



Gisselquist  
Technology, LLC



主要/博客  
关于我们  
FPGA地狱  
教程  
正规训练  
测验  
项目  
网站索引  
@zipcpu  
红迪网  
I●支持

# 为 AXI 处理构建防滑缓冲器

2019 年 5 月 22 日

我目前正在构建和验证几个 [AXI](#) 内核，主要是为了证明我的 [AXI 正式验证 IP](#) 内核有效。这些内核的一些示例包括：

- [Xilinx 的演示内核](#)
- [我自己的 AXI 从内核](#)
- [我自己的 AXI 横杆](#)
- [总线故障隔离器](#)
- [一个 WB 到 AXI 转换器，以及](#)  
AXI 到 WB 转换器。后一个实际上是两个转换器，AXI 写入到 WB 桥和 AXI 读取到 WB 桥，就像 [AXI-lite 到 WB 转换器](#) 也分为 [AXI-lite \(写入\) 到 WB 转换器](#) 和 [AXI-lite \(读\) 到 WB 转换器](#)。然后，这两者将通过 [WB 仲裁器](#) 连接在一起，就像使用 [AXI-lite 到 WB 转换器](#) 一样。

这些内核中的大多数已经通过了 [形式验证](#) 检查。然而，除了 [Xilinx 内核](#) 之外，这些都没有通过 [FPGA 检查](#)——我假设其他人已经使用过，尽管我自己没有使用过。

这些核心目前都保存在我的 [Wishbone 到 AXI 桥接](#) 存储库中。他们不在那里，因为他们真的属于那里，而是因为缺乏更好的地方。

我已经写过关于 [正式验证 Xilinx 的 AXI 演示内核](#) 的博客。我甚至写过关于 [正式验证 Xilinx 的 AXI-lite 演示内核](#) 的博客，以及 [演示如何构建无错误的 AXI-lite 内核](#)。我也想为我的 [AXI \(完整\) 从属内核](#) 做同样的事情。

事实上，我想写一些或所有这些其他内核的博客。它们每个都有一些非常迷人和有用的功能。

- 例如，[AXI 从内核](#) 设计为能够在读取和写入通道上保持 100% 的吞吐量。相比之下，[Xilinx 的内核](#) 只能实现不到 50% 的读取吞吐量，而在写入通道上则接近 100%，尽管它并没有完全做到这一点。
- [AXI 交叉开关](#) 在几个方面都不同寻常。首先，它是一个公开的、开源的、经过正式验证的、经过 [正式验证](#) 的交叉开关，这很不寻常。每个人都试图模拟设计的两半，赛灵思 [互联](#) 的主端和从端？一个能够被 [验证的开源 AXI 交叉开关](#) 将非常强大。

此外，如果 [Xilinx 的 AXI 演示内核](#) 或他们的 [AXI-lite 演示内核](#) 有任何迹象，那么这个 [交叉开关](#) 的吞吐量将超过两倍。同样，这两个内核都存在潜在的错误，尚未经过 [正式验证](#)。

Xilinx 的 [交叉开关](#) 可能存在类似的潜在错误或限制。虽然我很好奇，但我无法访问他们的 [交叉开关](#) 中的逻辑 来找出答案。

- 最后，我全新的[总线故障隔离器](#) 将允许您将未经验证的 [AXI](#)设计连接到更大的系统，因为知道[总线故障隔离器](#)将识别您的内核的 [AXI](#) 接口与 我们讨论过的可用于验证和任意从机，并在任何错误的情况下返回 [总线错误](#)。

想一想。当我使用 [Cyclone-V](#)时，我自己的设计中有一个错误，两个[Wishbone 总线](#)响应合并为一个。[Cyclone-V](#) 上的 ARM 然后挂起等待响应。它从来没有来过。无论我尝试什么，我都无法进入设计以查看发生了什么。如果我有这个[总线故障隔离器](#)，我的损坏设计中的故障就会被检测到并 返回[总线 错误](#)。然后，我可以[使用逻辑](#)来挖掘问题所在以找到错误。更好的是，[总线故障隔离器](#) 现在具有恢复模式，允许在复位周期后访问从设备。

这听起来像是一张“免费进入 [FPGA 地狱](#)”的卡吗？

然而，在所有这些设计中都有一个关键组件。如果没有这个密钥，我将无法进行任何高性能 [AXI](#)设计。该关键部件是[防滑缓冲器](#)。

我知道，[我前段时间称这些为“双缓冲器”](#)，但我真的开始喜欢“[防滑缓冲器](#)”这个词了。它更好地捕捉了这个想法，所以我将转换术语并从现在开始将这些东西称为“[skid buffer](#)”。

如果您要构建或以其他方式使用 [AXI](#)设计，您确实需要了解基本的 [防滑缓冲器](#)。确实，这就是这篇文章的全部内容：我本来打算发布关于我的[AXI 从内核](#)的文章，但我意识到我要么需要花很长的篇幅来解释[防滑缓冲器](#)，要么我需要将这些材料分成自己的帖子。

## 防滑缓冲器的基本概念

那么，到底什么是[防滑缓冲器](#)？由于需要在已注册的纯数据 上下文 中创建停止信号，因此产生了[滑行缓冲区](#)。

只是为了举例说明，[ZipCPU](#)不使用 [skid buffers](#)。因此，如果 CPU 需要等待长指令，例如内存加载或除法，则 [读取操作数阶段](#) 需要暂停，以免丢失一条指令。如果 [读取操作数阶段](#) 停止，则 [解码阶段](#) 需要停止。如果 [解码阶段](#) 需要停止，[预取](#) 需要停止等等。

```
always @(*)
begin
    div_stall = div_busy;
    op_stall  = div_stall | mem_busy | cpu_halt | // other things
    dcd_stall = op_stall | pipeline_hazard;
    pf_stall  = .. ///
end
```

因为[ZipCPU](#)在内部使用 [组合停顿信号](#)，所以当停顿信号到达 [预取](#)阶段时，在下一个时钟沿之前信号中没有多少余量了。事实上，这是我尝试以 更高的时钟频率运行[ZipCPU](#)时遇到的问题之一。（这不是唯一的问题……）

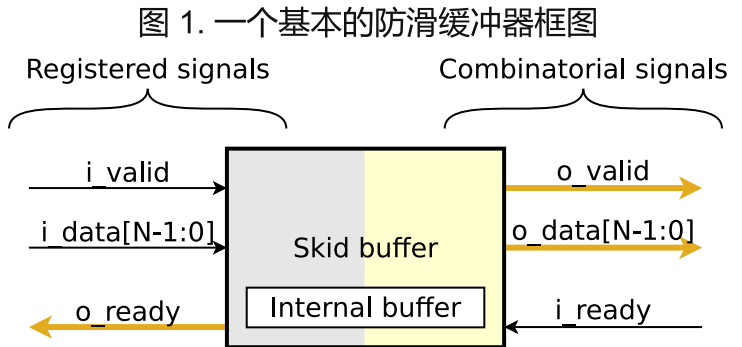
如果我改用了**防滑缓冲器**，则可能会**注册停止信号**，从而破坏时间累积。

问题是，如果**停止信号**被注册，那么**管道中的前一个处理阶段直到完成其处理并将其值注册到下一组触发器**时才知道停止。在这一点上，数据需要去某个地方或被丢弃。

输入一个**防滑缓冲器**，如图 1 右侧所示。

图 1 中的**防滑缓冲器**的目标是在一侧的组合逻辑和另一侧的注册逻辑之间架起桥梁——假设输出停止信号（即 `!o_ready`）只能是注册信号。

在这种情况下，我使用了**AXI** 信号约定，因此这个**防滑缓冲区** `VALID` 在传入和 `READY` 传出接口上都有信号。



该**防滑缓冲器**必须满足两个大标准。首先，如果有的话 `VALID & !READY`，相应的数据值必须在下一个时钟周期内保持不变。其次，在此过程中不会丢失任何数据。

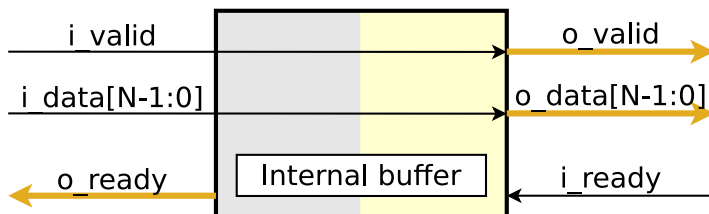
其中第一个很容易**正式**表达。

```
// First, we assume this property of the input
assume property (@(posedge i_clk)
    disable iff (i_reset)
    (i_valid && !o_ready) | => i_valid && $stable(i_data));

// Then we assert it when describing the output
assert property (@(posedge i_clk)
    disable iff (i_reset)
    (o_valid && !i_ready) | => o_valid && $stable(o_data));
```

在我所有的**AXI**内核中，我都希望尽快使用这些数据。这意味着我不想在传入数据中添加不必要的缓冲区或路径逻辑。

图 2. 没有停顿，缓冲区就像一个直通设备



因此，当一切顺利且没有任何事情发生停滞时，**防滑缓冲器**需要像直通设备一样运行，如图 2 所示。

在这种情况下，输入的有效信号和数据信号都通过**内核**并直接进入输出。

但是，如果输出端口停止，那么我们需要将所有内容复制到内部缓冲区，即“skid”缓冲区，以免输入数据值在下一个周期丢失。这就是右图3的意思。在该图中，传入的有效线和数据线直接复制到缓冲区。

图 3. 将传入数据复制到内部缓冲区

这为缓冲器提供了自己的内部有效线和数据线。再往下，当我们开始讨论这个核心的实现时，`r_valid` 我们将分别命名它们 `r_data`——但我已经超越了自己。

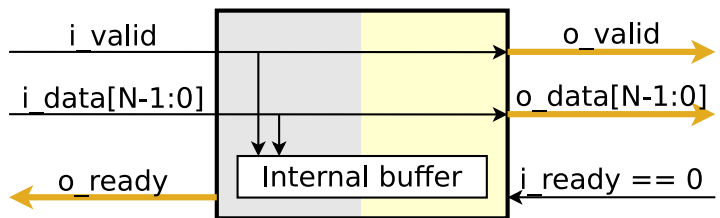
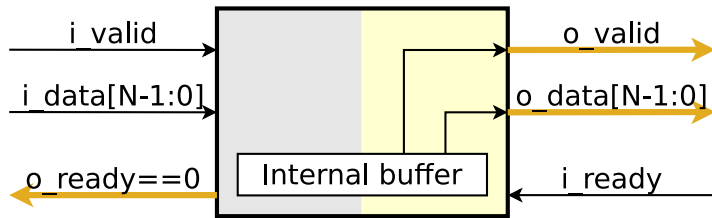


图 4. 失速信号向上游传播



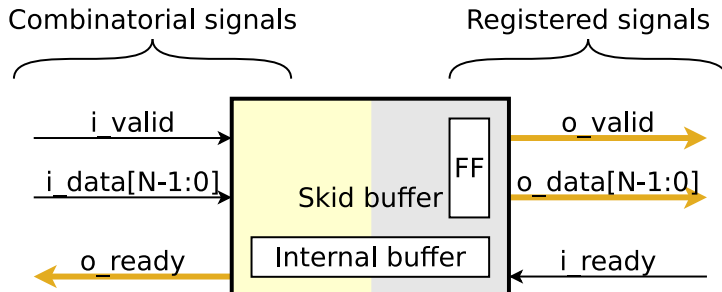
在下一个时钟周期，内核可以输出上一个周期的输入值——它刚刚缓冲的值。此外，传入接口可能会移动到其下一个值，但此时传入 `o_ready` 信号已下降，因此馈送此接口的模块现在知道它需要等待。这在左侧的图 4 中显示。

所有这一切的巧妙之处在于，实现防滑缓冲区所需的逻辑相当简单，所以让我们来看看今天构建其中一个所需的逻辑。

## 可选增强功能

在构建了第一个防滑缓冲器后，我很快意识到需要一个类似的防滑缓冲器，但输出侧颠倒了。也就是说，是否可以在传入侧为组合且传出侧已注册的位置创建防滑缓冲区？

图 5. Skid 缓冲区可以用来注册输出



我们的防滑缓冲器可能如图 5 所示。

在这种情况下，最大的区别是用于注册输出的触发器集。

这听起来很简单。但如果我两者都想要呢？如果我可以在传入 AXI 通道上使用防滑缓冲区时使用组合逻辑来创建传出接口，并在使用防滑缓冲区

驱动传出 AXI 通道时在传入接口上使用组合逻辑会怎样？为了支持这两个目的，我创建了一个可用于在它们之间进行选择的参数，. 设置此参数后，`skid buffer` 将注册所有输出。如果未设置，则可以组合驱动输出。在所有情况下，都会记录上游停止信号。 `OPT_OUTREG`

当然，这很容易构建，对吧？

不完全的。我想要更多。

我注意到，在我的许多 Wishbone 总线设计中，Wishbone 信号具有非常高的扇出。它们在整个芯片上发送数据。当如此高的扇出信号以最小化 LUT 的方式设置时，它们最终可能会经常转换——即使有效信号（`STB` 对于 Wishbone）很低，因此另一端没有阅读器正在收听。此外，这些高扇出转换中的每一个都需要供电。

如果我想在有效信号指示输出不活动时强制任何未使用的数据线为零怎么办？这可能会防止不必要的转换，甚至可能会降低我设计的功耗。（这还有待确定，但这个设计是我为一两个实验

做准备的一部分，看看是否是这样。)

为此，我分配了另一个选项，`OPT_LOWPOWER`。`OPT_LOWPOWER` 实际上是由一组 [形式](#) 属性定义的。如果 `OPT_LOWPOWER` 设置了，那么任何时候 `!o_valid` 都为真，那么 `o_data` 应该为零。

```
assert property (@(posedge i_clk)
    !o_valid |-> (o_data == 0));
```

支持这两个选项也意味着我们将根据所选择的选项从本质上设计四种单独的[防滑缓冲器](#)设计。这些选项将在整个设计中混合，以控制逻辑生成。因此，在进行验证时，我们还需要确保我们验证了[此代码](#)中的所有四种设计。

## Verilog 代码

按照我的常规做法，我将跳过大部分介绍性代码，直接跳到[防滑缓冲区示例](#)本身。

```
module skidbuffer(i_clk, i_reset,
                 i_valid, o_ready, i_data,
                 o_valid, i_ready, o_data);
    parameter      [0:0]    OPT_LOWPOWER = 0;
    parameter      [0:0]    OPT_OUTREG = 1;
```

或者，也许我们将主要跳过这个介绍性代码。

我想指出，两者 `OPT_LOWPOWER` 都是 `OPT_OUTREG` 单比特参数，使真/假逻辑测试更简单，并防止我们意外尝试将值设置为真 (1) 或假 (0) 以外的值。

```
parameter      DW = 8;
input  wire      i_clk, i_reset;
input  wire      i_valid;
output reg       o_ready;
input  wire      [DW-1:0] i_data;
output reg       o_valid;
input  wire      i_ready;
output reg       [DW-1:0] o_data;

//
// We'll start with skid buffer itself
//
reg          r_valid;
reg          [DW-1:0] r_data;
```

内部防滑缓冲器本身由两个信号捕获。第一个，`r_valid`，只是表示内部缓冲区中有有效数据。这如上面的图 3 所示，它显示了进入内部缓冲区的数据，以及上面的图 4，它显示了从内部缓冲区出来的数据。

这个 `r_valid` 信号的基本逻辑是我们希望它在任何时候有一个有效的输入信号但输出路径被停止时变高。

```
initial      r_valid = 0;
always @(posedge i_clk)
if (i_reset)
    r_valid <= 0;
else if ((i_valid && o_ready) && (o_valid && !i_ready))
    // We have incoming data, but the output is stalled
    r_valid <= 1;
```

一旦传入的就绪信号恢复正常，我们就可以恢复正常操作，再次充当直通设备。

```
else if (i_ready)
    r_valid <= 0;
```

数据逻辑更简单。每当输出组合端准备好时，我们都会悄悄地将输入值复制到我们的缓冲区分中。

```
always @(posedge i_clk)
if (o_ready)
    r_data <= i_data;
```

但是，上面的逻辑并没有保留我们的 `OPT_LOWPOWER` 属性，如下所示，只要为假，`r_data` 它就应该为零。`r_valid`

```
assert property (@(posedge i_clk)
    !r_valid |-> r_data == 0);
```

为了实现这种低功耗逻辑，我们需要确保它 `r_data` 从零开始。不仅如此，任何时候设计被重置 `r_valid` 都会被重置为零，所以我们需要 `r_data` 在这两种情况下都设置为零——但前提 `OPT_LOWPOWER` 是设置了。

```
initial      r_data = 0;
always @(posedge i_clk)
if (OPT_LOWPOWER && i_reset)
    r_data <= 0;
```

这也意味着，只要输出端没有停滞，我们也需要保持 `r_data` 为零。

```
else if (OPT_LOWPOWER && (!o_valid || i_ready))
    r_data <= 0;
```

最后，我们可以在传出/上游端没有停滞的任何时候复制数据，就像以前一样。



```
else if ((!OPT_LOWPPOWER || i_valid) && o_ready)
    r_data <= i_data;
```

或者更确切地说，我们不能，因为那不太正确。如果我们在这两种 `OPT_LOWPPOWER` 模式下，并且我们正在注册我们的输出，那么我们需要确保我们只 `i_valid` 在为真时设置这个值。否则，如果 `OPT_LOWPPOWER` 为真，则输入和 `r_valid` 属性将强制输出为零。

虽然我可以这样写，

```
else if ((! (OPT_LOWPPOWER && OPT_OUTREG) || i_valid) && o_ready)
```

我更喜欢使用[德摩根定律](#)扩展逻辑。因此，下面的条件捕获了相同的逻辑。

```
else if ((!OPT_LOWPPOWER || !OPT_OUTREG || i_valid) && o_ready)
    r_data <= i_data;
```

在我使用它们的最初几年中，我错过了[防滑缓冲器](#)实现的一个非常深刻的关键特性：输出停止信号由内部缓冲器的有效信号给出。两者是完全等价的信号。好吧，我承认在我运行[正式](#)证明之前我自己都不相信，但这无关紧要。在这种情况下，由于我们使用 [AXI](#) READY/VALID 表示法，这意味着输出的 READY（未停止）信号与我们的 VALID 信号相反。

```
always @(*)
    o_ready = !r_valid;
```

在意识到这一点之前，我构建并实现了许多[skid buffer](#)。即使我第一次看到这种等效性，仍然需要一些时间（和[正式](#)的证明）才能相信它。也就是说，它很好地简化了任何实现。

现在我们已经处理了内部缓冲区，我们可以转到传出接口。不过，我们需要将此逻辑分成两部分：一部分用于输出寄存器未缓冲的简单情况，另一部分用于它们的情况。

```
generate if (!OPT_OUTREG)
begin
```

在未注册的情况下，我们希望我们的输出端口在任何时候都有效，无论是输入端口是有效的，还是我们的[skid buffer](#)中有数据。

```
always @(*)
    o_valid = (i_valid || r_valid);
```

这也是界面的组合方面，所以你可能会注意到 `always @(*)`。

至于我们的输出数据，我们希望它在缓冲区处于活动状态的任何时候都来自缓冲区，否则就是通过。

```
always @(*)
  if (r_valid)
    o_data = r_data;
  else
    o_data = i_data;
```

嗯，差不多。如果传入 `i_data` 没有观察到低功率属性怎么办？在这种情况下，我们只需要设置 `o_data` 传入 `i_data` 值（如果 `i_valid` 也设置了），否则我们希望强制输出为零。

```
always @(*)
  if (r_valid)
    o_data = r_data;
  else if (!OPT_LOWPOWER || i_valid)
    o_data = i_data;
  else
    o_data = 0;
```

否则，传出接口逻辑似乎很简单。但是我们注册传出数据的情况呢？

那只是有点棘手。

也许有效线没有任何不同，除了它可以重置的现实。

```
end else begin

  initial      o_valid = 0;
  always @(posedge i_clk)
  if (i_reset)
    o_valid <= 0;
  else if (!o_valid || i_ready)
    o_valid <= (i_valid || r_valid);
```

也就是说，我以前被这种逻辑烧死了，所以我已经到了我总是用上面的结构来构建它的地步。注意 `if (!o_valid || i_ready)` 条件。这是让我抓到几次的作品。它与说的基本相同，但使用德摩根定律 `if (!(o_valid && !i_ready))` 重写，因此它描述了任何时候传出接口没有停止。

我的问题是我一直想向这样的通道添加其他逻辑，就像我们在[文章中讨论的最常见的 AXI 错误](#)一样。

请注意：如果您使用此基本握手来处理已注册的信号，您将只想使用此模式，仅此而已！我怎么知道这个？因为每次我做不同的事情时，[正式的](#) 工具都会纠正我。这似乎是受此类[握手](#)规则约束的信号的唯一有效方法。

这是否意味着这种格式也适用于 `o_data` 信号？绝对地！



`o_data` 我们从随时重置开始 `OPT_LOWPOWER`，然后如果输出停止，我们将拒绝任何进一步的逻辑。

```
initial      o_data = 0;
always @(posedge i_clk)
  if (OPT_LOWPOWER && i_reset)
    o_data <= 0;
  else if (!o_valid || i_ready)
    begin
```

现在可以在下一个逻辑花絮中找到在注册上下文中进行这项工作的关键。首先，如果缓冲区中有东西，则需要将其移至输出端口。如果不是，但如果输入端口有东西进入，那么我们将设置为该输出。

```
        if (r_valid)
            o_data <= r_data;
        else if (!OPT_LOWPOWER || i_valid)
            o_data <= i_data;
        else
            o_data <= 0;
    end

end endgenerate
```

不过，和以前一样，我们可以进行优化，但如果我们处于 `OPT_LOWPOWER` 模式中，则不能。

在我离开这个话题之前，请注意我使用的关键特性 `OPT_LOWPOWER`：如果没有设置，那么所有的 `OPT_LOWPOWER` 逻辑（保存初始语句）都会消失。因为是一个常数，所以如果清楚 `OPT_LOWPOWER` 的话，合成器可以处理优化这个逻辑。`OPT_LOWPOWER` 基本上也是如此 `OPT_OUTREG`，但该信号还有更多情况。

这就是 `skid buffer` 的实现。如您所见，逻辑非常简单，实际上只有两个内部寄存器与之关联：

`r_valid` 和 `r_data`。如果输出也被注册，那么 `o_valid` 和 `o_data` 也将被注册。

这导致我们进入下一步：证明[这个实现](#)是有效的，并且它做了它应该做的事情。

## 正式验证

我们已经看到了上面的几个 [形式](#) 属性。稍后我将在下面再次重复这些内容。现在，让我们从重置属性开始。

任何复位后，需要清除所有有效行。我们可以假设这是我们的输入信号。

```
`ifdef FORMAL
    // Reset properties
```

```
property RESET_CLEARS_IVALID;
    @(posedge i_clk) i_reset | => !i_valid;
endproperty
```

在这种情况下，我将其声明为命名属性——SystemVerilog 断言语言的一个特性。我稍后会回到这个问题上，要么断言，要么假设它。

我们还想假设任何时候传入接口都停止，即输入端有有效数据但 `o_ready` 为低电平时，有效信号需要继续到下一个时钟周期并且不允许更改数据。

```
property IDATA_HELD_WHEN_NOT_READY;
    @(posedge i_clk) disable iff (i_reset)
        i_valid && !o_ready | => i_valid && $stable(i_data);
endproperty
```

现在这就是我将这些声明为命名属性的原因：当我使用 [此缓冲区验证我的AXI 从属内核](#)时，我意识到这些[假设可能会使证明无效](#)。它们必须转换为该证明的断言，而作为假设保留在其中。

为了处理这个问题，我创建了一个 `SKIDBUFFER` 宏来确定是否应该假定或断言属性。使用这个宏，我可以根据需要选择假设或断言。

```
`ifdef SKIDBUFFER
    assume property (RESET_CLEARS_IVALID);
    assume property (IDATA_HELD_WHEN_NOT_READY);
`else
    assert RESET_CLEARS_IVALID;
    assert IDATA_HELD_WHEN_NOT_READY;
`endif
```

这些是描述传入接口的仅有的两个假设。

在输出端，我们将快速重复重置属性：在任何重置之后，都需要清除两个有效信号。

```
assert property (@(posedge i_clk)
    i_reset | => !r_valid && !o_valid);
```

我们现在可以开始遍历我们的内部和输出信号。

我们要保留的重要规则是，任何时候在输出端口上有一个未完成的请求被停止，即 `o_valid && !i_ready`，那么该请求必须在下一个时钟保持不变。

```
// Rule #1:
//      Once o_valid goes high, the data cannot change until the
//      clock after i_ready
assert property (@(posedge i_clk)
```

```

disable iff (i_reset)
o_valid && !i_ready
|=> (o_valid && $stable(o_data));

```

只是意味着如果重置为高，`disable iff (i_reset)` 我们将不会检查此测试。就个人而言，我认为这是不言而喻的，但是，众所周知，[正式](#) 工具有时会不同意我的看法。

第二条规则试图捕获“不得丢弃任何数据”策略。具体来说，如果传入端口上有数据，那么它要么需要转到输出端，要么需要缓冲。

```

// Rule #2:
//      All incoming data must either go directly to the
//      output port, or into the skid buffer
assert property (@(posedge i_clk)
    disable iff (i_reset)
    (i_valid && o_ready
        && (!OPT_OUTREG || o_valid) && !i_ready)
    |=> (r_valid && r_data == $past(i_data)));

```

其他情况呢？好吧，如果是 `!i_valid` 或 `i_valid && !o_ready`，那么我们需要担心的输入端口上不会发生任何事情。由于 `r_valid` 等价于 `!o_ready`，我们知道唯一有趣的情况 `r_valid` 是低的情况。如果 `r_valid` 是低和 `i_ready` 高，核心是一个简单的通过，快速代码检查将证明有效。这就留下了 `r_valid` 低和 `i_ready` 低的情况——我们上面提到的情况。

不过，这并不能完全捕捉到所有内容。我们现在讨论了信息应该如何流过这个设计，而不是设计应该如何回到空闲状态。这很重要，而且我之前没有检查返回空闲状态，这让我很伤心。因此，我们要确保设计将返回空闲状态。

所以任何时间 `i_ready` 在传出接口上都是真的，那么一切都应该被清除。在下一个时钟上，只有当它也是真的 `o_valid` 时才应该 `i_valid` 是真的。

```

// Rule #3:
//      After the last transaction, o_valid should become idle
generate if (!OPT_OUTREG)
begin

    assert property (@(posedge i_clk)
        disable iff (i_reset)
        i_ready |=> (o_valid == i_valid));

```

但是，如果我们在传出接口上注册端口怎么办？

在这种情况下，将适用两条规则。首先，任何时候输入被接受，然后 `o_valid` 在下一个时钟应该是高电平。

```

end else begin

    assert property (@(posedge i_clk)
        disable iff (i_reset)
        i_valid && o_ready | => o_valid);

```

其次，任何时间 `i_ready` 都是真实的，并且输入或缓冲区中都没有任何内容，然后 `o_valid` 应该在下一个时钟清除。

```

    assert property (@(posedge i_clk)
        disable iff (i_reset)
        !i_valid && !r_valid && i_ready | => !o_valid);

end endgenerate

```

这检查了 `.` 的上升和下降 `o_valid`。看起来很简单。

但是呢 `r_valid`？

好吧，如果 `r_valid` 在传出端口为真 `i_ready`，则防滑缓冲区被复制到传出端口，并且 `r_valid` 必须在下一个时钟取消断言。上面的图 4 就是这种情况。

```

// Rule #4
//      Same thing, but this time for r_valid
assert property (@(posedge i_clk)
    r_valid && i_ready | => !r_valid);

```

如果传入接口上也有东西进来怎么办？它不会发生。记住，`o_ready = !r_valid`。因此，如果 `r_valid` 为高，则传入接口停止，因此我们可以忽略它。

这留下了我们上面讨论的两个特殊的低功耗特性。我们只想强制执行那些 if `OPT_LOWPOWER` 设置，否则我们想忽略它们。因此，我们将使用生成块来捕获这些检查。这意味着如果 `OPT_LOWPOWER` 未设置，则综合工具（即 `yosys`）甚至不会创建支持这些检查的逻辑。

```

generate if (OPT_LOWPOWER)
begin
    //
    // If OPT_LOWPOWER is set, o_data and r_data both need
    // to be zero any time !o_valid or !r_valid respectively
    assert property (@(posedge i_clk)
        !o_valid | -> o_data == 0);

    assert property (@(posedge i_clk)
        !r_valid | -> r_data == 0);

```

```

        // else
        //     if OPT_LOWPOWER isn't set, we can lower our logic
        //     count by not forcing these values to zero.

    end endgenerate

```

这些都是我们需要知道的所有属性，但它真的有效吗？

为此，我们将转向覆盖。

## 覆盖

与上面的安全（断言/假设）属性不同，如果没有找到使断言为假的踪迹，同时保持所有假设为真，则可以证明其为真，只有在至少找到一条踪迹时，覆盖才会成功。Cover 对于发现假设中的错误，证明可以进行特定操作等等非常有用。

当您只想要显示设计有效的痕迹时，它也非常有价值。

因此，让我们构建这样一个以核心空闲开始和结束的跟踪。在中间，我们会坚持 `i_ready` 线从高到低切换两次，然后再回到高位。

哦，我有没有提到我们只想在这个单元被单独验证的情况下检查封面？否则，可能是父模块永远不会使这个 `cover()` 陈述为真——这不会是一个错误。

```

`ifdef SKIDBUFFER
    reg    f_changed_data;

    // Cover test
    cover property @(posedge i_clk)
        disable iff (i_reset)
        (!o_valid && !i_valid)
        ##1 i_valid && i_ready [*3]
        ##1 i_valid && !i_ready
        ##1 i_valid && i_ready [*2]
        ##1 i_valid && !i_ready [*2]
        ##1 i_valid && i_ready [*3]
        // Wait for the design to clear
        ##1 o_valid && i_ready [*0:5]
        ##1 (!o_valid && !i_valid && f_changed_data));

```

结果跟踪很有趣，但可能会更好。特别是，传入的数据全为零。虽然这是有效的，但它并不是很有启发性。我宁愿能够从跟踪中“看到”各种数据线正在正常运行。也许如果我们坚持传入的数据是一个计数器？

执行此操作的简单方法是添加另一个寄存器以及与之相关的一些逻辑。让我们称之为 `f_changed_data`，并用它来表示我们的数据在整个覆盖跟踪中“正确”地变化。也就是说，

`f_changed_data` 如果传入的数据计数，将捕获——这样我们就可以更容易地可视化跟踪和正在发生的事情。

在许多方面，这不是您典型的“正式”财产。它不使用任何正式的语言特性（除了 `$past()`）。但是，如果整个世界开始看起来像一个 Verilog 问题，那么解决方案很容易就是一个简单的 Verilog 逻辑。

我们将首先将此 `f_changed_data` 标志设置为 `true`，如果我们希望在覆盖语句中看到的任何规则发生更改，则清除该标志。

```
initial          f_changed_data = 0;
always @(posedge i_clk)
  if (i_reset)
    f_changed_data <= 1;
```

`i_data` 只允许在以下时钟上更改 `!i_valid || o_ready`。

```
else if (i_valid && $past(!i_valid || o_ready))
  begin
```

在这种情况下，我们将清除 `f_changed_data` 任何 `i_data` 不增加的时间。

```
    if (i_data != $past(i_data + 1))
      f_changed_data <= 0;
```

同样，我们希望在输入不是一直有效的任何时候清除这个值。

```
    end else if (!i_valid && i_data != 0)
      f_changed_data <= 0;
```

瞧！现在，我们在图 6 中有一个很好的轨迹，显示了这个核心是如何工作的。

是的，此跟踪已被编辑，但仅进行了最低限度的编辑。

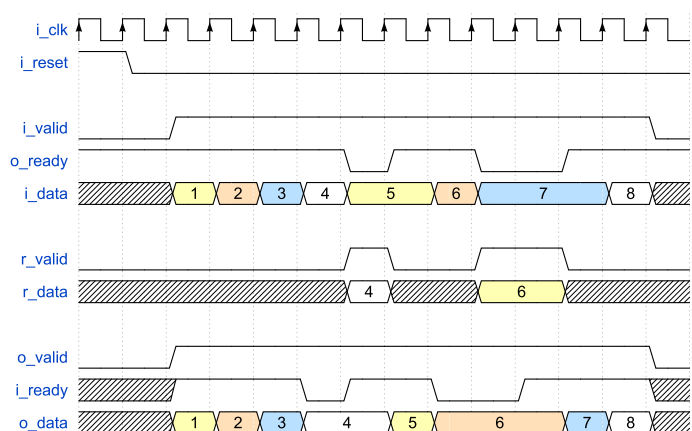
## SymbiYosys 脚本

在大多数情况下，用于驱动此类证明的 SymbiYosys 脚本 非常基础。然而，在这种情况下，脚本有几个关键特性 值得讨论。

首先，您可能还记得我在上面提到过我们需要进行四个单独的证明才能尝试我们

`OPT_LOWLOGIC` 和 `OPT_OUTREG` 参数的所有组合。使用 SymbiYosys 配置文件 `[tasks]` 的部

图 6. 来自这个防滑缓冲区的覆盖轨迹





分，这些中的每一个都可以分成自己的证明。我们还将为封面证明创建一个任务。

```
[tasks]
prfc prf
prfo prf          opt_outreg
lpc  prf opt_lowpower
lpo  prf opt_lowpower opt_outreg
cvr
```

如果您以前从未见过 [SymbiYosys 文件](#) `[tasks]` 的某个部分，那么您将大有收获。本节中的每一行都定义了一个单独的 [正式](#) 运行。行中的第一个标识符给出了运行的名称，随后的标识符是标签，然后将其应用于运行并在配置时有用。

第二部分，即 `[options]` 部分，展示了使用任务的第一个原因：我们所有的归纳证明都可以在 3 步内完成，而封面通行证需要 20 步。在这里，您会注意到上面的每个任务都有 `prf` 标签或 `cvr` 名称（也是标签）。这允许我们为每个通道设置不同的深度。

```
[options]
prf: mode prove
prf: depth 3
cvr: mode cover
cvr: depth 20
```

这个设计很简单，我们使用什么引擎并不重要，所以我们将使用默认值。

```
[engines]
smtbmc
```

然而，真正的行动是在该 `[script]` 部分。本节包含一系列要提供给 [yosys](#) 的命令，以控制设计的处理方式。这也意味着 如果您遇到任何令人困惑的事情，您可以使用 [yosys 命令](#)。 `help`

我们将通过定义 `SKIDBUFFER` 宏开始脚本，然后将 我们的代码读入 [yosys](#)。

```
[script]
read -define SKIDBUFFER
read -formal skidbuffer.v
```

我的设计通常由许多部分组成。对于这些设计，我会在这里用一行来读取每个输入文件。

然而，下一步是从外部控制参数。 [yosys](#) 对该命令进行了新的扩展， `hierarchy` 以使这更容易。基本上，该 `hierarchy` 命令会找到顶层模块并实例化它下面的所有逻辑。在我们的例子中，我们希望根据证明实例化特定的逻辑。因此，我们将使用该 `chparam` 选项 `hierarchy` 来设置这些参数。

[这与我之前讨论](#) 的方法不同。以前，我会写，

```
opt_outreg:    chparam -set OPT_OUTREG 1    skidbuffer
~opt_outreg:   chparam -set OPT_OUTREG 0    skidbuffer
opt_lowpower:  chparam -set OPT_LOWPOWER 1  skidbuffer
~opt_lowpower: chparam -set OPT_LOWPOWER 0  skidbuffer
```

这种用法存在一些问题，因此现在已被弃用。其中一个问题是yosys会在每次调用 `chparam`。由于沿途参数设置不兼容而导致的任何细化错误都可能导致整个过程停止。

相反，使用该 `hierarchy` 命令可以一次设置每个参数。因此我们可能想要使用，

```
prfc: hierarchy -top skidbuffer -chparam OPT_LOWPOWER 0 -chparam OPT_OUTREG 0
prfo: hierarchy -top skidbuffer -chparam OPT_LOWPOWER 0 -chparam OPT_OUTREG 1
lpc:  hierarchy -top skidbuffer -chparam OPT_LOWPOWER 1 -chparam OPT_OUTREG 0
lpo:  hierarchy -top skidbuffer -chparam OPT_LOWPOWER 1 -chparam OPT_OUTREG 1
```

这种方法的问题很简单：如果你有 20 个不同的任务，所有的任务都重复相同的选项怎么办？

在这种情况下，[SymbiYosys 的 python 接口](#) 可以帮助我们。

以下脚本将独立检查我们的两个参数，并创建一个名为 `cmd` 的字符串变量，其中包含包含 `hierarchy` 适当值的行。然后，当最后 `output(cmd);` 发出调用时，`cmd` 字符串将被写入 驱动各自证明的各个yosys脚本中。

```
--pycode-begin--
cmd = "hierarchy -top skidbuffer"
cmd += " -chparam OPT_LOWPOWER %d" % (1 if "opt_lowpower" in tags else 0)
cmd += " -chparam OPT_OUTREG    %d" % (1 if "opt_outreg"    in tags else 0)
output(cmd);
--pycode-end--
prep -top skidbuffer
```

最后 `[files]` 一节是相当不起眼的。它仅列出了此证明中使用的文件。在这种情况下，它只是 `skidbuffer.v` 文件。

```
[files]
skidbuffer.v
```

[SymbiYosys](#) 会在运行证明之前将此文件复制到你处理目录中。

整个证明现在可以使用，

```
% sby -f skidbuffer.sby
```

或者，您可以将其集成到您的 [形式验证 Makefile](#)中，然后运行

```
% make
```

如果您需要示例，请随意查看 我用于这些 [AXI项目的Makefile](#) 。

## 结论

Skid 缓冲器非常强大，非常有用，尤其是在使用 [AXI](#)时。事实上，我几乎在所有[各种 AXI 设计](#)中都使用了防滑缓冲器。一遍又一遍地使用相同的逻辑，因此创建一个文件来捕获此逻辑并简化设计是有意义的。它们真的就是那么有用。

我只是希望我早点把这个逻辑分离到它自己的模块中，因为现在我有很多很多相同逻辑的副本需要维护。为此，我要感谢[Eric LaForest](#) 为我树立了一个更好的榜样。我还要向您推荐[他关于防滑缓冲器的博客文章](#)以供进一步阅读。

对于那些不熟悉[SystemVerilog](#) 的并发断言语言或无法访问商业 [SymbioticEDA 套件](#)的人来说，您可能会发现 [这个关于并发断言的替代方案和等价物的讨论](#)很有价值。


---

*完成的愿望对灵魂来说是甜蜜的：但对于愚蠢的人来说，远离邪恶是可憎的。（箴 13:19）*

---

## Gisselquist Technology 的 ZipCPU

[zipcpu@gmail.com](mailto:zipcpu@gmail.com)

 [ZipCPU](#)  
[@zipcpu](#)

 BECOME A PATRON

ZipCPU 博客，主要介绍如何讨论 FPGA 和软核 CPU 设计。该站点将专注于 Verilog 解决方案，使用专门的开源 IP 产品进行 FPGA 设计。特别关注的领域包括通常被更多主流 FPGA 设计课程遗漏的主题，例如如何调试 FPGA 设计。