(機械で日本語に翻訳)

Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.0

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 This document may also be slightly outdated in relation to the original.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、ほとんどの場合、 IP core との直接インターフェースについて API を理解する必要はありません。

IP core との直接インターフェースが必要な場合でも、ほとんどすべての API 定義は単純なルールから推測できます。 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。このルールは暗黙のうちに、 Xillybus IP core が接続されている logic にいつどのようにアクセスするかについて、仮定を立てる必要がないことも意味します。データ アクセスは連続的である場合もあれば、特定の瞬間にデータ交換がなく、任意の長さの時間ギャップが存在する場合もあります。 FIFO またはメモリは、このようなアクセス パターンに問題がないため、 IP coreと直接インターフェイスする logic にも問題はありません。

特に IP core と FIFOの間のデータ フローに説明のつかない一時停止が見られた場合、 IP coreの不規則なアクセス パターンをバグと見なすのはよくある間違いです。

まれな状況を正しく処理できない可能性があるため、 IP coreと直接インターフェイスする application logic を設計する場合、少なくとも designの初期段階では、logicの消費を減らすためにそうしないことをお勧めします。必要な機能が IP coreとの直接インターフェイスを必要としない限り、 user application logicのバグを回避するために FIFOs または block RAMs を使用することをお勧めします。

このガイドでは、このダイレクト API を定義し、一般的なアプリケーションについて詳しく説明します。このドキュメントに記載されている詳細に取り掛かる前に、次のガイドで提案されているように、 Xillybusの初期経験を積むことをお勧めします。

- 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

このガイドの一部では、同期と非同期の streamsの違いを理解していることも前提としています。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 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、 Xillybus IP coresのサブセットです。したがって、 “Xillybus” という名前は、別段の記載がない限り、このドキュメントでは 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 との間のすべての信号は、 IP core 自体によって供給される bus_clkの rising edge と同期している必要があります。

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、プラットフォームに応じた周波数を持ちます。 ベースライン IP core (リビジョン A) の場合、 bus_clk の周波数は、最大帯域幅 (宣伝されている) がそれぞれ 200 MB/s、 400 MB/s 、または 800 MB/sであるかどうかに応じて、 62.5 MHz、125 MHz 、または 250 MHzのいずれかになります。

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。

ほとんどの場合、 PCIe ブロックまたはそれを生成する processor core を構成することにより、限られた選択肢のリスト内で clockの周波数を変更する可能性があります。

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 によって駆動される application logic は、適切な timing constraints によってもカバーされます。 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。 timing constraints は、 IP core を駆動する clock から bus_clkによって駆動される application logic に伝搬されます。

これは、すべての application logic を bus_clkで駆動する必要があり、データ ソースとデータ コンシューマも同様であると言っているわけではありません。多くの場合、 dual-clock FIFO は IP coreと一緒に使用されます。 一方は Xillybus IP coreに接続されているため、 bus_clkと同期しています。したがって、 application logic は

FIFOの反対側に接続され、 application logicの clockと同期され、任意の許容周波数を持つことができます。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にも使用されます。

2.2 データ幅

ベースライン Xillybus IP cores (リビジョン A) では、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 ビット幅のデータで動作します。 XillyUSBと同様に後のリビジョンでは、より広いデータ 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。 両側が IP coreに接続された FIFO を備えているため、2 つの streamsに loopback を実装しています。

demo bundle 内の 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。

データ フローの方向に応じて、 FIFO の “empty” または “full” 信号が Xillybus IP coreに接続され、 Xillybus IP coreはこれらの信号を使用してデータ転送のバーストを開始するかどうかを決定します。

バーストが開始されると、これらの信号に依存して、 Xillybus IP core が空の FIFO から読み取ろうとしたり、満杯の FIFO に書き込もうとしたりしないようにします。

ただし、 FIFOでバーストがいつ開始されるかについては、準備ができていることが示されている場合でも、保証はありません。また、そのようなバーストの長さに関する確実性もありません。たとえば、 IP core は、データが空になる前であっても、バーストの途中で FIFO からのデータのフェッチを停止する可能性があります。

原則として、 Xillybus IP core はそれに接続されているすべての FIFOs に均等にサービスを提供しようとします。 “empty” または “full” を頻繁にアクティブにしないため、より速く満たされる傾向がある FIFOs には、より長いバーストが付与されます。

この単純なアービトレーション方式により、急速にいっぱいになる傾向がある FIFOs との効率的な通信が保証され、同時に、低速でデータを受信する FIFOs 上の低 latency との通信が保証されます。

FIFOの深さに関しては、 Xillybus IP core は任意の値で動作しますが、予想されるデータ フローに対処するには、この属性を選択する必要があります。これは試行錯誤の問題である場合もありますが、 2 kBytes の深さの FIFO は、データ レートが高い場合でも、非同期 streamにはほぼ常に正しい選択です。

この選択の背後にある理論的根拠は、 Xillybus core が overflow または underflowを引き起こすのに十分な長さのこのサイズの FIFO を無視する可能性が低いということです。これはもちろん、 hostで実行されている user application software によって DMA buffers が十分に迅速に満たされるか空にされる限り、当てはまります。そうでない場合、解決策は DMA buffers を大きくすることです。 FPGAで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合理的ではありません。

IP core が FIFO からデータを読み取るために接続されている場合、通常の FIFO の動作が期待されます (FWFT, First Word Fall Throughとは対照的に)。

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 にな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 IP core に示すと (“empty” または “full”の値が適切な場合)、 core のステート マシンは、少なくとも 1 つのデータ要素の転送につながる一連のイベントを開始できます。 .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 つの clock cycleの間に “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の状態の整合性に関しては無害ですが、 streamで予期しない予測不可能なデータが流れる可能性があります。

同じことが “full” シグナルにも当てはまります。 IP cores がデータ ワードを書き込む前に Low から High に変化すると、 IP core は 1 つの clock cycleの間に “full” 信号を無視する場合があります。繰り返しますが、これは IP core 自体には無害ですが、データ ワードが失われます。

適切に設計された FIFO は、 Xillybus IP coreとの通信準備ができている間にリセットされることによってのみ、この障害状態を作成できます。いずれにせよ、これは悪い習慣です。

IP core が application logicと直接接続されている場合は、この点に関して標準の FIFO を模倣するように注意する必要があります。

3

信号の説明

3.1 FPGA シグナルの命名規則

bus_clk と quiesceの 2 つのグローバル シグナルを除いて、すべてのシグナルは単純な規則に従います。たとえば、 write enable 信号の名前は user_w_write_32_wrenです。この名前は、次の 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 への Streams には、代わりに “r” があります。アドレス信号は双方向に適用されるため、この部分はありません。hostの視点は、 “w” または “r”の選択に使用され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 (IN) が続き、信号が IP coreへの入力であること、または IP coreからの出力である場合は (OUT) であることを示します。

3.2 host FPGA 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 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。これは標準の FIFOs の動作です。したがって、 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) のみ、このルールに注意する必要があります。

  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**) – この信号は、 host 上の関連する device file が書き込み用に開かれている場合にハイになります (ファイルが読み取り専用で開かれている場合、許可されている場合、この信号は変更されません)。このシグナルは、オプションで、ファイルが閉じられたときに 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として使用されます)。

host 上の複数のプロセス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、すべてのプロセスがファイルを閉じるまでハイのままです。

3.3 FPGA host 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、 read enableが高い結果として FIFO が変更した場合を除き、変更してはなりません。つまり、 user_r_{devfile}_rden が High になった後、 clock cycle でのみ変化す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。この信号が高い場合、 IP core は次の clock cycleの user_r_{devfile}_data に有効なデータが存在することを期待します。

- 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。これは標準の FIFOs の動作なので、このルールに注意する必要があるのは、 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) だけです。

  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。これは 'empty' 信号に似ていますが、ハイになると、ファイルが閉じられて再び開かれるまで、 core は FIFO から読み取りません (つまり、 user_r_{devfile}_rden はローに保たれます)。

  hostでは、 application software は、この信号が高くなる前に IP core が受信したデータの読み取りを終了し、 read() 関数を呼び出したときに EOF を受信します。

  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 に変更する必要があります。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)。

許可されていないときに 'eof' が High に変化しないようにする 1 つの可能性は、 Verilogで次の形式の combinatoric functionの出力にすることです。

```
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**) – この信号は、 host 上の関連する device file が読み取り用に開かれている場合にハイになります (ファイルが書き込み専用で開かれている場合、許可されている場合、この信号は変更されません)。このシグナルは、オプションで、ファイルが閉じられたときに 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として使用されます)。

  host 上の複数のプロセス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、すべてのプロセスがファイルを閉じるまでハイのままです。

  'eof' 信号と 'open' 信号の間に直接接続はありません。 'open' 信号は、 hostでファイルが閉じられたときに Low に変化しますが、 'eof' 信号が High に変化したときや、 EOF が hostに配信されたときではありません。

3.4 メモリ インターフェイス信号

Xillybus インターフェイスは、アドレス信号も持つように構成できます。アドレスの increment は、読み取りサイクルと書き込みサイクルで自動的に発生します。また、 application software は、ファイル内で seeking 用の標準 API を使用することにより、アドレスに任意の値を設定できます (例: lseek())。

上記の信号の一部と次に詳述する信号を使用すると、標準の block RAM を IP coreに簡単に接続でき、 block RAMのメモリ アレイを host でファイルとして使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 アレイに対する読み取りおよび書き込み操作になります。 host は、読み取りまたは書き込み操作の長さに応じて、単一のメモリ要素またはセグメントにアクセスできます。

また、 block RAMのような信号を提供する FPGAに registers のアレイを実装することにより、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に、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。

これらは、この目的のための 2 つの追加信号です。

- user_{devfile}_addr (**OUT**) – この信号には、現時点でのアドレスが含まれます。 read enable または write enable のいずれかが High の場合、これが読み取りまたは書き込み対象のアドレスです。この信号を block RAMのアドレス入力に直接接続すると、当然のことながら機能します。この信号の幅は、最大 32 ビットまで設定可能です。

  読み取りまたは書き込み操作が、この信号の幅で可能な最大アドレスを超えると、アドレスの値はゼロに戻ります。範囲外の lseek() への関数呼び出しは、この信号に要求されたアドレスの LSBs の値を割り当てます。

- user_{devfile}_addr_update (**OUT**) – この信号は、 hostでの lseek() への関数呼び出しの結果として、1 つの clock cycle の間ハイになります。この信号の目的は、アドレスの更新の結果として、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、こ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。 また、 'update' 信号が High だった後の clock cycle でも High に変化する可能性があります。

  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;
```

この例では、アドレスが更新されるのと同じ clock cycle で 'update' 信号が高いこともわかります。

'empty' はいつでも Low に変化する可能性があるため、(アドレスに関係なく) アドレス更新のたびに 'empty' を High に変更し、 'empty' を Low に戻すことができるかどうかを評価するのに時間がかか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。

関連する host 上の 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 になると)、アドレスがリセットされ、その値がゼロに変わります。ただし、これはアドレス更新とは見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

hosts 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 は driverをまだロードしていないか、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にまったく接続されていない場合。

この信号の意図は synchronous resetとして機能することですが、ほとんどの場合必要ではありません。 quiescent state に存在することの副作用の 1 つは、すべての

ファイルが閉じられることです。そのため、 application logic は reset シグナルとして *_open シグナルのみに依存できます。 '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性は、たとえば次のような場合によく発生します。

- ビデオ ソースからのフレーム グラブ
- アナログからデジタルへのコンバーター (ADC) からのデータ サンプル
- 他のデータ ソースからのデータの取得
- FPGAからのデバッグ情報の受信

このようなアプリケーションでは、データ レートが高くなる可能性があり、データフローの継続性を保証する必要があります。 データが失われることは許されません。

data acquisition アプリケーションは、データを FIFOに書き込むことにより、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 application logic を実装する方法に焦点を当てます。これを実現するために、 application logic は連続性が失われた時点でデータの流れを停止し、この時点で EOF を送信します。このようにして、 application software は、到着する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がアクティブになることはありませんが、アクティブになると、問題を認識し、解決する機会が得られます。

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。

それにもかかわらず、特定の host プログラミング技術を含む連続した stream データを維持する方法があります。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

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

以下では、 Xillybus を使用して連続ソースから 32 ビット幅の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、キャプチャされたデータ 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サンプル コードは、次のリンク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で 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。

サンプル コードは、標準の 32 ビット幅の dual clock FIFOの instantiation を作成します。サンプル コードの synthesis を実行する前に、この FIFO をツール (Vivado または Quartusなど) で生成してください。この FIFO の名前は async_fifo_32で、深さは 512 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と同期しているとします。したがって、データは単純に標準の dual-clock FIFOに供給されます。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 はファイルが閉じられるとリセットされ、その user_r_read_32_* 信号は以前と同じように接続されます。

4.4 キャプチャーコントロール

capture_en 信号は、キャプチャされたデータの write enable 信号として機能します。次の 2 つの状況のいずれかでは、データのキャプチャは行われません。

- ファイルを閉じたとき
- FIFO が満杯であるか、過去に満杯だった場合

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 user_r_read_32_openのコピーですが、 capture_clkの clock domain にあります。

ビデオ データのフレームの開始を待機する、またはデバッグに data acquisition を使用する場合に特定のエラー状態を待機するなど、他のアプリケーション固有の条件を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ハイになり、ファイルが閉じられたときにのみローに戻ります。そのため、 FIFO がいっぱいになると、データ キャプチャが停止し、ファイルが開かれている限り再開されません。

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。実際の信号をキャプチャする場合は、*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;
```

これはほぼ予想どおりです。 FIFOの capture_full が高くなると、 capture_has_been_full が高くなり、ファイルが閉じると低くなります。もう 1 つの信号 capture_has_been_nonfullは、別の問題を解決します。 FIFO はリセット時に 'full' 信号を高く維持するため、 capture_full が低く (FIFO がリセットから抜け出したことを意味する)、その後高くなった (FIFO が実際にいっぱいになったことを意味する) 場合にのみ、 capture_has_been_full は高くなるはずです。

したがって、このコードはやや複雑ですが、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、 IP coreによって読み取られました)。
- FIFOは以前にいっぱいになったため、これ以上データは書き込まれません。

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 (これは組み合わせ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;
```

サンプル コードでわかるように、 has_been_full は、 clock domain crossing から bus_clkへの次の capture_has_been_full です。

user_r_read_32_eof は、 APIに従って、許可されている場合にのみローからハイになることに注意してください。これは、セクション 3.3で提案されているように、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:

このテストの実行では、 *EOF* のメカニズムが動作するように、 *IP core* 設定の悪い例を意図的に示しています。 このテストに使用された *IP core* の *buffers* は小さく、 *stream* は同期型です *(*これは *data acquisition* アプリケーションでは正しくありません*)*。実際のテストは、はるかにうまく機能します。

キャプチャされたデータの再現性を確保するために、データ ソースは、送信された単語の数をカウントするだけの偽のデータ ジェネレータとして選択されます。EOF までのデータの量は、コンピュータが別のことをしているときにビジー状態になり、 device fileからの一連の読み取りを一時的に無視したかによって異なります。

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。コマンド ライン ユーティリティ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
```

そのため、最初の試行では約 71 MB がキャプチャされましたが、2 回目の試行では 7 MB のみがキャプチャされました。各実行のデータ量は、オペレーティング システムが読み取りプロセスを無視して別の処理を行う前に受信したデータ量によって異なります。ほとんどの場合、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ました。

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、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 システムが十分に注意をそらされました。

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

最後に、キャプチャされたファイルの 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  |................|
```

予想どおり、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。偽のデータを生成するために使用されるカウンターは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止するのに役立ちます。

たとえば、 FPGA から hostの方向では、 host上の application software によってまだ消費されていない、 FPGA の FIFO から (Xillybus IP coreによって) 読み取られたデータの量を知ることを意味します。

同様に、反対方向では、これは application software が streamに書き込んだが、FPGA の FIFO に到達していないデータの量を知ることを意味します (FIFO がいっぱいで、データが application logicによって消費されるのを待っているため)。

次に示すように、 Xillybusの機能を使用してこれを実装する簡単な方法があるため、Xillybus はこれ専用の機能を提供しません。

提案された解決策を説明するために、 demo bundle の FPGA から host までの 32-bit stream 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要素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;
```

count_data の値は、この目的のために別の専用 Xillybus stream (FPGA から hostへ) を介して host に公開できます。 これは、 count_data をこの別の 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って行われます。

'eof' port と 'empty' port は常に Low に保つ必要があります。この専用 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、同期するように構成されます。

これにより、 application software はこの stream からいつでも 4 バイトを読み取ることができ、 count_dataの更新された値を取得できます。

あるいは、 3.4セクションで提案されているように、 count_data を registersのアレイ内の register にすることもできます。

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。

この余分な streamによってこの値がソフトウェアに認識されると、 buffers のデータ量は、 count_dataと、 application software が開かれてから 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)。もちろん、これには、ソフトウェアがこの streamから読み取るデータの量を追跡する必要があります。

host から FPGAへの反対方向では、同様のカウンターを 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;
```

同じ原理により、 application software は関連する device fileに書き込むデータの

量を追跡し、 FPGA上の FIFO に書き込まれた要素の数を通知します。これは、count_dataの値を読み取ることによって行われます。

上記の両方のケース スタディでは、 FIFOs のデータは計算に含まれず、 Xillybus が buffersに保持するデータのみが含まれます。 FIFOsに保存されているデータを含め、エンド ツー エンドの番号を取得することが必要な場合があります。この目的のために、 FIFOs の反対側での操作をカウントする必要があります。つまり、最初のケースでは FIFO に書き込まれる要素の数、2 番目のケースでは FIFO から読み取られる要素の数です。

ただし、 FIFO の反対側が bus_clk ではない clock (たとえば、このセクションで前述した capture_clk) と同期している場合、これを実装するのは難しいかもしれません。これは、 count_data がこの他の clock と同期しているためです。そのため、count_dataの値を IP coreに接続するには、 clock domain crossing が必要です。したがって、 bus_clk 自体が data acquisition または再生に使用される clock でない限り、精度と単純さの間にはトレードオフがあります。

5

推奨されるシミュレーション方法

5.1 全般的

満足のいくシミュレーションとは何かは、好みと作業慣行の問題です。それにもかかわらず、シミュレーションには、特定のことが期待どおりに機能するという前提が常にあります。シミュレートするのは興味深いこともありますが、実行するには複雑すぎたり時間がかかりすぎたりすることがあります。

このセクションでは、シミュレーション プロセスに関する一連の仮定と制限、および Xillybus IP coreを含むシステムをシミュレートするためのアプローチを提案します。これらのガイドラインは、その性質上、このドキュメントの残りのガイドラインほど必須ではありません。

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ストレス テストが行われています。したがって、シミュレーションで IP core 自体にバグが見つかる可能性は低いです。テラバイト単位のデータ トランスポートやさまざまな負荷パターンでバグが見つからない場合です。

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 はどちらも応答が異なり、遅延も異なりますが、これはほとんど予測できません。その上、 bus (PCIe、 AXI または USB) の latency 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、包括的なシミュレーションはほとんど不可能です。

これに照らして、 FIFO が Xillybus IP coreと接続するポイントまで application logic をシミュレート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 非同期 streamsのシミュレーション

stream が非同期に設定されている場合、 IP core は FIFO との間で (streamの方向に応じて) データを転送し、 FIFO が overflow または underflowの状態に到達しないようにします。

これは、 host 上のアプリケーション ソフトウェアが I/O 操作を十分頻繁に実行し、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あり、シミュレーションによって検証することは有益です。確認すべき 2 つの側面があります。

- 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への stream をテストする場合、 FIFO が underflowに影響されたときに何が起こるかについて、 FIFO が空になったように見せることによって、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、 'empty' 信号 (application logicに接続されている) を高に変更します。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、 FIFO へのデータのプッシュを一時停止するだけで、 FIFO が空になることがあります。

同様に、 FPGA から hostへの stream の場合、 'full' ラインをハイに変更して、FIFOの overflowをテストできます。または、代わりに、 test bench が FIFO からのデータのフェッチを一定期間停止して、同じ効果を得ることができます。

application logic が stream (または IP coreの総帯域幅) の帯域幅制限を超えようとするため、データの連続性が失われる可能性があります。これが可能である場合 (多くのアプリケーションではそうではありません)、 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、 streamの意図した帯域幅によって制限されるデータ レートで FIFO を満たしたり空にしたりすることで実現できます。

5.3 同期 streamsのシミュレーション

非同期 streamと比較すると、同期 stream は、 IP coreのデータ フローが連続的で

はないという点で (シミュレーション目的で) 異なります。 IP core は、 host (read() または write()) に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、 I/Oに対するアプリケーション ソフトウェアの要求によって大きく左右されます。したがって、 IP core をシミュレートする test bench の部分は、アプリケーション ソフトウェアのアクセス パターンを考慮し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、同期 stream はあまり優先されないオプションであるため、 overflow または underflow の可能性は関連性がない可能性があります。それでも、これらの条件をシミュレートする方法は、非同期 streamsの場合と同じです。

5.4 簡略化されたシミュレーション方法

overflow と underflowの条件をシミュレートすることに興味がない人は、 IP coreをシミュレートするためのより簡単なオプションがあります。たとえば、 host から FPGA 方向では、 read enable 信号が High のときに rising clock edge ごとにファイルからデータ ワードを読み取るだけで、 FIFO を test bench に実装できます。この FIFO の単純化されたビューは、 host が常にデータを FIFO に十分速く書き込み、空にならないようにするという前提に基づいています。

逆方向では、 write enable 信号がハイの場合、 test bench はワードをファイルに書き込みます。以前と同様に、これは、 host が常に FIFO からデータを読み取るのに十分な速さでデータがいっぱいにならないようにすることを前提としています。

連続性が途切れる可能性を見逃さないアプローチです。むしろ、壊れたデータの連続性は、シミュレーションの範囲を超えた何かの結果である可能性が最も高いことを認識しています。 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。そのようなシナリオが現実的な可能性がある場合、この簡略化されたシミュレーション オプションは適切ではない可能性があります。