(机器翻译成中文)

Xillybus FPGA designer's guide

Xillybus Ltd.
www.xillybus.com

Version 3.2

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 This document may also be slightly outdated in relation to the original.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 **3**

2 一般准则 **5**

2.1 Clocking 5

2.2 数据宽度 6

2.3 通过 FIFO 连接 6

2.4 “empty” 和 “full” 信号的行为 7

3 信号说明 **8**

3.1 FPGA 信号的命名约定 8

3.2 host 到 FPGA 传输的信号 8

3.3 FPGA 到 host 传输的信号 9

3.4 内存接口信号 11

3.5 quiesce 信号 13

4 实施 data acquisition **14**

4.1 介绍 14

4.2 示例代码 15

4.3 FIFO 连接 16

4.4 Data acquisition控制 17

4.5 生成 EOF 19

4.6 试运行 20

4.7 监控缓冲数据量 21

5 simulation 的建议方法 **24**

5.1 一般的 24

5.2 模拟asynchronous streams 24

5.3 模拟synchronous streams 25

5.4 simulation的简化方法 25

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，通常无需详细了解 API 即可使用 IP core。

API的绝大多数规则都可以从一个简单的原理推导出来： user application logic 的行为应与 FIFO 或 block RAM 完全相同。即使 application logic 直接与 IP core 连接也是如此。

除此之外，这一原则意味着 Xillybus IP core 和 user application logic 之间的数据交换按照 IP core 规定的速度进行。数据流可能会暂时停止并稍后以不可预测的方式恢复。在其他情况下，数据流将是连续的。当 IP core 直接连接到 FIFO 或 block RAM 时，这不会造成任何问题。同样，即使数据流以不可预测的方式开始和停止，直接与 IP core 连接的 user application logic 也应该正常运行。

将 IP core 的不规则访问模式视为错误是一个常见的错误。 IP core 和 FIFO 之间的数据流出现无法解释的暂停可能看起来很可疑，但它们并不表示有任何故障。

不建议直接与 IP core 连接（即不使用 FIFO 或 block RAMs）以减少 logic consumption。这在 design 的早期阶段尤其如此。如果user application logic直接连接到IP core，不规则的数据流可能会暴露application logic中的错误。

本指南介绍了与直接连接 application logic 相关的 API，并详细介绍了流行的应用程序。

建议先获得 Xillybus 的初步体验，然后再深入研究本指南中介绍的详细信息。请参考这些文档：

- 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

本指南还假设您了解同步和异步 streams 之间的区别。这两个文档的第 2 节对此进行了讨论：

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

XillyUSB IP cores 公开相同的 API，并且是 Xillybus IP cores 的子集。因此，除非另有说明，否则本指南中的名称 “Xillybus” 也指 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

所有往返于 Xillybus IP core 的 signals 必须与 bus_clk 的 rising edge 同步。该 clock 由 IP core 提供。

对于基于 PCIe 的 Xillybus IP cores，该 clock 由 PCIe 块生成。 clock 的频率取决于平台： 对于 baseline IP core（修订版 A），bus_clk 的频率为 62.5 MHz、125 MHz 或 250 MHz。这取决于最大带宽（如广告所示）是否分别为 200 MB/s、400 MB/s 或 800 MB/s。

随着后来的修订（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 与 125 MHz 的 bus_clk 配合使用。

通常可以在有限的选择列表中更改 clock 的频率。这是通过配置 PCIe 块或生成 clock 的 processor core 来完成的。

如果 PCIe 块的 timing constraints 设置正确（如 demo bundles 中），依赖于 bus_clk 的 application logic 也会被正确的 timing constraints 覆盖： 该工具会自动为基于 PCIe 块的 timing constraints 的 bus_clk 创建 timing constraints。这同样适用于基于 Zynq 的平台以及 XillyUSB。

这并不是说 application logic 需要与 bus_clk 同步。同样，数据的来源或数据的目的地也不需要与 bus_clk 同步。当涉及不同的 clock 时，dual-clock FIFO 通常与 IP core 一起使用： FIFO 的一侧连接到 Xillybus IP core。因此，这一侧与 bus_clk 同步。 application logic 连接到 FIFO 的另一侧。此端与application logic的clock同步。因此，FIFO不仅可以用作短期临时存储，还可以用于clock domain crossing。

2.2 数据宽度

每个 FIFO 或内存接口都可处理 8 位、16 位或 32 位宽度的数据。基线 Xillybus IP cores（修订版 A）也是如此。后来的版本以及 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，两侧都连接到 IP core。这在两个 streams 上实现了一个 loopback。

demo bundle中的FIFOs配置为两侧通用的clock。当 FIFO 用于 clock domain crossing 时，这不适合。在这种情况下，应使用 dual-clock FIFO（通常称为 “asynchronous FIFO”）。

当FIFO用于从host到FPGA的stream时，该FIFO的“full”信号应连接到Xillybus IP core。IP Core 使用该信号来确定是否可以启动突发数据传输。

同样的原理也适用于从 FPGA 到 host 的 stream： 出于相同目的，“empty” 信号应连接到 Xillybus IP core。 IP core 预期具有常规 FIFO 的行为（而不是 FWFT、First Word Fall Through）。

一旦突发开始，Xillybus IP core 继续依赖这些信号（“empty” 和 “full”）： 这些信号防止 IP core 从空的 FIFO 读取或写入完整的 FIFO。

然而，即使 FIFO 指示它已准备好突发数据，Xillybus IP core 也可能不会立即开始突发。即使 FIFO 允许继续突发，IP core 也可能会在中间停止数据突发。数据流的模式明显是随机的，这是正常的。

一般规则是 Xillybus IP core 尝试平等地服务于与其连接的所有 FIFOs。 IP core 为 FIFOs 提供更长的突发，往往会更快地填充，因为这些 FIFOs 不会经常激活其 “empty” 或 “full”。

这种简单的仲裁方法可确保与容易快速填满的 FIFOs 进行高效通信。同时在FIFOs上实现了以较低速率接收数据的低latency。

至于FIFO的深度，原则上Xillybus IP core适用于任何深度。然而，应该选择该属性来应对预期的数据流。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。但这有时是一个反复试验的问题。

2 kBytes 的深度通常就足够了，因为 Xillybus core 不太可能在足以导致 overflow 或 underflow 的时间段内忽略此尺寸的 FIFO。只要在 host 上运行的 user application software 消耗或提供数据的速度足够快，这当然是正确的。如果不是这种情况，解决方案可能是加大 DMA buffers 的尺寸。试图用更大的 FIFO 来解决这个问题是不合理的，因为 FPGA 上的内存要少得多。

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，这两个信号可以随时变为低电平。

Xillybus IP core 依赖于这种行为： 当 FIFO 指示已准备好进行数据传输时（如果适用，使用低 “empty” 或 “full”），IP core 中的 state machine 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 “empty” 信号在 IP core 从 FIFO 获取任何数据之前变为高电平，则 IP core 可能会在 clock cycle 期间忽略 “empty” 信号。对于 IP core 自身的完整性而言，此类事件是无害的，但可能会导致意外且不可预测的数据流。

这同样适用于 “full” 信号： 如果在 IP core 将数据字写入 FIFO 之前该信号从低电平变为高电平，则 IP core 可能会在一次 clock cycle 期间忽略 “full” 信号。再次强调，这对 IP core 本身无害，但可能会导致一个数据字的丢失。

正确设计的 FIFO 只有在准备好与 Xillybus IP core 通信的同时复位时才会产生这种故障情况。无论如何，这种情况通常应该避免。

如果 application logic 直接与 IP core 连接（无需 FIFO），则模仿标准 FIFO 关于 “empty” 或 “full” 的行为非常重要。

3 信号说明

3.1 FPGA 信号的命名约定

除了 bus_clk 和 quiesce 这两个全局信号外，所有信号都遵循简单的约定。例如，write enable 信号可能具有名称 user_w_write_32_wren。该名称分为四个部分：

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从host到FPGA（host “write”）的stream。 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，host 的观点是针对 “w” 或 “r” 的选择而采取的。
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。

每个信号名称后跟有 (IN)，表示该信号是 IP core 的输入，如果是 IP core 的输出，则表示该信号是 (OUT)。

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 IP core 不能再写入数据。

  重要：仅在 clock cycle 上，'full' 信号在写入周期后可能从低电平变为高电平。所有标准 FIFOs 的行为都是如此，因此仅当 IP core 直接与 application logic 连接（即中间没有 FIFO）时，此规则才相关。

  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打开用于写入时，该信号为高（如果文件打开为只读，则在允许的情况下，不会改变该信号）。该信号可用于在文件关闭时重置 FIFO 或其他 logic（用作 active low reset）。

  如果 host 上的多个 processes 打开文件（例如，由于调用 fork() 函数），则该信号将保持高电平，直到所有 processes 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期间的数据。仅当 FIFO 需要更改该信号时才允许更改该信号。换句话说，只有在 user_r_{devfile}_rden 为高电平后，它才可能在 clock cycle 上发生变化。

- user_r_{devfile}_rden (**OUT**) – 该信号是 FIFO 的 read enable 信号： 当该信号为高电平时，user_r_{devfile}_data 必须包含后续 clock cycle 上的有效数据。

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

  重要： 仅在 clock cycle 上，'empty' 信号在读周期后可能从低电平变为高电平。所有标准 FIFOs 的行为都是如此，因此仅当 IP core 直接与 application logic 连接（即中间没有 FIFO）时，此规则才相关。

  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**) – 该信号告诉 core 生成 end-of-file。高 'eof' 的结果也是 core 将不再从 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 将在此信号变高之前完成读取 IP core 接收到的所有数据。只有这样，host 在调用 read() 函数时才会收到 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' 信号类似，只有 clock cycle 上的 'eof' 信号在读周期后才允许变为高电平。但有一个例外： 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' 符合此规则的一种可能性是将 'eof' 定义为 combinatorial function 的输出。在Verilog中，可能会这样写：

```
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 打开读时，该信号为高（如果文件打开为只写，在允许的情况下，不会改变该信号）。该信号可用于在文件关闭时重置 FIFO 或其他 logic（用作 active low reset）。

  如果 host 上的多个 processes 打开文件（例如，由于调用 fork() 函数），则该信号将保持高电平，直到所有 processes 都关闭该文件。

  'eof'信号和'open'信号之间没有直接连接。当 host 上的文件关闭时，'open' 信号将变为低电平，与 'eof' 无关。但请注意，application software 通常通过关闭文件来响应 EOF。因此很容易错误地认为这些信号之间存在联系。

3.4 内存接口信号

Xillybus device file 可配置为具有地址信号。 application software 通过使用文件中 seeking 的标准 API 为该信号分配一个值（例如 lseek()）。此外，由于读取周期和写入周期，FPGA 上会自动出现地址的 increment。

标准 block RAM 可轻松与 IP core 连接。这是通过使用上面已经提到的与 FIFOs 有关的信号以及下面详细介绍的信号来完成的。结果是 block RAM 的内存阵列可作为文件供 host 使用： 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内存阵列的段： 这取决于读或写操作的长度。

还可以实现 registers 阵列，其行为类似于 FPGA 上的 block RAM。通过这样做，host 可以轻松访问这些 registers。

'empty' 和 'full' 信号可用于减慢需要 wait states 的存储器的读写操作，或者当有其他原因短暂延迟操作时。

存储器接口需要两个附加信号：

- user_{devfile}_addr (**OUT**) – 该信号包含当前时间的地址。当read enable为高电平时，需要从此地址进行读操作。当write enable为高电平时，需要对该地址进行写操作。将此信号直接连接到 block RAM 的 address input 将按预期工作。该信号的宽度最多可配置为 32 位。

  在最大地址（取决于地址信号的宽度）处执行读或写操作后，地址值返回零。如果对 lseek() 的函数调用的值超出范围，则仅将 LSBs 复制到该信号。

- user_{devfile}_addr_update (**OUT**) – 由于对 host 上的 lseek() 进行函数调用，该信号在一个 clock cycle 期间为高电平。当 'addr' 信号具有更新值时，'update' 信号在同一 clock cycle 上为高电平。

  该信号的目的是让 application logic 有机会表明由于地址更新，它需要时间来准备读取数据。这是通过将 'empty' 信号更改为高电平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： 在 'update' 为高电平之后，它也可以在 clock cycle 上更改为高电平。

  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;
```

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）时将 'empty' 更改为高电平是合理的，然后让 logic 评估 'empty' 是否可以更改回低电平。

'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（即当user_w_{devfile}_open和user_r_{devfile}_open均为低电平时），地址被重置，因此其值变为零。但请注意，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 host 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host已卸载它。
- 在 Windows 上： 当host即将进入hibernation。
- 使用 XillyUSB： 此外，当设备根本未连接到计算机时也是如此。

该信号的目的是用作 synchronous reset，但该信号很可能不是必需的： 当IP core处于非活动状态（即quiescent state）时，所有文件都被关闭。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

经常需要从 FPGA 捕获数据到计算机，例如：

- 来自 video 信号源的 Frame grabbing。
- 来自模数转换器 (ADC) 的数据。
- 从 FPGA 接收调试信息。

对于此类应用，数据速率可能很高。尽管如此，必须保证数据流的连续性： 不允许丢失数据。

通过将数据写入 FIFO，可以使用 Xillybus 轻松实现 data acquisition 应用程序。本节重点介绍如何保证到达host的数据是连续的。

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维护连续的 stream 数据。实现此目标的首要条件是使用能够以所需速率传输数据的 Xillybus stream。最重要的是，应该使用某些 host programming 技术。两个编程指南中都广泛讨论了这个问题：

- 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

但即使design完美执行，stream的数据连续性也总有被打破的可能： 操作系统的本质是允许CPU长时间脱离application software。

因此，第一个目标是确保 stream 数据的连续性实际上永远不会被破坏。第二个目标是确保即使尽一切努力仍发生这种情况，该事件也会被注意到。更重要的是，保证到达 host 的所有数据都是连续的。

为了实现第二个目标，application logic 应在连续性被破坏的点停止数据流。此后，EOF 会发送到 host，以便告诉 host 出现了问题。这样，application software 就可以相信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被激活。但当它发生时，它可以让人们意识到问题，并有机会解决它。

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

4.2 示例代码

有两种方法可以修改 design 以包含停止机制:

- 使用经过修改的 FIFO,如果连续性损坏,它会停止数据流。此修改版 FIFO 是标准 FIFO 的包装。它还产生 EOF 信号。
- 对使用 FIFO 的 logic 进行修改。

下面显示和解释的示例代码可以从该 link 下载:

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

zip 文件由三个文件组成:

- eof_fifo.v,用Verilog编写
- xillycapture.v,用Verilog编写
- xillycapture.vhd,用VHDL编写

有两种方法可以尝试该示例。在这两种方式中,您都应该编辑 xillydemo.v 或 xillydemo.vhd。

第一种可能性是使用 eof_fifo.v: 将 fifo_32x512 的 instantiation 替换为 eof_fifo 的 instantiation。然后将user_r_read_32_eof连接到这个FIFO的eof port。

除此之外,建议生成数据源而不是 loopback。这可能是仅用于测试的虚假数据源(例如 counter)。

第二种可能性是使用 xillycapture.v 或 xillycapture.vhd: 断开 demo bundle 中与 read_32 相关的信号,并将示例代码插入到这些文件之一中。

请注意,在这两个文件中,有一个名为 “slowdown” 的信号。该信号的目的是降低虚假数据源的数据速率。当使用真实的数据源时,应删除该信号。

在这两种可能性中,示例代码执行标准 dual clock FIFO 的 instantiation。这个FIFO的宽度是32 bits。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。这款FIFO的名字应该是async_fifo_32。 512字的深度就足够了。

本节的其余部分基于 xillycapture.v 和 xillycapture.vhd。但所解释的原理也与理解 eof_fifo.v 相关。

4.3 FIFO 连接

假设数据源与capture_clk同步。因此，数据以常规方式连接到标准 dual-clock FIFO。该 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：当文件关闭时，FIFO 复位，其 user_r_read_32_* 信号的连接与 demo bundle 中一样。

4.4 Data acquisition控制

capture_en 信号作为 write enable 信号工作。有以下三种情况会阻止向 FIFO 写入数据：

- 文件关闭时
- 当FIFO满时
- 当FIFO过去已满时，自从文件被打开

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capture_clk的clock domain的user_r_read_32_open的副本。

在实际应用中，写入 FIFO 常常存在其他条件。例如，等待video frame的开始，或者

等待特定的错误条件（当使用data acquisition进行调试时）。可以根据需要将此类条件添加到此表达式中（借助 logic AND ）。

当FIFO满时，信号capture_has_been_full变为高电平，只有当文件关闭时，信号capture_has_been_fu
恢复为低电平。因此，当 FIFO 已满时，data acquisition 会停止，并且只要文件保持打开状态就不会再次启动。

重要的:

在示例代码中，*capture_en* 有不同的定义，这有助于减慢假数据源的速度。实际应用时，*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;
```

当FIFO的capture_full变高时，capture_has_been_full变高。当文件关闭时，capture_has_been_full变低。

另一个信号 capture_has_been_nonfull 解决了另一个问题：只要FIFO复位，FIFO的'full'信号就为高电平。由于这个原因，当'full'信号为高电平时，capture_has_been_full不应为高电平。换句话说，只有当 capture_full 一直处于低电平（意味着 FIFO 退出复位）然后又变为高电平（意味着 FIFO 确实已满）时，capture_has_been_full 才应该处于高电平。

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中的所有数据已被消耗（即所有数据已被IP core读取）。
- 不再有数据写入FIFO，因为FIFO过去已经满了。

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 combinatorial function）：

```
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 借助 clock domain crossing 将 capture_has_been_full 的值复制到 bus_clk。

请注意，user_r_read_32_eof 在 API 允许的情况下从低到高。这是因为 user_r_read_32_empty 中有一个 logical AND，如 3.3 部分中所建议的。

4.6 试运行

重要的:

此测试运行故意展示了 *IP core* 配置不合适的不良示例。这个故意错误的目的是为了演示 *EOF* 如何发挥作用。用于此测试的 *IP core* 具有小型 *buffers* 和 *synchronous stream*。对于 *data acquisition* 应用程序来说，这些都是不正确的选择。正确配置的 *IP core* 的性能不会像下面所示的那么差。

为了保证传输数据的可重复性，数据源选择为一个简单的计数器，对发送的字数进行计数。 EOF之前的数据量是随机的： 当计算机忙于做其他事情并暂时忽略从 device file 读取数据的任务时，就会出现 EOF。

显示的是针对 Linux 的测试运行，但它也可以在 Windows 上运行。有关运行 command line utilities 的更多信息可以在以下任一指南中找到：

- 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，第一次尝试时收集了大约 71 MB，但第二次尝试时仅收集了 7 MB。每次运行中的数据量取决于操作系统忽略读取过程以执行其他操作之前接收到的数据量。最有可能的是，读取过程被短暂停止以写入磁盘。

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
```

在这两项测试中，移动计算机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： 这些结果确实很糟糕，因为使用的是 synchronous stream。使用 asynchronous 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  |................|
```

正如预期的那样，数据包含一个向上计数序列。用于生成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 Xillybus 和 buffers 中属于特定 stream 的数据量。这可以帮助控制 latency、防止 overflow 或 underflow，或防止 application software 在对 read() 或 write() 的函数调用期间休眠。

例如，关于从 FPGA 到 host 的数据流： buffers 中可能存储有大量数据，因为 IP Core 已从 FPGA 中的 FIFO 读取了此数据，但 application software 尚未消耗此数据。通常需要知道有多少数据正在等待这样的情况。

同样，在相反的方向： application software可能已经将数据写入stream，但尚未到达FPGA中的FIFO。直接原因是FPGA中的FIFO已满，因此无法从IP core接受更多数据。然而，真正的解释是数据正在等待application logic消耗。

Xillybus 不提供用于估算 buffers 中数据量的专用功能。不过，有一种简单的方法可以利用 Xillybus 的现有功能来实现此功能，如下所示。

为了解释建议的解决方案，假设 demo bundle 中的 streams（FPGA 至 host，32 位）之一用于 data acquisition。

以下计数器用于计算自文件打开以来从 FIFO（由 IP core）获取的数据字数：

```
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 可以是 registers 阵列中的 register，如 3.4 部分中所建议的。

另一种解决方案是在 IP core 上添加另一个 Xillybus stream（从 FPGA 到 host）。该 stream 用于通过将 count_data 直接连接到该附加 stream 的 data port（即通常连接到 FIFO 的数据输出的 port）来将 count_data 的值发送到 host。

stream 的 'eof' port 和 'empty' port 应始终保持在低电平。通过将 IP Core Factory 中的 "use" 参数设置为 "Command and status"，该 stream 应配置为 synchronous stream。这样，application software就可以随时从这台stream中读取4个字节，从而得到count_data的更新值。

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。

buffers 中的数据量可以计算为 count_data 与 application software 自打开以来从 device file（即本例中的 /dev/xillybus_read_32）读取的数据量之间的差值。当然，软件必须跟踪从 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;
```

这遵循相同的原理： application software 跟踪其写入相关 device file 的数据量。当需要知道buffers中存储了多少数据时，application software读取count_data。该数据量的计算方式为已写入的数据量（自打开以来写入 device file）与 count_data 的值之间的差值。

请注意，在到目前为止的讨论中，FIFOs 中的数据并未包含在计算中： 仅考虑 Xillybus 在 buffers 中保存的数据。有时需要获取端到端号码，包括存储在 FIFOs 中的数据。为此，需要计算FIFOs对面的操作。换句话说，这是从 FPGA 到 host 的 stream 写入 FIFO 的元素数量。在相反的方向上，这是从 FIFO 读取的元素数量。

但是，如果 FIFO 的另一端与不同的 clock（例如前面介绍的 capture_clk）同步，则这可能更难实现。这是因为 count_data 也需要与另一个 clock 同步。因此，需要 clock domain crossing 将 count_data 的值连接到 IP core。因此，当两个不同的 clocks 连接到 FIFO 时，需要在准确性和简单性之间进行权衡。

5 simulation 的建议方法

5.1 一般的

什么是令人满意的simulation是品味和工作方法的问题。尽管如此，总是针对 simulation 做出假设。这些假设包括特定功能元素按预期工作的期望。因此，用 simulation 检查这些功能元件是没有意义的。某些功能元素也可能有利于模拟，但这样做过于复杂或耗时。

本节提出了有关 simulation 流程的一些假设和限制。还讨论了涉及 Xillybus IP core 的系统的 simulation 的方法。这些准则本质上比本文档其余部分的准则更易于讨论。

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广泛的测试。因此，在 simulation 的帮助下不太可能在 IP core 本身中发现错误： 如果在 TB 级数据传输和大范围使用模式下未发现错误，则 simulation 不太可能发现此类错误。

此外，IP core 的行为很大程度上取决于 host 的响应： driver 和 application software 的响应方式不同，delays 也不同，这几乎是不可预测的。最重要的是，bus（PCIe、AXI 或 USB）的 latency 同样是随机的，因此不可预测。因此，全面的 simulation 几乎是不可能的。

有鉴于此，建议在 FIFO 与 Xillybus IP core 连接之前执行 simulation 或 application logic。因此，IP core 被模拟为 black box，根据数据的方向耗尽或填充 FIFO。

5.2 模拟asynchronous streams

当 stream 配置为 asynchronous 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 的带宽能力足以满足其任务，这一点就成立。这两个条件是正确设计的项目的结果。借助 simulation 验证

design 的两个方面可能会有好处：

- 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）。

对于从 host 到 FPGA 的 stream，测试 FIFO 遇到 underflow 时会发生什么情况是有益的。建议通过使 FIFO 看起来变空来模拟此事件。例如，如果 FIFO 是 test bench 的一部分，则 test bench 将 'empty' 信号（连接到 application logic ）更改为高电平。或者，test bench 中模拟来自 host 的数据流的部分可能会停止将数据推送到 FIFO 一段时间。 FIFO 会因此变空。

同样，对于从 FPGA 到 host 的 stream： 可以将'full'线改为高电平，来测试FIFO的overflow。或者，test bench 可以停止从 FIFO 获取数据一段时间，产生相同的效果。

破坏stream数据连续性的一种可能原因是application logic试图超出stream的带宽限制（或IP core总带宽的限制）。如果存在这种可能性，还建议test bench模拟带宽限制。这可以通过确保 test bench（充当 IP core ）以受 stream 预期带宽限制的数据速率填充或清空 FIFO 来完成。

但请注意，在许多应用中，此类 simulation 是不必要的，因为 application logic 无法超出带宽限制。

5.3 模拟synchronous streams

就 simulation 而言，synchronous stream 的主要区别在于 IP core 的数据流不是连续的： 对于 synchronous stream，仅当 host（read() 或 write()）上有挂起的函数调用时，IP core 才会与 FIFO 传输数据。

因此，IP core 的行为更多地依赖于 application software 对 I/O 的请求。因此，test bench 中模拟 IP core 的部分在编写时必须考虑到 application software 的访问模式。

对于 synchronous stream 模拟 overflow 或 underflow 可能无关紧要，因为当 stream 的目的是交换大量数据时，synchronous stream 是不太优选的选择。不过，模拟这些条件的方法与 asynchronous streams 相同。

5.4 simulation的简化方法

如果没有兴趣测试 overflow 和 underflow 的响应，则有一种更简单的方法适用于 IP

core 的 simulation。例如，在host到FPGA方向： 当 read enable 信号为高电平时，只需从每个 rising clock edge 的文件中读取数据字即可在 test bench 中实现 FIFO。FIFO 的简化视图依赖于这样的假设：host 通过足够快地将数据写入相关 device file 来防止 FIFO 变空。

相反，当 write enable 信号为高电平时，test bench 将字写入文件。与之前类似，假设 host 始终通过足够快的读取数据来防止 FIFO 变满。

这种方法并没有忽视数据流连续性被破坏的可能性。相反，这种方法认识到数据流中断很可能是超出 simulation 范围的原因造成的： 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，这种方法忽略了 application logic 尝试超出 stream 带宽限制的可能性。如果这种情况确实存在，那么这种针对 simulation 的简化方法可能还不够。