(机器翻译成中文)

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

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 This document may also be slightly outdated in relation to the original.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

2 一般准则 5
2.1 Getting started . . . 5
2.2 block design中的显着元素 . . . 6

3 与 application logic集成 8
3.1 基础知识 . . . 8
3.2 Clocking . . . 9
3.2.1 一般的 . . . 9
3.2.2 设置 application clock . . . 9
3.2.3 bus_clk 信号 . . . 10

4 加速/协处理最佳实践 11
4.1 吞吐量与 latency . . . 11
4.2 数据宽度和性能 . . . 11
4.3 该做什么和不该做什么 . . . 12

5 应用自定义 Xillybus IP core 13

6 Vivado HLS 集成 15
6.1 概述 . . . 15
6.2 HLS synthesis . . . 15
6.3 与 FPGA 项目集成 . . . 16
6.4 示例 synthesis 代码 . . . 17
6.5 对 synthesis的 C/C++ 代码的修改 . . . 20
6.6 simple.c: host 程序示例 . . . 22
6.7 practical.c: 一个实用的 host 程序 . . . 24
6.8 Design 注意事项 . . . 29
6.8.1 使用多个 AXI streams . . . 29
6.8.2 application clock的频率 . . . 30
6.8.3 重置 logic . . . 31

1 介绍

Xillybus Block Design Flow 是 Verilog / VHDL design flow的替代品，适用于那些不习惯使用 logic相关的 HDL 语言进行修改和设计的人。其主要目的是让没有 FPGA 背景的设计人员无需获得 FPGA相关技能即可访问 coprocessing / acceleration 功能。其中，它旨在作为一种简单的方式在 Xilinx的 Vivado High Level Synthesis (HLS) 生成的 logic 与运行 Linux 或 Microsoft Windows的计算机或 embedded 平台之间交换数据。

Block Design Flow 偏离了 Xillybus通过 FPGA FIFOs与 Xillybus IP core 通信的主要概念。相反，用户应用程序 logic 通过 AXI Stream 接口直接连接到 Xillybus IP block 。这大大简化了工作，但需要注意区别，特别是当 Xillybus的文档在 FPGA中提到 FIFOs 时，这与 Block Design Flow无关：不是每个 FIFO，而是 Block Design的GUI中有简单的电线。

Xillybus的 Block Design Flow 不应与用于设置 Zynq processor 环境或以其他方式连接 logic blocks的 block design 图表相混淆：如果应用此类 block designs，则它们是不相关的，并且无论选择用于连接 Xillybus的 IP core 与 application logic的方法如何，都可以共存。

Xillybus 允许设计人员通过以下方式专注于与应用程序相关的生产性工作：

- 提供一个工作启动项目，该项目已准备好用于 compilation 到 FPGA bitstream 中。本项目借助 Xillybus的 IP core，在 FPGA 和电脑 host 之间建立了一个简单直观的数据交换，
- 提供一个示例 High Level Synthesis (HLS) 项目，用于演示 logic design 和 C/C++，以及本指南中解释的关键元素（参见 6部分），
- 允许使用 Vivado的 block design 工具非常简单地将 IP blocks 集成到 FPGA design
- 为 Linux 和 Windows 提供 drivers ，在 host上提供简单的编程接口，

- 提供一个 Web 工具，该工具自动创建由 data streams 组成的自定义 Xillybus IP cores ，这些 data streams 是专门为给定项目配置的。

由于 Block Design Flow 依赖于 Xilinx的 Vivado的 block design 工具，因此仅限于该工具涵盖的 FPGAs 。因此仅支持 Xilinx的 series-7 FPGAs 及更高版本（包括 Ultrascale 设备）。

尽管 Block Design Flow易于使用，但它只能访问 Xillybus的一部分功能，因此不建议那些熟悉基于 Verilog 或 VHDL的 FPGA design 的人使用。但是对于某些应用程序，例如 IP core 或基于 HLS的 hardware acceleration / coprocessing， Xillybus功能差异的影响可以忽略不计。

XillyUSB不支持 Block Design Flow 。

2 一般准则

2.1 Getting started

原则上，使用 Vivado为 Block Design Flow 设置项目如针对预期平台的相应 Getting Started 指南中所述：

- 对于 Xillinux 捆绑包： Getting started with Xillinux for Zynq-7000
- 对于 PCIe 捆绑包： Getting started with the FPGA demo bundle for Xilinx

遵循这些指南时，请务必使用 **blockdesign**/ 子目录中的 xillydemo-vivado.tcl script 。

重要的:

本指南不与 *Xillybus*网站上名为 *"FPGA coprocessing for C/C++ programmers"* 的教程一起使用。技术细节以及所展示的示例项目存在一些差异。为了避免混淆，建议您遵循本指南（适用于 *Block Design Flow*）或网站教程（适用于 *Verilog / VHDL design*）。

bitfile 的生成和使用与上述 Getting Started 指南中的操作相同： bitfile 可以从 "out of the box"捆绑包中立即生成，并且这些指南中描述的 loopback 测试的工作方式相同。但是请注意， seekable stream xillybus_mem_8 在 Block Design Flow中不起作用，如 4.3部分所述。

Block Design Flow 与 Xillybus的 IP core 的接口是在 Vivado的 block design 工具中进行的： 生成工程后，在 Vivado的左侧菜单栏中选择 "Open Block Design" 打开 block design 。

在基于 PCIe 的 designs 上（即不是 Xillinux），显示如下图：

使用 Xillinux 时，“Open Block Design” 打开 Zynq processor的环境—— block design 这部分不应该在与 *Xillybus*相关的任务中修改。相反，应该打开标记为 “blockdesign” 的 block （双击），显示以下 block diagram：

2.2 block design中的显着元素

Xillybus block design有几个关键要素值得注意：

- Xillybus stream 端口（“from_host_*” 和 “to_host_*”）：这些是标准 AXI Stream 端口，由最小的信号集组成：TDATA、TVALID 和 TREADY。block design 中每个端口的名称都以 “from_host” 或 “to_host” 开头，以标记接口的方向。block design 中端口名称的其余部分是 device file的名称，如 host所示，减去 “xillybus” 前缀。

  例如，在 Linux host上名为 `/dev/xillybus_write_32` 的 device file 或 Windows 计算机上的 `\\.\xillybus_write_32` 可以在名为 from_host_write_32的端口上的 block design 上访问。

- Loopbacks：最初，from_host_write_32 连接到 to_host_read_32，from_host_write_8 连接到 to_host_read_8。这会将写入名为 xillybus_write_32 的 device file 的所有数据循环回 xillybus_read_32。write_8 / read_8 对也是如此。loopback 是 Getting Started 指南中描述的 “Hello world” 测试起作用的原因。

  为了与 application logic集成，应将相应的 loopback 连接与 Vivado的 block design GUI移除，并应与 application logic的合适的 AXI Stream 端口进行连接。

- 在某些情况下，名为 xillybus_smb 和 xillybus_audio 的 streams 连接到上面的层次结构，因为它们用于支持板的音频接口。这些 streams 应该被忽略（即被视为进入 block design中的 processor design 层次结构的其余信号）。

- 每个 Xillybus stream的“*_open” 端口：每个 AXI Stream 端口都有一个对应的端口，其后缀为 _open，在 host上打开相关 Xillybus device file 时为高电平（’1’）。该信号可以选择性地用于重置连接到 stream的任何 application logic，因此每次打开 device file 时它都处于已知状态。

- Clocking Wizard (stream_clk_gen) block：为 application logic生成一个 clock，它基于来自 Xillybus接口的 clock。Xillybus IP core的所有 AXI Stream ports 都与此 block的输出同步。

  建议不要对这个 block 做任何改动，除了 output clock的频率。特别是，block 的名称必须保留（stream_clk_gen），因为与 design 的 implementation 相关的某些 scripts （timing constraints）通过其名称引用此 block的输出。

  请参阅下面的 3.2 部分。

- 外部端口，例如 GPIO_LEDS[0:3]：连接到 block design之上的层次结构的端口。这些连接不应更改，但它们的信号可能会被其中的块采样。例如，在 Xillinux 包中，ap_clk 进入上层，但也可以在 block design 视图中使用。

请注意，mem_8 stream没有端口。block diagram中没有 Seekable streams。有关这方面的更多信息，请参阅 4.3 部分。

3 与 application logic集成

3.1 基础知识

与 application logic 的集成是通过使用 Vivado的 block design GUI完成的： IP blocks 添加到 block design 并根据需要进行连接。

Vivado's High Level Synthesis (HLS)生成的 IP blocks 的集成请参考 6章节。

需要注意的是，虽然在 Xillybus的文档中经常说 FPGA 中的 application logic 通过 FIFOs与 host 通信，但 Block Design Flow 并非如此（但仅适用于 Verilog / VHDL design flow）。生成 AXI Stream 接口的 Xillybus的 IP Core 中的 glue logic 已经包括 FIFOs （其中 clock domain crossing 在 bus_clk 和 ap_clk之间）。因此，与 VHDL / Verilog design flow不同，在使用 Block Design Flow 时， application logic 不需要部署任何 FIFOs 即可与 Xillybus的 IP core 接口。

对于 FPGA 和 host之间的数据交换，将 application logic 连接到专用的 AXI Stream 端口（可能在断开 loopbacks之后）。这些端口只提供 TDATA、 TVALID 和 TREADY，尤其是 TLAST 信号。因此，每个 AXI Stream stream 都包含一个 infinite data stream（与 TLAST 信号允许的数据包接口相反）。这与 Xillybus的 device files 的 infinite stream 本质是一致的。

Xillybus streams 可用于在 FPGA 和 host之间交换数据包，如以下任何指南中的第 6.3 节所述：

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

3.2 Clocking

3.2.1 一般的

为简单起见，连接 user application logic 和 Xillybus IP Core 的所有信号都必须由单个 clock驱动，该信号由 block design中的 Clocking Wizard block 产生。这个 clock（“application clock”）是 Clocking Wizard的 clk_out1，也是 Xillybus IP Core block的 ap_clk 输入。

用这个单一的 clock驱动整个 user application block 通常很方便，因此它的所有内部 logic 以及接口都依赖于它。例如，由 Vivado HLS' synthesizer 生成的 logic 有一个 clock input （命名为 ap_clk）。将此 clock input 连接到 Clocking Wizard的输出可以保证 AXI Stream 端口与 Xillybus IP Core的 block 的连接正常工作。

请注意， FPGA 工具有时会根据频率本身来指代 clock的频率，通常在 MHz中，有时称为 clock period，通常在 ns中。 clock的频率是 clock period的 reciprocal ，例如 100 MHz 相当于 10 ns的 clock period 。

3.2.2 设置 application clock

application clock的频率可以设置为提高性能或作为实现 bitfile工作的一个步骤：更快的 clock 产生更高的处理吞吐量（除非某些其他瓶颈限制了性能），但也需要更多 FPGA的 logic 元素及其对 Xilinx的利用率工具。

如果 application clock的频率选得太高，项目中的 compilation 会以不符合 timing constraints为由，导入 FPGA bitstream 文件失败。这也称为 “timing failure”。这种情况意味着执行 implementation 的工具未能以确保可靠运行的方式利用 logic ，而 logic 是由 clock定义的频率驱动的。本文中的 “timing constraints” 是对系统中 clocks 频率的要求。

始终允许降低 application clock的频率（在 clock generator的限制范围内），但会减慢其驱动的 logic 的运行速度。

要设置 application clock的频率，请在 block design 视图中双击 Clock Wizard (stream_clk_gen) 的 block 。在 Vivado中将打开一个配置窗口。选择 “Output Clocks” 选项卡并更改 clk_out1的 “Output Clock Requested” 频率。 “Actual” 列中的频率显示了 clock synthesizer将产生的频率。它可能与请求的频率略有不同，因为 output clock 是通过将 input clock 乘以一个有理数得出的，该有理数是从一组有限的允许值中挑选出来的。

当 clock 仅用于 application logic 及其与 Xillybus IP core的接口时，对所请求频率的微小转移是无害的。

Clocking Wizard 中的其他参数不应更改。

3.2.3 bus_clk 信号

Xillybus IP Core的内部 logic 是由 bus_clk驱动的，在 block design 中暴露出来只是为了允许从 bus_clk派生 application clock 。该信号通常没有其他用途，因为 application logic 只需要 ap_clk 用于其内部 logic 和与 Xillybus IP Core的接口。

然而，为了发现吞吐量瓶颈，bus_clk的频率可能是有意义的。例如，如果 bus_clk 以 100 MHz运行，则可以通过 32 位宽数据接口的最大理论带宽是 400 MB/s，因为 Xillybus的内部 data pipe 以 bus_clk的速率运行。如果 ap_clk 以更高的频率运行并且在 ap_clk的每个周期都推送数据，则数据速度很可能会由于 AXI Stream 流控制信号（TREADY 和 TVALID）而减慢。

出于这个原因，在尝试最大化应用程序的吞吐量时，应考虑 bus_clk的频率，特别是在数据接口预计包含长突发（或连续）数据流量的情况下。

bus_clk 的频率可以在 “Clocking Options” 选项卡下找到，作为 primary input clock的频率，也就是 clk_in1。此参数告知 Clocking Wizard 在其输入端预期的频率，因此可用于了解特定 Xillybus 捆绑包的 bus_clk 频率。

4 加速/协处理最佳实践

4.1 吞吐量与 latency

基于增强指令集（例如 x86 系列的 MMX 命令、 AES的加密扩展和 ARM的 NEON 扩展）的传统硬件加速与使用外部硬件（如 GPGPU 和 FPGA）的加速之间存在显着差异。由于增强型 instruction sets 是 processor执行流程的一部分，它们用较短的指令替换了较长的机器代码指令序列，并减少了在结果可用之前所需的周期数。

另一方面，外部硬件加速（包括FPGA 加速）并不一定会减少直到结果可用的时间，因为 latency 向外部硬件传输数据和从外部硬件传输数据的时间很重要。此外，由于pipelining的原因，处理时间也可能比 processor的要长得多，而且可能其 clock的频率更低。

因此，外部硬件加速的优势不是 latency （获得结果的速度）而是吞吐量（处理数据的速率）。为了利用这一优势，重要的是保持进出加速硬件的数据流，而不是在启动下一个操作之前等待一个操作的结果。

使用 FPGA 进行适当加速的技术在这两个文档的第 6.6 节中进行了详细说明：

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

4.2 数据宽度和性能

对于需要相对较高数据带宽的应用程序，建议对数据密集型 streams使用 32 位宽的 streams （或更宽），因为 8 位和 16 位宽的 streams 对 host的数据 bus 的利用效率较低。

原因是字以 bus的速率通过 Xillybus 内部数据路径传输。因此，传输 8 位字与 32 位

字占用相同的时隙，从而使其实际上慢了四倍。

这也会影响其他 streams 在给定时间竞争底层传输，因为数据路径被较慢的数据元素占用。

该指南不适用于 B/XL/XXL Xillybus IP cores版本，它以相同的效率传输窄 streams 。

4.3 该做什么和不该做什么

使用 Block Design Flow时需要注意几个问题：

- Block Design Flow不支持带有地址端口（“address/data streams”、“seekable streams”）的Streams 。如果 Xillybus IP Core 包含这样的 streams，它们在GUI中不会显示为端口，但在 host 端会正常显示。尝试从 host 上的此类 stream 读取将立即产生 end-of-file 条件。 write() 函数调用不会返回，因为另一端没有 data sink 。

  因此，建议在与 Block Design Flow一起使用的定制 IP cores 中避免使用 seekable streams ，以避免混淆和稍微浪费 FPGA logic。

- 请勿更改名为 “stream_clk_gen” （Clocking Wizard）的 block ，除非必要时更改其输出频率，如 3.2.2部分所述。

  对 input clock的频率进行更改，对配置进行其他更改，或者将其从 design 中移除并更换为新的 Clocking Wizard IP block 可能会导致无法满足 timing constraints （可能是因为某些 timing constraints 例外引用了 block的名称）。

  设置错误的输入频率可能会导致 FPGA design的行为不可靠。

- 注意 clocks 的连接方式很重要。特别是不要在 bus_clk 和 ap_clk之间混用。

- 确保 Xillybus streams 是异步的，当 stream的预期用途是 “Data exchange with coprocessor”时，默认 IP core 和自定义 IP cores 中的 autoset 选项就是这种情况。

  这会导致在 host 上进行的 write() 函数调用在 driver的 DMA buffers中有足够的数据空间时立即返回，从而确保更顺畅的数据传输和更高的带宽性能。

  为了更好地理解这个主题，请参阅 Xillybus host application programming guide for Linux 或 Xillybus host application programming guide for Windows的第 2 节。

5 应用自定义 Xillybus IP core

Web 应用程序允许用户配置和下载自定义 Xillybus IP cores，直接在 Xillybus的网站上选择 streams 的数量及其属性。然后从站点下载专门生成的自定义 IP core ，通常在几分钟后。

要生成和下载自定义 IP core，请访问 Xillybus网站上的 IP Core Factory 。该过程相当简单，如有必要， The guide to defining a custom Xillybus IP core 会提供免费信息。

> **重要的:**
>
> *Seekable streams* （带 *"address/data"* 接口）在 *Block Design Flow*中是不可见的，因为 *AXI Stream* 连接不支持地址线。在 *core* 中拥有这样的 *streams* 是相当无害的，但会导致 *FPGA logic* 资源的轻微浪费，并且可能会造成混淆，因为它们确实出现在 *host* 端，但不会出现在 *block design*中。

定义自定义 IP core 后，生成并下载其捆绑包。

自定义 IP core 捆绑包的 README 文件中的说明与 Verilog / VHDL design flow 相关，应忽略。相反，应采取以下步骤：

- 为自定义 IP core的文件创建一个新目录。这个目录的 absolute path 在使用这个自定义 IP core的过程中必须保持固定，所以建议放在不会被误删的地方。

  将下载的自定义 IP core 捆绑包解压缩到此目录中。

- 在 Vivado中打开 block design 。

- 将 block design的视图另存为 pdf 文件以供参考：右键单击 block design区域的某处，然后选择 "Save as pdf file..."。

- 在 Tools 菜单下选择 “Run Tcl Script...” （在主菜单栏上） 。导航到自定义 IP core 捆绑包解压缩到的目录，然后输入 xillybus_block 子目录。选择 insertcore.tcl。
- script 将用定制的 IP core替换现有的 Xillybus IP Core ，并尝试重新连接与应用无关的接线。由于自动重组，对象也可能在 block design 图表中移动。
- 将 application logic的 AXI Stream 接口连接到更新后的 Xillybus IP core。
- 与运行 script之前创建的 pdf 文件进行比较，并根据需要进行更正。

  application logic 相关连接均未重新连接，其他连接也可能丢失。
- 验证 Xillybus IP Core的 block 下方和下方的标题是否与新 IP core的名称匹配。

请注意， script 通过名称查找 blocks、端口和接口。因此，如果用户更改了这些名称，它可能会在恢复连接时部分（并且静默）失败。

至此，项目的 implementation 就可以和以前一样了。 Xillybus的 driver 用于 host（Linux 和 Windows 类似）也适用于自定义 IP core ，因为它会自动检测新 IP core的配置。

因此，在用定制的 IP core替换后，无需在 host 上安装任何东西。

作为参考，这些是 insertcore.tcl script的执行步骤：

- 将自定义 IP core 的目录添加到 Vivado的 IP Catalog 中的 IP Core repositories 列表中并强制重新扫描 repositories，这样就发现了新的自定义 IP Core 并添加到 Catalog 中
- 从 block design中删除以前的 Xillybus IP （如果存在）
- 将自定义 IP core 添加到 design，并根据需要升级其版本
- 通过查找名称列表并互连具有这些名称的所有端口（如果存在），尝试重新连接通往上述层次结构的电线以及通往 stream_clk_gen的 block的电线。
- 仅在 Zynq 上：将 Xillybus IP core 的 bus 地址设置为其默认值（从 `0x50000000`开始的 4 kB 段）
- 重置项目的 synthesis 运行，以便下一个 implementation 反映所做的更改。

6

Vivado HLS 集成

6.1 概述

本节演示将简单 C function 的 compilation 转换为 IP block，以及如何将其集成到 Xillybus的 Block Design flow中。

本节所基于的示例项目可在以下网址下载

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

建议将下载的文件解压到与 Xillybus 项目容易关联的目录中，后期无法移动。

在示例项目中区分两种不同类型的 C 源很重要：

- 执行代码：与任何计算机程序一样，在计算机或 embedded 平台 (“host”) 上运行，并使用 FPGA 卸载某些操作。

  在示例项目中，可以在 host/ 子目录下找到示例文件。

- synthesis的代码：旨在由 Vivado HLS翻译成 logic 。

  在示例项目中，可以在 coprocess/example/src/main.c 找到

与常见的 C/C++ 编程不同， host 程序不调用 synthesized function。相反，它将执行功能所需的数据组织在一个数据结构中，并使用一个简单的 API将其传输到 synthesized function，这将进一步描述。在稍后阶段，它将返回数据收集为从 synthesized function 与类似 API发送的数据结构。

6.2 HLS synthesis

本节中使用的 C 中的示例代码在 6.4节中进行了概述。

启动 Vivado HLS，然后打开 HLS 项目：在欢迎页面上选择 “Open Project”，导航到 HLS 项目包解压缩到的位置，然后选择名称为 “coprocess”的文件夹。

更改项目的部件号：选择 Solution >Solution Settings... >Synthesis 并将 “Part Selection” 更改为预期的 FPGA。

通过选择 Solution >Synthesis >Active Solution （或单击工具栏上的相应图标）启动项目的 compilation (“synthesize”)。 console上会出现很多文字，包括几个 warnings （这是正常的）。不应出现错误。

HLS的 console 选项卡最后几行中的以下消息很容易识别成功的 compilation：

```
Finished C synthesis.
```

只有当 synthesis 成功时， synthesis 报告才会出现在 console 选项卡上方。

有关 Vivado HLS的更多信息，请参阅其用户指南（UG902）。

6.3 与 FPGA 项目集成

在 Vivado HLS中，选择 Solution >Export RTL 并选择 “IP Catalog” 作为 Format Selection。对于 “Evaluate Generated RTL”，请选择 Verilog，并且不要选中此选项下的任何一个复选框。单击 OK。

这可能需要几分钟，并以类似

```
Finished export RTL.
```

现在在 Vivado 中（即不在 Vivado HLS中）打开 Xillydemo 项目（在 2.1部分中设置），然后打开 Block Design。使用 Xillinux (Zynq) 时，打开名为 “blockdesign”的 block 。

按如下方式添加 HLS IP block： 右键单击 block design 图表区域中的某处，然后选择 “IP Settings...”。在 “Repository Manager” 选项卡下，单击绿色加号以添加 repository。导航到并选择在 6.2 部分中选择的相同 “coprocess” 目录以打开 HLS 项目。 Vivado 应以弹出窗口响应，指示添加了一个 repository 。单击 “OK” 按钮两次以确认。

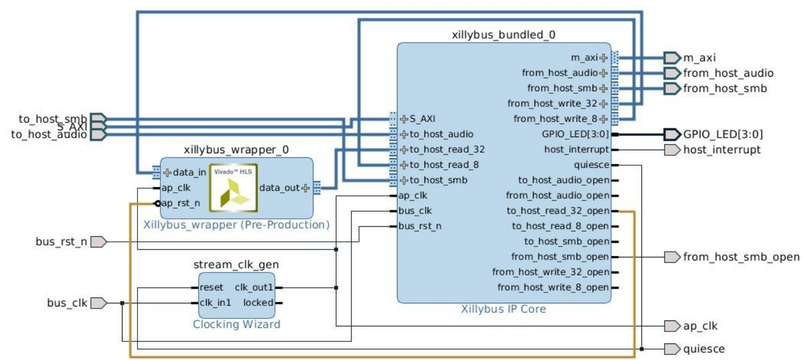
现在将 IP block 添加到 block design： 再次右键单击 block design 图表区域中的某处。选择 “Add IP...” 并从列表中选择 Xillybus_wrapper IP （在搜索框中键入 “wrapper” 可能会更容易）。

图中将出现一个名为 xillybus_wrapper_0的新 block。断开 to_host_read_32 和 from_host_write_32 之间的电线（即断开 loopback）。

然后按如下方式连接 xillybus_wrapper block：

- data_in 与 from_host_write_32
- data_out 与 to_host_read_32
- ap_rst_n 与 to_host_read_32_open
- ap_clk 和 ap_clk （也是 Clocking Wizard的 clk_out1 输出）

结果应该是这样的（显示为基于 Xillinux的 block design）：



ap_rst_n 和 to_host_read_32_open 之间的连接使 logic 在 xillybus_wrapper的 block 内部保持复位状态，除非 xillybus_read_32 device file 在 host 上打开（未打开文件时to_host_read_32_open 为低电平，复位输入为低电平有效）。假设在 host 上运行的软件在尝试与此 block通信之前打开此 device file ，这可确保每次运行软件时 logic 的响应一致。

此时可以执行 implementation 获取 bitstream：在 Vivado的窗口底部，选择 “Design Runs” 选项卡，在 “synth_1” 上右击选择 “Reset Runs”。确认重置 synth_1。

然后单击左侧栏的 “Generate Bitstream” 。

6.4 示例 synthesis 代码

为了阐明 HLS 如何与 Xillybus一起工作，该示例演示了三角正弦的计算和 integer的简单运算，两者都包含在一个简单的自定义函数 mycalc()中。

coprocess/example/src/main.c 启动如下：

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

extern float sinf(float);

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

像往常一样，有几个 #include statements。 “math.h” 包含对于正弦函数是必需的。

还有一个简单的功能， mycalc() ， 它扮演了 “synthesized function”的角色。这是一个非常简单的函数，用于演示 floating point 和 integer的算术运算。 High-Level Synthesis Guide UG902 提供了有关如何执行更有用任务的更多信息。

接下来在 main.c中，有包装函数 xillybus_wrapper()，它是 synthesized function 和 Xillybus之间的桥梁，因此负责来回打包和解包数据。

在示例的情况下，它通过 data stream（由 “data_in” 参数表示）从 host 接受 integer 和 floating point 格式的数字。它使用 “data_out” 参数返回 integer 加一和 floating point 数的（三角）正弦值。

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

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

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

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

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

xillybus_wrapper() 用两个 pointers声明，两个 int类型的变量。这些函数参数变成了未来 IP block 的两个 AXI Stream 端口以包含在 block design中：它们每个都有一个 #pragma 语句，通知 HLS 它们应该被视为 “axis”类型的接口。

“#pragma AP” 和 “#pragma HLS” 是可以互换的——前者是基于 C Synthesizer以前的名称 (Auto Pilot)，而后者在 Xilinx的最新文档中可见。

由于 “int” 被 HLS视为 32 位字，因此各个 AXI Stream 接口将具有 32 位宽的数据接口。

当然可以更改参数列表以及 pragmas 以获得任何一组 AXI Stream 输入和输出。

ap_ctrl_none 的 pragma 声明告诉 compiler 不要为（不存在的） return value生成端口。

接下来是 “execution”的一些代码： 获取输入数据。每个 *data_in++ 操作都会获取一个来自 host的 32 位字。在所示代码中，第一个字被解释为 unsigned integer，并放入 x1。第二个字被视为 32 位 float，并存储在 x2中。

然后是对 mycalc()的函数调用，即 “synthesized function”。这个函数返回一个结果作为它的返回值，第二个数据通过改变 x2返回。

包装函数将 x2 的更新值复制到新变量 y2中。如果此代码的 compilation 旨在在processor上执行，这可能看起来是一个冗余操作。然而，当使用 HLS时，这对于让compiler 处理稍后转换为 float 是必要的。这反映了 HLS compiler的一些古怪行为，但这是使用 pointer的微妙问题之一：即使在 C 代码中定义了内存阵列和 pointer，HLS compiler 也不会生成其中任何一个。 pointer 的使用只是对我们想要完成的任务的一个提示，有时这些提示需要稍微推动一下。

最后，将结果发送回 host：每个 *data_out++ 向计算机发送一个 32 位字，并从float进行适当的转换。

请注意， *data_in++ 和 *data_out++ 运算符并没有真正移动 pointers，并且没有底层内存数组。相反，这些符号表示将数据从 AXI stream 接口移动到 AXI stream 接口（最终从 Xillybus streams移动到 Xillybus streams）。因此，使用 “data_in” 和 “data_out” 变量的唯一方法是 *data_in++ 和 *data_out++ （High-Level Synthesis Guide 提供其他可能性，特别是固定大小的数组）。

另请注意，由于此代码被翻译成 logic，而不是由 processor运行，因此这些 C 命令的唯一意义是在给定数据输入 stream 的情况下产生预期的数据输出 stream 。然而，没有承诺何时发出数据（ HLS报告中给出的一系列可能的 latencies除外）。

因此，输入数据的分配顺序很重要，因为它强制了输入数据的解释方式。另一方面，由于发送的第一个输出 y1仅取决于到达的第一个输入 x1，因此允许在第二个输入到达之前发送第一个输出。这与代码执行的直观顺序性质相矛盾，但在硬件加速的上下文中毫无意义，因为总体结果是相同的。

此外，如果 data_in AXI stream 不断地输入数据，则包装函数 “runs” 会重复，就好像它在说：

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

通过 *data_in++ 命令尽快获取新数据，很可能填充 logic的内部 pipeline （根据HLS的报告，在示例项目中超过 70 个阶段）。因此，与 processor的代码执行不同，processor会获取一对词，处理它们，发出两个输出词，然后才获取第二对词， HLS 解释可能很好地在 data_in 获取 70 个词，然后才会出现任何内容在 data_out AXI stream上。

6.5 对 synthesis的 C/C++ 代码的修改

可以通过向包装函数添加参数并将它们声明为接口端口来创建其他 AXI Stream 端口，如示例中所示。

当然可以对示例 design的 C 代码进行其他更改。

建议以与 *data_in++ 和 *data_out++相同的样式实现 I/O ，或参考 High-Level Synthesis Guide (UG902) 了解其他可能性。它也是学习编码技术的推荐资源。

重要的:

进行更改后不要只在 *Vivado* 中单击 *"Generate Bitstream"*： 重复启动 *bitstream* 的 *implementation* 而不升级 *block*， 如下所述，可能会导致 *bitfile*的 *implementation* 看似成功，但基于 *HLS block*的过时版本。

在示例项目中进行更改后，从 6.2部分中的 "HLS synthesis" 重新开始，并使用 Vivado一直到 implementation ，并在 Vivado中更新 HLS block 。

换句话说：

- Vivado HLS：在 HLS中运行项目的 compilation 。 HLS synthesizer 总是在开始新的文件之前清理由以前的 compilations生成的文件。
- Vivado HLS：导出到 IP Catalog bundle。
- 在 Vivado （不是 Vivado HLS）中，升级 xillybus_wrapper 的块（实际上是根据它的变化更新）：打开 block design view，回复页面顶部的消息，说 block 需要升级。如果未找到此消息，请在 Tcl Console中键入 "report_ip_status -name status" 。单击底部的 "Upgrade Selected" 按钮。随后会出现一个确认升级成功的对话框，以及一个请求生成输出产品的对话框。在第二个对话框中单击 "Skip" 。
- Vivado：验证 design runs 是否无效：在 Vivado窗口的底部，选择 "Design Runs" 选项卡。 synth_1的 Status 列中应该显示 Synthesis Out-of-date 。
- Vivado：**除非 design runs 无效，否则**尝试以下操作： 刷新 IP 目录：右键单击 block design 图表区域中的某处，然后选择 "IP Settings..."。在 "Repository Manager" 选项卡下，单击底部的 "Refresh All" 按钮。可能还需要单击同一对话框的 "General" 选项卡上的 "Clear Cache" 。在此之后，返回升级 xillybus_wrapper的 block 。

  如果在上一项中发现 *design runs* 无效，则无需执行任何这些操作。
- Vivado：重置 synth_1 运行
- Vivado：生成 bitstream

6.6 simple.c： host 程序示例

在示例项目中，有两个 C 文件的示例 host 程序： simple.c 和 practical.c。这些展示了项目的 host 方面。

两者都是为 Linux host编写的，例如 compilation

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

然而，它们很容易适应 Windows （见下文） 。

重要的:

simple.c 不应**作为实际 *host* 编程的示例**，特别是由于其以下缺点：

- 只处理一个元素。在 *write()* 和 *read()* 对函数调用上循环会导致性能下降。
- 必须检查 *write()* 和 *read()* 操作的返回值是否正确操作。为简单起见，这已被省略，但会使程序不可靠。

第 *6.7* 节概述了更好的编码技术。

simple.c 文件以 #include statements开头：

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

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

接下来是 main() 函数的经典声明，以及一些变量的声明：

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

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

struct 变量将在下面讨论。

程序首先打开两个 device files，其行为类似于 named pipes，用于与 logic通信：/dev/xillybus_read_32 和 /dev/xillybus_write_32。回想一下 Xillybus 捆绑包的设置，这两个文件是由 Xillybus的 driver生成的。

正如 6.3部分所指出的， ap_rst_n 连接到 block design 图中的 to_host_read_32_open ，因此打开 /dev/xillybus_read_32 会使 logic 退出复位。这就是为什么在数据传输之前打开这两个文件的原因。

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

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

接下来，到实际执行。 “tologic” 结构填充了几个值以传输到 logic，然后直接从内存写入 xillybus_write_32。实际上，这会写入 8 个字节，或更准确地说，是两个 32 位字。第一个是 tologic.v1中的整数 123 ，第二个是 tologic.v2中的 float 。因此设置了 tologic 结构以匹配 logic 对数据的期望：第一个 *data_in++ 指令一个 integer ，第二个 float 。

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

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

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

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

回想一下 6.4 部分，包装代码从 data_in stream获取两个 32 位字。第一个字转到 “x1”，第二个字转到 “tmp”，然后 “tmp” 立即转换为 float。这与 “tologic” 结构的两个 32 位元素匹配。

然后从 FPGA读回数据。同样的原理也适用于 “fromlogic”。

simple.c 以一个共同的总结结束：

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

  return 0;
}
```

将发送到 /dev/xillybus_write_32 的数据量与包装函数中 *data_in++ 操作的数量相匹配是至关重要的。如果发送的数据太少， synthesized function 可能根本无法执行。如果太多，下面的执行可能会出错。

在本例中， “tologic” 和 “fromlogic”选择了相同的结构格式，但不必拘泥于此。重要的是发送和接收的数据与包装函数的 *data_in++ 和 *data_out++ 操作数同步。

这个程序的执行应该是

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

最后，对 Windows 用户的说明，他们可能需要进行以下全部或部分调整：

- 将文件名字符串从 “/dev/xillybus_read_32” 更改为 “\\\\.\\xillybus_read_32” （Windows 上的实际文件名是 \\.\xillybus_read_32，但 escaping 是必须的）。第二个文件名更改为 “\\\\.\\xillybus_write_32”。
- 将 unistd.h 的 #include 语句替换为 io.h
- 用 _open()、 _read()、 _write() 和 _close()替换对 open()、 read()、 write() 和 close() 的函数调用

6.7 practical.c：一个实用的 host 程序

simple.c 示例以简洁的方式概述了数据交换，但在实际系统中需要进行一些更改：

以下差异最为显着：

- 分配和发送一个结构数组，而不是生成单个数据集进行处理。同样，从 logic接收到一组数据。这减少了 I/O overhead 以及 latencies的影响，这是由软件和硬件造成的。这是通过硬件加速获得性能改进的关键方法。
- 该程序分为两个进程，一个用于写入，一个用于读取数据。使这两个任务独立可以防止处理由于缺乏数据来处理而停止。这种独立性可以通过 threads （特别是在 Windows中）或使用 select() 函数调用来实现。

- read() 和 write() 函数调用正确，保证 I/O可靠。为此目的添加的 while循环可能看起来很麻烦，但它们对于正确响应这些函数调用的部分完成（不是所有字节读取或写入）是必要的，这是负载下的常见情况。 EINTR 错误也会根据需要进行处理，以便对 POSIX signals做出正确反应，这可能会意外发送到正在运行的进程。

现在来简要介绍一下 practical.c。一、 headers：

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

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

和相同的结构，加上定义 N，每块数据的元素数。

```
#define N 1000

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

一个常见的 main() 函数定义和一些变量：

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

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

像以前一样打开的文件：

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

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

实际执行从 fork() 开始分为两个进程。

```
pid = fork();

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

父进程准备要处理的数据并将其写入 FPGA。它会关闭 read file descriptor，因为此进程不使用它。保持打开状态将使 device file 保持打开状态，直到两个进程都关闭了它们的 file descriptor （或退出），这不是这里想要的行为。

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

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

接下来，用数据填充结构数组。这就解释了为什么为每组数据定义一个结构以进行处理是有意义的。

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

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

  buf = (char *) tologic;
```

请注意， “buf” 定义为 char的 pointer 到 buffer ，指向结构数组。这种转换是必需的，因为发送数据的 while循环将 buffer 视为任何要传输的数据块。

接下来， while循环用于写入数据。它可能看起来不必要地复杂，但却是确保数据可靠写入的最短方法。建议在实际应用中采用此代码。

```
donebytes = 0;

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

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

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

  donebytes += rc;
}
```

在此示例中，仅发送一个块（并在另一端接收）。在实际代码中，循环上面两段代码是正确的。

性能测试表明，32 kBytes 的块大小通常会给出最好的结果。

由于在此示例中只发送了一个块，因此进程退出。在关闭文件之前的一秒钟内休眠可确保 logic 在所有数据都被耗尽之前不会重置。当 block design 如 6.3部分所示时，这是没有意义的，因为 ap_rst_n 转到 to_host_read_32_open，而 from_host_write_32_open 根本没有连接。

尽管如此，这表明了一个很好的惯例，即不立即关闭 file descriptor ，除非需要快速退出。当项目变得更加复杂时，这可以避免一些混乱。

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

close(fdw);
return 0;
```

接下来我们有子进程，以类似的方式开始：

```
  } else {
    close(fdw);

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

    buf = (char *) fromlogic;
```

再一次，这是从 device file读取数据的推荐方法：

```
    donebytes = 0;

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

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

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

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

      donebytes += rc;
    }
```

然后打印出数据：

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

    sleep(1); // Let the output drain

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

再一次，进程在关闭 file descriptor之前休眠一秒钟，再一次，在这种特定情况下没有必要：关闭 file descriptor 确实会重置 logic，但在这种情况下它是无害的，因为所有输出都已获取，到了这一点。

如前所述，除非快速退出是有益的，否则这一秒的睡眠可以避免混乱，尤其是在生成其他输出 streams 时，例如用于调试。

6.8 Design 注意事项

6.8.1 使用多个 AXI streams

示例项目显示了每个方向一个 stream 的基本情况。然而，通过向包装函数添加参数来添加 streams 以在 IP block 上输入和/或输出是微不足道的，同时 pragmas 将这些参数声明为 AXI streams。

例如，三个输入 streams 而不是一个：

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

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

}
```

通过配置自定义 IP core，将 streams 添加到 Xillybus IP core 同样简单，如 5部分所述。

额外的 streams 可用于各种场景，其中包括：

- 在单独的 streams中发送数据和元信息。例如，如果需要将数据分成数据包，则将它们的长度发送到一个专用的 stream，将数据发送到另一个。这允许在知道其长度之前发送数据包的开头。
- 发送自然分开排列的数据，例如不同图像的像素扫描（更多内容见下文）。
- 用于调试：将中间数据发送到 host 进行验证。

在使用多个 streams时，务必牢记它们：如果任何输入 stream 缺少数据，或者如果输出 stream的相应 device file 未打开（或 overflow 带有数据）， logic的执行流程可能会停止。如果输出 stream 用于调试，这一点尤其重要：当使用系统进行正常操作时，很容易忘记用于调试的 stream 。因为来自这个 stream 的数据没有被消耗，这会导致执行的混乱停止，通常是在几个数据周期之后。

以乍一看似乎不合适的方式向 logic 硬件提供数据通常是明智的。例如，上面显示的三输入示例对于每个操作需要三个数据元素的图像处理算法很有用：假设从左到右、从上到下扫描图像。为了生成像素输出，该算法需要来自两个先前图像的相应像素以及当前图像的像素。在这种情况下，可以通过一个 stream 将当前图像发送到 FPGA，并通过另外两个 streams 将之前的两个图像并行发送。

这似乎浪费了 I/O 数据带宽和大量不必要的内存复制。尤其是 processor 在 “shuffling data”中涉及这么多，可能会让人感觉不对。撇开主观看法不谈，内存复制的实现在每个现代 processor 架构上都是一项高度优化的任务，而 processor 经常加载其他与应用程序相关的任务，这使得内存复制负载可以忽略不计。

因此，即使从资源利用率的角度来看，直接向 logic 提供数据并不是最理想的，但 processor 上的额外负载通常相当小，因为它通常需要处理其他繁重的任务。这通常是显着简化 design 的合理价格。

6.8.2 application clock的频率

HLS 产生的 logic 由 block design的 application clock 驱动， stream_clk_gen的 block 产生。由于这个 clock 是 logic的时基，它的执行率与 clock的频率成正比。除非 AXI stream 端口的数据传输成为瓶颈，否则更高的 application clock 频率意味着处理吞吐量的成比例加速。

然而， application clock的频率可以达到多高是有限制的，这取决于 FPGA 的 logic 资源以及如何利用它们来执行所需的任务。以下是 design 流程中的相关里程碑：

1. Vivado HLS 允许用户为 design设置 clock 的预期频率，指定 application clock 的所需频率（使用 Solution >Solution Settings）。 HLS 仅使用此参数作为提示，允许它在必要和可能时为生产更快的 logic 做出额外努力。

2. 当 Vivado HLS 完成其 compilation时，它会显示可能达到的 clock频率的估计（在 HLS的 GUI的 Synthesis 选项卡的 “Performance Estimates” 部分中的 “Timing” 下）。

3. 用户在 Vivado的 block design中设置 application clock 的频率，如 3.2 部分（特别是 3.2.2 部分）所述。自然的选择是 clock的频率，如第 2 项中估计的那样，或者更低。请注意，这是在 Vivado中完成的，而不是在 Vivado HLS中完成的。

4. 当 Vivado 将整个 design 的 implementation 完成为 FPGA的 bitstream 时，它会通知用户是否成功组织 logic 以满足与 clock相关的所有要求。这包括满足第 3 项中设置的 clock的频率。

所以归结为最后一个里程碑，如果 Vivado 能够满足与第 3 项中选择的 application clock 频率相关的 timing constraints 。

HLS 和 stream_clk_gen中的默认 clock period 是 10 ns (100 MHz)。通常最好保持这种选择，除非：

- Vivado 无法满足 timing constraints，在这种情况下应选择较慢的 clock 。
- 如果有增加处理吞吐量的动机，则应尝试要求更快的 clock 。这通常是调整 clock频率以及对 design 本身和 HLS pragmas 进行更改以获得改进结果的迭代过程。

6.8.3 重置 logic

当 C/C++ 代码被翻译成 logic时，它实际上并没有运行，而是保持了它自己的执行流程的状态。为了使 logic 模仿 processor执行程序的行为，确保从程序的开头开始执行是必不可少的。这是通过重置 logic来实现的。

在大多数情况下，直观的行为是当 host的程序开始执行时， FPGA 中的程序从它的开头开始。由于在 host 上运行的任何进程在访问它们之前都会打开 device files ，并且这些文件至少在进程终止时必须关闭，所以当一个或多个 device files 关闭时重置 logic 是很自然的。

Xillybus IP Core 中的每个 stream 都有一个 *_open 端口，当相应的 device file 打开时，该端口为高电平（’1’）。由于 HLS block 有一个低电平有效复位输入 ap_rst_n（默认情况下），将 *_open 输出直接连接到 ap_rst_n 输入会产生所需的结果：当文件关闭时， *_open 信号为低电平（’0’）。这将 logic 保持在复位状态。

可能需要组合多个 *_open 端口，以保持 logic 直到所有 device files 都打开，或直到其中任何一个打开。这是通过添加简单的 logic gate blocks来实现的，这些 logic gate

blocks在 Vivado的 IP catalog上可用。如何生成复位信号的选择取决于 host 程序的设置方式。

无论哪种方式，重要的是要确保 host 在按要求打开 device files 之前不会尝试与 HLS block 交换数据，以确保复位信号变为非活动状态。为简单起见，最好在开始与 HLS block 进行任何数据交换之前打开所有与 HLS block 相关的 device files ，然后将它们全部关闭以进行清理。