

Vivado Design Suite ユーザー ガイド

高位合成

UG902 (v2020.1) 2021 年 5 月 4 日

この資料は表記のバージョンの英語版を翻訳したもので、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。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2021 年 5 月 4 日 バージョン 2020.1	
C++ クラスおよびテンプレート	コンストラクター、デストラクター、および仮想関数のサポートについて詳しく説明したセクションを削除。
config_export	「オプション」サブセクションのコマンドをアップデート。
config_sdx	「オプション」サブセクションのコマンドをアップデート。

目次

改訂履歴.....	2
第 1 章: 高位合成.....	5
高位合成の利点.....	5
高位合成の基礎.....	6
Vivado HLS の理解.....	11
Vivado HLS の使用.....	17
効率的なハードウェアのためのデータ型.....	64
インターフェイスの管理.....	70
デザインの最適化.....	106
RTL の検証.....	159
RTL デザインのエクスポート.....	171
第 2 章: 高位合成の C ライブラリ.....	177
任意精度型ライブラリ.....	177
HLS ストリーム ライブラリ.....	190
HLS 数学ライブラリ.....	199
HLS ビデオ ライブラリ.....	208
HLS IP ライブラリ.....	208
HLS 線形代数ライブラリ.....	236
HLS DSP ライブラリ.....	246
HLS SQL ライブラリ.....	247
第 3 章: 高位合成コーディング スタイル.....	249
サポートされない C コンストラクト.....	249
C テストベンチ.....	253
関数.....	259
RTL ブラック ボックス.....	260
ループ.....	266
配列.....	273
データ型.....	281
C ビルトイン関数.....	303
ハードウェア効率の良い C コード.....	304
C++ クラスおよびテンプレート.....	320
アサート.....	326
SystemC の合成.....	328
第 4 章: 高位合成リファレンス ガイド.....	346
コマンド リファレンス.....	346

GUI リファレンス.....	411
インターフェイス 合成リファレンス.....	414
AXI4-Lite スレーブの C ドライバーのリファレンス.....	430
HLS ビデオ関数ライブラリ.....	442
HLS 線形代数関数.....	442
HLS DSP ライブラリ関数.....	450
HLS SQL ライブラリ関数.....	463
C の任意精度型.....	466
C++ の任意精度型.....	478
C++ の任意精度固定小数点型.....	495
SystemC 型と Vivado HLS 型の比較.....	515
RTL ブラックボックスの JSON ファイル.....	521
付録 A: その他のリソースおよび法的通知.....	525
ザイリンクス リソース.....	525
Documentation Navigator およびデザイン ハブ.....	525
参考資料.....	525
お読みください: 重要な法的通知.....	526

高位合成

ザイリンクス Vivado® 高位合成 (HLS) は、C 仕様をレジスタ トランスファー レベル (RTL) コードに変換し、ザイリンクス FPGA (フィールド プログラマブル ゲート アレイ) に合成できる状態にするツールです。C 仕様は、C、C++、または SystemC で記述できます。FPGA を使用すると、従来のプロセッサよりもパフォーマンス、コスト、消費電力の点で優れた並列アーキテクチャを達成できます。この章では、高位合成の概要を説明します。

注記: FPGA アーキテクチャおよび Vivado HLS の基本的な概念については、『Vivado 高位合成を使用した FPGA デザインの概要』 ([UG998](#)) を参照してください。

高位合成の利点

高位合成にはハードウェア ドメインとソフトウェア ドメインの橋渡しをする役割があり、主に次のような利点があります。

- ハードウェア設計における生産性の改善
ハードウェア設計者は、高い抽象度で設計しながら、高パフォーマンスのハードウェアを作成できます。
- ソフトウェア設計におけるシステム パフォーマンスの改善
ソフトウェア設計者は、アルゴリズムの計算負荷の高い部分を新しいコンパイル ターゲット (FPGA) に移動することによりアクセラレーションを達成できます。

高位合成設計手法を使用すると、次が可能になります。

- C レベルでアルゴリズムを開発
開発に時間のかかるインプリメンテーションの詳細を抽象化したレベルで作業できます。
- C レベルで検証
機能の正しさを従来のハードウェア記述言語よりもすばやく検証できます。
- 最適化指示子を使用して C 合成プロセスを制御
パフォーマンスの高い特定のハードウェア インプリメンテーションを作成できます。
- 最適化指示子を使用して C ソース コードから複数のインプリメンテーションを作成
さまざまなデザイン オプションを試すことで、最適なインプリメンテーションを見つけられる可能性が高くなります。
- 解読しやすく移植可能な C ソース コードを作成
C ソースのターゲットを別の FPGA デバイスに変更したり、C ソースを新しいプロジェクトに組み込むことができます。

高位合成の基礎

高位合成には、次の段階が含まれます。

- スケジューリング

各クロック サイクルでどの演算を実行するかを次に基づいて決定します。

- ・ クロック サイクルの長さまたはクロック周波数
- ・ 演算が終了するまでにかかる時間 (ターゲット デバイスで定義)
- ・ ユーザー仕様の最適化指示子

クロック周期が長い場合や、より高速な FPGA がターゲットの場合、1 つのクロック サイクルでより多くの演算を完了できるので、すべての演算を 1 クロックで完了することもあります。クロック周期が短い場合や、より低速な FPGA がターゲットの場合は、演算が自動的に複数のクロック サイクルにスケジューリングされ、マルチサイクル リソースとしてインプリメントする必要のある演算がある場合もあります。

- バインディング

スケジューリングされた各演算をどのハードウェア リソースにインプリメントするのかを決定します。最適なリソースをインプリメントするため、ターゲット デバイスに関する情報が使用されます。

- 制御ロジックの抽出

制御ロジックを抽出して、RTL デザインで演算を順序付ける有限ステート マシン (FSM) を作成します。

高位合成では、C コードが次のように合成されます。

- 最上位関数の引数は RTL の I/O ポートに合成
- C 関数は RTL 階層のブロックに合成

C コードにサブ関数の階層が含まれる場合、最終的な RTL デザインに、元の C 関数の階層と 1:1 で対応するモジュールまたはエンティティの階層が含まれます。関数のすべてのインスタンスに同じ RTL インプリメンテーションまたはブロックが使用されます。

- C 関数のループはデフォルトでは非展開

ループが展開されていない場合、合成ではそのループの 1 反復に対してロジックが作成され、RTL デザインでこのロジックがループの反復ごとに順に実行されます。最適化指示子を使用すると、ループを展開でき、反復が並列実行されるようになります。ループも、有限ステート マシンの細粒度インプリメンテーション (ループ パイプライン) またはより粗い粒度のハンドシェイク ベースのインプリメンテーション (データフロー) のいずれかを使用してパイプライン処理できます。

- C コードの配列は最終的な FPGA デザインではブロック RAM または UltraRAM に合成

配列が最上位関数インターフェイスにある場合は、高位合成でその配列がデザイン外のブロック RAM にアクセスするためのポートとしてインプリメントされます。

高位合成で、デフォルト動作、制約および指定された最適化指示子に基づいて最適化されたインプリメンテーションが作成されます。最適化指示子を使用すると、内部ロジックと I/O ポートのデフォルト動作を変更および制御でき、同じ C コードからさまざまなハードウェア インプリメンテーションを生成できます。

デザインが要件を満たしているかどうか判断するには、高位合成で生成された合成レポートのパフォーマンス メトリクスを確認してください。レポートを解析すると、最適化指示子を使用してインプリメンテーションをさらに改良できます。合成レポートには、次のパフォーマンス メトリクスに関する情報が含まれます。

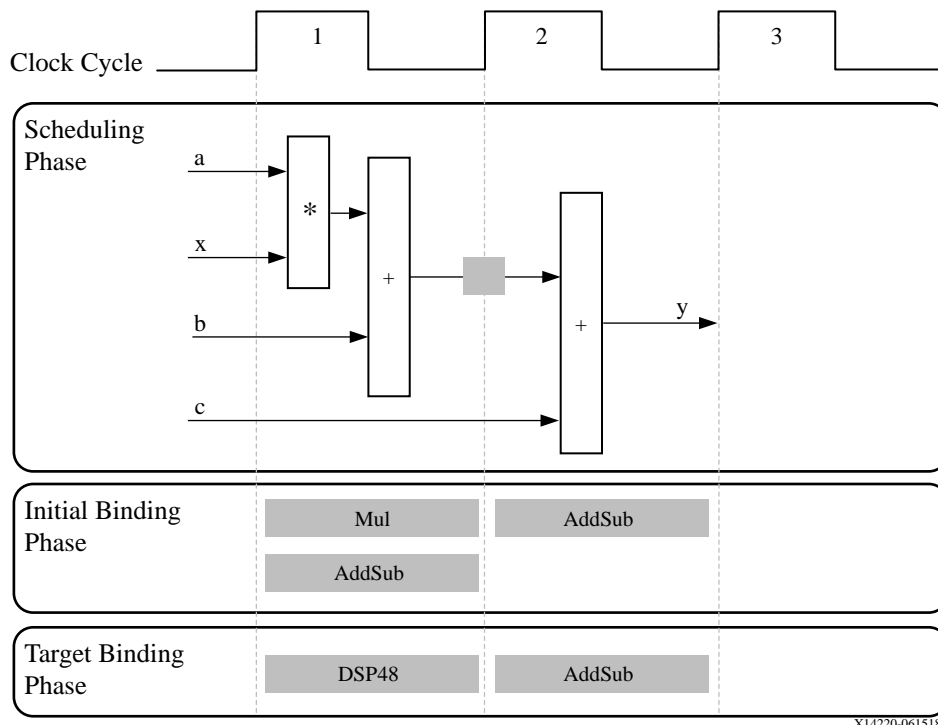
- エリア: ルックアップ テーブル (LUT)、レジスタ、ブロック RAM、DSP48 など、FPGA で使用可能なリソースに基づいてデザインをインプリメントするのに必要なハードウェア リソースの量。
- レイテンシ: 関数がすべての出力値を計算するのに必要なクロック サイクル数。
- 開始間隔 (II): 関数が新しい入力データを受信できるようになるまでのクロック サイクル数。
- ループの反復レイテンシ: ループの 1 反復を完了するのにかかるクロック サイクル数。
- ループ開始間隔: ループの次の反復でデータの処理が開始するまでのクロック サイクル数。
- ループ レイテンシ: ループのすべての反復を実行するのにかかるサイクル数。

スケジューリングおよびバインディングの例

次の図に、このコード例のスケジューリングおよびバインディングの例を示します。

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```

図 1: スケジューリングおよびバインディングの例



X14220-061518

この例のスケジューリング段階では、次の演算が各クロック サイクルで発生するようにスケジューリングされます。

- 1 番目のクロック サイクル: 乗算と 1 回目の加算
- 2 つ目のクロック サイクル: 2 回目の加算と出力生成

注記: 図の 1 番目と 2 番目のクロック サイクルの間にある正方形は、内部レジスタに変数が格納されることを示しています。この例では、高位合成で加算の出力が 1 クロック サイクルでレジスタに入力されることだけが必要です。1 番目のサイクルで `x`、`a`、`b` データ ポートが読み出され、2 番目のサイクルで `c` データ ポートが読み出されて `y` 出力が生成されます。

最終的なハードウェア インプリメンテーションでは、最上位関数への引数が I/O (入力および出力) ポートとしてインプリメントされます。この例では、引数は単純なデータ ポートです。各入力変数は `char` 型なので、入力データ ポートはすべて 8 ビット幅です。関数 `return` は 32 ビットの `int` 型なので、出力データ ポートは 32 ビット幅です。



重要: ハードウェアに C コードをインプリメントする利点は、すべての演算をより少ないクロック数で完了できることです。この例の場合、演算は 2 クロック サイクルだけで完了します。CPU では、このような単純なコード例でも、完了するのにさらに多くのクロック サイクルが必要です。

この例の初期バインディング段階では、乗算が組み合わせ乗算器 (Mul) を使用してインプリメントされ、両方の加算が組み合わせ加減算器 (AddSub) を使用してインプリメントされています。

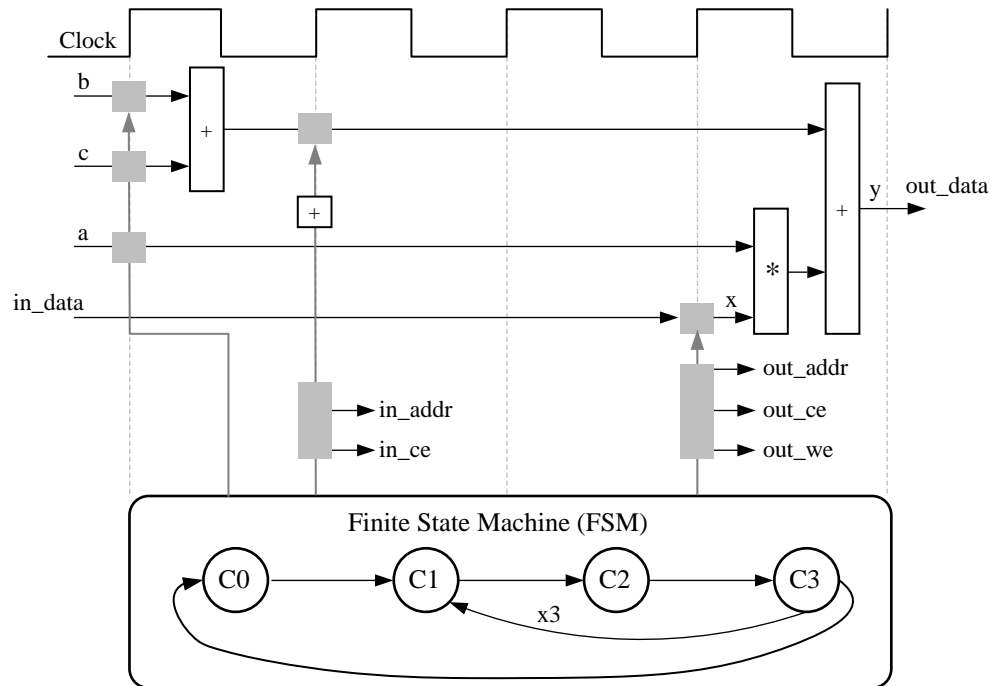
ターゲット バインディング段階では、乗算器と加算演算の 1 つが DSP48 リソースを使用してインプリメントされています。DSP48 リソースは、FPGA アーキテクチャで使用可能な計算ブロックで、高パフォーマンスと効率的なインプリメンテーションの理想的なバランスを達成します。

制御ロジックの抽出と I/O ポートのインプリメント例

次の図に、ここに示すコード例の制御ロジックの抽出と I/O ポートのインプリメンテーションを示します。

```
void foo(int in[3], char a, char b, char c, int out[3]) {
    int x,y;
    for(int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x + b + c;
        out[i] = y;
    }
}
```


図 2: 制御ロジックの抽出と I/O ポートのインプリメンテーションの例



X14218

このコード例では前の例と同じ演算が実行されますが、演算が for ループ内で実行され、関数引数のうちの 2 つが配列である点が異なります。結果のデザインでは、コードがスケジューリングされたときに for ループ内のロジックが 3 回実行されます。高位合成では C コードから制御ロジックが自動的に抽出され、RTL デザインでこれらの演算を順序付ける有限状態マシン (FSM) が作成されます。最終的な RTL デザインでは、最上位関数の引数がポートとしてインプリメントされます。char 型のスカラー変数は標準の 8 ビット データ バス ポートにマップされます。in および out などの配列引数には、データ コレクション全体が含まれます。

高位合成では、配列はデフォルトでブロック RAM に合成されますが、FIFO、分散 RAM、個別のレジスタなどに合成することも可能です。最上位関数で配列を引数として使用すると、ブロック RAM が最上位関数外にあると想定され、データ ポート、アドレス ポート、必要なチップ イネーブルまたはライト イネーブル信号など、デザイン外のブロック RAM にアクセスするためのポートが自動的に作成されます。

FSM では、データをいつレジスタに格納するかと、I/O 制御信号のステートが制御されます。FSM は C0 ステートで開始し、次のクロックで C1 ステートに遷移し、その後 C2 ステート、C3 ステートに遷移します。そして C1 (および C2、C3) を 3 回反復してから、C0 ステートに戻ります。

注記: これは、C コードの for ループの制御構造と類似しています。ステートの完全なシーケンスでは、C0、{C1, C2, C3}、{C1, C2, C3}、{C1, C2, C3} の後、C0 に戻ります。

b と *c* の加算が必要なのは 1 回だけなので、この演算は for ループ外に出され、C0 ステートに挿入されます。この加算結果は、C3 ステートに遷移するたびに再利用されます。

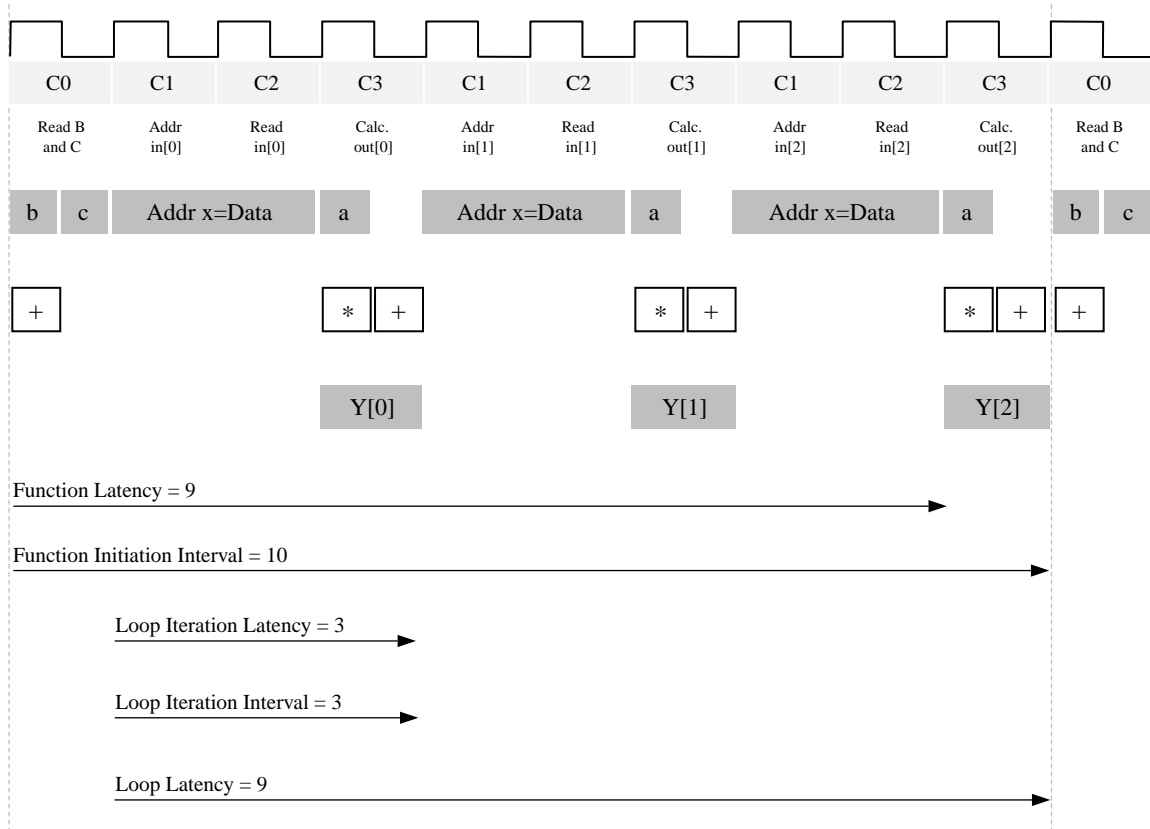
デザインでは *in* のデータが読み出され、*x* に格納されます。FSM の C1 で最初の要素のアドレスが生成されます。また、C1 ステートでは、C1、C2、C3 ステートを何回反復する必要があるのかを知るために、加算器がインクリメントされます。C2 ステートでは、ブロック RAM から *in* のデータが返され、変数 *x* として格納されます。

計算に必要なその他の値が a ポートから読み出され、最初の y 出力が生成されます。FSM で正しいアドレスと制御信号が生成され、この値がブロック外に格納されます。この後デザインは $C1$ ステートに戻り、配列/ブロック RAM の in から次の値が読み出されます。このプロセスは、すべての出力が書き込まれるまで継続されます。そして $C0$ ステートに戻り、 b と c の次の値が読み出されて、プロセスが再度実行されます。

パフォーマンス メトリクス の例

次の図に、前の例のコードの実行 (各クロック サイクルのステート、読み出し、計算、書き込みなど) をサイクルごとに示します。

図 3: レイテンシと開始間隔の例



X14219

この例のパフォーマンス メトリクスは次のとおりです。

- レイテンシ: すべての値を出力するのに 9 クロック サイクルかかります。
注記: 出力が配列の場合、レイテンシは配列の最後の出力値までで計測されます。
- II: II は 10 なので、関数が新しい入力データのセットの読み出しを開始してから次のセットのプロセスを開始するまでに 10 クロック サイクルかかります。
注記: 1 つの関数の実行を完了するのにかかる時間は、1 トランザクションと呼ばれます。この例では、関数が次のトランザクション用のデータを受信できるまでに 11 クロック サイクルかかります。
- ループの反復レイテンシ: 各ループ反復のレイテンシは 3 クロック サイクルです。
- ループ II: 開始間隔は 3 です。

- ループ レイテンシ: レイテンシは 9 クロック サイクルです。

Vivado HLS の理解

ザイリンクス Vivado HLS ツールでは、C 関数がハードウェア システムに統合可能な IP ブロックに合成されます。Vivado HLS はほかのザイリンクス デザイン ツールに緊密に統合されており、ユーザーの C アルゴリズムに最適なインプリメンテーションを作成するための包括的な言語サポートと機能が含まれます。

Vivado HLS デザイン フローは、次のとおりです。

1. C アルゴリズムをコンパイル、実行 (シミュレーション)、およびデバッグ。
2. C アルゴリズムを RTL シミュレーションに合成 (オプションでユーザーの最適化指示子を使用可能)。
3. 包括的なレポートを生成してデザインを解析。
4. プッシュボタン フローを使用して RTL インプリメンテーションを検証。
5. RTL インプリメンテーションを選択した IP フォーマットにパッケージ。

注記: 高位合成では、コンパイル済み C プログラムの実行を「C シミュレーション」と呼びます。C アルゴリズムを実行すると、そのアルゴリズムが正しく機能するかどうかを検証する関数がシミュレーションされます。

入力および出力

Vivado® HLS の入力には、次が使用されます。

- C、C++、または SystemC で記述された C 関数
これは Vivado HLS への主な入力です。この関数には、サブ関数の階層を含めることができます。
- 制約
制約は必須で、クロック周期、クロックのばらつき、FPGA ターゲットなどが含まれます。指定しない場合、クロックのばらつきはデフォルトでクロック周期の 12.5% に設定されます。
- 指示子
指示子はオプションで、合成プロセスで特定の動作または最適化がインプリメントされます。
- C テストベンチおよびその他の関連ファイル
Vivado HLS では C テストベンチを使用して合成前に C 関数がシミュレーションされ、C/RTL 協調シミュレーションを使用して RTL 出力が検証されます。

C 入力ファイル、指示子、および制約は Vivado のグラフィカル ユーザー インターフェイス (GUI) を使用して Vivado HLS プロジェクトにインタラクティブに追加できるほか、コマンド プロンプトに Tcl コマンドを入力しても追加できます。また、Tcl ファイルを作成してバッチ モードでコマンドを実行することもできます。

Vivado HLS の出力には、次が使用されます。

- ハードウェア記述言語 (HDL) 形式の RTL インプリメンテーション ファイル
これは Vivado HLS からの主な出力です。Vivado 合成を使用すると、RTL をゲート レベルのインプリメンテーションおよび FPGA ビットストリーム ファイルに合成できます。RTL は、次の業界標準フォーマットで出力できます。

- 。 VHDL (IEEE 1076-2000)
- 。 Verilog (IEEE 1364-2001)

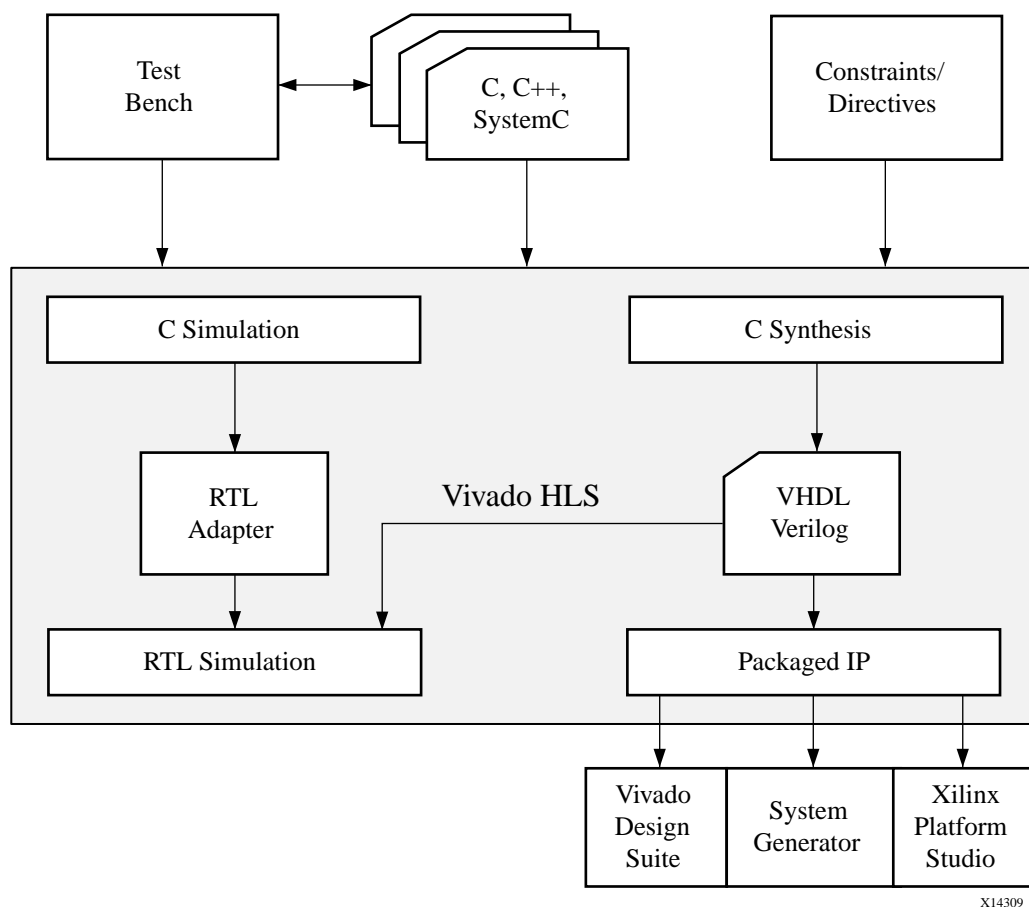
Vivado HLS ではインプリメンテーション ファイルを IP ブロックとしてパッケージし、ザイリンクス デザイン フローのその他のツールで使用できるようにできます。論理合成を使用すると、パッケージした IP を FPGA ビットストリームに合成できます。

• レポート ファイル

この出力は、合成、C/RTL シミュレーションおよび IP パッケージの後に生成されます。

次の図に、Vivado HLS の入力および出力ファイルの概要を示します。

図 4: Vivado HLS のデザイン フロー



テストベンチ、言語サポート、C ライブラリ

C プログラムでは、最上位関数は `main()` と呼ばれます。Vivado® HLS デザイン フローでは、合成で `main()` の下のサブ関数を最上位関数として指定できます。最上位関数 `main()` は合成できません。次はその他の規則です。

- 合成で最上位関数として選択できる関数は 1 つだけです。
- 最上位関数を合成すると、その下の階層にある関数も合成されます。
- 最上位関数の下の階層にない階層を合成するには、関数を合成用に 1 つの最上位関数に統合する必要があります。

テストベンチ

Vivado® HLS デザイン フローを使用する場合、論理的に正しくない C 関数を合成して、インプリメンテーションの詳細を解析し、関数が予測どおりに実行されない原因とつきとめるのは時間の無駄です。生産性を改善するには、合成前にテストベンチを使用して C 関数が正しく機能するかどうかを検証しておく必要があります。

C テストベンチには、`main()` 関数と、合成で最上位関数の下に含まれないサブ関数がすべて含まれます。これらの関数では、合成用に関数にスティミュラスを供給し、その出力を消費することにより、合成用の最上位関数が正しく機能しているかどうかを検証されます。

Vivado HLS ではこのテストベンチを使用して、C シミュレーションがコンパイルされて実行されます。[Launch Debugger] オプションを選択すると、コンパイル プロセス中にフル C デバッグ環境を開いて、C シミュレーションを解析できます。



推奨: Vivado HLS ではテストベンチを使用して合成前に C 関数が検証されて、RTL 出力が自動的に検証されるので、テストベンチを使用することを強くお勧めします。

言語サポート

Vivado HLS では、C コンパイル/シミュレーションに次の規格がサポートされています。

- ANSI-C (GCC 4.6)
- C++ (G++ 4.6)
- SystemC (IEEE 1666-2006、バージョン 2.2)

C、C++、および SystemC 言語コンストラクト

Vivado HLS では多くの C、C++、SystemC 言語コンストラクトと、float および double 型も含めた各言語のすべてのネイティブ データ型がサポートされていますが、合成では次のコンストラクトはサポートされていません。

- ダイナミックなメモリ割り当て
FPGA には決まったリソース セットがあるので、メモリ リソースのダイナミックな作成と解放はサポートされません。
- オペレーティング システム (OS) 操作
FPGA に入出力されるすべてのデータは、入力ポートから読み出されるか、出力ポートに書き込まれる必要があります。ファイルの読み出し/書き込みのような OS 操作または時間や日付のような OS クエリはサポートされません。これらの操作は C テストベンチで実行され、そのデータが合成用に関数引数として関数に渡されます。

サポートされる C コンストラクトとサポートされない C コンストラクトの詳細と主なコンストラクトの例は、[第 3 章: 高位合成コーディング スタイル](#)を参照してください。

C ライブラリ

C ライブラリには、FPGA へのインプリメンテーション用に最適化された関数およびコンストラクトが含まれます。これらのライブラリを使用すると、高い結果の品質 (QoR) を達成できるようになり、最終出力がリソースを最適に使用したパフォーマンスの優れたデザインにします。ライブラリは C、C++、または SystemC で提供されているので、C 関数に組み込んで、合成前に論理的に問題ないかどうかをシミュレーションで検証できます。

Vivado® HLS からは、標準 C 言語を拡張するために次の C ライブラリが提供されています。

- 任意精度データ型

- 半精度 (16 ビット) 浮動小数点データ型
- 数学演算
- FFT (Fast Fourier Transform) および FIR (Finite Impulse Response) を含むザイリンクス IP ファンクション
- シフトレジスタ LUT (SRL) リソースを最大限に使用できるようにする FPGA リソース ファンクション

C ライブラリの例

C ライブラリを使用すると、標準 C 型を使用するよりも高い QoR を達成できます。標準 C 型は、8 ビット境界 (8、16、32、64 ビット) に基づいています。ハードウェア プラットフォームをターゲットにする際は通常、特定のデータ幅のデータ型を使用した方が効率的です。

たとえば、通信プロトコル用のフィルター関数を含むデザインがデータ送信要件を満たすのに 10 ビットの入力データと 18 ビットの出力データを必要とする場合、標準 C データ型を使用すると、入力データが少なくとも 16 ビット、出力データが少なくとも 32 ビット必要です。最終的なハードウェアでは、これにより入力と出力の間に必要よりもデータ幅が広いデータパスが作成されるので、リソースがより多く使用され、遅延も長くなり (32 ビット x 32 ビットの乗算は 18 ビット x 18 ビットの乗算よりも長くなる)、完了するのにさらに多くのクロック サイクルが必要になります。

このデザインで任意精度データ型を使用すると、合成前に C コードで正確なビット サイズを指定でき、アップデートされた C コードをシミュレーションできるので、合成前に C シミュレーションで出力の質を検証できます。任意精度データ型は C および C++ 用に提供されており、1 ~ 1024 ビットまでの幅のデータ型を記述できます。たとえば、最大 32768 ビットまでの C++ 型を記述できます。

注記: Vivado HLS では内部ロジックが最適化され、出力ポートにファンアウトされないデータ ビットおよびロジックは削除されるので、任意精度型は関数の境界でのみ必要となります。

合成、最適化、解析

Vivado® HLS はプロジェクト ベースです。プロジェクトにはそれぞれ 1 セットの C コードが含まれ、複数のソリューションを含めることができます。各ソリューションに異なる制約および最適化指示子を設定できます。Vivado HLS の GUI で、各ソリューションの結果を解析および比較できます。

次に、Vivado HLS デザイン プロセスの合成、最適化、解析の手順を示します。

1. 最初のソリューションでプロジェクトを作成。
2. C シミュレーションがエラーなく実行されることを検証。
3. 合成を実行して結果セットを取得。
4. 結果の解析。

結果を解析したら、プロジェクトに異なる制約および最適化指示子を指定したソリューションを新しく作成して、そのソリューションを合成できます。このプロセスは、デザインが必要なパフォーマンス特性になるまで繰り返すことができます。複数のソリューションを使用することで、前の結果を保持したまま開発を進めることができます。

最適化

Vivado® HLS を使用すると、次を含むさまざまな最適化指示子をデザインに適用できます。

- タスクがパイプライン処理される (現在のタスクが終了する前に次のタスクが開始できる) ように指定。
- 関数、ループ、および領域の完了するまでのレイテンシを指定。
- 使用されるリソースの数を制限。

- コードから継承または暗示された依存度を無効にし、指定した操作を許可。たとえば、ビデオ ストリームのように初期データ値を削除または無視できる場合、パフォーマンスが改善できるのであれば、メモリの書き込み前に読み出しできるようにします。
- I/O プロトコルを選択して、最終的なデザインを同じ I/O プロトコルのほかのハードウェア ブロックに接続。

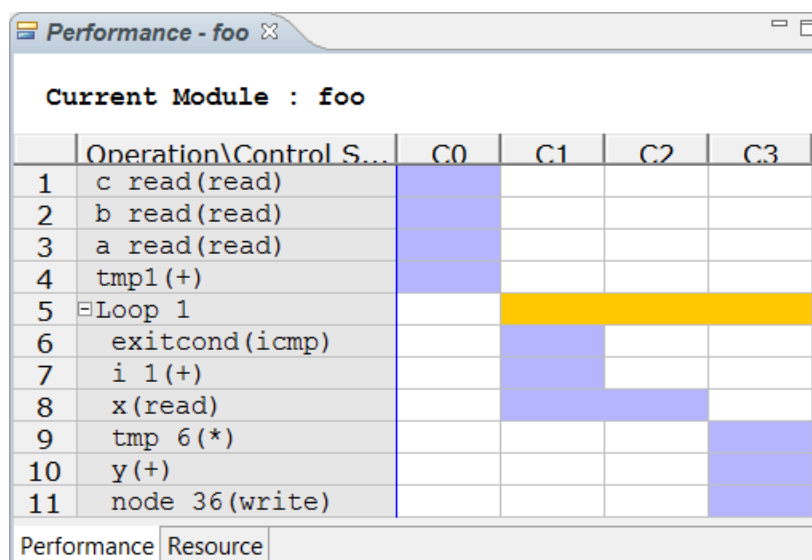
注記: サブ関数で使用する I/O プロトコルは Vivado HLS により自動的に決定されます。ポートにレジスタを付けるかどうかを指定することを除き、これらのポートを制御することはできません。

Vivado HLS の GUI を使用すると、最適化指示子をソース コードに直接配置できます。または、Tcl コマンドを使用して最適化指示子を適用することもできます。

解析

合成が終了すると、Vivado® HLS で合成レポートが自動的に作成され、インプリメンテーションのパフォーマンスを理解するのに使用できます。Vivado HLS の GUI の [Analyze] パースペクティブの [Performance] ビューで、結果をインタラクティブに解析できます。次の図に、**制御ロジックの抽出と I/O ポートのインプリメント例**の [Performance] ビューを示します。

図 5: Vivado HLS の解析例



Current Module : foo		C0	C1	C2	C3
1	c read(read)				
2	b read(read)				
3	a read(read)				
4	tmp1(+)				
5	Loop 1				
6	exitcond(icmp)				
7	i 1(+)				
8	x(read)				
9	tmp 6(*)				
10	y(+)				
11	node 36(write)				

[Performance] ビューには、各ステートの詳細が表示されます。

- C0: 最初のステートで、a、b、c ポートの読み出し操作と加算演算が含まれます。
- C1 および C2: デザインがループに入り、ループのインクリメント カウンターと終了条件がチェックされます。この後、変数 x にデータが読み出されます。これには 2 クロック サイクルかかります。デザインはブロック RAM にアクセスし、1 クロック サイクルでアドレス、次のクロック サイクルでデータを読み出すので、2 クロック サイクル必要です。
- C3: 計算が実行されて y ポートに出力が書き込まれます。この後、ループが開始点に戻ります。

RTL 検証

C テストベンチをプロジェクトに追加すると、それを RTL の機能が元の C と同じかどうかを検証するために使用できます。C テストベンチでは、最上位関数からの出力が合成のために検証され、RTL が論理的に同じである場合は、最上位関数 `main()` に 0 が返されます。Vivado® HLS では、この戻り値が C シミュレーションと C/RTL 協調シミュレーションの両方で使用され、結果が正しいかどうか判断されます。C テストベンチから 0 以外の値が返された場合、Vivado HLS でシミュレーションでエラーが発生したことがレポートされます。詳細は、[テストベンチ要件](#) を参照してください。



ヒント: Vivado HLS では、C/RTL 協調シミュレーションを実行する構造が自動的に作成され、サポートされる次のいずれかの RTL シミュレータを使用してシミュレーションが実行されます。

- Vivado シミュレータ (XSim)
- ModelSim シミュレータ
- VCS
- NCSim
- Riviera
- Xcelium

シミュレーションに Verilog または VHDL HDL を選択した場合、Vivado HLS では指定した HDL シミュレータが使用されます。ザイリンクス デザイン ツールには、Vivado シミュレータが含まれます。サードパーティの HDL シミュレータを使用する場合は、そのサードパーティ ベンダーからのライセンスが必要です。VCS および NCSim シミュレータは Linux OS でのみサポートされます。

RTL のエクスポート

Vivado® HLS を使用すると、RTL をエクスポートし、最終的な RTL 出力ファイルを次の IP 形式のいずれかのザイリンクス IP としてパッケージできます。

- Vivado IP カタログ

Vivado Design Suite の Vivado IP カタログにインポートします。

- System Generator for DSP

System Generator に HLS デザインをインポートします。

- 合成済みチェックポイント (.dcp)

Vivado Design Suite チェックポイントをインポートするのと同じ方法で Vivado Design Suite に直接インポートします。

注記: 合成済みチェックポイント形式を使用すると、論理合成が開始され、RTL インプリメンテーションがゲートレベル インプリメンテーションにコンパイルされ、IP パッケージに含まれます。

合成済みチェックポイント以外のすべての IP 形式に対しては、オプションで Vivado HLS 内から論理合成を実行して、RTL 合成またはインプリメンテーションの結果を評価できます。このオプションの手順により、IP パッケージのハンドオフ前に Vivado HLS で提供されるタイミングおよびエリアの見積もりを確認できます。このゲートレベルの結果は、パッケージした IP には含まれません。

注記: Vivado HLS では、各 FOGA ごとのビルトイン ライブラリに基づいてタイミングおよびエリア リソースが見積もられます。論理合成を使用して RTL をゲート レベルのインプリメンテーションにコンパイルし、FPGA でそのゲートの物理的配置とゲート間の接続の配線を実行した場合は、論理合成でさらに最適化が実行されて、Vivado HLS の見積もりが変わることもあります。

Vivado HLS の使用

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] アイコンをダブルクリックします。

図 6: Vivado HLS の GUI ボタン



Linux で Vivado HLS を起動するには (または Vivado HLS コマンド プロンプトから起動するには)、コマンド プロンプトで次のコマンドを実行します。

```
$ vivado_hls
```

次の図に示す Vivado HLS の GUI が開きます。

図 7: Vivado HLS の GUI の Welcome ページ



[Quick Start] フィールドからは、次のタスクを実行できます。

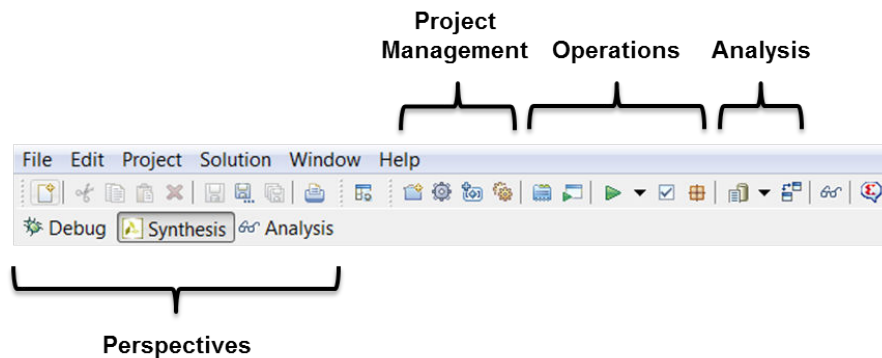
- [Create New Project]: プロジェクト設定ウィザードが起動します。
- [Open Project]: 既存プロジェクトを選択するか、最近使用したプロジェクトのリストから選択します。
- [Open Example Project]: Vivado HLS のサンプル プロジェクトを開きます。

[Documentation] からは、次のタスクを実行できます。

- [Tutorials]: 『Vivado Design Suite チュートリアル: 高位合成』 (UG871) を開きます。
- [User Guide]: この資料、『Vivado Design Suite ユーザー ガイド: 高位合成』 (UG902) を開きます。
- [Release Notes Guide]: 最新のソフトウェア バージョンの『Vivado Design Suite ユーザー ガイド: リリース ノート、インストール、およびライセンス』 (UG973) を開きます。

Vivado HLS を使用する主なコマンド ボタンは、ツールバーに含まれます。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、合成は C/RTL 協調シミュレーションが実行されるよりも前に実行される必要があるため、C/RTL 協調シミュレーション ツールバー ボタンは、合成が終了するまで淡色表示になります。

図 8: Vivado HLS の制御



図の「Project Management」(プロジェクト管理) グループのボタンを左から順に説明します。

- [Create New Project]: New Project ウィザードが開きます。
- [Project Settings]: 現在のプロジェクト設定を変更できます。
- [New Solution]: [New Solution] ダイアログ ボックスが開きます。
- [Solution Settings]: 現在のソリューション設定を変更できます。

ツールバーの「Operation」(ツール操作) グループのボタンを左から順に説明します。

- [Index C Source]: C ソースのアノテーションを更新します。
- [Run C Simulation]: [C Simulation] ダイアログ ボックスが開きます。
- [C Synthesis]: Vivado HLS で C ソース コードを開始します。
- [Run C/RTL Cosimulation]: RTL 出力を検証します。
- [Export RTL]: RTL を必要な IP 出力形式でパッケージします。

ツールバーの「Analysis」(デザイン解析) グループのボタンを左から順に説明します。

- [Open Report]: C 合成レポートを開くか、ドロップダウン リストからその他のレポートを開きます。
- [Compare Reports]: 異なるソリューションからのレポートを比較します。

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。また、Vivado HLS の GUI には、次の 3 つのパーспекティブがあります。パーспекティブを選択すると、ビューが選択したタスクに適した表示に切り替わります。

- [Debug] パerspекティブで C デバッガーが開きます。
- [Synthesis] パerspекティブはデフォルトのパーспекティブで、合成を実行するためのビューが含まれます。
- [Analysis] パerspекティブは、合成終了後にデザインを詳細に解析するために使用します。このパーспекティブには、合成レポートよりも詳細な情報が含まれます。

パーспекティブは、ボタンでいつでも切り替えることができます。

この章の残りの部分では、Vivado HLS の使用方法について説明します。説明するトピックは次のとおりです。

- Vivado HLS 合成レポートの作成方法。
- C コードのシミュレーションおよびデバッグ方法。
- デザインの合成、新しいソリューションの作成、最適化の追加方法。
- デザイン解析の実行方法。
- RTL 出力の検証およびパッケージ方法。
- Vivado HLS Tcl コマンドおよびバッチ モードの使用法。

デザイン サンプル、チュートリアル、その他のリソースの詳細は、この章の最後に示します。

新規合成プロジェクトの作成

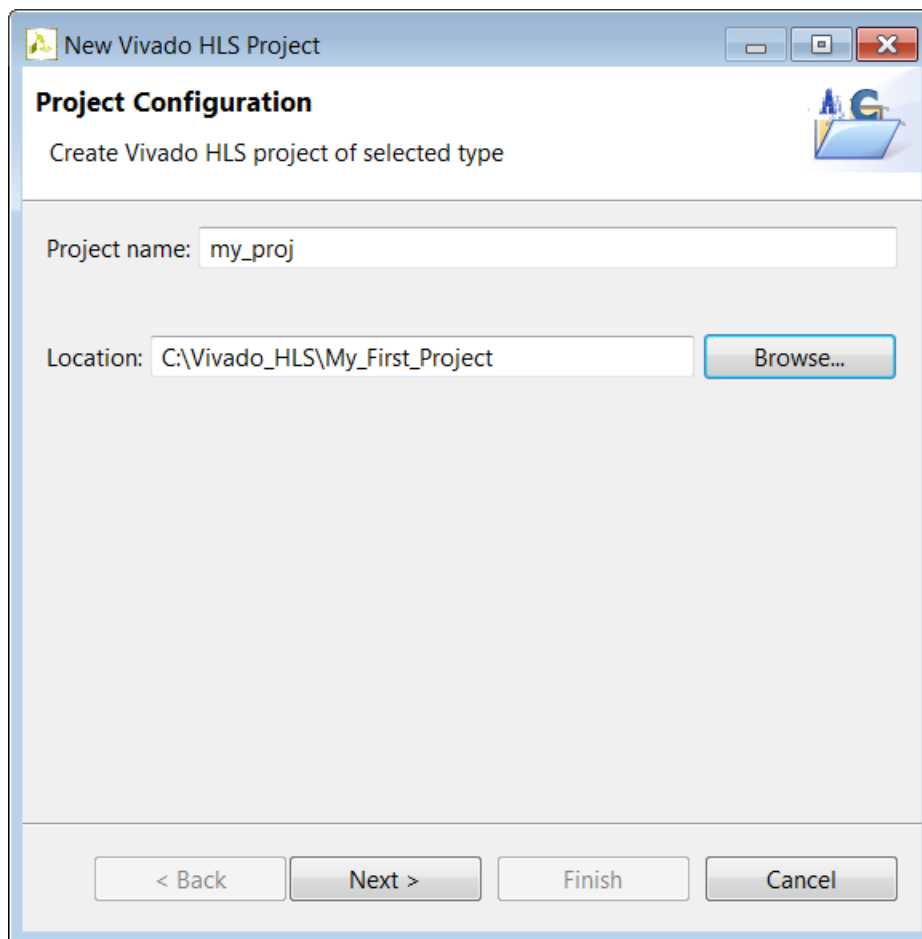
新規プロジェクトを作成するには、ウェルカム ページで [Create New Project] リンクをクリックするか、[File > New Project] メニュー コマンドをクリックします。これで、[新規合成プロジェクトの作成](#)にあるプロジェクト ウィザードが開き、次の設定ができます。

- [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。
- [Location]: プロジェクトを保存するディレクトリを指定します。



注意: Windows オペレーティング システムではパスの長さが 260 文字までに制限されており、これが Vivado ツールに影響する可能性があります。この問題を回避するには、プロジェクトの作成、IP および Manage IP プロジェクトの定義、ブロック デザインの作成を実行する際に、ディレクトリの場所および名前をできるだけ短くしてください。

図 9: プロジェクト仕様



[Next >] ボタンをクリックすると、ウィザードが 2 つ目のページに移動し、プロジェクトの C ソース ファイルの詳細を入力できます ([新規合成プロジェクトの作成](#))。

- [Top Function]: 合成する最上位関数の名前を指定します。C ファイルを最初に追加した場合は、[Browse] ボタンを使用して C 階層を表示し、合成の最上位関数を選択できます。[Browse] ボタンはソース ファイルを追加するまで淡色表示になります。

注記: プロジェクトが SystemC として指定される場合は、Vivado HLS で自動的に最上位関数が認識されるので、この手順は必要ありません。

[Add Files] ボタンを使用すると、プロジェクトにソース コードを追加できます。



重要: [Add Files] ボタン (または Tcl コマンドの `add_files` を使用してプロジェクトにヘッダー ファイル (拡張子は `.h`) は追加できません。

Vivado HLS では、次のディレクトリが検索パスに自動的に追加されます。

- 作業ディレクトリ
注記: 作業ディレクトリには、Vivado HLS のプロジェクト ディレクトリが含まれます。
- プロジェクトに追加する C ファイルを含むディレクトリすべて

これらのディレクトリに含まれるヘッダー ファイルは自動的にプロジェクトに追加されます。その他すべてのヘッダー ファイルへのパスは [Edit CFLAGS] ボタンで指定する必要があります。

[Edit CFLAGS] ボタンでは、C コードをコンパイルするのに必要な C コンパイラ フラグ オプションを指定します。これらのコンパイラ フラグ オプションは、gcc または g++ で使用されるものと同じです。次の例のように、C コンパイラ フラグには、ヘッダー ファイルへのパス名、マクロ仕様、およびコンパイラ指示子が含まれます。

- [-I/project/source/headers]: 関連するヘッダー ファイルへの検索パスを指定します。
注記: プロジェクト ディレクトリではなく、作業ディレクトリに関連する相対パス名を指定する必要があります。
- [-DMACRO_1]: コンパイル中に MACRO_1 を定義します。
- [-fnested-functions]: ネストされた関数を含むデザインに必要な指示子を定義します。

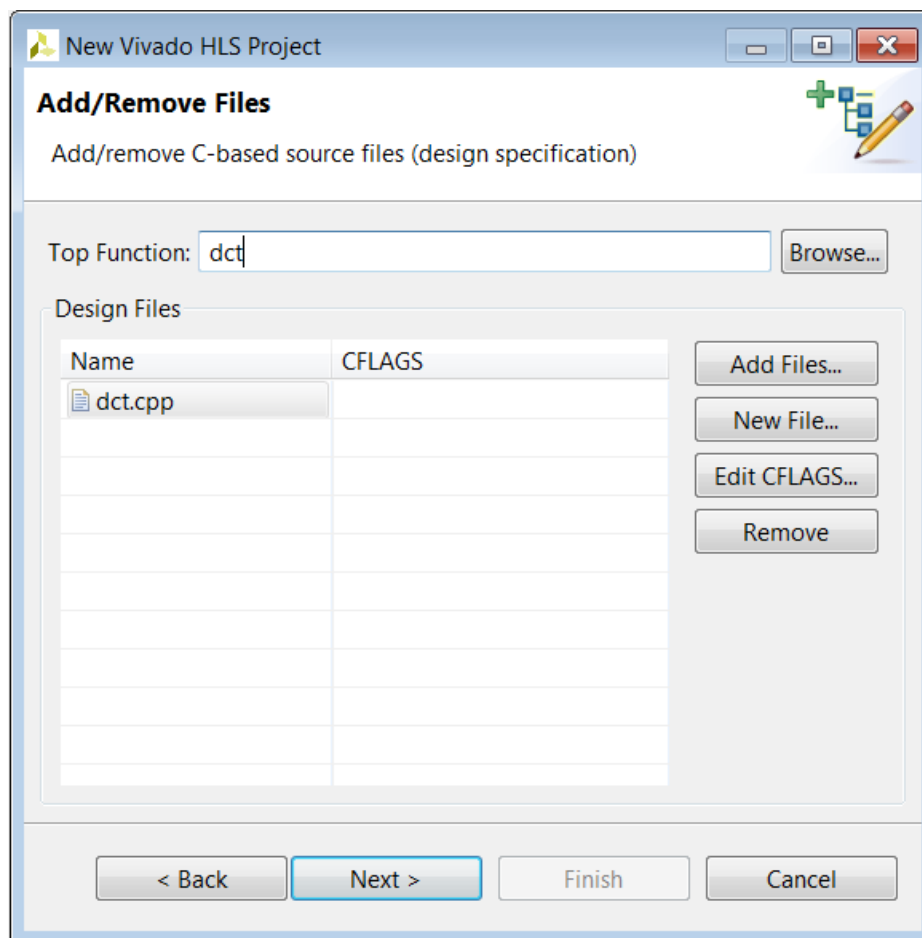


ヒント: サポートされる [Edit CFLAGS] オプションのリストは、GNU コンパイラ コレクション (GCC) のウェブサイトの [Option Summary] ページ (<http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>) を参照してください。



ヒント: `$::env(MY_ENV_VAR)` を使用すると、CFLAGS で環境変数を指定できます。たとえば、`$MY_ENV_VAR/include` というディレクトリをコンパイルには、`-I$::env(MY_ENV_VAR)/include in CFLAGS` と指定します。

図 10: プロジェクトのソース ファイル

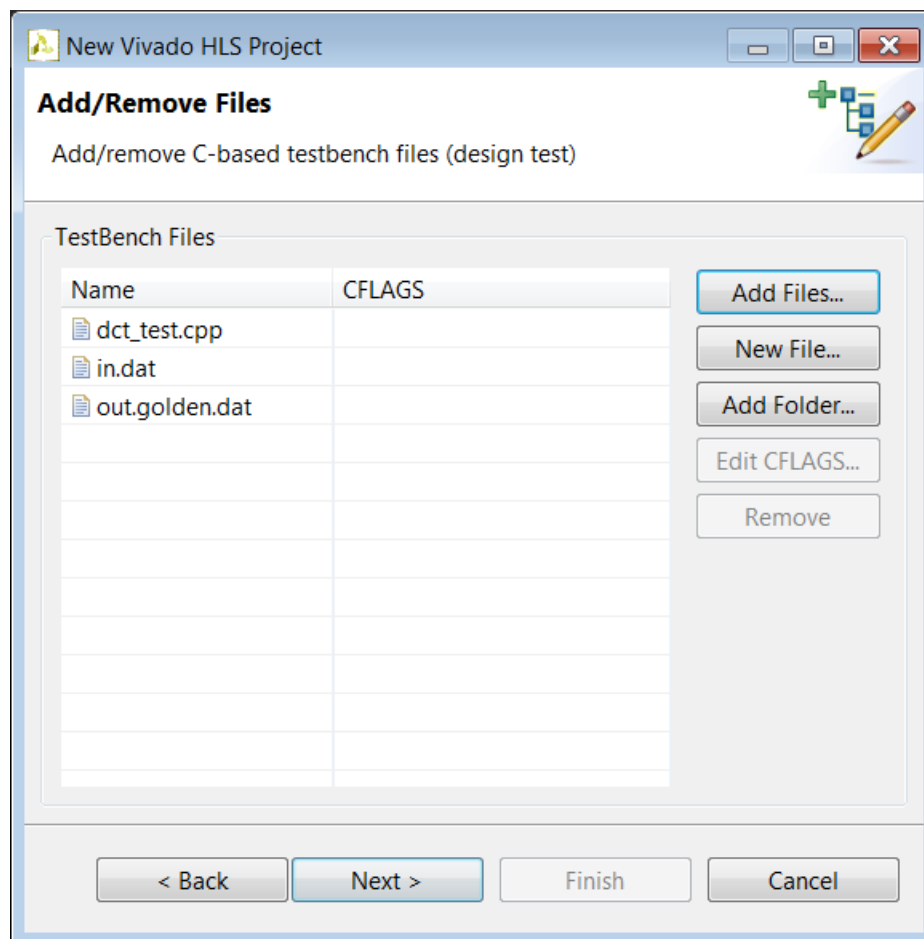


ウィザードの次のページでは、テストベンチに関連するファイルをプロジェクトに追加します。

注記: デザイン ファイルではなくテストベンチに関連付けられたヘッダー ファイルがある SystemC デザインの場合、[Add Files] ボタンを使用してプロジェクトにヘッダー ファイルを追加する必要があります。

Vivado HLS に含まれるほとんどのサンプル デザインの場合、テストベンチはデザインとは別のファイルになっています。テストベンチと合成する関数を別ファイルにしておく、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される関数と同じファイル内に含まれる場合、次の手順で示すように、ソース ファイルとしてテストベンチ ファイルを追加する必要があります。

図 11: プロジェクトのテストベンチ ファイル



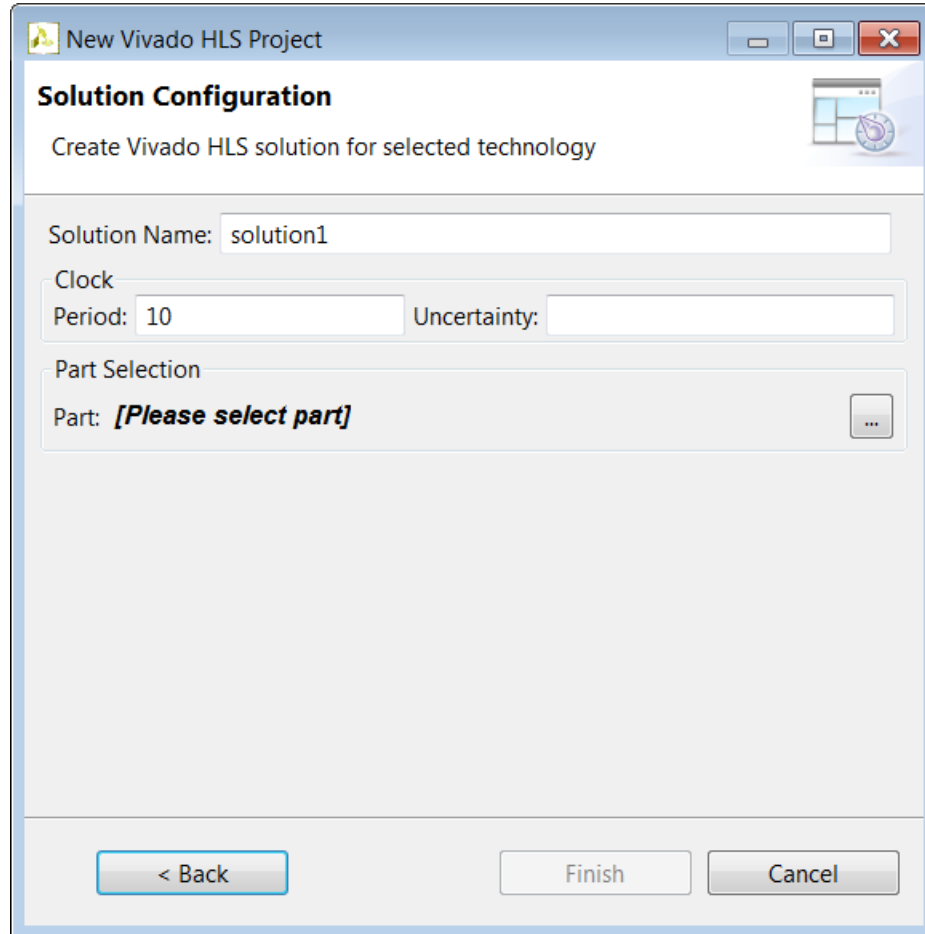
C ソース ファイルの場合と同様、[Add Files] ボタンをクリックして C テストベンチを追加し、[Edit CFLAGS] ボタンをクリックして C コンパイラ オプションを含めます。

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。前述の例では、テストベンチで入力スティミュラスを供給する `in.dat` ファイルと、予測結果を読み込む `out.golden.dat` ファイルが使用されます。テストベンチからこれらのファイルにアクセスされるので、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、[Add Folders] ボタンで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、ここに情報を入力する必要はありません。[Next >] をクリックして、次の図のウィザードの最後のページに進み、最初のソリューションの詳細を指定します。

図 12: 最初のソリューションの設定



New Vivado HLS Project

Solution Configuration

Create Vivado HLS solution for selected technology

Solution Name:

Clock

Period: Uncertainty:

Part Selection

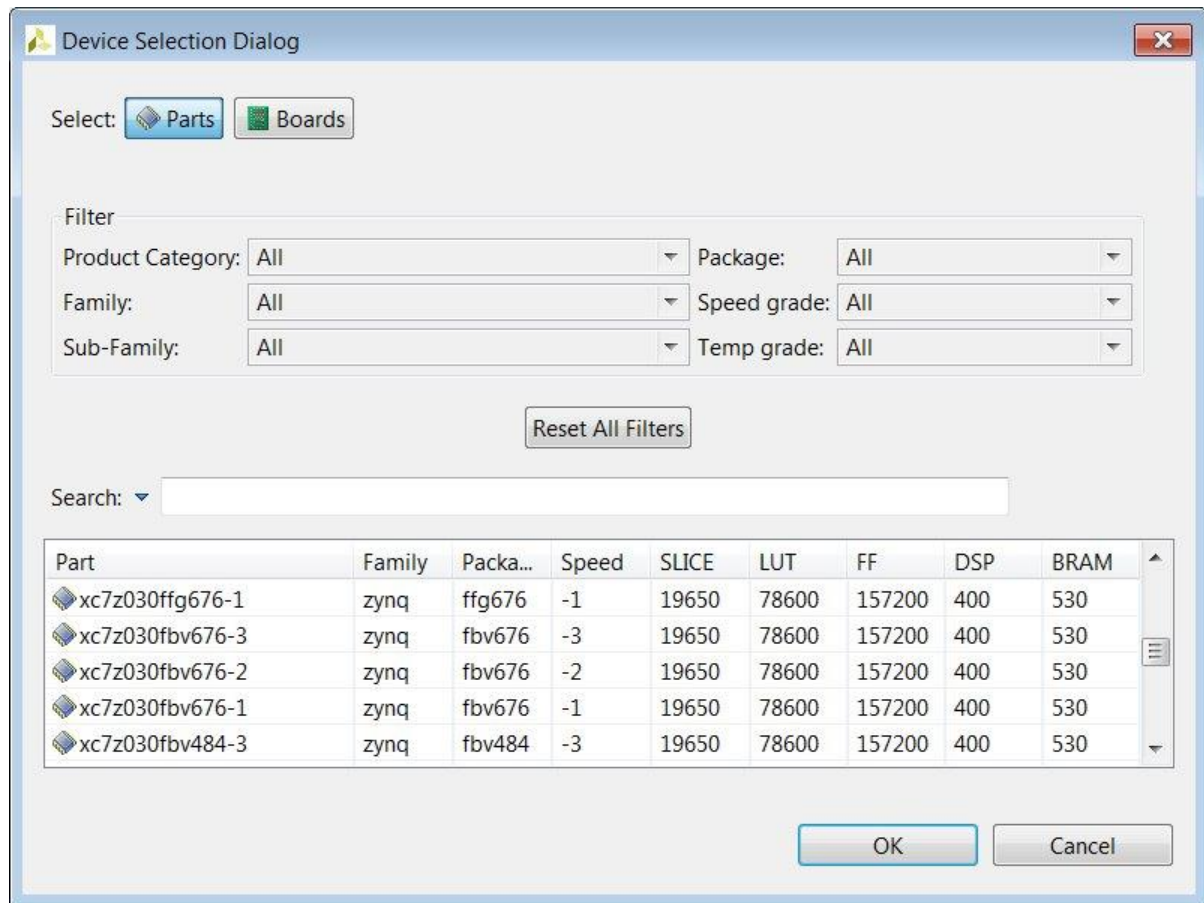
Part: **[Please select part]**

< Back Finish Cancel

新規プロジェクト ウィザードの最後の画面では、最初のソリューションの詳細を指定できます。

- [Solution Name]: Vivado HLS ではデフォルトでソリューション名に「`solution1`」が指定されますが、これは変更できます。
- [Clock Period]: ns または MHz (例: 150 MHz) を付けて指定した周波数の単位でクロック周波数を指定します。
- [Uncertainty]: 合成で使用するクロック周期は、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 では、内部モデルを使用して、各 FPGA の演算の遅延が見積もられます。[Uncertainty] に入力するクロックのばらつきの値では、RTL 合成、配置配線によるネット遅延の増加を考慮に入れた制御可能な差分を指定できます。ナノ秒 (ns) またはパーセント (%) で指定しない場合、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]: 適切なパーツをクリックして選択します。

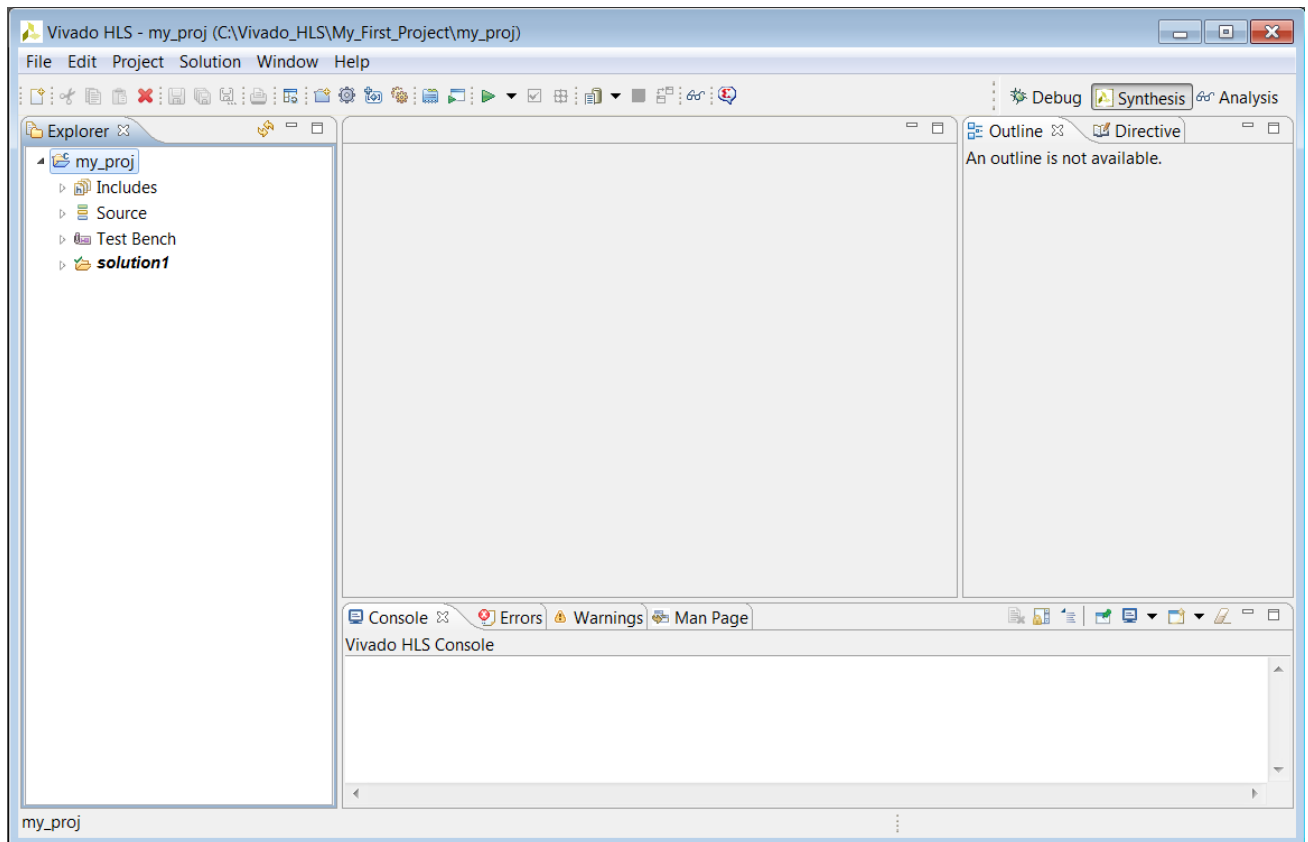
図 13: パーツの選択



ターゲットにする FPGA を選択します。デバイス リストに表示されるデバイス数は、フィルターを使用すると削減されます。ターゲットがボードの場合、左上でボードを選択すると、デバイス リストがサポートされるボードのリストに変わります (Vivado HLS では自動的に正しいターゲット デバイスが選択されます)。

[Finish] をクリックし、プロジェクトを開きます。

図 14: Vivado HLS での新規プロジェクトの GUI



Vivado HLS のグラフィカル ユーザー インターフェイスには、次の 4 つのエリアが含まれます。

- 左側のエクスプローラー エリアでは、プロジェクト階層を表示できます。この階層はディスクのプロジェクト ディレクトリと同じです。
- 真ん中の情報エリアには、ファイルが表示されます。ファイルはエクスプローラー エリアでダブルクリックすると開きます。
- 右側の補足エリアには、情報エリアで開いたファイルに関する情報が表示されます。
- 下部のコンソール エリアには、Vivado HLS を実行したときの出力が表示されます。

C コードのシミュレーション

Vivado® HLS フローでの検証には、大きく分けて 2 つのプロセスが含まれます。

- C プログラムが正しく必要な機能をインプリメントするかどうかを確認するための合成前の検証。
- RTL が正しいかどうか確認する合成後の検証。

どちらのプロセスも、「C シミュレーション」と「C/RTL 協調シミュレーション」と呼ばれます。

合成前に、合成する関数を C シミュレーションを使用したテストベンチで検証する必要があります。C テストベンチには、最上位関数 `main()` と合成される関数が含まれます。その他の関数が含まれることもあります。理想的なテストベンチには、次のような特徴があります。

- テストベンチにはセルフチェック機能があり、合成される関数からの結果が正しいかどうかを検証されます。
- 結果が正しい場合は、テストベンチから `main()` に値 0 が返されます。正しくない場合は、0 以外の値が返されます。


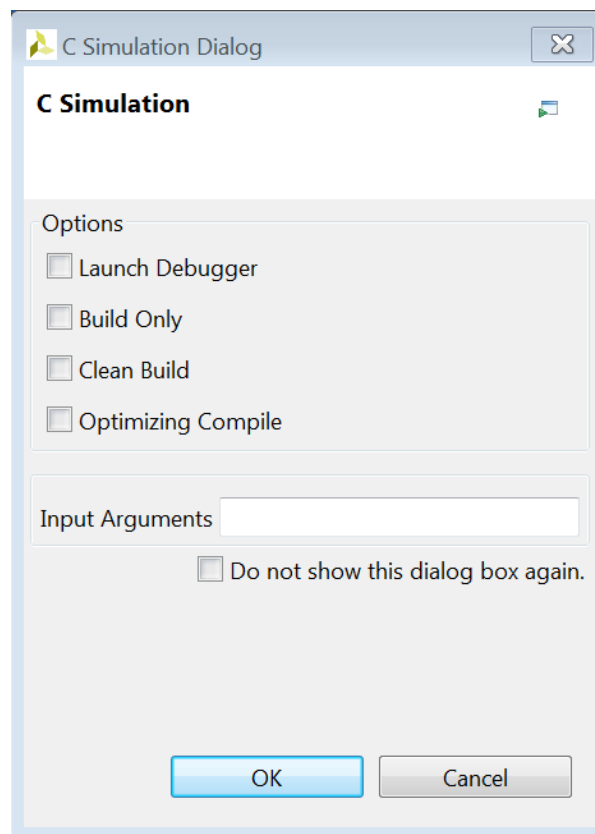
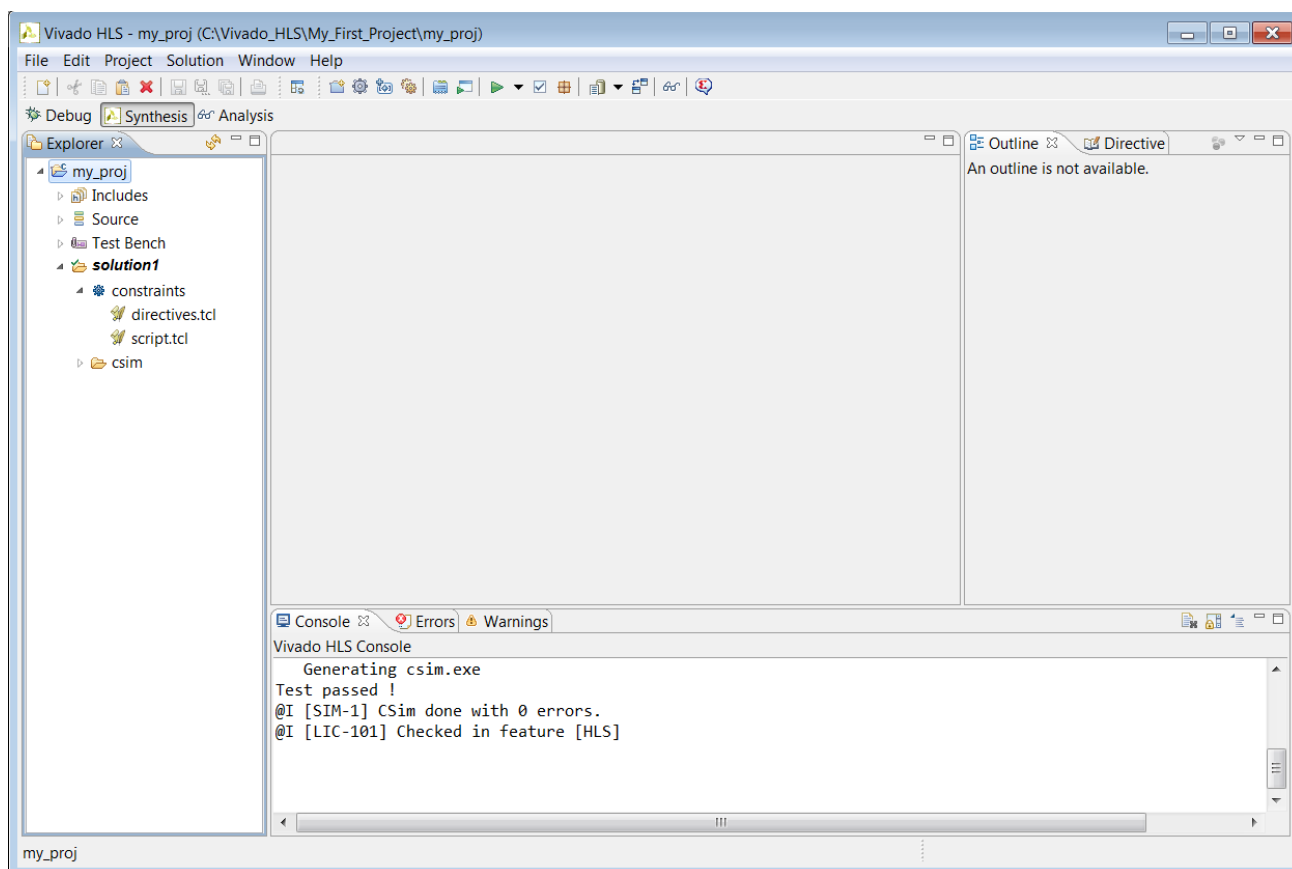
[Run C Simulation] ツールバー ボタン  をクリックすると、[C Simulation] ダイアログ ボックスが開きます。

図 15: [C Simulation] ダイアログ ボックス



ダイアログ ボックスでオプションを選択しなければ、C コードがコンパイルされて、C シミュレーションが自動的に実行されます。結果は次の図のようになります。C コードが問題なくシミュレーションされたら、[Console] ウィンドウに次のようなメッセージが表示されます。テストベンチは、「Test Passed!」というメッセージと一緒に、使用された `printf` コマンドをコンソールに返します。

図 16: C デザインのコンパイル結果

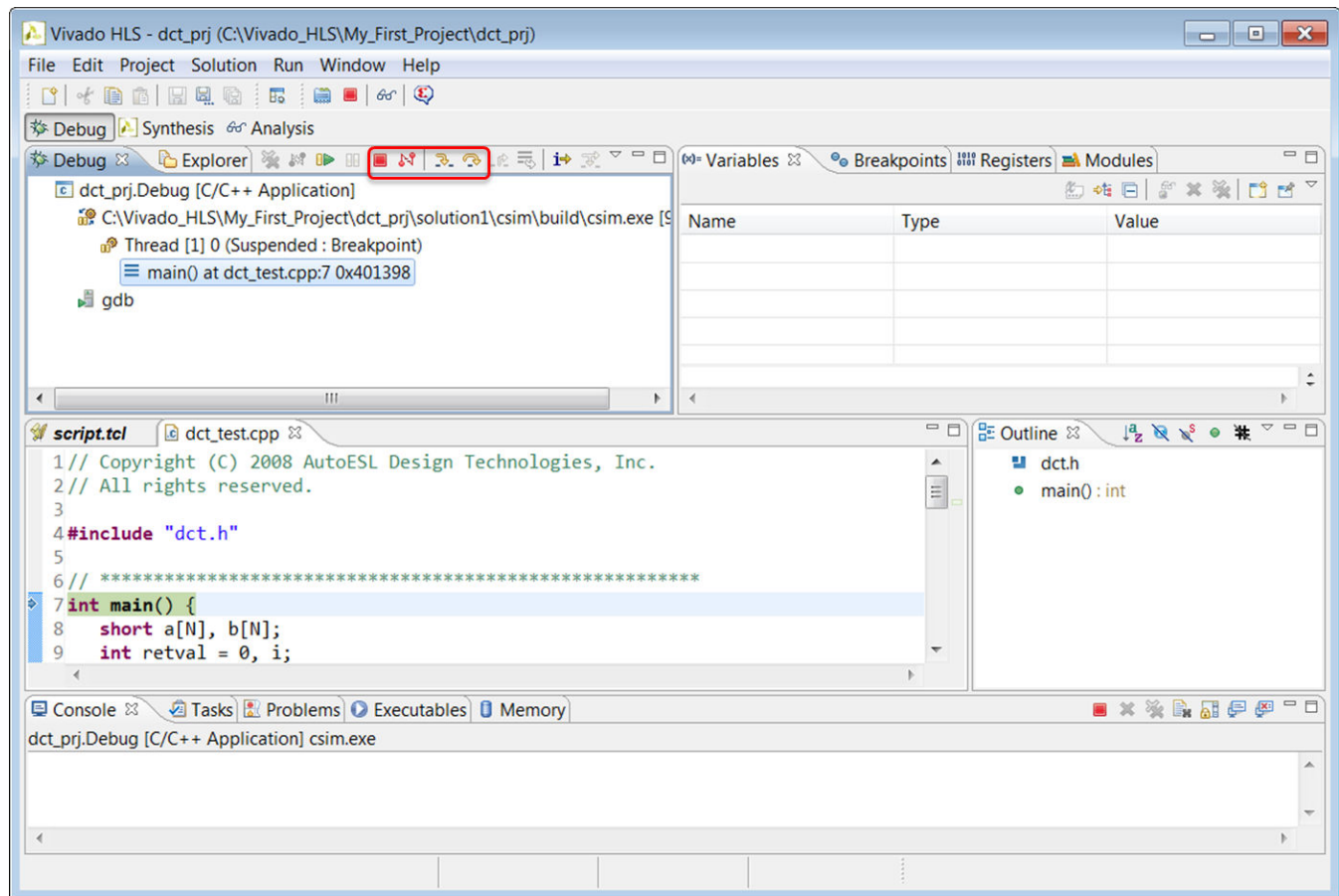


[C Simulation Dialog] ダイアログ ボックスのオプションは、次のとおりです。

- [Launch Debugger]: C コードがコンパイルされ、[Debug] パースペクティブが自動的に開きます。[Debug] パースペクティブから左上の [Synthesis] パースペクティブ ボタンをクリックすると、合成表示に戻ります。
- [Build Only]: C コードはコンパイルされますが、シミュレーションは実行されません。
- [Clean Build]: コードをコンパイルする前に既存の実行ファイルおよびオブジェクト ファイルをプロジェクトから削除します。
- [Optimized Compile]: デフォルトではデザインはデバッグ情報を使用してコンパイルされ、[Debug] パースペクティブで解析されます。このオプションでは、デザインをコンパイルする際により高いレベルの最適化エフォートが使用されますが、デバッガーで必要とされるすべての情報は削除されます。これにより、コンパイル時間は増加する可能性があります。シミュレーション実行時間は削減されます。
- [Compiler]: コードをコンパイルするのに、gcc/g++ を使用するかどうかを選択できます。

[Launch Debugger] オプションを選択すると、自動的に [Debug] パースペクティブに切り替わり、次の図のようにデバッグ環境が開きます。これは、すべての機能を備えた C デバッグ環境です。ステップ ボタン (次の図の赤で囲った部分) を使用すると、コードを 1 行ずつ移動でき、ブレークポイントを設定したり、変数の値を直接表示したりできます。

図 17: C デバッグ環境

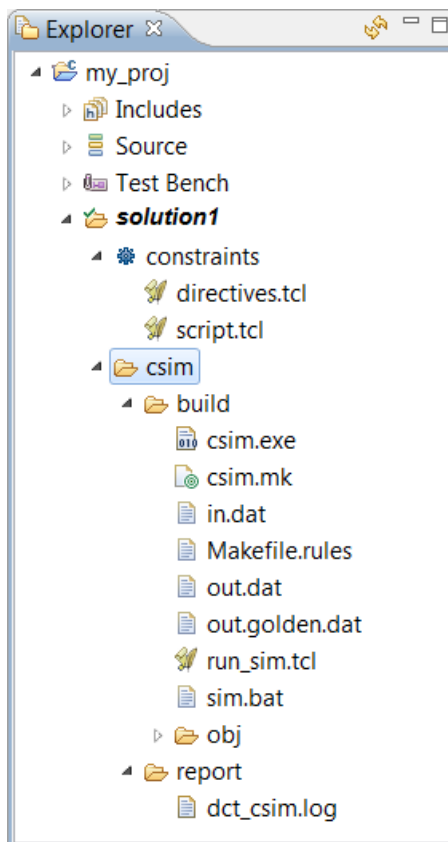


ヒント: [Synthesis] パースペクティブ ボタンをクリックして、標準の合成ウィンドウに戻ります。

C シミュレーション出力の確認

C シミュレーションが終了したら、次に示すように solution フォルダ内に csim フォルダが作成されます。

図 18: C シミュレーションの出力ファイル



`csim/build` フォルダは、C シミュレーションに関するすべてのファイルの主なディレクトリです。

- テストベンチで読み込まれるすべてのファイルがこのフォルダにコピーされます。
- このファイルに C 実行ファイルの `csim.exe` が作成され、実行されます。
- テストベンチにより出力されるすべてのファイルがこのフォルダに作成されます。

[C Simulation] ダイアログ ボックスで [Build Only] をオンにすると、このフォルダ内に `csim.exe` ファイルが作成されますが、ファイルは実行されません。コマンド シェルからこのファイルを実行すると、C シミュレーションが手動で実行できます。Windows の場合は、Vivado® HLS コマンド シェルを [スタート] メニューから使用します。

`csim/report` フォルダには、C シミュレーションのログ ファイルが含まれます。

この後の Vivado HLS デザイン フローの手順は、合成の実行です。


C コードの合成

このセクションでは、次の点について説明します。

- 最初のソリューションの作成。
- 合成の出力確認。
- 合成結果の解析。
- 新規ソリューションの作成。

- 最適化指示の適用。

最初のソリューションの作成

ツールバー ボタンの [C Synthesis]  または [Solution > Run C Synthesis] をクリックし、デザインを RTL インプリメンテーションに合成します。合成プロセス中は、メッセージが [Console] ビューに表示されます。

メッセージには、次のような合成プロセスの進捗状況が示されます。

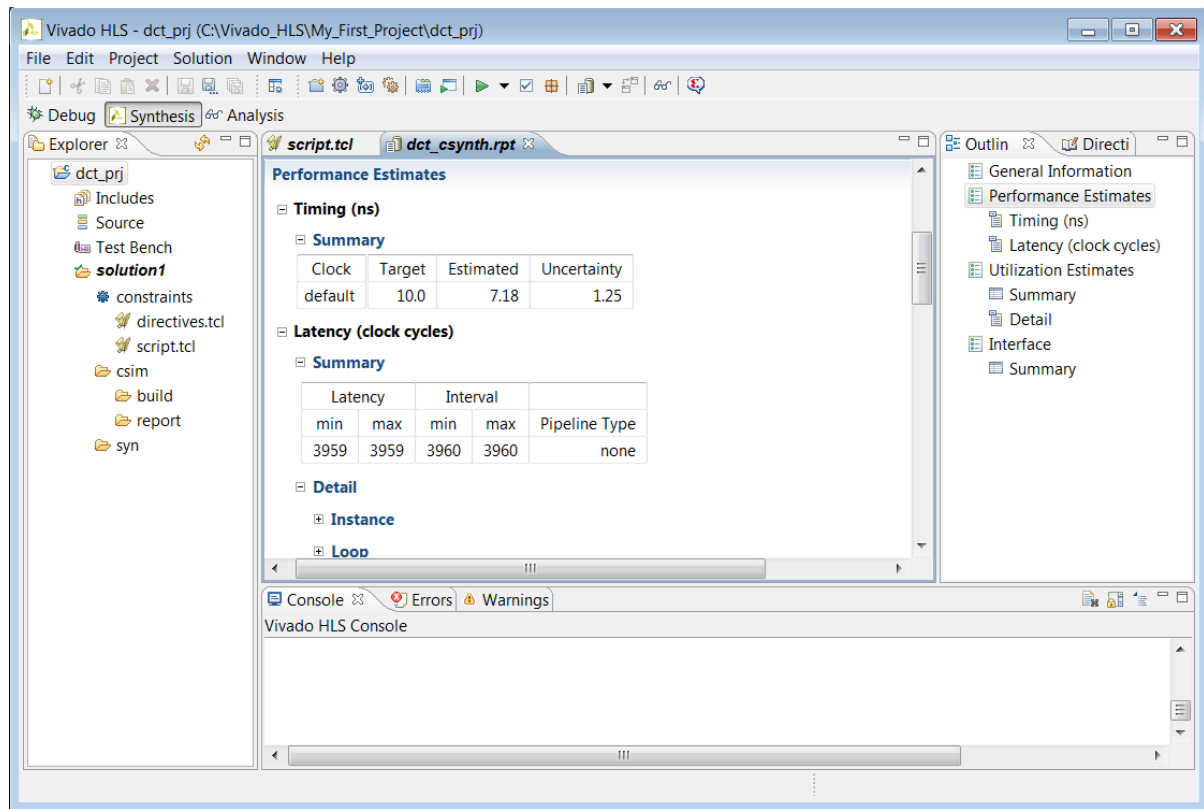
```
INFO: [HLS 200-10] Opening and resetting project
'C:/Vivado_HLS/My_First_Project/proj_dct'.
INFO: [HLS 200-10] Adding design file 'dct.cpp' to the project
INFO: [HLS 200-10] Adding test bench file 'dct_test.cpp' to the project
INFO: [HLS 200-10] Adding test bench file 'in.dat' to the project
INFO: [HLS 200-10] Adding test bench file 'out.golden.dat' to the project
INFO: [HLS 200-10] Opening and resetting solution
'C:/Vivado_HLS/My_First_Project/proj_dct/solution1'.
INFO: [HLS 200-10] Cleaning up the solution database.
INFO: [HLS 200-10] Setting target device to 'xc7k160tfbg484-1'
INFO: [SYN 201-201] Setting up clock 'default' with a period of 4ns.
```

GUI では、メッセージにリンクが付いていて、さらに情報が表示されることもあります。次の例の場合、メッセージ XFORM 203-602 に下線が付いていて、リンクが付いているのがわかります。このメッセージをクリックすると、そのメッセージが表示される理由と回避策が表示されます。この場合、Vivado® HLS では自動的に小型の関数がインライン展開されるので、INLINE 指示子を `-off` オプションを指定して使用すると、自動的なインライン展開が実行されないようにできます。

```
INFO: [XFORM 203-602] Inlining function 'read_data' into 'dct' (dct.cpp:85)
automatically.
INFO: [XFORM 203-602] Inlining function 'write_data' into 'dct'
(dct.cpp:90) automatically.
```

合成が終了したら、次の図に示すように、最上位関数の合成レポートが情報エリアに自動的に開きます。

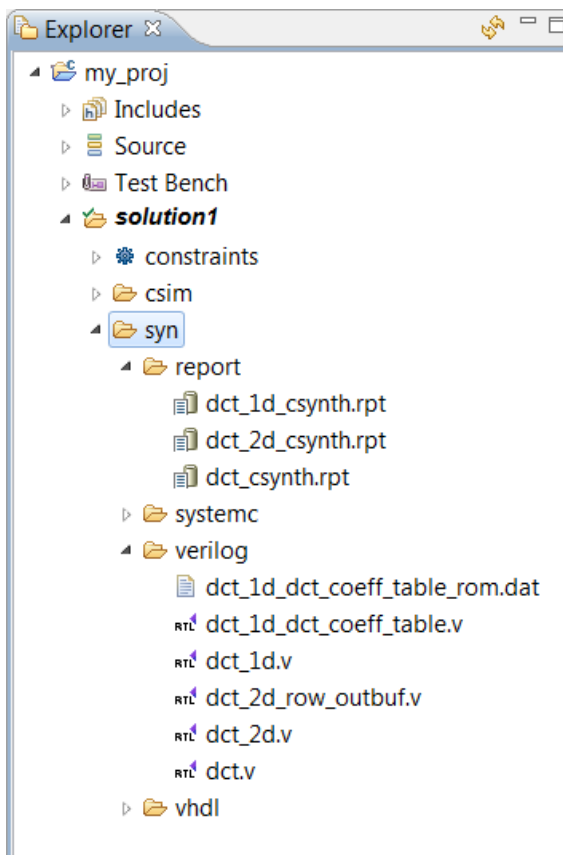
図 19: 合成レポート



合成の出力確認

合成が終了すると、solution フォルダ内に `syn` フォルダが作成されます。

図 20: C 合成の出力ファイル



syn フォルダーには、4 つのサブフォルダーが含まれます。RTL 出力形式ごとに report フォルダーと 1 つのフォルダーが含まれます。

Vivado® HLS では、関数が `INLINE` 指示子を使用してインライン展開されていなければ、最上位関数に対して 1 つ、デザインに含まれる各サブ関数に対してそれぞれ 1 つずつレポート ファイルが report フォルダーに作成されるようになっています。最上位関数のレポートには、デザイン全体の詳細が含まれます。

verilog、vhdl、および systemc フォルダーには出力 RTL ファイルが含まれます。前の図は、展開された verilog フォルダーを示しています。最上位ファイル名は、合成の最上位関数と同じ名前になります。C デザインには、関数ごとに 1 つの RTL ファイルがあります (インライン展開されていない場合)。サブブロック (ブロック RAM、パイプライン処理された乗算器など) をインプリメントするためのその他の RTL ファイルがある場合もあります。



重要: ザイリンクス では、これらのファイルを RTL 合成に使用することはお勧めしていません。ザイリンクス では、その代わりに、このデザイン フローの後半で説明するパッケージ済み IP 出力ファイルを使用することを勧めしています。この注記の後の説明を参照してください。

浮動小数点のデザインなど、Vivado HLS でザイリンクス IP が使用される場合、RTL ディレクトリには RTL 合成中に IP を作成するためのスクリプトが含まれます。syn フォルダーのファイルを RTL 合成で使用する場合は、これらのフォルダー内にあるスクリプト ファイルを正しく使用するの、ユーザーの責任になります。パッケージされた IP が使用される場合、このプロセスはザイリンクス ツールにより自動的に実行されます。

C 合成結果の解析

RTL デザインを解析するために、次の 2 つの機能が提供されています。

- 合成レポート
- [Analysis] パースペクティブ

また、RTL 環境を使用する場合のために、Vivado® HLS では IP パッケージ プロセス中に 2 つのプロジェクトが作成されます。

- Vivado Design Suite プロジェクト
- Vivado IP インテグレーター プロジェクト

合成レポート

合成が終了したら、最上位関数の合成レポートが情報エリアに自動的に開きます。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。右側の [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。

次の表に、合成レポートのカテゴリを説明します。

表 1: 合成レポートのカテゴリ

カテゴリ	説明
General Information	結果が生成されるタイミング、使用されたソフトウェア バージョン、プロジェクト名、ソリューション名、デバイスの詳細です。
Performance Estimates → Timing	ターゲット クロック周波数、クロックのばらつき、達成可能な最大クロック周波数の見積もりです。
Performance Estimates → Latency → Summary	このブロックとこのブロックにインスタンス化されたサブブロックのレイテンシと開始間隔 (II) がレポートされます。 C ソースのこのレベルで呼び出される各サブ関数は、インライン展開されていなければ、それぞれ RTL ブロックのインスタンスになります。 レイテンシは、出力値を生成するのにかかるクロック サイクル数です。開始間隔は、新しい入力が適用されるまでのクロック サイクル数です。 PIPELINE 指示子がない場合、レイテンシは開始間隔 (II) よりも 1 サイクル短くなります (最後の出力が書き込まれたときに次の入力を読み出される)。
Performance Estimates → Latency → Detail	このブロック内のインスタンス (サブ関数) およびループのレイテンシと開始間隔 (II) についてのレポートです。ループにサブループが含まれる場合は、ループ階層が表示されます。 最小および最大レイテンシの値は、ループのすべての反復を実行するためのレイテンシを示します。コードに条件分岐があると、最小値と最大値が異なるものになることがあります。 「Iteration Latency」は、ループの 1 反復のレイテンシです。 ループのレイテンシが可変で決定できない場合は、クエスチョン マーク (?) が表示されます。この表の後の説明を参照してください。 指定したターゲット開始間隔は、達成された実際の開始間隔の横に表示されます。 tripcount は、ループの反復の総数を表示します。
Utilization 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) がレポートされます。

表 1: 合成レポートのカテゴリ (続き)

カテゴリ	説明
Utilization Estimates → Details → Instance	ここにリストされるリソースは、この階層レベルにインスタンス化されたサブブロックに使用されるものです。 デザインに RTL 階層がない場合は、インスタンスはレポートされません。 インスタンスがある場合は、そのインスタンスの名前をクリックすると、そのインスタンスの合成レポートが開きます。
Utilization Estimates → 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がリストされます。 Vivado HLS では、メモリの 1 つのバンクが使用される場合はシングルポート BRAM が、2 つのバンクが使用される場合はデュアルポート BRAM がレポートされます。
Utilization Estimates → Details → FIFO	ここにリストされるリソースは、この階層レベルに FIFO をインプリメンテーションするために使用されたものです。
Utilization Estimates → Details → Shift Register	ザイリンクス SRL コンポーネントにマップされたすべてのシフトレジスタのサマリを示します。 SRL コンポーネントへの追加マップは RTL 合成中に発生します。
Utilization Estimates → Details → Expressions	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に使用されるリソースを示します。 演算に対する入力ポートのビット幅が表示されます。
Utilization Estimates → Details → Multiplexors	ここにリストされるリソースは、この階層レベルのマルチプレクサーをインプリメントするために使用されるものです。 マルチプレクサーの入力幅が表示されます。
Utilization Estimates → Details → Register	この階層レベルのレジスタすべてのリストが表示されます。このレポートには、レジスタのビット幅が含まれます。
Interface Summary → Interface	関数引数がどのように RTL ポートに合成されるかが示されます。 RTL ポート名は、プロトコルとソース オブジェクトでグループ分けされます。これらは、ソース オブジェクトが記述された IO プロトコルで合成されると作成される RTL ポートです。

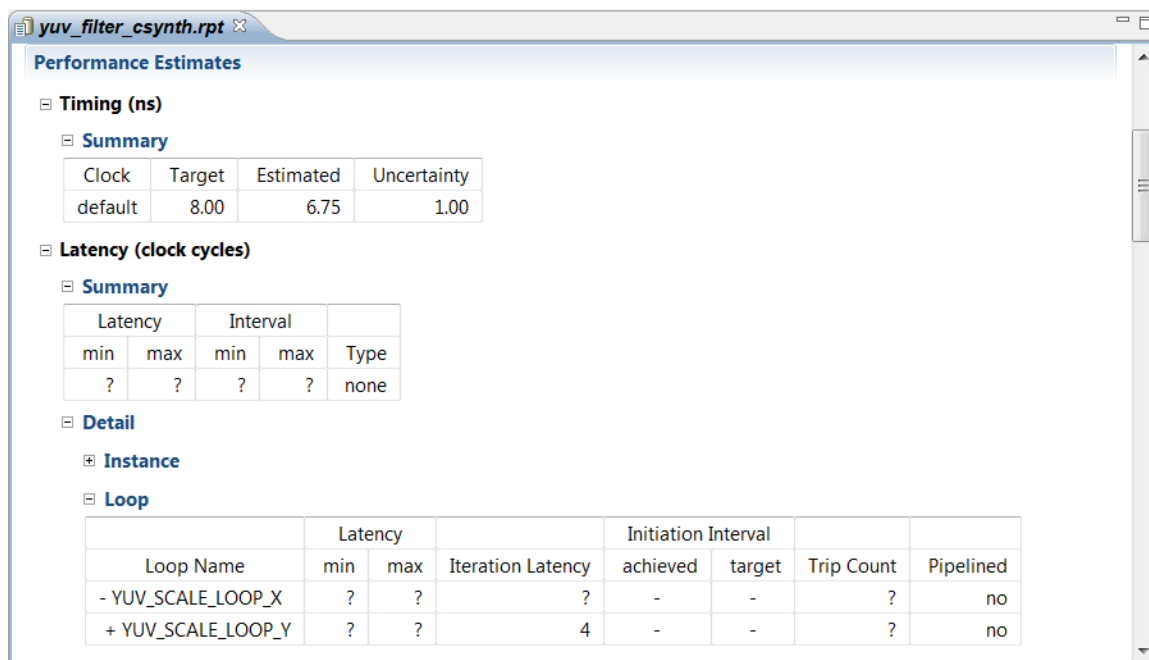
一部のザイリンクス デバイスには、スタックド シリコン インターフェイス (SSI) テクノロジが使用されています。これらのデバイスでは、使用可能なリソースが複数の SLR (Super Logic Region) に分割されます。ターゲット テクノロジとして SSI テクノロジ デバイスを選択する場合、リソース使用率レポートには、SLR の使用率とデバイス全体の使用率の両方が示されます。



重要: SSI テクノロジ デバイスを使用する場合、Vivado HLS で作成されたロジックを 1 つの SLR 内にフィットさせることが重要です。

Vivado HLS を初めて使用する場合、次の図のような合成レポートが表示されることがあります。レイテンシの値はすべてクエスチョン マーク (?) で表示されています。

図 21: 合成レポート



The screenshot shows the 'Performance Estimates' window in Vivado HLS. It contains two main sections: 'Timing (ns)' and 'Latency (clock cycles)'. The 'Timing (ns)' section has a 'Summary' table with columns: Clock, Target, Estimated, and Uncertainty. The 'Latency (clock cycles)' section has a 'Summary' table with columns: Latency (min, max), Interval (min, max), and Type. Below this is a 'Detail' section with an 'Instance' table and a 'Loop' table. The 'Loop' table has columns: Loop Name, Latency (min, max), Iteration Latency, Initiation Interval (achieved, target), Trip Count, and Pipelined.

Clock	Target	Estimated	Uncertainty
default	8.00	6.75	1.00

Latency		Interval		Type
min	max	min	max	
?	?	?	?	none

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- YUV_SCALE_LOOP_X	?	?	?	-	-	?	no
+ YUV_SCALE_LOOP_Y	?	?	4	-	-	?	no

Vivado HLS では、各ループの反復回数を決定する解析が実行されます。ループ反復制限が可変の場合、Vivado HLS では最大上限が決定できません。

次の例の場合、for ループの最大反復数が `num_samples` 入力の値で指定できます。`num_samples` の値は C 関数で定義されず、外部から関数に読み込まれます。

```
void foo (char 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可変インデックス付きのループによって異なる場合、Vivado HLS ではそのループのレイテンシが不明の状態 (? マーク) として表示されます。

TRIPCOUNT 指示子をループに適用すると、ループ反復数を手動で指定でき、有益な数値がレポートに含まれるようになります。Vivado HLS では、`-max` オプションでループが繰り返される反復の最大数を、`-min` オプションで最小数を指定します。

注記: TRIPCOUNT 指示子は、合成結果には影響しません。

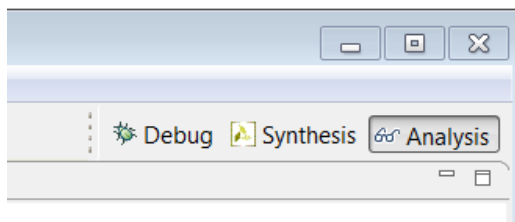
TRIPCOUNT の値はレポート目的のみに使用され、Vivado HLS で生成されるレポートに意味のあるレイテンシおよび間隔の範囲が示されるようになります。これにより、異なるソリューション間で意味のある比較ができるようになります。

C の assert マクロをコード内で使用すると、Vivado HLS でループの制限を指定できるほか、これらの制限と同じサイズのハードウェアを作成できます。

[Analysis] パースペクティブ

レポート結果の解析には、合成レポートだけでなく、[Analysis] パースペクティブを使用することもできます。
[Analysis] パースペクティブを開くには、次の図に示すように [Analysis] ボタンをクリックします。

図 22: [Analysis] パースペクティブ



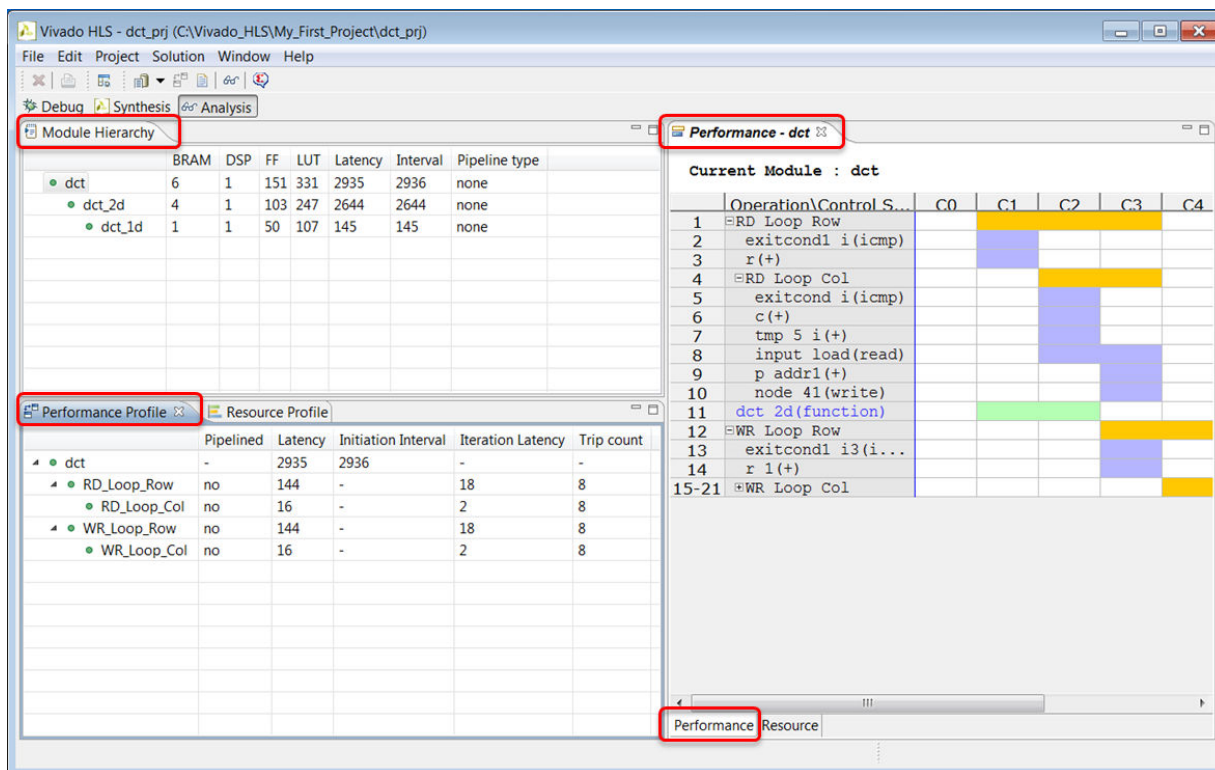
[Analysis] パースペクティブには、表形式と画像形式の両方でデザインのパフォーマンスおよびリソースが表示され、どちらの表示も連動するようになっています。次の図は、[Analysis] パースペクティブを最初に開いたときのデフォルトのウィンドウ設定です。

[Module Hierarchy] ビューには、RTL デザイン全体の概要が表示されます。

- このビューは、デザイン階層をナビゲーションするために使用します。
- [Module Hierarchy] タブには、RTL 階層の各ブロックのリソースおよびレイテンシが表示されます。

この dct デザインでは、ブロック RAM ボックスの 6 つと約 300 個の LUT が使用され、約 3000 クロックサイクルのレイテンシがあります。サブブロック dct_2b は、4 つのブロック RAM (約 250 個の LUT およびレイテンシ約 2600 サイクル) に接続されます。このデザインのほとんどのリソースとレイテンシはサブブロック dct_2b によるもので、このブロックを最初に解析する必要があることは明らかです。

図 23: Vivado HLS の [Analysis] パースペクティブ

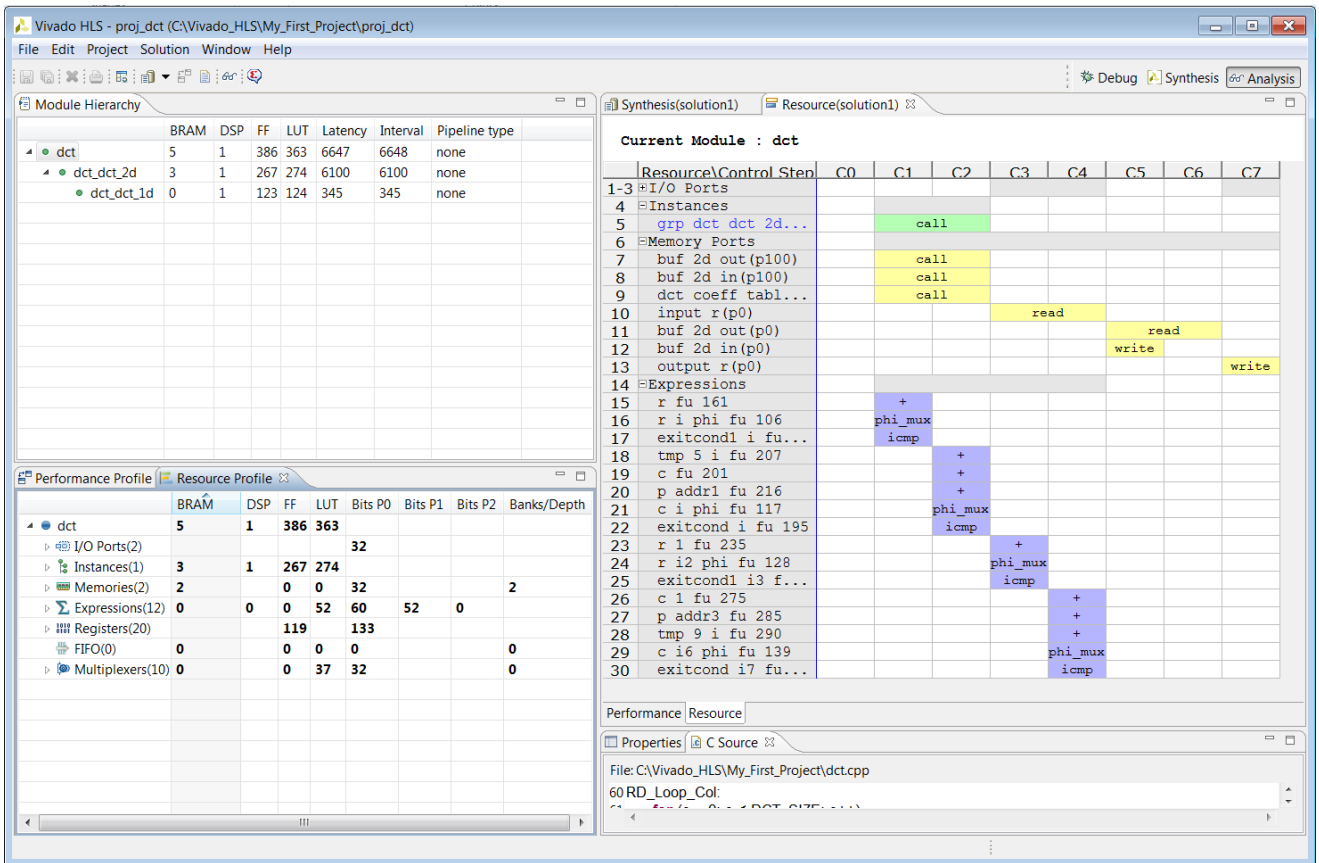


[Performance Profile] ビューには、[Module Hierarchy] ビューで現在選択されているブロック (この場合、dct ブロック) のパフォーマンスの詳細が表示されます。

- ブロックのパフォーマンスは、含まれるサブブロックの関数とこの階層レベルに含まれるロジックによります。[Performance Profile] ビューには、パフォーマンス全体に貢献するこの階層レベルのアイテムが表示されます。
- パフォーマンスは、レイテンシと開始間隔 (II) によって測定されます。このビューには、ブロックのパipeline 処理の有無も表示されます。
- この例では、2 つのループ (RD_Loop_Row および WR_Loop_Row) がこの階層レベルにロジックとしてインプリメントされ、どちらにもサブループが含まれ、どちらもレイテンシは 144 クロック サイクルです。両方のループのレイテンシを同様に dct に含まれる dct_2d のレイテンシに追加すると、dct ブロックのレイテンシの合計が得られます。

[Analysis] パースペクティブでは、リソース使用量も解析できます。次の図は、[Resource Profile] および [Resource] ビューを示しています。

図 24: [Resource Profile] を表示する [Analysis] パースペクティブ



[Resource Profile] ビューには、この階層レベルで使用されたリソースが表示されます。この例の場合、ほとんどのリソースがインスタンス (このブロック内にインスタンス化されたブロック) によるものです。

[Expressions] を展開すると、この階層レベルのほとんどのリソースが加算器をインプリメントするために使用されているのがわかります。

[Resource] タブには、使用された演算の制御ステートが示されます。この例の場合、すべての加算器演算が別の加算器リソースに関連付けられています。加算器の共有はありません。横方向のそれぞれの行に加算演算が複数ある場合は、同じリソースが別のステートまたはクロック サイクルで複数回使用されていることを示します。

加算器は、メモリ アクセスされる同じサイクルで使用され、各メモリごとに別のものが使用されています。これは、関連する C コードをクリックすると確認できます。

スケジュール ビューアー

スケジュール ビューアーには、合成済み RTL が詳細に表示され、並列処理、タイミング違反、データ依存を阻むループ依存がないかどうかを確認できます。

- このビューアーには、右側の [Analysis] パースペクティブからアクセスできます。
- [Module Hierarchy] ウィンドウでモジュールを右クリックして [Open Schedule Viewer] を選択すると、各ブロックのスケジューリングを表示できます。[Module Hierarchy] ウィンドウには、開始間隔 (II) またはタイミング違反が直接示されます。タイミング違反がある場合は、特定モジュールの負のスラック合計も示されます。


注記: このウィンドウのメニュー ボタンを使用すると、II またはタイミング違反を示すブロックをフィルターして表示できます。

スケジュール ビューアーのメイン ウィンドウは、次のようになっています。

- 縦軸は演算およびループの名前を示します。
- 演算はトポロジ順に表示されるので、n 行目の演算は前の行の演算からのみ駆動され、後の行の演算の入力のみを駆動します。

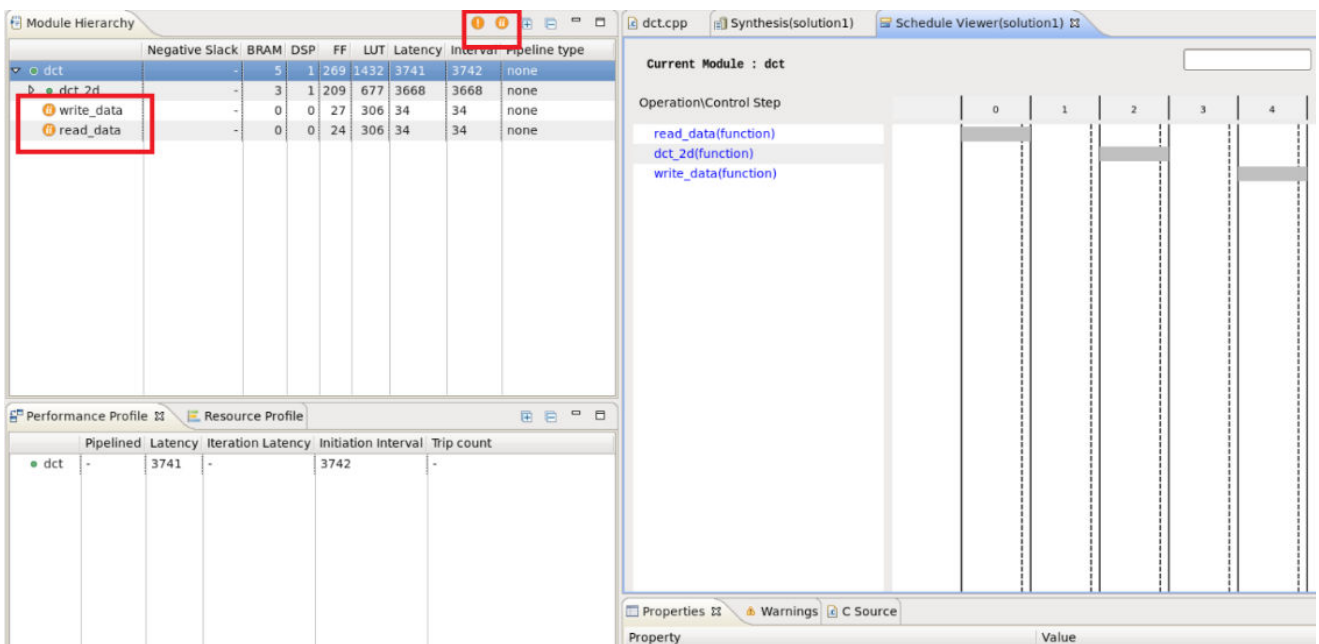
次の例の場合、最上位関数のみが次の順序で表示されています。

- read_data
- dct_2d
- write_data

- 横軸のグレーのバーは、サイクルを順序どおりに表示します。
- 縦の点線は、クロックのばらつき用のクロック周期の潜在的な予約部分を示します。この時間は、配置配線のよ
うな Vivado バックエンド プロセス用にツールで制御されます。
- 演算ごとに、グレーのボックスが表示されます。通常このボックスは、クロック サイクル合計の割合と同様、
演算の遅延に従って横方向にサイズ指定されます。この例のように、関数呼び出しの場合、提供されるサイクル
情報は op レイテンシと同じになります。この場合、read_data 関数の op レイテンシは 1 です。
- 複数サイクル演算の場合、op ボックスを横切る棒線が表示されます。表示されるさまざまな要素はすべて、スケ
ジュール ビューアー メニューの右上の [Legend] ボタン  をクリックするとリストされます。
- 最も重要なことに、ソース ロケーションはどの演算とも関連付けられます。演算をダブルクリックすると、入力
ソース コードでその演算のソースがハイライトされます。

関数呼び出しの場合は、提供されるサイクル情報は op レイテンシです。この場合、次の図の [Properties] タブに示
すように、read_data の op レイテンシは 1 です。

図 25: スケジュール ビューアー



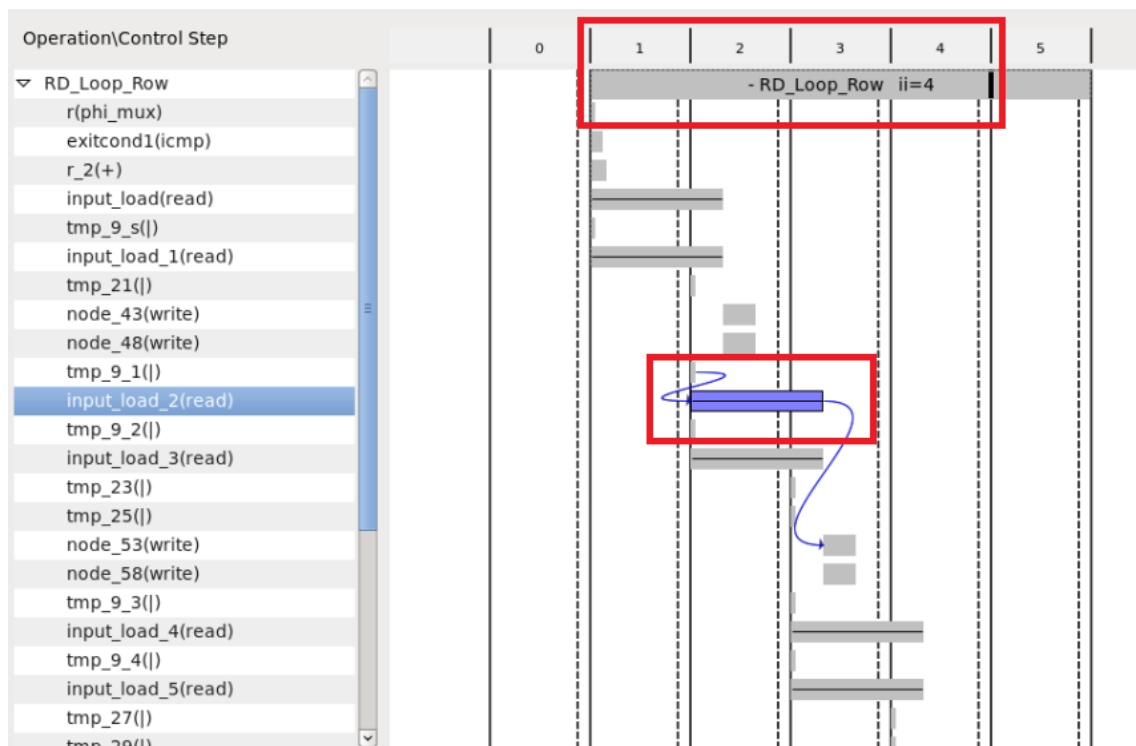
- [Module Hierarchy] ウィンドウで read_data 階層をクリックし、RD_Loop_Row というループを確認します。
 - 。これはパイプライン処理されたループで、開始間隔 (II) はループ バー内にはっきりと示されます。スケジューラ ビューアーでは、1 つの反復全体を表示できるように、パイプライン処理されたループは展開が閉じた状態で表示されます。オーバーラップがわかりやすいように、II はループ マーカー内に太いクロック境界線で表示されます。
 - 。1 つの反復のレイテンシ合計は、ループ マーカーの示すクロック数と同じになります。この場合、5 サイクル (1-5) です。
- タイミング違反

次の図には、タイミング違反があります。タイミング違反ビューは、[Module Hierarchy] ウィンドウでモジュールを選択して右クリックするか、スケジューラ ビューアー メニューのプルダウン メニューを使用するとナビゲートできます。

タイミング違反とは、使用可能なクロック サイクルよりも時間を必要とする演算のパスのことです。こういった問題のサイクルを見やすくするため、問題のあったサイクルは延長されて表示されるので、実際のサイクル バウンダリが移動されて、同じサイクルに属するすべてが不透明なボックスで表示されるようになります。

デフォルトでは、すべての依存 (青い線) がクリティカル タイミング パスの各演算間に表示されます。

図 26: 違反の原因となる演算

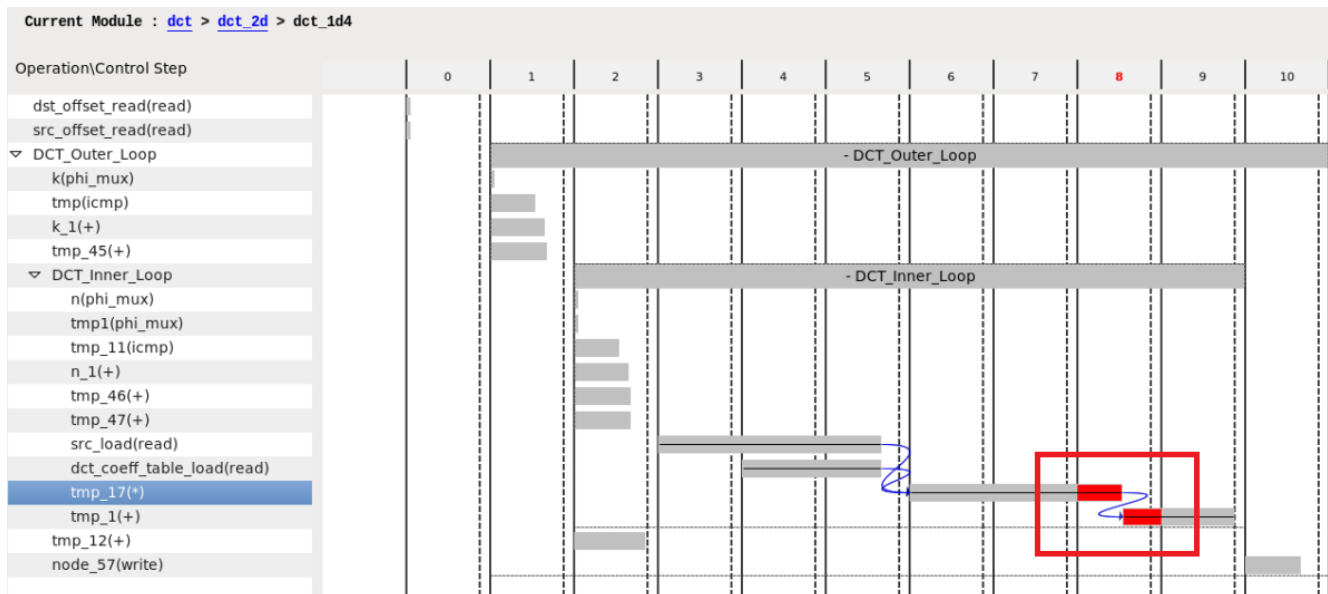



- このビューアーには、標準的な演算子の依存が表示できます。演算を選択すると、特定の演算子の依存が青い矢印で表示されます。これにより、データ依存の詳細な解析ができます。
- II 違反を解析することもできます。モジュールにこのような違反が含まれる場合は、モジュール階層のコンテキストメニューまたはビューアーのドロップダウンから違反を表示できます。

次の図には、II 全体を生成するパスがあり、値を次の反復が開始する前に計算して、パスを短くして II を削減する必要があることを意味します。

ソース コードで演算を見つけるには、その演算をダブルクリックします。ソース ビューアーが表示され、ソース 内のオブジェクトのルートを確認できるようになります。

図 27: タイミング違反



- スケジュール ビューアーのメニュー バーでフィルター ボタン  をクリックすると、スケジュール ビューアーに 表示する演算をダイナミックにフィルターできます。演算は、タイプ別またはクラスター別にフィルターできます。
 - タイプでフィルターすると、機能に基づいて表示される演算を制限できます。たとえば、加算、乗算、および関数呼び出しのみを表示すると、AND および OR などの小さな演算は表示されなくなります。
 - クラスターでフィルターすると、スケジューラで基本演算をグループ分けしてから、それらを 1 つのコンポーネントとしてスケジューリングできます。クラスター フィルター設定を使用すると、クラスターを色表示したり、展開を閉じて 1 つの大きな演算として表示したりすることもできます。これで、スケジュールがより簡易的に表示できます。

データフロー ビューアー

DATAFLOW 指示子が関数に適用されている場合、[Analysis] パースペクティブにデザインの構造を示すデータフロー ビューアーが表示されます。このビューには、さまざまなプロセスおよびそれに関連するプロデューサーとコンシューマー関係を示すデータフローがグラフで表示されます。


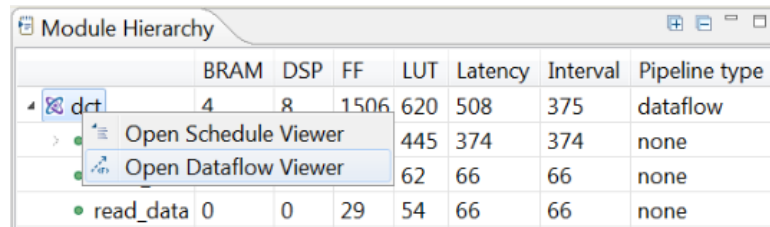
次の図に示す [dct] 関数の横の  アイコンは、データフロー ビューがあることを意味します。関数を右クリックして、データフロー ビューを開きます。

図 28: データフロー ビューを開く



	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	4	8	1506	620	508	375	dataflow
Open Schedule Viewer				445	374	374	none
Open Dataflow Viewer				62	66	66	none
read_data	0	0	29	54	66	66	none

[Analysis] パースペクティブは、非常にインタラクティブです。[Analysis] パースペクティブの詳細は、『Vivado Design Suite チュートリアル: 高位合成』(UG871) の「デザイン解析」セクションを参照してください。



ヒント: Tcl フローがデザインを作成するのに使用されていても、プロジェクトは GUI で開いて、[Analysis] ビューでデザインを解析できます。

[Synthesis] パースペクティブ ボタンをクリックして、合成表示に戻ります。

通常は、デザイン解析後に新しいソリューションを作成して、最適化指示子を適用できます。これに新しいソリューションを使用すると、さまざまなソリューションを比較できます。

新規ソリューションの作成

Vivado HLS の最も一般的な使用法は、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、初期の合成実行を保持したまま、比較できます。


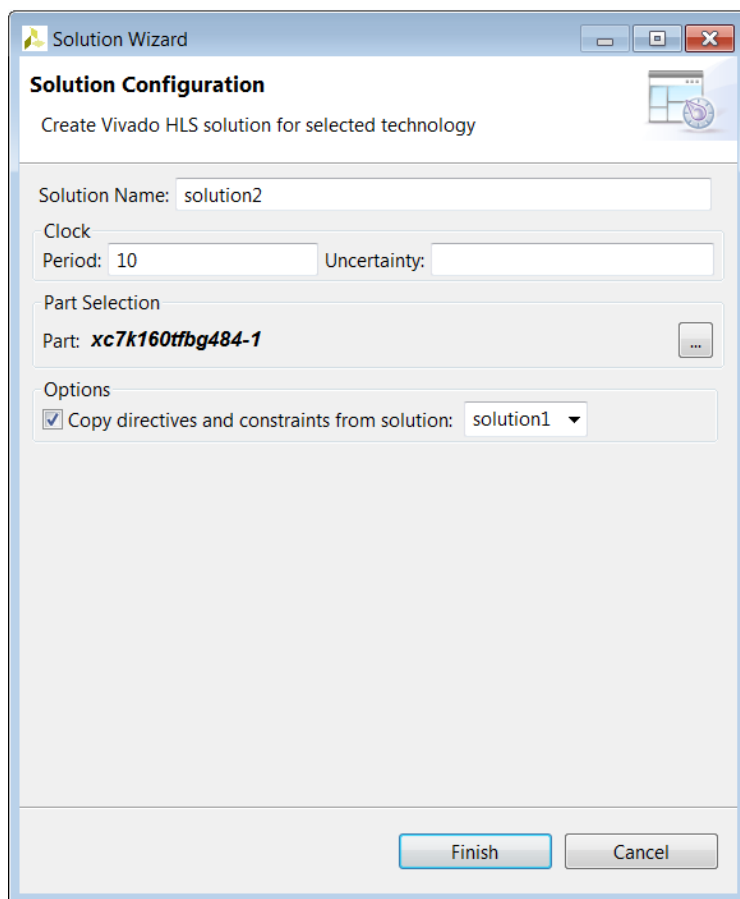
新しいソリューションを作成するには、[New Solution] ツールバー ボタン  をクリックするか、[Project > New Solution] をクリックします。これにより、次の図のようなソリューション ウィザードが開きます。

図 29: ソリューション ウィザード



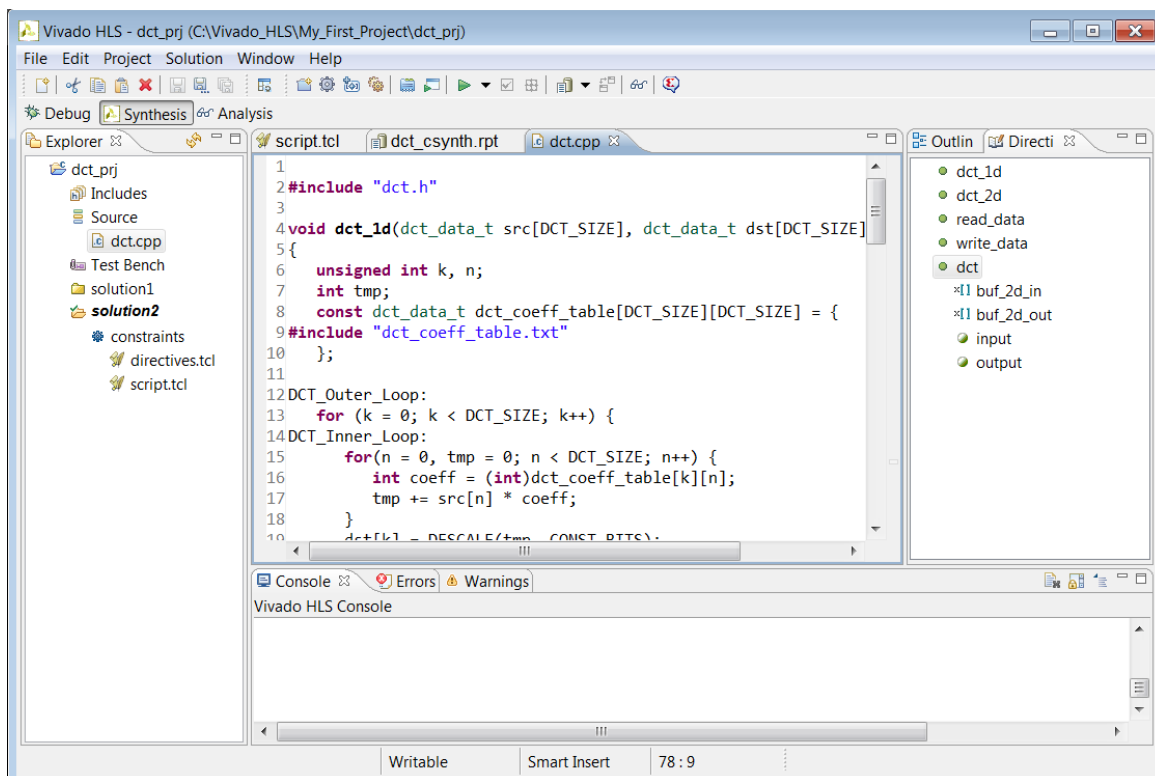
ソリューション ウィザードには、New Project ウィザードの最後のページと同じオプションに加えて、既存ソリューションに適用される指示子やカスタム制約をコピーするオプションがあります。コピーしたソリューションは、編集または削除もできます。

新しいソリューションが作成されたら、最適化指示子を追加できます (前のソリューションからコピーした場合は変更できます)。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。カスタム制約は設定オプションを使用して適用できます。詳細は、[デザインの最適化](#)を参照してください。

最適化指示子の適用

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。次の図に示すように、[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に開きます。

図 30: ソースおよび指示子



情報エリアでソースコードを選択して、右側の [Directives] タブをクリックして、ファイルの指示子を表示したり変更したりします。[Directive] タブには、開いているソースコードで指示子を適用できるオブジェクトおよびスコープがすべて表示されます。

注記: ほかの C ファイルでオブジェクトに指示子を適用するには、そのファイルを開いて情報エリアでそれをアクティブにする必要があります。

オブジェクトを Vivado HLS の GUI で選択して指示子を適用することもできますが、Vivado HLS ではすべての指示子はそのオブジェクトを含むスコープに適用されます。たとえば、INTERFACE 指示子を Vivado HLS の GUI でインターフェイスオブジェクトに適用できます。Vivado HLS では、指示子が最上位関数(スコープ)に適用され、インターフェイスポート(オブジェクト)が指示子で識別されます。次の例の場合、foo 関数の data_in ポートが AXI4-Lite インターフェイスとして指定されます。

```
set_directive_interface -mode s_axilite "foo" adata_in
```

最適化指示子は、次のオブジェクトおよびスコープに適用できます。

- インターフェイス

指示子をインターフェイスに適用すると、インターフェイスは最上位関数に含まれるので、Vivado HLS では指示子が最上位関数に適用されます。

- 関数

指示子を関数に適用する場合、Vivado HLS でその関数のスコープ内のオブジェクトすべてに指示子が適用されます。プラグマの効果は、関数の次の階層レベルで停止します。唯一の例外は、階層のループを繰り返し展開する PIPELINE 指示子など、-recursive オプションを必要とする指示子です。

- ループ

指示子をループに適用する場合、Vivado HLS でそのループのスコープ内のオブジェクトすべてに指示子が適用されます。たとえば、LOOP_MERGE 指示子をループに適用した場合、Vivado HLS では指示子はそのループ自体ではなく、ループ内のサブループに適用されます。

注記: 指示子が適用されたループは、同じ階層レベルの同等レベルのループとは統合されません。

- 配列

指示子を配列に適用する場合、Vivado HLS で指示子はその配列を含むスコープに適用されます。

- 領域

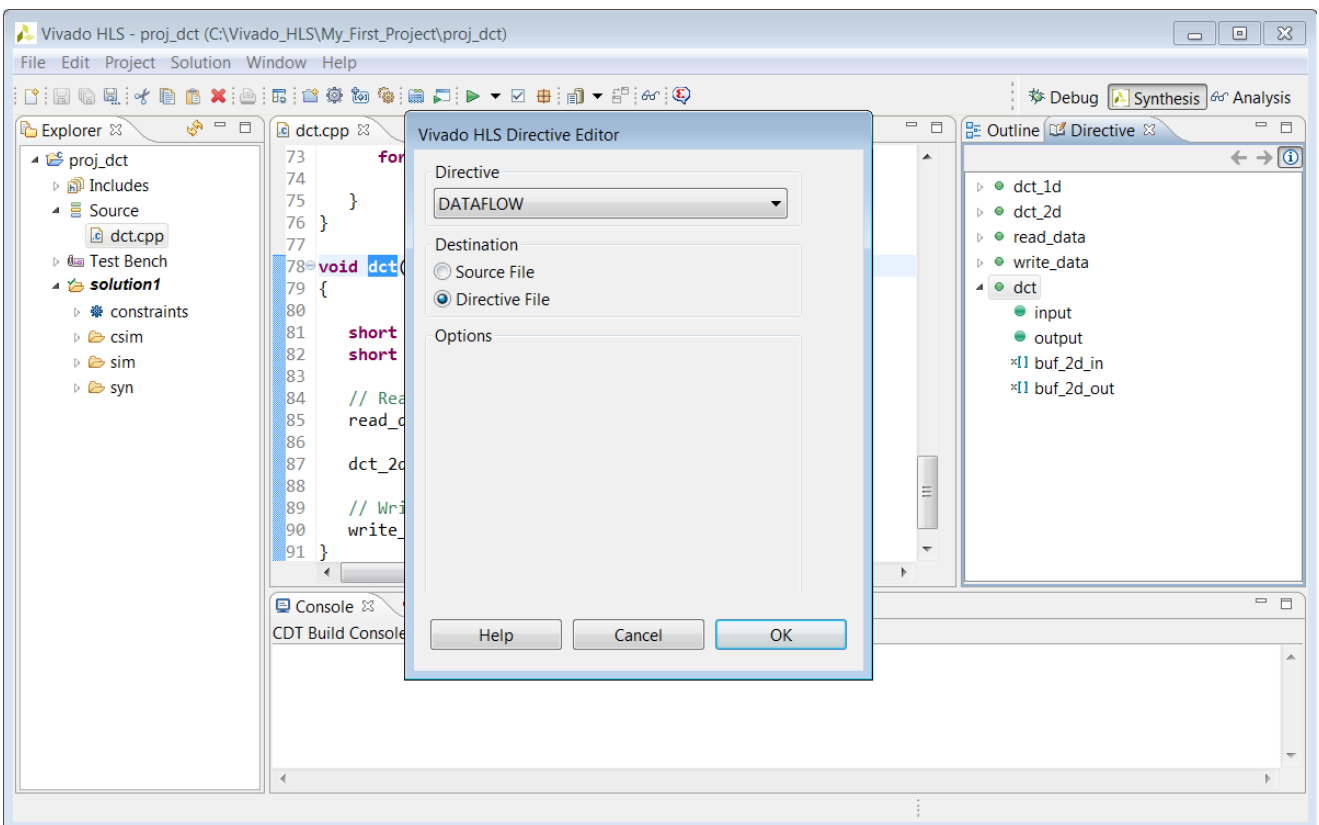
指示子を領域に適用する場合、Vivado HLS で指示子はその領域のスコープ全体に適用されます。領域は、{ } で囲まれたエリアです。次に例を示します。

```
{
  the scope between these braces is a region
}
```

注記: 指示子は、関数およびループに適用したのと同じように領域に適用できます。

指示子を適用するには、[Directives] 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 [Directive Editor] ダイアログ ボックスを開きます。ドロップダウン リストから指示子を選択します。ドロップダウン リストには、選択したオブジェクトまたはスコープに追加できる指示子のみが表示されます。たとえば、配列オブジェクトを選択した場合、ドロップダウン リストには PIPELINE 指示子は表示されません (配列はパイプライン処理できないため)。次の図に、DCT 関数に DATAFLOW 指示子を追加するところを示します。

図 31: 指示子の追加



Tcl コマンドまたはエンベデッド プラグマの使用

[Vivado HLS Directive Editor] ダイアログ ボックスの [Destination] セクションで、次のデスティネーション設定のいずれかを指定できます。

- [Directive File]: Vivado HLS は指示子を Tcl コマンドとして solution ディレクトリの directives.tcl ファイルに挿入します。
- [Source File]: Vivado HLS は指示子をプラグマとして C ソース ファイルに直接挿入します。

次の表に、これら 2 つの方法の利点と欠点を示します。

表 2: Tcl コマンド vs プラグマ

指示子の形式	利点	欠点
指示子ファイル (Tcl コマンド)	各ソリューションに独立した指示子が含まれます。デザインでさまざまな指示子を試す場合に最適です。 ソリューションを合成し直すと、そのソリューションで指定した指示子のみが適用されます。	C ソース ファイルをサードパーティに送付またはアーカイブする場合は、directives.tcl ファイルも含める必要があります。 結果を再作成するには directives.tcl ファイルが必要です。
ソース コード (プラグマ)	最適化指示子は C ソース コードに埋め込まれます。 C ソース ファイルをサードパーティに C IP として送付する場合に最適です。同じ結果を再作成するのにほかのファイルは必要ありません。 TRIPCOUNT および INTERFACE などの変更される可能性が低い指示子に便利な方法です。	最適化指示子がコードに埋め込まれていると、合成し直したときにそれらが自動的にすべてのソリューションに適用されます。

プラグマ属性の値を指定する際、リテラル値 (1、55、3.14 など) を使用するか、#define を使用してマクロを渡すことができます。次の例では、プラグマをリテラル値で指定しています。

```
#pragma HLS ARRAY_PARTITION variable = k_matrix_val  cyclic  factor=5
```

次の例では、定義されているマクロを使用しています。

```
#define E 5
#pragma HLS ARRAY_PARTITION variable = k_matrix_val  cyclic  factor=E
```



重要: プラグマの値を指定するのに、ユーザー定義のマクロはサポートされていないので使用しないでください。

プラグマ検証

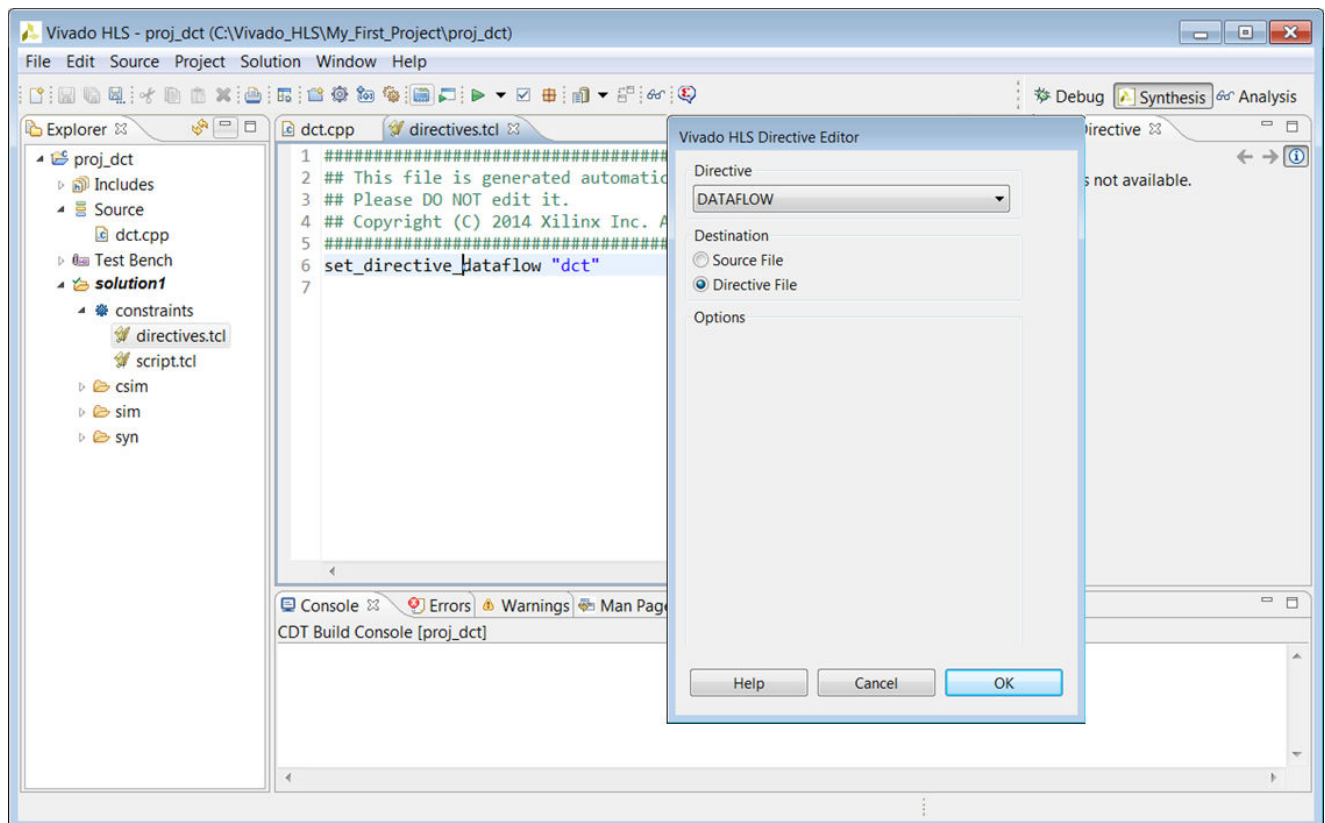
ツールは、C 合成中に変数、関数、およびループのプラグマを検証します。この検証には、プラグマの競合も含まれます。

たとえば、配列が宣言されると、ブロック RAM にデフォルトでマップされます。配列を分割または再形成することはできますが、これらは同時に使用することはできません。同じ変数で配列の分割および再形成を間違えて指定すると、次のようなエラーメッセージが表示され、合成が停止します。

```
WARNING: [XFORM 203-180] Applying partition directive (core.cpp:12:1) and
reshape
directive (core.cpp:13:1) on the same variable 'A' (core.cpp:11) may lead
to
unexpected synthesis behaviors. INFO: [XFORM 203-131] Reshaping array 'A'
(core.cpp:11) in dimension 1 completely. ERROR: [XFORM 203-103] Cannot
partition
array 'A' (core.cpp:11): variable is not an array. ERROR: [HLS 200-70] Pre-
synthesis
failed. command 'ap_source' returned error code
```

次の図に、指示子ファイルに DATAFLOW 指示子を追加するところを示します。directives.tcl ファイルはソリューションの constraints フォルダにあり、情報エリアに結果の Tcl コマンドを含めて表示されています。

図 32: Tcl 指示子の追加



指示子が Tcl コマンドとして適用される場合、その Tcl コマンドでスコープまたはそのスコープ内のスコープとオブジェクトが指定されます。ループおよび領域の場合、Tcl コマンドでこれらのスコープにラベルを付ける必要があります。ループまたは領域に現時点でラベルがない場合は、ラベルについて尋ねるダイアログボックスが表示されます。

次の例は、ループおよび領域のラベルありの場合となしの場合の例です。

```
// Example of a loop with no label
for(i=0; i<3;i++ {
    printf("This is loop WITHOUT a label \n");
}

// Example of a loop with a label
My_For_Loop:for(i=0; i<3;i++ {
    printf("This loop has the label My_For_Loop \n");
}

// Example of a region with no label
{
    printf("The scope between these braces has NO label");
}

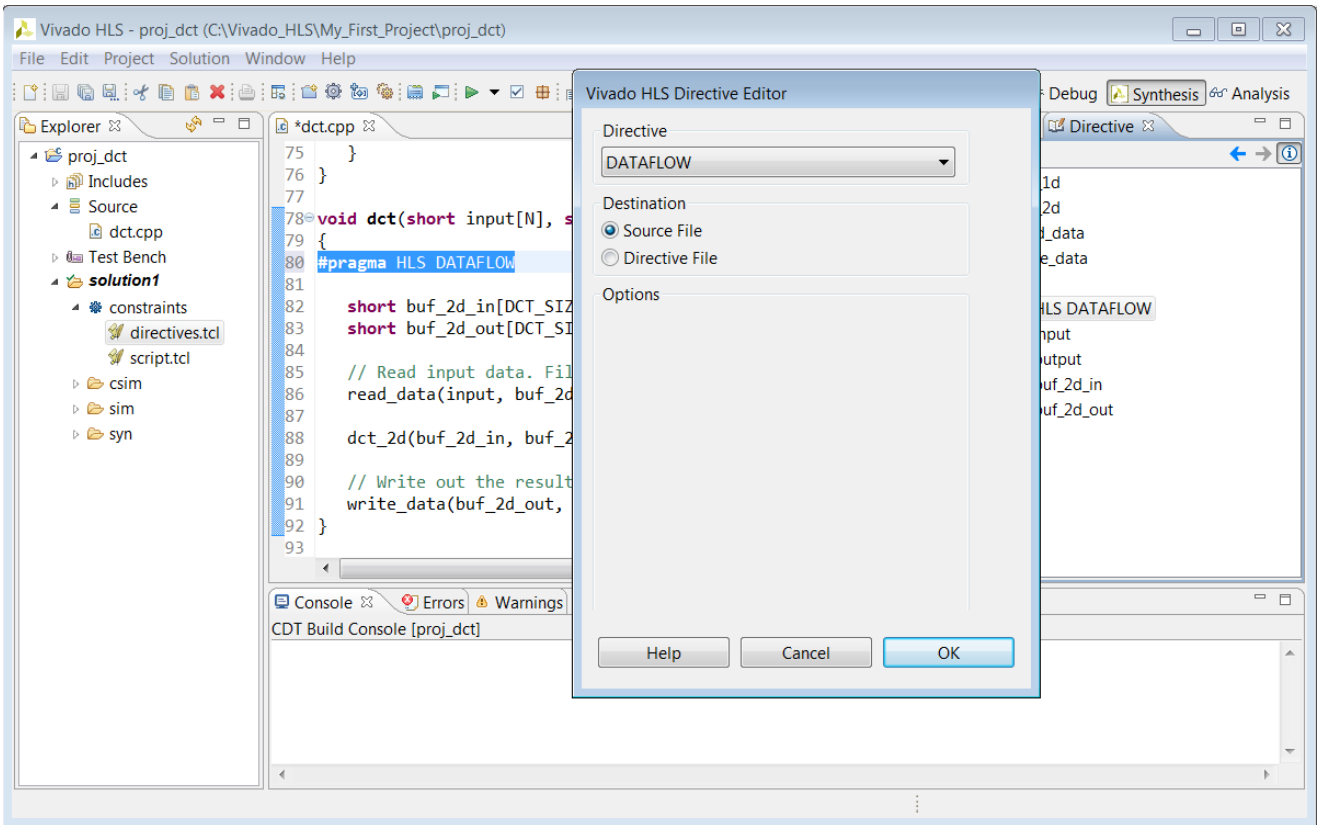
// Example of a NAMED region
My_Region:{
    printf("The scope between these braces HAS the label My_Region");
}
```



ヒント: これらのループでは、合成レポートを読みやすくなります。ラベルのないループには、ラベルが自動的に生成されます。

次の図は、指示子をソース ファイルに追加し、その結果のソース コードを情報エリアに開いたところを示しています。ソース コードには、最適化指示子を指定するプラグマが含まれるようになっています。

図 33: プラグマ指示子の追加



どちらの場合も、合成を実行すると、指示子が適用されて最適化が実行されます。ラベルまたはプラグマを挿入してコードを変更すると、合成前にコードを保存するかどうか尋ねるダイアログ ボックスが表示されます。

グローバル変数への最適化指示子の適用

指示子は、スコープまたはスコープ内のオブジェクトにのみ適用できます。関数のスコープ外部で宣言されるグローバル変数には直接適用できません。

グローバル変数に指示子を適用するには、グローバル変数が使用されるスコープ (関数、ループ、または領域) に指示子を適用します。その変数が使用されるスコープの [Directives] タブを開き、指示子を適用し、[Vivado HLS Directive Editor] ダイアログ ボックスに手動で変数名を入力します。

クラス オブジェクトへの最適化指示子の適用

最適化指示子は、クラスで定義されるオブジェクトまたはスコープにも適用できます。違いは、通常クラスはヘッダー ファイルで定義されている点です。次のいずれかの方法でヘッダー ファイルを開きます。

- [Explorer] タブで Include フォルダを開いて、ヘッダー ファイルをダブルクリックして開きます。
- C ソース内からは、ヘッダー ファイル (#include 文) にカーソルを置いて、[Ctrl] キーを押しながらそのヘッダー ファイルをクリックします。

[Directives] タブにヘッダー ファイルのオブジェクトが表示され、指示子が適用できます。



注意: ヘッダー ファイルにプリAGMAとして指示子を適用する場合は、注意が必要です。ファイルはほかのユーザーに使用されたり、ほかのプロジェクトで使用されたりする可能性があります。プリAGMAとして追加された指示子は、ヘッダー ファイルがデザインに含まれるたびに適用されます。

テンプレートへの最適化指示子の適用

Tcl コマンドを使用して最適化指示子をテンプレートに手動で適用するには、クラス メソッドを参照する際にテンプレート引数とクラスを指定します。たとえば、次のような C++ コードがあるとします。

```
template <uint32 SIZE, uint32 RATE>
void DES10<SIZE,RATE>::calcRUN() {}
```

関数に INLINE 指示子を指定するには、次の Tcl コマンドを使用します。

```
set_directive_inline DES10<SIZE,RATE>::calcRUN
```

#define とプリAGMA指示子の使用

プリAGMA指示子では、define 文で指定した値の使用はもともとサポートされていません。次のコードでは、define 文を使用してストリームの深さを指定しようとしませんが、コンパイルはされません。



ヒント: 深さの引数は明確な値で指定する必要があります。

```
#include <hls_stream.h>
using namespace hls;

#define STREAM_IN_DEPTH 8

void foo (stream<int> &InStream, stream<int> &OutStream) {

// Illegal pragma
#pragma HLS stream depth=STREAM_IN_DEPTH variable=InStream

// Legal pragma
#pragma HLS stream depth=8 variable=OutStream

}
```

#define が不必要な場合、const int などの定数を使用できます。次に例を示します。

```
const int MY_DEPTH=1024;
#pragma HLS stream variable=my_var depth=MY_DEPTH
```

C コードのマクロを使用すると、この機能をインプリメントできます。マクロを使用するには、マクロの階層レベルを使用します。これにより、拡張が正しく実行されるようになります。このコードの場合、次のようにするとコンパイルされるようになります。

```
#include <hls_stream.h>
using namespace hls;

#define PRAGMA_SUB(x) _Pragma (#x)
```

```
#define PRAGMA_HLS(x) PRAGMA_SUB(x)
#define STREAM_IN_DEPTH 8

void foo (stream<int> &InStream, stream<int> &OutStream) {

// Legal pragmas
PRAGMA_HLS(HLS stream depth=STREAM_IN_DEPTH variable=InStream)
#pragma HLS stream depth=8 variable=OutStream

}
```

最適化指示子の適用エラー

最適化指示子が適用されると、Vivado HLS では進捗状況の詳細がコンソール (およびログ ファイル) に出力されます。次は、II = 1 (開始間隔 1) の C 関数に PIPELINE 指示子が適用されたのに、合成でこの目的が達成できなかった例です。

```
INFO: [SCHED 11] Starting scheduling ...
INFO: [SCHED 61] Pipelining function 'array_RAM'.
WARNING: [SCHED 63] Unable to schedule the whole 2 cycles 'load' operation
('d_i_load', array_RAM.c:98) on array 'd_i' within the first cycle (II = 1).
WARNING: [SCHED 63] Please consider increasing the target initiation
interval of the
pipeline.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('idx_load_2',
array_RAM.c:98) on array 'idx' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 4, Depth: 6.
INFO: [SCHED 11] Finished scheduling.
```



重要: Vivado HLS では最適化指示子が達成できない場合、最適化ターゲットが自動的に緩められ、低めのパフォーマンス ターゲットでデザインの作成が試みられますが、ターゲットを緩めることができない場合は、エラーになって停止します。

低めの最適化ターゲットでデザインを作成するため、Vivado HLS から次の 3 つの重要なタイプの情報が提供可能になっています。

- 現在の C コードと最適化指示子で達成可能なターゲット パフォーマンス。
- 高めのパフォーマンス ターゲットを満たすことができなかった理由のリスト。
- さらに詳細な情報を提供してエラーの理由を理解できるようにするための解析可能なデザイン。

例の場合、SCHED-69 メッセージに、ターゲット開始間隔 (II) を達成できなかった理由がポートに制限があったためであることが記述されています。理由は、デザインはブロック RAM にアクセスする必要があり、ブロック RAM に最大 2 ポートしかないからです。

このようなエラーが発生したら、問題を解析します。この例の場合、コードの 52 行目を解析し、Analysis パースペクティブモードを使用してボトルネックを判明させて、ポート数 2 つを超える要件を減らすことができるのか、ポート数をどのように増加できるのかを決定します。

デザインが最適化されて必要なパフォーマンスが達成されたら、RTL を検証して、合成結果を IP としてパッケージできます。

RTL の検証

[C/RTL cosimulation] ツールバー ボタン () または [Solution > Run C/RTL cosimulation] をクリックして RTL 結果を検証します。

次の図に示す [C/RTL cosimulation] ダイアログ ボックスでは、検証に使用する RTL 出力タイプ (Verilog または VHDL) を選択して、シミュレーションにどの HDL シミュレータを使用するかを指定できます。

図 34: [C/RTL Co-Simulation] ダイアログ ボックス



検証が終了すると、コンソールに SIM-1000 というメッセージが表示され、検証で問題が検出されなかったことが示されます。C テストベンチの `printf` コマンドの結果がコンソールに表示されます。

```
INFO: [COSIM 316] Starting C post checking ...
Test passed !
INFO: [COSIM 1000] *** C/RTL co-simulation finished: PASS ***
```

シミュレーション レポートが情報エリアに自動的に開き、エラーなし (PASS) またはエラーあり (FAIL) と、レイテンシおよび II の測定の統計が表示されます。



重要: C/RTL 協調シミュレーションは、C テストベンチから値 0 が返された場合にのみパスになります。協調シミュレーションは、テストベンチ内のシナリオをテストして、パスすれば True か 0 を返し、パスしなければ False か 1 を返します。

C/RTL 協調シミュレーションの出力確認

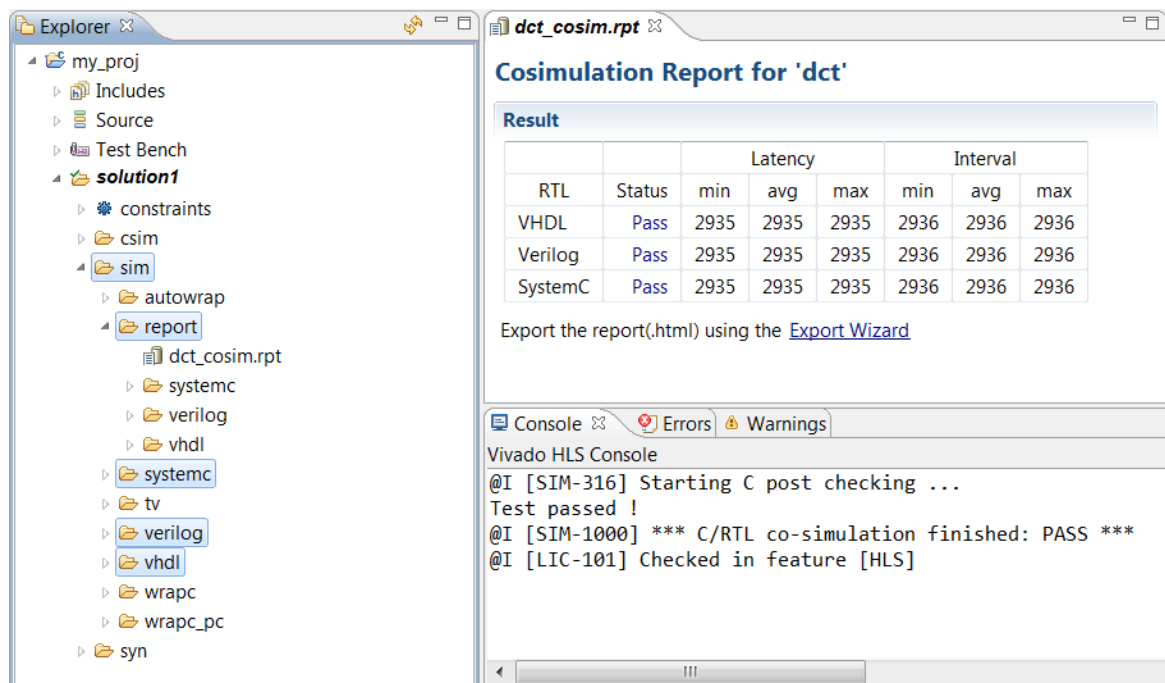
RTL 検証が終了したら、`sim` ディレクトリが `solution` フォルダに作成されます。次の図に、作成されたサブフォルダを示します。

- レポート フォルダには、シミュレーションされた RTL のタイプごとのレポートおよびログ ファイルが含まれます。
- 検証される RTL のタイプごとに検証フォルダが作成されます。検証フォルダの名前は、`verilog` または `vhdl` です。RTL 形式が検証されなかった場合、フォルダは作成されません。
- シミュレーションに使用された RTL ファイルは、検証フォルダに保存されます。
- RTL シミュレーションは、検証フォルダ内で実行されます。
- トレース ファイルなどの出力は、検証フォルダに書き込まれます。
- Vivado HLS で使用される作業フォルダは、`autowrap`、`tv`、`wrap` および `wrap_pc` です。これらのフォルダにユーザー ファイルはありません。

[C/RTL Co-Simulation] ダイアログ ボックスで [Setup Only] オプションを選択すると、検証フォルダに実行ファイ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。シミュレーションは、コマンド プロンプトでこのシミュレーション実行ファイルを実行すると、手動で実行できます。

注記: RTL 検証プロセスの詳細は、[RTL の検証](#)を参照してください。

図 35: RTL 検証の出力



Cosimulation Report for 'dict'

Result

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	2935	2935	2935	2936	2936	2936
Verilog	Pass	2935	2935	2935	2936	2936	2936
SystemC	Pass	2935	2935	2935	2936	2936	2936

Export the report(html) using the [Export Wizard](#)

Vivado HLS Console

```
@I [SIM-316] Starting C post checking ...
Test passed !
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [HLS]
```

IP のパッケージ


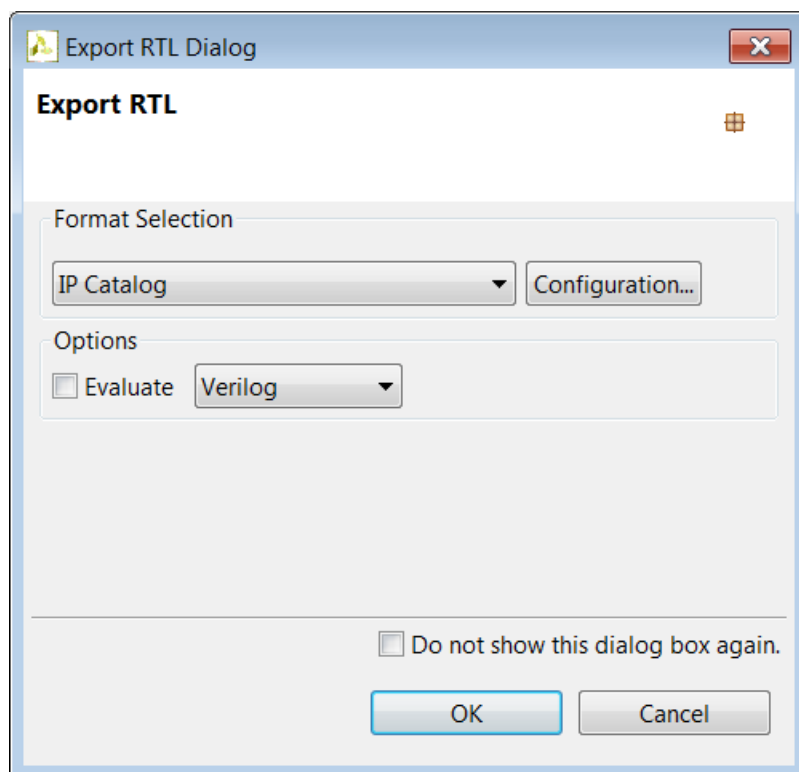
Vivado HLS デザイン フローの最後の手順では、RTL 出力を IP としてパッケージします。[Export RTL] ツールバー ボタン () または [Solution] → [Export RTL] をクリックし、[Export RTL] ダイアログ ボックスを開きます。

図 36: [RTL Export] ダイアログ ボックス

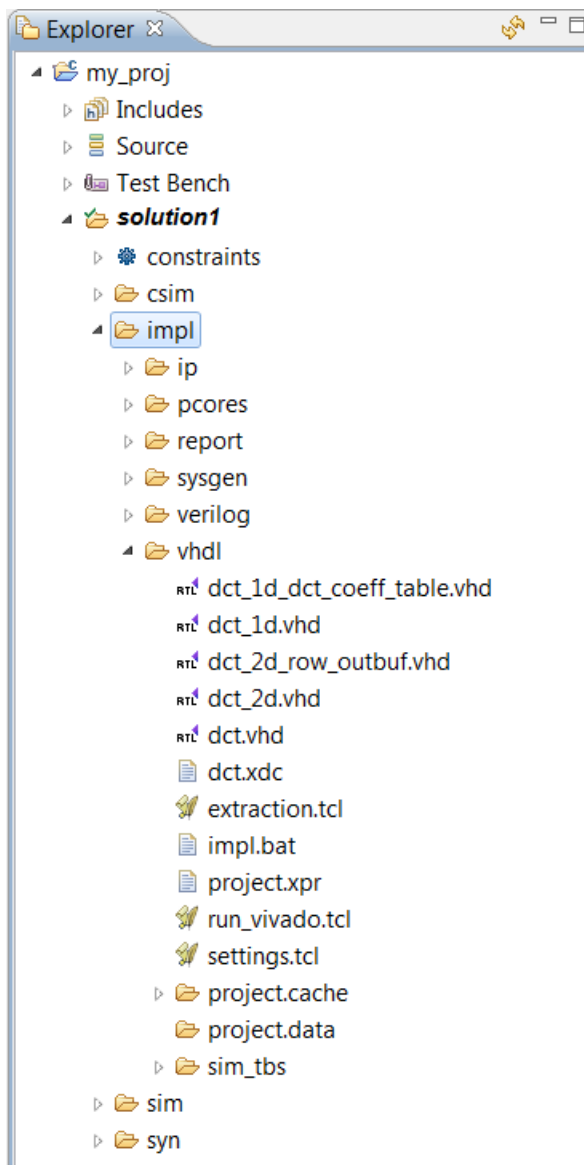


[Format Selection] ドロップダウン リストの選択肢は、合成でターゲットにした FPGA デバイスによって異なります。

IP パッケージの出力確認

[Export RTL] プロセスが終了すると、solution フォルダーに impl フォルダーが作成されます。

図 37: [Export RTL] の出力結果



どの場合にも出力には、次が含まれます。

- `report` フォルダ。[Flow] オプションをオンにすると、Verilog および VHDL 合成またはインプリメンテーションのレポートがこのフォルダに含まれます。
- `verilog` フォルダ。Verilog 形式の RTL 出力ファイルが含まれます。[Flow] オプションをオンにすると、このフォルダで RTL 合成またはインプリメンテーションが実行されます。
- `vhdl` フォルダ。VHDL 形式の RTL 出力ファイルが含まれます。[Flow] オプションをオンにすると、このフォルダで RTL 合成またはインプリメンテーションが実行されます。



重要: サイリンクス では、ユーザーの RTL 合成プロジェクトの場合、`verilog` または `vhdl` フォルダのファイルを直接使用することはお勧めしていません。サイリンクスでは、その代わりに、次に説明するパッケージ済み IP 出力ファイルを使用することを勧めします。この注記の後の説明を参照してください。

浮動小数点のデザインなど、Vivado HLS でザイリンクス IP が使用される場合、RTL ディレクトリには RTL 合成中に IP を作成するためのスクリプトが含まれます。verilog または vhd1 フォルダのファイルをコピーして RTL 合成で使用する場合、これらのフォルダ内にあるスクリプト ファイルを正しく使用するの、ユーザーの責任になります。パッケージされた IP が使用される場合、このプロセスはザイリンクス ツールにより自動的に実行されます。

[Format Selection] ドロップダウンからは、作成するその他のフォルダを指定します。指定可能なフォーマットは、[IP Catalog]、[System Generator for DSP]、および [Synthesized Checkpoint (.dcp)] です。

表 3: [RTL Export] の選択肢

フォーマット	サブフォルダー	コメント
IP Catalog	ip	Vivado IP カタログに追加できる ZIP ファイルが含まれます。ip フォルダには、ZIP ファイルの内容 (抽出済み) も含まれます。 7 シリーズまたは Zynq-7000 SoC よりも古い FPGA デバイスの場合、このオプションは使用できません。
System Generator for DSP	sysgen	この出力は、System Generator for DSP の Vivado エディションに追加できます。 7 シリーズまたは Zynq-7000 SoC よりも古い FPGA デバイスの場合、このオプションは使用できません。
Synthesized Checkpoint (.dcp)	ip	このオプションでは、Vivado Design Suite のデザインに直接追加できる Vivado チェックポイント ファイルが作成されます。 これには、RTL 合成が実行されている必要があります。このオプションをオンにすると、flow オプションと syn 設定が自動的に選択されます。 出力には、IP を HDL ファイルにインスタンス化するために使用できる HDL ラッパーが含まれます。

Vivado RTL プロジェクトの例

[Export RTL] を実行すると、Vivado RTL プロジェクトが自動的に作成されます。RTL デザインに慣れていて、Vivado RTL 環境を使用しているハードウェア設計者の場合は、これにより RTL が解析しやすくなります。

前の図に示すように、project.xpr ファイルが verilog および vhd1 フォルダに作成されます。このファイルは Vivado Design Suite 内で RTL 出力を直接開くために使用されます。

C/RTL 協調シミュレーションが Vivado HLS で実行されていれば、Vivado プロジェクトに RTL テストベンチが含まれ、デザインがシミュレーションできます。

Vivado RTL プロジェクトには、最上位デザインとして Vivado HLS からの RTL 出力が含まれます。通常、このデザインは IP としてより大きな Vivado RTL プロジェクトに統合されるべきです。この Vivado プロジェクトは、デザイン解析目的にのみ提供されており、インプリメンテーション用には提供されていません。

IP インテグレーター プロジェクトの例

[IP Catalog] を出力フォーマットとして選択すると、出力フォルダ impl/ip/example が作成されます。このフォルダには、IP インテグレーターのプロジェクトを作成するための実行ファイル (ipi_example.bat または ipi_example.csh) が含まれます。

IP インテグレーター プロジェクトを作成するには、コマンド プロンプトで ipi_example.* ファイルを実行し、作成済みの Vivado IP インテグレーター プロジェクト ファイルを開きます。

プロジェクトのアーカイブ

業界標準の ZIP に Vivado HLS プロジェクトを圧縮するには、[File > Archive] をクリックします。ZIP ファイルの名前は、[Archive Name] オプションで指定します。デフォルト設定は次のように変更できます。

- デフォルトでは、現在のアクティブ ソリューションのみがアーカイブされます。すべてのソリューションがアーカイブされるようにするには、[Active Solution Only] オプションをオフにします。
- デフォルトでは、アーカイブにはアーカイブされたソリューションからの出力結果すべてが含まれます。入力ファイルのみをアーカイブする場合は、[Include Run Results] オプションをオフにします。

コマンド プロンプトと Tcl インターフェイスの使用

Windows の場合、[スタート] メニューから [ザイリンクス Design Tools] → [Vivado 2018.x] → [Vivado HLS] → [Vivado HLS 2018.x Command Prompt] をクリックして Vivado HLS コマンド プロンプトを起動します。

Windows および Linux の場合、`vivado_hls` コマンドに `-i` オプションを付けて Vivado HLS をインタラクティブ モードで開きます。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。ログ ファイルに別の名前を指定するには、`-l <log_file>` オプションを使用します。

コマンドに関する資料にアクセスするには、`help` コマンドを使用します。すべてのコマンドのリストは、次を入力すると参照できます。

```
vivado_hls> help
```

各コマンドのヘルプは、コマンド名を指定すると表示されます。

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

コマンドまたはコマンド オプションは、自動補完機能を使用して入力できます。1 文字を指定して Tab キーを押すと、そのコマンドまたはコマンド オプションを完了するために可能性のあるオプションすべてがリストされます。さらに多くの文字を入力すると、フィルター機能により可能性のある選択肢を狭めることができます。たとえば、`open` と入力して Tab キーを押すと、`open` で始まるすべてのコマンドがリストされます。

```
vivado_hls> open <press tab key>
open
open_project
open_solution
```

`open_p` と入力して Tab キーを押すと、それ以外に `open_p` で始まるコマンドがないので、`open_project` が自動的に入力されます。

`exit` コマンドを入力すると、インタラクティブなモードが終了して、シェル プロンプトに戻ります。

```
vivado_hls> exit
```

その他の Vivado HLS のオプションは次のとおりです。

- `vivado_hls -p`: 指定したプロジェクトが開きます。

- `vivado_hls -nosplash`: Vivado HLS のスプラッシュ画面なしに GUI が開きます。
- `vivado_hls -r`: インストールのルート ディレクトリが表示されます。
- `vivado_hls -s`: システムのタイプ (Linux、Win など) が表示されます。
- `vivado_hls -v`: リリース バージョン番号が表示されます。

Tcl スクリプトに含まれるコマンドは、`-f <script_file>` オプションを使用すると、バッチモードで実行されます。

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

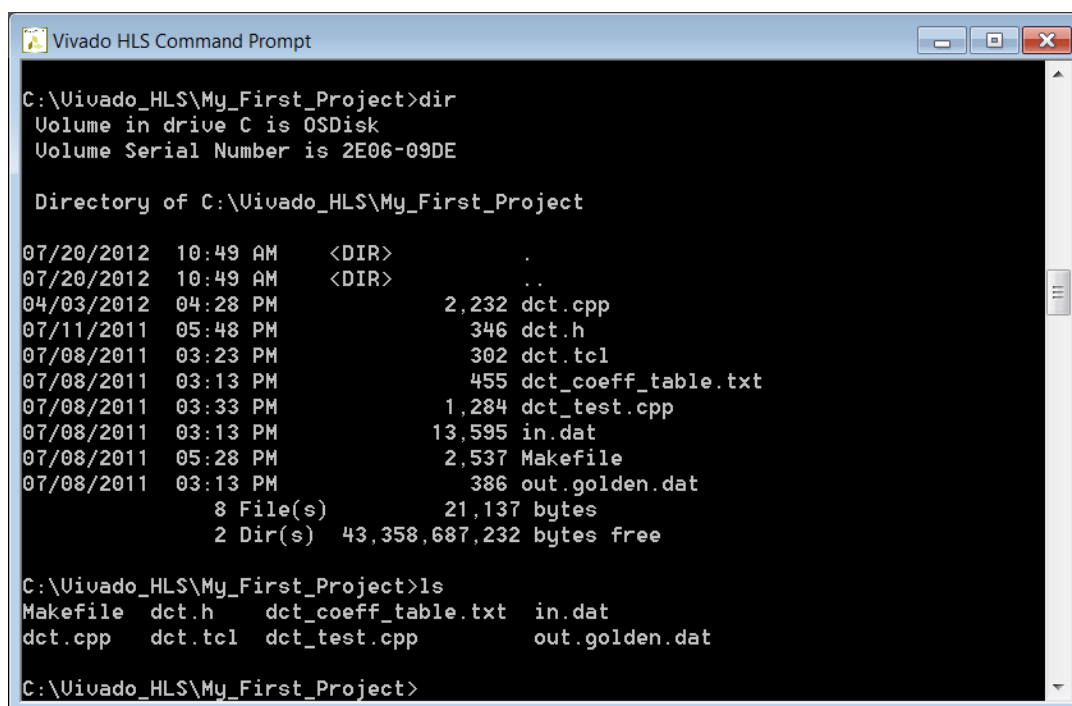
GUI でプロジェクトを作成した場合の Tcl コマンドは、solution ディレクトリの `script.tcl` ファイルに保存されます。Tcl バッチ スクリプトを作成する場合、`script.tcl` ファイルを開始点として使用すると便利です。

Windows コマンド プロンプトの理解

Windows OS では、Vivado HLS コマンド プロンプトは 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。

次は、Linux の `ls` コマンドおよび DOS の `dir` コマンドのいずれかまたは両方を使用して、ディレクトリの内容をリストしたところを示しています。

図 38: Vivado HLS コマンド プロンプト



```
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>
```

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、サポートされません。

- makefile の Linux パスは、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、パス名にクォーテーションが付いたもの (FOO := ":/") に変換されます。

ランタイムおよび容量の改善

C/RTL 協調シミュレーションに関する問題については、[RTL の検証](#)の `reduce_diskspace` オプションを参照してください。このセクションでは、合成ランタイムに関する問題について説明します。

Vivado HLS では、演算が階層別にスケジューリングされます。ループ内の演算がスケジューリングされてから、ループ、サブ関数、関数を使用する演算がスケジューリングされます。Vivado HLS のランタイムは、次のような場合に増加します。

- スケジューリングするオブジェクトが多い。
- 自由度および可能性が多くて確認に時間がかかる。

Vivado HLS では、オブジェクトがスケジューリングされます。オブジェクトが浮動小数点の乗算または単一のレジスタであっても、スケジューリングされるのはまだオブジェクトです。浮動小数点の乗算は終了に複数サイクルかかる可能性があり、インプリメントに多くのリソースが使用されますが、スケジューリングのレベルでは 1 つのオブジェクトとして扱われます。

ループの展開および配列の分割により、スケジューリングするオブジェクト数が増えるので、ランタイムが増加する可能性は高くなります。関数のインライン展開でもこの階層レベルでスケジューリングするオブジェクト数が増えるので、ランタイムが増加します。これらの最適化は、パフォーマンスを満たすために必要な場合もありますが、単純にすべての配列を分割したり、すべてのループを展開したり、すべての関数をインライン展開すると、ランタイムが増加する可能性があるため、注意が必要です。これらの最適化は、前述の最適化ストラテジを使用し、注意して適用してください。

パフォーマンスを達成するために配列を分割する必要がある場合、`config_array_partition` に `throughput_driven` オプションを使用して、スループット要件に基づいて配列を分割してみてください。

ループを展開する必要がある場合、または上位階層で PIPELINE 指示子が使用されて自動的にループが展開される場合は、ループ ボディを別の関数として取り込んでみてください。これにより、ループが展開されたときにロジックのコピーが複数作成されるのではなく、すべてのロジックが 1 つの関数に含まれるようになります。定義された階層のオブジェクトの 1 セットは早めにスケジューリングされるようになります。展開されたループがパイプライン領域で使用される場合は、この関数をパイプライン処理するようにしてください。

コードの自由度もランタイムに影響することがあります。Vivado HLS は、スループットが最も高く、レイテンシは最も低く、エリアは最小のデザインを見つけるタスクをデフォルトで与えられた上級設計者のようなものです。Vivado HLS の制約が増えると、探すオプションが少なくなるので、実行速度も速くなります。コード内のスコープ (ループ、関数または領域) に対してレイテンシ制約を使用することを考慮してください。LATENCY 指示子を同じ最小値および最大値で設定すると、そのスコープ内で可能性のある最適化の検索が減少します。

最後に、`config_schedule` 設定でスケジューリング中に使用されるエフォート レベルを制御します。これによる影響は通常上記の手法よりも少ないですが、考慮する価値はあります。デフォルトのストラテジは、Medium に設定されています。

これが Low に設定されると、Vivado HLS により、最初の結果を改善するのに費やす時間が減らされます。特に演算が多いために確認する組み合わせが多くなってしまような場合は、Low を使用することをお勧めします。デザインは理想的にはならないかもしれませんが、要件は満たして、理想に近い状態にはなる可能性があります。Low 設定で進展したら、デフォルト設定を使用して最終的な結果を作成します。

Vivado HLS で run ストラテジが 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、追加で 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、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、さまざま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なことがあります。通常は、run ストラテジはデフォルトの Medium のままにしておくことをお勧めします。

サンプル デザインおよび参考資料

チュートリアル

チュートリアルは、『Vivado Design Suite チュートリアル: 高位合成』(UG871) から取得できます。次の表は、チュートリアルの演習リストを示しています。

表 4: Vivado HLS チュートリアルの演習

チュートリアル演習	説明
Vivado HLS の概要	FIR デザインを使用した Vivado HLS の操作と主な機能の概要です。
C 検証	Hamming ウィンドウ デザインを使用して C シミュレーションを説明し、C デバッグ環境を使用して C アルゴリズムを検証します。
インターフェイス合成	インターフェイス合成を使用して、さまざまな RTL インターフェイス ポート タイプの作成方法を説明します。
任意精度型	固定小数点の任意精度型を使用して浮動小数点のファンクションをインプリメントし、さらに最適なハードウェアを生成する方法を示します。
デザイン解析	[Analysis] パースペクティブモードを使用して DCT ブロックのパフォーマンスを改善する方法を示します。
デザイン最適化	行列乗算の例を使用してアルゴリズムがどのように最適化されるかについて説明します。このチュートリアルでは、最初の状態に変更を加えることで特定のハードウェアインプリメンテーションが必要になることを示します。
RTL 検証	RTL 検証機能の使用と RTL 信号波形の解析方法を示します。
IP インテグレーターでの HLS IP の使用	2 つの HLS プリプロセスとポストプロセスブロックを IP インテグレーターを使用して FFT IP ブロックに接続する方法を示します。
Zynq-7000 SoC プロセッサ デザインでの HLS IP の使用	CPU を使用し、AXI4-Lite インターフェイスと DDR メモリと Vivado HLS ブロックを行き来する DMA ストリーミングデータを使用して、Vivado HLS ブロックを制御する方法について説明します。CPU ソースコードと SDK で必要な手順が含まれます。
System Generator for DSP での HLS IP の使用	HLS ブロックを System Generator for DSP デザイン内で使用する方法について説明します。

デザイン例

Vivado HLS サンプル デザインを [Welcome] ページから開くには、[Open Example Project] をクリックします。Examples ウィザードで、[Design Examples] をクリックしてデザインを選択します。

注記: [Welcome] ページは、Vivado HLS の GUI を起動すると表示されます。[Help] → [Welcome] をクリックすると、いつでもこのページにアクセスできます。

また、Vivado Design Suite のインストール ディレクトリ `Vivado_HLS\2018.x\examples\design` から直接コード例を開くこともできます。

各デザイン例については、次の表で説明します。

表 5: Vivado HLS のデザイン例

デザイン例	説明
2D_convolution_with_linebuffer	リソースを節約するための <code>hls::streams</code> およびライン バッファを使用してインプリメントされた 2D たたみ込み。
FFT > fft_ifft	FFT IP を使用した反転 FFT。
FFT > fft_single	パイプラインされたストリーミング I/O 付きのシングル 1024 ポイント フォワード。
FIR > fir_2ch_int	2 つのインターリーブされたチャンネルを使用した FIR フィルター。
FIR > fir_3stage	3 つの FIR (ハーフバンド FIR → ハーフバンド FIR → SRRC (Square Root Raise Cosine) FIR) を直列に接続した FIR チェーン。
FIR > fir_config	FIR CONFIG チャンネルを使用してアップデートした係数を含む FIR フィルター。
FIR > fir_srrc	SRRC FIR フィルター。
__builtin_ctz	gcc ビルトインの <code>count trailing zero</code> 関数を使用してインプリメントされたプライオリティ エンコーダー (32 および 64 ビットバージョン)。
axi_lite	AXI4-Lite インターフェイス
axi_master	AXI4 マスター インターフェイス
axi_stream_no_side_channel_data	C コードにサイドチャンネル データを含む AXI4-Stream インターフェイス。
axi_stream_side_channel_data	サイドチャンネル データを使用した AXI4-Stream インターフェイス。
dds > dds_mode_fixed	fixed モードで使用する位相オフセットと位相インクリメントの両方を使用した DDS IP。
dds > dds_mode_none	fixed モードの位相オフセットと位相インクリメントなし (mode=none) で作成された DDS IP。
dsp > atan2	HLS DSP ライブラリからの <code>arctan</code> 関数。
dsp > awgn	HLS DSP ライブラリからの Additive White Gaussian Noise (awgn) 関数。
dsp > cmpy_complex	複素データ型を使用した固定小数点複素乗算器。
dsp > cmpy_scalar	実数成分と虚数成分ごとに別々のスカラー データ型を使用した固定小数点複素乗算器。
dsp > convolution_encoder	ユーザー定義のたたみ込みコードおよび制約の長さに基づいて、入力データ ストリームのたたみ込みエンコードを実行する HLS DSP ライブラリからの <code>convolution_encoder</code> 関数。
dsp > nco	HLS DSP ライブラリからの数値制御型オシレーター (NCO) 関数。
dsp > sqrt	HLS DSP ライブラリからの平方根関数の固定小数点の CORDIC (COordinate Rotation DIgital Computer) インプリメンテーション。
dsp > viterbi_decoder	HLS DSP ライブラリからの Viterbi デコーダー。

表 5: Vivado HLS のデザイン例 (続き)

デザイン例	説明
fp_mul_pow2	2 のべき乗を使用した効率的なエリアおよびタイミングの浮動小数点乗算インプリメンテーション (小型の加算器 1 つと、浮動小数点コアと DSP リソースの代わりにオプションのリミットチェック ボックスをいくつか使用)。
fxp_sqrt	ビット シリアルで完全にパイプライン可能な ap_fixed 型の平方根インプリメンテーション。
hls_stream	hls::stream を使用したマルチレート データフロー (8 ビット I/O、32 ビット データ プロセスおよびデシメーション)。
linear_algebra > cholesky	パラメーター変更可能なコレスキー関数。
linear_algebra > cholesky_alt	代替コレスキー関数。
linear_algebra > cholesky_alt_inverse	別のインプリメンテーションを選択するためのカスタマイズされた特性クラスを使用したコレスキー関数。
linear_algebra > cholesky_complex	複素データ型を使用したコレスキー関数。
linear_algebra > cholesky_inverse	パラメーター変更可能なコレスキー反転関数。
linear_algebra > implementation_targets	インプリメンテーション ターゲット例。 詳細は、 線形代数関数の最適化 を参照してください。
linear_algebra > matrix_multiply	パラメーター指定可能な行列乗算関数。
linear_algebra > matrix_multiply_alt	代替行列乗算関数。
linear_algebra > qr_inverse	パラメーター変更可能な QR 反転関数。
linear_algebra > qrf	パラメーター変更可能な QRF 関数。
linear_algebra > qrf_alt	別のパラメーター変更可能な QRF 関数。
linear_algebra > svd	パラメーター変更可能な SVD 関数。
linear_algebra > svd_pairs	代替ペアの SVD インプリメンテーションを使用したパラメーター変更可能な SVD 関数。
loop_labels > loop_label	ラベル付きループ。
loop_labels > no_loop_label	ラベルなしループ。
memory_porting_and_ii	配列分割指示子を使用して改善された開始間隔。
perfect_loop > perfect	完全ループ。
perfect_loop > semi_perfect	半完全ループ。
rom_init_c	ROM がインプリメントされるようにするためにサブ関数を使用してコード記述された配列。
window_fn_float	単精度浮動小数点のウィンドウイング関数。コンパイル時間選択 (Rectangular (none)、Hann、Hamming または Gaussian) をした C++ テンプレート クラス例。
window_fn_fxpt	固定小数点ウィンドウイング関数。コンパイル時間選択 (Rectangular (none)、Hann、Hamming または Gaussian) をした C++ テンプレート クラス例。

コード例

Vivado HLS コード例には、さまざまなコーディング手法の例が提供されています。これらは、さまざまな C、C++、SystemC コンストラクトの Vivado HLS 合成結果を示すための小さい例です。

Vivado HLS サンプル コードを [Welcome] ページから開くには、[Open Example Project] をクリックします。Examples ウィザードで、[Coding Style Examples] をクリックしてデザインを選択します。

注記: [Welcome] ページは、Vivado HLS の GUI を起動すると表示されます。[Help]→[Welcome] をクリックすると、いつでもこのページにアクセスできます。

また、Vivado Design Suite のインストール ディレクトリ `Vivado_HLS\2018.x\examples\coding` から直接コード例を開くこともできます。

各コード例については、次の表で説明します。

表 6: Vivado VHDL コード例

コード例	説明
apint_arith	C の <code>ap_cint</code> 型の使用。
apint_promotion	C の <code>ap_cint</code> 型の整数拡張問題を回避するために必要な型変換。
array_arith	インターフェイス 配列での演算の使用。
array_FIFO	FIFO インターフェイスのインプリメンテーション。
array_mem_bottleneck	配列へのアクセスによるパフォーマンスのボトルネック作成。
array_mem_perform	<code>array_mem_bottleneck</code> 例によるパフォーマンス ボトルネックのソリューション。
array_RAM	ブロック RAM インターフェイスのインプリメンテーション。
array_ROM	ROM がどのように自動的に推論されるかを示す例。
array_ROM_math_init	さらに複雑な場合の ROM の推論方法を示す例。
cpp_ap_fixed	C++ の <code>ap_int</code> 型の使用。
cpp_ap_int_arith	演算用の C++ の <code>ap_int</code> 型の使用。
cpp_FIR	オブジェクト指向のコード スタイルを使用した C++ デザイン例。
cpp_math	IEEE exact ではない演算の結果を比較する際にテストベンチでトレランスをどのように使用するかを示す浮動小数点の数学デザイン例。
cpp_template	C++ テンプレート例。
func_sized	インターフェイスでデータ幅を定義することで演算のサイズを固定。
hier_func	テストベンチおよびデザイン ファイルとしてファイルを追加する例。
hier_func2	テストベンチおよびデザイン ファイルとしてファイルを追加する例。階層の下位ブロックを合成する例。
hier_func3	テストベンチとデザイン関数を同じファイルに組み合わせる例。
hier_func4	合成されるコードを回避するための定義済み macro <code>_SYNTHESIS_</code> の使用。 <code>_SYNTHESIS_</code> マクロは、合成されるコードにのみ使用します。このマクロは C シミュレーションまたは C RTL 協調シミュレーションには従っていないので、テストベンチには使用しないでください。
loop_functions	並列実行のためのループの関数への変換。
loop_imperfect	不完全なループ例。
loop_max_bounds	ループが展開できるように最大境界の使用。
loop_perfect	完全なループ例。
loop_pipeline	ループ パイプラインの例。
loop_sequential	シーケンシャル ループ。

表 6: Vivado VHDL コード例 (続き)

コード例	説明
loop_sequential_assert	assert 文の使用。
loop_var	変数境界付きのループ。
malloc_removed	コードからのマクロの削除例。
pointer_arith	ポインター演算例。
pointer_array	ポインターの配列。
pointer_basic	基本的なポインター例。
pointer_cast_native	ネイティブ C 型間のポインター型変換。
pointer_double	Pointer-to-Pointer の例。
pointer_multi	複数ポインター ターゲットを使用する例。
pointer_stream_better	volatile キーワードをインターフェイスで使用方法を示す例。
pointer_stream_good	明示的なポインター演算を使用したマルチリード ポインターの例。
sc_combo_method	SystemC の組み合わせデザイン例。
sc_FIFO_port	SystemC FIFO ポインター例。
sc_multi_clock	複数クロックを含む SystemC 例。
sc_RAM_port	SystemC ブロック RAM ポート例。
sc_sequ_thead	SystemC シーケンシャル デザイン例。
struct_port	インターフェイスでの struct の使用。
sum_io	最上位インターフェイス ポートの例。
types_composite	複合データ型。
types_float_double	float 型から double 型への変換。
types_global	グローバル変数の使用。
types_standard	標準 C 型の例。
types_union	union の例。

効率的なハードウェアのためのデータ型

C ベースのネイティブ データ型は、8 ビット境界 (8、16、32、64 ビット) にあります。RTL バス (ハードウェアに対応) では、任意精度のデータ長がサポートされます。標準 C データ型を使用すると、効率の悪いハードウェアになることがあります。たとえば、FPGA での基本的な乗算単位は DSP48 マクロで、18*18 ビットの乗算器が提供されます。17 ビットの乗算が必要な場合は、32 ビットの C データ型でこれをインプリメントしないようにしてください。32 ビットの C データ型の場合、乗算器をインプリメントするのに DSP48 マクロが 1 つで十分なのに 3 つも必要となるからです。

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てから、C シミュレーションを再実行して機能が同一または使用可能であることを検証できる点です。ビット幅が狭い方が小型で高速なハードウェア演算になります。これにより、より多くのロジックが FPGA に配置できるようになり、ロジックがより高速のクロック周波数で実行されるようになります。

ハードウェア効率の高いデータ型の利点

次のコードは、基本的な算術演算を実行します。

```
#include "types.h"

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
                ) {

    // Basic arithmetic operations
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}
```

dinA_t、dinB_t などのデータ型は、ヘッダー ファイル types.h で定義されます。types.h などのプロジェクト全体に適用されるヘッダー ファイルを使用すると、標準 C 型から任意精度型に簡単に移行でき、任意精度型を最適なサイズに調整できるようになります。

上記の例のデータ型は次のように定義されているとします。

```
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;
```

この場合、合成後の結果は次のようになります。

```
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock   | Target | Estimated | Uncertainty |
  +-----+-----+-----+-----+
  | default | 4.00   | 3.85      | 0.50        |
  +-----+-----+-----+-----+

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+
  | Latency | Interval | Pipeline |
  | min | max | min | max | Type |
  +-----+-----+-----+-----+
  | 66 | 66 | 67 | 67 | none |
  +-----+-----+-----+-----+

* Summary:
+-----+-----+-----+-----+-----+
| Name          | BRAM_18K | DSP48E | FF   | LUT   |
+-----+-----+-----+-----+-----+
| Expression    |          |        | 0    | 17    |
| FIFO          |          |        | -    | -     |
```

Instance		-	1	17920	17152
Memory		-	-	-	-
Multiplexer		-	-	-	-
Register		-	-	7	-
+-----+					
Total		0	1	17927	17169
+-----+					
Available		650	600	202800	101400
+-----+					
Utilization (%)		0	~0	8	16
+-----+					

次の例のように、データ幅をインプリメントするのに標準 C 型を使用する必要がなく、一部の幅は狭いが、次に小さい標準 C 型よりは広い場合があるとします。

```
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;
```

合成後の結果には、最大クロック周波数とレイテンシが改善し、エリア使用量が 75% 削減されたことが示されます。

```
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+-----+
  | default | 4.00| 3.49| 0.50|
  +-----+-----+-----+-----+

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+
  | Latency | Interval | Pipeline|
  | min | max | min | max | Type |
  +-----+-----+-----+-----+
  | 35| 35| 36| 36| none |
  +-----+-----+-----+-----+

* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K| DSP48E| FF | LUT |
+-----+-----+-----+-----+
| Expression | -| -| 0| 13|
| FIFO | -| -| -| -|
| Instance | -| 1| 4764| 4560|
| Memory | -| -| -| -|
| Multiplexer | -| -| -| -|
| Register | -| -| 6| -|
+-----+-----+-----+-----+
| Total | 0| 1| 4770| 4573|
+-----+-----+-----+-----+
```

Available		650		600		202800		101400	
+-----+									
Utilization (%)		0		~0		2		4	
+-----+									

2 つのデザイン間でレイテンシが大きく異なるのは、除算および余りの計算に複数サイクルかかるからです。デザインを無理に標準 C データ型に適合させるよりも、正確なデータ型を使用した方が、同じ精度で高速でリソース使用量の少ない高品質の FPGA インプリメンテーションが得られます。

任意精度整数データ型の概要

Vivado® HLS では、C および C++ の整数および固定小数点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。

表 7: 任意精度データ型

言語	整数型	必要なヘッダー
C	[u]int<W> (1024 bits)	#include "ap_cint.h"
C++	ap_[u]int<W> (1024 ビット) 32K ビット幅までに拡張可能。	#include "ap_int.h"
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
System C	sc_[u]int<W> (64 ビット) sc_[u]bigint<W> (512 ビット)	#include "systemc.h"
System C	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h"

Vivado® HLS には、任意精度型を定義するヘッダー ファイルもスタンドアロン パッケージとして含まれており、ソース コードで使えるようになっています。xilinx_hls_lib_<release_number>.tgz パッケージは、Vivado® HLS インストール ディレクトリの include ディレクトリに含まれます。パッケージには、ap_cint.h で定義された C 任意精度データ型は含まれません。これらのデータ型は、標準 C コンパイラとは一緒に使用できず、Vivado® HLS でのみ使用できるようになっています。

C 言語での任意精度整数型

C 言語では、ヘッダー ファイル ap_cint.h で任意精度の整数型 [u]int が定義されます。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。

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 で任意精度の整数型 ap_[u]int が定義されます。C++ 関数で任意精度の整数型を使用するには、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
```

ap_[u]int 型に使用可能なデフォルトの最大幅は 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。



注意: ROM 合成は、APFixed: を使用した場合時間がかかることがあります。int に変更すると、合成は速くなります。次に例を示します。

```
static ap_fixed<32> a[32][depth] =
```

次のように変更できます。

```
static int a[32][depth] =
```

次は AP_INT_MAX_W を上書きする例です。

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"

ap_int<4096> very_wide_var;
```

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。ヘッダー ファイルには、SystemC の sc_int<>、sc_uint<>、sc_bigint<>、および sc_bignint<> 型が含まれます。

任意精度固定小数点データ型の概要

固定小数点データ型では、整数ビットおよび小数ビットとしてデータが記述されます。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 ビットが小数部を定義しています。変数は、符号付きとして指定され、量子化モードは正の無限大の方向に丸めらるよう設定されています。オーバーフロー モードは指定されていないので、オーバーフローにはデフォルトの折り返しモードが使用されます。

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

変数にさまざまなビット数や精度が含まれるような計算を実行する場合は、2 進小数点が自動的に揃えられます。

固定小数点を使用して実行された C++/SystemC のビヘイビアは結果のハードウェアと同じになります。これにより、ビット精度、量子化、およびオーバーフロー ビヘイビアを高速の C レベル シミュレーションで解析できるようになります。

固定小数点型は、終了するのに多くのクロック サイクルを必要とする浮動小数点型の代わりに使用できます。浮動小数点型の範囲がすべて必要な場合を除き、固定小数点型で同じ精度をインプリメントでき、より小型で高速なハードウェアにできることがよくあります。

次の表に、`ap_fixed` 型の識別子についてまとめます。

表 8: 固定小数点の識別子のまとめ

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 (2 進小数点より上位のビット数)		
Q	量子化モード 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	ap_fixed 型	説明
	SC_RND	AP_RND	Round to plus infinity
	SC_RND_ZERO	AP_RND_ZERO	Round to zero
	SC_RND_MIN_INF	AP_RND_MIN_INF	Round to minus infinity
	SC_RND_INF	AP_RND_INF	Round to infinity
	SC_RND_CONV	AP_RND_CONV	収束丸め
	SC_TRN	AP_TRN	負の無限大への切り捨て (デフォルト)
	SC_TRN_ZERO	AP_TRN_ZERO	0 への切り捨て
O	オーバーフロー モード。 演算結果が最大値 (負の数値の場合は最小値) を超えるビヘイビアを検出 (値は結果変数に格納可能)。		
	SystemC 型	ap_fixed 型	説明
	SC_SAT	AP_SAT	飽和
	SC_SAT_ZERO	AP_SAT_ZERO	0 への飽和
	SC_SAT_SYM	AP_SAT_SYM	対称飽和
	SC_WRAP	AP_WRAP	折り返し (デフォルト)
	SC_WRAP_SM	AP_WRAP_SM	符号絶対値のラップ
N	オーバーフロー折り返しモードでの飽和ビット数を定義。		

`ap_[u]fixed` データ型に使用可能なデフォルトの最大幅は 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。



注意: ROM 合成は、`static APFixed_2_2 CAcode_sat[32][CACODE_LEN] =` の場合遅く、`APFixed` を `int (static int CAcode_sat[32][CACODE_LEN] =)` に変更すると、合成が速くなります。

次は AP_INT_MAX_W を上書きする例です。

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_fixed.h"

ap_fixed<4096> very_wide_var;
```

Vivado HLS を使用する場合は、任意精度データ型の使用をお勧めします。前の例で示したように、任意精度データ型を使用すると、ハードウェア インプリメンテーションの質がかなり向上するという利点があります。

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フォーマル関数引数を使用して 0 時間で実行されます。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実行する必要があり、通常は特定の I/O (入力/出力) プロトコルを使用して実行されます。

Vivado HLS では、使用する I/O プロトコルを指定するのに次のソリューションがサポートされます。

- インターフェイス合成: 効率的な業界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作成されます。

インターフェイス合成

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の概要

次のコードは、インターフェイス合成の全体的な概要を示しています。

```
#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;

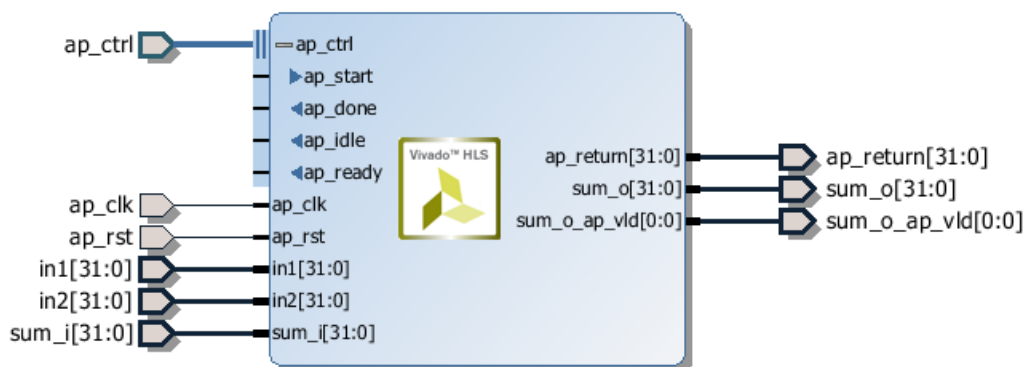
    return temp;
}
```

上記の例には、次が含まれます。

- 2 つの値渡し入力 `in1` および `in2`。
- 読み出しおよび書き込みの両方に使用されるポインター `sum`。
- 関数 `return`、`temp` の値。

デフォルトのインターフェイス合成設定では、デザインは次の図に示すように、ポートを含む RTL ブロックに合成されます。

図 39: デフォルト インターフェイスの合成後の RTL ポート



Vivado HLS は RTL デザインに 3 種類のポートを作成します。

- クロックおよびリセット ポート: `ap_clk` および `ap_rst`。
- ブロック レベルのインターフェイス プロトコル。これらは、上の図では `ap_start`、`ap_done`、`ap_ready`、および `ap_idle` に展開されています。
- ポート レベルインターフェイス プロトコル。最上位関数の各引数および関数の戻り値 (関数が値を返す場合) に対して作成されます。この例の場合、これらのポートは `in1`、`in2`、`sum_i`、`sum_o`、`sum_o_ap_vld`、および `ap_return` です。

クロックおよびリセット ポート

操作が 1 サイクルでは終了しない場合に作成されます。

チップ イネーブル ポートは [Solution]→[Solution Settings]→[General] で `config_interface` コンフィギュレーションを使用すると、ブロック全体に追加できます。

リセットの動作は、`config_rtl` コンフィギュレーションで制御されます。

ブロック レベルのインターフェイス プロトコル

ブロック レベルのインターフェイス プロトコルは、デフォルトでデザインに追加されます。これらの信号はポート レベルの I/O プロトコルとは別にブロックを制御します。これらのポートは、ブロックのデータ処理の開始 (`ap_start`)、新しい入力を受け入れ準備 (`ap_ready`)、デザインのアイドル状態 (`ap_idle`)、操作の終了 (`ap_done`) を制御します。

ポート レベルのインターフェイス プロトコル

この信号グループは、データ ポートです。作成される I/O プロトコルは、C 引数のタイプとデフォルトによって異なります。ブロック レベルのプロトコルが使用されてブロックの演算が開始したら、ポート レベルの I/O プロトコルを使用して、データをブロック内外に順に送信できます。

デフォルトでは、入力の値渡し引数とポインターが単純なワイヤ ポートとして合成されます。ハンドシェイク信号は関連付けられません。このため、上記の例の場合、入力ポートは I/O プロトコルなしで、データ ポートのみでインプリメントされます。ポートに I/O プロトコルがない場合 (デフォルトまたはそのように設計されている場合)、入力データは読み出されるまで安定している必要があります。

デフォルトでは、出力ポインターは関連する出力 Valid 信号を使用してインプリメントされ、出力データが有効になるタイミングを示します。上記の例の場合、出力ポートが関連する出力 Valid ポート (sum_o_ap_vld) を使用してインプリメントされ、いつポートのデータが有効になって読み出し可能になるかを示すようになっています。出力ポートに関連付けられた I/O ポートがない場合は、データを読み出すタイミングを判断するのは困難です。出力には常に I/O プロトコルを使用するようにすることをお勧めします。

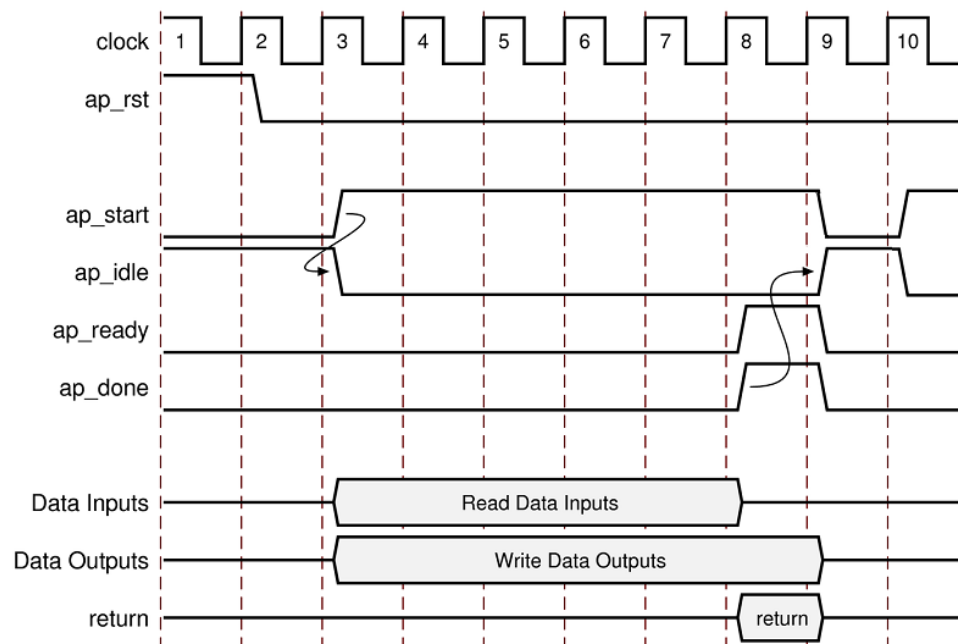
読み出しおよび書き込みが実行される関数引数は、入力ポートと出力ポートに分割されます。上記の例の場合、sum は入力ポート sum_i および出力ポート sum_o に関連する I/O プロトコル ポート sum_o_ap_vld を含めてインプリメントされます。

関数に戻り値がある場合、その戻り値を示す出力ポート ap_return がインプリメントされます。デザインが 1 つのトランザクション (C 関数の 1 つの実行) で終了する場合、ブロックレベルのプロトコルで ap_done 信号を使用して関数が終了したことが示されます。これは、ap_return ポートのデータが有効で読み出し可能であることも示します。

注記: 最上位関数の戻り値はポインターにはできません。

たとえば、次のコード例はタイミング ビヘイビアーを示しています (そのターゲット テクノロジとクロック周波数でクロック サイクルごとに 1 つの加算が可能)。

図 40: 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、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クロックで読み出されます。読み出されるタイミングは Vivado HLS で自動的にスケジューリングされます。すべての入力を読み出されると、ap_ready 信号が High にアサートされます。
- 出力 sum が計算されると、関連する出力ハンドシェイク (sum_o_ap_vld) でデータが有効なことが示されます。
- 関数が終了すると、ap_done がアサートされます。これは、ap_return のデータが有効であることも示します。
- ap_idle ポートが High にアサートされ、デザインが再開を待機した状態であることを示します。

インターフェイス合成の I/O プロトコル

インターフェイス合成で作成されるインターフェイスのタイプは、C 引数のデータ型、デフォルトのインターフェイスモードおよび INTERFACE 最適化指示子によって異なります。次の図に、各 C 引数のデータ型で指定可能なインターフェイス プロトコル モードを示します。この図では、次の省略語が使用されています。

- D: 各データ型のデフォルトのインターフェイス モード。

注記: 不適切なインターフェイスが指定されると、Vivado HLS でメッセージが表示され、デフォルトのインターフェイスモードがインプリメントされます。

- I: 読み出しのみの入力引数。
- O: 書き込みのみの出力引数。
- I/O: 読み出しおよび書き込みの両方に使用される入力/出力引数。

図 41: データ型およびインターフェイス合成のサポート

Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									



Supported D = Default Interface



Not Supported

X14293

波形を含めたインターフェイス プロトコルの詳細は、[インターフェイス合成リファレンス](#)を参照してください。次は、各インターフェイス モードの概要を示しています。

ブロック レベルのインターフェイス プロトコル

ブロック レベルのインターフェイス プロトコルは、`ap_ctrl_none`、`ap_ctrl_hs`、および `ap_ctrl_chain` です。関数または関数リターンに対してのみ指定できます。指示子が GUI で指定される場合は、これらのプロトコルがその関数リターンに適用されます。関数で戻り値が使用されない場合でも、ブロック レベルのプロトコルは関数リターンに指定できます。

デフォルトのプロトコルは、前の例で説明された `ap_ctrl_hs` モードです。`ap_ctrl_chain` プロトコルは `ap_ctrl_hs` と類似していますが、補足のために入力ポート `ap_continue` が追加されている点が異なります。関数が終了したときに `ap_continue` ポートがロジック 0 であれば、ブロックが操作を停止し、次のトランザクションに進まなくなります。次のトランザクションは `ap_continue` がロジック 1 にアサートされた場合にのみ処理されます。

`ap_ctrl_none` モードでは、ブロック レベルの I/O プロトコルなしにデザインがインプリメントされます。

関数の戻り値が AXI4-Lite インターフェイス (`s_axilite`) として指定される場合も、ブロック レベルのインターフェイスのポートはすべて AXI4-Lite インターフェイスにまとめられます。これは、CPU などの別のデバイスがこのブロックの動作の開始および停止を設定および制御する際によく使用される方法です。

ポート レベルのインターフェイス プロトコル: AXI4 インターフェイス

Vivado HLS でサポートされる AXI4 インターフェイスには、AXI4-Stream (`axis`)、AXI4-Lite (`s_axilite`) および AXI4 マスター インターフェイス (`m_axi`) などがあり、次のように指定できます。

- AXI4-Stream インターフェイス: 入力引数または出力引数のみにしか指定できず、入力/出力引数には指定できません。
- AXI4-Lite インターフェイス: ストリーム以外のどのタイプの引数にでも指定できます。複数の引数を同じ AXI4-Lite インターフェイスにまとめることができます。
- AXI4 マスター インターフェイス: 配列およびポインター (C++ ではリファレンスも) にのみ指定できます。複数の引数を同じ AXI4 インターフェイスにまとめることができます。

ポート レベルのインターフェイス プロトコル: I/O プロトコルなし

`ap_none` および `ap_stable` モードは、ポートに I/O プロトコルを追加しないことを指定します。これらのモードを指定すると、引数は関連信号のないデータ ポートとしてインプリメントされます。`ap_none` モードは、スカラー入力のデフォルトです。`ap_stable` モードは、デバイスがリセット モードのときにのみ変化するコンフィギュレーション入力用のモードです。

ポート レベルのインターフェイス プロトコル: ワイヤ ハンドシェイク

インターフェイス モード `ap_hs` には、データ ポートの付いた 2 方向のハンドシェイク信号が含まれます。業界標準の有効 (`valid`) と承認 (`acknowledge`) ハンドシェイクに使用されます。`ap_vld` モードの場合は `valid` ポートしかなく、`ap_ack` モードの場合は `acknowledge` ポートしかありません。

入出力引数には、`ap_ovld` モードを使用します。入出力が入力ポートと出力ポートに分割される場合は、`ap_none` モードが入力ポートに、`ap_vld` モードが出力ポートに使用されます。これが、読み出しと書き込みの両方に使用されるポインター引数のデフォルトです。

`ap_hs` モードは、順に読み出しまたは書き込みが実行される配列に使用できます。Vivado HLS では読み出しまたは書き込みアクセスが順に実行されない場合はそれが検出され、エラー メッセージが表示されて合成が停止します。アクセス順序が決定できない場合は、Vivado HLS で警告メッセージが表示されます。

ポート レベルのインターフェイス プロトコル: メモリ インターフェイス

配列引数は、デフォルトでは `ap_memory` インターフェイスとしてインプリメントされます。これは、データ ポート、アドレス ポート、チップイネーブル ポート、ライトイネーブル ポートを含む標準ブロック RAM インターフェイスです。

`ap_memory` インターフェイスは、デュアル ポート インターフェイスのシングル ポートとしてインプリメントされる可能性があります。Vivado HLS でデュアル ポート インターフェイスを使用することで開始間隔 (II) が削減されると判断された場合は、デュアル ポート インターフェイスが自動的にインプリメントされます。RESOURCE 指示子を使用してメモリ リソースを指定し、シングル ポート ブロック RAM を含む配列に指定した場合は、シングル ポート インターフェイスがインプリメントされます。また、RESOURCE でデュアル ポート インターフェイスを指定していても、このインターフェイスを使用しても利点がないと Vivado HLS で判断された場合は、シングル ポート インターフェイスが自動的にインプリメントされます。

`bram` インターフェイス モードは、`ap_memory` と機能的には同じですが、デザインが Vivado IP インテグレーターで使用される際のポートのインプリメント方法が異なります。

- `ap_memory` インターフェイスは複数の別々のポートとして表示されます。
- `bram` インターフェイスは 1 つのまとまったポートとして表示され、1 つのポイント ツー ポイント接続を使用してザイリンクス ブロック RAM に接続できます。

配列がシーケンシャルにアクセスされる場合は、`ap_fifo` インターフェイスを使用できます。`ap_hs` インターフェイスを使用した場合と同様、データ アクセスがシーケンシャルでないと判断されたら Vivado HLS は停止します。アクセスがシーケンシャルかどうか判別できなかった場合は警告メッセージが表示され、シーケンシャルであると判断された場合はメッセージは表示されません。`ap_fifo` は、読み出しまたは書き込みのいずれかにのみ使用可能で、両方には使用できません。

`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ーションできます。バスブリッジでは、バースト書き込みをキャッシュできるようになっている必要があります。

インターフェイス合成および構造体

インターフェイスの構造体 (struct) は、デフォルトでメンバー要素に分割され、ポートは各メンバー要素に個別にインプリメントされます。INTERFACE 指示子を使用しない場合、構造体のメンバー要素がインプリメントされます。

構造体の配列は、複数の配列 (構造体の各メンバーに別々の配列) としてインプリメントされます。

1 つの構造体に含まれるすべての要素を 1 つの幅のベクターにパックするには、DATA_PACK 最適化指示子を行います。これにより、構造体のすべてのメンバーを同時に読み出しおよび書き込みできます。構造体のメンバー要素は C コードに記述されている順序でベクターに配置されます (構造体の最初の要素がベクターの LSB に、最後の要素がベクターの MSB に配置される)。構造体の配列は個別の配列要素に分割され、下位から上位の順のベクターに配置されます。

注記: DATA_PACK 最適化では、ほかの構造体を含む構造体のパックはサポートされません。

大きな配列を含む構造体に DATA_PACK 最適化を使用する際には注意が必要です。配列に `int` 型の要素が 4096 個含まれる場合、ベクター (およびポート) の幅は $4096 \times 32 = 131072$ ビットになります。Vivado HLS でこの RTL デザインは作成できますが、FPGA インプリメンテーション中に論理合成でこれが配線できることはほとんどありません。

単一幅のベクターを DATA_PACK 指示子を使用して作成した場合、1 クロック サイクルでより多くのデータにアクセスできるようになります。これは、構造体(struct) に配列(array) が含まれている場合です。データが 1 クロック サイクルでアクセスできると、このデータを消費するループを展開してスループットが改善される場合にのみ、Vivado HLS でこれらのループが自動的に展開されます。ループを完全または部分展開すると、1 クロック サイクルでこれらの追加データを消費するのに十分なハードウェアを作成できます。この機能は `config_unroll` コマンドと `tripcount_threshold` オプションを使用して制御されます。次の例では、展開するとスループットが改善される場合にのみトリップカウントが 16 未満のループが自動的に展開されます。

```
config_unroll -tripcount_threshold 16
```

注記: 構造体は、DATA_PACK 最適化を使用してパックされた場合、AXIM インターフェイスのみでサポートされます。

DATA_PACK を使用した struct ポートが AXI4 インターフェイスを使用してインプリメントされる場合は、DATA_PACK に `-byte_pad` オプションを付けてみてください。`-byte_pad` オプションを使用すると、メンバー要素が 8 ビット境界に自動的に揃えられます。このアライメントは、一部のザイリンクス IP では必須です。DATA_PACK を使用する AXI4 ポートをインプリメントする場合は、それが接続されるザイリンクス IP の資料を参照して、バイト アライメントが必要かどうかを判断してください。

次の図は、次のコード例の struct ポートをインプリメントするオプションを示しています。

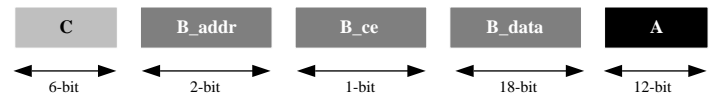
```
typedef struct{
    int12 A;
    int18 B[4];
    int6 C;
} my_data;

void foo(my_data *a )
```

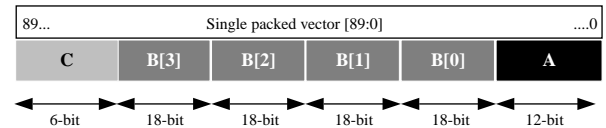
- デフォルトでは、メンバーが個別のポートとしてインプリメントされます。配列には、複数のポート (data、addr など) が含まれます。
- DATA_PACK を使用すると単一幅のポートになります。
- DATA_PACK を `struct_level` バイト パディングを指定して使用すると、構造体全体が次の 8 ビット境界に揃えられます。
- DATA_PACK を `field_level` バイト パディングを指定して使用すると、各構造体メンバーが次の 8 ビット境界に揃えられます。
- データ パックで作成されたポートまたはバスの最大ビット幅は 8192 です。

図 42: DATA_PACK -byte_pad アライメント オプション

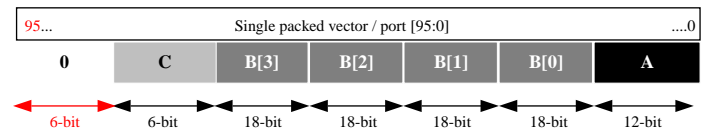
Struct Port Implementation



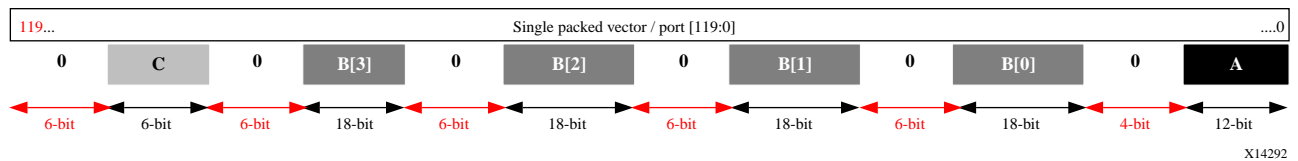
DATA_PACK optimization



DATA_PACK optimization with byte_pad on the struct_level



DATA_PACK optimization with byte_pad on the field_level



X14292

構造体に配列が含まれる場合、これらの配列は、ARRAY_PARTITION 指示子を使用して分割、ARRAY_RESHAPE 指示子を使用して配列を再形成、または分割された要素をより幅の広い配列にまとめ直すことにより、最適化できます。DATA_PACK 指示子を使用すると、ARRAY_RESHAPE と同様の動作が実行され、再形成された配列が構造体内のほかの要素とまとめられます。

構造体は DATA_PACK を使用して最適化した後に分割または再形成することはできません。DATA_PACK、ARRAY_PARTITION、および ARRAY_RESHAPE 指示子は一緒に使用できません。

インターフェイス合成およびマルチアクセス ポインター

複数回アクセスされるポインターを使用すると、合成後に予期しない動作になることがあります。次の例では、ポインター d_i が 4 回読み出され、ポインター d_o に 2 回書き込まれます。

```
#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;
}
```

このコードを合成すると、入力ポートが 1 回読み出され、出力ポートに 1 回書き込まれる RTL になります。標準的な C コンパイラと同様、Vivado HLS では最適化で不要なポインター アクセスが削除されます。上記のコードを `d_i` で 4 回読み出し、`d_o` に 2 回書き込むようにインプリメントするには、次の例のようにポインターを `volatile` と指定する必要があります。

```
#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;
}
```

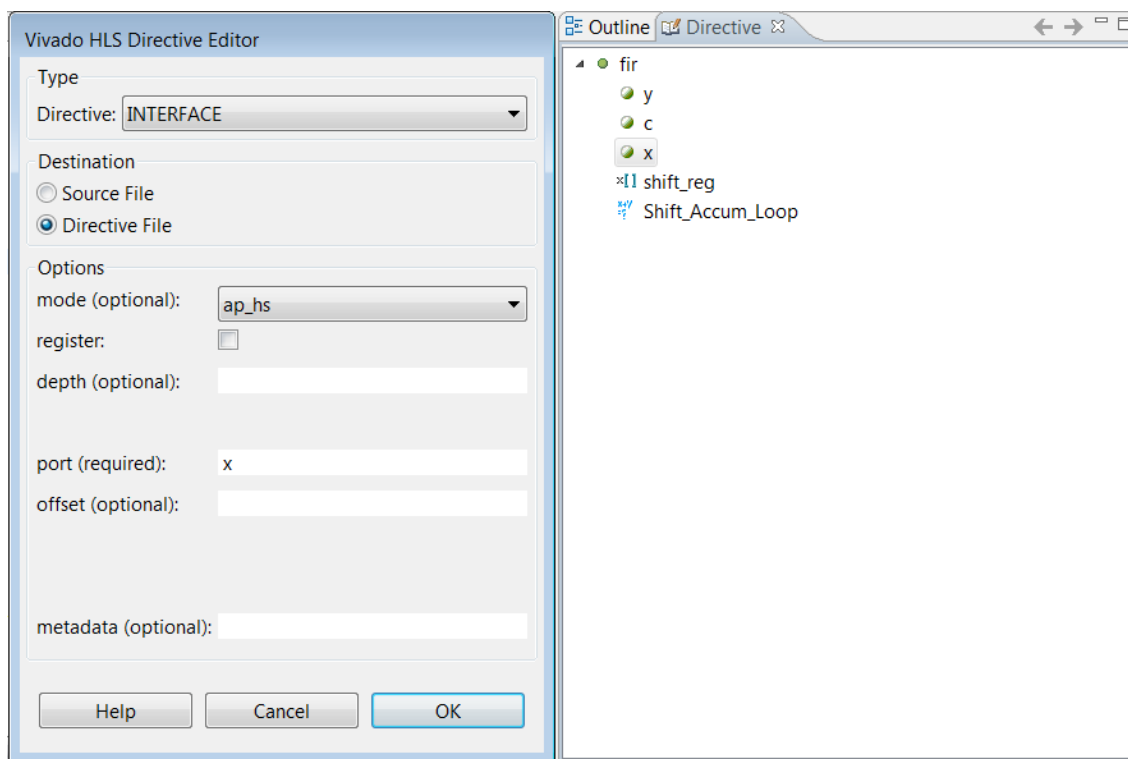
この C コードでも問題となる可能性があります。テストベンチを使用した場合、`d_i` に複数の値を提供する方法はなく、最後の書き込み以外の `d_o` への書き込みを検証する方法也没有ありません。マルチアクセス ポインターはサポートされていますが、`hls::stream` クラスを使用して必要な動作をインプリメントすることをお勧めします。

`hls::stream` クラスの詳細は、[HLS ストリーム ライブラリ](#)を参照してください。

インターフェイスの指定

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。ポートのインターフェイス モードを指定するには、次の図のように [Directive] タブでポートを右クリックして [Insert Directive] をクリックし、[Vivado HLS Directive Editor] ダイアログ ボックスを開きます。

図 43: ポート インターフェイスの指定



[Vivado HLS Directive Editor] ダイアログ ボックスでは次のオプションを設定します。

- [mode]

ドロップダウン リストからインターフェイス モードを選択します。

- [register]

すべての値渡し読み出しを演算の最初のサイクルで実行します。出力ポートでは、[register] オプションをオンにすると出力にレジスタが付きます。[register] オプションは、デザインのどの関数にも適用できます。メモリ、FIFO、および AXI4 インターフェイスの場合は、レジスタ オプションを設定しても効果はありません。

- [depth]

テストベンチからデザインに提供されるサンプル数と、テストベンチで格納する必要がある出力値の数を指定します。いずれか数値の大きい方を使用してください。

注記: 1 つのトランザクションでポインターの読み出しまたは書き込みが複数回実行される場合は、C/RTL 協調シミュレーションのために depth オプションを使用する必要があります。このオプションは配列には必要ありませんし、hls::stream コンストラクトを使用する場合にも不用で、インターフェイスでポインターを使用する場合にのみ必要になります。

[depth] オプションの値が小さすぎると、C/RTL 協調シミュレーションが次のようなデッドロック状態になる可能性があります。

1. 入力読み出しが、テストベンチでは提供できないデータを待機中に停止する。
2. ストレージがフルであるため、データを書き込もうとすると、出力の書き込みが停止する。

- [port]

このオプションは必須です。デフォルトでは、Vivado HLS はポートにレジスタは付けません。

注記: ブロック レベルの I/O プロトコルを指定するには、Vivado HLS GUI で最上位関数を選択し、その関数の戻り値としてポートを指定します。

- [offset]

このオプションは AXI4 インターフェイスの場合に使用します。

インターフェイス コンフィギュレーションは、[Solution]→[Solution Settings]→[General]→[config_interface] から設定できます。コンフィギュレーション設定を使用して次を実行します。

- 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。
- 構造体の要素によって作成されたがデザインで使用されないポートなど、未接続のポートを削除します。
- すべてのグローバル 変数に対して RTL ポートを作成します。

どの C 関数でもグローバル変数 (関数のスコープ外で定義) を使用できます。デフォルトではグローバル変数で RTL ポートが作成されることはありません。Vivado HLS では、変数は最終デザイン内にあると想定されます。Vivado HLS では、config_interface コンフィギュレーション設定で expose_global を選択すると、グローバル変数に対してポートが作成されるようになります。

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 ポートは SC_MODULE インターフェイスですべて記述し、ポートの動作はソース コードにすべて記述する必要があります。インターフェイス合成は、次をサポートするために提供されています。

- メモリ ブロック RAM インターフェイス
- AXI4-Stream インターフェイス
- AXI4-Lite インターフェイス
- AXI4 マスター インターフェイス

SystemC のインターフェイス合成を実行するプロセスは、同じインターフェイスを C または C++ デザインに追加するプロセスとは異なります。

- メモリのブロック RAM インターフェイスと AXI4 マスター インターフェイスでは、SystemC データ ポートが Vivado HLS ポートに置換される必要があります。
- AXI4-Stream および AXI4-Lite スレーブ インターフェイスには指示子のみが必要ですが、SystemC デザインに指示子を追加する場合は、別のプロセスがあります。

SystemC でのインターフェイス指示子の適用

指示子を SystemC ソース コードにプラグマとして追加する際に、ポートが SC_MODULE 宣言で指定されていると、プラグマ指示子は追加できないので、SC_MODULE で呼び出された関数内に追加する必要があります。

GUI を使用して指示子を追加する場合:

- C ソース コードと [Directives] タブを開きます。
- 指示子を必要とする関数を選択します。
- 右クリックして関数に INTERFACE 指示子を指定します。

指示子は、SC_MODULE のどの関数にでも適用できますが、ここでは変数が使用される関数に追加します。

ブロック RAM メモリ ポート

SystemC デザインのインターフェイスに配列ポートが含まれているとします。

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

my_array ポートは、ブロック RAM インターフェイス ポートではなく、内部ブロック RAM に合成されます。

Vivado HLS のヘッダー ファイル ap_mem_if.h を含めると、同じポートが ap_mem_port<data_width, address_bits> ポートとして指定されます。ap_mem_port データ型は、指定したデータとアドレス バス幅で ap_memory ポート プロトコルを使用して標準ブロック RAM インターフェイスに合成されます。

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

ap_mem_port が SystemC デザインに追加されると、関連する ap_mem_chn を SystemC テストベンチに追加して ap_mem_port を駆動する必要があります。テストベンチでは、ap_mem_chn が定義され、次に示すインスタンスに添付されています。

```
#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 のインストール ディレクトリ内の include ディレクトリに含まれ、シミュレーションが Vivado HLS 外で実行される場合は必ず含める必要があります。

SystemC の AXI4-Stream インターフェイス

AXI4-Stream インターフェイスは、sc_fifo_in または sc_fifo_out 型の SystemC ポートに追加できます。次は、典型的な SystemC デザインの最上位です。典型的な例なので、SC_MODULE とポートはヘッダー ファイルで定義されています。

```
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);
}
};
```

AXI4-Stream インターフェイスを作成するには、RESOURCE 指示子を使用してポートが AXI4-Stream リソースに接続されるように指定する必要があります。たとえば、上記のインターフェイスの場合、指示子は SC_MODULE で呼び出される関数に追加され、din および dout ポートは AXI4 Stream リソースが含まれるように指定されています。

```
#include "sc_FIFO_port.h"

void sc_FIFO_port::Prc1()
{
    //Initialization
    write_done = false;

    wait();
    while(true)
    {
        while (!start.read()) wait();
        write_done = false;

        for(int i=0;i<100; i++)
            share_mem[i] = i;

        write_done = true;
        wait();
    } //end of while(true)
}

void sc_FIFO_port::Prc2()
{
    #pragma HLS resource core=AXI4Stream variable=din
    #pragma HLS resource core=AXI4Stream variable=dout
    //Initialization
    done = false;

    wait();


    while(true)
    {
        while (!start.read()) wait();
        wait();
        while (!write_done) wait();
        for(int i=0;i<100; i++)
        {
            dout.write(share_mem[i]+din.read());
```

```

    }

    done = true;
    wait();
} //end of while(true)
}

```

SystemC デザインが合成されると、標準的な RTL FIFO ポートを含んだ RTL デザインになります。[Export RTL] ツールバー ボタン  を使用してデザインが IP としてパッケージされると、出力は AXI4-Stream インターフェイスを使用したデザインになります。

SystemC の AXI4-Lite インターフェイス

AXI4-Lite スレーブ インターフェイスは、`sc_in` または `sc_out` 型の SystemC ポートに追加できます。次の例は、典型的な SystemC デザインの最上位を示しています。この場合、典型的な例なので、`SC_MODULE` とポートはヘッダー ファイルで定義されています。

```

SC_MODULE(sc_sequ_ctypead){
    //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_ctypead){

    //Process Registration
    SC_CTHREAD(accum,clk.pos());
    reset_signal_is(reset,true);
    }
};

```

AXI4-Lite インターフェイスを作成するには、`RESOURCE` 指示子を使用してポートが AXI4-Lite リソースに接続されるように指定する必要があります。たとえば、上記のインターフェイスの場合、次の例のように `start`、`a`、`en`、`sum` および `vld` が同じ AXI4-Lite インターフェイスの `slv0` にまとめられており、すべてのポートが同じ `bus_bundle` 名で指定されて、同じ AXI4-Lite インターフェイスにまとめられています。

```

#include "sc_sequ_ctypead.h"


void sc_sequ_ctypead::accum(){
    //Group ports into AXI4 slave slv0
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0"
    variable=start
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=en
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=sum
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=vld
}

```

```
//Initialization
acc=0;
sum.write(0);
vld.write(false);
wait();

// Process the data
while(true) {
// Wait for start
wait();
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);
}
}
```

SystemC デザインが合成されると、標準的な RTL ポートを含んだ RTL デザインになります。[Export RTL] ツールバー ボタン  を使用してデザインが IP としてパッケージされると、出力は AXI4-Lite インターフェイスを使用したデザインになります。

SystemC の AXI4 マスター インターフェイス

ほとんどの標準的な SystemC デザインでは、Vivado HLS の `ap_bus` I/O プロトコルのビヘイビアーを使用してポートを指定する必要はありませんが、デザインに AXI4 マスター バス インターフェイスが必要な場合は、`ap_bus` I/O プロトコルが必要です。

SystemC デザインに AXI4 マスター インターフェイスを指定するには、次の手順に従ってください。

- Vivado HLS の `AXI4M_bus_port` 型を使用してインターフェイスを `ap_bus` I/O プロトコルで作成します。
- `AXI4M` リソースをポートに割り当てます。

次の例は、`bus_if` という `AXI4M_bus_port` を SystemC デザインに追加するところを示しています。

- ヘッダー ファイル `AXI4_if.h` をデザインに追加する必要があります。
- ポートは `AXI4M_bus_port<type>` として定義し、`type` には使用するデータ型 (この例の場合は `sc_fixed` 型) を指定します。

注記: `AXI4M_bus_port` に使用されるデータ型は、8 ビットの倍数である必要があります。また、`struct` はこのデータ型ではサポートされません。

```
#include "systemc.h"
#include "AXI4_if.h"
#include "tlm.h"
using namespace tlm;

#define DT sc_fixed<32, 8>

SC_MODULE(dut)
{
```

```
//Ports
sc_in<bool> clock; //clock input
sc_in<bool> reset;
sc_in<bool> start;
sc_out<int> dout;
AXI4M_bus_port<sc_fixed<32, 8> > bus_if;

//Variables

//Constructor
SC_CTOR(dut)
//:bus_if ("bus_if")
{
    //Process Registration
    SC_CTHREAD(P1,clock.pos());
    reset_signal_is(reset,true);
}
}
```

次に、変数 `bus_if` が SystemC 関数でどのようにアクセスされ、標準またはバースト読み出しおよび書き込みが生成されるのかを示します。

```
//Process Declaration
void P1() {
    //Initialization
    dout.write(10);
    int addr = 10;
    DT tmp[10];
    wait();
    while(1) {
        tmp[0]=10;
        tmp[1]=11;
        tmp[2]=12;

        while (!start.read()) wait();
        // Port read
        tmp[0] = bus_if->read(addr);

        // Port burst read
        bus_if->burst_read(addr,2,tmp);

        // Port write
        bus_if->write(addr, tmp);

        // Port burst write
        bus_if->burst_write(addr,2,tmp);

        dout.write(tmp[0].to_int());
        addr+=2;
        wait();
    }
}
```

AXI4M_bus_port ポート クラスがデザインで使用される場合、次に示すように、テストベンチにそれと一致する HLS バス インターフェイス チャネルの hls_bus_chn<start_addr > を含める必要があります。

```
#include <systemc.h>
#include "tlm.h"
using namespace tlm;

#include "hls_bus_if.h"
#include "AE_clock.h"
#include "driver.h"
#ifdef __RTL_SIMULATION__
#include "dut_rtl_wrapper.h"
#define dut dut_rtl_wrapper
#else
#include "dut.h"
#endif

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);

    // hls_bus_chan<type>
    // bus_variable("name", start_addr, end_addr)
    //
    hls_bus_chn<sc_fixed<32, 8> > bus_mem("bus_mem",0,1024);

    sc_signal<bool>          s_clk;
    sc_signal<bool>          reset;
    sc_signal<bool>          start;
    sc_signal<int>           dout;

    AE_Clock    U_AE_Clock("U_AE_Clock", 10);
    dut          U_dut("U_dut");
    driver        U_driver("U_driver");

    U_AE_Clock.reset(reset);
    U_AE_Clock.clk(s_clk);

    U_dut.clock(s_clk);
    U_dut.reset(reset);
    U_dut.start(start);
    U_dut.dout(dout);
    U_dut.bus_if(bus_mem);

    U_driver.clk(s_clk);
    U_driver.start(start);
    U_driver.dout(dout);

    int end_time = 8000;

    cout << "INFO: Simulating " << endl;
```

```
// start simulation
sc_start(end_time, SC_NS);

return U_driver.ret;
};
```

合成された RTL デザインには、ap_bus I/O プロトコルを使用したインターフェイスが含まれます。

AXI4M_bus_port クラスが使用されると、ap_bus インターフェイスを使用した RTL デザインになります。[Export RTL] を使用してデザインが IP としてパッケージされると、出力は AXI4 マスター ポートのデザインになります。

AXI4 インターフェイスの使用

AXI4-Stream インターフェイス

AXI4-Stream インターフェイスは、どの入力引数でも、どの配列またはポインター出力引数にでも使用できます。AXI4-Stream インターフェイスはデータをシーケンシャル ストリーミングで送信するので、読み込みと書き出しの両方を実行する引数とは併用できません。AXI4-Stream インターフェイスは常に次のバイトに符号拡張されます。たとえば、12 ビットのデータ値は 16 ビットに符号拡張されます。

複数の HLS IP ブロックと AXI4-Stream インターフェイスがデザインに統合される場合に組み合わせフィードバックパスが作成されないようにするため、AXI4-Stream インターフェイスは常にレジスタ付きインターフェイスとしてインプリメントされます。AXI-Stream インターフェイスには、AXI-Stream インターフェイスのレジスタのインプリメント方法を制御する 4 つのレジスタ モードが含まれます。

- Forward: TDATA および TVALID 信号のみにレジスタが付きます。
- Reverse: TREADY 信号のみがレジスタに送信されます。
- Both: すべての信号 (TDATA, TREADY および TVALID) にレジスタが付きます。これがデフォルトです。
- Off: どのポート信号もレジスタに送信されません。

AXI-Stream サイドチャンネル信号はデータ信号と考慮され、TDATA にレジスタが付けられるとレジスタが付けられます。



推奨: HLS で生成した IP ブロックを AXI4-Stream インターフェイスに接続する際は、少なくとも 1 つのインターフェイスをレジスタ付きのインターフェイスとしてインプリメントするか、ブロックを AXI4-Stream Register Slice を介して接続する必要があります。

デザインで AXI4-Stream を使用するには、基本的に次の 2 つの方法があります。

- AXI-Stream をサイドチャンネルなしで使用。
- AXI-Stream をサイドチャンネルありで使用。

この 2 つ目のユース ケースでは、AXI4-Stream 規格の一部であるオプションのサイドチャンネルを C コードで直接使用することも可能です。

サイドチャネルなしの AXI4-Stream インターフェイス

AXI4-Stream は、関数引数に AXI4 サイドチャネル エLEMENT が含まれない場合は、サイドチャネルなしで使用されます。次の例は、データ型が標準 C の `int` 型のデザインを示しています。この例では、両方のインターフェイスが AXI4-Stream を使用してインプリメントされます。

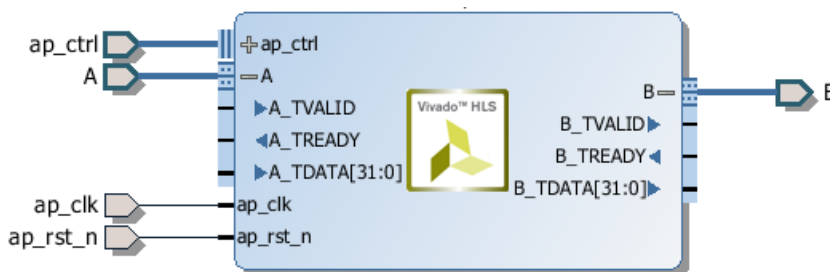
```
void example(int A[50], int B[50]) {
//Set the HLS native interface types
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

    int i;

    for(i = 0; i < 50; i++){
        B[i] = A[i] + 5;
    }
}
```

合成後は、次の図に示すように、どちらの引数もデータ ポートと標準 AXI4-Stream の TVALID および TREADY ポートを使用してインプリメントされます。

図 44: サイドチャネルなしの AXI4-Stream インターフェイス



構造体 (struct) と `DATA_PACK` 指示子を使用すると、複数の変数を同じ AXI4-Stream インターフェイスにまとめることができます。最上位関数への引数が構造体の場合、Vivado HLS では構造体がデフォルトで個別の要素分割され、構造体の各メンバーが個別のポートとしてインプリメントされます。ただし、`DATA_PACK` 指示子を使用すると、構造体の要素を単一幅のベクターにパックでき、構造体のすべての要素が同じ AXI4-Stream インターフェイスにインプリメントされるようになります。

サイドチャネルありの AXI4-Stream インターフェイス

サイドチャネルは、AXI4-Stream 規格の一部であるオプションの信号です。サイドチャネル信号は、構造体のメンバー要素が AXI4-Stream のサイドチャネル信号の名前と一致していれば、C コードで構造体を使用して直接参照および制御できます。AXI4-Stream サイドチャネル信号はデータ信号で、TDATA がレジスタに入力されるとレジスタに入力されます。この例は、Vivado HLS に含まれます。Vivado HLS の `include` ディレクトリには `ap_axi_sdata.h` ファイルが含まれます。次の構造体が含まれます。

```
#include "ap_int.h"
#include "ap_axi_sdata.h"

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;
};

```

どちらの構造体にも最上位メンバーとして、オプションの AXI4-Stream サイドチャネル信号の名前と一致する変数が含まれています。構造体にこれらの名前の要素が含まれる場合は、提供されているヘッダー ファイルを使用する必要はありません。ユーザー定義の構造体を作成することもできます。上記の構造体では `ap_int` 型とテンプレートが使用されているので、このヘッダー ファイルは C++ デザインでのみ使用できます。

注記: `valid` および `ready` 信号は、AXI4-Stream では必須の信号なので、Vivado HLS で常にインプリメントされます。これらは、構造体を使用して制御できません。

次の例は、サイドチャネルを C コードで直接使用する方法とインターフェイスへのインプリメント方法を示しています。この例は、符号付き 32 ビットデータ型を使用しています。

```

#include "ap_axi_sdata.h"

void example(ap_axis<32,2,5,6> A[50], ap_axis<32,2,5,6> B[50]){
//Map ports to Vivado HLS interfaces
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

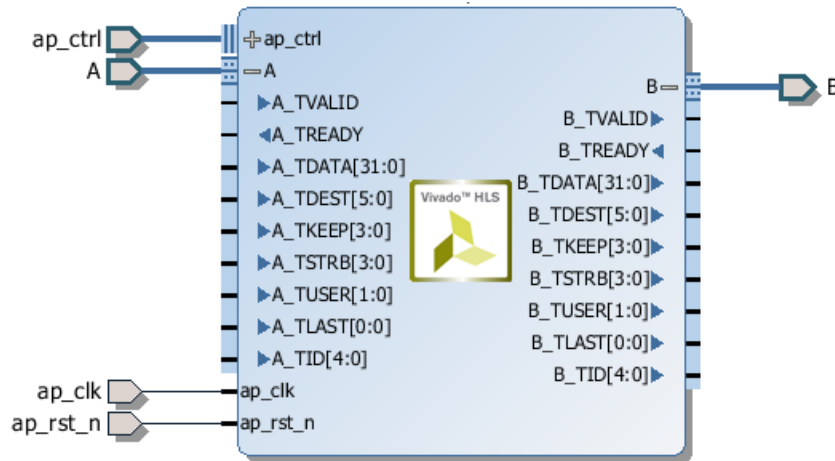
    int i;

    for(i = 0; i < 50; i++){
        B[i].data = A[i].data.to_int() + 5;
        B[i].keep = A[i].keep;
        B[i].strb = A[i].strb;
        B[i].user = A[i].user;
        B[i].last = A[i].last;
        B[i].id = A[i].id;
        B[i].dest = A[i].dest;
    }
}

```

合成後は、どちらの引数もデータポートと標準 AXI4-Stream の TVALID および TREADY プロトコルポート、および構造体で記述したオプションのポートすべてを使用してインプリメントされます。

図 45: サイドチャンネルありの AXI4-Stream インターフェイス



構造体の AXI4-Stream インターフェイスへのパック

AXI4-Stream インターフェイスと一緒に構造体を使用する場合は、デフォルトの合成動作が異なります。構造体のデフォルトの合成動作については、[インターフェイス合成および構造体](#)を参照してください。

サイドチャンネルなしで AXI4-Stream インターフェイスを使用する場合に、関数引数が構造体の場合は、次のようになります。

- Vivado HLS で自動的に `DATA_PACK` 指示子が適用され、構造体のすべての要素が単一幅のデータベクターにまとめられます。インターフェイスは、関連する `TVALID` および `TREADY` 信号を使用して単一幅のデータベクターとしてインプリメントされます。
- `DATA_PACK` 指示子を構造体に手動で適用する場合、その構造体の要素はすべて単一幅のデータベクターにまとめられ、`DATA_PACK` 指示子に AXI アライメント オプションが適用されます。インターフェイスは、関連する `TVALID` および `TREADY` 信号を使用して単一幅のデータベクターとしてインプリメントされます。

サイドチャンネルと AXI4-Stream インターフェイスを使用する場合、関数引数自体が構造体 (AXI-Stream struct) で、データ構造体 (データ自体が構造体) とサイドチャンネル信号と一緒に含めることができます。

- Vivado HLS では自動的にデータ構造体に対して `DATA_PACK` 指示子が適用され、そのデータ構造体のすべての要素が単一幅のデータベクターにまとめられます。インターフェイスは、関連するサイドチャンネルの `TVALID` および `TREADY` 信号を使用して単一幅のデータベクターとしてインプリメントされます。
- `DATA_PACK` 指示子をデータ構造体に手動で適用する場合、そのデータ構造体の要素はすべて単一幅のデータベクターにまとめられ、`DATA_PACK` 指示子に AXI アライメント オプションが適用されます。インターフェイスは、関連するサイドチャンネルの `TVALID` および `TREADY` 信号を使用して単一幅のデータベクターとしてインプリメントされます。
- `DATA_PACK` 指示子が AXI-Stream 構造体に適用される場合、関数引数、データ構造体、サイドチャンネル信号が単一幅のベクターにまとめられます。インターフェイスは、`TVALID` および `TREADY` 信号を使用して単一幅のデータベクターとしてインプリメントされます。

AXI4-Lite インターフェイス

AXI4-Lite インターフェイスを使用すると、デザインを CPU やマイクロプロセッサで制御できるようになります。Vivado HLS の AXI4-Lite インターフェイスを使用すると、次が実行できるようになります。

- 複数のポートを同じ AXI4-Lite インターフェイスにまとめることができます。

- プロセッサで実行されるコードと一緒に使用できる C ドライバー ファイルが出力できます。

注記: これにより、C アプリケーション プログラム インターフェイス (API) 関数のセットが提供されるので、ソフトウェアからハードウェアを簡単に制御できるようになります。これは、デザインを IP カタログにエクスポートする場合に便利です。

次に、Vivado HLS で関数の戻り値を含む複数の引数が AXI4-Lite インターフェイスとしてインプリメントされる例を示します。各インターフェイスで `bundle` オプションに同じ名前が使用されるので、各ポートが同じ AXI4-Lite インターフェイスにまとめられます。

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=c bundle=BUS_A offset=0x0400
#pragma HLS INTERFACE ap_vld port=b

    *c += *a + *b;
}
```

注記: `bundle` オプションを使用しない場合、Vivado HLS は AXI4-Lite インターフェイスを使用して、指定されたすべての引数を同じデフォルトのバンドルにまとめ、自動的にポートに名前を付けます。

また、I/O プロトコルも AXI4-Lite インターフェイスにまとめられたポートへ割り当てることができます。上記の例の場合、Vivado HLS がポート `b` を `ap_vld` インターフェイスとしてインプリメントし、ポート `b` を AXI4-Lite インターフェイスにまとめます。これにより、AXI4-Lite インターフェイスにポート `b` データ用のレジスタ、ポート `b` が読み出されたことを承認する出力用のレジスタ、ポート `b` の入力 Valid 信号用のレジスタが含まれます。

ポート `b` が読み出されるたびに、Vivado HLS は自動的に入力有効信号をクリアにして、ロジック 0 にレジスタをリセットします。入力 Valid レジスタがロジック 1 に設定されない場合、`b` データ レジスタのデータは有効だと判断されないで、デザインが停止し、Valid レジスタの設定を待機する状態になります。



推奨: ザイリンクスでは、デザインの操作中に使用しやすいように、AXI4-Lite インターフェイスにまとめられたポートに追加の I/O プロトコルは含めないようにしておくことを勧めしています。ただし、ザイリンクスでは、AXI4-Lite インターフェイスの `return` ポートに関連するブロックレベルの I/O プロトコルは含めることを勧めしています。

配列は、`bram` インターフェイスを使用して AXI4-Lite インターフェイスに割り当てることができません。配列は、デフォルトの `ap_memory` インターフェイスを使用してのみ AXI4-Lite インターフェイスに割り当てることができます。I/O プロトコルの `ap_stable` で指定した引数も AXI4-Lite インターフェイスには割り当てることができません。

AXI4-Lite インターフェイスにまとめられた変数は関数引数であり、それ自体は C コードでデフォルト値を割り当てることができないので、AXI4-Lite インターフェイスのどのレジスタもデフォルト値を割り当てられない可能性があります。レジスタは `config_rtl` コマンドを使用するとリセットを付けてインプリメントできますが、それ以外のデフォルト値を割り当ててはできません。

デフォルトでは、Vivado HLS で AXI4-Lite インターフェイスにまとめられる各ポートのアドレスが自動的に割り当てられます。Vivado HLS からは、C ドライバー ファイルに割り当てられたアドレスが提供されます。詳細は、[C ドライバー ファイル](#) を参照してください。アドレスを明示的に定義するには、上記の例の引数 `c` で示したように、`offset` オプションを使用します。



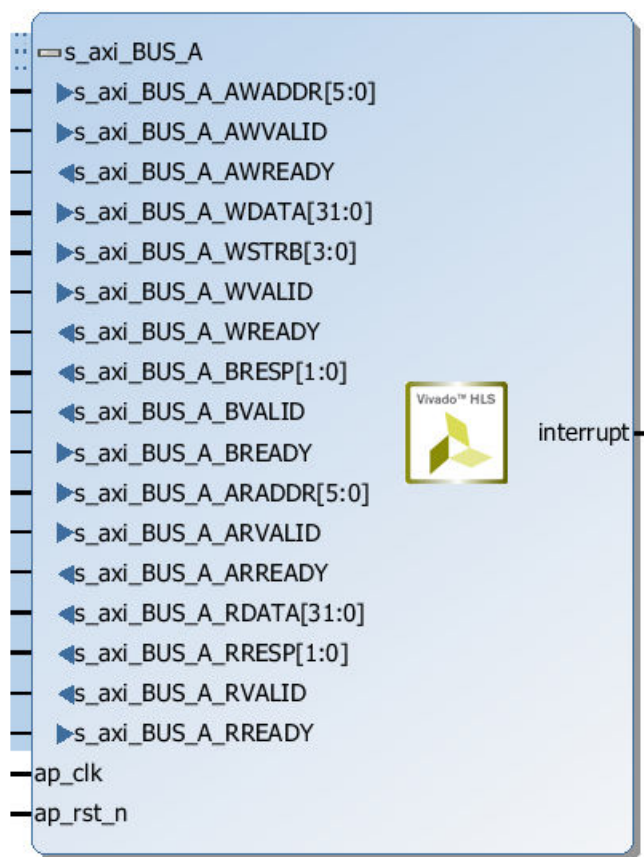
重要: AXI4-Lite インターフェイスの場合、Vivado HLS ではブロックレベル I/O プロトコル信号および割り込み制御に対してアドレス 0x0000 ~ 0x000C が予約されます。

合成後、Vivado HLS は次の図に示すように AXI4-Lite インターフェイスにポートをインプリメントします。Vivado HLS は、AXI4-Lite インターフェイスに関数リターンを含めて、割り込みポートを作成します。割り込みは、AXI4-Lite インターフェイスを使用してプログラムできます。割り込みは、次のブロック レベル プロトコルから駆動することもできます。

- `ap_done`: 関数がすべての操作を終了したことを示します。
- `ap_ready`: 関数が新しい入力データを受信する準備ができたことを示します。

インターフェイスは、C ドライバー ファイルを使用してプログラムできます。

図 46: まとめられた RTL ポートを含む AXI4-Lite スレーブ インターフェイス



AXI4-Lite インターフェイスの制御クロックおよびリセット

デフォルトでは、Vivado HLS では AXI4-Lite インターフェイスと合成済みデザインに同じクロックが使用されます。また、Vivado HLS では、AXI4-Lite インターフェイスのすべてのレジスタが合成されたロジック (`ap_clk`) に使用されるクロックに接続されます。

INTERFACE 指示子に `clock` オプションを使用すると、AXI4-Lite ポートごとに別々のクロックを指定できます (オプション)。クロックを AXI4-Lite インターフェイスに接続する場合は、次のプロトコルを使用する必要があります。

- AXI4-Lite インターフェイス クロックは、合成されたロジック (`ap_clk`) に使用されるクロックと同期している必要があります。つまり、どちらのクロックも同じマスター ジェネレーター クロックから派生している必要があります。

- AXI4-Lite インターフェイス クロックの周波数は、合成されたロジック (ap_clk) に使用されるクロックの周波数以下にする必要があります。

インターフェイス指示子に clock オプションを使用する場合、clock オプションは各バンドルの 1 つの関数引数に指定するだけです。Vivado HLS では、バンドル内のその他すべての関数引数が同じクロックとリセットを使用してインプリメントされます。Vivado HLS では、リセット信号が ap_rst_ の後にクロック名が付いた名前で生成されます。生成されるリセット信号は、config_rtl コマンドとは関係なくアクティブ Low になります。

次の例では、Vivado HLS で関数引数 a および b が AXI_clk1 というクロックと関連するリセット ポートと一緒に AXI4-Lite ポートにまとめられるところを示しています。

```
// Default AXI-Lite interface implemented with independent clock called
AXI_clk1
#pragma HLS interface s_axilite port=a clock=AXI_clk1
#pragma HLS interface s_axilite port=b
```

次の例では、Vivado HLS で関数引数 c および d が CTRL1 というクロックと関連するリセット ポートを使用して AXI4-Lite ポートの AXI_clk2 にまとめられるところを示しています。

```
// CTRL1 AXI-Lite bundle implemented with a separate clock (called AXI_clk2)
#pragma HLS interface s_axilite port=c bundle=CTRL1 clock=AXI_clk2
#pragma HLS interface s_axilite port=d bundle=CTRL1
```

C ドライバー ファイル

AXI4-Lite スレーブ インターフェイスがインプリメントされると、C ドライバー ファイルのセットが自動的に作成されます。これらの C ドライバー ファイルは、CPU で実行されるどのソフトウェアにも統合できる API を含んでおり、AXI4-Lite スレーブ インターフェイスを介してデバイスとの通信に使用されます。

C ドライバー ファイルは、デザインが IP カタログで IP としてパッケージされると作成されます。

ドライバー ファイルは、スタンドアロン モードと Linux モード用に作成されます。スタンドアロン モードの場合、ドライバーがその他のザイリンクス スタンドアロン ドライバーと同じように使用されます。Linux モードの場合、すべての C ファイル (.c) とヘッダー ファイル (.h) がソフトウェア プロジェクトにコピーされます。

合成では、ドライバー ファイルと API 関数に最上位関数からの名前が一部使用されます。上記の例の場合、最上位関数の名前は example です。最上位関数が DUT であれば、次の説明に使用されている「example」は「DUT」になります。ドライバー ファイルがパッケージされた IP (solution フォルダ内の impl ディレクトリ) に作成されます。

表 9: example というデザインの C ドライバー ファイル

ファイルパス	使用モード	説明
data/example.mdd	スタンドアロン	ドライバー定義ファイル。
data/example.tcl	スタンドアロン	ソフトウェアを SDK プロジェクトに統合するために SDK で使用。
src/xexample_hw.h	両方	すべての内部レジスタのアドレス オフセットを定義。
src/xexample.h	両方	API 定義
src/xexample.c	両方	標準 API インプリメンテーション
src/xexample_sinit.c	スタンドアロン	初期化 API インプリメンテーション
src/xexample_linux.c	Linux	初期化 API インプリメンテーション

表 9: example というデザインの C ドライバー ファイル (続き)

ファイルパス	使用モード	説明
src/Makefile	スタンドアロン	makefile

xexample.h ファイルでは、次の 2 つの構造体が定義されています。

- XExample_Config: IP インスタンスのコンフィギュレーション情報 (各 AXI4-Lite スレーブ インターフェイスのベース アドレス) を保持するのに使用されます。
- XExample: IP インスタンスのポインターを保持するのに使用されます。ほとんどの API ではこのインスタンス ポインターが最初の引数として認識されます。

標準 API インプリメンテーションは、xexample.c、xexample_sinit.c、xexample_linux.c ファイルで提供されており、次の操作を実行する関数が含まれます。

- デバイスを初期化
- デバイスを制御し、そのステータスをクエリ
- レジスタの読み出し/書き込み
- 割り込みの設定、監視および制御

次の表は、C ドライバー ファイルで提供される API 関数をそれぞれリストしています。

表 10: C ドライバーの API 関数

API 関数	説明
XExample_Initialize	この API は InstancePtr に値を書き込みます。その後ほかの API で使用できます。MMU がシステムで使用される場合を除き、この API を呼び出してデバイスを初期化することをお勧めします。
XExample_CfgInitialize	デバイス コンフィギュレーションを初期化します。システムで MMU が使用される場合は、この関数を呼び出す前に XDut_Config 変数のベース アドレスを仮想ベース アドレスに置換します。Linux システムでは使用できません。
XExample_LookupConfig	ID を指定してデバイスのコンフィギュレーション情報を取得します。コンフィギュレーション情報には、物理ベース アドレスが含まれます。Linux システムでは使用できません。
XExample_Release	Linux で UIO デバイスを解放します。munmap によるマップを削除します。プロセスが停止されると、マップは自動的に削除されます。Linux システムでのみ使用できます。
XExample_Start	デバイスを起動します。この関数は、デバイスの ap_start ポートをアサートします。デバイスに ap_start ポートがある場合にのみ使用できます。
XExample_IsDone	デバイスが前の実行を終了したかどうかをチェックします。この関数は、デバイスの ap_done ポートの値を返します。デバイスに ap_done ポートがある場合にのみ使用できます。
XExample_IsIdle	デバイスがアイドル ステートかどうかをチェックします。この関数は、デバイスの ap_idle ポートの値を返します。デバイスに ap_idle ポートがある場合にのみ使用できます。
XExample_IsReady	デバイスが次の入力を受信する準備ができているかどうかをチェックします。この関数は、デバイスの ap_ready ポートの値を返します。デバイスに ap_ready ポートがある場合にのみ使用できます。

表 10: C ドライバーの API 関数 (続き)

API 関数	説明
XExample_Continue	ap_continue ポートをアサートします。デバイスに ap_continue ポートがある場合にのみ使用できます。
XExample_EnableAutoRestart	デバイスでの自動再開をイネーブルにします。これが設定されると、デバイスは現在のトランザクションが終了したら、次のトランザクションを自動的に開始します。
XExample_DisableAutoRestart	自動再開をディスエーブルにします。
XExample_Set_ARG	ARG ポートに値 (top 関数のスカラー引数) を書き込みます。ARG が入力ポートの場合にのみ使用できます。
XExample_Set_ARG_vld	ARG_vld ポートをアサートします。ARG が入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XExample_Set_ARG_ack	ARG_ack ポートをアサートします。ARG が出力ポートで、ap_hs または ap_ack インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XExample_Get_ARG	ARG から値を読み出します。ARG ポートがデバイスの出力ポートの場合にのみ使用できます。
XExample_Get_ARG_vld	ARG_vld から値を読み出します。ARG がデバイスの出力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XExample_Get_ARG_ack	ARG_ack から値を読み出します。ARG がデバイスの入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XExample_Get_ARG_BaseAddress	インターフェイス内の配列のベース アドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XExample_Get_ARG_HighAddress	配列の一番上位のアドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XExample_Get_ARG_TotalBytes	配列を格納するのに使用されるバイトの合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、複数の要素が 32 ビットデータ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XExample_Get_ARG_BitWidth	配列内の各要素のビット幅を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、複数の要素が 32 ビットデータ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XExample_Get_ARG_Depth	配列内の要素の合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、Vivado HLS により複数の要素が 32 ビットデータ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XExample_Write_ARG_Words	32 ビット ワード長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

表 10: C ドライバーの API 関数 (続き)

API 関数	説明
XExample_Read_ARG_Words	配列から 32 ビット ワードの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XExample_Write_ARG_Bytes	バイト長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XExample_Read_ARG_Bytes	配列からバイトの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと読み込まれるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XExample_InterruptGlobalEnable	割り込み出力をイネーブルにします。割り込み関数は、ap_start がある場合にのみ使用できます。
XExample_InterruptGlobalDisable	割り込み出力をディスエーブルにします。
XExample_InterruptEnable	割り込みソースをイネーブルにします。最大で 2 つの割り込みソース (ap_done に source 0、ap_ready に source 1) がある可能性があります。
XExample_InterruptDisable	割り込みソースをディスエーブルにします。
XExample_InterruptClear	割り込みステータスをクリアにします。
XExample_InterruptGetEnabled	割り込みソースがイネーブルかどうかをチェックします。
XExample_InterruptGetStatus	割り込みソースがトリガーされたかどうかをチェックします。



重要: C ドライバー API は、常に符号なしの 32 ビット型 (U32) を使用します。C コードのデータを予測されるデータ型に変換しておく必要のあることもあります。

C ドライバー ファイルと float 型

C ドライバー ファイルは、常にデータ転送に 32 ビット符号なし整数 (U32) を使用します。次の例では、関数で float 型の引数 a と r1 が使用されています。a の値が設定され、r1 の値が返されます。

```
float caculate(float a, float *r1)
{
    #pragma HLS INTERFACE ap_vld register port=r1
    #pragma HLS INTERFACE s_axilite port=a
    #pragma HLS INTERFACE s_axilite port=r1
    #pragma HLS INTERFACE s_axilite port=return

    *r1 = 0.5f*a;
    return (a>0);
}
```


合成後、Vivado HLS ですべてのポートがデフォルトの AXI4-Lite インターフェイスにまとめられ、C ドライバー ファイルが作成されますが、次の図に示すように、ドライバー ファイルでは U32 型が使用されています。

```
// API to set the value of A
void XCaculate_SetA(XCaculate *InstancePtr, u32 Data) {
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    XCaculate_WriteReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_A_DATA, Data);
}

// API to get the value of R1
u32 XCaculate_GetR1(XCaculate *InstancePtr) {
    u32 Data;

    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    Data = XCaculate_ReadReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_R1_DATA);
    return Data;
}
```

これらの関数が float 型で直接使用できる場合は、書き込み値および読み出し値が予測される float 型と一貫しません。これらの関数をソフトウェアで使用する場合は、次を使用して型を変換します。

```
float a=3.0f,r1;
u32 ua,url;

// cast float "a" to type U32
XCaculate_SetA(&calculate,*((u32*)&a));
url=XCaculate_GetR1(&calculate);

// cast return type U32 to float type for "r1"
r1=*((float*)&url);
```

ハードウェアの制御

ハードウェア ヘッダー ファイル (この例の場合 xexample_hw.h) には、AXI4-Lite スレーブ インターフェイスにまとめられたポートのメモリ マップド ロケーションのリストが含まれます。

```
// 0x00 : Control signals
//      bit 0 - ap_start (Read/Write/SC)
//      bit 1 - ap_done (Read/COR)
//      bit 2 - ap_idle (Read)
//      bit 3 - ap_ready (Read)
//      bit 7 - auto_restart (Read/Write)
//      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)
//      bit 1 - Channel 1 (ap_ready)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//      bit 0 - Channel 0 (ap_done)
//      others - reserved
```

```
// 0x10 : Data signal of a
//         bit 7~0 - a[7:0] (Read/Write)
//         others - reserved
// 0x14 : reserved
// 0x18 : Data signal of b
//         bit 7~0 - b[7:0] (Read/Write)
//         others - reserved
// 0x1c : reserved
// 0x20 : Data signal of c_i
//         bit 7~0 - c_i[7:0] (Read/Write)
//         others - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//         bit 7~0 - c_o[7:0] (Read)
//         others - reserved
// 0x2c : Control signal of c_o
//         bit 0 - c_o_ap_vld (Read/COR)
//         others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
// Clear on
// Handshake)
```

AXI4-Lite スレーブ インターフェイスのレジスタを正しくプログラムするには、ハードウェア ポートがどのように動作するかを理解しておく必要があります。インターフェイス合成で説明したように、ブロックは同じポート プロトコルで操作されます。

たとえば、ブロック操作を開始するには `ap_start` レジスタを 1 に設定する必要があります。次にデバイスがレジスタから AXI4-Lite スレーブ インターフェイスにまとめられた入力を読み出します。ブロックの演算が終了すると、ハードウェアの出力ポートにより `ap_done`、`ap_idle`、および `ap_ready` が設定され、AXI4-Lite スレーブ インターフェイスにまとめられた出力ポートの結果が適切なレジスタから読み出されます。

上記の例の関数引数 `c` のインプリメンテーションは、ハードウェア ポートの動作をある程度理解しておくことが重要であることも示しています。関数引数 `c` は読み出しおよび書き込みの両方に使用されるので、インターフェイス合成に示すように、入力ポート `c_i` および出力ポート `c_o` として別々にインプリメントされます。

AXI4-Lite スレーブ インターフェイスをプログラムするには、関数を 1 回だけ実行するフローがまず推奨されます。

- 割り込み関数を使用して、割り込みの動作方法を決定します。
- ブロックの入力ポートにレジスタ値を読み込みます。上記の例では、これは API 関数の `XExample_Set_a`、`XExample_Set_b` および `XExample_Set_c_i` を使用して実行されます。
- `XExample_Start` を使用して `ap_start` ビットを 1 に設定し、関数を開始します。このレジスタには、上記のヘッダー ファイルに記述されているように、セルフ クリーニング機能があります。トランザクションが 1 つ終了すると、ブロックは動作を一時停止します。
- 関数が実行されます。生成された割り込みが処理されます。
- 出力レジスタが読み出されます。上記の例では、これは API 関数の `XExample_Get_c_o_vld` (データが有効なことを確認) と `XExample_Get_c_o` を使用して実行されます。

注記: AXI4-Lite スレーブ インターフェイスのレジスタは、ポートと同じ I/O プロトコルに従います。この場合、出力 `valid` がロジック 1 に設定されて、データが有効かどうかを示されます。

- 次のトランザクションでも繰り返します。

2 つ目に推奨されるフローは、ブロックの継続実行です。このモードでは、AXI4-Lite スレーブ インターフェイスに含まれる入力ポートは、コンフィギュレーションを実行するポートのみである必要があります。ブロックは通常、CPU よりもかなり高速で実行されます。ブロックが入力を待つ必要がある場合、ブロックはほとんどの時間を待機に費やすことになります。

- 割り込み関数を使用して、割り込みの動作方法を決定します。
- ブロックの入力ポートにレジスタ値を読み込みます。上記の例では、これは API 関数の `XExample_Set_a`、`XExample_Set_a` および `XExample_Set_c_i` を使用して実行されます。
- API `XExample_EnableAutoRestart` を使用して自動開始関数を設定します。
- 関数が実行されます。各ポートの I/O プロトコルにより、ブロックを介して処理されるデータが同期されます。
- 生成された割り込みが処理されます。出力レジスタにはこの操作中アクセスできますが、データが頻繁に変更される可能性があります。
- API 関数 `XExample_DisableAutoRestart` を使用して、これ以上実行されないようにします。
- 出力レジスタが読み出されます。上記の例では、これは API 関数の `XExample_Get_c_o` および `XExample_Set_c_o_vld` を使用して実行されます。

ソフトウェアの制御

API 関数は、CPU で実行されるソフトウェアで使用すると、ハードウェア ブロックを制御できます。プロセスの概要は次のとおりです。

- ハードウェア インスタンスのインスタンスを作成
- デバイス コンフィギュレーションを検索
- デバイスを初期化
- HLS ブロックの入力パラメーターを設定
- デバイスを開始して結果を読み出し

このプロセスを抽象化すると、次のようになります。ソフトウェア制御のすべての例は、Zynq-7000 SoC のチュートリアルに含まれます。

```
#include "xexample.h" // Device driver for HLS HW block
#include "xparameters.h"

// HLS HW instance
XExample HlsExample;
XExample_Config *ExamplePtr

int main() {
    int res_hw;

    // Look Up the device configuration
    ExamplePtr = XExample_LookupConfig(XPAR_XEXAMPLE_0_DEVICE_ID);
    if (!ExamplePtr) {
        print("ERROR: Lookup of accelerator configuration failed.\n\r");
        return XST_FAILURE;
    }

    // Initialize the Device
    status = XExample_CfgInitialize(&HlsExample, ExamplePtr);
    if (status != XST_SUCCESS) {
        print("ERROR: Could not initialize accelerator.\n\r");
        exit(-1);
    }

    //Set the input parameters of the HLS block
    XExample_Set_a(&HlsExample, 42);
    XExample_Set_b(&HlsExample, 12);
    XExample_Set_c_i(&HlsExample, 1);
```

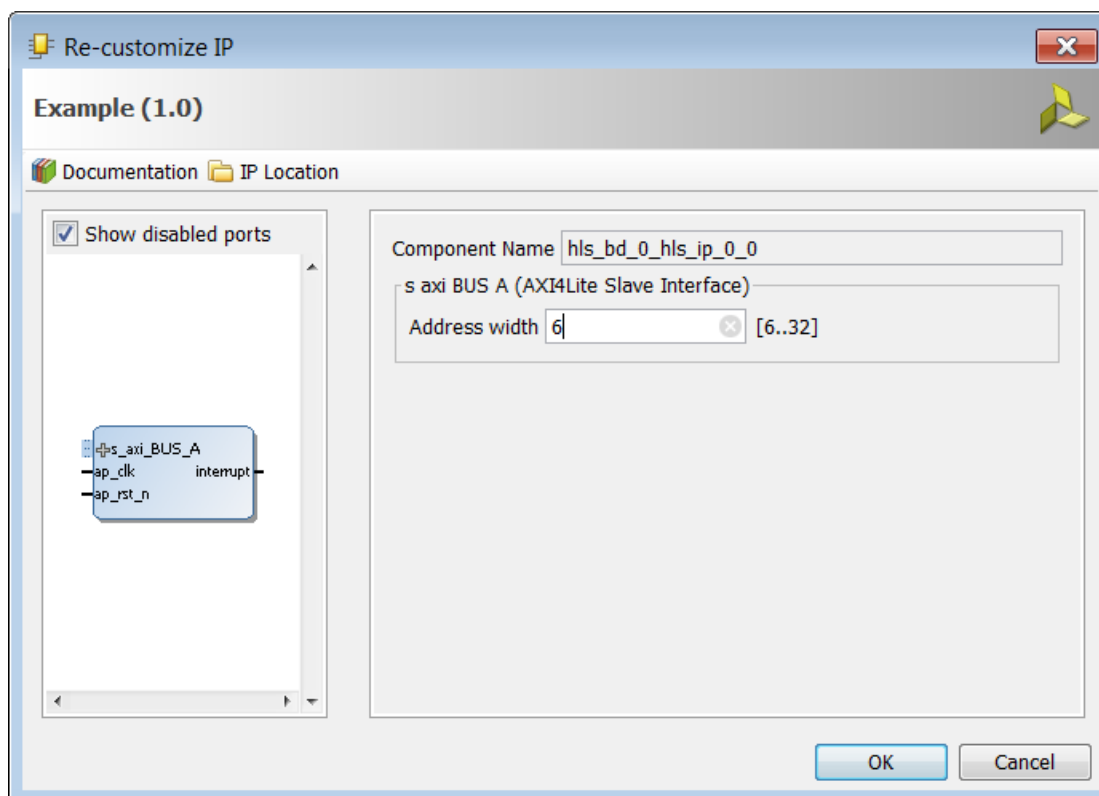
```
// Start the device and read the results
XExample_Start(&HlsExample);
do {
    res_hw = XExample_Get_c_o(&HlsExample);
} while (XExample_Get_c_o(&HlsExample) == 0); // wait for valid data output
print("Detected HLS peripheral complete. Result received.\n\r");
}
```

IP インテグレーターでの AXI4-Lite スレーブ インターフェイスのカスタマイズ

AXI4-Lite スレーブ インターフェイスを使用した HLS の RTL デザインが Vivado IP インテグレーターでのデザインに統合されると、ブロックをカスタマイズできるようになります。IP インテグレーターのブロック図で HLS ブロックを選択し、右クリックして [Customize Block] をクリックします。

アドレス幅はデフォルトで必要最小限のサイズに設定されます。これを変更して、アドレス サイズが 32 ビット未満のブロックに接続します。

図 47: IP インテグレーターでの AXI4-Lite スレーブ インターフェイスのカスタマイズ



AXI4 マスター インターフェイス

AXI4 マスター インターフェイスは配列またはポインター/リファレンス引数で使用でき、Vivado HLS では次のいずれかのモードでインプリメントされます。

- 個別のデータ転送
- バースト モードのデータ転送

個別のデータ転送では、Vivado HLS で各アドレスのデータの 1 つの要素が読み出されるか書き込まれます。次の例に、1 つの読み出しおよび 1 つの書き込み操作を示します。Vivado HLS で AXI インターフェイスにアドレスが生成され、1 つのデータ値とアドレスが読み出され、1 つのデータ値が書き込まれます。インターフェイスは、アドレスごとに 1 つのデータ値を転送します。

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

    acc += *d;
    *d = acc;
}
```

バースト モード転送では、Vivado HLS で 1 つのベース アドレスを使用してデータが読み出したり書き込まれたり、その後複数のシーケンシャル データ サンプルが続くので、より高いデータ スループットを達成できます。バースト モードは、C の `memcpy` 関数を使用したり、`for` ループをパイプライン処理する場合に使用できます。

注記: C の `memcpy` 関数は、AXI4 マスター インターフェイスで指定された最上位関数の引数のデータ転送に使用された場合にのみ、合成でサポートされます。

次の例に、`memcpy` 関数を使用したバースト モードのコピーを示します。最上位関数の引数 `a` は、AXI4 マスター インターフェイスとして指定されています。

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

//Port a is assigned to an AXI4 master interface

    int i;
    int buff[50];

    //memcpy creates a burst access to memory
    memcpy(buff,(const int*)a,50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }

    memcpy((int *)a,buff,50*sizeof(int));
}
```

この例が合成されると、次の図のようなインターフェイスになります。

注記: この図では、AXI4 インターフェイスは展開されていません。

図 48: AXI4 インターフェイス



次の例は前の例と同じコードですが、データ出力をコピーするのに `for` ループを使用しています。

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

//Port a is assigned to an AXI4 master interface

    int i;
    int buff[50];

    //memcpy creates a burst access to memory
    memcpy(buff,(const int*)a,50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }

    for(i=0; i < 50; i++){
#pragma HLS PIPELINE
        a[i] = buff[i];
    }
}
```

バースト読み出しまたは書き込みをインプリメントするのに `for` ループを使用する場合は、次の要件に従ってください。

- ループをパイプライン処理する
- 昇順でアドレスにアクセスする
- 条件文内にアクセスを記述しない
- 入れ子状態のループの場合、バースト動作が抑制されてしまわないよう、ループを平坦にしない

注記: ポートが別の AXI ポートにまとめられない場合、`for` ループ内で使用できるのは読み出し 1 つと書き込み 1 つのみです。次の例では、別の AXI インターフェイスを使用して、2 つの読み出しをバースト モードで実行するところを示します。

次の例では、Vivado HLS でポート読み出しがバースト転送としてインプリメントされます。ポート `a` は `bundle` オプションを使用しないで指定され、デフォルトの AXI インターフェイスにインプリメントされます。ポート `b` は該当するバンドルを使用して指定され、`d2_port` という別の AXI インターフェイスにインプリメントされます。

```
void example(volatile int *a, int *b){

#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE m_axi depth=50 port=b bundle=d2_port

    int i;
    int buff[50];

    //copy data in
    for(i=0; i < 50; i++){
#pragma HLS PIPELINE
        buff[i] = a[i] + b[i];
    }
    ...
}
```

注記: 構造体は、DATA_PACK 最適化を使用してパックされた場合、AXIM インターフェイスのみでサポートされません。

AXI4 バースト動作の制御

最適な AXI4 インターフェイスとは、バスへのアクセス待機中にデザインが停止せず、バス アクセスが承認された後の読み出し/書き込み待機中にバスが停止しないようなインターフェイスです。最適な AXI4 インターフェイスを作成するには、INTERFACE 指示子に次のオプションを使用して、バースト ビヘイビアーを指定し、AXI4 インターフェイスの効率を改善します。

これらのオプションの中には、内部ストレージを使用してデータをバッファリングするため、エリアおよびリソースへの影響があるものもあります。

- `latency`: AXI4 インターフェイスのレイテンシを指定し、読み出しまたは書き込みの指定サイクル (レイテンシ) 前にバス要求を開始できるようにします。このレイテンシの値が少なすぎる場合は、デザインの準備が早過ぎるので、バスを待つために一時停止することがあります。レイテンシの値が多すぎる場合は、バス アクセスは承認されても、デザインがアクセスを開始するのを待つためにバスが一時停止することがあります。
- `max_read_burst_length`: バースト転送中に読み出されるデータ値の最大数を指定します。
- `num_read_outstanding`: デザインが停止する前に、AXI4 バスに対して応答なしで送信できる読み出し要求の数を指定します。これにより、デザインの内部ストレージである FIFO のサイズ (`num_read_outstanding*max_read_burst_length*word_size`) が決まります。
- `max_write_burst_length`: バースト転送中に書き込まれるデータ値の最大数を指定します。
- `num_write_outstanding`: デザインが停止する前に、AXI4 バスに対して応答なしで送信できる書き込み要求の数を指定します。これにより、デザインの内部ストレージである FIFO のサイズ (`num_read_outstanding*max_read_burst_length*word_size`) が決まります。

次に、これらのオプションを使用した例を示します。

```
#pragma HLS interface m_axi port=input offset=slave bundle=gmem0
depth=1024*1024*16/(512/8)
latency=100
num_read_outstanding=32
num_write_outstanding=32
max_read_burst_length=16
max_write_burst_length=16
```

インターフェイスは、レイテンシ 100 になるように指定されています。Vivado HLS では、デザインが AXI4 バスにアクセスできるようになる 100 クロック サイクル前に、バースト アクセス要求をスケジューリングしようとします。バスの効率をさらに向上するには、`num_write_outstanding` および `num_read_outstanding` オプションを使用して、デザインに最大 32 個の読み出しおよび書き込みアクセスを格納できるバッファ容量が含まれるようにします。これで、バス要求が送信されるまでデザインの処理を続行できるようになります。最後に、`max_read_burst_length` および `max_write_burst_length` オプションを使用することで、最大バースト サイズが 16 になり、AXI4 インターフェイスがそれを超える時間バスを保持しないようにします。

これらのオプションを使用すると、AXI4 インターフェイスの動作をシステム用に最適化できます。動作の効率は、これらの値が正しく設定されているかどうかによって異なります。

64 ビット アドレス機能のある AXI4 インターフェイスを作成

Vivado HLS では、デフォルトで 32 ビット バスの AXI4 ポートがインプリメントされますが、次のように `m_axi_addr64` インターフェイス コンフィギュレーション オプションを使用すると、64 ビット アドレス バスの AXI4 インターフェイスをインプリメントできます。

1. [Solution] → [Solution Settings] をクリックします。

2. [Solution Settings] ダイアログ ボックスで [General] カテゴリをクリックし、[Add] をクリックします。
3. [Add Command] ダイアログ ボックスで [config_interface] を選択して [m_axi_addr64] をオンにします。



重要: [m_axi_addr64] を選択する場合、Vivado HLS は 64 ビット アドレス バスでデザインに すべての AXI4 インターフェイスをインプリメントします。

AXI4 インターフェイスのアドレス オフセットの制御

デフォルトでは、AXI4 マスター インターフェイスはすべての 読み出しおよび書き込み操作をアドレス 0x00000000 から開始します。たとえば、次のコードの場合、50 個のアドレス値を表すアドレス 0x00000000 ~ 0x000000c7 (32 ビット ワード 50 個で 200 バイトを提供) からデータを読み出してから、同じアドレスにデータを再び書き込みます。

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS

    int i;
    int buff[50];

    memcpy(buff, (const int*)a, 50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }
    memcpy((int *)a, buff, 50*sizeof(int));
}
```

アドレス オフセットを使用するには、INTERFACE 指示子に `-offset` を付けて、次のいずれかを指定します。

- `off`: オフセット アドレスは使用しません。これがデフォルトです。
- `direct`: デザインにアドレス オフセットを適用するための 32 ビット ポートが追加されます。
- `slave`: AXI4-Lite インターフェイス内にアドレス オフセットを適用するための 32 ビット レジスタが追加されます。

Vivado HLS では、最終的な RTL にアドレス オフセットは AXI4 マスター インターフェイスで生成される読み出しまたは書き込みアドレスに直接適用され、システムのアドレス位置にアクセスできるようになります。

AXI インターフェイスで `slave` オプションを使用する場合は、デザイン インターフェイスに AXI4-Lite ポートを使用する必要があります。ザイリンクスでは、次のプラグマを使用して AXI4-Lite インターフェイスをインプリメントすることを勧めます。

```
#pragma HLS INTERFACE s_axilite port=return
```

また、`slave` オプションを使用しており、複数の AXI4-Lite インターフェイスを使用した場合は、AXI マスター ポートのオフセット レジスタが正しい AXI4-Lite インターフェイスにまとめられていることを確認する必要があります。次の例では、`a` ポートが、オフセットと AXI4-Lite インターフェイス (`AXI_Lite_1` および `AXI_Lite_2`) と一緒に AXI マスター インターフェイスとしてインプリメントされています。

```
#pragma HLS INTERFACE m_axi port=a depth=50 offset=slave
#pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite_1
#pragma HLS INTERFACE s_axilite port=b bundle=AXI_Lite_2
```


a ポートのオフセットレジスタが AXI_Lite_1 という AXI4-Lite インターフェイスにまとめられるようにするには、次の INTERFACE 指示子が必要です。

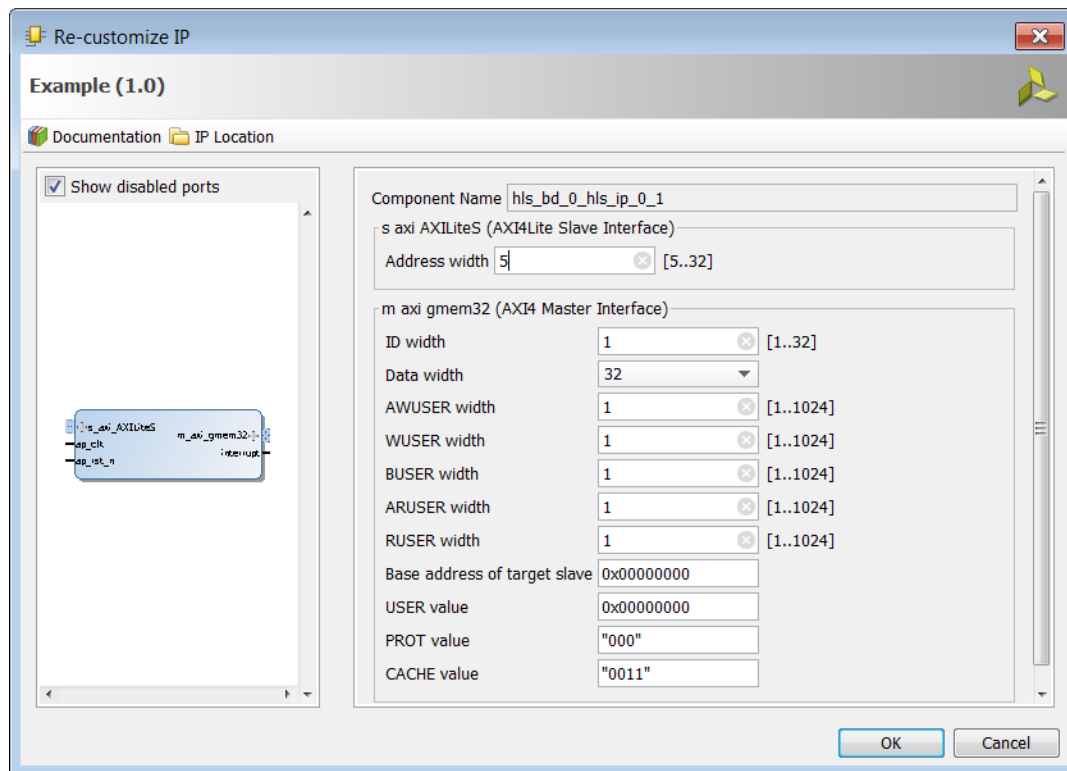
```
#pragma HLS INTERFACE s_axilite port=a bundle=AXI_Lite_1
```

IP インテグレーターでの AXI4 Master インターフェイスのカスタマイズ

AXI4 マスター インターフェイスを使用した HLS の RTL デザインを Vivado IP インテグレーターでのデザインに組み込む際には、ブロックをカスタマイズできます。IP インテグレーターでブロック図で HLS ブロックを選択し、右クリックして [Customize Block] をクリックし、設定をカスタマイズします。AXI4 パラメーターのすべての説明については、『Vivado Design Suite: AXI リファレンス ガイド』(UG1037) のこの[セクション](#)を参照してください。

次の図は、次のデザインの [Re-Customise IP] ダイアログ ボックスを示しています。このデザインには、AXI4-Lite ポートが含まれています。

図 49: IP インテグレーターでの AXI4 Master インターフェイスのカスタマイズ



SSI テクノロジ デバイスのインターフェイスの管理

一部のザイリンクス デバイスには、スタックド シリコン インターフェイス (SSI) テクノロジが使用されています。これらのデバイスでは、使用可能なリソースが複数の SLR (Super Logic Region) に分割されます。SLR 間の接続には、スーパー ロング ライン (SSL) 配線が使用されます。SSL 配線の場合、通常標準の FPGA 配線よりも遅延の発生する可能性が高くなります。デザインが最高のパフォーマンスで動作するようにするため、次のガイドラインに従ってください。

- SLR 出力と SLR 入力両方の SLR 間をまたがるすべての信号をレジスタに入力。
- I/O バッファを介して SLR に入力または SLR から出力する場合は、信号をレジスタ入力する必要なし。

- Vivado HLS で作成されたロジックが 1 つの SLR 内にフィットするようにする。

注記: ターゲット テクノロジとして SSI テクノロジ デバイスを選択する場合、リソース使用率レポートには、SLR の使用率とデバイス全体の使用率の両方が示されます。

ロジックが 1 つの SLR デバイス内に含まれる場合、Vivado で `config_interface` コマンドに `register_io` オプションを使用できます。このオプションを使用すると、すべてのブロック入力か出力、または両方ともが自動的にレジスタに入力されるようになります。このオプションは、スカラーの場合にのみ必要です。配列ポートはすべて自動的にレジスタ入力されます。

`register_io` オプションの設定は、次のとおりです。

- `off`: どの入力も出力もレジスタには送信されません。
- `scalar_in`: すべての入力がレジスタに送信されます。
- `scalar_out`: すべての出力がレジスタに送信されます。
- `scalar_all`: すべての入力および出力がレジスタに送信されます。

注記: RTL のブロック レベルのフロアプランと共に `register_io` オプションを使用すると、SSI テクノロジ デバイスをターゲットにしたロジックが最大クロック レートで実行されるようになります。

デザインの最適化

このセクションでは、Vivado HLS でパフォーマンスおよびエリア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について説明します。

次の表は、Vivado HLS で提供される最適化指示子をリストしています。

表 11: Vivado HLS の最適化指示子

指示子	説明
ALLOCATION	使用される演算、コア、または関数の数を制限します。これにより、ハードウェア リソースが共有されるので、レイテンシが増加される可能性があります。
ARRAY_MAP	複数の小型の配列を 1 つの大型の配列にまとめ、ブロック RAM リソースを削減します。
ARRAY_PARTITION	大型の配列を複数の配列または個別のレジスタに分割し、データへのアクセスを改善し、ブロック RAM のボトルネックを削除します。
ARRAY_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。多数のブロック RAM を使用せずにブロック RAM アクセスを向上するのに有益です。
CLOCK	SystemC デザインの場合、 <code>create_clock</code> コマンドを使用して複数のクロックを指定し、この指示子を使用して個々の SC_MODULE に適用できます。
DATA_PACK	構造体 (struct) のデータ フィールドをワード幅が広い 1 つのスカラーにパックします。
DATAFLOW	タスク レベルのパイプライン処理を有効にし、関数およびループが同時に実行されるようにします。スループットおよびレイテンシの最適化に使用します。
DEPENDENCE	ループ キャリー依存を克服し、ループをパイプライン処理できるようにする (またはより短い間隔でパイプラインできるようにする) 追加情報を提供します。
EXPRESSION_BALANCE	自動演算調整をオフにできます。
FUNCTION_INSTANTIATE	同じ関数の異なるインスタンスをローカルで最適化できます。
INLINE	関数をインライン展開し、このレベルの関数の階層を削除します。関数の境界を超えたロジック最適化をイネーブルにし、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/間隔を改善します。

表 11: Vivado HLS の最適化指示子 (続き)

指示子	説明
INTERFACE	関数記述から RTL ポートをどのように作成するかを指定します。
LATENCY	最小および最大レイテンシ制約を指定します。
LOOP_FLATTEN	入れ子のループを 1 つのループに展開し、レイテンシを改善します。
LOOP_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、共有を増やして最適化を向上します。
LOOP_TRIPCOUNT	範囲が可変のループに使用されます。ループの反復回数を見積もりを指定します。これは合成には影響がなく、レポートにのみ影響します。
OCCURRENCE	関数またはループをパイプライン処理する際に、あるロケーションのコードがそれを含む関数またはループのコードよりも低速で実行されることを指定します。
PIPELINE	ループまたは関数内で演算をオーバーラップできるようにして、開始間隔を削減します。
PROTOCOL	プロトコル領域になるコードの領域を指定します。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。
RESET	特定のステート変数 (グローバルまたはスタティック) のリセットを追加または削除するために使用します。
RESOURCE	変数 (配列、算術演算、関数引数) を RTL にインプリメントするのに使用するライブラリ リソース (コア) を指定します。
STREAM	データフロー最適化中に特定の配列を FIFO または RAM メモリ チャンネルとしてインプリメントするよう指定します。hls::stream を使用している場合に、STREAM 最適化指示子を使用して hls::stream の設定を無効にします。
TOP	合成の最上位関数はプロジェクト設定で指定します。この指示子は、関数を合成の最上位として指定するために使用できます。これにより、新しくプロジェクトを作成しなくても、同じプロジェクト内の別のソリューションを合成の最上位関数として指定できます。
UNROLL	for ループを展開して、ループ本体のインスタンスを複数作成し、その命令が別々にスケジュールできるようにします。

Vivado HLS には、最適化指示子だけでなく、多くのコンフィギュレーション設定が提供されています。コンフィギュレーション設定は合成のデフォルト設定を変更するために使用されます。コンフィギュレーション設定は、次の表にリストされています。

表 12: Vivado HLS のコンフィギュレーション

GUI 指示子	説明
Config Array Partition	グローバル配列を含めた配列の分割方法と、分割が配列ポートに影響するかどうかを指定します。
Config Bind	合成のバインド段階で使用するエフォート レベルを指定します。使用される演算数をグローバルに最小限に抑えるために使用します。
Config Compile	自動ループ パイプラインおよび浮動小数点の math 最適化など、合成特有の最適化を制御します。
Config Dataflow	DATAFLOW 最適化でのデフォルトのメモリ チャンネルと FIFO の深さを指定します。
Config Interface	最上位関数の引数に関連付けられていない I/O ポートが制御され、最終的な RTL から未使用のポートが削除できます。
Config RTL	ファイルおよびモジュールの命名、リセット形式、FSM エンコーディングを含めた出力 RTL を制御できます。
Config Schedule	合成のスケジューリング段階中に使用するエフォート レベルおよび出力メッセージの詳細度合いを決定します。

最適化およびコンフィギュレーションの適用方法の詳細は、[最適化指示子の適用](#)を参照してください。コンフィギュレーションは、[Solution] → [Solution Settings] → [General] から [Add] ボタンで追加できます。

最適化については、通常のデザインへの適用方法の内容に含まれます。

クロック、リセット、および RTL 出力については、一緒に説明します。クロック周波数とターゲット デバイスは、最適化を実行する最初の制約です。Vivado HLS では、各クロック サイクルヘターゲット デバイスからなるべく多くの演算を配置しようとしています。最終的な RTL で使用されるリセット形式は、FSM エンコーディング スタイルなどの設定と共に config_rtl コンフィギュレーションを使用して制御されます。

[Optimizing for Throughput] では、タスクがパイプライン処理されてパフォーマンスが改善され、タスク間のするデータフローが改善され、構造が最適化されてパフォーマンスを制限している可能性のあるアドレス問題が改善される通常の方法が実行されます。

[Optimizing for Latency] では、レイテンシ制約とループ反復の手法が使用され、終了するのに必要なクロック サイクル数が削減されます。

エリアを改善するには、演算のインプリメンテーション方法 (演算数の制御、ハードウェアでの演算のインプリメント方法など) に焦点を置いてください。

Vivado HLS には、プラグマおよび指示子以外にも既存の最適化済み RTL を HLS デザイン フローに組み込む方法があります。詳細は、[RTL ブラック ボックス](#) を参照してください。

クロック、リセット、および RTL 出力

クロック周波数の指定

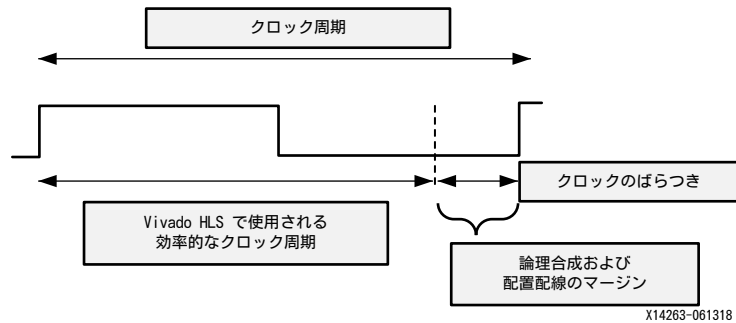
C および C++ デザインでは、クロックは 1 つしかサポートされず、デザイン内のすべての関数に、同じクロックが適用されます。

SystemC デザインでは、SC_MODULE ごと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 コマンドに -name オプションを使用して複数の名前を指定してクロックを作成し、CLOCK 指示子またはプラグマを使用して SC_MODULE を含む関数を指定し、そのクロックで合成されるようにします。各 SC_MODULE の合成に使用できるクロックは、1 つのみです。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は、複数のクロックを関数を介して分配できますが、SC_MODULE はそれぞれ 1 つのクロックにしか応答しません。

クロック周期は、[Solutions] → [Solutions Setting] で ns に指定します。Vivado HLS では、クロックのばらつきの概念を使用して、ユーザー定義のタイミング マージンが提供されています。Vivado HLS では、クロック周波数とデバイス ターゲット情報を使用して演算のタイミングが見積もられますが、最終的なコンポーネント配置およびネット配線はわかりません。これらの演算は、出力 RTL の論理合成で実行されるからです。このような場合、Vivado HLS では正確な遅延は判別できません。

Vivado HLS では、次の図に示すようにクロック周期からクロックのばらつきを差し引いて、合成に使用されるクロック周期が計算されます。

図 50: クロック周期とマージン



これにより、ユーザー指定のマージンが使用されて、論理合成や配置配線などのダウストリーム プロセスで操作を完了するのに十分なタイミング マージンが供給されます。FPGA デバイスのほとんどのリソースが使用されている場合、セルの配置とセルを接続するためのネットの配線が理想的なものにならないことがあります。タイミング遅延が予想外に大きくなってしまうことがあります。このような場合、Vivado HLS ではタイミング マージンを増加すると、各クロック サイクルにロジックが多くパックされすぎてしまうようなデザインは作成されなくなり、理想的な配置配線オプションよりもオプションが少ない場合でも RTL 合成でタイミングを満たすことができるようになります。

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% です。この値は、クロック周期とは別に明示的に指定できます。

Vivado HLS ではすべての制約 (タイミング、スループット、レイテンシ) を満たそうとしますが、制約を満たすことができない場合でも、Vivado HLS は常に RTL デザインを出力します。

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、Vivado HLS は次のような SCHED-644 というメッセージを表示し、達成可能な最良のパフォーマンスのデザインを出力します。

```
@W [SCHED-644] Max operation delay (<operation_name> 2.39ns) exceeds the
effective
cycle time
```

Vivado HLS では、特定のパスのタイミング要件が満たされない場合でも、それ以外のすべてのパスではタイミングが達成されます。これにより、ダウストリームの論理合成でエラーのあったパスに対して、より高い最適化レベルを使用したり、特別な処理を実行することで、タイミングを満たすことができるかどうかの評価できるようになっています。



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 (Vivado HLS で出力デザインが生成されたとしても、すべてのパフォーマンス制約が満たされた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。

`config_schedule` コマンドに `relax_ii_for_timing` オプションを付けると、デフォルトのタイミング ビヘイビアが変更できます。このオプションを指定した場合、クロック周期を満たすことができないパスが検出されると、Vivado HLS でパイプライン指示子の開始間隔 (II) が自動的に緩められます。このオプションは、PIPELINE 指示子に II 値が付いていない (II=1 になる) 場合にのみ使用できます。II 値を PIPELINE 指示子に指定してある場合は、`relax_ii_for_timing` オプションを使用しても何も起こりません。

合成が終了すると、デザイン レポートが階層の各関数ごとに生成され、solution のレポート フォルダで確認できるようになります。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ーをターゲットする前に、レイテンシの指定方法およびインプリメンテーション コアを指定する方法を見直してみてください。

リセットの指定

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

初期化動作

C では、static 修飾子で定義された変数およびグローバルに定義された変数はデフォルトで 0 に初期化されますが、オプションで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) で割り当てられ、その後再び割り当てられることはありません。どちらの場合も同じ初期値が RTL にインプリメントされます。

- RTL シミュレーション中、変数は C コードと同じ値で初期化されます。
- 同じ変数は、FPGA をプログラムするために使用されるビットストリームで初期化されます。デバイスが電源投入されると、変数が初期化状態で開始されます。

変数は、C コードと同じ初期状態で開始しますが、この初期状態に強制的に戻す方法はありません。これらの初期状態に戻すには、変数をリセットと一緒にインプリメントする必要があります。

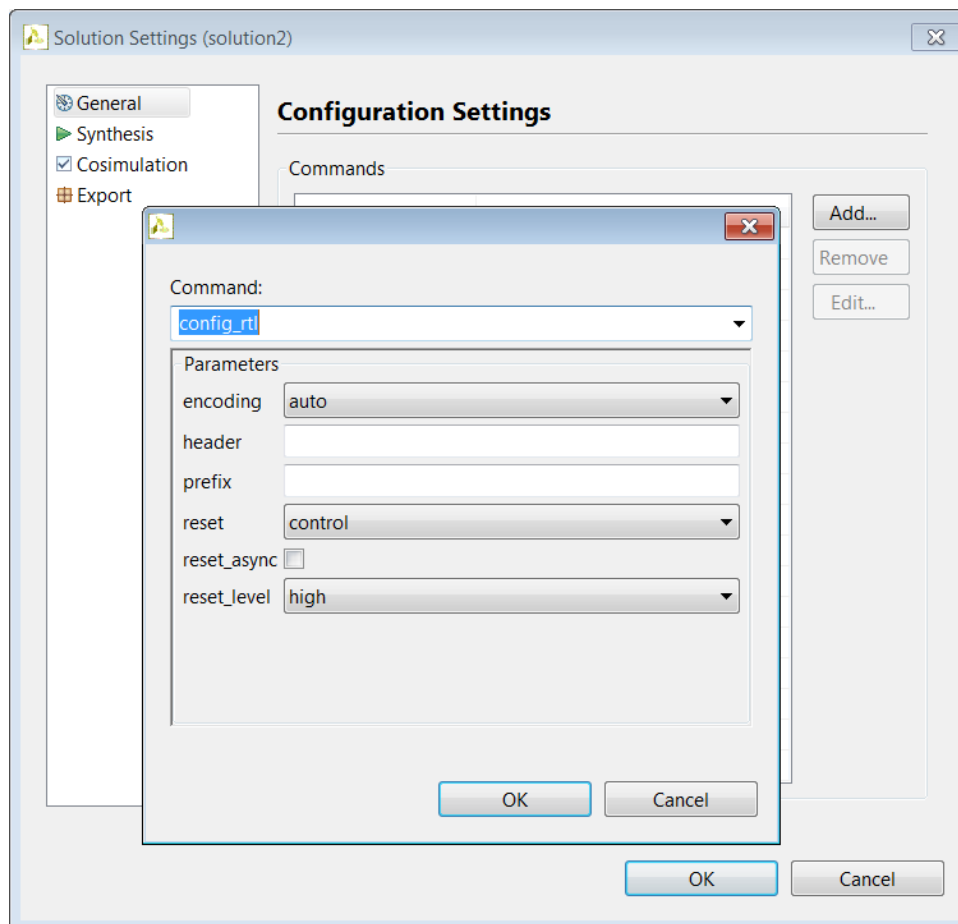


重要: 最上位関数の引数は、AXI4-Lite インターフェイスにインプリメントできます。関数引数に対して C/C++ で初期値を指定する方法はないので、これらの変数は RTL で初期化できません。初期化すると、C/C++ コードとは機能的に動作が異なる RTL デザインが作成され、C/RTL 協調シミュレーション中に検証エラーになります。

リセット動作の制御

リセット ポートは、リセット信号が適用されたときにリセット ポートに接続されたレジスタおよびブロック RAM を初期値に戻すために FPGA で使用されます。RTL リセット ポートの存在と動作は、次の図に示す `config_rtl` コンフィギュレーションで制御されます。このコンフィギュレーションにアクセスするには、[Solution] → [Solution Settings] → [General] → [Add] → [config_rtl] をクリックします。

図 51: RTL コンフィギュレーション



リセット設定では、リセットの極性設定、およびリセットが同期か非同期かのいずれかを設定できますが、[reset] オプションを使用すると、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。



重要: AXI4 インターフェイスがデザインで使用される場合は、`config_rtl` コンフィギュレーションの設定に関係なく、リセット極性は自動的にアクティブ Low に変更されます。これは、AXI4 規格の要件です。

[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]: すべての制御レジスタがリセットされるようになります (デフォルト)。制御レジスタは、ステートマシンに使用されるレジスタで、I/O プロトコル信号を生成するために使用されます。この設定により、デザインはその動作ステートを即座に開始できるようになります。
- [state]: 制御レジスタにリセットが (control 設定と同様) 追加されるほか、C コードのスタティックおよびグローバル変数から派生するレジスタまたはメモリにリセットが追加されます。この設定により、リセットが適用されたら、C コードで初期化されたスタティックおよびグローバル変数がそれらの初期化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がすべてリセットされます。

RESET 指示子を使用すると、さらに詳細にリセットを制御できます。変数がスタティックまたはグローバルの場合、RESET 指示子を使用してリセットを明示的に追加できたり、RESET 指示子の `off` オプションを使用して変数からリセットを削除したりできます。これはスタティック配列またはグローバル配列がデザインに含まれる場合に特に便利があります。



重要: リセットに `state` または `all` オプションを使用する際には、配列の影響に注意するようにしてください。

配列の初期化およびリセット

配列はスタティック変数として定義されるのが一般的であり、すべての要素が 0 に初期化され、通常ブロック RAM としてインプリメントされます。リセット オプション `state` または `all` が使用されると、すべての要素がブロック RAM としてインプリメントされ、リセット後に初期値に戻ります。これにより、RTL デザインでかなり不適切な条件が 2 つできてしまうことがあります。

- 電源投入時の初期化とは異なり、明示的なリセットでは RTL デザインでブロック RAM 内の各アドレスに反復的に値を設定する必要あり (N が大きい場合は多数のクロック サイクルがかかることあり)。
- リセットがデザインのすべての配列に追加される。

このようなブロック RAM すべてにリセット ロジックが配置され、RAM のすべての要素をリセットするためのサイクル オーバーヘッドが発生しないようにするには、次を実行します。

- デフォルトの `control` リセット モードを使用し、RESET 指示子を使用して、スタティックまたはグローバル変数を個別にリセットに指定します。
- または、`state` リセット モードを使用してから、RESET 指示子に `off` オプションを使用して、特定のスタティックまたはグローバル変数からリセットを削除します。

RTL 出力

Vivado HLS からの RTL 出力のさまざまな特性は、上記の図に示す `config_rtl` コンフィギュレーションを使用して制御できます。

- RTL ステート マシンで使用される FSM エンコーディングのタイプを指定。
- `-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追加。
- `prefix` オプションを使用して独自の名前を指定 (これがすべての RTL 出力ファイルに追加)。
- RTL ポートの名前を小文字に変更。

デフォルトの FSM コード形式は `onehot` です。その他、`auto`、`binary`、および `gray` などのオプションが使用できます。`auto` を選択すると、Vivado HLS ではデフォルトの `onehot` を使用してエンコーディング スタイルがインプリメントされますが、Vivado Design Suite で論理合成中に FSM スタイルが抽出されてインプリメントし直されることがあります。それ以外のエンコーディング スタイル (`binary`、`onehot`、`gray`) を選択している場合は、エンコーディング スタイルをザイリンクス論理合成ツールで最適化し直すことはできません。

RTL 出力ファイルの名前は、合成の最上位関数の名前から派生されます。異なる RTL ブロックが同じ最上位関数から作成される場合、RTL ファイルは同じ名前になり、同じ RTL プロジェクトにまとめることはできません。`prefix` オプションを使用すると、RTL ファイルが同じ最上位関数から生成できるので (デフォルトでは最上位関数と同じ名前になる)、同じディレクトリに簡単にまとめることができます。`lower_case_name` オプションを使用すると、出力 RTL で使用される名前が小文字だけになります。これにより、AXI インターフェイス用のポートなど、Vivado HLS で作成される I/O プロトコル ポートが、最終的な RTL では、デフォルト ポート名の `s_axis_<port>_TDATA` ではなく、`s_axis_<port>_tdata` のように指定されます。

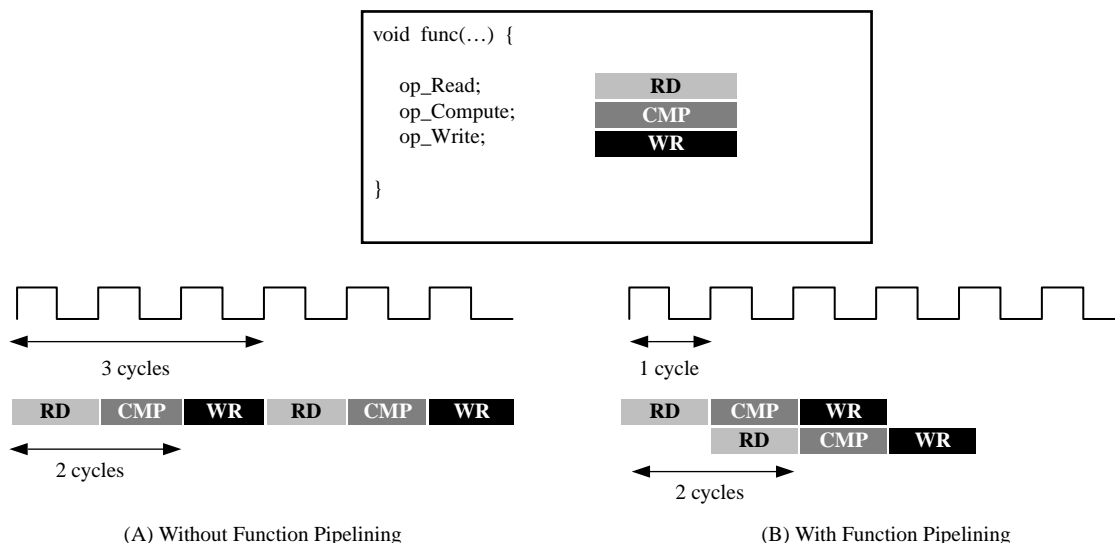
スループットの最適化

次の最適化を使用すると、スループットを改善できたり、開始間隔を削減したりできます。

関数およびループのパイプライン処理

パイプライン処理を実行すると、演算を同時に実行できるようになります。各実行ステップをすべての演算を完了しなくても、次の演算を開始できます。パイプライン処理は、関数およびループに対して実行できます。次の図は、関数のパイプラインによるスループットの改善を示しています。

図 52: 関数のパイプライン動作



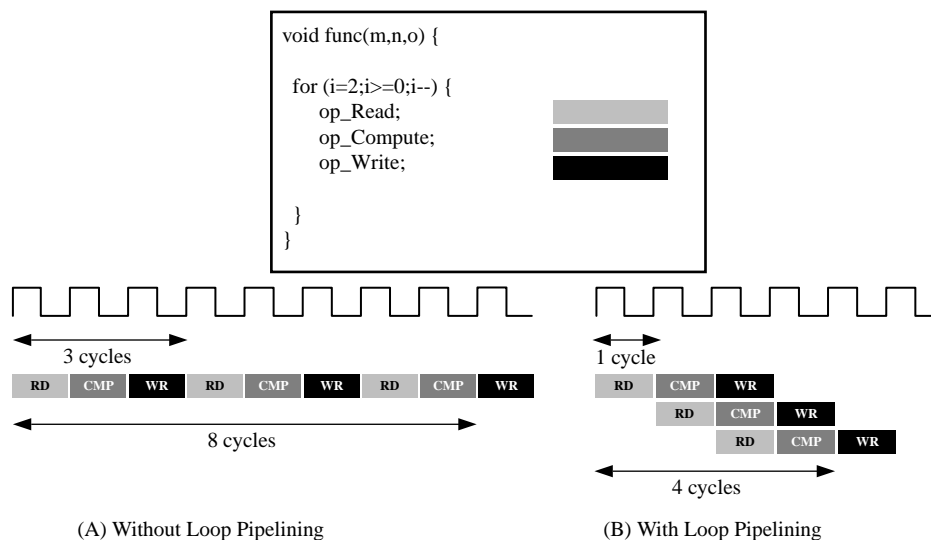
X14269

パイプライン処理を実行しない場合、上記の例では、関数で入力が入力が 3 クロック サイクルごとにこの例で読み出され、値が 2 サイクル後に出力されます。関数の開始間隔 (II) は 3 で、レイテンシは 3 です。パイプライン処理を実行すると、出力レイテンシの変更なしで、新しい入力が各サイクルで読み出されるようになります (II=1)。

ループのパイプライン処理を使用すると、ループ内の演算がオーバーラップ方式でインプリメントできるようになります。次の図の (A) はデフォルトの順次演算を示しています。各入力は 3 クロック サイクルごとに処理され (II=3)、最後の出力が書き出されるまでに 8 クロック サイクルかかっています。

(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るようになり、同じハードウェア リソースを使用して、開始間隔 (II) とレイテンシの両方を向上できます。

図 53: ループのパイプライン処理



X14277

関数またはループをパイプライン処理するには、PIPELINE 指示子を使用します。関数またはループ本体に指定します。指定しなければ開始間隔 (II) はデフォルトで 1 ですが、明示的に指定することもできます。

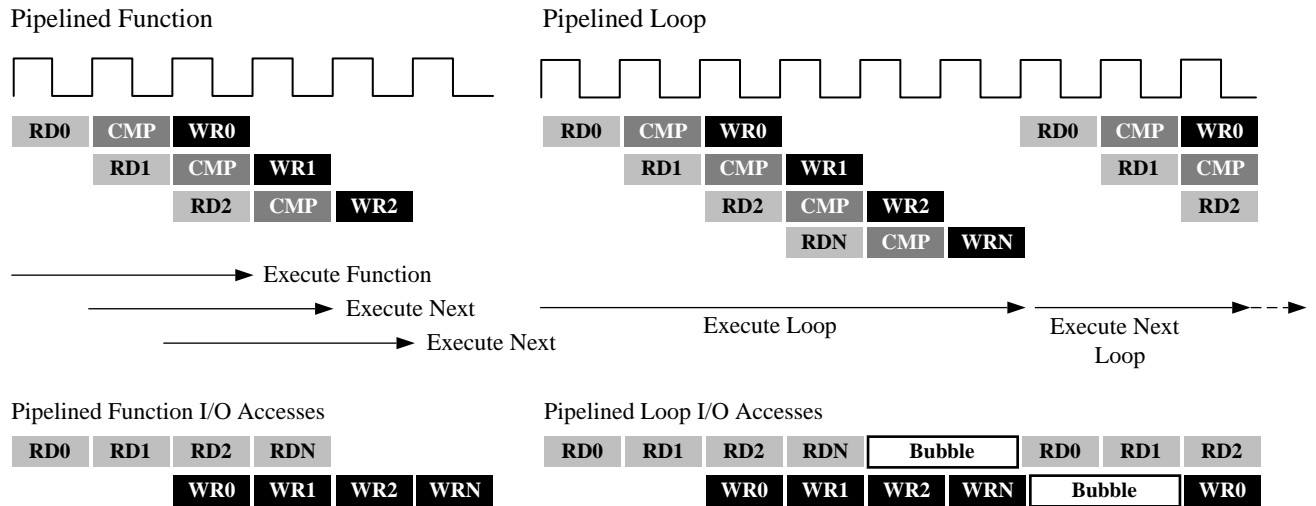
パイプライン処理は指定した領域のみに適用され、その下の階層には適用されません。ただし、その階層内にあるすべてのループが自動的に展開されます。指定した関数の下の階層にあるサブ関数は、個別にパイプライン処理する必要があります。サブ関数をパイプライン処理すると、その上のパイプライン処理された関数でパイプライン処理による最大限にパフォーマンスを改善できます。パイプライン処理されている最上位関数の下にあるサブ関数がパイプライン処理されていないと、パイプライン処理によるパフォーマンスの改善が制限される可能性があります。

パイプライン処理された関数とループのビヘイビアーには違いがあります。

- 関数の場合、パイプラインは永遠に実行されて、終了しません。
- ループの場合、パイプラインはループのすべての反復が終了するまで実行されます。

このビヘイビアーの違いについては、次の図を参照してください。

図 54: 関数およびループのパイプライン ビヘイビアー



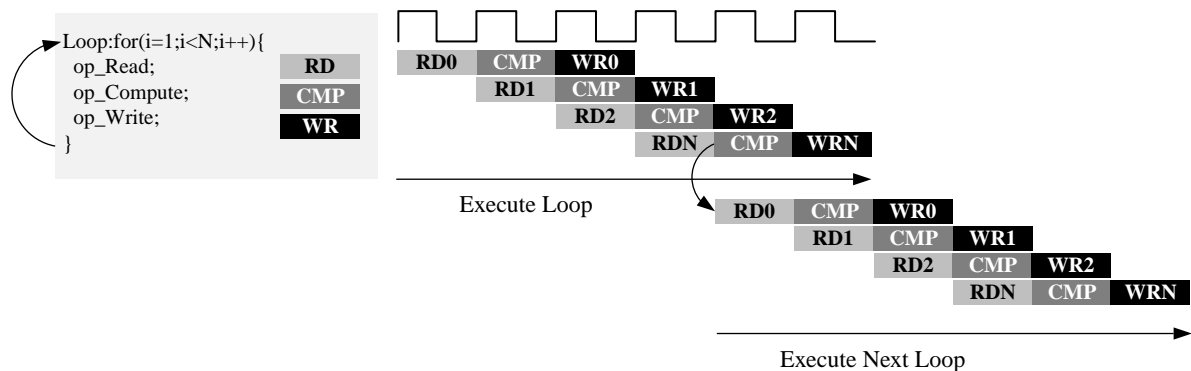
X14302

ビヘイビアーの違いは、パイプラインへの入力および出力の処理方法に影響します。上の図に示すように、パイプライン処理された関数は新しい入力を継続して読み出し、新しい出力を継続して書き込みます。反対に、ループは次のループの開始前にまずそのループ内の演算をすべて終了する必要がありますので、パイプライン処理されたループによりデータストリームに「バブル」(ループが最後の反復の実行を終了したときに新しい入力を読み出されないポイントと、ループが新しいループ反復を開始するときに新しい出力が書き込まれないポイント)が発生します。

パイプライン処理されたループを巻き戻してパフォーマンスを改善

前の図に示す問題を回避するため、PIPELINE プラグマには `rewind` というオプションのコマンドがあります。このコマンドを使用すると、`rewind` ループが最上位関数またはデータフロー プロセスの一番外側で、データフロー領域が複数回呼び出される場合に、このループへの連続する呼び出しの反復をオーバーラップできます。

次の図に、ループをパイプラインする際に `rewind` オプションを使用した場合の動作を示します。ループは、ループ反復カウントが終了すると再実行されます。通常は直後に再実行されますが、遅延があることもあり、それは GUI で表示されるようになっています。

図 55: `rewind` オプションを使用したループのパイプライン


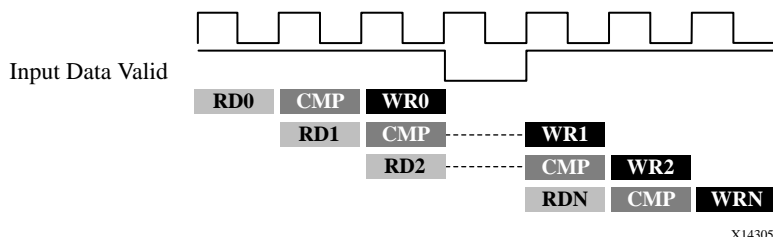
X14303

注記: ループが DATAFLOW 領域で使用される場合、Vivado HLS は自動的に連続する反復をオーバーラップさせるようにループをインプリメントします。詳細は、[タスク レベルの同時処理: データフロー最適化](#) を参照してください。

パイプラインのフラッシュ

パイプライン処理は、パイプラインにデータが入力され続けられる限り継続して実行されます。処理するデータがなければ、パイプラインは一時停止します。これを次の図に示します。入力データの valid 信号が Low になり、データがないことが示されています。処理するデータが新しく入ってくれば、パイプラインは動作を再開します。

図 56: 一時停止するループのパイプライン



場合によっては、空にできる (フラッシュできる) パイプラインが必要なこともあります。このために、flush オプションが提供されています。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し、最後の入力が処理されてパイプラインから出力されるまで、処理が継続されてその後の各パイプライン段が閉鎖されます。

ループの自動パイプライン

config_compile を使用すると、反復回数に基づくループの自動パイプライン処理をイネーブルにできます。この設定にアクセスするには、[Solution] → [Solution Settings] → [General] → [Add] → [config_compile] をクリックします。

反復回数は、pipeline_loops オプションで設定します。反復回数がこの設定値よりも少ないループは、すべて自動的にパイプライン処理されます。デフォルトは 0 で、ループの自動パイプライン処理は実行されません。

次のようなコードがあるとします。

```
for (y = 0; y < 480; y++) {
    for (x = 0; x < 640; x++) {
        for (i = 0; i < 5; i++) {
            // do something 5 times
            ...
        }
    }
}
```

pipeline_loops オプションを 6 に設定すると、上記のコードの最内 for ループは自動的にパイプライン処理されます。これは、次のコードと同じです。

```
for (y = 0; y < 480; y++) {
    for (x = 0; x < 640; x++) {
        for (i = 0; i < 5; i++) {
            #pragma HLS PIPELINE II=1
            // do something 5 times
            ...
        }
    }
}
```

自動パイプライン処理を適用するべきでないループがデザインに含まれる場合は、そのループに PIPELINE 指示子を off 設定で適用します。この指示子を off に設定すると、ループの自動パイプライン処理は実行されなくなります。



重要: Vivado HLS では、ユーザー指定の指示子をすべて実行された後に、`config_compile pipeline_loops` オプションが適用されます。たとえば、Vivado HLS でループにユーザー指定の UNROLL 指示子が適用される場合、このループがまず展開されるので、自動ループ パイプラインは適用できません。

パイプライン エラーの対処方法

関数がパイプライン処理されると、その階層内にあるすべてのループが自動的に展開されます。これはパイプライン処理のために必要です。ループに変数境界があると展開できず、関数がパイプライン処理されなくなります。

スタティック変数

スタティック変数は、ループの反復間でデータを維持するために使用される変数で、最終的なインプリメンテーションでレジスタになることがよくあります。これがパイプライン処理された関数に使用されていると、`vivado_hls` でデザインを十分に最適化できず、開始間隔 (II) が要件を超えることがあります。

次に、この状況の典型的な例を示します。

```
function_foo()
{
    static bool change = 0
    if (condition_xyz){
        change = x; // store
    }
    y = change; // load
}
```

`vivado_hls` でこのコードを最適化できない場合、ストア操作に 1 サイクル、ロード操作に 1 サイクル必要となります。この関数がパイプラインの一部である場合は、スタティック変更変数によりループ運搬依存が作成されるので、パイプラインを最小限の開始間隔である 2 を使用してインプリメントする必要があります。

これを回避する方法の 1 つは、コードを次のように書き直すことです。これにより、ループの反復ごとに read または write のみが存在するようになるので、デザインが II=1 でスケジューリングできるようになります。

```
function_readstream()
{
    static bool change = 0
    bool change_temp = 0;
    if (condition_xyz)
    {
        change = x; // store
        change_temp = x;
    }
    else
    {
        change_temp = change; // load
    }
    y = change_temp;
}
```

配列の分割によるパイプラインの改善

関数をパイプライン処理すると、次のようなエラー メッセージが表示されることがあります。

```
INFO: [SCHED 204-61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p0 is 1, current
assignments:
WARNING: [SCHED 204-69] 'load' operation ('mem_load', bottleneck.c:62)
on array
'mem',
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p1 is 1, current
assignments:
WARNING: [SCHED 204-69] 'load' operation ('mem_load_1',
bottleneck.c:62) on array
'mem',
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

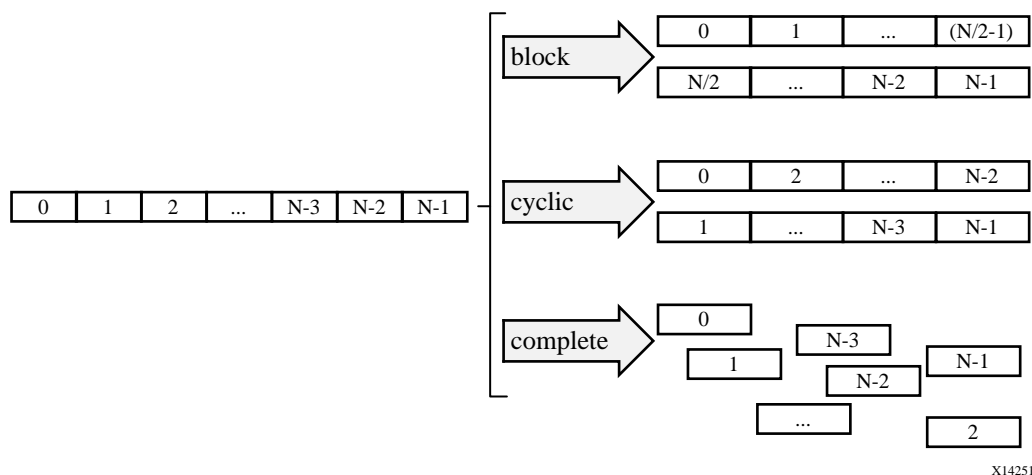
上記の Vivado HLS のメッセージは、メモリ ポートが限られているため、メモリへの load (読み込み) 操作 (mem_load_2) をスケジューリングできず、指定された開始間隔 (II) 1 を達成できないことを示しています。「The resource limit for core:RAM:mem:p0 is 1」と記述されているように core:RAM:mem:p0 のリソース制限は 1 で、62 行目の mem_load 演算で使用されてしまっています。ブロック RAM の 2 つ目のポートにもリソースが 1 つしかなく、これも mem_load_1 演算で使用されています。Vivado HLS は、このメモリ ポートの競合により、最終的な II が指定した 1 ではなく 2 になったことをレポートします。

この問題は、通常配列が原因で発生します。配列は、ブロック RAM (最大でも 2 つのデータ ポートしかない) としてインプリメントされます。これにより、read/write (または load/store) 集中型アルゴリズムのスループットが制限される可能性があります。配列 (1 つのブロック RAM リソース) を複数の小型の配列 (複数のブロック RAM) に分割してポート数を増加することにより、バンド幅を向上させることができます。

配列を分割するには、ARRAY_PARTITION 指示子を使用します。Vivado HLS には、次の図に示すように 3 つの配列分割方法があります。これらの分割方法は、次のとおりです。

- block: 元の配列の連続した要素が同じサイズのに分割されます。
- cyclic: 元の配列の要素がインターリーブされて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の要素に分割されます。これは、メモリの複数のレジスタへの分解に相当します。

図 57: 配列の分割



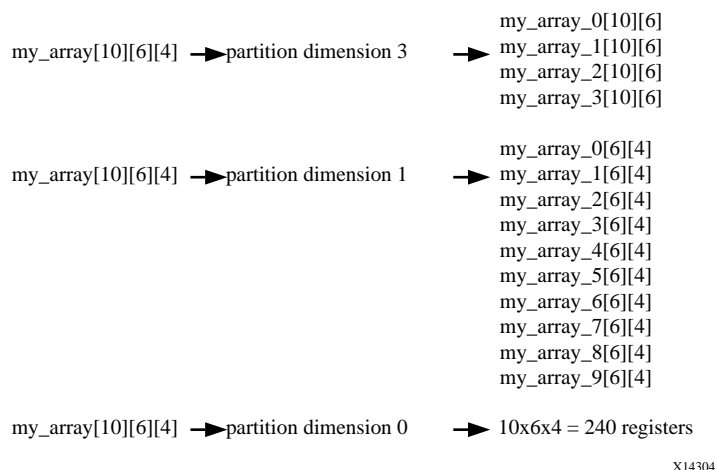
block および cyclic 分割では、`factor` オプションを使用して作成する配列の数を指定できます。前の図では、`factor` オプション 2 が使用され、配列が 2 つの配列に分割されています。配列の要素数がこの係数の整数倍ではない場合、最後の配列に含まれる要素数は少なくなります。

多次元配列を分割する際は、`dimension` オプションを使用して、分割する次元を指定できます。次の図に、次のコード例を異なる `dimension` オプションを使用して分割した場合を示します。

```
void foo (...) {
    int my_array[10][6][4];
    ...
}
```

`dimension` を 3 に設定すると 4 つの配列に、1 に設定すると 10 個の配列に分割されます。`dimension` を 0 に設定すると、すべての次元が分割されます。

図 58: 多次元配列の分割



配列の自動分割

`config_array_partition` は、配列を要素数に基づいて自動的に分割する方法を指定します。この設定にアクセスするには、[Solution] → [Solution Settings] → [General] → [Add] → [config_array_partition] をクリックします。

`throughput_driven` オプションを使用すると、分割のしきい値を調整し、分割を完全に自動化できます。
`throughput_driven` オプションを選択すると、指定したスループットを達成するために配列が自動的に分割されます。

Vivado HLS での依存

Vivado HLS では、C ソース コードに対応するハードウェア データパスが作成されます。

パイプライン指示子がない場合は、順次実行されるので依存を考慮する必要はありませんが、デザインがパイプライン処理される場合は、Vivado HLS で生成されるハードウェアに対してプロセッサ アーキテクチャと同じ依存が必要になります。

データ依存またはメモリ依存は通常、読み出しまたは書き込みの後に読み出しまたは書き込みが実行される場合に発生します。

- RAW (read-after-write) は true 依存とも呼ばれ、命令 (および命令で読み出させる/使用されるデータ) は前の演算の結果に依存します。

- l1: $t = a * b;$

- l2: $c = t + 1;$

l2 の読み出しは、l1 の t の書き込みに依存します。命令の順序を変えると、前の t の値が使用されます。

- WAR (write-after-read) は anti 依存とも呼ばれ、前の命令がデータを読み出すまで、現在の命令の書き込みでレジスタまたはメモリをアップデートできません。

- l1: $b = t + a;$

- l2: $t = 3;$

l2 の書き込みを l1 の前に実行することはできません。l1 の前に実行すると、 b の結果が無効になります。

- WAW (write-after-write) は、レジスタまたはメモリを特定の順序で書き込む必要がある場合の依存です。この順序に従わないと、ほかの命令が破損することがあります。

- l1: $t = a * b;$

- l2: $c = t + 1;$

- l3: $t = 1;$

l3 の書き込みは、l1 の書き込み後に実行する必要があります。そうしないと、l2 の結果が無効になります。

- read-after-read では、変数が揮発性と宣言されていない場合は、命令の順序を自由に並べ替えることができるので、依存はありません。変数が揮発性と宣言されている場合は、命令の順序を保持する必要があります。

たとえば、パイプラインが生成される場合、後の段階で読み出されるレジスタまたはメモリの位置が前の書き込みで変更されないようにする必要があります。これが true 依存または RAW (read-after-write) 依存です。次はその具体例です。

```
int top(int a, int b) {
    int t,c;
    I1: t = a * b;
    I2: c = t + 1;
    return c;
}
```

変数 `t` に依存があるので、`I2` は `I1` が完了するまで評価できません。ハードウェアでは、乗算に 3 クロックかかる場合、`I2` がその分だけ遅延されます。上記の関数がパイプライン処理されると、Vivado HLS でこれが true 依存として検出されて、演算が適切にスケジューリングされます。データ転送最適化を使用して RAW 依存が削除されて、関数が `II=1` で動作できるようになります。

メモリ依存はこの例が変数だけでなく、配列に適用される場合に発生します。

```
int top(int a) {
    int r=1,rnext,m,i,out;
    static int mem[256];
    L1: for(i=0;i<=254;i++) {
        #pragma HLS PIPELINE II=1
        I1: m = r * a; mem[i+1] = m; // line 7
        I2: rnext = mem[i]; r = rnext; // line 8
    }
    return r;
}
```

上記の例では、ループ `L1` のスケジューリングにより、次のようなスケジューリング警告メッセージが表示されます。

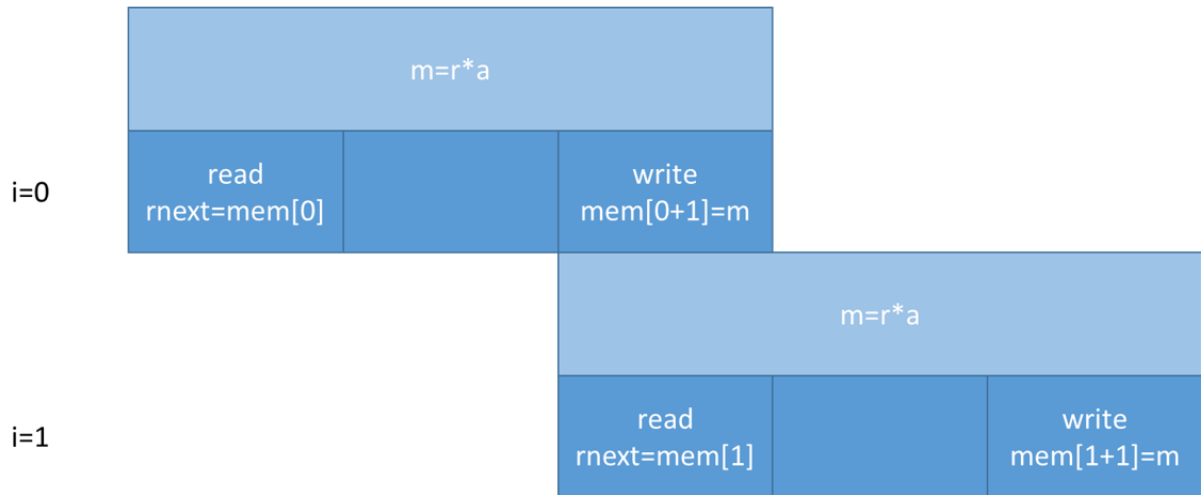
```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1,
distance = 1)
between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7 on array
'mem' and
'load' operation ('rnext', top.cpp:8) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

ユーザーがインデックスを書き込んで別のインデックスを読み出しているので、このループの同じ反復内には問題はありません。2 つの命令は同時に並列で実行可能ですが、2、3 回の反復間は、読み出しと書き込みを監視するようにしてください。

```
// Iteration for i=0
I1: m = r * a; mem[1] = m; // line 7
I2: rnext = mem[0]; r = rnext; // line 8
// Iteration for i=1
I1: m = r * a; mem[2] = m; // line 7
I2: rnext = mem[1]; r = rnext; // line 8
// Iteration for i=2
I1: m = r * a; mem[3] = m; // line 7
I2: rnext = mem[2]; r = rnext; // line 8
```

2つの連続した反復を見てみると、I1からの乗算結果 m (レイテンシ=2) がループの次の反復の I2 で読み出され、 r_{next} に代入されます。この場合、次の反復は前の計算の書き込みが終了する前に $mem[i]$ の読み出しを開始できないので、RAW 依存があります。

図 59: 依存の例



クロック周波数が増加すると、乗算器でより多くのパイプライン段が必要になり、レイテンシが増加します。これにより、II も増加します。

次のコードでは、演算がスワップされ、機能が変更されています。

```
int top(int a) {
    int r,m,i;
    static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     r = mem[i];           // line 7
I2:     m = r * a , mem[i+1]=m; // line 8
    }
    return r;
}
```

次のようなスケジューリング警告が表示されます。

```
INFO: [SCHED 204-61] Pipelining loop 'L1'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 2,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
```

```
(II = 3,
distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 4, Depth: 4.
```

この後の読み出しと書き込みを数反復間監視してください。

```
Iteration with i=0
I1:      r = mem[0];           // line 7
I2:      m = r * a , mem[1]=m; // line 8
Iteration with i=1
I1:      r = mem[1];           // line 7
I2:      m = r * a , mem[2]=m; // line 8
Iteration with i=2
I1:      r = mem[2];           // line 7
I2:      m = r * a , mem[3]=m; // line 8
```

II が長くなるのは、WAR 依存では `mem[i]` から `r` が読み出され、乗算が実行されてから、`mem[i+1]` に書き込まれるからです。

false 依存の削除によるループのパイプライン改善

false 依存とは、コンパイラが保守的すぎる場合に発生する依存のことです。これらの依存は、実際のコードにはありませんが、コンパイラでは判断できません。これらの依存があると、ループ パイプラインがされないことがあります。

次の例は、フォルス依存を示しています。この例では、読み出しおよび書き込みが同じループ反復内の 2 つの異なるアドレスにアクセスします。これらのアドレスはどちらも入力データに依存し、`hist` 配列の個別エレメントを指定できます。このため、Vivado HLS ではこれらのアクセスのどちらも同じ位置にアクセスできると想定されます。この結果、配列への読み出しと書き込みが交互のサイクルでスケジューラされ、ループ II が 2 になります。ただし、このコードは、`hist[old]` および `hist[val]` が `if(old == val)` 条件の `else` 分岐に含まれるため、これらが同じ位置にアクセスすることがないことを示しています。

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
    for(i = 0; i < INPUT SIZE; i++)
    {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val)
        {
            acc = acc + 1;
        }
        else
        {
            hist[old] = acc;
            acc = hist[val] + 1;
        }

        old = val;
    }

    hist[old] = acc;
}
```

この非効率性を克服するには、DEPENDENCE 指示子を指定して、Vivado HLS に依存に関する情報を入力します。

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
    #pragma HLS DEPENDENCE variable=hist intra RAW false
    for(i = 0; i < INPUT_SIZE; i++)
    {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val)
        {
            acc = acc + 1;
        }
        else
        {
            hist[old] = acc;
            acc = hist[val] + 1;
        }

        old = val;
    }

    hist[old] = acc;
}
```

注記: FALSE 依存でないものを FALSE 依存と指定すると、ハードウェアが正しく機能しなくなる可能性があります。依存が正しいかどうか (TRUE または FALSE) を確認してから指定してください。

依存には、主に二種類あります。

- Inter: 依存が同じループ内の別の反復間にあることを指定します。

Vivado HLS でこのタイプの依存を 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が不可能になります。

- Intra: 同じ反復の開始と終了でアクセスされる配列など、ループ内の同じ反復内の依存を指定します。

このタイプの依存を FALSE に設定すると、Vivado HLS によりループ内で演算を自由に移動でき、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。

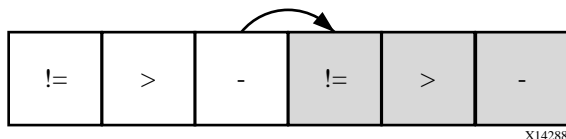
スカラーの依存

スカラーの依存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 データの依存は、次のようになります。

```
while (a != b) {
    if (a > b) a -= b;
    else b -= a;
}
```

このループの次の反復は、次の図に示すように、現在の反復が計算されて、a および b の値がアップデートされるまで開始しません。

図 60: スカラーの依存



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 で指定した開始間隔 (II) でパイプライン処理できない場合は、開始間隔が増加されます。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、パイプライン処理されないデザインが出力されます。

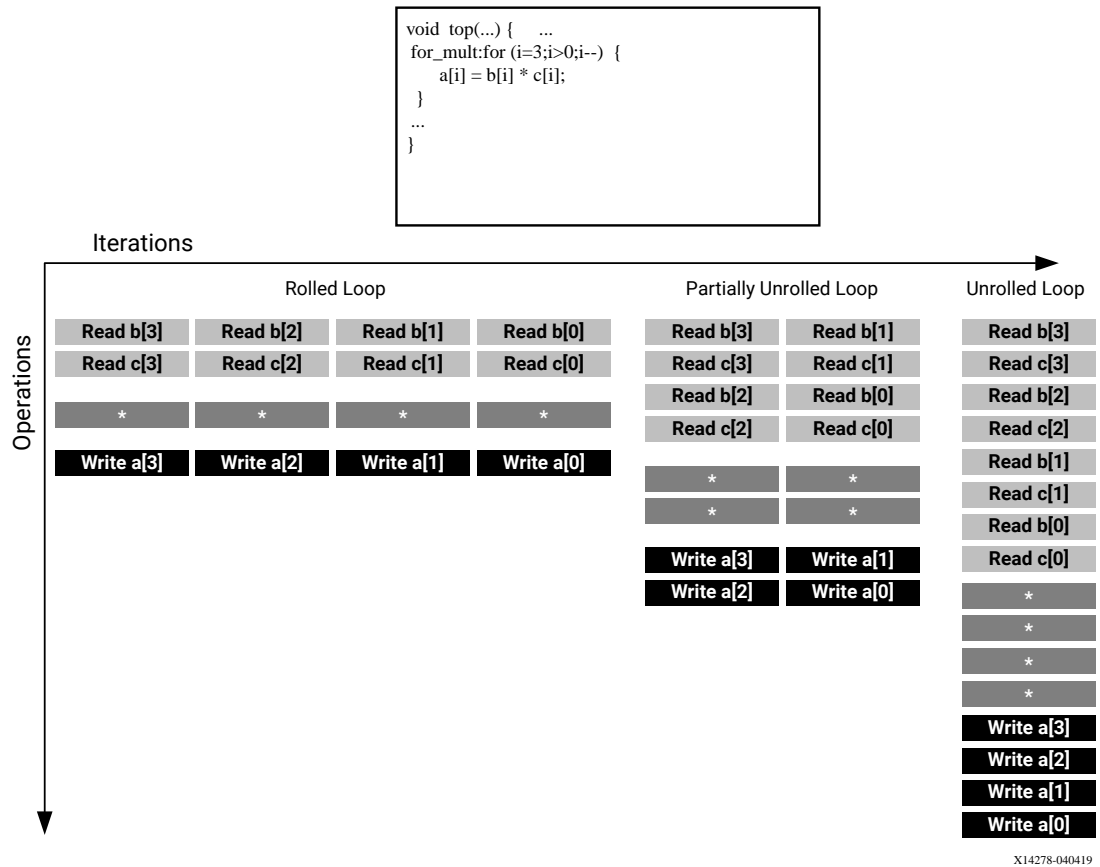
最適なループ展開によるパイプラインの改善

Vivado HLS では、デフォルトではループは展開されません。これらの非展開ループは、ループの各反復で使用されるハードウェア リソースを 1 つ生成します。これにより、リソース効率の高いブロックが作成されますが、パフォーマンスのボトルネックとなることがあります。

Vivado HLS では、UNROLL を使用すると、ループを展開または一部展開できます。

次の図に、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。この例では、配列 `a[i]`、`b[i]`、and `c[i]` がブロック RAM にマップされると想定されます。この例は、ループの展開を適用することにより簡単に多数のインプリメンテーションを作成できることを示しています。

図 61: ループ展開の詳細



- Rolled Loop (非展開ループ): ループが非展開の場合、各反復は別々のクロック 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、必要なのは乗算器 1 つだけで、各ブロック RAM はシングルポート RAM にできます。
- Partially Unrolled Loop (部分的に展開されたループ): この例では、ループは係数 2 で部分的に展開されます。各 RAM に対して 2 つの読み出しまたは書き込みを同じクロック サイクルで実行するために、このインプリメンテーションには乗算器 2 つとデュアルポート RAM が必要となりますが、完了するのに 2 クロック サイクルしかかからず、展開されていないループと比較すると、開始間隔もレイテンシも半分ですみます。
- Unrolled loop (展開ループ): 完全に展開されたバージョンでは、すべてのループ動作が 1 クロック サイクルで実行できます。乗算器が 4 つ必要になります。このインプリメンテーションでは、同じクロック サイクルで 4 つの読み出しと 4 つの書き込みが実行される必要があります。ブロック RAM には最大で 2 つのポートしか含めることができないので、このインプリメンテーションでは配列を分割する必要があります。

ループ展開は、デザインのループごとに UNROLL 指示子を適用すると実行できます。または、関数に UNROLL 指示子を適用して、その関数のスコープ内のすべてのループが展開されるようにします。

ループが完全に展開される場合、データ依存およびリソースが許容される限り、すべての操作が並列処理で実行されます。ループの 1 回の反復での演算に前の反復からの結果が必要な場合、それらは並列では実行されませんが、データが使用可能になるとすぐに実行されます。ループを完全に展開して最適化すると、通常ループ ボディにロジックのコピーが複数含まれるようになります。

次のコード例は、ループ展開を使用して最適化されたデザインを作成する方法を示しています。この例では、データが配列にインターリーブされたチャンネルとして格納されています。ループが $ll=1$ でパイプライン処理される場合は、各チャンネルは 8 ブロック サイクルごとにのみ読み出されて書き込まれます。

```
// Array Order : 0 1 2 3 4 5 6 7 8      9      10      etc. 16
etc...
// Sample Order: A0 B0 C0 D0 E0 F0 G0 H0 A1      B1      C2      etc. A2
etc...
// Output Order: A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2
etc...

#define CHANNELS 8
#define SAMPLES 400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```

`factor` を 8 に設定してループを部分的に展開すると、チャンネルごと (8 つ目のサンプルごと) に並列処理できるようになります (入力配列と出力配列も `cyclic` で分割し、クロック サイクルごとに複数アクセスを可能にした場合)。ループが `rewind` オプションでパイプラインされても、最上位またはデータフロー領域内のいずれかで並列で呼び出されていれば、すべての 8 チャンネルが並列で継続して処理されます。

```
void foo (dout_t d_out[N], din_t d_in[N]) {
    #pragma HLS ARRAY_PARTITION variable=d_i cyclic factor=8 dim=1 partition
    #pragma HLS ARRAY_PARTITION variable=d_o cyclic factor=8 dim=1 partition

    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        #pragma HLS PIPELINE rewind
        #pragma HLS UNROLL factor=8

        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```

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 exit 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、次の例のようにコードが変更されます。このコードでは、機能が同じになるように、break コンストラクトが使用されます。

```
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 (この場合は 2) が最大反復カウント N の整数係数だとわかっている場合は、skip_exit_check オプションを使用すると、exit チェックと関連ロジックが削除されます。展開の結果は、次のようになります。

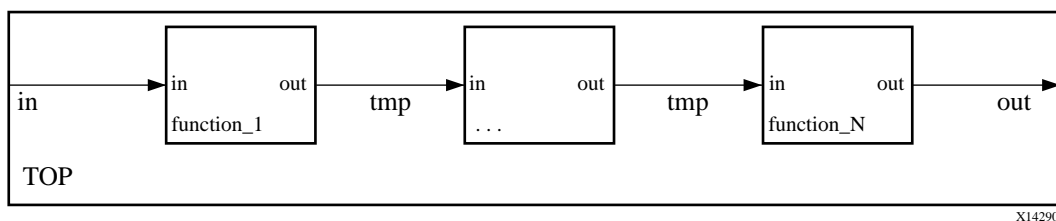
```
for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
```

これによりエリアが最小限に抑えられ、制御ロジックが単純になります。

タスク レベルの同時処理: データフロー最適化

データフロー最適化は、次の図に示すように、一連の順次タスク (関数やループなど) のセットに使用するのが便利です。

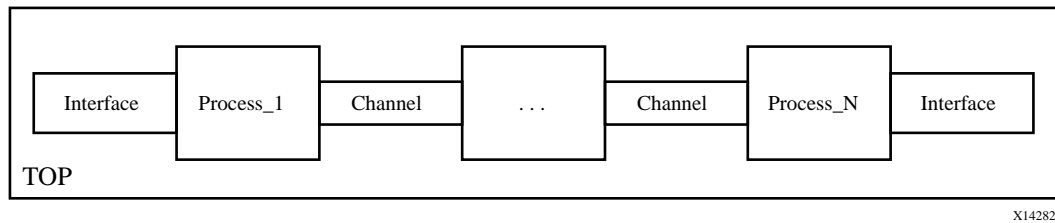
図 62: 順次機能の記述



上の図は、3 つの一連のタスクを示していますが、通信構造は表示されているよりも複雑である可能性があります。

この一連の順次タスクを使用すると、データフロー最適化で次の図に示すような同時処理プロセスのアーキテクチャが作成されます。データフロー最適化は、デザインのスループットとレイテンシを向上する優れた方法です。

図 63: 並列処理アーキテクチャ

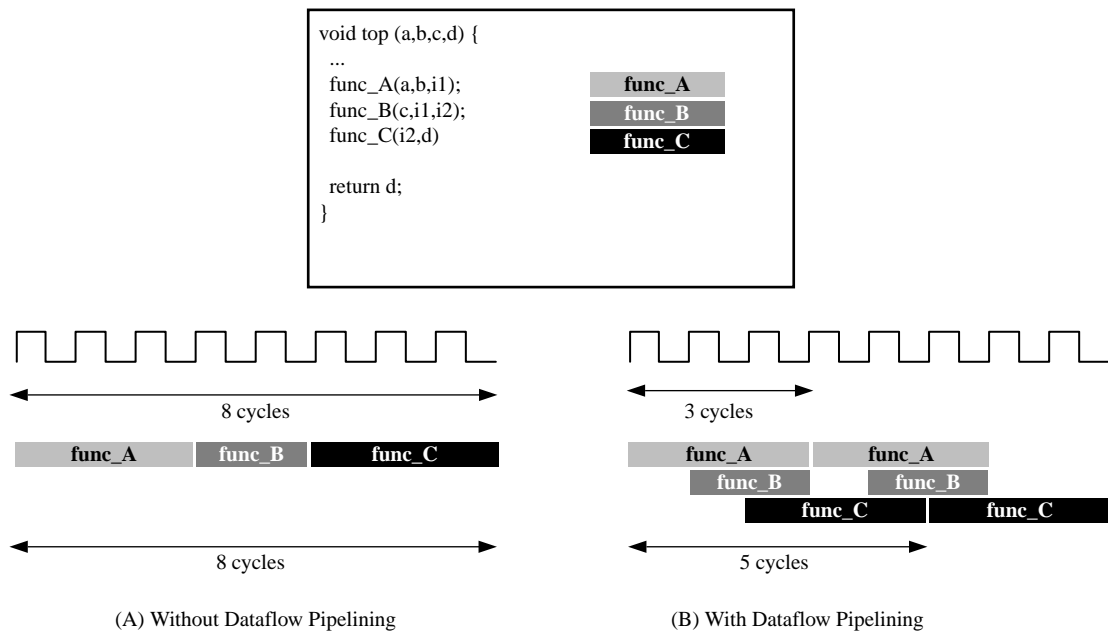


次の図に、データフロー最適化によりタスクの実行をオーバーラップさせることにより、デザイン全体のスループットが向上し、レイテンシが削減することを示します。

次の図では、(A) はデータフロー最適化を適用していない場合を示しています。インプリメンテーションで新しい入力 `func_A` が処理できるようになるまでに 8 サイクル、出力が `func_C` に書き込まれるまでに 8 サイクルが必要です。

(B) は同じ例でデータフロー最適化を適用した場合を示しています。 `func_A` が 3 クロック サイクルごとに新しい入力の処理を開始できる (開始間隔が短くなる) ので、最終値を出力するまでに必要なのは 5 クロック サイクルとなり、レイテンシが短くなっています。

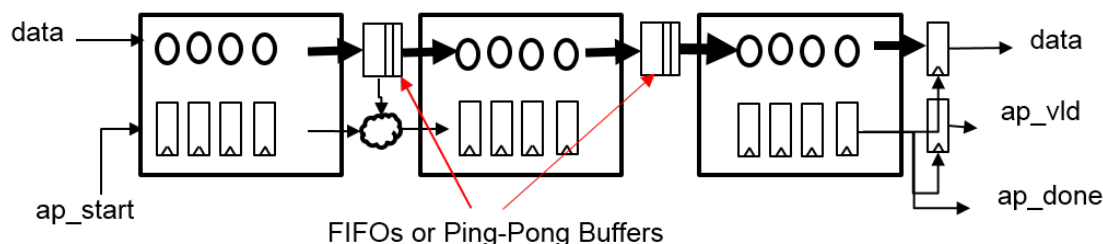
図 64: データフロー最適化



このタイプの並列処理では、ハードウェアに多少のオーバーヘッドが発生します。関数本体やループ本体などの領域にデータフロー最適化を適用するよう指定されている場合、Vivado HLS で関数本体またはループ本体が解析され、データフローをモデル化する個別のチャンネルが作成され、データフロー領域の各タスクの結果が保存されます。これらのチャンネルは、スカラー変数の場合は単純な FIFO、配列などのスカラー変数以外のものの場合はピンポン (PIPO) バッファです。これらの各チャンネルには、FIFO またはピンポン バッファがフルまたは空であることを示す信号も含まれます。これらの信号は、完全にデータ ドリブンのハンドシェイク インターフェイスです。個別の FIFO また

はピンポン バッファにより、Vivado HLS で各タスクをそれぞれのタイミングで実行でき、スループットは入力バッファおよび出力バッファが使用可能かどうかのみによって制限されます。この手法では、通常のパイプラインインプリメンテーションよりもタスクの実行をより良くインターリーブできますが、追加の FIFO またはピンポン バッファ用のブロック RAM が使用されます。前の図は、次の図の同じ例をデータフロー領域で実現した構造を示しています。

図 65: データフロー最適化で作成された構造



データフロー最適化では、スタティックにパイプライン処理されたソリューションよりもパフォーマンスを向上できる可能性があります。厳密な中央制御型のパイプライン ストールが、FIFO またはピンポン バッファを使用した配布ハンドシェイク アーキテクチャに置き換えられます。データフロー最適化は、一連のプロセスだけでなく、有向非巡回グラフ DAG 構造にも使用できます。プロセスが FIFO に接続されていると 1 つの反復内でオーバーラップする場合と、PIPO および FIFO を介して異なる反復間でオーバーラップする場合の、2 種類のオーバーラップが生成される可能性もあります。

正規形式

Vivado HLS では、データフロー最適化を適用するために領域が変更されます。ザイリンクスでは、この領域 (正規領域と呼ばれる) 内のコードは正規形式を使用して記述することをお勧めします。データフロー最適化を適用するには、2 つの正規形式があります。

1. 関数がインライン展開されていない関数の正規形式。

```
void dataflow(Input0, Input1, Output0, Output1)
{
    #pragma HLS dataflow
    UserDataTypes C0, C1, C2;
    func1(read Input0, read Input1, write C0, write C1);
    func2(read C0, read C1, write C2);
    func3(read C2, write Output0, write Output1);
}
```

2. ループ本体内のデータフロー。

ループ (含まれる関数がインライン展開されていない) の場合、積分ループ変数は次のように設定する必要があります。

- a. ループ ヘッダーで宣言され 0 に設定される初期値。
- b. ループ条件は正の定数または定数の関数引数。
- c. 1 ずつインクリメント。

- d. データフロー プラグマはループ内にある必要あり。

```
void dataflow(Input0, Input1, Output0, Output1)
{
    for (int i = 0; i < N; i++)
    {
        #pragma HLS dataflow
        UserDataTypes C0, C1, C2;
        func1(read Input0, read Input1, write C0, write C1);
        func2(read C0, read C0, read C1, write C2);
        func3(read C2, write Output0, write Output1);
    }
}
```

正規本体

正規領域内では、正規本体は次のガイドラインにしたがっている必要があります。

1. ローカルの非スタティック スカラーまたは配列/ポインター変数、もしくはローカルのスタティック ストリーム変数を使用します。ローカル変数は、関数本体 (関数内のデータフローの場合) またはループ本体 (ループ内のデータフローの場合) で宣言します。
2. データをフィードバックなしで、1 つの関数から別の関数に後に渡される関数呼び出しシーケンスの条件は次のとおりです。
 - a. 変数 (スカラー以外) に含めることができるのは、1 つの読み出しプロセスと 1 つの書き込みプロセスのみです。
 - b. ローカル変数を使用する場合は write before read (受信前に送信) を使用します。
 - c. 関数引数を使用する場合、read before write (送信前に受信) を使用します。本体内の反依存はデザインで保持される必要があります。
 - d. 関数の戻り値型は void である必要があります。
 - e. 変数を介した異なるプロセス間のループ キャリー依存はないようにします。
 - 正規ループ内 (1 つの反復で書き出されて次の反復で読み込まれる値など)。
 - 最上位関数への連続呼び出し内 (1 つの反復で書き出されて次の反復で読み込まれる入力引数など)。

データフロー チェック

Vivado HLS には、データフロー チェッカーが含まれており、オンにすると、コードが推奨される標準形式に従っているかどうかを確認され、問題があると、エラーや警告メッセージを表示します。デフォルトでは、このチェッカーは warning に設定されています。チェッカーは config_dataflow Tcl コマンドの strict モードで error に設定したり、off を選択してディスエーブルにしたりできます。

```
config_dataflow -strict_mode (off | error | warning)
```

データフロー最適化の制限

データフロー最適化では、タスク (関数およびループ) 間、および最大限のパフォーマンスを得るため理想的にはパイプライン処理された関数およびループ間のデータの流れが最適化されます。そのためタスクを順につなげる必要はありませんが、データの転送方法にいくつかの制限があります。

次のような条件下では、Vivado® HLS でデータフロー最適化により実行可能なオーバーラップが阻止されたり制限されたりする可能性があります。

- シングル プロデューサー コンシューマー違反
- タスクのバイパス
- タスク間のフィードバック
- タスクの条件付き実行
- 複数の exit 条件を持つループ



重要: これらのコーディング スタイルのいずれかが使用されている場合、Vivado HLS で状況を説明するメッセージが表示されます。

注記: DATAFLOW 指示子を使用する場合は、[Analysis] パースペクティブのデータフロー ビューアーで構造を確認できます。

シングル プロデューサー コンシューマー違反

Vivado HLS でデータフロー最適化が実行されるようにするには、タスク間で渡される要素すべてがシングル プロデューサー コンシューマー モデルに従っている必要があります。変数はそれぞれ 1 つのタスクから駆動され、1 つのタスクでのみ消費される必要があります。次のコード例では、temp1 がファンアウトして、Loop2 および Loop3 の両方に入力されています。これは、シングル プロデューサー コンシューマー モデルに違反しています。

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp1[j] * 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out2[k] = temp1[k] * 456;
    }
}
```

これを修正したコードでは、Split 関数を使用して、シングル プロデューサー コンシューマー デザインを作成しています。この場合、データは Loop1 から Split 関数に流れ、その後 Loop2 および Loop3 に流れます。データが 4 つのタスクすべての間で流れるようになったので、Vivado HLS でデータフロー最適化を実行できます。

```
void Split (in[N], out1[N], out2[N]) {
    // Duplicated data
    L1:for(int i=1;i<N;i++) {
        out1[i] = in[i];
        out2[i] = in[i];
    }
}

void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
    Split(temp1, temp2, temp3);
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp2[j] * 123;
    }
}
```

```

}
Loop3: for(int k = 0; k < N; k++) {
    data_out2[k] = temp3[k] * 456;
}
}

```

タスクのバイパス

また、データは通常 1 つのタスクから次のタスクに流れる必要があります。タスクをバイパスすると、データフロー最適化のパフォーマンスが低下する可能性があります。次の例では、Loop1 で temp1 および temp2 の値が生成されますが、次のタスク Loop2 では temp1 の値しか使用されません。temp2 の値は Loop2 の後まで使用されません。このため、temp2 はシーケンスの次のタスクをバイパスすることになり、データフロー最適化のパフォーマンスが制限される可能性があります。

```

void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp3[j] = temp1[j] + 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp2[k] + temp3[k];
    }
}

```

この例ではループ反復制限がすべて同じなので、Loop2 が temp2 を使用して、temp4 を出力するようにコードを変更できます。これで、データが 1 つのタスクから次のタスクへ流れるようになります。

```

void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N], temp2[N], temp3[N], temp4[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp3[j] = temp1[j] + 123;
        temp4[j] = temp2[j];
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp4[k] + temp3[k];
    }
}

```

タスク間のフィードバック

フィードバックは、DATAFLOW 領域で 1 つのタスクからの出力が前のタスクで消費される場合に起こります。タスク間のフィードバックは、DATAFLOW 領域内では使用できません。このため、Vivado HLS でフィードバックが検出されると、状況によって警告メッセージが表示され、データフロー最適化は実行されません。

タスクの条件付き実行

データフロー最適化では、条件付きで実行されるタスクは最適化されません。次は、この制限を示す例です。この例では、Loop1 および Loop2 の条件を実行すると、データが次のループに流れなくなり、Vivado HLS でこれらのループ間のデータフローが最適化されなくなります。

```
void foo(int data_in1[N], int data_out[N], int sel) {
    int temp1[N], temp2[N];

    if (sel) {
        Loop1: for(int i = 0; i < N; i++) {
            temp1[i] = data_in[i] * 123;
            temp2[i] = data_in[i];
        }
    } else {
        Loop2: for(int j = 0; j < N; j++) {
            temp1[j] = data_in[j] * 321;
            temp2[j] = data_in[j];
        }
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

どの場合にも各ループが実行されるようにするには、コードを次のように変更する必要があります。この例の場合、条件文を最初のループに移動します。これで、両方のループが常に実行され、データが常に次のループに流れるようになります。

```
void foo(int data_in[N], int data_out[N], int sel) {
    int temp1[N], temp2[N];

    Loop1: for(int i = 0; i < N; i++) {
        if (sel) {
            temp1[i] = data_in[i] * 123;
        } else {
            temp1[i] = data_in[i] * 321;
        }
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp2[j] = data_in[j];
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

複数の終了条件を持つループ

複数の終了ポイントのあるループは、DATAFLOW 領域では使用できません。次の例では、Loop2 に 3 つの終了条件があります。

- N の値により終了 (ループは $k \geq N$ の場合に終了)。
- break 文による終了。

- continue 文による終了。

```
#include "ap_cint.h"
#define N 16

typedef int8 din_t;
typedef int15 dout_t;
typedef uint8 dsc_t;
typedef uint1 dsel_t;

void multi_exit(din_t data_in[N], dsc_t scale, dsel_t select, dout_t
data_out[N]) {
    dout_t temp1[N], temp2[N];
    int i,k;

    Loop1: for(i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }

    Loop2: for(k = 0; k < N; k++) {
        switch(select) {
            case 0: data_out[k] = temp1[k] + temp2[k];
            case 1: continue;
            default: break;
        }
    }
}
```

ループの終了条件は常にループ境界で定義されるので、break または continue 文を使用すると、ループを DATAFLOW 領域で使用できなくなります。

最後に、DATAFLOW 最適化には階層インプリメンテーションはありません。サブ関数またはループに DATAFLOW 最適化が有益な可能性のあるタスクが含まれる場合、DATAFLOW 最適化をそのループまたはサブ関数に適用するか、サブ関数をインライン展開する必要があります。

注記: std::complex は DATAFLOW 領域内では直接使用できません。これはネイティブ データ型として定義して、プロデューサー内で型変換する必要があります。

```
#dataflow
float A[N][2];
prod(A, in);
cons(out,A);

Producer(std::complex &)
{
}
```

データフロー メモリ チャンネルの設定

Vivado HLS では、タスク間のチャンネルを、データのプロデューサーとコンシューマーのアクセス パターンによって、ピンポン バッファまたは FIFO のいずれかでインプリメントされます。

- スカラー、ポインター、リファレンス パラメーターの場合、Vivado HLS でチャンネルが FIFO としてインプリメントされます。
- パラメーター (プロデューサーまたはコンシューマー) が配列の場合、Vivado HLS でチャンネルがピンポン バッファまたは FIFO としてインプリメントされます。

- 。 Vivado HLS でデータが順番どおり (シーケンシャルに) アクセスされると決定されると、Vivado HLS はメモリ チャンネルを深さ 2 の FIFO としてインプリメントします。
- 。 Vivado HLS でデータが順にアクセスされるかを判断できない場合、または任意の順序でアクセスされていると判断された場合、Vivado HLS はメモリ チャンネルをピンポン バッファ (プロデューサーまたはコンシューマーの配列の最大サイズで定義された 2 つのブロック RAM) としてインプリメントします。

注記: ピンポン バッファが使用されると、すべてのサンプルが常にチャンネルに損失なしで保持されるようになりますが、これが控えめすぎる場合もあります。

タスク間で使用されるデフォルトのチャンネルを明示的に指定するには、`config_dataflow` を使用します。これを使用すると、デザイン内のすべてのチャンネルにデフォルト チャンネルが設定されます。チャンネルで使用されるメモリ サイズを削減し、反復内で重複できるようにするには、FIFO を使用します。FIFO 内の深さ (要素数) を明示的に設定する場合は、`-fifo_depth` オプションを使用します。

FIFO チャンネルのサイズを指定すると、デフォルトの設定が上書きされます。デザイン内のタスクが指定した FIFO サイズよりも速いレートでサンプルを入力または出力する場合、FIFO が空またはフルになることがあります。この場合、読み出しまたは書き込みはできないので、デザインの動作が停止し、デッドロック状態になってしまうことがあります。

注記: デッドロック状態は、C/RTL 協調シミュレーションを実行する場合、またはこのブロックが完全なシステムで使用される場合にのみ発生します。

ザイリンクスでは、FIFO の深さを設定する場合は、最初は深さを転送されるデータの最大値 (タスク間で渡される配列のサイズ) に設定して C/RTL 協調シミュレーションで問題がないことを確認し、その後 FIFO のサイズを削減してそれでも問題がないことを C/RTL 協調シミュレーションで再度確認することを勧めます。RTL 協調シミュレーションがエラーになる場合は、FIFO のサイズが小さすぎて、停止やデッドロック状態を回避できなかった可能性があります。

配列をピンポン バッファまたは FIFO として指定

デフォルトでは、ランダム アクセスを可能にするためすべての配列がピンポン バッファとしてインプリメントされます。これらのバッファは、必要に応じてサイズ指定可能です。たとえば、タスクをバイパスする場合など、パフォーマンスが落ちることがあります。パフォーマンスへの影響を少なくするには、次のように STREAM 指示子を使用してこれらのバッファのサイズを増加し、プロデューサーとコンシューマーにスラックを追加します。

```
void top ( ... ) {
#pragma HLS dataflow
    int A[1024];
#pragma HLS stream off variable=A depth=3

    producer(A, B, ...); // producer writes A and B
    middle(B, C, ...);   // middle reads B and writes C
    consumer(A, C, ...); // consumer reads A and C
}
```

最上位関数インターフェイスの配列を `ap_fifo`、`axis`、または `ap_hs` インターフェイス タイプに設定すると、自動的にストリーミングとして設定されます。

FIFO がインプリメンテーションに必要な場合は、STREAM 指示子を使用してデザイン内ですべての配列をストリーミングとして指定する必要があります。

注記: STREAM 指示子を配列に適用すると、ハードウェアにインプリメントされる FIFO に配列と同じ数の要素が含まれます。FIFO のサイズを指定するには、`-depth` オプションを使用します。

STREAM 指示子を使用すると、`config_dataflow` コンフィギュレーションで指定したデフォルトのインプリメンテーションから DATAFLOW 領域の配列を変更することもできます。

- `config_dataflow default_channel` がピンポンとして設定される場合、配列に STREAM 指示子を適用すると、どの配列も FIFO としてインプリメントできます。

注記: FIFO インプリメンテーションを使用するためには、配列はストリーミング手法でアクセスされる必要があります。

- `config_dataflow default_channel` が FIFO に設定されるか、Vivado HLS で DATAFLOW 領域のデータがストリーミング手法でアクセスされると自動的に判断された場合、STREAM 指示子に `-off` オプションを付けて配列に適用すると、どの配列もピンポン インプリメンテーションとしてインプリメントできます。



重要: このアクセスを保持するには、`volatile` 修飾子を使用してコンパイラ最適化 (特にデッドコード削除) が実行されないようにする必要があります。

DATAFLOW 領域の配列がストリーミングとして指定され、FIFO としてインプリメントされる場合、FIFO に元の配列と同じ数の要素を含める必要は通常ありません。DATAFLOW 領域のタスクでは、各データ サンプルは使用可能になるとすぐに使用されます。`config_dataflow` コマンドに `-fifo_depth` オプションを指定するか、STREAM 指示子に `-depth` を指定すると、FIFO のサイズを最低限必要な要素数に設定して、データが一時停止しないようにできます。`-off` オプションを選択する場合は、`-off` オプションでピンポンの深さ (ブロック数) を設定します。深さは少なくとも 2 にする必要があります。

コンパイラ FIFO の深さの指定

開始の伝搬

コンパイラが自動的に開始[start FIFO]FIFO[start token]を作成して、開始トークンを内部プロセスまで伝搬することがあります。このような FIFO がパフォーマンスのボトルネックになることがあるので、その場合は次のコマンドを使用してデフォルト サイズの 2 を増加します。

```
config_dataflow -start_fifo_depth <value>
```

プロデューサーとコンシューマー間に制限のないスラックが必要で、内部プロセスが恒久的に実行され、入力および出力 (FIFO または PIPO) により安全に駆動される場合は、次のプラグマを使用して、指定したデータフロー領域のローカルでこれらの開始 FIFO をユーザーの責任において削除できます。

```
#pragma HLS DATAFLOW disable_start_propagation
```

スカラーの伝搬

コンパイラは、C/C++ コードからのスカラーの一部をスカラー[scale FIFOs]FIFO を介して自動的にプロセス間を伝搬します。このような FIFO がパフォーマンスのボトルネックになったり、デッドロックの原因となることがあるので、その場合は次のコマンドを使用してサイズを設定します (デフォルト値は `-fifo_depth` に設定)。

```
config_dataflow -scalar_fifo_depth <value>
```

stable 配列

`stable` プラグマを使用すると、データフロー領域の入力または出力変数をマークして、それらの該当する同期を削除できます (ユーザーがこの削除を本当に正しいと確認した場合のみ)。

```
void dataflow_region(int A[...], ...
#pragma HLS stable variable=A
#pragma HLS dataflow
    proc1(...);
    proc2(A, ...);
```

`stable` プラグマがなく、`A` が `proc2` で読み出される場合、`proc2` は、それが含まれるデータフロー領域で (`ap_start` を使用して) 初期同期の一部になります。つまり、`proc1` は `proc2` も再開可能な状態になるまでは再開されないで、データフロー反復が重複されなくなり、パフォーマンスが落ちてしまうことがあります。`stable` プラグマは、この同期が必ずしも正確性を保持するわけではないことを示しています。

前の例では、`stable` プラグマがない場合、`A` が `proc2` で読み出されるとすると、`proc2` がタスクをバイパスするので、パフォーマンスが落ちます。

`stable` プラグマがある場合、コンパイラは次のような判断をします。

- `A` が `proc2` で読み出される場合、`dataflow_region` の実行中に、読み出されるメモリ位置はほかのプロセスや呼び出したコンテキストで上書きされません。
- `A` が `proc2` で書き込まれる場合、書き込まれたメモリ位置は `dataflow_region` の実行中に、それらの定義前にほかのプロセスや呼び出したコンテキストによって読み出されません。

これは、データフロー領域がまだ開始していない場合や実行を終了していない場合にのみ、呼び出し元がこれらの変数をアップデートまたは読み出す場合によく使用されます。

データフロー内での `ap_ctrl_none` の使用

`ap_ctrl_none` ブロック レベル I/O プロトコルを使用すると、`ap_ctrl_hs` および `ap_ctrl_chain` プロトコルの柔軟性の低い同期スキームを使用せずに済みます。これらのプロトコルでは、領域内のプロセスすべてが同じ回数分実行され、その C ビヘイビアとより一致ようになります。

ただし、たとえば、作業を複数のより遅い作業に分散させてより頻繁に実行することで、プロセスを速めることを目的とすることがあります。

データフロー領域 (ループ内のデータフローを除く) では、次を指定可能です。

```
#pragma HLS interface ap_ctrl_none port=return
```

ただし、これは次の条件が満たされている場合にのみ可能です。

- 領域とそれに含まれるすべてのプロセスが FIFO (`hls::stream`、ストリーム配列、`AXIS`) を介してのみ通信され、メモリは介さない。
- 最上位デザインまでの領域の親すべてが、次の要件に従っている。
 - すべてがデータフロー領域である (ループ内のデータフローを除く)。
 - すべてが `ap_ctrl_none` を指定している。

つまり、次のものは階層内の `ap_ctrl_none` を使用するデータフロー領域の親にはなりません。

- シーケンシャルまたはパイプライン FSM
- `for` ループ (ループ内のデータフロー) 内のデータフロー領域

このプラグマを使用すると、その領域内のプロセスを同期するのに `ap_ctrl_chain` が使用されなくなります。これらは入力 FIFO のデータが使用できるかどうか、および出力 FIFO のスペースによって実行されたり、停止したりします。次に例を示します。

```
void region(...) {
#pragma HLS dataflow
#pragma HLS interface ap_ctrl_none port=return
    hls::stream<int> outStream1, outStream2;
    demux(inStream, outStream1, outStream2);
    worker1(outStream1, ...);
    worker2(outStream2, ...);
}
```

この例では、`demux` は `worker1` および `worker2` の 2 倍の頻度で実行できます。たとえば、`II=1` の設定で、`worker1` およ `worker2` に `II=2` を設定しても、グローバル `II=1` を達成できます。

注記:

- C シミュレーションが動作するようにするには、ノンブロッキング読み出しを少ない頻度で実効されるプロセス内で注意して使用する必要があることがあります。
- プラグマは、領域内の個別のプロセスではなく、領域に適用されます。
- デッドロック検出は協調シミュレーションでディセーブルにする必要があります。これは、[cosim_design](#) で説明されている `-disable_deadlock_detection` オプションを使用して設定できます。

レイテンシの最適化

レイテンシ制約の使用

Vivado HLS では、どのスコープでもレイテンシ制約の使用がサポートされます。レイテンシ制約は `LATENCY` 指示子を使用して指定します。

最大または最小 `LATENCY` 制約をスコープに設定すると、Vivado HLS で関数のすべての演算が指定のクロック サイクル内に完了するようになります。

ループに適用されたレイテンシ指示子は、ループの 1 回の反復に必要なレイテンシを指定します。つまり、次の例のように、ループ ボディのレイテンシを指定します。

```
Loop_A: for (i=0; i<N; i++) {
#pragma HLS latency max=10
    ..Loop Body...
}
```

すべてのループ反復のレイテンシ合計を制限するのが目的の場合は、次の例のようにループ全体を含む領域にレイテンシ指示子を指定する必要があります。

```
Region_All_Loop_A: {
#pragma HLS latency max=10
Loop_A: for (i=0; i<N; i++)
{
    ..Loop Body...
}
}
```

この場合、ループが展開されていたとしても、レイテンシ指示子ですべてのループ演算の最大制限が設定されます。

Vivado HLS で最大レイテンシ制約を満たすことができない場合、レイテンシ制約が緩和され、できるだけ最適な結果が達成されます。

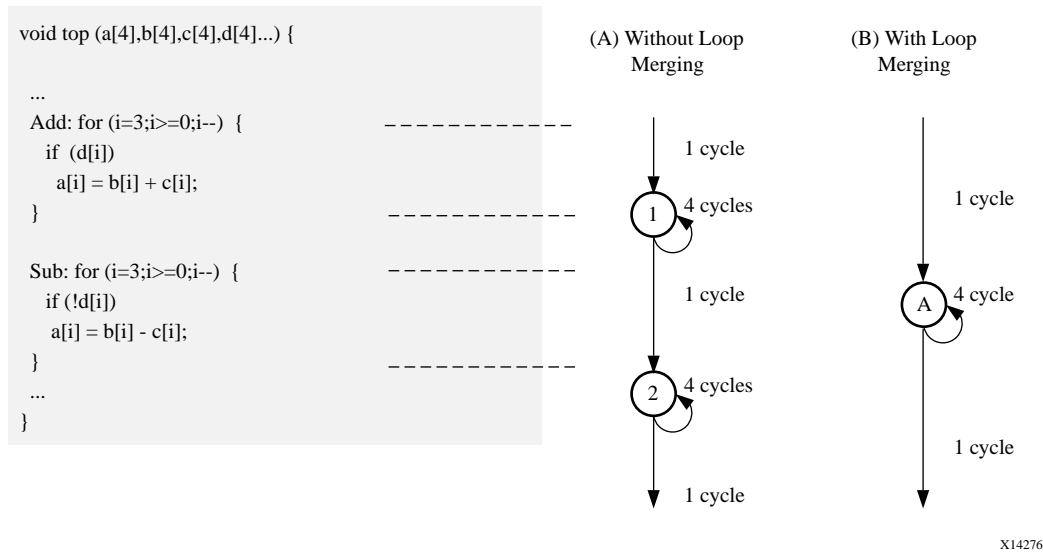
最小レイテンシ制約を設定し、Vivado HLS で必要とされる最小値よりも短いレイテンシでデザインが生成される場合は、最小レイテンシが満たされるようにダミー クロック サイクルが挿入されます。

レイテンシを削減するためのシーケンシャル ループの結合

非展開ループからは、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、余分なクロック サイクルが追加されるので、これ以上の最適化ができなくなります。

次の図は、簡潔に見えるコーディング スタイルが RTL デザインのパフォーマンスに悪影響を与えている例を示しています。

図 66: ループ指示子



前の図の (A) は、デザインの非展開ループそれぞれで少なくとも FSM の 1 ステートが作成されるところを示しています。このようなステート間の移行では、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。

この単純な例の場合は、Add ループの else 分岐でも問題を解決できることは明らかですが、より複雑なコードでは簡単にはわからないことあるので、よりわかりやすいコーディング スタイルを使用した方が有益です。

LOOP_MERGE 最適化指示子を使用すると、ループが自動的に結合されるようになります。LOOP_MERGE 指示子は、適用されたスコープ内のループすべてを結合しようとします。上記の例の場合、ループの結合により、終了するのに 6 クロックしか必要のない前の図の (B) のような制御構造が作成されます。

ループを結合すると、ループ内のロジックが一緒に最適化されるようになります。上記の例の場合、デュアルポートブロック RAM を使用すると、加算および乗算が並列で実行できるようになります。

現時点では、Vivado HLS でのループ統合には、次のような制限があります。

- 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。
- ループの境界が定数の場合、最大定数値が結合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ループを結合することはできません。
- 結合されるループの間のコードによる悪影響がないことを確認し、このコードを複数回実行したときに同じ結果になるようにします (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 アクセスが含まれている場合は、ループどうしを結合できません。ループを結合すると、FIFO からの読み出しおよび書き込みの順序が変更されることがあります。

入れ子のループの平坦化によるレイテンシの改善

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 (入れ子) になったループ間を移動するためには追加のクロック サイクルが必要になります。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。

ここに示す小さい例の場合、これはループ `Outer` を実行するのに余分のクロック サイクルが 200 かかることを意味します。

```
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
}
...
}
```

Vivado HLS で提供されている `set_directive_loop_flatten` コマンドを使用すると、ラベル付きの完全な入れ子のループとほぼ完全な入れ子のループを平坦にできるので、最適なハードウェア パフォーマンスにするためにコードを記述し直さなくても、ループ内の演算を実行するのにかかるサイクル数を削減できます。

- 完全ループ ネスト: 一番内側のループのみにループ ボディがあり、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。
- 半完全ループ ネスト: 一番内側のループのみにループ ボディがあり、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可変であったり、ループ本体が一番内側のループにのみ含まれているとは限らないような不完全なループ ネストの場合は、コードの構造を変更するか、ループ ボディのループを展開して、完全なループ ネストを作成してみてください。

入れ子のループに指示子を適用する際は、ループ ボディを含む一番内側のループに適用する必要があります。

```
set_directive_loop_flatten top/Inner
```

ループの平坦化は、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。個々のループに設定できるほか、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することもできます。

エリアの最適化

データ型およびビット幅

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LEMENT のサイズと演算子に直接影響します。変数に 12 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、開始間隔 (II) およびレイテンシ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。大きすぎる ELEMENT があると、ハードウェア リソースでエリアが無駄になります。
- 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。

関数のインライン展開

関数をインライン展開すると、関数階層が削除されます。関数をインライン展開するには、`INLINE` 指示子を使用します。関数をインライン展開すると、関数内でコンポーネントをより共有できたり、呼び出し関数のロジックで最適化できるので、エリアが削減されることがあります。このタイプの関数のインライン展開も Vivado HLS で自動的に実行されます。小型の関数は自動的にインライン展開されます。

インライン展開により、関数の共有を制御しやすくなります。関数が共有されるには、それらが同じ階層レベル内で使用される必要があります。このコード例の場合、`foo_top` 関数により `foo` が 2 回呼び出され、`foo_sub` 関数が呼び出されます。

```
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);
    ...
}
```

`foo_sub` 関数をインライン展開し、`ALLOCATION` 指示子を使用して `foo` 関数の 1 つのインスタンスのみを指定すると、`foo` 関数の 1 つのインスタンスのみを含むデザイン (上記の例の 1/3 のエリア) になります。

```
foo_sub (p, q) {
    #pragma HLS INLINE
    int q1 = q + 10;
    foo(p1,q); // foo_3
    ...
}
void foo_top { a, b, c, d} {
```

```
#pragma HLS ALLOCATION instances=foo limit=1 function
...
foo(a,b); //foo_1
foo(a,c); //foo_2
foo_sub(a,d);
...
}
```

INLINE 指示子を `recursive` オプションを指定して使用すると、指定した関数の下の関数がすべてインライン展開されます。`recursive` オプションを最上位関数に使用すると、デザイン内のすべての関数階層が削除されます。

INLINE の `off` オプションを使用すると、関数がインライン展開されないようにできます。このオプションを使用し、Vivado HLS で関数が自動的にインライン展開されるのを回避できます。

INLINE 指示子は、ソース コード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、最適なアーキテクチャを探す効果的な手法として使用できます。

多くの配列の 1 つの大きな配列へのマップ

C コード内に小さい配列が多くある場合は、それらを 1 つの大きな配列にマップすると、必要なブロック RAM の数が通常削減されます。

各配列は、ブロック RAM または UltraRAM (デバイスでサポートされる場合) にマップされます。FPGA で提供される基本的なブロック RAM の単位は 18K です。多くの小さい配列で 18K がフルで使用されない場合、小さい配列を 1 つの大きな配列にマップすることでブロック RAM リソースを効率的に使用できます。ブロック RAM が 18K を超える場合は、それらが複数の 18K 単位に自動的にマップされます。デザイン内のブロック RAM については、合成レポートの [Utilization Report] → [Details] → [Memory] を参照してください。

ARRAY_MAP 指示子では、複数の小さい配列を 1 つの大きい配列にマップするために次の 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。

水平配列マップ

次のコード例の場合、2 つの配列が含まれており、それらが 2 つの RAM コンポーネントにマップされます。

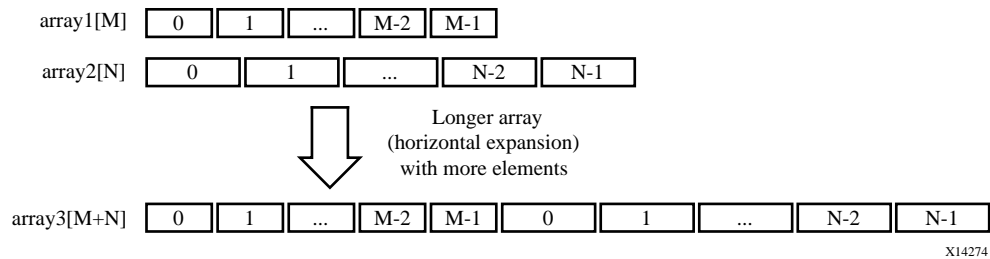
```
void foo (...) {
    int8  array1[M];
    int12 array2[N];
    ...
loop_1: for(i=0;i<M;i++) {
    array1[i] = ...;
    array2[i] = ...;
    ...
}
...
}
```


次の例では、配列 `array1` および `array2` が `array3` として指定した 1 つの配列にまとめられます。

```
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    #pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
    #pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
    ...
    loop_1: for(i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

この例の場合、次の図のように、ARRAY_MAP 指示子で配列が変更されています。

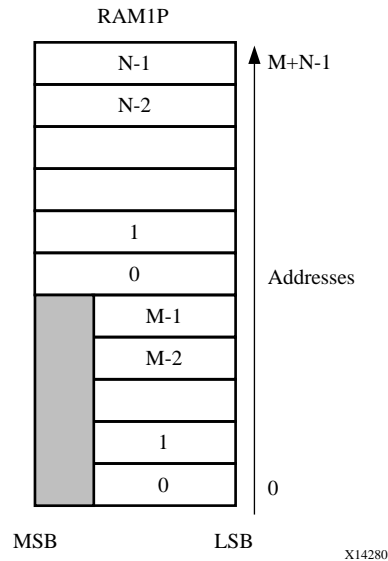
図 67: 水平マップ



水平マップを使用すると、複数の小さい配列が 1 つの大きい配列にマップされます。マップは大きい配列のロケーション 0 から開始され、コマンドが指定した順序通りにマップされます。Vivado HLS の GUI では、これはメニューコマンドを使用して指定した配列順序になります。Tcl 環境では、実行したコマンドの順序になります。

次の図に示すような垂直マップを使用する場合、ブロック RAM のインプリメンテーションは、次の図のようになります。

図 68: 水平マップのメモリ

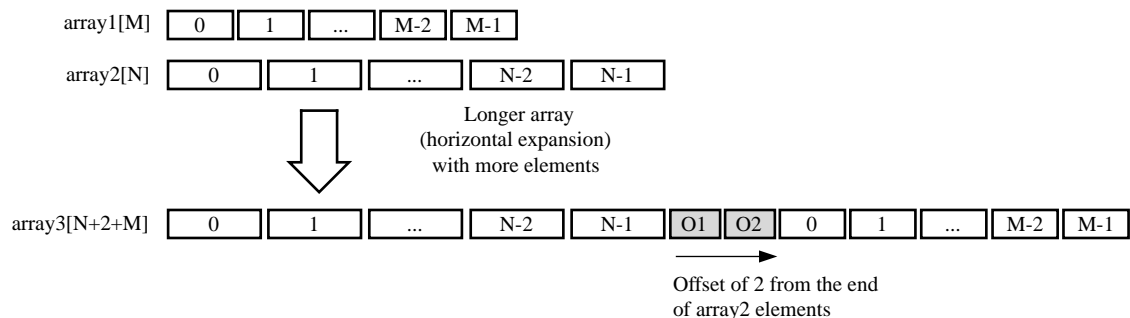


ARRAY_MAP 指示子に `offset` オプションを使用すると、`horizontal` オプションを使用したときに追加される位置の連続した配列を指定できます。上記の例と順序を逆にし (array2 を指定してから array1 を指定)、`offset` を追加すると、次のようになります。

```
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    #pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
    #pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal offset=2
    ...
    loop_1: for(i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

これにより、次の図に示すような結果になります。

図 69: オフセットを含む水平マップ



X14273

マップ後に、新しいインスタンスにマップした変数のいずれかに RESOURCE 指示子を適用すると、上記の例の新しく作成された配列 `array3` が特定のブロック RAM または UltraRAM にマップできます。

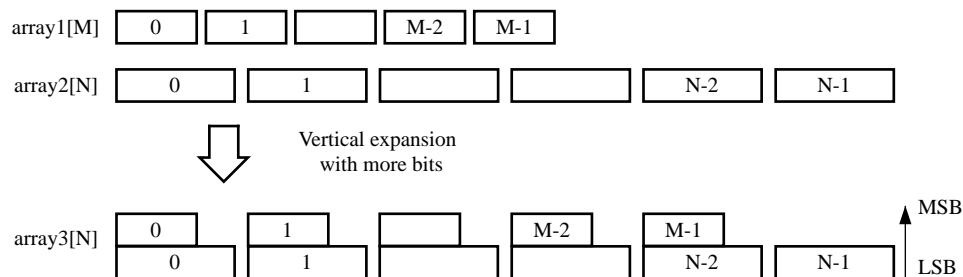
水平マップによりブロック RAM コンポーネントの数は削減できるので、エリアは改善されますが、ブロック RAM ポート数は減るので、スループットおよびパフォーマンスに影響する可能性があります。この制限を解消するため、Vivado HLS では垂直マップがサポートされています。

垂直配列のマップ

垂直マップの場合、配列は連結されて、より高い帯域幅の配列が生成されます。垂直マップは、`INLINE` 指示子に垂直オプションを使用すると適用されます。次の図は、前述の例に垂直マップを適用した結果を示しています。

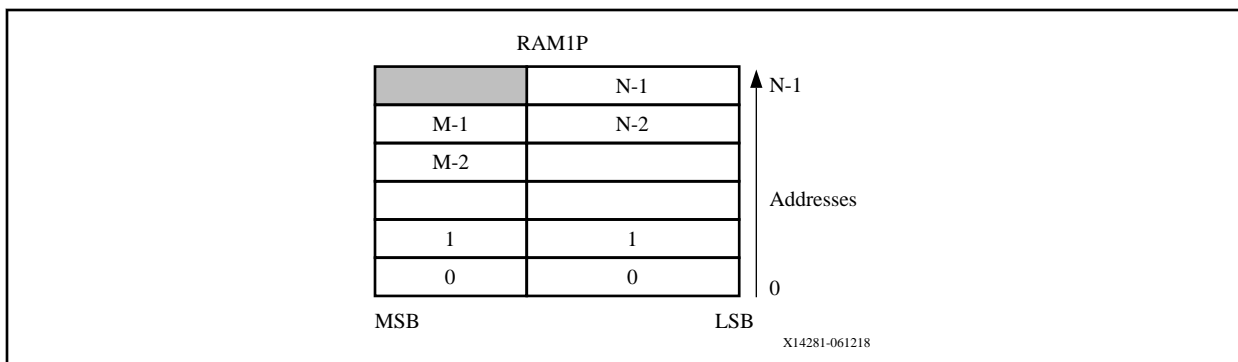
```
void foo (...) {
    int8  array1[M];
    int12 array2[N];
    #pragma HLS ARRAY_MAP variable=array2 instance=array3 vertical
    #pragma HLS ARRAY_MAP variable=array1 instance=array3 vertical
    ...
    loop_1: for(i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

図 70: 垂直マップ



XI4312

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 から開始され、最後の配列が MSB で終了されます。垂直マップの後には、次の図のように新しく作成された配列が 1 つのブロック RAM コンポーネントにインプリメントされます。



配列のマッピングおよび注意事項



重要: 配列変換のオブジェクトは、適用されるその他の指示子よりも前にソース コードに含めておく必要があります。

`horizontal` マッピングを使用して分割された配列からの要素を 1 つの配列にマッピングするには、分割される配列の個別の要素を `ARRAY_MAP` 指示子で指定する必要があります。たとえば、次の Tcl コマンドは `accum` 配列を分割し、分割された要素と一緒にマッピングします。

```
#pragma HLS array_partition variable=m_accum cyclic factor=2 dim=1
#pragma HLS array_partition variable=v_accum cyclic factor=2 dim=1
#pragma HLS array_map variable=m_accum[0] instance=_accum horizontal
#pragma HLS array_map variable=v_accum[0] instance=mv_accum horizontal
#pragma HLS array_map variable=m_accum[1] instance=mv_accum_1 horizontal
#pragma HLS array_map variable=v_accum[1] instance=mv_accum_1 horizontal
```

グローバル配列もマッピングできますが、結果の配列インスタンスがグローバルになり、同じ配列インスタンスにマッピング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ピングされると、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。

配列の関数引数は、それらが同じ関数に対する引数の場合にのみマッピングされることがあります。

配列の形状変更

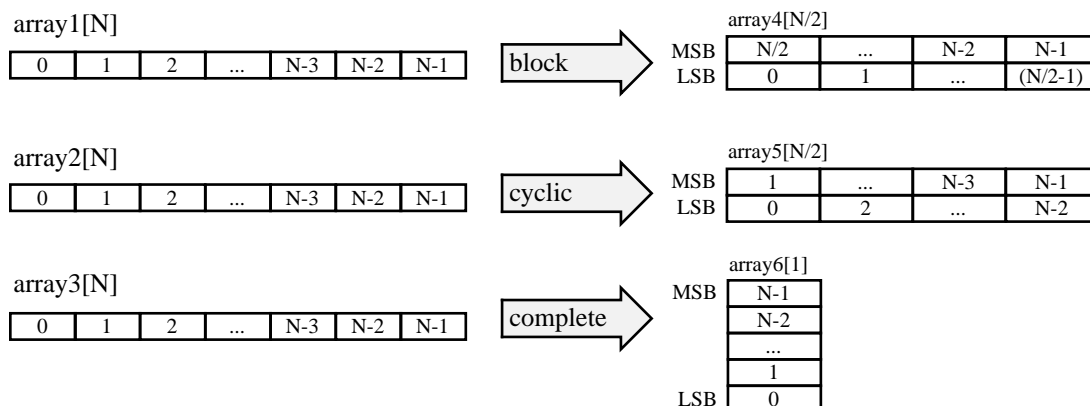
`ARRAY_RESHAPE` 指示子は、`ARRAY_MAP` の垂直モードと `ARRAY_PARTITIONING` を組み合わせ、データに並列アクセスできる分割属性の利点を活かしながら、ブロック RAM 数を削減できます。

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];
    #pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
    ...
}
```

`ARRAY_RESHAPE` 指示子では配列が次の図に示すように変形されます。

図 71: 配列の形状変更



X14307

ARRAY_RESHAPE 指示子を使用すると、1 クロック サイクルでより多くのデータにアクセスできるようになります。より多くのデータが 1 クロック サイクルでアクセスできる場合、このデータを消費するループを展開することでスループットが改善される場合にのみ、Vivado HLS でこれらのループが自動的に展開されます。ループを完全または部分展開すると、1 クロック サイクルでこれらの追加データを消費するのに十分なハードウェアを作成できます。この機能は `config_unroll` コマンドと `tripcount_threshold` オプションを使用して制御されます。次の例では、展開するとスループットが改善される場合にのみトリップカウントが 16 未満のループが自動的に展開されます。

```
config_unroll -tripcount_threshold 16
```

関数インスタンス化 (FUNCTION_INSTANTIATE)

関数のインスタンス化とは、関数の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単純になり、レイテンシおよびスループットを向上できる可能性があります。

FUNCTION_INSTANTIATE 指示子を使用すると、関数が呼び出されたときに関数の一部の入力値が定数値であることがあるという事実を利用して、周辺の制御構造が簡略化され、より最適化されたより小さい関数ブロックが作成されます。これについて、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 foo_sub(bool mode){
    #pragma HLS FUNCTION_INSTANTIATE variable=mode
    if (mode) {
        // code segment 1
    } else {
        // code segment 2
    }
}

void foo(){
    #pragma HLS FUNCTION_INSTANTIATE variable=select
    foo_sub(true);
    foo_sub(false);
}
```

`foo_sub` 関数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、実行される内容が異なります。`foo_sub` 関数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複雑になってしまいます。

`FUNCTION_INSTANTIATE` 最適化を使用すると、各インスタンスを個別に最適化できるので、機能およびエリアを削減できます。`FUNCTION_INSTANTIATE` 最適化を実行すると、上記のコードは 2 つの個別の関数に変換され、それぞれを次に示すように異なるモード値用に最適化できます。

```
void foo_sub1() {
    // code segment 1
}

void foo_sub1() {
    // code segment 2
}

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

関数が異なる階層レベルで使用されていて、インライン展開またはコード変更を広範囲で実行しないと関数の共有が困難な場合、`FUNCTION_INSTANTIATE` を使用するのがエリアを削減する最適な方法となり得ます。ローカルで最適化された小さなコピーが多数ある方が、共有できない大きなコピーが多数あるよりも効率的です。

ハードウェア リソースの制御

Vivado HLS の合成では、次の基本的なタスクが実行されます。

- 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。
演算子で、加算、乗算、配列の読み出しおよび書き込みなど C コードの演算が示されます。
- 次に、ハードウェア演算をインプリメントする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 (加算器、乗算器、パイプライン乗算器、ブロック RAM など) です。

各段階ごとに制御が提供されるので、ハードウェア インプリメンテーションをより粒度の精度が高いレベルで制御できます。

演算子数の制限

デフォルトでは Vivado HLS はまず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、明示的に演算子の数を制限してエリア削減する必要があります。デザインの演算子数を制限する方法はエリア削減には効率的な方法で、演算を共有させるとエリアが削減しやすくなります。

ALLOCATION 指示子を使用すると、演算子、コア、または関数がデザインで使用される数を制限できます。たとえば、foo というデザインには 317 個の乗算が含まれるのに、FPGA には 256 個の乗算器リソース (DSP48) しかないとします。次に示す ALLOCATION 指示子は、最大 256 の乗算 (mul) 演算子を含むデザインを作成するよう、Vivado HLS に指示します。

```
dout_t array_arith (dio_t d[317]) {
    static int acc;
    int i;
    #pragma HLS ALLOCATION instances=mul limit=256 operation

    for (i=0;i<317;i++) {
        #pragma HLS UNROLL
        acc += acc * d[i];
    }
    rerun acc;
}
```

注記: ALLOCATION 制限を必要な数よりも大きな値に指定すると、Vivado HLS ではその制限で指定されたリソース数を使用するか、必要な最大数を使用しようとするので、共有される量が減ります。

type オプションを使用すると、ALLOCATION 指示子で演算、コア、または関数を制限するかどうかを指定できます。次の表に、ALLOCATION 指示子を使用して制御可能なすべての演算をリストします。

表 14: Vivado HLS の演算子

演算子	説明
add	整数の加算
ashr	四則演算右シフト
dadd	倍精度浮動小数点の加算
dcmp	倍精度浮動小数点の比較
ddiv	倍精度浮動小数点の除算
dmul	倍精度浮動小数点の乗算
drecip	倍精度浮動小数点の逆数
drem	倍精度浮動小数点の剰余
drsqrt	倍精度浮動小数点の逆数平方根
dsub	倍精度浮動小数点の減算
dsqrt	倍精度浮動小数点の平方根
fadd	単精度浮動小数点の加算
fcmp	単精度浮動小数点の比較
fdiv	単精度浮動小数点の除算
fmul	単精度浮動小数点の乗算
frecip	単精度浮動小数点の逆数
frem	単精度浮動小数点の剰余
frsqrt	単精度浮動小数点の逆数平方根
fsub	単精度浮動小数点の減算
fsqrt	単精度浮動小数点の平方根
icmp	整数の比較
lshr	論理演算右シフト
mul	乗算

表 14: Vivado HLS の演算子 (続き)

演算子	説明
sdiv	符号付き除算
shl	左シフト
srem	符号付き剰余
sub	減算
udiv	符号なし除算
urem	符号なし剰余

グローバルに演算子を最小限にする方法

ALLOCATION 指示子は、ほかの指示子と同様、スコープ (関数、ループ、領域) 内で指定されます。config_bind コンフィギュレーションを使用すると、デザイン全体で演算子が最小限に抑えられます。

デザイン全体で演算子を最小限に抑えるには、config_bind コンフィギュレーションに min_op オプションを付けて実行します。前の表の演算子はすべてこの方法で制限できます。

コンフィギュレーションを適用すると、そのソリューションで実行されるすべての合成動作に適用されます。ソリューションをいったん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動作すべてに適用されます。

config_bind コンフィギュレーションの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か、open_solution -reset を使用してソリューションを開くと、削除できます。

ハードウェア コアの制御

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 デバイスにより指定された遅延、ユーザーの指定した指示子が使用され、演算子をインプリメントするのにどのコアを使用するかが決定されます。たとえば、乗算器演算をインプリメントする場合、Vivado HLS で組み合わせ乗算器コアが使用されることもあれば、パイプライン乗算器コアが使用されることもあります。

合成中に演算子にマップされるコアは、それらの演算子と同じ方法で制限できます。乗算演算子の総数を制限する代わりに、組み合わせ乗算器コアの数を制限して、残りの乗算がパイプライン乗算器を使用して実行されるようにすることもできます (逆も可能)。これは、ALLOCATION 指示子の type オプションに core を指定すると実行できます。

RESOURCE 指示子を使用すると、特定の演算にどのコアを使用するか明示的に指定できます。次の例の場合、2 段のパイプライン乗算器を指定して変数の乗算がインプリメントされるようになります。次のコマンドでは、変数 c に対して 2 段のパイプライン乗算器を使用することが Vivado HLS に伝えられます。変数 d にどのコアを使用するかは、Vivado HLS で決定されます。

```
int foo (int a, int b) {
    int c, d;
    #pragma HLS RESOURCE variable=c latency=2
    c = a*b;
    d = a*c;

    return d;
}
```

次の例では、RESOURCE 指示子で temp 変数の add 演算が指定され、AddSub_DSP コアを使用してインプリメントされています。これにより、最終的なデザインで演算が DSP48 プリミティブを使用してインプリメントされるようになります (デフォルトでは、加算演算は LUT を使用してインプリメントされます)。

```
void apint_arith(dinA_t inA, dinB_t inB,
                dout1_t *out1
                ) {

    dout2_t temp;
    #pragma HLS RESOURCE variable=temp core=AddSub_DSP

    temp = inB + inA;
    *out1 = temp;

}
```

list_core コマンドを使用すると、ライブラリで使用可能なコアの詳細を表示できます。list_core は Tcl コマンド インターフェイスでのみ使用でき、デバイスは set_part コマンドを使用して指定する必要があります。デバイスを指定しない場合、コマンドは実行されません。

list_core コマンドに -operation オプションを使用すると、指定した演算でインプリメント可能なライブラリ内のすべてのコアがリストされます。次の表は、標準 RTL 論理演算 (加算、乗算、比較など) をインプリメントするために使用するコアをリストしています。

表 15: 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。Vivado HLS で必要なパイプライン段数が決定されます。
AddSub_DSP	DSP48 を使用して add または sub 演算がインプリメントされるようにします (DSP48 内の加算器または減算器が使用されます)。
DivnS	N 段のパイプライン除算器。
DSP48	1 つの DSP48 マクロセルにインプリメンテーションできるようにするビット幅を使用した乗算。これには、パイプライン乗算と前置加算器または後置加算器 (もしくはその両方) がまとめられた乗算を含めることができます。このコアは、最大レイテンシ 4 でのみパイプラインできます。4 を超える値は、4 で飽和します。
Mul	標準 DSP48 マクロセルのサイズを超えるビット幅の組み合わせ乗算器。 1 つの DSP48 マクロセルを使用してインプリメント可能な乗算器は DSP48 コアにマップされます。
MulnS	標準 DSP48 マクロセルのサイズを超えるビット幅の N 段のパイプライン乗算器。 10 ビット以上の乗算は、DSP48 マクロセルにインプリメントされます。これ以下のビット数の乗算は、LUT を使用してインプリメントされます。1 つの DSP48 マクロセルを使用してインプリメント可能な乗算器は DSP48 コアにマップされます。
Mul_LUT	LUT を使用してインプリメントされる乗算器。 注記: これは C POD (Plain Old Data) 型にのみ適用されます。Vivado HLS 型 (ap_int、ap_fixed など) と一緒に使用できません。

演算で浮動小数点型が使用される場合は、標準的なコアのほかに、次の浮動小数点コアが使用されます。浮動小数点コアがデバイスでサポートされているかどうかは、各デバイスの資料を参照してください。

表 16: 浮動小数点コア

コア	説明
FAddSub_nodsp	DSP48 プリミティブなしでインプリメントされる浮動小数点加算器または減算器。
FAddSub_fulldsp	DSP48 プリミティブだけを使用してインプリメントされる浮動小数点加算器または減算器。
FDiv	浮動小数点除算器。
FExp_nodsp	DSP48 プリミティブなしでインプリメントされる浮動小数点指数演算。
FExp_meddsp	DSP48 プリミティブが調整されてインプリメントされる浮動小数点指数演算。
FExp_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる浮動小数点指数演算。
FLog_nodsp	DSP48 プリミティブなしでインプリメントされる浮動小数点対数演算。
FLog_meddsp	DSP48 プリミティブが調整される浮動小数点対数演算。
FLog_fulldsp	DSP48 プリミティブのみが使用される浮動小数点対数演算。
FMul_nodsp	DSP48 プリミティブなしでインプリメントされる浮動小数点乗算器。
FMul_meddsp	DSP48 プリミティブが調整されてインプリメントされる浮動小数点乗算器。
FMul_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる浮動小数点乗算器。
FMul_maxdsp	最大数の DSP48 プリミティブがインプリメントされる浮動小数点乗算器。
FRSqrt_nodsp	DSP48 プリミティブありでインプリメントされる浮動小数点逆数平方根。
FRSqrt_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる浮動小数点逆数平方根。
FRecip_nodsp	DSP48 プリミティブなしでインプリメントされる浮動小数点逆数。
FRecip_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる浮動小数点逆数。
FSqrt	浮動小数点平方根。
DAddSub_nodsp	DSP48 プリミティブなしでインプリメントされる倍精度の浮動小数点加算器または減算器。
DAddSub_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる倍精度の浮動小数点加算器または減算器。
DDiv	倍精度の浮動小数点除算器。
DExp_nodsp	DSP48 プリミティブなしでインプリメントされる倍精度の浮動小数点指数演算。
DExp_meddsp	DSP48 プリミティブが調整されてインプリメントされる倍精度の浮動小数点指数演算。
DExp_fulldsp	DSP48 プリミティブのみでインプリメントされる倍精度の浮動小数点指数演算。
DLog_nodsp	DSP48 プリミティブなしでインプリメントされる倍精度の浮動小数点対数演算。
DLog_meddsp	DSP48 プリミティブが調整される倍精度の浮動小数点対数演算。
DLog_fulldsp	DSP48 プリミティブのみを使用した倍精度の浮動小数点対数演算。
DMul_nodsp	DSP48 プリミティブなしでインプリメントされる倍精度の浮動小数点乗算器。
DMul_meddsp	DSP48 プリミティブが調整されてインプリメントされる倍精度の浮動小数点乗算器。
DMul_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる倍精度の浮動小数点乗算器。
DMul_maxdsp	最大数の DSP48 プリミティブがインプリメントされる倍精度の浮動小数点乗算器。
DRSqrt	倍精度の浮動小数点逆数平方根。
DRecip	倍精度の浮動小数点逆数。
DSqrt	倍精度の浮動小数点平方根。
HAddSub_nodsp	DSP48 プリミティブなしでインプリメントされる半精度の浮動小数点加算器または減算器。

表 16: 浮動小数点コア (続き)

コア	説明
HDiv	半精度の浮動小数点除算器。
HMul_nodsp	DSP48 プリミティブなしでインプリメントされる半精度の浮動小数点乗算器。
HMul_fulldsp	DSP48 プリミティブのみを使用してインプリメントされる半精度の浮動小数点乗算器。
HMul_maxdsp	最大数の DSP48 プリミティブがインプリメントされる半精度の浮動小数点乗算器。
HSqrt	半精度浮動小数点平方根。

次の表には、レジスタやメモリなどのストレージ エLEMENTをインプリメントするコアがリストされています。

表 17: ストレージ コア

コア	説明
FIFO	FIFO。ブロック RAM または分散 RAM のどちらを使用して RTL にインプリメントされるかは Vivado HLS で決定されます。
FIFO_BRAM	ブロック RAM 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。ブロック RAM または分散 RAM のどちらを使用して RTL にインプリメントされるかは Vivado HLS で決定されます。
RAM_1P_BRAM	ブロック RAM を使用してインプリメントされるシングル ポート RAM。
RAM_1P_LUTRAM	分散 RAM としてインプリメントされるシングル ポート RAM。
RAM_1P_URAM	Ultra RAM を使用してインプリメントされるシングル ポート RAM。
RAM_2P	1 つのポートで読み出し、もう 1 つのポートで読み出しと書き込みの両方を実行可能なデュアル ポート RAM。ブロック RAM または分散 RAM のどちらを使用して RTL にインプリメントされるかは Vivado HLS で決定されます。
RAM_2P_BRAM	1 つのポートで読み出し、もう 1 つのポートで読み出しと書き込みの両方を実行可能なブロック RAM を使用してインプリメントされたデュアル ポート RAM。
RAM_2P_LUTRAM	1 つのポートで読み出し、もう 1 つのポートで読み出しと書き込みの両方を実行可能な分散 RAM を使用してインプリメントされたデュアル ポート RAM。
RAM_2P_URAM	1 つのポートで読み出し、もう 1 つのポートで読み出しと書き込みの両方を実行可能な Ultra RAM を使用してインプリメントされたデュアル ポート RAM。
RAM_S2P_BRAM	1 つのポートで読み出し、もう 1 つのポートで書き込みを実行可能なブロック RAM を使用してインプリメントされたデュアル ポート RAM。
RAM_S2P_LUTRAM	1 つのポートで読み出し、もう 1 つのポートで書き込みを実行可能な分散 RAM としてインプリメントされたデュアル ポート RAM。
RAM_S2P_URAM	1 つのポートで読み出し、もう 1 つのポートで書き込みを実行可能な Ultra RAM を使用してインプリメントされたデュアル ポート RAM。
RAM_T2P_BRAM	ブロック RAM を使用してインプリメントされた両方のポートで読み出しと書き込みの両方をサポートする真のデュアル ポート RAM。
RAM_T2P_URAM	Ultra RAM を使用してインプリメントされた両方のポートで読み出しと書き込みの両方をサポートする真のデュアル ポート RAM。
ROM_1P	シングル ポート ROM。ブロック RAM または LUT のどちらを使用して RTL にインプリメントされるかは Vivado HLS で決定されます。
ROM_1P_BRAM	ブロック RAM を使用してインプリメントされたシングル ポート ROM。

表 17: ストレージ コア (続き)

コア	説明
ROM_nP_BRAM	ブロック RAM を使用してインプリメントされたマルチ ポート ROM。ポート数は Vivado HLS で自動的に決定されます。
ROM_1P_LUTRAM	分散 RAM を使用してインプリメントされたシングル ポート ROM。
ROM_nP_LUTRAM	分散 RAM を使用してインプリメントされたマルチ ポート ROM。ポート数は Vivado HLS で自動的に決定されます。
ROM_2P	デュアル ポート ROM。ブロック ROM または分散 ROM のどちらを使用して RTL にインプリメントされるかは Vivado HLS で決定されます。
ROM_2P_BRAM	ブロック RAM を使用してインプリメントされたデュアル ポート ROM。
ROM_2P_LUTRAM	分散 ROM としてインプリメントされたデュアル ポート ROM。

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。たとえば、次のコードでは RESOURCE 指示子で `out1` の乗算に 3 段のパイプライン乗算器が使用されてインプリメントされています。

```
void foo(...) {
#pragma HLS RESOURCE variable=out1 latency=3

// Basic arithmetic operations
*out1 = inA * inB;
*out2 = inB + inA;
*out3 = inC / inA;
*out4 = inD % inA;
}
```

代入で複数の同じ演算子が指定されている場合、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、この例の最初の乗算 (`inA * inB`) がパイプライン乗算器を使用してインプリメントされる場合は、次のようになります。

```
*out1 = inA * inB * inC;
```

このコードは、`Result_tmp` 変数に指定した指示子を使用して次のように変更する必要があります。

```
#pragma HLS RESOURCE variable=Result_tmp latency=3
Result_tmp = inA * inB;
*out1 = Result_tmp * inC;
```

ハードウェア コアのグローバル最適化

`config_bind` コンフィギュレーションでは、バインディング プロセスが制御されます。このコンフィギュレーションでは、コアを演算子にバインディングする際にどれくらいのエフォートを使用するのか指示できます。Vivado HLS では、デフォルトでタイミングおよびエリア間で最適なバランスになるコアが選択されます。`config_bind` は、どの演算子が使用されるかにも影響します。

```
config_bind -effort [low | medium | high] -min_op <list>
```

`config_bind` コマンドは、アクティブ ソリューション内でのみ使用できます。このバインディングのデフォルトの `run` ストラテジは `medium` です。

- low エフォート: タイミング共有が少なくなり、ランタイムは速くなりますが、最終的な RTL が大きくなってしまいう可能性があります。共有の可能性がほとんどないとわかっている場合や、共有の必要がなく、可能性を探すために CPU サイクルを無駄にしたい場合などに便利です。
- medium エフォート: Vivado HLS のデフォルトの設定で、演算を共有しようとしませんが、最終的に合理的な時間で終わるようになります。
- high エフォート: 最大限に共有しようとし、ランタイムにも制限がありません。Vivado HLS は可能性のある組み合わせがすべて確認されるまで続行されます。

ロジックの最適化

演算子のパイプラインの制御

Vivado HLS では、内部演算に使用するパイプライン レベルが自動的に決定されます。RESOURCE 指示子に `-latency` オプションを付けると、パイプライン段数を明示的に指定して、Vivado HLS で決定された数を上書きできます。

RTL 合成では、配置配線後に発生する可能性のあるタイミング問題を改善しやすいように、追加でパイプライン レジスタが使用されることがあります。通常は演算の出力にレジスタを追加すると、出力データパスのタイミングが改善しやすくなります。演算の入力にレジスタを追加すると、入力データパスと FSM からの制御ロジックの両方のタイミングが改善しやすくなります。

パイプライン段を追加する際の規則は次のとおりです。

- 指定したレイテンシが Vivado HLS で決定されたレイテンシよりも 1 サイクル長い場合、Vivado HLS は演算の出力に新しい出力レジスタを追加します。
- 指定したレイテンシが Vivado HLS で決定されたレイテンシよりも 2 サイクル長い場合、Vivado HLS は演算の出力と演算の入力に新しいレジスタを追加します。
- レイテンシが Vivado HLS で決定されたレイテンシよりも 3 サイクルまたはそれ以上指定される場合、Vivado HLS は演算の出力と演算の入力側に新しい出力レジスタを追加します。Vivado HLS は、追加レジスタの位置を自動的に決定します。

デザインで使用されている特定コアのすべてのインスタンスで、深さの同じパイプラインをパイプライン処理する場合、`config_core` コンフィギュレーションを使用できます。このコンフィギュレーションを設定するには、次を実行します。

1. [Solutions] → [Solution Settings] をクリックします。
2. [Solution Settings] ダイアログ ボックスで [General] カテゴリを選択し、[Add] をクリックします。
3. [Add Command] ダイアログ ボックで `config_core` コマンドを選択して、パラメーターを指定します。

たとえば、次のコンフィギュレーションは、DSP48 コアでインプリメントされているすべての操作がレイテンシ 3 でパイプライン処理されることを指定しています。レイテンシ 3 はこのコアでの最大レイテンシです。

```
config_core DSP48 -latency 3
```

次のコンフィギュレーションは、RAM_1P_BRAM コアでインプリメントされているすべてのブロック RAM がレイテンシ 3 でパイプライン処理されることを指定しています。

```
config_core RAM_1P_BRAM -latency 3
```



重要: Vivado HLS では、明示的な RESOURCE 指示子 (配列をインプリメントするために使用するコアを指定) を使用して、コア コンフィギュレーションがブロック RAM に適用されます。配列がデフォルト コアを使用してインプリメントされている場合は、このコア コンフィギュレーションを使用してもブロック RAM には影響しません。

論理演算式の最適化

合成中は、駆動電流の削減やビット幅の最小化など、複数の最適化が実行されます。自動最適化には、演算式バランス調整も含まれます。

演算式バランス調整では、演算子を並べ替えてバランスの取れたツリーを構築することにより、レイテンシを削減します。

- 整数演算の演算式バランス調整はデフォルトでオンになっていますが、オフにすることもできます。
- 浮動小数点演算では、演算式バランス調整はデフォルトでオフになっていますが、オンにすることも可能です。

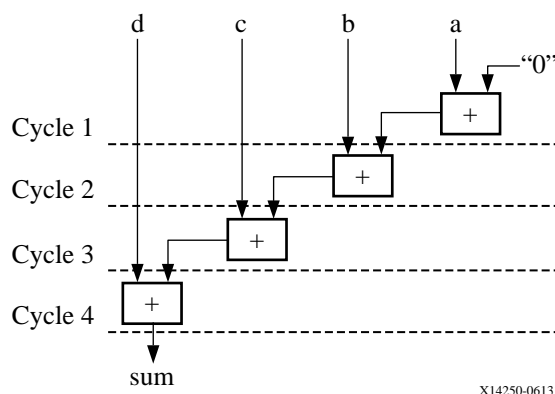
たとえば、+= および *= などの代入演算子を使用した次のようなシーケンシャル コードがあるとします。

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d)
{
    data_t sum;

    sum = 0;
    sum += a;
    sum += b;
    sum += c;
    sum += d;
    return sum;
}
```

演算式バランス調整が実行されない場合、各加算に 1 クロック サイクルが必要だとすると、次の図に示すように sumsum の計算に 4 クロック サイクルかかります。

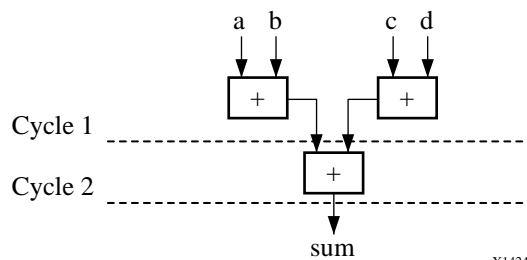
図 72: 加算器ツリー



X14250-061318

ただし、加算 $a+b$ および $c+d$ は並列実行できるので、レイテンシを削減できます。このように計算のバランスが調整されると、計算は 2 クロック サイクルで終了します。演算式バランス調整を使用すると共有はできず、エリアは増加します。

図 73: バランス調整後の加算器ツリー



整数に対しては、EXPRESSION_BALANCE 最適化指示子を `off` に設定すると、演算式バランス調整をオフにできます。デフォルトでは、Vivado HLS で `float` または `double` 型の演算に対しては EXPRESSION_BALANCE 最適化は実行されません。`float` および `double` 型を合成する際には、C シミュレーションと結果が同じになるように、Vivado HLS で C コードで実行される演算順序が維持されます。たとえば、次のコード例では、すべての変数が `float` または `double` 型です。O1 および O2 は、同じ基本的な計算を実行しているように見えますが、値は同じではありません。

```
A=B*C; A=B*F;
D=E*F; D=E*C;
O1=A*D O2=A*D;
```

これは、`float` または `double` 型の演算における C 標準の飽和および丸めの結果です。このため、Vivado HLS では `float` または `double` 型の変数が使用され、デフォルトで演算式バランス調整が実行されない場合、演算の順序が常に維持されます。

`float` および `double` 型で演算式バランス調整を有効にするには、`config_compile` コンフィギュレーション オプションを次のように使用します。

1. [Solution > Solution Settings] をクリックします。
2. [Solution Settings] ダイアログ ボックスで [General] カテゴリをクリックし、[Add] をクリックします。
3. [Add Command] ダイアログ ボックスで [`config_compile`] を選択して [`unsafe_math_operations`] をオンにします。

この設定をオンにすると、Vivado HLS でさらに最適なデザインが生成されるように演算の順序が変更されることがありますが、C/RTL 協調シミュレーションの結果が C シミュレーションと異なるものになる可能性があります。

`unsafe_math_operations` を使用すると、`no_signed_zeros` 最適化も使用されます。`no_signed_zeros` 最適化により、`float` および `double` 型を使用した次の演算式が同一になります。

```
x - 0.0 = x;
x + 0.0 = x;
0.0 - x = -x;
x - x = 0.0;
x*0.0 = 0.0;
```

`no_signed_zeros` 最適化が使用されない場合は、丸めが実行されるので、上記の演算式は同一になりません。この最適化は、`config_compile` でこのオプションのみをオンにすると、演算式バランス調整なしで使用できます。



ヒント: `unsafe_math_operations` および `no_signed_zero` 最適化が使用されると、RTL インプリメンテーションが C シミュレーションとは異なる結果になります。テストベンチでは、結果の多少の違いは無視できるので、範囲を確認し、厳密には比較しないでください。

RTL の検証

合成後の検証は、C/RTL 協調シミュレーション機能 (合成前の C テストベンチを再利用して出力 RTL の検証実行) を使用して自動化されています。

RTL の自動検証

C/RTL 協調シミュレーションでは、C テストベンチを使用して RTL デザインが自動的に検証されます。検証プロセスには、次の図に示すように 3 つの段階があります。

- C シミュレーションが実行され、最上位関数または Device-Under-Test (DUT) への入力が入力ベクターとして保存されます。
- この入力ベクターは Vivado HLS で作成された RTL を使用して RTL シミュレーションに使用されます。RTL からの出力が出力ベクターとして保存されます。
- RTL シミュレーションからの出力ベクターが C テストベンチに適用され、合成の関数の後に結果が正しいかどうかを検証されます。C テストベンチで結果の検証が実行されます。

次は、検証の進捗状況を示すために Vivado HLS で表示されるメッセージです。

C シミュレーション:

```
[SIM-14] Instrumenting C test bench (wrapc)
[SIM-302] Generating test vectors(wrapc)
```

この段階では、C シミュレーションが実行されているので、C テストベンチによって生成されるメッセージはすべて、コンソール ビューまたはログ ファイルに出力されます。

RTL シミュレーション:

```
[SIM-333] Generating C post check test bench
[SIM-12] Generating RTL test bench
[SIM-323] Starting Verilog simulation (Issued when Verilog is the RTL
verified)
[SIM-322] Starting VHDL simulation (Issued when VHDL is the RTL verified)
```

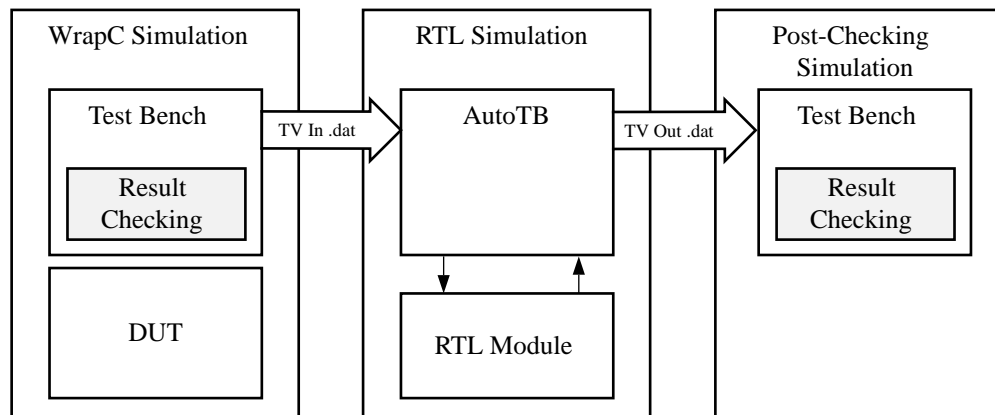
この段階では、RTL シミュレーションからのメッセージはすべてコンソール ビューかログ ファイルに出力されます。

C テストベンチの結果チェック:

```
[SIM-316] Starting C post checking
[SIM-1000] C/RTL co-simulation finished: PASS (If test bench returns a 0)
[SIM-4] C/RTL co-simulation finished: FAIL (If the test bench returns non-
zero)
```

C/RTL 協調シミュレーション フローの C テストベンチの重要性については、次に説明します。

図 74: RTL 検証フロー



X14311

C/RTL 協調シミュレーション機能を問題なく使用するには、次が必要です。

- テストベンチがセルフチェックで、問題がなかった場合は 0 を、問題があった場合は 0 以外の値を返す。
- 正しいインターフェイス合成オプションを選択する。
- 検索パスにサードパーティ シミュレータが存在する。
- デザイン インターフェイスの配列または構造体 (struct) では、[協調シミュレーションでサポートされない最適化](#)にリストする最適化指示子または最適化指示子の組み合わせを使用していない。

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、セルフチェック テストベンチを使用する必要があります。次のコード例は、セルフチェック テストベンチの重要な機能を示しています。

```
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;
}
```

このセルフチェック テストベンチでは、結果が `output.golden.dat` ファイルの既知の良い結果と比較されます。

このチェックは、さまざまな方法で実行できます。これは、その一例です。

Vivado HLS デザイン フローでは、`main()` 関数への戻り値がそれぞれ次を示します。

- 0: 結果が正しいことを示します。
- 0 以外: 結果が正しくないことを示します。

注記: テストベンチから 0 以外の値が返されることがあります。複雑なテストベンチには、相違点やエラーのタイプによって異なる値を返すものもあります。C シミュレーションまたは C/RTL 協調シミュレーション後にテストベンチから 0 以外の値が返されると、Vivado HLS でエラーまたはシミュレーション エラーがレポートされます。



推奨: `main()` 関数の戻り値はシステム環境で解釈されるので、では、戻り値を 8 ビットの範囲に制約して、ポータビリティと安全性を上げることをお勧めします。



注意: テストベンチで結果がチェックされるようにするのは、ユーザーの責任です。テストベンチで結果がチェックされないのに 0 が返された場合は、Vivado HLS で結果が実際にチェックされなかったのにシミュレーション テストが「PASS」と示されます。出力データが正しくて有効であって、テストベンチから `main()` 関数に 0 が返されない場合は、Vivado HLS でシミュレーション エラーがレポートされます。

インターフェイス合成要件

C/RTL 協調シミュレーションを使用して RTL デザインを検証するには、次の条件のうち少なくとも 1 つが満たされている必要があります。

- 最上位関数が `ap_ctrl_hs` または `ap_ctrl_chain` ブロック レベル インターフェイスを使用して合成されている。
- デザインが純粋に組み合わせである。
- 最上位関数の開始間隔 (II) が 1 になっている。
- すべてのインターフェイスが、`ap_hs` または `axis` インターフェイス モードでインプリメントされるストリーミングの配列である。

注記: `hls::stream` 変数は自動的に `ap_fifo` インターフェイスとしてインプリメントされます。

これらの条件の少なくとも 1 つでも満たされない場合、次のメッセージが表示されて C/RTL 協調シミュレーションが停止します。

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3)
designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```



重要: デザインがブロック レベルの I/O プロトコル `ap_ctrl_none` を使用するように指定されていて、デザインにブロッキング以外の動作を使用する `hls::stream` が含まれる場合は、C/RTL 協調シミュレーションが必ず終了するとは限りません。

最上位関数引数が AXI4-Lite インターフェイスとして指定されている場合、関数の戻り値も AXI4-Lite インターフェイスとして指定する必要があります。

RTL シミュレータのサポート

先の要件が満たされていれば、C/RTL 協調シミュレーションを使用し、Verilog または VHDL で RTL デザインを検証できます。デフォルトのシミュレーション言語は Verilog ですが、VHDL を指定することもできます。デフォルト シミュレータは Vivado シミュレータ (XSim) ですが、次のいずれかのシミュレータを使用しても C/RTL 協調シミュレーションを実行できます。

- Vivado シミュレータ (XSim)
- ModelSim シミュレータ
- VCS シミュレータ
- NC-Sim シミュレータ
- Riviera シミュレータ
- Xcelium



重要: サードパーティ シミュレータ (たとえば、ModelSim、VCS、Riviera) を使用して RTL デザインを検証するには、そのシミュレータの実行ファイルがシステム検索パスに含まれており、適切なライセンスが使用可能な状態になっている必要があります。これらのシミュレータの設定の詳細は、サードパーティ ベンダーの資料を参照してください。



重要: SystemC デザインを検証する際には、ModelSim シミュレータを選択して、適切なライセンスを使用した C コンパイラ機能が含まれるようにしておく必要があります。

協調シミュレーションでサポートされない最適化

RTL 自動検証は、配列またはインターフェイスの構造体内の配列に、実行される複数の変換がある場合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 1 つのみを使用する必要があります。

- 同じサイズの配列の垂直マップ
- 再形成
- パーティション
- 構造体の DATA_PACK

次の最適化が最上位関数インターフェイスで使用されると、C/RTL 協調シミュレーションによる検証は実行できません。

- 水平マップ
- 異なるサイズの配列の垂直マップ
- ほかの構造体をメンバーとして含む構造体の DATA_PACK
- レジスタ スライスを一時的に AXI の条件付きアクセスは、サポートされません。
- 配列のストリームへのマップ。

IP コアのシミュレーション

デザインを浮動小数点コアを使用してインプリメントする場合、RTL シミュレータでその浮動小数点コアのビット精度モデルを使用できるようにしておく必要があります。これは、RTL シミュレーションをザイリンクス Vivado シミュレータで Verilog および VHDL を使用して実行すると自動的に設定されます。

サポートされるサードパーティの HDL シミュレータの場合、ザイリンクス浮動小数点ライブラリをあらかじめコンパイルし、シミュレータのライブラリに追加する必要があります。次の例は、浮動小数点ライブラリが VCS シミュレータで使用できるように、Verilog でコンパイルされる手順を示しています。

1. Vivado (Vivado HLS ではなく) を開いて、[Tcl Console] ウィンドウに次のコマンドを入力します。

```
compile_simlib -simulator vcs_mx -family all -language verilog
```

2. このコマンドにより、現在のディレクトリに浮動小数点ライブラリが作成されます。
3. ディレクトリ名については Vivado のコンソール ウィンドウを参照してください (例: ./rev3_1)。

この後、このライブラリは次のコマンドを使用すると、Vivado HLS から参照できるようになります。

```
cosim_design -trace_level all -tool vcs -compiled_library_dir/  
<path_to_compile_library>/rev3_1
```

C/RTL 協調シミュレーションの使用


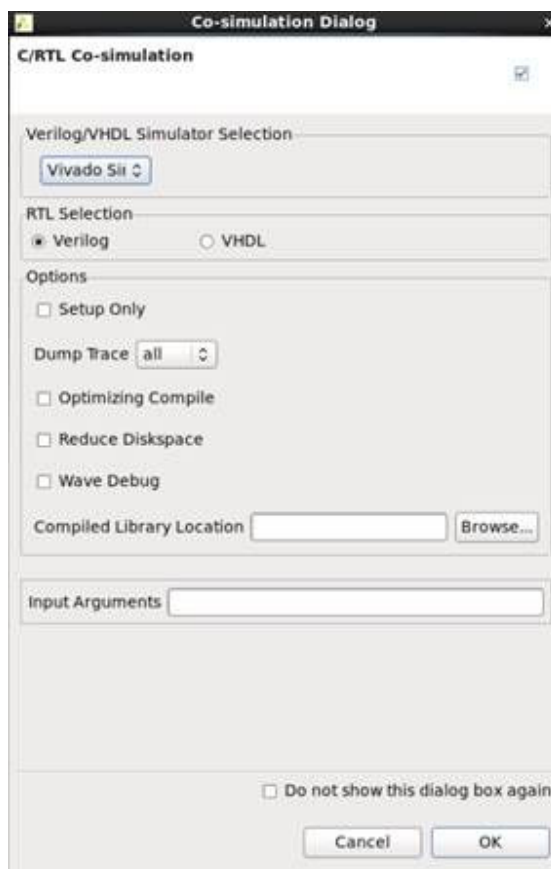
GUI から C/RTL 協調シミュレーションを実行するには、[C/RTL Cosimulation] ツールバー ボタン  をクリックします。これにより、次の図のようなシミュレーション ウィザードが開きます。

図 75: C/RTL Co-Simulation ウィザード



シミュレーションする RTL (Verilog または VHDL) を選択します。[Directive] ドロップダウン リストで指示子を選択します。

次は、そのオプションです。

- [Setup Only]: シミュレーションを実行するのに必要なファイルすべて (ラッパー、アダプター、およびスクリプト) が作成されますが、シミュレータは実行されません。シミュレーションは、RTL シミュレーション フォルダ <solution_name>/sim/<RTL> 内からコマンド シェルで実行できます。
- [Dump Trace]: 各関数のトレース ファイルが生成され、<solution>/sim/<RTL> フォルダに保存されます。ドロップダウン リストからは、どの信号をトレース ファイルに保存するか選択できます。デザインに含まれるすべての信号をトレースするように選択できるほか、最上位ポートだけをトレースしたり、どの信号もトレースしないように指定することもできます。トレース ファイルの使用の詳細は、選択した RTL シミュレータの資料を参照してください。
- [Optimizing Compile]: C テストベンチをコンパイルするのに高レベルの最適化が使用されるようになります。このオプションを使用すると、コンパイル時間は増加しますが、シミュレーションの実行は高速になります。
- [Reduce Disk Space]: 上記に示すフローで、RTL シミュレーションの実行前にすべてのトランザクションの結果が保存されます。データ ファイルが大きくなることもあります。reduce_diskspace オプションを使用すると、一度に 1 つのトランザクションが実行できるようになり、ファイルに必要なディスク容量が削減できます。関数が C テストベンチで N 回実行される場合、reduce_diskspace オプションを使用すると、RTL シミュレーションが別々に N 回実行されるようになり、シミュレーションの実行速度が遅くなります。
- [Compiled Library Location]: サードパーティの RTL シミュレータのコンパイル済みライブラリのディレクトリを指定します。

サードパーティ RTL シミュレータでシミュレーションする場合、デザインに IP が使用されていれば、RTL シミュレーションを実行する前にその IP の RTL シミュレーション モデルを使用する必要があります。RTL シミュレーションモデルを作成または取得するには、IP プロバイダーにご連絡ください。

- [Input Arguments]: テストベンチに必要な引数を指定できます。

RTL シミュレーションの実行

Vivado HLS では、次のプロジェクトのサブディレクトリ <SOLUTION>/sim/<RTL> の RTL シミュレーションが実行されます。

説明:

- SOLUTION はソリューションの名前。
- RTL はシミュレーションに選択された RTL のタイプ。

協調シミュレーション中に C テストベンチにより生成されたファイルとシミュレータで生成されたトレース ファイルは、すべてこのディレクトリに保存されます。たとえば、C テストベンチが比較の出力結果を保存する場合は、このディレクトリの出力ファイルを確認し、予測結果とそれを比較します。

指示子の検証

C/RTL 協調シミュレーションでは、DEPENDENCE および DATAFLOW 指示子が自動的に検証されます。

DATAFLOW 指示子を使用してタスクをパイプライン処理すると、タスク間のデータフローを円滑にするため、タスク間にチャンネルが挿入されます。通常、チャンネルは FIFO を使用してインプリメントされ、FIFO の深さは STREAM 指示子または config_dataflow コマンドで指定されます。FIFO の深さが小さすぎると、RTL シミュレーションが停止することがあります。たとえば、FIFO の深さ 2 に指定されている場合、コンシューマー タスクでデータ値が読み出される前にプロデューサー タスクが 3 つの値を書き込むと、FIFO がプロデューサーをブロックします。場合によっては、これによりデザイン全体が停止していきることがあります。

C/RTL 協調シミュレーションでは、次のように DATAFLOW 領域のチャンネルが RTL シミュレーションの停止の原因になっていることを示すメッセージが表示されます。

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
// ERROR!!! DEADLOCK DETECTED at 1292000 ns! SIMULATION WILL BE STOPPED! //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Dependence cycle 1:
// (1): Process: hls_fft_1kxburst.fft_rank_rad2_nr_man_9_U0
//      Channel: hls_fft_1kxburst.stage_chan_in1_0_V_s_U, FULL
//      Channel: hls_fft_1kxburst.stage_chan_in1_1_V_s_U, FULL
//      Channel: hls_fft_1kxburst.stage_chan_in1_0_V_1_U, FULL
//      Channel: hls_fft_1kxburst.stage_chan_in1_1_V_1_U, FULL
// (2): Process: hls_fft_1kxburst.fft_rank_rad2_nr_man_6_U0
//      Channel: hls_fft_1kxburst.stage_chan_in1_2_V_s_U, EMPTY
//      Channel: hls_fft_1kxburst.stage_chan_in1_2_V_1_U, EMPTY
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Totally 1 cycles detected!
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

この場合、タスク間のチャンネルのインプリメンテーションを確認し、FIFO が生成されるデータを保持するのに十分な大きさであることを確認してください。

同様に、RTL テストベンチも、DEPENDENCE 指示子を使用して指定された偽依存を自動的に確認するよう設定されています。これは、依存は false ではなく、機能的に有効なデザインを達成するために削除する必要のあることを示しています。

RTL シミュレーションの解析

C/RTL 協調シミュレーションが終了すると、シミュレーション レポートが開いて、測定されたレイテンシと II が表示されます。これらの結果は、デザイン全体の絶対最短パスと最長パスに基づいた HLS 合成後にレポートされた値とは異なることがあります。C/RTL 協調シミュレーション後の結果には、指定したシミュレーション データ セットのレイテンシと II の実際の値が表示されており、別の入力スティミュラスを使用すると変わる可能性があります。

パイプライン処理されていないデザインの場合、C/RTL 協調シミュレーションで `ap_start` および `ap_done` 信号間のレイテンシが測定されます。すべての演算が終了した 1 サイクル後にデザインで新しい入力を読み出されるので、II はレイテンシより 1 長くなります。現在のトランザクションが終了してからしか次のトランザクションは開始されません。

パイプライン処理されたデザインでは、最初のトランザクションが終了する前に新しい入力を読み出されるので、トランザクションが終了する前に複数の `ap_start` および `ap_ready` 信号がある可能性があります。この場合、C/RTL 協調シミュレーションでレイテンシはデータ入力値とデータ出力値間のサイクル数として測定されます。II は、新しい入力を要求するために使用される `ap_ready` 信号間のサイクル数です。

注記: パイプライン処理されたデザインの場合、C/RTL 協調シミュレーションの II 値は、デザインが複数トランザクション間シミュレーションされる場合にのみ有効です。

C/RTL 協調シミュレーションからの波形は、[Open Wave Viewer] ツールバー ボタンで確認できます。RTL 波形を表示するには、C/RTL 協調シミュレーションを実行する前に次のオプションを選択しておく必要があります。

- [Verilog/VHDL Simulator Selection]: [Vivado Simulator] を選択します。ザイリンクス 7 シリーズ以降のデバイスの場合は、[Auto] を選択することもできます。
- [Dump Trace]: [all] または [port] を選択します。

C/RTL 協調シミュレーションが終了したら、Vivado IDE の [Open Wave Viewer] バー ボタンで RTL 波形を開きます。

注記: この方法で Vivado IDE を開く場合は、拡大/縮小、波形基数などの波形解析機能しか使用できません。

波形ビューアー

波形ビューアーには、デザイン内のプロセスがすべて可視化され、2 つのセクションに分割されます。

- HLS プロセスのサマリ

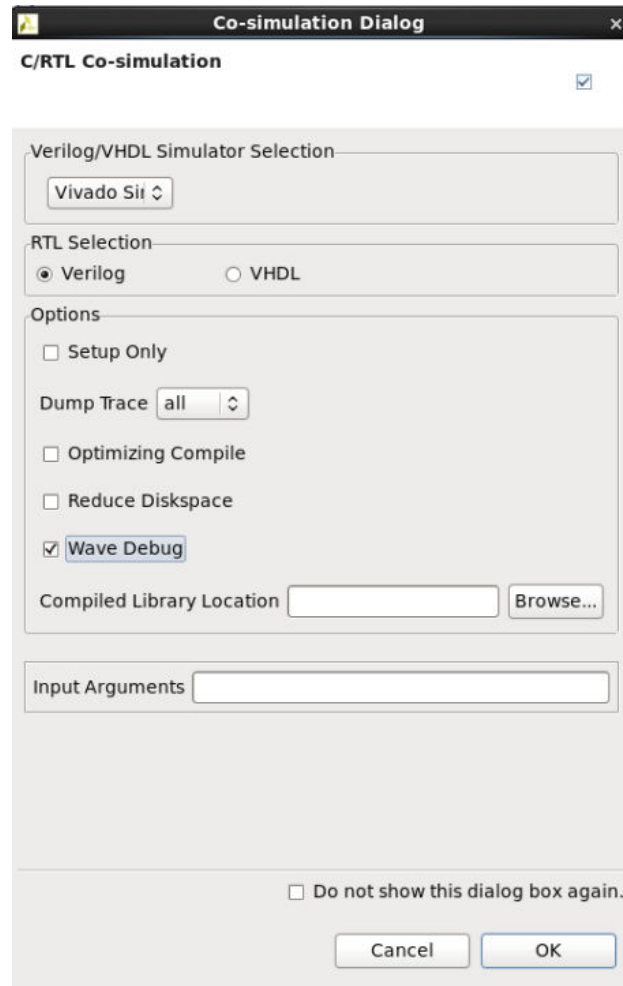
すべてのプロセスのアクティビティ レポートの階層表示が含まれます。たとえば、生成された RTL 内に含まれるデータフローと順次プロセスの両方が含まれます。
- データフロー解析

データフロー領域内のタスクに関する詳細なアクティビティ 情報が含まれます。

HLS デザイン内でアクティビティ プロセスを可視化すると、プロセス アクティビティおよび最上位モジュールの各アクティビティ 内の長さの詳細なプロファイリングが可能になります。これにより、個別プロセスのパフォーマンスだけでなく、それぞれのプロセスの全体的な並行処理実行を解析しやすくなります。

実行全体を占めるプロセスの方がパフォーマンスを改善する可能性が高く、プロセス実行時間を削減できます。この可視化は、Vivado シミュレータの協調シミュレーション中に使用できます。これには、[Co-simulation Dialog] ダイアログ ボックスで [Wave Debug] オプションをオンにします。

図 76: [Wave Debug] をオン

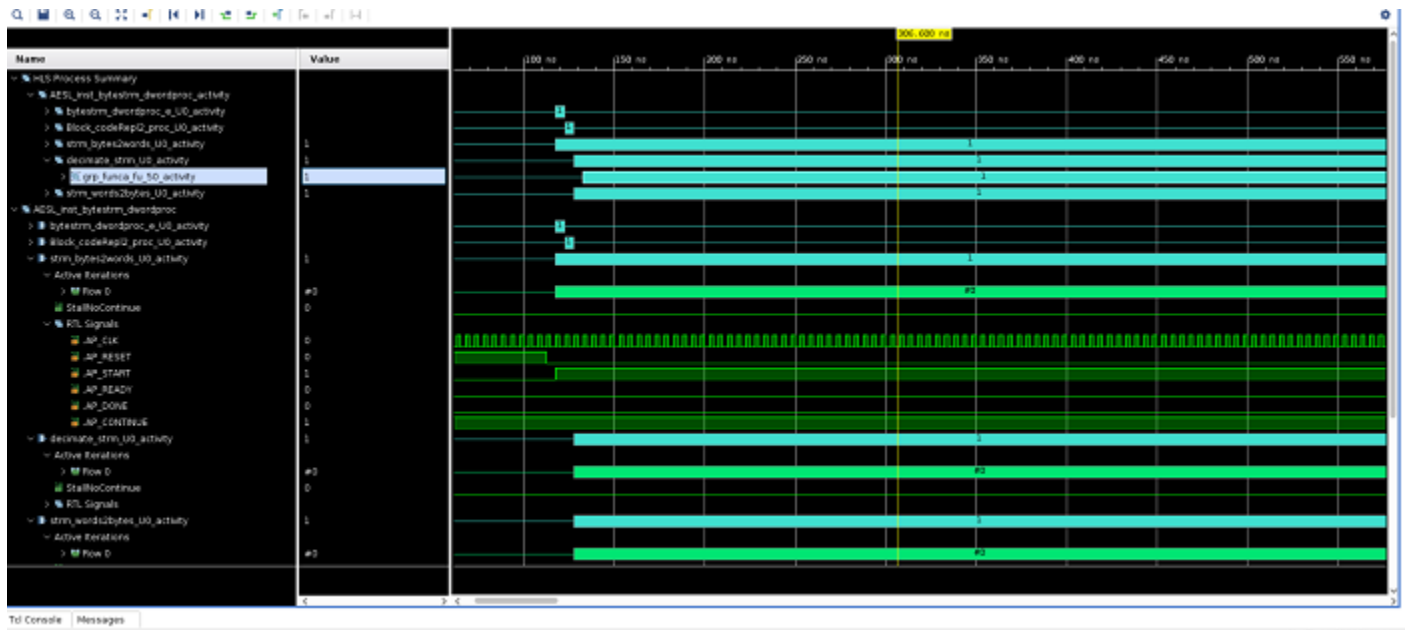


ビューアーは、次のセグメントに分割されます。

- HLS プロセスのサマリ
 - DUT 名: <name>
 - 関数: <function name>
- データフロー解析
 - DUT 名: <name>
 - 関数: <function name>
 - [Dataflow/Pipeline Activity]: データフロー プロセスとしてインプリメントされる場合に並列実行される関数の数を示します。
 - [Active Iterations]: 現在のアクティブなデータフローのイテレーションを示します。行数は現在の実行を表示するためにダイナミックに増加していきます。
 - [StallNoContinue]: これは、データフローで出力のストールがあるかどうかを伝えるストール信号です (関数は完了しますが、隣接するデータフロー プロセスからの続行信号は受信しません)。

- 。 [RTL Signals]: データフロー プロセスのトランザクション ビューを解釈するのに使用された RTL 制御信号です。

図 77: 波形ビューアー



C/RTL 協調シミュレーションのデバッグ

C/RTL 協調シミュレーションが終了すると、通常 Vivado HLS からシミュレーションが正常に完了し、RTL デザインの機能が最初の C コードと一致していることが示されます。Vivado HLS では、C/RTL 協調シミュレーションでエラーがあると、次のようなメッセージが表示されます。

```
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

C/RTL 協調シミュレーションでエラーが発生する主な原因は、次のとおりです。

- 環境設定が間違っている
- 適用された最適化指示子がサポートされていないか、間違っている
- C テストベンチまたは C ソース コードに問題がある

C/RTL 協調シミュレーションのエラーをデバッグするには、次のセクションで説明するようにチェックを実行します。C/RTL 協調シミュレーションのエラーを解決できない場合は、[ザイリンクス サポート](#)からアンサー、資料、ダウンロード、フォーラムなどのサポート資料を参照してください。

環境の設定

次の表に示す環境設定を確認してください。

表 18: 環境設定のデバッグ

質問	必要な操作
サードパーティ シミュレータを使用していますか。	シミュレータ実行ファイルへのパスがシステム検索パスに指定されていることを確認します。 Vivado シミュレータを使用する場合は、検索パスを指定する必要はありません。
Linux を実行していますか。	セットアップ ファイル (.cshrc、.bashrc など) に cd (change directory) コマンドが含まれていないことを確認します。C/RTL 協調シミュレーションを開始すると、新しいシェルプロセスが生成されます。セットアップ ファイルに cd コマンドが含まれると、シェルが別のディレクトリで実行され、C/RTL 協調シミュレーションでエラーが発生します。

最適化指示子

次の表に示す最適化指示子を確認してください。

表 19: デバッグ最適化指示

質問	必要な操作
DEPENDENCE 指示子を使用していますか。	DEPENDENCE を削除して、C/RTL 協調シミュレーションが正常に完了するかどうかを確認します。協調シミュレーションが正常に完了する場合は、DEPENDENCE の TRUE または FALSE 設定が間違っている可能性があります。
最上位インターフェイスで volatile ポインターが使用されていますか。	DEPTH オプションが INTERFACE 指示子に指定されているかどうかを確認します。インターフェイスに volatile ポインターが使用されている場合、各トランザクションまたは C 関数の各実行ごとにポートで実行される読み出し/書き込みの数を指定する必要があります。
データフロー最適化と共に FIFO を使用していますか。	標準ピンポン バッファを使用して C/RTL 協調シミュレーションが正常に完了するかどうかを確認します。 FIFO チャンネルのサイズを指定せずに C/RTL 協調シミュレーションが正常に完了するかどうかを確認し、チャンネルが C コードの配列サイズにデフォルトでなるようにします。 C/RTL 協調シミュレーションが停止するまで FIFO チャンネルのサイズを削減します。停止する場合は、チャンネル サイズが小さすぎることを意味します。デザインを確認して、FIFO に最適なサイズを決定してください。FIFO のサイズを個別に指定するには、STREAM 指示子を使用できます。
サポートされているインターフェイスを使用していますか。	サポートされるインターフェイス モードを使用するようにしてください。詳細は、 インターフェイス合成要件 を参照してください。
インターフェイスの配列に複数の最適化指示子を適用していますか。	一緒に機能するよう設計された最適化を使用するようにしてください。詳細は、 協調シミュレーションでサポートされない最適化 を参照してください。
ストリームにマップされるインターフェイスに配列を使用していますか。	インターフェイス レベルのストリーミング (DUT の最上位関数) を使用するには、hls::stream を使用します。

C テストベンチおよび C ソース コード

次の表に示す C テストベンチおよび C ソース コードを確認してください。

表 20: C テストベンチおよび C ソース コードのデバッグ

質問	必要な操作
C テストベンチで結果がチェックされ、結果が正しい場合に値 0 が返されますか。	C/RTL 協調シミュレーションで C テストベンチから 0 が返されるようにしてください。結果が正しい場合でも、C テストベンチから値 0 が返されないと、C/RTL 協調シミュレーションでエラーがレポートされます。
C テストベンチは乱数に基づいて入力データを作成していますか。	乱数生成に固定シードを使用するようにテストベンチを変更します。乱数生成のシードがタイム ベース シードなどの変数に基づいている場合、シミュレーションに使用されるデータがテストベンチを実行するたびに異なり、結果が異なってしまいます。
複数回アクセスされる最上位インターフェイスでポインターを使用していますか。	1 つのトランザクション (C 関数 1 回の実行) 内で複数回アクセスされるポインターには <code>volatile</code> ポインターを使用してください。 <code>volatile</code> ポインターを使用しないと、C 標準に準拠するため、最初の読み出しと最後の書き込み以外のすべてが最適化で削除されます。
C コードに定義されていない値が含まれていたり、範囲外の配列アクセスが実行されたりしていますか。	全配列がすべてのアクセスと一致する正しいサイズになっていることを確認してください。問題の原因として、ループ範囲が配列のサイズを超えていることがよくあります (例: N-1 のサイズ指定された配列に N アクセス)。 C シミュレーションの結果に問題がないかどうか、出力値が乱数データ値に割り当てられていないかどうかを確認します。 Vivado HLS 外で業界標準の Valgrind アプリケーションを使用して、C コードに未定義や範囲外の問題がないかどうかを確認することも考慮してください。 C 関数は、未定義または範囲外の変数があっても、実行して完了する可能性があります。C シミュレーションでは未定義の値に乱数が割り当てられますが、RTL シミュレーションでは不明または X 値が割り当てられます。
デザインに浮動小数点の数学演算を使用していますか。	精密な比較 (exact) を実行する代わりに、C テストベンチ結果が許容範囲のエラーであるかどうかを確認します。浮動小数点の数学演算の中には、RTL インプリメンテーションが C と同じではないものがあります。詳細は、 検証および数学関数 を参照してください。 サードパーティ シミュレータに、浮動小数点コアの RTL シミュレーション モデルを供給するようにします。詳細は、 IP コアのシミュレーション を参照してください。
ザイリンクス IP ブロックとサードパーティ シミュレータを使用していますか。	ザイリンクス IP の HDL モデルへのバスがサードパーティ シミュレータに渡されるようにします。
データ レートを変更する (間引きや補間など) デザインで <code>hls::stream</code> コンストラクトを使用していますか。	デザインを解析し、STREAM 指示子を使用して <code>hls::stream</code> をインプリメントするのに使用する FIFO のサイズを増加します。 デフォルトでは、 <code>hls::stream</code> が深さ 2 の FIFO としてインプリメントされます。たとえば補間などによってデータ レートが増加する場合は、デフォルトの FIFO サイズ 2 では小さすぎ、C/RTL 協調シミュレーションが停止することがあります。
シミュレーションで大容量のデータ セットを使用していますか。	C/RTL 協調シミュレーションを実行する際は <code>reduce_diskspace</code> オプションを使用してください。このモードでは、Vivado HLS は一度に 1 トランザクションしか実行しません。シミュレーションは多少低速になる可能性がありますが、ストレージおよびシステム容量の問題は制限されます。 C/RTL 協調シミュレーションでは、すべてのトランザクションが一度に検証されます。最上位関数が複数回呼び出されると (ビデオの複数フレームをシミュレーションする場合など)、シミュレーション全体の入力および出力のデータがディスクに保存されます。マシンの設定と OS によっては、これがパフォーマンスや実行の問題の原因となることがあります。

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 デザイン フローのその他のツールでできるように、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、次の出力形式にパッケージできます。

- IP カタログ フォーマットの IP (Vivado Design Suite で使用)
- System Generator for DSP IP (Vivado System Generator for DSP で使用)
- 合成済みチェックポイント (.dcp)

次の表は、それぞれの詳細と共にエクスポート可能なフォーマットを示しています。

表 21: [RTL Export] の選択肢

フォーマット	サブフォルダー	コメント
IP Catalog	ip	Vivado IP カタログに追加できる ZIP ファイルが含まれます。 ip フォルダーには、ZIP ファイルの内容 (抽出済み) も含まれます。 7 シリーズまたは Zynq-7000 SoC よりも古い FPGA デバイスの場合、このオプションは使用できません。
System Generator for DSP	sysgen	この出力は、System Generator for DSP の Vivado エディションに追加できます。 7 シリーズまたは Zynq-7000 SoC よりも古い FPGA デバイスの場合、このオプションは使用できません。
Synthesized Checkpoint (.dcp)	ip	このオプションでは、Vivado Design Suite のデザインに直接追加できる Vivado チェックポイント ファイルが作成されます。 これには、RTL 合成が実行されている必要があります。このオプションをオンにすると、flow オプションと syn 設定が自動的に選択されます。 出力には、IP を HDL ファイルにインスタンスエートするために使用できる HDL ラッパーが含まれます。

パッケージされた出力形式のほか、RTL ファイルを <project_name>/<solution_name>/impl ディレクトリの下にある verilog および vhdl ディレクトリからスタンドアロン ファイル (パッケージ済みフォーマットの一部ではなく) として入手できます。

RTL ファイル以外にも、これらのディレクトリには Vivado Design Suite 用のプロジェクト ファイルが含まれます。project.xpr ファイルを開くと、デザイン (Verilog または VHDL) が Vivado プロジェクト内で開き、解析できるようになります。Vivado HLS プロジェクトで C/RTL 協調シミュレーションが実行されると、その C/RTL 協調シミュレーション ファイルが Vivado プロジェクト内に表示されます。

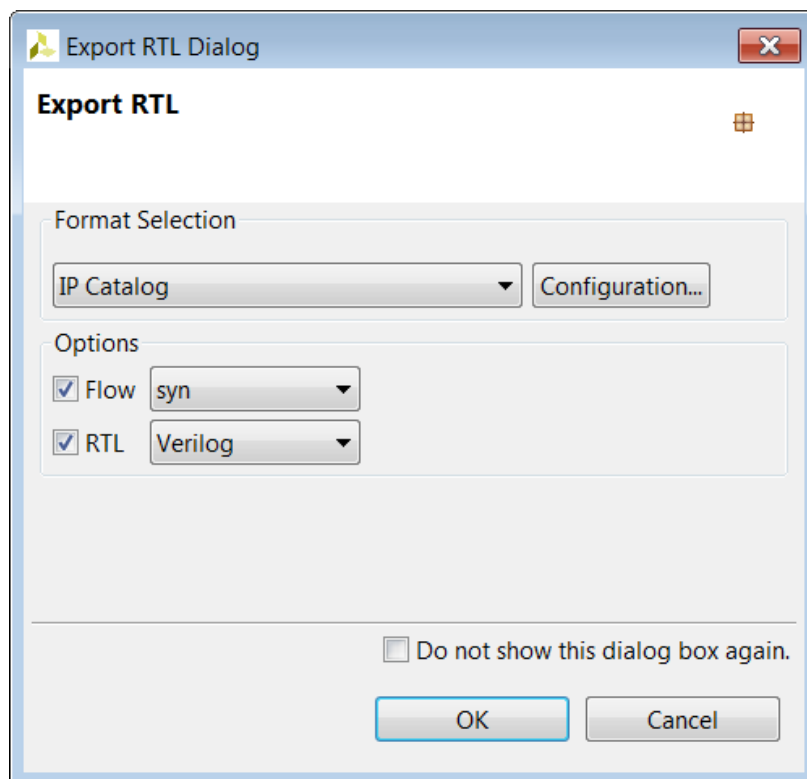
RTL の合成

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 ブロック RAM 数などの RTL 合成後の見積もりが含まれます。Vivado HLS ではダウンストリームの RTL 合成での正確な最適化結果がわからず、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見積もりです。

デザインをエクスポートする前には、論理合成を実行して、見積もりの正確さを確認できます。次の図に示す [Flow] オプションでは、エクスポート プロセス中に `syn` オプションの場合は RTL 合成が、`impl` オプションの場合は RTL 合成およびインプリメンテーションが実行され、RTL デザインがゲートに合成されるか、配置配線済みのインプリメンテーションになります。

RTL 合成オプションはレポートされる見積もりを確認するために提供されています。ほとんどの場合、これらの RTL 結果はパッケージされる IP には含まれません。


図 78: [Export RTL] ダイアログ ボックス



ほとんどのエクスポート形式では、RTL 合成が `verilog` または `vhdl` ディレクトリ (前の図のドロップダウン リストを使用して RTL 合成用を選択された方の HDL いずれか) で実行されますが、その RTL 合成の結果はパッケージされる IP には含まれません。

注記: 合成済みのデザイン チェックポイント (`.dcp`) は常に合成済み RTL としてエクスポートされます。[Flow] オプションは、合成またはインプリメンテーションの結果を評価するために使用しますが、エクスポートされたパッケージには常に合成済みネットリストが含まれます。

IP カタログ形式用のパッケージ

合成および RTL 検証が終了したら、[Export RTL] ツールバー ボタン () をクリックして、[Export RTL] ダイアログ ボックスを開きます。

[IP Catalog] から [Format Selection] を選択します。

[Configuration] オプションを使用すると、エクスポートしたパッケージに次の ID タグが埋め込まれます。これらのフィールドを使用すると、Vivado IP カタログ内でパッケージされた RTL が識別しやすくなります。

[Configuration] 情報は、デザインを IP カタログに読み込んだときに、同じデザイン内の複数のインスタンスを区別するために使用されます。たとえば、インプリメンテーションが IP カタログ用にパッケージされてから、新しいソリューションが作成されて IP としてパッケージされると、新しいソリューションにはデフォルトで同じ名前とコンフィギュレーション情報が含まれます。新しいソリューションが IP カタログに追加されると、IP カタログではそれが同じ IP のアップデート バージョンとして認識され、IP カタログに追加された最新バージョンの方が使用されるようになります。

または、`config_rtl` コンフィギュレーションに `prefix` を使用すると、出力デザインおよびファイルの名前に独自の接頭辞を付けることができます。

configuration 設定に値がない場合は、次の値が使用されます。

- [Vendor]: xilinx.com
- [Library]: hls
- [Version]: 1.0
- [Description]: An IP generated by Vivado HLS
- [Display Name]: このフィールドはデフォルトで空白のままになります。
- [Taxonomy]: このフィールドはデフォルトで空白のままになります。

パッケージ プロセスが終了したら、ZIP ファイルのアーカイブ ディレクトリ (`<project_name>/<solution_name>/impl/ip`) を Vivado IP カタログにインポートして、Vivado デザイン (RTL または IP インテグレーター) で使用します。

ソフトウェア ドライバー ファイル

AXI4-Lite スレーブ インターフェイスを含むデザインの場合、エクスポート プロセスでソフトウェア ドライバー ファイルのセットが作成されます。これらの C ドライバー ファイルは SDK C プロジェクトに含めて、AXI4-Lite スレーブ ポートにアクセスするために使用できます。

ソフトウェア ドライバー ファイルは、`<project_name>/<solution_name>/impl/ip/drivers` ディレクトリに書き込まれ、パッケージの ZIP アーカイブに含められます。C ドライバー ファイルの詳細は、[AXI4-Lite インターフェイス](#) を参照してください。

System Generator への IP のエクスポート


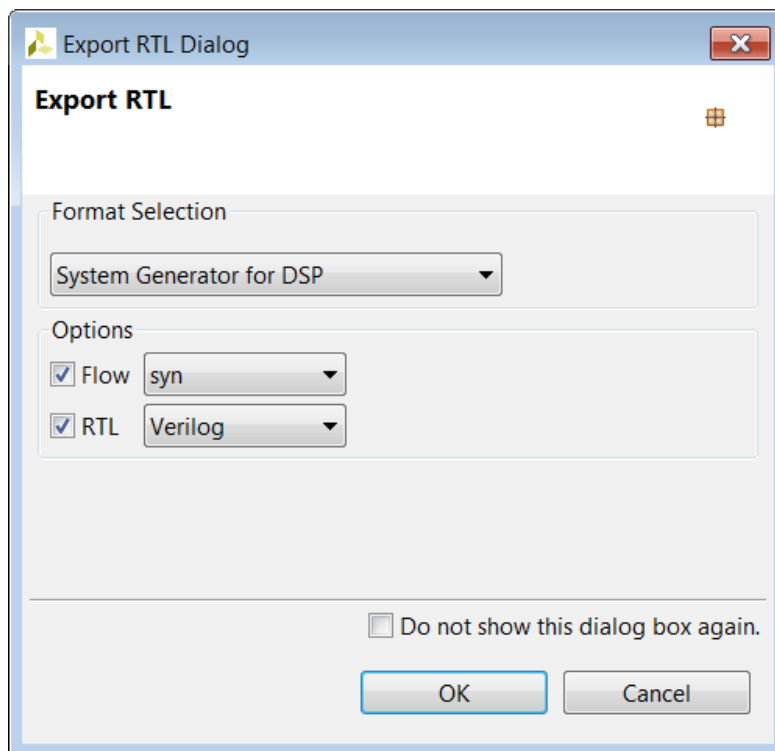
合成および RTL 検証が終了したら、[Export RTL] ツールバー ボタン () をクリックして、[Export RTL Dialog] ダイアログ ボックスを開きます。

図 79: System Generator への RTL のエクスポート



配置配線後のリソースと IP ブロックのタイミング統計が必要であれば、[Flow] オプションをオンにして、RTL 言語を選択します。

[OK] をクリックして IP パッケージを生成します。このパッケージは <project_name>/<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 ヘド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。

[Flow] オプションをオンにすると、RTL 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が、IP パッケージには含まれません。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. 新しくインストールした 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。たとえば <project_name>/<solution_name>/impl/sysgen の場合は、<project_name>/<solution_name> ディレクトリを指定して [Apply] をクリックします。

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がわからないので、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、マップ、データ パック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 ブロックを System Generator に含めた後、次のようになる必要があります。

- 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の詳細は、System Generator の資料を参照してください。

合成済みチェックポイントのエクスポート


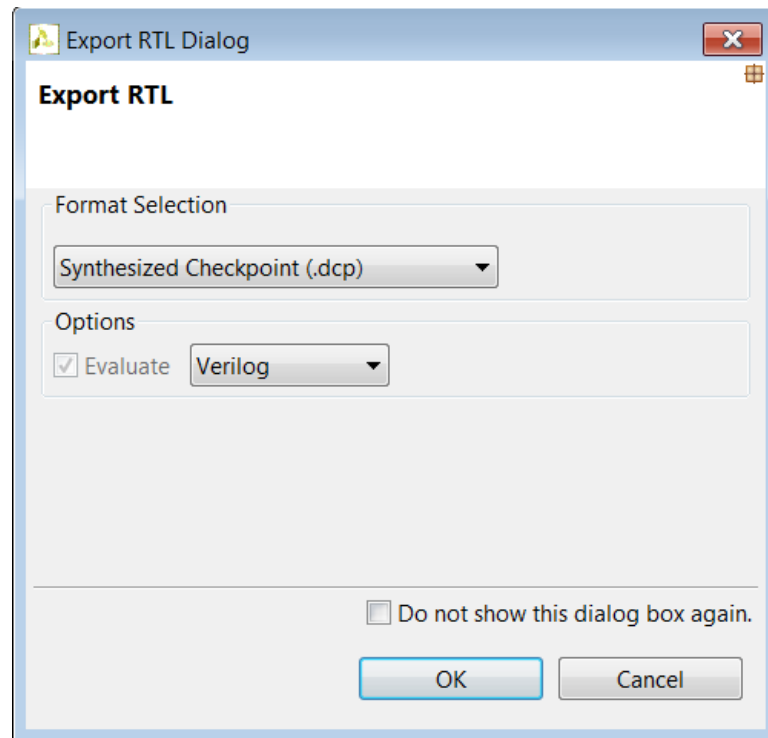
合成および RTL 検証が終了したら、[Export RTL] ツールバー ボタン () をクリックして、[Export RTL Dialog] ダイアログ ボックスを開きます。

図 80: 合成済みチェックポイントへの RTL のエクスポート



デザインがデザイン チェックポイント IP としてパッケージされる場合、パッケージよりも前にまず合成を実行する必要があります。

[OK] をクリックすると、デザイン チェックポイント パッケージが生成されます。このパッケージは `<project_name>/<solution_name>/impl/ip` ディレクトリに書き込まれます。デザイン チェックポイント ファイルは、その他のデザイン チェックポイントと同じように Vivado Design Suite プロジェクトで使用できます。

高位合成の C ライブラリ

Vivado[®] HLS の C ライブラリを使用すると、一般的なハードウェア デザイン コンストラクトおよび関数を C で簡単に記述して RTL に合成できます。Vivado HLS には次の C ライブラリが含まれます。

- 任意精度データ型ライブラリ
- HLS ストリーム ライブラリ
- HLS 数学ライブラリ
- HLS ビデオ ライブラリ
- HLS IP ライブラリ
- HLS 線形代数ライブラリ
- HLS DSP ライブラリ

ライブラリ ヘッダー ファイルを含めると、各 C ライブラリをデザインで使用できます。これらのヘッダー ファイルは、Vivado HLS のインストール ディレクトリ内の `include` ディレクトリにあります。



重要: Vivado HLS C ライブラリのヘッダー ファイルは、デザインが Vivado HLS で使用される場合はインクルード パスに含める必要はありません。ライブラリ ヘッダー ファイルへのパスは、自動的に追加されます。

任意精度型ライブラリ

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 パス (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メカニズムが必要です。たとえば、17 ビット乗算器が必要な場合に、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++ の整数および固定小数点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てから、C シミュレーションを再実行して機能が同一または使用可能であることを検証できる点です。

関連情報

[float および double 型](#)

任意精度データ型

Vivado[®] HLS では、次の表に示すように任意精度の整数データ型が提供されており、指定した幅の境界内で整数値を管理できるようになっています。

表 22: 整数データ型

言語	整数型	必要なヘッダー
C	[u]int<precision> (1024 ビット)	gcc #include "ap_cint.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"

Vivado HLS には、任意精度型を定義するヘッダー ファイルもスタンドアロン パッケージとして含まれており、ソース コードで使用できるようになっています。xilinx_hls_lib_<release_number>.tgz パッケージは、Vivado HLS インストール ディレクトリの include ディレクトリに含まれます。

C 言語での整数任意精度型

C 言語では、ヘッダー ファイル ap_cint.h で任意精度の整数型 [u]int が定義されます。

注記: xilinx_hls_lib_<release_number>.tgz パッケージには、ap_cint.h で定義された C 任意精度データ型は含まれていません。これらのデータ型は、標準 C コンパイラとは一緒に使用できず、Vivado HLS cpcc コンパイラでのみ使用できるようになっています。

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、符号付きの型の場合は intN、符号なしの型の場合は uintN (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_cint.h"

void foo_top () {

    int9   var1;           // 9-bit
    uint10 var2;           // 10-bit unsigned
}
```

C++ 言語での整数任意精度型

ヘッダー ファイル ap_int.h では、C++ の任意精度整数型の ap_[u]int 型が定義されます。C++ 関数で任意精度の整数型を使用するには、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、符号付きの型の場合は ap_int<N>、符号なしの型の場合は 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
```

SystemC 言語での任意精度整数型

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。ヘッダー ファイルには、SystemC の `sc_int<>`、`sc_uint<>`、`sc_bigint<>`、および `sc_bignint<>` 型が含まれます。

任意精度固定小数点データ型

Vivado HLS では、固定小数点データ型を使用することが重要です。これは、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が、合成で作成したハードウェアの結果と同じになるからです。これにより、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ようになります。

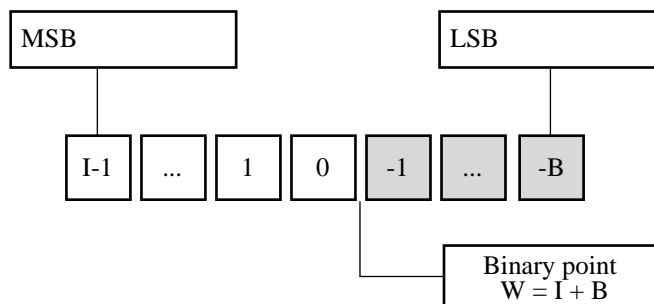
Vivado HLS では、C++ および SystemC 関数で使用するための任意精度固定小数点データ型が提供されています。

表 23: 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++	<code>ap_[u]fixed<W,I,Q,O,N></code>	<code>#include "ap_fixed.h"</code>
System C	<code>sc_[u]fixed<W,I,Q,O,N></code>	<code>#define SC_INCLUDE_FX</code> <code>[#define SC_FX_EXCLUDE_OTHER]</code> <code>#include "systemc.h"</code>

これらのデータ型では、次の図に示すように、指定の合計幅および整数幅の境界内で実数 (整数以外) の値が管理されます。

図 81: 固定小数点型



X14268

固定小数点の識別子のまとめ

次の表に、固定小数点データ型でサポートされる演算の簡単な概要を示します。

表 24: 固定小数点の識別子のまとめ

識別子	説明		
W I	[Word length in bits]: 整数値をビット数で指定 (整数部のビット数)		
Q	[Quantization mode]: 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	ap_fixed 型	説明
	SC_RND	AP_RND	Round to plus infinity
	SC_RND_ZERO	AP_RND_ZERO	Round to zero
	SC_RND_MIN_INF	AP_RND_MIN_INF	Round to minus infinity
	SC_RND_INF	AP_RND_INF	Round to infinity
	SC_RND_CONV	AP_RND_CONV	収束丸め
	SC_TRN	AP_TRN	負の無限大への切り捨て (デフォルト)
	SC_TRN_ZERO	AP_TRN_ZERO	0 への切り捨て
O	[Overflow mode]: 演算結果が結果を格納するのに使用される変数に格納可能な最大値 (負の値の場合は最小値) を超えた場合の動作を指定します。		
	SystemC 型	ap_fixed 型	説明
	SC_SAT	AP_SAT	飽和
	SC_SAT_ZERO	AP_SAT_ZERO	0 への飽和
	SC_SAT_SYM	AP_SAT_SYM	対称飽和
	SC_WRAP	AP_WRAP	折り返し (デフォルト)
	SC_WRAP_SM	AP_WRAP_SM	符号絶対値のラップ
N	オーバーフロー折り返しモードの場合の飽和ビット数を定義します。		

ap_fixed を使用した例

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。変数は符号付きで指定されており、量子化モードは正の無限大へ丸められるように設定され、オーバーフローにはデフォルトの折り返しモードが使用されます。

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

sc_fixed を使用した例

次の `sc_fixed` 型の例では、22 ビット変数 (整数部 21 ビット) を示しており、最小精度 0.5 だけが有効になっています。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;
...
```

C の任意精度整数データ型

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。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。

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。

C の任意精度データ型の利点

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 シミュレーションおよび解析

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 言語では、ヘッダー ファイル `ap_cint.h` で任意精度の整数型 `[u]int#W` が定義されます。次に例を示します。

- `int8` は 8 ビットの符号付き整数データ型を表します。
- `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。

次のコード例は、基本演算例 (標準データ型) のコードを繰り返したものです。両方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 "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;

}

```

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。次の例を[標準データ型](#)の基本演算の例と比較してください。

- 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ために削減されています。たとえば、8ビット入力 `inA` は6ビット入力に削減されています。
- 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ビットだけを必要とします。

```

#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);

```

前の例が合成されると、[標準データ型](#)の基本演算の例と同じ機能のデザイン(データは前の例で指定された範囲内)になります。ビット幅が少ないとロジックも削減されるので、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。

関数は、合成前に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` ファイルにビット 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た属性がコンパイルされますが、その属性の意味は解釈されません。このため、コードのビット精度ビヘイビアが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 `gcc` (またはその他のサードパーティ C コンパイラ) により -4 ではなく、10 進数値の 4 として処理されます。

注記: この問題は、C 任意精度型を使用する場合にのみ発生し、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。

Vivado HLS では、任意精度型が使用されていると認識されると、独自のビルトイン C コンパイラの `apcc` が使用され、この問題が自動的に回避されます。このコンパイラは `gcc` と互換性がありますが、任意精度型と演算は正しく解釈されます。`apcc` コンパイラは、コマンド プロンプトで `gcc` を `apcc` に変更すると起動できます。

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

C コードで任意精度型を使用すると、C デザインを Vivado HLS の C デバッガーで解析できなくなります。ザイリンクスでは、デザインをデバッグする必要がある場合、次のいずれかの方法に従うことをお勧めしています。

- `printf` または `fprintf` 関数を使用して解析用にデータ値を出力します。
- 任意精度型をネイティブ C 型 (`int`、`char`、`short` など) に置き換えます。この方法を使用すると、アルゴリズム自体の演算はデバッグしやすくなりますが、そのアルゴリズムのビット精度結果を解析する必要がある場合には役立ちません。
- C 関数を C++ に変更し、デバッガーの制限がない C++ 任意精度型を使用します。

整数拡張

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;
```

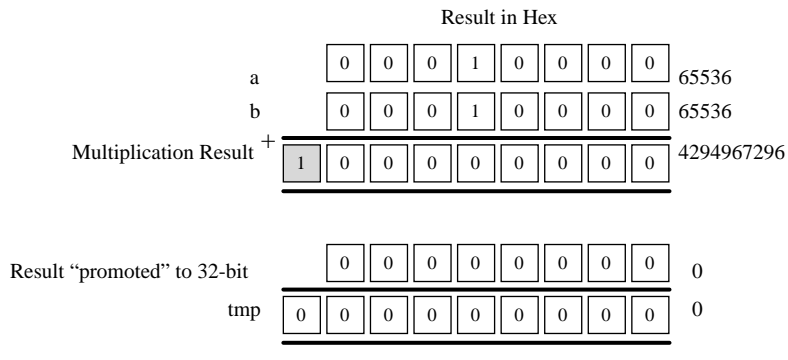
整数拡張は、次の方法を使用すると発生します。このために結果が予測と異なってしまう可能性があります。

整数拡張では、C コンパイラで次が実行されます。

- 乗算入力をネイティブ ビット サイズ (32 ビット) に拡張。
- 32 ビットの結果を生成する乗算を実行。
- その結果を 32 ビット変数の `tmp` に代入。

これにより、次の図に示すような動作になり、正しい結果になりません。

図 82: 整数拡張



Vivado HLS では、C シミュレーションと同じ結果になるので、Vivado HLS は 32 ビットの乗算結果を 36 ビットの結果に符号拡張したハードウェアを作成します。

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。次は、乗算前に乗算器への入力を 36 ビット値に型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 36 ビット乗算になります。

次の例は、整数拡張を回避するための型変換の方法を示しています。

```
#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;
}
```

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) よりも大きくなる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。

C の任意精度整数型: リファレンス情報

[C の任意精度型](#) には、次の情報が含まれています。

- 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 (シフト値が負の場合は反対方向にシフト) の記述も含めた演算子の動作の詳細。

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。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: デフォルトの幅の最大許容幅は 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ます。SystemC ヘッダー ファイルの `systemc.h` を含め、SystemC データ型を使用します。

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ため、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 (apcc 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。

ファイル拡張子が `.c` から `.cpp` に変更されることはよくあるので、これらの問題がない場合はファイルを C++ としてコンパイルできます。

C++ 言語では、ヘッダー ファイル `ap_int.h` で任意精度の整数型 `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。

次のコード例は、基本演算例 (標準データ型) のコードを繰り返したものです。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;

}
```

前の例をアップデートしたこの例では、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。

標準データ型 の基本演算の例と比較すると、入力データ型が実際の入力データの最大サイズを表すよう削減されています (8 ビット入力 `inA` が 6 ビット入力に削減されているなど)。出力はより正確になるよう調整されています。たとえば `inA` と `inB` の合計である `out2` に必要なのは、32 ビットではなく 13 ビットのみです。

次に、C++ 任意精度型を使用した基本演算の例を示します。

```
#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
```

C++ の任意精度整数型 が合成されると、標準データ型 および C の任意精度データ型の利点 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 C の任意精度データ型の利点 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` 関数で使わ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());
```

C++ の任意精度整数型: リファレンス情報

メソッド、合成動作、`ap_(u)int<N>` 任意精度小数点型を使用する場合の詳細は、C++ の任意精度型 を参照してください。このセクションには、次の項目が含まれます。

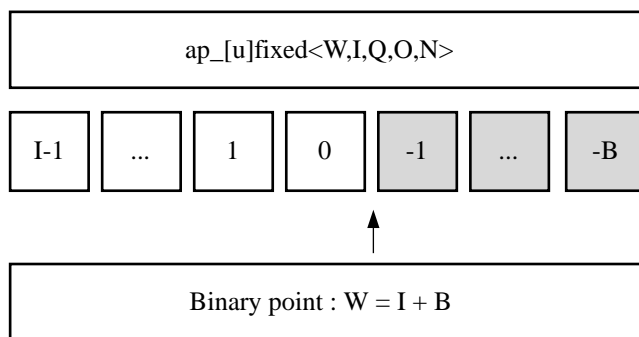
- 任意精度の整数に (1024 ビットを超える値も含めて) 定数および初期値を代入する方法。
- 表示、連結、ビット スライス、範囲選択などの Vivado HLS のヘルパー メソッドの詳細。
- シフト演算 (シフト値が負の場合は反対方向にシフト) の記述も含めた演算子の動作の詳細。

C++ の任意精度固定小数点型

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活かすことができます。次の図は、これらの固定小数点型の基本的な機能についてまとめています。

- 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 を定義できます。
- 整数部の桁数 I を指定すると、ワードの小数部 $W-I$ (次の図の B) も定義されます。
- 丸めまたは量子化 (Q) のタイプを選択できます。
- オーバーフロー動作 (O および N) を選択できます。

図 83: 任意精度固定小数点型



X14233



ヒント: 任意精度の固定小数点型は、コードにヘッダー ファイル `ap_fixed.h` が含まれていると使用できます。

任意精度の固定小数点型の方がCシミュレーション中により多くのメモリを必要とします。ap-[u]fixed型のか
なり大きな配列を使用する場合は、[配列](#)のCシミュレーションの説明を参照してください。

固定小数点型を使用すると、次のような利点があります。

- 小数を簡単に記述できます。
- 整数部および小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処理されます。
- 結果を精度を示すのに小数部ビットが足りない場合など、多くの丸めの実行を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処理するオプションもあります。

これらの属性は、次のコード例に含まれています。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 小数部。
- 6 ビット入力: 3 ビット整数値 + 3 小数部。
- 累算用の 22 ビット変数: 22 ビット入力: 17 ビット整数値 + 5 小数部。
- 結果用の 36 ビット変数: 22 ビット入力: 30 ビット整数値 + 6 小数部。

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。

次のコードのサンプルは、ap_fixed 型を示しています。

```
#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;
}
```

ap-(u)fixed 型を使用すると、C++ シミュレーションがビット精度になります。高速シミュレーションを使用する
と、アルゴリズムとその精度を検証できます。合成後、RTL では同じビット精度の動作になります。

任意精度の固定小数点型を使用すると、コードにリテラル値を自由に代入できます。上記の例で使用されたテストベ
ンチに示すように、in1 および in2 の値は宣言済みで、定数値が代入されます (次の例を参照)。

演算子が関係する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。この
ようにしないと、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。次の例に示すように、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。

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;
```

```
#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);
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;
}
```

固定小数点の識別子のまとめ

次の表は、量子化モードとオーバーフロー モードを示しています。



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 ビヘイビアー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ます。負の無限大への丸め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 25: 固定小数点の識別子のまとめ

識別子	説明	
W	ワード長をビット数で指定	
I	整数値のビット数 (整数部のビット数)	
Q	量子化モード: 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高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
	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 (デフォルト)
	AP_TRN_ZERO	0 への切り捨て
O	オーバーフロー モード: 演算結果が結果を格納するのに使用される変数の最大値を超え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	符号絶対値のラップ
N	折り返しモードでの飽和ビット数。	

C++ の任意精度固定小数点型: リファレンス情報

メソッド、合成動作、`ap_(u)fixed<N>` 固定小数点型を使用する場合の詳細は、[C++ の任意精度固定小数点型](#) を参照してください。このセクションには、次の項目が含まれます。

- 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。
- オーバーフロー モードおよび飽和モードの詳細。
- 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 (シフト値が負の場合は反対方向にシフト) の記述も含めた演算子の動作の詳細。



重要: 処理するコンパイラには、その言語の適切なヘッダー ファイルを使用する必要があります。

HLS ストリーム ライブラリ

ストリーミング データとは、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のことです。ストリーミングには、アドレス管理は必要ありません。

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。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。

- C コードでは、 `hls::stream<>` が無限深さの FIFO のように動作します。 `hls::stream<>` のサイズを定義する必要はありません。
- これらは、シーケンシャルに読み出されて書き込まれます。つまり、データが `hls::stream<>` から読み出されると、それが再び読み出されることはありません。
- 最上位インターフェイスの `hls::stream<>` は `ap_fifo` インターフェイスを使用してデフォルトでインプリメントされます。
- `hls::stream<>` は、深さ 2 の FIFO としてインプリメントされます。このデフォルト サイズを変更するには、最適化指示子の `STREAM` を使用します。

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。このセクションには、次のトピックが含まれます。

- 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。
- ブロッキング読み出しおよび書き込み。
- ノンブロッキング読み出しおよび書き込み。
- FIFO の深さの制御。

注記: `hls::stream` クラスは常に C++ 参照引数として関数間で使用されます。たとえば、 `&my_stream` がその例です。



重要: `hls::stream` クラスは、C++ デザインでのみ使用されます。 `stream` 配列はサポートされません。

C モデルと 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モデル化されます。ストリームを C++ でシミュレーションするために、深さを指定する必要はありません。ストリームは関数内または関数へのインターフェイスに使用できます。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。

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。

`hls::stream` が最上位インターフェイスに使用されると、RTL ではデフォルトで FIFO インターフェイス (`ap_fifo`) としてインプリメントされますが、ハンドシェイク インターフェイス (`ap_hs`) または AXI-Stream インターフェイス (`axis`) としてもインプリメントできます。

`hls::stream` がデザイン関数内で使用され、ハードウェアに合成される場合は、デフォルトで深さ 2 の FIFO としてインプリメントされます。補間が使用される場合など、ハードウェアで生成されるエレメントすべてが保持できるように FIFO の深さを増加する必要のあることもあります。ハードウェアで生成されるデータ サンプルすべてを保持するのに十分な大きさがないと、FIFO が停止する可能性があります (C/RTL 協調シミュレーションおよびハードウェアインプリメンテーションで確認できます)。FIFO の深さは、STREAM 指示子に `depth` オプションを使用して調整できます。この例は、デザイン例 `hls_stream` に示されています。



重要: デフォルトの DATAFLOW 以外の領域で `hls::stream` 変数を使用する場合は、正しくサイズ指定するようにしてください。

`hls::stream` をタスク (サブ関数またはループ) 間のデータ転送に使用する場合は、データが 1 つのタスクから次のタスクに流れる DATAFLOW 領域にタスクをインプリメントすることを考慮してみてください。デフォルト (DATAFLOW 以外) では、各タスクが次のタスクの開始前に完了するので、`hls::stream` 変数をインプリメントするのに使用した FIFO のサイズをプロデューサー タスクで生成されるデータ サンプルすべてを保持するのに十分な大きさにする必要があります。`hls::stream` 変数のサイズを増加しないと、次のようなエラー メッセージが表示されます。

```
ERROR: [XFORM 203-733] An internal stream xxxx.xxxx.V.user.V' with default
size is
used in a non-dataflow region, which may result in deadlock. Please
consider to
resize the stream using the directive 'set_directive_stream' or the 'HLS
stream'
pragma.
```

このエラー メッセージは、DATAFLOW でない領域 (デフォルトの FIFO の深さ 2) がプロデューサー タスクにより FIFO に書き込まれるデータ サンプルすべてを保持するのに十分な大きさではない可能性があることを示しています。

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 ストリームは内部 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 スコープで定義されるストリームは、その他のグローバル変数と同じ規則に従います。

HLS ストリームの使用

`hls::stream<>` オブジェクトを使用するには、`hls_stream.h` ヘッダー ファイルを含めます。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おり、`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
```

ストリームでは、スコープ付きの命名規則を使用する必要があるため、ザイリンクスでは、上記の例のようにスコープ付き (`hls::`) 命名規則を使用することを勧めますが、`hls` 名前空間を使用する場合は、前述の例を次のように書き直してください。

```
#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
```

ストリームを `hls::stream<T>` として指定すると、`T` のデータ型は次のいずれかになります。

- C++ ネイティブ データ型
- Vivado HLS 任意精度型 (例: `ap_int<>`、`ap_ufixed<>`)
- 上記のいずれかの型を含んだユーザー定義の構造体

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。

ストリームには、オプションで名前を付けることができます。ストリームに名前を定義すると、その名前がレポートで使用されるようになります。たとえば、Vivado HLS では入力ストリームからのすべてのエレメントがシミュレーション中に読み出されるようになっているかどうか自動的に確認されます。次の2つのストリームがあるとします。

```
stream<uint8_t> bytestr_in1;
stream<uint8_t> bytestr_in2("input_stream2");
```

ストリームに残ったままのエレメントがあれば、次のように警告メッセージが表示されます。この例の場合、`bytestr_in2` に関するメッセージが表示されています。

```
WARNING: Hls::stream 'hls::stream<unsigned char>.1' contains leftover data,
which
may result in RTL simulation hanging.
WARNING: Hls::stream 'input_stream2' contains leftover data, which may
result in RTL
simulation hanging.
```

ストリームが関数を出入りする際は、次の例のようにリファレンスごとに渡される必要があります。

```
void stream_function (
    hls::stream<uint8_t> &strm_out,
    hls::stream<uint8_t> &strm_in,
    uint16_t strm_len
)
```

Vivado HLS では、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両方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 ポートと AXI4-Stream リソースを使用して定義されるストリーミング ポートには、ノンブロッキング アクセスを使用しないでください。

ストリームを使用したデザイン全体の例は、Vivado HLS のサンプル デザイン例に含まれます。GUI の Welcome ページからデザイン例に含まれる `hls_stream` の例を参照してください。

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 からの読み出しや、フル ストリーム FIFO への書き込みが実行されようとした場合、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(ブロック)します。

トランザクションの進捗なしにシミュレータの実行が続くと、C/RTL 協調シミュレーションが停止することがあります。次は、停止をする典型的な例で、RTL シミュレーション時間が増加しても内部トランザクションの進捗がありません。

```
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transaction
Progress"] @
"Simulation Time"
////////////////////////////////////
// RTL Simulation : 0 / 1 [0.00%] @ "110000"
// RTL Simulation : 0 / 1 [0.00%] @ "202000"
// RTL Simulation : 0 / 1 [0.00%] @ "404000"
```

ブロッキング書き込みメソッド

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、左側に引数として、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;
```

ブロッキング読み出しメソッド

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、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、空のストリームからの読み出しやフル ストリームへの書き込みがあっても、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返します (問題ない場合は `true`、問題がある場合は `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。



重要: ノンブロッキング ビヘイビアは、`ap_fifo` プロトコルを使用したインターフェイスでのみサポートされます。つまり、AXI-Stream 規格とザイリンクスの `ap_hs` IO プロトコルでは、ノンブロッキング アクセスはサポートされません。

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。



重要: ブロック レベルの I/O プロトコルの `ap_ctrl_none` を使用するように指定されていて、デザインにブロッキング以外の動作を使用する `hls::stream` が含まれる場合は、C/RTL 協調シミュレーションが必ず終了するとは限りません。

ノンブロッキング書き込み

このメソッドは、`src_var` 変数を `my_stream` ストリームに挿入し、正しく挿入された場合はブール値 `true` が返されます。それ以外の場合は `false` が返され、キューは変更されません。

```
// 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` を返します。

```
// 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)
```

このメソッドはストリームから値を読み出し、正しく読み出せた場合は `true` が返されます。それ以外の場合は `false` が返され、キューは変更されません。

```
// 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)
```

`hls::stream<>` が空の場合に `true` を返します。

```
// Usage of bool empty(void)

hls::stream<int> my_stream;
int dst_var;
bool stream_empty;

stream_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、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;
}

```

RTL の FIFO の深さの制御

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 2 は十分な深さです。これは、ストリーミング データが通常 一度に 1 サンプルを処理するからです。

インプリメンテーションで FIFO の深さが 2 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は停止します。

このため、ストリーム オブジェクトは GUI の [Directive] タブには表示されないので、STREAM 指示子を [Directive] タブからは直接適用できません。

`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) 関数で右クリックします。

- STREAM 指示子を選択します。
- `variable` 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、次のいずれかを実行します。

- `directives.tcl` ファイルで STREAM 指示子を手動で指定します。
- `source` に `pragma` として追加します。

C/RTL 協調シミュレーションのサポート

Vivado HLS の C/RTL 協調シミュレーション機能では、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はサポートされません。Vivado HLS ではこれらの構造およびクラスは合成でサポートされます。

```

typedef struct {
    hls::stream<uint8_t> a;
    hls::stream<uint16_t> b;
} strm_struct_t;

void dut_top(strm_struct_t indata, strm_struct_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。内部リンケージのみの `hls::stream<>` オブジェクトには、このような制限はありません。

HLS 数学ライブラリ

Vivado HLS の `math` ライブラリ (`hls_math.h`) は、標準 C (`math.h`) および C++ (`cmath.h`) ライブラリの合成をサポートするために提供されており、合成中の算術演算を指定するために自動的に使用されます。すべての関数の浮動小数点と一部の関数の固定小数点がサポートされます。

`hls_math.h` ライブラリはオプションで標準 C `math` ライブラリ (`cmath.h`) の代わり C++ ソース コードで使用できますが、C ソース コードでは使用できません。Vivado HLS では、適切なシミュレーション インプリメンテーションを使用して、C シミュレーションと C/RTL 協調シミュレーション間で精度に違いがでないようにされます。

HLS math ライブラリの精度

HLS 数学関数は、`hls_math.h` から合成可能なビット概算関数としてインプリメントされます。ビット概算の HLS `math` ライブラリ関数の精度は、標準 C 関数の精度と同じにはなりません。ビット概算インプリメンテーションでは、標準 C `math` ライブラリ の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。関数の精度は ULP (Unit of Least Precision) で指定されます。この精度の違いは、C シミュレーションと C/RTL 協調シミュレーションの両方に影響します。

ULP の違いは、通常 1-4 ULP の範囲です。

- 標準 C `math` ライブラリが C ソース コードで使用されると、関数の中に標準 C `math` ライブラリとは ULP の異なるものがあるため、C シミュレーションと C/RTL 協調シミュレーション間で違いがあることがあります。
- HLS `math` ライブラリが C ソース コードで使用される場合は、C シミュレーションと C/RTL 協調シミュレーション間で違いはありませんが、HLS `math` ファイルを使用した C シミュレーションは、標準 C `math` ライブラリを使用した C シミュレーションとは異なることがあります。

また、次の 7 つの関数の場合、コンパイルおよび C シミュレーションの実行に使用する C 標準によって精度が異なることがあります。

- `copysign`
- `fpclassify`
- `isinf`
- `isfinite`
- `isnan`
- `isnormal`
- `signbit`

C90 モード

通常 `isinf`、`isnan`、および `copysign` だけがシステム 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返します。これにより、浮動小数点を返さなければならない場合に、倍制度から浮動小数点への変換ブロックがハードウェアに導入されるので、合成後に予期しない結果になることがあります。

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の元を提供されています。通常の GCC ファイルは `isnormal` をリダイレクトはしませんが、それを `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。

`copysign` は常に倍精度の結果を返します。これにより、浮動小数点を返さなければならない場合に、倍精度から浮動小数点への変換ブロックがハードウェアに導入されるので、合成後に予期しない結果になることがあります。

cmath を使用した C++

C99 モード (`mode(-std=c99)`) と同様ですが、次の点が異なります。

- システムヘッダーファイルが通常異なります。
- 関数は、次の場合適切にオーバーロードされます。

```
◦ float(). snan(double)

◦ isinf(double)
```

`copysign` および `copysignf` は `namespace std;` を使用した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。最適な結果となるので、ザイリンクスでは次の使用をお勧めします。

- C の場合は `-std=c99`
- C および C++ の場合は `-fno-builtin`

注記: `-std=c99` などの C コンパイル オプションを指定するには、Tcl コマンドの `add_files` に `-cflags` オプションを使用します。または、[Project Settings] ダイアログ ボックスの [Edit CFLAGS] ボタンを使用します。

HLS math ライブラリ

HLS ビデオ ライブラリには、次の関数が含まれます。すべての関数で半精度型 (`half`)、単精度型 (`float`)、および倍精度型 (`double`) がサポートされます。



重要: 次にリストする関数 `func` には、`half_func` という半精度のみの関数と `funcf` という単精度のみの関数もライブラリに含まれます。

半精度型、単精度型、倍精度型を混合して使用する場合は、最終的な FPGA インプリメンテーションで型変換ハードウェアが使用されないように、よくある合成エラーを確認してください。

三角関数

<code>acos</code>	<code>acospi</code>	<code>asin</code>	<code>asinpi</code>
<code>atan</code>	<code>atan2</code>	<code>atan2pi</code>	<code>cos</code>
<code>cospi</code>	<code>sin</code>	<code>sincos</code>	<code>sinpi</code>
<code>tan</code>	<code>tanpi</code>		

双曲線関数

<code>acosh</code>	<code>asinh</code>	<code>atanh</code>	<code>cosh</code>
<code>sinh</code>	<code>tanh</code>		

指数関数

exp	exp10	exp2	expm1
frexp	ldexp	modf	

対数関数

ilogb	log	log10	log1p
-------	-----	-------	-------

べき関数

cbrt	hypot	pow	rsqrt
sqrt			

誤差関数

erf	erfc
-----	------

丸め関数

ceil	floor	llrint	llround
lrint	lround	nearbyint	rint
round	trunc		

剰余関数

fmod	remainder	remquo
------	-----------	--------

浮動小数点

copysign	nan	nextafter	nexttoward
----------	-----	-----------	------------

差分関数

fdim	fmax	fmin	maxmag
minmag			

その他の関数

abs	divide	fabs	fma
fract	mad	recip	

分類関数

fpclassify	isfinite	isinf	isnan
isnormal	signbit		

比較関数

isgreater	isgreaterequal	isless	islessequal
islessgreater	isunordered		

関係関数

all	any	bitselect	isequal
isnotequal	isordered	select	

固定小数点の数学関数

固定小数点のインプリメンテーションは、次の数学関数用にも提供されています。

固定小数点の数学関数すべてで、次のビット幅仕様の `ap_[u]fixed` および `ap_[u]int` 型がサポートされます。

1. `ap_fixed<W, I>` ($I \leq 33$ および $W - I \leq 32$)
2. `ap_ufixed<W, I>` ($I \leq 32$ および $W - I \leq 32$)
3. `ap_int<I>` ($I > 33$)
4. `ap_uint<I>` ($I > 32$)

三角関数

cos	sin	tan	acos	asin	atan	atan2	sincos
cospi	sinpi						

双曲線関数

cosh	sinh	tanh	acosh	asinh	atanh
------	------	------	-------	-------	-------

指数関数

exp	frexp	modf	exp2	expm1
-----	-------	------	------	-------

対数関数

log	log10	ilogb	log1p
-----	-------	-------	-------

べき関数

pow	sqrt	rsqrt	cbrt	hypot
-----	------	-------	------	-------

誤差関数

erf	erfc
-----	------

丸め関数

ceil floor trunc round rint nearbyint

固定小数点

nextafter nexttoward

差分関数

erf erfc fdim fmax fmin maxmag minmag

その他の関数

fabs recip abs fract divide

分類関数

signbit

比較関数

isgreater isgreaterequal isless islessequal islessgreater

関係関数

isequal isnotequal any all bitselect

固定小数点型の場合、関数値の精度は少し落ちますが、RTL インプリメンテーションはより小さく高速になります。

固定小数点型の数学関数をインプリメントするには、次の手順に従います。

1. 固定小数点のインプリメンテーションがサポートされるかどうかを判断します。
2. 数学関数を `ap_fixed` 型を使用してアップデートします。
3. C シミュレーションを実行して、デザインが必要な精度でまだ動作するかどうかを検証します。C シミュレーションは、RTL インプリメンテーションと同じビット精度型を使用して実行されます。
4. デザインを合成します。

たとえば、`sin` 関数の固定小数点インプリメンテーションは、次のように数学関数と共に固定小数点型を使用して指定されます。

```
#include "hls_math.h"
#include "ap_fixed.h"

ap_fixed<32,2> my_input, my_output;

my_input = 24.675;
my_output = sin(my_input);
```

固定小数点の数学関数を使用する場合、結果の型が入力と同じ幅と整数ビットである必要があります。

検証および数学関数

標準 C math ライブラリが C ソースで使用されると、C シミュレーション結果と C/RTL 協調シミュレーション結果が異なることがあります。ソース コードの数学関数のいずれかに標準 C math ライブラリと異なる ULP がある場合、RTL がシミュレーションされたときの結果が異なってしまうことがあります。

hls_math.h ライブラリが C コードに含まれると、C シミュレーションと RTL 協調シミュレーションの結果は同じになります。ただし、hls_math.h を使用した C シミュレーションの結果は、標準 C ライブラリを使用した結果と同じにはなりません。hls_math.h ライブラリを使用すると、単に C シミュレーションが C/RTL 協調シミュレーション結果と一致するようになります。どちらの場合も同じ RTL インプリメンテーションが作成されます。次に、数学関数を使用した場合に検証を実行するために使用する可能性のあるオプションについて説明します。

検証オプション 1: 標準 math ライブラリおよび差の検証

このオプションでは、ソース コードで標準 C math ライブラリが使用されます。合成されるいずれかの関数に正確な精度がある場合は、C/RTL 協調シミュレーションは C シミュレーションと異なります。次にこの方法の例を示します。

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

この場合、C シミュレーションと C/RTL 協調シミュレーションの結果は異なります。テストベンチで生成される結果は、シミュレーションを実行した作業ディレクトリに保存されます。

- C シミュレーション:<project>/<solution>/csim/build フォルダー
- C/RTL 協調シミュレーション:<project>/<solution>/sim/<RTL> フォルダー

<project> はプロジェクト フォルダー、<solution> はソリューション フォルダーの名前、<RTL> は検証される RTL のタイプ (verilog または vhdl) です。次の図に、合成前の結果ファイル (左側) と合成後の RTL 結果ファイル (右側) の典型的な比較を示します。出力は 3 列目に表示されています。

図 84: 合成前後のシミュレーションと合成後のシミュレーションの違い

	result.dat		proj_cpp_math.prj/solution1/sim/systemc/result.dat
1	0.0000000000000000	1	0.0000000000000000
2	1.0000000000000000	2	1.0000000000000000
3	2.0000000000000000	3	2.0000000000000000
4	3.0000000000000000	4	3.0000000000000000
5	4.0000000000000000	5	4.0000000000000000
6	5.0000000000000000	6	5.0000000000000000
7	6.0000000000000000	7	6.0000000000000000
8	7.0000000000000000	8	7.0000000000000000
9	8.0000000000000000	9	8.0000000000000000
10	9.0000000000000000	10	9.0000000000000000
11	10.0000000000000000	11	10.0000000000000000
12	11.0000000000000000	12	11.0000000000000000
13	12.0000000000000000	13	12.0000000000000000
14	13.0000000000000000	14	13.0000000000000000
15	14.0000000000000000	15	14.0000000000000000
16	15.0000000000000000	16	15.0000000000000000
17	16.0000000000000000	17	16.0000000000000000
18	17.0000000000000000	18	17.0000000000000000
19	18.0000000000000000	19	18.0000000000000000
20	19.0000000000000000	20	19.0000000000000000
21	20.0000000000000000	21	20.0000000000000000
22	21.0000000000000000	22	21.0000000000000000
23	22.0000000000000000	23	22.0000000000000000
24	23.0000000000000000	24	23.0000000000000000
25	24.0000000000000000	25	24.0000000000000000
26	25.0000000000000000	26	25.0000000000000000
27	26.0000000000000000	27	26.0000000000000000
28	27.0000000000000000	28	27.0000000000000000
29	28.0000000000000000	29	28.0000000000000000
30	29.0000000000000000	30	29.0000000000000000

合成前のシミュレーションと合成後のシミュレーションの結果の差はわずかです。この差が最終 RTL インプリメンテーションで許容できるものであるかどうかは、ユーザーが判断する必要があります。

これらの差を処理するには、結果をチェックするテストベンチを使用し、エラーの許容範囲内に収まっているかどうかを確認するのが推奨されるフローです。これには、同じ関数を 2 バージョン (合成用に 1 つ、参照用に 1 つ) 作成します。次の例では、cpp_math 関数のみが合成されます。

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}

data_t cpp_math_sw(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

デザインを検証するテストベンチでは、次の例のように diff 変数を使用し、両方の関数の出力を比較してその差が判断されます。C シミュレーションでは、両方の関数の出力は同じになります。C/RTL 協調シミュレーションでは、cpp_math 関数で異なる結果が出力され、その差がチェックされます。

```
int main() {
    data_t angle = 0.01;
    data_t output, exp_output, diff;
    int retval=0;
```

```

for (data_t i = 0; i <= 250; i++) {
    output = cpp_math(angle);
    exp_output = cpp_math_sw(angle);

    // Check for differences
    diff = ( (exp_output > output) ? exp_output - output : output -
exp_output);
    if (diff > 0.00000005) {
        printf("Difference %.10f exceeds tolerance at angle %.10f \n", diff,
angle);
        retval=1;
    }

    angle = angle + .1;
}

if (retval != 0) {
    printf("Test failed !!!\n");
    retval=1;
} else {
    printf("Test passed !\n");
}
// Return 0 if the test passes
return retval;
}

```

差のマージンを 0.00000005 に下げると、このテストベンチで C/RTL 協調シミュレーション中の差が示されます。

```

Difference 0.00000000596 at angle 1.1100001335
Difference 0.00000000596 at angle 1.2100001574
Difference 0.00000000596 at angle 1.5100002289
Difference 0.00000000596 at angle 1.6100002527
etc..

```

標準 C math ライブラリ (math.h および cmath.h) を使用する場合、精度の差が許容可能かどうかを検証するためのスマート テストベンチを作成します。

検証オプション 2: HLS math ライブラリおよび差の検証

別の検証オプションは、ソース コードを HLS math ライブラリを使用するように変換する方法です。このオプションを使用する場合、C シミュレーションと C/RTL 協調シミュレーション結果に差はなくなります。次の例では、上記のコードを hls_math.h ライブラリを使用するように変更しています。

注記: このオプションは、C++ でのみ使用可能です。

- hls_math.h ヘッダー ファイルを含めます。
- 数学関数を同等の hls:: 関数に置き換えます。

```

#include <cmath>
#include "hls_math.h"
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

```

```
data_t cpp_math(data_t angle) {  
    data_t s = hls::sinf(angle);  
    data_t c = hls::cosf(angle);  
    return hls::sqrtf(s*s+c*c);  
}
```

検証オプション 3: HLS math ライブラリ ファイルおよび差の検証

HLS math ライブラリ ファイル `lib_hlsm.cpp` をデザイン ファイルとして含めると、Vivado HLS で C シミュレーションに HLS math ライブラリが使用されます。このオプションはオプション 2 と同じですが、C コードを変更する必要はありません。

HLS math ライブラリ ファイルは、Vivado HLS インストール ディレクトリの `src` ディレクトリに含まれます。ファイルをローカル フォルダにコピーして、ファイルを標準のデザイン ファイルとして追加します。

注記: このオプションは、C++ でのみ使用可能です。

このオプションを使用すると、オプション 2 を使用した場合と同様に HLS math ファイルを使用した C シミュレーション結果と、このファイルを追加しない前に取得したシミュレーション結果が異なるものになります。これらの差は、オプション 1 と同様のスマート テストベンチを使用して C シミュレーションで検証する必要があります。

よくある合成エラー

次は数学関数を合成する際によく発生する使用エラーです。これらは数学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`

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` および `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`

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、単精度関数 (`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。

注意

`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次のコードを C++ コードに追加する必要があります。

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

「`float` および `double` 型」で説明したように、`float` および `double` 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な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



重要: Vivado® HLS ビデオ ライブラリは、ザイリンクス GitHub (<https://github.com/Xilinx/xfopencv>) に移動されました。

HLS IP ライブラリ

Vivado HLS には、多くのザイリンクス IP ブロックをインプリメントするための C ライブラリが提供されています。この C ライブラリを使用すると、次のザイリンクス IP ブロックが C ソース コードから直接推論でき、FPGA で高品質のインプリメンテーションになるようになります。

表 26: HLS IP ライブラリ

ライブラリ ヘッダー ファイル	説明
hls_fft.h	ザイリンクス LogiCORE IP FFT を C でシミュレーション、ザイリンクス LogiCORE ブロックを使用してインプリメント可能。
hls_sslib.h	完全に合成可能なスーパー サンプル レート (SSR) FFT で毎クロック サイクルで複数の入力サンプルを処理可能。
hls_fir.h	ザイリンクス LogiCORE IP FIR を C でシミュレーション、ザイリンクス LogiCORE ブロックを使用してインプリメント可能。
hls_dds.h	ザイリンクス LogiCORE IP DDS を C でシミュレーション、ザイリンクス LogiCORE ブロックを使用してインプリメント可能。
ap_shift_reg.h	ザイリンクス SRL プリミティブを使用して直接インプリメントされるシフトレジスタをインプリメントするための C++ クラスを提供。

FFT IP ライブラリ

ザイリンクス FFT ブロックは、hls_fft.h ライブラリを使用すると C++ デザイン内で呼び出すことができます。このセクションでは、FFT を C++ コードで設定する方法を説明します。



推奨: ザイリンクスでは、この IP の機能のインプリメント方法および機能の使用方法については、『Fast Fourier Transform LogiCORE IP 製品ガイド』(PG109) を参照することをお勧めしています。

C++ コードで FFT を使用するには、次の手順に従います。

1. コードに hls_fft.h ライブラリを含めます。
2. 定義済み構造体 hls::ip_fft::params_t を使用してデフォルト パラメーターを設定します。
3. ランタイム コンフィギュレーションを定義します。
4. FFT 関数を呼び出します。
5. ランタイム ステータスをチェックします (オプション)。

次のコード例に、これらの各手順の実行方法を示します。各手順の詳細は、次のとおりです。

まず、ソース コードに FFT ライブラリを含めます。このヘッダー ファイルは、Vivado HLS のインストール ディレクトリの include ディレクトリに含まれています。このディレクトリは、Vivado HLS が実行されると自動的に検索されます。

```
#include "hls_fft.h"
```

FFT のスタティック パラメーターを定義します。これには、動的に変化しない入力幅、チャンネル数、アーキテクチャ タイプなどが含まれます。FFT ライブラリにはパラメーター指定構造体 `hls::ip_fft::params_t` が含まれ、すべてのスタティック パラメーターをデフォルト値で初期化できます。

この例では、出力順序、コンフィギュレーション ポートとステータス ポートの幅のデフォルト値を、定義済み構造体に基づくユーザー定義の構造体 `param1` を使用して変更しています。

```
struct param1 : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
    static const unsigned config_width = FFT_CONFIG_WIDTH;
    static const unsigned status_width = FFT_STATUS_WIDTH;
};
```

ランタイム コンフィギュレーションとランタイム ステータスのデータ型と変数を定義します。これらの値は動的に変化する可能性があるため、変更可能で API からアクセス可能な C コードの変数として定義します。

```
typedef hls::ip_fft::config_t<param1> config_t;
typedef hls::ip_fft::status_t<param1> status_t;
config_t fft_config1;
status_t fft_status1;
```

次に、ランタイム コンフィギュレーションを設定します。この例では、`direction` 変数の値に基づいて FFT の方向 (前方向または逆方向) を設定し、スケーリング スケジュールの値も設定しています。

```
fft_config1.setDir(direction);
fft_config1.setSch(0x2AB);
```

HLS 名前空間と定義済みスタティック コンフィギュレーション (この例では `param1`) を使用して FFT 関数を呼び出します。関数のパラメーターは、順に入力データ、出力データ、出力ステータス、入力コンフィギュレーションを示します。

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

最後に出力ステータスをチェックします。この例では、オーバーフロー フラグがチェックされて、結果が `ovflo` 変数に格納されます。

```
*ovflo = fft_status1->getOvflo();
```

FFT C ライブラリを使用したデザイン例は、[Help] → [Welcome] → [Open Example Project] → [Design Examples] → [FFT] をクリックして表示される Vivado HLS のサンプルに含まれます。

FFT のスタティック パラメーター

FFT のスタティック パラメーターは、FFT の設定方法を定義し、FFT のサイズ、サイズを動的に変更可能にするか、インプリメンテーションをパイプライン処理するか `radix_4_burst_io` を使用するかなどの固定パラメーターを指定します。

hls_fft.h ヘッダー ファイルでは、スタティック パラメーターのデフォルト値を設定するのに使用可能な hls::ip_fft::params_t という構造体が定義されています。デフォルト値を使用する場合は、FFT 関数と共にパラメーター指定構造体を直接使用できます。

```
hls::fft<hls::ip_fft::params_t >
(xn1, xk1, &fft_status1, &fft_config1);
```

より一般的には、パラメーターの一部がデフォルト値以外の値に変更されます。これには、デフォルトのパラメーター指定構造体に基づく新しいユーザー定義のパラメーター指定構造体を作成し、一部のデフォルト値を変更します。

次の例では、新しいユーザー構造体 my_fft_config を定義し、出力順序を新しい値 (natural_order に変更) に変更しています。FFT のその他すべてのスタティック パラメーターにはデフォルト値が使用されます。

```
struct my_fft_config : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
};

hls::fft<my_fft_config >
(xn1, xk1, &fft_status1, &fft_config1);
```

パラメーター指定構造体 hls::ip_fft::params_t については、[FFT の構造体パラメーター](#)を参照してください。パラメーターのデフォルト値および使用可能な値のリストは、[FFT の構造体パラメーターの値](#)を参照してください。



推奨: サイリンクスでは、パラメーターの詳細およびその設定の影響については、『Fast Fourier Transform LogiCORE IP 製品ガイド』([PG109](#))を参照することをお勧めしています。

FFT の構造体パラメーター

表 27: FFT の構造体パラメーター

パラメーター	説明
input_width	データ入力ポート幅。
output_width	データ出力ポート幅。
status_width	出力ステータス ポート幅。
config_width	入力コンフィギュレーション ポート幅。
max_nfft	FFT データ セットのサイズを $1 \leq \text{max_nfft}$ に指定します。
has_nfft	FFT のサイズをランタイムに設定できるようにするかどうかを指定します。
channels	チャンネル数。
arch_opt	インプリメンテーション アーキテクチャ。
phase_factor_width	内部位相係数の精度を設定します。
ordering_opt	出力順序モード。
ovflo	オーバーフロー モードをイネーブルにします。
scaling_opt	スケーリング オプションを定義します。
rounding_opt	丸めモードを定義します。
mem_data	データ メモリにブロック RAM を使用するか分散 RAM を使用するかを指定します。

表 27: FFT の構造体パラメーター (続き)

パラメーター	説明
mem_phase_factors	位相係数メモリにブロック RAM を使用するか分散 RAM を使用するかを指定します。
mem_reorder	出力順序並べ替えメモリにブロック RAM を使用するか分散 RAM を使用するかを指定します。
stages_block_ram	インプリメンテーションで使用されるブロック RAM の段数を定義します。
mem_hybrid	ブロック RAM がデータ、位相係数、または順序並べ替えバッファに指定されている場合に、ブロック RAM と分散 RAM のハイブリッドを使用して特定のコンフィギュレーションでのブロック RAM 数を削減するかどうかを指定します。
complex_mult_type	複素乗算器に使用する乗算器のタイプを定義します。
butterfly_type	FFT バタフライに使用されるインプリメンテーションを定義します。

整数またはブール型ではないパラメーター値を指定する際は、HLS FFT 名前空間を使用する必要があります。

たとえば、次の表の butterfly_type パラメーターに使用可能な値は use_luts および use_xtremesp_slices です。C プログラムでは butterfly_type = hls::ip_fft::use_luts および butterfly_type = hls::ip_fft::use_xtremesp_slices を使用する必要があります。

FFT の構造体パラメーターの値

次の表に、FFT IP のすべての機能を示します。この表に示されていない機能は、Vivado HLS のインプリメンテーションではサポートされていません。

表 28: FFT の構造体パラメーターの値

パラメーター	C データ型	デフォルト値	有効な値
input_width	unsigned	16	8 ~ 34
output_width	unsigned	16	input_width ~ (input_width + max_nfft + 1)
status_width	unsigned	8	FFT コンフィギュレーションによって異なる
config_width	unsigned	16	FFT コンフィギュレーションによって異なる
max_nfft	unsigned	10	3 ~ 16
has_nfft	bool	false	True、False
channels	unsigned	1	1 ~ 12
arch_opt	unsigned	pipelined_streaming_io	automatically_select pipelined_streaming_io radix_4_burst_io radix_2_burst_io radix_2_lite_burst_io
phase_factor_width	unsigned	16	8 ~ 34
ordering_opt	unsigned	bit_reversed_order	bit_reversed_order natural_order

表 28: FFT の構造体パラメーターの値 (続き)

パラメーター	C データ型	デフォルト値	有効な値
ovflo	bool	true	false true
scaling_opt	unsigned	scaled	scaled unscaled block_floating_point
rounding_opt	unsigned	truncation	truncation convergent_rounding
mem_data	unsigned	block_ram	block_ram distributed_ram
mem_phase_factors	unsigned	block_ram	block_ram distributed_ram
mem_reorder	unsigned	block_ram	block_ram distributed_ram
stages_block_ram	unsigned	(max_nfft < 10) ? 0 : (max_nfft - 9)	0 ~ 11
mem_hybrid	bool	false	false true
complex_mult_type	unsigned	use_mults_resources	use_luts use_mults_resources use_mults_performance
butterfly_type	unsigned	use_luts	use_luts use_xtremesp_slices

FFT ランタイム コンフィギュレーションとランタイム ステータス

FFT では、コンフィギュレーション ポートとステータス ポートを介したランタイム コンフィギュレーションおよびランタイム ステータスの監視がサポートされます。これらのポートは FFT 関数への引数として定義され、次の例では `fft_status1` および `fft_config1` 変数として記述されています。

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

ランタイム コンフィギュレーションとランタイム ステータスには、FFT の C ライブラリから定義済み構造体を使用してアクセスできます。

- `hls::ip_fft::config_t<param1>`
- `hls::ip_fft::status_t<param1>`

注記: どちらの場合も、構造体にはスタティック パラメーター指定構造体が必要で、例では `param1` と記述されています。スタティック パラメーター指定構造体の詳細は、前のセクションを参照してください。

ランタイム コンフィギュレーション構造体を使用すると、C コードで次を実行できます。

- ランタイム コンフィギュレーションがイネーブルの場合は、FFT 長を設定
- FFT の方向を前方向または逆方向に設定
- スケーリング スケジュールを設定

FFT 長は、次のように設定できます。

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Set FFT length to 512 => log2(512) =>9
fft_config1->setNfft(9);
```



重要: ランタイム中に指定する長さは、スタティック コンフィギュレーションの `max_nfft` で定義されているサイズを超えることはできません。

FFT の方向は、次のように設定できます。

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Forward FFT
fft_config1->setDir(1);
// Inverse FFT
fft_config1->setDir(0);
```

FFT のスケーリング スケジュールは、次のように設定できます。

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
fft_config1->setSch(0x2AB);
```

出力ステータス ポートには、次を判断するために、定義済み構造体を使用してアクセスできます。

- FFT 中にオーバーフローが発生したかどうか
- ブロックの指数値

FFT オーバーフロー モードは、次のようにチェックできます。

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Check the overflow flag
bool *ovflo = fft_status1->getOvflo();
```



重要: 各トランザクションが完了した後、オーバーフロー ステータスを確認して FFT が正しく動作していることを確認してください。

ブロックの指数値は、次を使用すると取得できます。

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Obtain the block exponent
unsigned int *blk_exp = fft_status1->getBlkExp();
```

FFT 関数の使用

FFT 関数は、HLS 名前空間で定義され、次のように呼び出すことができます。

```
hls::fft<STATIC_PARAM> (
    INPUT_DATA_ARRAY,
    OUTPUT_DATA_ARRAY,
    OUTPUT_STATUS,
    INPUT_RUN_TIME_CONFIGURATION);
```

STATIC_PARAM は、FFT のスタティック パラメーターを定義するスタティック パラメーター指定構造体です。

入力データおよび出力データは、配列 (INPUT_DATA_ARRAY および OUTPUT_DATA_ARRAY) として関数に供給されます。最終的なインプリメンテーションでは、FFT RTL ブロックのポートは AXI4-Stream ポートとしてインプリメントされます。データフロー最適化 (set_directive_dataflow) を使用した領域では、配列がストリーミング配列としてインプリメントされるので、ザイリンクスでは常に FFT 関数を使用することを勧めします。または、両方の配列を set_directive_stream コマンドを使用してストリーミングとして指定します。



重要: FFT は、パイプラインされた領域では使用できません。高度なパフォーマンスの演算が必要な場合、FFT の前後でループまたは関数をパイプライン処理して、その領域のループおよび関数すべてでデータフロー最適化を使用します。

配列のデータ型は、float または ap_fixed のいずれかにできます。

```
typedef float data_t;
complex<data_t> xn[FFT_LENGTH];
complex<data_t> xk[FFT_LENGTH];
```

固定小数点型を使用する場合は、Vivado HLS 任意精度型 ap_fixed を使用する必要があります。

```
#include "ap_fixed.h"
typedef ap_fixed<FFT_INPUT_WIDTH,1> data_in_t;
typedef ap_fixed<FFT_OUTPUT_WIDTH,FFT_OUTPUT_WIDTH-FFT_INPUT_WIDTH+1>
data_out_t;
#include <complex>
typedef hls::x_complex<data_in_t> cmpxDData;
typedef hls::x_complex<data_out_t> cmpxDDataOut;
```

どちらの場合でも、FFT のパラメーターを同じ正しいデータ サイズに指定する必要があります。浮動小数点型の場合、データ幅は常に 32 ビットで、その他のサイズを指定しても無効と判断されます。



重要: FFT の入力および出力幅は、サポートされる範囲内の任意の値に設定できます。入力および出力パラメーターに接続される変数は、8 ビット単位で定義する必要があります。たとえば出力幅を 33 ビットに設定する場合は、出力変数は 40 ビット変数として定義する必要があります。

FFT のマルチチャネル機能は、入力データと出力データの 2 次元配列を使用すると使用できます。この場合、最初の次元が各チャネル、2 つ目の次元が FFT データを表すように配列データを設定する必要があります。

```
typedef float data_t;
static complex<data_t> xn[CHANNEL][FFT_LENGTH];
static complex<data_t> xk[CHANNEL][FFT_LENGTH];
```

FFT コアは、データをインターリーブされたチャンネル (たとえば、ch0-data0、ch1-data0、ch2-data0 など、ch0-data1、ch1-data1、ch2-data2、など) として生成および消費します。このため、FFT の入力配列または出力配列をデータを読み出しおよび書き込みするのと同じ順序でストリーミングするには、次の例に示すように、まずチャンネルインデックスを順に処理することにより、複数のチャンネルの 2 次元配列を充填または空にする必要があります。

```
cmpxDat    in_fft[FFT_CHANNELS][FFT_LENGTH];
cmpxDat    out_fft[FFT_CHANNELS][FFT_LENGTH];

// Write to FFT Input Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        in_fft[j][i] = in.read().data;
    }
}

// Read from FFT Output Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        out.data = out_fft[j][i];
    }
}
```

FFT C ライブラリを使用したデザイン例は、[Help]→[Welcome]→[Open Example Project]→[Design Examples]→[FFT] をクリックして表示される Vivado HLS のサンプルに含まれます。

SSR FFT IP ライブラリ

概要

Vivado HLS には、毎クロック サイクルで複数の入力サンプルを処理可能なシストリック アーキテクチャを含む完全に合成可能なスーパー サンプルデータ レート (SSR) FFT が含まれます。各サイクルで並列で処理されるサンプル数は、SSR 係数で表されます。この FFT は C++ テンプレートの関数としてインプリメントされます。その構造は `ssr_fft_default_params` 型の C++ 構造体であるテンプレート パラメーターを使用してパラメーター指定できます。新しい構造は、デフォルト構造を拡張して、メンバー制約を次のように上書きすると定義できます。

```
struct ssr_fft_fix_params:ssr_fft_default_params
{
    static const int N=1024;
    static const int R=4;
    static const scaling_mode_enum scaling_mode=SSR_FFT_GROW_TO_MAX_WIDTH;
    static const fft_output_order_enum output_data_order=SSR_FFT_NATURAL;
    static const int twiddle_table_word_length=18;
    static const int twiddle_table_intger_part_length=2;
};
```

上記の構造では、次が定義されます。

- N: 変換のサイズまたは長さ
- R: 並列で処理されるサンプル数
- scaling_mode: 列挙型としてのスケーリング モード
- output_data_order: 出力データ順序を自然な順序にするか、桁が逆順にするかを定義
- twiddle_table_word_length: 回転テーブル因子を格納するために使用されるビット数合計を定義

- `twiddle_table_intger_part_length`: 回転の整数を格納するために使用される整数ビット数

ユーザー定義のC++ 構造体は、FFT を呼び出す際のテンプレート パラメーターとして使用できます。

```
hls::ssr_fft::fft<ssr_fft_fix_params>(...);
```

パフォーマンス

FFT スループット (開始間隔) は L/R で計算できます。R は SSR 値で、L は変換されるサンプル数です。R (SSR 値) に使用できる値は 2、4、8、16 です。これらの値を使用すると、一番遅い UltraScale+ スピードグレード デバイスをターゲットにした場合に、300-550 MHz の F_{max} が可能になります。

データ型

FFT は固定小数点データ型 (`std::complex<ap_fixed<>>`) に基づいており、合成およびインプリメンテーションで使用されます。それ以外の場合、浮動小数点をシミュレーションに使用できます。

データ ビット幅を 27 ビット (整数 + 分数) に制限すると、1 つの DSP ブロックに直接マップされるので、最適な結果になります。大きな入力を使用することもできますが、 F_{max} の速度が落ち、使用率が落ちることがあります。最後に、複素指数関数/回転因子ストレージは 18 ビット (16F+2I ビット) に設定されることに注意してください。18 ビットを選択すると、18x27 ビット乗算器を含むザイリンクス FPGA の DSP ブロックでその乗算器が使用できるようになります。

FFT 段中のデータ ビット増加の管理:

FFT では、FFT 段間のビット増加を管理する 3 つのモードがサポートされています。これら 3 つのモードは、各段でのビット増加を可能にしたり、ビット増加なしで各段でスケーリングを使用したり、27 ビットまでビット増加を可能にしてからスケーリングを使用し始めたりするのに使用できます。詳細は、次のとおりです。

- `SSR_FFT_GROW_TO_MAX_WIDTH`: パラメーター構造で `scaling_mode` 制約を `SSR_FFT_GROW_TO_MAX_WIDTH` に設定すると、最初の段から指定した最大ビット幅までの段間の増加を指定できます。出力ビット幅は、27 ビットまで増加してから、飽和します。出力ビット幅は各段で $\log_2(R)$ ビットずつ増加して、バタフライ演算を DSP にマップし続けます。このオプションは、最初の入力ビット幅が 27 ビット未満の場合に便利です。
- `SSR_FFT_SCALE`: パラメーター構造で `scaling_mode` 制約を `SSR_FFT_SCALE` に設定すると、各段の出力でスケーリングが有効になります。出力は、段ごとにスケーリングされるので、精度は落ちます。サイズ L および $\text{Radix} = \text{SSR} = R$ の FFT には $\log_2(L)$ 段があります。このオプションは入力ビット幅が既に 27 ビットに近く、乗算を DSP にマップするために出力が 27 ビットを超えないようにする必要がある場合に便利です。
- `SSR_FFT_NO_SCALE`: パラメーター構造で `scaling_mode` 制約を `SSR_FFT_NO_SCALE` に設定すると、各段ごとにビット増加が可能になるので、出力は各段ごとに $\log_2(R)$ で無制限に増加します。この設定は、高い精度が必要な場合に便利です。ただし、出力ビット幅が 27 ビットを超えて増加する場合、乗算が DSP にのみマップされるわけではなく、FPGA ファブリック ロジックを組み合わせて使用し始めることがあります。これにより、クロック速度とリソース使用率が落ちることがあります。

SSR FFT 固定小数点設定の使用に推奨されるフロー

SSR FFT では、複数のスケーリング モードがサポートされ、入力ビット幅および指数値 (ルックアップ テーブルの \sin/\cos) を格納するのに必要なビット幅を定義するオプションがあります。出力信号の質を定義する信号対ノイズ比は、これらのさまざまなパラメーターの選択、および実数値の連続信号または浮動小数点信号を固定小数点に変換するのに使用する量子化スキームによって異なります。FFT の出力の信号対ノイズ比を良くするには、信号の範囲および解像度 (基本的には整数ビットと分数ビット) を注意して選択する必要があります。SSR FFT HLS IP を使用する際は、次のフローを使用することをお勧めします。

SSR FFT の float モデルから開始

現在のところ、SSR FFT には `ap_fixed<>`、`float`、および `double` 型を使用できます。次の表は、合成およびシミュレーションのサポートのリストです。

表 29: SSR FFT 型のサポート

データ型	合成のサポート	シミュレーションのサポート
<code>std::complex < ap_fixed <> ></code>	あり	あり
<code>std::complex<float></code>	なし	あり
<code>std::complex<double></code>	なし	あり

まずは、`std::complex<>` で `float/double` 内部型から開始し、Matlab/Python/Octave/Simulink (ゴールデン テストベクターを生成するのに使用されるモデリング言語またはツールどれでも可) などの基準モデルと比較した SNR を検証します。SSR FFT の合成可能なバージョンは、現在のところ `ap_fixed<>` 内部型しかサポートしないので、次は固定小数点モデルを試すところから開始します。

固定小数点のモデリングおよびインプリメンテーション

固定小数点モデルから開始

前に固定小数点モデルを使用したことがある場合は、まず `SSR_FFT_NO_SCALING` スケーリング モードを使用することをお勧めします。入力ビット幅は、次のように選択する必要があります。

最初の固定小数点モデルを `ap_fixed<WL, IL>` 型で作成します。全体的な入力型は `std::complex <ap_fixed<WL, IL>` で、入力の実数部と仮数部が格納されます。

説明:

- `IL`: 整数ビット。入力範囲に基づいて選択されます。
- `WL`: ワード長 = `IL` + `FL`。 `FL` は分数ビット幅で、入力解像度に基づいて選択されます。

この場合、スケーリング モード選択ため、SSR FFT はどのスケーリングも使用しませんので、出力でスケーリング エラーが発生する可能性はありません。スケーリング モードをスケーリングなしに設定すると、入力サンプルを示すのに使用された整数ビットおよび分数ビットなどのその他の固定小数点パラメーターを試すことができます。単純なのは、入力範囲および解像度に基づいて入力を示すのに必要なビットを選択する方法ですが、その他の入力特性によっては、ユーザーがこれらのビット幅を最適化できます。

入力ビット幅の選択

入力幅の選択は、入力データ特性と必要な解像度によって異なり、基本的にテスト データの範囲と解像度によってデータに依存した選択になります。シミュレーション目的には、整数ビットと分数ビットを示すため、任意で多くのビット数を選択できます。インプリメンテーションでは、必要な SNR に留意して最適な選択をするようにしてください。

推奨されるストラテジは、次のとおりです。

- スケーリング モードは `SSR_FFT_NO_SCALING` に固定したままにします。
- SSR FFT の出力で信号対ノイズ比を確認して、整数および分数用の入力ビットを変更します。
- 出力 SNR 要件が最低限必要なビットの条件を満たすようにビット幅を削減します。

SNR 要件が満たされたら、複素指数関数テーブルおよび SSR FFT の出力スケーリング オプションを格納するのに必要なビットなど、その他の固定小数点演算に進むことができます。

回転因子またはサイン/コサイン ルックアップ テーブルの量子化

サイン/コサイン ルックアップ テーブル (回転因子/複素指数関数) の量子化に使用されるビット数は変更できます。推奨される設定は、合計 18 ビットで、分数に 2 ビットです。この設定にすると、乗算中に twiddle/sin/cos 入力をザイリンクス FPGA の DSP ブロックの 18 ビット入力にマップできます。このモデルは合成でき、その他のより大きなビット幅で動作できますが、乗算が 1 つの DSP ブロックにマップされず、複数の DSP ブロックまたは FPGA ファブリックを使用してインプリメントされるので、パフォーマンスが落ちることがあります。

回転因子幅の削減は、回転因子ストレージの初期設定が 18 ビットを超える場合に便利です。デフォルトでは、18 ビット (符号付き整数部に 2 ビットが予約) を使用するように設定されます。この 2 ビットは、ルックアップ テーブルに -1 値を正しく示すために必要です。

最適なスケーリング モードの選択

入力ビット幅を選択して、回転因子をスケーリングなしに設定して、固定小数点の SSR FFT の出力で許容範囲の SNR または二乗平均平方根 (RMS) エラーになるようになったら、ほかのスケーリング モードを試してみることができます。SSR FFT には、3 つのスケーリング モードを使用できます。まず、SSR_FFT_NO_SCALING ストラテジから開始することをお勧めします。出力で許容範囲の SNR/RMS エラーがある場合は、SSR_FFT_GROW_TO_MAX_WIDTH に切り替えます。それでも、許容範囲の SNR/RMS エラーがある場合は、SSR_FFT_SCALE に切り替えます。

SSR_FFT_NO_SCALING

これは、最初に試してみることをお勧めするモードです。スケーリングはなしになりますが、出力ビット幅は各段ごとに $\log_2(R=SSR)$ ずつ増加します。たとえば、FFT のサイズが $N=64$ で $SSR=R=4$ を選択した場合、SSR FFT は $\log_4(64) = 3$ 段になります。入力ビット幅が W の場合、出力ビット幅は $W+3*2=W+6$ になります。このため、出力は $\log_2(N)*\log_2(R)$ ビット増加します。

SSR_FFT_NO_SCALING を使用すると、計算の正確性は保持されますが、ハードウェア コストは最大になります。SSR FFT 計算は、1 つの段が次の段に送信されるような、チェーンの段で実行されます。

ビット増加を制御しないと、1 つの段の出力幅が制限を超えて増加し、乗算器が FPGA の DSP ブロックにマップしなくなってしまい、デザイン パフォーマンスが速度の面でかなり下がることがあります。たとえば、 $\log_2(N) * \log_2(R) + \text{Input Bit Width}(\text{IL}+\text{FL}) > \max(\text{DSP Block Multiplier Inputs})$ のデザインがあるとする、その他 2 つの使用可能なスケーリング スキームを試してみてください。DSP48 を含む FPGA デバイスの場合、 18×27 乗算器を含むザイリンクス DSP48 ブロックでは、条件は $\log_2(N) * \log_2(R) + \text{Input Bit Width} > 27$ になります。

SSR_FFT_GROW_TO_MAX_WIDTH

このモードの場合は、ハイブリッド方式を使用します。まず、増加する余裕がある場合は、ビット増加ができます。FFT 段の開始段階で出力ビット幅が DSP ブロックにマップされた幅よりも小さい場合は、ビット増加ができます。ビット幅が DSP ブロックにマップ可能な幅を超えると、出力のスケーリングが開始されます。

SSR_FFT_SCALE

指定した FFT サイズの N と SSR 係数で、出力が DSP 乗算器ブロックが指定の FPGA デバイスで処理できない制限を超えて増加することがわかっている場合、SSR_FFT_SCALE オプションを選択すると、各段でスケーリングを設定できます。このオプションでは、各段で出力を $\log_2(SSR=R)$ 分右シフトすることで、段ごとに出力がスケーリングされます。

固定小数点モデルを作成する際のガイドラインを提供し、SSR FFT で使用可能なオプションについて説明する場合にのみ推奨されるフローです。デザインの SNR/RMS 要件によって、指定したアプリケーションのさまざまなパフォーマンスおよび SNR/RMS 要件を確認しつつ、これらすべてのパラメーターを注意して選択する必要があります。

SSR FFT IP ライブラリの使用方法

SSR FFT は library `hls_ssr_lib.h` ライブラリを使用すると、C++ デザイン内で使用できます。このセクションはその使用例と、C++ ベースの HLS デザインで使用するためのその他のインターフェイス レベルの詳細について説明します。

SSR FFT IP ライブラリを使用する方法は次のとおりです。

1. `hls_ssr_lib.h` ヘッダーを含めます。

```
#include <hls_ssr_lib.h>
```

2. `ssr_fft_default_params` を拡張する C++ 構造体を定義します。

```
struct ssr_fft_params:ssr_fft_default_params
{
    static const int N=SSR_FFT_L;
    static const int R=SSR_FFT_R;
    static const scaling_mode_enum
        scaling_mode=SSR_FFT_GROW_TO_MAX_WIDTH;

    static const fft_output_order_enum
        output_data_order=SSR_FFT_NATURAL;
    static const int twiddle_table_word_length=18;
    static const int twiddle_table_intger_part_length=2;
};
```

3. SSR FFT を次のように呼び出します。

```
hls::ssr_fft::fft<ssr_fft_params>(inD,outD);
```

`inD` および `outD` は `ap_fixed`、`float`、または `double` 型の 2 次元の複素数配列です。合成およびシミュレーションの使用法については既に前の表で説明しています。I/O 配列は次のように宣言します。

- 固定小数点型: まず入力型を定義してから、データ型の特性を使用して `ssr_fft_params` 構造に基づいた出力型を計算します (出力型の計算ではビット増加と入力ビット幅に基づいたスケーリング モードが考慮されます)。

```
typedef std::complex< ap_fixed<16,8> > I_TYPE;
typedef
hls::ssr_fft::ssr_fft_output_type<ssr_fft_params,I_TYPE>:t_ssr_fft_out
O_TYPE;
I_TYPE inD[SSR_FFT_R][SSR_FFT_L/SSR_FFT_R];
O_TYPE outD [R][L/R];
```

この場合、`SSR_FFT_R` が SSR 係数を定義し、`SSR_FFT_L` が FFT 変換のサイズを定義します。

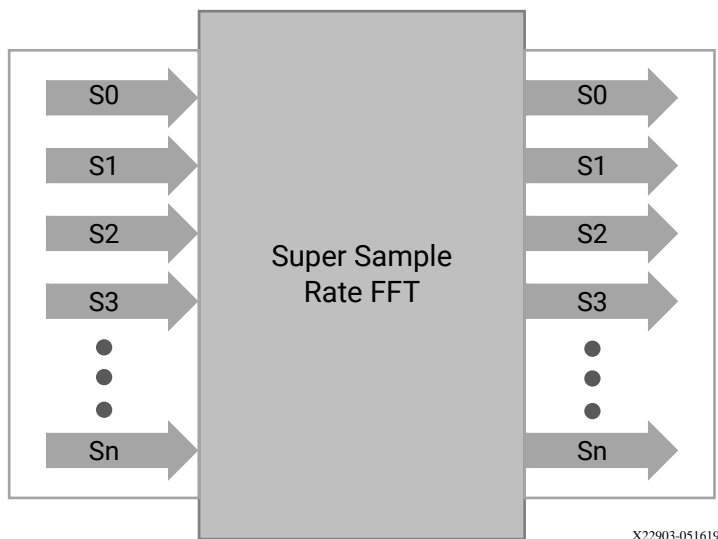
- `float` および `double` 型: まず `double/float` 入力型を定義してから、データ型の特性を使用して `ssr_fft_params` 構造に基づいた出力型を計算します。`float` 型の場合、出力型の計算は入力と同じデータ型を返します。

```
typedef std::complex< float/double > I_TYPE;
typedef
hls::ssr_fft::ssr_fft_output_type<ssr_fft_params,I_TYPE>:t_ssr_fft_out
O_TYPE;
I_TYPE inD[SSR_FFT_R][SSR_FFT_L/SSR_FFT_R];
O_TYPE outD[SSR_FFT_R][SSR_FFT_L/SSR_FFT_R];
```

SSR FFT の入力配列の読み出しおよび書き込みに関する注意点

SSR FFT HLS IP は合成後、次の図に示すように入力および出力の両方で、FIFO インターフェイスを使用したストリーミング ブロックにマップされます。

図 85: 合成後の SSR FFT HLS IP



X22903-051619

IP 記述内に含まれる HLS プラグマは、合成中に I/O 配列内の 2 次元を時間と幅のストリームにそれぞれマップします。2 つ目の次元には、HLS STREAM プラグマが使用されます。1 つ目の次元では、データ パッキング、パーティション、形状変更のプラグマを使用して、1 つの幅の広いストリームが作成されます。

たとえば、入力および出力配列が次のように宣言されるとします。

```
I_TYPE inD[R][L/R];
O_TYPE outD[R][L/R];
```

L/R サイズの次元が時間と次元 (R サイズが R 幅の 1 つのストリーム) にマップされます。このマップにより、SSR FFT を使用して C++ デザインを記述する際に、コンシューマーとプロデューサーでこれらの配列の読み出しおよび書き込み方法を指定する制約をいくつか設定できます。これらの制約は、配列次元から時間と並列の幅アクセスへの物理マッピングに基づきます。SSR FFT の I/O 配列の読み出しおよび書き込みは、次のように実行されます。

1. 入力は、1 つ目の次元にアクセスするループが内側のループになる入れ子のループ内に記述される必要があります。外側のループは、時間/2 つ目の次元にアクセスします。

```
for( int t=0; t<L/R; t++)
{
    for (int r=0; r < R : r++)
    {
        inD[r][t] = ..... ;
    }
}
```

- 出力は、同じ方法で読み出しにする必要があります。

```
for( int t=0; t<L/R; t++)
{
    for (int r=0; r <R : r++)
    {
        ... = outD[r][t] ;
    }
}
```

- SSR FFT IP が入力チェーンまたは出力チェーンの別の HLS IP に面している場合、読み出しおよび書き込みを実行する内側のループを展開する必要があります。

データフロー領域での SSR FFT の使用 - ストリーミングおよびノンストリーミング接続

SSR FFT は、HLS データフロー最適化にかなり依存します。SSR FFT は、FFT 入力や出力に次の 2 つの方法で接続できます。

- ストリーミング接続
- ノンストリーミング接続

ストリーミング接続

入力にストリーミング接続する場合は、次のようなコードになります。

```
#pragma HLS DATAFLOW

in_dummy_proc (... , fft_in);
hls:ssr_fft::fft<ssr_fft_params>(fft_in, fft_out)
out_dummy_proc(fft_out, ...)
...
...
...
```

入力プロデューサーの制約は幅の広いストリームを生成するようにし、出力コンシューマーの制約は幅の広いストリームを取り入れるようにします。これらの制約については、前のセクションでも説明しました。

ノンストリーミング接続

現在の SSR FFT のバージョンでは、出力および入力でのノンストリーミング接続がサポートされませんが、必要に応じて入力/出力にアダプターを配置すると、ストリームを別のインターフェイスに変換できます。たとえば、次のコードはストリーミング インターフェイスをメモリ ベースのインターフェイスにマップする入力アダプターを記述しています。

```
template < type name TYPE, int R, int L >
void fft_input_adapter (TYPE inData[R][L/R], TYPE outDataStream[R][L/R])
{
    #pragma HLS INLINE off
    #pragma HLS DATA_PACK variable=inData
    #pragma HLS ARRAY_RESHAPE variable=inData complete dim=1
    for(int t=0; t<L/R; t++)
    {
        #pragma HLS PIPELINE II=1
        for (int r = 0; r< R; ++r)
        {
            outDataStream [r][t] = inData[r][t];
        }
    }
}
```

```

}

.
.
. // Usage of Adapter at input side:

#pragma HLS DATAFLOW
    in_proc_memory_based(..., in_data_mem_based)
    fft_input_adapter<TYPE_NAME, R, L>( in_data_mem_based,
    fft_in_stream_based);
    hls:ssr_fft::fft<ssr_fft_params>(fft_in_stream_based,
    fft_out_stream_based)
    out_dummy_proc(fft_out_stream_based, ....)
...
...
...

```

注記: 出力側のアダプターも同様の方法で構築できます。

FIR フィルター IP ライブラリ

ザイリンクス FIR ブロックは、hls_fir.h ライブラリを使用すると C++ デザイン内で呼び出すことができます。このセクションでは、FIR を C++ コードで設定する方法を説明します。



推奨: ザイリンクスでは、この IP の機能のインプリメント方法および機能の使用方法について、『FIR Compiler LogiCORE IP 製品ガイド』(PG149)を確認することをお勧めしています。

C++ コードで FIR を使用するには、次の手順に従います。

1. コードに hls_fir.h ライブラリを含めます。
2. 定義済み構造体 hls::ip_fir::params_t を使用してスタティック パラメーターを設定します。
3. FIR 関数を呼び出します。
4. ランタイム入力コンフィギュレーションを定義してパラメーターの一部を動的に変更します (オプション)。

次のコード例に、これらの各手順の実行方法を示します。各手順の詳細は、次のとおりです。

まず、ソース コードに FIR ライブラリを含めます。このヘッダー ファイルは、Vivado HLS インストール ディレクトリの include ディレクトリにあります。このファイルは、Vivado HLS を実行すると自動的に検索されます。Vivado HLS 内でコンパイルする場合は、このディレクトリへのパスを指定する必要はありません。

```
#include "hls_fir.h"
```

FIR のスタティック パラメーターを定義します。これには、入力幅、係数、フィルター レート (single、decimation、hilbert) などのスタティック属性が含まれます。FIR ライブラリにはパラメーター指定構造体 hls::ip_fir::params_t が含まれ、すべてのスタティック パラメーターをデフォルト値で初期化できます。

次の例では、係数を coeff_vec 配列として定義し、定義済み構造体に基づくユーザー定義構造体 myconfig を使用して、係数値、入力幅、および量子化モードのデフォルト値を変更しています。

```

struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
    static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
    static const unsigned input_width = INPUT_WIDTH;
    static const unsigned quantization = hls::ip_fir::quantize_only;
};

```

HLS 名前空間と定義済みスタティック パラメーター (この例では `myconfig`) を使用して FIR 関数のインスタンスを作成し、`run` メソッドで関数を呼び出して関数を実行します。関数引数は、順に入力データ、出力データです。

```
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out);
```

オプションで、ランタイム入力コンフィギュレーションを使用できます。FIR のモードには、インターリーブされたチャンネルで係数がどのように使用されるか、または係数の再読み込みがいつ必要かが、この入力のデータによって決定されるものがあります。これは動的に設定できるので、変数として定義されます。この入力コンフィギュレーションが必要なモードの詳細は、『FIR Compiler LogiCORE IP 製品ガイド』 (PG149) を参照してください。

ランタイム入力コンフィギュレーションを使用する場合、FIR 関数を入力データ、出力データ、および入力コンフィギュレーションの3つの引数を使用して呼び出します。

```
// Define the configuration type
typedef ap_uint<8> config_t;
// Define the configuration variable
config_t fir_config = 8;
// Use the configuration in the FFT
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out, &fir_config);
```

FIR C ライブラリを使用したデザイン例は、[Help] → [Welcome] → [Open Example Project] → [Design Examples] → [FIR] をクリックして表示される Vivado HLS のサンプルに含まれます。

FIR のスタティック パラメーター

FIR のスタティック パラメーターは、入力幅と出力幅、小数ビット数、係数値、補間レートおよび間引きレートなどの変化しない値を指定します。これらのほとんどにデフォルト値がありますが、係数にはデフォルト値はありません。

`hls_fir.h` ヘッダー ファイルでは、スタティック パラメーターのデフォルト値を設定するのに使用可能な `struct hls::ip_fir::params_t` が定義されます。



重要: 係数にはデフォルト値は定義されないため、ザイリンクスでは、FIR を直接初期化するために定義済みの構造体を使用することをお勧めしていません。スタティック パラメーターを指定するには、係数を指定する新しいユーザー定義構造体を使用する必要があります。

次の例では、新しい係数値を指定する新しいユーザーの `struct my_config` を定義しています。係数は配列 `coeff_vec` として指定されています。FIR のその他すべてのパラメーターにはデフォルト値が使用されます。

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srcc_coeffs_len];
};
static hls::FIR<myconfig> fir1;
fir1.run(fir_in, fir_out);
```

FIR のスタティック パラメーターにパラメーター指定構造体 `hls::ip_fir::params_t` に使用されるパラメーターを示します。パラメーターのデフォルト値および使用可能な値のリストは、FIR の構造体パラメーターの値を参照してください。



推奨: ザイリンクスでは、パラメーターの詳細およびその設定の影響について、『FIR Compiler LogiCORE IP 製品ガイド』 (PG149) を参照することをお勧めします。

FIR の構造体パラメーター

表 30: FIR の構造体パラメーター

パラメーター	説明
input_width	データ入力ポート幅
input_fractional_bits	入力ポートの小数ビットの数
output_width	データ出力ポート幅
output_fractional_bits	出力ポートの小数ビットの数
coeff_width	係数のビット幅
coeff_fractional_bits	係数の小数ビットの数
num_coefs	係数の数
coeff_sets	係数セットの数
input_length	入力データのサンプル数
output_length	出力データのサンプル数
num_channels	処理されるデータのチャンネル数を指定
total_num_coeff	係数の総数
coeff_vec[total_num_coeff]	係数配列
filter_type	フィルタに使用されるフィルタ タイプをインプリメント
rate_change	整数または小数レートの変更を指定
interp_rate	補間レート
decim_rate	間引きレート
zero_pack_factor	補間で使用される係数 0 の数
rate_specification	レートを周波数または周期として指定
hardware_oversampling_rate	オーバーサンプリングのレートを指定
sample_period	ハードウェアのオーバーサンプリング周期
sample_frequency	ハードウェアのオーバーサンプリング周波数
quantization	使用される量子化方法
best_precision	最適な精度をイネーブルまたはディスエーブル
coeff_structure	使用する係数構造のタイプ
output_rounding_mode	出力で使用する丸めのタイプ
filter_arch	シストリックまたは転置型アーキテクチャを選択
optimization_goal	最適化目標を速度またはエリアに指定
inter_column_pipe_length	DSP 列間に必要なパイプライン長
column_config	DSP48 列数を指定
config_method	DSP48 列のコンフィギュレーション方法を指定
coeff_padding	フィルタの先頭に追加される 0 パディングの数

整数またはブール型ではないパラメーター値を指定する際は、HLS FIR 名前空間を使用する必要があります。

たとえば、`rate_change` に使用可能な値は、次の表で `integer` および `fixed_fractional` と示されており、C プログラムでは `rate_change = hls::ip_fir::integer` および `rate_change = hls::ip_fir::fixed_fractional` を使用する必要があります。

FIR の構造体パラメーターの値

次の表に、FIR IP のすべての機能を示します。この表に示されていない機能は、Vivado HLS のインプリメンテーションではサポートされていません。

表 31: FIR の構造体パラメーターの値

パラメーター	C データ型	デフォルト値	有効な値
input_width	unsigned	16	制限なし
input_fractional_bits	unsigned	0	input_width のサイズにより制限
output_width	unsigned	24	制限なし
output_fractional_bits	unsigned	0	output_width のサイズにより制限
coeff_width	unsigned	16	制限なし
coeff_fractional_bits	unsigned	0	coeff_width のサイズにより制限
num_coeffs	bool	21	フル
coeff_sets	unsigned	1	1 ~ 1024
input_length	unsigned	21	制限なし
output_length	unsigned	21	制限なし
num_channels	unsigned	1	1 ~ 1024
total_num_coeff	unsigned	21	num_coeffs * coeff_sets
coeff_vec[total_num_coeff]	double array	なし	該当なし
filter_type	unsigned	single_rate	single_rate、interpolation、decimation、hilbert_filter、interpolated
rate_change	unsigned	integer	integer、fixed_fractional
interp_rate	unsigned	1	1 ~ 1024
decim_rate	unsigned	1	1 ~ 1024
zero_pack_factor	unsigned	1	1-8
rate_specification	unsigned	period	frequency、period
hardware_oversampling_rate	unsigned	1	制限なし
sample_period	bool	1	制限なし
sample_frequency	unsigned	0.001	制限なし
quantization	unsigned	integer_coefficients	integer_coefficients、quantize_only、maximize_dynamic_range
best_precision	unsigned	false	false true
coeff_structure	unsigned	non_symmetric	inferred、non_symmetric、symmetric、negative_symmetric、half_band、hilbert

表 31: FIR の構造体パラメーターの値 (続き)

パラメーター	C データ型	デフォルト値	有効な値
output_rounding_mode	unsigned	full_precision	full_precision、truncate_lsbs、non_symmetric_rounding_down、non_symmetric_rounding_up、symmetric_rounding_to_zero、symmetric_rounding_to_infinity、convergent_rounding_to_even、convergent_rounding_to_odd
filter_arch	unsigned	systolic_multiply_accumulate	systolic_multiply_accumulate、transpose_multiply_accumulate
optimization_goal	unsigned	area	area、speed
inter_column_pipe_length	unsigned	4	1-16
column_config	unsigned	1	使用される DSP48 の数によって制限
config_method	unsigned	single	single、by_channel
coeff_padding	bool	false	false true

FIR 関数の使用

FIR 関数は、HLS 名前空間で定義され、次のように呼び出すことができます。

```
// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;
// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY, OUTPUT_DATA_ARRAY);
```

STATIC_PARAM は、FIR の最もスタティックなパラメーターを定義するスタティック パラメーター指定構造体で、

入力データおよび出力データは、配列 (INPUT_DATA_ARRAY および OUTPUT_DATA_ARRAY) として関数に供給されます。最終的なインプリメンテーションでは、FIR IP のこれらのポートは AXI4-Stream ポートとしてインプリメントされます。データフロー最適化 (set_directive_dataflow) を使用した領域では、配列がストリーミング配列としてインプリメントされるので、サイリンクスでは常に FFT 関数を使用することを勧めします。または、両方の配列を set_directive_stream コマンドを使用してストリーミングとして指定します。



重要: FIR は、パイプラインされた領域では使用できません。高パフォーマンスの演算が必要な場合、FIR の前後でループまたは関数をパイプライン処理し、その領域のすべてのループおよび関数でデータフロー最適化を使用します。

FIR のマルチチャネル機能は、シングル入力およびシングル出力配列のデータをインターリーブすることでサポートされます。

- 入力配列のサイズは、すべてのサンプルに十分な大きさである必要があります (num_channels * input_length)。

- 出力配列のサイズは、すべての出力サンプルを含めるように指定する必要があります (num_channels * output_length)。

次のコード例は、2チャンネルで、データがどのようにインターリーブされるかを示しています。最上位関数に入力データのチャンネルが2つ (din_i、din_q) および出力データのチャンネルが2つ (dout_i、dout_q) 含まれています。フロントエンド (fe) とバックエンド (be) の2つの関数を使用して、FIR入力配列のデータを正しく並べ、FIR出力配列からそれを抽出しています。

```
void dummy_fe(din_t din_i[LENGTH], din_t din_q[LENGTH], din_t
out[FIR_LENGTH]) {
    for (unsigned i = 0; i < LENGTH; ++i) {
        out[2*i] = din_i[i];
        out[2*i + 1] = din_q[i];
    }
}
void dummy_be(dout_t in[FIR_LENGTH], dout_t dout_i[LENGTH], dout_t
dout_q[LENGTH]) {
    for(unsigned i = 0; i < LENGTH; ++i) {
        dout_i[i] = in[2*i];
        dout_q[i] = in[2*i+1];
    }
}
void fir_top(din_t din_i[LENGTH], din_t din_q[LENGTH],
dout_t dout_i[LENGTH], dout_t dout_q[LENGTH]) {

    din_t fir_in[FIR_LENGTH];
    dout_t fir_out[FIR_LENGTH];
    static hls::FIR<myconfig> fir1;

    dummy_fe(din_i, din_q, fir_in);
    fir1.run(fir_in, fir_out);
    dummy_be(fir_out, dout_i, dout_q);
}
```

オプションの FIR ランタイム コンフィギュレーション

演算モードによっては、係数の使用方法を設定するため、FIR に別の入力が必要なこともあります。この入力コンフィギュレーションが必要なモードの詳細は、『FIR Compiler LogiCORE IP 製品ガイド』(PG149) を参照してください。

この入力コンフィギュレーションは、標準 8 ビット データ型の ap_int.h を使用して C コードで実行できます。この例では、fir_top.h ヘッダー ファイルで FIR と ap_fixed ライブラリの使用が指定され、多くのデザイン パラメータ値が定義された後、それらに基づいて固定小数点型がいくつか定義されています。

```
#include "ap_fixed.h"
#include "hls_fir.h"

const unsigned FIR_LENGTH = 21;
const unsigned INPUT_WIDTH = 16;
const unsigned INPUT_FRACTIONAL_BITS = 0;
const unsigned OUTPUT_WIDTH = 24;
const unsigned OUTPUT_FRACTIONAL_BITS = 0;
const unsigned COEFF_WIDTH = 16;
const unsigned COEFF_FRACTIONAL_BITS = 0;
const unsigned COEFF_NUM = 7;
const unsigned COEFF_SETS = 3;
const unsigned INPUT_LENGTH = FIR_LENGTH;
const unsigned OUTPUT_LENGTH = FIR_LENGTH;
```

```
const unsigned CHAN_NUM = 1;
typedef ap_fixed<INPUT_WIDTH, INPUT_WIDTH - INPUT_FRACTIONAL_BITS> s_data_t;
typedef ap_fixed<OUTPUT_WIDTH, OUTPUT_WIDTH - OUTPUT_FRACTIONAL_BITS>
m_data_t;
typedef ap_uint<8> config_t;
```

最上位コードには、ヘッダー ファイルの情報が含まれ、ビット幅を指定するのに使用したのと同じ定数値を使用してスタティックでパラメーター指定された構造体を作成されるので、C コードと FIR コンフィギュレーションが一致するようになり、係数が指定されます。最上位では、ヘッダー ファイルで 8 ビット データとして定義された入力コンフィギュレーションが FIR に渡されます。

```
#include "fir_top.h"

struct param1 : hls::ip_fir::params_t {
    static const double coeff_vec[total_num_coeff];
    static const unsigned input_length = INPUT_LENGTH;
    static const unsigned output_length = OUTPUT_LENGTH;
    static const unsigned num_coeffs = COEFF_NUM;
    static const unsigned coeff_sets = COEFF_SETS;
};
const double param1::coeff_vec[total_num_coeff] =
    {6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6};

void dummy_fe(s_data_t in[INPUT_LENGTH], s_data_t out[INPUT_LENGTH],
              config_t* config_in, config_t* config_out)
{
    *config_out = *config_in;
    for(unsigned i = 0; i < INPUT_LENGTH; ++i)
        out[i] = in[i];
}

void dummy_be(m_data_t in[OUTPUT_LENGTH], m_data_t out[OUTPUT_LENGTH])
{
    for(unsigned i = 0; i < OUTPUT_LENGTH; ++i)
        out[i] = in[i];
}

// DUT
void fir_top(s_data_t in[INPUT_LENGTH],
             m_data_t out[OUTPUT_LENGTH],
             config_t* config)
{
    s_data_t fir_in[INPUT_LENGTH];
    m_data_t fir_out[OUTPUT_LENGTH];
    config_t fir_config;
    // Create struct for config
    static hls::FIR<param1> fir1;

    //=====
    // Dataflow process
    dummy_fe(in, fir_in, config, &fir_config);
    fir1.run(fir_in, fir_out, &fir_config);
    dummy_be(fir_out, out);
    //=====
}
```

FIR C ライブラリを使用したデザイン例は、[Help]→[Welcome]→[Open Example Project]→[Design Examples]→[FIR] をクリックして表示される Vivado HLS のサンプルに含まれます。

DDS IP ライブラリ

hls_dds.h ライブラリを使用し、C++ デザイン内でザイリンクス DDS (Direct Digital Synthesizer) IP ブロックを使用できます。このセクションでは、DDS IP を C++ コードで設定する方法を説明します。



推奨: ザイリンクスでは、この IP の機能のインプリメント方法および機能の使用方法について、『DDS Compiler LogiCORE IP 製品ガイド』(PG141) を参照することをお勧めしています。



重要: DDS IP コアの C IP インプリメンテーションでは、Phase_Increment および Phase_Offset パラメーターに fixed モードがサポートされ、Phase_Offset には none モードもサポートされますが、programmable および streaming モードはサポートされません。

C++ コードで DDS を使用するには、次の手順に従います。

1. コードに hls_dds.h ライブラリを含めます。
2. 定義済み構造体 hls::ip_dds::params_t を使用してデフォルト パラメーターを設定します。
3. DDS 関数を呼び出します。

まず、ソース コードに DDS ライブラリを含めます。このヘッダー ファイルは、Vivado HLS のインストール ディレクトリの include ディレクトリに含まれています。このディレクトリは、Vivado HLS が実行されると自動的に検索されます。

```
#include "hls_dds.h"
```

DDS のスタティク パラメーターを定義します。たとえば、位相幅、クロック レート、位相およびインクリメントのオフセットを定義します。DDS C ライブラリにはパラメーター指定構造体 hls::ip_dds::params_t が含まれ、すべてのスタティク パラメーターをデフォルト値で初期化できます。この構造体の値のいずれかを再定義することで、インプリメンテーションをカスタマイズできます。

次の例では、既存の定義済み構造体 hls::ip_dds::params_t に基づくユーザー定義の構造体 param1 を使用して、位相幅、クロック レート、位相オフセット、チャンネル数のデフォルト値を変更する方法を示しています。

```
struct param1 : hls::ip_dds::params_t {
    static const unsigned Phase_Width = PHASEWIDTH;
    static const double DDS_Clock_Rate = 25.0;
    static const double PINC[16];
    static const double POFF[16];
};
```

HLS 名前空間と定義済みのスタティク パラメーターを使用して、DDS 関数のインスタンス (param1 など) を作成します。その後、run メソッドを使用して関数呼び出して実行します。次の例では、データおよび位相関数引数が順に示されています。

```
static hls::DDS<config1> dds1;
dds1.run(data_channel, phase_channel);
```

DDS の C ライブラリを使用するサンプル デザインにアクセスするには、[Help]→[Welcome]→[Open Example Project]→[Design Examples]→[DDS] をクリックします。

DDS のスタティック パラメーター

DDS のスタティック パラメーターは、クロック レート、位相間隔、モードなど、DDS を設定する方法を定義します。hls_dds.h ヘッダー ファイルは、スタティック パラメーターのデフォルト値を設定する hls::ip_dds::params_t 構造体を定義します。デフォルト値を使用するには、パラメーター設定構造体を DDS 関数で直接使用できます。

```
static hls::DDS< hls::ip_dds::params_t > dds1;
dds1.run(data_channel, phase_channel);
```

次の表に、パラメーター指定構造体 hls::ip_dds::params_t のパラメーターを示します。



推奨: ザイリンクスでは、パラメーターおよび値の詳細について、『DDS Compiler LogiCORE IP 製品ガイド』(PG141) を参照することをお勧めします。

表 32: DDS の構造体パラメーター

パラメーター	説明
DDS_Clock_Rate	DDS 出力のクロック レートを指定します。
Channels	チャンネル数を指定します。DDS および位相ジェネレーターは最大 16 チャンネルまでサポート可能です。チャンネルは時分割されるので、チャンネルごとの実効クロック周波数を低減します。
Mode_of_Operation	次の操作モードの 1 つを指定します。 標準モード: SIN/COS LUT へのアクセスに使用する前に累算された位相を切り捨てることのできる場合に使用します。 ラスタライズ モード: 希望の周波数とシステム クロックが有理分数で関連している場合に使用します。
Modulus	システム クロック周波数と希望の周波数との関係を定義します。 ラスタライズ モードでのみ使用します。
Spurious_Free_Dynamic_Range	DDS で生成されるトーンのターゲット純度を指定します。
Frequency_Resolution	Hz で最小周波数分解能を指定し、また、関連付けられている位相インクリメント (PINC) および位相オフセット (POFF) 値を含む、位相アキュムレータで使用する位相幅を決定します。
Noise_Shaping	位相切り捨て、ディザリング、またはテイラー級数訂正を使用するかどうかを決定します。
Phase_Width	次の幅を設定します。 m_axis_phase_tdata 内の PHASE_OUT フィールド DDS が SIN/COS LUT のみに設定される場合は、s_axis_phase_tdata 内の位相フィールド 位相アキュムレータ 関連付けられている位相インクリメントおよびオフセットレジスタ s_axis_config_tdata の位相フィールド ラスタライズ モードの場合、位相幅は有効な入力範囲 [0, Modulus-1]、つまり log2 (Modulus-1) を切り上げた値を記述するのに必要なビット数として固定されています。
Output_Width	m_axis_data_tdata 内の SINE および COSINE フィールドの幅を設定します。このパラメーターで提供される SFDR は、選択されている Noise Shaping オプションによって異なります。
Phase_Increment	位相インクリメント値を選択します。

表 32: DDS の構造体パラメーター (続き)

パラメーター	説明
Phase_Offset	位相オフセット値を選択します。
Output_Selection	m_axis_data_tdata バスの SINE、COSINE、またはその両方への出力を選択します。
Negative_Sine	ランタイム時に SINE フィールドをネゲートします。
Negative_Cosine	ランタイム時に COSINE フィールドをネゲートします。
Amplitude_Mode	振幅をフル レンジまたは単位円に設定します。
Memory_Type	SIN/COS LUT のインプリメンテーションを制御します。
Optimization_Goal	インプリメンテーションで最高速を狙うか、リソース使用率をできる限り抑えるかを決定します。
DSP48_Use	位相アキュムレータおよび位相オフセット、ディザ ノイズ追加、またはその両方をインプリメントします。
Latency_Configuration	最適化目標に基づいた最適値にコアのレイテンシを設定します。
Latency	レイテンシ値を指定します。
Output_Form	出力形式を 2 の補数または符号絶対値に設定します。一般的には SINE および COSINE の出力形式は 2 の補数です。ただし、象限対称が使用されている場合は、出力形式を符号絶対値に変えることができます。
PINC[XIP_DDS_CHANNELS_MAX]	各出力チャンネルの位相インクリメントの値を設定します。
POFF[XIP_DDS_CHANNELS_MAX]	各出力チャンネルの位相オフセットの値を設定します。

DDS の構造体パラメーター値

次の表に、パラメーター設定構造体 `hls::ip_dds::params_t` のパラメーター値を示します。

表 33: DDS の構造体パラメーター値

パラメーター	C データ型	デフォルト値	有効値
DDS_Clock_Rate	double	20.0	任意の倍精度値
Channels	unsigned	1	1 ~ 16
Mode_of_Operation	unsigned	XIP_DDS_MOO_CONVENTIONAL	XIP_DDS_MOO_CONVENTIONAL は累計位相を切り捨てます。 XIP_DDS_MOO_RASTERIZED はラスタライズ モードを選択します。
Modulus	unsigned	200	129 ~ 256
Spurious_Free_Dynamic_Range	double	20.0	18.0 ~ 150.0
Frequency_Resolution	double	10.0	0.000000001 ~ 125000000

表 33: DDS の構造体パラメーター値 (続き)

パラメーター	C データ型	デフォルト値	有効値
Noise_Shaping	unsigned	XIP_DDS_NS_NONE	XIP_DDS_NS_NONE は位相切り捨て DDS を生成します。 XIP_DDS_NS_DITHER は、SFDR を改善するために位相ディザイザーを使用しますが、ノイズレベルが増加します。 XIP_DDS_NS_TAYLOR は、位相切り捨てにより破棄されるビットを使用してサイン/コサイン値を補間します。 XIP_DDS_NS_AUTO は自動的にノイズシェーピングを決定します。
Phase_Width	unsigned	16	8 の整数倍数である必要があります。
Output_Width	unsigned	16	8 の整数倍数である必要があります。
Phase_Increment	unsigned	XIP_DDS_PINCPOFF_FIXED	XIP_DDS_PINCPOFF_FIXED は、生成時に PINC を固定します。PINC をランタイム時に変更することはできません。 この値しかサポートされていません。
Phase_Offset	unsigned	XIP_DDS_PINCPOFF_NONE	XIP_DDS_PINCPOFF_NONE は位相オフセットを生成しません。 XIP_DDS_PINCPOFF_FIXED は、生成時に POFF を固定します。POFF をランタイム時に変更することはできません。
Output_Selection	unsigned	XIP_DDS_OUT_SIN_AND_COS	XIP_DDS_OUT_SIN_ONLY はサイン出力のみを生成します。 XIP_DDS_OUT_COS_ONLY はコサイン出力のみを生成します。 XIP_DDS_OUT_SIN_AND_COS はサインおよびコサイン出力の両方を出力します。
Negative_Sine	unsigned	XIP_DDS_ABSENT	XIP_DDS_ABSENT は標準正弦波を生成します。 XIP_DDS_PRESENT は正弦波をネゲートします。
Negative_Cosine	bool	XIP_DDS_ABSENT	XIP_DDS_ABSENT は標準正弦波を生成します。 XIP_DDS_PRESENT は正弦波をネゲートします。

表 33: DDS の構造体パラメーター値 (続き)

パラメーター	C データ型	デフォルト値	有効値
Amplitude_Mode	unsigned	XIP_DDS_FULL_RANGE	XIP_DDS_FULL_RANGE はまず 2 進小数点の出力幅に振幅を正規化します。たとえば、8 ビット出力のバイナリ振幅は 100000000 - 10 で、値は 01111110 ~ 11111110 です。これは、わずかに 1 に満たない値から -1 よりもわずかに大きい値までに対応します。 XIP_DDS_UNIT_CIRCLE はハーフ フル レンジに振幅を正規化します。つまり 01000 . (+0.5) から 110000 .. (-0.5) までの値になります。
Memory_Type	unsigned	XIP_DDS_MEM_AUTO	XIP_DDS_MEM_AUTO は、テーブルが 1 層のメモリに含められる場合に分散 ROM を選択し、そうでない場合はブロック RAM を選択します。 XIP_DDS_MEM_BLOCK は常にブロック RAM を使用します。 XIP_DDS_MEM_DIST は常に分散 RAM を選択します。
Optimization_Goal	unsigned	XIP_DDS_OPTGOAL_AUTO	XIP_DDS_OPTGOAL_AUTO は自動的に最適化目標を選択します。 XIP_DDS_OPTGOAL_AREA はエリアを目標に最適化を実行します。 XIP_DDS_OPTGOAL_SPEED はパフォーマンスを目標に最適化を実行します。
DSP48_Use	unsigned	XIP_DDS_DSP_MIN	XIP_DDS_DSP_MIN は、位相アキュムレータおよび位相オフセット、ディザ ノイズ追加、またはその両方を FPGA ロジックにインプリメントします。 XIP_DDS_DSP_MAX は、位相アキュムレータおよび位相オフセット、ディザ ノイズ追加、またはその両方を DSP スライスを使用してインプリメントします。シングル チャネルの場合は、プログラマブルな位相インクリメント、位相オフセット、またはその両方を格納するレジスタを DSP スライスは提供することもできるため、ファブリック リソースを節約できます。
Latency_Configuration	unsigned	XIP_DDS_LATENCY_AUTO	XIP_DDS_LATENCY_AUTO は自動的にレイテンシを決定します。 XIP_DDS_LATENCY_MANUAL は、レイテンシ オプションを使用してレイテンシを手動で指定します。
Latency	unsigned	5	任意の値

表 33: DDS の構造体パラメーター値 (続き)

パラメーター	C データ型	デフォルト値	有効値
Output_Form	unsigned	XIP_DDS_OUTPUT_TWOS	XIP_DDS_OUTPUT_TWOS は 2 の補数を出力します。 XIP_DDS_OUTPUT_SIGN_MAG は符号絶対値を出力します。
PINC[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	各出力チャンネルの位相インクリメントの値
POFF[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	各出力チャンネルの位相オフセットの値

SRL IP ライブラリ

C コードは、再利用、読みやすさ、パフォーマンスなどの複数の要件を満たすように記述されます。現時点では、C コードは高位合成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。

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が追加され、既存のデータが配列内でシフトされ、最も古いデータ値が破棄されます。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ハードウェアにインプリメントする場合、配列を個々の要素に完全に分割し、RTL 内の要素間のデータ依存性からシフトレジスタを暗示させるのが最も一般的です。

論理合成では、通常 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常にインプリメントされるようになっています。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

読み出しメソッドでは、シフトレジスタの指定した位置から読み出しできます。

Vivado HLS には、`ap_shift_reg` クラスを定義する `ap_shift_reg.h` ヘッダー ファイルがスタンドアロン パッケージとして含まれており、ユーザーのソース コードで使用できるようになっています。
`xilinx_hls_lib_<release_number>.tgz` パッケージは、Vivado HLS インストール ディレクトリの `include` ディレクトリに含まれます。

```
// 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

`shift` メソッドでは、読み出し、書き込みおよびシフト演算を実行できます。

```
// 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、シフト演算を変数で制御およびイネーブルにできます。

```
// 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。

HLS 線形代数ライブラリ

HLS 線形代数ライブラリには、よく使用される C++ 線形代数関数が多く含まれます。HLS 線形代数ライブラリの関数では、次の表にリストするように 2 次元配列を使用して行列が表されます。

表 34: HLS 線形代数ライブラリ

関数	データ型	インプリメンテーション形式
cholesky	float ap_fixed x_complex<float> x_complex<ap_fixed>	合成済み
cholesky_inverse	float ap_fixed x_complex<float> x_complex<ap_fixed>	合成済み
matrix_multiply	float ap_fixed x_complex<float> x_complex<ap_fixed>	合成済み
qrf	float x_complex<float>	合成済み
qr_inverse	float x_complex<float>	合成済み
svd	float x_complex<float>	合成済み

線形代数関数では、すべて 2 次元配列を使用して行列が示されます。すべての関数で、実数および複素数データに対して浮動小数点 (単精度) 入力がサポートされます。関数のサブセットでは、実数および複素数データに対して ap_fixed (固定小数点) 入力がサポートされます。ap_fixed データ型の精度および丸めビヘイビアは、必要であればユーザーが定義できます。

線形代数ライブラリの使用

HLS の線形代数関数は次のいずれかの方法で参照できます。

- スコープ付き命名規則は、次のように使用します。

```
#include "hls_linear_algebra.h"

hls::cholesky(In_Array, Out_Array);
```

- hls 名前空間は次のように使用します。

```
#include "hls_linear_algebra.h"
using namespace hls; // Namespace specified after the header files

cholesky(In_Array, Out_Array);
```

線形代数関数の最適化

線形代数関数を使用する際は、RTL インプリメンテーションの最適化レベルを指定する必要があります。最適化のレベルとタイプは、C コードがどのように記述されているかと、Vivado HLS 指示子が C コードにどのように適用されるかによって異なります。

Vivado HLS では最適化のプロセスを単純化するために線形代数ライブラリ関数が提供されており、これらのライブラリ関数には複数のコード アーキテクチャと埋め込み型最適化指示子が含まれます。C++ コンフィギュレーション クラスを使用すると、使用する C コードと適用する最適化指示子を選択できます。

実際に実行される最適化は関数によって異なりますが、コンフィギュレーション クラスにより、RTL インプリメンテーションの最適化レベルを次のように指定できます。

- Small: リソースとスループットが小さくなります。
- Balanced: リソースとスループットのバランスが取られます。
- Fast: リソースは多くなりますが、スループットは大きくなります。

Vivado HLS には、線形代数ライブラリの各関数のコンフィギュレーション クラスをどのように使用するか示したサンプル プロジェクトが含まれています。これらのサンプルをテンプレートとして使用し、特定のインプリメンテーション ターゲットの関数用に Vivado HLS を設定する方法を学んでください。各サンプルには、複数の C コード アーキテクチャを含む C++ ソース ファイルが別の C++ 関数として含まれます。

注記: 最上位 C++ 関数を識別するには、`directives.tcl` ファイルが Vivado HLS GUI の [Directive] で TOP 指示子を探してください。

これらのサンプルは、Vivado HLS の Welcome 画面から開くことができます。

1. [Open Example Project] をクリックします。
2. [Examples] ダイアログ ボックスの [Design Examples] → [linear_algebra] → [implementation_targets] をクリックします。

注記: [Welcome] ページは、Vivado HLS の GUI を起動すると表示されます。[Help] → [Welcome] をクリックすると、いつでもこのページにアクセスできます。

どの最適化がデザインに最適なのかを決めるには、Vivado HLS の [Compare Reports] を使用してソリューションごとのパフォーマンスおよび使用量の見積もりを比較します。見積もりを比較するには、[Solution] → [Run C Synthesis] → [All Solutions] をクリックし、すべてのプロジェクト ソリューションに対して合成を実行する必要があります。

コレスキー

インプリメンテーションの制御

次の表に、リソース使用量、関数のスループット (開始間隔)、および関数のレイテンシに影響する主な要因を示します。小、中、大は、要因間で相対的なものです。

表 35: Cholesky の主な要因のまとめ

主な要因	値	リソース	スループット	レイテンシ
アーキテクチャ (ARCH)	0	小	小	大
	1	中	中	中
	2	大	大	小
内部ループのパイプライン処理 (INNER_II)	1	大	大	小
	>1	小	小	大
内部ループ展開 (UNROLL_FACTOR)	1	小	小	大
	>1	大	大	小

主な機能

- アーキテクチャ
 - 0: 最小の DSP 使用量と最小のスループットを使用します。
 - 1: DSP 使用量は多くなりますが、スループットは増加してメモリ使用量は最小限になります。この値では、スループットをさらに増加するための内部ループ展開はサポートされません。
 - 2: DSP とメモリ使用量が最大になります。この値では、内部ループ展開がサポートされ、DSP リソースの増加を抑えつつ全体的なスループットが改善されます。これは、デザインを吟味するのに最も柔軟性のあるアーキテクチャです。
- 内部ループのパイプライン処理
 - >1: ARCH 2 の場合、Vivado HLS でリソースを共有して DSP 使用量を削減できるようになります。複雑な浮動小数点データ型を使用する場合、値を 2 または 4 に設定すると、DSP 使用量がかなり削減されます。
- 内部ループ展開
 - ARCH 2 の場合、指定した係数で処理されるループをインプリメントするのに必要なハードウェアのコピーが作成され、該当するループ反復数が並列で実行され、スループットが増加しますが、DSP とメモリ使用量も増加します。

仕様

適切なクラス メンバーを定義し直すと、次の `hls::cholesky_traits` ベース クラスから派生したコンフィギュレーション クラスを使用してすべての指示子を指定できます。

```
struct MY_CONFIG :
hls::cholesky_traits<LOWER_TRIANGULAR, ROWS_COLS_A, MAT_IN_T, MAT_OUT_T>{
    static const int ARCH = 2;
    static const int INNER_II = 2;
    static const int UNROLL_FACTOR = 1;
};
```

コンフィギュレーション クラスは、次のようにテンプレート パラメーターとして `hls::cholesky_top` 関数に供給されます。

```
hls::cholesky_top<LOWER_TRIANGULAR, ROWS_COLS_A, MY_CONFIG, MAT_IN_T, MAT_OUT_T>(A, L);
```

`hls::cholesky` 関数では、次のデフォルト コンフィギュレーションが使用されます。

```
hls::cholesky<LOWER_TRIANGULAR, ROWS_COLS_A, MAT_IN_T, MAT_OUT_T>(A, L);
```

コレスキー反転および QR 反転

インプリメンテーションの制御

次の表に、リソース使用量、関数のスループット (開始間隔)、および関数のレイテンシに影響する主な要因を示します。小、中、大は、要因間で相対的なものです。

表 36: **Inverse** の主な要因のまとめ

主な要因	値	リソース	スループット	レイテンシ
サブ関数のインプリメンテーション ターゲット (コレスキー/QRF および行列乗算)	Small	小	小	大
	Balanced	中	中	中
	Fast	大	大	小
後退代入の内部および対角ループ パイプライン	1	大	大	小
	>1	小	小	大
DATAFLOW 指示子	Yes	中	大	大
INLINE 指示子	Yes	小	小	大

主な機能

次は、先ほどの表の主な要因に関する追加情報です。

- サブ関数のインプリメンテーション
 - サブ関数をコレスキーまたは QRF、後退代入、行列乗算の順に実行します。これらのサブ関数に選択されたインプリメンテーションにより、反転関数のリソース使用量および関数のスループット/レイテンシが決まります。
- 後退代入の内部および対角ループ パイプライン
 - >1: Vivado HLS でリソースを共有して DSP 使用量を削減できるようになります。
- DATAFLOW 指示子
 - 順次タスクのパイプライン処理により、関数のスループットが、各サブ関数のレイテンシの合計ではなくサブ関数の最大レイテンシに基づく開始間隔に増加します。関数のスループットは全体的なレイテンシの増加と共にかなり増加するので、追加のメモリ リソースが必要となります。
- INLINE 指示子
 - サブ関数の階層が削除され、Vivado HLS でリソースを共有しやすくなるので、DSP およびメモリ使用量を削減できます。



ヒント: DATAFLOW 指示子を適切なサブ関数インプリメンテーションと組み合わせることで、リソースと Inverse 関数のスループットを特定の要件を満たすように調整できます。

仕様

次のように DATAFLOW 指示子が `hls::cholesky_inverse_top` または `hls::qr_inverse_top` 関数に適用されます。

```
set_directive_dataflow "cholesky_inverse_top"
```

INLINE 指示子も同じように適用されます。

```
set_directive_inline -recursive "cholesky_inverse_top"
```

適切なクラス メンバーを定義し直すと、次の `hls::cholesky_inverse_traits` または `hls::qr_inverse_traits` ベース クラスから派生したコンフィギュレーション クラスを使用して各サブ関数のインプリメンテーションを指定できます。

```
typedef hls::cholesky_inverse_traits<ROWS_COLS_A,
    MAT_IN_T,
    MAT_OUT_T> MY_DFLT_CFG;

struct MY_CONFIG : MY_DFLT_CFG {
    struct CHOLESKY_TRAITS :
        hls::cholesky_traits<false,
            ROWS_COLS_A,
            MAT_IN_T,
            MY_DFLT_CFG::CHOLESKY_OUT> {
        static const int ARCH = 1;
    };
    struct BACK_SUB_CONFIG :
        hls::back_substitute_traits<ROWS_COLS_A,
            MY_DFLT_CFG::CHOLESKY_OUT,
            MY_DFLT_CFG::BACK_SUBSTITUTE_OUT> {
        static const int INNER_II = 2;
        static const int DIAG_II = 2;
    };
    struct MULTIPLIER_CONFIG :
        hls::matrix_multiply_traits<hls::NoTranspose,
            hls::ConjugateTranspose,
            ROWS_COLS_A,
            ROWS_COLS_A,
            ROWS_COLS_A,
            ROWS_COLS_A,
            MY_DFLT_CFG::BACK_SUBSTITUTE_OUT,
            MAT_OUT_T> {
        static const int INNER_II = 2;
    };
};
```

コンフィギュレーション クラスは、次のようにテンプレート パラメーターとして `hls::cholesky_inverse_top` または `hls::qr_inverse_top` 関数に供給されます。

```
hls::cholesky_inverse_top<ROWS_COLS_A,MY_CONFIG,MAT_IN_T,MAT_OUT_T>(A,INVERSE_A,inverse_OK);
```


hls::cholesky_inverse または hls::qr_inverse 関数では、次のデフォルト コンフィギュレーションが使用されます。

```
hls::cholesky_inverse<ROWS_COLS_A,MAT_IN_T,MAT_OUT_T>(A, INVERSE_A, inverse_OK);
```

行列乗算

インプリメンテーションの制御

次の表に、リソース使用量、関数のスループット (開始間隔)、および関数のレイテンシに影響する主な要因を示します。小、中、大は、要因間で相対的なものです。

表 37: 行列乗算の主な係数のまとめ

主な要因	値	リソース	スループット	レイテンシ
アーキテクチャ (ARCH)	2 (浮動小数点)	小	小	大
	3 (浮動小数点)	大	大	小
	0 (固定小数点)	小	小	大
	2 (固定小数点)	中	中	中
	4 (固定小数点)	大	大	小
内部ループのパイプライン処理 (INNER_II)	1	大	大	小
	>1	小	小	大
内部ループ展開 (UNROLL_FACTOR)	1	小	小	大
	>1	大	大	小
リソース指示子 (RESOURCE)	LUTRAM	中	なし	なし

主な機能

- アーキテクチャ

ARCH の主な係数では、インプリメンテーション データ型に基づいてアーキテクチャが選択されます。

- 浮動小数点データ型

- 2: 内部累積ループで開始間隔 (II) 1 で最大スループットが達成されるようになります。この値では、内部ループの部分的な展開がサポートされ、DSP リソースの増加を抑えつつ全体的なスループットが改善されます。
- 3: 完全に展開された内部累積ループがインプリメントされ、最大の DSP リソースの数とスループットが使用されます。

- 固定小数点データ型

- 0: 最小のリソース使用量と最小のスループットが使用されます。
- 2: 内部ループの部分的な展開がサポートされ、DSP リソースの増加を抑えつつ全体的なスループットが改善されます。
- 4: 完全に展開された内部累積ループがインプリメントされ、最大の DSP リソースの数とスループットが使用されます。

- 内部ループのパイプライン処理
 - >1: 複雑な浮動小数点データ型を使用する場合、リソースが共有され、DSP 使用量が削減されます。値を 2 または 4 に設定すると、DSP 使用量がかなり削減されます。
- 内部ループ展開
 - ARCH 2 の場合、指定した係数で処理されるループをインプリメントするのに必要なハードウェアのコピーが作成され、該当するループ反復数が並列で実行され、スループットが増加しますが、DSP とメモリ使用量も増加します。
 - ARCH 3 または 4 の場合、累積ループが完全に展開されます。
- リソース指示子

Vivado HLS は、デフォルトで配列をインプリメントするためにブロック RAM が使用されます。

 - ARCH 2 の場合、累積ループが部分的に展開されるので、Vivado HLS では複数ブロック RAM で `sum_mult` 配列が分割されます。
 - 分割されたサイズにブロック RAM を使用する必要がない場合は、`RESOURCE` 指示子を使用して LUTRAM を指定します。

仕様

適切なクラス メンバーを定義し直すと、`RESOURCE` 指示子を除き、次の `hls::matrix_multiply_traits` ベースクラスから派生したコンフィギュレーション クラスを使用してすべての指示子を指定できます。

```
struct MY_CONFIG: hls::matrix_multiply_traits<hls::NoTranspose,
hls::NoTranspose,
A_ROWS,
A_COLS,
B_ROWS,
B_COLS,
MATRIX_T,
MATRIX_T>{
static const int ARCH          = 2;
static const int INNER_II      = 1;
static const int UNROLL_FACTOR = 2;
};
```

コンフィギュレーション クラスは、次のようにテンプレート パラメーターとして `hls::matrix_multiply_top` 関数に供給されます。

```
hls::matrix_multiply_top<hls::NoTranspose,hls::NoTranspose,A_ROWS,A_COLS,B_ROWS,B_COLS,C_ROWS,C_COLS,MY_CONFIG,MATRIX_T,MATRIX_T>(A,B,C);
```

`hls::matrix_multiply` 関数では、次のデフォルト コンフィギュレーションが使用されます。

```
hls::matrix_multiply<hls::NoTranspose,hls::NoTranspose,A_ROWS,A_COLS,B_ROWS,B_COLS,C_ROWS,C_COLS,MATRIX_T,MATRIX_T>(A,B,C);
```

ARCH 2 を選択すると、次のように RESOURCE 指示子が hls::matrix_multiply_alt2 関数の sum_mult 配列に適用されます。

```
set_directive_resource -core RAM_S2P_LUTRAM "matrix_multiply_alt2" sum_mult
```

QRF

インプリメンテーションの制御

次の表に、リソース使用量、関数のスループット (開始間隔)、および関数のレイテンシに影響する主な要因を示します。小、中、大は、要因間で相対的なものです。

表 38: QRF の主な要因のまとめ

主な要因	値	リソース	スループット	レイテンシ
Q および R アップデート ループ パイプライン (UPDATE_II)	2	大	大	小
	>2	小	小	大
Q および R アップデート ループ展開 (UNROLL_FACTOR)	1	小	小	大
	>1	大	大	小
ローテーション ループ パイプライン処理 (CALC_ROT_II)	1	大	大	小
	>1	小	小	大

主な機能

次は、前の表の主な要因に関するその他の情報です。

- Q および R アップデート ループ パイプライン処理
 - 2: 最小の達成可能な開始間隔 (II) に 2 を設定します。これで、Q および R 行列乗算の要件 (アップデート ループの反復ごとに 2 つの書き込み) が満たされます。
 - >2: Vivado HLS でリソースをさらに共有して DSP 使用量を削減できるようになります。複雑な浮動小数点データ型を使用する場合、値を 4 または 8 に設定すると、DSP 使用量がかなり削減されます。
- Q および R アップデート ループ展開
 - 指定した係数で処理されるループをインプリメントするのに必要はハードウェアのコピーが作成され、該当するループ反復数が並列で実行され、スループットが増加しますが、DSP とメモリ使用量も増加します。
- ローテーション ループ パイプライン処理
 - Vivado HLS でリソースを共有して DSP 使用量を削減できるようになります。

仕様

適切なクラス メンバーを定義し直すと、次の `hls::qrf_traits` ベース クラスから派生したコンフィギュレーション クラスを使用してすべての指示子を指定できます。

```
struct MY_CONFIG : hls::qrf_traits<A_ROWS, A_COLS, MAT_IN_T, MAT_OUT_T>{
    static const int CALC_ROT_II = 4;
    static const int UPDATE_II= 4;
    static const int UNROLL_FACTOR= 2;
};
```

コンフィギュレーション クラスは、次のようにテンプレート パラメーターとして `hls::qrf_top` 関数に供給されます。

```
hls::qrf_top<TRANPOSED_Q, A_ROWS, A_COLS, MY_CONFIG, MAT_IN_T, MAT_OUT_T>(A, Q, R)
;
```

`hls::qrf` 関数では、次のデフォルト コンフィギュレーションが使用されます。

```
hls::qrf<TRANPOSED_Q, A_ROWS, A_COLS, MAT_IN_T, MAT_OUT_T>(A, Q, R);
```

SVD

インプリメンテーションの制御

次の表に、リソース使用量、関数のスループット (開始間隔)、および関数のレイテンシに影響する主な要因を示します。小、中、大は、要因間で相対的なものです。

表 39: SVD の主な係数のまとめ

主な要因	値	リソース	スループット	レイテンシ
ALLOCATION 指示子 (<code>vm2x1_base</code> 制限)	1	小	小	大
	>1	大	大	小
非対角ループ パイプライン (<code>OFF_DIAG_II</code>)	4	大	大	小
	>4	小	小	大
対角ループ パイプライン (<code>DIAG_II</code>)	1	大	大	小
	>1	小	小	大
反復 (<code>NUM_SWEEP</code>)	<10	なし	大	小
逆数平方根演算子	組み合わせ演算子	中	大	小

主な機能

次は、前の表の主な要因に関するその他の情報です。

- ALLOCATION 指示子
 - インプリメントされた 2x1 ベクター ドット積の数が制限されます。Vivado HLS では、指定した数の 2x1 ベクター ドット積カーネルを使用するように SVD 関数がスケジューリングされます。

注記: SVD アルゴリズムは、複雑なデータ型の場合は特に計算負荷が高く、`ALLOCATION` 指示子を使用するのが、リソース使用量とスループットのバランスを調整するのに最適な方法です。

- 非対角ループ パイプライン
 - 4: 最小の達成可能な開始間隔 (II) に 4 を設定します。これで、S、U および V 配列の要件 (非対角ループの反復ごとに 4 つの書き込み) が満たされます。
 - >4: Vivado HLS でリソースをさらに共有して DSP 使用量を削減できるようになります。
- 対角ループ パイプライン
 - >1: Vivado HLS でリソースが共有できるようになります。
- 反復

SVD 関数で反復的な両面のヤコビ法が使用されます。

 - 10: デフォルトの反復数が設定されます。
 - <10: 必要なパフォーマンスを達成するのに最小限の反復数が設定され、関数スループットが最大化されます。
- 逆数平方根演算子
 - 離散演算子よりもレイテンシがかなり短くなります。

注記: デフォルトでは、Vivado HLS では組み合わせられた `rsqrt` 演算子は使用されず、離散除算と `sqrt` 演算子を使用されます。`-unsafe_math_optimizations` コンパイラ オプションを選択すると、`rsqrt` 演算子を使用できるようになります。

仕様

`ALLOCATION` 指示子は次のように `INLINE` 指示子と組み合わせて `hls::svd_pairs` 関数に適用できます。

```
set_directive_inline -off "vm2x1_base"
set_directive_allocation -limit 1 -type function "svd_pairs" vm2x1_base
```

`-unsafe_math_optimizations` コンパイラ オプションは次のように選択できます。

```
config_compile -unsafe_math_optimizations
```

適切なクラス メンバーを定義し直すと、次の `hls::svd_traits` ベース クラスから派生したコンフィギュレーション クラスを使用してその他すべての指示子を指定できます。

```
struct MY_CONFIG : hls::svd_traits<A_ROWS, A_COLS, MATRIX_IN_T, MATRIX_OUT_T> {
    static const int NUM_SWEEPS = 6;
    static const int DIAG_II = 4;
    static const int OFF_DIAG_II = 4;
};
```

コンフィギュレーション クラスは、次のようにテンプレート パラメーターとして `hls::svd_top` 関数に供給されます。

```
hls::svd_top<A_ROWS, A_COLS, MY_CONFIG, MATRIX_IN_T, MATRIX_OUT_T>(A, S, U, V);
```

`hls::svd` 関数では、次のデフォルト コンフィギュレーションが使用されます。

```
hls::svd<A_ROWS, A_COLS, MATRIX_IN_T, MATRIX_OUT_T>(A, S, U, V);
```

HLS DSP ライブラリ

HLS DSP ライブラリには、C++ で DSP システムをモデルするブロック構築関数、特に SDR アプリケーションで使われる関数が含まれています。次の表には、HLS DSP ライブラリの関数がまとめられています。

表 40: HLS DSP ライブラリ

関数	データ型	インプリメンテーション形式
atan2	入力: std::complex< ap_fixed > 出力: ap_ufixed	合成済み
awgn	入力: ap_ufixed 出力: ap_int	合成済み
cmpy	入力: std::complex< ap_fixed > 出力: std::complex< ap_fixed >	合成済み
convolution_encoder	入力: ap_uint 出力: ap_uint	合成済み
nco	入力: ap_uint 出力: std::complex< ap_int >	合成済み
sqrt	入力: ap_ufixed、ap_int 出力: ap_ufixed、ap_int	合成済み
viterbi_decoder	入力: ap_uint 出力: ap_uint	合成済み

関数では、入力および出力データを必要に応じて記述するのに Vivado HLS の固定精度型 `ap_[u]int` および `ap_[u]fixed` が使用されます。柔軟性を最大限にするため、関数には最低限のインターフェイス タイプがあります。たとえば、1 サンプル入力に 1 サンプル出力といった、簡単なスループット モデルの関数はポインター インターフェイスを使用します。viterbi_decoder など、レート変更をする関数はインターフェイスに `hls::stream` タイプを使用します。

任意の関数に対しポインター インターフェイスや AXI4-Stream インターフェイスの `hls::stream` を作成するなど、既存ライブラリをコピーして、インターフェイスをさらに複雑にできます。ただし、複雑なインターフェイスはさらにリソースを必要とします。

Vivado HLS では、ほとんどのライブラリ エlementをテンプレート化した C++ クラスとして提供されています。これらは、ヘッダー ファイル (`hls_dsp.h`) で、コンストラクター、デストラクター、演算子アクセス関数を使用して記述されています。

DSP ライブラリの使用

DSP 関数は次のいずれかの方法で参照できます。

- スコープ付き命名規則は、次のように使用します。

```
#include <hls_dsp.h>
static hls::awgn<output_width> my_awgn(seed);
my_awgn(snr, noise);
```

- hls 名前空間は次のように使用します。

```
#include <hls_dsp.h>
using namespace hls;
static awgn<output_width> my_awgn(seed);
my_awgn(snr, noise);
```

DSP ライブラリの関数には、ソース コードの pragma として合成指示子が含まれています。これにより、一般的な要件を満たすため、関数を合成するにあたり、Vivado HLS を誘導していきます。関数はスループットが最大になるように最適化されますが、これが一般的なユース ケースです。たとえば、テンプレートのパラメーター コンフィギュレーションにかかわらず、開始間隔 (II) が 1 になるように、配列が完全に分割される可能性があります。

既存の最適化の削除や追加最適化は次のように適用します。

- DSP 関数に最適化を適用するには、Vivado HLS の GUI で hls_dsp.h ヘッダー ファイルを開いて、次のいずれかの操作を実行します。
 - 。 [Ctrl] キーを押しながら [#include "hls_dsp.h"] をクリックします。
 - 。 [Explorer] タブを使用して、[Includes] フォルダーの下のファイルを参照します。
- 指示子として最適化を追加したり削除するには、[Information] でヘッダー ファイルを開き、[Directives] タブを使用します。

注記: pragma として最適化を追加する場合、Vivado HLS はライブラリに最適化を保存し、デザインにヘッダーを追加するたびにそれを適用します。pragma として最適化を追加するには、ファイルへの書き込み権限が必要になる場合があります。



ヒント: RTL インプリメンテーションを変更するのに関数を変更する必要がある場合は、接頭辞 `TIP` のあるライブラリ ソース コードのコメントを検索します。このコメントには、pragma の配置や指示子の適用に適した位置が示されています。

HLS SQL ライブラリ

SQL ライブラリには、C++ の SQL 構築ブロック関数が含まれます。次の表は、HLS SQL ライブラリの関数を示しています。

表 41: HLS SQL ライブラリ

関数	データ型	注記
hls_alg::sha224	入力: hls::stream<unsigned char> 出力: hls::stream<unsigned char>	SHA-2 ファミリーから SHA-224 アルゴリズムをインプリメント。
hls_alg::sha256	入力: hls::stream<unsigned char> unsigned long long 出力: hls::stream<unsigned char>	SHA-2 ファミリーから SHA-256 アルゴリズムをインプリメント。

表 41: HLS SQL ライブラリ (続き)

関数	データ型	注記
hls_alg::sort	入力: hls::stream<T> 出力: hls::stream<T>	Bitonic 分類アルゴリズムをインプリメント。 T はデータ型。

Vivado HLS には、これらのライブラリ エLEMENT が hls_db namespace でテンプレート化された C++ 関数として提供されています。すべての SQL 関数の説明については、第 4 章「HLS SQL ライブラリ関数」を参照してください。

SQL ライブラリの使用

SQL 関数は次のいずれかの方法で参照できます。

```
#include <hls_alg.h>
hls_alg::sha256(in_stream, in_stream_depth, out_stream);
```

SQL ライブラリの関数には、ソース コードの pragma として合成指示子が含まれています。これにより、一般的な要件を満たすため、関数を合成するにあたり、Vivado HLS を誘導していきます。

高位合成コーディング スタイル

この章では、さまざまな C、C++、および SystemC のコンストラクトがどのように FPGA ハードウェア インプリメンテーションに合成されるかについて説明します。



重要: この資料で「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。

この資料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。次のいずれかの方法を使用すると、コード例を入手できます。

- 最初のウェルカム画面で [Open Example Project] をクリック。
注記: ウェルカム画面は、[Help]→[Welcome] をクリックするといつでも表示できます。
- Vivado HLS インストール ディレクトリの `examples/coding` ディレクトリ。

サポートされない C コンストラクト

Vivado® HLS では、広範囲の C 言語がサポートされていますが、合成不可能なコンストラクト、後のデザイン フローでエラーを発生させるコンストラクトもあります。このセクションでは、関数を合成してデバイスにインプリメントできるようにするため、コードに加える必要のある変更について説明します。

合成するには、次の条件を満たしている必要があります。

- C 関数にデザインの機能全体が含まれている。
- OS へのシステム コールで実行できる機能なし。
- C コンストラクトのサイズが固定されているか、制限されている。
- これらのコンストラクトのインプリメンテーションがあいまいでない。

システム コール

システム コールは、C プログラムが実行されている OS で一部のタスクを実行するので、合成できません。

Vivado® HLS では、`printf()` や `fprintf(stdout,)` などのデータを表示するだけでアルゴリズムの実行には影響しないシステム コールは無視されます。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 に対して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 `__SYNTHESIS__` マクロが定義されます。これにより、`__SYNTHESIS__` マクロを使用して、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。

注記: `__SYNTHESIS__` マクロは、合成されるコードにのみ使用します。このマクロはCシミュレーションまたはCRTL協調シミュレーションには従っていないので、テストベンチには使用しないでください。



注意: `__SYNTHESIS__` をコードまたはコンパイラ オプションで定義または定義解除すると、コンパイルエラーが発生することがあります。

次に、サブ関数からの中間結果をハードドライブのファイルに保存するコード例を示します。`__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(dint_t *in1, dint_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(dint_t A, dint_t B, dout_t *C, dout_t *D)
{
    dint_t apb, amb;

    sumsub_func(&A, &B, &apb, &amb);
#ifdef __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);
}
```

`__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コードが異なるものになります。



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合成の結果が異なるものになります。このようなコードのエラーはデバッグしにくいので、`__SYNTHESIS__` マクロは機能を変更するために使用しないでください。

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。次に、`malloc()` を使用するデザインを合成可能なバージョンに変換するコード例を示します。次の2つの便利なコーディングスタイル手法が含まれます。

- このデザインでは `__SYNTHESIS__` マクロは使用しません。

合成可能なバージョンまたは合成不可能なバージョンの選択には、ユーザー定義の `NO_SYNTH` マクロを使用します。これにより、同じコードがCでシミュレーションされ、Vivado® HLSで合成されます。

- `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なります。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) {

#ifdef 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;
}
```

コード変更はデザインの機能に影響するので、ザイリンクスでは `__SYNTHESIS__` マクロの使用はお勧めしません。ザイリンクスでは、次の手順をお勧めします。

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、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を検証します。
4. 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、同じコードが合成されるようになります。Cでのダイナミックメモリ使用に関する制限と同様、Vivado HLSでは、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() {â};
};

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

foo(a);
```

ポインタの制限

一般的なポインタの型変換

Vivado HLS では、一般的なポインタの型変換はサポートされませんが、ネイティブ C 型間ではサポートされます。

ポインタ配列

Vivado HLS では、各ポインタがスカラーまたはスカラーの配列を指定する場合に、ポインタ配列が合成でサポートされます。ポインタ配列では、別のポインタを指定することはできません。

関数ポインタ

関数ポインタはサポートされていません。

再帰関数

再帰関数は合成できません。これは、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));
}
```

Vivado® HLS では、有限数の関数呼び出しがある末尾再帰は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、テンプレートで末尾再帰をインプリメントできます。C++ については、次で説明します。

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。

注記: 合成では `std::complex` のような標準データ型がサポートされます。

C 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を検証されます。この手順は、テストベンチで実行されます。テストベンチが良いと、生産性がかなりあがります。

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。アルゴリズムが正しいことがわかっているので、合成前にコード変更を検証できます。
- Vivado® HLS では 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検証できます。

注記: テストベンチに入力引数を提供されるようにするには、[Project] → [Project Settings] で [Simulation] をクリックし、[Input Arguments] オプションを使用します。テストベンチには、ユーザーがインタラクティブに入力することはできません。Vivado HLS の GUI にはコマンド コンソールがないので、テストベンチ実行中のユーザー入力は受け入れられません。

ザイリンクスでは、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をお勧めしています。次に、次の 2 つのサブ関数を呼び出す `hier_func` 関数を含むデザインのコード例を示します。

- `sumsub_func`: 加算および減算を実行。
- `shift_func`: シフトを実行。

次のデータ型はヘッダー ファイル (`hier_func.h`) で定義されています。

```
#include "hier_func.h"

int sumsub_func(dint_t *in1, dint_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(dint_t A, dint_t B, dout_t *C, dout_t *D)
{
```

```
dint_t apb, amb;

sumsub_func(&A,&B,&apb,&amb);
shift_func(&apb,&amb,C,D);
}
```

最上位関数には複数のサブ関数を含めることができますが、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。

`hier_func` 関数を合成する方法は、次のとおりです。

1. 上記の例に示すように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ます。
2. `hier_func` を最上位関数として指定します。

合成後:

- 最上位関数に対する引数(上記の例では A、B、C、および D)は RTL ポートに合成されます。
- 最上位に含まれる関数(上記の例では `sumsub_func` および `shift_func`)は、階層ブロックに合成されます。

上記の例のヘッダー 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になり、読みやすくなることを示しています。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変数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
```

この例のヘッダー 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。これらの定義は、同じヘッダー ファイルも含んだテストベンチで使用されます。

次のコード例は、最初の例の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 corre
return retval;
}
```

生産的なテストベンチ

テストベンチ例は、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出されますが、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、次のように設定されます。
 - 0: 結果が正しいことを示します。
 - 0 以外: 結果が正しくないことを示します。

注記: テストベンチから 0 以外の値が返されることがあります。複雑なテストベンチには、相違点やエラーのタイプによって異なる値を返すものもあります。C シミュレーションまたは C/RTL 協調シミュレーション後にテストベンチから 0 以外の値が返されると、Vivado® HLS でエラーまたはシミュレーション エラーがレポートされます。



推奨: `main()` 関数の戻り値はシステム環境で解釈されるので、ザイリンクスでは、戻り値を 8 ビットの範囲に制約して、ポータビリティと安全性を上げることをお勧めします。



注意: テストベンチで結果がチェックされるようにするのは、ユーザーの責任です。テストベンチで結果がチェックされないのに 0 が返された場合は、Vivado HLS で結果が実際にチェックされなかったのにシミュレーション テストが「PASS」と示されます。出力データが正しくて有効であって、テストベンチから `main()` 関数に 0 が返されない場合は、Vivado HLS でシミュレーション エラーがレポートされます。

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 関数への変更をすばや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、次のようなファイルです。

- テストベンチによりアクセスされるファイル。
- テストベンチが正しく動作するために必要なファイル。

テストベンチ例に示す `inA.dat` および `inB.dat` などが、このようなファイルの例です。これらのファイルは、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。

次の例は、[C テストベンチ](#)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、`sumsub_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。
- 次の例のファイルをデザイン ファイルおよび and プロジェクト ファイルの両方次の例ではとして追加します。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ス化されていなくても、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);
}
```

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。次の例は、[C テストベンチ](#) ~ [C テストベンチ](#)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。ほか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、その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;
}

```

関数

最上位関数は合成後に最上位 RTL デザインになり、サブ関数は RTL デザインの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 および VHDL) が作成されます。

関数のインライン展開

サブ関数をオプションでインライン展開し、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。関数をインライン展開することにより最適化が改善されることはありますが、ランタイムは増加する可能性があります。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ます。



ヒント: Vivado® HLS では、小型の関数のインライン展開は自動的に実行されます。小型の関数の自動インライン展開をにするには、その関数に対して `inline` 指示子を `off` に設定します。

関数がインライン展開されると、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。

コーディング スタイルの影響

関数のコーディング スタイルは、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ます。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 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。

次のコード例のように、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;
}
```

2 つの関数入力に任意精度型を使用するだけで、Vivado HLS で 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。ザイリンクスでは、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勧めしています。

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。

RTL ブラック ボックス

RTL ブラック ボックスを使用すると、既存の RTL IP を HLS デザインに統合して、HLS デザイン フローで実行できるようになります。RTL IP は、シーケンシャル、パイプライン、またはデータフロー領域で使用できます。RTL IP を HLS に統合するには、次のファイルが必要です。

1. ブラック ボックス記述ファイル
2. RTL IP ファイル

3. RTL の C インプリメンテーション

RTL IP を HLS デザインに統合する手順は、次のとおりです。

1. RTL IP の C インプリメンテーションを作成します。
2. HLS デザインの C インプリメンテーション関数を呼び出します。
3. 必要なフィールドを含めた JSON ファイルを作成します。サンプル JSON ファイルとこのフォーマットに関する情報は、[RTL ブラックボックスの JSON ファイル](#) を参照してください。
4. `add_files` オプションを使用して JSON ファイルを `script.tcl` ファイルに追加します。

```
add_files -blackbox my_file.json
```

5. C シミュレーション、合成、協調シミュレーションなどの HLS デザイン フローを実行します。

要件および制限

- HLS 内での RTL ブラック ボックスのサポートは C++ に制限されます。
- HLS 内では、RTL ブラック ボックスは最上位インターフェイスの I/O 信号には接続できません。
- HLS 内では、RTL ブラック ボックスは直接 DUT としては使用できません。
- HLS 内では、RTL ブラック ボックスで構造体またはクラス タイプのインターフェイスがサポートされません。
- HLS 内での RTL ブラック ボックスでは次のインターフェイス プロトコルがサポートされます。
 - `hls::stream`: RTL ブラック ボックス IP は `hls::stream` インターフェイスをサポートします。この特定のデータ型が C で使用される場合は、RTL ブラック ボックス IP でこの引数に FIFO インターフェイスを使用します。
 - 配列: RTL ブラック ボックス IP は RAM (配列) インターフェイスをサポートします。このコンストラクトが C で使用される場合は、RTL ブラック ボックス IP で対応する引数に次の RAM インターフェイスの 1 つを使用します。
 - ・ シングル ポート RAM - `RAM_1P`
 - ・ デュアル ポート RAM - `RAM_T2P`
 - C スカラーおよび入力ポインター: RTL ブラック ボックス IP では、C スカラーと入力ポインターがシーケンシャルおよびパイプライン領域でのみサポートされます (データフロー領域ではサポートされません)。このコンストラクトが C で使用される場合は、RTL IP で `wire` を使用します。
 - `inout` および `out` ポインター: RTL ブラック ボックス IP では、`inout` および `out` ポインターがシーケンシャルおよびパイプライン領域でのみサポートされます (データフロー領域ではサポートされません)。このコンストラクトを C で使用する場合、RTL IP は `out` ポインターに `ap_vld` を、`inout` ポインターに `ap_ovld` を使用する必要があります。
- HLS に提供される RTL IP ファイルは Verilog (.v) である必要があります。
- RTL IP モジュールには一意的クロック信号と一意的リセット信号 (立ち上がりレベル High) が含まれている必要があります。
- RTL IP モジュールには、RTL IP をイネーブルまたは停止する CE 信号が含まれている必要があります。
- RTL IP は、`ap_ctrl_chain` プロトコルを使用する必要があります。詳細は、[ブロック レベル I/O プロトコル](#) を参照してください。

JSON ファイルの制限:

- `[c_function_name]` フィールドは、C 関数モデルと同じにする必要があります。
- `[rtl_top_module_name]` は `[c_function_name]` と同じにする必要があります。

- 未使用の [c_parameters] フィールドは、テンプレートから削除されます。
- [c_parameter] フィールドはすべて [rtl_port] フィールドに関連付ける必要があります。

注記: RTL ブラック ボックスを使用する場合、その他すべての HLS デザインの制限事項も適用されたままです。

JSON ファイルのフォーマット

次の表に、JSON ファイルのフォーマットを示します。

表 42: JSON ファイルのフォーマット

項目	属性	説明
c_function_name		ブラック ボックスの C++ 関数名
rtl_top_module_name		ブラック ボックスの RTL 関数名
c_files	c_file	ブラック ボックス モジュールに使用される C ファイルを指定。
	cflag	該当する C ファイルに必要なコンパイル オプションを提供。
rtl_files		ブラック ボックス モジュールの RTL ファイルを指定。

表 42: JSON ファイルのフォーマット (続き)

項目	属性	説明
c_parameters	c_name	ブラック ボックス C++ 関数に使用される引数の名前を指定。
	c_port_direction	<p>該当する C 引数のアクセス方向。</p> <ul style="list-style-type: none"> in: ブラック ボックス C++ 関数でのみ読み出し。 out: ブラック ボックス C++ 関数でのみ書き込み。 inout: ブラック ボックス C++ 関数で読み出しおよび書き込みの両方を実行。
	RAM_type	<p>該当する C 引数が RTL の RAM プロトコルを使用する場合の RAM タイプを指定。次の 2 つのタイプの RAM を使用可能です。</p> <ul style="list-style-type: none"> RAM_1P: 1 ポートの RAM モジュール用 RAM_T2P: 2 ポートの RAM モジュール用 <p>該当する C 引数が RTL の RAM プロトコルを使用しない場合、この属性は除外。</p>
	rtl_ports	<p>該当する C 引数の RTL ポート プロトコル信号を指定。次の 5 つのタイプの RTL ポート プロトコルを使用可能:</p> <ul style="list-style-type: none"> wire: C 引数は、スカラーまたは input 方向のポインターを使用する場合は、wire にマップ可能。 ap_vld: C 引数は、out 方向のポインターを使用する場合は ap_vld にマップ可能。 ap_ovld: C 引数は、inout 方向のポインターを使用する場合は ap_ovld にマップ可能。 FIFO: C 引数は、hls::stream データ型を使用する場合は FIFO にマップ可能。 RAM: C 引数は、配列型を使用する場合は RAM にマップ可能。配列型では、inout 方向がサポートされます。 <p>上記で指定した RTL ポート プロトコルには、関連する制御信号があり、JSON ファイルで指定する必要があります。使用方法の詳細は、次の表を参照してください。</p>
c_return	c_port_direction	out にする必要があります。
	rtl_ports	RTL ブラック ボックス IP で使用される該当する RTL ポート名を指定。

表 42: JSON ファイルのフォーマット (続き)

項目	属性	説明
rtl_common_signal	module_clock	RTL ブラック ボックス モジュールの一意のクロック信号。
	module_reset	RTL ブラック ボックス モジュールのリセット信号を指定。リセット信号は、アクティブ High のまたは正の valid にする必要があります。
	module_clock_enable	RTL ブラック ボックス モジュールのクロック イネーブル信号を指定。イネーブル信号は、アクティブ High にする必要があります。
	ap_ctrl_chain_protocol_idle	RTL ブラック ボックス モジュール用の ap_ctrl_chain プロトコルの ap_idle 信号。
	ap_ctrl_chain_protocol_start	RTL ブラック ボックス モジュール用の ap_ctrl_chain プロトコルの ap_start 信号。
	ap_ctrl_chain_protocol_ready	RTL ブラック ボックス IP 用の ap_ctrl_chain プロトコルの ap_ready 信号。
	ap_ctrl_chain_protocol_done	ブラック ボックス RTL モジュール用の ap_ctrl_chain プロトコルの ap_done 信号。
	ap_ctrl_chain_protocol_continue	RTL ブラック ボックス モジュール用の ap_ctrl_chain プロトコルの ap_continue 信号。
rtl_performance	latency	RTL ブラック ボックス モジュールのレイテンシを指定。負以外の整数値にする必要があります。組み合わせ RTL IP の場合は 0 を、それ以外の場合はその RTL モジュールのレイテンシをそのまま指定します。
	II	関数が新しい入力データを受信できるようになるまでのクロック サイクル数。負以外の整数値にする必要があります。0 はブラック ボックスがパイプライン処理できないことを意味します。それ以外の場合、ブラック ボックス モジュールはパイプライン処理されます。
rtl_resource_usage	FF	RTL ブラック ボックス モジュールのレジスタ使用率を指定。
	LUT	RTL ブラック ボックス モジュールの LUT 使用率を指定。
	BRAM	RTL ブラック ボックス モジュールのブロック RAM 使用率を指定。
	URAM	RTL ブラック ボックス モジュールの URAM 使用率を指定。
	DSP	RTL ブラック ボックス モジュールの DSP 使用率を指定。

表 43: RTL ポート プロトコル

RTL ポート プロトコル	RAM タイプ	C ポートの方向	属性	ユーザー定義の名前	注記
wire		in	data_read_in	RTL ブラック ボックス IP で使用されるユーザー定義の名前を指定。たとえば、wire の場合、RTL ポート名が flag であれば、JSON ファイルのフォーマットは "data_read-in" : "flag" になります。	
ap_vld		out	data_write_out		
			data_write_valid		
ap_ovld		inout	data_read_in		負の valid にする必要あり。
			data_write_out		
			data_write_valid		
FIFO		in	FIFO_empty_flag		
			FIFO_read_enable		
			FIFO_data_read_in		
		out	FIFO_full_flag		負の valid にする必要あり。
			FIFO_write_enable		
			FIFO_data_write_out		
RAM	RAM_1P	in	RAM_address		
			RAM_clock_enable		
			RAM_data_read_in		
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
		inout	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		

表 43: RTL ポート プロトコル (続き)

RTL ポート プロトコル	RAM タイプ	C ポートの方向	属性	ユーザー定義の名前	注記
RAM	RAM_T2P	in	RAM_address	RTL ブラック ボックス IP で使用されるユーザー定義の名前を指定。たとえば、wire の場合、RTL ポート名が flag であれば、JSON ファイルのフォーマットは "data_read-in" : "flag" になります。	_snd の付いた信号は RAM の 2 つ目のポートに属します。_snd の付いていない信号は 1 つ目のポートに属します。
			RAM_clock_enable		
			RAM_data_read_in		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_data_read_in_snd		
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_write_enable_snd		
		inout	RAM_data_write_out_snd		
			RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_write_enable_snd		
			RAM_data_write_out_snd		
			RAM_data_read_in_snd		

注記: RTL ブラック ボックスのビヘイビア C 関数モデルも、推奨される HLS コーディング スタイルに従う必要があります。

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、パイプライン処理、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最小限に抑えることができます。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が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。次のコード例では、ループ境界が最上位入力から駆動される変数 (`width`) により決定されます。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;
}
```

上記の例で設計を最適化しようとする、変数ループ境界により発生する問題が明らかになります。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。

ループ境界が可変の場合、Vivado HLS ではそのレイテンシが正確な値ではなく、クエスション マーク (?) でレポートされます。次に、上記の例の合成結果を示します。

```
+ Summary of overall latency (clock cycles):
* Best-case latency:      ?
* Worst-case latency:     ?
+ Summary of loop latency (clock cycles):
+ LOOP_X:
* Trip count:  ?
* Latency:      ?
```

可変ループ境界には、デザインのパフォーマンスが不明になるという問題もあります。この問題は次の2つの方法で回避できます。

- `tripcount` 指示子を使用します。この方法の詳細は、次に説明します。
- C コードで `assert` マクロをコード内で使用します。

tripcount 指示子を使用すると、ループに指定する最小および最大の両方またはいずれかの tripcount を指定できます。tripcount はループの繰り返し回数を意味します。最初の例のように最大の 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
* Worst-case latency:    34
+ Summary of loop latency (clock cycles):
+ LOOP_X:
* Trip count: 0 ~ 32
* Latency:    0 ~ 32
```

tripcount 指示子を使用しても、合成結果には影響はありません。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ます。tripcount の値は Vivado HLS で数がレポートされるので、さまざまなソリューションからのレポートを比較できます。この同じループ境界情報を合成に使用するには、C コードをアップデートする必要があります。

tripcount 指示子を使用しても、合成結果には影響ありません。

次に、最初の例をより短い開始間隔で最適化する手順を示します。

- ループを展開し、並列累算が実行されるようにします。
- 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せん。

```
@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なくなります。

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。可変ループ境界のコードは、次のコード例に示す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; x++) {
        if (x<width) {
```

```

    out_accum += A[x];
  }
}

return out_accum;
}

```

上記の例の 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、 $N(32)$ 個のコピーがあります。このループ本体の各コピーには、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、実行時間も短縮されます。次のコード例は、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;
}

```

最内ループ (LOOP_J) がパイプライン処理されると、ハードウェア (単一の乗算器) には LOOP_J のコピーが 1 つ作成されます。Vivado® HLS では、できるだけループをフラットにするので、この場合は 20x20 の繰り返しの 1 つの新しいループが作成されます。スケジューリングする必要があるのは、乗算器演算 1 つと配列アクセス 1 回のみです。これらをスケジューリングしておく、ループの繰り返しの 1 つのループ本体のエントリティ (20X20 のループ繰り返し) としてスケジューリングできます。



ヒント: 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、ループ本体のコピーが 20 個作成されるので、乗算器 20 個と配列 20 個のアクセスをスケジューリングする必要があります。LOOP_I の各反復は 1 つの要素としてスケジューリングできます。

最上位関数がパイプライン処理されると、両方のループを展開する必要があるため、乗算器 400 個と配列 400 個のアクセスをスケジューリング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ほぼありません。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したときに、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、すべてのサブループが展開されるので、スケジューリングする必要のある演算の数が大幅に増えることがあります (実行時間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点で最高のパフォーマンスのデザインが得られます。

上記のオプションをまとめると、次のようになります。

- `LOOP_J` をパイプライン処理
レイテンシは約 400 サイクル (20 x 20) になり、100 個未満の LUT およびレジスタが必要になります (I/O 制御および FSM は常に存在)。
- `LOOP_I` をパイプライン処理
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。
- 関数 `loop_pipeline` をパイプライン処理
レイテンシは約 10 サイクル (デュアル ポート アクセス 20 回) になりますが、何千個の 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。

不完全な入れ子のループ

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、内側のループをフラット化し、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) によるサイクルを削除することにより、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。

不完全な入れ子のループの場合、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。デザインに入れ子の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化されるようにします。ログ ファイルまたは合成レポートで、ループ ラベルが結合されていること (`LOOP_I` と `LOOP_J` が `LOOP_I-LOOP_J` としてレポートされるなど) を確認してください。

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リングされます。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リングされますが、ループを並列に実行することはできません。

次のコード例が合成されると、`SUM_X` ループがスケジューリングされ、その後 `SUM_Y` ループがスケジューリングされます。`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ても、`SUM_X` の後にスケジューリングされます。

```
#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<xlmit; 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;
}
}
```

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。次のコード例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) {

    sub_func(A,X,xlimit);
    sub_func(B,Y,ylimit);
}
```

前の例が合成されると、レイテンシはシーケンシャル ループの例の半分になります。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。

シーケンシャル ループの例には、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わかりやすいです。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。次のループを開始するには、前のループの各反復が終了する必要があります。

```
Minim_Loop: while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

このループはパイプライン処理できません。ループの前の反復が終了するまで次の反復を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。

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 HLS inline off
        SRL:for (k = N-1; k >= 0; --k) {
        #pragma HLS 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 `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 HLS inline off
    SRL:for (k = N-1; k >= 0; --k) {
#pragma HLS 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);
};

```

配列

コーディング スタイルによって合成後の配列のインプリメンテーションがどのように変わるかについて説明する前に、C シミュレーションなど合成が実行される前に発生する可能性のある問題について説明します。

次のようになり大きな配列が指定される場合は、C シミュレーションでメモリ不足によりエラーになる可能性があります。

```

#include "ap_cint.h"

int i, acc;
// Use an arbitrary precision type
int32 la0[100000000], la1[100000000];

for (i=0 ; i < 100000000; i++) {
    acc = acc + la0[i] + la1[i];
}

```

シミュレーションはメモリ不足のためエラーになることがあります。これは、配列がメモリに存在するスタックに配置され、OS で管理されローカル ディスクを使用可能なヒープには配置されないためです。

つまり、デザインを実行したときにメモリ不足になり、次のような状況によって問題が悪化することもあります。

- PC では使用可能なメモリが大型の Linux ボックスよりも少ないことがよくあり、使用可能なメモリが少ないことがあります。
- 任意精度型を使用すると、標準 C 型よりも多くのメモリが必要になるので、この問題が悪化する可能性があります。
- C++ および SystemC のより複雑な固定小数点任意精度型を使用すると、さらに多くのメモリが必要となるので、問題が悪化する可能性があります。

C/C++ コード開発のメモリ リソースを向上するには、リンカー オプションを使用してスタックのサイズを増加するのが標準的な方法です。たとえば、`-Wl,--stack,10485760` のようにスタック サイズを明示的に設定します。Vivado® HLS でこれを適用するには、[Project Settings]→[Simulation]→[Linker flags] をクリックするか、次のように Tcl コマンドのオプションとして指定します。

```
csim_design -ldflags {-Wl,--stack,10485760}
cosim_design -ldflags {-Wl,--stack,10485760}
```

マシンに十分なメモリがない場合は、スタック サイズを増加しても効果はありません。

次の例のように、シミュレーションにはダイナミック メモリ割り当てを、合成には固定サイズの配列を使用して、問題を回避してください。この場合、必要なメモリはヒープに割り当てられ、OS で管理されるので、ローカル ディスク空間が使用できるようになります。

このようなコードへの変更は、シミュレーションされるコードと合成されるコードが異なってしまうため、理想的ではありませんが、デザインを実行するにはこれしか方法がない場合があります。これを実行した場合は、C テストベンチでこの配列にアクセスするすべての点が記述されるようにしてください。これにより、`cosim_design` で実行される RTL シミュレーションでメモリ アクセスが正しいかどうかを検証されるようになります。

```
#include "ap_cint.h"

int i, acc;
#ifdef __SYNTHESIS__
    // Use an arbitrary precision type & array for synthesis
    int32 la0[10000000], la1[10000000];
#else
    // Use an arbitrary precision type & dynamic memory for simulation
    int32 *la0 = malloc(10000000 * sizeof(int32));
    int32 *la1 = malloc(10000000 * sizeof(int32));
#endif
    for (i=0 ; i < 10000000; i++) {
        acc = acc + la0[i] + la1[i];
    }
```

注記: `__SYNTHESIS__` マクロは、合成されるコードにのみ使用します。このマクロは C シミュレーションまたは C RTL 協調シミュレーションには従っていないので、テストベンチには使用しないでください。

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。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 1024 未満の配列は、SRL に最適化されます。1024 を超える配列は、最適化設定によって内部ブロック RAM、LUTRAM、UltraRAM に合成されます。

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よくあります。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必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。

Vivado HLS では、ポインター配列がサポートされます。各ポインターは、スカラーまたはスカラーの配列のみを指定できます。

注記: 配列はサイズ指定する必要があります。たとえば、`Array[10]`; のようなサイズ指定された配列はサポートされますが、`Array[]`; のようにサイズ指定のない配列はサポートされません。

配列アクセスとパフォーマンス

次のコード例では、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。この例では、配列 `mem[N]` に 3 回アクセスして合計を作成しています。

```
#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;
}
```

合成中、配列は RAM としてインプリメントされます。RAM をシングルポート RAM として指定すると、クロックサイクルごとに新しいループ反復を処理するように SUM_LOOP ループをパイプライン処理することは不可能です。

SUM_LOOP を開始間隔 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 を達成できなかったため、Vivado® HLS により制約が緩和されます。

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

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 つの読み込み (および 1 つの書き出し) が実行できる点にあります。

- SUM_LOOP サイクル 1: `mem[i]` を読み出し
- SUM_LOOP サイクル 2: `mem[i-1]` を読み出して値を合計
- SUM_LOOP サイクル 3: `mem[i-2]` を読み出して値を合計

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。

上記の例のコードをスループット 1 でパイプラインができるように変更すると、次のコードになります。次のコード例では、先行読み出しを実行し、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;
}
```

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、このような指示子を使用される場合、コードを変更する必要はありません。配列は、ブロックまたは個々の要素に分割できます。Vivado HLS で配列が個々の要素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。個々の要素に分割される場合、各要素は 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、分割によりさらに多くの要素に並列アクセスできるようになり、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

配列アクセスでは、配列が FIFO としてインプリメントされる場合に特に注意が必要です。これは、データフロー最適化を使用する場合によく発生します。

FIFO へのアクセスは、位置 0 から順に実行される必要があります。また、1 つの配列が複数の位置で読み出される場合、コードで FIFO アクセスの順序を厳密に制御する必要があります。通常、ファンアウトが複数の配列は、アクセス順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ません。

インターフェイスの配列

Vivado® HLS では、配列がメモリ エlement にデフォルトで合成されます。配列を最上位関数への引数として使用した場合、Vivado HLS では次が想定されます。

- メモリはオフチップ。

Vivado HLS では、メモリにアクセスするためにインターフェイス ポートを合成。

- メモリはレイテンシ 1 の標準ブロック RAM。

データはアドレスの供給後 1 クロック サイクルで準備完了。

Vivado HLS でのこれらのポートの作成方法は、次のように設定します。

- INTERFACE 指示子を使用してインターフェイスを RAM または FIFO インターフェイスとして指定。
- RESOURCE 指示子を使用してシングル ポートまたはデュアル ポート RAM として RAM を指定。
- RESOURCE 指示子を使用して RAM レイテンシを指定。
- 配列最適化指示子 (Array_Partition、Array_Map または Array_Reshape) を使用すると、配列の構造を設定し直せ、I/O ポートの数も変更できます。



ヒント: データへのアクセスがメモリ (RAM または FIFO) ポートから制限されるため、インターフェイス上の配列がパフォーマンスの障害となることがあります。これらの障害は通常、指示子を使用することにより回避できます。

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、[配列インターフェイス](#)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.
```

配列インターフェイス

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 では次が使用されます。

- シングル ポート RAM (デフォルト)。
- 開始間隔やレイテンシが削減される場合はデュアル ポート RAM。

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。配列は複数の小型の配列に分割でき、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、配列のすべての要素をスカラー要素に分割する機能も含まれます。関数インターフェイスの場合は、配列のすべての要素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うことがあります。この場合、オフチップ ブロック RAM 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。

次のコード例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]];
    }
}
```

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-loop` により各クロック サイクルで読み出しおよび書き込みできるのは 1 要素のみであるため、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-loop` が展開されると、Vivado HLS ではデュアルポートが使用されます。これにより、同時に複数の要素を読み出すことができ、開始間隔を改善できます。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。

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、上記の例の配列が個々の要素に分割され、`for-loop` が展開されていれば、各配列の 4 つの要素すべてが同時にアクセスされます。

`RESOURCE` 指示子を使用しても、RAM のレイテンシを指定できます。これにより、Vivado HLS はインターフェイスで外部 SPRAM に 1 を超えるレイテンシを指定できるようになります。

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 ポートを使用する場合は、配列のアクセスがシーケンシャルであ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であるかどうか判断されます。

表 44: Vivado HLS のシーケンシャル アクセス解析

シーケンシャル アクセス	Vivado HLS の動作
○	FIFO ポートをインプリメントします。
×	<ol style="list-style-type: none"> エラー メッセージを表示します。 合成を停止します。
不定	<ol style="list-style-type: none"> 警告メッセージが表示されます。 FIFO ポートをインプリメントします。

注記: アクセスが実際にシーケンシャルではない場合、RTL シミュレーション不一致になります。

次のコード例では、Vivado HLS でアクセスがシーケンシャルかどうかを判断できません。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;
    #pragma HLS INTERFACE ap_fifo port=d_i
    #pragma HLS INTERFACE ap_fifo port=d_o
    // Breaks FIFO interface d_o[3] = d_i[2];
    For-Loop: for (i=0;i<4;i++) {
        d_o[i] = d_i[idx[i]];
    }
}
```

この場合、変数 `idx` の動作により FIFO インターフェイスが正しく作成できるかどうか決まります。

- `idx` がシーケンシャルに増加すれば、FIFO インターフェイスを作成できます。

- `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機能しない可能性があるため、Vivado HLS で合成中に次のようなメッセージが表示され、FIFO インターフェイスが作成されます。

```
@W [XFORM-124] Array 'd_i': may have improper streaming access(es).
```

上記の例で `//Breaks FIFO interface` コメントが削除されると、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上記の例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、FIFO インターフェイスではなくストリーミング インターフェイスでインプリメントされる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され、書き込まれる必要があります。これは FIFO リンクの特性と一致するポイントツーポイントの接続に変換される可能性があります。
- 配列の読み出しは、配列の書き込みと同じ順序で実行される必要があります。FIFO チャンネルではランダム アクセスがサポートされないため、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 動作には解析できないため、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。

配列の初期化



推奨: ザイリンクスでは、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勧めしています。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るほか、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` によっては演算は実行されません。

```
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。関数が実行されるたびに、`coeff` 配列には前の実行からの値が記録されます。C コードでは、static 配列はメモリが RTL で動作するように動作します。

```
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 ビットストリームの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、大容量メモリの初期化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後に `static` 変数をその初期ステートに戻す (デフォルトではない) 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。メモリをリセット後に初期ステートに戻す場合、これが演算上のオーバーヘッドとなり、値をリセットするのに複数サイクル必要となり、値を各メモリ アドレスに書き込む必要があります。

ROM のインプリメント

Vivado® HLS では、メモリを合成するために配列に `static` 修飾子を指定したり、メモリを ROM に推論するために `const` 修飾子を使用したりすることは必須ではありません。Vivado HLS は、デザインを解析して最適なハードウェアを作成を試みます。

ザイリンクス 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 でのデザインの解析では ROM を使用する必要があることが常に推論できるわけではないので、読み出しのみの配列には `const` 修飾子を使用することもお勧めします。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とインターリーブしないようにします。
- 異なる値を同じ配列要素に格納しないようにします (すべての書き込みをコード内でグループ化)。
- 要素値の計算は、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 (`math.h` ライブラリからの関数など)、配列初期化を別の関数に配置すると、ROM が推論されるようになります。次の例では、配列 `sin_table[256]` 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(dint_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] = (dint_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];
}
```



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 デザインでコアは必要ありません。

データ型

実行ファイルへコンパイルされる C 関数で使用するデータ型は、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` 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通常複数アクセスが必要になるので、実行時間に影響します。

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。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ので、RTL のエリアおよびパフォーマンスも決まります。

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移植できます。

C 規格では、整数型 `(unsigned)long` で 64 ビットが 64 ビット OS、32 ビットが 32 ビット OS としてインプリメントされます。合成ではこの動作に合わせて、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。Windows OS の場合、Microsoft 社により OS に関係なく `long` 型が 32 ビットに定義されます。

- 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` を使用します。
- 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` を使用します。

注記: C/C++ コンパイル オプションの `-m32` を使用すると、コードが C シミュレーション用にコンパイルされ、32 ビット アーキテクチャの仕様に合成でき、`long` 型が 32 ビット値でインプリメントされます。このオプションは、`add_files` コマンドに `-CFLAGS` オプションを使用して適用します。

ザイリンクスでは、すべての変数のデータ型を 1 つの共通のヘッダー ファイルで定義することをお勧めします。このファイルは、すべてのソース ファイルに含めることができます。

- 通常の Vivado HLS プロジェクト フロー中には、たとえばサイズを削減したり、ハードウェア インプリメンテーションをより効率的にできるようにするために、変更可能なデータ型があります。
- 抽象度の高いレベルで変更をしておく利点の 1 つは、新しいデザイン インプリメンテーションをすばやく作成できる点にあります。通常同じファイルが後のプロジェクトで使用されますが、別の (より小型、より大型、またはより正確な) データ型を使用することもできます。

これらのタスクはどちらも、データ型が 1 つの箇所に変更できる場合は、より簡単に達成できます。または、複数ファイルを編集します。



重要: ヘッダー ファイルでマクロを使用する場合は、常に独自の名前を使用してください。たとえば、`_TYPES_H` という名前のマクロがヘッダー ファイルで定義されている場合、よくある名前なのでほかのファイルで定義されている可能性があり、ほかのコードがイネーブルまたはディスエーブルになって、予期しない問題が発生することがあります。

標準データ型

次のコード例に、基本的ないくつかの算術演算の実行を示します。

```
#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;

}
```

上記の例のデータ型は、次のコード例に示すヘッダー ファイル `types_standard.h` で定義されています。これには、次のデータ型をどのように使用できるかが示されています。

- 標準符号付き型
- 符号なし型
- 固定長整数型 (`stdint.h` ヘッダー ファイルを含有)

```
#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);
```

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 ビット乗算器です。これは、8 ビットの `char` 型を 16 ビットの `short` で乗算するには、24 ビット乗算器が必要だからです。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 8 ビットです。出力が 8 ビットの unsigned char 型なので、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 に追加されます。
- out3 (32 ビットの固定幅型) 出力では、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。
- 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 の結果が示すように、HLS では可能な限り最小の演算子が使用され、必要な出力ビット幅に一致するように結果が拡張されます。結果 out2 では、入力の 1 つが 16 ビットですが、必要なのは 8 ビット出力なので、8 ビット加算器を使用できます。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 規格に従って合成されます。

- 単精度 32 ビット
 - 仮数部 24 ビット
 - 指数部 8 ビット
- 単精度 64 ビット
 - 仮数部 53 ビット
 - 指数部 11 ビット



推奨: 浮動小数点型を使用する場合、ザイリンクスでは『Vivado HLS を使用した浮動小数点デザイン』(XAPP599) を参照することをお勧めします。

float 型および double 型は、標準演算 (+、-、* など) だけでなく、math.h (C++ の場合は cmath.h) にもよく使用されます。このセクションでは、標準演算のサポートについて説明されます。

次に、標準データ型で使用されたヘッダー ファイルに double 型および float 型を定義したコード例を示します。

```
#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);
```

このアップデートされたヘッダー ファイルは、`sqrtf()` 関数が見られる次のコード例で使われます。

```
#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);
}
```

上記の例を合成すると、64 ビットの倍精度乗算、加算、除算演算子が作成前述の例されます。これらの演算子は、適切な浮動小数点のザイリンクス IP カタログ 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 ビットの単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()` が使われると、`inD` で使われる 32 ビットの単精度浮動単型の型変換のためにロジックが追加されます。`sqrtf()` は単精度 (`float`) 関数ですが、`out4: sqrt()` は倍精度 (`double`) 関数です。

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。

```
float foo_f      = 3.1459;
float var_f = sqrt(foo_f);
```

上記のコードは、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()` 関数を使えば、次のようになります。

- ハードウェアで型コンバーターが不要になる。
- エリアが節約される。
- タイミングが改善される。

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: Vivado HLS では float 型および double 型を合成する際に演算順序が厳しく守られるため、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る並列計算の利点を活かすことができない場合もあります。

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。

任意精度型

Vivado HLS では、float および double 型で説明するように、任意精度型が提供されています。

複合型

Vivado HLS では、合成で複合型がサポートされます。

- struct
- enum
- union

構造体 (struct)

構造体が最上位関数への引数として使用されると、合成で作成されるポートはその構造体のメンバーを直接反映したものになります。スカラー メンバーは標準的なスカラー ポートとして、配列はデフォルトでメモリ ポートとしてインプリメントされます。

このデザイン例の場合、次のコード例に示すように 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);
```

- 次のコード例では、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。

```
#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;
}
```

すべての関数引数および関数の戻り値は、次のようにポートに合成されます。

- Struct 要素 A は 16 ビット ポートになります。
- Struct 要素 B は 4 つの要素にアクセスする RAM ポートになります。

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。

構造体の要素は、データパック最適化により 1 つのベクターにパックできます。この最適化の実行に関する詳細は、`set_directive_data_pack` コマンドを参照してください。また、`config_interface` コマンドに `-trim_dangling_ports` オプションを使用すると、構造体の未使用の要素をインターフェイスから削除できます。

列挙型 (enum)

次のコード例のヘッダー ファイルでは、enum 型をいくつか定義し、それらを struct で使用しています。この struct は、別の struct で使用されます。これにより、複雑なデータ型がわかりやすくなります。

次のコード例は、複雑な定義文 (`MAD_NSBSAMPLES`) の指定および合成方法を示しています。

```
#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);
```

次の例では、前の例で定義された struct および enum 型が使用されています。enum が最上位関数への引数に使用されると、標準の C コンパイルの動作に準拠するため、32 ビット値として合成されます。enum 型がデザイン内部に対して指定される場合、Vivado HLS で必要なビット数まで最適化で減らされます。

次のコード例は、合成中に printf 文が無視されるように記述されています。

```
#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;
}

```

共用体 (union)

次のコード例では、double および struct を使用して共用体を作成しています。C コンパイラと異なり、union のすべてのフィールドに合成で必ず同じメモリ (合成の場合、レジスタ) が使用されるとは限りません。Vivado HLS では、最適なハードウェアを提供するため最適化が実行されます。

```

#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) & (0x7fffffffffffffffffLL);
}

```

Vivado HLS では、次はサポートされません。

- 最上位関数インターフェイスの共用体。
- 合成でのポインター再変換。このため、共用体には別のデータ型へのポインターまたは別のデータ型の配列へのポインターは保持できません。
- 別の変数を介した共用体へのアクセス。同じ共用体を前の例のように使用する場合、次はサポートされません。

```

for (int i = 0; i < 6; ++i)
if (i<3)
    A[i] = intfp.intval.a + B[i];
else
    A[i] = intfp.intval.b + B[i];
}

```


- ただし、次のように明示的に記述し直すことができます。

```
A[0] = intfp.intval.a + B[0];
A[1] = intfp.intval.a + B[1];
A[2] = intfp.intval.a + B[2];
A[3] = intfp.intval.b + B[3];
A[4] = intfp.intval.b + B[4];
A[5] = intfp.intval.b + B[5];
```

共用体の合成では、ネイティブ C 型とユーザー定義の型間の型変換はサポートされません。

Vivado HLS デザインでは、1 つのデータ型から別のデータ型へ生のビットを変換するために union が使用されることがよくあります。通常、この生ビットの変換は、最上位ポート インターフェイスで浮動小数点値を使用する場合に必要です。次にその例を示します。

```
typedef float T;
unsigned int value; // the "input" of the conversion
T myhalfvalue; // the "output" of the conversion
union
{
    unsigned int as_uint32;
    T as_floatingpoint;
} my_converter;
my_converter.as_uint32 = value;
myhalfvalue = my_converter.as_floatingpoint;
```

このタイプのコードは浮動小数点 C データ型では問題なく、変更すれば double 型でも問題ありません。half 型はクラスであり、共用体では使用できないので、typedef および int を short に変更することはできません。その代わりに、次のコードを使用できます。

```
typedef half T;
short value;
T myhalfvalue = static_cast<T>(value);
```

同様に、反対方向の変換では、value=static_cast<ap_uint<16>>(myhalfvalue) または static_cast<unsigned short>(myhalfvalue) を使用します。

```
ap_fixed<16,4> afix = 1.5;
ap_fixed<20,6> bfix = 1.25;
half ahlf = afix.to_half();
half bhlf = bfix.to_half();
```

また、ヘルパー クラスの fp_struct<half> を使用して、data() または to_int() を使用して変換することもできます。hls/utils/x_hls_utils.h ヘッダー ファイルを使用してください。

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、修飾子が関数の動作に影響を与え、最適化を実行してより最適なハードウェア デザインを作成できるからです。この例は、各修飾子の概要の後に示します。

揮発性 (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、読み出し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します。`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主に説明しています。

任意精度型では、算術演算での `volatile` 修飾子はサポートされません。`volatile` 修飾子を使用した任意精度型は、演算式で使用する前に、`volatile` 以外のデータ型に割り当てる必要があります。

関連情報

[揮発性データの理解](#)

スタティック型 (static)

関数内の `static` 型は、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ーは、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。

合成後にレジスタになるのは `static` 型だけではありません。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` 型でなくても、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値 0 を割り当てるか、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。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。

`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。

定数型 (const)

`const` 型は、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ありませんので、初期化する必要があります。ほとんどの `const` 変数は、通常 RTL デザインでは定数になります。Vivado HLS は定数伝搬を実行して、不必要なハードウェアを削除します。

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 で自動分割は実行されない)。`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

次のコード例は、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 でどのようにデザインが解析され、最適なインプリメンテーションが判断されるかを示しています。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];
}

```

前述の例の場合では、Vivado HLS で `lookup_table` 変数が最終 RTL でメモリ エlement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。

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。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。

次のコード例では、グローバル変数のデフォルトの合成ビヘイビアーが示され、3つのグローバル変数が使用されています。この例では配列が使用されますが、Vivado HLS ではすべてのタイプのグローバル変数タイプ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) {
    int i, lidx;

    // Move elements in the input array
    for (i=0; i<N; ++i) {
        lidx=i;
        if(lidx+idx>N-1)
            lidx=i-N;
        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;
    }
}

```

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 ポートとしては使用可能になりません。デフォルトでは、次のようになります。

- 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。

I/O ポートとしてのグローバル変数の使用

グローバル変数はデフォルトでは I/O ポートとしては使用可能になりませんが、`expose_global` オプションで使用可能にできます。インターフェイス コンフィギュレーションで `expose_global` オプションを使用すると、すべてのグローバル変数が RTL インターフェイスでポートとして使用可能になります。インターフェイス コンフィギュレーションは、次のいずれかから設定できます。

- [Solution Settings] → [General] または
- `config_interface Tcl` コマンド

グローバル変数がインターフェイス コンフィギュレーションにより使用可能になると、デザイン内でのみアクセスされるようなものも含め、デザインのグローバル変数がすべて I/O ポートとして使用可能になります。

また、グローバル変数がスタティック修飾子を使用して指定されると、I/O ポートには合成できません。

Vivado HLS では、合成でグローバル変数がサポートはされますが、サイリンクスではグローバル変数を広範囲に使用するコードはお勧めしていません。

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場合は、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 (読み出しまたは書き込み) される場合。
- ポインター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。

次のコード例に、複数のオブジェクトをポイントするポインターの合成サポートを示します。

```
#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];
}
```

Vivado HLS のポインター トゥ ポインターは合成ではサポートされますが、最上位インターフェイスでは (最上位関数への引数としては) サポートされません。ポインター トゥ ポインターを複数の関数で使用する場合、Vivado HLS でポインター トゥ ポインターを使用する関数すべてがインライン展開されます。複数の関数をインライン展開すると、実行時間が長くなります。

```
#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 pointer to 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);
}
```

ポインターの配列も合成できます。次のコード例では、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。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;
}
```

ポインタの型変換は、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。次のコード例では、`int` 型が `char` 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;
}
```

Vivado HLS では、一般的なデータ型間のポインタ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;
```

インターフェイスのポインター

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

次のコード例のような、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;
}
```

インターフェイスのポインターは、関数呼び出しごとに一度だけ読み出しまたは書き込みされます。テストベンチは次のコード例のようになります。

```
#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;
}

```

C 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。次の例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;
    }
}

```

次に、この例をサポートするテストベンチのコード例を示します。累積を実行するループが pointer_arith 関数内に含まれるようになったので、テストベンチにより配列 d[5] で指定されたアドレス空間に適切な値が指定されます。

```

#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;
}

```

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!

```

ポインター演算では、ポインター データは順序どおりにアクセスされません。ワイヤ、ハンドシェイク、または FIFO インターフェイスでは、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、制御信号により処理が許可されたときに読み出しおよび書き込みを実行します。

どちらの場合も、データは順序どおりに要素 0 から到着し、書き込まれる必要があります。ポインター演算を使用したインターフェイスの例では、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、テストベンチの配列 `d[5]` の 2 つ目の要素です。

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。Vivado HLS では、ワイヤ、ハンドシェイクまたは FIFO インターフェイスを使用するとこれはサポートされません。ポインター演算を使用したインターフェイスの例のコードは、`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 (読み出しまたは書き込み) 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、次の例のようにインターフェイスの配列を使用してコードを変更する必要があります。これは、RAM インターフェイス (`ap_memory`) を使用して合成できます。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、順序どおりでなくても実行できます。

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 データでのみ使用可能で、データのインデックスを 0 から順に作成する場合以外は、ポインター演算と共に使用することはできません。

```

#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;
}
}
```

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では、ポインターを使用して複数回アクセスする場合に、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と発生します。

- 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。
- 合成前に C を検証し、その意図する動作と C 記述が正しいことを確認します。

ザイリンクスでは、関数の引数に複数回アクセスする必要がある場合は、ストリームを使用してデザインを記述することを勧めます。このセクションで説明される問題が発生しないようにするため、ストリームを使用してください。次の表のデザインでは、[コード例](#)が使用されています。

表 45: サンプル デザインのシナリオ

サンプル デザイン	説明
pointer_stream_bad	同じ関数内でポインターに複数回アクセスする場合に volatile 修飾子が必要な理由。
pointer_stream_better	最上位インターフェイスにこのようなポインターが含まれるデザインでは、意図した動作が正しくモデリングされていることを確認するために、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ことを示します。

次のコード例では、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。アクセスは FIFO インターフェイスでインプリメントされ、データは最終的な RTL インプリメンテーションからストリーミングされることを意図しています。

```
#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;
}
```

次に、このデザインを検証するテストベンチのコード例を示します。

```
#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;
}
```

揮発性データの理解

マルチアクセス ポインター インターフェイス例のコードは、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 (またはハンドシェイク) 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、次のことが確実にになります。

- アップストリームのプロデューサー 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コンシューマー 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。

このコードが標準 C コンパイラでコンパイルされる場合、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。コンパイラについては、`d_i` 上のデータが関数の実行中に変化することが示されないの、関係あるのは `d_o` への最終の書き込みのみです。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 ポートが設計者の意図どおりにインプリメントされるように記述されていないことです。

- RTL ポートは、トランザクション中に複数回読み出しおよび書き込みされ、データ入力および出力をストリーミングすることが可能であるはずですが。

- テストベンチは1つの入力値だけを供給し、1つの出力値だけを返します。[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](#) のCシミュレーションの結果は次のようになり、各入力が4回累積されていることを示しています。同じ値が1回読み出され、毎回累積されており、4つの個別の読み出しが実行されるわけではありません。

```
Din Dout
0    0
1    4
2    8
3   12
```

- このデザインでRTLポートに対して読み出しおよび書き込みが複数回実行されるようにするには、`volatile` 修飾子を使用します。次にそのコード例を示します。

`volatile` 修飾子を使用すると、C/C コンパイラ (および Vivado HLS) でポインター アクセスに関して何も想定されず、データが揮発性であり変化する可能性があるとして解釈されます。



ヒント: ポインター アクセスは最適化しないでください。

```
#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;
}
```

上記の例は [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](#) と同じようにシミュレーションされますが、`volatile` 修飾子により次のようになります。

- ポインター アクセスが最適化されなくなります。
- 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回実行されるRTLデザインが得られます。

`volatile` キーワードを使用しても、このコーディング スタイル (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、読み出しは4回実行されますが、同じデータが4回読み出されています。書き込みは2回、それぞれ正しいデータで実行されますが、テストベンチでは最後の書き込みのデータのみが取り込まれます。

注記: 中間アクセスを表示するには、`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、適切なビューアーでそのトレース ファイルを確認します。

上記のマルチアクセス `volatile` ポインター インターフェイスの例は、ワイヤ インターフェイスを使用してインプリメントできます。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。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の検証はエラーになります。テストベンチでは、正しく読み出しおよび書き込みが記述されません。

ストリーミング データ インターフェイスの記述

ソフトウェアとは異なりハードウェア システムは並列性が高いので、ストリーミング データの利点を活かすことができます。データはデザインに連続して供給され、デザインから連続して出力されます。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。

揮発性データの理解 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、次のような複数の方法があります。

- マルチアクセス `volatile` ポインター インターフェイスの例に示すように、`volatile` 修飾子を追加します。テストベンチには固有の読み出しおよび書き込みが記述されていないので、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、トレース ファイルの波形を確認すると、正しい読み出しおよび書き込みが実行されています。
- 固有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。次に例を示します。
- 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。

次に、テストベンチから 4 つの異なる値を読み出し、2 つの異なる値を書き込むようにアップデートしたコードを示します。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 "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;
}
```

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記述するように、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、1 つのトランザクションのみを記述しています。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;
}
```

このテストベンチにより、次のような結果のアルゴリズムが検証されます。

- 1つのトランザクションから2つの出力が生成されます。
- 出力は最初の2つの入力読み出しが累算されたものと、次の2つの入力読み出しと最初の出力が累算されたものになります。

```
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

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回読み出しおよび書き込みが実行されたかを判断できません。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。

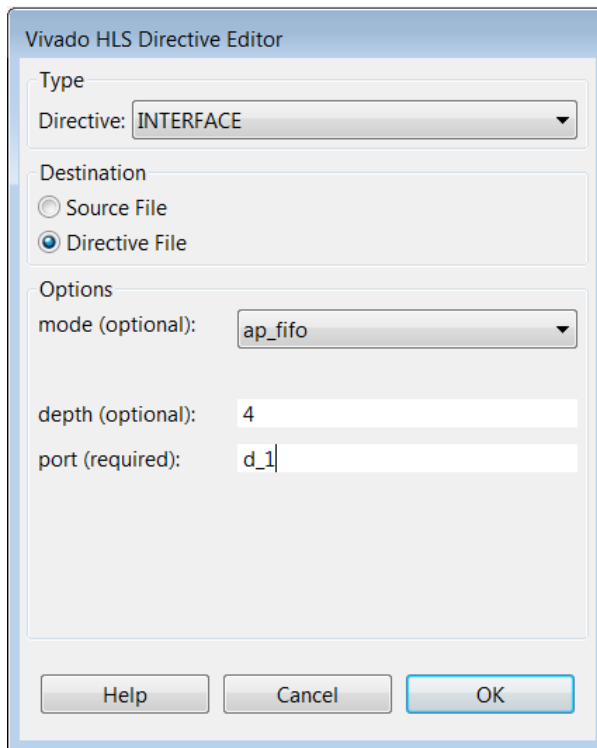
```
void pointer_stream_good (volatile dout_t *d_o, volatile din_t *d_i)
```

Vivado HLS に対して配列の最大サイズなど、インターフェイスで値がいくつ必要か示さないと、Vivado HLS では1つの値が想定されて、1つの入力および1つの出力のみで C/RTL 協調シミュレーションが作成されます。

RTL ポートが実際には複数の値の読み出しまたは書き込みを実行する場合、RTL 協調シミュレーションが停止します。RTL 協調シミュレーションでは、RTL デザインに接続されるプロデューサー ブロックとコンシューマー ブロックがモデル化されます。複数の値が必要な場合、複数の値の読み出しまたは書き込みが実行されると、読み出す値がないか書き込むスペースがないため、RTL デザインは停止します。

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また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、次の図に示すように [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 86: [Vivado HLS Directive Editor] ダイアログ ボックスの [depth] オプション



上記の例では、引数またはポート `d_1` が深さ 4 の FIFO インターフェイスになるように設定されています。これにより、RTL 協調シミュレーションで RTL を正しく検証するのに十分な値が供給されます。

C ビルトイン関数

Vivado HLS では、次の C ビルトイン関数がサポートされます。

- `__builtin_clz(unsigned int x)`: `x` の最上位ビットから数えた先行ゼロのビット数を返します。 `x` が 0 の場合は、結果は未定義になります。
- `__builtin_ctz(unsigned int x)`: `x` の再開ビットから数えた後置ゼロのビット数を返します。 `x` が 0 の場合は、結果は未定義になります。

次に、これらの関数の使用例を示します。この例は、in0 の先行ゼロの数と in1 の後置ゼロの数 0 の数の合計を返します。

```
int foo (int in0, int in1) {
    int ldz0 = __builtin_clz(in0);
    int ldz1 = __builtin_ctz(in1);
    return (ldz0 + ldz1);
}
```

ハードウェア効率の良い C コード

C コードを CPU 用にコンパイルすると、C コードがコンパイラにより変換され、CPU マシン命令のセットに最適化されます。多くの場合、この段階で開発者の仕事は終わりです。ただし、パフォーマンスを改善する必要がある場合は、次のいくつか、またはすべてを実行します。

- コンパイラで実行可能な追加の最適化があるかどうかを理解。
- プロセッサ アーキテクチャに関する理解を深め、アーキテクチャ特定の動作 (組み合わせ分岐の削減により命令パイプラインを改善するなど) を活かすようにコードを修正。
- 主な操作を並列で実行するための CPU 特有の組み込み機能を使用して C コードを変更 (例: Arm NEON 組み込みなど)。

同じ手法は、DSP または GPU 用に記述されたコードおよび FPGA を使用する場合にも適用できます (FPGA デバイスは単に別のターゲット)。

Vivado HLS で合成された C コードが FPGA で実行され、同じ機能が C シミュレーションとして提供されます。この段階で開発者の仕事が終わりになることもあります。

ただし、FPGA デバイスは並列性が高くパフォーマンスに優れており、演算がシーケンシャルに実行されるプロセッサよりもかなり高速に演算を実行できるので、C コードをインプリメントするために FPGA が選択されるのが一般的です。

ここでは、C コードが達成可能な結果に与える影響を理解し、最初の 3 つの項目の利点を最大限に活用するために C コードを変更する方法を説明します。

典型的なたたみ込み関数の C コード

ここでは画像に適用される標準的なたたみ込み関数を使用して、FPGA では可能であったパフォーマンスが C コードによりどのように劣化するかをお見せします。この例では、データに対してまず水平たたみ込みが実行された後、垂直たたみ込みが実行されます。画像のエッジのデータはたたみ込み範囲外にあるので、最後に境界周囲のデータが処理されます。

アルゴリズムでの処理は、次の順で実行されます。

```
template<typename T, int K>
static void convolution_orig(
    int width,
    int height,
    const T *src,
    T *dst,
    const T *hcoeff,
```



```
const T *vcoeff) {

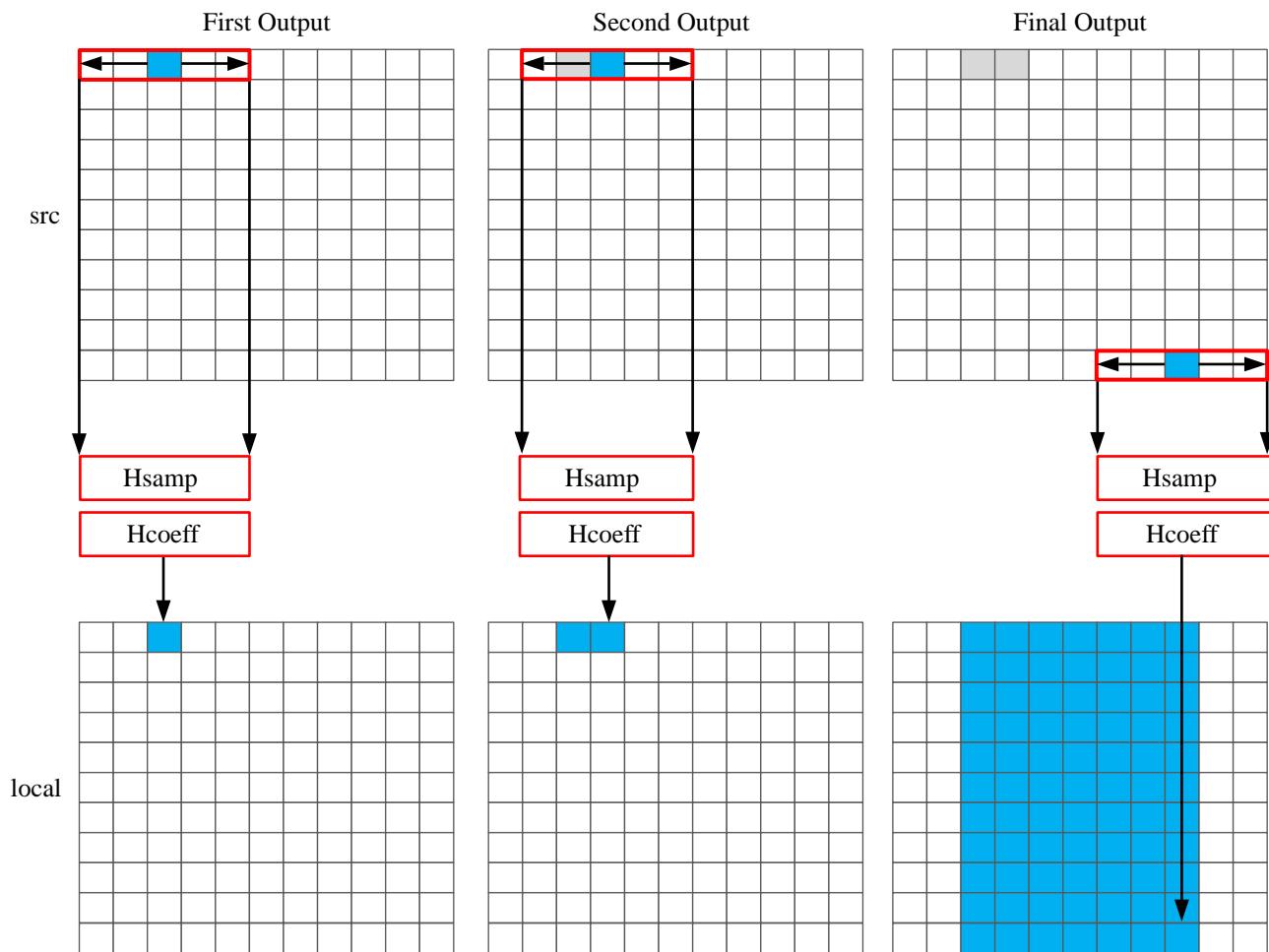
T local[MAX_IMG_ROWS*MAX_IMG_COLS];

// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
  HconvWfor(int row = border_width; row < width - border_width; row++){
    Hconv:for(int i = - border_width; i <= border_width; i++){
      }
    }
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
  VconvW:for(int row = 0; row < width; row++){
    Vconv:for(int i = - border_width; i <= border_width; i++){
      }
    }
// Border pixels
Top_Border:for(int col = 0; col < border_width; col++){
}
Side_Border:for(int col = border_width; col < height - border_width; col++){
}
Bottom_Border:for(int col = height - border_width; col < height; col++){
}
}
```

水平たたみ込み

最初に、次の図のように水平方向のたたみ込みを実行します。

図 87: 水平たたみ込み



X14296

たたみ込みは、K 個のデータ サンプルと K 個のたたみ込み係数を使用して実行されます。上の図では K の値は 5 ですが、この値はコードで定義されます。たたみ込みを実行するには、K 個以上のデータ サンプルが必要です。たたみ込みウィンドウは、画像外にあるピクセルを含む必要があるため、最初のピクセルでは開始できません。

対称たたみ込みを実行すると、src 入力からの最初の K 個のデータ サンプルが水平係数でたたみ込まれ、最初の出力が計算されます。2 つ目の出力を計算するには、次の K 個のデータ サンプルが使用されます。この計算は、最後の出力が書き込まれるまで各行に対して実行されます。

最終結果は、青色で示すようにより小さな画像になります。垂直境界沿いのピクセルは、後で処理されます。

この操作を実行する C コードは次のとおりです。

```
const int conv_size = K;
const int border_width = int(conv_size / 2);

#ifdef __SYNTHESIS__
    T * const local = new T[MAX_IMG_ROWS*MAX_IMG_COLS];
#else // Static storage allocation for HLS, dynamic otherwise
    T local[MAX_IMG_ROWS*MAX_IMG_COLS];

```

```
#endif

Clear_Local:for(int i = 0; i < height * width; i++){
    local[i]=0;
}
// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
    HconvWfor(int row = border_width; row < width - border_width; row++){
        int pixel = col * width + row;
        Hconv:for(int i = - border_width; i <= border_width; i++){
            local[pixel] += src[pixel + i] * hcoeff[i + border_width];
        }
    }
}
```

注記: `__SYNTHESIS__` マクロは、合成されるコードにのみ使用します。このマクロは C シミュレーションまたは C RTL 協調シミュレーションには従っていないので、テストベンチには使用しないでください。

このコードは簡単でわかりやすいものですが、この C コードにはいくつかの問題があり、3 つの問題はハードウェア結果の質に悪影響を及ぼします。

最初の問題は、ストレージ要件が 2 つ別々にあることです。結果は内部の `local` 配列に格納されます。これには、`HEIGHT*WIDTH` の配列が必要で、標準ビデオ画像の `1920*1080` の場合は `2,073,600` の値が保持されます。Windows システムによっては、この量のローカル ストレージにより問題が発生することがあります。local 配列のデータはスタックに配置され、OS で管理されるヒープに含まれません。

このような問題を回避するには、`__SYNTHESIS__` マクロを使用します。このマクロは、合成が実行されると自動的に定義されます。上記のコードでは、C シミュレーション中にダイナミック メモリ割り当てを使用してコンパイルの問題を回避しており、合成中はスタティック ストレージのみが使用されます。このマクロを使用する場合の欠点は、C シミュレーションで検証されたコードが合成されるコードとは異なるものになることです。この例の場合はコードは複雑ではないので、動作は同じになります。

FPGA インプリメンテーションの質の最初の問題は、`local` 配列にあります。これは配列なので、内部 FPGA ブロック RAM を使用してインプリメントされます。これは、FPGA 内にインプリメントするにはかなり大きいメモリであり、より大きくてコストのかかる FPGA デバイスが必要となる可能性があります。データフロー最適化を使用して、小型で効率的な FIFO を介してデータをストリーミングすると、ブロック RAM の使用を最小限に抑えることはできますが、データがストリーミングされるようにする必要があります。

次の問題は、`local` 配列の初期化です。`Clear_Local` ループは `local` 配列の値を 0 に設定するために使用されます。このループはパイプラインされていても、インプリメントには約 2 百万クロック サイクル (`HEIGHT*WIDTH`) が必要となります。これと同じデータの初期化は、`HConv` ループ内の一時的な変数を使用して、書き込み前に累積を初期化することにより実行できます。

最後の問題は、データのスループットがデータ アクセス パターンにより制限されることです。

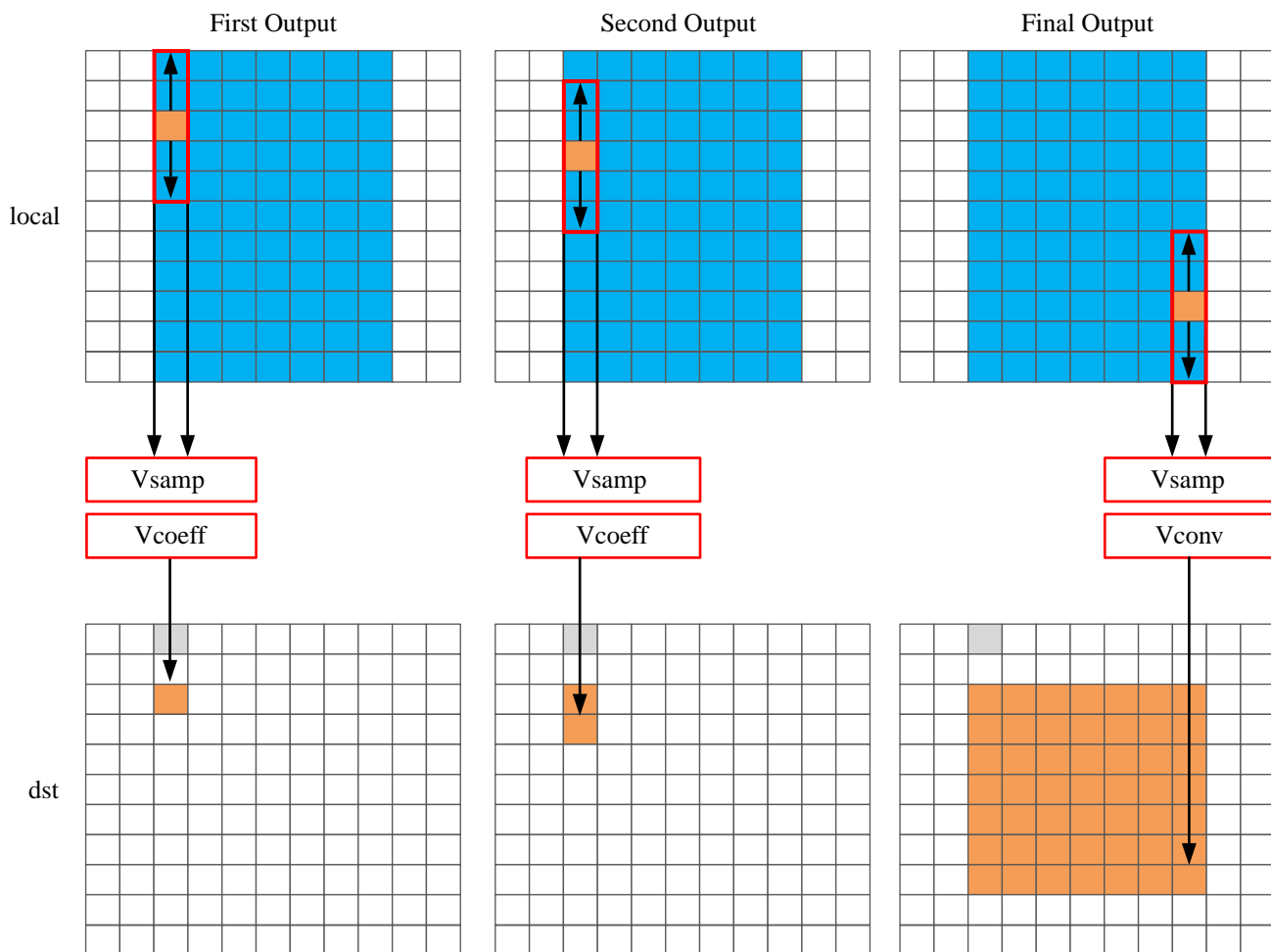
- 最初の出力を作成するため、最初の K 個の値が入力から読み出されます。
- 2 つ目の出力を計算するには、同じ K-1 値がデータ入力ポートを介して再度読み出されます。
- このデータの再読み出しプロセスは、画像全体で繰り返されます。

パフォーマンスの優れた FPGA にするには、最上位関数引数へのアクセスを最小限に抑えることが重要課題の 1 つとなります。最上位関数の引数は、RTL ブロックのデータ ポートになります。上記のコードでは、データを何度も読み出す必要があるため、DMA 操作を使用してプロセッサから直接ストリーミングできません。入力を再度読み出すと、FPGA がサンプルを処理するレートも制限されます。

垂直たたみ込み

次の段階では、次の図に示す垂直たたみ込みを実行します。

図 88: 垂直たたみ込み



X14299

垂直たたみ込みのプロセスは、水平たたみ込みと似ています。たたみ込み係数(この場合は `Vcoeff`)を使用したたたみ込みには、K 個のデータ サンプルが必要です。垂直方向の最初の K 個のサンプルを使用して最初の出力が作成された後、次の K 個の値を使用して 2 つ目の出力が作成されます。この処理は、最後の出力が作成されるまで各列に対して実行されます。

水平および垂直境界の効果により、垂直たたみ込み後の画像はソース画像 `src` よりも小さくなります。

これらの操作を実行するコードは、次のとおりです。

```
Clear_Dst:for(int i = 0; i < height * width; i++){
    dst[i]=0;
}
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
```

```
VconvW:for(int row = 0; row < width; row++){
    int pixel = col * width + row;
    Vconv:for(int i = - border_width; i <= border_width; i++){
        int offset = i * width;
        dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
    }
}
```

このコードには、水平たたみ込みコードを使用して既に説明した問題と同様の問題があります。

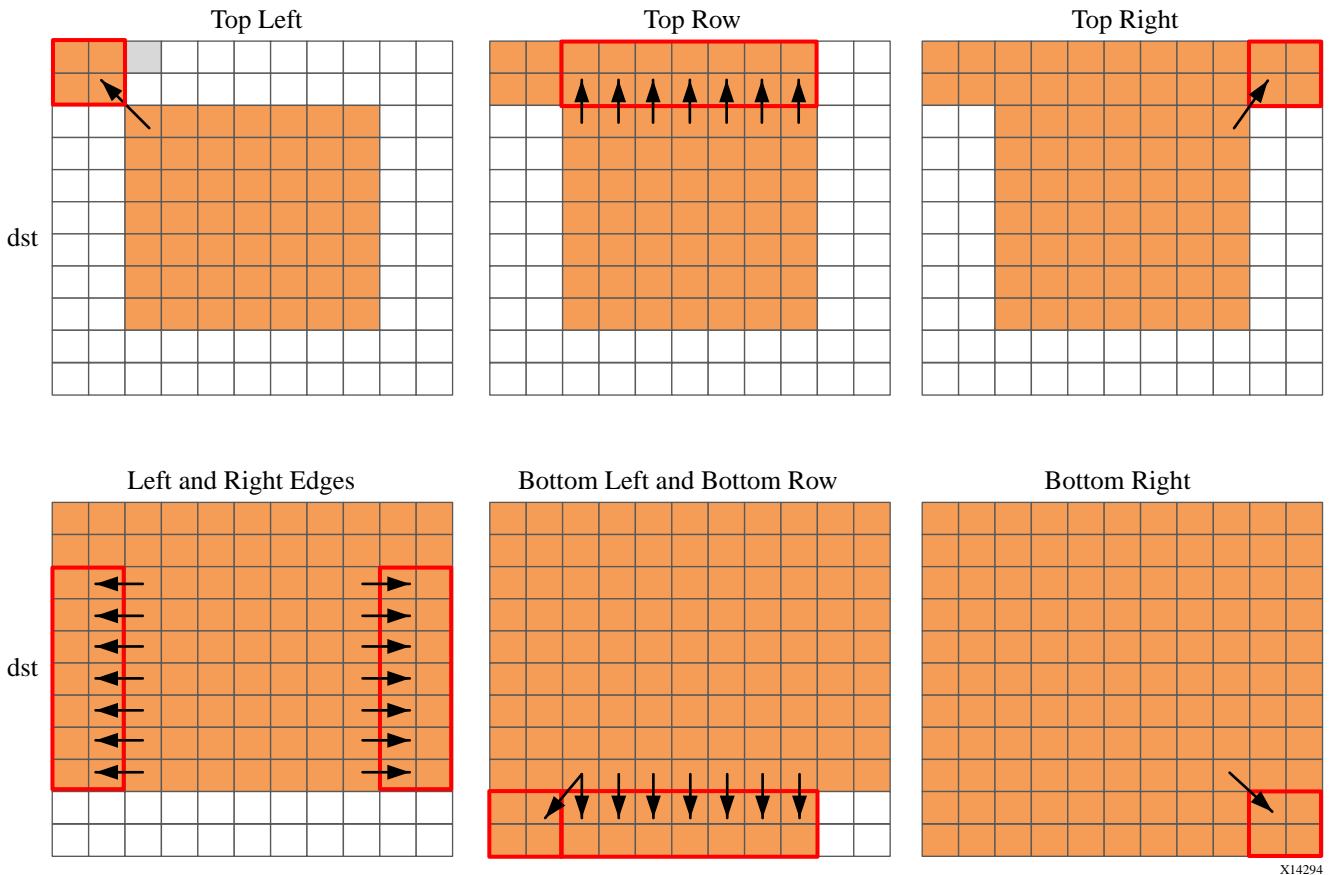
- 出力画像 `dst` の値を 0 に設定するのに、多数のクロック サイクルが費やされます。この場合、1920*1080 画像サイズに対してさらに約 2 百万サイクルが必要です。
- `local` 配列に格納されたデータを再度読み出すために、各ピクセルが複数回アクセスされます。
- 出力配列/ポート `dst` に対しても、各ピクセルが複数回書き込まれます。

上記のコードには、`local` 配列へのアクセス パターンの問題もあります。アルゴリズムでは、最初の計算を実行するために行 `K` のデータが使用可能になっていることが必要です。次の列に進む前に各行のデータを処理するため、画像全体がローカルに格納されている必要があります。また、データは `local` 配列外にはストリーミングされないのので、データフロー最適化で作成されたメモリ チャンネルをインプリメントするために FIFO を使用することはできません。このデザインにデータフロー最適化を使用すると、このメモリ チャンネルにピンポン バッファが必要で、インプリメンテーションに必要なメモリは倍の 4 百万データ サンプルになり、そのすべてを FPGA ローカルに格納する必要があります。

境界ピクセル

たたみ込みの最後の段階では、境界周辺のデータを作成します。これらのピクセルは、たたみ込み出力の最も近いピクセルを再利用することにより作成できます。次の図に、これをどのように達成するかを示します。

図 89: たたみ込み境界サンプル



境界領域は、最も近い有効な値を使用して作成されます。図に示す操作は、次のコードで実行されます。

```
int border_width_offset = border_width * width;
int border_height_offset = (height - border_width - 1) * width;
// Border pixels
Top_Border:for(int col = 0; col < border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + width - border_width - 1];
    }
}

Side_Border:for(int col = border_width; col < height - border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
```

```

    int pixel = offset + row;
    dst[pixel] = dst[offset + border_width];
}
for(int row = width - border_width; row < width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[offset + width - border_width - 1];
}
}

Bottom_Border:for(int col = height - border_width; col < height; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + width - border_width - 1];
    }
}
}

```

このコードには、データに繰り返しアクセスするという同じ問題があります。FPGA 外の dst 配列に格納されたデータは、入力データとして複数回読み出されることが可能になっている必要があります。最初のループでも、dst[border_width_offset + border_width] が複数回読み出されますが、border_width_offset および border_width の値は変更されません。

コーディング スタイルがパフォーマンスと FPGA インプリメンテーションの質に悪影響を与える問題の最後は、どのように異なる条件を指定するかはの構造です。for ループで各条件 (top-left、top-row など) に対して演算が処理される場合は、次のように最適化します。

この場合、変数境界 (width 入力の値に基づく) を使用するサブループがあるため、最上位ループ (Top_Border、Side_Border、Bottom_Border) はパイプライン処理できません。この場合、サブループをパイプライン処理し、パイプライン処理されたループの各セットを順に実行する必要があります。

最上位ループをパイプライン処理してサブループを展開するか、サブループを個別にパイプライン処理するのは、ループの制限と FPGA デバイスで使用可能なリソース数によって決まります。最上位ループの制限が小さい場合は、ループを展開してハードウェアを複製することによりパフォーマンスを満たします。最上位ループの制限が大きい場合は、サブループをパイプライン処理して、それらをループ (Top_Border、Side_Border、Bottom_Border) 内で順に実行することによりパフォーマンスを落とします。

この標準的なたたみ込みアルゴリズムの説明に示すように、次のコーディング スタイルを使用すると、FPGA インプリメンテーションのパフォーマンスおよびサイズに悪影響を及ぼします。

- 配列のデフォルト値を設定すると、クロック サイクル数が多くなり、パフォーマンスが低下します。
- データを複数回読み出すと、クロック サイクルが増加し、パフォーマンスも劣化します。
- 任意またはランダム アクセス方法でデータにアクセスする場合は、データを配列にローカルに格納する必要があります、リソースが費やされてしまいます。

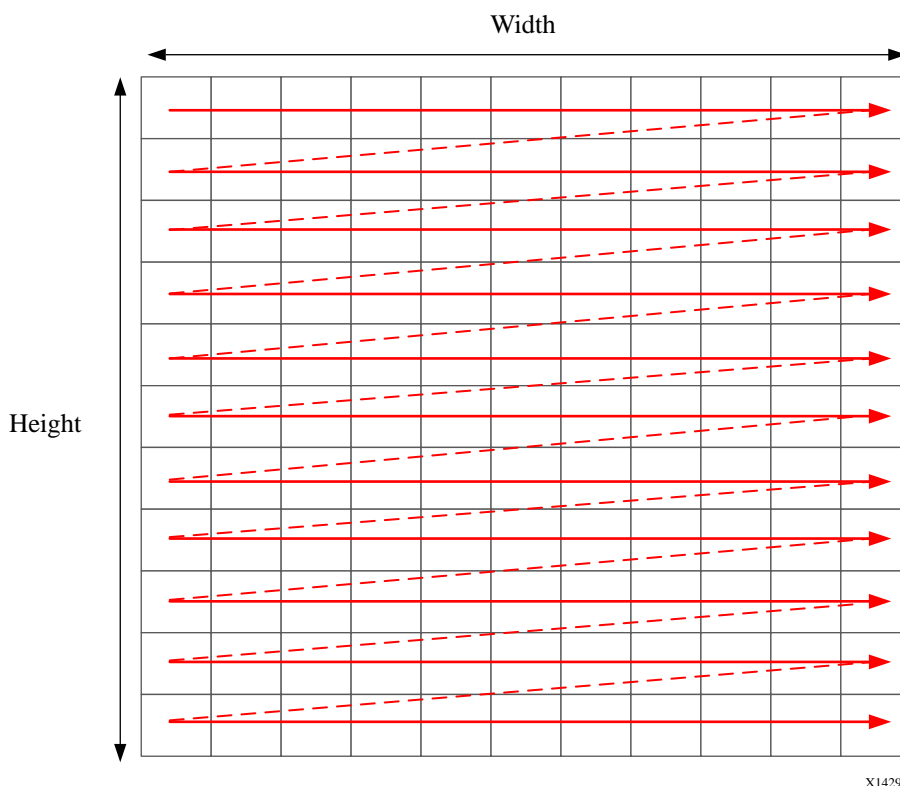
データおよびデータ再利用の続行フロー

前のセクションで説明したたたみ込みの例 (パフォーマンスの優れたリソース使用量が最小のデザイン) をインプリメントする際には、システム全体で FPGA インプリメンテーションがどのように使用されるのかについて考慮する必要があります。理想的なビヘイビアは、データ サンプルが FPGA 全体を一定して流れることです。

- システム中のデータフローを最大限にします。データフローを制限するようなコーディング手法やアルゴリズムビヘイビアは避けてください。
- データの再利用を最大限にします。ローカル キャッシュを使用して、同じデータを何回も読み出す必要がないようにし、入力データのフローが途切れないようにします。

最初の段階では、FPGA 内外に対する I/O 操作が最適になるようにします。たたみ込みアルゴリズムは、画像に対して実行されます。画像からのデータは、次の図に示すような標準のラスタ走査順序で転送されます。

図 90: ラスタ走査の順序



X14298

データが CPU またはシステム メモリから FPGA に転送されると、通常はこのストリーミング方法で転送されます。FPGA から転送されたデータをシステムに戻す場合も、この方法で実行する必要があります。

ストリーミング データに対する HLS ストリームの使用

前述のコードを改善する方法の 1 つは、通常 `hls::stream` と呼ばれる HLS ストリーム コンストラクトを使用することです。`hls::stream` オブジェクトは、配列と同じ方法でデータ サンプルを格納するために使用できます。`hls::stream` のデータには、順次アクセスしかできません。C コードでは、`hls::stream` が無限深さの FIFO のように動作します。

hls::streams を使用してコードを記述すると、hls::stream により FPGA でのインプリメンテーションに理想的なコーディングスタイルが使用されるので、通常 FPGA に高パフォーマンスでリソース使用量の少ないデザインが作成されます。

hls::stream から同じデータを何度も読み出すことはできません。データが hls::stream から読み出されると、データはストリームに存在しなくなります。これにより、このコーディング方法を削除できます。

hls::stream からのデータが再び必要な場合は、キャッシュに入力する必要があります。これも FPGA に合成されるようにコードを記述するのに推奨される方法の 1 つです。

hls::stream を使用すると、FPGA インプリメンテーションに理想的な方法で C コードが開発されるようになります。

hls::stream が合成されると、1 要素の深さの FIFO チャネルとして自動的にインプリメントされます。これは、パイプラインされたタスクの接続には理想的なハードウェアです。

hls::streams を必ず使用する必要はなく、同じインプリメンテーションは C コードの配列を使用しても実行できます。ただし、hls::stream コンストラクトを使用すると、優れたコードにようになります。

hls::stream を使用した場合の新しい最適化されたコードの概要は次のようになります。

```
template<typename T, int K>
static void convolution_strm(
    int width,
    int height,
    hls::stream<T> &src,
    hls::stream<T> &dst,
    const T *hcoeff,
    const T *vcoeff)
{

    hls::stream<T> hconv("hconv");
    hls::stream<T> vconv("vconv");
    // These assertions let HLS know the upper bounds of loops
    assert(height < MAX_IMG_ROWS);
    assert(width < MAX_IMG_COLS);
    assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

    // Horizontal convolution
    HConvH:for(int col = 0; col < height; col++) {
        HConvW:for(int row = 0; row < width; row++) {
            HConv:for(int i = 0; i < K; i++) {
            }
        }
    }
    // Vertical convolution
    VConvH:for(int col = 0; col < height; col++) {
        VConvW:for(int row = 0; row < vconv_xlim; row++) {
            VConv:for(int i = 0; i < K; i++) {
            }
        }
    }

    Border:for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
        }
    }
}
```

前述のコードと比較すると、このコードには明らかな違いがいくつかあります。

- 入力および出力データは hls::streams として記述されるようになりました。

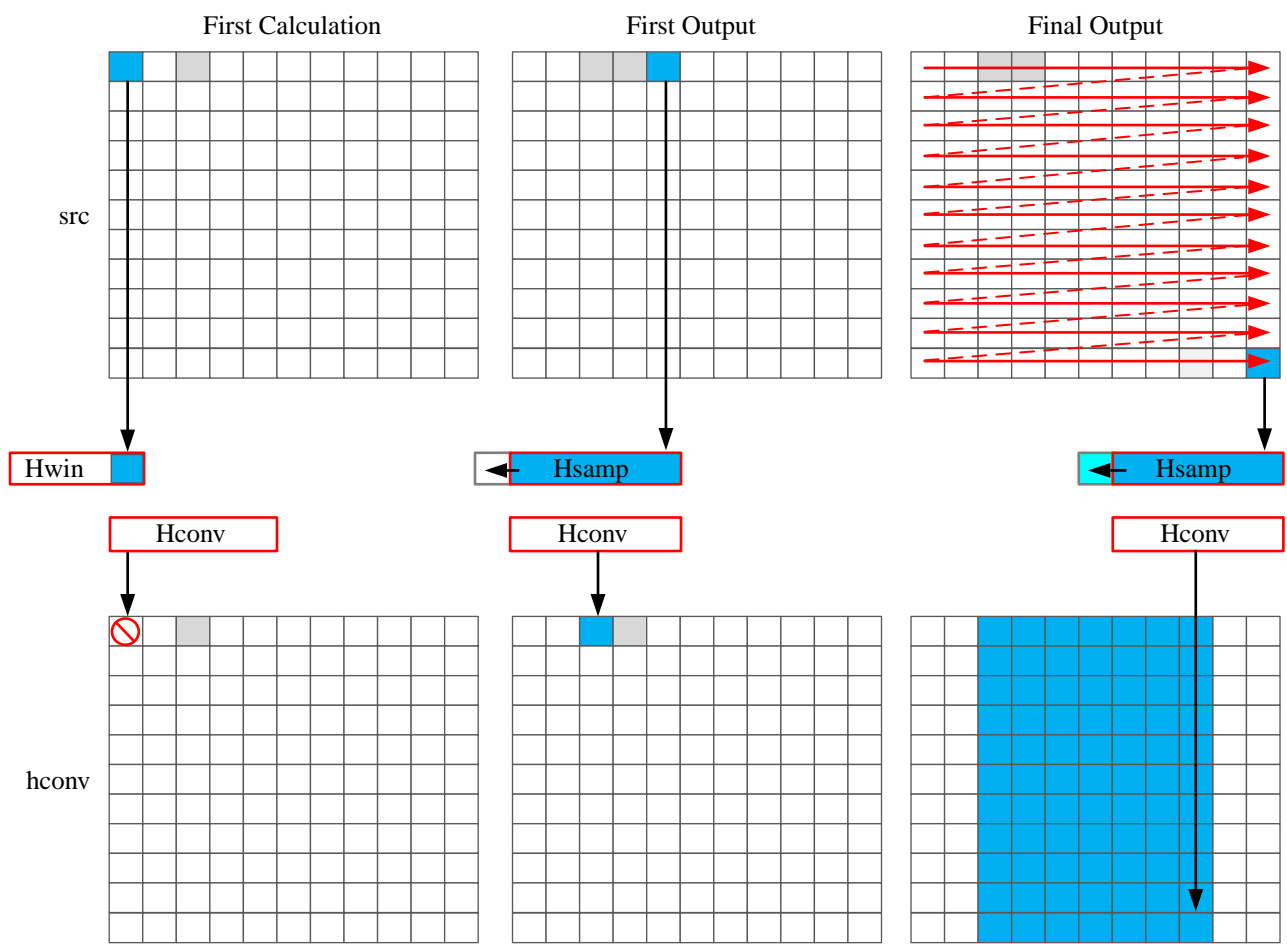
- HEIGHT*WDITH サイズの単一の local 配列の代わりに、2 つの内部 hls::streams が使用され、水平および垂直たたみ込みの出力が保存されています。

また、assert 文がいくつか使用され、ループ境界の最大値が指定されています。これにより、Vivado HLS で可変の境界付きループのレイテンシが自動的にレポートされ、そのループ境界が最適化されるようになるので、この方法はおすすめです。

水平たたみ込み

FPGA インプリメンテーションで効率的な方法で計算を実行するため、水平たたみ込みは次の図に示すように計算されます。

図 91: 水平たたみ込みのストリーミング



X14297

hls::stream を使用すると、データへのランダム アクセスを実行するのではなく、最初のサンプルを最初に読み出す優れたアルゴリズムになります。このアルゴリズムでは、前の K サンプルを使用してたたみ込み結果を計算する必要があるため、サンプルが一時キャッシュ hwin にコピーされます。最初の計算では、hwin に結果を計算するのに十分な値が含まれていないので、出力値は書き込まれません。

アルゴリズムは入力サンプルを継続的に読み出し、`hwin` キャッシュに格納します。新しいサンプルが読み出されるたびに、不要なサンプルが `hwin` から排出されます。K 番目の入力を読み込まれると、最初の出力値を書き込むことができるようになります。

このように、最後のサンプルが読み込まれるまで行ごとに処理されます。この段階で `hwin` に格納されているのは最後の K 個のサンプルだけであり、そのすべてがたたみ込み計算に必要となります。

これらの処理を実行するコードは、次のとおりです。

```
// Horizontal convolution
HConvW:for(int row = 0; row < width; row++) {
  HconvW:for(int row = border_width; row < width - border_width; row++){
    T in_val = src.read();
    T out_val = 0;
    HConv:for(int i = 0; i < K; i++) {
      hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
      out_val += hwin[i] * hcoeff[i];
    }
    if (row >= K - 1)
      hconv << out_val;
  }
}
```

上記のコードでは、一時変数 `out_val` を使用してたたみ込み計算が実行されています。この変数は、計算の実行前に 0 に設定されるので、前の例で示したように、値をリセットするために 2 百万クロック サイクルを費やす必要はありません。

プロセス全体を通して、`src` 入力のサンプルはラスターストリーミング方法で処理されます。すべてのサンプルが順番に読み込まれます。タスクからの出力は破棄または使用されますが、タスクは常に計算を実行し続けます。この点が、CPU で実行するために記述されたコードと異なります。

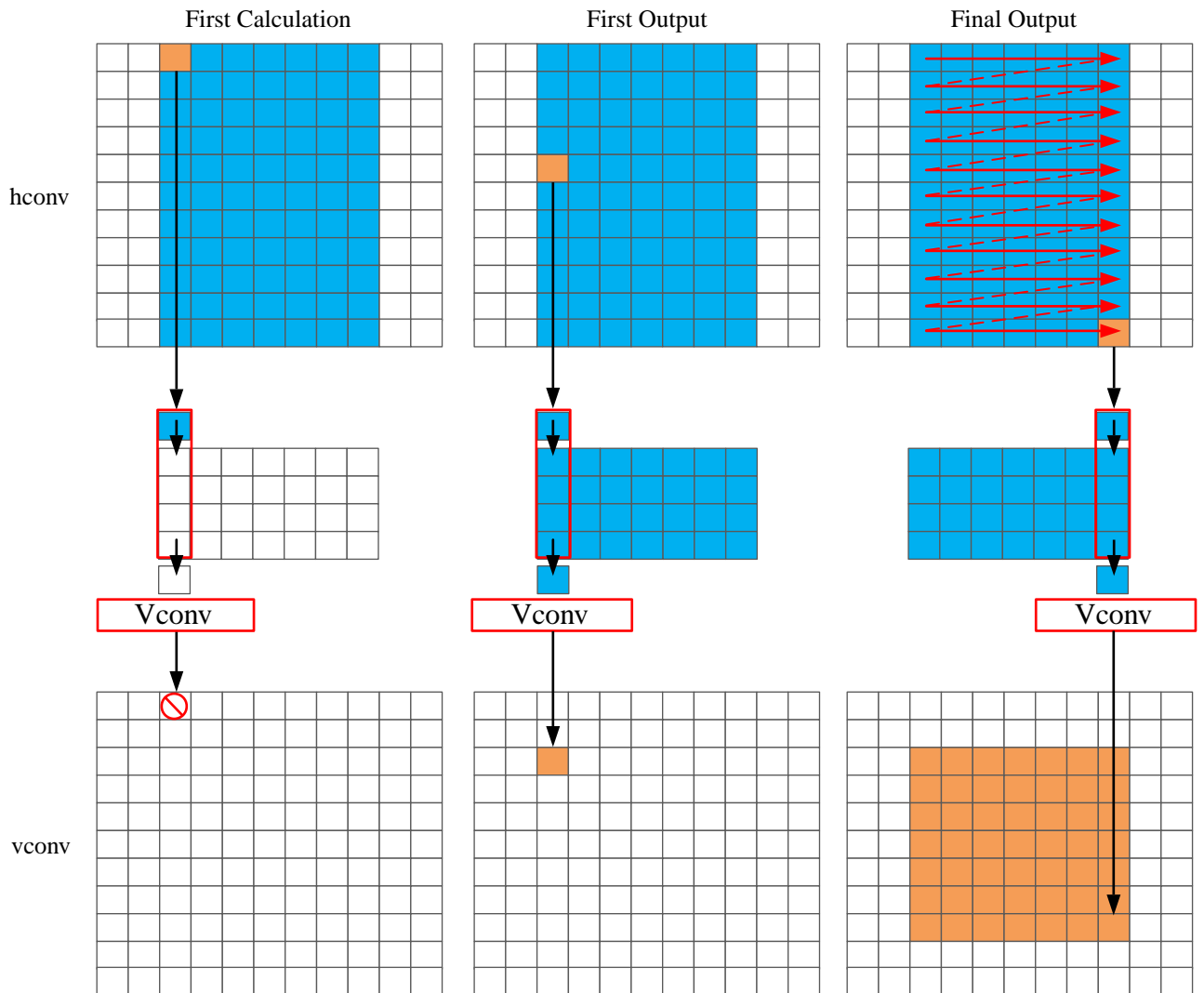
CPU アーキテクチャの場合、条件文または分岐文はよく回避されます。プログラムが分岐を必要とする場合は、CPU フェッチ パイプラインに格納された命令が失われます。FPGA アーキテクチャの場合、各条件分岐に対してハードウェアに別のパスが既に存在するので、パイプラインされたタスク内の分岐のためにパフォーマンスが落ちることはなく、単にどの分岐を使用するかといった問題だけです。

出力は、垂直たたみ込みループで使用するため、`hls::stream` の `hconv` に格納されます。

垂直たたみ込み

垂直たたみ込みでは、FPGA 向けのストリーミング データ モデルを記述するのが困難です。データには列ごとにアクセスする必要がありますが、画像全体を格納するのは望ましくありません。ソリューションは、次の図に示すようにラインバッファを使用することです。

図 92: 垂直たたみ込みのストリーミング



X14300

先ほどと同様、サンプルはストリーミング方式で読み出されますが、この場合は `hls::stream` の `hconv` から読み出されます。このアルゴリズムでは、最初のサンプルを処理するのに少なくとも $K-1$ 行のデータが必要です。これより前に実行された計算はすべて削除されます。

ラインバッファには、 $K-1$ 行のデータを格納できます。新しいサンプルが読み込まれるたびに、別のサンプルがラインバッファから排出されます。つまり、最新のサンプルが計算に使用されると、そのサンプルがラインバッファに格納され、古いサンプルが排出されます。この結果、キャッシュされる必要があるのは K 行ではなく、 $K-1$ 行のみになります。ラインバッファにはローカルで格納するために複数行が必要ですが、たたみ込みのカーネルサイズ K はフルビデオ画像の 1080 行よりもかなり小さくなります。

最初の計算は、K 行目にある最初のサンプルが読み込まれると実行されます。その後、最後のピクセルが読み込まれるまで値が出力されます。

```
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
  VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS DEPENDENCE variable=linebuf inter false
#pragma HLS PIPELINE
    T in_val = hconv.read();
    T out_val = 0;
    VConv:for(int i = 0; i < K; i++) {
      T vwin_val = i < K - 1 ? linebuf[i][row] : in_val;
      out_val += vwin_val * vcoeff[i];
      if (i > 0)
        linebuf[i - 1][row] = vwin_val;
    }
    if (col >= K - 1)
      vconv << out_val;
  }
}
```

上記のコードでは、デザインのサンプルがすべてストリーミング方式で処理されます。タスクは、継続して実行されます。hls::stream コンストラクトを使用すると、データがローカルにキャッシュされます。これは、FPGA をターゲットにする場合の理想的なストラテジです。

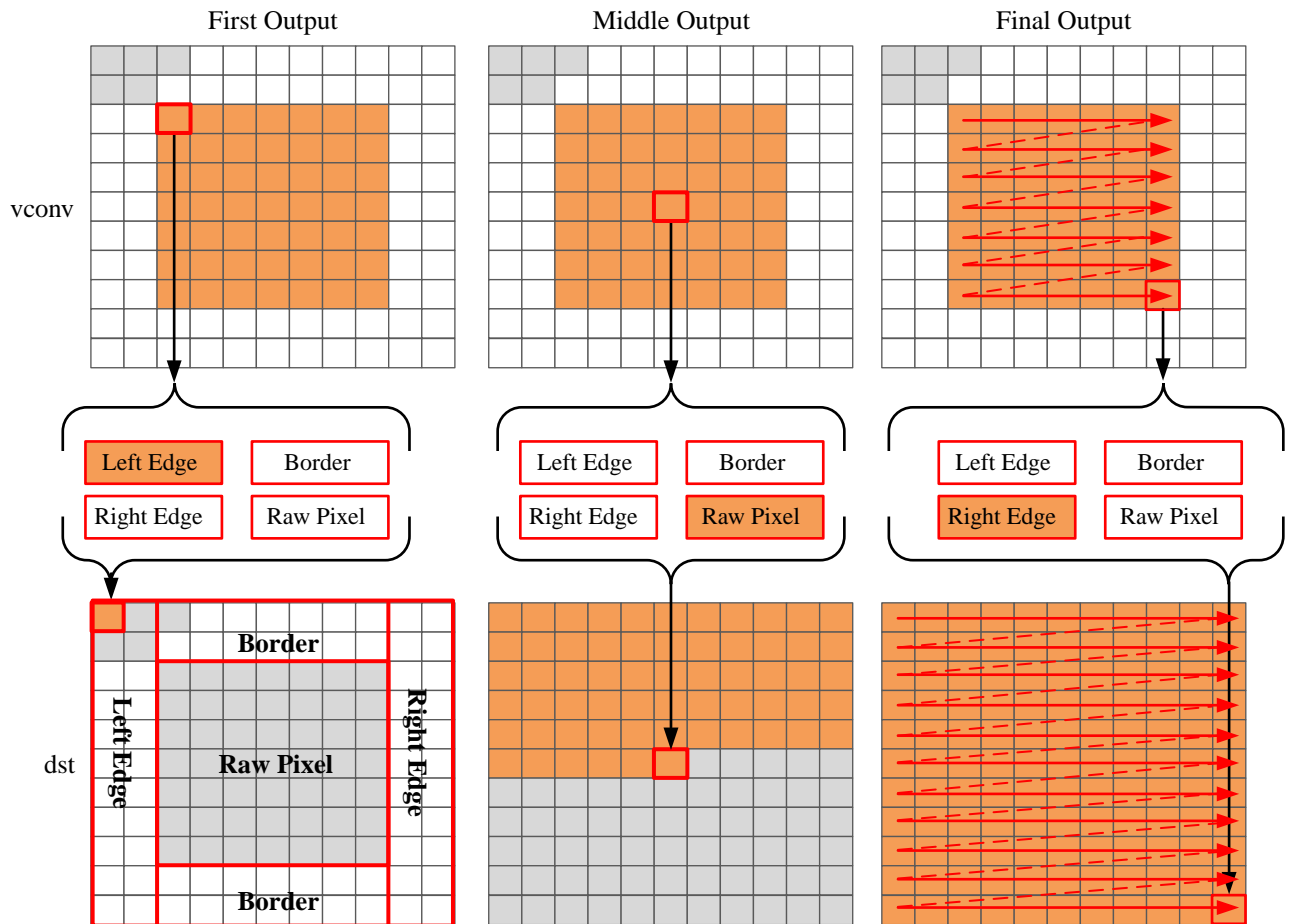
境界ピクセル

このアルゴリズムの最後には、エッジ ピクセルを境界領域に複製します。一定したフローやデータ/データ再利用を実現するには、アルゴリズムで hls::stream とキャッシュが使用されるようにします。

次の図に、境界サンプルがどのように画像に組み込まれるかを示します。

- 各サンプルが垂直たたみ込みからの vconv 出力から読み込まれます。
- 次に、サンプルが4つのピクセル タイプのいずれかとしてキャッシュに格納されます。
- サンプルが出力ストリームに書き出されます。

図 93: 境界サンプルのストリーミング



X14295

次は、境界ピクセルの位置を決定するコードです。

```
Border:for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
    T pix_in, l_edge_pix, r_edge_pix, pix_out;
    #pragma HLS PIPELINE
    if (i == 0 || (i > border_width && i < height - border_width)) {
      if (j < width - (K - 1)) {
        pix_in = vconv.read();
        borderbuf[j] = pix_in;
      }
      if (j == 0) {
        l_edge_pix = pix_in;
      }
      if (j == width - K) {
        r_edge_pix = pix_in;
      }
    }
    if (j <= border_width) {
      pix_out = l_edge_pix;
    } else if (j >= width - border_width - 1) {
      pix_out = r_edge_pix;
    }
  }
}
```

```

    } else {
        pix_out = borderbuf[j - border_width];
    }
    dst << pix_out;
}
}
}
}

```

このコードの明らかな違いは、タスク内に条件文が多く使用されている点です。これにより、タスクはパイプラインされたら、データを続けて処理し、条件文の結果によりパイプラインの実行が影響を受けることはありません。結果は出力値には影響しますが、パイプラインは入力サンプルが使用できる限り、処理され続けます。

この FPGA に最適なアルゴリズムを含む最終的なコードには、次の最適化指示子を使用されます。

```

template<typename T, int K>
static void convolution_strm(
int width,
int height,
hls::stream<T> &src,
hls::stream<T> &dst,
const T *hcoeff,
const T *vcoeff)
{
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITION variable=linebuf dim=1 complete

hls::stream<T> hconv("hconv");
hls::stream<T> vconv("vconv");
// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
    HConvW:for(int row = 0; row < width; row++) {
#pragma HLS PIPELINE
        HConv:for(int i = 0; i < K; i++) {
        }
    }
}
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS PIPELINE
#pragma HLS DEPENDENCE variable=linebuf inter false
        VConv:for(int i = 0; i < K; i++) {
        }
    }
}

Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
#pragma HLS PIPELINE
    }
}
}

```

各タスクは、同レベルでパイプラインされます。ラインバッファはレジスタに完全に分割され、ブロック RAM ポートの不足によって読み出しまたは書き込みが制限されることはありません。ラインバッファには、依存指示子も必要です。すべてのタスクがデータフロー領域で実行され、タスクが同時処理されるようになります。hls::streams は 1 要素を含む FIFO として自動的にインプリメントされます。

まとめ: 効率的なハードウェアのための C 言語

データ入力の読み込みを最小限にします。データがブロックに読み込まれると、多くの並列パスに簡単に供給できますが、入力ポートがパフォーマンスのボトルネックになることがあります。データを読み込んだ後、再利用する必要がある場合は、ローカル キャッシュを使用します。

配列、特に大型の配列へのアクセスを最小限に抑えます。配列はブロック RAM にインプリメントされますが、ブロック RAM では I/O ポートと同様ポート数が限られるので、パフォーマンスのボトルネックになることがあります。配列は小型の配列および個別のレジスタに分割できますが、大型の配列を分割すると使用されるレジスタ数が多くなります。小型のローカル キャッシュを使用して累積などの結果を保持してから、最終結果を配列に書き出すようにします。

パイプライン処理されたタスクであっても、タスクを条件で実行するのではなく、パイプライン処理されたタスク内で条件分岐を実行するようにします。条件文は、パイプラインで別々のパスとしてインプリメントされます。データが 1 つのタスクから次のタスク内で実行される条件に流れるようにすると、システムのパフォーマンスが向上します。

入力の読み出しと同様に、ポートはボトルネックになるので、出力の書き込みも最低限にします。追加ポートを複製すると、単に問題がシステムに先送りになるだけです。

ストリーミング方式でデータを処理する C コードの場合、優れたコード記述になるので、hls::streams を使用することをお勧めします。なぜ FPGA が必要なパフォーマンスで動作しないのかデバッグするよりも、優れたパフォーマンスの FPGA インプリメンテーションになる C のアルゴリズムを設計する方が生産的です。

C++ クラスおよびテンプレート

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。合成の最上位は、関数である必要があります。クラスは、合成では最上位にできません。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スエートする必要があります。最上位クラスは、単にテストベンチにインスタンスエートしないようにしてください。次のコード例は、CFir クラス (次で説明するヘッダー ファイルで定義) がどのように最上位関数 cpp_FIR にインスタンスエートされ、FIR フィルターのインプリメントに使用されるかを示しています。

```
#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);
}
```



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スエートする必要があります。

上記の C++ FIR フィルターの例で、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調べる前に、Vivado HLS では合成中に標準出力ストリームの `cout` が無視され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。

```
INFO [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
INFO [SYNCHK-101] Discarding unsynthesizable system call:
'std::ostream::operator<<' (cpp_FIR.h:108)
INFO [SYNCHK-101] Discarding unsynthesizable system call: 'std::operator<<
<std::char_traits<char> >' (cpp_FIR.h:110)
```

次のコード例に示す `cpp_FIR.h` ヘッダー ファイルには、`CFir` クラスの定義およびそれに関連するメンバー関数が含まれています。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。どちらも `main` アルゴリズムを実行するために使用され、`cout` と共に使用すると 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.h"
};

// 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);

```

次のコード例に示される C++ FIR フィルターのテストベンチは、最上位関数 `cpp_FIR` がどのように呼び出されて検証されるか示しています。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。
- 結果が正しいと確認されれば、テストベンチは 0 を返します。

```

#include "cpp_FIR.h"

int main() {
    ofstream result;
    data_t output;
    int retval=0;

    // Open a file to saves the results
    result.open(result.dat);

    // Apply stimuli, call the top-level function and saves 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;
}

```

cpp_FIR の C++ テストベンチ

指示子をクラスで定義したオブジェクトに適用するには、次の手順に従ってください。

1. クラスが定義されているファイル (通常はヘッダー ファイル) を開きます。
2. [Directive] タブを使用して指示子を適用します。

関数と同様、1 つのクラスのインスタンスはすべて 同じ最適化が適用されます。

グローバル変数およびクラス

ザイリンクスでは、クラスでグローバル変数を使用することは推奨していません。使用すると、一部の最適化が実行されないことがあります。次のコード例では、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++;
}

```

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

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。Vivado では、最上位関数のテンプレート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。

テンプレートを使用した固有のインスタンスの作成

テンプレート関数のスタティック変数は、テンプレート引数の異なる値に対してそれぞれ複製されます。

```
template<int NC, int K>
void startK(int* dout) {
    static int acc=0;
    acc += K;
    *dout = acc;
}

void foo(int* dout) {
    startK<0,1> (dout);
}

void goo(int* dout) {
    startK<1,1> (dout);
}

int main() {
    int dout0,dout1;
    for (int i=0;i<10;i++) {
        foo(&dout0);
        goo(&dout1);
        cout <<"dout0/1 = "<<dout0<<" / "<<dout1<<endl;
    }
    return 0;
}
```

再帰関数でのテンプレートの使用

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。

次のコード例では、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);
}
```

アサート

C の assert マクロは、範囲情報をアサートする場合に合成でサポートされます。たとえば、変数とループ境界の上限を指定できます。

ループ境界が可変である場合、Vivado HLS ではそのループの反復すべてのレイテンシを判断できず、レイテンシがクエスチョン マーク (?) で表示されます。tripcount 指示子は Vivado HLS にループ境界の情報を渡すためには使用できませんが、この情報はレポート目的にのみ使用され、合成結果には影響しません (tripcount 指示子の有無に関係なく同じサイズのハードウェアが作成されます)。

次のコード例は、assert を使用して Vivado HLS に変数の最大範囲を伝え、さらに適したハードウェアを作成するためにどのように assert を使用するかを示しています。

assert を使用する前に、assert マクロを定義するヘッダー ファイルを含める必要があります。この例では、これはヘッダー ファイルに含まれます。

```
#ifndef _loop_sequential_assert_H_
#define _loop_sequential_assert_H_

#include <stdio.h>
#include <assert.h>
#include ap_cint.h
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint8 dsel_t;

void loop_sequential_assert(din_t A[N], din_t B[N], dout_t X[N], dout_t
Y[N], dsel_t
xlimit, dsel_t ylimit);

#endif
```

main コードでは、各ループの前に次の 2 つの assert 文を記述します。

```
assert(xlimit<32);
...
assert(ylimit<16);
...
```

これらのアサートは、次を実行します。

- アサートが偽で値が指定の値よりも大きい場合、C シミュレーションでエラーが発生します。このため、合成前に C コードをシミュレーションすることが重要です。合成前にデザインが有効であることを確認してください。
- この変数の範囲がこの値を超えないことを Vivado HLS に伝えます。この情報は、RTL の変数のサイズ (この場合はループの反復回数) を最適化するために使用できます。

次のコード例に、これらの assert 文を示します。

```
#include "loop_sequential_assert.h"

void loop_sequential_assert(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;

    assert(xlimit<32);
    SUM_X:for (i=0;i<=xlimit; i++) {
        X_accum += A[i];
        X[i] = X_accum;
    }

    assert(ylimit<16);
    SUM_Y:for (i=0;i<=ylimit; i++) {
        Y_accum += B[i];
        Y[i] = Y_accum;
    }
}
```

このコードは assert マクロを除けば [ループの並列処理](#) と同じですが、合成後の合成レポートには 2 つの重要な相違点があります。

assert マクロを使用しない場合、レポートは次のようになり、ループ境界の変数が 8 ビット変数である d_sel データ型であるため、ループのトリップカウントは 1 ~ 256 の間で変化する可能性があります。

```
* Loop Latency:
+-----+-----+-----+
|Target II |Trip Count |Pipelined |
+-----+-----+-----+
|- SUM_X   |1 ~ 256   |no        |
|- SUM_Y   |1 ~ 256   |no        |
+-----+-----+-----+
```

assert マクロを使用した場合、レポートには SUM_X および SUM_Y ループのトリップカウントが 32 および 16 であることが示されます。assert により値が 32 および 16 を超えることはないので、Vivado HLS でレポートにこれが使用されます。

```
* Loop Latency:
+-----+-----+-----+
|Target II |Trip Count |Pipelined |
+-----+-----+-----+
|- SUM_X   |1 ~ 32    |no        |
|- SUM_Y   |1 ~ 16    |no        |
+-----+-----+-----+
```

また、tripcount 指示子を使用する場合と異なり、assert 文を使用すると最適なハードウェアを提供できます。assert を使用しない場合、最終的なハードウェアでループの最大反復回数 256 に合わせた変数およびカウンタが使用されます。

```
* Expression:
```

Operation	Variable Name	DSP48E	FF	LUT
+	X_accum_1_fu_182_p2	10	10	13
+	Y_accum_1_fu_209_p2	10	10	13
+	indvar_next6_fu_158_p2	10	10	19
+	indvar_next_fu_194_p2	10	10	19
+	tmp1_fu_172_p2	10	10	19
+	tmp_fu_147_p2	10	10	19
icmp	exitcond1_fu_189_p2	10	10	19
icmp	exitcond_fu_153_p2	10	10	19
Total		10	10	180

assert を使用して変数範囲を限定したコードでは、変数範囲を最大値よりも小さくできるので、より小型の RTL デザインが得られます。

```
* Expression:
```

Operation	Variable Name	DSP48E	FF	LUT
+	X_accum_1_fu_176_p2	10	10	13
+	Y_accum_1_fu_207_p2	10	10	13
+	i_2_fu_158_p2	10	10	16
+	i_3_fu_192_p2	10	10	15
icmp	tmp_2_fu_153_p2	10	10	17
icmp	tmp_9_fu_187_p2	10	10	16
Total		10	10	150

アサートでは、デザインに含まれるどの変数の範囲でも指定できます。アサートを使用する場合は、可能なすべての状況を想定した C シミュレーションを実行することが重要になります。これにより、Vivado HLS で使用されるアサートが有効であることを確認できます。

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。ライブラリは、Accellera ウェブサイト (www.accellera.org) から入手できます。Vivado HLS では SystemC Synthesizable Subset (Draft 1.3) のために SystemC バージョン 2.1 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ったもので、ザイリンクスでは、合成の基本的なコード規則を理解するために、これらの章を理解しておくことをお勧めしています。



重要: C および C++ デザインの場合と同様、合成では最上位関数が 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、SC_THREAD および SC_HAS_PROCESS を使用して記述される場合、または SC_MODULES がほかの SC_MODULES 内にインスタンス化される場合、デザインは合成できます。

デザインの最上位の SC_MODULE はテンプレートにはできません。テンプレートはサブモジュールでのみ使用できます。

モジュールのコンストラクターでは、モジュールの定義またはインスタンス化しかできず、機能を含めることができません。

SC_MODULE は別の SC_MODULE 内には定義できません。ただし、これらは後に示す方法で定義できます。

SC_MODULE の使用

階層モジュールの定義はサポートされません。モジュールが別のモジュール内で定義される場合 (1 つ目の SC_MODULE の例)、モジュールが入れ子にならないバージョン (2 つ目の SC_MODULE の例) に変換する必要があります。

```
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);
    }
};
```

```
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);
    }
};

```

また、次の例に示すように、SC_MODULE は別の SC_MODULE から派生できません。

```

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;
    â€¦
};

```



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1つ目の SC_MODULE の例のような場合は、2つ目の SC_MODULE の例に示すように変換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);
    ...
};

```

```
dut::dut(sc_module_name nm)
{
    SC_METHOD(process);
    sensitive<<in0;
}
```

```
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;
    }
    ~(){}
};
```

Vivado HLS では、合成で SC_THREAD がサポートされません。

SC_METHOD の使用

次の例は、半加算器を記述するための、SC_METHOD を使用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 c_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;
    }
};
```

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。次のコード例に示すように、half_adder 関数は sc_combo_method.cpp ファイルで指定され、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);

#ifdef __SYNTHESIS__
cout << Sum is << a << ^ << b << = << s << : <<
sc_time_stamp() <<endl;
cout << Car is << a << & << b << = << c << : <<
sc_time_stamp() <<endl;
#endif
```

注記: 上記の例では、`__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。

`__SYNTHESIS__` マクロは、合成されるコードにのみ使用します。このマクロは C シミュレーションまたは C RTL 協調シミュレーションには従っていないので、テストベンチには使用しないでください。

次のコード例は、前の例のテストベンチを示しています。このテストベンチには、Vivado HLS を使用する場合には必要な重要属性が複数含まれています。

```
#ifndef __RTL_SIMULATION__
#include "sc_combo_method_rtl_wrapper.h"
#define sc_combo_method sc_combo_method_RTL_wrapper
#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;
};
```

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、テストベンチに上記の例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。

```
#ifdef __RTL_SIMULATION__
#include "DUT_rtl_wrapper.h"
#define DUT DUT_RTL_wrapper
#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追加する必要があります。追加しないと、`cosim_design` の RTL シミュレーションでエラーが発生します。



推奨: 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剰に表示されることはなくなります。

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。この警告は、未知の値 (X) が未知の値をサポートしないデータ型に割り当てられた場合、ステミュラスがテストベンチから適用される前などに表示されますが、通常は無視しても問題ありません。

上記の例のテストベンチは、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を返します。この場合、結果は `tb_driver` 関数内で検証されますが、戻り値がチェックされた後、最上位テストベンチに返されます。

```
if (U_tb_driver.retval != 0) {
    printf(Test failed  !!!\n);
} else {
    printf(Test passed !\n);
}
return U_tb_driver.retval;
```

SC_MODULES のインスタンスエート

SC_MODULE の階層インスタンスエーションは、次のコード例に示すように合成できます。このコード例では、[SC_METHOD の使用](#) 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ス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_method U_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;
    }
};
```

次のコード例では、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。

```
#include "sc_hier_inst.h"

void sc_hier_inst::full_adder(){
    carry_out= carry1.read() | carry2.read();
}
```

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。次のコード例では、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);
    }
};
```

- 次のコード例は、accum 関数を示しています。この例では、次を示しています。
- 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();
    }
}
```

ループの合成

ループを使用してコード記述する際は、Vivado HLS の SystemC スケジューリング規則 (Vivado HLS は新しいステートで開始することでループを常に合成) を考慮する必要があります。たとえば、次のようなデザインがあるとします。

注記: この例では、最低限のコード部分のみを示しています。

```
sc_in<bool> start;
sc_in<bool> enable;

process code:
    unsigned count = 0;
    while (!start.read()) wait();
    for(int i=0;i<100; i++)
    {
        if(enable.read()) count++;
        wait();
    }
```

次のテストベンチ スティミュラスが使用されているとします。

```
start = true;
enable=true;
wait(1);
start = false;
wait(99);
enable=false;
```


このデザインはCシミュレーション中に実行され、イネーブル信号がサンプリングされ、カウントが100に達成します。合成後、SystemC ループ スケジューリング規則ではループが新しい状態で開始し、ループ内のどの動作もこのポイントの後スケジューリングされる必要があります。たとえば、次のコードは `First Loop Clock` という `wait` 文を示しています。

```
sc_in<bool> start;
sc_in<bool> enable;

process code:
    unsigned count = 0;
    while (!start.read()) wait();
    for(int i=0;i<100; i++)
    {
        wait(); //First Loop Clock
        if(enable.read()) count++;
        wait();
    }
```

最初のクロックが開始信号をサンプリングした後、新しいクロックがはじめて `enable` 信号がサンプリングされるまで2クロックサイクルの遅延があります。この新しいクロックは、テストベンチの2つ目のクロックと同時に発生します。これが連続99クロックの最初のクロックです。3つ目のテストベンチクロック(連続99クロックの2つ目のクロック)では、`enable` 信号がはじめてサンプリングされます。この場合、RTL デザインは `enable` が `false` に設定される前に99までだけカウントします。



推奨: ザイリンクスでは、SystemC でループをコード記述する場合は、ループの最初のアイテムとして `wait()` 文を配置することをお勧めしています。

次の例では、`wait()` 文が合成済みループの最初のクロックまたはステートになります。

```
sc_in<bool> start;
sc_in<bool> enable;

process code:
    unsigned count = 0;
    while (!start.read()) wait();
    for(int i=0;i<100; i++)
    {
        wait(); // Put the 'wait()' at the beginning of the loop
        if(enable.read()) count++;
    }
```

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。

次のコード例では2つのクロック (`clock` および `clock2`) を含むデザインを示しています。

- クロックの1つは `Prcl` 関数を実行する `SC_CTHREAD` をアクティベートするために使用されます。
- もう1つは `Prcl2` 関数を実行する `SC_CTHREAD` をアクティベートするために使用されます。

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 `clock` 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。

```
#includesystemc.h
#includeetlm.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);
    }
};
```

通信チャネル

スレッド、メソッド、モジュール (スレッドおよびメソッドを含む) 間の通信は、チャネルを使用してのみ実行する必要があります。スレッド間の通信には単純な変数は使用できません。

ザイリンクスでは、異なるプロセス (スレッド、メソッド) 間の通信に `sc_buffer` または `sc_signal` を使用することをお勧めしています。`sc_fifo` および `tlm_fifo` は、複数の値が最初の値が読み出されるよりも前に書き込まれる場合に使用できます。

`sc_fifo` および `tlm_fifo` の場合、合成では次のメソッドがサポートされます。

- ノンブロッキングの読み出し/書き込み
- ブロッキングの読み出し/書き込み
- `num_available()/num_free()`
- `nb_can_put()/nb_can_get()`

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。C および C++ と異なり、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しかインターフェイス 合成しか実行されません。

最上位インターフェイスのポートは、すべて次のいずれかの型を使用している必要があります。

- `sc_in_clk`
- `sc_in`
- `sc_out`
- `sc_inout`
- `sc_fifo_in`
- `sc_fifo_out`
- `ap_mem_if`
- `AXI4M_bus_port`

サポートされるメモリ インターフェイスを除き、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。サポートされるメモリ インターフェイスは、次のとおりです。

- `sc_fifo_in`
- `sc_fifo_out`
- `ap_mem_if`

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あります。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。

Vivado HLS では、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イベント ベースの記述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、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 RTL の RAM には変換しません。次の SystemC コードでは、Vivado HLS の指示子を使用して配列ポートを個々の要素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)
{
    ...
}
};
```

RAM ポートの合成コード例

これらの配列を個々の要素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。

次のコード例は、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。このコード例では、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_sysc` 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。

注記: Vivado HLS 環境では、`include/ap_sysc` ディレクトリが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。これについては、次のコード例で示します。

```
#includesystemc.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, RAM_2P> dout;
    ap_mem_port<int, int, 100, RAM_2P> 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);
}
};
```

- `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要素に使用されるデータ型です。上記の例の場合、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、配列内のすべての要素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が発生します。
- `number_of_elements` は記述される配列に含まれる要素数を指定します。
- `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 I/O ポートを決定します。使用可能なターゲットのリストは、次の表を参照してください。

次の表のメモリ 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リング方法に影響します。たとえば、デュアル ポート RAM の場合、次のようになります。

- I/O ポート数はシングル ポート RAM の 2 倍になります。
- 内部操作が並列で実行されるようにスケジューリングできることがあります (ループおよびデータ依存性のようなコード コンストラクトで許容される場合のみ)。

表 46: SystemC の `ap_mem_port` のメモリ ターゲット

ターゲット RAM	説明
RAM_1P	シングル ポート RAM
RAM_2P	デュアル ポート RAM
RAM_T2P	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
ROM_1P	シングル ポート ROM
ROM_2P	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アクセスされるようになります。

```
dout[i] = share_mem[i] + din[i];
```

次のコード例は、上記の例をサポートするテストベンチを示しています。`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。

```
#ifdef __RTL_SIMULATION__
#include "sc_RAM_port_rtl_wrapper.h"
#define sc_RAM_port sc_RAM_port_RTL_wrapper
#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, RAM_2P> dout;
    ap_mem_chn<int,int, 100, RAM_2P> 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;
};
```

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。次のコード例は、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 ポートおよび次のような関連する FIFO 制御信号が含まれます。

- 入力には `empty` および `read` ポート。
- 出力には `full` および `write` ポート。

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 RTL テストベンチに追加されます。

```
#includesystemc.h
#includetlm.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);
    }
};
```

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

- `SC_MODULE` は別の `SC_MODULE` 内には入れ子にできません。
- `SC_MODULE` は別の `SC_MODULE` から派生できません。
- Vivado HLS では、`SC_THREAD` がサポートされません。
- Vivado HLS では、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ス化

SC_MODULE は new を使用してインスタンス化できません。次のコード (SC_MODULE(TOP)) は、その後の例のように変更する必要があります。

```
{
    sc_in<T> din;
    sc_out<T> dout;

    M1 *t0;

    SC_CTOR(TOP) {
        t0 = new M1(t0);
        t0->din(din);
        t0->dout(dout);
    }
}
```

```
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。次に例を示します。

```
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)
    { ... }
};
```

モジュール コンストラクターのコード例

仮想関数

Vivado HLS では、仮想関数がサポートされません。たとえば、次のコードは仮想関数を使用しているので、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

Vivado HLS では `sc_out` ポートの読み出しがサポートされません。たとえば、次のコードは、`out0` に読み出しがあるため、サポートされません。

```
SC_MODULE(DUT)
{
    sc_in<T> in0;
    sc_out<T> out0;
    ...
    void process()
    {
        int var=in0.read()+out0.read();
        out0.write(var);
    }
};
```

高位合成リファレンス ガイド

コマンド リファレンス

add_files

説明

現在のプロジェクトにデザイン ソース ファイルを追加します。

デザイン ソースに含まれるヘッダー ファイルが現在のディレクトリで検索されます。ほか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。

構文

```
add_files [OPTIONS] <src_files>
```

- `<src_files>`: デザインの記述を含むソース ファイルをリストします。

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。

これらのファイルは合成されませんが、`cosim_design` コマンドにより合成後検証が実行されるときに使用されます。

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_files` 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です。

```
-blackbox <file_name.json>
```

RTL ブラック ボックスに使用する JSON ファイルを指定します。このファイルの情報は、合成と C および協調シミュレーションで HLS コンパイラにより使用されます。詳細は、[RTL ブラック ボックス](#) を参照してください。

```
- csimflags <string>
```

シミュレーション コンパイル オプションの文字列です。このオプションで指定したフラグは、シミュレーション コンパイル (C シミュレーションおよび RTL 協調シミュレーションを含み、合成コンパイルは含まない) にのみ適用されます。このオプションは、`-cflags` オプションには影響しません。

プラグマ

同等のプラグマはありません。

例

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_files a.cpp
add_files b.cpp
add_files c.cpp
```

1 行で複数のファイルを追加します。

```
add_files "a.cpp b.cpp c.cpp"
```

`USE_RANDOM` というマクロを有効にするコンパイラ フラグを使用して 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` を指定します。

```
add_files top.cpp -cflags "-DUSE_RANDOM -I./lib_functions"
```

`-tb` オプションを使用すると、プロジェクトにテストベンチ ファイルを追加できます。この例では、1 行で次の複数のファイルを追加しています。

- テストベンチ `a_test.cpp`
- テストベンチで読み出されるすべてのデータ ファイル:
 - `input_stimuli.dat`
 - `out.gold.dat`

```
add_files -tb "a_test.cpp input_stimuli.dat out.gold.dat"
```

上記の例でテストベンチ データ ファイルが `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できます。

```
add_files -tb a_test.cpp
add_files -tb test_data
```

close_project

説明

現在のプロジェクトを閉じます。現在のプロジェクトが Vivado® HLS セッションでアクティブではなくなります。

`close_project` コマンドでは、次が実行されます。

- プロジェクト特定のコマンドおよびソリューション特定のコマンドを入力できなくなります。
- 必須ではありません。新しいプロジェクトを開くと、現在のプロジェクトが閉じます。

構文

```
close_project
```

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

```
close_project
```

- 現在のプロジェクトを閉じます。
- すべての結果を保存します。

close_solution

説明

現在のソリューションを閉じます。現在のソリューションが Vivado HLS セッションでアクティブではなくなります。

次の `close_solution` コマンドを実行

- ソリューション特定のコマンドを入力できなくなります。
- 必須ではありません。新しいソリューションを開くと、現在のソリューションが閉じます。

構文

```
close_solution
```

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

```
close_solution
```

- 現在のプロジェクトを閉じます。
- すべての結果を保存します。

config_array_partition

説明

配列の分割のデフォルト動作を指定します。

構文

```
config_array_partition [OPTIONS]
```

オプション

```
-auto_partition_threshold <int>
```

配列 (定数インデックスがないものも含む) を分割するためのしきい値を設定します。

配列の要素数が指定したしきい値よりも少ない場合、その配列にインターフェイスまたはコア仕様が適用されていなければ、個々の要素に自動的に分割されます。デフォルトは 4 です。

```
-auto_promotion_threshold <int>
```

定数インデックスを含む配列を分割するためのしきい値を設定します。

配列の要素数が指定したしきい値よりも少なく、配列に定数インデックス (インデックスが変数ではない) が含まれる場合、個々の要素に自動的に分割されます。デフォルトは 64 です。

```
-include_extern_globals
```

スループットに基づく自動分割から外部グローバル配列を含めます。

```
-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要素のサイズになります。

```
-maximum_size <int>
```

配列を分割する最大サイズを指定します。

```
-scalarize_all
```

デザインのすべての配列を個々の要素に分割します。

```
-throughput_driven
```

スループットに基づく配列の自動分割を有効にします。

Vivado HLS で、配列を個々の要素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判断されます。

プラグマ

同等のプラグマはありません。

例

要素数が 12 未満の配列 (グローバル配列を除く) をすべて個々の要素に分割します。

```
_config_array_partition -auto_partition_threshold 12 -
_include_extern_globals_
```

Vivado HLS で、スループットを改善するために、関数インターフェイスの配列も含め、どの配列を分割するかが判断されます。

```
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を個々の要素に分割します。

```
config_array_partition -scalarize_all
```

config_bind

説明

マイクロ アーキテクチャ バインディングのデフォルト オプションを設定します。

バインディングとは、加算、乗算、シフトなどの演算子を特定の RTL インプリメンテーションにマップするプロセスです。たとえば乗算 (mult) 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。

構文

```
config_bind [OPTIONS]
```

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を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 に設定すると、実行時間が短縮されます。最適化をほとんど実行できないような場合、たとえばすべての if-else 文の各分岐に相互に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役立ちます。
- High に設定すると、実行時間は長くなりますが、通常は結果が向上します。

```
-min_op <string>
```

特定の演算子のインスタンス数を最小限に抑えます。コードにこのような演算子が複数含まれる場合、演算子が最小限の RTL リソース (コア) で共有されます。

次の演算子を引数として指定できます。

- 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 - 論理演算右シフト
- ashr - 四則演算右シフト
- shl - 左シフト

プラグマ

同等のプラグマはありません。

例

Vivado HLS で次のように処理されるよう指定します。

- バインディング プロセスでのエフォート レベルを上げる。
- 演算子をインプリメントするため、さらに多くのオプションを試す。
- より良いリソース使用率でデザインが生成されるようにする。

```
config_bind -effort high
```

乗算演算子の数が最小限に抑えられるので、RTL の乗算器の数が最小になります。

```
config_bind -min_op mul
```

config_compile

説明

フロントエンドのコンパイルのデフォルト動作を設定します。

構文

```
config_compile [OPTIONS]
```

オプション

```
-ignore_long_run_time
```

多数のロードまたはストア命令に実行時間が長いという警告をスキップします。

```
-name_max_length <threshold>
```

関数名の最大長を指定します。名前がしきい値 (<threshold>) よりも長くなる場合、名前の末尾が切り捨てられます。デフォルトは 80 です。

```
-no_signed_zeros
```

浮動小数点 0 の符号の有無を無視するので、コンパイラで浮動小数点の演算に対して積極的な最適化を実行できます。デフォルト値は off です。

注記: このオプションを使用すると、浮動小数点の計算結果が変わり、C/RTL 協調シミュレーションと一致しくなくなります。テストベンチがこの差に耐えられるようにして、正確な値ではなく、その差異を確認するようにしてください。テストベンチでこのような差異を使用し、それに耐えられるようにするには、[コード例の表 1-6](#) の `cpp_math` の例を参照してください。

```
-pipeline_loops <threshold>
```

ループを自動的にパイプライン処理する場合に使用する小さいしきい値を指定します。デフォルトでは、ループは自動的にパイプライン処理されません。

このオプションを適用すると、最内ループのトリップカウントがしきい値よりも大きい場合は最内ループがパイプライン処理されます。最内ループのトリップカウントがしきい値以下の場合は、その親ループがパイプライン処理されます。最内ループに親ループがない場合は、最内ループがトリップカウントにかかわらずパイプライン処理されます。

しきい値が大きいほど、親ループがパイプライン処理される確率が高くなり、実行時間が長くなります。

```
-unsafe_math_optimizations
```

浮動小数点 0 の符号の有無を無視し、結合的な浮動小数点演算をイネーブルにして、コンパイラで浮動小数点の演算に対して積極的な最適化を実行できるようになります。デフォルト値は off です。

注記: このオプションを使用すると、浮動小数点の計算結果が変わり、C/RTL 協調シミュレーションと一致しくなくなります。テストベンチがこの差に耐えられるようにして、正確な値ではなく、その差異を確認するようにしてください。テストベンチでこのような差異を使用し、それに耐えられるようにするには、[コード例の表 1-6](#) の `cpp_math` の例を参照してください。

プラグマ

同等のプラグマはありません。

例

最内ループのトリップカウントが 30 よりも大きい場合は最内ループをパイプライン処理し、30 以下の場合は親ループをパイプライン処理します。

```
config_compile -pipeline_loops 30
```


浮動小数点 0 の符号の有無を無視します。

```
config_compile -no_signed_zeros
```

浮動小数点 0 の符号の有無を無視し、結合的な浮動小数点演算をイネーブルにします。

```
config_compile -unsafe_math_optimizations
```

config_core

説明

指定したコアをグローバルに設定します。

構文

```
config_core [OPTIONS] <core>
```

オプション

- <core> <string>

コア名を指定します。

- -latency <int>

スケジューリング中に使用されるコアの新しいデフォルト レイテンシを指定します。

プラグマ

config_core と同等のプラグマはありません。

例

DSP48 コアのデフォルト レイテンシを変更します。

```
config_core DSP48 -latency 4
```

config_dataflow

説明

- データフロー パイプライン (set_directive_dataflow コマンドによりインプリメントされる) のデフォルト動作を指定します。
- デフォルトのチャンネル メモリ タイプおよびその深さを指定できます。

構文

```
config_dataflow [OPTIONS]
```

オプション

```
-default_channel [fifo|pingpong]
```

デフォルトでは、データフロー パイプラインを使用する場合、関数間またはループ間のデータをバッファースするのピンポン形式 (pingpong) でコンフィギュレーションされた RAM メモリが使用されます。ストリーミング データを使用する (データの読み出しおよび書き込みを常に順に実行する) 場合は、FIFO メモリの方が効率的なので、FIFO をデフォルトのメモリ タイプとして選択できます。



ヒント: FIFO アクセスを実行するには、set_directive_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>
```

デフォルトの FIFO の深さを指定します。デフォルトの深さは 2 です。

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無視されます。指定しない場合、デフォルトの深さは 2 です。これが FIFO に変換される配列である場合は、デフォルト サイズは元の配列のサイズになります。場合によっては、この設定が控えめすぎ、必要以上に大きな FIFO が作成されてしまうことがあります。このオプションは FIFO のサイズが必要以上に大きいとわかっている場合に使用してください。



注意: このオプションを使用する場合は注意が必要です。FIFO の深さが足りないと、デッドロックになる可能性があります。

```
-scalar_fifo_depth
```

スカラー伝搬 FIFO の最小値を指定します。

コンパイラは、スカラー伝搬を介して C コードを FIFO に変換します。これらの FIFO の最小サイズは -start_fifo_depth で設定できます。このオプションが指定されない場合は、-fifo_depth の値が使用されます。

```
-start_fifo_depth
```

伝搬が開始される FIFO の最小の深さを指定します。

このオプションは、送信および受信間のチャネルが FIFO の場合にのみ有効です。このオプションのデフォルト値は、-fifo_depth オプションと同じで 2 です。このような FIFO によりデッドロックが発生することがあるので、その場合はこのオプションを使用して FIFO の深さを増加します。

プラグマ

同等のプラグマはありません。

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深さ 6 の FIFO に変更します。

```
config_dataflow -default_channel fifo -fifo_depth 6
```



注意: デザイン インプリメンテーションで要素数が 6 より多い FIFO が必要な場合、この設定では RTL 検証エラーが発生します。このオプションはユーザーによる上書きなので、使用する際には注意が必要です。

config_export

説明

ダウンストリーム ツールを実行したり、Vivado IP または Vitis プロジェクト (XO) をパッケージングする際に使用可能な `export_design` のオプションを設定します。

構文

```
config_export [OPTIONS]
```

オプション

```
-description <string>
```

生成した IP カタログ IP の説明を表示します。

```
-display_name
```

生成する IP の表示名を指定します。

```
-flow (syn|impl)
```

RTL 合成を使用して、指定した HDL のさらに正確なタイミングと使用量データを取得します。 `syn` オプションを指定すると RTL 合成が実行され、 `impl` オプションを指定すると RTL 合成とインプリメンテーション(合成済みゲートの詳細な配置配線を含む)の両方が実行されます。 Vivado HLS の GUI では、これらのオプションはそれぞれ [Vivado Synthesis] および [Vivado Synthesis, place and route stage] というチェック ボックスになっています。

```
-format (ipcatalog|sysgen|syn-dcp)
```

IP をパッケージするフォーマットを指定します。サポートされるフォーマットは、次のとおりです。

- `sysgen`
ザイリンクス デザイン スイートの System Generator for DSP で使用できるフォーマット (7 シリーズ デバイスのみ)
- `ip_catalog`
ザイリンクス IP カタログに追加するのに適したフォーマット (7 シリーズ デバイスのデフォルト)
- `syn-dcp`
Vivado の合成済みチェックポイント ファイルこのオプションを使用すると、RTL 合成が自動的に実行されます。

```
-sdx_tcl
```

生成する IP の名前を指定します。

```
-library
```

生成する IP カタログ IP のライブラリ名を指定します。

```
-rtl (verilog |VHDL)
```

-flow オプションを指定した場合に使用する HDL を選択します。指定しない場合は Verilog がデフォルト言語です。

```
-vitis_tcl
```

出力 Tcl ファイルのディレクトリを制御します。

```
-taxonomy
```

このオプションは IP のパッケージに使用します。

```
-vendor
```

生成する IP カタログ IP のベンダー文字列を指定します。

```
-version
```

生成する IP カタログのバージョン文字列を指定します。

```
-vivado_ip_cache <path-to-ip-cache>
```

OOO Vivado プロジェクトに追加される IP キャッシュへのパス。キャッシュに到達すると、RTL 合成のランタイムを削減します。デフォルトは none です。

```
-vivado_impl_strategy {default|<strategy>}
```

export_design -evaluate Vivado run 内で使用されるインプリメンテーション ストラテジを制御します。

このオプションの値は、default または有効な Vivado インプリメンテーション ストラテジの名前になります。

```
-vivado_phys_opt {none|place|route|all}
```

物理最適化を export_design -evaluate Vivado run 内でイネーブルにするかどうかを指定します。

有効な値は次のとおりです。

- none: 物理最適化はイネーブルになりません。
- place: 配置後に実行されます。これがデフォルトです。
- route: 配線後に実行されます。
- all: 配置後および配線後の両方で実行されます。

```
-vivado_synth_design_args {args...}
```

デフォルトは -directive sdx_optimization_effort_high です。

このオプションの値は、export_design -evaluate Vivado 合成 run 内で synth_design に渡されます。

```
-vivado_synth_strategy {default|<strategy>}
```

`export_design -evaluate` Vivado run 内で使用される合成ストラテジを制御します。

このオプションの値は、`default` または有効な Vivado 合成ストラテジの名前になります。

```
-vivado_report_level
```

このオプションを使用すると、使用量レポートとタイミング レポートが作成されます。デフォルト モードは 0 に設定されます。

- 0: 合成および配置配線の両方の終了後に使用量およびタイミング レポートが作成されます。
- 1: 合成および配置配線の両方の終了後に使用量、タイミング、および解析レポートが作成されます。
- 2: 合成および配置配線の両方の終了後に使用量、タイミング、解析、およびフェイルファースト レポートが作成されます。

プラグマ

同等のプラグマはありません。

例

config_interface

説明

インターフェイス合成中に各関数の RTL ポートをインプリメントするのに使用されるデフォルトのインターフェイス オプションを指定します。

構文

```
config_interface [OPTIONS]
```

オプション

```
-clock_enable
```

デザインにクロック イネーブル ポート (`ap_ce`) を追加します。

クロック イネーブルは、アクティブ Low の場合、すべてのクロック動作が実行されないようにし、すべてのシーケンシャル動作がディスエーブルにします。

```
-expose_global
```

グローバル変数を I/O ポートとして使用できるようにします。

変数がグローバルとして作成されても、すべての読み出しおよび書き込みがデザインに対してローカルである場合は、リソースがデザイン内に作成されるので、RTL に I/O ポートは必要ありません。



推奨: グローバル変数が RTL ブロック外の外部ソースまたはデスティネーションになるようにする場合は、このオプションを使用してポートを作成します。

```
-m_axi_addr64
```

デザイン内のすべての M_AXI ポートに対する 64 ビットのアドレス指定をグローバルにイネーブルにします。

```
-m_axi_offset (off|direct|slave)
```

デザイン内のすべての M_AXI インターフェイスのオフセット ポートをグローバルに制御します。

- off (デフォルト)

オフセット ポートは生成されません。

- direct

スカラー入力のオフセット ポートを生成します。

- slave

オフセット ポートを生成して、それを自動的に AXI4-Lite スレーブにマップします。

```
-register_io (off|scalar_in|scalar_out|scalar_all)
```

最上位関数のすべての入力/出力用のレジスタのオン/オフをグローバルに制御します。デフォルト値は off です。

```
-trim_dangling_port
```

構造体に基づいて、インターフェイスのデフォルト動作を上書きします。

デフォルトでは、ブロック インターフェイスでアンパック型構造体のメンバーすべてが、デザイン ブロックで使用されるかどうかに関係なく、RTL ポートになります。これを on に設定すると、生成したブロックで使用されないインターフェイス ポートが削除されます。

プラグマ

同等のプラグマはありません。

例

- グローバル変数を I/O ポートとして使用できるようにします。
- クロック イネーブル ポートを追加します。

```
config_interface -expose_global -clock_enable
```

config_rtl

説明

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設定します。RTL で特定の ID を使用できるようにもします。

デフォルトでは、これらの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。特定の RTL モデルをオプションで指定できます。

構文

```
config_rtl [OPTIONS] <model_name>
```

オプション

```
-heaheader <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。



ヒント: このオプションを使用すると、出力 RTL ファイルにユーザー指定の ID が含まれるようになります。

```
-auto_prefix
```

最上位関数名に接頭辞を自動的に付けます。config_rtl -prefix オプションを使用した場合は無視されます。

```
-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。

```
-enable_maxiConservative
```

AXI マスターに、書き込みチャネル バッファに十分なデータが貯まるまでは、書き込みリクエストを送信しないように伝えるモードです。

```
-reset (none|control|state|all)
```

C コードで初期化される変数は RTL (およびビットストリーム) では常に同じ値に初期化されます。この初期化はパワ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 オプションの設定により、レジスタおよびメモリのリセット方法が決まります。

- none

デザインにリセットを追加しません。

- control (デフォルト)

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変数またはグローバル変数から生成された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。

```
-reset_async
```

すべてのレジスタで非同期リセットを使用します。

このオプションを指定しない場合は、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をアクティブ Low またはアクティブ High にします。

デフォルトは High です。

```
-encoding (binary|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 onehot です。

auto エンコーディングを使用すると、Vivado HLS でコーディング スタイルが決定されますが、サイリンクスの論理合成ツール Vivado で論理合成中に FSM スタイルを抽出して再インプリメントできます。それ以外のエンコーディング スタイルを選択している場合は、エンコーディング スタイルをサイリンクス論理合成ツールで最適化し直すことはできません。

プラグマ

同等のプラグマはありません。

例

出力 RTL ですべてのレジスタが非同期のアクティブ Low リセット信号でリセットされるように設定します。

```
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

説明

Vivado HLS で実行されるスケジューリングのデフォルト タイプを設定します。

構文

```
config_schedule [OPTIONS]
```


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 に設定すると、実行時間が短縮されます。デザイン インプリメンテーションに選択肢があまりない場合に指定すると有益な場合があります。
- High に設定すると、実行時間は長くなりますが、通常は結果が向上します。

```
-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場合にクリティカル パスを表示します。

```
-relax_ii_for_timing
```

スケジューリングでタイミング要件を満たすためにパイプライン処理されたループまたは関数の II を緩和します。通常は、スケジューリングによりタイミングを満たせないデザインが作成されることがあり、タイミング要件が満たされるように論理合成が使用されます。このオプションを使用すると、常にスケジューリングでタイミングを満たすよう指示され、タイミング要件を満たすためにスループット ターゲット (II) が緩和されます。

プラグマ

同等のプラグマはありません。

例

実行時間を短縮するためデフォルトのスケジューリング エフォートを Low に変更します。

```
config_schedule -effort low
```

config_sdx

説明

HLS または XOCC モードのいずれかでツールのコンパイラを実行します。

構文

```
config_sdx [OPTIONS]
```

オプション

```
-target (none|xocc|sds)
```

- none
HLS スタンドアロン モードでツールを実行します。
- xocc

XOCC モードでツールを実行し、XOCC 特有のチェックを有効にします。

- sds

SDSoC フローで使用可能な Vivado IP を生成するように HLS を設定します。

config_unroll

説明

ループ インデックス制限 (またはトリップカウント) に基づいてループを自動的に展開します。

構文

```
config_unroll -tripcount_threshold <value>
```

オプション

```
-tripcount_threshold
```

指定した値よりも反復回数が少ないすべてのループを自動的に展開します。

例

次のコマンドでは、反復回数が 18 未満のループがすべてスケジューリング中に自動的に展開されます。

```
config_unroll -tripcount_threshold 18
```

cosim_design

説明

元の C ベースのテストベンチを使用して、合成済み RTL の合成後協調シミュレーションを実行します。

テストベンチ用のファイルを指定するには、次のコマンドを実行します。

```
add_files -tb
```

シミュレーションはアクティブ ソリューションのサブディレクトリ `sim/<HDL>` で実行されます。

- `<HDL>` は `-rtl` オプションで指定されます。

`cosim_design` でデザインを検証するには、次を使用する必要があります。

- `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。
- 各出力ポートに次のいずれかのインターフェイス モードを使用する必要があります。
 - `ap_vld`
 - `ap_ovld`

- ap_hs
- ap_memory
- ap_fifo
- ap_bus

インターフェイス モードは、書き込み Valid 信号を使用していつ出力が記述されるかを指定します。

構文

```
cosim_design [OPTIONS]
```

オプション

```
-argv <string>
```

<string> はメイン C 関数に渡されます。

ビヘイビアー テストベンチの引数リストを指定します。

```
-compiled_library_dir <string>
```

サードパーティ シミュレータを使用したシミュレーション中のコンパイル済みライブラリのディレクトリを指定します。<string> は、コンパイル済みライブラリのディレクトリへのパスです。

```
-coverage
```

VCS シミュレータを使用したシミュレーションでの範囲を指定します。

```
-disable_deadlock_detection
```

協調シミュレーションのデッドロック検出機能をディスエーブルにします。

```
-ignore_init <integer>
```

最初の <integer> クロック サイクル間の比較チェックを無効にします。

これは、RTL が不明の値 (hX) で開始することがわかっている場合に便利です。

```
-ldflags <string>
```

協調シミュレーションでリンカーに渡すオプションを指定します。

通常は、インクルード パス情報または C テストベンチのライブラリ情報を渡すのに使用されます。

```
-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イネーブルにします。

```
-reduce_diskspace
```

ディスク容量を節約するフローを実行します。シミュレーション中に使用されるディスク容量が削減されますが、実行時間とメモリ容量が増加する可能性があります。

```
-rtl (vhdl|verilog)
```

C/RTL シミュレーションに使用する RTL を指定します。デフォルトは verilog です。HDL シミュレータを選択するには、-tool オプションを使用します。デフォルトは xsim です。

```
-setup
```

すべてのシミュレーション ファイルをアクティブ ソリューションの sim/<HDL> ディレクトリに作成しますが、シミュレーションは実行しません。

```
-tool (*auto*|vcs|modelsim|riviera|isim|xsim|ncsim|xceleium)
```

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を指定します。

```
-trace_level (*none*|all|port)
```

実行されるトレース ファイルの出力レベルを指定します。

C/RTL 協調シミュレーション中の波形トレースのレベルを指定します。all を使用するとすべてのポートおよび信号波形がトレース ファイルに保存され、port を使用すると最上位ポートの波形トレースのみが保存されます。シミュレーションが実行されると、トレース ファイルが現在のソリューションの sim/<RTL> ディレクトリに保存されます。<RTL> ディレクトリは、-rtl オプションでの選択 (verilog または vhd) によって異なります。

デフォルトは none です。

最適化を実行しない場合は、cosim_design により最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っても、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。結果の実行ファイルがより高速に実行される可能性はありますが、実行時間が短縮されるかどうかはデザインによります。ランタイム目的で最適化を実行すると、大型関数に必要なメモリ使用量が多なる可能性があります。

```
-wave_debug
```

データフローおよび順次プロセスと同様、生成した RTL ではすべてのプロセスが視覚化されます。これは、協調シミュレーションに Vivado シミュレータを使用する場合にのみサポートされます。

プラグマ

同等のプラグマはありません。

例

Vivado シミュレータ を使用して検証を実行します。

```
cosim_design
```

VCS シミュレータを使用して Verilog RTL を検証し、波形トレース ファイルの保存をイネーブルにします。

```
cosim_design -tool VCS -rtl verilog -coverage -trace_level all
```

ModelSim を使用して VHDL RTL を検証します。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"
```

create_clock

説明

現在の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 ソリューションのコンテキストでのみ実行可能です。クロック周期は、最適化 (指定のクロック周期で可能な限り多数の演算をチェーン接続) を駆動する制約です。

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。SystemC デザインでは、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 に適用できます。

構文

```
create_clock -period <number> [OPTIONS]
```

オプション

```
-name <string>
```

クロック名を指定します。

名前を指定しない場合は、デフォルト名が使用されます。

```
-period <number>
```

クロック周期をナノ秒 (ns) または MHz で指定します。

- 単位を指定しない場合は、ns が使用されます。
- 周期を指定しない場合は、デフォルトの 10 ns 周期が使用されます。

プラグマ

同等のプラグマはありません。

例

50 ns のクロック周期を指定します。

```
create_clock -period 50
```

デフォルト周期 10 ns を使用してクロックを指定します。

```
create_clock
```

SystemC デザインでは、複数の指定クロックを作成できます (`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
```

csim_design

説明

指定した C テストベンチを使用して合成前の C シミュレーションをコンパイルして実行します。

テストベンチ用のファイルを指定するには、`add_file -tb` を使用します。シミュレーションの作業ディレクトリは、アクティブ ソリューション内の `csim` です。

構文

```
csim_design [OPTIONS]
```

オプション

```
-o
```

コンパイルの最適化をイネーブルにします。

デフォルトでは、コンパイルがデバッグ モードで実行され、デバッグがイネーブルになります。

```
-argv <string>
```

C テストベンチの引数リストを指定します。

`<string>` は C テストベンチの `<main>` 関数に渡されます。

```
-clean
```

クリーン ビルドをイネーブルにします。

このオプションを指定しない場合、`csim_design` ではインクリメンタル コンパイルが実行されます。

```
-ldflags <string>
```

C シミュレーションでリンカーに渡すオプションを指定します。

このオプションは通常、C テストベンチおよびデザインのライブラリ情報を渡すために使用されます。

```
-compiler (*gcc*)
```

C シミュレーションに使用するコンパイラを選択します。デフォルトのコンパイラは gcc (C++ の場合は g++) です。

```
-mflags <string>
```

C シミュレーションでコンパイラに渡すオプションを指定します。

通常、コンパイル速度を上げるために使用されます。

```
-setup
```

アクティブ ソリューションの `csim` ディレクトリに C シミュレーション バイナリを作成しますが、シミュレーションは実行しません。

プラグマ

同等のプラグマはありません。

例

C シミュレーションをコンパイルして実行します。

```
csim_design
```

ソース デザインとテストベンチをコンパイルし、シミュレーション バイナリを生成しますが、バイナリは実行しません。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 `csim/build` ディレクトリで `run.sh` を実行します。

```
csim_design -O -setup
```

csynth_design

説明

アクティブ ソリューション用に 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 ソリューションのコンテキストでのみ実行可能です。データベースにあるエラボレート済みデザインは、設定されている制約に基づいて、スケジューリングされて RTL にマップされます。

構文

```
csynth_design
```

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

最上位デザインで Vivado HLS を実行します。

```
csynth_design
```

delete_project

説明

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。

`delete_project` コマンドでは、削除前に、該当するプロジェクト ディレクトリ `<project>` が有効な Vivado HLS プロジェクトであるかどうかチェックされます。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、このコマンドは実行されません。

構文

```
delete_project <project>
```

- `<project>`: プロジェクト名を指定します。

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

`Project_1` ディレクトリおよびそれに含まれるものをすべて削除することで、`Project_1` を削除します。

```
delete_project Project_1
```

delete_solution

構文

```
delete_solution <solution>
```

- `<solution>`: 削除するソリューションを指定します。

説明

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、このコマンドは実行されません。

プラグマ

同等のプラグマはありません。

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
```

export_design

説明

ダウストリーム ツール用に RTL の合成済みデザインをエクスポートし、IP としてパッケージにします。

サポートされる IP フォーマットは次のとおりです。

- Vivado IP カタログ
- DCP フォーマット
- System Generator

パッケージされたデザインは、次のサブディレクトリのいずれかのアクティブ ソリューションの `impl` ディレクトリに保存されます。

- `ip`
- `sysgen`

構文

```
export_design [OPTIONS]
```

オプション

```
-flow (syn|impl)
```

RTL 合成を使用して、指定した HDL のさらに正確なタイミングと使用量データを取得します。`syn` オプションを使用すると RTL 合成が実行され、`impl` オプションを使用すると RTL 合成とインプリメンテーション (合成済みゲートの詳細な配置配線) の両方が実行されます。

Vivado HLS の GUI では、これらのオプションはそれぞれ [Vivado Synthesis] および [Vivado Synthesis, place and route stage] というチェック ボックスになっています。

```
-format (sysgen|ip_catalog|syn_dcp)
```

IP をパッケージするフォーマットを指定します。

サポートされるフォーマットは、次のとおりです。

- `sysgen`

Vivado デザイン スイートの System Generator for DSP で使用できるフォーマット (ザイリンクス 7 シリーズ デバイスのみ)

- `ip_catalog`

Vivado IP カタログに追加するのに適したフォーマット (ザイリンクス 7 シリーズ デバイスのデフォルト)

- `syn_dcp`

Vivado Design Suite の合成済みチェックポイント。このオプションを使用すると、RTL 合成が自動的に実行されます。

```
-rtl (verilog|vhdl)
```

`-flow` オプションを指定した場合に使用する HDL を選択します。指定しない場合は Verilog がデフォルト言語です。

```
-xo <path-to-output-xo>
```

XO ファイルの直接出力を指定します。

プラグマ

同等のプラグマはありません。

例

System Generator 用に RTL をエクスポートします。

```
export_design -format sysgen
```

IP カタログ用に RTL をエクスポートします。VHDL を評価して、Vivado ツールを使用してより正確なタイミングおよび使用量データを取得します。

```
export_design -flow syn -rtl vhdl -format ip_catalog
```

help

説明

- `<cmd>` に何も指定しない場合、Vivado HLS Tcl コマンドがすべてリストされます。
- Vivado HLS Tcl コマンドを引数として指定すると、そのコマンドの情報が表示されます。

有効な Vivado HLS コマンドに対しては、引数を入力する際に Tab キーを押して自動補完機能を使用できます。

構文

```
help [OPTIONS] <cmd>
```

- `<cmd>`: ヘルプを表示するコマンドを指定します。

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

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
```

add_files コマンドのヘルプを表示します。

```
help add_files
```

list_core

説明

現在インストールされているライブラリに含まれ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。

エラボレーション後は、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、演算子はライブラリのコアにマップされて RTL デザインがインプリメントされます。複数の演算子を 1 つのコアの同じインスタンスにマップし、同じ RTL リソースを共有することも可能です。

list_core コマンドで次の関連オプションを使用すると、使用可能な演算子およびコアをリストできます。

- [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、メモリまたはストレージをインプリメントするタイプなどです。

オプションを指定しない場合は、ライブラリのコアがすべて表示されます。



ヒント: list_core コマンドと set_directive_resource を使用して表示される情報は、特定の演算を特定のコアにインプリメントするために使用します。

構文

```
list_core [OPTIONS]
```

オプション

```
-operation (opers)
```

指定した演算をインプリメント可能なライブラリ内のコアをリストします。演算は次のとおりです。

- add - 加算

- sub - 減算
- mul - 乗算
- udiv - 符号なし除算
- urem - 符号なし剰余 (モジュロ演算子)
- srem - 符号付き剰余 (モジュロ演算子)
- icmp - 整数比較
- shl - 左シフト
- lshr - 論理演算右シフト
- 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
- fforead - FIFO 読み出し
- fifowrite - FIFO 書き込み
- fifonbread - ノンブロッキング FIFO 読み出し
- fifonbwrite - ノンブロッキング FIFO 書き込み

```
-type functional_unit | storage | connector | adapter | ip_block ( )
```

指定したタイプのコアのみをリストします。

- [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lementをインプリメントするコア。

- [Connectors]

直接接続やストリーミング ストレージ エlementを含む、デザイン内の接続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、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 Blocks]

ユーザーが追加した IP コア。

プラグマ

同等のプラグマはありません。

例

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可能なコアをすべてリストします。

```
list_core -operation add
```

ライブラリにある使用可能なメモリ (ストレージ) コアをすべてリストします。

```
list_core -type storage
```



ヒント: 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_part

説明

- ファミリを指定すると、サポートされるデバイス ファミリまたはそのファミリでサポートされるパーツが表示されます。
- ファミリを指定しない場合は、サポートされるすべてのファミリがリストされます。



ヒント: コマンドをオプションを指定せずに実行して表示されたサポート ファミリの中から 1 つを選択し、それをオプションとして指定すると、そのファミリのサポート パーツがリストされます。

構文

```
list_part [OPTIONS]
```

プラグマ

同等のプラグマはありません。

例

サポートされるすべてのファミリをリストします。

```
list_part
```

サポートされる Virtex®-6 パーツをすべてリストします。

```
list_part virtex6
```

open_project

説明

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できるプロジェクトは 1 つのみです。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。

プロジェクトを閉じるには、次のいずれかを実行します。

- `close_project` コマンドを使用します。
- `open_project` コマンドで別のプロジェクトを開きます。

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するには、`delete_project` コマンドを使用します。

構文

```
open_project [OPTIONS] <project>
```

- `<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。
- 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が削除されます。関連付けられているソリューションのディレクトリおよびファイルは保持されますが、結果は無効になっている可能性があります。

`delete_project` コマンドにも `-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 データが削除されます。



推奨: このオプションは、Tcl スクリプトを使用して Vivado HLS を実行している場合に使用してください。それ以外の場合は、`add_files` コマンドを実行するたびにファイルが既存のデータに追加されます。

プラグマ

同等のプラグマはありません。

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。

```
open_project -reset Project_2
```



推奨: Tcl スクリプトでこの方法を使用すると、既存のプロジェクト データにソースまたはライブラリ ファイルが追加されなくなります。

open_solution

説明

アクティブ プロジェクトで既存のソリューションを開くか、新規ソリューションを作成します。



注意: 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にできるソリューションは 1 つのみです。

各ソリューションは、現在のプロジェクト ディレクトリの下の子ディレクトリで管理されます。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。

ソリューションを閉じるには、次のいずれかを実行します。

- `close_solution` コマンドを実行します。
- `open_solution` コマンドで別のソリューションを開きます。

プロジェクトからソリューションを削除し、対応するサブディレクトリを削除するには、`delete_solution` コマンドを使用します。

構文

```
open_solution [OPTIONS] <solution>
```

- `<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にソリューション データをリセットします。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。

プラグマ

同等のプラグマはありません。

例

アクティブ プロジェクトで、`Solution_1` という名前の既存のソリューションを開くか、新規ソリューションを作成します。

```
open_solution Solution_1
```

アクティブ プロジェクトでソリューションを開きます。既存のデータを削除します。

```
open_solution -reset Solution_2
```



推奨: Tcl スクリプトでこの方法を使用すると、既存のソリューション データに追加されなくなります。

set_clock_uncertainty

説明

`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、有効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 ns または % で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、有効クロック周期に基づいてデザインが最適化され、論理合成および配線で考慮されるようにダウンストリーム ツールにマージンが渡されます。このコマンドはアクティブ ソリューションのコンテキストでのみ実行可能です。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に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、各クロックに異なるばらつきを指定できます。

構文

```
set_clock_uncertainty <uncertainty> <clock_list>
```

- `<uncertainty>`: クロック周期でマージンとして使用する部分 () を表す値をナノ秒 (ns) で指定します。
- `<clock_list>`: ばらつきを適用するクロックのリストを指定します。何も指定しない場合、すべてのクロックに適用されます。

プラグマ

同等のプラグマはありません。

例

クロックのばらつきまたはマージンを 0.5 ns に指定します。これにより、Vivado HLS で使用可能なクロック周期が 0.5 ns 削減されます。

```
set_clock_uncertainty 0.5
```

この SystemC の例では、クロック ドメインを 2 つ作成し、ドメインごとに異なるクロックのばらつきを指定しています。

```
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
```



ヒント: SystemC デザインでは、複数のクロックがサポートされます。クロックを適切な関数に適用するには、`set_directive_clock` コマンドを使用します。

set_directive_allocation

説明

リソース割り当てのためのインスタンス制限を指定します。

特定の関数または演算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し、制限できます。たとえば、C ソース コードに `foo_sub` という関数のインスタンスが 4 つ 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 つだけにできます。4 つのインスタンスすべてが、同じ RTL ブロックを使用してインプリメントされます。

構文

```
set_directive_allocation [OPTIONS] <location> <instances>
```

- <location>: 場所を `function[/label]` の形式で指定します。
- <instances> には関数または演算子を指定します。

関数には、`set_directive_inline` コマンドでインライン展開されたり、Vivado HLS で自動的にインライン展開されたりしていない元の C コードの関数を指定できます。

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`: Integer Compare
- `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 (`-type` オプションで指定されているタイプのインスタンス) の最大数を設定します。

```
-type [function|operation]
```

インスタンス タイプには、`function` (デフォルト) または `operation` を指定できます。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS allocation \
    instances=<Instance Name List> \
    limit=<Integer Value> \
    <operation, function>
```

例

関数 `foo` のインスタンスが複数含まれているデザイン `foo_top` に対して、RTL での `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 HLS allocation instances=foo limit=2 function
```

`My_func` のインプリメンテーションに使用される乗算の数を 1 に制限します。この制限は、`My_func` のサブ関数に含まれる乗算器には適用されません。サブ関数のインプリメンテーションに使用される乗算器の数を制限するには、そのサブ関数に `ALLOCATION` 指示子を指定するか、サブ関数を関数 `My_func` 内にインライン展開します。

```
set_directive_allocation -limit 1 -type operation My_func mul
#pragma HLS allocation instances=mul limit=1 operation
```

set_directive_array_map

説明

小型の配列を大型の配列にマップします。

通常は、`set_directive_array_map` コマンド (同じ `-instance` ターゲット) を使用して、複数の小型の配列を 1 つの大型の配列にマップします。この大型の配列は、1 つの大型メモリ (RAM または FIFO) リソースに配置できます。

新しいターゲットが次のどちらの連結なのかを指定するには、`-mode` オプションを使用します。

- 要素 (水平マップ)
- ビット幅 (垂直マップ)

配列は `set_directive_array_map` コマンドが発行された順に、次のものから連結されます。

- 水平マップの場合はターゲット要素 0。
- 垂直マップの場合はビット 0。

構文

```
set_directive_array_map [OPTIONS] <location> <array>
```

`<location>`: 配列変数を含む場所を `function[/label]` の形式で指定します。`<variable>` は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です。

オプション

```
-instance <string>
```

現在の配列変数をマップする新しい配列インスタンス名を指定します。

```
-mode (horizontal|vertical)
```

- デフォルトは水平マップ (`horizontal`) で、配列は要素の多い配列に連結されます。

- 垂直マップ (vertical) では、ワード数の大きい配列に連結されます。

```
-offset <integer>
```



重要: 水平マップ専用のオプションです。

現在のマップ操作でターゲット インスタンスの絶対オフセットを指定します。次に例を示します。

- 配列変数の要素 0 は、新しいターゲットの要素 <int> にマップされます。
- ほかの要素は新しいターゲットの <int+1>、<int+2> などにマップされます。

値を指定しない場合は、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、配列の連結がターゲットの次の未使用要素から開始されます。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS array_map \  
    variable=<variable> \  
    instance=<instance> \  
    <horizontal, vertical> \  
    offset=<int>
```

例

次のコマンドでは、関数 `foo` の配列 `A[10]` および `B[15]` を、新しい 1 つの配列 `AB[25]` にマップしています。

- 要素 `AB[0]` は `A[0]` と同じになります。
- 要素 `AB[10]` は `B[0]` と同じになります (`-offset` 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 HLS array_map variable=A instance=AB horizontal  
#pragma HLS array_map variable=B instance=AB horizontal
```

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` の要素数は `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 HLS array_map variable=C instance=CD vertical  
#pragma HLS array_map variable=D instance=CD vertical
```

set_directive_array_partition

説明

配列をより小型の配列または個々の要素に分割します。

この分割により、次のようになります。

- 1つの大型メモリではなく、複数の小型メモリまたは複数のレジスタを含む RTL が生成されます。
- 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ます。
- デザインのスループットが向上する可能性があります。
- より多くのメモリ インスタンスまたはレジスタが必要となります。

構文

```
set_directive_array_partition [OPTIONS] <location> <array>
```

- `<<location>>`: 配列変数を含む場所を `function[/label]` の形式で指定します。
- `<<array>>`: パーティションする配列変数を指定します。

オプション

```
-dim <integer>
```

注記: 複数次元の配列にのみ使用します。

配列のどの次元を分割するかを指定します。

- 0 を指定すると、すべての次元が指定したオプションで分割されます。
- その他の値を指定すると、その次元のみが分割されます。たとえば、1 を指定した場合、最初の次元のみが分割されます。

```
-factor <integer>
```

注記: このオプションは、`block` または `cyclic` タイプのパーティションにのみ使用します。

作成する小型配列の数を指定します。

```
-type (block|cyclic|complete)
```

- `block` タイプのパーティションでは、元の配列の連続したブロックから小型配列が作成されます。N が `-factor` オプションで定義される整数だとすると、1つの配列が N 個のブロックに分割されます。
- `cyclic` パーティションでは、元の配列の要素をインターリーブすることにより小型配列が作成されます。たとえば、`-factor` が 3 の場合、要素は次のように割り当てられます。
 - 。 要素 0 は 1 番目の新しい配列。
 - 。 要素 1 は 2 番目の新しい配列。
 - 。 要素 2 は 3 番目の新しい配列。
 - 。 要素 3 は再び 1 番目の新しい配列。
- `complete` 分割では、配列を個々の要素に分割します。1次元配列の場合は、メモリが個々のレジスタに分割されます。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。

デフォルトは `complete` です。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS array_partition \
    variable=<variable> \
    <block, cyclic, complete> \
    factor=<int> \
    dim=<int>
```

例

関数 `foo` の配列 `AB[13]` を 4 つの配列に分割します。4 は 13 の因数ではないので、次のように分割されます。

- 3 つの配列に要素が 3 個ずつ含まれます。
- 1 つの配列に 4 つのエレメント (`AB[9:12]`) が含まれます。

```
set_directive_array_partition -type block -factor 4 foo AB
#pragma HLS array_partition variable=AB block factor=4
```

関数 `foo` の配列 `AB[6][4]` を各次元が `[6][2]` の 2 つの配列に分割します。

```
set_directive_array_partition -type block -factor 2 -dim 2 foo AB
#pragma HLS array_partition variable=AB block factor=2 dim=2
```

関数 `foo` の `AB[4][10][6]` のすべての次元を個々の要素に分割します。

```
set_directive_array_partition -type complete -dim 0 foo AB
#pragma HLS array_partition variable=AB complete dim=0
```

set_directive_array_reshape

説明

配列の分割と垂直配列マップを組み合わせ、要素数が少なくワード数の大きい新しい配列を 1 つ作成します。

次の `set_directive_array_reshape` コマンドを実行

1. 配列を複数の配列に分割 (`set_directive_array_partition` と同じ)。
2. 分割された配列が垂直方向を自動的にまとめて (`set_directive_array_map-type vertical` と同じ)、ワード数の大きい 1 つの配列を新しく作成。

構文

```
set_directive_array_reshape [OPTIONS] <location> <array>
```

- `<location>`: 配列変数を含む場所を `function[/label]` の形式で指定します。
- `<array>`: 再形成する配列変数を指定します。

オプション

```
-dim <integer>integer>
```

注記: 複数次元の配列にのみ使用します。

配列のどの次元を再形成するかを指定します。

- `value = 0` にすると、すべての次元が指定したオプションで分割されます。
- その他の値を指定すると、その次元のみが分割されます。たとえば `value = 1` の場合、最初の次元のみが分割されます。

```
-factor <integer>
```

注記: このオプションは、`block` または `cyclic` タイプの再形成にのみ使用します。

作成する一時的な小型配列の数を指定します。

```
-type (block|cyclic|complete)
```

- `block` タイプの再形成では、元の配列の連続したブロックから小型配列が作成されます。これにより、配列が N 個 (N は `-factor` オプションで定義されている整数値) のブロックに分割され、その N 個のブロックが `word-width*N` で 1 つの配列にまとめられます。デフォルトは `complete` です。
- `cyclic` タイプでは、元の配列の要素をインターリーブすることにより小型配列が作成されます。たとえば `-factor 3` の場合、要素 0 は新しく作成される 1 番目の配列に割り当てられ、要素 1 は 2 番目、要素 2 は 3 番目、要素 3 は 1 番目の配列に割り当てられます。最終的な配列は、新しい配列を 1 つの配列に垂直連結 (ワードを連結してワード数が大きいものを作成) したものにります。
- `complete` 再形成では、配列を一時的に個々の要素に分割してから、ワード数の大きい 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 N 個の M ビット要素を含む場合、 $N*M$ ビットのレジスタとなる)。

```
-object
```

注記: コンテナ配列にのみ使用します。

コンテナ内のオブジェクトを再形成します。このオプションを指定すると、オブジェクトのすべての次元が再形成されますが、コンテナの次元はすべて保持されます。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS array_reshape \
    variable=<variable> \
    <block, cyclic, complete> \
    factor=<int> \
    dim=<int>
```

例

関数 `foo` の 8 ビット配列 `AB[17]` を、5 つの要素を含む新しい 32 ビット配列 1 つに再形成します。

4 は 13 の因数ではないので、次のように分割されます。

- `AB[17]` は 5 番目の要素の下位 8 ビットに配置。

- 5 番目の要素の残りは未使用。

```
set_directive_array_reshape -type block -factor 4 foo AB
#pragma HLS array_reshape variable=AB block factor=4
```

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。この次元 2 の幅は 2 倍です。

```
set_directive_array_reshape -type block -factor 2 -dim 2 foo AB
#pragma HLS array_reshape variable=AB block factor=2 dim=2
```

関数 `foo` の 8 ビット配列 `AB[4][2][2]` を、ビット幅が $4 \times 2 \times 2 \times 8 (=128)$ の 1 要素配列 (1 つのレジスタ) に再形成します。

```
set_directive_array_reshape -type complete -dim 0 foo AB
#pragma HLS array_reshape variable=AB complete dim=0
```

set_directive_clock

説明

指定したクロックを指定した関数に適用します。

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。 `create_clock` で指定したクロック周期が、デザイン内のすべての関数に適用されます。

SystemC デザインでは、複数のクロックがサポートされます。 `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` に適用できます。各 `SC_MODULE` は、1 つのクロックを使用して合成されます。

構文

```
set_directive_clock <location> <domain>
```

- `<location>`: クロックを適用する関数を指定します。
- `<domain>`: `create_clock` コマンドの `-name` オプションで指定したクロック名を指定します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS clock domain=<string>
```

例

次のような SystemC デザインがあるとします。

- 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。
- 最上位ではその関数内で `fast_clock` のみを使用されます。
- サブブロック `foo` では `slow_clock` のみを使用されます。

この場合、下のコマンドは次を実行します。

- 両方のクロックを作成します。

- `fast_clock` を `foo_top` に適用します。
- `slow_clock` をサブブロック `foo` に適用。

```
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 HLS clock domain=fast_clock
#pragma HLS clock domain=slow_clock
```

注記: `create_clock` と同等のプラグマはありません。

set_directive_dataflow

説明

関数またはループにデータフロー最適化を実行し、RTL インプリメンテーションの同時実行性を向上します。

C 記述では、すべての演算が順次に実行されます。`set_directive_allocation` などのリソースを制限する指示子を指定しない場合は、Vivado HLS ではレイテンシを最小限に抑え、同時実行性を向上するように処理されます。

ただし、データ依存性のためにこれが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またはループは、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ます。そのため、そのデータを消費する次の関数またはループの演算を開始できません。

ただし、前の関数またはループがすべての演算を完了する前に、次の関数またはループの演算を開始できるようにすることは可能です。

データフロー最適化を指定すると、Vivado HLS では次が実行されます。

- 順次関数/ループ間のデータフローを解析します。
- プロデューサー関数またはループが完了する前にコンシューマー関数またはループを開始できるようにするチャネルを (ピンポン RAM または FIFO に基づいて) 作成しようとします。

これにより関数またはループが並列実行され、次を達成できます。

- レイテンシが低減します。
- RTL デザインのスループットが向上します。

開始間隔 (II) (関数またはループの開始から次の関数またはループの開始までのサイクル数) が指定されていない場合は、Vivado HLS で開始間隔が最小になるようにし、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することが試みられます。

構文

```
set_directive_dataflow <location>
```

- `<location>`: データフロー最適化を実行する場所を `function[/label]` の形式で指定します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS dataflow
```

例

次の例では、関数 `foo` 内にデータフロー最適化を指定しています。

```
set_directive_dataflow foo
#pragma HLS dataflow
```

set_directive_data_pack

説明

構造体 (struct) のデータ フィールドをワード幅が広い 1 つのスカラーにパックします。

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て幅の広いスカラーにまとめられ、ほかのスカラー フィールドと共にパックされます。

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。最初のフィールドにワードの最下位部のように、すべてのフィールドがマップされるまで配置されます。

注記: DATA_PACK 最適化では、ほかの構造体を含む構造体のパックはサポートされません。

構文

```
set_directive_data_pack [OPTIONS] <location> <variable>
```

- `<location>`: パックする変数を含む場所を `function[/label]` の形式で指定します。
- `<variable>`: パックする変数を指定します。

オプション

```
-instance <string>
```

パック後の変数の名前を指定します。何も指定しない場合は、`variable` の名前が使用されます。

```
-byte_pad (struct_level|field_level)
```

8 ビット境界でデータをパックするかどうかを指定します。

- `struct_level`: 構造体を最初にパックしてから、それを 8 ビット境界でパックします。
- `field_level`: 各フィールドをそれぞれ 8 ビット境界でパックしてから、構造体をパックします。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS data_pack variable=<variable> instance=<string>
```

例

次の例では、関数 `foo` の 8 ビット フィールド 3 つを持つ構造体 (`typedef struct {unsigned char R, G, B;} pixel`) の配列 `AB[17]` を 24 ビットの 17 要素の配列 1 つにパックしています。

```
set_directive_data_pack foo AB
#pragma HLS data_pack variable=AB
```

次の例では、関数 `foo` にある 3 つの 8 ビットフィールドを含む struct ポインター `AB` (`typedef struct {unsigned char R, G, B;} pixel`) を 1 つの 24 ビット ポインターにパックしています。

```
set_directive_data_pack foo AB
#pragma HLS data_pack variable=AB
```

set_directive_dependence

説明

Vivado HLS では、次の依存度が検出されます。

- ループ内 (ループ独立依存)。
- 同じループの反復間 (ループ運搬依存)。

こうした依存は、演算をスケジューリングするタイミング、特に関数およびループのパイプライン処理に影響します。

- ループ独立依存

同じ要素が同じループ反復でアクセスされます。

```
for (i=0;i<N;i++) {
    A[i]=x;
    y=A[i];
}
```

- ループキャリー依存

同じ要素が異なるループ反復でアクセスされます。

```
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や、外部要件を満たす必要があるような状況 (2 つの入力が同じインデックスにならない場合など) では、依存解析が保守的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し、偽依存を解決できます。

構文

```
set_directive_dependence [OPTIONS] <location>
```

- `<location>`: 依存を指定する場所を `function[/label]` の形式で指定します。

オプション

```
-class (array|pointer)
```

依存を明確にする必要がある変数のクラスを指定します。これは、`-variable` オプションが使用されているときは使用できません。

```
-dependent (true|false)
```

依存を使用する必要があるか (`true`)、削除するか (`false`) を指定します。デフォルトは `true` です。

```
-direction (RAW|WAR|WAW)
```

注記: ループ運搬依存にのみ使用します。

依存のタイプは次のように指定します。

- **RAW (Read-After-Write - 真の依存)**
書き込み命令により値が書き込まれ、その値が読み出し命令で使用されます。
- **WAR (Write-After-Read - アンチ依存)**
読み出し命令で値が取得され、その値が書き込み命令で上書きされます。
- **WAW (Write-After-Write - 出力依存)**
2 つの書き込み命令により、特定の順序で同じロケーションに書き込みが実行されます。

```
-distance <integer>
```

注記: `-dependent` が `true` に設定されているループ運搬依存でのみ使用します。

配列アクセスの反復間隔を指定します。

```
-type (intra|inter)
```

依存のタイプを指定します。

- 同じループ反復内 (`intra`)。
- 異なるループ反復間 (`inter`) (デフォルト)。

```
-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` オプションが使用されているときは使用できません。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS dependence \
    variable=<variable> \
    <array, pointer> \
    <inter, intra> \
    <RAW, WAR, WAW> \
    distance=<int> \
    <false, true>
```

プラグマのオプションはすべて必須です。inter/intra または false/true を指定しないと、デフォルトで次のようになります。

```
#pragma HLS DEPENDENCE variable=xxx inter false
```

例

次の例では、関数 `foo` の `loop_1` の同じ反復での `Var1` 間の依存を削除しています。

```
set_directive_dependence -variable Var1 -type intra \
-dependent false foo/loop_1
#pragma HLS dependence variable=Var1 intra false
```

`foo` 関数の `loop_2` にあるすべての配列の依存により、同じループ反復ではすべての読み出しが書き込みの後に実行されるように Vivado HLS に伝えられます。

```
set_directive_dependence -class array -type intra \
-dependent true -direction RAW foo/loop_2
#pragma HLS dependence array inter RAW true
```

set_directive_expression_balance

説明

C ベースの仕様は演算シーケンスで記述され、RTL で長い演算チェーンとなることがあります。クロック周期が短い場合にデザイン レイテンシが増加する可能性があります。

デフォルトでは、Vivado HLS で演算の関連性および接続性を考慮して、演算が並べ替えられます。これにより、ツリーのバランスが取られてチェーンが短くなるので、ハードウェア リソースは増加しますが、レイテンシを削減できる可能性があります。

`set_directive_expression_balance` コマンドを使用すると、この演算調整を指定範囲内でオン/オフにできます。

構文

```
set_directive_expression_balance [OPTIONS] <location>
```

- `<location>`: 演算調整をオン/オフにする場所を `function[/label]` の形式で指定します。

オプション

```
-off
```

演算調整を指定の場所でオフにします。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS expression_balance <off>
```

例

関数 `My_Func` 内で演算式バランス調整をオフにします。

```
set_directive_expression_balance -off My_Func
#pragma HLS expression_balance off
```

関数 `My_Func2` で演算調整をオンにします。

```
set_directive_expression_balance My_Func2
#pragma HLS expression_balance
```

set_directive_function_instantiate

説明

デフォルトでは次のようになります。

- 関数は RTL で個別の階層ブロックのままになります。
- 同じ階層レベルにある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に固有の RTL インプリメンテーションを作成し、各インスタンスを最適化できるようにします。

デフォルトでは、次のコードから、3 つのインスタンスすべてに対して関数 `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);
}
```

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。

構文

```
set_directive_function_instantiate <location> <variable>
```

- `<location>`: 関数の固有のインスタンスを作成する場所を `function[/label]` の形式で指定します。
- `variable <string>`: 定数として指定する引数 `<string>` を指定します。

オプション

このコマンドにはオプションはありません。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS function_instantiate variable=<variable>
```

例

先ほどの例は、次の Tcl (または関数 `foo_sub` に配置されたプラグマ) により、関数 `foo_sub` の各インスタンスが入力 `incr` に対して個別に最適化されるようにしています。

```
set_directive_function_instantiate foo_sub incr  
#pragma HLS function_instantiate variable=incr
```

set_directive_inline

説明

関数を階層の別エンティティとして削除します。関数をインライン展開すると、別の階層として表示されなくなります。

関数をインライン展開すると、関数内の演算が共有され、周辺の演算と効率よく最適化されるようになります。ただし、インライン展開された関数は共有できないので、エリアが増加する可能性があります。

デフォルトでは、インライン展開は関数階層のすぐ下の階層でのみ実行されます。

構文

```
set_directive_inline [OPTIONS] <location>
```

- `<location>`: インライン展開を実行する場所を `function[/label]` の形式で指定します。

オプション

```
-off
```

関数のインライン展開をオフにし、指定の関数がインライン展開されないようにします。たとえば、呼び出し関数に `-recursive` オプションが指定されている場合、ほかのすべての関数がインライン展開されても、特定の関数がインライン展開されないようにできます。

```
-recursive
```

デフォルトでは、インライン展開は 1 つの階層でのみ実行されます。指定の関数に含まれる関数はインライン展開されません。 `-recursive` オプションを使用すると、階層全体ですべての関数が再帰的にインライン展開されます。

```
-region
```

指定の範囲内の関数をすべてインライン展開します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS inline <region | recursive | off>
```

例

foo_top に含まれるすべての関数をインライン展開します。下位関数はインライン展開されません。

```
set_directive_inline -region foo_top
#pragma HLS inline region
```

foo_sub1 関数のみをインライン展開します。

```
set_directive_inline foo_sub1
#pragma HLS inline
```

foo_sub2 を除く、foo_top にあるすべての関数が階層全体で再帰的にインライン展開されます。1 つ目のプラグマは foo_top に配置されています。2 つ目のプラグマは 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 HLS inline region recursive
#pragma HLS inline off
```

set_directive_interface

説明

インターフェイス合成で関数記述から RTL ポートをどのように作成するかを指定します。

RTL インプリメンテーションのポートは次のものから導出されます。

- 指定されている任意の関数レベルのプロトコル。
- 関数指数。
- 最上位関数によりアクセスされ、スコープ外部で定義されるグローバル変数。

関数レベルのハンドシェイク:

- 関数がいつ演算を開始するかを制御。
- 関数レベルのハンドシェイクは、次のタイミングを示します。
 - 演算の終了
 - アイドル状態
 - 新しい入力を受信する準備完了

関数レベルのプロトコルのインプリメンテーションは次のようになります。

- ap_ctrl_none、ap_ctrl_hs または ap_ctrl_chain モードで制御。
- 最上位関数の名前のみが必要。

注記: プラグマに関数 `return` を指定。

各関数引数は、有効ハンドシェイクや肯定応答ハンドシェイクなどの独自の I/O プロトコルを持つように指定できます。

グローバル変数がアクセスされても、すべての読み出しおよび書き込みがデザインのローカルである場合は、リソースはデザイン内に作成され、RTL に I/O ポートは必要ありません。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、インターフェイスは標準関数引数と同様の方法で指定します。例を参照してください。

`set_directive_interface` がサブ関数に使用されている場合は、`-register` オプションを使用できます。`-mode` オプションは、サブ関数ではサポートされません。

構文

```
set_directive_interface [OPTIONS] <location> <port>
```

- `<location>`: 関数インターフェイスまたはレジスタを指定する場所を `function[/label]` の形式で指定します。
- `<port>`: インターフェイスを合成するパラメーター (関数引数またはグローバル変数) を指定します。
`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

`-bundle <string>`: 関数引数を AXI ポートにまとめます。デフォルトでは、Vivado HLS は、AXI4-Lite インターフェイスとして指定されたすべての関数引数は 1 つの AXI4-Lite ポートにまとめます。Vivado HLS は、AXI4 インターフェイスとして指定されたすべての関数引数も 1 つの AXI4 ポートにまとめます。`-bundle` オプションでは、`<string>` が同じ関数引数がすべて明示的に同じインターフェイス ポートにまとめられ、RTL ポートの名前が `<string>` になります。

```
-mode (ap_none|ap_stable|ap_vld|ap_ack|ap_hs|ap_ovld|ap_fifo| ap_bus|
ap_memory|bram|axis|s_axilite|m_axi|ap_ctrl_none|ap_ctrl_hs |ap_ctrl_chain)
```

次は、Vivado HLS で `-mode` オプションがどのようにインプリメントされるかをまとめたものです。詳細は、[インターフェイス合成リファレンス](#)を参照してください。

- `ap_none`: プロトコルなし。インターフェイスはデータ ポートです。
- `ap_stable`: プロトコルなし。インターフェイスはデータ ポートです。Vivado HLS では、リセット後はデータ ポートが常に安定していると想定され、内部最適化により不要なレジスタが削除されます。
- `ap_vld`: データ ポートと、データが読み出しまたは書き込みに対して有効になったことを示す `valid` ポートがインプリメントされます。
- `ap_ack`: データ ポートと、データが読み出された/書き込まれたことをする `acknowledge` ポートがインプリメントされます。
- `ap_hs`: データ ポートと、データが読み出しおよび書き込みに対して有効になったことを示す `valid` ポートおよびデータが読み出された/書き込まれたことを肯定応答する `acknowledge` ポートがインプリメントされます。
- `ap_ovld`: 出力データ ポートと、データが読み出しまたは書き込みに対して有効になったことを示す `valid` ポートがインプリメントされます。

注記: Vivado HLS では、入力引数または読み出し/書き込み引数の入力部分は `ap_none` モードを使用してインプリメントされます。

- `ap_fifo`: 標準 FIFO インターフェイスのポートが、アクティブ Low FIFO の `empty` および `full` ポートが関連付けられたデータ入力および出力ポートを使用してインプリメントされます。

注記: このインターフェイスは、読み出し引数または書き込み引数のみに使用できます。ap_fifo モードでは双方向の読み出し/書き込み引数はサポートされません。

- ap_bus: ポインターおよび参照渡しポートがバス インターフェイスとしてインプリメントされます。
- ap_memory: 配列引数が標準 RAM インターフェイスとしてインプリメントされます。Vivado IP インテグレーターで RTL デザインを使用する場合は、メモリ インターフェイスは個別のポートととして表示されます。
- bram: 配列引数が標準 RAM インターフェイスとしてインプリメントされます。Vivado IP インテグレーターで RTL デザインを使用する場合は、メモリ インターフェイスはシングル ポートとして表示されます。
- axis: すべてのポートが AXI4-Stream インターフェイスとしてインプリメントされます。
- s_axilite: すべてのポートが AXI4-Lite インターフェイスとしてインプリメントされます。Vivado HLS では、RTL のエクスポート中に C ドライバー ファイルの関連セットが生成されます。
- m_axi: すべてのポートが AXI4 インターフェイスとしてインプリメントされます。32 ビット (デフォルト) または 64 ビットのアドレス ポートを指定し、アドレス オフセットを制御するには、config_interface コマンドを使用できます。
- ap_ctrl_none: ブロック レベル I/O プロトコルなし。

注記: ap_ctrl_none を使用すると、C/RTL の協調シミュレーション機能を使用してデザインを検証できなくなることがあります。

- ap_ctrl_hs: デザインの演算を start し、デザインが idle、done、および新しい入力データに対して ready になっていることを示すブロック レベルの制御ポート セットをインプリメントします。

注記: ap_ctrl_hs モードはデフォルトのブロック レベル I/O プロトコルです。

- ap_ctrl_chain: デザインの演算を start および continue し、デザインが idle、done、および新しい入力データに対して ready になっていることを示すブロック レベルの制御ポート セットをインプリメントします。

-name <string>:: ポートの名前を変更します。生成された RTL ポートでこの名前が使用されます。

-depth: テストベンチで処理されるサンプルの最大数を指定します。この設定は、Vivado HLS で RTL 協調シミュレーション用に作成される検証アダプターに必要な FIFO の最大サイズを示します。このオプションは、ap_fifo または ap_bus モードを使用するポインター インターフェイスに必要です。

-register: 信号および関連プロトコル信号にレジスタを付け、少なくとも関数実行の最終サイクルまで信号が保持されます。このオプションは、最上位関数の次のスカラー インターフェイスに適用されます。

- ap_none
- ap_ack
- ap_vld
- ap_ovld
- ap_hs
- ap_fifo

-register_mode (both|forward|reverse|off): レジスタをフォワード パス (TDATA および TVALID)、リバース パス (TREADY)、またはその両方のパス (TDATA、TVALID、および TREADY) に配置するか、レジスタをどの信号にも配置しないか (off) を指定します。デフォルトは both です。AXI-Stream サイドチャネル信号はデータ信号と考慮され、TDATA にレジスタが付けられるとレジスタが付けられます。

-offset <string>: AXI4-Lite および AXI4 インターフェイスのアドレス オフセットを制御します。AXI4-Lite インターフェイスの場合、<string> はレジスタ マップのアドレスを指定します。AXI インターフェイスの場合、<string> には次を指定します。

- `off`: オフセット ポートは生成しません。
- `direct`: スカラー入力のオフセット ポートを生成します。
- `slave`: オフセット ポートを生成し、AXI4-Lite スレーブ インターフェイスに自動的にマップします。

`-clock <string>`: デフォルトでは、AXI4-Lite インターフェイス クロックはシステム クロックと同じです。このオプションを使用すると、AXI4-Lite インターフェイスに別のクロックを指定できます。`-bundle` オプションを使用して複数の最上位関数引数を 1 つの AXI4-Lite インターフェイスにまとめている場合は、`clock` オプションはバンドルメンバーの 1 つにのみ指定します。

`- latency <value>`: このオプションは、`ap_memory` および AXIM インターフェイスに使用できます。

- `ap_memory` インターフェイスでは、インターフェイス オプションでインターフェイスを起動する RAM リソースの読み出しレイテンシを指定します。デフォルトでは、1 クロック サイクルの読み出し演算が使用されます。このオプションでは、複数クロック サイクルの読み出しレイテンシを使用した外部 RAM を記述できます。
- AXIM インターフェイスでは、AXI4 インターフェイスのレイテンシを指定し、読み出しまたは書き込みの指定サイクル (レイテンシ) 前にバス要求を開始できるようにします。このレイテンシ値が小さすぎると、デザインが準備完了になるのが早すぎ、バスを待つために停止する可能性があります。レイテンシ値が大きすぎると、バス アクセスはデザインがアクセスを開始するのを待つためにアイドル状態になる可能性があります。

`-max_read_burst_length`: AXIM インターフェイスと一緒に使用する場合、このオプションでバースト転送で読み出されるデータの最大数を指定します。

`-max_write_burst_length`: AXIM インターフェイスと一緒に使用する場合、このオプションでバースト転送で書き込まれるデータの最大数を指定します。

`-num_read_outstanding`: AXIM インターフェイスと一緒に使用する場合、このオプションでデザインが停止するまでに、AXI4 バスに対して応答なしで読み込み要求を送信できる回数を指定します。これによって、デザイン内の内部ストレージで、つまり FIFO のサイズが変わります。

```
num_read_outstanding*max_read_burst_length*word_size.
```

`-num_write_outstanding`: AXIM インターフェイスと一緒に使用する場合、このオプションでデザインが停止するまでに、AXI4 バスに対して応答なしで書き込み要求を送信できる回数を指定します。これによって、デザイン内の内部ストレージで、つまり FIFO のサイズが変わります。

```
num_read_outstanding*max_read_burst_length*word_size
```

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS interface <mode> register port=<string>
```

例

関数 `foo` の関数レベルのハンドシェイクをオフにします。

```
set_directive_interface -mode ap_ctrl_none foo
#pragma HLS interface ap_ctrl_none port=return
```

関数 `foo` の引数 `InData` に `ap_vld` インターフェイスを含め、入力にレジスタを付けます。

```
set_directive_interface -mode ap_vld -register foo InData
#pragma HLS interface ap_vld register port=InData
```

関数 `foo` で使用されるグローバル変数 `lookup_table` を RTL デザインでポートとして処理し、インターフェイスを `ap_memory` に指定します。

```
set_directive_interface -mode ap_memory foo lookup_table
```

set_directive_latency

説明

関数、ループ、または領域にレイテンシの最大値と最小値のいずれか、または両方を設定します。

Vivado HLS は常に最小レイテンシを目標にします。レイテンシの最大値および最小値を指定すると、Vivado HLS の動作は次のようになります。

- レイテンシが最小値未満。
指定した最小値よりも短いレイテンシが達成できる場合、Vivado HLS はリソースの共有率を上げるために、レイテンシを指定値まで拡張します。
- レイテンシが最小値より長い。
制約は満たされ、これ以上の最適化は実行されません。
- 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実行されません。
- レイテンシが最大値より長い。
Vivado HLS で最大値以下でスケジューリングできない場合、指定された制約を満たすことができるよう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、警告が表示され、Vivado HLS で達成可能な最小レイテンシでデザインが作成されます。

構文

```
set_directive_latency [OPTIONS] <location>
```

- `<location>`: 制約を設定する場所 (関数 ループ、または領域) を `function[/label]` の形式で指定します。

オプション

```
-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。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS latency \  
    min=<int> \  
    max=<int>
```

例

関数 `foo` のレイテンシの最小値を 4、最大値を 8 に指定します。

```
set_directive_latency -min=4 -max=8 foo  
#pragma HLS latency min=4 max=8
```

関数 `foo` にあるループ `loop_row` の最大レイテンシを 12 に指定します。プラグマはループ本体に記述します。

```
set_directive_latency -max=12 foo/loop_row  
#pragma HLS latency max=12
```

set_directive_loop_flatten

説明

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化します。

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。入れ子のループをフラットにすると、それらを 1 つのループとして最適化できるので、クロック サイクル数を削減でき、ループ本文のロジックをさらに最適化することが可能です。



推奨: この指示子は、ループ階層の最内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化できます。

- 完全ループの入れ子
 - 最内ループのみにループ本体の内容が含まれます。
 - ループ文の間に指定されるロジックはありません。
 - すべてのループ範囲は定数です。

- 半完全ループの入れ子
 - 最内ループのみにループ本体の内容が含まれます。
 - ループ文の間に指定されるロジックはありません。
 - 最外ループの範囲は変数にできます。

- 不完全ループの入れ子

内側のループの範囲が変数であったり、ループ本体が内側のループにのみ含まれているとは限らない場合、コードの構造を変更するか、ループ本体内のループを展開して、完全ループの入れ子を作成してみてください。

構文

```
set_directive_loop_flatten [OPTIONS] <location>
```

- <location>: 場所 (最内ループ) を function[/label] の形式で指定します。

オプション

```
-off
```

フラット化を実行しません。

指定ロケーション内の一部のループはフラットにせず、それ以外のループをフラットにできます。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS loop_flatten off
```

例

関数 `foo` 内の `loop_1` と、ループ階層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化します。プラグマを `loop_1` の本体に記述します。

```
set_directive_loop_flatten foo/loop_1
#pragma HLS loop_flatten
```

関数 `foo` の `loop_2` でループがフラット化されないようにします。プラグマを `loop_2` の本体に記述します。

```
set_directive_loop_flatten -off foo/loop_2
#pragma HLS loop_flatten off
```

set_directive_loop_merge

説明

すべてのループを 1 つのループに結合します。

ループを結合すると、次が可能になります。

- RTL でループ本文のインプリメンテーション間の移行に必要なクロック サイクル数を削減できます。
- ループを並列にインプリメントできます (可能な場合)。

ループ結合の規則は、次のとおりです。

- ループの境界が変数の場合、同じ値 (反復回数) である必要があります。
- ループの境界が定数の場合、最大定数値が結合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ループを結合することはできません。

- 結合するループ間のコードが、結合により悪影響を受けないようにします。このコードを複数回実行しても常に同じ結果になるようにする必要があります。
 - `a=b` は使用可能。
 - `a=a+1` は使用不可。
- ループに FIFO 読み出しが含まれる場合は、ループは結合できません。結合により読み出しの順序が変更されてしまうためです。FIFO または FIFO インターフェイスからの読み出しは、常に順序どおりに実行される必要があります。

構文

```
set_directive_loop_merge <location>
```

- `<location>`: ループのある場所を `function[/label]` の形式で指定します。

オプション

```
-force
```

Vivado HLS で警告が出力されても、ループが結合されます。結合したループが問題なく動作するかどうかは、ユーザーが確認する必要があります。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS loop_merge force
```

例

次の例では、関数 `foo` の連続するすべてのループを 1 つのループに結合しています。

```
set_directive_loop_merge foo
#pragma HLS loop_merge
```

関数 `foo` の `loop_2` 内にあるすべてのループ (`loop_2` を除くすべて) を `-force` オプションを使用して結合します。プラグマを `loop_2` の本体に記述します。

```
set_directive_loop_merge -force foo/loop_2
#pragma HLS loop_merge force
```

set_directive_loop_tripcount

説明

ループで実行される反復回数の合計を指定します。Vivado HLS により、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がレポートされます。ループ レイテンシは、トリップカウント (ループの反復回数) に依存します。

トリップカウントは、定数値であることもあり、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異なる場合もあります。

Vivado HLS でトリップカウントを決定できないことがあります。これは、トリップカウントの決定に使用される変数が次のような場合です。

- 入力引数。
- ダイナミック演算により計算される変数。

このような場合、ループ レイテンシが不明になることがあります。

`set_directive_loop_tripcount` コマンドを使用すると、ループのトリップカウントの最小値および最大値を指定し、最適化を駆動するデザイン解析を実行できるようにすることが可能です。これにより、ループ レイテンシがデザイン レイテンシ全体のどの程度を占めているかをレポートで確認できます。

構文

```
set_directive_loop_tripcount [OPTIONS] <location>
```

- `<location>`: トリップカウントを指定するループの場所を `function[/label]` の形式で指定します。

オプション

```
-avg <integer>
```

反復の平均回数を指定します。

```
-max <integer>
```

反復の最大回数を指定します。

```
-min <integer>
```

反復の最小回数を指定します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS loop_tripcount \
    min=<int> \
    max=<int>
```

例

関数 `foo` の `loop_1` は、次のように指定されています。

- 最小トリップカウント 12
- 最大トリップカウント 16

```
set_directive_loop_tripcount -min 12 -max 16 -avg 14 foo/loop_1
#pragma HLS loop_tripcount min=12 max=16 avg=14
```

set_directive_occurrence

説明

関数またはループをパイプライン処理する際に、ある場所のコードがそれを含む関数またはループのコードよりも低速で実行されるように指定します。

これにより、実行速度が遅いコード部分が低レートでパイプライン処理されるようにでき、最上位パイプライン内で共有できる可能性があります。次に例を示します。

- ループは N 回反復する。
- ループの一部は条件文で有効になり、M 回しか実行されない (N は M の整数倍)。
- 条件文で有効になるコードの実行頻度は N/M。

N が開始間隔 II でパイプライン処理される場合、条件文が設定されている関数/ループは、次のようになります。

- これよりも高い II の値でパイプライン処理される可能性があります。

注記:

つまり低速であり、このコードの実行頻度は低くなります。

- これを含む高いレートのパイプライン内の方がより良い共有が実行される可能性があります。

このような領域に実行頻度を指定すると、この領域内の関数およびループがそれを含む関数またはループよりも遅い開始間隔でパイプライン処理されます。

構文

```
set_directive_occurrence [OPTIONS] <location>
```

- <location>: 実行速度が遅い場所を指定します。

オプション

```
-cycle <int>
```

N/M の実行頻度を指定します。

- N: 領域を含む関数/ループの実行回数。
- M: 条件文領域の実行回数。

N は M の整数倍数である必要があります。

プリAGMA

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS occurrence cycle=<int>
```


例

関数 `foo` の領域 `Cond_Region` の実行頻度を 4 に指定します。この領域は、それを含むコードよりも 4 倍低速で実行されます。

```
set_directive_occurrence -cycle 4 foo/Cond_Region
#pragma HLS occurrence cycle=4
```

set_directive_pipeline

説明

次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 クロック 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、各クロック サイクルで新しい入力処理されます。-II オプションで開始間隔を指定することもできます。

Vivado HLS で指定した開始間隔でデザインを作成できない場合は、次が実行されます。

- 警告メッセージが表示されます。
- 達成可能な最短の開始間隔でデザインが作成されます。

警告メッセージを参考にデザインを解析し、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判断します。

構文

```
set_directive_pipeline [OPTIONS] <location>
```

説明:

- <location>: パイプライン処理が実行する場所を `function[/label]` の形式で指定します。

オプション

```
-II <integer>
```

パイプラインの開始間隔を指定します。

Vivado HLS では、この指定を満たすことが試みられます。データの依存性によって、実際の開始間隔はこの指定より大きくなる場合があります。

```
-enable_flush
```

パイプラインの入力で有効であったデータが非アクティブになった場合に、データをフラッシュして空にするパイプラインをインプリメントします。この機能は、パイプライン処理された関数でのみサポートされ、パイプライン処理されたループではサポートされません。

```
-rewind
```

注記: ループにのみ使用可能です。

巻き戻しをイネーブルにします。巻き戻しでは、1 つのループ反復の終了と次の反復の開始の間に一時停止のない連続ループ パイプライン処理が実行されます。

巻き戻しは、最上位関数内に 1 つのループしかない (完全なループ ネスト) 場合にのみ効果的です。ループ前のコード部分は、次のようになります。

- 初期化と認識されます。
- パイプラインで一度だけ実行されます。
- 条件文 (if-else) を含むことはできません。

```
-off
```

指定のループまたは関数のパイプライン処理をオフにします。config_compile -pipeline_loops でループのパイプラインがグローバルに指定されている場合に使用できます。このオプションを使用すると、指定したループがパイプライン処理されなくなります。

プリAGMA

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS pipeline \
    II=<int> \
    enable_flush \
```

例

関数 foo が開始間隔 1 でパイプライン処理されます。

```
set_directive_pipeline foo
#pragma HLS pipeline
```

set_directive_reset

説明

特定のステート変数 (グローバルまたはスタティック) のリセットを追加または削除します。

構文

```
set_directive_reset [OPTIONS] <location> <variable>
```

- <<location>> では、変数を定義する場所を function[/label] の形式で指定します。
- <<variable>> では、指示子を適用する変数を指定します。

オプション

```
-off
```

- -off を指定すると、指定した変数に対してリセットは生成されません。

- `-off` を指定しない場合、指定した変数に対してリセットが生成されます。

プラグマ

C ソース コードの変数のライフ サイクルの境界内に配置します。

```
#pragma HLS reset variable=a off
```

例

グローバル リセット設定が `none` または `control` の場合でも、関数 `foo` の変数 `static int a` にリセットを追加します。

```
set_directive_reset foo a  
#pragma HLS reset variable=a
```

グローバル リセット設定が `state` または `all` の場合でも、関数 `foo` の変数 `static int a` からリセットを削除します。

```
set_directive_reset -off foo a  
#pragma HLS reset variable=a off
```

set_directive_resource

説明

RTL に変数をインプリメントするためのリソース (コア) を指定します。変数は、次のいずれかになります。

- 配列
- 演算式
- 関数の引数

Vivado HLS では、ハードウェア コアを使用して演算がインプリメントされます。演算をインプリメントできるコアがライブラリに複数ある場合、`set_directive_resource` コマンドを使用して使用するコアを指定できます。コアのリストを生成するには、`list_core` コマンドを使用します。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判断されます。

配列をインプリメントするのに使用するライブラリのメモリ エlement を指定するには、`set_directive_resource` コマンドを使用します。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。配列に関連付けられているメモリによって RTL で使用されるポートが決まるので、これは最上位関数インターフェイスの配列には重要な方法です。

`-latency` オプションを使用すると、コアのレイテンシを指定できます。インターフェイスのブロック RAM の場合、`-latency` オプションを指定すると、たとえばレイテンシ 2 または 3 の SRAM をサポートするなど、インターフェイスにオフチップの標準でない SRAM を記述できます。内部演算の場合、`-latency` オプションを使用すると、演算をより多くのパイプライン段を使用してインプリメントできます。これらの追加のパイプライン段により、RTL 合成中にタイミング問題を解決しやすくなります。



重要: `-latency` オプションを使用するには、使用可能な複数段のコアを演算に含める必要があります。Vivado HLS には、基本的な算術演算 (加算、減算、乗算、除算)、すべての浮動小数点演算、およびすべてのブロック RAM 用に複数段コアが含まれています。



推奨: より良い結果を得るため、ザイリンクスでは、C の場合は `-std=c99` を、C および C++ の場合は `-fno-builtin` を使用することを勧めます。`-std=c99` などの C コンパイル オプションを指定するには、Tcl コマンドの `add_files` に `-cflags` オプションを使用します。または、[Project Settings] ダイアログ ボックスの [Edit CFLAGS] ボタンを使用します。

構文

```
set_directive_resource -core <string> <location> <variable>
```

- `<location>`: 変数のある場所を `function[/label]` の形式で指定します。
- `<variable>`: 変数を指定します。

オプション

```
-core <string>
```

テクノロジー ライブラリで定義されているのと同じように、コアの名前を指定します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS resource \
    variable=<variable> \
    core=<core>
    latency=<latency>
```

例

変数 `coeffs[128]` は最上位関数 `foo_top` への引数です。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 コアで定義されます。

```
set_directive_resource -core RAM_1P foo_top coeffs
#pragma HLS resource variable=coeffs core=RAM_1P
```

関数 `foo` に `Result=A*B` というコードがある場合に、乗算が 2 段パイプライン乗算器コアを使用してインプリメントされるように指定します。

```
set_directive_resource -latency 2 foo Result
#pragma HLS RESOURCE variable=Result latency=2
```

URAM を使用してメモリをインプリメントするには、次のように指定します。

```
#pragma HLS RESOURCE variable=array core=RAM_1P_URAM uram
```

set_directive_stable

説明

`stable` プラグマは、データフロー領域の開始と終了時に同期を生成する際に、変数 (データフロー領域の入力または出力) を無視できることを示すために使用されます。

構文

```
set_directive_stable <location> <variable>
```

- <location> は、指示子が制約される関数名またはループ名です。
- <variable> は、制約される配列の名前です。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS stable variable=A
```

例

次の例の場合、stable プラグマがないと、proc1 および proc2 がその入力 (A も含む) の読み出しを承認するために同期されます。stable プラグマがあると、A は同期の必要な入力としては認識されなくなります。

```
void dataflow_region(int A[...], int B[...] ...  
#pragma HLS stable variable=A  
#pragma HLS dataflow  
    proc1(...);  
    proc2(A, ...);
```

set_directive_stream

説明

デフォルトでは、配列変数は RAM 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 インターフェイスの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するサブ関数では、配列引数は RAM のピンポン バッファ 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する配列は、RAM のピンポン バッファ チャネルを使用してインプリメントされます。

配列に格納されているデータが順次に消費または生成される場合は、RAM ではなく FIFO を使用し、ストリーミングデータを使用するほうが効率的です。

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としてインプリメントされます。



重要: アクセスを保持するには、volatile 修飾子を使用して (特定のデッド コード削除で) コンパイラ最適化がされないようにする必要があります。

構文

```
set_directive_stream [OPTIONS] <location> <variable>
```

- <location>: 配列変数を含む場所を `function[/label]` の形式で指定します。

- `<variable>`: FIFO としてインプリメントする配列変数を指定します。

オプション

```
-depth <integer>
```

注記: DATAFLOW チャンネルの配列ストリーミングにのみ適用されます。

RTL にインプリメントされる FIFO の深さは、デフォルトでは C コードで指定した配列と同じサイズになります。このオプションを使用すると、FIFO のサイズを変更できます。

配列が DATAFLOW 領域にインプリメントされる場合は、`-depth` オプションで FIFO のサイズを削減する方法がよく使用されます。たとえば、DATAFLOW 領域ですべてのループおよび関数がデータを $II=2$ のレートで処理する場合、データはクロック サイクルごとに生成および消費されるので、大型 FIFO は必要ありません。この場合、`-depth` オプションで FIFO サイズを 2 に削減すると、RTL デザインでエリアを削減できます。

これと同じ機能は、`config_dataflow` コマンドで `-depth` オプションを指定すると、DATAFLOW 領域のすべての配列に使用できます。`set_directive_stream` に `-depth` オプションを使用すると、`config_dataflow` を使用して指定したデフォルト設定を上書きできます。

```
-dim <int>
```

ストリーミングする配列の次元を指定します。デフォルトは次元 1 です。1 次元配列の場合は `dim` を 1 に、2 次元配列の場合は `dim` を 2 に設定します。

次元は、データフロー領域内のコンシューマーとプロデューサーのモデル間にあるデザイン内部のストリームのみ指定できます。デザイン インターフェイスでのストリームには適用できません。

```
-off
```

注記: DATAFLOW チャンネルの配列ストリーミングにのみ適用されます。

`config_dataflow -default_channel fifo` コマンドを使用すると、デザインのすべての配列に `set_directive_stream` がグローバルに適用されます。このオプションを使用すると、指定の配列でストリーミングをオフにでき、デフォルトが RAM ピンポン バッファ ベースのチャンネルに戻ります。

注記: `-off` オプションを選択する場合は、`-depth` オプションでピンポンの深さ (ブロック数) を設定します。深さは少なくとも 2 にする必要があります。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS stream
    variable=<variable> \
    off \
    depth=<int>
```

例

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_stream foo A
#pragma HLS STREAM variable=A
```

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します。この場合、プラグマは `loop_1` 内に挿入する必要があります。

```
set_directive_stream -depth 12 foo/loop_1 B
#pragma HLS STREAM variable=B depth=12
```

次の例では、配列 `C` のストリーミングをディスエーブルにしています。この例では、ストリーミングが `config_dataflow` でイネーブルになっていると想定しています。

```
set_directive_stream -off foo C
#pragma HLS STREAM variable=C off
```

set_directive_top

説明

関数に名前を付けます。指定した名前は、`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。



推奨: 指示子は、アクティブ ソリューションで指定します。この後、その新しい名前を `set_top` コマンドで使用します。

構文

```
set_directive_top [OPTIONS] <location>
```

- `<location>`: 名前を変更する関数を指定します。

オプション

```
-name <string>
```

`set_top` コマンドで使用する名前を指定します。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS top \
    name=<string>
```

例

関数 `foo_long_name` の名前を `DESIGN_TOP` に変更し、最上位として指定します。コード内にプラグマが含まれる場合でも、GUI プロジェクト設定で指定した最上位で `set_top` コマンドを実行する必要があります。

```
set_directive_top -name DESIGN_TOP foo_long_name
#pragma HLS top name=DESIGN_TOP
set_top DESIGN_TOP
```

set_directive_unroll

説明

ループ本体のコピーを複数作成することによりループを変換します。

ループは、ループ帰納変数で指定されている反復回数実行されます。反復回数は、break やループの exit 変数の変更など、ループ本体内のロジックにも影響します。ループは、ループ本題を表すロジックのブロックで RTL にインプリメントされ、同じ反復回数実行されます。

set_directive_unroll コマンドを使用すると、ループを完全に展開できます。ループを展開すると、RTL にループの反復回数と同じ数のループ本文のコピーが作成されます。または、ループを係数 N で部分展開し、ループ本文の N 個のコピーを作成してループ反復回数を調整できます。

部分展開に使用される係数 N が元のループ反復回数の整数倍でない場合は、ループ本文の展開された部分の後の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。

構文

```
set_directive_unroll [OPTIONS] <location>
```

- <location>: 展開するループの場所を function[/label] の形式で指定します。

オプション

```
-factor <integer>
```

以外の整数値を指定して、部分展開が実行されるようにします。

ループ本文は、ここで指定した回数分繰り返され、それに応じて反復回数が調整されます。

```
-region
```

ループ内にあるすべてのループが展開されますが、それを含むループ自体は展開されません。

次のような例があるとします。

- ループ loop_1 内の同じループ階層に loop_2 および loop_3 というループあります。
- 指定するループ (loop_1 など) はコード内の領域またはロケーションでもあります。
- コードのセクションは、{ } かっこで囲まれます。
- unroll 指示子をロケーション <function>/loop_1 に指定すると、loop_1 が展開されます。

-region オプションを使用すると、指示子は指定した領域を含むループにのみ適用されます。これにより、次のような結果になります。

- loop_1 は展開されないままになります。
- loop_1 内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
```


-factor を指定した場合 (部分展開) にのみ有効です。

- [Fixed bounds]

反復回数が係数の倍数である場合は、終了条件はチェックされません。

反復回数が係数の整数倍でない場合は、次のようになります。

- ・ 展開は実行されません。
- ・ 処理を続行するには終了チェックを実行する必要があることを示す警告メッセージが表示されます。

- [Variable bounds]

終了条件チェックが削除されます。次を確認してください。

- 。 可変境界が係数の整数倍である。
- 。 終了チェックが不要である。

プラグマ

C ソースの必要なロケーションの境界内に配置します。

```
#pragma HLS unroll \
    skip_exit_check \
    factor=<int> \
    region
```

例

関数 `foo` 内のループ `L1` を展開します。プラグマは `L1` の本体に記述します。

```
set_directive_unroll foo/L1
#pragma HLS unroll
```

関数 `foo` のループ `L2` の展開係数を 4 に指定します。最終チェックを削除します。プラグマは `L2` の本体に記述します。

```
set_directive_unroll -skip_exit_check -factor 4 foo/L2
#pragma HLS unroll skip_exit_check factor=4
```

関数 `foo` のループ `L3` 内にあるループをすべて展開しますが、ループ `L3` 自体は展開しません。-region オプションは、ループ ラベルではなく、ループを含む領域として考慮される場所を指定します。

```
set_directive_unroll -region foo/L3
#pragma HLS unroll region
```

set_part

説明

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 ソリューションのコンテキストでのみ実行可能です。

構文

```
set_part <device_specification>
```

- `<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 デバイスを設定するデバイス仕様を指定します。
- `<device_family>`: デバイス ファミリ名を指定します。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、パッケージ、スピード グレード情報を指定します。

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

次の例に示すようにデバイス ファミリ名を指定すると、Vivado HLS に含まれる FPGA ライブラリを現在のソリューションに追加できます。この場合、Vivado HLS の FPGA ライブラリでこのデバイス ファミリに対して指定されている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 virtex7
```

Vivado HLS に含まれる FPGA ライブラリでは、特定のデバイスをパッケージおよびスピード グレード情報と共に指定することもできます。

```
set_part xc6v1x240tff1156-1
```

set_top

説明

合成す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に含まれます。

構文

```
set_top <top>
```

- `<top>`: 合成する関数を指定します。

オプション

このコマンドにはオプションはありません。

プラグマ

同等のプラグマはありません。

例

foo_top を最上位関数として設定します。

```
set_top foo_top
```

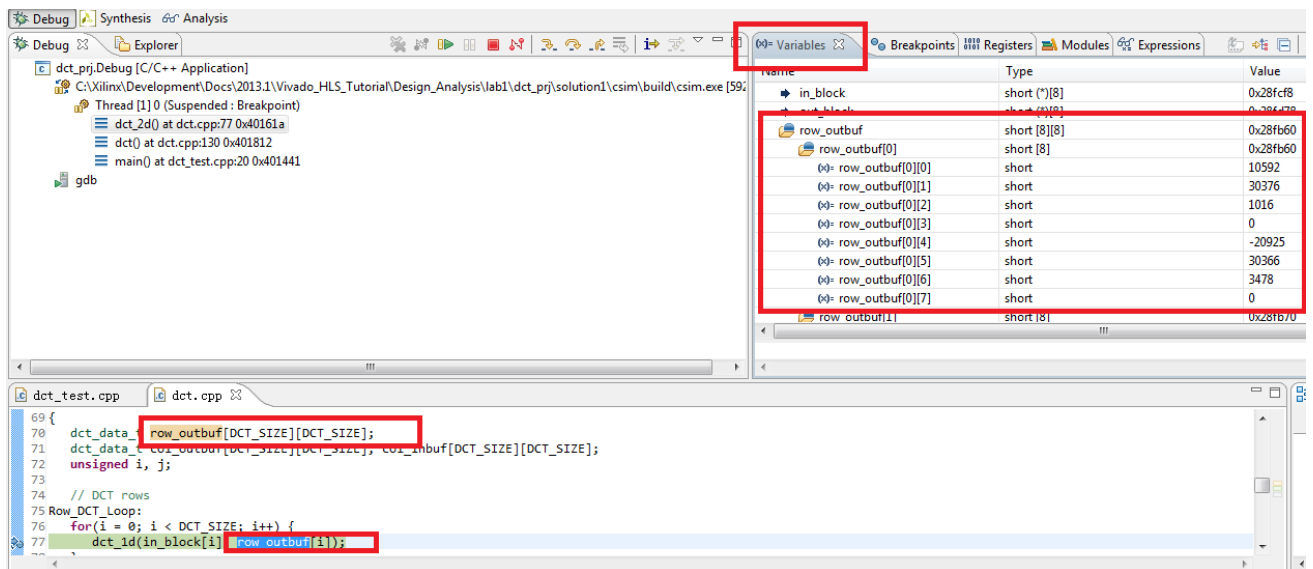
GUI リファレンス

このリファレンス セクションでは、Vivado HLS の GUI の使用、制御、カスタマイズ方法について説明します。

変数の確認

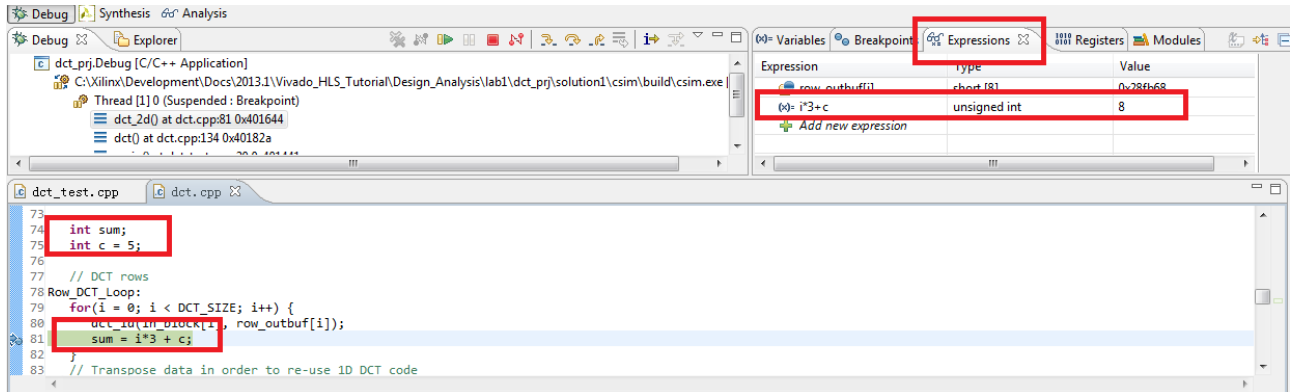
変数と式の値は、[Debug] パースペクティブで直接確認できます。次の図に、各変数の値の確認方法を示します。

図 94: 変数の確認



式の値は、[Expressions] ビューを使用して確認できます。

図 95: 演算式の確認

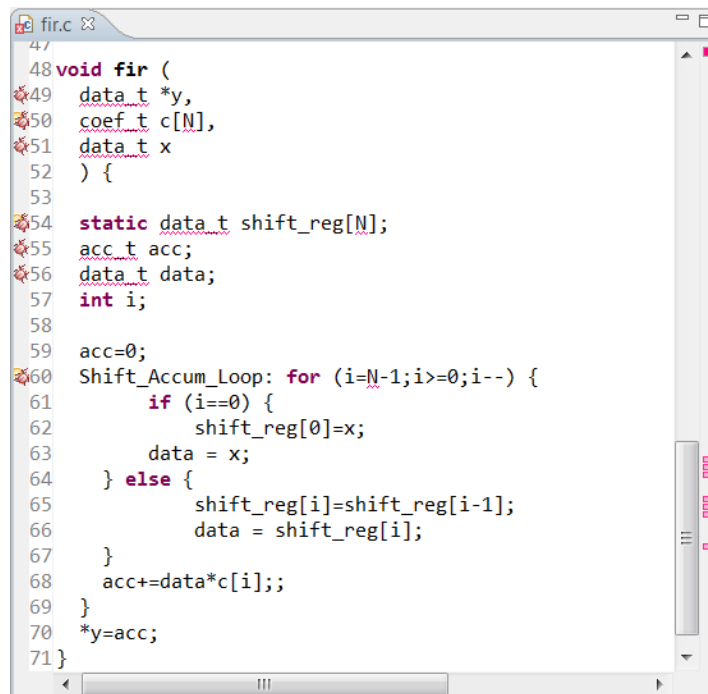


ヘッダー ファイル情報の解決

デフォルトの Vivado HLS の GUI では、すべてのコード参照を解決するために、すべてのヘッダー ファイルが解析されます。次の図に示すように、未解決の参照は GUI でハイライトされます。

- 左側のサイドバー: 現在のビューで未定義の参照がマークされます。
- 右側のサイドバー: ファイル全体で未解決の参照がマークされます。

図 96: C ファイルのインデックス



重要: C シミュレーションまたは合成を実行する前に、コードから未定義の参照を削除しておくことが重要です。未定義の参照をチェックするには、コード ビューアーに示される変数または値が不明または定義できないことを示す表示を確認してください。未定義の参照は、指示子ウィンドウに表示されません。

ヘッダー ファイルで定義されたコードを解決できない場合、未定義の参照が発生します。未定義の参照の主な原因は、次のとおりです。

- コードが最近ファイルに追加された。

コードが新しい場合、ヘッダー ファイルが保存されていることを確認します。ヘッダー ファイルを保存すると、Vivado HLS でヘッダー ファイルに自動的にインデックスが付けられ、コード参照がアップデートされます。

- ヘッダー ファイルが検索パスに含まれない。

ヘッダー ファイルが `include` 文を使用して C コードに含まれ、ヘッダー ファイルへのディレクトリが検索パスに含まれ、ヘッダー ファイルがプロジェクトに追加された C ファイルと同じディレクトリに含まれていることを確認します。

注記: 検索パスを明示的に追加するには、[Solution]→[Solution Settings] をクリックし、[Synthesis] または [Simulation] をクリックして [Edit CFLAGS] ボタンを使用します。詳細は、[新規合成プロジェクトの作成](#)を参照してください。

- 自動インデックスがディスエーブル。

Vivado HLS ですべてのヘッダー ファイルが自動的に解析されるようにします。[Project > Project Settings] をクリックして [Project Settings] ダイアログ ボックスを開きます。[General] をクリックし、次の図に示すように [Disable Parsing All Header Files] をオフにします。これにより、Vivado HLS で CPU サイクルを使用してヘッダー ファイルが自動的に確認されるので、GUI の反応時間は短くなります。


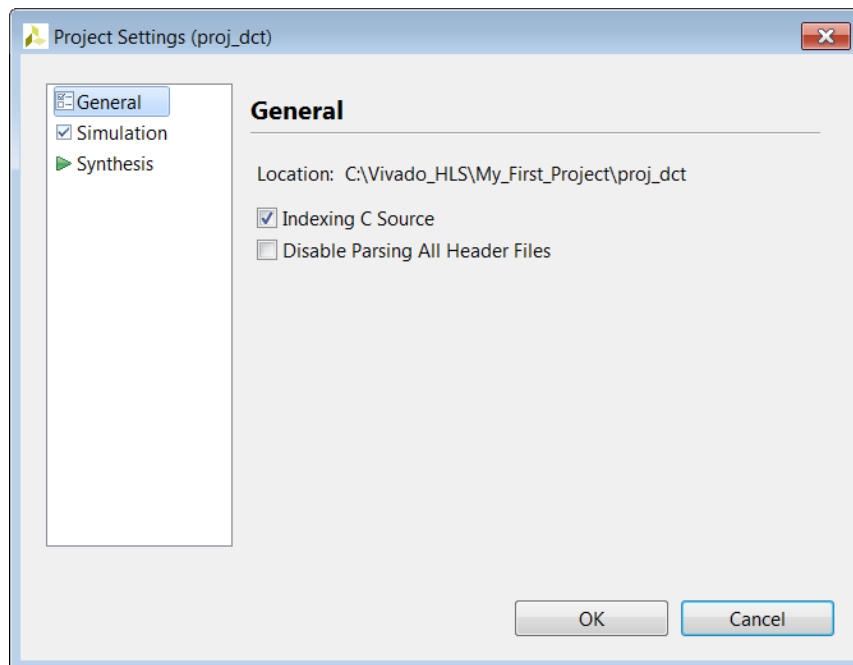
注記: Vivado HLS ですべての C ファイルにインデックスが付けられるように手動で指定するには、[Index C files] ツールバー ボタン  をクリックします。

図 97: ヘッダー ファイルの解析を制御



ソース コードのコメント表示の解決

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] ビューでプロジェクトを選択します。
2. 右クリックして [Properties > Resource] 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ドを選択します。

GUI のカスタマイズ

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かったり、デザインに適さない設定であったりすることがあります。このセクションでは、次のカスタマイズ方法について説明します。

- [Console] ビューのバッファ サイズ。
- デフォルトの主な動作。

[Console] ビューのカスタマイズ

[Console] ビューには、合成および検証などの操作中のメッセージがすべて表示されます。

このビューのデフォルトのバッファ サイズは 80,000 文字ですが、これを変更するか、またはすべてのメッセージが表示されるように制限を解除できます。これは、[Window] → [Preferences] → [Run/Debug] → [Console] をクリックすると設定できます。

主な動作のカスタマイズ

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。

たとえば、キーの組み合わせ [Ctrl] + [Tab] を使用すると、デフォルトでは情報エリアのアクティブ ビューがソース コードとヘッダー ファイル間で切り替わります。[Ctrl] + [Tab] をもう一度押すと、アクティブなタブが再び切り替わります。

- [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、Ctrl + Tab の組み合わせを削除します。
- [Command] 列で [Next Tab] を選択し、[Binding] ダイアログ ボックスにカーソルを置いて [Ctrl] キーを押しながら [Tab] キーを押します。これで、[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されます。

次の検索結果を示すホット キーは、Microsoft Visual Studio スキームを使用してインプリメントできます。[Window] → [Preference] → [General] → [Keys] で Default スキームを Microsoft Visual Studio スキームに変更します。

[Preferences] メニューの左側のペインから各項目を選択することにより、GUI 環境を詳細にカスタマイズでき、生産性を向上できます。

インターフェイス合成リファレンス

このセクションでは、Vivado HLS のインターフェイス プロトコル モードについて説明します。

ブロック レベル I/O プロトコル

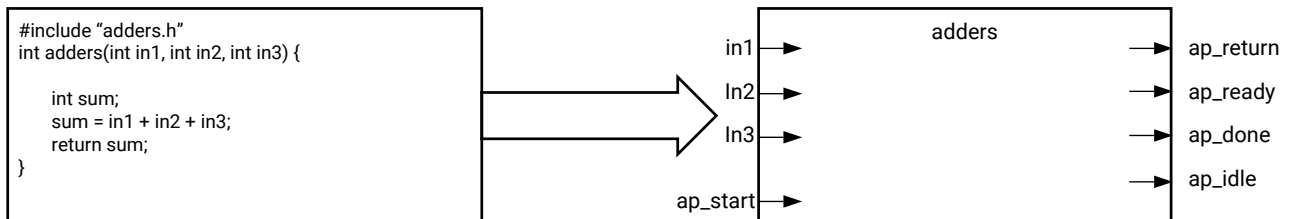
Vivado HLS では、ap_ctrl_none、ap_ctrl_hs、および ap_ctrl_chain インターフェイスを使用して、RTL をブロック レベルのハンドシェイク信号を含めてインプリメントするかどうかを指定します。ブロック レベルのハンドシェイク信号では、次が指定されます。

- デザインが演算の実行を開始するタイミング
- 演算が終了するタイミング
- デザインがアイドル状態になって新しい入力に対する準備ができるタイミング

これらのブロックレベルの I/O プロトコルは関数または関数の戻り値に指定できます。C コードで値が返されない場合でも、関数の戻り値にブロックレベルの I/O プロトコルを指定できます。C コードで関数の戻り値が使用される場合は、Vivado HLS は戻り値用に `ap_return` 出力ポートを作成します。

デフォルトは `ap_ctrl_hs` ブロックレベル I/O プロトコルです。次の図に、Vivado HLS で関数に `ap_ctrl_hs` がインプリメントされる場合の結果の RTL ポートと動作を示します。この例では、関数に `return` 文を使用して値を返しており、Vivado HLS で RTL デザインに `ap_return` 出力ポートが作成されます。関数の `return` 文が C コードに含まれていない場合は、このポートは作成されません。

図 98: `ap_ctrl_hs` インターフェイスの例



X14267

`ap_ctrl_chain` インターフェイス モードは `ap_ctrl_hs` と似ていますが、バックプレッシャーを適用するために入力信号 `ap_continue` が追加されている点が異なります。Vivado HLS ブロックをチェーン接続する場合は、`ap_ctrl_chain` ブロックレベル I/O プロトコルを使用することをお勧めします。

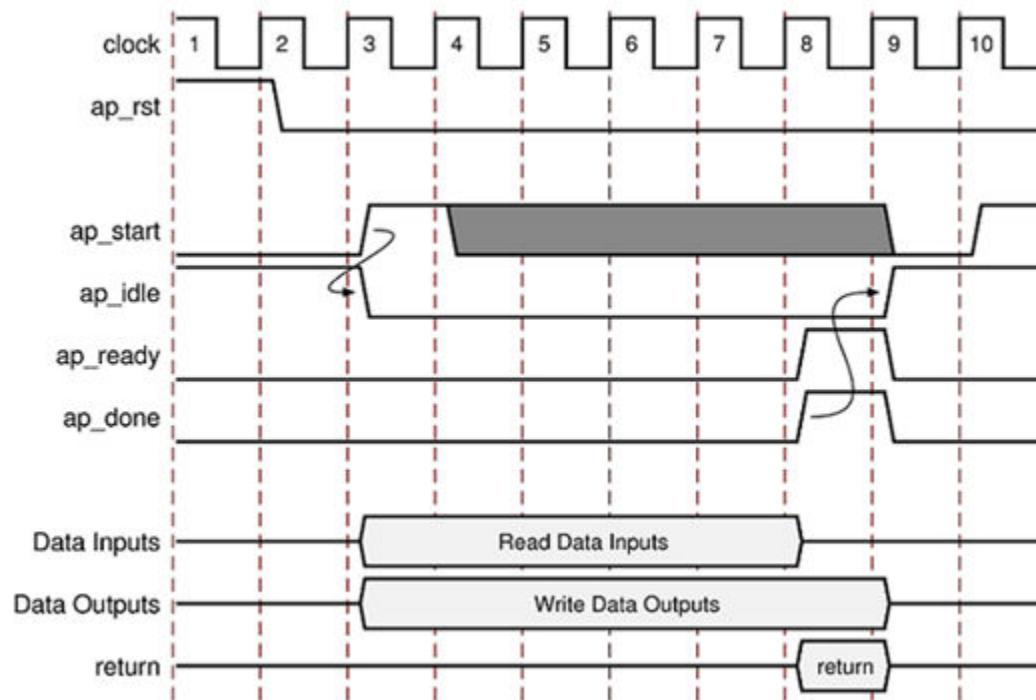
`ap_ctrl_none`

`ap_ctrl_none` ブロックレベル I/O プロトコルを指定する場合は、[ブロックレベル I/O プロトコル](#) に示すハンドシェイク信号ポート (`ap_start`、`ap_idle`、`ap_ready`、および `ap_done`) は作成されません。ブロックレベル I/O プロトコルをデザインに指定しない場合、C/RTL 協調シミュレーションを使用して RTL デザインを検証する際に、[インターフェイス合成要件](#)の条件に従ってください。

`ap_ctrl_hs`

次の図に、パイプライン処理されていないデザインの `ap_ctrl_hs` I/O プロトコルで作成されるブロックレベルハンドシェイク信号の動作を示します。

図 99: ap_ctrl_hs インターフェイスの動作



リセット後は、次ようになります。

1. ap_start が High になるとブロックが操作を開始します。
2. ap_idle 出力は即座に Low になり、デザインがアイドル状態でなくなったことを示します。
3. ap_start 信号は ap_ready が High になるまで High のままである必要があります。ap_ready が High になると、次のようになります。
 - ap_start が High のままの場合、次のトランザクションを開始します。
 - ap_start が Low になると、現在のトランザクションが完了して、操作を停止します。
4. 入力ポートからデータを読み出せるようになります。

注記: 入力ポートでは、このブロック レベル I/O プロトコルからは独立した、ポート レベル I/O プロトコルを使用できます。詳細は、[ポート レベル I/O プロトコル](#)を参照してください。
5. 出力ポートにデータを書き込めるようになります。

注記: 出力ポートでは、このブロック レベル I/O プロトコルからは独立した、ポート レベル I/O プロトコルを使用できます。詳細は、[ポート レベル I/O プロトコル](#)を参照してください。
6. ブロックの操作が完了すると、ap_done 出力が High になります。

注記: ap_return ポートがある場合、このポートの値は ap_done が High になると有効になります。つまり、ap_done 信号は ap_return 出力のデータが有効であることも示します。
7. デザインが新しい入力を受信可能な状態になると、ap_ready 信号が High になります。次に、ap_ready 信号に関する追加情報を示します。
 - ap_ready 信号はデザインが動作を始めるまで非アクティブ (Low) です。
 - パイプライン処理されていないデザインでは、ap_ready 信号は ap_done と同時にアサートされます。

- パイプライン処理されたデザインでは、`ap_start` が High になった後のどのサイクルでも `ap_ready` 信号が High になる可能性があります。これは、デザインがどのようにパイプライン処理されたかによって異なります。
 - `ap_ready` が High のときに `ap_start` が Low になると、デザインは `ap_done` が High になるまで実行されてから停止します。
 - `ap_ready` が High のときに `ap_start` が High になると、次のトランザクションが即座に開始され、デザインが動作を続行します。
8. `ap_idle` は、デザインがアイドル状態で動作していないことを示します。次に、`ap_idle` 信号に関する追加情報を示します。
- `ap_ready` が High のときに `ap_start` が Low になると、デザインは動作を停止し、`ap_idle` は `ap_done` の 1 サイクル後に High になります。
 - `ap_ready` が High のときに `ap_start` が High になると、デザインは動作を継続し、`ap_idle` は Low のままになります。

ap_ctrl_chain

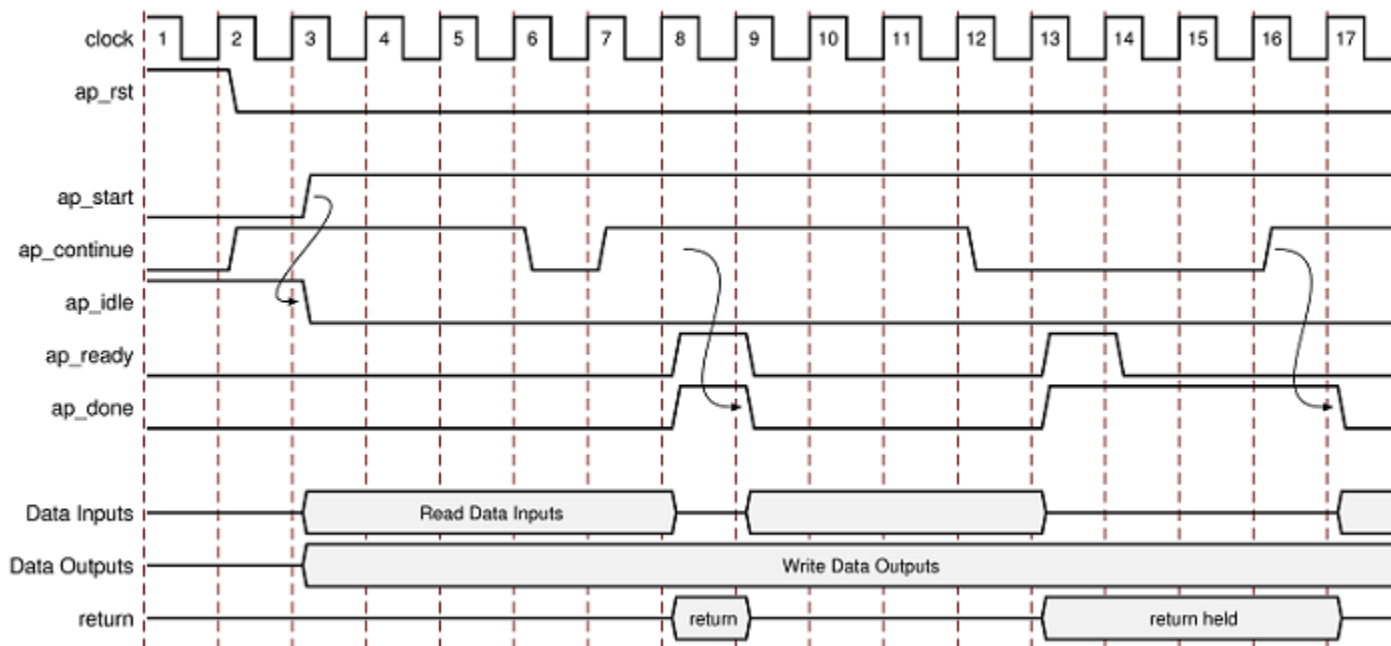
`ap_ctrl_chain` ブロックレベル I/O プロトコルは `ap_ctrl_hs` と似ていますが、`ap_continue` ポートが追加されている点が異なります。`ap_continue` 信号がアクティブ High の場合、出力データを使用するダウンストリームブロックが新しいデータ入力を読み出す準備ができたことを示します。ダウンストリームブロックで新しいデータ入力を消費できない場合は、`ap_continue` が Low になり、アップストリームで追加データが生成されなくなります。

ダウンストリームブロックの `ap_ready` ポートは `ap_continue` ポートを直接駆動できます。次に、`ap_continue` ポートに関する追加情報を示します。

- `ap_done` が High のときに `ap_continue` が High の場合、デザインは動作を継続します。それ以外のブロックレベル I/O 信号の動作は、`ap_ctrl_hs` ブロックレベル I/O プロトコルと同じです。
- `ap_done` が High のときに `ap_continue` 信号が Low の場合、`ap_done` 信号は High のままになり、`ap_return` ポートがある場合は `ap_return` ポートのデータは有効なままになります。

次の図では、`ap_done` が High で `ap_continue` が High なので、最初のトランザクションの終了直後に、2 つ目のトランザクションが開始しますが、`ap_continue` が High にアサートされなければ、2 つ目のトランザクションが終了すると停止します。

図 100: ap_ctrl_chain インターフェイスの動作



ポート レベル I/O プロトコル

ap_none

`ap_none` ポート レベル I/O プロトコルは最も単純なインターフェイス タイプで、ほかの信号は関連付けられません。入力データ信号にも出力データ信号にも、データの読み出しまたは書き込みをいつ実行するかを示す制御ポートは含まれません。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のみです。

`ap_none` インターフェイスに追加のハードウェア オーバーヘッドは必要ありません。ただし、`ap_none` インターフェイスには次が必要です。

- 次のいずれかを実行するプロデューサー ブロック:
 - 正しい時間に入力ポートにデータを供給
 - トランザクション中デザインが終了するまでデータを保持
- 正しい時間に出力ポートを読み出すコンシューマー ブロック

注記: `ap_none` インターフェイスを配列引数に使用することはできません。

ap_stable

`ap_none` と同様に、`ap_stable` ポート レベル I/O プロトコルでは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、コンフィギュレーション データを供給するポートなど、変化することはないが通常の動作中は安定していて変化しないデータに使用されます。`ap_stable` タイプを使用すると、Vivado HLS に次の情報が伝えられます。

- ポートに適用されるデータが通常の動作中は安定していて変化しませんが、最適化可能な定数値ではありません。
- このポートからのファンアウトにはレジスタを付ける必要はありません。

注記: `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)

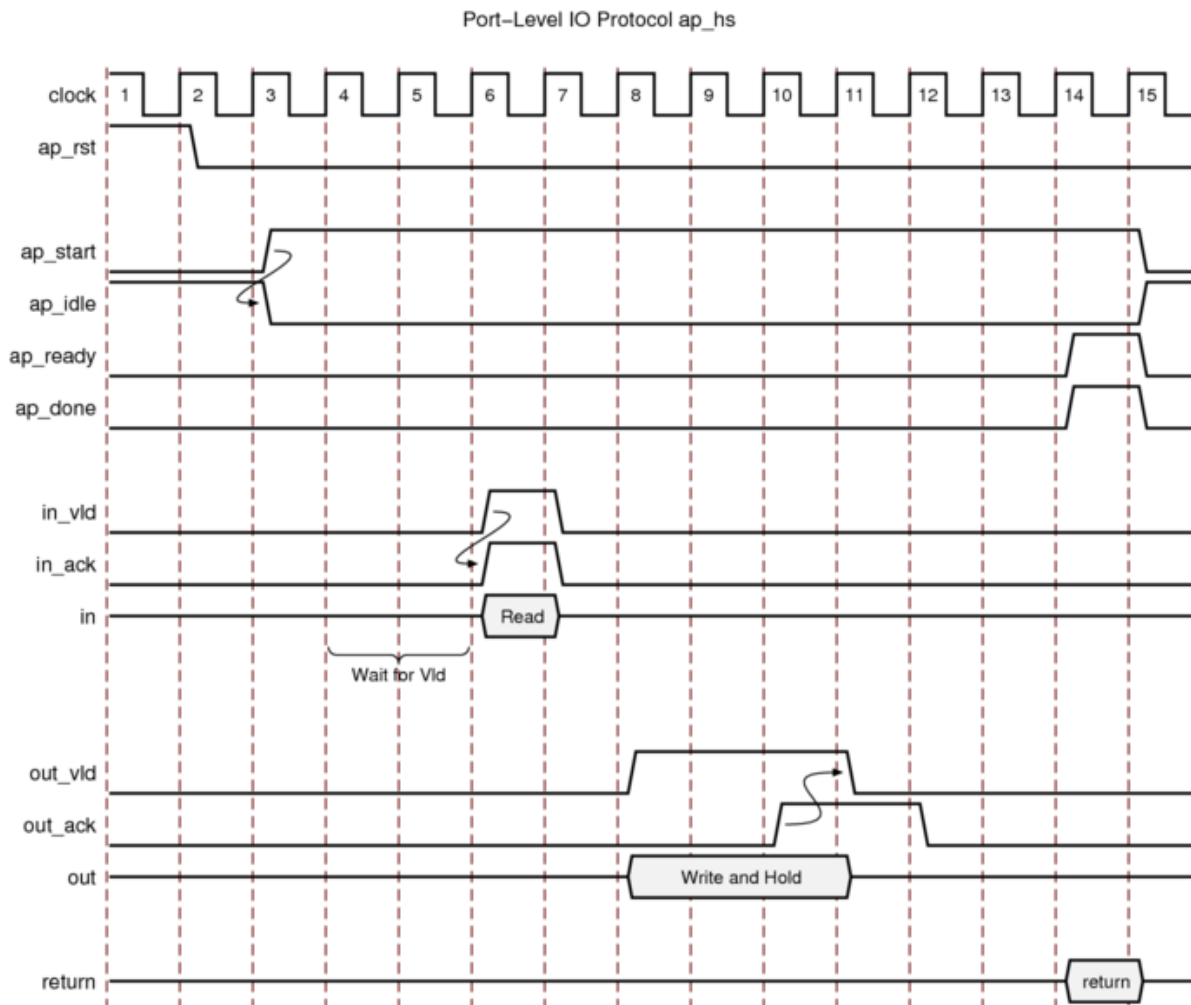
`ap_hs` ポート レベル I/O プロトコルを使用すると、開発プロセス中の柔軟性が最大となり、ボトムアップおよびトップダウン デザイン フローの両方が可能になります。2 方向ハンドシェイクによりすべてのブロック間の通信が安全に実行されるので、正しい動作に手動の介入や想定は必要ありません。`ap_hs` ポート レベル I/O プロトコルには、次の信号が含まれます。

- データ ポート
- データが受信されたことを示す肯定応答 (ACK) 信号
- データが読み出されたことを示す Valid 信号

次の図に、入力ポートと出力ポートでの `ap_hs` インターフェイスの動作を示します。この例では、入力ポート名は `in`、出力ポート名は `out` です。

注記: 制御信号名は、元のポート名に基づいています。たとえば、データ入力 `in` の Valid ポート名は `in_vld` です。

図 101: ap_hs インターフェ이스の動作



入力では、次のようになります。

- **ap_start** が High になると、ブロックが通常の動作を開始します。
- デザインが入力データを読み出す準備ができて、**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。

注記: 前の図は、この動作を示しています。この例では、デザインはクロック サイクル 4 でデータ入力 **in** を読み出す準備ができ、データを読み出す前に **in_vld** を待ちます。

- **in_vld** 入力が High になると、**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。

出力では、次のようになります。

- **ap_start** が High になると、ブロックが通常の動作を開始します。
- 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。
- **in_ack** 入力が Low の場合、デザインが停止し、**in_ack** 入力が High になるまで待機します。
- **in_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_ack

ap_ack ポート レベル I/O プロトコルは ap_hs インターフェイス タイプのサブセットです。ap_ack ポート レベル I/O プロトコルには、次の信号が含まれます。

- データ ポート
- データが受信されたことを示す肯定応答 (ACK) 信号
 - 入力引数に対しては、出力 ACK ポートが生成され、入力を読み出されるサイクルで High になります。
 - 出力引数に対しては、Vivado HLS で入力 ACK ポートがインプリメントされ、出力を読み出されたことが確認されます。

注記: 書き込み操作後、デザインが停止し、ACK 入力 が High になって出力がコンシューマー 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。



注意: C/RTL 協調シミュレーションでは、出力ポートに ap_ack を使用するデザインを検証できません。

ap_vld

ap_vld は ap_hs インターフェイス タイプのサブセットです。ap_vld ポート レベル I/O プロトコルには、次の信号が含まれます。

- データ ポート
- データを読み出されたことを示す Valid 信号
 - 入力引数に対しては、デザインは Valid 信号が High になった直後にデータ ポートを読み出します。デザインが新しいデータを読み出す準備ができていなくても、データ ポートがサンプリングされ、必要になるまでデータが内部で保持されます。
 - 出力引数に対しては、Vivado HLS で出力 valid ポートがインプリメントされ、出力ポートのデータが有効になったことが示されます。

ap_ovld

ap_ovld は ap_hs インターフェイス タイプのサブセットです。ap_ovld ポート レベル I/O プロトコルには、次の信号が含まれます。

- データ ポート
- データを読み出されたことを示す Valid 信号
 - 入力引数と、入出力引数の入力部分に対しては、デザインはデフォルトで ap_none タイプになります。
 - 出力引数と、入出力引数の出力部分に対しては、デザインは ap_vld タイプをインプリメントします。

ap_memory、bram

ap_memory および bram インターフェイス ポート レベル I/O プロトコルは、配列引数をインプリメントするために使用されます。このタイプのポート レベル I/O プロトコルは、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LEMENT (RAM、ROM など) と通信するために使用されます。

注記: メモリ エLEMENT への順次アクセスのみが必要な場合は、ap_fifo インターフェイスを使用してください。ap_fifo インターフェイスを使用すると、アドレス生成は実行されないため、ハードウェア オーバーヘッドが削減します。

ap_memory と bram のインターフェイス ポート レベル I/O プロトコルは同じで、Vivado IP インテグレーターでのブロックの表示方法だけが異なります。

- ap_memory インターフェイスは個別のポートとして表示される。
- bram インターフェイスは 1 つのポートとして表示される。IP インテグレーターでは、1 つの接続を使用して、すべてのポートへの接続を作成できます。

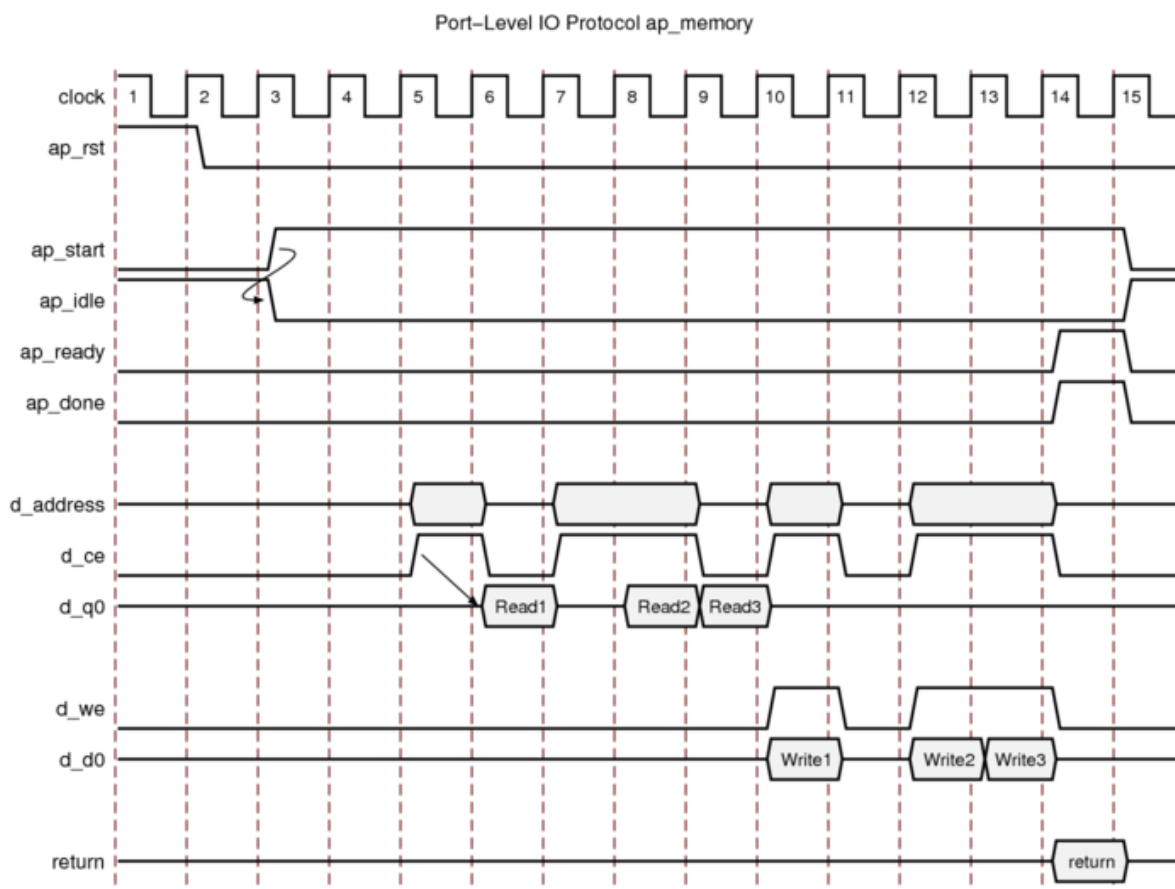
ap_memory インターフェイスを使用する場合は、RESOURCE 指示子を使用して配列ターゲットを指定してください。配列に対してターゲットを指定しない場合は、Vivado HLS でシングル ポートまたはデュアル ポート RAM インターフェイスのいずれかが指定されます。



ヒント: 合成を実行する前に、配列引数が RESOURCE 指示子を使用して正しいメモリ タイプをターゲットにしているかどうかを確認してください。変更されたメモリを使用して合成し直すと、スケジューリングおよび RTL の結果が異なることがあります。

次の図では、d という配列がシングル ポート ブロック RAM として指定されています。ポート名は、C 関数引数に基づいています。たとえば、C 関数が d の場合、チップ イネーブルは d_ce、入力データは BRAM の output/q ポートに基づいて d_q0 になります。

図 102: ap_memory インターフェイスの動作



リセット後は、次のようになります。

- ap_start が High になると、ブロックが通常の動作を開始します。

- 出力信号 `d_ce` をアサートしたときに出力アドレス ポートにアドレスを供給することにより、読み出しが実行されます。

注記: デフォルトのブロック RAM では、入力データ `d_q0` が次のクロック サイクルで有効になると想定されます。RESOURCE 指示子を使用すると、RAM の読み出しレイテンシを長くできます。

- 書き込み操作は、出力ポート `d_ce` と `d_we` がアサートされ、同時にアドレスと出力データ `d_d0` が供給されると実行されます。

ap_fifo

出力ポートが書き込まれると、デザインがメモリ エlementにアクセスする必要があり、そのアクセスが常に順番に実行される、つまりランダム アクセスが必要ない場合、`ap_fifo` に関連する出力 `valid` 信号インターフェイスを使用するのが最もハードウェア効率の良いアプローチです。`ap_fifo` ポート レベル I/O プロトコルでは、次がサポートされます。

- ポートを FIFO に接続できるようになる
- 完全な 2 方向の empty-full 通信を可能にする
- 配列、ポインター、参照渡し引数タイプに使用できる

注記: `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。[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、このコーディング スタイルを使用する場合の `volatile` 修飾子の重要性を理解してください。

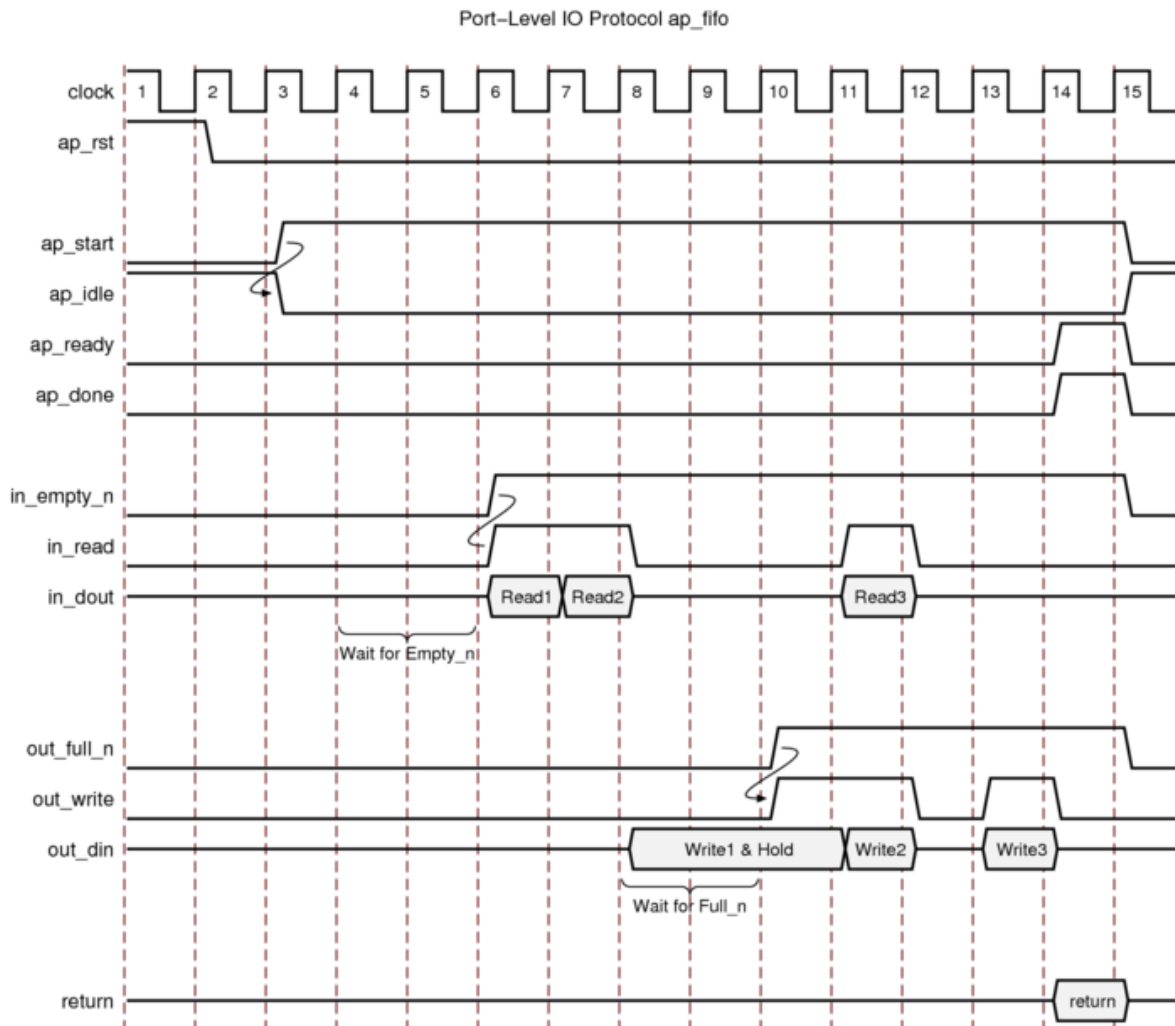
次の例では、`in1` は現在のアドレスにアクセスし、その後現在のアドレスの上 2 つのアドレスにアクセスして、最後に 1 つ下のアドレスにアクセスするポインターです。

```
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、アクセスが順次でない場合はエラー メッセージが表示され、停止します。順次アドレスではないアドレス位置から読み出すには、`ap_memory` または `bram` インターフェイスを使用します。

読み出しおよび書き込みの両方に使用される場合、引数に `ap_fifo` インターフェイスは指定できません。`ap_fifo` インターフェイスは、入力引数または出力引数にのみ指定可能です。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デザインは、次の図のように動作します。

図 103: ap_fifo インターフェ이스の動作



入力では、次のようになります。

- ap_start が High になると、ブロックが通常の動作を開始します。
- 入力ポートで読み出し準備ができたのに 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 入力ポートが High になって FIFO にデータが含まれていることが示されると、出力 ACK の in_read が High にアサートされてデータがこのサイクルで読み出されたことが示されます。

出力では、次のようになります。

- ap_start が High になると、ブロックが通常の動作を開始します。
- 出力ポートで書き込み準備ができて、FIFO がフルの場合 (out_full_n 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)、出力 ACK 信号の out_write が High になり、出力データが有効であることが示されます。

- `-rewind` オプションで最上位関数または最上位ループがパイプライン処理されると、Vivado HLS では `_lwr` が接尾語に付いた追加の出力ポートが作成されます。FIFO インターフェイスへの最後の書き込みが終了すると、`_lwr` ポートがアクティブ High になります。

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。`ap_bus` インターフェイスは特定のバス規格に従っていないので、システムバスと通信するバスブリッジと共に使用できます。バスブリッジでは、すべてのバースト書き込みをキャッシュできる必要があります。

注記: `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。[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、このコーディングスタイルを使用する場合の `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 `memcpy` が使用される場合に、データ転送を実行します。バーストモードでは、インターフェイスでベースアドレスと転送サイズが示されます。この後、データサンプルが連続するサイクルで転送されます。

注記: `memcpy` 関数でアクセスされる配列は、レジスタには分割できません。

図 4-11 および図 4-12 は、この例に示すように、`ap_bus` インターフェイスが引数 `d` に適用された標準モードでの読み出しと書き込み動作を示します。

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

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

次の例は、C 関数 `memcpy` とバーストモードを使用したところを示しています。

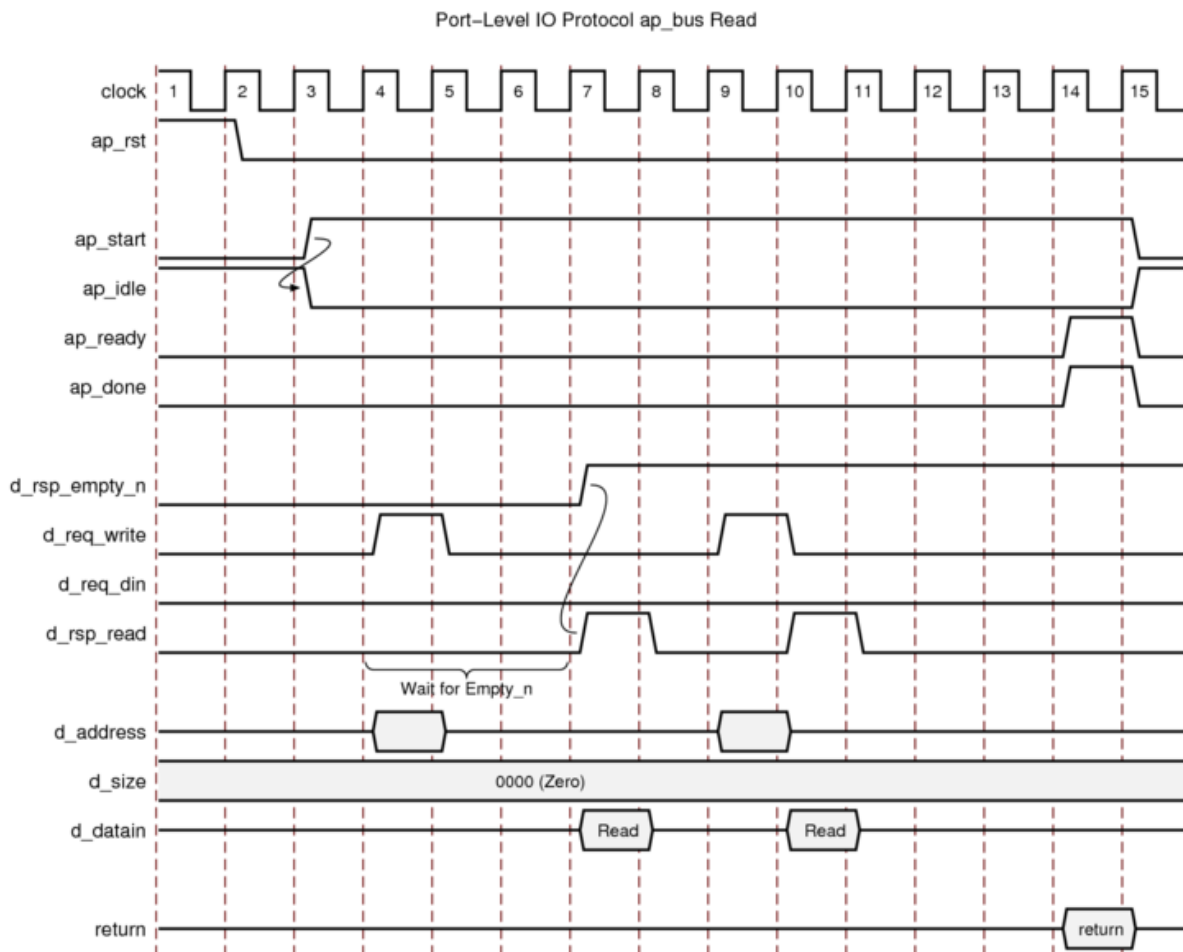
```
void bus (int *d) {
    int buf1[4], buf2[4];
    int i;

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

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

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

図 104: ap_bus インターフェイスの動作: 標準読み出し

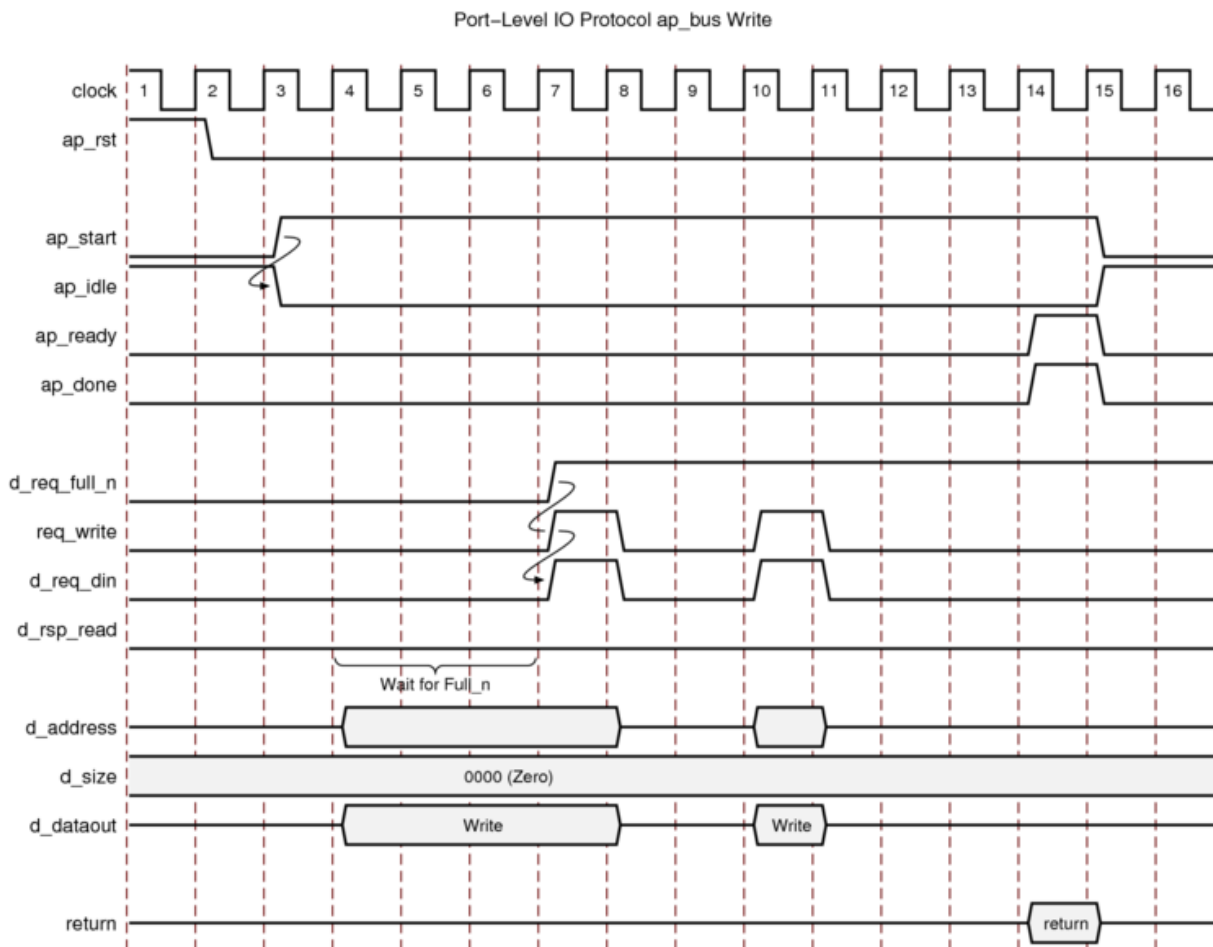


リセット後は、次ようになります。

- 開始後、ブロックが通常の動作を開始します。
- 読み出しを実行する必要があるのに、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、読み出しが実行されます。
 - address は出力です。
 - 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即座に出力信号 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必要があって、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、読み出しが実行されます。
 - address は出力です。

- 。 次のクロック サイクルで出力信号 `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。

図 105: `ap_bus` インターフェイスの動作: 標準書き込み

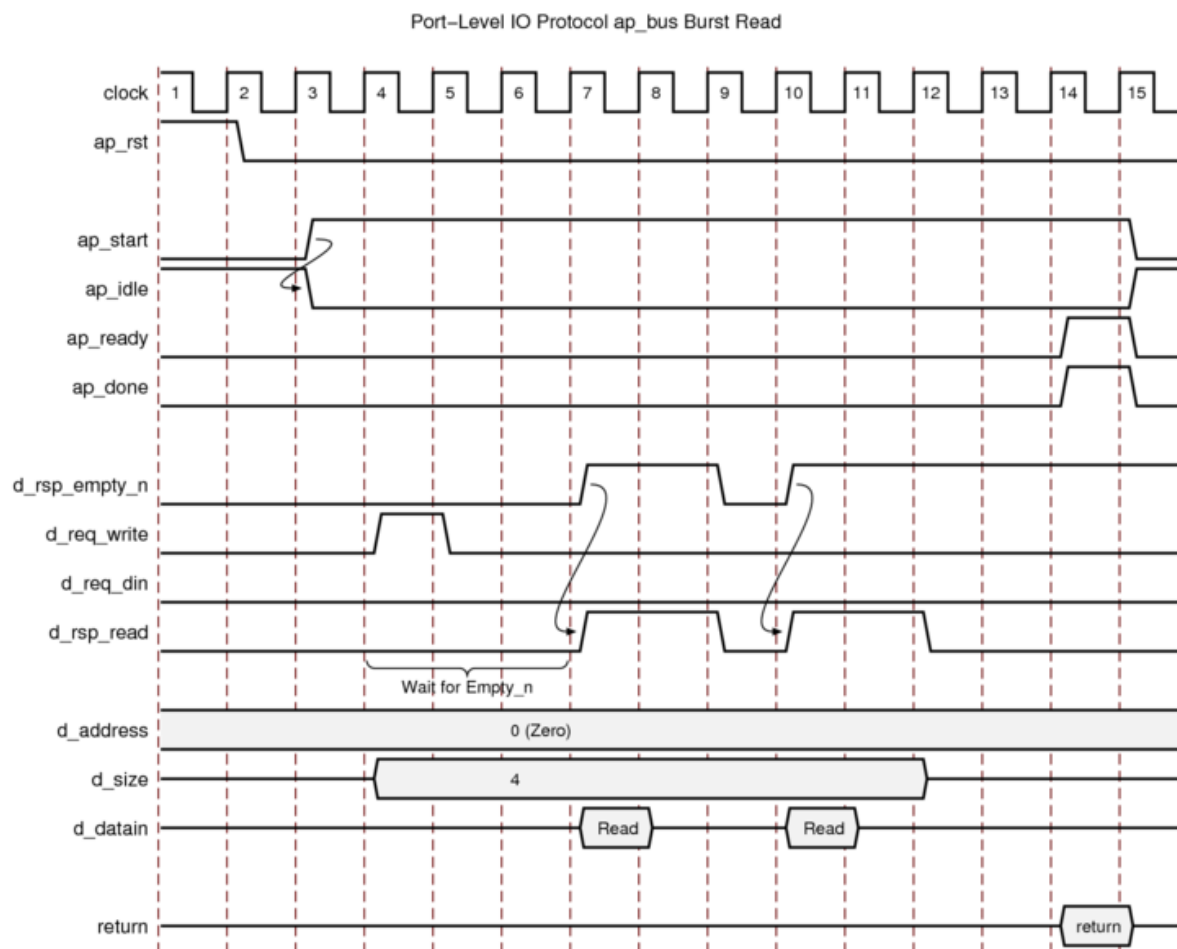


リセット後は、次のようになります。

- ・ 開始後、ブロックが通常の動作を開始します。
- ・ 書き込みを実行する必要があるのに、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、書き込み操作が実行されます。
 - 。 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必要があって、バス ブリッジ FIFO にデータがある場合は (`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、書き込み操作が実行されます。
- 。 アドレスおよびデータが出力です。
- 。 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。

図 106: ap_bus インターフェイスの動作: バースト読み出し

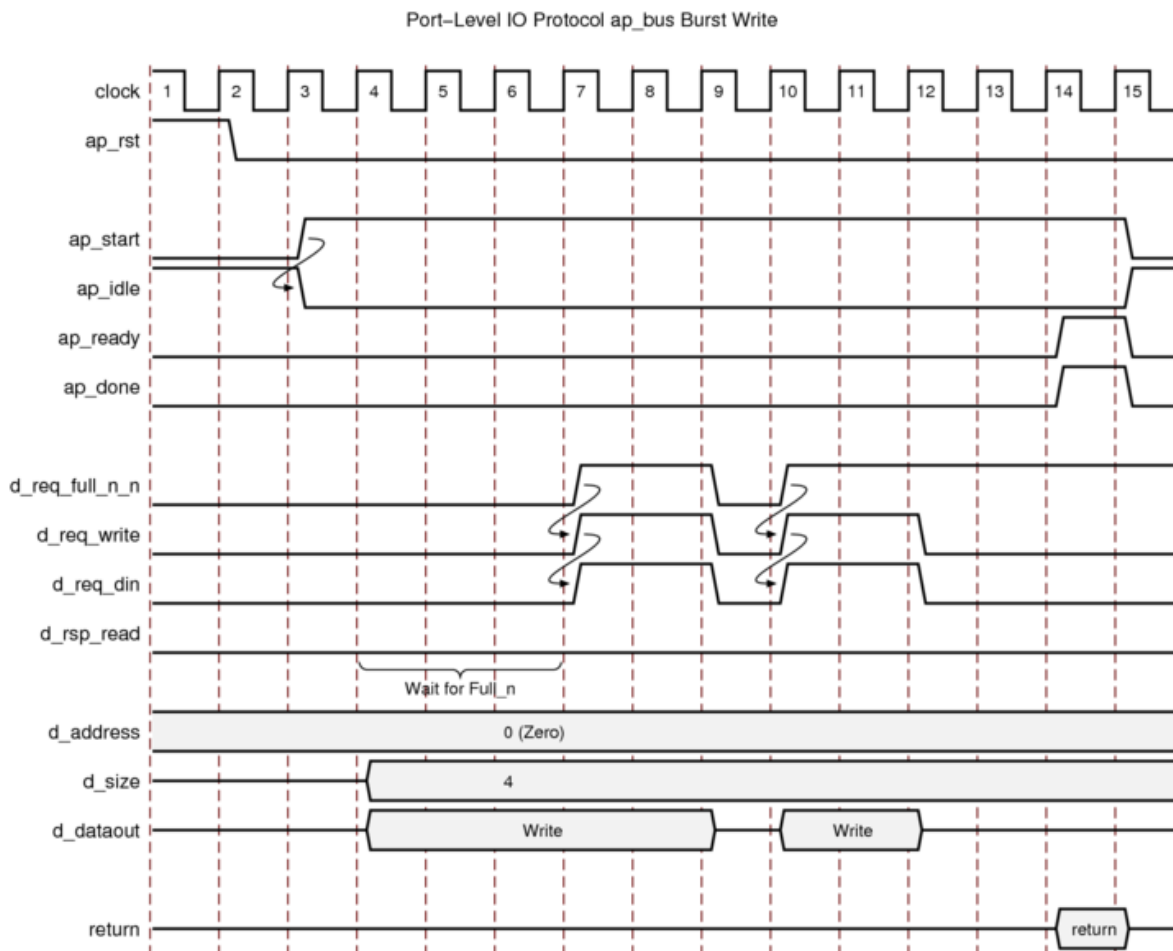


リセット後は、次のようになります。

- 。 開始後、ブロックが通常の動作を開始します。
- 。 読み出しを実行する必要があるのに、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、読み出しが実行されます。
 - 。 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` がすぐにアサートされ、次の `N` クロック エッジ (`N` は出力ポートの値) 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、データ転送が即座に停止され、データが読み出されるようになるまで待機します。

図 107: ap_bus インターフェイスの動作: バースト書き込み



リセット後は、次ようになります。

- 開始後、ブロックが通常の動作を開始します。
- 書き込みを実行する必要があるのに、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、書き込み操作が実行されます。
 - 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 はサイズ出力ポート d_size の値)。
- 書き込みが実行される必要があるときにバスブリッジ FIFO にスペースがあれば (d_rsp_full_n が High)、転送が開始され、デザインは停止して、FIFO がフルになるのを待ちます。

axis

axis モードでは、AXI4-Stream I/O プロトコルが指定されます。タイミングおよびポートを含む AXI4-Stream インターフェイスの詳細は、『Vivado Design Suite AXI リファレンス ガイド』(UG1037)を参照してください。この I/O プロトコルの機能すべてについては、[AXI4 インターフェイスの使用](#)を参照してください。

s_axilite

s_axilite モードでは、AXI4-Lite スレーブ I/O プロトコルが指定されます。タイミングおよびポートを含む AXI4-Lite スレーブ インターフェイスの詳細は、『Vivado Design Suite AXI リファレンス ガイド』(UG1037)を参照してください。この I/O プロトコルの機能すべてについては、[AXI4 インターフェイスの使用](#)を参照してください。

m_axi

m_axi モードでは、AXI4 マスター I/O プロトコルが指定されます。タイミングおよびポートを含む AXI4 マスター インターフェイスの詳細は、『Vivado Design Suite AXI リファレンス ガイド』(UG1037)を参照してください。この I/O プロトコルの機能すべてについては、[AXI4 インターフェイスの使用](#)を参照してください。

AXI4-Lite スレーブの C ドライバーのリファレンス

AXI4-Lite スレーブ インターフェイスがデザインに追加されると、C ドライバー ファイルのセットが自動的に作成されます。これらの C ドライバー ファイルは、CPU で実行されるどのソフトウェアにも統合できる API を含んでおり、AXI4-Lite インターフェイスを介してデバイスとの通信に使用されます。

API 関数には、最上位関数からの名前が一部使用されます。このセクションでは、最上位関数は DUT と想定します。次の表は、C ドライバー ファイルで提供される API 関数をそれぞれリストしています。

表 47: C ドライバーの API 関数

API 関数	説明
XDut_Initialize	この API は InstancePtr に値を書き込みます。その後ほかの API で使用できます。サイリンクスでは、MMU がシステムで使用される場合を除き、この API を呼び出してデバイスを初期化することを勧めしています。
XDut_CfgInitialize	デバイス コンフィギュレーションを初期化します。システムで MMU が使用される場合は、この関数を呼び出す前に XDut_Config 変数のベース アドレスを仮想ベース アドレスに置換します。Linux システムでは使用できません。
XDut_LookupConfig	ID を指定してデバイスのコンフィギュレーション情報を取得します。コンフィギュレーション情報には、物理ベース アドレスが含まれます。Linux システムでは使用できません。
XDut_Release	Linux で UIO デバイスを解放します。munmap によるマップを削除します。プロセスが停止されると、マップは自動的に削除されます。Linux システムでのみ使用できます。
XDut_Start	デバイスを起動します。この関数は、デバイスの ap_start ポートをアサートします。デバイスに ap_start ポートがある場合にのみ使用できます。
XDut_IsDone	デバイスが前の実行を終了したかどうかをチェックします。この関数は、デバイスの ap_done ポートの値を返します。デバイスに ap_done ポートがある場合にのみ使用できます。
XDut_IsIdle	デバイスがアイドル ステートかどうかをチェックします。この関数は、デバイスの ap_idle ポートの値を返します。デバイスに ap_idle ポートがある場合にのみ使用できます。

表 47: C ドライバーの API 関数 (続き)

API 関数	説明
XDut_IsReady	デバイスが次の入力を受信する準備ができているかどうかをチェックします。この関数は、デバイスの ap_ready ポートの値を返します。デバイスに ap_ready ポートがある場合にのみ使用できます。
XDut_Continue	ap_continue ポートをアサートします。デバイスに ap_continue ポートがある場合にのみ使用できます。
XDut_EnableAutoRestart	デバイスでの自動再開をイネーブルにします。これが設定されると、デバイスは現在のトランザクションが終了したら、次のトランザクションを自動的に開始します。
XDut_DisableAutoRestart	自動再開をディスエーブルにします。
XDut_Set_ARG	ARG ポートに値 (top 関数のスカラー引数) を書き込みます。ARG が入力ポートの場合にのみ使用できます。
XDut_Set_ARG_vld	ARG_vld ポートをアサートします。ARG が入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XDut_Set_ARG_ack	ARG_ack ポートをアサートします。ARG が出力ポートで、ap_hs または ap_ack インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XDut_Get_ARG	ARG から値を読み出します。ARG ポートがデバイスの出力ポートの場合にのみ使用できます。
XDut_Get_ARG_vld	ARG_vld から値を読み出します。ARG がデバイスの出力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XDut_Get_ARG_ack	ARG_ack から値を読み出します。ARG がデバイスの入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。
XDut_Get_ARG_BaseAddress	インターフェイス内の配列のベース アドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XDut_Get_ARG_HighAddress	配列の一番上位のアドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XDut_Get_ARG_TotalBytes	配列を格納するのに使用されるバイトの合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XDut_Get_ARG_BitWidth	配列内の各要素のビット幅を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XDut_Get_ARG_Depth	配列内の要素の合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。 配列の要素が 16 ビット未満の場合は、Vivado HLS により複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。
XDut_Write_ARG_Words	32 ビットワード長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XDut_Read_ARG_Words	配列から 32 ビットワードの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XDut_Write_ARG_Bytes	バイト長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。
XDut_Read_ARG_Bytes	配列からバイトの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと読み込まれるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

表 47: C ドライバーの API 関数 (続き)

API 関数	説明
<code>XDut_InterruptGlobalEnable</code>	割り込み出力をイネーブルにします。割り込み関数は、 <code>ap_start</code> がある場合にのみ使用できます。
<code>XDut_InterruptGlobalDisable</code>	割り込み出力をディスエーブルにします。
<code>XDut_InterruptEnable</code>	割り込みソースをイネーブルにします。最大で 2 つの割り込みソース (<code>ap_done</code> に <code>source 0</code> 、 <code>ap_ready</code> に <code>source 1</code>) がある可能性があります
<code>XDut_InterruptDisable</code>	割り込みソースをディスエーブルにします。
<code>XDut_InterruptClear</code>	割り込みステータスをクリアにします。
<code>XDut_InterruptGetEnabled</code>	割り込みソースがイネーブルかどうかをチェックします。
<code>XDut_InterruptGetStatus</code>	割り込みソースがトリガーされたかどうかをチェックします。

API 関数の詳細は、次に示します。

XDut_Initialize

構文

```
int XDut_Initialize(XDut *InstancePtr, u16 DeviceId);
```

```
int XDut_Initialize(XDut *InstancePtr, const char* InstanceName);
```

説明

`int XDut_Initialize(XDut *InstancePtr, u16 DeviceId)`: スタンドアロン システムで使用する場合に、デバイスを初期化します。この API は `InstancePtr` に適切な値を書き込みます。この後、ほかの API で使用できます。ザイリンクスでは、この API を呼び出してデバイスを初期化することをお勧めします。ただし、システムで MMU が使用されている場合は、`XDut_CfgInitialize` を使用してください。

`int XDut_Initialize(XDut *InstancePtr, const char* InstanceName)`: Linux システムで使用する場合は、特別に命名された `uio` デバイスを初期化します。最大で 5 メモリ マップを作成して、`sysfs` で `uio` デバイス情報を使用して `mmap` でスレーブ ベース アドレスを割り当てます。

- `InstancePtr`: デバイス インスタンスへのポインター。
- `DeviceId`: `xparameters.h` で定義されたデバイス ID。
- `InstanceName`: `uio` デバイスの名前。
- 戻り値: 問題がなかった場合は `XST_SUCCESS`、それ以外の場合はエラーあり。

XDut_CfgInitialize

コード例

```
XDut_CfgInitializeint XDut_CfgInitialize(XDut *InstancePtr, XDut_Config *ConfigPtr);
```

説明

システムに MMU が使用されている場合にデバイスを初期化します。この場合、AXI4-Lite スレーブの有効アドレスが `xparameters.h` で定義されたものと異なり、デバイスを初期化するのに API が必要となります。

- InstancePtr: デバイス インスタンスへのポインター。
- DeviceId: XDut_Config へのポインター。
- 戻り値: 問題がなかった場合は XST_SUCCESS、それ以外の場合はエラーあり。

XDut_LookupConfig

コード例

```
XDut_Config* XDut_LookupConfig(u16 DeviceId);
```

説明

この関数は、ID を指定してデバイスのコンフィギュレーション情報を取得します。

- DeviceId: xparameters.h で定義されたデバイス ID。
- 戻り値: デバイス ID が DeviceId のデバイスのコンフィギュレーション情報を保持する XDut_LookupConfig 変数へのポインター。一致する DeviceId が見つからなかった場合は NULL。

XDut_Release

コード例

```
int XDut_Release(XDut *InstancePtr);
```

説明

uio デバイスを解放します。munmap によるマップを削除します。プロセスが停止されると、マップは自動的に削除されます。

- InstanceName: uio デバイスの名前。
- 戻り値: 問題がなかった場合は XST_SUCCESS、それ以外の場合はエラーあり。

XDut_Start

コード例

```
void XDut_Start(XDut *InstancePtr);
```

説明

デバイスを起動します。この関数は、デバイスの ap_start ポートをアサートします。デバイスに ap_start ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_IsDone

構文

```
void XDut_IsDone(XDut *InstancePtr);
```

説明

デバイスが前の実行を終了したかどうかをチェックします。この関数は、デバイスの ap_done ポートの値を返します。デバイスに ap_done ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_IsIdle

コード例

```
void XDut_IsIdle(XDut *InstancePtr);
```

説明

デバイスがアイドル ステートかどうかをチェックします。この関数は、デバイスの ap_idle ポートの値を返します。デバイスに ap_idle ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_IsReady

構文

```
void XDut_IsReady(XDut *InstancePtr);
```

説明

デバイスが次の入力を受信する準備ができているかどうかをチェックします。この関数は、デバイスの ap_ready ポートの値を返します。デバイスに ap_ready ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_Continue

コード例

```
void XExample_Continue(XExample *InstancePtr);
```

説明

ap_continue ポートをアサートします。デバイスに ap_continue ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_EnableAutoRestart

コード例

```
void XDut_EnableAutoRestart(XDut *InstancePtr);
```

説明

デバイスでの自動再開をイネーブルにします。イネーブルの場合は、次が実行されます。

- デバイスにより ap_done がアサートされると、すぐに ap_start ポートがアサートされ、次のトランザクションが自動的に開始します。
- ブロック レベルの I/O プロトコル ap_ctrl_chain がデバイスにインプリメントされた場合、デバイスにより ap_ready がアサートされたとき、および ap_done がアサートされて ap_continue がアサートされたときに、次のトランザクションが自動的に再開します (ap_start がアサートされる)。

ap_start ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_DisableAutoRestart

コード例

```
void XDut_DisableAutoRestart(XDut *InstancePtr);
```

説明

自動再開をディスエーブルにします。ap_start ポートがある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_Set_ARG

コード例

```
void XDut_Set_ARG(XDut *InstancePtr, u32 Data);
```

説明

ARG ポートに値 (最上位関数のスカラー引数) を書き込みます。ARG が入力ポートの場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。
- Data: 書き込む値です。

XDut_Set_ARG_vld

コード例

```
void XDut_Set_ARG_vld(XDut *InstancePtr);
```

説明

ARG_vld ポートをアサートします。ARG が入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_Set_ARG_ack

コード例

```
void XDut_Set_ARG_ack(XDut *InstancePtr);
```

説明

ARG_ack ポートをアサートします。ARG が出力ポートで、ap_hs または ap_ack インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_Get_ARG

コード例

```
u32 XDut_Get_ARG(XDut *InstancePtr);
```

説明

ARG から値を読み出します。ARG ポートがデバイスの出力ポートの場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: ARG の値。

XDut_Get_ARG_vld

コード例

```
u32 XDut_Get_ARG_vld(XDut *InstancePtr);
```

説明

ARG_vld から値を読み出します。ARG がデバイスの出力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: ARG_vld の値。

XDut_Get_ARG_ack

コード例

```
u32 XDut_Get_ARG_ack(XDut *InstancePtr);
```

説明

ARG_ack から値を読み出します。ARG ポートがデバイスの入力ポートで、ap_hs または ap_vld インターフェイス プロトコルを使用してインプリメントされる場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: ARG_ack の値。

XDut_Get_ARG_BaseAddress

コード例

```
u32 XDut_Get_ARG_BaseAddress(XDut *InstancePtr);
```

説明

インターフェイス内の配列のベース アドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: 配列のベース アドレス。

XDut_Get_ARG_HighAddress

コード例

```
u32 XDut_Get_ARG_HighAddress(XDut *InstancePtr);
```

説明

配列の一番上位のアドレスを返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: 配列の最上位の要素のアドレス。

XDut_Get_ARG_TotalBytes

コード例

```
u32 XDut_Get_ARG_TotalBytes(XDut *InstancePtr);
```

説明

配列を格納するのに使用されるバイトの合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

配列の要素が 16 ビット未満の場合は、Vivado HLS により複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: 配列を格納するのに使用される総バイト数。

XDut_Get_ARG_BitWidth

コード例

```
u32 XDut_Get_ARG_BitWidth(XDut *InstancePtr);
```

説明

配列内の各要素のビット幅を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

配列の要素が 16 ビット未満の場合は、Vivado HLS により複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。

- InstancePtr: デバイス インスタンスへのポインター。

戻り値: 配列内の各要素のビット幅。

XDut_Get_ARG_Depth

コード例

```
u32 XDut_Get_ARG_Depth(XDut *InstancePtr);
```

説明

配列内の要素の合計数を返します。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

配列の要素が 16 ビット未満の場合は、Vivado HLS により複数の要素が 32 ビット データ幅の AXI4-Lite インターフェイスにまとめられます。要素のビット幅が 32 ビットを超える場合は、Vivado HLS により各要素が連続する複数のアドレスに格納されます。

- InstancePtr: デバイス インスタンスへのポインター。

配列内の要素の合計数を返します。

XDut_Write_ARG_Words

構文

```
u32 XDut_Write_ARG_Words(XDut *InstancePtr, int offset, int *data, int length);
```

説明

32 ビット ワード長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。
- offset: AXI4-Lite インターフェイスのアドレス。
- data: 格納するデータ値へのポインター。
- length: 格納するデータの長さ。

戻り値: 指定したアドレスからのデータの書き込み長。

XDut_Read_ARG_Words

コード例

```
u32 XDut_Read_ARG_Words(XDut *InstancePtr, int offset, int *data, int length);
```

説明

配列から 32 ビット ワードの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。
- offset: ARG のアドレス。
- data: デバイス バッファへのポインター。
- length: 格納するデータの長さ。

戻り値: 指定したアドレスからのデータの読み出し長。

XDut_Write_ARG_Bytes

コード例

```
u32 XDut_Write_ARG_Bytes(XDut *InstancePtr, int offset, char *data, int length);
```

説明

バイト長を指定した AXI4-Lite インターフェイスのアドレスに書き込みます。この API には、ベース アドレスからのオフセット アドレスと格納されるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。
- offset: ARG のアドレス。
- data: 格納するデータ値へのポインター。
- length: 格納するデータの長さ。

戻り値: 指定したアドレスからのデータの書き込み長。

XDut_Read_ARG_Bytes

コード例

```
u32 XDut_Read_ARG_Bytes(XDut *InstancePtr, int offset, char *data, int length);
```

説明

配列からバイトの長さが読み出されます。この API には、データ ターゲット、ベース アドレスからのオフセット アドレスと読み込まれるデータの長さが必要です。ARG が AXI4-Lite インターフェイスにまとめられる 配列の場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。
- offset: ARG のアドレス。
- data: デバイス バッファへのポインター。
- length: 読み出すデータの長さ。

戻り値: 指定したアドレスからのデータの読み出し長。

XDut_InterruptGlobalEnable

コード例

```
void XDut_InterruptGlobalEnable(XDut *InstancePtr);
```

説明

割り込み出力をイネーブルにします。割り込み関数は、ap_start がある場合にのみ使用できます。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_InterruptGlobalDisable

構文

```
void XDut_InterruptGlobalDisable(XDut *InstancePtr);
```

説明

割り込み出力をディスエーブルにします。

- InstancePtr: デバイス インスタンスへのポインター。

XDut_InterruptEnable

構文

```
void XDut_InterruptEnable(XDut *InstancePtr, u32 Mask);
```

説明

割り込みソースをイネーブルにします。最大で 2 つの割り込みソース (ap_done には source 0、ap_ready には source 1) がある可能性があります。

- InstancePtr: デバイス インスタンスへのポインター。
- Mask: ビット マスク。
 - ビット n = 1: 割り込みソース n をイネーブル。
 - ビット n = 0: 変更なし。

XDut_InterruptDisable

コード例

```
void XDut_InterruptDisable(XDut *InstancePtr, u32 Mask);
```

説明

割り込みソースをディスエーブルにします。

- InstancePtr: デバイス インスタンスへのポインター。
- Mask: ビット マスク。
 - ビット n = 1: 割り込みソース n をディスエーブル。
 - ビット n = 0: 変更なし。

XDut_InterruptClear

コード例

```
void XDut_InterruptClear(XDut *InstancePtr, u32 Mask);
```

説明

割り込みステータスをクリアにします。

- InstancePtr: デバイス インスタンスへのポインター。
- Mask: ビット マスク。
 - ビット n = 1: 割り込みステータス n をトグル。
 - ビット n = 0: 変更なし。

XDut_InterruptGetEnabled

コード例

```
u32 XDut_InterruptGetEnabled(XDut *InstancePtr);
```

説明

割り込みソースがイネーブルかどうかをチェックします。

- InstancePtr: デバイス インスタンスへのポインター。
- 戻り値: ビット マスク。
 - 。 ビット n = 1: イネーブル。
 - 。 ビット n = 0: ディスエーブル。

XDut_InterruptGetStatus

コード例

```
u32 XDut_InterruptGetStatus(XDut *InstancePtr);
```

説明

割り込みソースがトリガーされたかどうかをチェックします。

- InstancePtr: デバイス インスタンスへのポインター。
- 戻り値: ビット マスク。
 - 。 ビット n = 1: トリガーあり。
 - 。 ビット n = 0: トリガーなし。

HLS ビデオ関数ライブラリ



重要: Vivado® HLS ビデオ ライブラリは、ザイリンクス GitHub (<https://github.com/Xilinx/xfopencv>) に移動されました。

HLS 線形代数関数

このセクションでは、Vivado HLS の線形代数を処理する関数について説明します。

matrix_multiply

コード例

```
template<
class TransposeFormA,
class TransposeFormB,
int RowsA,
int ColsA,
int RowsB,
int ColsB,
int RowsC,
int ColsC,
typename InputType,
typename OutputType>
void matrix_multiply(
    const InputType A[RowsA][ColsA],
    const InputType B[RowsB][ColsB],
    OutputType C[RowsC][ColsC]);
```

説明

$C=AB$

- 2つの行列の積を計算し、3つ目の行列を返します。
- オプションで入力行列を転置 (複素データ型の場合は共役転置) します。
- 展開された浮動小数点のインプリメンテーション用に代替アーキテクチャが提供されています。

パラメーター

表 48: パラメーター

パラメーター	説明
TransposeFormA	行列 A の転置要件 (NoTranspose、Transpose、ConjugateTranspose)。
TransposeFormB	行列 B の転置要件 (NoTranspose、Transpose、ConjugateTranspose)。
RowsA	行列 A の行数
ColsA	行列 A の列数
RowsB	行列 B の行数
ColsB	行列 B の列数
RowsC	行列 C の行数
ColsC	行列 C の列数
InputType	入力データ型
OutputType	出力データ型

$\text{ColsA} \neq \text{RowsB}$ の場合、関数はアサートを実行し、コンパイル エラーまたは合成エラーになります。A と B の転置要件は、チェックが作成される前に満たされます。

引数

表 49: 引数

引数	説明
A	1 つ目の入力行列
B	2 つ目の入力行列
C	AB (出力行列の積)

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- ap_fixed
- float
- x_complex<ap_fixed>
- x_complex<float>

入力データの想定

- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアーになります。

cholesky

コード例

```
template<
bool LowerTriangularL,
int RowsColsA,
typename InputType,
typename OutputType>
int cholesky(
const InputType A[RowsColsA][RowsColsA],
OutputType L[RowsColsA][RowsColsA])
```

説明

$A=LL^*$

- 入力行列 A のコレスキー分解を計算し、行列 L を返します。
- 出力行列 L は LowerTriangularL パラメーターに基づいて上三角行列または下三角行列になります。
- 行列 L の未使用部分の要素は 0 に設定されます。

パラメーター

表 50: パラメーター

パラメーター	説明
RowsColsA	入力行列および出力行列の行と列の次元
LowerTriangularL	下三角出力または上三角出力のどちらが必要かを選択します。
InputType	入力データ型
OutputType	出力データ型

引数

表 51: 引数

引数	説明
A	エルミート/対称正定値入力行列
L	下三角または上三角出力行列

戻り値

- 正しく処理された場合は 0
- エラーになった場合は 1。関数では負の数の平方根を見つけようとします。たとえば、入力行列 A がエルミート/対称正定値行列ではない場合などにエラーになります。

サポートされるデータ型

- ap_fixed
- float
- x_complex<ap_fixed>
- x_complex<float>

入力データの想定

- 関数では入力行列が対称正定値行列 (複素数入力の場合はエルミート正定値行列) であると想定されます。
- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアーになります。

qrf

コード例

```
template<
bool TransposeQ,
int RowsA,
int ColsA,
typename InputType,
```

```
typename OutputType>
void qrf(
    const InputType A[RowsA][ColsA],
    OutputType Q[RowsA][ColsA],
    OutputType R[RowsA][ColsA])
```

説明

$A=QR$

- 入力行列 A の完全な QR 因数分解 (QR 分解) が計算され、直行出力行列 Q と上三角行列 R が出力されます。
- 出力行列 Q は TransposeQ パラメーターに基づいてオプションで転置されることがあります。
- 出力行列 R の下三角要素は 0 にはなりません。
- 薄い (エコノミーな) QR 分解はインプリメントされません。

パラメーター

表 52: パラメーター

パラメーター	説明
TransposeQ	Q 行列を転置するべきかどうかを選択。
RowsA	入力行列 A の行数
ColsA	入力行列 A の列数
InputType	入力データ型
OutputType	出力データ型

- 関数は $RowsA < ColsA$ の場合、コンパイル エラーになるか合成エラーになります。

引数

表 53: 引数

引数	説明
A	入力マトリックス
Q	直行出力行列
R	上三角出力行列

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- float
- x_complex<float>

入力データの想定

- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアになります。

cholesky_inverse

コード例

```
template <
    int RowsColsA,
    typename InputType,
    typename OutputType>
void cholesky_inverse(const InputType A[RowsColsA][RowsColsA],
                     OutputType InverseA[RowsColsA][RowsColsA],
                     int& cholesky_success)
```

説明

$$AA^{-1} = I$$

- 正定値対称入力行列の逆数をコレスキー分解方法で計算し、InverseA 行列を出力します。

パラメーター

表 54: パラメーター

パラメーター	説明
RowsColsA	入力行列および出力行列の行と列の次元
InputType	入力データ型
OutputType	出力データ型

引数

表 55: 引数

引数	説明
A	平方根エルミート/対称正定値入力行列
InverseA	入力行列の逆数
cholesky_success	正しく処理された場合は 0 エラーになった場合は 1。コレスキー関数は、負の数の平方根を見つけようとします。入力行列 A は対称正定値行列ではありません。

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- ap_fixed

- float
- x_complex<ap_fixed>
- x_complex<float>

入力データの想定

- 関数では入力行列が対称正定値行列 (複素数入力の場合はエルミート正定値行列) であると想定されます。
- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアになります。

qr_inverse

コード例

```
template <
    int RowsColsA,
    typename InputType,
    typename OutputType>
void qr_inverse(const InputType A[RowsColsA][RowsColsA],
                OutputType InverseA[RowsColsA][RowsColsA],
                int& A_singular)
```

説明

 $AA^{-1}=I$

- 入力行列 A の逆数を QR 因数分解方法で計算し、InverseA 行列を出力します。

パラメーター

表 56: パラメーター

パラメーター	説明
RowsColsA	入力行列および出力行列の行と列の次元。
InputType	入力データ型
OutputType	出力データ型

引数

表 57: 引数

引数	説明
A	入力行列 A
InverseA	入力行列の逆数
A_singular	正しく処理された場合は 0 行列 A が特異行列の場合は 1

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- float
- x_complex<float>

入力データの想定

- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアになります。

svd

コード例

```
template<
int RowsA,
int ColsA,
typename InputType,
typename OutputType>
void svd(
const InputType A[RowsA][ColsA],
OutputType S[RowsA][ColsA],
OutputType U[RowsA][RowsA],
OutputType V[ColsA][ColsA])
```

説明

$A=USV^*$

- 入力行列 A の特異値分解を計算して、行列 U、S、および V を出力します。
- 平方根行列のみがサポートされます。
- 反復両面ヤコビ方法を使用してインプリメントされます。

パラメーター

表 58: パラメーター

パラメーター	説明
RowsA	行の次元
ColsA	列の次元
InputType	入力データ型
OutputType	出力データ型

- RowsA != ColsA の場合、関数はアサートを実行し、コンパイル エラーまたは合成エラーになります。

引数

表 59: 引数

引数	説明
A	入力マトリックス
S	入力行列の特異値
U	入力行列の左特異ベクター
V	入力行列の右特異ベクター

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- float
- x_complex<float>

入力データの想定

- 浮動小数点型の場合、非正規の入力値はサポートされません。使用されると、合成されたハードウェアでこれらが 0 にフラッシュされ、ソフトウェア シミュレーションと異なるビヘイビアになります。

例

この例では、基本的なテストベンチを提供し、各線形代数関数のパラメーター指定方法とインスタンスエーション方法を示します。各関数の例 (複数の例があることもあり) は、Vivado HLS の examples ディレクトリから入手できます。

```
<VIVADO_HLS>/examples/design/linear_algebra
```

各サンプルには、次のファイルが含まれます。

- <example>.cpp: ライブラリ関数をインスタンスエートする最上位合成ラッパー。
- <example>.h: 行列サイズ、データ型、可能な場合はアーキテクチャ選択などを定義するヘッダー ファイル。
- <example>_tb.cpp: 最上位合成ラッパーをインスタンスエートする基本的なテストベンチ。
- run_hls.tcl: Vivado HLS のサンプル プロジェクトを設定する Tcl コマンド。

```
vivado_hls -f run_hls.tcl
```

- directives.tcl: (オプション) 最適化/インプリメンテーション指示子を適用する追加の Tcl コマンド。

HLS DSP ライブラリ関数

HLS DSP ライブラリには、C++ で DSP システムをモデルするブロック構築関数、特に SDR アプリケーションで使われる関数が含まれています。

HLS DSP 関数

このセクションでは、Vivado HLS の DSP 関数について説明します。

awgn

コード例

```
template<
    int OutputWidth>
class awgn {
public:
    typedef ap_ufixed<8,4, AP_RND, AP_SAT> t_ input_scale;
    static const int LFSR_SECTION_WIDTH = 32;
    static const int NUM_NOISE_GENS = 4;
    static const int LFSR_WIDTH = LFSR_SECTION_WIDTH*NUM_NOISE_GENS;
    void awgn(ap_uint<LFSR_WIDTH> seed);
    void ~awgn();
    void operator()(t_ input_scale &snr,
        ap_int<OutputWidth> &noise);
```

説明

- 入力信号対ノイズ比 (SNR) で決められたガウス ノイズを出力します。BPSK 信号の場合、0 dB は約 7% のビット エラー レート (BER) になります。これは、 $E_b/N_0 = 0$ のとき $E_b = 1$ ですが、BPSK チャネルのノイズ電力は $N_0/2$ など、ノイズ変動は信号変動の半分になるからです。詳細は、MathWorks 社のウェブサイトの AWGN ページ (<https://www.mathworks.com/help/comm/ug/awgn-channel.html>) を参照してください。
- SNR 入力は信号対ノイズ比をデシベルで表し、その範囲は 1 デシベルの 1/16 ごとに、0.0 から 16.0 までとなります。
- ノイズ値がコンフィギュレーションで設定可能な値を超える場合、正の値または負の値の最大数で飽和します。
- 関数は、合算されるノイズ ジェネレーターを複数使用し、出力値を生成するのに、中心極限定理を利用します。デフォルトでは、これら複数のジェネレーターはパイプライン処理され、展開されます。これは、ターゲット アプリケーションが高クロック レートで開始間隔が 1 となる高レートの BER テスト用だからです。

パラメーター

表 60: パラメーター

テンプレート パラメーター	説明
OutputWidth	出力値のビット数。SNR は、ソフト BPSK 信号対ノイズ比を指定します。値は 01000 および 11000 です。範囲は 8 から 32 ビットです。

表 61: コンストラクター引数

引数	説明
seed	ノイズ ジェネレーター内の LFSR のシード値。

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 62: 引数

引数	説明
snr	信号対ノイズ比入力
noise	出力ノイズ

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- 入力
 - ap_ufixed
詳細は、ヘッダー ファイル hls_awgn.h の typedef t_input_scale の定義を参照してください。
- 出力
 - ap_int

入力データの想定

- なし

nco

コード例

```
template<
    int AccumWidth,
    int PhaseAngleWidth,
    int SuperSampleRate,
    int OutputWidth,
    class DualOutputCmpyImpl,
    class SingleOutputCmpyImpl,
    class SingleOutputNegCmpyImpl>
class nco {
public:
    void nco(const ap_uint<AccumWidth> InitPinc,
            const ap_uint<AccumWidth> InitPoff);
    void ~nco();
    void operator()(
        stream< ap_uint<AccumWidth> > &pinc,
        stream< ap_uint<AccumWidth> > &poff,
        stream< t_nco_output_data<SuperSampleRate,OutputWidth> >
        &outputData
    );
};
```

説明

- 数値制御型オシレーター (NCO) 機能を実行します。

- サンプル レートがクロック レートを超える、スーパー サンプル レート (SSR) をサポートするため、複数のパラレル データ サンプルを各クロック サイクルで出力する必要があります。
- SSR モードの場合、位相インクリメント (pinc) を変更すると、内部割込みが発生します。N サイクル間 (N は $\text{uperSampleRate}/2 + 1$) で pinc に 3 つ以上の変更を加えない限り、出力サンプルには影響しません。

パラメーター

表 63: パラメーター

テンプレート パラメーター	説明
AccumWidth	位相アキュムレータのビット数。合成可能な周波数の精度を決定します。範囲は 4 から 48 です。
PhaseAngleWidth	sin/cos ルックアップで直接使用されるビット数。大きい値を指定すると出力はより正確になりますが、ルックアップ テーブルのサイズが大きくなります。範囲は 4 から 16 です。
SuperSampleRate	クロック サイクルごとの出力サンプル数。範囲は 1 から 16 です。
OutputWidth	各出力 (サインおよびコサイン) の幅。範囲は 4 から 32 です。
DualOutputCmpyImpl	NcoDualOutputCmpyFiveMult または NcoDualOutputCmpyFourMult. のクラスを使用して、5 乗算器 (5 DSP48) または 4 乗算器 (6 DSP48) で、デュアル出力の複素乗算器をインプリメントします。詳細は、hls_nco.h を参照してください。
SingleOutputCmpyImpl	NcoSingleOutputCmpyThreeMult または NcoSingleOutputCmpyFourMult. のクラスを使用して、3 乗算器 (3 DSP48) または 4 乗算器 (4 DSP48) で、シングル出力の複素乗算器をインプリメントします。詳細は、hls_nco.h を参照してください。
SingleOutputNegCmpyImpl	NcoSingleOutputCmpyThreeMult または NcoSingleOutputCmpyFourMult. のクラスを使用して、3 乗算器 (3 DSP48) または 4 乗算器 (4 DSP48) で、シングル出力の二ゲート複素乗算器をインプリメントします。詳細は、hls_nco.h を参照してください。

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 64: 引数

引数	説明
pinc	pinc は位相インクリメントです。サンプルごとに $\text{pinc}/2^{\text{AccumWidth}} * 2\pi$ ずつ、出力の位相は遷移します。
poff	poff は位相オフセットです。これは累計位相に追加されます。 $\text{poff}/2^{\text{AccumWidth}} * 2\pi$ 分、出力の位相はオフセットします。
outputData	サインおよびコサインの出力。出力コンポーネントのマグニチュードは、およそ、 $\cos(\phi) * 2^{\text{OutputWidth}-1}$ および $\sin(\phi) * 2^{\text{OutputWidth}-1}$ になります。 ϕ は位相アキュムレータで記述された位相で、poff により適宜オフセットされます。

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- 入力
 - ap_uint
- 出力
 - std::complex< ap_int >

詳細は、hls_nco.h の struct t_nco_output_data の定義を参照してください。

入力データの想定

- なし

convolution_encoder

コード例

```
template<
  int OutputWidth,
  bool Punctured,
  bool DualOutput,
  int InputRate,
  int OutputRate,
  int ConstraintLength,
  int PunctureCode0,
  int PunctureCode1,
  int ConvolutionCode0,
  int ConvolutionCode1,
  int ConvolutionCode2,
  int ConvolutionCode3,
  int ConvolutionCode4,
  int ConvolutionCode5,
  int ConvolutionCode6>
class convolution_encoder {
public:
  convolution_encoder();
  ~convolution_encoder();
  void operator()(stream< ap_uint<1> > &inputData,
    stream< ap_uint<OutputWidth> > &outputData);
```

説明

- ユーザー定義のたたみ込みコードおよび制約の長さに基づいて、入力データ ストリームのたたみ込みエンコードを実行します。
- データのパンクチャ (オプション)
- デュアル チャネル出力 (オプション)

パラメーター

表 65: パラメーター

テンプレート パラメーター	説明
OutputWidth	出力バスのビット数を定義します。Punctured=true および DualOutput=false の場合は 1 ビット、DualOutput=true、else OutputRate ビットの場合は 2 ビットです。
Punctured	これが true の場合は、データのパンクチャがイネーブルになります。
DualOutput	これが true の場合は、パンクチャされたデータでデュアル出力がイネーブルになります。
InputRate	コード レートの分子を定義します。
OutputRate	コード レートの分母を定義します。
ConstraintLength	制約長 K はエンコーダーのレジスタ数に 1 を足したものです。
PunctureCode0	Punctured=true のときの出力 0 のパンクチャ コード。長さ (2 進数) はパンクチャ入力レートと等しくする必要があります。両方の PunctureCode パラメーターの 1 の合計数は出力レートに等しくなります。
PunctureCode1	Punctured=true のときの出力 1 のパンクチャ コード。長さ (2 進数) はパンクチャ入力レートと等しくする必要があります。両方の PunctureCode パラメーターの 1 の合計数は出力レートに等しくなります。
ConvolutionCode0	レートが 1/2 から 1/7 のときのたたみ込みコード。すべてのたたみ込みコードの長さ (2 進数) は制約の長さの値と等しくする必要があります。
ConvolutionCode1	レートが 1/2 から 1/7 のときのたたみ込みコード。
ConvolutionCode2	レートが 1/3 から 1/7 のときのたたみ込みコード。
ConvolutionCode3	レートが 1/4 から 1/7 のときのたたみ込みコード。
ConvolutionCode4	レートが 1/5 から 1/7 のときのたたみ込みコード。
ConvolutionCode5	レートが 1/6 から 1/7 のときのたたみ込みコード。
ConvolutionCode6	レートが 1/7 のときのたたみ込みコード。

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 66: 引数

引数	説明
inputData	エンコードされるシングル ビット データ ストリーム。
outputData	エンコードされたデータ ストリーム。Punctured=true (1 ビット幅) または DualOutput=true (2 ビット幅) でない場合は OutputRate ビット幅です。

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- ap_uint

入力データの想定

- なし

viterbi_decoder

コード例

```
template<
    int ConstraintLength,
    int TracebackLength,
    bool HasEraseInput,
    bool SoftData,
    int InputDataWidth,
    int SoftDataFormat,
    int OutputRate,
    int ConvolutionCode0,
    int ConvolutionCode1,
    int ConvolutionCode2,
    int ConvolutionCode3,
    int ConvolutionCode4,
    int ConvolutionCode5,
    int ConvolutionCode6>
class viterbi_decoder {
public:
    viterbi_decoder();
    ~viterbi_decoder();
    void operator()(stream<
viterbi_decoder_input<OutputRate, InputDataWidth, HasEraseInput> > &inputData,
    stream< ap_uint<1> > &outputData)
```

説明

- たたみ込みによりエンコードされたデータ ストリームのビタビ デコードを実行します。
- ハードまたはソフト データをサポートします。
- オフセット バイナリおよび符号付きマグニチュード ソフト データ フォーマットをサポートします。
- 抹消 (パンクチャリング) をサポートします。

パラメーター

表 67: パラメーター

テンプレート パラメーター	説明
ConstraintLength	制約の長さ K です。サポートされている範囲は 3 から 9 までです。
TracebackLength	デコード中にトレリスを介して追跡するステート数。最低 6x の ConstraintLength を使用、またはパンクチャ コードの場合は最低 12x の ConstraintLength を使用します。
HasEraseInput	true のとき、パンクチャ コードで抹消 (ヌル シンボル) をフラグするため、抹消入力のコアにあります。

表 67: パラメーター (続き)

テンプレート パラメーター	説明
SoftData	true のとき、関数はソフト (マルチ ビット) 入力データを受け入れます。
InputDataWidth	入力データの幅を指定します。ハード データの場合は 1、ソフト データの場合は 3 から 5 に設定されます。
SoftDataFormat	ソフト データ フォーマットを指定します。0 の場合は符号付マグニチュード、1 の場合はオフセット バイナリです。
OutputRate	一致しているたたみ込みエンコーダーの出力レートを指定します。デコーダーの入力バス数を決定します。
ConvolutionCode0	レートが 1/2 から 1/7 のときのたたみ込みコード。 すべてのたたみ込みコードの長さ (2 進数) は制約の長さの値と等しくする必要があります。
ConvolutionCode1	レートが 1/2 から 1/7 のときのたたみ込みコード。
ConvolutionCode2	レートが 1/3 から 1/7 のときのたたみ込みコード。
ConvolutionCode3	レートが 1/4 から 1/7 のときのたたみ込みコード。
ConvolutionCode4	レートが 1/5 から 1/7 のときのたたみ込みコード。
ConvolutionCode5	レートが 1/6 から 1/7 のときのたたみ込みコード。
ConvolutionCode6	レートが 1/7 のときのたたみ込みコード。

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 68: 引数

引数	説明
inputData	バンクチュア コードが使用されている場合は、付随する抹消信号でたたみ込みエンコードされたデータ ストリーム。データバスは $\text{OutputRate} \times \text{InputDataWidth}$ ビット幅になります。抹消バスは OutputRate ビット幅になります。
outputData	デコードされた単一ビットのデータ ストリーム。

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- 入力
 - ap_uint

詳細は、hls_viterbi_decoder.h の struct viterbi_decoder_input の定義を参照してください。
- 出力
 - ap_uint

入力データの想定

- なし

atan2

コード例

```
template <
    int PhaseFormat,
    int InputWidth,
    int OutputWidth,
    int RoundMode>
void atan2(const typename atan2_input<InputWidth>::cartesian &x,
           typename atan2_output<OutputWidth>::phase &atanX)
```

説明

- 引数 2 つの逆正接の CORDIC ベースの固定小数点インプリメンテーション
- コンフィギャラブルな入力および出力の幅
- コンフィギャラブルな位相フォーマット
- コンフィギャラブルな丸めモード

パラメーター

表 69: パラメーター

テンプレート パラメーター	説明
PhaseFormat	位相をラジアンで表すか、スケールされたラジアン ($\pi * 1$ ラジアン) で表すかを選択します。
InputWidth	全体的な入力幅を定義します。
OutputWidth	全体的な出力幅を定義します。
RoundMode	出力データに適用する丸めモードを選択します。 0 = 切り捨て 1 = 正の無限大への丸め 2 = 正および負の無限大への丸め 3 = 最近接偶数への丸め

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 70: 引数

引数	説明
x	2 つの整数ビットおよび $[-1,1]$ の範囲で InputWidth-2 小数ビットのある入力データ。
atanX	3 つの整数ビットおよび $[-1,1]$ の範囲で OutputWidth-3 小数ビットのある x の 4 象限逆正接。

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- 入力
 - `std::complex<ap_fixed>`
 詳細は、`hls_cordic_functions.h` の `struct cordic_inputs` および `hls_atan2_cordic.h` の `struct atan2_input` の定義を参照してください。
- 出力
 - `ap_fixed`
 詳細は、`hls_cordic_functions.h` の `struct cordic_outputs` および `hls_atan2_cordic.h` の `struct atan2_output` の定義を参照してください。

入力データの想定

- なし

sqrt

コード例

```
template <
    int DataFormat,
    int InputWidth,
    int OutputWidth,
    int RoundMode>
void sqrt(const typename sqrt_input<InputWidth, DataFormat>::in &x,
          typename sqrt_output<OutputWidth, DataFormat>::out &sqrtX)
```

説明

- 平方根の CORDIC ベースの固定小数点インプリメンテーション
- 符号なしの小数または符号なしの整数のデータ フォーマットをサポート
- コンフィギャラブルな丸めモード

パラメーター

表 71: パラメーター

テンプレート パラメーター	説明
DataFormat	符号なしの小数 (1 ビットの整数幅) または符号なしの整数のフォーマットを選択します。
InputWidth	全体的な入力幅を定義します。
OutputWidth	全体的な出力幅を定義します。

表 71: パラメーター (続き)

テンプレート パラメーター	説明
RoundMode	出力データに適用する丸めモードを選択します。 0 = 切り捨て 1 = 正の無限大への丸め 2 = 正および負の無限大への丸め、3 = 最近接偶数への丸め

注記: テンプレート パラメーター コンフィギュレーションが有効であることを検証するため、C シミュレーション中にパラメーターがチェックされます。

引数

表 72: 引数

引数	説明
x	入力データ。
sqrtX	入力データの平方根。

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- 入力
 - ap_ufixed
 - ap_uint
詳細は、hls_cordic_functions.h の struct cordic_inputs および hls_sqrt_cordic.h の struct sqrt_input の定義を参照してください。
- 出力
 - ap_ufixed
 - ap_uint
詳細は、hls_cordic_functions.h の struct cordic_inputs および hls_sqrt_cordic.h の struct sqrt_input の定義を参照してください。

入力データの想定

- なし

cmPy

コード例

- スカラー インターフェイス

```
template <
class Architecture,
int W1, int I1, ap_q_mode Q1, ap_o_mode O1, int N1,
int W2, int I2, ap_q_mode Q2, ap_o_mode O2, int N2>
void cmPy (const ap_fixed<W1, I1, Q1, O1, N1> &ar,
const ap_fixed<W1, I1, Q1, O1, N1> &ai,
const ap_fixed<W1, I1, Q1, O1, N1> &br,
const ap_fixed<W1, I1, Q1, O1, N1> &bi,
ap_fixed<W2, I2, Q2, O2, N2> &pr,
ap_fixed<W2, I2, Q2, O2, N2> &pi);
```

- std::complex インターフェイス

```
template <
class Architecture,
int W1, int I1, ap_q_mode Q1, ap_o_mode O1, int N1,
int W2, int I2, ap_q_mode Q2, ap_o_mode O2, int N2>
void cmPy (const std::complex< ap_fixed<W1, I1, Q1, O1, N1> > &a,
const std::complex< ap_fixed<W1, I1, Q1, O1, N1> > &b,
std::complex< ap_fixed<W2, I2, Q2, O2, N2> > &p);
```

説明

- 固定小数点の複素乗算を実行
- 3 乗算または 4 乗算の構造をインプリメント
- スカラーまたは std::complex インターフェイスをサポート

パラメーター

表 73: パラメーター

テンプレート パラメーター	説明
アーキテクチャ	3 乗算または 4 乗算のアーキテクチャを選択します。 CmpyThreeMult または CmpyFourMult を使用して指定します。
W1、I1、Q1、O1、N1	被乗数および乗数の固定小数点パラメーター。
W2、I2、Q2、O2、N2	積の固定小数点パラメーター。

引数

表 74: スカラー インターフェイス引数

引数	説明
ar	被乗数実数コンポーネント
ai	被乗数虚数コンポーネント
br	乗数実数コンポーネント

表 74: スカラー インターフェイス引数 (続き)

引数	説明
bi	乗数虚数コンポーネント
pr	積実数コンポーネント
pi	積虚数コンポーネント

表 75: `std::complex` インターフェイス引数

引数	説明
a	被乗数
b	乗数
p	積

戻り値

- 該当なし (void 関数)

サポートされる基本データ型

- `ap_fixed`
- `std::complex<ap_fixed>`

入力データの想定

- なし

HLS DSP サンプル デザイン

Vivado HLS の DSP サンプル デザインは、基本的なテストベンチを提供し、各関数のパラメーター設定方法およびインスタンス化方法を説明します。各関数に対し 1 つまたは複数の例があります。

Vivado HLS サンプル デザインを [Welcome] ページから開くには、[Open Example Project] をクリックします。Examples ウィザードで、[Design Examples]→[dsp] と順にクリックしてデザインを選択します。

注記: [Welcome] ページは、Vivado HLS の GUI を起動すると表示されます。[Help]→[Welcome] をクリックすると、いつでもこのページにアクセスできます。

また、Vivado Design Suite のインストール ディレクトリ `Vivado_HLS\2018.x\examples\design\dsp` から直接コード例を開くこともできます。

各サンプルには、次のファイルが含まれます。

- `<example>.cpp`: ライブラリ クラスをインスタンス化するための最上位合成ラッパー。
- `<example>.h`: パラメーター値を定義するヘッダー ファイル。
- `<example>_tb.cpp`: 最上位合成ラッパーを実行するための基本的なテストベンチ。
- `run_hls.tcl`: Vivado HLS のサンプル プロジェクトを設定する Tcl コマンド。

```
vivado_hls -f run_hls.tcl
```

注記: 一部のサンプル デザインには `directives.tcl` も含まれます。これは、最適化およびインプリメンテーションの指示子を適用するための追加 Tcl コマンドを提供するものです。

HLS SQL ライブラリ関数

このセクションでは、Vivado HLS の SQL ライブラリについて説明します。

`hls_alg::sha224`

コード例

```
namespace hls_alg {
template <typename msg_T, typename hash_T>
void sha224(hls::stream<msg_T>& msg_strm, uint64_t len,
hls::stream<hash_T>&
hash_strm);
}
```

説明

- 入力メッセージの SHA-224 ハッシュ値を計算します。
- メッセージ長は、バイト数で指定します。
- 複数の別の出力ストリーム幅が提供されます。

パラメーター

表 76: `hls_alg::sha224` パラメーター

パラメーター	説明
<code>msg_T</code>	入力ストリーム データ型。
<code>hash_T</code>	出力ストリーム データ型。

引数

表 77: `hls_alg::sha224` 引数

引数	説明
<code>msg_strm</code>	入力メッセージ ストリーム。
<code>len</code>	メッセージ長 (バイト)。
<code>hash_strm</code>	出力ハッシュ ストリーム。 <code>hash_T</code> が符号なしの <code>char</code> または符号なしの <code>int</code> の場合、呼び出し元はこのストリームをそれぞれ 28 回または 7 回読み込んで、224 ビットのフル メッセージを取得します。

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- msg_T: 符号なし char。
- hash_T: 符号なし char、符号なし int および ap_uint<224>。

入力データの想定

- msg_strm には len バイトに対して十分なデータが含まれると想定されますが、十分でない場合は、関数がブロックします。

hls_alg::sha256

コード例

```
namespace hls_alg {
template <typename msg_T, typename hash_T>
void sha256(hls::stream<msg_T>& msg_strm, uint64_t len,
hls::stream<hash_T>&
hash_strm);
}
```

説明

- 入力メッセージの SHA-256 ハッシュ値を計算します。
- メッセージ長は、バイト数で指定します。
- 複数の別の出力ストリーム幅が提供されます。

パラメーター

表 78: hls_alg::sha256 パラメーター

パラメーター	説明
msg_T	入力ストリーム データ型。
hash_T	出力ストリーム データ型。

引数

表 79: hls_alg::sha256 引数

引数	説明
msg_strm	入力メッセージ ストリーム。
len	メッセージ長 (バイト)。
hash_strm	出力ハッシュ ストリーム。 hash_T が符号なしの char または符号なしの int の場合、呼び出し元はこのストリームをそれぞれ 32 回または 8 回読み込んで、256 ビットのフル メッセージを取得します。

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- msg_T: 符号なし char。
- hash_T: 符号なし char、符号なし int および ap_uint<256>。

入力データの想定

- msg_strm には len バイトに対して十分なデータが含まれると想定されますが、十分でない場合は、関数がブロックします。

hls_alg::sort

コード例

```
namespace hls_alg {
template <typename T, uint64_t len>
void sort(hls::stream<T>& input, hls::stream<T>& output);
}
```

説明

- 入力ベクターを hls::stream 形式で分類します。
- ベクター長はテンプレート パラメーター len です。

パラメーター

表 80: hls_alg::sort

パラメーター	説明
T	ストリーム データ型。
len	ベクター長 (ストリーム形式) は 2 のべき乗にする必要があります。

引数

表 81: hls_alg::sort

引数	説明
入力	入力ストリーム
出力	出力ストリーム

戻り値

- 該当なし (void 関数)

サポートされるデータ型

- T: 演算子 [<] の定義を使用したクラス
- len: 2 のべき乗。

C の任意精度型

このセクションでは、次について説明します。

- C 言語デザイン用に Vivado HLS で提供される任意精度 (AP) 型。
- C の `int#w` 型用の関連関数。

[u]int#W 型のコンパイル

[u]int#W 型を使用するには、[u]int#W 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。

これら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。

[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`

説明:

- `#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、基数 (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` の値に変換します。どの基数でも、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#W apint_string2bits[_radix](const char*, int N)
```

これは、C 言語で許容されるよりも大きな値の整数定数を作成するのに使用されます。小さな値でも機能しますが、既存の C 言語の定数値で指定する方が簡単です。

```
#include <stdio.h>
#include "ap_cint.h"

int128 a;

// Set a to the value hex 00000000000000000000123456789ABCDEF0
a = apint_string2bits_hex("-123456789ABCDEF", 128);
```

値は文字列から直接代入することもできます。

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` の値に変換します。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 言語で許容されるよりも大きな値の整数定数を作成するのに使用されます。この関数は通常、ファイルから情報を読み出すためにテストベンチで使用されます。

`test.dat` ファイル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出力されます。

```
#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
        // 000000000000000000000000123456789ABCDF0
        // FFFFFFFFFFFFFFFFFFEDCBA9876543212
        // FFFFFFFFFFFFFFFFFFEEEEEEEEEEEEEEEEEEFC

        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 52719
printf( lu%40lu\n,c);
// lu 81985529216486895
printf(llu%40llu\n,c);
// llu 81985529216486895

printf( o%40o\n,c); // Unsigned integer in octal format
// o 21152746757
printf( ho%40ho\n,c);
// ho 146757
printf( lo%40lo\n,c);
// lo 4432126361152746757
printf(llo%40llo\n,c);
// llo 4432126361152746757

printf( x%40x\n,c); // Unsigned integer in hexadecimal format [0-9a-f]
// x 89abcdef
printf( hx%40hx\n,c);
```

```
// hx cdef
printf(  lx%40lx\n,c);
// lx 123456789abcdef
printf(llx%40llx\n,c);
// llx 123456789abcdef

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()

これは、C 言語で許容される値よりも大きな値の整数を表示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なります。

[illegible]

apint_fprint()

これは、C 言語で許容される値よりも大きな値の整数を表示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、通常との C 演算子を使用した式でも自由に使用できます。ただし、一部には予期しない動作が見られ、説明が必要なことがあります。

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、符号にかかわらず、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

ビット幅が大きいソース変数値を幅の小さなものへ代入すると、値が切り捨てられ、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失われます。

この切り捨てが実行されるとき、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。

任意精度演算を実行するため、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の、オペランドの幅および符号で決まります。戻り値の詳細は、各演算子のセクションで説明します。

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::RType [u]int#W::operator + ([u]int#W 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 1 つ) の和を計算します。

和の幅は、次のようになります。

- 2 つのオペランドの幅の大きいほうよりも 1 ビット大きくなる。
- 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る。

減算

```
[u]int#W::RType [u]int#W::operator - ([u]int#W op)
```

- 2 つの整数の差を計算する式です。
- 相違値の幅は、次のようになります。
 - 。 2 つのオペランドの幅の大きいほうよりも 1 ビット大きくなる。
 - 。 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かかわらず、差は符号付きとして処理されます。

乗算

```
[u]int#W::RType [u]int#W::operator * ([u]int#W op)
```

- 2つの整数値の積を返します。
- 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::RType [u]int#W::operator / ([u]int#W op)
```

- 2つの整数値の商を計算します。
- 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 を足したものの幅となります。
- 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。

剰余

```
[u]int#W::RType [u]int#W::operator % ([u]int#W op)
```

- 2つの整数値の整数除算の余りを返します。
- 剰余とオペランド両方に同じ符号が付いている場合、剰余の幅は、オペランドの幅の最小値です。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 を足したものの幅となります。

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ます。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 OR

```
[u]int#W::RType [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 AND (bitwise_and)

```
[u]int#W::RType [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 XOR

```
[u]int#W::RType [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)
```



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。ザイリンクスでは、予期しない動作を防ぐため、シフト式を代入される方の型に明示的に型変換することをお勧めしています。

複合代入演算子

Vivado HLS では、次の複合演算子がサポートされています。

- *=
- /=
- %=
- +=

- `--`
- `<<=`
- `>>=`
- `&=`
- `^=`
- `=`

RHS 式は計算されてから、基本演算子に RHS オペランドとして渡され、LHS 変数にその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

Vivado HLS では、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)
```

以下

```
bool [u]int#W::operator <= ([u]int#W op)
```

以上

```
bool [u]int#W::operator >= ([u]int#W op)
```

ビット レベルの演算: 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最適であることがよくあります。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_[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 レベル演算を実行するために、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)
```

ビット数を示す整数値を任意精度の整数値で返します。1 つの型または値で使用できます。

```
int5 Var1, Res1;

Var1 = -1;
Res1 = apint_bitwidthof(Var1); // Res1 is assigned 5
Res1 = apint_bitwidthof(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。



推奨: 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 型に明示的に型変換します。

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 ビットを選択し、それを返します。

ソースは [u]int#W 型、インデックス引数は 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ます。最下位ビットの指数は 0 です。最大指数はこの [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 の指定ビット、インデックス指定した値 (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) を指定します。
- 引数 Low は、範囲内の最下位ビット (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 の値より小さい場合、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)
```

- High および Low 間のソース指定ビットを 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)
```

- その値のビットすべてに AND 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。

```
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
```

- -1 との比較と同じです。一致すれば 1 が返されます。一致しない場合は 0 が返されます。すべてのビットが 1 であることから確認できます。

OR を使用した低減

apint_or_reduce()

```
int apint_or_reduce(int#N value)
```

- その値のビットすべてに OR 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。
- 0 との比較と同じで、一致すれば 0 が返され、一致しない場合は 1 が返されます。

```
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)
```

- その値のビットすべてに XOR 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。
- ワード内の 1 を数えることと同じです。この演算では次が実行されます。
 - 偶数がある場合は 0 が返されます。
 - 奇数 (偶数パリティ) がある場合は 1 が返されます。

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_xor_reduce(Var1); // Res1 is assigned 0

Var1= 1;
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)
```

- その値のビットすべてに NAND 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。

- -1 (すべて 1) に対してこの値を比較することと同じで、一致すれば false が返され、一致しない場合は true が返されます。

```
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)
```

- その値のビットすべてに NOR 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。
- 0 (すべて 0) に対してこの値を比較することと同じで、一致すれば true が返され、一致しない場合は false が返されます。

```
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)
```

- その値のビットすべてに XNOR 演算を適用します。
- 結果の単一ビットを整数値として返します (これはブール型に変換可能)。
- ワード内の 1 を数えることと同じです。
- この演算では次が実行されます。
 - 奇数がある場合は 0 が返されます。

- 。 偶数 (奇数パリティ) がある場合は 1 が返されます。

```
int5 Var1, Res1;

Var1= 0;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 1

Var1= 1;
Res1 = apint_xnor_reduce(Var1); // Res1 is assigned 0
```

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]int<> 型のコンパイル

`ap_[u]int<>` クラスを使用するには、`ap_[u]int<>` 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際は、`g++` コンパイルに `-I/<HLS_HOME>/include` オプションを追加するなど、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。

ap_[u] 変数の宣言/定義

次のようにそれぞれ符号付きおよび符号なしのクラスが別々にあります。

- `ap_int<int_W>` (符号付き)
- `ap_uint<int_W>` (符号なし)

テンプレート パラメーター `int_W` は宣言されている変数の合計幅を指定します。

次の例に示すように、C/C++ の `typedef` 文を使用してユーザー定義型を作成できます。

```
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
```

デフォルトの幅の最大許容幅は 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。

次に、AP_INT_MAX_W を上書きする例を示します。

```
#define AP_INT_MAX_W 4096 // Must be defined before next line
#include "ap_int.h"

ap_int<4096> very_wide_var;
```

定数 (リテラル) からの初期化および代入

クラス コンストラクター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できます。

ただし、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るので、通常、整数リテラルの 64 ビット制限 (LL または ULL 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ー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 (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++ 形式の型変換をする必要があります。

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", 16); // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567", 16);
```

注記: 協調シミュレーション中に予期しない結果にならないようにするには、`ap_uint<N> a = {0}` は初期化しないようにしてください。

`ap_[u]<>` コンストラクターは、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できます。コンストラクターの呼び出しに 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); // COMPILER-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、コンストラクター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 hexadecimal format

a_6bit_var = ap_int<6>("0b42", 2); // COMPILER-TIME ERROR! "42" is not binary
```

ビット幅が 53 ビットを超える場合、`ap_[u]fixed` 値は次のように文字列で初期化する必要があります。

```
ap_ufixed<72,10> Val("2460508560057040035.375");
```

コンソール I/O (出力) のサポート

Vivado HLS には、`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。

C++ の標準出力ストリームの使用

`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、次の C++ の標準出力ストリームのいずれかを使用する方法です。

```
std::cout (#include <iostream> または <iostream.h>)
```

ストリーム挿入演算子 `<<` は、任意の `ap_[u]fixed` 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。次のストリーム マニピュレーターもサポートされています。

- `dec` (10 進数)
- `hex` (16 進数)
- `oct` (8 進数)

これにより、値を指定どおりにフォーマットできます。

次の例では、値を出力するのに `cout` が使用されています。

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_ufixed<72> Val("10fedcba9876543210");

cout << Val << endl; // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

標準 C ライブラリの使用

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。

1. `ap_[u]fixed` クラスの `to_string()` メソッドを使用し、値を C++ の `std::string` に変換します。
2. その結果を `std::string` クラスの `c_str()` メソッドを使用して、ヌルで終了する C 文字に変換します。

オプションの引数 1 (基数の指定)

`ap_uint::to_string()` メソッドで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。有効な基数の引数値は、次のとおりです。

- 2 (2 進数) (デフォルト)
- 8 (8 進数)
- 10 (10 進数)
- 16 (16 進数)

オプションの引数 2 (符号付きの値として出力)

`ap_uint::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。

次の例では、値を出力するのに `printf` が使用されています。

```
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()); // => "-7F0123456789ABCDEF0"
```

ap_[u]<> 型を使用した式

`ap_[u]<>` 型の変数は、通常 C/C++ 演算子を使用した式では自由に使用できます。ただし、一部には予期しない動作が見られることがあります。これについては、次を参照してください。

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 (`ap_int<>`) の値を幅の大きなものへ代入すると、符号にかかわらず、値はデステーション変数の幅 (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。次に例を示します。

```
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

ビット幅が大きいソース変数値を幅の小さなものへ代入すると、値が切り捨てられ、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失われます。

この切り捨てが実行されるとき、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 メソッドおよび演算子

`ap_uint` 型では、wide `ap_uint` (> 64 ビット) からビルトイン C/C++ 整数型への暗示的な変換はサポートされません。たとえば次のコード例の場合、if 文内で `ap_int[65]` から `bool` 型への暗示的な型変換により 0 が返されるので、戻り値は 1 になります。

```
bool nonzero(ap_uint<65> data) {
    return data; // This leads to implicit truncation to 64b int
}

int main() {
    if (nonzero((ap_uint<65>)1 << 64)) {
        return 0;
    }
    printf(FAIL"\n");
    return 1;
}
```

幅の広い `ap_uint` 型をビルトインの整数に変換するには、`ap_uint` に含まれる次の明示的な変換関数を使用します。

- `to_int()`
- `to_long()`
- `to_bool()`

通常、ネイティブの C/C++ 整数型で実行可能な演算は、`ap_uint` 型で演算子をオーバーロードするとサポートされます。

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実行できます。

2 進数の演算子

任意精度演算を実行するため、標準 2 進数整数演算子がオーバーロードされます。次の演算子では次のいずれかが使用されます。

- `ap_uint` の 2 つのオペランド
- `ap_uint` 型 1 つと C/C++ 整数型 1 つ

次に例を示します。

- `char`
- `short`
- `int`

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の、オペランドの幅および符号で決まります。戻り値の詳細は、各演算子のセクションで説明します。

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)
```

次の和が返されます。

- 2 つの `ap_[u]int`、または
- 1 つの `ap_[u]int` および C/C++ 整数型

和の幅は、次のようになります。

- 2 つのオペランドの幅の広い方に 1 ビット追加。
- 幅の広い方が符号なしで幅の狭い方が符号付きの場合は 2 ビット追加。

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る。

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。

差の値の幅は、次のようになります。

- 2 つのオペランドの幅の広い方に 1 ビット追加。
- 幅の広い方が符号なしで幅の狭い方が符号付きの場合は 2 ビット追加。

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かかわらず、差は符号付きとして処理されます。

乗算

```
ap_(u)int::RType ap_(u)int::operator * (ap_(u)int op)
```

2 つの整数値の積を返します。

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::RType ap_(u)int::operator / (ap_(u)int op)
```

2 つの整数値の商を計算します。

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 を足したものの幅となります。

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。

剰余

```
ap_(u)int::RType ap_(u)int::operator % (ap_(u)int op)
```

2 つの整数値の整数除算の余りを返します。

剰余とオペランド両方に同じ符号が付いている場合、剰余の幅は、オペランドの幅の最小値です。

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。



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ス化されます。

次は、演算子の例です。

```
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ます。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 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 AND (bitwise_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 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 - ()
```

次が返されます。

- 符号付きの場合は同じ幅でオペランドのネゲートされた値。
- 符号なしの場合はその幅に 1 を足した値。

戻り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 ~ ()
```

同じ幅および符号の、オペランドのビット単位の NOT を返します。

論理の反転

```
bool ap_(u)int::operator ! ()
```

オペランドが 0 ではない場合にブール値 `false` を返します。

オペランドが 0 の場合は `true` を返します。

三項演算子

標準 C int 型で三項演算子を使用する場合は、1つのデータ型から別のデータ型に明示的に変換して、両方の結果が同じデータ型になるようにします。次に例を示します。

```
// Integer type is cast to ap_int type
ap_int<32> testc3(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<32>(a):b;
}
// ap_int type is cast to an integer type
ap_int<32> testc4(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?a+1:(int)b;
}
// Integer type is cast to ap_int type
ap_int<32> testc5(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<33>(a):b+1;
}
```

シフト演算子

各シフト演算子には次の2つのバージョンがあります。

- 符号なしの右辺 (RHS) オペランド
- 符号付きの右辺 (RHS) オペランド

符号付き RHS に負の値が与えられるとシフト演算の方向が逆になります。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方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。ザイリンクスでは、予期しない動作を防ぐため、シフト式を代入される方の型に明示的に型変換することをお勧めしています。

次は、シフト演算の例です。

```
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

Vivado HLS では、次の複合演算子がサポートされています。

- *=
- /=
- %=
- +=
- -=
- <<=
- >>=
- &=
- ^=
- |=

RHS 式はまず評価されてから RHS オペランドとして基本演算子に供給され、結果が LHS 変数に代入されます。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。

```
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。

プレインクリメント

```
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す。

そのインクリメントされた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、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

Vivado HLS では、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++ 基本整数型と比較できます。

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)
```

以下

```
bool ap_(u)int::operator <= (ap_(u)int op)
```

以上

```
bool ap_(u)int::operator >= (ap_(u)int op)
```


その他のクラス メソッド、演算子、データ メンバー

次のセクションでは、その他のクラス メソッド、演算子、およびデータ メンバーについて説明します。

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 レベル演算を実行するために、次のメソッドが提供されています。

長さ

```
int ap_(u)int::length ()
```

`ap_[u]int` 変数の合計ビット数を表す整数値を返します。

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下位ビットが出力されます。`concat()` メソッドの場合は下位ビットが出力されます。

オーバーロードされたカンマ演算子を使用するときはかっこが必要です。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。



推奨: 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 `ap_[u]int` 型に明示的に型変換します。

```
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 ビットを選択し、それを返します。

戻り値は、`ap_[u]int` 変数の対応するビットをセットまたはクリアできる参照値です。

ビット引数は `int` 値にする必要があります。選択するビットの指数を指定します。最下位ビットの指数は 0 です。最大指数はこの `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ています。

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)、`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` の値より小さい場合、ビットは逆順で返されます。

```
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(3,0) = Val2(4, 1); // Yields: 0x55
Rslt = Val1.range(4, 7); // Yields: 0xA; bit-reversed!
```

and_reduce

```
bool ap_(u)int::and_reduce ()
```

- この `ap_(u)int` のすべてのビットに AND 演算を適用します。
- 結果の単一ビットを返します。
- この値を -1 (すべて 1) と比較することと同じで、一致する場合は `true` が返され、一致しない場合は `false` が返されます。

or_reduce

```
bool ap_(u)int::or_reduce ()
```

- この `ap_(u)int` のすべてのビットに OR 演算を適用します。
- 結果の単一ビットを返します。
- この値を 0 (すべて 0) と比較することと同じで、一致する場合は `false` が返され、一致しない場合は `true` が返されます。

xor_reduce

```
bool ap_(u)int::xor_reduce ()
```

- この `ap_int` のすべてのビットに XOR 演算を適用します。
- 結果の単一ビットを返します。

- この値の 1 のビット数を数えて、偶数の場合は `false`、奇数の場合は `true` を返すことと同じです。

nand_reduce

```
bool ap_(u)int::nand_reduce ()
```

- この `ap_int` のすべてのビットに NAND 演算を適用します。
- 結果の単一ビットを返します。
- この値を -1 (すべて 1) と比較することと同じで、一致する場合は `false` が返され、一致しない場合は `true` が返されます。

nor_reduce

```
bool ap_int::nor_reduce ()
```

- この `ap_int` のすべてのビットに NOR 演算を適用します。
- 結果の単一ビットを返します。
- この値を 0 (すべて 0) と比較することと同じで、一致する場合は `true` が返され、一致しない場合は `false` が返されます。

xnor_reduce

```
bool ap_(u)int::xnor_reduce ()
```

- この `ap_(u)int` のすべてのビットに XNOR 演算を適用します。
- 結果の単一ビットを返します。
- この値の 1 のビット数を数えて、偶数の場合は `true`、奇数の場合は `false` を返すことと同じです。

ビット リダクション メソッドの例

```
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
```

ビットの逆転

```
void ap_(u)int::reverse ()
```

`ap_[u]int` インスタンスの内容を逆転します。

- LSB は MSB になります。

- MSB は LSB になります。

逆転メソッドの例

```
ap_uint<8> Val = 0x12;
Val.reverse(); // Yields: 0x48
```

ビット値のテスト

```
bool ap_(u)int::test (unsigned i)
```

`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 の場合は true、それ以外の場合は false を返します。

テスト メソッドの例

```
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)
```

`ap_(u)int` インスタンス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)
```

`ap_(u)int` インスタンスの指定ビットを 1 に設定します。

ビットのクリア (0)

```
void ap_(u)int::clear(unsigned i)
```

`ap_(u)int` インスタンスの指定ビットを 0 に設定します。

ビットの反転

```
void ap_(u)int::invert(unsigned i)
```

`ap_(u)int` インスタンスの関数引数で指定したビットを反転します。指定したビットは、元の値が 0 の場合は 1 になり、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(4, false); // Yields: 0x03
Val.set(7); // Yields: 0x83
Val.clear(1); // Yields: 0x81
Val.invert(4); // Yields: 0x91
```

右に回転

```
void ap_(u)int:: rrotate(unsigned n)
```

`ap_(u)int` インスタンスを `n` 桁右に回転します。

左に回転

```
void ap_(u)int:: lrotate(unsigned n)
```

`ap_(u)int` インスタンスを `n` 桁左に回転します。

```
ap_uint<8> Val = 0x12;
Val.rrotate(3); // Yields: 0x42
Val.lrotate(6); // Yields: 0x90
```

ビット単位 NOT (bitwise_not)

```
void ap_(u)int:: b_not()
```

- `ap_(u)int` インスタンスのすべてのビットの 補数を求めます。

```
ap_uint<8> Val = 0x12;
Val.b_not(); // Yields: 0xED
```

ビット単位 NOT の例

符号のテスト

```
bool ap_int:: sign()
```

- `ap_(u)int` インスタンスが負の値であるかどうかをチェックします。
- 負の値の場合は `true` が返されます。
- 正の値の場合は `false` が返されます。

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 ()
```

- `ap_[u]int` に含まれる値のネイティブ C/C++ (ほとんどのシステムで 32 ビット) 整数を返します。
- 値が `[unsigned] int` で表示されるよりも大きい場合は、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 ()
```

- `ap_[u]int` に含まれる値のネイティブ C/C++ の 64 ビットの整数を返します。
- 値が `[unsigned] int` で表示されるよりも大きい場合は、切り捨てが実行されます。

C/C++ の double 型

```
double ap_(u)int::to_double ()
```

- `ap_[u]int` に含まれるネイティブ C/C++ の `double` の 64 ビット浮動小数点表記を返します。
- `ap_[u]int` が 53 ビット (`double` の仮数のビット数) より大きい場合、結果の `double` に正確な値が含まれないことがあります。



推奨: `ap_[u]int` をほかのデータ型に変換するには、ザイリンクスでは C 形式のキャストを使用する代わりに、メンバー関数を明示的に呼び出すことをお勧めします。

sizeof

標準 C++ `sizeof()` 関数は、`ap_[u]int` やその他のクラスまたはオブジェクトのインスタンスと一緒に使用できません。`ap_int<>` 型はクラスで、`sizeof` からはそのクラスまたはインスタンス オブジェクトで使用されたストレージが返されます。`sizeof(ap_int<N>)` からは、常に使用されたバイト数が返されます。次に例を示します。

```
sizeof(ap_int<127>)=16
sizeof(ap_int<128>)=16
sizeof(ap_int<129>)=24
sizeof(ap_int<130>)=24
```

コンパイル時のデータ型属性へのアクセス

`ap_[u]int<>` 型には、変数のサイズをコンパイル時に決定できるようにするスタティック メンバーが含まれます。このデータ型に含まれる `static const` メンバーの `width` により、自動的にデータ型の幅が割り当てられます。

```
static const int width = _AP_W;
```

既存の `ap_[u]int<>` 型を抽出するのに `width` データ メンバーを使用すると、コンパイル時に別の `ap_[u]int<>` 型を作成できます。次の例に、変数 `Res` のサイズを変数 `Val1` と `Val2` 変数よりも 1 ビット大きく定義する方法を示します。

```
// Definition of basic data type
#define INPUT_DATA_WIDTH 8
typedef ap_int<INPUT_DATA_WIDTH> data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at compile-time to be 1-bit greater than data
type
data_t
ap_int<data_t::width+1> Res = Val1 + Val2;
```

これにより、`data_t` の `INPUT_DATA_WIDTH` の値がアップデートされた場合でも、Vivado HLS で加算によるビット増加が正しく認識されるようになります。

C++ の任意精度固定小数点型

Vivado HLS では、小数部分の計算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。次の例に、固定小数点の演算の利点を示します。

```
ap_fixed<11, 6> Var1 = 22.96875; // 11-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5; // 12-bit word, 1 fractional bit
ap_fixed<16,11> Res1; // 16-bit signed word, 5 fractional bits

Res1 = Var1 + Var2; // Result is 535.46875
```

`Var1` および `Var2` の精度は異なりますが、固定小数点型を使用すると、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。C コードでは、小数点を揃えるために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データ型は、整数ビットおよび小数ビットの両方を完全に格納するのに十分な大きさにする必要があります。

そうでない場合、`ap_fixed` 型で次が実行されます。

- オーバーフロー処理 (指定されたデータ型でサポートされるよりも MSB が多い場合)
- 量子化 (指定されたデータ型でサポートされるよりも LSB が少ない場合)

`ap_[u]fixed` 型では、オーバーフローおよび量子化にさまざまなオプションを使用できます。次のセクションで、その詳細を説明します。

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 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ます。
- 右側にあるビットは値の小数部を示します。

`ap_[u]fixed` type は、次のように定義されています。

```
ap_[u]fixed<int W,  
int I,  
ap_q_mode Q,  
ap_o_mode O,  
ap_sat_bits N>;
```

量子化モード

正の無限大への丸め	<code>AP_RND</code>
0 への丸め	<code>AP_RND_ZERO</code>
負の無限大への丸め	<code>AP_RND_MIN_INF</code>
無限大への丸め	<code>AP_RND_INF</code>
Convergent rounding	<code>AP_RND_CONV</code>
切り捨て	<code>AP_TRN</code>
Truncation to zero	<code>AP_TRN_ZERO</code>

AP_RND

- 値を特定の `ap_[u]fixed` 型で表現可能な最も近い値に丸めます。

```
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

- 値を表示可能な最も近い値に丸めます。
- 0 の方向へ丸めます。
 - 正の値の場合は、冗長ビットを削除します。
 - 負の値の場合は、表示可能な最も近い値になるよう下位ビットを追加します。

```
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

- 値を表示可能な最も近い値に丸めます。
- 負の無限大の方向へ丸めます。
 - 正の値の場合は、冗長ビットを削除します。
 - 負の値の場合は、下位ビットを追加します。

```
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

- 値を表示可能な最も近い値に丸めます。
- 丸めは LSB によって異なります。
 - 。 正の値では、LSB が設定されている場合は正の無限大の方向へ丸めます。それ以外の場合は負の無限大の方向へ丸めます。
 - 。 負の値では、LSB が設定されている場合は負の無限大の方向へ丸めます。それ以外の場合は正の無限大の方向へ丸めます。

```
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

- 値を表示可能な最も近い値に丸めます。
- 丸めは LSB によって異なります。
 - 。 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ます。
 - 。 それ以外の場合は負の無限大の方向へ丸めます。

```
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

- 値を常に負の無限大の方向へ丸めます。

```
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

値を次のように丸めます。

- 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、ゼロの方向へ丸めます。

```
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

0 への飽和

対称飽和

折り返し

符号絶対値の折り返し

AP_SAT

AP_SAT_ZERO

AP_SAT_SYM

AP_WRAP

AP_WRAP_SM

AP_SAT

値が次のように飽和されます。

- オーバーフローが発生したときは値が最大値に飽和されます。
- 負のオーバーフローのときは最小値に飽和されます。

```
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: -8.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_ZERO

オーバーフローまたは負のオーバーフローの場合に値を 0 にします。

```
ap_fixed<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_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_SYM

値が次のように飽和されます。

- オーバーフローが発生したときは値が最大値に飽和されます。
- 負のオーバーフローのときは最小値に飽和されます。
 - 符号付き `ap_fixed` 型の場合は負の最大値
 - 符号なし `ap_ufixed` 型の場合は 0

```
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 7.0
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: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_WRAP

オーバーフローが発生したときは値が折り返されます。

```
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; // Yields: -1.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: -3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; // Yields: 3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: 13.0
```

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。
- 符号なしの場合は、最大値に達すると 0 に戻ります。
- 符号付きの場合は、最大値に達すると最小値に戻ります。

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

値が符号絶対値で折り返されます。

```
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。
 1. 重複している MSB が削除されます。
 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。
 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<> variables 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、g++ 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。

ap_[u]fixed<> 変数の宣言と定義

次のようにそれぞれ符号付きおよび符号なしのクラスが別々にあります。

- ap_fixed<W,I> (符号付き)
- ap_ufixed<W,I> (符号なし)

C/C++ の typedef 文を使用すると、ユーザー定義型を作成できます。

```
#include "ap_fixed.h" // use ap_[u]fixed<> types

typedef ap_ufixed<128,32> uint128_t; // 128-bit user defined type,
// 32 integer bits
```

ユーザー定義型の例

定数 (リテラル) からの初期化および代入

ap_[u]fixed 変数は、次の一般的な 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、単精度または倍精度の浮動小数点です。

固定小数点変数に割り当てられる値は、定数の精度によって制限されます。上記の[定数 \(リテラル\) からの初期化および代入](#)の基準に従って、固定小数点変数のすべてのビットがストリングで記述された精度に従って生成されるようにしてください。

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
ap_fixed<36, 30> = -0x123.456p-1;
```

ap_[u]fixed 型では、それらが std::complex 型の配列で 사용되는場合は、初期化がサポートされません。

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {{ 1, -0 }, { 0.9, -0.006 }, etc.}
```

初期化値は、まず std::complex に変換する必要があります。

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {std::complex<coeff_t>( 1,
-0 ),
std::complex<coeff_t>(0.9, -0.006 ), etc.}
```

コンソール I/O (出力) のサポート

Vivado HLS には、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]fixed 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、次の C++ の標準出力ストリーム std::cout (#include <iostream> or <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]fixed 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。次のストリーム マニピュレーターもサポートされているので、それぞれの進数値をフォーマットできます。

- dec (10 進数)
- hex (16 進数)
- oct (8 進数)

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_fixed<6,3, AP_RND, AP_WRAP> Val = 3.25;

cout << Val << endl;      // Yields: 3.25
```

標準 C ライブラリの使用

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。

1. `ap_[u]fixed` クラスの `to_string()` メソッドを使用し、値を C++ の `std::string` に変換します。
2. その結果を `std::string` クラスの `c_str()` メソッドを使用して、ヌルで終了する C 文字に変換します。

オプションの引数 1 (基数の指定)

`ap[u]int::to_string()` メソッドで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。有効な基数の引数値は、次のとおりです。

- 2 (2 進数)
- 8 (8 進数)
- 10 (10 進数)
- 16 (16 進数) (デフォルト)

オプションの引数 2 (符号付きの値として出力)

`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。

```
ap_fixed<6,3, AP_RND, AP_WRAP> Val = 3.25;

printf("%s \n", in2.to_string().c_str()); // Yields: 0b011.010
printf("%s \n", in2.to_string(10).c_str()); //Yields: 3.25
```

`ap_[u]fixed` 型は、次の C++ マニピュレーター関数でサポートされます。

- `setprecision`
- `setw`
- `setfill`

`setprecision` マニピュレーターでは、使用する小数点以下の精度を設定します。1 つのパラメーター `f` を小数点以下の精度の値として使用します。`n` は表示する合計有効桁数の最大値 (小数点の前と後の両方をカウント) を指定します。

f のデフォルト値は 6 で、ネイティブ C の float 型と一貫しています。

```
ap_fixed<64, 32> f = 3.14159;
cout << setprecision (5) << f << endl;
cout << setprecision (9) << f << endl;
f = 123456;
cout << setprecision (5) << f << endl;
```

上記を実行すると、実際の精度が指定した精度を超える場合に丸められ、次の結果が表示されます。

```
3.1416
3.14159
1.2346e+05
```

setw マニピュレーターは、次を実行します。

- フィールド幅に使用される文字数を設定します。
- 1 つのパラメーター w を幅の値として使用します。

説明:

- w: 出力表示で使用する最小文字数を指定します。

表示の標準幅がフィールド幅よりも短い場合は、埋め字で補足されます。埋め字は setfill マニピュレーターで制御され、1 つのパラメーター f が補足文字として使用されます。

次に例を示します。

```
ap_fixed<65,32> aa = 123456;
int precision = 5;
cout<<setprecision(precision)<<setw(13)<<setfill('T')<<a<<endl;
```

この出力は、次のようになります。

```
TTT1.2346e+05
```

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、その使用方法是 C/C++ 言語のほかの浮動小数点型または変数と同じです。

次に注意してください。

- 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ます。ビット幅が小さい方の値を大きな方の値に代入するとき、符号を得るには型変換 (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

デスティ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する場合は、切り捨てが実行されます。

クラス メソッド、演算子、データ メンバー

通常、ネイティブの C/C++ 整数型で実行可能な有効な演算は、`ap_[u]fixed` 型で演算子のオーバーロードを使用するとサポートされます。オーバーロードされた演算子に加え、ビット レベルの演算を実行しやすくする演算子およびメソッドが含まれています。

2 進数の算術演算子

加算

```
ap_[u]fixed::RType ap_[u]fixed::operator + (ap_[u]fixed op)
```

任意精度の固定小数点と指定したオペランド `op` を加算します。

オペランドには、次のいずれかの整数型を使用できます。

- `ap_[u]fixed`
- `ap_[u]int`
- C/C++

結果のデータ型 `ap_[u]fixed::RType` は、2 つのオペランドのデータ型によります。

```
ap_fixed<76, 63> Result;

ap_fixed<5, 2> Val1 = 1.125;
ap_fixed<75, 62> Val2 = 6721.35595703125;

Result = Val1 + Val2; //Yields 6722.480957
```

`Val2` の方が整数部および小数部のビット幅が広いいため、結果値をすべて格納できるようにするため、結果のデータ型の幅はこのビット幅に 1 を足したものになります。

データの幅を指定すると、次に示すように、べき関数を使用してリソースが制御されます。同様に、ザイリンクスでは、固定小数点演算の幅を指定する代わりに、格納された結果の幅を指定することをお勧めします。

```
ap_ufixed<16,6> x=5;
ap_ufixed<16,7>y=hl::rsqrt<16,6>(x+x);
```

減算

```
ap_[u]fixed::RType ap_[u]fixed::operator - (ap_[u]fixed op)
```

任意精度の固定小数点値から指定したオペランド `op` を減算します。

結果のデータ型 `ap_[u]fixed::RType` は、2 つのオペランドのデータ型によります。

```
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 の方が整数部および小数部のビット幅が広い場合、結果値をすべて格納できるようにするため、結果のデータ型の幅はこのビット幅に 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)
```

任意精度の固定小数点値を指定したオペランド op と乗算します。

```
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
```

この例では、Val1 と Val2 を乗算しています。この結果のデータ型は、整数部のビット幅と小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)
```

任意精度の固定小数点値を指定したオペランド op で除算します。

```
ap_fixed<84, 66> Result;

ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Val2 / Val1; // Yields 5974.538628
```

この例では、Val1 で Val2 を除算しています。十分な精度を保持するため、次のようになります。

- 結果のデータ型の整数ビット幅は、Val2 の整数ビット幅と、Val1 の小数ビット幅の和になります。
- 結果のデータ型の小数ビット幅は、Val2 の小数ビット幅と等しくなります。

ビット単位の論理演算子

ビット単位 OR

```
ap_[u]fixed::RType ap_[u]fixed::operator | (ap_[u]fixed op)
```

任意精度の固定小数点と任意のオペランド op にビット単位演算を適用します。

```
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 AND

```
ap_[u]fixed::RType ap_[u]fixed::operator & (ap_[u]fixed op)
```

任意精度の固定小数点と任意のオペランド `op` にビット単位演算を適用します。

```
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 XOR

```
ap_[u]fixed::RType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

任意精度の固定小数点と任意のオペランド `op` に `xor` ビット単位演算を適用します。

```
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

前置インクリメント

```
ap_[u]fixed ap_[u]fixed::operator ++ ()
```

任意精度の固定小数点変数を 1 インクリメントします。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ++Val1; // Yields 6.125000
```

後置インクリメント

```
ap_[u]fixed ap_[u]fixed::operator ++ (int)
```

この演算関数は、次を実行します。

- 任意精度の固定小数点変数を 1 インクリメント。
- この任意精度の固定小数点の元の値を返す。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1++; // Yields 5.125000
```

前置デクリメント

```
ap_[u]fixed ap_[u]fixed::operator -- ()
```

任意精度の固定小数点変数を 1 デクリメントします。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = --Val1; // Yields 4.125000
```

後置デクリメント

```
ap_[u]fixed ap_[u]fixed::operator -- (int)
```

この演算関数は、次を実行します。

- 任意精度の固定小数点変数を 1 デクリメント。
- この任意精度の固定小数点の元の値を返す。

```
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。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1; // Yields 5.125000
```

減算

```
ap_[u]fixed::RTyp 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。

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = -Val1; // Yields -5.125000
```

0 との等価比較

```
bool ap_[u]fixed::operator ! ()
```

この演算関数は、次を実行します。

- 任意精度の固定小数点変数を 0 と比較。
- 結果を返す。

```
bool Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1; // Yields false
```

ビット単位の反転

```
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。

```
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)
```

この演算関数は、次を実行します。

- 指定した整数オペランド分左にシフト。
- 結果を返す。

オペランドには次の C/C++ の整数型を使用できます。

- char
- short
- int
- long

左シフト演算で返されるデータ型は、シフトされたデータ型と同じ幅になります。

注記: 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されていません。

```
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。左シフト演算の結果のデータ型は Val と同じであるため、次のようになります。

- Val の上位 2 ビットがシフト アウトされます。
- 結果は -10.5 になります。

結果が 21.5 になるようにするには、`ap_ufixed<10, 7>(Val)` のように、まず `Val` を `ap_fixed<10, 7>` にキャスト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)
```

この演算子は、次を実行します。

- 指定した整数オペランド分左にシフト。
- 結果を返す。

オペランドが正か負かによって、シフトの方向が変わります。

- オペランドが正の場合は右シフトが実行されます。
- オペランドが負の場合は左シフト (逆方向) が実行されます。

オペランドには次の C/C++ の整数型を使用できます。

- `char`
- `short`
- `int`
- `long`

右シフト演算で返されるデータ型は、シフトされたデータ型と同じ幅になります。

```
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)
```

この演算関数は、次を実行します。

- 指定した整数オペランド分右にシフト。
- 結果を返す。

オペランドには次の C/C++ の整数型を使用できます。

- `char`
- `short`
- `int`
- `long`

右シフト演算で返されるデータ型は、シフトされたデータ型と同じ幅になります。

```
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` の小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)
```

この演算子は、次を実行します。

- 指定した整数オペランド分右にシフト。
- 結果を返す。

オペランドが正か負かによって、シフトの方向が変わります。

- オペランドが正の場合は右シフトが実行されます。
- オペランドが負の場合は左シフト (逆方向) が実行されます。

オペランドには C/C++ の整数型 (`char`、`short`、`int`、または `long`) を使用できます。

右シフト演算で返されるデータ型は、シフトされるデータ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
```

比較演算子

等価比較

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

任意精度の固定小数点変数と指定したオペランドを比較します。

等しい場合は `true`、等しくない場合は `false` を返します。

オペランド `op` のデータ型には、`ap_[u]fixed`、`ap_int` または C/C++ の整数型を使用できます。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
```

不等価比較

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

任意精度の固定小数点変数を指定したオペランドと比較します。

等しくない場合は `true`、等しい場合は `false` を返します。

オペランド `op` のデータ型には、次を使用できます。

- `ap_[u]fixed`
- `ap_int`
- C または C++ の整数型

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
```

以上

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

変数を指定のオペランドと比較します。

変数がオペランド以上の場合は `true`、そうでない場合は `false` を返します。

オペランド `op` のデータ型には、`ap_[u]fixed`、`ap_int` または C/C++ の整数型を使用できます。

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
```

以下

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

変数と指定のオペランドを比較し、変数がオペランド以下の場合は `true`、そうでない場合は `false` を返します。

オペランド `op` のデータ型には、`ap_[u]fixed`、`ap_int` または C/C++ の整数型を使用できます。

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)
```

変数を指定のオペランドと比較し、変数がオペランドより大きい場合は `true`、そうでない場合は `false` を返します。

オペランド `op` のデータ型には、`ap_[u]fixed`、`ap_int` または C/C++ の整数型を使用できます。

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)
```

変数と指定のオペランドを比較し、変数がオペランド未満の場合は `true`、そうでない場合は `false` を返します。

オペランド `op` のデータ型には、`ap_[u]fixed`、`ap_int` または C/C++ の整数型を使用できます。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

ビットの選択と設定

```
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。

戻り値は、`ap_[u]fixed` 変数の対応するビットをセットまたはクリアできる参照値です。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。最下位ビットの指数は 0 です。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。

結果のデータ型は `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 セレクト演算子 `[]` に似ていますが、1 ビットではなくビットの範囲を指定し、

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` は範囲の上限のビットを指定し、引数 `Lo` は下限の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順で返されます。

戻り値のデータ型 `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。

戻り値のデータ型 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 ()
```

ビット数を表す整数値を任意精度の整数値で返します。データ型または値に使用できます。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 ()
```

このメンバー関数は、固定小数点値を IEEE の倍精度フォーマットに変換します。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
double Result;

Result = MyAPFixed.to_double(); // Yields 333.789
```

固定小数点から浮動小数点

```
float ap_[u]fixed::to_float()
```

このメンバー関数は、固定小数点値を IEEE の浮動精度フォーマットに変換します。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
float Result;

Result = MyAPFixed.to_float(); // Yields 333.789
```

固定小数点から半精度浮動小数点

```
half ap_[u]fixed::to_half()
```

このメンバー関数は、固定小数点値を HLS の半精度 (16 ビット) 浮動精度フォーマットに変換します。次に例を示します。

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
half Result;

Result = MyAPFixed.to_half(); // Yields 333.789
```

固定小数点から ap_int 型

```
ap_int ap_[u]fixed::to_ap_int ()
```

このメンバー関数は、この固定小数点値を、小数ビットを切り捨てた整数ビットのみを含む ap_int 型に変換します。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 ()
```

このメンバー関数は、固定小数点値を C のビルトイン整数型に変換します。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
```



推奨: ap_[u]fixed をほかのデータ型に変換するには、サイリンクスでは C 形式のキャストを使用する代わりに、メンバー関数を明示的に呼び出すことをお勧めします。

コンパイル時のデータ型属性へのアクセス

ap_[u]fixed<> 型には、データ型のサイズと設定がコンパイル時に決定されるようにする複数のスタティック メンバーがあります。このデータ型には、static const メンバーの、width、iwidth、qmode および omode があります。

```
static const int width = _AP_W;
static const int iwidth = _AP_I;
static const ap_q_mode qmode = _AP_Q;
static const ap_o_mode omode = _AP_O;
```

これらのデータ メンバーは、既存の `ap_[u]fixed<>` 型から次の情報を抽出するために使用できます。

- `width`: データ型の幅。
- `iwidth`: データ型の整数部分の幅。
- `qmode`: データ型の量子化モード。
- `omode`: データ型のオーバーフロー モード。

これらのデータ メンバーを使用すると、たとえば既存の `ap_[u]fixed<>` 型のデータ幅を抽出して、コンパイル時に別の `ap_[u]fixed<>` 型を作成できます。

次の例は、変数 `Res` のサイズを、変数 `Val1` および `Val2` よりも 1 ビット大きくし、同じ量子化モードで定義します。

```
// Definition of basic data type
#define INPUT_DATA_WIDTH 12
#define IN_INTG_WIDTH 6
#define IN_QMODE AP_RND_ZERO
#define IN_OMODE AP_WRAP
typedef ap_fixed<INPUT_DATA_WIDTH, IN_INTG_WIDTH, IN_QMODE, IN_OMODE>
data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at run-time to be 1-bit greater than
INPUT_DATA_WIDTH
// The bit growth in Res will be in the integer bits
ap_int<data_t::width+1, data_t::iwidth+1, data_t::qmode, data_t::omode> Res
= Val1 +
Val2;
```

これにより、`INPUT_DATA_WIDTH`、`IN_INTG_WIDTH`、または `data_t` の量子化モードの値をアップデートした場合でも、Vivado HLS で加算によるビット増加が正しくモデリングされるようになります。

SystemC 型と Vivado HLS 型の比較

Vivado HLS 型は SystemC 型と同様で、実質上すべてのクラスに互換性があります。Vivado HLS 型を使用して記述されたコードは、通常 SystemC デザインに移行でき、SystemC 型を使用して記述されたコードは Vivado HLS に移行できます。

Vivado HLS 型と SystemC 型のビヘイビアーには、違いもあります。これらの違いについては、次のセクションに分けて説明します。

- デフォルト コンストラクター
- 整数除算
- 整数係数
- 負のシフト
- 左シフトのオーバーフロー
- `range` 演算
- 固定小数点の除算
- 固定小数点のライトシフト

- 固定小数点の左シフト

デフォルト コンストラクター

SystemC では、次の型のコンストラクターにより、プログラム実行前に値が 0 に初期化されます。

- `sc_[u]int`
- `sc_[u]bigen`
- `sc_[u]fixed`

次の Vivado HLS 型はこのコンストラクターでは初期化されません。

- `ap_[u]int`
- `ap_[u]fixed`

Vivado HLS のビット精度データ型:

- `ap_[u]int`
デフォルトの初期化なし
- `ap_[u]fixed`
デフォルトの初期化なし

SystemC のビット精度データ型:

- `sc_[u]int`
デフォルトで 0 に初期化
- `sc_big[u]int`
デフォルトで 0 に初期化
- `sc_[u]fixed`
デフォルトで 0 に初期化



注意: SystemC 型を Vivado HLS 型に移行する場合は、変数が読み出されないように、または書き込まれるまで条件文に使用されないようにしてください。

出力が書き込まれたかどうかに関係なく、SystemC デザインのすべての出力がデフォルト値の 0 で表示され始めることがあります。この場合、Vivado HLS 型と同じ変数はそれが書き込まれるまで未知の状態のままです。

整数除算

整数除算を使用する場合、Vivado HLS 型は `sc_big[u]int` 型と同じになりますが、`sc_[u]int` とは異なるビヘイビアになります。次の図に例を示します。

図 108: 整数除算の相違点

ap_(u)int	=	sc_big(u)int	≠	sc_(u)int
<pre>#include "ap_int.h" ap_uint<15>dividend = 32757; ap_int<15>divisor = -2; ap_int<21>ret = dividend/divisor;</pre>		<pre>#include "systemc.h" sc_biguint<15>dividend = 32757; sc_bigint<15>divisor = -2; sc_bigint<21>ret = dividend/divisor;</pre>		<pre>#include "systemc.h" sc_uint<15>dividend = 32757; sc_int<15>divisor = -2; sc_int<21>ret = dividend/divisor;</pre>
<div>dividend</div> <div>divisor</div> <div>ret</div> <div>7 F F F 32767</div> <div>7 F F E -2</div> <div>1 F C 0 0 1 -16383</div>		<div>dividend</div> <div>divisor</div> <div>ret</div> <div>7 F F F 32767</div> <div>7 F F E -2</div> <div>1 F C 0 0 1 -16383</div>		<div>dividend</div> <div>divisor</div> <div>ret</div> <div>7 F F F 32767</div> <div>7 F F E -2</div> <div>0 0 0 0 0 0 0</div>

X14224

SystemC の `sc_int` 型は、符号なしの整数が負の符号付き整数で除算されると 0 を返します。Vivado HLS 型の場合、SystemC の `sc_bigint` と同様、負の結果を示します。

整数係数

剰余演算子を使用する場合、Vivado HLS 型は `sc_big[u]int` 型と同じになりますが、`sc_[u]int` とは異なるビヘイビアになります。次の図に例を示します。

図 109: 整数剰余の相違点

ap_(u)int	=	sc_big(u)int	≠	sc_(u)int
<pre>#include "ap_int.h" ap_uint<15>dividend = 18; ap_int<15>divisor = -5; ap_int<21>ret = dividend%divisor;</pre>		<pre>#include "systemc.h" sc_biguint<15>dividend = 18; sc_bigint<15>divisor = -5; sc_bigint<21>ret = dividend%divisor;</pre>		<pre>#include "systemc.h" sc_uint<15>dividend = 18; sc_int<15>divisor = -5; sc_int<21>ret = dividend%divisor;</pre>
<div>dividend</div> <div>divisor</div> <div>ret</div> <div>0 0 1 2 18</div> <div>7 F F B -5</div> <div>0 0 0 0 0 3 3</div>		<div>dividend</div> <div>divisor</div> <div>ret</div> <div>0 0 1 2 18</div> <div>7 F F B -5</div> <div>0 0 0 0 0 3 3</div>		<div>dividend</div> <div>divisor</div> <div>ret</div> <div>0 0 1 2 18</div> <div>7 F F B -5</div> <div>0 0 0 0 1 2 18</div>

X14225

SystemC の `sc_int` 型は、次の場合に剰余演算の被除数の値を返します。

- 被除数 (dividend) が符号なしの整数。
- 除数 (divisor) が負の符号付き整数。

Vivado HLS 型の場合 (例: SystemC の `sc_bigint` 型)、正の剰余演算結果を返します。

負のシフト

シフト演算の値が負の数値の場合、Vivado HLS の `ap_[u]int` 型は値を逆方向にシフトします。たとえば、右シフトの場合は左シフトを返します。

この場合、SystemC 型の `sc_[u]int` と `sc_big[u]int` の動作は異なります。次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。

図 110: 負のシフトの相違点

ap_(u)int	≠	sc_big(u)int	≠	sc_(u)int
<pre>#include "ap_int.h" ap_uint<15>op = 24; ap_int<15>shift = -2; ap_int<21>ret = op>>shift;</pre>		<pre>#include "systemc.h" sc_biguint<15>op = 24; sc_bigint<15>shift = -2; sc_bigint<21>ret = op>>shift;</pre>		<pre>#include "systemc.h" sc_uint<15>op = 24; sc_int<15>shift = -2; sc_int<21>ret = op>>shift;</pre>
op 0 0 1 8 24 shift 7 F F E -2 ret 0 0 0 0 6 0 96		op 0 0 1 8 24 shift 7 F F E -2 ret 0 0 0 0 1 8 24		op 0 0 1 8 24 shift 7 F F E -2 ret 0 0 0 0 0 0 0

X14226

次の表に、負のシフトの相違点をまとめます。

表 82: 負のシフトの相違点のサマリ

型	動作
ap_[u]int	反対方向にシフト
sc_[u]int	0 を返す
sc_big[u]int	シフトなし

左シフトのオーバーフロー

シフト演算が実行され、結果が出力や代入された変数ではなく、入力変数からオーバーフローする場合、Vivado HLS 型と SystemC 型のビヘイビアは異なります。

- Vivado HLS の ap_[u]int は、値をシフトしてから代入をするので、上位ビットはオーバーフローして失われます。
- SystemC の sc_big(u)int および sc_(u)int 型の場合は、両方とも結果を代入してから、上位ビットを保持しつつシフトを実行します。
- 次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。

図 111: 左シフトのオーバーフローの相違点

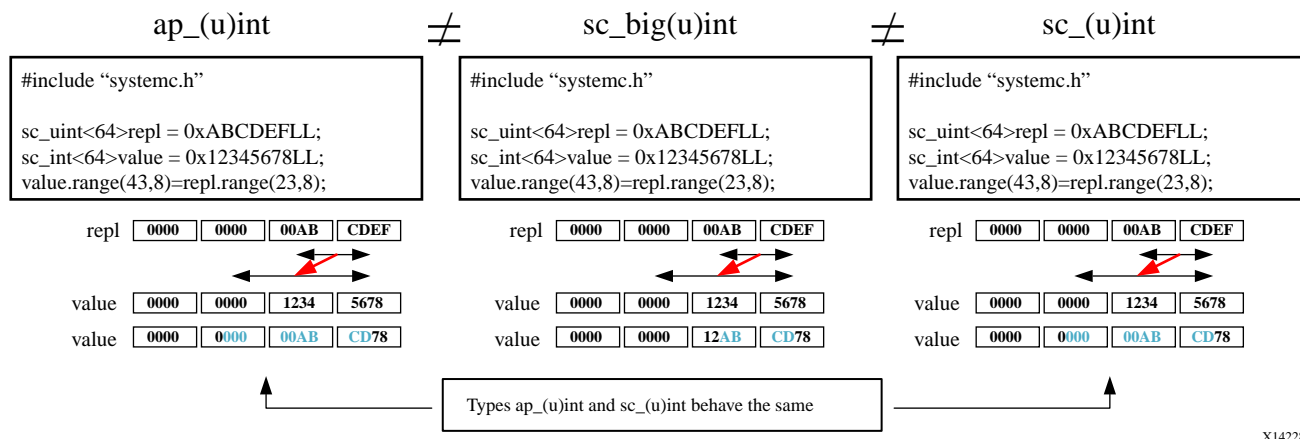
ap_(u)int	≠	sc_big(u)int	=	sc_(u)int
<pre>#include "ap_int.h" ap_uint<15>op = 0x7234; ap_int<15>shift = 4; ap_int<21>ret = op<<shift;</pre>		<pre>#include "systemc.h" sc_biguint<15>op = 0x7234; sc_bigint<15>shift = 4; sc_bigint<21>ret = op<<shift;</pre>		<pre>#include "systemc.h" sc_uint<15>op = 0x7234; sc_int<15>shift = 4; sc_int<21>ret = op<<shift;</pre>
op 7 2 3 4 29236 shift 0 0 0 4 4 ret 0 0 2 3 4 0 9024		op 7 2 3 4 29236 shift 0 0 0 4 4 ret 0 7 2 3 4 0 467776		op 7 2 3 4 29236 shift 0 0 0 4 4 ret 0 7 2 3 4 0 467776

X14227

range 演算

range 演算を使用したときに、ソースとデスティネーションの範囲のサイズが異なる場合は、動作が異なります。次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。次のサマリを参照してください。

図 112: range 演算の相違点



X14228

- Vivado HLS の `ap_[u]int` 型および SystemC の `sc_big[u]int` 型では、指定した範囲を置き換え、ターゲットの範囲を 0 で埋めるように拡張されます。
- SystemC の `sc_big[u]int` 型では、ソースの範囲のみを使用してアップデートされます。

固定小数点型の除算

異なるサイズの固定小数点型の変数を使用して除算を実行する場合、小数値の代入方法が Vivado HLS 型と SystemC 型で異なります。

`ap_[u]fixed` 型の場合、小数部分はその被除数よりも大きくはなりません。SystemC の `sc_[u]fixed` 型では、除算での小数の精度が保持されます。 `ap_[u]fixed` 型を使用する場合、代入前に新しい変数の幅に型変換することにより、小数部分を保持できます。

次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。

図 113: 固定小数点の除算の相違点

ap_(u)fixed	≠	sc_(u)fixed
<pre>#include "ap_fixed.h" ap_fixed<3,3> dividend=2; ap_fixed<4,4> divisor=4; ap_fixed<4,2> ret=dividend/divisor; //casting required to keep precision ap_fixed<4,2> ret2=ap_fixed<4,2>(dividend)/divisor;</pre>		<pre>#include "systemc.h" #define SC_INCLUDE_FX sc_fixed<3,3> dividend=2; sc_fixed<4,4> divisor=4; sc_fixed<4,2> ret=dividend/divisor;</pre>
<div>dividend</div> <div>0 1 0 0</div> <div>2.0</div> <div>divisor</div> <div>0 1 0 0</div> <div>4.0</div> <div>ret</div> <div>0 0 0 0 0 0</div> <div>0</div> <div>ret2</div> <div>0 0 0 0 1 0</div> <div>0.5</div>		<div>dividend</div> <div>0 1 0 0</div> <div>2.0</div> <div>divisor</div> <div>0 1 0 0</div> <div>4.0</div> <div>ret</div> <div>0 0 0 0 1 0</div> <div>0.5</div>

X14229

固定小数点型の右シフト

右シフト演算が実行される場合、Vivado HLS と SystemC の動作は異なります。

- Vivado HLS の固定小数点型で実行されると、シフトが実行されてから、値が代入されます。
- SystemC の固定小数点型の場合、値が代入されてから、シフトが実行されます。

結果がより多くの小数ビットを含む固定小数点型の場合、SystemC 型ではさらに追加で精度が保持されます。

次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。

図 114: 右シフトを使用した固定小数点型の違い

ap_(u)fixed	≠	sc_(u)fixed
<pre>#include "ap_fixed.h" ap_fixed<5,3,AP_RND,AP_SAT> val=3.75 ap_fixed<5,3,AP_RND,AP_SAT> res=val>>2; ap_fixed<7,3,AP_RND,AP_SAT> res2=val>>2;</pre>		<pre>#include "systemc.h" #define SC_INCLUDE_FX sc_fixed<5,3,AP_RND,AP_SAT> val=3.75 sc_fixed<5,3,AP_RND,AP_SAT> res=val>>2; sc_fixed<7,3,AP_RND,AP_SAT> res2=val>>2;</pre>
<div>val</div> <div>0 1 1 1 1</div> <div>3.75</div> <div>res</div> <div>0 0 0 1 1</div> <div>0.75</div> <div>res2</div> <div>0 0 0 1 1 0 0</div> <div>0.75</div>		<div>val</div> <div>0 1 1 1 1</div> <div>3.75</div> <div>res</div> <div>0 0 0 1 1</div> <div>0.75</div> <div>res2</div> <div>0 0 0 1 1 1 1</div> <div>0.9375</div>

X14230

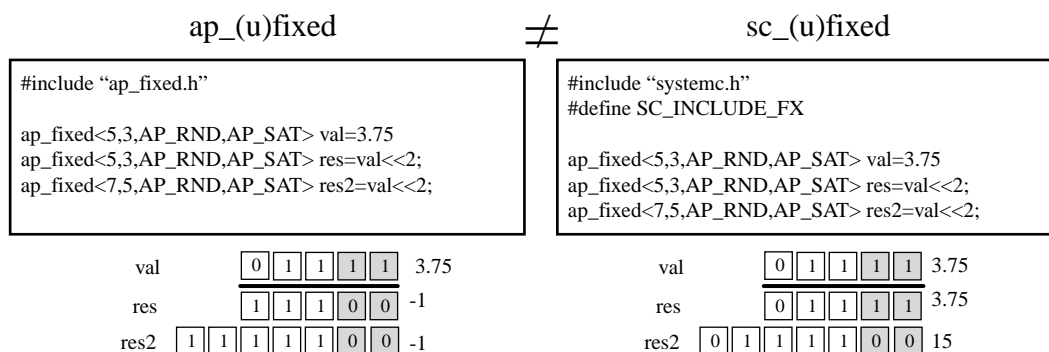
量子化モードの型は、ap_[u]fixed の右シフトの結果には影響しません。ザイリンクスでは、シフト演算よりも前に結果の型のサイズに代入しておくことを勧めしています。

固定小数点型の左シフト

ap_[u]fixed 型を使用して左シフト演算を実行すると、オペランドが符号拡張されてから、シフトされて、代入されます。SystemC の sc_[u]fixed 型の場合は、代入してからシフトが実行されます。この場合、Vivado HLS 型はすべての符号を保持します。

次の図に、Vivado HLS 型と SystemC 型でのこの演算の例を示します。

図 115: 左シフトを使用した固定小数点型の違い



X14231

RTL ブロックボックスの JSON ファイル

JSON ファイルの例

このセクションでは、RTL ブロック ボックスに必要な JSON ファイルを手動で書き出す方法について説明します。次は JSON ファイルの例です。

```
{
  "c_function_name" : "foo",
  "rtl_top_module_name" : "foo",
  "c_files" :
  [
    {
      "c_file" : "../..a/top.cpp",
      "cflag" : ""
    },
    {
      "c_file" : "xx.cpp",
      "cflag" : "-D KF"
    }
  ],
  "rtl_files" : [
    "../..foo.v",
    "xx.v"
  ],
  "c_parameters" : [{
    "c_name" : "a",
    "c_port_direction" : "in",
    "rtl_ports" : {
      "data_read_in" : "a"
    }
  },
  {
    "c_name" : "b",
    "c_port_direction" : "in",
    "rtl_ports" : {
```

```

        "data_read_in" : "b"
    }
},
{
    "c_name" : "c",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "c",
        "data_write_valid" : "c_ap_vld"
    }
},
{
    "c_name" : "d",
    "c_port_direction" : "inout",
    "rtl_ports" : {
        "data_read_in" : "d_i",
        "data_write_out" : "d_o",
        "data_write_valid" : "d_o_ap_vld"
    }
},
{
    "c_name" : "e",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "FIFO_empty_flag" : "e_empty_n",
        "FIFO_read_enable" : "e_read",
        "FIFO_data_read_in" : "e"
    }
},
{
    "c_name" : "f",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "FIFO_full_flag" : "f_full_n",
        "FIFO_write_enable" : "f_write",
        "FIFO_data_write_out" : "f"
    }
},
{
    "c_name" : "g",
    "c_port_direction" : "in",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : {
        "RAM_address" : "g_address0",
        "RAM_clock_enable" : "g_ce0",
        "RAM_data_read_in" : "g_q0"
    }
},
{
    "c_name" : "h",
    "c_port_direction" : "out",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : {
        "RAM_address" : "h_address0",
        "RAM_clock_enable" : "h_ce0",
        "RAM_write_enable" : "h_we0",
        "RAM_data_write_out" : "h_d0"
    }
},
{
    "c_name" : "i",
    "c_port_direction" : "inout",
    "RAM_type" : "RAM_1P",

```

```

        "rtl_ports" : {
            "RAM_address" : "i_address0",
            "RAM_clock_enable" : "i_ce0",
            "RAM_write_enable" : "i_we0",
            "RAM_data_write_out" : "i_d0",
            "RAM_data_read_in" : "i_q0"
        }
    },
    {
        "c_name" : "j",
        "c_port_direction" : "in",
        "RAM_type" : "RAM_T2P",
        "rtl_ports" : {
            "RAM_address" : "j_address0",
            "RAM_clock_enable" : "j_ce0",
            "RAM_data_read_in" : "j_q0",
            "RAM_address_snd" : "j_address1",
            "RAM_clock_enable_snd" : "j_ce1",
            "RAM_data_read_in_snd" : "j_q1"
        }
    },
    {
        "c_name" : "k",
        "c_port_direction" : "out",
        "RAM_type" : "RAM_T2P",
        "rtl_ports" : {
            "RAM_address" : "k_address0",
            "RAM_clock_enable" : "k_ce0",
            "RAM_write_enable" : "k_we0",
            "RAM_data_write_out" : "k_d0",
            "RAM_address_snd" : "k_address1",
            "RAM_clock_enable_snd" : "k_ce1",
            "RAM_write_enable_snd" : "k_we1",
            "RAM_data_write_out_snd" : "k_d1"
        }
    },
    {
        "c_name" : "l",
        "c_port_direction" : "inout",
        "RAM_type" : "RAM_T2P",
        "rtl_ports" : {
            "RAM_address" : "l_address0",
            "RAM_clock_enable" : "l_ce0",
            "RAM_write_enable" : "l_we0",
            "RAM_data_write_out" : "l_d0",
            "RAM_data_read_in" : "l_q0",
            "RAM_address_snd" : "l_address1",
            "RAM_clock_enable_snd" : "l_ce1",
            "RAM_write_enable_snd" : "l_we1",
            "RAM_data_write_out_snd" : "l_d1",
            "RAM_data_read_in_snd" : "l_q1"
        }
    }
}],
"c_return" : {
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "ap_return"
    }
},
"rtl_common_signal" : {
    "module_clock" : "ap_clk",
    "module_reset" : "ap_rst",
    "module_clock_enable" : "ap_ce",

```

```

        "ap_ctrl_chain_protocol_idle" : "ap_idle",
        "ap_ctrl_chain_protocol_start" : "ap_start",
        "ap_ctrl_chain_protocol_ready" : "ap_ready",
        "ap_ctrl_chain_protocol_done" : "ap_done",
        "ap_ctrl_chain_protocol_continue" : "ap_continue"
    },
    "rtl_performance" : {
        "latency" : "6",
        "II" : "2"
    },
    "rtl_resource_usage" : {
        "FF" : "0",
        "LUT" : "0",
        "BRAM" : "0",
        "URAM" : "0",
        "DSP" : "0"
    }
}

```

その他のリソースおよび法的通知

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、[ザイリンクス サポート](#) サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [Start] → [All Programs] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hubs View] タブをクリックします。
- ザイリンクス ウェブサイトで[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

このガイドの補足情報は、次の資料を参照してください。

日本語版のバージョンは、英語版より古い場合があります。

1. 『Vivado 高位合成を使用した FPGA デザインの概要』 ([UG998](#))
2. 『Vivado Design Suite チュートリアル: 高位合成』 ([UG871](#))

3. 『Vivado Design Suite ユーザー ガイド: リリース ノート、インストール、およびライセンス』 (UG973)
4. 『Vivado HLS を使用した浮動小数点デザイン』 (XAPP599)
5. 『LogiCORE IP Fast Fourier Transform 製品ガイド』 (PG109)
6. 『LogiCORE IP FIR Compiler 製品ガイド』 (PG149)
7. 『LogiCORE IP DDS Compiler 製品ガイド』 (PG141)
8. 『Vivado Design Suite: AXI リファレンス ガイド』 (UG1037)
9. 『Accelerating OpenCV Applications with Zynq-7000 SoC Using Vivado HLS Video Libraries』 (XAPP1167)
10. 『UltraFast 高生産性設計手法ガイド』 (UG1197)
11. GCC ウェブサイトのオプション サマリ ページ (gcc.gnu.org/onlinedocs/gcc/Option-Summary.html)
12. Accellera ウェブサイト (<http://www.accellera.org/>)
13. MathWorks ウェブサイトの AWGN ページ (<http://www.mathworks.com/help/comm/ug/awgn-channel.html>)
14. Vivado® Design Suite 資料

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとし、また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うこととなります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ 設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとし、セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとし、また、

商標

© Copyright 2012-2021 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリックス社の商標です。OpenCL および OpenCL のロゴは Apple Inc. の商標であり、Khronos による許可を受けて使用されています。PCI、PCIe、および PCI Express は PCI-SIG の商標であり、ライセンスに基づいて使用されています。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、および MPCore は、EU およびその他の各国の Arm Limited の商標です。すべてのその他の商標は、それぞれの所有者に帰属します。

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、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。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。