LabVIEW ハイパフォーマンス FPGA 開発ガイド

LabVIEW RIO アプリケーションを最大活用するためのテクニック集

監修

日本ナショナルインスツルメンツ株式会社

西原　慶

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS

バージョン

この資料は英語版の　**Revision No. 1.1 – February 2014** 版の日本語訳資料です。

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS

目次

NATIONAL INSTRUMENTS

1. はじめに

FPGA（Field-programmable gate array）技術は、専用カスタムハードウェアの性能と信頼性を実現します。LabVIEW FPGA を使用すると、デスクトップシステムやリアルタイムシステムをプログラムする場合と同じ開発環境で、FPGA 技術を利用することができます。

高性能の LabVIEW FPGA アプリケーションを使用することで、タイミングや FPGA リソースなど、NI の再構成可能 I/O（RIO）デバイスの能力を最大限に引き出すことができます。本ガイドでは、高性能アプリケーションの開発に役立てていただけるよう、LabVIEW FPGA の最適化の概念やテクニックなど一般的な概要について解説します。

1.1. 対象とする読者

LabVIEW または LabVIEW FPGA Module をすでに使用されている方は、本書により産業に特化した上級概念を習得にできます。そうすることで、高スループット、高精度タイミング制御、FPGA リソースの効率化など、より厳しいアプリケーションの要件に対応することができます。

「高性能」という語は相対的な概念ですが、一般に下位レベルの詳細部を示すものとして使用しています。設計の性能をさらに高める必要がある場合に LabVIEW FPGA のプログラミングを難しくするのは、まさにその部分です。高性能アプリケーションにおける注意事項については、**FPGA ベースの高性能設計**を参照してください。

NI RIO ハードウェアプラットフォームは、下位レベル実装の様々な部分に対処していますので、アプリケーション特有の難題解決に専念することができます。デジタル設計の経験があり、VHDL、Verilog、または EDA（Electronic Design Automation）ツールに詳しい方は、本書を利用することで、LabVIEW FPGA でも同様なことができ、さらに NI RIO ハードウェアプラットフォームとの緊密な統合によって生産性が向上することがご理解いただけます。

1.2. 前提条件と参考資料

本ガイドは、LabVIEW プログラミングと LabVIEW FPGA の基本タスク、例えば FPGA VI の作成、コンパイル、シミュレーション、デプロイ等に熟知している読者を想定しています。これらの事項については、製品に付属のドキュメントを参照してください。

さらに、FPGA とはどのようなものか、またご自身のアプリケーションで使用するメリットについても基本的な理解が必要です。FPGA ベースの設計に関する基本情報につきましては、**ni.com** にあるホワイトペーパー、FPGA の内部を参照してください。

本ガイドは、LabVIEW FPGA アプリケーションを設計し最適化するのに必要な情報をまとめたものです。LabVIEW の特定のバージョンを対象としていません。API と環境に特化した情報については、バージョンによって異なる場合があるため、ご使用の LabVIEW バージョンに付属の製品マニュアルを参照してください。また本ガイドでは、必要に応じ付属のサンプルやオンラインチュートリアル、その他の資料についても参照しています。

本書は、CompactRIO 制御/監視アプリケーションの開発に役立つベストプラクティスを集めた NI CompactRIO 開発ガイド[2]と同様の資料です。LabVIEW FPGA プログラミングの基本情報につきましては、NI CompactRIO 開発ガイド[2]の「LabVIEW FPGA でハードウェアをカスタマイズ」のセクションを参照してください。本書は、そのセクションをベースに、さらに性能を向上させることを目的とした内容となっています。

関連情報

[1] FPGA の基本
http://www.ni.com/white-paper/6983/ja/

[2] NI CompactRIO 開発ガイド
http://www.ni.com/compactriodevguide/ja/

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS™

2. FPGA ベースの高性能設計

2.1. FPGA の利点

FPGA を利用したプラットフォームは、並列処理の性能が高くカスタマイズ可能であるため、上級の処理・制御タスクをハードウェアの速度で実行することができます。

CPU や GPU に比べると FPGA は比較的低速のクロックですが、そのクロックの差は、1 つのクロックサイクル内で複数、順次、並列の処理が可能な専用回路によって補います。FPGA の超並列プログラミング機能と NI RIO デバイスの優れた I/O 統合性を組み合わせることで、高速スループット、確定性の向上、応答時間の短縮が実現できるため、高速転送、デジタル信号処理（DSP）、制御、デジタルプロトコルなどのアプリケーションに最適です。

2.2. 高性能の LabVIEW FPGA

LabVIEW FPGA で標準の LabVIEW プログラミングテクニックを使用すれば、FPGA ベースアプローチのメリットが直ちに実感できます。上級アプリケーションでは、スループット、タイミング、リソース、数値精度のいずれかの特性について、さらにシステムの性能向上が必要となる場合もあります。

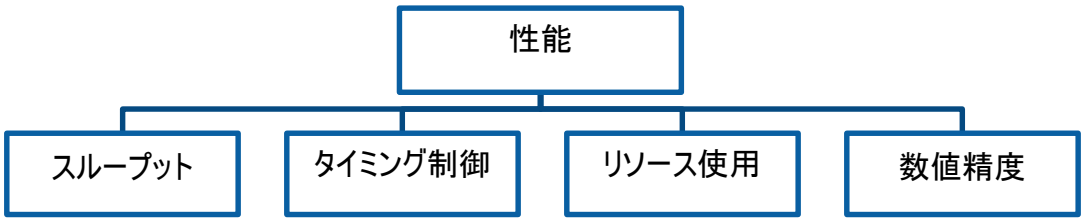

図 1. 高性能 RIO アプリケーションの各特性は互いに関連しています。例えばスループットを向上させると、タイミングとリソース使用に影響があります。

これらの特性は、多くの場合相互に関連しています。これらの特性のいずれか 1 つに集中して設計を改善しようとすると、他の特性にも影響することが少なくありません。よい影響の場合もありますが、多くは悪い方に影響します。例えば、スループットだけにとらわれていると、他のタイミング要件が満たされなくなる可能性があります。そのため各特性と互いの関連性について理解しておく必要があります。このセクションでは、様々な検討事項に関する基本定義について解説します。本ガイドでは、全編を通して関連するテクニックについてさらに詳しく説明します。

スループット

スループットは、DSP とデータ処理アプリケーションで鍵となる特性です。通信、製造、医療、航空宇宙/防衛、科学計測システムにおける技術の進歩は、短時間で処理しなければならない大量のデータを生み出しました。スループットは、単位時間あたりの仕事で計測します。多くの LabVIEW RIO アプリケーションでは、仕事とはサンプルの処理または転送のことをいいます。スループットは通常毎秒のサンプル数で表しますが、同様に毎秒のバイト、ピクセル、画像、フレーム、操作の数で表すこともあります。高速フーリエ変換（FFT）は、処理関数の一例で、毎秒の FFT、フレーム、またはサンプル数で表します。

本ガイドの**スループット最適化テクニック** で、クロックレートやアルゴリズムの並列化特性など、スループットに影響する要素について詳しく説明するほか、LabVIEW FPGA アプリケーションを作成する際にスループットを向上させるテクニックについても紹介します。

タイミング制御

タイミング制御とは、システム内で関心対象のイベント間の時間を指定したり計測したりする機能のことをいいます。正確なタイミング制御を行うことは、デジタルプロトコルアプリケーションや高速制御アプリケーションには重要です。システムの通信機能や制御対象のシステムの安定性に影響を及ぼすからです。多くの制御アプリケーションで、確実な応答時間が求められます。応答時間とは、制御対象システムのサンプリングからコントローラの出力が更新されるまでの時間のことです。これは I/O 遅延時間ともいいます。デジタルプロトコルアプリケーションでは、タイミング仕様というと、データまたは伝達信号に関連するイベント間の最短、最長時間を示すことがあります。LabVIEW FPGA アプリケーションを作成する場合、設計はハードウェア回路に反映されますので、高速タイミング応答でジッタの少ない設計を作成することができます。

遅延時間に関する詳しい説明や、LabVIEW FPGA を使用してアプリケーションを開発する際に、より高速かつ高精度のタイミング応答を実現するためのテクニックについては、本書のタイミングの最適化テクニックを参照してください。

FPGA リソース使用

FPGA のハードウェアリソース数は有限で、多くの場合プロセッサやマイクロコントローラよりストレージが制約されています。ご自身の設計が FPGA に適しているかどうかを見極めることは、開発プロセスにおいて重要です。また FPGA は、ロジック、信号処理、メモリブロックなど異なるタイプのリソースから構成されており、いずれか 1 つが足りなくなることで設計全体がコンパイルできなくなる可能性があります。

さらに重要なのは、リソース使用によって、スループットやタイミングの性能が著しく影響を受けかねないという点です。FPGA に搭載されている各種リソースの説明、FPGA に適した設計、性能を向上させる方法については、本書のリソースの最適化テクニックを参照してください。

数値精度

数値精度とは、アプリケーションが正しく動作するのに必要な桁数またはビット数があるかどうかを示すものです。精度が不十分な場合、な相互作用するアルゴリズムではわずかな数値誤差が積み重なって、全く異なる結果が出ることがあります。整数に使用するビット数や、固定小数点数の整数部と分数部、浮動小数点数のダイナミックレンジなど、システムの変数を表すビット数が FPGA アプリケーションの性能やリソース使用に影響する可能性があります。

適切な数値精度を特定するための具体的なテクニックについては、本書では説明していませんが、LabVIEW FPGA 設計の最適化を行うにあたってよく生じる問題であり、最適化の他要素について学習する際にも考慮が必要となります。

3. NI RIO ハードウェアプラットフォームを理解する

お客様のアプリケーションの要件には、前の章で紹介した項目が全て含まれていることもあるでしょう。それらの項目は相互に関連していますので、プログラミング時に適切なトレードオフを選択する為に各項目の相互作用について正しく理解することが重要です。本ガイドの以降のセクションでは、それぞれの項目について詳しく説明します。本セクションでは、NI RIO プラットフォームの様々な形を解説し、特定のアプリケーションに適したハードウェア特性について説明します。

3.1. PXI/PC 用 NI RIO

PXI プラットフォーム

PXI（PCI eXtensions for Instrumentation）とは、高性能アプリケーション向けに開発された堅牢なモジュール式計測プラットフォームです。PCI/PCI Express バス技術に特殊な同期バスを統合することで、製造テスト、航空宇宙/防衛、機械監視、自動車、工業用テストといったアプリケーション向けに、低コストながら高性能の実装プラットフォームを実現しています。PXI は、PXI Systems Alliance（PXISA）が管理するオープン工業規格です。PXI Systems Alliance は、PXI 規格の推進のほか、相互運用性の確保や PXI 仕様の管理に取り組む団体です。

図 2. PXI プラットフォームは、モジュール式計測、高性能処理、データ転送、同期バスを統合したデスクトップ/リアルタイムアプリケーション開発プラットフォームで、高度な FPGA アプリケーションのソリューション開発に利用できます。

PXI Express は、PCI Express バスを活用したポイントツーポイントバストポロジを採用していますので、最大 4 GB/秒[1] のスループットで各デバイスがバスにそれぞれ直接アクセスできます。内部での同期クロックやトリガのルーティングには、内蔵のタイミング/同期ラインが使用されます。PXI シャーシは、専用の 10 MHz システム基準クロック、PXI トリガバス、スタートリガバス、スロット間ローカルバスを搭載していますが、一方 PXI Express シャーシは、さらに 100 MHz の差動システムクロック、差動信号、差動スタートリガを搭載しているため、高度なタイミング/同期にも対応できるようになっています。

NI では、PXI プラットフォームで使用可能な複数の RIO 製品をご用意していますので、帯域幅や同期機能を高性能アプリケーションに活用することができます。

[1] 本書では GB = 10^9 バイト、MB = 10^6 バイト、MiB = 2^{20} バイト、GiB = 2^{30} バイト

NI FlexRIO

FlexRIO は、PXI プラットフォームがベースになった研究、開発、テスト、試作、実装用汎用デバイスのシリーズです。FlexRIO デバイスは、LabVIEW FPGA Module でプログラミングできる大容量 FPGA と、高性能なアナログ/デジタル I/O を搭載したアダプタモジュールからなっています。アダプタモジュールは交換可能で、それにより LabVIEW FPGA プログラミング環境で使える I/O が決まります。

図 3. FlexRIO デバイスは、アダプタモジュールに直接接続できる大容量 FPGA を搭載していますので、様々な I/O オプションが利用できます。

FlexRIO FPGA Module は、Xilinx Virtex-5/Kintex-7 FPGA とオンボードダイナミック RAM（DRAM）を搭載し、FPGA への I/O を備えた FlexRIO アダプタモジュールに接続することができます。アダプタモジュールのインタフェースは、132 ラインの汎用デジタル I/O が FPGA ピンに直接接続されています。その他にも、電源、クロック、インタフェースの定義に必要な補助回路を搭載しています。その 132 ラインは、シングルエンド操作の場合最大 400 M ビット/秒、差動操作の場合最大 1 G ビット/秒、最大 I/O 帯域幅は 66 G ビット/秒（8.25 GB/秒）に構成することができます。

FlexRIO FPGA Module 間で、1.5 GB/秒のレートでデータを直接転送できます。最大 16 ラインのストリームをサポートしていますので、ホストの CPU リソースに負荷をかけずに、複数の FPGA を使用した複雑な通信スキームをシンプルにできます。FlexRIO デバイス間でデータを直接転送する方法につきましては、本ガイドのデータ転送メカニズムを参照してください。

NI R シリーズマルチファンクション RIO

標準の NI マルチファンクション DAQ ボードは、異なるサンプリングレートで様々な信号を計測・生成できますが、R シリーズマルチファンクション RIO デバイスはさらに一歩進んで、以前は DAQ でできるとは思わなかったようなタスクが FPGA で可能となりました。

NI の R シリーズマルチファンクション RIO デバイスは、FPGA 技術とアナログ入出力とデジタル I/O ラインを 1 つのデバイスに統合することで、価格と性能を最適なバランスで実現しています。NI R シリーズマルチファンクション RIO デバイスは、PCI、PCI Express、PXI、USB バスをサポートしています。ケースにパッケージされた製品とボードオンリーのオプションがあります。

図 4. NI R シリーズマルチファンクションデバイスは、汎用のマルチファンクション DAQ に、LabVIEW でプログラムできる FPGA が追加されたものです。

NI R シリーズマルチファンクション RIO デバイスは、チャンネルごとに専用のアナログ/デジタルコンバータ(ADC)を装備しているため、独立したタイミング/トリガ機能と最大 1 MS/秒のサンプリングレートが実現します。マルチレートサンプリング、個々のチャンネルトリガといった専用機能があり、これらは通常の DAQ ハードウェアの機能にはないものです。

ソフトウェア設計型モジュール式計測器

NI は、初のソフトウェア設計型計測器、NI PXIe-5644R ベクトル信号トランシーバ(VST)を発売しました。これは、NI-RFSG/NI-RFSA ドライバを使用して、RF ベクトル信号生成(VSG)および収集(VSA)に使用できるデバイスです。

図 5.ベクトル信号トランシーバ(VST)は、LabVIEW でカスタマイズ可能な大容量 FPGA を搭載した 3 スロットの PXI フォームファクタで、計測器クラスのベクトル信号生成および収集を統合したものです。

RF ハードウェアが小型で高性能なのに加え、VST は LabVIEW FPGA Module で FPGA をカスタマイズできるという点で画期的です。つまり制約となるのはベンダが定義する計測器自体ではなく、アプリケーションの要件のみということです。それにより柔軟性が格段に向上し、さらに FPGA ベースの処理と制御によってアプリケーションのニーズへの対応能力も高まっています。

3.2. NI RIO を使用したコンパクトな組込アプリケーション

CompactRIO

CompactRIO は、3 つのコンポーネントからなる堅牢で再構成可能な組込システムです。そのコンポーネントとは、リアルタイムオペレーティングシステム（RTOS）を搭載したプロセッサ、再構成可能 FPGA、そして交換可能な工業用 I/O モジュールです。

図 6. CompactRIO はモジュール式 I/O を備えた堅牢でコンパクトな実装プラットフォームで、高性能の分散型組込アプリケーションに適しています。

CompactRIO システムは、組込コントローラと再構成可能シャーシから構成されます。組込コントローラは、確定性と信頼性に優れた動作が求められる LabVIEW Real-Time アプリケーションをスタンドアロンで確実に実行します。また、浮動小数点の数式処理と解析にも優れています。

CompactRIO システムの中核をなすのは、再構成可能 I/O FPGA コアを持つ組込シャーシです。FPGA が得意とするのは、高速ロジックと高精度タイミングを必要とするタスクです。また FPGA は、多彩な高性能 I/O 機能を持つ I/O モジュールに直接接続されています。

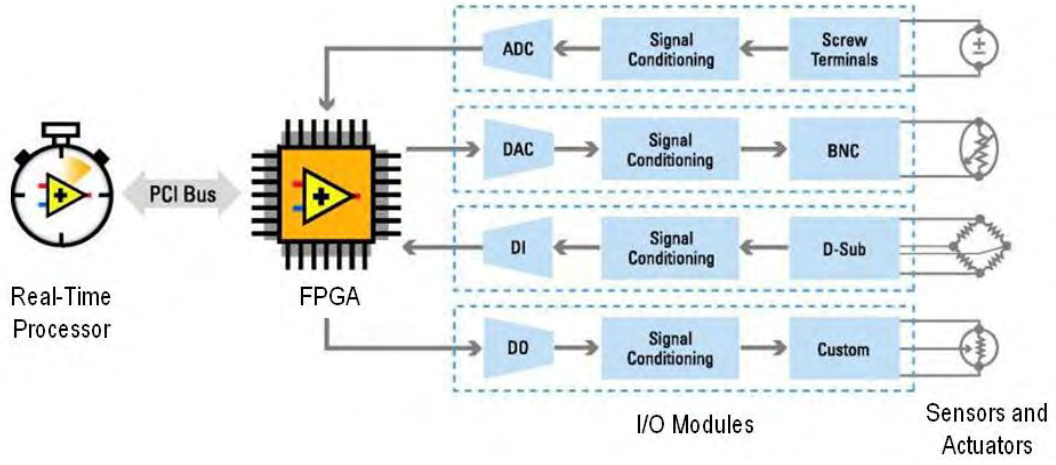

図 7.CompactRIO は LabVIEW RIO アーキテクチャのよい例です。リアルタイム OS を搭載した専用プロセッサ、ハードウェア機能を再構成できる

FPGA、そして広範なモジュール式 I/O オプションがあります。

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS™

Single-Board RIO

Single-Board RIO 製品は、高性能および優れた信頼性を必要とする大量オーダーおよび OEM 用の制御/データ収集アプリケーション向けに設計された組込ボードです。オープンな組込アーキテクチャ、コンパクトサイズ、柔軟性を特長とする商用（COTS）ハードウェアを使用すると、カスタム組込システムを短期間で市場に投入することができます。

図 8. Single-Board RIO なら、組込 RIO プラットフォームが小型のフォームファクタや実装アプリケーションでも利用できます。

3.3. FPGA プラットフォームを選択する

どのプラットフォームと NI RIO デバイスファミリを使用するかは、アプリケーションの要件によって決まります。ただし各デバイスファミリの中でも、FPGA のサイズや速度など、様々なオプションがあります。LabVIEW FPGA アプリケーションでは、ハードウェアのあるなしに拘らず、任意の NI RIO デバイス用に設計をコンパイルできます。また設計はソフトウェアでシミュレーションして、コンパイル前に動作を確認することも可能です。ただし実世界の I/O 信号を使用してハードウェア速度でシステムを検証する場合は、ハードウェアも必要です。NI RIO デバイスファミリの中でも最大容量の FPGA デバイスを試作用に選び、後で FPGA リソースの性能に合わせてアプリケーションを最適化して、適切なサイズのデバイスに代えるユーザも少なくありません。

FPGA 上で基本的なタイミングや、トリガ、同期しか行わないアプリケーションなら、小容量の FPGA でかまわないでしょう。制御、デジタルフィルタ、複雑なアナログトリガなど、さらなる信号処理が必要なアプリケーションには、大容量の FPGA をお勧めします。

関連情報

[1]	PXI とは http://www.ni.com/pxi/whatis/
[2]	FlexRIO とは http://www.ni.com/flexrio/whatis/
[3]	R シリーズマルチファンクション RIO http://www.ni.com/rseries/
[4]	CompactRIO とは？ http://www.ni.com/compactrio/whatis/
[5]	Single-Board RIO とは？ http://www.ni.com/singleboard/whatis/

[6]	USB 対応 NI R シリーズ http://sine.ni.com/nips/cds/view/p/lang/en/nid/212013
[7]	適切な FPGA ハードウェアを選択 http://www.ni.com/fpga-hardware/choose-hardware/

4. シングルサイクルタイミングループを使用した高性能プログラミング

LabVIEW での高性能 FPGA プログラミングに関連する概念の多くでは、シングルサイクルタイミングループ（SCTL）を有効に活用することが求められます。SCTL とは、リソースの使用を軽減し、高スループットと高精度タイミング制御を可能にする LabVIEW FPGA の主要ストラクチャです。SCTL のプログラミングパラダイムは、FPGA 回路の動作により近く、LabVIEW コードを FPGA に実装する際に高い自由度が得られます。

4.1. SCTL vs 標準の LabVIEW FPGA コード

SCTL について理解するため、標準の LabVIEW FPGA コードを While ループまたは For ループ内に配置した時のコンパイル動作についてまず確認します。

LabVIEW FPGA Module を使ってプログラムすると、ダイアグラムの内容がハードウェアに変換されますので、一般にダイアグラム内の各ノードは、回路の構成要素としての FPGA と同じものを表します。SCTL の外にコードを配置すると、そのハードウェア構成要素の中をデータが流れる際に、LabVIEW はその実行を制御します。その結果、全ての入力に有効なデータが入った時のみ要素が実行するよう、追加の回路が必要となります。このような実行モデルは、構造化データフローと呼ばれます。

構造化データフローは、従来のプログラム実行モデルに従っています。関数は全ての入力パラメータにデータが入るまで実行せず、関数が返されるまで呼び出し側はブロックされます。CPU でコードを実行する場合、CPU には固定の汎用回路があり、コードとデータに逐次的に対応します。ただし、CPU と異なり FPGA の方にあるコードは専門性の高い並列回路となり、データは電気信号の形で流れます。

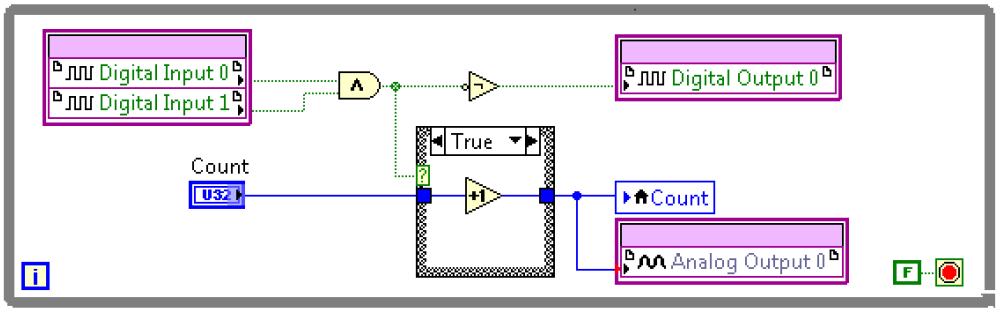

図 9. 標準の LabVIEW FPGA コードは、データ依存関係がある場合に逐次的に実行する回路に組み込まれます。

コードが While ループ内でコンパイルした場合、LabVIEW では関数から関数へとハードウェアレジスタ（小型ストレージ要素）がクロックデータに挿入されますので、LabVIEW の構造化データフロー機能が強化されます。

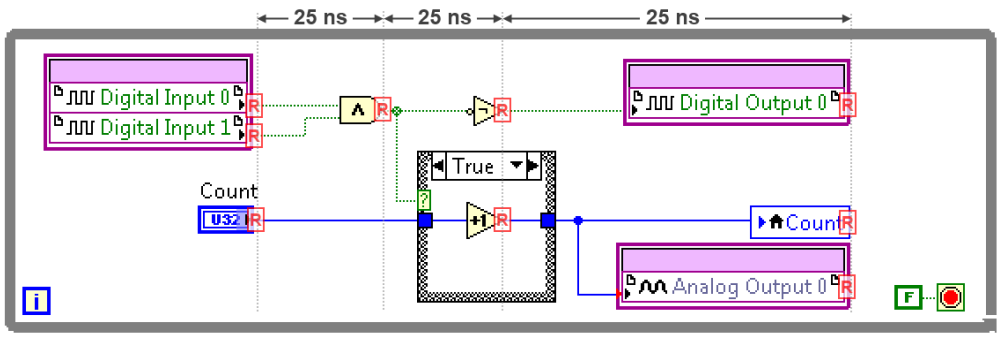

図 10. 標準の LabVIEW FPGA コードは、各関数のレジスタを使用して、通常 40 MHz（25ns 周期）の最上位クロックの各サイクルにデータをラッチします。この図で四角に R と書かれているのがハードウェアレジスタです。

それらのレジスタ要素は、データがノードからノードへと流れる際に、関数の実行を制御し、データを各クロックサイクルにラッチします。各ノードは、1 サイクルまたは複数のサイクルを実行できます。LabVIEW FPGA アプリケーションでは、SCTL の外に配置されたコードは、コンパイルによって、または LabVIEW FPGA Module のバージョンによって、タイミング動作が異なります。

構造化データフローを維持するということは、データ依存関係を持つノードチェーンで、ノードの一部のみが一定時間アクティブに実行していて、回路の他の部分はデータを待機しているということです。

このような逐次的実行は専用 FPGA 回路の使用方法としては非効率のように見えますが、多くのアプリケーションでは十分です。構造化データフローモデルを採用することで、デスクトップでの作業に似た LabVIEW コードを作成できますので、実装の詳細について気にすることなく、FPGA 上でコードを実行することができます。その結果、優れた性能と確定性、そして I/O 統合性が実現できます。

4.2. SCTL について理解する

SCTL の機能と中に配置されたコードを理解することは、優れた LabVIEW FPGA アプリケーションを作成する上で重要です。SCTL は、LabVIEW FPGA アプリケーションに固有のストラクチャです。SCTL ストラクチャと中に配置されたコードは、LabVIEW タイミングループと内部のコードと見た目は非常に似ています。しかし SCTL 内のコードは動作が大きく異なり、必ず特定の FPGA クロックの 1 クロックサイクル内で実行するようになっています。SCTL の主要コンセプトは以下の 5 つです。

1. ストラクチャ
ストラクチャは、LabVIEW では特有の意味を持ちます。データが境界を流れる際に、データの範囲と有限的な転移点を提供するものです。SCTL は構造体で、ダイアグラム内の他のストラクチャに関して構造化データフローモデルに従います。具体的にいうと、SCTL は全ての入力、あるいはトンネル/シフトレジスタを通る全てのワイヤがデータを受け取らない限り、実行を開始できないということです。同様に、SCTL は中のコードの実行が終了しなければ、トンネルを通って出力を生成することはできません。

2. ループ
SCTL は、コードの実行方法を決めるストラクチャであるだけでなく、プログラムループでもあります。
内部コードを繰り返し実行し、While ループの規則に従って、少なくとも 1 回実行します。停止条件は実行時にいつでも変更できます。

3. クロック

SCTL には必ずクロックがあります。クロックは、SCTL の内容から作成された回路の動作に適用される周波数を決定します。SCTL 内の回路は全て同じクロックを使用するため、クロック領域とも呼ばれます。

SCTL クロックを定義できる機能は、複数のクロックを持つ設計において基本となるメカニズムです。様々な SCTL が、複数のクロック領域を定義する異なるクロックを使用することがあります。クロックのレートとソースはコンパイル時に決める必要があるため、実行が始まってから変更することはできません。

4. 最大反復遅延時間

SCTL は、コードの実行に使用されるクロックを指定するだけでなく、中に含まれる全てのコードを 1 クロックサイクル内で実行する必要があります。

5. 異なる実行パラダイム

SCTL の実行パラダイムは、LabVIEW の標準の実行とは異なり、中に含まれる全てのコードが 1 クロックサイクル内で実行します。反復遅延要件を満たすため、LabVIEW FPGA のコンパイル時に構造化データフローの実行に使用されるフロー制御回路が削除されます。

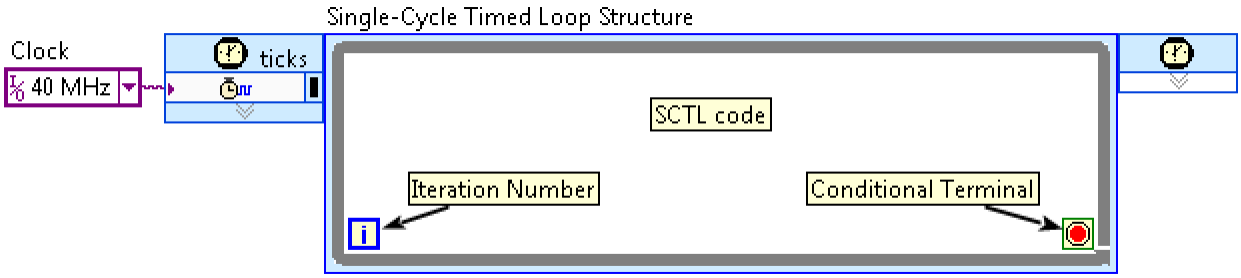

図 11. SCTL は、コードに使用するクロックを指定し、最大反復遅延が 1 クロックサイクルのループストラクチャです。

フロー制御回路を取り除くことで、FPGA リソースの使用量が減らせるだけではありません。最も重要なのは、ダイアグラムが並列回路のように動作し、データが電気信号として SCTL 中を無制限に流れることができるようになることです。信号はロジックによって変換され、I/O、ストレージ要素、制御器/表示器など、ダイアグラム上の特定点にラッチされます。

信号はダイアグラム内のそうした特定のポイントにラッチされているため、次のクロックエッジが到達する（次の反復の開始点）前に確定値に整定する必要があります。このラッチ動作にループのルーピングモデルが組み合わされることで、FPGA コンパイルのツールチェーンによって統合された同期回路が作成されます。

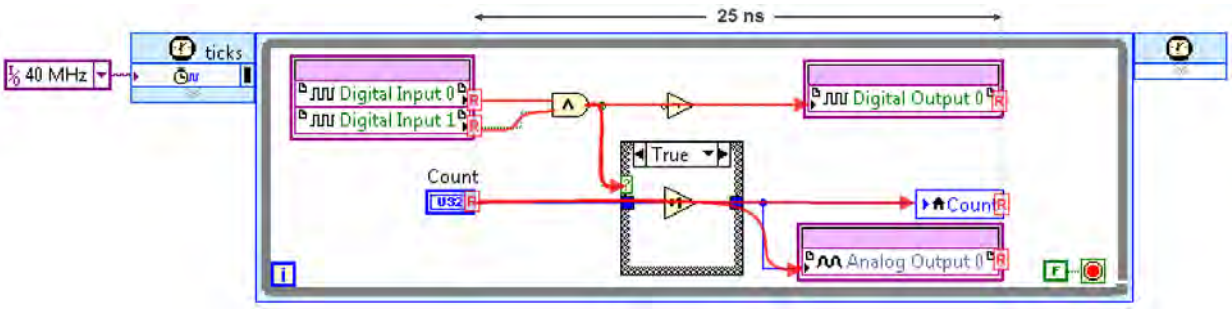

図 12. SCTL は、信号が内部を流れ、I/O、制御器/表示器、その他のノードなど、特定点でのみラッチされる同期回路を表すものです。SCTL は、その内容を動作させるのに使用するクロックを指定し、信号はレジスタ間を 1 クロックサイクル（この例では 25 ns）内で移動します。

同期回路の「同期」という語は、標準のプログラミング言語で関数コールメカニズムのことをいう場合とは意味が異なります。同期回路とは、単にクロックによって稼動する回路のことです。同期関数コールは、関数が結果を返すまで、呼び出し側の実行がブロックされます。

4.3. SCTL の利点

SCTL で指定されている 1 サイクルの反復遅延は、コンパイル時に生成される VHDL コードとともに、Xilinx コンパイルツールチェーンにまで受け継がれます。Xilinx コンパイラは SCTL コードを統合し、1 サイクル要件を回路の制約として処理します。コンパイルが正しく行われると、作成された設計は確実に全ての信号を 1 クロックサイクル内で実行し整定します。この動作は、CPU 上の LabVIEW タイミングループとは異なります。LabVIEW タイミングループは、実行時間は保証されておらず、実行時に検証する必要があります。

反復遅延の制約は、SCTL パラダイムの中で重要な部分です。コードの一部の実行に 1 サイクル以上かかることも少なくない標準の LabVIEW FPGA コードと異なる部分だからです。このタイミング保証があるため、イベント間のタイミングや全体の実行レートを指定するメカニズムとして、SCTL を使用することができます。

- SCTL の同じ反復内に含まれる 2 つのイベント間の最大時間は、1 ループ期間内に制限されます。これは 2 つのイベント間の最大遅延を指定する必要がある場合に便利です。
- 連続する反復のイベント間で最小サイクル数を指定、制限、保証することができます。デジタル通信プロトコルなどのアプリケーションでイベントの計測や制御を行う場合に便利です。
- ループレートがコンパイラによって保証されていることがわかれば、スループットを指定することができます。ループと、反復ごとに処理されるサンプル数が、スループットの目安となります。

4.4. SCTL の制約

SCTL コードは見た目こそ他の LabVIEW コードと似ていますが、プログラミング/実行パラダイムでは、これまでとは考え方を変え、異なる概念で性能を向上させる必要があります。

SCTL 内でサポートされる関数とストラクチャ

SCTL には反復遅延要件があるため、SCTL の内部に配置できる LabVIEW の関数とストラクチャには制限があります。

また SCTL 内での実行に複数サイクルが必要な関数の場合、新しいデータが利用可能になった時に、他のロジックに通知するためのハンドシェイク信号がさらに必要になります。これらの関数については、ハンドシェイクメカニズムとともに**高スループット IP を統合する**で解説します。複数サイクルで実行する関数で、ハンドシェイク信号がないものは、SCTL 内では使用できません。例えば以下のような関数です。

- 商/余りなど、一部の演算関数
- ループタイマ/待機関数
- 一部のアナログ入力/アナログ出力 I/O ノード
- 再入不可サブ VI の複数のインスタンス（再入負荷 VI の使用による影響についてはタイミング最適化テクニックを参照）
- 浮動小数点データタイプを扱うほとんどの関数

For ループや While ループなどのループストラクチャも、SCTL 内では使用できません。何らかの反復アルゴリズムを必要とするサブ VI を SCTL 内で使用する予定なら、難しい問題が生じます。そのような場合は、SCTL 内で使用するサブ VI を明示的に作成して、フィードバックノードを使用して全反復の値を保存し追跡する必要があります。

図 13 のコードは、固定サイズの配列を適切な場所にソートする基本的な方法を示します。これを見ると、ACTL で使用するためにコードを適合させる必要があることがわかります。

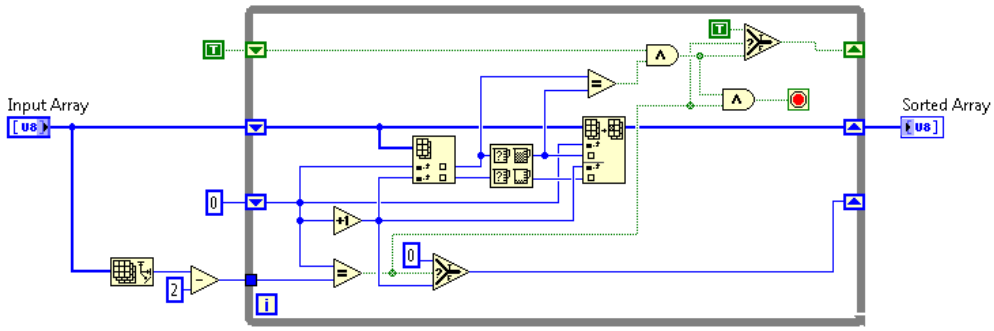

図 13. 標準の LabVIEW FPGA コードで作成した反復アルゴリズムの例。配列を適切な場所にソートすることができます。

SCTL バージョンは、図 14 に示すように、レジスタが内部に移動し回路が追加されて、配列の走査時に指標を追跡します。

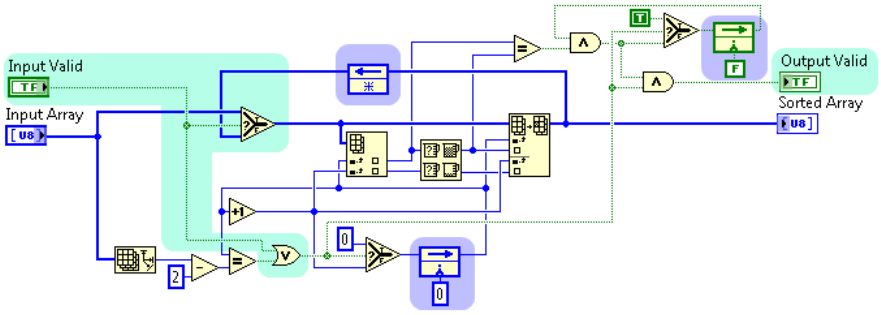

図 14. SCTL バージョンの配列ソートアルゴリズムが 1 つの配列入力で複数回呼び出されることで、シフトレジスタがフィードバックノード（青い部分）に置き換わり、配列走査時に指標を追跡するコードが追加され、ハンドシェイク信号（緑の部分）が追加されます。それにより、新しいデータの到着がわかり、ソートされた配列出力を受け取れるタイミングを後続のブロックに伝えることができます。

図 14 では、サブ VI の出力はアルゴリズムが完了するまで利用できませんが、サブ VI が完了するには複数回呼び出す必要があることがわかります。出力に信号を追加して、配列がソートされたタイミングを示す必要があります。配列ソートは、実行に複数サイクルを要し、ハンドシェイク信号が必要な関数の一例です。これについては**高スループット IP を統合する**で説明します。

ネストループを含む、反復計算用のループを使用する関数は、しばしばアルゴリズム VI と呼ばれます。それらの VI を LabVIEW FPGA を使用して移植し、最適化するには少々困難が伴います。NI では、そのようなニーズに応えるため、LabVIEW FPGA IP Builder という LabVIEW FPGA アドオンを開発しました。大きな配列やネストループ、浮動小数点タイプを処理する際に使用して、SCTL 内での使用に最適な IP を生成することができます。

SCTL ストラクチャ間でのデータ転送

異なる SCTL クロック間でデータをやり取りするのは、極めて難しい場合があります。データ転送メカニズムで詳しく説明しますが、異なるレートで動作したり、未知あるいは可変の位相関係で動作するループは、データを出力したりラッチする時間も異なります。そのため、ブロックメモリ FIFO（first-in first-out）構造など、安全な転送メカニズムを採用していなかったり、安全でないメカニズムで独自に同期を追加していないと、データの損失、重複、破損が生じる可能性があります。

失敗しないコンパイル

アプリケーションによっては、非常に高速のスループットと低遅延が必要なものもあります。そのようなニーズを満たすには、SCTL のクロックレートを高めるのが一般的な方法です。クロックレートとリソース使用量が増えると、ツールチェーンは厳しい遅延時間の制約に従おうとするため、コンパイル時間が長くなり、コンパイルが失敗に終わる可能性が高まります。

信号が同期回路で整定するまでにかかる時間は、伝搬遅延とも呼ばれます。伝播遅延には、論理遅延と経路遅延の 2 つの要素があります。

経路遅延とは、信号が回路のコンポーネント間を移動するのにかかる時間のことです。それらの信号は光の速度に近い高速で移動しますが、クロックレートもそれに対応できる速さのため、遅延を計算に入れる必要があります。トレースと回路も理想的な導体ではないので、わずかな負荷によってデジタル信号から特性的なパルス形状が失われ、クロックレートの上

昇に伴って長いトレースにおけるインテグリティが保証されにくくなります。

論理遅延とは、信号が論理コンポーネント中を伝搬するのにかかる時間、例えば比較関数の入力から出力に到達するまでの時間などのことをいいます。

ダイアグラムが大きくなると、論理が追加されて関連するコンポーネントの距離が離れ、さらに伝搬遅延が大きくなることがあります。

SCTL の中で最も伝搬遅延が長いデータ経路のことをクリティカルパスといいます。クリティカルパスの伝搬遅延が SCTL のクロック周期より長いと、コンパイルはエラーになります。LabVIEW FPGA には、コンパイル結果をダイアグラムのコンポーネントに戻す仕組みがありますので、エラーになった経路の最適化を試みることもできます。

最適化テクニックは、アプリケーションの目的によって異なります。高性能の LabVIEW FPGA アプリケーションの開発で考慮すべき最も一般的な要素は、スループット、遅延、リソース使用量です。それらの要素を決める際はトレードオフを考慮する必要があります。例えば、スループットを上げると、リソース使用量が増えて遅延が長くなるという具合です。また、アルゴリズムで可能な場合はコードを並列化することで遅延を低減することができますが、それは FPGA のリソース使用量の増加につながります。そのようなトレードオフに対処するには、反復的な最適化作業が必要となります。以降の章では、スループット、遅延、FPGA リソース使用を最適化する様々なアプローチについて解説します。

関連情報

[1] LabVIEW FPGA IP Builder を使用して VI を FPGA 向けに最適化し移植する
http://www.ni.com/white-paper/14036/en/

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS™

This page intentionally left blank.

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS™

5. スループット最適化テクニック

スループットは、信号、画像、データ処理アプリケーションにおいては一般的な要件です。FPGA は、一般にデータストリームのインライン処理に使用されますので、必要なスループットレートを場合によっては無限に維持する必要があります。スループットレートは、サンプル数または時間あたりのデータ量で計測しますが、本章では、スループットは毎秒処理または転送されるサンプル数で計測したものとします。

スループットは、以下の 3 つの要素によって決まります。

1. 使用しているクロックのレート（サイクル数/時間）
2. IP が呼び出しごとに受け入れるサンプル数（サンプル数/呼び出し）
3. アルゴリズムがもう一度呼び出せるようになるまでに必要なサイクル数（サイクル数/呼び出し）。これは開始間隔（II: Initiation Interval）または起動周期とも呼ばれます。

これらの要素を組み合わせて、以下の定義によりサンプル数/時間が算出されます。

$$スループット = (クロックレート \times 呼び出し毎のサンプル数) / 開始間隔$$

この方程式から、以下をすることでスループットを向上できることがわかります。

- クロックレートを増やす
- 各呼び出しで処理されるサンプル数を増やす
- 開始間隔を減らす

5.1. クロックレートを増やす

コンポーネントのスループットを高める方法としては、トップレベルダイアグラムクロックまたは SCTL クロックのクロックレートを上げるのが最もダイレクトな方法です。正しく行うと、他の要素は全て同じという仮定で、クロックレートの向上に比例してスループットも向上します。そのため、可能な場合はまずクロックレートの向上を検討することをお勧めします。

ただしクロックレートを上げると、他にも影響がある点にも注意する必要があります。これまでの章で説明したように、クロックレートを上げることで、コンパイルプロセスが難しくなり時間もかかるようになるため、対象のプラットフォームで求められるタイミングの制約要件を満たす回路を作成するのは、最終的に不可能になってしまいます。本章のクリティカルパスの短縮セクションで、クリティカルパスを短縮することでレートを向上させるヒントを紹介しています。

ただし設計によってはクロックレートを上げられないものもあります。設計の部位によっては、IP が実行する際システムの他の部分との特定のクロック関係を必要とするものもあります。例えば一部の I/O ブロックは、I/O サブシステムと同じクロック領域で実行する必要があります。コンポーネントレベル IP（CLIP）統合メカニズムを採用する外部 DRAM インタフェースは、DRAM インタフェースのクロック領域からのアクセスが必要な場合もあります。そのため設計を複数の SCTL に分割し、それぞれに適したクロック領域を使用する必要があります。

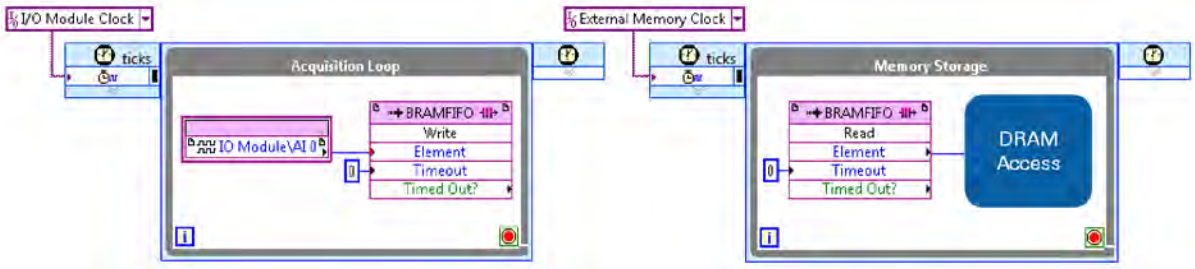

図 15. I/O とやり取りしたり CLIP を介して DRAM にアクセスするコードは、適切なクロック領域に常駐している必要があります。アプリケーションが両タスクを実行する場合は、複数の SCTL に分割することが必要になります。

高クロックレートが必要なのは設計の一部のみということもあるでしょう。コードの一部が低速のクロックレートでも望ましいスループットを達成できるのであれば、設計を複数のループに分割することをお勧めします。そうすることでコンパイルプロセスが容易になり速度が向上します。

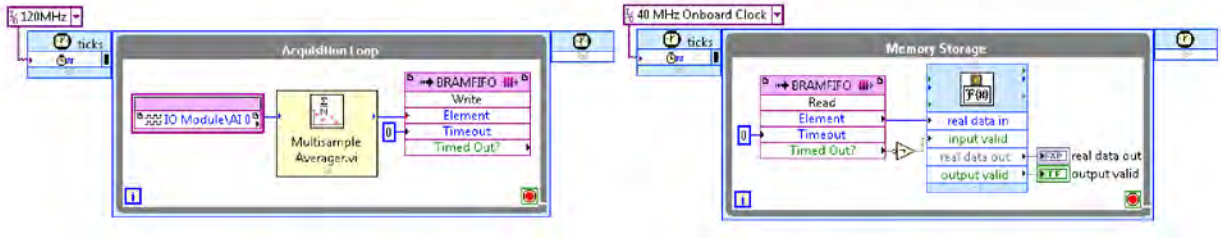

図 16. 設計は部分によってスループット要件が異なる場合があります。複数の SCTL に分割することで、高速レートが求められる部分で希望通りのクロックレートを実現することが可能となります。

異なるクロック領域で実行するループ間でデータをやり取りする際には、注意が必要です。詳細については、データ転送メカニズムのセクションを参照してください。

5.2. 各呼び出しで処理されるサンプル数を増やす

各 VI 呼び出し、つまり SCTL の各反復で処理されるデータサンプル数は、通常設計で決められた特性です。例えば、1 回の呼び出しで同じチャンネルの複数のサンプルを処理するアルゴリズムの並列バージョンを実装できますので、クロックレートと起動間隔は一定のままでスループットを高めることが可能です。

並列化が可能かどうかは、アルゴリズムの特性によって異なります。複数の I/O チャンネルに対し同じ操作を行う場合など、独立したデータセット間で明確な並列化が可能な操作もあります。FPGA は本来並列特性を持ち、SCTL 内ではループストラクチャを使えないという制約があることから、そのような場合は一般にロジックを複製して各チャンネルを個別に処理することになります。しかしそれをするとリソース使用が増えます。LabVIEW でチャンネルを別々に処理できるアプリケーションの例としては、各チャンネルの最大値を特定するアプリケーションなどがあります。

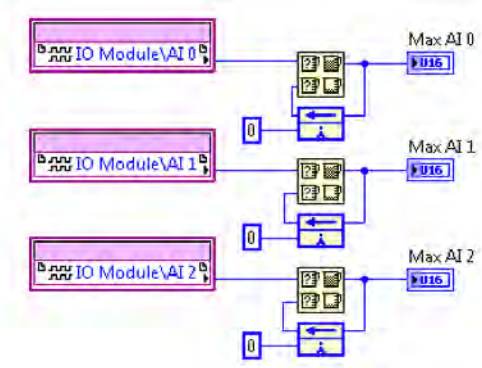

図 17. FPGA では、チャンネルごとにコードを複製することで、複数のチャンネルで LabVIEW が同時に動作することが可能です。

さらに興味深い例として、同じデータセットからのサンプルを別々に処理することもできます。例えば同じ I/O チャンネル（データストリーム）からの複数のサンプルを同時に処理する場合などです。それが可能なのは、前のチャンネルサンプルの情報が不要な操作のみです。そのような操作の例として、チャンネルサンプルの係数によるスケールがあります。

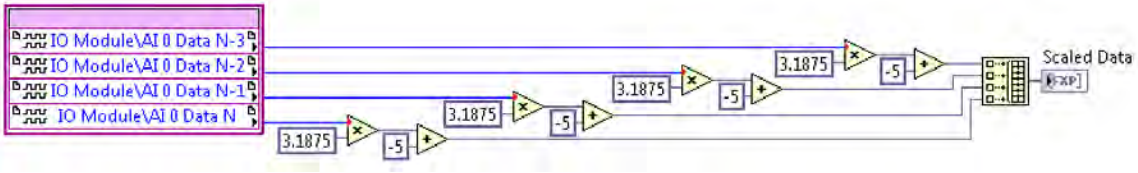

図 18. 一部のハードウェアモジュールは FPGA の 1 クロックサイクルにつき複数のサンプルを生成するため、それには並列アプローチが必要となります。

一部の FlexRIO アダプタモジュールは、FPGA の 1 サイクルにつき複数のチャンネルサンプルを生成するため、1 チャンネルの全サンプルを処理するにはそのような並列性が必要になります。そうした関数は、マルチサンプル/マルチサイクル IP と呼ばれます。

FPGA は本来並列性を備えたデバイスなので、複数のサンプルやデータセットの並列処理が、スループット向上の有効なオプションとなります。ただし FPGA もリソースは限られているので、並列化によってリソース使用量も比例して増えます。コピーを作成して明示的にコードを並列化すると、読み取りと管理の難しいコードになってしまいます。並列化、つまり 1 回の呼び出しまたはループの反復で複数のサンプルを処理することが適しているかどうかは、慎重に考慮する必要があります。その他の並列化テクニックについては、本書で後ほど説明します。

5.3. クリティカルパスの短縮

スループットを最適化する際、SCTL のクリティカルパスを短縮するのは、クロックレートを向上させるのが最終目標です。SCTL のクリティカルパスの伝搬遅延が SCTL のクロック周期より長いと、コンパイルはエラーになります。LabVIEW でコンパイルエラーが起こると、クリティカルパスがハイライトされます。その情報を使用して、経路全体の伝搬遅延を短縮し、同じかより速いクロックレートで設計を再コンパイルすることができます。

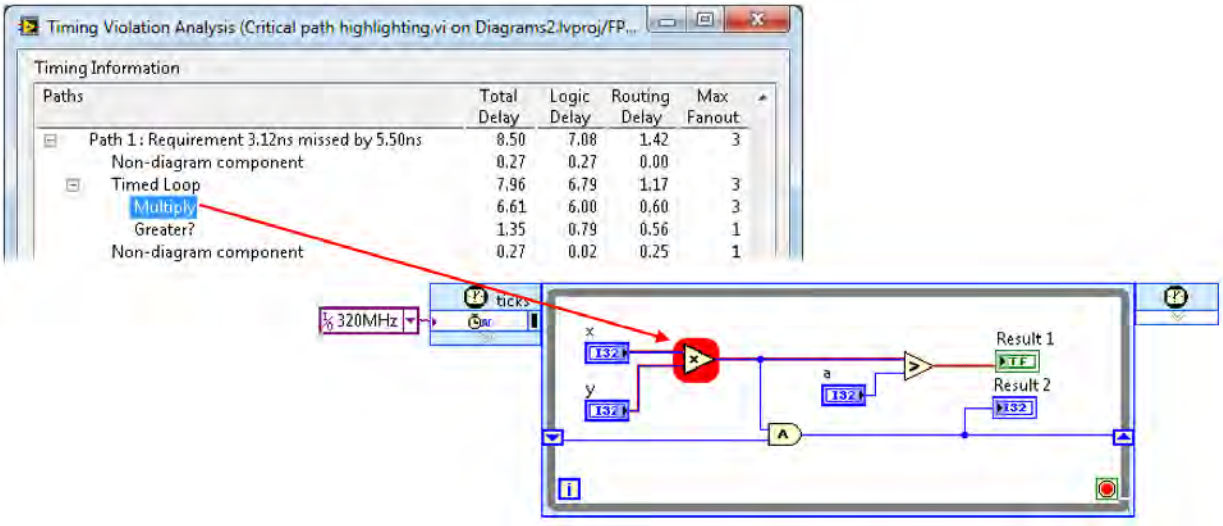

図 19. LabVIEW FPGA は FPGA コンパイル結果を解析し、タイミング制限に間に合わなかった経路をダイアグラム上で直接特定するので、それらを修正することができます。

操作を並列化する

クリティカルパスの長さを短縮するには、まず不要な依存性を取り除きます。データ依存性による逐次実行が必要な部分を見つけ、操作が本当にその順序で行われなければならないのか検討します。クリティカルパスから操作を削除することもできます。そうするには、ワイヤを分岐し、データの複数のコピーに対し異なる操作を同時に実行します。結果は図 20 に示すように後で結合することができます。

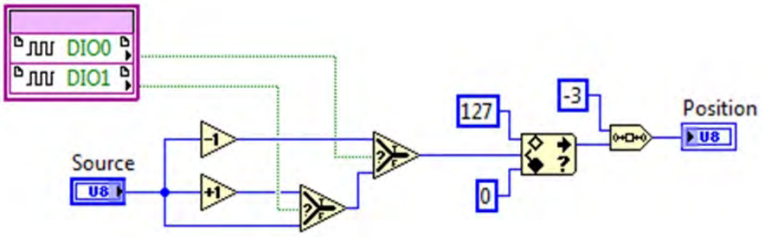

図 20. リソースが許す場合は、ダイアグラム上の独立した操作を明示的に並列化して、コードのハードウェア実装として FPGA を利用することができます。

このアプローチが適しているかどうかは、操作の特性によって異なります。タイミングの最適化テクニック で説明しますが、他のコーディングパターンを採用して並列性を高めることも可能です。そうすることでコード全体の遅延を低減することができます。

データタイプの最適化

データを表すビット数は、クリティカルパスにおける伝搬遅延に多大な影響をもたらします。そのため、精度/確度要件を満たす最小データタイプを使用することで、希望の SCTL クロックレートが実現しやすくなります。

FPGA では、より大きなデータタイプを処理するには多くのロジックが必要で、ビット数に比例して増えます。それらのビットを 1 つのロジックセクションから次のロジックセクションへと運ぶため、電気経路が追加されます。その経路の長さは様々です。信号が全経路で整定するためには、タイミングの制約を緩める必要があります。そうすることで回路がさらに高レートで実行するのを防ぐことができます。

データタイプを最適化することで、効率的にリソースの使用量を減らし、実現可能なクロックレートを高めて、その結果設計のスループットが向上します。データタイプの最適化については、リソースの最適化テクニックでさらに詳しく説明します。

設計のパイプライン処理

パイプライン処理とは、コードのスループットを上げる方法として最も一般的なテクニックです。SCTL のクリティカルパス上にレジスタを挿入して、コードをより短い同時セクションに分割するものです。短いコードセクションは実行に時間がかからないため、SCTL のクロックレートを向上させることができます。

以下の例では、データ経路は 1 つですが、これはクリティカルパスでもあります。SCTL を 200 MHz で実行したい場合、ダイアグラムは値を入力から出力へ伝搬し、5 ns 以内に整定できなくてはなりません。

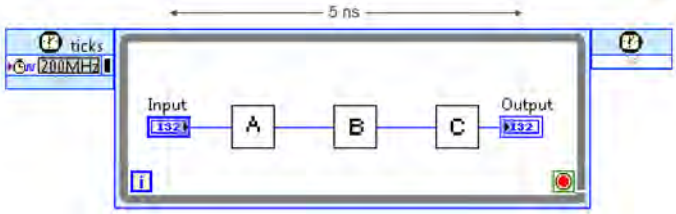

図 21. SCTL が 200 MHz クロックを使用するよう構成されている場合、このコードシーケンスは 5 ns 未満で実行する必要があります。

サブ VI A、B、C は FPGA 上で実行するのにそれぞれ 3、4、5 ns かかるとすると、このシーケンス全体の所要時間は 12 ns となり、実現可能な速度は 83 MHz にしかならないことになります。そのため 200 MHz のコンパイルは、このケースでは失敗です。

上記の VI を最適化するには、フィードバックノードという形で、通常はフォワード表記でレジスタを追加します。すると各サブ VI は、ループが実行を開始すると同時に、データ値の処理を開始することができます。フィードフォワードノードを使用すると、値をラッチし反復間で保持することで、実行を分割し同時実行することができます。それによってクリティカルパスは 3 つのセグメントに分割されます。クリティカルパスを短くすると、クロックレートが向上し、この例のように望みどおりの 200 MHz でクロックが実行できます。つまりスループットが 2.4 倍に向上したということです。

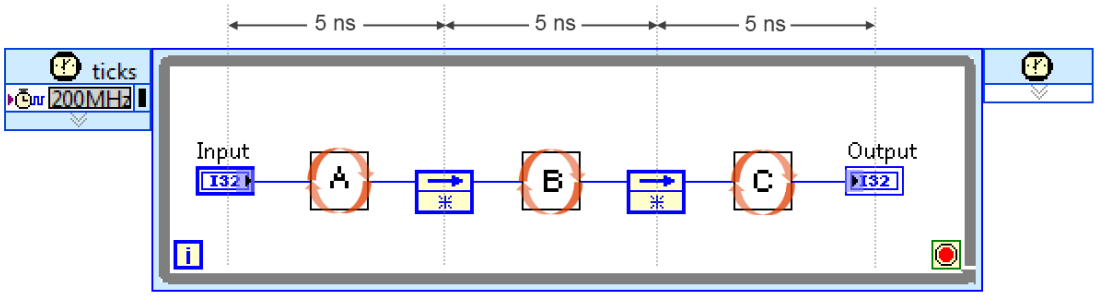

図 22. パイプライン処理によって VI を同時に実行できます。5 ns という遅延制限は経路のそれぞれのセグメントに適用されますので、200 MHz でのループ実行が可能になりました。

パイプライン処理を行うと、処理遅延が増える可能性があります。上記の例で、元のダイアグラムの 1 サイクルで入力から出力へ運ばれるサンプルは、3 サイクルかかることになります。目的の 1 つがクリティカルパスを分割してクロックレートを高めることなので、除かれた遅延の一部がクロックレートの高速化により復活しています。ただしロジックのさらなる遅延とクリティカルパスの不完全な分割により、高クロックレートで遅延時間を完全にオフセットすることはできません。現実には、パイプライン処理を行うと、高スループットを実現するためのトレードオフとして、高クロックレートで遅延時間が長くなります。

上記の例では、予測される最大クロックレート 83 MHz における元の遅延時間は以下のようになります。

$$\text{元の遅延時間} = \frac{1\ cycle}{83\ MHz} = 12\ us$$

パイプライン処理した設計の場合、200 MHz で 3 サイクルと仮定して、遅延時間が 25 パーセント増えています。

$$\text{パイプライン処理の遅延時間} = \frac{3\ cycle}{200\ MHz} = 15\ us$$

ストリーミングアプリケーションには、このスループットと遅延時間のトレードオフの影響を受けるものもあるので、注意すべき重要事項です。

図 23 に示すタイミングダイアグラムで、実行順を確認することができます。サンプルは、サブスクリプトの番号で表しています。サブ VI A が 2 つ目のサンプルの処理を始めた時、サブ VI B は前のサイクルのサブ VI A の出力を処理しています。

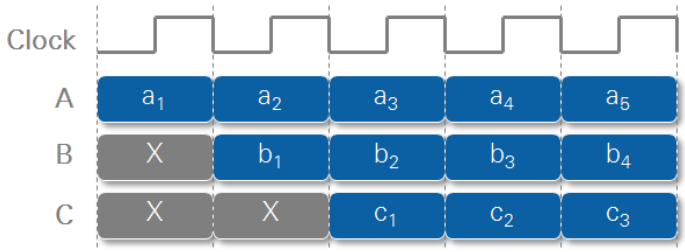

図 23. このタイミングダイアグラムは、同じ反復内で SCTL の異なる部分が異なるサンプルを処理していることを示しています。そのため高クロックレートが可能ですが、サンプルを入力から出力まで伝搬するのにより多くのサイクルが必要となるため、パイプライン処理が開始したらデータの有効性への対応が必要となります。

B は最初のサイクルでは未知の値を処理していることがわかりますが、生成されるデータは無効の可能性もあります。C もやはり最初の 2 サイクルで未知の値を処理しています。パイプライン処理を実行すると、ダイアグラム上のデータの有効性という問題が発生します。

パイプライン処理された設計では、パイプラインは有効な出力を生成する前に「埋める」、つまり「準備」する必要があります。このプロセスに複数のクロックサイクルが費やされます。また、下流のブロック、特に I/O、保存値のアップデート、通信といった副次的効果をもたらすものは、パイプラインが埋められる際に無効データについて通知を受け取る必要があります。

無効なデータを下流のブロックに通知するには、図 24 に示すように、データとともに運ばれるデータ有効信号をしようするのが最もシンプルな方法です。

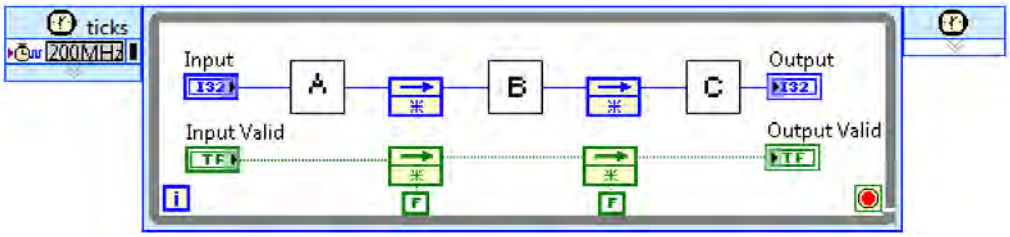

図 24. パイプライン処理された設計では、複数のサイクルで有効なサンプルを生成しますので、多くの場合出力が下流ブロックによって利用されるタイミングを示すブール信号をデータに付加することが必要になります。

データ経路が様々な場所で分岐・合流することで、ダイアグラムが煩雑になる場合があります。処理分岐をまたがってデータの一貫性が求められる設計の場合は、遅延のバランスを取ることで、適切なサイクル関係で関数がサンプルを処理できるようにする必要があります。図 25 は、コードの分岐間で遅延のバランスを取る例を示しています。ここではレジスタノードを 1 つ使用して、構成ダイアログで遅延値を 2 に設定しています。

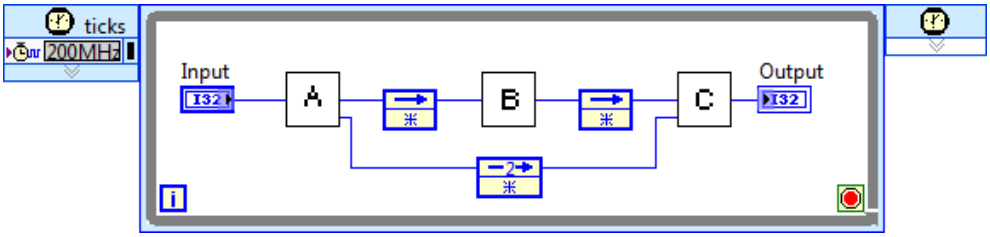

図 25. フィードバックノードは、データ処理をバッファして複数のサンプル分遅延させることができます。そうすることで、複数の処理分岐間でデータ遅延のバランスを取ることができます。

信号の有効性を示す信号も、データがダイアグラム中を流れる際に分岐したり結合したりしなければなりません。有効なデータの取り込みや生成に複数のサイクルを要するような関数を使用すると、フロー制御のコンセプトがさらに複雑になりますので、データの有効性をダイアグラム上で明示的に処理することが必要になります。4 線式ハンドシェイクや AXI プロトコルといったフロー制御テクニックについては、**高スループット IP を統合する**で解説します。

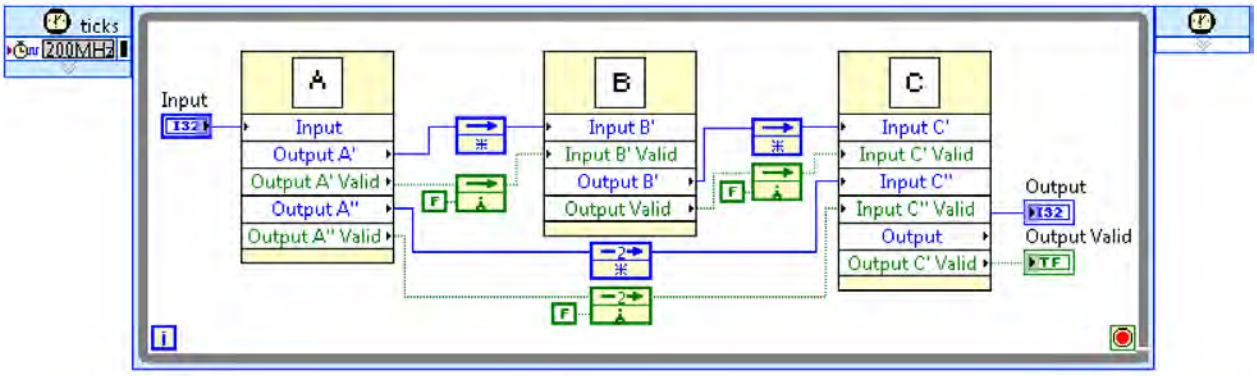

図 26. 各サブ VI が適切な遅延関係で確実に有効なデータを処理するよう、データ有効信号が分岐処理チェーンに追加され、出力においてデータの有効性を示します。

5.4. 開始間隔を短縮する

1 つの入力に複数のサイクルを要する IP（マルチサンプル/マルチサイクル IP）がダイアグラムに含まれている場合、そのような IP は初期化の間隔を狭めることでスループットを上げることができます。IP の開始間隔を狭めるということは、より頻繁にサンプルを受け取り、呼び出しと呼び出しの間のサイクルが少なくなるようにコードを変更するということです。

図 27 は、2 サイクルごとにサンプルを処理するコードのシーケンスを示しています。コードは 3 つの段階に分かれており、それぞれサブ VI A、B、C で表します。パイプラインレジスタはすでに追加されていますので、SCTL のクロックレートを上げることでスループットを高めることができます。

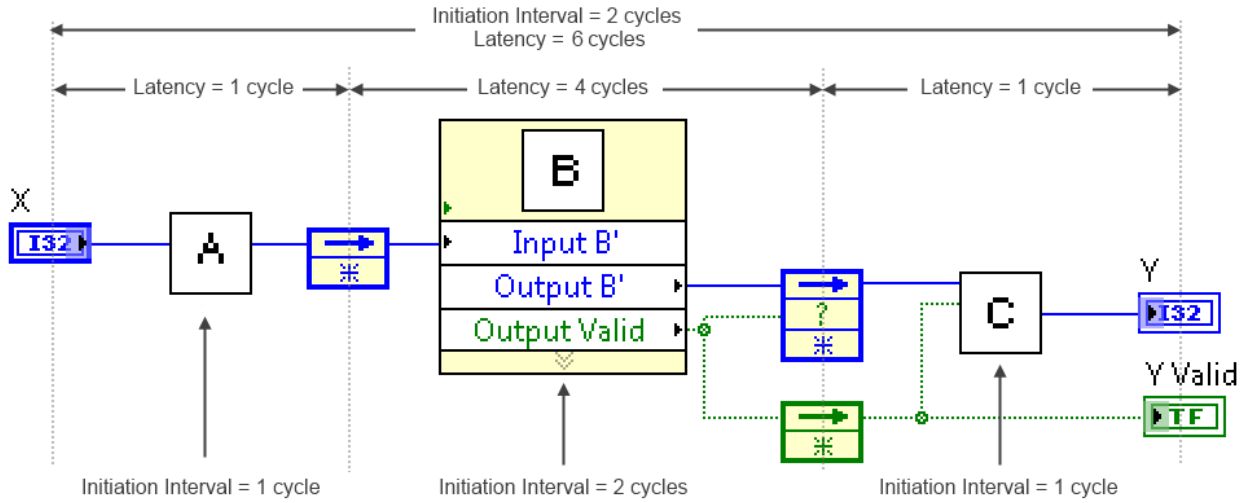

図 27. このコードはサブ VI B で 2 サイクル分の開始間隔となっています。コードパスが一度に 1 つの入力を処理する際、実現可能な最小開始間隔は、パス内の関数の最大開始間隔になります。コードパスの遅延時間は、パス内の個々の遅延の合計です。わかりやすくするため、このダイアグラムでは、パイプラインの開始時にデータの有効性について下流ブロックに知らせるためのコードは省略しています。

このコード例の重要な特徴は、サブ VI B が各入力で 2 サイクル必要な処理を実行すると、2 サイクル分の開始間隔になるという点です。サブ VI B および C の開始間隔が 1 サイクルでも、コードのパスシーケンス全体で実現可能な最小開始間隔は 2 サイクルで、チェーンの最高開始間隔と同じです。そこから以下のようなことがわかります。

1. このコードシーケンスは、X 端子から入ってくるサンプルを 1 つおきに無視しています。
   高スループット IP を統合するで説明したとおり、上流コードは、このことをすでに計算に入れているか、ハンドシェイク信号を使用して新しい入力を供給できるタイミングを知らされている必要があります。
2. サブ VI B の出力有効信号を伝搬して、下流のコードブロックに出力の有効性を知らせる必要があります。

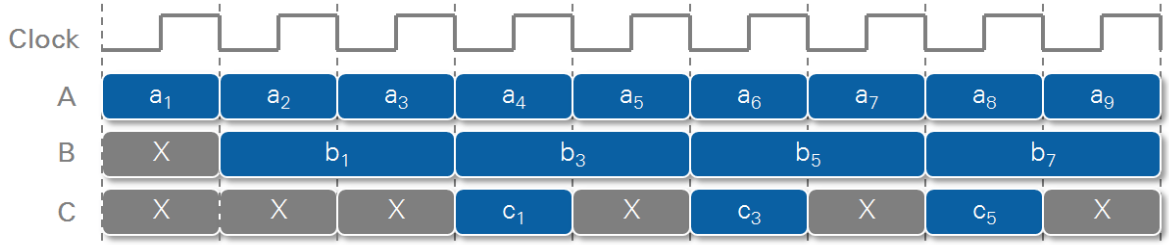

図 28. 遅延時間は、サンプルが入力から出力まで移動するのにかかるサイクル数として計測できますが、開始間隔は、IP が新しい入力を受け取れる頻度をサイクル数で表したもので、スループットと反比例します。上記の例のタイミングダイアグラムが示すコードは、遅延時間が 4 サイクル、開始間隔が 2 サイクルです。

このダイアグラムでは、遅延時間が開始間隔と異なることがわかります。この場合、サンプルは入力 X から出力 Y へ移動するのに 4 サイクルかかりますが、このダイアグラムは X の新しい入力を 1 つおきのサイクルでしか受け取ることができません。パイプライン処理された設計では、あるサンプルが入力から出力へ移動するのにかかる時間より少ないサイクル経過後にデータを受け取ります。これが遅延時間を計測する 1 つの方法です。

パイプライン処理されたプロセスのよい例が製造アセンブリラインです。製造アセンブリラインでは、複数の工程で製品の異なる部品を組み立て、特定の時間において様々な命令に基づき作業をします。パイプライン処理されたプロセスにステージを追加するということは、プロセスをよりシンプルなステージに分割することです。シンプルなステージは実行が早いため全体のスループットは向上しますが、作業者とアセンブリ機器は増やさなければなりません。

例えば 10 ステージの製造アセンブリラインは、毎分 1 デバイスのスループットで、1 分ごとに新しいデバイスの組み立てを開始するとします。個々のデバイスは 10 ステージを経過する必要がありますので、各ステージの時間が 1 分であるとすると、デバイス製造全体の遅延時間は 10 分になります。パイプラインステージは、小さなステージに分割できないこともあります。図 29 のサブ VI B のように、パイプラインをさらに分割できないとすると、実行を複製しオーバーラップさせることで、コードが 1 サイクルで処理できるサンプル数を増やすことができます。前の例からどのように変わったかにご注目ください。First Call 関数を使用して、サブ VI B の 2 つめのコピーの実行を 1 サイクル分遅らせているのがわかります。

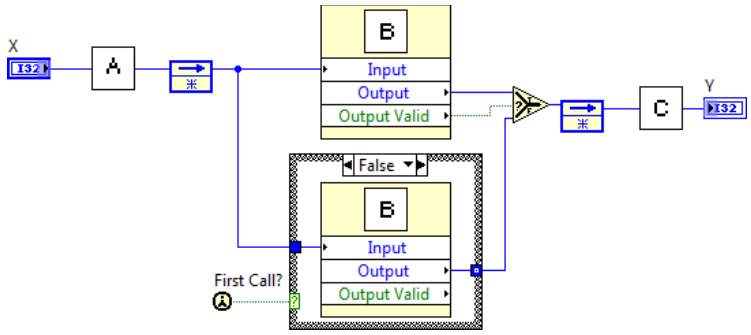

図 29. サブ VI B を複製すると、2 サンプルを同時に処理することで、1 サイクルあたり 1 サンプルの処理が可能となり、コード全体のスループットが倍になります。

変更されたダイアグラムでは、毎サイクルで 1 サンプルを受け取るようになりましたので、元のスループットが倍になり、その分リソースを多く使用しています。サブ VI B の処理は、各サンプル間で依存せず個別に実行できると仮定しています。状態を保持する操作やサンプルの順序が厳密に決まっている場合は、このように並列化することはできません。

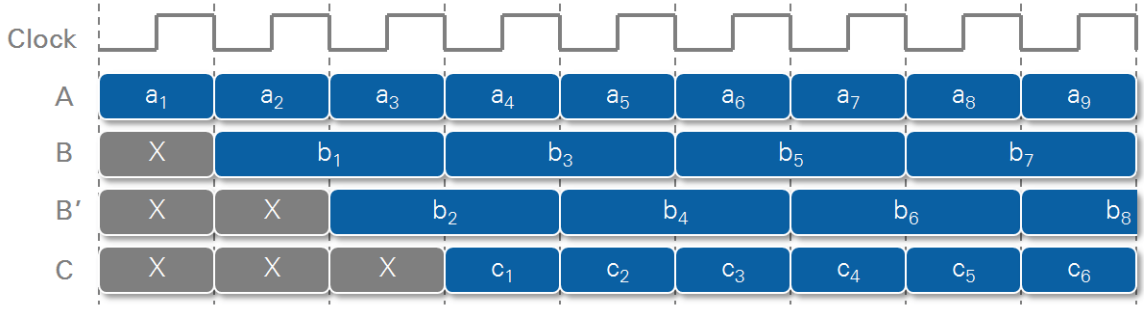

図 30. サブ VI B を複製し、ずらして実行することで、全体の開始間隔が 1 サイクルになります。コードは SCTL の各反復で新しい入力を受け取れるようになったため、このコード例のスループットが倍になります。

このテクニックでは、パイプライン処理と複製の両方を行います。コードと状態が複製され、複数の逐次サンプルで同時に処理されます。これは前の章で説明した並列化テクニックとは異なります。並列コードでは、1 回のループ反復で複数のサンプルを受け取るか、同じサンプルで別の操作を並列で実行します。上述の例では、1 回のループ反復ごとに 1 サンプルを受け取りますが、それぞれ進捗度の異なる 2 つの独立したサンプルを同時に処理します。

6. 高スループット IP を統合する

前の章では、独自の処理 IP を作成するための一般的な最適化パターンについて解説しました。独自の IP の作成、検証、管理にかかる時間を節約するため、できる限り既存の IP を使用することをお勧めします。本章では、LabVIEW FPGA IP の共通ソースについて説明し、アプリケーションへの統合方法を解説します。

6.1. 推奨の LabVIEW FPGA IP ソース

LabVIEW FPGA の IP には、複数のソースがあります。以下の順で検討することをお勧めします。

1. LabVIEW FPGA パレット

   LabVIEW FPGA パレットには、ナショナルインスツルメンツが NI RIO デバイスとの互換性の検証と最適化を行っている IP が含まれています。アプリケーションや市場に特化した様々なツールキットをパレットに追加することで、専門的な IP を構築する手間が省けます。

2. 内蔵の IP Generator

   IP Generator とは、一般的なアルゴリズムや関数から、特定の性能や機能要件に合わせた IP を生成するためのツールです。IP Generator のには、以下のようなものがあります。

   a. Xilinx CORE Generator system
   b. LabVIEW Digital Filter Design Toolkit その他のジェネレータ
   c. LabVIEW FPGA IP Builder モジュール

3. LabVIEW FPGA コミュニティ

   LabVIEW ユーザコミュニティでは、LabVIEW FPGA IP の共有や再利用ができる場が設けられています。コミュニティの例としては、LabVIEW ツールネットワーク、IPNet、NI FlexRIO/Software-Designed Instruments IP コミュニティなどがあります。

4. ハードウェア記述言語（HDL）IP

   LabVIEW FPGA には、VHDL や Verilog など、HDL をベースにした IP を統合するメカニズムがあります。

以降のセクションでは、上記のソースから作成した IP の例や、関連する IP 統合テクニックを紹介します。

6.2. LabVIEW FPGA 高スループット関数パレット

高スループット FPGA 関数パレットには、高性能アプリケーション用に特化して開発された IP が含まれています。そのパレットには SCTL 以外で使用できるノートも含まれていますが、ほとんどは SCTL 内で使用することを目的としたものです。高スループットパレットのほとんどの関数には構成ダイアログが付属していますので、動作やリソース使用、性能特性を調整することができます。

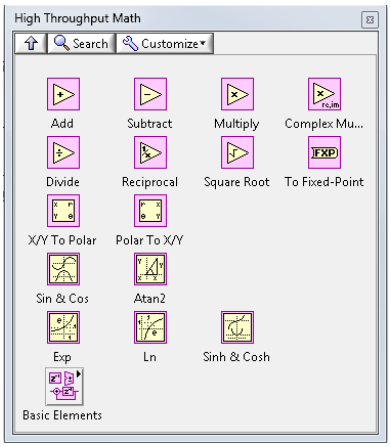

図 31. 高スループット数学パレットには、SCTL 内で使用する目的で設計された関数が含まれています。また、「離散遅延」や「DSP48」ノードなど、専用の FPGA リソースを表示するサブパレットもあります。

高スループット関数は、FPGA コンポーネントの専門性とリソースの局所性を利用することで、ロジックとルーティングリソースの使用量を減らしています。さらにパイプライン処理テクニックを使用して、高速クロックレートを実現しています。

それらの関数に関するドキュメントには、スループット、遅延、リソース、数値精度のトレードオフのバランスを取る構成方法が詳しく記載されています。構成の方法によって、性能と専用 FPGA リソースの活用方法にどのような影響があるかを示すノードの例を、以下にいくつか紹介します。

高スループット積関数

このノードは、2 つの固定小数点数の積を計算します。図 32 は、このノードの構成ダイアログを示しています。この構成ダイアログには、指定した構成の予測される性能と数値動作についてのフィードバックも表示されます。

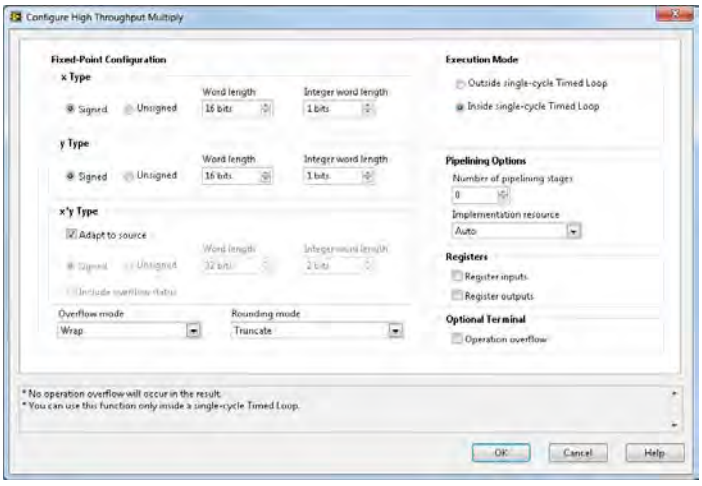

図 32. 高スループット積関数は、機能と性能のニーズに合わせて調整できる詳細な構成ダイアログがあります。

これらの設定は、ノードの性能に影響します。

固定小数点構成

リソース最適化テクニックで説明したように、ノードの性能とほとんどのロジック/演算操作は、オペランドと出力の幅に大きな影響を受けます。

オーバーフローモード

オーバーフロー状態は、出力端子のタイプが 2 つの入力の積を保持するのに十分な大きさがない場合に発生する可能性があります。オーバーフロー状態を検出してレポートし、オーバーフロー状態が発生した際に、過剰データが失われないように別の回路を追加する必要があります。

オーバーフロー状態の詳細と、各設定を使用すべき状況については、製品マニュアルを参照してください。一般に、折り返しモードは性能が高く、使用するリソースも低く抑えられます。これも数値精度と性能のトレードオフの一例と考えることができます。

丸め込みモード

丸め込みが行われるのは、値の予測精度が、その値を表すタイプの精度を上回る場合です。LabVIEW ではその値が強制変換されますので、精度が低下する可能性があります。

丸め込みを行うにはロジックを追加する必要がありますので、リソース使用量が増えて性能が制限されます。一般に、可能なら切り下げモードを使用するのが望ましいとされます。より精度の低い出力に対応するため最下位ビットを切り捨てるからです。しかし、この方法では負の無限大に向かって数値結果にバイアスがかかる可能性があります。丸め込みモードにおけるこの数値バイアスは、数値精度と性能におけるトレードオフのもう 1 つの例です。

丸め込みとオーバーフローが FPGA の性能、リソース使用、数値精度にもたらす影響の詳細については、リソース最適化テクニックの章、または LabVIEW 製品マニュアルの固定小数点データタイプ（FPGA Module）のトピックを参照してください。

パイプライン処理、レジスタ、実装のリソース

SCTL 内で使用するよう構成する場合、乗算器は内部パイプライン処理を実施して、入出力端子にレジスタを追加します。それにより処理経路が分割され、高速でのコンパイルが可能となります。

さらに、関数の実装時にルックアップテーブル（LUT）と内蔵の乗算器（SDP48 ブロック）の自動選択をするかどうかを指定することで、コンパイラが使用するリソースのタイプを決めることができます。DSP48 ブロックは Xilinx の専用 FPGA エレメントですが、LUT は汎用ロジックブロックです。上述の例から、高スループットノードによる専用リソースの活用方法や、リソースのタイプによるリソース使用のバランスの取り方がわかります。DSP48 の詳細については、本章の **DSP48 ノード**のセクションを参照してください。リソース最適化テクニックの章では、LUT と専用 FPGA リソースの役割についてさらに詳しく説明しています。

図 33 に示すように、実現可能な最高性能は、実装リソースやパイプライン段数、データタイプ幅によって異なります。

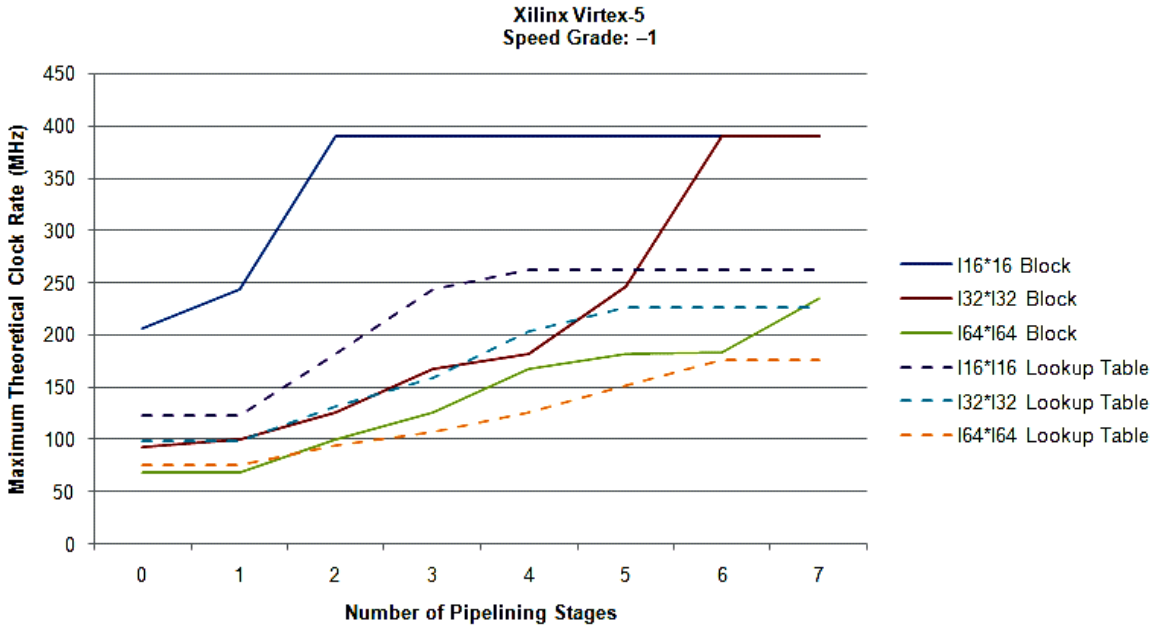

図 33. 積関数の最大クロックレートは、データタイプ、パイプライン段数、関数の実装に使われているリソースのタイプによって決まります。狭いタイプで DSP48 ブロックを使用する場合が最も高性能になります。

6.3. IP ハンドシェイクプロトコル

SCTL 内で使用するよう構成する場合、高スループット数学関数パレットにあるノードは、複数のサイクルを使って出力を初期化または生成するため、インタフェースにハンドシェイクプロトコルが必要となります。

ハンドシェイクプロトコルとは、例えば DSP や画像処理アプリケーションにおいて、処理チェーンの上流または下流ブロックに、出力の有効性や新しい入力受信の準備状態などの情報を伝える信号の集合です。

スループット最適化テクニックの章で説明したように、ハンドシェイクは SCTL 内で必要なものです。その理由は、マルチサイクルノードは有効なデータを計算するのに複数のサイクルを要するのに対し、SCTL はそれらのノードが毎クロックサイクルで何らかのデータを返すように強制するためです。したがって、マルチサイクルノードは毎クロックサイクルで有効なデータを返しません。アルゴリズムの数値的有効性を確認するには、このデータに依存するノードが、データが有効かどうかを認識する必要があります。

1 サイクルで実行できる関数は、ハンドシェイク情報を上流に伝搬することも可能です。これが必要なのは、実行時に下流の関数がデータを常に受信できるわけではなく、データの流入は上流コードにて制御する必要があるからです。

4 線式ハンドシェイクプロトコル

高スループット数学関数パレットにある関数は、ハンドシェイクプロトコルをサポートしていますので、SCTL 内に処理チェーンを作成することができます。このプロトコルでは 4 つの端子を使用するため、FPGA 関数/VI のハンドシェイクは 4 線式プロトコルとしても知られています。このプロトコルには、以下の端子が含まれます。

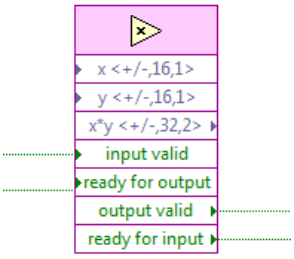

図 34. 高スループット積関数は、4 線式ハンドシェイクインタフェースを提供する関数の一例です。4 つの信号を使用して、SCTL 内の有効データのフローを制御します。

入力有効

ハンドシェイク関数は、受け取るデータの有効性について認識している必要があります。この信号は、関数に渡されたデータが有効かどうかを実行前に特定します。上流コードで 4 線式プロトコルを実装すると、上流の出力有効信号を入力有効端子に接続することができます。

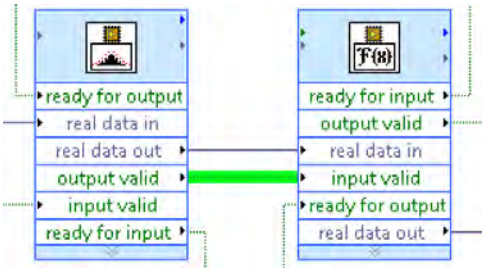

図 35. 4 線式ハンドシェイクインタフェースをサポートする 2 つの関数は、上流関数の出力有効信号を下流関数の入力有効入力に接続して、有効なデータの転送を知らせます。

受信データは各サイクルで有効と認識されている場合もあります。そのような状況では、図 36 に示すように、この入力に TRUE 定数を配線します。

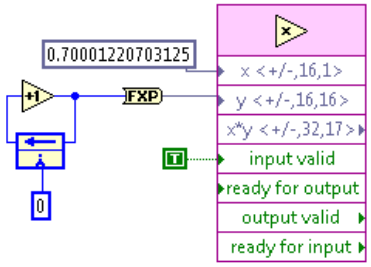

図 36. 受信データが常に有効であると予測される場合でも、入力有効端子に値を配線する必要があります。

上流コードでは出力有効信号が生成されないこともあるので、データの有効性を特定する他のメカニズムを見つける必要があります。図 37 に示す例では、「FIFO 読み取り」メソッドのタイムアウト出力がデータ有効性を示す役割を果たしています。タイムアウト出力が TRUE の場合、FIFO 読み取りメソッドから受信するデータは無効になります。

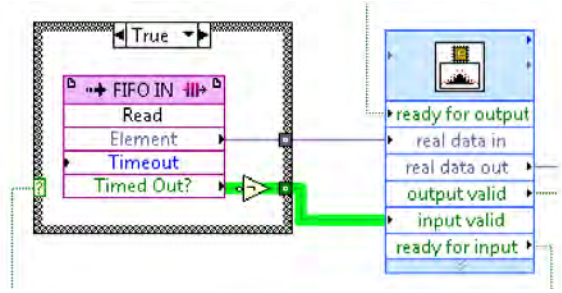

図 37. 関数やノードには 4 線式ハンドシェイクインタフェースを持たないものもありますが、FIFO 読み取りメソッドのケースと同様、データの有効性を推測する方法はあります。FPGA FIFO メソッドの要素出力は、「タイムアウト?」端子が TRUE の場合無効になります。

出力有効

関数の実行後、この出力は、関連の出力信号が有効なデータを含んでいるかを示します。

下流コードでは、渡されるデータの有効性を確定する必要があることもしばしばです。特に I/O、ストレージや状態の変化、データ転送操作など、副次的影響があればなおさらです。

下流コードでも 4 線式プロトコルが実装できれば、その出力有効信号を適切な下流関数の入力有効端子に接続することができます。

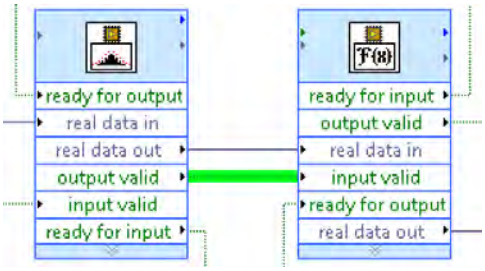

図 38. すでに説明したように、4 線式ハンドシェイク関数は、上流関数の出力有効信号を下流関数の入力有効入力に接続することで、各データの有効性を伝えます。

下流コードに、ハンドシェイク端子が内蔵されていないなど、渡されるデータの有効性について情報を伝える手段がない場合もあります。例えば、多くの I/O ノードや、LabVIEW FPGA の通信/ストレージコンストラクトなどです。そのような場合、図 39 に示すように、ケースストラクチャを使用して、ノードが無効なデータを生成した際、下流コードが実行しないようにすることができます。

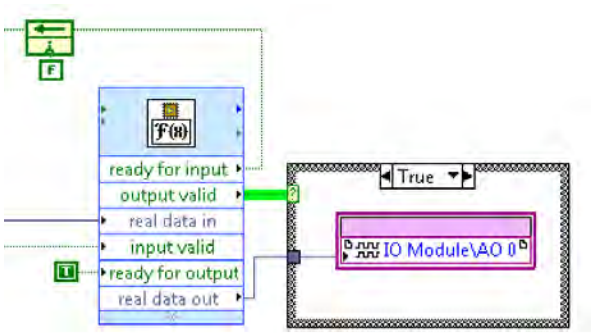

図 39. 関数やノードには 4 線式ハンドシェイクインタフェースを持たず、無効なデータが渡されても実行を防げないものもあります。ケースストラクチャを使用して、出力有効信号の値に基づき、下流コードを選んで実行することができます。

出力準備完了

この入力信号は、下流ノードがこのノードから新しい出力を受け取ることができるかどうかを示します。また、下流ノードは 4 線式に対応しているため、この信号により、現在の反復でデータを受け取れる準備ができていると解釈することができます。

信号は通常下流ノードから来るので、フィードバックノードを使用してダイアグラム上でサイクルを防ぐ必要があります。また、下流コードが最初の呼び出しで常にデータを受け取ることができない限り、フィードバックノードは FALSE で初期化する必要があります。

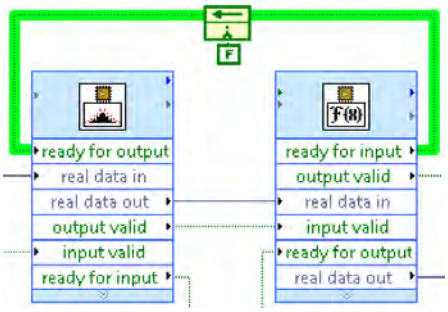

図 40. 出力準備完了入力は、下流コードがループの現在の実行で生成されたデータを受信できることをハンドシェイク関数に伝えます。

下流ノードは 4 線式に対応していないものもありますが、現在または次の反復でデータを受け取れるかどうかを特定する信号が含まれている可能性があります。「FIFO 書き込み」メソッドは、その一例です。

FIFO 書き込みメソッドのタイムアウト端子は、最新の要素が FIFO に書き込めなかった場合（多くの場合満杯のため）に TRUE を出力します。これを利用して、図 41 に示すように、上流ブロックがそれ以上のデータを出力しないようにしたり、出力しようとした要素値を保持して、後で書き込み操作を再実行することができます。

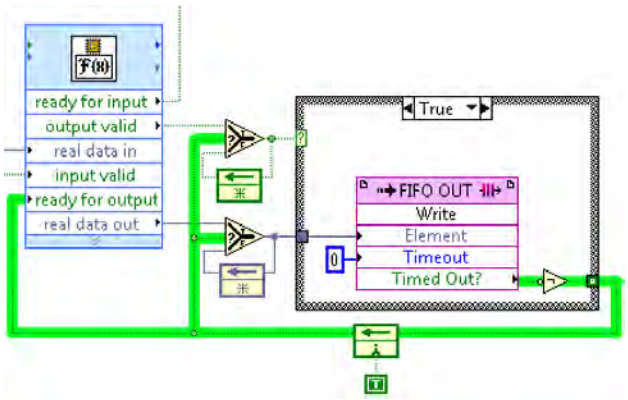

図 41. FIFO 書き込みメソッドのタイムアウトは、書き込みを試みた後 FIFO がいっぱいであることを示します。そのため上流ノードは新しい値の出力を停止し、FIFO に空きができるまで最新の要素が保持されます。このようなコードパターンは、上流コードでハンドシェイクインタフェースを使用している際にデータの損失を防ぐために重要です。

入力準備完了

この出力は、ハンドシェイクノードが次のサイクルでデータを利用できるかどうかを上流ノードに知らせます。信号は上流への伝搬されるため、フィードバックノードを使用してダイアグラム内のサイクルを防ぐ必要があります。

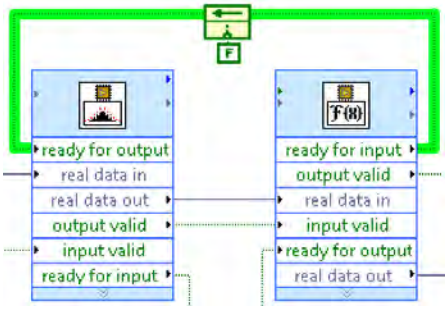

図 42. 4 線式インタフェースを持つ 2 つの関数が接続されると、下流関数の入力準備完了信号はフィードバックノードを通って上流へ 1 つ戻る必要があります。

上流コードは、4 線式に対応していない場合もあれば、ハンドシェイク端子を搭載していない場合もあります。そのようなケースでは、入力準備完了信号は、解釈し直すか、ケーストラクチャを制御して、上流ノードが実行しないようにすることが必要になる場合があります。

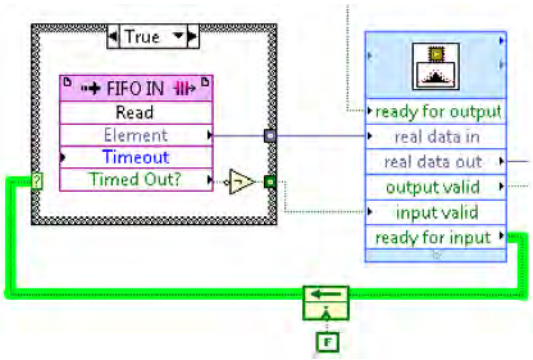

図 43. 上流コードが 4 線式ハンドシェイクインタフェースを備えていない場合、入力準備完了信号はコードの実行に使用されることになります。ケースストラクチャや他のメカニズムを使用して、ハンドシェイク関数の受信準備ができたときに、コードが有効なデータのみを生成するようにします。

ハンドシェイクロジックを検証する

ハンドシェイク信号の書き込みが正しく行われないと、誤った動作やデータの損失、データの破損、データの重複、性能の低下などにつながる可能性があります。ハンドシェイク接続を検証するには、シミュレーションモードでコードを実行し、SCTL の複数の反復で動作を確認します。

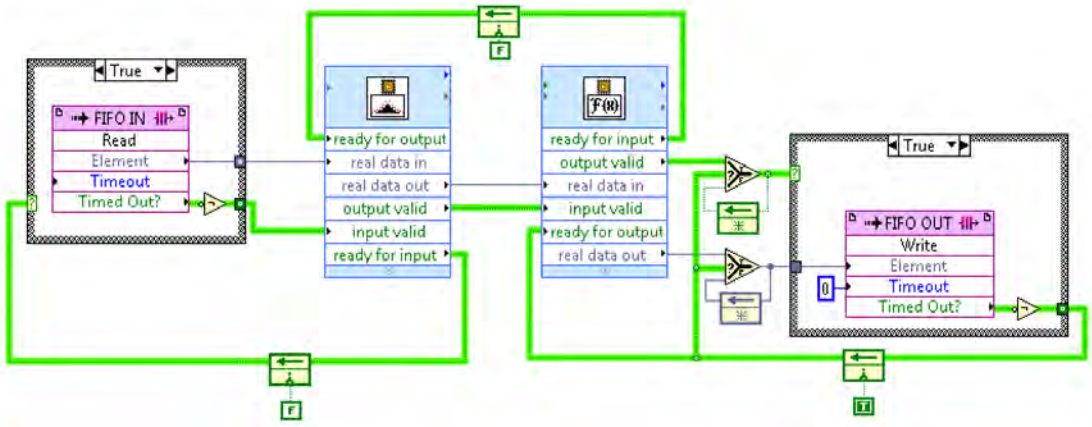

図 44. この SCTL コードは、FIFO からサンプルを読み取り、窓関数で処理をし FFT を実行してから、サンプルを FIFO にもう一度書き込み、別のループで処理を行います。コードには上記のセクションで説明した 4 線式ハンドシェイクプロセスのほとんどが含まれています。このダイアグラムのワイヤの多くは、有効データのフローを維持するためのハンドシェイク信号です。それらの信号を検証することは、コードの機能の正確性を確保するのに重要です。

また、シミュレーションモードを使用して IP の遅延時間と開始間隔を計測することもできます。それには、サンプルがチェーン内を伝搬するのにかかるサイクル数と、PI が新しい入力を受信できるようになる（入力準備完了端子によって伝えられる）までにかかるサイクル数を特定します。

サンプリングプローブは、LabVIEW FPGA IP のハンドシェイクと機能動作を検証するための LabVIEW の新しいプローブです。サンプリングプローブを使用すると、SCTL がシミュレーションモードで実行する際、図 45 に示すようにデジタル波形で

表示されます。

NATIONAL INSTRUMENTS

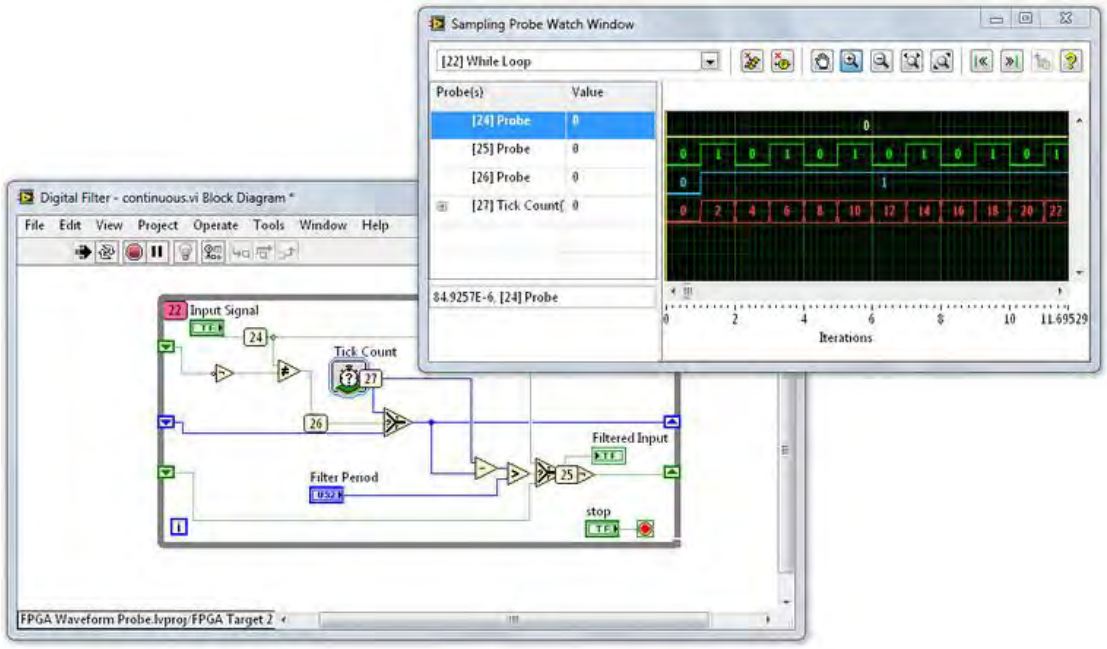

図 45. サンプリングプローブを使用すると、シミュレーションモードでワイヤ値の履歴が表示されます。ハンドシェイク信号の機能など、IP の正確性と性能を検証するのに便利です。

6.4. 処理チェーンのスループットを特定する

多くの LabVIEW FPGA 関数は、開始間隔に関する情報を提供します。その情報を使用して、SCTL 内で接続された一連のノードで、実現可能な最大スループットを計算できます。**スループット最適化テクニック**の章で説明したように、ノードチェーンの開始間隔は、一連のノードの最大開始間隔と同等です。開始間隔が最大のノードが、処理チェーンの中でボトルネックとなります。

ノードの開始間隔は、構成ダイアログ、詳細ヘルプ、リファレンストピックに記述されています。開始間隔のことをスループットと呼ぶこともありますが、1 サンプルあたりのサイクル数で表すので、スループットとは反比例した数値となります。

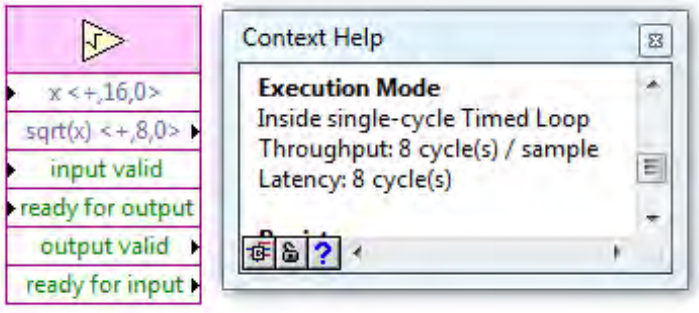

図 46. 多くの高スループット関数の詳細ヘルプには、指定した関数構成に基づいて、予測されるスループット（開始間隔サイクル）と遅延時間が記載されています。

6.5. DSP48 ノード

DSP48 ノードは、LabVIEW FPGA で使用できる、高性能処理 IP を作成するための専用 FPGA 要素の一例です。

DSP4E は、Xilinx Virtex-5 など特定の FPGA デバイス製品に搭載されたデジタル信号処理要素です。このスライスを使用して、乗算アキュムレータ、乗算加算器、1～N ステップカウンタなど様々な種類の演算を実行することができます。またこのスライスを使用して、AND、OR、XOR 演算といった各種論理演算を実行することも可能です。

Xilinx 社のデバイス製品には、DSP48E や DSP48E1 など様々な DSP48 ノードが搭載されていますが、いずれも基本機能は似通っています。複数の DSP48E スライスをカスケードすることで、FPGA ファブリックリソースを追加で使用しなくても、複素乗算器や n タップ有限インパルス応答（FIR）フィルタなど、より複雑な機能を実装することが可能です。そのため、DSP48 スライスから作成した IP は高クロックレートでコンパイルすることができます。

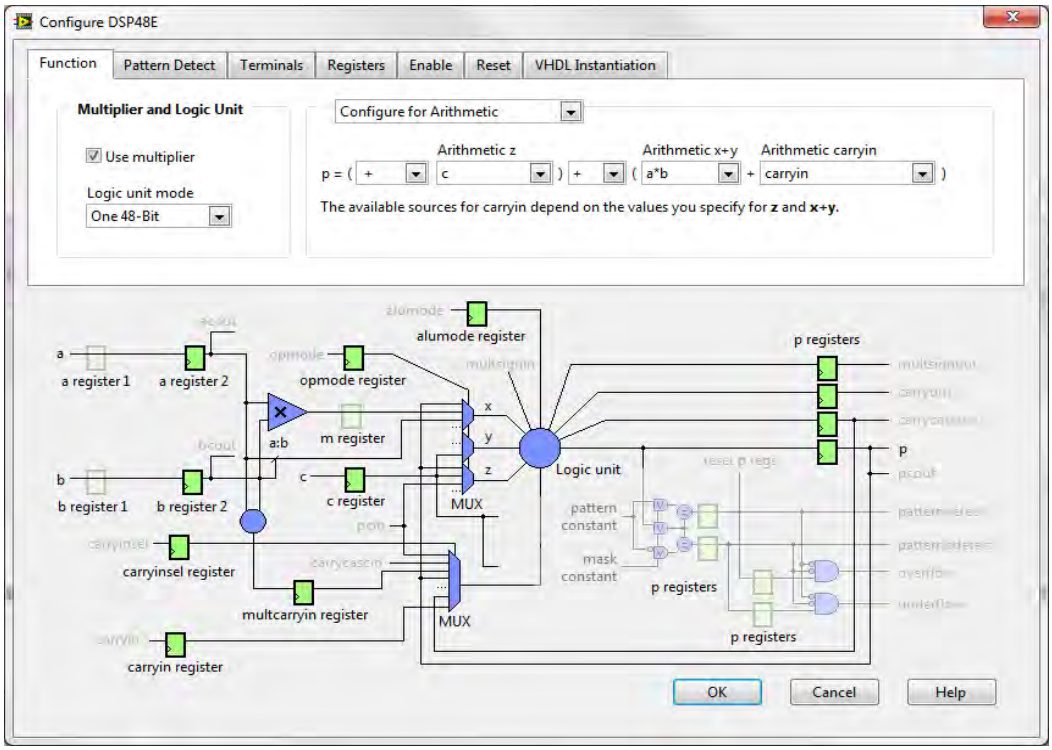

図 47. DSP48E ノードの構成ダイアログでは、様々な機能を有効にして構成すると、DSP48E 回路が図で表示されます。

このノードの使用は難度が高いため、DSP48 は究極の最適化が求められるような場合にのみ使用することをお勧めします。このノードを構成する際には、事前に Xilinx 社の Web サイト（xilinx.com）にある『Xilinx Virtex-5 FPGA XtremeDSP Design Considerations』の「DSP48E Description and Specifics」を参照してください。「DSP48E アプリケーション」の章には、多彩なサンプルアプリケーションのほか、機能や構成の詳細、注意点などの情報が掲載されています。同様の高性能要件の課題に取り組む際には、本章を参照することをお勧めします。

このノードで、以下のサンプル IP のタスクを実行することができます。

基本の数学アプリケーション

- 25 x 18 2 の補数積
- 2 入力 48 ビット和
- 4 入力 46 ビット和
- 2 入力 48 ビットダイナミック和/差
- 3 入力 47 ビットダイナミック和/差
- 25 x 18 積＋48 ビット和/差
- 拡張積
- 浮動小数点乗算と 59 x 59 符号付き乗算
- 25 x 18 積＋48 ビット加算カスケード
- 除算フィルタ
- 多相補間 FIR フィルタ
- 多相デシメーション FIR フィルタ
- マルチチャンネル FIR
- フィルタ係数のプリロード

上級数学アプリケーション

- 累積
- 2 入力 48 ビット和累積
- ダイナミック和/差累積
- 96 ビット和/差
- 96 ビット累積
- MACC/MACC 拡張
- 25 x 18 複素数乗算
- 35 x 25 複素数乗算
- 25 x 18 複素数 MACC

論理/ビットフィールドアプリケーション

- 2 つの 48 ビット入力ビット単位論理関数
- ダイナミックシフタ
- 18 ビットバレルシフタ
- 48 ビットカウンタ
- Single Instruction Multiple Data（SIMD）演算
- SIMD 絶対値（24）
- バスマルチプレクサ

パターン検出アプリケーション

- ダイナミック C 入力パターンマッチ
- オーバーフロー/アンダーフロー/飽和
- オーバーフロー/アンダーフロー/飽和
- 論理ユニットとパターン検出
- 48 ビットカウンタ自動リセット

丸め込みアプリケーション

- 丸め込み処理
- ダイナミック/スタティック小数点
- 対称丸め
- ランダム丸め
- 収束丸め
- 収束丸め：LSB 補正テクニック
- ダイナミック収束丸め：キャリー補正テクニック

さらに LabVIEW FPGA には、ノード構成時の参照として使用できる DSP48 スライスのサンプルが付属しています。DSP48 サンプルは、NI サンプルファインダで「DSP48」を検索すると見つけることができます。

6.6. 高速フーリエ変換

FPGA のリソースは限られているため、SCTL 内に作成された処理チェーンは、データに対しポイントごとに処理を行い、1 つのブロックから次のブロックへ値をストリーミングします。一部の関数では、データブロックを一度に 1 つずつ処理する必要があるため、処理可能となるまでに複数サイクルを使ってサンプルをバッファするものもあります。そのような関数には、「FPGA 数学＆解析」VI/関数パレットにある高速フーリエ変換（FFT）関数があります。

FFT 関数は、データのブロック、つまりフレームを処理しますが、その際ストリーミング、連続、またはバーストパターンから選ぶことができます。

バースト操作時には、FFT 関数は一度に 1 ポイントずつ供給されたデータを受け取りバッファします。十分なデータを収集してフレームがいっぱいになったら、新しいサンプルの受け取りを停止し、バッファされたフレームの処理を開始します。数サイ

クル後に、FFT 関数は出力フレームを一度に 1 ポイントずつ提供します。そして図 48 に示すように、最新の結果フレームをクロック出力した後、しばらくしてから新しいフレームのサンプルを受け取ります。

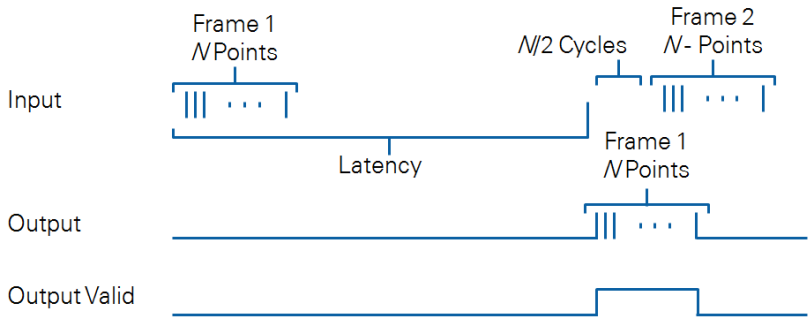

図 48. FFT 関数はバーストまたはストリーミングモードで動作します。バーストモードでは、一度に 1 フレームずつ処理し、数サイクルを使って結果フレームを受信、処理、出力してから新しい入力フレームを受け取ります。

バーストモードでは、開始間隔は複数サイクルになります。FFT 関数の構成ダイアログでは、予測される開始間隔に対するフィードバックを提供します。開始間隔は、FFT のブロックサイズと、内部パイプラインと並列化の程度によって決まります。

ストリーミングモードでは、FFT 関数はスループット最適化テクニックで解説した原理に基づいて、1 サイクルあたり 1 サンプルの開始間隔を実現します。その場合新しいサンプルを受信し続けることができます。それでも処理を行うにはデータのバッファが必要なので、図 49 に示すように、現在のフレームの処理と前の結果の出力がまだ終わらないうちに、バッファと回路を関数に追加して次の FFT 入力ブロックを受信します。

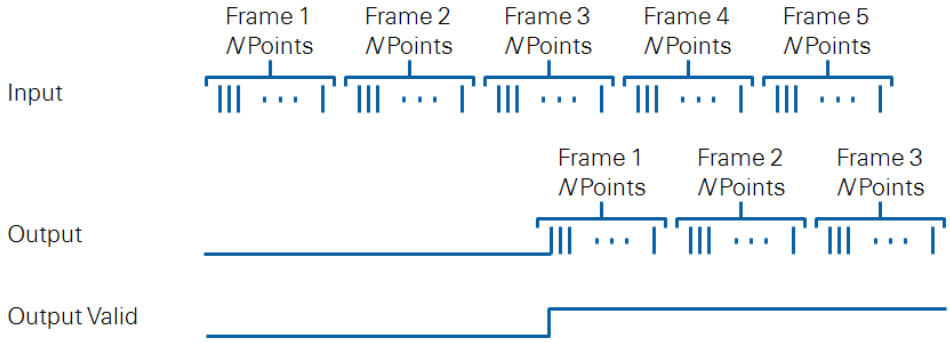

図 49. ストリーミングモードでは、FFT 関数は全サイクルで新しいサンプルを受信します。内部的には、FFT はサンプルをバッファし、FFT 演算を並列化して、以前のフレームの結果を同時に出力するようになっています。このモードが最速ですが、リソースを多く使用します。

最高の FFT 性能を実現するストリーミングモードでは、受信する全サンプルに対し FFT を実行することができます。しかし内部構造が複雑で、一度に複数のフレームを保持し処理するには複製する必要があるため、かなりのリソースを使用します。

6.7. Xilinx 社の CORE Generator IP システム

LabVIEW FPGA を使用すると、Xilinx Coregen IP 関数パレットから直接 Xilinx の CORE Generator IP システムにアクセスすることができます。CORE Generator IP システムとは、自動車、通信、信号処理、その他様々なアプリケーション向けの、アーキテクチャ、ドメイン、市場に特化した IP を集めたものです。Xilinx 社では、高性能設計を組み合わせて使用する Xilinx FPGA のユーザ向けに最適化された IP ブロックを提供しています。ブロックは LabVIEW FPGA 開発環境との緊密な統合が可能で、ステップごとの構成ダイアログも付属していますので、指定したコアの性能やデータタイプなど IP 固有の動作をカスタマイズすることができます。

図 50 は、Xilinx CORE Generator の FIR フィルタコアを示しています。左はダイアグラム上に表示される LabVIEW FPGA 関数、右は構成ダイアログです。コアに含まれる構成設定を使用して、アプリケーションの要件を満たすよう FIR フィルタをカスタマイズすることができます。

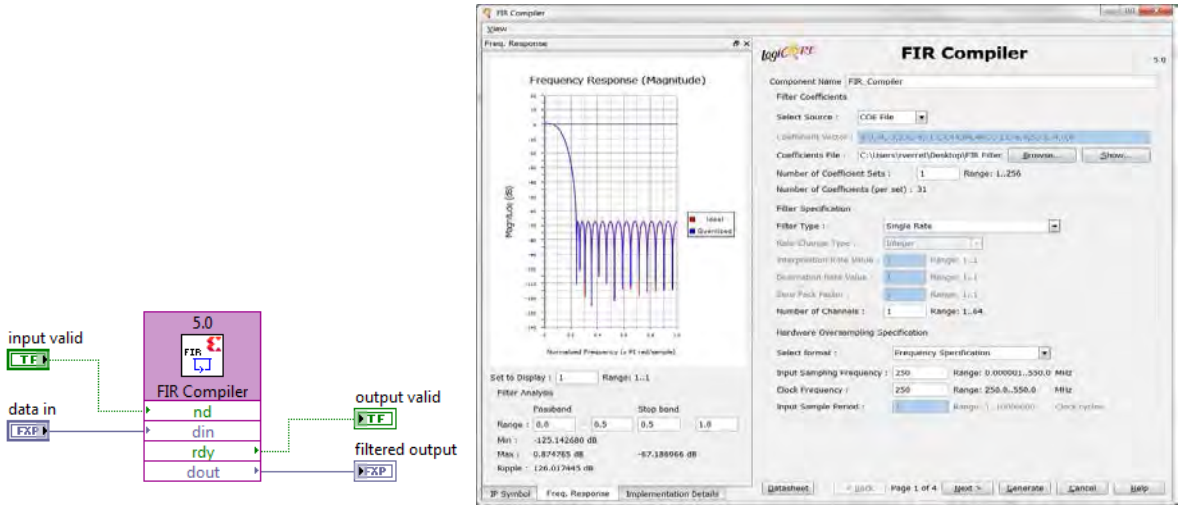

図 50. LabVIEW FPGA は Xilinx 社の CORE Generator IP システムの IP と緊密に統合できます。関数にはそれぞれパレットがあり、LabVIEW 開発環境から直接適切な Xilinx CORE 構成ダイアログを起動することができます。

これらの関数は、FPGA デバイスのメーカーである Xilinx 社が開発したものであるため、NI の同等関数に比べ機能や性能、リソース使用の面で優れているものもあります。そのため設計を最適化する際に使用を検討する価値はあります。LabVIEW FPGA パレットに含まれている Xilinx CORE Generator IP の中で、特に最高性能の関数は、ライセンス料が必要なものもありますが、専門性の高い IP をお探しなら、実装とテストに使用した際のコスト効果は十分です。

Xilinx CORE Generator FFT IP

Xilinx 社の FFT Core は、LabVIEW FPGA Module の FFT Express VI の機能を補う一般的な IP ブロックの一例です。Xilinx Core は、FFT Express VI と同様、ストリーミングモードや 3 種類のバーストモードのいずれかで動作することが可能で、それにより性能とリソースのトレードオフも異なります。さらに、Xilinx FFT はストリーミングモードの際かなりのリソースを消費します。他に方法がないような場合以外は、このモードを使用しないことをお勧めします。図 51 の図は、各モードの性能とリソースのトレードオフを示すガイドとして、Xilinx 社が作成したものです。

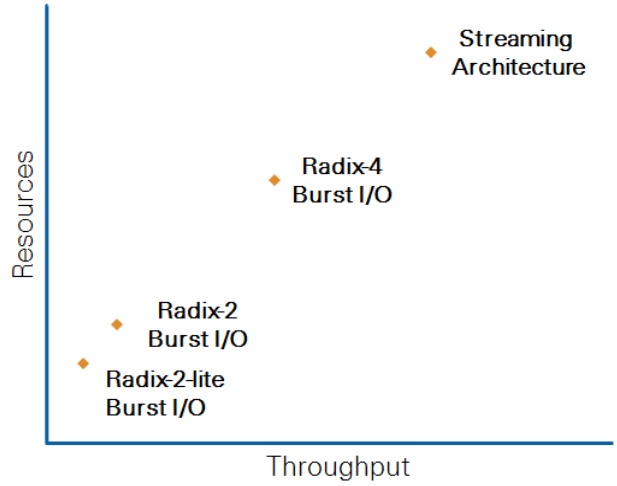

図 51. Xilinx FFT の各操作モードは、それぞれスループットとリソースのトレードオフが異なります。バーストモードはリソース使用量が最少ですが、各フレームの処理に複数サイクルを要します。ストリーミングモードはデータを連続的に処理できますが、かなりのリソースを使用します。

FFT Express VI と Xilinx CORE Generator FFT のどちらかを選ぶ際の参考として、Xilinx CORE Generator FFT には以下のようなメリットがあります。

- スループットを最大限にする設計のため、通常高クロックレートでコンパイル可能
- 8192 サンプルを超えるフレームサイズをサポート
- バーストモード時マルチチャンネル操作をサポート
- 浮動小数点データタイプをサポート（ただしかなりのリソースを消費）
- コンパイル時に予め定義した数値までランタイム構成が可能な FFT フレームサイズ（フレームサイズを変更した場合はコアのリセットが必要）
- フレームごとに順モードまたは逆モードで構成が可能（FFT 関数を使用して順 FFT を実行し、周波数領域で信号を操作してから、同じ FFT 関数/ハードウェア回路を再利用して逆 FFT を実行し、信号を時間領域に再度変換することができます）

Xilinx CORE Generator FFT には以下のような課題があります。

- Xilin CORE Generator FFT は、FFT Express VI に比べ構成が難しいことがあります。
- 4 線式プロトコルに似た標準のハンドシェイクインタフェースを搭載していません。例えば、入力有効信号がないため、本章ですでに説明したように、関数をケーススストラクチャに配置して、無効なデータを供給しないようにする必要があります。
- コアの最高性能を引き出すには高度なカスタマイズが必要で、設計にある特定の条件が求められます。例えば、最高性能のモードでは、サンプルが逆順序に並べられた状態でフレームが生成されます。スループットレートを維持するには、下流ブロックでこのような制約に対処可能でなくてはなりません。

コアを使用した設計を検討している場合は、Xilinx LogiCORE IP Fast Fourier Transform データシートの最初の数章を参照することをお勧めします。この文書は技術的な内容である程度のデジタル設計の知識が必要ですが、最高の FFT 性能を実現するには有益な情報です。LabVIEW Coregen IP パレットにある他の Xilinx CORE Generator IP についても同じことが言えます。

AXI プロトコル

ナショナルインスツルメンツでは、一部ハードウェアターゲット上の特定の Xilinx CORE Generator IP 向けハンドシェイクプロトコルである AXI（Advanced eXtensible Interface）を採用しています。ARM が 1996 年に発表した AXI プロトコルは、高性能の高周波アプリケーションで、IP の関数ブロックを相互接続するための業界標準バスインタフェースとなっています。

LabVIEW FPGA Module で使用する場合、信号接続に関しては LabVIEW 4 線式ハンドシェイクプロトコルと似通っています。どのデータにも、値を含む信号と値が有効かどうかを示す信号の 2 つの信号があります

AXI の使用時は、AXI IP に接続する際フィードバックノードを使用する必要があります。AXI プロトコルと LabVIEW 4 線式ハンドシェイクプロトコルの主な違いは、信号の名称と、SCTL 内で IP を接続する際のフィードバックノードの配置です。AXI ノートを他の AXI ノードと接続する方法や、AXI ノードを 4 線式プロトコル対応のノードに接続する方法に関する詳細な情報については、LabVIEW FPGA 製品マニュアルの「FPGA VI で AXI IP をインタフェース接続する」トピックを参照してください。

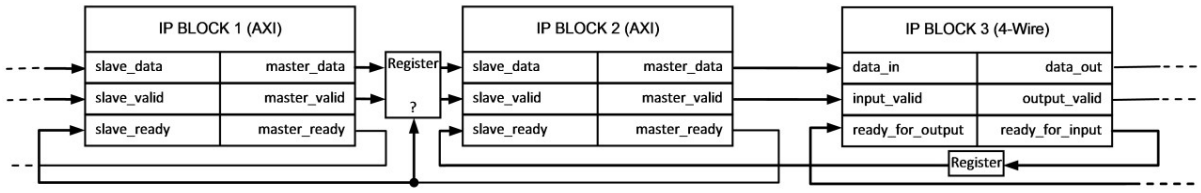

図 52. AXI 信号と 4 線式ハンドシェイク信号は、名前は異なりますが意味は似通っています。フィードバックノード（「レジスタ」と表示）の配置は、相互接続されている IP の種類によって異なります。

6.8. HDL IP を統合する

ハードウェア記述言語（HDL）とは、FPGA や特定用途向け集積回路（ASIC）でハードウェアコンポーネントを設計する際に使用するテキストベース言語です。この言語は、ロジック、信号、接続、並列性、タイミングといったハードウェア概念を表す標準的な方法です。また、開発者はゲートごとに回路を指定しなくても、構造、動作、低レベル実装などの細部を表すことができます。

本ガイドでは、HKL IP とは VHDL または Verilog で作成した IP、あるいはそのような言語から生成したネットリストファイルを示します。

VHDL と Verilog は、デジタル設計において長く使用されてきた歴史があり、ほとんどの EDA ツールでサポートされています。FPGA の設計も従来この方法で行われていました。ネットリストとは、HDL 設計のコンパイル済み出力のことを示す用語です。ネットリスト表記は、FPGA ハードウェア要素と、要素間の接続を直接的に指定するので、最終的な回路に近くなって

います。ネットリストを使用すれば、元の HDL コードを実際に開示することなく、IP を統合したり共有したりすることも可能です。

HDL IP は、様々な場所から入手できます。従来型のデジタル設計ツールを使用している場合は、HDL IP がすでにある可能性があるほか、HDL IP を開発する技術も持っている可能性があります。また、opencores.org などのコミュニティサイトでは、オープンソースの HDL IP を見つけることができます。他のデジタル設計企業も市場に特化した HDL IP を開発し認可しています。

IP をすでに持っていたり、HDL を使用して IP を作成する方が簡単な方は、HDL IP を LabVIEW FPGA の設計に統合することもできます。HDL なら下位レベルのハードウェアやツールチェーンの設定が変更できるので、極めて高い性能やリソース制御が必要な場合に利用できます。

HDL IP を LabVIEW FPGA にインポートするには、2 つのメカニズムがあります。

1. IP 統合ノード（IPIN）を使用する
2. コンポーネントレベル IP（CLIP）として

IPIN と CLIP のどちらを選ぶか

IPIN と CLIP のメカニズムは多くの点で異なりますが、どちらかを選ぶ際には、統合する IP の種類に基づいて決める必要があります。

入力を供給して呼び出しごとに IP が出力を生成する場合など、機能的、アルゴリズム的な特性を持つ IP や処理関数として動作する IP を統合する際は、IPIN を使用します。

アプリケーションの複数個所とやり取りする IP など依存関係のない非同期コンポーネントや、メモリ、SPI、I²C など何らかの外部デジタルコンポーネントと通信する IP を統合する場合は、CLIP を使用します。

IPIN は SCTL 内に配置する必要があり、LabVIEW の呼び出しモデルと同様、データがノードに渡されると、ノードは出力データを生成し、そのデータを利用する別のノードに渡します。IPIN は HDL コードから直接 I/O にアクセスすることはできません。そのため、必要な値を入力または出力として受信または生成する必要があります。IPIN を SCTL 内に配置すると、IP の 1 インスタンスとなります。ノードを 2 つ配置したら、IP の 2 つのインスタンスとなります。

IPIN と異なり、CLIP は LabVIEW プロジェクトのみに表れるため、ダイアグラムに表示されません。CLIP は FPGA VI 外で動作するものであり、特定の CLIP のインスタンスを複数作成することができます。LabVIEW FPGA のダイアグラムは、CLIP アクセサリノードを介して CLIP IP と通信します。CLIP アクセサリノードとは、I/O ノードと似たもので、CLIP インタフェースレジスタの読み取りと書き込みが行えます。FlexRIO ハードウェアでは、I/O ピンにアクセスすることもできます。このタイプの CLIP は「ソケット付き」CLIP とも呼ばれます。

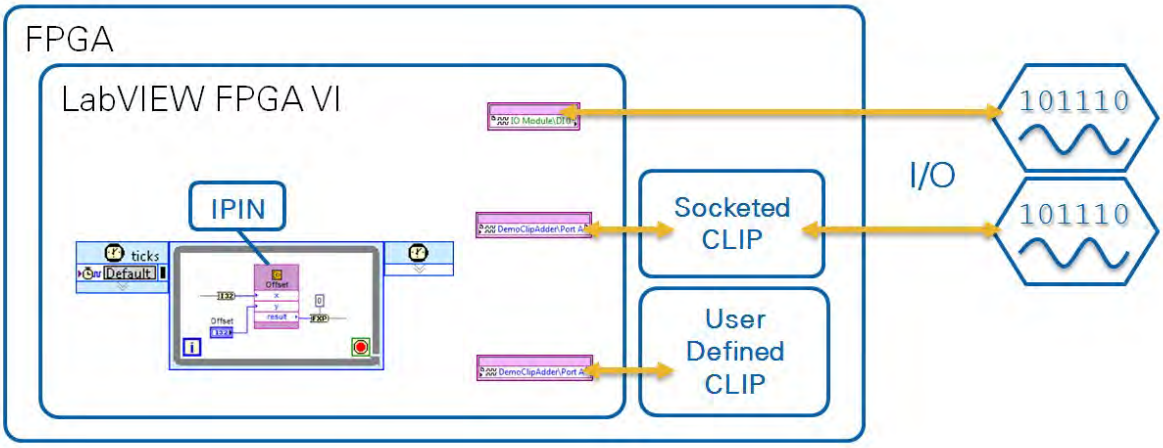

図 53. CLIP と IPIN は LabVIEW FPGA の HDL 統合メカニズムです。IPIN は、IP を関数としてモデリングできる場合に適しています。CLIP は、IP が独自のコンポーネントとして実行する必要がある場合に適したオプションです。CLIP は、RIO デバイスの種類により、I/O にアクセスできることもあります。

以下の表は、CLIP と IPIN のその他の相違点を示しています。

CLIP	IPIN
複数のクロック領域をサポートし、クロックを作成し FPGA VI と共有可能	SCTL が内部の IPIN ノードのクロックを指定
FPGA VI との通信をサポート	LabVIEW FPGA ダイアグラムに含まれる
特定のターゲットで FPGA I/O へのアクセスをサポート	HDL IP 内から I/O へのアクセスは不可
制約ファイルをサポート	浮動小数点をはじめ上級データタイプを受け取り可能
Mentor Graphics QuestaSim や Xilinx ISim などサードパーティシミュレータに対応	LabVIEW FPGA の内蔵シミュレーション機能をサポート

これらの統合メカニズムの詳細については、製品マニュアルと製品に含まれているチュートリアルサンプルを参照してください。

6.9. IP をソフトウェア設計型計測器に統合する

NI ベクトル信号トランシーバ（VST）などのソフトウェア設計型計測器は、LabVIEW FPGA Module を使用して動作をカスタマイズすることができます。例えば VST のケースでは、NI-RFSG と NI-RFSA のドライバ機能以上のものが必要な場合、2 つのソフトウェアスタックオプションがあります。最初のオプションは、ハードウェアドライバ拡張機能 FPGA Extensions を使用して共通の変更や部分修正を行うものです。2 つ目のオプションは、LabVIEW のオープンソフトウェアスタックを使用して動作をフルカスタマイズするものです。

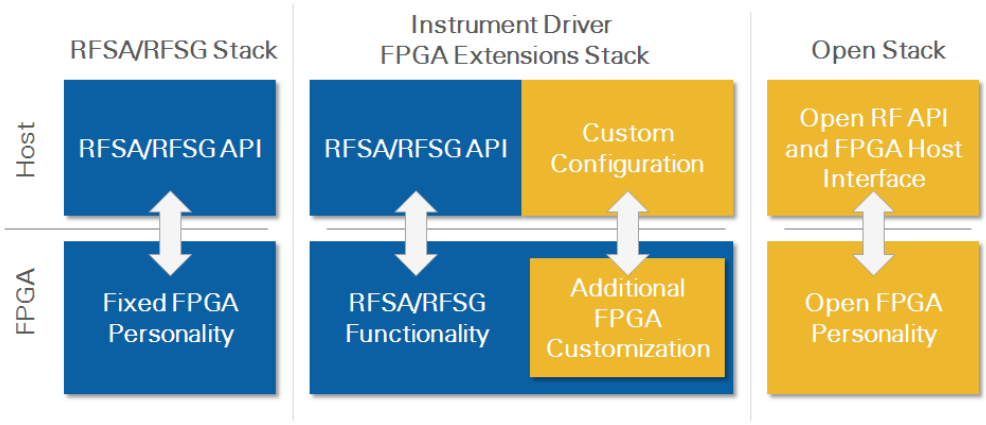

図 54. VST は、標準の RFSA/RFSG 計測器ドライバ、完全なオープンソフトウェアスタック、または中間のハードウェアドライバ拡張機能 FPGA Extensions とともに使用して、計測器ドライバの標準機能のメリットを活用した一般的なカスタマイズを行うことができます。

オープンスタックオプションには、一般的なアプリケーションの実装に適した完全なオープンアーキテクチャを提供するサンプルプロジェクトが付属しています。スタックのホスト側も FPGA 側も必要に応じて変更することができます。この方式は、ホスト側と FPGA 側の両方で一から開発する場合に適しています。例えばナショナルインスツルメンツのアライアンスパートナー、Averna 社の DOCSIS Channel Emulator など、デバイスのフロントエンド I/O 機能を利用して、完全なカスタム計測器を開発することができます。

ハードウェアドライバ拡張機能 FPGA Extensions

ハードウェアドライバ拡張機能 FPGA Extensions を使用すると、VST の場合であれば、NI RFSA/NI-RFSG ドライバなど標準の計測器ドライバを使用しつつ、FPGA 上で計測器機能の変更や拡張を行うことができます。

スタックの FPGA 部分は、標準の計測器ドライバが想定する計測器機能を実装し、トリガやフィルタ、その他の信号処理など、特定の機能をオーバーライドしたり拡張したりできる場所を明確に特定します。

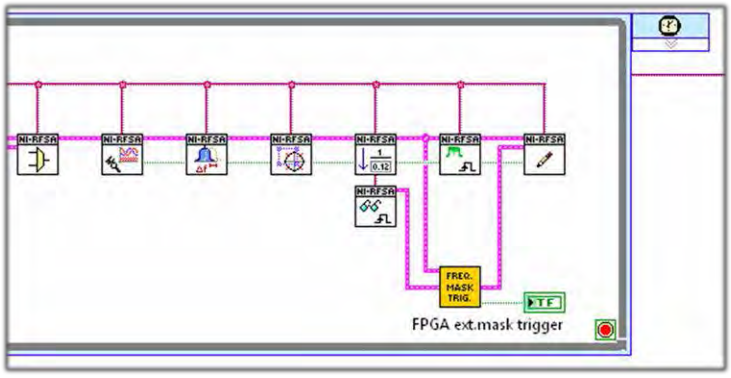

図 55. ハードウェアドライバ拡張機能 FPGA Extensions により、標準の RFSA/RFSG 計測器ドライバ API と互換性のある FPGA コードが使用可能になります。同時に、FPGA の主要部分を特定し、カスタマイズすることもできます。

6.10. コミュニティから入手した IP を統合する

プロジェクトに含まれていない関数を実装する際に使用できる高性能 LabVIEW FPGA IP を、外部から入手することもできます。本セクションではそのような入手先をいくつか紹介します。

FlexRIO/ソフトウェア設計型計測器 IP コミュニティ

このコミュニティは、ソフトウェア設計型計測器と FlexRIO デバイス 向けのサンプルと IP を集めたセントラルレポジトリです。I^2C から完全なチャンネルエミュレーションのサンプルまで、高性能デバイス向けの特定のプロトコルや関数のほか、疑問点の解決に役立つユーザコミュニティも見つけることができます。

ソフトウェア設計型計測器向けサンプルの多くは、プロジェクトに含まれる LabVIEW サンプルプロジェクトをベースに開発されています。それらのサンプルは、ハードウェアドライバ拡張機能 FPGA Extensions の概念を利用したものや、FlexRIO 計測器開発ライブラリを IP の構成単位として利用するものがあります。

FlexRIO 計測器開発ライブラリは、収集エンジン、DRAM インタフェース、トリガロジックなど、計測器に共通して搭載されている FPGA 機能や、関連するホスト API を提供する LabVIEW ホストや FPGA コードを集めたものです。ライブラリはモジュール式なので、必要なコンポーネントだけを選ぶことができます。またライブラリは効率的な実装が可能なので、アプリケーションのニーズに合わせて、提供されたコードを必要に応じ変更することもできます。

IPNet と LabVIEW ツールネットワーク

IPNet とは、NI 開発者とユーザコミュニティにより LabVIEW FPGA 向けに作成された IP とサンプルを集めたものです。LabVIEW ツールネットワークは、例えば LabVIEW RIO プラットフォーム向けに特別に作成されたものなど、様々なアプリケーションやデバイスに使用できる認定済みのサードパーティアドオンが利用できるサイトです。

ご自身で LabVIEW FPGA IP を作成する前に、IPNet と LabVIEW ツールネットワークを検索してみることをお勧めします。

関連情報

[1] ピアツーピアストリーミングの概要
http://www.ni.com/white-paper/10801/en

[2] Xilinx Virtex-5 FPGA XtremeDSP Design Considerations
http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[3] Xilinx LogiCORE IP Fast Fourier Transform Data Sheet
http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf

[4] FPGA VI で AXI IP を接続する
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/xilinxip_using/

[5] OpenCores
www.opecores.org

[6] DOCSIS チャンネルエミュレータ（DCE） – Averna
http://sine.ni.com/nips/cds/view/p/lang/en/nid/211379

[7] ハードウェアドライバの拡張機能 FPGA extensions を使用する(英語)
http://www.ni.com/white-paper/14648/en/

[8] FlexRIO 計測器開発ライブラリ
https://decibel.ni.com/content/docs/DOC-15799

[9] LabVIEW Digital Filter Design Toolkit
http://sine.ni.com/nips/cds/view/p/lang/en/nid/209040

[10] ソフトウェア設計型計測器/FlexRIO のサンプルと IP(英語)
https://decibel.ni.com/content/groups/software-designed-instrument-and-ni-flexrio-examples-and-ip

[11] ハードウェアドライバの拡張機能 FPGA extensions 概要
http://www.ni.com/white-paper/14646/en/

[12] LabVIEW FPGA IPNet
http://www.ni.com/ipnet/

[13] LabVIEW ツールネットワーク
http://www.ni.com/labview-tools-network/

NATIONAL INSTRUMENTS

7. タイミングの最適化テクニック

アプリケーションでは多くの場合、対象となるイベント間の時間を指定する必要があります。LabVIEW FPGA アプリケーションでは、これらのイベントは通常 I/O と関連付けられています。NI の FPGA ベースデバイスの応答速度が速いのは、I/O に近いハードウェア内で処理が行われるためです。この章では、非常に優れた確度・精度でイベント間のタイミングを制御・計測できる様々なテクニックをご紹介します。

制御アプリケーションでレイテンシが重要なのは、システムの状態がコントローラロジックによってサンプリング・処理された後、制御出力が一定の時間内で生成されるためです。制御アプリケーションでは多くの場合、低いレイテンシが求められます。

デジタルプロトコルアプリケーションでは、タイミングの制限も指定します。外部デバイスとの適切な通信を実行するには、この制限を守る必要があります。この場合、イベント間の正確なタイミングと再現性を維持することで、電気信号の整定とホールド要件が満たされ、正確なデータ伝送を行うことができるようになります。

7.1. SCTL でレイテンシを判断・指定する

処理コードパスの合計タイミングは、コンポーネントのレイテンシの合計です。SCTL（シングルサイクルタイミングループ）で正確なタイミングを保証できるのは、各反復にぴったり 1 クロックサイクルであるためです。複数の SCTL 反復全てにおいてイベント間のレイテンシは次のように表されます。

$$\text{レイテンシ} = \frac{\text{サイクル}}{\text{クロックレート}}$$

つまり、クロックレート、または対象となるイベント間のサイクル数を変更することで、レイテンシを決めることができます。

SCTL のクロックレートによって、タイミング分解能が決まります。クロックレートが高くなれば、タイミング分解能が高くなりますが、設計をコンパイルしにくくなる場合があります。したがって、クロックレートを高くする場合、求める分解能を得ることを考慮して、必要最小限の数値にとどめておく必要があります。

SCTL 内のイベント間のレイテンシを上げるには、その時間を SCTL の複数の反復の間で分散させつつ、分解能が適切な数値になるようなクロックレートを維持する必要があります。その最も基本的なやり方はフィードバックノードを挿入して、データが SCTL 内を通過するときに、遅延のサイクルを追加するというものです。

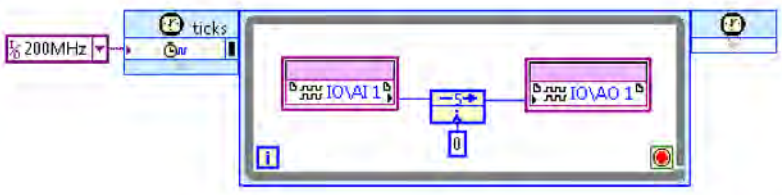

図 56. 入力と出力の間に 25 ナノ秒の遅延を追加できたのは、5 という遅延値を持ったフィードバックノードを挿入したからである。

LabVIEW のステートマシンパターンはより高度なやり方でイベント間の時間を制御します。それは、各ステートが、次のステートに移動する前に、経過したサイクル数を追跡できるためです。正確さが保証された反復レイテンシは、異なるステートの

イベント間のタイミングを制御する手段となります。

Revision No. 1.0 – January 2017

NATIONAL INSTRUMENTS

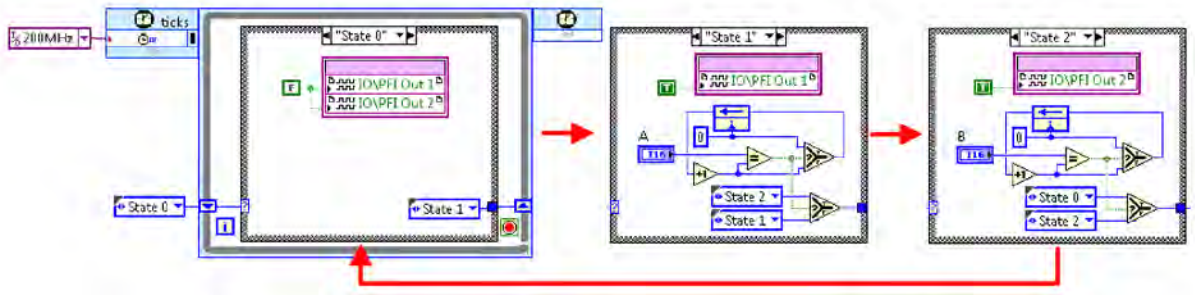

図 57. SCTL を使ったステートマシンパターンは、サイクル数をカウントすることで、デジタル出力イベント間のタイミングを制御する。最初のステートである State 0 は、2 つのデジタル出力ラインをアサート解除し、State 1 に進む。State 1 は、PFI Out 1 をアサートして、A サイクル待ってから State 2 に進む。State 2 は、B サイクル分、PFI Out 2 をアサートしてから、State 0 に戻る。I/O イベント間のタイミングは、パラメータ A と B によって決まる。この場合、分解能は 5 ナノ秒で、SCTL クロックは 200 MHz である。

イベント間の時間を減らすのは簡単ではありません。特に、イベントが既に複数の SCTL 反復にわたる場合、レイテンシは理想値より大きくなっています。レイテンシを小さくする最善の方法は、サイクル数やクロックレートの数値の大きさによって異なります。サイクル数が既に少ない場合、いくつかのサイクルを削除しただけで、レイテンシに大きな影響を与える可能性があります。反対に、サイクル数が既に多い場合（千単位など）、レイテンシに意味のある影響を与えられるだけのサイクルを削除するのは難しいかもしれません。この場合、パイプライン以外の手段でクロックレートを増やすのが最善策かもしれません。ここからは、レイテンシを小さくできるテクニックについてご紹介します。

7.2. 並列処理を使ってレイテンシを小さくする

「スループット最適化テクニック」の章でも述べたとおり、FPGA の特徴である並列化のメリットを活用して、処理を同時実行し、全体のレイテンシを小さくすることができます。ただし、この場合は追加リソースが必要になるデメリットがあります。並列処理はクリティカルパスを短縮することもできます。これによって、パイプラインレジスタや遅延の追加クロックサイクルを使用せずに、高いクロックレートを実現できます。

並列化を行う際、コードを大幅に変更しなければならない場合があります。そのため、並列化はアルゴリズムの特性によっては困難だったり、不可能だったりすることがあります。残念ながら、アルゴリズムの並列化を必ず成功させる方法はありません。ただし、採用できる一般的な並列化パターンがあります。

データが I/O チャンネルなどの独立したストリームから構成されている場合、最大化、最小化、平均化といった演算は、ブロックダイアグラム上でコードを複製して、独立した要素/チャンネルの数と一致させることで、簡単に並列化できます。

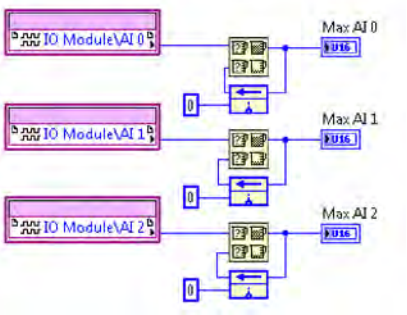

図 58. 独立したデータ要素に対して複数の同時演算を実行して、FPGA デバイス固有の並列処理のメリットを活用できる。

演算の特性が、1 つのデータセット内のサブセットに対して実行できるものである場合、この演算を何回も複製して、同じ割合でレイテンシを小さくできます。例えば、画像データに対してしきい値処理を適用すると、全てのピクセルに対して並列処理を行うことができます。コードを n 回複製すると、画像のしきい値処理のレイテンシは n 倍小さくなります。

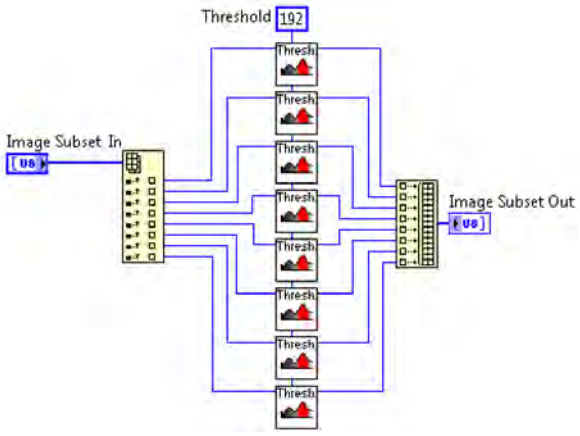

図 59. 特定の演算によって、同じデータストリームの複数部分を独立して処理することができる。このコードでは、しきい値処理を 8 つのシーケンシャルな画像ピクセルに対して適用するため、スループットが 8 倍向上し、レイテンシが 8 倍小さくなる。

一見、データを分離しようがなく、演算を複製できない場合でも、部分的に並列化することなら可能です。乗算や加算など、結合的な性質（演算順序を変更しても結果が変わらない性質）がある演算を使って、複数の項目を統合する場合、ツリー構造を作成して、組み合わせパスの長さを短縮することができます。図 60 の例では、配列の要素の加算がツリー構造を使うことでより速く処理できることを示しています。

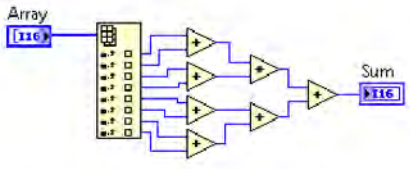

図 60. ツリー構造によって、対数的な加速がもたらされる。n 個の要素の処理に必要なステージ数は、およそ $\log_2(n)$である。これは、n 組を連続処理する方法よりも優れている。

同様のパターンを適用して、ある数の集合の最小値と最大値を判断することもできます。これもまた結合的な性質を利用した演算です。図 61 に示したように、2 つのツリー構造を部分的に組み合わせたものになります。

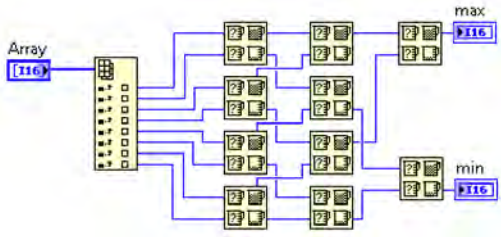

図 61. このダイアグラムは、2 つの数のうち大きいほうを選択することで、小さいほうの数も特定できるという考え方を利用したもので、複数の目的に対して同じロジック関数（最大と最小）を使用する。二重のツリー構造により、要素の数に対するして対数を取った値の倍数で、演算のレイテンシが小さくなる。

このツリーパターンをさらに一般化して、格子構造にすることができます。それは、以下の図 62 に示したように、「バブル」ソートアルゴリズムを用いて配列要素をソートしたいときなど、演算が複数の値を生成する場合です。

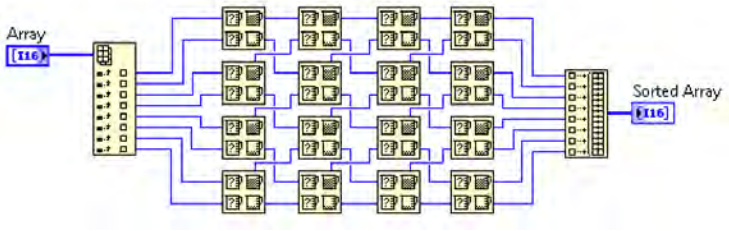

図 62. このバブルソートアルゴリズムを格子構造に当てはめることで、部分的な並列処理と演算の結合規則を活用して、1 つのクロックサイクルで固定サイズの処理を行う。

このパターンでは、「シングルサイクルタイミングループを使った高性能プログラミング」で示した最初の配列ソートアルゴリズムよりも大幅に演算が速くなります。最初の配列ソートアルゴリズムでは、n 個の要素の配列を最大 n 回通過して、データをソートする必要がありました。しかし、図 62 では、クロックレートと配列のサイズに依存して、1 つの SCTL サイクルで実行できます。

7.3. パイプラインレジスタを削除する

「スループットの最適化テクニック」の章で説明したとおり、パイプライン処理は、スループットを上げますが、代わりにレイテンシが大きくなるというデメリットがあります。その結果、重要なレイテンシパスで任意のパイプラインレジスタを削除することで、レイテンシを小さくすることができます。関数によって黙示的に追加されたパイプライン処理ステージがあるかどうかを確認する必要もあります。例えば、高スループット FPGA パレットにあるようなものです。

7.4. データタイプの最適化

「**リソースの最適化テクニック**」の章で述べた推奨方法は、伝播遅延の縮小に大きな影響を与える場合があります。これによって、少ないパイプライン処理ステージで SCTL クロックレートを上げ、全体のレイテンシを下げることができます。

8. リソースの最適化テクニック

FPGA は、性能と論理要素の数の観点からはムーアの法則に沿ってきましたが、リソースのこととなると、いまだ相対的に制約があります。FPGA のリソースは注意深く管理する必要があります。その理由は、設計が適切でなくなる可能性があるからだけではなく、達成できるクロックレートと性能に大きな影響を与える可能性があるためです。

8.1. FPGA のリソースタイプ

FPGA を構成する異なるコンポーネントを知ることで、設計が進むにつれてリソースがどのように割り当てられ、最適化するにはどのような選択肢があるのかがわかりやすくなります。FPGA は主に 2 種類のコンポーネントから構成されています。構成可能な論理ブロック(スライス)と、I/O やブロック RAM などの専用機能を持ったリソースです。

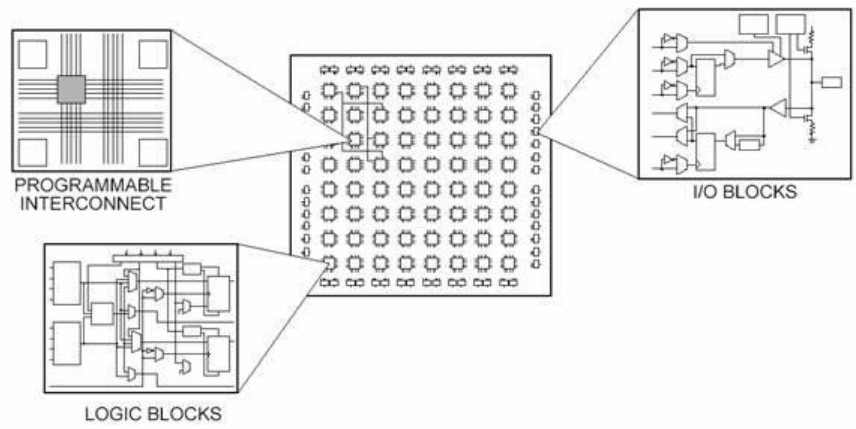

図 63. FPGA リソースは、論理ブロック(スライス)と、I/O やブロック RAM などの専用ブロックにカテゴリ分けできる。プログラム可能な内部配線がこれらのブロック間で信号をルーティングする。

論理リソースはスライスごとにグループ化され、論理関数を実行できる構成可能なブロックを作成します。1 つのスライスには LUT、フリップフロップ回路、マルチプレクサの集合が含まれています。

- LUT は、FPGA に接続された論理ゲートの集合です。LUT は、あらゆる入力の組み合わせに対する出力をあらかじめ定義したリストを保存しており、論理演算の出力をすばやく得る手段を提供します。
- MUX とも表記するマルチプレクサは、複数の入力信号から 1 つを選択して出力する回路です。
- フリップフロップ回路は、2 つの安定した状態を保持できる回路で、1 ビットを表します。フリップフロップ回路は、レジスタに一種であり、ダイアグラム内の数値など、ビットパターンを保存することができます。FPGA 上のレジスタには、クロック、入力データ、出力データ、有効な信号ポートがあります。全てのクロックサイクルで、入力値はラッチされて内部保存されます。出力は内部で保存されたデータと一致するように更新されます。FPGA VI は、レジスタを使って、フィードバック、ループシフトレジスタなど、多数の LabVIEW 構成概念を実装します。

FPGA の種類によって、スライスと LUT の実装のされ方は異なります。例えば、Virtex-II FPGA のスライスには 2 つの LUT と 2 つのフリップフロップ回路が含まれますが、Virtex-5 FPGA のスライスには、4 つの LUT と 4 つのフリップフロップ回路が含まれます。LUT に対する入力の数は、一般的には 2～6 個ですが、これもまた FPGA の種類によって異なります。

専用の FPGA リソースは、非常に特定された関数を実行します。例えば、DSP、クロック管理、大規模なストレージ要素（ブロック RAM）、I/O などです。これらのリソースは通常少なく、FPGA の種類によって数百から数万にもなるため、注意して管理する必要があります。

ブロックランダムアクセスメモリ（ブロック RAM）は、FPGA 全体に組み込まれており、データを保存します。LabVIEW がブロック RAM を使おうとするのは、メモリと FIFO を統合するときです。メモリと FIFO については、「**データ転送メカニズム**」の章で詳しく説明しています。FPGA の処理能力が限界に近づいたら、LabVIEW がこうした項目を実装するのに使用するリソースの種類を指定して、設計リソースのバランスをマニュアルで整えることができます。

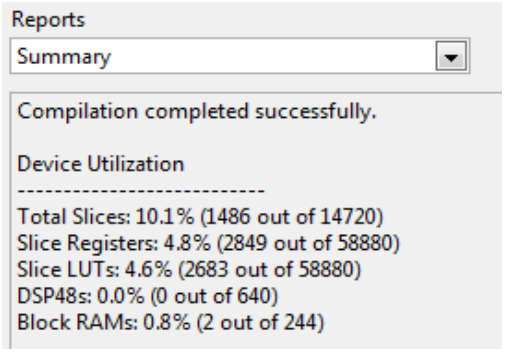

図 64. LabVIEW FPGA コンパイル結果は、リソースの種類と設計で使用されているパーセンテージを示す。

8.2. 高密度の FPGA

設計に論理を追加すると、次の理由からコンパイルができないことがあります。

- FPGA に専用のリソースタイプが不足する。
- コンパイラがコンポーネント間のルートを作成する手段を見つけられない。
- 伝播および論理遅延によって、要求されたクロックレートでの設計コンパイルができない。

FPGA のリソースは、リソースの各タイプごとに数の制限があります。設計をコンパイルしようとして、何らかのタイプのリソースが不足する場合があるかもしれません。さらに、論理リソースは複数のスライスにグループ分けされています。あるスライスが部分的に使用できず、それ以外の部分が特殊な方法で構成されている場合、特定のリソースタイプを最大限に利用することは困難となります。一方で、異なるタイプにまたがってリソースのバランスを再度整え、設計を最適化することもできます。これについては、この章の後半のセクション「**リソースのバランス**」で説明します。

コンパイラの仕事は全てのコンポーネントを FPGA の構成に配置することです。設計が小規模の場合、コンパイラがリソースを適切に配置できる確率は高まります。そして、構成がタイミング要件を満たす場合には配置の効率はやや低くなる場合があります。

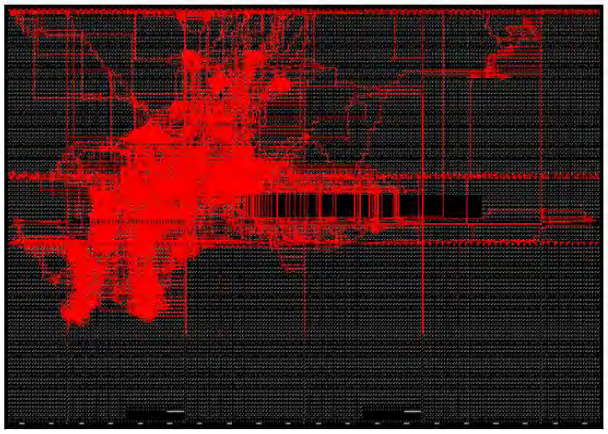

図 65. FPGA 設計のサンプルハードウェアレイアウト。規模が小さい設計はコンパイルがしやすく、要求されるタイミング制約が満たされている限り、リソースの配置については、効率が悪くなる場合があるかもしれない。

論理リソースの密度が高くなると、コンポーネントは互いに離れて配置されるようになり、ルーティングがより難しくなり、論理および伝播遅延が高まります。これにより、コンパイル時間が長くなり、コンパイルできない確率が高まります。以前は同じクロックレートでコンパイルに成功した部分にまで同じ現象が生じます。

図 66. FPGA の密度が高くなると、設計コンポーネントの配置と短いルートを見つけることが難しくなり、タイミングエラーにつながる。

リソースを最適化すれば、コンパイラは設計の要素同士をつなぐルートを見つけやすくなり、よりよいルーティングができるようになります。これにより、コンパイルが可能になります。リソースの最適化とバランス調整を行うことで、クロックレートを上げることもできるようになります。これはより短いルートで、ルーティング遅延が小さくなるためです。次のセクションでは、FPGA を最大限に利用できる、リソースの削減とリソースのバランス調整のヒントをご紹介します。

8.3. データタイプによってリソースを最適化する

スカラタイプの最適化

FPGA で値を表すために使用するビット数（「データタイプの幅」と呼ばれる場合もある）は、LabVIEW FPGA アプリケーションでリソース使用率と性能を向上する最大要素です。幅のあるデータタイプは、設計全体で値を維持してルーティ

ングするために、それだけ大きな回路が必要になります。また、そういったデータタイプはリソースの消費も速くなり、FPGA 全体に各値を伝播するのに使用する各トレースのタイミングに同時対応するのも難しくなります。

ほとんどの LabVIEW FPGA 関数は、入出力のビット数と比例するリソースを必要とします。「和」関数の場合、入力が大きいと、関数が必要とするリソースも多くなります。リソースが多いと、伝播遅延の論理遅延要素が長くなります。これによって、全体の最大クロックレートが制限されます。

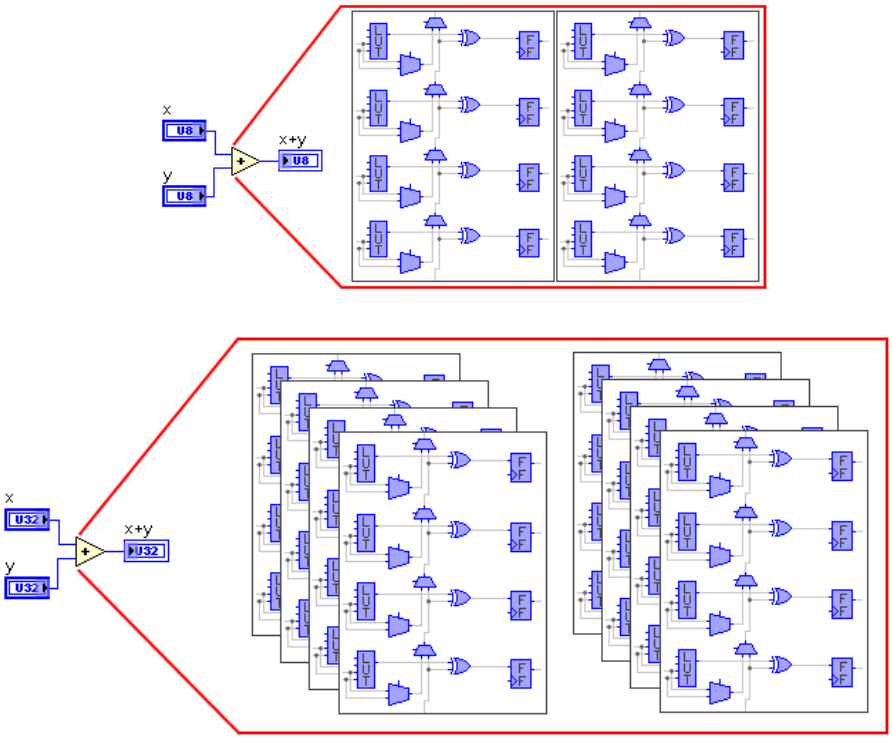

図 67. 単純な 8 ビット「和」関数の実装には、8 つの LUT（加算されるビットの各ペアに 1 つずつ使用）と、オーバーフロー（桁上がり）に対応する専用のキャリーチェーン論理が必要。FPGA デバイスの種類の中には、これらの LUT が 2 つのスライスにわたって分散されている場合がある。32 ビットの「和」関数は 4 倍のリソースを使用する。

LUT を使用して実装する関数の多くがこのモデルに従っています。例えば、減算は 2 の補数の追加として実装します。「比較」関数もまた、入力の幅が増えると、より多くのリソースを必要とするようになります。

スライス内の LUT は、FPGA の再構成可能な論理のほとんどを構成しており、様々な演算の実装に使用することができます。論理スライスが比較的多くても、スライス間のルーティングは、FPGA の密度が高くなるにつれ、難しくなります。つまり、デバイスにあるスライスを全て使用することは物理的に不可能です。

関数の中には、LUT だけでは実装できないものもあります。対数計算は複数の記録リソースを必要とし、乗算は DSP スライスを使用します。スライスと専用の FPGA 要素の比率は関数によって異なり、FPGA のリソースレイアウトに影響します。その結果、設計を FPGA 構成に実装したとき、一部のリソースタイプが他のリソースタイプよりも速く不足する場合があります。

データタイプの最適化は、設計で使用するデータやその他の値に適用して、状態を表したり、メッセージとして渡したりします。例えば、符号なし 8 ビット整数（U8）を使用して、ケースストラクチャから選択できるケースは、256 種類です。LabVIEW のデフォルトタイプである I32 を使用すると、必要以上のビット数になり、リソースを無駄にしてしまいます。

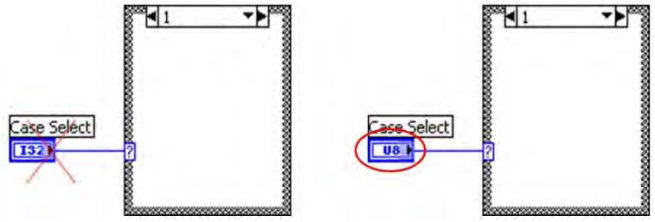

図 68. リソースを節約するには、設計の各部分に必要とされる異なる値を表すことのできる最小データタイプを選択する。ケースストラクチャのセレクタは、デフォルトで I32 になるが、ほとんどのケースストラクチャで、必要なケースはほんの少しであるため、U8 で十分である。

固定小数点タイプでのデータタイプのビット幅の増加

固定小数点データタイプには、浮動小数点データタイプの柔軟性が全くないわけではないのですが、整数演算のサイズと速度のメリットが保たれています。固定小数点数は、ユーザが指定した範囲内で、ユーザ指定の精度で表された有理数を示します。固定小数点数は、1～64 ビットの範囲の任意のサイズで指定できます。符号付きまたは符号なしのどちらかで、固定小数点数を構成できます。

固定小数点タイプに対応する数論的関数のほとんどは、入力のビット幅を基にして自動的に出力のビット幅を決定します。関数の出力のビット幅を決定するとき、LabVIEW は、オーバーフローを防ぎ、精度を保つために保守的な手法を取ります。つまり、整数と分数部分を表すのに使用するビット数を増やすという方法です。演算上はこれで問題ないのですが、後続のデータタイプのビット幅は急増し、結果としてリソースの使用率が高まります。

処理コードを流れるデータの範囲と精度がわかっている場合、処理コードのパスに沿った関数の出力のビット幅をマニュアルで指定すれば、固定小数点のビット幅の増長を抑えることができます。これには慎重な確認が必要とされるため、時間のかかる作業になる可能性があります。NI が推奨するのは、アルゴリズムを設計して、総合的なユニットテストを使用することで機能を検証する方法です。基本的なアルゴリズムが幅広い入力値に対応すれば、関数の出力のビット幅を微調整し、関数テストを使用して演算の挙動を検証することで、比較的容易に所望の出力のビット幅をマニュアル指定することができます。

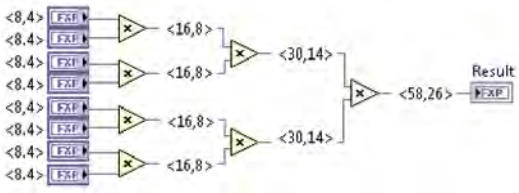

図 69. 固定小数点の出力幅は演算のタイプに応じて急増する可能性がある。この場合、LabVIEW はデータの幅を自動的に伝播し、8 ビットのオペランドから 58 ビットの結果へと拡大する。関数の出力幅をオーバーライドすることで、この増加をマニュアルで抑えることができる。

配列とクラスタの最適化

配列演算では、配列のサイズに直接比例した形でリソースが使用されます。例えば、配列をスカラ値で増分する場合、加算は配列の各要素のインスタンスを作成します。同じ原理がクラスタとより大きなデータタイプに適用されます。つまり、配列とクラスタタイプのサイズを最小化することで、FPGA のリソースを節約し、性能を向上させることができます。

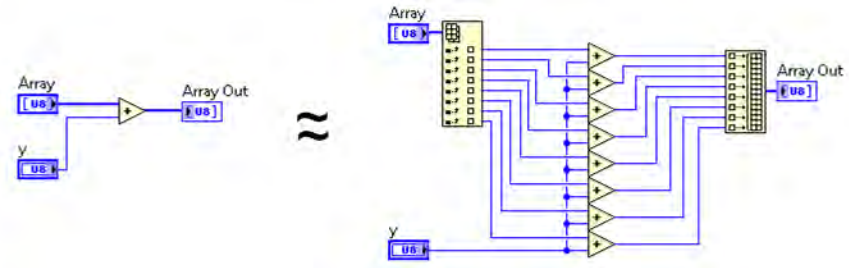

図 70. 配列演算は、配列の各要素の演算を複製することで、実装される。これによって、FPGA のリソース使用率が大幅に増加する場合がある。

演算の中には、コンパイルが FPGA のルーティングのみに帰着するものもあります。定数値を使った配列指標はコンパイラによって最適化されて、ストレージ要素と、保存された値を使用する論理回路との間のルートとなります。配列内の他の要素は、使用されない場合、コンパイラによって削除される可能性があります。

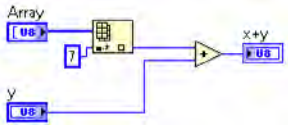

図 17. この例では、配列入力の指標 7 に数字を加算している。指標 7 の取得に関連付けられたスライス論理はない。指標 7 を y に追加するときのみ論理が必要になる。

制御器を指標配列関数に接続するには、論理を追加して、選択した配列要素にルーティングする必要があります。指標の値が実行時に定数であっても、コンパイラは、実行時に指標の値が変化する場合に対処する回路を挿入する必要があります。

8.4. フロントパネルの制御器と表示器の使用を最小限に抑える

FPGA のリソースの使用を最適化する場合、トップレベルの FPGA VI のフロントパネルに作成する制御器と表示器の数を最小限にする必要があります。「**データ転送メカニズム**」セクションで述べたように、トップレベルの FPGA VI の各フロントパネルオブジェクトは、FPGA とホストシステムとの間の通信のレジスタ項目としての役割を果たします。これには、ストレージとアドレス論理が必要です。したがって、未使用のフロントパネルオブジェクトを削除するか、ダイアグラムの定数と置き換える必要があります。

大規模な配列またはクラスタをフロントパネルの表示器または制御器として作成することも、リソースの使用率を上げてしまうため、回避するべきです。「**データ転送メカニズム**」章で説明したように、12 個以上の配列要素をホストとの間で転送する必要がある場合、FPGA ホストインタフェース FIFO を使用します。フロントパネルオブジェクトが、VI 内でデータを転送する手段として使用されている場合、グローバル変数と置き換えて、リソースを節約することができます。

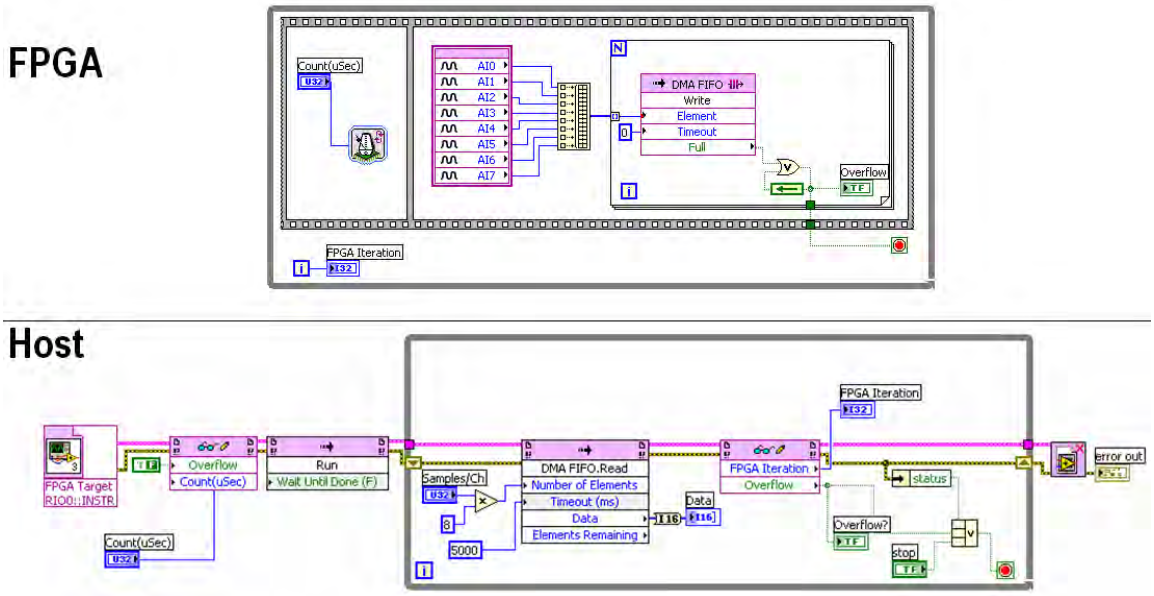

図 72. この例は、ホストインタフェース DMA FIFO を使って、アナログサンプルの配列をホストへ転送しているところ。

8.5. 出力のオーバーフローおよび丸め込みオプションの微調整

オーバーフローオプション

「高スループット IP の統合」の章で説明したように、オーバーフロー状態は、出力端子のタイプが、演算結果を保持するのに十分な大きさがない場合に発生します。一般的に、オーバーフローは通常望ましくないため、LabVIEW は、オーバーフローを回避するような出力タイプの幅を適用します。しかし、アルゴリズムと処理するデータの知識を生かし、小さい出力タイプを使って、FPGA リソースの使用率を減らすことができます。マニュアルで出力タイプを指定する場合、ほとんどの関数は、「飽和」または「切り捨て」モードを使ってオーバーフローを処理できるように構成できます。

「飽和」モードでは、値が出力タイプの範囲内かどうかが判断され、適切な制限値に強制変換されます。これには、回路がオーバーフロー状態を検出して、出力の適切なオーバーフロー値を選択する必要があります。

「切り捨て」モードでは、値が出力範囲内になるまで、単純に有効ビットが破棄されます。このモードに必要な FPGA リソースは少なくてすみますが、オーバーフロー状態が発生した場合、数値に大きな差が出る可能性があります。

標準的な固定小数点の演算操作の出力タイプをマニュアルで指定する場合、LabVIEW はデフォルトで「飽和」オーバーフローモードになり、数値の正確さのほうが優先されます。「高スループット IP の統合」の章で述べた高スループット数学関数は、「切り捨て」オーバーフローモードがデフォルトで、性能とリソースの効率のほうが優先されます。オーバーフローは関数ごとに調整して、リソース使用率と数値の正確さとのバランスを取ることができます。

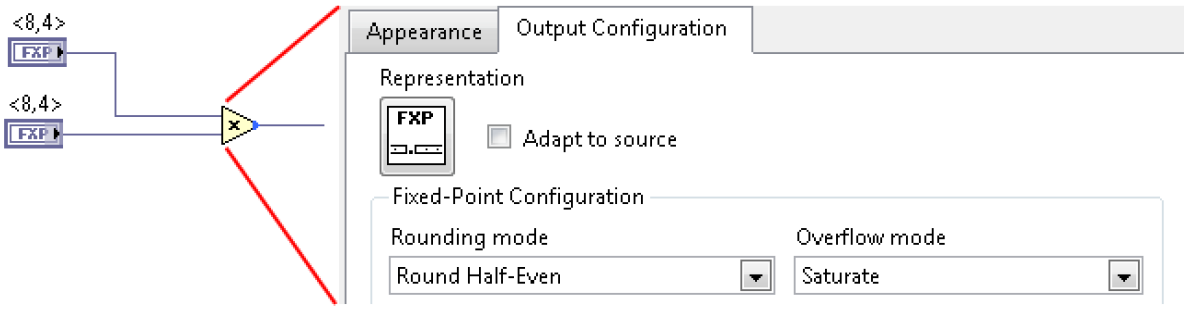

図 73. 出力タイプの幅をマニュアルで指定する場合、青色の点が標準の固定小数点の演算関数の出力端子に表示される。デフォルトで「飽和」オーバーフローモード、「四捨五入−最下位ビットを特定」丸め込みモードになっており、数値の正確さが優先される。

丸め込みオプション

丸め込みが起こるのは、予測された値の正確さが、値を表すタイプの正確さを上回った場合です。LabVIEW は、数値の正確さを維持する出力の幅を自動的に選択します。割り算や平方根といった不正確な演算子は常に何らかの丸め込みを必要とします。丸め込みを実行するように関数を構成するには、「切り下げ」、「四捨五入 − 中間値は切り上げ」、または「四捨五入−最下位ビットを特定」モードを使います。

「切り下げ」モードは、単純に最下位ビットを削除するため、FPGA リソースは必要としません。しかし、このモードは、ほとんどのデータストリームで最大の平均誤差を生みます。その理由はゼロに向かってバイアスがかかるためです。

「四捨五入 − 中間値は切り上げ」モードは、そのタイプが表せる最も近い値に丸め込みを行います。丸め込みを行う値がちょうど 2 つの有効値の中間になる値の場合、このモードは値を 2 つの値の大きいほうに丸め込みます。LabVIEW は出力値に 1 ビットを追加してから、切り下げます。この丸め込みモードは、大きいほうの値に向かってより小さいバイアスがかかるため、「切り下げ」モードより正確な出力値を生成します。しかし、「四捨五入 − 中間値は切り上げ」は、性能とリソースにより大きな影響を与えます。その理由は、出力タイプのビット幅と比例する論理の追加を必要とするためです。

「四捨五入−最下位ビットを特定」モードは、そのタイプが表せる最も近い値に丸め込みを行います。丸め込みを行う値がちょうど 2 つの有効値の中間になる値の場合、丸め込みが行われた後、LabVIEW は、最下位ビットになる値のビットをチェックします。ビットが 0 の場合、値は出力タイプが表せる 2 つの値のうちより小さい値に丸め込まれます。ビットが 0 ではない場合、値は 2 つの値のうちより大きい値に丸め込まれます。「四捨五入−最下位ビットを特定」モードは、他のモードよりも正確な出力値を生成しますが、最も多くの FPGA リソースを必要とし、性能を最も大きく低下させます。

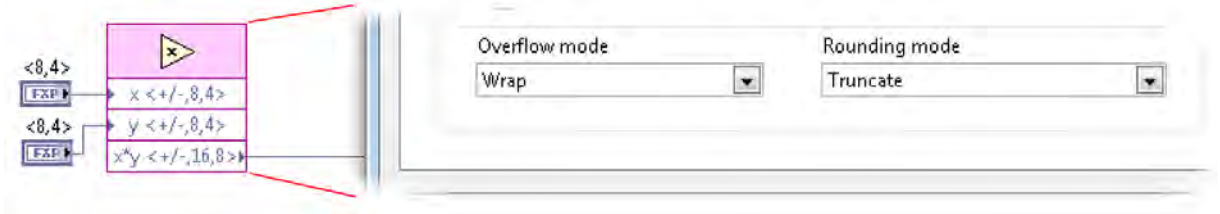

図 74. 高スループット数学関数の出力タイプの幅を指定する場合、デフォルトで「飽和」オーバーフローモードと「切り下げ」モードになり、性能とリソースの効率が優先される。

関数の出力タイプを指定する場合、LabVIEW では、標準の固定小数点の演算関数に対して、デフォルトで「四捨五入−最下位ビットを特定」モードになり、数値の正確さを優先します。「高スループット IP の統合」の章で述べた高スループット数学関数は、「切り下げ」丸め込みモードがデフォルトで、性能とリソースの効率のほうが優先されます。出力の幅と丸め込みモードは関数ごとに調整して、リソース使用率と数値の正確さとのバランスを取ることができます。

8.6. フィードバックノードの初期化

シフトレジスタは、ループストラクチャが入力されるたびに初期化されます。シフトレジスタと同様、フィードバックノードの初期値は設定可能で、初期化の仕方が性能とリソースの使用率に影響を与える場合があります。

配線されていないフィードバックノードは、デフォルトでコンパイルまたはロード時に初期化されます。「コンパイルまたはロード時に初期化」では、FPGA ビットファイルが FPGA にロードされるとすぐに、データタイプのデフォルト値がフィードバックノードの初期値として指定されます。値をフィードバックノードの初期化端子に配線すると、初期化モードは「最初の呼び出し」に切り替わります。

「最初の呼び出し」オプションが便利なのは、フィードバックノードを FPGA VI を実行するたびに元の値にリセットしたい場合です。「最初の呼び出し」での初期化では、マルチプレクサを使用して、初期値と現在の値から選択するので、リソース使用率がやや高まります。ほとんどのアプリケーションは、デバイスの再起動ごとに FPGA ビットファイルをロードして実行するので、コンパイルまたはロード時に初期化するほうがよりよい選択だと言えます。ホストアプリケーションが実行ごとに FPGA ビットファイルをロードする場合、または以前の実行時の値を再利用することが許される場合、「コンパイルまたはロード時に初期化」オプションを使用して、マルチプレクサを節約し、そのコードパスで到達可能なクロックレートをわずかに上げることができます。

図 75. フィードバックノードで「コンパイルまたはロード時に初期化」が設定されていると、使用するリソースが少なくなり、コンパイルの速度が速くなる。

「最初の呼び出し」初期化モードを使用する場合、マルチプレクサをレジスタの前と後ろのどちらに追加するのかを選択します。これは高度な初期化オプションです。クリティカルパスを遅延セグメントに分割し、コンパイルの速度を上げようとしてうまくいかないときに役立ちます。マルチプレクサをレジスタの片側に移動すると、伝播および論理遅延がセグメント間を移動します。これによって、パスの遅延を分散する場合、細かく分散することが可能になります。

8.7. リソースのバランス

リソースのバランス調整は、特定のタイプのリソースが不足してコンパイルできない場合の解決手段として役立ちます。設計の一部に特定のリソースを指定したり、使用するようにすれば、より FPGA リソースを効率的に使用できます。バランシングによって、設計をより高速に実行することも可能です。リソースタイプによって異なるクロックレートで実行でき、リソース間で必要な論理のルーティング量が異なるためです。

フィードバックノード、メモリ、FIFO、レジスタといった項目は、多くのリソースを消費することもありますが、多くの場合、不揮発性ストレージと通信タスクに使われます。これについては、「**データ転送メカニズム**」の章でより詳しく説明しています。

フィードバックノード

シフトレジスタ、およびフィードバックノードは、コンパイラによって LUT に 1 つ以上が最適化されない限り、フリップフロップ回路を使って実装されます。コードがフィードバックノードの初期値に依存せずに機能する場合、「フィードバックノード」プロパティダイアログの「FPGA 実装」タブで、「FPGA リセットメソッドを無視」オプションを選択することでリソースをさらに節約することができます。これによって、コンパイラがレジスタからリセット論理を削除して、フリップフロップ回路ではなく、シフトレジスタルックアップテーブル（SRL）を使って実装できるようになります。SRL は、複数の遅延を 1 つの LUT に統合できるため、フリップフロップ回路に関連する FPGA リソースを大幅に削減できます。

「離散遅延」関数の挙動は、フィードバックノードと似ていますが、SRL を使って複数の遅延サイクルを実装します。「離散遅延」関数は、実行時の遅延の変化に対応します。あらかじめ対応可能な最大値を設定し、整数のスカラ値および配列、固定小数点、ブールデータタイプ、クラスタ、クラスタの配列に対応します。

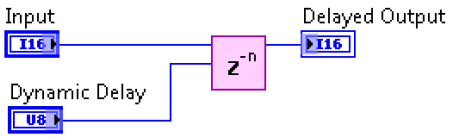

図 76.「離散遅延」ブロックはブールデータのみで動作するが、SRL リソースを使って実装されるため、フィードバックノードの代わりに使用して、リソース使用率のバランスを取ることができる。

FPGA FIFO

FPGA FIFO は、ストレージとして使用したり、FPGA のループ間でデータを受け渡しする手段として使用できます。FPGA に FIFO を実装するのに使用するリソースのタイプを指定してリソースのバランスを取る際、フリップフロップ回路、LUT、ブロックメモリ（BRAM）のうちから選択できます。

フリップフロップ回路では、パフォーマンスが最も高くなりますが、最大 100 バイトのサイズの小さい FIFO にしか使用してはいけません。ストレージ要素あたりに多くの FPGA 論理を消費するためです。

LUT はフリップフロップ回路よりもリソース使用効率が良いため、100 から 300 バイト範囲の大規模な FIFO に適しています。

ブロックメモリ FIFO は、FPGA 構成全体に分散された専用のブロックメモリを使ってデータを保存します。このオプションは、FPGA デバイスによって、300 バイト以上、最大数キロバイトの FIFO に使用できます。「**データ転送メカニズム**」の章で述べたように、ブロックメモリ FIFO は、クロック領域全体にデータを転送できる、唯一の FIFO の実装手段です。また、このオプションは他の 2 つよりもレイテンシが長くなります。

ブロックメモリ FIFO の使用時、FIFO 制御論理の実装リソースを指定することができます。新しい FPGA には、FIFO コントローラが内蔵されています。これによって、大量の FPGA リソースを節約し、スライスファブリックを使って、制御論理よりも高いクロックレートに対応できます。

配列定数

LabVIEW での配列定数の実装方法を指定すれば、FPGA アプリケーションを最適化することができます。異なる種類のメモリのメリットが必要でない限り、あるいは他のタスクのためにブロックメモリの容量を解放する必要がない限り、ブロックメモリを使った配列定数の実装を選択します。ブロックメモリは、他のメモリタイプと比べると、FPGA 論理リソースを消費せず、高いクロックレートでコンパイルする傾向があります。

LabVIEW はデフォルトで、特定のコードパターンに基づいて、ブロックメモリ、LUT、フリップフロップ回路のどれかを自動的に選択し、配列定数を実装します。フィードバックノードに保存された配列は、実装リソースとしてブロック RAM を使用する可能性があり、RAM としての扱いが可能になります。ブロック RAM を使用している場合、読み取り 1 つと書き込み 1 つのみ、STCL 内の標準の配列関数を使って、配列にアクセスできます。また、ブロック RAM へのアクセスは、1 つのクロックサイクルを丸々使うため、フィードバックノードは各アクセスの直後に配置する必要があります。

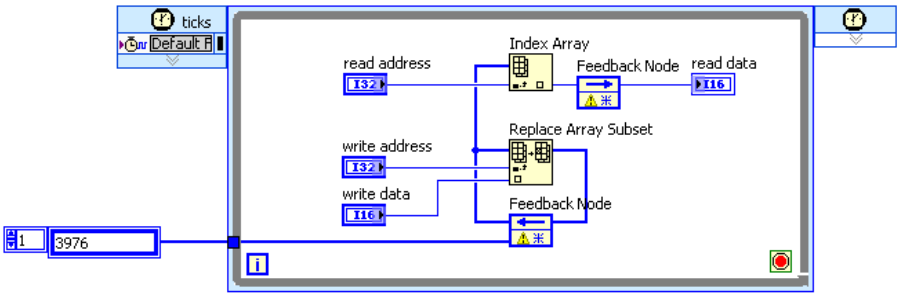

図 77. この配列定数はブロック RAM として実装され、フィードバックノードに保存される。これによって、標準の配列関数を使うだけで、SCTL の 1 サイクル内で読み取りと書き込みアクセスが可能になる。

読み取りアクセスのみ必要な場合、SCTL 内で、配列を配置した直後に「指標配列」関数と「フィードバックノード」を配置すれば、ブロック RAM を配列の実装リソースとすることができます。

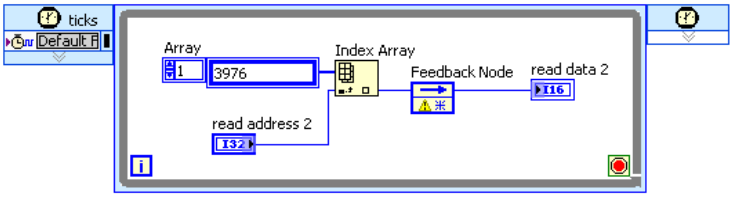

図 78. 上記パターンを使えば、読み取りのみの配列をブロック RAM で実装することもできる。

メモリ項目に対するデュアルポートアクセス

デュアルポート読み取りアクセスのメモリを「メモリプロパティ」ダイアログの「インタフェース」ページで構成し、ブロックメモリリソー

スの使用を減らし、実行時間を短縮することができます。

NATIONAL INSTRUMENTS

2 つの読み取りポートを使えば、2 つの異なるアドレスのデータに同時にアクセスできます。2 つのメモリメソッドノードを配置することができ、これらは 1 つのストラクチャ（SCTL など）、あるいは FPGA VI 内の異なる場所にあるデュアルポートメモリを読み取ります。

8.8. 論理のマルチプレクス

リソースを最適化するもう 1 つの方法として、論理の再利用があります。高いクロックレートで処理コードを実行し、時間の経過とともにアクセスをマルチプレクスすることで、同じ回路を使って、独立したデータストリームを処理することができます。SCTL を使ってこれを行う方法として、元の SCTL クロックレートの数倍のクロックレートで、処理論理を別の SCTL に移動する方法があります。ブロック RAM FIFO を使って、クロック領域全体でデータを転送してから、共通のブロックセットを使って、サンプル処理を交互で行う必要があります。

コードのマルチプレクスが意味をなすのは、比較的大きな処理タスクに対してのみです。それは、記録と通信のオーバーヘッドがあるためです。通信オーバーヘッドとは、2 つのループ間でサンプルを移動することによって生じる、レイテンシ、リソース、レートの潜在的負担のことです。記録オーバーヘッドは、追加する必要のある論理を指します。この論理を追加して、指定した時間にどのサンプルを処理する必要があるのかを追跡します。長時間ステートに依存する処理コードは、適切なステート情報を保存し、取得する必要もあります。それは、このコードが独立したデータストリームの間で交互に処理を行うためです。

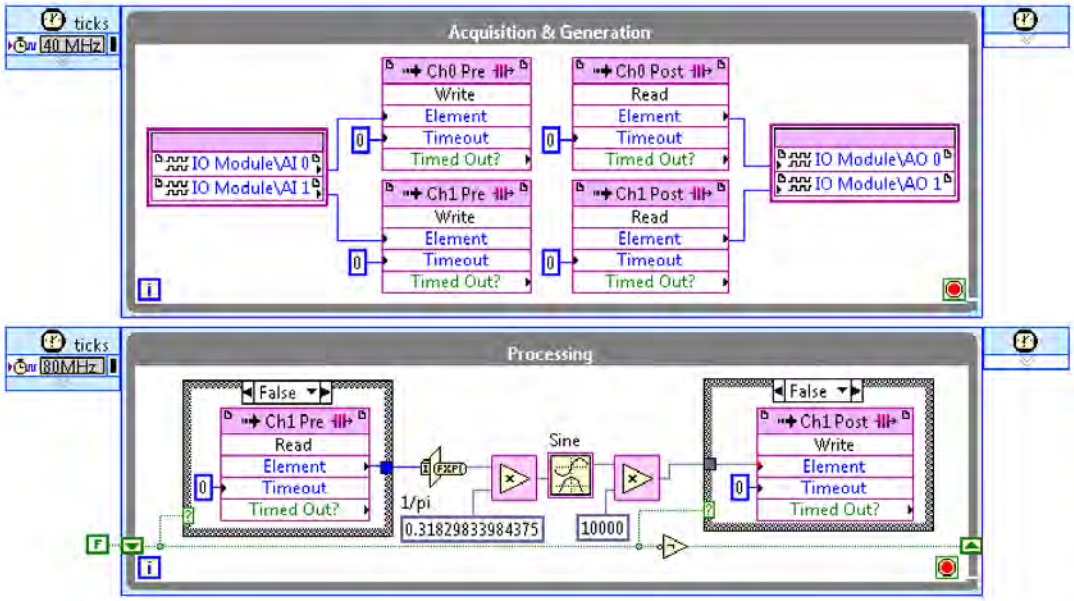

図 79. 処理コードは 2 つの独立したデータチャンネル全体で時間をかけてマルチプレクスを行う。収集ループは I/O モジュールからデータを読み取り、専用 FIFO を使って両方のサンプルを処理ループに転送する。処理ループは収集ループの 2 倍のレートで実行され、各 FIFO からの読み取り、処理の実行、適切な FIFO への結果の書き込みが交互に行われる。この手法では、処理に必要なリソースを 2 分の 1 に減らす代わりに、通信と記録のオーバーヘッドが生じる。

DSP 演算はマルチプレクスの対象として最適だと言えます。通常、デジタル論理タスクよりも多くのリソースを消費するためです。FPGA のスペースが不足している場合、除算、平方根、その他の複雑な IP のように、リソースを多く消費する演算もマルチプレクスに最適だと言えます。

Xilinx CORE Generator IP を使用する、あるいは LabVIEW FPGA IP Builder アドオンを使って独自の IP を作成する場合、浮動小数点演算も SCTL 内で行うことが可能です。単精度浮動小数点演算は多数のリソースを使用しますが、、特定のアルゴリズムの実装を簡素化できます。その結果、単精度浮動数点演算もまた、マルチプレクスに最適です。

SCTL の外で処理を行うとき、サブ VI を再入不可能に設定し、ダイアグラムに複数のコピーを配置することでサブ VI をマルチプレクスすることができます。LabVIEW が各コピーへのアクセスを調整することで、実行が直列化し、リソース使用の効率が向上します。FPGA は、多数の制御/計測アプリケーションにおいて I/O よりもはるかに高速であるため、アクセスが直列化しても、スループットおよびレイテンシの要件が満たされることが多く、同時に FPGA リソースを節約することができます。

8.9. SCTL をリソースの節約手段として使う

既存の FPGA アプリケーションに SCTL が含まれていない場合でも、リソースを最適化するために使用を検討する場合があるかもしれません。LabVIEW は SCTL 内のコードを自動的に最適化して、While ループ内の同じコードと比べて、実行を高速化し、使用する FPGA のスペースを減らします。

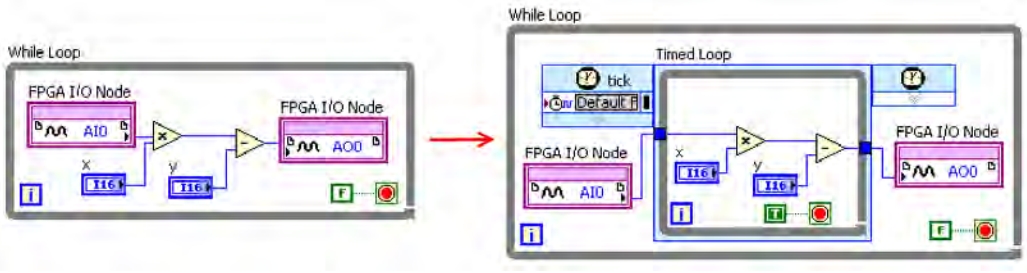

図 80. While ループ内で SCTL を使うことができる。定数を SCTL の停止条件に配線する、または内側の 4 線式ハンドシェイク IP の「出力有効」信号を使用して、ループを停止し、結果を SCTL ではない他のコードに渡す。

関連情報

[1] LabVIEW FPGA IP Builder を使用して VI を最適化・移植して FPGA で使用する（英語）
http://www.ni.com/white-paper/14036/en/

[2] 配列定数のメモリ使用率を最適化する（英語）
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_array_memory_implement/

9. データ転送メカニズム

データ転送は、高性能 LabVIEW FPGA アプリケーションの重要な側面であり、システム内で並列プロセスとして実装されます。これらの並列プロセスは、1 つの FPGA、複数の FPGA、またはホストシステムで並列ループの形をとります。システムの一般的な概念として、通信は FPGA 内、FPGA とホストとの間、FPGA デバイス間、または FPGA デバイスと他の計測器との間で行われます。

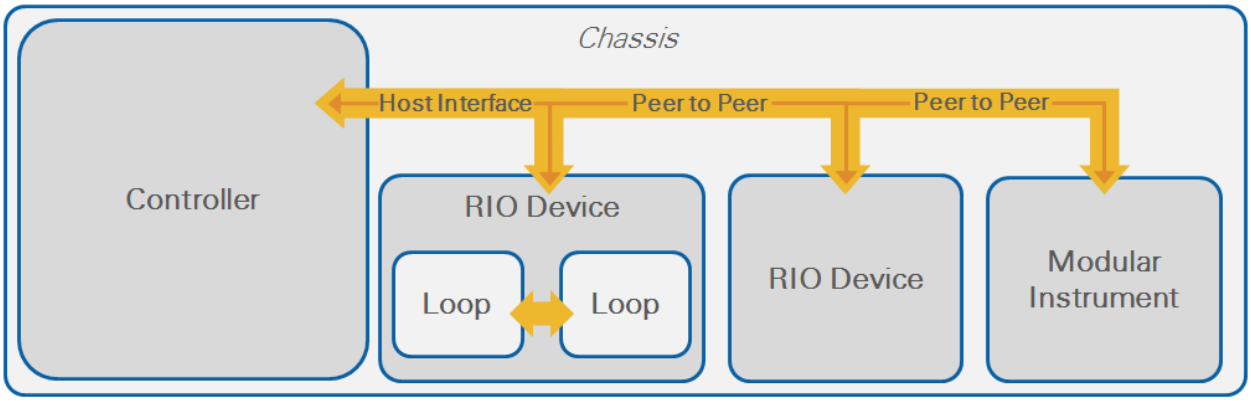

図 81. データ転送は、FPGA 内のループ間、FPGA デバイス間、FPGA デバイスと他のデバイスとの間、または FPGA デバイスとホストシステムとの間で行われる可能性がある。

この章は、異なるデータ転送メカニズムと、それらに関連したパフォーマンス特性の概要について述べます。

9.1. データ転送メカニズムのスループットとレイテンシ

アプリケーションが複数のデバイスにまたがる場合、またはデバイス上の複数の処理にまたがる場合、データ転送メカニズムの効率は、アプリケーションの性能に影響を及ぼします。このセクションでは、この後のセクションで取り上げるデータ転送メカニズムの性能を解析する概念について説明します。

どのようなデータ転送メカニズムも、2 つのプロセス（ソースとシンク）間でのやり取りとしてモデル化できます。そのやり取りとは、データがチャンネルを介してソースからシンクへ送られることです。

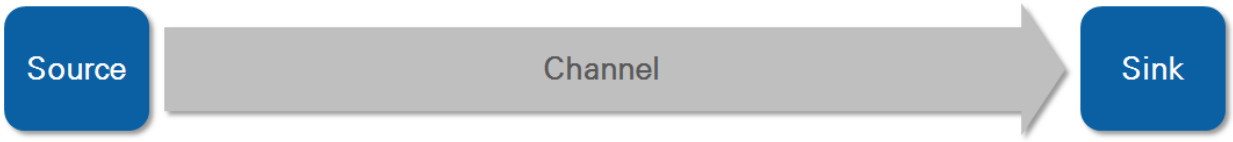

図 82. データ転送とは、チャンネルを介してデータをやり取りする 2 つのピアとしてモデル化できる。ソースがデータを生成し、シンクがそのデータを処理する。

ソース、シンク、およびチャンネルによって、データ転送メカニズムの効率の良いスループットが決まります。ソースは、単位時間あたりに供給できるデータ量を決定します。チャンネルは、データが通過できるだけの十分な帯域幅を備えている必要があります。シンクはデータが到達した際に保存、または処理できる必要があります。

帯域幅は、任意のチャンネル構成における、実現可能な最大スループットです。帯域幅がスループットと異なる点は、スループットが、チャンネルを通過する実際のデータ量を指す点においてです。チャンネル帯域幅にほぼ等しいスループットレートを実現するには、特定の転送パターンを使うか、特定のサイズのデータを書き込む必要がある場合があります。

データソースは、断続的に、一気にデータを転送する可能性があります。バースト（データをまとめて一気に転送する方式）パターンが予測可能な場合、チャンネルがデータ転送で占有されている時間の割合として、デューティーサイクルを定義することができます。また、最小転送サイズというものがあり、これは、転送開始前に必要な最小限のデータ量として、ソースまたはチャンネルによって定義されます。

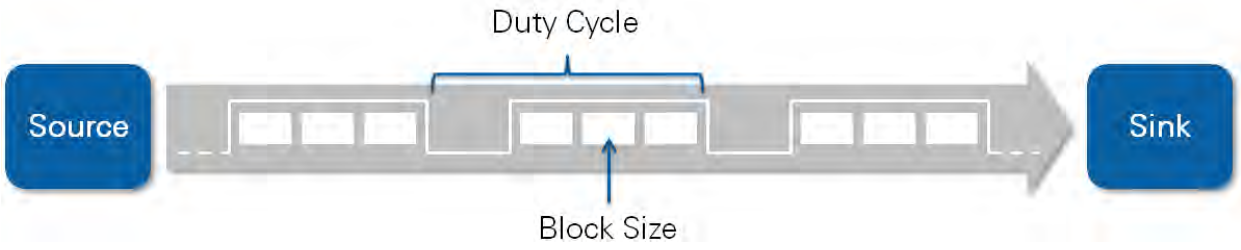

図 83. データのバースト転送は、1 つ以上のブロックから構成できる。バーストが均一かつ周期的な場合、チャンネルの効率的なスループットは、チャンネル帯域幅にデューティーサイクルを乗算したものである。

データのバースト、予測不能な転送パターン、その他の一時的なチャンネル/ソース条件（バスのコリジョン、ジッタなど）が、ピークおよび平均スループットを決める要因となります。平均スループットは、無限の時間を通して、維持できるスループットとして定義されます。ソース、シンク、またはチャンネルが、要求された平均スループットで転送を達成できない場合、転送は最終的に失敗します。ピークスループットは一時的には平均スループットを上回ることがありますが、チャンネル帯域幅を上回ることはできません。

データ転送が成功するのは、シンクがピークスループットレートでデータに対応できる場合、あるいは、平均スループットレートで、バッファを使って一時的にデータを保存し、バッチ処理する場合のみです。

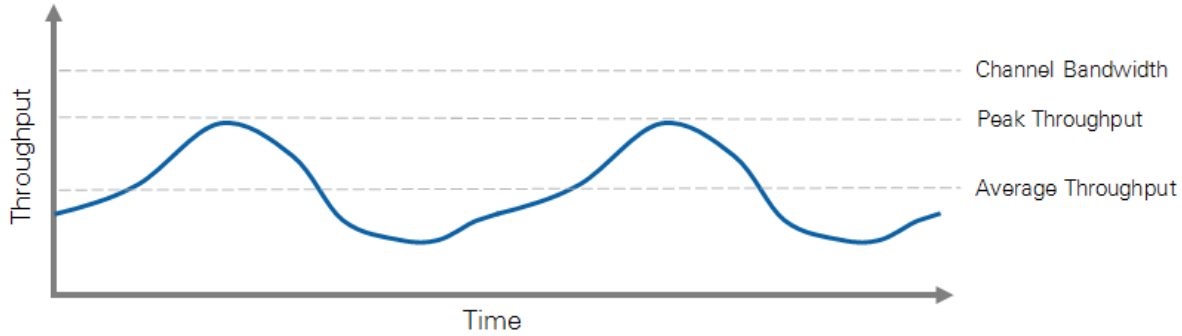

図 83. 効率的なスループットは時間が経つと変化する可能性があるが、チャンネル帯域幅を上回ることはできない。ソースとシンクは平均スループットで転送を行い、転送を無限に維持する必要がある。ソースとシンクは一時的なピークスループットの状態にも対応する。これには通常バッファを使う。

バッファリングによって、予測されるデータのバーストや予測できない過渡条件が吸収されることによって、求められた平均ス

ループットを維持することができます。より大きなバッファは、より大きな過渡的イベントを吸収することができますが、その場合、リソースの使用率とレイテンシが高くなるというデメリットがあります。

図 85. バッファリングは、ソースまたはシンクに一時的にデータを保存することで、過渡的条件に対処する。

データ転送のレイテンシは、転送中のデータ量に通信チャンネルのレイテンシを加えたものに直接比例します。PCI Express などの FPGA 内通信およびローカル通信バスは通常高速で、いかなるときでもチャンネルに多くのデータを保持しません。その結果、ほとんどのレイテンシは、ソースおよびシンクのバッファにあるデータから生じます。レイテンシは、予測される転送中のデータ量を通信メカニズムの平均スループットで乗算することで算出することができます。

9.2. FPGA 内でデータを転送する

マルチループおよびマルチレート設計

高性能な LabVIEW アプリケーションをダイアグラム上で複数のループに分散させることはよくあります。複数のループに分散させる場合、次のような点を考慮します。

タスク分割—優れたプログラミング例では、可能な場合、独立したループに分割タスクを論理的にマッピングします。そうすることで、コードのメンテナンスと可読性の負担が軽くなります。このように設計した場合、ループは独立したプロセスの役割を担うことになり、FPGA の並列処理を活用して、コンカレント処理を行うことができます。複数の処理ループが同じデータストリームに働きかけることがあるため、ループ間でのデータのやり取りによるパフォーマンスへの影響を理解することが重要です。

強制クロック領域—特定の FPGA 構成概念は、特定のクロック領域でしか使用できないため、アプリケーションの一部は強制的に特定のクロック領域に割り当てられます。通常これが当てはまるのは、アプリケーションが I/O や DRAM など、FPGA の外部コンポーネントとやり取りをする場合です。データを高速入力ソースから読み取って、DRAM に転送するアプリケーションは、強制的に最低 2 つのループを持つことになります。1 つは入力用で、もう 1 つは DRAM への書き込み用です。そして、これにはループ間の通信メカニズムが必要とされます。

クロック領域の最適化—特定の I/O 構成概念によって特定のクロック領域を使用しなければならないとき、残りの処理タスクが他のクロック領域に配置される可能性があることを知っておく必要があります。たとえば、データストリームを平均化またはデシメーションした後、より低いスループットのストリームで対応できるようになります。つまり、後続の関数をより低いレートでクロックできるようになる場合があります。レートが低くなると、コンパイル時間が短縮され、リソースの使用率も低くなり、コンパイルが成功する可能性が高まります。

「**リソースの最適化テクニック**」章では、この手法の例を取り上げています。リソースを節約するため、処理関数は、別々のループで実行されることで、経時的に共有されます。

異なる種類のループ—最終的には、FPGA 設計全体で異なる種類のループを使用することになる場合があります。アプリケーションの全てが SCTL のパフォーマンスのメリットを必要とはしないためです。結果として、アプリケーションを複数のループに分割して、SCTL の比較的厳しい要件を回避する場合もあるでしょう。

アプリケーションを複数のループに分割することで、データ通信に問題が生じる場合があります。通信する SCTL 内の論理が異なるクロックを使用する場合、その論理は異なるクロック領域で実行されます。SCTL が異なるクロック領域を定義する場合、通信メカニズムの選択と構成に注意を払わないと、データが失われたり、上書きされたりする可能性があります。

通信メカニズムによって、データのコヒーレンス維持に対する、同期およびバッファリングのポリシーの度合いは異なります。パフォーマンスやリソースの使用率にデメリットが生じる場合も多々あります。データのタギングや同期変数など、同期メカニズムを追加すれば、ループ間における実行順序を保証したり、必要に応じて通信メカニズムを強化することができます。

ループ間の通信メカニズム

ローカルおよびグローバル変数

変数はループ間で情報をブロードキャストします。変数はバッファリングされないため、最も新しく書き込まれた値のみが必要な場合、便利です。SCTL で変数を使用する場合、書き込みは 1 つに制限されますが、読み取りは複数許可されます。

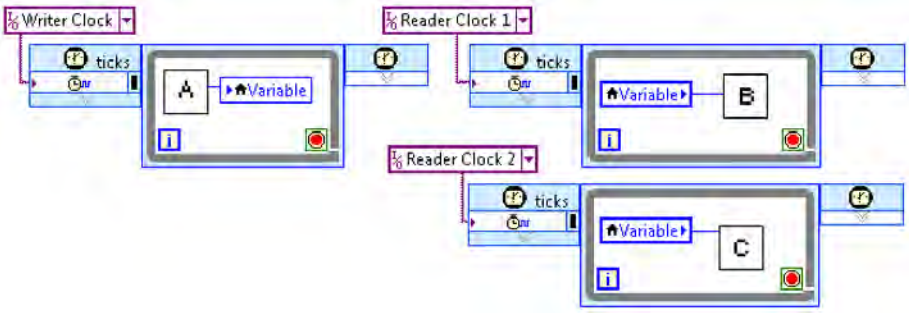

図 86. 変数を使って、ループ全体で値を保存し、ネットワーク共有する

変数のアクセス範囲は、変数の種類によって異なります。ローカル変数は、それらを定義する VI 内だけでアクセス可能です。グローバル変数は、FPGA 上で実行される任意の VI からアクセス可能です。ローカル変数は、フロントパネル上に制御器または表示器の形で表示されます。その結果、「**リソースの最適化テクニック**」章で述べたとおり、リソースの節約が必要な場合、グローバル変数またはレジスタがより適した選択肢になります。

データのコヒーレンスは、ある 1 つの変数の全てのビットに対して保証されます。言い換えれば、書き込まれた値は、レシーバによって読み取られる際、全てのビットに対してコヒーレントだということです。複数の変数間でのコヒーレンスは保証されないため、同時に複数の変数の値が重要な場合、変数をクラスタの一部としてバンドル化するか、同期メカニズムを重複させる必要があります。変数を使用するけれども、書き込みと読み取りを同期する必要がある場合、この章の「エラー! 基準ソースがありません」で述べているとおり、ハンドシェイク項目の使用をご検討ください。

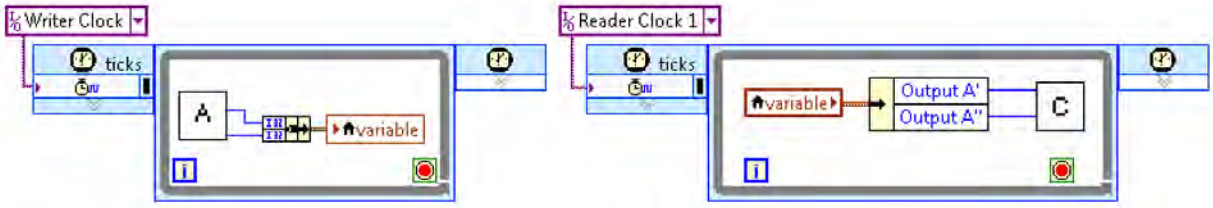

図 87. この例では、複数の値をループ間で共有する際、クラスタ変数を使用して、コヒーレンスを保証している。

レイテンシに関して言えば、変数はいくつかのサイクルを経て、書き込みから読み取りへ値を伝播する場合があり、サイクルの数は読み取り間で異なる可能性があります。つまり、常に複数の読み取りで同じ変数を共有する必要がある場合、同期またはタギングを追加する形で、適切な手順を踏む必要があります。また、値がループ間で転送される際にどれだけの時間がかかるかわからないため、コードに計測機能を施したり、いつデータに対して書き込みまたは読み取りが行われたのかを検出する同期メカニズムを追加したりする必要があります。

変数は内蔵バッファリングや同期メカニズムを提供しないため、通常はデータ転送には採用されません。その結果、スループット特性はレイテンシほど意味を持ちません。

レジスタ項目

レジスタ項目は、機能、リソース使用、パフォーマンスにおいて、変数と類似しています。変数とは対照的に、VI 定義レジスタ構成ノードを使って、レジスタ項目のリファレンスを取得できます。レジスタ項目のリファレンスを使って、再利用可能なコードを書き込むには、サブ VI にリファレンスワイヤを渡します。レジスタ項目に対するこれらのリファレンスはコンパイル時に終了します。

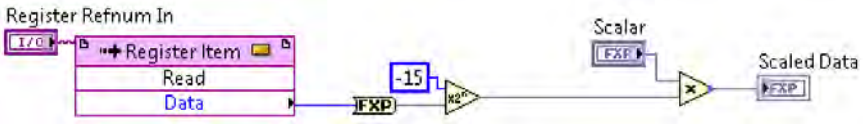

図 88. レジスタリファレンスワイヤを使って、共有データを処理する、再利用可能なサブ VI を書き込むことができる。同じサブ VI を設計全体を通して使用できるが、呼び出し VI 側で関連付けられたリファレンスの値によって、異なるレジスタからデータを読み取ることができる。

FIFO 項目

FIFO を使えば、先入れ先出しのアクセスポリシーにのっとった、ループ間の高スループット無損失転送が可能です。FIFO はデータをバッファリングします。ソースまたはシンクがデータのバーストを生成したり、受け入れたりする可能性のある場合に便利な機能です。

FIFO はプロジェクトまたは VI によって定義することができ、類似したスループットおよびレイテンシ特性を備えています。プロジェクトで定義した FIFO は異なる VI のループ間でデータを転送します。複数の VI 上で共通に使用可能なグローバル FIFO の定義と構成パラメータを設定する場合は、LabVIEW プロジェクト上で行います。。

VI で定義される FIFO は、VI 定義 FIFO 構成ノードを使って、ダイアグラムで直接定義されます。このノードは、ダイアグラムで親 VI のインスタンスが作成されるごとに、FIFO のインスタンスを作成します。構成ノードは、再利用コードを記述し、VI 間でデータを転送する際に使用できるリファレンスも提供します。

FIFO には、「タイムアウト?」端子があり、「Read」または「Write」メソッドの成功を示します。読み取り時のタイムアウトは、FIFO が空で、要素の値が定義されていないことを示します。したがって、値を後続のノードで使用することはできません。「高スループット IP を統合する」章で述べたとおり、この信号は 4 線式プロトコルの反転した「output valid」に相当します。

図 89. SCTL 内の FIFO 読み取り操作では、ゼロのタイムアウト値を使用する必要がある。「Time Out?」出力は空の FIFO を示す。

書き込み時のタイムアウトは、FIFO が満杯で、要素が書き込めなかったことを示します。アプリケーションがデータ損失を許容できない場合、書き込もうとした値をキャッシュし、書き込み操作を再試行する論理を追加する必要があります。

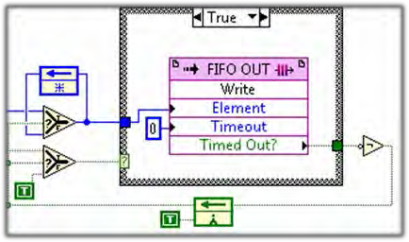

図 90. FIFO への書き込みがタイムアウト状態になる場合、それに対処する論理を追加する必要がある。書き込もうとした値はフィードバックノードを使用してキャッシュされ、FIFO に空きができて書き込み操作が成功するまで、上方のコードが追加の値を生成しないようにする。

FIFO は、一時に読み取りまたは書き込みができる要素の数をクエリする方法も提示します。これらのメソッドは、FIFO 要素数を追跡し、提示する回路を追加します。回路を追加すると、SCTL が実現できるクロックレートにマイナスの影響が及びます。それでも、これらのメソッドが便利なのは、FIFO がデータのバーストを生成する IP とやり取りする場合です。予測されるデータのバーストのサイズが不明の場合、通常、単純に「タイムアウト?」端子を使用して、必要に応じてハンドシェイクを許可するほうが適切に対応できます。

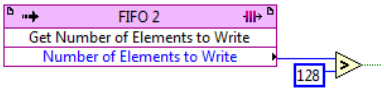

図 91.「書き込み可能要素数を取得」メソッドは、通常、FIFO が予測される出力データバーストを受け入れられるようになるまで、バーストの多い IP を実行させないようにする場合に使用される。

「**リソースの最適化テクニック**」で述べたように、プロジェクトおよび VI 定義の FIFO の実装リソースタイプは指定できます。

同じクロック領域を共有する SCTL 間でデータを受け渡しする場合、フリップフロップを実装すると、最も高いパフォーマンスが得られますが、スライスを多数使用するため、最大 100 バイト前後の小さい FIFO に使用するべきです。LUT ベースの FIFO は中間にあたるオプションで、100 バイトから 300 バイトの範囲の FIFO に適しています。

ブロックメモリは、クロック領域全体のデータ転送に対応する、唯一の FIFO 実装手段です。言い換えれば、ブロックメモリ FIFO は、様々なベースから得られる様々なレートやクロックを使って、ダイアグラム全体のデータ転送に対応します。たった 1 つのクロック領域からブロックメモリ FIFO へデータを書き込み、別の領域から読み込むことができます。

同じクロック領域内の SCTL から FIFO が書き込まれたり、読み込まれたりするとき、LabVIEW はより実装しやすい同期機能を使って、レイテンシを低くすることが出来るので、より高いクロックレートでコンパイルできる可能性が高くなります。

相対的に容量の大きい FIFO（300 バイト超）を実装する場合、ブロックメモリ FIFO を使います。ブロックメモリ FIFO には、書き込まれたデータが読み取りに至るまで、最大 6 クロックサイクルのレイテンシが許容されます。

ハンドシェイク項目

ハンドシェイク項目は、ハンドシェイクを使って、1 つの書き込みと 1 つの読み取りの間で無損失のデータ転送を確約する、バッファのないメカニズムです。ハンドシェイク項目は FIFO のように 1 つの要素として動作しますが、独自のメカニズムとして存在し、FIFO では不可能なことを可能にするような処理を行います。ハンドシェイク項目は SCTL 内でのみ使用でき、クロック領域間で通信する際に使用できます。

FIFO と同様、ハンドシェイク項目はコンテンツを消去するオプションも提供します。また、非破壊的な読み取りを行う手段も提供します。読み取り側において、ハンドシェイク項目のコンテンツを読み取る際、コンテンツを空にしたり、データが読み取られたことを書き込み側に通知することなく、後から明示的な応答を発行します。それによって、書き込み側は新しい値を供給することができます。読み取りと書き込みが同期する必要がある場合、この手段のほうが変数より適しています。

ハンドシェイク項目は、クロック領域全体で値を転送するのに、複数のクロックサイクルを必要とします。これはそのまま通信遅延につながります。変数やレジスタ項目と同様、ハンドシェイク項目はバッファリングを提供しません。したがって、データ転送には向いておらず、スループットのパフォーマンスはレイテンシほど意味を持ちません。

レジスタや FIFO と同様、ハンドシェイク項目は VI またはプロジェクトによって定義されます。どちらのタイプのパフォーマンス特性も、スループットとレイテンシに関しては同じです。ハンドシェイク項目は、FIFO よりも消費する FPGA 論理リソースが少なく、ブロックメモリを消費しません。

メモリ項目

メモリ項目は主要な FPGA ストレージメカニズムを表し、FPGA 内でデータを転送します。アプリケーションの異なる部分からアクセスできる、汎用的なアドレス可能スペースを提供します。

メモリ項目の実装リソースタイプは、次のいずれかに構成できます。

ブロックメモリ—ブロックメモリを使用するメモリ項目は、他のタイプのメモリ項目と比べて高いクロックレートでコンパイルします。ブロックメモリは、読み取り/書き込みアクセスやデュアルポート読み取りアクセス用に構成することができます。また、ブロックメモリを使って実装されたメモリ項目を使い、1 つのクロック領域にデータを書き込み、異なるクロック領域からデータを読み取ることもできます。このような実装においては、各メモリ項目に対して、1 つの書き込みノードと 1 つの読み取りノードしか使用することができません。ブロックメモリは、ストレージ目的では FPGA 論理リソースを消費しません。

ブロックメモリを使って実装されたメモリ項目を複数のクロック領域に使う場合、同じアドレスに対して同時に読み取りと書き込みを行うことができますが、誤ったデータを読み取る可能性もあります。メモリ項目にはハンドシェイクが組み込まれていません。したがって、同じ項目を処理する読み取りと書き込みとの間の衝突を回避したり、メモリ全体でデータのコヒーレンスを保証したり、無損失な形でデータを転送したりするには、ハンドシェイクコードを追加する必要があります。

一般的に、無損失転送が重要な場合、FIFO が望ましいメカニズムです。メモリ項目が役に立つのは、データへのランダムアクセスが重要で、読み取りが認識するメモリ状態が、長時間にわたる全ての状態ではなく、最新の状態のみであることが許容される場合です。

ブロックメモリの実装では、クロックサイクル全体を使って、要求されたアドレスを読み取る必要があります。結果としては、フィードバックノードをデータ出力の後に配置して、次のクロックサイクルの間にデータの値を読み取る必要があります。これによって、読み取り操作のレイテンシ全体に 1 サイクルが追加されます。

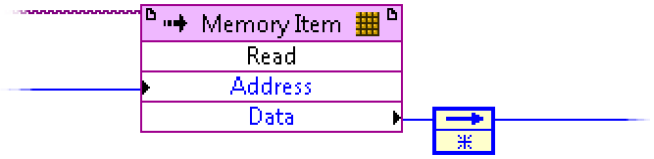

図 92. ブロックメモリを使ったメモリ項目は、値を取得するのにサイクル全体を必要とする。したがって、フィードバックノードを読み取り操作の直後に配置する必要がある。

ルックアップテーブル（LUT）—分散型 RAM としても知られている LUT は、FPGA に接続された論理ゲートから構成されています。LUT は論理リソースを消費しますが、それは論理リソース、あるいはメモリとして機能する場合があるからです。

メモリ項目が LUT とともに実装されている場合、読み取りの出力を初期化されていないシフトレジスタに直接配線する必要はありません。アドレスパラメータを供給するサイクル内で、メモリ項目から読み取ることもできます。

LUT をメモリ項目に使用するのは、SCTL のメモリにアクセスする場合、1 サイクル内でデータを読み取る必要がある場合、残りのブロックメモリが限られている場合のいずれかの場合です。

ダイナミック RAM（DRAM）—DRAM は、FPGA 外のメモリ形式で、一部の NI RIO デバイスで使用できます。DRAM は、数百から数千メガバイトの大容量ストレージを供給します。結果として、FPGA 上のどこにも収まりきらないデータを保存する際に使用できます。ただし、DRAM は FPGA の外にあるため、アプリケーションは 1 つのクロックサイクル内で DRAM からデータを受信することはできません。データを受信するには、「データを要求」および「データを取得」メソッドを使用する必要があります。メモリへ逐次アクセスした場合、複数の読み取り要求が衝突する際に、確定的なタイミングが回避される場合があります。

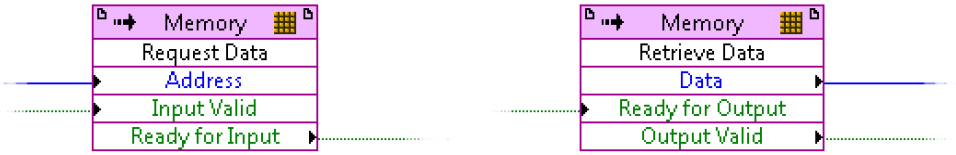

図 93. DRAM の読み取りは、特定のアドレスからデータを要求し、数サイクル後に「データを取得」メソッドの「Output Valid」信号が「TRUE」になった際に取得するという形で行われる。

メモリへ逐次アクセスした場合、複数の読み取り要求が衝突する際に、確定的なタイミングが保たれない場合があります。また、ダイナミック RAM は、定期的に更新される必要があります。これによって、更新されたメモリアドレスが読み取り要求または書き込み要求と衝突する際に、タイミングに不確定性が生じます。

DRAM のパフォーマンスを最適化するには、データの要求の送信や DRAM へのデータの書き込みをバースト転送し、例えば、1 クロックサイクルで行うなどします。

SCTL 間通信のメカニズムのまとめ

次の表は、SCTL 間通信のメカニズムの各特徴をまとめたものです。

転送方法	損失が多い	SCTL へのアクセス数	FPGA リソース	クロック領域をまたぐ [1]	主な用途
グローバル変数	はい	1 書き込み 複数読み取り	論理	はい	1 つ以上の VI 間で現在の変数値を共有
ローカル変数					1 つの VI とフロントパネル項目の間で現在の変数値を共有
レジスタ項目	はい	複数書き込み [2,3] 複数読み取り	論理	はい [4]	1 つ以上の VI 間で現在のレジスタ値を共有する再利用可能な IP を書き込む
メモリ項目	はい	複数書き込み [2,3,8] 複数読み取り [2,8]	LUT	いいえ	1 つのクロック領域から少量のメモリ（100 バイト）への高速かつランダムな読み取り/書き込みアクセス
			BRAM	はい [4,5,6,7,9]	最大 2 つのクロック領域から中程度の量のメモリ（キロバイト）へのランダムな読み取り/書き込みアクセス

			DRAM	はい [5,6,7]	最大 2 つのクロック領域から大量のメモリ（メガバイト）へのレイテンシの大きいランダムな読み取り/書き込みアクセス
ハンドシェイク項目	いいえ	1 書き込み 1 読み取り	論理	はい	最大 2 つのクロック領域の間で同期した状態で値を共有する再利用可能な IP を書き込む
FIFO	いいえ [10]	複数書き込み [2,3] 複数読み取り [2]	BRAM	はい [3,4,5,7]	最大 2 つのクロック領域のループ間における無損失かつ大規模なデータ転送およびストレージバッファ
			LUT	いいえ	1 つのクロック領域内における無損失かつ中程度のデータ転送およびストレージバッファ
			フリップフロップ		1 つのクロック領域内における高速かつ無損失かつ小規模なデータ転送およびストレージバッファ

1. クロック領域の横断とは、そのメカニズムを使って、異なるクロック領域にある SCTL 間でデータを受け渡しできるかどうかを示しています。これらのメカニズムは全て、同じクロック領域を使った SCTL 間でのデータの受け渡しに使うことができます。
2. 複数のユーザがアクセスする必要がある場合、アービトレーションを明示的に無効にする必要があります。
3. 複数の書き込みがある場合、独自の同期メカニズムを実装して、データ破損や、書き込みの衝突による誤ったアドレス指定を回避する必要があります。
4. 全ての書き込みは同じクロック領域に属している必要があります。
5. 全ての読み取りは同じクロック領域に属している必要があります。
6. クロック領域を横断するには、デュアルクロックオプションを有効にする必要があります。
7. 複数の読み取りがある場合、独自の同期メカニズムを実装して、データ破損や、読み取りの衝突による誤ったアドレス指定を回避する必要があります。
8. ブロックメモリを使ったメモリ項目は複数の読み取り SCTL に対応します。1 つの書き込みと 1 つの読み取りにデュアルポート BRAM インタフェースを使い、アクセスのジッタを最小化します。
9. 複数のクロック領域のブロックメモリを使って実装されたメモリ項目を使う場合、同じアドレスに対して、読み取りと書き込みを同時に行うことが可能です。ただし、それを行うと、誤ったデータを読み込む場合があります。
10. 「Timed Out?」端子を使って、「Read」または「Write」メソッドが成功したかどうかを判断します。

9.3. FPGA とホストシステムとの間でデータを転送する

FPGA ホストインタフェースは、ホストと FPGA との間でデータを転送する主なメカニズムを統合します。ホストベース処理は、LabVIEW RIO アプリケーションで鍵となる役割を果たすことができるため、異なる FPGA インタフェースメカニズムのレイテンシとスループットの特性を把握しておくことが重要です。このセクションでは、これらのメカニズムのパフォーマンス面と、推奨される使用法について説明します。

FPGA ホストインタフェースを使用して、ホストコンピュータと FPGA との間でデータを転送し、次のようなことができます。

- FPGA でまかなえる以上のデータ処理を実装する。
- FPGA ターゲットで不可能な操作を行う。例えば、倍精度浮動小数点や拡張精度浮動小数点の計算など。
- FPGA ターゲットをより大きなシステムのコンポーネントとして使用して、多層的なアプリケーションを作成する。
- データをディスクに保存する。

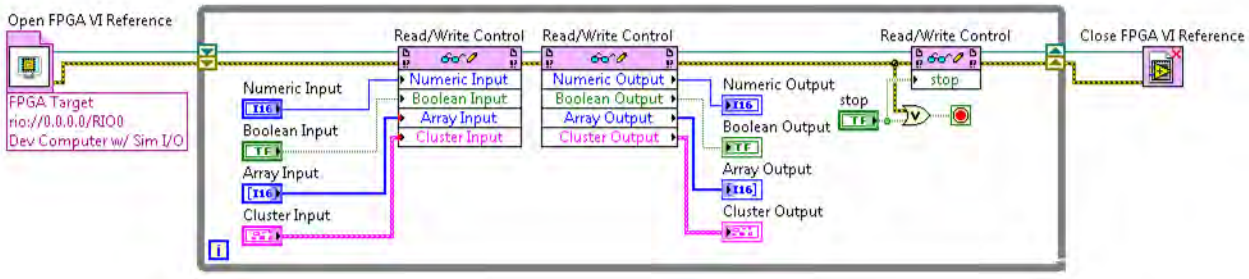

図 94. FPGA ホストインタフェースは、FPGA とホストとの間でデータを転送する主なメカニズムを統合する。トップレベルの FPGA VI 制御器および表示器にプロパティノード経由でアクセスする、オブジェクト指向パラダイムが使用される。メソッドをインタフェースで呼び出して、DMA FIFO アクセスなどの異なる操作を実行することができる。

トップレベル VI の制御器と表示器

LabVIEW FPGA モジュールは、トップレベル FPGA VI の全ての制御器および表示器に対して、ハードウェアレジスタのレジスタマップを作成します。FPGA ホストインタフェースから、これらのレジスタに対して、読み取りおよび書き込みができます。ドライバおよびローレベル I/O プログラミングに詳しい場合、これが FPGA に置き換えた場合のメモリマッピングされたレジスタアクセス（MMRA：Memory-Mapped Register Access）だと考えることができるでしょう。

「制御器を読み取る/書き込む」関数は、数値/ブール制御器などのスカラデータと、配列やクラスタなどの構造化データに対応します。「**リソースの最適化テクニック**」章で説明したように、アレイは多くの FPGA リソースを消費するため、FPGA VI のフロントパネルでアレイを使いすぎないように注意する必要があります。

トップレベル FPGA VI の制御器および表示器を使うと、FPGA との通信において最も低いレイテンシを実現できます。マシンによって異なるものの、FPGA に対する読み取りと書き込みは通常、マイクロ秒範囲です。これによって、ステータスの交換や、構成パラメータの設定が効率よくできるようになります。マルチコアホストを使い、CPU コアを特定のレジスタからの読み取り専用にすることで、最も低いレイテンシと正確な同期が実現します。LabVIEW Real-Time ホストを使うと、レイテンシが良くなり、確定性も加わるため、FPGA に対する最大のアクセスレイテンシが抑制されます。

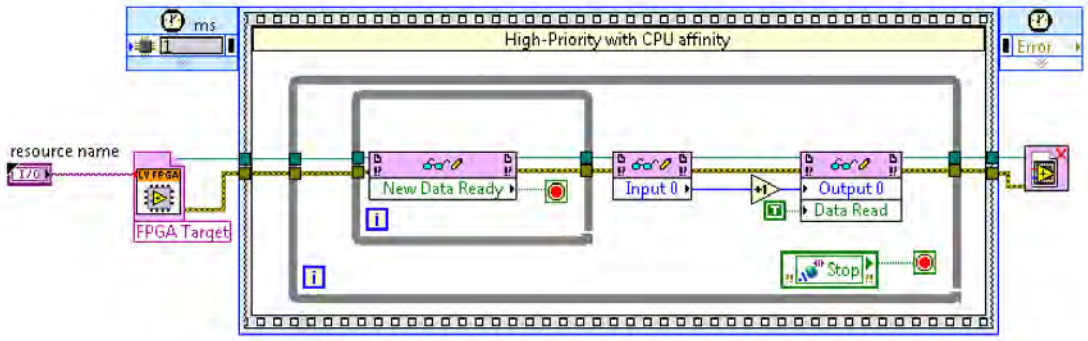

図 95. タイミングシーケンスストラクチャは、FPGA VI 制御器にポーリングするホスト側のループに、CPU の親和性と高優先度を割り当てる。このアプローチによって、FPGA とホストの間で時間制約のあるデータを転送する際、最も低いレイテンシが得られる。

FPGA はホストインタフェースレジスタのアップデートを、ホストがそれらを読み取るよりもはるかに速く実行できることを覚えておいてください。その結果、ホストアプリケーションが値を受信し損ねやすくなります。複数の読み取り間でデータのコヒーレンスを維持したい場合や、FPGA とホストとの間でデータを無損失で転送したい場合、追加のハンドシェイクや、異なる同期メカニズムが必要になります。FPGA ホストインタフェース FIFO はホストメモリに直接アクセスするため、FPGA とホストとの間で大量の情報を転送する場合に推奨される方法です。

FPGA ホストインタフェース FIFO（直接メモリアクセス）

FPGA ホストインタフェース FIFO は DMA（直接メモリアクセス）を使用して、高速かつほとんどプロセッサを使わずに、データをバッファリングし、ホストシステムメモリに対して転送します。この手段は、大きなデータブロックを送信する場合、フロントパネルの制御器および表示器に比べて、効率のよいメカニズムになると言えます。

FPGA ホストインタフェース FIFO の API は、標準 FPGA FIFO の API と似ています。ホストインタフェース FIFO は一方向の転送メカニズムで、ホストから FPGA へ、あるいは FPGA からホストへ転送するように構成することができます。ホストインタフェース FIFO の API は、FPGA とホストのどちら側からアクセスしているかによって異なります。

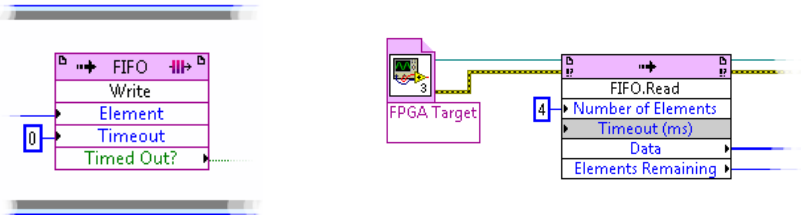

図 96. ホストインタフェース FIFO は、FPGA とホストシステムとの間の DMA 通信の複雑性を抽象化する。この例は、FPGA 側から 1 度に 1 つの要素を書き込み、ホスト側で 1 度に複数の要素を読み込む方法を示している。

FPGA ホストインタフェース FIFO を構成して最高のパフォーマンスを引き出す

作成するコードは、過渡条件を処理し、防ぐことで、ホストインタフェース FIFO で可能な限り最高のスループットを実現する必要があります。このセクションの後のほうで述べていますが、FIFO はデータに対して「準備」することができます。転送プロセスの初期段階で、DMA エンジンの初期化順序、ソース、およびシンクを制御して、エラーの可能性を最小限にすることができます。

DMA チャンネルは 2 つの FIFO バッファから構成されます。ホストコンピュータのバッファと FPGA ターゲットのバッファです。各側でそれぞれのバッファに働きかけ、特定の条件が満たされると、DMA エンジンが一方からもう一方へデータを転送します。

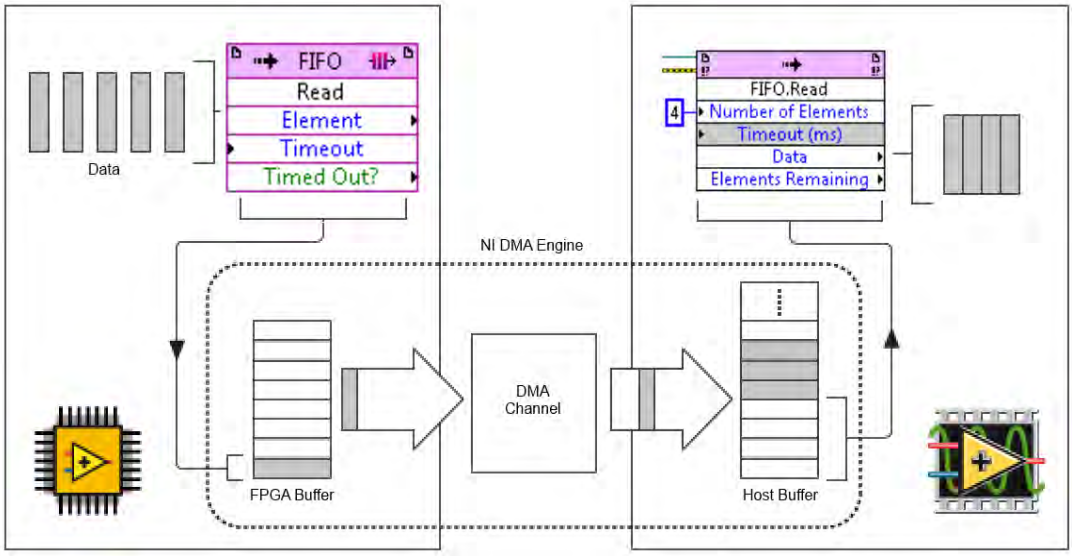

図 97. DMA エンジンはローカルバス経由でデータを転送し、ホストメモリへ直接アクセスする。そのときの転送速度はバス帯域幅にほぼ等しく、ホスト CPU の負荷はほとんどない。

ホスト側バッファは、転送プロセスが開始される前に、実行時にホストアプリケーションによって構成できます。ホスト側バッファの現在のデフォルトサイズは 10,000 要素より大きく、FPGA 側バッファサイズの 2 倍です。NI では、オーバーフローまたはアンダーフローエラーが発生した場合、このバッファを 4,096 要素の倍数のサイズに拡張することをお勧めしています。ガイドラインとして、ホスト側バッファを少なくとも、一度に読み取りまたは書き込みを行う要素の数の 5 倍に設定することをお勧めしています。レイテンシが懸念事項でない場合、ホスト側バッファを拡張して、数秒間データを保留します。

FPGA 側バッファは、FPGA の論理リソースを使って実装されます。FPGA 側バッファのサイズはコンパイル前に、FIFO プロパティのダイアログボックスを使って設定することができます。このダイアログボックスには、LabVIEW プロジェクトの FIFO 項目からアクセスします。FPGA 側バッファのデフォルトは 1,023 要素です。FPGA はこのバッファをホストよりもはるかに速く提供することができます。また、FPGA は通常リソースに制限があります。そのため、NI では、このバッファのサイズを拡張することでパフォーマンスが向上するという確固たる証拠がない限り、バッファのサイズは変えないことをお勧めしています。設計が完成に近づくと、レイテンシが大きな懸念事項ではない場合、コンパイルプロセスが許す限り、FPGA 側バッファを拡張することを検討できます。

多くの場合、FPGA は、DMA エンジンに対するデータの提供と供給に関しては、ホストシステムよりはるかに速度が速くなります。また、FPGA は、一時的なバッファスペースの観点からも、ホストよりも多くのリソースの制約があります。したがって、CPU はできるだけ頻繁に読み取りを行って、FPGA バッファのサイズを小さく維持する必要があります。

FPGA がデータをホストに転送するとき、CPU は FPGA バッファを可能な限り空にしておこうとします。そのようにしておくことにより、過渡条件が一時的に CPU やローカルバスに影響を与えた場合、FPGA がデータ保存しておく場所を確保できるからです。ホストがデータを FPGA に転送するとき、ホストは FPGA バッファを可能な限り満杯にしておこうとします。そのようにしておくことにより、過渡条件が発生した場合、FPGA に処理するべきデータが十分ある状態になるからです。

FPGA からホストへデータを転送する場合、ホストが読み取りまたは開始メソッドの呼び出しを実行するまで、DMA エンジンは開始されないことを知っておく必要があります。DMA エンジンが実行されていない場合、FPGA の FIFO バッファはすぐにオーバーフロー状態になってしまいます。したがって、FPGA 側の FIFO に書き込む前に、ホスト側の開始または読み取りメソッドを呼び出して、DMA を開始する必要があります。

転送が開始されると、ホストは、読み取りメソッドを呼び出して、ホスト側バッファから読み取りを行います。ホスト側バッファが満杯になった場合、DMA エンジンはデータの転送を停止し、FPGA 側 FIFO がタイムアウト状態となった時点でオーバーフローを報告します。

ホスト側読み取りメソッドは、ポーリングメカニズムを使用して、タイムアウトになる前に要求されたデータの量を確認します。ホスト VI のタイミングループまたはタイミングシーケンスストラクチャ内に読み取りメソッドを配置して、CPU コアをこのプロセス専用にすると、CPU の負荷が高くなるのと引き換えに最高のパフォーマンスが得られます。ホスト側が DMA 転送に対応しており、読み取りの呼び出しがデータを返さない際には、タイムアウト値を低く設定して、明示的にマイクロ秒待機する関数を呼び出すことで、待機中の CPU 時間をシステム上の他のプロセスに利用することができます。

ホストから FPGA へデータを転送する場合、または FPGA に常に処理または出力するデータがあることが重要な場合、FPGA 側からデータを読み取るプロセスを開始する前に FIFO に書き込みを行います。この場合、より大きいホスト側バッファのほうが過渡条件に対応するには適しています。ただし、転送レイテンシが高くなります。

ホストインタフェース FIFO のレイテンシとスループット

前述のとおり、レイテンシは、転送中のデータ量とそれに伴うメカニズム上のオーバーヘッドが加味されたものに直接比例します。転送中のデータの大半は、FPGA 側バッファとホスト側バッファのいずれかにあります。

FPGA からホストへデータを転送する場合、ホスト側バッファは定常状態でほぼ空である必要があります。さらに、FPGA 側バッファは、相対的に小さくある必要があります。それにより、DMA エンジンがデータブロックをホストに転送する頻度からレイテンシを判断できるからです。DMA エンジンは、次の条件のいずれかが満たされると必ずホストへデータを転送します。

- FPGA 側バッファが 4 分の 1 満たされる。
- FPGA 側バッファが最低 512 バイト（PCI Express フルパケット）
- DMA コントローラの強制転送タイマが発動する（このタイマは約 1 マイクロ秒）

ホストから FPGA へデータを転送する場合、ホスト側バッファは定常状態でほぼ満杯である必要があります。さらに、FPGA 側バッファは相対的に空のままである必要があります。そのため、平均レイテンシが平均スループットで分割されたホスト側バッファのサイズによって決定されます。平均スループットは FPGA が DMA FIFO から読み取りを行う頻度によって決まり、最大システムバス帯域幅以下になります。

FPGA DMA システム帯域幅は、選択するバス技術とプラットフォームによって決まります。バス帯域幅は NI が最新バス技術を採用すると即座に向上するため、必ず製品マニュアルで最新の仕様を確認するようにしてください。NI ハードウェア製品の現在の帯域幅の性能には、次のような例があります。

- FlexRIO デバイスは現在、PXI Express シャーシで PCI Express 技術を活用することで、最高の FPGA ベース DMA スループットを提供します。
    - FlexRIO シリーズの NI PXIe-797xR は、4 つの PCI Express 2.0 レーンを使用して、1.7 GB/秒の一方向スループットを実現します。また、双方向にデータを転送する場合は 1 方向あたり 1.2 GB/秒を実現します。
    - FlexRIO シリーズの NI PXIe-796xR は、FPGA への転送の場合、800 MB/秒、FPGA からの転送の場合、838 MB/秒、双方向にデータ転送の場合、1 方向あたり 800 MB/秒を実現します。
- cRIO-9068 は、高速 AXI バスを使用して、1 個の DMA チャンネルの場合、300 MB/秒、16 個の DMA チャンネルの場合、合計で 300 MB/秒を実現します。
- マルチコア cRIO-9082 は、PCI バスを使用して FPGA をプロセッサに接続し、通常、90～100 MB/秒の帯域幅を実現します。

ホストインタフェース FIFO のバッファのコピーを減らす

ホストインタフェース FIFO に対して読み取りまたは書き込みを行う場合、LabVIEW と NI-RIO ドライバメモリバッファとの間でデータがコピーされます。これらの追加的なデータコピーは CPU サイクルを消費し、FIFO を使って作成できるアプリケーションのサイズを制限します。LabVIEW FPGA モジュールは DMA FIFO API を拡張して、NI-RIO ドライバ側の DMA バッファに直接アクセスすることを可能にします。これには、LabVIEW で、「読み取り領域を取得」および「書き込み領域を取得」FIFO メソッドをデータ値リファレンスの概念と組み合わせる必要があります[3]。

領域 API を使えば、NI-RIO バッファに対するデータ値リファレンスを取得でき、In Place 要素ストラクチャの領域内で直接データを操作できます。ストラクチャ内でリファレンスデータを読み取れば、データはコピーされません。また、いかなる書き込み操作も、直接ドライバ側バッファに対して実行されます。データ値リファレンスは使用した後、削除して、ドライバが次の転送操作に対してメモリを再利用できるようにする必要があります。

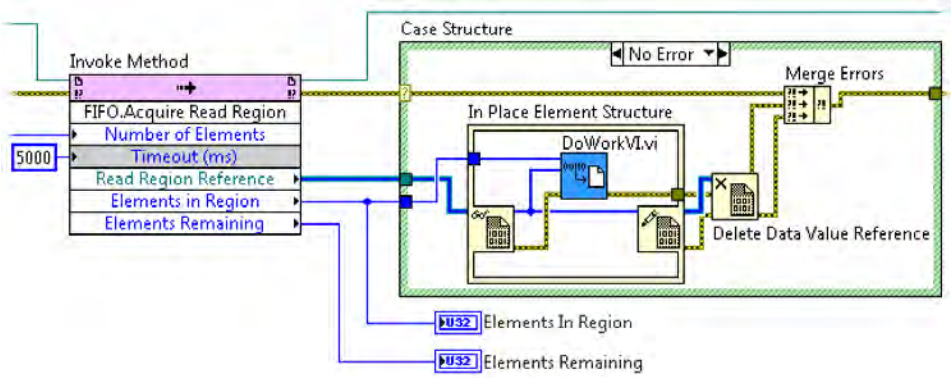

図 98. このブロックダイアグラムは、「読み取り領域を取得」メソッドで構成された「メソッドをインボーク」関数を使用して、バッファからの読み取りを行う領域を取得する。次に、リファレンスを In Place 要素ストラクチャを使って読み取り領域に渡す。そこで、バッファのコピーなしで、さらなる処理が行われる。

割り込み

LabVIEW FPGA デバイスはハードウェア割り込みを使って、ホストと同期させることができます。割り込みは、ホストが FPGA からの信号を待つアプリケーションで便利です。例えば、インジケータのステータスをポーリングして、イベントが発生したかどうかを確認するようなアプリケーションでは連続的にインジケータからの読み取りを行えるだけのホストの処理能力が必要となってしまいます。一方、割り込みはポーリングを用いないないメソッドで、ホストがデバイスにステータスのクエリを行う必要がないため、CPU とバスのリソースを解放できます。

割り込みはポーリングと比較して、1 回の使用あたりのコストが高くなります。これは OS のオーバーヘッドのためです。割り込みが生じるとき、多くの場合、OS は複数のデバイスをクエリして、割り込みのソースを特定します。そのため、割り込みのレイテンシは、バス、CPU、OS の選び方によって決まります。

平均の割り込みレイテンシが最も低くなるのは、高性能組込コントローラを搭載した PXI システムです。この場合、レイテンシは通常、数マイクロ秒の範囲内になります。CompactRIO コントローラでは通常、割り込みレイテンシは数十マイクロ秒内で、最大でも 100 マイクロ秒です。コントローラに LabVIEW Real-Time を使用すると、確定的な OS 応答のおかげで、割り込みレイテンシが一貫したものになります。

9.4. デバイス間でのデータ転送

ピアツーピア転送

PXI では、ピアツーピア転送を使って、FPGA ターゲットから直接もう一つの FPGA ターゲットへ、ホストプロセッサをバイパスし、高レートでデータを転送します。また、ピアツーピア転送を使って、FPGA デバイスと FPGA ではないデバイスとの間でデータを

転送することもできます。FPGA ではないデバイスとは、例えば、NI のデジタイザ、任意信号発生器といったモジュール計測器です。

ピアツーピア転送を実行するには、PCI Express 技術を活用して、PXI シャーシ内のデバイス間で直接的な一方向通信パスを設定します。直接接続が便利なのは、高性能 LabVIEW FPGA アプリケーションをシャーシ内の複数の FlexRIO デバイスに配布する場合です。アプリケーションが FlexRIO デバイスで容量不足になった場合、単純にシャーシにボードを追加し、そのボードに対してデータを転送して、さらなる処理を行うことができます。

NI にはピアツーピア転送に対応するモジュール式計測器が複数あるため、これらの高精度計測器を使って、データを取得し、直接 FlexRIO デバイスに転送して、処理を行うことができます。例えば、高スループット PXI Express ベースデジタイザは、FPGA デバイスにデータを転送して、スペクトル解析を実行できるため、ホストプロセッサを解放して、その他のタスクを実行することができます。PXI Express はツリートポロジを使用します。つまり、デバイスに適したスロットを慎重に選択することで、ピアツーピア転送を使用して、システム内の他のピアツーピア転送に影響することなく、デジタイザから FPGA デバイスにデータを直接送信することができます。

ピアツーピア転送は、LabVIEW FPGA のその他の FIFO と同様、FIFO インタフェースの役割を果たします。このインタフェースでは、FIFO で使用可能なサンプル数とスロット数、ピアツーピア転送の状態を判断することもできます。この API を使用して、1 つのデバイスからデータを書き込み、数マイクロ秒後に別のデバイスから読み取ることができます。

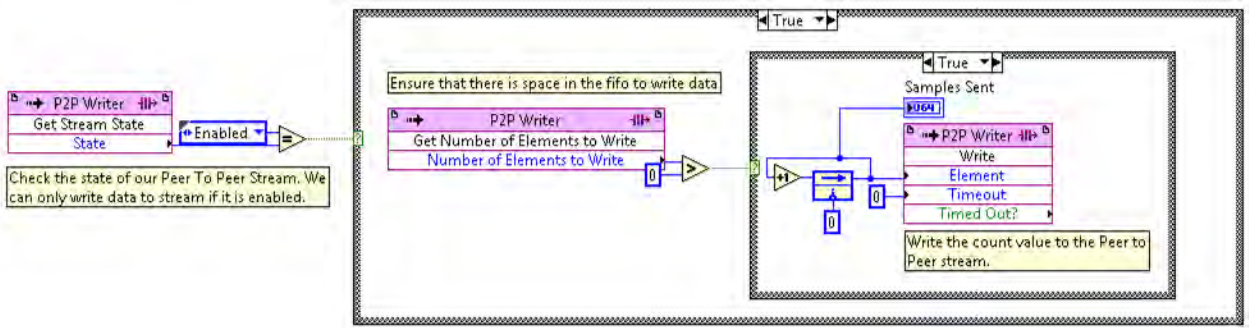

図 99. FPGA 側のピアツーピア転送 API は、ピアのバッファで使用可能な容量や要素数、転送状態をクエリすることができるなど、ホストインタフェースと FPGA FIFO と同様、FIFO インタフェースの機能を果たす。

ホストシステムのピアツーピア接続を設定するには、API を使って、ピアツーピアエンドポイントをリンクし、転送を可能にします。設定が完了すると、FPGA がピア間で送信を行うため、ホストは関与しなくなります。

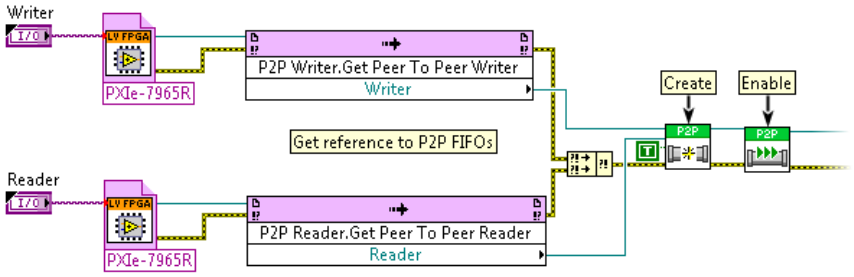

図 100. ピアツーピア転送はホスト側 API で設定し、監視する。

ピアツーピアのスループットは、使用するデバイス、シャーシ、スロット構成、コントローラによって決まります。ピアツーピア転送アプリケーションに適したハードウェアの選択と構成を行う際は、次のセクションで述べる例と要素を検討してください。

ピアツーピア転送に適したデバイスの検討

ピアツーピアデバイスは、求められたレートでデータを供給し、受信できる必要があります。そのため、バスインタフェースおよび処理論理の速度に要求を課すことになります。実現可能な全体のレートは、最低速度のピアによって制限されます。

- FlexRIO デバイスの NI PXIe-797xR は、PCI Express 2.0 通信に対応しており、ピアツーピア通信では最高スループットを実現します。これらのデバイスは 4 つの PCI Express レーンを利用して、一方向転送で最大 1.5 GB/秒のスループットを実現します。
- FlexRIO デバイスの NI PXIe-796x は、PCI Express 1.0 レーンを使用し、モジュールに対する入出力が 800 MB/秒以上でデータを転送します。双方向で同時転送する場合、FPGA モジュールは、1 方向あたり 700 MB/秒以上のレートを実現できます。

ピアツーピア転送に適したシャーシの検討

PCI Express バックプレーンスイッチは、データをシャーシ経由でルーティングし、広帯域ポイントツーポイント接続によってピアツーピアデータ転送を可能にします。帯域幅は、シャーシスロットに収容された、同じ PCI Express スイッチに直接接続されたモジュールのスイッチによって決まります。

NI PXIe-1085 シャーシは、全てのスロットにおいて、8 つの PCI Express 2.0 レーンを使って、最大 4 GB/秒を実現します。デバイスが持っている性能は、バスの性能を上回っては使用できませんが、将来デバイスをアップグレードする場合、帯域幅を対応させることはできます。同様に、PXI Express 2.0 デバイスは、PXI Express 1.0 シャーシで使用することはできますが、PXI Express 1.0 の帯域幅を超えて機能させることはできません。

また、シャーシ設計で、デバイスに合わせて選んだスロットに依存した帯域幅を設定することもできます。ピアツーピア転送システムのモジュールが同じ PCI Express スイッチに接続されていない場合、データはバックプレーンの他のスイッチ、またはホストコントローラのスイッチを使って、渡される必要があります。このようにした場合、スループットは、これら他のスイッチの能力に依存したものになります。

NI PXIe-1085 シャーシには、2 つの PCI Express スイッチがあり、8 つの PCI Express 2.0 レーンによって接続されており、スイッチ間帯域幅は 4 GB/秒になります。1 つのピアがスロット 2 とスロット 9 の間の任意のスロットに挿入され、もう 1 つのピアがスロット 11 とスロット 18 の任意のスロットに挿入されると、このピアツーピア接続は、スイッチ間接続よって共有されます。この方式で十分な数のピアを追加していくと、最終的にスイッチ間接続は飽和状態になり、スループットに影響が出ます。そのため、可能であれば、ピアツーピア転送の 1 組のピアは同じスイッチに配置することが推奨されます。

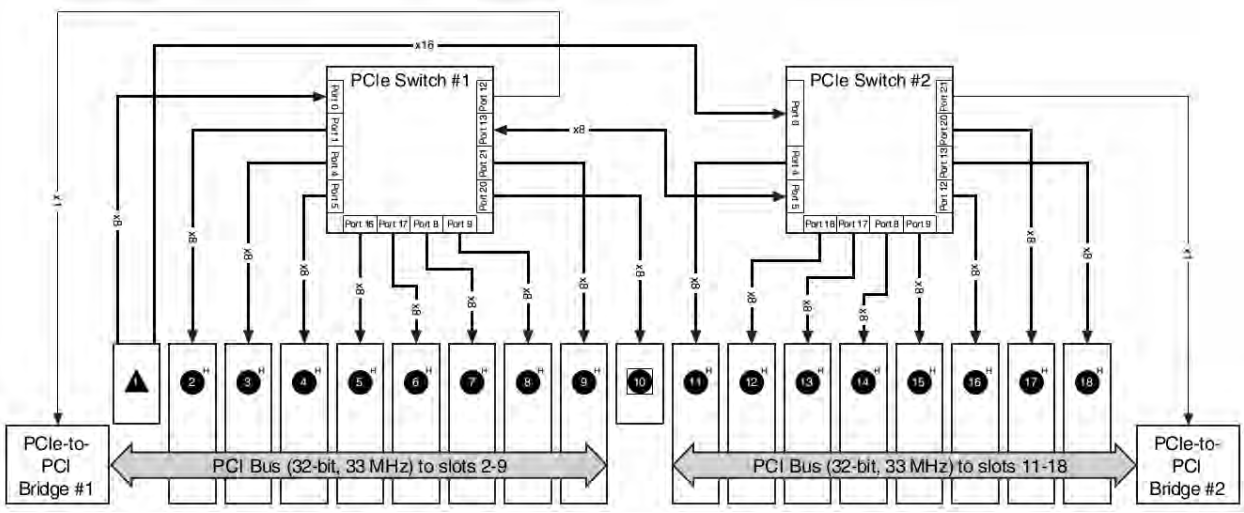

図 101. NI PXIe-1085 シャーシのバックプレーンには 2 つの PCI Express 2.0 スイッチがあり、18 個のシャーシスロットを相互接続している。

NI PXIe-1065 シャーシの場合、2 つのシャーシセグメントがあります。スロット 9 から 14 に収容されたデバイスは、バックプレーンの PCI Express スイッチを直接使用して相互通信します。スロット 7 と 8 に収容されたデバイスは、ホストコントローラのオンボードチップセットスイッチを経由する必要があり、帯域幅はコントローラに依存することになります。

例えば、NI 8130 などの古いコントローラの場合、ピアツーピア転送が 1 シャーシセグメントから、もう 1 つのシャーシセグメントに対して行われる場合、帯域幅が約 640 MB/秒に制限されます。NI 8108 などの一部のコントローラは、ピアツーピア転送に対応しませんが、デバイスがシャーシスイッチに直接接続されている場合、転送は可能です。

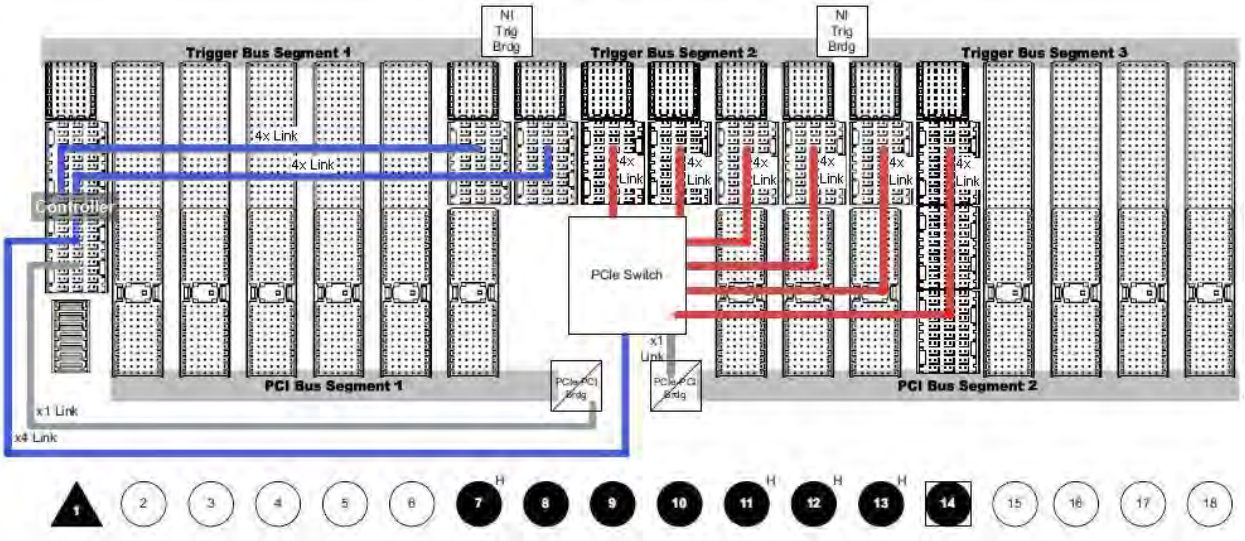

図 102. NI PXIe-1065 シャーシバックプレーンの場合、スロット 7 と 8 に接続されたデバイスは、ピアツーピア転送を行う際、スロット 1 のホストコントローラスイッチを使う必要がある。これにより、スループットはコントローラに依存することになる。

複数のピアツーピア転送が同時に実行される場合、1 つのリンクで同じ方向に転送される際の帯域幅は、均等に共有されます。反対方向の同時転送では、他のスループットにわずかに悪い影響が出ます。それは、データソースに戻る確認メッセージがあるためです。

最後に、複数の FPGA をまたいだデータ転送が必要な閉ループ制御アプリケーションを作成しようとする際、ピアツーピアのレイテンシは重要です。ピアツーピアのレイテンシは通常、2～4 マイクロ秒ですが、転送がシステムの他のトラフィックと競合している場合、一時的に 10 マイクロ秒以上に高くなる場合があります。したがって、レイテンシが重要な場合、ピアツーピア転送は専用の PCI Express スイッチに隔離する必要があります。

関連情報

[1] DMA ベストプラクティス（英語）http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpga_dma_best_practices

[2] ピアツーピアストリーミングの概要
http://www.ni.com/white-paper/10801/en

[3] DMA FIFO にアクセスする際の効率向上（FPGA モジュール）（英語）
http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgahelp/fpga_zerocopy_dma/

10. 次のステップ

LabVIEW 製品マニュアルは、このガイドで説明した多数のタスクの達成方法について詳しく説明しています。ハードウェアマニュアルには、NI RIO デバイスの機能とパフォーマンス特性に関する重要な情報も含まれています。

NI のサポートページ ni.com/support からは、マニュアル、技術サポートデータベース、ドキュメント、チュートリアル、サンプルコード、コミュニティフォーラム、技術サポート、カスタマサービスを利用できます。

10.1. トレーニング

NI では、高スループット LabVIEW アプリケーションの講義形式トレーニングコースをご用意しています。高スループット LabVIEW FPGA コースでは、このガイドで取り上げた多数のトピックをより詳しく扱っています。また、理解度をチェックするための演習も含まれています。

このガイドは、高性能 LabVIEW FPGA アプリケーションというテーマに沿って、シングルサイクルタイミングループ内のプログラミングに焦点を当てています。全てのアプリケーション、つまり、アプリケーションのあらゆる部分で、SCTL プログラミングが必要なわけではありません。標準の LabVIEW FPGA コースでは、SCTL 以外の LabVIEW FPGA アプリケーションのプログラミングが必要な基本概念の多くを網羅しています。

10.2. NI RIO プラットフォームの評価版

LabVIEW および LabVIEW FPGA モジュールの評価版をダウンロードしてお使いいただくことができます。NI ハードウェアがない場合でも、FPGA 設計を作成、シミュレート、コンパイルすることができます。また、LabVIEW RIO 評価版キット[4]（NI RIO プラットフォームを評価するための低コストオプション）をお試しいただくこともできます。

10.3. NI アライアンスパートナーとサービス

NI パートナーは、アプリケーションの設計、統合、実装をサポートいたします。NI のアライアンスパートナーネットワークは、グラフィカルシステム開発を利用して、優れたソリューションと高品質製品をユーザに提供する企業が世界中から 700 社以上参加するプログラムです。

NI プラットフォームを使った特定のアプリケーションの課題に関するご相談がございましたら、NI のフィールドエンジニアまでお問い合わせください。

関連情報

[1]	高スループット LaVIEW トレーニング http://sine.ni.com/tacs/app/overview/p/ap/of/lang/en/ol/en/oc/us/pg/1/sn/n24:16770/id/2158/
[2]	LabVIEW FPGA トレーニング http://sine.ni.com/tacs/app/overview/p/ap/of/lang/en/ol/en/oc/us/pg/1/sn/n24:4769,n8:4398/id/1597/
[3]	セルフペースオンライントレーニング http://www.ni.com/white-paper/14457/en/

NATIONAL INSTRUMENTS

[4]	LabVIEW RIO アーキテクチャ評価オプション http://www.ni.com/rioeval/
[5]	NI アライアンスパートナーネットワーク http://www.ni.com/alliance/

11. 改訂内容とご意見

NI では、品質の良い内容をお届けするため、皆様からのご意見をお待ちしております。このガイドの今後の改訂版に反映させたいご意見がございましたら、hprioguide@ni.com までお送りください。

さらに説明が必要だと感じる部分がございましたら、LabVIEW FPGA ディスカッションフォーラムまでご質問をお寄せください。アプリケーション、サポート、研究開発に携わるエンジニアがお答えいたします。ご意見・ご質問には、最初に「FPGA」という言葉をご記入いただき、「LabVIEW」セクションにご投稿ください。また、「hprioguide」というタグを追加いただくと、より速くご対応させていただくことができます。

改訂番号	日付	改訂内容
1.0	02/14/2014	Initial release
1.1	02/25/2014	Integrating High-Throughput IP chapter: ▪ Truncate rounding biases toward negative infinity. ▪ Overflow and rounding tweaks not needed unless overriding the output type.
		Resource Optimization chapter: ▪ Overflow only coerces, never rounds ▪ Overflow and rounding tweaks not needed unless overriding the output type. ▪ Round-Half-Up biased toward greater values and does not require comparison logic. ▪ Discrete Delay function now supports different data types. ▪ Discrete Delay function supports dynamic delays.
		Minor grammar corrections, clarifications, and figure fixes.