(机器翻译成中文)

Xillybus FPGA designer's guide

Xillybus Ltd.

www.xillybus.com

Version 3.0

本文档已由计算机自动翻译，可能会导致语言不清晰。 与原始文件相比，该文件也可能略微过时。

如果可能，请参阅英文文档。

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 This document may also be slightly outdated in relation to the original.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

介绍

Xillybus的 IP cores 旨在通过 FIFO 或 dual-port block RAM与 user application logic 连接。因此，在大多数情况下，没有必要了解 API 与 IP core 的直接接口。

即使需要与 IP core 的直接接口，几乎所有 API 定义都可以从一个简单的规则推导出来： user application logic 的行为应该与 FIFO 或 block RAM完全一样。隐含地，这条规则还意味着不应该对 Xillybus IP core 访问与其连接的 logic 的方式和时间做出任何假设。数据访问可以是连续的，或者可能存在任意长度的时间间隔，在任何给定时刻都没有数据交换。 FIFO 或内存在这种访问模式下不会有任何问题，因此任何直接与 IP core接口的 logic 都不应该有任何问题。

将 IP core的不规则访问模式视为错误是一个常见的错误，尤其是在观察到 IP core 和 FIFO之间的数据流出现无法解释的暂停之后。

由于可能无法正确处理罕见情况，因此在设计直接与 IP core接口的 application logic 时，为了减少 logic的消耗，建议不要这样做，至少在 design的早期阶段不要这样做。除非所需的功能需要与 IP core直接接口，否则建议使用 FIFOs 或 block RAMs 以避免 user application logic中的错误。

本指南定义了这个直接 API，并详细说明了流行的应用程序。在深入了解本文档提供的详细信息之前，建议先获得 Xillybus的初步体验，如以下指南中所建议的：

- 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

本指南的某些部分还假设您了解同步和异步 streams之间的区别。这在这两个指南的第 2 节中进行了讨论：

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

XillyUSB IP cores 暴露了相同的 API，并且是 Xillybus IP cores的子集。因此，除非另有说明，否则名称 “Xillybus” 在本文档中也是指 XillyUSB IP cores 。

对于那些好奇的人，可以在 Xillybus host application programming guide for Linux 或 Xillybus host application programming guide for Windows的附录 A 中找到关于如何实现 Xillybus 的简要说明。

2

一般准则

2.1 Clocking

所有来自和到 Xillybus IP core 的信号都必须与 bus_clk的 rising edge 同步，这由 IP core 本身提供。

对于基于 PCIe的 Xillybus IP cores ，此 clock 由 PCIe 模块生成，其频率取决于平台： 对于基线 IP core （修订版 A）， bus_clk 的频率是 62.5 MHz、 125 MHz 或 250 MHz，具体取决于最大带宽（如广告所示）是 200 MB/s、 400 MB/s 还是 800 MB/s。

随着后来的修订版（B、 XL 和 XXL）， bus_clk 的频率为 250 MHz。基于

Zynq的平台通常具有 100 MHz的 bus_clk 。 XillyUSB 可与 125 MHz的 bus_clk 配合使用。

在大多数情况下，通过配置 PCIe 模块或生成它的 processor core ，可以在有限的选择列表中更改 clock的频率。

如果 PCIe 块的 timing constraints 设置正确（如 demo bundles中），则由 bus_clk 驱动的 application logic 也会被正确的 timing constraints 覆盖。这同样适用于基于 Zynq的平台以及 XillyUSB： timing constraints 从驱动 IP core 的 clock 传播到由 bus_clk驱动的 application logic 。

这并不是说所有 application logic 都需要由 bus_clk驱动，数据源和数据消费者也不需要。通常， dual-clock FIFO 与 IP core一起使用： 一侧连接到 Xillybus IP core，因此与 bus_clk同步。 application logic 相应地连接到 FIFO的另一侧，与 application logic的 clock同步，可以具有任何允许的频率。因此， FIFO 不仅用作短期临时存储，还用于 clock domain crossing。

2.2 数据宽度

使用基线 Xillybus IP cores （修订版 A），每个 FIFO 或内存接口都可以处理 8 位、16 位或 32 位宽度的数据。后来的修订版以及 XillyUSB支持更广泛的数据接口。

更宽的数据允许更高的带宽性能，并且在自然传输字宽于 8 位的应用中也更方便。另一方面， host 端的固有数据宽度仍然是 8 位（一个字节），因为 read() 和 write() 函数调用以字节为单位定义了它们的长度。

The guide to defining a custom Xillybus IP core中简要讨论了选择数据宽度的注意事项。

2.3 通过 FIFO连接

demo bundle 演示了如何连接 FIFO： 它有一个 FIFO，两边都连接到 IP core，因此在两个 streams上实现一个 loopback 。

demo bundle 中的 FIFOs 配置为两侧通用的 clock 。当 FIFO 用于 clock domain crossing时，这是不合适的，在这种情况下，应该使用 dual-clock FIFO （通常称为 “asynchronous FIFO”）。

根据数据流的方向， FIFO 的 “empty” 或 “full” 信号连接到 Xillybus IP core， Xillybus IP core使用这些信号来确定是否应该启动数据传输突发。

一旦突发开始，这些信号就会被用来确保 Xillybus IP core 不会尝试从空的 FIFO 读取或写入满的 FIFO 。

但是，不能保证 FIFO何时启动突发，即使它表明它已准备好。这种爆发的长度也没有任何确定性。例如， IP core 很可能会在突发中间停止从 FIFO 获取数据，甚至在它为空之前。

一般规则是 Xillybus IP core 尝试平等地为与其连接的所有 FIFOs 提供服务。 FIFOs 往往会被更快地填充，将获得更长的爆发，因为他们不会经常激活他们的 “empty” 或 “full” 。

这种简单的仲裁方法可确保与 FIFOs 的高效通信，而 FIFOs 往往会被快速填充，同时 FIFOs 上的低 latency 以较低的速率接收数据。

至于 FIFO的深度， Xillybus IP core 适用于任何值，但应选择此属性以应对预期的数据流。尽管有时这需要反复试验，但深度为 2 kBytes 的 FIFO 几乎始终是异步 stream的正确选择，即使对于高数据速率也是如此。

这种选择背后的基本原理是， Xillybus core 不太可能忽视这种尺寸的 FIFO 足够长的时间来导致 overflow 或 underflow。只要在 host上运行的 user application software 足够快地填充或清空 DMA buffers ，这当然是正确的。如果不是这种情况，解决方

案可能是使 DMA buffers 更大。试图用更大的 FIFO 解决这个问题是不合理的，因为FPGA上的内存要少得多。

如果 IP core 连接到 FIFO 以从中读取数据，则它期望常规 FIFO 的行为（与 FWFT、First Word Fall Through相对）。

2.4 “empty” 和 “full” 信号的行为

在正常工作的 FIFO中， “empty” 信号可以在 read enable 为高电平后仅一个 clock cycle 由低电平变为高电平。同样，在 write enable 为高电平后， “full” 信号只能由一个 clock cycle 从低电平变为高电平。

当然，这两个信号随时可能变低。

Xillybus IP core 依赖于这种行为： 当 FIFO 向 IP core 指示它已准备好进行数据传输（通过低 “empty” 或 “full”，如果适用）， core 中的状态机可能会启动一系列事件，这将导致至少一个数据元素的传输.因此，如果在 IP core 从 FIFO获取任何数据之前 “empty” 信号变为高电平，则 IP core 可能会在一个 clock cycle期间忽略 “empty” 信号。就 IP core的状态完整性而言，此类事件是无害的，但可能会导致 stream中出现意外和不可预测的数据流动。

“full” 信号也是如此： 如果在 IP cores 向其写入数据字之前它从低电平变为高电平，则 IP core 可能会在一个 clock cycle期间忽略 “full” 信号。再一次，这对 IP core 本身是无害的，但会导致数据字丢失。

正确设计的 FIFO 只能在准备好与 Xillybus IP core通信时通过复位来产生这种故障情况。无论如何，这都是不好的做法。

如果 IP core 直接与 application logic接口，则必须注意模仿标准的 FIFO 。

3

信号说明

3.1 FPGA 信号的命名约定

除了两个全局信号 bus_clk 和 quiesce外，所有信号都遵循一个简单的约定。例如，write enable 信号的名称可能为 user_w_write_32_wren。此名称分为四个部分：

1. “user” 前缀是所有用户接口信号通用的。
2. “w” 部分表示该信号属于 stream 从 host 到 FPGA (host “write”)。 Streams 从 FPGA 到 host 有一个 “r” 代替。地址信号没有这部分，因为它们适用于两个方向。请注意，选择 “w” 或 “r”时采用 host的视角。
3. “write_32” 字符串出现在相关 device file的名称中： /dev/xillybus_write_32 或 /dev/xillyusb_00_write_32（如适用）。
4. 后缀表示信号的含义。

在本节的其余部分中， device file 名称（第三个组件）表示为 {devfile} 以避免混淆。

每个信号名称后跟 (IN) 表示该信号是 IP core的输入，或 (OUT) 当它是 IP core的输出时。

3.2 host 到 FPGA 传输的信号

- user_w_{devfile}_data (**OUT**) – 该信号包含写周期中的数据。
- user_w_{devfile}_wren (**OUT**) – 该信号是 FIFO的 write enable 信号： 当 user_w_{devfile}_data 信号上有有效数据应写入 FIFO （或任何其他模仿 FIFO行为的 logic ）时，它为高电平。

- user_w_{devfile}_full (**IN**) – 该信号通知 core 不能再写入数据。

  重要： 'full' 信号只能在 clock cycle 上在一个写周期后从低变为高。这是标准 FIFOs 的行为方式，因此仅当 IP core 与 application logic 直接连接（即中间没有 FIFO ）时才需要注意此规则。

  此规则的原因是 Xillybus IP core 将低 'full' 信号视为绿灯以开始从 host传输数据。不符合此规则可能会导致忽略 'full' 条件的零星写入。

  'full' 信号的典型 Verilog 实现应该是这样的：

```
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也一样：

```
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**) – 当 host 上的相关 device file 为写打开时该信号为高电平（如果文件为只读打开，当允许时，不会改变该信号）。当文件关闭（用作 active low reset）时，此信号可以选择用于重置 FIFO 或其他 logic 。

  如果一个文件被 host 上的多个进程打开（例如，作为调用 fork() 函数的结果），该信号将保持高电平，直到所有进程都关闭该文件。

3.3 FPGA 到 host 传输的信号

- user_r_{devfile}_data (**IN**) – 该信号包含读取周期中的数据。该信号不得改变，除非 FIFO 会因 read enable高电平而改变它。换句话说，它可能只在 user_r_{devfile}_rden 为高电平后在 clock cycle 上发生变化。

- user_r_{devfile}_rden (**OUT**) – 该信号是 FIFO的 read enable 信号： 当此信号为高电平时， IP core 期望在下一个 clock cycle上的 user_r_{devfile}_data 上存在有效数据。

- user_r_{devfile}_empty (**IN**) – 该信号通知 core 无法读取更多数据。

  重要： 'empty' 信号可能仅在 clock cycle 上在读取周期后从低电平变为高电平。这是标准 FIFOs 的行为方式，因此仅当 IP core 与 application logic 直接连接（即中间没有 FIFO ）时才需要注意此规则。

  此规则的原因是 Xillybus IP core 将低 'empty' 信号视为绿灯开始向 host传输数据。不遵守此规则可能会导致忽略 FIFO 为空的零星读取。

  'empty' 信号的典型 Verilog 实现应该是这样的：

```
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也一样：

```
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**) – 这个信号告诉 core 生成一个 end-of-file。这就像一个 'empty' 信号，但在它一直为高电平后， core 不会从 FIFO 读取（即 user_r_{devfile}_rden 保持低电平），直到文件关闭并重新打开。

  在 host上， application software 会在这个信号变高之前读完 IP core 收到的数据，然后调用 read() 函数时会收到一个 EOF 。

  请注意，如果仍有数据未被 application software读取， 'eof' 信号不会立即在 host 上引起 EOF 。 EOF 在 host 上的交付是根据常识进行的，即在 host读取所有数据之后。

'eof' 信号一直高电平后，之后是高电平还是低电平都无所谓。 IP core 会记住 EOF 请求，直到文件关闭。关于 'empty' 信号，在 'eof' 变高的同时变高也没关系。实际上，从 'eof' 为高电平的那一刻起， 'empty' 信号就完全不重要了，直到文件关闭。

与 'empty' 信号一样， 'eof' 信号必须在读取周期后仅在 clock cycle 上变为高电平。但是有一个例外： 当 'empty' 信号已经为高电平时， 'eof' 可能随时变为高电平。如果 host在等待数据时休眠，此异常可用于立即终止 host上的 read() 函数调用。

在不符合此规则的情况下将 'eof' 更改为高会生成 EOF，但它可能无法正常工作： EOF之前可能会丢失一些数据，或者在 EOF之前添加无关的数据，甚至在 EOF 之后（所以 application software 在 EOF之后接收数据，这是非法的）。

确保 'eof' 在不允许时不会变为高电平的一种可能性是将其作为 combinatoric function的输出，其形式为 Verilog：

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

或者在 VHDL中：

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

使用这种方法，当 'empty' 为低电平时， 'eof' 信号始终为低电平。

- user_r_{devfile}_open (**OUT**) – 当 host 上的相关 device file 为读打开时该信号为高电平（如果文件为只写打开，当允许时，不会改变该信号）。当文件关闭（用作 active low reset）时，此信号可以选择用于重置 FIFO 或其他 logic 。

  如果一个文件被 host 上的多个进程打开（例如，作为调用 fork() 函数的结果），该信号将保持高电平，直到所有进程都关闭该文件。

  'eof' 信号和 'open' 信号之间没有直接联系。 'open' 信号在 host上关闭文件时会变为低电平，而不是在 'eof' 信号变为高电平时，也不在 host上交付 EOF 时。

3.4 内存接口信号

Xillybus 接口也可以配置为具有地址信号。地址的 increment 在读取和写入周期自动发生。此外， application software 可以通过在文件（例如 lseek() ）中使用 seeking 的标准 API 来为地址设置任意值。

通过上面提到的一些信号，以及接下来详述的信号，标准的 block RAM 很容易与 IP core连接，使 block RAM的内存阵列可以作为文件提供给 host： 对文件的读写操作

会导致对内存阵列的读写操作。 host 可以访问单个内存元素或段，具体取决于读取或写入操作的长度。

此外，通过在 FPGA上实现 registers 阵列（呈现 block RAM之类的信号）， host可以轻松访问这些 registers 。

'empty' 和 'full' 信号可用于减慢对需要 wait states的存储器的读写操作，或者当有其他原因暂时延迟操作时。

这是用于此目的的两个附加信号：

- user_{devfile}_addr (**OUT**) – 该信号包含当前的地址。当 read enable 或 write enable 为高时，这是要读取或写入的地址。将此信号直接连接到 block RAM的地址输入将按自然预期工作。该信号的宽度最多可配置为 32 位。

  当读取或写入操作达到此信号宽度可能的最大地址以上时，地址的值将返回零。超出范围的对 lseek() 的函数调用将导致 this 信号被分配所请求地址的 LSBs 的值。

- user_{devfile}_addr_update (**OUT**) – 由于在 host上对 lseek() 的函数调用，该信号在一个 clock cycle 期间为高电平。该信号的目的是让 application logic 有机会表明它需要时间来准备读取数据，这是地址更新的结果。这是通过将 'empty' 信号更改为高以响应此类更新来完成的。

  为此， 'empty' 在一个读取周期后只能将一个 clock cycle 变为高电平的规则有一个例外： 它也可以在 'update' 信号为高电平之后的 clock cycle 上变为高电平。

  因此，以下 Verilog 代码是正确的：

```
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;
```

  在 VHDL中也是如此：

```
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;
```

在此示例中，我们还可以看到， 'update' 信号在地址更新时在同一 clock cycle 上为高电平。

请注意，由于 'empty' 可以随时变为低电平，因此每次地址更新（无论地址如何）都将 'empty' 变为高电平是有意义的，然后花时间评估 'empty' 是否可以变回低电平.

'full' 信号也可以以类似的方式变为高电平，尽管尚不清楚为什么这应该有用。

当 host 上的相关 device file 关闭时（即 user_w_{devfile}_open 和 user_r_{devfile}_open 均为低电平时），地址被复位，因此其值变为零。但是请注意，这不被视为地址更新，即 user_{devfile}_addr_update 保持低电平。

3.5 quiesce 信号

当 hosts 预计 IP core 完全不活动 (quiescent state) 时， quiesce 信号为高电平。这通常是在：

- host 尚未加载 driver，或者已将其卸载。
- 在 Windows上： 当 host 即将进入 hibernation。
- 使用 XillyUSB： 此外，当设备根本没有连接到计算机时。

该信号的目的是用作 synchronous reset，但很可能没有必要： 在 quiescent state 中的副作用之一是所有文件都已关闭，因此 application logic 可以单独依赖 *_open 信号作为 reset 信号。 'quiesce' 信号可以用作 reset的更全局形式。

4

实施 data acquisition

4.1 介绍

经常需要将数据从 FPGA 捕获到计算机，例如出于以下需求：

- 从视频源抓取帧
- 来自模数转换器 (ADC) 的数据样本
- 从其他数据源获取数据
- 从 FPGA接收调试信息

对于这样的应用，数据速率可以很高，并且必须保证数据流的连续性： 不允许任何数据丢失。

通过将数据写入 FIFO，使用 Xillybus 可以轻松实现 data acquisition 应用程序。本节重点介绍如何实现 application logic ，以保证到达 host 的数据是连续的。为此， application logic 在连续性中断的点停止数据流，并在该点发送 EOF 。这样，application software 可以依赖到达的数据确实是连续的。

理想情况下，这种停止机制永远不应该激活，但是当它激活时，它可以让人们意识到问题，以及解决问题的机会。

从理论上讲，确保外围设备和计算机之间的持续数据速率是不可能的，因为操作系统可能会尽可能长时间地剥夺 CPU 与 application software 的联系。

尽管如此，仍然有一些方法可以保持数据的连续 stream ，其中涉及某些 host 编程技术。这个问题在两个编程指南中都有广泛的讨论：

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

特别要注意这两个指南的第 4 节，其中讨论了如何处理高数据速率。

对于高带宽应用，还建议参考这两个指南之一的第 5 节，其中包含需要注意的几个主题：

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

下面将展示如何使用 Xillybus 从连续源捕获 32 位宽的数据。本节的重点是确保到达 host 的所有数据都是捕获数据源的可靠副本。

4.2 示例代码

下面显示和解释的示例代码可以从这个链接作为模块下载：

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

zip 文件由 xillycapture.v 和 xillycapture.vhd两个文件组成。这些分别用 Verilog 和 VHDL编写。为了试用该示例，请编辑 xillydemo.v 或 xillydemo.vhd： 断开 demo bundle中与 read_32 相关的信号，插入示例代码。

示例代码生成标准的 32 位宽 dual clock FIFO的 instantiation 。在尝试执行示例代码的 synthesis 之前，请使用工具（例如 Vivado 或 Quartus）生成此 FIFO 。这个 FIFO 的名字应该是 async_fifo_32，深度512就够了。

请注意，在示例代码中，有一个名为 “slowdown”的信号。该信号的目的是降低假数据源的数据速率。当使用真实数据源时，应删除此信号。

4.3 FIFO 连接

假设数据源与 capture_clk同步。因此，数据被简单地馈送到标准 dual-clock FIFO。此 FIFO 连接数据源和 Xillybus IP core。

在 Verilog中：

```
 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)
    );
```

在 VHDL中：

```
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;
```

这与 demo bundle非常相似： FIFO 文件关闭时复位，其 user_r_read_32_* 信号连接如前。

4.4 捕获控制

capture_en 信号用作捕获数据的 write enable 信号。在以下两种情况之一中不会发生数据捕获：

- 文件关闭时
- 当 FIFO 已满或过去已满时

因此 capture_en （在 Verilog中）的条件归结为：

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

在 VHDL中：

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

capture_open 信号是 user_r_read_32_open的副本，但在 capture_clk的 clock domain 中。

其他特定于应用程序的条件，例如等待视频数据中的帧开始，或在使用 data acquisition 进行调试时等待某个错误条件，可以根据需要添加到此表达式中（通过 logic AND）。

capture_has_been_full 信号在 FIFO 满时变为高电平，只有在文件关闭时才返回低电平。因此，当 FIFO 已满时，数据捕获将停止，并且只要打开文件就不会恢复。

重要的:

在示例代码中， *capture_en*有不同的定义，这有助于减慢虚假数据源的速度。捕捉真实信号时， *capture_en* 应改为上述。

现在来看在 Verilog中实现 capture_has_been_full 的代码：

```
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;
```

这几乎与预期的一样： 当 FIFO的 capture_full 变高时， capture_has_been_full 变高，当文件关闭时它变低。另一个信号 capture_has_been_nonfull解决了一个不同的问题： 由于 FIFO 在复位时保持 'full' 信号为高电平，因此只有当 capture_full 为低电平（表示 FIFO 退出复位）然后变为高电平（表示 FIFO 已满）时， capture_has_been_full 才应为高电平。

所以这段代码有点复杂，但是一旦理解了原理，就很简单了。

4.5 生成 EOF

当满足以下两个条件时，会生成 end-of-file：

- FIFO 中的所有数据都已被消耗（即已被 IP core读取）。
- 不会再向 FIFO写入数据，因为它过去已满。

在 Verilog中，这写为：

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

在 VHDL 中（注意这是组合的）：

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

从示例代码中可以看出， has_been_full 是 clock domain crossing 到 bus_clk之后的 capture_has_been_full 。

请注意， user_r_read_32_eof 仅在允许的情况下根据 API从低到高。这是因为有一个 logical AND 和 user_r_read_32_empty，正如 3.3部分所建议的那样。

4.6 试运行

重要的:

此测试运行故意显示了 *IP core* 设置的不良示例，因此 *EOF* 的机制开始起作用：用于此测试的 *IP core* 具有较小的 *buffers* ， 而 *stream* 是同步的（这对于 *data acquisition* 应用程序是错误的）。现实生活中的测试效果明显更好。

为了保证捕获数据的可重复性，数据源被选为虚假数据生成器，它只统计发送的字数。 EOF 之前的数据量取决于计算机何时忙于做其他事情，并且暂时忽略了从 device file读取的顺序。

测试运行显示为 Linux，但它也可以在 Windows 上运行。可以在以下任一指南中找到有关运行命令行实用程序的更多信息：

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

这是测试运行的样子：

```
$ 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
```

因此，第一次尝试捕获了大约 71 MB ，但第二次仅捕获了 7 MB 。每次运行的数据量取决于在操作系统忽略读取过程执行其他操作之前接收到的数据量。最有可能的是，为了写入磁盘，读取过程被短暂停止。

但即使通过将所有数据发送到 /dev/null来丢弃所有数据，它最终也会停止（尝试 “man dd” 了解有关 dd 实用程序的更多信息）：

```
$ 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
```

在这两个测试中，移动鼠标都会停止数据流。这足以分散操作系统的注意力。

再次强调： 这些都是非常糟糕的结果，因为使用了同步 stream 。使用异步 stream和正确数量的 DMA buffers，根本不会出现此类问题。

最后，我们将查看其中一个捕获的文件中的内容：

```
$ 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  |................|
```

正如所料，数据包含一个向上计数的序列。用于生成虚假数据的计数器永远不会重置，这就是序列不从 0 开始的原因。

4.7 监控缓冲数据量

通常需要跟踪在 Xillybus的 buffers 中保存了多少属于某个 stream的数据。这有助于控制 latency，防止 overflow 或 underflow，或防止 application software 在函数调用 read() 或 write()期间休眠。

例如，在从 FPGA 到 host的方向上，这意味着知道从 FPGA 中的 FIFO 读取了多少数据（由 Xillybus IP core），这些数据尚未被 host上的 application software 消耗。

同样，在相反的方向上，这意味着知道 application software 向 stream写入了多少数据，而在 FPGA 中还没有到达 FIFO （因为 FIFO 已满，等待数据被 application logic消耗）。

Xillybus 没有为此提供专用功能，部分原因是使用 Xillybus的功能有一种简单的方法可以实现此功能，如下所示。

为了解释建议的解决方案，假设从 FPGA 到 demo bundle 中的 host 的 32-bit stream 用于 data acquisition。

以下计数器用于计算自文件打开以来 IP core 从 FIFO 获取的数据元素的数量：

```
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;
```

为此目的， count_data 的值可以通过另一个专用 Xillybus stream （从 FPGA 到 host）暴露给 host： 这是通过将 count_data 直接连接到另一个 stream的 data port （即通常连接到 FIFO的数据输出的 port ）来完成的。

'eof' port 和 'empty' port 应始终保持在低位。通过将 IP Core Factory 中的 "use" 参数设置为 "Command and status"，这个专用的 stream 被配置为同步的。

这样， application software 可以随时从这个 stream 中读取 4 个字节，得到 count_data的更新值。

或者，在适当的情况下， count_data 可以是 registers数组中的 register ， 如 3.4部分中所建议的那样。

请注意， count_data 与 bus_clk同步，因此可以直接连接到 Xillybus IP core的 data port。

一旦软件通过这个额外的 stream知道了这个值， buffers 中的数据量可以计算为 count_data与 application software 自打开以来从其 device file 读取的数据量之间的差值（即本例中为 /dev/xillybus_read_32 ）。当然，这需要软件跟踪从 stream读取的数据量。

在相反的方向，从 host 到 FPGA，可以在 FPGA 中维护一个类似的计数器

```
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;
```

通过同样的原理， application software 会跟踪它向相关 device file写入了多少数据，并通知自己在 FPGA上向 FIFO 写入了多少元素。这是通过读取 count_data的值来完成的。

在上述两个案例研究中， FIFOs 中的数据均未包含在计算中，而仅包含 Xillybus 保留在 buffers中的数据。有时需要获取端到端的编号，包括存储在 FIFOs中的数据。为此，应计算 FIFOs 对面的操作，即第一种情况下写入 FIFO 的元素数量，以及第二种情况下从 FIFO 读取的元素数量。

但是，如果 FIFO 的另一端与不是 bus_clk 的 clock 同步（例如本节前面介绍的 capture_clk），这可能更难实现。这是因为 count_data 也因此与其他 clock 同步。因此，需要 clock domain crossing 将 count_data的值连接到 IP core。因此，需要在准确性和简单性之间进行权衡，除非 bus_clk 本身就是用于 data acquisition 或播放的 clock 。

5

建议的模拟实践

5.1 一般的

什么是令人满意的模拟取决于品味和工作实践。然而，在模拟中总是假设某些事情按预期工作。也可能有某些有趣的事情可以模拟，但执行起来太复杂或太耗时。

本节提出了一组关于模拟过程的假设和限制，以及模拟涉及 Xillybus IP core的系统的方法。就其性质而言，这些指南不如本文档其余部分中的指南那么重要。

Xillybus IP core 及其 driver 是一个复杂的系统，已经在各种场景下进行了压力测试。因此，如果不是在 TB 级的数据传输中发现错误，并且负载模式范围很广，则不太可能通过模拟在 IP core 本身中找到错误。

此外， IP core的行为很大程度上取决于 host的响应： driver 和 application software 都以不同的方式和不同的延迟做出响应，这几乎是不可预测的。最重要的是， bus 的 latency （PCIe、 AXI 或 USB）同样是随机的，因此无法预测。因此，全面的模拟几乎是不可能的。

鉴于此，建议将 application logic 模拟到 FIFO 与 Xillybus IP core连接的位置。因此，IP core 被模拟为 black box ，它会根据数据的方向排出或填充 FIFO 。

5.2 模拟异步 streams

当 stream 配置为异步时， IP core 与 FIFO 之间传输数据（取决于 stream的方向），因此 FIFO 永远不会达到 overflow 或 underflow的状态。

只要 host 上的应用软件足够频繁地执行 I/O 操作，并且 Xillybus的带宽能力足以完成其任务，这就是正确的。这两个条件是正确设计项目的结果，通过模拟验证它们可能是有益的。有两个方面要看：

- FIFO 是否到达 overflow 或 underflow （取决于方向）。
- application logic 是否正确响应此类故障情况，例如，如 4.5部分中所建议的那样。

为了模拟正确的操作，可以假设 IP core 以最大速率向 FIFO 传输数据或从 FIFO 传输数据，只要相关的 'open' 信号为高电平（表示文件由 host打开）。

为了测试从 host 到 FPGA的 stream ，当 FIFO 遭受 underflow的影响时会发生什么，建议通过使 FIFO 看起来为空来模拟此事件。例如，如果 FIFO 是 test bench的一部分，它会将 'empty' 信号（连接到 application logic）变为高电平。或者， test bench 中模拟来自 host 的数据流的部分可能只是停止向 FIFO 推送数据一段时间，这会导致它变为空。

同样对于从 FPGA 到 host的 stream ， 'full' 线可以更改为高，以测试 FIFO的 overflow。或者， test bench 可以在一段时间内停止从 FIFO 获取数据，产生相同的效果。

破坏数据连续性的一种可能性是因为 application logic 试图超过 stream 的带宽限制（或 IP core的总带宽）。如果这是可能的（在许多应用中并非如此），还建议 test bench 模拟带宽限制。它可以通过使用受 stream预期带宽限制的数据速率填充或清空 FIFO 来实现。

5.3 模拟同步 streams

与异步 stream相比，同步 stream 的不同之处（出于模拟目的）在于 IP core的数据流不连续： 仅当 host （read() 或 write() ）上有挂起的函数调用时， IP core 才会将数据传输到 FIFO 或从 FIFO 传输数据。

因此， IP core的行为在很大程度上取决于应用软件对 I/O的请求。因此，在编写模拟 IP core 的 test bench 部分时，必须考虑应用软件的访问模式。

因为当 stream 的目的是交换大量数据时，同步 stream 不是首选选项，所以 overflow 或 underflow 的可能性可能不那么相关。然而，模拟这些条件的方法与异步 streams相同。

5.4 一种简化的模拟方法

那些对模拟 overflow 和 underflow的条件不感兴趣的人，有一个更简单的选项来模拟 IP core。例如，在 host 到 FPGA 方向上， FIFO 可以在 test bench 中实现，只需在 read enable 信号为高电平时从每个 rising clock edge 的文件中读取数据字即可。

FIFO 的这种简化视图依赖于 host 将始终以足够快的速度将数据写入 FIFO 以确保它永远不会变空的假设。

在相反的方向，当 write enable 信号为高电平时， test bench 将字写入文件。和以前一样，这假设 host 总是以足够快的速度从 FIFO 读取数据，以确保它永远不会被填满。

这种方法并没有忽视连续性可能中断的可能性。相反，它认识到破坏的数据连续性很可能是超出模拟范围的结果： DMA buffers太浅， application software响应能力差，或者 host的整体状况导致 CPU 被剥夺。如果这样的事件真的发生， application logic 应该让 host 意识到这一点。如上所述，可以模拟这种机制。

然而，这种方法确实忽略了 application logic 试图超过 stream的带宽限制的可能性。如果这种情况是现实的可能性，那么这种简化的模拟选项可能还不够。