(機械で日本語に翻訳)

Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.1

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 This document may also be slightly outdated in relation to the original.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.

1

序章

Xillybus の IP cores は、FIFO または dual-port block RAM を介して user application logic とインターフェースすることを目的としています。したがって、通常は API を詳しく理解していなくても IP core を使用することができます。

API のルールの大部分は、次のような単純な原則から推測できます。 user application logic は、FIFO または block RAM とまったく同じように動作する必要があります。これは、application logic が IP core と直接接続する場合にも当てはまります。

特に、この原則は、Xillybus IP core と user application logic の間のデータ交換が IP core によって規定されたペースで行われることを意味します。データ フローは一時的に停止し、後で予測できない方法で再開される場合があります。他の状況では、データ フローは継続的になります。 IP core を FIFO または block RAM に直接接続する場合、これは問題になりません。同様に、IP core と直接接続する user application logic も、データ フローが予期しない方法で開始および停止した場合でも適切に動作する必要があります。

IP core の不規則なアクセス パターンをバグと考えるのはよくある間違いです。 IP core と FIFO の間のデータ フローで説明できない一時停止が発生すると、不審に思えるかもしれませんが、誤動作を示すものではありません。

logic consumption を減らすために、IP core と直接接続する (つまり、FIFO または block RAMs を使用しない) ことはお勧めできません。これは特に design の初期段階に当てはまります。 user application logic が IP core に直接接続されている場合、不規則なデータ フローにより application logic のバグが露呈する可能性があります。

このガイドでは、application logic との直接インターフェイスに関連する API について説明し、一般的なアプリケーションについても詳しく説明します。

このガイドで説明する詳細を掘り下げる前に、Xillybus の初期体験を積むことをお

勧めします。次のドキュメントを参照してください。

- Getting started with the FPGA demo bundle for Xilinx
- Getting started with the FPGA demo bundle for Intel FPGA
- Getting started with Xillinux for Zynq-7000
- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

また、このガイドは、同期 streams と非同期 streams の違いを理解していることを前提としています。これについては、次の 2 つのドキュメントのいずれかのセクション 2 で説明されています。

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

XillyUSB IP cores は同じ API を公開しており、Xillybus IP cores のサブセットです。したがって、特に断りのない限り、このガイドでは “Xillybus” という名前は XillyUSB IP cores も指します。

興味のある方は、Xillybus host application programming guide for Linux または Xillybus host application programming guide for Windowsの付録 A に Xillybus の実装方法に関する簡単な説明があります。

2

一般的なガイドライン

2.1 Clocking

Xillybus IP core との間のすべての signals は、bus_clk の rising edge と同期する必要があります。この clock は IP core によって提供されます。

PCIe に基づく Xillybus IP cores の場合、この clock は PCIe ブロックによって生成されます。 clock の周波数はプラットフォームによって異なります。 baseline IP core (リビジョン A) の場合、bus_clk の周波数は 62.5 MHz、125 MHz、または 250 MHz のいずれかです。これは、最大帯域幅 (公表されているとおり) がそれぞれ 200 MB/s、400 MB/s、または 800 MB/s であるかどうかによって異なります。

後のリビジョン (B、XL および XXL) では、bus_clk の周波数は 250 MHz です。

Zynq ベースのプラットフォームには通常、100 MHz の bus_clk があります。XillyUSB は、125 MHz の bus_clk で動作します。

多くの場合、限られた選択肢のリスト内で clock の周波数を変更できる可能性があります。これは、clock を生成する PCIe ブロックまたは processor core を構成することによって行われます。

PCIe ブロックの timing constraints が (demo bundles のように) 正しく設定されている場合、bus_clk に依存する application logic は適切な timing constraints によってもカバーされます。 このツールは、PCIe ブロックの timing constraints に基づいて、bus_clk 用の timing constraints を自動的に作成します。 XillyUSB だけでなく Zynq ベースのプラットフォームにも同じことが当てはまります。

これは、application logic が bus_clk と同期する必要があるということではありません。同様に、データのソースまたはデータの宛先が bus_clk と同期している必要はありません。別の clock が関係する場合は、dual-clock FIFO が IP core と一緒に使用されることがよくあります。 FIFO の片側は Xillybus IP core に接続されていま

す。したがって、こちら側は bus_clk と同期します。 application logic は FIFO の反対側に接続されています。こちら側はapplication logicのclockと同期しています。したがって、FIFO は短期の一時ストレージとしてだけでなく、clock domain crossing としても使用されます。

2.2 データ幅

各 FIFO またはメモリ インターフェイスは、8 ビット、16 ビット、または 32 ビット幅のデータを処理します。これは、ベースライン Xillybus IP cores (リビジョン A) に当てはまります。 XillyUSB と同様に、それ以降のリビジョンでは、より広範なデータ インターフェイスがサポートされています。

データ幅が広いほど、帯域幅のパフォーマンスが高くなり、本来の伝送ワードが 8 ビットよりも広いアプリケーションでより便利になります。一方、host 側の固有のデータ幅は 8 ビット (1 バイト) のままです。これは、read() および write() 関数呼び出しがそれらの長さをバイト単位で定義するためです。

データ幅を選択する際の考慮事項については、The guide to defining a custom Xillybus IP coreで簡単に説明しています。

2.3 FIFO を介したインターフェース

demo bundle は、FIFO の接続方法を示しています。 FIFO の両側が IP core に接続されています。これは、2 つの streams に loopback を実装します。

demo bundle の FIFOs は、両側で共通の clock 用に構成されています。これは、FIFO を clock domain crossing に使用する場合には適していません。この場合、dual-clock FIFO (“asynchronous FIFO” と呼ばれることが多い) を使用する必要があります。

hostからFPGAまでstreamにFIFOを使用する場合、このFIFOの“full”信号をXillybus IP coreに接続する必要があります。 IP Core は、この信号を使用して、バーストデータ転送を開始できるかどうかを判断します。

同じ原則が stream から FPGA までの host にも当てはまります。 同じ目的で、“empty” 信号を Xillybus IP core に接続する必要があります。 IP core は、(FWFT、First Word Fall Through とは対照的に) 通常の FIFO の動作を期待します。

バーストが開始されると、Xillybus IP core は引き続き次の信号 (“empty” および “full”) に依存します。 これらの信号は、IP core が空の FIFO から読み取ったり、完全な FIFO に書き込んだりすることを防ぎます。

ただし、FIFO がデータのバーストの準備ができていることを示していても、Xillybus

IP core はバーストをすぐに開始しない場合があります。 FIFO でバーストの継続が許可されている場合でも、IP core はデータ バーストを途中で停止する場合があります。データ フローのパターンが一見ランダムであるのは正常です。

一般的なルールとして、Xillybus IP core は接続されているすべての FIFOs に平等にサービスを提供しようとします。これらの FIFOs は “empty” または “full” を頻繁にアクティブ化しないため、IP core は、より早くいっぱいになる傾向がある FIFOs に長いバーストを許可します。

このシンプルな調停方法により、すぐにいっぱいになる傾向にある FIFOs との効率的な通信が保証されます。同時に、より低いレートでデータを受信する FIFOs 上の低 latency が実現されます。

FIFO の深さに関しては、Xillybus IP core は原則としてどの深さでも動作します。ただし、この属性は、予想されるデータ フローに対処するために選択する必要があります。 2 kBytes の深さを持つ FIFO は、データ レートが高い場合でも、ほとんどの場合、非同期 stream にとって正しい選択です。しかし、これには試行錯誤が必要な場合もあります。

overflow または underflow が発生するほど長い期間、Xillybus core がこのサイズの FIFO を無視する可能性は低いため、通常は 2 kBytes の深さがあれば十分です。もちろん、これは、host 上で実行される user application software が十分な速度でデータを消費または供給する限り当てはまります。そうでない場合、解決策は DMA buffers を大きくすることかもしれません。 FPGA にはメモリがはるかに少ないため、より大きな FIFO でこれを解決しようとするのは無理があります。

2.4 “empty” および “full” 信号の動作

正常に動作している FIFO では、read enable が High になった後、clock cycle が 1 つだけ “empty” 信号が Low から High に変化する可能性があります。同様に、“full” 信号は、write enable が High になった後、clock cycle が 1 回だけ Low から High に変化する可能性があります。

もちろん、これら 2 つの信号はいつでも Low に変化する可能性があります。

Xillybus IP core は、次の動作に依存しています。 FIFO がデータ転送の準備ができていることを示すと (該当する場合、“empty” または “full” が低い場合)、IP core 内の state machine が一連のイベントを開始することがあります。これにより、少なくとも 1 つのデータ要素が転送されます。したがって、IP core が FIFO からデータをフェッチする前に “empty” 信号が High に変化すると、IP core が 1 回の clock cycle 中に “empty” 信号を無視する可能性があります。このようなイベントは、IP core 自体の整合性に関しては無害ですが、予期しない予測不可能なデータ フローにつなが

る可能性があります。

同じことが “full” 信号にも当てはまります。 IP core が FIFO にデータ ワードを書き込む前にこの信号が Low から High に変化すると、IP core は 1 回の clock cycle 中に “full” 信号を無視する可能性があります。繰り返しますが、これは IP core 自体には無害ですが、1 つのデータ ワードが失われる可能性があります。

適切に設計された FIFO は、Xillybus IP core との通信の準備が整うと同時にリセットされた場合にのみ、この障害状態を引き起こす可能性があります。この状況は通常、とにかく避けるべきです。

application logic が (FIFO なしで) IP core に直接接続されている場合、“empty” または “full” に関して標準の FIFO の動作を模倣することが重要です。

3

信号の説明

3.1 FPGA シグナルの命名規則

2 つのグローバル信号 bus_clk と quiesce を除き、すべての信号は単純な規則に従います。たとえば、write enable 信号には user_w_write_32_wren という名前が付いている場合があります。この名前は 4 つのコンポーネントに分かれています。

1. “user” プレフィックスは、すべてのユーザー インターフェイス信号に共通です。
2. “w” の部分は、この信号が host から FPGA (host “write”) までの stream に属することを示します。 FPGA から host までの Streams には、代わりに “r” が搭載されています。アドレス信号は両方向に適用されるため、この部分はありません。 “w” か “r” の選択に関しては、host の視点が取られていることに注意してください。
3. “write_32” 文字列は、関連する device file の名前に表示されます。該当する場合、/dev/xillybus_write_32 または /dev/xillyusb_00_write_32。
4. サフィックスは信号の意味を表します。

このセクションの残りの部分では、混乱を避けるために device file 名 (3 番目のコンポーネント) を {devfile} と表記します。

各信号名の後には、その信号が IP core への入力であることを示す (IN) が続き、IP core からの出力である場合は (OUT) が続きます。

3.2 hostFPGA伝送用信号

- user_w_{devfile}_data (**OUT**) – この信号には、書き込みサイクル中のデータが含まれます。
- user_w_{devfile}_wren (**OUT**) – この信号は、FIFO への write enable 信号です。 FIFO (または FIFO の動作を模倣するその他の logic) に書き込む必要がある有効なデータが user_w_{devfile}_data 信号にある場合、これは High です。
- user_w_{devfile}_full (**IN**) – この信号は、これ以上データを書き込むことができないことを IP core に通知します。

  重要: 'full' 信号は、書き込みサイクル後に clock cycle でのみ Low から High に変化する場合があります。すべての標準 FIFOs はこのように動作するため、このルールは IP core が application logic に直接接続されている場合 (つまり、中間に FIFO がない場合) にのみ関係します。

  このルールの理由は、Xillybus IP core が低 'full' 信号を青信号として扱い、host からのデータ転送を開始するためです。この規則に従わないと、'full' 条件を無視する散発的な書き込みが発生する可能性があります。

  'full' シグナルの典型的な Verilog 実装は、次のようになります。

```
always @(posedge bus_clk)
  if (ready_to_get_more_data)
    user_w_mydevice_full <= 0; // Turn low any time
  else if (user_w_mydevice_wren && { ... some condition ... } )
    user_w_mydevice_full <= 1; // Only in conjunction with wren
```

  VHDL でも同じ:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_get_more_data = '1') then
      user_w_mydevice_full <= '0'; -- Turn low any time
    elsif (user_w_mydevice_wren = '1' and { some condition } )
      user_w_mydevice_full <= '1'; -- Turn high only with wren
    end if;
  end if;
end process;
```

- user_w_{devfile}_open (**OUT**) – host 上の関連する device file が書き込み用にオープンされている場合、この信号は High になります (ファイルが読み取り

専用でオープンされており、許可されている場合、この信号は変更されません)。この信号は、ファイルが閉じられているときに FIFO または他の logic をリセットするために使用できます (active low reset として使用)。

host 上の複数の processes によってファイルが開かれた場合 (たとえば、fork() 関数の呼び出しの結果として)、この信号はすべての processes がファイルを閉じるまで High のままになります。

3.3 FPGAhost伝送用信号

- user_r_{devfile}_data (**IN**) – この信号には、読み取りサイクル中のデータが含まれます。この信号は、FIFO が変更する場合にのみ変更が許可されます。言い換えれば、user_r_{devfile}_rden が High になった後、clock cycle でのみ変更される可能性があります。
- user_r_{devfile}_rden (**OUT**) – この信号は、FIFO への read enable 信号です。この信号が High の場合、user_r_{devfile}_data には後続の clock cycle 上の有効なデータが含まれている必要があります。
- user_r_{devfile}_empty (**IN**) – この信号は、これ以上データを読み取ることができないことを core に通知します。

  重要: 'empty' 信号は、読み取りサイクル後に clock cycle でのみ Low から High に変化する場合があります。すべての標準 FIFOs はこのように動作するため、このルールは IP core が application logic に直接接続されている場合 (つまり、中間に FIFO がない場合) にのみ関係します。

  このルールの理由は、Xillybus IP core が低い 'empty' 信号を青信号として扱い、host へのデータ転送を開始するためです。この規則に従わないと、FIFO が空であることを無視する散発的な読み取りが発生する可能性があります。

  'empty' シグナルの典型的な Verilog 実装は、次のようになります。

```
always @(posedge bus_clk)
  if (ready_to_give_more_data)
    user_r_mydevice_empty <= 0; // Turn low any time
  else if (user_r_mydevice_rden && { ... some condition ... } )
    user_r_mydevice_empty <= 1; // Turn high only with rden
```

  VHDL でも同じ:

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if (ready_to_give_more_data = '1') then
      user_r_mydevice_empty <= '0'; -- Turn low any time
    elsif (user_r_mydevice_rden = '1' and { some condition } )
      user_r_mydevice_empty <= '1'; -- Turn high only with rden
    end if;
  end if;
end process;
```

- user_r_{devfile}_eof (**IN**) – この信号は、core に end-of-file を生成するように指示します。 'eof' が高くなると、core は FIFO から読み取れなくなります (つまり、ファイルが閉じられて再度開かれるまで、user_r_{devfile}_rden は低く保たれます)。

  host では、application software は、この信号が High になる前に IP core が受信したすべてのデータの読み取りを完了します。その場合にのみ、host は read() 関数を呼び出したときに EOF を受け取ります。

  application software によって読み取られていないデータがまだある場合、'eof' 信号は host で EOF をすぐには引き起こさないことに注意してください。 host での EOF の配信は、常識に基づいて行われます。つまり、すべてのデータが host によって読み取られた後です。

  'eof' 信号がハイになった後は、その後ハイを維持するかローに変化するかは問題ではありません。 IP core は、ファイルが閉じられるまで EOF 要求を記憶します。 'empty' 信号に関しては、'eof' がハイになると同時にハイに変化しても問題ありません。実際、'eof' が高い瞬間から、ファイルが閉じられるまで、'empty' 信号はまったく問題になりません。

  'empty' 信号と同様に、'eof' 信号は読み取りサイクル後に clock cycle でのみ High に変化することが許可されます。ただし、例外が 1 つあります。 'empty' 信号がすでにハイの場合、'eof' はいつでもハイに変更できます。この例外を使用して、host で read() 関数呼び出しを即座に終了させることができます (データの待機中にスリープ状態になる場合)。

  このルールに従わずに 'eof' を高に変更すると、EOF が生成されますが、正確に機能しない可能性があります。 EOF の直前で一部のデータが失われたり、EOF の前、または EOF の後に無関係なデータが追加されたりする可能性があります (したがって、application software は EOF の後にデータを受信しますが、これは違法です)。

'eof' がこのルールに準拠していることを確認する 1 つの方法は、'eof' を combinatorial function の出力として定義することです。 Verilog では、次のように記述できます。

```
assign user_r_mydevice_eof = user_r_mydevice_empty && [ ... ];
```

または VHDL では:

```
user_r_mydevice_eof <= user_r_mydevice_empty and [ ... ];
```

この方法では、'empty' が Low の場合、'eof' 信号は常に Low になります。

- user_r_{devfile}_open (**OUT**) – host 上の関連する device file が読み取り用にオープンされている場合、この信号は High になります (ファイルが書き込み専用でオープンされている場合、許可されている場合、この信号は変更されません)。この信号は、ファイルが閉じられているときに FIFO または他の logic をリセットするために使用できます (active low reset として使用)。

  host 上の複数の processes によってファイルが開かれた場合 (たとえば、fork() 関数の呼び出しの結果として)、この信号はすべての processes がファイルを閉じるまで High のままになります。

  'eof' 信号と 'open' 信号の間には直接的な接続はありません。 'open' 信号は、'eof' に関係なく、host でファイルが閉じられると Low に変化します。ただし、application software は通常、ファイルを閉じることで EOF に応答することに注意してください。したがって、これらの信号間に接続があると誤って信じてしまいがちです。

3.4 メモリ インターフェイス信号

Xillybus device file はアドレス信号を持つように構成できます。 application software は、ファイル (例: lseek()) で seeking の標準 API を使用して、この信号に値を割り当てます。また、アドレスの increment は、読み取りサイクルと書き込みサイクルの結果として FPGA 上で自動的に発生します。

標準の block RAM は IP core に簡単に接続できます。これは、FIFOs に関連してすでに説明した信号と、以下で詳しく説明する信号を使用して行われます。その結果、block RAM のメモリ アレイがファイルとして host で利用できるようになります。 ファイルに対する読み取りおよび書き込み操作は、メモリ配列に対する読み取りおよび書き込み操作になります。 host は、単一のメモリ要素またはメモリ アレイのセグメントにアクセスできます。 これは、読み取りまたは書き込み操作の長さによって異なります。

FPGA 上で block RAM のように動作する registers の配列を実装することも可能です。そうすることで、これらの registers は host から簡単にアクセスできるようになります。

'empty' および 'full' 信号は、wait states を必要とするメモリの読み取りおよび書き込み操作を遅くするため、または操作を一時的に遅らせる別の理由がある場合に使用できます。

メモリ インターフェイスには、次の 2 つの追加信号が必要です。

- user_{devfile}_addr (**OUT**) – この信号には現時点のアドレスが含まれます。read enable が High の場合、このアドレスからの読み取り操作が必要です。write enable が High の場合、このアドレスへの書き込み操作が必要です。この信号を block RAM の address input に直接接続すると、期待どおりに動作します。この信号の幅は最大 32 ビットまで設定可能です。

  最大アドレスでの読み取りまたは書き込み操作の後、アドレスの値はゼロに戻ります (アドレス信号の幅に応じて)。 lseek() への関数呼び出しの値が範囲外の場合、LSBs のみがこの信号にコピーされます。

- user_{devfile}_addr_update (**OUT**) – この信号は、host 上の lseek() への関数呼び出しの結果として、1 回の clock cycle の間 High になります。 'addr' 信号の値が更新されているため、'update' 信号は同じ clock cycle 上で High になります。

  この信号の目的は、アドレスの更新の結果、読み取り用のデータを準備する時間が必要であることを application logic に示す機会を与えることです。これは、そのような更新に応答して 'empty' 信号を High に変更することによって行われます。

  この目的のために、'empty' が読み取りサイクル後に 1 つの clock cycle のみを High に変更できるという規則には 1 つの例外があります。 'update' が High になった後の clock cycle で High に変更することもできます。

  したがって、次の Verilog コードは正しいです。

```
always @(posedge bus_clk)
  if ( { ... memory is ready ... } )
    user_r_mydevice_empty <= 0;
  else if ((user_mydevice_addr_update) &&
           ( user_mydevice_addr > { ... some limit ...} ))
    user_r_mydevice_empty <= 1;
```

  VHDL でも同じです。

```
process (bus_clk)
begin
  if (bus_clk'event and bus_clk = '1') then
    if ( { ... memory is ready ... } ) then
      user_r_mydevice_empty <= '0';
    elsif (user_mydevice_addr_update = '1'
          and user_mydevice_addr > { ... some limit ...} )
      user_r_mydevice_empty <= '1';
    end if;
  end if;
end process;
```

'empty' はいつでも Low に変更できるため、(アドレスに関係なく) アドレスが更新されるたびに 'empty' を High に変更し、'empty' を Low に戻すことができるかどうかを logic に評価させるのが合理的であることに注意してください。

'full' 信号も同様の方法で高に変化する可能性がありますが、これが役立つ理由は明らかではありません。

host 上の関連する device file が閉じられると (つまり、user_w_{devfile}_open と user_r_{devfile}_open が両方とも Low のとき)、アドレスがリセットされるため、その値はゼロに変わります。ただし、これはアドレス更新とはみなされないことに注意してください。つまり、user_{devfile}_addr_update は Low のままです。

3.5 quiesce 信号

host が IP core が完全に非アクティブ (quiescent state) であることを予期している場合、quiesce 信号は High です。これは通常、次の場合です。

- host が driver をまだロードしていないか、host がアンロードしています。
- Windows の場合: host が hibernation に入ろうとしているとき。
- XillyUSB の場合: また、デバイスがコンピューターにまったく接続されていない場合も同様です。

この信号の目的は synchronous reset として使用することですが、おそらくこの信号は必要ありません。 IP core が非アクティブ状態 (つまり quiescent state) にあるときは、すべてのファイルが閉じられます。したがって、application logic は *_open 信号のみを reset 信号として利用できます。 'quiesce' 信号は、reset のよりグローバルな形式として使用できます。

4

data acquisitionの実装

4.1 序章

FPGA からコンピュータにデータをキャプチャする必要がよく発生します。たとえば、次のとおりです。

- video 信号源からの Frame grabbing。
- アナログ - デジタル コンバーター (ADC) からのデータ。
- FPGA からデバッグ情報を受信します。

このようなアプリケーションではデータ速度が高くなる可能性があります。ただし、データ フローの連続性は保証される必要があります。 データの損失は許されません。

data acquisition アプリケーションは、データを FIFO に書き込むことで、Xillybus で簡単に実装できます。このセクションでは、host に到着するデータが連続していることを保証する方法に焦点を当てます。

理論的には、周辺機器とコンピュータの間で持続的なデータ レートを確保することは不可能です。オペレーティング システムが application software から CPU を必要なだけ奪う可能性があるためです。

それでも、連続した stream データを維持する方法はあります。この目標を達成するための最初の明白な条件は、必要な速度でデータを転送できる Xillybus stream を使用することです。それに加えて、特定の host programming テクニックを使用する必要があります。この問題は、次の両方のプログラミング ガイドで広く説明されています。

- Xillybus host application programming guide for Linux

- Xillybus host application programming guide for Windows

特に、これら 2 つのガイドのセクション 4 に注意してください。このセクションでは、高いデータ レートを扱う方法について説明しています。

高帯域幅のアプリケーションについては、これら 2 つのガイドのいずれかのセクション 5 を参照することもお勧めします。セクション 5 には、注意すべきいくつかのトピックが含まれています。

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

ただし、design が完全に実行されたとしても、stream のデータの連続性が壊れる可能性が常にあります。 オペレーティング システムの性質上、長期間にわたって CPU を application software から奪うことが許可されています。

したがって、最初の目標は、stream のデータの連続性が事実上決して損なわれないようにすることです。 2 番目の目標は、あらゆる努力にもかかわらずこの問題が発生した場合に、このイベントが確実に認識されるようにすることです。さらに重要なのは、host に到着するすべてのデータが連続していることが保証されていることです。

この 2 番目の目標を達成するには、application logic は連続性が切断された時点でデータの流れを停止する必要があります。この時点の後で、host に問題が発生したことを伝えるために、EOF が host に送信されます。このようにして、application software は、到着したデータが実際に連続していることを信頼できます。

理想的には、この停止メカニズムは決してアクティブにならないようにしてください。しかし、そうなったとき、それは問題を認識し、それを解決する機会を与えます。

以下では、Xillybus を使用して連続ソースからデータをキャプチャする方法を示します。このセクションでは、host に到着するすべてのデータがデータ ソースの信頼できるコピーであることを確認することに重点を置いています。

4.2 サンプルコード

以下に示して説明するコード例は、この link からモジュールとしてダウンロードできます。

http://xillybus.com/downloads/xillycapture.zip

zip ファイルは、xillycapture.v と xillycapture.vhd の 2 つのファイルで構成されます。これらはそれぞれ Verilog と VHDL で書かれています。この例を試すには、xillydemo.v または xillydemo.vhd を編集します。 demo bundle 内の read_32 に関連する信号を切断し、代わりにサンプル コードを挿入します。

このサンプル コードは、標準 dual clock FIFO の instantiation を実行します。このFIFOの幅は32 bitsです。サンプル コードの synthesis を実行する前に、ツール (Vivado または Quartus など) を使用してこの FIFO を生成します。この FIFO の名前は async_fifo_32 になるはずです。深さは 512 ワードで十分です。

サンプル コードには、“slowdown” という名前の信号があることに注意してください。この信号の目的は、偽のデータ ソースのデータ レートを下げることです。実際のデータ ソースを使用する場合は、この信号を削除する必要があります。

4.3 FIFO 接続

データ ソースが capture_clk と同期していると仮定します。したがって、データは通常の方法で標準の dual-clock FIFO に接続されます。この FIFO は、データ ソースと Xillybus IP core の間を接続します。

Verilog では:

```
async_fifo_32 fifo_32
  (
   .rst(!user_r_read_32_open),
   .wr_clk(capture_clk),
   .rd_clk(bus_clk),
   .din(capture_data),
   .wr_en(capture_en),
   .rd_en(user_r_read_32_rden),
   .dout(user_r_read_32_data),
   .full(capture_full),
   .empty(user_r_read_32_empty)
   );
```

そして VHDL では:

```
fifo_32 : async_fifo_32
  port map(
    rst         => reset_32,
    wr_clk      => capture_clk,
    rd_clk      => bus_clk,
    din         => capture_data,
    wr_en       => capture_en,
    rd_en       => user_r_read_32_rden,
    dout        => user_r_read_32_data,
    full        => capture_full,
    empty       => user_r_read_32_empty
    );

reset_32 <= not user_r_read_32_open;
```

これは demo bundle と非常によく似ています。 FIFO はファイルを閉じるとリセットされ、user_r_read_32_* 信号は demo bundle と同様に接続されます。

4.4 Data acquisitionコントロール

capture_en 信号は write enable 信号として機能します。 FIFO へのデータの書き込みを妨げる状況が 3 つあります。

- ファイルを閉じたとき
- FIFOがいっぱいの場合
- 過去にファイルを開いてからFIFOがいっぱいになった場合

したがって、 capture_en (Verilog 内) の条件は次のようになります。

```
assign capture_en = capture_open && !capture_full &&
                    !capture_has_been_full ;
```

そして VHDL では:

```
capture_en <= capture_open and not capture_full
              and not capture_has_been_full ;
```

capture_open 信号は、capture_clk の clock domain 用の user_r_read_32_open のコピーです。

実際のアプリケーションでは、FIFO への書き込みには他の条件が存在することがよくあります。たとえば、video frame の開始を待機するか、特定のエラー状態を待機します (デバッグに data acquisition を使用する場合)。この種の条件は、必要に応じてこの式に追加できます (logic AND のおかげで)。

信号 capture_has_been_full は、FIFO がいっぱいになると High に変化し、ファイルが閉じられた場合にのみ Low に戻ります。したがって、FIFO がいっぱいになると、data acquisition は停止し、ファイルが開いている限り再起動しません。

重要:

サンプル コードには *capture_en* の別の定義があり、偽のデータ ソースの速度を低下させるのに役立ちます。実際のアプリケーションでは、*capture_en* を上記に変更する必要があります。

Verilog で capture_has_been_full を実装するコードに進みます。

```
always @(posedge capture_clk)
  begin
    if (!capture_full)
      capture_has_been_nonfull <= 1;
    else if (!capture_open)
      capture_has_been_nonfull <= 0;

    if (capture_full && capture_has_been_nonfull)
      capture_has_been_full <= 1;
    else if (!capture_open)
      capture_has_been_full <= 0;
   end
```

VHDL:

```
process (capture_clk)
begin
  if (capture_clk'event and capture_clk = '1') then
    if  ( capture_full = '0' ) then
      capture_has_been_nonfull <= '1' ;
    elsif  ( capture_open = '0' ) then
      capture_has_been_nonfull <= '0' ;
    end if;

    if (capture_full = '1' and capture_has_been_nonfull = '1') then
      capture_has_been_full <= '1' ;
    elsif  ( capture_open = '0' ) then
      capture_has_been_full <= '0' ;
    end if;

  end if;
end process;
```

FIFO の capture_full が High になると、capture_has_been_full も High になります。ファイルが閉じると、capture_has_been_full は Low になります。

もう 1 つの信号である capture_has_been_nonfull は、別の問題を解決します。FIFO がリセットされている限り、FIFO の 'full' 信号は High になります。この理由により 'full' 信号が High の場合、capture_has_been_full は High であってはなりません。言い換えれば、capture_has_been_full は、capture_full が Low (FIFO がリセットから復帰したことを意味する) であり、その後 High になった (FIFO が実際にフルになったことを意味する) 場合にのみ High になる必要があります。

したがって、このコードは少し複雑ですが、原理を理解すれば非常に簡単です。

4.5 EOF の生成

次の 2 つの条件が満たされると、end-of-file が生成されます。

- FIFO 内のすべてのデータが消費されました (つまり、すべてのデータが IP core によって読み取られました)。
- FIFO は過去にいっぱいになったため、これ以上データは FIFO に書き込まれません。

Verilog では、これは次のように記述されます。

```
assign user_r_read_32_eof = user_r_read_32_empty && has_been_full;
```

VHDL では (これは combinatorial function であることに注意してください):

```
user_r_read_32_eof <= user_r_read_32_empty and has_been_full;
```

コード例でわかるように、has_been_full は clock domain crossing によって capture_has_been_full の値を bus_clk にコピーします。

user_r_read_32_eof は、API で許可されているとおりに Low から High に変化することに注意してください。これは、セクション 3.3 で示唆されているように、user_r_read_32_empty を備えた logical AND があるためです。

4.6 テストラン

重要:

このテスト実行は、*IP core* の不適切な構成の悪い例を意図的に示しています。この意図的な間違いの目的は、*EOF* がどのように動作するかを示すことです。このテストに使用された *IP core* には、*synchronous stream* を備えた小型の *buffers* が搭載されていました。これらは、*data acquisition* アプリケーションにとっては間違った選択です。適切に構成された *IP core* は、以下に示すほどパフォーマンスが低下することはありません。

送信データの再現性を保証するために、データ ソースは送信ワード数をカウントする単純なカウンターとして選択されます。 EOFまでのデータ量はランダム： EOF は、コンピュータが別の作業で忙しくなり、device file からの読み取りタスクを一時的に無視したときに発生しました。

テスト実行は Linux で示されていますが、Windows でも実行できます。 command line utilities の実行の詳細については、次のいずれかのガイドを参照してください。

- Getting started with Xillybus on a Linux host
- Getting started with Xillybus on a Windows host

テスト実行は次のようになります。

```
$ cat /dev/xillybus_read_32 > first
$ cat /dev/xillybus_read_32 > second
$ ls -l
```

```
total 77740
-rw-rw-r--. 1 liveuser liveuser 71727100 Jul 13 15:31 first
-rw-rw-r--. 1 liveuser liveuser  7874556 Jul 13 15:31 second
```

したがって、最初の試行では約 71 MB が収集されましたが、2 回目の試行では 7 MB しか収集されませんでした。各実行のデータ量は、オペレーティング システムが他の処理を行うために読み取りプロセスを無視する前に受信したデータの量によって異なります。おそらく、ディスクに書き込むために読み取りプロセスが一時的に停止されたと考えられます。

ただし、/dev/null に送信してすべてのデータを破棄しても、最終的には停止します (dd ユーティリティの詳細については、“man dd” を試してください)。

```
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+34365 records in
0+34365 records out
140756988 bytes (141 MB) copied, 18.0364 s, 7.8 MB/s
$ dd if=/dev/xillybus_read_32 of=/dev/null bs=1M
0+6027 records in
0+6027 records out
24684540 bytes (25 MB) copied, 3.16028 s, 7.8 MB/s
```

これら 2 つのテストの両方で、コンピューターのマウスを動かすとデータ フローが停止しました。これにより、オペレーティング システムの注意が十分に妨げられました。

ここでも、次のことを強調することが重要です。 synchronous stream が使用されているため、これらは非常に悪い結果です。 asynchronous stream と適切な量の DMA buffers を使用すれば、この種の問題はまったく予想されません。

最後に、ファイルの 1 つに何が含まれているかを確認します。

```
$ hexdump -C -v first | head
00000000  f8 fb a2 01 f9 fb a2 01  fa fb a2 01 fb fb a2 01  |................|
00000010  fc fb a2 01 fd fb a2 01  fe fb a2 01 ff fb a2 01  |................|
00000020  00 fc a2 01 01 fc a2 01  02 fc a2 01 03 fc a2 01  |................|
00000030  04 fc a2 01 05 fc a2 01  06 fc a2 01 07 fc a2 01  |................|
00000040  08 fc a2 01 09 fc a2 01  0a fc a2 01 0b fc a2 01  |................|
00000050  0c fc a2 01 0d fc a2 01  0e fc a2 01 0f fc a2 01  |................|
00000060  10 fc a2 01 11 fc a2 01  12 fc a2 01 13 fc a2 01  |................|
00000070  14 fc a2 01 15 fc a2 01  16 fc a2 01 17 fc a2 01  |................|
00000080  18 fc a2 01 19 fc a2 01  1a fc a2 01 1b fc a2 01  |................|
00000090  1c fc a2 01 1d fc a2 01  1e fc a2 01 1f fc a2 01  |................|
```

予想通り、データにはカウントアップシーケンスが含まれています。データの生成に使用されるカウンターは決してリセットされないため、シーケンスは 0 から始まりません。

4.7 バッファリングされたデータの量の監視

特定の stream に属する Xillybus の buffers に保持されているデータの量を追跡したいことがよくあります。これは、latency の制御、overflow または underflow の防止、または read() または write() への関数呼び出し中に application software がスリープするのを防ぐのに役立ちます。

たとえば、FPGA から host へのデータ フローに関しては、次のようになります。IP Core は FPGA の FIFO からこのデータを読み取っているため、buffers には大量のデータが保存されている可能性がありますが、application software はこのデータをまだ消費していません。このように待機しているデータの量を知りたいことがよくあります。

同様に、反対方向: application softwareがstreamに書き込んだデータがFPGAのFIFOにはまだ届いていない可能性があります。直接の理由は、FPGA の FIFO がいっぱいであるため、IP core からこれ以上データを受け入れることができないことです。ただし、実際の説明は、データが application logic によって消費されるのを待っているということです。

Xillybus には、buffers のデータ量を見積もるための専用機能がありません。ただし、次に示すように、Xillybus の既存の機能を使用してこの機能を実装する簡単な方法があります。

提案された解決策を説明するために、demo bundle の streams の 1 つ (FPGA host、32 ビット) が data acquisition に使用されているとします。

次のカウンタは、ファイルが開かれてから (IP core によって) FIFO からフェッチされたデータ ワードの数をカウントするために使用されます。

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_r_read_32_open)
    count_data <= 0;
  else if (user_r_read_32_rden)
    count_data <= count_data + 1;
```

3.4 セクションで示唆されているように、count_data は registers のアレイ内の register にすることができます。

別の解決策は、IP core に別の Xillybus stream (FPGA から host へ) を追加することです。この stream は、count_data をこの追加の stream の data port (つまり、通常 FIFO のデータ出力に接続されている port) に直接接続することにより、count_data の値を host に送信するために使用されます。

この stream の 'eof' port と 'empty' port は常に Low に保つ必要があります。この stream は、IP Core Factory の "use" パラメータを "Command and status" に設定することにより、synchronous stream として構成する必要があります。その結果、application software は、count_data の更新された値を取得するために、いつでもこの stream から 4 バイトを読み取ることができます。

count_data は bus_clk と同期しているため、Xillybus IP core の data port に直接接続できることに注意してください。

buffers のデータ量は、count_data と、application software がオープンされてから device file から読み取ったデータ量 (つまり、この例では /dev/xillybus_read_32) との差として計算できます。もちろん、ソフトウェアはこの stream から読み取るデータの量を追跡する必要があります。

逆方向 (host から FPGA) では、同様のカウンタを FPGA で維持できます。

```
reg [31:0] count_data;

always @(posedge bus_clk)
  if (!user_w_write_32_open)
    count_data <= 0;
  else if (user_w_write_32_wren)
    count_data <= count_data + 1;
```

これは同じ原理で機能します。 application software は、関連する device file に書き込むデータの量を追跡します。 buffers に保存されているデータの量を知る必要がある場合、application software は count_data を読み取ります。このデータ量は、(device file がオープンされてから device file に書き込まれた) データ量と count_data の値の差として計算されます。

これまでの説明では、FIFOs のデータが計算に含まれていないことに注意してください。 Xillybus が buffers 内に保持するデータのみが考慮されました。 FIFOs に保存されているデータを含むエンドツーエンドの番号を取得したい場合があります。この目的のために、FIFOs の反対側の操作をカウントする必要があります。つまり、FPGA から host までの stream 用の FIFO に書き込まれる要素の数です。逆に、これは FIFO から読み取られる要素の数です。

ただし、FIFO の反対側が別の clock (たとえば、前述の capture_clk) と同期している

場合、これを実装するのはより困難になる可能性があります。これは、count_data がこの他の clock とも同期する必要があるためです。その結果、count_data の値を IP core に接続するには clock domain crossing が必要になります。したがって、2 つの異なる clocks を FIFO に接続する場合、精度と単純さの間にトレードオフが生じます。

5

simulation で推奨される方法

5.1 全般的

満足のいく simulation とは、好みと作業方法の問題です。それにもかかわらず、仮定は常に simulation に関して行われます。これらの仮定には、特定の機能要素が期待どおりに動作するという期待が含まれます。したがって、simulation でこれらの機能要素を調べるのは無意味です。シミュレーションすると有益な特定の機能要素も存在する可能性がありますが、シミュレーションすると複雑すぎるか、時間がかかりすぎます。

このセクションでは、simulation プロセスに関するいくつかの前提条件と制限事項を提案します。 Xillybus IP core を含むシステムの simulation に対するアプローチについても説明します。これらのガイドラインは、その性質上、この文書の残りの部分に比べて議論が容易です。

Xillybus IP core とその driver は複雑なシステムであり、さまざまなシナリオで広範囲にテストされています。したがって、simulation の助けを借りて IP core 自体のバグが見つかる可能性は低いです。 テラバイト規模のデータ転送や広範囲の使用パターンでバグが見つからなかった場合、simulation でそのようなバグが見つかる可能性は低いでしょう。

さらに、IP core の動作は host からの応答に大きく依存します。 driver と application software は両方とも、異なる方法で異なる delays で応答しますが、これはほとんど予測不可能です。さらに、bus の latency (PCIe、AXI、または USB) も同様にランダムであるため、予測できません。したがって、包括的な simulation を実現することはほぼ不可能です。

これを考慮して、FIFO が Xillybus IP core に接続される時点までは、application logic の simulation を実行することをお勧めします。したがって、IP core は、データの方向に応じて、この FIFO をドレインまたはフィルする black box としてシミュ

レートされます。

5.2 asynchronous streamsのシミュレーション

stream が asynchronous に設定されている場合、IP core は (stream の方向に応じて) FIFO との間でデータを転送するため、FIFO は overflow または underflow の状態に到達しません。

これは、host 上のアプリケーション ソフトウェアが I/O 操作を十分な頻度で実行し、Xillybus の帯域幅能力がその使命に十分である限り当てはまります。これら 2 つの条件は、適切に設計されたプロジェクトの結果です。 simulation を利用して design の 2 つの側面を検証すると有益です。

- FIFO が overflow または underflow に到達するかどうか (方向によって異なります)。
- セクション 4.5 で提案されているように、application logic がそのような障害状況に正しく応答するかどうか。

適切な操作をシミュレートするために、関連する 'open' 信号が高い (ファイルが host によって開かれていることを示す) 限り、IP core が FIFO との間で最大速度でデータを転送すると仮定できます。

host から FPGA までの stream の場合、FIFO が underflow の影響を受けるときに何が起こるかをテストすることは有益です。 FIFO が空になったように見せて、このイベントをシミュレートすることをお勧めします。たとえば、FIFO が test bench の一部である場合、test bench は 'empty' 信号 (application logic に接続されている) を High に変更します。あるいは、host からのデータ フローをシミュレートする test bench の部分が、一定期間、FIFO へのデータのプッシュを単に停止する可能性もあります。この結果、FIFO は空になります。

FPGA から host への stream の場合も同様です。 'full' ラインを高に変更して、FIFO の overflow をテストできます。あるいは、test bench が FIFO からのデータのフェッチを一定期間停止することもでき、同じ効果が得られます。

stream のデータの連続性が損なわれる理由の 1 つは、application logic が stream の帯域幅制限 (または IP core の合計帯域幅の制限) を超えようとすることです。この可能性がある場合は、test bench で帯域幅制限をシミュレートすることもお勧めします。これは、test bench (IP core として機能) が、stream の意図された帯域幅によって制限されるデータ レートで FIFO を埋めるか空にすることによって実行できます。

ただし、application logic は帯域幅の制限を超えることができないため、多くのアプリケーションではこの種の simulation は不要であることに注意してください。

5.3 synchronous streamsのシミュレーション

simulation の目的での synchronous stream の主な違いは、IP core のデータ フローが連続的ではないことです。 synchronous stream の場合、IP core は、host (read() または write()) で保留中の関数呼び出しがある場合にのみ、FIFO との間でデータを転送します。

したがって、IP core の動作は、application software の I/O に対する要求により大きく依存します。したがって、IP core をシミュレートする test bench の部分は、application software のアクセス パターンを念頭に置いて作成する必要があります。

stream の目的が大量のデータを交換することである場合、synchronous stream はあまり好ましくないオプションであるため、synchronous stream に対して overflow または underflow をシミュレートすることは無関係である可能性があります。ただし、これらの条件をシミュレートする方法は asynchronous streams の場合と同じです。

5.4 simulation の簡略化された方法

overflow および underflow への応答をテストすることに興味がない場合は、simulation または IP core 用のより簡単な方法があります。たとえば、host から FPGA 方向の場合: FIFO は、read enable 信号が High のときに各 rising clock edge のファイルからデータ ワードを読み取るだけで test bench に実装できます。 FIFO のこの簡略化されたビューは、host が関連する device file に十分な速さでデータを書き込むことで FIFO が空になるのを防ぐという前提に基づいています。

逆に、write enable 信号が High の場合、test bench はワードをファイルに書き込みます。以前と同様に、host はデータを十分に速く読み取ることで、FIFO がいっぱいになるのを常に防ぐという前提があります。

このアプローチは、データ フローの連続性が損なわれる可能性を無視しません。むしろ、このアプローチでは、壊れたデータ フローは simulation の範囲を超えた何かの結果である可能性が高いと認識しています。 DMA buffers が浅すぎる、application software の応答性が低い、または host の全体的な状態に起因する CPU の剥奪。そのようなイベントが実際に発生した場合、application logic は host にそれを認識させる必要があります。すでに上で示唆したように、このメカニズムはシミュレートできます。

ただし、このアプローチでは、application logic が stream の帯域幅制限を超えようとする可能性が無視されます。このようなシナリオが現実的な可能性である場合、simulation 用のこの簡略化された方法は適切ではない可能性があります。