(機械で日本語に翻訳)

The guide to Xillybus Block Design Flow for non-HDL users (deprecated)

Xillybus Ltd.

www.xillybus.com

Version 3.2

この文書はコンピューターによって英語から自動的に翻訳されているため、言語が不明瞭になる可能性があります。このドキュメントは、元のドキュメントに比べて少し古くなっている可能性もあります。

可能であれば、英語のドキュメントを参照してください。

This document has been automatically translated from English by a computer, which may result in unclear language. This document may also be slightly outdated in relation to the original.

If possible, please refer to the document in English.

Block Design Flow はもう入手できません。

このドキュメントは、既存のプロジェクトのみをサポートすることを目的としています。

1

序章

Xillybus Block Design Flow は Verilog / VHDL design flow の代替であり、logic 関連の HDL 言語での変更と設計に慣れていない人を対象としています。その主な目的は、FPGA のバックグラウンドを持たない設計者が、FPGA 関連のスキルを習得する必要なく、coprocessing/acceleration 機能にアクセスできるようにすることです。とりわけ、Xilinx の Vivado High Level Synthesis (HLS) によって生成された logic と、Linux または Microsoft Windows を実行しているコンピューターまたは embedded プラットフォームとの間でデータを交換するための簡単な手段として意図されています。

Block Design Flow は、FPGA FIFOs を介して Xillybus IP core と通信するという Xillybus の主な概念から逸脱しています。代わりに、ユーザー アプリケーション logic は、AXI Stream インターフェイスを介して Xillybus IP block に直接接続します。これにより作業が大幅に簡素化されますが、違いを認識する必要があります。特に、Xillybus のドキュメントで FPGA の FIFOs について言及されている場合、これは Block Design Flow とは無関係です。各 FIFO の代わりに、Block Design の GUI に単純なワイヤがあります。

Xillybus の Block Design Flow を、Zynq processor 環境のセットアップや logic blocks 間の接続に使用される block design ダイアグラムと混同しないでください。このような block designs が適用された場合、それらは無関係であり、Xillybus の IP core を application logic に接続するために選択された方法に関係なく共存できます。

Xillybus を使用すると、設計者は次の方法で生産的なアプリケーション関連の作業に集中できます。

- compilation から FPGA bitstream への準備が整った作業スターター プロジェクトをそのまま提供します。このプロジェクトは、Xillybus の IP core のおかげで、FPGA とコンピューター host の間のシンプルで直感的なデータ交換を

セットアップします。

- C/C++ で logic design をデモンストレーションするためのサンプル High Level Synthesis (HLS) プロジェクトを提供し、このガイドで説明されている主要な要素を使用します (セクション 6 を参照)。
- Vivado の block design ツールを使用して、IP blocks を FPGA design に非常に簡単に統合できます。
- host上でシンプルなプログラミングインターフェースを提供するLinuxおよびWindows用のdriversを供給し、
- 特定のプロジェクト専用に構成された data streams で構成されるカスタム Xillybus IP cores を自動的に作成する Web ツールを提供します。

Block Design Flow は Xilinx の Vivado の block design ツールに依存しているため、このツールの対象となる FPGAs に限定されます。したがって、Xilinx の series-7 FPGAs 以降 (Ultrascale デバイスを含む) のみがサポートされます。

Block Design Flow は使いやすいですが、Xillybus の機能のサブセットにしかアクセスできないため、Verilog または VHDL に基づく FPGA design に精通しているユーザーにはお勧めできません。ただし、IP core または HLS ベースの hardware acceleration/coprocessing などの特定のアプリケーションでは、Xillybus の機能の違いによる影響は無視できます。

Block Design Flow は XillyUSB ではサポートされていません。

2

一般的なガイドライン

2.1 Getting started

原則として、Block Design Flow のプロジェクトのセットアップは、Vivado を使用して、目的のプラットフォームのそれぞれの Getting Started ガイドで説明されているとおりです。

- Xillinux バンドルの場合: Getting started with Xillinux for Zynq-7000
- PCIe バンドルの場合: Getting started with the FPGA demo bundle for Xilinx

これらのガイドに従うときは、必ず **blockdesign**/ サブディレクトリにある xillydemo-vivado.tcl script を使用してください。

重要:

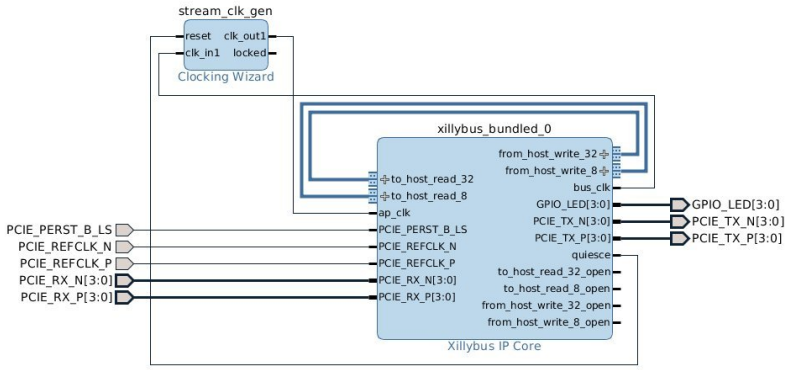
このガイドは、*Xillybus* の *Web* サイトにある *"FPGA coprocessing for C/C++ programmers"* という名前のチュートリアルには対応していません。技術的な詳細と提示されたサンプル プロジェクトにはいくつかの違いがあります。混乱を避けるために、このガイド *(Block Design Flow* の場合*)* または *Web* サイトのチュートリアル *(Verilog / VHDL design* の場合*)* に従うことをお勧めします。

bitfile の生成と使用は、上記の Getting Started ガイドと同じように行われます。bitfile はバンドル "out of the box" からすぐに生成でき、これらのガイドで説明されている loopback テストは同じように機能します。ただし、セクション 4.3 で説明されているように、seekable stream xillybus_mem_8 は Block Design Flow では機能しないことに注意してください。

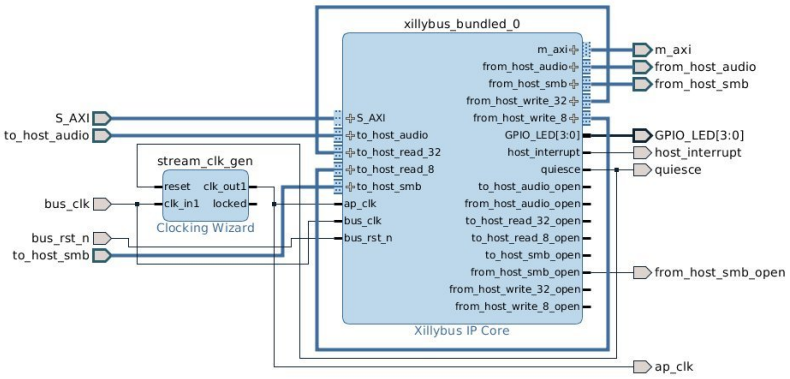
Block Design Flow は、Xillybus の IP core とのインターフェースが Vivado の block design ツールで行われるという点で異なります。プロジェクトを生成し

た後、Vivado の左側のメニュー バーで “Open Block Design” を選択して block design を開きます。

PCIe に基づく designs (つまり、 Xillinux ではない) では、次の図が表示されます。



Xillinux を使用すると、“Open Block Design” は Zynq processor の環境を開きます。この block design の部分は、Xillybus に関連するタスクでは変更しないでください。むしろ、“blockdesign” とマークされている block を (ダブルクリックで) 開く必要があります。これにより、次の block diagram が表示されます。



2.2 block designの注目すべき要素

Xillybus block design には注目すべき重要な要素がいくつかあります。

- Xillybus stream ポート (“from_host_*” および “to_host_*”): これらは標準の AXI Stream ポートであり、最小限の信号セット (TDATA、TVALID、および TREADY) で構成されています。 block design の各ポートの名前は、インター

フェイスの方向を示すために “from_host” または “to_host” のいずれかで始まります。 block design の残りのポート名は、host で表示される device file の名前から “xillybus” プレフィックスを除いたものです。

たとえば、Linux host で `/dev/xillybus_write_32` という名前の device file、または Windows コンピュータで `\\.\xillybus_write_32` という名前の device file は、from_host_write_32 という名前のポートの block design でアクセスできます。

- Loopbacks: 最初に、from_host_write_32 は to_host_read_32 に接続され、from_host_write_8 は to_host_read_8 に接続されます。これにより、xillybus_write_32 という名前の device file に書き込まれたすべてのデータが xillybus_read_32 にループバックされます。同じことが write_8/read_8 ペアにも当てはまります。この loopback によって、Getting Started ガイドで説明されている “Hello world” テストが機能します。

  application logic と統合するには、それぞれの loopback 接続を Vivado の block design GUI で削除し、application logic の適切な AXI Stream ポートと接続する必要があります。

- 場合によっては、xillybus_smb と xillybus_audio という名前の streams が、ボードのオーディオ インターフェイスをサポートするために使用されるため、上の階層に接続されます。これらの streams は無視する必要があります (つまり、 block design の processor design 階層に送られる残りのシグナルとして扱われます)。

- 各 Xillybus stream の “*_open” ポート: 各 AXI Stream ポートには _open サフィックスが付いた対応するポートがあり、関連する Xillybus device file が host で開いている場合、これは高くなります ('1')。この信号は、オプションで stream に接続されている application logic をリセットするために使用できるため、device file が開かれるたびに既知の状態になります。

- Clocking Wizard (stream_clk_gen) block: Xillybus のインターフェースに由来する clock に基づいて、application logic 用の clock を生成します。 Xillybus IP core の AXI Stream ports はすべて、この block の出力と同期しています。

  output clock の周波数を除いて、この block に変更を加えないことをお勧めします。特に、design の implementation (timing constraints) に関連する特定の scripts は、この block の出力をその名前で参照するため、block の名前は (stream_clk_gen) のままにする必要があります。

  以下のセクション 3.2 を参照してください。

- GPIO_LEDS[0:3] などの外部ポート: block design の上の階層に接続されているポート。これらの接続は変更すべきではありませんが、それらの信号はその中のブロックによってサンプリングされる場合があります。たとえば、Xillinux バンドルでは、ap_clk は上位階層に移動しますが、block design ビュー内でも使用できます。

mem_8 stream にはポートがないことに注意してください。 Seekable streams は block diagram にはありません。これについての詳細はセクション 4.3 を参照してください。

3

application logic との統合

3.1 基礎

application logic との統合は、Vivado の block design GUI を使用して行われます。IP blocks は block design に追加され、必要に応じて接続されます。

Vivado's High Level Synthesis (HLS) によって生成された IP blocks の統合については、セクション 6 を参照してください。

Xillybus のドキュメントでは、FPGA の application logic は FIFOs を介して host と通信するとよく言われますが、これは Block Design Flow には当てはまりません(ただし Verilog / VHDL design flow の場合のみ)。 AXI Stream インターフェースを生成する Xillybus の IP Core 内の glue logic には、すでに FIFOs が含まれています (特に bus_clk と ap_clk の間の clock domain crossing 用)。その結果、application logic は、VHDL / Verilog design flow とは異なり、Block Design Flow が使用されている場合、Xillybus の IP core とインターフェイスするために FIFOs を展開する必要はありません。

FPGA と host 間のデータ交換のために、application logic を専用の AXI Stream ポートに接続します (おそらく loopbacks を切断した後)。これらのポートは、TDATA、TVALID、および TREADY のみを提供し、特に TLAST 信号は提供しません。その結果、各 AXI Stream stream は infinite data stream を具現化します (TLAST 信号が可能にするパケット インターフェイスとは対照的に)。これは、一般に Xillybus の device files の infinite stream の性質と一致しています。

Xillybus streams は、FPGA と host の間でパケットを交換するために使用できます。これについては、次の 2 つのガイドのいずれかのセクション 6.3 で説明されています。

- Xillybus host application programming guide for Linux

- Xillybus host application programming guide for Windows

3.2 Clocking

3.2.1 全般的

簡単にするために、user application logic を Xillybus IP Core に接続するすべての信号は、block design 内の Clocking Wizard block によって生成される単一の clock によって駆動される必要があります。この clock (“application clock”) は Clocking Wizard の clk_out1 であり、Xillybus IP Core block の ap_clk 入力でもあります。

この単一の clock で user application block 全体を駆動すると便利な場合が多いため、内部の logic とインターフェイスはすべて logic に依存します。たとえば、Vivado HLS’ synthesizer によって生成された logic には、単一の clock input (ap_clk という名前) があります。この clock input を Clocking Wizard の出力に接続すると、Xillybus IP Core の block との AXI Stream ポート接続が正しく機能することが保証されます。

FPGA ツールは、clock の周波数を周波数自体の観点から参照する場合があり、通常は MHz で、clock period として一般的に ns で参照されることに注意してください。 clock の周波数は clock period の reciprocal であるため、たとえば 100 MHz は 10 ns の clock period と同等です。

3.2.2 application clockの設定

application clock の周波数は、パフォーマンスを向上させるため、または動作する bitfile を達成するためのステップとして設定できます。より高速な clock は、より高い処理スループットをもたらします (他のボトルネックがパフォーマンスを制限しない限り) が、FPGA の logic 要素と Xilinx によるその使用により多くを要求します。ツール。

application clock の周波数が高すぎると、プロジェクトの compilation を FPGA bitstream ファイルに変換すると、timing constraints に適合しないという理由で失敗します。これは、“timing failure” とも呼ばれます。この状況は、implementation を実行するツールが、logic が定義された clock の周波数によって駆動される一方で、信頼できる動作を保証するような方法で logic を利用できなかったことを意味します。このコンテキストでの “timing constraints” は、システム内の clocks の周波数に関する要件です。

application clock の周波数を下げることは (clock generator の制限内で) 常に許可されますが、駆動する logic の動作が遅くなります。

application clock の周波数を設定するには、block design ビューで Clock Wizard (stream_clk_gen) の block をダブルクリックします。 Vivado で構成ウィンドウが開きます。 “Output Clocks” タブを選択し、clk_out1 の “Output Clock Requested” 周波数を変更します。 “Actual” 列の周波数は、clock synthesizer によって生成される周波数を示しています。 output clock は、input clock に有理数を掛けて得られるため、要求された周波数とはわずかに異なる場合があります。有理数は、許可された限られた値のセットから選択されます。

clock が application logic および Xillybus IP core とのインターフェースのみに使用される場合、要求された周波数からのわずかな逸脱は無害です。

Clocking Wizard のその他のパラメータは変更しないでください。

3.2.3 bus_clk 信号

Xillybus IP Core の内部 logic は、bus_clk から application clock を派生させるためだけに block design で公開されている bus_clk によって駆動されます。 application logic は、内部の logic と Xillybus IP Core とのインターフェースに ap_clk しか必要としないため、通常、この信号を他に使用することはありません。

ただし、bus_clk の周波数は、スループットのボトルネックを特定するために重要な場合があります。たとえば、bus_clk が 100 MHz で実行される場合、Xillybus の内部 data pipe は bus_clk の速度で実行されるため、32 ビット幅のデータ インターフェイスを通過できる理論上の最大帯域幅は 400 MB/s です。 ap_clk がより高い周波数で実行され、データが ap_clk の各サイクルでプッシュされる場合、AXI Stream フロー制御信号 (TREADY および TVALID) によってデータ ペースが遅くなる可能性があります。

このため、アプリケーションのスループットを最大化しようとするとき、特にデータ インターフェイスに長いバースト (または連続的な) データ トラフィックが含まれると予想される場合は、bus_clk の周波数を考慮する必要があります。

bus_clk の周波数は、clk_in1 である primary input clock の周波数として、“Clocking Options” タブの下にあります。このパラメーターは、Clocking Wizard にその入力で期待される周波数を通知するため、特定の Xillybus バンドルの bus_clk の周波数を知るために使用できます。

4

アクセラレーション / コプロセッシングのベストプラクティス

4.1 スループット vs. latency

強化された命令セット (x86 ファミリの MMX コマンド、AES の暗号拡張、ARM の NEON 拡張など) に基づく従来のハードウェア アクセラレーションと、GPGPU や FPGA などの外部ハードウェアによるアクセラレーションには大きな違いがあります。強化された instruction sets は processor の実行フローの一部であるため、機械語命令の長いシーケンスを短い命令に置き換え、結果が得られるまでに必要なサイクル数を減らします。

一方、外部ハードウェア アクセラレーション (FPGA アクセラレーションを含む) は、外部ハードウェアとの間でデータを転送する latency の重要性により、結果が利用可能になるまでの時間を必ずしも短縮しません。さらに、pipelining とおそらく clock の周波数が低いため、処理時間も processor よりも大幅に長くなる可能性があります。

したがって、外部ハードウェア アクセラレーションの利点は、latency (結果が得られる速さ) ではなく、スループット (データが処理される速度) です。この利点を利用するには、次の操作を開始する前に 1 つの操作の結果を待つのではなく、高速化するハードウェアとの間でやり取りされるデータ フローを維持することが重要です。

FPGA を使用した適切な高速化の手法は、次の 2 つのドキュメントのいずれかのセクション 6.6 で詳しく説明されています。

- Xillybus host application programming guide for Linux
- Xillybus host application programming guide for Windows

4.2 データ幅とパフォーマンス

比較的高いデータ帯域幅を必要とするアプリケーションの場合、データ集約型の streams には 32 ビット幅の streams (またはそれ以上) を使用することをお勧めします。これは、8 および 16 ビット幅の streams が host の bus をあまり効率的に利用しないためです。

その理由は、ワードが bus のレートで Xillybus 内部データ パスを介して転送されるためです。その結果、8 ビット ワードの転送には 32 ビット ワードと同じタイム スロットが必要となり、事実上 4 倍遅くなります。

これは、データ パスが低速のデータ要素で占有されるため、特定の時点で基盤となるトランスポートを競合する他の streams にも影響を与えます。

このガイドラインは、狭い streams を同じ効率で転送するリビジョン B/XL/XXL Xillybus IP cores には適用されません。

4.3 規則ルール

Block Design Flow を使用する際に注意すべき点がいくつかあります。

- アドレス ポート (“address/data streams”、“seekable streams”) を持つ Streams は、Block Design Flow ではサポートされていません。 Xillybus IP Core にそのような streams が含まれている場合、それらは GUI ではポートとして表示されませんが、host 側では正常に表示されます。 host でそのような stream から読み取ろうとすると、即座に end-of-file 状態になります。反対側に data sink がないため、write() 関数呼び出しは返されません。

  したがって、混乱と FPGA logic のわずかな浪費を避けるために、Block Design Flow での使用を意図したカスタム IP cores では seekable streams を避けることをお勧めします。

- セクション 3.2.2 で説明されているように、必要に応じて出力周波数を変更する場合を除いて、“stream_clk_gen” という名前の block (Clocking Wizard) に変更を加えないでください。

  input clock の周波数を変更したり、構成に他の変更を加えたり、design から取り外して新しい Clocking Wizard IP block に交換したりすると、timing constraints に適合しなくなる可能性があります (timing constraints の例外が block の名前を参照しているためと考えられます)。

  誤った入力周波数を設定すると、FPGA design の動作が不安定になる場合があります。

- clocks の接続方法に注意することが重要です。特に、bus_clk と ap_clk を混在させないでください。
- Xillybus streams が非同期であることを確認してください。これは、デフォルトの IP core と、stream の使用目的が “Data exchange with coprocessor” である場合のカスタム IP cores での autoset の選択の場合です。

  これにより、host で行われた write() 関数呼び出しは、DMA buffers にデータ用の十分なスペースがある場合はすぐに返され、よりスムーズなデータ転送とより高い帯域幅パフォーマンスが保証されます。

  このトピックをよりよく理解するには、Xillybus host application programming guide for Linux または Xillybus host application programming guide for Windowsのセクション 2 を参照してください。

5

カスタム Xillybus IP core の適用

Web アプリケーションを使用すると、ユーザーはカスタム Xillybus IP cores を構成してダウンロードし、streams の数とその属性を Xillybus の Web サイトで直接選択できます。特別に生成されたカスタム IP core は、通常数分後にサイトからダウンロードされます。

カスタム IP core を生成してダウンロードするには、Xillybus の Web サイトで IP Core Factory にアクセスしてください。このプロセスは非常に簡単で、必要に応じて The guide to defining a custom Xillybus IP core が補足情報を提供します。

重要:

AXI Stream 接続はアドレス ワイヤをサポートできないため、*Seekable streams (“address/data”* インターフェイスを使用*)* は *Block Design Flow* には表示されません。このような *streams* を *core* に配置してもほとんど害はありませんが、*FPGA logic* リソースがわずかに無駄になり、*block design* ではなく *host* 側に表示されるため、混乱が生じる可能性があります。

カスタム IP core が定義されたら、そのバンドルを生成してダウンロードします。

カスタム IP core バンドルの README ファイルの指示は Verilog / VHDL design flow に関連するものであり、無視する必要があります。代わりに、次の手順を実行する必要があります。

- カスタム IP core のファイル用の新しいディレクトリを作成します。このディレクトリの absolute path は、このカスタム IP core を使用する間は固定しておく必要があるため、誤って削除されないように配置することをお勧めします。

ダウンロードしたカスタム IP core バンドルをこのディレクトリに解凍します。

- Vivado で block design を開きます。
- 参照用に block design のビューを pdf ファイルとして保存します。block design の領域のどこかを右クリックし、“Save as pdf file...” を選択します。
- Tools メニュー (メイン メニュー バー) の下にある “Run Tcl Script...” を選択します。カスタム IP core バンドルが解凍されたディレクトリに移動し、xillybus_block サブディレクトリを入力します。 insertcore.tcl を選択します。
- script は、既存の Xillybus IP Core をカスタム IP core に置き換え、アプリケーションに関係のない配線の再接続も試みます。オブジェクトは、自動再編成により block design ダイアグラム内で移動することもあります。
- application logic の AXI Stream インターフェイスを更新された Xillybus IP core に接続します。
- script を実行する前に作成された pdf ファイルと比較し、必要に応じて修正します。

  application logic 関連の接続はどれも再接続されず、他の接続も失われている可能性があります。
- 以下および Xillybus IP Core の block の下のキャプションが新しい IP core の名前と一致することを確認します。

script は、blocks、ポート、およびインターフェイスを名前で検索することに注意してください。したがって、これらの名前がユーザーによって変更されている場合、接続の復元で部分的に (そしてサイレントに) 失敗する可能性があります。

この時点から、プロジェクトの implementation は以前と同じように実行できます。host 用の Xillybus の driver (Linux と Windows の両方) は、新しい IP core の構成を自動的に検出するため、カスタム IP core でも動作します。

したがって、カスタム IP core に交換した後、host に何かをインストールする必要はありません。

参考までに、insertcore.tcl script の実行手順は次のとおりです。

- カスタム IP core のディレクトリを Vivado の IP Catalog の IP Core repositories のリストに追加し、repositories の再スキャンを強制して、新しいカスタム IP Core が検出され、Catalog に追加されるようにします。

- 以前の Xillybus IP が存在する場合は、block design から削除します
- カスタム IP core を design に追加し、必要に応じてそのバージョンをアップグレードします
- 名前のリストを検索し、存在する場合はこれらの名前を持つすべてのポートを相互接続することにより、上の階層へのワイヤと stream_clk_gen の block へのワイヤの再接続を試みます。
- Zynq のみ: Xillybus IP core の bus アドレスをデフォルト値 (`0x50000000` で始まる 4 kB セグメント) に設定します。
- プロジェクトの synthesis 実行をリセットして、次の implementation に変更が反映されるようにします。

6

Vivado HLS 統合

6.1 概要

このセクションでは、単純な C function から IP block への compilation と、Xillybus の Block Design flow への統合方法を示します。

このセクションのベースとなるサンプル プロジェクトは、次の URL からダウンロードできます。

http://xillybus.com/downloads/hls-axis-starter-1.0.zip

ダウンロードしたファイルは、後の段階で移動できないため、Xillybus プロジェクトに簡単に関連付けられるディレクトリに解凍することをお勧めします。

サンプル プロジェクトでは、2 つの異なる種類の C ソースを区別することが重要です。

- 実行コード: コンピューターまたは embedded プラットフォーム (“host”) 上で実行され、他のコンピューター プログラムと同様に実行され、FPGA を使用して特定の操作をオフロードします。

  サンプル プロジェクトでは、サンプル ファイルは host/ サブディレクトリにあります。

- synthesis のコード: Vivado HLS による logic への変換を目的としています。

  サンプル プロジェクトでは、coprocess/example/src/main.c にあります。

一般的な C/C++ プログラミングとは異なり、host プログラムは synthesized function を呼び出しません。むしろ、機能を実行するために必要なデータをデータ構造に編成し、単純な API を使用して synthesized function に送信します。これについ

ては後で説明します。後の段階で、同様の API を使用して synthesized function から送信されたデータ構造として戻りデータを収集します。

6.2 HLS synthesis

このセクションで使用される C のサンプル コードは、セクション 6.4 で概説されています。

Vivado HLS を起動し、HLS プロジェクトを開きます。ウェルカム ページで “Open Project” を選択し、HLS プロジェクト バンドルが解凍された場所に移動し、“coprocess” という名前のフォルダーを選択します。

プロジェクトの部品番号を変更します。Solution >Solution Settings... >Synthesis を選択し、“Part Selection” を目的の FPGA に変更します。

Solution >Synthesis >Active Solution を選択して (またはツールバーの対応するアイコンをクリックして)、プロジェクト (“synthesize”) の compilation を開始します。いくつかの warnings (これは正常です) を含め、console には多くのテキストが表示されます。エラーは発生しません。

compilation の成功は、HLS の console タブの最後の数行に次のメッセージが表示されることで簡単に認識できます。

```
Finished C synthesis.
```

synthesis が成功した場合にのみ、synthesis レポートも console タブの上に表示されます。

Vivado HLS の詳細については、そのユーザー ガイド (UG902) を参照してください。

6.3 FPGA プロジェクトとの統合

Vivado HLS で、Solution >Export RTL を選択し、“IP Catalog” を Format Selection として選択します。 “Evaluate Generated RTL” の場合は Verilog を選択し、この下のチェックボックスはどちらもチェックしないでください。 OK をクリックします。

これには数分かかる場合があり、次のようなもので終了します

```
Finished export RTL.
```

Xillydemo プロジェクト (セクション 2.1 で設定) を Vivado (つまり Vivado HLS ではない) で開き、Block Design を開きます。 Xillinux (Zynq) を使用する場合は、“blockdesign” という名前の block を開きます。

次のように HLS IP block を追加します。block design ダイアグラム領域のどこかを右クリックし、“IP Settings...” を選択します。 “Repository Manager” タブで、repository を追加するための緑色のプラス記号をクリックします。セクション 6.2 で選択した同じ “coprocess” ディレクトリに移動して選択し、HLS プロジェクトを開きます。 Vivado は、1 つの repository が追加されたことを示すポップアップウィンドウで応答する必要があります。 “OK” ボタンを 2 回クリックして確認します。

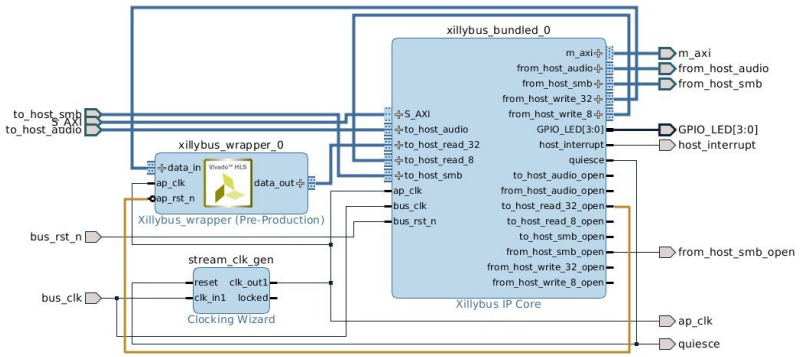
ここで、IP block を block design に追加します。もう一度、block design ダイアグラム領域のどこかを右クリックします。 “Add IP...” を選択し、リストから Xillybus_wrapper IP を選択します (検索ボックスに “wrapper” と入力すると、これが簡単になる可能性があります)。

xillybus_wrapper_0 という名前の新しい block が図に表示されます。 to_host_read_32 と from_host_write_32 の間の配線を外します (つまり、loopback を外します)。

次に、xillybus_wrapper block を次のように接続します。

- data_in と from_host_write_32
- data_out と to_host_read_32
- ap_rst_n と to_host_read_32_open
- ap_clk と ap_clk (Clocking Wizard の clk_out1 出力でもある)

結果は次のようになります (Xillinux ベースの block design の場合)。



ap_rst_n と to_host_read_32_open の間の接続は、xillybus_read_32 device file が host で開かれていない限り、logic を xillybus_wrapper の block 内でリセット状態に保ちます (ファイルが開かれていない場合、to_host_read_32_open は Low であり、

リセット入力はアクティブ Low です)。 host で実行されているソフトウェアがこの block との通信を試みる前にこの device file を開くと仮定すると、ソフトウェアが実行されるたびに logic からの一貫した応答が保証されます。

この時点で、implementation を実行して bitstream を取得できます。Vivado のウィンドウの下部で “Design Runs” タブを選択し、“synth_1” を右クリックして “Reset Runs” を選択します。 synth_1 のリセットを確認します。

次に、左側のバーで “Generate Bitstream” をクリックします。

6.4 synthesis コードの例

HLS が Xillybus でどのように機能するかを明確にするために、この例では三角関数のサインの計算と integer での簡単な操作を示します。これらは両方とも単純なカスタム関数 mycalc() でカバーされています。

coprocess/example/src/main.c は次のように起動します。

```
#include <math.h>
#include <stdint.h>

extern float sinf(float);

int mycalc(int a, float *x2) {
  *x2 = sinf(*x2);
  return a + 1;
}
```

いつものように、#include statements がいくつかあります。正弦関数には “math.h” の組み込みが必要です。

そして、“synthesized function” の役割を担う単純な機能、mycalc() があります。これは、floating point と integer の算術演算を示す非常に単純な関数です。 High-Level Synthesis Guide UG902 は、より有用なタスクを実装する方法に関する詳細情報を提供します。

main.c の次はラッパー関数 xillybus_wrapper() です。これは synthesized function と Xillybus の間のブリッジであり、データのパックとアンパックを行ったり来たりします。

この例の場合、host から data stream までの integer および floating point 形式の数値を受け入れます。これは、“data_in” 引数によって表されます。 “data_out” 引数を使用して、integer プラス 1 と floating point 数値の (三角関数) サインを返します。

```
void xillybus_wrapper(int *data_in, int *data_out) {
#pragma AP interface axis port=data_in
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

  uint32_t x1, tmp, y1;
  float x2, y2;

  // Handle input data
  x1 = *data_in++;
  tmp = *data_in++;
  x2 = *((float *) &tmp); // Convert uint32_t to float

  // Run the calculations
  y1 = mycalc(x1, &x2);
  y2 = x2; // This helps HLS in the conversion below

  // Handle output data
  tmp = *((uint32_t *) &y2); // Convert float to uint32_t
  *data_out++ = y1;
  *data_out++ = tmp;
}
```

xillybus_wrapper() は、2 つの pointers で宣言されており、どちらも int 型の変数に対して宣言されています。これらの関数引数は、block design に含めるために、将来の IP block の 2 つの AXI Stream ポートに変わります。それぞれに、“axis” 型のインターフェイスと見なす必要があることを HLS に通知する #pragma ステートメントがあります。

“#pragma AP” と “#pragma HLS” は互換性があります。前者は C Synthesizer の以前の名前 (Auto Pilot) に基づいており、後者は Xilinx の最近のドキュメントに記載されています。

“int” は HLS によって 32 ビット ワードと見なされるため、それぞれの AXI Stream インターフェイスには 32 ビット幅のデータ インターフェイスがあります。

もちろん、AXI Stream の入力と出力の任意のセットを取得するために、pragmas と同様に引数のリストを変更することも可能です。

ap_ctrl_none の pragma 宣言は、(存在しない) return value のポートを生成しないように compiler に指示します。

次に、“execution” のコードがあります。入力データがフェッチされます。各

*data_in++ 操作は、host から発信された 32 ビット ワードをフェッチします。示されているコードでは、最初の単語は unsigned integer として解釈され、x1 に入れられます。 2 番目のワードは 32 ビットの float として扱われ、x2 に格納されます。

次に、mycalc()、“synthesized function” への関数呼び出しがあります。この関数は戻り値として 1 つの結果を返し、2 番目のデータは x2 を変更することによって戻ります。

ラッパー関数は、x2 の更新された値を新しい変数 y2 にコピーします。これは、このコードの compilation が processor での実行を意図していた場合、冗長な操作のように見えるかもしれません。ただし、HLS を使用する場合、これは compiler が後で float への変換を処理するために必要です。これは HLS compiler のやや風変わりな動作を反映していますが、これは pointer を使用する際のデリケートな問題の 1 つです。メモリ アレイとそれに対する pointer が C コードで定義されていても、HLS compiler はそれらのいずれも生成しません。 pointer の使用は、私たちが達成したいことのヒントにすぎず、これらのヒントを少しプッシュする必要がある場合もあります。

最後に、結果が host に送り返されます。各 *data_out++ は、float からの適切な変換を使用して、32 ビット ワードをコンピュータに送信します。

*data_in++ および *data_out++ 演算子は実際には pointers を移動せず、基になるメモリ配列がないことに注意してください。むしろ、これらは AXI stream インターフェースとの間 (そして最終的には Xillybus streams との間) との間でデータを移動することを象徴しています。したがって、“data_in” および “data_out” 変数が使用される唯一の方法は、*data_in++ および *data_out++ です (High-Level Synthesis Guide は他の可能性、特に固定サイズの配列を提供します)。

また、このコードは logic に変換され、processor では実行されないため、これらの C コマンドの唯一の意味は、データの入力 stream が与えられた場合に期待される出力 stream のデータを生成することです。ただし、データがいつ出力されるかについての約束はありません (HLS のレポートに示されている可能性のある latencies の範囲を除いて)。

したがって、入力データの割り当ての順序は、入力データがどのように解釈されるかを強制するという意味で重要です。一方、送信される最初の出力である y1 は、最初に到着する入力である x1 のみに依存するため、2 番目の入力が到着する前に最初の出力が送信されることが許容されます。これは、コード実行の直感的なシーケンシャルな性質と矛盾しますが、全体的な結果は同じであるため、ハードウェア アクセラレーションのコンテキストでは意味がありません。

さらに、data_in AXI stream に常にデータが供給される場合、ラッパー関数 “runs”

は次のように繰り返します。

```
while (1) // This while-loop isn't written anywhere!
  xillybus_wrapper(data_in, data_out);
```

新しいデータは、*data_in++ コマンドによってできるだけ早くフェッチされ、logic の内部 pipeline (HLS のレポートによると、サンプル プロジェクトでは 70 ステージより長い) を満たす可能性が非常に高くなります。したがって、processor がコードを実行すると、単語のペアを取得して処理し、2 つの出力単語を発行してから 2 番目の単語のペアを取得するのとは異なり、HLS の解釈では、何かが出力される前に data_in で 70 単語を取得する可能性があります。 data_out AXI streamで。

6.5 synthesis 用の C/C++ コードの変更

追加の AXI Stream ポートは、例に示すように、ラッパー関数に引数を追加し、これらをインターフェイス ポートとして宣言することで作成できます。

もちろん、サンプル design の C コードに他の変更を加えることができます。

*data_in++ および *data_out++ で示されているのと同じスタイルで I/O を実装するか、他の可能性について High-Level Synthesis Guide (UG902) を参照することをお勧めします。また、コーディング テクニックを学ぶための推奨ソースでもあります。

重要:

変更を行った後、*Vivado* で *"Generate Bitstream"* をクリックしないでください。以下に詳述するように、*block* をアップグレードせずに *bitstream* の *implementation* を繰り返し起動すると、*bitfile* の *implementation* が成功したように見えますが、*HLS block* の古いバージョンに基づいています。

サンプル プロジェクトに変更を加えた後、セクション 6.2 の "HLS synthesis" からやり直して、Vivado で implementation まで進み、さらに Vivado で HLS block を更新します。

言い換えると：

- Vivado HLS: HLS でプロジェクトの compilation を実行します。 HLS synthesizer は、新しいファイルを開始する前に、以前の compilations によって生成されたファイルを常にクリーンアップします。
- Vivado HLS: IP Catalog bundle にエクスポートします。

- Vivado (Vivado HLS ではありません) で、xillybus_wrapper のブロックをアップグレードします (実際には、変更後に更新します): block design view を開き、ページの上部にある block のアップグレードが必要であるというメッセージに応答します。このメッセージが見つからない場合は、Tcl Console に “report_ip_status -name status” と入力します。下部にある “Upgrade Selected” ボタンをクリックします。これに続いて、アップグレードが成功したことを確認するダイアログ ボックスと、出力ファイルの生成を要求するダイアログ ボックスが表示されます。 2 番目のダイアログ ボックスで “Skip” をクリックします。
- Vivado: design runs が無効化されたことを確認します。Vivado のウィンドウの下部で、“Design Runs” タブを選択します。 synth_1 の Status 列に Synthesis Out-of-date と表示されます。
- Vivado: **design runs が無効化されていない限り**、次のことを試みます。 IP カタログを更新します。block design ダイアグラム領域のどこかを右クリックし、“IP Settings...” を選択します。 “Repository Manager” タブの下で、下部にある “Refresh All” ボタンをクリックします。同じダイアログ ボックスの “General” タブで “Clear Cache” をクリックする必要がある場合もあります。この後、xillybus_wrapper の block のアップグレードに戻ります。

  上記の前の項目で *design runs* が無効であることが判明した場合、これらのアクションは必要ありません。
- Vivado: synth_1 の実行をリセットします
- Vivado: bitstream を生成します。

6.6 simple.c: host プログラムの例

サンプル プロジェクトには、simple.c と practical.c の 2 つの C ファイルとしてサンプル host プログラムがあります。これらは、プロジェクトの host 側を示しています。

どちらも Linux host 用に書かれており、compilation 用に書かれています。

```
# gcc -O3 -Wall simple.c -o simple
```

ただし、Windows には簡単に適用できます (以下を参照)。

重要:

simple.c は、特に次の欠点があるため、実際の *host* プログラミング**の例として使用しないで**ください。

- 単一の要素のみが処理されます。関数呼び出しの *write()* と *read()* のペアでループすると、パフォーマンスが低下します。
- *write()* および *read()* 操作の戻り値は、適切な操作のためにチェックする必要があります。これは簡単にするために省略されていますが、プログラムの信頼性が低くなります。

セクション *6.7* では、より優れたコーディング手法について概説しています。

simple.c ファイルは #include statements で始まります。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

これに続いて、いくつかの変数の宣言とともに、main() 関数の古典的な宣言が続きます。

```
int main(int argc, char *argv[]) {
  int fdr, fdw;

  struct {
    uint32_t v1;
    float v2;
  } tologic, fromlogic;
```

struct 変数については以下で説明します。

プログラムは、named pipes のように動作し、logic との通信に使用される 2 つの device files (/dev/xillybus_read_32 および /dev/xillybus_write_32) を開くことから始まります。 Xillybus バンドルの設定から、これら 2 つのファイルが Xillybus の driver によって生成されることを思い出してください。

セクション 6.3 で指摘したように、ap_rst_n は block design ダイアグラムの

to_host_read_32_open に接続されているため、/dev/xillybus_read_32 を開くと logic がリセットされなくなります。これが、データ送信の前に両方のファイルが開かれる理由です。

```
fdr = open("/dev/xillybus_read_32", O_RDONLY);
fdw = open("/dev/xillybus_write_32", O_WRONLY);

if ((fdr < 0) || (fdw < 0)) {
  perror("Failed to open Xillybus device file(s)");
  exit(1);
}
```

次に、実際の実行に移ります。 “tologic” 構造体には、logic に送信するためのいくつかの値が取り込まれ、その後、メモリから xillybus_write_32 に直接書き込まれます。実際には、これは 8 バイト、より正確には 2 つの 32 ビット ワードを書き込みます。 1 つ目は tologic.v1 に入れられた整数 123 であり、2 つ目は tologic.v2 に入れられた float です。したがって、tologic 構造は、データの logic の期待に一致するように設定されました。最初の *data_in++ 命令によって 1 つの integer が、2 番目の命令によって 1 つの float が実行されます。

```
tologic.v1 = 123;
tologic.v2 = 0.78539816; // ~ pi/4

// Not checking return values of write() and read(). This must
// be done in a real-life program to ensure reliability.

write(fdw, (void *) &tologic, sizeof(tologic));
read(fdr, (void *) &fromlogic, sizeof(fromlogic));

printf("FPGA said: %d + 1 = %d and also "
       "sin(%f) = %f\n",
       tologic.v1, fromlogic.v1,
       tologic.v2, fromlogic.v2);
```

セクション 6.4 から、ラッパー コードが data_in stream から 2 つの 32 ビット ワードをフェッチすることを思い出してください。最初のワードは “x1” に、2 番目のワードは “tmp” に送られ、“tmp” はすぐに float に変換されます。これは、“tologic” 構造体の 2 つの 32 ビット要素と一致します。

これに続いて、FPGA からデータを読み戻します。 “fromlogic” にも同じ原則が適用されます。

simple.c は一般的なまとめで終わります:

```
  close(fdr);
  close(fdw);

  return 0;
}
```

/dev/xillybus_write_32 に送信されるデータの量を、ラッパー関数の *data_in++ 操作の数と一致させることが重要です。送信されるデータが少なすぎる場合、synthesized function はまったく実行されない場合があります。多すぎると、次の実行が失敗する可能性があります。

この例では、“tologic” と “fromlogic” に同じ構造形式が選択されていますが、これに固執する必要はありません。送受信されるデータがラッパー関数の *data_in++ および *data_out++ 操作の数と同期していることが重要です。

このプログラムの実行は、

```
# ./simple
FPGA said: 123 + 1 = 124 and also sin(0.785398) = 0.707107
```

最後に、次の調整のすべてまたは一部を行う必要がある Windows ユーザーへのメモ:

- ファイル名の文字列を “/dev/xillybus_read_32” から “\\\\.\\xillybus_read_32” に変更します (Windows の実際のファイル名は \\.\xillybus_read_32 ですが、escaping が必要です)。 2 番目のファイル名は “\\\\.\\xillybus_write_32” に変わります。
- unistd.h の #include ステートメントを io.h に置き換えます。
- open()、read()、write()、および close() への関数呼び出しを _open()、_read()、_write()、および _close() に置き換えます。

6.7 practical.c: 実用的な host プログラム

simple.c の例では、データ交換の概要が簡潔に示されていますが、実際のシステムではいくつかの変更が必要です。

次の違いが最も顕著です。

- 処理のために単一のデータ セットを生成するのではなく、構造体の配列が割り当てられて送信されます。同様に、データの配列が logic から受信されま

す。これにより、I/O overhead だけでなく、ソフトウェアとハードウェアによって引き起こされる latencies の影響も軽減されます。これは、ハードウェア アクセラレーションでパフォーマンスを向上させるための重要な方法です。

- プログラムは、データの書き込み用と読み取り用の 2 つのプロセスに分岐します。これら 2 つのタスクを独立させることで、どちらかの側で処理するデータが不足して処理が停止するのを防ぐことができます。この独立性は、threads (特に Windows) または select() 関数呼び出しを使用して実現できます。
- read() および write() 関数呼び出しは、信頼できる I/O を確保するために正しく行われます。この目的のために追加された while ループは扱いにくいように見えるかもしれませんが、これらの関数呼び出しの部分的な完了 (すべてのバイトが読み書きされるわけではない) に正しく応答する必要があります。EINTR エラーは、実行中のプロセスに誤って送信される可能性がある POSIX signals に適切に対応するために必要に応じて処理されます。

次に、practical.c の簡単なウォークスルーに進みます。まず、headers:

```
#include <stdio.h>
#include <unistd.h>

#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
```

同じ構造に加えて、データのチャンクあたりの要素数である N を定義します。

```
#define N 1000

struct packet {
  uint32_t v1;
  float v2;
};
```

一般的な main() 関数の定義といくつかの変数:

```
int main(int argc, char *argv[]) {

  int fdr, fdw, rc, donebytes;
  char *buf;
  pid_t pid;
  struct packet *tologic, *fromlogic;
  int i;
  float a, da;
```

以前のように開いたファイル:

```
  fdr = open("/dev/xillybus_read_32", O_RDONLY);
  fdw = open("/dev/xillybus_write_32", O_WRONLY);

  if ((fdr < 0) || (fdw < 0)) {
    perror("Failed to open Xillybus device file(s)");
    exit(1);
  }
```

実際の実行は、fork() から始まり、2 つのプロセスになります。

```
  pid = fork();

  if (pid < 0) {
    perror("Failed to fork()");
    exit(1);
  }
```

親プロセスは、処理のためにデータを準備し、それを FPGA に書き込みます。このプロセスでは使用されないため、read file descriptor を閉じます。開いたままにしておくと、両方のプロセスが file descriptor を閉じる (または終了する) まで、device file が開いたままになります。これは、ここでは望ましい動作ではありません。

```
  if (pid) {
    close(fdr);

    tologic = malloc(sizeof(struct packet) * N);
    if (!tologic) {
      fprintf(stderr, "Failed to allocate memory\n");
      exit(1);
    }
```

次に、構造体の配列にデータを入力します。これは、処理するデータのセットごとに構造を定義することが理にかなっている理由を説明しています。

```
// Fill array of structures with just some numbers
da = 6.283185 / ((float) N);

for (i=0, a=0.0; i<N; i++, a+=da) {
  tologic[i].v1 = i;
  tologic[i].v2 = a;
}

buf = (char *) tologic;
```

“buf” は、pointer から char の buffer として定義され、構造体の配列を指すことに注意してください。データを送信する while ループは buffer を送信用のデータのチャンクとして扱うため、この変換が必要です。

次に、データを書き込むための while ループ。不必要に複雑に思えるかもしれませんが、データが確実に書き込まれるようにするための最短の方法です。実際のアプリケーションでは、このコードをそのまま採用することをお勧めします。

```
donebytes = 0;

while (donebytes < sizeof(struct packet) * N) {
  rc = write(fdw, buf + donebytes,
             sizeof(struct packet) * N - donebytes);

  if ((rc < 0) && (errno == EINTR))
    continue;

  if (rc <= 0) {
    perror("write() failed");
    exit(1);
  }

  donebytes += rc;
}
```

この例では、1 つのチャンクのみが送信されます (相手側で受信されます)。実際のコードでは、上記の 2 つのコードをループするのが正しいです。

パフォーマンス テストでは、通常、32 kBytes のチャンク サイズで最良の結果が得られることが示されています。

この例では 1 つのチャンクのみが送信されるため、プロセスは終了します。ファイルを閉じる前に 1 秒間スリープすると、すべてのデータが logic から排出される前に logic がリセットされないことが保証されます。 ap_rst_n が to_host_read_32_open

に接続され、from_host_write_32_open がまったく接続されていないため、block design がセクション 6.3 に示されている場合、これは意味がありません。

それにもかかわらず、これは、すぐに終了する必要がない限り、file descriptor をすぐに閉じないという良い規則を示しています。これにより、プロジェクトがより複雑になったときに混乱を避けることができます。

```
    sleep(1); // Let the output drain

    close(fdw);
    return 0;
```

次に、同様の方法で開始する子プロセスがあります。

```
  } else {
    close(fdw);

    fromlogic = malloc(sizeof(struct packet) * N);
    if (!fromlogic) {
      fprintf(stderr, "Failed to allocate memory\n");
      exit(1);
    }

    buf = (char *) fromlogic;
```

繰り返しますが、これは device file からデータを読み取るための推奨される方法です。

```
    donebytes = 0;

    while (donebytes < sizeof(struct packet) * N) {
      rc = read(fdr, buf + donebytes,
                sizeof(struct packet) * N - donebytes);

      if ((rc < 0) && (errno == EINTR))
        continue;

      if (rc < 0) {
        perror("read() failed");
        exit(1);
      }

      if (rc == 0) {
        fprintf(stderr, "Reached read EOF!? Should never happen.\n");
        exit(0);
      }

      donebytes += rc;
    }
```

そして、データが印刷されます：

```
    for (i=0; i<N; i++)
      printf("%d: %f\n", fromlogic[i].v1, fromlogic[i].v2);

    sleep(1); // Let the output drain

    close(fdr);
    return 0;
  }
}
```

もう一度、プロセスは file descriptor を閉じる前に 1 秒間スリープしますが、この特定のケースでは必要ありません: file descriptor を閉じると実際に logic がリセットされますが、この場合はすべての出力がフェッチされているため無害です。この時点に到達するまでに。

前述のように、すぐに終了することが有益でない限り、この 1 秒間のスリープは、特に他の出力 streams がデバッグなどのために生成される場合に、混乱を避けることができます。

6.8 Design に関する考慮事項

6.8.1 複数の AXI streams での作業

サンプル プロジェクトは、各方向に 1 つの stream の基本的なケースを示しています。ただし、引数を AXI streams として宣言するための pragmas とともに、ラッパー関数に引数を追加することにより、IP block の入力および/または出力用に streams を追加するのは簡単です。

たとえば、1 つではなく 3 つの streams を入力します。

```
void xillybus_wrapper(int *d1, int *d2, int *d3, int *data_out) {
#pragma AP interface axis port=d1
#pragma AP interface axis port=d2
#pragma AP interface axis port=d3
#pragma AP interface axis port=data_out
#pragma AP interface ap_ctrl_none port=return

  *data_out++ = thefunc(*d1++, *d2++, *d3++);

}
```

セクション 5 で説明されているように、streams を Xillybus IP core に追加することは、カスタム IP core を構成することによって同様に簡単です。

追加の streams は、さまざまなシナリオで役立ちます。

- データとメタ情報を別々の streams で送信します。たとえば、データをパケットに分割する必要がある場合は、その長さを 1 つの専用 stream で送信し、データを別の stream で送信します。これにより、長さがわかる前にパケットの先頭を送信できます。
- 自然に別々に配置されたデータを送信します。たとえば、異なる画像のピクセル スキャン (これについては以下で詳しく説明します)。
- デバッグの場合: 検証のために中間データを host に送信します。

複数の streams を使用する場合は、それらすべてを念頭に置くことが重要です。logic の実行フローは、入力 stream にデータがない場合、または出力 stream のそれぞれの device file が開かれていない場合 (またはデータで overflow に苦しむ場合) に停止する可能性があります。これは、出力 stream がデバッグ用である場合に特に重要です。システムを通常の操作に使用する場合、デバッグ用の stream を忘

れがちです。この stream からのデータは消費されないため、通常は数回のデータサイクルの後、混乱を招く実行停止が発生します。

logic ハードウェアに、一見不適切に見える方法でデータを供給することは、多くの場合賢明です。たとえば、上記の 3 入力の例は、操作ごとに 3 つのデータ要素を必要とする画像処理アルゴリズムに役立ちます。画像が左から右、上から下にスキャンされるとします。ピクセル出力を生成するために、アルゴリズムは、現在の画像のピクセルと共に、2 つの前の画像からのそれぞれのピクセルを必要とします。このような場合、1 つの stream を介して現在の画像を FPGA に送信し、別の 2 つの streams を介して前の 2 つの画像を並行して送信することができます。

これは、I/O データ帯域幅の浪費と多くの不必要なメモリ コピーのように見えるかもしれません。特に、“shuffling data” に processor が大きく関わっているのは違和感があるかもしれません。主観的な認識はさておき、メモリ コピーの実装は、最新のすべての processor アーキテクチャで高度に最適化されたタスクであり、processor には多くの場合、他のアプリケーション関連のタスクがロードされるため、メモリ コピーの負荷は無視できます。

そのため、logic にデータを直接供給することは、リソース使用率の観点からは最適ではありませんが、processor の余分な負荷は通常、処理する他の負荷の高いタスクがあることを考えると、通常はかなり小さくなります。これは、多くの場合、design を大幅に簡素化するための妥当な価格です。

6.8.2 application clockの周波数

HLS によって生成された logic は、stream_clk_gen の block によって生成された block design の application clock によって駆動されます。この clock は logic のタイムベースであるため、その実行速度は clock の周波数に比例します。 AXI stream ポートのデータ転送がボトルネックにならない限り、application clock の周波数が高くなると、それに比例して処理スループットが高速化されます。

ただし、FPGA の logic リソースと、必要なタスクを実装するためにそれらがどのように利用されているかに応じて、application clock の周波数がどれだけ高くなるかに制限があります。これらは、design プロセスに関連するマイルストーンです。

1. Vivado HLS を使用すると、clock の目的の周波数を design に設定し、application clock の目的の周波数を指定できます (Solution >Solution Settings を使用)。このパラメーターは、HLS によって単にヒントとして使用され、必要かつ可能な場合に、より高速な logic を生成するための特別な努力を行うことができます。
2. Vivado HLS が compilation を終了すると、HLS の GUI の Synthesis タブの

“Performance Estimates” セクションの “Timing” の下で、達成可能と思われる clock の周波数の推定値が表示されます。

3. ユーザーは、セクション 3.2 (特にセクション 3.2.2) で説明されているように、Vivado の block design で application clock の周波数を設定します。自然な選択は、項目 2 で推定した clock の周波数、またはそれ以下です。これは Vivado HLS ではなく Vivado で行われることに注意してください。

4. Vivado が design 全体の implementation を FPGA 用の bitstream に仕上げると、clock に関連するすべての要件を満たすように logic を編成することに成功したかどうかがユーザーに通知されます。これには、項目 3 で設定した clock の周波数を満たすことが含まれます。

つまり、最後のマイルストーンに要約され、項目 3 で選択された application clock の周波数に関連する Vivado が timing constraints を満たすことができた場合.

HLS および stream_clk_gen のデフォルトの clock period は 10 ns (100 MHz) です。次の場合を除き、多くの場合、この選択を続けることが最善です。

- Vivado は timing constraints に適合しません。この場合、低速の clock を選択する必要があります。
- 処理スループットを向上させる動機がある場合は、より高速な clock を要求する試みを行う必要があります。これは、多くの場合、clock の周波数を調整し、design 自体と HLS pragmas を変更して結果を改善する反復プロセスです。

6.8.3 logic のリセット

C/C++ コードは logic に変換されるため、実際には実行されず、独自の実行フローの状態が維持されます。 logic が processor のプログラム実行の動作を模倣するためには、実行がプログラムの最初から開始されることを確認することが不可欠です。これは、logic をリセットすることによって実現されます。

ほとんどの場合、直観的な動作は、host のプログラムが実行を開始すると、FPGA のプログラムが最初から開始することです。 host で実行されるプロセスは、アクセスする前に device files を開き、これらのファイルは少なくともプロセスが終了したときに必ず閉じられるため、1 つ以上の device files が閉じられたときに logic をリセットするのは自然なことです。

Xillybus IP Core 内の各 stream には *_open ポートがあり、それぞれの device file が開かれているときはハイ (’1’) です。 HLS block にはアクティブ ロウのリセット入

力 ap_rst_n (デフォルト) があるため、*_open 出力を ap_rst_n 入力に直接接続すると、望ましい結果が得られます。ファイルが閉じられると、*_open 信号はローになります ('0')。これにより、logic がリセット状態に保持されます。

すべての device files が開かれるまで、またはいずれかが開かれるまで logic を保持するために、いくつかの *_open ポートを組み合わせることが望ましい場合があります。これは、Vivado の IP catalog で利用できる単純な logic gate blocks を追加することによって実現されます。リセット信号を生成する方法の選択は、host プログラムのセットアップ方法によって異なります。

いずれにせよ、host が必要に応じて device files を開くまで HLS block とデータ交換を試みないようにして、リセット信号が非アクティブになるようにすることが重要です。簡単にするために、HLS block とのデータ交換を開始する前に、HLS block に関連するすべての device files を開き、クリーンアップのためにすべてを閉じることをお勧めします。