。

次のようなコードがあるとします。

```
void foo (...){
 int array1[N];
 int array2[N];
 int array3[N];
 ...
}
```

次のコマンドは、デフォルトの係数 2 を使用して array1、array2、array3 配列を 3 つの新しい配列に変更して、block、cyclic、complete タイプを作成しています。

```
set_directive_array_reshape -type block -instance array4 foo array1
set_directive_array_reshape -type cyclic -instance array5 foo array2
set_directive_array_reshape -type complete -instance array6 foo array3
```

図 2-76 は、上記のコマンドの結果です。

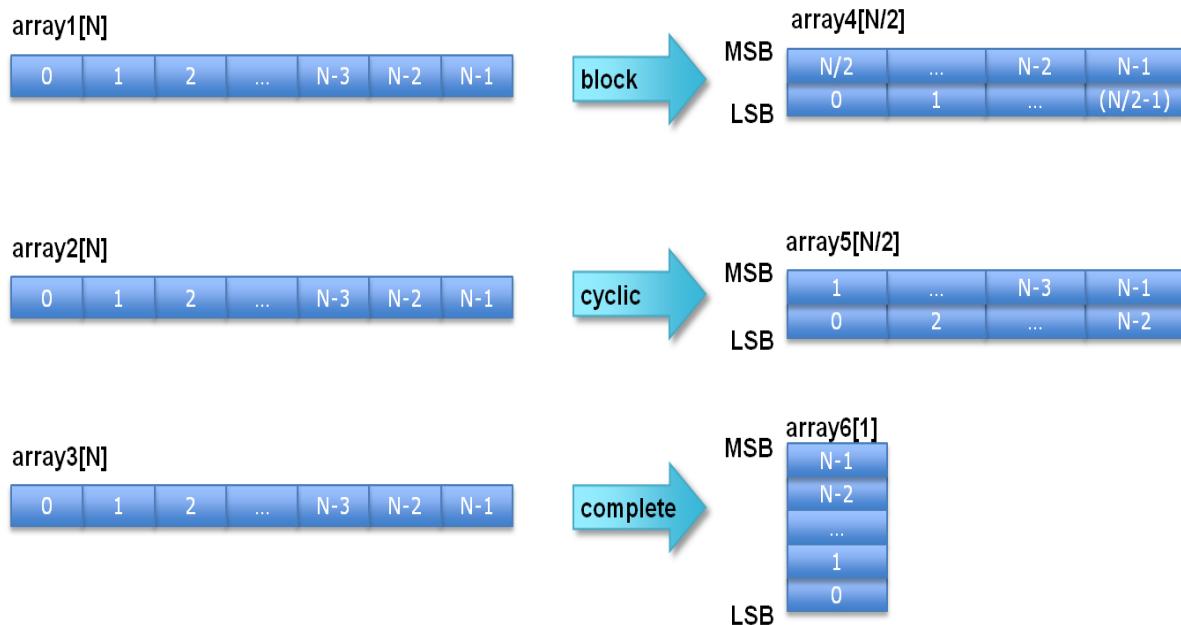


図 2-76 : 配列の変更

## 配列のストリーミング

complete 分割で配列をレジスタに削減しておかないと、デフォルトではすべての配列がメモリ エレメントとしてインプリメントされます。つまり、すべての配列は RAM リソースに割り当てられ、そのテクノロジのライブラリの RAM コアで指定されたデータ、アドレス、チップおよびイネーブル信号を使用してアクセスされます。

RAM ではなく FIFO を使用するには、配列をストリーミングとして指定する必要があります。次の配列は、自動的にストリーミングとして指定されます。

インターフェイスの配列が `ap_fifo` インターフェイス タイプに設定されると、自動的にストリーミングとして設定されます。

その他すべての配列は、FIFO インターフェイスが必要な場合は、ストリーミングとして指定する必要があります。これには、データフロー パイプライン処理が指定されたときに関数またはループ間にストリーミング インターフェイスが必要な場合も含まれます。

配列は、次のコマンドでストリーミングに指定できます。

```
set_directive_array_stream foo array1
```

データフロー チャネルに含まれる配列にコマンドを使用する場合は、このコマンドに対して `-only` オプションを使用します。`-depth` オプションを使用するとデフォルトの FIFO の深さ (最大配列のサイズ) が上書きされ、`-off` オプションを使用すると `config_dataflow` コマンドが上書きされます。デフォルトのチャネルが FIFO に指定される場合は、`set_directive_array_stream -off` オプションを使用して、配列がストリーミングされないようにし、ピンポン バッファーが使用されるようにします。

## ロジック構造の最適化

Vivado HLS で作成されるロジック構造は、最も重要です。関数およびループが並列操作のために最適化され、最小レイテンシのために結合、インライン化、平坦化され、最大スループットのためにパイプライン処理され、障害を削減するために配列アクセスが解析されると、デザイン パフォーマンスおよびその改善方法を検出または制限するのは最終的にはロジック構造だけになります。

デザインにインプリメントされたロジック構造のタイプを検出するには、次のような項目があります。

- クロック レート
- ターゲット デバイス
- 演算子選択
- ハードウェア リソースの制御
- 構造体パッキング
- 演算調整
- エラボレーション エフォート

クロック、ターゲット デバイス、ステート マシン エンコーディングおよびリセットの影響については、「デザイン 最適化」を参照してください。これらの項目の詳細は、ここで説明するその他の最適化手法を適用する前に確認してください。

### 演算子選択

合成中、Vivado HLS ではデバイステクノロジ ライブラリから演算子 (+、-、\*、/、% など) のインプリメンテーションが選択されます。デフォルトではタイミングおよびエリア間で最適なバランスになる演算子が選択されます。config\_bind コマンドを使用すると、どの演算子が使用されるかが決まり、演算子の数も最小に抑えることができます。

```
config_bind -effort [low | medium | high] -min_op <list> -reset
```

config\_bind コマンドは、アクティブ ソリューション内でのみ実行できます。コマンドを実行すると、そのソリューションで実行されるすべての合成操作に適用されます。ソリューションを一旦閉じてから開き直しても、指定したコンフィギュレーションが新しい合成操作すべてに適用されます。

config\_bind コマンドで適用されたコンフィギュレーションはすべて -reset オプションを使用すると削除できます。open\_solution -reset を使用するとソリューションを開いたときにリセットできます。

このバインディング操作のデフォルトのエフォート レベルは medium です。

- [Low Effort] : 共有にかける時間を少なくします。ランタイムは高速になりますが、最終的な RTL は大きくなる可能性があります。共有の可能性がほとんどないとわかっている場合や、必要がなく、可能性を探すためにCPU サイクルを無駄にしたくない場合などに便利です。
- [Medium Effort] : デフォルトで、高位合成は操作を共有しようとしていますが、合理的な時間で終了するようにします。
- [High Effort] : 最大限に共有しようとします。ランタイムに制限はありません。高位合成は、共有のすべての可能性のある組み合わせが確認されるまで続行されます。

エフォート レベルは最上位関数のすべての演算子に影響を与えます。バインディングはデザイン全体に対して設定されます。

-min\_op を使用すると、特定の操作が RTL で最小化されます。たとえば、デザインに 12 個の乗算器が含まれる場合、次のコマンドにより、このデザインに最小限必要な乗算器数が検索されます。

```
config_bind -min_op mul
```

使用可能な演算子のリストについては、「高位合成リファレンス ガイド」の章の「config\_bind コマンド」を参照してください。

## ハードウェアリソースの制御

RTL をインプリメントするのに使用されるリソースは、合成中に明確に指定できます。または、合成で使用できるリソースに一般的な制限を付けることができます。これらの方法は、タイミング(つまりレイテンシとスループット)とエリアの両方を改善するために使用できます。

特定操作に使用されるリソースは直接指定できます。リソースは、GUI を使用して(または `pragma` として)指定でき、`set_directive_resource` コマンドを使用する関数のどの変数にでも適用できます。

高位合成では、この例に示すようにコードが読み出されます。

```
int foo (
 int a,
 int b
) {
 int c, d;
 c = a*b;
 d = a*c;
 return d;
}
```

変数 `c` および `d` に使用される乗算は、標準的な `mul`(乗算器) 演算子として内部データベースにインプリメントされます。使用可能な演算子のリストについては、「高位合成演算子およびコア ガイド」の章を参照してください。

合成が実行されると、Vivado HLS ではクロックで指定されたタイミング制約、ターゲット デバイスで指定された遅延、およびその演算子を指定するのにどのコアを使用するか決定するユーザー制約すべてが使用されます。使用されるコアは、組み合わせコアの `multiplier` だったり、`Mul2S` というパイプライン乗算器コアだったりします。使用可能なコアのリストについては、「高位合成演算子およびコア ガイド」の章を参照してください。

どのコアを使用する必要があるか明示的に指定するには、`RESOURCE` 指示子を使用します。次のコマンドを使用すると、Vivado HLS で変数 `c` に対して 2 段のパイプライン乗算器が使用されます。

```
set_directive_resource -core Mul2S foo c
```

タイミングまたはエリアを改善するために特定の演算子を選択するだけでなく、デザインで使用される演算子の総数を制限して、演算子を共有し、エリアを改善(タイミングまたはレイテンシが増加することが多い)することもできます。上記と同じコード例で、次のコマンドを実行するとします。

```
set_directive_allocation -limit 1 -type operation foo mul
```

これはインプリメンテーションを `mul` 操作 1 つに制限します(デフォルトは制限なし)。これにより、Vivado HLS で関数 `foo` に対して 1 つの乗算器が使用されるようになります。

## 構造体パッキング

構造体のメンバーを 1 つの幅のワードにパッキングすると、各エレメントにそれぞれ関連する制御オーバーヘッドを削減できるので、デザインの小型化および高速化の両方を達成できます。

3 つの 8 ビット `char` エレメントを 1 つの幅のワードのエレメントにパックするには、`PACK` 指示子を使用できます。新しいワードでは、構造体の最初のエレメントが最下位ビット (LSB) を、最後のエレメントが最上位ビット (MSB) を占領します。構造体に配列が含まれる場合、配列が `complete` 分割で変更され、ほかのスカラーと一緒にパックされます。

`my_data` 構文体が `foo` 関数で使用されているとします。

```

typedef struct{
 unsigned char A;
 unsigned char B;
 unsigned char C;
}my_data;

void foo(my_data a_in[50], my_data b_out[50])
{
 int i;

 for(i=0; i < 50; i++){
 b_out[i].A = (a_in[i].A >> 1) + 10;
 b_out[i].B = (a_in[i].B >> 2);
 b_out[i].C = (a_in[i].C >> 3) + 100;
 }
}

```

次のコマンドは、メンバーの a-in を a-in といいう新しい変数にパックし (-instance オプションが使用されない場合は同じ名前が使用されます)、構造体 b\_in のメンバーを新しい new\_var にパックしています。

```

set_directive_data_pack foo a_in
set_directive_data_pack -instance new_var foo b_out

```

どちらの場合も、新しい変数は 24 ビット幅(3 つの 8 ビット char タイプ)になります。

注記 : データ パッキングで作成されたポートまたはバスの最大ビット幅は 8192 です。

## 演算調整

合成中は、駆動電流の削減やビット幅の最小化など、多くの最適化が自動的に実行されます。このような自動最適化の中には、演算調整も含まれます。

演算調整は、直接制御可能な最適化の 1 つです。この最適化では、バランスの取れたツリーを構築してレイテンシを削減するように、演算子が並び替えられます。演算調整はデフォルトでオンになっていますが、オフになっていることもあります。

ソフトウェアプログラマーは、代入演算子を使用してシーケンシャルなコードを記述することがよくありますが、このシーケンシャルコードはレイテンシには悪影響を与えることがあります。

次の例を確認してください。

```

int foo_top (short a, short b, short c, short d)
{
 int i;
 int sum;

 sum = 0;
 sum += a;
 sum += b;
 sum += c;
 sum += d;
 return sum;
}

```

上記のコードはシーケンシャルに実行されるはずです。加算にはそれぞれ 1 クロック サイクルかかるとすると、sum の計算が終了するのに 4 クロック サイクルかかります(図 2-77)。

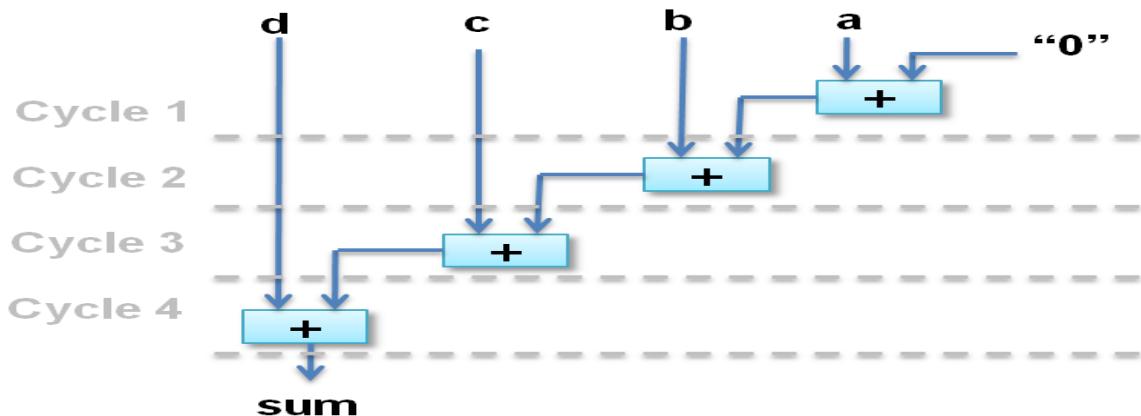


图 2-77: 加算器ツリー

ただし、加算  $(a+b)$  と  $(c+d)$  は並列で実行できるので、レイテンシを削減できます。このように計算が調整されると、計算は 2 クロック サイクルで終了します (图 2-78)。

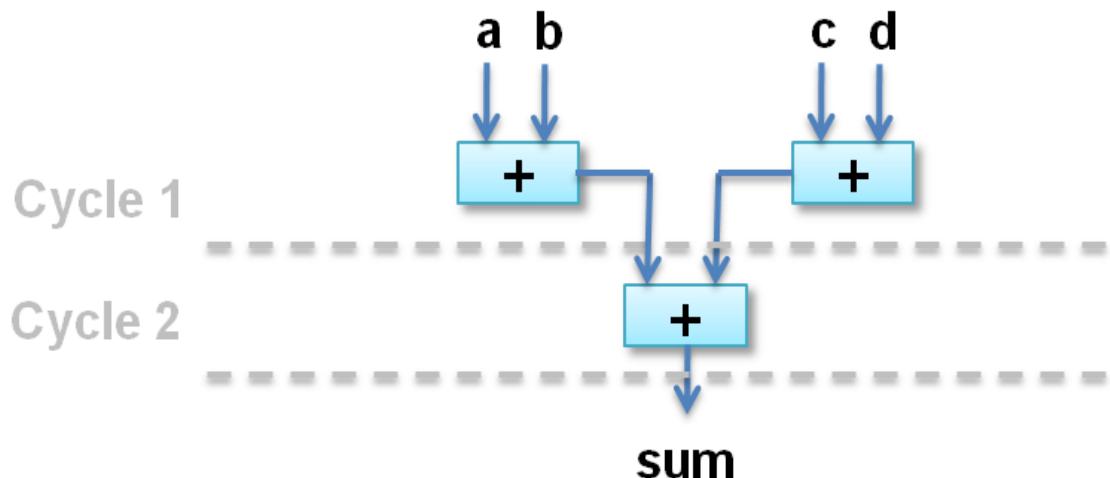


图 2-78: 調整後の加算器ツリー

通常、演算調整を使用すると共有はできず、エリアは増加します。图 2-77 からの例では、各クロック サイクルに必要な加算器は 1 つだけなので、デザインがパイプライン処理されていない限り、1 つの加算器がデザイン全体に使用されます。图 2-78 からの例では、最低 2 つ (デザインがパイプライン処理される場合は 3 つ) の加算器が必要です。

次のコマンドでは、関数 `foo` で演算調整をオフにしています。

```
set_directive_expression_balance -off foo
```

## エラボレーション エフォート

入力関数で最初に実行されるプロセスは、エラボレーションです。C/C++/SystemC の機能は、エラボレーション中に一般的なロジック構造に変換されます。エラボレーション中に使用されるエラボレーション レベルは、合成の開始点に影響します。

デフォルトでは、エラボレーション中に使用されるエフォート レベルは std (標準) です。メモリ/CPU 使用量および最適化のバランスが最適になるので、ほとんどのデザインの場合、標準エフォート レベルで十分です。

エラボレーション中のエフォート レベルは、-effort オプションで設定できます。

```
elaboration -effort [low | std | high]
```

## 低エフォート レベル

low エフォート レベルは、デザインのエラボレーションに時間がかかりすぎる場合にのみ推奨されます。最適化は、共有が少ないためにエリアが大きくなってしまう可能性のある if-else または分岐条件では実行されません。

## 標準エフォート レベル

デフォルトで、最適化とランタイム間が調整されます。

## 高エフォート レベル

high エフォート レベルが使用されると、タイミングとエリアの両方に役立つ初期データベースでさらに多くの最適化が実行されます。検索スペースを増加するもの(相互に排他的なパスが多く含まれる場合、共有の可能性が多くある場合、デザイン制約がない場合など)はすべてランタイムも増加させます。

---

# 検証

合成後の検証は、cosim\_design 機能を使用すると自動化できるので、合成前のテストベンチを再利用して出力された RTL で検証をスムーズに実行できます。

合成が終了すると、Vivado HLS は結果の RTL を syn ディレクトリに書き出します(図 2-79)。これらのファイルを適切に作成された RTL テストベンチと共に使用すると、デザインを検証できます。

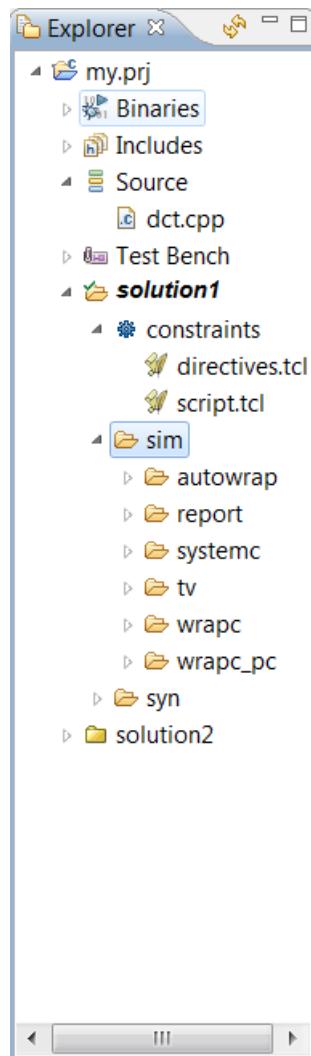


図 2-79: 出力ディレクトリ構造

ただし、Vivado HLS では、RTL デザインを検証するさらに生産的な手法である `cosim_design` が提供されています。

## RTL の自動検証

`cosim_design` 機能を使用すると、合成前の検証で作成した既存の C レベルのテストベンチを再利用し、ビルドインの SystemC RTL シミュレータまたはサードパーティの HDL シミュレータを使用して RTL を自動的に検証することができます。

`cosim_design` 機能を使用するには、次が必要です。

- 正しいインターフェイス合成オプションが選択されている
- テストベンチはセルフチェックで、値 0 を戻す
- サードパーティ シミュレータが検索パスで指定されている

## インターフェイス合成要件

cosim\_design 機能を使用して、自動的に RTL デザインを検証させるには、次の要件を満たしている必要があります。

- C および C++ デザインの最上位関数が ap\_ctrl\_hs インターフェイスを使用して合成されている必要があります。
  - このインターフェイスでは、トランザクションをいつ開始するか、およびトランザクション終了時にデータをいつ取り込むかを制御するため、start、done、idle ポートが作成されます。
- C および C++ デザインでは、各出力ポートに次のインターフェイスのいずれかが使用される必要があります (これらはデータ Valid 信号を提供する唯一のインターフェイスで、出力データを取り込むために必要です)。
  - ap\_vld
  - ap\_ovld
  - ap\_hs
  - ap\_memory
  - ap\_fifo
  - ap\_bus

SystemC デザインではインターフェイス合成が使用されないので、SystemC デザインの場合はこのような要件はありません。

## サポートされない最適化

配列またはインターフェイスの構造体内の配列ごとに複数の変換が実行されるような場合、自動 RTL 検証はサポートされません。

自動検証が実行されるようにするには、関数インターフェイスの配列、または関数インターフェイスの構造体内の配列で次の最適化のいずれかを使用する必要があります (複数は使用できません)。

- RESHAPE
- PARTITION
- 構造体エレメントのデータパック

## テストベンチ要件

RTL デザインが元の C コードと同じ結果になるかどうかを検証するには、検証を実行するために使用するテストベンチでセルフチェックが可能になっている必要があります。テストベンチのセルフチェックの重要な機能については、次の例で説明します。

```
int main () {
 int ret=0;
 ...
 // Execute (DUT) Function
 ...

 // Write the output results to a file
 ...

 // Check the results
 ret = system("diff --brief -w output.dat output.golden.dat");

 if (ret != 0) {
 printf("Test failed !!!\n");
 ret=1;
 } else {
 printf("Test passed !\n");
 }
}
```

```
 }
 ...
 return ret;
}
```

- 関数からの出力をファイルに書き出します。
- 結果を既存の既知の問題のない結果と比較します。
- 結果が正しい場合は値 0 を戻します。
- 結果が正しくない場合は 0 以外の値を戻します。
  - どのような値でも戻される可能性があります。洗練されたテストベンチであれば、相違点やエラーのタイプによって異なる値を戻す可能性があります。

上記のようなテストベンチでは、結果を自動的にチェックすることで生産性が改善され、ユーザーが手動で検証する必要がなくなります。

最上位テストベンチの main() 関数で 0 が戻されると、検証が問題なく終わったことを示すメッセージが表示されます。

```
@I [SIM-1] *** cosim_design finished:PASS ***
```

**注記** : テストベンチが 0 を戻しても、RTL 結果がセルフチェックされなかった場合も、上記のような SIM-1 メッセージが表示され、結果が実際にチェックされたわけでなくとも、シミュレーションテストをパスしたことが示されます。

テストベンチでは、予測動作に対して結果がチェックされるようにしてください。

0 以外の値が戻されると、戻された値を示すメッセージと、RTL 検証がエラーになったことを示すメッセージの、2 つのメッセージが表示されます。

**注記** : 20 が戻される場合は、テストベンチに戻り値がないことを意味します。テストベンチには、必ず結果をセルフチェックする機能を含め、正しい場合は 0 を戻すようにする必要があります。

## シミュレーション不一致のデバッグ

テストベンチがセルフチェックされ、RTL からの結果が C コードとは異なることが示され場合は、次の方法を使用すると、その差異をデバッグして、それが RTL のバグであることを確認できます。

1. C 検証段階からの結果が double 型または float 型から作成されたものではないことを確認します。double 型または float 型の結果を比較する場合、テストベンチでは絶対値ではなく範囲で比較されるようにする必要があります。これは、C コンパイルおよび合成で適用された最適化レベルによって、操作の順番が結果に影響するからです。
2. 必要な場合は、どのサンプル/サイクルで違いが最初に観測されるかを表示するように、テストベンチを変更します。
3. シミュレーションを実行する際に、[Dump Trace] オプションを選択して VCD を作成し、VCD ファイルを開くことのできるサードパーティツール (ModelSim、Verdi など) で出力波形を確認します。
4. CDT および RTL VCD ファイルのデバッガーを使用して C および RTL の共通点を探し、いつどの段階でこの 2 つが異なったのかを見つけます。

デバッグ段階には、C ソースに printf 文の追加や特定結果のファイルへの書き込みが含まれますが、違いをデバッグする基本的な方法は上記の方法になります。

## RTL シミュレータのサポート

上記の要件が満たされれば、有効な言語およびシミュレータの組み合わせ(表 2-11)を使用して、cosim\_design で RTL デザインを検証できます。

表 2-11 : cosim\_design シミュレーション サポート

| シミュレータ  | OSCI   | ModelSim | VCS    |
|---------|--------|----------|--------|
| SystemC | サポートあり | サポートなし   | サポートなし |
| Verilog | サポートなし | サポートあり   | サポートあり |
| VHDL    | サポートなし | サポートあり   | サポートあり |

SystemC RTL 出力を検証する際、Vivado HLS ではビルドインの SystemC カーネルで RTL が検証されます。これにはライセンスは必要なく、合成で使用されるのと同じバージョンの SystemC が使用されるので、RTL デザインは常に Vivado HLS を使用して検証できます。

RTL HDL デザイン (Verilog または VHDL) のいずれかを検証するには、表 2-12 に示すサードパーティ シミュレータのいずれかを使用できます。

サードパーティ シミュレータを使用する場合は、OS の検索パスにその実行ファイルを指定する必要があります。元のテストベンチもデザインと一緒にシミュレーションする必要があるので、シミュレータには C 協調シミュレーションライセンスが必要です。

## RTL 検証

シミュレーションは、GUI の [Cosimulation] ツールバー ボタンから起動できます。



図 2-80 : ツールバー

次のようなダイアログ ボックスが表示されます (図 2-81)。

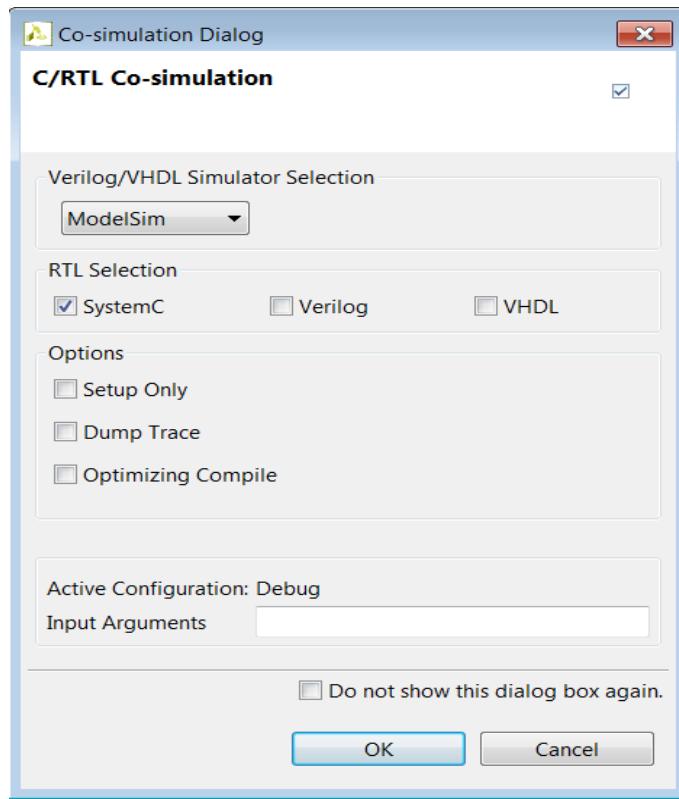


図 2-81 : [Co-simulation Dialog] ダイアログ ボックス

このダイアログ ボックスでは、シミュレーションに使用可能な RTL 言語が選択できます (複数選択可)。特定の言語を使用してシミュレーションするには、ドロップダウン メニューから [Skip] 以外の設定を選択します。このドロップダウン メニューには、その言語でサポートされるシミュレータがリストされます。

シミュレータは Vivado HLS コマンド ライン インターフェイスから Tcl コマンドの `cosim_design` を使用しても実行できます。

```
Simulate VHDL RTL using the ModelSim simulator
cosim_design -tool modelsim -rtl vhdl

#Simulate systemc RTL using the OSCI simulator
cosim_design -rtl systemc
```

検証が実行されると、シミュレーションファイルとテストベンチ内で使用されたアダプターにより、図 2-79 に示すのような sim ディレクトリが生成されます。これらはユーザーが確認することを目的には作成されていませんが、エンコードや保護はされていません。

[Setup Only] をオンにすると、デザインを検証するためのスクリプト、アダプター、ラッパーが作成されますが、シミュレータは実行されません。[Dump Trace] をオンにすると、デザインの各関数に対して VCD トレース ファイルが書き出されます。これらのファイルは、sim ディレクトリ (図 2-79) の下の該当する HDL ディレクトリに保存されます。

[Optimizing Compiler] をオンにすると、C テストベンチおよび SystemC アダプターをコンパイルするための最適化オプションが使用されます。これにより、コンパイル時間は長くなりますが、ランタイム パフォーマンスは改善されます。

[Input Arguments] では、テストベンチで必要とされる引数を指定できます。アクティブなコンパイル コンフィギュレーションは、このダイアログ ボックスの [Active Configuration] に表示されます。

## RTL デザインのエクスポート

Vivado HLS フローの最後の段階では、ザイリンクス フローのその他のツールで使用できるように、RTL デザインを IP (Intellectual Property) のブロックとしてエクスポートします。RTL デザインは、次の 3 タイプの IP ブロックとしてエクスポートできます。

- IP-XACT フォーマットの IP (Vivado で使用)
- System Generator IP ブロック
- Pcore フォーマットの IP (EDK で使用)

これらのエクスポート フォーマットにはそれぞれ、デザインのインプリメントに使用されるデバイス テクノロジおよび使用される Vivado HLS のバージョンによって制限があります。表 2-12 は、どのテクノロジを Vivado HLS の各バージョンでエクスポートできるかのサマリを示しています。Vivado HLS (System Edition (SE)) および Vivado HLS (スタンダードアロン) バージョンの違いについては、「Vivado HLS ライセンス付きテクノロジ」セクションを参照してください。

表 2-12 : RTL エクスポート フローのデバイス サポート

|                         | Vivado HLS (SE) | Vivado HLS (スタンダードアロン)                                    |
|-------------------------|-----------------|-----------------------------------------------------------|
| <b>IP-XACT</b>          | 7 シリーズ          | 7 シリーズ                                                    |
| <b>System Generator</b> | 7 シリーズ          | 7 シリーズ                                                    |
| <b>EDK Pcore</b>        | サポートなし          | 7 シリーズ、Spartan-3、Spartan-6、Virtex-4、Virtex-5、および Virtex-6 |

通常、出力パッケージには Verilog と VHDL の RTL の両方が含まれますが、デザインでバス インターフェイス (AXI4、PLB、FSL インターフェイスなど) が使用される場合は、パッケージには Verilog の RTL のみが output されます。

## バス インターフェイス

バス インターフェイスを使用すると、IP ブロックをより簡単にシステムのその他のブロックに接続できるので、IP ブロックをエクスポートする際は通常バス インターフェイス (AXI4、PLB、FSL など) を多用します。

RTL をエクスポートする前に、「インターフェイスの管理」の「バス インターフェイスの指定」セクションを参照してください。

上記に示すとおり、バス インターフェイスを使用すると、エクスポートされる IP には Verilog RTL のみが output されます。

## RTL 合成

Vivado HLS での合成結果のレポートには、クロック周波数、レジスタ数、LUT および BRAM 数などの RTL 合成後に予測される結果の概算が含まれます。Vivado HLS ではダウンストリーム RTL 合成での正確な最適化がわからず、実際の配線遅延もわからず、配置配線後の最終的なタイミングもわからないので、これらの結果はあくまでも概算です。

デザインをエクスポートする前には、Verilog または VHDL バージョンのデザインで 図 2-82 に示す [Evaluation] オプションを使用すると、ロジック合成を実行できます。

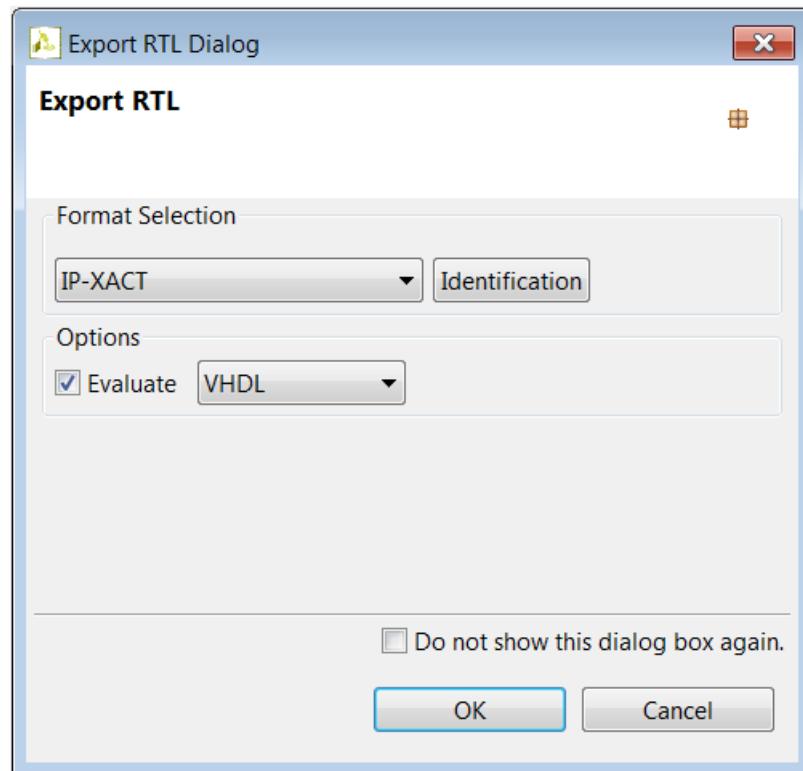


図 2-82 : [Export RTL Dialog] ダイアログ ボックス

ロジック合成は、次のように適切なザイリンクス RTL 合成製品で自動的に実行されます。

- 7 シリーズ デバイスでは Vivado RTL 合成が使用されます。
- Zynq および 7 シリーズ以外のデバイスでは ISE が使用されます。

**注記 :** RTL 合成を実行するには、システムの検索パスにロジック合成のバイナリ実行ファイルを指定しておく必要があります。詳細は、『ザイリンクス デザイン ツール：インストールおよびライセンス ガイド』を参照してください。

この [Evaluation] オプションを使用すると、IP がデザイン フローの次の段階に送信される前に、Vivado HLS 環境内で HLS の概算が確認されるようになります。RTL 合成の結果は <Project\_Directory>/<Solution\_Name>/impl/<HDL\_Version> ディレクトリに格納され、エクスポートされる IP には含まれません。

この RTL IP ブロックが高集積の RTL デザインに含まれ、RTL 合成が再実行されると、これらの結果はわずかに異なることに注意してください。[Evaluation] オプションは最終結果が Vivado HLS の概算と近くなるかどうかを素早く確認する目的にのみ使用できます。

## パッケージの識別

IP-XACT および Pcore のフォーマットでは、エクスポートされるパッケージに ID タグを埋め込むことができます。これらのフィールドはオプションですが、何も指定されない場合は、次のデフォルト文字列が使用されます (Pcore フォーマットにはバージョン識別文字列のみが含まれます)。

- version: 1.00.a
- library: hls
- vendor: xilinx.com

- description: An IP generated by Vivado HLS

## Tcl コマンド

RTL のエクスポートは、Tcl コマンドの `export_design` でサポートされます。このコマンドは、`csynth_design` コマンドの後にのみ実行できますが、異なるオプションを使用して複数回実行できます。

このコマンドの詳細は、「高位合成リファレンス ガイド」の章か、GUI メニューの [Help] → [Man Page] を参照してください。

## IP-XACT フォーマットでのエクスポート

合成および RTL 検証が終了したら、メイン メニューから [Solution] → [Export RTL] をクリックするか、[Explorer] タブでソリューションを右クリックするか、[Export RTL] ツールバー アイコン (図 2-83) をクリックして、[Export RTL Dialog] ダイアログ ボックスを開きます。



図 2-83 : [Export RTL] ツールバー ボタン

[Format Selection] セクションで [IP-XACT] を選択します (図 2-82)。これはデフォルトのオプションです。

IP-XACT パッケージの ID 情報フィールドは、その IP の [Identification] ダイアログ ボックスでカスタマイズできます。すべてのフィールドには最初はデフォルト値が指定されています。

IP ブロックの配置配線後のリソースおよびタイミング統計が必要な場合は、[Evaluate] をオンにし、必要な RTL 言語を指定します。

[OK] をクリックして IP-XACT パッケージを生成します。このパッケージは、`<Project_Directory>/<Solution_Name>/impl/ip` ディレクトリに書き込まれます。この `ip` ディレクトリには、ZIP ファイルが含まれますが、これが IP-XACT パッケージです。

[Evaluate] をオンにした場合は、ロジック合成が実行され、最終的なタイミングおよびリソースがレポートされます。RTL 合成の詳細は、前述の「RTL 合成」を参照してください。

## IP-XACT パッケージの Vivado へのインポート

Vivado HLS で生成した IP-XACT パッケージは、GUI で次の手順を実行すると、Vivado Design Suite の IP カタログにインポートできます (その他の IP インポート方法については、Vivado の資料を参照してください)。

1. プロジェクトを Vivado Design Suite の GUI で開きます。
2. Flow Navigator (デフォルトレイアウトでは一番左) で [Project Manager] → [IP Catalog] をクリックします。
3. [IP Catalog] ビュー (デフォルトレイアウトでは右上) で [Add IP] アイコン (左下の緑の + マークが付いた黄色のシンボル) をクリックします。[Add IP] ダイアログ ボックスが開きます。
4. [Repository Path] が設定されていないか、別のパスを指定する場合は、それを選択するか作成します。
5. 参照アイコンをクリックし、[Select IP File] ダイアログ ボックスを開きます。

6. Vivado HLS で生成されたパッケージの ZIP ファイルを参照して選択します。このダイアログ ボックスとその前に表示していたダイアログ ボックスで [OK] をクリックします。
7. 処理が終了したら、IP のヘディングに「VIVADO HLS IP」と表示されます。
8. ザイリンクスの提供する IP はこの段階で同じように追加できます。

ユーザーの生成した IP をインポートしたレポジトリ ディレクトリは、[Tools] → [Options] の [General] カテゴリで [IP Catalog] フィールドまでスクロールダウンして [Add Directories] で指定しておくと、IP カタログへ追加することができます。

## Pcore フォーマットでのエクスポート

合成および RTL 検証が終了したら、メイン メニューから [Solution] → [Export RTL] をクリックするか、[Explorer] タブでソリューションを右クリックするか、[Export RTL] ツールバー アイコン (図 2-83) をクリックして、[Export RTL Dialog] ダイアログ ボックスを開きます。

[Format Selection] セクションで [Pcore for EDK] を選択します (図 2-84)。Pcore パッケージのバージョン情報フィールドは、その IP の [Identification] ダイアログ ボックスでカスタマイズできます。デフォルトは、1.00.a です。

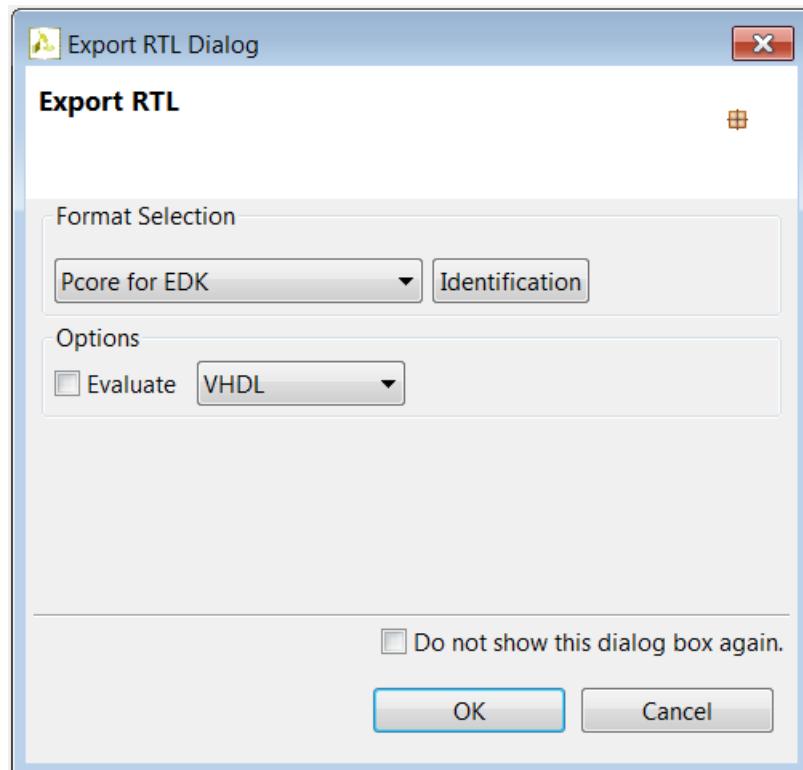


図 2-84 : RTL を Pcore としてエクスポート

IP ブロックの配置配線後のリソースおよびタイミング統計が必要な場合は、[Evaluate] をオンにし、必要な RTL 言語を指定します。

[OK] をクリックして Pcore パッケージを生成します。このパッケージは、<Project\_Directory>/<Solution\_Name>/impl/pcores ディレクトリに書き込まれます。この pcores ディレクトリには <top\_level\_design\_name>\_top\_v<Version\_String> というディレクトリが含まれますが、これが Pcore パッケージです。

Pcore パッケージ内の `include` ディレクトリには AXI4 Lite スレーブまたは PLB 4.6(スレーブ) インターフェイスなどの、デザインに含まれるスレーブ インターフェイスのソフトウェア ファイルが含まれます。これらのファイルの使用については、「インターフェイスの管理」の「バス インターフェイスの指定」を参照してください。

[Evaluate] をオンにした場合は、ロジック合成が実行され、最終的なタイミングおよびリソースがレポートされます。RTL 合成の詳細は、前述の「RTL 合成」を参照してください。

## Pcore パッケージの EDK 環境へのインポート

Vivado HLS 生成の Pcore パッケージは、`pcore` ディレクトリの内容を EDK プロジェクトの `pcore` ディレクトリにコピーすると、EDK 環境にインポートできます。

1. `<Project_Directory>/<Solution_Name>/impl/pcores/*` ディレクトリを `<EDK_Project>/pcores` ディレクトリにコピーします。
2. EDK プロジェクトで、その IP ブロックが [Project Local PCores] の下に表示されます。
3. IP ブロックが [Project Local PCores] の下に表示されない場合は、[Project] → [Rescan Local Repository] をクリックして、手動でローカル Pcore 情報をアップデートします。

## System Generatorへのエクスポート

System Generator へインポートされる IP ブロックには、必ずクロック イネーブル ポートが必要です。Vivado HLS では、System Generator 環境にデザインをエクスポートする前に、この条件が満たされているかどうかがチェックされます。デザインにクロック イネーブル ポートが含まれない場合は、Vivado HLS で次のようなエラー メッセージが表示されます。

```
@E[IMPL-64] Clock enable port is required for exporting the design to System Generator. Please use 'config_interface -clock_enable' to generate the port.
```

**注記** : C 合成が実行される前にデザインにクロック イネーブル ポートを追加する必要があります。

上記のエラー メッセージに示されるように、クロック イネーブル ポートは `config_interface` コマンドで `-clock_enable` オプションを使用すると追加されます。

このインターフェイス コンフィギュレーションは、GUI または Tcl コマンドで実行できます。

A. GUI の場合、ソリューションを選択します。[Solution] → [Solution Settings] → [General] → [Add] → [config\_interface] を選択し、[clock\_enable] をオンにして、インターフェイス コンフィギュレーション設定を開きます。

または

B. Tcl スクリプトの `csynth_design` コマンドの前に `config_interface -clock_enable` を追加します。

## ポートの最適化

最上位関数の引数が合成プロセス中に 1 つの複合ポートにまとめられると、そのポートのタイプ情報はわからないので、System Generator IP ブロックには含まれません。

このため、ポートで再形成、マッピング、データ パッキング最適化などを使用した場合は、これらの複合ポートに対してポート タイプ情報を System Generator で指定する必要があります。

System Generator でタイプ情報を手動で指定するには、まず複合ポートがどのように作成されたか理解しておいてから、Vivado HLS ブロックをシステムのほかのブロックに接続する際に System Generator 内でスライスおよび再変換ブロックを使用します。

次に例を示します。

- R、G、B の 3 つの 8 ビット入出力ポートが 24 ビット入力ポート (RGB\_in) と 24 ビット出力ポート (RGB\_out) にパックされているとします。

IP ブロックが System Generator に含まれる際には、次のようにになっている必要があります。

- 24 ビット入力ポート (RGB\_in) は、3 つの 8 ビット入力信号 (Rin、Gin、Bin) を正しくまとめた System Generator ブロックで駆動される必要があります。
- 24 ビット出力ポート (RGB\_out) は、3 つの 8 ビット入力信号 (Rout、Bout、Gout) に正しく分割される必要があります。

複合タイプのポートに接続するためのスライスおよび再変換ブロックの使用方法にの詳細については、System Generator の資料を参照してください。

## RTL のエクスポート

合成および RTL 検証が終了したら、メイン メニューから [Solution] → [Export RTL] をクリックするか、[Explorer] タブでソリューションを右クリックするか、[Export RTL] ツールバー アイコン (図 2-83) をクリックして、[Export RTL Dialog] ダイアログ ボックスを開きます。

[Format Selection] セクションで [System Generator For DSP] を選択します (図 2-85)。

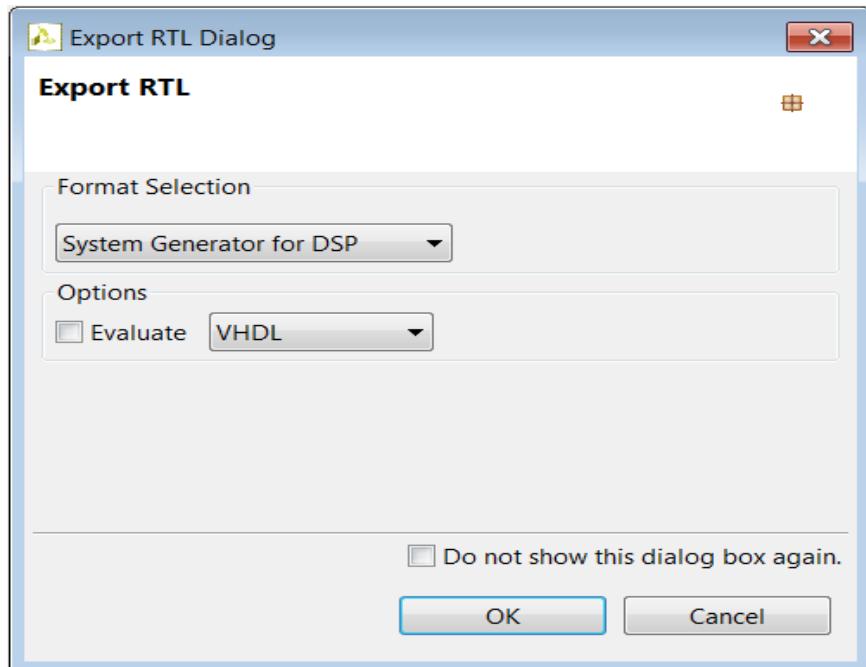


図 2-85 : System Generator への RTL のエクスポート

IP ブロックの配置配線後のリソースおよびタイミング統計が必要な場合は、[Evaluate] をオンにし、必要な RTL 言語を指定します。

[OK] をクリックして System Generator パッケージを生成します。このパッケージは、`<Project_Directory>/<Solution_Name>/impl/sysgen` ディレクトリに書き込まれます。このディレクトリには、System Generator ヘデザインをインポートするために必要なものがすべて含まれます。

[Evaluate] をオンにした場合は、ロジック合成が実行され、最終的なタイミングおよびリソースがレポートされます。RTL 合成の詳細は、前述の「RTL 合成」を参照してください。

## System Generator への RTL のインポート

Vivado HLS で生成された System Generator パッケージは、次の手順で System Generator にインポートできます。

1. System Generator デザイン内で右クリックし、XilinxBlockAdd オプションを使用して新しいブロックをインストールします。
2. ダイアログ ボックスのリストをスクロールダウンし、[Vivado HLS] を選択します。
3. 新しくインスタンシエートした Vivado HLS ブロックをダブルクリックし、[Block Parameters] ダイアログ ボックスを開きます。
4. Vivado HLS ブロックがエクスポートされた solution ディレクトリを参照します。上記の例の  
<Project\_Directory>/<Solution\_Name>/impl/sysgen の場合、  
<Project\_Directory>/<Solution\_Name> ディレクトリを参照し、[apply] をクリックします。

# 高位合成演算子およびコア ガイド

Vivado 高位合成 (HLS) は、C、C++、または SystemC デザイン仕様をレジスタ トランസファー レベル (RTL) インプリメンテーションに変換し、それをザイリンクス フィールド プログラマブル ゲート アレイ (FPGA) に合成することができます。

このために、Vivado HLS では次の作業が実行されます。

- まず、C、C++、または SystemC のソース コードが演算子を含む内部データベースにエラボレートされます。
  - 演算子は、加算、乗算、配列読み出しおよび書き込みなど C コードの演算を表します。
- 合成中、Vivado HLS により、Vivado HLS ライブラリからのコアに演算子がマップされます。
  - コアは、デザイン作成のために使用される特定ハードウェア コンポーネントです (加算器、乗算器、パイプライン乗算器、ブロック RAM など)。
  - 各ザイリンクス テクノロジ (Spartan®-6、Virtex®-7 などのザイリンクス デバイス) のライブラリが提供されています。

本書では、Vivado HLS でサポートされている演算子およびそのライブラリについて説明します。

---

## 合成の概要

C ソース コードをレジスタ トランസファー レベル (RTL) 記述に変換するため Vivado HLS を実行すると、ソース コードは演算子を含む内部データベースにエラボレートされます。

演算子はこの章で説明されていますが、基本的には、加算、シフト、乗算、ビット スライス、配列アクセスなど、C ソース コードの演算で構成されています。

Vivado HLS でデザインが合成されるときこの内部データベースが使用されます。合成は「スケジューリング」と「バインディング」という 2 つのプロセスから成り立っています。

## スケジューリングとは

スケジューリングとは、どのサイクルで演算を実行するかが Vivado HLS で決定されるプロセスです。

たとえば、同じクロック サイクルで 2 つの加算がスケジュールされていると、この 2 つの加算が同じハードウェア 加算器を使用することはできません。しかし、異なるクロック サイクルでスケジュールされている場合、同じ加算器を使用でき、リソースを節約することができます。特に制約や指示子がなければ、最小限のレイテンシを達成するため、Vivado HLS はまずデザインをスケジュールします。つまり、同じクロック サイクルで加算を 2 つスケジュールする必要がある可能性があります。

## バインディングとは

スケジューリングが完了したら、次はバインディングプロセスです。このプロセスでは、テクノロジライブラリからの特定ハードウェアインプリメンテーション(またはコア)にスケジュール済みの演算が関連付けられます。

たとえば、ソースコードのある乗算は標準の組み合わせ乗算器でインプリメントすることができる一方で、別の乗算はパイプライン乗算器でインプリメントすることができます。パイプライン乗算には2段必要となるので、スケジューリング中にこの点が考慮されているはずです。

Vivado HLSでは、バインディングの影響(演算のインプリメンテーションに使用される特定コアの情報)はスケジューリングプロセス中に考慮されます。[158ページの図1](#)にはスケジューリングおよびバインディングのプロセスが示されています。こうしてバインディング中の決定事項によってデザインが再スケジュールされることがない、つまり、際限なく繰り返しが実行されることがないようになっています。

バインディング中に使用可能なコアの種類は選択されているデバイスによって異なります。Vivado HLSでは、ザイリンクスデバイスごとにライブラリが提供されているので、異なるライブラリのコアには異なる遅延およびタイミングが設定されています。各コアに関連付けられている遅延は、1つのクロックサイクルにどのコアをスケジュールできるかに影響を及ぼします。

作成されるスケジュールはどのコアに演算子が関連付けられているかによって異なります。このため、スケジューリングプロセスではバインディングのデザインへの影響が考慮されます。

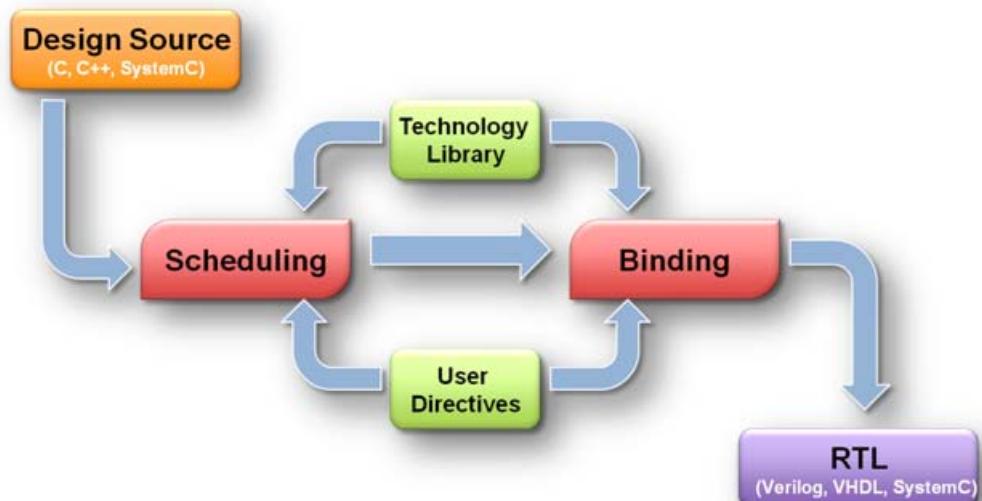


図1: 合成プロセス

デバイスの選択に加え、Vivado HLSではスケジューリングおよびバインディングプロセスを制御するためのコマンドおよび指示子が数多く提供されています。

説明は簡単なのですが、合成プロセスはインプリメンテーションにおいて複雑で、デザイン作成にあたり次のような多くの要素が考慮されます。

- コードの演算子
- デザインスケジュール
- コアのタイミング遅延
- ユーザー制約および指示子
- バインディングプロセス

RTL インプリメンテーションで使用可能なコアを理解することは、ハイパフォーマンス デザインを達成するために重要です。次のセクションでは、Vivado HLS の演算およびコアについて説明します。

## 演算子、コア、指示子について

高位合成中、C、C++、または SystemC のソース コードの演算子が内部データベースで識別および表示されます。内部データベースのリストまたはアクセス用のコマンドはありませんが、演算は Vivado HLS の GUI にある [Design Viewer] で確認することができます。159 ページの図 1-1 は [Design Viewer] の例です。

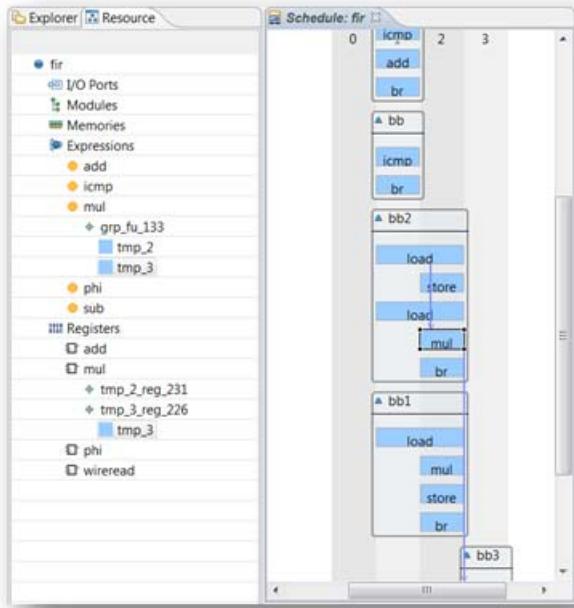


図 1-1 : [Design Viewer] の演算子

図 1-1 で多くの演算が確認できます。たとえば、加算は add、乗算は mul、配列読み出しは load、比較は cmp、ブレークは br で表されています。

図 2-85 は、[Design Viewer] での演算子バインディングの結果がどのように表示されるかを示しています。ブロック [bb2] の [mul] 演算が [Schedule] (図の右側) で選択されていると、自動的に [Resource] (図の左側) でもハイライトされ、mul 演算がハードウェア乗算器リソースおよびインスタンス [grp\_fu\_133] に関連付けられていることが分かります。インスタンス [grp\_fu\_133] (RTL でも同じ名前が使用される) もまた、2 つ目の乗算用に使用されることが分かります。[grp\_fu\_133] 内に [tmp2] および [tmp3] の 2 つの演算があり、この 1 つのハードウェアインスタンスが 2 つの mul 演算に対して使用されることになります。

- 乗算などの演算は、特定コアを使用して、特定ハードウェア乗算器で RTL デザインにインプリメントされます。すべての演算がコアにマップされるわけではありません。
- ループやスイッチ文からのブレークなどの演算は制御フロー動作であり、ライブラリからのコアではなくロジックを使用してインプリメントされます。このような演算はユーザー指示子で制御することはできません。

# 演算子およびコアの制御

Vivado HLS では次のような方法で演算およびコアの使用が制御されます。

- 演算の割り当てプロセスを指示
- 演算の特定リソースを指示
- スケジューリング エフォート
- 演算子をコアにバインドするプロセスを制御
- コアの詳細をリスト

## 演算子の制限

Vivado HLS ではデザインで使用される演算子の数を制限することができます。たとえば、`foo` というデザインには乗算が 317 あるのに FPGA には乗算器が 256 しかない場合、Vivado HLS で次の割り当てコマンドを使用して、乗算 (`mul`) 演算子の最大数を 256 に設定してデザイン スケジュールを作成することができます。

```
set_directive_allocation -limit 256 -type operation foo mul
```

この割り当て指示子は、コアおよび特定下位関数のインスタンス数を制限するのにも同じように使用することができるため、`-type` オプションに `operation` を指定しています。

デフォルトでは Vivado HLS はパフォーマンスを最大限にしようとするため、場合によっては、エリア低減のため明示的に演算子の数を制限する必要があります。

- デフォルトでは、特に制約や指示子がなければ、Vivado HLS は最小限のレイテンシ (入力から出力までのサイクル数が最小) でデザインを作成します。
- 最大または最小レイテンシを指定するため指示子が適用されると、Vivado HLS はこれらのレイテンシ制約を満たそうとします。
- パイプラインのために指示子が適用されると、Vivado HLS はまずこれらの制約を満たそうとし、次に任意のレイテンシ制約を最小限に抑えるか、または満たそうとします。

レイテンシを最小限に抑えるということは最終デザインでより多くのコアが使用されることを意味します。たとえば、2 クロック サイクルかけて同じ加算器を共有するのではなく、同じクロック サイクルで 2 つの加算器を使用します。

演算子の数を制限すると、最終デザインで使用されるコアが少なくなりますが、レイテンシを増加させることになる可能性があります。

[164 ページの表 1](#) には `set_directive_allocation` コマンドを使用して制御できる演算がすべてリストされています。

## リソースの制御

`set_directive_resource` コマンドは演算に対しどのコアを使用するかを指定します。この指示子はスケジューリング プロセス中にコアを特定し、明示的にバインディングを指定します。

テクノロジ ライブラリの詳細をリストすると、各コア (リソース) でどの演算をインプリメントできるかが表示されます。詳細は [163 ページの「コアの詳細」](#) を参照してください。

`set_directive_resource` は、配列のインプリメント用のメモリ エレメントを指定するのによく使用されます。

Vivado HLS は解析を実行して、各演算にどのコアを使用できるのか、また使用すべきかを決定することができます。

---

**キー コンセプト**：配列の場合、各配列に対しテクノロジ ライブラリから特定メモリを指定する必要があります。何も指定されていない場合、最大スループットおよび最小レイテンシを提供するメモリ（シングルまたはデュアルポート）が高位合成（HLS）で自動的に選択されます。

---

このリソース指示子は、リソースのターゲットとして割り当てられている変数を使用します。関数 `foo` に `Result=A*B` というコードがあるとします。この例では、`Mul2S` という 2 段のパイプライン乗算器コアを使用して乗算がインプリメントされるように指定します。

```
set_directive_resource -core Mul2S foo Result
```

変数が複数の演算子で使用されている場合、各演算子に対し変数は 1 つとなるようにコードを変更する必要があります。たとえば、

```
Result = (A*B) + C;
```

これを次のように変更します。

```
Result_tmp = (A*B);
Result = Result_tmp + C;
```

`Result_tmp` の指示子は乗算リソースを制御し、`Result` は加算リソースを制御します。

## スケジュールの制御

`config_schedule` コマンドは、スケジューリング中のエフォートを制御し、デザインで制約を満たすことができなかったパスを正確に識別します。

### スケジューリング エフォート

Vivado HLS でのエフォート レベル、high、low、および medium の概要を説明します。

Vivado HLS でエフォート レベルが `high` に設定されている場合、制約が満たされた後でも、さらに小型または高速なデザインが作成できるかどうかを判断するため、追加 CPU サイクルおよびメモリが使用されます。この作業によりさらに質の高いデザインが作成される可能性がありますが、完了までに時間がかかり、またより多くのメモリが必要になります。僅差で目標を満たすことができないデザインや、いろいろな最適化の組み合わせが可能なデザインであれば、このストラテジは有益といえるでしょう。

デザインによっては、コードの記述方法によりあまり最適化できないものがあります。たとえば、ある変数から別の変数にデータを移動させるようにコードが記述されている場合、ほかの作業を行う必要はなく、ただそのコードをインプリメントするだけです。Vivado HLS は、時間を費やしてもあまり改善されないという判断をするのではなく、改善の余地があるかどうかを検討するためのソフトウェアです。このようなデザインの場合は、エフォート レベルを `low` に設定しておくと、時間をかけずに同じ結果を得ることができます。

つまり、エフォート レベルはデフォルトの `medium` のままにしておくのがベストです。



**ヒント** : 演算子とコアをいろいろ組み合わせたりする余地があまりないデザインや、実行に時間がかかるデザインの場合は、エフォート レベルを `low` にしておくと、あまり時間をかけずに同じ結果が得られる可能性があります。



**ヒント** : エリアやタイミングにおいてパフォーマンス要件を僅差で満たすことができないデザインの場合は、さらに改善の余地があるかどうかを確認するため、エフォート レベルを `high` に設定するとよいでしょう。

### クリティカルパス解析

`config_schedule` コマンドには `-verbose` オプションもあります。このオプションを指定すると、スケジューリングで制約を満たすことができない場合、Vivado HLS によりフルクリティカルパスが表示されます。

## バインディングの制御

`config_bind` コマンドはバインディング プロセスを制御します。このコマンドは、コアを演算子に関連付けるときのエフォート レベルを指定し、またエリア要件の厳しいデザインで演算子を最小限に抑えるために直接制御ができます。

### バインディング エフォート

スケジューリング エフォートの一般ガイドラインがバインディングにも当てはまります。この場合、演算に対し使用可能なコアが多くあるデザインは、あまり選択肢のないデザインよりも、エフォート レベルを高くすると利点があります。

各演算子をインプリメントすることができるコアの数を確認するには、`list_core` コマンドを使用することができます。このコマンドの詳細は「[コアの詳細](#)」を参照してください。

### 演算子を最小限に抑える

デザインでもっとリソースが共有されるようにするには `config_bind` コマンドを使用することができます。

- config\_bind コマンドに -min\_op オプションを設定すると、Vivado HLS で最小数の指定演算子でデザインが作成されます。

たとえば、次のようにコマンドを入力すると、最小数の add 演算子でデザインが作成されます。

```
config_bind -min_op add
```

このコマンドはバインディングプロセスに影響するので、異なるクロックサイクルで演算子がスケジュールされているときにはのみ効果があります。ほかの制約を満たすため(レイテンシおよびスループット)、同じクロックサイクルに演算子がスケジュールされている場合、config\_bind コマンドはこれらの演算子には効果がありません。スケジューリングの結果に影響を及ぼす割り当て指示子をまずは使用して演算子の数を制限してください。

このコマンドオプションはMUXの使用率を変更するのに通常使用されます。同じコアで演算が共有されていると、LUTにインプリメントされている追加MUXがタイミングとエリアの両方に著しく影響を及ぼす可能性があります。タイミング違反の可能性がある場合は、Vivado HLS では通常MUXの使用に保守的な措置を取るのが一般的です。

## コアの詳細

list\_core コマンドは、ライブラリで使用可能なコアの詳細を表示するのに使用されます。

list\_core を使用するには、set\_part コマンドを使用してデバイスを選択する必要があります。デバイスが選択されていない場合は、コマンドの効果はありません。

- list\_core コマンドの -operation オプションは、ライブラリにあるコアのうち、指定された演算でインプリメントできるものをすべてリストします。

164 ページの表 1 には、使用可能な演算がリストされています。オプションを指定せずにこのコマンドを使用すると、ターゲットデバイスに対して使用可能なコアがすべてリストされます。

- type オプションは、カテゴリ別にコアをリストしたい場合に使用します。
  - Function Units** - 標準 RTL 演算(加算、乗算、比較など)をインプリメントするコア
  - Storage** - レジスタやメモリなどのストレージエレメントをインプリメントするコア
  - Adapter** - IP 生成時に最上位デザインを接続するために使用するインターフェイスをインプリメントするコア。これらのインターフェイスは、エンベデッド開発キット(EDK)などのIP生成フローで使用される RTL ラッパーにインプリメントされます。
  - IP Blocks** - ユーザーが追加する IP コア
  - Connectors** - デザイン内の接続をインプリメントするコア。直接接続やストリーミングストレージエレメントがこれに含まれます。

164 ページの「高位合成のコア」にある表にザイリンクスデバイスで使用される標準コアがリストされています。

# 高位合成の演算子

表 1 には高位合成(HLS)で使用される演算子がリストされています。

この表の各列には各演算についての次の情報が記載されています。

- [Design Viewer] で表示可能かどうか
- set\_directive\_allocation および set\_directive\_resource の指示子または、config\_bind コマンドで制御可能かどうか
- 関連付けられているコアをライブラリからリストできるかどうか

表 1: 高位合成の演算子

| 演算子        | 説明                | Design Viewer | 指示子による制御 | ライブラリコアのリスト |
|------------|-------------------|---------------|----------|-------------|
| add        | 加算                | X             | X        | X           |
| ashr       | 四則演算右シフト          | X             | X        | X           |
| br         | ブレーク              | X             |          |             |
| fiforead   | FIFO 読み出し         | X             |          | X           |
| fifowrite  | FIFO 書き込み         | X             |          | X           |
| fifoappend | 非ブロッキング FIFO 読み出し | X             |          | X           |
| fifoappend | 非ブロッキング FIFO 書き込み | X             |          | X           |
| icmp       | 整数比較              | X             | X        | X           |
| load       | メモリ読み出し           | X             |          | X           |
| lshr       | 論理演算右シフト          | X             | X        | X           |
| mul        | 乗算                | X             | X        | X           |
| mux        | マルチプレクサ           | X             |          | X           |
| phi        | マルチプレクサ           | X             |          |             |
| sdiv       | 符号付き除算            |               | X        |             |
| shl        | 左シフト              | X             | X        | X           |
| srem       | 符号付き剰余            | X             |          | X           |
| store      | メモリ書き込み           | X             |          | X           |
| sub        | 減算                | X             | X        | X           |
| udiv       | 符号なし除算            | X             | X        | X           |
| urem       | 符号なし剰余            | X             |          | X           |
| srem       | 符号付き剰余            | X             |          | X           |
| wireread   | I/O 読み出し          | X             |          |             |
| wirewrite  | I/O 書き込み          | X             |          |             |

## 高位合成のコア

Vivado HLS コアは次のカテゴリでリストすることができます (list\_core コマンドでも使用可能)。

- ・ 「ファンクション ユニット コア」
- ・ 「ストレージ コア」
- ・ 「コネクターコア」
- ・ 「アダプターコア」
- ・ 「浮動小数点コア」
- ・ IP ブロック



**重要:** これはユーザーによってライブラリに追加されるブロックのことで、本書には記載されていません。

これらのコアの詳細は、[表 2 から 167 ページの表 6](#) を参照してください。

## ファンクションユニットコア

[表 2](#) には標準 RTL 演算 ( 加算、乗算、比較など ) をインプリメントするコアがリストされています。

**表 2: ファンクションコア**

| コア       | 説明                                                        |
|----------|-----------------------------------------------------------|
| AddSub   | 加算器および減算器の両方をインプリメントするのに使用します。                            |
| AddSubnS | N 段のパイプライン加算器または減算器です。Vivado HLS で自動的に必要なパイプライン段数が決定されます。 |
| Cmp      | コンパレータ                                                    |
| Div      | 除算器                                                       |
| Mul      | 組み合わせ乗算器                                                  |
| Mul2S    | 2 段パイプライン乗算器                                              |
| Mul3S    | 3 段パイプライン乗算器                                              |
| Mul4S    | 4 段パイプライン乗算器                                              |
| Mul5S    | 5 段パイプライン乗算器                                              |
| Mul6S    | 6 段パイプライン乗算器                                              |
| MulnS    | N 段パイプライン乗算器 Vivado HLS で自動的に必要なパイプライン段数が決定されます。          |
| Sel      | ジェネリックな選択演算子。通常、マルチプレクサとしてインプリメントされます。                    |

## ストレージコア

[表 3](#) にはレジスタやメモリなどのストレージエレメントをインプリメントするコアがリストされています。

**表 3: ストレージコア**

| コア            | 説明                                                                                            |
|---------------|-----------------------------------------------------------------------------------------------|
| FIFO          | FIFO です。Vivado HLS で BRAM または分散 RAM のいずれかで RTL にインプリメントするかが決定されます。                            |
| FIFO_BRAM     | BRAM でインプリメントされる FIFO。                                                                        |
| FIFO_LUTRAM   | 分散 RAM としてインプリメントされる FIFO。                                                                    |
| FIFO_SRL      | SRL でインプリメントされる FIFO。                                                                         |
| RAM_1P        | シングルポート RAM。Vivado HLS で BRAM または分散 RAM のいずれかで RTL にインプリメントするかが決定されます。                        |
| RAM_1P_BRAM   | シングルポート RAM で、BRAM でインプリメントされます。                                                              |
| RAM_1P_LUTRAM | シングルポート RAM で、分散 RAM としてインプリメントされます。                                                          |
| RAM_2P        | 読み出しポートと書き込みポートを別々に使用するデュアルポート RAM。Vivado HLS で BRAM または分散 RAM のいずれかで RTL にインプリメントするかが決定されます。 |

表3:ストレージコア(続き)

| コア            | 説明                                                                |
|---------------|-------------------------------------------------------------------|
| RAM_2P_BRAM   | 読み出しポートと書き込みポートを別々に使用するデュアルポートRAMで、BRAMでインプリメントされます。              |
| RAM_2P_LUTRAM | 読み出しポートと書き込みポートを別々に使用するデュアルポートRAMで、分散RAMとしてインプリメントされます。           |
| RAM_T2P_BRAM  | 入力および出力の両方で読み出しポートと書き込みポートの両方をサポートするデュアルポートRAMで、BRAMでインプリメントされます。 |
| RAM_2P_1S     | デュアルポート非同期RAMで、LUTにインプリメントされます。                                   |
| ROM_1P        | シングルポートROM。Vivado HLSでBRAMまたはLUTのいずれかでRTLにインプリメントするかが決定されます。      |
| ROM_1P_BRAM   | シングルポートROMで、BRAMでインプリメントされます。                                     |
| ROM_1P_LUTRAM | シングルポートROMで、分散ROMとしてインプリメントされます。                                  |
| ROM_1P_1S     | デュアルポート非同期ROMで、LUTにインプリメントされます。                                   |
| ROM_2P        | デュアルポートROM。Vivado HLSでBRAMまたは分散ROMのいずれかでRTLにインプリメントするかが決定されます。    |
| ROM_2P_BRAM   | デュアルポートROMで、BRAMでインプリメントされます。                                     |
| RAM_2P_LUTRAM | デュアルポートROMで、分散ROMとしてインプリメントされます。                                  |

## コネクターコア

表4にはデザイン内の接続をインプリメントするコアがリストされています。直接接続やストリーミングストレージエレメントがこれに含まれます。

表4:コネクターコア

| コア  | 説明      |
|-----|---------|
| Mux | マルチプレクサ |

## アダプターコア

表5には生成時に最上位デザインを接続するために使用するインターフェイスをインプリメントするコアがリストされています。これらのインターフェイスは、EDKでのIP生成フローで使用されるRTLラッパーにインプリメントされます。

表5:アダプターコア

| コア        | 説明                           |
|-----------|------------------------------|
| FSL       | 標準ザイリンクスFSLインターフェイス          |
| NPI64M    | ネイティブマルチポートメモリコントローラインターフェイス |
| PLB46S    | 標準ザイリンクスPLB46スレーブインターフェイス    |
| PLB46M    | 標準ザイリンクスPLB46マスターインターフェイス    |
| AXI4LiteS | AXI4ライトのスレーブインターフェイス         |

表 5: アダプターコア (続き)

| コア         | 説明                  |
|------------|---------------------|
| AXI4M      | AXI4 マスター インターフェイス  |
| AXI4Stream | AXI4 ストリームのインターフェイス |

## 浮動小数点コア

Vivado HLS では各ザイリンクス デバイスに対し次の浮動小数点コアがサポートされています。演算子または関数に浮動小数点コアがない場合、Vivado HLS ではその浮動小数点演算子を合成できず、合成が停止します。

表 6: 浮動小数点コア

| コア              | 7 シリーズ | Virtex-6 | Virtex-5 | Virtex-4 | Spartan-6 | Spartan-3 |
|-----------------|--------|----------|----------|----------|-----------|-----------|
| FAddSub         | X      | X        | X        | X        | X         | X         |
| FAddSub_nodsp   | X      | X        | X        | -        | -         | -         |
| FAddSub_fulldsp | X      | X        | X        | -        | -         | -         |
| FCmp            | X      | X        | X        | X        | X         | X         |
| FDiv            | X      | X        | X        | X        | X         | X         |
| FMul            | X      | X        | X        | X        | X         | X         |
| FMul_nodsp      | X      | X        | X        | -        | X         | X         |
| FMul_meddsp     | X      | X        | X        | -        | X         | X         |
| FMul_fulldsp    | X      | X        | X        | -        | X         | X         |
| FMul_maxdsp     | X      | X        | X        | -        | X         | X         |
| FRSqrt          | X      | X        | X        | -        | -         | -         |
| FRSqrt_nodsp    | X      | X        | X        | -        | -         | -         |
| FRSqrt_fulldsp  | X      | X        | X        | -        | -         | -         |
| FRecip          | X      | X        | X        | -        | -         | -         |
| FRecip_nodsp    | X      | X        | X        | -        | -         | -         |
| FRecip_fulldsp  | X      | X        | X        | -        | -         | -         |
| FSqrt           | X      | X        | X        | X        | X         | X         |
| DAddSub         | X      | X        | X        | X        | X         | X         |
| DAddSub_nodsp   | X      | X        | X        | -        | -         | -         |
| DAddSub_fulldsp | X      | X        | X        | -        | -         | -         |
| DCmp            | X      | X        | X        | X        | X         | X         |
| DDiv            | X      | X        | X        | X        | X         | X         |
| DMul            | X      | X        | X        | X        | X         | X         |
| DMul_nodsp      | X      | X        | X        | -        | X         | X         |
| DMul_meddsp     | X      | X        | X        | -        | -         | -         |
| DMul_fulldsp    | X      | X        | X        | -        | X         | X         |
| DMul_maxdsp     | X      | X        | X        | -        | X         | X         |

表 6: 浮動小数点コア (続き)

| コア     | 7 シリーズ | Virtex-6 | Virtex-5 | Virtex-4 | Spartan-6 | Spartan-3 |
|--------|--------|----------|----------|----------|-----------|-----------|
| DRSqrt | X      | X        | X        | X        | X         | X         |
| DRecip | X      | X        | X        | -        | -         | -         |
| DSqrt  | X      | X        | X        | -        | -         | -         |

# 高位合成コーディングスタイルガイド

## はじめに

このセクションでは、このガイドで使用される構文の表記規則を示します。

## 表記規則

このガイドでは、次の表記規則が使用されています。

表 4-1: 構文の表記規則

| 表記規則                                                      | 説明                                                                     |
|-----------------------------------------------------------|------------------------------------------------------------------------|
| Command                                                   | コマンド構文、メニュー、またはキーボード キーを示します。                                          |
| <variable>                                                | ユーザー定義の値を示します。                                                         |
| <u>choice1</u>   choice2                                  | 選択可能な複数の項目を示します。下線が付いているものがデフォルトです。                                    |
| [option]                                                  | オプションで入力するオブジェクトを示します。                                                 |
| {repeat}                                                  | オブジェクトが 0 回以上繰り返されることを示します。                                            |
| Ctrl + c                                                  | キーボード キーの組み合わせを示します。この例の場合は、Ctrl キーを押しながら c キーを押すことを示します。              |
| [Menu] → [Item]                                           | メニュー コマンドへのパスを示します。[Item] は、[Menu] をクリックしたときにドロップダウンで表示されるメニュー コマンドです。 |
| RMB                                                       | Right Mouse Button (右マウス ボタン) の略で、文脈依存コマンドを表示します。                      |
| <variable> ::= choice<br>\$ bash_command<br>% tcl_command | 構文およびスクリプトの例を示します (bash シェル、Perl スクリプト、Tcl スクリプトなど)。                   |

## はじめに

このコーディングスタイルガイドには、ザイリンクス® FPGA デバイスのインプリメンテーション用に C コード (C++ および SystemC も含む) を記述する方法を示します。まず、Vivado™ 高位合成 (HLS) で、C コードがレジスタトランスマニアーレベル (RTL) 記述に合成されます。この後、RTL デザインがザイリンクスのゲート レベルのプリミティブに合成されます。

Vivado HLS の詳細およびザイリンクス FPGA にインプリメンテーションするためのツールフローについては、[第 2 章「高位合成ユーザー ガイド」](#) を参照してください。

最初のセクションでは、Vivado HLS を使用した C プログラミングの基礎と、HLS ツールでの C プログラミング言語のさまざまな構文のハードウェアインプリメンテーションへの合成方法について説明します。その後のセクションには、C++ および SystemC (ハードウェア動作の記述に使用される C++ ルーチンのクラス ライブラリで、[www.accellera.org](http://www.accellera.org) から入手可能) などの拡張 C 言語に関するガイドラインを示します。

**注記 :**C 言語に関する記述は C++ および SystemC にも該当するので、本書で C コードと記述されている場合は、特に記述のない限り、C だけでなく C++、SystemC にも該当します。

C コードで記述されるアルゴリズムは多くのアプリケーションで広く使用されており、標準マイクロプロセッサ (CPU)、グラフィックプロセッサ (GPU)、リアルタイムオペレーティングシステム (RTOS) で使用されるマイクロコントローラー、およびデジタルシグナルプロセッサ (DSP) を含む多くのターゲットで実行されます。どの場合も、コンパイルされた C コードが適切なパフォーマンスで実行されます。C コードはそのターゲットデバイス用に最適化され、ハイ パフォーマンスの動作になります。この章では、コードをどのように変更するとハードウェアの質およびパフォーマンスが改善できるかについて説明します。

## コード例

本書のコード例はそれぞれ Vivado HLS リリースの一部として含まれています。コード例には、次の方法でアクセスできます。

- Vivado HLS の開始ページの [Browse Examples] リンク
- Vivado HLS インストールディレクトリの examples/coding ディレクトリ

コード例のディレクトリ名は、合成の最上位関数と同じ名前になります。

コード例は Vivado HLS の GUI またはコマンド プロンプトで提供されている Tcl スクリプトを使用すると開くことができます。

このコーディング ガイドの例には、関連するヘッダーファイルについて記述されていることがよくあります。例に含まれるヘッダーファイルは、examples ディレクトリから表示できます。

**注記 :**ヘッダーファイルでは、通常最上位関数およびテストベンチのデータ型が定義されます。

## C の合成

すべての C プログラムの最上位は `main()` 関数です。高位合成では、`main()` よりも下位にある関数がすべて合成できます。Vivado HLS では、合成される関数は「最上位関数」または「デザインファイル」、これよりも上位の関数は「テストベンチ」と記述されます。テストベンチは、合成される最上位関数の動作を有効にするために使用されます。

## 最上位デザイン

通常、合成用の最上位関数はテストベンチとは分けて記述し、ヘッダーファイルを使用するようにする方法が推奨されます。

- テストベンチには、ディスクへのファイル I/O のアクセスといった、通常ハードウェアに合成できない動作が含まれます。
- ヘッダーファイルを使用することで、テストベンチとデザイン ファイルで使用される定義を共有し、アップデートできます。
- [例 4-1](#) は、`hier_func` 関数が次の 2 つの関数を呼び出すデザインを示しています。
  - `sumsub_func`: 加算および減算を実行
  - `shift_func`: シフトを実行
- `din_t`, `dint_t` および `dout_t` 型はヘッダーファイル `hier_func.h` で定義されています。

```

#include "hier_func.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
 *outSum = *in1 + *in2;
 *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
 *outA = *in1 >> 1;
 *outB = *in2 >> 2;
}

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D)
{
 dint_t apb, amb;

 sumsub_func(&A, &B, &apb, &amb);
 shift_func(&apb, &amb, C, D);
}

```

#### 例 4-1: 階層デザイン例

最上位関数には複数の関数を含めることができます、合成できるのは 1 つの最上位関数のみです。複数の関数を合成するには、それらを 1 つの最上位関数にまとめます。

たとえば、hier\_func 関数を合成するには、[例 4-1](#) に示すファイルをデザインファイルとして Vivado HLS プロジェクトに追加し、その最上位関数を hier\_func と指定します。後のセクションに説明しますが、最上位関数に対する引数 (例 4-1 では A、B、C、および D) は RTL ポートに合成され、最上位に含まれる関数 (例 4-1 では sumsub\_func および shift\_func) は階層ブロックに合成されます。

例 4-1 のヘッダー ファイル hier\_func.h には、マクロの使用方法が記述されており、`typedef` 文を使用することで、コードがさらにポータブルで読みやすくなっています。次は、`typedef` 文でデータ型を許可し、最終的な FPGA インプリメンテーションでエリアとパフォーマンスの両方が改善されるようにデータバスのビット幅を指定しています。

```

#ifndef _HIER_FUNC_H_
#define _HIER_FUNC_H_

#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

void hier_func(din_t A, din_t B, dout_t *C, dout_t *D);

#endif

```

#### 例 4-2: 階層デザイン例のヘッダー ファイル

ヘッダー ファイルには、デザイン ファイルでは必須でない NUM\_TRANS のような定義がいくつか含まれます。これらは同じヘッダー ファイルを含むテストベンチで使用されます。

## テストベンチ

ブロックの合成では、まず C 関数が正しいかどうかが検証されます。これはテストベンチで実行されるので、テストベンチが優れていると生産性がかなり上がります。

C 関数は、RTL シミュレーションよりもかなり速く実行されるので、合成よりも前に C を使用してアルゴリズムを開発および検証した方が、RTL を開発するよりも生産性が向上します。

- C の開発時間を効率的に使用するには、既知の良い結果に対して関数の結果をチェックするテストベンチを用意することが重要になります。これにより、合成前にコード変更を検証でき、アルゴリズムが正しいとわかつている状態にできます。
- Vivado HLS では、この C テストベンチを再利用して RTL デザインが検証されます (Vivado HLS を使用する場合は RTL テストベンチを作成する必要はありません)。テストベンチで最上位関数からの結果がチェックされると、その RTL がシミュレーションで自動的に検証されます。例 4-3 は、例 4-1 デザインのテストベンチを示しています。

```
#include "hier_func.h"

int main() {
 // Data storage
 int a[NUM_TRANS], b[NUM_TRANS];
 int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
 int c[NUM_TRANS], d[NUM_TRANS];

 // Function data (to/from function)
 int a_actual, b_actual;
 int c_actual, d_actual;

 // Misc
 int retval=0, i, i_trans, tmp;
 FILE *fp;

 // Load input data from files
 fp=fopen("tb_data/inA.dat", "r");
 for (i=0; i<NUM_TRANS; i++){
 fscanf(fp, "%d", &tmp);
 a[i] = tmp;
 }
 fclose(fp);

 fp=fopen("tb_data/inB.dat", "r");
 for (i=0; i<NUM_TRANS; i++){
 fscanf(fp, "%d", &tmp);
 b[i] = tmp;
 }
 fclose(fp);

 // Execute the function multiple times (multiple transactions)
 for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){

 //Apply next data values
 a_actual = a[i_trans];
 b_actual = b[i_trans];

 hier_func(a_actual, b_actual, &c_actual, &d_actual);

 //Store outputs
 c[i_trans] = c_actual;
 d[i_trans] = d_actual;
 }
}
```

```

// Load expected output data from files
fp=fopen("tb_data/outC.golden.dat", "r");
for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 c_expected[i] = tmp;
}
fclose(fp);

fp=fopen("tb_data/outD.golden.dat", "r");
for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 d_expected[i] = tmp;
}
fclose(fp);

// Check outputs against expected
for (i = 0; i < NUM_TRANS-1; ++i) {
 if(c[i] != c_expected[i]){
 retval = 1;
 }
 if(d[i] != d_expected[i]){
 retval = 1;
 }
}

// Print Results
if(retval == 0){
 printf(" *** *** *** \n");
 printf(" Results are good \n");
 printf(" *** *** *** \n");
} else {
 printf(" *** *** *** \n");
 printf(" Mismatch: retval=%d \n", retval);
 printf(" *** *** *** \n");
}

// Return 0 if outputs are correct
return retval;
}

```

### 例 4-3: テストベンチの例

## 生産的なテストベンチの作成

例 4-3 は、次のような生産的なテストベンチの属性をいくつか示しています。

- 合成される最上位関数 (hier\_func) は、NUM\_TRANS マクロの定義 (ヘッダーファイル 例 4-2 で指定) どおり、トランザクション複数回分実行されるので、多くの異なるデータ値が適用および検証できます。テストベンチは、実行されるさまざまなテストでのみ有効です。
- 関数出力は、既知の良い値に対して比較されます。既知の良い値は、この例ではファイルから読み込まれますが、テストベンチの一部として計算させることもできます。
- main() 関数の戻り値は、結果が正しいと確認されると 0、結果が既知の良い値と一致しない場合は 0 以外の値になります。



**ヒント** : テストベンチが 0 を返さない場合、Vivado HLS で実行される RTL 検証ではシミュレーション エラーがレポートされます。自動 RTL 検証を利用するには、結果が正しいと確認されたときに 0 を返すようにテストベンチを記述しておきます。

これらの属性を含むテストベンチを使用すると、合成前の C 関数への変更を素早くテストして検証でき、RTL で再利用できるので、RTL の検証が簡単になります。

## デザインファイルとテストベンチ ファイル

Vivado HLS では RTL 検証に C テストベンチが再利用されるので、テストベンチおよび関連ファイルを Vivado HLS に追加するときにテストベンチとして指定する必要があります。

テストベンチに関連付けられたファイルは、テストベンチが正しく動作するために必要なファイルです。このようなファイルの例は、[例 4-3](#) に示す `inA.dat` や `inB.dat` などで、これらは Vivado HLS プロジェクトにテストベンチファイルとして追加する必要があります。

Vivado HLS プロジェクトでテストベンチを指定するために、デザインとテストベンチを別々のファイルに分ける必要はありませんが、推奨はされます。

[例 4-1](#) は、[例 4-4](#) と同じデザインですが、最上位関数の名前が 2 つの例の区別を付けるため、`hier_func2` に変更されている点のみが違います。

`hier_func` が `hier_func2` に変更されている点を除くと、同じヘッダー ファイルとテストベンチが使用されているので、`sumsum_func` 関数を最上位関数として合成するためには Vivado HLS で次を変更する必要があります。

- Vivado HLS で `sumsub_func` を最上位関数として設定します。
- [例 4-4](#) のファイルをデザイン ファイルとプロジェクト ファイルの両方として追加します。これにより、`sumsub_func` よりも上位にある `hier_func2` 関数はテストベンチの一部になるので、RTL シミュレーションに含める必要があります。

`sumsub_func` 関数は `main()` 関数内にインスタンシエートされていなくても、残りの関数 (`hier_func2` および `shift_func`) により、正しく動作していて、テストベンチの一部であることが確認されます。

```
#include "hier_func2.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
 *outSum = *in1 + *in2;
 *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
 *outA = *in1 >> 1;
 *outB = *in2 >> 2;
}

void hier_func2(din_t A, din_t B, dout_t *C, dout_t *D)
{
 dint_t apb, amb;

 sumsub_func(&A, &B, &apb, &amb);
 shift_func(&apb, &amb, C, D);
}
```

### 例 4-4: 新しい最上位

## テストベンチとデザイン ファイルの結合

デザイン ファイルとテストベンチを 1 つのデザイン ファイルに含めることもできます。例 4-5 は 例 4-1 ~ 例 4-3 と同じ機能を持ちますが、すべてが 1 つのファイルにまとめられている点が異なります(他の例と区別するため、hier\_func 関数の名前は hier\_func3 に変更しています)。



**重要:** テストベンチとデザインが 1 つのファイルになっている場合は、ファイルをデザイン ファイルとテストベンチ ファイルの両方として Vivado HLS プロジェクトに追加する必要があります。

```
#include <stdio.h>

#define NUM_TRANS 40

typedef int din_t;
typedef int dint_t;
typedef int dout_t;

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
 *outSum = *in1 + *in2;
 *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
 *outA = *in1 >> 1;
 *outB = *in2 >> 2;
}

void hier_func3(din_t A, din_t B, dout_t *C, dout_t *D)
{
 dint_t apb, amb;

 sumsub_func(&A, &B, &apb, &amb);
 shift_func(&apb, &amb, C, D);
}

int main() {
 // Data storage
 int a[NUM_TRANS], b[NUM_TRANS];
 int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
 int c[NUM_TRANS], d[NUM_TRANS];

 //Function data (to/from function)
 int a_actual, b_actual;
 int c_actual, d_actual;

 // Misc
 int retval=0, i, i_trans, tmp;
 FILE *fp;
 // Load input data from files
 fp=fopen("tb_data/inA.dat", "r");
 for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 a[i] = tmp;
 }
 fclose(fp);
```

```
fp=fopen("tb_data/inB.dat","r");
for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 b[i] = tmp;
}
fclose(fp);

// Execute the function multiple times (multiple transactions)
for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++) {

 //Apply next data values
 a_actual = a[i_trans];
 b_actual = b[i_trans];

 hier_func3(a_actual, b_actual, &c_actual, &d_actual);

 //Store outputs
 c[i_trans] = c_actual;
 d[i_trans] = d_actual;
}

// Load expected output data from files
fp=fopen("tb_data/outC.golden.dat","r");
for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 c_expected[i] = tmp;
}
fclose(fp);

fp=fopen("tb_data/outD.golden.dat", "r");
for (i=0; i<NUM_TRANS; i++) {
 fscanf(fp, "%d", &tmp);
 d_expected[i] = tmp;
}
fclose(fp);

// Check outputs against expected
for (i = 0; i < NUM_TRANS-1; ++i) {
 if(c[i] != c_expected[i]){
 retval = 1;
 }
 if(d[i] != d_expected[i]){
 retval = 1;
 }
}

// Print Results
if(retval == 0){
 printf(" *** *** *** *** \n");
 printf(" Results are good \n");
 printf(" *** *** *** *** \n");
} else {
 printf(" *** *** *** *** \n");
 printf(" Mismatch: retval=%d \n", retval);
 printf(" *** *** *** *** \n");
}

// Return 0 if outputs are correct
```

```

 return retval;
}

```

例 4-5: テストベンチおよび最上位デザイン

## 最上位の引数: RTL インターフェイスポート

最上位関数が合成されると、関数への引数(またはパラメーター)は RTL ポートに合成されます。このプロセスは、インターフェイス合成と呼ばれます。

### インターフェイス合成

図 4-6 に示すコードは、インターフェイス合成の概要を示しています。この例には、値渡し入力 2 つ(in1 および in2)、読み出しおよび書き込み両方のポインター(sum)、関数の戻り値(temp の値)が含まれます。

```

#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

 dout_t temp;

 *sum = in1 + in2 + *sum;
 temp = in1 + in2;

 return temp;
}

```

例 4-6: インターフェイス合成の例

デフォルトでは、デザインは 図 4-1 に示すポートを使用して RTL ブロックに合成されます。

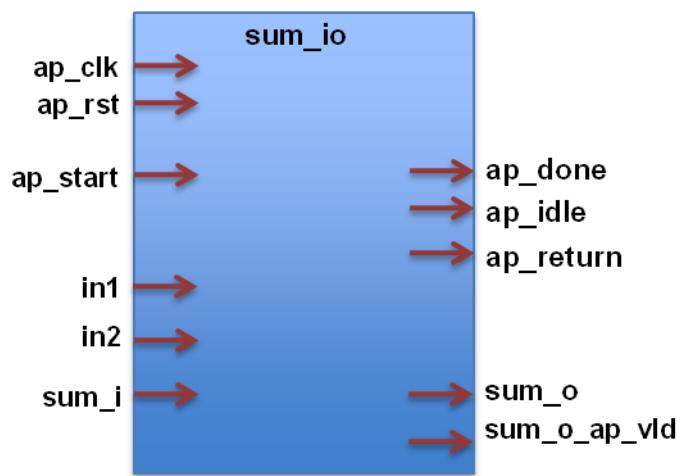


図 4-1: デフォルトのインターフェイス合成後の RTL ポート

Vivado HLS では、ポートに対して次が実行されます。そのポートについて説明します。

- クロックおよびリセットポートをデザインに追加します。
- デザイン レベルのハンドシェイク信号(ap\_start、ap\_done および ap\_idle ポート)をデフォルトで追加します。

- 関数に戻り値がない場合、ap\_return 出力ポートを RTL インターフェイスに追加します。
- Vivado HLS は関数引数からの読み出しおよび関数引数への書き込みをした後、それらを別々の入力および出力ポート (図 4-1 の sum\_i および sum\_o) に合成します。
- Vivado HLS は、デフォルトで入力の値渡し引数とポインターを単純なワイヤ ポートとして合成します。ハンドシェイク信号は関連付けられません。
- デフォルトでは、出力ポインターは関連する出力 Valid 信号を使用して合成され、出力データが有効になるタイミングを示します。

Vivado HLS が RTL ポートを合成すると、單一サイクルなのか複数サイクルなのかによって、ポートの読み出しおよび書き込みに必要なハードウェアが自動的に作成されます。例 4-6 のコードは、図 4-2 のタイミングビヘイビアを示しています (クロックサイクルごとに 1 つ追加できるターゲット テクノロジとクロック周波数であると仮定しています)。

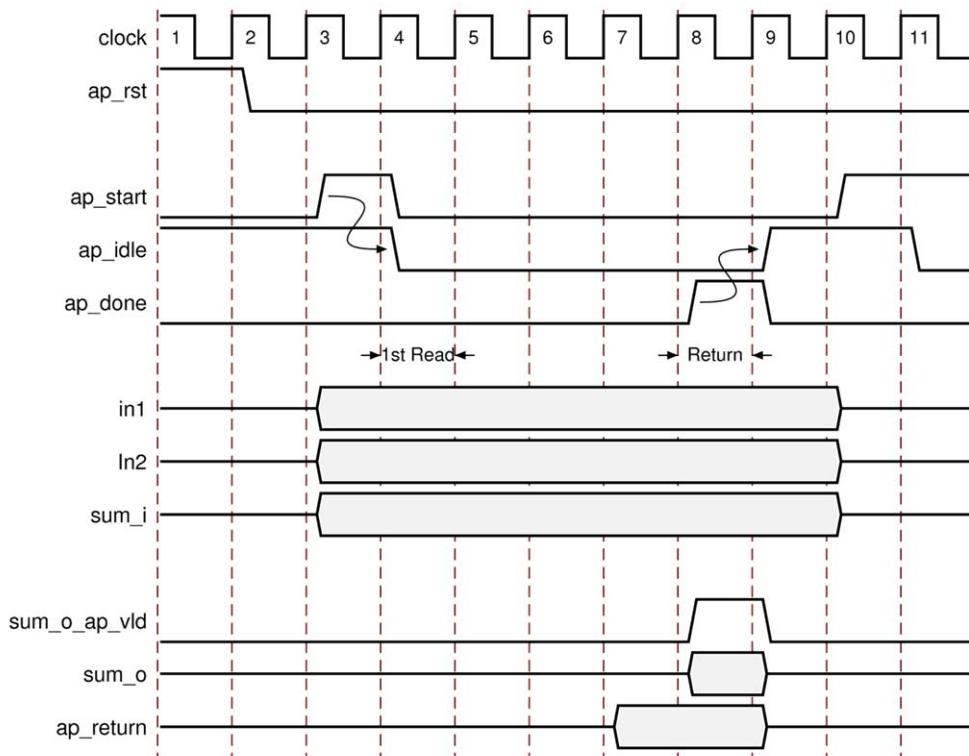


図 4-2: デフォルト合成での RTL ポートのタイミング

- デザインは ap\_start が High にアサートされると開始されます。
- ap\_idle 信号が Low にアサートされると、デザインが動作していることを示します。
- 入力データは最初のサイクル後のどのクロックでも読み出されます (読み出しのタイミングは Vivado HLS で自動的にスケジュールされます)。
- sum 出力が計算されると、データが有効かどうかが関連する出力ハンドシェイク (sum\_o\_ap\_vld) で示されます。
- 関数が終了したら ap\_done がアサートされます。これは、ap\_return のデータが有効かどうかを示します。
- ap\_idle ポートが High にアサートされると、デザインが再び開始されるのを待っている状態であることを示します。

インターフェイス合成およびそのさまざまなオプションについては、[第 2 章「高位合成ユーザー ガイド」](#)を参照してください。ここで理解する重要な点は、次のとおりです。

- インターフェイス合成では、デザインのデータ シーケンスが自動的に処理されるので、ユーザーは適切なインターフェイスを選択するだけです。
- ワイヤ ポート、單一方向および双方向のハンドシェイク、RAM アクセス ポート、および FIFO ポートなど、多くのタイプのインターフェイスが合成できます。
- インターフェイスは、タイプが違っても同じソース コードから合成できることがあります。同じコードが双方向ハンドシェイクとして指定され、`in1`、`in2`、および `sum` を使用して合成されると、RTL ポートは図 4-3 のようになります。

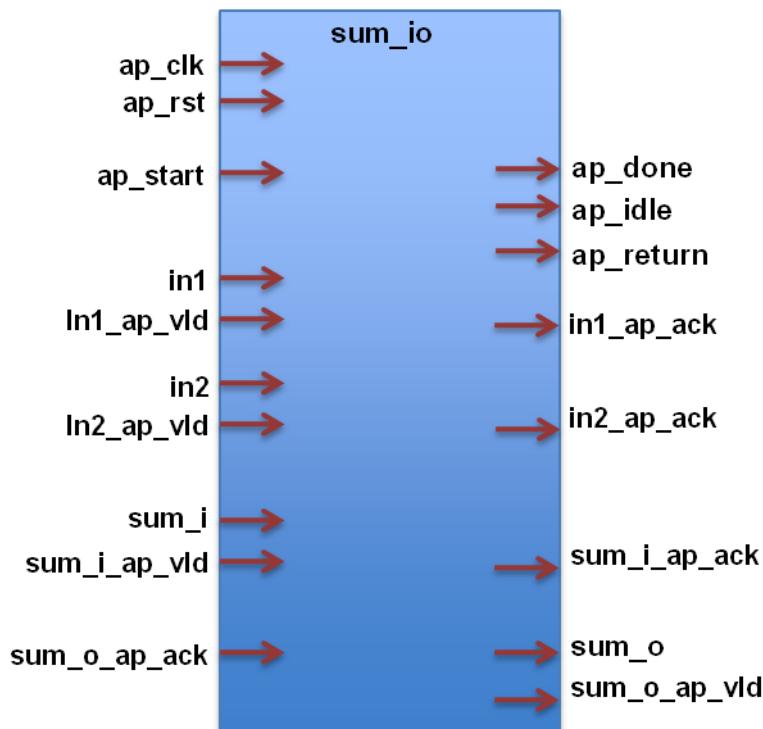


図 4-3:インターフェイス合成を指定した後の RTL ポート

ここからは、コード形式が RTL ポートに与える影響について説明します。

## ポインター

ポインターは、最上位関数への引数として使用できます。ポインターは目標どおりの RTL インターフェイスおよびデザインを合成後に達成する際、問題の原因となることがあるので、合成中にポインターがどのようにインプリメントされるか理解するのが重要になります。

### 基本的なポインター

例 4-7 のような、最上位インターフェイスに基本的なポインターを含む関数は、Vivado HLS では問題となりません。ポインターは単純なワイヤ インターフェイスかハンドシェイクを使用したインターフェイスプロトコルのいずれかに合成できます。



ヒント:FIFO インターフェイスとして合成するには、ポインターを読み出し専用または書き込み専用にする必要があります。

```

#include "pointer_basic.h"

void pointer_basic (dio_t *d) {
 static dio_t acc = 0;

```

```

 acc += *d;
 *d = acc;
}

```

#### 例 4-7: 基本的なポインター インターフェイス

テストベンチで使用する例は、[例 4-8](#) に示します。

```

#include "pointer_basic.h"

int main () {
 dio_t d;
 int i, retval=0;
 FILE *fp;

 // Save the results to a file
 fp=fopen("result.dat","w");
 printf(" Din Dout\n", i, d);

 // Create input data
 // Call the function to operate on the data
 for (i=0;i<4;i++) {
 d = i;
 pointer_basic(&d);
 fprintf(fp, "%d \n", d);
 printf(" %d %d\n", i, d);
 }
 fclose(fp);

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
 if (retval != 0) {
 printf("Test failed!!!\n");
 retval=1;
 } else {
 printf("Test passed!\n");
 }

 // Return 0 if the test
 return retval;
}

```

#### 例 4-8: 基本的なポインター インターフェイスのテストベンチ

C 関数および RTL シミュレーションでは、この単純なデータ セットを使用して正しい操作が(可能性のある操作すべてではありませんが)検証されます。

```

Din Dout
0 0
1 1
2 3
3 6
Test passed!

```

## ポインター演算

ポインター演算 (pointer\_arith) を使用すると、RTL に合成可能なインターフェイスが制限されます。例 4-9 は同じコードですが、データ値を 2 つ目の値から累積するために、単純なポインター演算が使用されている点が異なります。

```
#include "pointer_arith.h"

void pointer_arith (dio_t *d) {
 static int acc = 0;
 int i;

 for (i=0;i<4;i++) {
 acc += *(d+i+1);
 *(d+i) = acc;
 }
}
```

### 例 4-9: ポインター演算を使用したインターフェイス

例 4-10 は、この例をサポートするテストベンチです。累積を実行するためのループが pointer\_arith 関数内に含まれるようになったため、テストベンチにより、d[5] 配列で指定されたアドレス空間が適切な値を使用して生成されます。

```
#include "pointer_arith.h"

int main () {
 dio_t d[5], ref[5];
 int i, retval=0;
 FILE *fp;

 // Create input data
 for (i=0;i<5;i++) {
 d[i] = i;
 ref[i] = i;
 }

 // Call the function to operate on the data
 pointer_arith(d);

 // Save the results to a file
 fp=fopen("result.dat","w");
 printf(" Din Dout\n", i, d);
 for (i=0;i<4;i++) {
 fprintf(fp, "%d \n", d[i]);
 printf(" %d %d\n", ref[i], d[i]);
 }
 fclose(fp);

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
 if (retval != 0) {
 printf("Test failed!!!\n");
 retval=1;
 } else {
 printf("Test passed!\n");
 }

 // Return 0 if the test
```

```

 return retval;
 }

```

#### 例 4-10: ポインター演算関数のテストベンチ

これは、シミュレーションすると、次のような出力になります。

```

Din Dout
 0 1
 1 3
 2 6
 3 10
Test passed!

```

ポインター演算に関する問題は、ポインターデータへのアクセスが順番どおりでないことがあります。ワイヤ、ハンドシェイクまたは FIFO インターフェイスの場合は、順序どおりにアクセスされます。

- ワイヤインターフェイスはデザインがデータを消費するか、書き込む準備ができたときにデータを読み出します。
- ハンドシェイクおよび FIFO インターフェイスは、制御信号が処理を進める許可をしたときに読み出しおよび書き込みします。

どちらの場合も、データは順序通りにエレメント 0 から到着する(書き込まれる)必要があります。例 4-9 では、最初のデータ値がインデックス 1 から読み出されるように記述されています( $i$  が 0 で開始され、 $0+1=1$ )。これは、テストベンチの  $d[5]$  配列から 2 つ目のエレメントです。

これがハードウェアにインプリメントされる際には、何らかのデータインデックス形式が必要になります。これはワイヤ、ハンドシェイクまたは FIFO インターフェイスではサポートされません。例 4-9 のコードは、ap\_bus インターフェイスを使用してのみ合成できます。このインターフェイスでは、データがアクセス(読み出しおよび書き込み)されたときにデータにインデックスを付けるためのアドレスが提供されます。

または、ポインターではなく、インターフェイスの配列を使用してコードを例 4-11 のように修正する必要があります。これは RAM インターフェイス(ap\_memory)を使用すると、合成でインプリメントできます。このインターフェイスは、アドレスを付けてデータのインデックスを作成できるので、順序どおりでなくとも実行できます。

ワイヤ、ハンドシェイク、FIFO インターフェイスは、ストリーミングデータでのみ使用できるので、ポインター演算と一緒にには使用できません(0 からデータのインデックスを作成して、順番に進める場合は例外)。

ap\_bus および ap\_memory インターフェイス タイプの詳細は、[第 2 章「高位合成ユーザー ガイド」](#) および [第 5 章「高位合成コマンド リファレンス ガイド」](#) を参照してください。

```

#include "array_arith.h"

void array_arith (dio_t d[5]) {
 static int acc = 0;
 int i;

 for (i=0;i<4;i++) {
 acc += d[i+1];
 d[i] = acc;
 }
}

```

#### 例 4-11: 配列演算

#### マルチアクセス ポインター インターフェイス: ストリーミング データ

最上位関数の引数リストにポインターが使用されるデザインの場合、ポインターを使用して複数アクセスを実行する際に特に注意する必要があります。複数アクセスとは、同じ関数でポインターが複数回読み出されたり書き込まれたりすることです。

次に注意してください。

- 複数アクセスされる関数の引数には、`volatile` 修飾子を使用する必要があります。
- Vivado HLS 内で協調シミュレーションを使用して RTL を検証する場合、最上位関数では、このような引数に指定したポートインターフェイスのアクセス数を含める必要があります。
- 合成の前に C を検証し、目的と C 記述が一致していることを確認してください。



**推奨** : 関数の引数が複数回アクセスされる必要がある場合は、「ストリーミングデータを使用した設計」に説明するように、ストリームを使用してデザインを記述することをお勧めします。ストリームを使用すると、このセクションで示すような問題は発生しません。

このセクションでは、`pointer_stream_bad` デザイン例を使用し、同じ関数内でポインターに複数回アクセスする場合になぜ `volatile` 修飾子が必要なのか説明します。また、`pointer_stream_better` デザイン例を使用し、このようなポインターが最上位インターフェイスに含まれる場合、目的どおりの動作が正しく記述されているかどうかを確認するために、なぜ C テストベンチを使用してデザインを検証する必要があるのかについて説明します。

**例 4-12** では、入力ポインターの `d_i` が 4 回読み出され、出力ポインターの `d_o` が 2 回書き込まれ、アクセスが FIFO インターフェイスによりインプリメント (最終的な RTL インプリメンテーションからストリーミングデータを読み出しおよび書き込み) されるようにしています。

```
#include "pointer_stream_bad.h"

void pointer_stream_bad (dout_t *d_o, din_t *d_i) {
 din_t acc = 0;

 acc += *d_i;
 acc += *d_i;
 *d_o = acc;
 acc += *d_i;
 acc += *d_i;
 *d_o = acc;
}
```

#### 例 4-12 : マルチアクセス ポインター インターフェイス

**例 4-13** は、このデザインを検証するテストベンチを示しています。

```
#include "pointer_stream_bad.h"

int main () {
 din_t d_i;
 dout_t d_o;
 int retval=0;
 FILE *fp;

 // Open a file for the output results
 fp=fopen("result.dat","w");

 // Call the function to operate on the data
 for (d_i=0;d_i<4;d_i++) {
 pointer_stream_bad(&d_o,&d_i);
 fprintf(fp, "%d %d\n", d_i, d_o);
 }
 fclose(fp);

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
}
```

```

if (retval != 0) {
 printf("Test failed !!!\n");
 retval=1;
} else {
 printf("Test passed !\n");
}

// Return 0 if the test
return retval;
}

```

#### 例 4-13: マルチアクセス ポインターのテストベンチ

#### 揮発性データの理解

例 4-12 のコードは、入力ポインター `d_i` および出力ポインター `d_o` を、RTL で FIFO (またはハンドシェーク) インターフェイスとしてインプリメントすることを意図して記述されています。これにより、次のことが確実になります。

- ・ アップストリームの送信ブロックは、RTL ポート `d_i` で読み出しが実行されるたびに新しいデータを供給します。
- ・ ダウンストリームの受信ブロックは、RTL ポート `d_o` で書き込みが実行されるたびに新しいデータを受信します。

ただし、このコードを標準 C コンパイラでコンパイルすると、各ポインターへの複数アクセスが 1 つのアクセスに削減されます。コンパイラには、`d_i` 上のデータが関数の実行中に変化することは示されておらず、`d_o` への最終の書き込みのみが必要であると解釈されます (ほかの書き込みは、関数が完了するまでに上書きされます)。

Vivado HLS での処理は、gcc コンパイラの動作と一致しており、複数の読み出しおよび書き込みは 1 つの読み出し操作および 1 つの書き込み操作に最適化されます。RTL が検証されると、各ポートでは 1 つの読み出しおよび書き込み操作のみが実行されます。

このデザインの基本的な問題は、テストベンチとデザインで RTL ポートのインプリメント方法が設計者の意図どおりに記述されていないことがあります。

- ・ 設計者は、RTL ポートがトランザクション中に複数回読み出しおよび書き込み (データ入力および出力がストリーミング) されることを意図しています。
- ・ テストベンチは 1 つの入力値だけを提供し、1 つの出力値だけを戻します。例 4-12 の C シミュレーションの結果は次のようになります。各入力が 4 回累積されますが、読み出されて各回で累積されるのは同じ値で、4 回読み出しがあるわけではありません。

```

Din Dout
0 0
1 4
2 8
3 12

```

このデザインは、例 4-14 に示すように `volatile` 修飾子を使用すると、RTL ポートに複数回読み出しおよび書き込みができるようになります。

`volatile` キーワードを使用すると、C コンパイラおよび Vivado HLS でデータは揮発性であり変化する可能性があると解釈され、複数アクセスが 1 つのアクセスに最適化されることはなくなります。

```

#include "pointer_stream_better.h"

void pointer_stream_better (volatile dout_t *d_o, volatile din_t *d_i) {
 din_t acc = 0;

 acc += *d_i;
 acc += *d_i;
 *d_o = acc;
}

```

```

acc += *d_i;
acc += *d_i;
*d_o = acc;
}

```

#### 例 4-14: マルチアクセス volatile ポインター インターフェイス

例 4-14 は 例 4-12 と同じようにシミュレーションされますが、volatile 修飾子によりポインターアクセスの最適化がされないので、入力ポート `d_i` で 4 回の読み出し、出力ポート `d_o` で 2 回の書き込みが実行される RTL デザインになります。

ただし、volatile キーワードを使用しても、このコード形式（ポインターに複数回アクセスする）には、関数とテストベンチに読み出しおよび書き込みが明示的に記述されていないという問題がまだあります。

この場合、読み出しが 4 回実行されますが、同じデータが 4 回読み込まれてしまいます。書き込みは 2 回別々にあり、それぞれ正しいデータで実行されますが、テストベンチは最後の書き込みのデータのみを取り込みます（cosim\_design をイネーブルにして RTL シミュレーション中にトレースファイルを作成し、その VCD ファイルを確認すると、中間アクセスを表示可能です）。

**注記**：例 4-14 は、ワイヤインターフェイスを使用するとインプリメントできますが、FIFO インターフェイスが指定される場合は、Vivado HLS で各読み出しごとに新しいデータをストリーミングする RTL テストベンチが作成されます。この場合、テストベンチから使用できる新しいデータはないので、RTL は検証できません。ここでの問題は、テストベンチで正しく読み出しおよび書き込みが記述されていないことにあります。

#### ストリーミング データ インターフェイスの記述

ハードウェアシステムには並列処理機能があるので、ソフトウェアと違ってストリーミングデータの利点を生かすことができます。データはデザインに連続して提供され、デザインからデータは連続して出力されます。このため、RTL デザインは既存データを処理し終わる前に新しいデータを受信できます。

例 4-14 に示すように、ソフトウェアでのストリーミングデータの記述は、特に既存のハードウェアインプリメンテーション（既に並列/ストリーミング処理機能が存在し、記述する必要あり）を表すソフトウェアを記述する際に重要です。

これには、次のように複数の方法があります。

- 単に volatile 指示子を追加します（例 4-14）。テストベンチには一意の読み出しおよび書き込みが記述されないので、元の C テストベンチを使用した RTL シミュレーションはエラーになりますが、VCD 波形を確認すると、正しい読み出しおよび書き込みが実行されたことがわかります。
- 一意の読み出しおよび書き込みを明示的に記述するようにコードを変更します。これは、例 4-15 に示します。
- ストリーミングデータ型を使用するようにコードを変更します。ストリーミングデータ型を使用すると、ストリーミングデータを使用してハードウェアが適切に記述されるようになります。この詳細は、第 2 章「高位合成ユーザーガイド」で説明します。

例 4-15 のコードは、テストベンチから 4 つの異なる値が読み出され、2 つの異なる値が書き込まれるようにアップデートしたコードです。ポインターのアクセスはシーケンシャルでロケーション 0 から開始するので、合成ではストリーミングインターフェイスが使用できます。

```

#include "pointer_stream_good.h"

void pointer_stream_good (volatile dout_t *d_o, volatile din_t *d_i) {
 din_t acc = 0;

 acc += *d_i;
 acc += *(d_i+1);
 *d_o = acc;
 acc += *(d_i+2);
 acc += *(d_i+3);
 *(d_o+1) = acc;
}

```

#### 例 4-15 : 明示的なマルチアクセスの volatile ポインター インターフェイス

テストベンチがアップデートされ、関数が各トランザクションで 4 つの固有の値を読み出すように記述されます。新しいテストベンチには、1 つのトランザクションのみが記述されます。複数のトランザクションを記述するには、入力データ セットを増加し、関数を複数回呼び出す必要があります。

```

#include "pointer_stream_good.h"

int main () {
 din_t d_i[4];
 dout_t d_o[4];
 int i, retval=0;
 FILE *fp;

 // Create input data
 for (i=0;i<4;i++) {
 d_i[i] = i;
 }

 // Call the function to operate on the data
 pointer_stream_good(d_o,d_i);

 // Save the results to a file
 fp=fopen("result.dat","w");
 for (i=0;i<4;i++) {
 if (i<2)
 fprintf(fp, "%d %d\n", d_i[i], d_o[i]);
 else
 fprintf(fp, "%d\n", d_i[i]);
 }
 fclose(fp);

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
 if (retval != 0) {
 printf("Test failed !!!\n");
 retval=1;
 } else {
 printf("Test passed !\n");
 }

 // Return 0 if the test
 return retval;
}

```

#### 例 4-16 : 明示的なマルチアクセスの volatile ポインターのテストベンチ

このテストベンチにより、次のような結果が得られます。1つのトランザクションから2つの出力が生成され、1つの出力は最初の2つの入力読み出しを加算したもの、2つ目の出力は次の2つの入力読み出しと1つ目の出力を加算したものになります。

```
Din Dout
0 1
1 6
2
3
```

ポインターが関数インターフェイスで複数回アクセスされる際に注意すべき最後の問題は、RTLシミュレーションの記述です。

### マルチアクセス ポインターおよび RTL シミュレーション

`cosim_design` で RTL を検証するため、Vivado HLS では RTL にアダプターを含む SystemC ラッパーが作成されます。このラッパーが既存の C テストベンチにインスタンシエートされます(図 4-3)。

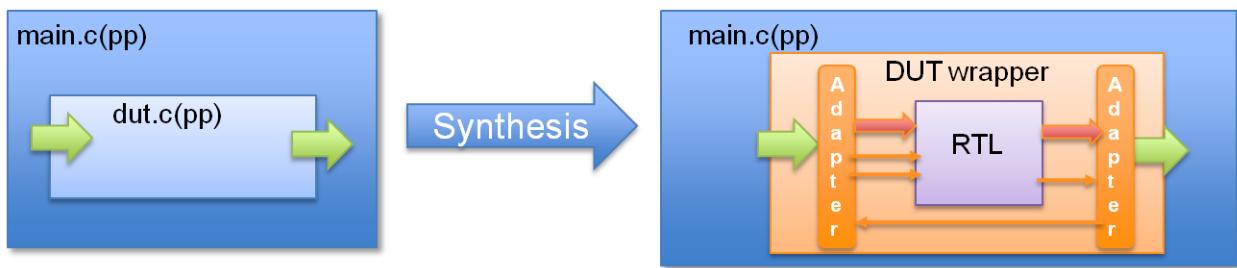


図 4-3 : cosim\_design ラッパーの概要

Vivado HLS で作成されるラッパーには、RTL インターフェイスにハンドシェークが必要な場合はそれが記述され、テストベンチから供給される DUT への入力値が RTL デザインで必要なときに準備されるようにする必要があります。これには、アダプターにストレージが必要です。

インターフェイスのポインターが複数回アクセスされた場合、Vivado HLS では関数インターフェイスから何度読み出しおよび書き込みが実行されたか判別できません。値がいくつ読み出されるか、または書き込まれるかを Vivado HLS に示すような関数インターフェイスの引数はありません。

```
void pointer_stream_good (volatile dout_t *d_o, volatile din_t *d_i)
```

### 例 4-17 : volatile ポインター インターフェイス

配列の最大サイズなど、インターフェイスに値がいくつ必要かを示すようなものがない限り、Vivado HLS では1つの値であると想定され、1つの入力および1つの出力用のシミュレーションラッパーが作成されます。

RTL ポートが実際には複数の値を読み出しありは書き込みする場合、これにより RTL の `cosim_design` シミュレーションは停止します。ラッパーには RTL デザインに接続される送信ブロックと受信ブロックが記述されます。1つの値しか記述されない場合、複数の値を読み出しありは書き込もうとすると、RTL デザインが停止します。

インターフェイスでマルチアクセス ポインターが使用される場合、インターフェイスでの読み出しありは書き込みの最大回数を Vivado HLS に示す必要があります。インターフェイスを指定する場合、[Vivaod HLS Directive Editor] ダイアログ ボックスの [Directive] で [INTERFACE] を選択し、[depth] オプションを設定します(188 ページの図 3)。

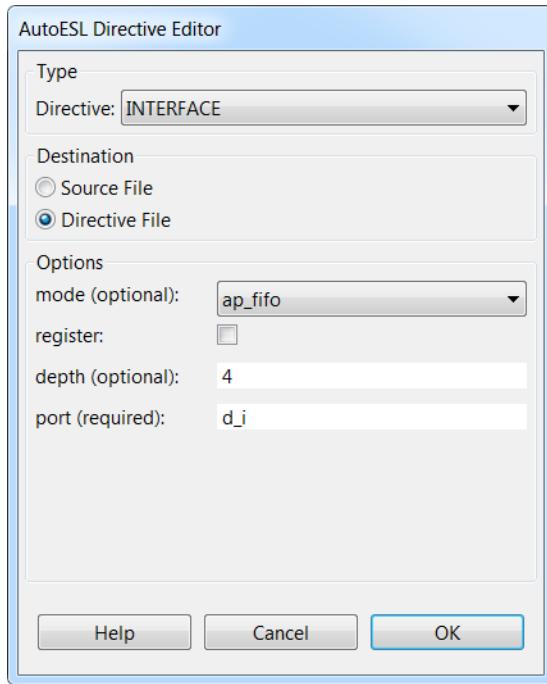


図 4-3 : [Vivado HLS Directive Editor] ダイアログ ボックス : [depth] オプション

上記の例では、引数/ポート `d_i` の (FIFO インターフェイスの) 深さが 4 に設定されており、`cosim_design` で RTL を正しく検証するのに十分な値が供給されています。

## インターフェイスの配列

Vivado HLS では、配列がメモリ エレメントにデフォルトで合成されます。配列が最上位関数への引数として使用されると、メモリはオフチップであると仮定され、インターフェイス ポートがメモリにアクセスするために合成されます。

Vivado HLS には、これらのポートの作成方法を設定する機能があります。

- メモリは、シングルまたはデュアルポート RAM として指定できます。
- インターフェイスは、FIFO インターフェイスとして指定できます。
- Vivado HLS の配列最適化指示子 (`partition`、`map` および `reshape`) を使用すると、配列の構造を設定し直せ、IO ポートの数も変更できます。

インターフェイスの配列がデザインにもたらす主な問題は、データへのアクセスがメモリ (RAM または FIFO) ポートからに制限されるために、パフォーマンスに障害が出ることです。これらの問題は、通常指示子を使用すると回避できます。

合成可能なコードで配列を使用する場合は、配列を必ずサイズ指定する必要があります。たとえば、例 4-18 の宣言 `d_i[4]` が `d_i[]` に変更されると、Vivado HLS でデザインが合成できないことを示すメッセージが表示されます。

```
@E [SYNCHK-61] array_RAM.c:52: unsupported memory access on variable 'd_i' which is
(or contains) an array with unknown size at compile time.
```

## 配列最適化指示子

どのタイプの RAM を使用するか (シングル ポートとデュアル ポートのどちらの RAM ポートを作成するか) を明示的に指定するには、`resource` 指示子を使用します。`resource` 指示子が指定されない場合、Vivado HLS はデフォルトの

シングルポート RAM を使用しますが、スループットの改善やレイテンシの削減が可能な場合はデュアルポート RAM を使用します。

インターフェイスで配列をリコンフィギュレーションするには、`partition`、`map` および `reshape` 指示子を指定します。配列は複数の小さい配列に分割でき、それぞれに別のインターフェイスを使用できます。これには、配列のすべてのエレメントをスカラーエレメントに分割できる機能も含まれます。関数インターフェイスの場合は、配列のすべてのエレメントに対してそれぞれ別のポートが作成されます。これにより並列アクセスは最大になりますが、さらにポートが作成されるため、上の階層で配線問題が発生することがあります。

同様に、小さい配列は 1 つの大きな配列にまとめられ、1 つのインターフェイスになってしまふことがあります。この場合、オフチップ BRAM へのマップは改善されますが、パフォーマンスに障害が出る可能性があることに注意してください。これらのトレードオフは Vivado HLS の最適化指示子を使用すると発生するので、コード自体には影響しません。

## RAM インターフェイス

例 4-18 に示す関数の配列引数は、デフォルトでシングルポート RAM インターフェイスに合成されます。

```
#include "array_RAM.h"

void array_RAM (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
 int i;

 For_Loop: for (i=0;i<4;i++) {
 d_o[i] = d_i[idx[i]];
 }
}
```

### 例 4-18: RAM インターフェイス

シングルポート RAM インターフェイスが使用されるのは、`for` ループで各クロックサイクルの読み出しおよび書き込みができるのは 1 エレメントのみであるため、デュアルポート RAM インターフェイスを使用する利点がないからです。

`for` ループが展開されると、同時に複数エレメントを読み出すことができ、スループットが増加するため、Vivado HLS では自動的にデュアルポートが使用されます。RAM インターフェイスのタイプは、`resource` 指示子を適用すると設定できます。

前述のとおり、インターフェイスで配列に関する問題がある場合は、通常スループットに関係しており、最適化指示子で処理できます。たとえば、例 4-18 の配列が個別エレメントに分割され、`for` ループが展開されていれば、各配列の 4 つのエレメントすべてが同時にアクセスされます。

## FIFO インターフェイス

Vivado HLS では、配列引数を RTL で FIFO ポートとしてインプリメントできます。FIFO ポートを使用する予定がある場合は、配列のアクセスがシーケンシャルになっていることを確認してください。Vivado HLS ではアクセスがシーケンシャルかどうかを確認する解析が実行されます。

- アクセスがシーケンシャルであると確認されると、FIFO ポートがインプリメントされます。
- シーケンシャルでないと判断される場合は、エラー メッセージが表示され、合成が停止します。
- シーケンシャルかどうかが判断できない場合は、警告メッセージが表示され、FIFO ポートのインプリメンテーションに進みます。アクセスが実際にシーケンシャルではなかった場合、RTL シミュレーション不一致になります。

例 4-19 は、Vivado HLS がアクセスがシーケンシャルかどうか判断できない例を示しています。この例では、`d_i` と `d_o` の両方が合成中に FIFO インターフェイスを使用してインプリメントされるように指定されています。

```

#include "array_FIFO.h"

void array_FIFO (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
 int i;

 // Breaks FIFO interface d_o[3] = d_i[2];
 For_Loop: for (i=0;i<4;i++) {
 d_o[i] = d_i[idx[i]];
 }
}

```

### 例 4-19 : FIFO インターフェイスのストリーミング

この例の場合、FIFO インターフェイスが問題なく作成できるかどうかを判断する `idx` が使用されています。

- `idx` がシーケンシャルに増加すれば、FIFO インターフェイスが作成できます。
- `idx` にランダムな値が使用されると、FIFO インターフェイスが RTL にインプリメントされる際にエラーになります。

このインターフェイスは問題ない可能性もあるため、Vivado HLS は合成中に次のような警告メッセージを表示し、FIFO インターフェイスの作成に進みます。

```
@W [XFORM-124] Array 'd_i': may have improper streaming access(es).
```

例 4-19 のコメント ("//Breaks FIFO interface") が削除されると、Vivado HLS では配列へのアクセスがシーケンシャルではないと自動的に判断され、FIFO インターフェイスが指定されている場合はエラー メッセージが表示されて停止します。

**注記** : FIFO ポートは読み出しおよび書き込み配列用には合成されません。入力配列と出力配列は例 4-19 のように別々に作成する必要があります。

次の一般的な規則は、ストリーミングされる (FIFO インターフェイスでインプリメントされた) 配列に適用されます。

- 配列は、1 ループまたは関数でのみ読み出し/書き込みする必要があります。これは FIFO リンクの特性と一致するポイント トゥ ポイントの接続に変換される可能性があります。
- 配列の読み出しへは、配列の書き込みと同じ順序である必要があります。FIFO チャネルではランダム アクセスがサポートされないので、配列は先入れ先出し (First In First Out) 動作に従ったプログラムで使用される必要があります。
- FIFO からの読み出しおよび書き込みに使用されるインデックスは、コンパイル時に解析される必要があります。ランタイム計測に基づいた配列のアドレス指定は、FIFO 動作用には解析できないので、配列が FIFO に変換されなくなります。

最上位インターフェイスに配列をインプリメントまたは最適化するのに、コードを変更する必要は通常ありません。インターフェイスの配列のコードを変更する必要があるのは、配列が構造体 (struct) の一部である場合のみです。

## インターフェイスの構造体

構造体 (struct) が最上位関数への引数として使用されると、その構造体が渡し値引数であるかポインターであるかによって、合成で作成されるポートは異なります。

例 4-20 の場合、`struct data_t` がヘッダーファイルで定義されています。この構造体には、次の 2 つのデータメンバーが含まれます。

- `short` 型 (16 ビット) の符号なしベクター `A`
  - 4 つの `unsigned char` 型 (8 ビット) の配列 `B`
- ```

typedef struct {
    unsigned short A;

```

```

unsigned char B[4];
} data_t;

data_t struct_port(data_t i_val, data_t *i_pt, data_t *o_pt);

```

例 4-20: ヘッダー ファイルの構造体宣言

例 4-21 では、構造体が渡し値引数 (i_val から o_val の戻り値) とポインター (*i_pt から *o_pt) の両方として使用されています。どちらの方法でも加算後に入力から出力に渡される結果は同じようになりますが、渡し値引数とポインター引数がポートとして合成される方法が異なります。

```

#include "struct_port.h"

data_t struct_port(
    data_t i_val,
    data_t *i_pt,
    data_t *o_pt
) {

    data_t o_val;
    int i;

    // Transfer pass-by-value structs
    o_val.A = i_val.A+2;
    for (i=0;i<4;i++) {
        o_val.B[i] = i_val.B[i]+2;
    }

    // Transfer pointer structs
    o_pt->A = i_pt->A+3;
    for (i=0;i<4;i++) {
        o_pt->B[i] = i_pt->B[i]+3;
    }

    return o_val;
}

```

例 4-21: 渡し値およびポインターとしての構造体

渡し値入力引数の場合、構造体の配列は完全に別々のエレメントに分割されます。

- struct エレメント A は 16 ビット ポートになります。
- struct エレメント B は 4 つの 8 ビット ポートになります。

ポインターおよび関数戻り値の場合、構造体の配列は標準配列と同じように合成され、次のメモリ インターフェイスになります。

- struct エレメント A は 16 ビット ポートになります。
- struct エレメント B は 4 つのエレメントにアクセスする RAM ポートになります。

大きな配列を含む構造体を使用する場合、渡し値構造体をポインターに変換しておくことをお勧めします。変換されない場合、配列が個別エレメントに完全に分割され、それぞれ独自のポートでインプリメントされます。たとえば、配列に 1024 エレメントが含まれる場合は、1024 個の別々の RTL ポートを使用してインプリメントされます。

Vivado HLS で合成できる構造体のサイズや複雑さに制限はありません。構造体には必要なだけの配列次元およびメンバーを含めることができます。構造体のインプリメンテーションでの唯一の制限は、ストリーミングとして(たとえば配列が FIFO インターフェイスとして)インプリメントされる場合にあります。この場合、そのインターフェイスの配列に適用するのと同じ一般規則に従う必要があります。

データ型

実行ファイルへコンパイルされる C 関数で使用されるデータ型は、結果の精度、メモリ要件、パフォーマンスに影響します。

- 32 ビット整数の `int` データ型には、さらに多くのデータを保持できるので、8 ビットの `char` 型よりも精度が高くなりますが、さらに多くのストレージが必要となります。
- 64 ビットの `long long` 型が 32 ビットシステムで使用されると、このような値を読み出しおよび書き込みするためには通常複数アクセスが必要になるので、ランタイムに影響がでます。

同様に、C 関数が RTL インプリメンテーションに合成される場合も、データ型が RTL デザインの精度、エリア、パフォーマンスに影響します。これは、変数に使用されるデータ型により、必要な演算子のサイズが決まるからです。

Vivado HLS では、固定長整数型を含むすべての標準 C データ型の合成がサポートされます。

- `(unsigned) char`、`(unsigned) short`、`(unsigned) int`
- `(unsigned) long`、`(unsigned) long long`
- `(unsigned) intN_t` (N は `stdint.h` で定義される 8、16、32 および 64)
- `float`、`double`

固定長整数型を使用すると、デザインをシステムのすべてのデータ型間でポータブルにできます。

注記 : 整数型 `(unsigned) long` は 64 ビットを 64 ビット OS、32 ビットを 32 ビット OS としてインプリメントします。合成ではこのビヘイビアが比較され、Vivado HLS が実行される OS タイプによって、異なるサイズの演算子が生成されるので、異なる RTL デザインが生成されます。

32 ビットの場合、`(unsigned) long` データ型の代わりに、`(unsigned) int` または `(unsigned) int32_t` を使用する必要があります。

64 ビットの場合、`(unsigned) long` データ型の代わりに、`(unsigned) long long` または `(unsigned) int64_t` を使用する必要があります。

標準データ型

例 4-22 は、実行される基本的な演算子のいくつかを示しています。

```
#include "types_standard.h"

void types_standard(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
                     dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}
```

例 4-22 : 基本的演算

例 4-22 のデータ型は例 4-23 に示すヘッダーファイル `types_standard.h` で定義されています。標準符号付き型、符号なし型、およびヘッダーファイル `stdint.h` を含めることで固定長整数型が使用されます。

```
#include <stdio.h>
#include <stdint.h>
```

```

#define N 9

typedef char din_A;
typedef short din_B;
typedef int din_C;
typedef long long din_D;

typedef int dout_1;
typedef unsigned char dout_2;
typedef int32_t dout_3;
typedef int64_t dout_4;

void types_standard(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);

```

例 4-23: 基本的な演算データ型定義

これらのデータ型は、合成後、次の演算子およびポート サイズになります。

- out1 結果を計算するために使用される乗算器は 24 ビット乗算器になります。これは 8 ビットの char 型が 16 ビットの short で乗算されるには、24 ビット乗算器が必要だからです。この結果は、出力ポート幅と一致するように 32 ビットまで符号拡張されます。
- out2 に使用される加算器は出力が 8 ビットの unsigned char 型なので 8 ビットになり、inB (16 ビットの short) の下位 8 ビットのみが 8 ビットの char 型の inA へ追加されます。
- out3 (32 ビットの固定幅型) 出力の場合、8 ビットの char 型の inA が 32 ビット値に拡張され、32 ビット (int 型) の inC 入力を使用して 32 ビットの除算演算が実行されます。
- 同様に、64 ビット モジュールの演算は 64 ビットの long long 型 inD と 64 ビットに符号拡張された 8 ビットの char 型 inA を使用して実行され、64 ビットの出力結果 out4 が作成されます。

out1 結果が示すように、Vivado HLS では可能な限り最小の演算子が使用され、結果が単に拡張されて、必要な出力ビット幅に一致するようにされます。同様に、out2 結果の場合、入力の 1 つが 16 ビットであっても、8 ビット出力しか必要でないため、8 ビット加算器が使用できます。ただし、out3 および out4 結果が示すように、すべてのビットが必要であれば、フルサイズの演算子が合成されます。

float および double 型

Vivado HLS では、合成で float および double 型がサポートされます。どちらのデータ型も IEEE-754 規格に従つて合成されます。

- 単精度 (32 ビット): 仮数部 24 ビット、指数部 8 ビット
- 倍精度 (64 ビット): 仮数部 53 ビット、指数部 11 ビット

float および double 型は、標準演算 (+、-、* など) に使用するだけでなく、math.h (C++ の場合は cmath.h) でもよく使用されます。このセクションでは、標準演算がどのようにサポートされるか示します。C および C++ の math ライブ ラリを合成する方法の詳細は、「高位合成演算子およびコア ガイド」の章を参照してください。

例 4-24 は、例 4-22 で使用されたヘッダーファイルに double および float データ型を定義したものです。

```

#include <stdio.h>
#include <stdint.h>
#include <math.h>

#define N 9

typedef double din_A;
typedef double din_B;

```

```

typedef double din_C;
typedef float din_D;

typedef double dout_1;
typedef double dout_2;
typedef double dout_3;
typedef float dout_4;

void types_float_double(din_A inA,din_B inB,din_C inC,din_D inD,dout_1
*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);

```

例 4-24 : float および double 型

このアップデートされたヘッダーファイルは、`sqrtf()` 関数が使用される [例 4-25](#) で使用されます。

```

#include "types_float_double.h"

void types_float_double(
    din_A inA,
    din_B inB,
    din_C inC,
    din_D inD,
    dout_1 *out1,
    dout_2 *out2,
    dout_3 *out3,
    dout_4 *out4
) {

    // Basic arithmetic & math.h sqrtf()
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = sqrtf(inD);
}

}

```

例 4-25 : float および double 型の使用

[例 4-25](#) が合成されると、64 ビットの倍精度乗算、加算、除算演算子になり、これらの演算子は適切な浮動小数点のザイリンクス CORE Generator コアでインプリメントされます。

`sqrtf()` を使用する平方根は、32 ビットの单精度浮動小数点コアを使用してインプリメントされます。倍精度の平方根関数 `sqrt()` が使用されると、`inD` で使用される单精度 `float` 型の型変換のためにロジックが追加されます。`out4: sqrt()` は倍精度 (`double`) 関数、`sqrtf()` は单精度 (`float`) 関数です。



注意 : C 関数では、`float-to-double` および `double-to-float` 変換ユニットがハードウェアで推論されるので、`float` 型と `double` 型を混合する場合には注意が必要です。

次のようなコードがあるとします。

```

float foo_f      = 3.1459;
float var_f = sqrt(foo_f);

```

これは次のようなハードウェアになります。

```

wire(foo_t)
→ Float-to-Double Converter unit
→ Double-Precision Square Root unit
→ Double-to-Float Converter unit
→ wire (var_f)

```

`sqrtf()` 関数を使用すると、ハードウェアで型コンバーターが必要なくなるので、エリアが節約でき、タイミングが改善します。

`float` 型および `double` 型の演算は、浮動小数点演算子の LogicCORE コアに合成されます。表 2-1 は、各ザイリンクス ファミリーで使用可能なコアを示しています。

テクノロジで特定の LogicCORE エレメントがサポートされない(表 2-1 で X で表示される) 場合は、デザインが合成できず、エラーメッセージが表示されて Vivado HLS が停止します。

表 2-1: 浮動小数点コア

コア	7 シリーズ	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X

表 2-1 のコアでは、多数の DSP48 が使用されるか、1 つも使用されないようなコア(例: `DMul_nodsp` および `DMul_maxdsp`) で演算がインプリメントされることがあります。Vivado HLS はデフォルトでは最大数の DSP48 を含むコアを使用して演算をインプリメントします。または、`resource` 指示子を使用してどのコアを使用するべきか明示的に指定することもできます。

`float` および `double` 型を合成する際には、Vivado HLS が C コードで実行される演算順序を維持して、C シミュレーションと結果が同じになるようにします。飽和および切り捨てのため、次は単精度および倍精度演算で必ずしも同じになるわけではありません。

```

A=B*C;          A=B*F;
D=E*F;          D=E*C;
O1=A*D;         O2=A*D;

```

`float` および `double` 型を使用する場合、`O1` と `O2` は必ずしも同じになるわけではありません。



ヒント : デザインによっては、ループの展開や部分展開などの最適化により、並列計算の利点を生かすことのできる場合もあります。これは、float および double 型を合成する際に Vivado HLS で演算順序が厳しく守られるからです。

C++ デザインの場合は、Vivado HLS の最もよく使用される数学関数のビットを概算する機能を使用できます。

複合データ型

合成では、次の複合データ型がサポートされます。

- array
- enum
- struct
- union

例 4-26 のヘッダーファイルでは、enum 型を定義し、それらを struct で使用しています。これが別の struct で使用されることで、複雑なデータ型がわかりやすく記述できます。

また、例 4-26 は、複雑な定義文 (MAD_NSBSAMPLES) の指定および合成方法も示しています。

```
#include <stdio.h>

enum mad_layer {
    MAD_LAYER_I    = 1,
    MAD_LAYER_II   = 2,
    MAD_LAYER_III  = 3
};

enum mad_mode {
    MAD_MODE_SINGLE_CHANNEL = 0,
    MAD_MODE_DUAL_CHANNEL  = 1,
    MAD_MODE_JOINT_STEREO   = 2,
    MAD_MODE_STEREO         = 3
};

enum mad_emphasis {
    MAD_EMPHASIS_NONE = 0,
    MAD_EMPHASIS_50_15_US = 1,
    MAD_EMPHASIS_CCITT_J_17 = 3
};

typedef signed int mad_fixed_t;

typedef struct mad_header {
    enum mad_layer layer;
    enum mad_mode mode;
    int mode_extension;
    enum mad_emphasis emphasis;

    unsigned long long bitrate;
    unsigned int samplerate;

    unsigned short crc_check;
    unsigned short crc_target;

    int flags;
    int private_bits;
}
```

```

    } header_t;

typedef struct mad_frame {
    header_t header;
    int options;
    mad_fixed_t sbsample[2][36][32];
} frame_t;

#define MAD_NSBSAMPLES(header) \
((header)->layer == MAD_LAYER_I ?12 : \
((header)->layer == MAD_LAYER_III && \
((header)->flags & 17) ?18 :36))

void types_composite(frame_t *frame);

```

例 4-26: enum、struct および複雑な定義

例 4-26 では、例 4-27 で定義された struct および enum 型が使用されています。標準 C のコンパイルでは、enum 型が 32 ビット値であると仮定され、合成後に 32 ビット値になります。

また、例 4-27 は、合成中に printf 文が自動的に無視されるように記述しています。

```

#include "types_composite.h"

void types_composite(frame_t *frame)
{
    if (frame->header.mode != MAD_MODE_SINGLE_CHANNEL) {
        unsigned int ns, s, sb;
        mad_fixed_t left, right;

        ns = MAD_NSBSAMPLES(&frame->header);
        printf("Samples from header %d \n", ns);

        for (s = 0; s < ns; ++s) {
            for (sb = 0; sb < 32; ++sb) {
                left = frame->sbsample[0][s][sb];
                right = frame->sbsample[1][s][sb];
                frame->sbsample[0][s][sb] = (left + right) / 2;
            }
        }
        frame->header.mode = MAD_MODE_SINGLE_CHANNEL;
    }
}

```

例 4-27: 複雑なデータ型の使用

例 4-28 では、double および struct を使用して union が作成されています。C コンパイルと異なり、合成で必ず同じメモリ (レジスタ) が union のすべてのフィールドに使用されるとは限りません。Vivado HLS では、どのような最適化でも最適なハードウェアが提供されます。

注記: ポインター再変換は合成ではサポートされません。この場合、union はポインターを別のデータ型 (または別のデータ型の配列) に保持できません。

```

#include "types_union.h"

dout_t types_union(din_t N, dinfp_t F)
{
    union {

```

```

        struct { int a; int b; } intval;
        double fpval;
    } intfp;
    unsigned long long one, exp;

    // Set a floating-point value in union intfp
    intfp.fpval = F;

    // Slice out lower bits and add to shifted input
    one = intfp.intval.a;
    exp = (N & 0x7FF);

    return ((exp << 52) + one) & (0x7fffffffffffffLL);
}

```

例 4-28 : union

型修飾子

型修飾子は、高位合成で作成されるハードウェアに直接影響します。通常、修飾子は次に示すように合成結果に影響を与えますが（予測可能）、Vivado HLS は修飾子の解釈によってのみ制限されます。これは、修飾子が関数ビヘイビアに影響し、最適化を実行して最適なハードウェア デザインを作成できるためです。この例は、各修飾子の概要の後に示します。

volatile

volatile 修飾子は、ポインターが関数インターフェイスで複数回アクセスされるときの、読み出しありまたは書き込みの実行回数に影響します。volatile 修飾子は、階層内のすべての関数に影響を与えます。volatile 修飾子については、最上位インターフェイスに関するセクションで説明しています（[184 ページの「揮発性データの理解」](#)を参照）。

static

関数内の static は、関数呼び出し間の値を保持します。ハードウェア デザインの同等のビヘイビアは、レジスタ付きの変数（フリップフロップまたはメモリ）です。正しく実行されるために変数を C 関数の static 型にする必要がある場合、最終 RTL デザインでは確実にレジスタになります。値は、関数およびデザインの起動中維持されている必要があります。

ただし、static 型だけが合成後にレジスタになるというわけではありません。RTL デザインでどの変数がレジスタとしてインプリメントされる必要があるかは、Vivado HLS で判断されます。たとえば、変数引数が複数サイクル間保持される必要がある場合、C 関数の元の変数が static 型でなくとも、Vivado HLS ではその値を保持するためにレジスタが作成されます。

Vivado HLS は static の初期化動作に従って、初期化中にレジスタへ自動的に値を 0 に割り当てるか、指定された初期化値に割り当てます。つまり、static 変数は RTL コードと FPGA ビットストリームで初期化されます。ただし、リセット信号がアサートされるたびに変数が初期化し直されるわけではありません。

注記 : static 初期化値をシステム リセット時にインプリメントする方法については、RTL コンフィギュレーション（`config_rtl` コマンド）を参照してください。

const

const 型では、変数の値がアップデートされないことを指定します。変数は読み出されますが、書き込まれることはないので、初期化される必要があります。ほとんどの const 変数は、通常 RTL デザインでは定数に削減されます（Vivado HLS は定数伝搬を実行して、不必要的ハードウェアを削除します）。

ただし、配列の場合、const 変数は最終 RTL デザインで ROM としてインプリメントされます（小さい配列では Vivado HLS で自動分割が実行されないため）。const 修飾子で指定された配列は、static の場合と同様、RTL および FPGA ビットストリームで初期化されます（これらは書き込まれることがないため、リセットする必要はありません）。

Vivado HLS の最適化

例 4-29 では、配列が `static` または `const` 修飾子で指定されていなくても、Vivado HLS で ROM がインプリメントされる例を示しています。これは、Vivado HLS がデザインを解析し、最適なインプリメンテーションを自動的に判断することを示します。修飾子の有無による影響はありますが、最終的な RTL はこれによって決定はされません。

```
#include "array_ROM.h"

dout_t array_ROM(din1_t inval, din2_t idx)
{
    din1_t lookup_table[256];
    dint_t i;

    for (i = 0; i < 256; i++) {
        lookup_table[i] = 256 * (i - 128);
    }

    return (dout_t)inval * (dout_t)lookup_table[idx];
}
```

例 4-29 : static または const を使用しない ROM インプリメンテーション

例 4-29 の場合、Vivado HLS で `lookup_table` 変数が最終 RTL でメモリエレメントになるのが最適なインプリメンテーションであると判断できます。配列の場合にこれをどのように達成すればよいかについては、「ROM のインプリメンテーション」セクションを参照してください。

グローバル変数

グローバル変数はコード内で自由に使用でき、完全に合成可能ですが、デフォルトではグローバル変数は RTL インターフェイスのポートとしては公開されません。グローバル変数がどのように合成されるかについては、例 4-30 を参照してください。

例 4-30 では、グローバル変数が 3 つ使用されています。この例では配列が使用されますが、すべてのタイプのグローバル変数がサポートされています。

- 値は配列 `Ain` から読み出されます。
- 配列 `Aint` は `Ain` からの値を変換して `Aout` に渡すために使用されます。
- 出力は配列 `Aout` に書き込まれます。

```
din_t Ain[N];
din_t Aint[N];
dout_t Aout[N/2];

void types_global(din1_t idx) {
    din_t i, lidx;

    // Move elements in the input array
    for (i=0; i<N; ++i) {
        lidx=i;
        if (lidx+idx>N-1)
            lidx=N-1;
        Aint[lidx] = Ain[lidx+idx] + Ain[lidx];
    }

    // Sum to half the elements
    for (i=0; i<(N/2); i++) {
        Aout[i] = (Aint[i] + Aint[i+1])/2;
    }
}
```

例 4-30 : グローバル変数

デフォルトでは、合成後、RTL デザインのポートは `idx` だけになります。グローバル変数はデフォルトでは RTL ポートとしては公開されません。デフォルトの場合、配列 `Ain` は読み出し元の内部 RAM、配列 `Aout` は書き込み先の内部 RAM です。

Vivado HLS インターフェイスコンフィギュレーションの `expose_global` オプションを使用すると、グローバル変数を RTL インターフェイスのポートとして公開できます。この場合、外部 RAM として `Ain` および `Aout` の両方にアクセスするためのポートが作成されます。また、ポートは内部 RAM へのアクセスを示すために作成されることもあります。

注記：グローバル変数が公開されると、デザインに対して内部アクセスしかないものも含め、デザイン内のすべてのグローバル変数が RTL ポートとして公開されます。

グローバル変数は合成でサポートされますが、グローバル変数を広範囲に使用するコードはお勧めしません。

ポインター

ポインターは C コードで広範囲に使用され、合成でもサポートされます。ポインターを使用する際は、次に注意してください。

- ポインターが同じ関数内で複数回アクセス（読み出しちゃは書き込み）される場合（これに関する問題については、「マルチアクセス ポインター インターフェイス：ストリーミング データ」を参照してください）。
- ポインターの配列を使用する場合、各ポインターが別のポインターではなく、スカラーまたはスカラー配列を指定する必要あり
- ポインターの型変換は標準 C 型間の変換の場合にのみサポートあり

前述の例の多くは、Vivado HLS を使用して C ポインターが合成できることを示していました。ポインターのサポートされる合成には、ポインターが複数のオブジェクトを指定する場合も含まれます（[例 4-31](#)）。

```
#include "pointer_multi.h"

dout_t pointer_multi (sel_t sel, din_t pos) {
    static const dout_t a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    static const dout_t b[8] = {8, 7, 6, 5, 4, 3, 2, 1};

    dout_t* ptr;
    if (sel)
        ptr = a;
    else
        ptr = b;

    return ptr[pos];
}
```

例 4-31: 複数のポインター ターゲット

ダブルポインターも合成でサポートされます（[例 4-32](#)）。ダブルポインターが複数の関数で使用されると、Vivado HLS ではすべての関数をそれが使用される箇所でインライン化します。複数の関数がインライン化されると、ランタイムが増加する可能性があります。

```
#include "pointer_double.h"

data_t sub(data_t ptr[10], data_t size, data_t**flagPtr)
{
    data_t x, i;

    x = 0;
    // Sum x if AND of local index and double-pointer index is true
    for(i=0; i<size; ++i)
```

```

        if (**flagPtr & i)
            x += *(ptr+i);
    return x;
}

data_t pointer_double(data_t pos, data_t x, data_t* flag)
{
    data_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    data_t* ptrFlag;
    data_t i;

    ptrFlag = flag;

    // Write x into index position pos
    if (pos >= 0 & pos < 10)
        *(array+pos) = x;

    // Pass same index (as pos) as pointer to another function
    return sub(array, 10, &ptrFlag);
}

```

例 4-32: ダブルポインター

ポインターの配列も合成できます。例 4-33 では、ポインターの配列がグローバル配列の 2 次元の開始位置を格納するために使用されています。

注記: ポインターの配列内のポインターは、スカラーまたはスカラーの配列のみを指定でき、ほかのポインターを指定することはできません。

```

#include "pointer_array.h"

data_t A[N][10];

data_t pointer_array(data_t B[N*10]) {
    data_t i,j;
    data_t sum1;

    // Array of pointers
    data_t* PtrA[N];

    // Store global array locations in temp pointer array
    for (i=0; i<N; ++i)
        PtrA[i] = &(A[i][0]);

    // Copy input array using pointers
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            *(PtrA[i]+j) = B[i*10 + j];

    // Sum input array
    sum1 = 0;
    for(i=0; i<N; ++i)
        for(j=0; j<10; ++j)
            sum1 += *(PtrA[i] + j);

    return sum1;
}

```

例 4-33: ポインター配列

ポインターの型変換は、ネイティブ C 型が使用される場合に合成でサポートされます。例 4-34 では、`data_t` (`char`) 型が、型に変換されています。

```
#define N 1024

typedef int data_t;
typedef char dint_t;

data_t pointer_cast_native (data_t index, data_t A[N]) {
    dint_t* ptr;
    data_t i = 0, result = 0;
    ptr = (dint_t*)(&A[index]);

    // Sum from the indexed value as a different type
    for (i = 0; i < 4*(N/10); ++i) {
        result += *ptr;
        ptr+=1;
    }
    return result;
}
```

例 4-34: ネイティブ型を使用したポインター型変換

ただし、一般型間のポインター型変換はサポートされません。たとえば、符号付きの値の (struct) 複合型が作成されると、ポインターを型変換して符号なしの値を代入することはできません。

```
struct {
    short first;
    short second;
} pair;

// Not supported for synthesis
*(unsigned*)pair = -1U;
```

このような場合、値はネイティブ型を使用して代入する必要があります。

```
struct {
    short first;
    short second;
} pair;

// Assigned value
pair.first = -1U;
pair.second = -1U;
```

C ライブライ

Vivado HLS には、C で簡単に記述して RTL に合成できるよう、共通のハードウェア デザイン構文を含む C ライブライアリが多くの含まれています。このセクションでは次の C ライブライアリについて説明します。

- `hls_math.h` ライブライアリ
- `hls_video.h` ライブライアリ

`hls_math.h` ライブラリには、標準 C/C++ `math.h` ライブラリで最もよく使用される関数の合成可能なバージョンが含まれています。

`hls_video.h` ライブラリには、ビデオ アルゴリズムが C でさらに簡単に取り込まれるようにするビデオ データ型とクラスが含まれます。

hls_math ライブラリ

Vivado HLS ライブラリの `hls_math.h` は、標準 C および C++ ライブラリの浮動小数点関数 (`math.h` および `cmath.h`) の合成をサポートするために提供されています。

表 2-2 は、合成される `math.h` および `cmath.h` の関数をリストしています。これらの関数の中には、浮動小数点 LogicCORE を使用してインプリメントされるものもあります。それ以外は、`hls_math.h` ライブラリを使用し、ビット概算インプリメンテーションとしてインプリメントされます。

- ビット概算インプリメンテーションには、標準動作と同じ精度はありません。精度はほとんどの動作範囲で通常 1 ULP (Unit in the Last Place) 以内になりますが、100 ULP までになる可能性もあります。
- ビット概算インプリメンテーションでは、C/C++ ソフトウェア バージョンとは異なる下位アルゴリズムを使用して結果が達成されることがあります。

表 2-2 にリストされていない `math.h` または `cmath.h` ライブラリの関数は合成できません。Vivado HLS はエラー メッセージを表示して停止します。

`hls_math.h` ライブラリは、合成が実行されると Vivado HLS で自動的に呼び出されます。オプションでソースに含めることもできます。`hls_math.h` ライブラリを含めた場合と含めない場合の違いは、C シミュレーション結果に表れます。これについては、後で説明します。

- 表 2-2 の浮動小数点 C 関数の場合、その関数の倍精度バージョンはありません。この場合、そのサポートは表 2-2 で「該当なし」と記載されています。
- 精度 (ULP) の列には、演算子入力値の範囲全体での最小から最大の精度の差異がリストされています。

表 2-2 : `math.h` - ビット概算のサポートされる関数

関数	浮動小数点	倍精度	精度 (ULP)	LogicCORE
<code>ceilf</code>	サポートあり	該当なし	正確	サポートなし
<code>copysignf</code>	サポートあり	該当なし	正確	サポートなし
<code>fabsf</code>	サポートあり	該当なし	正確	サポートなし
<code>floorf</code>	サポートあり	該当なし	正確	サポートなし
<code>logf</code>	サポートあり	該当なし	1 ~ 5	サポートなし
<code>cosf</code>	サポートあり	該当なし	1 ~ 100	サポートなし
<code>sinf</code>	サポートあり	該当なし	1 ~ 100	サポートなし
<code>abs</code>	サポートあり	サポートあり	正確	サポートなし
<code>ceil</code>	サポートあり	サポートあり	正確	サポートなし
<code>copysign</code>	サポートあり	サポートあり	正確	サポートなし
<code>cos</code>	サポートあり	サポートあり	浮動小数点の場合は 2、倍精度の場合は 5	サポートなし
<code>fabs</code>	サポートあり	サポートあり	正確	サポートなし
<code>floor</code>	サポートあり	サポートあり	正確	サポートなし

表 2-2 : math.h - ビット概算のサポートされる関数

関数	浮動小数点	倍精度	精度 (ULP)	LogicCORE
fpclassify	サポートあり	サポートあり	正確	サポートなし
isfinite	サポートあり	サポートあり	正確	サポートなし
isinf	サポートあり	サポートあり	正確	サポートなし
isnan	サポートあり	サポートあり	正確	サポートなし
isnormal	サポートあり	サポートあり	正確	サポートなし
log	サポートあり	サポートあり	浮動小数点の場合は 1、倍精度の場合は 16	サポートなし
log10	サポートあり	サポートあり	浮動小数点の場合は 1、倍精度の場合は 16	サポートなし
recip	サポートあり	サポートあり	正確	サポートあり
round	サポートあり	サポートあり	正確	サポートなし
rsqrt	サポートあり	サポートあり	正確	サポートあり
signbit	サポートあり	サポートあり	正確	サポートなし
sin	サポートあり	サポートあり	浮動小数点の場合は 2、倍精度の場合は 5	サポートなし
sqrt	サポートあり	サポートあり	正確	サポートあり
trunc	サポートあり	サポートあり	正確	サポートなし

表 2-2 の次の 7 つの関数には、注意が必要です。

- isinf
- isnan
- copysign
- fpclassify
- isfinite
- isnormal
- signbit

上記の関数をコンパイルするために使用される C 標準によって、結果は次のように異なります。

C90 モード : isinf, isnan、および copysign だけは、通常システム ヘッダー ファイルで提供され、倍精度で動作します。特に copysign は常に倍精度の結果を戻します。これにより、浮動小数点を戻さなければいけない場合、double-to-float 変換ブロックはハードウェアにしか導入できないので、合成後で予想外の結果になってしまうことがあります。

C99 モード (-std=c99) : 7 つの関数はすべて、システム ヘッダー ファイルにより __isnan(double) および __isnan(float) にリダイレクトされるという予測で提供されています。通常の GCC ファイルは isnormal をリダイレクトはしませんが、fpclassify を使用してインプリメントします。

math.h を使用した C++ : 7 つの関数はすべて、通常システム ヘッダー ファイルで提供され、倍精度で動作します。特に copysign は常に倍精度の結果を戻します。これにより、浮動小数点を戻さなければいけない場合、double-to-float 変換ブロックはハードウェアにしか導入できないので、合成後で予想外の結果になってしまうことがあります。

cmath を使用した C++: C99 モード (-std=c99) と同様ですが、システムヘッダーファイルは通常異なり、関数は float() 用に適切にオーバーロードされます。isnan(double)、isinf(double)、copysign、copysignf は「using namespace std;」の場合でもビルトインとして処理されます。

cmath および namespace std を使用した C++: 問題なし

注記: GCC は通常 isinf および copysign をビルトインとしてインプリメントし、isnan を浮動小数点比較に削減します。 デフォルトで -fno-builtin を使用して、この動作にならないようにし、最適な動作にする必要があります。

C の場合は -std=c99 を、C および C++ の場合は -fno-builtin を使用することをお勧めします。

シミュレーションの違い

シミュレーション結果は C シミュレーションにどの C ライブライが使用されたかによって異なり、C と RTL シミュレーションの比較にはビット概算インプリメンテーションが使用されます。表 2-3 は、シミュレーション結果の違いを示しています。

つまり次のようにになります。

- math.h または cmath.h ライブライが使用されると、C と RTL 結果は異なります。この違いの例は、下記を参照してください。
- hls_math.h ライブライが使用されると、これらの結果と math.h または cmath.h を使用して取得される結果は異なります。
- math.h または cmath.h ライブライが使用され、lib_hls.cpp ファイルが含まれると、結果は hls_math.h が使用された場合と同じになります。

表 2-3: シミュレーション結果の違い

	math.h または cmath.h	hls_math.h	math.h または cmath.h (lib_hls.cpp 含有)
C 検証	結果 A	結果 C	結果 C
RTL シミュレーション	結果 B	結果 C	結果 C

次の math.h 関数の C++ 合成例では、sinf、cosf、および sqrtf を使用した C++ デザインを示しており、Vivado HLS でのデザインの合成方法とその合成前と合成後の結果がどれくらい違うかを示しています。

```
#include "cpp_math.h"

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

ヘッダーファイル cpp_math.h は math.h 関数のヘッダーファイルの C++ 合成の例で、cmath ライブライを呼び出し (math.h は C で使用)、標準の名前空間 (namespace) を使用して、データ型 data_t を float 型として定義しています。

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;
```

```
data_t cpp_math(data_t angle);
```

math.h 関数のテストベンチの C++ 合成で示されるテストベンチ例では、結果が合成前シミュレーションで検証されますが、エラーは合成後の RTL シミュレーションで戻されます。これについては、次に説明します。

```
#include "cpp_math.h"

int main() {
    ofstream result;
    data_t angle = 0.01;
    data_t output;
    int retval=0;

    result.open("result.dat");
    // Persistent manipulators
    result << right << fixed << setbase(10) << setprecision(15);

    for (data_t i = 0; i <= 250; i++)
    {
        output = cpp_math(angle);

        result << setw(10) << i;
        result << setw(20) << angle;
        result << setw(20) << output;
        result << endl;

        angle = angle + .1;
    }
    result.close();

    // Compare the results file with the golden results
    retval = system("diff --brief -w result.dat result.golden.dat");
    if (retval != 0) {
        printf("Test failed !!!\n");
        retval=1;
    } else {
        printf("Test passed !\n");
    }

    // Return 0 if the test passes
    return retval;
}
```

テストベンチは出力結果を比較し、result.dat ファイルに保存します。予測データ値は result.golden.dat に含まれます。合成後、math.h 関数の RTL インプリメンテーションはその関数のビット概算バージョンなので、RTL シミュレーションで出力される results.dat ファイルは予測される結果とは異なる可能性があります。

Vivado HLS は RTL シミュレーションをプロジェクトの下位ディレクトリである <SOLUTION>/sim/<HDL> で実行します。この場合、SOLUTION はソリューションの名前、HDL は RTL シミュレーション用に選択された HDL のタイプになります。たとえば、プロジェクトを次のように設定したとします。

- プロジェクト名は proj_cpp_math.prj
- ソリューションは solution1
- RTL は systemc を使用してシミュレーション

RTL シミュレーションの出力は proj_cpp_math.prj/solution1/sim/ systemc/results.dat ファイルに保存

図 4-3 は、合成前の result.dat ファイルと合成後の RTL の result.dat ファイルの比較を示しています。出力値は 3 列目に表示されています。

	result.dat			proj_cpp_math.prj/solution1/sim/systemc/result.dat		
1	0.0000000000000000	0.009999999776483	1.0000000000000000	1	0.0000000000000000	0.009999999776483
2	1.0000000000000000	0.10999999403954	1.0000000000000000	2	1.0000000000000000	0.10999999403954
3	2.0000000000000000	0.209999993443489	1.0000000000000000	3	2.0000000000000000	0.209999993443489
4	3.0000000000000000	0.310000002384186	1.0000000000000000	4	3.0000000000000000	0.310000002384186
5	4.0000000000000000	0.409999996423721	1.0000000000000000	5	4.0000000000000000	0.409999996423721
6	5.0000000000000000	0.509999990463257	1.0000000000000000	6	5.0000000000000000	0.509999990463257
7	6.0000000000000000	0.610000014305115	0.99999940395355	7	6.0000000000000000	0.610000014305115
8	7.0000000000000000	0.710000038146973	1.0000000000000000	8	7.0000000000000000	0.710000038146973
9	8.0000000000000000	0.810000061988831	1.0000000000000000	9	8.0000000000000000	0.810000061988831
10	9.0000000000000000	0.910000085830688	1.0000000000000000	10	9.0000000000000000	0.910000085830688
11	10.0000000000000000	1.010000109672546	1.0000000000000000	11	10.0000000000000000	1.010000109672546
12	11.0000000000000000	1.110000133514404	1.0000000000000000	12	11.0000000000000000	1.110000133514404
13	12.0000000000000000	1.210000157356262	0.99999940395355	13	12.0000000000000000	1.210000157356262
14	13.0000000000000000	1.310000181198120	0.99999940395355	14	13.0000000000000000	1.310000181198120
15	14.0000000000000000	1.410000205039978	1.0000000000000000	15	14.0000000000000000	1.410000205039978
16	15.0000000000000000	1.510000228881836	1.0000000000000000	16	15.0000000000000000	1.510000228881836
17	16.0000000000000000	1.610000252723694	1.0000000000000000	17	16.0000000000000000	1.610000252723694
18	17.0000000000000000	1.710000276565552	1.0000000000000000	18	17.0000000000000000	1.710000276565552
19	18.0000000000000000	1.810000300407410	1.0000000000000000	19	18.0000000000000000	1.810000300407410
20	19.0000000000000000	1.910000324249268	0.99999940395355	20	19.0000000000000000	1.910000324249268
21	20.0000000000000000	2.010000228881836	0.99999940395355	21	20.0000000000000000	2.010000228881836
22	21.0000000000000000	2.110000133514404	1.0000000000000000	22	21.0000000000000000	2.110000133514404
23	22.0000000000000000	2.21000038146973	1.0000000000000000	23	22.0000000000000000	2.21000038146973
24	23.0000000000000000	2.30999942779541	1.0000000000000000	24	23.0000000000000000	2.30999942779541
25	24.0000000000000000	2.409999847412109	1.0000000000000000	25	24.0000000000000000	2.409999847412109
26	25.0000000000000000	2.50999752044678	1.0000000000000000	26	25.0000000000000000	2.50999752044678
27	26.0000000000000000	2.60999656677246	1.0000000000000000	27	26.0000000000000000	2.60999656677246
28	27.0000000000000000	2.70999561309814	0.99999940395355	28	27.0000000000000000	2.70999561309814
29	28.0000000000000000	2.80999465942383	1.0000000000000000	29	28.0000000000000000	2.80999465942383
30	29.0000000000000000	2.90999370574951	1.0000000000000000	30	29.0000000000000000	2.90999370574951

図 4-3 : 合成前後のシミュレーションの違い

このこのアルゴリズムとテストベンチでは、合成前の合成後のシミュレーション結果は小数点分異なります。疑問点は、これらの小数点部分が最終 RTL インプリメンテーションで使用可能かどうかです。

これらの違いを処理するのに推奨されるフローは、結果をチェックしてエラーの許容範囲内に収まるかどうかを確認するスマート テストベンチを使用するフローです。

- math.h およびcmath.h ライブライを (この例の場合のように) 使用する場合は、精度の違いが許容範囲かどうかを確認します。
- または、hls_math.h を使用するように関数を変換し、これらの結果と math.h または cmath.h を使用した場合の結果との相違点を確認します。

注記 : 浮動小数点および倍精度の演算および関数を使用する場合は、範囲を使用して結果を確認することをお勧めします。絶対比較を使用するとエラーになることがあります。同様の (まったく同じではない) 方法で計算した 2 セットの結果を C コードで比較した場合ですら、エラーになることがあります。

よくある合成エラー

次は math 関数を合成する際によく発生する使用エラーです。これらは math 関数の合成の利点を生かすために、C 関数を C++ 関数へ変換した場合によく発生します。

C++ の cmath.h ヘッダーファイルが使用されると、浮動小数点関数 (sinf, cosf など) が使用できます。これらはハードウェアで 32 ビット演算になります。cmath.h ヘッダーファイルは標準関数 (sin, cos など) もオーバーロードするので、float および double 型にも使用できます。

C の `math.h` ライブラリが使用される場合は、32 ビットの浮動小数点演算を合成するために、浮動小数点関数 (`sinf`、`cosf` など) が必要です。すべての標準関数呼び出し (`sin`、`cos` など) は合成されると倍精度および 64 ビットの倍精度演算になります。

注記 : `math.h` サポートの利点を生かすために C 関数を C++ に変換する場合は、Vivado HLS で合成する前に新しい C++ コードが正しくコンパイルされるようにする必要があります。

たとえば、`sqrtf()` が `math.h` と一緒にコードで使用される場合は、それをサポートするために "C" `float sqrtf(float);` を C++ コードに追加する必要があります。

また、「`float` および `double` 型」セクションに示したように、`double` および `float` 型の混合に関する警告メッセージに従って、型変換によって不必要的ハードウェアが作成されないようにします。

HLS ビデオ ライブラリ

Vivado HLS ビデオ ライブラリには、`hls_video.h` ヘッダー ファイルを使用する必要があります。このヘッダー ファイルには、Vivado HLS で提供されるすべてのイメージおよびビデオ プロセス専用のビデオ型および関数が含まれます。

Vivado HLS ビデオ ライブラリを使用する場合、デザインが C++ で記述されて `hls namespace` が使用されているか、型およびクラスがスコープ付きの名前を使用している必要があります。

```
#include <hls_video.h>

hls::rgb_8 video_data[1920][1080]

または

#include <hls_video.h>
using namespace hls;

rgb_8 video_data[1920][1080]
```

データ型

次のデータ型がライブラリに含まれます。すべてのデータ型では、現在のところ 8 ビット データのみがサポートされます。

表 2-4: ビデオ データ型

データ型名	フィールド 0(8 ビット)	フィールド 1(8 ビット)	フィールド 2(8 ビット)	フィールド 3(8 ビット)
yuv422_8	Y	UV	使用なし	使用なし
yuv444_8	Y	U	V	使用なし
rgb_8	G	B	R	使用なし
yuva422_8	Y	UV	a	使用なし
yuva444_8	Y	U	V	a
rgba_8	G	B	R	a
yuva420_8	Y	aUV	使用なし	使用なし
yuvd422_8	U	UV	D	使用なし
yuvd444_8	Y	U	V	D
rgbd_8	G	B	R	D

表 2-4: ビデオ データ型

データ型名	フィールド 0(8 ビット)	フィールド 1(8 ビット)	フィールド 2(8 ビット)	フィールド 3(8 ビット)
bayer_8	RGB	使用なし	使用なし	使用なし
luma_8	Y	使用なし	使用なし	使用なし

hls_video.h ライブラリが含まれ、hls namespace が定義されると、[例 4-5](#) にリストされるデータ型を自由に使用できるようになります。

```
#include <hls_video.h>
using namespace hls;

rgb_8 video_data[1920][1080]
```

メモリ ライン バッファー

linebuffer クラスは、ユーザーのアルゴリズム コード内でライン バッファーを簡単に宣言および管理可能にする C++ クラスです。このクラスには、ライン バッファーをインスタンシエートおよび操作するために必要なメソッドがすべて含まれます。linebuffer クラスは、すべてのデータ型で使用できます。

linebuffer クラスの主な機能は、次のとおりです。

- パラメーター指定によるすべてのデータ型のサポート
- ユーザー定義の行数および列数
- メモリのバンド幅を増加するために行を別々のメモリ バンクへ自動的にバンキング
- アルゴリズム デザインのライン バッファーの使用およびデバッグをするメソッドすべてを提供

linebuffer クラスには、次のメソッドが含まれます。

- print()
- shift_up();
- shift_down()
- insert_bottom()
- insert_top()
- getval(row,column)

linebuffer の使用方法を説明するため、すべての例の最初に次のデータ セットが使用されます。

表 2-5: linebuffer 例のデータ セット

	列 0	列 1	列 2	列 3	列 4
行 2	1	2	3	4	5
行 1	6	7	8	9	10
行 0	11	12	13	14	15

ライバッファーの宣言

ライバッファーは、次のデータ型を使用してアルゴリズムにインスタンシエートできます。

```
hls::linebuffer<type, rows, columns>
```

表 2-5 のデータを保持するライン バッファーは、次のように宣言できます。

```
hls::linebuffer<char, 3, 5> Buff_A;
```

ライン バッファーの内容の表示

linebuffer クラスには、ライン バッファーに格納されるデータを表示するために `print` メソッドが含まれます。ライン バッファーには多数の列を含めることができるので、`print` メソッドを使用すると、開始列値から終了列値までのすべての行が表示されます。

次に例を示します。

```
Buff_A.print(1, 3);
```

これは次のようにになります。

```
行 2 : 234
行 1 : 789
行 0 : 121314
```

ライン バッファーでのデータの挿入とシフト

linebuffer クラスでは、ライン バッファーをインスタンシエートするブロックに入力されるデータは、ラスター走査順に並び替えられるので、新しいデータがそれ以前のデータとは別の列に格納されます。

新しい値を挿入する場合は、前の有限数値は列に保存された状態のままで、その列の行間の垂直シフトが必要になります。シフトが終了したら、新しいデータ値は列の最上部または最下部のいずれかに挿入できます。

たとえば、値 100 をライン バッファーの列 2 の最上部に挿入するには、次を使用します。

```
Buff_A.shift_down(2);
Buff_A.insert_top(100, 2);
```

これにより、表 2-6 に示す新しいデータ セットになります。

表 2-6 : `shift_down` および `insert_top` を使用した後のデータ セット

	列 0	列 1	列 2	列 3	列 4
行 2	1	2	100	4	5
行 1	6	7	3	9	10
行 0	11	12	8	14	15

同様に、値 100 をライン バッファー セット (表 2-7) の列 2 の最下部に挿入するには、次を使用します。

```
Buff_A.shift_up(2);
Buff_A.insert_bottom(100, 2);
```

これにより、表 2-7 に示す新しいデータ セットになります。

表 2-7 : `shift_up` および `insert_bottom` を使用した後のデータ セット

	列 0	列 1	列 2	列 3	列 4
行 2	1	2	100	4	5
行 1	6	7	3	9	10
行 0	11	12	8	14	15

`shift` メソッドと `insert` メソッドのどちらも処理する列の値を必要とします。

データの取得

linebuffer インスタンスで格納された値はすべて `getval(row, column)` メソッドで取り出すことができます。このメソッドでは、ラインバッファー内のどの位置の値でも戻すことができます。次に例を示します。

```
Value = Buff_A.getval(1, 3);
```

これにより、変数 `Value` に値 9 が代入されます。

メモリ ウィンドウ

ウィンドウ C++ クラスを使用すると、2 次元のメモリ ウィンドウを宣言および管理できます。このクラスの主な機能は、次のとおりです。

- パラメーター指定によるすべてのデータ型のサポート
- ユーザー定義の行数および列数
- 最大バンド幅のため、個別レジスターへの自動分割
- メモリ ウィンドウを使用およびデバッグするメソッドすべてを提供

メモリ ウィンドウ クラスは、次のメソッドでサポートされます。

- `print()`
- `shift_up()`
- `shift_down()`
- `shift_left()`
- `shift_right()`
- `insert(value, row, column)`
- `insert_bottom()`
- `insert_top()`
- `insert_left()`
- `insert_right()`
- `getval(row, column)`

メモリ ウィンドウの使用方法を説明するため、すべての例の最初に次のデータ セットが使用されます。

表 2-8: メモリ ウィンドウ例のデータ セット

	列 0	列 1	列 2
行 2	1	2	3
行 1	6	7	8
行 0	11	12	13

メモリ ウィンドウの宣言

メモリ ウィンドウは、次のデータ型を使用してアルゴリズムにインスタンシエートできます。

```
hls::window<type, rows, columns>
```

データを保持するメモリ ウィンドウは次のように宣言できます。

```
hls::window<char, 3, 3> Buff_B;
```

メモリ ウィンドウの内容の表示

window クラスには、メモリ ウィンドウの内容を表示するために、print メソッドが含まれます。デフォルトでは、このメソッドはウィンドウ内に格納されるすべての値を表示します。次に例を示します。

```
Buff_B.print();
```

これは次のようにになります。

```
Window Size3x3
Col012
Row 2123
Row 1678
Row 0111213
```

すべての window クラス インスタンシエーションで row 0 はメモリ ウィンドウの最下部になると仮定されます。

メモリ ウィンドウでのデータのシフト

window クラスには、メモリ ウィンドウ内に格納されたデータを上下左右に移動するメソッドが含まれます。これらのシフト操作により、新しいデータ用にメモリ ウィンドウ内のスペースを空けることができます。

表 4-9 のデータ セットから開始するとして、次を実行したとします。

```
Buff_B.shift_up();
Buff_B.print();
```

これは、次の結果になります。

```
Window Size3x3
Col0 1 2
Row 267 8
Row 1111213
Row 0New dataNew dataNew data
```

同様に、表 2-8 のデータ セットから開始するとして、次を実行したとします。

```
Buff_B.shift_down();
Buff_B.print();
```

これは、次の結果になります。

```
Window Size3x3
Col 0 1 2
Row 2New dataNew dataNew data
Row 11 2 3
Row 06 7 8
```

また、次を実行したとします。

```
Buff_B.shift_left();
Buff_B.print();
```

データが左にシフトされて、次のようにになります。

```
Window Size3x3
Col012
Row 223New data
Row 178New data
Row 01213New data
```

最後に次を実行したとします。

```
Buff_B.shift_right();
Buff_B.print();
```

これは、次の結果になります。

```
Window Size3x3
Col0 1 2
Row 2New data12
Row 1New data67
Row 0New data1112
```

メモリ ウィンドウのデータの挿入および取得

window クラスを使用すると、メモリ ウィンドウ内にデータを挿入したり、データを取得したりできます。メモリ ウィンドウの境界上にデータのブロック挿入をすることもできます。

`insert(value, row, column)` を使用すると、データはメモリ ウィンドウのどの位置にでも挿入できます。たとえば、値 100 は次を実行するとメモリ ウィンドウの行 1 列 1 に挿入できます。

```
Buff_B.insert(100,1,1);
Buff_B.print();
```

これは `print` メソッドで表示すると、次のようにになります。

```
Window Size3x3
Col012
Row 2123
Row 161008
Row 0111213
```

ブロック レベルの挿入には、バウンダリ上に挿入するデータ エレメントの配列を指定する必要があります。window クラスで提供されるメソッドは、次のとおりです。

- `insert_bottom`
- `insert_top`
- `insert_left`
- `insert_right`

たとえば、C が 3 エレメントの配列で、各配列の値は 50 で、メモリ ウィンドウの最下部バウンダリに値 50 を挿入する場合は、次のように指定します。

```
char C[3] = {50, 50, 50};
Buff_B.insert_bottom(C);
Buff_B.print();
```

これは次のようにになります。

```
Window Size3x3
Col012
Row 2123
Row 161008
Row 0505050
```

window クラスのその他のエッジ挿入メソッドは、`insert_bottom()` メソッドと同じように使用できます。

メモリ ウィンドウからデータを取得するには、`getval(row, column)` を使用します。次に例を示します。

```
A = Buff_B.getval(0,1);
```

これは次のようにになります。

A = 50

ハードウェアのコード記述方法

Cコードは、再利用、読みやすさ、パフォーマンスなどの多数の要件を満たすように記述されます。ここまでのことろ、Cコードは高位合成した後に最適なハードウェアが得られるように記述されていませんでした。

ただし、再利用、読みやすさ、パフォーマンス要件と同様に、特定のコード記述方法またはあらかじめ定義されたコンストラクトを使用することにより、合成結果がより適切なハードウェアになるように、またはアルゴリズムをより簡単に検証してCでハードウェアをより適切に記述できるようになります。

C++でのユーザー定義レジスタ

通常、C関数で確実にレジスタとして実装される変数は、`static`修飾子が付いたもの、グローバルに定義されているもの（グローバルがポートとなる場合を除く）、およびメモリリソースをターゲットとした配列です。

その他の変数は、合成中の決定によって、レジスタに実装される場合とそうでない場合があります。合成で、変数が複数サイクル間レジスタに保持される必要があると判断される場合と、作成されたサイクルと同じサイクルで使用できるのでレジスタは不要であると判断される場合があります。上記の変数のみが、RTLで確実にレジスタに実装されます。

次の例に示すように、`Reg`というC++関数を使用すると、特定の変数を最終的なRTLデザインで確実にレジスタとして実装されます。

```
#include "foo.h"

template<class T>
T Reg(T in) {
#pragma AP INLINE off
#pragma AP INTERFACE port=return register
    return in;
}

int foo (int in1, int in2 ) {
    int tmp=in1*(0x0800-in2);
    // int out = tmp >> 10;
    int out = Reg(tmp >> 10);
    return( out );
}

int foo_top(int inA, int inB) {

    int res1 = foo(inA, inB);
    return(res1);
}
```

このコードの重要な点は、次のとおりです。

- 関数`Reg`が作成され、出力の関数`return`の値がレジスタに格納されます。
- サブ関数では、インターフェイス指示子は関数の引数（RTLポート）をレジスタに格納するためのみに使用できます。インターフェイス指示子は、サブ関数の引数のI/Oプロトコルを指定するために使用することはできません。

- 関数 `Reg` では、Vivado HLS でこの関数が自動的にインライン化されないように、インライン化がディスエーブルになっています。
 - 関数 `Reg` をインライン化すると、個別の関数でなくなるため、関数で実行されるレジスタ操作が無効になります。
- サブ関数 `foo` の乗算とシフト演算の出力が、関数 `Reg` の入力として使用されます。
 - 関数 `Reg` の出力にはレジスタが付いているので、これによりこの演算の結果が RTL 出力のレジスタに格納されます。
- この変数の元のソース コードはコメントアウトされているので、この手法を適用するのが簡単です。

通常、Vivado HLS により最終 RTL でレジスタにインプリメントされる変数が決定されますが、このコード記述方法を使用すると、コードの機能を変えずに特定の変数をレジスタにインプリメントできます。

SRL リソースへの直接マップ[†]

多くの C アルゴリズムでは、データを配列内で順次シフトします。配列の開始に新しい値を追加し、既存のデータを配列内でシフトし、最も古いデータ値は破棄されます。この操作は、ハードウェアではシフトレジスタとしてインプリメントされます。

C からシフトレジスタをインプリメントする場合、配列を個々のエレメントに完全に分割し、RTL 内のエレメント間のデータ依存性によりシフトレジスタを示すのが最も一般的です。

論理合成では、RTL シフトレジスタがシフトレジスタを効率的にインプリメントするザイリンクス SRL リソースにインプリメントされます。問題は、論理合成で RTL シフトレジスタが SRL コンポーネントを使用してインプリメントされない場合があることです。

- シフトレジスタの中央にあるデータにアクセスする場合、論理合成では SRL を直接推論できません。
- SRL が最適であっても、ほかの要因により論理合成でシフトレジスタがフリップフロップにインプリメントされることがあります。論理合成は、複雑なプロセスです。

Vivado HLS では、C++ クラス `ap_shift_reg` が提供されており、C コードで定義されたシフトレジスタが SRL リソースを使用してインプリメントされるようにできます。`ap_shift_reg` クラスでは、SRL コンポーネントでサポートされるさまざまな読み出しおよび書き込みアクセスを実行する方法が 2 つあります。

シフトレジスタからの読み出し

読み出しへは、シフトレジスタの指定の位置から読み出すことができます。

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 2 of Sreg into var1
var1 = Sreg.read(2);
```

データの読み出しおよび書き込みして、シフト

シフトでは、読み出し、書き込みおよびシフト操作を実行できます。

```
// Include the Class
```

```

#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 3 of Sreg into var1
// THEN shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3);

```

読み出しおよび書き込みして、シフトをイネーブル制御

shift メソッドではイネーブル入力もサポートされており、シフト操作を変数でイネーブル制御できます。

```

// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;

// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3,En);

```

ap_shift_reg クラスを使用すると、Vivado HLS で各シフト レジスタに対して固有の RTL コンポーネントが作成されます。論理合成が実行されると、このコンポーネントが SRL リソースに合成されます。

ストリーミング データを使用した設計

ストリーミング データは、データ サンプルが最初のサンプルからシーケンシャル順に送信されるタイプのデータ転送です。ストリーミングには、アドレス管理が必要とされません。

ストリーミング データを使用するデザインは C で記述するのが困難な場合があります。「マルチアクセス ポインター インターフェイス：ストリーミング データ」セクションの説明のとおり、複数の読み出しおよび書き込みを実行するためにポインターを使用すると、型修飾子とテストベンチの構築方法が記述されるので、問題が発生する可能性があります。

Vivado HLS には、ストリーミング データ構造を記述するための C++ テンプレート クラスの `hls::stream<>` が含まれます。`hls::stream<>` クラスを使用してインプリメントしたストリームには、次の属性があります。

このセクションでは、ストリーミング データを使用したデザインを `hls::stream<>` クラスを使用してより簡単に記述する方法を示します。このセクションのトピックには、次が含まれます。

- ストリームを使用した記述とストリームの RTL インプリメンテーションの概要
- グローバル ストリーム変数の規則
- ストリームの使用方法

- 読み出しおよび書き込みのブロッキング
- 読み出しおよび書き込みのノンブロッキング
- FIFO の深さの制御

C 記述と RTL インプリメンテーション

ストリームは、ソフトウェア（および RTL 協調シミュレーション中のテストベンチ）では無限大のキューとして記述されます。ストリームをシミュレーションするために C で深さを指定する必要はありません。ストリームは、関数内および関数へのインターフェイス上で使用できます。内部ストリームは関数パラメーターとして渡すことができます。

注記：ストリームは C++ ベースのデザインでのみ使用できます。各 `hls::stream<>` オブジェクトは 1 プロセスで書き込まれ、1 プロセスで読み出されるようにする必要があります。たとえば、DATAFLOW デザインでは、各ストリームグリには 1 つの送信プロセスと 1 つの受信プロセスだけが含まれます。

RTL では、ストリームは FIFO またはフルハンドシェイク インターフェイス ポートのいずれかとしてインプリメントされます。デフォルトのインターフェイス ポートは `ap_fifo` ポートです。このポートはオプションで最上位インターフェイスの `ap_hs` ポートとしてインプリメントすることもできます。内部ストリームは FIFO インターフェイスとしてのみインプリメントされます。

ストリームで作成された FIFO はデフォルトで深さ 1 を使用して、2 エレメント レジスタ ベースの FIFO としてインプリメントされます。FIFO の深さにはオプションで STREAM 指示子を使用できます。

グローバルおよびローカルストリーム

ストリームは、ローカルまたはグローバルのいずれかに定義できます。ローカルストリームは常に内部 FIFO としてインプリメントされます。グローバルストリームは FIFO またはポートとしてインプリメントされる可能性があります。

- 読み出し専用または書き込み専用のグローバルに定義されたストリームは、最上位 RTL ブロックの外部ポートとして推論されます。
- 最上位関数より下位の階層での読み出しおよび書き込み両方用のグローバルに定義されたストリームは、内部 FIFO としてインプリメントされます。

グローバル変数は、使用方法、ツール オプションおよび宣言で指定された C++ の修飾子によって、内部 FIFO または外部ポート（最上位 RTL ポート）のいずれかとして HLS デザインへ変換されます。

グローバルに宣言された `hls::stream<>` オブジェクトが内部 FIFO または外部ポート インターフェイスとして作成されるかは、次の規則に従って決定されます。

- `static` 修飾子が定義で指定されると、C/C++ に準拠するため、定義文と同じソースファイルで定義された関数からしかストリームにはアクセスできません。この場合、ストリームは内部で、FIFO としてインプリメントされます。
- `config_interface` コマンドが有効で、`-expose_global` オプションで `static` 以外のグローバル `hls::stream<>` オブジェクトすべてが指定される場合、オブジェクトは（ほかの型の `static` 以外のグローバル変数も）最上位 RTL ではポートとしてインプリメントされます。

ポートとしてインプリメントされない場合は、`hls::stream<>` オブジェクトは内部 FIFO としてデザインから読み出しおよび書き込まれます。内部リンクエージを持つようにできなかった場合は、最上位ポートとして公開されます。

ストリームの使用

`hls::stream<>` オブジェクトを使用するには、ヘッダーファイルの `hls_stream.h` を含める必要があります。このファイルは、Vivado HLS のインストールディレクトリのインクルード ディレクトリに含まれます（このディレクトリはシミュレーションおよび合成中に自動検索されるので、検索パスに追加する必要はありません）。

ストリーミングデータ オブジェクトは、型と変数名を指定して定義します。次の例では、128 ビットの符号なし整数型が定義されて、`my_wide_stream` というストリーム変数を作成するために使用されています。

```
#include <ap_int.h>
```

```
#include <hls_stream.h>

typedef ap_uint<128> uint128_t; // 128-bit user defined type
hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

ストリームが `hls::stream<T>` として指定されている場合は、T型はC++ネイティブデータ型、HLS任意精度型(例：`ap_int<>`、`ap_ufixed<>`など)またはユーザー定義(`typedef`)の構造型のいずれかの可能性があります。

注記：メソッド(メンバー関数)を含む通常のユーザー定義のクラス(または構造)は、ストリーム変数の型(T)として使用されるべきではありません。

ストリームは上記の例のようにスコープ付きの名前を使用するか、「`using namespace hls;`」文の付いたファイルスコープに `hls namespace` を含める必要があります。namespaceが使用される場合、上記の例は次のように記述できます。

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;

typedef ap_uint<128> uint128_t; // 128-bit user defined type
stream<uint128_t> my_wide_stream; // hls:: no longer required
```

>演算子よりも、演算子を使用してメソッドを選択できるようになるので、`foo(hls::stream<char> &foo, ...)`のように参照渡し変数のみを使用することをお勧めします。

ブロッキングおよびノンブロッキングアクセスのメソッドがサポートされます。

- ノンブロッキングアクセスは、FIFOインターフェイスとしてのみインプリメントできます。
- `ap_fifo`ポートとしてインプリメントされるストリーミングポート(AXI4Streamリソースで定義される)には、ノンブロッキングアクセスを使用するべきではありません。

読み出しおよび書き込みのブロッキング

`hls::stream<>`オブジェクトへの基本的なアクセスは読み出しおよび書き込みのブロッキングで、これはクラスメソッド(メンバー関数)を使用して達成できます。これらのメソッドは、空のストリーム FIFOからの読み出いや、フルストリーム FIFOへの書き込みがあった場合、または `ap_hs`インターフェイスプロトコルにマップされたストリームに対してフルハンドシェイクが達成されるまで、実行を停止(ロック)します。

書き込みメソッドのブロッキング

次の例では、`src_var`変数がストリームに含まれています。

```
// Usage of void write(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream.write(src_var);
```

<<演算子はオーバーロードされているので、C++ストリームのストリーム挿入演算子(例：`iostreams`、`filestreams`など)と同様の方法で使用できます。書き込まれる `hls::stream<>` オブジェクトは、左側の引数として、書き込まれる値は右側に記述されます。

```
// Usage of void operator << (T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream << src_var;
```

読み出しメソッドのブロッキング

このメソッドでは、ストリームの冒頭から読み出し、値を `dst_var` 変数に代入します。

```
// Usage of void read(T &rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream.read(dst_var);
```

または、ストリームの次のオブジェクトは、単にそのストリームを左側のオブジェクトに代入 (=、+=、などを使用) すると読み出すことができます。

```
// Usage of T read(void)

hls::stream<int> my_stream;

int dst_var = my_stream.read();
```

>>演算子はオーバーロードされ、C++ ストリームのストリーム抽出演算子(例: `iostreams`、`filestreams` など)のように使用できます。 `hls::stream` は LHS 引数、およびデスティネーション変数の RHS として提供されています。

```
// Usage of void operator >> (T & rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream >> dst_var;
```

ノンブロッキング読み出しおよび書き込み

読み出しおよび書き込みのノンブロッキング メソッドも提供されています。これにより、空のストリーム FIFO からの読み出しやフルストリーム FIFO への書き込みがあつても、実行を続行させることができます。

これらのメソッドは、アクセスのステータスを示すブール値を戻します(問題なければ `true`、それ以外は `false`)。 `hls::stream` ストリームのステータスをテストするために、別のメソッドも提供されています。

注記: ノンブロッキング アクセスの詳細なメソッドはどれも、`ap_hs` プロトコルが選択された `hls::stream` インタフェイスでは使用できません。

C シミュレーション中、ストリームには無限サイズが含まれます。このため、ストリームがフルの場合は C シミュレーションで検証はできません。これらのメソッドは、FIFO サイズ(デフォルト サイズの 1 か STREAM 指示子で定義された任意のサイズ)が定義された場合の RTL シミュレーション中にのみ検証できます。

ノンブロッキング書き込み

このメソッドでは、`src_var` 変数を `my_stream` ストリームに含めようとし、含まれる場合はブール値 `true` が戻されるようにしています。それ以外の場合は `false` が戻されて、キューは影響を受けません。

```
// Usage of void write_nb(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;
```

```

if (my_stream.write_nb(src_var)) {
    // Perform standard operations
    ...
} else {
    // Write did not occur
    return;
}

```

フルかどうかのテスト

```
bool full(void)
```

このメソッドは、`hls::stream` オブジェクトがフルの場合にのみ `true` を戻します。

```

// Usage of bool full(void)

hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;

stream_full = my_stream.full();

```

ノンブロッキング読み出し

```
bool read_nb(T & rdata)
```

このメソッドでは、ストリームから値を読み出そうとし、問題なく読み出せた場合は `true` が戻されます。それ以外の場合は `false` が戻されて、キューは影響を受けません。

```

// Usage of void read_nb(const T & wdata)

hls::stream<int> my_stream;
int dst_var;

if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
    ...
} else {
    // Read did not occur
    return;
}

```

空かどうかのテスト

```
bool empty(void)
```

このメソッドは、`hls::stream` が空の場合にのみ `true` を戻します。

```

// Usage of bool empty(void)

hls::stream<int> my_stream;
int dst_var;
bool stream_empty;

fifo_empty = my_stream.empty();

```

次の例は、ノンブロッキング アクセスとフル/空のテストを組み合わせて、RTL FIFO がフルまたは空の場合にエラーをどのように処理するかを示しています。

```
#include "hls_stream.h"
using namespace hls;

typedef struct {
    short    data;
    bool     valid;
    bool     invert;
} input_interface;

bool invert(stream<input_interface>& in_data_1,
            stream<input_interface>& in_data_2,
            stream<short>& output
) {
    input_interface in;
    bool full_n;

    // Read an input value or return
    if (!in_data_1.read_nb(in))
        if (!in_data_2.read_nb(in))
            return false;

    // If the valid data is written, return not-full (full_n) as true
    if (in.valid) {
        if (in.invert)
            full_n = output.write_nb(~in.data);
        else
            full_n = output.write_nb(in.data);
    }
    return full_n;
}
```

ストリームを使用したデザイン例全体は、[Help] → [Welcome] → [Browse Examples] → [Design Examples] → [hls_stream] をクリックして表示される Vivado HLS のサンプルデザイン例のセクションに含まれます。

RTL の FIFO の深さの制御

ストリーミングデータは通常 1 度に 1 サンプルを処理するので、ストリーミングデータを使用するほとんどのデザインでは、デフォルトの RTL FIFO の深さ 1 は十分な深さです。

インプリメンテーションで FIFO の深さが 1 より多く必要なマルチレート デザインの場合は、STREAM 指示子を使用して RTL シミュレーションが終了するために必要な深さを設定する必要があります。FIFO の深さが十分でない場合、RTL 協調シミュレーションが停止することがあります。

ストリーム オブジェクトは、GUI の [Directives] タブからは表示できません。このため、STREAM 指示子は GUI の [Directives] タブからは直接適用できませんが、`hls::stream<>` オブジェクトが宣言されている（または引数リストに使用されているか存在している）関数で右クリックすると、STREAM 指示子が選択されます。変数フィールドには、そのストリーム変数名を入力します。

または、`directives.tcl` ファイルで STREAM 指示子を手動で指定するか、ソースに `pragma` として追加します。

RTL 協調シミュレーションのサポート

現時点では、Vivado HLS の RTL 協調シミュレーション機能では次の状態はサポートされていません（合成でサポートされます）。

最上位インターフェイスの `hls::stream<>` の配列

```
void dut_top(uint16_t odata[N], hls::stream<uint8_t>chan[4]) { ... }
```

最上位インターフェイスに `hls::stream<>` メンバーを含む構造またはクラス

```
typedef struct {
    hls::stream<uint8_t> a;
    hls::stream<uint16_t> b;
} strm_strct_t;
```

```
void dut_top(strm_strct_t indata, strm_strct_t outdata) { ... }
```

これらの制限は、最上位関数の引数とグローバルに宣言されたオブジェクトの両方に適用されます。ストリームの配列または構造体が合成に使用される場合は、外部の RTL シミュレータとユーザーの作成した HDL テストベンチを使用してデザインを検証する必要があります。内部リンクエージの厳しい `hls::stream<>` オブジェクトには、このような制限はありません。

C の任意精度整数型

C のネイティブデータ型は、8 ビット境界(8、16、32、64 ビット)にあります。ただし、RTL 信号および演算では、任意のビット長がサポートされます。Vivado HLS では C の任意精度データ型が提供されており、C コードの変数および演算を任意のビット幅(6 ビット、17 ビット、234 ビットなど、最大 1024 ビットまで)で指定できます。

注記 : Vivado HLS では、C++ の任意精度データ型も提供されており、SystemC の一部である任意精度データ型がサポートされます。これらのデータ型については、C++ および SystemC コード記述方法でそれぞれ説明します。

任意精度データ型を使用する利点は、次のとおりです。

- ハードウェアの質の改善: たとえば 17 ビットの乗算器が必要な場合、計算で調度 17 ビットが使用されるように指定するために任意精度型を使用できます。
 - 任意精度データ型を使用しない場合、このような乗算(17 ビット)は 32 ビットの整数データ型を使用してインプリメントする必要があるので、複数の DSP48 コンポーネントを使用して乗算がインプリメントされることになります。
- 正確な C シミュレーション/解析: C コードの任意精度データ型を使用すると、正確なビット幅を使用して C シミュレーションを実行でき、合成前にアルゴリズムの機能(および精度)を検証できます。

このセクションの残りの部分では、任意精度型の使用方法と注意すべき問題点について記述します。任意精度型の詳細な説明については、次の内容を含む本書の最後のリファレンス セクション(C 任意精度型)を参照してください。

- 任意精度の整数に(64 ビットを超える値も含めて)定数および初期値を代入する方法
- 表示、連結、ビットスライス、範囲選択などの Vivado HLS のヘルパー関数の詳細
- シフト演算(負のシフト値は反対方向でシフトになる)の記述も含めた演算子の動作の詳細

C 言語での任意精度データ型の使用

C 言語では、ヘッダーファイル `ap_cint.h` により任意精度の整数データ型 `[u] int#W` が定義されます。たとえば、`int8` は 8 ビットの符号付き整数データ型を表し、`uint234` は 234 ビットの符号なし整数データ型を表します。

`ap_cint.h` ファイルは、`$HLS_ROOT/include` ディレクトリ (`$HLS_ROOT` は Vivado HLS のインストールディレクトリ) にあります。

例 4-35 のコードは、前述の基本的な演算例(例 4-22)のコードを繰り返したもので、どちらの例でも合成される最上位関数のデータ型は `dinA_t`、`dinB_t` などと指定されています。

```
#include "apint_arith.h"
```

```

void apint_arith(din_A  inA, din_B  inB, din_C  inC, din_D  inD,
                  out_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4
) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;

}

```

例 4-35 : 基本的演算

2 つの例の違いは、データ型の定義方法です。C 関数で任意精度の整数データ型を使用するには、次の手順に従います。

- ソース コードにヘッダーファイル ap_cint.h を追加します。
- ネイティブ C 型を任意精度型の intN または uintN (N はビット サイズを表す 1 ~ 1024 の値) に変更します。

データ型はヘッダーファイル apint_arith.h で定義されます (例 4-36)。例 4-22 と比較すると、入力データ型は単に実際の入力データの最大サイズを表すように削減されています (例: 8 ビット入力 inA を 6 ビット入力に削減)。ただし、出力はさらに正確に改良されています。たとえば inA と inB の合計である out2 は 32 ビットではなく 13 ビットだけ必要です。

```

#include <stdio.h>
#include "ap_cint.h"

// Previous data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;

typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;

void apint_arith(dinA_t inA, dinB_t inB, dinC_t inC, dinD_t inD, dout1_t
*out1, dout2_t *out2, dout3_t *out3, dout4_t *out4);

```

例 4-36 : 基本演算 APINT 型

例 4-36 が合成されると、例 4-22 と同じ機能のデザイン (データは例 4-36 で指定された範囲内) になりますが、最終 RTL デザインでは、エリアは小さく、クロック速度は高速になります。

ただし、合成前に関数がコンパイルおよび検証される必要があります。

C 言語での任意精度データ型の検証

任意精度型を作成するには、`ap_cint.h` ファイルのビット サイズを定義するための属性を追加します。`gcc` などの標準 C コンパイラでは、ヘッダーファイルで使用される属性がコンパイルされますが、その属性の意味は解釈できません。標準 C コンパイラで最終的な実行ファイルが作成されるときに、次のようなメッセージが表示されます。

```
$HLS_ROOT/include/etc/autopilot_dt.def:1036: warning: bit-width attribute directive
ignored
```

この後、シミュレーションにネイティブ C データ型が使用されますが、コードのビット精度動作は結果に反映されません。たとえば、2 進数記述 100 の 3 ビットの整数値は -4 ではなく、10 進数値の 4 として処理されます。

Vivado HLS に含まれる `apcc` というコンパイラを使用すると、この制限を克服し、関数をビット精度でコンパイルおよびシミュレーションできます。



重要: ビット精度型が C に含まれる場合、デザインは `apcc` コンパイラを使用してコンパイルおよびシミュレーションされる必要があります。

`apcc` コンパイラは [Project] → [Project Settings] → [Simulation] で [Use APCC Compiler] をオンにすると使用できます (図 4-3)。

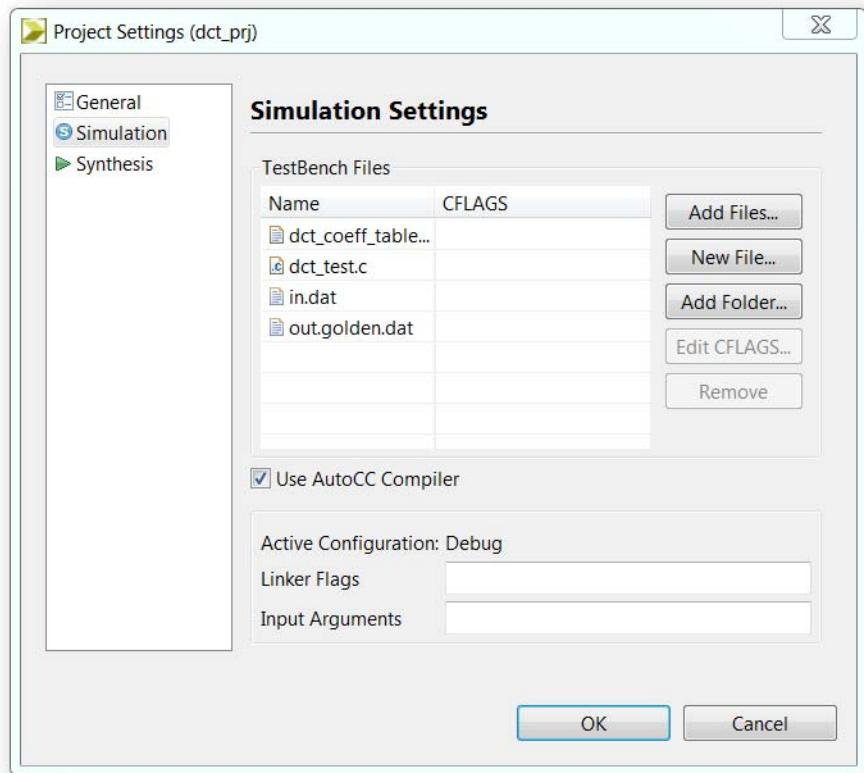


図 4-3 : APCC コンパイラのイネーブル

コマンド プロンプトでコンパイルする場合は、シェルプロンプトで `apcc` コンパイラを使用してください。これは `gcc` に互換しており、任意精度演算を正しく処理します (ビット幅情報で指定された制限を保持)。

apcc を使用すると Vivado HLS ヘッダーファイルが自動的に含まれ (-I\$HLS_ROOT/include は不要)、デザインを正しいビット精度の動作でシミュレーションできます。

```
$ apcc -o foo_top foo_top.c tb_foo_top.c
$ ./foo_top
```

つまり、C 言語で任意精度型を使用する場合は、apcc コンパイラを使用してコンパイルしてください。

- GUI で [Use APCC Compiler] をオンにします。
- コマンド プロンプトで gcc の代わりに apcc を使用します。

C++ または SystemC で指定された関数では、任意精度型を使用した場合にそのような制限はありません。または、ファイルの拡張子を .cpp に変更し、C++ 任意精度型を使用して、C++ コンパイラを使用してコンパイル/シミュレーションします。

C 言語での任意精度データ型のデバッグ

注記 : C コードをコンパイルするのに apcc が使用されると、デザインを Vivado HLS の C デバッガーで解析することはできなくなります。これが、C コードで任意精度型を使用する欠点です。

デザインをデバッグする必要がある場合は、次の方法に従ってください。

- アルゴリズムの操作にデバッガーでの解析が必要な場合、ネイティブ C 型 (int, char, short など) を使用し、コードをデバッガーで開いてアルゴリズムの操作が正しいかどうかを検証します。
 - これは、通常すべてのデータ型がヘッダーファイルで定義されていると、データ型スループットが 1 つのロケーションで変更できるので、簡単に実行できます。
- アルゴリズムの操作が正しいとわかっている場合、および任意精度型のビット精度の高い性質を解析する必要があるだけの場合、C シミュレーションを実行し、printf および fprintf 関数を使用して解析用にデータ値を出力します。

整数拡張問題

任意精度演算の結果がネイティブビット値の 8、16、32、64 ビット境界を越える場合は、注意が必要です。次の例では、2 つの 18 ビット値が乗算され、結果が 36 ビットに格納されています。

```
#include "ap_cint.h"

int18  a,b;
int36  tmp;

tmp = a * b;
```

ただし、この例の場合、整数拡張が実行されるため、結果は予測どおりになりません。

整数拡張では、C コンパイラが乗算結果を 18 ビットから次のネイティブビット サイズ (32 ビット) に拡張し、その結果を 32 ビット変数の tmp に代入します。これにより、動作と不正な結果が 図 4-3 に示すようになります。



図 4-3 : 整数拡張

Vivado HLS では、C シミュレーション同じ結果になるので、32 ビットの乗算結果が 36 ビットの結果に符号拡張されるハードウェアが作成されます。

整数拡張問題は、演算子入力を出力サイズの型に変更すると回避できます。図 4-37 は、乗算前に乗算器への入力を 36 ビット値に変換する例を示しています。これで C シミュレーション中に正しい(予測どおりの)結果になり、RTL で予測された 36 ビット乗算になります。

```
#include "ap_cint.h"

typedef int18 din_t;
typedef int36 dout_t;

dout_t apint_promotion(din_t a,din_t b) {
    dout_t tmp;

    tmp = (dout_t)a * (dout_t)b;
    return tmp;
}
```

例 4-37 : 整数拡張を回避するための型変換

整数拡張を避けるための型変換は、演算結果が次のネイティブ境界(8、16、32 または 64)よりも大きい場合にのみ必要です。これは、加算および減算よりも乗算でよく発生します。

整数拡張問題は、C++ または SystemC の任意精度型を使用する場合は発生しません。

関数

最上位関数は、合成後に最上位 RTL デザインになり、下位関数は RTL デザインではブロックに合成されます。



重要 : 最上位関数は、static 関数にはできません。

合成後、デザインの各関数に対して独自の合成レポートと RTL HDL ファイル(Verilog、VHDL、SystemC)が作成されます。下位関数は、オプションでインライン化して、そのロジックと周囲の関数のロジックとを統合できます。インライン化する関数により最適化が改善されることはありますが、より多くのロジックと可能性がメモリに保持されて解析されるので、ランタイムは増加します。Vivado HLS では、小さい関数のインライン化が自動的に実行されます(その関数に対する `inline` 指示子を `off` に設定するとディスエーブルにできます)。関数がインライン化されると、その関数に対するレポートおよび別の RTL ファイルは作成されません。ロジックおよびループは 1 つ上の階層の関数と統合されます。

関数のコード記述方法は、主に関数の引数およびインターフェイスに影響します。

関数への引数サイズが正確に記述されていれば、Vivado HLS でこの情報がデザインに伝搬されるので、すべての変数に対して任意精度型を作成する必要はありません。次の例では、2つの整数が乗算されますが、下位の 24 ビットのみが結果に使用されます。

```
#include "ap_cint.h"

int24 foo(int x, int y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

このコードが合成されると、出力が 24 ビットに切り捨てられた 32 ビット乗算器になります。

ただし、例 4-38 のように入力のサイズが正しく 12 ビット型 (int12) になっていれば、最終 RTL では 24 ビット乗算器が使用されます。

```
#include "ap_cint.h"
typedef int12 din_t;
typedef int24 dout_t;

dout_t func_sized(din_t x, din_t y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

例 4-38 : サイズの正しい関数引数

2 つの関数入力に任意精度型を使用するだけで、Vivado HLS では 24 ビット乗算器を使用するデザインが作成され、12 ビット型がデザインに伝搬されます。階層内のすべての関数の引数のサイズを正しく記述することをお勧めします。

通常は、変数が関数インターフェイスから、特に最上位関数インターフェイスから直接駆動されると、最適化の中に実行されないものがでてくることがあります。典型的な例は、入力がループ インデックスの上位制限として使用される場合です。

ループ

ループはアルゴリズムの動作を示す簡単な方法で、C コードでよく使用されます。ループは合成でサポートされ、パイプライン化、展開、部分展開、統合、平坦化できます。

最適化の展開、部分展開、平坦化、統合により、コードを変更したかのようにループ構造を効率的に変更できるので、ループの最適化に必要なコード変更を少なくできます。ただし、特定の状況にしか適用できない最適化もあり、そのためにコードを変更する必要のあることもあります。



推奨：ループ インデックス変数にグローバル変数を使用すると、実行されない最適化があるため、使用しないでください。

可変ループ境界

Vivado HLS で適用できる最適化の中には、ループに可変境界があると実行されないものがあります。例 4-39 は、ループ境界が可変幅で決定され、最上位入力から駆動されます。この例の場合、Vivado HLS ではループがいつ終了するのか判断できないので、ループには可変境界があると考えられます。

```
#include "ap_cint.h"
```

```

#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t code028(din_t A[N], dsel_t width) {

    dout_t out_accum=0;
    dsel_t x;

    LOOP_X:for (x=0;x<width; x++) {
        out_accum += A[x];
    }

    return out_accum;
}

```

例 4-39 : 可変ループ境界

例 4-39 のデザインを最適化しようとすると、可変ループ境界による問題がわかります。

可変ループ境界に関する最初の問題は、Vivado HLS でループのレイテンシが決定できなくなる点です。Vivado HLS はループの一巡目が終了するまでのレイテンシは決定できますが、可変幅の正確な値はスタティックに決定できないので、何度繰り返されるのかは判断できず、ループレイテンシ(ループの繰り返しすべてを完全に実行するまでのサイクル数)はレポートできません。

可変ループ境界があれば、Vivado HLS ではそのレイテンシを正確な値ではなく、疑問符(?)でレポートします。次は、例 4-39 の合成後の結果を示しています。

```

+ Summary of overall latency (clock cycles):
  * Best-case latency:    ?
  * Average-case latency: ?
  * Worst-case latency:   ?
+ Summary of loop latency (clock cycles):
  + LOOP_X:
    * Trip count:?
    * Latency:      ?

```

このように、可変ループ境界の最初の問題は、デザインのパフォーマンスが未知になるという点にあります。

Vivado HLS では、この問題を回避するために tripcount 指示子が提供されています。tripcount 指示子を使用すると、ループに指定する最小、平均、最大の tripcount(ループの繰り返し回数)を指定できます。例 4-39 で最大の tripcount の 32 が LOOP_X に適用されると、レポートは次のようにアップデートされます。

```

+ Summary of overall latency (clock cycles):
  * Best-case latency:    2
  * Average-case latency:18
  * Worst-case latency:   34
+ Summary of loop latency (clock cycles):
  + LOOP_X:
    * Trip count:0 ~ 32
    * Latency:      0 ~ 32

```

注記 : tripcount 指示子に対してユーザーが指定した値は、レポートにのみ使用され、合成には影響しません。tripcount の値は、単に Vivado HLS でレポートの数値を変更可能にします。これにより、最適化の効果がわかりやすくなり、ソリューションが比較できるようになります。

tripcount 指示子を使用しても、合成結果には影響ありません。

次は、高スループットの [例 4-39](#) を最適化する手順を示しています。

- ループを展開し、並列累算が行われるようにします。
- 配列入力を分割しないと、並列累算が 1 つのメモリポートに制限されます。

これらの最適化が適用されると、Vivado HLS で可変境界ループに関する最大の問題を示すメッセージが表示されます。

```
@W [XFORM-503] Cannot unroll loop 'LOOP_X' in function 'code028': cannot completely
unroll a loop with a variable trip count.
```

可変境界ループは展開できないので、`unroll` 指示子が適用できないだけでなく、そのループの上のレベルのパイプライン処理もできません。



重要: ループまたは関数がパイプライン処理されると、Vivado HLS はその関数またはループの下の階層ですべてのループを展開します。この階層に可変境界を含むループがあると、パイプライン処理はできません。

この問題は、ループ内で条件付き実行を使用し、ループの繰り返し回数を固定値にすると回避できます。[例 4-39](#) のコードは、[例 4-40](#) のように書き直すことができます。この例では、ループ境界は可変幅の最大値に設定され、ループ本体は条件付きで実行されます。

```
#include "ap_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 dsel_t;

dout_t loop_max_bounds(din_t A[N], dsel_t width) {

    dout_t out_accum=0;
    dsel_t x;

    LOOP_X:for (x=0;x<N-1; x++) {
        if (x<width) {
            out_accum += A[x];
        }
    }

    return out_accum;
}
```

例 4-40: 可変ループ境界 (修正後)

[例 4-40](#) の `for` ループ (`LOOP_X`) は展開できます。これは、ループに上位境界があり、Vivado HLS でハードウェアがどれくらい作成されるか認識されるからです。RTL デザインのループ本体には `N` (32) コピーができ、このループ本体の各コピーにそれに関する条件付きロジックが含まれ、可変幅の値によって実行されます。

ループのパイプライン処理

ループをパイプライン処理する際は、通常一番内部のループをパイプライン処理すると、エリアとパフォーマンスの最適なバランスがわかります。これにより、ランタイムも最速になります。[例 4-41](#) のコードは、ループおよび関数をパイプライン処理した場合のトレードオフを示しています。

```

#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {

    int i,j;
    static dout_t acc;

    LOOP_I:for(i=0; i < 20; i++) {
        LOOP_J: for(j=0; j < 20; j++) {
            acc += A[i] * j;
        }
    }

    return acc;
}

```

例 4-41: ループのパイプライン

一番内側のループ (LOOP_J) がパイプライン処理されると、ハードウェアには LOOP_J のコピーが 1 つ (乗算器 1 つ) でき、Vivado HLS はその外側のループ (LOOP_I) を使用して LOOP_J に新しいデータを提供します。乗算器演算 1 つと配列アクセス 1 回のみをスケジュールする必要があります。これらをスケジュールしておくと、ループの繰り返しは 1 つのループ本体のエンティティ (20X20 のループ繰り返し) としてスケジュールできます。

注記 : ループまたは関数がパイプライン処理される場合は、そのループまたは関数の下の階層にあるループを展開する必要があります。

外側のループ (LOOP_I) がパイプライン処理されると、内部のループ (LOOP_J) が展開され、そのループの本体のコピーが 20 個作成されます。このため、乗算器 20 個と配列アクセス 20 回をスケジュールする必要があります。これで LOOP_I の各繰り返しを 1 つのエンティティとしてスケジュールできるようになります。

最上位関数がパイプライン処理される場合は、どちらのループも展開する必要があります。このため、乗算器 400 個と配列アクセス 400 回をスケジュールする必要があります。ただし、Vivado HLS で 400 個の乗算器が作成されることはありません。これは、ほとんどのデザインで、データ依存性のために最大の並列処理ができないことがよくあるからです。たとえば、この例の場合、デュアルポート RAM が A[N] に使用されても、デザインはクロックサイクルの A[N] の 2 つの値にしかアクセスできません。

パイプライン処理する階層レベルを選択すると、たとえば一番内側のループをパイプライン処理すると、ほとんどのアプリケーションで一般的に許容されるスループットで最小のハードウェアが提供されます。階層の上位をパイプライン処理すると、すべての下位ループが展開されるので、スケジュールするためにさらに多くの演算が作成されますが (ランタイムとメモリ容量に影響する可能性あり)、スループットとレイテンシの観点から、通常はパフォーマンスの最も高いデザインになります。

上記のオプションをまとめると、次のようにになります。

- **LOOP_J のパイプライン** : レイテンシは約 400 サイクル (20X20) になり、100 個未満の LUT およびレジスタが必要になります (IO 制御および FSM は常にあります)。
- **LOOP_I のパイプライン** : レイテンシは約 20 サイクルになりますが、数百個の LUT およびレジスタが必要になります。ロジック数は、最初のオプションの約 20 倍の数からロジック最適化で処理されるロジックを引いた数になります。
- **パイプライン関数 loop_pipeline** : レイテンシは約 10 個 (デュアルポートアクセス 20 回) になりますが、何千個もの LUT およびレジスタが必要となります。ロジック数は最初のオプションの約 400 倍からロジック最適化で処理されるロジックを引いた数になります。

不完全なネスト ループ

一番内側のループ階層がパイプライン処理されると、Vivado HLS はネストされたループをフラットにして、ループの遷移 (ループの入出時にループインデックスで実行されるチェック) によって発生したサイクルを削除することで、レイテンシを削減してスループット全体を改善します。このようなチェックは、1 つのループから次のループへの遷移

の際にクロック遅延を発生させます。例 4-41 のように一番内側のループをパイプライン処理すると、Vivado HLS で次のようなメッセージが表示されます。

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_pipeline'.
```

ネストループは、ループが完全または半完全である場合にのみフラットにできます。

- 完全ループ
 - 一番内側のループにのみ本体が含まれます。
 - ループ文間に指定されるロジックはありません。
 - ループ境界は定数です。
- 半完全ループ
 - 一番内側のループにのみ本体が含まれます。
 - ループ文間に指定されるロジックはありません。
 - 一番外側のループは可変にできます。

例 4-42 は、ループネストが不完全な例を示しています。

```
#include "loop_imperfect.h"

void loop_imperfect(din_t A[N], dout_t B[N]) {
    int i,j;
    dint_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        acc = 0;
        LOOP_J: for(j=0; j < 20; j++) {
            acc += A[i] * j;
        }
        B[i] = acc / 20;
    }
}
```

例 4-42: 不完全なネストループ

LOOP_I 内 (LOOP_J 外側) の acc および配列 B[N] への代入があると、ループがフラットにならないようにできます。例 4-42 の LOOP_J がパイプライン処理されると、合成レポートには次のように表示されます。

- ```
+ Summary of loop latency (clock cycles):
 + LOOP_I:
 * Trip count:20
 * Latency: 480
 + LOOP_J:
 * Trip count: 20
 * Latency: 21
 * Pipeline II: 1
 * Pipeline depth:2
```
- パイプラインの深さにより、LOOP\_J を 1 回反復するのに 2 クロックかかるのがわかります (これはデバイス テクノロジおよびクロック周期によって異なります)。
  - 新しい反復は、各クロックサイクルごとに開始できます。Pipeline II は 1 です (II はループ本体の各新規実行間のサイクルを示す開始間隔 (Initiation Interval))。

- 最初の反復が結果を出力するまでに 2 サイクルかかります。パイプライン処理のため、続く反復実行は前の反復を使用して平行に実行され、1 クロック後に値を出力します。ループのレイテンシの合計は、 $2 + \text{残りの } 19 \text{ 回の反復ごとに } 1 \text{ で、} 21$  になります。
- LOOP\_I は 20 反復を実行するのに 480 クロックを必要とするので、LOOP\_I の各反復は 24 クロック サイクルです。LOOP\_J への入出には 3 サイクルのオーバーヘッドがあります ( $24 - 21 = 3$ )。

不完全なループ ネストの場合、またはループをフラットにできない場合、ループの入出のためにクロック サイクルが追加されます。例 4-42 のコードは、ネストされたループを完全にし、フラットにできるようにするために記述し直すことができます。

例 4-43 は、例 4-42 と同じ機能ですが、ループがフラットにできるように、条件文を LOOP\_J ループに追加したところを示しています。

```
#include "loop_perfect.h"

void loop_perfect(din_t A[N], dout_t B[N]) {
 int i,j;
 dint_t acc;

 LOOP_I:for(i=0; i < 20; i++) {
 LOOP_J: for(j=0; j < 20; j++) {
 if(j==0) acc = 0;
 acc += A[i] * j;
 if(j==19) B[i] = acc / 20;
 }
 }
}
```

#### 例 4-43: 完全なネスト ループ

例 4-43 が合成されると、このループはフラットになります。

```
@I [XFORM-541] Flattening a loop nest 'LOOP_I' in function 'loop_perfect'.
```

合成レポートには、レイテンシの改善が示されます。

```
+ Summary of loop latency (clock cycles):
+ LOOP_I_LOOP_J:
 * Trip count: 400
 * Latency: 401
 * Pipeline II: 1
 * Pipeline depth: 2
```

デザインにネスト ループが含まれる場合は、結果を解析して、なるべく多くのループがフラットになるようにします。ログ ファイルまたは上記のような合成レポートで、ループ ラベルが統合されたかどうかを確認してください (LOOP\_I および LOOP\_J は LOOP\_I\_LOOP\_J としてレポートされるようになります)。

## ループの並列処理

Vivado HLS では、レイテンシを削減するために、ロジックおよび関数ができるだけ早い段階でスケジュールされます。これを実行するため、なるべく多くのロジック演算および関数が並列にスケジュールされますが、ループを並列に実行することはできません。

例 4-44 が合成されると、SUM\_X ループがスケジュールされ、その後 SUM\_Y ループがスケジュールされます。SUM\_Y ループの開始は SUM\_X ループの完了を待つ必要がなくとも、SUM\_X の後にスケジュールされます。

```

#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
 dsel_t xlimit, dsel_t ylimit) {

 dout_t X_accum=0;
 dout_t Y_accum=0;
 int i,j;

 SUM_X:for (i=0;i<xlimit; i++) {
 X_accum += A[i];
 X[i] = X_accum;
 }

 SUM_Y:for (i=0;i<ylimit; i++) {
 Y_accum += B[i];
 Y[i] = Y_accum;
 }
}

```

#### 例 4-44: シーケンシャル ループ

これらのループには異なる境界 (xlimit および ylimit) があるため、統合はできません。ただし、[例 4-45](#) のようにループを別の関数に含めると、まったく同じ機能を達成でき、どちらのループも処理できます。

```

#include "loop_functions.h"

void sub_func(din_t I[N], dout_t O[N], dsel_t limit) {
 int i;
 dout_t accum=0;

 SUM:for (i=0;i<limit; i++) {
 accum += I[i];
 O[i] = accum;
 }
}

void loop_functions(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
 dsel_t xlimit, dsel_t ylimit) {

 dout_t X_accum=0;
 dout_t Y_accum=0;
 int i,j;

 sub_func(A,X,xlimit);
 sub_func(B,Y,ylimit);
}

```

#### 例 4-45: 関数としてのシーケンシャル ループ

[例 4-45](#) が合成されると、レイテンシは [例 4-44](#) の半分になります。これは、ループが関数として並列に実行できるようになったからです。

[例 4-44](#) には、dataflow 最適化も使用できます。ここに示す並列処理のため、関数にループを取り込む方法は、dataflow 最適化が使用できない場合に使用します。たとえば、より大型のデザイン例では、dataflow 最適化を最上位のすべてのループ/関数、および各最上位ループおよび関数間に配置されたメモリに適用できます。

## ループ依存性

ループ依存性は、ループを最適化されないように(通常はパイプライン処理)するデータ依存性のことです。これらは、ループの1回の反復内またはループ内の異なる反復間にできます。

ループ依存性を理解するには、極端な例を見てみるのが簡単です。次の例では、ループの結果がそのループの継続/終了条件として使用されています。ループの各反復は次のループが開始できる状態になる前に終了している必要があります。

```
Minim_Loop: while (a != b) {
 if (a > b)
 a -= b;
 else
 b -= a;
}
```

この場合、ループはパイプライン処理できません。ループの次の反復は前の反復が終了するまで開始できないからです。

すべてのループ依存性がこのように極端なわけではありませんが、ほかの演算が終了するまで開始できない演算があることに注意してください。ソリューションとしては、最初の演算ができるだけ早い段階で実行されるようにします。

ループ依存性はすべてのデータ型で発生する可能性はありますが、特に配列を使用する場合によく発生します。これについては、次の「配列」セクションで説明します。

## 配列

配列は、通常合成後にメモリ (RAM、ROM、または FIFO) としてインプリメントされます。「インターフェイスの配列」セクションに記述したように、最上位関数インターフェイスの配列はメモリ外部にアクセスする RTL ポートとして合成されます。デザインに対して内部にある配列は、最適化設定によって内部 BRAM、LUTRAM またはレジスタとして合成されます。

ループと同様、配列も簡単なコード構文なので、C プログラムでよく使用されます。また、ループのように、Vivado HLS には多くの最適化が含まれ、コードを修正しなくても RTL のインプリメンテーションを最適化するための指示子が含まれます。

配列により RTL で問題となるのは、次のような場合です。

- 配列アクセスがパフォーマンスの障害となってしまうことがあります。メモリとしてインプリメントされると、メモリポートの数によりデータへのアクセスが制限されます。
- 配列初期化は、注意して実行しないと、RTL でのリセットおよび初期化が不要に長くなってしまうことがあります。
- 読み出しアクセスのみを必要とする配列は、RTL では ROM としてインプリメントされるようにする必要があります。

ポインターの説明に示したように、ポインターの配列はサポートされますが、各ポインターはスカラーかスカラーの配列しか指定できません。

**注記** : 配列はサイズ指定する必要があります。

サポートされる例 : `Array[10];`

サポートされない例 : `Array[];`

## 配列アクセス

例 4-46 のコードでは、配列へのアクセスにより最終 RTL デザインでパフォーマンスが制限されます。`mem[N]` 配列へのアクセスが 3 回あり、合計結果が作成されています。

```
#include "array_mem_bottleneck.h"

dout_t array_mem_bottleneck(din_t mem[N]) {
 dout_t sum=0;
 int i;

 SUM_LOOP:for(i=2;i<N;++i)
 sum += mem[i] + mem[i-1] + mem[i-2];

 return sum;
}
```

#### 例 4-46:配列とメモリの障害

合成中、配列は RAM としてインプリメントされます。RAM がシングルポート RAM として指定される場合、SUM\_LOOP ループをパイプライン処理して、各クロックサイクルごとに新規ループ反復を処理できます。

SUM\_LOOP を初期間隔 1 でパイプライン処理しようとすると、次のようなメッセージが表示されます(スループット 1 が達成できなかったために、Vivado HLS は自動的に制約を解除します)。

```
@I [SCHED-61] Pipelining loop 'SUM_LOOP'.
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_1',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 1).
@W [SCHED-69] Unable to schedule 'load' operation ('mem_load_2',
array_mem_bottleneck.c:54) on array 'mem' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result:Target II:1, Final II:3, Depth:4.
```

ここでの問題は、シングルポート RAM にはシングルデータポートしかないので、各クロックサイクルで 1 つの読み出し(および 1 つの書き込み)が実行できる点にあります。

- SUM\_LOOP サイクル 1: read mem[i];
- SUM\_LOOP サイクル 2: read mem[i-1], sum values;
- SUM\_LOOP サイクル 3: read mem[i-1], sum values;

デュアルポート RAM も使用できますが、クロックサイクルごとに 2 つのアクセスしか許容されません。合計値を計算するのに 3 つの読み出しが必要なので、クロックサイクルごとに新しい反復でループをパイプライン処理するためには、クロックサイクルごとに 3 つのアクセスが必要になります。



**注意:** メモリまたはメモリポートとしてインプリメントされる配列は、パフォーマンスの障害となることがよくあります。

例 4-46 をスループット 1 でパイプラインができるように変更すると、例 4-47 のコードになります。例 4-47 では、先行読み出しを実行して、データアクセスを手動でパイプライン処理することで、ループの各反復で指定される配列読み出しが 1 回だけになっています。これにより、パフォーマンスを達成するためには、シングルポート RAM だけが必要となります。

```
#include "array_mem_perform.h"

dout_t array_mem_perform(din_t mem[N]) {
 din_t tmp0, tmp1, tmp2;
 dout_t sum=0;
 int i;

 tmp0 = mem[0];
 tmp1 = mem[1];
```

```

SUM_LOOP:for (i = 2; i < N; i++) {
 tmp2 = mem[i];
 sum += tmp2 + tmp1 + tmp0;
 tmp0 = tmp1;
 tmp1 = tmp2;
}

return sum;
}

```

#### 例 4-47: パフォーマンス アクセスを使用した配列とメモリ

Vivado HLS には、配列のインプリメントおよびアクセス方法を変更できる最適化指示子が多くあります。通常、このような指示子が使用される場合、コードを変更する必要はありません。配列は、ブロックまたは個別エレメントに分割できます。Vivado HLS で配列が個別エレメントに自動的に分割されることもあります。これは、自動分割のコンフィギュレーション設定を使用すると指定できます。

配列が複数のブロックに分割されると、1つの配列は複数の RTL RAM ブロックとしてインプリメントされます。エレメントに分割される場合、各エレメントは RTL でレジスタとしてインプリメントされます。どちらの場合も、分割でさらに多くのエレメントが並列にアクセスできるようになります。パフォーマンスが向上します。デザインのトレードオフは、パフォーマンスとそれを達成するために必要な RAM またはレジスタの数です。

#### FIFO アクセス

配列アクセスに特に注意が必要なのは、配列が FIFO としてインプリメントされる場合です。よくあるのは、dataflow 最適化を使用する場合です。

FIFO へのアクセスは、位置 0 から開始してシーケンシャルな順番である必要があります。また、配列が複数位置で読み出される場合、コードでは FIFO アクセスの順番を厳しく指定する必要があります。複数ファンアウトを含む配列は、アクセスの順番を指定するコードを追加しないと、FIFO としてインプリメントできないことがあります。

#### 配列の初期化



**推奨:「型修飾子」** セクションで示すとおり、必須ではありませんが、static 修飾子を使用してメモリとしてインプリメントされる配列を指定することをお勧めします。これにより、Vivado HLS で RTL のメモリを使用して配列をインプリメントできるようになります。また、static 型の初期化動作を使用できるようになります。

次のコード例では、配列は値のセットを使用して初期化されます。関数が実行されるたびに、coeff 配列にこれらの値が代入されます。合成後、coeff をインプリメントする RAM が実行されるたびに、これらの値が読み込まれます。シングルポート RAM の場合は、8 クロックサイクルかかります。1024 の配列の場合、当然 1024 クロックサイクルかかります。この間、coeff によっては演算が行われません。

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

次のコードでは、static 修飾子を使用して coeff 配列を定義しています。配列は実行開始時に指定した値で初期化されます。関数は実行されますが、coeff 配列は前の実行からの値を記録しています。メモリが RTL で動作するように、static 配列は C コードで動作します。

```
static int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

また、変数に static 修飾子が含まれる場合、Vivado HLS は RTL デザインおよび FPGA ビットストリームで変数を初期化します。これにより、メモリを初期化するのに複数クロックサイクルも必要なくなり、大容量メモリが使用オーバーヘッドにならなくなります。

リセットを適用した(デフォルトではない)後に static 変数がその初期ステートに戻すかどうかは、RTL コンフィギュレーションコマンドで指定できます。メモリがリセット後に初期ステートに戻るのであれば、使用オーバーヘッドになり、値をリセットするのに複数サイクルが必要となり、値はそれぞれ各メモリアドレスに書き込まれる必要があります。

## ROM のインプリメント

例 4-29 に示すように、Vivado HLS ではメモリを合成するのに `static` 修飾子を、そのメモリを ROM に推論するのに `const` 修飾子を使用して配列を指定する必要はありません。Vivado HLS は、デザインの解析を実行し、最も最適なハードウェアを作成しようとしますが、

メモリにする配列には `static` 修飾子を使用することをお勧めします。これは、「配列の初期化」に示すように、`static` 型が RTL のメモリとほぼ同じように動作するためです。

Vivado HLS では常に ROM がデザイン解析で使用されるように推論されるわけではないので、配列が読み出し専用の場合は、`const` 修飾子も推奨されます。ローカルの `static` (グローバル以外) 配列は読み出しの前に書き込まれる必要があるというのが ROM の自動推論の一般的な規則です。次の例のように指定すると、ROM が推論されやすくなります。

- 配列をそれを使用する関数でできるだけ早い段階で初期化します。
- 書き込みをまとめます。
- `array(ROM)` 初期化書き込みを初期化コード以外のコードでインターリープしないようにします。
- 異なる値を同じ配列エレメントに格納しないようにします (書き込みはすべてコード内でまとめます)。
- エレメント値の計算は、初期化ループ カウンター変数を除いて、定数以外 (コンパイル時) のデザイン変数に依存しないようにします。

ROM を初期化するのに複雑な代入が使用される場合、たとえば `math.h` ライブラリからの関数例では、配列初期化を別々の関数に配置すると、ROM が推論されるようにできます。例 4-48 では、`sin_table[256]` 配列がメモリとして推論され、RTL 合成後は ROM としてインプリメントされます。

```
#include "array_ROM_math_init.h"
#include <math.h>

void init_sin_table(din1_t sin_table[256])
{
 int i;
 for (i = 0; i < 256; i++) {
 dint_t real_val = sin(M_PI * (dint_t)(i - 128) / 256.0);
 sin_table[i] = (din1_t)(32768.0 * real_val);
 }
}

dout_t array_ROM_math_init(din1_t inval, din2_t idx)
{
 short sin_table[256];
 init_sin_table(sin_table);
 return (int)inval * (int)sin_table[idx];
}
```

### 例 4-48: `math.h` を使用した ROM の初期化



ヒント: `sin()` 関数の結果は定数値になるので、`sin()` 関数をインプリメントするのに、RTL デザインでコアはありません。`sin()` 関数は表 2-1 にリストされるコアの 1 つではないので、C の合成でサポートされません (C++ で `math.h` 関数を使用する場合は、「C++ の合成」を参照してください)。

## サポートされない C コンストラクト

Vivado HLS では、C 言語が広範囲にわたってサポートされますが、合成ができず、後のデザインフローでエラーを発生させる C コンストラクトもあります。このセクションでは、このような関数が合成され FPGA デバイスにインプリメントされる場合の、コード変更が必要な箇所について説明します。

一般的な規則として、C 関数を合成できるようにするには、デザインのすべての機能を含め (OS へのシステム コールで機能を実行することは不可)、C コンストラクトが固定のサイズであり、これらのコンストラクトのインプリメンテーションが明確である必要があります。

## システム コール

システム コールは、C プログラムを実行している OS で一部のタスクが実行されるので、合成できません。

Vivado HLS は自動的によく使用される、`printf()` および `fprintf(stdout, )` のようなデータを表示するだけでアルゴリズムの実行には影響のないシステム コールを無視しますが、通常はシステム コールは合成できないので、合成前に関数から削除しておく必要があります。`getc()`、`time()`、`sleep()` などのシステム コールも、OS に呼び出しを実行するので合成できません。

Vivado HLS では、合成が実行されたときに `macro __SYNTHESIS__` が自動的に定義されます。これにより、`__SYNTHESIS__ macro` を使用して、デザインから合成不可能なコードを削除できるようになります。

例 4-49 は、下位関数からの中間結果が保存されるハード ドライブのファイルに保存される例を示しています。`__SYNTHESIS__` マクロを使用すると、合成不可能なファイルの書き込みが合成中に無視されるようになります。

```
#include "hier_func4.h"

int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
 *outSum = *in1 + *in2;
 *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
 *outA = *in1 >> 1;
 *outB = *in2 >> 2;
}

void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
 dint_t apb, amb;

 sumsub_func(&A, &B, &apb, &amb);
#ifndef __SYNTHESIS__
 FILE *fp1;// The following code is ignored for synthesis
 char filename[255];
 sprintf(filename, "Out_apb_%03d.dat", apb);
 fp1=fopen(filename, "w");
 fprintf(fp1, "%d \n", apb);
 fclose(fp1);
#endif
 shift_func(&apb, &amb, C, D);
}
```

### 例 4-49 : デバッグ用のファイルの書き込み

`__SYNTHESIS__` マクロを使用すると、C 関数からコード自体を削除せずに、合成不可能なコードを削除できます。ただし、このマクロを使用すると、シミュレーション用の C コードと合成用の C コードが異なることになります。



**注意 :** `__SYNTHESIS__` マクロを C コードの機能を変更するために使用すると、C シミュレーションと C 合成の結果が異なります。このようなコードのエラーはデバッグしにくいので、機能を変更するために `__SYNTHESIS__` マクロを使用するのはお勧めできません。

## ダイナミック メモリの使用

システム内のメモリ割り当てを管理するシステム コール、たとえば `malloc()`、`alloc()`、および `free()` は、OS のメモリにあるリソースを使用し、ランタイム中に作成およびリリースされます。ハードウェア インプリメンテーションを合成できるようにするには、デザインに必要なリソースがすべて指定され、含まれている必要があります。

メモリ割り当てのシステム コールは、合成前にデザインから削除する必要がありますが、ダイナミック メモリ操作がデザインの機能を定義するために使用されているので、同等の範囲が制限された表現に変換する必要があります。

例 4-50 は、`malloc()` を使用するデザインが合成可能なバージョンに変換される例を示しています。

例 4-50 の例には、次の 2 つのコード記述方法が示されます。

- 1. デザインでは `__SYNTHESIS__` が使用されず、合成可能なバージョンと合成不可能なバージョン間を選択にユーザー定義の `NO_SYNTH` マクロが使用されます。これにより、まったく同じコードが C でシミュレーションされて、Vivado HLS で合成されるようになります。
- 2. `malloc()` を使用する元のデザインのポインターは、固定サイズのエレメント用に書き直す必要はありません。固定サイズのリソースは作成でき、既存ポインターは単に固定サイズのリソースを指定するようにできます。この方法により、既存デザインを手動で書き直す必要はなくなります。

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifndef NO_SYNTH
 long long *out_accum = malloc (sizeof(long long));
 int* array_local = malloc (64 * sizeof(int));
#else
 long long _out_accum;
 long long *out_accum = &_out_accum;
 int _array_local[64];
 int* array_local = &_array_local[0];
#endif
 int i,j;

 LOOP_SHIFT:for (i=0;i<N-1; i++) {
 if (i<width)
 *(array_local+i)=din[i];
 else
 *(array_local+i)=din[i]>>2;
 }

 *out_accum=0;
 LOOP_ACCUM:for (j=0;j<N-1; j++) {
 *out_accum += *(array_local+j);
 }

 return *out_accum;
}
}
```

### 例 4-50 : `malloc()` の固定サイズ リソースへの変換



**推奨 :** ここでの変更はデザインの機能に影響するので、`__SYNTHESIS__` マクロの使用はお勧めできません。推奨される方法は、次のとおりです。

- 1.ユーザー定義のマクロ `NO_SYNTH` をコードに追加して、コードを修正します。
- 2.`NO_SYNTH` マクロをイネーブルにし、C シミュレーションを実行して結果を保存します。

3.NO\_SYNTH マクロをディスエーブルにし (たとえば、例 50 のようにコメントアウトし)、C シミュレーションを実行して結果が同じになるかどうか検証します。

このユーザー定義のマクロをディスエーブルにして合成を実行します。この方法を使用すると、アップデートされたコードが C シミュレーションで検証され、まったく同じコードが合成されるようにできます。

## ポインターの制限

### 一般的なポインターの型変換

ポインターの型変換は通常サポートされませんが、ネイティブ C 型同士ではサポートされます。ポインターの型変換については、「[ポインター](#)」を参照してください。

### ポインターの配列

ポインターの配列は、各ポインターがスカラーまたはスカラーの配列を指定する場合に、合成でサポートされます。ポインターの配列では、別のポインターを指定することはできません。ポインターの配列については、「[ポインター](#)」を参照してください。

## 再帰関数

再帰関数は合成できません。再帰関数は、次のような再帰が恒久的に繰り返される可能性のある関数に適用されます。

```
unsigned foo (unsigned n)
{
 if (n == 0 || n == 1) return 1;
 return (foo(n-2) + foo(n-1));
}
```

有限数の関数呼び出しがある末尾再帰もサポートされません。

```
unsigned foo (unsigned m, unsigned n)
{
 if (m == 0) return n;
 if (n == 0) return m;
 return foo(n, m%n);
}
```

C++ では、末尾再帰をインプリメントするためにテンプレートを使用できます。C++ については、次で説明します。

## C++ の合成

このセクションでは、Vivado HLS の合成で使用される C++ 言語について説明します。このセクションで C の合成に関する説明される項目 (最上位関数の引数、ポインター、ループ、配列など) のほとんどは、C++ のコード記述にも適用されます。このセクションの前に「C の合成」セクションを読んでおいてください。

「C の合成」の項目で C++ に適用されないものは、C で使用される任意精度型 (C++ には独自の任意精度型あり) および C で任意精度型をコンパイルする際の制限です。apcc は C++ シミュレーションには必要なく、C++ の任意精度型では整数拡張問題は発生しません。

このセクションでは、クラス、テンプレート、C++ 任意精度型、math.h ライブライアリおよび標準テンプレート ライブライアリのサポートなど、合成に関する C++ のその他の言語機能について説明します。

Vivado HLS では、C++ 関数には標準の C++ ファイル拡張子 (.cpp、.cxx など) が付いていると予測されます。適切に名前が変更され、Vivado HLS C 任意精度型の (u) int# で使用されない標準 C 言語関数は、C++ デザインとして合成できます。C++ オブジェクト指向のコーディング形式を使用するための要件はありません。

## C++ クラス

C++ クラスは、Vivado HLS での合成で完全にサポートされています。合成での最上位は関数である必要があり、クラスは合成で最上位にはできません。クラスのメンバー関数を合成するには、クラス自体を関数にインスタンシエートする必要があります。最上位クラスは、単純にテストベンチにインスタンシエートするべきではありません。例 4-51 は、CFIR クラス(次で説明するヘッダー ファイルで定義)が最上位関数の cpp\_FIR にインスタンシエートされ、FIR フィルターをインプリメントするために使用されるコードを示しています。

```
#include "cpp_FIR.h"

// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
{
 static CFir<coef_t, data_t, acc_t> fir1;

 cout << fir1;

 return fir1(x);
}
```

例 4-51: C++ FIR フィルター



**重要:** クラスおよびクラス メンバー関数は、合成では最上位にできません。クラスは最上位関数にインスタンシエートする必要があります。

例 4-51 で、デザインをインプリメントするために使用されるクラスを検証する前に、Vivado HLS が合成中に標準出力ストリーム カウントを自動的に無視することに注意してください。Vivado HLS で合成されると、次のような警告メッセージが表示されます。

```
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
@I [SYNCHK-101] Discarding unsynthesizable system call:'std::operator<<
<std::char_traits<char> >' (cpp_FIR.h:110)
@
```

次の例 4-52 に示す cpp\_FIR.h ヘッダー ファイルには、CFir クラスおよびそれに関連するメンバー関数が定義されています。この例では、演算子のメンバー関数 () および << はオーバーロードされる演算子です。それぞれ main アルゴリズムを実行するために使用され、cout と一緒に使用すると、C シミュレーション中に表示されるデータをフォーマットできます。

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define N 85

typedef int coef_t;
typedef int data_t;
typedef int acc_t;
```

```

// Class CFir definition
template<class coef_T, class data_T, class acc_T>
class CFir {
protected:
 static const coef_T c[N];
 data_T shift_reg[N-1];
private:
public:
 data_T operator()(data_T x);
 template<class coef_TT, class data_TT, class acc_TT>
 friend ostream&
 operator<<(ostream& o, const CFir<coef_TT, data_TT, acc_TT> &f);
};

// Load FIR coefficients
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[N] = {
 #include "cpp_FIR.inc"
};

// FIR main algorithm
template<class coef_T, class data_T, class acc_T>
data_T CFir<coef_T, data_T, acc_T>::operator()(data_T x) {
 int i;
 acc_t acc = 0;
 data_t m;

 loop: for (i = N-1; i >= 0; i--) {
 if (i == 0) {
 m = x;
 shift_reg[0] = x;
 } else {
 m = shift_reg[i-1];
 if (i != (N-1))
 shift_reg[i] = shift_reg[i - 1];
 }
 acc += m * c[i];
 }
 return acc;
}

// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T, data_T, acc_T> &f) {
 for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i++) {
 o << "shift_reg[" << i << "] = " << f.shift_reg[i] << endl;
 }
 o << "_____ " << endl;
 return o;
}

data_t cpp_FIR(data_t x);

```

#### 例 4-52: クラスを定義する C++ ヘッダー ファイル

テストベンチ 例 4-51 は 例 4-53 で示され、最上位関数の `cpp_FIR` が呼び出されて検証されているところを示しています。この例には、Vivado HLS 合成用にテストベンチの重要な属性が含まれます。

- 出力結果は、既知の良い値に対して比較されます。
- テストベンチは結果が正しいと確認されれば 0 を戻します。

テストベンチの詳細は、「[生産的なテストベンチの作成](#)」を参照してください。

```
#include "cpp_FIR.h"

int main() {
 ofstream result;
 data_t output;
 int retval=0;

 // Open a file to save the results
 result.open("result.dat");

 // Apply stimuli, call the top-level function and save the results
 for (int i = 0; i <= 250; i++)
 {
 output = cpp_FIR(i);

 result << setw(10) << i;
 result << setw(20) << output;
 result << endl;
 }
 result.close();

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
 if (retval != 0) {
 printf("Test failed !!!\n");
 retval=1;
 } else {
 printf("Test passed !\n");
 }

 // Return 0 if the test
 return retval;
}
```

#### 例 4-53:cpp\_FIR の C++ テストベンチ

### コンストラクター、デストラクターおよび仮想関数

クラス コンストラクターおよびデストラクターは、クラス オブジェクトが宣言されると含まれて合成されます。

Vivado HLS がエラボレーション中に関数を静的に決定できる場合、仮想関数は、抽象的なものも含め、合成でサポートされます。次は、仮想関数が合成でサポートされない例です。

- 仮想関数はマルチレイヤー インヘリタンス クラス階層で定義できますが、シングル インヘリタンスを使用してのみ定義できます。
- 動的なポリモーフィズムは、ポインター オブジェクトがコンパイル時に決定できる場合にのみサポートされます。たとえば、このようなポインターは if-else やループ文では使用できません。
- STL コンテナは、オブジェクトのポインターを含め、ポリモーフィズム関数を呼び出すためには使用できません。次に例を示します。

```
vector<base *> base_ptrs(10);
```

```

//Push_back some base_ptrs to vector.
for (int i = 0; i < base_ptrs.size(); ++i) {
 //Static elaboration cannot resolve base_ptrs[i] to actual data type.
 base_ptrs[i]->virtual_function();
}

• base オブジェクト ポインターがグローバル変数の場合はサポートされません。次に例を示します。
Base *base_ptr;

void func()
{

 base_ptr->virtual_function();

}

• base オブジェクト ポインターはクラス定義のメンバー変数にはできません。
// Static elaboration cannot bind base object pointer with correct data type.
class A
{

 Base *base_ptr;
 void set_base(Base *base_ptr);
 void some_func();

};

void A::set_base(Base *ptr)
{
 this.base_ptr = ptr;
}

void A::some_func()
{

 base_ptr->virtual_function();

}

• base オブジェクト ポインターまたはリファレンスがコンストラクターの関数パラメーター リストにある場合は、Vivado HLS でそれは変換されません (ISO C++ 規格のセクション 12.7 に記述されています。ビヘイビアは定義されないこともあります)。
class A {
 A(Base *b) {
 b-> virtual _ function ();
 }
};

```

## グローバル変数およびクラス

クラスでグローバル変数を使用すると、実行されない最適化があるので、使用しないようにしてください。例 4-54 では、フィルターのコンポーネントを作成するのにクラスが使用されています (polyd\_cell クラスはシフト、乗算、累算を実行するコンポーネントとして使用されます)。

```

typedef long long acc_t;
typedef int mult_t;
typedef char data_t;
typedef char coef_t;

```

```
#define TAPS 3
#define PHASES 4
#define DATA_SAMPLES 256
#define CELL_SAMPLES 12

// Use k on line 73 static int k;

template <typename T0, typename T1, typename T2, typename T3, int N>
class polyd_cell {
private:
public:
 T0 areg;
 T0 breg;
 T2 mreg;
 T1 preg;
 T0 shift[N];
 int k; //line 73
 T0 shift_output;
 void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
 {
 Function_label0:

 if (col==0) {
 SHIFT:for (k = N-1; k >= 0; --k) {
 if (k > 0)
 shift[k] = shift[k-1];
 else
 shift[k] = data;
 }
 *dataOut = shift_output;
 shift_output = shift[N-1];
 }
 *pcout = (shift[4*col]* coeff) + pcin;
 }
};

// Top-level function with class instantiated
void cpp_class_data (
 acc_t *dataOut,
 coef_t coeff1[PHASES][TAPS],
 coef_t coeff2[PHASES][TAPS],
 data_t dataIn[DATA_SAMPLES],
 int row
) {
 acc_t pcin0 = 0;
 acc_t pcout0, pcout1;
 data_t dout0, dout1;
 int col;
 static acc_t accum=0;
 static int sample_count = 0;
 static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell0;
 static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell1;

 COL:for (col = 0; col <= TAPS-1; ++col) {
```

```

polyd_cell0.exec(&pcout0, &dout0, pcin0, coeff1[row][col], dataIn[sample_count],
col);

polyd_cell1.exec(&pcout1, &dout1, pcout0, coeff2[row][col], dout0, col);

 if ((row==0) && (col==2)) {
 *dataOut = accum;
 accum = pcout1;
 } else {
 accum = pcout1 + accum;
 }

}
sample_count++;
}

```

#### 例 4-54: ループ インデックスの C++ クラス データ メンバー

polyd\_cell クラス内には、データをシフトするための SHIFT ループがあります。SHIFT ループで使用されるループ インデックスの k が削除され、k のグローバル インデックスに置換されると(前の例でも記述していましたが static int k と記述されてコメントアウトされていました)、Vivado HLS では polyd\_cell クラスの使用されるループまたは関数がパイプライン処理できなくなります。Vivado HLS では、次のようなメッセージが表示されます。

```
@W [XFORM-503] Cannot unroll loop 'SHIFT' in function 'polyd_cell<char, long long,
int, char, 12>::exec' completely: variable loop bound.
```

ループ インデックスにはグローバル変数以外のローカル変数を使用すると、Vivado HLS ですべての最適化が実行されます。

## テンプレート

前の例に示すように、Vivado HLS では合成用に C++ のテンプレートの使用がサポートされます。ただし、テンプレートは最上位関数ではサポートされません。



**重要:** 最上位関数にはテンプレートを使用できません。

例 4-52 および 例 4-54 に示すような一般的なテンプレートの使用方法だけでなく、テンプレートは標準 C 合成ではサポートされない再帰関数をインプリメントするために使用することもできます。

例 4-55 では、末尾再帰のフィボナッチ アルゴリズムをインプリメントするために、テンプレート化された struct が使用されています。合成を実行するためには、再帰の最終呼び出しをインプリメントするのに終端クラスを使用する必要があります。この場合、テンプレート サイズは 1 が使用されます。

```

//Tail recursive call
template<data_t N> struct fibon_s {
 template<typename T>
 static T fibon_f(T a, T b) {
 return fibon_s<N-1>::fibon_f(b, (a+b));
 }
};

// Termination condition
template<> struct fibon_s<1> {
 template<typename T>
 static T fibon_f(T a, T b) {

```

```

 return b;
 }
};

void cpp_template(data_t a, data_t b, data_t &dout) {
 dout = fibon_s<FIB_N>::fibon_f(a,b);
}

```

例 4-55: テンプレートを使用した C++ の末尾再帰

## データ型

C 言語の「データ型」セクションで説明したように、Vivado HLS では C++ と同じ標準データ型が合成でサポートされます。

C++ 任意精度整数もサポートされます。C++ 任意精度整数には、C で使用されるものとは異なり、シミュレーション制限がありません。C++ では、任意精度の固定小数点型もサポートされます。

### C++ の任意精度整数型

C++ のネイティブデータ型は、8 ビット境界 (8、16、32、64 ビット) にあります。ただし、RTL 信号および演算では、任意のビット長がサポートされます。

Vivado HLS では C++ の任意精度データ型が提供されており、C コードの変数および演算を任意のビット幅 (6 ビット、17 ビット、234 ビットなど、最大 1024 ビットまで) で指定できます。



**ヒント:** 幅のデフォルト最大値は 1024 ビットです。このデフォルトは、ap\_int.h ヘッダーファイルを含める前に、32768 以下の正の整数値でマクロ AP\_INT\_MAX\_W を定義すると上書きすることができます。

C++ では SystemC 規格で定義された任意精度型の使用がサポートされるので、単に SystemC ヘッダーファイルの systemc.h を含めるだけで、SystemC データ型が使用できます。SystemC データ型の詳細は、SystemC のセクションを参照してください。

任意精度データ型には、ネイティブ C++ データ型よりも次のような利点があります。

- ハードウェアの質の改善 :たとえば 17 ビットの乗算器が必要な場合、計算で調度 17 ビットが使用されるように指定するために任意精度型を使用できます。
  - 任意精度データ型を使用しない場合、このような乗算 (17 ビット) は 32 ビットの整数データ型を使用してインプリメントする必要があるので、複数の DSP48 コンポーネントを使用して乗算がインプリメントされることになります。
- 正確な C++ シミュレーション/解析 :C++ コードの任意精度データ型を使用すると、正確なビット幅を使用して C++ シミュレーションを実行でき、合成前にアルゴリズムの機能 (および精度) を検証できます。

C++ の任意精度型には、C の任意精度型で発生するような問題はありません。

- C++ 任意精度型は、標準 C++ コンパイラでコンパイルされます (「C 言語での任意精度データ型の検証」に示すように、appc と同等のものは C++ にはありません)。
- C++ 任意精度型には、整数拡張問題はありません。

ファイル拡張子を .c から .cpp に変更するのは珍しくないので、ファイルは上記のような問題がない場合は C++ としてコンパイルできます。

このセクションの残りの部分では、任意精度型の使用方法について説明します。任意精度型の詳細な説明については、次の内容を含む本書の最後のリファレンスセクション (「[C++ の任意精度型](#)」) を参照してください。

- 任意精度の整数に (1024 ビットを超える値も含めて) 定数および初期値を代入する方法

- 表示、連結、ビットスライス、範囲選択などの Vivado HLS のヘルパー メソッドの詳細
- シフト演算(負のシフト値は反対方向でシフトになる)の記述も含めた演算子の動作の詳細

## C++ 言語での任意精度データ型の使用

C++ 言語では、ヘッダー ファイル `ap_int.h` により任意精度の整数データ型 `ap_(u)int<W>` が定義されます。たとえば、`ap_int<8>` は 8 ビットの符号付き整数データ型を表し、`ap_uint<234>` は 234 ビットの符号なし整数データ型を表します。

`ap_int.h` ファイルは、\$HLS\_ROOT/include ディレクトリ (\$HLS\_ROOT は Vivado HLS のインストール ディレクトリ) にあります。

例 4-56 のコードは、前述の基本的な演算例(例 4-22 および例 4-35)のコードを繰り返したもので、この例で合成される最上位関数のデータ型は `dinA_t`、`dinB_t` などと指定されています。

```
#include "cpp_ap_int_arith.h"

void cpp_ap_int_arith(din_A inA, din_B inB, din_C inC, din_D inD,
 dout_1 *out1, dout_2 *out2, dout_3 *out3, dout_4 *out4)
{
 // Basic arithmetic operations
 *out1 = inA * inB;
 *out2 = inB + inA;
 *out3 = inC / inA;
 *out4 = inD % inA;
}
```

### 例 4-56: C++ 型で書き直した基本演算

このアップデートでは、C++ 任意精度型が使用されています。

- ソース コードにヘッダー ファイル `ap_int.h` を追加します。
- ネイティブ C++ 型を任意精度型 `ap_int<N>` または `ap_uint<N>` に変更します。ここでの N のビット サイズは 1 ~ 1024 になります(前述したとおり、これは必要であれば 32 ビットに拡張できます)。

データ型はヘッダー ファイル `cpp_ap_int_arith.h` で定義されます(例 4-36)。

例 4-22 と比較すると、入力データ型は単に実際の入力データの最大サイズを表すように削減されています(例: 8 ビット入力 `inA` を 6 ビット入力に削減)。ただし、出力はさらに正確に改良されています。たとえば `inA` と `inB` の合計である `out2` は 32 ビットではなく 13 ビットだけ必要です。

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_

#include <stdio.h>
#include "ap_int.h"

#define N 9

// Old data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
//typedef unsigned int dout2_t;
```

```

//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef ap_int<6> dinA_t;
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;

typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;

void cpp_ap_int_arith(dinA_t inA,dinB_t inB,dinC_t inC,dinD_t inD,dout1_t
*out1,dout2_t *out2,dout3_t *out3,dout4_t *out4);

#endif

```

### 例 4-57 : C++ 任意精度型を使用した基本演算

例 4-56 が合成されると、例 4-22 および例 4-36 と同じ機能を持つデザインになります。テストベンチをできる限り例 4-36 のようにするには、C++ カウント演算を使用して結果をファイルに出力するよりも、ビルトイン ap\_int メソッドの `.to_int()` を使用して ap\_int の結果を標準 `fprintf` 関数で使用される整数型に変換します。

```

fprintf(fp, "%d*%d=%d; %d+%d=%d; %d/%d=%d; %d mod %d=%d;\n",
 inA.to_int(), inB.to_int(), out1.to_int(),
 inB.to_int(), inA.to_int(), out2.to_int(),
 inC.to_int(), inA.to_int(), out3.to_int(),
 inD.to_int(), inA.to_int(), out4.to_int());

```

注記 : 「[C++ の任意精度型](#)」セクションには、このメソッド、合成動作、`ap_(u)int<N>` 任意精度データ型を使用したすべての側面など、詳細が説明されています。

## C++ の任意精度 (AP) 固定小数点型

C++ 関数を使用すると、Vivado HLS に含まれる任意精度の固定小数点型の利点を生かすことができます。図 4-3 は、これらの固定小数点型の基本的な機能をまとめています。

- ワードは符号付き (`ap_fixed`) または符号なし (`ap_ufixed`) にできます。
- 任意サイズのワード幅  $W$  は定義できます。
- 整数部  $I$  の桁数により、ワード  $W-I$  の整数部も定義されます (図 4-3 では  $B$ )。
- 丸めまたは量子化 ( $Q$ ) のタイプを選択できます。
- オーバーフロー ビヘイビア ( $O$  および  $N$ ) を選択できます。

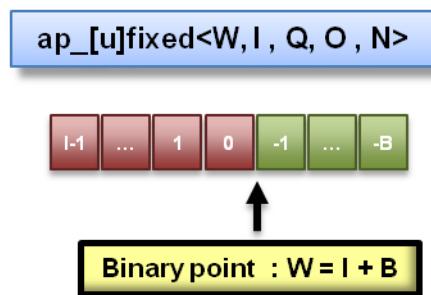


図 4-3 : 任意精度の固定小数点型

任意精度の固定小数点型は、ヘッダーファイル `ap_fixed.h` がコードに含まれていると使用できます。

固定小数点型を使用すると、次のような利点があります。

- 少数を簡単に記述できます。
- 整数部および少数部のビット数が異なる変数の場合、小数点のアライメントが自動的に処理されます。
- 結果を正確に示すだけの少数部ビットが足らない場合のため、多くの丸めの実行を自動的に処理するオプションがあります。
- 結果が整数部ビットよりも大きい場合のため、多くの変数のオーバーフローを自動的に処理するオプションもあります。

これらの属性は、[例 4-58](#) のコードに含まれています。まず、ヘッダーファイル `ap_fixed.h` が含まれ、`ap_fixed` 型が `typedef` 文により定義されます。

- 10 ビット入力 : 8 ビット整数値 + 2 少数部
- 6 ビット入力 : 3 ビット整数値 + 3 少数部
- 累算用の 22 ビット変数 : 17 ビット整数値 + 5 少数部
- 結果用の 36 ビット変数 : 30 ビット整数値 + 6 少数部

関数には、演算実行後の小数点のアライメントを管理するコードは含まれません。これは、自動的に実行されます。

```
#include "ap_fixed.h"

typedef ap_ufixed<10,8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6,3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2) {
 static dint_t sum;
 sum += d_in1;
 return sum * d_in2;
}
```

#### 例 4-58 : AP\_Fixed 小数点の例

[表 2-9](#) は、量子化モードおよびオーバーフロー モードを示しています。詳細は、「C++ の任意精度 (AP) 固定小数点型」を参照してください。



**ヒント** : 標準ハードウェア演算(折り返しおよび切り捨て)のデフォルトビヘイビア以上を実行する量子化モードおよびオーバーフロー モードを使用すると、さらに多くの関連ハードウェアを含む演算子になり、負の無限大や対称的な飽和などのさらにアドバンスなモードをインプリメントするために、ロジック (LUT) が必要になります。

表 2-9 : 固定小数点の識別子

| 識別子    | 説明                                                                       |
|--------|--------------------------------------------------------------------------|
| W      | ワード長をビット数で指定                                                             |
| I      | 整数値をビット数で指定 (整数部のビット数)                                                   |
| Q      | 量子化モードを指定。量子化モードは、結果の保存に使用される変数の最小の小数ビットで定義できるよりも大きい精度が生成された場合の動作を指定します。 |
| モード    | 説明                                                                       |
| AP_RND | 正の無限大への丸め                                                                |

表 2-9: 固定小数点の識別子 (Cont'd)

| 識別子            | 説明                                                                   |              |
|----------------|----------------------------------------------------------------------|--------------|
| AP_RND_ZERO    | 0 への丸め                                                               |              |
| AP_RND_MIN_INF | 負の無限大への丸め                                                            |              |
| AP_RND_INF     | 無限大への丸め                                                              |              |
| AP_RND_CONV    | 収束丸め                                                                 |              |
| AP_TRN         | 負の無限大への切り捨て                                                          |              |
| AP_TRN_ZERO    | 0 への切り捨て (デフォルト)                                                     |              |
| O              | オーバーフロー モードを指定。オーバーフロー モードは、結果の保存に使用される変数よりも多くのビットが生成された場合の動作を指定します。 |              |
|                | モード                                                                  | 説明           |
|                | AP_SAT                                                               | 飽和           |
|                | AP_SAT_ZERO                                                          | 0 への飽和       |
|                | AP_SAT_SYM                                                           | 対称飽和         |
|                | AP_WRAP                                                              | 折り返し (デフォルト) |
|                | AP_WRAP_SM                                                           | 符号絶対値の折り返し   |
| N              | 折り返しモードでの飽和ビット数                                                      |              |

`ap_(u)fixed` 型を使用すると、C++ シミュレーションのビット精度は正しくなり、アルゴリズムとその精度を検証するために高速シミュレーションを使用することができます。合成後、RTL ではまったく同じビット精度の動作になります。

例 4-59 で使用される 例 4-58 のテストベンチに示すように、ビット精度の固定小数点型を使用すると、リテラル値を自由に代入できます。ここでの `in1` および `in2` の値は宣言済みで、定数値が代入されます。

演算子を含むリテラル値を代入する際は、まずリテラル値を `ap_(u)fixed` 型に変換する必要があります。こうしないと、C コンパイラおよび Vivado HLS でこのリテラル値が整数または `float/double` 型として認識され、最適な演算子が検索されなくなります。たとえば、`in1 = in1 + din1_t(0.25)` の代入では、リテラル値 0.25 が `ap_fixed` 型に変換されます。

```

int main()
{
 ofstream result;
 din1_t in1 = 0.25;
 din2_t in2 = 2.125;
 dout_t output;
 int retval=0;

 result.open("result.dat");
 // Persistent manipulators
 result << right << fixed << setbase(10) << setprecision(15);

 for (int i = 0; i <= 250; i++)
 {
 output = cpp_ap_fixed(in1,in2);

 result << setw(10) << i;
 result << setw(20) << in1;
 result << setw(20) << in2;
 result << setw(20) << output;
 }
}

```

```

 result << endl;

 in1 = in1 + din1_t(0.25);
 in2 = in2 - din2_t(0.125);
 }
 result.close();

 // Compare the results file with the golden results
 retval = system("diff --brief -w result.dat result.golden.dat");
 if (retval != 0) {
 printf("Test failed !!!\n");
 retval=1;
 } else {
 printf("Test passed !\n");
 }

 // Return 0 if the test passes
 return retval;
}

```

#### 例 4-59 : AP\_Fixed 小数点のテストベンチ例

## サポートされない C++ コンストラクト

このセクションでは、「サポートされない C コンストラクト」にリストされていない合成不可能な C++ コンストラクトをリストします。

## ダイナミックオブジェクト

C でのダイナミックメモリ使用に関する制限と同様、ダイナミックに作成/削除される C++ オブジェクトも合成でサポートされません。これには、ポリモーフィズム関数およびダイナミック仮想関数の呼び出しが含まれます。次のコードは、ランタイムで新しい関数を作成するので、合成できません。

```

Class A {
public:
 virtual void bar() {...};
};

void fun(A* a) {
 a->bar();
}
A* a = 0;
if (base)
 A= new A();
else
 A = new B();

foo(a);

```

## 標準テンプレートライブラリ

C++ 標準テンプレートライブラリ (STL) には、再帰関数が含まれており、ダイナミックメモリ割り当てが使用されます。そのため、STL は合成できません。STL を使用する場合は、再帰、ダイナミックメモリ割り当て、ダイナミックなオブジェクトの作成/削除などの特性を持たない同じ機能のローカル関数を作成します。

# SystemC の合成

Vivado HLS では、ハードウェア記述に使用される C++ クラス ライブラリである SystemC (IEEE 規格 1666) がサポートされます。これは、[www.systemc.org](http://www.systemc.org) から入手できます。Vivado HLS では SystemC バージョン 2.1 および SystemC Synthesizable Subset (ドラフト 1.3) がサポートされます。

このセクションでは、Vivado HLS を使用して SystemC 関数を合成する際の詳細について説明します。ここに含まれる情報は、前述の「C の合成」および「C++ の合成」の章の情報に記載されなかつたもので、合成の基本的なコード規則を理解するには、これらの章を理解しておく必要があります。



**重要 : C および C++ デザインの場合と同様、合成の最上位関数は C コンパイルの最上位 `sc_main()` の下にある必要があります。** `sc_main()` は合成では最上位関数にはできません。

## デザインの記述

合成の最上位は、`SC_MODULE` である必要があります。SystemC コンストラクタープロセスの `SC_METHOD` および `SC_CTHREAD` を使用して記述される場合、または `SC_MODULES` がほかの `SC_MODULES` 内にインスタンシエートされる場合、デザインは合成できます。

`SC_MODULE` は、別の `SC_MODULE` 内で定義できません (これらは後に示す方法で定義できます)。次の例では、モジュールが別のモジュール内で定義されています。

```
SC_MODULE(nested1)
{
 SC_MODULE(nested2)
 {
 sc_in<int> in0;
 sc_out<int> out0;
 SC_CTOR(nested2)
 {
 SC_METHOD(process);
 sensitive<<in0;
 }
 void process()
 {
 int var =10;
 out0.write(in0.read()+var);
 }
 };

 sc_in<int> in0;
 sc_out<int> out0;
 nested2 nd;
 SC_CTOR(nested1)
 :nd("nested2")
 {
 nd.in0(in0);
 nd.out0(out0);
 }
};
```

次に示すようなバージョンに変換する必要があります。この場合、モジュールはネストされません。

```
SC_MODULE(nested2)
{
```

```

sc_in<int> in0;
sc_out<int> out0;
SC_CTOR(nested2)
{
 SC_METHOD(process);
 sensitive<<in0;
}
void process()
{
 int var = 10;
 out0.write(in0.read() + var);
}
};

SC_MODULE(nested1)
{
 sc_in<int> in0;
 sc_out<int> out0;
 nested2 nd;
 SC_CTOR(nested1)
 :nd("nested2")
 {
 nd.in0(in0);
 nd.out0(out0);
 }
};

```

同様に、SC\_MODULE は別の SC\_MODULE からは派生できません。

```

SC_MODULE(BASE)
{
 sc_in<bool> clock; //clock input
 sc_in<bool> reset;
 SC_CTOR(BASE) {}

};

class DUT: public BASE
{
public:
 sc_in<bool> start;
 sc_in<sc_uint<8> > din;
 ...
};

```

---

 **推奨** : モジュールコンストラクターをモジュール内で定義します。

---

次のような例があるとします。

```

SC_MODULE(dut) {
 sc_in<int> in0;
 sc_out<int> out0;
 SC_HAS_PROCESS(dut);
 dut(sc_module_name nm);
 ...
};

dut::dut(sc_module_name nm)

```

```

 {
 SC_METHOD(process);
 sensitive<<in0;
 }

```

これは次のように変更する必要があります。

```

SC_MODULE(dut) {
 sc_in<int> in0;
 sc_out<int> out0;

 SC_HAS_PROCESS(dut);
 dut(sc_module_name nm)
 : sc_module(nm)
 {
 SC_METHOD(process);
 sensitive<<in0;
 }
 ...
};


```

SC\_THREAD は合成でサポートされません。

## SC\_METHOD の使用

例 4-60 は、SC\_METHOD を使用して半加算器を記述した小規模な組み合わせデザインのヘッダー ファイル sc\_combo\_method.h を示しています。最上位デザイン名の sc\_combo\_method は SC\_MODULE で指定されています。

```

#include <systemc.h>

SC_MODULE(sc_combo_method) {
 //Ports
 sc_in<sc_uint<1>> a,b;
 sc_out<sc_uint<1>> sum,carry;

 //Process Declaration
 void half_adder();

 //Constructor
 SC_CTOR(sc_combo_method) {

 //Process Registration
 SC_METHOD(half_adder);
 sensitive<<a<<b;
 }
};


```

### 例 4-60 : SystemC の組み合わせデザインのヘッダー ファイル

デザインには、a と b の 2 つのシングルビット入力ポートが含まれます。SC\_METHOD はどちらの入力ポートのステートの変更にも影響され、half\_adder 関数を実行します。half\_adder 関数は sc\_combo\_method.cpp ファイルで指定され (例 4-61)、出力ポートのキャリー値を計算します。

```

#include "sc_combo_method.h"

void sc_combo_method::half_adder() {
 bool s,c;

```

```

s=a.read() ^ b.read();
c=a.read() & b.read();
sum.write(s);
carry.write(c);

#ifndef __SYNTHESIS__
 cout << "Sum is " << a << " ^ " << b << " = " << s << ":" <<
 sc_time_stamp() << endl;
 cout << "Car is " << a << " & " << b << " = " << c << ":" <<
 sc_time_stamp() << endl;
#endif

```

#### 例 4-61: SystemC 組み合わせデザインの main 関数

例 4-61 は、`__SYNTHESIS__` マクロを使用して、C シミュレーション中に値を表示するために使用される `cout` 文が合成されないようにする方法を示しています。

例 4-61 のテストベンチは、例 4-62 のようになります。このテストベンチには、Vivado HLS を使用する場合に必要な重要属性が多く含まれています。

```

#ifndef __RTL_SIMULATION__
#include "sc_combo_method_rtl_wrap.h"
#define sc_combo_method sc_combo_method_RTL_transactor
#else
#include "sc_combo_method.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"

int sc_main (int argc , char *argv[])
{
 sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
 sc_report_handler::set_actions(SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
 sc_report_handler::set_actions(SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
 sc_report_handler::set_actions(SC_ID_OBJECT_EXISTS_, SC_LOG);

 sc_signal<bool> s_reset;
 sc_signal<sc_uint<1> > s_a;
 sc_signal<sc_uint<1> > s_b;
 sc_signal<sc_uint<1> > s_sum;
 sc_signal<sc_uint<1> > s_carry;

 // Create a 10ns period clock signal
 sc_clock s_clk("s_clk",10,SC_NS);

 tb_init U_tb_init("U_tb_init");
 sc_combo_method U_dut("U_dut");
 tb_driver U_tb_driver("U_tb_driver");

 // Generate a clock and reset to drive the sim
 U_tb_init.clk(s_clk);
 U_tb_init.reset(s_reset);

 // Connect the DUT
 U_dut.a(s_a);
 U_dut.b(s_b);
 U_dut.sum(s_sum);
 U_dut.carry(s_carry);

```

```

// Drive stimuli from dat* ports
// Capture results at out* ports
U_tb_driver.clk(s_clk);
U_tb_driver.reset(s_reset);
U_tb_driver.dat_a(s_a);
U_tb_driver.dat_b(s_b);
U_tb_driver.out_sum(s_sum);
U_tb_driver.out_carry(s_carry);

// Sim for 200
int end_time = 200;

cout << "INFO:Simulating " << endl;

// start simulation
sc_start(end_time, SC_NS);

if (U_tb_driver.retval != 0) {
 printf("Test failed !!!\n");
} else {
 printf("Test passed !\n");
}
return U_tb_driver.retval;
};

```

#### 例 4-62: SystemC 組み合わせデザインのテストベンチ

Vivado HLS の `cosim_design` 機能を使用して RTL シミュレーションを実行するには、テストベンチに例 4-62 の一番上に示すマクロを含める必要があります。デザインに DUT という名前が付いている場合、次を使用する必要があります。DUT は実際のデザイン名に置き換えられます。

```

#ifndef __RTL_SIMULATION__
#include "DUT_rtl_wrap.h"
#define DUT DUT_RTL_transactor
#else
#include "DUT.h" //Original unmodified code
#endif

```

デザインヘッダーファイルが含まれるテストベンチにこれを追加しない場合、`cosim_design` の RTL シミュレーションでエラーが発生します。

Vivado HLS で使用されるすべての SystemC テストベンチ ファイルに、レポート ハンドラー関数(例 4-62)を追加する必要があります。

```

sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);
sc_report_handler::set_actions(SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions(SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions(SC_ID_OBJECT_EXISTS_, SC_LOG);

```

これらの設定により、RTL シミュレーション中にメッセージが過多に表示されることなくなります。

これらのメッセージの中で最も重要なのは、次の警告文です。

```
Warning: (W212) sc_logic value 'X' cannot be converted to bool
```

合成済みデザイン周辺に配置されるアダプターは、未知の値 (X) で開始されます。すべての SystemC 型で未知の値 (X) がサポートされるわけではありません。この警告メッセージは、デザインに適切なデータ ハンドシェイクが含まれない限り、無視できます。

例 4-62 のテストベンチは、最後に結果のチェックを実行し、結果が正しい場合は値 0 を戻します。この場合、結果は tb\_driver 関数内で検証されますが、戻り値はチェックされた後、最上位テストベンチに戻されます。

```
if (U_tb_driver.retval != 0) {
 printf("Test failed !!!\n");
} else {
 printf("Test passed !\n");
}
return U_tb_driver.retval;
```

## SC\_MODULES のインスタンシエート

SC\_MODULE の階層インスタンシエーションは合成できます (例 4-63) 例 4-63 では、例 4-60 からの半加算器デザイン (sc\_combo\_method) の 2 つのインスタンスが全加算器デザインを作成するためにインスタンシエーションされています。

```
#include <systemc.h>
#include "sc_combo_method.h"

SC_MODULE(sc_hier_inst) {
 //Ports
 sc_in<sc_uint<1> > a, b, carry_in;
 sc_out<sc_uint<1> > sum, carry_out;

 //Variables
 sc_signal<sc_uint<1> > carry1, sum_int, carry2;

 //Process Declaration
 void full_adder();

 //Half-Adder Instances
 sc_combo_methodU_1, U_2;

 //Constructor
 SC_CTOR(sc_hier_inst)
 :U_1("U_1")
 ,U_2("U_2")
 {
 // Half-adder inst 1
 U_1.a(a);
 U_1.b(b);
 U_1.sum(sum_int);
 U_1.carry(carry1);

 // Half-adder inst 2
 U_2.a(sum_int);
 U_2.b(carry_in);
 U_2.sum(sum);
 U_2.carry(carry2);

 //Process Registration
 SC_METHOD(full_adder);
 sensitive<<carry1<<carry2;
 }
};
```

例 4-63 : SystemC 階層デザイン例

full\_adder 関数は、carry\_out 信号のロジックを作成するために使用されています(例 4-64)。

```
#include "sc_hier_inst.h"

void sc_hier_inst::full_adder(){
 carry_out= carry1.read() | carry2.read();
}
```

#### 例 4-64 : SystemC の full\_adder 関数

### SC\_CTHREAD の使用

コンストラクタープロセスの SC\_CTHREAD は、クロック付きプロセス(スレッド)を記述するために使用され、シーケンシャルデザインを記述する主な方法です。例 4-65 では、シーケンシャルデザインの主な属性を示しています。

- データには関連するハンドシェイク信号が含まれるので、合成前後に同じテストベンチを使用してデータが処理されるようにできます。
- クロックの SC\_CTHREAD は、関数が実行されるタイミングを記述するために使用されます。
- SC\_CTHREAD では、リセット動作がサポートされます。

```
#include <systemc.h>

SC_MODULE(sc_sequ_cthread) {
 //Ports
 sc_in <bool> clk;
 sc_in <bool> reset;
 sc_in <bool> start;
 sc_in<sc_uint<16> > a;
 sc_in<bool> en;
 sc_out<sc_uint<16> > sum;
 sc_out<bool> vld;

 //Variables
 sc_uint<16> acc;

 //Process Declaration
 void accum();

 //Constructor
 SC_CTOR(sc_sequ_cthread) {
 //Process Registration
 SC_CTHREAD(accum,clk.pos());
 reset_signal_is(reset,true);
 }
};
```

#### 例 4-65 : SystemC の SC\_CTHREAD

例 4-66 は accum 関数を示しています。この例での重要な点は、次のとおりです。

- コア記述プロセスは、無限の while() ループで、中に wait() 文が含まれます。
- 変数の初期化は、無限の while() ループよりも前に実行されます。このコードは、リセットが SC\_CTHREAD で認識されると実行されます。
- データ読み出しおよび書き込みは、ハンドシェイクプロトコルで確認されます。

```

#include "sc_sequ_cthread.h"

void sc_sequ_cthread::accum() {

 //Initialization
 acc=0;
 sum.write(0);
 vld.write(false);
 wait();

 // Process the data
 while(true) {
 // Wait for start
 while (!start.read()) wait();

 // Read if valid input available
 if (en) {
 acc = acc + a.read();
 sum.write(acc);
 vld.write(true);
 } else {
 vld.write(false);
 }
 wait();
 }
}

```

#### 例 4-66: SystemC の SC\_CTHREAD 関数

### 複数クロックの合成

SystemC では、C および C++ 合成とは異なり、複数クロックを使用するデザインがサポートされます。複数クロックデザインでは、各クロックに関連する機能が SC\_CTHREAD で取り込まれる必要があります。

例 4-67 は、clock および clock2 という名前の 2 つのクロックを含むデザインを示しています。1 つは Prc1 関数を実行する SC\_CTHREAD をアクティベートし、もう 1 つは Prc2 関数を実行する SC\_CTHREAD をアクティベートするためを使用されます。合成後は、Prc1 関数に関連するすべてのシーケンシャルロジックにクロックが付き、clock2 は Prc2 関数のシーケンシャルロジックすべてを駆動します。

```

#include "systemc.h"
#include "tlm.h"
using namespace tlm;

SC_MODULE(sc_multi_clock)
{
 //Ports
 sc_in<bool> clock;
 sc_in<bool> clock2;
 sc_in<bool> reset;
 sc_in<bool> start;
 sc_out<bool> done;
 sc_fifo_out<int> dout;
 sc_fifo_in<int> din;

 //Variables
 int share_mem[100];
 bool write_done;
}

```

```

//Process Declaration
void Prc1();
void Prc2();

//Constructor
SC_CTOR(sc_multi_clock)
{
 //Process Registration
 SC_CTHREAD(Prc1,clock.pos());
 reset_signal_is(reset,true);

 SC_CTHREAD(Prc2,clock2.pos());
 reset_signal_is(reset,true);
}

```

例 4-67: SystemC の複数クロック デザイン

## 最上位 SystemC ポート

SystemC デザインに含まれるポートは、ソース コードで指定されます。SystemC と C/C++ との主な違いは、SystemC の場合、Vivado HLS でサポートされるメモリ インターフェイスでインターフェイス合成しか実行されない点です。最上位インターフェイスのポートはすべて `sc_in_clk`、`sc_in`、`sc_out`、`sc_inout`、`sc_fifo_in`、`sc_fifo_out` または `ap_mem_if` 型にする必要があります。

サポートされるメモリ インターフェイス (`sc_fifo_in`、`sc_fifo_out` および `ap_mem_if`) を除き、デザインとテストベンチ間のすべてのハンドシェイクは SystemC 関数で明示的に記述しておく必要があります。

**注記** : Vivado HLS は、タイミング要件を満たすために必要であれば、SystemC デザインにクロック サイクルを追加することができます。合成後のクロック数は異なる可能性があるので、SystemC デザインではテストベンチを使用してすべてのデータ転送にハンドシェイク信号を付ける必要があります。

TLM 2.0 を使用したトランザクション レベルの記述および記述に基づいたイベントは合成ではサポートされません。

## SystemC のインターフェイス合成

通常、Vivado HLS では SystemC のインターフェイス合成は実行されませんが、前述のとおり、RAM および FIFO ポートなどのメモリ インターフェイスではインターフェイス合成がサポートされることがあります。

### RAM ポートの合成

C および C++ の合成とは異なり、Vivado HLS は配列ポートを自動的に RTL の RAM には変換しません。次の SystemC コードの場合、Vivado HLS の指示子を使用して配列ポートを個々のエレメントに分割する必要があります。こうしないと、このコードは合成できません。

```

SC_MODULE(dut)
{
 sc_in<T> in0[N];
 sc_out<T>out0[N];

 ...
 SC_CTOR(dut)
 {
 ...
 }
}

```

これらの配列を個々のエレメントに分割する指示子は、次のとおりです。

```
set_directive_array_partition dut in0 -type complete
set_directive_array_partition dut out0 -type complete
```

Nが大きな数の場合は、RTL インターフェイスに多くのスカラー ポートが作成されます。

例 4-68 は、RAM インターフェイスが SystemC シミュレーションでどのように記述され、Vivado HLS で完全に合成されるかを示しています。例 4-68 では、配列が RAM ポートに合成可能な ap\_mem\_if 型に置き換えられています。

- ap\_mem\_port 型を使用するには、Vivado HLS インストールディレクトリの include/ap\_systc ディレクトリにある ap\_mem\_if.h ヘッダーファイルを含める必要があります。
  - Vivado HLS 環境では、include/ap\_systc ディレクトリは自動的に含まれます。
- din および dout の配列は ap\_mem\_port 型に置き換えられています (このフィールドについては例 4-68 の後で説明します)。

```
#include "systemc.h"
#include "ap_mem_if.h"

SC_MODULE(sc_RAM_port)
{
 //Ports
 sc_in <bool> clock;
 sc_in <bool> reset;
 sc_in <bool> start;
 sc_out<bool> done;
 //sc_out<int> dout[100];
 //sc_in<int> din[100];
 ap_mem_port<int, int, 100, RAM2P> dout;
 ap_mem_port<int, int, 100, RAM2P> din;

 //Variables
 int share_mem[100];
 sc_signal<bool> write_done;

 //Process Declaration
 void Prc1();
 void Prc2();

 //Constructor
 SC_CTOR(sc_RAM_port)
 : dout ("dout"),
 din ("din")
 {
 //Process Registration
 SC_CTHREAD(Prc1,clock.pos());
 reset_signal_is(reset,true);

 SC_CTHREAD(Prc2,clock.pos());
 reset_signal_is(reset,true);
 }
};
```

#### 例 4-68 : SystemC の RAM インターフェイス

ap\_mem\_port 型のフォーマットは、次のとおりです。

```
ap_mem_port (<data_type>, < address_type>, <number_of_elements>, <Mem_Target>)
```

- `data_type` は格納されたデータ エレメントに使用されるデータ型です。[例 4-68](#) では、これらは標準の `int` 型です。
- `address_type` はアドレス バスに使用されるデータ型です。このデータ型には、配列内のすべてのエレメントをアドレス指定するのに十分なデータビットが含まれる必要があります。含まれない場合、C シミュレーションでエラーになります。
- `number_of_elements` は記述される配列に含まれるエレメント数を指定します。
- `Mem_Target` は、このポートを接続するメモリを指定するので、最終 RTL の IO ポートを決定します。使用可能なターゲットのリストは、[表 2-10](#) を参照してください。

[表 2-10](#) のメモリ ターゲットは、合成で作成されるポートとデザイン内での操作のスケジュール方法に影響します。たとえば、デュアル ポート RAM の IO ポートはシングル ポート RAM の 2 倍になり、内部操作が並列で実行されるようにスケジュールできることがあります（ループのようなコード コンストラクトとデータ依存性で許容される場合のみ）。

表 2-10 : SystemC の `ap_mem_port` のメモリ ターゲット

| ターゲット<br>RAM | 説明                                               |
|--------------|--------------------------------------------------|
| RAM1P        | シングル ポート RAM                                     |
| RAM2P        | デュアル ポート RAM                                     |
| RAMT2P       | 入力および出力の両方で読み出しポートと書き込みポートの両方をサポートするデュアル ポート RAM |
| ROM1P        | シングル ポート ROM                                     |
| ROM2P        | デュアル ポート ROM                                     |

`ap_mem_port` がインターフェイスで定義されると、変数はほかの配列と同じ方法でコードから単にアクセスされるようになります。

```
dout[i] = share_mem[i] + din[i];
```

[例 4-68](#) をサポートするテストベンチは、[例 4-69](#) のようになります。`ap_mem_port` 型はテストベンチの `ap_mem_chn` 型でサポートされる必要があります。`ap_mem_chn` 型は `ap_mem_if.h` ヘッダーファイルで定義されます。サポートされるフィールドは、`ap_mem_port` と同じです。

```
#ifndef __RTL_SIMULATION__
#include "sc_RAM_port_rtl_wrap.h"
#define sc_RAM_port sc_RAM_port_RTL_transactor
#else
#include "sc_RAM_port.h"
#endif
#include "tb_init.h"
#include "tb_driver.h"
#include "ap_mem_if.h"

int sc_main (int argc , char *argv[])
{
 sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO NOTHING);
 sc_report_handler::set_actions(SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
 sc_report_handler::set_actions(SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
 sc_report_handler::set_actions(SC_ID_OBJECT_EXISTS_, SC_LOG);

 sc_signal<bool> s_reset;
 sc_signal<bool> s_start;
 sc_signal<bool> s_done;
 ap_mem_chn<int,int, 100, RAM2P> dout;
 ap_mem_chn<int,int, 100, RAM2P> din;
```

```

// Create a 10ns period clock signal
sc_clock s_clk("s_clk",10,SC_NS);

tb_init U_tb_init("U_tb_init");
sc_RAM_port U_dut("U_dut");
tb_driver U_tb_driver("U_tb_driver");

// Generate a clock and reset to drive the sim
U_tb_init.clk(s_clk);
U_tb_init.reset(s_reset);
U_tb_init.done(s_done);
U_tb_init.start(s_start);

// Connect the DUT
U_dut.clock(s_clk);
U_dut.reset(s_reset);
U_dut.done(s_done);
U_dut.start(s_start);
U_dut.dout(dout);
U_dut.din(din);

// Drive inputs and Capture outputs
U_tb_driver.clk(s_clk);
U_tb_driver.reset(s_reset);
U_tb_driver.start(s_start);
U_tb_driver.done(s_done);
U_tb_driver.dout(dout);
U_tb_driver.din(din);

// Sim
int end_time = 1100;

cout << "INFO:Simulating " << endl;

// start simulation
sc_start(end_time, SC_NS);

if (U_tb_driver.retval != 0) {
 printf("Test failed !!!\n");
} else {
 printf("Test passed !\n");
}
return U_tb_driver.retval;
};

```

#### 例 4-69 : SystemC の RAM インターフェイスのテストベンチ

#### FIFO ポートの合成

最上位インターフェイスの FIFO ポートは、標準 SystemC の `sc_fifo_in` および `sc_fifo_out` ポートから直接合成できます。次の例 4-70 は、インターフェイスで FIFO ポートを使用する例を示しています。

合成後、各 FIFO ポートにはデータポートおよび関連する FIFO 制御信号(入力には `empty` および `read` ポート、出力には `full` および `write` ポート)が含まれます。FIFO を使用すると、データ転送を同期するのに必要なハンドシェイクが自動的に RTL テストベンチに追加されます。

```

#include "systemc.h"
#include "tlm.h"

```

```

using namespace tlm;

SC_MODULE(sc_FIFO_port)
{
 //Ports
 sc_in <bool> clock;
 sc_in <bool> reset;
 sc_in <bool> start;
 sc_out<bool> done;
 sc_fifo_out<int> dout;
 sc_fifo_in<int> din;

 //Variables
 int share_mem[100];
 bool write_done;

 //Process Declaration
 void Prc1();
 void Prc2();

 //Constructor
 SC_CTOR(sc_FIFO_port)
 {
 //Process Registration
 SC_CTHREAD(Prc1,clock.pos());
 reset_signal_is(reset,true);

 SC_CTHREAD(Prc2,clock.pos());
 reset_signal_is(reset,true);
 }
}

```

例 4-70: SystemC の FIFO インターフェイス

## サポートされない SystemC コンストラクト

### モジュールおよびコンストラクター

前述した点をここでも参照のため繰り返します。次はサポートされません。

- SC\_MODULE は別の SC\_MODULE 内にはネストできません。
- SC\_MODULE は別の SC\_MODULE からは派生できません。
- SC\_THREAD はサポートされていません (クロック付きバージョンの SC\_CTHREAD はサポートされます)。

### モジュールのインスタンシエーション

SC\_MODULE は new を使用してインスタンシエートできません。SC\_MODULE(TOP)

```

{
 sc_in<T> din;
 sc_out<T> dout;

 M1 *t0;

 SC_CTOR(TOP) {

```

```

 t0 = new M1("t0");
 t0->din(din);
 t0->dout(dout);
 }
}

```

これは次のように変更する必要があります。

```

SC_MODULE(TOP)
{
 sc_in<T> din;
 sc_out<T> dout;

 M1 t0;

 SC_CTOR(TOP)
 : t0("t0")
 {
 t0.din(din);
 t0.dout(dout);
 }
}

```

## モジュールのコンストラクター

モジュールコンストラクターと一緒に使用できるのは name パラメーターのみです。データ型 int の変数 temp には次を渡すことができません。

```

SC_MODULE(dut) {
 sc_in<int> in0;
 sc_out<int> out0;
 int var;
 SC_HAS_PROCESS(dut);
 dut(sc_module_name nm, int temp)
:sc_module(nm), var(temp)
 { ... }
};

```

## 関数

仮想関数はサポートされません。次のコードは、仮想関数が使用されているので合成できません。

```

SC_MODULE(DUT)
{
 sc_in<int> in0;
 sc_out<int> out0;

 virtual int foo(int var1)
 {
 return var1+10;
 }

 void process()
 {
 int var=foo(in0.read());
 out0.write(var);
 }
 ...
};

```

## 最上位インターフェイスポート

`sc_out` ポートの読み出しありはサポートされません。次の例は、`out0` に読み出しがあるため、使用できません。

```
SC_MODULE (DUT)
{
 sc_in<T> in0;
 sc_out<T>out0;
 ...
 void process()
 {
 int var=in0.read()+out0.read();
 out0.write(var);
 }
};
```

## C の任意精度型

このセクションでは、C 言語デザイン用に Vivado HLS で提供される任意精度(AP)型について明します。サブセクションでは、C の `int#W` 型用の関連関数について説明します。



**重要:** `[u]int#W` 型が使用されている場合は、この型が正しくシミュレーションされるようにするために、プロジェクト設定で `apcc` オプションを選択する必要があります。この型のファンクションはデバッガーでは解析することはできません。

### [u]int#W 型のコンパイル

`[u]int#W` 型を使用するには、`[u]int#W` 変数を参照するソース ファイルすべてに `ap_cint.h` ヘッダー ファイルを含める必要があります。

この型を使用するソフトウェア モデルをコンパイルするとき、Vivado HLS のヘッダー ファイルの場所を指定する必要がある場合があります。たとえば、`gcc` コンパイルには `-I/<HLS_HOME>/include` オプションを追加するなどします。

`gcc -O3` オプションを使用してコンパイルする場合には、ソフトウェア モデルでベスト パフォーマンスが見られます。

### [u]int#W 変数の宣言/定義

C 型には次のようにそれぞれ符号付きおよび符号なしがあります。

- `int#W`
- `uint#W`

「#W」は数値で、宣言されている変数の合計幅を指定します。

次の例にあるように、C/C++ の `typedef` 文を使用してユーザー定義型を作成することができます。

```
include "ap_cint.h" // use [u]int#W types

typedef uint128 uint128_t; // 128-bit user defined type
int96 my_wide_var; // a global variable declaration
```

幅の最大値は 1024 ビットです。

## 定数(リテラル)からの初期化および代入

[u] int#W 変数は、ネイティブ整数データ型でサポートされているのと同じ整数定数で初期化することができます。定数は [u] int#W 変数の最大幅までゼロまたは符号拡張されます。

```
#include "ap_cint.h"

uint15 a = 0;
uint52 b = 1234567890U;
uint52 c = 0o12345670UL;
uint96 d = 0x123456789ABCDEFULL;
```

64 ビットよりも大きなビット幅には、次のファンクションを使用することができます。

### apint\_string2bits()

このセクションでも、関連する関数の使用について説明します。

- apint\_string2bits\_bin()
- apint\_string2bits\_oct()
- apint\_string2bits\_hex()

これらの関数は基數(10 進数、2 進数、8 進数、16 進数)の制約内で指定された桁の定数文字列を指定したビット幅 N の付いた値に変換します。どの基數の場合も、数値には負の値を示すためにマイナス記号 (-) を付けることができます。

```
int#Napint_string2bits[_radix] (const char*, int N)
```

これは、C 言語で許容される値よりも大きな値で整数定数をコンストラクトするのに使用されます。小さな値でも問題はありません。既存の C 言語の定数値で指定するのに簡単な方を使用します。

```
#include <stdio.h>
#include "ap_cint.h"

int128 a;

// Set a to the value hex 0000000000000000123456789ABCDEF0
a = a-apint_string2bits_hex("-123456789ABCDEF", 128);
```

また、値は文字列から直接代入することができます。

### apint\_vstring2bits()

この関数は 16 進数の制約内で指定された桁の文字列を指定したビット幅 N の付いた値に変換します。数値には負の値を示すためにマイナス記号 (-) を付けることができます。

これは、C 言語で許容される値よりも大きな値で整数定数をコンストラクトするのに使用されます。このファンクションは通常、ファイルから情報を読み出すため、テストベンチで使用されます。

ファイル `test.dat` には次のデータが含まれているとします。

```
123456789ABCDEF
-123456789ABCDEF
-5
```

テストベンチで使用されている場合、この関数で次の値が output されます。

```
#include <stdio.h>
#include "ap_cint.h"
```

```

typedef data_t;

int128 test (
 int128 t a
) {
 return a+1;
}

int main () {
 FILE *fp;
 char vstring[33];

 fp = fopen("test.dat", "r");

 while (fscanf(fp, "%s", vstring)==1) {

 // Supply function "test" with the following values
 // 0000000000000000123456789ABCDF0
 // FFFFFFFFFFFFFFEDCBA9876543212
 // FFFFFFFFFFFFFF9999999999999999FC

 test(apint_vstring2bits_hex(vstring,128));
 printf("\n");
 }

 fclose(fp);
 return 0;
}

```

## コンソール I/O (出力) のサポート

[u] int#W 変数は、ネイティブ整数データ型でサポートされているのと同じ変換指定子で出力することができます。変換指定子に基づいてフィットするビットのみが出力されます。

```

#include "ap_cint.h"

uint164 c = 0x123456789ABCDEFULL;

printf(" d%40d\n",c); // Signed integer in decimal format
// d -1985229329
printf(" hd%40hd\n",c); // Short integer
// hd -12817
printf(" ld%40ld\n",c); // Long integer
// ld 81985529216486895
printf(" lld%40lld\n",c); // Long long integer
// lld 81985529216486895

printf(" u%40u\n",c); // Unsigned integer in decimal format
// u 2309737967
printf(" hu%40hu\n",c); // hu
// hu 52719
printf(" lu%40lu\n",c); // lu
// lu 81985529216486895
printf(" llu%40llu\n",c); // llu
// llu 81985529216486895

printf(" o%40o\n",c); // Unsigned integer in octal format
// o 21152746757

```

```
printf(" ho%40ho\n",c); // ho
printf(" lo%40lo\n",c); 146757
// lo
printf("llo%40llo\n",c); 4432126361152746757
// llo
// llo

printf(" x%40x\n",c); // Unsigned integer in hexadecimal format [0-9a-f]
// x
printf(" hx%40hx\n",c); 89abcdef
// hx
printf(" lx%40lx\n",c); cdef
// lx
printf("llx%40llx\n",c); 123456789abcdef
// llx
// llx

printf(" X%40X\n",c); // Unsigned integer in hexadecimal format [0-9A-F]
// X
{ 89ABCDEF
```

[u] int#W 変数の初期化および代入の場合と同様に、64 ビットよりも大きな値の出力をサポートするための機能が提供されています。

## apint\_print()

C言語で許容される値よりも大きな値で整数を出力するのに使用されます。この関数は、基数(2、8、10、16)に基づいて処理された値を `stdout` に出力します。

```
void apint print(int#N value, int radix)
```

apint printf() が使用されているときの結果値の例は次のようにになります。

## apint\_fprint()

C言語で許容される値よりも大きな値で整数を出力するのに使用されます。このファンクションは、基數(2、8、10、16)に基づいて処理された値をファイルに出力します。

```
void apint_fprint(FILE* file, int#N value, int radix)
```

## [u]int#W 型を使用した式

[u] int#W 型の変数は、C 演算子を使用した式ではほぼ自由に使用することができます。しかし、一部には予期しない動作が見られるものもあるため説明が必要になります。

## ビット幅が小さい値を幅が大きいものへ代入する場合のゼロまたは符号拡張

ビット幅が小さい符号付き変数の値を幅の大きなものへ代入すると、符号に関わらず、値はデスティネーション変数の幅(大きいほうの幅)に符号拡張されます。

同様に、符号なしのビット幅の小さい変数は代入前にゼロ拡張されます。

代入で予期動作を得るには、ソース変数の明示的な型変換が必要になることがあります。

## ビット幅が大きい値を幅が小さいものへ代入する場合の切り捨て

ビット幅が大きい値を幅の小さなものへ代入すると、デスティネーション値(幅の小さいほうの値)の最上位ビット(MSB)を超えたビットはすべて切り捨てられます。

この切り捨てが行われるとき、符号情報が特別に処理されるわけではないので、予期しない動作が見られる可能性があります。予期しない動作を防ぐには、明示的な型変換を利用します。

## 2進数の演算子

通常、ネイティブのC整数データ型で実行可能な有効な演算は [u] int#W型でサポートされています。

任意精度演算を行うため、標準2進数整数演算子がオーバーロードされます。次の演算子ではすべて、[u] int#Wの2つのオペランドが使用されるか、または [u] int#W型を1つとC/C++整数データ型1つ(char, short, intなど)が使用されます。

結果値の幅および符号は、デスティネーション変数(または式)の幅に基づいて符号拡張、ゼロの追加、または切り捨てが実行される前に、オペランドの幅および符号で決まります。返される値の詳細は各演算子で説明されています。

式に ap\_[u] int型およびC/C++整数型の両方が含まれている場合、C++型では次の幅が使用されます。

- char: 8ビット
- short: 16ビット
- int: 32ビット
- long: 32ビット
- long long: 64ビット

### 加算

```
[u] int#W::RTType [u] int#W::operator + ([u] int#W op)
```

この式は、2つのap\_[u]int(またはap\_[u]int1つとC/C++整数型を1つ)の和を計算します。

和の幅は、オペランドの幅の大きいほうよりも1ビット大きくなります(幅の広いほうが符号なしで幅の小さいほうが符号付きの場合のみ2ビット大きくなる)。

オペランドのいずれか(または両方)が符号付きであれば、和は符号付きとして処理されます。

### 減算

```
[u] int#W::RTType [u] int#W::operator - ([u] int#W op)
```

2つの整数の差を計算する式です。

差の幅は、オペランドの幅の大きいほうよりも1ビット大きくなります(幅の広いほうが符号なしで幅の小さいほうが符号付きの場合のみ2ビット大きくなる)。代入前に、デスティネーション変数の幅に基づいて、符号拡張、ゼロの追加、または切り捨てが実行されます。

オペランドの符号に関わらず、差は符号付きとして処理されます。

## 乗算

```
[u] int#W::RTType [u] int#W::operator * ([u] int#W op)
```

2 つの整数値の積を計算します。

積の幅はオペランドの幅の合計です。

オペランドのいずれかが符号付きであれば、積は符号付きとして処理されます。

## 除算

```
[u] int#W::RTType [u] int#W::operator / ([u] int#W op)
```

2 つの整数値の積を計算します。

除数が符号なしである場合、商の幅は被除数の幅となり、そうでない場合は、商の幅は被除数の幅に 1 を足したものとなります。

オペランドのいずれかが符号付きであれば、商は符号付きとして処理されます。

**注記** : 除算演算子を Vivado HLS で合成すると、生成された RTL に適切にパラメーター設定されたザイリンクス LogiCORE™ IP の除算コアがインスタンシエートされます。

## 剰余

```
[u] int#W::RTType [u] int#W::operator % ([u] int#W op)
```

2 つの整数値の整数除算の余りを計算します。

オペランドがどちらも同じ符号である場合は、剰余の幅はオペランドの幅の最小値となります。除数が符号なし、被除数が符号付きの場合は、剰余の幅は除数の幅に 1 を足したものとなります。

商の符号は被除数のものと同じものとして処理されます。

**注記** : 剰余 (%) 演算子を Vivado HLS で合成すると、生成された RTL で適宜パラメーター設定されたザイリンクス LogiCORE 除算コアがインスタンシエートされます。

## ビット単位の論理演算子

ビット単位の論理演算子はすべて、2 つのオペランドの幅の最大値となる幅の値を返し、両方のオペランドが符号なしの場合にのみ符号なしとして処理され、そうでない場合は、符号付きとして処理されます。

符号拡張(またはゼロの追加)は、デスティネーション変数ではなく、式の符号に基づいて実行されます。

### ビット単位の OR

```
[u] int#W::RTType [u] int#W::operator | ([u] int#W op)
```

2 つのオペランドのビット単位の OR 値を返します。

### ビット単位の AND

```
[u] int#W::RTType [u] int#W::operator & ([u] int#W op)
```

2 つのオペランドのビット単位の AND 値を返します。

### ビット単位の XOR

```
[u] int#W::RTType [u] int#W::operator ^ ([u] int#W op)
```

2 つのオペランドのビット単位の XOR 値を返します。

## シフト演算子

各シフト演算子には 2 つのバージョンがあり、1 つは符号なしの右辺 (RHS) オペランド、もう 1 つは符号付きの RHS です。

符号付き RHS に負の値が与えられるとシフト演算の方向を逆にします。たとえば、RHS オペランドの絶対値によるシフトが逆の方向で発生します。

シフト演算子は、左辺 (LHS) オペランドと同じ幅の値を返します。C/C++ の場合と同じように、右シフトの LHS オペランドが符号付きの場合、符号ビットは、LHS オペランドの符号を維持しつつ、最上位ビットにコピーされます。

### 符号なし整数右シフト

```
[u] int#W [u] int#W::operator << (ap_uint<int_W2> op)
```

### 整数右シフト

```
[u] int#W [u] int#W::operator << (ap_int<int_W2> op)
```

### 符号なし整数左シフト

```
[u] int#W [u] int#W::operator >> (ap_uint<int_W2> op)
```

### 整数左シフト

```
[u] int#W [u] int#W::operator >> (ap_int<int_W2> op)
```

左シフト演算子の結果を幅が広いほうのデスティネーション変数に代入すると、情報の一部またはすべてが失われる場合がありますので注意してください。予期しない動作を防ぐには、シフト式を代入されるほうの型に明示的に型変換するようしてください。

## 複合代入演算子

次の複合演算子がサポートされています。

**\*=    /=    %=    +=    -=    <<=    >>=    &=    ^=    |=**

RHS 式はまず求めてから、基本演算子に RHS オペランドとして与えられ、LHS 変数に結果が代入されます。式の長さ、符号、符号拡張か切り捨てかのルールは、関連する演算に対し上記で説明されているように、適用されます。

## 関係演算子

すべての関係演算子がサポートされており、比較結果に基づいてブール形式の値が返されます。ap\_[u] int 型の変数は、これらの演算子を使用して C/C++ 基本整数型に比較することができます。

### 等号

```
bool [u] int#W::operator == ([u] int#W op)
```

### 等号否定

```
bool [u] int#W::operator != ([u] int#W op)
```

### 小なり

```
bool [u] int#W::operator < ([u] int#W op)
```

## 大なり

```
bool [u] int#W::operator > ([u] int#W op)
```

## 小なりイコール

```
bool [u] int#W::operator <= ([u] int#W op)
```

## 大なりイコール

```
bool [u] int#W::operator >= ([u] int#W op)
```

# ビット レベル演算 : サポートされる関数

[u] int#W 型を使用すると変数をビット レベルの精度で計算することができます。ビット レベル演算を実行するには(ハードウェア)アルゴリズムを使用するのがよいでしょう。Vivado HLS では次のような関数が提供されています。

## ビット操作

ap\_int[u] int 型変数に格納されている値に基づいて一般的なビット レベル演算を行うために次のメソッドが提供されています。

### 長さ

*apint\_bitwidthof()*

```
int apint_bitwidthof(type_or_value)
```

この関数は、ビット数を表す任意精度の整数値を返し、任意の型または値で使用することができます。

```
int5 Var1, Res1;

Var1= -1;
Res1 = apint_bitwidthof(Var1); // Res1 is assigned 5
Res1 = apint_and_reduce(int7); // Res1 is assigned 7
```

### 連結

*apint\_concatenate()*

```
int#(N+M) apint_concatenate(int#N first, int#M second)
```

2 つの [u] int#W 変数を連結します。返された値の幅はオペランドの幅の和です。

引数の高い値および低い値は、それぞれ結果の高位および低位ビットになります。

整数リテラルを含むネイティブ C 型は、予期しない結果を避けるため、連結前に該当する [u] int#W 型に明示的にキャストする必要があります。

## ビット選択

*apint\_get\_bit()*

```
int apint_get_bit(int#N source, int index)
```

任意精度の整数値から 1 ビット選択し、それを返します。

ソースは [u] int#W 型であり、指數引数は整数値である必要があります。選択するビットの指數を指定し、最下位ビットは指數 0 になります。最大指數はこの [u] int#W のビット幅から 1 を引いた値になります。

## ビット値の設定

```
apint_set_bit()
int#N apint_set_bit(int#N source, int index, int value)
```

[u] int#W インスタンス ソースの指定ビット、指数、を指定値(0 または 1)に設定します。

## 範囲選択

```
apint_get_range()
int#N apint_get_range(int#N source, int high, int low)
```

引数で指定されるビットの範囲で表される値を返します。

引数 high は、範囲内の最上位ビット (MSB) を、low は最下位ビット (LSB) を指定します。

ソース変数の LSB は 0 の位置にあります。引数 high の値が low のものより小さい場合、ビットは逆順で返されます。

## 範囲値設定

```
apint_set_range()
int#N apint_set_range(int#N source, int high, int low, int#M part)
```

引数 high、low で指定されるソースの範囲の指定ビットを part の値に設定します。

## ビット低減

### AND を使用した低減

```
apint_and_reduce()
int apint_and_reduce(int#N value)
```

値のすべてのビットに対し AND 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。

```
int5 Var1, Res1;
Var1= -1;
Res1 = apint_and_reduce(Var1); // Res1 is assigned 1
Var1= 1;
Res1 = apint_and_reduce(Var1); // Res1 is assigned 0
```

この演算は、-1 と比較して一致すれば 1 を返し、一致しなければ 0 を返します。すべてのビットが 1 であることをチェックする方法もあります。

### OR を使用した低減

```
apint_or_reduce()
int apint_or_reduce(int#N value)
```

値のすべてのビットに対し OR 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。この演算は、0 と比較して一致すれば 0 を返し、一致しなければ 1 を返します。

```
int5 Var1, Res1;
Var1= 1;
```

```
Res1 = apint_or_reduce(Var1); // Res1 is assigned 1

Var1= 0;
Res1 = apint_or_reduce(Var1); // Res1 is assigned 0
```

### XOR を使用した低減

```
apint_xor_reduce()

int apint_xor_reduce(int#N value)
```

値のすべてのビットに対し OR 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。この演算は、ワードの 1 をカウントし、偶数があれば 1 を返し、奇数(偶数パリティ)があれば 0 を返します。

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_xor_reduce(Var1); // Res1 is assigned 0

Var1= 0;
Res1 = apint_xor_reduce(Var1); // Res1 is assigned 1
```

### NAND を使用した低減

```
apint_nand_reduce()

int apint_nand_reduce(int#N value)
```

値のすべてのビットに対し NAND 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。この値を -1(すべて 1) と比較し、一致すれば false、そうでなければ true を返します。

```
int5 Var1, Res1;

Var1= 1;
Res1 = apint_nand_reduce(Var1); // Res1 is assigned 1

Var1= -1;
Res1 = apint_nand_reduce(Var1); // Res1 is assigned 0
```

### NOR を使用した低減

```
apint_nor_reduce()

int apint_nor_reduce(int#N value)
```

値のすべてのビットに対し NOR 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。この値を 0(すべて 0) と比較し、一致すれば true、そうでなければ false を返します。

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_nor_reduce(Var1); // Res1 is assigned 1

Var1= 1;
Res1 = apint_nor_reduce(Var1); // Res1 is assigned 0
```

### XNOR を使用した低減

```
apint_xnor_reduce()
```

```

int apint_xnor_reduce(int#N value)

値のすべてのビットに対し XNOR 演算を行い、整数値として 1 ビットの結果値を返します(ブールに型変換可能)。この演算は、ワードの 1 をカウントし、奇数があれば 1 を返し、偶数(偶数パリティ)があれば 0 を返します。

int5 Var1, Res1;

Var1= 0;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 0

Var1= 1;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 1

```

## C++ の任意精度型

Vivado HLS では、ソフトウェアとハードウェア モデリング間での一貫した、ビット精度の動作で、任意精度(またはビット精度)の整数データ型をインプリメントする `ap_[u]int<>` という C++ のテンプレート クラスが提供されています。

このクラスは、ネイティブ C 整数型で使用可能なすべての四則演算、ビット単位、論理、関係演算子を提供しています。さらに、64 ビットよりも幅が広い変数の初期化および変換を可能にするハードウェア演算の一部を処理するためのメソッドも、このクラスで提供しています。演算子およびクラス メソッドの詳細は次を参照してください。

### ap\_[u]<> 型のコンパイル

`ap_[u]fixed<>` クラスを使用するには、`ap_[u]fixed<>` 変数を参照するソース ファイルすべてに `ap_int.h` ヘッダーファイルを含める必要があります。

これらのクラスを使用するソフトウェア モデルをコンパイルするとき、Vivado HLS のヘッダーファイルの場所を指定する必要がある場合があります。たとえば、`gcc` コンパイルには `-I/<HLS_HOME>/include` オプションを追加するなどします。

`g++ -O3` オプションを使用してコンパイルする場合には、ソフトウェア モデルでベスト パフォーマンスが見られます。

### ap\_[u] 変数の宣言/定義

クラスには、符号付きの `ap_int<int_W>` および 符号なしの `ap_uint<int_W>` があります。テンプレート パラメーターの `int_W` は宣言されている変数の合計幅を指定します。

次の例にあるように、C/C++ の `typedef` 文を使用してユーザー定義型を作成することができます。

```

#include "ap_int.h" // use ap_[u]fixed<> types
typedef ap_uint<128> uint128_t; // 128-bit user defined type
ap_int<96> my_wide_var; // a global variable declaration

```

幅のデフォルト最大値は 1024 ビットです。このデフォルトは、`ap_int.h` ヘッダーファイルを含める前に、32768 以下の正の整数値でマクロ `AP_INT_MAX_W` を定義すると上書きすることができます。



**注意:** `AP_INT_MAX_W` の値をあまり高く設定すると、ソフトウェアのコンパイルおよび実行に時間がかかる可能性があります。

`AP_INT_MAX_W` の上書きの例は次のとおりです。

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"

ap_int<2048> very_wide_var;
```

## 定数(リテラル)からの初期化および代入

クラス コンストラクタおよび代入演算子のオーバーロードを利用し、標準 C/C++ の整数リテラルを使用して `ap_[u]fixed<>` 変数を初期化および代入することができます。

しかし、この `ap_[u]fixed<>` 変数への値の代入方法は、C++ およびソフトウェアが実行されるシステムの制限の対象となり、通常、整数リテラルの 64 ビット制限(LL または ULL の接頭辞など)の影響を受けます。

`ap_[u]fixed<>` クラスでは、任意の長さ(変数の幅以下のもの)の文字列から初期化が可能なコンストラクタが提供されています。

デフォルトでは、文字列に有効な 16 進数の 0 から 9 までの数字および a から f の文字のみが含まれている限り、文字列は 16 進数値として処理されます。そのような文字列から値を代入するには、文字列を該当する型へと、明示的に C++ 形式の型変換を行う必要があります。

64 ビットよりも大きな値を含む初期化および代入の例は次のとおりです。

```
ap_int<42> a_42b_var(-1424692392255LL); // long long decimal format
a_42b_var = 0x14BB648B13FLL; // hexadecimal format

a_42b_var = -1; // negative int literal sign-extended to full width

ap_uint<96> wide_var("76543210fedcba9876543210"); // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567");
```

`ap_[u]>>` コンストラクタは、基数 2、8、10、または 16 で数値を表わして文字を処理するように明示的に指定することができます。コンストラクタの呼び出しに 2 番目のパラメーターとして該当する基数値を追加すると、この設定ができます。

指定されている基数に対して無効な文字が文字リテラルに含まれていると、コンパイル エラーが発生します。

それぞれの基数フォーマットの例は次のとおりです。

```
ap_int<6> a_6bit_var("101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("55", 10); // decimal format
a_6bit_var = ap_int<6>("2A", 16); // 42d in hexadecimal format

a_6bit_var = ap_int<6>("42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

文字列でエンコードされている基数は、「0」の後に「b」、「o」、または「x」を続けた接頭辞を付けると、コンストラクタで自動推論することもできます。「0b」、「0o」、および「0x」という接頭辞はそれぞれ 2 進数、8 進数、16 進数のフォーマットに対応しています。

この初期化文字列フォーマットを使用した例は次のとおりです。

```
ap_int<6> a_6bit_var("0b101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("0o40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("0x2A", 16); // 42d in hexidecimal format

a_6bit_var = ap_int<6>("0b42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

## コンソール I/O (出力) のサポート

`ap_[u]fixed<>` 変数の初期化および代入の場合と同様に、64 ビットよりも大きな値の出力をサポートするための機能が提供されています。

`ap_[u]int` 変数に格納されている値を出力するのに最も簡単な方法は、C++ の標準出力ストリーム `std::cout` (`#include <iostream>` または `<iostream.h>`) を使用する方法です。ストリーム挿入演算子 `<<` は、任意の `ap_[u]int` 変数に対する範囲全体に含まれる値が正しく出力されるように、オーバーロードされます。ストリームマニピュレーターの `dec`、`hex` および `oct` もサポートされていて、それぞれ 10 進数、16 進数、または 8 進数で値をフォーマットすることができます。

値の出力に `cout` を使用した例は次のとおりです。

```
#include <iostream.h>
// Alternative:#include <iostream>

ap_uint<72> Val("10fedcba9876543210");

cout << Val << endl; // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

標準 C ライブライ ( `#include <stdio.h>` ) を使用して、64 ビットよりも大きな値を出力することもできます。まず、値を C++ の `std::string` に変換してから、C の文字列に変換します。`ap_[u]int` クラスでは、最初の変換を行うのに `to_string()` メソッドが提供されており、`std::string` クラスには、ヌルで終了する文字列に変換するために `c_str()` メソッドがあります。

`ap[u]int::to_string()` メソッドは目的の基数を指定するオプションの引数を渡すことができます。この基数引数に有効な値は 2、8、10、および 16 で、それぞれ 2 進数、8 進数、10 進数、16 進数を指定します。デフォルト値は 16 です。

`ap_[u]int::to_string()` の 2 つ目のオプションの引数は、10 進数以外のフォーマットで符号付きの値を出力するかどうかを指定します。この引数はブール形式です。デフォルト値は `false` で、10 進数以外のフォーマットで符号なしの値が出力されます。

値の出力に `printf` を使用した例は次のとおりです。

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str()); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str()); // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str()); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str()); // => "-7F0123456789ABCDF0"
```

## ap\_[u]<> 型を使用した式

`ap_[u]<>` 型の変数は、C/C++ 演算子を使用した式ではほぼ自由に使用することができます。しかし、一部には予期しない動作が見られるものもあるため説明が必要になります。

### ビット幅が小さい値を幅が大きいものへ代入する場合のゼロまたは符号拡張

ビット幅が小さい符号付き (`ap_int<>`) 変数の値を幅の大きなものへ代入すると、符号に関わらず、値はデステイネーション変数 (幅の大きい変数) の幅に符号拡張されます。

同様に、符号なしのビット幅の小さい変数は代入前にゼロ拡張されます。

代入で予期動作を得るには、次に示すように、ソース変数の明示的な型変換が必要になることがあります。

```
ap_uint<10> Result;
```

```

ap_int<7> Val1 = 0x7f;
ap_uint<6> Val2 = 0x3f;

Result = Val1; // Yields:0x3ff (sign-extended)
Result = Val2; // Yields:0x03f (zero-padded)

Result = ap_uint<7>(Val1); // Yields:0x07f (zero-padded)
Result = ap_int<6>(Val2); // Yields:0x3ff (sign-extended)

```

## ビット幅が大きい値を幅が小さいものへ代入する場合の切り捨て

ビット幅が大きい値を幅の小さなものへ代入すると、デスティネーション値(幅の小さいほうの値)の最上位ビット(MSB)を超えたビットはすべて切り捨てられます。

この切り捨てが行われるとき、符号情報が特別に処理されるわけではないので、予期しない動作が見られる可能性があります。予期しない動作を防ぐには、明示的な型変換を利用します。

## クラス演算子およびメソッド

通常、ネイティブの C/C++ 整数データ型で実行可能な有効な演算は、演算子をオーバーロードすることで、`ap_[u] int` 型でサポートされています。

オーバーロードされた演算子に加え、一部の特定演算子およびメソッドを使用してビット レベルの演算を簡単にすることができます。

### 2 進数の演算子

任意精度演算を行うため、標準 2 進数整数演算子がオーバーロードされます。次の演算子ではすべて、`ap_[u] int` の 2 つのオペランドが使用されるか、または `ap_[u] int` 型を 1 つと C/C++ 整数データ型 1 つ(`char`、`short`、`int`など)が使用されます。

結果値の幅および符号は、デスティネーション変数(または式)の幅に基づいて符号拡張、ゼロの追加、または切り捨てが実行される前に、オペランドの幅および符号で決まります。返される値の詳細は各演算子で説明されています。

式に `ap_[u] int` 型および C/C++ 整数型の両方が含まれている場合、C++ 型では次の幅が使用されます。

- `char :8` ビット
- `short :16` ビット
- `int :32` ビット
- `long :32` ビット
- `long long :64` ビット

### 加算

```
ap_(u)int::RType ap_(u)int::operator + (ap_(u)int op)
```

この式は、2 つの `ap_[u] int`(または `ap_[u] int` 1 つと C/C++ 整数型を 1 つ)の和を計算します。

和の幅は、オペランドの幅の大きいほうよりも 1 ビット大きくなります(幅の広いほうが符号なしで幅の小さいほうが符号付きの場合のみ 2 ビット大きくなる)。

オペランドのいずれか(または両方)が符号付きであれば、和は符号付きとして処理されます。

### 減算

```
ap_(u)int::RType ap_(u)int::operator - (ap_(u)int op)
```

2 つの整数の差を計算する式です。

差の幅は、オペランドの幅の大きいほうよりも 1 ビット大きくなります(幅の広いほうが符号なしで幅の小さいほうが符号付きの場合のみ 2 ビット大きくなる)。代入前に、デスティネーション変数の幅に基づいて、符号拡張、ゼロの追加、または切り捨てが実行されます。

オペランドの符号に関わらず、差は符号付きとして処理されます。

## 乗算

```
ap_(u) int::RTType ap_(u) int::operator * (ap_(u) int op)
```

2 つの整数値の積を計算します。

積の幅はオペランドの幅の合計です。

オペランドのいずれかが符号付きであれば、積は符号付きとして処理されます。

## 除算

```
ap_(u) int::RTType ap_(u) int::operator / (ap_(u) int op)
```

2 つの整数値の積を計算します。

除数が符号なしである場合、商の幅は被除数の幅となり、そうでない場合は、商の幅は被除数の幅に 1 を足したものとなります。

オペランドのいずれかが符号付きであれば、商は符号付きとして処理されます。



**重要:** 除算演算子を Vivado HLS で合成すると、生成された RTL で適宜パラメーター設定されたザイリンクス LogiCORE 除算コアがインスタンシエートされます。

## 剰余

```
ap_(u) int::RTType ap_(u) int::operator % (ap_(u) int op)
```

2 つの整数値の整数除算の余りを計算します。

オペランドがどちらも同じ符号である場合は、剰余の幅はオペランドの幅の最小値となります。除数が符号なし、被除数が符号付きの場合は、剰余の幅は除数の幅に 1 を足したものとなります。

商の符号は被除数のものと同じものとして処理されます。



**重要:** 剰余 (%) 演算子を Vivado HLS で合成すると、生成された RTL で適宜パラメーター設定されたザイリンクス LogiCORE 除算コアがインスタンシエートされます。

演算子の例:

```
ap_uint<71> Rslt;
ap_uint<42> Val1 = 5;
ap_int<23> Val2 = -8;

Rslt = Val1 + Val2; // Yields:-3 (43 bits) sign-extended to 71 bits
Rslt = Val1 - Val2; // Yields:+3 sign extended to 71 bits
Rslt = Val1 * Val2; // Yields:-40 (65 bits) sign extended to 71 bits
Rslt = 50 / Val2; // Yields:-6 (33 bits) sign extended to 71 bits
Rslt = 50 % Val2; // Yields:+2 (23 bits) sign extended to 71 bits
```

## ビット単位の論理演算子

ビット単位の論理演算子はすべて、2つのオペランドの幅の最大値となる幅の値を返し、両方のオペランドが符号なしの場合にのみ符号なしとして処理され、そうでない場合は、符号付きとして処理されます。

符号拡張(またはゼロの追加)は、デステイネーション変数ではなく、式の符号に基づいて実行されます。

### ビット単位の OR

```
ap_(u)int::RType ap_(u)int::operator | (ap_(u)int op)
```

2つのオペランドのビット単位の OR 値を返します。

### ビット単位の AND

```
ap_(u)int::RType ap_(u)int::operator & (ap_(u)int op)
```

2つのオペランドのビット単位の AND 値を返します。

### ビット単位の XOR

```
ap_(u)int::RType ap_(u)int::operator ^ (ap_(u)int op)
```

2つのオペランドのビット単位の XOR 値を返します。

## 単項演算子

### 加算

```
ap_(u)int ap_(u)int::operator + ()
```

ap\_[u]int オペランドのセルフ コピーを返します。

### 減算

```
ap_(u)int::RType ap_(u)int::operator - ()
```

符号付きの場合は同じ幅でオペランドのニゲートされた値を返し、符号なしの場合はその幅に 1 を足した値を返します。

返される値は常に符号付きです。

### ビット単位の逆

```
ap_(u)int::RType ap_(u)int::operator ~ ()
```

同じ幅および符号で、オペランドのビット単位の逆(NOT)を返します。

### 等価ゼロ

```
bool ap_(u)int::operator !()
```

オペランドがゼロ(0)の場合のみブール形式の `false` を返し、そうでない場合は `true` を返します。

## シフト演算子

各シフト演算子には 2つのバージョンがあり、1つは符号なしの右辺(RHS)オペランド、もう1つは符号付きの RHS です。

符号付き RHS に負の値が与えられるとシフト演算の方向を逆にします。たとえば、RHS オペランドの絶対値によるシフトが逆の方向で発生します。

シフト演算子は、左辺 (LHS) オペランドと同じ幅の値を返します。C/C++ の場合と同じように、右シフトの LHS オペランドが符号付きの場合、符号ビットは、LHS オペランドの符号を維持しつつ、最上位ビットにコピーされます。

### 符号なし整数右シフト

```
ap_(u)int ap_(u)int::operator << (ap_uint<int_W2> op)
```

### 整数右シフト

```
ap_(u)int ap_(u)int::operator << (ap_int<int_W2> op)
```

### 符号なし整数左シフト

```
ap_(u)int ap_(u)int::operator >> (ap_uint<int_W2> op)
```

### 整数左シフト

```
ap_(u)int ap_(u)int::operator >> (ap_int<int_W2> op)
```



**注意:** 左シフト演算子の結果を幅が広いほうのデスティネーション変数に代入すると、情報の一部またはすべてが失われる場合がありますので注意してください。予期しない動作を防ぐには、シフト式を代入されるほうの型に明示的に型変換するようにしてください。

シフト演算の例:

```
ap_uint<13> Rslt;
ap_uint<7> Val1 = 0x41;
Rslt = Val1 << 6; // Yields:0x0040, i.e. msb of Val1 is lost
Rslt = ap_uint<13>(Val1) << 6; // Yields:0x1040, no info lost
ap_int<7> Val2 = -63;
Rslt = Val2 >> 4; //Yields:0x1ffc, sign is maintained and extended
```

### 複合代入演算子

次の複合演算子がサポートされています。

**\*=    /=    %=    +=    -=    <<=    >>=    &=    ^=    |=**

RHS 式はまず求めてから、基本演算子に RHS オペランドとして与えられ、LHS 変数に結果が代入されます。式の長さ、符号、符号拡張か切り捨てかのルールは、関連する演算に対し上記で説明されているように、適用されます。

複合代入文の例:

```
ap_uint<10> Val1 = 630;
ap_int<3> Val2 = -3;
ap_uint<5> Val3 = 27;

Val1 += Val2 - Val3; // Yields:600 and is equivalent to:
// Val1 = ap_uint<10>(ap_int<11>(Val1) +
// ap_int<11>((ap_int<6>(Val2) -
// ap_int<6>(Val3))));
```

## インクリメントおよびデクリメント演算子

インクリメントおよびデクリメントの演算子が提供されています。オペランドと同じ幅の値を返しますが、両方のオペランドが符号なしの場合のみ値は符号なしで、そうでない場合は、符号付きです。

### プリインクリメント

```
ap_(u)int& ap_(u)int::operator ++ ()
```

オペランドのインクリメントされた値を返し、またその値をオペランドに代入します。

### ポストインクリメント

```
const ap_(u)int ap_(u)int::operator ++ (int)
```

インクリメントされた値をオペランド変数に代入する前に、オペランドの値を返します。

### プレデクリメント

```
ap_(u)int& ap_(u)int::operator -- ()
```

オペランドのデクリメントされた値を返し、またその値をオペランドに代入します。

### ポストデクリメント

```
const ap_(u)int ap_(u)int::operator -- (int)
```

デクリメントされた値をオペランド変数に代入する前に、オペランドの値を返します。

## 関係演算子

すべての関係演算子がサポートされており、比較結果に基づいてブール形式の値が返されます。ap\_[u]int 型の変数は、これらの演算子を使用して C/C++ 基本整数型に比較することができます。

### 等号

```
bool ap_(u)int::operator == (ap_(u)int op)
```

### 等号否定

```
bool ap_(u)int::operator != (ap_(u)int op)
```

### 小なり

```
bool ap_(u)int::operator < (ap_(u)int op)
```

### 大なり

```
bool ap_(u)int::operator > (ap_(u)int op)
```

### 小なりイコール

```
bool ap_(u)int::operator <= (ap_(u)int op)
```

### 大なりイコール

```
bool ap_(u)int::operator >= (ap_(u)int op)
```

## その他のクラス メソッド および 演算子

### ビット レベル 演算

`ap_[u] int` 型変数に格納されている値に基づいて一般的なビット レベル演算を行うために次のメソッドが提供されています。

#### 長さ

```
int ap_(u)int::length ()
```

このメソッドは整数値を返し、`ap_[u] int` 変数でのビット合計を出力します。

#### 連結

```
ap_concat_ref ap_(u)int::concat (ap_(u)int low)
ap_concat_ref ap_(u)int::operator , (ap_(u)int high, ap_(u)int low)
```

2 つの `ap_[u] int` 変数を連結します。返された値の幅はオペランドの幅の和です。

`high` および `low` の引数にはそれぞれ結果値の高位ビットまたは低位ビットが出力されます。`concat()` メソッドには `low` の引数を使用します。

オーバーロードされたカンマ演算子を使用するときは括弧が必要です。カンマ演算子バージョンが代入の `LHS` に現れることもあります。

整数リテラルを含むネイティブ C/C++ 型は、予期しない結果を避けるため、連結前に該当する `ap_[u] int` 型に明示的に型変換する必要があります。

連結の例は次のとおりです。

```
ap_uint<10> Rslt;
ap_int<3> Val1 = -3;
ap_int<7> Val2 = 54;

Rslt = (Val2, Val1); // Yields:0x1B5
Rslt = Val1.concat(Val2); // Yields:0x2B6
(Val1, Val2) = 0xAB; // Yields:Val1 == 1, Val2 == 43
```

### ビット選択

```
ap_bit_ref ap_(u)int::operator [] (int bit)
```

任意精度の整数値から 1 ビット選択し、それを返します。

返された値は参照値で、この `ap_[u] int` での対応ビットを設定または消去するために使用します。

`bit` 引数は `int` 値である必要があります。選択するビットの指数を指定し、最下位ビットは指数 0 になります。最高指数はこの `ap_[u] int` のビット幅から 1 を引いた値になります。

`ap_bit_ref` はビットで指定されているこの `ap_[u] int` インスタンスの 1 ビットへの参照を表します。

### 範囲選択

```
ap_range_ref ap_(u)int::range (unsigned Hi, unsigned Lo)
ap_range_ref ap_(u)int::operator () (unsigned Hi, unsigned Lo)
```

引数で指定されるビットの範囲で表される値を返します。

引数 Hi は、範囲内の最上位ビット (MSB) を、Lo は最下位ビット (LSB) を指定します。

ソース変数の LSB は 0 の位置にあります。引数 Hi の値が Lo のものより小さい場合、ビットは逆順で返されます。

範囲選択を使用した例は次のようにになっています。

```
ap_uint<4> Rslt;

ap_uint<8> Val1 = 0x5f;
ap_uint<8> Val2 = 0xaa;

Rslt = Val1.range(3, 0); // Yields:0xF
Val1(3,0) = Val2(3, 0); // Yields:0x5A
Val1(4,1) = Val2(4, 1); // Yields:0x55
Rslt = Val1.range(7, 4); // Yields:0xA; bit-reversed!
```

### AND を使用した低減

```
bool ap_(u)int::and_reduce ()
```

AND 演算をこの `ap_(u)int` のすべてのビットに適用し、結果値をシングルビットで返します。この値を -1(すべて 0) と比較し、一致すれば `true`、そうでなければ `false` を返します。

### OR を使用した低減

```
bool ap_(u)int::or_reduce ()
```

OR 演算をこの `ap_(u)int` のすべてのビットに適用し、結果値をシングルビットで返します。この値を 0(すべてゼロ) と比較し、一致すれば `false`、そうでなければ `true` を返します。

### XOR を使用した低減

```
bool ap_(u)int::xor_reduce ()
```

XOR 演算をこの `ap_int` のすべてのビットに適用し、結果値をシングルビットで返します。これは、この値の 1 ビットの数をカウントするのと同じで、カウント数が偶数であれば `false`、奇数であれば `true` を返します。

### NAND を使用した低減

```
bool ap_(u)int::nand_reduce ()
```

NAND 演算をこの `ap_int` のすべてのビットに適用し、結果値をシングルビットで返します。この値を -1(すべて 1) と比較し、一致すれば `false`、そうでなければ `true` を返します。

### NOR を使用した低減

```
bool ap_int::nor_reduce ()
```

NOR 演算をこの `ap_int` のすべてのビットに適用し、結果値をシングルビットで返します。この値を 0(すべて 0) と比較し、一致すれば `true`、そうでなければ `false` を返します。

### XNOR を使用した低減

```
bool ap_(u)int::xnor_reduce ()
```

XNOR 演算をこの `ap_(u)int` のすべてのビットに適用し、結果値をシングルビットで返します。これは、この値の 1 ビットの数をカウントするのと同じで、カウント数が偶数であれば `true`、奇数であれば `false` を返します。

さまざまなビット低減メソッドの例は次のようになっています。

```
ap_uint<8> Val = 0xaa;
```

```

bool t = Val.and_reduce(); // Yields: false
t = Val.or_reduce(); // Yields: true
t = Val.xor_reduce(); // Yields: false
t = Val.nand_reduce(); // Yields: true
t = Val.nor_reduce(); // Yields: false
t = Val.xnor_reduce(); // Yields: true

```

## ビットのリバース

```
void ap_(u)int::reverse ()
```

このメンバー ファンクションは、ap\_(u)int インスタンスの内容を逆転させます。たとえば、LSB は MSB、またはその逆になります。

例：

```

ap_uint<8> Val = 0x12;

Val.reverse(); // Yields:0x48

```

## ビット値のテスト

```
bool ap_(u)int::test (unsigned i)
```

このメンバー ファンクションは、ap\_(u)int インスタンスの指定ビットが 1 であるかどうかをチェックします。1 であれば true、そうでなければ false を返します。

例：

```

ap_uint<8> Val = 0x12;

bool t = Val.test(5); // Yields: true

```

## ビット値の設定

```

void ap_(u)int::set (unsigned i, bool v)
void ap_(u)int::set_bit (unsigned i, bool v)

```

このメンバー 関数は、ap\_(u)int の指定ビットを整数 v の値に設定します。

### ビットの設定 (1)

```
void ap_(u)int::set (unsigned i)
```

このメンバー 関数は、ap\_(u)int の指定ビットを 1 に設定します。

### ビットのクリア (0)

```
void ap_(u)int::clear(unsigned i)
```

このメンバー 関数は、ap\_(u)int の指定ビットを 0(ゼロ) に設定します。

### ビットの反転

```
void ap_(u)int::invert(unsigned i)
```

このメンバー 関数は、ap\_(u)int インスタンスの第 i 位ビットを反転させます。たとえば、第 i 位ビットは、元の値が 1 の場合 0 となり、0 の場合は 1 になります。

ビットの設定、クリア、反転メソッドの例：

```
ap_uint<8> Val = 0x12;
Val.set(0, 1); // Yields:0x13
Val.set_bit(5, false); // Yields:0x03
Val.set(7); // Yields:0x83
Val.clear(1); // Yields:0x81
Val.invert(5); // Yields:0x91
```

### 右に移動

```
void ap_(u)int::rrotate(unsigned n)
```

このメンバー関数は、ap\_(u)int インスタンスを *n* 桁右に移動させます。

### 左に移動

```
void ap_(u)int::lrotate(unsigned n)
```

このメンバー関数は、ap\_(u)int インスタンスを *n* 桁左に移動させます。

移動メソッドの例：

```
ap_uint<8> Val = 0x12;
Val.rrotate(3); // Yields:0x42
Val.lrotate(6); // Yields:0x90
```

### ビット単位 NOT

```
void ap_(u)int::b_not()
```

このメンバー関数は、ap\_(u)int インスタンスの各ビットを反転させ補数にします。

例：

```
ap_uint<8> Val = 0x12;
Val.b_not(); // Yields:0xED
```

### 符号のテスト

```
bool ap_int::sign()
```

このメンバー関数は、ap\_(u)int インスタンスの符号をチェックします。負の場合は `true` を、正の場合は `false` を返します。

## 明示的な変換メソッド

### C/C++ の (u)int 型

```
int ap_(u)int::to_int()
unsigned ap_(u)int::to_uint()
```

これらのメソッドは、ap\_[u]int に含まれている値でネイティブ C/C++ (ほとんどのシステムで 32 ビット) の整数を返します。値が [unsigned] int で表されるものより大きい場合は、切り捨てが実行されます。

### C/C++ 64 ビット (u)int 型

```
long long ap_(u)int::to_int64()
```

```
unsigned long long ap_(u)int::to_uint64 ()
```

これらのメソッドは、`ap_[u]int` に含まれている値でネイティブ C/C++ 64 ビットの整数を返します。値が `[unsigned] int` で表されるものより大きい場合は、切り捨てが実行されます。

#### C/C++ double 型

```
double ap_(u)int::to_double ()
```

このメソッドは、`ap_[u]int` に含まれている値のネイティブ C/C++ の `double` 型 64 ビットの浮動小数点値を返します。`ap_[u]int` が 53 ビット (`double` の仮数部のビット数) よりも大きい場合、`double` 型の値は期待値とまったく同じにならないことがあります。

## C++ の任意精度 (AP) 固定小数点型

Vivado HLS では、分数算術を簡単に処理できるよう、固定小数点型がサポートされています。この利点は次の例で説明されています。

```
ap_fixed<10, 5> Var1 = 22.96875; // 10-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5; // 12-bit word, 1 fractional bit
ap_fixed<13,5> Res1; // 13-bit signed word, 5 fractional bits

Res1 = Var1 + Var2; // Result is 535.46875
```

`Var1` および `Var2` の精度は異なるのですが、固定小数点型を使用することで、演算 (この場合は加算) が実行される前に小数点が揃えられます。小数点を揃えるために C コードで演算を実行する必要はありません。

固定小数点演算の結果値を格納するのに使用される型は、整数ビットおよび分数ビットの両方の結果値を完全に格納するのに十分な大きさである必要があります。

そうではない場合、`ap_fixed` 型でオーバーフロー (結果値の MSB が型でサポートされているよりも多い場合) および量子化 (結果値の LSB が型でサポートされているよりも少ない場合) が自動的に実行されます。`ap_[u]fixed` 型では、オーバーフローおよび量子化の実行方法に関してオプションが多く提供されており、次のセクションでその説明をします。

## ap\_[u]fixed 表現

`ap_[u]fixed` 型では、固定小数値は 2 進小数点の位置を指定した、ビットのシーケンスで表現されます。2 進小数点の左側にあるビットは値の整数部を示し、右側にあるビットは値の小数部を示します。

`ap_[u]fixed` 型は次のように定義されます。

```
ap_[u]fixed<int W,
 int I,
 ap_q_mode Q,
 ap_o_mode O,
 ap_sat_bits N>;
```

- `W` 属性には、ワードのビット合計数を指定するパラメーターを設定します。このパラメーター値には定数の整数式のみを使用します。
- `I` 属性には、整数部を表すビット数を指定するパラメーターを設定します。`I` の値は `W` と等しいかそれ以下である必要があります。小数部を表すためのビット数は `W` から `I` を差し引いた値です。このパラメーター値には定数の整数式のみを使用します。
- `Q` 属性には量子化モードを指定するパラメーターを設定します。このパラメーター値にはあらかじめ定義されている列挙値のみを使用します。デフォルト値は `AP_TRN` です。

- 0 属性にはオーバーフロー モードを指定するパラメーターを設定します。このパラメーター値にはあらかじめ定義されている列挙値のみを使用します。デフォルト値は AP\_WRAP です。
- N 属性には、オーバーフロー ラップ モードで使用される飽和ビット数を設定します。このパラメーター値には定数の整数式のみを使用します。デフォルト値は 0 です。

**注記** : 例にあるように、量子化、オーバーフロー、飽和パラメーターが指定されていない場合は、デフォルト設定が使用されます。

量子化およびオーバーフローのモードは次で説明します。

## 量子化モード

|           |                  |
|-----------|------------------|
| 正の無限大への丸め | 「AP_RND」         |
| 0 への丸め    | 「AP_RND_ZERO」    |
| 負の無限大への丸め | 「AP_RND_MIN_INF」 |
| 無限大への丸め   | 「AP_RND_INF」     |
| 収束丸め      | 「AP_RND_CONV」    |
| 切り捨て      | 「AP_TRN」         |
| 0 への切り捨て  | 「AP_TRN_ZERO」    |

### AP\_RND

AP\_RND 量子化モードでは、特定の `ap_[u]fixed` 型の近似値に値が丸められます。

次に例を示します。

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25; // Yields:1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.0
```

### AP\_RND\_ZERO

AP\_RND\_ZERO 量子化モードでは、近似値に値が丸められ、その丸めはゼロのほうに向かって実行されます。つまり、正の値の場合は重複ビットが削除され、負の値の場合は、近似値なるように LSB が追加されます。

次に例を示します。

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields:1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.0
```

### AP\_RND\_MIN\_INF

AP\_RND\_MIN\_INF 量子化モードでは、近似値に値が丸められ、その丸めは負の無限大に向かって実行されます。つまり、正の値に対しては重複ビットが削除され、負の値に対しては LSB が追加されます。

次に例を示します。

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25; // Yields:1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.5
```

### AP\_RND\_INF

AP\_RND\_INF 量子化モードでは、近似値に値が丸められ、その丸めは LSB によって変わります。

\* 正の値に対しては、 LSB が設定されている場合は正の無限大の方向に丸められ、そうでない場合は負の無限大の方向に丸められます。

\* 負の値に対しては、 LSB が設定されている場合は負の無限大の方向に丸められ、そうでない場合は正の無限大の方向に丸められます。

次に例を示します。

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25; // Yields:1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.5
```

### AP\_RND\_CONV

AP\_RND\_CONV 量子化モードでは、近似値に値が丸められ、その丸めは LSB によって変わります。 LSB が設定されている場合は正の無限大の方向に丸められ、そうでない場合は負の無限大の方向に丸められます。

次に例を示します。

```
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = 0.75; // Yields:1.0
ap_fixed<3, 2, AP_RND_CONV, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.0
```

### AP\_TRN

AP\_TRN 量子化モードでは、近似値に値が丸められ、その丸めは常に負の無限大に向かって実行されます。

次に例を示します。

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25; // Yields:1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.5
```

### AP\_TRN\_ZERO

AP\_TRN\_ZERO 量子化モードでは、近似値に値が丸められます。

- \* 正の値の場合、AP\_TRN モードと同じ丸めになります。
- \* 負の値の場合はゼロの方向へ丸められます。

次に例を示します。

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields:1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields:-1.0
```

## オーバーフロー モード

|            |               |
|------------|---------------|
| 飽和         | 「AP_SAT」      |
| 0 への飽和     | 「AP_SAT_ZERO」 |
| 対称飽和       | 「AP_SAT_SYM」  |
| 折り返し       | 「AP_WRAP」     |
| 符号絶対値の折り返し | 「AP_WRAP_SM」  |

### AP\_SAT

AP\_SAT オーバーフロー モードでは、オーバーフローが発生したときは値が最大値に飽和し、負のオーバーフローのときは負の最大値に飽和します。

次に例を示します。

```
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields:15.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields:7.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields:0.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields:-8.0
```

### AP\_SAT\_ZERO

AP\_SAT\_ZERO オーバーフロー モードでは、オーバーフローまたは負のオーバーフローが発生したときに値が 0 になります。

次に例を示します。

```
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields:0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields:0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields:0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields:0.0
```

### AP\_SAT\_SYM

AP\_SAT\_SYM オーバーフロー モードでは、オーバーフローが発生したときは値が最大値に飽和し、負のオーバーフローのときは最小値(符号付きの `ap_fixed` 型の場合は負の最大値で、符号なしの `ap_ufixed` 型の場合は 0)に飽和します。

次に例を示します。

```
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields:15.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields:7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields:0.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields:-8.0
```

### AP\_WRAP

AP\_WRAP オーバーフロー モードでは、オーバーフローが発生したときに値が折り返されます。

次に例を示します。

```
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; // Yields:3.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; // Yields:-1.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields:13.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields:-3.0
```

N の値が 0 の場合(デフォルトのオーバーフロー モード):

- 範囲外の MSB はすべて削除されます。
- 符号なしの数値の場合、最大値に達したら 0 に折り返されます。
- 符号付きの数値の場合、最大値に達したら最小値に折り返されます。

N が 0 より大きい場合:

- N が 0 より大きい場合、MSB は飽和するか、または 1 に設定されます。
- 符号ビットは保持されるため、正の値は正のまま、負の値は負のままになります。
- 飽和していないビットは LSB 側からコピーされます。

### AP\_WRAP\_SM

AP\_WRAP\_SM オーバーフロー モードでは、値が符号絶対値で折り返されます。

次に例を示します。

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0; // Yields:-4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0; // Yields:2.0
```

N の値が 0 の場合 (デフォルトのオーバーフロー モード):

- このモードでは符号絶対値のラップが使用されます。
- 符号ビットは最下位の削除されたビットの値に設定されます。
- 最上位の残りのビットが元の MSB とは異なる場合、残っているビットすべてが反転されます。
- MSB が同じである場合は、ほかのビットがコピーされます。
  - 手順 1:最初の重複している MSB が削除されます。
  - 手順 2:新しい符号ビットは削除されたビットの最下位ビットです。この場合は 0 です。
  - 手順 3:新しい符号ビットが新しい値の符号と比較されます。
- 異なる場合は、すべての数値が反転されます。この場合は数値は異なります。

N が 0 より大きい場合 :

- 符号絶対値の飽和が使用されます。
- N 個の MSB が 1 に飽和します。
- N =0 のケースと同様の動作になりますが、正の数値は正のまま、負の数値は負のままになる点が異なります。

## ap\_[u]fixed<> 型のコンパイル

ap\_[u]fixed<> クラスを使用するには、ap\_[u]fixed<> 変数を参照するソース ファイルすべてに ap\_fixed.h ヘッダーファイルを含める必要があります。

これらのクラスを使用するソフトウェア モデルをコンパイルするとき、Vivado HLS のヘッダーファイルの場所を指定する必要がある場合があります。たとえば、gcc コンパイルには -I/<HLS\_HOME>/include オプションを追加するなどします。

g++ -O3 オプションを使用してコンパイルする場合には、ソフトウェア モデルでベスト パフォーマンスが見られます。

## ap\_[u]fixed<> 変数の宣言/定義

クラスには、ap\_fixed<W, I> および ap\_ufixed<W, I> という符号付きおよび符号なしのものがあります。

次の例にあるように、C/C++ の `typedef` 文を使用してユーザー定義型を作成することができます。

```
include "ap_fixed.h" // use ap_[u]fixed<> types

typedef ap_ufixed<128,32> uint128_t; // 128-bit user defined type,
 // 32 integer bits
```

## 定数 (リテラル) からの初期化および代入

ap\_[u]fixed 変数は、一般的な C/C++ 幅の標準浮動小数点定数で初期化されます (float 型の場合は 32 ビット、double 型の場合は 64 ビット)。つまり、通常は、浮動小数点値は单精度または倍精度です。

このような浮動小数点定数は、値の符号により、任意精度の固定小数点変数のフル サイズの幅に処理および変換されます。

次に例を示します。

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
```

## コンソール I/O (出力) のサポート

`ap_[u]fixed<>` 変数の初期化および代入の場合と同様に、64 ビットよりも大きな値の出力をサポートするための機能が提供されています。

`ap_[u]int` 変数に格納されている値を出力するのに最も簡単な方法は、C++ の標準出力ストリーム `std::cout` (`#include <iostream>` または `<iostream.h>`) を使用する方法です。ストリーム挿入演算子 `<<` は、任意の `ap_[u]int` 変数に対する範囲全体に含まれる値が正しく出力されるように、オーバーロードされます。ストリームマニピュレーターの `dec`、`hex` および `oct` もサポートされていて、それぞれ 10 進数、16 進数、または 8 進数で値をフォーマットすることができます。

値の出力に `cout` を使用した例は次のとおりです。

```
#include <iostream.h>
// Alternative:#include <iostream>

ap_uint<72> Val("10fedcba9876543210");

cout << Val << endl; // Yields:"313512663723845890576"
cout << hex << val << endl; // Yields:"10fedcba9876543210"
cout << oct << val << endl; // Yields:"41773345651416625031020"
```

標準 C ライブラリ (`#include <stdio.h>`) を使用して、64 ビットよりも大きな値を出力することもできます。また、値を C++ の `std::string` に変換してから、C の文字列に変換します。`ap_[u]int` クラスでは、最初の変換を行うのに `to_string()` メソッドが提供されており、`std::string` クラスには、ヌルで終了する文字列に変換するために `c_str()` メソッドがあります。

`ap[u]int::to_string()` メソッドは目的の基数を指定するオプションの引数を渡すことができます。この基数引数に有効な値は 2、8、10、および 16 で、それぞれ 2 進数、8 進数、10 進数、16 進数を指定します。デフォルト値は 16 です。

`ap_[u]int::to_string()` の 2 つ目のオプションの引数は、10 進数以外のフォーマットで符号付きの値を出力するかどうかを指定します。この引数はブール形式です。デフォルト値は `false` で、10 進数以外のフォーマットで符号なしの値が出力されます。

値の出力に `printf` を使用した例は次のとおりです。

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str()); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str()); // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str()); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str()); // => "-7F0123456789ABCDF0"
```

## ap\_[u]fixed<> 型を使用した式

任意精度の固定小数点値は、C/C++ でサポートされている演算子を使用する式で使用できます。任意精度の固定小数点型または変数を定義し終えたら、その使用方法は C/C++ 言語のほかの浮動小数点型または変数と同じです。ただし、注意点がいくつかあります。

## ゼロおよび符号拡張

ソース値の符号により、ビット幅が小さいほうの値はすべてゼロが追加されるかまたは符号拡張されることを以前説明しました。ビット幅が小さいほうの値を大きなほうに代入するとき、符号を得るにはキャストを挿入する必要がある場合があります。

### 切り捨て

デステイネーション変数よりもビット幅が大きな値の任意精度の固定小数点の代入を行う場合は、切り捨てが実行されます。

## クラス演算子およびメソッド

通常、ネイティブの C/C++ 整数データ型で実行可能な有効な演算は、演算子をオーバーロードすることで、`ap_[u]fixed` 型でサポートされています。オーバーロードされた演算子に加え、一部の特定演算子およびメソッドを使用してビット レベルの演算を簡単にすることができます。

### 2 進数の演算子

#### 加算

```
ap_[u]fixed::RType ap_[u]fixed::operator + (ap_[u]fixed op)
```

任意のオペランド `op` を使用し任意精度の固定小数点値を加算します。



**ヒント :** オペランドには、`ap_[u]fixed`、`ap_[u]int` または C/C++ 整数型を使用することができます。結果型 `ap_[u]fixed::RType` は 2 つのオペランドの型によって異なります。

次に例を示します。

```
ap_fixed<76, 63> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 + Val2; //Yields 6722.480957
```

`Val2` は、整数部および分数部の両方でビット幅が大きいため、結果値をすべて格納できるようにするには、結果型はこの同じビット幅に 1 を足したものになります。

#### 減算

```
ap_[u]fixed::RType ap_[u]fixed::operator - (ap_[u]fixed op)
```

任意のオペランド `op` を使用し任意精度の固定小数点値を減算します。

結果型 `ap_[u]fixed::RType` は 2 つのオペランドの型によって異なります。

次に例を示します。

```
ap_fixed<76, 63> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 - Val1; // Yields 6720.23057
```

Val2 は、整数部および分数部の両方でビット幅が大きいため、結果値をすべて格納できるようにするには、結果型はこの同じビット幅に 1 を足したものになります。

## 乗算

```
ap_[u]fixed::RType ap_[u]fixed::operator * (ap_[u]fixed op)
```

任意のオペランド op を使用し任意精度の固定小数点値を乗算します。

次に例を示します。

```
ap_fixed<80, 64> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 * Val2; // Yields 7561.525452
```

Val1 と Val2 を掛け合わせています。この結果型は、整数部のビット幅と分数部のビット幅の和になります。

## 除算

```
ap_[u]fixed::RType ap_[u]fixed::operator / (ap_[u]fixed op)
```

任意のオペランド op を使用し任意精度の固定小数点値を除算します。

次に例を示します。

```
ap_fixed<84, 66> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 / Val1; // Yields 5974.538628
```

Val1 と Val2 を減算しています。十分な精度を保持するには、次のようにになります。

- 結果型の整数ビット幅は、Val1 の整数ビット幅と、Val2 の分数ビット幅の和になります。
- 結果型の分数ビット幅は、Val1 の分数ビット幅と、Val2 の全ビット幅の和になります。

## ビット単位の論理演算子

### ビット単位の OR

```
ap_[u]fixed::RType ap_[u]fixed::operator | (ap_[u]fixed op)
```

ビット単位の OR 演算が任意精度の固定小数点値および任意のオペランド op に適用されます。

次に例を示します。

```
ap_fixed<75, 62> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 | Val2; // Yields 6271.480957
```

### ビット単位の AND

```
ap_[u]fixed::RType ap_[u]fixed::operator & (ap_[u]fixed op)
```

ビット単位の AND 演算が任意精度の固定小数点値および任意のオペランド op に適用されます。

次に例を示します。

```
ap_fixed<75, 62> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;
Result = Val1 & Val2; // Yields 1.00000
```

## ビット単位の XOR

```
ap_[u]fixed::RTType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

ビット単位の XOR 演算が任意精度の固定小数点値および任意のオペランド op に適用されます。

次に例を示します。

```
ap_fixed<75, 62> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;
Result = Val1 ^ Val2; // Yields 6720.480957
```

## インクリメントおよびデクリメント演算子

### プリインクリメント

```
ap_[u]fixed ap_[u]fixed::operator ++ ()
```

任意精度の固定小数点変数を 1 ずつインクリメントします。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;
Result = ++Val1; // Yields 6.125000
```

### ポストインクリメント

```
ap_[u]fixed ap_[u]fixed::operator ++ (int)
```

任意精度の固定小数点変数を 1 ずつインクリメントし、この任意精度の固定小数点の元の値を返します。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;
Result = Val1++; // Yields 5.125000
```

### プレデクリメント

```
ap_[u]fixed ap_[u]fixed::operator -- ()
```

任意精度の固定小数点変数を 1 ずつデクリメントします。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = --Val1; // Yields 4.125000
```

## ポストデクリメント

```
ap_[u]fixed ap_[u]fixed::operator -- (int)
```

任意精度の固定小数点変数を 1 ずつデクリメントし、この任意精度の固定小数点の元の値を返します。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1--; // Yields 5.125000
```

## 単項演算子

### 加算

```
ap_[u]fixed ap_[u]fixed::operator + ()
```

任意精度の固定小数点変数のセルフ コピーを返します。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1; // Yields 5.125000
```

### 減算

```
ap_[u]fixed::RTYPE ap_[u]fixed::operator - ()
```

任意精度の固定小数点変数の負の値を返します。

次に例を示します。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = -Val1; // Yields -5.125000
```

### 等価ゼロ

```
bool ap_[u]fixed::operator !=()
```

任意精度の固定小数点変数を 0 と比較し、結果を返します。

次に例を示します。

```
bool Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1; // Yields false
```

## ビット単位の逆

```
ap_[u]fixed::RType ap_[u]fixed::operator ~ ()
```

任意精度の固定小数点変数のビット単位の補数を返します。

次に例を示します。

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ~Val1; // Yields -5.25
```

## シフト演算子

### 符号なし左シフト

```
ap_[u]fixed ap_[u]fixed::operator << (ap_uint<_W2> op)
```

任意の整数オペランドで左にシフトさせ、結果値を返します。オペランドには C/C++ の整数型を使用することができます (char、short、int、または long)。

左シフト演算の返される型は、シフトされている型と同じ幅になります。

**注記** : 現在シフトではオーバーフローまたは量子化モードはサポートできません。

次に例を示します。

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val << sh; // Yields -10.5
```

結果のビット幅は ( $W = 25, I = 15$ ) ですが、左シフト演算の結果型は Val と同じ型であるため、Val の高位 2 ビットがシフト アウトされ、結果が -10.5 となります。

結果値に 21.5 が必要な場合は、`ap_ufixed<10, 7>(Val)` のように、まず Val を`ap_fixed<10, 7>` に型変換する必要があります。

### 符号付き左シフト

```
ap_[u]fixed ap_[u]fixed::operator << (ap_int<_W2> op)
```

任意の整数オペランドで左にシフトさせ、結果値を返します。シフトの方向はオペランドが正か負かにより変わります。

- オペランドが正の場合は右シフトになります。
- オペランドが負の場合は左シフト (逆方向) になります。

オペランドには C/C++ の整数型を使用することができます (char、short、int、または long)。

右シフト演算の返される型は、シフトされている型と同じ幅になります。

次に例を示します。

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;
```

```

ap_int<4> Sh = 2;
Result = Val << sh; // Shift left, yields -10.25

Sh = -2;
Result = Val << sh; // Shift right, yields 1.25

```

## 符号なし右シフト

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_uint<_W2> op)
```

任意の整数オペランドで右にシフトさせ、結果値を返します。オペランドには C/C++ の整数型を使用することができます (char、short、int、または long)。

右シフト演算の返される型は、シフトされている型と同じ幅になります。

次に例を示します。

```

ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val >> sh; // Yields 1.25

```

すべての上位ビットを保持する必要がある場合は、`ap_fixed<10, 5>(Val)` のように、まず Val の分数部のビット幅を拡張します。

## 符号付き右シフト

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_int<_W2> op)
```

任意の整数オペランドで右にシフトさせ、結果値を返します。シフトの方向はオペランドが正か負かにより変わります。

- オペランドが正の場合は右シフトになります。
- オペランドが負の場合は左シフト (逆方向)になります。

オペランドには C/C++ の整数型を使用することができます (char、short、int、または long)。

右シフト演算の返される型は、シフトされている型と同じ幅になります。次に例を示します。

```

ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val >> sh; // Shift right, yields 1.25

Sh = -2;
Result = Val >> sh; // Shift left, yields -10.5

1.25

```

## 関係演算子

### 等号

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

任意のオペランドで任意精度の固定小数点変数を比較し、等しければ `true` を返し、等しくなければ `false` を返します。

オペランド op には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。次に例を示します。

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 == Val2; // Yields true
Result = Val1 == Val3; // Yields false
```

## 等号否定

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

任意のオペランドで任意精度の固定小数点変数を比較し、等しくなければ `true` を返し、等しければ `false` を返します。

オペランド op には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。次に例を示します。

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 != Val2; // Yields false
Result = Val1 != Val3; // Yields true
```

## 大なりイコール

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

任意のオペランドで変数を比較し、等しい、またはオペランドより大きい場合は `true` を返し、そうでない場合は `false` を返します。

オペランド op には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。

次に例を示します。

```
bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 >= Val2; // Yields true
Result = Val1 >= Val3; // Yields false
```

## 小なりイコール

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

任意のオペランドで変数を比較し、等しい、またはオペランドより小さい場合は `true` を返し、そうでなければ `false` を返します。

オペランド op には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。

次に例を示します。

```
bool Result;
```

```

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 <= Val2; // Yields true
Result = Val1 <= Val3; // Yields true

```

## 大なり

```
bool ap_[u]fixed::operator > (ap_[u]fixed op)
```

任意のオペランドで変数を比較し、オペランドより大きい場合は `true` を返し、そうでなければ `false` を返します。

オペランド `op` には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。

次に例を示します。

```

bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 > Val2; // Yields false
Result = Val1 > Val3; // Yields false

```

## 小なり

```
bool ap_[u]fixed::operator < (ap_[u]fixed op)
```

任意のオペランドで変数を比較し、オペランドより小さい場合は `true` を返し、そうでなければ `false` を返します。

オペランド `op` には、`ap_[u]fixed`、`ap_int` または C/C++ 整数型を使用することができます。次に例を示します。

```

bool Result;

ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 < Val2; // Yields false
Result = Val1 < Val3; // Yields true

```

## ビット演算子

### ビットの選択および設定

```
af_bit_ref ap_[u]fixed::operator [] (int bit)
```

任意精度の固定小数点値から 1 ビット選択し、それを返します。

返された値は参照値で、`ap_[u]fixed` 変数での対応ビットを設定または消去するために使用することができます。ビット引数は整数値である必要があり、選択するビットの指数を指定します。最下位ビットは指数 0 になります。最大指数はこの `ap_[u]fixed` 変数のビット幅から 1 を引いた値になります。

結果型は `af_bit_ref` で値は 0 または 1 です。次に例を示します。

```
ap_int<8, 5> Value = 1.375;
```

```

Value[3]; // Yields 1
Value[4]; // Yields 0

Value[2] = 1; // Yields 1.875
Value[3] = 0; // Yields 0.875

```

## ビット範囲

```

af_range_ref af_(u)fixed::range (unsigned Hi, unsigned Lo)
af_range_ref af_(u)fixed::operator [] (unsigned Hi, unsigned Lo)

```

この演算は、ビット選択の演算子 [] に似ていますが、1 ビットではなくビットの範囲に対して演算が行われ、

任意精度の固定小数点変数からビットのグループを選択します。引数 Hi は範囲の上のほうのビットを指定し、引数 Lo は最下位ビットを指定します。Lo が Hi よりも大きい場合は、選択されたビットが逆の順序で返されます。

返される型 af\_range\_ref は、Hi および Lo で指定される ap\_[u]fixed 変数の範囲の参照です。次に例を示します。

```

ap_uint<4> Result = 0;
ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(3, 0); // Yields:0x5
Value(3, 0) = Repl(3, 0); // Yields:-1.5

// when Lo > Hi, return the reverse bits string
Result = Value.range(0, 3); // Yields:0xA

```

## 範囲選択

```

af_range_ref af_(u)fixed::range ()
af_range_ref af_(u)fixed::operator []

```

これは、範囲選択演算子 [] の特殊ケースで、任意精度の固定小数点値からすべてのビットを標準の順序で選択します。

返される型 af\_range\_ref は、Hi = W - 1 および Lo = 0 で指定される範囲の参照です。次に例を示します。

```

ap_uint<4> Result = 0;

ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(); // Yields:0x5
Value() = Repl(3, 0); // Yields:-1.5

```

## 長さ

```
int ap_[u]fixed::length ()
```

任意精度の固定小数点値でビット数を示す整数値を返します。1 つの型または値で使用することができます。次に例を示します。

```

ap_ufixed<128, 64> My128APFixed;

int bitwidth = My128APFixed.length(); // Yields 128

```

## 明示的な変換メソッド

### 固定小数点から double 型

```
double ap_[u]fixed::to_double ()
```

このメンバー ファンクションは、IEEE の倍精度フォーマットで固定小数点値を返します。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
double Result;

Result = MyAPFixed.to_double(); // Yields 333.789
```

### 固定小数点から ap\_int 型

```
ap_int ap_[u]fixed::to_ap_int ()
```

このメンバー ファンクションは、この固定小数点値を、すべての整数ビットを取り込む ap\_int 型に明示的に変換します(分数ビットは切り捨て)。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
ap_uint<77> Result;

Result = MyAPFixed.to_ap_int(); //Yields 333
```

### 固定小数点から整数型

```
int ap_[u]fixed::to_int ()
unsigned ap_[u]fixed::to_uint ()
ap_slong ap_[u]fixed::to_int64 ()
ap_ulong ap_[u]fixed::to_uint64 ()
```

このメンバー ファンクションは、この固定小数点値を C のビルドイン整数型に明示的に変換します。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
unsigned int Result;

Result = MyAPFixed.to_uint(); //Yields 333

unsigned long long Result;
Result = MyAPFixed.to_uint64(); //Yields 333
```

# 高位合成コマンド リファレンス ガイド

## 高位合成コマンドの使用

アクティブ デザイン プロジェクトで高位合成 (HLS) コマンドを実行する前に、基本的な Vivado HLS の概念を理解しておくことをお勧めします。

1. Vivado HLS では、プロジェクトベース構造でデザインが保存されます。
2. Vivado HLS の最適化は、領域、ロケーション、またはコード内で指定します。
3. Vivado HLS の最適化は Tcl コマンドまたはソース コードのプラグマとして指定できます (どちらのオプションもテキスト ファイルで編集できるほか、Vivado HLS の GUI でも実行できます)。

これらの各概念については、本章で説明します。

## プロジェクトの管理

Vivado HLS では、プロジェクトベースのデータベースが使用され、合成および検証が管理され、結果が保存されます。デザインは、複数のソリューションを含むプロジェクトとして表示されます。

ソース コードおよびテストベンチはこのプロジェクトに保存されます。ソリューションは、ターゲット テクノロジ (FPGA ファミリおよびデバイス) を指定し、指示子を適用し、同じソース コードの異なるインプリメンテーションを作成するために使用されます。

Vivado HLS のプロジェクトおよびソリューションは、Vivado HLS で使用されるディレクトリ構造に直接反映されます。[図5-1](#) は、結果が生成された後の Vivado HLS ディレクトリの例です。

[図5-1](#) の例には、次が表示されています。

- 一番上のプロジェクトは `project.prj` で、すべてのプロジェクト データが同じ名前のプロジェクト ディレクトリに保存されています。
- プロジェクトにはソース コードとテストベンチ ファイルが含まれます。
- このプロジェクトには `solution1` と `solution2` の 2 つのソリューションがあります。
- 現在のところ、アクティブ ソリューションは **solution1** (太字でハイライト) です。
- シミュレーション結果は `sim` ディレクトリに保存されます。
- 合成結果は `syn` ディレクトリに保存されます。
  - `syn` ディレクトリには、Verilog、VHDL、SystemC で作成された RTL 出力が含まれます。
  - `syn` ディレクトリには、生成されたレポートも含まれます。

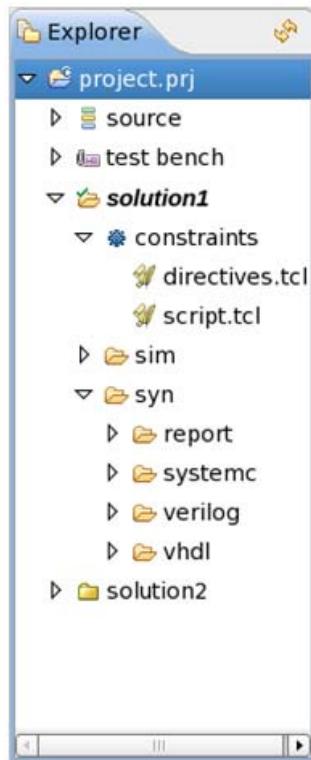


図 5-1: プロジェクトおよびソリューションの構造

Vivado HLS コマンドを使用する際は、アクティブプロジェクトまたはソリューション内でしか使用できないコマンドがあることに注意してください。通常、プロジェクトは同じソースコードのセットに使用され、ソリューションはそのソースコードのさまざまなインプリメンテーションを作成するために使用されます。

**注記** : 新しいプロジェクトを開くと、既存のプロジェクトは自動的に閉じ、新しいソリューションを開くと、既存のソリューションは自動的に閉じます。

## 高位合成の最適化のロケーション

Vivado HLS での最適化は、関数、ループ、領域、配列およびインターフェイス パラメーターで指定できます。このセクションでは、最適化の適用方法と影響するロケーションについて説明します。

最適化は、次の 3 つのいずれかの方法で指定します。

1. GUI の [Directive] タブを使用します。
2. オブジェクトが独自に識別できるのであれば（ループおよび領域にはラベルが必要）、対話型プロンプトまたはバッチファイルに Tcl コマンドを使用して指示子を指定します。
3. プラグマ指示子をソースコードに直接挿入します。

これらの方法で指定される最適化は、ソースコード内の指定ロケーション（スコープ）に適用されます。

次の例は、ソースコードの概要を示しています。

```
int foo_sub_A (int mem_1[64], ...) {
 for_A: for (int n = 0; n < 3; ++n) {
 ...
 }
 ...
}
```

```

}
int foo_sub_B (int mem_1[64], int i) {
 for_B:for (int n = 0; n < 4; ++n) {
 ...
 }
 ...
}
void foo_top (int mem_1[64], int mem_2[64]) {
 ...
 for_top: for (int i = 0; i < 64; ++i) {
 my_label:{

 }
 }
}

```

- 図5-2 に、このコード例がどのように [Directive] タブに表示されるかを示します。[Directive] タブは、[Explorer] タブでソース コードを選択し、補足エリア (GUI の右側) で [Directive] タブをクリックすると表示できます。

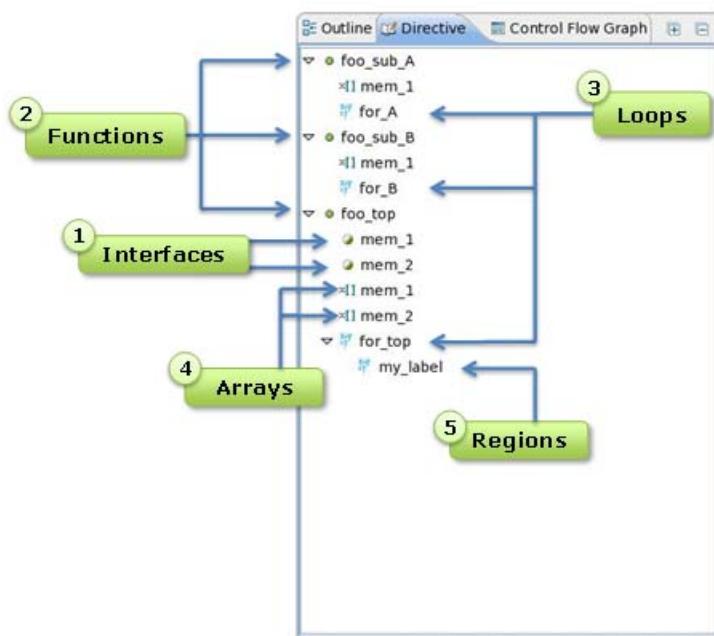


図 5-2 : [Directive] タブのオブジェクト

- インターフェイス : インターフェイスに適用される指示子は、そのインターフェイスのオブジェクト (関数、パラメーター、グローバルまたは戻り値) に適用されるか、何にも適用されません。
- 関数 : 関数に適用される指示子は、その関数のスコープ内のオブジェクトすべてに対して実行されます。指示子の影響は、その関数階層の次のレベルで停止します。ただし、すべてが繰り返しフラットにされてから展開される PIPELINE 指示子の場合や指示子で -recursive オプションがサポートされていて適用される場合などは例外です。
  - 図5-2 の例の場合、foo\_top に適用された指示子は foo\_sub\_A または foo\_sub\_B の操作に影響しません (上記の例外は除く)。
- ループ : ループに適用される指示子は、そのループのスコープ内のオブジェクトすべてに影響します。
  - ループに MERGE 指示子が使用される場合、指示子はループ自体ではなく、下位ループに適用されます。ループは同じ階層レベルのループとは結合されず、下位ループと結合されます。
  - PIPELINE 指示子もループ内のオブジェクトに適用されます。これは、ループ自体のパイプライン処理と同じです。

4. 配列：指示子は配列に直接適用できます。この場合、配列自体にのみ適用できるほか、複数の配列を含む関数、ループ、領域に適用でき、指示子は囲まれているすべての配列に適用されます。
5. 領域：コードの領域は波括弧 {} で囲むと作成できます。ラベル付き領域に適用される指示子は、その領域内のオブジェクトに適用されます。

**注記** : Tcl コマンドを使用して指示子を適用するには、上記の例のようにループおよび領域にラベルが必要です (上記の例ではループ ラベル `for_top` と領域ラベル `my_label`)。

インターフェイスおよび配列以外のその他のオブジェクト (関数、ループ、領域) は、複数の文を含むコードの「エリア」を示します。このため、本章のコマンド ページにはターゲットを「ロケーション」としてリストしています。指示子がロケーションに適用されると、明示的に指定がない限り、そのロケーション内のすべてのオブジェクトに適用されることに注意してください。

## コマンドおよびプログラマ

Vivado HLS の最適化は、前のセクションで説明したように、主にコード内のロケーションに基づいています。最適化を指定するこのモデルでは、コード内でのプログラマの使用がサポートされています。

この例の場合、プログラマは `for` ループを展開するためにコードに挿入されています。プログラマの後は、常に `AP` キーワードと指示子が指定されます。

```
for (int i = 0; i < 64; ++i) {
 #pragma AP UNROLL
 ...
}
```

C コードで入力される指示子は、C 仕様を使用するインプリメンテーションすべてで共有されます。C 仕様にすべての最適化指示子が含まれるようになる必要があることもあります、ほとんどの場合は、プログラマをソースと分けて、別のソリューションが作成できるようになる方が望まれます (プログラマが使用されると、そのソースを使用するソリューションすべてに対して同じ最適化が実行されるからです)。

上記の例では、ループにラベルが付いているか、`UNROLL` 指示子を GUI の [Directives] タブを使用してソース コードに追加できたとすると、コマンド ライン インターフェイスに `set_directive_loop_unroll` コマンドを使用できます。

Vivado HLS で `help` コマンドを使用すると、次のように各最適化に関連するプログラマがリストされます。

```
>autopilot help
...
set_directive_allocation - Directive ALLOCATION
set_directive_array_map - Directive ARRAY_MAP
set_directive_array_partition - Directive ARRAY_PARTITION
set_directive_array_reshape - Directive ARRAY_RESHAPE
set_directive_array_stream - Directive ARRAY_STREAM
set_directive_dataflow - Directive DATAFLOW
set_directive_dependence - Directive DEPENDENCE
set_directive_expression_balance - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline - Directive INLINE
set_directive_interface - Directive INTERFACE
set_directive_latency - Directive LATENCY
set_directive_loop_flatten - Directive LOOP_FLATTEN
set_directive_loop_merge - Directive LOOP_MERGE
set_directive_loop_tripcount - Directive LOOP_TRIPCOUNT
set_directive_occurrence - Directive OCCURRENCE
set_directive_pipeline - Directive PIPELINE
set_directive_platform - Directive PLATFORM
set_directive_power - Directive POWER
set_directive_protocol - Directive PROTOCOL
```

```

set_directive_resource - Directive RESOURCE
set_directive_top - Directive TOP
set_directive_unroll - Directive UNROLL
...

```

どれを最適化するか、コマンドまたはプラグマが何かを決定する一番簡単な方法は、次のとおりです。

1. コードを GUI で開きます。
2. 最適化可能なオブジェクトすべてが表示された [Directive] タブ (図5-1) をクリックします。
3. オブジェクトを選択して、右クリックで指示子を指定します。
  - a. [Destination] を [Into Directive File] にし、コマンドが constraints/directive.tcl に含まれるようにします。

または

- b. [Destination] を [Into Source File] にし、プラグマがコードに直接挿入されるようにします。

## 高位合成コマンド

### add\_file

#### 構文

```
add_file [OPTIONS] <src_files>
```

#### 説明

add\_file コマンドは、デザインソースファイルを現在のプロジェクトに追加します。このデザインソースに含まれるヘッダーファイルは現在のディレクトリで自動的に検索されます。これ以外のディレクトリに保存されているヘッダーファイルを使用するには、-cflags オプションを使用して、検索パスにそれらのディレクトリを追加します。

<src\_files> - デザイン記述のあるソースファイルのリスト

#### オプション

-tb - デザインのテストベンチの一部として使用されるファイルを指定します。cosim\_design コマンドにより合成後検証が実行されるとき、これらのファイルは合成はされませんが使用されます。このオプションが使用されているときは、ソースファイルのリストにデザインファイルを含めることはできません。デザインファイルおよびテストベンチファイルを追加するには、別の add\_file コマンドを使用します。

-cflags <string> - GCC コンパイル オプションの文字列

-type (c | sc) - ソースファイルが C/C++ (c) または SystemC (sc) であるかどうかを指定します。デフォルトは C/C++ です。

## Pragma

`add_file` コマンドと同等の `pragma` はありません。

### 例

次の例は、プロジェクトに 3 つのデザインファイルを追加します。

```
add_file a.cpp
add_file b.cpp
add_file c.cpp
```

1 コマンド行で複数のファイルを追加することもできます。

```
add_file "a.cpp b.cpp c.cpp"
```

次の例は、`USE_RANDOM` というマクロを有効にするコンパイラ フラグと共に `SystemC` ファイルを追加し、ヘッダーファイルを検索するための追加のサブディレクトリ `./lib_functions` を指定しています。

```
add_file -type sc top.cpp -cflags "-DUSE_RANDOM -I./lib_functions"
```

`-tb` オプションはプロジェクトにテストベンチ ファイルを追加するために使用します。この例では、1 つのコマンドで複数のファイルを追加します。追加されるファイルは、テストベンチの `a_test.cpp`、このテストベンチで読み込まれるデータ ファイル `input_stimuli.dat` および `out.gold.dat` です。

```
add_file -tb "a_test.cpp input_stimuli.dat out.gold.dat"
```

前の例にあるテストベンチのデータ ファイルが、たとえば `test_data` というディレクトリに保存されている場合は、個々のファイルを指定する代わりに、このディレクトリをプロジェクトに追加することができます。

```
add_file -tb a_test.cpp
add_file -tb test_data
```

---

## autoimpl

### 構文

```
autoimpl [OPTIONS]
```

### 説明

RTL のゲート レベル インプリメンテーションを作成するためのスクリプトを自動的に作成して実行するコマンドです。

デフォルトの `Vivado HLS` で生成されたスクリプトの代わりに、インプリメンテーションにユーザー定義のスクリプトを使用するよう指定することができます。

インプリメンテーション スクリプトはアクティブ ソリューションの `impl/<rtl>` というサブディレクトリにあります。デフォルトでは、`autoimpl` コマンドは、ゲート レベルのインプリメンテーションを作成するためこのディレクトリでスクリプトを実行します。ロジック合成を実行せず、スクリプトのみを作成するには `-setup` オプションを使用します。

EDK 環境内で使用する pcore インプリメンテーションを作成するには、**-export** および **-custom\_ports** オプションを使用します。**-xil\_coregen** オプションは、CORE Generator を起動して、ロジック合成の前に BRAM や浮動小数点ブロックなどの最適化されたコンポーネントを作成およびインスタンシエートします。

## オプション

**-export** - 該当ラッパーおよび外部アダプターを含めたデザインの追加インプリメンテーションが作成されます。これはコアモデルとして EDK にインポートすることができます(プロジェクトの pcore ディレクトリに直接コピー可能)。

**-par** - このオプションは、**-tool** オプションを使用して Synopsys 社の Synplify ツールを指定する場合にのみ使用することができます。このオプションが指定されていると、ISE Design Suite を使用して配置配線インプリメンテーションが実行されます(そうでない場合は配置配線は実行されません)。

**-rtl (verilog|vhdl)** - RTL インプリメンテーションにどの HDL を使用するかを指定します。デフォルトは **verilog** です。

**-setup** - このオプションが指定されている場合、アクティブソリューションの **impl/<rtl>** ディレクトリにインプリメンテーションファイルがすべて作成されますが、インプリメンテーションは実行されません。

**-tool** - ゲート レベル インプリメンテーションを作成するのにどの RTL 合成ツールを使用するかを指定します。このダイアログ ボックスには、次のようなオプションがあります。

- **ise** - ザイリンクス ISE Design Suite (デフォルト)
- **synplify** - Synopsys 社の Synplify

**-tool** オプションで追加ツール オプションを指定することもできます。このオプションはコマンド ラインでのみ使用でき、GUI では使用できません。たとえば、Synplify Professional 版の特定ライセンスを指定するには、次のように入力します。

```
autoimpl -tool "synplify_pro -licensetype synplifypro_xilinx"
-xil_coregen
```

また、具体的に実行ファイルを指定することもできます。たとえば、**synplify\_pro**、**synplify\_pro\_dp**、または **synplify\_premier** を指定する場合は、次のように入力します。

```
autoimpl - tool "synplify_premier"
```

BRAM や浮動小数点ブロックなど、RTL にコンポーネント用に最適化されたネットリストをインプリメントするには、このオプションでは CORE Generator ツールフローが使用されます。

## pragma

autoimpl コマンドと同等の **pragma** はありません。

## 例

この例では、インプリメンテーション ツール用のスクリプトがすべて作成されますが、インプリメンテーションは起動されません。

```
autoimpl -setup
```

次の例では、Synplify(デフォルトの Verilog RTL)を使用してデザインがインプリメントされます。

```
autoimpl -tool synplify
```

次の例では、ISE Design Suite を使用して VHDL をインプリメントするスクリプトが作成されますが、ISE Design Suite は実行されません。また、合成前に RTL にインスタンシエートされるメモリや FIFO などの最適化モジュールを作成するため、CORE Generator が実行され、EDK で使用するための pcore ディレクトリも作成されます。

```
autoimpl -rtl vhdl -setup -xil_coregen -export
```

## cosim\_design

### 構文

```
cosim_design [OPTIONS]
```

### 説明

元の C ベースのテストベンチを使用して、合成された RTL の合成後協調シミュレーションを実行するコマンドです。テストベンチのファイルは、`add_file -tb` で指定します。シミュレーションは、アクティブソリューションの `sim/<HDL>` というサブディレクトリで実行されます。<HDL> には `-rtl` オプションで選択される値が入ります。

`cosim_design` でデザインを検証するには、デザインで `ap_ctrl_hs` というインターフェイスモードを使用する必要があります。また、出力が書き込まれるとき、書き込み有効信号で識別される、`ap_vld`、`ap_ovld`、`ap_hs`、`ap_memory`、`ap_fifo` または `ap_bus` のいずれかのインターフェイスモードを各出力ポートで使用する必要があります。

### オプション

**-o** - C テストベンチおよび RTL ラッパーの最適化コンパイルを実行します。最適化を実行しない場合は、`cosim_design` でできるかぎり短時間でテストベンチがコンパイルされます。可能であれば、コンパイルに時間がかかる場合でも、ランタイムパフォーマンスを改善するために最適化を実行してください。

**注記** : 実行ファイルのランタイム速度が向上するかもしれません、ランタイムの改善はデザインに依存しています。ランタイム目的で最適化を行うと、大型関数のためにメモリ使用量が多くなる可能性があります。

**-argv <string>** - ビヘイビアーテストベンチの引数リストを指定します。<string> はメイン C 関数に渡されます。

**-coverage** - VCS シミュレーターを使用したシミュレーションの範囲を指定します。

**-ignore\_init <integer> - <integer>** で指定される最初のクロックサイクル数に対し、比較チェックを無効にします。これは、未知の値 (hX) で RTL が開始するタイミングを確認するのに便利です。

**-ldflags <string>** - 協調シミュレーション用にリンカーに渡す必要のあるオプションを指定します。通常は、C テストベンチのパス情報またはライブラリ情報渡すのに使用されます。

**-mfflags <string>** - SystemC シミュレーション用にリンカーに渡す必要のあるオプションを指定します。通常、コンパイルの速度を上げるために使用されます。

**-rtl (systemc | vhdl | verilog)** - C テストベンチでの検証にどの RTL を使用するかを選択します。Verilog および VHDL の場合は、**-tool** オプションを使用してシミュレーターを指定する必要があります。デフォルトは `systemc` です。

**-setup** - このオプションが指定されている場合、アクティブソリューションの `sim/<HDL>` ディレクトリにシミュレーションファイルがすべて作成されますが、シミュレーションは実行されません。

**-tool (vcs | modelsim | riviera)** - C テストベンチを使用して RTL を協調シミュレーションするのに使用するシミュレーターを選択します。SystemC の協調シミュレーションにはツールを指定する必要はありません。Vivado HLS に含まれている SystemC カーネルが使用されます。

**-trace\_level (none|all)** - 実行される VCD 出力レベルを指定します。 **all** を指定すると、すべてのポートおよび信号が VCD ファイルに保存されます。VCD ファイルは、シミュレーションが実行されると、現在のソリューションの `sim/<HDL>` ディレクトリに保存されます。デフォルトは **none** です。

## pragma

`cosim_design` コマンドと同等の `pragma` はありません。

## 例

SystemC RTL を使用した検証が実行されます。

```
cosim_design
```

Verilog RTL を検証するため VCS シミュレーターを使用し、VCD カバレッジ機能を有効にして、VCD フォーマットですべての信号を保存します。

```
cosim_design -tool VCS -rtl verilog -coverage -trace_level all
```

次の例では、ModelSim を使用して VHDL RTL が検証され、値 5 および 1 がテストベンチ関数に渡され RTL 検証で使用されます。

```
cosim_design -tool modelsim -rtl vhdl -argv "5 1"
```

次の例では、SystemC RTL 用に最適化されたシミュレーションモデルが作成されますが、シミュレーションは実行されません。シミュレーションを実行するには、アクティブソリューションの `sim/systemc` ディレクトリで `run.sh` を実行します。

```
cosim_design -O -setup
```

## autosyn

## 構文

```
autosyn
```

## 説明

`autosyn` コマンドは、アクティブソリューションの Vivado HLS データベースを合成します。このコマンドはアクティブソリューションに対してのみ実行されます。データベースにあるエラボレートされたデザインは、設定されている制約に基づいてスケジュールされ RTL にマップされます。

## pragma

`autosyn` コマンドと同等の `pragma` はありません。

## 例

最上位デザインで Vivado HLS 実行します。

autosyn

---

## close\_project

### 構文

`close_project`

### 説明

`close_project` コマンドは現在のプロジェクトを閉じ、そのプロジェクトは Vivado HLS セッションでアクティブではなくなります。プロジェクトやソリューション特定のコマンドを入力することを防ぐコマンドですが、新しいプロジェクトを開いたり作成すると、作業中のプロジェクトは自動的に閉じるので、実際には必要ありません。

### pragma

`close_project` コマンドと同等の `pragma` はありません。

### 例

現在のプロジェクトを閉じます。結果はすべて自動的に保存されます。

`close_project`

---

## close\_solution

### 構文

`close_solution`

### 説明

`close_solution` コマンドは現在のソリューションを閉じ、そのソリューションは Vivado HLS セッションでアクティブではなくなります。ソリューション特定のコマンドを入力することを防ぐコマンドですが、新しいソリューションを開いたり作成すると、作業中のソリューションは自動的に閉じるので、実際には必要ありません。

### pragma

`close_solution` コマンドと同等の `pragma` はありません。

### 例

現在のソリューションを閉じます。結果はすべて自動的に保存されます。

```
close_solution
```

---

## config\_array\_partition

### 構文

```
config_array_partition [OPTIONS]
```

### 説明

このコマンドを使用して、配列分割のデフォルト動作を指定します。

### オプション

**-auto\_partition\_threshold** <int> - 配列 (定数指數のないものも含む) を自動的に分割するためのしきい値を設定します。指定されたしきい値よりもエレメント数が少ない配列は、インターフェイス/コア仕様が配列に適用されない限り、個々のエレメントに自動的に分割されます。デフォルトは 4 です。

**-auto\_promotion\_threshold** <int> - 定数指數を持つ配列を自動的に分割するためのしきい値を設定します。指定されたしきい値よりもエレメント数が少なく、また定数指數を持つ(指數が変数ではない)配列は、個々のエレメントに自動的に分割されます。デフォルトは 64 です。

**-exclude\_extern\_globals** - スループットに基づいた自動分割から外部グローバル配列を除外します。デフォルトでは、**-throughput\_driven** オプションが選択されているとき外部グローバル配列は分割されます。**-throughput\_driven** が選択されていないと、このオプションは効果がありません。

**-include\_ports** - I/O 配列の自動分割を有効にします。1 つの配列 I/O ポートが複数のポートに分割され、各ポートのサイズは個々の配列エレメントのサイズになります。

**-scalarize\_all** - デザインにあるすべての配列を個々のエレメントに分割します。

**-throughput\_driven** - スループットに基づいて配列の自動分割を有効にします。配列を個々のエレメントに分割することで指定のスループット要件を満たすことができるかどうかは Vivado HLS で自動的に判断されます。

### pragma

config\_array\_partition コマンドと同等の **pragma** はありません。

### 例

この例では、デザインの、グローバル配列を除くエレメント数が 12 未満の配列がすべて自動的に個々のエレメントに分割されます。

```
config_array_partition auto_partition_threshold 12 -exclude_extern_globals
```

この例では、スループットを改善するため、関数インターフェイスの配列を含む、分割する配列が Vivado HLS で自動的に判断されます。

```
config_array_partition -throughput_driven -include_ports
```

グローバル配列を含む、デザインのすべての配列が個々のエレメントに分割されます。

```
config_array_partition -scalarize_all
```

---

## config\_bind

### 構文

```
config_bind [OPTIONS]
```

### 説明

このコマンドを使用して、マイクロアーキテクチャ バインディングのデフォルト オプションを指定します。バインディングは、加算、乗算、シフトなどの演算子が特定の RTL インプリメンテーションにマップされるプロセスのこととを指します。たとえば、乗算演算は組み合わせ、またはパイプライン RTL 乗算器としてインプリメントされます。

### オプション

**-effort (low|medium|high)** - ランタイムと最適化のトレードオフと制御する最適化エフォート レベルを設定します。デフォルトは **medium** です。

**low** はランタイムを向上させます。最適化があまり実行できないようなケース、たとえば、各分岐のすべての if-else 文に相互排他的な演算子があり、演算子の共有ができないような場合にメリットがあります。

**high** は、ランタイムが長くなりますが、通常結果の質を向上させることができます。

**-min\_op <string>** - Vivado HLS で特定演算子のインスタンス数を最小限に抑えるようにするオプションです。演算子が複数コードにある場合、最も少ない数の RTL リソース (コア) で演算子が共有されるようになります。

このコマンド オプションの引数として次の演算子を指定することができます。

- **add** - 加算
- **sub** - 減算
- **mul** - 乗算
- **icmp** - 整数比較
- **sdiv** - 符号付き除算
- **udiv** - 符号なし除算
- **srem** - 符号付き剰余
- **urem** - 符号なし剰余
- **lshr** - 論理右シフト
- **ashr** - 四則演算右シフト
- **shl** - 左シフト

### pragma

config\_bind コマンドと同等の **pragma** はありません。

## 例

この例では、バインディングプロセスでのエフォートレベルを上げ、演算子をインプリメントためいろいろなオプションを試し、よりよいリソース使用率でデザインが生成されるようになります。

```
config_bind -effort high
```

この例では、乗算演算子の数を最小限に抑え、最も少ない数の乗算器で RTL にインプリメントされます。

```
config_bind -min_op mul
```

---

## config\_dataflow

### 構文

```
config_dataflow [OPTIONS]
```

### 説明

このコマンドは、データフローパイプラインのデフォルト動作を指定します (set\_directive\_dataflow コマンドによりインプリメントされる)。このコンフィギュレーションコマンドを使用すると、デフォルトのチャネルメモリタイプおよびその深さを指定することができます。

### オプション

**-default\_channel (fifo|pingpong)** - データフローパイプラインが使用されているとき、関数間またはループ間のデータをバッファーするのに、ピンポン形式でコンフィギュレーションされている RAM メモリがデフォルトで使用されます。ストリーミングデータが使用されている場合 (データの読み出しおよび書き込みがが常に逐次行われる場合) は、FIFO メモリのほうが効果的で、デフォルトのメモリタイプとして選択することができます。

注記 : FIFO アクセスを実行するには、set\_directive\_array\_stream コマンドを使用して配列をストリーミングに設定する必要があります。

**-fifo\_depth<integer>** - FIFO のデフォルトの深さを整数値で指定します。このオプションは、ピンポン形式のメモリが使用されている場合は効果がありません。チャネルで使用する FIFO が指定されていない場合は、最大出力または入力 (いずれかの大きいほう) のサイズに設定されます。場合によっては、この設定が控えめすぎて必要以上に大きな FIFO が作成される可能性があります。このオプションは FIFO のサイズが必要以上に大きい場合に使用できます。このオプションを間違って使用すると、デザインが正しく動作しなくなる可能性があるので、使用にあたっては注意してください。

### pragma

config\_dataflow コマンドと同等の pragma はありません。

## 例

デフォルトチャネルをピンポン形式のメモリから FIFO に変更します。

```
config_dataflow -default_channel
```

デフォルトチャネルをピンポン形式のメモリから幅が 6 の FIFO に変更します。

注記: エレメント数が 6 を超える FIFO がデザインインプリメンテーションで必要な場合、この設定では RTL 検証エラーが発生します。ユーザーはこのオプションを上書きすることができますが、使用にあたっては注意が必要です。

```
config_dataflow -default_channel fifo -fifo_depth 6
```

## config\_interface

### 構文

```
config_interface [OPTIONS]
```

### 説明

config\_interface コマンドは、インターフェイス合成中に各関数引数の RTL ポートのインプリメンテーションに使用されるデフォルトインターフェイスを指定します。関数引数には、値渡しの変数、ポインター、配列、参照渡し変数(入力言語で使用できるもの)を使用することができます。

さらに、config\_interface コマンドは、start や done などの関数レベルの制御のデフォルトインターフェイスを指定したり、関数で使用されているグローバル変数を RTL デザインのポートとして表すのに使用することができます。

関数引数に none が明示的に指定されている場合や、互換性のないインターフェイスタイプが指定されている場合は、デフォルトのインターフェイスが使用されます。インターフェイスタイプがすべて含まれているリストは次を参照してください。また、各インターフェイスの詳細は、「高位合成ユーザーガイド」の章を参照してください。

### インターフェイスタイプ

**ap\_none** - このインターフェイスでは、追加ハンドシェイクや同期ポートは提供されず、また、配列を除く、すべての関数引数に適用することができます。配列引数を除く、すべての読み出し専用引数(入力ポート)のデフォルトインターフェイスタイプです。

**ap\_ack** - 配列を除くすべての関数引数に対し指定することができます。また、RTL ブロックにより入力データが読み出されたこと、またはダウンストリームの RTL ブロックにより出力データが読み出されたことを確認するための追加承認ポートを提供します。

**ap\_vld** - 配列を除くすべての関数引数に対し指定することができます。また、入力データが有効で読み出し可能なとき、または出力データが有効なときを示すための追加データ有効ポートを提供します。

**ap\_ovld - ap\_vld** インターフェイスと同じですが、書き込み専用引数(RTL 出力ポート)に対してのみ提供される点が異なります。配列引数を除く、すべての書き込み専用引数のデフォルトインターフェイスタイプです。

**ap\_hs** - 完全 2 方向の承認および有効ハンドシェイクによってサポートされている RTL ポートで各引数をインプリメントします。これは、配列を除くすべての関数引数に対し指定することができます。

**ap\_fifo** - このインターフェイスは、ポインター、配列、または参照渡しの引数に対し指定することができます。このインターフェイスは、関連付けられた Empty、Full、Data Valid 信号を使用して FIFO への読み出しおよび書き込みとしてデータアクセスをインプリメントします。

**ap\_bus** - このインターフェイスは、一般的な DMA インターフェイスと同じような汎用バスアクセスとして、ポインターおよび参照渡し変数をインプリメントします。

**ap\_memory** - このインターフェイスは配列引数のデフォルト タイプで、配列引数に対してのみ指定することができます。このインターフェイスは、関連付けられたアドレス、チップ イネーブル、ライト イネーブル制御信号を使用して、RAM のデータ値として配列エレメントにアクセスする RTL インプリメンテーションになります。配列に対し

テクノロジ ライブラリにある、使用される RAM リソースを確認するには、`set_directive_resource` コマンドを使用します。これで、使用可能なポート数およびどの制御信号をインプリメントするのかを指定できます。

`ap_ctrl_none` および `ap_ctrl_hs` - 関数の戻り引数にのみ指定することができます。`ap_ctrl_hs` はデフォルトで、入力 start 信号、出力 idle および done 信号といった関数レベルの制御信号を追加します。関数の戻り引数がある場合は、done 信号で戻り値が有効であるかどうかを確認できます。`ap_ctrl_none` タイプは、これらの信号がデザインに追加されないようにします。

## オプション

`-all (ap_none|ap_stable|ap_ack|ap_vld|ap_ovld|ap_hs|ap_ctrl_none|`

`ap_ctrl_hs|ap_fifo|ap_bus|ap_memory)` - すべてのポート タイプ (入力、出力、入出力) および関数レベルのハンドシェイクのデフォルト インターフェイス タイプです。デフォルトは `ap_none` です。

`-clock_enable` - クロック イネーブル ポート (`ap_ce`) をデザインに追加します。クロック イネーブルがアクティブ Low のときは、すべてのクロック操作が停止になり、すべての順次操作が無効になります。

`-expose_global` - I/O ポートとしてグローバル変数を表します。変数がグローバルとして作成されていても、すべての読み出しおよび書き込みアクセスがローカルになっている場合、リソースはデザインで作成され、RTL に I/O ポートの必要はありません。しかし、グローバル変数が外部ソースとして扱われる場合、または RTL ブロック外のデスティネーションである場合は、このオプションを使用してポートを作成する必要があります。

`-in (ap_none|ap_stable|ap_ack|ap_vld|ap_hs|ap_fifo|ap_bus|ap_memory)` - すべての入力 (読み出しのみ) 引数のデフォルト インターフェイス タイプを指定します。デフォルトは `ap_none` です。

`-inout (ap_none|ap_stable|ap_ack|ap_vld|ap_ovld|ap_hs|ap_fifo|ap_bus|ap_memory)` - すべての入出力 (読み出し/書き込み) 引数のデフォルト インターフェイス タイプを指定します。デフォルトは `ap_none` です。

`-out (ap_none|ap_ack|ap_vld|ap_ovld|ap_hs|ap_fifo|ap_bus|ap_memory)` - すべての出力 (書き込みのみ) 引数のデフォルト インターフェイス タイプを指定します。デフォルトは `ap_none` です。

`-return (ap_ctrl_none|ap_ctrl_hs)` - 関数レベルのハンドシェイクの使用を指定します。デフォルトは `ap_ctrl_hs` です。

## pragma

`config_interface` コマンドと同等の `pragma` はありません。

## 例

入力ポートに承認インターフェイスを使用します。

```
config_interface -in ap_ack
```

すべての出力にハンドシェイク インターフェイスを使用するように指定します。

```
config_interface -out ap_hs
```

関数レベルのハンドシェイク信号をインプリメントしないようにします。

```
config_interface -return ap_ctrl_none
```

有効なインターフェイスとしてすべての I/O ポート (読み出し/書き込み) をコンフィギュレーションし、クロック イネーブル ポートをデザインに追加します。

```
config_interface -inout ap_vld -clock_enable
```

# config\_rtl

## 構文

```
config_rtl [OPTIONS] <model_name>
```

## 説明

このコマンドは、使用されるリセットのタイプ、ステート マシンのエンコーディングなど、出力 RTL のさまざまな属性をコンフィギュレーションし、RTL でユーザー ID 情報を使用できるようにします。

デフォルトでは、オプションは最上位デザイン、およびそのデザイン内のすべての RTL ブロックに適用されます。また、特定 RTL モデルを指定することも可能です。

<model\_name> - コンフィギュレーションする RTL モデル名を指定します。何も指定されていない場合は、最上位デザイン (およびそのサブ ブロックすべて) が対象になります。

## オプション

**-header <string> - <string>** ファイルの内容をコメントとしてすべての出力 RTL およびシミュレーションファイルの冒頭に挿入します。これで出力 RTL ファイルにユーザー指定の ID 情報が含まれるようになります。

**-prefix <string>** - すべての RTL エンティティ / モジュール名に追加する接頭辞を指定します。

**-reset (none|control|state|all)** - C コードで初期化される変数は常に RTL でも (つまりビットストリームでも) 同じ値に初期化されます。ただし、この初期化はパワーオン時にのみ実行され、デザインにリセットが適用されたときには繰り返されません。-reset オプションを使用して適用された設定により、レジスタ/メモリのリセット方法が決まります。デフォルトは **control** です。

- **none** - デザインにリセットは追加させません。
- **control** - ステート マシンで使用される制御レジスタや、I/O プロトコル信号の生成に使用される制御レジスタをリセットします。
- **state** - C コードのスタティック/グローバル変数から派生する制御レジスタおよびレジスタ/メモリをリセットします。C コードのスタティック/グローバル変数はその初期化値にリセットされます。
- **all** - デザインのすべてのレジスタおよびメモリをリセットします。C コードのスタティック/グローバル変数はその初期化値にリセットされます。

**-reset\_async** - すべてのレジスタで非同期リセットが使用されるようになります。このオプションが指定されていない場合は、同期リセットが使用されます。

**-reset\_level (low|high)** - リセット信号の極性がアクティブ Low またはアクティブ High になります。デフォルトは **high** です。

**-encoding (bin|onehot|gray)** - デザインのステート マシンで使用されるエンコーディング形式を指定します。デフォルトは **bin** です。

## pragma

config\_rtl コマンドと同等の **pragma** はありません。

## 例

この例では、非同期のアクティブ Low リセット信号を使用してすべてのレジスタがリセットされるように出力 RTL をコンフィギュレーションします。

```
config_rtl -reset all -reset_async -reset_level low
my_message.txt ファイルの内容をコメントとしてすべての RTL 出力ファイルに追加します。
config_rtl -header my_message.txt
```

---

## config\_schedule

### 構文

```
config_schedule [OPTIONS]
```

### 説明

Vivado HLS で実行されるスケジューリングのデフォルト タイプをコンフィギュレーションします。

### オプション

**-effort (high|medium|low)** - スケジューリング中に使用するエフォートを指定します。デフォルトは **medium** です。**low** は、ランタイムを改善しますが、デザインインプリメンテーションに改善の余地があまりないケースで指定するとよいでしょう。**high** は、ランタイムが長くなりますが、通常結果の質を向上させることができます。

**-verbose** - スケジューリングで指示子や制約を満たすことができないときのクリティカルパスを出力します。

### pragma

config\_schedule コマンドと同等の **pragma** はありません。

### 例

ランタイムを短縮させるためデフォルトのスケジュールエフォートを **low** に変更します。

```
config_schedule -effort low
```

---

## create\_clock

### 構文

```
create_clock -period <number> [OPTIONS]
```

## 説明

`create_clock` コマンドはアクティブソリューションの仮想クロックを作成します。このコマンドはアクティブソリューションに対してのみ実行されます。そのクロック周期は自動最適化(指定クロック周期で可能な限りの数の演算をチーブ接続)を駆動する制約です。

C および C++ デザインの場合は、シングルクロックのみがサポートされています。SystemC デザインの場合は、複数の指定クロックを作成し、`set_directive_clock` コマンドを使用して異なる SC\_MODULE に供給することができます。

## オプション

**-name <string>** - クロック名を指定します。名前が指定されていない場合は、デフォルト名が使用されます。

**-period <number>** - ナノ秒(ns)またはMHzでクロック周期を指定します。単位が指定されていない場合は、デフォルトでnsが使用されます。周期が指定されていない場合は、デフォルトで10nsの周期が使用されます。

## pragma

`create_clock` コマンドと同等の `pragma` はありません。

## 例

50ns のクロック周期を指定します。

```
create_clock -period 50
```

この例では、デフォルト周期 10ns でクロックを作成します。

```
create_clock
```

SystemC デザインの場合は、複数の指定クロックを作成することができます(`set_directive_clock` コマンドを使用)。

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
```

クロック周波数を MHz で指定します。

```
create_clock -period 100MHz
```

## delete\_project

### 構文

```
delete_project <project>
```

## 説明

`delete_project` コマンドはプロジェクトに関連付けられているディレクトリを削除します。

対応するプロジェクトディレクトリ *<project>* が有効なVivado HLS プロジェクトであることを確認してから、削除します。現在の作業ディレクトリにディレクトリ *<project>* が存在しない場合は、このコマンドは効果がありません。

*<project>* - プロジェクト名を指定します。

## pragma

`delete_project` コマンドと同等の `pragma` はありません。

## 例

ディレクトリ `./Project_1` およびそれに含まれるものすべて削除して `Project_1` を削除します。

```
delete_project Project_1
```

---

## delete\_solution

### 構文

```
delete_solution <solution>
```

### 説明

`delete_solution` コマンドは、アクティブプロジェクトからソリューションを削除し、プロジェクトディレクトリから *<solution>* サブディレクトリを削除します。

プロジェクトディレクトリにソリューションが存在しない場合は、このコマンドは効果がありません。

*<solution>* - 削除するソリューションの名前を指定します。

## pragma

`delete_solution` コマンドと同等の `pragma` はありません。

## 例

アクティブプロジェクトから `Solution_1` というソリューションを削除し、アクティブプロジェクトからサブディレクトリ `Solution_1` を削除します。

```
delete_solution Solution_1
```

# elaborate

## 構文

```
elaborate [OPTIONS]
```

## 説明

ソースファイルをコンパイルし、アクティブソリューションのVivado HLS データベースを作成します。このコマンドはアクティブソリューションに対してのみ実行されます。

設定されている指示子に基づいて、関数、ループ、および配列の初期プロセスが一部実行されます。

## オプション

**-effort (low|medium|high)** - デフォルトのエフォート レベルは **medium** で、ランタイムおよび QoR (結果の品質) のバランスを取って最適化が行われます。**high** は、ベストの QoR が得られますが、実行に時間がかかります。**low** は、ベストのランタイムが得られますが、最適なデザインが得られるとは限りません。最適化の余地があまりないケースでのみ使用するようにしてください。

## pragma

elaborate コマンドと同等の **pragma** はありません。

## 例

すべての入力ソースファイルおよびライブラリを指定した後、内部モデルを作成するためデザインをエラボレートします。このモデルは RTL に合成することができます。

```
elaborate
```

エフォート レベルを **high** にしてデザインをエラボレートします。

```
elaborate -effort high
```

# help

## 構文

```
help [OPTIONS] <cmd>
```

## 説明

<cmd> に何も指定しない場合、**help** コマンドはVivado HLS Tcl コマンドをすべてリストします。

Vivado HLS コマンドを引数に入力すると、`help` コマンドはその特定コマンドの情報を表示します。有効なVivado HLS コマンドに対しコマンド引数を入力するときタブキーを押すと自動完了します。

## オプション

`<cmd>` - ヘルプを表示させたいコマンド名を入力します。

### pragma

`help` コマンドと同等の `pragma` はありません。

### 例

`add_file` コマンドのヘルプを表示します。

```
help add_file
```

Vivado HLS で使用されるすべてのコマンドおよび指示子のヘルプを表示します。

```
help
```

---

## list\_core

### 構文

```
list_core [OPTIONS]
```

### 説明

現在インストールされているライブラリにあるコアをすべてリストします。コアは、出力 RTL に加算、乗算、メモリなどの演算をインプリメントするために使用するコンポーネントです。

エラボレーションの後、RTL の演算は内部データベースに演算子として表示されます。スケジューリング中、RTL デザインをインプリメントするため、演算子はライブラリのコアにマップされます。複数の演算子が 1 つのコアの同じインスタンスにマップされ、同じ RTL リソースを共有することも可能です。

次の関連オプションを使用して、`list_core` コマンドでは使用可能な演算子およびコアを」リストすることができます。

- `operation` - 各演算をインプリメントするために、ライブラリで使用可能なコアを表示します。
- `type` - 使用可能なコアをタイプ別に表示します。たとえば、論理演算をインプリメントするタイプ、メモリ/ストレージをインプリメントするタイプなどです。

オプションが指定されていない場合は、ライブラリのコアがすべて表示されます。

`list_core` コマンドで表示される情報は、特定演算を特定コアにインプリメントするため `set_directive_resource` コマンドで使用することができます。

## オプション

**-operation (opers)** - ライブラリにある、指定されている演算をインプリメントすることができるコアをリストします。演算のリストは次のとおりです。

- **add** - 加算
- **sub** - 減算
- **mul** - 乗算
- **udiv** - 符号なし除算
- **urem** - 符号なし剰余
- **srem** - 符号付き剰余
- **icmp** - 整数比較
- **shl** - 左シフト
- **lshr** - 論理右シフト
- **ashr** - 四則演算右シフト
- **mux** - マルチプレクサー
- **load** - メモリ読み出し
- **store** - メモリ書き込み
- **fiforead** - FIFO 読み出し
- **fifowrite** - FIFO 書き込み
- **fifobread** - 非ブロッキング FIFO 読み出し
- **fifobwrite** - 非ブロッキング FIFO 書き込み

**-type (functional\_unit|storage|connector|interface|ip\_block)** - 指定されているタイプのコアのみをリストします。

- **functional\_unit** - 加算、乗算、比較など、標準 RTL 演算をインプリメントするコア。
- **storage** - レジスタやメモリなどストレージエレメントをインプリメントするコア。
- **connector** - デザイン内のコネクティビティをインプリメントするコア。直接接続やストリーミングストレージエレメントがこれに含まれます。
- **interface** - IP 生成時に最上位デザインを接続するのに使用されるインターフェイスをインプリメントするコア。これらのインターフェイスは IP 生成フロー (ザイリンクス EDK) で使用される RTL ラッパーにインプリメントされます。
- **ip\_block** - ユーザーによって追加されている任意の IP コア。

## pragma

list\_core コマンドと同等の pragma はありません。

## 例

この例では、現在インストールされているライブラリにあるコアの中で、**add** 演算をインプリメントすることができるコアをすべてリストします。

```
list_core -operation add
```

この例では、ライブラリにある使用可能なメモリ (ストレージ) コアがすべてリストされます。使用可能なメモリの 1 つを使用して配列をインプリメントするには、**set\_directive\_resource** コマンドを使用します。

```
list_core -type storage
```

---

## list\_part

### 構文

```
list_part [OPTIONS]
```

### 説明

このコマンドは、サポートデバイスファミリ、またはあるファミリのサポートパートを表示します。オプションが指定されていない場合は、サポートファミリがすべてリストされます。ファミリのパートをリストするには、コマンドにオプションを指定しないときに表示されるサポートファミリの中から1つ選択し、それをオプションに指定すると、そのファミリのサポートパートがリストされます。

### pragma

list\_part コマンドと同等の pragma はありません。

### 例

次の例では、サポートファミリがすべてリストされます。

```
list_part
```

次の例では、virtex6 のパートがリストされます。

```
list_part virtex6
```

---

## open\_project

### 構文

```
open_project [OPTIONS] <project>
```

### 説明

open\_project コマンドは、既存プロジェクトを開くか、または新規プロジェクトを作成します。

Vivado HLS の1セッションでアクティブなプロジェクトは1つだけです。1プロジェクトには複数のソリューションを含めることができます。プロジェクトは close\_project コマンドで閉じるか、または、open\_project コマンドで別のプロジェクトを開くと、このプロジェクトを閉じることができます。 delete\_project コマンドは、プロジェクトディレクトリおよびそれに関連付けられているソリューションをすべて完全にディスクから削除します。

<project> - プロジェクト名を指定します。

## オプション

**-reset** - 既存のプロジェクトデータを削除してプロジェクトをリセットします。このオプションは、Tclスクリプトを使用してVivado HLSを実行しているときに使用してください。それ以外の場合は、`add_file` または `add_library` コマンドが新しく実行されるたびに追加ファイルが既存データに追加されます。

デザインソースファイル、ヘッダーファイルの検索パス、最上位関数に関するプロジェクト情報は削除されます。関連ソリューションディレクトリおよびファイルは保持されますが、結果が無効になってしまっている可能性があります。`delete_project` コマンドも **-reset** オプションと同じ効果があり、すべてのソリューションデータを削除します。

### pragma

`open_project` コマンドと同等の `pragma` はありません。

### 例

`Project_1` という名前の新規または既存プロジェクトを開きます。

```
open_project Project_1
```

プロジェクトを開き、既存データをすべて削除します。これは Tclスクリプトを使用している場合に使用してください。既存プロジェクトデータにソースまたはライブラリファイルが追加されるのを防ぎます。

```
open_project -reset Project_2
```

## open\_solution

### 構文

```
open_solution [OPTIONS] <solution>
```

### 説明

`open_solution` コマンドは、アクティブプロジェクトで既存のソリューションを開いたり、新規ソリューションを作成します。アクティブプロジェクトがないときにソリューションを開いたり作成しようとすると、エラーが発生します。Vivado HLS の 1セッションでアクティブなソリューションは 1つだけです。

現在のプロジェクトディレクトリの下に各ソリューションのサブディレクトリがあり管理されています。ソリューションが現在の作業ディレクトリに存在しない場合は、新しいソリューションが作成されます。ソリューションは `close_solution` コマンドで閉じるか、または、`open_solution` コマンドで別のソリューションを開くと、このソリューションを閉じることができます。`delete_solution` コマンドを使用すると、プロジェクトからソリューションが削除され、対応するサブディレクトリも削除されます。

*<solution>* - ソリューション名を指定します。

## オプション

**-reset** - ソリューションが既に存在する場合は、ソリューションデータがリセットされます。ライブラリ、制約、および指示子に関する前のソリューション情報は削除されます。合成、検証、インプリメンテーションの結果も削除されます。

### pragma

`open_solution` コマンドと同等の `pragma` はありません。

### 例

アクティブプロジェクトに、`solution_1` という名前の新規ソリューションを作成するか、既存のソリューションを開きます。

```
open_solution Solution_1
```

アクティブプロジェクトにソリューションを開き、既存データをすべて削除します。これは Tcl スクリプトを使用している場合に使用してください。既存ソリューションデータにデータが追加されるのを防ぎます。

```
open_solution -reset Solution_2
```

## set\_clock\_uncertainty

### 構文

```
set_clock_uncertainty <uncertainty> <clock_list>
```

### 説明

`set_clock_uncertainty` コマンドは、`create_clock` で定義されているクロック周期のマージンを設定します。マージンは、実質のクロック周期を作成するためクロック周期から差し引かれます。クロックのばらつきが定義されていない場合は、デフォルトでクロック周期の 12.5% となります。

Vivado HLS では、実効クロック周期に基づいてデザインが最適化され、ロジック合成および配線のためマージンが計算されます。このコマンドはアクティブソリューションに対してのみ実行されます。Vivado HLS では検証およびインプリメンテーションのすべての出力ファイルで指定されたクロック周期が使用されます。

`create_clock` コマンドで複数の名前が付けられたクロックが指定されている SystemC デザインの場合、各クロックを指定して、クロックごとに異なるばらつきを指定することができます。

`<uncertainty>` - マージンとして使用されるクロック周期を表す値で、単位はナノ秒 (ns) です。

`<clock_list>` - マージンとして使用されるクロック周期を表す値で、単位はナノ秒 (ns) です。

### pragma

`set_clock_uncertainty` コマンドと同等の `pragma` はありません。

## 例

クロックのばらつき/マージンを 0.5ns に指定します。Vivado HLS で使用可能な、実質クロック周期は 0.5ns 差し引いたものとなります。

```
set_clock_uncertainty 0.5
```

SystemC の例です。クロック ドメインが 2 つ作成されていて、各ドメインに異なるクロックのばらつきが指定されています。SystemC デザインではマルチ クロックがサポートされています。該当関数をクロックに適用するには `set_directive_clock` コマンドを使用します。

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
set_clock_uncertainty 0.5 fast_clock
set_clock_uncertainty 1.5 slow_clock
```

## set\_directive\_allocation

### 構文

```
set_directive_allocation [OPTIONS] <location> <instances>
```

### 説明

リソース割り当てのためインスタンスを制限します。特定関数または演算子をインプリメントするのに使用される RTL インスタンスの数を定義または制限します。

たとえば、`foo_sub` という関数のインスタンスが 4 つ C ソース コードにある場合、`set_directive_allocation` コマンドを使用して、最終 RTL では `foo_sub` のインスタンスを 1 つだけにすることができます(同じ RTL ブロックを使用して 4 つのインスタンスがすべてインプリメントされます)。

`<location>` - 関数[/ラベル] のフォーマットでのロケーション

`<instances>` - 関数または演算子

元の C コードにある任意の関数で、`set_directive_inline` または Vivado HLS でインライン化されていないものを指定することができます。

演算子のリストは次のとおりです(C ソース コードに演算のインスタンスがある場合)。

- **add** - 加算
- **sub** - 減算
- **mul** - 乗算
- **icmp** - 整数比較
- **sdiv** - 符号付き除算
- **udiv** - 符号なし除算
- **srem** - 符号付き剰余
- **urem** - 符号なし剰余
- **lshr** - 論理右シフト

- **ashr** - 四則演算右シフト
- **shl** - 左シフト

## オプション

**-limit <integer>** - RTL デザインで使用するインスタンスの最大数 (-**type** オプションで指定されているタイプのインスタンス)

**-type (function|operation)** - インスタンス タイプには **function** または **operation** を指定します。デフォルトは **function** です。

## pragma

C ソース コードの必要なロケーションの境界内に **pragma** を挿入する必要があります。

フォーマットおよびオプションは次のようになります。

```
#pragma AP allocation \
 instances=<Instance Name List> \
 limit=<Integer Value> \
 <operation, function>
```

## 例

関数 **foo** のインスタンスが複数あるデザイン **foo\_top** で、このコマンド (pragma も含める) は、RTL で **foo** のインスタンス数を 2 に制限します。

```
set_directive_allocation -limit 2 -type function foo_top foo
#pragma AP allocation instances=foo limit=2 function
```

次のコマンド (pragma も含める) は、**My\_func** のインプリメンテーションで使用される乗算器の数を 1 に制限します。

**注記** : この制限は、**My\_func** の下位関数にある乗算器には適用されません。下位関数にインプリメンテーションで使用される乗算器の数を制限するには、その下位関数に割り当て指示子を指定するか、下位関数を **My\_func** に埋め込みます。

```
set_directive_allocation -limit 1 -type operation My_func mul
#pragma AP allocation instances=mul limit=1 operation
```

## set\_directive\_array\_map

### 構文

```
set_directive_array_map [OPTIONS] <location> <array>
```

## 説明

このコマンドは小型配列を大型配列にマップします。通常、複数の `set_directive_array_map` コマンドを使用して、複数の小型配列を 1 つの大型配列にマップし、1 つの大型メモリ (RAM または FIFO) リソースにターゲットすることができます。

**-mode** オプションは、新しいターゲットがエレメントを連結させたもの (水平マッピング) なのか、ビット幅 (垂直マッピング) なのかを決定するために使用します。配列は、`set_directive_array_map` コマンドが出力された順番で連結され、水平マッピングの場合はターゲット エレメント 0 から開始、垂直マッピングの場合はビット 0 から開始します。

**<location>** - 関数 [/ラベル] のフォーマットで配列変数を含むロケーションを指定します。

**<variable>** - 新しいターゲット 配列 インスタンスにマップする配列変数名を指定します。

## オプション

**-instance <string>** - 現在の配列変数がマップされる新しい配列 インスタンス名を指定します。

**-mode (horizontal|vertical)** - 水平マッピングではターゲットに複数のエレメントが含まれるように複数の配列が連結されます。垂直マッピングでは 1 つの配列に連結され、ターゲットのワード数が大きくなります。デフォルトは **horizontal** です。

**-offset <integer>** - これは水平マッピングにのみ関連したオプションで、現在のマップ操作のターゲット インスタンスでの絶対オフセットを示す整数値を指定します。配列変数のエレメント 0 は、新しいターゲットのエレメント **<int>** にマップされます (たとえば、ほかのエレメントは **<int+1>**、**<int+2>** にマップされます)。値が指定されていない場合は、重複を避けるため、Vivado HLS で必要なオフセットが自動的に計算されます (たとえばターゲットの次の未使用エレメントで配列の連結が開始されます)。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP array_map \
 variable=<variable> \
 instance=<instance> \
 <horizontal, vertical> \
 offset=<int>
```

## 例

次のコマンド (pragma も含む) は、関数 `foo` の配列 `A[10]` および `B[15]` を、新しい 1 つの配列 `AB[25]` にマップします。エレメント `AB[0]` は `A[0]`、エレメント `AB[10]` は `B[0]` と同じで (オフセット オプションが使用されていないため)、配列 `AB[25]` のビット幅は `A[10]` または `B[15]` の最大ビット幅になります。

```
set_directive_array_map -instance AB -mode horizontal foo A
set_directive_array_map -instance AB -mode horizontal foo B
#pragma AP array_map variable=A instance=AB horizontal
#pragma AP array_map variable=B instance=AB horizontal
```

この例では、配列 `C` および `D` が新しい配列 `CD` に連結され、ビット数は `C` および `D` のビット数を足したものになります。`CD` のエレメント数は `C` または `D` の最大数になります。

```
set_directive_array_map -instance CD -mode vertical foo C
set_directive_array_map -instance CD -mode vertical foo D
#pragma AP array_map variable=C instance=CD vertical
#pragma AP array_map variable=D instance=CD vertical
```

---

## set\_directive\_array\_partition

### 構文

```
set_directive_array_partition [OPTIONS] <location> <array>
```

### 説明

配列をより小さな配列または個々のエレメントに分割します。結果として RTL には、1 つの大きなメモリではなく、複数の小型メモリまたは複数のレジスタがインプリメントされます。ストレージの読み出しおよび書き込みポートの数が増加し、デザインのスループットが改善される可能性がありますが、より多くのメモリ インスタンスやレジスタが必要となります。

<location> - 関数[/ラベル] のフォーマットで配列変数を含むロケーションを指定します。

<solution> - 分割する配列の名前を指定します。

### オプション

**-dim <integer>** - 複数次元の配列にのみ使用するオプションで、配列のどの次元を分割するかを指定します。値が 0 の場合は、指定されているオプションとともにすべての次元が分割されます。その他の値が指定されている場合は、その次元のみが分割されます。たとえば、値が 1 の場合は、最初の次元のみが分割されます。

**-factor <integer>** - 作成する小型配列の数を整数で指定します。このオプションは、**block** または **cyclic** タイプの分割にのみ使用します。

**-type (block|cyclic|complete)** - 元の配列の連続したブロックから小型配列を分割します。1 つの配列を N 個のブロックに分割するオプションで、N は -factor で定義されている整数です。デフォルトは **complete** です。

循環分割は、元の配列のエレメントをインターリーブして小型配列を作成します。たとえば、**-factor** が 3 の場合、エレメント 0 は新しく作成される 1 番目の配列に割り当てられ、エレメント 1 は 2 番目、エレメント 3 は 3 番目、そしてエレメント 4 はまた 1 番目の配列に割り当てられます。

完全分割は、配列を個々のエレメントに分割します。1 次元配列の場合は、配列が個々のエレメントに分割されます。複数次元の配列の場合は、各次元の分割を指定するか、または **-dim 0** を使用してすべての次元を分割します。

### pragma

C ソース コードの必要なロケーションの境界内に **pragma** を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP array_partition \
 variable=<variable> \
 <block, cyclic, complete> \
```

```
factor=<int> \
dim=<int>
```

## 例

次の例は、関数 `foo` の配列 `AB[13]` を 4 つの配列に分割します (関連 `pragma` も含む)。4 は 13 の倍数整数ではないので、3 つの配列に 13 個のエレメント、1 つの配列に 4 つのエレメント (エレメント `AB[9:12]` を含む) に分割されます。

```
set_directive_array_partition -type block -factor 4 foo AB
#pragma AP array_partition variable=AB block factor=4
```

次の例は、関数 `foo` の配列 `AB[6][4]` を 2 つの配列に分割し、各次元が `[6][2]` になります。

```
set_directive_array_partition -type block -factor 2 -dim 2 foo AB
#pragma AP array_partition variable=AB block factor=2 dim=2
```

次のコマンドでは、関数 `foo` の `AB[4][10][6]` のすべての次元が、個々のエレメントに分割されます。

```
set_directive_array_partition -type complete -dim 0 foo AB
#pragma AP array_partition variable=AB complete dim=0
```

## set\_directive\_array\_reshape

### 構文

```
set_directive_array_reshape [OPTIONS] <location> <array>
```

### 説明

このコマンドは、配列 分割と垂直配列 マッピングをまとめ、エレメント数は少なくワード数は大きい配列を 1 つ新規作成します。

まず、配列が複数の配列に分割され (`set_directive_array_partition` と同じ)、次に分割された配列が垂直方向に自動的にまとめられ (`set_directive_array_map -type vertical` と同じ)、ワード数の大きい 1 つの配列が新規作成されます。

`<location>` - 関数[/ラベル] のフォーマットで配列変数を含むロケーションを指定します。

`<solution>` - まとめ直す配列変数の名前を指定します。

### オプション

**-dim <integer>** - 複数次元の配列にのみ使用するオプションで、配列のどの次元をまとめるかを指定します。値が 0 の場合は、指定されているオプションとともにすべての次元が分割されます。その他の値が指定されている場合は、その次元のみが分割されます。たとえば、値が 1 の場合は、最初の次元のみが分割されます。

**-factor <integer>** - 作成する一時的な小型配列の数を整数で指定します。これは `-type` に `block` または `cyclic` が指定されているのみに使用するオプションです。

**-type (block|cyclic|complete)** - 元の配列の連続したブロックから小型配列を作成します。N 個のブロックに配列を分割し (N は-factor オプションで定義されている数)、N 個のブロックを 1 つの配列にまとめます。ワード幅は N を掛けた値になります。デフォルトは **complete** です。

循環タイプは、元の配列のエレメントをインターリーブして小型配列を作成します。たとえば、**-factor** が 3 の場合、エレメント 0 は新しく作成される 1 番目の配列に割り当てられ、エレメント 1 は 2 番目、エレメント 3 は 3 番目、そしてエレメント 4 はまた 1 番目の配列に割り当てられます。最終配列は、新しい配列を 1 つの配列に垂直連結になります (ワード数が大きいものを作成するにはワード連結)。

完全タイプは、配列を一時的に個々のエレメントに分割してから、ワード数の大きな 1 つの配列にまとめます。1 次元配列の場合、これはワード数が非常に大きいレジスタを 1 つ作成するのと同じです (元の配列が M ビットの N 個のエレメントであった場合、N\*M ビットのレジスタになります)。

## pragma

C ソース コードの必要なロケーションの境界内に **pragma** を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP array_reshape \
 variable=<variable> \
 <block, cyclic, complete> \
 factor=<int> \
 dim=<int>
```

## 例

次のコマンド (同等の **pragma** も含む) は、関数 **foo** の 8 ビット 配列 **AB[17]** を、エレメントを 5 つ含む 32 ビット 配列 1 つにまとめます。4 は 13 の整数倍数ではないので、**AB[17]** は 5 番目のエレメントの下位 8 ビットにあります (5 番目のエレメントの余りは未使用)。

```
set_directive_array_reshape -type block -factor 4 foo AB
#pragma AP array_reshape variable=AB block factor=4
```

次の例は、関数 **foo** の配列 **AB[6][4]** を、次元 [6][2] の新しい配列 1 つに分割します。次元 2 は 2 倍幅です。

```
set_directive_array_reshape -type block -factor 2 -dim 2 foo AB
#pragma AP array_reshape variable=AB block factor=2 dim=2
```

次のコマンドは、関数 **foo** の 8 ビット 配列 **AB[4][2][2]** をシングル エレメントの配列 (1 つのレジスタ) にし、ビット幅は  $4*2*2*8 (=128)$  になります。

```
set_directive_array_reshape -type complete -dim 0 foo AB
#pragma AP array_reshape variable=AB complete dim=0
```

## set\_directive\_array\_stream

### 構文

```
set_directive_array_stream [OPTIONS] <location> <variable>
```

## 説明

デフォルトでは、配列変数は RAM メモリとしてインプリメントされます。

- 最上位関数の配列パラメーターは RAM インターフェイスポートとしてインプリメントされます。
- 一般配列は、読み出しおよび書き込みアクセス用に RAM としてインプリメントされます。
- データフロー最適化に関連した下位関数では、配列引数は RAM のピンポン バッファーチャネルを使用してインプリメントされます。
- ループベースのデータフロー最適化に関連した配列は RAM のピンポン バッファーチャネルを使用してインプリメントされます。

しかし、配列に保存されているデータが順次に入力/出力される場合は、RAM ではなく FIFO が使用されるストリーミングデータを使用するほうが効率的です。

**注記** : 最上位関数の引数のインターフェイス タイプが `ap_fifo` に指定されているときは、配列は自動的にストリーミングであると判別されます。

`<location>` - 関数[/ラベル] のフォーマットで配列変数を含むロケーションを指定します。

`<variable>` - FIFO としてインプリメントされる配列変数名を指定します。

## オプション

**-depth** `<integer>` - データフロー チャネルの配列ストリーミングにのみ関連し、`config_dataflow` コマンドで(グローバルに) 指定されたデフォルトの FIFO の深さを上書きするのに使用します。

**-off** - データフロー チャネルの配列ストリーミングにのみ関連しています。使用する場合は、`config_dataflow -default_channel fifo` コマンドによりデザイン全体の配列に `set_directive_array_stream` がグローバルに適用されます。このオプションは、特定配列でストリーミングをオフにします(また、RAM ピンポン バッファーベース チャネルが使用されるようになります)。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP array_stream
 variable=<variable> \
 off \
 depth=<int>
```

## 例

次のコマンド(同等の `pragma` も表示)は、関数 `foo` の配列 `A[10]` をストリーミングにし、FIFO としてインプリメントします。

```
set_directive_array_stream foo A
#pragma AP array_reshape variable=A
```

次の例では、関数 `foo` の `loop_1` というループにある配列 `B` が深さ 12 の FIFO でストリーミングされるように設定されます。この場合、`pragma` は `loop_1` 内に挿入する必要があります。

```
set_directive_array_stream -depth 12 foo/loop_1 B
```

```
#pragma AP array_reshape variable=B depth=12
```

次の例では、配列 C のストリーミングがディスエーブルになります(この例では config\_dataflow でイネーブルになっていると想定)。

```
set_directive_array_stream -off foo C
#pragma AP array_reshape variable=C off
```

## set\_directive\_clock

### 構文

```
set_directive_clock <location> <domain>
```

### 説明

指定クロックを指定関数に適用します。

C および C++ デザインでは、シングルクロックのみがサポートされており、create\_clock で指定されたクロック周期は自動的にデザインのすべての関数に適用されます。

SystemC デザインではマルチクロックがサポートされています。create\_clock コマンドを使用して複数のクロックを指定し、set\_directive\_clock コマンドを使用して個々の SC\_MODULE に適用することができます。各 SC\_MODULE はシングルクロックで合成されます。

<location> - 指定クロックが適用される関数の名前を指定します。

<domain> - create\_clock コマンドの -name オプションで指定されているクロック名を指定します。

### pragma

C ソースコードの必要なロケーションの境界内に pragma を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP clock domain=<string>
```

### 例

SystemC デザインの場合、最上位の foo\_top には fast\_clock および slow\_clock というクロックポートがありますが、最上位ではその関数内で fast\_clock のみを使用し、下位ブロックの foo は slow\_clock のみを使用します。次のコマンドは両方のクロックを作成し、fast\_clock を foo\_top に、slow\_clock を下位ブロック foo に適用します。同等の pragma も表示されており、該当する関数内に配置する必要があります。

注記 : create\_clock と同等の pragma はありません。

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
```

```
set_directive_clock foo_top fast_clock
set_directive_clock foo slow_clock
```

```
#pragma AP clock domain=fast_clock
#pragma AP clock domain=slow_clock
```

---

## set\_directive\_dataflow

### 構文

```
set_directive_dataflow [OPTIONS] <location>
```

### 説明

set\_directive\_dataflow コマンドは、関数またはループでデータフロー最適化が実行されるように指定するコマンドで、RTL インプリメンテーションの同時実行を改善します。

C 記述では、すべての演算は順次に実行されます。Vivado HLS は、set\_directive\_allocation などリソースを制限する指示子がない場合は、自動的にレイテンシを最小限に押さえ、同時実行性を改善しようとしますが、データ依存性のために制限されることがあります。たとえば、配列にアクセスする関数/ループは完了する前に配列への読み出し/書き込みアクセスをすべて終了する必要があり、データを使用する次の関数/ループが演算を開始できなくなります。

しかし、前の関数/ループが完了するために、次の関数/ループで演算を実行開始することは可能です。

データフロー最適化が指定されていると、Vivado HLS は順次関数/ループ間のデータフローを解析し、(ピンポン RAM または FIFO に基づいて)チャネルを作成しようとします。これで、前の関数/ループが完了する前に、次の関数/ループが実行開始することができ、関数/ループがパラレルで実行され、レイテンシが低減し、RTL デザインのスループットが改善されます。

開始インターバル(関数/ループの開始から次のものの開始までの間のサイクル数)が指定されていない場合は、Vivado HLS は開始インターバルを最小限に抑えて、データが使用可能になったらすぐに演算を開始できるようにします。

<location> - 関数/[ラベル] のフォーマットでデータフロー最適化を実行するロケーションを指定します。

### オプション

**-interval <integer>** - 開始インターバルを整数で指定します (II)。最初の関数/ループの実行から次の関数/ループの実行開始までのサイクル数です。

### pragma

C ソース コードの必要なロケーションの境界内に pragma を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP dataflow interval=<int>
```

### 例

この例は、関数 foo 内のデータフロー最適化を指定します。同等の pragma も表示します。

```
set_directive_dataflow foo
#pragma AP dataflow interval=3
```

この例では、関数 My\_Func でデータフローが指定されており、開始インターバルは 3 になっています。

```
set_directive_dataflow -interval 3 My_func
#pragma AP dataflow interval=3
```

## set\_directive\_data\_pack

### 構文

```
set_directive_data_pack [OPTIONS] <location> <variable>
```

### 説明

この指示子は、構造体のデータ フィールドを 1 つのスカラー (ワード数が大きい) にパックします。構造体内で宣言されている配列はすべて完全に分割され、幅の広いスカラーにまとめられ、ほかのスカラ フィールドを使用してパックされます。

ワード数のビット アライメントは、構造体フィールドの宣言から自動推論されます。すべてのフィールドがマップされるまで、最初のフィールドにはワードの最下位部が入ります。

<location> - 関数[/ラベル] のフォーマットで変数がパックされるロケーションを指定します。

<variable> - パックされる変数の名前を指定します。

### オプション

**-instance <string>** - パッキング後の変数の名前を指定します。何も指定されていない場合は、variable に入力されている名前が使用されます。

### pragma

C ソース コードの必要なロケーションの境界内に pragma を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP data_pack variable=<variable> instance=<string>
```

### 例

次のコマンドは (同等の pragma も表示)、関数 foo にある 8 ビットのフィールドが 3 つある構造体配列 AB[17] を 24 ビットの 17 エレメントの配列 1 つにパックします。

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

次のコマンドは (同等の pragma も表示)、関数 foo にある 8 ビットのフィールドが 3 つある構造体ポインター AB を 1 つの 24 ビット ポインターにパックします。

```
set_directive_data_pack foo AB
#pragma AP data_pack variable=AB
```

# set\_directive\_dependence

## 構文

```
set_directive_dependence [OPTIONS] <location>
```

## 説明

独立したループ間の依存(ループ独立型の依存)、または同じループの反復間の依存(ループ運搬型の依存)が Vivado HLS で自動的に検出されます。こうした依存は、演算がスケジュールできるタイミング、特に関数およびループのパイプライン化処理に影響します。

ループ独立型の依存: 同じループの反復で同じエレメントがアクセスされます。

```
for (i=0;i<N;i++) {
 A[i]=x;
 y=A[i];
}
```

ループ運搬型の依存: 異なるループ反復で同じエレメントがアクセスされます。

```
for (i=0;i<N;i++) {
 A[i]=A[i-1]*2;
}
```

変数依存の配列 インデックス化や、外部要件を満たす必要がある場合(2つの入力が同じ指数にならない場合など)などの状況では、依存分析は厳しすぎることがあります。 `set_directive_dependence` コマンドを使用すると、依存を明示的に指定でき、不正な依存を解決することができます。

`<location>` - 関数[/ラベル] のフォーマットで依存のロケーションを指定します。

## オプション

**-class (array|pointer)** - 依存を明示的にする必要がある変数のクラスを指定します。これは、**-variable** オプションとは相互排他的です。

**-dependent (true|false)** - 依存を使用する必要がある (true) か、削除する (false) を指定します。デフォルトは `false` です。

**-direction (RAW|WAR|WAW)** - ループ運搬型の依存の場合にのみ使用します。依存のタイプを次のように指定します。

- **RAW** (書き込み後に読み出し - 真の依存) - 書き込み命令で使用される値が、読み出し命令で使用されます。
- **WAR** (読み出し後に書き込み - アンチ依存) - 読み出し命令の値は、書き込み命令で上書きされます。
- **WAW** (書き込み後に書き込み - 出力依存) - ある順序で同じロケーションに 2 つの書き込み命令により書き込みが実行されます。

**-distance <integer>** - これは、**-dependent** が `true` に設定されているループ運搬型依存でのみ使用します。配列アクセスの反復の間隔を正の整数で指定します。

**-type (intra|inter)** - 同じループ内の反復 (intra) または異なるループ間の反復 (inter) かを指定します。デフォルトは `inter` です。

**-variable <variable>** - 依存指示子を考慮するための具体的な変数を指定します。これは、**-class** オプションとは相互排他的です。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP dependence \
 variable=<variable> \
 <array, pointer> \
 <inter, intra> \
 <RAW, WAR, WAW> \
 distance=<int> \
 <false, true>
```

## 例

次の例では、関数 `foo` の `loop_1` の同じ反復での `Var1` の依存性を削除します。同等の `pragma` も表示します。

```
set_directive_dependence -variable Var1 -type intra \
 -dependent false foo/loop_1
#pragma AP dependence variable=Var1 intra false
```

次の例では、関数 `foo` の `loop_2` にあるすべての配列の依存性を、同じループ 関数ですべての読み出しが書き込みの後に実行されるように設定します。

```
set_directive_dependence -class array -type inter \
 -dependent true -direction RAW foo/loop_2
#pragma AP dependence array inter RAW true
```

## set\_directive\_expression\_balance

### 構文

```
set_directive_expression_balance [OPTIONS] <location>
```

### 説明

C ベースの仕様は演算シーケンスで書かれていることがあります。その場合、RTL での演算チェーンが長くなり、僅かなクロック周期でデザイン レイテンシが増加する可能性が出てきます。

デフォルトでは、Vivado HLS では、ハードウェア上でのレイテンシを低減しつつ、演算チェーンを短くすることができるバランスのとれたツリーを作成するため、演算の関連性や接続性を考慮して、演算が並び替えられます。

`set_directive_expression_balance` コマンドを使用すると、このバランス機能を指定範囲内でオン/オフにすることができます。

`<location>` - 関数[/ラベル] のフォーマットで、バランス機能をオン/オフにするロケーションを指定します。

## オプション

-off - バランス機能を指定ロケーションでオフにします。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP expression_balance <off>
```

### 例

次の例では、関数 `My_Func` 内でバランス機能をオフにします。同等の `pragma` も表示します。

```
set_directive_expression_balance -off My_Func
#pragma AP expression_balance off
```

次の例では、関数 `My_Func2` でバランス機能をオンにします。

```
set_directive_expression_balance My_Func2
#pragma AP expression_balance
```

## set\_directive\_function\_instantiate

### 構文

```
set_directive_function_instantiate <location> <variable>
```

### 説明

デフォルトで次のようにになります。

- 関数は RTL で個別の階層ブロックのまま残ります。
- 同じ階層レベルにある、任意関数のすべてのインスタンスは、同じ RTL インプリメンテーション (ブロック) を使用します。

`set_directive_function_instantiate` コマンドは、関数のインスタンスごとに 1 つの RTL インプリメンテーションを作成するのに使用します。これで各インスタンスを最適化できるようになります。

デフォルトでは、次のコードで、3 つのインスタンスすべてに対し、関数 `foo_sub` の RTL インプリメンテーションが 1 つ作成されます。

```
char foo_sub(char inval, char incr)
{
 return inval + incr;
}
void foo(char inval1, char inval2, char inval3,
 char *outval1, char *outval2, char * outval3)
{
```

```

 *outval1 = foo_sub(inval1, 1);
 *outval2 = foo_sub(inval2, 2);
 *outval3 = foo_sub(inval3, 3);
}

```

以下の例に示す方法でこの指示子を使用すると、3つのバージョンの関数 `foo_sub` が作成され、変数 `incr` に対してそれぞれが個別に最適化されます。

`<location>` - 関数[/ラベル] のフォーマットで関数のインスタンスを個別に作成するロケーションを指定します。

`variable <string>` - 定数として指定する引数 `<string>` を指定します。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP function_instantiate variable=<variable>
```

## 例

この例では、次の Tcl(または関数 `foo_sub` に配置された `pragma`) により、`foo_sub` の入力されている `incr` に従って各インスタンスが個別に最適化されます。

```
set_directive_function_instantiate incr foo_sub
#pragma AP function_instantiate variable=incr
```

## set\_directive\_inline

### 構文

```
set_directive_inline [OPTIONS] <location>
```

### 説明

階層の別エンティティとしての関数を削除します。関数をインライン化すると、1階層としては表示されなくなります。

場合によっては、関数をインライン化すると、関数内の演算が共有され、周辺の演算と共に効率よく最適化されることがあります。しかし、インライン化された関数は共有できないため、エリア増大につながるケースもあります。

デフォルトでは、関数階層のすぐ下の階層でのみインライン化が実行されます。

`<location>` - 関数[/ラベル] のフォーマットでインライン化を実行するロケーションを指定します。

## オプション

**-off** - 関数のインライン化をオフにし、また特定関数がインラインされないように使用します。たとえば、**-recursive** オプションが呼び出し関数に使用されている場合、ほかの関数はインライン化されても、特定の呼び出された関数がインライン化されないようにすることができます。

**-recursive** - デフォルトでは 1 階層でのみ関数がインライン化されます。指定関数内の下位関数はインライン化されません。 **-recursive** オプションは階層全体の関数をすべて再帰的にインライン化します。

**-region** - 指定範囲の関数をすべてインライン化します。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP inline <region | recursive | off>
```

## 例

次の例では、`foo_top` にあるすべての関数をインライン化します (しかし下位にある関数はインライン化されません)。

```
set_directive_inline -region foo_top
#pragma AP inline region
```

次の例では、`foo_sub1` という関数のみがインライン化されます。

```
set_directive_inline foo_sub1
#pragma AP inline
```

次のコマンドでは、`foo_sub2` を除く、`foo_top` にあるすべての関数が階層全体を通して再帰的にインライン化されます。1 つ目の `pragma` は `foo_top` に配置されています。2 つ目の `pragma` は `foo_sub2` に配置されています。

```
set_directive_inline -region -recursive foo_top
set_directive_inline -off foo_sub2
#pragma AP inline region recursive
#pragma AP inline off
```

## set\_directive\_interface

### 構文

```
set_directive_interface [OPTIONS] <location> <port>
```

### 説明

`set_directive_interface` コマンドは、インターフェイス合成で関数記述からどのように RTL ポートが作成されるかを指定します。

RTL インプリメンテーションのポートは次のものから派生します。

- 指定されている、任意の関数レベルのプロトコル
- 関数引数
- グローバル変数 - 最上位関数がアクセスし、外部定義されている。

関数レベルのハンドシェイクは、関数が演算を開始するタイミングを制御し、演算を終了し、アイドルになり、(パイプライン化された関数の場合は)新しい入力を受信する準備が完了するタイミングを示します。関数レベルのプロトコルのインプリメンテーションは **ap\_ctrl\_none** または **ap\_ctrl\_hs** という 2 つのモードで制御され、最上位関数名のみが必要です (pragma には `return` という関数を指定する必要がある)。

各関数引数は、独自の I/O プロトコルを持つように指定することができます (有効なハンドシェイクや承認ハンドシェイクなど)。

グローバル変数にアクセスがあつても、すべての読み出しおよび書き込みアクセスがローカルになっている場合、リソースはデザインで作成され、RTL に I/O ポートの必要はありません。しかし、グローバル変数が外部ソースまたはデスティネーションである場合は、独自のインターフェイスが標準関数引数と同じような方法で指定されている必要があります (例を参照)。

`set_directive_interface` が下位関数に使用されている場合は、**-register** オプションのみを使用することができます。**-mode** オプションは下位関数ではサポートされていません。

**<location>** - 関数 [/ラベル] のフォーマットで、関数インターフェイスまたはレジスタを介した出力のロケーションを指定します。

**<port>** - インターフェイスを合成するのに必要なパラメーター (関数引数またはグローバル変数) **ap\_ctrl\_none** または **ap\_ctrl\_hs** モードが使用されている場合は、これは不要です。

## オプション

**-mode** (**ap\_ctrl\_none** | **ap\_ctrl\_hs** | **ap\_none** | **ap\_stable** | **ap\_vld** | **ap\_ovld** | **ap\_ack** | **ap\_hs** | **ap\_fifo** | **ap\_memory** | **ap\_bus**) - 該当プロトコルを選択します。

**-mode** 値でインプリメントされる関数プロトコルは次のとおりです。

- **ap\_ctrl\_none** - 関数レベルのハンドシェイクなし
- **ap\_ctrl\_hs** - これがデフォルトで、関数レベルのハンドシェイク プロトコルがインプリメントされます。入力ポート **ap\_start** は、関数が演算を開始するため High になる必要があります (すべての関数レベル信号はアクティブ High)。出力ポート **ap\_done** は、関数が終了したことを示し (また関数の戻り値がある場合は、戻り値が有効であることを示す)、出力ポート **ap\_idle** は、関数がアイドル状態であることを示します。パイプライン化された関数では、追加出力ポート **ap\_ready** がインプリメントされ、その関数が新しい入力データを受信する準備完了状態にあることを示します。

関数引数およびグローバル変数の場合は、各引数タイプに対し、次のデフォルト プロトコルが使用されます。

- 読み出し専用 (入力) - **ap\_none**
- 書き込み専用 (出力) - **ap\_vld**
- 読み出し-書き込み (入出力) - **ap\_ovld**
- 配列 - **ap\_memory**

関数引数およびグローバル変数をインプリメントする RTL ポートは、次の **-mode** 値で指定されます。

- **ap\_none** - プロトコルなしこれは、単純書き込みに対応します。
- **ap\_stable** - 入力ポートにのみ使用します。このポートの値がリセット後安定していく、次のリセットまで変化しません。このプロトコルは **ap\_none** モードのようにインプリメントされますが、信号ファンアウトで内部最適化が実行されます。

**注記** : これは定数値として考慮されず、単に変動しない値です。

- **ap\_vld** - このデータ ポートと共に動作する追加有効ポートが作成されます (**<port\_name>\_vld**)。入力ポート場合、関連の入力有効ポートがアサートされるまで、読み出しにより関数が停止されます。出力ポートの場合、データを書き込むとき、出力有効信号がアサートされます。

- **ap\_ack** - このデータポートと共に動作する追加承認ポートが作成されます (<port\_name>\_ack)。入力ポートの場合、値を読み出すとき、読み出しにより出力承認がアサートされます。関連の入力承認ポートがアサートされるまで、出力書き込みにより関数が停止されます。
- **ap\_hs** - このデータポートと共に動作する追加有効ポートが作成されます (<port\_name>\_vld) and acknowledge (<port\_name>\_ack)。入力ポートの場合、読み出しにより、入力有効がアサートされるまで関数が停止され、データが読み出されたときに出力承認信号がアサートされます。出力書き込みにより、データが書き込まれたとき出力有効がアサートされ、関連入力承認ポートがアサートされるまで関数が停止されます。
- **ap\_ovld** - 入力信号の場合、モード **ap\_none** のように動作し、プロトコルは追加されません。出力信号の場合は、**ap\_vld** モードのように動作します。入出力信号の場合、入力は **ap\_none** モードとしてインプリメントされ、出力は **ap\_vld** モードとしてインプリメントされます。
- **ap\_memory** - 外部 RAM へのアクセスとして配列引数がインプリメントされます。外部 RAM からの読み出しおよびその RAM への書き込みを行うため、データ、アドレス、および RAM 制御ポート (CE や WE など) が作成されます。特定の信号およびデータポート数はアクセスしている RAM によって決まります。配列引数は、`set_directive_resource` コマンドを使用してテクノロジライブラリにある特定 RAM をターゲットにする必要があります (または Vivado HLS で自動的に使用する RAM が選択されます)。
- **ap\_fifo** - FIFO アクセスとして、配列、ポインター、参照渡し変数のポートをインプリメントします。データ入力ポートは外部 FIFO から値を読み出す準備完了が整うと、関連の出力読み出しポート (<port\_name>\_read) をアサートします。また、値読み出し可能であることを示すため入力可能なポート (<port\_name>\_empty\_n) がアサートされるまで、関数を停止します。ポートに値を書き込んだことを示すため、出力データポートは出力書き込みポート (<port\_name>\_write) をアサートします。また、新しい出力用に外部 FIFO に空きスペースがあることを示すため、関連入力可能ポート (<port\_name>\_full\_n) がアサートされるまで、関数を停止します。このインターフェイスモードには **-depth** オプションを使用する必要があります。
- **ap\_bus** - バスインターフェイスとして、ポインター、参照渡し変数のポートをインプリメントします。標準 FIFO バスインターフェイスとのバーストアクセスをサポートするため、多数の制御信号を使用して、入力および出力両方のポートが合成されます。このインターフェイスの詳細は、「高位合成ユーザー ガイド」の章を参照してください。このインターフェイスモードには **-depth** オプションを使用する必要があります。

**-depth** - **ap\_fifo** または **ap\_bus** モードを使用しているポインターインターフェイスに必要です。テストベンチで処理されるサンプルの最大数を指定するのに使用する必要があります。これは、RTL 協調シミュレーション用に作成された検証アダプターに必要な FIFO の最大サイズを Vivado HLS に渡すために必要です。

**-register** - 最上位関数の場合、**ap\_none**、**ap\_ack**、**ap\_vld**、**ap\_ovld**、**ap\_hs** のスカラーインターフェイスに使用します。信号 (および任意の関連プロトコル信号) はレジスタを介し、少なくとも関数実行の最終サイクルまで保持されるようになります。このオプションを使用するには、**ap\_ctrl\_hs** 関数プロトコルが有効になっている必要があります。このオプションが **ap\_ctrl\_hs** と共に使用されると、関数の戻り値がレジスタを介すようになります。

関数実行の終わりまで、出力および制御信号をレジスタに保存するため、このオプションを下位関数に使用することができます。

## pragma

C ソースコードの必要なロケーションの境界内に **pragma** を挿入する必要があります。

フォーマットおよびオプションは次のようになります。

```
#pragma AP interface <mode> register port=<string>
```

## 例

次の例は、関数 **foo** の関数レベルハンドシェイクをオフにします。

```
set_directive_interface -mode ap_ctrl_none foo
#pragma AP interface ap_ctrl_none port=return
```

次の例では、関数 `foo` の `InData` に `ap_vld` インターフェイスが指定され、入力がレジスタを介すように設定されます。

```
set_directive_interface -mode ap_vld -register foo InData
#pragma AP interface ap_vld register port=InData
```

次の例では、関数 `foo` で使用されるグローバル変数 `lookup_table` が RTL デザインでポートとして処理され、インターフェイスには `ap_memory` が指定されます。

```
set_directive_interface -mode ap_memory foo look_table
```

## set\_directive\_latency

### 構文

```
set_directive_latency [OPTIONS] <location>
```

### 説明

関数、ループ、またはリージョンにレイテンシの最大値または最小値を設定します。Vivado HLS は常に最小値を目標にします。レイテンシの最大値および最小値が指定されるときの Vivado HLS の動作は次のようにになります。

- レイテンシが最小値未満 - レイテンシが最小値に満たない場合、レイテンシが指定値まで拡張され、リソースがさらに共有される可能性があります。
- レイテンシが最小値より大きい - 制約は満たされ、これ以上の最適化は行われません。
- レイテンシが最大値未満 - 制約は満たされ、これ以上の最適化は行われません。
- レイテンシが最大値より大きい - 最大値以下でスケジュールできない場合、指定されている制約を満たすことができるよう自動的にエフォート レベルが上げられます。それでも最大レイテンシを満たすことができない場合は、警告が output され、達成可能な最小値のレイテンシでデザインが作成されます。

`<location>` - 関数[/ラベル] フォーマットで制約を設定するロケーション(関数 ループ、またはリージョン)を指定します。

### オプション

`-max <integer>` - 最大レイテンシを整数で指定します。

`-min <integer>` - 最小レイテンシを整数で指定します。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP latency \
 min=<int> \
 max=<int>
```

## 例

次の例では、関数 `foo` の最小レイテンシが 4、最大が 8 に指定されます。

```
set_directive_latency -min=8 -max=8 foo
#pragma AP latency min=4 max=4
```

関数 `foo` で、ループ `loop_row` の最大レイテンシを 12 に指定します。`pragma` はループの本文に挿入する必要があります。

```
set_directive_latency -max=12 foo/loop_row
#pragma AP latency max=12
```

## set\_directive\_loop\_flatten

### 構文

```
set_directive_loop_flatten [OPTIONS] <location>
```

### 説明

このコマンドは、入れ子になっているループを 1 つのループ階層にフラットにするために使用します。RTL インプリメンテーションでは、ループ階層にあるループ間の移動に 1 クロック サイクルかかります。入れ子ループをフラットにすることで 1 つのループとして最適化することができ、多くのクロック サイクルを必要としなくなり、また、ループ本文のロジックの最適化をさらにすすめることができます。

この指示子は、ループ階層の一番内側にあるループに適用する必要があります。完全または半完全ループのみをこの方法でフラットすることができます。

- 完全ループ ネスト - 最も内側にあるループにのみループ本文が含まれていて、ループ本文の間にロジックは指定されていません。また、ループの境界はすべて定数です。
- 半完全ループ ネスト - 最も内側にあるループにのみループ本文が含まれていて、ループ本文の間にロジックは指定されていません。しかし一番外側のループ境界は変数である可能性があります。

内側のループの範囲が変数であったり、ループ本体が最内側のループにのみ含まれているとは限らない不完全ループ ネストは、コードの構造を変更するか、ループ本体の中のループを展開して、完全ループ ネストを作成してみてください。

`<location>` - 関数[/ラベル] フォーマットでロケーション(最も内側のループ)を指定します。

### オプション

**-off** - フラットにならないようにします。一部のループがフラットにならないようにするために使用します。これ以外の指定ロケーションのループはフラットされます。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようになります。

```
#pragma AP loop_flatten off
```

## 例

関数 `foo` の `loop_1`、ループ階層上でこれより上にあるすべてのループ（完全または半完全）を 1 つのループにフラットにします。`pragma` は `loop_1` の本文に挿入する必要があります。

```
set_directive_loop_flatten foo/loop_1
#pragma AP loop_flatten
```

次の例では、関数 `foo` の `loop_2` でループがフラットされないようにします。`pragma` は `loop_2` の本文に挿入する必要があります。

```
set_directive_loop_flatten -off foo/loop_2
#pragma AP loop_flatten off
```

## set\_directive\_loop\_merge

### 構文

```
set_directive_loop_merge <location>
```

### 説明

すべてのループを 1 つのループにマージします。ループをマージすると、ループ本文のインプリメンテーション間を移動するのに RTL で必要なクロック サイクル数を低減することができ、また、可能であればループをパラレルにインプリメントすることができます。

ループをマージする場合のルールは次のようになっています。

- ループの境界が変数の場合、同じ値を設定しておく必要があります（同じ数の反復）。
- ループの境界が定数の場合、最大定数值がマージされたループの境界として使用されます。
- 境界が変数のループと、定数のものを一緒にマージさせることはできません。
- マージするループ間のコードは、複数回実行しても常に同じ結果を出力する必要があります（`a=b` は有効、`a=a+1` は無効）。
- FIFO の読み出しが含まれるループはマージできません。マージを行うと、読み出しの順序が変更になります。 FIFO からの読み出しありは FIFO インターフェイスは常にシーケンスである必要があります。

`<location>` - 関数[/ラベル] のフォーマットでループのあるロケーションを指定します。

### オプション

**-force** - Vivado HLS で警告が出力されても、ループがマージされます。この場合は、マージされるループが正しく機能するようにユーザーが確認する必要があります。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP loop_merge force
```

## 例

次の例では、関数 `foo` の連続するすべてのループが 1 つのループにマージされます。

```
set_directive_loop_merge foo
#pragma AP loop_merge
```

次の例では、関数 `foo` の `loop_2` の内側にあるすべてのループ (`loop_2` を除くすべて) が `-force` オプションを使用してマージされます。`pragma` は `loop_2` の本文に挿入する必要があります。

```
set_directive_loop_merge -force foo/loop_2
#pragma AP loop_merge force
```

## set\_directive\_loop\_tripcount

### 構文

```
set_directive_loop_tripcount [OPTIONS] <location>
```

### 説明

ループによって実行される反復回数は、ループのトリップカウントと呼ばれます。Vivado HLS は、各ループの合計レイテンシ、つまりループのすべての反復を実行するためのサイクル数をレポートします。このループ レイテンシはトリップカウント (ループ反復の数) というわけです。

トリップカウントは定数値になることもあります、これは、ループ式 ( $x < y$  など) で使用される変数の値や、ループ内の制御文によって変わります。トリップカウントを決定するのに使用される変数が入力引数であったり、ダイナミック演算により計算される変数であったりすると、Vivado HLS でトリップカウントが決定できず、ループ レイテンシが未知の値になることがあります。

デザイン解析で最適化レベルを判断できるよう、`set_directive_loop_tripcount` コマンドを使用して、ループのトリップカウントの最小値、平均値、最大値を指定し、デザイン レイテンシ全体にループ レイテンシがどの程度占めているのかをレポートで確認することができます。

`<location>` - 関数/[ラベル] のフォーマットでトップカウントのロケーションを指定します。

### オプション

- avg <integer>** - 平均レイテンシを指定します。
- max <integer>** - 最大レイテンシを指定します。
- min <integer>** - 最小レイテンシを指定します。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようになります。

```
#pragma AP loop_tripcount \
 min=<int> \
 max=<int> \
 avg=<int>
```

## 例

次の例では、関数 `foo` の `loop_1` のトリップカウントの最小値が 12、平均が 14、最大値が 16 になるように指定されます。

```
set_directive_loop_tripcount -min 12 -max 14 -avg 16 foo/loop_1
#pragma AP loop_tripcount min=12 max=14 avg=16
```

## set\_directive\_loop\_unroll

### 構文

```
set_directive_unroll [OPTIONS] <location>
```

### 説明

ループ本文のコピーを複数作成することでループを変換します。

ループは、ループ変換で指定されている反復回数分実行されます。反復の回数は、ループ本文内の任意ロジックによっても変わります(ループ終了変数へのブレークや変更など)。ループは RTL にロジックのブロックによりインプリメンテーションされます。このブロックはループ本文を表し、同じ反復回数分実行されます。

`set_directive_loop_unroll` コマンドを使用すると、ループを完全に展開することができ、RTL にループ反復回数と同じ数のループ本文のコピーを作成できます。または、ループを係数  $N$  で部分的に展開し、 $N$  数のループ本文のコピーを作成し、それに応じてループ反復数を調整することができます。

部分展開に使用される係数  $N$  が元のループ反復数の整数倍数でない場合は、ループ本文の展開された箇所の終わりごとに元の終了条件をチェックする必要があります。

ループを完全に展開するには、ループの境界がコンパイル時に認識される必要があります。これは部分展開には必要ありません。

`<location>` - 関数[/ラベル] フォーマットで展開するループのロケーションを指定します。

### オプション

**-factor <integer>** - 部分展開の係数をゼロでない整数値を指定します。ループ本文は、この数値で指定された回数分繰り返され、反復もそれに応じて調整されます。

**-region** - 任意ループ内にあるすべてのループを展開するときに指定します。ただし指定したループそのものは展開されません。

次の例では、ループ `loop_1` 内に `loop_2` および `loop_3` というループがループ階層の同じレベルにあります。`loop_1` など、指定されたループは、`{ }` かっこで囲まれたコードの一部でもあります。展開指示子が `<function>/loop_1` に指定されている場合、`loop_1` が展開されます。

`-region` オプションは、指定されたリージョンに含まれるループにのみ適用されます。`loop_1` は展開されず、その内側にあるすべてのループ (`loop_2` および `loop_3`) が展開されます。

`-skip_exit_check` - 係数が指定されている場合 (部分展開) にのみ、このオプションは有効です。

- 固定境界 - 反復回数が係数の倍数である場合は、終了条件がチェックされません。反復回数が係数の整数倍数である場合は、展開が実行されず、警告が出力されます (終了チェックは処理続行のために必ず実行されます)。
- 可変境界 - 終了条件チェックが削除されます。可変境界が係数の整数倍数であり、終了チェックが不要であることを確認してください。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP unroll \
 skip_exit_check \
 factor=<int> \
 region
```

## 例

次の例では、関数 `foo` のループ `L1` を展開します。`pragma` は `L1` の本文に挿入する必要があります。

```
set_directive_loop_unroll foo/L1
#pragma AP unroll
```

次の例では、関数 `foo` のループ `L2` を係数 4 で展開し、最終チェックが削除されるように指定します。`pragma` は `L2` の本文に挿入する必要があります。

```
set_directive_loop_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

次の例では、関数 `foo` のループ `L3` が展開されますが、ループ `L3` 自体は展開されません。`-region` オプションでは、展開するループを指定しますが、実際に展開されるのはその内側だけです。

```
set_directive_loop_unroll -region foo/L3
#pragma AP unroll region
```

## set\_directive\_occurrence

### 構文

```
set_directive_occurrence [OPTIONS] <location>
```

## 説明

関数またはループをパイプライン化するとき、囲んでいる(外側にある)関数/ループのコードよりも実行速度が遅いコードを含むロケーションを指定するのに使用します。これで、実行速度が遅いコード部分が低速でパイプライン化され、また、最上位パイプライン内でよりよく共有される可能性もあります。

たとえば、反復数が  $N$  回のループがあり、そのループの一部が条件文で保護されていて、 $M$  回しか実行されないとします(この場合  $N$  は  $M$  の整数倍数)。条件文で保護されているコードは、 $N/M$  の実現値となります。

$N$  が初期間隔  $II$  でパイプライン化される場合、条件文で保護されている関数/ループは、これよりも高い値でパイプライン化することができ(低速、つまりこのコードの実行頻度は低い)、またよりよい共有が行われる可能性があります。

こうしたリージョンを識別すると、このリージョンの関数およびループが外側にある関数/ループよりも遅い初期間隔でパイプライン化されるようになります。

*<location>* - 実行速度の遅いロケーションを指定します。

## オプション

**-cycle <int>** -  $N/M$  の実行頻度を指定します。  $N$  は囲んでいる関数/ループの実行回数で、 $M$  は条件文で保護されている箇所が実行される回数です。  $N$  は  $M$  の整数倍数である必要があります。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP occurrence cycle=<int>
```

## 例

次の例では、関数 `foo` のリージョン `Cond_Region` の実行頻度を 4 に指定します。つまり、囲んでいるコードよりも 4 倍低速でこの箇所が実行されます。

```
set_directive_occurrence -cycle 4 foo/Cond_Region
#pragma AP occurrence cycle=4
```

## set\_directive\_pipeline

### 構文

```
set_directive_pipeline [OPTIONS] <location>
```

## 説明

`set_directive_pipeline` コマンドでは、次のパイプライン処理の詳細を指定します。

- 関数のパイプライン処理

- ループのパイプライン処理

パイプライン処理された関数またはループは、N サイクル (N は開始間隔 (II)) ごとに新しい入力を処理できます。デフォルトの開始間隔は 1 で、クロック サイクルごとに新しい入力が処理されます。-II オプションで開始間隔を指定することもできます。

指定した開始間隔でデザインを作成できない場合は、警告メッセージが表示され、最低限可能な開始間隔でデザインが作成されます。この後、警告メッセージを使用してデザインを解析し、必要な開始間隔を満たしてデザインを作成するためにどの手順が必要なのかを分析します。

<location> - 関数/[ラベル] フォーマットでパイプライン処理するロケーションの名前を指定します。

## オプション

-II <integer> - パイプラインの開始間隔を整数で指定します。Vivado HLS では、データの依存性に基づいてこの要求を満たそうとします。実際の結果は、これより大きい間隔になります。

-enable\_flush - パイプラインの入力が停止したときに、パイプライン段をフラッシュできるパイプラインがインプリメントされます。追加で制御ロジックがインプリメントされ、エリアは大きくなります。この機能はオプションです。

-rewind - ループにのみ適用できるオプションで、巻き戻しをイネーブルにします。これにより、連続するループのパイプライン (1 つのループの繰り返しの終わりと次の開始の間に一時停止なし) がイネーブルになります。巻き戻しは、最上位関数内に 1 つのループしかない (または完全なループ ネストがある) 場合にのみ効果的です。ループ前のコード部分は、初期化として認識され、パイプラインで 1 度だけ実行されます。条件文 (if-else) は含むことができません。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP pipeline \
 II=<int> \
 enable_flush \
 rewind
```

## 例

次の例では、`foo` 関数が開始間隔 1 でパイプライン処理されています。

```
set_directive_pipeline foo
#pragma AP pipeline
```

次の例では、`foo` 関数の `loop_1` ループが開始間隔 4 でパイプライン処理されており、パイプラインのフラッシュがイネーブルになっています。

```
set_directive_pipeline -II 4 -enable_flush foo/loop_1
#pragma AP pipeline II=4 enable_flush
```

# set\_directive\_protocol

## 構文

```
set_directive_protocol [OPTIONS] <location>
```

## 説明

このコマンドは、コードの領域であるプロトコル領域を指定するもので、コードで明示的に指定していない限り、Vivado HLS でクロック動作が挿入されます。プロトコル領域は、手動でインターフェイス プロトコルを指定するために使用します。Vivado HLS では、関数引数からの読み出しおよび関数引数への書き込みも含め、動作間にクロックは挿入されません。このため、読み出しおよび書き込みは RTL に従います。

クロック動作は、`ap_wait()` 文 (`ap_utils.h` を含む) を使用すると C で指定できるほか、`wait()` 文 (`systemc.h` を含む) を使用すると C++ および SystemC でも指定できます。`ap_wait` および `wait` 文は、それぞれ C および C++ デザインのシミュレーションには影響せず、Vivado HLS でのみ認識されます。

コードの領域は、領域を波括弧 `{ }` で囲んで名前をつけると、C コードで作成できます。たとえば、`io_section:{..lines of C code...}` は、`io_section` という領域を定義しています。

`<location>` - 関数[/`ラベル`] のフォーマットで、外部のプロトコル要件に対応して、サイクルの正確な方法でインプリメンテーションされるように、ロケーションを指定します。

## オプション

**-mode (floating|fixed)** - デフォルト モード (**floating**) では、プロトコル領域外の文に対応するコードが最終的な RTL のプロトコル文内で重複できるようになります。プロトコル領域のサイクルは正確なままですが、その他の動作は同時に発生できます。

`fixed` モードの場合は、重複がないようになります。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP protocol \
 <floating, fixed>
```

## 例

この例では、`foo` 関数の `io_section` 領域を `fixed` プロトコル領域として定義しています。`pragma` は `io_section` 領域内に置く必要があります。

```
set_directive_protocol -mode fixed foo/io_section
#pragma AP protocol fixed
```

## set\_directive\_resource

### 構文

```
set_directive_resource -core <string> <location> <variable>
```

### 説明

特定のライブラリ リソース(コア)が RTL で変数(配列、演算式、関数の引数)をインプリメントするために使用されるようにします。

Vivado HLS では、現在読み込まれているライブラリで使用可能なコアを使用して、コードに動作がインプリメントされます。変数のインプリメントにライブラリの複数のコアを使用できる場合、`set_directive_resource` コマンドでどのコアを使用するか指定します。`list_core` コマンドを使用すると、ライブラリの使用可能なコアをリストできます。リソースを指定しない場合は、Vivado HLS でどのリソースを使用するかが指定されます。

`set_directive_resource` は、ライブラリのどのメモリ エレメントを使用して配列をインプリメントするか指定するために最もよく使用されます。これにより、たとえば、配列をシングル ポート RAM とデュアル ポート RAM のどちらとしてインプリメントするかを指定できます。RTL のポートは配列にに関連付けられたメモリによって決まるので、これは最上位関数インターフェイスの配列に特に重要な方法です。

`<location>` - 変数を含む関数[/ラベル] のフォーマットでロケーションを指定します。

`<variable>` - 変数名を指定します。

### オプション

`-core <string>` - テクノロジ ライブラリで指定されているように、コアの名前を指定します。

`-port_map <string>` - IP 生成フローを使用してポート マップを指定するために使用するオプションで、アダプターのポートを使用してデザインにポートがマップされます。このオプションの引数は、デザイン ポートおよびアダプター ポートの Tcl リストです。

`-metadata <string>` - IP 生成フローを使用する際にバス オプションを指定するために使用するオプションです。このオプションの引数は、バス操作指示子のクオーテーションで囲まれたリストです。

### pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP resource \
 variable=<variable> \
 core=<core>
```

### 例

この例では、変数 `coeffs[128]` が最上位関数の `foo_top` に対する引数です。この指示子により、`coeffs` がライブラリからの `RAM_1P` コアを使用してインプリメントされるように指定されます。`coeffs` の値にアクセスするために RTL で作成されるポートが、`RAM_1P` コアで定義されるものになります。

```
set_directive_resource -core RAM_1P foo_top coeffs
#pragma AP resource variable=coeffs core=RAM_1P
```

関数 foo に Result=A\*B というコードがあるとします。この例では、Mul2S という 2 段のパイプライン乗算器コアを使用して乗算がインプリメントされるように指定します。

```
set_directive_resource -core Mul2S foo Result
#pragma AP resource variable=Result core=Mul2S
```

## set\_directive\_top

### 構文

```
set_directive_top [OPTIONS] <location>
```

### 説明

この指示子は関数に名前を付けます。この名前を set\_top コマンドで使用できます。これは通常 C++ のクラスのメンバー関数を合成するために使用されます。

指示子は、アクティブソリューションで指定してから、その新しい名前で set\_top コマンドを使用する必要があります。

<location> - 名前を変更する関数の名前

### オプション

-name <string> - set\_top コマンドで使用される名前を指定します。

### pragma

C ソース コードの必要なロケーションの境界内に pragma を挿入する必要があります。

フォーマットおよびオプションは次のようになります。

```
#pragma AP top \
 name=<string>
```

### 例

この例では、foo\_long\_name 関数の名前が DESIGN\_TOP に変更され、最上位として指定されています。コード内に pragma が置かれる場合でも、set\_top コマンドを含める必要があります。または、GUI のプロジェクト設定で最上位を指定する必要があります。

```
set_directive_top -name DESIGN_TOP foo_long_name
#pragma AP top name=DESIGN_TOP
set_top DESIGN_TOP
```

# set\_directive\_unroll

## 構文

```
set_directive_unroll [OPTIONS] <location>
```

## 説明

このコマンドは `set_directive_loop_unroll` に変わっており (ここには同じ引数とオプションを記述しておきます)、今後廃止される予定です。

ループ本文のコピーを複数作成することでループを変換します。

ループは、ループ変換で指定されている反復回数分実行されます。反復の回数は、ループ本文内の任意ロジックによっても変わります (ループ終了変数へのブレークや変更など)。ループは RTL にロジックのブロックによりインプリメンテーションされます。このブロックはループ本文を表し、同じ反復回数分実行されます。

`set_directive_unroll` コマンドを使用すると、ループを完全に展開することができます、RTL にループ反復回数と同じ数のループ本文のコピーを作成できます。または、ループを係数 N で部分的に展開し、N 数のループ本文のコピーを作成し、それに応じてループ反復数を調整することができます。

部分展開に使用される係数 N が元のループ反復数の整数倍数でない場合は、ループ本文の展開された箇所の終わりごとに元の終了条件をチェックする必要があります。

ループを完全に展開するには、ループの境界がコンパイル時に認識される必要があります。これは部分展開には必要ありません。

`<location>` - 関数[/ラベル] フォーマットで展開するループのロケーションを指定します。

## オプション

**-factor <integer>** - 部分展開の係数をゼロでない整数値を指定します。ループ本文は、この数値で指定された回数分繰り返され、反復もそれに応じて調整されます。

**-region** - 任意ループ内にあるすべてのループを展開するときに指定します。ただし指定したループそのものは展開されません。

次の例では、ループ `loop_1` 内に `loop_2` および `loop_3` というループがループ階層の同じレベルにあります。`loop_1` など、指定されたループは、`{ }` かっこで囲まれたコードの一部でもあります。展開指示子が `<function>/loop_1` に指定されている場合、`loop_1` が展開されます。

`-region` オプションは、指定されたリージョンに含まれるループにのみ適用されます。`loop_1` は展開されず、その内側にあるすべてのループ (`loop_2` および `loop_3`) が展開されます。

**-skip\_exit\_check** - 係数が指定されている場合 (部分展開) にのみ、このオプションは有効です。

- 固定境界 - 反復回数が係数の倍数である場合は、終了条件がチェックされません。反復回数が係数の整数倍数である場合は、展開が実行されず、警告が出力されます (終了チェックは処理続行のために必ず実行されます)。
- 可変境界 - 終了条件チェックが削除されます。可変境界が係数の整数倍数であり、終了チェックが不要であることを確認してください。

## pragma

C ソース コードの必要なロケーションの境界内に `pragma` を挿入する必要があります。

フォーマットおよびオプションは次のようにになります。

```
#pragma AP unroll \
 skip_exit_check \
 factor=<int> \
 region
```

### 例

次の例では、関数 `foo` のループ `L1` を展開します。 `pragma` は `L1` の本文に挿入する必要があります。

```
set_directive_unroll foo/L1
#pragma AP unroll
```

次の例では、関数 `foo` のループ `L2` を係数 4 で展開し、最終チェックが削除されるように指定します。 `pragma` は `L2` の本文に挿入する必要があります。

```
set_directive_unroll -skip_exit_check -factor 4 foo/L2
#pragma AP unroll skip_exit_check factor=4
```

次の例では、関数 `foo` のループ `L3` が展開されますが、ループ `L3` 自体は展開されません。 `-region` オプションでは、展開するループを指定しますが、実際に展開されるのはその内側だけです。

```
set_directive_unroll -region foo/L3
#pragma AP unroll region
```

## set\_part

### 構文

```
set_part <device_specification>
```

### 説明

`set_part` コマンドは現在のソリューションのターゲット デバイスを設定します。このコマンドはアクティブ ソリューションに対してのみ実行されます。

`<device_specification>` - Vivado HLS での合成およびインプリメンテーションのターゲット デバイスを設定します。

`<device_family>` - デバイス仕様は単にデバイス ファミリ名だけにできます。これにより、そのファミリ内のデフォルト デバイスが使用されます。

`<device><package><speed_grade>` - デバイス仕様はデバイス、パッケージ、スピード グレード情報を含むターゲット デバイス名にもできます。

## pragma

set\_part コマンドと同等の pragma はありません。

### 例

Vivado HLS に含まれる FPGA ライブラリは、次の例に示すように、デバイス ファミリ名を指定するだけで現在のソリューションに追加できます。この場合、このデバイス ファミリに対して Vivado HLS の FPGA ライブラリで指定されたデフォルトのデバイス、パッケージおよびスピード グレードが使用されます。

```
set_part virtex6
```

Vivado HLS に含まれる FPGA ライブラリは、特定のデバイスをパッケージおよびスピード グレード情報を使用して指定することもできます。

```
set_part xc6vlx240tff1156-1
```

---

## set\_top

### 構文

```
set_top <top>
```

### 説明

set\_top コマンドは合成される最上位関数を定義します。この関数で呼び出される関数もすべてデザインの一部になります。

<top> - 合成される関数の名前を指定します。

## pragma

set\_top コマンドと同等の pragma はありません。

### 例

次の例は、最上位関数を foo\_top として設定しています。

```
set_top foo_top
```

## その他のリソース

---

### ザイリンクス リソース

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

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

ザイリンクス資料で使用される用語集は、次を参照してください。

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

---

### ソリューションセンター

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

---

### リファレンス

- Vivado Design Suite 2012.2 関連の資料  
([http://japan.xilinx.com/support/documentation/dt\\_vivado\\_vivado2012-2.htm](http://japan.xilinx.com/support/documentation/dt_vivado_vivado2012-2.htm))