Vivado 高位合成を使用した FPGA デザインの概要

UG998 (v1.1) 2019 年 1 月 22 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

この資料は、現在 Vitis 高位合成ユーザー ガイド (UG1399) に置き換わっています。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2019 年 1 月 22 日 バージョン 1.1	
資料全体	編集上のアップデート。
「DSP ブロック」	DSP ブロックに関する情報をアップデート。
「ストレージ エレメント」	UltraRAM に関する情報を追加。
2019 年 7 月 2 日 バージョン 1.0	
初版。	なし

目次

改訂履歴 . . . 2

第 1 章: はじめに
概要 . . . 5
プログラミング モデル . . . 6
このガイドの構成 . . . 8

第 2 章: FPGA とは
概要 . . . 10
FPGA アーキテクチャ . . . 10
FPGA の並列処理 vs プロセッサ アーキテクチャ . . . 16

第 3 章: ハードウェア デザインの基本概念
概要 . . . 21
クロック周波数 . . . 22
レイテンシおよびパイプライン処理 . . . 25
スループット . . . 26
メモリのアーキテクチャおよびレイアウト . . . 27

第 4 章: Vivado 高位合成 (HLS)
概要 . . . 30
演算 . . . 31
条件文 . . . 33
ループ . . . 34
関数 . . . 35
ダイナミック メモリ割り当て . . . 36
ポインター . . . 37

第 5 章: 計算中心のアルゴリズム
概要 . . . 39
データ レートの最適化 . . . 41

第 6 章: 制御中心のアルゴリズム
概要 . . . 46
C++ で記述される制御 . . . 47
UDP パケットの処理 . . . 52

第 7 章: ソフトウェア検証および Vivado HLS
概要 . . . 57
ソフトウェア テストベンチ . . . 57
コード カバレッジ . . . 59

初期化されない変数 . . . 61
範囲外のメモリ アクセス . . . 61
協調シミュレーション . . . 62
C/C++ 検証が不可能な場合 . . . 64

第 8 章: 複数のプログラムの統合

概要 . . . 65
AXI . . . 65
デザイン例: Zynq-7000 SoC 上で実行するアプリケーション . . . 69

第 9 章: 完成したアプリケーションの検証

概要 . . . 79
スタンドアロンの計算システム . . . 79
プロセッサ ベースのシステム . . . 81

付録 A: 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 . . . 84
ソリューション センター . . . 84
Documentation Navigator およびデザイン ハブ . . . 84
参考資料 . . . 85
お読みください: 重要な法的通知 . . . 85

XILINX

第 1 章

はじめに

概要

ソフトウェアはすべてのアプリケーションの基本です。用途がエンターテインメントやゲーミングであっても、通信または医療であっても、現在使用されている製品の多くは、まずソフトウェア モデルまたはプロトタイプを作成するところから始まります。ソフトウェア エンジニアは、システムのパフォーマンスおよびプログラマビリティに基づいて、プロジェクトを市場に出すために最も適したインプリメンテーション プラットフォームを判断する仕事を担います。その判断を下すため、ソフトウェア エンジニアには、プログラミングのテクニックだけでなく、さまざまなハードウェア プロセッシング プラットフォームも活用します。

プログラミングに関しては、過去数十年間で、コード再利用を目的としたオブジェクト指向プログラミングと、アルゴリズム パフォーマンスの向上を目指した並列処理機能において、進歩が重ねられてきました。プログラミング言語、フレームワーク、ツールが進歩したおかげで、ソフトウェア エンジニアは、特定の問題の解決に向け、プロトタイプをすばやく作成し、さまざまな方法をテストできるようになりました。ただ、ソリューションをすばやくプロトタイプ化することが求められるので、考慮しなければならないことも 2 点あります。1 点は、さまざまなアルゴリズムを比較検討する上で、それをどのように解析し、定量化するかです。この点についてはほかの資料で詳しく説明されているので、このガイドでは詳しくは説明しません。もう 1 点は、アルゴリズムをどこで実行するかです。このガイドでは、この点を FPGA (フィールド プロフラマブル ゲート アレイ) に関連させて説明します。

アルゴリズムをどこで実行するかに関連し、並列処理および同時処理への関心も高まっています。ソフトウェア プログラムの並列処理への関心は新しいものではありませんが、プロセッサや ASIC デザインにおける動向に後押しされて、再び関心が高まっています。これまでは、ソフトウェア エンジニアの視点から見て、ソフトウェア アルゴリズムでできるだけ高いパフォーマンスを得るには、選択肢はカスタム回路か FPGA かの 2 つでした。

最もコストがかかるのは、アルゴリズムをハードウェア エンジニアに委ね、カスタム回路を開発するオプションです。このオプションのコストは次に基づいています。

- カスタム回路を開発するコスト
- アルゴリズムをハードウェアに変換するのにかかる時間

消費電力や計算スループット、ロジックの集積度など、ハードウェア製造技術において著しい進歩はありましたが、用途を特化したカスタム回路 (ASIC) の開発コストは依然として高くつきます。ASIC の製造工程はコストがかさむので、製品を何百万個という単位で出荷できるようなアプリケーションでないと、採算が合いません。

2 つ目のオプションは FPGA を利用するもので、ASIC の開発コストの問題を解消できます。FPGA を使用すると、基本的なプログラマブル ロジック エレメントで構成された既製のコンポーネントを使用して、アルゴリズムのカスタム回路インプリメンテーションを作成できます。このプラットフォームでは、生産量が何百万に満たない場合でも、ASIC 開発のように工程が複雑にならず、コストもかけずに、消費電力を抑えて良いパフォーマンスを得ることができます。FPGA でも、ASIC と同様に、並列処理を利用したアルゴリズムのカスタム回路をインプリメントできます。

プログラミング モデル

FPGA の使用に踏み切るときに決め手の 1 つになるのが、ハードウェア プラットフォームのプログラミング モデルです。ソフトウェア アルゴリズムは通常、コンピューティング プラットフォームの詳細を抽象化する C/C++ などの高級プログラミング言語で記述されます。これらの言語はコードのすばやい反復実行、段階的な改善、コードの移植性に優れていますが、これらの特性はソフトウェア エンジニアにとって非常に重要です。ここ数十年間で、これらの言語によりアルゴリズムが高速に実行できるようになり、プロセッサおよびソフトウェア コンパイラの開発が促進されました。

もともと、ソフトウェアの実行時間の短縮は、プロセッサのクロック周波数を高めること、用途が特化されたプロセッサを使用することの 2 点に基づいていました。長年、ソフトウェアの実行時間を短縮するには、次世代のプロセッサが登場するまで 1 年待つのが常識でした。クロック周波数が上がるほど、ソフトウェア プログラムも高速に実行されるようになりました。この方法でよい結果が得られるケースもありましたが、多くのアプリケーションでは、競争に勝てる製品を市場に出すには、プロセッサのクロック周波数を高めてソフトウェア プログラムの実行を段階的に高速化するだけでは不十分でした。

そこで、その問題を克服する目的で、特殊プロセッサが作られました。特殊プロセッサには、DSP (デジタル信号処理) やグラフィックス プロセッシング ユニット (GPU) など多くの種類がありますが、これらの特殊プロセッサはどれも、C などの高級言語で記述されたアルゴリズムを実行できるのに加え、ターゲットのソフトウェア アプリケーションの実行を改善するためのアクセラレータを備えています。

標準プロセッサおよび特殊プロセッサの設計において近年パラダイム シフトが起き、どちらのタイプのプロセッサでも、ソフトウェア プログラムを高速化するのに、クロック周波数を高めるのではなく、チップごとにより多くのプロセッシング コアを追加するようになりました。こうしてマルチコア プロセッサが登場し、ソフトウェアのパフォーマンスを向上する最先端のテクニックとしてプログラムの並列処理が見直されるようになりました。ソフトウェア エンジニアは、パフォーマンスを向上するため、並列処理を効率よく実行できるアルゴリズムを構築する必要があります。アルゴリズム設計に必要なテクニックには、FPGA 設計と同じ基本要素が使用されます。FPGA とプロセッサとの主な違いは、プログラミング モデルです。

従来、FPGA のプログラミング モデルは、C/C++ ではなく、レジスタ トランスファー レベル (RTL) による記述が中心でした。このデザイン入力モデルは、ASIC デザインと完全に互換性がありますが、ソフトウェア エンジニアリングにおけるアセンブリ言語でのプログラミングに似ています。図 1-1 に、デザイン入力方法として RTL を使用した従来の FPGA デザイン フローを異なる計算プラットフォームと比較し、プログラミング モデルの違いがインプリメンテーション時間および達成可能なパフォーマンスにどのように影響するかを示します。

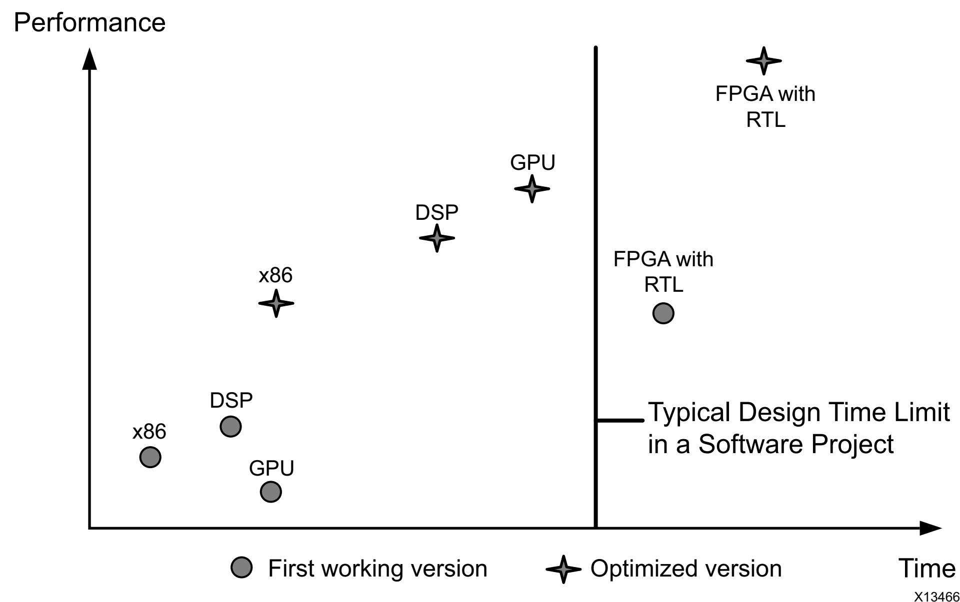

図 1-1: RTL デザイン入力を使用した設計時間 vs. アプリケーションのパフォーマンス

図 1-1 に示すように、機能するソフトウェア プログラムの初期バージョンは、標準プロセッサおよび特殊プロセッサの両方で、プロジェクト デザイン サイクルの比較的早期に得られます。初期バージョンが得られたら、どのインプリメンテーション プラットフォームでも、最高のパフォーマンスを達成するために作業を進めます。

この図は、FPGA プラットフォーム用に同じソフトウェア アプリケーションを開発するのにかかる時間も示しています。アプリケーションの初期バージョンも最適化バージョンも、同じ段階の標準プロセッサと特殊プロセッサと比較すれば、かなり良いパフォーマンスが得られています。RTL コードで記述され、FPGA で最適化されたアプリケーションのインプリメンテーションで、最も高いパフォーマンスが得られています。

ただし、このインプリメンテーションの達成にかかる開発時間は、一般的なソフトウェア開発の範囲を超えています。そのため、従来 FPGA は、複数のプロセッサを含むデザインなど、ほかの方法では達成できないパフォーマンスを必要とするアプリケーションのみに使用が限られていました。

近年のザイリンクスによる技術の進歩により、プロセッサと FPGA のプログラミング モデルに差はなくなりました。プロセッサのアーキテクチャに合わせて C などの高級言語用にさまざまなコンパイラが存在するように、ザイリンクスの Vivado® 高位合成 (HLS) コンパイラも、ザイリンクスの FPGA をターゲットとする C/C++ プログラムに同じ機能を提供します。図 1-2 に、Vivado HLS コンパイラを、使用可能なほかのプロセッサ ソリューションと比較した結果を示します。

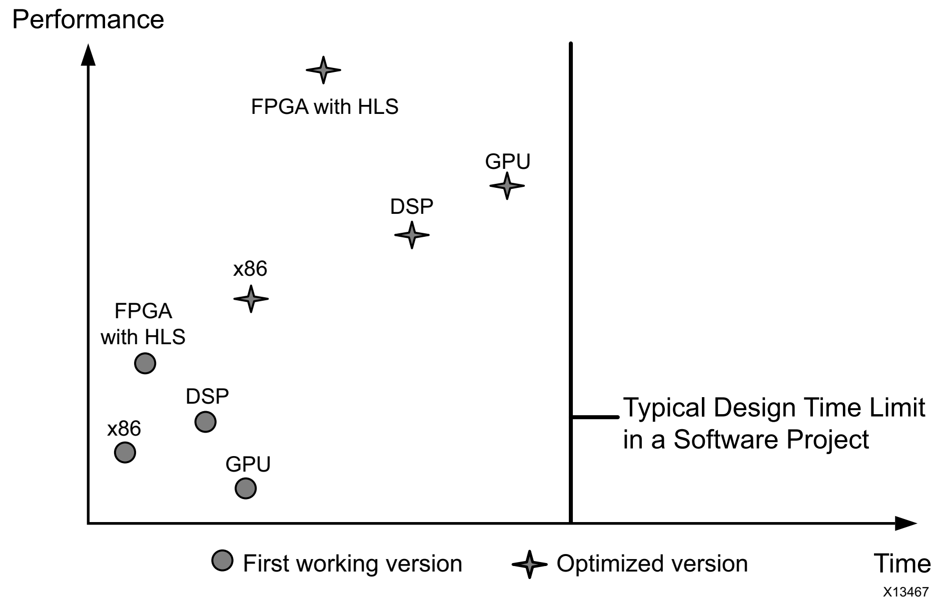

図 1-2: Vivado HLS コンパイラを使用した設計時間 vs. アプリケーションのパフォーマンス

このガイドの構成

同じ C/C++ アプリケーションでも、FPGA とほかのプロセッサのパフォーマンスには大きな差があります。このガイドの次の章では、その劇的なパフォーマンスの差の理由を説明し、Vivado HLS コンパイラがどのように機能するのかを示します。

第 2 章: FPGA とは

第 2 章「FPGA とは」では、FPGA で使用可能な計算エレメントを紹介し、これらのエレメントをプロセッサと比較します。また、FPGA メモリ階層とロジック エレメントについてと、それらのエレメントがどのように相互に関連しているのかを説明します。

第 3 章: ハードウェア デザインの基本概念

プロセッサのハードウェアと FPGA のハードウェアの違いは、各ターゲットのコンパイラの機能に影響します。第 3 章「ハードウェア デザインの基本概念」では、FPGA およびプロセッサのどちらのデザインにも適用されるハードウェアの基本概念を説明します。これらの基本概念を理解しておくと、Vivado HLS を使用して最適なプロセッシング アーキテクチャを作成するのに役立ちます。

第 4 章: Vivado 高位合成 (HLS)

第 4 章「Vivado 高位合成 (HLS)」では、ザイリンクスの Vivado HLS コンパイラを紹介します。この章では、2 章および 3 章からの概念に基づいて、C/C++ プログラムが FPGA にどのようにコンパイルされるのかを説明します。コンパイラが並列処理部分をどのように抽出し、メモリを構成し、FPGA 内の複数のプログラムを接続する方法を中心に説明します。

第 5 章: 計算中心のアルゴリズム

アルゴリズム解析に関する資料は多数ありますが、計算中心のアルゴリズムと制御中心のアルゴリズムとの違いは、ほとんどインプリメンテーション プラットフォームに依存しています。第 5 章「計算中心のアルゴリズム」では、FPGA の計算中心のアルゴリズムを定義し、具体例およびベスト プラクティスを示します。

第 6 章: 制御中心のアルゴリズム

制御中心のアルゴリズムは、プロセッサと FPGA のどちらでもインプリメントできます。インプリメンテーションの選択肢は、アルゴリズムに必要な応答時間によって異なります。第 6 章「制御中心のアルゴリズム」では、制御中心のアルゴリズムに関するインプリメンテーションの選択肢の概要を説明し、ユーザー データグラム プロトコル (UDP) のパケット プロセッシングのネットワーキング例を示します。

第 7 章: ソフトウェア検証および Vivado HLS

ほかのすべてのコンパイラと同様、Vivado HLS コンパイラの出力の質および正確さは入力ソフトウェアに左右されます。第 7 章「ソフトウェア検証および Vivado HLS」では、Vivado HLS コンパイラを使用する際に推奨されるソフトウェア テクニックを紹介します。一般的なコード記述エラー例とその Vivado HLS への影響、および各問題の解決策を示します。また、プログラムの動作が C レベルでは完全には検証できない場合にどうすればよいかを説明するセクションもあります。

第 8 章: 複数のプログラムの統合

ほとんどのプロセッサがアプリケーション実行のため複数のプログラムを実行するのと同様に、FPGA でも特定のアプリケーションを実行するために、複数のプログラムまたはモジュールを構築できます。第 8 章「複数のプログラムの統合」では、FPGA で複数のモジュールを接続する方法と、プロセッサを使用したこれらのモジュールの制御方法を説明します。FPGA ファブリックと Arm® Cortex™-A9 プロセッサを統合したザイリンクス Zynq®-7000 SoC に焦点を当てます。コンシューマーとプロデューサーの例を使用して、完全なシステム開発、統合、および設計のトレードオフについても説明します。

第 9 章: 完成したアプリケーションの検証

FPGA の場合、完成したアプリケーションからハードウェア システムが作成されます。このシステムは、FPGA ファブリックに 1 つまたは複数のモジュールを含めるのと同時に、プロセッサでコードを実行できます。第 9 章「完成したアプリケーションの検証」では、ターゲット アプリケーションを正しく実行するための推奨事項およびベスト プラクティスを示します。

XILINX

第 2 章

FPGA とは

概要

FPGA は、製造後に異なるアルゴリズムに合わせてプログラムできる集積回路 (IC) です。現在の FPGA デバイスは、ソフトウェア アルゴリズムをインプリメントするため、最高 200 万個のコンフィギュレーション可能なロジック セルで構成されています。従来の FPGA デザイン フローは、プロセッサよりも一般的な IC に近いものですが、FPGA には IC の開発と比べてコスト面で大きな利点があり、IC と同レベルのパフォーマンスを達成できます。IC と比較した場合の FPGA のもう 1 つの利点は、ダイナミックにリコンフィギュレーションできる点です。リコンフィギュレーションとは、プロセッサにプログラムを読み込むのと同じですが、FPGA ファブリックで使用可能なリソースの一部またはすべてに影響を与える可能性があります。

Vivado® HLS コンパイラを使用する際は、FPGA ファブリックで使用可能なリソースと、ターゲット アプリケーションを実行する際にそれらがどのようにかかわり合うのかの基礎を理解していることが重要です。この章では、Vivado HLS を使用してアルゴリズムに最適な計算アーキテクチャを作成するのに必要な FPGA の基本情報を示します。

FPGA アーキテクチャ

FPGA は、基本的に次のエレメントから構成されています。

- ルックアップ テーブル (**LUT**): 論理演算を実行します。
- フリップフロップ (**FF**): LUT の結果を格納します。
- ワイヤ: エレメントを相互に接続します。
- 入力/出力 (**I/O**) パッド: 物理的に使用可能なポートで、FPGA にデータを入力および FPGA からデータを出力します。

これらのエレメントを組み合わせたものが、図 2-1 に示す FPGA アーキテクチャです。この構造はどのアルゴリズムをインプリメントするにも十分ですが、結果のインプリメンテーションは計算スループット、必要なリソース、達成可能なクロック周波数の点で効率が制限されます。

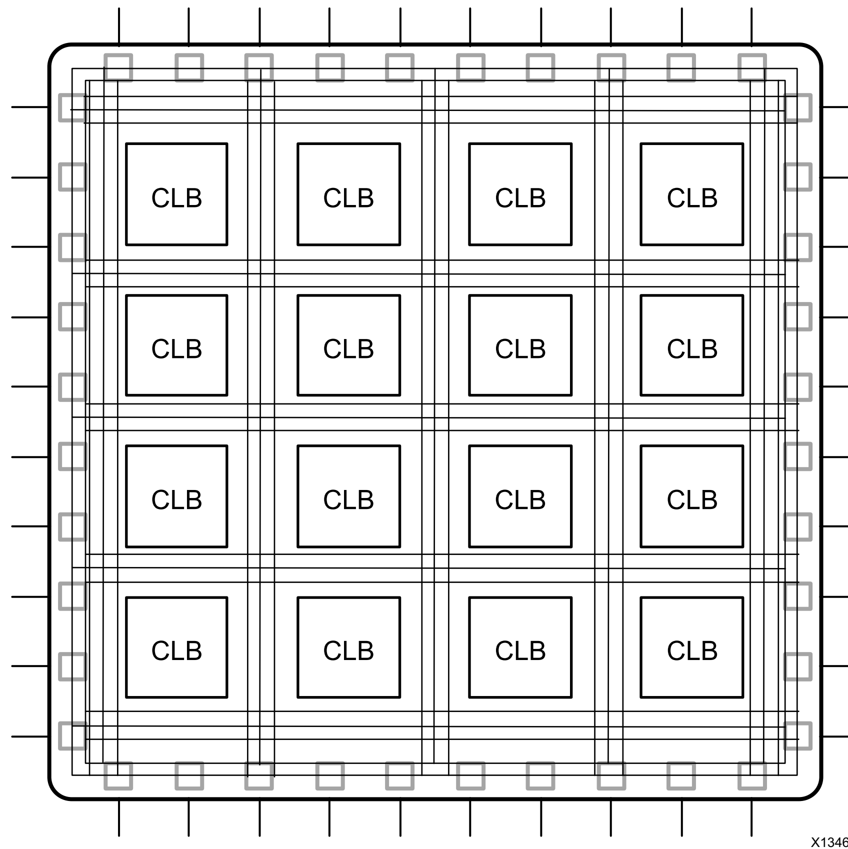

図 2-1: 基本 FPGA アーキテクチャ

現代の FPGA アーキテクチャには、基本エレメントだけでなく、デバイスの計算集積度および効率を高める追加の計算ブロックやデータ ストレージ ブロックが含まれています。これらの追加エレメントについてはこの後のセクションで説明しますが、次のようなものがあります。

- 分散データ ストレージ用のエンベデッド メモリ
- 異なるクロック レートで FPGA ファブリックを駆動するための位相ロック ループ (PLL)
- 高速シリアル トランシーバー
- オフチップ メモリ コントローラー
- 積和ブロック

これらのエレメントを組み合わせることにより、プロセッサで実行するソフトウェア アルゴリズムであればどんなものでも FPGA に柔軟にインプリメントできます。図 2-2 に、現代の FPGA アーキテクチャを示します。

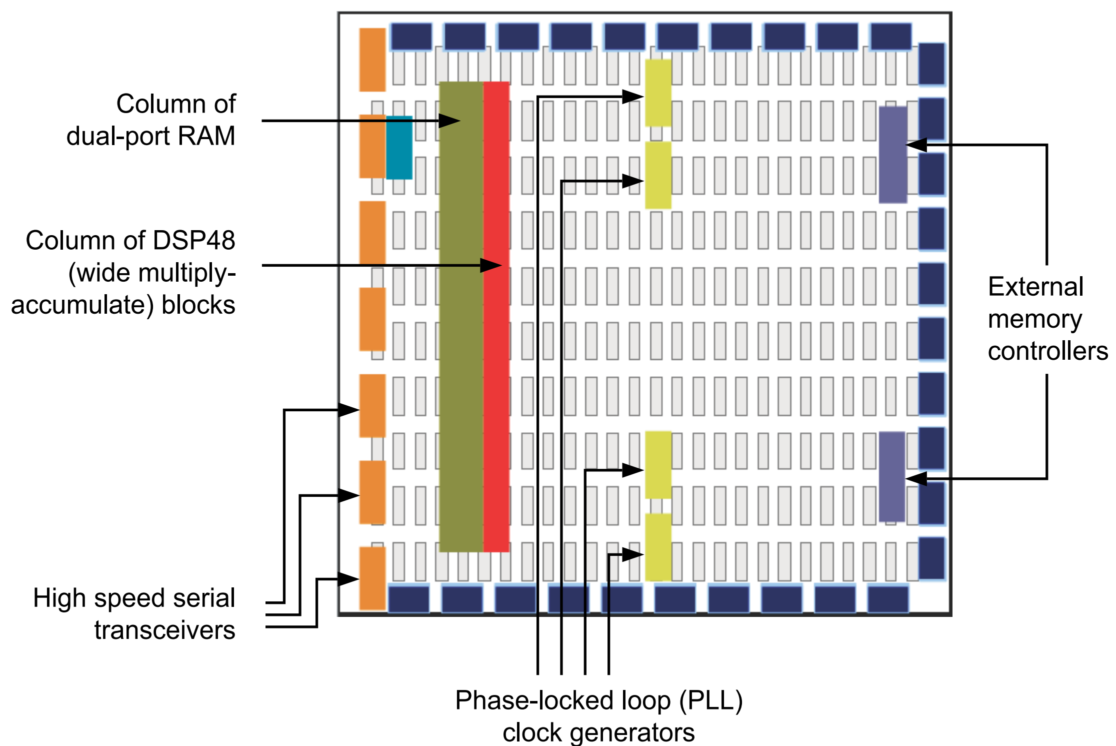

図 2-2: 現代の FPGA アーキテクチャ

LUT

LUT は FPGA の基本構築ブロックで、*N* 個のブール型変数のロジック関数をインプリメントできます。このエレメントは実質的には真理値表で、複数の入力を組み合わせてさまざまな関数をインプリメントし、出力値を生成します。真理値表のサイズは *N* で決まります (*N* は LUT の入力数)。一般的な *N* 入力 LUT の場合、この真理値表でアクセスできるメモリ ロケーションの数は次のようになります。

$$2^N \quad \text{式 2-1}$$

つまり、この真理値表でインプリメントできる関数の数は次のようになります。

$$2^{N^N} \quad \text{式 2-2}$$

注記: ザイリンクス FPGA デバイスの場合、*N* は通常 6 です。

LUT のハードウェア インプリメンテーションは、マルチプレクサーのセットに接続されたメモリ セルの集合体と考えることができます。LUT の入力は、最終的に結果値を選択するマルチプレクサーのセレクター ビットとして機能します。LUT は、関数の計算エンジンとデータ ストレージ エレメントのどちらにも使用できるので、これを念頭に置いておくことが重要です。図 2-3 に、LUT のこの機能表現を示します。

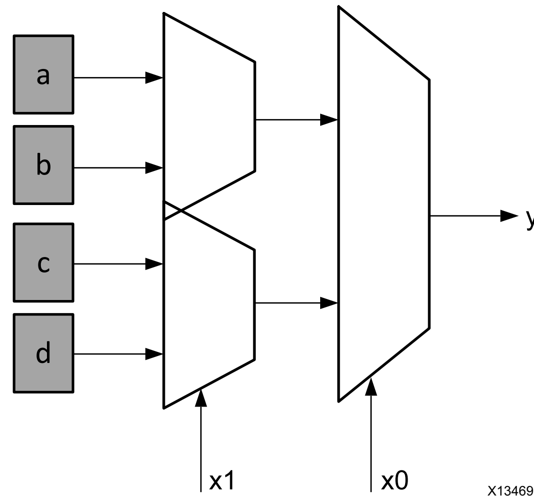

図 2-3: メモリ セルの集合体としての LUT による機能表現

フリップフロップ

フリップフロップは FPGA ファブリック内の基本ストレージ ユニットです。フリップフロップは、ロジックのパイプライン処理およびデータ格納を補助するため、常に LUT とペアになっています。フリップフロップの基本構造には、データ入力、クロック入力、クロック イネーブル、リセット、データ出力が含まれます。通常の動作では、データ入力ポートの値がラッチされ、クロックのパルスごとに出力に渡されます。クロック イネーブル ピンは、フリップフロップで特定の値を 2 クロック パルス間以上保持するために使用します。新しいデータ入力は、クロックおよびクロック イネーブルが 1 のときにのみラッチされ、データ出力ポートに渡されます。図 2-4 に、フリップフロップの構造を示します。

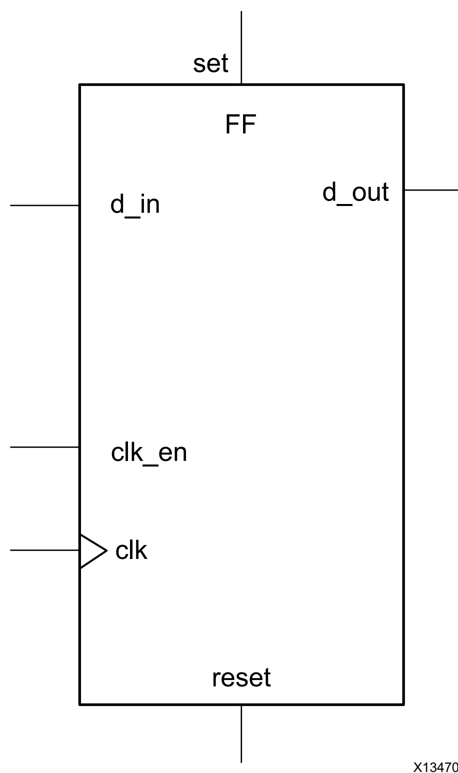

図 2-4: フリップフロップの構造

DSP ブロック

ザイリンクス FPGA で使用可能なエレメントの中で最も複雑な計算ブロックは、図 2-5 に示す DSP ブロックです。これは、FPGA ファブリックに組み込まれている論理演算ユニット (ALU) で、1 つのチェーンの中に 3 つの異なるブロックが含まれています。DSP の計算チェーンでは、加減算ユニットが乗算器に接続され、その乗算器が加算/減算/累算エンジンに接続されます。このチェーンを利用し、1 つの DSP ユニットで次の式の関数をインプリメントできます。

$$p = a \times (b + d) + c \quad \text{式 2-3}$$

または

$$p += a \times (b + d) \quad \text{式 2-4}$$

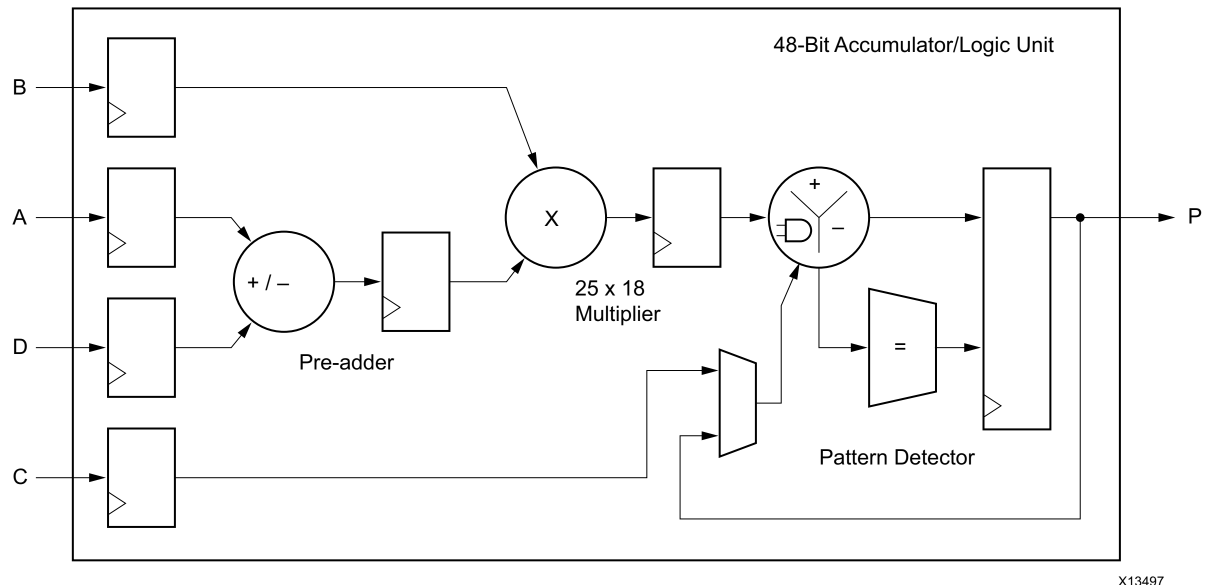

図 2-5: DSP ブロックの構造

ストレージ エレメント

FPGA デバイスにはエンベデッド メモリ エレメントが含まれており、ランダム アクセス メモリ (RAM)、読み出し専用メモリ (ROM)、またはシフト レジスタとして使用できます。メモリ エレメントには、ブロック RAM (BRAM)、UltraRAM ブロック (URAM)、LUT、シフト レジスタ (SRL) があります。

BRAM は、FPGA ファブリックにインスタンシエートされたデュアル ポート RAM モジュールで、比較的大きなデータ セットのオンチップ ストレージを提供します。デバイスで使用可能な BRAM には、18k または 36k ビットを格納できる 2 種類があります。使用可能なメモリの数はデバイスごとに異なります。これらのメモリはデュアル ポートなので、同じクロック サイクルで並行して複数のロケーションにアクセスできます。

C/C++ コードでの配列の記述方法によって、BRAM で RAM または ROM をインプリメントできます。RAM と ROM の唯一の違いは、データがストレージ エレメントにいつ書き込まれるかです。RAM コンフィギュレーションでは、回路の実行中いつでもデータを読み出したり書き込んだりできます。一方、ROM コンフィギュレーションでは、回路の実行中にはデータの読み出ししかできません。また、ROM のデータは FPGA コンフィギュレーションの一部として書き込まれるので、変更できません。

UltraRAM ブロックはデュアル ポート、288 Kb の同期 RAM で、深さ 4096 ビット、幅 72 ビットの固定コンフィギュレーションです。このブロックは UltraScale+ デバイスで使用でき、ストレージ量は BRAM の 8 倍です。

前に説明したように、LUT は小型のメモリで、デバイスのコンフィギュレーション中に真理値表の内容が書き込まれます。ザイリンクス FPGA の LUT 構造は柔軟性が高いので、LUT ブロックは 64 ビットのメモリとして使用でき、一般的に分散メモリと呼ばれています。LUT は FPGA デバイスで使用可能なメモリの中では最高速で、ファブリックのどこにでもインスタンシエートしてインプリメントされた回路のパフォーマンスを向上できます。

シフト レジスタは、レジスタをチェーン接続したものです。この構造は、フィルターなどにおいて、計算パスでデータを再利用するためのものです。たとえば、基本的なフィルターは乗算器のチェーンで構成され、データ サンプルを係数のセットで乗算します。入力データの格納にシフト レジスタを使用すると、データ サンプルはクロック サイクルごとにチェーンの次の乗算器に移動していきます。図 2-6 に、シフト レジスタの例を示します。

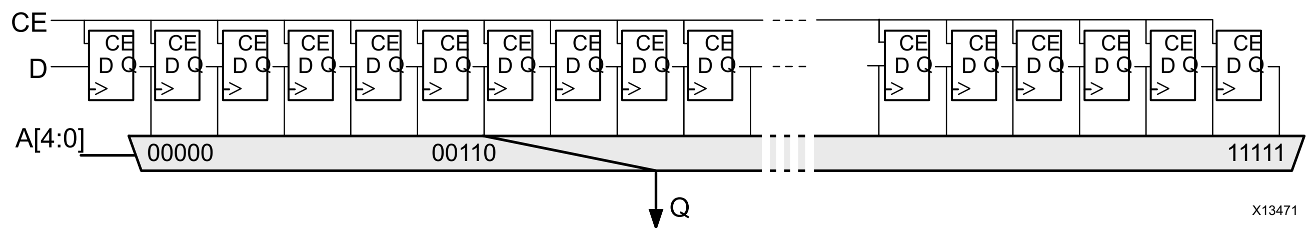

図 2-6: アドレス指定可能なシフト レジスタの構造

FPGA の並列処理 vs プロセッサ アーキテクチャ

FPGA ファブリックの構造では、アプリケーション実行をプロセッサ アーキテクチャよりも高い割合で並列処理できます。Vivado HLS コンパイラで生成されたソフトウェア プログラム用のカスタム プロセッシング アーキテクチャでは、実行パラダイムがプロセッサとは異なるので、アプリケーションをプロセッサから FPGA に移行する場合は、この点を考慮する必要があります。FPGA 実行パラダイムの利点を確認するため、このセクションではプロセッサのプログラム実行について簡単に説明します。

プロセッサでのプログラム実行

プロセッサでは、その種類にかかわらず、プログラムは命令のシーケンスとして実行され、それらの命令がソフトウェア アプリケーション用の計算に変換されます。この命令の順序は、GNU コンパイラ コレクション (GCC) などのプロセッサのコンパイラ ツールで生成されます。コンパイラ ツールにより、C/C++ で記述されたアルゴリズムがプロセッサにネイティブのアセンブリ言語コードに変換されます。プロセッサ コンパイラには、次の式の C 関数を入力します。

$$z = a + b; \quad \text{式 2-5}$$

この関数が次のアセンブリ コードに変換されます。

```
ADD $R1,$R2,$R3
```

図 2-7: アセンブリ コードで記述された計算

プロセッサの内部レジスタでは、図 2-7 のアセンブリ コードは *z* 値を算出するための加算を定義します。このコードでは、計算の入力値がレジスタ R1 および R2 に格納され、結果値がレジスタ R3 に格納されます。このコードは単純で、*z* 値の計算に必要なすべての命令が表現されているわけではありません。このコードは、データがプロセッサに到着してからの計算のみを実行します。そのため、中央のメモリからデータをプロセッサのレジスタに読み込んで、その結果をメモリにライトバックするためのアセンブリ言語命令をコンパイラで追加作成する必要があります。*z* 値を計算するアセンブリ プログラムは次のようになります。

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

図 2-8: Z 値を計算するアセンブリ プログラム

図 2-8 のコードは、2 つの値を足すという単純な演算でも、複数のアセンブリ命令になることを示しています。各命令の計算レイテンシは命令タイプによって異なります。たとえば、a と b の位置が変わると、LD 演算の完了までにかかるクロック サイクル数が変わります。値がプロセッサのキャッシュにある場合は、その読み込み操作は数十クロック サイクルで完了します。値がメインのダブル データ レート (DDR) メモリにある場合は、読み込み操作が完了するまでに数百から数千のクロック サイクルが必要になります。値がハード ドライブにある場合は、さらに時間がかかります。ソフトウェア エンジニアが、キャッシュ ヒット率を上げてプロセッサが命令ごとに費やす時間を短縮するため、アルゴリズムを再構築してメモリ内のデータの空間局所性を高めるのに多大な時間を費やすのは、このためです。

重要: ソフトウェア エンジニアが使用可能なプロセッサのキャッシュに合わせてアルゴリズムを再構築するのにかかる努力は、同じ演算を FPGA にインプリメントする場合は必要ありません。

FPGA でのプログラム実行

FPGA は本質的に並列処理に対応したファブリックであり、プロセッサ上で実行できるどんな演算関数でも FPGA にインプリメントできます。プロセッサとの主な違いは、ソフトウェア記述を RTL に変換するのに使用される Vivado HLS コンパイラがキャッシュおよび統合メモリ空間の制限を受けない点です。

z の計算は、Vivado HLS により、出力オペランドのサイズに合わせて複数の LUT にコンパイルされます。たとえば、元のソフトウェア プログラムでは、変数 *a*、*b*、および *z* のデータ型が short で定義されているとします。16 ビットのデータ コンテナーを定義するこのデータ型は、Vivado HLS により 16 個の LUT としてインプリメントされます。

注記: 原則として、LUT 1 個が計算の 1 ビットに相当します。

z の計算に使用される LUT はこの演算専用です。あらゆる計算で同じ ALU を共有するプロセッサとは異なり、FPGA のインプリメンテーションでは、ソフトウェア アルゴリズムの計算ごとに独立した LUT セットがインスタンシエートされます。

計算ごとに独立した LUT リソースを割り当てるだけでなく、FPGA のメモリ アーキテクチャとメモリ アクセスのコストもプロセッサとは異なります。FPGA のインプリメンテーションでは、Vivado HLS コンパイラにより、メモリが複数のストレージ バンクに配置され、そのメモリを使用する演算のできるだけ近くになるようにします。こうすることで、プロセッサの機能をはるかに上回る広いメモリ帯域幅を即座に実現できます。たとえば、ザイリンクスの Kintex®-7 410T デバイスでは、合計 1,590 18 キロ ビットの BRAM が使用可能です。このデバイスのメモリ レイアウトでは、レジスタ レベルで 1 秒ごとに 0.5M ビット、BRAM レベルで 1 秒ごとに 23T ビットのメモリ帯域幅を達成できます。

計算スループットおよびメモリ帯域幅に関しては、Vivado HLS コンパイラは、スケジューリング、パイプライン処理、データフローなどのプロセスを使用することにより FPGA ファブリックの機能を活用します。これらのプロセスはユーザーには透過的ですが、ソフトウェア アプリケーションの最適な回路レベル インプリメンテーションを達成するために、不可欠なソフトウェア コンパイル プロセスです。

スケジューリング

スケジューリングは、異なる演算間のデータおよび制御の依存性を特定し、どの演算をいつ実行するかを決定するプロセスです。従来の FPGA デザインでは、スケジューリングは手動のプロセスで、ハードウェア インプリメンテーション用のソフトウェア アルゴリズムの並列処理とも呼ばれています。

Vivado HLS では、隣接する演算間の依存関係だけでなく、時間の依存関係も解析されます。このため、同じクロック サイクルで実行できる演算はグループにまとめられ、関数呼び出しがオーバーラップするようハードウェアが設定されます。関数呼び出しをオーバーラップさせることにより、現在の関数呼び出しを完了してからでないと同じ演算セットへの次の関数呼び出しを実行できないというプロセッサの制限が取り除かれます。このプロセスは「パイプライン処理」と呼ばれ、次のセクションおよび残りの章で詳細に説明します。

パイプライン処理

パイプライン処理は、アルゴリズムのハードウェア インプリメンテーションにおけるデータの依存性を回避し、並列処理レベルを上げるためのデジタル デザイン手法です。等価の関数が得られるように元のソフトウェア インプリメンテーションのデータ依存性は保持されますが、必要な回路は独立した段のチェーンに分割されます。チェーンのすべての段は、同じクロック サイクルで並列実行されます。唯一の違いは各段のデータのソースで、計算の各段は、前のクロック サイクルで前の段で計算された結果からデータ値を受信します。たとえば、次の関数を計算するため、Vivado HLS コンパイラにより、乗算器ブロックが 1 つ、加算器ブロックが 2 つインスタンシエートされます。

$$y = (a \times x) + b + c \qquad \text{式 2-6}$$

図 2-9 に、この計算の構造とパイプライン処理の効果を示します。関数例の 2 つのインプリメンテーションを示します。上のインプリメンテーションは、パイプライン処理を使用せずに y を計算するのに必要なデータパスです。このインプリメンテーションは、C/C++ 関数と同じように動作し、すべての入力値が計算開始時にわかっている必要があり、1 回に計算される結果値 y は 1 つのみです。下のインプリメンテーションは、同じ回路をパイプライン処理したものです。

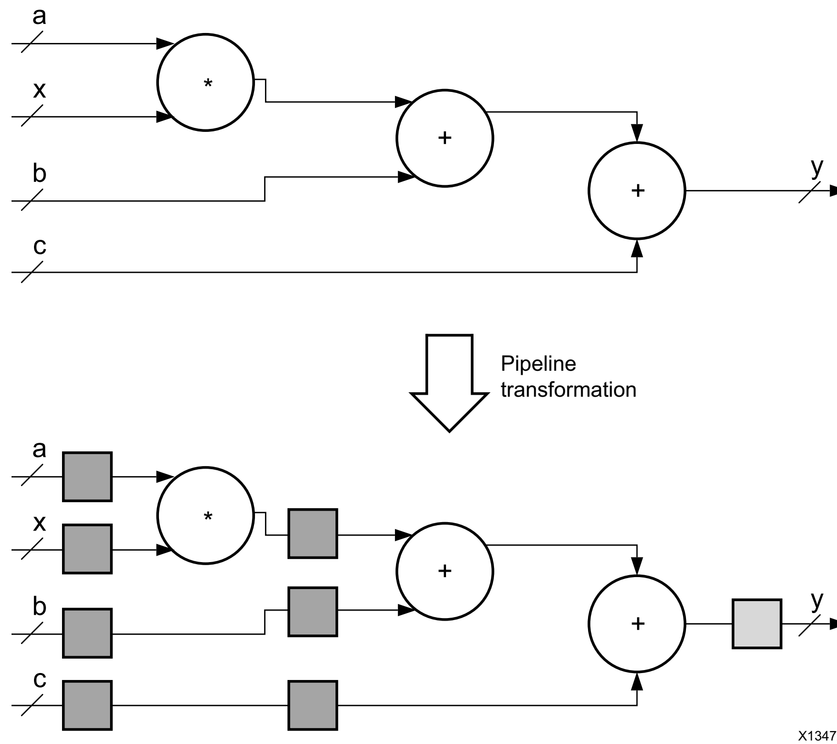

図 2-9: 計算関数の FPGA インプリメンテーション

図 2-9 のデータパスの正方形は、FPGA ファブリックでフリップフロップ ブロックでインプリメントされるレジスタを表わします。各正方形は 1 クロック サイクルとしてカウントされます。パイプライン処理バージョンでは、各結果値 *y* の計算に 3 クロック サイクルかかります。レジスタを追加することにより、各ブロックは時間的に複数の計算セクションに分けられます。乗算器のあるセクションと 2 つの加算器のあるセクションを並列して実行できるので、関数の全体的な計算レイテンシが削減されます。データパスの 2 つのセクションを並列実行することにより、ブロックは実質的に値 *y* と *y'* を並行して計算します (*y'* は式 2-6 の次の実行の結果値)。最初の *y* の計算 (「パイプライン充填時間」とも呼ばれる) に 3 クロック サイクルかかります。最初の計算の後は、計算パイプラインで現在とその次の *y* の計算がオーバーラップしているので、各クロック サイクルで *y* の新しい値が出力されます。

図 2-10 に示すパイプライン アーキテクチャでは、生のデータ (濃いグレー)、途中まで計算されたデータ (白)、最終データ (薄いグレー) が同時に存在し、各段の結果がその段のレジスタ セットに格納されます。このような計算のレイテンシは複数サイクルになりますが、新しい結果値を毎クロック サイクル計算できます。

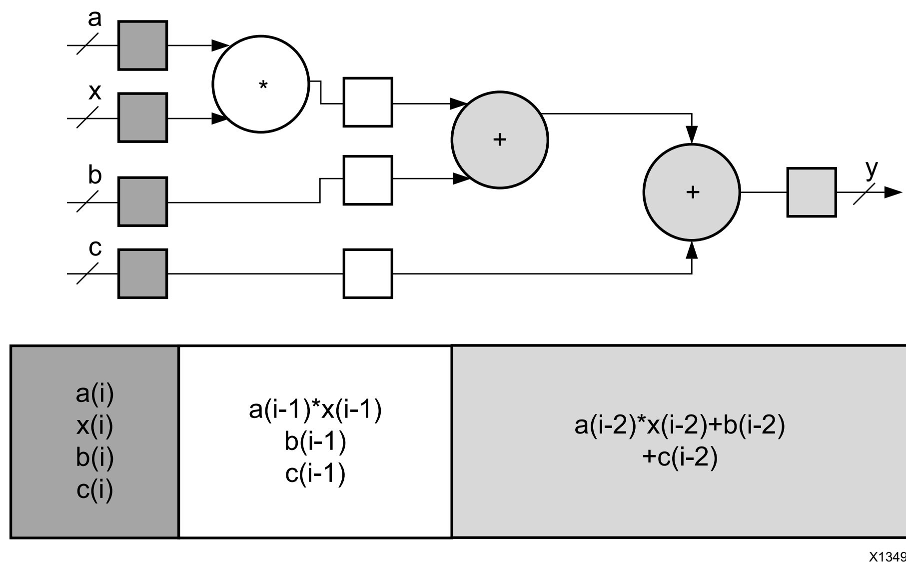

図 2-10: パイプライン アーキテクチャ

データフロー

データフローもデジタル デザインのテクニックの 1 つで、概念的にはパイプライン処理に似ています。データフローは、粗粒度での並列処理を実行します。ソフトウェア実行では、データフローは 1 プログラム内の関数の並列実行に利用されます。

Vivado HLS では、プログラムの異なる関数の相互関係が入力と出力に基づいて評価され、データフロー レベルの並列処理が抽出されます。並列処理で最も単純なのは、複数の関数が異なるデータ セットを処理し、データをやり取りしない場合です。この場合、Vivado HLS により、各関数に FPGA ロジック リソースが割り当てられ、ブロックはそれぞれ独立して実行されます。1 つの関数の結果が別の関数に供給されると複雑になりますが、それはソフトウェア プログラムでは普通です。これは「コンシューマーとプロデューサーの依存関係」と呼ばれます。

Vivado HLS では、このコンシューマーとプロデューサーの依存関係の使用モデルが 2 つサポートされています。最初のユース モデルでは、コンシューマーが演算を開始する前に、プロデューサーがデータ セットを作成します。並列処理は、BRAM メモリのペアを 2 つのメモリ バンクに配置することにより達成します。各関数は、1 回の関数呼び出しでは、いずれかのメモリ バンクにのみアクセスできます。新しい関数呼び出しが開始すると、Vivado HLS で生成された回路により、プロデューサーとコンシューマーのメモリ接続を切り替えます。この方法であれば、プログラムは機能的には正しく動作しますが、関数呼び出し間で達成できる並行処理のレベルには限界があります。

もう 1 つの使用モデルでは、コンシューマーはプロデューサーからの部分的な結果を使用して計算を開始できるので、達成可能な並行処理のレベルが関数呼び出し内の実行にも拡張されます。両方の関数用に Vivado HLS で生成されたモジュールは、FIFO メモリ回路を使用して接続されます。このメモリ回路はソフトウェア プログラミングのキューとして動作し、モジュール間をデータレベルで同期化させます。1 回の関数呼び出しの間に、両方のハードウェア モジュールがそれぞれプログラムを実行します。ただし、コンシューマー モジュールは、プロデューサーからのデータをある程度待ってから計算を開始します。Vivado HLS の用語では、このコンシューマー モジュールの待機時間は「間隔」または「開始間隔 (II)」と呼ばれます。

XILINX

第 3 章

ハードウェア デザインの基本概念

概要

プロセッサと FPGA とでは、プロセッシング アーキテクチャが固定されているかどうかが主な違いの 1 つです。この違いは、各ターゲットに対するコンパイラの動作に直接影響を与えます。プロセッサでは、計算アーキテクチャは固定されており、コンパイラの仕事はソフトウェア アプリケーションを使用可能なプロセッシング構造にどのように収めるかを決定することです。パフォーマンスは、アプリケーションをプロセッサの機能にどれだけうまくマップできるかと、正しく実行するのに必要なプロセッサ命令の数によります。

FPGA では、多数の構築ブロックで構成された白紙のキャンバスに似ています。Vivado® HLS コンパイラの仕事は、これらの構築ブロックからソフトウェア プログラムがうまくフィットするプロセッシング アーキテクチャを作成することです。Vivado HLS コンパイラを制御して、最良のプロセッシング アーキテクチャを作成するには、ハードウェアの設計概念の基礎知識が必要になります。

この章では、FPGA とプロセッサ ベースのデザイン両方に当てはまる一般的な設計概念と、これらの概念が互いにどう関連しているのかを説明します。この章では、FPGA デザインの詳細は説明しません。プロセッサのコンパイラと同様に、Vivado HLS コンパイラもアルゴリズムの FPGA ロジック ファブリックへのインプリメンテーションに関する下位情報を処理します。

クロック周波数

アルゴリズムを実行するプラットフォームを決める際、まず検討すべきものの 1 つはプロセッサのクロック周波数です。一般的に、クロック周波数が高ければ、アルゴリズムの実行も高速になります。どのプロセッサを選ぶべきかを検討しているのであれば、クロック周波数を優先的に考えるのは妥当ですが、プロセッサか FPGA かの選択では、クロック周波数だけで判断するのは間違いのもとです。

これは、プロセッサと FPGA の公称クロック周波数の違いに関連しています。たとえば、プロセッサと FPGA のクロック周波数を比較する場合、表 3-1 のような比較を見せられることがあります。

表 3-1: 最大クロック周波数の比較例

プロセッサ	FPGA
2 GHz	500 MHz

表 3-1 の値を単純に比べると、プロセッサのパフォーマンスは FPGA の 4 倍であるように見えます。この単純な比較では、プロセッサと FPGA の違いはクロック周波数のみであるという誤解を招きかねません。実際には、ほかにも違いはあります。

まず、プロセッサと FPGA とでは、ソフトウェア プログラムの実行方法が異なります。プロセッサは、一般的なハードウェア プラットフォーム上でどんなプログラムも実行できます。この一般的なプラットフォームがプロセッサの中核であり、固定のアーキテクチャが定義されていて、そこにすべてのソフトウェアをフィットさせる必要があります。プロセッサのコンパイラは、プロセッサのアーキテクチャを理解しており、ユーザー ソフトウェアを命令セットにコンパイルします。結果の命令セットは、図 3-1 に示すように、常に同じ順序で実行されます。

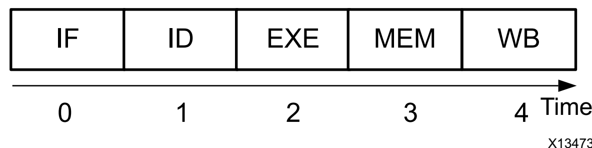

図 3-1: プロセッサの命令実行段階

標準プロセッサであっても、特殊プロセッサであっても、命令の実行は常に同じです。ユーザー アプリケーションの各命令は、すべて次の段階を経過する必要があります。

1. 命令フェッチ (IF)
2. 命令デコード (ID)
3. 実行 (EXE)
4. メモリ操作 (MEM)
5. ライトバック (WB)

表 3-2 に、各段の目的を示します。

表 3-2: 命令の処理段階

段階	説明
IF	プログラム メモリから命令を取り出します。
ID	命令をデコードして演算および演算子を決定します。
EXE	使用可能なハードウェア上で命令を実行します。標準プロセッサの場合は、論理演算ユニット (ALU) または浮動小数点ユニット (FPU) です。特殊プロセッサの場合は、命令処理のこの段階で標準プロセッサの機能に固定機能アクセラレータが追加されます。
MEM	メモリ操作を使用して、次の命令のデータをフェッチします。
WB	命令結果をローカル レジスタまたはグローバル メモリに書き込みます。

現代のほとんどのプロセッサには命令実行パスのコピーが複数含まれているので、それらをある程度オーバーラップさせて命令を実行することが可能です。プロセッサの命令は通常相互依存しているため、命令実行ハードウェアのコピーをオーバーラップさせるにしても、完璧とはいきません。うまくいったとしても、プロセッサを使用することで追加されるオーバーヘッド段のみをオーバーラップさせることができる程度です。アプリケーションの計算が実行される EXE 段では、計算が順次実行されます。これは、EXE 段のリソースには限りがあり、命令どうしが依存しているからです。

図 3-2 に、複数の命令をある程度並列実行するプロセッサを示します。これは、すべての命令をできる限り速く実行するプロセッサでのベスト ケースです。このベスト ケースでも、プロセッサはクロック サイクルごとに EXE 段を 1 段しか処理できません。つまり、ユーザー アプリケーションは、クロック サイクルごとに 1 演算ずつ進みます。コンパイラで 5 つの EXE 段をすべて並列実行できると判断されたとしても、このプロセス構造では達成できません。

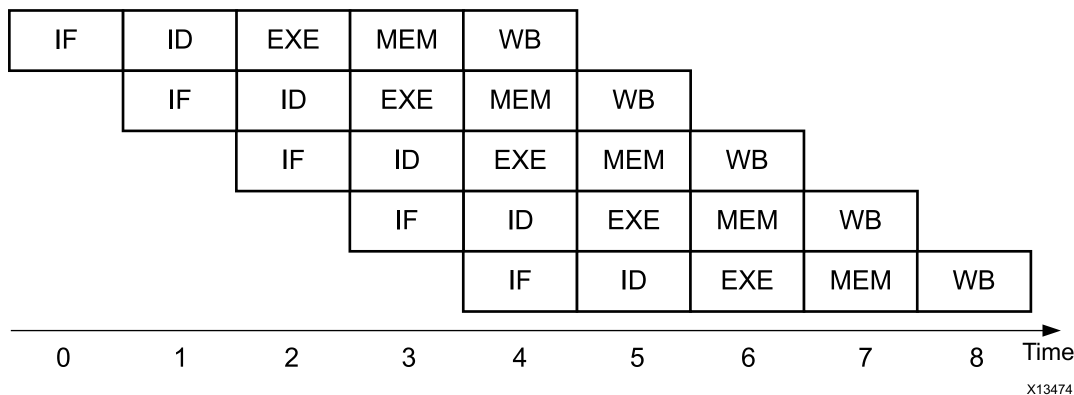

図 3-2: 複数の命令実行ユニットのあるプロセッサ

FPGA では、すべてのソフトウェアが一般的な計算プラットフォームで実行されるわけではありません。1 回に 1 つのプログラムがそのプログラムのカスタム回路で実行されます。このため、ユーザー アプリケーションを変更すると FPGA の回路も変更されます。FPGA での処理では、図 3-1 とは異なり、EXE 段は 図 3-3 のようになります。MEM 段があるかどうかはアプリケーションによります。

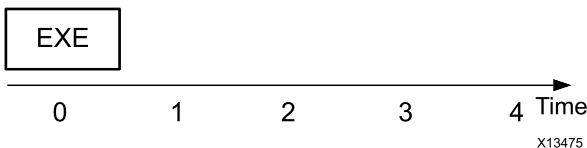

図 3-3: FPGA の命令実行段階

このように FPGA には柔軟性があるため、Vivado HLS コンパイラはプラットフォームのオーバーヘッド段を考慮する必要はなく、命令を最大限に並列実行する方法が検索されます。図 3-2 と同じ処理を考えると、FPGA で同じソフトウェアを実行する場合のプロファイルは図 3-4 のようになります。

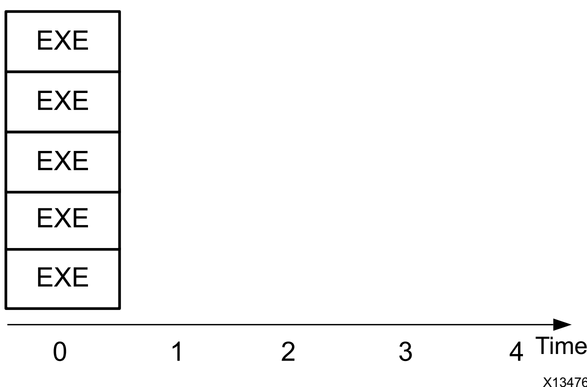

図 3-4: 複数の命令実行ユニットのある FPGA

図 3-2 と図 3-4 を比較すると、FPGA のパフォーマンスはプロセッサの 9 倍です。実際の数値はアプリケーションによって異なりますが、計算負荷の高いアプリケーションでは一般的に、FPGA のパフォーマンスはプロセッサの少なくとも 10 倍になります。

さらに、クロック周波数に固執していると見過ごしがちになるのが、ソフトウェア プログラムの消費電力です。消費電力は次の式で概算できます。

$$P = \frac{1}{2}cFV^2 \quad \text{式 3-1}$$

式 3-1 からわかるように、消費電力とクロック周波数の関係は、実験データでも裏付けられていますが、計算量が同じであれば、プロセッサのほうが FPGA よりも消費電力が大きくなります。FPGA では、ソフトウェア プログラムごとにカスタム回路が作成されるので、プロセッサのような命令解釈のオーバーヘッドなしで、プロセッサよりも低いクロック周波数で演算を最大限に並列実行できます。

推奨: プロセッサを使用するか FPGA を使用するかを検討している場合は、最大クロック周波数を比較するのではなく、スループットおよびレイテンシに基づいて、アプリケーション要件および計算量を解析することを推奨します。

レイテンシおよびパイプライン処理

レイテンシとは、命令または命令セットを完了させてアプリケーションの結果値の生成するまでにかかるクロック サイクル数を指します。図 3-1 の基本プロセッサ アーキテクチャの場合、1 つの命令のレイテンシは 5 クロック サイクルです。アプリケーションに 5 つの命令がある場合、この単純なモデルの全体的なレイテンシは 25 クロック サイクルになります。つまり、アプリケーションの結果は 25 クロック サイクル経過するまで得られません。

FPGA とプロセッサの両方において、アプリケーションのレイテンシはパフォーマンスの重要な指標です。どちらの場合も、レイテンシに問題があれば、パイプライン処理を利用して解決されます。プロセッサの場合、パイプライン処理を使用すると、現在の命令が完了する前に、次の命令を開始することが可能です。つまり、命令セットの処理に必要なオーバーヘッド段をオーバーラップさせることができます。プロセッサでパイプライン処理を使用した場合のベスト ケースは、図 3-2 に示されています。プロセッサでは、命令実行をオーバーラップさせることにより、5 つの命令を含むアプリケーションで 9 クロック サイクルのレイテンシを達成できます。

FPGA の場合、命令処理に関するオーバーヘッド サイクルは存在せず、レイテンシは元のプロセッサ命令の EXE 段の実行に何クロック サイクルかかるかで計測されます。図 3-3 では、レイテンシは 1 クロック サイクルです。並行処理もレイテンシにおいて重要な役割を果たします。5 つの命令を含むアプリケーションの場合、図 3-4 に示すように、FPGA のレイテンシは 1 クロック サイクルです。FPGA でのレイテンシは 1 クロック サイクルなので、なぜパイプライン処理に利点があるのか、わかりにくいかもしれません。FPGA でのパイプライン処理も、プロセッサの場合と同じで、アプリケーションのパフォーマンスを向上することが目的です。

前にも説明したように、FPGA は構築ブロックで構成された白紙のキャンバスのようなもので、アプリケーションをインプリメントするにはこれらのブロックを接続する必要があります。Vivado HLS コンパイラは、これらのブロックを直接、またはレジスタを介して接続します。図 3-5 に、図 3-3 の EXE 段を 5 つの構築ブロックを使用してインプリメントした場合を示します。

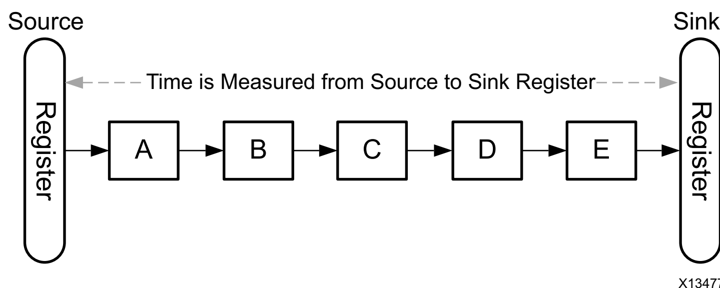

図 3-5: パイプライン処理が使用されていない FPGA インプリメンテーション

FPGA での操作時間は、ソース レジスタからシンク レジスタまで信号が移動するのにかかる時間です。図 3-5 の各構築ブロックの実行に 2 ns 必要だとすると、現在のデザインではこの機能を実行するのに 10 ns かかります。レイテンシは 1 クロック サイクルのままですが、クロック周波数は 100 MHz に制限されます。この 100 MHz という制限は、FPGA でのクロック周波数の定義に由来します。FPGA 回路では、クロック周波数は、ソース レジスタからシンク レジスタまで信号が移動する最長時間として定義されます。

FPGA でのパイプライン処理は、大きな計算ブロックを小さいセグメントに分割するため、さらにレジスタが挿入されます。計算を分割していくと、クロック サイクル数で表されるレイテンシは増加しますが、カスタム回路をより高いクロック周波数で実行できるようになるので、パフォーマンスが向上します。

図 3-6 に、図 3-5 のプロセッシング アーキテクチャを完全にパイプライン処理した後のインプリメンテーションを示します。「完全にパイプライン処理した」というのは、FPGA 回路の各構築ブロック間にレジスタが 1 つずつ挿入されたという意味です。レジスタの追加により、回路のタイミング要件は 10 ns から 2 ns に変更されるので、最大クロック周波数は 500 MHz になります。さらに、計算をレジスタで区切られた領域に分割することで、各ブロックが常に稼働状態となり、アプリケーション スループットが向上します。

ただし、パイプライン処理には問題が 1 つあります。回路のレイテンシです。図 3-5 の元の回路には、低いクロック周波数で 1 クロック サイクルのレイテンシがあります。図 3-6 の回路には、高いクロック周波数で 5 クロック サイクルのレイテンシがあります。

重要: パイプライン処理によって発生するレイテンシは、FPGA の設計中に検討すべきトレードオフの 1 つです。

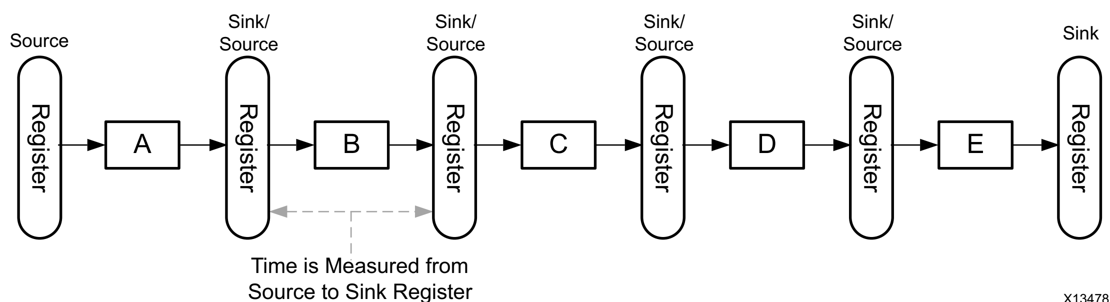

図 3-6: パイプライン処理を使用した FPGA インプリメンテーション

スループット

スループットも、インプリメンテーションの全体的なパフォーマンス指標の 1 つです。スループットとは、プロセッシング ロジックが次のデータ サンプルを受信できるようになるまでにかかるクロック サイクル数です。ただし、回路のクロック周波数によりスループット値の意味が変わることを思い出してください。

たとえば、図 3-5 および図 3-6 は、入力データ サンプル間に 1 クロック サイクルを必要とするインプリメンテーションです。主な違いは、図 3-5 のインプリメンテーションでは入力サンプル間に 10 ns 必要で、図 3-6 のインプリメンテーションでは 2 ns だけであるという点です。これを考慮すると、2 番目のインプリメンテーションの方が、入力データ レートが高いので、高いパフォーマンスが得られるのは明らかです。

注記: このセクションのスループットの定義は、プロセッサ上で実行しているアプリケーションを解析するときにも適用可能です。

メモリのアーキテクチャおよびレイアウト

選択したインプリメンテーション プラットフォームのメモリ アーキテクチャは、物理エレメントの 1 つで、ソフトウェア アプリケーションのパフォーマンスに影響する可能性があります。達成可能なパフォーマンスの上限は、メモリ アーキテクチャによって決まります。プロセッサまたは FPGA 上のアプリケーションは、使用可能な計算リソースの種類や数にかかわらず、すべてあるパフォーマンスでメモリ バウンドとなります。FPGA デザインでは、メモリ バウンドがどこで発生するのか、データ レイアウトやメモリ構成の影響をどのように受けるのかを理解しておくと有益です。

プロセッサ ベースのシステムでは、プロセッサのタイプにかかわらず、実質的に同じメモリ アーキテクチャにアプリケーションをフィットさせる必要があります。この共通性により、プロセッサ間でのアプリケーションの移行プロセスは単純です。ソフトウェア エンジニアにとって馴染みのある共通メモリ アーキテクチャは、プロセッサへのデータ送信に必要なクロック サイクル数に基づいて、低速、中速、または高速のメモリで構成されています。表 3-3 に、これらのメモリの分類を示します。

表 3-3: メモリ タイプの定義

メモリ タイプ	定義
低速	ハード ドライブのような大容量ストレージ デバイス
中速	DDR メモリ
高速	オンチップ キャッシュ メモリ (サイズはプロセッサによって異なる)

この表のメモリ アーキテクチャでは、1 つの大型メモリ空間を想定しています。このメモリ空間内で、プログラム データを格納するための領域を割り当てたり、割り当てを解除したりします。データの物理的な位置と、データが階層内をどう移動するかは、計算プラットフォームで管理され、ユーザーには透過的です。この種のシステムでは、パフォーマンスを高めるには、できる限りキャッシュのデータを再利用するしかありません。

そのため、ソフトウェア エンジニアは、キャッシュ トレースを確認したり、データ局所性を増加するためにソフトウェア アルゴリズムを再構築したり、プログラムの即時メモリ フットプリントを最小限に抑えるためメモリ割り当てを管理したりすることに時間を費やす必要があります。プロセッサを変えてもこれらのテクニックはすべて有効ですが、結果は異なります。最大限のパフォーマンスを得るため、ソフトウェア プログラムを実行するプロセッサに合わせて調整する必要があります。

プロセッサ ベースのメモリで作業した経験のあるソフトウェア エンジニアが FPGA のメモリで作業し始めると、FPGA には固定されたオンチップ メモリ アーキテクチャがないという違いに直面します。FPGA ベースのシステムは低速および中速メモリに接続できますが、使用可能な高速メモリは大きく異なります。既存のキャッシュを最大限に活用するようソフトウェアを再構築するのではなく、Vivado HLS コンパイラでアルゴリズムのデータ レイアウトに最適な高速メモリ アーキテクチャが構築されます。結果の FPGA インプリメンテーションには、サイズの異なる 1 つまたは複数の内部バンクがあり、それぞれ個別にアクセスできます。

図 3-7 のコード例に、プログラムのメモリ要件を満たすためのベスト プラクティスを示します。

Processor Code

```
void foo(......)
{
   int *A = (int *)malloc(10 * sizeof(int));
   ....
   free(A);
}
```

FPGA Code

```
void foo(......)
{
   int A[10];
   ....

}
```

図 3-7: プロセッサおよび FPGA コード例

経験豊富なソフトウェア エンジニアは、FPGA コードではダイナミック メモリ割り当てがないことに驚きを感じるかもしれません。プロセッサベースのシステムでは、基本メモリ アーキテクチャが固定されているので、ダイナミックメモリ割り当てはベスト プラクティスとして長く利用されてきました。

これとは対照的に、Vivado HLS コンパイラでは、アプリケーションに合わせたメモリ アーキテクチャが構築されます。構築されるメモリ アーキテクチャは、プログラムのメモリ ブロック サイズと、プログラム実行時にデータがどのように使用されるかによって決まります。Vivado HLS などの最新の FPGA コンパイラでは、アプリケーションのメモリ要件がコンパイル時に完全に解析可能であることが求められます。

Vivado HLS のスタティックなメモリ割り当ての利点は、配列 A のメモリを複数の方法でインプリメントできる点です。アルゴリズムでの計算によりますが、Vivado HLS コンパイラでは、A のメモリをレジスタ、シフト レジスタ、FIFO、または BRAM としてインプリメントできます。

注記: Vivado HLS コンパイラでは、ダイナミックメモリ割り当ては制限されていますが、その代わり、ポインターが完全サポートされています。ポインターの詳細は、第 4 章の「ポインター」を参照してください。

レジスタ

できる限り高速のメモリ構造にするには、メモリをレジスタとしてインプリメントします。このインプリメンテーションでは、A の各要素はそれぞれ独立した要素になります。各要素は計算に組み込まれ、ロジックのアドレス指定や追加の遅延なしで使用されます。

シフト レジスタ

プロセッサ プログラミングの観点から言うと、シフト レジスタは特殊なキューと考えることができます。このインプリメンテーションでは、A の各要素は計算の異なる部分で繰り返し使用されます。シフト レジスタの主な特徴は、A のどの要素にでも毎クロック サイクル アクセスできることです。隣接する次のストレージ コンテナーにデータをすべて移動させるのに、1 クロック サイクルしかかかりません。

FIFO

FIFO は、入力が 1 つ、出力が 1 つのキューだと考えることができます。このような構造は通常、プログラムのループ間または関数間でデータを送信するために使用されます。FIFO にはアドレス指定ロジックはないので、インプリメンテーションの詳細はすべて Vivado HLS コンパイラにより処理されます。

BRAM

BRAM は、FPGA ファブリックに組み込まれているランダム アクセス メモリです。ザイリンクスの FPGA デバイスには、これらのエンベデッド メモリが多数含まれていますが、その数はデバイスによって異なります。プロセッサ プログラミングの観点から言うと、この種のメモリは次の特徴を持つキャッシュだと考えることができます。

- プロセッサ キャッシュでは一般的なキャッシュ コヒーレンシ、衝突、キャッシュ ミス追跡ロジックはインプリメントされない。
- デバイスに電源が入っている間のみ値を保持。
- 2 つの異なるメモリ ロケーションに並列同時サイクルでアクセス可能。

XILINX

第 4 章

Vivado 高位合成 (HLS)

概要

ザイリンクス Vivado® 高位合成 (HLS) コンパイラは、標準および特殊プロセッサの両方でアプリケーションを開発する場合と同様のプログラミング環境を提供します。Vivado HLS は、C/C++ プログラムの処理、解析、最適化の主要テクノロジをプロセッサ コンパイラとを共有しています。主な違いは、アプリケーションの実行ターゲットです。

実行ハードウェアとして FPGA ファブリックをターゲットにすることにより、Vivado HLS では、1 つのメモリ空間および限られた計算リソースによるパフォーマンスのボトルネックの問題に対処する必要なく、スループット、消費電力、レイテンシに基づいてコードを最適化できます。これにより、機能のデモ用だけでなく、計算負荷の高いソフトウェア アルゴリズムを実際の製品にインプリメントできます。この章では、VivadoHLS コンパイラがどのように機能するのか、従来のソフトウェア コンパイラとはどう違うのかを説明します。

Vivado HLS コンパイラをターゲットとするアプリケーション コードでは、プロセッサ コンパイラと共通したカテゴリが使用されます。Vivado HLS では、すべてのプログラムで次が解析されます。

- 演算
- 条件文
- ループ
- 関数

重要: Vivado HLS は、ほとんどどんな C/C++ プログラムでもコンパイルできます。Vivado HLS でのコード処理における唯一の制限は、1 つのメモリ空間を持つプロセッサで使用されるダイナミック言語コンストラクトです。この章で説明しますが、Vivado HLS を使用する場合、注意が必要なダイナミック コンストラクトは主にメモリ割り当てとポインターです。

演算

演算とは、アプリケーションの結果値の計算に関わる算術部分と論理部分の両方を指します。比較文は「条件文」で処理されるので、この定義からは意図的に除外しています。

演算における Vivado HLS とプロセッサ コンパイラの主な違いは、制限事項です。プロセッサ コンパイラでは、固定プロセッシング アーキテクチャが使用されるので、演算の依存性を制限するか、メモリ レイアウトを調整してキャッシュ パフォーマンスを最大化することでしか、パフォーマンスを改善することはできません。Vivado HLS には固定プロセッシング アーキテクチャがないので、ユーザー入力に基づいてアルゴリズム特定のプラットフォームが構築されます。このため、このセクションの例で説明するように、アプリケーションのパフォーマンスをスループット、レイテンシ、消費電力の面から改善できます。

図 4-1 に、結果値 `F[i]` を計算する 3 つの演算を示します。

```
A[i] = B[i] * C[i];
D[i] = B[i] * E[i];
F[i] = A[i] + D[i];
```

図 4-1: 3 つの演算のサンプル コード

プロセッサを使用すると、実行プロファイルは図 4-2 のようになります。このアプリケーション プロファイルは、中央演算処理装置 (CPU) での命令処理の EXE 段のみに焦点を当てています。プロセッサと FPGA で共有されているのは、この命令処理段だけです。この例では、実行トレースは、アルゴリズムに基づいてではなく、実行プラットフォームにより順次になります。アルゴリズムに基づけば、`A[i]` と `D[i]` の値は任意の順序でまたは同時に計算できます。アルゴリズム上の唯一の制限は、これらの値が `F[i]` の前に計算する必要があるということです。

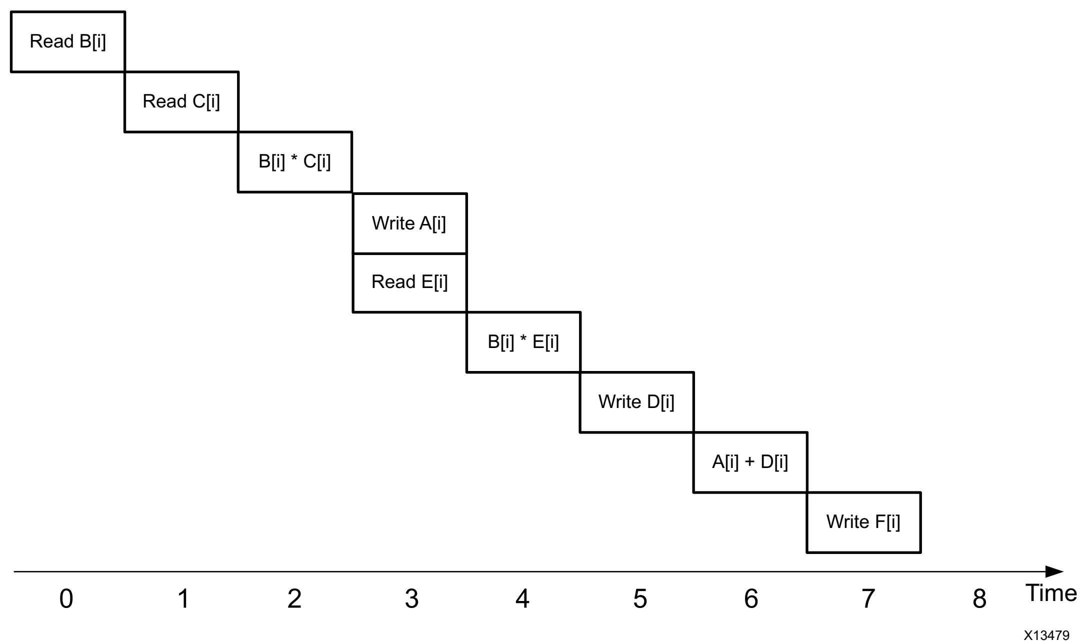

図 4-2: プロセッサでのサンプル コードの実行

図 4-3 に、Vivado HSL のデフォルト設定を使用して図 4-1 のコードを FPGA にコンパイルした結果を示します。この実行プロファイルは、乗算と加算が順次に実行されるという点でプロセッサの実行プロファイルに似ています。ユーザー アプリケーションのインプリメンテーションに必要な構築ブロック数を最小限に抑えるため、この動作がデフォルトになっています。FPGA には固定プロセッシング アーキテクチャはありませんが、各デバイスで構築ブロックの最大数は決まっています。そのため、FPGA リソースを、アプリケーションのパフォーマンス、デバイスごとのアプリケーション数に対して評価できます。

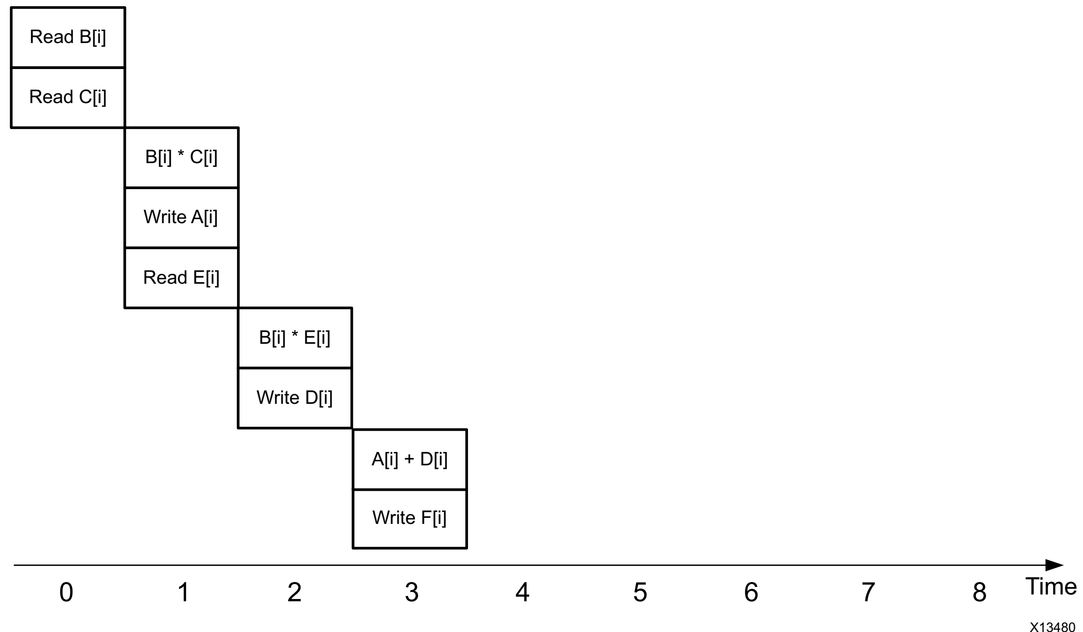

図 4-3: FPGA での HLS コードのデフォルト実行

デフォルト動作であっても、このインプリメンテーションは、アルゴリズム用にカスタム メモリ アーキテクチャが作成されているので、プロセッサ実行のパフォーマンスを上回ります。プロセッサでは、配列 A、B、C、D、E、および F は 1 つのメモリ空間に格納され、一度に 1 つしかアクセスできません。Vivado HLS では、これらのメモリが検出されて、配列ごとに独立したメモリ バンクが作成されるので、配列 B と C の読み出しはオーバーラップします。

クロック サイクル 1 で配列 E の読み出しがスケジュールされていますが、これは Vivado HLS の自動リソース最適化の 1 つです。メモリ操作に対しては、Vivado HLS で、データを含むバンクと、計算中に値がどこで使用されるかが解析されます。配列 E はクロック サイクル 0 で読み出される可能性もありますが、Vivado HLS では、回路の一時データ ストレージの量を減らすため、メモリ操作はデータが使用される場所のなるべく近くに自動的に配置されます。E の値を使用する乗算はクロック サイクル 2 まで実行されないので、この読み出しアクセスをクロック サイクル 1 より前にスケジュールしても利点はありません。

Vivado HLS では、変数のサイズを変更可能なデータ型がいくつか提供されており、ユーザーが生成された回路のサイズを制御することもできます。ほかのコンパイラと同様、Vivado HLS でも整数型、単精度型、倍精度型を使用できます。これらを利用するとソフトウェアを FPGA にすばやく移行できますが、プロセッサで使用可能な 32 ビットおよび 64 ビットのデータパスであるために発生するアルゴリズムの効率の悪さが隠される可能性があります。

たとえば、図 4-1 に示すコードの配列 B、C、E に 20 ビット値のみが必要であるとします。元のプロセッサのコードでは、精度が失われないようにするため、配列 A、D、F で 64 ビットの値を格納できるビット サイズにする必要があります。このコードをそのまま Vivado HLS でコンパイルすることもできますが、非効率の 64 ビット データパスがインプリメントされ、アルゴリズムで必要なリソースよりも多くのリソースが使用されます。

図 4-4 に、Vivado HLS の任意精度型を使用して図 4-1 のコードを変更した例を示します。これらのデータ型を使用すると、アルゴリズムを正しく動作させるのに最低限必要な精度をソフトウェア レベルですばやく試行し、検証できます。任意精度型を使用すると、計算のインプリメンテーションに必要なリソース数が削減されるだけでなく、演算の完了に必要なロジック段数を減らすこともでき、デザインのレイテンシが削減されます。

```
ap_int<40> A[10], D[10];
ap_int<41> F[10];
ap_int<20> B[10], C[10], E[10];
...
A[i] = B[i] * C[i];
D[i] = B[i] * E[i];
F[i] = A[i] + D[i];
```

図 4-4: HLS の任意精度型を使用したコード例

第 3 章「ハードウェア デザインの基本概念」で説明したように、パイプライン処理 (計算をレジスタで区切られた領域に分割) は、ターゲット クロック周波数を達成する上で不可欠な FPGA の設計テクニックです。この最適化では、演算のサイズに基づいて、Vivado HLS で自動的にインプリメントされます。大型の演算が複数の計算段に分けられるので、それに応じて回路のレイテンシが増加します。

条件文

条件文とは通常、if、if-else、case 文としてインプリメントされるプログラム制御フロー文を指します。条件文は、ほとんどのアルゴリズムの不可欠な要素で、Vivado HLS を含むすべてのコンパイラで完全にサポートされています。コンパイラ間での唯一の違いは、これらの条件文がどのようにインプリメントされるかです。

プロセッサ コンパイラの場合、条件文は分岐操作に変換され、それがコンテキスト スイッチになる場合とならない場合があります。分岐により依存関係が発生し、メモリから次にフェッチされる命令に影響するので、図 3-2 に示す命令実行の最大限にパックすることができません。このような不確実性があると、プロセッサの実行パイプラインでボトルネックが発生し、プログラムのパフォーマンスに直接影響します。

FPGA では、条件文を使用しても、プロセッサのようなパフォーマンスへの影響はありません。Vivado HLS では、条件文の各分岐で記述されている回路がすべて作成されます。このため、条件文のランタイム実行では、コンテキスト スイッチではなく、2 つの結果値のいずれかが選択されます。

ループ

ループは、反復計算の記述によく使用されるプログラミング構文です。Vivado HLS などのコンパイラではループがサポートされていないと誤解されることがあります。初期の FPGA 用コンパイラではそうだったかもしれませんが、Vivado HLS ではループは完全にサポートされており、標準プロセッサ コンパイラの機能以上の変換も可能です。図 4-5 に、単純なループ例を示します。

```
for(i=0; i < 10; i++)
{
     A = A + (B[i] * C[i]);
}
```

図 4-5: ループ コード

ここでは便宜上、インプリメンテーション プラットフォームに関係なく、このループは反復ごとに 4 クロック サイクルかかると想定します。プロセッサでは、図 4-6 に示すように、コンパイラはループ反復を順次スケジュールせざるおえず、合計実行時間は 40 サイクルになります。

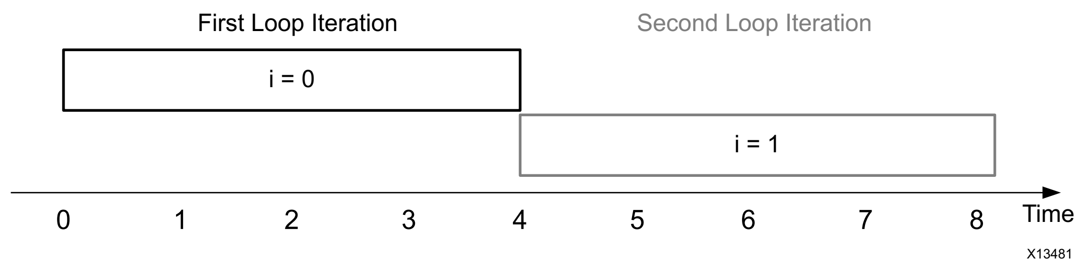

図 4-6: プロセッサでのループ反復のスケジュール

Vivado HLS にはこのような制約はありません。Vivado HLS ではアルゴリズム用のハードウェアが作成されるので、反復実行をパイプライン処理して、ループの実行プロファイルを変更できます。ループの反復のパイプライン処理は、並列処理の概念をループ反復内から反復間に拡張したものです。

反復レイテンシを低減するため、Vivado HLS で最初に適用される自動最適化は、ループ反復本体に対する演算子の並列処理です。その次に適用される最適化は、ループ反復のパイプライン処理です。この最適化は、FPGA インプリメンテーションのリソース使用率および入力データ レートに影響するので、ユーザーの介入が必要になります。

Vivado HLS のデフォルト動作では、図 4-6 に示すように、プロセッサと同じスケジュールでループが実行されます。つまり、図 4-5 のコードは、レイテンシが 40 サイクル、入力データ レートは 4 サイクルごとに 1 回です。この例では、入力データ レートは B と C の値を入力からサンプリングするスピードで定義されます。

Vivado HLS では、ループの反復を並列処理またはパイプライン処理することにより、計算レイテンシを低減して入力データ レートを高くすることが可能です。反復のパイプライン処理のレベルは、ループの開始間隔 (II) を設定することによりユーザーが制御します。ループの II は、ある反復が開始してから、次の反復が開始するまでの間のクロック サイクル数です。図 4-7 に、II の値を 1 に設定した後のループのスケジュールを示します。

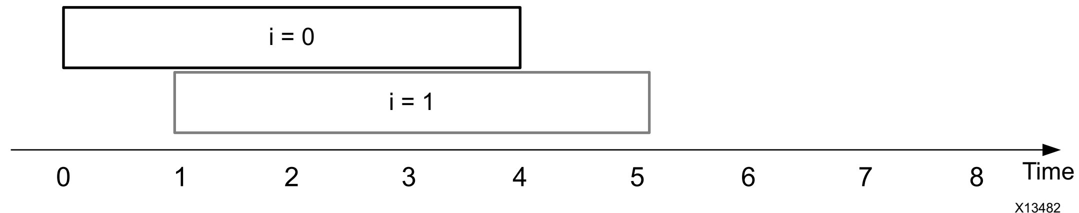

図 4-7: II = 1 に設定したループ反復のスケジュール

Vivado HLS は、この結果を達成するため、ループ反復 0 と 1 の間のデータの依存関係およびリソース競合を解析し、問題を次のように自動的に解決します。

- データの依存関係を解決するため、ループ本体の演算の 1 つを変更するか、ユーザーにアルゴリズムの変更を推奨します。
- リソース競合を解決するため、リソースのコピーをより多くインスタンシエートするか、ユーザーにアルゴリズムの変更を推奨します。

表 4-1 に、ループのパイプライン処理の効果を示します。

表 4-1: 異なるコンパイラでのループ実行プロファイル

コンパイラ	ループ実行レイテンシ	入力データ レート
プロセッサ	40	4 クロック サイクルごと
デフォルトの HLS	40	4 クロック サイクルごと
HLS、II = 1	14	1 クロック サイクルごと

関数

関数は、演算子、ループ、およびほかの関数を含むことができるプログラミング階層です。関数の処理は、Vivado HLS でもプロセッサ コンパイラでも、ループの処理に似ています。

Vivado HLS では、ループと関数の主な違いは用語です。Vivado HLS では、ループも関数も並列実行できます。ループでは、演算子とループの反復の間に明らかな階層があるので、この変換は一般的にパイプライン処理と呼ばれます。関数では、ループ本体外の演算とループ内の演算が同じ階層にあるので、パイプライン処理という用語を使用すると混乱を招く可能性があります。Vivado HLS を使用する場合にこの混乱を避けるため、関数呼び出し実行の並列処理は「データフロー最適化」と呼ばれます。

データフロー最適化により、Vivado HLS で同じプログラム階層にあるすべての関数に独立したハードウェア モジュールが作成されます。これらのハードウェア モジュールは同時実行可能で、データ転送中に自己同期化します。

ダイナミック メモリ割り当て

ダイナミック メモリ割り当ては、C および C++ で使用可能なメモリ管理方法の 1 つです。この方法では、プログラムの実行中に必要なだけメモリを割り当てることができます。割り当てられたメモリのサイズは、プログラムの実行ごとに変動する可能性があり、第 3 章「ハードウェア デザインの基本概念」で説明したように中央のメモリから割り当てられます。表 4-2 に、通常ダイナミック メモリ割り当てに使用される関数呼び出しを 示します。

表 4-2: ダイナミック メモリ管理で使用される関数

C	C++
malloc()	new()
calloc()	delete()
free()	

第 3 章「ハードウェア デザインの基本概念」で説明したように、FPGA には Vivado HLS コンパイラでユーザー アプリケーションをフィットさせる必要がある固定のメモリ アーキテクチャはありません。その代わり、アルゴリズム独自の要件に基づいて、メモリ アーキテクチャが合成されます。このため、FPGA にインプリメンテーションするために Vivado HLS コンパイラで処理するすべてのコードには、コンパイル時に解析可能なメモリ割り当てのみを使用する必要があります。

Vivado HLS で処理するコードが合成可能であることを確認するため、デザイン解析の前にコンパイラによりコード準拠テストが実行されます。この準拠テストでは、Vivado HLS に適していないコーディング スタイルがチェックされます。コードを変更してダイナミック メモリ割り当てをすべて削除するのは、ユーザーの責任です。

図 4-8 のコードでは、32 ビットの値を 10 個格納するメモリ領域を割り当てています。

```
int *A = malloc(10*sizeof(int));
```

図 4-8: ダイナミック メモリ割り当て

このコード例は定数メモリ割り当てですが、Vivado HLS のコード準拠段では `malloc` 文の内容は解析されません。Vivado HLS では、図 4-8 のように割り当てが定数であっても、表 4-2 にあるキーワードが含まれているコードは合成できません。このコードを Vivado HLS で合成できるように変更するには、次の 2 つの方法があります。次に示すコード例で、これら 2 つの方法とそれぞれの FPGA インプリメンテーションへの影響を説明します。

図 4-9 のコードは、C/C++ プログラムによる自動メモリ割り当てを示します。このタイプのメモリは、Vivado HLS で C/C++ で規定されている動作に厳密に準拠してインプリメントされます。つまり、配列 `A` の格納用に作成されたメモリには、この配列を含む関数呼び出しが実行されている間のみ有効なデータ値が格納されます。そのため、毎回関数呼び出しを使用する前に `A` に有効なデータを配置する必要があります。

```
int A[10];
```

図 4-9: Vivado HLS に準拠した自動メモリ割り当て

図 4-10 のコードは、C/C++ プログラムによるスタティック メモリ割り当てを示します。このタイプのメモリ割り当てでは、配列 A の内容は、プログラムが完全にシャットダウンされるまで、すべての関数呼び出しで使用できます。Vivado HLS を使用する場合、配列 A 用にインプリメントされるメモリには、回路に電源が投入されている間は有効なデータが含まれます。

```
static int A[10];
```

図 4-10: Vivado HLS に準拠したスタティック メモリ割り当て

自動メモリ割り当てでも、スタティック メモリ割り当てでも、プロセッサ上で実行するアルゴリズムの全体的なソフトウェア メモリ フットプリントが増える可能性があります。FPGA インプリメンテーション用に C/C++ でアルゴリズムを指定する際、ユーザー アプリケーションの全体的な目標を考えることが最も重要です。つまり、FPGA にコンパイルするときは、ベストなソフトウェア アルゴリズム インプリメンテーションを作成することが主な目標ではないということです。Vivado HLS のようなツールを使用するときは、ツールができるだけ最適なハードウェア アーキテクチャを推論できるようにアルゴリズムを記述するが目標になります。そうすることで、最終的に最適なインプリメンテーションを達成できます。

ポインター

ポインターはメモリ内の位置を指定するアドレスです。C/C++ プログラムでの一般的なポインターの使用方法には、関数パラメーター、配列処理、ポインター トゥ ポインター、型キャストなどがあります。ポインターには柔軟性があるので、C/C++ コードでは便利でよく使用されます。Vivado HLS コンパイラでは、コンパイル時に完全に解析可能なポインターの使用がサポートされています。解析可能なポインターは、ランタイム情報を使用しなくても、ペンと紙での計算でも完全に表現および計算できます。

図 4-8 のコードには、ダイナミックに割り当てられたメモリの 1 領域を参照するポインターが使用されています。前にも説明したように、この使用方法ではポインターのデスティネーション アドレスがプログラムを実行しないとわからないので、Vivado HLS ではサポートされていません。これは、メモリの管理にポインターを使用することが Vivado HLS コンパイラでサポートされていないということではありません。図 4-11 に、メモリへのアクセスにポインターを使用するのに有効なコーディング スタイルを示します。

```
int A[10];
int *pA;

pA = A;
```

図 4-11: ポインターを使用した配列アクセスの管理

ポインター pA の使用はすべて解析可能であり、配列 A にマップし戻すことができるので、このコードは有効です。配列 A は自動メモリ割り当てにより作成されているので、A のプロパティを Vivado HLS で完全に判断できます。

メモリおよびポインターでは、外部メモリへのアクセスもサポートされます。Vivado HLS を使用する場合、関数パラメーター上のポインター アクセスは、変数または外部メモリです。Vivado HLS では、外部メモリは、コンパイラで生成された RTL の範囲外のメモリとして定義されます。つまり、メモリは FPGA の別の関数にあるか、または DDR などのオフチップ メモリにあるということです。

図 4-12 のコードに示すように、関数 foo は、data_in というパラメーターを使用した Vivado HLS の最上位モジュールです。data_in に複数のポインター アクセスがあるので、Vivado HLS では、この関数パラメーターが、ハードウェア レベルのバス プロトコルを介してアクセスする必要のある外部メモリ モジュールであると推論されます。AXI (Advanced eXtensible Interface) プロトコルなどのバス プロトコルは、複数の関数がどのよう接続され、通信するのかを指定します。

```
void foo(int *data_in,...)
{
   int item1, item2, item3;

  item1 = *data_in;
  item2 = *(data_in + 1);
  item3 = *(data_in + 2);
  ...
}
```

図 4-12: 外部メモリへのポインター

第 5 章

計算中心のアルゴリズム

概要

アルゴリズム解析についての資料は多数ありますが、計算中心のアルゴリズムと制御中心のアルゴリズムとの違いは、ほとんどインプリメンテーション プラットフォームに依存しています。この章では、Vivado® HLS コンパイラおよび FPGA における計算中心のアルゴリズムを定義します。また、HLS で生成されたインプリメンテーションのパフォーマンスを最大限にするための例およびベスト プラクティスも示します。

計算中心のアルゴリズムは、タスクごとに 1 回設定され、タスクの実行中はその動作を変更できません。ハードウェアにおけるタスクは、C/C++ プログラムにおける関数呼び出しと同じです。タスクのサイズは HLS ユーザーが決めることができます。

推奨: 通常は、アルゴリズムでの自然な分割に基づいてタスクのサイズを決定することを推奨します。

図 5-1 に、Sobel エッジ検出操作のコードを示します。この計算中心のアルゴリズム例は、異なるサイズのタスクに分割できます。このアルゴリズムは、各ピクセルの勾配を x および y 方向で計算して画像のある領域のエッジを求める 2 次元フィルター操作を実行します。このコードは、このまま HLS で FPGA インプリメンテーションにコンパイルできます。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
             for(colOffset= -1; colOffset <= 1; colOffset++){
                x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
             }
          }
     edge_weight = ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

図 5-1: Sobel エッジ検出アルゴリズム: タスク定義 1

このアルゴリズムを正しく最適化するには、まずタスクのサイズを決める必要があります。タスクのサイズが決まると、生成されたハードウェア モジュールをコンフィギュレーションする頻度と、そのモジュールが新しいデータを受信する頻度が決まります。図 5-2 と図 5-3 に、図 5-1 のコードで可能な 2 種類のタスク定義を示します。図 5-1 のコードを 1 つのタスクとして定義する選択肢もあります。

タスク定義 2 (図 5-2) では、勾配計算のみのハードウェア モジュールが作成されます。勾配は 3x3 ピクセルごとに計算され、ラインや画像フレームのコンセプトはサポートされません。タスク定義 2 には、実行される作業量と、アルゴリズムの自然な作業分けが一致しないという問題があります。Sobel エッジ検出は画像全体で実行されるので、このタスク サイズを選ぶ場合は、HLS で構築されるタスク プロセッサで処理できるように、画像を 3x3 ピクセルに分割する方法を決める必要があります。また、アルゴリズムのこの機能を完了させるには、プロセッサまたは追加のハードウェア モジュールが必要になります。

```
for(rowOffset = -1; rowOffset <= 1; rowOffset++){
      for(colOffset= -1; colOffset <= 1; colOffset++){
                   x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                   y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
      }
}
```

図 5-2: Sobel エッジ検出アルゴリズム: タスク定義 2

タスク定義 3 (図 5-3) では、1 タスクで 1 ピクセル ラインが処理されます。このタスク定義では、アルゴリズムの機能全体をインプリメントするのに追加するモジュールが少なくて済むので、タスク定義 1 より優れています。また、この方法では、制御プロセッサとのやりとりがラインごとに 1 回で済みます。ただし、1 タスクで 1 回に 1 ライン処理するようにすると、最終結果の計算に複数のラインが必要になる点が問題です。タスク定義 3 を選択すると、HLS 生成のハードウェア モジュールに画像ラインを順次読み込むために複雑な制御メカニズムが必要になる可能性があります。

```
for(j = 0; j < width; j++){
        x_dir = 0;
        y_dir = 0;
        if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
        for(rowOffset = -1; rowOffset <= 1; rowOffset++){
            for(colOffset= -1; colOffset <= 1; colOffset++){
               x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
               y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
            }
         }
      edge_weight =ABS(x_dir) + ABS(y_dir);
      output_image[i][j] = edge_weight;
}
```

図 5-3: Sobel エッジ検出アルゴリズム: タスクの選択肢 3

このアルゴリズムにはタスク定義 1 (図 5-1) を選択するのがベストです。その理由は、このタスク定義が 図 5-1 のコードで記述されている関数呼び出しごとのフル画像と一致するからです。この定義では、生成された FPGA インプリメンテーションのコンフィギュレーションは 1 画像フレームが処理される間固定されるので、計算中心のタスクになります。処理される画像のサイズはフレームごとに変更できますが、タスクが開始した後に変更することはできません。

タスクに適切なサイズが決まったら、HLS コンパイラのオプションを使用してアルゴリズムのインプリメンテーションを最適化する必要があります。図 5-1 のコードでは、FPGA インプリメンテーションのターゲットとなる画像サイズは 1080 ピクセルで、1 秒ごとに 60 フレーム処理されます。これは、150 MHz のクロック周波数で 1920 x 1080 ピクセル、クロック サイクルごとに 1 ピクセルの入力データ レートで処理するハードウェア モジュールになります。

データ レートの最適化

HLS コンパイラでは、コード最適化はベースラインのコンパイルから始まります。ベースラインのコンパイルは、インプリメンテーションでボトルネックとなる箇所を見極めることと、異なる最適化の効果を計測するための基準点を設定することの 2 つを目的にしています。ベースラインのコンパイルでは、できる限り FPGA のリソースを使用せず、最も低い入力データ レートでアルゴリズムのインプリメンテーションを構築します。この章の例では、ベースラインのコンパイルで、40 クロック サイクルごとに 1 ピクセルの入力データ レートが得られます。

HLS コンパイラを使用する場合、生成されたインプリメンテーションの入力データ レートおよび並列処理能力を高めるには、パイプライン処理を最適化します。第 2 章「FPGA とは」および第 3 章「ハードウェア デザインの基本概念」で説明されているように、パイプライン処理では 1 つの大きな計算を同時実行できる小さな段に分割します。パイプライン処理がループに適用されると、ループの開始間隔 (II) が設定されます。

ループ II は、i+1 の反復を開始するのに必要なクロック サイクル数を設定することにより、ループの入力データ レートを制御します。アルゴリズム コードのどこでパイプラインを最適化するかは、設計者が選択できます。図 5-4 に、ウィンドウの計算にパイプライン プラグマを適用した例を示します。

注記: HLS コンパイラで使用可能なプラグマの詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) [参照 1] を参照してください。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
              for(colOffset= -1; colOffset <= 1; colOffset++){
#pragma HLS PIPELINE
                 x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                 y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
              }
          }
     edge_weight =ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

図 5-4: ウィンドウの計算にパイプライン プラグマを適用

図 5-4 に、アルゴリズムのソースにコンパイラ プラグマとして直接適用されているパイプライン最適化の例を示します。コードのこのレベルにパイプライン プラグマを適用すると、クロック サイクルごとに 3x3 フィルター ウィンドウの 1 フィールドが計算されます。つまり、3x3 のウィンドウで乗算を実行するには、9 クロック サイクル必要で、さらに、結果のピクセルを生成するのに 1 クロック サイクル必要になります。アプリケーション レベルでは、入力サンプル レートが 10 クロック サイクルごとに 1 ピクセルになり、アプリケーションの要件を満たすには十分ではありません。

図 5-5 では、パイプライン プラグマを画像の列全体にまたがる j ループに適用しています。パイプライン プラグマをこのループに適用することにより、HLS インプリメンテーションでクロック サイクルごとに 1 ピクセルの入力データ レートを達成できます。この入力データ レートを達成するため、コンパイラでまずウィンドウ計算ループを完全に展開し、勾配の乗算がすべて並列に計算されるようにします。この展開により、追加ハードウェアがインスタンシエートされ、必要なメモリ帯域幅が高くなり、入力画像に対し、クロック サイクルごとにメモリ操作が 9 回必要になります。

```
for(i = 0; i < height; i++){
     for(j = 0; j < width; j++){
#pragma HLS PIPELINE
          x_dir = 0;
          y_dir = 0;
          if((i > 0) && (i < (height-1)) && (j > 0) && (j<(width-1))){
          for(rowOffset = -1; rowOffset <= 1; rowOffset++){
              for(colOffset= -1; colOffset <= 1; colOffset++){
                 x_dir = x_dir + input_image[i+rowOffset][j+colOffset] * Gx[1+rowOffset][1+colOffset];
                 y_dir = y_dir + input_image[i+rowOffset][j+colOffset] * Gy[1+rowOffset][1+colOffset];
              }
          }
     edge_weight =ABS(x_dir) + ABS(y_dir);
     output_image[i][j] = edge_weight;
}
```

図 5-5: パイプライン プラグマを j ループに適用

HLS コンパイラでアルゴリズムで記述されているよりも高いメモリ帯域幅が必要だと判断される可能性はありますが、アルゴリズムの機能に影響するような変更は自動的には加えられません。この例では、パイプライン最適化で 9 つの同時メモリ アクセスが必要ですが、HLS で生成されたモジュールの境界外のメモリではそれを満たすことはできません。

外部メモリのポート数に関係なく、HLS で生成されたモジュールでは、クロックサイクルごとに 1 トランザクションを操作できるシングル ポート 1 つにしか接続できません。このため、アルゴリズムを変更して、メモリ帯域幅要件をモジュールの入力ポートから外し、HLS コンパイラにより生成されたメモリでその要件が満たされるようにする必要があります。この内部メモリは、プロセッサのキャッシュに似ています。Sobel エッジ検出のような画像処理アルゴリズムでは、このローカル メモリは「ライン バッファー」と呼ばれます。

ライン バッファーはマルチバンク内部メモリで、生成されたインプリメンテーションでクロック サイクルごとに 3 本のラインからピクセルに同時アクセスできるようにします。ライン バッファーをインプリメントするアルゴリズムでは、計算の開始前に、計算要件を満たすのに十分なデータをライン バッファーにためる時間を割り当てる必要があります。つまり、計算する結果ごとにメモリに 9 つのアクセスが必要であるという要件を満たすには、ライン バッファー内のデータの移動だけでなく、アルゴリズムの変更による追加帯域幅も考慮する必要があります。

図 5-6 に、ライン バッファー内の画像ピクセルの動きを示します。

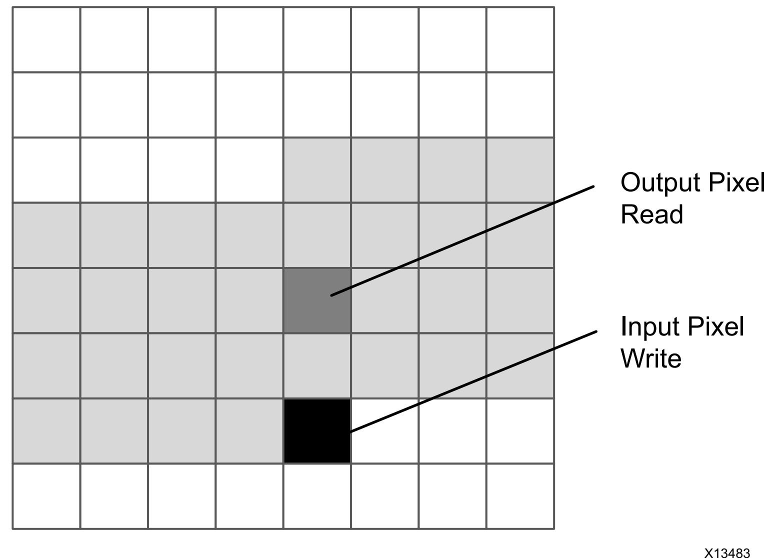

図 5-6: ライン バッファー内のデータの動き

薄いグレーの四角は、このメモリ構造に現在格納されているピクセルを示します。このブロックは、正しい機能を得るために必要な最小限の数のピクセルのみを格納するのが目的で、画像全体の格納はしません。先ほど述べたように、このメモリ構造が追加されることにより、入力ピクセルのサンプリングと出力ピクセルの計算との間に遅延が発生します。図 5-5 のコードに示す 3x3 ウィンドウ演算の場合、最初の出力ピクセルを計算する前に、ライン バッファーに 2 本の画像ラインおよび 3 本目のラインの最初の 3 ピクセルを格納しておく必要があります。濃いグレーと黒の四角はこのレイテンシを示します。黒の四角は、ソースの画像の次の入力ピクセルが書き込まれる位置を示します。濃いグレーの四角は、出力画像で現在計算されているピクセルの位置を示します。

HLS では、FPGA ファブリックの BRAM リソースを使用して、ライン バッファーがインプリメントされます。これらのデュアル ポート メモリ エレメントはバンクに配置され、1 バンクが 1 ラインに対応します。このため、アルゴリズムの計算に使用可能なメモリ帯域幅は、クロック サイクルごとに 1 ピクセルからクロック サイクルごとに 3 ピクセルと 3 倍になります。それでも、クロック サイクルごとに 9 ピクセルの要件を満たすには不十分です。

クロック サイクルごとに 9 ピクセルの要件を満たすには、ライン バッファーに加え、アルゴリズムにメモリ ウィンドウを 1 つ追加する必要があります。メモリ ウィンドウとは、FPGA ファブリックのフリップフロップを使用してインプリメントされるストレージ エレメントのことです。このメモリの各レジスタには、ほかのレジスタから独立して、ほかのレジスタと同時にアクセスすることが可能です。論理的には、フリップフロップで構成されたメモリは、C/C++ で記述されたアルゴリズムに最も合った形にすることができます。

図 5-7 に、Sobel エッジ検出アルゴリズムのメモリ ウィンドウを示します。

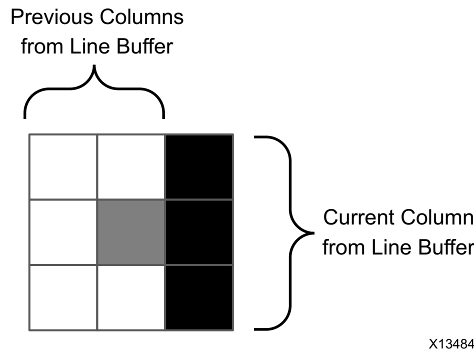

図 5-7: メモリ ウィンドウ

中央にあるグレーのピクセルは、勾配が計算されているピクセルを示します。黒の列は、ライン バッファーから供給される 3 ピクセルを示します。このウィンドウの内容はクロック サイクルごとに左シフトし、ライン バッファーから次に供給される 3 ピクセル分の空きが作られます。メモリ ウィンドウのデータ再利用および分散インプリメンテーションにより、アルゴリズムで必要な 9 つのメモリ操作が可能になります。このメモリでは、デザインに追加レイテンシは発生しません。ウィンドウのデータ移動は、ライン バッファーの移動と同時に実行されます。

図 5-8 に、層構造のメモリ アーキテクチャを介した入力から計算が実行されるまでの全体的なデータの流れを示します。

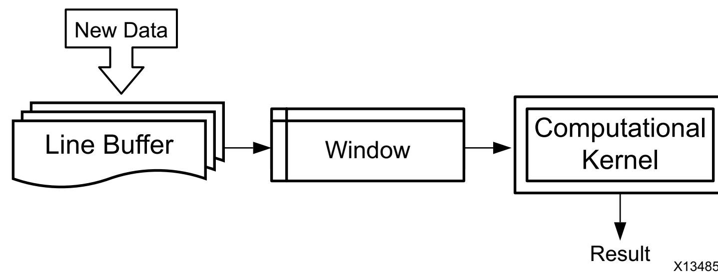

図 5-8: 入力から計算までのデータの流れ

図 5-9 に、層構造のメモリ アーキテクチャをインプリメントするために必要なアルゴリズム コードの変更を示します。この層構造のアーキテクチャにより、HLS で生成されたインプリメンテーションでクロック サイクルごとに 1 ピクセルの入力データ レートを達成できます。このコード例では、アルゴリズムの計算カーネルは `sobel_operator` 関数にあります。このコードでの主な変更は、rows および cols ループがそれぞれ 1 反復分拡張されていることです。この拡張が、ライン バッファーにより追加されるタスク実行時間の理由です。さらに、ライン バッファーへの書き込み操作は、元の画像境界線に基づく `if` 条件文で保護されています。アルゴリズムの出力書き込み操作は、出力画像の位置に基づいており、元の画像からは 1 行、1 列分オフセットされています。

図 5-9 に示すように、計算中心のアプリケーションには、for ループ、if-else 文などの制御文を埋め込むことができます。この種のアルゴリズムの主な特徴は、その機能および動作がタスク実行中は固定されていることです。HLS で生成されたモジュールでは、コンフィギュレーションに基づいてデータのバッチが処理されます。コンフィギュレーションは各タスクごとに変更できますが、タスク実行中は変更できません。

ヒント: ライン バッファー用のライブラリは、HLS コンパイラで使用可能なビデオ ライブラリの一部として含まれています。詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) [参照 1] を参照してください。

```
for(row = 0; row < rows+1; row++){
     for(col = 0; col < cols+1; col++){
          if(col < cols){
                     buff_A.shift_up(col);
                     temp = buff_A.getval(0,col);
          }
          if(col < cols & row < rows) {
                buff_A.insert_bottom(rgb2y(input_pixel[row][col]),col);
          }
          buff_C.shift_right();
          if(col < cols){
                 buff_C.insert(buff_A.getval(2,col),0,2);
                 buff_C.insert(temp,1,2);
                 buff_C.insert(rgb2y(tempx),2,2);
          }
          if( row <= 1 || col <= 1 || row > (rows-1) || col > (cols-1)){
                 edge.R = edge.G = edge.B = 0;
          }
          else{
                 edge = sobel_operator(&buff_C);
          }
          if(row > 0 && col > 0){
                  AXI_PIXEL output_pixel;
                  output_pixel.data = (edge.B, edge.G);
                  output_pixel.data = (output_pixel.data, edge.R);
                  out_pix[row-1][col-1] = output_pixel;
         }
       }
     }
}
```

図 5-9: ライン バッファーを使用した Sobel エッジ検出コード

XILINX

第 6 章

制御中心のアルゴリズム

概要

制御中心のアルゴリズムは、システムレベルの要因に基づいてタスク実行中に変更可能なアルゴリズムです。計算中心のアルゴリズムでは、1 つのタスクを実行している間、同じ演算がすべての入力データ値に適用されますが、制御中心のアルゴリズムでは、入力ポートの現在のステータスに基づいて演算が決定されます。この章では、Vivado® HLS コンパイラでこのタイプのアプリケーションを最適化するベスト プラクティスを紹介します。

C++ で記述される制御

ベスト プラクティスを紹介する前に、まず C および C++ 言語で制御がどのように記述されるのかを確認します。

ループ

ループは、反復計算の記述に使用されるプログラミングの基本的な構文です。ほかのコンパイラと同じように、Vivado HLS でもループは、for、while、do-while で表現できます。この構文は、Vivado HLS でコンパイルされるすべてのタイプのアプリケーションでサポートされています。第 5 章「計算中心のアルゴリズム」で示した Sobel エッジ検出アルゴリズムの例のように、ループは計算負荷の高いアルゴリズムを C/C++ で表現するのに不可欠な構文です。

図 6-1 に、for ループを使用した例と、Vivado HLS によるコンパイルの効果を示します。この図は、Vivado HLS のコンパイルで、計算と制御ロジックの両方が 1 つの FPGA インプリメンテーションの一部として生成されることを示しています。FPGA ファブリック用の前世代のコンパイラとは異なり、Vivado HLS コンパイラでは、制御の構文と計算の構文は区別されません。この図のコードでは、Vivado HLS により、ループでの算術演算用にパイプライン処理されたデータパスが生成されます。このタイプのインプリメンテーションでは、ループの反復内と反復間の両方で計算を並列処理することにより、実行レイテンシが低減されます。また、Vivado HLS インプリメンテーションには、このロジックだけでなく、ループ コントローラー ロジックも含まれます。値 y の計算にハードウェアが何回実行されるのかは、このループ コントローラー ロジックによって決定されます。

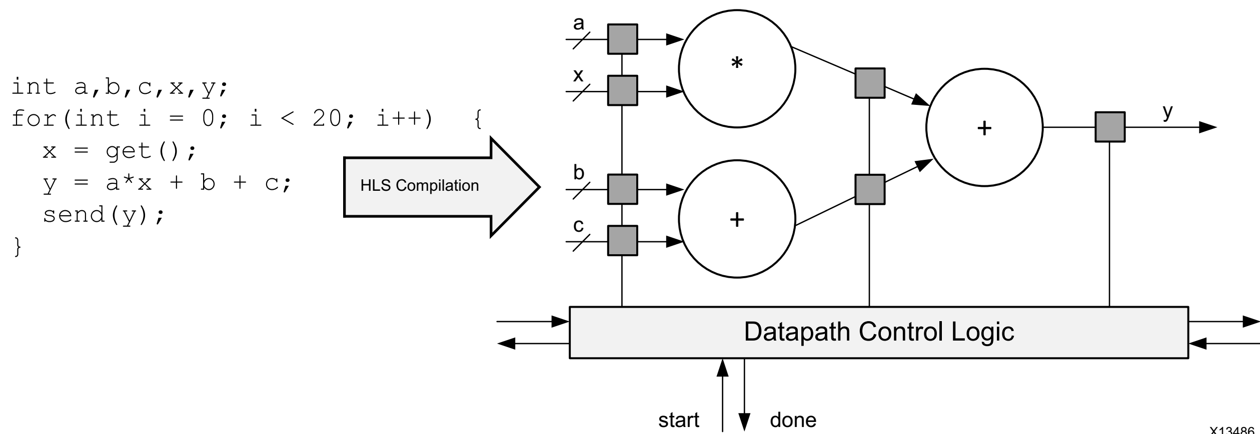

図 6-1: ループの例

条件文

C および C++ では、条件文は通常 if-else 文として記述されますが、ハードウェア インプリメンテーションでは、トリガー値に基づいて、2 つの結果値または 2 つの実行パスが選択されます。この便利な構文を活用することにより、変数または関数レベルでアルゴリズムを制御できます。いずれのユース ケースも Vivado HLS コンパイラで完全にサポートされています。

図 6-2 に、if 文を使用してアルゴリズムにある 2 つの関数のいずれかを選択する if-else 文の例を示します。Vivado HLS コンパイラで生成されたインプリメンテーションにより、`function_a` と `function_b` の両方に FPGA リソースが割り当てられます。これらのハードウェア回路は並列実行し、同じクロック サイクルで結果値が生成されるようにバランスが取られています。2 つの計算結果値の選択には、元のソース コードの条件トリガーが使用されます。

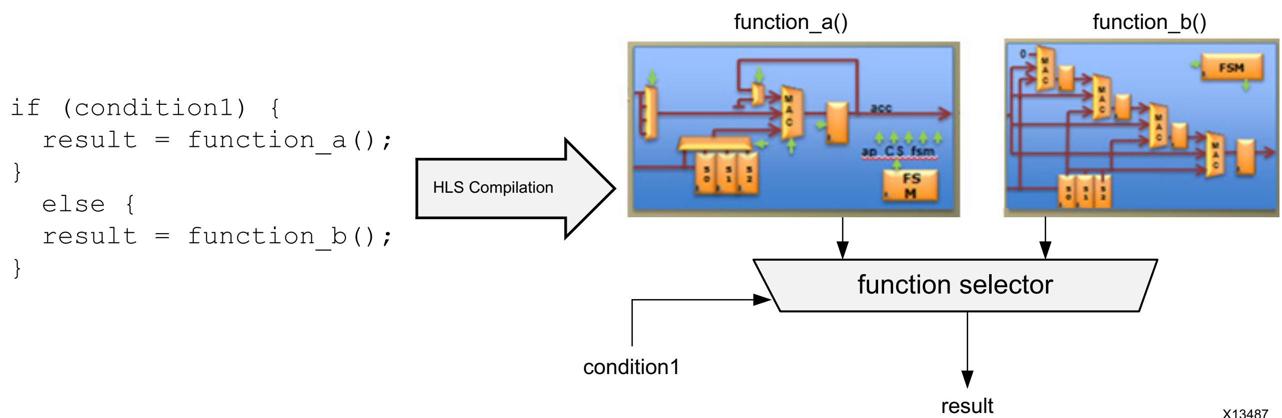

図 6-2: if-else 文の例

case 文

case 文は、入力変数の値に基づき、プログラム内の演算またはイベントが発生する順序を定義します。この構文は計算中心のアルゴリズムでも使用できますが、システム レベルでの変更が直接モジュールの実行に影響を与える制御中心のアルゴリズムでより広く使用されています。また、ほとんどの使用モデルで、case 文により 1 つのプログラム制御領域から別の領域への遷移が明示的に定義されます。

図 6-3 に、case 文の例と、Vivado HLS でのコンパイル結果を示します。case 文は、コンパイラによりハードウェアの有限ステート マシン (FSM) に変換されます。FSM の配列は、ステート間の遷移を表わし、このコード例の case 遷移に対応します。FSM の各ステートには、プログラム制御領域内の計算ロジックも含まれます。

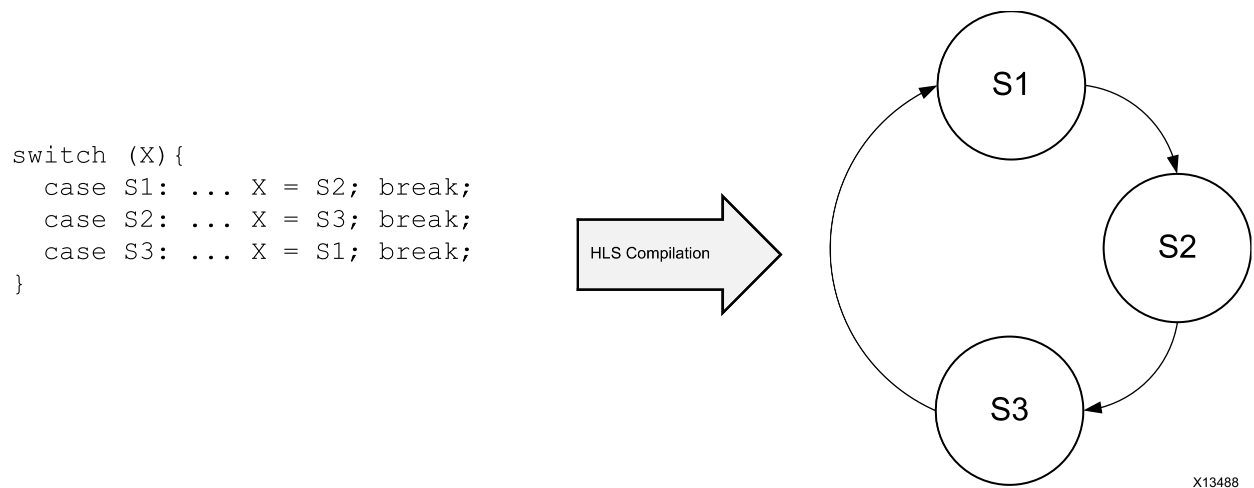

図 6-3: case 文の例

制御システムの分類

制御中心のアプリケーションをコード記述したら、次にアプリケーションを実行するプラットフォームを決める必要があります。かつては、プラットフォームにプロセッサが選択されることが多く、Zynq®-7000 SoC で示されるように、現在でもプロセッサを選択するのがベストであるユース ケースは数多く存在しますが、HLS コンパイラの登場により、FPGA ファブリックに制御アルゴリズムをインプリメントするのにボトルネックとなるステート マシンの最適化や複雑さといった問題がなくなりました。設計者には、制御アルゴリズムをプロセッサで実行するか、または FPGA ファブリックで Vivado HLS で生成されたカスタム コントローラーとして実行するかの選択肢があります。どちらを選択するのかは、アルゴリズムの応答時間要件と FPGA ファブリックのリソース使用率によります。

表 6-1 に、外部イベントへの応答時間による制御アルゴリズムの分類を示します。

表 6-1: 制御システムの分類

制御タイプ	実行時間 (クロック サイクル)	推奨されるインプリメンテーション
非常に低速	≥ 1,000,000	X86 型プロセッサ、DSP、または Zynq-7000 SoC
低速	100,000 から 1,000,000 まで	X86 型プロセッサ、DSP、または Zynq-7000 SoC (HLS で生成されたアクセラレータを使用)
中速	1,000 から 100,000 まで	Zynq-7000 SoC (HLS で生成されたアクセラレータを使用)
高速	≤ 1,000	HLS 生成されたカスタム コントローラー

応答時間が非常に低速なデザインでは、プロセッサにインプリメントするのが最適です。この場合、FPGA ファブリックに計算中心のアルゴリズムをコンパイルするのにより大きなスペースが確保されます。図 6-4 に、制御応答時間が非常に低速なシステムの例を示します。

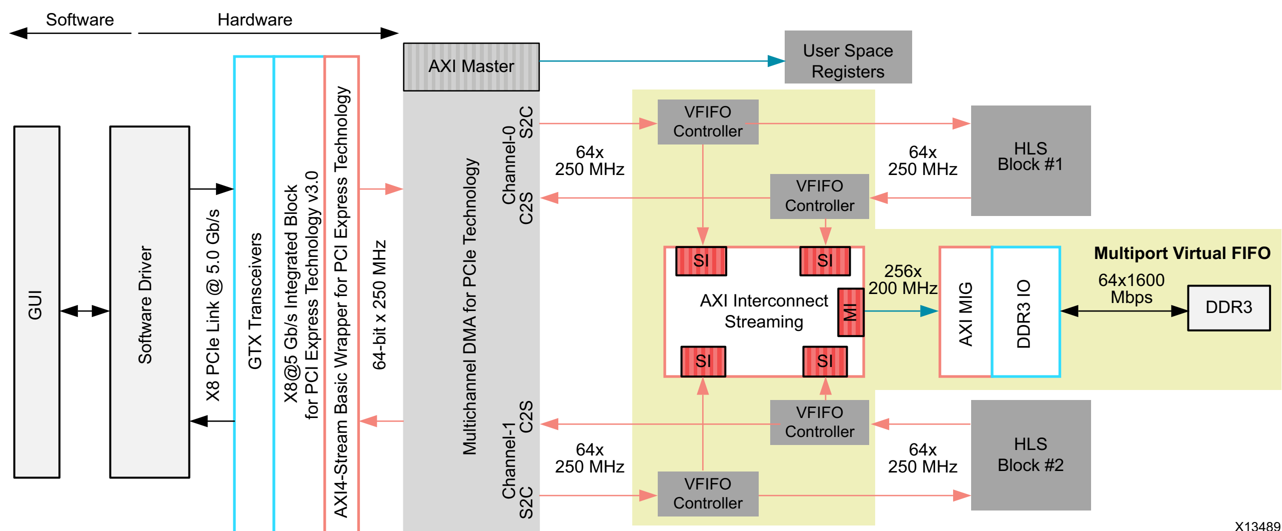

図 6-4: 非常に低速な制御の例

表の「低速」または「中速」のカテゴリに当てはまるのは、中程度のスピードを必要とするデザインで、そのインプリメンテーションには、プロセッサを複数使用するか、または FPGA ファブリックのカスタム ロジックを使用するかのが適しています。これらの場合は、制御アルゴリズムに、ハードウェア モジュールとしてインプリメントする必要のあるクリティカルな関数があります。このタイプのシステムでは、通信レイテンシや制御プロセッサの処理スピードを補うため、ハードウェア コプロセッサが使用されます。図 6-5 に、ハードウェア コプロセッサを必要とするシステムの例を示します。

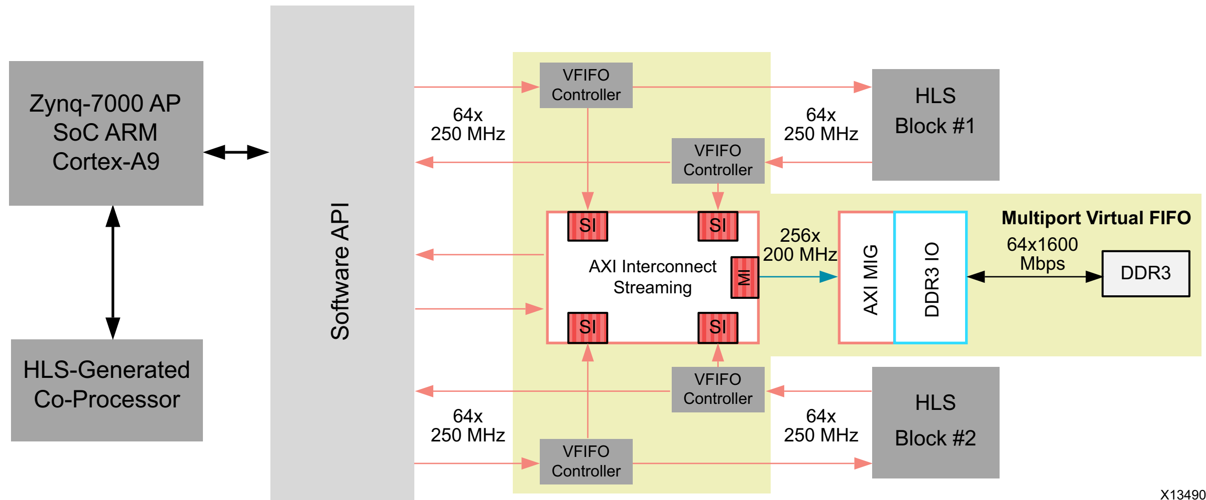

図 6-5: HLS で生成されたコプロセッサを使用したシステムの例

最後のカテゴリは「高速」応答時間です。このカテゴリには、プロセッサで提供できる以上に高速な応答時間と計算スループットが求められる制御アプリケーションが当てはまります。Vivado HLS コンパイラが登場して以来、このカテゴリに入るアルゴリズムが増えてきています。たとえば、Vivado HLS コンパイラは、Zynq-7000 SoC のプロセッサ アクセラレータ モジュールの生成に頻繁に使用されるようになってきています。

UDP パケットの処理

ユーザー データグラム プロトコル (UDP) は、コンピューター ネットワーキング アプロケーションで使用されるステートレス データ転送プロトコルです。このプロトコルは、パケットの配信を確約するものでも、失われたパケットを回復するものでもありません。有線または無線のチャネルでできるだけ高速にパケットを送信するためのプロトコルです。このプロトコルで達成可能なデータ レートは、すべてのパケットを受信することよりもデータ レートが重要となるインターネット電話や動画ストリーミングなどのアプリケーションで標準となっています。

このプロトコルでは、データ配信とステートは追跡されませんが、それでも制御中心のアプリケーションです。
UDP パケット プロセッサは次の点を制御します。

- ライン転送レートで入力データ パケットを解析
- ネットワークからの制御パケットに応答
- 転送するデータ パケットのフォーマット
- 転送チャネル割り込みの処理

これらのすべての制御は、図 6-6 に示すような複雑なステート マシンになります。Vivado HLS コンパイラが登場する以前は、このレベルの複雑な制御は、パフォーマンスを犠牲にすることになったとしても、常にプロセッサで処理されていました。このインプリメンテーションが選択される主な理由は、手動デザイン フローでこのサイズの FSM を効率よく記述してバランスを取るのは難しいからです。

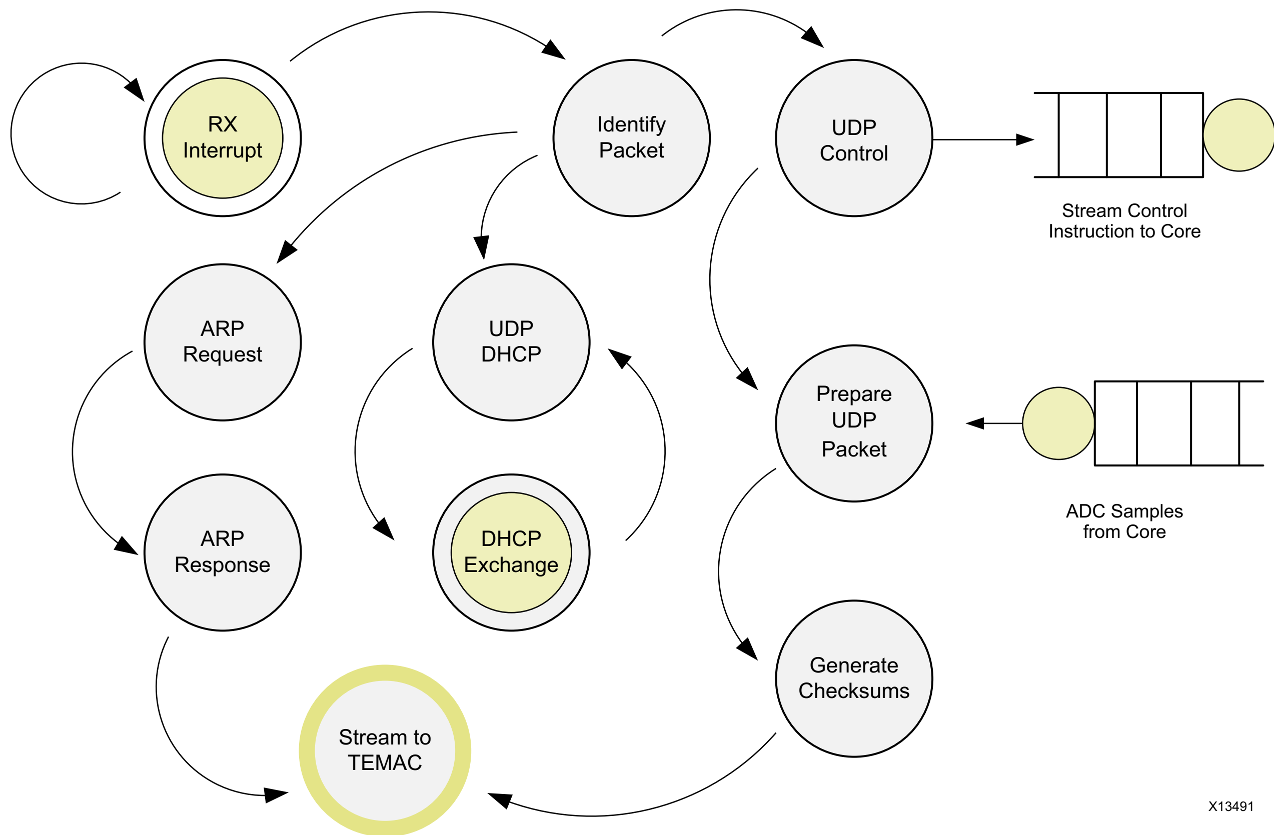

図 6-6: UDP パケットを処理する FSM

図 6-6 に示すように、UDP パケットを処理する FSM はステートが複雑にかかわり合うネットワークです。各ステートで、異なるパケット処理フェーズが処理されます。ステートどうしが複雑にかかわり合っているだけでなく、どのステートもシステム レベルのイベントに割り込まれる可能性があります。これらのイベントは、アプリケーションからのステータス情報の要求をトリガーしたり、次のパケットの処理方法を変更したりする場合があります。計算中心のアプリケーションとは異なり、パケット処理のタスク サイズは確実に定義されていません。どのパケットも解析する必要があるので、タスクはデバイスに電源が入っている限り恒久的です。UDP を処理する FSM のインプリメンテーションはまず、最上位の関数シグネチャから始まります。

図 6-7 に、Vivado HLS コンパイラを使用して FPGA インプリメンテーションをターゲットとした UDP パケット処理エンジンの最上位関数シグネチャを示します。この関数には、モジュールと、システムのそれ以外の部分との間の物理的な通信バッファーをモデル化するため、配列が使用されています。また、配列ではない関数変数をマークするため、volatile キーワードが使用されている点にも注目してください。図 6-6 に示すように、実行段の間は、このコントローラーがシステムからの割り込みを処理できる状態である必要があります。この要件で問題になるのは、C および C++ の両方で指定されている関数変数の動作です。

```
bool udp_proc(const Xuint8 device_mac[6],
                            const Xuint32 rxdescriptor[RX_DESCRIPTOR_RAM_SIZE],
                            const Xuint8 rxram[MAX_RX_RAM],
                            const volatile bool *dma_start_ack,
                            const volatile bool *rx_irq,
                            const volatile bool Xuint8 *rx_status,
                            const volatile bool *tx_rts,
                            volatile bool *dma_start,
                            volatile bool *rx_irq_ack,
                            volatile bool *rx_lock_page)
```

図 6-7: UDP プロセッサの関数シグネチャ

C および C++ では、関数呼び出しが発行されると、関数変数がサンプリングされ、関数メモリ空間のローカル コピーに格納されます。つまり、複数のメモリ空間に同じ変数が格納される可能性があるだけでなく、C/C++ プログラムで次の関数呼び出しがあるまで変数値の変更は検出されません。volatile キーワードは言語的にこの問題を解決します。エンベデッド ソフトウェアの開発者にとっては馴染み深いこの構文は、関数が呼び出されている間は変数値が変化する可能性があることを C/C++ コンパイラに伝えます。そのため、volatile 変数は、コードで使用されるたびに、関数ポートから直接アクセスする必要があります。この構文でデータ アクセスの問題は解決されますが、変数の内部コピーは削除されません。

複数のメモリ空間でデータが重複する可能性がある問題は、const 修飾子を使用して解決できます。この修飾子が関数ポートに適用されると、コンパイラで関数メモリ空間に変数のローカル コピーが作成されなくなり、変数ポートに対して直接読み出しまたは書き込み操作が実行されます。ハードウェアでは、const volatile 修飾子を使用することにより、タスク実行中にシステムが外部入力に応答するので、応答レイテンシが削減されます。

図 6-8 に、UDP 制御 FSM の主な処理を記述したコードを示します。


```
if(!server_init){
            *dma_start = false;
            *rx_irq_ack = false;
            *rx_lock_page = false;
            cs_trigger = false;
            U0:for(int i = 0; i < 6; i++) local_mac[i] = device_mac[i];
             server_init = setup_lan(rx_irq, rxdescriptor,rxram,rx_status,rx_lock_page);

           }else{
             tx_bc = servlet(dma_start,dma_start_ack,rx_irq,rx_irq_ack,
                                       rxdescriptor,rxram,rx_status,rx_lock_page,tx_rts,
                                       &cs_trigger);
             if(tx_bc > 0){
                   temac_0:for(tx_a = 0; tx_a < tx_bc; tx_a++){
                                         if(tx_a == (tx_bc - 1))
                                              temac_din = 0xFF00 | txram[tx_a];
                                         else
                                              temac_din = txram[tx_a];
                                       temac_txif.write(temac_din);
             }
           }
}
```

図 6-8: UDP FSM のメインの関数

UDP 制御 FSM の実行は、初期化と標準実行の 2 段階に分かれます。初期化は、FPGA インプリメンテーションがリセットから解放されるとすぐに開始します。この段階では、ステータス フラグがデフォルト値に設定され、ブロックのメディア アクセス コントローラー (MAC) アドレスがメモリから読み出されます。MAC アドレスは、DHCP アドレスが割り当てられるネットワーク ID です。UDP コントローラーでそのアドレスがブロードキャストされると、ネットワークの IP アドレスを要求および登録するため、ネットワーク制御パケットの処理が開始します。コントローラーがネットワークに正しく登録されると、標準操作モードに切り替わり、UDP パケットの生成が開始します。このコードでは、この機能に加え、1 つの制御中心アプリケーションで制御エレメントと計算エレメントをまとめる方法も示しています。

図 6-8 のコードは、2 つの実行段階をベースにした制御階層の 1 つの階層を示します。実際には、制御中心のアプリケーションはこの例よりも複雑で、階層的な制御構造になります。プロセッサ用の記述と同じ方法で制御階層をキャプチャできるのは、Vivado HLS とほかのハードウェア用コンパイラとの主な違いの 1 つです。

図 6-9 に、Vivado HLS コンパイラ用に階層的な制御を記述する方法の例を示します。この図は図 6-8 の servlet 関数の一部です。servlet 関数は、初期化後の UDP コントローラーのすべての演算段階を制御します。このコードに示すように、次の演算を決めるため、モジュールはシステムレベルの信号と常時通信しています。さらに、このコーディング スタイルでは、入れ子の case 文と、プロセッサ用のコードに一般的な計算関数の組み合わせが維持されます。これにより、C/C++ の機能がキャプチャしやすくなるだけでなく、プロセッサから FPGA にコードを移行しやすくなります。

```
case IDLE :
          *dma_start = false;
          *rx_lock_page = false;
          *rx_irq_ack = false;
          *cs_trigger = false;

          if(*tx_rts) state = TXFIFO_0;
          else if(*rx_irq){
                switch(*rx_status){
                            case 0x00: state = UDP_0; break;
                            case 0x40: state = DHCP_0; break;
                            case 0x20: state = ARP_0; break;
                            default: state = ERROR_0; break;
                            }
          }
          break;
case UDP_0:
          digest_rxdescriptor(rxdescriptor);
          ...
          break;
...
```


図 6-9: UDP 処理での階層制御領域

UDP プロセッサなどの制御中心アプリケーションは、Vivado HLS コンパイラを使用して FPGA にコンパイルおよびインプリメントできます。そのため、このタイプのコードをインプリメントする場合は、アプリケーションに必要な制御コードとそれ以外の関数の間でのリソースのトレードオフを考慮する必要があります。Vivado HLS コンパイラを使用してアプリケーション全体を開発すると、異なるパフォーマンス段階で、制御中心の関数とデータ中心の関数に必要なリソースの量をユーザーが決定できます。Vivado HLS コンパイラには複数の What-if 状況を生成する機能があり、デザイン変数を変更してスループット、エリア、レイテンシがどうなるかを検討できます。

第 7 章

ソフトウェア検証および Vivado HLS

概要

プロセッサ コンパイラと同様、Vivado® HLS コンパイラの出力の質および正確さは入力ソフトウェアによります。この章では、Vivado® HLS コンパイラを対象にしたソフトウェアの質を向上するために推奨されるテクニックを紹介します。一般的なコード記述エラー、その Vivado HLS でのコンパイルへの影響、各問題の解決策を、例を示しながら説明します。また、プログラムの動作を C/C++ のシミュレーション レベルでは完全に検証できない場合の対策についても説明します。

ソフトウェア テストベンチ

Vivado HLS で生成されたモジュールの検証には、ソフトウェア テストベンチが必要です。テストベンチは次のような重要な機能を果たします。

- FPGA インプリメンテーションをターゲットにしたソフトウェアが正しく実行され、セグメンテーション違反が発生しないことを確認。
- アルゴリズムが正しく機能していることを確認。

セグメンテーション違反は、Vivado HLS だけでなく、どのコンパイラでも発生する問題ですが、コード エラーの検出方法は異なります。プロセッサ ベースの実行では、プロセッサで認識されていないメモリ ロケーションにプログラムがアクセスしようとすると、セグメンテーション違反が発生します。このエラーの最も一般的な原因は、メモリがポインターに割り当てられて接続される前に、ユーザー プログラムがポインター アドレスに関連付けられるメモリ ロケーションにアクセスしようとすることです。このエラーは、次のイベント シーケンスに基づいて、ランタイム時に比較的簡単に検出できます。

1. プロセッサによりメモリ アクセス違反が検出され、OS に通知される。
2. OS からエラーの原因となっているプログラムまたはプロセスにエラー信号が送付される。
3. OS からエラー信号を受信したプログラムが終了し、解析用にコア ダンプ ファイルが生成される。

Vivado HLS で生成されたインプリメンテーションでは、プログラムの実行を監視するプロセッサや OS がないので、セグメンテーション違反を検出するのは困難です。セグメンテーション違反が発生したことがわかるのは、回路で間違った結果値が生成された場合のみです。ただし、間違った結果値が計算される原因は複数あるので、これだけではセグメンテーション違反の根本的な原因を判断することはできません。

推奨: Vivado HLS を使用する場合は、まずソフトウェア テストベンチがプロセッサ上で問題なくコンパイルおよび実行されることを確認することをお勧めします。これにより、HLS で生成されたインプリメンテーションでセグメンテーション違反が発生することはなくなります。

ソフトウェア テストベンチのもう 1 つの目的は、FPGA をターゲットにしたアルゴリズムが正しく機能することを確認することです。Vivado HLS コンパイラでは、生成されたハードウェア インプリメンテーションが元の C/C++ コードと機能的に等価であることのみが確約されます。したがって、ハードウェア検証での作業を最小限に抑えるには、良質のソフトウェア テストベンチが必要になります。

アルゴリズムのソフトウェア インプリメンテーションで何千または何億というデータセット テストが実行されれば、ソフトウェア テストベンチの質は高いと言えます。これにより、アルゴリズムが正しくキャプチャされていると確信できます。ただし、多数のテスト ベクターを使用しても、FPGA デザインのハードウェア検証中に Vivado HLS で生成された出力にエラーが検出される可能性はあります。ハードウェア検証中に機能上のエラーが検出された場合は、ソフトウェア テストベンチが完全ではなかったことを意味します。エラーを発生させたテスト ベクターを C/C++ の実行に使用すると、アルゴリズムで間違った構文を発見できます。

重要: エラーは生成された RTL で直接修正しないでください。機能の正しさに問題がある場合は、ソフトウェア アルゴリズムの機能が間違っていることが直接の原因です。

ヒント: Vivado HLS を使用して FPGA インプリメンテーションをターゲットにしたアルゴリズムに使用されるソフトウェア テストベンチには、コーディング スタイルに制限はありません。有効な C/C++ コーディング スタイルや構文であれば、アルゴリズムが機能的に正しいことを検証するために、どんなものでも自由に使用できます。

コード カバレッジ

コード カバレッジとは、デザインのコード文でテストベンチ コードに実行される割合を指します。このメトリクスは、gcov などのツールで生成可能で、アルゴリズムの実行に使用されるテスト ベクターの質を判断するのに役立ちます。

アルゴリズムが十分にテストできていると見なすには、テストベンチのカバレッジが 90% 以上である必要があります。これは、テスト ベクターで case 文、if-else の条件文、for ループのすべての分岐がトリガーされるということです。コード カバレッジ ツールで生成されるレポートでは、全体的なカバレッジを確認できるだけでなく、ある関数のどの部分が実行され、どの部分が実行されていないのかを知ることができます。

図 7-1 に、gcov でテストされたサンプル アプリケーションを示します。

Test Bench Code

```
int main()
{
   int i;
   int B[10];
   int C[10];
   int result;

   for(i=0; i < 10; i++){
        B[i] = i;
        C[i] = i;
   }
   result = example(B,C);
   return result;
}
```

Algorithm Code

```
int example(int B[10], int C[10])
{
   int i;
   int A=0;

   for(i=0; i < 10; i++){
         A += B[i] * C[i];
         if(i == 11)
              A = 0;
   }
   return A;
}
```

図 7-1: コード カバレッジのサンプル アプリケーション

gcov を実行するには、コードをコンパイルする際に、プログラム実行のプロファイリングに必要な情報を生成する追加フラグを使用する必要があります。`example.c` に図 7-1 のコードが 含まれていると、gcov を図 7-2 に示すコマンド シーケンスで実行できます。

```
gcc -fprofile-arcs -ftest-coverage example.c
./a.out
gcov example.c
```

図 7-2: gcov のコマンド シーケンス

gcov の結果を見ると、プログラム行の 92.31% が実行されており、Vivado HLS のコード カバレッジの最低ラインである 90% を満たしています。ところが、表 7-1 に示すように、コードの各行の実行回数に関して興味深いデータが出ています。

表 7-1: gcov で解析されたサンプル コード

実行回数	コード行
-	`int example(int B[10], int C[10])`
1	`{`
-	`int i;`
1	`int A = 0;`
11	`for(i=0; i < 10; i++){`
10	`A += B[i] * C[i];`
10	`if(i == 11)`
0	`A = 0;`
-	`}`
1	`return A;`
-	`}`

for ループ内の `A = 0` という代入が一度も実行されていません。これは、この代入を制御する条件文に問題がある可能性を示しています。条件文 `i == 11` は、図 7-1 に記述されているループの境界では真になることはありません。これが正しい動作であるかどうかをアルゴリズムで確認する必要があります。Vivado HLS では、`A` への `0` の代入など、達成しない C/C++ の文は、回路から削除されるべきデッド コードとして検出されます。

初期化されない変数

変数を宣言したときに 0 に初期化しないコードを記述すると、変数は初期化されません。図 7-3 に、初期化されない変数を含むコードの抜粋を示します。

```
int A;
int B;
...
A = B * 100;
```

図 7-3: 初期化されない変数を含むコードの一部

このコード例では、変数 A は読み出される前に値が代入されるので問題にはなりません。変数 B は値が代入される前に計算に使用されるので問題です。B をこのように使用すると、C および C++ の両方で動作が未定義になります。プロセッサによっては、宣言文で B に 0 を自動的に代入し、問題を回避するものもありますが、Vivado HLS ではそのような方法は使用されません。

Vivado HLS では、ユーザー コードに未定義の動作があると、それが最適化によりインプリメンテーションから削除される可能性があります。この最適化の影響がほかの部分にも及び、回路が機能しなくなる可能性があります。生成されたインプリメンテーションに空の RTL ファイルが含まれている場合、このタイプのエラーが発生しているということです。

このタイプのエラーをさらに確実に検出するには、valgrind や Coverity などのコード解析ツールを使用します。どちらのツールも、ユーザー プログラムに初期化されていない変数があると検出されます。初期化されない変数の問題は、ほかのソフトウェア品質問題と同様に、コードを Vivado HLS を使用してコンパイルする前に、解決しておく必要があります。

範囲外のメモリ アクセス

Vivado HLS では、メモリ アクセスは、配列上の演算、またはポインターを介した外部メモリ上の演算として表現されます。範囲外のメモリ アクセスは、Vivado HLS によりメモリ ブロックに変換された配列からのメモリ アクセスのことを指します。図 7-4 に、範囲外のメモリ アクセスを記述したサンプル コードを示します。

```
int A[10];
...
for(i = 0; i < 11; i++){
      A[i] = i + 5;
}
```

図 7-4: 範囲外のメモリ アクセスのコード例

このコードは、割り当てられているメモリの範囲外にあるロケーションの配列 A にデータを書き込もうとしています。プロセッサ コンパイラでは、このようなアドレス オーバーフローがあると、アドレス カウンターが 0 にリセットされます。つまり、図 7-4 のコードをプロセッサで実行すると、位置 A[0] の内容は 5 ではなく 15 になります。この結果は機能的には間違っていますが、このようなエラーがプログラムをクラッシュさせることは通常ありません。

Vivado HLS を使用している場合、無効なアドレスにアクセスすると、一連のイベントがトリガーされ、生成された回路で回復不可能なランタイム エラーが発生します。Vivado HLS のインプリメンテーションでは、ソフトウェア アルゴリズムが正しく検証されていると想定されるので、生成された FPGA インプリメンテーションにはエラー リカバリ ロジックは含まれません。このため、図 7-4 のコードを配列 A の値を格納するブロック RAM エレメントにインプリメントすると、無効なメモリ アドレスが生成されます。その後、ブロック RAM により、Vivado HLS インプリメンテーションでは予期されていなかったエラー条件が出力されますが、このエラーは無視されます。ブロック RAM からのエラーが無視されたため、システムがハングし、デバイスをリブートしないと解決できなくなります。

回路をコンパイルする前にこうした問題を検出するには、valgrind などのダイナミック コード チェッカーを使用することをお勧めします。valgrind は、C/C++ プログラムの質をチェックしてプロファイルするためのツールです。valgrind に含まれる Memcheck ツールにより、コンパイル済みの C/C++ プログラムが実行され、その実行中のメモリ操作がすべて監視されます。このツールにより、次のようなクリティカルな問題がフラグされます。

- 初期化されていない変数の使用 (図 7-3)
- 無効なメモリ アクセス要求 (図 7-4)

推奨: FPGA 実行用のソフトウェア関数を Vivado HLS を使用してコンパイルする前に、ダイナミック コード チェッカーで検出される問題をすべて解決しておくことをお勧めします。

協調シミュレーション

C/C++ プログラムの解析および機能性テスト用のツールを活用すると、Vivado HLS インプリメンテーションに影響を及ぼす問題のほとんどを事前に検出できます。ただし、これらのツールでは、順次 C/C++ プログラムを並列処理しても正しく機能するかどうかを検証することはできません。この問題は、Vivado HLS コンパイラの協調シミュレーション プロセスで解決されます。

協調シミュレーションとは、生成された FPGA インプリメンテーションをソフトウェアのシミュレーション中に使用されるのと同じ C/C++ テストベンチで実行するプロセスのことです。C/C++ テストベンチと生成済み RTL との通信は、Vivado HLS により処理され、ユーザーには透過的です。このプロセスの一部として、Vivado HLS から Vivado シミュレータなどのハードウェア シミュレータが起動し、RTL がデバイス上でどのように機能するのかがエミュレートされます。このシミュレーションの主な目的は、ユーザーによる並列処理ガイダンスによりアルゴリズムの機能が正しいものでなくなっていないかをチェックすることです。

デフォルトでは、元の C/C++ で記述されたコードと機能的に等価になるように、Vivado HLS は並列処理前のアルゴリズムの依存関係に従います。アルゴリズムの依存関係を完全に解析できない場合は、保守的に依存関係に従います。この結果、アプリケーションがターゲット パフォーマンス目標に達しないインプリメンテーションが生成される可能性があります。図 7-5 に、Vivado HLS で保守的な処理がトリガーされるコード例を示します。

```
for(i=0; i < M; i++){
      A[k+i] = A[i] + .....;
       B[i] = A[i] * .....;
}
```

図 7-5: 保守的な HLS インプリメンテーションをトリガーする依存関係のコード例

このコードは、配列 A および B で実行しているループを示しており、解析の問題は配列 A で発生します。配列 A へのインデックスは、ループ変数 i および変数 k によって決まります。この例では、変数 k は関数パラメーターで、その値はコンパイル時には未知です。そのため、A[k+i] への書き込みが、B[i] の計算に使用される A[i] の読み出しとは別のロケーションで実行されることを Vivado HLS では証明できません。そこで、Vivado HLS では、アルゴリズムに依存関係があると想定し、A[k+i] および B[i] の計算が元の C/C++ ソースに記述されている順に実行されるとみなされます。ユーザーは、この依存関係を無効にし、Vivado HLS で A[k+i] と B[i] を並列計算する回路が生成されるようにすることができます。この効果は生成された回路のみに反映されるので、協調シミュレーションによってしか検証できません。

協調シミュレーションを使用する場合は、これがプロセッサ上で実行される並列ハードウェアのシミュレーションであることを忘れないようにすることが重要です。そのため、C/C++ のシミュレーションよりも約 10000 倍低速になります。また、協調シミュレーションは、アルゴリズムが機能的に正しいことを検証するのが目的ではなく、Vivado HLS コンパイラへのユーザーのガイダンスによりアルゴリズムが機能しなくなっていないかを確認するのが目的です。

推奨: 協調シミュレーションは、アルゴリズムの機能検証で使用されるテスト ベクターのサブセットでのみ実行することを推奨します。

C/C++ 検証が不可能な場合

Vivado HLS のほとんどのユース ケースでは、アルゴリズムの機能的な正しさを C/C++ シミュレーションで完全に検証できます。ただし、アルゴリズムの C/C++ 記述には、Vivado HLS でコンパイルを実行する前に完全には検証できないものもあります。図 7-6 に、そのようなコード例を示します。

```
case IDLE :
            *dma_start = false;
            *rx_lock_page = false;
            *rx_irq_ack = false;
            *cs_trigger = false;

            if(*tx_rts) state = TXFIFO_0;
            else if(*rx_irq){
                  switch(*rx_status){
                              case 0x00: state = UDP_0; break;
                              case 0x40: state = DHCP_0; break;
                              case 0x20: state = ARP_0; break;
                              default: state = ERROR_0; break;
                  }
            }
            break;
```

図 7-6: volatile 型を使用したコード例

このコードは、C で記述された UDP パケット処理エンジンの抜粋です。この例では、すべてのポインターが volatile キーワードを使用して宣言されています。volatile キーワードは、デバイスのドライバーを開発するときによく使用されます。このキーワードは、ポインターが関数実行中に変化する可能性のあるストレージ エレメントに接続されていることをコンパイラに通知します。このタイプのポインターは、ソース コードで指定されるたびに読み出しまたは書き込みする必要があります。ポインター アクセスを結合する従来のコンパイラ最適化も、volatile キーワードによりオフになります。

volatile 型のデータの問題は、コードの動作を C/C++ シミュレーションでは完全には検証できないことです。C/C++ シミュレーションでは、テスト中の関数を実行しているときにポインターの値を変更することはできません。そのため、このタイプのコードは、Vivado HLS でのコンパイル終了後の RTL シミュレーションでしか完全に検証できません。生成された回路で C/C++ ソースの各 volatile ポインターに可能なすべてのケースをテストするため、RTL テストベンチをユーザーが記述する必要があります。この場合、協調シミュレーションは C/C++ シミュレーションで使用可能なテスト ベクターにより制限されるため、使用できません。

第 8 章

複数のプログラムの統合

概要

ほとんどのプロセッサでアプリケーションを実行するため複数のプログラムを実行するのと同様に、FPGA もアプリケーションを実行するため複数のプログラムまたはモジュールをインスタンシエートします。この章では、FPGA で複数のモジュールを接続する方法およびプロセッサを使用したこれらのモジュールの制御方法を説明します。この章の例では、プロセッサと FPGA ファブリックの接続を示すため、ザイリンクス Zynq®-7000 SoC を使用します。

Zynq-7000 SoC は、新しいクラスのデバイスの中で低消費電力ソフトウェア実行をターゲットとする初のデバイスです。このデバイスは、Arm® Cortex™-A9 のマルチコア プロセッサと FPGA ファブリックを 1 つのチップに組み合わせています。このデバイスでの統合により、コプロセッサまたはアクセラレーション ソリューションに関連する通信レイテンシやボトルネックが取り除かれます。また、プロセッサ上で実行するコードと、FPGA 用に Vivado® HLS でコンパイルされたコードとの間でデータを転送するための PCIe® ブリッジも不要です。これら 2 つの計算ドメインのインターコネクトには、AXI (Advanced eXtensible Interface) プロトコルが使用されます。

AXI

AXI プロトコルは、マイクロコントローラー バス アーキテクチャの AMBA® (Advanced Microcontroller Bus Architecture) ファミリの 1 つです。この規格は、システム内のモジュール間のデータ転送方法を定義します。Zynq-7000 SoC 上で実行するアプリケーションを対象した AXI 通信のユース ケースには次のものがあります。

- メモリ マップド スレーブ
- メモリ マップド マスター
- ダイレクト ポイント ツー ポイント ストリーム

注記: AXI の詳細およびザイリンクス FPGA へのインプリメンテーション方法は、『Vivado Design Suite: AXI リファレンス ガイド』(UG1037) [参照 2] を参照してください。

メモリ マップド スレーブ

AXI4-Lite はメモリ マップド スレーブ接続で、プロセッサ ベースのシステムのデバイス ドライバーと同じ通信メカニズムを使用します。プロセッサ コードは、デバイス ドライバーに関数呼び出しを発行し、スレーブのアクセラレータ コアにアクセスします。デバイス ドライバー (Vivado HLS で自動生成) は、タスク実行を設定してトリガーするため、アクセラレータのレジスタにアクセスします。これらのレジスタはプロセッサのメモリ空間にあり、ドライバーなしで直接アクセスすることも可能です。

FPGA ファブリックのスレーブ アクセラレータは、それ自体ではデータ転送を開始できません。特にこのタイプのインターフェイスでは、アクセラレータがそのタスクを完了するためにメイン メモリとのデータ転送を開始できません。図 8-1 に、このインターフェイスのトランザクションを示します。この図は、1 つのトランザクションでクロック サイクルがどこで消費されるかを示しています。トランザクションのシーケンスおよびタイミング バジェットを理解すると、このインターフェイスがアプリケーションに適しているかどうか、アプリケーションのパフォーマンスにどのような影響を与えるかを正しく判断することが可能になります。

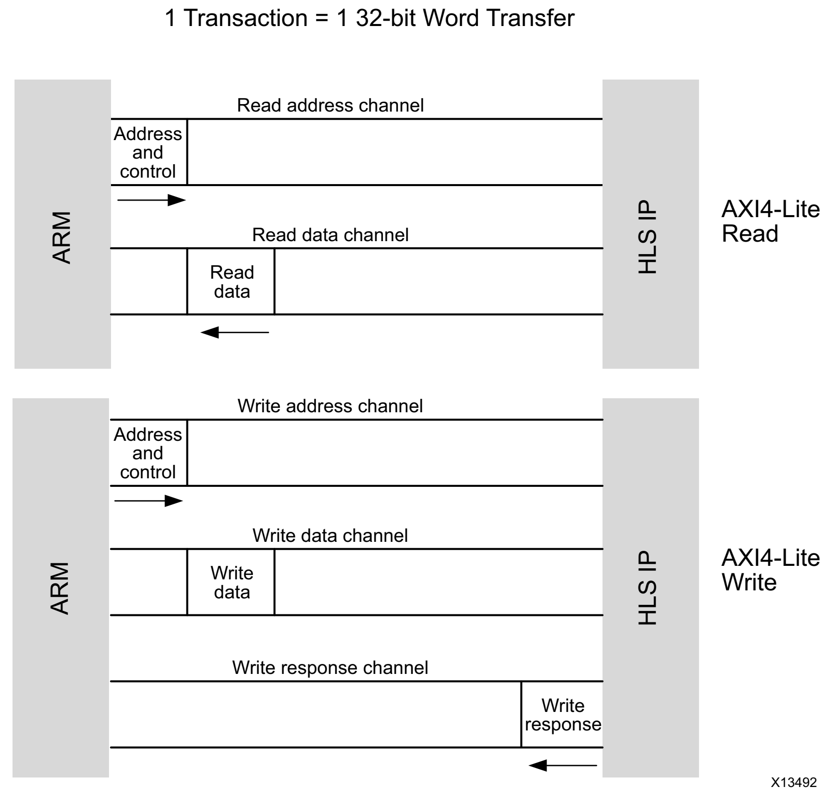

図 8-1: AXI4-Lite のトランザクション

メモリ マップド マスター

AXI4 はメモリ マップド マスター インターフェイスで、HLS で生成されたモジュールがプロセッサの介入なしに DDR メモリなどのデバイスへデータ トランザクションを開始できるようにします。このインターフェイスを使用したブロックは、プロセッサがメイン メモリからデータをコピーして転送するのにかかる時間を省くことができるので、アプリケーションの計算スループットを高めることができます。

AXI4 インターフェイスに関連付けられている関数ポートは、プロセッサからはアクセスできません。そのため、AXI4 インターフェイスを使用するモジュールには、AXI4-Lite インターフェイスに接続されている関数パラメーターをいくつか含めておくのがベスト プラクティスとして推奨されます。スレーブ インターフェイスでは、関数がタスク データをフェッチするベース アドレスをプロセッサが通信できます。トランザクションのベース アドレスが設定されたら、メモリとアクセラレーション モジュールとの間のデータ転送からプロセッサを削除できます。

図 8-2 に AXI4 インターフェイスのトランザクションのタイミングを示します。この図は、トランザクションのシーケンスと、それに関連したオーバーヘッドを示しており、このインターフェイスが特定のアプリケーションに適しているかどうかの判断に役立ちます。

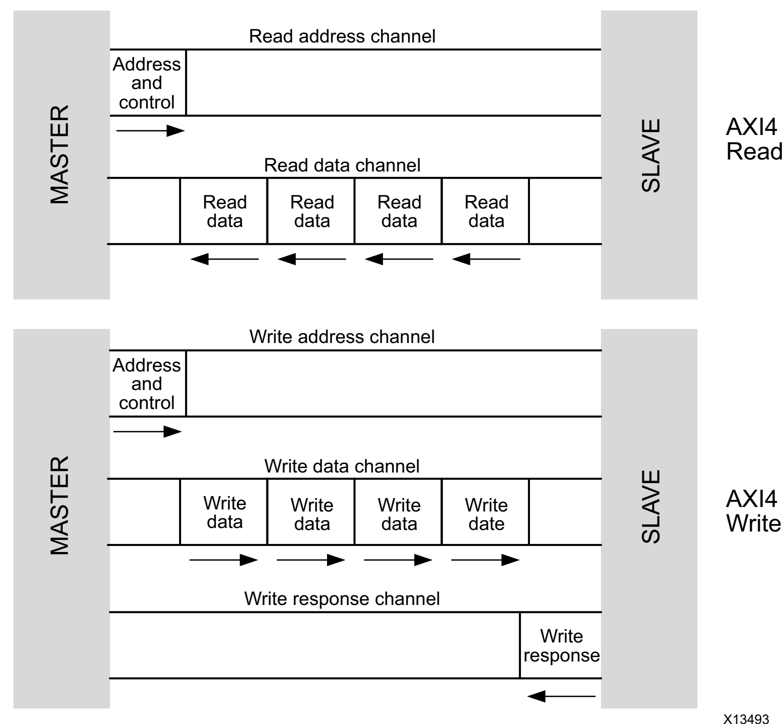

図 8-2: AXI4 のトランザクション

ダイレクト ポイント ツー ポイント ストリーム

AXI4-Stream は、FPGA ファブリック内の 2 つのモジュール間を接続するポイント ツー ポイントの通信チャネルです。AXI4 の場合と同様に、この転送チャネルはプロセッサのメモリ空間では見えません。また、アドレス指定やメモリからのデータ フェッチに関するオーバーヘッドもありません。モジュール間のデータ転送には、FIFO を使用します。

AXI4-Stream はソフトウェア開発における関数間のキューのようなもので、FPGA ファブリックにコンパイルされる関数間のデータ転送にはこの AXI4-Stream が優先的に使用されます。このタイプのデータ転送チャネルに接続される関数は、並列に実行され、チャネルのステートの基づいて自己同期化します。ストリームの入力に接続されている関数は「プロデューサー」と呼ばれ、チャネルに空きがある限りデータを送信します。ストリームの出力に接続されている関数は「コンシューマー」と呼ばれ、チャネルが空でない限りデータを受信します。

コンシューマーもプロデューサーも、それぞれ独立して AXI4-Stream チャネルとデータをやり取りします。チャネルのステートにより、関数はトランザクションを完了するか、またはチャネルの準備が整うまで待機します。関数の全体的なスループットがシステム レベルの要件を満たしていれば、データが失われたり、スキップされたりすることはありません。

図 8-3 に、AXI4-Stream データ転送チャネルのトランザクションのタイミングを示します。このチャネルからはアドレス指定ロジックは提供されません。またストレージの容量はユーザーにより定義されます。AXI4-Stream の深さはデフォルトでは 1 で、プロデューサーとコンシューマーはロックステップで動作します。プロデューサーとコンシューマーがどの程度カップリングするかは、AXI4-Stream チャネルのストレージ量によります。

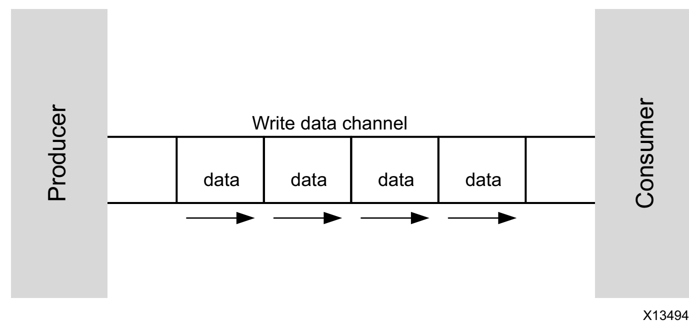

図 8-3: AXI4-Stream のトランザクション

デザイン例: Zynq-7000 SoC 上で実行するアプリケーション

このデザイン例では、プロセッサ コードを Zynq-7000 SoC 上で実行するアプリケーションに変換する方法を説明します。移行プロセスを次の手順で実行していきます。

- プロセッサ コードの解析および分割
- Vivado HLS でのプログラムのコンパイル
- Vivado IP インテグレーターを使用したシステムの構築
- プロセッサ コードと FPGA ファブリック関数の接続

注記: Zynq-7000 デバイス内にある Arm Cortex-A9 プロセッサでは、シングル プログラム実行と、Linux などの完全な OS の両方をサポートできます。いずれの場合も、アプリケーションの構築に必要な手順は同じです。ここでは、シングル プログラム実行に焦点を当て、アプリケーション移行プロセスを説明します。

プロセッサ コードの解析および分割

Zynq-7000 デバイスをターゲットとするソフトウェア アプリケーションのほとんどは、標準の x86 型プロセッサまたは DSP プロセッサ上で実行するアプリケーションとして開発を開始します。そのため、デザインの移行プロセスの最初の手順では、Arm Cortex-A9 プロセッサ用にプログラムをコンパイルし、そのパフォーマンスを解析します。Arm プロセッサ上で実行するプログラムのパフォーマンス解析データを使用すると、プロセッサと FPGA ファブリックとの間で元のコードをどのように分割するかを決定するのに役立ちます。

図 8-4 に、元のプロセッサ コードを示します。

```
#include <iostream>
#include "strm_test.h"

using namespace std;

int main(void)
{
   unsigned err_cnt = 0;
   data_out_t hw_result, expected =0
   strm_data strm_array[MAX_STRM_LEN];
   strm_param_t strm_len = 42;

   // Generate expected result
   for(int i = 0; i < strm_len; i++){
         expected += i + 1;
    }

    producer(strm_array,strm_len);
    consumer(&hw_result,strm_array,strm_len);

    // Test result
    if(hw_result != expected){
        cout << "!!!ERROR";
        err_cnt++;
    }else{
         cout << "*** Test Passed";
    }
    cout << "-expected:" << expected;
    cout << " Got:" << hw_result << endl;
    return err_cnt;
}
```

図 8-4: プロセッサ コード

このデザインは、プロデューサー (producer) とコンシューマー (consumer) という 2 つのサブ関数を呼び出す main 関数で構成されています。Arm プロセッサにコンパイルした後、プログラム パフォーマンスを解析するには次の 2 つの方法があります。

- 時間の計測

  この方法では、コードにタイマーを追加し、プロセッサ上で各サブ関数の実行にかかる時間を計測します。

- コード プロファイリング ツールの使用

  この方法は 1 番目の方法のようにコードを変更せず、gprof などのツールを使用して、1 つの関数の実行にかかる時間を計測し、その関数が何回呼び出されるのかの統計を取ります。

この例で gprof を実行すると、producer および consumer の両方の関数がアプリケーションのパフォーマンスのボトルネックになっていることがわかります。そこで、この両方の関数を FPGA ファブリックにインプリメントすることにします。関数を FPGA インプリメンテーション用に指定したら、関数ポートを解析して最適なハードウェア インターフェイスを判断する必要があります。

図 8-5 に、producer 関数のシグネチャを示します。

```
void producer(strm_data_t strm_out[MAX_STRM_LEN],strm_param_t strm_len)
```

図 8-5: Producer 関数のシグネチャ

producer 関数には、次のポートが含まれています。

- `strm_out`

  このポートは、関数出力に使用される配列で、consumer 関数の対応する入力に接続されます。producer および consumer の両方の関数が順次キューとしてこの配列にアクセスするので、これに最適なハードウェア インターフェイスは AXI4-Stream です。

- `strm_len`

  この関数パラメーターは、producer により供給される必要がある入力です。そのため、このポートは AXI4-Lite インターフェイスにマップする必要があります。

図 8-6 に、consumer 関数のシグネチャを示します。

```
void strm_consumer(data_out_t *dout, strm_data_t
        strm_in[MAX_STRM_LEN], strm_param_t strm_len)
```

図 8-6: Consumer 関数のシグネチャ

consumer 関数には、次のポートが含まれています。

- `strm_in`

  この配列ポートは、producer 関数と同じ配列に接続されます。そのため、このポートは AXI4-Stream インターフェイスに接続する必要があります。

- `strm_len`

  この関数パラメーターは、producer 関数のものと同じ役割を果たします。producer 関数と同様、AXI4-Lite インターフェイスにインプリメントされます。

- `dout`

  これは出力ポートです。このデザインにはほかに FPGA ファブリック モジュールがないので、値をプロセッサに戻すしか選択肢はありません。プロセッサ割り込みを発行すると、FPGA ファブリックからプロセッサに直接データが戻されます。割り込みに対する肯定応答が送信されたら、データを入れるメモリ スペースがあるかどうかをプロセッサがクエリします。dout 関数パラメーターは、プロセッサ プログラムからアクセスできるように、AXI4-Lite インターフェイスにマップする必要があります。

Vivado HLS でのプログラムのコンパイル

FPGA ファブリックで実行する関数を特定したら、次に Vivado HLS でのコンパイル用にソース コードを準備します。この例では、producer および consumer の両方の関数を独立したモジュールとして FPGA ファブリックにインプリメントします。1 つのコンパイル プロジェクトは FPGA ファブリックで 1 つのモジュールになるので、Vivado HLS を 2 回実行して producer および consumer のモジュールを生成する必要があります。

推奨: 複数のプロジェクトまたはモジュールを使用する場合は、ソース コードを個別のファイルに分けることをお勧めします。これにより、1 つのモジュールで発生したコンパイルの問題が、ほかのモジュールに影響するのを防ぐことができます。

Vivado HLS でのコンパイルは、ツール コマンド言語 (Tcl) スクリプト ファイルを使用して制御できます。Tcl スクリプト ファイルはコンパイルの makefile のようなもので、インプリメントする関数とターゲット FPGA デバイスを指定します。

図 8-7 に、producer 関数の Vivado HLS コンパイル用の Tcl スクリプト ファイルを示します。

```
## Project Setup
open_project producer_prj
set_top producer
add_file strm_producer.cpp
add_file -tb strm_consumer.cpp
add_file -tb strm_test.cpp

### Solution Setup
open_solution "solution1"
set_part {xc7z020clg484-1}
create_clock -period 5

### Compilation
csynth_design
export_design -format ipxact
```

図 8-7: producer 関数の Vivado HLS コンパイル用の Tcl スクリプト ファイル

このスクリプトは次のセクションに分かれています。

- Project Setup (プロジェクト設定)

  このセクションには、ソース ファイルおよびコンパイルする関数名が含まれます。Vivado HLS では、対話形式でデザインのソース コードに指示子やプラグマを適用していきます。その後のデザインの各調整は「ソリューション」と呼ばれます。どのプロジェクトにも最低 1 つのソリューションがあります。

- Solution Setup (ソリューション設定)

  このセクションでは、ソフトウェア関数をコンパイルするクロック周波数およびデバイスが設定されます。指示子を使用する場合、このセクションにソリューション指示子が含められます。

- Compilation (コンパイル)

  このセクションでは、RTL 生成およびパッケージが実行されます。Vivado HLS プログラムの Zynq-7000 デバイス アプリケーションへのアセンブリには、システム構築ツールである Vivado IP インテグレーターが必要です。IP インテグレーターでは、ソフトウェア オブジェクト ファイルと同等のものにモジュールをパックする必要があります。

  注記: IP および IP インテグレーターの詳細は、『Vivado Design Suite ユーザー ガイド: IP を使用した設計の概要 (UG896) [参照 3] および 『Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計 (UG994) [参照 4] を参照してください。

producer および consumer 関数の最適化には、生成されたモジュールおよびそのインターフェイスの並列処理を指定するため、プラグマが必要です。図 8-8 に、最適化された producer 関数のコードを示します。

```
#include "strm_test.h"

void producer(strm_data_t strm_out[MAX_STRM_LEN],strm_param_t strm_len)
{
//Interface Behavior
#pragma HLS INTERFACE ap_none port=strm_len
#pragma HLS INTERFACE ap_fifo port=strm_out

//Interface Mapping
#pragma HLS RESOURCE variable=strm_out core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
#pragma HLS RESOURCE variable=strm_len core=AXIS4LiteS metadata="-bus_bundle CONTROL_BUS"

for(int i = 0; i < strm_len; i++){
#pragma HLS PIPELINE
     strm_out[i] = i + 1;
    }
}
```

図 8-8: 最適化された producer 関数

producer 関数はパイプライン プラグマによって並列処理されます。これにより、i および i+1 の反復を 1 クロック サイクルずらして開始するインプリメンテーションが作成されます。パイプライン プラグマだけでなく、コードにはインターフェイス プラグマも使用されています。

インターフェイス プラグマは、FPGA ファブリックでのモジュールの接続を定義します。この定義プロセスは、インターフェイスの動作とインターフェイスのマップに分かれています。この例では、次のようになります。

1. strm_out ポートの ap_fifo インターフェイスにより、配列がハードウェア FIFO に変換されます。
2. リソース プラグマを使用して、物理的な FIFO が AXI4-Stream インターフェイスにマップされます。
3. strm_len 関数パラメーターはまず ap_none インターフェイスの動作に割り当てられ、それから AXI4-Lite インターフェイスにマップされます。

   注記: AXI4-Lite インターフェイスにより、プロセッサから strm_len の値が正しい順序で処理されます。そのため、Vivado HLS で生成されたモジュールでは、このポート上でさらに同期化を実行する必要はありません。

図 8-9 に、consumer 関数のコードを示します。この関数は、producer 関数と同じように最適化され、同じプラグマが含まれます。

```
#include "strm_test.h"

void consumer(data_out_t *dout, strm_data_t strm_in[MAX_STRM_LEN],
                           strm_param_t strm_len)
{
//Interface Behavior
#pragma HLS INTERFACE ap_none port=dout
#pragma HLS INTERFACE ap_none port=strm_len
#pragma HLS INTERFACE ap_fifo port=strm_in

//Interface Mapping
#pragma HLS RESOURCE variable=strm_in core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
#pragma HLS RESOURCE variable=strm_len core=AXIS4LiteS metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE variable=dout core=AXIS4LiteS metadata="-bus_bundle CONTROL_BUS"

data_out_t accum = 0;

    for(int i = 0; i < strm_len; i++){
#pragma HLS PIPELINE
     accum += strm_in[i];
    }
    *dout = accum;
}
```

図 8-9: 最適化された consumer 関数

Vivado IP インテグレーターを使用したシステムの構築

Vivado IP インテグレーターは、システム構築に使用するザイリンクス FPGA デザイン ツールです。このツールの 1 つの使用方法として、Vivado HLS コンパイラで生成されたブロックを、ユーザー アプリケーションを実行するプロセッシング プラットフォームに接続できます。IP インテグレーターはソフトウェア開発におけるリンカーのようなもので、すべてのプログラム オブジェクトを 1 つのビットストリームにまとめます。ビットストリームとは、FPGA ファブリックのプログラムに使用するバイナリ ファイルのことです。

プロセッサ コードと FPGA ファブリック関数の接続

FPGA ファブリックをプログラムするバイナリを IP インテグレーターで作成したら、次にプロセッサ上で実行するソフトウェアを作成する必要があります。このソフトウェアの目的は、FPGA ファブリックの関数を初期化し、実行を開始して、デバイスからの結果を受信することです。アプリケーション全体の機能を元のプロセッサ コードと等価にするには、FPGA ファブリックで実行する各関数に、Arm Cortex-A9 プロセッサ上で実行する次の機能をコードに含める必要があります。

- アドレス マップ
- 初期化
- 開始関数
- 割り込み処理ルーチン (ISR)
- プロセッサ実行テーブルでの割り込み登録
- システムを実行するための新しい main 関数

これらの機能は、FPGA ファブリックで実行している producer および consumer 関数の両方に必要です。図 8-10 には、producer 関数のコードをのみを示します。

```
XStrm_producer_Config producer_config={
             0,
             XPAR_STRM_PRODUCER_TOP_0_S_AXI_CONTROL_BUS_BASEADDR
};
```

図 8-10: プロセッサのプログラム空間でのハードウェア関数のコンフィギュレーション

このコードは、プロセッサのプログラム空間での producer ハードウェア モジュールのコンフィギュレーションを示しています。最初のパラメーターは、producer 関数のインスタンスでファブリックからアクセスされるものを示します。ファブリックには producer のインスタンスが 1 つしかないので、このパラメーターの値は 0 になります。ベース アドレスは IP インテグレーターでのシステム構築の段階で定義されます。このアドレスは、プロセッサからアクセス可能なメモリ空間内のメモリ マップド アクセラレータのロケーションを表わします。

図 8-11 に、producer ハードウェア モジュールがプロセッサ上で実行しているプログラムにアクセスできるようにするのに必要な初期化関数を示します。

```
int ProducerSetup(){
       return XStrm_producer_Initialize(&producer,&producer_config);
}
```

図 8-11: ハードウェア関数の初期化

図 8-12 では、producer ハードウェア モジュールがタスク実行を開始するよう設定しています。この関数は、モジュールの割り込みを既知のステートに設定し、タスク実行を開始します。

```
void ProducerStart(void *InstancePtr){
        XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;
        XStrm_producer_InterruptEnable(pProducer,1);
        XStrm_producer_InterruptGlobalEnable(pProducer);
        XStrm_producer_Start(pProducer);
}
```

図 8-12: ハードウェア関数の開始

図 8-13 の ISR は、FPGA ファブリックの producer 関数からの割り込みにプロセッサがどう応答するかを説明しています。ISR の内容はアプリケーションによって異なります。このコードは、Zynq-7000 デバイスの Vivado HLS で生成されたモジュールと正しく通信するのに必要な最小限の ISR を示しています。

```
void Producer(void *InstancePtr){
        XStrm_producer *pProducer = (XStrm_producer *)InstancePtr;

        //Disable the global interrupt from the producer
        XStrm_producer_InterruptGlobalDisable(pProducer);
        XStrm_producer_InterruptDisable(pProducer,0xffffffff);

        //clear the local interrupt
        XStrm_producer_InterruptClear(pProducer,1);

        ProducerDone = 1;
        //restart the core if it should be run again
        if(RunProducer){
            ProducerStart(pProducer);
        }
}
```

図 8-13: 割り込み処理ルーチン (ISR)

すべての割り込み処理ルーチン (ISR) は、プロセッサ実行テーブルに登録しておく必要があります。プロセッサの割り込みコントローラーが初期化されたら、main プログラムによりユーザー アプリケーションの実行が開始されます。図 8-14 に、Zynq-7000 デバイスの実行テーブルを設定する方法を示します。

```
int SetupInterrupt()
{
   //This function sets up the interrupt on the ARM
   int result;
   XScuGic_Config *pCfg =
   XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
   if(pCfg == NULL){
          print("Interrupt Configuration Lookup Failed\n\r");
          return XST_FAILURE;
   }
   result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
   if(result != XST_SUCCESS){
            return result;
   }
   //self test
   result = XScuGic_SelfTest(&ScuGic);
   if(result != XST_SUCCESS){
              return result;
   }
   // Initialize the exception handler
   Xil_ExceptionInit();
   //Register the exception handler
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
   //Enable the exception handler
   Xil_ExceptionEnable();
   //Connect the Producer ISR to the exception table
   result = XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INT,
                                                (Xil_InterruptHandler)ProducerIsr,&producer);
   if(result != XST_SUCCESS){
                  return result;
    }
    //Connect the Consumer ISR to the exception table
    result = XScuGic_Connect(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR,
                                               (Xil_InterruptHandler)ConsumerIsr,&consumer);
    if(result != XST_SUCCESS){
                  return result;
    }
    //Enable the interrupts for the Producer and Consumer
    XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_PRODUCER_TOP_0_INTERRUPT_INTR);
    XScuGic_Enable(&ScuGic,XPAR_FABRIC_STRM_CONSUMER_TOP_0_INTERRUPT_INTR);
     return XST_SUCCESS;
}
```

図 8-14: プロセッサ実行テーブルの設定

図 8-15 に、アプリケーションの新しい main プログラムを示します。ハードウェアおよびプロセッサ環境が設定されたら、この例のプロセッサで実行すべき計算はもうありません。すべての計算は、Vivado HLS のコンパイルにより FPGA ファブリックに移動してます。この例でのプロセッサの目的は、各ハード ウェア モジュールでタスクを起動し、モジュールがタスクを完了した後に結果値を収集することです。

```
int main()
{
  Init_platform();

  print("Producer Consumer Example\n\r");
  int length;
  int status;
  int result;
  length = 50;
  printf("Length of stream = %d\n\r",length);

  status = ProducerSetup();
  if(status != XST_SUCCESS){
     print("Producer setup failed\n\r");
  }
  status = ConsumerSetup();
  if(status != XST_SUCCESS){
      print("Consumer setup failed\n\r");
  }
  //Setup the interrupt
  status = SetupInterrupt();
  if(status != XST_SUCCESS){
       print("Interrupt setup failed\n\r");
   }

  XStrm_consumer_SetStrm_len(&consumer, length);
  XStrm_producer_Set_Strm_len(&producer,length);

  ProducerStart(&producer);
  ConsumerStart(&consumer);

  while(!ProducerDone) print("waiting for producer to finish\n\r");
  while(!ConsumerResult) print("waiting for consumer to finish\n\r");

  result = XStrm_consumer_GetDout(&consumer);
  printf("Consumer result = %d\n\r",result);
  print("Finished\n\r");

  cleanup_platform();

  return 0;
}
```

図 8-15: プロセッサの main 関数

完成したアプリケーションの検証

概要

FPGA デザインにおける完全なアプリケーションとは、デザインのソフトウェア記述により表現された機能をインプリメントするハードウェア システムのことを指します。Vivado® HLS コンパイラを使用して FPGA 上に構築できるシステムには、主に次の 2 つのカテゴリがあります。

- スタンドアロンの計算システム
- プロセッサ ベースのシステム

スタンドアロンの計算システム

スタンドアロンの計算システムとは、1 つまたは複数の Vivado HLS 生成のモジュールを接続することにより作成された、ソフトウェア アプリケーションをインプリメントする FPGA インプリメンテーションです。このタイプのシステムでは、アルゴリズムのコンフィギュレーションは固定されており、デバイス コンフィギュレーション中に読み込まれます。Vivado HLS コンパイラで生成されたモジュールは、データ転送用に外部 FPGA ピンに接続され、トランザクションを受信します。スタンドアロンの計算システムの検証は、最も早期に実行される検証で、次の段階に分けられます。

- モジュール検証
- コネクティビティ検証
- アプリケーション検証
- デバイス検証

モジュール検証

Vivado HLS で生成されたブロックのモジュール検証は、第 7 章「ソフトウェア検証および Vivado HLS」に詳しく説明されています。ブロックがソフトウェア シミュレーションと協調シミュレーションの両方で機能的に正しく動作することが完全に検証されたら、このブロックのインシステム エラー耐性をテストする必要があります。

ソフトウェア シミュレーションおよび協調シミュレーションではどちらも、アルゴリズムが機能的に正しく動作することが個別にテストされます。つまり、すべての入力および出力が理想的に処理された場合に、アルゴリズムおよびコンパイル済みのモジュール正しく機能することが確認されます。このレベルで詳細にテストしておくと、データがモジュールに供給された後に正しく機能するようになります。また、この段階でモジュールの内部プロセッシング コアがエラーの原因となる可能性を取り除くことにより、後の段階で実行する検証の負担を軽減できます。このモジュール レベルの検証で唯一テストされないのは、インターフェイスでの間違ったハンドシェイクからモジュールが完全回復できるかどうかの検証です。

インシステム テストでは、入力および出力ポートが間違ってトグルされたときに Vivado HLS で生成されたモジュールがどのように動作するかがテストされます。このテストの目的は、テスト中のモジュールをクラッシュさせるなど、悪影響を与える可能性のある I/O の動作を取り除くことです。この手法でテストされる不適切なユース ケースは次のとおりです。

- 不安定なクロック信号のトグル
- リセット操作およびランダムなリセット パルス
- さまざまなレートでデータを受信する入力ポート
- さまざまなレートでサンプリングされる出力ポート
- インターフェイス プロトコル違反

上記のユース ケースはシステム レベルの動作例ですが、これらをテストすることにより、Vivado HLS で生成されたモジュールがあらゆる状況下で正しく機能することが検証されます。この段階で必要なテスト量は、使用するインターフェイスの種類やシステムの統合手法によって異なります。Vivado HLS のデフォルト設定を使用して、AXI に準拠したインターフェイスを生成すると、間違ったシステム レベルの動作のテストベンチを記述するのを回避できます。AXI 準拠のインターフェイスは、Vivado HLS コンパイラの開発者により完全にテストおよび検証されています。

コネクティビティ検証

コネクティビティ検証とは、アプリケーションに含まれるモジュールが正しく接続されていることをチェックするための一連のテストのことを指します。モジュール検証と同じように、必要なテスト量はシステムの統合手法によって異なります。第 8 章「複数のプログラムの統合」で説明したように、アプリケーションは、手動または FPGA デザイン ツールを使用して統合できます。

FPGA デザイン ツールを使用する場合は、ザイリンクス System Generator および Vivado IP インテグレーターの両方のデザイン フローを使用できます。これらのグラフィカル ツールは、モジュール接続に関連するすべての側面が扱われます。どちらのツールでも、フローの一部として、アプリケーションの各モジュールのポート タイプおよびプロトコル準拠がチェックされます。各モジュールでモジュール検証が実行した場合は、どちらのフローでも、ユーザーによるコネクティビティ テストを追加で実行する必要はありません。

手動の統合フローを使用する場合は、ユーザーが RTL でアプリケーションの最上位モジュールを記述し、アプリケーションを構成する各モジュールの RTL ポートをすべて接続する必要があります。これはエラーが最も発生しやすいフローなので、検証が必要になります。ただし、Vivado HLS コンパイラのデフォルト設定を使用して、すべてのモジュール ポートに対し AXI インターフェイスを生成すれば、必要なテスト量は削減できます。

AXI インターフェイスを中心に構築したシステムの場合は、コネクティビティはバス ファンクション モデル (BFM) を使用して検証できます。BFM は、ザイリンクスで既に検証されている AXI バスおよびポイント ツー ポイントの動作を提供します。これらのモデルはトラフィック ジェネレーターに使用可能で、RTL シミュレーションの一部として Vivado HLS で生成されたモジュールが正しく接続されていることを確認するのに役立ちます。

重要: このシミュレーションの目的は、接続と、システム内のデータフローが正しいことをチェックすることのみです。コネクティビティ検証では、アプリケーションが機能的に正しく動作するかどうかは検証されません。

アプリケーション検証

アプリケーション検証は、FPGA デバイス上でアプリケーションを実行する前の最終段階です。ここまでの段階では、アプリケーションを構成する個々のアルゴリズムの質をチェックし、すべてのものが正しく接続されているかを確認することに焦点を置いていました。アプリケーション検証では、元のソフトウェア モデルが FPGA インプリメンテーションと一致しているかどうかを確認することに焦点が置かれます。アプリケーションが Vivado HLS で生成されたモジュール 1 つだけで構成されている場合は、この段階はモジュール検証と同じです。アプリケーションが Vivado HLS で生成された複数のモジュールで構成されている場合は、この検証プロセスは元のソフトウェア モデルから始まります。

RTL シミュレーションで使用するアプリケーションの入力および出力テスト ベクターをソフトウェア モデルから抽出する必要があります。ハードウェア インプリメンテーションの構築は複数の段階で検証されるので、アプリケーション検証は徹底的なシミュレーションにする必要はありません。このシミュレーションでは、FPGA インプリメンテーションに確信を持てるように必要なだけテスト ベクターを実行できます。

デバイス検証

自動または手動の統合フローを使用し、アプリケーションを RTL にアセンブルした後、FPGA をプログラムするのに必要なバイナリまたはビットストリームを生成するため、さらにコンパイルを実行する必要があります。FPGA デザインの用語では、RTL をビットストリームにコンパイルする作業はロジック合成、インプリメンテーション、ビットストリーム生成と呼ばれます。ビットストリームを生成したら、FPGA デバイスをプログラムできます。ハードウェアが設計者が指定した時間正しく動作すれば、アプリケーションは検証されたことになります。

プロセッサ ベースのシステム

モジュールおよびコネクティビティ検証段階では、プロセッサ ベースのシステムの検証フローはスタンドアロン システムのものと同じです。主な違いは、アプリケーションの一部がプロセッサで実行されることです。Zynq®-7000 SoC では、アプリケーションの一部はエンベデッド Arm® Cortex™-A9 プロセッサ上で実行され、それ以外の部分は Vivado HLS で FPGA ファブリックにコンパイルされます。この分割により検証が難しくなりますが、これは次の手法を使用して解決できます。

- Hardware in the Loop (HIL) 検証
- 仮想プラットフォーム (VP) 検証

Hardware in the Loop (HIL) 検証

HIL 検証とは、テスト中のシステムの一部のシミュレーションを FPGA ファブリックで実行する検証手法です。Zynq-7000 SoC では、プロセッサをターゲットとするアプリケーション コードは、デバイスの Arm Cortex-A9 プロセッサ上で実行されます。Vivado HLS でコンパイルされたコードは、RTL シミュレーションで実行されます。

図 9-1 に Zynq-7000 デバイスの HIL 検証の概要を示します。この図のシステムは実験的な設定になっており、現在コマーシャル ボードとして提供されている ZC702 評価ボードおよび Vivado シミュレータが含まれています。またこの図は、プロセッシング システム (PS) およびプログラマブル ロジック (PL) ユニットの概念も示しています。PS はデュアル Arm Cortex-A9 プロセッサを指しており、プロセッシング サブシステムとも呼ばれます。PL は Zynq-7000 デバイス内の FPGA ロジックを指し、ここに Vivado HLS で生成されたモジュールがマップされます。

ヒント: HIL 検証には、プロセッサにアクセスするためのボードが必要で、この手法はどの Zynq-7000 SoC ボードでも機能します。

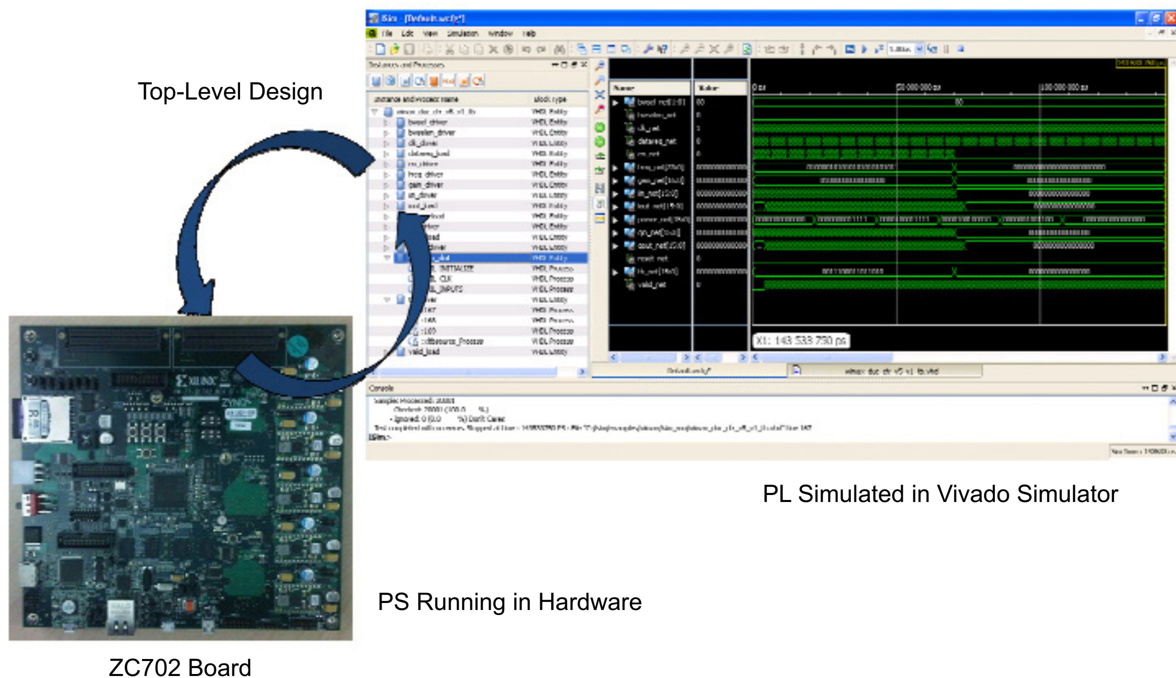

図 9-1: HIL 検証の概要 (Zynq-7000 SoC)

HIL 検証の主な利点は次のとおりです。

- プロセッサ モデルと実際のプロセッサとの間でシミュレーションに矛盾がない
- プロセッサ上で実行するコードが FPGA デバイスのスピードで実行される
- 生成された各モジュールが RTL シミュレーションでどのように動作するかを完全に確認可能

HIL 検証を使用する場合は、この手法のパフォーマンス特性を念頭に置くことが重要です。プロセッサ コードは実際のハードウェアで実行しますが、FPGA ファブリックは設計者のワークステーションでシミュレーションされます。第 8 章「複数のプログラムの統合」で説明したように、RTL シミュレーションは相対的に時間のかかるプロセスです。そのため HIL 検証では、アプリケーションのあらゆるユース ケースを検証するのではなく、プロセッサと FPGA ファブリックの間の通信を検証するだけにとどめてください。HIL 検証を使用してチェックすべき主なアプリケーション動作は、次のとおりです。

- Vivado HLS ドライバーのプロセッサ コードへの統合
- PS から PL へのコンフィギュレーション パラメーターの書き込み
- PL から PS への割り込み

ソフトウェア アルゴリズムの RTL インプリメンテーションだけでなく、プロセッサと生成されたハードウェア モジュールとが通信するのに必要なソフトウェア ドライバーも Vivado HLS コンパイラにより生成されます。このドライバーにより、アクセラレータの開始および停止、コンフィギュレーション、データ転送が処理されます。このドライバーは、Linux およびスタンドアロンのソフトウェア アプリケーション用のものがあります。

注記: スタンドアロンのソフトウェア アプリケーションとは、プロセッサが 1 つのプログラムのみを実行し、OS サポートを必要としないシステムのことです。

仮想プラットフォーム (VP) 検証

仮想プラットフォームは、ソフトウェア開発とハードウェア開発をオーバーラップさせるための確立された手法で、Zynq-7000 SoC で使用できます。仮想プラットフォームとは、アプリケーションとそれを実行するハードウェア プラットフォームの両方のソフトウェア シミュレーションを指します。デザインの PL 部分に使用されるモデルは、C、C++、SystemC、または RTL で記述可能です。このシミュレーション プラットフォームは、ハードウェア インプリメンテーションへの忠実性が異なるほかの推奨検証段階の代用として使用できます。

仮想プラットフォームの最速なユース ケースでは、PL をターゲットとするアプリケーション モジュールが、Vivado HLS コンパイラに提供されている C/C++ のソース コードを使用してシミュレーションされます。この設定では機能的に正しいシミュレーションが可能なので、アルゴリズムで正しい計算が実行されているかどうかをテストできます。モジュールは Vivado HLS で最適化されてコンパイルされるので、生成された RTL でモジュールのソフトウェア バージョンを置き換え、コネクティビティ テストとタイミング ドライバー シミュレーションをイネーブルにすることができます。

重要: RTL モジュールを追加すると、仮想プラットフォームでのランタイムに影響し、実行速度が遅くなることに注意してください。

デバイス検証

デバイス検証では、プロセッサと FPGA ファブリックが通信するアプリケーションのユース ケースすべてをチェックします。スタンドアロン実行の場合と同様に、このプロセスでは、完全なアプリケーションをある一定期間 Zynq-7000 SoC 上で実行する必要があります。このテストでは、デザイン内の PS と PL の間の通信に関するアプリケーションのコーナー ケースをすべてチェックすることが目的です。

付録 A

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、ザイリンクス サポート サイトを参照してください。

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、デザイン 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。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hubs View] タブをクリックします。
- ザイリンクス ウェブサイトで デザイン ハブ ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの Documentation Navigator ページを参照してください。

注意: DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

注記: 日本語版のバージョンは、英語版より古い場合があります。

1. Vivado® Design Suite ユーザー ガイド: 高位合成 (UG902)
2. 『Vivado Design Suite: AXI リファレンス ガイド』(UG1037: 英語版、日本語版)
3. Vivado Design Suite ユーザー ガイド: IP を使用した設計 (UG896)
4. Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計 (UG994)
5. Vivado Design Suite の資料 (japan.xilinx.com/support/index.html/content/xilinx/en/supportNav/design_tools/vivado_design_suite.html)

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、https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。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。https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品責任の制限を規定する適用法令および規則にのみ従うものとします。

© Copyright 2013-2019 Xilinx, Inc. Xilinx、Xilinx のロゴ、Artix、ISE、Kintex、Spartan、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他各国のザイリンクス社の商標です。すべてのその他の商標は、それぞれの保有者に帰属します。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。