片上系统实验报告

汉化版

姓名：陈宇阳 学号：3220101054 姓名：邓高远 学号：3220106412

2025

目录

[实验内容概述 2](#_Toc203500130)

[设备和工具介绍 2](#_Toc203500131)

[实验报告 2](#_Toc203500132)

[**第 2 阶段：HDMI 控制器设计** 2](#_Toc203500133)

[**第 3 阶段：用于 HDMI 控制器设计的 AXI\_Lite Slave 接口** 8](#_Toc203500134)

[**第 4 阶段：用于 HDMI 控制器设计的 AXI\_Full 主接口** 16](#_Toc203500135)

[**第 5 阶段：实时摄像机输入到 HDMI 输出** 28](#_Toc203500136)

[问题记录 46](#_Toc203500137)

[个人总结 50](#_Toc203500138)

[参考文献 52](#_Toc203500139)

[附录 53](#_Toc203500140)

# 实验内容概述

该实验总共有 5 个阶段。 第一阶段是一个基础项目，目标是让学生熟悉使用强大的 ZYNQ FPGA 芯片及其集成的 Arm CPU 进行 Vivado 模块设计的工作流程。真正有挑战性的工作从第 2 阶段开始，通过该阶段，我们需要设计一个 HDMI 控制器且先不将其集成到 SoC 系统中。完成第 2 阶段后，我们将在第 3 阶段为 HDMI 控制器设计一个 AXI\_Lite 总线从接口，使其可以由 CPU上运行的程序进行控制。然后，设计一个 AXI\_Full master接口并采用DDR显存-SRAM行缓存结构，以满足阶段 4 中更高的显示要求。最后，我们将尝试使用我们设计的电路来在屏幕上实时显示相机内容。为了关注实验的主要内容，本报告不包括较简单的第 1 阶段内容。

# 设备和工具介绍

我们在实验中使用的设备和工具如下：

一个搭载 xc7z020clg484 芯片的Zedboard开发板

b.Vivado 和 Vivado SDK 2018.2与2019.1

c. Raspberry Pi Camera V2.1，搭载 IMX219摄像头

# 实验报告

## **第 2 阶段：HDMI 控制器设计**

在第 2 阶段，我们将使用 Verilog HDL 设计一个 HDMI 控制器。在设计 1080P HDMI 控制器之前，我们需要设计一个 720P HDMI 控制器，它的时序要求更低，更容易满足。

我们需要知道的第一件事是 HDMI 是如何工作的。 [[1]](https://en.wikipedia.org/wiki/HDMI)**高清多媒体接口** （HDMI） 是一种专有的数字接口，用于在设备之间传输高质量的视频和音频信号。它通常用于连接电视、计算机显示器、投影仪、游戏机和个人计算机等设备。HDMI 支持未压缩的视频以及压缩或未压缩的数字音频，允许一根电缆同时传输这两种信号。 通常，HDMI 通过一种称为 **Transition-minimized differential signaling** （TMDS） 的差分信号来驱动外部显示器。但在 Zedboard 上，我们使用 HDMI 发射器芯片 ADV7511 来生成所需的信号。所以，我们应该做的是向 ADV7511 发出所需的信号。ADV7511 支持多种模式，但它仅在Zedboard 上支持“HDMI 1.4和DVI 1.0兼容，支持 1080P60和16位、YCbCr、4：2：2 模式颜色”。因此我们的任务是向ADV7511发送 正确的控制信号，控制信号如下表所示：

|  |  |
| --- | --- |
| 信号名称 | 描述 |
| hdmi\_scl | I2C 接口。支持 1.8V 至 3.3V 的 CMOS 逻辑电平 |
| hdmi\_sda |
| hdmi\_clk | Video clock input（视频时钟输入）。支持 1.8V 至 3.3V 的典型 CMOS 逻辑电平 |
| hdmi\_hsync | 水平同步输入 |
| hdmi\_vsync | 垂直同步输入 |
| hdmi\_de | 数据 启用数字视频的信号输入 |
| hdmi\_d[15：0] | 视频数据输入 |

表 1.HDMI 信号

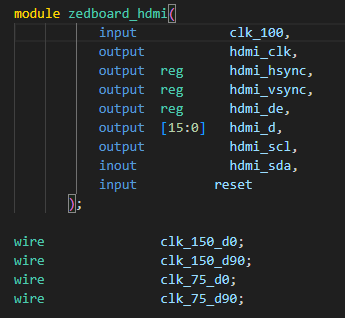
Verilog HDL 代码中的信号定义如下图所示：

图 1.HDMI 信号

这里的关键控制信号是 hdmi\_hsync 和 hdmi\_vsync。它们的工作原理如图 2 所示。

图 2 中的元素是：

**Horizontal Timing （across a single line） （水平计时 （跨单行）：**

**HSync**：标记新行开始的水平同步信号。

**Horizontal Sync Time**：HSync 脉冲的持续时间。

**Back Porch**：在可见区域开始之前，HSync 脉冲后的空闲时间。

**水平活动开始**：可见视频的开始。

**可寻址视频**：实际的可见图像内容。

**左/右边框**：图像周围的可选填充。

**Front Porch**：下一个 HSync 之前的可见区域之后的空闲时间。

**水平空白开始/结束**：标记水平消隐间隔的开始和结束，其中包括前廊、同步和后廊。

**Vertical Timing （沿线帧）：**

**VSync**：标记新帧开始的垂直同步信号。

**Vertical Sync Time**：VSync 脉冲的持续时间。

**Back Porch**：可见内容之前的 VSync 脉冲之后的空闲行。

**垂直活动起点**：可见线条的起点。

**可寻址视频**：视频内容的实际行数。

**上/下边框**：可见内容周围的可选空行。

**Front Porch**：可见内容之后和 VSync 之前的空闲行。

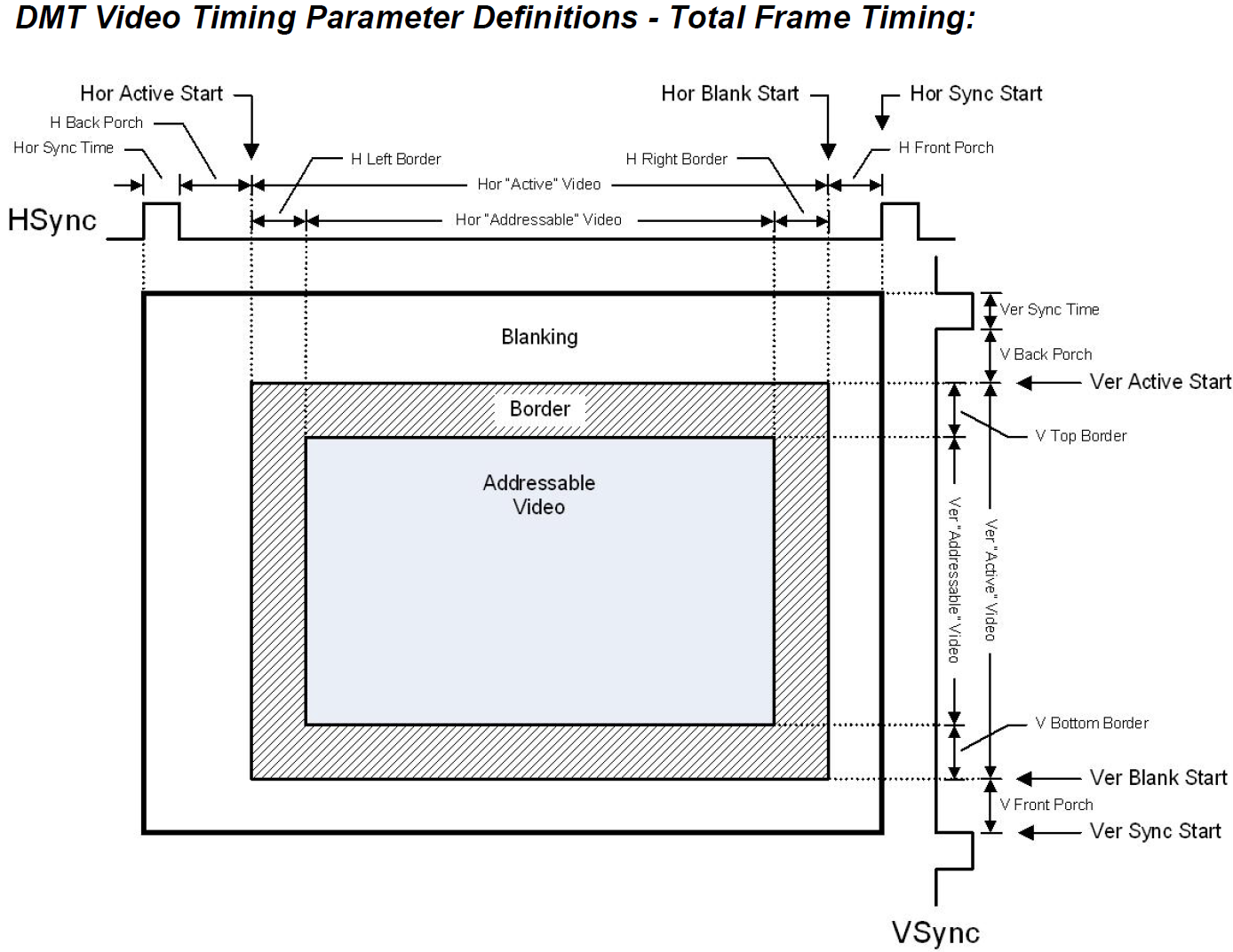
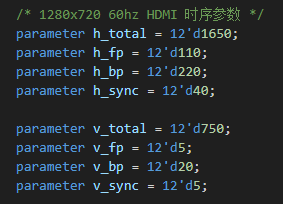
**垂直消隐开始/结束**： 标记垂直消隐周期的开始和结束。

图 2.HDMI 定时

在 720P HDMI Timing 中，图 2 中的参数定义如图 3 所示：

图 3.720P HDMI 参数

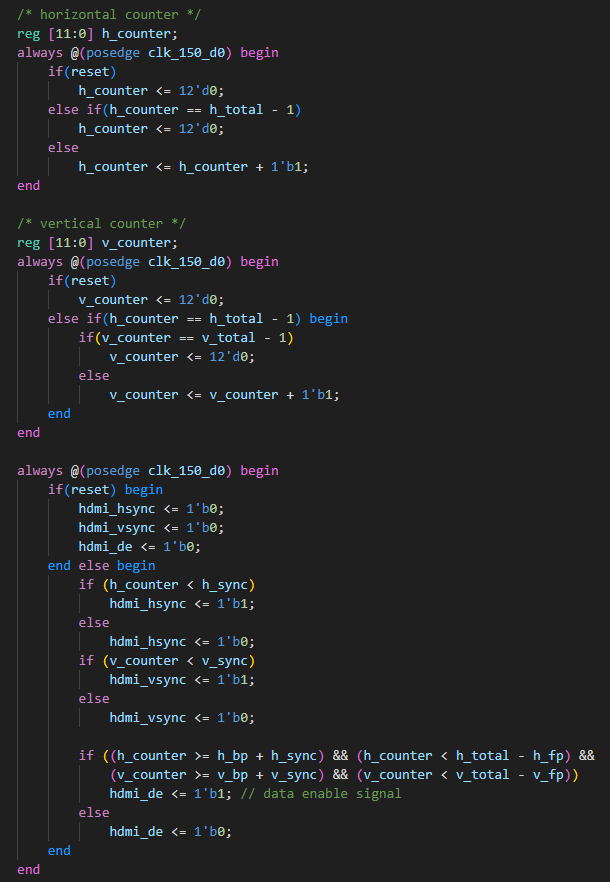
然后，我们便设计出可以在正确的时刻驱动ADV7511控制信号的电路。Verilog HDL 代码如图 4 所示。

图 4.生成 hdmi\_vsync、hdmi\_hsync 和 hdmi\_de

带有ADV7511的I2C 初始化代码。**（代码在文件 i2c\_sender.v**） 和时钟生成模块**（clk\_pll.v，720P 为 75MHz，1080P 为 150MHz，如果找不到 pll 文件，就创建一个新的，这里我就不介绍如何创建 pll 的 IP**），我们现在可以尝试驱动显示。2.3 阶段有 9 个子实验。这里我们仅展示子实验 8 和子实验 9 的代码和结果，他们的功能与代码涵盖了之前子实验的内容。

我们使用BRAM存储将要显示的图片的像素信息，并用coe文件初始化。所以关键任务便是根据 HDMI 控制信号的时序找到我们需要的像素信息，也就是计算 BRAM 中的地址，并将存储在该位置的数据传递给 ADV7511。

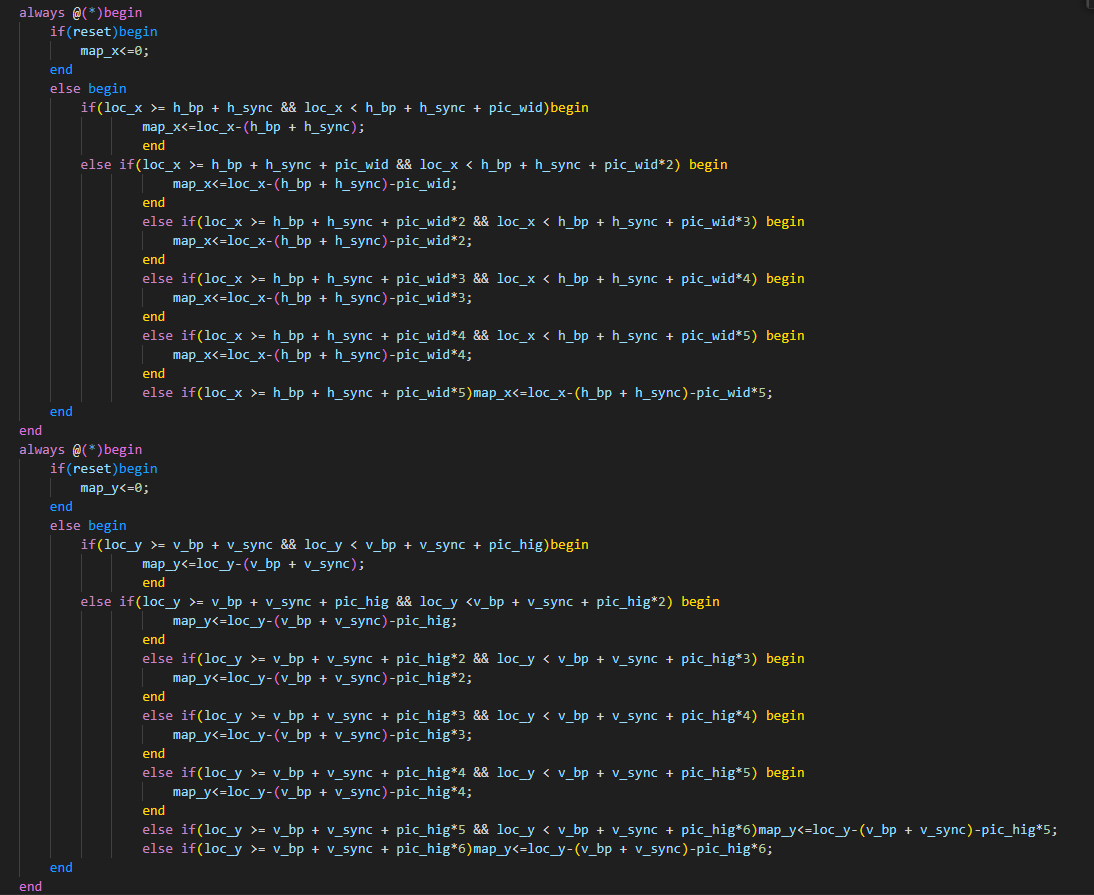
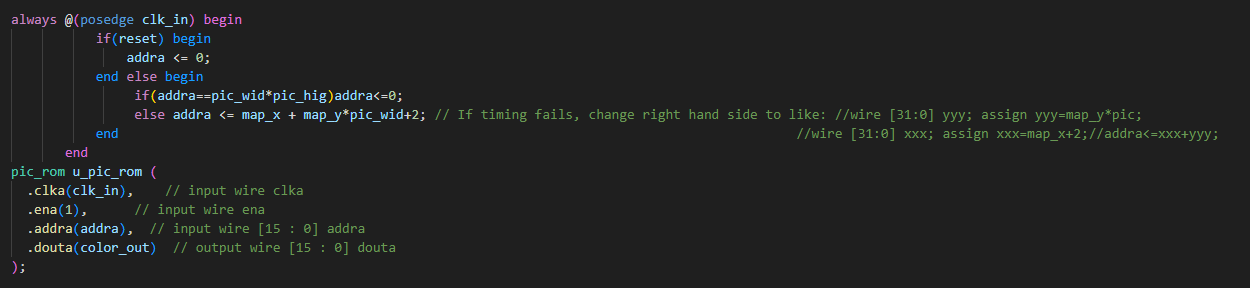
由于BRAM可用资源较小，选择创建包括小型 Airbus 的 320x175 图片的并重复显示该图片，因此地址映射如图 5 所示。

图 5.SRAM位置映射

在此代码中，***loc\_x*** 和 ***loc\_y 是根据*** h\_counter 和 v\_counter 在***显示屏上的当前像素地址***。使用图 5 中的查找表，我们成功地将其映射到 BRAM中的x和y值。然后将通过乘法和加法得到 BRAM 中的实际地址，如图 6 所示。需要注意的是，因为 BRAM 获取地址并输出数据需要一个或两个 clock cycles（取决于你如何设置 BRAM IP core）的延迟，所以应当在addra上加 2 以提前获取数据，使得其输出数据时和设计的HDMI控制信号一致。

图 6. SRAM地址计算

以下是结果的图片：

图 7.单个小飞机

图 8.循环小飞机

**注意：Dynamic Bricks 和 Dynamic ball 的结果位于目录 “./pics” 中。**

## **第 3 阶段：用于 HDMI 控制器设计的 AXI\_Lite Slave 接口**

[[2]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#AXI4-Lite)AXI4-Lite 是 AXI4 协议的子集，提供[类似寄存器](https://en.wikipedia.org/wiki/Processor_register" \o "Processor register)的结构，具有更低的功能和复杂性。 显著差异包括：

1. 所有 burst 仅由 1 拍组成
2. 所有数据访问都使用完整的数据总线宽度，可以是 32 位或 64 位

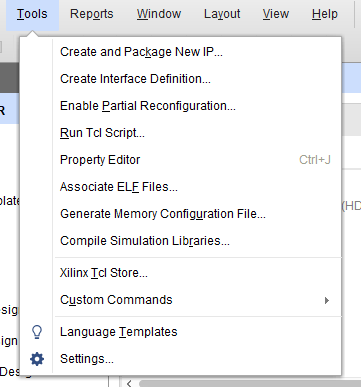
AXI4-Lite 删除了部分 AXI4 信号，但对其余信号遵循 AXI4 规范。作为 AXI4 的子集，AXI4-Lite 事务与 AXI4 器件完全兼容，无需  [额外的转换逻辑即可实现 AXI4-Lite 启动器和 AXI4 目标之间的](https://en.wikipedia.org/wiki/Interoperability)互作性。

|  |  |  |
| --- | --- | --- |
| **信号** [**（1）**](https://docs.amd.com/r/iDBE_pP8gVw3O_auFL6zrQ/LSQAg_LbI9Kgl7s357XzmA?section=XREF_64859_1_This_signal) | **方向** | **描述** |
| s\_axi\_ctrl\_aclk | 输入 | 时钟信号。该 bus 接口的所有 inputs/outputs 都与该 clock对齐。 |
| s\_axi\_ctrl\_aresetn | 输入 | 低电平有效同步复位信号 |
| s\_axi\_ctrl\_awalid | 输入 | 写入地址有效。此信号指示通道正在向有效的写入地址发送信号。 |
| s\_axi\_ctrl\_awready | 输出 | 写入地址就绪。此信号表示从站已准备好接受地址。 |
| s\_axi\_ctrl\_awaddr | 输入 | 写入地址。写入地址给出了交易的地址。 |
| s\_axi\_ctrl\_wvalid | 输入 | 写入有效。此信号指示有效的写入数据可用。 |
| s\_axi\_ctrl\_wready | 输出 | Write ready（准备好写）。此信号表示 slave 可以接受写入数据。 |
| s\_axi\_ctrl\_wdata | 输入 | 写入数据。 |
| s\_axi\_ctrl\_bvalid | 输出 | 写入响应有效。此信号表示通道正在发出有效的写入响应信号。 |
| s\_axi\_ctrl\_bready | 输入 | Write response ready（写入响应就绪）。此信号表示 master 可以接受写入响应。 |
| s\_axi\_ctrl\_bresp | 输出 | 写入响应。此信号指示写入事务的状态。 |
| s\_axi\_ctrl\_arvalid | 输入 | 读取地址有效。此信号指示通道正在向有效的读取地址发送信号。 |
| s\_axi\_ctrl\_arready | 输出 | 读取地址就绪。此信号表示从站已准备好接受地址。 |
| s\_axi\_ctrl\_araddr | 输入 | 读取地址。读取地址给出交易的地址。 |
| s\_axi\_ctrl\_rvalid | 输出 | 读取有效。此信号表示通道正在向所需的读取数据发送信号。 |
| s\_axi\_ctrl\_rready | 输入 | 阅读就绪。此信号表示 master 可以接受读取的数据和响应信息。 |
| s\_axi\_ctrl\_rdata | 输出 | 读取数据。 |
| s\_axi\_ctrl\_rresp | 输出 | 读取响应。此信号指示读取传输的状态。 |
| **注：**此信号描述摘自 Arm AMBA 协议规范。 | | |

表 2.AXI\_Lite信号[[3]](https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Lite-Interface-Signals)

在了解了 AXI 协议的工作机制之后，我们现在能够为 HDMI 控制器设计一个 AXI\_Lite Slave 接口，然后更改 CPU 上运行的程序在屏幕上显示的内容。

首先，我们单击 **Tools （工具）、Create and Package New IP （创建并打包新 IP）**

图 9.第 1 步

然后点击 **下一个** 在 弹出窗口 中进入此窗口。之后，选择

**创建新的 AXI4 外围设备**， 然后单击 **Next**。

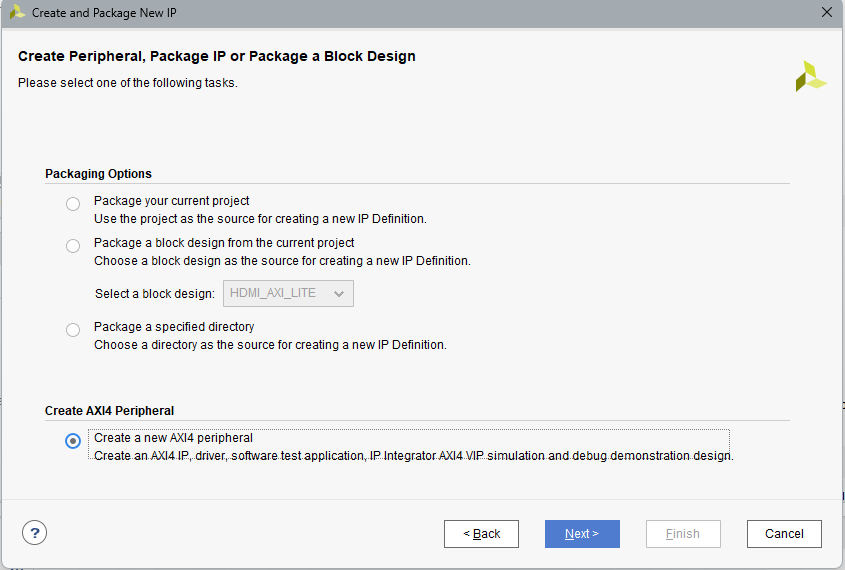


图 10.步骤 2

如图 11 所示，命名要创建的 IP 后，我们进入如图 12 所示的窗口。

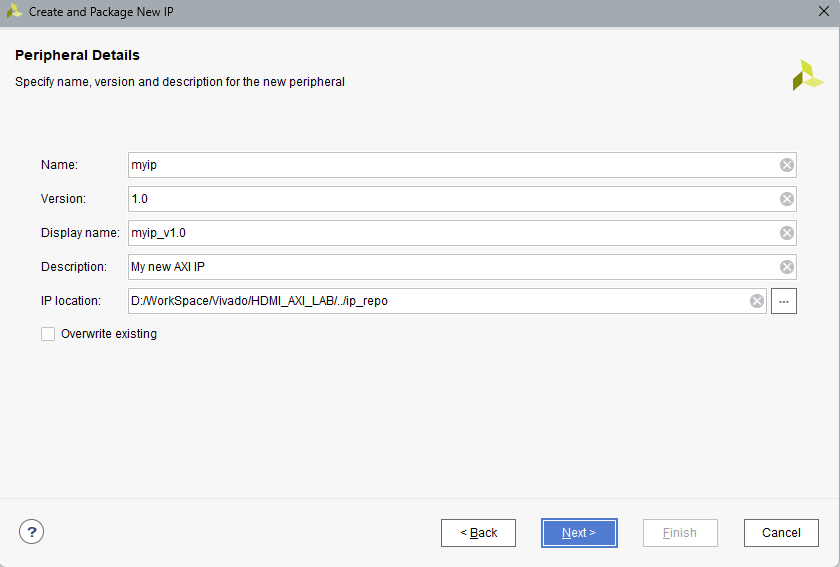


图 11.步骤 3

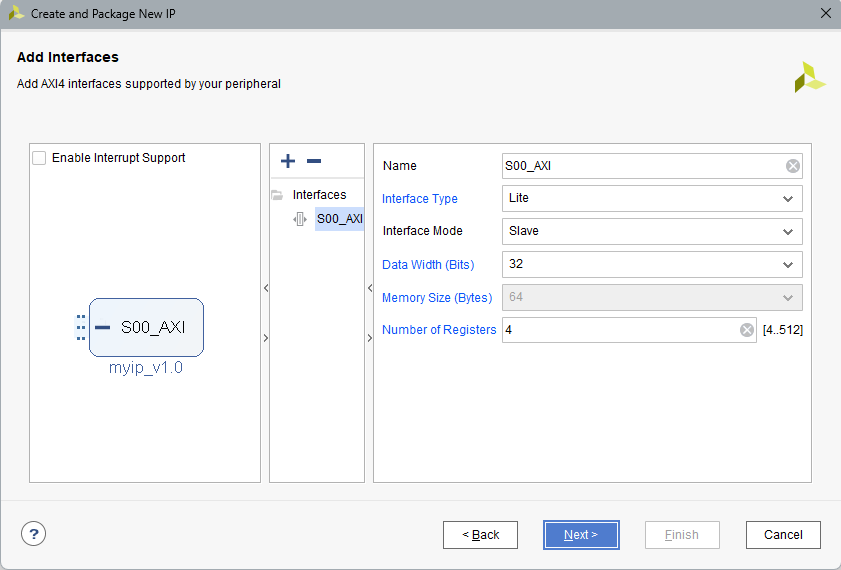


图 12.步骤 4

在如图 12 所示的窗口中，我们选择生成一个 AXI\_Lite Slave 接口。AXI 总线的数据宽度为 32 位，此处的从寄存器数量为 4。设置好所有内容后，单击 **Next** 进入最后一步，如图 13 所示。

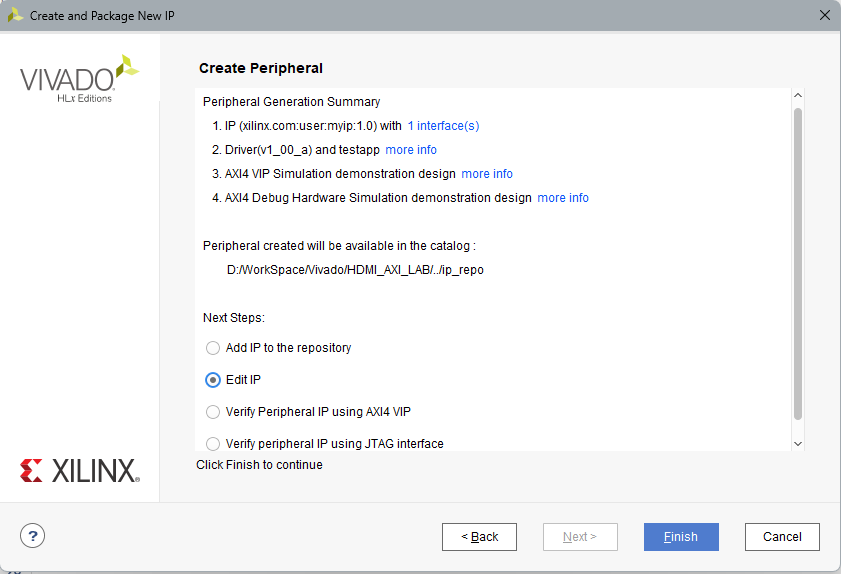


图 13.步骤 5

单击 Finish **后**，我们输入一个新项目来编辑我们想要打包的 IP。Vivado 生成 AXI 外设的两个文件。我们可以将端口添加到指定的区域，如图 14、15、16 和 17 所示。

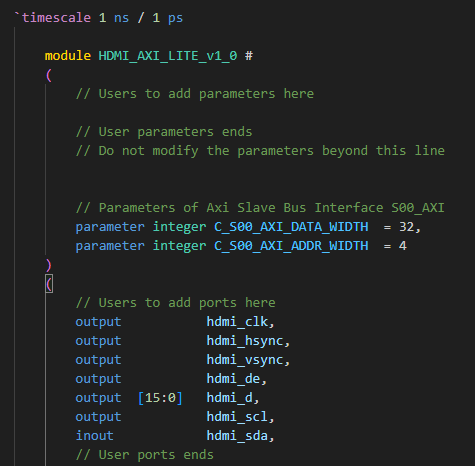


图 14.添加用户端口

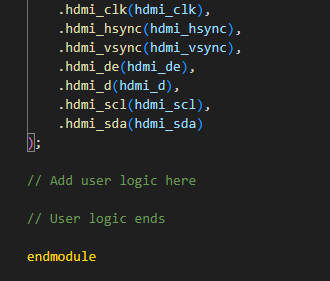


图 15.新添加端口的实例化

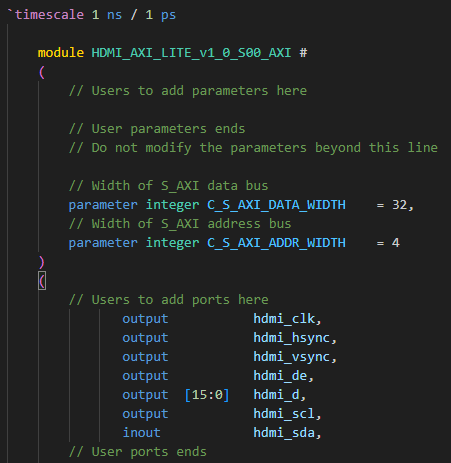


图 16.将用户端口添加到较低级别

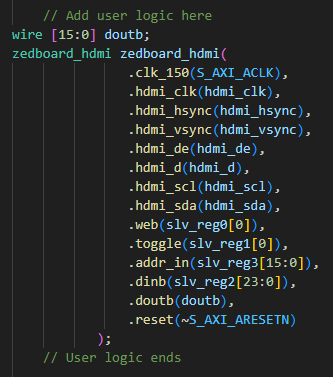


图 17.自定义模块的实例化

除了 clock signals 和 reset 信号等全局信号，以及我们之前介绍的 HDMI 信号外，我们还添加了 web***、***toggle***、***addr\_in***、***dinb ***和 doubt*** 等新信号。web（write en b）是存储像素信息的 BRAM 的写入启用信号。当 web 为 1 时，BRAM 可以写入 。需要注意的是，slv\_reg0、slv\_reg1、slv\_reg2 和 slv\_reg3 是 CPU 程序可以写入的四个从寄存器。BRAM 的端口 B（***web，dinb 和 addrb 或 addr\_in 在高级模块中***）被设计为由 CPU（***slv\_reg0[0]，slv\_reg2[23：0]，slv\_reg3[15：0]***）作，  ***这里不使用 doubt，所以你可以忽略它。***

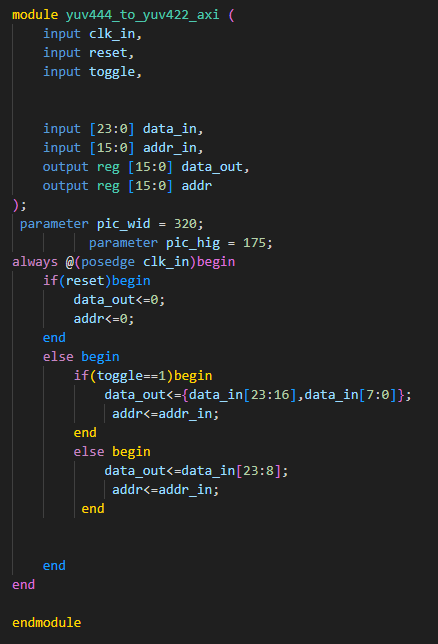
至于 Dynamic Bricks 和 Dynamic ball 的结果，它们与第 2 阶段的结果相似，因此我不会 在这里和之后重复展示它们。并且由于我们的疏忽，我们忘记记录 “Repeatedly Display many Airbus” 的结果。所以我只介绍它的代码。按照要求，这里的 Airbus.h 格式是 YUV444，但这里的 ADV7511 只能取 YUV422。因此，我们需要传输格式。考虑到 使用 CPU 传输格式不仅速度慢，而且消耗大量资源，我们决定设计一个硬件加速模块，将格式从 YUV444 传输到 YUV422。该模块的 Verilog HDL 代码如图 18 所示。

图 18.YUV444 至 YUV422

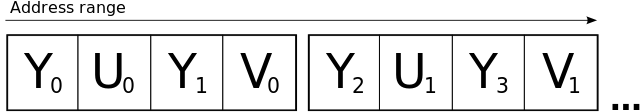
要理解这段代码，我们需要解释一下这两种格式 fisrt。 [[4]](https://en.wikipedia.org/wiki/Y%E2%80%B2UV)Y′UV，也写作 YUV，是 PAL 模拟彩色电视标准中的颜色模型。颜色被描述为一个 Y′ 分量 （luma） 和两个色度分量 U 和 V。素数符号 （'） 表示亮度是根据 Gamma 校正的 RGB 输入计算的，并且它与真实亮度不同。今天，术语 YUV 在计算机行业中通常用于描述使用 YCbCr 编码的色彩空间。 YUV422 是 YUV 的压缩形式（YUV444）（这也是使用 YUV 编码的原因，压缩起来很方便，人眼对 Y 很敏感，所以Y没有被压缩，人眼对U和 V 并不敏感。因此为了压缩数据，U和V会使用很多子采样的方法），要显示一个像素，需要对完整的 Y/U/V 信息进行压缩，YUV422 采用的是以下方法：

图 19.YUV422型

如图 19 所示，总共有 8 个字节，描述了这 4 个像素的颜色信息。

1. 第一个点是 Y0U0V0;
2. 第二个点是 Y1U0V0;
3. 第三点是 Y2U1V1;
4. 第四点是 Y3U1V1;

即每个像素都有自己的Y，相邻的两个点共享U和V，丢失了一半的U和V信息，因为人眼对 U 和 V 不敏感，很多时候对显示效果影响不大。

因此，如果要将 YUV444 传输到 YUV422，则需要如图  18 中的代码所示的信号切换。当 ***切换***为 1 时，将输入 24 位 YUV444 数据的 前 8 位和 后 8 位放在一起，得到输出的 16 位 YUV422 数据。当切换 为 0 时，取输入 24 位 YUV444 数据的前 16 位作为输出的 16 位 YUV422 数据。并且 ***toggle*** 由 CPU 上运行的程序进行控制，以确保toggle 信号、地址信号和数据信号的传递是同步的。

在 CPU 上运行的程序是用C 语言编写的。如图 20 所示，我们定义了 AXI\_Lite Slave IP 的基址，以及四个 Slave 寄存器的 Offset。

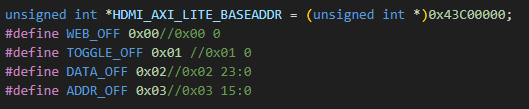


图 20. 地址 定义

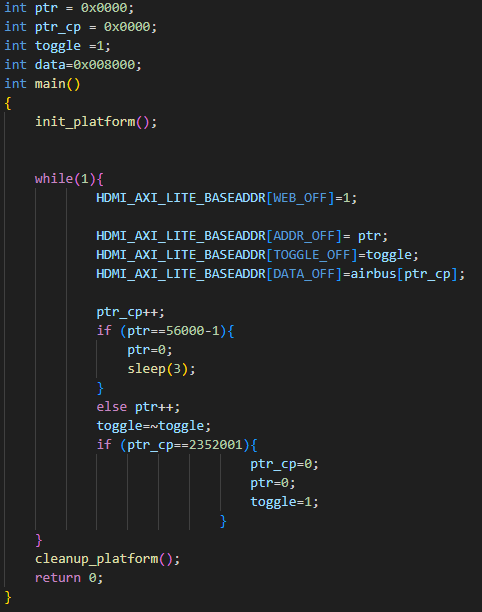
在图 21 中展示了该程序的主要功能。该程序将连续将数据和地址从***存储图片像素信息的*** airbus 数组与切换信号一起发送到从寄存器。结果是许多不同的 airbus 图片将按顺序重复显示。每张照片每次将显示3秒。

图 21.C 源代码

## **第 4 阶段：用于 HDMI 控制器设计的 AXI\_Full 主接口**

在实现为 HDMI 控制器创建 AXI\_Lite Slave 接口的目标后，我们将把HDMI 控制器放在一个 AXI\_Full Master 接口中以在屏幕上显示 1920x1080 的图片。AXI\_Full 是Advanced eXtensible Interface 的全功能版本，它具有握手机制、独立的读写通道、 和基于突发的协议。

首先是关于握手机制。 [[5]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface) AXI 定义了一种基本的握手机制，由 **xVALID** 和 **xREADY** 信号组成。xVALID 信号由源驱动，以通知目标实体 channel 上的 payload 有效，并且可以从该 clock cycle 开始读取。同样，xREADY 信号由接收主体驱动，以通知总线已准备好接收数据。当 xVALID 和 xREADY 信号在同一 clock cycle中都为高电平时，数据 payload 被视为已传输，并且 source 可以通过保持高 xVALID 来提供新的数据 payload，或者通过置低 xVALID 来终止传输。单个数据传输，即 xVALID 和 xREADY 均为高电平时的 clock cycle，称为 “beat”。为这些信号的控制定义了两个主要规则：

（1）. 源不得等待高 xREADY 置位 xVALID。

（2）. 断开 xVALID 后，源必须保持断开，直到发生握手。

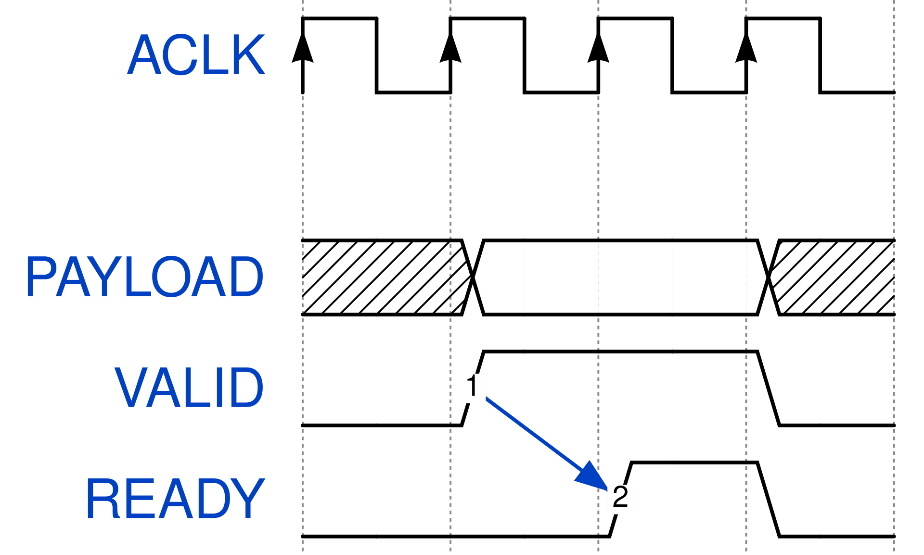
由于这种握手机制，源和目标都可以控制数据流，并在需要时限制速度。

图 22.握手机制

接下来是独立的读写通道。 在 AXI 规范中，描述了五个通道：

1. 读取地址通道 （AR）
2. 读取数据通道 （R）
3. 写入地址通道 （AW）
4. 写入数据通道 （W）
5. 写入响应通道 （B）

除了一些基本的排序规则外，每个通道都是相互独立的，并且有自己的一对 xVALID/xREADY 握手信号。[[10]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface" \l "cite_note-10)

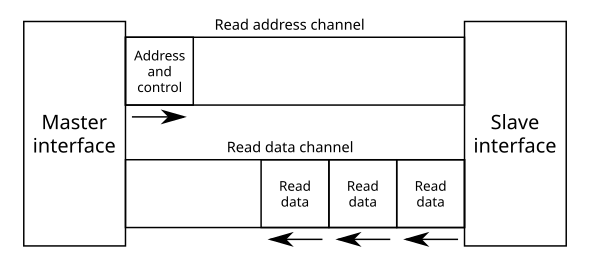
[](https://en.wikipedia.org/wiki/File:AXI_read_channels.svg)

图 23. AXI 读取地址和读取数据通道

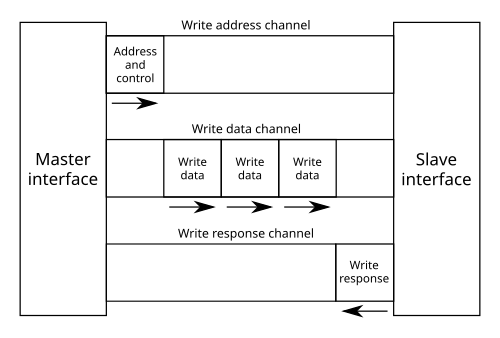
[](https://en.wikipedia.org/wiki/File:AXI_write_channels.svg)

图 24. AXI Write Address、Write Data 和 Write Response 通道

最重要的是基于 Burst 的协议，这是提高传输带宽的关键。基于突发的协议，这意味着单个请求可能有多个数据传输（或节拍）。这使得它在需要从特定地址模式读取大量数据或向特定地址模式传输大量数据的情况下非常有用。在 AXI 中，突发可以分为三种类型，由信号 ARBURST（用于读取）或 AWBURST（用于写入）选择：

1. 固定
2. INCR
3. 包装

在 FIXED bursts 中，传输中的每个节拍都有相同的地址。这对于在同一memory 位置进行重复访问非常有用，例如在读取或写入FIFO时。



另一方面，在 INCR 突发中，每个节拍的地址等于前一个节拍加上传输大小。这种突发类型通常用于读取或写入顺序内存区域。



WRAP 突发与 INCR 突发类似，因为每个传输的地址等于前一个地址加上传输大小。但是，对于 WRAP 突发，如果当前节拍的地址达到 “Higher Address boundary”，则会将其重置为 “Wrap boundary”：



跟

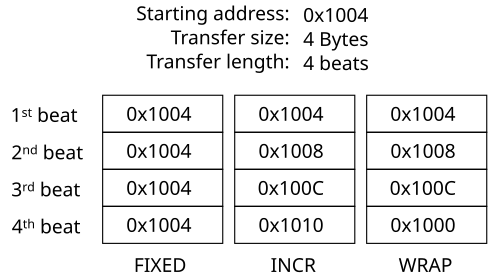


图 25.3 种不同类型的突发 传输

现在我们已经了解了 AXI\_Full 的基础知识，因此我们可以立即深入修改 Verilog HDL 代码。创建 AXI\_full 主接口的步骤与第 3 阶段创建 AXI\_Lite Slave 接口相同，唯一的区别是我们需要选择 **Full** 和 **Master**，而不是 Lite 和 Slave。

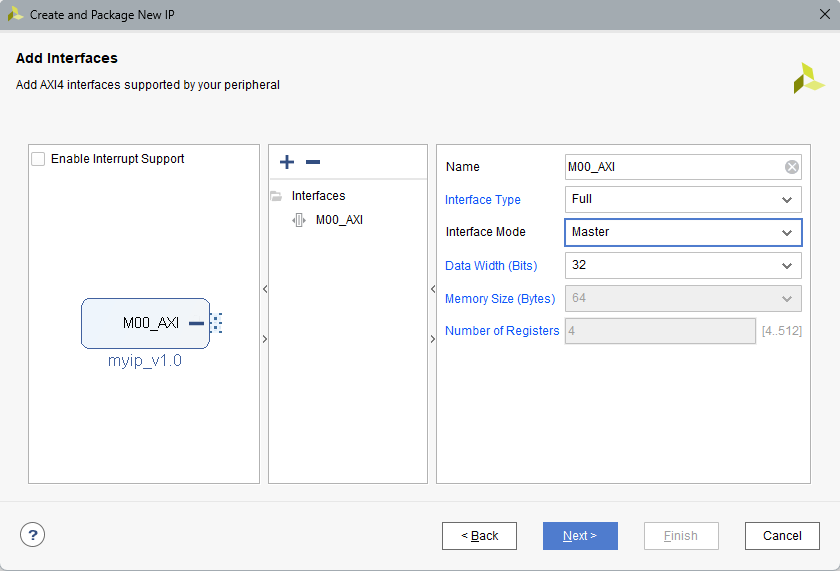


图 26.创建 AXI\_Full 主接口

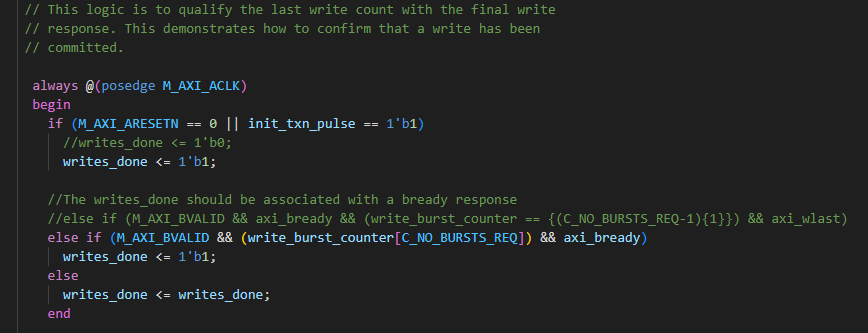
然后 Vivado 将生成 AXI\_Full Master 接口的 Verilog HDL 代码。由于在此阶段我们只需要使用 AXI 总线的读取通道，因此我们需要对生成的代码进行一些更改。首先，如图 27 所示，我们将信号writes\_done始终赋为 1 以跳过有限状态机 （FSM） 的INIT\_WRITE。

图 27.Writes\_done设置为 1

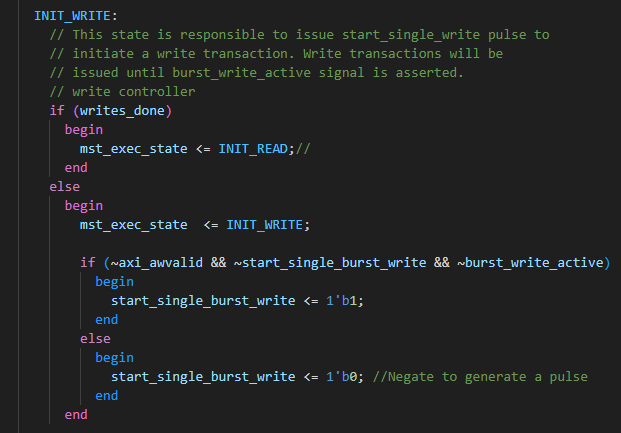


图 28.跳过 INIT\_WRITE 并输入 INIT\_READ

完成此设置后，AXI 总线现在可以在 CPU 准备就绪时重复进入突发传输。但是，burst传输应该何时、如何结束呢？现在我们将分析burst传输的机制。如下图所示，这里有一些重要的参数。



图 29.读取基址

首先是 ***C\_M\_TARGET\_SLAVE\_BASE\_ADDR***。在这里， 我们将其设置为 0x10000000，它 表示 DDR 中的特定部分。需要注意的是，编译 C 程序生成的所有内容的默认存储位置都在 DDR 中，从其基址 0x00000000 开始。所以我把图片信息的基址设置成0x10000000这样可以避免这两部分之间的冲突。

然后是***burst\_size\_bytes***。它有助于axi\_araddr在每次突发传输开始时告诉从属目标读取何处。在这里使用默认值 16 表示***C\_M\_AXI\_BURST\_LEN***，但它未能满足我们的时序要求：由于总线中不存在缓存，因此每一个burst都需要等待DDR开始读取的时间（约20个周期），需要屏幕上显示两行像素的时间才能从 DDR 中读取一行像素。发现问题后将其更改为值 64减少总burst数量，问题就解决了。



图 30.burst\_size\_bytes

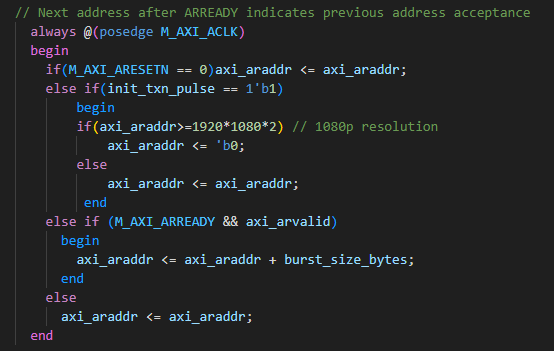


图 31.axi\_araddr

图 31 显示了 axi\_araddr 的逻辑。 这是表示基址越位的信号。因为我们使用 cacheline 作为缓冲区（我们后面会讨论这个 cache 行），所以我们需要每次读取一行，直到我们读取到最后一行最后一个像素，然后我们转到起点。因此，我们需要保留上次读取的地址，以便下次获取正确的地址，直到 axi\_addr = 1920x1080x2。其实，用 960x4x1080 写比较好，因为我们读取的数据宽度是 32 bits，也就是 4 字节，DDR 中的地址每次增加 4。由于 32 位数据包含 2 个像素的信息，因此需要读取 960 次才能从 DDR 获取一行像素。

现在我们可以看到，选择在一次突发期间将读取 64\*4字节的数据，因此我们需要 15 次突发来读取一行以填充缓存行。如图 32 和图 33 所示，我们更改了一些条件，使其在连续进行15次burst后停止。信号***read\_burst\_counter***在每次burst结束时增、加 1，直到第15个burst结束。当 ***read\_burst\_counter***达到 15 时，***reads\_done***将为 1，这表示传输结束并等待下一个传输开始的信号。

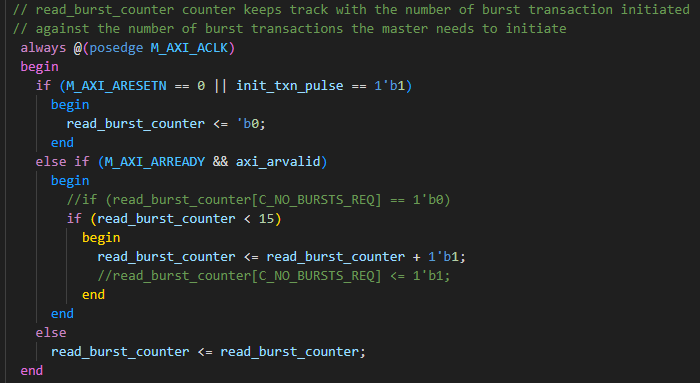
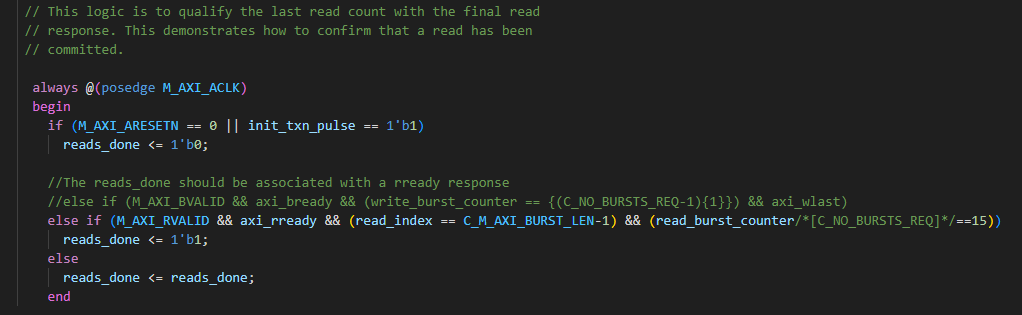
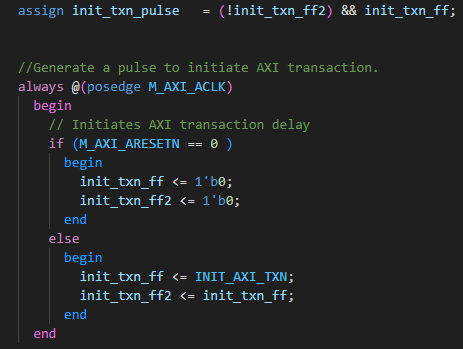
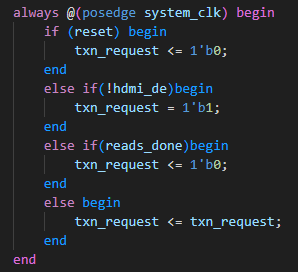


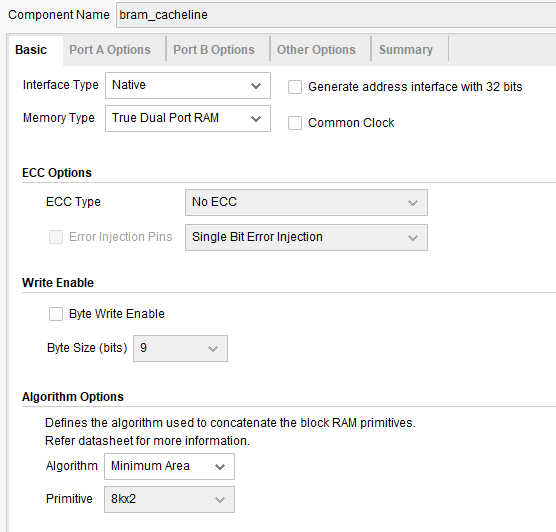
图 32.read\_burst\_counter

图 33.reads\_done

下一个问题是何时开始读入数据。信号 ***init\_txn\_pulse*** 表示传输的开始，它由 **INIT\_AXI\_TXN** 生成，如图 34 所示，这意味着我需要做的就是将INIT\_AXI\_TXN 从 0变为1 以开始传输。因此可以创建了一个名为 txn\_request 的信号作为INIT\_AXI\_TXN的输出信号，并在Block Desgin中连接他们，如图 35 所示。一旦准备好显示一行像素，txn\_request 就设置为 1 以开始传输；一旦缓存行中存储了传输入的一行像素，txn\_request 就设置为 0 以准备好下一次传输。

图 34.init\_txn\_pulse

图 35.txn\_request

我们关心的下一件事是 cacheline。缓存行是双端口 BRAM。 此 BRAM 的一侧接受来自 DDR 的数据，端口的另一侧 驱动 HDMI 控制器。此 cacheline 的大小为 32位数据宽度 \* 960 深度。cacheline 的设置如下几张图所示。

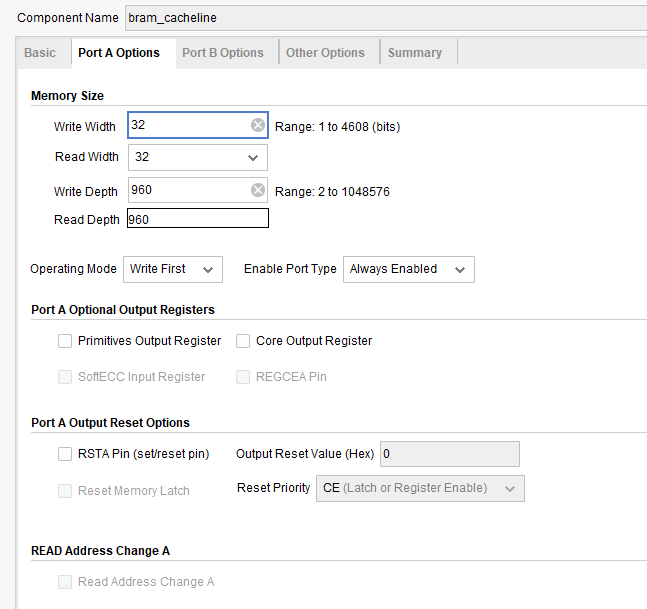
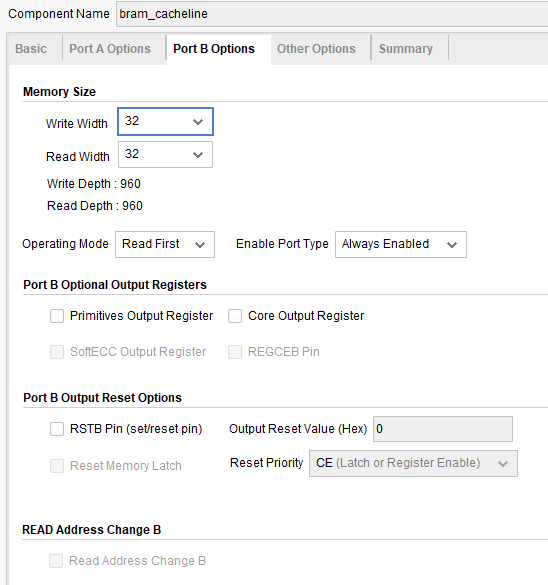
图 36.cacheline 设置 1

图 37.cacheline 设置 2

图 38.cacheline 设置 3

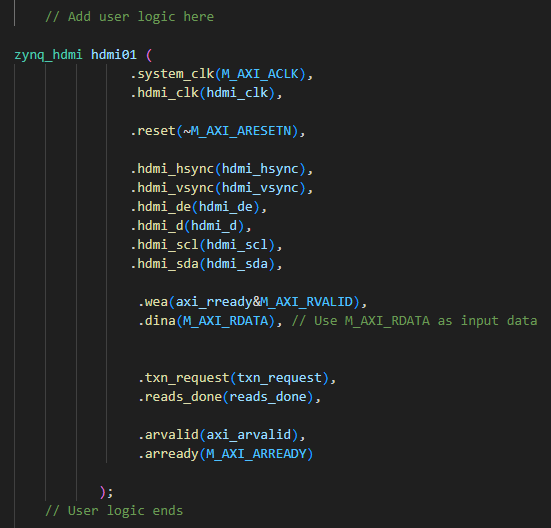
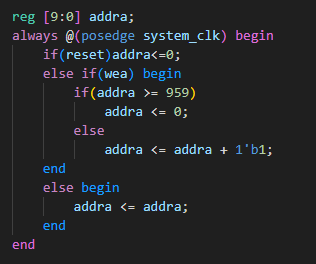
我们如何确保地址和数据是同步的，或者换句话说，我们如何确保数据存储在正确的位置？

图 39.WEA

如图 39 所示，我们使用 ***axi\_rready*** 和 ***M\_AXI\_RVALID 两个信号判断，***以确保只有在数据准备好后，才将其写入 cacheline。此外，只有当wea为1时，addra才能移动到下一个位置，如图 40 所示。

图 40.阿德拉

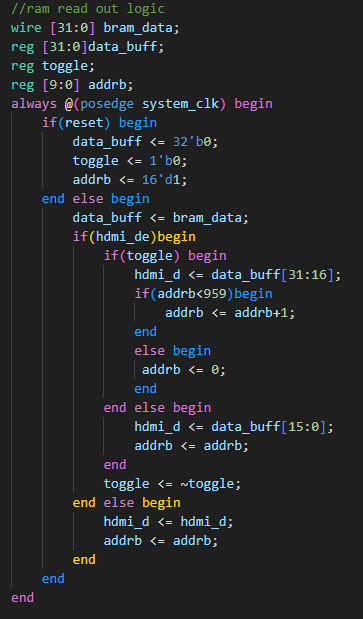
现在我们已经成功设计了cacheline 的 write port，所以我们可以继续修改cache line 的 read port。由于数据宽度为32位，这意味着我们一次将读出2个像素，因此我们需要一个寄存器来保存数据两个clock cycles，在此期间我们可以将其分成两个16位数据并输出它们。代码如图 41 所示。

图 41.Cacheline 数据输出

图 42.C 1080P飞机的源代码

电路设计测试完成后，我们现在可以专心修改C程序并测试。像素信息存储在数组***airbus[]*** 中，编译结果存储在 DDR 中，因此我们可以使用Xilinx库函数***Xil\_MemCpy()*** 将图片的整个帧复制到DDR特定位置供 HDMI 控制器读取。需要注意的是，在内存复制结束时需使用 ***Xil\_DCacheFlush()*** 来清理 CPU 缓存，否则显示屏上图片的下半部分将出现泪状水平条纹。

结果是 “./Pics/ 2\_1080\_Airbus\_Loop”，以及 另外两个必需的结果 “./Pics/ Dynamic ball” 和 “./Pics/ Dynamic Bricks 2”。

## **第 5 阶段：实时摄像机输入到 HDMI 输出**

在本节中，我们将使用 AXI\_Full Master 接口设计一个模块，该模块可用于驱动摄像头模块，接收来自摄像头模块的数据流，解码数据流并传输数据格式，以及将数据写入 DDR。然后借助我们在第 4 阶段设计的带有 AXI\_Full Master接口的 HDMI 控制器，我们可以在屏幕上实时显示相机捕获的内容。

所以首先要驱动相机。由于相机配置可能需要更改且IIC是慢速协议，我们可以使用软件驱动两个GPIO引脚，使我们能够在 IIC 协议下与相机通信。我们在这个实验中使用的相机是 IMX219。

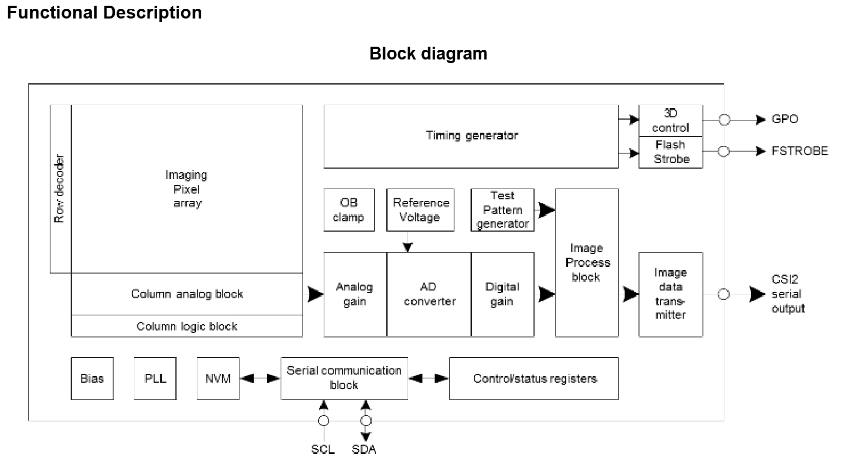


图 43.IMX219 框图

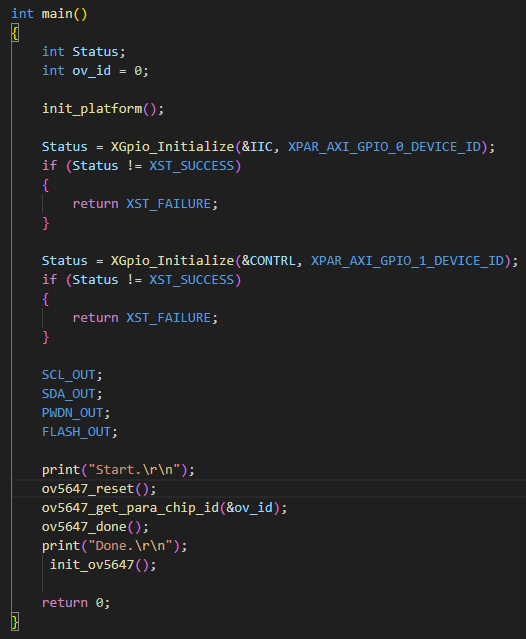
IMX219 和 Raspberry Pi Camera V2.1 的介绍存储在 “./docs” 中。我们将直接深入研究 C 源代码。我们使用 SCCB 协议**驱动**摄像头。

图 44.C SCCB 源代码

所以我在这里介绍 SCCB 协议。 [[6]](docs/SCCBSpec_AN.pdf)OmniVision Technologies， Inc. 已经定义并部署了串行摄像机控制总线 （SCCB），这是一种 3 线串行总线，用于控制 OmniVision 的 CAMERACHIPT 传感器系列中的大多数功能。在缩小的引脚封装部件中，SCCB 以改进的 2 线串行模式工作。OmniVision CAMERACHIP 传感器将仅作为从设备运行，配套的后端接口必须作为主设备进行置位。一个 SCCB 主设备可以连接到 SCCB 以控制至少一个 SCCB 从设备。可选的挂起控制信号为 SCCB 主设备提供了关闭 SCCB 系统电源的能力。

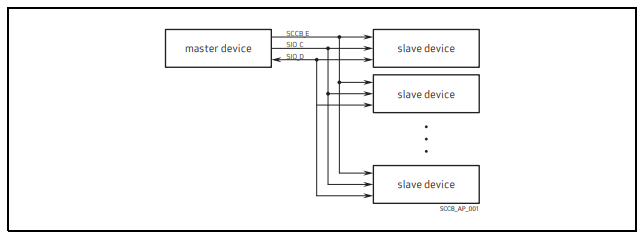


图 45.SCCB 功能框图

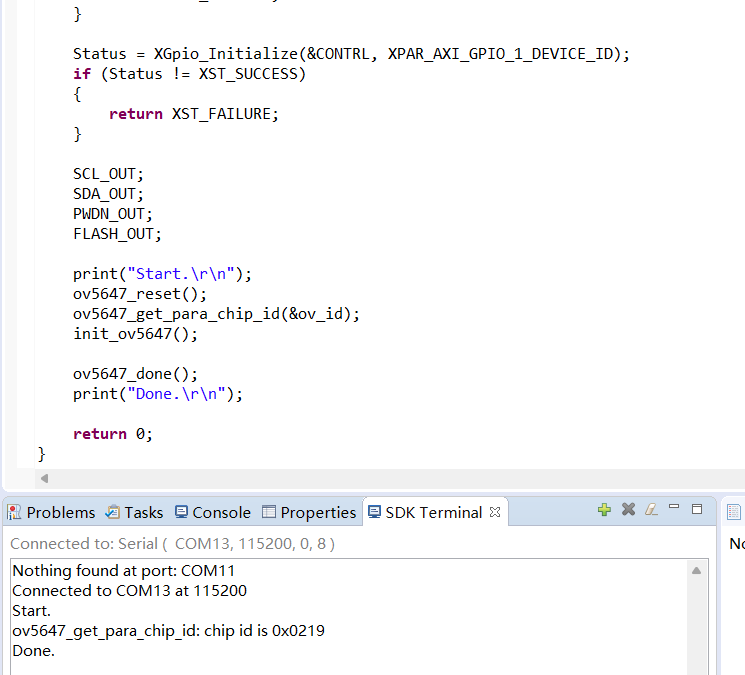
3 个基本引脚是 SDA、SCL 和 PWDN。我们打包了作这些引脚以驱动相机进入功能的过程，并在 main 函数中直接使用这些函数，如图 44 所示。有关打包函数的更多详细信息（包括最重要的register-level operations等），可以在 C Source 文件中看到。有关 SCCB 协议的更多信息，请参阅目录 “./docs” 中的文档。如果成功驱动了摄像头，则可以看到它返回了它的摄像头 ID，如图 46 所示。（注意：OV5647 不是我们使用的相机，而是IMX219。）

图 46.摄像机 ID

下一步是接收数据流并解码数据流。这指的是另一个称为 **MIPI** 协议的协议，以及 **DSI 的接口**。 [[7]](https://en.wikipedia.org/wiki/Display_Serial_Interface) **显示串行接口** （DSI） 是**移动行业处理器接口** （MIPI） 联盟制定的一项规范，旨在降低移动设备中显示控制器的成本。它通常针对 LCD 和类似的显示技术。它定义了主机、图像数据源和设备（即目标）之间的串行总线和通信协议。该接口是闭源的，这意味着该接口的规范不对公众开放。接口的维护是 MIPI 联盟的责任。只有法人（例如公司）才能成为成员。这些成员或由他们委托和批准的人员可以访问该规范，以便在其可能的应用程序中使用它。

在物理层，DSI 指定了高速（例如，D-PHY 2.0 为 4.5 Gbit/s/lane[3]）差分信号点对点串行总线。该总线包括一个高速 clock lane 和一个或多个 data lane。每个通道都由两根电线传输（由于差分信号）。所有通道都从 DSI 主机传输到 DSI 设备，除了第一个数据通道（通道 0），它能够进行总线转换 （BTA）作，允许它反转传输方向。当使用多个通道时，它们将并行用于传输数据，流中的每个连续字节都在下一个通道上传输。也就是说，如果使用 4 个通道，则同时传输 4 个字节，每个通道一个。链路在低功耗（LP）模式或高速（HS）模式下运行。在 low power 模式下，高速 clock 被禁用，信号 clocking 信息嵌入到数据中。在此模式下，数据速率不足以驱动显示器，但可用于发送配置信息和命令。高速模式使能高速时钟（频率从几十兆赫兹到一千兆赫以上），作为数据通道的位时钟。时钟速度因显示器的要求而异。由于其低电压信号和并行传输能力，高速模式仍然旨在降低功耗。

通信协议描述了两组指令。显示命令集 （DCS） 是一组用于控制显示设备的常用命令，其格式由 DSI 标准指定。它定义了可以寻址的 registers 及其作。它包括 sleep、enable 和 invert display 等基本命令。制造商命令集 （MCS） 是特定于设备的命令空间，其定义由设备制造商决定。它通常包括对非易失性存储器进行编程、设置特定器件寄存器（如 gamma 校正）或执行 DSI 标准中未描述的其他作所需的命令。这两组的数据包格式均由 DSI 标准指定。有短数据包和长数据包。短数据包长 4 字节;长数据包可以是最大 216 字节的任意长度。数据包由 DataID、字数、纠错码 （ECC）、有效负载和校验和 （CRC） 组成。需要从设备读回数据的命令会触发 BTA 事件，该事件允许设备使用请求的数据进行回复。设备无法发起传输;它只能回复主机请求。

总线上的图像数据与水平和垂直消隐间隔 （门廊） 的信号交错。数据是实时绘制显示的，而不是由设备存储的。这允许制造没有帧缓冲内存的更简单的显示设备。但是，这也意味着设备必须持续刷新（以每秒 30 或 60 帧等速度刷新），否则将丢失图像。图像数据仅在 HS 模式下发送。当处于 HS 模式时，命令在垂直消隐间隔内传输。

有关 MIPI 和 DSI 的更多详细信息，请参阅 文档 ***MIPI Alliance Specification for D-PHY*** 和 ***MIPI Alliance Specification for CSI-2*** 下的 目录 “./docs” 下。

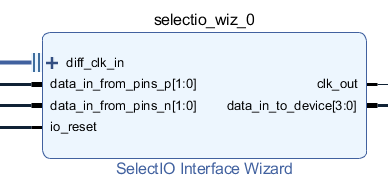
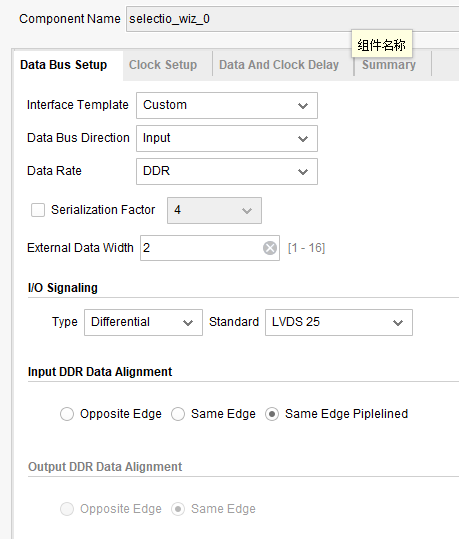
我们使用有限状态机 （FSM） 来捕获和恢复数据流中的有效数据。首先，我们使用一个名为 SelectIO Interface Wizard 的 IP 将差分信号转换为串行信号。然后我们进入 mipi 接口的设计。两个关键部分是 shift-register buffers 和 FSM。

图 47.SelectIO 接口向导



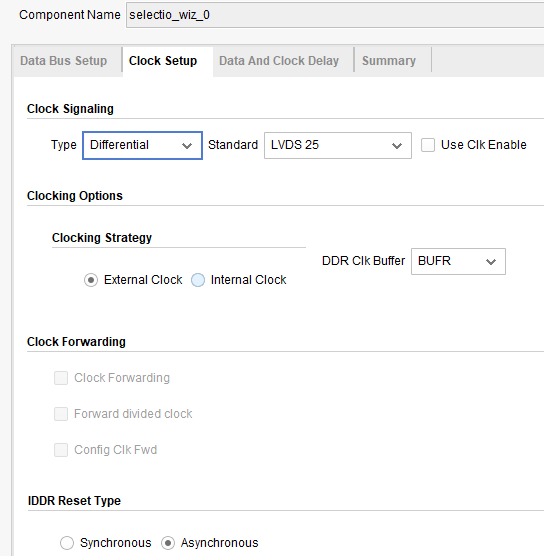
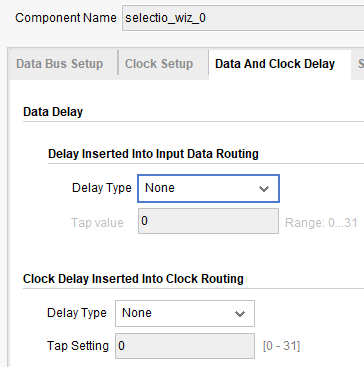
图 48.SelectIO 接口向导设置 1

图 49.SelectIO 接口向导设置 2

图 50.SelectIO 接口向导设置 3

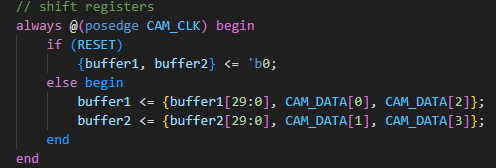
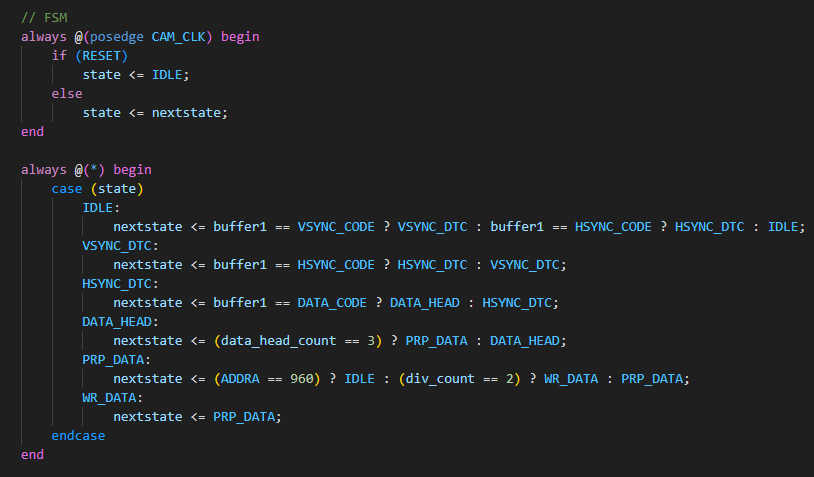
该模块使用两个32位移位寄存器 buffer1 和 buffer2来缓冲传入MIPI数据进行处理。这些寄存器收集并存储 4 位输入数据以形成 32 位字，可以与 MIPI 协议代码（例如，VSYNC\_CODE、HSYNC\_CODE、DATA\_CODE）进行比较或用于提取像素数据。因为 PCB 板的设计错误，输入数据 （CAM\_DATA\_i）需要被反转（~CAM\_DATA\_i）以输出 ***CAM\_DATA***。在 CAM\_CLK 的每个上升沿上，如果 ***RESET*** 处于活动状态，则 buffer1 和 buffer2 都被清零。否则，buffer1 通过将其当前内容 （***buffer1[29：0]）***与 ***CAM\_DATA[0]*** 和 ***CAM\_DATA[2]*** 连接起来，在 CAM\_DATA 的第 0 位和第 2 位（每个时钟周期 2 位）发生偏移。buffer2 通过将其当前内容 （***buffer2[29：0]）***与 ***CAM\_DATA[1]*** 和 ***CAM\_DATA[3***] 连接起来，在 CAM\_DATA 的第 1 位和第 3 位发生偏移.这导致 buffer1 和 buffer2 并行缓冲数据，每个缓冲区每个周期处理 4 位输入的两位，从而有效地将数据序列化为 32 位块。

图 51.Shift 寄存器缓冲区

FSM 根据 buffer1 和内部计数器的内容在状态之间转换，从而管理 MIPI 数据的解析。它由两个 always 块组成：一个用于更新当前状态，另一个用于确定下一个状态。

1. **状态定义**：
   1. **IDLE：**等待同步代码。
   2. **VSYNC\_DTC**：检测垂直同步。
   3. **HSYNC\_DTC**：检测水平同步。
   4. **DATA\_HEAD**：处理数据头。
   5. **PRP\_DATA**：准备像素数据。
   6. **WR\_DATA**：将像素数据写入输出。
2. **Next-State 逻辑**：
   1. **空闲：**
      1. 如果 buffer1 与 VSYNC\_CODE 匹配 （32'h00001D00），则转换为 VSYNC\_DTC。
      2. 如果 buffer1 与 HSYNC\_CODE 匹配 （32'h00001D40），则转换为 HSYNC\_DTC。
      3. 否则，将保持 IDLE 状态。
   2. **VSYNC\_DTC**：
      1. 如果 buffer1 与 HSYNC\_CODE 匹配，则移动到 HSYNC\_DTC;否则，保持VSYNC\_DTC。
   3. **HSYNC\_DTC**：
      1. 如果 buffer1 与 DATA\_CODE 匹配，则前进到 DATA\_HEAD （32'h00001D54）;否则，保持 HSYNC\_DTC。
   4. **DATA\_HEAD**：
      1. 当 data\_head\_count 达到 3 时过渡到 PRP\_DATA;否则，将保持DATA\_HEAD。
   5. **PRP\_DATA**：
      1. 当 div\_count 达到 2 时移动到 WR\_DATA。
      2. 如果 ADDRA 达到 960（行尾），则返回 IDLE。
      3. 否则，保持 PRP\_DATA。
   6. **WR\_DATA**：
      1. 写入数据后返回 PRP\_DATA。

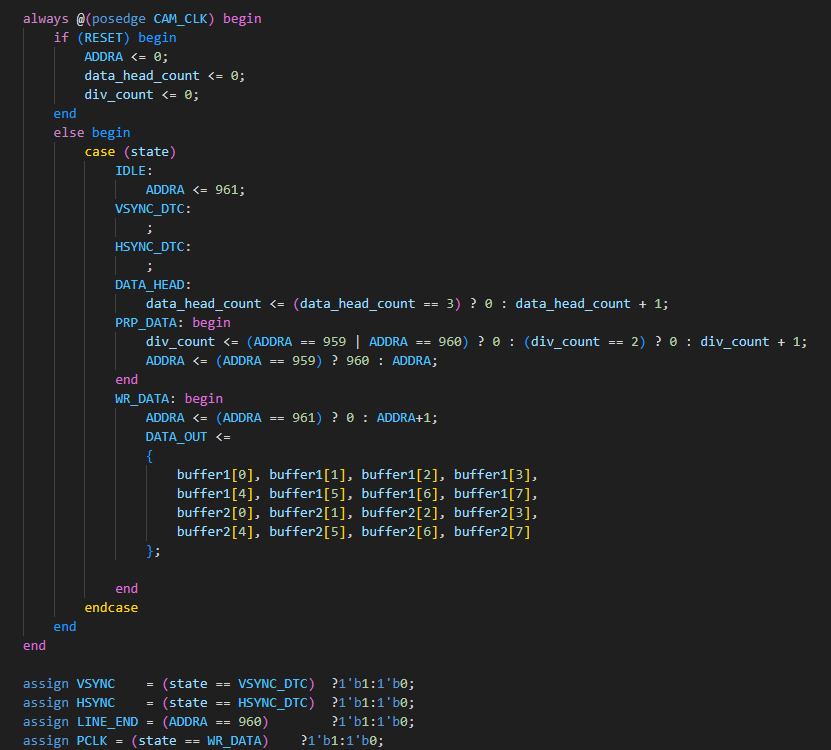
图 52.状态更改逻辑

图 53.每个状态的作

FSM 使用 buffer1 来检测同步代码，并依靠计数器（***data\_head\_count*** 和 ***div\_count***）来管理数据处理。缓冲区和 FSM 协同工作，通过将状态转换与缓冲的数据内容保持一致，确保准确解析 MIPI 数据。

但是如果使用 **ILA** 捕获 next\_state 的波形，则可能会发现 FSM 卡在 stae HSYNC 或 YSYNC。这可能是因为 external clock CAM\_CLK 缺少 od 约束。所以尝试在 xdc 文件中添加一个 clock constraint，比如：

**create\_clock -period 7.000 -name cam\_clk -waveform {0.000 3.500} [get\_pins path\_to\_your\_pin/cam\_clk]**

**set\_clock\_groups -name async\_group -asyncronous -group [get\_clocks cam\_clk] -group [get\_clocks clk\_fpga\_0]**

不幸的是，由于个人原因，我们提前很多天离开了实验室，所以我们没有时间验证它是否有效。

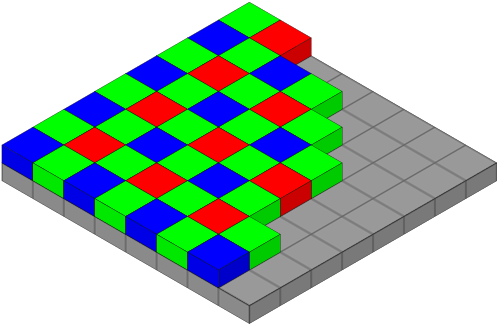
接下来是格式传输。从数据流中解析的数据是原始数据，具体来说，是 **Bayer** 格式。 [[8]](https://en.wikipedia.org/wiki/Bayer_filter) 拜耳滤光片马赛克是一种滤色器阵列 （CFA），用于将 RGB 滤色片排列在光电传感器的方形网格上。其特殊的彩色滤光片排列方式用于数码相机和摄像机中使用的大多数单芯片数字图像传感器，以创建彩色图像。滤镜图案为一半绿色、四分之一红色和四分之一蓝色，因此也称为 BGGR、RGBG、GRBG 或 RGGB。

图 54.拜耳过滤器

因此，将数据 从 Bayer 传输到 RGB 的过程称为**去马赛克**。 [[9]](https://en.wikipedia.org/wiki/Bayer_filter#Demosaicing) 可以通过不同的方式执行去马赛克。简单的方法对邻域中相同颜色的像素的颜色值进行插值。例如，一旦芯片暴露在图像中，就可以读取每个像素。带有绿色滤镜的像素提供绿色分量的精确测量值。此像素的红色和蓝色分量是从相邻像素处获得的。对于绿色像素，可以插值两个红色相邻像素以产生红色值，也可以插值两个蓝色像素以产生蓝色值。

这种简单的方法在颜色恒定或渐变平滑的区域效果很好，但它可能会导致伪影，例如在颜色或亮度突然变化的区域出现渗色，尤其是在图像中的锐利边缘处。因此，其他去马赛克方法会尝试识别高对比度边缘，并且仅沿这些边缘进行插值，而不跨这些边缘进行插值。

其他算法基于以下假设：即使在不断变化的光照条件下，图像中某个区域的颜色也相对恒定，因此颜色通道彼此高度相关。因此，首先对绿色通道进行插值，然后是红色通道，然后是蓝色通道，因此红绿和蓝绿的颜色比是恒定的。还有其他方法可以对图像内容做出不同的假设，并从此尝试开始计算缺失的颜色值。

有很多方法可以进行去马赛克。实验提供的代码使用了最简单的方法，如图 55 所示。 对于每个像素，其缺失的颜色分量直接取自最近点的相应颜色分量。

图 55.简单去马赛克

去马赛克后， 我们得到 RGB888 格式的数据。接下来是 将其转换为 YUV444 格式，然后再转换为 YUV422 格式。我们用于 RGB888 到 YUV444 的方程式为：



而 YUV444 到 YUV422 之前已经介绍过了。

实现这些转换的 Verilog HDL 代码使用纯组合逻辑。这种设计的优点是易于实现，并且没有clock cycle 延迟。但是纯组合逻辑实现 可能会导致高频率下的时序错误，尤其是对于许多的多clock-cycle操作，如乘法等。因此如果发生 timing 违规，应该添加 流水寄存器 以缩短关键逻辑路径，但需要注意调整地址与数据的对应关系，如图 57 所示。

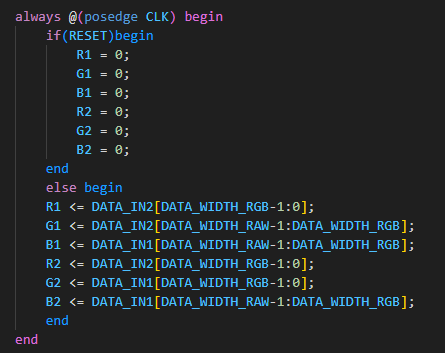


图 56.拜耳转 RGB888



图 57.RGB888 至 YUV444 （1）

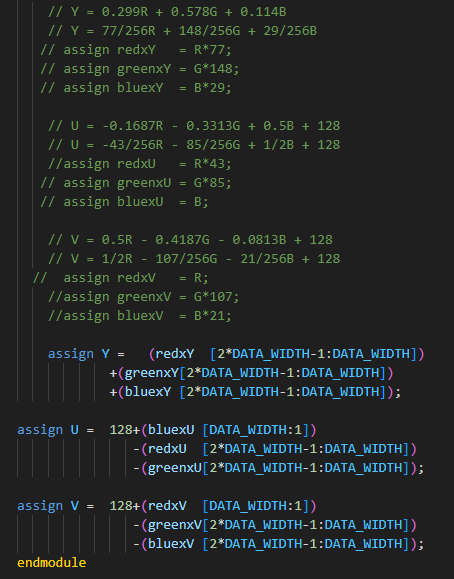


图 58.RGB888 转 YUV444 （2）

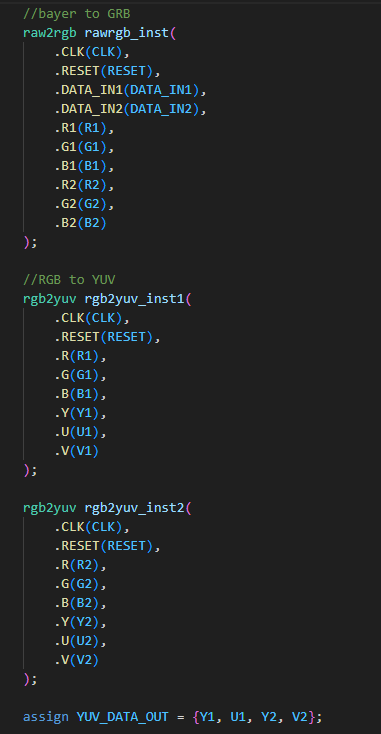


图 59.YUV444 至 YUV422

除了实验提供的代码外，我们还在网上搜索了一些更好的去马赛克和 RGB888 到 YUV444 的文件，可以在 “./codes/Phase5/ Better\_Bayer\_to\_YUV444” 中找到它们。需要注意的是，这些代码是独立的，没有被实例化，也可以在代码中进行 Gamma 校正等工作。

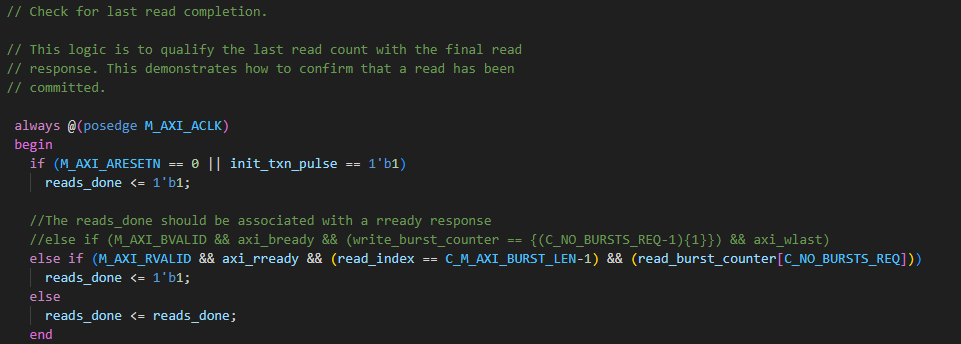
最后一件事是使用 Master 接口将数据写入 DDR AXI\_Full。我们将要做的与第 4 阶段所做的类似，只是这次我们要处理的是写入通道。所以首先，将 signal ***reads\_done*** 始终设置为 1 以关闭读取通道。

图 60.关闭读取通道

然后设置 slave target 地址、burst length、awaddr 等。这些都显示在下面的几个图中。



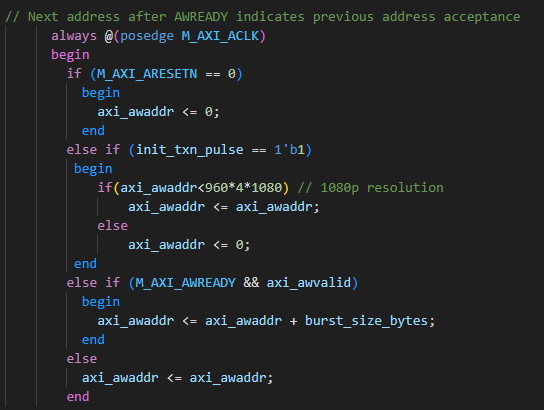
图 61.设置 Slave Tareget 地址

图 62.设置 axi\_awaddr

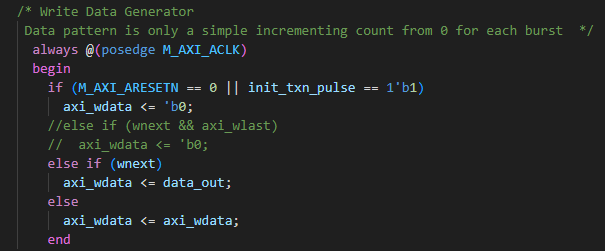


图 63.将 Signal data\_out 设置为 Output Data

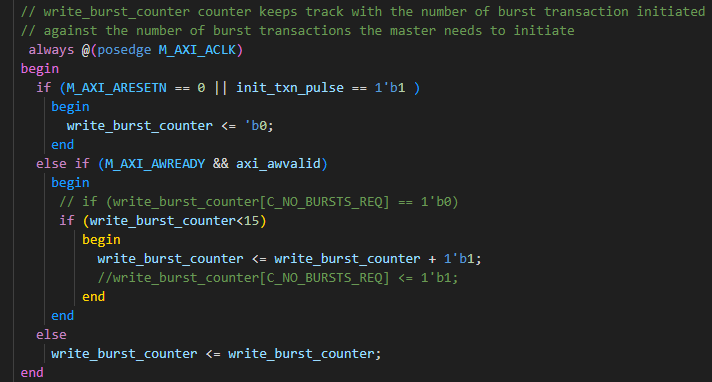


图 64.设置 write\_burst\_counter

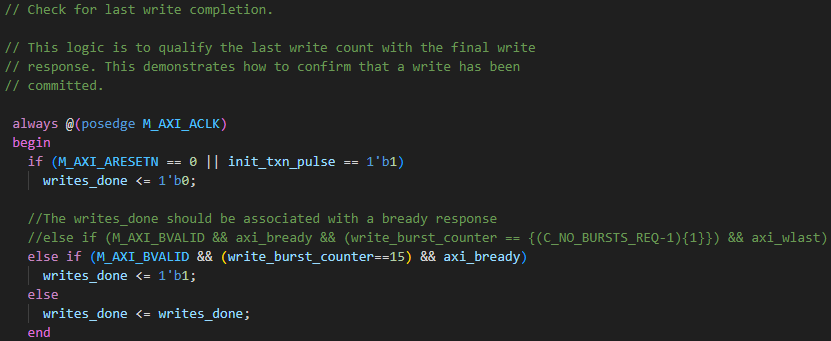


图 65.设置 writes\_done

完成这些 basic 设置后，我们需要确保如何将正确的数据传递到正确的位置。如图 66 所示，我们有两种解决方案：第一种是使用两个计数器 ***line\_count*** 和 ***pix\_count*** 来存储有关当前像素的地址信息。第二个是使用 ***wnext*** 来跟踪写入进度。我们最终选择后者，因为它更简单、健壮。

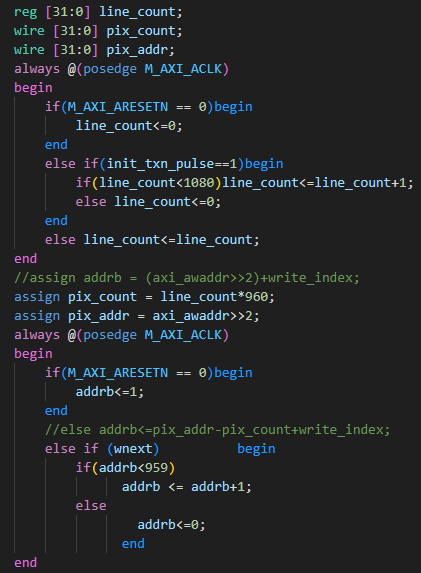


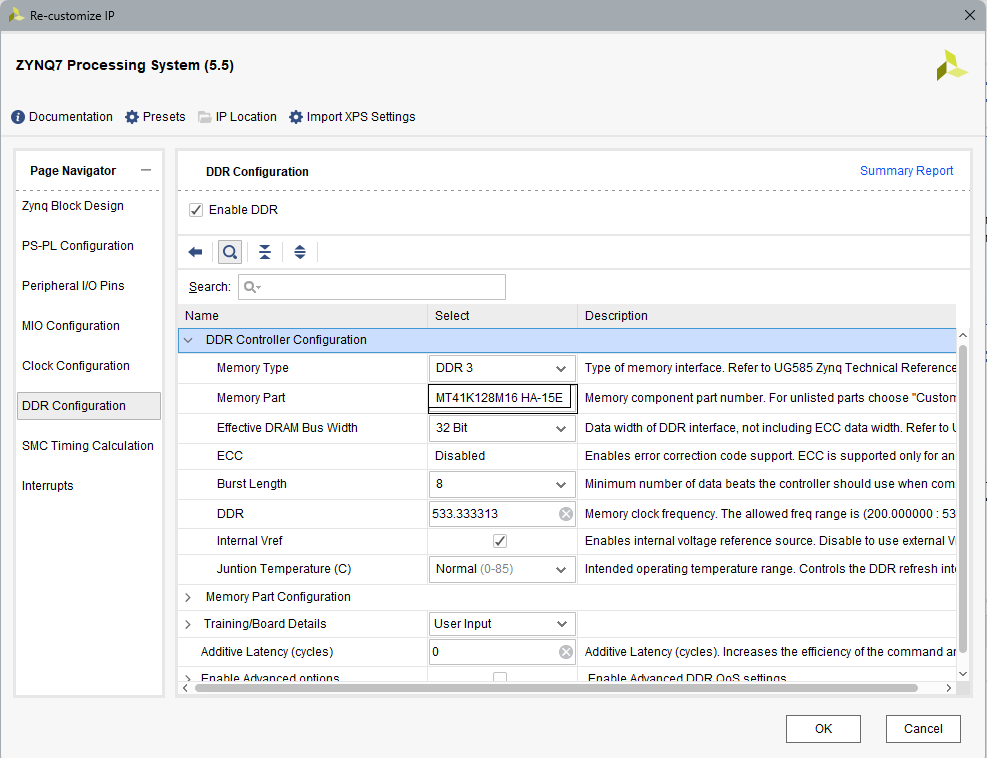
图 66.addrb 的逻辑

我们就不谈 BRAM 的输入端了，因为这个动态彩条实验的像素是经过计算的，所以无论是同步 BRAM 还是异步 BRAM，它们看起来总是一样的，你只需要将 BRAM 中的像素连续写入 DDR 即可。结果可以在 “./Pics/Master\_Colorbar” 目录中找到。

# 问题记录

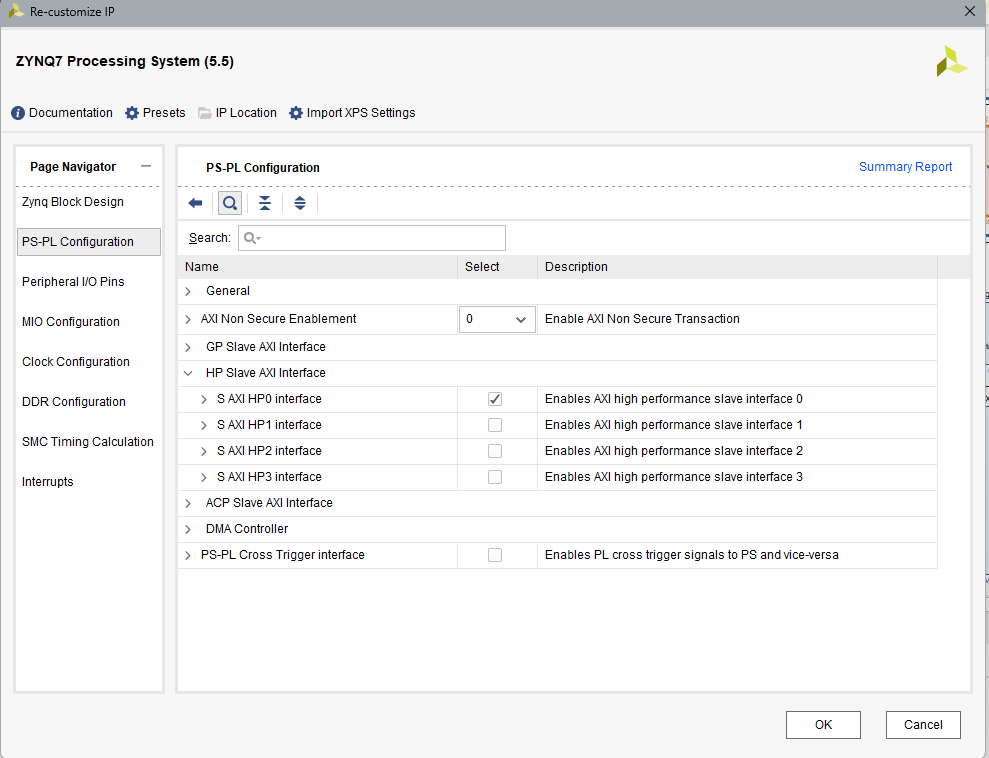
1. DDR 内存检测失败

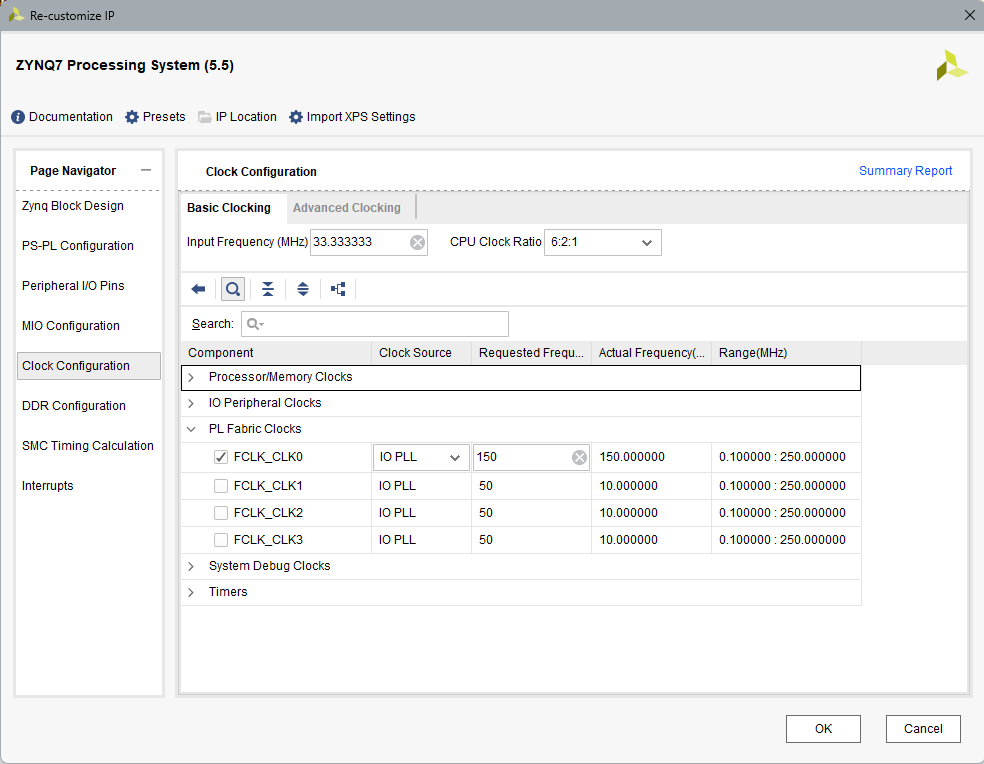
详细原因：通过软件令CPU尝试修改地址内的数据无效。

解决方法：确认 DDR 类型配置错误，ZYNQ Processing System 中应手动选择开发板实际使用的 MT41K128M16HA-15E，修改后内存检测通过。

（2） AXI 总线连接带宽不足

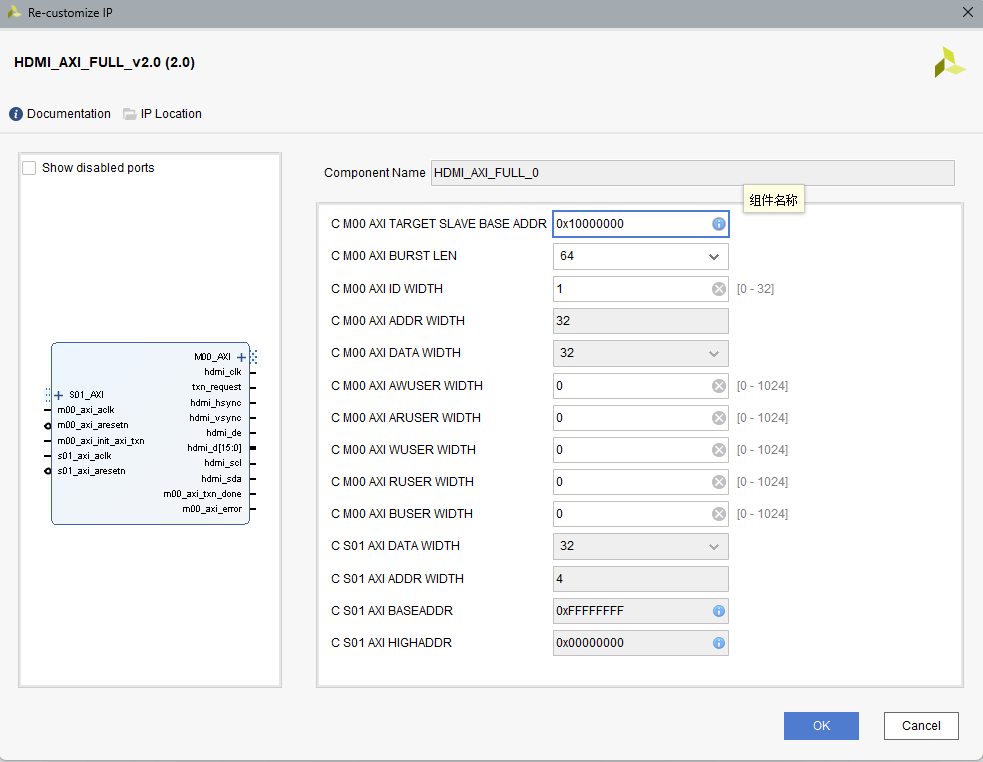
详细原因：图像输出不稳定，数据吞吐量不足。

解决方法：确保 AXI 接口连接到 HP0（High Performance 0） 端口，以获取高带宽支持 DDR 数据流动。

（3） 如果不想使用 PLL或需要双端口SRAM保持同步，则需要更改 FCLK。例如，将 FCLK 从默认的 100MHz 更改为 150MHz，作为 1080P60Hz HDMI 控制器和 AXI 总线的总时钟。

（4） DDR写入数据无法读到，烧录出现异常

详细原因：AXI 接口与软件寄存器地址不匹配

解决方法：AXI-Lite 接口使用 32 位（4 字节）地址偏移，应在硬件设计和 Xil\_Out32() 等函数中保持一致，确保地址对齐。

（5） HDMI 输出图像偏红

详细原因：可能是图像数据输出与同步信号未对齐。

解决方法：尝试将图像 SRAM 读取地址 加 1（addr + 1），对SRAM的读取时间做补偿并校准时序。

（6） 图像下半部分撕裂或残影

详细原因：显示画面未完全刷新，出现缓存数据残留。

解决方法：C 语言端在写入图像数据后调用 Xil\_DCacheFlush()，刷新缓存使数据同步至物理内存。

（7） HDMI 控制器无输出

详细问题：图像模块封装为 IP 后，板上无图像信号输出。

解决方法：如 HDMI 控制器时钟来源为 CPU 侧 FCLK，需先通过 SDK 下载并运行程序 启动 PS 时钟逻辑。

（8） 摄像头无法正常初始化

详细问题：按照 OV5647 初始化流程配置寄存器后，摄像头无响应。

解决方法：通过包装盒上树莓派的表示与购物网站的查找，确认摄像头模块型号为 IMX219而不是OV5647。需根据不同的定义文件，重新配置初始化流程。后摄像头依然无反应，查阅摄像头寄存器文档与仔细比较代码后发现最后未改变唤醒寄存器。添加额外的唤醒指令后成功。

（9） 720P 图像左侧出现两像素异常条纹

详细问题：图像左边固定区域显示异常，疑似地址错位。

解决方法：在 HDMI 控制器中增加两个中间信号用于预先计算 真实 x/y 坐标，提前一个周期完成地址生成，解决图像错位。

（10）1080P显示中Timing 报错

详细问题：Vivado Implementation 阶段 Timing Violation，图像颜色异常。

解决方法：使用 左移操作替代乘法（如 addr << 2）；添加 中间寄存器做流水线分段；使用计数器逻辑简化图像地址计算，提升系统时序稳定性。

（11）SRAM 写入的数据无效

详细问题：测试用LED 显示正常但 SRAM 无法写入

解决方法：查阅示例代码与网络资源发现AXI-Lite 接口寄存器地址单位为 32bit，访问 offset=1 实际为第二个寄存器。

# 个人总结

陈宇阳：通过参与这个实验，我获得了大量关于 AXI 总线的知识与实践经验，尤其是对其两种关键变体：AXI-Lite 和 AXI-Full 有了深入理解。起初，这些概念因其复杂性显得令人望而却步，但实验为我提供了一个结构化的学习环境，使我能够逐步掌握它们的运作原理。AXI-Lite 采用简化协议，帮助我理解轻量级数据传输的基本机制，非常适合用于控制和配置任务。相比之下，AXI-Full 则展示了基于突发传输的高性能事务能力，对于数据密集型应用尤为关键。通过动手实践，我对这两者的应用场景、信号结构和时序要求建立了清晰的认识。

本实验中最具启发性的内容之一是深入理解 AXI 框架中的主接口与从接口的作用。我学会了主设备如何发起事务并驱动通信流程，而从设备则负责响应并管理数据传输。在仿真和硬件平台上实现这些接口，使我掌握了它们在握手机制方面的细节，也意识到遵循协议规范对确保通信可靠性的至关重要性。

除了 AXI，本实验还拓宽了我对其他主流通信协议（如 HDMI、MIPI 和 SCCB）的视野。探索 HDMI 让我深入了解了高速多媒体数据传输机制，特别是在现代显示技术中的应用。MIPI 广泛应用于移动和嵌入式系统，使我认识到在带宽和能效之间取得平衡所面临的挑战。SCCB 虽然起初较为陌生，但作为相机模块配置的一种轻量级类IIC协议，其设计思路同样值得借鉴。将这些协议与 AXI 进行对比，加深了我对它们在架构设计与应用优化方面的理解。

实验中另一个让我兴奋的部分是学习 DDR内存技术。此前我几乎未接触过内存接口，这一部分对我来说全然新颖。借助项目驱动的学习方式，我探究了 DDR 的架构特点，包括其高速数据传输能力与较大的读写延迟。在实际项目中配置和优化 DDR 的交互过程充满挑战，也极具价值，因为它要求对信号完整性与时延控制格外关注。

或许这个实验带给我最重要的收获，是团队合作与协作能力的提升。在小组中共同设计、实现并调试复杂系统的过程中，我们逐步建立起责任共担、互相支持的协作氛围。面对如协议不匹配、性能优化等挑战，我们依赖清晰的沟通、合理的分工与双人协作来解决问题。这些经历让我体会到多元观点的价值与在压力下保持耐心的重要性，我相信这些软实力将在未来的合作中发挥重要作用。

总之，这次实验是一次深具影响力的学习经历。它不仅让我掌握了 AXI 总线、各类通信协议及 DDR 内存的技术知识，还帮助我锻炼了团队协作与批判性思维等核心素养。我期待将这些收获应用于未来的项目中，继续探索嵌入式系统设计的广阔天地。

邓高远: 通过本次实验，我系统地提升了在 SoC 平台上的软硬件协同开发能力，尤其在 C 语言驱动开发、接口协议调试、IMX 摄像头文档查阅与解析，以及基于波形与显示效果进行错误定位等方面积累了宝贵经验。

在多个实验模块中，我主要负责了 C 语言端的控制逻辑编写与验证，包括 AXI-Lite 控制寄存器访问、SRAM 数据写入、动态图像控制等任务。例如，在动态小球显示实验中，受限于硬件 SRAM 空间，我设计了一种软硬件协同策略：软件周期性调整小球位置坐标，硬件模块实时读取该位置并生成图像输出。通过定时器设定 16ms 左右的刷新周期，保证了动态效果与显示稳定性之间的平衡。

实验过程中，我经历了大量的软硬件联调任务。在 HDMI IP 封装为 AXI-Lite 外设时，最初显示屏无信号输出，排查发现需要先运行 PS 部分以启动 AXI 时钟。这一问题让我更加理解 ZYNQ 架构中硬核与软核之间的依赖关系，并通过调试逐步掌握其启动流程与时序约束。在另一个实验中，由于误用了 AXI 寄存器偏移单位，导致写入值未能正确传达至图像缓存，最终通过代码逐句对比和 ILA 波形抓取找到了问题根源。

我还承担了摄像头相关的部分工作，查阅了IMX219 芯片文档，理解了 SCCB 协议的细节与与标准 I²C 的差异。我基于 GPIO 接口实现了 SCCB 模拟协议的信号控制逻辑，最终分析找到了示例代码中没有wakeup的错误，成功验证了摄像头寄存器配置流程的正确性。

在 HDMI 图像输出与 DDR 交互实验中，我进一步强化了对时序和资源约束的理解。为排查显示图像位置异常、花屏等问题，我采用了“从显示效果倒推代码逻辑”的方式，结合仿真波形与现场逻辑分析仪（ILA）抓取的实际信号，精确定位到如地址打拍、乘法器延迟等细节问题。尤其在 1080P 图像处理实验中，通过使用移位替代乘法器、添加寄存器缓冲、改进查找表等手段，我解决了多个 timing violation 隐患，提升了图像处理的稳定性。

总体来看，本次实验显著提升了我的工程实践能力。它不仅让我深入掌握了 AXI 总线协议、嵌入式显示控制、摄像头通信等底层知识，还让我在波形分析、问题定位、软硬件联动调试等环节得到了综合训练。我更加理解了“系统设计”不仅仅是模块拼接，更关键的是对时序、资源、流程的整体把控。未来我希望将这些经验应用于更复杂的嵌入式系统开发中，不断提升自己的系统设计能力。

# 参考文献

[1] 维基百科， <https://en.wikipedia.org/wiki/HDMI>

[2] 维基百科， <https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#AXI4-Lite>

[3]Xilinx， <https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Lite-Interface-Signals>

[4] 维基百科， <https://en.wikipedia.org/wiki/Y%E2%80%B2UV>

[5] 维基百科， <https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface>

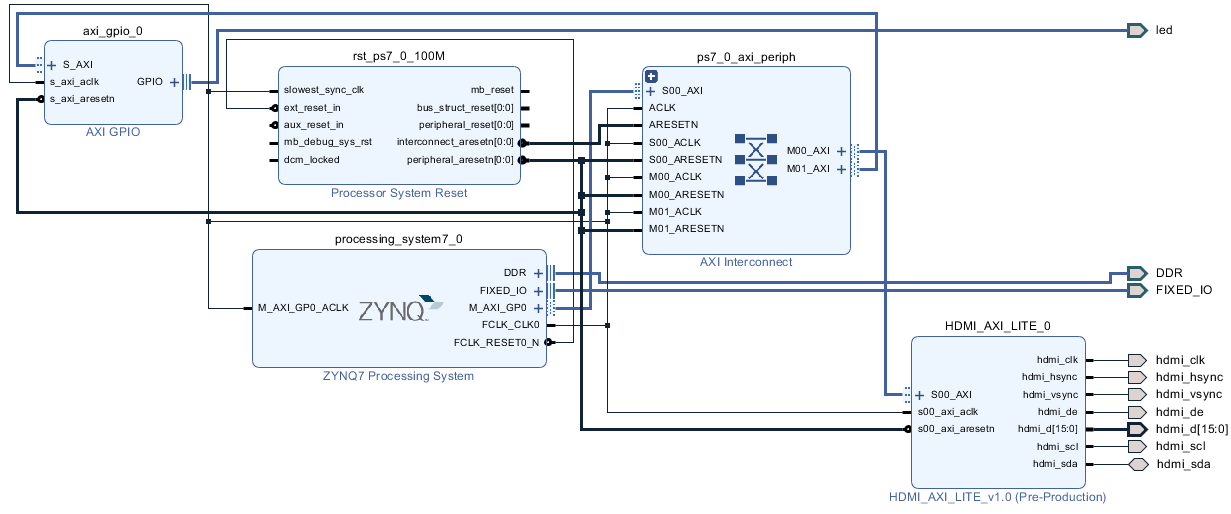
[6] [SCCBSpec\_AN.pdf](docs/SCCBSpec_AN.pdf)

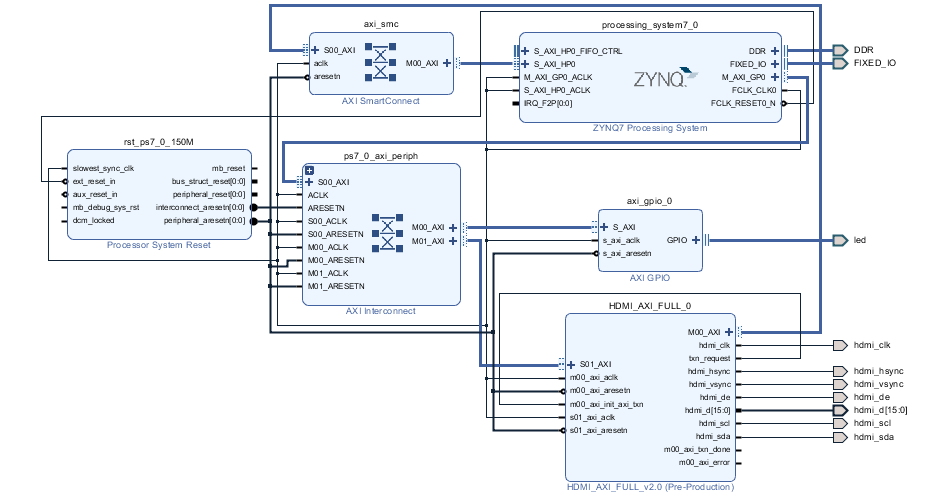
[7] 维基百科， <https://en.wikipedia.org/wiki/Display_Serial_Interface>

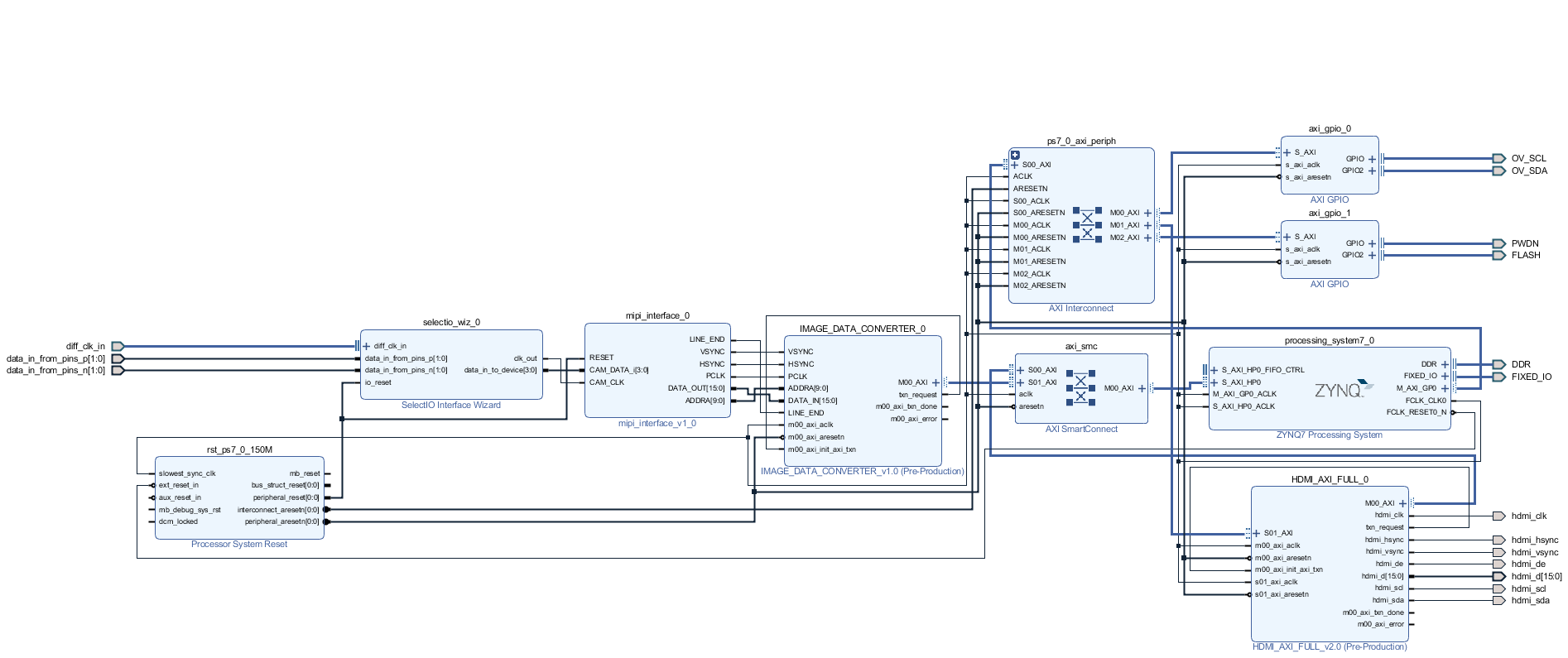
[8] 维基百科， <https://en.wikipedia.org/wiki/Bayer_filter>

[9] 维基百科， <https://en.wikipedia.org/wiki/Bayer_filter#Demosaicing>

# 附录

（1）  **第三阶段的**区块设计

（2）  **第 4 期**的区块设计

（3） Phase 5  **的 Block Design** for Camera 项目

（4）  **第 5 期彩条项目的**块设计

