

1 基于 ACM_VGAHDMI 的 IO_HDMI/DVI 显示

工程源码	---02_设计实例 ----- acz702_hdmi_color_bar_io_hdmi.zip
相关视频课程	
	为了方便区分，在工程命名时，我们使用 ic_hdmi 指代使用 IC 芯片实现 HDMI 显示的设计；用 io_hdmi 指代通过逻辑设计，使用 IO 接口实现 HDMI 显示的设计。

章节导读

上一章中，我们带领大家完成了基于 SiI9022 的 HDMI 彩条显示设计，但是本质上，我们只是使用 IIC 总线对 SiI9022 芯片进行了配置。至于输入的 RGB 数据和时序信号如何转换并编码成 HDMI 所需格式，完全由芯片自动完成。对于用户而言，不需要也不用关心这一过程。虽然这种方式能极大简化 HDMI 设计难度，但却是十分不利于学习的。同时，对于一些想要自己设计电路的用户来说，专用的 RGB 转 HDMI 芯片无疑会增加预算成本。

为此，本章我们将带领大家学习 HDMI/DVI 的数据链路原理，并基于该原理，带领大家使用逻辑资源实现 HDMI 接口，进而实现 720P 的 HDMI 彩条显示设计。

1.1 基于 FPGA 的 DVI 电路设计

上一章中，我们通过 SiI9022 实现了 HDMI 的彩条显示。本质上，是将 FPGA 输出的 24 位像素数据+3 位控制信号（HSYNC、VSYNC、DE）接入到 SiI9022 中，然后由 SiI9022 芯片完成数据的编码和串行发送。

在上述的设计方案中，仅与视频数据传输相关的信号就高达二十多个。这在一些 FPGA IO 相对紧张的应用中，属于较大的浪费了。

为了在节约 IO 资源的同时实现 HDMI 发送，就有 FPGA 厂家和开发者采用 FPGA 实现 HDMI 发送所需的 TMDS 编码和串行发送器。这种设计方案在 FPGA 内部实现了 HDMI 发送芯片的核心功能，并最终直接使用 FPGA 管脚输出符合 HDMI 协议规范的 HDMI 链路信号。使用这种方案不仅能降低对 FPGA 管脚资源的占用，还能简化电路设计，降低硬件成本。事实上，即使使用低端的 FPGA 器件，也能实现高达 720P 的图像传输，而使用更加高性能的 FPGA 芯片，或者带高速收发器的 FPGA 芯片，能够传输的图像尺寸将更大。所以，掌握使用 FPGA 编程实现 HDMI 接口的方法，不仅实用，而且经济划算。下图为使用 FPGA 直接编程实现 HDMI 接口的示意图：

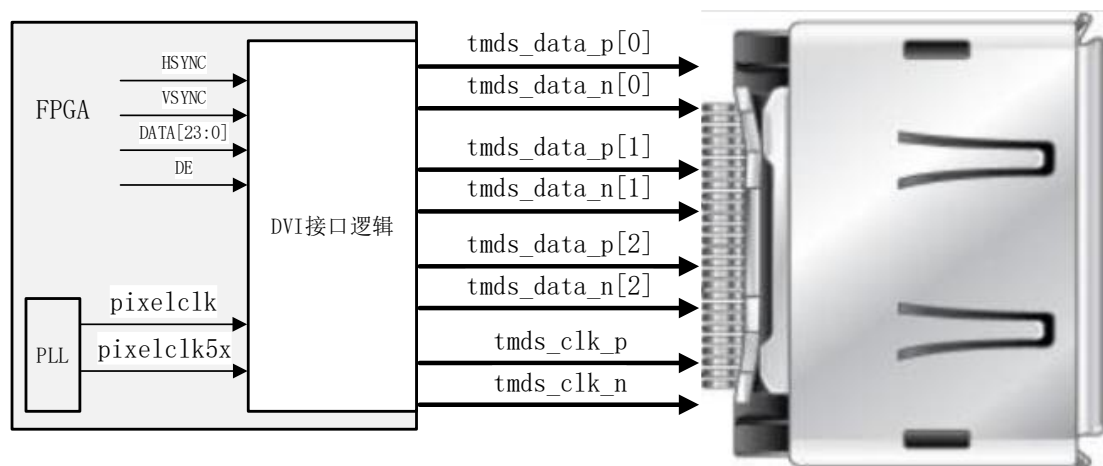


图 1-1 使用 FPGA 直接编程实现 HDMI 接口示意图

ACZ702 开发板上并未设计该电路，因此本章需要使用到 ACM_VGAHDMI 模块。该模块上集成了 VGA 和 HDMI 发送接口电路，其中 HDMI 发送接口电路如图 1-2 所示：

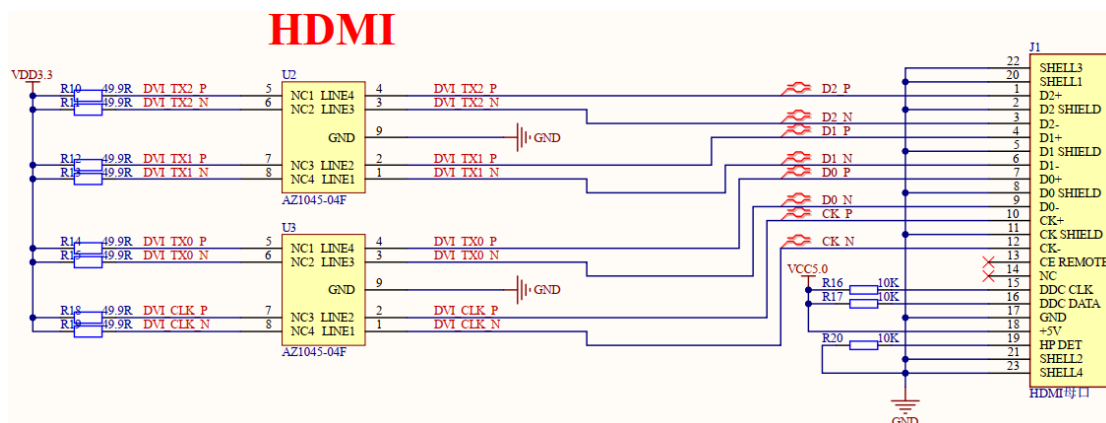


图 1-2 ACM_VGAHDMI 原理图

其中 AZ1045 芯片为 TVS 器件，保护线路，防止 HDMI 上差分线之间的压差过高。每条链路上都使用了一个电容用来隔断直流电压，电路精简，成本低廉，却能达到理想的传输效果。

既然有了硬件电路，即物理层已经就绪，剩下的就是 FPGA 中的编码了，该部分内容涉及的知识点就相对来说复杂很多了，以下将花重点篇幅介绍。

1.2 HDMI 与 DVI 的区别与联系

1.2.1 DVI 接口含义

DVI (Digital Visual Interface)，即数字视频接口，是一种视频接口标准，设

计的目的是用来传输未经压缩的数字化视频。目前广泛应用于 LCD、数字投影机显示设备上。此标准由显示业界数家领导厂商所组成的论坛：“数字显示工作小组”（Digital Display Working Group, DDWG）制订。DVI 接口可以发送未压缩的数字视频数据到显示设备。本规格部分兼容于 HDMI 标准。

1.2.2 HDMI 接口含义

高清晰度多媒体接口（High Definition Multimedia Interface）是一种全数位化影像和声音传送接口，可以传送无压缩的音频信号及视频信号。HDMI 接口可以提供高达 5Gbps 的数据传输带宽，可以传送无压缩的音频信号及高分辨率视频信号。同时无需在信号传送前进行数/模或者模/数转换，可以保证最高质量的影音信号传送。

1.2.3 HDMI 与 DVI 的区别

- 接口外观不同，下图中，左侧为 HDMI 接口，右侧为 DVI 接口，可以看得出 HDMI 接口体积比 DVI 小。
- HDMI 可以同时传输数字视频和音频信号，用一根电缆就可以了，DVI 只能传输数字视频信号，传输音频信号只能用另外的接口和电缆。
- HDMI 最高传输速率比 DVI 高，HDMI 可以传输更高清数字视频信号。
- 在保证不失真前提下 HDMI 传输距离比 DVI 远，HDMI 电缆最长可以达到 15 米，DVI 只能达到 8 米。



图 1-3 DVI 接口和 HDMI 接口

1.2.4 HDMI 与 DVI 的兼容性

HDMI 兼容 DVI，可以通过转换接口将视频信号接到 DVI 接口上。使用 DVI 标准发送的视频数据可以直接接入 HDMI 接口的 LCD 显示器正常显示。同样的，使用 HDMI 标准发送的视频数据在符合 DVI 规范的范围内，也能直接接到 DVI 接口的 LCD 显示器上正常显示。

在本节中讨论的使用 FPGA 实现 HDMI 发送的功能，实际只能算 DVI，因为我们没有加入 HDMI 所拥有的音频数据传输功能。而且受 FPGA 的 IO 口翻转速度限制，也仅能实现 DVI 规范所规定的传输速率，达不到 HDMI 规范的高速传输能力。但是为了接口易用，我们在硬件上使用了 HDMI 的连接接口。在下述描述内容中，将不再刻意区分 DVI 和 HDMI，介绍协议时多以 DVI 描述，介绍接口时则多以 HDMI 描述。

1.2.5 HDMI 与 DVI 接口对比

在 DVI 甚至 HDMI 接口出现之前，应用很广泛的一直是 VGA 接口，早期的 CRT 显示器多以 VGA 接口与计算机连接，哪怕是到了现在，还有不少的计算机设备将 VGA 接口作为标配功能。

VGA 接口传输的是模拟信号，通过模拟电压的变化来表示像素颜色。这种传输方式适合在早期的 CRT 显示设备中使用，因为 CRT 显示器的成像原理本身就是模拟信号的放大，变换等。通过 VGA 传输的模拟信号送入显示器后经过一系列的放大后可以直接驱动 CRT 显示器的电子枪扫描荧光屏。

而现代 LCD 液晶显示屏则采用液晶材质在会根据加载其两端的电压大小而改变透光性的特性实现的被动颜色显示，在液晶显示器中，通过将众多细小的液晶颗粒按照矩阵的形式排布在一起，实现显示面板。所以 LCD 液晶显示屏是以像素作为基本的成像单元，通过让像素点显示不同的颜色来实现彩色图案的显示。

一个 LCD 显示屏的物理像素数量是确定的，每个像素点的颜色都可以对应一个图像数据，而这个数据是数字信号，所以 LCD 显示屏从原理上讲可以认为是数字显示屏（虽然最终控制单颗液晶的透光程度时也会将这个数字信号转换为模拟电压信号，但是这已经是像素点级别的成像原理了，而非液晶显示器显示整幅图案的成像原理）。显示时，只需要得到对应像素点的颜色数据即可，所以这类显示屏接收的是数字信号。

下图为 LCD 液晶显示器分别使用 VGA 接口和 HDMI 接口与显卡传输图像

数据的数据流变换示意图。左图为 VGA 接口，右图为 HDMI 接口。

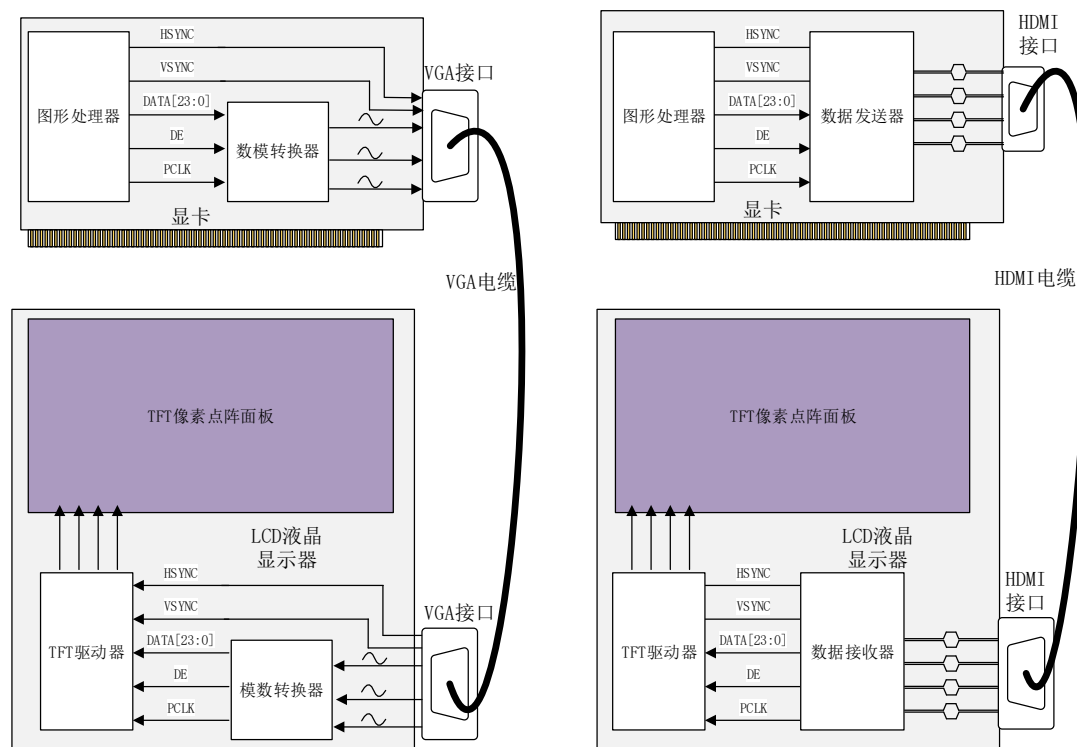


图 1-4 VGA 和 HDMI 接口与显卡传输图像数据流变换图

在使用 VGA 接口的传输方式中，图像数据从图形处理器（也就是常说的显卡芯片）输出（此时的信号为数字信号），为了能够变成 VGA 接口传输所需的模拟信号，会使用一片数模转换器转换为模拟信号后再输出到 VGA 接口上。然后模拟信号通过 VGA 电缆传输到 LCD 液晶显示屏的 VGA 接口，由于 TFT 像素面板为数字显示屏，每一个像素的颜色是由一个 TFT 驱动器实时的扫描描绘的。TFT 驱动器的输入要求为数字信号，所以 TFT 驱动器和 VGA 接口之间使用了一片模数转换器将 VGA 接口上的模拟信号转换为数字信号后再送给 TFT 驱动器使用。整个信号的变换流程为：

数字信号 -> 模拟信号 -> 模拟信号 -> 数字信号

在使用 HDMI 接口的传输方式中，图像数据从图形处理器（也就是常说的显卡芯片）输出（此时的信号为数字信号），为了能够将数字信号转换为 HDMI 标准的高速差分信号，会经过一个数据发送器。该数据发送器仅仅是改变了数据的传输方式，将原本的并行数据转换为高速串行数据输出，以符合 HDMI 协议标准，在 HDMI 接口上传输的内容实际还是数字信号。数字信号从显卡的 HDMI 接口经由 HDMI 线缆传输到 LCD 显示器的 HDMI 接口，再由 LCD 显示器内的数据接收器将 HDMI 接口上的高速串行数字信号转换为并行数据，提供

给 TFT 驱动器使用。整个信号的变换流程为：

数字信号 -> 数字信号 -> 数字信号 -> 数字信号

使用 HDMI 方式传输的过程中，数据的内容没有发生任何的变化。图形处理器发送的是什么数据，最终送到 TFT 驱动器的就是什么数据，不会有哪怕一位数据发生变化。所以图像还原度很高，而且由于 HDMI 传输方式为差分传输，其抵抗干扰的能力也比模拟信号高很多，一般不会受到干扰。

使用 VGA 方式传输数据，由于数据在传输过程中经过了两次数字和模拟信号间的变换，所以无法保证图形处理器发送的数据就一定是 TFT 驱动器收到的数据，中间总会有一定的差异，虽然这些差异可能不会对最终的显示效果产生明显的影响。但是由于数模转换和模数转换的存在，当数据的变化速度过快时，可能造成拖影现象，所谓拖影现象，就是指在图像颜色变化较大的边界，后一个像素的图像数据可能无法真正显示其想显示的颜色，而是介于需要显示的颜色和前一个像素的颜色之间的一种颜色。而且，模拟信号在传输过程中容易受到噪声的干扰。导致显示的图像中会出现很多杂点。

综上所述，使用 HDMI 传输图像具有还原度高，抗干扰能力强，数据带宽大的优势，目前新出产的计算机和显示器已经都标配 HDMI 接口了。就笔者使用体验来说，使用 VGA 接口传输图像内容时，受不同显示器和显卡的影响，很容易就会遇到拖影和噪声干扰的问题，而 HDMI 接口的则都能保证图像质量。

1.3 DVI 数据链路介绍

无论是 HDMI 还是 DVI 规范，其数据链路层都是使用 TMDS 编码方式。在数据传输过程中，包括了输入接口层、TMDS 发送器、TMDS 接收器和输出接口层。

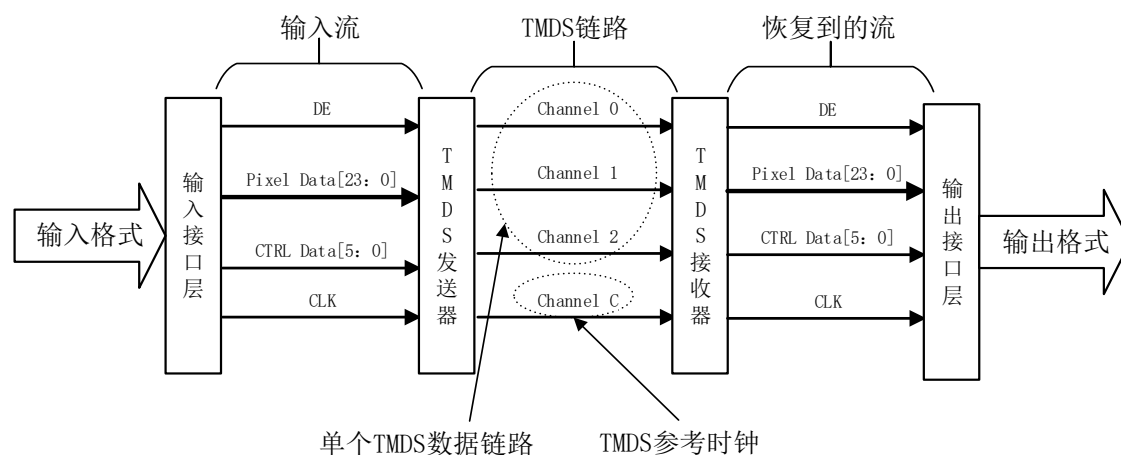


图 1-5 DVI 数据链路

1.3.1 输入接口层

输入接口层的信号格式为典型的 RGB 行场同步数字接口，该接口包括数据使能信号 DE（Data Enable）、24 位像素数据（Pixel Data）、6 位的控制数据（包括 HSYNC、VSYNC 和空信号）和同步时钟信号。

1.3.2 TMDS 发送器

TMDS 发送器完成对输入接口层的数据和控制信号按照 TMDS 编码方式进行编码，再将编码的数据通过高速串行接口输出，最终将输入接口层的信号编码进 4 个 TMDS 链路层中。关于输入接口层的信号到最终 TMDS 链路层的编码和串行化过程，将在下一节介绍。

1.3.3 TMDS 接收器

与 TMDS 发送器相反，TMDS 接收器的功能是将 TMDS 链路上的高速串行数据接收，解串，TMDS 解码，得到与输入接口层相同的控制信号和数据。

1.3.4 输出接口层

输出接口层将 TMDS 接收器解码得到的数据流和控制信号传递给最终的数据消费者，例如 RGB 接口的液晶显示面板。

1.4 TMDS 原理与实现

在上面介绍 DVI 数据链路层的时候，提到了 TMDS 发送器是将数据和控制信号进行编码并串行化后发送，那么什么是 TMDS 编码呢，TMDS 编码又是怎样的一种编码方式呢？

首先来说，TMDS 编码包含两个大的内容，传输控制信号的控制段和用来传输图像像素数据的数据段。一个完整的图像传输 TMDS 模块包含三个相同的编码和发送模块。每个发送模块包含 8 位的像素数据，对应 24 位像素数据中单个颜色的 8 位数据。2 个控制信号，这两个控制信号可以分别接行同步（HSYNC）、场同步（VSYNC）信号，也可以空置接 0。另外还有一个数据有效信号 DE，该信号用来区分控制数据和像素数据。当 DE 信号为高电平的时候，表明当前数据有效，编码器对 8 位的 Data 数据进行编码。当 DE 信号为低电平的时候，则对 2 位的控制信号进行编码。

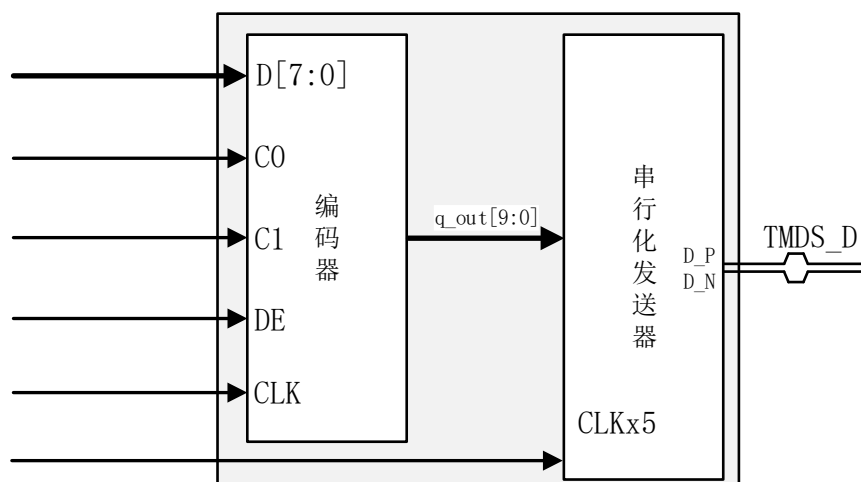


图 1-6 TMDs 原理与实现输入输出示意图

这里，串行发送器使用的时钟信号频率为编码器的 5 倍。至于为什么是 5 倍，这是因为串行发送器在发送数据时候是采用双数据速率的方式传输数据的，一个时钟周期可以传输 2 位信号，所以只需要 5 倍的编码器的时钟即可完成 10 位数据的及时传输。关于串行化发送器的详细发送原理，在本节内容的后续部分有详细介绍。

下图为使用 TMDs 方式传输视频数据的示意图，可以看到，从编码（Encoder）的角度，TMDs 发送器将图像数据和控制信号分成了 3 组，每组使用一个编码器加串行发送器。每个编码器对 8 位的图像数据和 2 位的控制信号组成。另外还对串行发送时钟进行了输出，作为 3 个数据/控制通道的同步时钟。

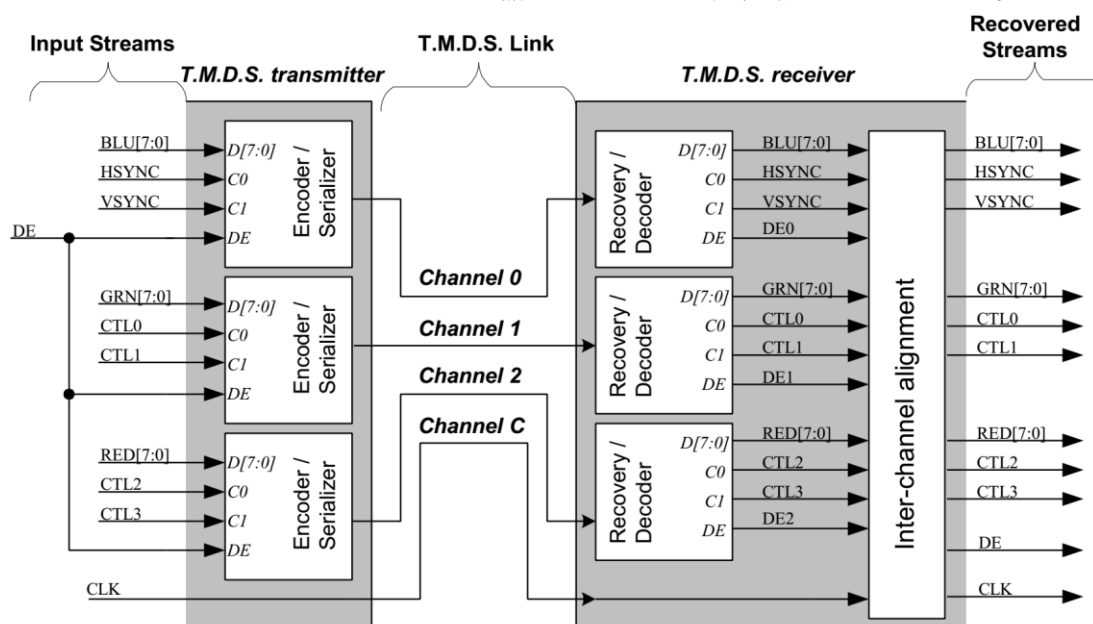


图 1-7 TMDs 传输通道编码解码概况图

第一组中，将像素数据中的蓝色（BLUE）分量和控制信号中的行同步（HSYNC）、场同步（VSYNC）划分成了一组，经过编码器和串行调制器后输出，名为 Channel 0。

第二组中，将像素数据中的绿色（GREEN）分量和两个空的控制信号 CTL0、CTL1 划分成了一组，经过编码器和串行调制器后输出，名为 Channel 1。在这一组中，CTL0 和 CTL1 接入的是空信号。

第三组中，将像素数据中的红色（RED）分量和两个空的控制信号 CTL2、CTL3 划分成了一组，经过编码器和串行调制器后输出，名为 Channel 2。在这一组中，CTL2 和 CTL3 接入的是空信号。

至此，关于 TMDS 对数据的划分方式就非常的清晰了。设计的重点就是编码器和串行化数据发送器的设计。

1.5 TMDS 最小化传输编码原理

什么是最小化传输，为什么要使用最小化传输呢？和各位一样，笔者在第一次听到这个概念的时候，首先头脑中冒出的也是这个疑问。

所谓最小化传输，其本质就是要通过对输入数据的变换，得到一个跳变次数最少的新数据。什么是跳变次数最少呢？举个例子。8 位的数据 0x55，其二进制值为 01010101b，这个数据中，相邻的两个数如果个数不一样，则认为是一次跳变。从最低位看起，bit[0] = 1，bit[1] = 0，0 和 1 的状态不同，则为一次跳变。再接着看，bit[1] = 0，bit[2] = 1，1 和 0 的状态又不同，所以又是一次跳变。对于 0x55，其 8 位数据中，任意相邻的两个位的值都不相同，所以认为 0x55 这个数据的跳变次数为 7。而对于 8 位的 0x80，其二进制值为 10000000b，其跳变次数为 1。所以不同的数据，其位跳变次数也是不一样的。所以，最小化传输，就是要通过一定的手段，将数据进行重新编码，让所有的编码后的数据其跳变次数都尽可能的少。

为什么要使用最小化传输呢？这个就涉及到信号在具体的线缆中传输的问题了。对于 TMDS 规范，其最终的数据是通过串行的方式将每个 10 位的数据逐位输出的。那么在传输过程中，假设前一个数据是 0，后一个数据是 1，则信号线在传完前一位数据之后，需要马上把电平设置为代表 1 的电平，这个过程中就必然会出现信号线高低电平的变化，信号的变化就会产生磁场。对信号线产生电磁干扰。所以，越少的信号翻转次数，就会产生越小的电磁干扰。

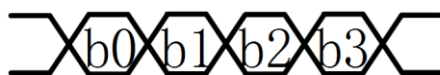


图 1-8

1.6 最小化传输实现原理

经过最小化传输方式实现的 TMDS，其用来作为像素数据的 TMDS 字符包含 5 个或更少的跳变，而用来作为控制数据的 TMDS 字符包含 7 个或更多的跳变。在空期间传送的多跳变内容形成解码端的字符边界的基础，这些字符在串行数据流中个体不是独一无二，但它们足够相似，使得在发送空间隙期间，解码器它们可以唯一地检测出它们连续的存在。

TMDS 数据通道传送的是一个连续的 10bit TMDS 字符流，在空期间，传送 4 个有显著特征的字符，它们直接对应编码器的 2 个控制信号的 4 个可能的状态。在数据有效期间，10bit 的字符包含 8bit 的像素数据，编码的字符提供近似的 DC 平衡，并最少化数据流的跳变次数，对有效像素数据的编码处理可以认为有两个阶段：第一个阶段是依据输入的 8bit 像素数据产生跳变最少的 9bit 代码字；第二阶段是产生一个 10bit 的代码字，最终的 TMDS 字符，将维持发送字符总体的 DC 平衡。

编码器在第一个阶段产生的 9bit 代码字由“8bit” + “1bit”组成，“8bit”反映输入的 8bit 数据位的跳变，“1bit”表示用来描述跳变的两个方法中哪一个被使用，无论哪种方法，输出的最低位都会与输入的最低位相匹配。用一个建立的初值，输出字的余下 7bit 的产生是按照顺序将输入的每一位与前一导出的位进行 XOR 或 NOR（XNOR）。使用 XOR 还是 XNOR 要看哪个方法使得编码结果包含最少的跳变，代码字的第 9 位用来表示导出输出代码是使用 XOR 还是 XNOR，这 9bit 代码字的解码方法很简单，就是相邻位的 XOR 或 XNOR 操作。从解码输入到解码器输出最低位不改变。

在有效数据期间，编码器执行使传输的数据流维持近似的 DC 平衡处理，这是通过选择性地反转第一阶段产生的 9bit 代码中的 8bit 数据位来实现的，第 10bit 被加到代码字上，表示是否进行了反转处理，编码器是基于跟踪发送流中 1 和 0 个数的不一致以及当前代码字 1 和 0 的数目来确定什么时候反转下一个 TMDS 字符。如果太多的 1 被发送，且输入包含的 1 多于 0，则代码字反转，这个发送端的动态编码决定在接收端可以很简单地解码出来，方法是以 TMDS 字符的第 10bit 决定是否对输入代码进行反转。

关于这个编码的详细方法，结合下述流程图将非常的容易理解。

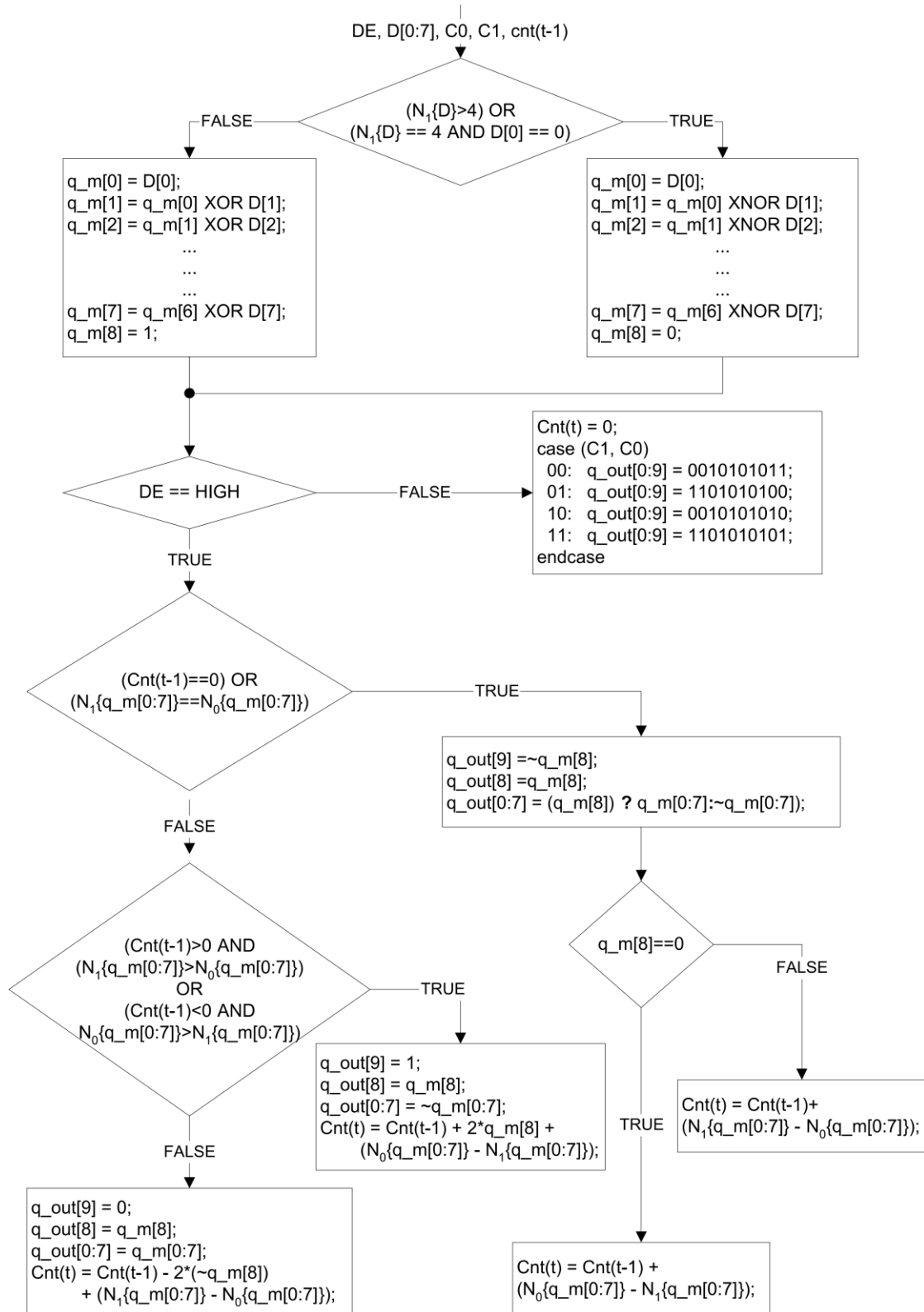


图 1-9 最小化传输编码

表 1-1 最小化传输信号名称含义和功能

信号名称	功能描述
D, C0, C1, DE	编码器输入信号，D 是 8 位的像素数据，C0 和 C1 为控制信号，DE 是数据有效信号（data enable）
cnt	这是一个寄存器，用来记录数据流的极性，一个大于 0 的正值表明了数据流中总共多传了多少个 1，一个小于 0 的负值表明了数据流中总共多传了多少个 0。 cnt{t-1} 为上一次传输的中的极性，cnt{t} 为当前输入的数据的极性。
q_m	最小化传输的编码结果，9 位，由输入的 8 位数据经过最小化传输编码原理编码得到。
q_out	对最小化传输编码结果的 9 位数据继续进行直流平衡编码后得到的 10 位输出结果。
N1{x}	这可以理解为一个函数，该函数会统计输入的参数 X 中 1 的个数
N0{X}	这可以理解为一个函数，该函数会统计输入的参数 X 中 0 的个数

1.6.1 数据编码算法流程

- 先统计输入的数据中有多少个 1。
- 根据输入数据中 1 的个数和输入数据的最低位的值，来确定编码方向。
- 根据编码方向，对输入数据进行编码，方向为 1，则采用异或编码方式，方向为 0，则采用同或编码方式，并将编码方向值作为编码后数据的第 9 位，也就是 q_m[8]。

（即：8bit -> 9bit：(q_m[7:0]是被编码后的数据，q_m[8]是表示方向：q_m[8]为 0，采用同或(^~)运算；q_m[8]为 1，采用异或(^)运算，这就是 TMDS 编码的第一阶段——最小化传输，也就是将 8 位数据变 9 位。)

1.6.2 直流平衡编码

直流平衡编码中，主要是根据前一个编码过程统计的整个数据流中的 1 和 0 的差值，来指导本次编码过程的差值，确保在整个数据流的传输过程中，传输的 1 和 0 的总个数是相差不大的。这样能够保证整个传输链路的直流平衡。

所谓直流平衡，就是指信号在传输中 0 和 1 的数据个数相同，则发送方和接收方直接的就不会有直流电流的传递，在通信系统中，直流平衡可以有效避免由于收发两端电压不稳引起的问题。

1.6.3 控制数据编码

TMDS 编码时将整个的传输内容分为像素数据和控制数据，当 DE 信号为高电平时，编码和传输的是像素数据，当 DE 为低电平时，编码和传输的是控制数据。在任何给定的输入时钟周期上，是对像素数据还是控制数据进行编码，取决于数据使能信号 DE 的状态。当 DE 为高位时，8 位像素数据被编码成 10 位转换最小化，直流平衡的 TMDS 序列。在控制周期，当 DE 低时，dvi 发送器将 2

位控制数据（C0，C1）编码成 10 位序列。下图显示 DVI 周期和 DE 信号之间的关系。

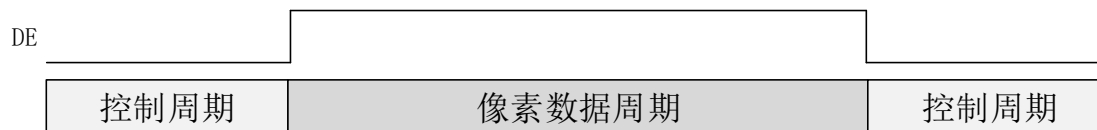


图 1-10 像素数据周期和控制周期切换控制信号

在上面的过程中，已经讨论了像素数据周期的编码方式，而在控制周期，编码则简单的多，只需要针对输入的 CTL0 和 CTL1 的状态输出不同的固定数据即可，查表方式即可实现，无需使用编码算法。

下表为 TMDS 的 3 个通道中每个通道的控制信号对应的信号内容。

表 1-2

TMDS Channel	C0	C1
0	HSYNC	VSYNC
1	CTL0 (0)	CTL1 (0)
2	CTL2 (0)	CTL3 (0)

下表为每个通道的 2 位控制信号的不同状态对应的编码输出结果值。

表 1-3

C1	C0	编码值
0	0	10'b1101010100
0	1	10'b0010101011
1	0	10'b0101010100
1	1	10'b1010101011

1.7 TMDS 编码实现

事实上，只要按照上述流程图中的顺序设计相应的逻辑电路，就能实现该编码器，下述代码为我们根据该流程图编写设计的编码器，实测能够正常的用于图像数据发送中。

```
module encode(  
    clk,  
    rst_n,  
    din,  
    c0,  
    c1,  
    de,  
    dout  
);  
  
input          clk;    // 像素时钟输入
```



```
input          rst_n;    // 异步复位高电平有效
input          [7:0] din;    // 数据输入, 需要寄存
input          c0;        // c0 输入
input          c1;        // c1 输入
input          de;        // 数据使能, 输入
output reg [9:0] dout;    // 数据输出

parameter CTL0 = 10'b1101010100;
parameter CTL1 = 10'b0010101011;
parameter CTL2 = 10'b0101010100;
parameter CTL3 = 10'b1010101011;

reg [3:0] n1d;           //统计输入的 8bit 数据中 1 的个数
reg [7:0] din_q;         //同步寄存输入的 8bit 数据 (统计需要一拍时间)

// 统计每次输入的 8bit 数据中 1 和 0 的个数。流水线输出,同步寄存输入的 8bit 数据
always @ (posedge clk) begin
    din_q <= din;
    n1d <= din[0] + din[1] + din[2] + din[3] + din[4] + din[5] + din[6] + din[7];
end

// 第一步: 8 bit -> 9 bit
// 参考 DVI 规范 1.0, 第 29 页, 图 3-5
wire decision1; //0
assign decision1 = (n1d > 4'h4) | ((n1d == 4'h4) & (din_q[0] == 1'b0));

// 最低位不变, 剩下的等于前一位跟对应的 din_q 相异或运算, 或者是同或运算
// q_m[0] = din_q[0];
// q_m[i+1] = q_m[i] ^ din_q[i+1]; q_m[8] = 1;
// q_m[i+1] = q_m[i] ^~ din_q[i+1]; q_m[8] = 0;
wire [8:0] q_m;
assign q_m[0] = din_q[0];
assign q_m[1] = (decision1) ? ~(q_m[0] ^ din_q[1]) : (q_m[0] ^ din_q[1]);
assign q_m[2] = (decision1) ? ~(q_m[1] ^ din_q[2]) : (q_m[1] ^ din_q[2]);
assign q_m[3] = (decision1) ? ~(q_m[2] ^ din_q[3]) : (q_m[2] ^ din_q[3]);
assign q_m[4] = (decision1) ? ~(q_m[3] ^ din_q[4]) : (q_m[3] ^ din_q[4]);
assign q_m[5] = (decision1) ? ~(q_m[4] ^ din_q[5]) : (q_m[4] ^ din_q[5]);
assign q_m[6] = (decision1) ? ~(q_m[5] ^ din_q[6]) : (q_m[5] ^ din_q[6]);
assign q_m[7] = (decision1) ? ~(q_m[6] ^ din_q[7]) : (q_m[6] ^ din_q[7]);
assign q_m[8] = (decision1) ? 1'b0 : 1'b1;

// 第二步: 9 bit -> 10 bit
// 参考 DVI 规范 1.0, 第 29 页, 图 3-5
reg [3:0] n1q_m, n0q_m; // 统计 q_m 中 1 和 0 的个数
always @ (posedge clk) begin
    n1q_m <= q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] + q_m[7];
    n0q_m <= 4'h8 - (q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] + q_m[7]);
end
```

```
end

reg [4:0] cnt; // 计数器差距统计: 统计 1 和 0 是否过量发送, 最高位(cnt[4])是符号位
wire decision2, decision3;
assign decision2 = (cnt == 5'h0) | (n1q_m == n0q_m);

// [(cnt > 0) and (N1q_m > N0q_m)] or [(cnt < 0) and (N0q_m > N1q_m)]
assign decision3 = (~cnt[4] & (n1q_m > n0q_m)) | (cnt[4] & (n0q_m > n1q_m));

// 流水线对齐(同步寄存器 2 拍)
reg [1:0] de_reg;
reg [1:0] c0_reg;
reg [1:0] c1_reg;
reg [8:0] q_m_reg;
always @ (posedge clk) begin
    de_reg <= {de_reg[0], de};
    c0_reg <= {c0_reg[0], c0};
    c1_reg <= {c1_reg[0], c1};
    q_m_reg <= q_m;
end

// 10-bit 数据输出
always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        dout <= 10'h0;
        cnt <= 5'd0;
    end else begin
        if (de_reg[1]) begin// 数据周期: 发送对应编码的数据
            if (decision2) begin
                dout[9] <= ~q_m_reg[8];
                dout[8] <= q_m_reg[8];
                dout[7:0] <= (q_m_reg[8]) ? q_m_reg[7:0] : ~q_m_reg[7:0];
                cnt <= (~q_m_reg[8]) ? (cnt + n0q_m - n1q_m):(cnt + n1q_m - n0q_m);
            end else begin if (decision3) begin
                dout[9] <= 1'b1;
                dout[8] <= q_m_reg[8];
                dout[7:0] <= ~q_m_reg;
                cnt <= cnt + {q_m_reg[8], 1'b0} + (n0q_m - n1q_m);
            end else begin
                dout[9] <= 1'b0;
                dout[8] <= q_m_reg[8];
                dout[7:0] <= q_m_reg[7:0];
                cnt <= cnt - {~q_m_reg[8], 1'b0} + (n1q_m - n0q_m);
            end
        end
    end
end else begin // 控制周期:发送控制信号
    cnt <= 5'd0;
```

```
case ({c1_reg[1], c0_reg[1]})
    2'b00: dout <= CTL0;
    2'b01: dout <= CTL1;
    2'b10: dout <= CTL2;
    default: dout <= CTL3;
endcase
end
end
end
endmodule
```

1.8 串行发送模块

1.8.1 串行发送原理

在完成了最小化传输编码之后，剩下的就是要将编码好的内容按照串行方式发送出去了。

对于发送器来说，使用 5 倍的编码器工作时钟速率，将编码的 10 位数据采用双数据速率（DDR）形式一位一位的输出。所以整个 TMD5 编码发送模块共需要 2 路时钟，一路供给编码器使用，其时钟速率等于输入接口层的时钟速率，另一路供串行发送器使用，其时钟速率等于输入接口层时钟速率的 5 倍。

至于经常有人疑问的，为啥串行发送时钟的频率是编码器的 5 倍而不是 10 倍，因为做过 UART 串口通信的人都知道，一个 10 位的数据（8 位数据位+1 位起始位+1 位停止位）要通过串口发出去，需要分 10 次发送，每次发送 1 位。而这里为啥只要 5 次呢？这是因为在这里，发送是在时钟的上升沿和下降沿各发送一位。也就是一个时钟周期可以发送 2 位数据，所以 10 位数据只需要 5 个时钟周期即可发完。这样每个 10 位的数据都可以在编码器的一个时钟周期内由串行发送器发送完毕，发送示意图如下图所示：

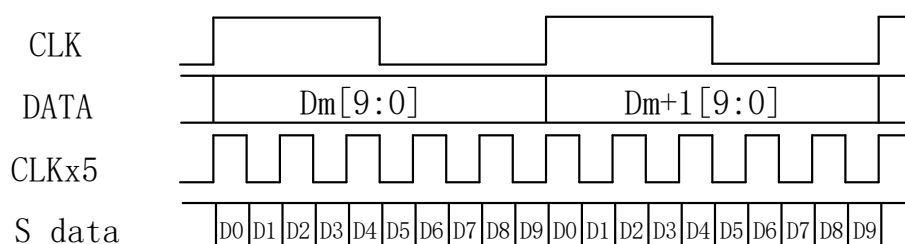


图 1-11 串行发送时序图

在时钟的上升沿和下降沿都发送数据，这就是 DDR（Double Data Rate 双数据速率）接口，DDR 接口能够在不改变传输时钟频率的情况下将单根信号线上传输的数据数量扩大一倍。就大家所感兴趣的 DDR SDRAM 与 SDRAM 来说，两者在结构上的最大区别就是 DDR SDRAM 使用了 DDR 接口来传输数据，使得相同的时钟频率下，DDR SDRAM 的数据传输带宽为 SDRAM 的 2 倍。

1.8.2 FPGA 实现 DDR 接口

几乎所有现在流行的 FPGA 都支持双数据速率接口，Xilinx 的 FPGA 也不例外。在使用时，我们可以在 Vivado 软件中通过调用原语来使用双数据速率 IO。双数据速率 IO 包括 IDDR（输入型双速率 IO）、ODDR（输出型双速率 IO）。通过调用双速率数据 IO 原语，就能够实现双数据速率传输了。打开 Vivado 工具栏 Tools 下的 Language Templates，在打开的搜索框中输入 DDR 就能看到。

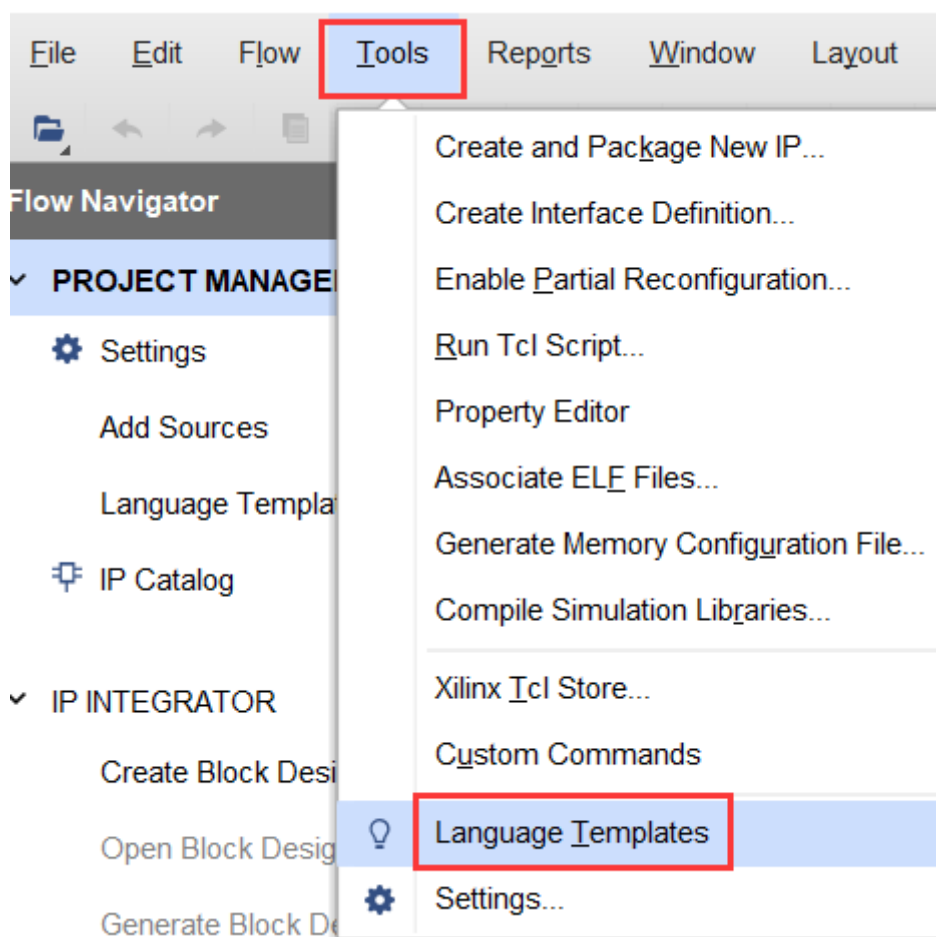


图 1-12 启动 language Templates

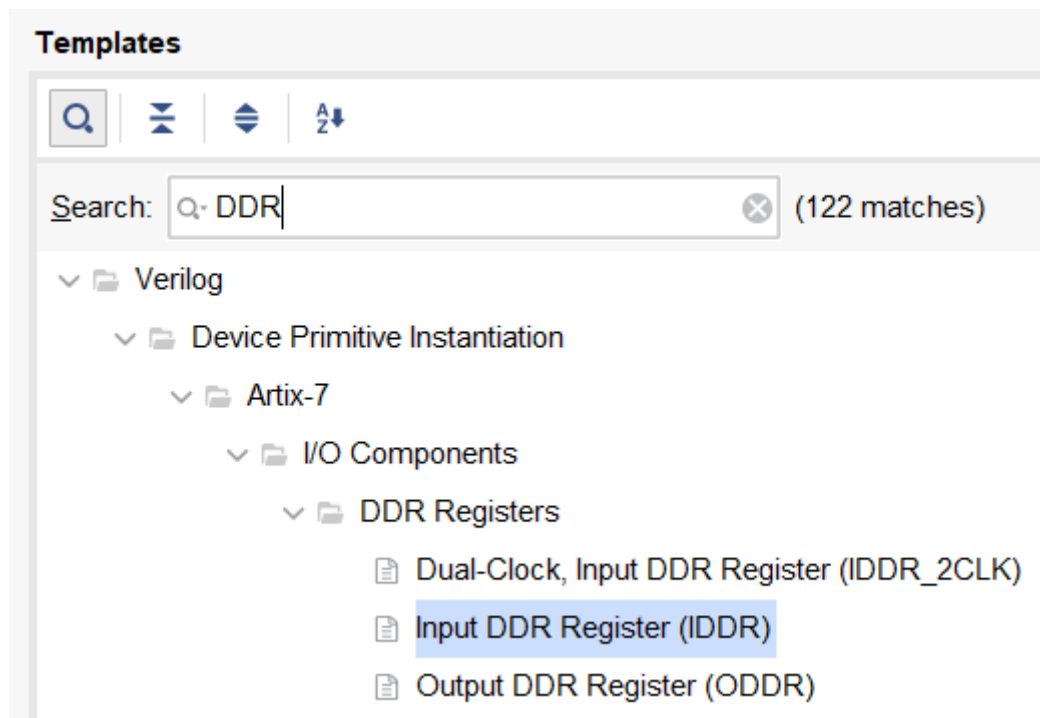


图 1-13 查找 DDR 原语

点击 ODDR，可以在 Template 的右边窗口看到该原语的例化模板，如下：

```
//      ODDR      : In order to incorporate this function into the
design,
//      Verilog    : the following instance declaration needs to be
placed
//      instance   : in the body of the design code. The instance name
//      declaration : (ODDR_inst) and/or the port declarations within
the
//      code       : parenthesis may be changed to properly reference
and
//                  : connect this function to the design. Delete or
comment
//                  : out inputs/outs that are not necessary.

// <-----Cut code below this line----->

// ODDR: Output Double Data Rate Output Register with Set, Reset
//      and Clock Enable.
//      Artix-7
// Xilinx HDL Language Template, version 2018.3

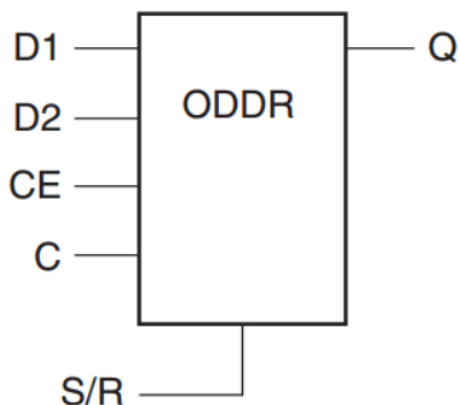
ODDR #(
    .DDR_CLK_EDGE("OPPOSITE_EDGE"), // "OPPOSITE_EDGE" or
"SAME_EDGE"
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1
```



```
.SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_inst (
  .Q(Q),    // 1-bit DDR output
  .C(C),    // 1-bit clock input
  .CE(CE),  // 1-bit clock enable input
  .D1(D1),  // 1-bit data input (positive edge)
  .D2(D2),  // 1-bit data input (negative edge)
  .R(R),    // 1-bit reset
  .S(S)     // 1-bit set
);

// End of ODDR_inst instantiation
```

ODDR 原语结构图如下：



ug471_c2_18_022715

图 1-14 ODDR 原语结构

其端口具体介绍如下：

表 1-4 ODDR 端口信号功能

端口信号名	I/O	位宽	描述
C	I	1	时钟输入
CE	I	1	时钟使能
D1	I	1	输入数据高段输出端口，用户逻辑
D2	I	1	输入数据低段输出端口，用户逻辑
S/R	I	1	复位/置位输入
Q	O	1	ODDR 寄存器输出

ODDR 参数 `DDR_CLK_EDGE` 为模式设置，ODDR 原语支持 `OPPOSITE_EDGE` 模式、`SAME_EDGE` 模式。`SAME_EDGE` 模式与 Virtex-6 架构相同，这个模式允许设计者在 ODDR 时钟的上升沿向 ODDR 原语提供数据输入，从而节省 CLB 和时钟资源，并提高性能。

(1) `OPPOSITE_EDGE` 模式

在此模式中，时钟边沿被用来以两倍的吞吐量从 FPGA 逻辑中捕获数据。这种结构与 virtex-6 的实现比较相似。两个输出都提供给 IOB 的数据输入或者三态控制输入。使用 OPPOSITE_EDGE 模式的输出 DDR 时序图如下图所示

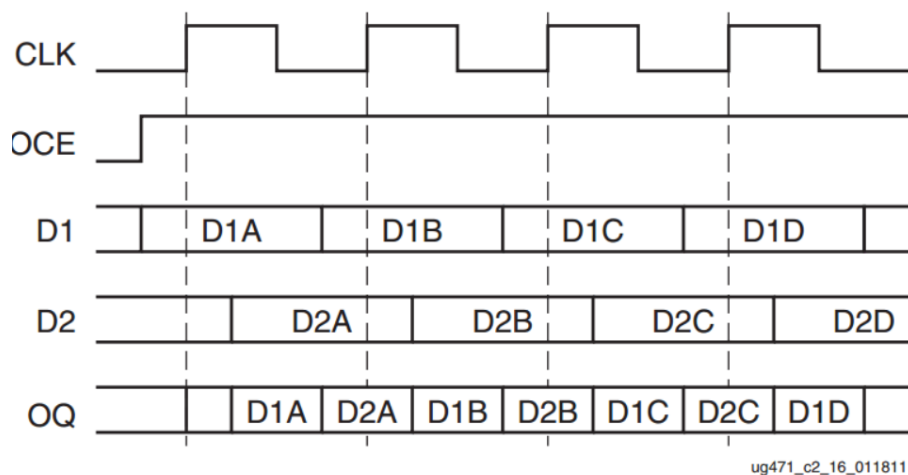


Figure 2-18: Output DDR Timing in OPPOSITE_EDGE Mode

图 1-15 OPPOSITE_EDGE 模式时序

(2) SAME_EDGE 模式

在此模式下，数据可以在相同的时钟边沿送给 IOB。相同的时钟沿将数据送给 IOB 可以避免建立时间违规，并允许用户使用最小的寄存器来执行更高的 DDR 频率来进行寄存器的延迟，而不是使用 CLB 寄存器。下图显示了使用 SAME_EDGE 模式的输出 DDR 的时序图。

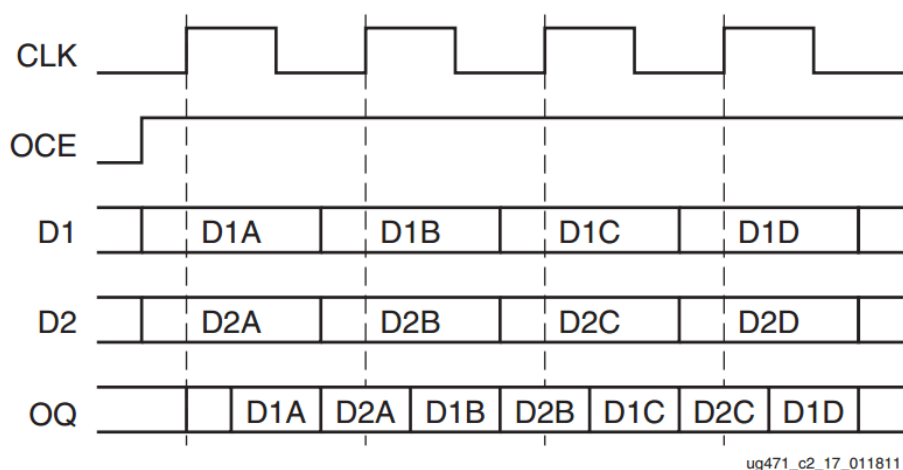


Figure 2-19: Output DDR Timing in SAME_EDGE Mode

图 1-16 SAME_EDGE 模式时序

本节设计模式采用 SAME_EDGE 模式。

1.8.3 TMDS 数据位与 DDR 接口对应关系

在 TMDS 发送中，数据从 FPGA 发出，所以只需要使用 `ddio_out` 核即可。下图为 1 位的 `ddio_out` 核示意图。

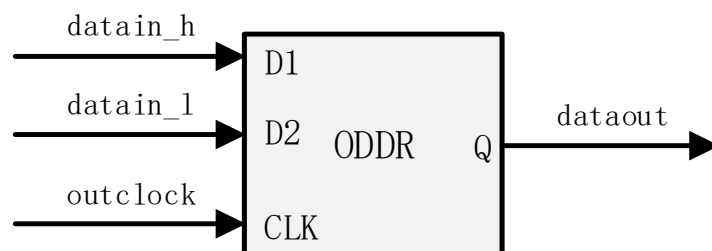


图 1-17 1 位的 `ddio_out` 核

DVI 总共有 4 个通道，其中三个通道为数据通道。由于每个通道需要在 5 个时钟周期内完成 10 位数据的输出，且输出时，从低位开始顺序输出，所以需要每时钟周期切换一次送给 `datain_h` 和 `datain_l` 端口的数据，具体每个时钟周期对应的 `datain_h` 和 `datain_l` 上连接的数据如下表所示：

表 1-5 每个时钟周期对应的 `datain_h` 和 `datain_l` 连接关系

	CLK0	CLK1	CLK2	CLK3	CLK4
<code>datain_h</code>	<code>q_out[0]</code>	<code>q_out[2]</code>	<code>q_out[4]</code>	<code>q_out[6]</code>	<code>q_out[8]</code>
<code>datain_l</code>	<code>q_out[1]</code>	<code>q_out[3]</code>	<code>q_out[5]</code>	<code>q_out[7]</code>	<code>q_out[9]</code>

所以实现时，只需要使用一个计数器，循环的对 5 个时钟周期计数（计数器计数满 4 之后下一个时钟周期清零，或称为模 5 计数器），并使用计数器的输出值作为多路选择器的选择端，选择当前将 `q_out` 中的哪两位数据接到 `datain_h` 和 `datain_l` 上。据此可以绘制出如下所示的逻辑电路图：

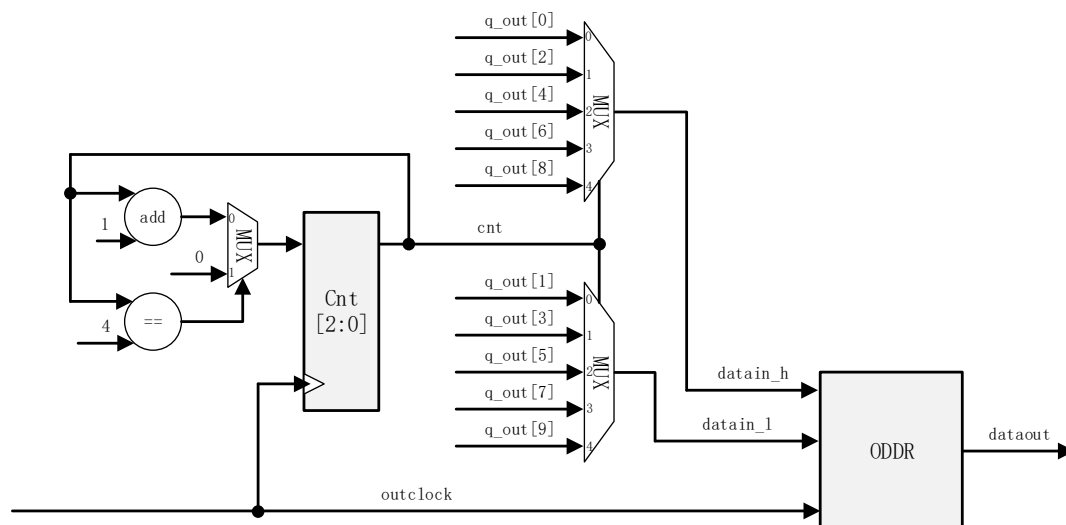


图 1-18 多路器思路切换输入的位

上述图中，使用的是多路器的思路来选择切换 `q_out` 中的每一位，事实上，也可以使用移位寄存器的方式来实现移位。

1.8.4 串行发送模块编码实现

通过上述分析可以知道，送给 `ddio_out` 的 `datain_l` 端口的数据为 `q_out` 的 1、3、5、7、9 位，送给 `ddio_out` 的 `datain_h` 端口的数据为 `q_out` 的 0、2、4、6、8 位。如果使用移位寄存器的方式，则需要先将每个通道的需要发送的高位和低位提取出来组成新的 2 个 5 位的数据，然后再分别移位。我们提供的参考设计中，使用的就是这种移位寄存器的思路。

首先将刚开始进来的三个通道的 10bit 数据都拆分成奇偶两路分别放到 `TMDS_x_l` 和 `TMDS_x_h` 中：

```
wire [4:0] TMDS_0_l =
{datain_0[9],datain_0[7],datain_0[5],datain_0[3],datain_0[1]};
wire [4:0] TMDS_0_h =
{datain_0[8],datain_0[6],datain_0[4],datain_0[2],datain_0[0]};

wire [4:0] TMDS_1_l =
{datain_1[9],datain_1[7],datain_1[5],datain_1[3],datain_1[1]};
wire [4:0] TMDS_1_h =
{datain_1[8],datain_1[6],datain_1[4],datain_1[2],datain_1[0]};

wire [4:0] TMDS_2_l =
{datain_2[9],datain_2[7],datain_2[5],datain_2[3],datain_2[1]};
wire [4:0] TMDS_2_h =
{datain_2[8],datain_2[6],datain_2[4],datain_2[2],datain_2[0]};

wire [4:0] TMDS_3_l =
{datain_3[9],datain_3[7],datain_3[5],datain_3[3],datain_3[1]};
wire [4:0] TMDS_3_h =
{datain_3[8],datain_3[6],datain_3[4],datain_3[2],datain_3[0]};
```

用 5 倍的速率时钟将数据送入移位寄存器，同时用了一个模 5 计数器 `TMDS_mod5`，计数到了 5 个后就更新一次数据

```
// 模 5 计数器
always @(posedge clkx5)
begin
    if(TMDS_mod5 >= 3'd4)
        TMDS_mod5 <= 3'd0;
    else
        TMDS_mod5 <= TMDS_mod5 + 3'd1;
```

```

end

// 5 倍速度移位发送数据
always @(posedge clkx5)
begin
    if(TMDS_mod5 == 3'd4)begin
        TMDS_shift_0h <= TMDS_0_h;
        TMDS_shift_0l <= TMDS_0_l;
        TMDS_shift_1h <= TMDS_1_h;
        TMDS_shift_1l <= TMDS_1_l;
        TMDS_shift_2h <= TMDS_2_h;
        TMDS_shift_2l <= TMDS_2_l;
        TMDS_shift_3h <= TMDS_3_h;
        TMDS_shift_3l <= TMDS_3_l;

    end
    else begin
        TMDS_shift_0h <= TMDS_shift_0h[4:1];
        TMDS_shift_0l <= TMDS_shift_0l[4:1];
        TMDS_shift_1h <= TMDS_shift_1h[4:1];
        TMDS_shift_1l <= TMDS_shift_1l[4:1];
        TMDS_shift_2h <= TMDS_shift_2h[4:1];
        TMDS_shift_2l <= TMDS_shift_2l[4:1];
        TMDS_shift_3h <= TMDS_shift_3h[4:1];
        TMDS_shift_3l <= TMDS_shift_3l[4:1];

    end
end
end

```

移位完成之后，只需要将每个通道的高低位移位寄存器的值分别送往对应的 ddio_out 模块的 datain_h 和 datain_l 端口即可。如下所示：

```

//Channel 0
ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYN"
) ODDR_0 (
    .Q (dataout_0 ),// 1-bit DDR output
    .C (clkx5 ),// 1-bit clock input
    .CE(1'b1 ),// 1-bit clock enable input
    .D1(TMDS_shift_0l[0] ),// 1-bit data input (positive edge)
    .D2(TMDS_shift_0h[0] ),// 1-bit data input (negative edge)
    .R (1'b0 ),// 1-bit reset
    .S (1'b0 ) // 1-bit set
);

//Channel 1
ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"

```



```
.INIT(1'b0),    // Initial value of Q: 1'b0 or 1'b1
.SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_1 (
.Q (dataout_1      ),// 1-bit DDR output
.C (clkx5          ),// 1-bit clock input
.CE(1'b1           ),// 1-bit clock enable input
.D1(TMDS_shift_1l[0]),// 1-bit data input (positive edge)
.D2(TMDS_shift_1h[0]),// 1-bit data input (negative edge)
.R (1'b0           ),// 1-bit reset
.S (1'b0           ) // 1-bit set
);

//Channel 2
ODDR #(
.DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
.INIT(1'b0),    // Initial value of Q: 1'b0 or 1'b1
.SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_2 (
.Q (dataout_2      ),// 1-bit DDR output
.C (clkx5          ),// 1-bit clock input
.CE(1'b1           ),// 1-bit clock enable input
.D1(TMDS_shift_2l[0]),// 1-bit data input (positive edge)
.D2(TMDS_shift_2h[0]),// 1-bit data input (negative edge)
.R (1'b0           ),// 1-bit reset
.S (1'b0           ) // 1-bit set
);

//Channel 3
ODDR #(
.DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
.INIT(1'b0),    // Initial value of Q: 1'b0 or 1'b1
.SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_3 (
.Q (dataout_3      ),// 1-bit DDR output
.C (clkx5          ),// 1-bit clock input
.CE(1'b1           ),// 1-bit clock enable input
.D1(TMDS_shift_3l[0]),// 1-bit data input (positive edge)
.D2(TMDS_shift_3h[0]),// 1-bit data input (negative edge)
.R (1'b0           ),// 1-bit reset
.S (1'b0           ) // 1-bit set
);
```

另外，对于 TMDS，其传输时使用的是差分传输方式，既对每一个通道都使用 2 根信号线，两根信号线传输的电平刚好相反，所以在最终输出时，需要再使用 Xilinx 的 OBUFDS 原语，用来将上述 ODDR 模块输出信号转成差分信号输出。

至此，整个串行发送模块的设计思路就完全介绍清楚了。相信稍有 Verilog 编程基础的读者都能根据此介绍完成相关代码的编写，以下为笔者提供的参考设计代码。

发送模块 serdes_4b_10to1 代码如下：

```
module serdes_4b_10to1(
    clkx5,
    datain_0,
    datain_1,
    datain_2,
    datain_3,
    dataout_0_p,
    dataout_0_n,
    dataout_1_p,
    dataout_1_n,
    dataout_2_p,
    dataout_2_n,
    dataout_3_p,
    dataout_3_n
);
    input          clkx5;           // 5x clock input
    input  [9:0]   datain_0;        // input data for serialisation
    input  [9:0]   datain_1;        // input data for serialisation
    input  [9:0]   datain_2;        // input data for serialisation
    input  [9:0]   datain_3;        // input data for serialisation
    output        dataout_0_p;      // out DDR data
    output        dataout_0_n;      // out DDR data
    output        dataout_1_p;      // out DDR data
    output        dataout_1_n;      // out DDR data
    output        dataout_2_p;      // out DDR data
    output        dataout_2_n;      // out DDR data
    output        dataout_3_p;      // out DDR data
    output        dataout_3_n;      // out DDR data

    wire dataout_0_n;
    wire dataout_1_n;
    wire dataout_2_n;
    wire dataout_3_n;

    reg [2:0] TMD5_mod5 = 0; // 模 5 计数器

    reg [4:0] TMD5_shift_0h = 0, TMD5_shift_0l = 0;
    reg [4:0] TMD5_shift_1h = 0, TMD5_shift_1l = 0;
    reg [4:0] TMD5_shift_2h = 0, TMD5_shift_2l = 0;
    reg [4:0] TMD5_shift_3h = 0, TMD5_shift_3l = 0;
```

```
wire [4:0] TMDS_0_l =
{datain_0[9],datain_0[7],datain_0[5],datain_0[3],datain_0[1]};
wire [4:0] TMDS_0_h =
{datain_0[8],datain_0[6],datain_0[4],datain_0[2],datain_0[0]};

wire [4:0] TMDS_1_l =
{datain_1[9],datain_1[7],datain_1[5],datain_1[3],datain_1[1]};
wire [4:0] TMDS_1_h =
{datain_1[8],datain_1[6],datain_1[4],datain_1[2],datain_1[0]};

wire [4:0] TMDS_2_l =
{datain_2[9],datain_2[7],datain_2[5],datain_2[3],datain_2[1]};
wire [4:0] TMDS_2_h =
{datain_2[8],datain_2[6],datain_2[4],datain_2[2],datain_2[0]};

wire [4:0] TMDS_3_l =
{datain_3[9],datain_3[7],datain_3[5],datain_3[3],datain_3[1]};
wire [4:0] TMDS_3_h =
{datain_3[8],datain_3[6],datain_3[4],datain_3[2],datain_3[0]};

// 5 倍速度移位发送数据
always @(posedge clkx5)
begin
    TMDS_mod5 <= (TMDS_mod5[2]) ? 3'd0 : TMDS_mod5 + 3'd1;
    TMDS_shift_0h <= TMDS_mod5[2] ? TMDS_0_h : TMDS_shift_0h[4:1];
    TMDS_shift_0l <= TMDS_mod5[2] ? TMDS_0_l : TMDS_shift_0l[4:1];
    TMDS_shift_1h <= TMDS_mod5[2] ? TMDS_1_h : TMDS_shift_1h[4:1];
    TMDS_shift_1l <= TMDS_mod5[2] ? TMDS_1_l : TMDS_shift_1l[4:1];
    TMDS_shift_2h <= TMDS_mod5[2] ? TMDS_2_h : TMDS_shift_2h[4:1];
    TMDS_shift_2l <= TMDS_mod5[2] ? TMDS_2_l : TMDS_shift_2l[4:1];
    TMDS_shift_3h <= TMDS_mod5[2] ? TMDS_3_h : TMDS_shift_3h[4:1];
    TMDS_shift_3l <= TMDS_mod5[2] ? TMDS_3_l : TMDS_shift_3l[4:1];
end

ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYN"
) ODDR_0 (
    .Q (dataout_0), // 1-bit DDR output
    .C (clkx5), // 1-bit clock input
    .CE(1'b1), // 1-bit clock enable input
    .D1(TMDS_shift_0l[0]), // 1-bit data input (positive edge)
    .D2(TMDS_shift_0h[0]), // 1-bit data input (negative edge)
    .R (1'b0), // 1-bit reset
    .S (1'b0) // 1-bit set

```

```
);

OBUFDS #(
    .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
    .SLEW("SLOW")           // Specify the output slew rate
) OBUFDS_0 (
    .O (dataout_0_p ),// Diff_p output (connect directly to top-level
port)
    .OB (dataout_0_n ),// Diff_n output (connect directly to top-level
port)
    .I (dataout_0 ) // Buffer input
);

ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_1 (
    .Q (dataout_1 ),// 1-bit DDR output
    .C (clkx5 ),// 1-bit clock input
    .CE(1'b1 ),// 1-bit clock enable input
    .D1(TMDS_shift_1l[0] ),// 1-bit data input (positive edge)
    .D2(TMDS_shift_1h[0] ),// 1-bit data input (negative edge)
    .R (1'b0 ),// 1-bit reset
    .S (1'b0 ) // 1-bit set
);

OBUFDS #(
    .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
    .SLEW("SLOW")           // Specify the output slew rate
) OBUFDS_1 (
    .O (dataout_1_p ),// Diff_p output (connect directly to top-level
port)
    .OB (dataout_1_n ),// Diff_n output (connect directly to top-level
port)
    .I (dataout_1 ) // Buffer input
);

ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_2 (
    .Q (dataout_2 ),// 1-bit DDR output
    .C (clkx5 ),// 1-bit clock input
    .CE(1'b1 ),// 1-bit clock enable input
    .D1(TMDS_shift_2l[0] ),// 1-bit data input (positive edge)
```

```
.D2(TMDS_shift_2h[0] ),// 1-bit data input (negative edge)
.R (1'b0           ),// 1-bit reset
.S (1'b0           ) // 1-bit set
);

OBUFDS #(
    .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
    .SLEW("SLOW")           // Specify the output slew rate
) OBUFDS_2 (
    .O (dataout_2_p ),// Diff_p output (connect directly to top-level
port)
    .OB (dataout_2_n ),// Diff_n output (connect directly to top-level
port)
    .I (dataout_2   ) // Buffer input
);

ODDR #(
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0),               // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC")           // Set/Reset type: "SYNC" or "ASYNC"
) ODDR_3 (
    .Q (dataout_3           ),// 1-bit DDR output
    .C (clkx5               ),// 1-bit clock input
    .CE(1'b1                ),// 1-bit clock enable input
    .D1(TMDS_shift_3l[0] ),// 1-bit data input (positive edge)
    .D2(TMDS_shift_3h[0] ),// 1-bit data input (negative edge)
    .R (1'b0                 ),// 1-bit reset
    .S (1'b0                 ) // 1-bit set
);

OBUFDS #(
    .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
    .SLEW("SLOW")           // Specify the output slew rate
) OBUFDS_3 (
    .O (dataout_3_p ),// Diff_p output (connect directly to top-level
port)
    .OB (dataout_3_n ),// Diff_n output (connect directly to top-level
port)
    .I (dataout_3   ) // Buffer input
);

endmodule
```

1.9 DVI 发送器实现

完成了底层编码和串行发送器的设计之后，对于 DVI 接口，只需要将编码

器和发送器例化并与图像数据流的数据和控制信号按照 DVI 规范链接到一起即可。如下图所示：

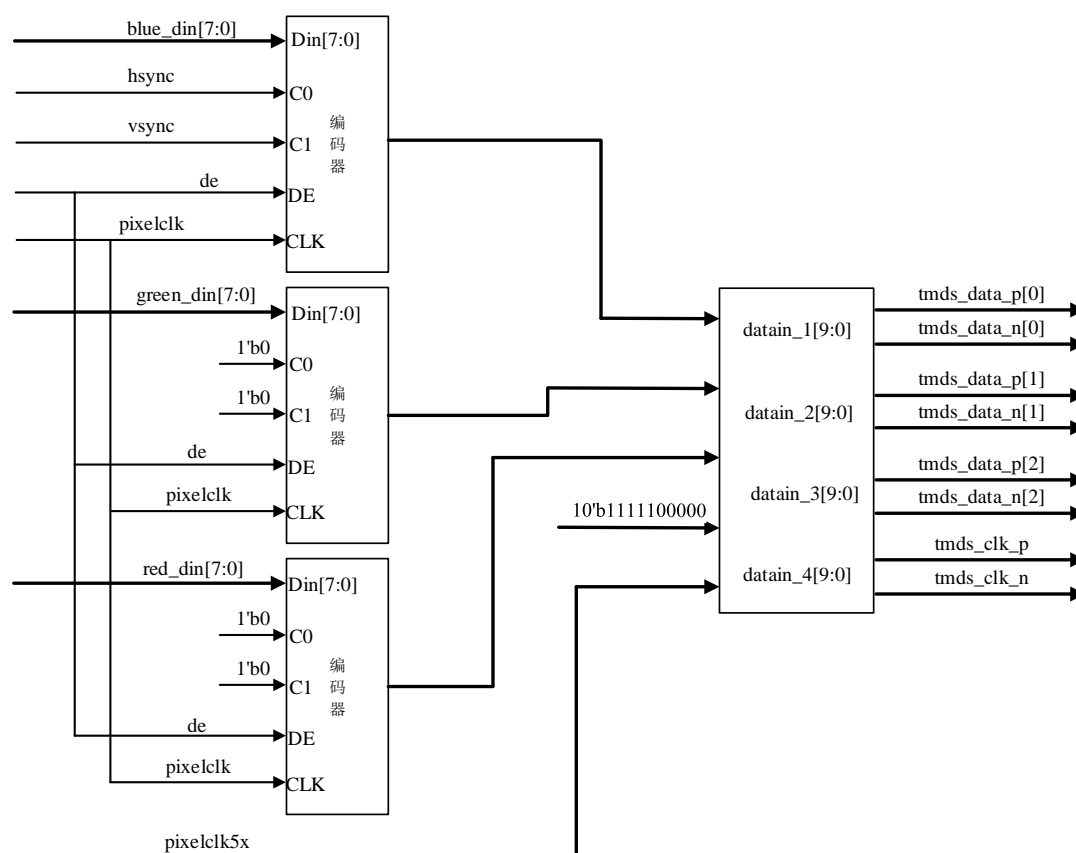


图 1-19 DVI 发送器原理图

TMDS 要求在发送时将像素时钟随数据一起发送，注意，这里的时钟频率应该是和 pixelclk 的频率一致，而不是和 pixelclk5x 一致，也就是说，tmds_clk 并不是数据的位同步时钟而是一个完整编码数据的同步时钟，每次 tmds_clk 的下降沿标志着新的 10 位数据的开始发送。tmds_clk 和 data 的关系如下图所示。

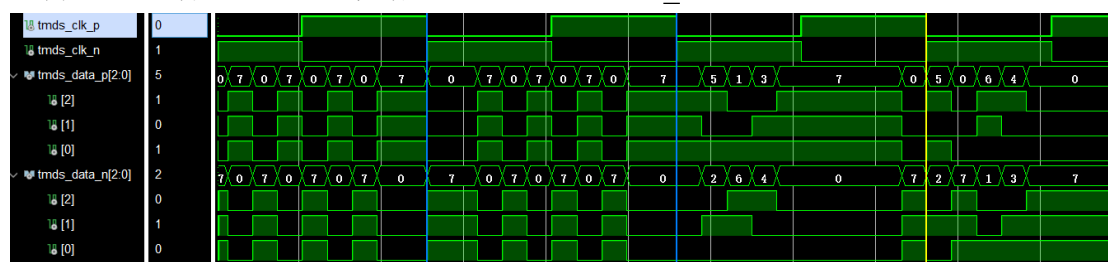


图 1-20 tmds_clk 和 data 的关系图

为了产生 tmds_clk，有两种方式，一种方式是直接将 pixelclk 及其取反信号输出作为 tmds_clk，另一种方式是使用 ODDR 以数据的形式产生。使用 ODDR 以数据的形式产生，能够一定程度上保证输出的 tmds_clk 和 tmds_data 之间拥有

相同的相位关系，便于数据在接收端进行同步，所以这里使用 ODDR 以数据的形式产生。产生这种波形的方式很简单，就是让 ODDR 引脚的前 2.5 个时钟周期输出低电平，后 2.5 个时钟周期输出高电平即可。此时，只需要对串行发送器的 datain_3 端口赋值一个常量 10'b1111100000 即可。完整的 DVI 编码发送器顶层如下所示。

```
module dvi_encoder(
    pixelclk,
    pixelclk5x,
    rst_p,
    blue_din,
    green_din,
    red_din,
    hsync,
    vsync,
    de,
    tmds_clk_p,
    tmds_clk_n,
    tmds_data_p,
    tmds_data_n
);
    input      pixelclk;           // system clock
    input      pixelclk5x;        // system clock x5
    input      rst_p;             // reset
    input [7:0] blue_din;         // Blue data in
    input [7:0] green_din;        // Green data in
    input [7:0] red_din;          // Red data in
    input      hsync;             // hsync data
    input      vsync;             // vsync data
    input      de;                // data enable
    output     tmds_clk_p;         //clock
    output     tmds_clk_n;        //clock
    output [2:0] tmds_data_p;      //rgb
    output [2:0] tmds_data_n;     //rgb

    wire [9:0] red;
    wire [9:0] green;
    wire [9:0] blue;

    encode encb(
        .clk      (pixelclk ),
        .rst_p     (rst_p    ),
        .din       (blue_din ),
        .c0        (hsync    ),
        .c1        (vsync    ),
        .de        (de       ),
```

```
.dout    (blue    )
);

encode encr(
    .clk    (pixelclk ),
    .rst_p   (rst_p    ),
    .din     (green_din ),
    .c0      (1'b0     ),
    .c1      (1'b0     ),
    .de      (de       ),
    .dout     (green   )
);

encode encg(
    .clk    (pixelclk ),
    .rst_p   (rst_p    ),
    .din     (red_din  ),
    .c0      (1'b0     ),
    .c1      (1'b0     ),
    .de      (de       ),
    .dout     (red     )
);

serdes_4b_10to1 serdes_4b_10to1_inst(
    .clkx5      (pixelclk5x      ),// 5x clock input
    .datain_0    (blue            ),// input data for serialisation
    .datain_1    (green           ),// input data for serialisation
    .datain_2    (red             ),// input data for serialisation
    .datain_3    (10'b1111100000 ),// input data for serialisation
    .dataout_0_p  (tmds_data_p[0] ),// out DDR data
    .dataout_0_n  (tmds_data_n[0] ),// out DDR data
    .dataout_1_p  (tmds_data_p[1] ),// out DDR data
    .dataout_1_n  (tmds_data_n[1] ),// out DDR data
    .dataout_2_p  (tmds_data_p[2] ),// out DDR data
    .dataout_2_n  (tmds_data_n[2] ),// out DDR data
    .dataout_3_p  (tmds_clk_p     ),// out DDR data
    .dataout_3_n  (tmds_clk_n     ) // out DDR data
);

endmodule
```

1.10 基于 DVI 接口的显示器彩条显示实验

接下来，我们将基于 ACZ702 开发板，通过彩条实验对设计好的 DVI 发送模块进行验证。设计将基于 1280*720 分辨率，输出的图像数据通过 HDMI 线缆，

在支持 HDMI 接口的显示器上显示。基于显示屏上显示的画面，判断设计是否成功。不同于直接使用开发板上的 SiI9022 芯片，本次设计是通过 IO 模拟 HDMI 接口，实现 HDMI 显示，该方法需要有硬件电路的支持，因此，本次设计需要使用到 ACM_VGAHDMI 模块。

1.10.1 ACM_VGAHDMI 模块说明

为了满足不同用户的显示需求，芯路恒公司提供了一个拓展模块 ACM_VGAHDMI。该模块拥有 HDMI 和 VGA 接口，支持 IO 模拟的 HDMI 输入/输出以及 VGA 输出。模块如图 1-21 所示：

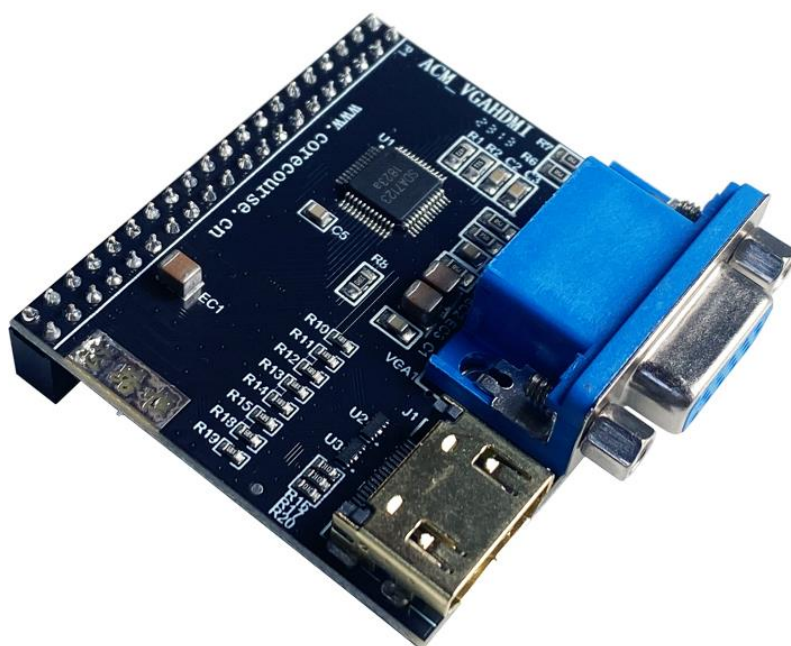


图 1-21 ACM_VGAHDMI 模块

模块采用基于 TMDS 编码的 HDMI 接口，该接口由于采用高速差分数字编码传输方式，具有更强的抗干扰能力，最高能够支持输出高清 1080P 的相关图像信息。本次设计我们便需要借助该模块来验证 DVI 发送器功能完整性。

1.10.2 系统结构

要验证 DVI 发送器功能的完整性，我们需要为其提供有效的 VGA 显示系统，因此需要复用到前面章节中设计好的多分辨率适配型 VGA 控制器。通过彩条实验，借助 VGA 控制器完产生相关显示数据和时序信号。显示数据和时序信号由我们设计好的 DVI 发送器进行串并转换和 TMDS 编码后，交由 ACM_VGAHDMI 模块通过 HDMI 接口传输到 HDMI 显示器上显示。

综上所述，本次设计的系统框图如图 1-22 所示：

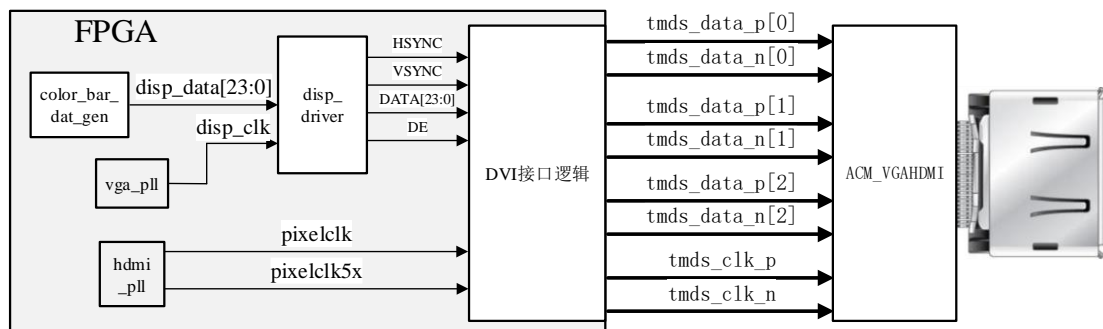


图 1-22 DVI 输出逻辑接口图

本节实验，配置 VGA 控制器输出 1280*720 分辨率的时序信号，在“disp_parameter_cfg.v”文件中设置使能“Resolution_1280x720”选项即可。另外，由于 TMDS 编码需要 2 路时钟，一路与 VGA 控制器同频，作为基本的逻辑工作时钟，另一路为 5 倍的 VGA 控制器时钟频率，用作 TMDS 串行编码的发送时钟。所以，相较于直接驱动 VGA 显示器，驱动 HDMI 显示器需要使用 PLL 产生相应频率的时钟信号（720p 为 74.25MHz），使用 DVI 接口输出，还需要使用 PLL 再产生一路 5 倍的时钟信号，对于 720p 分辨率，也就是 371.25MHz。下图为整个系统在 Vivado 中综合出来的 RTL 视图。

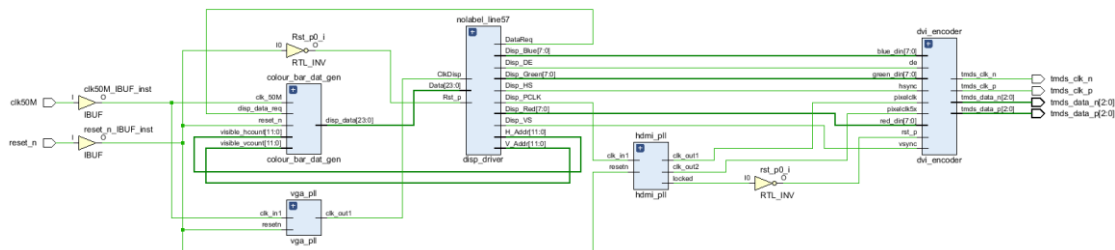


图 1-23 系统综合后的 RTL 视图

其中 colour_bar_dat_gen 为彩条数据产生模块，这里是为了让代码结构更加的清晰，将其独立出来作为了一个单独的模块。当然，由于分辨率变了，所以彩条发生器中对应的在什么位置显示什么颜色的坐标值也相应需要修改。

disp_driver 为 VGA 实验中设计的多分辨率适配的 VGA/TFT 控制器，负责产生相应分辨率的 VGA 驱动时序信号。dvi_encoder 则为本节内容介绍的 DVI 发送器，实现将 disp_driver 产生的 VGA 时序编码为 DVI 标准时序发送出去。

vga_pll 用于产生 disp_driver 模块输出 1280*720 分辨率所需的 74.25MHz 时钟，hdmi_pll 产生 DVI 编码器进行并串转换时所需的 74.25MHz 时钟和 371.25MHz 时钟（clk_out2）。至此，整个验证工程就设计完成了，接下来开始

硬件连接。

1.10.3 系统所需硬件

1. ACZ702 开发板
2. 电源线一根（可选）
3. Type-c 线一根
4. 拓展模块 ACM_VGAHDMI 一个
5. HDMI 电缆一根
6. 支持 HDMI 接口的液晶显示器一台

1.10.4 硬件连接

本次设计需要用到 ACM_VGAHDMI 模块，通过模块上的 HDMI 接口完成设计。硬件连接如图 1-24 所示：

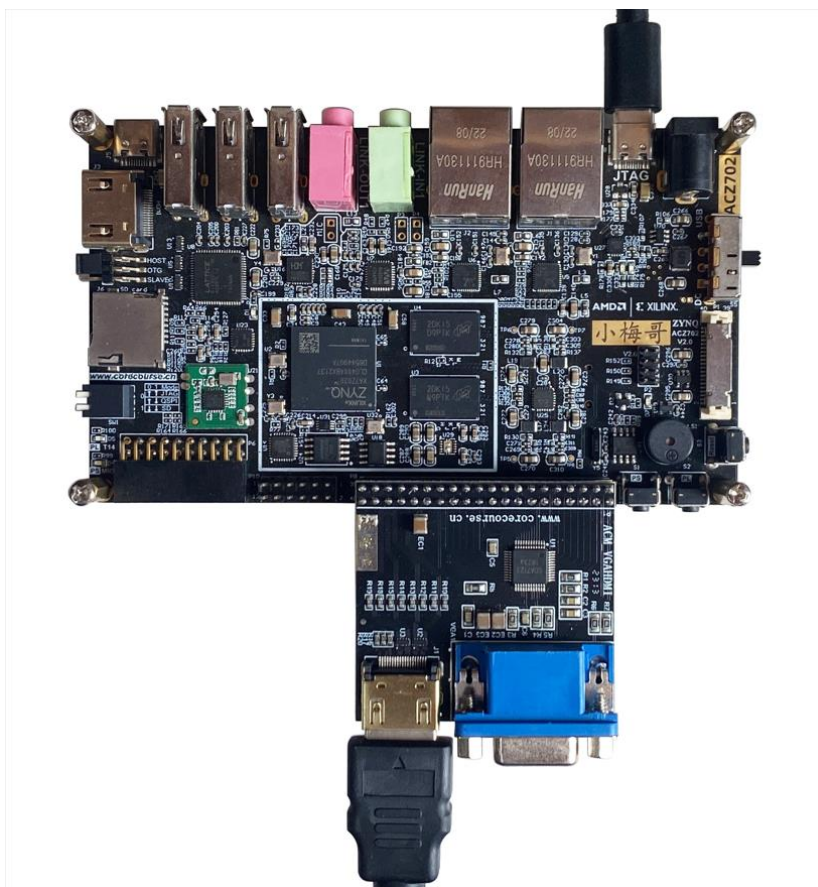


图 1-24 硬件连接

本次设计对供电要求不高，可以仅使用 type-c 供电。ACM_VGAHDMI 模块

通过 40pin 拓展接口与 ACZ702 开发板相连，正确连接时引脚应当一一对应。图像数据通过 ACM_VGAHDMI 模块上的 HDMI 接口输出，在连接 HDMI 线缆时请确保线缆一端连接在模块的 HDMI 接口，一端连接在显示器的 HDMI 接口上。

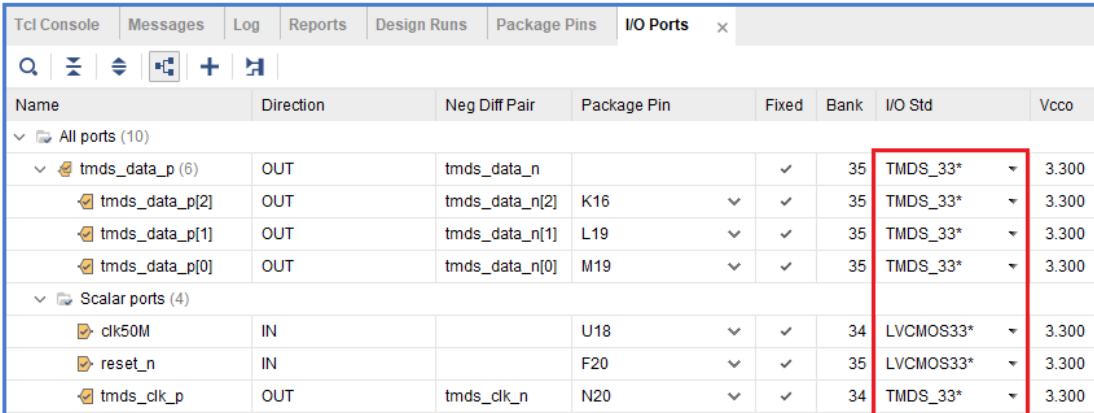
1.10.5 下载与验证

连接完硬件后，我们需要为设计分配引脚。除了时钟与复位引脚外，还有 ACM_VGAHDMI 模块上的 HDMI 引脚信号。本次设计引脚分配表如表 1-6 所示：

表 1-6 管脚分配表

Pin Name	Signal Name	Pin NO.	Pin Name	Signal Name	Pin NO.
FPGA_GCLK1	clk50M	U18	FPGA_KEY0	reset_n	F20
HDMI_D2_P	tmds_data_p[2]	K16	HDMI_D2_N	tmds_data_n[2]	J16
HDMI_D1_P	tmds_data_p[1]	L19	HDMI_D1_N	tmds_data_n[1]	L20
HDMI_D0_P	tmds_data_p[0]	M19	HDMI_D0_N	tmds_data_n[0]	M20
HDMI_CLK_P	tmds_clk_p	N20	HDMI_CLK_N	tmds_clk_n	P20

由于 hdmi 相关信号使用的是差分引脚，在进行引脚分配时，只需要分配 P 脚，vivado 会自动为我们匹配对应的 N 脚。分配完引脚后，还需要对引脚电平约束，其中 hdmi 差分引脚约束电平为 TMD5_33，其余引脚约束电平为 LVCMOS33，如图 1-25 所示：



Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco
All ports (10)							
tmds_data_p (6)							
tmds_data_p[2]	OUT	tmds_data_n[2]	K16	✓	35	TMD5_33*	3.300
tmds_data_p[1]	OUT	tmds_data_n[1]	L19	✓	35	TMD5_33*	3.300
tmds_data_p[0]	OUT	tmds_data_n[0]	M19	✓	35	TMD5_33*	3.300
Scalar ports (4)							
clk50M	IN		U18	✓	34	LVCMOS33*	3.300
reset_n	IN		F20	✓	35	LVCMOS33*	3.300
tmds_clk_p	OUT	tmds_clk_n	N20	✓	34	TMD5_33*	3.300

图 1-25 管脚约束

完成管脚绑定后，直接将工程进行全编译，并下载 bit 文件到开发板。

可以看到。该工程几乎就是在原本的 VGA/TFT 控制器工程基础上，把 VGA/TFT 控制器原本的输出信号再经过了一级 DVI 编码后通过差分形式输出，所以使用起来非常的方便。下图为该工程运行时在 HDMI 显示器上的实际显示效果。

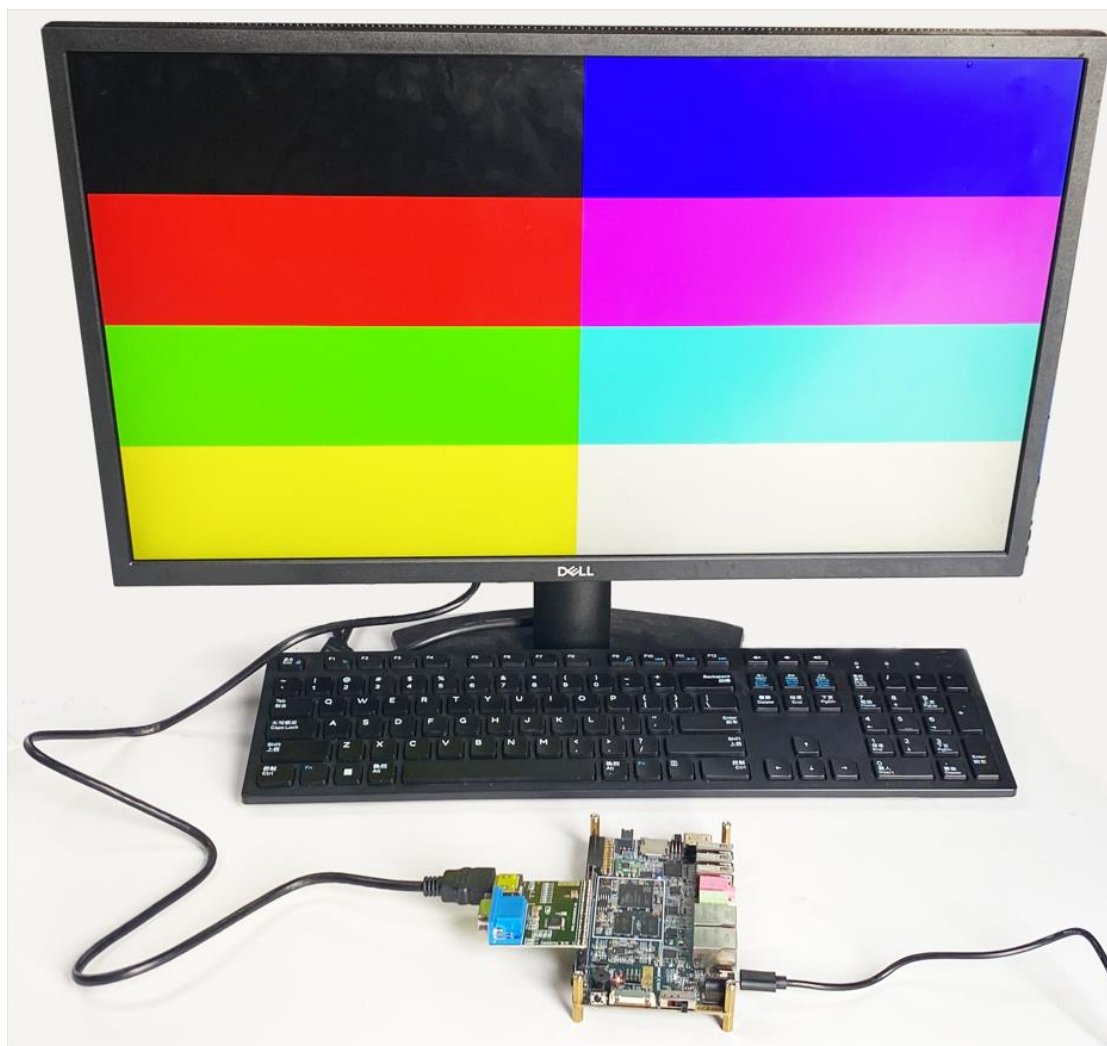


图 1-26 ACZ702 开发板彩条实验效果图

可以看到，彩条图像成功在 HDMI 显示器上显示，图像比例正常，各颜色区域层次清晰，显示分辨率为正确的 1280*720，说明本次设计的 DVI 发送模块功能正常，本次设计成功。

1.11 总结

本章，我们带大家学习 DVI/HDMI 接口原理以及实现方法，完成了 DVI 发送模块的设计，并通过具体的实验，验证了设计模块功能的正确性。

对于 ACZ702 开发板来说，要完成 HDMI 显示一共有两种方案，一种是通过配置 HDMI 芯片 SiI9022，实现 HDMI 显示。一种是通过代码实现 DVI 发送逻辑，借用外接模块 ACM_VGAHDMI 上的 HDMI 接口，实现 HDMI 显示。这两种方案各有自己的有缺点，用户在进行 DVI/HDMI 显示相关设计时，可以根据

自己的设计需求，灵活选择具体的实现方案。