SDSoC 環境

ユーザー ガイド

UG1027 (v2015.4) 2015 年 12 月 14 日

本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。

XILINX
ALL PROGRAMMABLE™

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	改訂内容
2015 年 12 月 14 日	2015.4	ソフトウェアの変更を反映するようアップデート
2015 年 9 月 30 日	2015.2.1	· 「AXI Performance Monitor を使用したパフォーマンス計測」の章を追加 · ソフトウェアの変更を反映するようアップデート
2015 年 7 月 20 日	2015.2	初版

目次

XILINX
ALL PROGRAMMABLE™

第 1 章

SDSoC 環境

SDSoC™ (Software-Defined Development Environment for System-on-Chip) 環境は、Eclipse ベースの統合設計環境 (IDE) で Zynq-7000 All Programmable SoC プラットフォームを使用してヘテロジニアス エンベデッド システムをインプリメントし、システム コンパイラで指定の関数をプログラマブル ロジックにコンパイルしながら C/C++ プログラムを完全なハードウェア/ソフトウェア システムに変換するツール スイートです。

SDSoC のシステム コンパイラではプログラムが解析され、ソフトウェア関数とハードウェア関数間のデータ フローが決定され、アプリケーション特定のシステム オン チップが生成されてプログラムが実現されます。高パフォーマンスを達成するには、各ハードウェア関数が独立したスレッドとして実行される必要があります。SDSoC システム コンパイラでは、プログラム セマンティクスを保持してハードウェアとソフトウェア スレッド間が同期されるようにするハードウェアおよびソフトウェアのコンポーネントが生成され、パイプライン計算および通信もイネーブルにできます。アプリケーション コードには、多数のハードウェア関数、特定のハードウェア関数の複数のインスタンス、およびプログラムの異なる部分からのハードウェア関数の呼び出しなどを含めることができます。

SDSoC IDE では、プロファイル、コンパイル、リンク、およびデバッグを含む ソフトウェア開発ワークフローがサポートされています。また、SDSoC 環境では、完全なハードウェア コンパイルを実行する前に、ハードウェア/ソフトウェア インターフェイスで what-if シナリオを試すことが可能なパフォーマンス予測機能が提供されています。

SDSoC システム コンパイラは、ベース プラットフォームをターゲットにし、Vivado® 高位合成 (HLS) ツールを起動して合成可能な C/C++ 関数をプログラマブル ロジックにコンパイルします。その後 Vivado Design Suite ツールを起動して、DMA、インターコネクト、ハードウェア バッファー、その他の IP、および FPGA ビットストリームを含むハードウェア システムを生成します。すべてのハードウェア関数呼び出しで元のビヘイビアーが保持されるようにするため、SDSoC システムのコンパイラはシステム特定のソフトウェア スタブおよびコンフィギュレーション データを生成します。このプログラムには、生成された IP ブロックを使用するのに必要なドライバーへの関数呼び出しが含まれています。アプリケーションおよび生成されたソフトウェアは、標準の GNU ツールチェーンを使用してコンパイルおよびリンクされます。

システム コンパイラでは、1 つのソースから完全なアプリケーションを生成することにより、プログラム レベルでデザインやアーキテクチャを繰り返し変更できるので、ターゲット プラットフォームで実行可能なプログラムを得るまでに要する時間を大幅に短縮できます。

入門

SDSoC™ 環境のダウンロードおよびインストール方法については、『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) を参照してください。このガイドには、プロジェクト作成、プログラマブル ロジックで実行する関数の指定、システム コンパイル、デバッグ、およびパフォーマンス予測における主要なワークフローが詳細な説明と体験型チュートリアルの形式で提供されています。これらのチュートリアルを体験するのが SDSoC 環境の概要を理解するのに最適な方法なので、アプリケーション開発前に実行しておくことをお勧めします。

次の点に注意してください。

- SDSoC システム コンパイラをコマンド ラインまたは makefile フローから実行する場合は、『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) に記述されているとおりにシェル環境を正しく設定しないとツールが正しく動作しません。
- SDSoC 環境にはビットストリーム、オブジェクト コード、および実行ファイルを作成するためのツールがすべて含まれています。ザイリンクス Vivado Design Suite および SDK (ソフトウェア開発キット) ツールを個別にインストールしている場合は、これらのインストールを SDSoC 環境と統合しないでください。

機能の概要

SDSoC™ 環境は、Zynq® デバイス内の ARM CPU の GNU ツールチェーンおよび標準ライブラリ (例 : glibc、OpenCV)、および Target Communication Framework (TCF) と GDB インタラクティブ デバッガー、Eclipse/CDT ベースの GUI 内のパフォーマンス解析パースペクティブ、コマンド ライン ツールなど、ザイリンクス ソフトウェア開発キット (SDK) の多数のツールを継承しています。

SDSoC 環境には、Zynq デバイスをターゲットとする完全なハードウェア/ソフトウェア システムを生成するシステム コンパイラ (`sdscc/sds++`)、プロジェクトとワークフローを作成して管理する Eclipse ベースのユーザー インターフェイス、ハードウェア/ソフトウェア インターフェイスのさまざまな what-if シナリオを試すためのシステム パフォーマンス予測機能などが含まれます。

SDSoC システム コンパイラでは、Vivado® HLS、IP インテグレーター (IPI)、データ移動およびインターコネクト用の IP ライブラリ、および RTL 合成、配置、配線、ビットストリーム生成ツールを含む Vivado Design Suite (System Edition) からのツールが使用されます。

SDSoC 環境で使用されるワークフローは、確立されたプラットフォーム ベースの設計手法を使用した、デザイン再利用の原則が基礎となっています。SDSoC システム コンパイラは、ターゲット プラットフォームを拡張することにより、アプリケーション特定のシステム オン チップを生成します。SDSoC 環境には、アプリケーション開発用の多数のプラットフォームおよびザイリンクス パートナーから提供されるプラットフォームが含まれています。『SDSoC 環境ユーザー ガイド : プラットフォームおよびライブラリ』(UG1146) に、Vivado Design Suite を使用した既存のデザイン ビルドおよび対応するソフトウェア ランタイム環境が SDSoC プラットフォームのビルドに使用されて SDSoC 環境で使用されるようにするために、プラットフォーム メタデータをキャプチャする方法が説明されています。

SDSoC プラットフォームは、ベース ハードウェアおよびソフトウェア アーキテクチャと、プロセッシング システム、外部メモリ インターフェイス、カスタム入力/出力、およびオペレーティング システム (ベアメタルの場合もあり)、ブートローダー、プラットフォーム ペリフェラルやルート ファイル システムなどのドライバー システム ランタイムを含むアプリケーション コンテキストを定義します。SDSoC 環境で作成するプロジェクトはすべて特定のプラットフォームをターゲットとし、SDSoC IDE に含まれるツールを使用して、そのプラットフォームをアプリケーション特定のハードウェア アクセラレータおよびアクセラレータをプラットフォームに接続するデータ モーション ネットワークでカスタマイズします。この方法を使用すると、さまざまなベース プラットフォーム向けに高度にカスタマイズされたアプリケーション特定のシステム オン チップを簡単に作成でき、ベース プラットフォームをさまざまなアプリケーション特定のシステム オン チップに再利用できます。

XILINX
ALL PROGRAMMABLE™

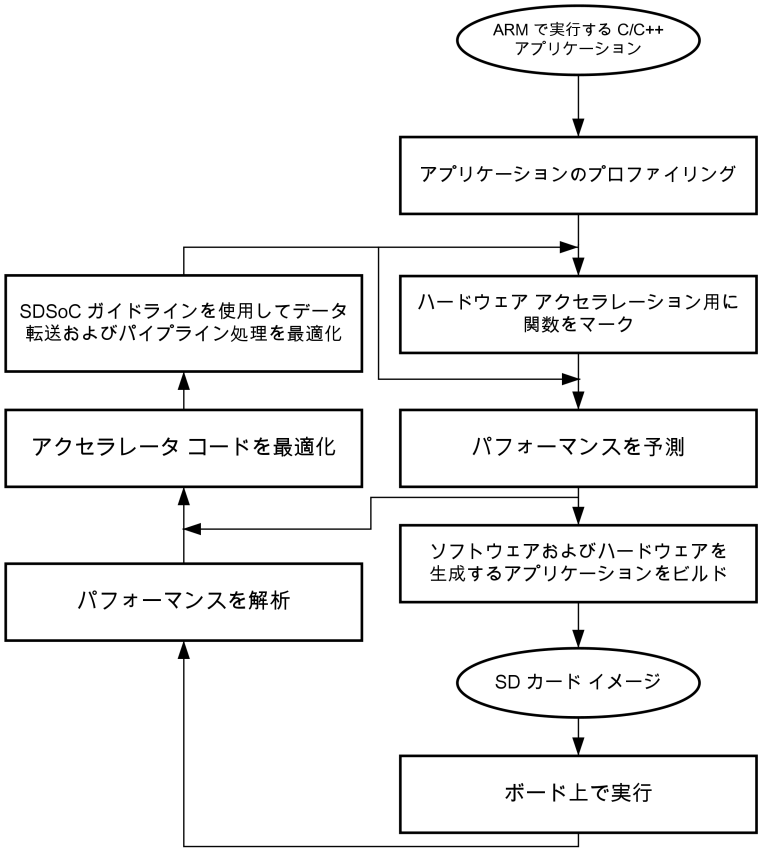
第 2 章

ユーザー デザイン フロー

SDSoC 環境は、ベース ハードウェアおよびブート オプションを含むターゲット ソフトウェア アーキテクチャを提供するプラットフォーム SoC から開始して、アプリケーション特定の効率的なシステム オン チップをビルドするためのツール スイートです。

次の図は、このツール スイートの主なコンポーネントを使用したデザイン フローです。説明のため、ここではデザイン フローは 1 つの手順から次の手順に順に進められますが、実際には異なるエントリ ポイントおよびエグジット ポイントを使用したほかのワークフローを選択することもできます。まず、ARM CPU 用にクロスコンパイルされたアプリケーションのソフトウェアのみのバージョンから開始します。主な目標は、プログラマブル ロジックに移行するプログラム部分を特定し、ベース プラットフォームにビルドされたハードウェアおよびソフトウェアにアプリケーションをインプリメントすることです。

図 2-1 : ユーザー デザイン フロー



X14740-070215

最初の手順では、開発プラットフォームを選択し、アプリケーションをクロスコンパイルして、プラットフォームで正しく実行されるかどうか確認します。次に、システム パフォーマンスを改善するため、プログラマブル ロジックに移行する計算負荷の高いホット スポットを特定し、ハードウェアにコンパイル可能な関数に分離します。この後、SDSoC システム コンパイラを起動して、アプリケーション用のシステム オン チップと SD カードを生成します。コードには必要であればパフォーマンスを解析するコードを含めて、SDSoC 環境内で指示子とツールのセットを使用して、システムおよびハードウェア関数を最適化できます。

システム生成プロセスは、SDSoC IDE を使用するか、SDSoC ターミナル シェルでコマンド ラインと makefile を使用して、sdscc/sds++ システム コンパイラで実行します。SDSoC IDE または sdscc コマンド ライン オプションを使用して、ハードウェアで実行する関数を選択、アクセラレータおよびシステム クロックを指定、およびデータ転送のプロパティ (DMA 転送の割り込みまたはポーリングなど) を設定します。プラグマをアプリケーション ソース コードに挿入してアクセラレータおよびデータ モーション ネットワークをインプリメントするための指示子をシステム コンパイラに供給することにより、システム マップおよび生成フローを制御できます。

完全なシステムのコンパイルには、CPU 用にオブジェクト コードをコンパイルするよりも時間がかかるので、SDSoC 環境には選択したハードウェア関数に対してソフトウェアのみのインプリメンテーションと比較したスピードアップを大まかに予測する機能があります。この予測は、生成されたシステムの特性および IP で提供されるハードウェア関数の予測に基づきます。

「ユーザー デザイン フロー」に示すように、デザイン プロセスは生成されたシステムがパフォーマンスおよびコストの目標を達成するまで繰り返し実行されます。

入門チュートリアル (『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) を参照) を実行済みで、プロジェクトの作成、ハードウェア関数の選択、コンパイル、プラットフォームでのアプリケーションの実行について理解していることを想定しています。そうでない場合は、続行する前にチュートリアルを実行することをお勧めします。

ターゲット プラットフォームのプロジェクト作成

SDSoC IDE で [File] → [New] → [SDSoC Project] をクリックして New Project ウィザードを開きます。プロジェクト名を入力して、[Platform] プルダウン メニューから開発用のプラットフォームを選択します。プラットフォームには、基本ハードウェア システム、OS を含むソフトウェア ランタイム、ブートローダー、およびルート ファイル システムが含まれています。SDSoC 環境のプロジェクトでは、プラットフォームが固定され、コマンド ライン オプションが自動的にすべての makefile に含まれます。プロジェクトのターゲットを新しいプラットフォームに変更するには、新しいプラットフォームを使用して新しいプロジェクトを作成し、作業中のプロジェクトから新しいプロジェクトにソース ファイルをコピーする必要があります。

makefile を SDSoC IDE 外で記述する場合は、sdscc の呼び出しごとに -sds-pf コマンド ライン オプションを含める必要があります。

```
sdscc -sds-pf <platform path name>
```

<platform path name> には、ファイル パスまたは <sdsoc_root>/platforms ディレクトリに含まれるプラットフォーム名を指定します。使用可能なベース プラットフォームをコマンド ラインで表示するには、次のコマンドを入力します。

```
sdscc -sds-pf-list
```

ベース プラットフォーム以外のサンプル プラットフォームは <sds_root>/samples/platforms ディレクトリに含まれています。SDSoC IDE でこれらのプラットフォームのいずれかを使用してプロジェクトを作成する場合は、[Other] をクリックし、サンプル プラットフォームを指定します。

データ モーション ネットワーク クロック

各プラットフォームでは 1 つまたは複数のクロック ソースがサポートされており、明示的に選択しない場合はデフォルトでその 1 つが使用されます。デフォルトのクロックはプラットフォームのプロバイダーにより定義されており、sdscc で生成されるデータ モーション ネットワークで使用されます。プラットフォームのクロックを表示するには、[SDSoC Project Overview] の [General] パネルで [Platform] をクリックします。別のクロック周波数を選択するには、[SDSoC Project Overview] の [Options] パネルで [Data Motion Network Clock Frequency] プルダウン メニューから選択するか、またはコマンド ラインで -dmclockid を使用してクロック ID を指定します。

```
sdscc -sds-pf zc702 -dmclockid 1
```

コマンド ラインでプラットフォームで使用可能なクロックを表示するには、次を実行します。

```
$ sdscc -sds-pf-info zc702
Platform Description
====================
Basic platform targeting the ZC702 board, which includes 1GB of DDR3, 16MB Quad-
SPI Flash and an SDIO card interface. More information at http://www.xilinx.com/
products/boards-and-kits/EK-Z7-ZC702-G.htm
Platform Information
====================
Name: zc702
Device
------
Architecture: zynq
 Device: xc7z020
  Package: clg484
  Speed grade: -1
 System Clocks
 -------------
  Clock ID Frequency
  ----------|------------
  666.666687
  0 166.666672
  1 142.857132
  2 100.000000
  3 200.000000
```

ARM プロセッサでのアプリケーションのコンパイルと実行

アプリケーション開発の最初の段階では、アプリケーション コードをクロスコンパイルして、ターゲット プラットフォームで実行します。SDSoC 環境に含まれる各プラットフォームにはビルド済みの SD カード イメージが含まれ、この SD カード イメージから、クロス コンパイルされたアプリケーションをブートおよび実行できます。プロジェクトでハードウェアに関数を選択しない場合は、このビルド済みイメージが使用されます。

コードを変更した場合は (ハードウェア関数への変更を含む)、ソフトウェアのみのコンパイルを実行し直して、変更によってプログラムに悪影響が出ないかどうかを確認すると有益です。ソフトウェアのみのコンパイルは、フル システム コンパイルよりもかなり高速で、ソフトウェアのみのデバッグを使用すると、ハードウェア/ソフトウェア デバッグよりも論理プログラム エラーをすばやく見つけることができます。

ザイリンクス SDK と同様、SDSoC 環境には Zynq® アーキテクチャ デバイス内の ARM CPU 用に 2 つの異なるツールチェーンが含まれます。

1. `arm-xilinx-linux-eabi` : Linux アプリケーション開発用
2. `arm-xilinx-gnueabi` : スタンドアロン (ベアメタル) および FreeRTOS アプリケーション用

GNU ツールチェーンは、プロジェクト作成中にオペレーティング システムを選択すると定義されます。SDSoC システム コンパイラ (`sdscc`/`sds++`) は、ハードウェア関数に関係しないすべてのソース ファイルを含む CPU のコードをコンパイルする際に対応するツールチェーンが自動的に起動します。

ARM CPU のオブジェクト コードはすべて GNU ツールチェーンを使用して生成されますが、`sdscc` (および `sds++`) コンパイラは Clang/LLVM フレームワークに基づいてビルドされているので、通常 GNU コンパイラよりも C/C++ 言語違反に対する寛容性が低くなっています。この結果、`sdscc` を使用すると、アプリケーションに必要なライブラリの一部によりフロントエンド コンパイラ エラーが発生することがあります。この場合、ソース ファイルをコンパイルするのに `sdscc` ではなく GNU ツールチェーンを直接使用してください。これには、makefile に入力するか、[Project Explorer] タブでファイル (またはフォルダー) を右クリックして、[C/C++ Build → Settings] → [SDSCC/SDS++ Compiler] をクリックし、`[Command]` に `GCC` または `g++` を入力します。

SDSoC システム コンパイラでは、デフォルトで `sd_card` というプロジェクト サブディレクトリに SD カード イメージが生成されます。Linux アプリケーションの場合、このディレクトリには次のファイルが含まれます。

- `README.TXT` : アプリケーションの実行方法の簡単な説明
- `BOOT.BIN` : FSBL (First Stage Boot Loader)、ブート プログラム (U-Boot) および FPGA ビットストリームを含むブート イメージ
- `uImage`、`devicetree.dtb`、`uramdisk.image.gz` : Linux ブート イメージ
- `<app>.elf` : アプリケーション バイナリ実行ファイル

アプリケーションを実行するには、`sd_card` ディレクトリの内容を SD カードにコピーし、ターゲット ボードに挿入します。ターゲットに対するシリアル ターミナル接続を開いて、ボードに電源を投入します (詳細は『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) を参照)。Linux が起動され、ルートとして自動的にログインされ、Bash シェルが表示されます。SD カードは `/mnt` に割り当てられます。このディレクトリから、`<app>.elf` を実行できます。

スタンドアロン アプリケーションの場合、ELF、ビットストリーム、ボード サポート パッケージ (BSP) が `BOOT.BIN` に含まれ、システム ブート後にアプリケーションが自動的に実行されます。

パフォーマンスを計測するためのコードのプロファイリングおよび計測用関数呼び出しの追加

ソフトウェア定義の SoC を作成する際の最初の主なタスクは、ハードウェアで実行した場合に全体的なパフォーマンスが大幅に向上する、ハードウェアにインプリメントするのに適したアプリケーション コードの部分を特定することです。計算集約的なプログラム ホット スポットはハードウェア アクセラレーションに適しており、ハードウェアと CPU およびメモリ間でデータをストリーミングして計算と通信をオーバーラップさせることができる場合は特に良い候補であると言えます。ソフトウェア プロファイリングは、プログラムの CPU 集約型の部分を特定する標準的な方法です。

SDSoC 環境には、`gprof`、非介入 Target Communication Framework (TCF) プロファイラー、Eclipse のパフォーマンス解析パースペクティブなどのザイリンクス SDK に含まれるパフォーマンスおよびプロファイリング機能がすべて含まれています。

スタンドアロン アプリケーションに対して TCF プロファイラーを実行するには、次の手順に従います。

1. アクティブ ビルド コンフィギュレーションを [SDDebug] に変更します。
2. [SDSoC Project Overview] で [Debug Application] をクリックします。ボードをコンピューターに接続して電源をオンにしておく必要があります。アプリケーションが `main()` に入る地点でブレークします。
3. [Window] → [Show View] → [Other] → [Debug] → [TCF Profiler] をクリックして TCF プロファイラーを起動します。
4. [TCF Profiler] タブの上部にある緑色の [Start] ボタンをクリックして TCF プロファイラーを開始します。[Profiler Configuration] ダイアログ ボックスで [Aggregate per function] をイネーブルにします。
5. [Resume] ボタンをクリックしてプロファイリングを開始します。プログラムが実行を完了し、`exit()` 関数でブレークします。
6. [TCF Profiler] タブで結果を確認します。

プロファイリングは、CPU プログラム カウンターのサンプリングおよび実行中のプログラムへの相関に基づいてホット スポットを検索するための統計的な手法です。プログラムのパフォーマンスを計測するもう 1 つの方法として、実行中のプログラムの異なる部分に実際にかかる時間を判断するため、アプリケーションに計測用の関数呼び出しを追加する方法があります。

SDSoC 環境の `sds_lib` ライブラリには、アプリケーション パフォーマンスの測定に使用可能な単純なソース コード アノテーション ベースのタイム スタンプ API が含まれます。

```
/*
 * @return value of free-running 64-bit Zynq(TM) global counter
 */
unsigned long long sds_clock_counter(void);
```

この API を使用してタイムスタンプを収集してそれらの差を調べることにより、プログラムの主要な部分の時間を判断できます。たとえば、次のコード例に示すように、データ転送やハードウェア関数の全体的な実行時間を計測できます。

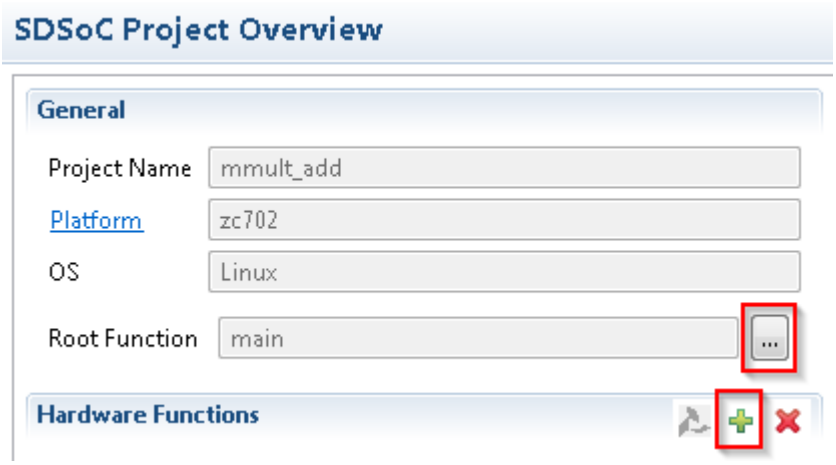
```
#include "sds_lib.h"
unsigned long long total_run_time = 0;
unsigned int num_calls = 0;
unsigned long long count_val = 0;
#define sds_clk_start(){ \
    count_val = sds_clock_counter(); \
    num_calls++; \
}
#define sds_clk_stop() { \
    long long tmp = sds_clock_counter(); \
    total_run_time += (tmp - count_val); \
}
#define avg_cpu_cycles()(total_run_time / num_calls)
#define NUM_TESTS 1024
extern void f();
void measure_f_runtime()
{
    for (int i = 0; i < NUM_TESTS; i++) {
        sds_clock_start();
        f();
        sds_clock_stop();
    }
    printf("Average cpu cycles f(): %ld\n", avg_cpu_cycles());
}
```

SDSoC 環境内のパフォーマンス予測機能はこの API を使用し、ハードウェア インプリメンテーションに選択された関数に計測用の関数呼び出しを自動的に追加し、ターゲット上でアプリケーションを実行して実際の実行時間を計測して、ハードウェア関数に予測される実行時間と比較します。

注記 : CPU 集約型の関数をプログラマブル ロジックに移行することはアプリケーションの分割に最も信頼性の高い経験則ですが、システム パフォーマンスを向上するにはアルゴリズムを変更してメモリ アクセスを最適化することが必要な場合があります。CPU の外部メモリへのランダム アクセス速度は、マルチレベル キャッシュおよび高速クロック (通常プログラマブル ロジックよりも 2 ～ 8 倍高速) により、プログラマブル ロジックで達成できる速度よりもかなり速くなります。大型のインデックス セットのインデックスを並べ替えるソート ルーチンなどの広いアドレス範囲に対するポインター変数の処理は CPU には適していますが、関数をプログラマブル ロジックに移動するとマイナスになることがあります。これはそのような計算関数がハードウェアに移動する良い候補ではないということではなく、コードまたはアルゴリズムを再構成することが必要な場合があるということです。これは、DSP および GPU コプロセッサでの既知の問題です。

関数のプログラマブル ロジックへの移動

新しいプロジェクトを作成したら [Project Explorer] に含まれている `project.sdsoc` をダブルクリックして [SDSoC Project Overview] を開くことができます。



[Hardware Functions] パネルの をクリックしてプログラムに含まれている関数のリストを表示します。このリストには、[General] パネルの [Root Function] 示されるルート関数に属するコールグラフに含まれている関数が表示されます。デフォルトでは main が選択されていますが、[...] をクリックすると別の関数ルートを選択できます。

ダイアログ ボックスでハードウェア アクセラレーション用の関数を選択して [OK] をクリックします。選択した関数がリスト ボックスに表示されます。Eclipse CDT のインデックス メカニズムは機能しない場合があるため、選択可能な関数を表示するのに選択ダイアログ ボックスを一度閉じて開き直す必要がある場合があります。関数がリストに表示されない場合は、[Project Explorer] でその関数が含まれているファイルに移動して展開表示し、関数のプロトタイプを右クリックして [Toggle HW/SW] をクリックします。

コマンド ラインでは、次の sdscc コマンド ライン オプションを使用して、ハードウェア用に foo_src.c ファイルに含まれる関数 foo を選択します。

```
-sds-hw foo foo_src.c -sds-end
```

foo で foo_sub0.c および foo_sub1.c ファイルに含まれるサブ関数が呼び出される場合は、-files オプションを使用します。

```
-sds-hw foo foo_src.c -files foo_sub0.c,foo_sub1.c -sds-end
```

データ モーション ネットワークは 1 つのクロックで動作しますが、より高いパフォーマンスを達成するために、ハードウェア関数を異なるクロック レートで実行できます。[Hardware Functions] パネルでリストから関数を選択し、[Clock Frequency] プルダウン メニューからクロック周波数を選択します。クロックの一部はハードウェア システムでインプリメントできない可能性があるので注意してください。

コマンド ラインでクロックを設定するには、sdscc -sds-pf-info <platform> を使用して対応するクロック ID を確認し、-clockid オプションを使用します。

```
-sds-hw foo foo_src.c -clockid 1 -sds-end
```

CPU で実行するために最適化されている関数をプログラマブル ロジックに移動するときは、高いパフォーマンスを達成するために通常コードを変更する必要があります。プログラムのガイドラインは、「プログラマ向け Vivado 高位合成ガイド」および「コード ガイドライン」を参照してください。

SDSCC/SDS++ パフォーマンス予測フロー オプション

完全なビットストリームのコンパイルには、ソフトウェアのコンパイルよりもかなり時間がかかることがあります。そのため、sdscc には、パフォーマンス予測オプションを使用して、ハードウェア関数呼び出しセットのランタイム改善を予測する機能が含まれています。[SDSoC Project Overview] に含まれている [Estimate Performance Speedup for HW Functions] をクリックすると、プロジェクトが SDEstimate ビルド コンフィギュレーションに切り替わります。

スピードアップの予測は、2 段階のプロセスです。まず、SDSoC IDE でハードウェア関数をコンパイルしてシステムを生成します。システムをビットストリームに合成する代わりに、sdscc でハードウェア関数の予測レイテンシとハードウェア関数呼び出し元の予測データ転送時間に基づいて、パフォーマンスが予測されます。生成されたパフォーマンス レポートの [Click Here] をクリックして計測用の関数呼び出しが追加されたソフトウェアをターゲット上で実行して、パフォーマンスのベースラインとパフォーマンス予測を取得します (詳細は『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) を参照)。

コマンド ラインからパフォーマンス予測を生成することもできます。ソフトウェア ランタイムに関するデータを収集するため、まず -perf-funcs オプションを使用してプロファイルする関数を指定し、-perf-root オプションを使用してプロファイルされる関数への呼び出しを含むルート関数を指定します。これで、sdscc コンパイラによりアプリケーションに関数呼び出しが追加され、ボードでアプリケーションを実行したときに自動的にランタイム データが収集されます。この関数呼び出しが追加されたアプリケーションをターゲットで実行すると、プログラムにより SD カードに sw_perf_data.xml というファイルが作成されます。このファイルには、その実行のランタイム パフォーマンス データが含まれます。

swdata.xml をホストにコピーし、ハードウェア関数呼び出しごとおよび -perf-root で指定した最上位関数でのパフォーマンス向上を予測するビルドを実行します。-perf-est オプションを使用して、swdata.xml をこのビルドの入力データとして指定します。

次の表に、アプリケーションをビルドするのに通常使用される sdscc オプションを示します。

オプション	説明
-perf-funcs function_name_list	計測されるソフトウェア アプリケーションでプロファイリングするすべての関数をカンマで区切って指定します。
-perf-root function_name	プロファイリングされる関数へのすべての呼び出しを含むルート関数を指定します。デフォルトは、関数 main です。
-perf-est data_file	計測されるソフトウェア アプリケーションをターゲット上で実行したときに生成されたランタイム データを含むファイルを指定します。ハードウェア アクセラレーションされた関数のパフォーマンス向上を予測します。このファイルのデフォルト名は、swdata.xml です。
-perf-est-hw-only	ソフトウェア実行データを収集せずに予測フローを実行します。このオプションを使用すると、ベースラインとの比較は含まれませんが、ハードウェア レイテンシおよびリソース使用率を確認できます。

注意 : プロファイル データを収集するためにボードの sd_card イメージを実行したら、cd /; sync; umount /mnt; を実行し、swdata.xml ファイルが SD カードに書き込まれるようにします。

パフォーマンス予測のための makefile ベースのフローの例は、<sdsoc_root>/samples/mmult_performance_estimation に含まれています。

XILINX
ALL PROGRAMMABLE™

第 3 章

SDSoC 環境でのトラブルシューティング

SDSoC™ 環境を使用する際に発生する主な問題には、次の 3 つがあります。

- ソフトウェア コンパイラで検出される典型的なソフトウェア構文エラー、またはデザインが大きすぎてターゲット プラットフォームにフィットしないなどの SDSoC 環境フロー特有のエラーにより、コンパイル/リンク時エラーが発生する可能性があります。
- ヌル ポインター アクセスなどの一般的なソフトウェアの問題、またはアクセラレータとの誤データの送受信などの SDSoC 環境特有の問題が原因で、ライタイム エラーが発生する可能性があります。
- アクセラレーションに使用したアルゴリズムの選択、アクセラレータとのデータの送受信にかかる時間、およびアクセラレータとデータ モーション ネットワークの実動作速度などが原因で、パフォーマンスの問題が発生する可能性があります。

コンパイルおよびリンク時エラーのトラブルシューティング

通常、コンパイル/リンク時エラーは、make を実行した場合にエラー メッセージで示されます。さらにプローブするには、SDSoC™ 環境で作成されたビルド ディレクトリの _sds/reports ディレクトリからログ ファイルと rpt ファイルを確認します。最後に生成されたログ ファイルには、該当する入力ファイルの構文エラーや、アクセラレータ ハードウェアまたはデータ モーション ネットワークの合成中にツールチェーンで生成されたエラーなど、通常エラーの原因が示されます。

次に SDSoC 環境特有のエラーに対処する方法のヒントを示します。

- SDSoC 開発チェーンのツールでレポートされたツール エラー
  - 該当するコードが「コード ガイドライン」に従っているかどうかを確認します。
  - プラグマの構文を確認します。
  - プラグマにスペルミスがないかどうか確認します。これが原因で正しい関数に適用されていない可能性があります。
- Vivado Design Suite 高位合成 (HLS) でタイミング要件を満たすことができない
  - SDSoC IDE (または `sdscc/sds++` コマンド ライン パラメーター) でアクセラレータに対して低速のクロック周波数を選択します。
  - HLS で高速のインプリメンテーションを生成できるようにコード構造を変更します。詳細は、「プログラマ向け Vivado 高位合成ガイド」を参照してください。
- Vivado ツールでタイミングを満たすことができない
  - SDSoC IDE でデータ モーション ネットワークまたはアクセラレータ (あるいはその両方) に対してより低速のクロック周波数を選択します (コマンド ラインからの場合は `sdscc/sds++` コマンド ライン パラメーターを使用)。
  - HLS ブロックをより高速なクロック周波数に合成して合成/インプリメンテーション ツールに余裕を持たせます。
  - HLS に渡される C/C++ コードを変更するか、HLS 指示子をさらに追加して HLS ブロックが速くなるようにします。
  - リソース使用量が約 80% を超える場合は (`_sds/ipi/*.log` の Vivado ツール レポートおよびその下位ディレクトリに含まれるその他のログ ファイルを参照)、デザインのサイズを減らします。デザインサイズの削減方法については、次の項目を参照してください。
- デザインが大きすぎてフィットしない
  - アクセラレーションする関数の数を減らします。
  - アクセラレータ関数のコーディング スタイルを、よりコンパクトなアクセラレータが生成されるように変更します。「プログラマ向け Vivado 高位合成ガイド」に説明されているメカニズムを使用して並列処理の量を削減します。
  - アクセラレータの複数のインスタンスが作成される原因となっているプラグマとコーディング スタイル (パイプライン処理) を変更します。
  - プラグマを使用して AXIDMA_SG の代わりに AXIFIFO のようなより小型のデータ ムーバーを選択します。
  - 入力および出力パラメーター/引数の数が少なくなるようハードウェア関数を記述し直します (特に、入力/出力がデータ ムーバー ハードウェアを共有できない連続ストリーム (シーケンシャル アクセス配列引数) タイプの場合)。

ランタイム エラーのトラブルシューティング

sdscc/sds++ を使用してコンパイルされたプログラムは、SDSoC™ 環境またはザイリンクス SDK で提供される標準のデバッガーを使用してデバッグできます。不正な結果、プログラムの早期終了、およびプログラムの応答停止などが典型的なランタイム エラーとして挙げられます。最初の 2 つのエラーは C/C++ プログラマにはよく知られており、デバッガーでコードをステップ スルーすることによりデバッグできます。

プログラムの応答停止は、#pragma SDS data access_pattern(A:SEQUENTIAL) を使用して作成したストリーミング接続を介して転送するデータ量を間違って指定したり、Vivado HLS 内の合成可能な関数でストリーミング インターフェイスを指定したり、ストリーミング ハードウェア インターフェイスを含むビルド済みライブラリの C 呼び出し可能なハードウェア関数のために発生するランタイム エラーです。プログラムの応答停止は、ストリームのコンシューマーがプロデューサーから送信されるデータを待機しているときに、プロデューサーがデータの送信を停止した場合に発生します。

ハードウェア関数からのストリーミング入力および出力となる次のようなコードがあるとします。

```
#pragma SDS data access_pattern(in_a:SEQENTIAL, out_b:SEQUENTIAL)
void f1(int in_a[20], int out_b[20]);     // declaration

void f1(int in_a[20], int out_b[20]) {    // definition
   int i;
   for (i=0; i < 19; i++) {
       out_b[i] = in_a[i];
   }
}
```

このループでは in_a ストリームが 19 回読み出されるのに、in_a[] のサイズが 20 になっています。このため、f1 にストリームされるすべてのデータが処理されるまで f1 の呼び出し元が待機すると、永久的に待機することになり、応答停止となります。同様に、f1 で 20 個の int 値が送信されるのを呼び出し元が待機すると、f1 では 19 個しか送信されないため、永久的に待機することになります。このようなプログラムの応答停止の原因となるプログラム エラーは、非シーケンシャル アクセスや関数内の不正なアクセス カウントなどのストリーミング アクセス エラーがフラグされるようコードで設定すると検出できます。ストリーミング アクセスの問題は、通常ログ ファイルに不正ストリーミング アクセス警告 (improper streaming access) として示され、これらが実際にエラーであるかをユーザーが確認する必要があります。

次に、ランタイム エラーのその他の原因を示します。

- wait() 文を不正に配置した場合は、次のようになります。
  - ハードウェア アクセラレータで正しい値が書き込まれる前にソフトウェアが無効なデータを読み出す
  - 関連するアクセラレータよりも前にブロッキング wait() が呼び出されて、システムが停止する
- メモリー貫性を持たせる #pragma SDS data mem_attribute プラグマの使用に一貫性がないと、不正な結果となることがある

パフォーマンス問題のトラブルシューティング

SDSoC 環境では、sds_clock_counter() 関数により基本的なパフォーマンス監視機能が提供されています。この関数を使用すると、アクセラレーションされるコードとされないコードなど、コード セクション間における実行時間の差異を調べることができます。

Vivado HLS レポート ファイル (_sds/vhls/…/*.rpt) でレイテンシ値を見ると、実際のハードウェア アクセラレーション時間を予測できます。SDSoC IDE プロジェクトの [Platform Details] で CPU クロック周波数を、[SDSoC Project Overview] でハードウェア関数のクロック周波数を確認できます。X アクセラレータのクロック サイクルのレイテンシは、X * (processor_clock_freq/accelerator_clock_freq) プロセッサ クロック サイクルです。実際の関数呼び出しにかかる時間とこの時間を比較すると、データ転送のオーバーヘッドを確認できます。

パフォーマンスを最大限に改善するには、アクセラレーションされる関数の実行に必要な時間が元のソフトウェア関数の実行に必要な時間よりもかなり短くなることが必要です。そうならない場合は、sdscc/sds++ コマンド ラインで別の clkid を選択して、アクセラレータをより高速の周波数で実行してみてください。この方法で改善が見られない場合は、データ転送のオーバーヘッドがアクセラレーションされる関数の実行時間に影響していないかを確認し、このオーバーヘッドを減らす必要があります。デフォルトの clkid はすべてのプラットフォームで 100 MHz です。特定のプラットフォームの clkid 値の詳細は、sdscc -sds-pf-info <platform name> を実行すると取得できます。

データ転送のオーバーヘッドが大きい場合は、次を変更すると改善される可能性があります。

- より多くのコードをアクセラレーションされる関数に移動して、この関数の計算にかかる時間を長くし、データ転送にかかる時間との比率を向上させます。
- コードを変更するかプラグマを使用して必要なデータのみを転送するようにし、転送するデータ量を減らします。

アプリケーションのデバッグ

SDSoC™ 環境を使用すると、SDSoC IDE を使用してプロジェクトを作成およびデバッグできます。プロジェクトは、ユーザー定義の makefile を使用して SDSoC IDE 外で作成することも可能で、コマンド ラインまたは SDSoC IDE のいずれでもでデバッグできます。

SDSoC IDE でのインタラクティブなデバッガーの使用については、『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) の「チュートリアル : システムのデバッグ」を参照してください。

XILINX
ALL PROGRAMMABLE™

第 4 章

システム パフォーマンスの向上

全体的なシステム パフォーマンスに影響する要素は多数あります。適切に設計されたシステムでは、すべてのハードウェア コンポーネントが有益な処理を実行するように、計算と通信のバランスが取られます。計算集約型のアプリケーションでは、ハードウェア アクセラレータのスループットを最大にし、レイテンシを最低限に抑えるようにしてください。メモリ集約型のアプリケーションでは、ハードウェアの一時的なおよび空間的な局所性を増加するようアルゴリズムを再構築することが必要な場合があります。たとえば、外部メモリへのランダム配列アクセスではなく、copy-loop や memcpy を追加してデータ ブロックをハードウェアに戻すなどです。

このセクションでは、コンパイラを制御して次を実行することにより全体的なシステム パフォーマンスを向上できるようにするため、SDSoC システム コンパイラの基本的な原則と推論規則について説明します。

- プログラマブル ロジックから外部メモリへのアクセスを向上
- プログラマブル ロジックでの同時処理および並列処理を増加

SDSoC 環境では、通信と計算のバランスが取られるようにハードウェア関数およびハードウェア関数への呼び出しを構成し、sdscc システム コンパイラに指示を与えるプラグマをソース コードに挿入することにより、システム生成プロセスを制御します。

プラットフォームおよびハードウェアにインプリメントするプログラムの関数セットを選択すると、アプリケーション ソース コードによりハードウェア/ソフトウェア インターフェイスが暗示的に定義されます。sdscc/sds++ システム コンパイラは、ハードウェア関数に関連するプログラム データ フローを解析し、各関数呼び出しをスケジュールして、プログラマブル ロジックにハードウェア関数を実現するハードウェア アクセラレータとデータ モーション ネットワークを生成します。これは、標準 ARM アプリケーション バイナリ インターフェイスを介してスタックに各関数呼び出しをインプリメントするのではなく、ハードウェア関数呼び出しを元のハードウェア関数と同じインターフェイスを持つ関数スタブに対する呼び出しとして再定義することにより実行されます。これらのスタブは、send / receive ミドルウェア層への下位関数呼び出しとしてインプリメントされます。このミドルウェア層は、プラットフォーム メモリ、CPU、およびハードウェア アクセラレータの間でデータを効率的に転送し、必要に応じて基になるカーネル ドライバーへのインターフェイスとなります。

send/receive 呼び出しは、配列引数のメモリ割り当て、ペイロード サイズ、関数引数の対応するハードウェア インターフェイスなどのプログラム特性と、メモリ アクセス パターンやハードウェア関数のレイテンシなどの関数特性に基づいて、データ ムーバー IPと共にハードウェアにインプリメントされます。

ソフトウェア プログラムとハードウェア関数の間の各転送には、データ ムーバーが必要です。データ ムーバーは、データを移動するハードウェア コンポーネントとオペレーティング システム特定のライブラリ関数で構成されます。次の表に、サポートされるデータ ムーバーとそれぞれの特性を示します。

図 4-1 : SDSoC でサポートされるデータ ムーバー

SDSoC Data Mover	Vivado IP Data Mover	Accelerator IP Port Types	Transfer Size	Contiguous Memory Only
axi_lite	processing_system7	register, axilite		
axi_dma_simple	axi_dma	bram, ap_fifo, axis	< 8 MB	✓
axi_dma_sg	axi_dma	bram, ap_fifo, axis		
axi_dma_2d	axi_dma	bram		✓
axi_fifo	axi_fifo_mm_s	bram, ap_fifo, axis	(≤ 300 B)	
zero_copy	accelerator IP	aximm master		✓

X14762-070315

スカラー変数は、常に axi_lite データ ムーバーを使用して AXI4-Lite バス インターフェイスを介して転送されます。配列引数の場合は、転送サイズ、ハードウェア関数のポート マップ、および関数呼び出しサイト情報に基づいてデータ ムーバーが推論されます。axi_dma_simple データ ムーバーは、最も効率的なバルク転送エンジンですが、8MB までの転送しかサポートされないので、これより大きい転送には axi_dma_sg (スキャッター ギャザー DMA) データ ムーバーが必要です。axi_fifo データ ムーバーには、DMA ほど多くのハードウェア リソースは必要ありませんが、転送レートが遅いので、最大 300 バイトのペイロードまでに使用することをお勧めします。

プログラム ソースの関数宣言の直前に次のようなプラグマを挿入すると、別のデータ ムーバーを選択できます。

```
#pragma SDS data data_mover(A:AXIDMA_SIMPLE)
```

#pragma SDS は常に処理されるので、「SDSoC でサポートされるデータ ムーバー」に示されているデータ ムーバーの要件に従っていることを必ず確認してください。

メモリ割り当て

sdscc/sds++ コンパイラは、プログラムを解析し、ソフトウェアとハードウェア間の各ハードウェア関数呼び出しの要件を満たすデータ ムーバーをペイロード サイズ、アクセラレータのハードウェア インターフェイス、および関数引数のプロパティに基づいて選択します。コンパイラで配列引数を物理的に隣接したメモリに確実に配置できる場合は、最も効率の高いデータ ムーバーを使用できます。次の sds_lib ライブラリ関数を使用して配列を割り当てまたはメモリ マップすると、メモリが物理的に隣接していることをコンパイラに通知できます。

```
sds_alloc(size_t size); // guarantees physically contiguous memory
sds_mmap(void *paddr, size_t size, void *vaddr); // paddr must point to contiguous memory
sds_register_dmabuf(void *vaddr, int fd); // assumes physically contiguous memory
```

プログラム構造が原因で sdscc コンパイラでメモリが隣接していることを推測できない場合は、次のような警告メッセージが表示されます。

```
WARNING: [SDSoC 0-0] Unable to determine the memory attributes passed to foo_arg_A of function
foo at foo.cpp:102
```

次のプラグマを関数宣言の直前に挿入すると、データが物理的に隣接するメモリに割り当てられていることをコンパイラに通知できます。このプラグマでは必ずしも物理的に隣接したメモリに割り当てられるとはかぎらないので、このようなメモリは sds_alloc を使用して割り当てるようにしてください。

```
#pragma SDS data mem_attribute (A:PHYSICAL_CONTIGUOUS) // default is NON_PHYSICAL_CONTIGUOUS
```

コピーおよび共有メモリ セマンティクス

デフォルトでは、ハードウェア関数呼び出しには関数引数のコピー インおよびコピー アウト セマンティクスが関係します。ハードウェア関数引数の共有メモリ モデルを強制することもできますが、バースト転送のスループットが良い一方で、プログラマブル ロジックから外部 DDR へのレイテンシが CPU と比較して大幅に長くなることに注意する必要があります。変数転送で共有メモリ セマンティクスを使用することを宣言するには、次のプラグマを関数宣言の直前に挿入します。

```
#pragma SDS data zero_copy(A[0:<array_size>]) // array_size = number of elements
```

合成可能なハードウェア関数内では、共有メモリから 1 語を読み書き (zero_copy プラグマを使用) するのは通常非効率です。memcpy を使用してメモリからデータをバーストで読み書きし、ローカル メモリに格納する方が効率的です。

コピーおよびゼロ コピー メモリ セマンティクスでは、プログラマブル ロジックと外部 DDR の間でデータをストリーミングしてメモリ効率を最大化し、変数に対して非シーケンシャル アクセスおよび繰り返しアクセスを実行する必要がある場合にハードウェア関数内のローカル メモリにデータを格納するのも効率的な方法です。たとえば、ビデオ アプリケーションでは通常データがピクセル ストリームとして入力され、FPGA メモリにライン バッファーがインプリメントされてピクセル ストリーム データへの複数アクセスがサポートされます。

ハードウェア関数で配列データ転送にストリーミング アクセスが許容される (各エレメントがインデックス順に 1 回だけアクセスされる) ことを sdscc に宣言するには、次のプラグマを関数プロトタイプの直前に挿入します。

```
#pragma SDS data access_pattern(A:SEQUENTIAL) // access pattern = SEQUENTIAL | RANDOM
```

ポインター型引数としてハードウェア関数に渡された配列では、コンパイラが転送サイズを推論できる場合もありますが、できない場合は次のようなメッセージが表示されます。

```
ERROR: [SDSoC 0:0] The bound callers of accelerator foo have different/

      indeterminate data size for port p.
```

次を使用して転送するデータのサイズを指定します。

```
#pragma SDS data copy(p[0:<array_size>]) // for example, int *p
```

データ転送サイズは関数呼び出しごとに変更可能で、プラグマ定義で <array_size> を関数呼び出しのスコープ内で設定することにより (サイズ設定のすべての変数がその関数へのスカラー引数)、ハードウェア関数で不要なデータ転送を回避できます。

```
#pragma SDS data copy(A[0:L+2*T/3]) // scalar arguments L, T to same function
```

データ キャッシュ コヒーレンシ

sdscc/sds++ コンパイラでは、システムで必要なデータ ムーバーに対して自動的にソフトウェア コンフィギュレーション コードが生成されます。このコードには、必要に応じて下位デバイスのドライバーへのインターフェイスも含まれます。デフォルトでは、システム コンパイラにより、CPU とハードウェア関数の間で渡される配列に割り当てられているメモリのキャッシュ コヒーレンシが保持されると想定されます。このため、ハードウェア関数にデータを転送する前にキャッシュをフラッシュしたり、ハードウェア関数からメモリにデータを転送する前にキャッシュを無効にしたりするコードがコンパイラにより生成される場合があります。いずれの動作も正確性のために必要ですが、パフォーマンスに影響します。たとえば Zynq® デバイスの HP ポートを使用する場合、CPU がメモリにアクセスしないことがわかっていれば、アプリケーションの正確性がキャッシュ コヒーレンシに依存しないことを示すことができます。不要なキャッシュ フラッシュのオーバーヘッドを回避するには、関数宣言の直前に次のプラグマを挿入します。

```
#pragma SDS data mem_attribute(A:NON_CACHEABLE) // default is CACHEABLE
```

配列をキャッシュ不可と宣言すると、メモリの指定の配列にアクセスする際にコンパイラでキャッシュ コヒーレンシを管理する必要はありませんが、必要に応じてユーザーが管理する必要があります。典型的な使用例として、一部のフレーム バッファーがプログラマブル ロジックでアクセスされるが CPU ではアクセスされないビデオ アプリケーションが挙げられます。

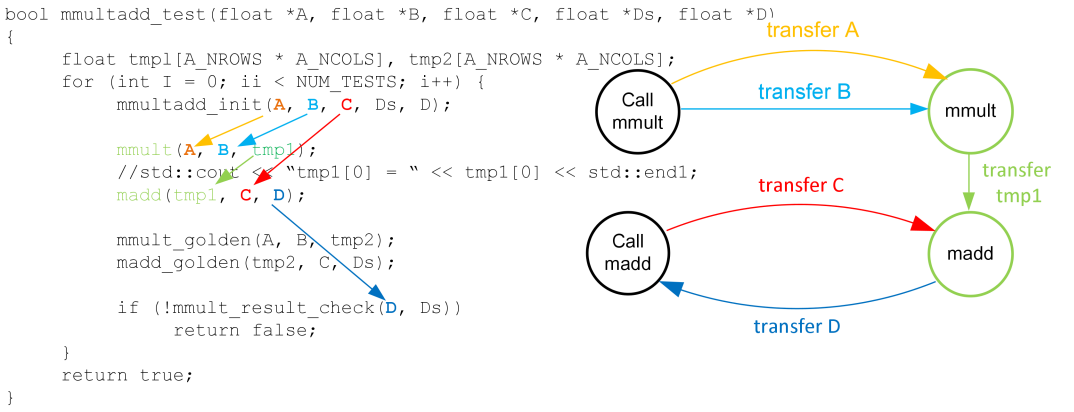
システムでの並列処理および同時処理の増加

同時処理のレベルを増加することは、システムの全体的なパフォーマンスを向上するための標準的な方法であり、並列処理のレベルを増加することは同時処理を増加させる標準的な方法です。プログラマブル ロジックは、同時実行されるアプリケーション特定のアクセラレータを含むアーキテクチャをインプリメントするのに適しており、特にデータ プロデューサーとコンシューマー間で同期化されるフロー制御ストリームを介した通信に適しています。

SDSoC 環境では、関数およびデータ ムーバー レベルでのマクロ アーキテクチャの並列処理、ハードウェア アクセラレータ内でのマクロ アーキテクチャの並列処理を制御できます。sdscc システム コンパイラでシステム接続とデータ ムーバーがどのように推論されるかを理解することにより、必要に応じてアプリケーション コードを構成してプラグマを適用して、アクセラレータとソフトウェア間のハードウェア接続、データ ムーバーの選択、ハードウェア関数のアクセラレータ インスタンス数、タスク レベルのソフトウェア制御を制御できます。Vivado HLS または C 呼び出し可能/リンク可能ライブラリとして組み込む IP 内で、マイクロ アーキテクチャの並列処理、同時処理、およびハードウェア関数のスループットを制御できます。「プログラマ向け Vivado 高位合成ガイド」に、SDSoC 環境内で使用可能な効率的なハードウェア関数マイクロアーキテクチャを作成するためのガイドラインおよび設計手法が説明されています。

システム レベルでは、ハードウェア関数間のデータ フローでプログラマブル ロジックとシステム メモリの間の引数転送が不要な場合は、sdscc コンパイラによりハードウェア関数がチェーン接続されます。たとえば、mmult および madd 関数がハードウェアに選択されている次の図に示すコードがあるとします。

図 4-2 : 直接接続を使用したハードウェア/ソフトウェアの接続

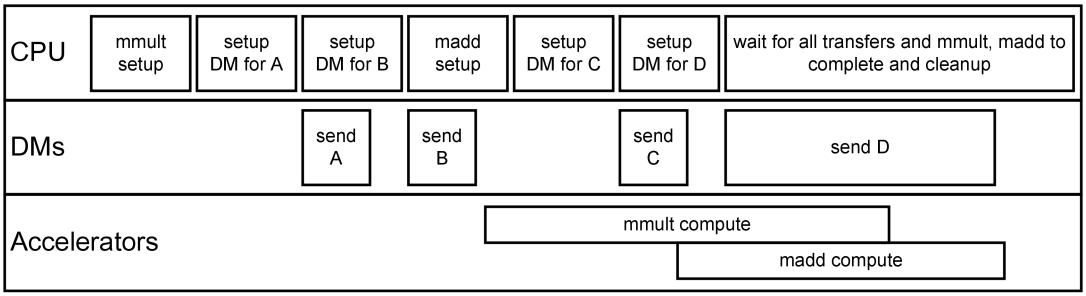


X14763-070115

2 つのハードウェア関数間でデータを渡すのには中間配列変数 tmp1 のみが使用されるので、sdscc システム コンパイラにより 2 つの関数が直接接続を使用してチェーン接続されます。

次の図に示すように、ハードウェアへの呼び出しのタイムラインを考慮すると有益です。

図 4-3 : mmult/madd 関数呼び出しのタイムライン

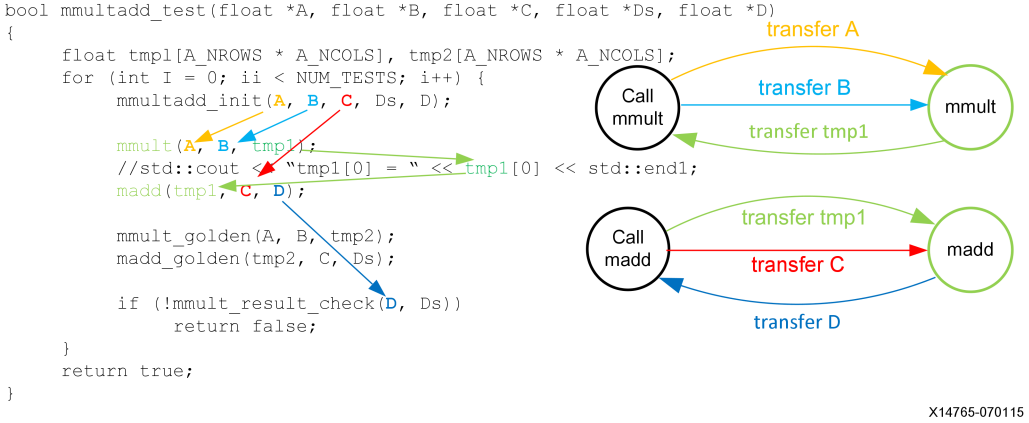


X14764-070115

プログラムでは元のプログラム セマンティクスが保持されますが、標準 ARM プロシージャ呼び出しシーケンスではなく、各ハードウェア関数呼び出しがデータ ムーバー (DM) とアクセラレータの両方に対してセットアップ、実行、およびクリーンアップを含む複数のフェーズに分割されます。CPU は各ハードウェア関数 (基になる IP 制御インターフェイス) と関数呼び出しのデータ転送をノンブロッキング API でセットアップし、すべての呼び出しと転送が完了するのを待ちます。図に示す例では、mmult と madd 関数の入力が使用可能になると、これらの関数が同時に実行されます。プログラム、データ ムーバー、アクセラレータ構造に基づいて sdscc により自動的に生成された制御コードにより、コンパイルされたプログラムですべての関数呼び出しが調整されます。

通常、sdscc コンパイラでアプリケーション コード内の関数呼び出しの悪影響を判断することはできないので (たとえば sdscc でリンクされたライブラリ内の関数のソース コードにアクセスできないなど)、ハードウェア関数呼び出し間で変数の中間アクセスが発生する場合は、データをメモリに戻す必要があります。たとえば、次の図に示すデバッグ プリント文のコメントを不注意にはずしてしまうと、大幅に異なるデータ転送グラフとなり、その結果システムおよびアプリケーションのパフォーマンスがまったく異なるものになる可能性があります。

図 4-4 : 直接接続が切断されたハードウェア/ソフトウェアの接続



プログラムでは、複数の呼び出しサイトから 1 つのハードウェア関数を呼び出すことができます。この場合、sdscc コンパイラは次のように動作します。関数呼び出しのどれかが直接接続データ フローとなった場合、sdscc により同様の直接接続をサービスするハードウェア関数のインスタンスと、メモリ (ソフトウェア) とプログラマブル ロジック間の残りの呼び出しをサービスするハードウェア関数のインスタンスが作成されます。

ハードウェア関数間を直接接続データ フローを使用してアプリケーション コードを構成するのが、プログラマブル ロジックで高パフォーマンスを達成する最適な方法の 1 つです。データ ストリームで接続されたアクセラレータの多段パイプランを作成することにより、同時実行の可能性が高くなります。

sdscc コンパイラを使用して並列処理と同時処理を増加させるには、もう 1 つ方法があります。ハードウェア関数を呼び出す直前に次のプラグマを挿入して、ハードウェア関数の複数のインスタンスが作成されるようにすることができます。

```
#pragma SDS async(<id>) // <id> a non-negative integer
```

このプラグマは、<id> で参照されているハードウェア インスタンスを作成します。ハードウェア関数用に生成された制御コードは、関数実行の完了を待たずに、すべてのセットアップが完了するとすぐに呼び出し元に戻ります。プログラムの適切なポイントに同じ <id> に対応する wait プラグマを挿入することにより、プログラムで正しく関数呼び出しを同期化する必要があります。

```
#pragma SDS wait(<id>) // <id> synchronizes to hardware function with <id>
```

次に、ハードウェア関数 mmult の 2 つのインスタンスを作成するコード例を示します。

```
{
#pragma SDS async(1)
 mmult(A, B, C); // instance 1
#pragma SDS async(2)
 mmult(D, E, F); // instance 2
#pragma SDS wait(1)
#pragma SDS wait(2)
}
```

async メカニズムにより、ハードウェア スレッドを明示的に処理して非常に高レベルの並列処理および同時処理を達成することもできますが、明示的なマルチスレッド プログラミング モデルでは同期に細心の注意を払い、非決定の動作やデッドロックを回避する必要があります。

XILINX
ALL PROGRAMMABLE™

第 5 章

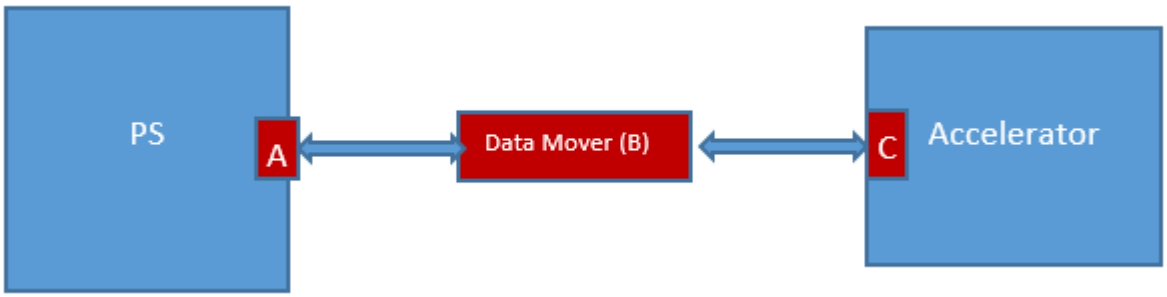
SDSoC でのデータ モーション ネットワークの生成

ここでは、SDSoC™ 環境でデータ モーション ネットワークを生成するためのコンポーネントについて説明し、SDSoC で生成されるデータ モーション ネットワークについて理解しやすくします。また、適切な SDSoC プラグマを使用してデータ モーション ネットワークを生成するためのガイドラインも提供します。

データ モーション ネットワーク

SDSoC™ のデータ モーション ネットワークは、アクセラレータのハードウェア インターフェイス、PS とアクセラレータ間とアクセラレータ同士のデータ ムーバー、PS のメモリ システム ポートの 3 つのコンポーネントから構成されます。次の図は、これらの 3 つのコンポーネントを示しています。

図 5-1 : データ モーション ネットワーク コンポーネント



アクセラレータのインターフェイス

SDSoC™ で生成されるアクセラレータのインターフェイスは、引数のデータ型によって異なります。

スカラー

スカラー引数の場合、アクセラレータの入力および出力を通すためにレジスタ インターフェイスが生成されます。

配列

配列を転送するためのアクセラレータのハードウェア インターフェイスは、そのアクセラレータが配列内のデータにどのようにアクセスするかによって、RAM インターフェイスかストリーミング インターフェイスのどちらかにできます。

RAM インターフェイスの場合、アクセラレータ内でデータがランダムにアクセスできるようになりますが、アクセラレータ内でメモリ アクセスが発生する前に、配列全体がアクセラレータに転送されるようにする必要があります。さらに、このインターフェイスを使用すると、配列を格納するため、アクセラレータ側に BRAM リソースが必要となります。

ストリーミング インターフェイスの場合は、配列全体を格納するためのメモリは必要ないので、配列エレメントの処理をパイプラインできます。 つまり、アクセラレータで前の配列エレメントを処理中に次の配列エレメントの処理を開始できます。ただし、ストリーミング インターフェイスを使用する場合、アクセラレータが厳しいシーケンシャル順序で配列にアクセスするようにする必要があり、転送されるデータ量はアクセラレータの予測と同じにする必要があります。

SDSoC では、デフォルトで配列に対して RAM インターフェイスが生成されるようになっていますが、ストリーミング インターフェイスを生成するよう指定するプラグマが含まれています。

構造体またはクラス

構造体/クラス (struct/class) 引数の場合、アクセラレータ インターフェイスは引数のプロパティによって異なります。

1. すべての構造体/クラス階層が削除されるように単一の構造体/クラスがフラットになります。インターフェイスはデータ メンバーごとに、スカラーであるか配列であるかによって異なります。
2. 構造体/クラスの配列は、プロセッサとアクセラレータ間で同じメモリ レイアウトになるようにパックされて、正しく揃えられる必要があります。SDSoC コンパイラは自動的にアクセラレータに対して data_pack 指示子を挿入し、ソース コードでその構造/クラスに対して正しいパックおよびアライメント属性を挿入するように促すメッセージが表示されます。構造体の配列を使用した例は、[Color Space Conversion] テンプレートに含まれています。

データ ムーバー

データ ムーバーは、PS とアクセラレータ間およびアクセラレータ同士の間でデータを転送します。SDSoC™ では、転送されるデータのプロパティとサイズに基づいてさまざまなタイプのデータ ムーバーが生成できます。

スカラー

スカラー データは、常に AXI_LITE データ ムーバーで転送されます。

配列

SDSoC では、メモリ属性と配列のデータ サイズによって、AXI_DMA_SG、AXI_DMA_SIMPLE、AXI_DMA_2D、AXI_FIFO、AXI_M または AXI_LITE データ ムーバーが生成できます。たとえば、配列が `malloc()` を使用してアロケートされているために、メモリが物理的に連続していない場合、SDSoC では通常 AXI_DMA_SG が生成されますが、データ サイズが 300 バイト未満の場合は AXI_FIFO が代わりに生成されます。これは、AXI_FIFO の方がデータ転送時間が AXI_DMA_SG よりも短く、PL リソースの占有も少ないからです。

構造体またはクラス

1 つの構造体/クラス (`struct/class`) はフラット化されるので、データ メンバーごとに、スカラーか配列かによって使用されるデータ ムーバーが異なります。構造体/クラス/の配列の場合、データ ムーバーの選択肢は前述の「配列」と同じです。

システム ポート

システム ポートは、データ ムーバーを PS に接続します。これは、Zynq の ACP ポートまたは AFI ポートのいずれかにできます。ACP ポートは、キャッシュ コヒーレンシ ポートで、キャッシュ コヒーレンシはハードウェアで維持されます。AFI ポートは、キャッシュ コヒーレンシ ポートではありません。キャッシュ コヒーレンシ (例：キャッシュ フラッシュおよびキャッシュ無効化) は、必要に応じてソフトウェアで維持されます。ACP ポートと AFI ポートのどちらを選択するかは、転送データのキャッシュ要件によって異なります。

データ モーション ネットワーク生成をガイドするための SDS プラグマの使用

SDS プラグマがない場合、SDSoC™ ではソース コードの解析に基づいてデータ モーション ネットワークが生成されます。ただし、SDSoC にはデータ モーション ネットワーク生成をガイドするためのプラグマもいくつか提供されています。

アクセラレータのインターフェイス

次の SDS プラグマは、アクセラレータのインターフェイス生成をガイドするために使用できます。

```
#pragma SDS data access_pattern(arg:pattern)
```

「pattern」は RANDOM か SEQUENTIAL のどちらか、「arg」はアクセラレータ ファンクションの配列引数名にします。

配列引数のアクセス パターンが RANDAM に指定されるとRAM インターフェイスが生成され、SEQUENTIAL に指定されるとストリーミング インターフェイスが生成されます。このプラグマに関しては、次の点に注意してください。

- 配列引数のデフォルトのアクセス パターンは RANDAM です。
- 指定したアクセス パターンは、アクセラレータ ファンクションのビヘイビアーと一貫している必要があります。SEQUENTIAL アクセス パターンの場合、ファンクションがすべての配列エレメントに厳しいシーケンシャル順序でアクセスする必要があります。
- このプラグマは、zero_copy プラグマがない引数にのみ適用できます。これについては、後で説明します。

データ ムーバー

配列を転送するのにどのデータ ムーバーを使用するかは、配列の 2 つの属性 (データ サイズと物理メモリの連続性) によって異なります。たとえば、メモリ サイズが 1 MB で物理的に連続していない場合 (malloc() でアロケートされる場合)、AXIDMA_SG を使用する必要があります。次の表は、これらのデータ ムーバーの適用基準を示しています。

表 5-1 : データ ムーバーの選択

データ ムーバー	物理メモリの連続性	データ サイズ (バイト)
AXIDMA_SG	連続または非連続	> 300
AXIDMA_Simple	連続	< 8M
AXIDMA_2D	連続	< 8M
AXI_FIFO	非連続	< 300

通常、SDSoC™ コンパイラではこれら 2 つの属性用にハードウェア アクセラレータへ転送される配列が解析され、適切なデータ ムーバーが選択されますが、このような解析がその時点でできないこともあります。SDSoC™ では、SDS プラグマを介してメモリ属性を指定するように尋ねる次のような警告メッセージが表示されます。

```
WARNING: [SDSoC 0-0] Unable to determine the memory attributes passed to rgb_data_in of function
img_process at C:/simple_sobel/src/main_app.c:84
```

メモリ属性を指定するプラグマは、次のようになります。

```
#pragma SDS data mem_attribute(arg:contiguity|cache)
```

contiguity は PHYSICAL_CONTIGUOUS または NON_PHYSICAL_CONTIGUOUS のいずれかにする必要があります。cache については、後で説明します。contiguity または cache のいずれかまたは両方を指定できます。両方の属性を指定する場合は、| で分けてください。データ サイズを指定するプラグマは、次のようになります。

```
#pragma SDS data copy(arg[offset:size])
```

size は、数値または任意の演算式にできます。

zero_copy データ ムーバー

前述したように、zero_copy はアクセラレータ インターフェイスとデータ ムーバーの両方に使用できるので、特殊なデータ ムーバーです。このプラグマの構文は、次のとおりです。

```
#pragma SDS data zero_copy(arg[offset:size])
```

[offset:size] はオプションで、配列のデータ転送サイズがコンパイル時に決定できない場合にのみ必要です。

デフォルトでは、SDSoC は配列引数の copy を実行します。つまり、データはデータ ムーバーを介して PS からアクセラレータに明示的にコピーされますが、この zero_copy プラグマを使用すると、SDSoC でアクセラレータの指定した引数に対して AXI-Master インターフェイスが生成され、アクセラレータ コードで指定したとおりに PS からデータが取得されます。

zero_copy プラグマを使用するには、配列に該当するメモリは物理的に連続している (sds_alloc でアロケートされる) 必要があります 。

システム ポート

システム ポートの選択は、データのキャッシュ属性とデータ サイズによって異なります。データが sds_alloc_non_cacheable() または sds_register_dmabuf() でアロケートされる場合は、キャッシュのフラッシュ/無効化を避けるために AFI ポートに接続することをお勧めします。その他の方法でデータがアロケートされる場合は、キャッシュのフラッシュ/無効化を高速にするため ACP ポートに接続することをお勧めします。データ サイズがキャッシュ サイズよりもかなり大きい場合は、ACP ポートを介してデータを転送すると、キャッシュがスラッシングしてしまうので、AFI ポートに接続してください。

SDSoC では、データが送信されてアクセラレータから受信されるようにするために、これらのメモリ属性が解析され、データ ムーバーが適切なシステム ポートに接続されるようになっていますが、ユーザーがコンパイラの指定を上書きしたい場合や、コンパイラでこのような解析が実行できない場合は、次のプラグマを使用するとデータのキャッシュ属性を指定できます。

```
#pragma SDS data mem_attribute (arg:contiguity|cache)
```

cache は、CACHEABLE または NON_CACHEABLE のいずれかにできます。このプラグマを使用すると、コンパイラ解析が上書きされるので、必ず正しいかどうかを確認してください。

データ サイズ プラグマ (#pragma SDS data copy および #pragma SDS data zero_copy) については既に説明しました。必ず指定したプラグマが正しいかどうか確認してください。

SDS プラグマ

どのデータ ムーバーを使用するか直接指定するには、次のプラグマを使用する必要があります。

```
#pragma SDS data data_mover(arg:dm)
```

dm は AXIDMA_SG、AXIDMA_SIMPLE、AXIDMA_2D、または AXI_FIFO にでき、arg はアクセラレータ ファンクションの配列またはポインター引数にできます。このプラグマを使用する場合は、上記の表の条件が満たされていないと、デザインがハードウェアで動作しなくなることがあるので注意してください。

接続するシステム ポートを直接指定する場合は、次のプラグマを使用してください。

```
#pragma SDS data sys_port(arg:port)
```

port は ACP または AFI のいずれかにできます。

XILINX
ALL PROGRAMMABLE™

第 6 章

コード ガイドライン

このセクションでは、SDSoC システム コンパイラを使用したアプリケーション プログラミングでの一般的なコーディング ガイドラインを示します。これらのガイドラインは、SDSoC 環境に含まれている GNU ツールチェーンを使用して Zynq® デバイス内の ARM CPU 用にクロス コンパイルされているアプリケーション コードから開始していることを前提としています。

sdscc/sds++ の起動に関するガイドライン

SDSoC IDE では、C++ ファイルに対して `sds++`、C ファイルに対して `sdscc` を起動する makefile が生成されますが、`sdscc/sds++` でコンパイルする必要があるのは次のようなコードを含むソース ファイルのみです。

- ハードウェア関数を定義するコード
- ハードウェア関数を呼び出すコード
- ハードウェア関数に送信されるバッファーを割り当てまたはメモリ マップするためなどに `sds_lib` 関数を使用するコード
- 上記の下流にある呼び出しグラフの推移閉包に関数を含むファイル

その他のソース ファイルは、ARM GNU ツールチェーンで問題なくコンパイルできます。

大型のソフトウェア プロジェクトには、`sdscc` で生成されたハードウェア アクセラレータおよびデータ モーション ネットワークに関連しない多数のファイルおよびライブラリが含まれている可能性があります。`sdscc` コンパイラで生成されたハードウェア システム (OpenCV ライブラリなど) に関連しないソース ファイルに対してエラーがレポートされた場合は、それらのファイル (またはフォルダー) を右クリックして [Properties] → [C/C++ Build] → [Settings] をクリックし、[Command] を `GCC` に変更することにより、これらのファイルを `sdscc` ではなく `GCC` でコンパイルしてください。

makefile ガイドライン

`<sdsoc_root>/samples` のデザインに含まれる makefile では、すべての `sdscc` ハードウェア関数オプションが 1 つのコマンド ラインにまとめられます。これは必須ではありませんが、ハードウェア関数を含むファイルに対して makefile のアクションを変更せずに制御構造全体とその依存性を makefile 内で保持できるという利点があります。

- SDSoC 環境コマンド ライン全体をキャプチャする make 変数を定義できます。たとえば、C++ ファイルに対して CC = sds++ ${SDSFLAGS} にし、C ファイルに sdscc を起動します。このようにすると、すべての SDSoC 環境オプションが ${CC} 変数にまとめられます。プラットフォームおよびターゲット OS をこの変数で定義します。
- ハードウェア関数を含む各ファイルに対してコマンド ラインに -sds-hw/-sds-end 節が必要です。次に例を示します。

```
-sds-hw foo foo.cpp -clkid 1 -sds-end
```

SDSoC コンパイラおよびリンカー オプションのリストは、「SDSSC/SDS++ コンパイラのコマンドおよびオプション」を参照するか、sdscc --help を使用してください。

一般的な C/C++ ガイドライン

- ハードウェア関数は、マスター スレッドで制御することにより同時に実行できます。プログラムには複数のスレッドおよびプロセスが含まれていることがありますが、ハードウェア関数を制御するマスター スレッドのみを含める必要があります。
- 最上位ハードウェア関数は、クラス メソッドではなくグローバル関数にする必要があり、オーバーロードさせることはできません。
- ハードウェア関数には、例外処理のサポートはありません。
- ハードウェア関数またはそのサブ関数内のグローバル変数がソフトウェアで実行中のほかの関数でも参照されている場合、そのグローバル変数を参照することはできません。
- ハードウェア関数が値を戻す場合、その型は 32 ビットのコンテナーに収まるスカラー型である必要があります。
- ハードウェア関数には、少なくとも 1 つの引数が含まれている必要があります。
- ハードウェア関数への出力または入出力引数は、1 度だけ設定します。ハードウェア関数内に出力または入出力スカラーに対する複数の引数が必要な場合は、ローカル変数を作成します。
- #ifdef および #ifndef プリプロセッサ文を含むコードを保護するため、定義済みマクロを使用します。マクロ名の前後にはアンダースコアを 2 つずつ付けます。例は「SDSSC/SDS++ コンパイラのコマンドおよびオプション」を参照してください。
  - __SDSCC__ マクロは、sdscc または sds++ を使用してソース ファイルをコンパイルするたびに定義されて -D オプションとしてサブツールに渡されます。また、コードが sdscc/sds++ でコンパイルされるか GNU ホスト コンパイラなどの別のコンパイラでコンパイルされるかに基づいてコードを保護するために使用できます。
  - sdscc または sds++ で Vivado HLS を使用したハードウェア アクセラレーション用にソース ファイルをコンパイルする場合は、__SDSVHLS__ マクロが定義され、-D オプションとして渡されるので、このマクロを使用して高位合成が実行されるかされないかに基づいてコードを保護することができます。

ハードウェア関数の引数型

SDSoC™ 環境の sdscc/sds++ システム コンパイラでは、C99 基本演算型の単一または配列、メンバーが単一または配列にフラット化される C99 基本演算型 struct または class (階層構造体をサポート)、メンバーが単一の C99 基本演算型にフラット化される struct の配列などになるような型のハードウェア関数の引数がサポートされます。スカラー引数は、32 ビットのコンテナーに収まる必要があります。SDSoC™ 環境では、引数型と次のプラグマに基づいて、各ハードウェア インターフェイス タイプが自動的に推論されます。

```
#pragma SDS data copy|zero_copy
#pragma SDS data access_pattern
```

インターフェイスの互換性が損なわれないようにするため、Vivado® HLS インターフェイス タイプ指示子およびプラグマをソース コードに含めるのは、「Vivado HLS 関数引数型」に示すように、sdscc で適切なハードウェア インターフェイス指示子が生成されないときのみにしてください。

- Vivado® HLS では、任意精度型の ap_fixed<int>、ap_int<int>、および hls::stream クラスが提供されています。SDSoC 環境では、最上位ハードウェア関数の引数の幅を 8、16、32、または 64 ビットにし、これらの宣言を #ifndef __SDS_VHLS__ を使用して保護して、char、short、int、または long long などの同様のサイズの C99 型に強制する必要があります。Vivado HLS hls::stream 引数は配列として sdscc/sds++ に渡す必要があります。<sdsoc_install_dir>/samples/hls_if/hls_stream のサンプルは、SDSoC 環境で HLS の hls::stream 型の引数を使用する方法を示します。
- デフォルトでは、ハードウェア関数への配列引数はデータをコピーすると転送されます。これは、#pragma SDS data copy を使用するのと同等です。このため、配列引数は入力として使用するか、出力として生成する必要があり、両方には使用しないようにします。ハードウェア関数で読み出しおよび書き込みされる配列の場合は、#pragma SDS data zero_copy を使用して、コンパイラにその配列は共有メモリ内に保持する必要があり、コピーされないように指示する必要があります。
- ハードウェア/ソフトウェア インターフェイスでのアライメントを確実にするため、ハードウェア関数の引数のデータ型として long または bool の配列を使用しないでください。

重要: ハードウェア関数のポインター引数には、特別な注意が必要です。ポインターは一般的で有益な抽象化ですが、sdscc および Vivado HLS ツールでは、Vivado HLS ツールでの合成方法のため、処理が困難となる可能性があります。

重要: デフォルトでは、プラグマがない場合、ポインター引数は C/C++ では 1 次元配列型を示す可能性もありますが、スカラー パラメーターとして処理されます。次に、使用可能なインターフェイス プラグマを示します。

- このプラグマは、共有メモリを使用したポインター セマンティクスを提供します。

```
#pragma SDS data zero_copy
```

- このプラグマは、引数をストリームにマップします。配列要素がインデックス順にアクセスされることが必要です。data copy プラグマは、sdscc システム コンパイラがデータ転送サイズを決定できず、エラーが発生した場合にのみ必要です。

```
#pragma SDS data copy(p[0:<p_size>)
#pragma SDS data access_pattern(p:SEQUENTIAL)
```

ハードウェア関数の配列に対して非シーケンシャル アクセスが必要な場合は、ポインター引数を A[1024] などのように次元を明示的に宣言した配列に変更する必要があります。

ハードウェア関数呼び出しのガイドライン

- SDSoC™ 環境で生成されたスタブ関数は、ハードウェア関数宣言内の対応する引数のコンパイル時に特定可能な配列範囲によって、正確なバイト数を転送します。ハードウェア関数で可変データ サイズを使用できる場合、次のプラグマを使用して、演算式で定義されたサイズのデータを転送するコードが生成されるように指定できます。

  ```
  #pragma SDS data copy|zero_copy(arg[0:<C_size_expr>]
  ```

  <C_size_expr> は関数宣言のスコープでコンパイルする必要があります。

  zero_copy プラグマは、引数を共有メモリにマップするよう指定します。

  意図するデータ転送サイズと実際のデータ転送サイズが異なると、システムがランタイム時に停止し、面倒なハードウェア デバッグでしか解決できない状況が発生することがあるので注意してください。

- DMA で転送された配列をキャッシュ ラインの境界 (L1 および L2 キャッシュ) に揃えます。これらの配列を割り当てるには、malloc() ではなく、SDSoC 環境で提供されている sds_alloc API か posix_memalign() を使用してください。
- 配列をページ境界に揃え、スキャッター ギャザー DMA で転送されるページ数を最小限に抑えます (malloc で割り当てられる配列など)。
- 次の場合は、sds_alloc を使用して配列を割り当てる必要があります。
  1. 配列に zero-copy プラグマを使用している
  2. シンプル DMA または 2D-DMA を使用するようにシステム コンパイラに明示的に指示するプラグマを使用している

  sds_lib.h から sds_alloc() を使用するには、sds_lib.h を含める前に stdlib.h を含める必要があります。stdlib.h は、size_t タイプを提供するために含めます。

XILINX
ALL PROGRAMMABLE™

第 7 章

プログラマ向け Vivado 高位合成ガイド

このセクションでは、プログラマブル ロジックにクロスコンパイル可能な効率的なコードを記述するための概要を示します。

SDSoC 環境では、Vivado HLS をプログラマブル ロジックのクロス コンパイラとして使用して、C/C++ 関数がハードウェアに変換されます。このセクションで説明される原則に従うと、合成済み関数のパフォーマンスを劇的に改善でき、アプリケーションの全体的なシステム パフォーマンスを大幅に向上できる可能性があります。

最上位ハードウェア関数のガイドライン

このセクションでは、Vivado HLS ハードウェア関数で ARM GNU ツールチェーンで生成されたオブジェクト コードと一貫したインターフェイスが使用されるようにするためのコード ガイドラインを示します。

最上位ハードウェア関数引数には標準 C99 データ型を使用

1. `long` データ型は使用しないでください。`long` データ型には、64 ビット アーキテクチャ (x64 など) と 32 ビット アーキテクチャ (Zynq® の ARM A9 など) の間での移植性がありません。
2. `bool` の配列は使用しないでください。`bool` の配列のメモリ レイアウトは、ARM GCC と Vivado® HLS で異なります。
3. データ幅が 8、16、32、または 64 以外の場合は `ap_int<>`, `ap_fixed<>`, `hls::stream` を使用しないでください。SDSoC 環境での `hls::stream` の使用方法については、`<SDSoC Installation Path>/samples/hls_if/hls_stream` にあるサンプル デザインを参照してください。

最上位ハードウェア関数の引数の HLS インターフェイス指示子を使用しない

最上位ハードウェア関数には、`HLS interface` プラグマは含めないようにしてください。SDSoC 環境で適切な HLS インターフェイス指示子が生成されます。SDSoC 環境で必要な HLS インターフェイス指示子が生成されるようにするため、最上位ハードウェア関数に次の 2 つの SDSoC 環境プラグマを指定できます。

`#pragma SDS data zero_copy()` : ハードウェアに 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る共有メモリ インターフェイスを生成します。

`#pragma SDS data access_pattern(argument:SEQUENTIAL)` : ハードウェアに FIFO インターフェイスとしてインプリメントされるストリーミング インターフェイスを生成します。

重要： 最上位関数の引数に対して `#pragma HLS interface` を使用してインターフェイスを指定すると、その引数に対する HLS インターフェイス指示子が SDSoC 環境で生成されないので、ユーザーの責任で生成されたハードウェア インターフェイスがその他すべての関数引数のハードウェア インターフェイスと一貫するようにしてください。互換性のない HLS インターフェイス タイプを使用した関数があると、意味不明な sdscc エラー メッセージが表示されるので、`HLS interface` を削除しておくことをお勧めします (必須ではありません)。

最適化ガイドライン

このセクションでは、ハードウェア関数のパフォーマンスを向上させる基本的な高位合成 (HLS) の 最適化手法をいくつか紹介します。これらの手法として、関数のインライン展開、ループおよび関数のパイプライン処理、ループ展開、ローカル メモリの帯域幅の増加、およびループと関数間のデータ フローのストリーミングが挙げられます。

関数のインライン展開

ソフトウェア関数のインライン展開と同様、ハードウェア関数のインライン展開にも利点があります。

関数をインライン展開すると、実際の引数と仮引数が解決された後に、関数呼び出しが関数本体のコピーに置き換えられます。インライン展開された関数は、別の階層として表示されなくなります。関数のインライン展開では、インライン関数内の演算が周辺の演算と一緒に効率的に最適化されるので、ループの全体的なレイテンシまたは開始間隔を向上できます。

関数をインライン展開するには、インライン展開する関数の本体の最初に「`#pragma HLS inline`」と入力します。次のコードでは、`mmult_kernel` 関数がインライン展開されるように Vivado HLS に指示されます。

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                  float in_B[A_NCOLS][B_NCOLS],
                  float out_C[A_NROWS][B_NCOLS])
{
#pragma HLS INLINE
     int index_a, index_b, index_d;
     // rest of code body omitted
}
```

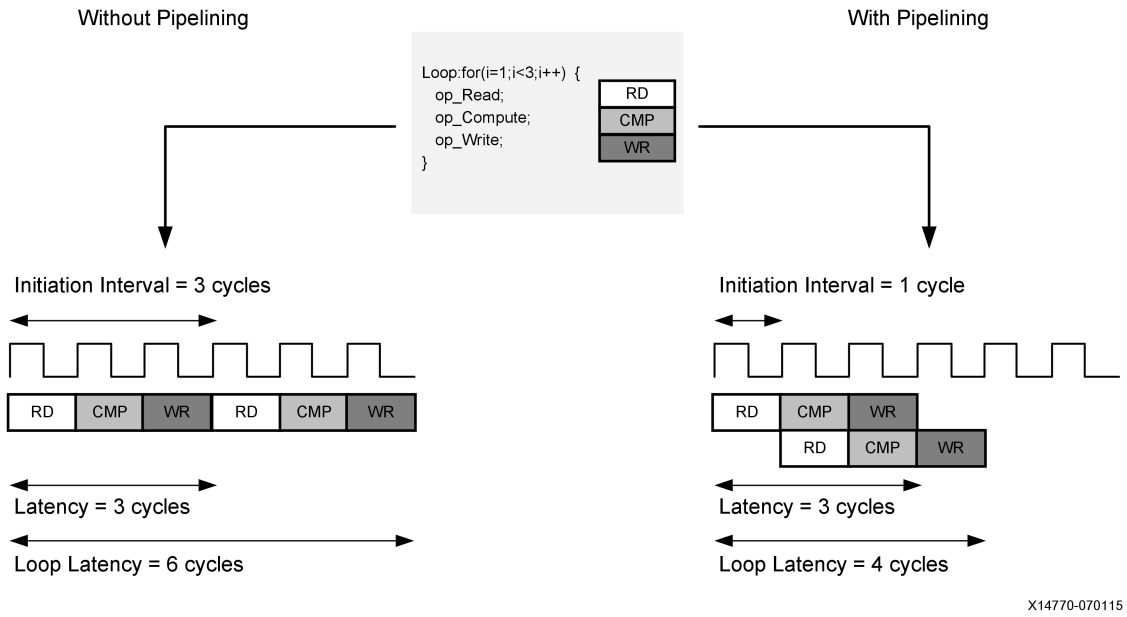
ループのパイプライン処理とループ展開

ループのパイプライン処理とループ展開は、どちらもループの繰り返し間の並列処理を可能にすることで、ハードウェア関数のパフォーマンスを改善する手法です。ここでは、ループのパイプライン処理とループ展開の基本的な概念とこれらの手法を使用するコード例を示し、これらの手法を使用して最適なパフォーマンスを達成する際に制限となる要因について説明します。

ループのパイプライン処理

C/C++ のような逐次言語の場合、ループの演算は順番に実行され、ループの次の繰り返しは、現在の繰り返しの最後の演算が終了してから開始されます。ループのパイプライン処理を使用すると、次の図に示すようにループ内の演算が並列方式でインプリメントできるようになります。

図 7-1 : ループのパイプライン処理



上図に示すように、パイプライン処理しない場合、2 つの RD 演算間に 3 クロック サイクルあるので、ループ全体が終了するのに 6 クロック サイクル必要となります。パイプライン処理を使用すると、2 つの RD 演算間は 1 クロック サイクルなので、ループ全体が終了するのに 4 クロック サイクルしか必要となりません。ループの次の繰り返しは現在の繰り返しが終了する前に開始できます。

開始間隔 (II) はループのパイプライン処理における重要な用語で、ループの連続する繰り返しの開始時間の差をクロック サイクル数で示します。「ループのパイプライン処理」の場合、ループの連続する繰り返しの開始時間の差は 1 クロック サイクルなので、開始間隔 (II) は 1 です。

ループをパイプライン処理するには、次に示すように、ループ本体の開始部分に #pragma HLS pipeline と記述します。Vivado HLS で、最小限の開始間隔でループのパイプライン処理が試みられます。

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
     for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
         float result = 0;
         for (index_d = 0; index_d < A_NCOLS; index_d++) {
             float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
             result += product_term;
         }
         out_C[index_a * B_NCOLS + index_b] = result;
     }
}
```

ループ展開

ループ展開は、ループの繰り返し間を並列処理するための別の手法で、ループ本体の複数コピーを作成して、ループの繰り返しカウンターをそれに合わせて調整します。次のコードは、展開されていないループを示しています。

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

ループを係数 2 で展開すると、次のようになります。

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

係数 N でループを展開すると、ループ本体の N 個の コピーが作成され、各コピーで参照されるループ変数 (前述の例の場合は `a[i+1]`) がそれに合わせてアップデートされ、ループの繰り返しカウンター (前述の例の場合は `i+=2`) もそれに合わせてアップデートされます。

ループ展開では、ループの各繰り返しにより多くの演算が作成されるので、Vivado HLS でこれらの演算を並列処理できるようになります。並列処理が増えると、スループットが増加し、システム パフォーマンスも向上します。係数 N がループの繰り返しの合計 (前述の例の場合は 10) よりも少ない場合、「部分展開」と呼ばれます。係数 N がループの繰り返し数と同じ場合は、「全展開」と呼ばれます。全展開の場合、コンパイル時にループ範囲がわかっている必要がありますが、並列処理は最大限に実行されます。

ループを展開するには、そのループの開始部分に `#pragma HLS unroll [factor=N]` を挿入します。オプションの `factor=N` を指定しない場合、ループは全展開されます。

```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

ループのパイプライン処理とループ展開で達成される並列処理を制限する要因

ループのパイプライン処理とループ展開は、どちらもループの繰り返し間の並列処理を可能にしますが、ループ繰り返し間の並列処理は、ループ繰り返し間のデータ依存性と、使用可能なハードウェア リソース数の 2 つの主な要因により制限されます。

連続する繰り返しにおける 1 つの繰り返しの演算から次の繰り返しの演算へのデータ依存性は「ループ キャリー依存性」と呼ばれ、現在の繰り返しの演算が終了して次の繰り返しの演算用のデータ入力が計算されるまで、次の繰り返しの演算を開始できないことを意味します。ループ キャリー依存性があると、ループのパイプライン処理を使用して達成可能な開始間隔とループ展開を使用して実行可能な並列処理が制限されます。

次の例は、変数 a と b を出力する演算と入力として使用する演算間でのループ キャリー依存性を示しています。

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

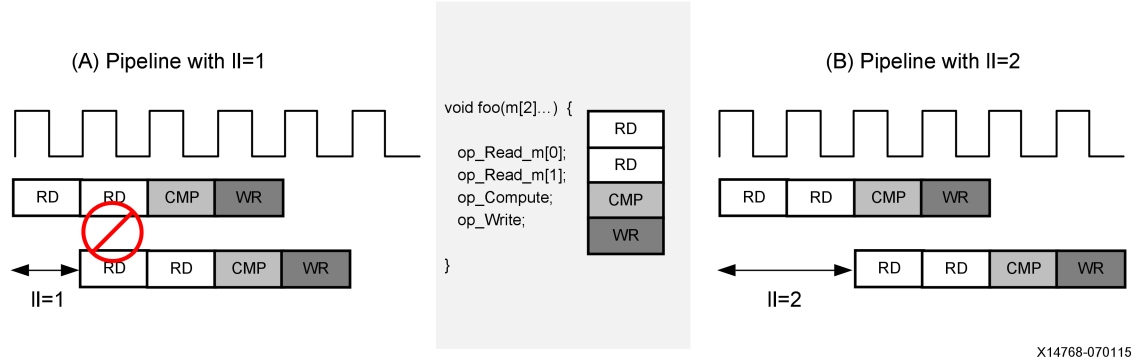
このループの次の繰り返しの演算は、現在の繰り返しが計算されて、a と b の値がアップデートされるまで開始できません。次の例に示すような配列アクセスは、ループ キャリー依存性のよくある原因です。

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

この例の場合、現在の繰り返しが配列の内容をアップデートするまでループの次の繰り返しを開始できません。ループのパイプライン処理の場合、最小の開始間隔はメモリ読み出し、加算、メモリ書き込みに必要な合計クロック サイクル数です。

使用可能なハードウェア リソース数もループのパイプライン処理およびループ展開のパフォーマンスを制限する要因です。次の図は、リソースの制限により発生する問題の例を示しています。この場合、ループを開始間隔 1 でパイプライン処理することはできません。

図 7-2 : リソースの競合



この例の場合、ループが開始間隔 1 でパイプライン処理されると、2 つの読み出しが実行されることになります。メモリにシングル ポートしかない場合、この 2 つの読み出しは同時に実行できず、2 サイクルで実行する必要があります。このため、最小の開始間隔は図の (B) に示すように 2 になります。同じことは、その他のハードウェア リソースでも発生します。たとえば、op_compute が DSP コアを使用してインプリメントされ、それが各サイクルごとに新しい入力を受信できず、このような DSP コアが 1 つしかない場合、op_compute はサイクルごとに DSP に出力できないので、開始間隔 1 は使用できません。

ローカル メモリ帯域幅の増加

このセクションでは、Vivado HLS で提供されているローカル メモリ帯域幅を増加するいくつかの方法を示します。これらの方法は、ループのパイプライン処理およびループ展開と共に使用してシステム パフォーマンスを向上できます。

C/C++ プログラムでは、配列は理解しやすく便利なコンストラクトです。配列を使用すると、アルゴリズムを簡単にキャプチャして理解できます。Vivado HLS では、各配列はデフォルトでは 1 つのポート メモリ リソースを使用してインプリメントされますが、このようなメモリ インプリメンテーションはパフォーマンス指向のプログラムでは最適なメモリ アーキテクチャでないことがあります。前のセクションの最後に、制限されたメモリ ポートにより発生するリソース競合の例を示しました。

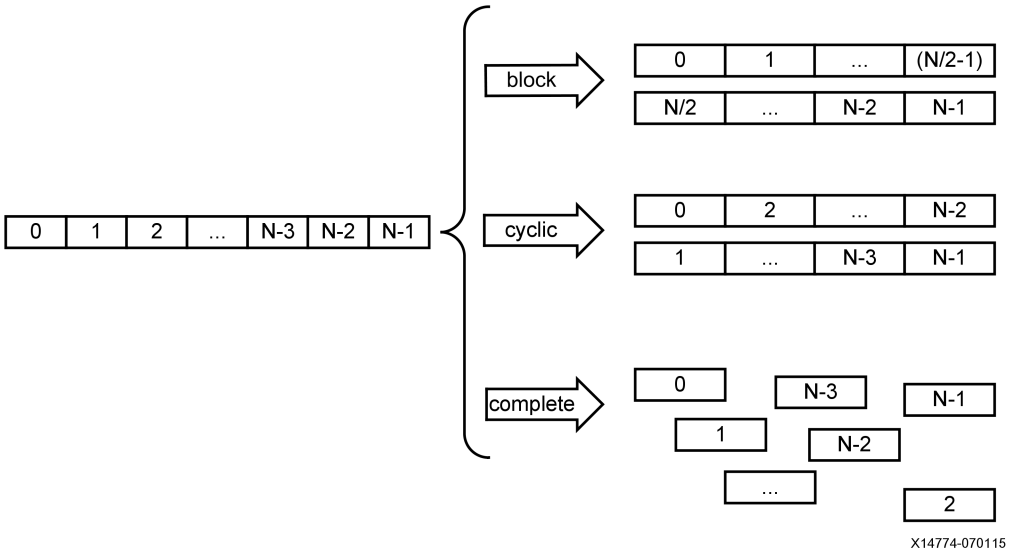
配列の分割

配列は、より小型の配列に分割できます。メモリの物理的なインプリメンテーションでは、読み出しポートと書き込みポートの数に制限があり、ロード/ストア集約型のアルゴリズムではスループットが制限されます。元の配列 (1 つのメモリ リソースとしてインプリメント) を複数の小型の配列 (複数のメモリとしてインプリメント) に分割してロード/ストア ポートの有効数を増加させることにより、メモリ帯域幅を向上できる場合があります。

Vivado HLS では、「配列の分割」に示すように、3 種類の配列分割があります。

1. block : 元の配列を、同じサイズの連続した要素のブロックに分割します。
2. cyclic : 元の配列を、元の配列の要素を交互に配置した同じサイズのブロックに分割します。
3. complete : 配列を個々の要素に分割します (デフォルト)。これは、配列をメモリとしてではなく複数のレジスタとしてインプリメントすることに対応します。

図 7-3 : 配列の分割



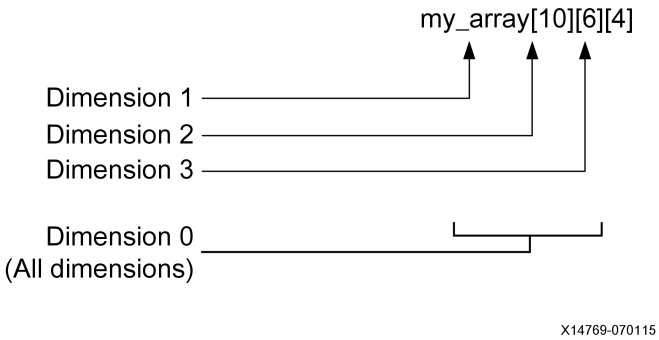
Vivado HLS で配列を分割するには、ハードウェア関数のソース コードに次を挿入します。

```
#pragma HLS array_partition variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>
```

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。「配列の分割」図では、係数として 2 が使用され、2 つの配列に分割されています。配列に含まれる要素数が指定された係数の整数倍でない場合、最後の配列に含まれる要素数はほかの配列よりも少なくなります。

多次元配列を分割する場合は、dim オプションを使用してどの次元を分割するかを指定できます。次の図に、多次元配列の異なる次元を分割した例を示します。

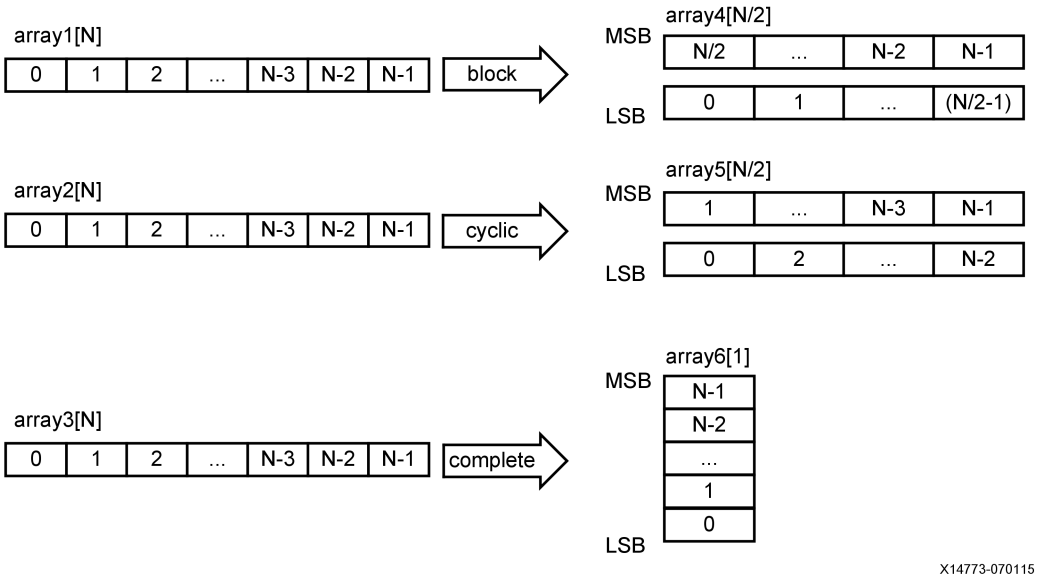
図 7-4 : 多次元配列の分割



配列の形状変更

配列の形状を変更して、メモリ帯域幅を増加できます。形状変更では、元の配列の 1 つの次元から異なる要素を取り出して、1 つの幅の広い要素に結合します。配列の形状変更は配列の分割に似ていますが、複数の配列に分割するのではなく、配列の要素の幅を広くします。次の図に、配列の形状変更の概念を示します。

図 7-5 : 配列の形状変更



Vivado HLS で配列の形状を変更するには、ハードウェア関数のソース コードに次を挿入します。

```
#pragma HLS array_reshape variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>
```

オプションは、配列分割プラグマと同じです。

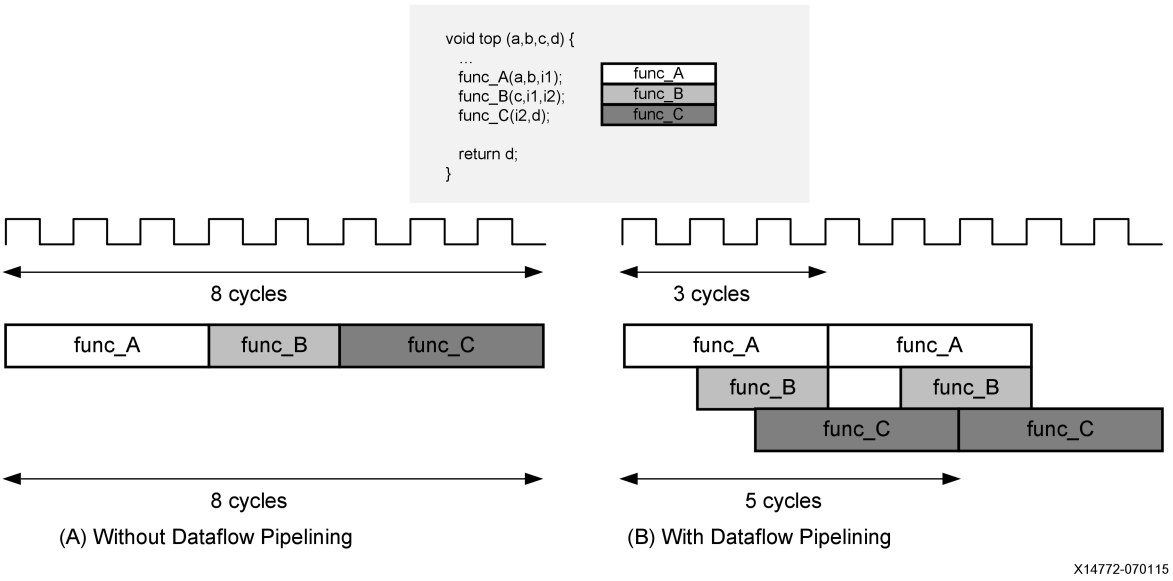
データ フローのパイプライン処理

これまでに説明した最適化手法はすべて、乗算、加算、メモリのロード/ストアなどの演算子レベルでの細粒度の並列処理最適化でした。これらの最適化では、これらの演算子間が並列処理されます。一方データ フローのパイプライン処理では、関数およびループのレベルで粗粒度の並列処理が実行されます。データ フロー パイプラン処理により、関数およびループの同時処理が増加します。

関数のデータ フローのパイプライン処理

Vivado HLS の一連の関数呼び出しは、デフォルトでは 1 つの関数が完了してから次の関数が開始します。「関数のデータ フロー パイプライン処理」の (A) に、関数のデータフ ロー パイプライン処理を実行しない場合のレイテンシを示します。3 つの関数に 8 クロック サイクルかかると仮定した場合、このコードでは func_A で新しい入力を処理できるようになるまでに 8 サイクルかかり、func_C で出力が書き込まれる (func_C の最後に出力が書き込まれると想定) までに 8 サイクルかかります。

図 7-6 : 関数のデータ フローのパイプライン処理



上記の図の (B) は、データ フロー パイプライン処理を使用した例を示しています。func_A の実行に 3 サイクルかかるとすると、func_A はこの 3 つの関数すべてが完了するまで待たずに、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ので、スループットが増加します。最終的な値は 5 サイクルで出力されるようになり、全体的なレイテンシが短くなります。

Vivado HLS では、関数のデータ フロー パイプライン処理は関数間にチャネルを挿入することにより実行されます。これらのチャネルは、データのプロデューサーおよびコンシューマーのアクセス パターンによって、ピンポン バッファーまたは FIFO としてインプリメントされます。

- 関数パラメーター (プロデューサーまたはコンシューマー) が配列の場合は、該当するチャネルがマルチバッファーとして標準メモリ アクセス (関連のアドレスおよび制御信号を使用) を使用してインプリメントされます。
- スカラー、ポインター、参照パラメーター、および関数の戻り値の場合は、チャネルは FIFO としてインプリメンテーションされます。この場合、アドレス生成は不要なので使用されるハードウェア リソースは少なくなりますが、データに順次アクセスする必要があります。

関数のデータ フロー パイプライン処理を使用するには、データ フロー最適化が必要な部分に `#pragma HLS dataflow` を挿入します。次に、コード例を示します。

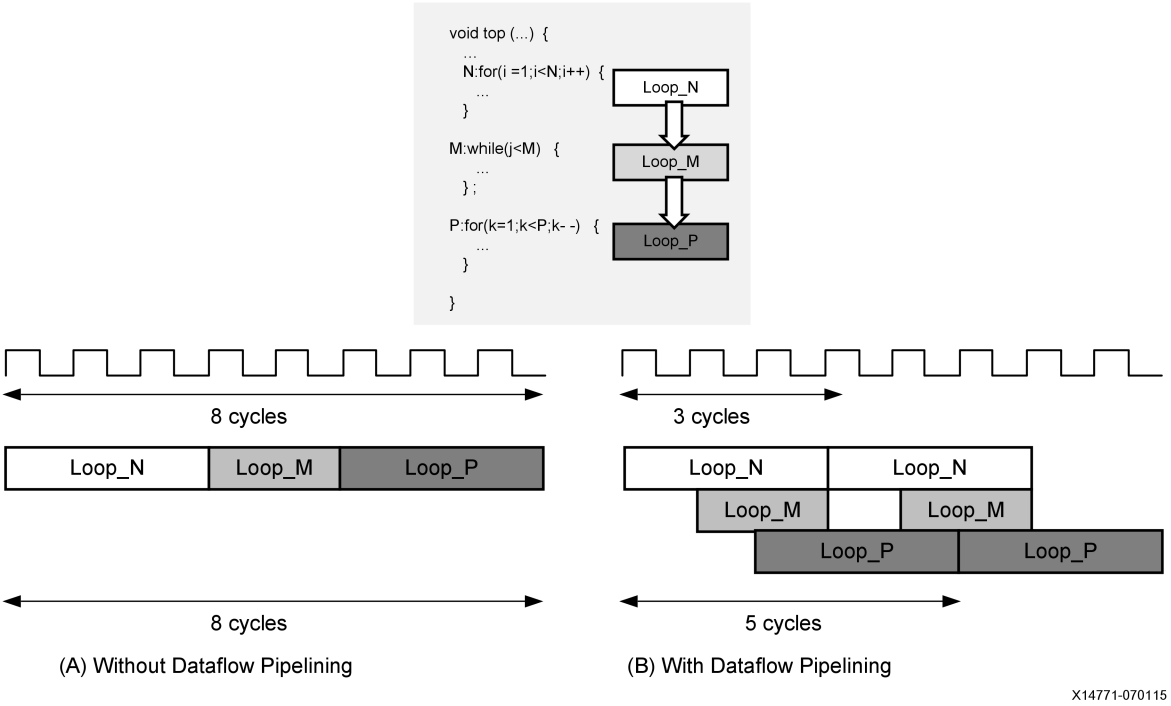
```
void top(a, b, c, d) {
#pragma HLS dataflow
    func_A(a, b, i1);
    func_B(c, i1, i2);
    func_C(i2, d);
}
```

ループのデータ フローのパイプライン処理

データ フロー パイプライン処理は、関数に適用するのと同様の方法でループにも適用できます。これにより、ループのシーケンス (通常は順次処理) がイネーブルになり、同時処理されるようになります。データ フロー パイプライン処理は、関数、ループ、またはすべて関数かすべてループを含む領域に適用する必要があります。ループと関数が混合したスコープに適用しないでください。

データ フロー パイプライン処理をループに適用した場合の利点については、「ループのデータ フロー パイプライン処理」を参照してください。データ フロー パイプライン処理を実行しない場合、ループ M を開始する前にループ N のすべての繰り返しを実行し、完了する必要があります。ループ M とループ P にも同様の関係があります。この例では、ループ N で新しい値を処理できるようになるまでに 8 サイクルかかり、出力が書き込まれる (ループ P が終了したときに出力が書き込まれると想定) までに 8 サイクルかかります。

図 7-7 : ループのデータ フローのパイプライン処理



データ フロー パイプラインを使用すると、これらのループが同時に処理されるようにできます。上記の図の (B) は、データ フロー パイプライン処理を使用した例を示しています。ループ M の実行に 3 サイクルかかるとすると、このコードでは 3 サイクルごとに新しい入力を受信できます。同様に、同じハードウェア リソースを使用して 5 サイクルごとに出力値を生成できます。Vivado HLS ではループ間に自動的にチャネルが挿入され、データが 1 つのループから次のループに非同期に流れるようになります。データ フロー パイプラインを使用した場合と同様、ループ間のチャネルはマルチバッファーか FIFO のいずれかとしてインプリメントされます。

ループのデータ フロー パイプライン処理を使用するには、データ フロー最適化が必要な部分に `#pragma HLS dataflow` を挿入します。

XILINX
ALL PROGRAMMABLE™

第 8 章

C 呼び出し可能なライブラリの使用

IP コアに使用する C 呼び出し可能なライブラリの作成方法については、『SDSoC 環境ユーザー ガイド : プラットフォームおよびライブラリ』(UG1146) の「ライブラリの作成」を参照してください。

C 呼び出し可能なライブラリの使用方法は、ソフトウェア ライブラリの使用方法と同様です。次の例に示すように、該当するソース ファイルでライブラリのヘッダー ファイルを `#include` で含めて、`sdscc -I<path>` オプションを使用してソース コードをコンパイルします。

```
> sdscc -c -I<path to header> -o main.o main.c
```

SDSoC IDE を使用する場合は、プロジェクトを右クリックして [C/C++ Build Settings] → [SDSCC Compiler] → [Directories] (C++ コンパイルの場合は [SDS++ Compiler] → [Directories]) をクリックしてこれらの `sdscc` オプションを追加します。

ライブラリをアプリケーションにリンクするには、-L<path> および -l<lib> オプションを使用します。

```
> sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -l<library_name> -o myApp.elf
```

標準 GNU リンカーを使用する場合と同様、`libMyLib.a` というライブラリの場合は、`-lMyLib` を使用します。

SDSoC IDE を使用する場合は、プロジェクトを右クリックして [C/C++ Build Settings] → [SDS++ Linker] → [Libraries] をクリックしてこれらの `sdscc` オプションを追加します。

C 呼び出し可能なライブラリを使用するコード例は、SDSoC™ 環境インストールの `samples/fir_lib/use` および `samples/rtl_lib/arraycopy/use` ディレクトリの下に含まれます。

XILINX
ALL PROGRAMMABLE™

第 9 章

Vivado Design Suite HLS ライブラリの使用

このセクションでは、SDSoC 環境で Vivado HLS ライブラリを使用する方法について説明します。

Vivado® 高位合成 (HLS) ライブラリは、SDSoC 環境の Vivado HLS インストールにソース コードとして含まれており、Vivado HLS を使用してプログラマブル ロジック向けにクロス コンパイルする予定のほかのソース コード同様にこれらのライブラリを使用できます。「ハードウェア関数の引数型」に説明されている規則にソース コードが従っている必要があります。このとき、関数でソフトウェア インターフェイスがアプリケーションにエクスポートされるようにするため、C/C++ ラッパー関数を作成することが必要な場合があります。

SDSoC IDE に含まれているすべてのベース プラットフォーム対応の FIR サンプル テンプレートに、HLS ライブラリを使用する例が含まれています。samples/hls_lib ディレクトリには、HLS math ライブラリを使用するコード例が数個含まれています。たとえば、samples/hls_lib/hls_math には平方根関数をインプリメントして使用する例が含まれています。

my_sqrt.h ファイルには次が含まれています。

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_

#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

my_sqrt.cpp ファイルには次が含まれています。

```
#include "my_sqrt.h"

void my_sqrt(float x, float *ret)
{
    *ret = sqrtf(x);
}
```

makefile にはこれらのファイルをコンパイルするコマンドが含まれています。

```
sds++ -c -hw my_sqrt -sds-pf zc702 my_sqrt.cpp
sds++ -c my_sqrt_test.cpp
sds++ my_sqrt.o my_sqrt_test.o -o my_sqrt_test.elf
```

XILINX
ALL PROGRAMMABLE™

第 10 章

アプリケーションをライブラリとしてエクスポート

SDSoC 環境でアプリケーションを作成するとき、SDSoC プラットフォームを始点として選択し、このプラットフォーム上のハードウェアにインプリメントする関数を指定します。SDSoC システム コンパイラでハードウェアに選択されている関数およびそれらに対応するデータ ムーバーを含むハードウェア デザインが作成され、これらのアクセラレータおよびデータ ムーバーとの通信に必要なソフトウェアも生成されます。システム コンパイルでは、デフォルトでビットストリーム、ファイル システム、オペレーティング システム、およびアプリケーション実行ファイルを含む完全なブート イメージが出力されます。

SDSoC のシステム コンパイラ オプションは、アプリケーション バイナリの代わりにスタティックまたは共有ライブラリのいずれかを生成するように変更でき、標準の GNU ツールチェーンを使用して残りのアプリケーションを開発するときにこのライブラリにリンクできます。つまり、同じハードウェア システムをターゲットして sdscc で生成されたブート環境を使用しながら、任意のソフトウェア開発環境で GNU ツールチェーンを使用してソフトウェアを開発できます。

ライブラリ フローの使用ケースの 1 つとして、アプリケーションをハードウェア特定の部分と、CPU で完全に実行される sdscc を使用してコンパイルする必要がないソフトウェア部分に分割するケースが挙げられます。ハードウェア アクセラレータを決定し、アプリケーション特定のハードウェア システムをビルドすると、ライブラリにより残りのソフトウェア アプリケーションが ARM ツールチェーンを使用して開発できるようになり、ソフトウェアを高速にコンパイルできます。

共有ライブラリへのエントリ ポイントは、システム コンパイル中に sdscc で生成される特定のスタブ関数で、これらをライブラリのヘッダー ファイルで宣言します。

重要： 共有ライブラリでは同じハードウェアが使用されるので、ライブラリへのエントリ ポイントと生成システム間で、メモリ割り当て、ハードウェア関数間のハードウェア上での接続性、およびデータ ムーバーの想定がすべて一貫していることを確実にする必要があります。アプリケーション コード全体が共有ライブラリとリンクされている場合は、システム生成中に sdscc による想定との一貫性のチェック (ハードウェア関数に渡されるバッファーのメモリ割り当てなど) は実行されません。ハードウェア内で直接接続されている複数のハードウェア関数の接続コンポーネントは 1 つの関数にラップし、この関数により個々のハードウェア関数へのアクセスを制御するようにすることを推奨します。

sdscc でシステムが生成されるときに、ハードウェア関数それぞれのエントリ ポイントを含むスタティック ライブラリが自動的に作成されます。SDSoC IDE で myApp という名前のプロジェクトを作成すると、ライブラリは <build_configuration>/ _sds/swstubs/libmyApp.a になります。コマンド ライン インターフェイスを使用して myApp.elf をビルドすると、ライブラリは _sds/swstubs/libmyApp.a になります。

スタブ関数のライブラリへのエントリ ポイントは、一連のハードウェア関数のプロトタイプとは完全には一致していません。sdscc コンパイラでは、複数のビットストリーム (つまり #pragma SDS partition を使用したパーティション) および複数のハードウェア関数インスタンスをサポートするため、ハードウェア関数の名前が自動的に分割された名前に変更されます。通常は、このような名前の変更に注意する必要はありませんが、デザインをライブラリとしてエクスポートするときには、ライブラリのヘッダー ファイルでスタブ関数を宣言する必要があります。

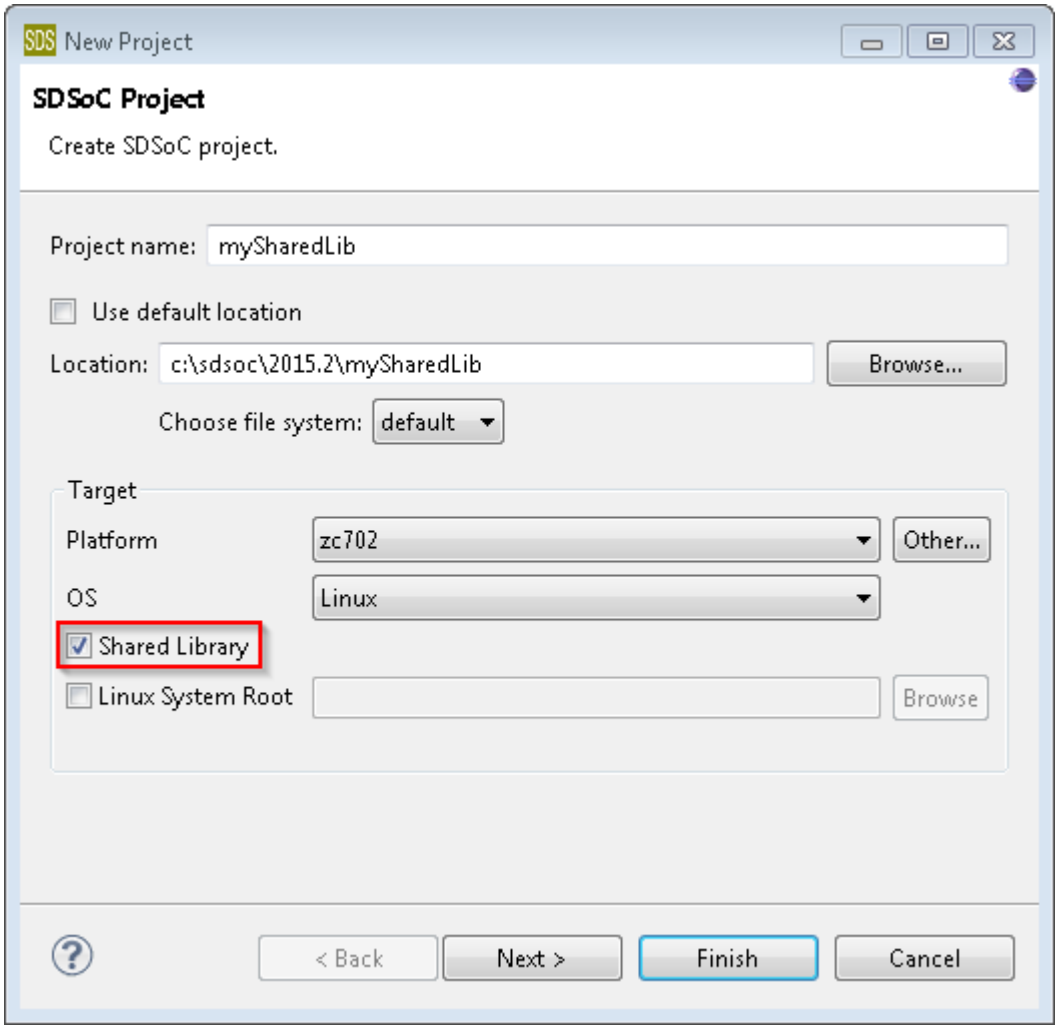
たとえば、ハードウェア関数 mmult_accel には、通常次の宣言が含まれています。

```
// mmult_accel.h
void _p0_mmult_accel_0(float[], float[], float[]);  // hardware function mmult_accel
```

どのハードウェア関数でも、エントリ ポイントは [Project Explorer] タブでライブラリを展開表示してライブラリ内の関数を調べることで簡単に判断できます。

完全なサンプルは、SDSoC 環境インストールの samples/mmult_static_lib/build ディレクトリに含まれています。

SDSoC IDEで共有ライブラリを作成するには、SDSoC 環境のプロジェクトを作成するときに [Shared Library] をオンにします。



共有ライブラリ libmySharedLib.so は、SD カードのブート イメージと共に作成されます。コマンド ラインを使用してデザインを共有ライブラリとしてエクスポートするには、ハードウェア関数および sdscc/sds++ の位置独立コード フラグ (-fPIC) を使用してこれらの関数を呼び出す関数を含むソース ファイルをコンパイルし、-shared オプションを使用してリンクします。

SDSoC IDE では、[Shared Library] をオンにすると、行列乗算共有ライブラリのサンプル テンプレートが提供されます。ハードウェア ブロックの接続性は、ライブラリを呼び出す方法が定義されているプロセス関数を含むソース ファイルを使用して決定されます。SDSoC システム コンパイラでは、この機能に基づいてシステムの接続性が決定されます。

```
File: mmult_call.c
#include "mmult_accel.h"

void mmult_call (float in_A[A_NROWS*A_NCOLS],
                 float in_B[A_NCOLS*B_NCOLS],
                 float out_C[A_NROWS*B_NCOLS])
{
    mmult_accel(in_A, in_B, out_C);
}
```

この例では、ハードウェア インプリメンテーションに選択されている関数 mmult_accel への呼び出しは 1 つですが、ライブラリに複数のハードウェア関数を指定できます。

ハードウェア関数は、sdscc をオブジェクト コードの位置を独立させる -fPIC フラグと共に使用してコンパイルされます。

```
sdscc -sds-pf zc702 -sds-hw mmult_accel mmult_accel.cpp -sds-end \
-c -fPIC mmult_accel.c -o mmult_accel.o
```

呼び出し関数コードも -fPIC フラグを使用してコンパイルする必要があります。

```
sdscc -sds-pf zc702 -c -fPIC mmult_call.c -o mmult_call.o
```

最後に、すべてのオブジェクト ファイルをリンクして共有ライブラリ オプションを指定します。

```
sdscc -sds-pf zc702 -shared mmult_accel.o mmult_call.o -o libmmult_accel.so
```

これで libmmult_accel.so ライブラリが作成されます。このライブラリは任意のソフトウェア開発環境またはコマンド ラインで標準の ARM GNU ツールチェーンを使用してリンクできます。

またこのコマンドでは、ライブラリにリンクするプログラムを実行するときに必要なブート ファイルを含む sd_card イメージも作成されます。

完全なサンプルは、SDSoC 環境インストールの samples/mmult_shared_lib/build ディレクトリに含まれています。

アプリケーション ライブラリへのリンク

SDSoC 環境のアプリケーション用に生成されるライブラリは、ほかのソフトウェア ライブラリと同様にリンクされます。#include を使用してライブラリに関連付けられているヘッダー ファイルをソース ファイルに含め、GCC -I オプションを使用してヘッダー ファイルへのディレクトリ パスを指定してコンパイルします。GCC -L オプションを使用してライブラリへのパスを指定し、-l オプションを使用してライブラリ名を宣言して、アプリケーションをリンクします。

たとえば、main 関数と共有ライブラリ内の mmult_accel 関数を呼び出す mmult.cpp という名前のファイルを作成したとします。このファイルをコンパイルするには、次のコマンドを使用します。

```
arm-xilinx-linux-gnueabi-g++ -c -O3 mmult.cpp -o mmult.o
```

アプリケーションをリンクするには、次のコマンドを使用します。

```
arm-xilinx-linux-gnueabi-g++ -O3 mmult.o -L./lib -lmmult_accel -lpthread \
-o mmult.elf
```

これにより mmult.elf 実行ファイルが作成され、これをブート ファイルと共に SD カードにコピーします。POSIX スレッド (pthread) ライブラリは、sdscc で生成されるソフトウェア ランタイム コードに必要です。

プログラムを実行するには、SDSoC 環境で作成された sd_card ディレクトリを SD カードにコピーし、ボードを起動して、コマンド プロンプトが表示されるのを待ちます。次のコマンドをボードで実行します。

```
sh-4.3# export LD_LIBRARY_PATH=/mnt
sh-4.3# /mnt/mmult.elf
```

完全なサンプルは、SDSoC 環境インストールの samples/mmult_shared_lib/use および samples/mmult_static_lib/use ディレクトリに含まれています。

XILINX
ALL PROGRAMMABLE™

第 11 章

アプリケーションのデバッグ

SDSoC™ 環境を使用すると、SDSoC IDE を使用してプロジェクトを作成およびデバッグできます。プロジェクトは、ユーザー定義の makefile を使用して SDSoC IDE 外で作成することも可能で、コマンド ラインまたは SDSoC IDE のいずれでもでデバッグできます。

SDSoC IDE でのインタラクティブなデバッガーの使用については、『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) の「チュートリアル : システムのデバッグ」を参照してください。

SDSoC IDE での Linux アプリケーションのデバッグ

SDSoC™ IDE でアプリケーションをデバッグするには、次の手順を使用します。

1. [SDDebug] をアクティブ ビルド コンフィギュレーションとして選択し、プロジェクトをビルドします。
2. 生成した `SDDebug/sd_card` イメージを SD カードにコピーして、それを使用してボードを起動します。
3. ボードがネットワークに接続されていることを確認し、コマンド プロンプトで `ifconfig eth0` を実行するなどして、IP アドレスをメモしておきます。
4. [Debug As] をクリックして新しいデバッグ コンフィギュレーションを作成し、ボードの IP アドレスを入力します。
5. [Debug] パースペクティブに切り替えて、デバッグを開始、停止、ステップ実行、ブレークポイントの設定、変数およびメモリの検証、または別のデバッグ操作を実行できます。

SDSoC IDE でのスタンドアロン アプリケーションのデバッグ

SDSoC™ IDE を使用してスタンドアロン (ベアメタル) アプリケーション プロジェクトをデバッグするには、次の手順に従います。

1. [SDDebug] をアクティブ ビルド コンフィギュレーションとして選択し、プロジェクトをビルドします。
2. ボードが JTAG デバッグ コネクタを使用してホスト コンピュータに接続されていることを確認します。
3. [Debug As] をクリックして、新しいデバッグ コンフィギュレーションを作成します。

   [Debug] パースペクティブに切り替えて、デバッグを開始、停止、ステップ実行、ブレークポイントの設定、変数およびメモリの検証、または別のデバッグ操作を実行できます。

   [SDSoC Project Overview] で [Debug Application] リンクをクリックすると、上記の手順を一気に実行できます。

FreeRTOS アプリケーションのデバッグ

SDSoC™ 環境で FreeRTOS アプリケーション プロジェクトを作成する場合は、スタンドアロン (ベアメタル) アプリケーション プロジェクトと同様の手順に従いアプリケーションをデバッグできます。

IP レジスタの監視および変更

mrd および mwr という 2 つの小さな実行ファイルを使用すると、メモリ マップされたプログラマブル ロジックのレジスタを監視および変更できます。これらの実行ファイルは、アクセスする物理アドレスを指定して実行します。

たとえば mrd 0x80000000 10 は、物理アドレス 0x80000000 で開始する 10 個の 4 バイト値を読み出し、それらを標準出力に表示します。mwr 0x80000000 20 は値 20 をアドレス 0x8000000 に書き込みます。

これらの実行ファイルは、ハードウェア関数および SDSoC™ 環境で生成されたその他の IP に含まれるメモリ マップされたレジスタのステートを監視および変更するために使用できます。

注意 : 存在しないアドレスにアクセスしようとすると、システムが停止する可能性があります。

パフォーマンスをデバッグする際のヒント

SDSoC 環境では、sds_clock_counter() 関数により基本的なパフォーマンス監視機能が提供されています。この関数を使用すると、アクセラレーションされるコードとされないコードなど、コード セクション間における実行時間の差異を調べることができます。

Vivado® HLS レポート ファイル (_sds/vhls/…/*.rpt) でレイテンシ数を見ると、実際のハードウェア アクセラレーション時間を予測できます。X アクセラレータのクロック サイクルのレイテンシは、X * (processor_clock_freq/accelerator_clock_freq) プロセッサ クロック サイクルです。実際の関数呼び出しにかかる時間とこの時間を比較すると、データ転送のオーバーヘッドを確認できます。

パフォーマンスを最大限に改善するには、アクセラレーションされる関数の実行に必要な時間が元のソフトウェア関数の実行に必要な時間よりもかなり短くなることが必要です。そうならない場合は、sdscc/sds++ コマンドラインで別の clkid を選択して、アクセラレータをより高速の周波数で実行してみてください。この方法で改善が見られない場合は、データ転送のオーバーヘッドがアクセラレーションされる関数の実行時間に影響していないかを確認し、このオーバーヘッドを減らす必要があります。デフォルトの clkid はすべてのプラットフォームで 100MHz です。特定のプラットフォームの clkid 値の詳細は、sdscc -sds-pf-info <platform name> を実行すると取得できます。

データ転送のオーバーヘッドが大きい場合は、次を変更すると改善される可能性があります。

- より多くのコードをアクセラレーションされる関数に移動して、この関数の計算にかかる時間を長くし、データ転送にかかる時間との比率を向上させます。
- コードを変更するかプラグマを使用して必要なデータのみを転送するようにし、転送するデータ量を減らします。

XILINX
ALL PROGRAMMABLE™

第 12 章

AXI Performance Monitor を使用したパフォーマンス計測

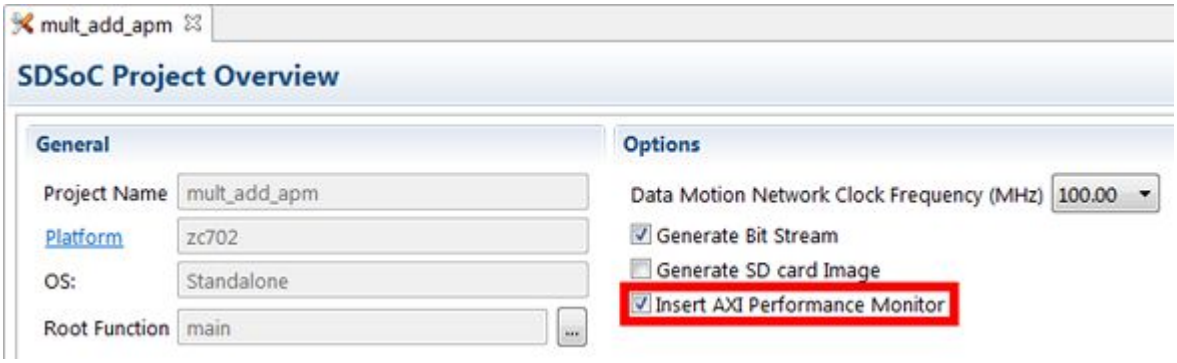
AXI Performance Monitor (APM) モジュールは、プロセッシング システム (PS) 内の ARM コアとプログラマブル ロジック (PL) 内のハードウェアの間のデータ転送に関する基本的な情報を監視するために使用します。読み出し/書き込みトランザクション数、システムのバス上の AXI トランザクション レイテンシなどの統計を収集します。

このセクションでは、システムへの APM コアの挿入方法、計測用に設定されたシステムの監視方法、および生成されたパフォーマンス データの表示方法を示します。

プロジェクトの作成と APM のインプリメント

SDSoC 環境を開き、任意のプラットフォームまたはオペレーティング システムを使用して新しい SDSoC プロジェクトを作成します。[Matrix Multiplication and Addition] テンプレートを選択します。

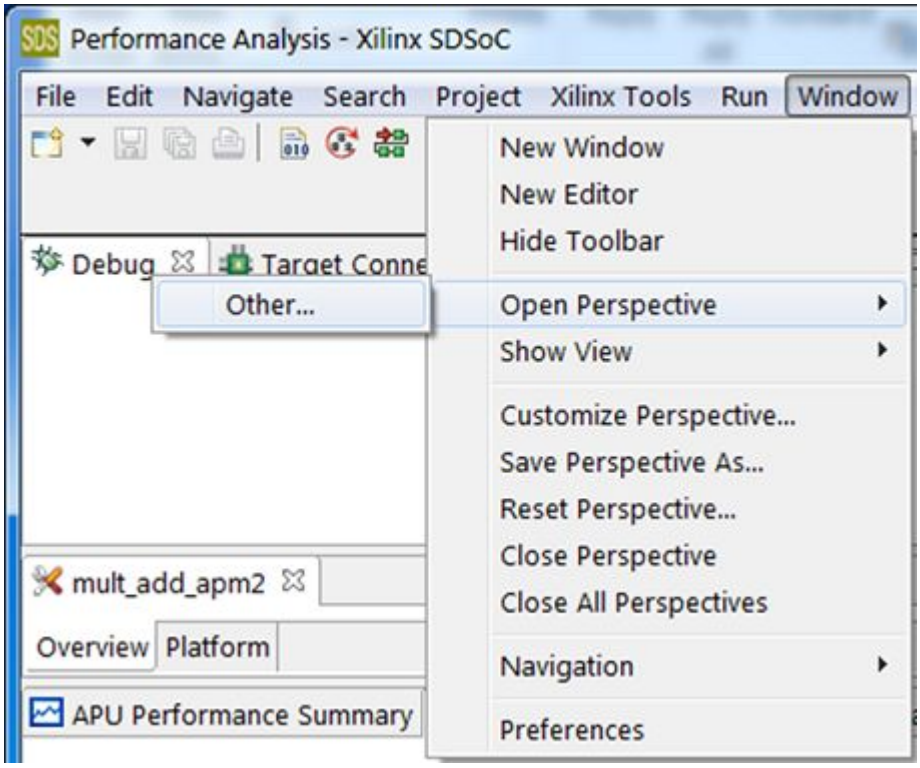
[SDSoC Project Overview] で [Insert AXI Performance Monitor] をオンにします。このオプションをオンにしてプロジェクトを作成すると、ハードウェア システムに APM IP コアが追加されます。APM IP は、プログラマブル ロジックの少量のリソースを使用します。SDSoC により、APM がハードウェア/ソフトウェア インターフェイス ポートであるアクセラレータ コヒーレンシ ポート (ACP)、汎用ポート (GP)、ハイ パフォーマンス ポート (HP) に接続されます。



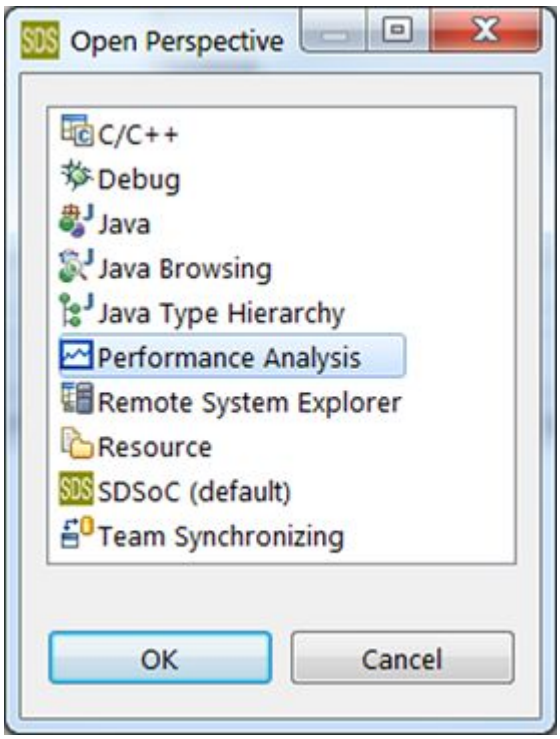
`mmult` および `madd` 関数をハードウェアでインプリメントされるように選択します。SDDebug コンフィギュレーション (デフォルト) を使用してプロジェクトをクリーンアップおよび構築します。

計測用に設定されたシステムの監視

ビルドが完了したら、ボードをコンピューターに接続してボードの電源を入れます。[Debug] ボタンをクリックしてターゲット上のアプリケーションを起動します。[Debug] パースペクティブに切り替えます。PL がプログラムされて ELF が起動した後、プログラムが main で停止します。[Window] → [Open Perspective] → [Other] をクリックします。

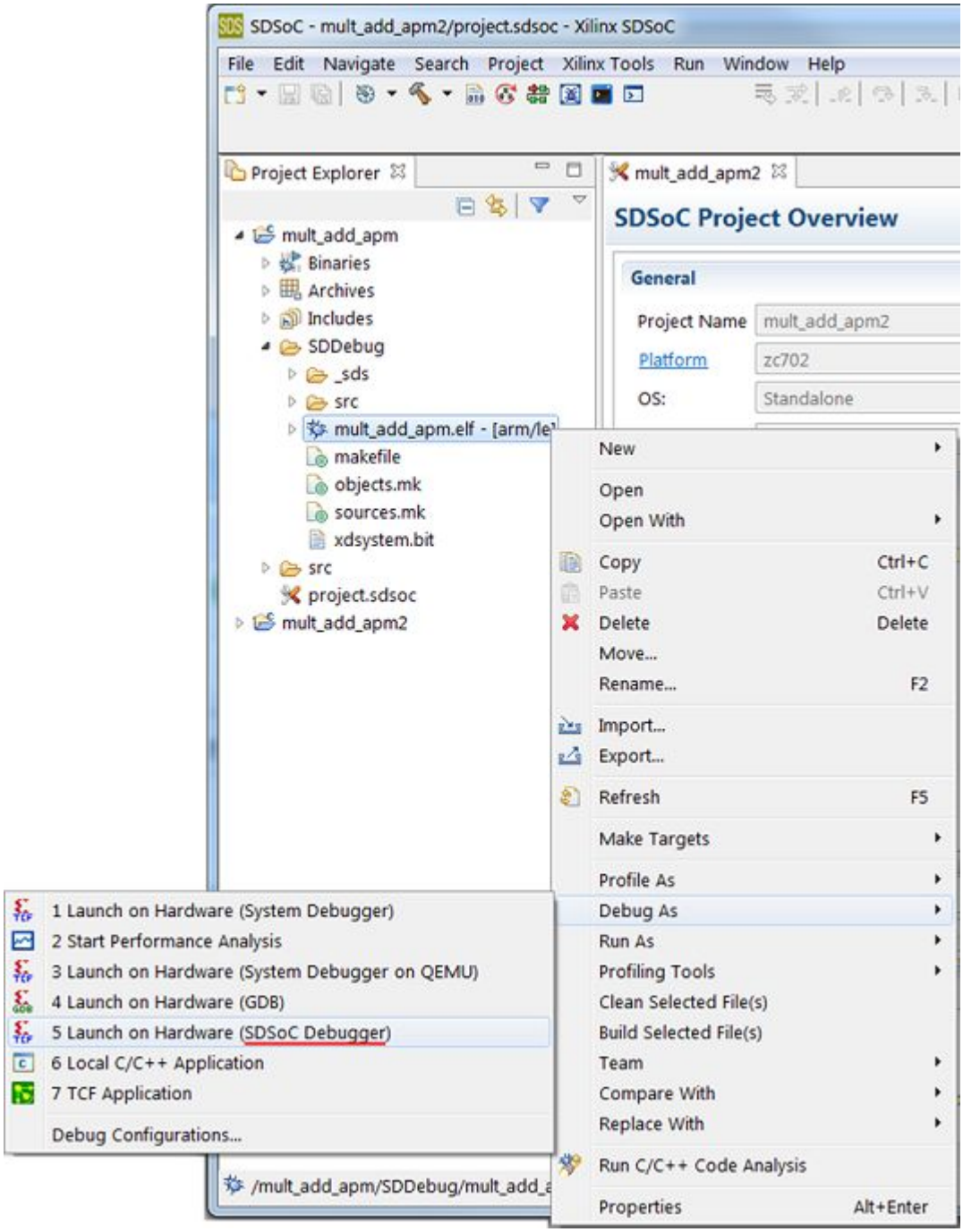


[Open Perspective] ダイアログ ボックスで [Performance Analysis] を選択し、[OK] をクリックします。



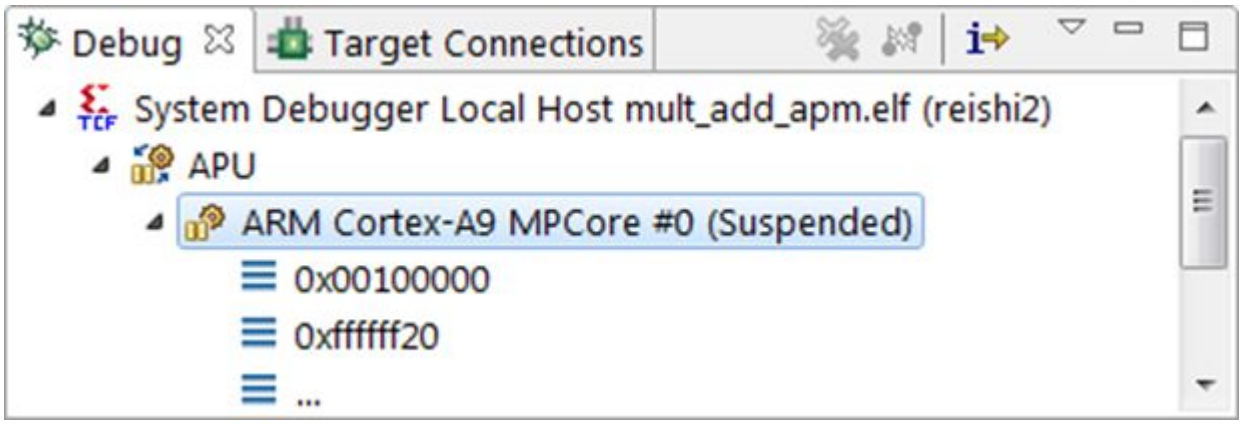
[SDSoC] パースペクティブに戻します。

[Project Explorer] タブの [SDDebug] フォルダーを展開表示します。ELF 実行ファイルを右クリックし、[Debug As] → [Launch on Hardware (SDSoC Debugger)] をクリックします。アプリケーションを再起動するようメッセージが表示されたら、[OK] をクリックします。

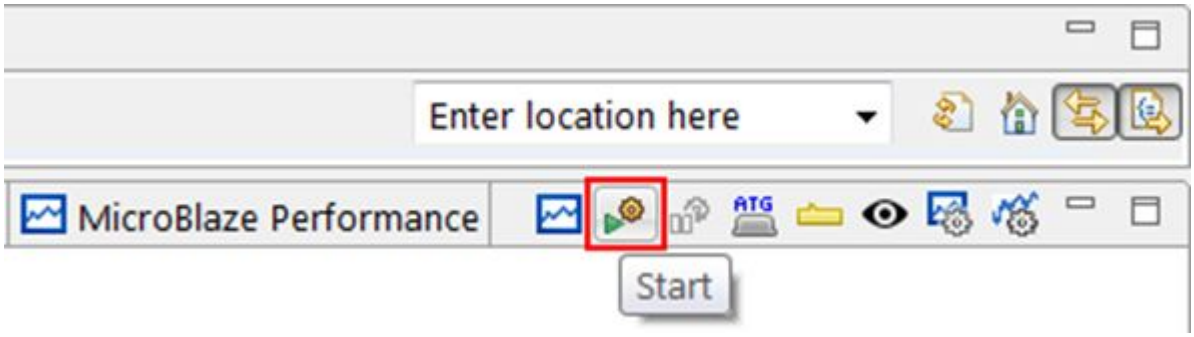


[Yes] をクリックして [Debug] パースペクティブに戻します。アプリケーションが起動して main 関数のブレークポイントで停止したら、[Performance Analysis] パフォーマンスに戻します。

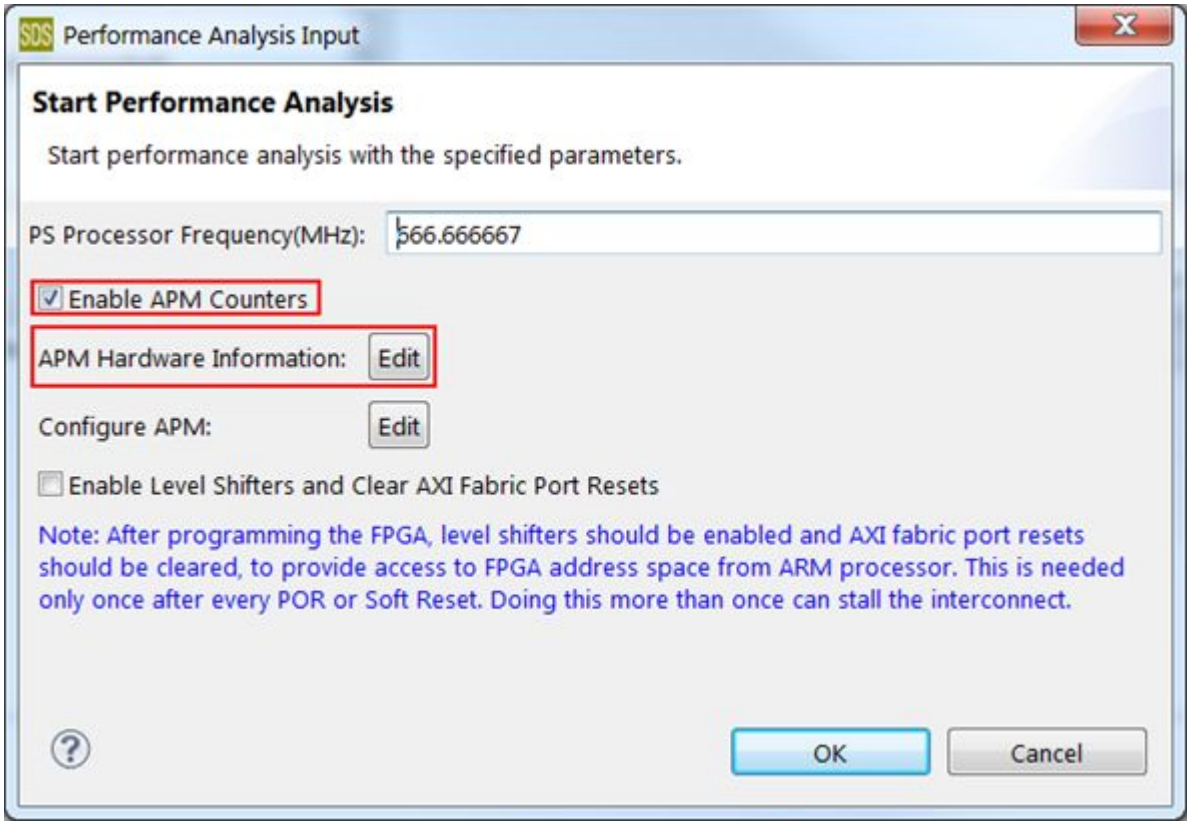
パースペクティブの左上の [Debug] タブで、[ARM Cortex-A9 MPCore #0] をクリックします。



[Start] ボタンをクリックします。[Performance Analysis Input] ダイアログ ボックスが開きます。

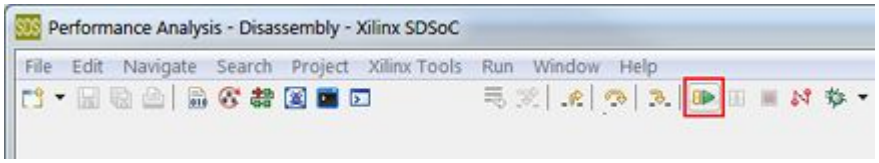


[Enable APM Counters] をオンにします。[APM Hardware Information] の右側にある [Edit] ボタンをクリックします。

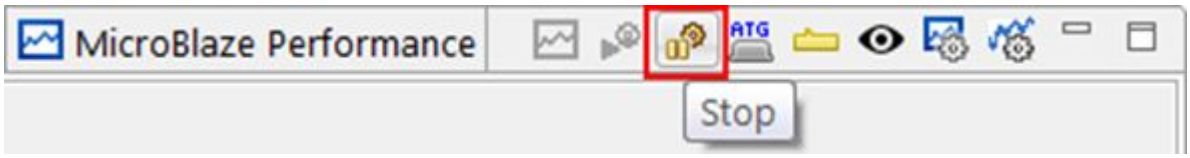


[APM Hardware Information] ダイアログ ボックスで [Load] ボタンをクリックします。`workspace_path/project/SDDebug/_sds/p0/ipi/zc702.sdk` に移動して `zc702.hdf` ファイルを選択します (この例では zc702 をプラットフォームとして使用していますが、ご使用のプラットフォームを指定してください)。[Open] をクリックし、[APM Hardware Information] ダイアログ ボックスで [OK] をクリックします。最後に [Performance Analysis Input] ダイアログ ボックスで [OK] をクリックします。

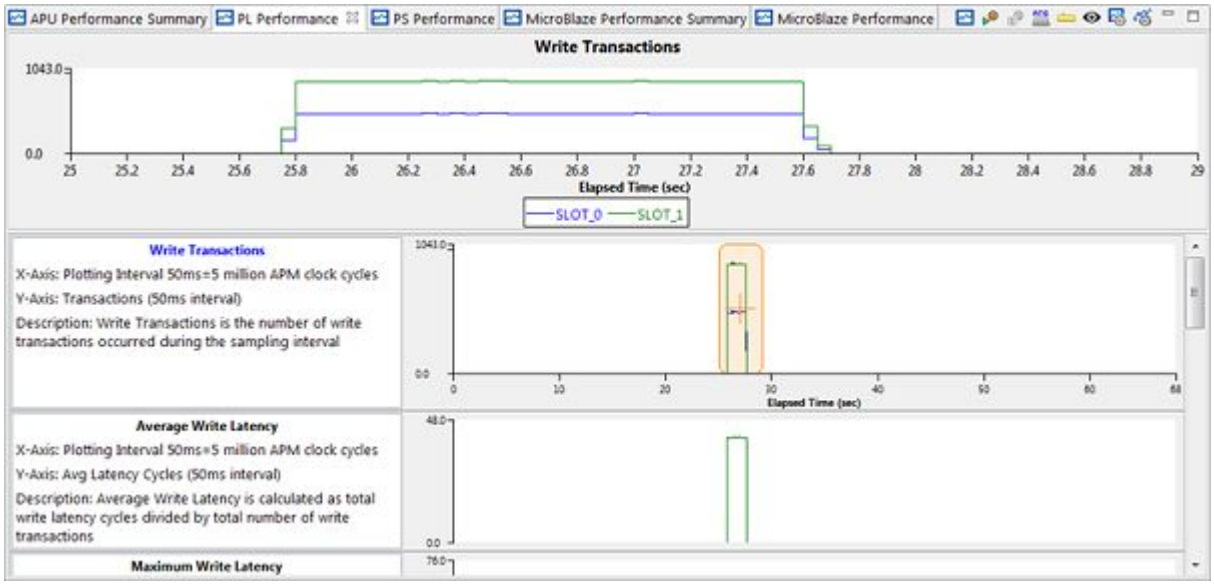
[PL Performance] タブに [Analysis] ビューが開きます。[Resume] をクリックしてアプリケーションを実行します。



プログラムの実行が終了したら [Stop] ボタンをクリックします。[Confirm Perspective Switch] ダイアログ ボックスに [Performance Analysis] パースペクティブにとどまるかどうかを確認するメッセージが表示されたら、[No] をクリックします。



パースペクティブの下部で解析プロットをスクロールし、異なるパフォーマンス統計を確認します。プロット エリアをクリックすると、パースペクティブの中央に拡大表示されます。下のオレンジ色のボックスにより、データの特定の時間範囲に焦点を置くことができます。



パフォーマンスの解析

このシステムでは、APM は PS と PL の間で使用されている 2 つのポート、アクセラレータ コヒーレンシ ポート (ACP) および汎用 AXI ポート (GP) に接続されます。乗算器および加算器アクセラレータ コアは、両方ともデータの入出力用に ACP に接続されます。GP ポートは、制御コマンドの発行およびアクセラレータ コアのステータスを取得するためにのみ使用され、データ転送には使用されません。青色のスロット 0 は GP ポートに接続され、緑色のスロット 1 は ACP に接続されています。

APM は、ACP および GP ポートにそれぞれ 1 つ、2 つの監視スロットと共に、プロファイル モードでコンフィギュレーションされます。プロファイル モードでは、各スロットにイベント カウント機能が含まれます。読み出しおよび書き込みに対して APM で算出される統計のタイプには、次のものが含まれます。

- トランザクション数：バス上で発生する要求の総数
- バイト数：送信されたバイトの総数 (書き込みスループットの算出に使用)
- レイテンシ：アドレス発行の開始から最後の要素が送信されるまでの時間

レイテンシおよびバイト カウンターの統計は、スループット (MB/s) を自動的に算出するために APM で使用されます。表示されるレイテンシおよびスループット値は、50 ミリ秒 (ms) 間隔です。最小、最大、平均も表示されます。

XILINX
ALL PROGRAMMABLE™

第 13 章

ターゲット オペレーティング システム サポート

SDSoC 環境では、現在のところ Linux、スタンドアロン (ベアメタル)、および FreeRTOS ターゲット オペレーティング システムがサポートされています。SDSoC システム コンパイラではターゲット OS 特定の特性評価データが使用されて各アプリケーションのデータ ムーバーが選択されるので、同じアプリケーション コードでも別のオペレーティング システム下で実行されると、別のハードウェア システムが生成されることがあります。

Linux アプリケーション

SDSoC™ 環境では、Zynq® デバイスで実行される Linux アプリケーションがサポートされるので、Linux オペレーティング システムを使用してハードウェアで実行されるプラグマをコンパイルできるようになっています。SDSoC 環境では、ライブラリにリンクすることで、オペレーティング システムで提供されるサービスを使用してハードウェアと通信できます。

使用方法

SDSoC 環境プログラムを Linux 向けにコンパイルおよびリンクするには、makefile の CFLAGS および LFLAGS に `-target-os linux` を含める必要があります。`-target-os linux` オプションが含まれていない場合、SDSoC™ 環境ではデフォルトで Linux オペレーティング システムがターゲットにされます。

SD ブート イメージは、`sd_card` ディレクトリの複数ファイルで構成されます。`BOOT.BIN` には、ボードへの電源投入後に直接起動されて、U-boot を起動する FSBL (First Stage Boot Loader) が含まれます。Linux ブートでは、デバイス ツリー、Linux カーネル、ramdisk イメージが使用されます。SD ブート イメージには、プログラマブル ロジックのコンフィギュレーションに使用されたアプリケーション ELF とハードウェア ビットストリームも含まれます。

サポートされるプラットフォーム

Linux は、すべての SDSoC™ プラットフォームでサポートされます。

制限

提供されている Linux オペレーティング システムでは、ビルド済みカーネル イメージ (3.19, Xilinx branch xilinx-v2015.2.03) と BusyBox を含む ramdisk が使用されます。Linux イメージと ramdisk イメージをユーザーのプラットフォームまたは条件に合わせてコンフィギュレーションするには、wiki.xilinx.com の方法に従って、Linux カーネルをダウンロードおよびビルドしてください。Linux ブート ファイルの詳細と Petalinux を使用してそれを作成す方法については、『SDSoC 環境ユーザー ガイド : プラットフォームおよびライブラリ』(UG1146) の「Linux ブート ファイル」を参照してください。

スタンドアロン ターゲット アプリケーション

SDSoC™ 環境では、Zynq® デバイスで実行される Linux アプリケーションに加え、プログラムをオペレーティング システムなしでハードウェア上で直接実行するようコンパイルできるスタンドアロン モードがサポートされています。SDSoC 環境では、通常はターゲット オペレーティング システムで実行されるサービスを実行するライブラリにリンクされます。

使用方法

SDSoC 環境プログラムをスタンドアロン モード向けにコンパイルおよびリンクするには、makefile の CFLAGS および LFLAGS に `-target-os standalone` を含める必要があります。

SD のブート イメージは、`sd_card` ディレクトリに含まれる `BOOT.BIN` ファイル 1 つで構成されます。このファイルには、FSBL (First Stage Boot Loader) とボードに電源が投入された直後に実行されるユーザー アプリケーションが含まれています。

サポートされるプラットフォーム

スタンドアロン モードは、次のプラットフォームでサポートされています。

- zc702 (ザイリンクス ZC702 評価ボードに基づく)
- zc706 (ザイリンクス ZC706 評価ボードに基づく)
- zed (zedboard.org からの ZedBoard に基づく)
- microzed (zedboard.org からの MicroZed ボードに基づく)
- zybo (の Zybo ボードに基づく)

制限

スタンドアロン モードでは、『OS およびライブラリ資料コレクション』(UG643) に示されているように、マルチスレッド、仮想メモリ、アドレス保護はサポートされません。ファイル システムへのアクセスは、通常の C API ではなく、libxilffs を使用する特別の API で実行されます。この使用例が、サンプル プログラム file_io_manr_sobel_standalone に示されています。このプログラムを Linux バージョン file_io_manr_sobel と比較すると、ファイル システムにアクセスするために何を変更する必要があるかがわかります。通常、ファイル システムにアクセスする手順は、いくつかの追加ファイルを含め、異なるタイプを使用し (FILE ではなく FIL を使用するなど)、ファイル システムへのアクセスに少し異なる API を使用し (fopen ではなく f_open を使用するなど)、ファイル操作を実行する前に DCache をディスエーブルにします。

重要 : ZedBoard では、ボードへの直列接続に 2、3 秒かかります。プログラムがそれより短い時間実行される場合、その出力は得られません。ZedBoard の電源を切って入れ直した場合、直列接続はオフになり、その後の実行でも出力は得られません。ZC702 および ZC706 ボードでは、電源を切って入れ直した場合にも直列接続はアクティブな状態に保持されるので、このような制限はありません。

FreeRTOS ターゲット アプリケーション

SDSoC™ 環境では、Zynq®-7000 AP SoC デバイスで実行される Linux アプリケーションに加え、Real Time Engineers Ltd (www.freertos.org) の FreeRTOS を使用するアプリケーションもサポートされており、スケジューリング、タスク間通信、タイミングおよび同期化を実行する API を使用したリアルタイム カーネルと共にプログラムをコンパイルできます。

SDSoC 環境には、FreeRTOS v8.2.1 ヘッダー ファイルおよびリアルタイム カーネル、API 関数、および Zynq デバイス特有のプラットフォーム コードを含むコンフィギュレーション済みライブラリが含まれています。また、C/C++ ベアメタル アプリケーションをサポートするのに必要なドライバーおよび関数を含むスタンドアロン ライブラリもビルドされます。

使用法

SDSoC™ 環境プログラムを FreeRTOS 向けにコンパイルおよびリンクするには、makefile のすべてのコンパイラおよびリンカーに -target-os freertos を含める必要があります。これは、通常 SDSoC 環境変数で指定され、次の図のようにコンパイラ ツールチェーン変数に含まれます。

```
SDSFLAGS = -sds-pf zc702 -target-os freertos \
 -sds-hw mmult_accel mmult_accel.cpp -sds-end \
 -poll-mode 1
CPP = sds++ ${SDSFLAGS}
CC = sds ${SDSFLAGS}
:
all: ${EXECUTABLE}
${EXECUTABLE}: ${OBJECTS}
 ${CPP} ${LFLAGS} ${OBJECTS} -o $@
%.o: %.cpp
 ${CPP} ${CFLAGS} $< -o $@
:
```

SDSoC 環境はアプリケーション ELF ファイルをリンクする際に、スタンドアロン (ベアメタル) ライブラリをビルドし、定義済みリンカー スクリプトを提供し、ヘッダーとプリビルド ライブラリを使用してコンフィギュレーション済み FreeRTOS カーネルを使用し、ARM GNU ツールチェーンが呼び出される際にそれらのパスを含めます (makefile にパスを指定する必要はありません)。

```
<path_to_install>/SDSoC/2015.4/arm-xilinx-eabi/include/freertos
<path_to_install>/SDSoC/2015.4/arm-xilinx-eabi/lib/freertos
```

SD のブート イメージは、sd_card ディレクトリに含まれる BOOT.BIN ファイル 1 つで構成されます。このファイルには、FSBL (First Stage Boot Loader) とボードに電源が投入された直後に実行されるユーザー アプリケーションが含まれています。

FreeRTOS アプリケーションを開発する際の SDSoC 環境の GUI フローは、スタンドアロン (ベアメタル) アプリケーションの場合と同じですが、ターゲット OS が FreeRTOS と指定されている点だけが違います。ユーザー アプリケーション コードには、次を含める必要があります。

- ハードウェア コンフィギュレーション関数
- xTaskCreate() API 関数を使用したタスク関数とタスク作成呼び出し
- vTaskStartScheduler() API 関数を使用したスケジューラー開始呼び出し
- vApplicationMallocFailedHook()、vApplicationStackOverflowHook()、vApplicationIdelHook()、vAssertCalled()、vApplicationTickHook()、および vInitialiseTimerForRunTimeStats などのコールバック関数

FreeRTOS v8.2.1 ソフトウェア ディストリビューションに含まれる Zynq®-7000 AP SoC シリーズ デモに基づいた単純な SDSoC 環境アプリケーションは、SDSoC GUI アプリケーション ウィザードおよび SDSoC 環境のインストールから入手できます。

```
<path_to_install>/SDSoC/2015.4/samples/mmult_datasize_freertos
<path_to_install>/SDSoC/2015.4/samples/mmult_optimized_sds_freerttos
```

スタンドアロン BSP を通常ターゲットとするユーザー アプリケーションまたはサンプル アプリケーションは、コンパイラおよびリンクで -target-os freertos オプションを使用してビルドできますが、FreeRTOS リンカー スクリプトが使用され、プリビルド FreeRTOS ライブラリに含まれる定義済みコールバック関数が使用されます。この方法でビルドされたアプリケーションは FreeRTOS API 関数を明示的に呼び出し、スタンドアロン アプリケーションとして実行します。このように FreeRTOS アプリケーションの開発を開始することは可能ですが、FreeRTOS API 関数とコールバックはできるだけ早期段階で組み込むことをお勧めします。

サポートされるプラットフォーム

FreeRTOS は次の2 つの Zynq®-7000 AP SoC プラットフォームでサポートされます。

- ZC702
- ZC706

制限およびインプリメンテーションの注意事項

SDSoC 環境の FreeRTOS では、スタンドアロン ボード サポート パッケージ (BSP) ライブラリがサポートされ、スタンドアロン モードと同じ制限があります。

パフォーマンス予測フローでは、FreeRTOS アプリケーションにより、データを収集してハードウェア パフォーマンス データと統合してレポートを作成するために必要なソフトウェア ランタイム データが収集されます。

SDSoC 環境では、ユーザーのためにプリビルドのコンフィギュレーション済み FreeRTOS v8.2.1 ライブラリが使用され、アプリケーション リンク時にスタンドアロン ライブラリがダイナミックにビルドされます。FreeRTOS ライブラリには、次のような特徴があります。

- プラットフォームに依存しないコードには、標準 FreeRTOS v8.2.1 ディストリビューションを使用してください。プラットフォームに依存するコードの場合は、FreeRTOS v8.2.1 (FreeRTOS リファレンス http://www.freertos.org/a00110.html、ダウンロード先 http://sourceforge.net/projects/freertos/files/FreeRTOS) に含まれるデフォルトの `FreeRTOSConfig.h` を使用します。
- メモリ割り当てのインプリメンテーションには `heap_3.c` を使用してください (FreeRTOS リファレンス http://www.freertos.org/a00111.html)。
- 次の FreeRTOS v8.2.1 ディストリビューション フォルダーからソースを使用してください。
  - `Demo/CORTEX_A9_Zynq_ZC702/RTOSDemo/src`
  - `Source`
  - `Source/include`
  - `Source/portable/GCC/ARM_CA9`
  - `Source/portable/MemMang`
- `<path_to_install>/SDSoC/2015.4 /platforms/<platform>/freertos/lscript.ld` のリンカー スクリプトを使用します (一時的にこのファイルを変更したバージョンを代わりに使用するには、ファイルをコピーしてから、リンカー オプションの `-Wl,-T -Wl,<path_to_your_linker_script>` を ELF ファイルを作成するのに使用した sdscc/sds++ コマンド ラインに追加します)。
- Zynq ZC702 のポート記述 (http://www.freertos.org/RTOS-Xilinx-Zynq.html) に基づいており、ユーザー アプリケーション コードではなく、プリビルド ライブラリの一部として置換関数の `memcpy()`、`memset()`、および `memcmp()` が含まれています。ザイリンクス® SDK-ベース BSP パッケージは使用しないでください。
- スタンドアロン アプリケーションが sdscc/sds++ `-target-os freertos` オプションとリンクするようにするための定義済みのコールバック関数を含みます。アプリケーションの一部として、これらの関数のユーザー独自のバージョンを定義しておくことをお勧めします。
  - `vApplicationMallocFailedHook`
  - `vApplicationStackOverflowHook`
  - `vApplicationIdleHook`
  - `vAssertCalled`
  - `vApplicationTickHook`
  - `vInitialiseTimerForRunTimeStats`

XILINX
ALL PROGRAMMABLE™

第 14 章

代表的なサンプル デザイン

SDSoC IDE 内でベース プラットフォームの 1 つに対して新しい SDSoC 環境プロジェクトを作成する際は、オプションで複数の代表的なデザインのいずれかを選択できます。

- 「ファイル I/O ビデオ」: 単純なファイル ベースのビデオ プロセッシング例
- 「合成可能 FIR フィルター」: Vivado HLS ライブラリを使用した例
- 「行列乗算」: 標準的な線形代数ハードウェア アクセラレータ
- 「C 呼び出し可能な RTL ライブラリの使用」: ハードウェア記述言語 (HDL) で記述されたパッケージ済み C 呼び出し可能 IP を使用した例

ファイル I/O ビデオ

フレーム バッファーの読み出し/書き込みを実行する代わりに、ファイルからビデオ データを読み出して処理済みデータをファイルにライトバックすると有益な場合があります。この手法を file_io_manr_sobel という単純なサンプル デザインで示します。このサンプル デザインでは、ZC702 のベース プラットフォームが使用されています。次に、main() 関数の全体的な構造を示します。

```
int main()
{
    // code omitted
    read_frames(in_filename, frames, rows, cols, …);
    process_frames(frames, …);
    write_frames(out_filename, frames, rows, cols, …);
    // code omitted
}
```

ビデオ ハードウェアでは入力と出力の同期が不要なため、process_frames() に含まれるソフトウェア ループは単純で、ハードウェア インプリメンテーションに manr と sobel_filter を選択した場合、ハードウェア関数パイプラインが作成されます。

```
for (int loop_cnt = 0; loop_cnt<frames; loop_cnt++) {
    // set up manr_in_current and manr_in_prev frames
    manr( nr_strength,manr_in_current, manr_in_prev, yc_out_tmp);
    sobel_filter(yc_out_tmp, out_frames[frame]);
 }
```

入力および出力ビデオ ファイルは YUV422 形式です。platform ディレクトリには、アクセラレータ コードで使用されるこれらのファイルをフレーム配列に変換するソースとフレーム配列をこれらのファイルに変換するソースが含まれています。最上位ディレクトリの makefile は、アプリケーション バイナリを生成するプラットフォームのソースと共にアプリケーション ソースをコンパイルします。

合成可能 FIR フィルター

SDSoC 環境に含まれる Vivado HLS ソース コード ライブラリの関数の多くは、SDSoC 環境の「コード ガイドライン」に従っていません。SDSoC 環境でこれらのライブラリを使用するには、通常関数をラップして、移植不可能なデータ型またはサポートされない言語コンストラクトから SDSoC システム コンパイラを隔離する必要があります。

合成可能 FIR フィルターのサンプル デザインでは、このようなライブラリ関数 (この場合は有限インパルス応答デジタル フィルターを計算するライブラリ関数) を使用する標準的な方法を示します。この例では、フィルター クラス コンストラクターおよび演算子を使用してサンプル ベースのフィルタリングを作成して実行します。SDSoC 環境内でこのクラスを使用するには、次のように関数ラッパー内にラップします。

```
void cpp_FIR(data_t x, data_t *ret)
{
    static CF<coef_t, data_t, acc_t> fir1;
    *ret = fir1(x);
}
```

このラッパー関数は、アプリケーション コードから起動できる最上位ハードウェア関数になります。

行列乗算

行列乗算は多くのアプリケーション ドメインで使用される計算負荷の高い一般的な操作です。SDSoC IDE には、すべてのベース プラットフォームに対するテンプレート例が含まれます。これらのコードには、「システム パフォーマンスの向上」で説明するように、メモリ割り当てとメモリ アクセスに SDSoC 環境のシステム最適化を有益に使用する方法が示されるほか、「最適化ガイドライン」で説明するように、関数インライン展開、ループ展開、配列の分割のような Vivado HLS 最適化が示されます。

C 呼び出し可能な RTL ライブラリの使用

SDSoC システム コンパイラでは、VHDL または Verilog のようなハードウェア記述言語 (HDL) のレジスタ トランスファー レベル (RTL) で記述した IP ブロックを使用してインプリメントされたハードウェア関数をライブラリに含めることができます。このようなライブラリを作成する方法については、「C 呼び出し可能な IP ライブラリの使用」を参照してください。この例では、SDSoC プロジェクトでライブラリを使用する方法を示します。

SDSoC IDE でこのサンプル デザインをビルドするには、新しい SDSoC プロジェクトを作成して、C 呼び出し可能な RTL ライブラリ テンプレートを選択します。`src/SDSoC_project_readme.txt` で説明されているように、まず SDSoC ターミナル ウィンドウからコマンド ラインでライブラリをビルドする必要があります。

ライブラリを使用してアプリケーションをビルドするには、「C 呼び出し可能な IP ライブラリの使用」で説明されるように `-l` および `-L` リンカー オプションを追加する必要があります。[Project Explorer] タブでプロジェクトを右クリックし、[C/C++ Build Settings] → [SDS++ Linker] → [Libraries] をクリックして、`-lrtl_arraycopy` および `-L<path to project>` オプションを追加します。

XILINX
ALL PROGRAMMABLE™

第 15 章

SDSoC のプラグマ仕様

このセクションでは、システム最適化を支援するための SDSoC システム コンパイラ sdscc/sds++ のプラグマ (指示子) について説明します。

SDSoC 環境専用のプラグマにはすべて #pragma SDS と最初に付いており、C/C++ ソース コードの関数宣言または関数呼び出しサイトの前にに挿入する必要があります。

ハードウェア アクセラレータを使用するヘテロジニアス エンベデッド システムをターゲットにするコンパイラには、業界標準となるような圧倒的に使用されているものはありませんが、プラグマおよびプラグマ構文は OpenCC のような規格と一貫するように定義されています。今後のリリースでは、広く使用される標準規格ができれば、SDSoC 環境でもその業界標準プラグマを導入する可能性があります。

データ転送サイズ

このプラグマの構文は、次のとおりです。

```
#pragma SDS data copy|zero_copy(ArrayName[offset:length])
```

このプラグマは、関数宣言の直前か、関数宣言に指定された別の #pragma SDS の直前に指定する必要があり、その関数の呼び出し元すべてに適用されます。

次に、この構文に関する注記のいくつかを示します。

- data copy は、データがプロセッサ メモリからハードウェア関数に明示的にコピーされることを意味します。「システム パフォーマンスの向上」に記述されるように、最適なデータ ムーバーでデータ転送が実行されます。data zero_copy は、ハードウェア関数が共有メモリから直接データにアクセスすることを意味します。この場合、ハードウェア関数は AXI4 バス インターフェイスを介して配列にアクセスする必要があります。
- 多次元配列では、各次元を指定する必要があります。たとえば 2 次元配列では、ArrayName[offset_dim1:length_dim1][offset_dim2:length2_dim2] を使用します。
- 同じプラグマで複数の配列をカンマ (,) で区切って指定できます。たとえば、copy(ArrayName1[offset1:length1], ArrayName2[offset2:length2]) のように指定します。
- [offset:length] はオプションです。
- ArrayName は関数定義の仮引数のいずれかである必要があります。つまり、プロトタイプ (パラメーター名はオプション) からではなく、関数定義からにする必要があります。
- offset は、対応する次元の最初の要素から数えた要素数です。コンパイル時定数にする必要があります。現在のところ、これは無視されます。
- length は、その次元で転送された要素数です。論理式が関数内で実行時に解決可能であれば、任意の論理式にすることができます。次に例を示します。

```
#pragma SDS data copy(InData[0:num_rows+3*num_coeffs_active + L*(P+1)])
#pragma SDS data copy(OutData[0:2*(L-M-R+2)+4*num_coeffs_active*(1+num_rows)])
void evw_accelerator (uint8_t M,
                      uint8_t R,
                      uint8_t P,
                      uint16_t L,
                      uint8_t num_coeffs_active,
                      uint8_t num_rows,
                      uint32_t InData[InDataLength],
                      uint32_t OutData[OutDataLength]);
```

このプラグマでは、配列引数用にハードウェア関数に転送される要素数が指定され、すべての関数呼び出しに適用されます。上記の例に示すように、length は定数にする必要はなく、同じ関数へのほかのスカラー パラメーターを含む C 演算式にできます。

配列引数にこのプラグマを指定しない場合、まず引数型がチェックされます。引数型がコンパイル時配列サイズである場合、コンパイラによりそれがデータ転送サイズとして使用されます。それ以外の場合、SDSoC 環境で呼び出しコードが解析され、配列のメモリ割り当て API (たとえば malloc または sds_alloc) に基づいて転送サイズが決定されます。解析でサイズを特定できないか、転送サイズに関して呼び出し元の間で不一致がある場合は、コンパイラからエラー メッセージが表示され、ユーザーがソース コードを変更する必要があります。

メモリの属性

仮想メモリをサポートする Linux のようなオペレーティング システムの場合、ユーザー空間の割り当てられたメモリがページ化されるために、システム パフォーマンスに影響が出ることがあります。システム パフォーマンスを改善するには、このセクションのプラグマを使用して、物理的に隣接したメモリに割り当てられた引数を宣言するか、コンパイラにキャッシュ コヒーレンシを実行する必要がないことを指定します。

物理的に隣接したメモリとデータ キャッシュ

重要 : このプラグマの構文およびインプリメンテーションは、今後のリリースで変更される可能性があります。

このプラグマの構文は、次のとおりです。

```
#pragma SDS data mem_attribute(ArrayName:cache|contiguity)
```

このプラグマは、関数宣言の直前か、関数宣言に指定された別の #pragma SDS の直前に指定する必要があり、その関数の呼び出し元すべてに適用されます。

次に、この構文に関する注記のいくつかを示します。

- ArrayName は、関数の仮引数の 1 つにする必要があります。
- cache は CACHEABLE または NON_CACHEABLE のいずれかにする必要があります。デフォルト値は CACHEABLE です。CACHEABLE に設定すると、コンパイラにより CPU と配列に割り当てられたメモリのアクセラレータとの間のキャッシュ コヒーレンシが保持されます。キャッシュ コヒーレンシを保持するため、HP ポートを使用している場合など、データをアクセラレータに転送する前にキャッシュをフラッシュし、アクセラレータからメモリにデータを転送する前にキャッシュを無効にする必要があることがあります。

  NON-CACHEABLE に設定すると、コンパイラにより指定のメモリのキャッシュ コヒーレンシは保持されません。ユーザーが必要に応じて実行する必要があります。このようにすると、コンパイラでメモリ ポートを割り当てる際の柔軟性が高くなります。典型的な使用ケースは、次のようなビデオ アプリケーションです。

  - 大きなビデオ データのキャッシュのフラッシュ/無効化が、システム パフォーマンスを大幅に低下させる可能性がある
  - ソフトウェア コードはビデオ データの読み出しまたは書き込みを実行しないので、プロセッサとアクセラレータの間のキャッシュ コヒーレンシは不要
- Contiguity は PHYSICAL_CONTIGUOUS または NON_PHYSICAL_CONTIGUOUS のいずれかにする必要があります。デフォルト値は NON_PHYSICAL_CONTIGUOUS です。PHYSICAL_CONTIGUOUS に設定すると、関連の ArrayName に対応するすべてのメモリは sds_alloc を使用して割り当てられます。NON_PHYSICAL_CONTIGUOUS に設定すると、関連の ArrayName は malloc を使用して割り当てられます。これは、SDSoC コンパイラでの最適なデータ ムーバーの選択に有益です。
- 1 つのプラグマで複数の配列をカンマ (,) で区切って指定できます。

データ アクセス パターン

このプラグマの構文は、次のとおりです。

```
#pragma SDS data access_pattern(ArrayName:pattern)
```

このプラグマは、アクセラレータの関数プロトタイプを含むヘッダー ファイルの関数宣言の直前で指定する必要があります。

次に、この構文に関する注記のいくつかを示します。

pattern は、SEQUENTIAL または RANDOM のいずれかに指定でき、デフォルトでは RANDOM が使用されます。

このプラグマを使用して、ハードウェア関数のデータ アクセス パターンを指定します。配列引数に Copy プラグマが指定されている場合は、SDSoC でこのプラグマの値が確認され、合成するハードウェア インターフェイスが決定されます。アクセス パターンに SEQUENTIAL が指定されている場合は、ap_fifo などのストリーミング インターフェイスが生成されます。RANDOM が指定されている場合は、RAM インターフェイスが生成されます。

データ ムーバーのタイプ

重要: このプラグマは、通常の使用にはお勧めしません。このプラグマは、コンパイラで生成されるデータ ムーバーでデザイン要件が満たされない場合のみに使用してください。

このプラグマの構文は、次のとおりです。

```
#pragma SDS data data_mover(ArrayName:DataMover)
```

このプラグマは、関数宣言の直前か、関数宣言に指定された別の #pragma SDS の直前に指定する必要があり、その関数の呼び出し元すべてに適用されます。

次に、この構文に関する注記のいくつかを示します。

- 1 つのプラグマで複数の配列をカンマ (,) で区切って指定できます。次に例を示します。

```
#pragma SDS data_mover(ArrayName:DataMover, ArrayName:DataMover)
```

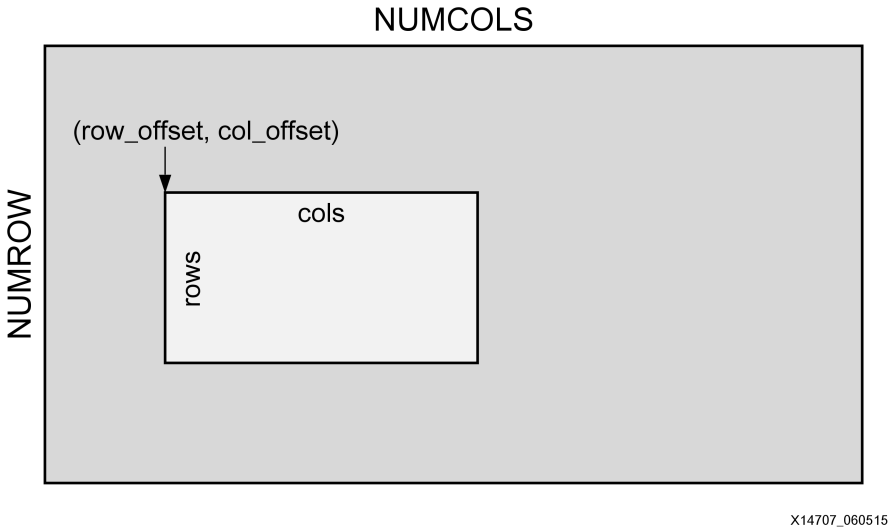
- ArrayName は、関数の仮引数の 1 つにする必要があります。
- DataMover は、AXIFIFO、AXIDMA_SG、AXIDMA_SIMPLE、または AXIDMA_2D にする必要があります。

このプラグマは、配列引数を転送するのに使用されるデータ ムーバー HW IP タイプを指定します。通常は、コンパイラによりコードが解析され、データ型が自動的に選択されます。このプラグマは、コンパイラでの推論を無効にするために使用できます。

AXIDMA_SIMPLE および AXIDMA_2D を使用するには、追加の要件があります。まず、対応する配列を sds_alloc() を使用して割り当てる必要があります。

- AXIDMA_2D では、各次元の 2D 配列のサイズを指定するためプラグマ SDS data dim が必要です。また、転送される 2D 配列の矩形サブ領域を指定するため SDS data copy プラグマも必要です。配列の 2 つ目の次元サイズ、サブ領域のオフセット、および列サイズは、すべて 64 ビット境界 (8 で除算可能なバイト数) に揃ったアドレスになることが必要です。
- 次に示す例では、AXIDMA_2D を適切に機能させるためには、NUMCOLS、row_offset、col_offset、および cols が 8 の倍数 (各 char のビット幅が 8) になるようにする必要があります。

```
#pragma SDS data data_mover(y_lap_in:AXIDMA_SIMPLE, y_lap_out:AXIDMA_2D)
#pragma SDS data dim(y_lap_out[NUMROWS][NUMCOLS])
#pragma SDS data copy(y_lap_out[row_offset:rows][col_offset:cols])
void laplacian_filter(unsigned char y_lap_in[NUMROWS*NUMCOLS],
          unsigned char y_lap_out[NUMROWS*NUMCOLS],
                     int rows, int cols, int row_offset, int col_offset);
```



外部メモリへの SDSoC プラットフォーム インターフェイス

重要 : このプラグマの構文およびインプリメンテーションは、今後のリリースで変更される可能性があります。

このプラグマの構文は、次のとおりです。

```
#pragma SDS data sys_port(ArrayName:port)
```

このプラグマは、関数宣言の直前か、関数宣言に指定された別の `#pragma SDS` の直前に指定する必要があり、その関数の呼び出し元すべてに適用されます。

次に、この構文に関する注記のいくつかを示します。

- `ArrayName` は、関数の仮引数の 1 つにする必要があります。
- `port` は、ACP、AFI、MIG のいずれかにする必要があります。Zynq-7000 All Programmable SoC では、ノンキャッシュ コヒーレント アクセス (AFI) 用にプログラマブル ロジックと外部メモリ (S_AXI_ACP) とハイ パフォーマンス ポート (S_AXI_HP) 間にキャッシュ コヒーレント インターフェイスが提供されています。`sys_port` プラグマが配列引数に対して指定される場合、メモリ属性 (キャッシュ可能か不可能か)、配列サイズ、使用されるデータ ムーバーなどに基づいて、SDSoC システム コンパイラで外部メモリへのインターフェイスが自動的に決定されます。このプラグマにより、メモリ ポートの SDSoC コンパイラーの選択を変更することができます。MIG は `zc706_mem` プラットフォームでのみ有効です。
- 1 つのプラグマで複数の配列をカンマ (,) で区切って指定できます。

ハードウェア バッファーのワード数

このプラグマの構文は、次のとおりです。

```
#pragma SDS data buffer_depth(ArrayName:BufferDepth)
```

重要 : このプラグマのハードウェア解釈は、今後のリリースで変更される可能性があります。

このプラグマは、関数宣言の直前か、関数宣言に指定された別の `#pragma SDS` の直前に指定する必要があり、その関数の呼び出し元すべてに適用されます。

次に、この構文に関する注記のいくつかを示します。

- 1 つのプラグマで複数の配列をカンマ (,) で区切って指定できます。次に例を示します。

```
#pragma SDS buffer_depth(ArrayName:BufferDepth, ArrayName:BufferDepth)
```

- `ArrayName` は、関数の仮引数の 1 つにする必要があります。
- `BufferDepth` はコンパイル時定数値にする必要があります。
- このプラグマは、BRAM または FIFO インターフェイスにマップされる配列のみに適用され、その配列の引数に割り当てるハードウェア バッファー数を指定します。ハードウェア バッファーでは、次を満たす必要があります。
  - BRAM : データ幅が ≤ 64 で、1 ≤ バッファーの深さ ≤ 4、2 ≤ 配列サイズ ≤ 16384
  - FIFO : データ幅が ≤ 64 で、2 ≤ バッファーの深さ * 配列サイズ ≤ 16384

関数の非同期実行

これらのプラグマ ペアはハードウェア関数の非同期実行をサポートしています。

これらのプラグマの構文は、次のとおりです。

```
#pragma SDS async(ID)
#pragma SDS wait(ID)
```

`async` プラグマはハードウェア関数呼び出しの直前で指定し、ハードウェア関数およびそのデータ転送の設定直後に制御を CPU に戻すようにコンパイラに指示します。

`wait` プラグマは、関連する `async` 関数およびデータ転送が完了するまで CPU が待機するようにプログラムに指示するように適切な位置に挿入する必要があります。

- `ID` は、コンパイル時の符号なし整数の定数にし、ハードウェア関数の識別子を示すようにします。つまり、同じハードウェア関数に別の `ID` を使用すると、その関数の別のハードウェア インスタンスになります。この結果、これらのプラグマを使用して複数のハードウェア インスタンスを強制的に作成することができます。
- `async` プラグマが含まれると、SDSoC システム コンパイラで関連する呼び出しに対してスタブ関数内に `sds_wait()` が生成されません。プログラムには、CPU で実行されている制御スレッドとハードウェア関数スレッドを同期させるために `sds_wait(ID)` または `#pragma SDS wait(ID)` を適切な位置に含める必要があります。`sds_wait(ID)` 関数呼び出しの代わりに `#pragma SDS wait(ID)` を使用すると、`sdscc` 以外のコンパイラでソース コードをコンパイルできる利点があります。この場合、`async` または `wait` プラグマは解釈されません。

パーティション仕様

SDSoC システム コンパイラ sdscc/sds++ では、ランタイム時にダイナミックに読み込まれた 1 つのアプリケーションに対して複数のビットストリームが自動的に生成されます。各ビットストリームには、それぞれパーティション識別子が含まれます。プラットフォームでは、たとえば再読み込み後にシャットダウンして再起動できないプラットフォーム ペリフェラルがある場合など、ビットストリームの再読み込みがサポートされないことがあります。

このプラグマの構文は、次のとおりです。

```
#pragma SDS partition(ID)
```

partition プラグマはハードウェア関数呼び出しの直前で指定し、そのパーティション ID にハードウェア関数のインプリメンテーションを割り当てるようにコンパイラに指示します。

- partition プラグマがない場合、ハードウェア関数はパーティション 0 にインプリメントされます。
- ID は正の整数にする必要があります。パーティション ID 0 は予約済みです。
- 次に、このプラグマの使用例を示します。

```
foo(a, b, c);
#pragma SDS partition (1)
bar(c, d);
#pragma SDS partition (2)
bar(d, e);
```

この例の場合、ハードウェア関数 foo には partition プラグマがないので、パーティション 0 にインプリメントされます。bar への最初の呼び出しはパーティション 1 に、2 つ目の bar はパーティション 2 にインプリメントされます。

このプラグマを使用する完全なサンプルは、samples/file_io_manr_sobel_partitions に含まれています。

XILINX
ALL PROGRAMMABLE™

第 16 章

SDSoC 環境の API

この章では、SDSoC 環境で開発されたアプリケーションに使用可能な sds_lib の関数について説明します。

注記： ライブラリを使用するには、ソース ファイルに #include "sds_lib.h" を含めます。sds_lib.h の前に stdlib.h 含め、size_t 型の宣言を提供する必要があります。

SDSoC™ 環境の API では、メモリ空間をマップし、非同期のアクセラレータの呼び出しが完了するのを待機する関数が提供されています。

`void sds_wait(unsigned int id)`
: id で指定されているキューの最初のアクセラレータが完了するのを待機します。「非同期関数の実行」に示すように、#pragma SDS wait(id) を使用することもできます。

`void *sds_alloc(size_t size)`
: 物理的に隣接している size バイトの配列を割り当てます。

`void *sds_alloc_non_cacheable(size_t size)`

: キャッシュ不可能としてマークされた size バイトの物理的に隣接している配列を割り当てます。この関数で割り当てられたメモリは、プロセッシング システムでキャッシュされません。このメモリへのポインターは、次と併せてハードウェア関数に渡す必要があります。

```
#pragma SDS data mem_attribute (p:NON_CACHEABLE)
```

`void sds_free(void *memptr)`
: sds_alloc() で割り当てられた配列を解放します。

`void *sds_mmap(void *physical_addr, size_t size, void *virtual_addr)`
: 物理アドレス (physical_addr) にあるメモリの size バイトにアクセスする仮想アドレス マップを作成します。

- physical_addr : マップする物理アドレス
- size : マップされる物理アドレスのサイズ
- virtual_addr :
  - ヌルではない場合、physical_addr に仮想アドレスが既にマップされていると判断され、sds_mmap によりマッピングがトラックされます。
  - ヌルの場合は、sds_mmap で mmap() により仮想アドレスが生成され、virtual_addr でこの値が割り当てられます。

`void *sds_munmap(void *virtual_addr)`

`sds_mmap()` を使用して作成された物理アドレスと関連付けられた仮想アドレスのマップを解除します。

`unsigned long long sds_clock_counter(void)`

細粒度の時間間隔計測に使用されるフリーランニング カウンターと関連付けられている値を返します。カウンターでは ARM CPU のクロック サイクルがカウントされ、0 に戻ります。

XILINX
ALL PROGRAMMABLE™

第 17 章

sdscc/sds++ コンパイラのコマンドおよびオプション

このセクションでは、sdscc/sds++ コンパイラのコマンドおよびオプションについて説明します。

名前

```
sdscc - SDSoC C compiler

sds++ - SDSoC C++ compiler
```

コマンドの概要

```
sdscc | sds++ [hardware_function_options] [system_options]
[performance_estimation_options] [options_passed_through_to_cross_compiler]
[-sds-pf platform_name] [-sds-pf-info platform_name] [-sds-pf-list] [-target-os os_name]
[-verbose] [ --help] [-version] [files]
```

ハードウェア関数オプション

```
[-sds-hw function_name file [-files file_list] [-hls-tcl hls_tcl_directives_file]
[-clkid clock_id_number] -sds-end]*
```

パフォーマンス予測オプション

```
[[-perf-funcs function_name_list -perf-root function_name] |
[-perf-est data_file][-perf-est-hw-only]]
```

システム オプション

```
[[-apm] [-dmclkid clock_id_number] [-impl-tcl tcl_file] [-mno-bitstream] [-mno-boot-files]
[-rebuild-hardware] [-poll-mode <0|1>] [-instrument-stub]]
```

sdscc/sds++ コンパイラは、C/C++ ソース ファイルを Zynq-7000 All Programmable SoC にインプリメントされたアプリケーション特定のハードウェア/ソフトウェア システム オン チップにコンパイルしてリンクします。

コマンドの使用方法とオプションは、sdscc と sds++ で同じです。

sdscc で認識されないオプションは、ARM クロスコンパイラに渡されます。-sds-hw ... -sds-end 節内のコンパイラ オプションは、foo.c が指定したハードウェア関数を含むファイルではない場合、-c foo.c オプションで無視されます。

アプリケーション ELF をリンクする場合、sdscc でハードウェア システムが作成されてインプリメントされ、ハードウェア システムを初期化するのに必要な ELF とブート ファイルを含む SD カード イメージが生成され、プログラマブル ロジックがコンフィギュレーションされ、ターゲット オペレーティング システムが実行されます。

Linux 以外のターゲット (スタンドアロン、FreeRTOS など) のアプリケーション ELF ファイルをリンクすると、<install_path>/platforms/<platform_name> フォルダーにあるデフォルトのリンカー スクリプトが使用されます。ユーザー定義のリンカー スクリプトが必要な場合は、-Wl,-T -Wl,<path_to_linker_script> リンカー オプションて指定できます。

ハードウェア インプリメンテーションにマークされた関数が含まれないシステムをビルドする際、sdscc ではターゲット プラットフォームで使用可能な場合は、ビルド済みハードウェアが使用されます。ビットストリーム生成を強制的に実行するには、-rebuild-hardware オプションを使用します。

レポート ファイルは、_sds/reports フォルダーに含まれます。

一般的なオプション

次のコマンド ライン オプションは、どの sdscc コマンドにも適用でき、指定した情報を表示します。

-sds-pf platform_name

オペレーティング システムとブート ファイルを含むベース システムのハードウェアおよびソフトウェアを定義するターゲット プラットフォームを指定します。platform_name は、SDSoC™ 環境インストールのプラットフォーム名であるか、プラットフォーム ファイルを含むフォルダーへのファイル パス (パスの最後のコンポーネントがプラットフォーム名と一致) です。プラットフォームでは、オペレーティング システムとブート ファイルを含むベース ハードウェアおよびソフトウェアが定義されます。アクセラレータのソース ファイルをコンパイルするときと ELF ファイルをリンクするときにこのオプションを使用します。使用可能なプラットフォームを表示するには、-sds-pf-list オプションを使用します。

-sds-pf-info platform_name

プラットフォームの一般的な情報を表示します。使用可能なプラットフォームを表示するには、-sds-pf-list オプションを使用します。

-sds-pf-list

使用可能なプラットフォームの一覧を表示して終了します。

-target-os os_name

ターゲット オペレーティング システムを指定します。選択した OS により、使用されるコンパイラ ツールチェーンとインクルード ファイル、sdscc で追加されたライブラリ パスなどが決まります。os_name は、次のいずれかになります。

- linux：Linux OS。コマンド ラインに -target-os オプションが指定されない場合は、これがデフォルトになります。
- standalone：スタンドアロンまたはベアメタル アプリケーション
- freertos：FreeRTOS

-verbose

STDOUT に詳細出力を表示します。

-version

STDOUT に sdscc バージョン情報を表示します。

--help

コマンド ラインのヘルプ情報を表示します。ダッシュ (-) が 2 個使用されることに注意してください。

ハードウェア関数オプション

ハードウェア関数オプションには、makefile 内の sdscc オプションをまとめて、コマンド ライン呼び出しを単純化し、既存の makefile に最小限の変更を加える機能があります。次の makefile の部分では、-sds-hw ブロックを使用して SDSFLAGS という makefile 変数に含まれるオプションをすべて収集し、CC の元の定義を sdscc ${SDSFLAGS} または sds++ ${SDSFLAGS} に置換しています。これにより、アプリケーションの元の makefile に最小限の変更を加えるだけで sdscc/sds++ コンパイラの makefile に変換できます。

```
APPSOURCES = add.cpp main.cpp
EXECUTABLE = add.elf

CROSS_COMPILE = arm-xilinx-linux-gnueabi-
AR = ${CROSS_COMPILE}ar
LD = ${CROSS_COMPILE}ld
#CC = ${CROSS_COMPILE}g++
PLATFORM = zc702
SDSFLAGS = -sds-pf ${PLATFORM} \
           -sds-hw add add.cpp -clkid 1 -sds-end \
           -dmclkid 2
CC = sds++ ${SDSFLAGS}

INCDIRS = -I..
LDDIRS =
LDLIBS =
CFLAGS = -Wall -g -c ${INCDIRS}
LDFLAGS = -g ${LDDIRS} ${LDLIBS}

SOURCES := $(patsubst %,../%,$(APPSOURCES))
OBJECTS := $(APPSOURCES:.cpp=.o)

.PHONY: all

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
 ${CC} ${OBJECTS} -o $@ ${LDFLAGS}

%.o: ../%.cpp
 ${CC} ${CFLAGS} $<
```

-sds-hw function_name file [-files file_list] [-hls-tcl hls_tcl_directives_file] [-clkid <n>]] -sds-end

sdscc コマンド ラインには 0 または複数の -sds-hw ブロックを含めることができ、それぞれのブロックで最初の引数に最上位ハードウェア関数を、2 番目の引数に含まれるソース ファイルを指定します。-sds-hw ブロックで関連付けられるファイル名がコンパイルされるソース ファイル名と同じ場合に、オプションが適用されます。-sds-hw ブロック外のオプションは必要に応じて適用されます。

-files file_list

Vivado® HLS を使用して現在の最上位関数をハードウェアにコンパイルするには、1 つまたは複数のファイルのカンマ区切りのリスト (空白なし) を指定する必要があります。これらのファイルのいずれかに HLS では使用されないがアプリケーション実行ファイルを生成するのに必要なソース コードが含まれている場合は、ファイルを個別にコンパイルしてオブジェクト ファイル (.o) を作成し、リンク段階でその他のオブジェクト ファイルとリンクする必要があります。

-hls-tcl hls_tcl_directives_file

Vivado® HLS ツールを使用してハードウェア アクセラレータを合成するときに HLS 指示子を含む Tcl ファイルを使用します。sdscc では、HLS 合成中に Vivado HLS ツールを駆動するのに使用する run.tcl ファイルが作成されます。この Tcl ファイルには次のコマンドが含まれています。

```
# synthesis directives
create_clock -period <clock_period>
config_rtl -reset_level low
source <sdsoc_generated_tcl_directives_file>
# end synthesis directives
```

-hls-tcl オプションを使用すると、SDSoC 環境で生成された Tcl ファイルの代わりにユーザー定義の Tcl ファイル が使用されます。指定した Tcl ファイルに正しく機能する指示子ファイルを含めてください。クロック周期はプラットフォームによって異なります。リセット レベルはアクティブ Low にする必要があります。

```
# synthesis directives
create_clock -period <clock_period>
config_rtl -reset_level low
# user-defined synthesis directives
source <user_hls_tcl_directives_file>
# end user-defined synthesis directives
# end synthesis directives
```

-clkid <n>

アクセラレータ ID を <n> に設定します。<n> の値を次の表に示します。プラットフォームの情報を表示するには、sdscc -sds-pf-info platform_name コマンドを使用します。clkid オプションを指定しない場合は、デフォルトのプラットフォームが使用されます。使用可能なプラットフォームおよび設定をすべて表示するには、sdscc -sds-pf-list コマンドを使用します。

プラットフォーム	<n> の値
zc702	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
zc702_hdmi	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 166MHz
zc706	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
zed および microzed	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
zybo	0 ～ 25MHz
	1 ～ 100MHz
	2 ～ 125MHz
	3 ～ 50MHz

コンパイラ マクロ

定義済みマクロを使用すると、#ifdef および #ifndef プリプロセッサ文を含むコードを保護できます。マクロ名の前後にはアンダースコア (_) を 2 つずつ付けます。__SDSCC__ マクロは、sdscc または sds++ を使用してソース ファイルをコンパイルするときに定義され、コードが sdscc/sds++ でコンパイルされるか GCC などの別のコンパイラでコンパイルされるかに基づいてコードを保護できます。

sdscc または sds++ で Vivado HLS を使用したハードウェア アクセラレーション向けにソース ファイルをコンパイルするときは、__SDSVHLS__ マクロが定義され、高位合成が実行されるかされないかに基づいてコードを保護できます。

次のコードでは、sdscc/sds++ でソース コードをコンパイルするときには __SDSCC__ マクロで sds_alloc() および sds_free() 関数が使用され、別のコンパイラ ツールを使用するときは malloc() および free() 関数が使用されるように記述されています。

```
#ifdef __SDSCC__
#include <stdlib.h>
#include "sds_lib.h"
#define malloc(x) (sds_alloc(x))
#define free(x) (sds_free(x))
#endif
```

次の例では、__SDSVHLS__ マクロが使用されており、Vivado HLS でハードウェアを生成するときとソフトウェア インプリメンテーションで使用するときとで保護される関数定義に含まれるコードが異なります。

```
#ifdef __SDSVHLS__
void mmult(ap_axiu<32,1,1,1> A[A_NROWS*A_NCOLS],
           ap_axiu<32,1,1,1> B[A_NCOLS*B_NCOLS],
           ap_axiu<32,1,1,1> C[A_NROWS*B_NCOLS])
#else
void mmult(float A[A_NROWS*A_NCOLS],
           float B[A_NCOLS*B_NCOLS],
           float C[A_NROWS*B_NCOLS])
#endif
```

システム オプション

-apm

AXI Performance Monitor (APM) IP ブロックを挿入し、生成されたすべてのハードウェア/ソフトウェア インターフェイスを監視します。SDSoC IDE の [Debug] パースペクティブで、[Performance Counters View] の [Start] ボタンをクリックすると、アプリケーションを実行する前に APM をアクティブにできます。SDSoC IDE の詳細は、『SDSoC 環境ユーザー ガイド： 入門』(UG1028) を参照してください。

-dmclkid <n>

データ モーション ネットワーク クロック ID を <n> に設定します。<n> の値を次の表に示します。プラットフォームの情報を表示するには、sdscc -sds-pf-info platform_name コマンドを使用します。dmclkid オプションを指定しない場合は、デフォルトのプラットフォームが使用されます。使用可能なプラットフォームおよび設定をすべて表示するには、sdscc -sds-pf-list コマンドを使用します。

プラットフォーム	<n> の値
ZC702 プラットフォーム	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
ZC702_HDMI プラットフォーム	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 166MHz
ZC702 プラットフォーム	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
Zed および MicroZed プラットフォーム	0 ～ 166MHz
	1 ～ 142MHz
	2 ～ 100MHz
	3 ～ 200MHz
ZYBO プラットフォーム	0 ～ 25MHz
	1 ～ 100MHz
	2 ～ 125MHz
	3 ～ 50MHz

-impl-tcl tcl_file

Sdscc/sds++ で通常生成されるコマンドを使用せず、合成およびインプリメンテーション コマンドを含む Vivado Tcl ファイルを使用するよう指定します。次のコード ブロックは、Vivado 合成およびインプリメンテーションをユーザー デザインで実行するために生成された sdscc/sds++ Tcl ファイルの例です。

```
# ************************************************
# Open the Vivado Project
# ************************************************
open_project /home/user/test/_sds/p0/ipi/zc702.xpr
# ************************************************
# Run synthesis and implementation
# ************************************************
set_property STEPS.OPT_DESIGN.IS_ENABLED true [get_runs impl_1]
set_property STEPS.OPT_DESIGN.ARGS.DIRECTIVE Default [get_runs impl_1]
reset_run synth_1
launch_runs synth_1
wait_on_run synth_1
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
# ************************************************
```

```
# Save bitstream for SD card creation
# ****************************************************
set bitstream [get_property top [current_fileset]].bit
set directory [get_property directory [current_run]]
file copy -force [file join $directory $bitstream] [file join $directory bitstre
am.bit]
```

-Impl-tcl オプションが指定された場合は、合成およびインプリメンテーション コマンドは指定されている Tcl ファイルを参照するコマンドと置き換えられます。Tcl ファイルには、コマンドのリストをコメントとして含め (launch_runs、reset_run、wait_on_run)、run 名 synth_1 および impl_1 を使用する必要があります。

```
# ****************************************************
# Open the Vivado Project
# ****************************************************
open_project /home/user/test/_sds/p0/ipi/zc702.xpr
# ****************************************************
# Run synthesis and implementation
# ****************************************************
# User synthesis and implementation TCL was specified.
# It must include these commands and run names :
#   launch_runs synth_1
#   reset_run   synth_1
#   wait_on_run synth_
#   launch_runs impl_1  -to_step write_bitstream
#   wait_on_run impl_1
# ****************************************************
source /home/user/test/impl.tcl
# End user implementation TCL
# ****************************************************
# Save bitstream for SD card creation
# ****************************************************
set bitstream [get_property top [current_fileset]].bit
set directory [get_property directory [current_run]]
file copy -force [file join $directory $bitstream] [file join $directory bitstre
am.bit]
```

次に示すサンプルの impl.tcl Tcl ファイルでは、opt_design および phys_opt_design コマンドが Explore 指示子と共に使用されています。

```
set_property STEPS.OPT_DESIGN.IS_ENABLED true [get_runs impl_1]
set_property STEPS.OPT_DESIGN.ARGS.DIRECTIVE Explore [get_runs impl_1]
set_property STEPS.PHYS_OPT_DESIGN.IS_ENABLED true [get_runs impl_1]
set_property STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE Explore [get_runs impl_1]
reset_run synth_1
launch_runs synth_1
wait_on_run synth_1
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
```

-mno-bitstream

プログラマブル ロジック (PL) をコンフィギュレーションするのに使用するデザインのビットストリームを生成しません。通常、ビットストリームは Vivado インプリメンテーション機能を実行して生成されますが、ビットストリームの生成にはデザインのサイズと複雑性によって数分から数時間かかります。このオプションを使用すると、ハードウェアの生成に影響しないフローを反復実行する場合にこの手順をディスエーブルにできます。アプリケーション ELF はビットストリーム生成前にコンパイルされます。

-mno-boot-files

SD カード イメージを sd_card フォルダーに生成しません。このフォルダーには、デバイスをブートして指定の OS を起動するのに必要なアプリケーション ELF およびファイルが含まれます。このオプションを使用すると、sd_card フォルダーの作成をディスエーブルにし、このフォルダーに含まれている以前のバージョンを保持できます。

-rebuild-hardware

ハードウェアにマップされた関数のないソフトウェアのみのデザインをビルドする場合、sdscc ではプラットフォーム内で使用可能な場合はビルド済みビットストリームが使用されますが、このオプションを使用するとシステム全体が強制的にビルドされます。

-poll-mode <0|1>

1 に設定すると DMA ポーリング モードがイネーブルになり、0 (デフォルト) に設定すると割り込みモードがイネーブルになります。たとえば DMA ポーリング モードを指定するには、`sdscc-poll-mode 1` オプションを追加します。

-instrument-stub

カウンター関数 `sds_clock_counter()` への呼び出しを含む生成されたハードウェア関数スタブを計測します。ハードウェア関数スタブを計測すると、関数への各呼び出しに対して、送信および受信関数を呼び出すのに必要な時間、待機した時間が表示されます。

Linux ブート後のオプションの PL コンフィギュレーション
`sdscc/sds++` で `sd_card` フォルダーにビットストリーム `.bin` ファイルが作成される場合、Linux がブートした後アプリケーション ELF を実行する前にこのビットストリーム ファイルを使用して PL をコンフィギュレーションできます。使用するエンベデッド Linux コマンドは `cat bin_file > /dev/xdevcfg` です。

XILINX
ALL PROGRAMMABLE™

付録 A

ハードウェア関数インターフェイスの詳細

重要： この付録には、SDSoC 環境を使用する際には通常必要のないハードウェア関数のハードウェア インターフェイスに関する参考資料を含めます。これらは、ソース コードに明示的な HLS ハードウェア インターフェイス プラグマを記述する必要がある場合にのみ必要です。たとえば、関数に 8 ストリーム入力または出力を超える入出力が必要な場合や、ユーザー IP に対して C 呼び出し可能/C リンク可能なライブラリを作成して、RTL を Vivado HLS で生成したハードウェア インターフェイスと一致させたい場合などです。

ハードウェア関数制御プロトコル

SDSoC システム コンパイラでは、ハードウェア関数に対する正しい制御プロトコルが自動的に決定されます。このセクションには、ソース コードに明示的な Vivado® HLS プラグマを記述せざるえない場合にのみ必要な参考資料が含まれます。たとえば、関数に 8 ストリーム入力または 8 ストリーム出力を超える入出力が必要な場合や、IP に対して C 呼び出し可能/C リンク可能なライブラリを作成して、RTL を Vivado® HLS で生成したハードウェア インターフェイスと同じにしたい場合などです。

SDSoC™ 環境では次のハードウェア関数制御プロトコルがサポートされており、ハードウェア インターフェイス定義に基づいて自動的に推論されます。自動生成されたソフトウェア スタブ関数により制御プロトコルがインプリメントされ、`<sds_install_root>/arm-xilinx*-gnueabi/include/cf_lib.h` に定義されている `cf_send_i()`、`cf_receive_i()`、および `cf_wait()` API を使用してデータ転送とハードウェア関数の実行が同期化されます。

- None：ソフトウェア制御インターフェイスなし。ハードウェア関数は、AXI ストリーム にマップされている引数に基づいて完全に自己同期する必要があります。スカラー引数またはメモリ マップされている引数を含めることはできません。AXI ストリーム ポートにはすべて TLAST および TKEEP 側帯波信号を含める必要があります。
- axis_acc_adapter：SDSoC 環境に含まれる Vivado® Design Suite HLS のハードウェア関数向けデフォルト インターフェイス。SDSoC 環境では、Vivado 高位合成のハードウェア関数とインターフェイスさせるために axis_accelerator_adapter IP インスタンスが自動的に挿入されます。この IP では、ソフトウェアのパイプライン処理用のパイプライン構造の AXI4-Lite 制御/データ インターフェイスと、ハードウェア関数をデータ モーション ネットワークより高い (または低い) クロック レートで実行するためのクロック乗せ換え回路が提供され、計算と通信のバランスが取られます。アダプターでもオプションで BRAM および FIFO インターフェイスにマップする引数のマルチバッファリングが提供されており、引数が自動的に AXI4-Stream にマップされます (buffer_depth プラグマについては「ハードウェア バッファーの深さ」を参照)。ハードウェア関数のインターフェイスには #pragma HLS interface s_axilite が指定された引数を含めることはできませんが、単一の AXI-MM マスター インターフェイス (offset=direct プラグマ属性付き) および TLAST と TKEEP 側帯波信号を含む AXI4-Stream インターフェイスにマップされる引数はいくつでも含めることができます。

  axis_accelerator_adapter IP では、最大 8 つの AXI4-Stream 入力まで、および最大 8 つの AXI4-Stream 出力までサポートされ、それぞれ BRAM または FIFO インターフェイスのいずれかにマップできます。IP には、AXI4-Lite レジスタ インターフェイスも含まれ、入力、出力、または入出力引数のいずれかに使用可能な 8 つの入力レジスタ、8 つの出力レジスタ、8 つの入力/出力レジスタを使用したスカラー引数がサポートされます。スカラー引数は、bool、char、short、int、または float 型にできます。関数戻り値は、出力スカラー レジスタにマップされます。これらの制約に従うことができないハードウェア関数には、generic_axi_lite 制御プロトコルを含める必要があります。
- generic_axi_lite：#pragma HLS interface s_axilite で引数がマップされた場合のネイティブの Vivado HLS 制御インターフェイス。このインターフェイスは、『SDSoC 環境ユーザー ガイド：プラットフォームおよびライブラリ』(UG1146) の「ライブラリの作成」に説明されている C 呼び出し可能な HDL IP 用です。ハードウェア制御レジスタは、次のビット エンコードを使用してオフセット 0x0 に配置する必要があります。

```
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

Vivado HLS 関数引数型

このセクションでは、Vivado® HLS を使用して SDSoC™ システム コンパイラでコンパイルされたハードウェア関数でサポートされるハードウェア インターフェイス タイプについて説明します。コンパイラでは、引数型 `#pragma SDS data copy|zero_copy` および `#pragma SDS data access_pattern` に基づいて自動的にハードウェア インターフェイスのタイプが判断されます。

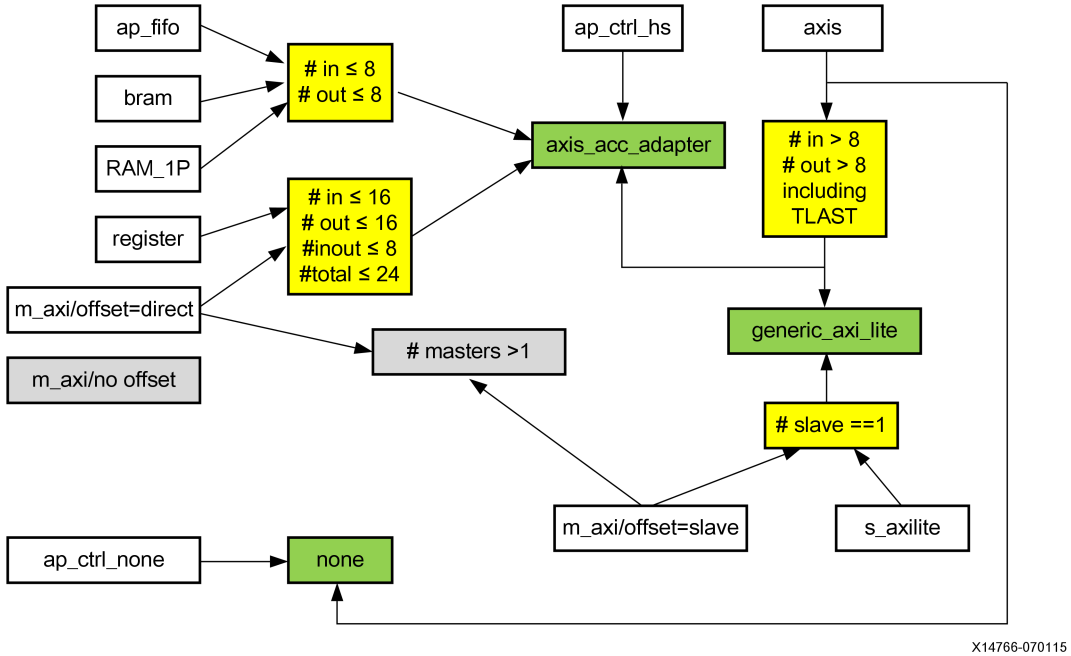
重要： インターフェイスの互換性が損なわれないようにするため、Vivado® HLS インターフェイス タイプ指示子およびプラグマをソース コードに含めるのは、`sdscc` で適切なハードウェア インターフェイス指示子が生成できないときのみにし、このセクションで説明する HLS インターフェイス タイプのみを使用してください。

- Vivado® HLS では、任意精度型の `ap_fixed<int>`、`ap_int<int>`、および `hls::stream` クラスが提供されています。SDSoC 環境では、最上位ハードウェア関数の引数の幅を 8、16、32、または 64 ビットにし、これらの宣言を `#ifndef __SDS_VHLS__` を使用して保護して、`char`、`short`、`int`、または `long long` などの同様のサイズの C99 型に強制する必要があります。Vivado HLS `hls::stream` 引数は配列として `sdscc/sds++` に渡す必要があります。`<sdsoc_install_dir>/samples/hls_if/hls_stream` のサンプルは、SDSoC 環境で HLS の `hls::stream` 型の引数を使用する方法を示します。
- デフォルトでは、ハードウェア関数への配列引数はデータをコピーすると転送されます。これは、`#pragma SDS data copy` を使用するのと同等です。このため、配列引数は入力として使用するか、出力として生成する必要があり、両方には使用しないようにします。ハードウェア関数で読み出しおよび書き込みされる配列の場合は、`#pragma SDS data zero_copy` を使用して、コンパイラにその配列は共有メモリ内に保持する必要があり、コピーされないように指示する必要があります。

`sdscc` コンパイラは、プログラム構造、ハードウェア関数のプロトタイプ、およびその引数型に基づいて、ハードウェア関数の制御プロトコルを選択します。ここからはシステム コンパイラでサポートされるハードウェア インターフェイス タイプについて説明しますが、Vivado HLS インターフェイス プラグマの明示的使用は、ツールのデフォルトと制御プロトコル要件が競合したときに予期せぬエラーが発生することを回避するため、推奨されていないことを念頭に置いておいてください。

次の図に、ハードウェア インターフェイスのタイプ (白色) とサポートされる制御プロトコル (緑色) の関係を示します。関連する制約は黄色で示されています。サポートされない HLS インターフェイス指示子はグレーで示されています。

図 17-1：ハードウェア関数の制御プロトコルとサポートされるハードウェア インターフェイス



SDSoC 環境では、次のハードウェア インターフェイス タイプがサポートされています。

- RAM：アクセラレータ関数宣言の直前に `#pragma SDS data access_pattern(A:RANDOM)` を使用します。SDSoC 環境では、オプションでアクセラレータにマルチバッファリングを含め、DMA プロトコルと互換性があるパケット化された AXI4-Stream チャネルに自動的にマップされます。ハードウェア関数には 8 入力以上の bram/ap_fifo 引数および 8 出力以上の bram/ap_fifo 引数は使用できません。

  `<sdsoc_install_dir>/samples/hls_if/mmult_hls_bram` のサンプルでは、SDSoC 環境で HLS の BRAM インターフェイスを使用する方法が示されています。

- FIFO：アクセラレータ関数宣言の直前に `#pragma SDS data access_pattern(A:SEQUENTIAL)` を使用します。SDSoC 環境では、DMA プロトコルと互換性があるパケット化された AXI4-Stream チャネルに自動的にマップされます。ハードウェア関数には 8 入力以上の bram/ap_fifo 引数および 8 出力以上の bram/ap_fifo 引数は使用できません。

  `<sdsoc_install_dir>/samples/hls_if/mmult_hls_ap_fifo` のサンプルでは、SDSoC 環境で HLS の ap_fifo インターフェイスを使用する方法が示されています。

- SCALAR：SDSoC 環境では、基本演算タイプ (8、16、または 32 ビット) の引数を AXI4-Lite インターフェイスを介してアクセス可能なレジスタに自動的にマップされます。複数のタスクの呼び出しでタスクのパイプライン処理をサポートするため、レジスタが FIFO として扱われます。HLS のハードウェア関数では、最大 8 個の入出力スカラー引数または 16 個の出力スカラー引数までサポートできます。ただし、戻り値を含めたスカラー引数の合計数が 24 を超えないようにします。これ以上の数のスカラー引数が必要な場合は、HLS プラグマを使用してすべてのスカラー引数を HLS で生成される AXI4-Lite インターフェイスに明示的にマップする必要があります。

  ハードウェア関数には、スカラー レジスタにマップされた引数と明示的に AXI4-Lite にマップされた引数の両方を含めることはできません。

- AXI4-Lite：ハードウェア関数で `#pragma HLS INTERFACE s_axilite port=arg` を使用します。このプラグマを含める場合は、メモリ マップド制御インターフェイスを HLS で生成するために `#pragma HLS INTERFACE s_axilite port=return` も必要です。コマンド インターフェイスとスカラー引数には FIFO がありません。ハードウェア関数には、AXI4-Lite インターフェイスを 1 つしか明示的に指定できないため、`ap_control` を含めたすべてのポートを 1 つの AXI4-Lite インターフェイスにまとめる必要があります。

- AXI-memory mapped (AXI-MM) master：`#pragma HLS INTERFACE m_axi port=arg` を使用して AXI4-Lite インターフェイスを介して物理アドレスを渡します。このモードでは、ハードウェア関数が独立したデータ ムーバーとして動作します。ハードウェア関数で引数が AXI-MM マスターにマップされる場合は、出力スカラー引数または戻り値も含める必要があります。

  `<sdsoc_install_dir>/samples/hls_if/mmult_hls_aximm` のサンプルでは、SDSoC 環境で HLS の AXI-MM インターフェイスを使用する方法が示されています。

- AXI4-Stream：ハードウェア関数で `#pragma HLS INTERFACE axis port=arg` を使用します。SDSoC 環境では、ハードウェア関数とそれに対応する AXI4-Stream インターフェイスの直接接続がサポートされています。`<sdsoc_install_dir>/samples/hls_if/mmult_hls_axis` のサンプルでは、SDSoC 環境で HLS の AXI4-Stream インターフェイスを使用する方法が示されています。

重要： AXI4-Stream 転送チャネルにマップする必要がある入力配列引数または出力配列引数がハードウェア関数に 8 個以上あるなど、このインターフェイスが絶対に必要な場合以外は、このインターフェイスは使用しないでください。それ以外の場合は、*`#pragma SDS data access_pattern(A:SEQUENTIAL)`* 属性を使用し、`sdscc` で自動的に配列転送が AXI4-Stream チャネルにマップされるようにします。

重要： DMA データ ムーバーを使用したデータ転送では AXI4-Stream の `TLAST` および `TKEEP` 側帯波信号が必要で、これらは HLS コードで明示的に記述する必要があります。

XILINX
ALL PROGRAMMABLE™

付録 B

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 リソースについては、ザイリンクス サポート サイトを参照してください。

ソリューション センター

デバイス、ツール、IP のサポートについては、ザイリンクス ソリューション センターを参照してください。デザイン アシスタント、アドバイザリ、トラブルシューティングのヒントなどが含まれます。

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. 『SDSoC 環境ユーザー ガイド : SDSoC 環境の概要』(UG1028) (SDSoC 環境の docs フォルダーからも入手可能)
2. 『SDSoC 環境ユーザー ガイド』(UG1027) (SDSoC 環境の docs フォルダーからも入手可能)
3. 『SDSoC 環境ユーザー ガイド : プラットフォームおよびライブラリ』(UG1146) (SDSoC 環境の docs フォルダーからも入手可能)
4. 『UltraFast エンベデッド デザイン設計手法ガイド』(UG1046 : 英語版、日本語版)
5. 『ZC702 評価ボード (Zynq-7000 XC7Z020 All Programmable SoC 用) ユーザー ガイド』(UG850)
6. 『Vivado Design Suite ユーザー ガイド : 高位合成』(UG902)
7. 『PetaLinux ツール資料ワークフロー チュートリアル』(UG1156)
8. Vivado® Design Suite 資料
9. 『Vivado Design Suite ユーザー ガイド : カスタム IP の作成とパッケージ』(UG1118)

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）ものとします。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、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になります。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。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。

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。