目录

1 Gowin 软件安装指导

我们工程中使用的 Gowin 软件是广东高云半导体股份有限公司的 FPGA 开发软件，读者可以去高云公司的官网下载最新的软件，本节内容将讲解 Gowin 软件的安装方法。

1.1 安装包获取

大家可以通过多种渠道获取到 Gowin 软件，包括但不限于紫光同创官网，各商家提供的网络下载链接。高云官网链接如下：

http://www.gowinsemi.com.cn

进入官网之后，找到开发者专区下的高云云源软件一栏，便可以看到 Gowin 软件的下载链接，点击下载安装包，其中包括商业版和教育版，商业版需要申请 license，教育版不需要，我们开发过程中使用的商业版的软件，接下来也将以此来进行软件安装说明。

1.2 安装步骤说明

安装时，先将 PDS 软件安装包解压到不含中文、空格、特殊字符的路径下，解压完成之后如下图 1-1 所示的文件。

图 1-1 Gowin 安装包

双击图中的.exe 文件，就会出现如下图 1-2 所示的界面，点击“Next”进入下一步。

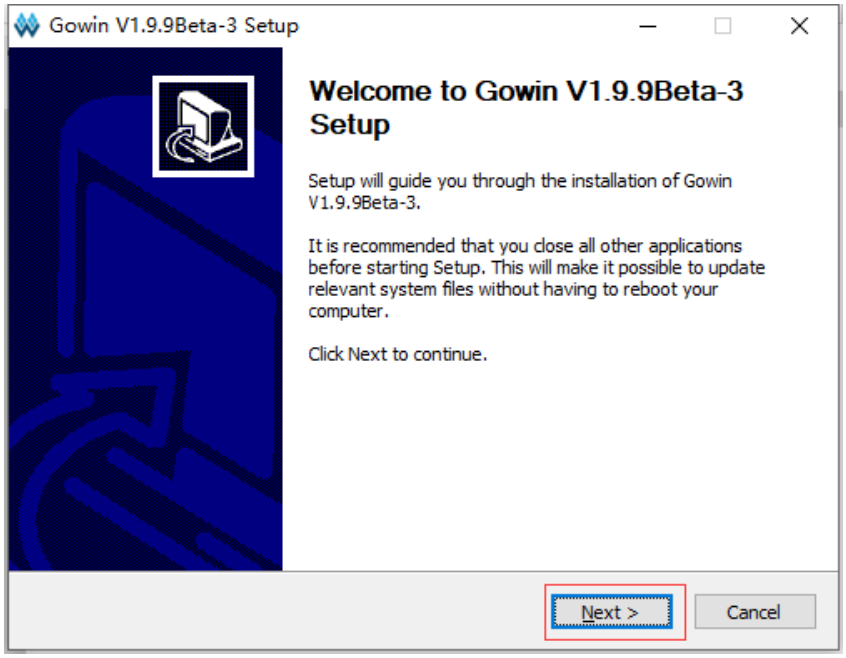


图 1-2 安装启动界面

之后会弹出如下图 1-3 所示的界面，在弹出的界面中选择“I Agree”。

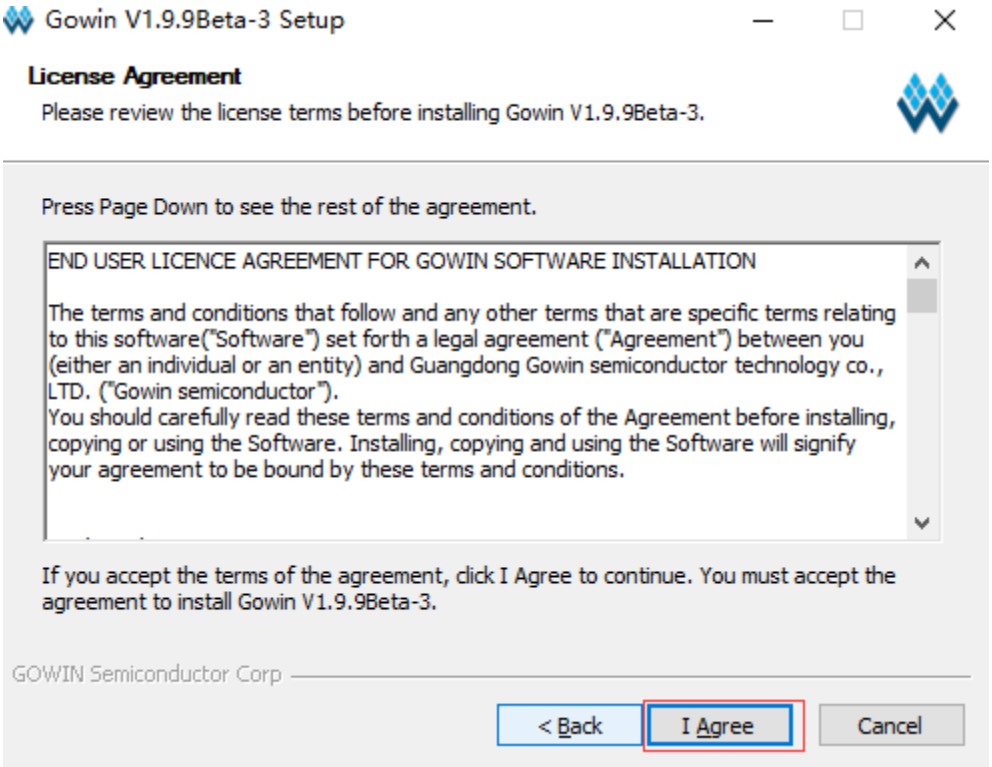


图 1-3 安装引导

点击“I Agree”之后会弹出如下图 1-4 所示的界面，点击“Next”进入下一步。

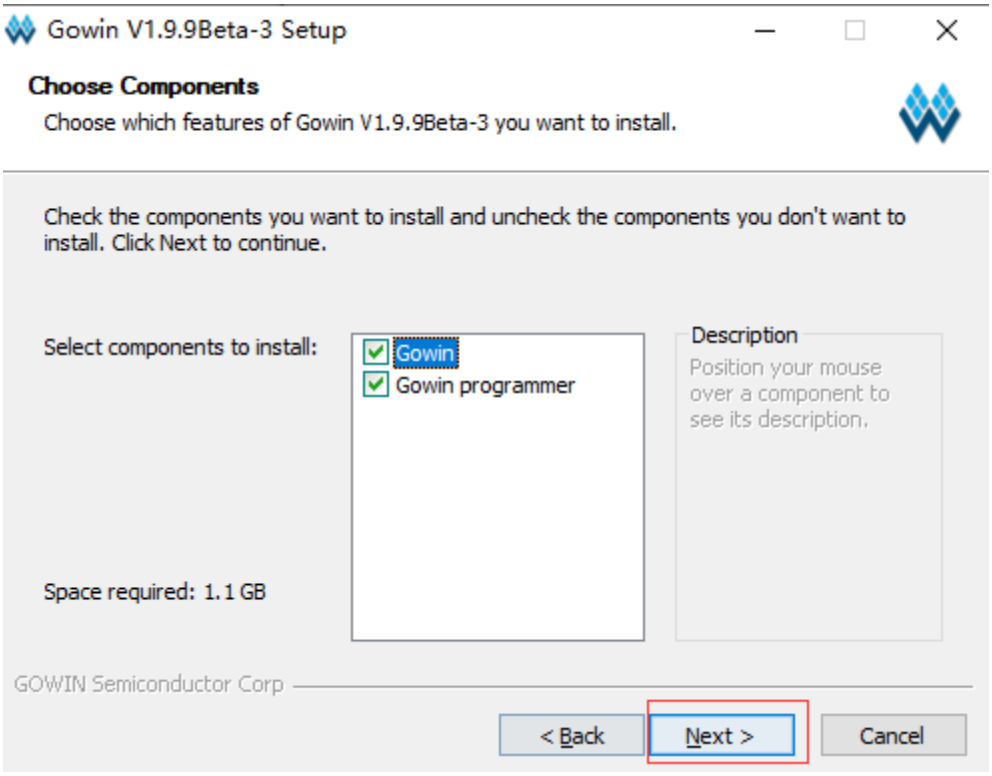


图 1-4 选择组件界面

然后弹出如下图 1-5 界面，点击“Browse”选择软件需要安装的路径（注意路径中不能包含中文），然后点击“Install”开始安装。

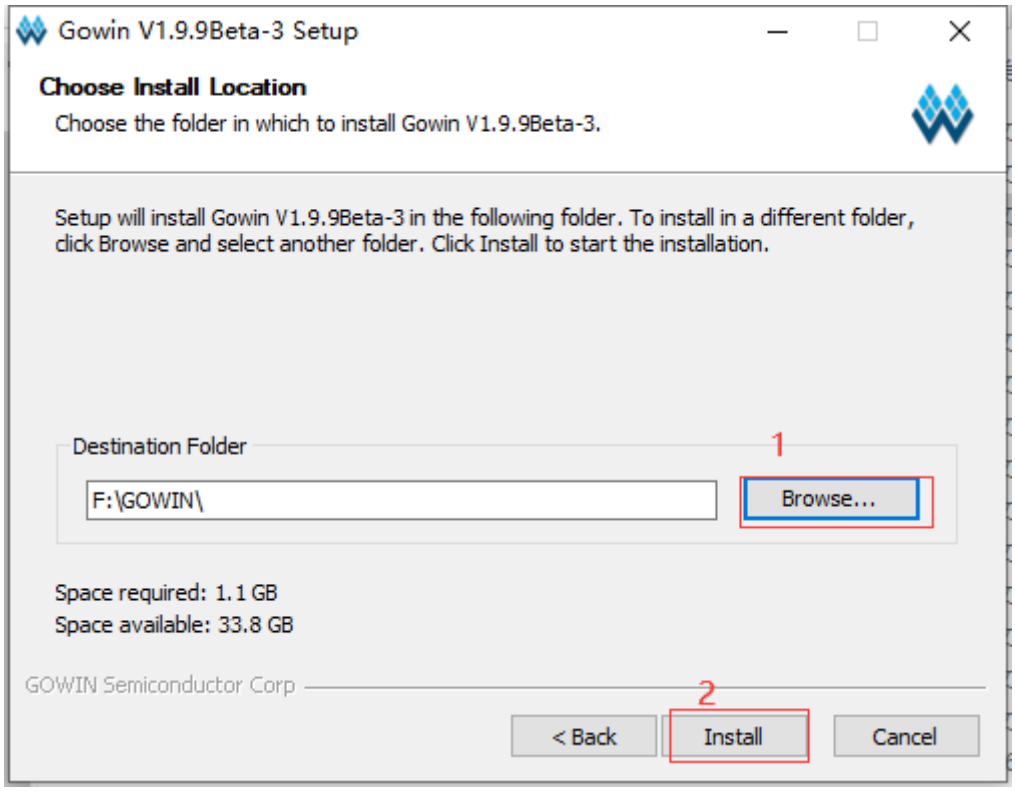


图 1-5 安装路径设置

点击“Install”之后，将会出现如下图 1-6 所示的界面，等到安装完成。

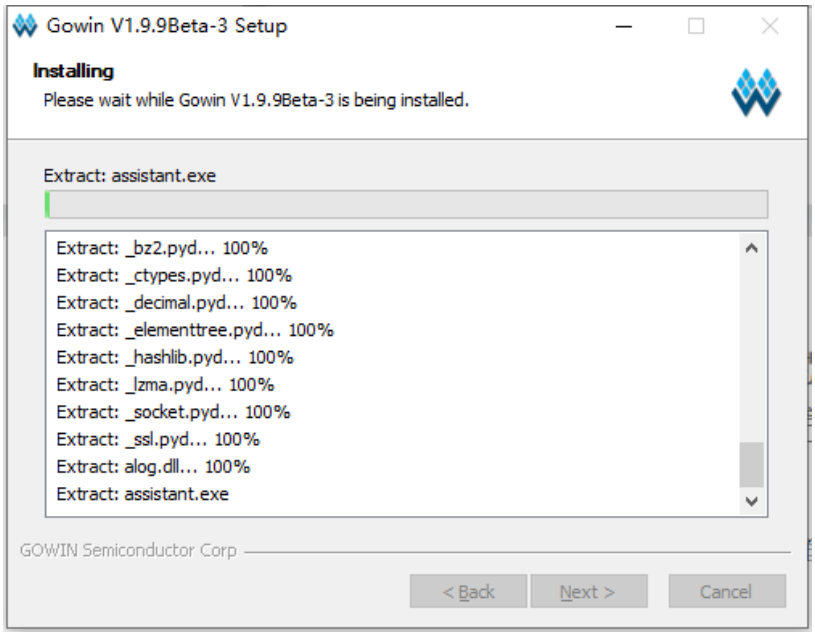


图 1-6 正在安装界面

安装完成之后，弹出如下所示的界面，点击“Finish”，完成 Gowin 软件的安装。

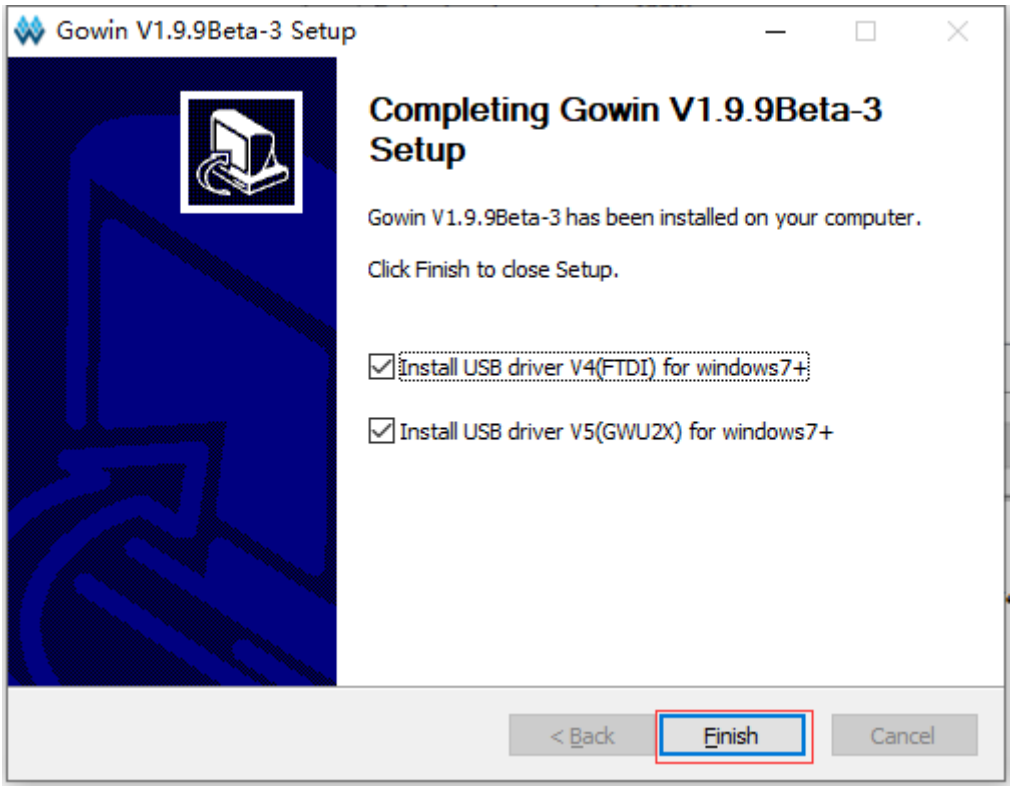


图 1-7 Gowin 软件安装完成

点击 Finish 之后，弹出如下图 1-8 所示的界面，点击“Extract”。

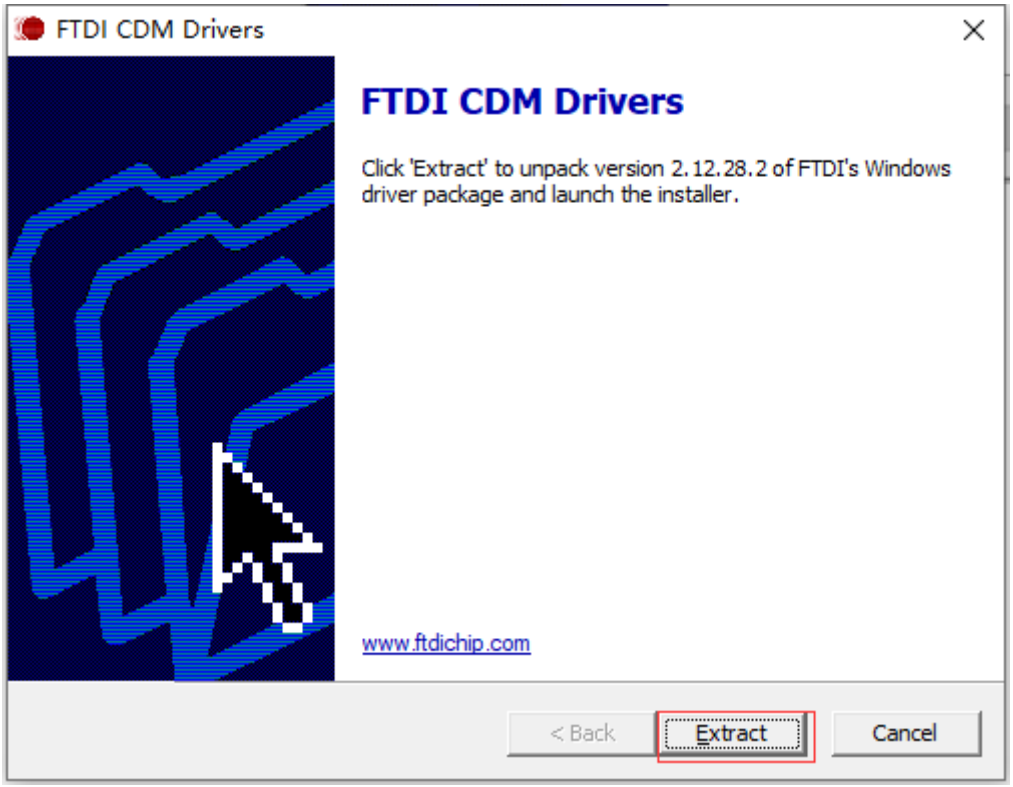


图 1-8 启动驱动安装界面

然后会弹出驱动安装向导，如下图 1-9 所示，点击“下一页”。

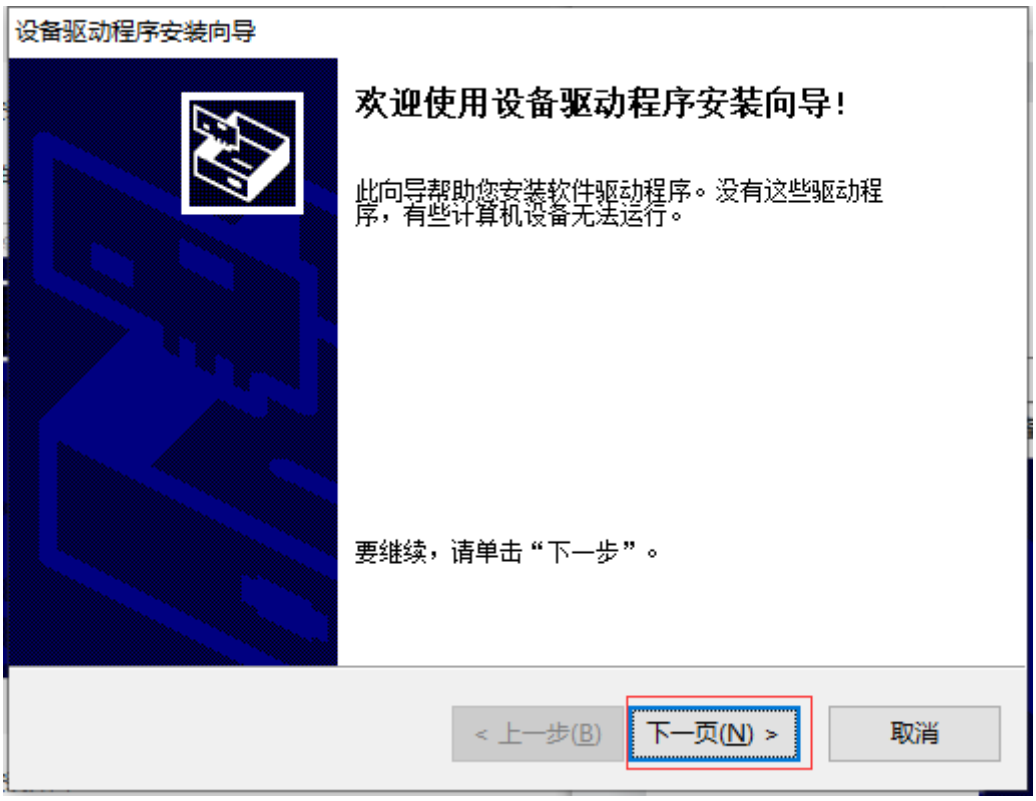


图 1-9 程序安装向导提示界面

随后弹出许可协议界面，如图 1-10 所示，依次点击“我接受这个协议”->“下一页”，

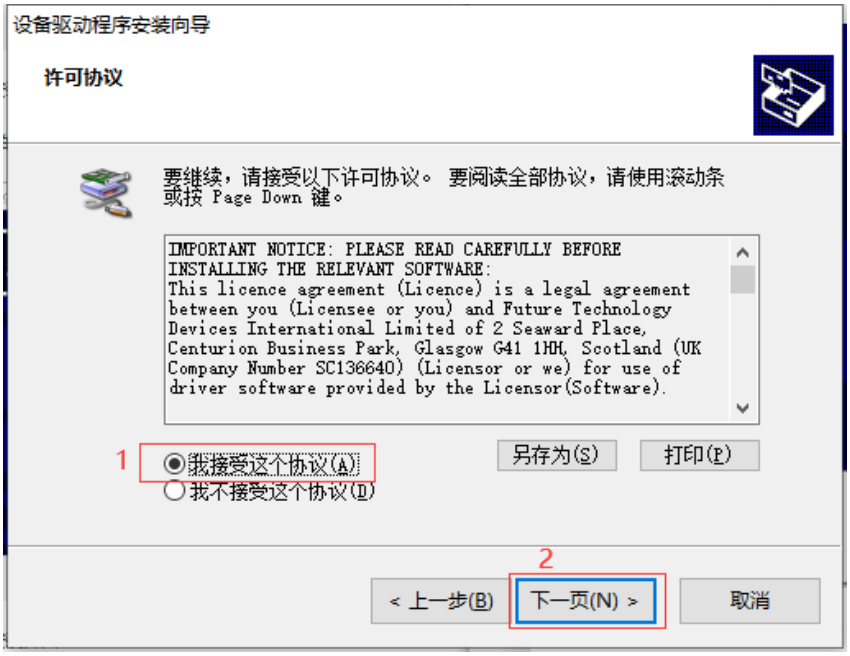


图 1-10 许可协议界面

最后弹出驱动安装完成界面，点击“完成”即可。

然后弹出如下图 1-11 所示的界面，点击 Browse 选择 GWU2X 文件安装路径，与我们之前软件安装路径一致。

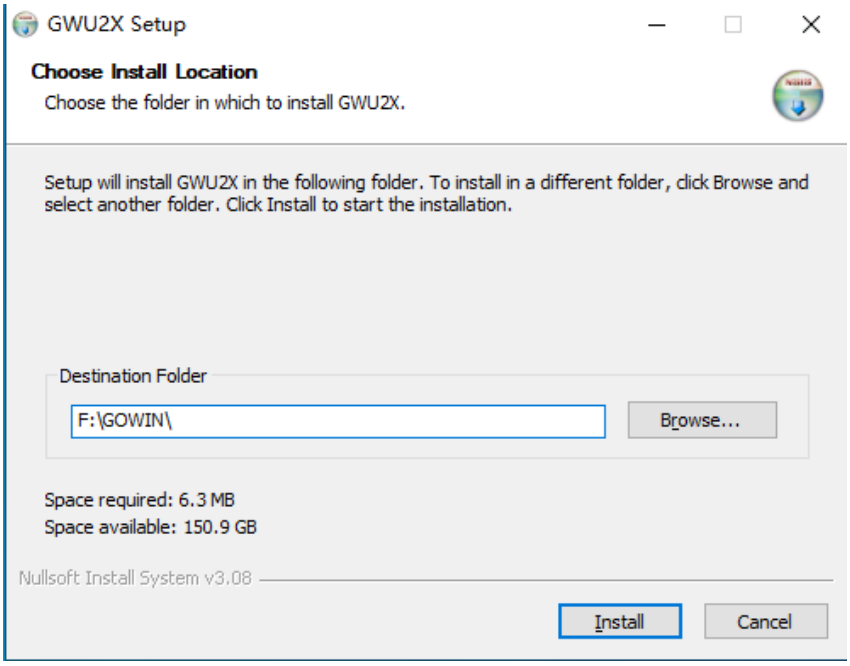


图 1-11 GWU2X 安装路径选择

然后等待安装完成，然后关闭界面，如下图 1-12 所示。

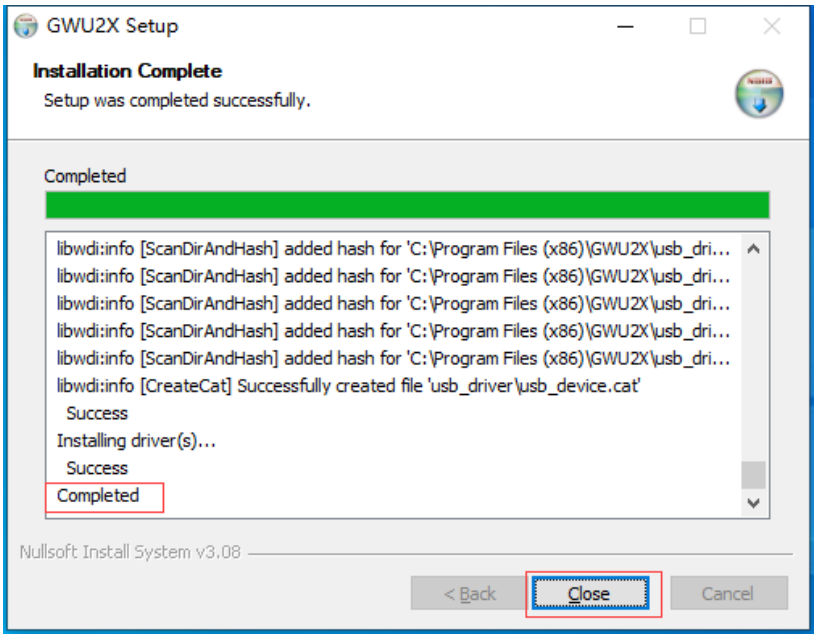


图 1-12 等待 GWU2X 安装完成

至此我们的 Gowin 软件就安装完成了。商业版的软件安装完成之后需要关联相应的 License 文件才能正常工作，为了方便管理 License 文件，我们将 License 文件文件放在 Gowin 软件的文件夹中，如图 1-13 所示为“License”文件存放位置。

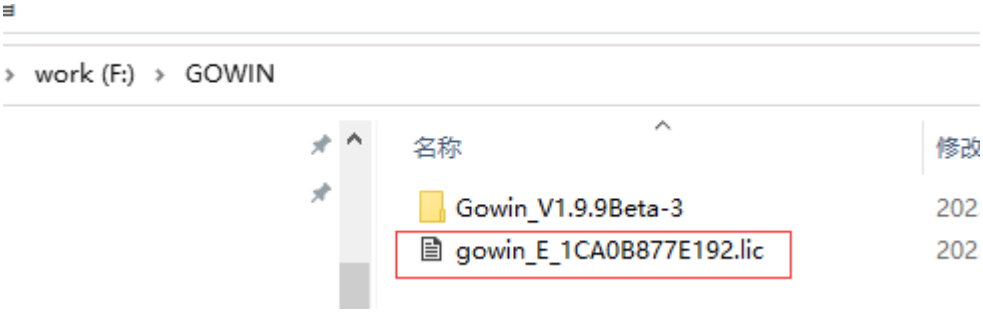


图 1-13License 文件存放位置

1.3 Gowin 软件中关联 Lincese

Gowin 软件安装完成之后，在桌面上会有 Gowin 软件的快捷方式，双击之后，弹出如下所示的界面，需要找到我们 License 的路径，进行关联。

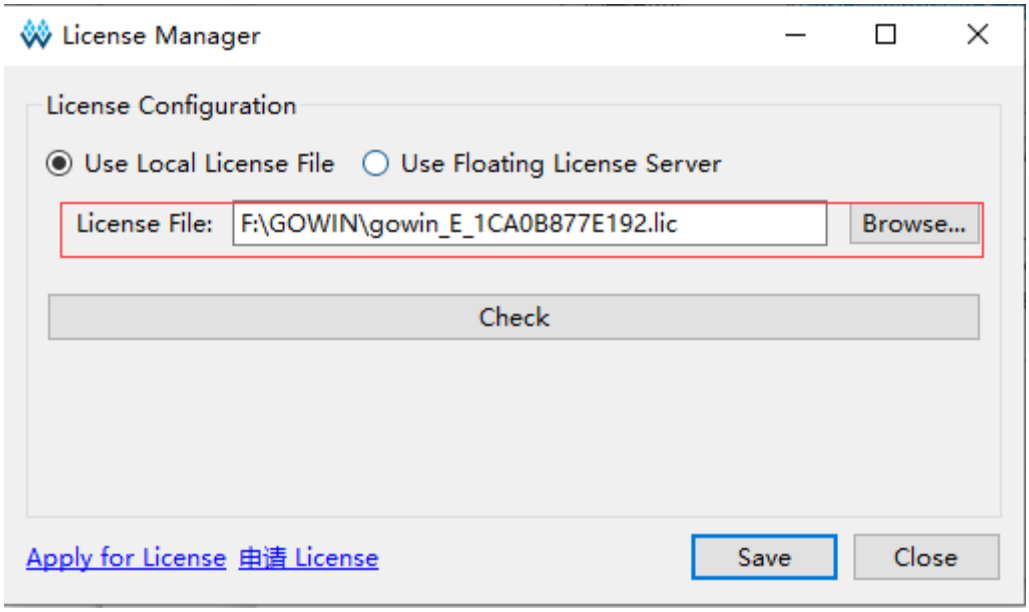


图 1-14 License 管理器中导入 Lincense 文件

Check 之后没有问题，点击 Save 保存，之后就可以打开软件使用了。

2 Gowin 软件基本开发流程

工程源码	|----02_设计实例 |----ch2_led_flash
相关视频课程	
说明	

章节导读

本章实验将带领用户熟悉一下 Gowin 开发软件的基本开发流程，整个流程包括建立工程、分析综合、物理约束、布局布线、板级验证及程序固化。

2.1 启动 Gowin 软件

Gowin 软件安装完毕之后会在你桌面建立快捷图片，如下图 2-1 所示。双击 Gowin 图标即可启动 Gowin 软件。

图 2-1 Gowin 桌面快捷方式

2.2 创建工程

创建设计工程的步骤主要包括如下步骤：

1. 启动 Gowin 集成开发环境，并进入到 Gowin 启动界面，如下所示。

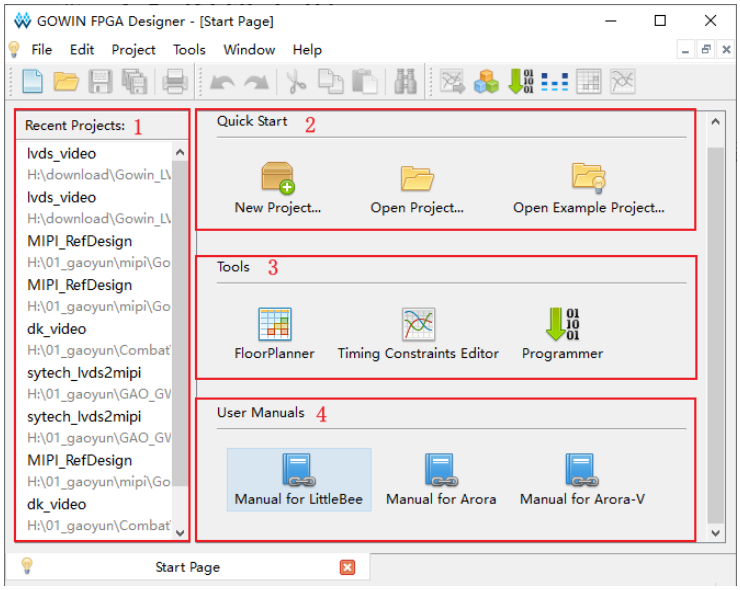


图 2-2 Gowin 集成开发环境的启动界面

在 Gowin 的启动界面分为 4 个板块，如下所示：

（1）Recent Projects：近期打开过的过程。

（2）Quick Start：快速开始，其中包括 New Project（创建一个新工程）、Open Project（打开已有工程）、Open Example Project（打开实例工程）

（3）Tools：工具，其中包括 FloorPlanner（物理约束编辑器）、Timing Constrains Editor（时序约束编辑器）、Programmer（数据流配置或下载）

（4）User Manuals：用户手册，点击可进入高云官网的产品介绍。

通过点击 Quick Start 中的 New Project 来新建一个 Gowin 设计工程。

2. 接着会打开创建新工程的向导，出现 New 对话框，如下图 2-3 所示，点击 OK 按钮。

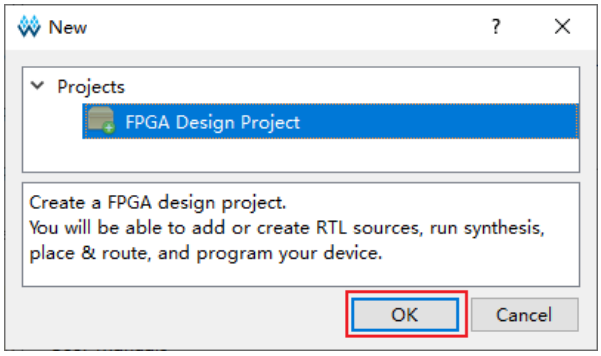


图 2-3 Gowin 创建工程导航界面

3. 进入 Project Name 对话框，如下图所示。填入工程名称以及工程路径，

需要注意的是，工程命名不能有中文字符和空格，否则在后续流程中会出现错误，文件路径也不能过长，windows 系统路径长度限制 260 个字符，Linx 系统限制 4096 个字符。根据实际情况确定是否勾选 Use as default project location（用作默认项目位置），勾选之后，下一次建立工程的时候，将会默认为此次设置的工程路径。设置完成之后，点击 next。

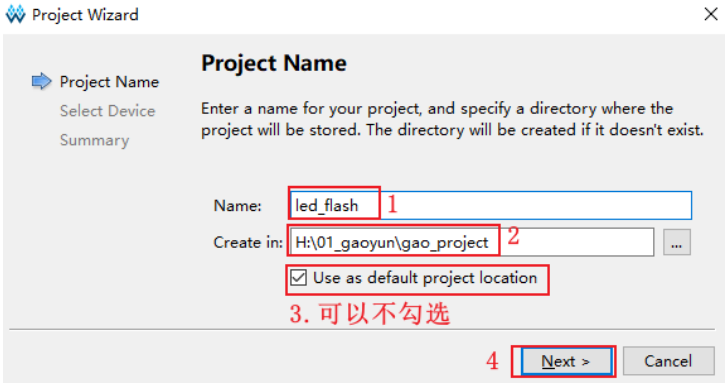


图 2-4 Gowin 创建功能 Project Name 界面

4. 随后出现 Select Device 对话框，如下图 2-5 所示。在这里我们需要选择设计所用的具体 FPGA 芯片型号，根据板卡所用 FPGA 的系列、封装、速度等级等来进行判断，用户也可以根据 FPGA 芯片上的字来进行判断。选择完成之后，点击 Next。

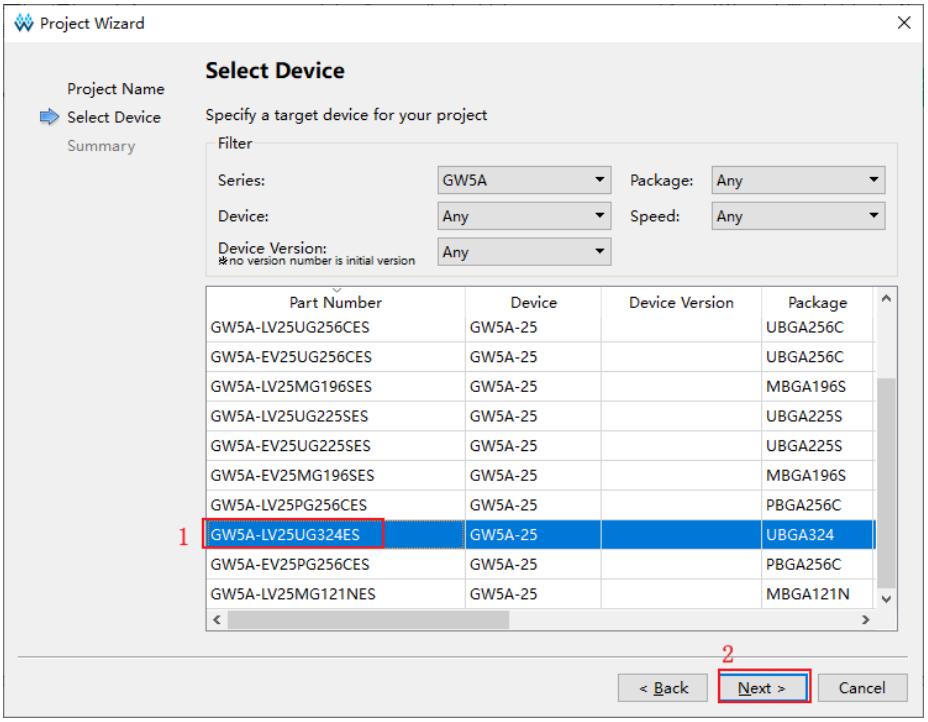


图 2-5 Gowin 选择 FPGA 器件类型

5. 最后出现 Summary 界面，如下图 2-6 所示，点击 Finish，完成工程的创建。

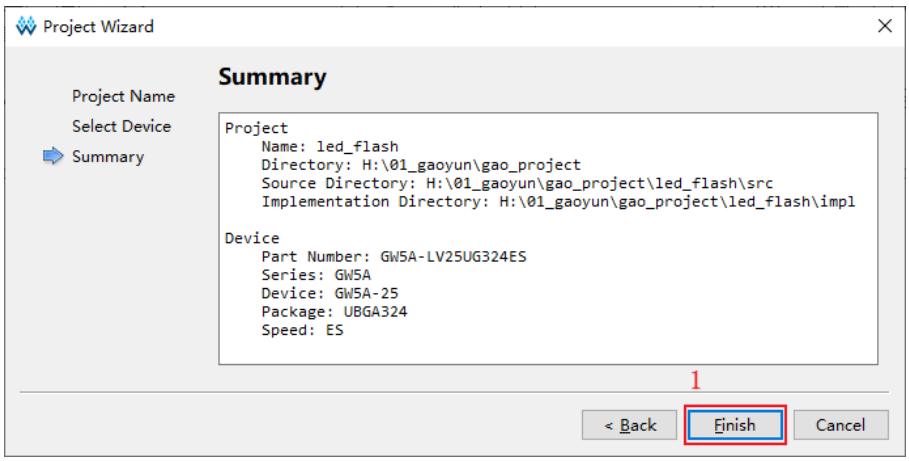


图 2-6 工程创建完成界面

6. 接下来我们会进入 Gowin 工程设计界面，如下图 2-7 所示。设计主界面主要包括 Design、Process、Hierarchy、Console 等模块。下面将带领大家熟悉 Gowin 工具设计界面。

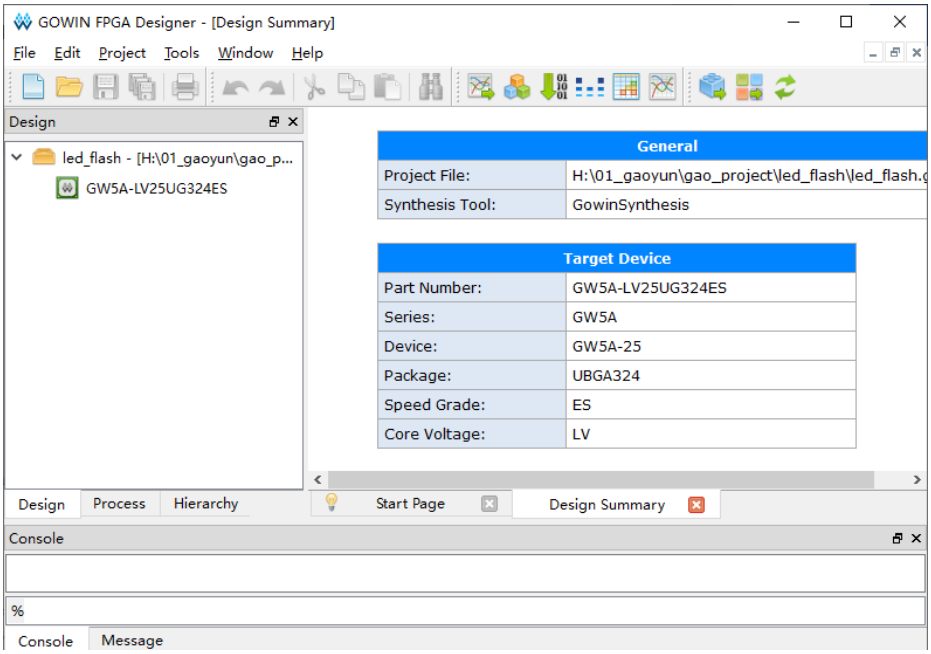


图 2-7 Gowin 工程设计界面

2.3 添加源文件

在使用 Gowin 工具完成工程创建之后，我们可以在 Gowin 中为工程添加源文件。接下来我们将通过创建一个 Verilog 源文件来演示该流程。

1. 通过点击工具栏的  或者依次点击菜单栏的 File->New…，点击之后如下所示，出现选择文件类型界面，这里我们选择 Verilog File 类型，然后点击 OK。

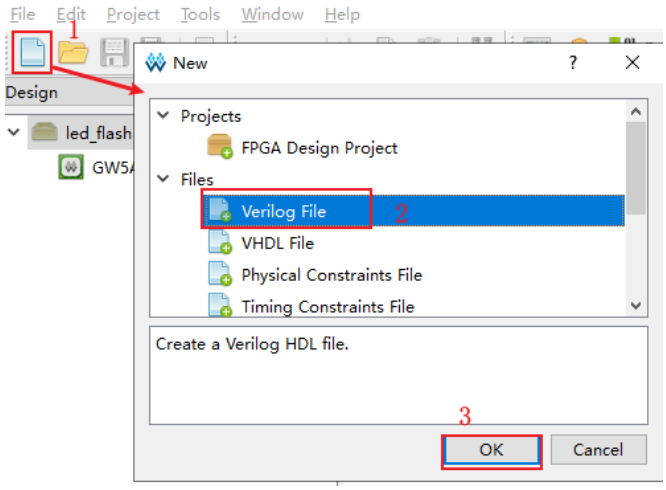


图 2-8 添加 Verilog 源文件

2. 随后出现 New Verilog file 界面，在该对话框下给文件命名，并选择文件保存路径，默认保存在工程目录文件夹的 src 文件夹中，然后勾选 Add to current project（添加到当前的工程目录中），如下图 2-9 所示。最后点击 OK。

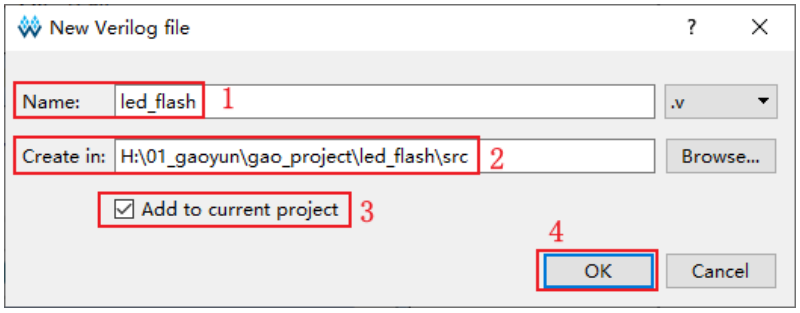


图 2-9 New Verilog file 配置界面

源文件添加完成之后，我们就可以看到在 Design 框中出现了新的文件夹 Verilog Files，该文件夹下就是我们刚刚新建的 Verilog 源文件，如下图 2-10 所示。

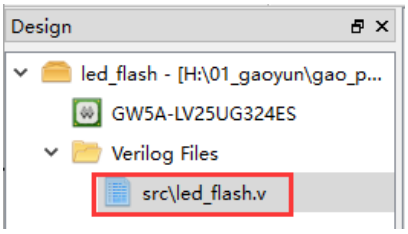


图 2-10 源文件添加完成示意图

3. 点击 led_flash.v 文件，编写本次实验工程代码，本次实验实现的功能就是翻转 LED 灯的状态，通过设计一个计数器来实现，如下：

```
module led_flash(
    input clk,
    input rst_n,
    output reg led
);
    reg [25:0] cnt;
    always @ (posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            cnt <= 26'd0;
        else if (cnt < 26'd49_999_999)
            cnt <= cnt + 1'b1;
        else
            cnt <= 26'd0;
    end

    always @ (posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            led <= 1'b0;
        else if (cnt == 26'd49_999_999)
            led <= ~led;
        else
        led <= led;
    end
endmodule
```

4. 编辑完成之后，Ctrl+S 保存源文件。至此源文件添加成功。

通过上述描述，完成了一个新的源文件建立，如果已经有编写好的 Verilog 源文件，可以通过点击工具栏中的  进行添加，点击之后，会弹 Open File 对话框，随后选择需要添加的文件，需要注意的是，这里需要添加的文件应该在该工程目录下。

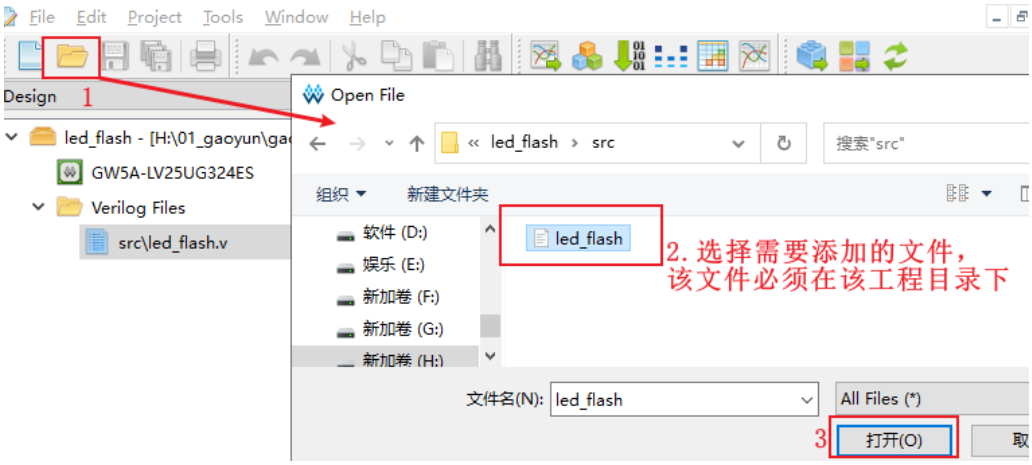


图 2-11 添加已有源文件

2.4 分析综合

我们可以依次点击 Process->Synthesize 进行分析综合，也可以直接点击工具栏中的  进行分析综合，通过分析综合后，生成基于 FPGA 底层资源的电路网表，如下图 2-12 所示，没有出现任何错误，则说明生成成功，如果出现错误，用户应该根据提示进行修改。

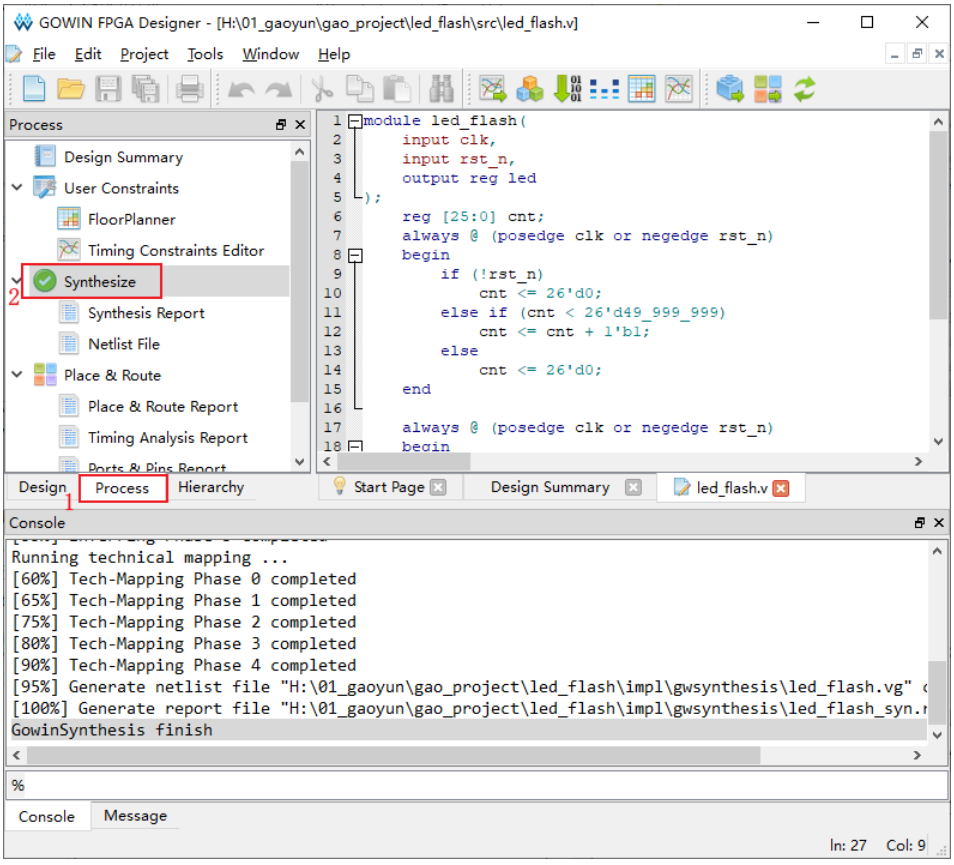


图 2-12 分析综合

综合完成之后，Hierarchy 窗口会自动显示当前工程设计中的资源信息，如下图 2-13 所示。

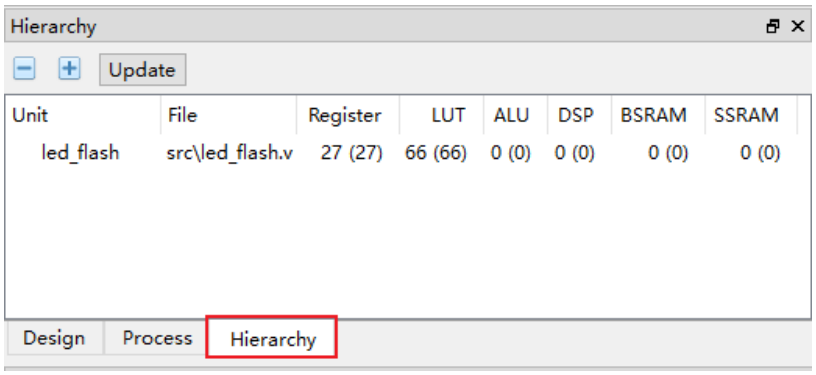


图 2-13 资源信息显示窗口

在 Hierarchy 窗口，我们将鼠标放置 led_flash.v 文件中，右击该文件，可以将该文件设置为顶层文件，如下图 2-14 所示。

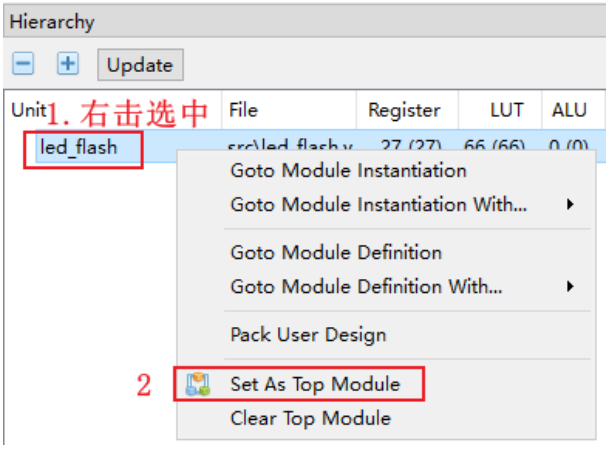


图 2-14 设置顶层文件

设置完成之后，该文件名称将会变成粗体，随后重新分析综合，便可以看到 Hierarchy 窗口如下图 2-15 所示。

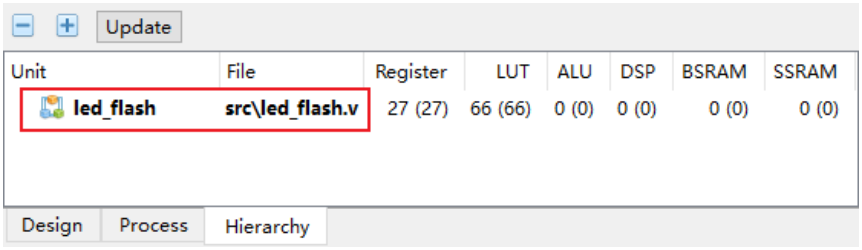


图 2-15 顶层模块设置成功

2.5 物理约束

我们可以通过点击 Process->FloorPanner 进行物理约束，也可以直接点击工具栏中的  ，如下图 2-16 所示。

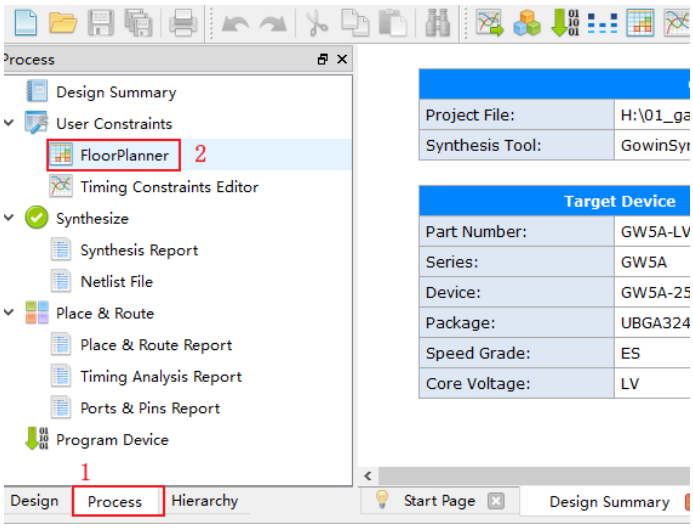


图 2-16 物理约束操作方式

随后弹出如下图 2-17 所示的对话框，由于我们没有新建一个.cst 文件用于存放物理约束，所以会弹出一个对话框，询问我们是否需要创建.cst 文件，这里直接点击 OK 即可。

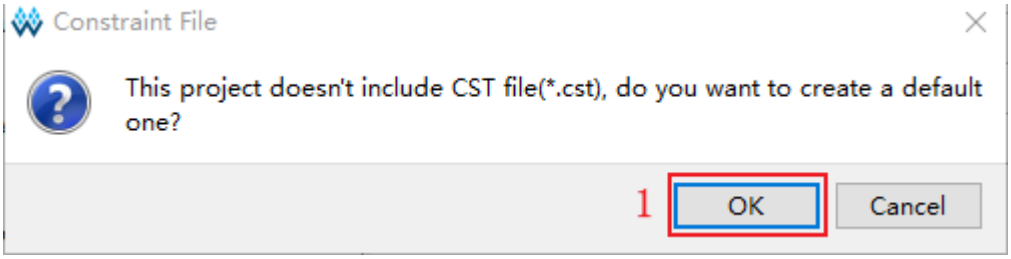


图 2-17 弹出是否新建物理约束文件对话框

进入 FloorPlanner 界面，点击 I/O Constraints，进行 I/O 约束，然后根据自己的板卡分配引脚和电平标准，如下图 2-18 所示。

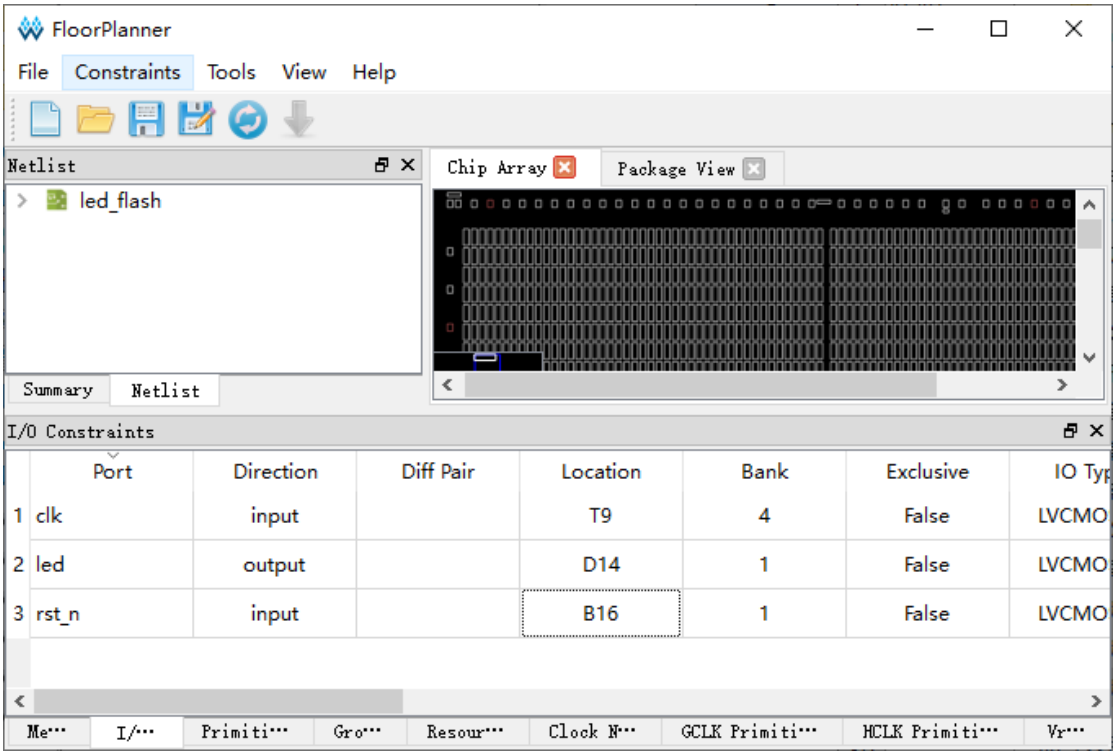


图 2-18 I/O 约束界面

约束完成之后，保存文件，如下图 2-19 所示。

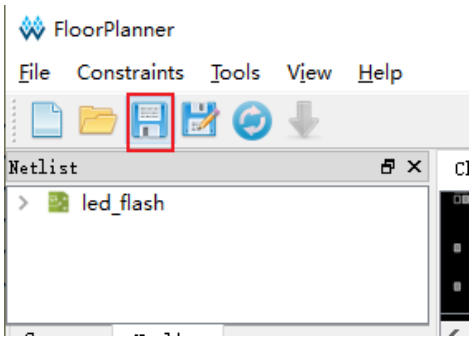


图 2-19 保存约束文件

约束完成之后，我们可以点击 Design->led_flash.cst 查看文件内容，如下图 2-20 所示。

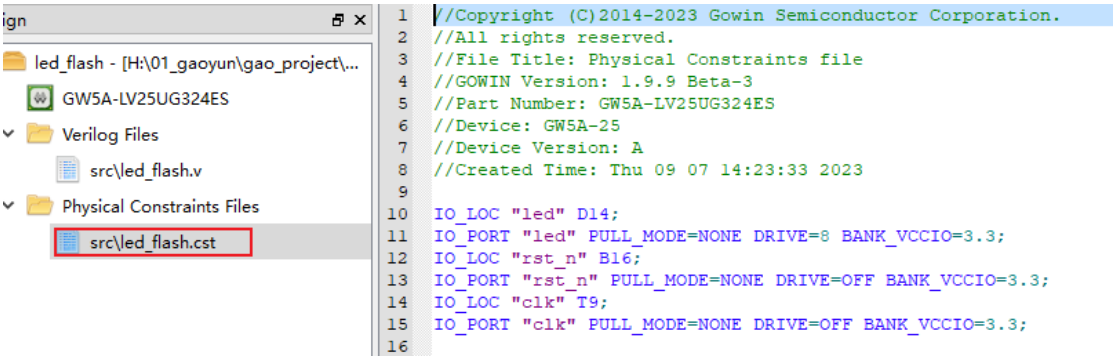


图 2-20 查看物理约束文件内容

从上图可以看出，文件中的内容和我们在 FloorPlanner 中设置的一致，说明我们物理约束成功。

2.6 布局布线

我们通过点击 Process->Place&Route 进行布局布线，布局布线成功之后我们便可以看到“Bitstream generation completed”，此时代表 bit 流数据生成成功，如下图 2-21 所示，如果失败，请用户根据 Console 中给的提示信息进行修改

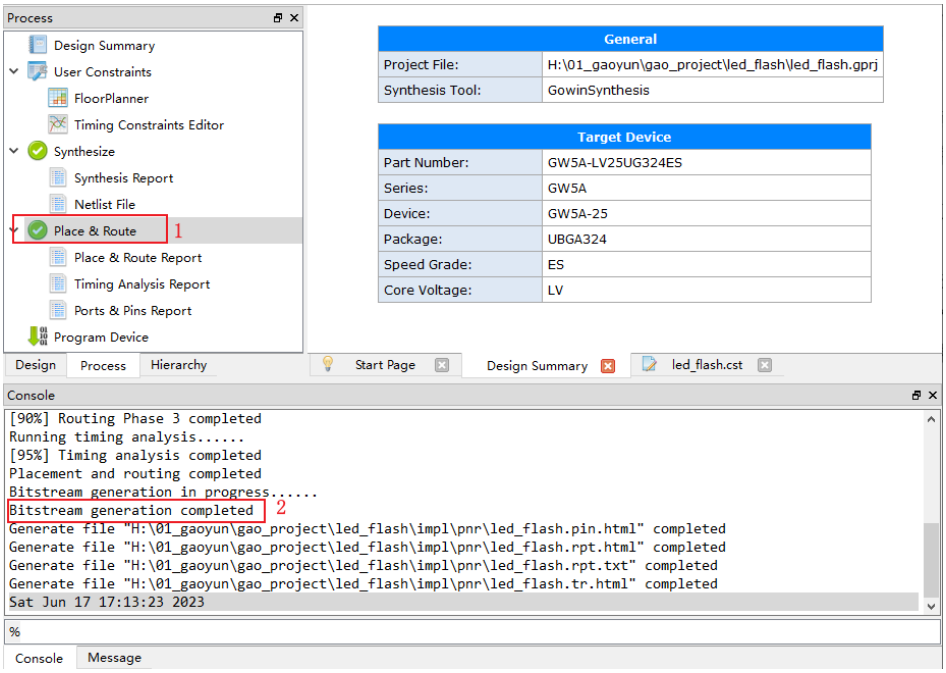


图 2-21 布局布线成功

2.7 板级验证

我们通过前面的步骤生成了实验所需的比特流文件，接下来就需要将比特流文件下载到 FPGA 芯片中，用于完成对 FPGA 的配置。

1. 依次将下载器、电源连接至开发板上，并打开电源，如下图 2-22 所示。

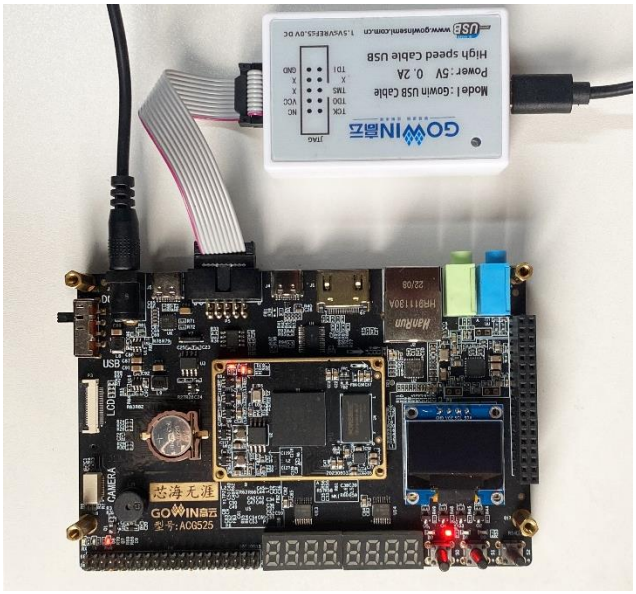


图 2-22 硬件连接图

2. 点击 Process->Program Device，弹出如下图 2-23 界面，说明检测到了设备，然后点击 save。如果显示 No USB Cable Connection，请检查自己的硬件连接。

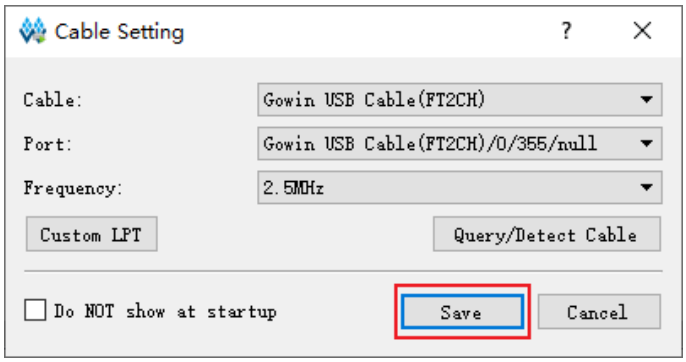


图 2-23 检测到设配提示

3. 点击 Gowin Programmer 中的  ，将比特流文件下载至 FPGA 芯片中，如下图 2-24 所示。下载成功之后，会打印出信息，如下图 2-25 所示。

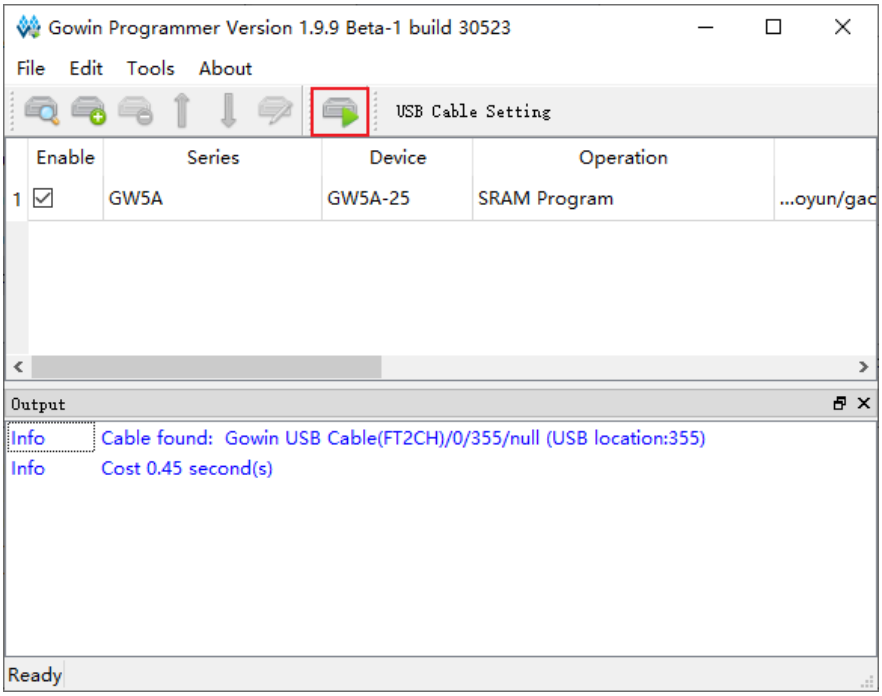


图 2-24 下载 bit 流文件

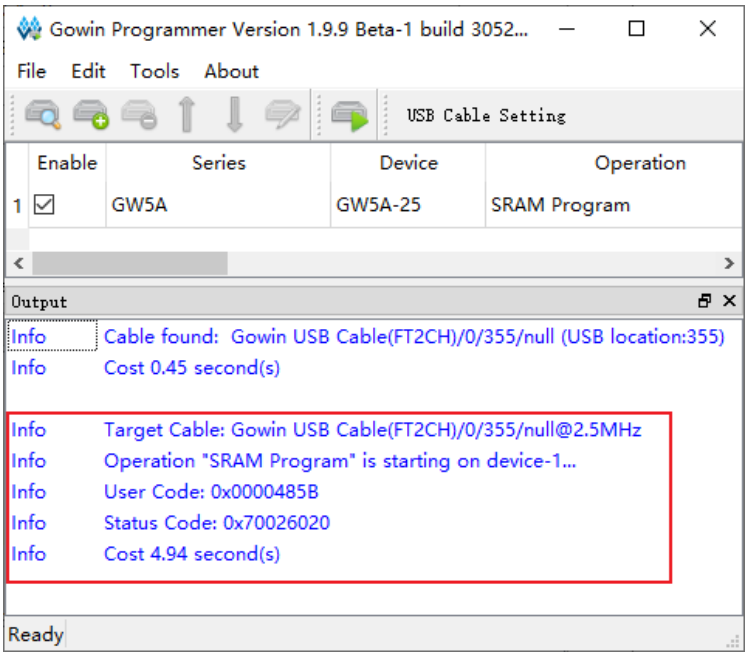


图 2-25 文件下载成功提示信息

4. 观察现象：此时我们可以看到开发板上对应的 LED 灯在闪烁，说明我们本次实验成功。

2.8 程序固化

我们通过前一小节下载的程序，在开发板重新上电之后，程序会丢失，为

了实现掉电不丢失，接下来将教大家如何进行高云 FPGA 芯片的固化。

1. 点击 Program Device，进入 Gowin Programmer 界面，双击 Operation 一栏中的 SRAM Program，进入 Device configuration 界面，如下图 2-26 所示。

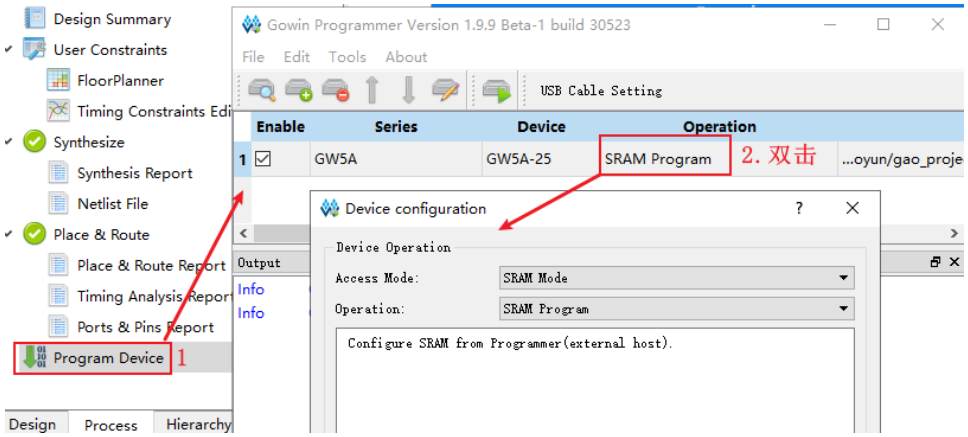


图 2-26 进入 Device configuration 界面操作示意

2. 在 Device configuration 界面，将 Access Mode 配置为 External Flash Mode 5AT，Operation 配置为 exFlash Erase,Program 5AT，然后点击 Save，如下图 2-27 所示。

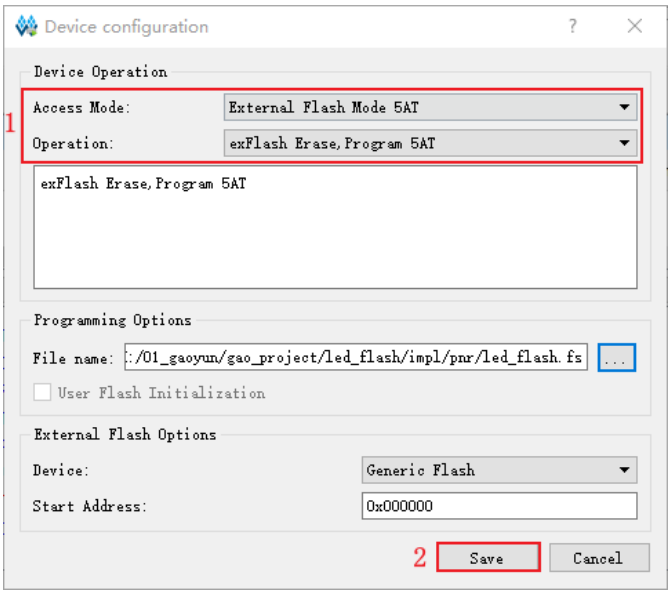


图 2-27 配置 Device configuration 界面

3. 点击 进行程序下载，下载成功之后，会打印出一下信息，如下图 2-28 所示。

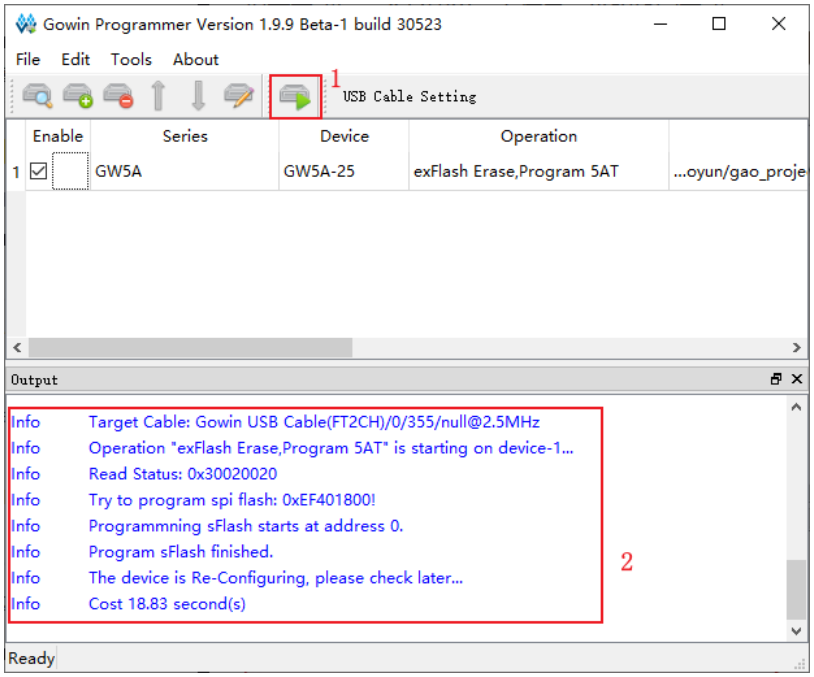


图 2-28 下载成功示意图

4. 然后重新给开发板上电，便可以看到开发板上运行的就是本次实验固化的程序。

2.9 思考与总结

本章实验带领大家熟悉了 Gowin 软件的基本开发流程，对于每个流程所应注意的细节，本章进行了一定的描述，对于一些遗漏的地方将会在后续章节使用到时补充说明。作为开始实验的第一个章节，本章手把手对整个设计流程进行了介绍，用户务必照着流程动手操作并熟悉整个过程，清楚每一步的目的和作用，避免因为步骤的缺失而导致工程报错，甚至是整个工程推倒重来。

3 Gowin 联合 Modelsim 仿真实验

工程源码	|----02_设计实例 |----ch3_led_flash_sim
相关视频课程	
说明	

章节导读

本章将带领大家学习并掌握 Modelsim 的仿真、如何在 modelsim 编译 Gowin 库，并且进行 Gowin 和 Modelsim 联合仿真。

3.1 Modelsim 编译 Gowin 器件库

Gowin 的 IP CORE 或原语仅在对应的开发平台里使用，第三方工具 Modelsim 无法获取 Gowin 的 IP CORE 内部的运行逻辑结果，因此无法直接进行仿真，需要进行 Gowin 库的编译。本章首先带领大家学习如何编译 Gowin 库，该步骤需要在第一次使用 Gowin FPGA 的时候进行，而编译出来的库后续可以随时使用，不用每次仿真之前再编译。

本章实验需要大家电脑上已经安装好了 Modelsim，如果还未安装 Modelsim，用户可以去我们的论坛上进行下载安装，下载链接如下：

【软件工具合集 2】各种各厂家 FPGA 开发软件下载地址

http://www.corecourse.cn/forum.php?mod=viewthread&tid=28768

3.1.1 新建 Modelsim 库

1. 打开 ModelSim 软件，依次点击菜单栏中【File】->【Change Directory】选项进入路径切换页面，如下图 3-1 所示。

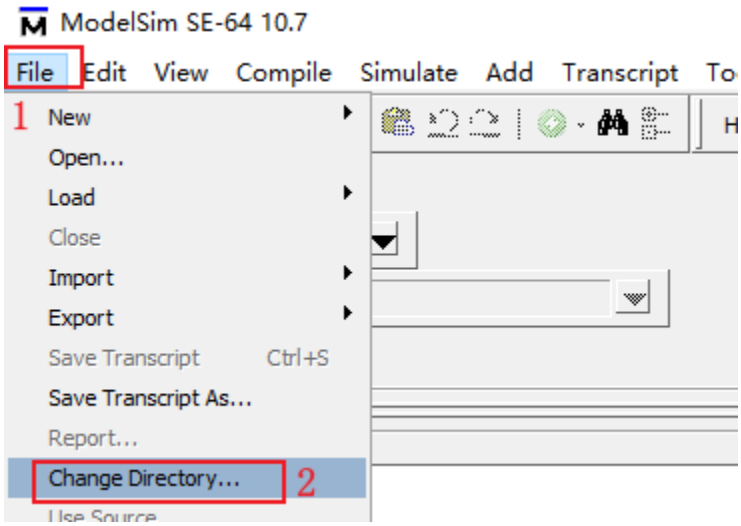


图 3-1 进入路径切换页面

2. 在打开的选择文件夹页面，定位到 Modelsim 软件安装目录，如笔者的是 G:\Modelsim，如下图 3-2 所示，随后点击选择文件夹。

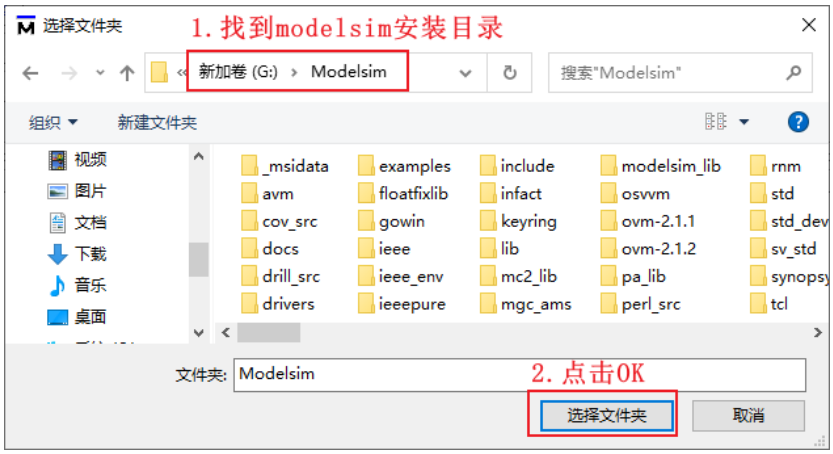


图 3-2 选择 Modelsim 安装目录文件夹

3. 在菜单栏依次点击【File】->【New】->【library】选项，仪打开新建库界面，如下图 3-3 所示。

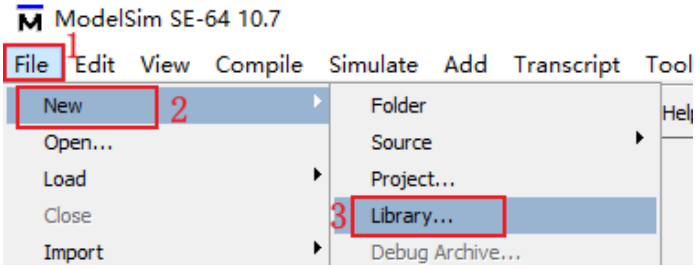


图 3-3 打开新建库界面

4. 在打开的界面中选择第一项“a new library”，“library Physical Name”

输入库名称，比如我们编译的是 Gowin 5A 系列的器件，取名为“gw5a”，然后点击 OK，如下所示。

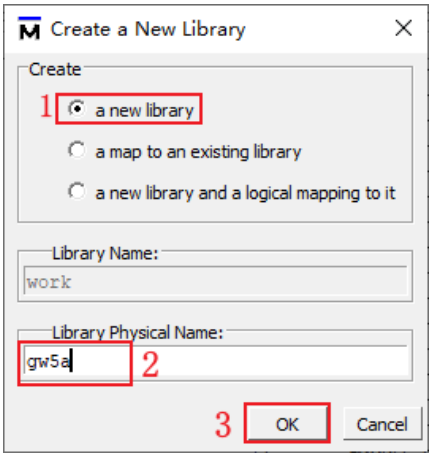


图 3-4 创建一个新库

5. 如果在 library 列表中出现了 gw5a（empty）库，且路径为前面设置的位置（工程根目录下），则表明库新建完成，如下图 3-5 所示。

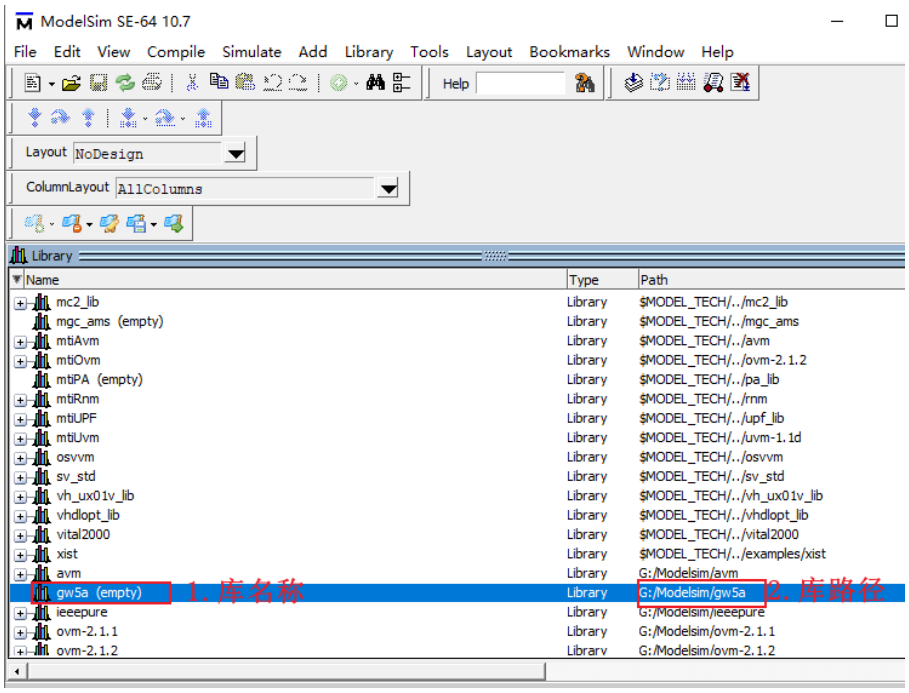


图 3-5 新库新建完成

3.1.2 编译 Gowin 的器件库文件

1. 在菜单栏中依次点击【Compile】->【Compile】以打开编译库文件界面，如下图 3-6 所示。

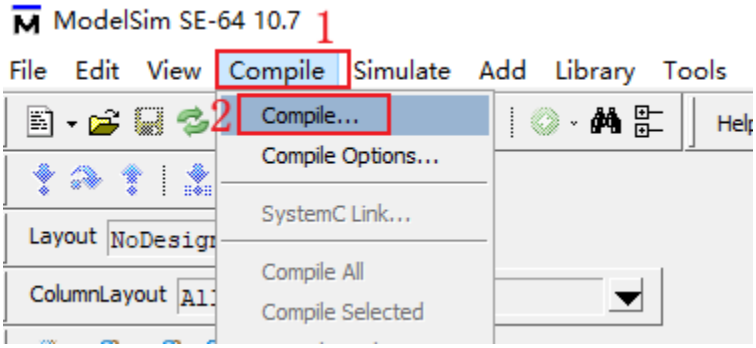


图 3-6 打开编译库文件界面

2. 在打开的界面，一定先下拉选择 library 为我们刚刚新建的 gw5a，然后修改文件路径为 Gowin 软件的 gw5a 系列器件的库路径，具体为：“ \Gowin\Gowin_V1.9.9Beta-1\IDE\simlib\gw5a”，如下图 3-7 所示。

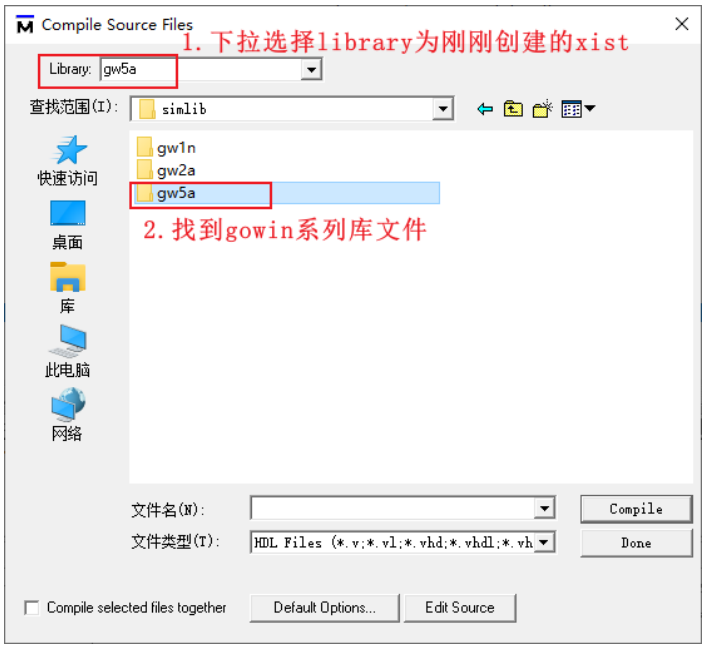


图 3-7 找到 Gowin 系列库文件

3. 在该路径下，找到 prim_sim.v 文件，然后点击 compile 按钮以开始进行编译，如下图 3-8 所示。此时 Modelsim 的 transcript 窗口会快速刷新各种编译过程信息，待编译完成后，点击 Done 按钮以完成编译。

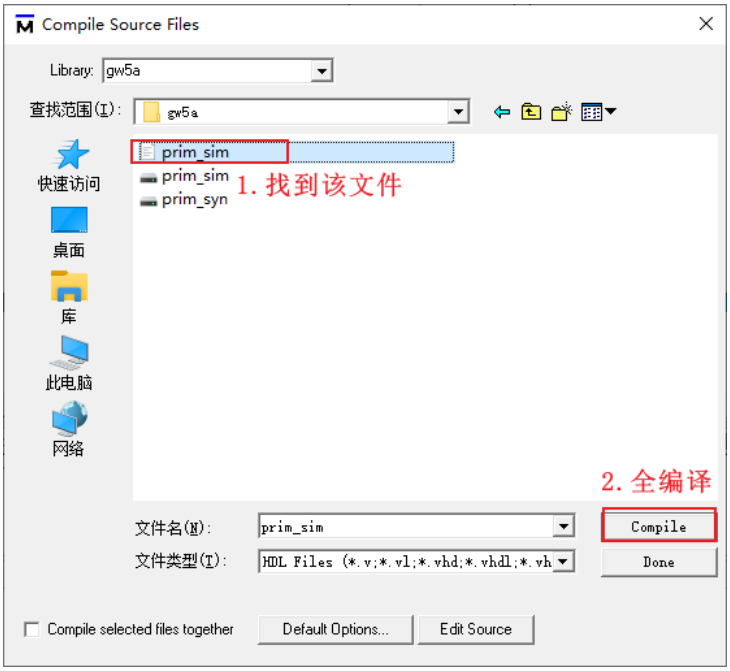


图 3-8 找到库文件进行编译

4. 编译完成之后，Transcript 界面会显示 0 错误，0 警告，并且 gw5a 这个这个库下面会出现各种元件，如下图 3-9 所示。

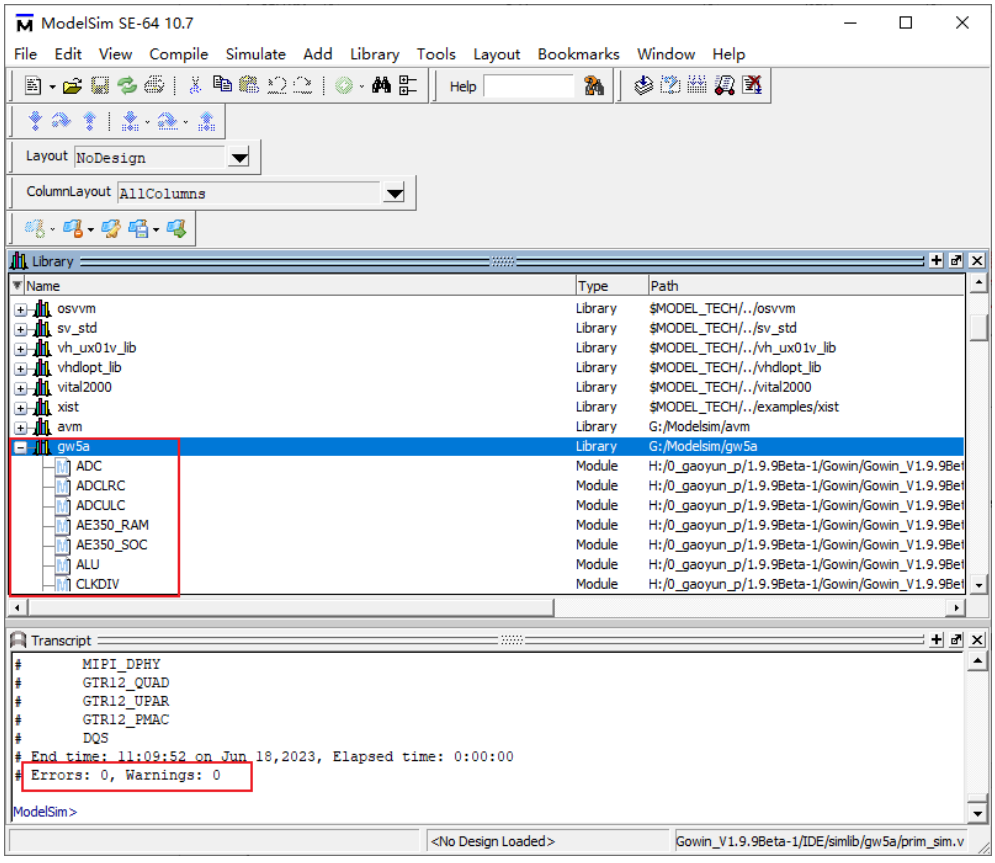


图 3-9 Gowin 系列器件库编译完成

3.1.3 添加库到 Modelsim 默认库列表

刚刚编译好的库，要想后续能够被 Modelsim 软件自动找到并添加使用，需要将其添加为 Modelsim 默认库之一。添加的方式很简单，如下所示：

1. 找到 Modelsim 安装位置，比如笔者电脑为 G:\Modelsim，在该路径下有个名为 modelsim.ini 的文件，这是一个只读文件，右键选中 Modelsim.ini 文件，点击属性，在属性设置界面，将其只读属性勾选去掉，如下图 3-10 所示。

图 3-10 去掉 modelsim.ini 的只读属性

2. 然后用记事本或者 notepad++d 打开文件，然后在该文件的 17 行左右，加上 gw5a 库的指定语句“gw5a = $MODEL_TECH/../gw5a”并保存，如下所示。

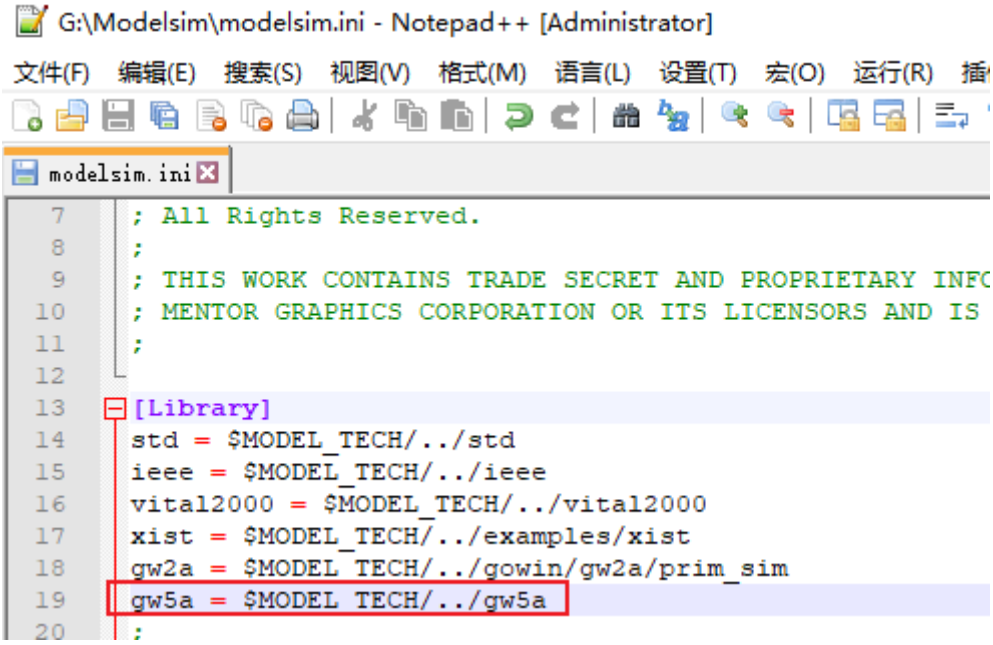


图 3-11 添加 gw5a 库文件路径

3. 恢复 modelsim.ini 的只读属性。

至此，关于如何使用 Modelsim 编译 Gowin 5A 系列 FPGA 的库文件方法就介绍完毕了，1N 和 2A 系列的器件同样可以参考上述的方式进行编译，下一步我们就可以使用 Modelsim 创建仿真工程并对设计工程进行仿真了。

3.2 Modelsim 仿真验证流程

在“Gowin 软件基本开发流程”一节中我们讲解了高云 FPGA 的基本开发流程，并点亮了 LED 灯使其闪烁，本节将在工程 led_flash 的基础上讲解高云 FPGA 联合 Modelsim 进行仿真验证的流程。

3.2.1 编写 Verilog 仿真设计代码

1. 首先打开工程，点击 新建一个 Verilog File，或者 Ctrl+N 快捷键，如下图 3-12 所示。

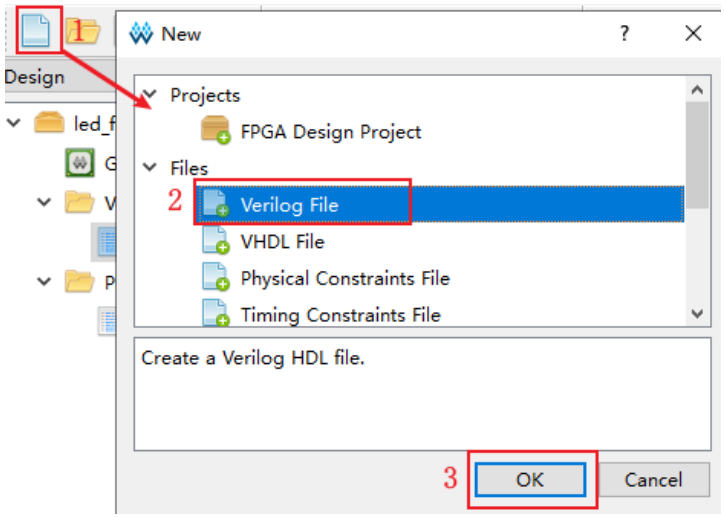


图 3-12 新建一个 Verilog File

2. 然后在弹出的 New Verilog File 中给文件命名为 led_flash_tb，默认文件保存路径不变，如下图 3-13 所示。

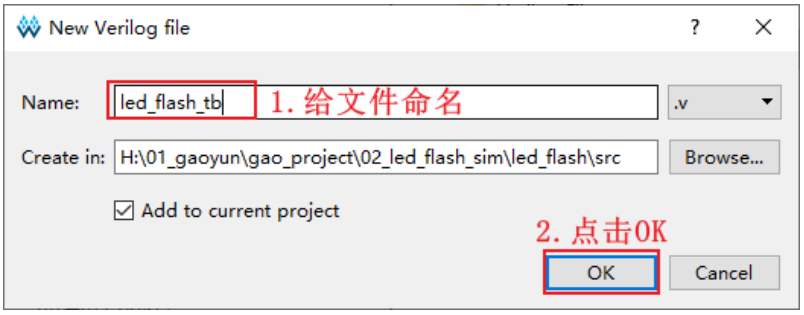


图 3-13 给仿真文件命名

3. 在打开的文件中输入代码内容，LED 灯闪烁的仿真文件，可以使用下述代码：

```
`timescale 1ns/1ns
module led_flash_tb();

    reg Clk;
    reg Rst_n;
    wire [3:0]Led;

    led_flash led_flash(
        .clk(Clk),
        .rst_n(Rst_n),
        .led(Led)
    );

    initial Clk = 1;
    always#10 Clk = ~Clk;
```

```
    initial begin
        Rst_n = 0;
        #201;
        Rst_n = 1;
        #200;
        #100000000;
    end

endmodule
```

4. 保存完 led_flash_tb 文件之后，进行分析和综合，检查是否有语法错误。

仿真文件设计完成后，我们就可以利用 Modelsim 进行工程的功能仿真测试。

3.2.2 建立 Modelsim 工程并添加仿真文件

1. 打开 Modelsim，在 Modelsim 建立一个新的 project。选择 File->New->project，如下图 3-14 所示。

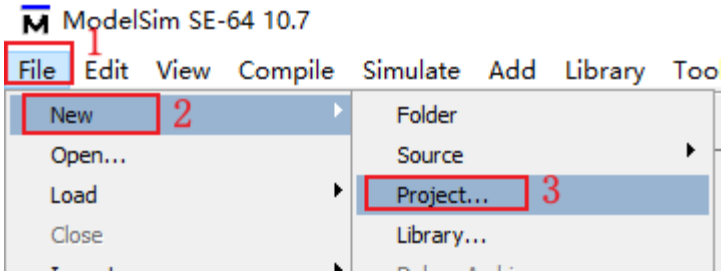


图 3-14 建立 Modelsim 工程

2. 进入建立新工程的设置界面，在“Project Name”一栏中填写工程名。这里我们把工程命名为相对应的工程名“led_flash”，“Project Location”是工程路径，点击“Browse”进行设置，根据需要我们把仿真工程保存到我们设计工程下，新建一个 sim 文件夹并选择，如下图 3-15 所示。

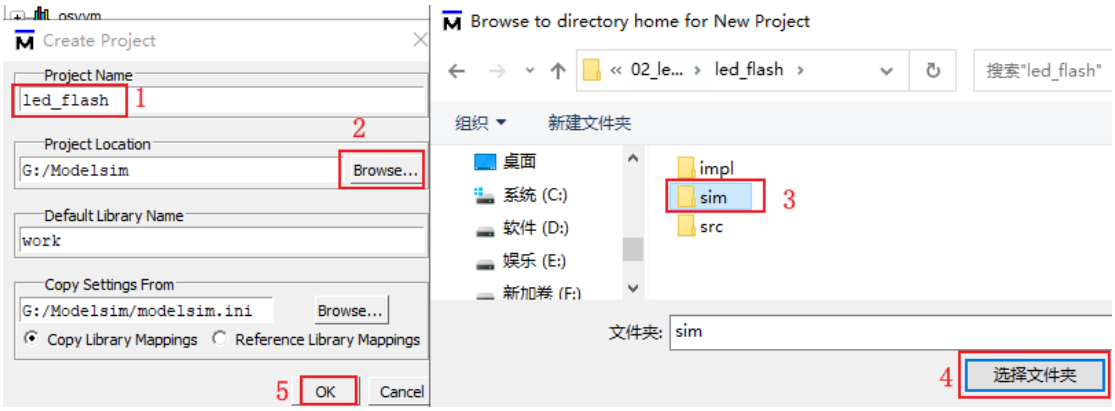


图 3-15 Modelsim 工程设置

3. 点击 OK 之后，出现如下图 3-16 所示的界面，从图中可以看出，有四种操作可以选择：Create New File（创建新文件）、Add Existing File（添加已有文件）、Create Simulation（创建仿真）和 Create New Floder（创建新文件夹），这里选择“Add Existing File”。

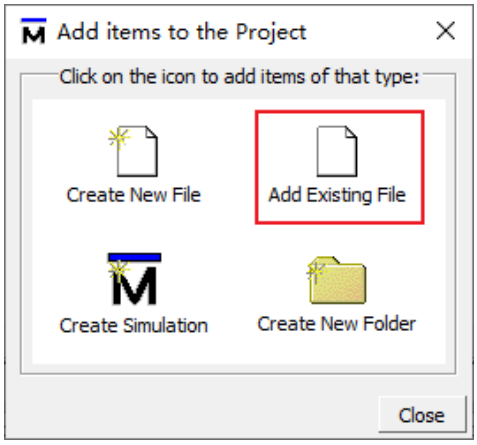


图 3-16 项目中添加文件对话框

4. 随后在弹出的对话框中，选择 Browse 选择本次实验需要添加的工程设计文件“led_flash.v”、“led_flash_tb.v”，如下图 3-17 所示。

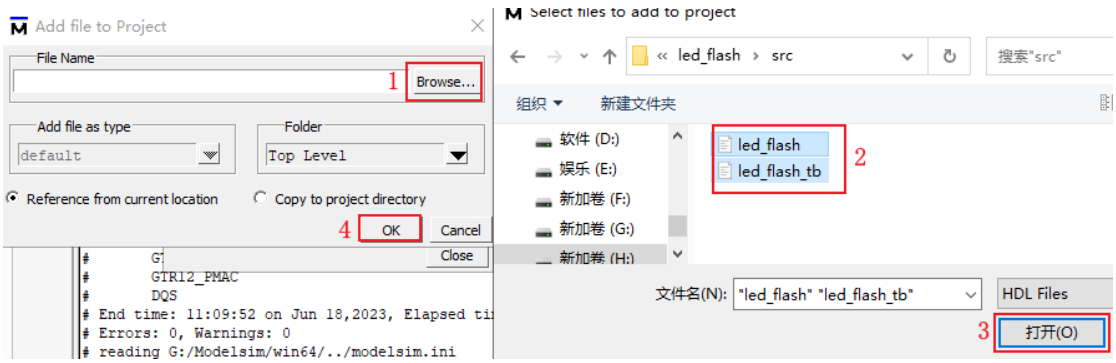


图 3-17 选择文件进行添加

添加完成之后，可以在 Modelsim 软件中看到我们添加的工程文件，如下图 3-18 所示。

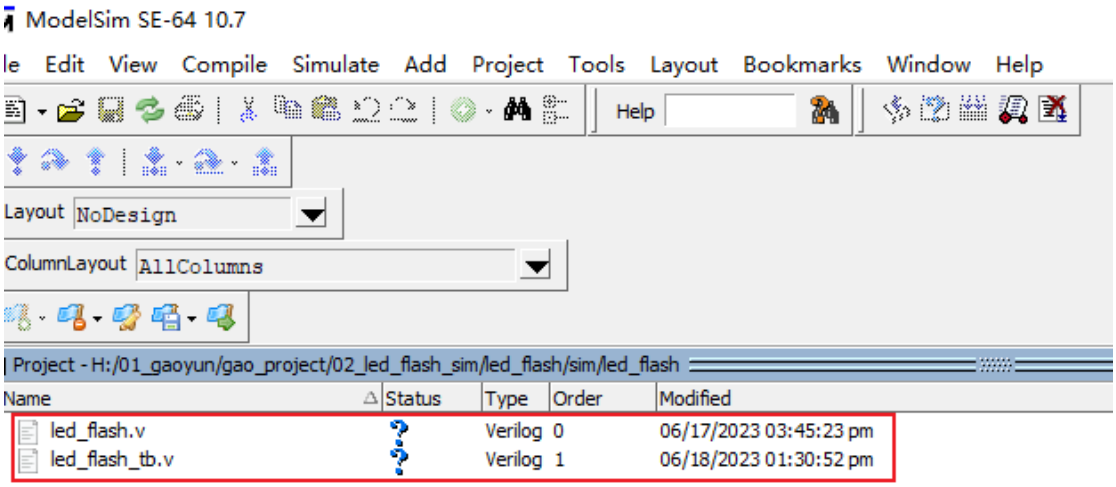


图 3-18 文件添加成功之后示意图

3.2.3 编译仿真文件

编译的方式有两种：第一种 Compile Selected（编译所选文件），编译所选文件功能需要先选中一个或几个文件，执行命令之后可以完成对选中文件的编译；第二种是 Compile All（编译全部文件），该命令是按编译顺序对工程中的所有文件进行编译。这里，我们选择编译所有文件，选择 Compile->Compile All，如下图 3-19 所示：

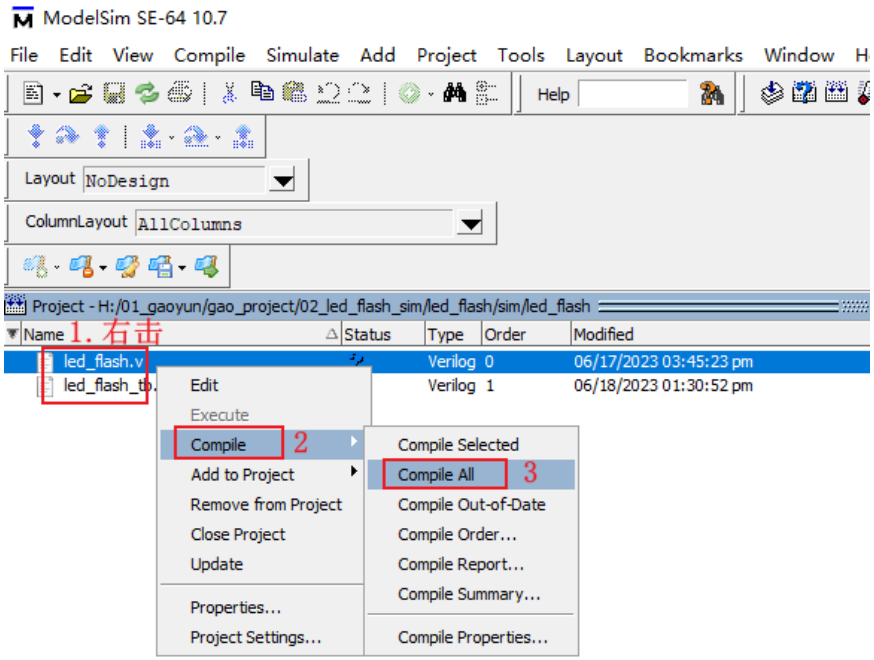


图 3-19 编译所有工程文件

编译完成之后，文件后面的 Status 从“？”变为“√”表示编译通过。下方的“Transcript”也提示文件编译成功，如下图 3-20 所示。

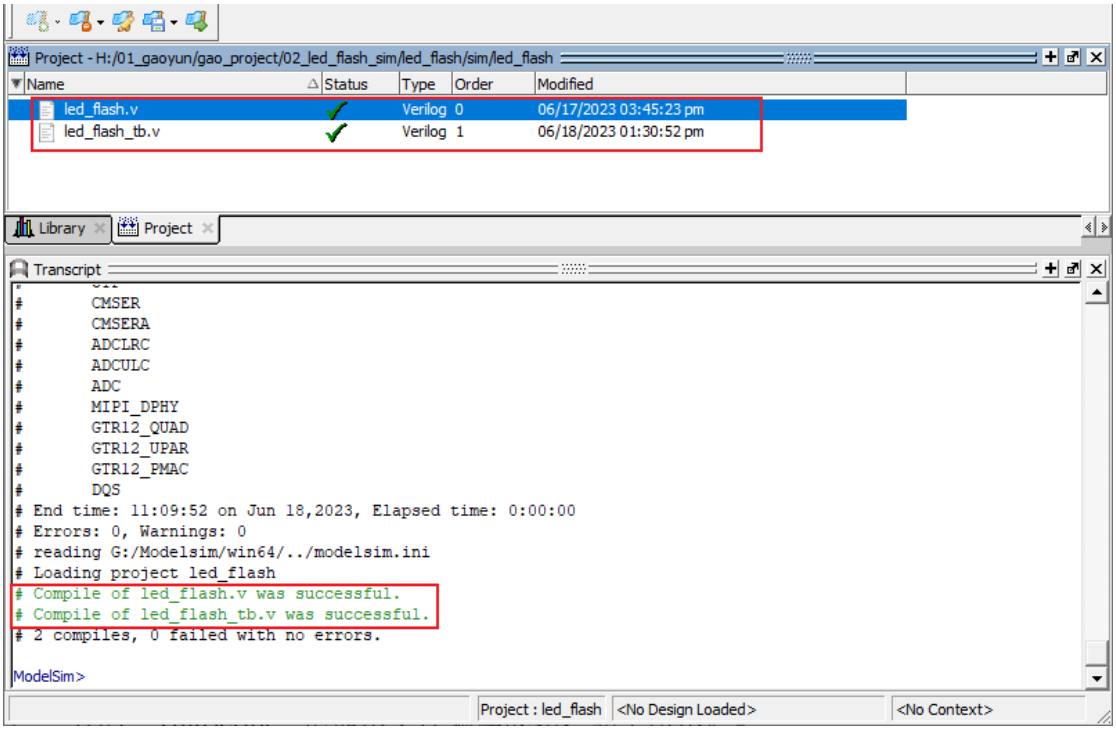


图 3-20 文件编译成功示意图

需要注意的是，这两个设计中不希望出现的状态：编译错误（显示红色的“×”）和包含警告的编译通过（对号的后面会出现一个黄色的三角符号）。编译错误即 Modelsim 无法完成文件的编译工作。通常这种情况是因为被编译文件中包含明显的语法错误，Modelsim 会识别出这些语法错误并提示使用者，使用者可根据 Modelsim 的提示信息进行修改。

3.2.4 配置仿真环境

编译完成后，接下来我们就开始配置仿真环境。

1. 我们在 project 状态栏中右键点击，然后选择“Add to Project”->“Simulation Configuration…”并点击，如下图 3-21 所示。

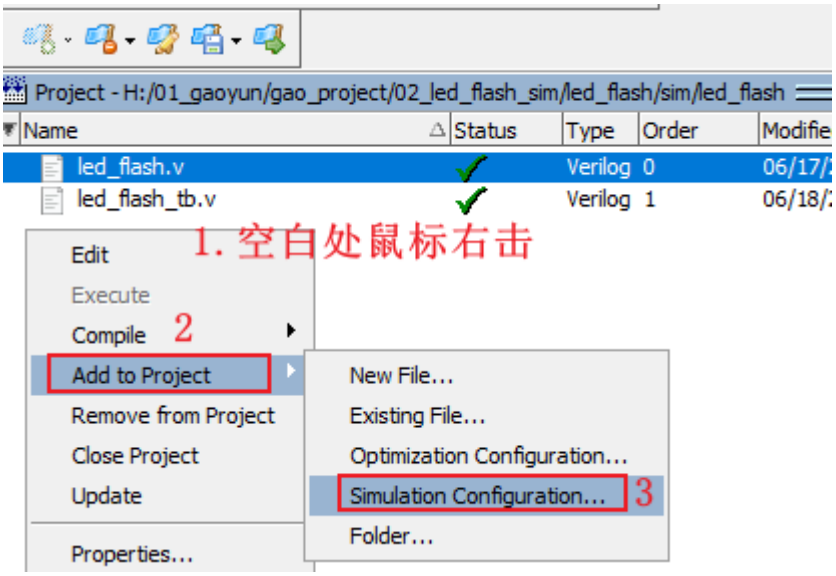


图 3-21 进入配置仿真环境操作示意图

2. 进入 Add Simulation Configuration 页面，我们在 Design 标签页面中选择 work 库中的“led_flash_tb”模块作为设计顶层，点击复制模块名作为仿真配置“Simulation Configuration Name”的命名，确保命名保持一致，如下图 3-22 所示。在复杂的工程设计中，我们可以设计多个不同的仿真配置顶层对工程进行仿真测试。

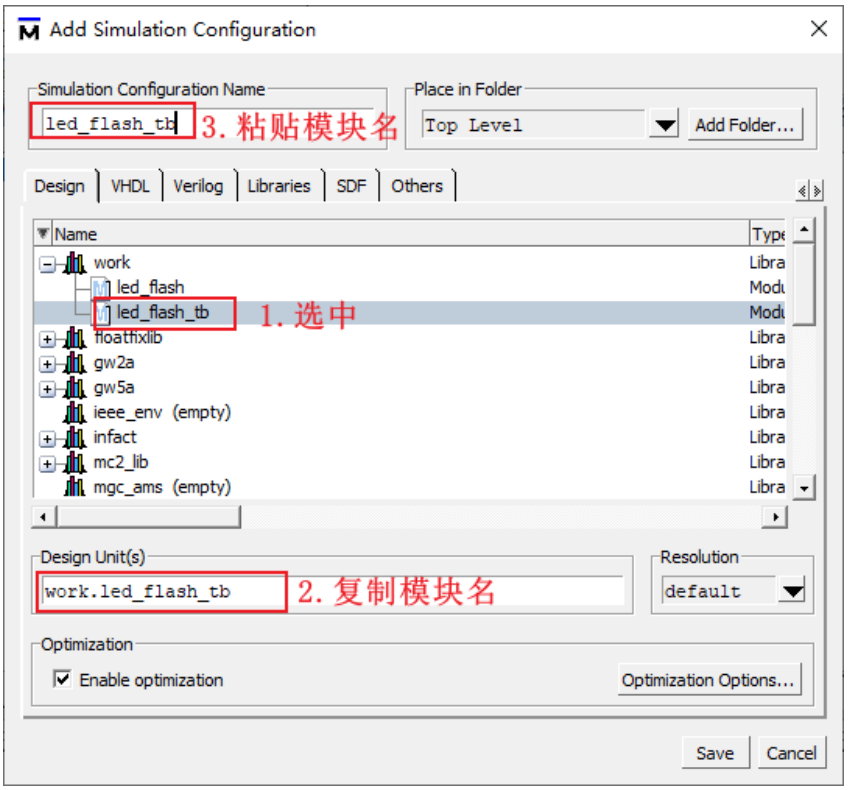


图 3-22 仿真环境界面配置一

3. 点击“Optimization Options…”，在“Optimization Options..”设置栏中选择“Apply full visibility to all modules(full debug module)”，点击“OK”，如下图 3-23 所示。

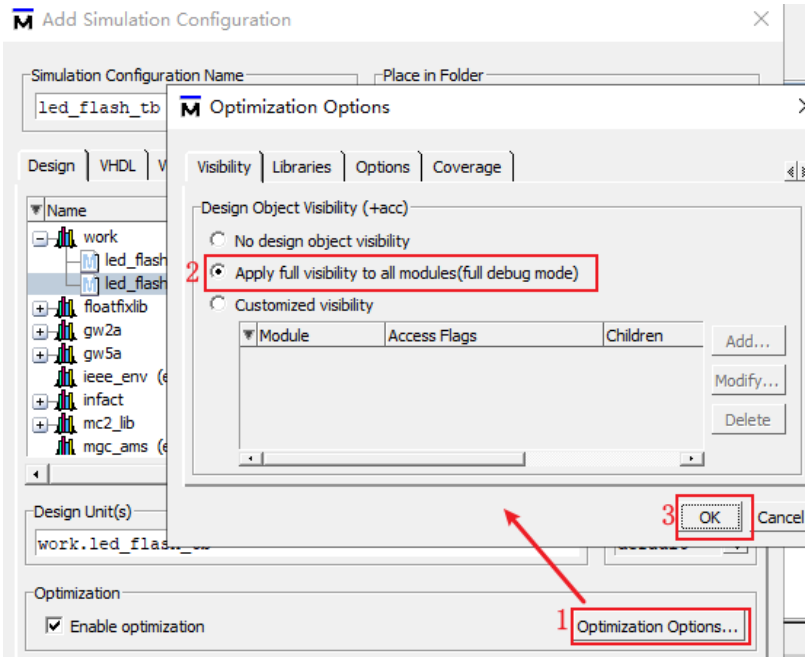


图 3-23 仿真环境界面配置二

4. 点击“libraries”设置栏，在“Search libraries(-L)”一栏中点击“Add…”添加我们新建的高云的库文件“gw5a”，在“Search Libraries First(-Lf)”同样选择库文件“gw5a”，最后点击“Save”保存设置，如下图 3-24 所示。

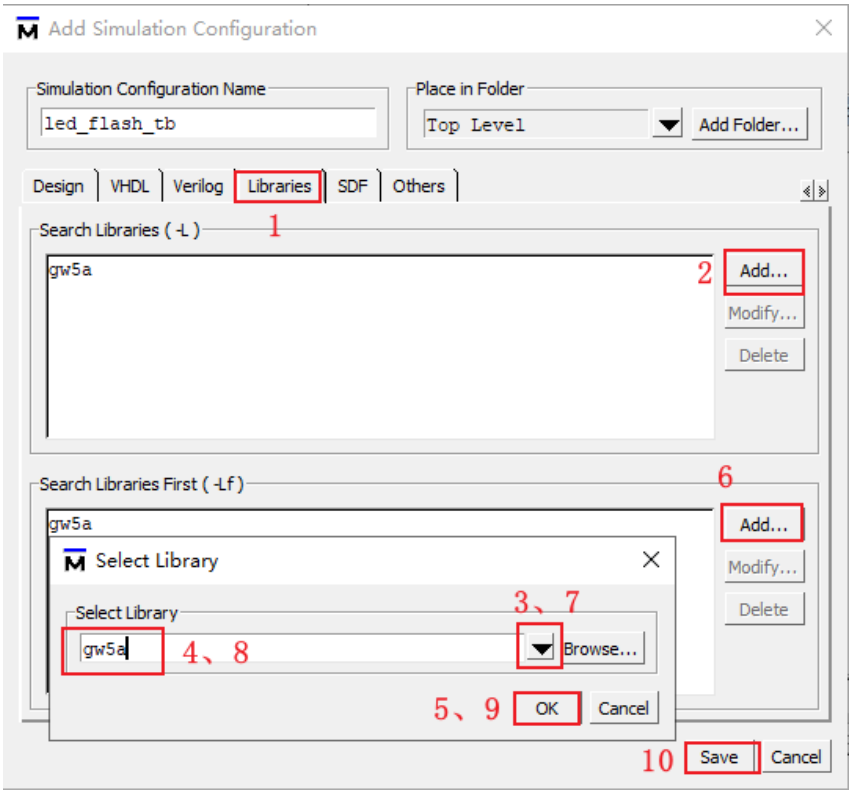


图 3-24 仿真环境配置界面三

通过上述步骤完成了对仿真环境配置，保存配置后，我们可以看到在“project”栏产生了仿真配置文件“led_flash”，如下图 3-25 所示。

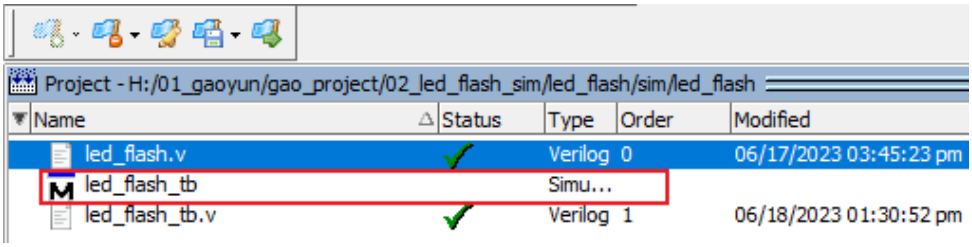


图 3-25 仿真文件生成成功示意图

3.2.5 仿真界面波形观察

1. 我们首先需要双击新生成的仿真文件“led_flash_tb”，进入“sim”界面，如下图 3-26 所示。

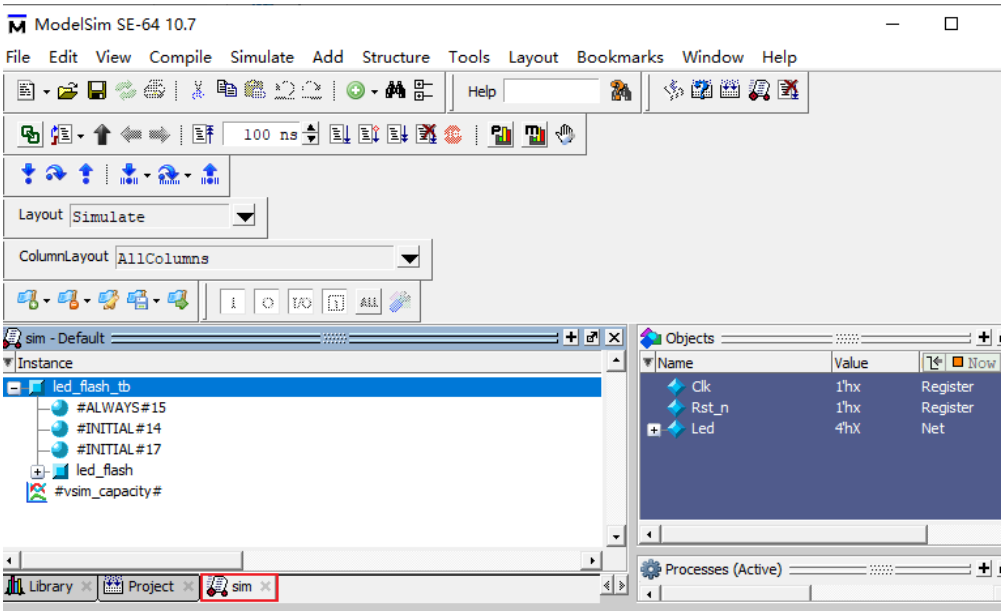


图 3-26 进入 sim 界面

2. 在“sim”界面我们可以添加我们想要观察模块的波形，选中模块，右键点击选择“Add Wave”，如下图 3-27 所示。

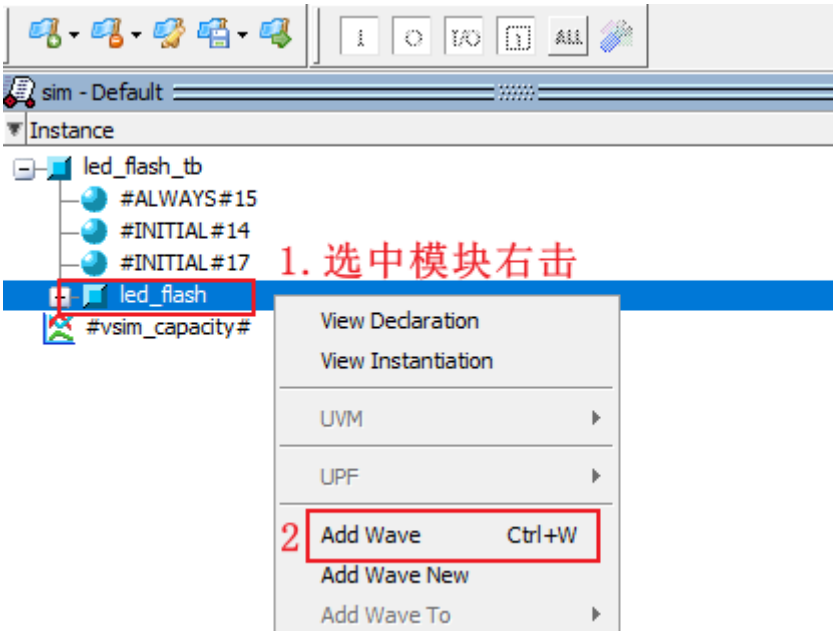


图 3-27 添加需要观测的模块波形

3. 添加好波形后回到“Library”栏，右键单击“work”点击 Update 将“led_flash_tb”文件更新在“work”栏，如下所示。

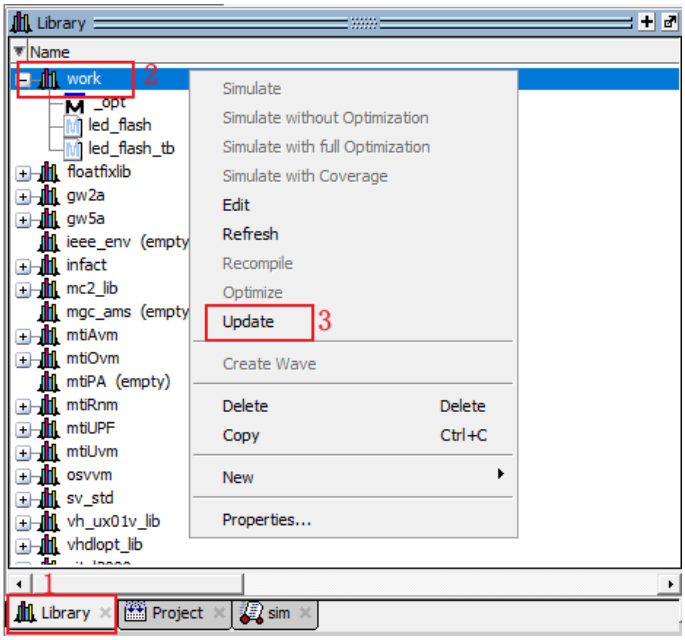


图 3-28 更新 work 栏

4. 点击 Run all，开始跑仿真，如下图 3-29 所示。

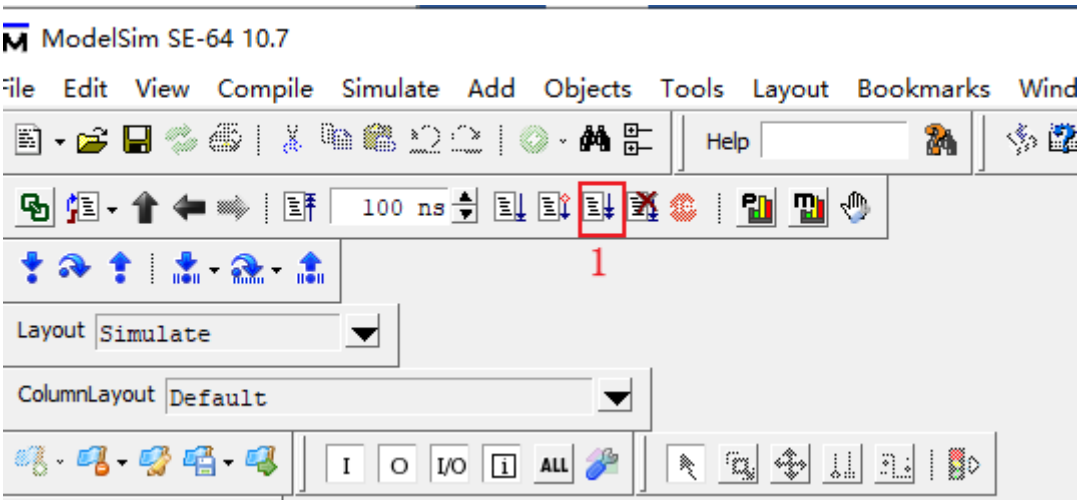


图 3-29 启动仿真

5. 会到 Wave 窗口观察波形，等待程序运行一会儿，我们便可以看到波形变化，如下图 3-30 所示。从下图可以看出，根据 cnt 的计数值，led 的状态发生了翻转。

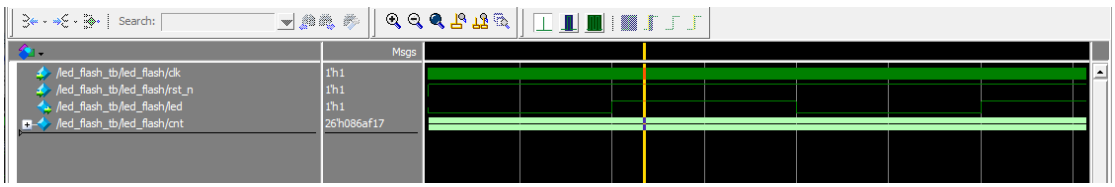


图 3-30 观察波形

如果想停止运行，点击 Wave 窗口上方工具栏的，重新开始是，此时点击“Zoom Full ”可以快速观察到波形，如果想要了解工具栏各个按钮的功能，将鼠标移至按钮处，便可以看到解释，比如继续跑仿真按钮，如下图 3-31 所示：

图 3-31 Wave 界面工具按钮功能查看

在开发过程中，在更改设计文件后，点击保存并检查语法无误后重新运行程序。在 Modelsim 中，我们进入“Library”界面，在“Work”目录下对“led_flash_tb”、“led_flash”进行重新“Recompile”，重新进行仿真波形的加载。

3.2.6 保存波形文件

在使用 modelsim 进行仿真时，对于一些很耗时间的仿真，可以保存仿真波形结果，下次可以直接打开查看。

首先，第一步保存 dataset sim，打开 sim 窗口，点击 file->save dataset sim 或在 sim 界面点击保存.wlf 文件，并给文件命名，操作如下所示。

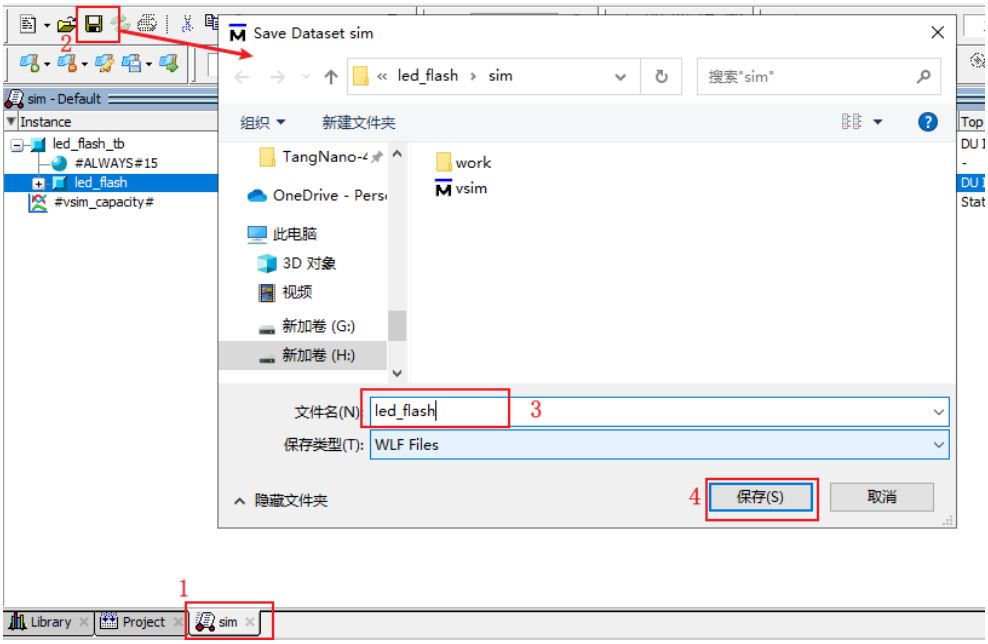


图 3-32 保存.wlf 文件

第二步，Wave 界面保存波形。点击保存图标之后，波形文件“wave.do”会默认保存在工程的仿真文件 sim 中，操作如下图 3-33 所示。

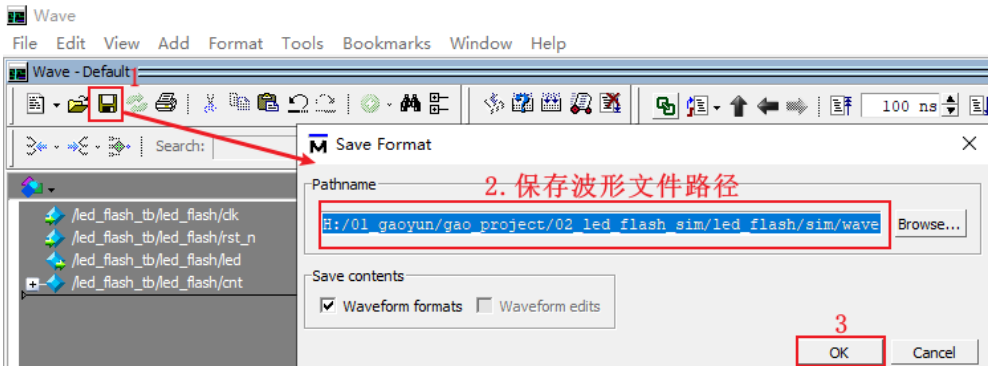


图 3-33 保存波形文件

3.2.7 重新打开仿真工程

在关闭仿真之后，用户需要重新打开仿真，只需要打开“.mpf”文件，如下图 3-34 所示。

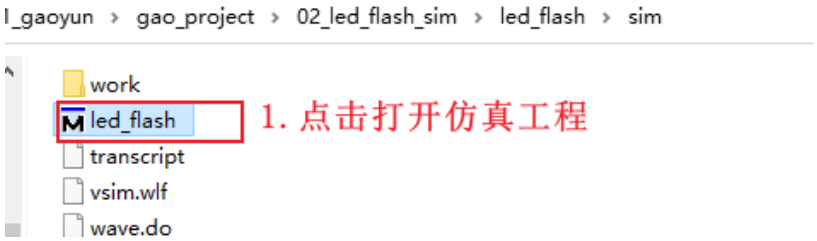


图 3-34 重新打开仿真工程

首次打开“.mpf”时，系统不会默认选择用 Modelsim 打开，读者就需要手动设置“.mpf”文件打开的方式，安装多个版本 Modelsim 的读者还需要注意设置对应版本的 Modelsim 软件，实验安装的版本为 modelsim-win64-10.7-se。

如果之前保存过波形文件，打开 Modelsim 软件之后，点击 file->all file，打开保存的.wlf 文件，操作如下图 3-35 所示。

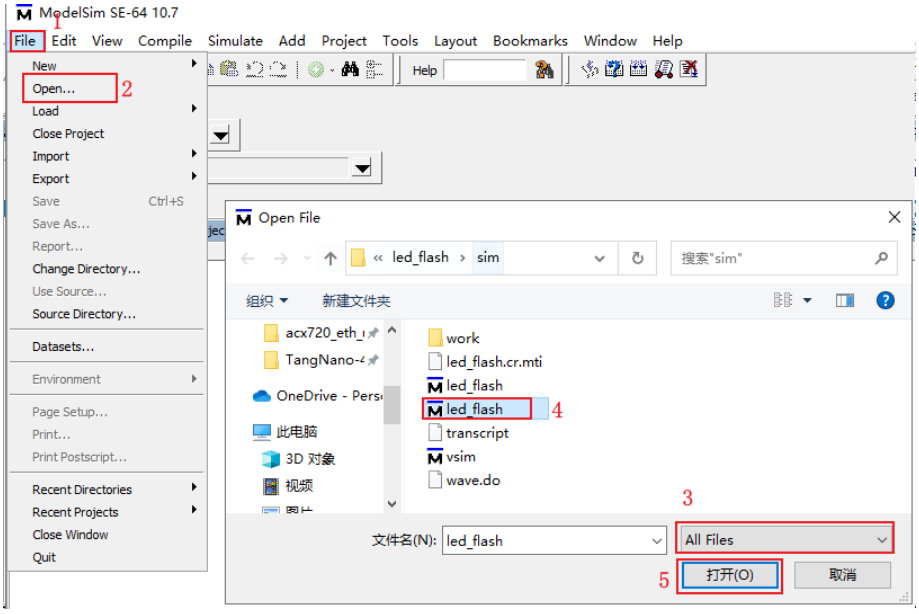


图 3-35 打开保存的.wlf 文件

随后点击 file->load->Macro file，打开保存的.do 文件，如下图 3-36 所示，打开之后，便可以看到我们之前保存的波形文件。

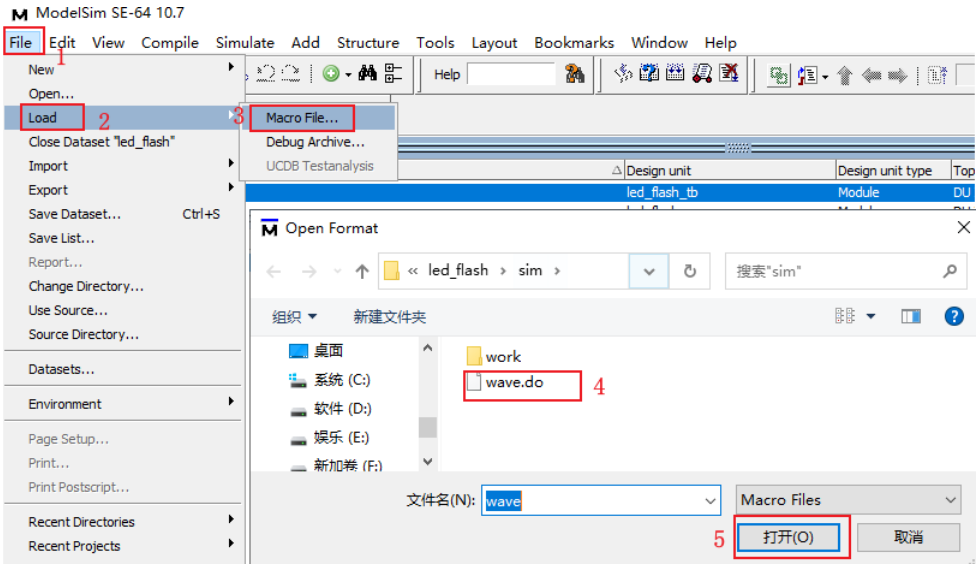


图 3-36 打开保存的波形文件

3.3 思考与总结

本章实验带领大家学习了如何使用 Modelsim 编译高云 FPGA 的器件库，如何使用 Modelsim 进行仿真，本章仿真实验源码 LED 灯状态变化的时间为 1S，仿真波形出现变化需要等待一会儿，如果想更快速看到波形变化，可以将计数器 cnt 的计数减小，以便更快看到波形的变化。

4 Gowin 在线逻辑分析仪的使用

工程源码	|----02_设计实例 |----ch4_led_flash_gao
相关视频课程	
说明	

章节导读

本章实验在前面实验的基础之上，带领大家学习 Gowin 在线逻辑分析仪的使用，这里我们只讲解该功能的基本使用流程，更多的功能将会在后面实验需要使用到的时候进行说明。

4.1 Gowin 在线逻辑分析仪简介

高云半导体在线逻辑分析仪（Gowin Analyzer Oscilloscope，以下简称 GAO）的作用是帮助用户更加简便地分析设计中信号之间的时序关系，快速进行系统分析和故障定位，提高设计效率。

GAO 的工作原理：FPGA 工作时利用器件中未使用的存储器资源，根据用户设定的触发条件将信号实时地保存到存储器中，通过 JTAG 接口实时读取信号的状态并将其显示在云源界面上。GAO 包括信号配置窗口和波形显示窗口。信号配置窗口主要用于把定位信息插入到设计中，该类定位信息主要基于采样时钟、触发单元和触发表达式：波形显示窗口通过 JTAG 接口连接云源和目标硬件，将信号配置窗口设置的采样信号的数据直观地通过波形显示出来。

GAO 支持 RTL 级信号捕获和综合后网表级信号捕获，并且提供 Standard GAO 和 Lite GAO 两种版本。Standard GAO 最多可以支持 16 个功能内核，每个内核可配置一个或多个触发端口，支持多级静态或动态触发表达式。Lite GAO 配置简便，无需设置触发条件，可以捕获信号的初始值，方便用户分析上电瞬间的工作状态。

4.2 GAO 文件配置

4.2.1 新建 GAO 配置文件

打开 Gowin 工程，点击  ，在弹出的 New 对话框中，选择新建一个 GAO Config File，如下图 4-1 所示，点击 OK。

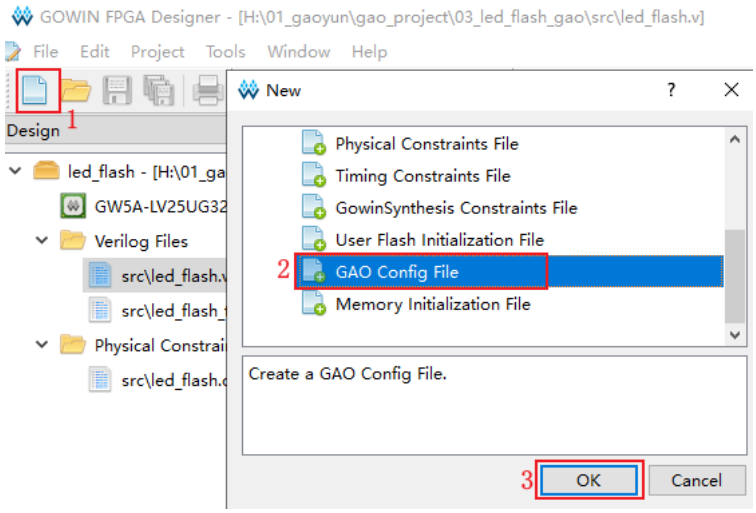


图 4-1 新建 Gao 配置文件

弹出 New GAO Wizard 对话框，Type 一栏中有两种选项：“For RTL Design”和“For Post-Synthesis Netlist”。其中“For RTL Design”类型用于捕获综合优化前 RTL 信号，生成配置文件扩展名.rao；“For Post-Synthesis Netlist”类型用于捕获综合优化后 Netlist 信号，生成配置文件扩展.gao。这里我们选择“For RTL Design”类型的 Standard GAO 配置文件，如下图 4-2 所示。

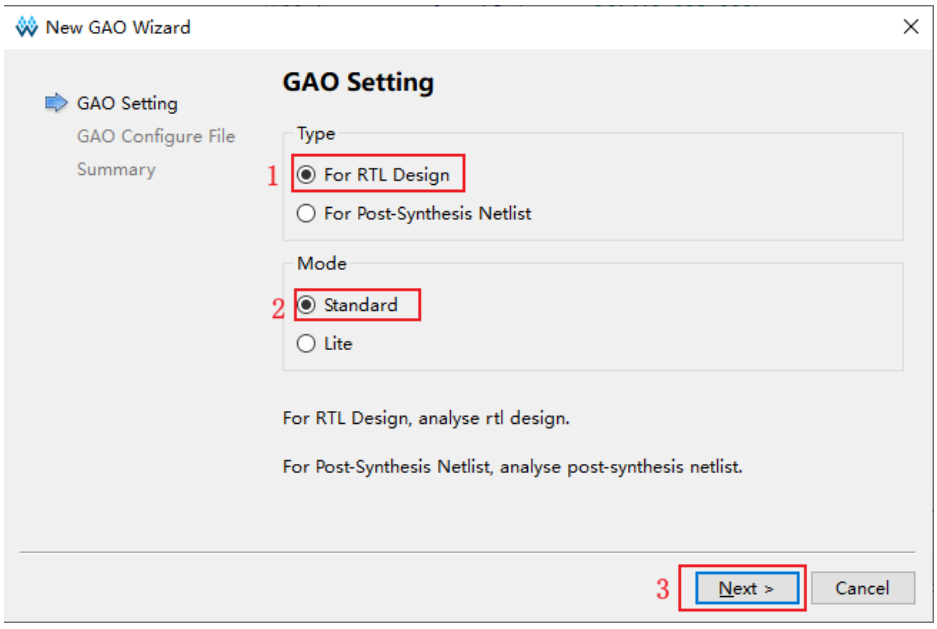


图 4-2 GAO 文件配置界面 1

然后在弹出的窗口中的“Name”编辑框中输入配置文件名称，默认和工程名保持一致，这里按照默认名称，单击“Next”按钮，如下所示。

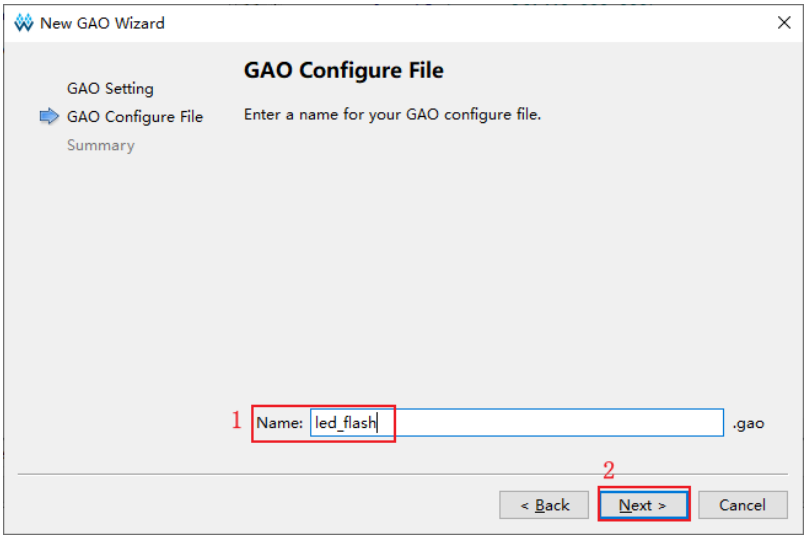


图 4-3 GAO 文件配置界面 2

最后弹出 Summary 对话框，点击 Finish 完成 GAO 文件的建立。

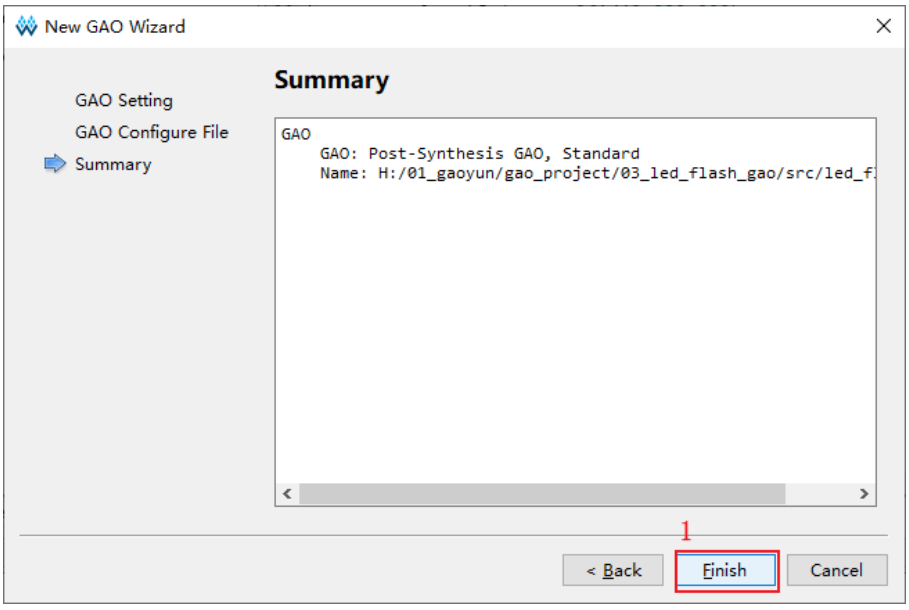


图 4-4 GAO 文件配置界面 3

4.2.2 启动 GAO 配置窗口

GAO 文件建立完成之后，将会在 Design 窗口出现 GAO Config Files，该栏下面显示的就是我们刚刚新建的 GAO 文件，双击进入 GAO 配置窗口，如下图 4-5 所示。GAO 配置窗口包括配置功能内核数量的 Ao Core 和对应 Core 的信号配置视图，其中 Core 信号配置视图包括配置信号触发条件的 Trigger Options 视图和配置信号采样条件的 Captrue Options 视图。

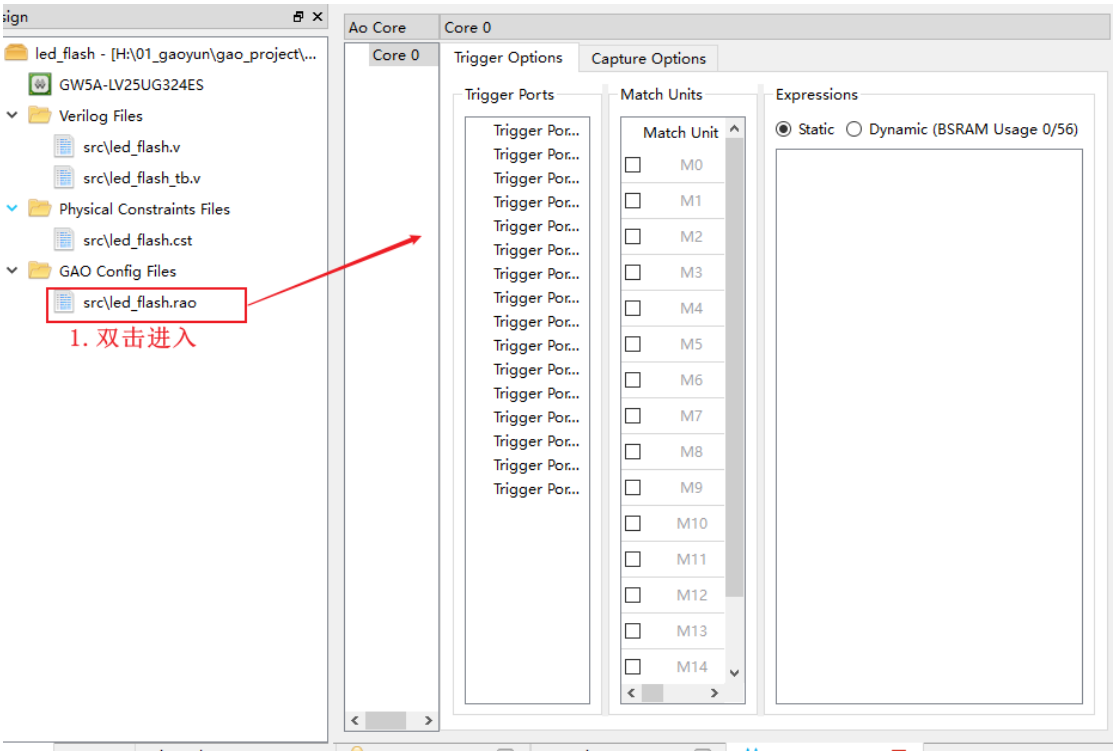


图 4-5 进入 GAO 配置窗口

注意，如果在双击.gao 文件的时候，出现如下图 4-6 警告，需要我们先分析综合（Synthesize）之后，然后再双击.gao 文件进入 GAO 配置窗口。

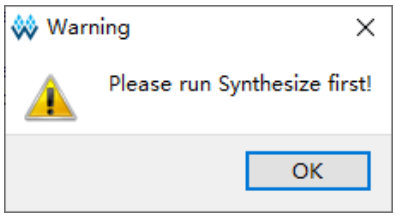


图 4-6 出现提示警告

4.2.3 配置 Standard Mode GAO

Standard Mode GAO 配置窗口用于功能内核数量、信号触发条件、信号采样条件的配置。

1. 配置功能内核数量

Ao Core 视图用于显示及管理当前工程所使用的功能内核数量，默认 Core 0，最多可支持 16 个 Core，可以再 Ao Core 视图的任意位置点击 Add 添加，或者 Remove 移除，如下图 4-7 所示，这里我们使用默认的 Core 0。

图 4-7 添加或者移除 Core 的操作示意图

2. 配置触发端口

Core 0 的 Trigger Ports 视图下用于配置功能内核触发端口，可以看到一共有 16 个触发端口，每个触发端口的宽度范围为 1~64。这里使用端口 0，双击之后，进入端口配置界面 Trigger Port，如下图 4-8 所示，图中的 MSB 和 LSB 分别表示触发端口的高位与低位。

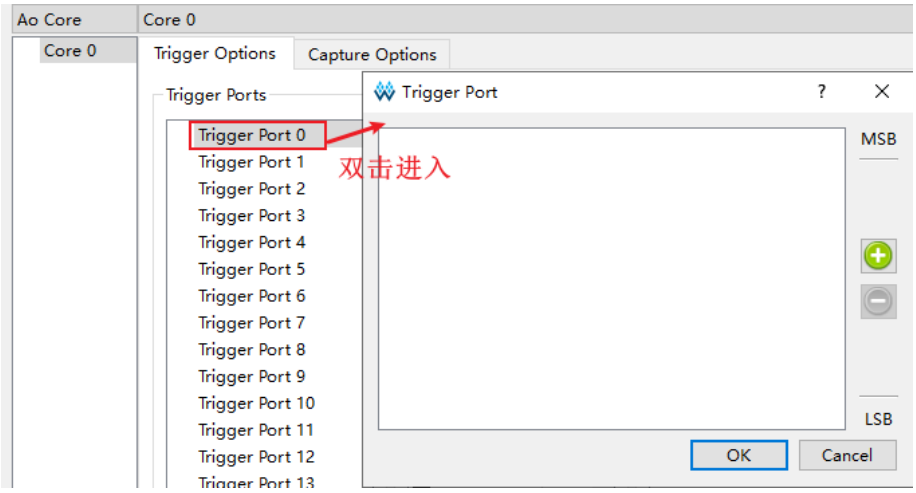


图 4-8 进入触发端口配置界面操作示意图

点击进入 Search Nets 对话框，在 Name 一栏中输入你想要添加的触发方式，比如本次实验想要观察的触发方式为 LED 灯的状态信号 led，输入 led，然后点击 Search 进行搜索，如下图 4-9 所示。

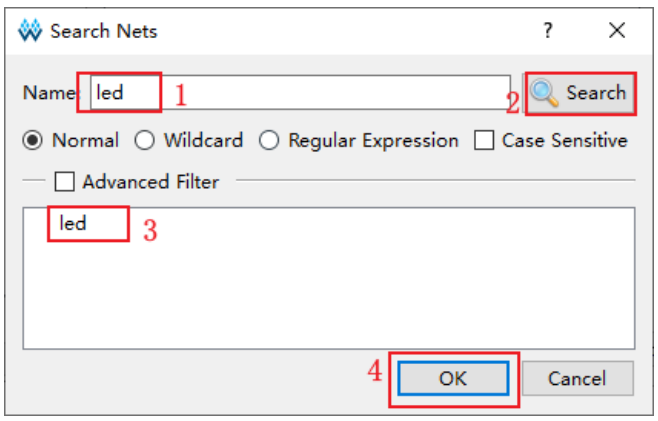


图 4-9 选择触发信号

然后回到 Trigger Port 点击 OK，如下图 4-10 所示。

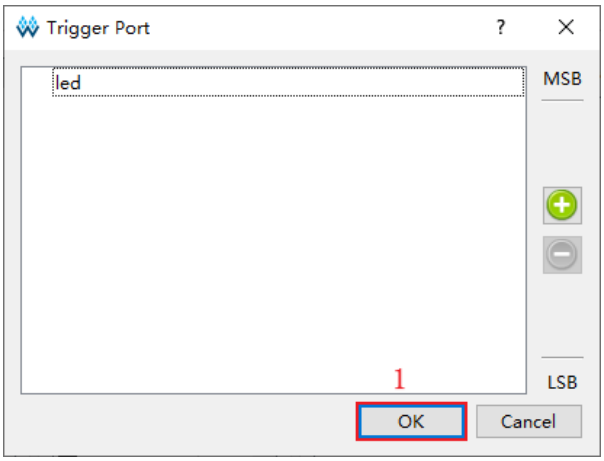


图 4-10 触发信号添加

3. 配置匹配单元

Match Units 视图用于配置触发端口的匹配单元，最多 16 个触发匹配单元，16 个匹配单元对应 M0~M15。匹配单元是 GAO 功能内核实现触发条件的最小单元，功能内核通过匹配单元对用户设计的触发端口信号进行处理，当触发端口信号满足要求时，可实现触发，这里我们选择匹配单元 M0，双击匹配单元，在弹出的“Match Unit Config”对话框中对触发条件进行配置，如下图 4-11 所示。

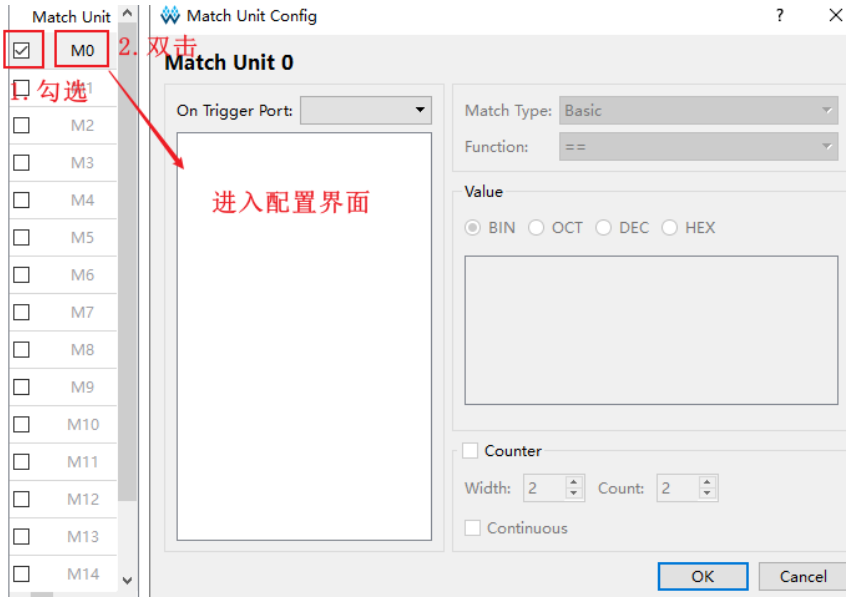


图 4-11 进入触发条件配置界面

在弹出的“Match Unit Config”对话框中，单击“On Trigger Port”下拉框，在下拉列表中选择触发端口 Trigger Port 0。对话框中还包含 “Match Type（类

型）”、“Function（函数）”、“Value（用于设置 Bit Value 值）”，对于这几项的说明如下表 4-1 所示。

表 4-1 触发匹配单元支持的匹配类型

类型	Bit Values	匹配函数	说明
Basic	0、1、X	==、!=	用于一般的信号比较，是一种比较节约资源的类型
Basic w/edges	0、1、X、R、F、B、N	==、!=、跳变检测	用在控制信号的跳变需要考虑的情况
Extended	0、1、X	==、!=、>、>=、<、<=	用在地址或数据信号的值需要考虑的情况
Extended w/edges	0、1、X、R、F、B、N	==、!= >、>=、<、<=、跳变检测	用在地址或数据信号的值和跳变都需要考虑的情况
Range	0、1、X	==、!= 、>=、<、<=、范围内检测、范围外检测	用在特定范围内地址或数据信号的值需要考虑的情况
Range w/edges	0、1、X、R、F、B、N	==、!= 、>=、<、<=、范围内检测、范围外检测、跳变检测	用在对特定范围内地址或数据的信号的值和跳变需要考虑的情况

注：Bit Values 中“0”表示低电平 0，“1”表示高电平，“X”表示均可，“R”表示上升沿 0->1 变化，“F”表示下降沿 1->0 变化，“B”表示上升沿或下降沿转换均可，“N”表示没有逻辑电平转换

每个触发匹配单元均有一个计数器，用于设置触发条件满足 N 次后开始采样数据，N 是计数器数值，这个和“Counter”复选框是否勾选有关，勾选“Counter”复选框，可设置使用计数器，若不使用计数器，则默认匹配 1 次后开始采集数据。

“Match Unit Config”对话框中各项配置用户可以根据自己的需求进行配置，这里我们按照默认配置即可，最终配置如下图 4-12 所示。

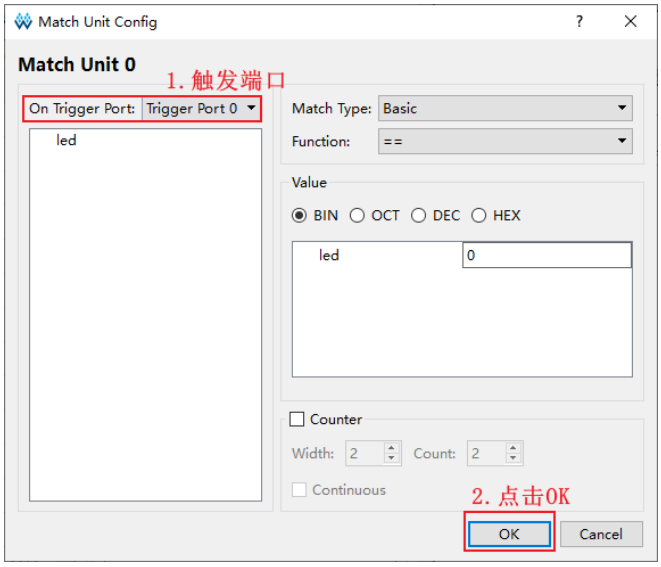


图 4-12 配置触发条件

4. 配置触发表达式

Expressions 视图用于设置触发表达式，一个功能内核最多 16 个触发表达式。右击 Expressions 视图任意处，选择“Add”，弹出 Expression 对话框，如下图 4-13 所示。

图 4-13 进入 Expressions 对话框

这里我们只有一个触发匹配单元 M0，点击 M0，然后点击 OK 即可，如下图 4-14 所示。

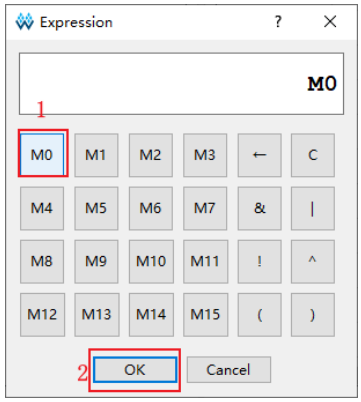


图 4-14 选择触发表达式

添加完成之后，如下图 4-15 所示，用户可以双击进行修改。

图 4-15 触发表达式添加成功

5. 配置采样信号

点击“Capture Options”进入采样信号配置视图，Capture Options 视图主要用于配置采样时钟、存储深度、采样数据信号灯信号采样信息，并显示当前 Ao Core 的 Capture Signals 使用的 BSRAM 资源数目，Capture Options 视图如下图 4-16 所示。

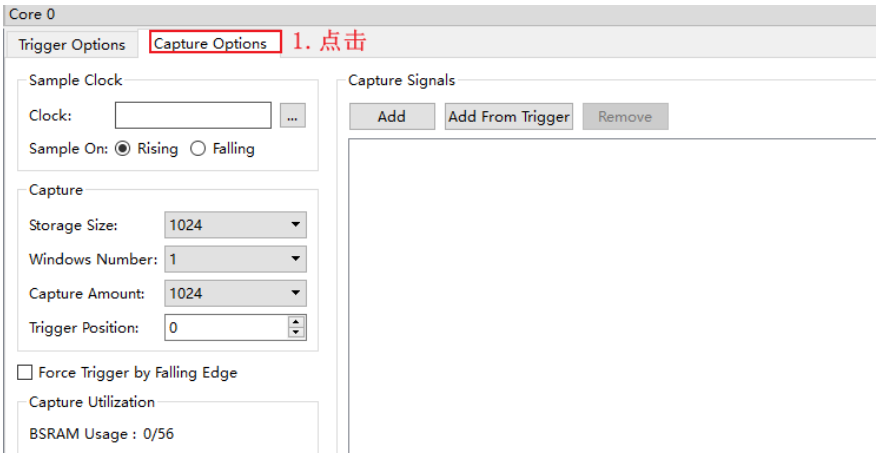


图 4-16 Capture Options 视图

首先，配置采样时钟 Clock，点击 Clock 后的 ... 进入 Search Nets 界面，在文件框中直接输入采样时钟 clk，然后点击 Search 搜索，搜索成功之后进行添加，如下图 4-17 所示。

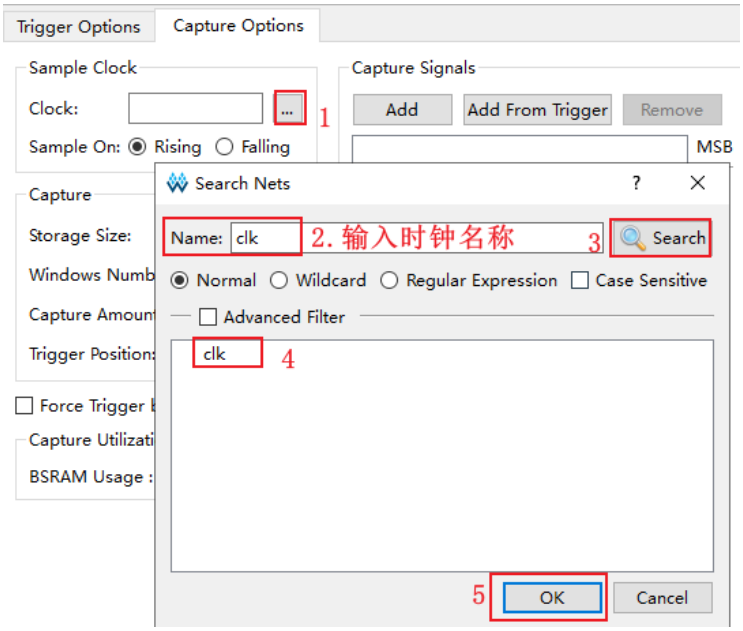


图 4-17 添加采样时钟

然后存储深度为 4096，采集窗口数目 1 个，采样长度也为 4096，触发点位置为 0，如下图 4-18 所示。

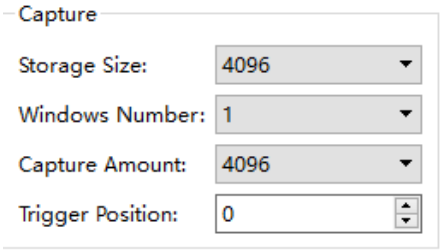


图 4-18 设置 Captrue 选项

选择需要采样的数据信号，点击“Add”按钮，选择需要功能内核采样存储数据的信号作为采样数据信号，然后弹出 Search Nets 对话框，选择所需的数据端口信号，点击“OK”即可完成配置，这里也可以添加 bus 信号，这里我们添加 cnt[25:0]信号进行观察，如下图 4-19 所示。

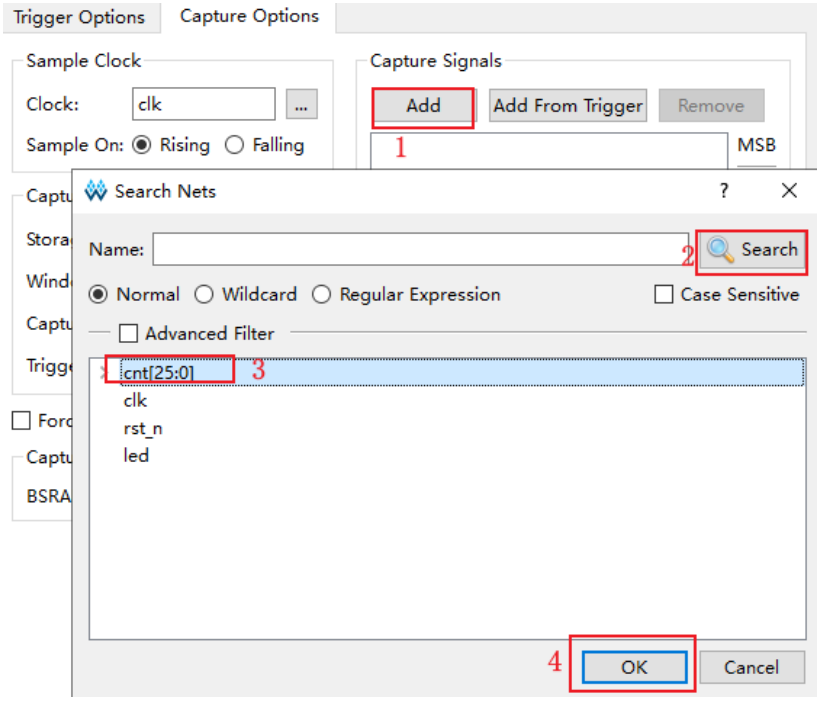


图 4-19 添加需要观察的信号

上图中还有一个按钮 “Add From Trigger”，该按钮直接使用触发端口采样触发信号作为采样数据信号，可在 Add From Trigger 下方列表中选择一个或多个触发端口，使用已经选择的触发端口采集信号作为采样数据信号，如下所示。

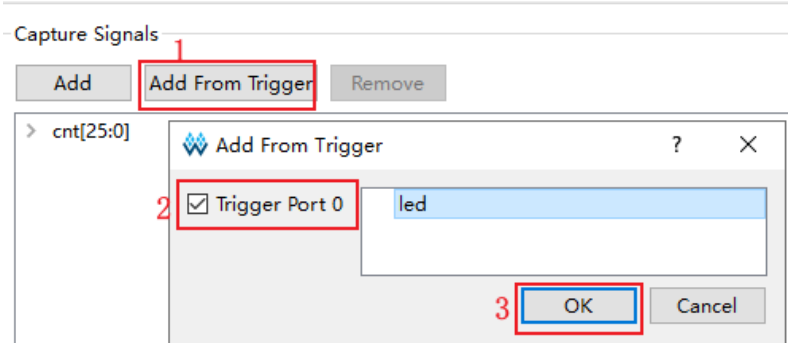


图 4-20 选择触发信号作为采样信号

添加完成之后，如下图 4-21 所示，选中信号之后，可以选择“Remove”清除信号。

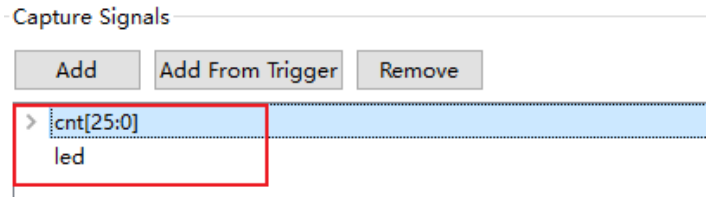


图 4-21 采样信号配置完成

至此，GAO 文件就配置完成了，Ctrl+S 保存，保存过程中没有报错，则代表配置没有出现错误，如果出现错误，请用户按照操作步骤检查自己的步骤是否完整，是不是哪个环节漏掉了，或者配置出错。

4.3 布局布线

每次修改完 GAO 文件之后，都需要点击“Place & Route”重新布局布线，如下图 4-22 所示。

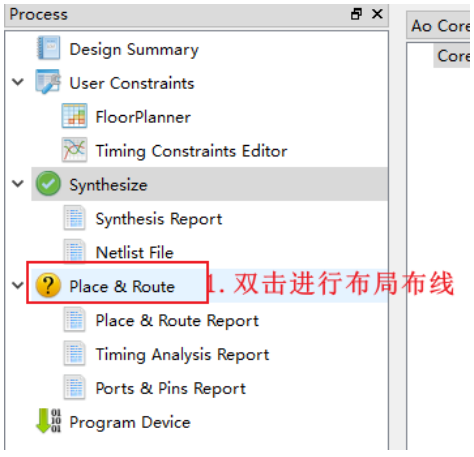


图 4-22 重新布局布线

4.4 下载数据流

点击 Program Device 下载数据流，操作如下所示。

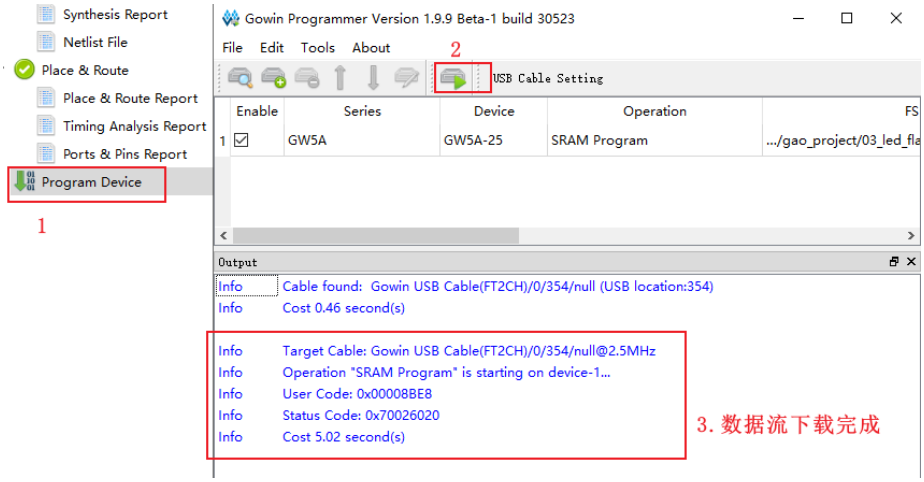


图 4-23 下载数据流完成

4.5 波形抓取

点击工具栏中的 按钮，进入高云在线逻辑分析界面，如下图 4-24 所示。一定要注意上方的 Cable 一定要是 Gowin USB Cable(FT2CH)。

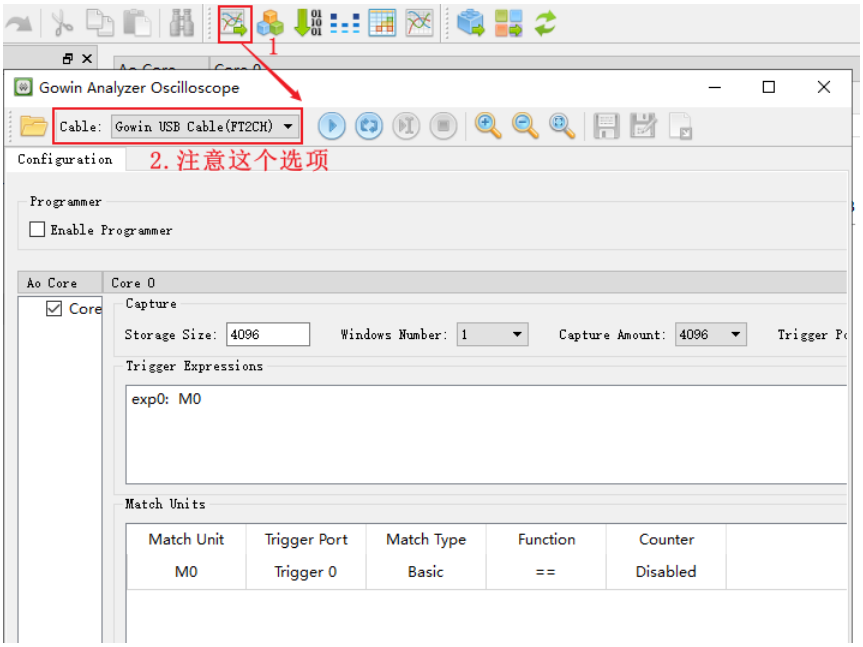


图 4-24 进入高云在线逻辑分析仪界面

从上述界面中，我们可以看出，还可以继续修改 Core 0 的采样配置，这里

我们不做修改，直接点击上方的观察波形，波形如下图 4-25 所示。

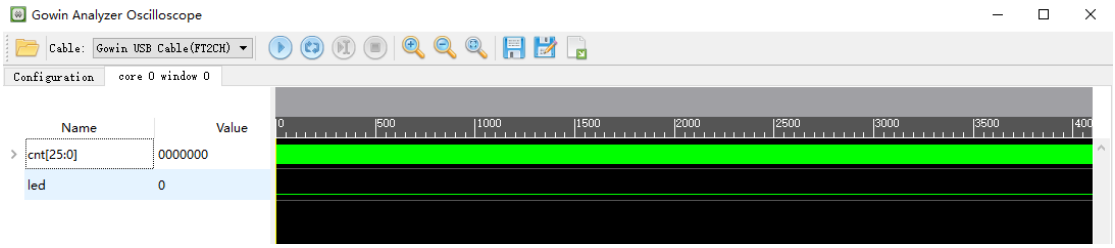


图 4-25 高云在线逻辑分析仪获取的波形界面

此时点击上方的按钮，观察 cnt 的变化，如下图 4-26 所示，可以看出 cnt 一直在累加，led 没有发生变化，是因为 cnt 的计数值太大，观察不到变化。

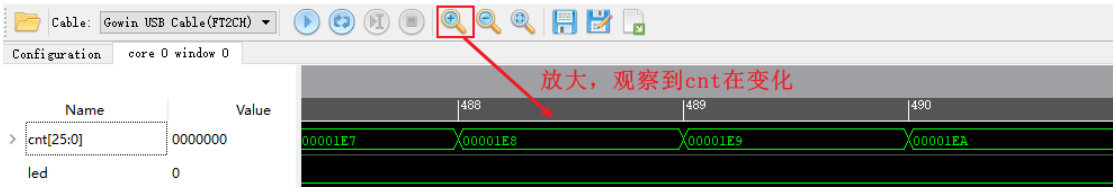


图 4-26 观察 cnt 变化

如果用户想看到 led 的变化，可以将 cnt 的值减小，比如设定为 299，如下图 4-27 所示。

```
reg [25:0] cnt;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt <= 26'd0;
    else if (cnt < 26'd299)
        cnt <= cnt + 1'b1;
    else
        cnt <= 26'd0;
end

always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        led <= 1'b0;
    else if (cnt == 26'd299)
        led <= ~led;
    else
    led <= led;
end
```

图 4-27 修改 cnt 参数

修改完成之后，还需修改 GAO 文件中的采样信号 cnt，如下图 4-28 所示，这里修改是因为实际使用到的计数只有 cnt 的第 0~8 位。

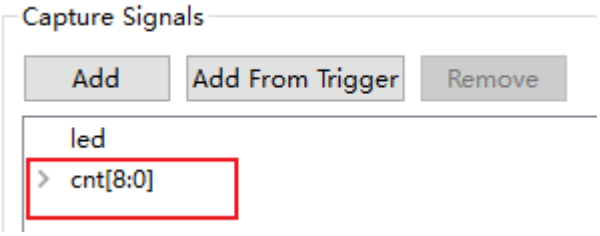


图 4-28 修改采样信号 cnt

重新编译下载数据流文件，此时我们通过肉眼看不到 LED 灯的变化，通过高云逻辑分析仪抓取波形如下图 4-29 所示。从图中可以看出 LED 灯的状态一直在发生变化。

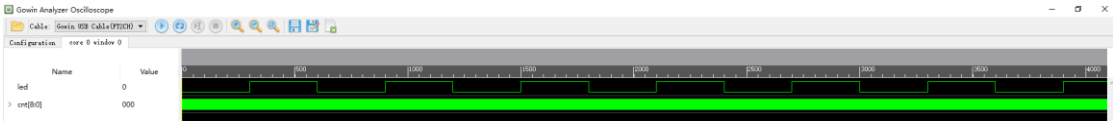


图 4-29 修改 cnt 之后 led 的变化

我们放大其中一段边缘变化如下所示，从图中可以看出，cnt 计数到 12B 之后 LED 发生变化，换成 10 进制，对应的就是 299，与我们代码设计一致。

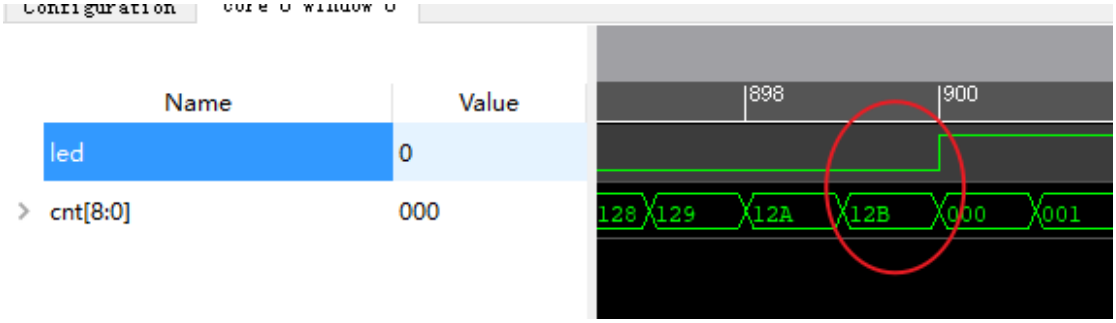


图 4-30 波形边缘变化波形图

至此，高云在线逻辑分析仪的基本使用流程就介绍完成了，还有很多功能在本次实验中并未使用到，将在后续使用到之后再做补充。

4.6 思考与总结

我们在进行 FPGA 的开发中，在线逻辑分析仪的使用非常广泛，本章实验带领大家熟悉了在 Gowin 软件中如何使用在线逻辑分析仪，希望用户根据我们的步骤多熟悉整个操作，以便于后续在自己设计的工程中使用。

5 时序逻辑电路设计之计数器

工程源码	|----02_设计实例 |----ch5_bin_counter
相关视频课程	
说明	

章节导读

时序逻辑电路是指电路任何时刻的稳态输出不仅取决于当前的输入，还与前一时刻输入形成的状态有关。这跟组合逻辑电路相反，组合逻辑的输出只会跟目前的输入成一种函数关系。换句话说，时序逻辑拥有储存元件来存储信息，而组合逻辑则没有。

时序逻辑电路分为很多种，本节将以最常用的计数器为例学习简单的时序逻辑电路设计，并比较与组合逻辑电路的区别。此处设计一个计数器，使学习板上的 LED 状态每 500ms 翻转一次。学习板上晶振为 50MHz，也就是说时钟周期为 20ns，这样可以计算得出 500ms = 500_000_000ns/20ns = 25_000_000，即需要计数器计数 25_000_000 次，也就是需要一个至少 25 位的计数器（225>25_000_000>224）。且每当计数次数达到需要清零并重新计数。

5.1 计数器工作原理

计数器的核心元件是触发器，基本功能是对脉冲进行计数，其所能记忆脉冲最大的数目称为该计数器的模/值。计数器常用在分频、定时等处。计数器的种类很多，按照计数方式的不同可以分为二进制计数器、十进制计数器以及任意进制计数器，按照触发器的时钟脉冲信号来源可分为同步计数器与异步计数器。按照计数增减可分为加法计数器、减法计数器以及可逆计数器。

Verilog HDL 之所以被称为硬件电路描述语言，就是因为我们不是在类似 C 一样进行普通的编程，而是在编写一个实际的硬件电路。下面将设计一个计数器，通过计数器控制一个 led 闪。上面提到计数器即为加法器、比较器、寄存器以及选择器构成，如下图 5-1 所示。

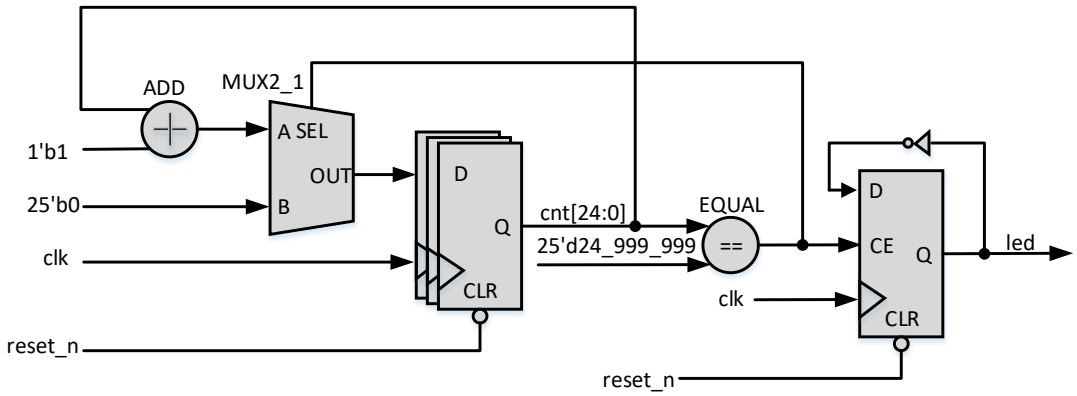


图 5-1 计数器控制 led 逻辑电路图

开发板上 LED 硬件电路如下图 5-2 所示，可以看出当控制端输出高电平时，LED 亮，输出低电平时，LED 灭。这里只需要每当计数器值记满后，翻转 LED 的控制端即可实现 LED 按照要求亮灭。

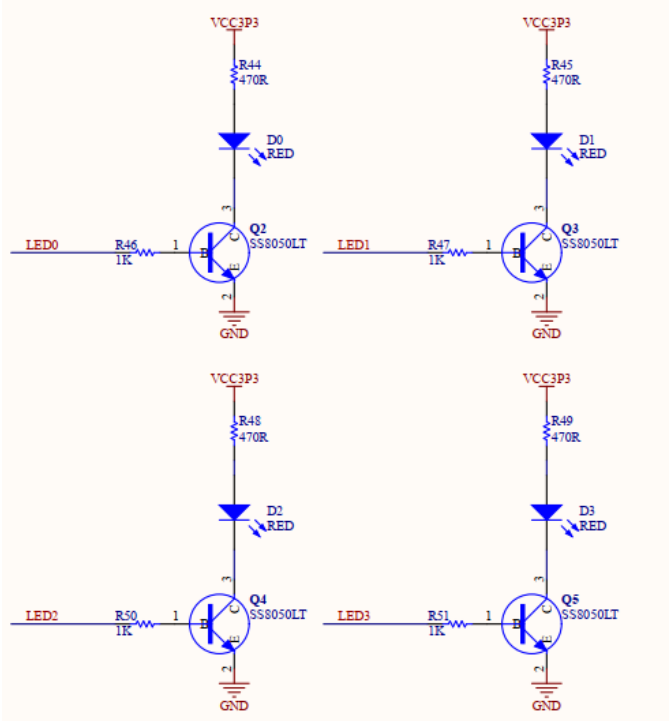


图 5-2 开发板上 LED 电路原理图

5.2 计数器 Verilog 实现

由上面分析可以得出本设计的模块接口示意图如下图所示，其各端口的功能描述如下表所示。

图 5-3 计数器模块示意图

表 5-1 计数器模块端口功能描述

端口名称	I/O	端口功能描述
clk	I	模块的工作时钟，频率为 50MHz，
reset_n	I	模块复位，低电平复位
led	O	LED 控制端，输出低电平时 LED 灯亮

5.2.1 工程建立

依据前面相关章节所述，新建一个名为 bin_counter 的工程。双击桌面 Gowin 软件的图标打开 Gowin 开发环境，按如下操作步骤进行：

（1）起始页——点击 New Project。

（2）New 页面——点击 OK。

（3）Project Name 界面——工程名 bin_counter，设置工程路径，点击 Next。

（4）Select Device 界面——选择芯片型号，根据自己板卡上的 FPGA 芯片进行选择（芯片上面对应的有型号），然后点击 Next。

（5）Summary 界面——点击 Finish，完成工程的创建。

建立好的工程如下图 5-4 所示。

General	
Project File:	H:\01_gaoyun\gao_project\05_bin_counter\bin_counter\bin_counter.gprj
Synthesis Tool:	GowinSynthesis

Target Device	
Part Number:	GW5A-LV25UG324ES
Series:	GW5A
Device:	GW5A-25
Package:	UBGA324
Speed Grade:	ES
Core Voltage:	LV

图 5-4 工程建立完成示意图

5.2.2 代码设计

从计数器模块框图以及计数器模块端口功能表的分析设计可得出端口列表。

```
module bin_counter
(
    clk,
    reset_n,
    led
);
    input clk;
    input reset_n;
    output led;
    reg led;

endmodule
```

从实验原理中可以看出需要一个计数器，因为计数器是从 0 开始计数而不是 1，因此在计数值计数到 25'd24_999_999 时清零而不是计数到 25'd25_000_000 时清零，这里计数器最大值使用 parameter 进行参数化定义表示，使用参数化定义的好处是通过修改 parameter 中定义最大值的数值可以全部修改设计中用到的这个参数。具体代码如下。

```
parameter MCNT = 24_999_999;

reg [24:0]cnt;  //定义计数器寄存器

//计数器计数进程
always@(posedge clk or posedge reset)
if(reset)
    cnt <= 25'd0;
else if(cnt == MCNT)
    cnt <= 25'd0;
else
    cnt <= cnt + 1'b1;
```

当计数器计数到预设的值后就让 led 取反一次，来达到亮灭翻转的目的。

```
//led 输出控制进程
always@(posedge clk or posedge reset)
if(reset)
    led <= 1'b1;
else if(cnt == MCNT)
    led <= ~led;
else
    led <= led;
```

5.3 功能仿真验证

将上面的设计内容进行分析和综合直至没有错误以及警告。新建计数器仿真文件。具体步骤如下。

（1）点击工具栏中的 新建仿真文件。

（2）New 弹窗中选择 Verilog File，点击 OK。

（3）New Verilog file 弹窗中 Name 一栏中给文件命名为 bin_counter_tb，其它默认，点击 OK，仿真文件建立成功。

如下图 5-5 所示，在 Design->Verilog Files 中出现新建的仿真文件，双击文件名进入编辑窗口。

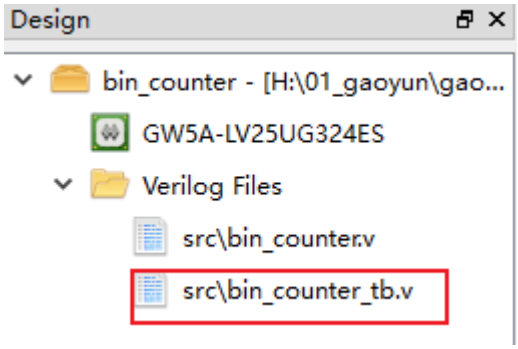


图 5-5 仿真文件新建成功示意图

仿真激励产生一个周期为 20ns 的时钟 clk 和复位信号，并且例化了需要测试的 bin_counter.v。

```
`timescale 1ns/1ns
`define CLOCK_PERIOD 20

module bin_counter_tb();
    reg  clk;
    reg  reset_n;
    wire led;
    bin_counter bin_counter_inst (
        .clk(clk),
        .reset_n(reset_n ),
        .led(led)
    );

    initial clk = 1;
    always #(`CLOCK_PERIOD/2) clk = ~clk;

    initial begin
```

```
        reset_n = 1'b0;
        #(`CLOCK_PERIOD *200 + 1);
        reset_n = 1'b1;
        #2000000000;
        $stop;
    end

endmodule
```

仿真文件添加完成之后，进行分析综合，查看是否有语法错误，检查无误之后，使用 Modelsim 对工程进行仿真验证。

5.3.1 建立 Modelsim 仿真工程

建立 Modelsim 仿真工程，操作如下所示：

（1）工程目录下，新建一个 sim 文件夹用于存放仿真工程文件。

（2）点击 Modelsim 图标，进入 Modelsim 软件。

（3）依次点击 File->New->Project，新建一个仿真工程。

（4）Create Project 弹窗中 Project Name 一栏中给工程命名为 bin_counter，点击 Browse 选择工程目录下的 sim 文件夹用于存放工程，点击 OK。

（5）弹出 Add items to the Project 的对话框，点击 Add Existing File->Browse 添加本次实验所需的仿真文件：bin_counter.v 和 bin_counter _tb.v，点击 OK。

（6）点击 Compile->Compile All，编译所有文件。

编译成功之后，如下图 5-6 所示，如果出现错误，请根据提示进行修改。

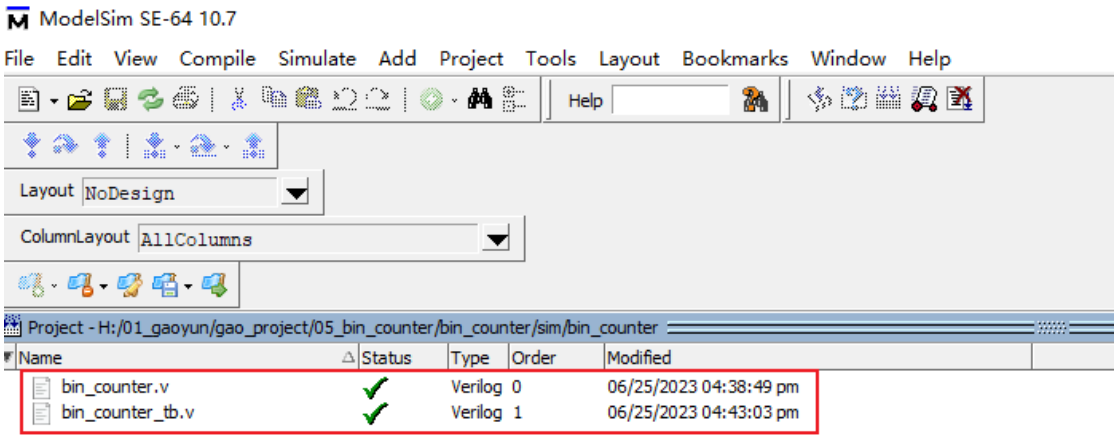


图 5-6 文件编译成功示意图

5.3.2 配置仿真环境

仿真工程建立并编译完成之后，接下来就需要进行仿真环境的配置，操作如下所示：

（1）我们在 project 状态栏中右键点击，然后选择“Add to Project”->“Simulation Configuration…”。

（2）进入 Add Simulation Configuration 页面，我们在 Design 标签页面中选择 work 库中的“bin_counter_tb”模块作为设计顶层，点击复制模块名作为仿真配置“Simulation Configuration Name”的命名，确保命名保持一致。

（3）点击“Optimization Options…”，在“Optimization Options..”设置栏中选择“Apply full visibility to all modules(full debug module)”，点击“OK”。

（4）点击“libraries”设置栏，在“Search libraries(-L)”一栏中点击“Add…”添加我们新建的高云的库文件“gw5a”，在“Search Libraries First(-Lf)”同样选择库文件“gw5a”，最后点击“Save”保存设置。

通过上述步骤完成了对仿真环境配置，保存配置后，我们可以看到在“project”栏产生了仿真配置文件“bin_counter_tb”，如下图 5-7 所示。

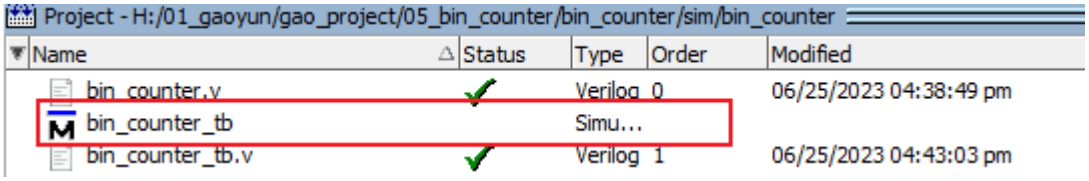


图 5-7 仿真文件生成成功示意图

5.3.3 仿真界面波形观察

我们首先需要双击新生成的仿真文件“decoder3_8_tb”，进入“sim”界面，在“sim”界面我们可以添加我们想要观察模块的波形，选中模块 decoder3_8_tb，右键点击选择“Add Wave”，进入 Wave 窗口，点击工具栏中的全速运行，仿真停止后点击，波形如下图 5-8。

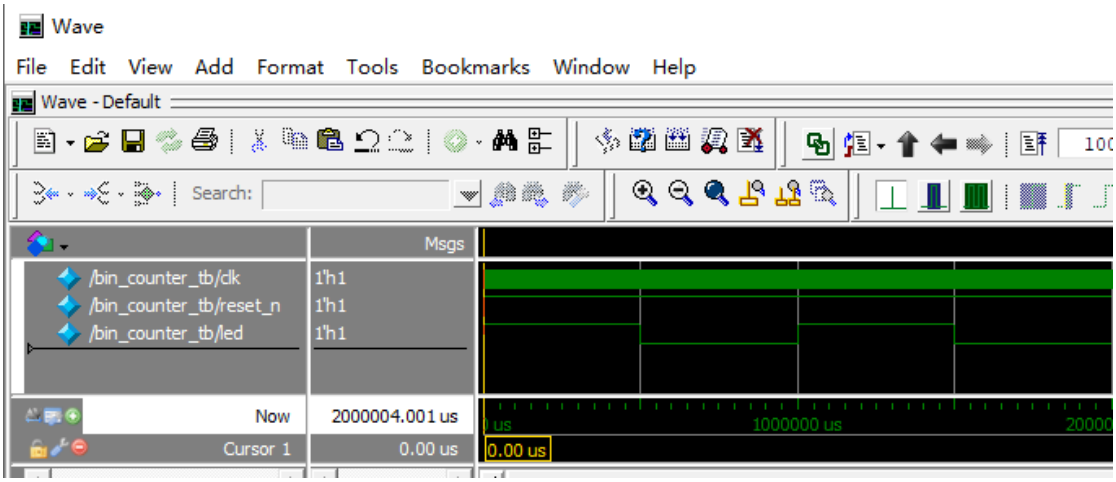


图 5-8 波形图

从图中得出结论如下：可以看出高低电平变化的时间均是 500000000ns 也就是 0.5s，符合既定的设计要求，至此功能仿真结束

上述仿真时间单位为 us，要判断 led 输出高低电平时间还需要自己通过时间单位换算，不太直观，可以对仿真显示时间单位进行设置修改，具体步骤如下。

（1）如下图 5-9 所示，在黄色框出区间内，鼠标右键点击，选择并点击 Grid,Timeline & Cursor Control…

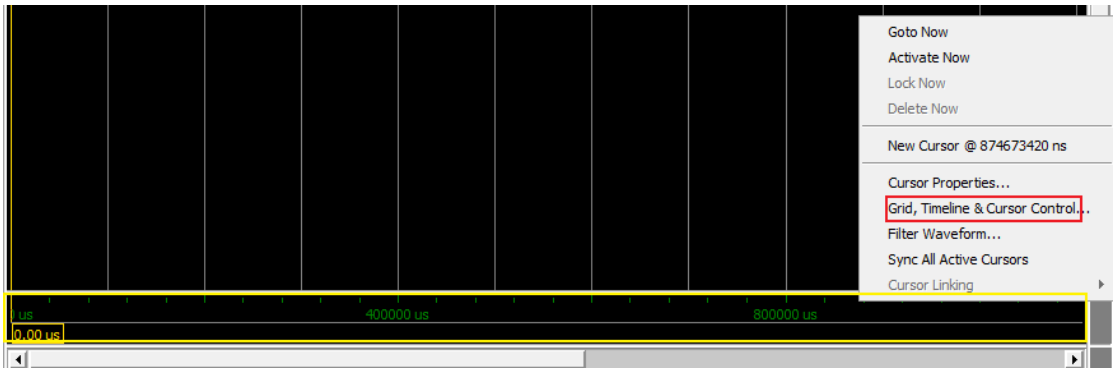


图 5-9 设置仿真显示时间单位入口

（2）出现如下图 5-10 所示的窗口，红色圈主的地方就是设置显示单位的地方，通过下拉框选择将 us/ns 改为 ms。（这里的时间单位根据实际仿真时间长度进行改变，选择一个合适的单位便于观察波形即可）。

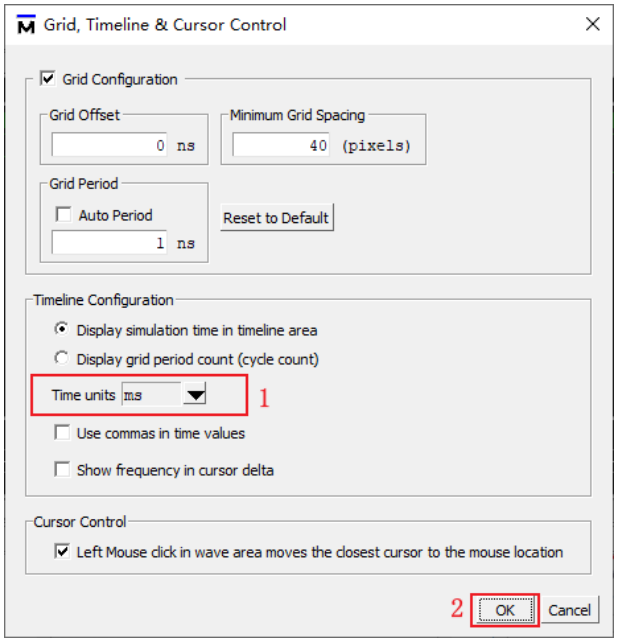


图 5-10 设置仿真显示时间单位界面

更改显示时间单位后去观察波形高低电平的时间，如下图 5-11 所示，显示单位已经更改为 ms 了，高低电平时间为 500ms。与上面是一样的，就是显示时间单位有所变化。

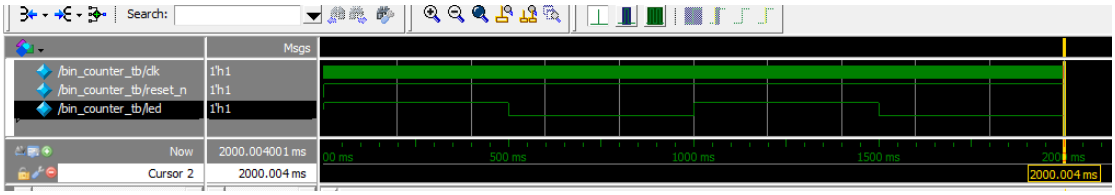


图 5-11 更改仿真显示时间单位后的波形图

在进行上述的功能仿真时可以发现电脑需要运行的时间较长，这是由于计数器的计数值太大。可以通过更改仿真计数的最大值来缩短仿真时间。这里介绍两种更改方式，第一种方式是将设计文件 bin_counter.v 中使用 parameter 进行参数化定义的计数值最大值修改为 24_999，具体代码如下。

```
parameter MCNT = 24_999;
```

第二种方式是在仿真文件 bin_counter_tb.v 中将计数器的最大值修改为 24_999，具体代码如下。

```
defparam bin_counter_inst.MCNT = 24_999;
```

通过上面两种方式在能验证设计正确性的基础上缩短仿真时间。比较上面的不同方式，可以发现，方式二相对来说会好些，方式二能够在不改变设计代码的前提下，仅仅通过仿真代码上的设计就可以实现缩短仿真时间的目的。修改后的仿真波形如下图 5-12 所示，波形图中高低电平变化时间变为 0.5ms，相

比 500ms 缩短了 1000 倍，也可以说明功能仿真正确。需要注意的是，修改代码之后，都需要重新在 Modelsim 重新编译、跑仿真。

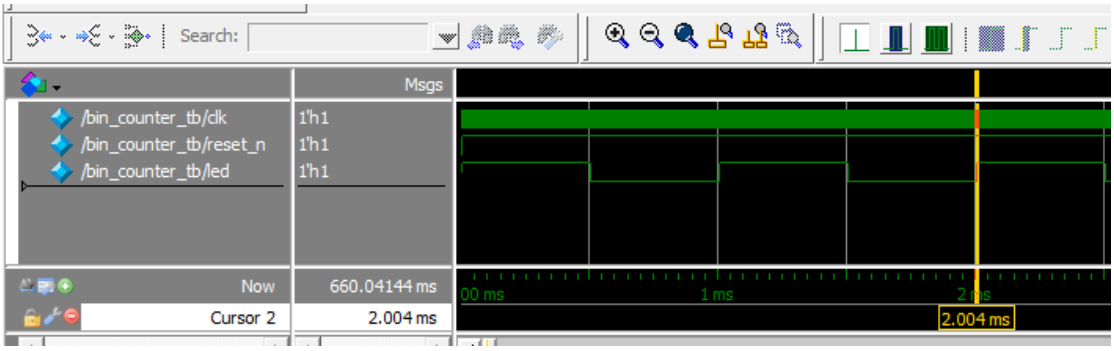


图 5-12 缩小计数值后的功能仿真波形

5.4 板级调试与验证

本次实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的数据流文件下载至开发板。
2. 下载完成后，开发板上的 LED 灯是否每隔 1S 中闪烁一次。

系统所需硬件：

1. 高云开发板。
2. 高云下载器及下载线。
3. 12V 的供电电源
4. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

5.4.1 添加 I/O 约束

在 Gowin 软件右边的 Process 界面点击 FloorPlanner，新建一个.cst 文件存放 I/O 约束，如下图 5-13 所示。

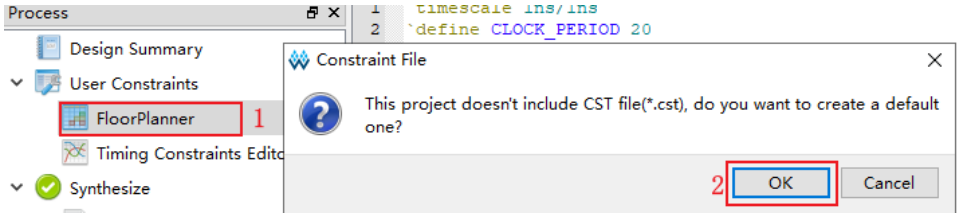


图 5-13 新建 I/O 约束

进入 FloorPlanner 界面之后，点击下方的 I/O Constraints 进入 I/O 约束界面。根据原理图对管脚位置进行添加，添加完成后如下图 5-14 所示。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	clk	input		T9	4	False	LVCMOS33	OFF
2	led	output		D14	1	False	LVCMOS33	8
3	reset_n	input		B16	1	False	LVCMOS33	OFF

图 5-14 引脚分配界面

这样管脚约束添加完成了。但是此时约束内容保存在内存中，还没有写入文件，点击工具栏保存按钮，或者直接 Ctrl+S。

关闭 FloorPlanner 界面之后，点击右侧的 Design，可以看到我们新添加的.cst 文件，双击可查看文件内容，如下图 5-15 所示。

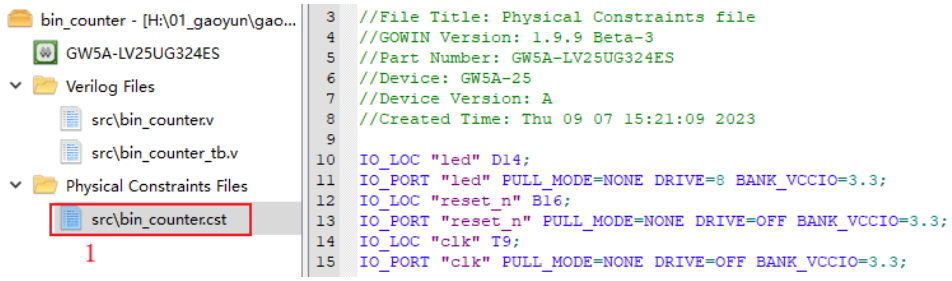


图 5-15 I/O 约束文件

5.4.2 添加时序约束

我们需要点击 Process 界面的“Timing Constraints Editor”进行时序约束，此时会弹出本工程没有约束文件，是否想要创建约束文件，点击 OK 即可，如下图 5-16 所示。

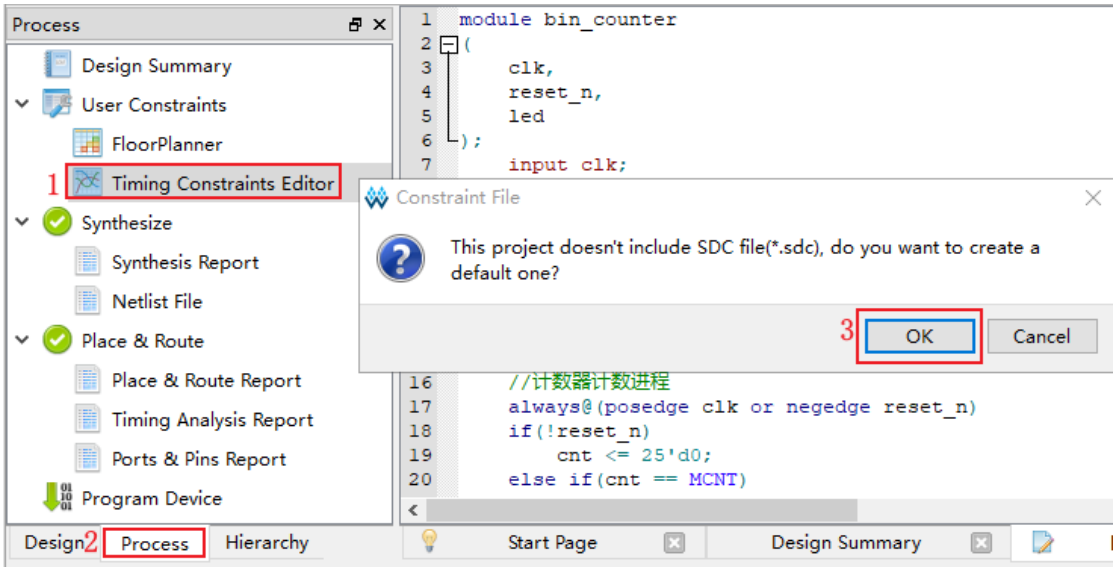


图 5-16 创建时序约束文件

随后进入时序约束界面，如下图 5-17 所示。

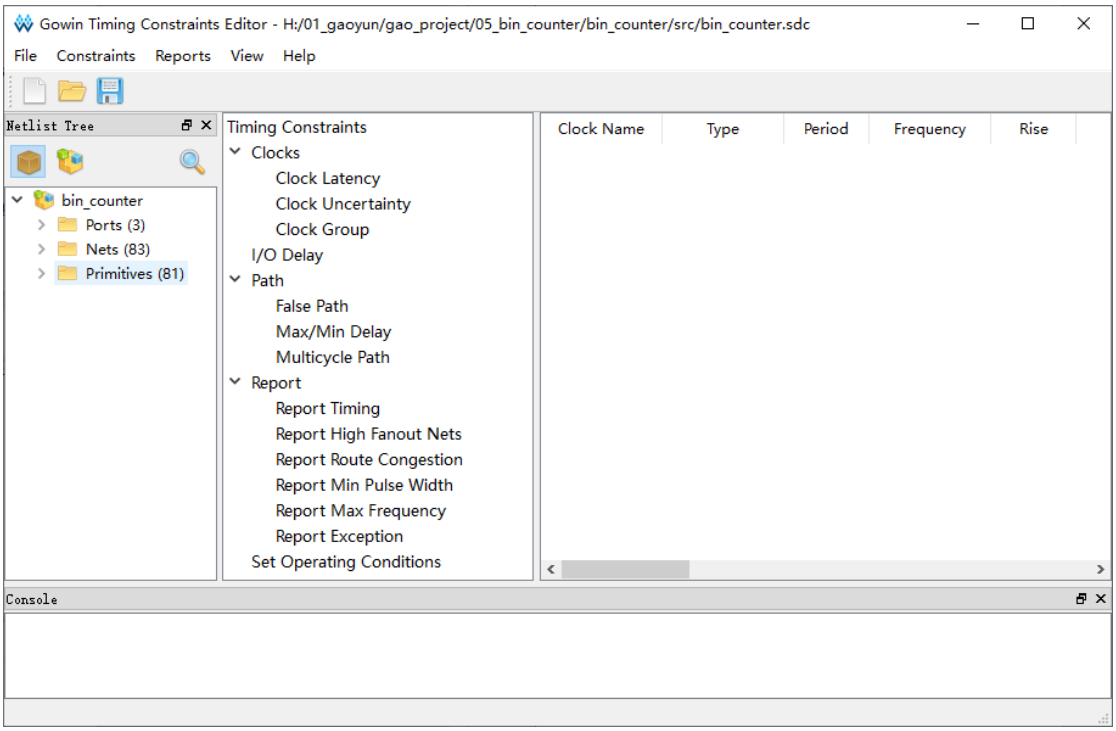


图 5-17 时序约束界面

选中 Timing Constraints 下的 Clocks，在右侧空白处，右键单击选择 Create Clock，如下图 5-18 所示。

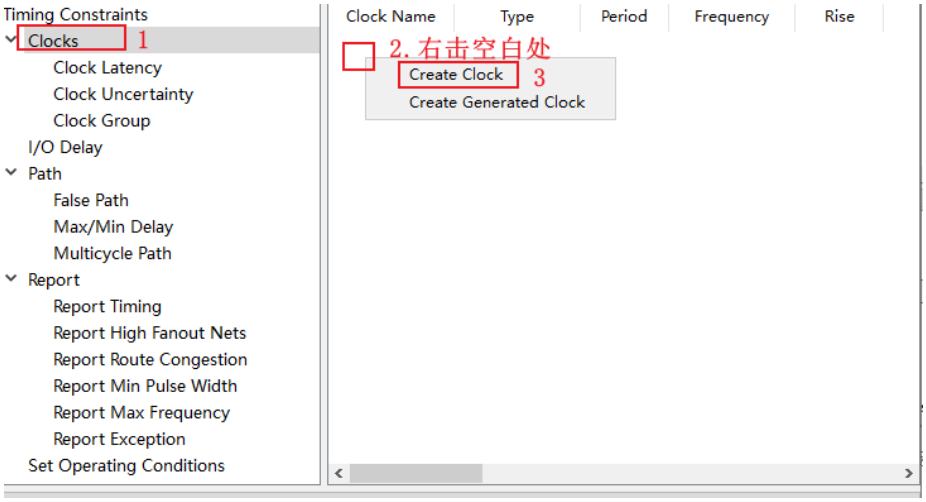


图 5-18 创建时钟约束操作步骤

弹出 Create Clock 界面，如下图 5-19 所示。

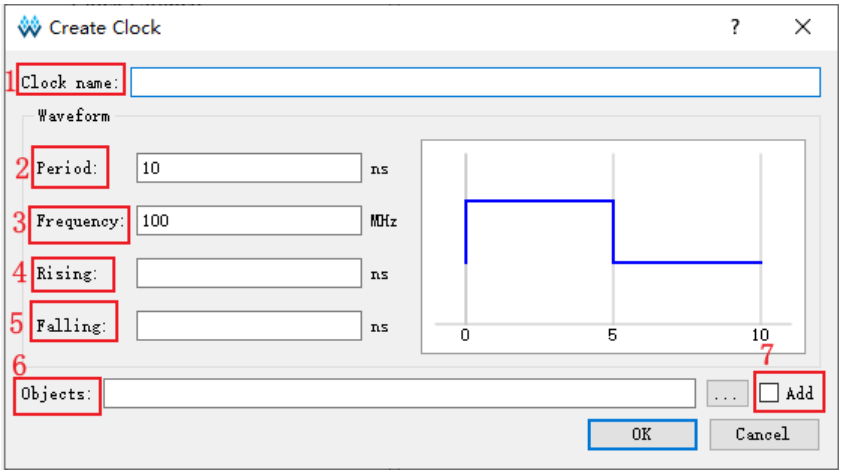


图 5-19 Create Clock 界面

对上图所示所示的各项参数进行简要说明：

1. Clock Name：时钟名，支持字母或下划线开头的标识符。
2. Period：周期，默认 10，浮点型，精确到千分位。
3. Frequency：频率。默认 100，浮点型，精确到千分位。
4. Rising：上升沿时刻，浮点型，精确到千分位。
5. Falling：下降沿时刻，浮点型，精确到千分位。
6. Objects：指定作用目标，通过 ... 选择进行内容的自动填充。
7. Add：在同一源上添加多个时钟时需要勾选。

我们点击 ... 进行约束时钟的选择设置，弹出如下图 5-20 所示的窗口，

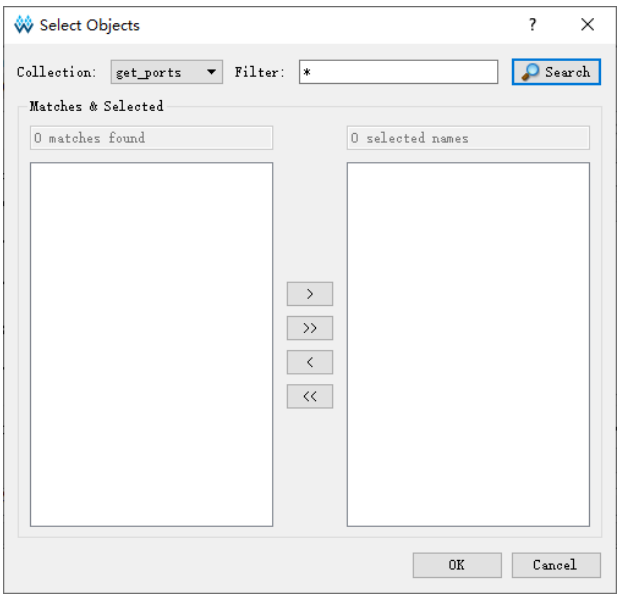


图 5-20 Select Objects 界面

这里需要约束的时钟是从管脚进来的 50M 系统时钟，依次点击 Search->clk-> >->OK，操作如下图 5-21 所示。

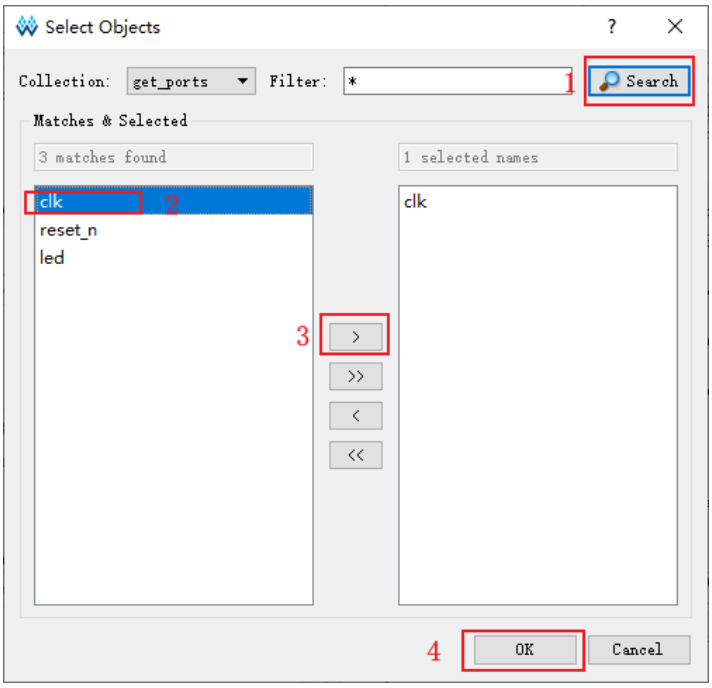


图 5-21 选择需要约束的时钟

然后重新回到 Create Clock 界面，时钟命名为 clk，周期为 20ns，频率为 50Mhz，Rising 为 0，Falling，最终 Create Clock 界面如下图 5-22 所示。

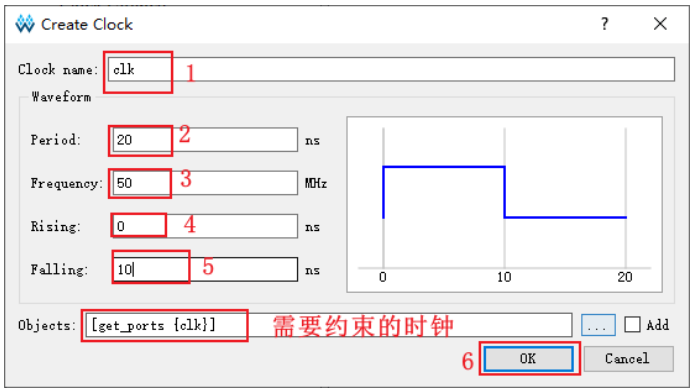


图 5-22 Create Clock 配置界面

添加完成之后，可以看到右侧空白处有了我们刚刚添加的时钟约束，点击 进行保存，如下图 5-23 所示。

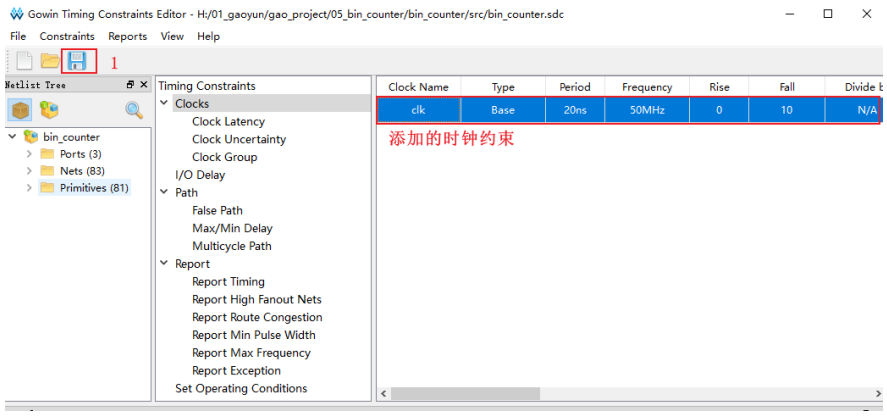


图 5-23 保存时序约束

通过上述操作，完成了 50Mhz 时钟的时序约束，关闭 Gowin Timing Constraints Editor 界面，此时在 Design 界面，可以看到我们添加的时序约束文件 bin_counter.sdc，双击可以查看约束文件内容，如下图 5-24 所示。

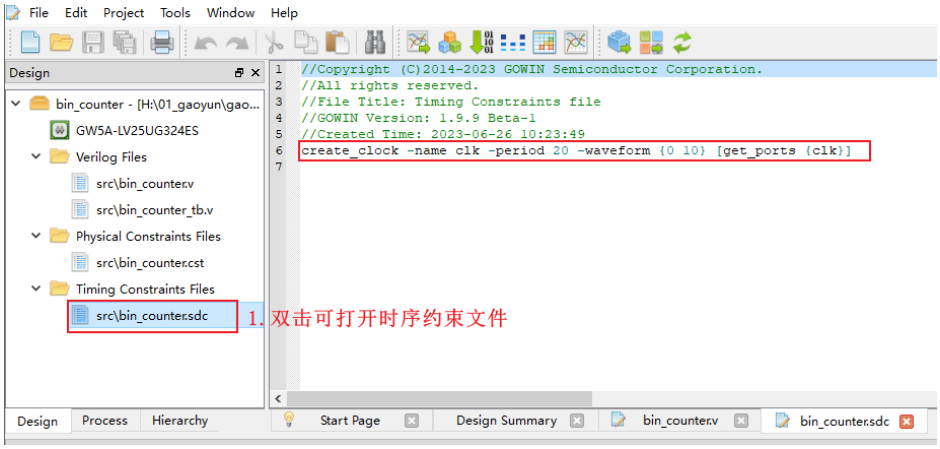


图 5-24 查看时序约束文件方式

5.4.3 布局布线

点击 Process 中的 Place & Route 进行布局布线，布局布线成功之后，会生成 bit 数据流，Console 提示如下图 5-25。

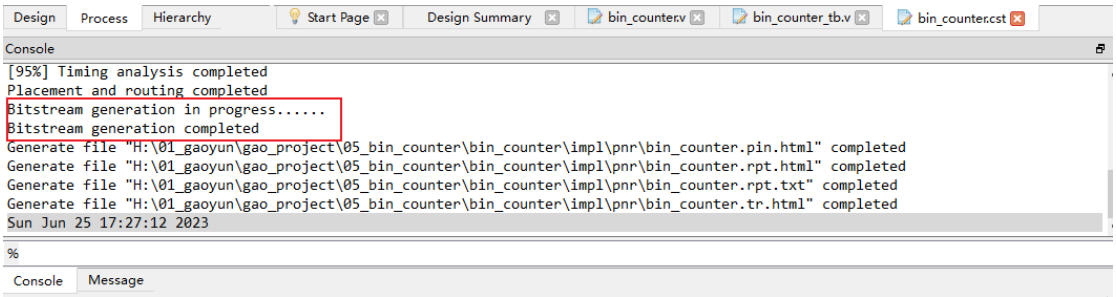


图 5-25 数据流生成成功示意图

5.4.4 硬件连接

将下载器、电源依次连接至开发板，整体的硬件连接如下图 5-26 所示。

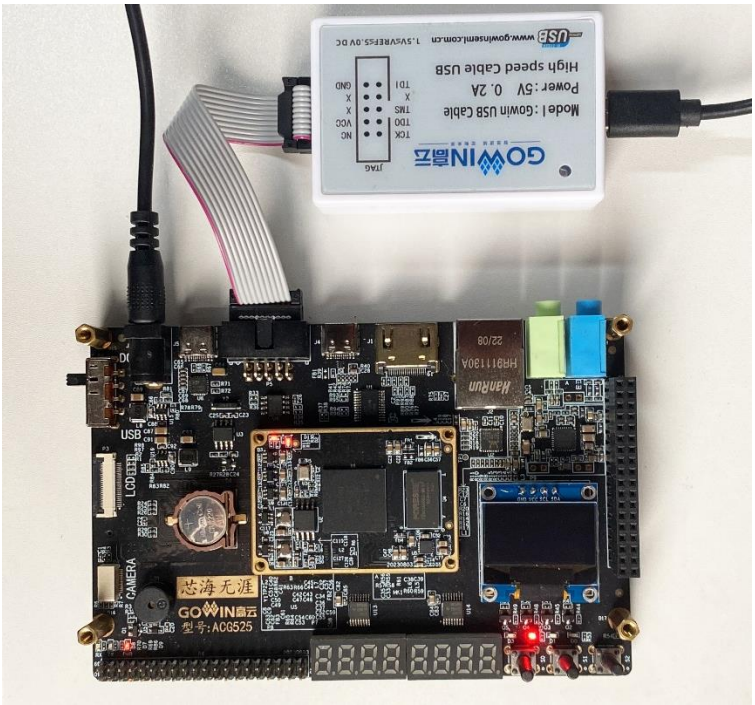


图 5-26 硬件连接图

5.4.5 下载数据流

点击 Program Device，进入下载界面，然后弹出 Cable Setting 界面，点击 Save，然后点击  下载文件，下载完成之后，如下图 5-27 所示。

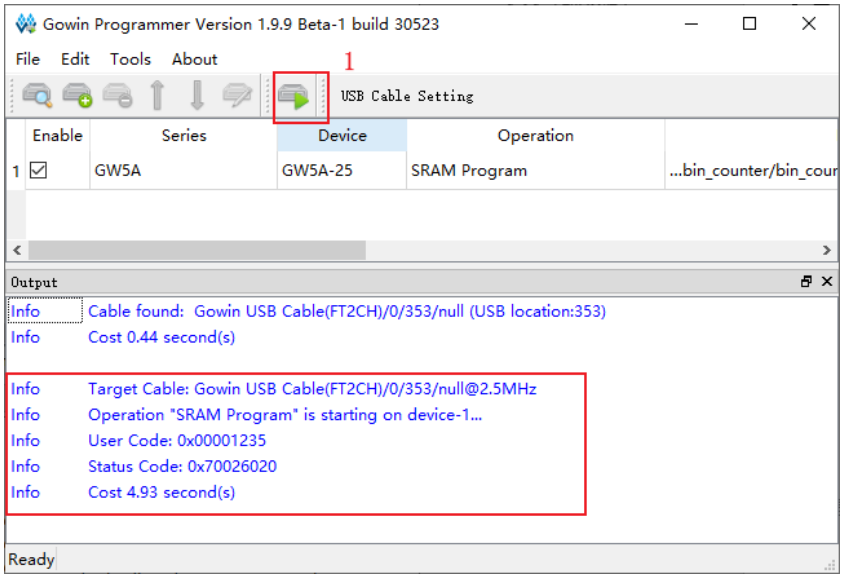


图 5-27 下载成功示意图

Program 完成后，可以看到 LED0 每间隔 1S 亮灭翻转一次。实验证明，设计达到预期。

5.5 思考与总结

本章以计数器为例学习了简单的时序逻辑设计，并简要介绍了一下时钟约束的方法。读者请以此为基础，设计出逻辑电路驱动 LED 灯以不同的频率闪烁，并进行仿真以及板级验证。

6 串口发送模块设计与验证

工程源码	|----02_设计实例 |----ch6_uart_tx
相关视频课程	
说明	

章节导读

在当今的电子系统中，经常需要板内、板间或者下位机与上位机之间进行数据的发送与接收，这就需要双方共同遵循一定的通信协议来保证数据传输的正确性。常见的协议有 UART（通用异步收发传输器）、IIC（双向两线总线）、SPI（串行外围总线）、USB2.0/3.0（通用串行总线）以及 Ethernet（以太网）等。在这些协议当中，最为基础的就是 UART，因其电路结构简单、成本较低，所以在注重性价比的情况下，使用非常广泛。

本章将学习 UART 通信的原理及其硬件电路设计，并使用 FPGA 来实现 UART 通信中的数据发送部分设计。

6.1 异步串行通信原理及电路设计

6.1.1 RS232 通信接口标准

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter，UART）是一种异步收发传输器，其在数据发送时将并行数据转换成串行数据来传输，在数据接收时将接收到的串行数据转换成并行数据，可以实现全双工传输和接收。它包括了 RS232、RS449、RS423、RS422 和 RS485 等接口标准规范和总线标准规范。换句话说，UART 是异步串行通信的总称。而 RS232、RS449、RS423、RS422 和 RS485 等，是对应各种异步串行通信口的接口标准和总线标准，它们规定了通信口的电气特性、传输速率、连接特性和接口的机械特性等内容。

本章要重点学习的 RS-232 是美国电子工业联盟（EIA）制定的串行数据通信的接口标准，原始编号全称是 EIA-RS-232（简称 232，RS232），被广泛用于计算机串行接口外设连接。其 DB9 接口的针脚定义如下图 6-1 所示，引脚功能如下表 6-1 针脚功能所示。若系统存在多个 UART 接口，则可分别称为 COM1、COM2 等。

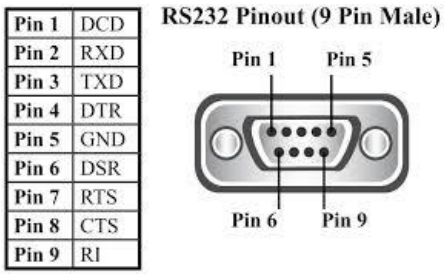


图 6-1 RS232,D89 针脚定义

表 6-1 针脚功能

脚位	简写	意义	说明
Pin1	CD	Carrier Detect	调制解调器通知电脑有载波被侦测到；
Pin2	RXD	Receiver	接收数据；
Pin3	TXD	Transmit	发送数据；
Pin4	DTR	Data Terminal Ready	电脑告诉调制解调器可以进行传输；
Pin5	GND	Ground	地线；
Pin6	DSR	Data Set Ready	调制解调器告诉电脑一切准备就绪；
Pin7	RTS	Request To Send	电脑要求调制解调器将数据提交；
Pin8	CTS	Clear To Send	调制解调器通知电脑可以传数据过来；
Pin9	RI	Ring Indicator	调制解调器通知电脑有电话进来。

6.1.2 UART 关键参数及时序图

UART 通信在使用前需要做多项设置，最常见的设置包括数据位数、波特率大小、奇偶校验类型和停止位数。

数据位（Data bits）：该参数定义单个 UART 数据传输在开始到停止期间发送的数据位数。可选择为：5、6、7 或者 8（默认）。

波特率（Baud）：是指从一设备发到另一设备的波特率，即每秒钟可以通信的数据比特个数。典型的波特率有 300, 1200, 2400, 9600, 19200, 115200 等。一般通信两端设备都要设为相同的波特率，但有些设备也可设置为自动检测波特率。

奇偶校验类型（Parity Type）：是用来验证数据的正确性。奇偶校验一般不使用，如果使用，则既可以做奇校验（Odd）也可以做偶校验（Even）。在偶校验中，因为奇偶校验位会被相应的置 1 或 0（一般是最高位或最低位），所以数据会被改变以使得所有传送的数位（含字符的各数位和校验位）中“1”的个数为偶数；在奇校验中，所有传送的数位（含字符的各数位和校验位）中“1”的个数为奇数。奇偶校验可以用于接收方检查传输是否发送生错误，如果某一字节中“1”的个数发生了错误，那么这个字节在传输中一定有错误发生。如果奇偶校验是正确的，那么要么没有发生错误，要么发生了偶数个的错误。如果用

户选择数据长度为 8 位，则因为没有多余的比特可被用来作为奇偶校验位，因此就叫做“无奇偶校验（Non）”。

停止位（Stop bits）：在每个字节的数据位发送完成之后，发送停止位，来标志着一次数据传输完成，同时用来帮助接收信号方硬件重同步。可选择为：1（默认）、1.5 或者 2 位。

在 RS-232 标准中，最常用的配置是 8N1(即八个数据位、无奇偶校验、一个停止位)，其发送一个字节时序图如下图 6-2 所示。按照一个完整的字节包括一位起始位、8 位数据位、一位停止位即总共十位数据来算，要想完整的实现这十位数据的发送，就需要 11 个波特率时钟脉冲，第 1 个脉冲标记一次传输的起始，第 11 个脉冲标记一次传输的结束。

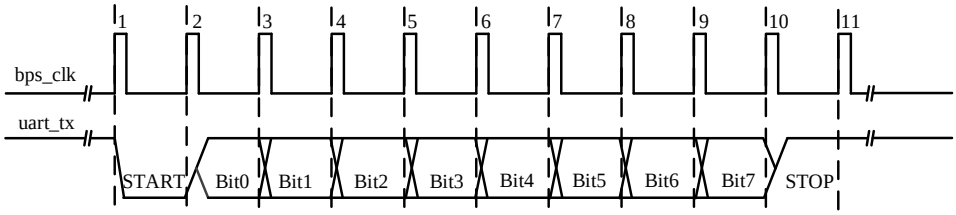


图 6-2 UART 发送一个字节时序图

bps_clk 信号的第一个上升沿到来时，字节发送模块开始发送起始位，接下来的 2 到 9 个上升沿，发送 8 个数据位，第 10 个上升沿到第 11 个上升沿为停止位的发送。

6.1.3 RS232 通信电路设计

RS232 通信协议需要一定的硬件支持，早期大多使用的方案是 RS232 转 TTL，这时需要 MAX232 或者 SP3232 等电平转换芯片来做数据转换。其外围电路简单，最少只需要 4 个电容即可正常工作，其典型电路图如下图所示。在这里只使用了两路通信中的一路，且通过加入的 D7、D8 两个发光二极管可以更好的观察数据状态。

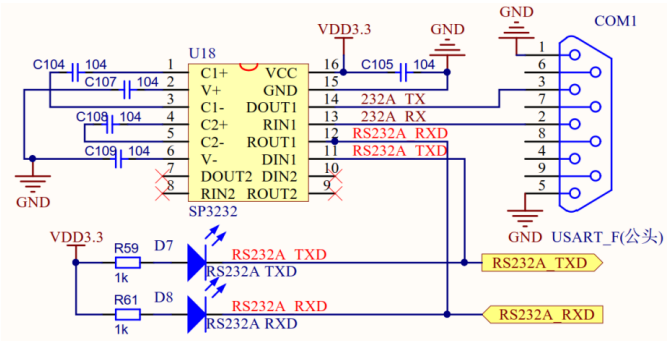


图 6-3 RS232 转 TTL 电路图

现在系统集成度越来越高，DB9 的 RS232 接口占用 PCB 面积过大，多数系统已经转用 USB 转串口，高云开发板也是采用的是 USB 转串口方案，其电路图如下图 6-4 所示。

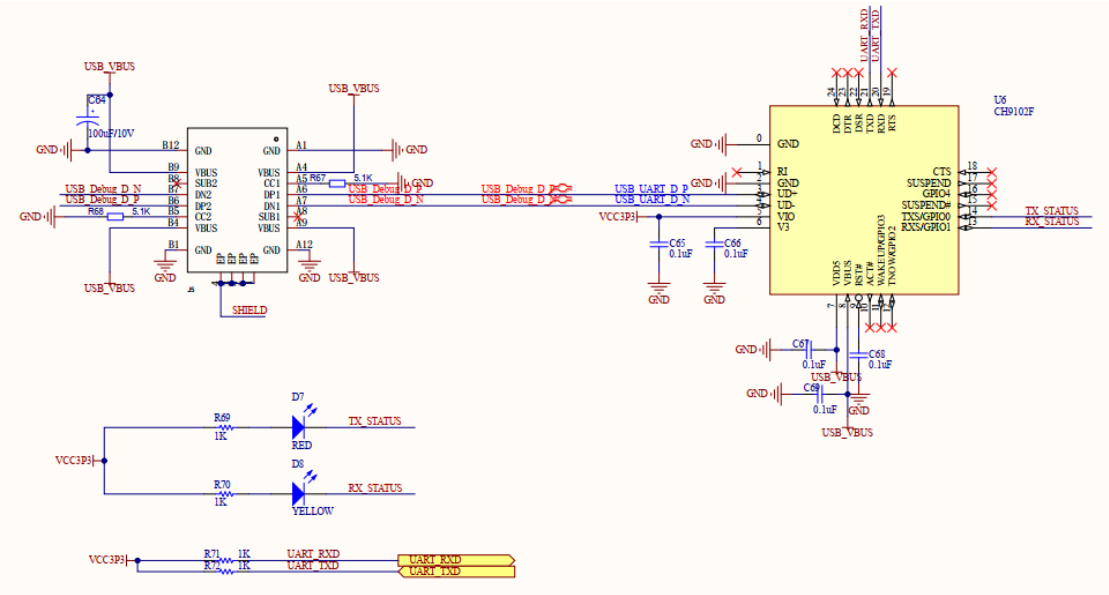


图 6-4 高云开发板串口电路图

CH9102F 是一个 USB 总线的转接芯片，实现 USB 转高速异步串口。提供了常用的 MODEM 联络信号，用于为计算机扩展异步串口，或者将普通的串口设备或者 MCU 直接升级到 USB 总线。其特点如下所示：

1. 全速 USB 设备接口，兼容 USB V2.0。
2. 内置固件，仿真标准串口，用于升级原串口外围设备，或者通过 USB 增加额外串口。
3. 支持免安装的操作系统内置 CDC 类驱动程序或者多功能高速率的 VCP 厂商驱动程序。
4. 硬件全双工串口，内置独立的收发缓冲区，支持通讯波特率 50bps～4Mbps。
5. 串口支持 5、6、7 或者 8 个数据位，支持奇校验、偶校验、空白、标志以及无校验。
6. 支持常用的 MODEM 联络信号 RTS、DTR、DCD、RI、DSR、CTS。支持 CTS 和 RTS 硬件自动流控。
7. 支持半双工，提供正在发送状态 TNOW 支持 RS485 切换。

8. 通过外加电平转换器件，支持 RS232 接口。
9. CH9102F 串口 I/O 独立供电，支持 5V、3.3V 和 2.5V 甚至 1.8V 电源电压。
10. CH9102X 串口 I/O 支持 3.3V 信号。
11. 内置上电复位，内置时钟，无需外部晶振。
12. CH9102F 内置 EEPROM，可配置芯片 VID、PID、最大电流值、厂商和产品信息字符串等参数。
13. 芯片内置 Unique ID(USB Serial Number)。
14. 提供 QFN24 和 QFN28 无铅封装，兼容 RoHS。

6.1.4 UART 驱动安装

在第一次使用串口时，我们需要先安装驱动， 操作步骤如下所示：

1. 首先去南京沁恒公司官网下载安装驱动的软件，官网如下所示：

https://www.wch.cn/

进入官网之后找到首页的 USB 芯片，然后进入新界面之后，依次选择 USB 转 UART、CH9102，点击之后，进入 CH9102 介绍界面，往下滑，找到 CH343SER.EXE 进行下载，整个操作步骤如下图 6-5 所示。

图 6-5 官网下载串口驱动安装软件

2. 双击打开软件，如果提示安全警告对话框，直接点击运行即可，如下图 6-6 所示。

图 6-6 运行软件

3. 打开之后，界面如下图 6-7 所示，直接点击安装即可。

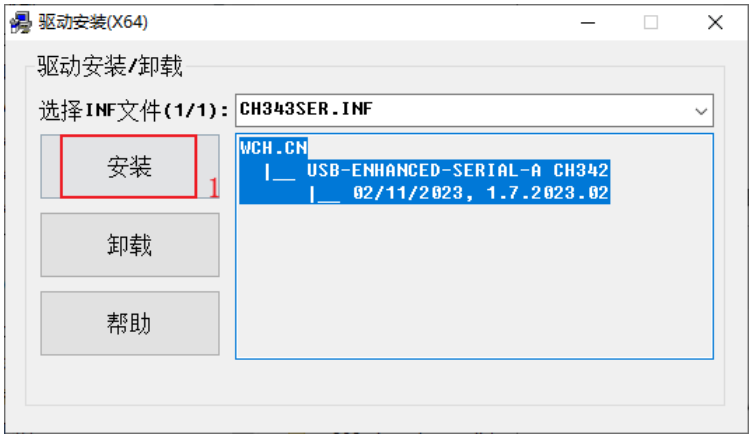


图 6-7 安装驱动

4. 等待安装，安装成功之后，会弹出预处理安装成功提示对话框，如下所示，此时说明我们驱动安装成功，点击确定即可。

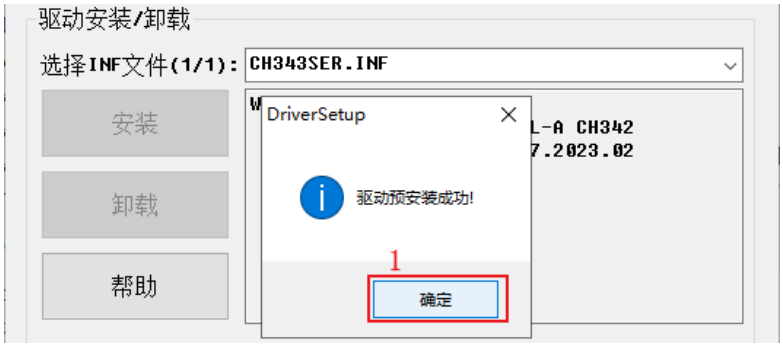


图 6-8 驱动安装成功提示对话框

6.2 UART 异步串口通信发送模块设计与实现

基于上述原理，本章要实现的串口发送模块整体框图，如下图 6-9 所示，其接口列表如下表 6-2 所示。

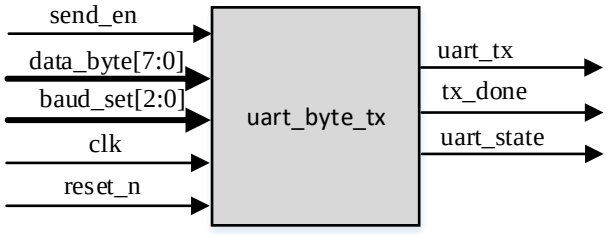


图 6-9 串口发送模块整体设计框图

表 6-2 模块接口列表

信号名称	I/O	功能描述
clk	I	模块全局时钟 50MHz
reset_n	I	模块全局复位信号
data_byte	I	待传输 8bit 数
send_en	I	发送使能信号
baud_set	I	波特率设置信号
uart_tx	O	串口发送信号输出
tx_done	O	发送结束信号，一个时钟周期高电平
uart_state	O	发送状态，处于发送状态时为 1

根据功能需求，串口发送模块可进一步细化为如下图 6-10 所示详细结构图，其中每一子模块的作用如下表 6-3 所示。其中绿色的框代表单一结构的寄存器，来实现数据的稳定输入以及输出。

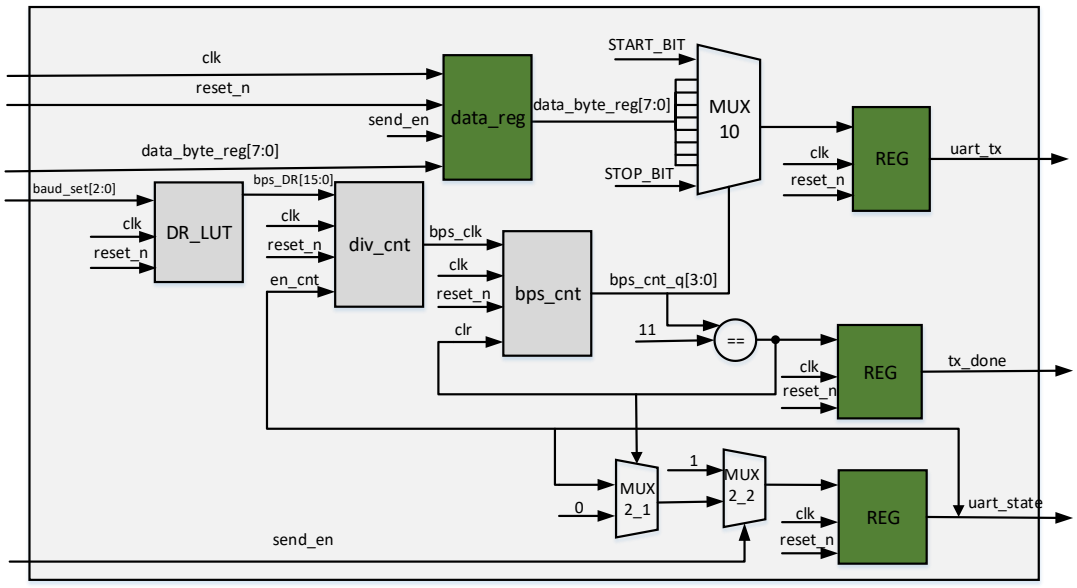


图 6-10 串口发送模块结构图

表 6-3 子功能模块功能描述

名称	功能描述
div_cnt	产生波特率时钟
bps_cnt	对波特率时钟进行计数
DR_LUT	查找表，以适用于不同波特率
MUX10	根据 bps_cnt 的值来确定需要发送位
“==”运算符	判断一次传输是否结束
MUX2	模块的使能信号

6.2.1 波特率时钟生成模块设计

从原理部分可知，波特率是 UART 通信中需要设置的参数之一。在波特率时钟生成模块中，计数器需要的计数值与波特率之间的关系如下表 6-4 所示，其中系统时钟周期为 System_clk_period，这里为 20ns。如果接入到该模块的时钟频率为其他值，需要根据具体的频率值修改该参数。

表 6-4 波特率计算

baud_set	波特率	波特率周期	波特率分频计数值	50M 系统时钟计数值
0	9600	104167ns	104167/ System_clk_period	5208-1
1	19200	52083ns	52083/ System_clk_period	2604-1
2	38400	26041ns	26041/ System_clk_period	1302-1
3	57600	17361ns	17361/ System_clk_period	868-1
4	115200	8680ns	8680/ System_clk_period	434-1

本模块的设计是为了保证模块的复用性。当需要不同的波特率时，只需设置不同的波特率时钟计数器的计数值。使用查找表即可实现，下面的设计代码中只包含了针对 5 个波特率的设置，如需要其他波特率可根据实际使用情况具体修改。

```
always@(posedge clk or posedge reset)
if(reset)
    bps_DR <= 16'd5207;
else begin
    case(baud_set)
        0:bps_DR <= 16'd5207;
        1:bps_DR <= 16'd2603;
        2:bps_DR <= 16'd1301;
        3:bps_DR <= 16'd867;
        4:bps_DR <= 16'd433;
        default:bps_DR <= 16'd5207;
    endcase
end
```

我们这里利用计数器生成波特率时钟。

```
//counter
```

```
always@(posedge clk or posedge reset)
if(reset)
    div_cnt <= 16'd0;
else if(uart_state)begin
    if(div_cnt == bps_DR)
        div_cnt <= 16'd0;
    else
        div_cnt <= div_cnt + 1'b1;
end
else
    div_cnt <= 16'd0;

// bps_clk gen
always@(posedge clk or posedge reset)
if(reset)
    bps_clk <= 1'b0;
else if(div_cnt == 16'd1)
    bps_clk <= 1'b1;
else
    bps_clk <= 1'b0;
```

所谓波特率生成，就是用一个定时器来定时，产生频率与对应波特率时钟频率相同的时钟信号。例如，我们使用波特率为 115200bps，则我们需要产生一个频率为 115200Hz 的时钟信号。那么如何产生这样一个 115200Hz 的时钟信号呢？这里，我们首先将 115200Hz 时钟信号的周期计算出来，1 秒钟为 1000_000_000ns，因此波特率时钟的周期 Tb= 1000000000/115200 =8680.6ns，即 115200 bps 的一个周期为 8680.6ns，那么，我们只需要设定我们的定时器定时时间为 8680.6ns，每当定时时间到，产生一个系统时钟周期长度的高脉冲信号即可。系统时钟频率为 50MHz，即周期为 20ns，那么，我们只需要计数 8680/20 个系统时钟，就可获得 8680ns 的定时，bps 115200=Tb/Tclk - 1=Tb*fclk - 1=fclk/115200-1。相应的，其它波特率定时值的计算与此相同。

为了能够通过外部控制波特率，设计中使用了一个 3 位的波特率选择端口：baud_set。通过给此端口不同的值，就能选择不同的波特率，此端口控制不同波特率的原理很简单，就是一个多路选择器，多路选择器通过选择不同的定时器计数最大值来设置不同的比特率时钟频率。baud_set 的值与各波特率的对应关系如下所示。

表 6-5 波特率和 baud_set 参数值对应关系

baud_set 值	波特率
000	9600bps
001	19200bps
010	38400bps

011	57600bps
100	115200bps

6.2.2 数据输出模块设计

通过对波特率时钟进行计数，来确定数据发送循环状态，代码如下：

```
//bps counter
always@(posedge clk or posedge reset)
if(reset)
    bps_cnt <= 4'd0;
else if(bps_cnt == 4'd11)
    bps_cnt <= 4'd0;
else if(bps_clk)
    bps_cnt <= bps_cnt + 1'b1;
else
    bps_cnt <= bps_cnt;
```

同样为了使得模块可以对其他模块进行控制或者调用，这里产生一个 byte 传送结束的信号。一个数据位传输结束后 tx_done 信号输出一个时钟的高电平。

```
always@(posedge clk or posedge reset)
if(reset)
    tx_done <= 1'b0;
else if(bps_cnt == 4'd11)
    tx_done <= 1'b1;
else
    tx_done <= 1'b0;
```

产生数据传输状态信号，即当在正常传输的时候 uart_state 信号为高电平，其他情况均为低电平。这里实现的电路结构同样是具有优先级顺序的，但与 C 语言本质是不同的。在图 6-10 中的 MUX2_1 与 MUX2_2 就是下面的设计实现的 if—else if—else 的电路结构。

```
always@(posedge clk or posedge reset)
if(reset)
    uart_state <= 1'b0;
else if(send_en)
    uart_state <= 1'b1;
else if(bps_cnt == 4'd11)
    uart_state <= 1'b0;
else
    uart_state <= uart_state;
```

由于串口是一个异步的收发器，因此为了保证发送的数据在时钟到来的时候是稳定的，这里也需要对输入数据进行寄存。

```
always@(posedge clk or posedge reset)
if(reset)
    data_byte_reg <= 8'd0;
```

```
else if(send_en)
    data_byte_reg <= data_byte;
else
    data_byte_reg <= data_byte_reg;
```

6.2.3 数据传输状态控制模块

在模块结构图 6-10 中还有一个十选一多路器，作用是根据 bps_cnt 的值来确定数据传输的状态。如时序图 6-2 所示，在不同的波特率时钟计数值时，有对应的传输数据。

```
always@(posedge clk or posedge reset)
if(reset)
    uart_tx <= 1'b1;
else begin
    case(bps_cnt)
        0:uart_tx <= 1'b1;
        1:uart_tx <= START_BIT;
        2:uart_tx <= data_byte_reg[0];
        3:uart_tx <= data_byte_reg[1];
        4:uart_tx <= data_byte_reg[2];
        5:uart_tx <= data_byte_reg[3];
        6:uart_tx <= data_byte_reg[4];
        7:uart_tx <= data_byte_reg[5];
        8:uart_tx <= data_byte_reg[6];
        9:uart_tx <= data_byte_reg[7];
        10:uart_tx <= STOP_BIT;
        default:uart_tx <= 1'b1;
    endcase
end
```

至此，代码设计完成，对设计进行分析综合直到没有错误。

6.3 激励创建及仿真测试

从上面分析生成的原理图与我们设计的模块结构图基本上一致，但这并不表示设计的模块的功能就能达到我们要求。这就需要对设计模块进行功能仿真，验证模块功能的正确性。首先设计仿真 testbench 文件，在仿真文件中产生了周期为 20ns 的时钟 clk 作为串口模块的工作时钟，然后生成了复位信号以及使能信号、待传输数据。这里将所有数据变化与系统时钟错开 1ns，是为了能更清楚看到输入输出数据与时钟的时序关系。

```
initial clk = 1;
always#(`CLK_PERIOD/2)clk = ~clk;
```

```
initial begin
    reset_n = 1'b0;
    data_byte = 8'd0;
    send_en = 1'd0;
    baud_set = 3'd4;
    #(`CLK_PERIOD*500 + 1 )
    reset_n = 1'b1;
    #(`CLK_PERIOD*50);

//send first byte
    data_byte = 8'haa;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;

    @(posedge tx_done)
    #(`CLK_PERIOD*5000);

//send second byte
    data_byte = 8'h55;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;

    @(posedge tx_done)
    #(`CLK_PERIOD*5000);
    $stop;
end
```

然后打开 Modelsim 新建功能，得到如下图 6-11 所示的波形文件。可以看出在复位和使能信号有效之前输出信号 uart_tx 均为 1，在复位结束以及使能后输出信号才开始发送一组串口数据，且当待发送数据为 0xaa 时，串行输出信号依次为1、0（起始位）、01010101b（LSB）、1（停止位）；当待发送数据为0x55，输出信号依次为 1、0（起始位）、10101010b（LSB）、1（停止位）。同时在发送串行数据过程中 uart_state 处于发送状态时为 1，每个字节数据发送完成后，tx_done 产生一个时钟周期脉冲，与设计的期望一致，仿真通过。

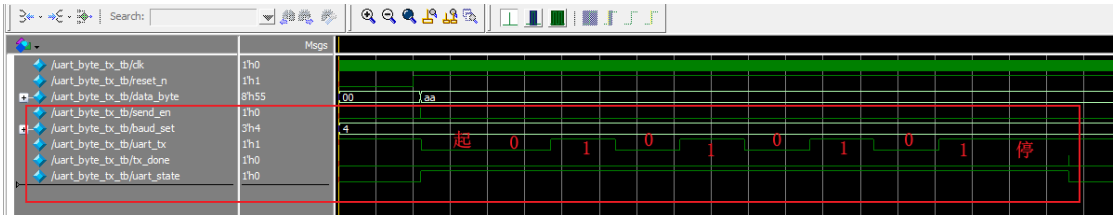


图 6-11 单 byte 数据传输仿真波形图

6.4 串口发送测试模块

我们对单字节串口发送模块进行板级验证之前，需要编写一个串口发送测试模块，去控制单字节串口发送模块何时发送数据以及发送的数据内容。这里我们的设计思路是：通过一个计数器，计数时间为 1S，当计数时间到了以后，产生 send_en 信号，当一次发送完成后，产生 tx_done 之后，需要发送的数据 data_byte 自加 1，这样我们就可以观察到串口每隔 1S 发送一个字节的数据且数据是递增的，baud_set 设置为 3'd0，对应的波特率也就是 9600，串口发送测试模块的代码如下所示：

```
module uart_tx_test(
    input clk,
    input reset_n,

    output uart_tx,
    output led
);
    parameter MCNT = 49_999_999;    //定时 1S

    reg [7:0] data_byte;
    reg send_en;

    //设置定时器定时发送数据
    reg [25:0]cnt;  //定义计数器寄存器
    wire tx_done;

    //计数器计数进程
    always@(posedge clk or negedge reset_n)
    if(!reset_n)
        cnt <= 25'd0;
    else if(cnt == MCNT)
        cnt <= 25'd0;
    else
        cnt <= cnt + 1'b1;

    always@(posedge clk or negedge reset_n)
    if(!reset_n)
        send_en <= 1'b0;
    else if(cnt == MCNT)
        send_en <= 1'b1;
    else
        send_en <= 1'b0;

    always@(posedge clk or negedge reset_n)
```

```
    if(!reset_n)
        data_byte <= 8'b0;
    else if(tx_done)
        data_byte <= data_byte + 1'b1;
    else
        data_byte <= data_byte;

    uart_byte_tx uart_byte_tx(
        .clk(clk),
        .reset_n(reset_n),

        .data_byte(data_byte),
        .send_en(send_en),
        .baud_set(3'd0),

        .uart_tx(uart_tx),
        .tx_done(tx_done),
        .uart_state(led)
    );
endmodule
```

6.5 板级验证

本次实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的数据流文件下载至开发板。
2. 打开电脑上的串口调试助手，可以看到每隔 1S 接收的数据从 00 开始递增。

系统所需硬件：

5. 高云开发板。
6. 高云下载器及下载线。
7. 12V 的供电电源。
8. Type-C 线一根。
9. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

6.5.1 添加 I/O 约束

在 Gowin 软件右边的 Process 界面点击 FloorPlanner，新建一个.cst 文件存放 I/O 约束，如下图 6-12 所示。

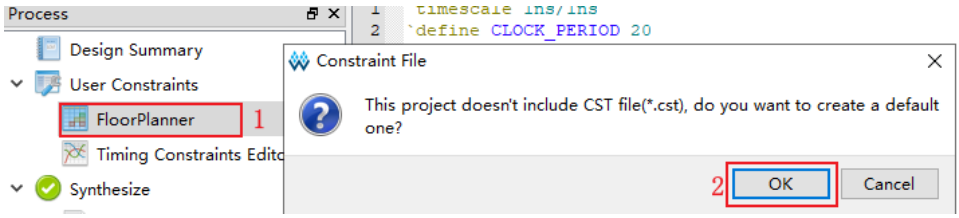


图 6-12 新建 I/O 约束

进入 FloorPlanner 界面之后，点击下方的 I/O Constraints 进入 I/O 约束界面。根据原理图对管脚位置进行添加，添加完成后如下图 6-13 所示。

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	clk	input		T9	4	False	LVCMOS33
2	led	output		V12	4	False	LVCMOS33
3	reset_n	input		B16	1	False	LVCMOS33
4	uart_tx	output		V8	5	False	LVCMOS33

图 6-13 引脚分配界面

这样管脚约束添加完成了。但是此时约束内容保存在内存中，还没有写入文件，点击工具栏保存按钮，或者直接 Ctrl+S。

6.5.2 布局布线

点击 Process 中的 Place & Route 进行布局布线，布局布线成功之后，会生成 bit 数据流，Console 提示如下图 6-14。

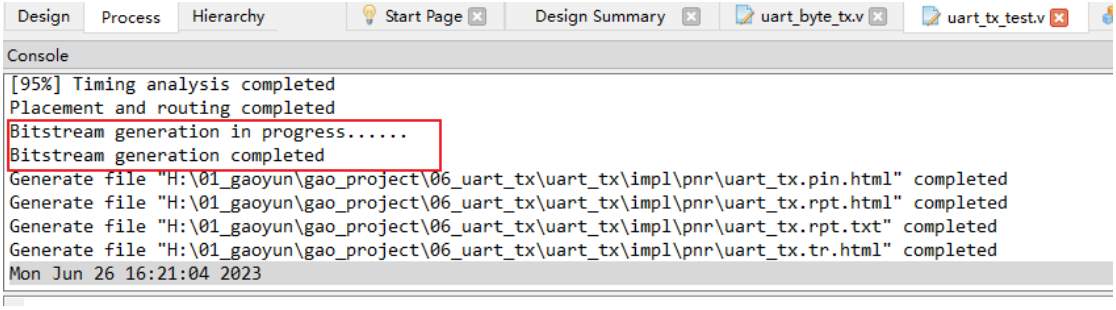


图 6-14 数据流生成成功示意图

6.5.3 硬件连接

将下载器、电源、串口线依次连接至开发板，整体的硬件连接如下图 5-26 所示。可以选择通过电源供电或者通过 USB 供电，用户可以自行选择。

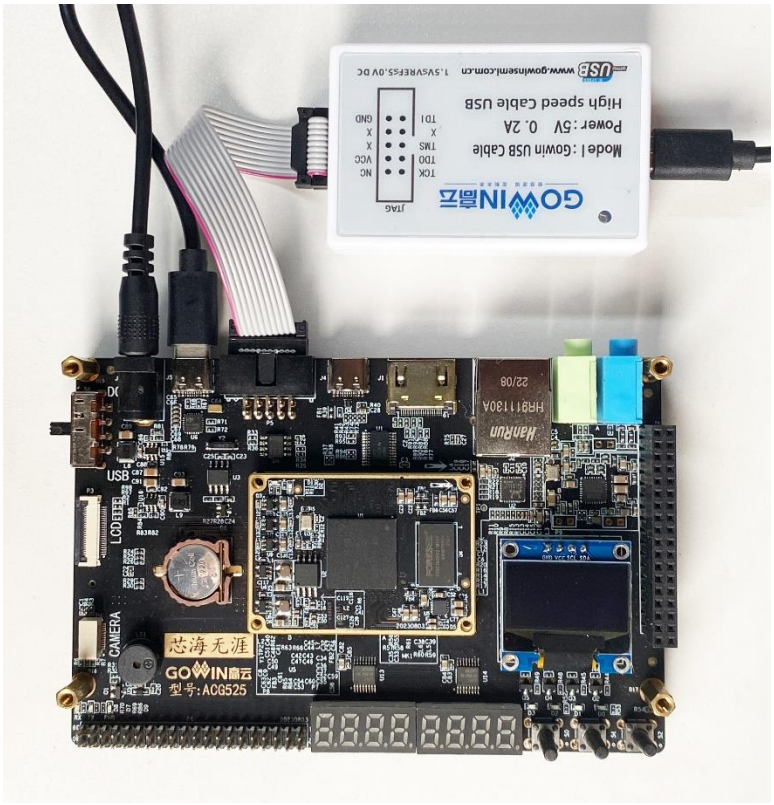


图 6-15 硬件连接图

6.5.4 配置串口调试助手

在 PC 机上打开一个串口软件，这里使用串口猎人，串口软件上设置开发板对应的串口号，可以打开设备管理器，查看端口号（开发板需要上电），如下图 6-16 所示。

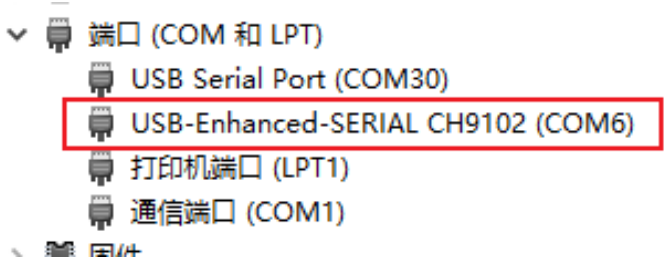


图 6-16 设备管理器对应端口号

然后设置波特率 9600，数据位 8bit，停止位 1bit，然后点击启动串行端口，可以在串口软件下面打印信息窗口看到具体连接情况，如下图 6-17 所示。

图 6-17 串口猎人工具的串口接收设置

6.5.5 下载数据流

点击 Program Device，进入下载界面，然后弹出 Cable Setting 界面，点击 Save，然后点击  下载文件，下载完成之后，如下图 6-18 所示。

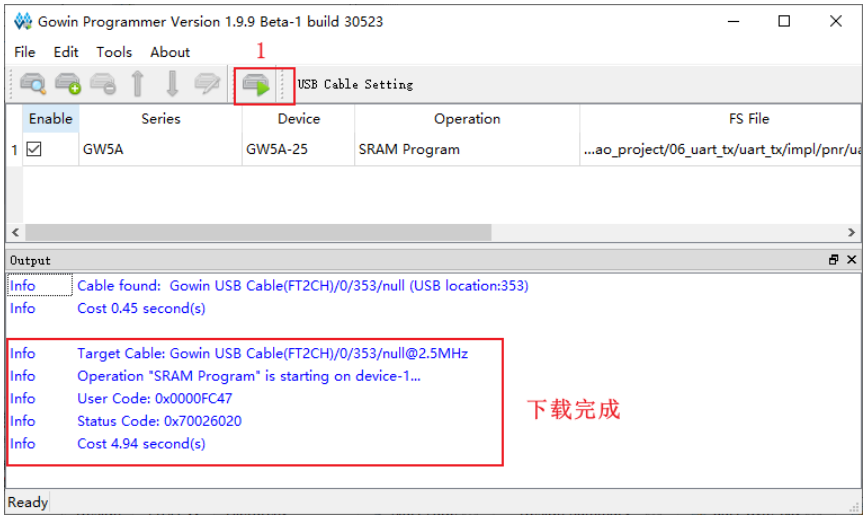


图 6-18 下载成功示意图

Program 完成之后，运行一会儿，可以看到串口助手接收到从 00 开始递增的数据，如下图 6-19 所示，可以看出满足设计预期，接收和发送的数据一致。

图 6-19 串口猎人接收数据

6.6 思考与总结

在本章中学习了 UART 的分类以及原理，介绍了 UART 协议的数据格式以及相关参数所代表的含义，并设计了串口的硬件电路，在板级调试时，通过编写测试文件控制串口发送的数据和控制信号，实现了 FPGA 发送数据在串口上位机上显示的功能。

7 串口接收模块设计与验证

工程源码	|----02_设计实例 |----ch7_uart_rx
相关视频课程	
说明	

章节导读

本章将学习 UART 的数据接收设计部分，针对实际使用中强电磁干扰可能会对数据的影响，本节提出一种改进型的串口接收模块设计方式。

7.1 串口接收原理分析

上一节中学习了串口发送模块的设计与实现，其 UART 发送端发送一个字节数据时序图如下图 7-1 所示。

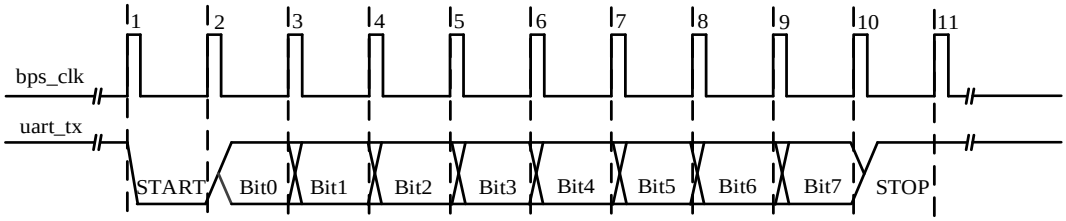


图 7-1 发送一字节的数据时序图

这一讲介绍串口接收模块的设计与实现。当对于数据线上的每一位进行采样时，一般情况下认为每一位数据的中间点是最稳定的。因此一般应用中，采集中间时刻时的电平即认为是此位数据的电平，如下图 7-2 所示。

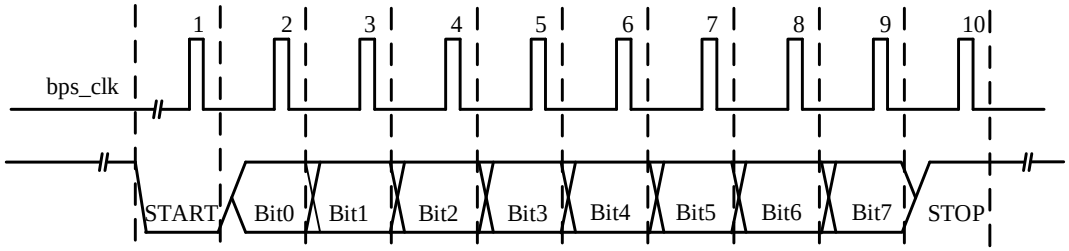


图 7-2 串口接收时序图

但是在实际工业应用中，现场往往有非常强的电磁干扰，只采样一次就作为该数据的电平状态是不可靠的。很有可能恰好采集到被干扰的信号而导致结果出错，因此这里提出以下改进型的单 bit 数据接收方式示意图，使用多次采样求概率的方式进行状态判定，如下图 7-3 所示。

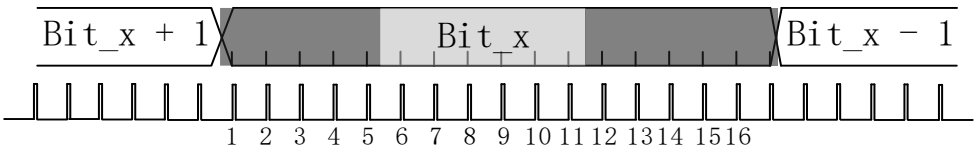


图 7-3 改进型串口接收方式示意图

在上图中，将每一位数据再平均分成了 16 小段。对于 Bit_x 这一位数据，考虑到数据在刚刚发生变化和即将发生变化的这一时期，数据极有可能不稳定的（用深灰色标出的两段），在这两个时间段采集数据，很有可能得到错误的结果，因此判定这两段时间的电平无效，采集时直接忽略。而中间这一时间段（用浅灰色标出），数据本身是比较稳定的，一般都代表了正确的结果。也就是前面提到的中间测量方式，但是也不排除该段数据受强电磁干扰而出现错误的电平脉冲。因此对这一段电平，进行多次采样，并求高低电平发生的概率，6 次采集结果中，取出现次数多的电平作为采样结果。例如，采样 6 次的结果分别为 1/1/1/1/0/1/，则取电平结果为 1，若为 0/0/1/0/0/0,，则取电平结果为 0，当 6 次采样结果中 1 和 0 各占一半（各 3 次），则可判断当前通信线路环境非常恶劣，数据不具有可靠性，不进行处理。

7.2 UART 异步串口通信接收模块设计与实现

7.2.1 串口接收模块接口设计

基于以上原理，串口接收模块整体框图如下图 7-4 所示，其接口列表如下表所示。

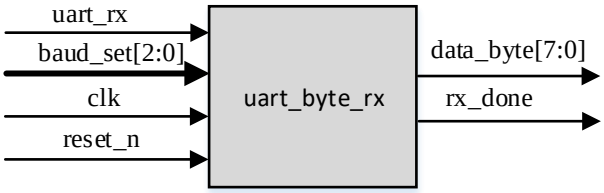


图 7-4 串口接收模块整体框图

表 7-1 模块接口列表

信号名称	I/O	功能描述
clk	I	模块系统时钟 50MHz
reset_n	I	模块异步复位信号
uart_rx	I	串行数据输入
baud_set	I	波特率选择信号
data_byte	O	并行数据输出

rx_done	O	接收结束信号

7.2.2 单 bit 异步信号同步设计

这里串口接收的信号 uart_rx 相对于 FPGA 内部信号来说是一个异步信号，如不进行处理直接将其输入使用，容易出现时序违例导致亚稳态。因此这里就需要先将信号同步到 FPGA 的时钟域内才可以供后续模块使用，常见的同步方法即使用两级触发器，也就是使用触发器对信号打两拍的方式进行与系统时钟进行同步，参考电路即如下图 7-5 所示。其中 uart_rx 为异步串口输入信号，uart_rx_sync2 为同步后的信号。

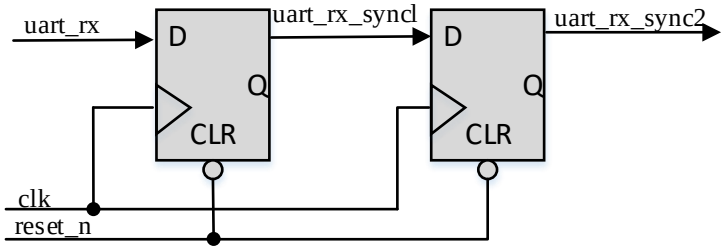


图 7-5 单 bit 信号同步示意图

将上述电路原理用 verilog 语言描述出来即为：

```
always@(posedge clk or posedge reset)
if(reset)begin
    uart_rx_sync1 <= 1'b0;
    uart_rx_sync2 <= 1'b0;
end
else begin
    uart_rx_sync1 <= uart_rx;
    uart_rx_sync2 <= uart_rx_sync1;
end
```

在一些高速设计中，也有使用三级触发器进行单 Bit 信号同步的设计，此方法只是为了提高平均故障时间（Mean Time Between Failure，MTBF），其电路为在两级触发器后再接一级触发器。

7.2.3 边沿检测设计

由串口接收时序图可知，数据接收的起始信号是串行数据由空闲的高电平变为低电平，即电平由高变低的下降沿，这就需要对下降沿进行检测，其原理就是利用寄存器在时钟信号的控制下，输入状态即为下一时刻输出状态这一特性进行比较判断，如下图 7-6 所示。

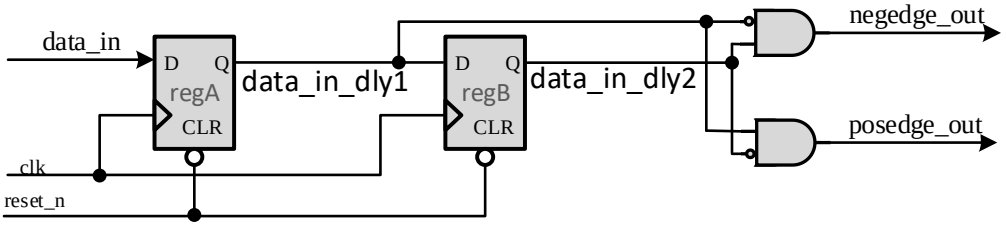


图 7-6 边缘检测原理

其检测过程，可以假设 data_in 从 0 变 1，也就是出现上升沿，具体时序图如下图 7-7。

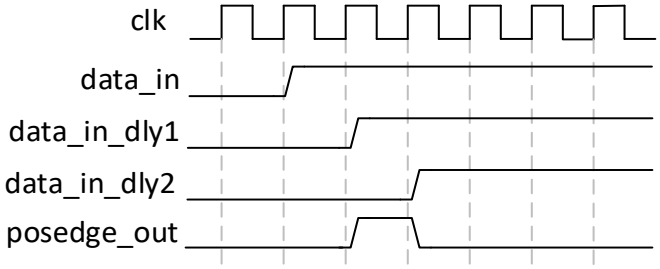


图 7-7 上升沿检测原理波形图

第一个寄存器 regA 的输出信号为 data_in_dly1，相对 data_in 延迟一个时钟周期；

第二个寄存器 regB 的输出信号为 data_in_dly2，相对 data_in_dly1 延迟一个时钟周期；对两个寄存器输出进行相关组合逻辑运算（data_in_dly1 & !data_in_dly2）则可检测出上升沿；

当 data_in 从 1 变为 0，也就是下降沿，具体时序图如下图 7-8。

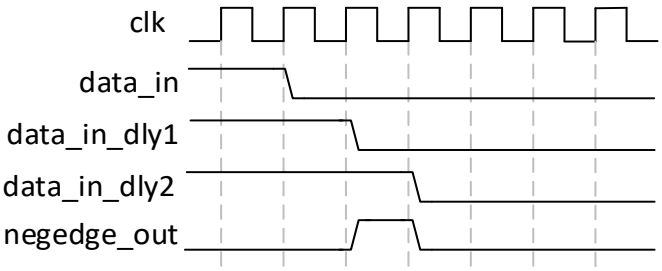


图 7-8 下降沿检测波形图

同理，对两个寄存器输出进行相关组合逻辑运算（!data_in_dly1 & data_in_dly2）则可检测出下降沿。通过上面的方法就实现了当 data_in 有上升沿时信号 pegedge_out 就会产生一个时钟周期的高电平，当有下降沿时信号 negedge_out 会产生一个时钟周期的高电平，没有上升沿或者下降沿变化时 pegedge_out 以及 negedge_out 保持低电平状态。上面使用的“!”是逻辑非运算，

对 4'b0110 逻辑非运算后是 4'b0000；而“~”是按位取反，对 4'b0110 按位取反后是 4'b1001，对于单 bit 数据使用“!”和“~”后的结果是一样的。

串口接收模块需要检测下降沿来识别串口接收信号的起始位，对同步后的串行输入数据检测下降沿，具体代码如下。

```
always@(posedge clk or posedge reset)
if(reset)begin
    uart_rx_reg1 <= 1'b0;
    uart_rx_reg2 <= 1'b0;
end
else begin
    uart_rx_reg1 <= uart_rx_sync2;
    uart_rx_reg2 <= uart_rx_reg1;
end
assign uart_rx_nedge = !uart_rx_reg1 & uart_rx_reg2;
```

7.2.4 采样时钟生成模块设计

串口接收模块主要构成之一即为波特率时钟生成模块。这里根据本章原理部分提到的过采样方式，即实际的采样频率是波特率的 16 倍，得出计数值与波特率之间的关系如表 7-2 采样时钟计算所示，其中系统时钟周期为 System_clk_period，这里为 20ns。

表 7-2 采样时钟计算

波特率	波特率周期	采样时钟分频计数值	System_clk_period = 20 计数值
9600	104167ns	104167/ System_clk_period/16	325-1
19200	52083ns	52083/ System_clk_period/16	163-1
38400	26041ns	26041/ System_clk_period/16	81-1
57600	17361ns	17361/ System_clk_period/16	54-1
115200	8680ns	8680/ System_clk_period/16	27-1

这里依旧使用一个选择器，来实现不同波特率与采样时钟分频计数值之间的对应关系。设计代码如下所示。

```
always@(posedge clk or posedge reset)
if(reset)
    bps_DR <= 16'd324;
else begin
    case(baud_set)
        0:bps_DR <= 16'd324;
        1:bps_DR <= 16'd162;
        2:bps_DR <= 16'd80;
        3:bps_DR <= 16'd53;
        4:bps_DR <= 16'd26;
        default:bps_DR <= 16'd324;
    endcase
```

```
end
```

现在产生采样时钟，即波特率时钟的 16 倍。

```
//counter
always@(posedge clk or posedge reset)
if(reset)
    div_cnt <= 16'd0;
else if(uart_state)begin
    if(div_cnt == bps_DR)
        div_cnt <= 16'd0;
    else
        div_cnt <= div_cnt + 1'b1;
end
else
    div_cnt <= 16'd0;

// bps_clk gen
always@(posedge clk or posedge reset)
if(reset)
    bps_clk <= 1'b0;
else if(div_cnt == 16'd1)
    bps_clk <= 1'b1;
else
    bps_clk <= 1'b0;
```

采样时钟计数器，计数器清零条件之一 bps_cnt == 8'd159 代表一个字节接收完毕。 (bps_cnt == 8'd12 && (START_BIT > 2))是实现起始位检测是否出错，在后面会对此进行详细解释。

```
//bps counter
always@(posedge clk or posedge reset)
if(reset)
    bps_cnt <= 8'd0;
else if(bps_cnt == 8'd159 | (bps_cnt == 8'd12 && (START_BIT > 2)))
    bps_cnt <= 8'd0;
else if(bps_clk)
    bps_cnt <= bps_cnt + 1'b1;
else
    bps_cnt <= bps_cnt;

always@(posedge clk or posedge reset)
if(reset)
    rx_done <= 1'b0;
else if(bps_cnt == 8'd159)
    rx_done <= 1'b1;
else
    rx_done <= 1'b0;
```

7.2.5 采样数据接收模块设计

以图 7-3 起始位为例，位于中间的采样时间段对应的 bps_cnt 值分别为 6、7、8、9、10、11，在这些时刻直接累加本位数据。然后，后一位数据的采样时间段的第一个 bps_cnt 值为前一位数据采样时间段的第一个 bps_cnt 值加 16。所以，起始位后面紧跟的第一个数据位 Bit0 的采样时间段的 bps_cnt 值分别是 22、23、24、25、26、27，同样，在这些时刻，直接累加本位数据。以此类推，可以得到其他位的采样时间段对应的 bps_cnt 值。

现解释为何在上面清零条件之一为(bps_cnt == 8'd12 && (START_BIT > 2))，理想情况下（真正的起始位）也就是当 bps_cnt 计数值为 12 时，START_BIT 的计算值应该为 0。而在实际中，有可能会出现一个干扰信号，而并非真正的起始信号，也导致下降沿的出现。此时，如果不加判断，直接视之为起始信号，进入接收状态，那么必然会接收到错误数据，也可能导致错过正确数据的接收。为了增加抗干扰能力，这里采样了这样的一种思路：当数据线上出现了下降沿时，先假设它是起始信号，然后对其进行中间段采样。如果这 6 次采样值累加结果大于 2，即 6 次采样中有至少一半的状态为高电平时，那么这显然不符合真正起始信号的持续低电平要求，此时就把刚才到来的下降沿视为干扰信号，而不作为起始信号。

```
always@(posedge clk or posedge reset)
if(reset)begin
    START_BIT <= 3'd0;
    data_byte_pre[0] <= 3'd0;
    data_byte_pre[1] <= 3'd0;
    data_byte_pre[2] <= 3'd0;
    data_byte_pre[3] <= 3'd0;
    data_byte_pre[4] <= 3'd0;
    data_byte_pre[5] <= 3'd0;
    data_byte_pre[6] <= 3'd0;
    data_byte_pre[7] <= 3'd0;
    STOP_BIT <= 3'd0;
end
else if(bps_clk)begin
    case(bps_cnt)
        0:begin
            START_BIT <= 3'd0;
            data_byte_pre[0] <= 3'd0;
            data_byte_pre[1] <= 3'd0;
            data_byte_pre[2] <= 3'd0;
            data_byte_pre[3] <= 3'd0;
```

```
            data_byte_pre[4] <= 3'd0;
            data_byte_pre[5] <= 3'd0;
            data_byte_pre[6] <= 3'd0;
            data_byte_pre[7] <= 3'd0;
            STOP_BIT <= 3'd0;
        end
        6 ,7 ,8 ,9 ,10,11:START_BIT <= START_BIT + uart_rx_sync2;
        22,23,24,25,26,27:data_byte_pre[0]<=data_byte_pre[0]+uart_rx_sync2;
        38,39,40,41,42,43:data_byte_pre[1] <= data_byte_pre[1] + uart_rx_sync2;
        54,55,56,57,58,59:data_byte_pre[2] <= data_byte_pre[2] + uart_rx_sync2;
        70,71,72,73,74,75:data_byte_pre[3] <= data_byte_pre[3] + uart_rx_sync2;
        86,87,88,89,90,91:data_byte_pre[4] <= data_byte_pre[4] + uart_rx_sync2;
        102,103,104,105,106,107:data_byte_pre[5]<=data_byte_pre[5]+uart_rx_sync2;
        118,119,120,121,122,123:data_byte_pre[6]<=data_byte_pre[6]+uart_rx_sync2;
        134,135,136,137,138,139:data_byte_pre[7]<=data_byte_pre[7]+uart_rx_sync2;
        150,151,152,153,154,155:STOP_BIT <= STOP_BIT + uart_rx_sync2;
    default:
        begin
            START_BIT <= START_BIT;
            data_byte_pre[0] <= data_byte_pre[0];
            data_byte_pre[1] <= data_byte_pre[1];
            data_byte_pre[2] <= data_byte_pre[2];
            data_byte_pre[3] <= data_byte_pre[3];
            data_byte_pre[4] <= data_byte_pre[4];
            data_byte_pre[5] <= data_byte_pre[5];
            data_byte_pre[6] <= data_byte_pre[6];
            data_byte_pre[7] <= data_byte_pre[7];
            STOP_BIT <= STOP_BIT;
        end
    endcase
end
```

7.2.6 数据状态判定模块

在原理部分介绍过，对一位数据需进行 6 次采样，然后取出现次数较多的数据作为采样结果，也就是说，6 次采样中出现次数多于 3 次的数据才能作为最终的有效数据。

对此，可以用接收到数据 data_byte_pre [n]结合数值比较器来判断，也可以直接令其等于当前位的最高位数据。以下面例子说明：当 data_byte_pre [n]分别为二进制的 011B/010B/100B/101B 时，这几个数据十进制格式分别为 3d/2d/4d/5d，可以发现大于等 4d 的为 100B/101B。当最高位是 1 即此时的数据累加值大于等于 4d，可以说明数据真实值为 1；当最高位是 0 即此时的数据累加值小于等于 3d，可以说明数据真实值为 0，因此只需判断最高位即可。

```
always@(posedge clk or posedge reset)
if(reset)
    data_byte <= 8'd0;
else if(bps_cnt == 8'd159)begin
    data_byte[0] <= data_byte_pre[0][2];
    data_byte[1] <= data_byte_pre[1][2];
    data_byte[2] <= data_byte_pre[2][2];
    data_byte[3] <= data_byte_pre[3][2];
    data_byte[4] <= data_byte_pre[4][2];
    data_byte[5] <= data_byte_pre[5][2];
    data_byte[6] <= data_byte_pre[6][2];
    data_byte[7] <= data_byte_pre[7][2];
end
```

7.3 仿真测试

完成设计之后，对其进行功能仿真。在下面的 testbench 文件中，这里产生数据的激励输入使用上一章的发送数据模块的输出来实现，因此激励文件只需在上一章的激励文件中，修改端口信息、例化本模块以及将发送模块输出的 uart_tx 连接到接收模块上的 uart_rx 即可。修改后的部分激励文件如下:

```
wire uart_tx_rx;

uart_byte_tx uart_byte_tx(
    .clk(clk),
    .reset_n(reset_n),

    .data_byte(data_byte_tx),
    .send_en(send_en),
    .baud_set(baud_set),

    .uart_tx(uart_tx_rx),
    .tx_done(tx_done),
    .uart_state(uart_state)
);

uart_byte_rx uart_byte_rx(
    .clk(clk),
    .reset_n(reset_n),

    .baud_set(baud_set),
    .uart_rx(uart_tx_rx),

    .data_byte(data_byte_rx),
    .rx_done(rx_done)
);
```

设计好仿真 testbench 文件后，点击进入 Modelsim 软件，新建工程用于功能仿真，可以看到如下图 7-9 所示的波形文件。串口发送模块 uart_tx 分别发送了 0xaa，0x55，串口接收模块 uart_rx 接收到串行数据并转换为并行输出，输出数据为 0xaa，0x55，与串口发送模块发送的数据完全一致。

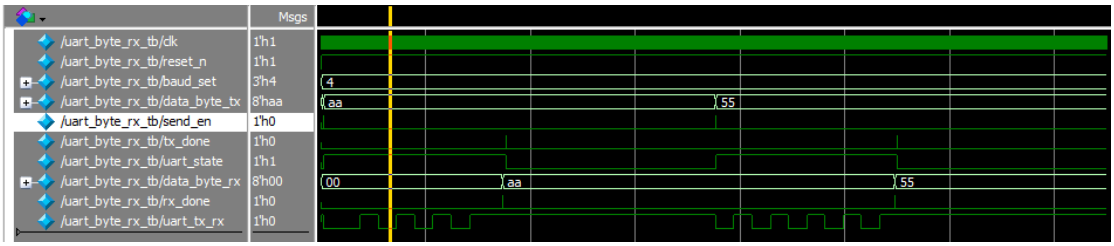


图 7-9 单 byte 数据接收仿真波形图

7.4 串口接收测试模块

新建一个 uart_rx_test.v，并将其设置为顶层设计。将 uart_byte_rx 和 uart_byte_tx 例化进该模块，每次串口接收到一个字节之后，产生 rx_done 信号，此时将 rx_done 信号赋值给 send_en，并将串口接收模块接收到的数据再通过串口发送模块发送出去，串口接收测试模块代码如下所示：

```
module uart_rx_test(
    input  clk,
    input  reset_n,
    input  uart_rx,
    output uart_tx
);
    reg [7:0] data_byte_tx;
    wire [7:0] data_byte_rx;
    reg send_en;
    wire tx_done;
    wire rx_done;
    always@(posedge clk or negedge reset_n)
    if(!reset_n) begin
        data_byte_tx <= 8'd0;
        send_en <= 1'd0;
    end
    else if(rx_done) begin
        data_byte_tx <= data_byte_rx;
        send_en <= rx_done;
    end
    else begin
        data_byte_tx <= data_byte_tx;
        send_en <= 1'd0;
    end
```

```
    uart_byte_tx uart_byte_tx(
        .clk(clk),
        .reset_n(reset_n),

        .data_byte(data_byte_tx),
        .send_en(send_en),
        .baud_set(3'd0),

        .uart_tx(uart_tx),
        .tx_done(tx_done),
        .uart_state( )
    );
    uart_byte_rx uart_byte_rx(
        .clk(clk),
        .reset_n(reset_n),

        .baud_set(3'd0),
        .uart_rx(uart_rx),

        .data_byte(data_byte_rx),
        .rx_done(rx_done)
    );
endmodule
```

7.5 板级验证

本次实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的数据流文件下载至开发板。
2. 打开电脑上的串口调试助手，发送一个字节的数据，然后观察串口助手能否将发送的数据再接收回来。

系统所需硬件：

1. 高云开发板。
2. 高云下载器及下载线。
3. 12V 的供电电源。
4. Type-C 线一根。
5. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

7.5.1 添加 I/O 约束

在 Gowin 软件右边的 Process 界面点击 FloorPlanner，新建一个.cst 文件存放 I/O 约束，如下图 7-10 所示。

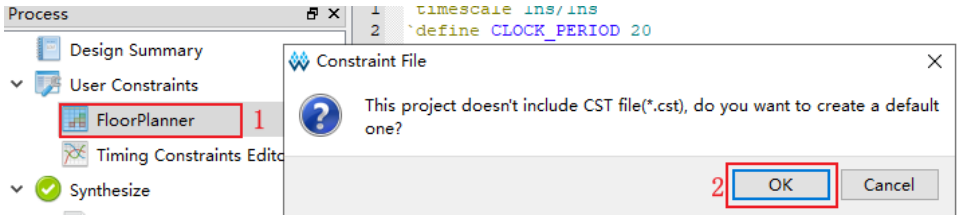


图 7-10 新建 I/O 约束

然后进行引脚约束，如下图 7-11 所示。

Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
clk	input		T9	4	False	LVCMOS33
reset_n	input		N9	5	False	LVCMOS33
uart_rx	input		U8	5	False	LVCMOS33
uart_tx	output		V8	5	False	LVCMOS33

图 7-11 引脚约束

7.5.2 布局布线

点击 Process 中的 Place & Route 进行布局布线，布局布线成功之后，会生成 bit 数据流，Console 提示如下图 7-12。

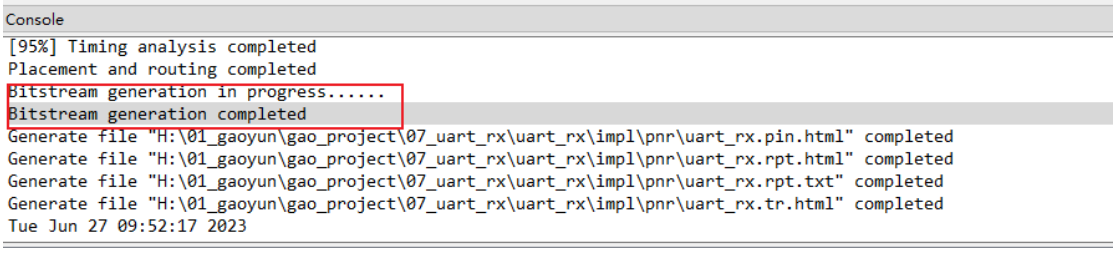


图 7-12 数据流生成成功示意图

7.5.3 硬件连接

将下载器、电源、串口线依次连接至开发板，整体的硬件连接如下图 7-13 所示。

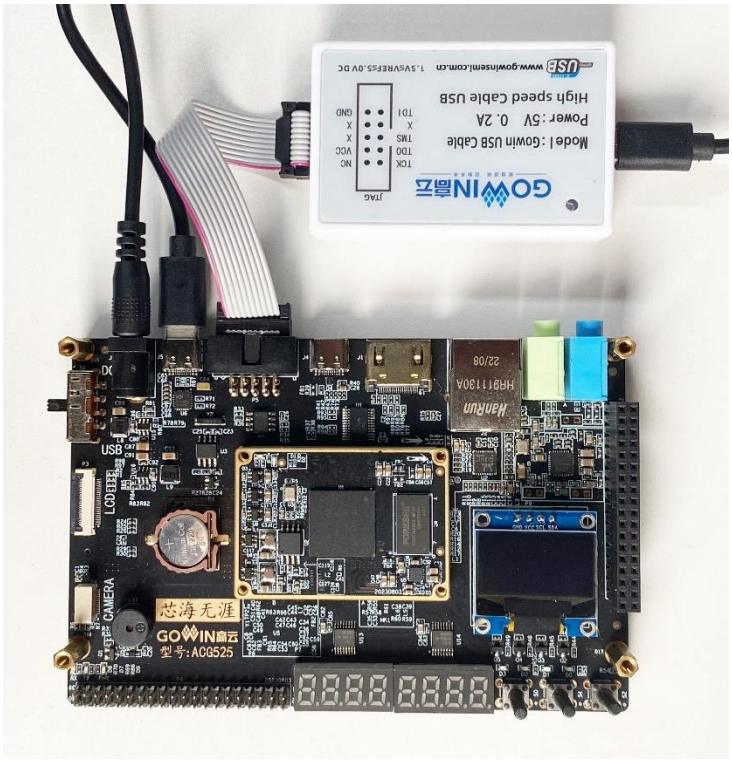


图 7-13 硬件连接图

7.5.4 配置串口调试助手

在 PC 机上打开一个串口软件，这里使用串口猎人，串口软件上设置开发板对应的串口号，波特率 9600，数据位 8bit，停止位 1bit，然后点击启动串行端口，可以在串口软件下面打印信息窗口看到具体连接情况，如下图 7-14 所示。

图 7-14 串口猎人工具的串口接收设置

7.5.5 下载数据流

点击 Program Device，进入下载界面，然后弹出 Cable Setting 界面，点击 Save，然后点击  下载文件，下载完成之后，如下图 7-15 所示。

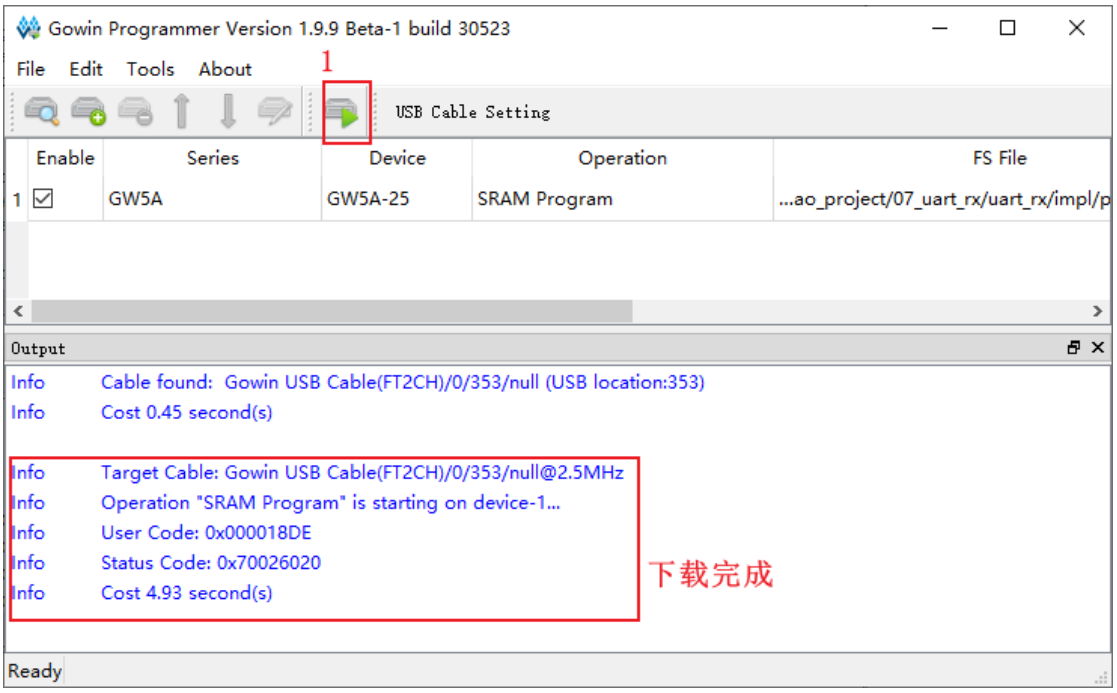


图 7-15 下载成功示意图

Program 之后，在发码区输入 AA，然后点击发送，再收码区可以接收到我们发送的的 AA，如下所示，由此可以看出串口单字节接收模块工作正常。

图 7-16 串口发送和接收数据一致

7.6 思考与总结

本章学习了串口接收的相关原理，在设计过程中针对工业现场的强电磁干扰等问题，提出了一种基于权重的改进型数据接收方式。并在板级测试中结合串口发送模块，实现了串口回环功能的测试，以此来验证串口接收模块工作正常，需要注意的是，本章实验的测试仅支持单字节的串口回环实验。如果一次发送多个字节的数据将会导致串口发送模块还未发送完，串口接收模块已经接收到了新的数据，导致串口助手最终接收的数据和发送的数据不一致，出现丢数据的情况。

8 串口控制 LED 灯

工程源码	|----02_设计实例 |----ch8_uart_rx_ctrl_led
相关视频课程	
说明	

章节导读

在 FPGA 设计中，我们常常会应用到串口，通过串口完成一些数据的收发。根据发送数据的类型不同，串口可以用于传图、控制、读写访问等场景。本节将基于前面章节中设计完成的串口接收模块，带领大家实现串口控制 LED 灯的设计。

设计允许用户通过串口，指定 LED 的亮灭模式以及每个状态变化的时间值。亮灭模式由 8 个亮灭状态构成，LED 会以这 8 个亮灭状态为循环，不断重复亮灭模式，从而达到串口控制 LED 亮灭的目的。

8.1 设计分析

从设计目的中我们可以知道，设计需要通过串口控制 LED 的亮灭和状态变化时间。这两者属于不同的功能，对应两个不同的控制字，为了方便称呼，我们将其分别称为状态控制字和周期控制字。

设计中 LED 以每 8 个状态为周期，不断循环变化，因此我们可以将状态控制字设定为 8 位，其中的每一位都对应 LED8 个状态中的其中一个状态。

周期控制字控制 LED 的变化时间，如果 LED 的变化时间过小，变化速度过快，人眼就无法观察到其变化过程。因此我们可以用一个稍大的，32 位数据来代表周期控制字。

因此，本次设计中，串口要控制 LED 实现所需功能，就需要发送 5 字节的控制字。例如，如果此时我们想控制 LED 按照亮灭亮灭亮灭亮灭的规律，每 5ms 变化一次，如图 8-1 所示：

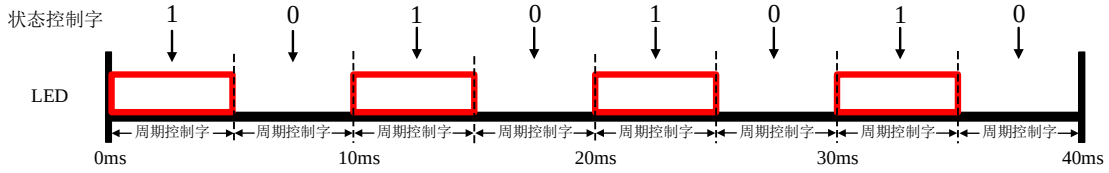


图 8-1 LED 亮灭控制（5ms）

那么我们只需要令状态控制字的值为 0xAA(8'b10101010)，然后通过周期控制字产生 5ms 的周期标志信号。每产生一次标志信号，就取状态控制字的一位，根据对应位的值控制指定 IO 输出高/低电平，驱动 LED 亮灭。如此重复，实现 LED 每 5ms 变化一次的亮灭亮灭亮灭亮灭循环。

同理，要想控制 LED 按照亮亮亮亮灭亮灭亮的规律，每 1s 变化一次，如图 8-2 所示：

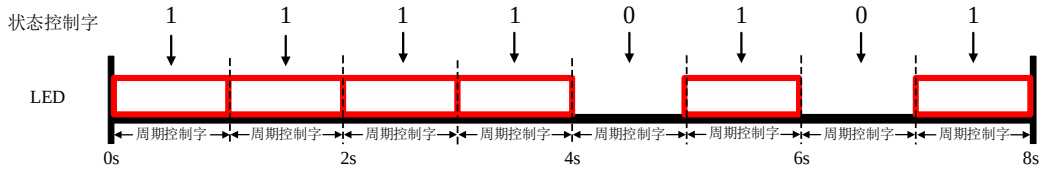


图 8-2 LED 亮灭控制（1S）

那么我们只需令状态控制字的值为 0xF5，通过周期控制字产生 1s 的周期标志信号即可。

8.2 定义串口传输协议帧

在确定了控制字数据的大小为 5 字节后， 为了确保接收到的数据是有效数据，同时也为了方便定位接收数据中各个字节的功能，我们可以先定义一个串口传输协议帧，如图 8-3 所示：

55	A5	time_set [31:24]	time_set [23:16]	time_set [15:8]	time_set [7:0]	ctrl [7:0]	F0

图 8-3 自定义串口传输协议帧

该协议帧是我们自定义的，由帧头（0x55 0xA5）+5 字节数据+帧尾（0xF0）组成。用户在主机上依据该传输协议帧下发数据，FPGA 在接收到数据后基于该协议对数据进行判断和解包。只有当串口接收到的连续多字节数据满足该协议时，才被认为是有效数据。随后将其中的 time_set 和 ctrl 分别作为周期控制字和控制控制字用于控制 LED。

通过这种方式，我们就能有效避免在传输过程中因为突发事故导致控制字多传、漏传乃至错传时，仍被作为有效指令生效的情况。

8.3 串口控制 LED 原理及思路

确定了设计传输的串口协议帧后，接下来就可以正式开始本次设计工作了。

对于设计来说，要通过串口控制 LED 的亮灭模式以及状态变化时间值，首先就需要有能够接收串口数据的能力。在前面的课程中，我们已经详细介绍并实现了串口接收的逻辑设计，本次设计中我们可以直接使用该设计。

串口接收模块每次只能传输单个字节的数据，而根据我们定义的串口协议帧，一个有效指令最少包含 8 个字节。因此需要一个串口指令转换模块，对串口接收模块输出的单字节数据进行 8 字节缓存，并基于缓存数据判断当前指令是否正确，提取出正确指令中状态控制字和周期控制字并输出。

输出的控制字最终用来控制 LED 的亮灭状态和周期。而要想实现该功能，我们可以定义一个计数器和一个位宽为 3 的标志信号。将周期控制字作为计数器的最大值，当计数器达到最大计数值时，让标志信号加1，并且计数器重新开始计数。通过这种方式，我们就能够通过周期控制字控制标志信号的变化频率。由于标志信号的位宽为 3，随着计数值的累加，标志信号会以固定的频率在 0~7 之间循环变化。而我们只需要在标志信号在不同值时，根据状态控制字对应位的值，驱动指定 IO 输出高/低电平，就能实现 LED 以固定频率，按照指定状态变化。

基于以上分析，要实现串口控制 LED 的亮灭及其频率，我们可以分为三部分。即：

1. 接收来自电脑的串行数据并将其转化为多个单字节数据输出。
2. 对单字节数据进行 8 字节缓存，判断数据是否为有效帧，提取出正确的控制字输出。
3. 根据控制字通过计数器和状态信号控制 LED 以一定的频率，8 个状态为周期，循环变化。

因此，本次设计的系统框图如下图 8-4 所示。

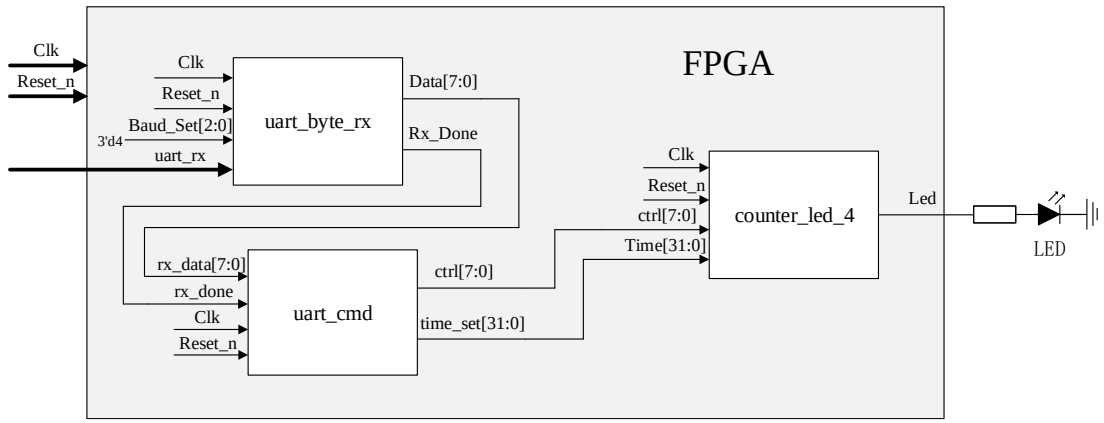


图 8-4 设计结构框图

其中 uart_byte_rx 为前面章节设计好的单字节串口接收模块；uart_cmd 为串口指令转换模块，该模块对单字节数据进行缓存，以得到连续的 8 个字节数据内容，然后每新收到一个字节数据，就对这连续 8 个字节内容的值进行帧结构检查，如果数据内容符合一个指令帧结构，则提取出周期控制字（time_set[31:0]）和状态控制字(ctrl[7:0])；counter_led_4 为计数驱动模块，根据输入的控制字，驱动 LED 灯以一定的频率和模式亮灭。

基于该设计结构，接下来就可以开始本次的模块设计了。

8.3.1 串口命令转换模块设计（uart_cmd）

串口接收模块每次会传输一字节数据，根据我们自定义的串口传输协议帧，要判断数据是否是用于控制 LED，需要最少 8 个字节。为此，我们需要对数据进行缓存，代码如下：

```
input Clk;
input [7:0]rx_data;

reg [7:0] data_str [7:0];
always@(posedge Clk)
if(rx_done)begin
    data_str[7] <= #1 rx_data;
    data_str[6] <= #1 data_str[7];
    data_str[5] <= #1 data_str[6];
    data_str[4] <= #1 data_str[5];
    data_str[3] <= #1 data_str[4];
    data_str[2] <= #1 data_str[3];
    data_str[1] <= #1 data_str[2];
    data_str[0] <= #1 data_str[1];
end
```

这里我们采用的缓存方式为移位缓存，该方式最大的好处就是这样缓存得到的数据是连续、变化且与串口传输协议帧长度一致的。通过这种方式，我们只需要在接收到新的字节数据时，对 data_str 内的 8 字节数据进行一次帧结构检测即可，而不用检测所有已接收到的数据。

一旦 data_str 内的数据满足我们定义的串口传输协议帧，就从中取出对应控制字输出。代码如下：

```
input Reset_n;
output reg[7:0]ctrl;
output reg[31:0]time_set;

reg r_rx_done;
always@(posedge Clk)
```

```
    r_rx_done <= rx_done;

always@(posedge Clk or negedge Reset_n)
if(!Reset_n) begin
    ctrl <= #1 0;
    time_set <= #1 0;
end else if(r_rx_done)begin
    if((data_str[0]==8'h55)&&(data_str[1]==8'hA5)&&(data_str[7]==8'hF0))begin
        time_set[31:24] <= #1 data_str[2];
        time_set[23:16] <= #1 data_str[3];
        time_set[15:8] <= #1 data_str[4];
        time_set[7:0] <= #1 data_str[5];
        ctrl <= #1 data_str[6];
    end
end
```

这里由于最高位的缓存数据与输入数据间存在一个时钟周期的延迟，因此需要对 rx_done 信号打一拍后再作为条件去判断缓存中的数据。

当r_rx_done信号有效时，代表模块接收了新一字节数据，此时便对data_str内缓存的数据进行帧结构判断。当数据符合帧结构时，便会被认为是有效指令，会从对应位中提取并组合成周期控制字和状态控制字，随后输出给计数驱动模块。

通过对接收数据进行 8 字节缓存以及帧结构判断，设计对用户下发的指令有着很强的容错性。即使用户在下发指令的过程中多发、漏发、错发了部分数据，设计也能根据帧结构对接收到的数据进行分析判断，并从中找到正确的指令。

8.3.2 计数驱动模块设计（counter_led_4）

计数驱动模块需要根据周期控制字计数并产生标志信号，根据状态控制字控制指定 IO 产生高/低电平，驱动 LED 亮灭。

标志信号的产生十分简单，只需要定义一个计数器，当计数器的值达到计周期控制字时，让标志信号自加一次。代码如下：

```
input Clk;
input Reset_n;
input [31:0]Time;

reg [31:0]counter;

always@(posedge Clk or negedge Reset_n)
if(!Reset_n)
    counter <= #1 0;
```

```
else if(counter >= Time - 1)
    counter <= #1 0;
else
    counter <= #1 counter + 1'b1;

reg [2:0]counter2;
always@(posedge Clk or negedge Reset_n)
if(!Reset_n)
    counter2 <= #1 0;
else if(counter >= Time - 1)
    counter2 <= #1 counter2 + 1'b1;
```

这里作为标志信号的 counter2 位宽为 3，可以用于表示 0~7，当超过 7 后便会自动溢出清零。用于控制 LED 亮灭状态的控制字也刚好有 8 位，因此我们可以通过 case 语句，根据 counter2 的值驱动 LED 亮灭，代码如下：

```
always@(posedge Clk or negedge Reset_n)
if(!Reset_n)
    Led <= #1 0;
else case(counter2)
    0:Led <= #1 Ctrl[0];
    1:Led <= #1 Ctrl[1];
    2:Led <= #1 Ctrl[2];
    3:Led <= #1 Ctrl[3];
    4:Led <= #1 Ctrl[4];
    5:Led <= #1 Ctrl[5];
    6:Led <= #1 Ctrl[6];
    7:Led <= #1 Ctrl[7];
    default:Led <= #1 Led;
endcase
```

每当计数器达到计数值时，counter2 的值就会自加一次导致 LED 的驱动发生改变。当 counter2 的值超过 7 后便会溢出清零，重新变为 0 ，以此，LED 便能以一定的频率在输入的 8 个状态中不断循环。

8.3.3 创建顶层封装

完成了各个模块的设计之后，我们还需要一个顶层模块，将这些模块串联起来，才能达到预期的功能。本次设计只需要各个模块进行顶层例化以及端口连接，完整代码如下：

```
module uart_rx_ctrl_led(
    Clk,
    Reset_n,
    Led,
    uart_rx
);
```

```
    input Clk;
    input Reset_n;
    output Led;
    input uart_rx;

    wire [7:0]ctrl;
    wire [31:0]time_set;
    wire [7:0]rx_data;
    wire rx_done;

    parameter Baud_Set = 3'd4;

    counter_led_4 counter_led(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Ctrl(ctrl),
        .Time(time_set),
        .Led(Led)
    );

    uart_cmd uart_cmd(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .ctrl(ctrl),
        .time_set(time_set)
    );

    uart_byte_rx uart_byte_rx(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Baud_Set(Baud_Set),
        .uart_rx(uart_rx),
        .Data(rx_data),
        .Rx_Done(rx_done)
    );

endmodule
```

至此，我们便完成了串口控制 LED 灯亮灭的设计。为了验证设计是否能够达到我们预期的功能，接下来我们还需要对其进行仿真验证，观察波形时序是否正确。

8.3.4 激励创建及仿真测试

在激励文件中，我们需要模拟串口，为设计提供输入数据。为了测试模块能否正确地识别到控制字数据，仿真中需要输入了一些干扰数据。本次激励文件完整代码如下：

```
`timescale 1ns / 1ps
module uart_rx_ctrl_led_tb();

    reg Clk;
    reg Reset_n;
    wire Led;
    reg uart_rx;
    wire [31:0] delay_time;

    uart_rx_ctrl_led uart_rx_ctrl_led(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Led(Led),
        .uart_rx(uart_rx)
    );
    parameter Baud_Set = 3'd4;

    assign delay_time = (Baud_Set == 3'd0) ? 20'd104166:
                        (Baud_Set == 3'd1) ? 20'd52083:
                        (Baud_Set == 3'd2) ? 20'd26041:
                        (Baud_Set == 3'd3) ? 20'd17361:
                                             20'd8680;


    initial Clk = 1;
    always#10 Clk = ~Clk;

    initial begin
       Reset_n = 0;
       uart_rx = 1;
       #201;
       Reset_n = 1;
       #200;

       uart_tx_byte(8'h55);
       #(delay_time*10);
       uart_tx_byte(8'ha5);
       #(delay_time*10);
       uart_tx_byte(8'h55);
       #(delay_time*10);
```

```
        uart_tx_byte(8'ha5);
        #(delay_time*10);
        uart_tx_byte(8'h00);
        #(delay_time*10);
        uart_tx_byte(8'h00);
        #(delay_time*10);
        uart_tx_byte(8'hc3);
        #(delay_time*10);
        uart_tx_byte(8'h50);
        #(delay_time*10);
        uart_tx_byte(8'hAA);
        #(delay_time*10);
        uart_tx_byte(8'hf0);
        #(delay_time*10);


        uart_tx_byte(8'h55);
        #(delay_time*10);
        uart_tx_byte(8'ha5);
        #(delay_time*10);
        uart_tx_byte(8'h9a);
        #(delay_time*10);
        uart_tx_byte(8'h78);
        #(delay_time*10);
        uart_tx_byte(8'h56);
        #(delay_time*10);
        uart_tx_byte(8'h34);
        #(delay_time*10);
        uart_tx_byte(8'h12);
        #(delay_time*10);
        uart_tx_byte(8'hf1);
        #(delay_time*10);
        #15000000;
        $stop;
    end

    task uart_tx_byte;
        input [7:0]tx_data;
        begin
            uart_rx = 1;
            #20;
            uart_rx = 0;
            #delay_time;
            uart_rx = tx_data[0];
            #delay_time;
            uart_rx = tx_data[1];
            #delay_time;
```

```
            uart_rx = tx_data[2];
            #delay_time;
            uart_rx = tx_data[3];
            #delay_time;
            uart_rx = tx_data[4];
            #delay_time;
            uart_rx = tx_data[5];
            #delay_time;
            uart_rx = tx_data[6];
            #delay_time;
            uart_rx = tx_data[7];
            #delay_time;
            uart_rx = 1;
            #delay_time;
        end
    endtask

endmodule
```

考虑到在不同波特率下，模拟串口输出时位与位之间、字节与字节之间的间隔不同，设计中使用 assign 根据用户设定的 baud_set 的值来自动定义延迟值。激励中设置的波特率为 115200，因此串口传输每一位的时间为 1/115200*10^9=8680ns。为了减少代码的重复性，我们使用 task 实现了模拟串口输出的过程。通过调用 task，就能模拟串口完成对待测模块指令下发工作。

可以看到，激励文件中除了下发有效指令外，还下发了一些干扰数据。其中，有效指令为 0x55_a5_00_00_c3_50_aa_f0，即计数器每计数 50000，也就是 1ms 转换一次 LED 的驱动电平。LED 则是按照亮灭亮灭亮灭亮灭（10101010，0xAA）的顺序变化。

编写完激励文件后，接下来就可以开始仿真，观察波形时序是否正确。本次设计仿真波形如图 8-5 所示：

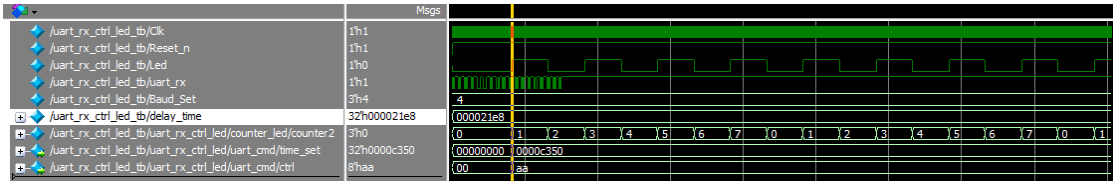


图 8-5 仿真波形

为了方便观察，这里添加了 uart_cmd 模块的 time_set 信号和 ctrl 信号以及 counter_led_4 模块的 counter2 信号。可以看到数据在被串口接收后 uart_cmd 模块成功识别出正确的指令，并获取出正确的数据：time_set 为 0xc350、ctrl 为 0xaa。驱动 LED 的电平在 counter2 信号变化时按照高低高低高低高低（亮灭亮

灭亮灭亮灭）的规律变化。counter2 在计数超过 7 后开始溢出清零，随后开始从 0 累加，LED 也仍然按照之前的规律不停亮灭。

可以看到，仿真波形与我们预期中的一致，说明设计正常，能够完整地实现预设功能。至此，我们还剩下最后一步，那就是通过板级验证，来验证设计在硬件上的实际效果。

8.4 板级验证

本节将对设计进行引脚分配，并基于分配好的工程所产生的 bit 数据流文件，在高云开发板进行烧录验证。通过电脑端的串口调试软件，下发指令实现对 LED 亮灭状态以及亮灭频率的控制。通过观察最终板级验证的结果与预期中是否一致，对设计进行验证。

8.4.1 管脚约束

虽然我们完成了设计的顶层封装，但是要想在高云硬件平台上进行测试，我们还需要对设计综合，确认无误后进行管脚分配并约束电平。本次设计管脚分配如表 8-1：

表 8-1 管脚约束表

Pin Name	Signial Name	Pin NO.	Pin Name	Signial Name	Pin NO.
CLK_G	CLK	T9	UART_RXD	uart_rx	U8
KEY1	Reset_n	B16	LED0	led	D14

配置完成后将对应引脚约束为 LVCMOS33，接下就可以生成比特流，在比特流生成完成后连接硬件准备烧录验证了。

8.4.2 系统所需硬件

1. 高云开发板 x1
2. 高云下载器 x1
3. 串口线 x1
4. 电源线 x1

8.4.3 硬件连接

连接开发板的下载线和电源线，使用串口线一端连接开发板上的 UART 接口，一端连接电脑的 USB 口，如下图 8-6 所示。

图 8-6 硬件连接

连接完成后，确保开发板电源拨码开关拨到 ON 侧，接下来将生成好的 bit 烧录到开发板中。

烧录完成后打开设备管理器，查询串口端口号，设备管理器中串口的名称如图 8-7：

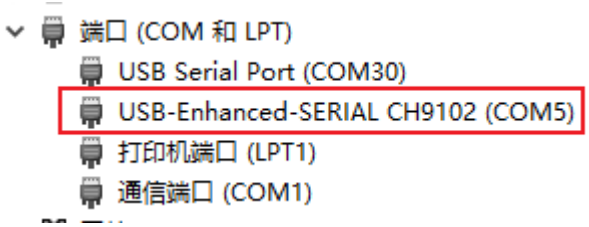


图 8-7 查询端口号

这里笔者电脑串口端口号为 COM5，用户在查询时，以自己电脑中的端口号为准。随后打开串口猎人，在软件中连接上刚刚查询到的端口，设置波特率为 115200。接着通过串口猎人下发指令，下发方式如图 8-8 所示：

图 8-8 串口猎人发码

可以看到其中有效指令为 55 A5 02 FA F0 80 AA F0，因此，周期控制字为 0x02FAF080（50_000_000），状态控制字为 0xAA（10101010）。即，LED 按照亮灭亮灭亮灭亮灭的规律，每 1s 变化一次，每 8 次变化为一组，不断循环变化。将指令和干扰数据一同下发到 FPGA 后，LED0 开始按照预期中亮灭。

随后，笔者又尝试下发了多次指令并特意多发、漏发、错发了一部分。为了使现象不受上一次指令的影响，每次下发指令前都对程序进行了复位。相关指令以及板级现象如下：

1. 下发指令 18 65 55 A5 02 FA F0 80 F0 F0 23 12，开发板上 LED0 亮 4s 后灭 4s，不断循环
2. 下发指令 55 A5 02 FA 80 A7 F0，开发板上 LED 无反应
3. 下发指令 55 A5 02 FA F0 80 F0 A0，开发板上 LED 无反应

操作 2 中，指令漏发，虽然检测到正确的帧头 55 A5，但是由于帧尾对应位为空，所以指令被判定为无效指令，LED 不会被驱动。与操作对应的板级现象吻合。

操作 3 中，指令错发，虽然检测到正确的帧头 55 A5，但是帧尾的值与我们

定义的值不一致，因此指令被判定为无效指令，LED 不会被驱动。与操作对应的板级现象吻合。

在四次指令下发测试中，LED 都很好地按照我们的预期被驱动。

至此，板级验证成功，设计能够成功地在高云开发板上实现预期功能，本次设计无误。

8.5 总结

本章带大家完成了串口控制 LED 灯的设计，设计使用串口下发指令，通过自定义的串口传输协议帧来确保指令的正确性并定位对应控制字。最终控制字被传给 LED 计数驱动模块，完成对 LED 亮灭状态以及频率的控制。

本章所设计的模块功能并不复杂，目的只是为读者提供一个使用串口实现控制功能的思路。串口的应用场景非常广泛，用户可以发散自己的思维，去思考并尝试更为复杂且有趣的设计。

9 多字节串口收发模块设计与验证

工程源码	|----02_设计实例 |----ch9_uart_loopback
相关视频课程	
说明	

章节导读

在前面章节中我们已经带大家完成了单字节串口收发模块的设计，但是这两个模块的应用场景相对较小。在大多数场景下，串口都是用于连续多字节的收发工作。因此，本章将基于已有的单字节串口收发模块，带大家完成多字节串口发送模块和多字节串口接收模块的设计，并通过仿真和板级测试对设计进行验证。

9.1 多字节串口发送原理与设计

要实现多字节串口发送，以常用的 8N1 配置为例(即八个数据位、无奇偶校验、一个停止位)，很多同学会有个误区，认为只要将数据填充进起始信号跟停止信号之间即可。例如我想用串口发送 16 位的数据 0x1234，根据以上思路，就应该模拟串口输出 0_0010_1100_0100_1000_1，如图 9-1 所示：

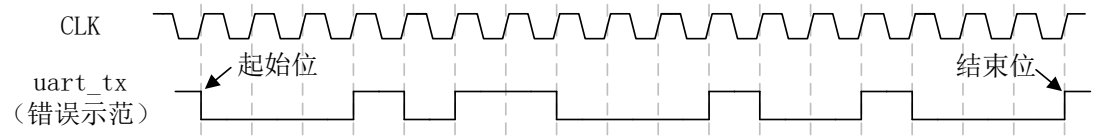


图 9-1 串口发送 0x1234（错误示范）

这些数据在被目的串口接收后，会按照 8N1 进行解析，提取出其中的数据位。一次解析需要接收 10 位数据，如果解析正确便会提取出中间的 8 位有效数据。要解析得到 16 位的 0x1234 就需要解析 2 次，也就是需要 20 位数据。而我们模拟串口时，仅发送了 18 位数据，所以无论最终解析出来的是什么，都不会是我们期望收到的 0x1234。

正确的传输应该是分为两次，第一次传输 0_0100_1000_1，第二次传输 0_0010_1100_1，如图 9-2 所示：

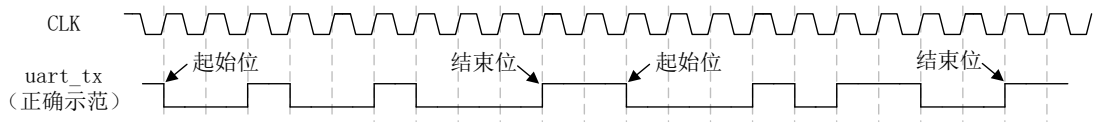


图 9-2 串口发送 0x1234

目的串口在接收到数据后，会按照 8N1 对数据解析，分两次得到 8'b0001_0010（0x12）和 8'b0011_0100（0x34）。将两个 8 位数据拼接后便能得到 0x1234。

因此，多字节的串口发送，可以看作连续多个单字节数据的串口发送过程。而在前面章节中，我们已经带大家学习并完成了单字节串口发送模块的设计。本章中我们只需要将待发送的数据拆分为多个单字节，然后调用单字节串口发送模块逐个发送，就能实现多字节数据的串口发送。

9.1.1 多字节串口发送原理

为了保证数据的连续性，我们可以使用单字节串口发送模块的单字节发送完成信号（byte_tx_done）作为条件信号。当该信号有效时，说明一字节数据发送完成，此时就需要输出下一字节数据给单字节串口发送模块。

一次多字节串口发送，所需发送的数据量是确定的，串口不可能一直发送下去。因此在每发送完一字节数据后，我们还需要判断发送是否完成。根据 byte_tx_done 信号计算当前已发送数据，再与接收到的数据位宽做对比。如果发送未完成，则继续下一字节的发送，一旦判断发送完成，产生串口发送完成信号，本次传输结束。

据此，我们可以设计一个状态机，用来产生单字节数据、判断单次字节发送是否完成、判断整个发送过程是否完成。因此，本次设计的串口发送模块的结构如图 9-3 所示：

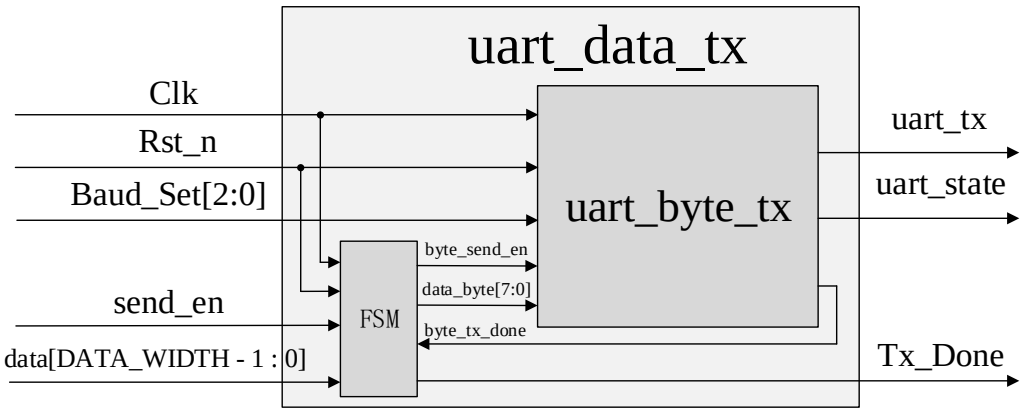


图 9-3 多字节串口发送模块

9.1.2 多字节串口发送模块设计

在确定了设计的整体功能及结构之后，接下来就是设计的代码实现了。首先是对单字节串口发送模块的例化，代码如下：

```
input Clk;
input Rst_n;
input [2:0]Baud_Set;
output uart_tx;
output uart_state;

reg [7:0] data_byte;
reg byte_send_en;
wire byte_tx_done;

uart_byte_tx uart_byte_tx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .data_byte(data_byte),
    .send_en(byte_send_en),
    .Baud_Set(Baud_Set),
    .uart_tx(uart_tx),
    .Tx_Done(byte_tx_done),
    .uart_state(uart_state)
);
```

为了使设计能够适配更多应用场景，我们可以使用 parameter 定义多字节串口发送模块的输入数据位宽。parameter用于在模块内部或外部定义常量，这种方式定义常量最大的好处就是在对模块实例化时，能够进行参数传递（重写）。

通过这种方式，我们可以使用 parameter 将多字节串口发送模块的输入数据位宽设定为 8，代码如下：

```
parameter DATA_WIDTH = 8;
```

这样，默认情况下模块只会实现单字节的串口发送。而如果我们想让模块一次发送多个字节，以 8 字节为例，我们只需要在顶层中通过 parameter 定义输入数据位宽为 64，参数就能够传递到子模块中，对常量值重写。

对于多字节串口发送模块而言，当输入的 send_en 有效时，表明传输开始，状态机产生一字节的发送数据以及发送使能信号给单字节串口发送模块。随后，状态机需要等待 byte_tx_done 信号，确认当前字节数据已经发送完毕。同时，状态机需要基于该信号计数，判断传输是否完成。如果未传输完成，便继续下一字节数据和使能信号的发送。如果传输完成，状态机便需要产生发送完成信号 tx_done，标志本次多字节发送结束。

综上所述，状态机需要至少四个状态，才能实现上述功能。四个状态分别用来等待发送使能、发起单字节数据发送、等待单字节数据发送完成、检查所有数据是否传输完成。对应的四种状态代码定义如下：

```
localparam S0 = 0;  //等待发送请求
```

```
localparam S1 = 1;  //发起单字节数据发送
localparam S2 = 2;  //等待单字节数据发送完成
localparam S3 = 3;  //检查所有数据是否发送完成
```

接下来实现四个状态的功能，首先是状态 S0，该状态需要等待发送请求信号 send_en，在 send_en 有效时，接收传入的数据，并使状态跳转到 S1。对应代码如下：

```
S0:
    begin
        data_byte <= 0;
        cnt <= 0;
        Tx_Done <= 0;
        if(send_en)begin
            state <= S1;
            data_r <= data;
        end
        else begin
            state <= S0;
            data_r <= data_r;
        end
    end
```

接下来是 S1 状态，该状态需要产生单字节发送使能信号以及单字节数据，并使状态跳转到 S2。考虑到在不同应用场景中，数据的字节序会有不同需求，我们可以通过使用 parameter 定义一个常量 MSB_FIRST 控制数据的大小端字节序。代码如下：

```
parameter MSB_FIRST = 1;
```

S1 状态代码如下：

```
S1:
    begin
        byte_send_en <= 1;
        if(MSB_FIRST == 1)begin
            data_byte <= data_r[DATA_WIDTH-1:DATA_WIDTH - 8];
            data_r <= data_r << 8;
        end
        else begin
            data_byte <= data_r[7:0];
            data_r <= data_r >> 8;
        end
        state <= S2;
    end
```

随后是 S2 状态，该状态需要等待单字节数据发送完成，也就是等待单字节串口发送模块的 byte_tx_done 信号。当该信号有效时，代表单字节数据发送完成，我们可以基于该信号计数当前已发送的数据量，并将状态跳转到 S3。代码

如下：

```
S2:
    begin
        byte_send_en <= 0;
        if(byte_tx_done)begin
            state <= S3;
            cnt <= cnt + 9'd8;
        end
        else
            state <= S2;
    end
```

最后便是 S3 状态，我们可以通过对比已发送数据量和输入位宽量来产生 Tx_Done 信号。当已发送数据量大于等于输入位宽时，说明数据全部发送完毕，此时可以产生 Tx_Done 信号并将状态跳转到 S0，等待下次传输开始。当已发送数据量小于输入位宽时，说明传输还未结束，此时将状态跳转到 S1，准备下一字节的数据传输。代码如下：

```
S3:
    if(cnt >= DATA_WIDTH)begin
        state <= S0;
        cnt <= 0;
        Tx_Done <= 1;
    end
    else begin
        state <= S1;
        Tx_Done <= 0;
    end
```

至此，我们便完成了多字节串口发送模块的设计。至于模块是否能够正常工作，还需要进一步验证，因此，接下来我们还需要创建一个仿真激励文件，对模块进行仿真验证。

9.1.3 激励创建及仿真测试

仿真激励的创建相对简单，只需在例化多字节串口发送模块时，设置好输入位宽以及大小端的值，并在激励中为设计产生输入数据以及发送使能信号即可。本次仿真激励文件完整代码如下：

```
module uart_data_tx_tb();

    parameter DATA_WIDTH = 32;
    parameter MSB_FIRST = 0;

    reg Clk;
    reg Rst_n;
```

```
reg [DATA_WIDTH - 1 : 0]data;
reg send_en;
wire uart_tx;
wire Tx_Done;
wire uart_state;

uart_data_tx
#(
    .DATA_WIDTH(DATA_WIDTH),
    .MSB_FIRST(MSB_FIRST)
)
uart_data_tx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .data(data),
    .send_en(send_en),
    .Baud_Set(3'd4),
    .uart_tx(uart_tx),
    .Tx_Done(Tx_Done),
    .uart_state(uart_state)
);
initial Clk = 1;
always #10 Clk = !Clk;
initial begin
    Rst_n = 0;
    data = 0;
    send_en = 0;
    #201;
    Rst_n = 1;
    #2000;
    data = 32'h01234567;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done);
    #1;
    data = 32'h12345678;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done);
    #1;
    data = 32'h23456789;
    send_en = 1;
```

```
        #20;
        send_en = 0;
        #20;
        @(posedge Tx_Done);
        #1;
        #2000;
        $stop;
    end
endmodule
```

激励文件中设置输入数据位宽为 32，输出为小端字节序，并输入了三个已知数据。因此，以第一次输入的数据为例，第一次多字节串口发送，输出的数据应该依次为 0x67、0x45、0x23、0x01，共 32 字节数据。后续两次发送结果同理。对比仿真波形与推论是否一致，我们便能知道设计是否正确。

本次仿真波形如图 9-4 所示：

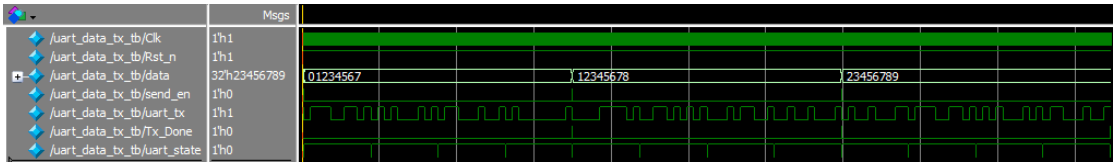


图 9-4 多字节串口发送模块仿真波形

可以看到三次数据都被成功接收，并且串口完成了三次多字节发送任务。为了方便查看串口具体发送的数据，这里又添加 uart_byte_tx 模块的 data_byte 和 Tx_Done 信号进行观察。完整波形时序如图 9-5 所示：

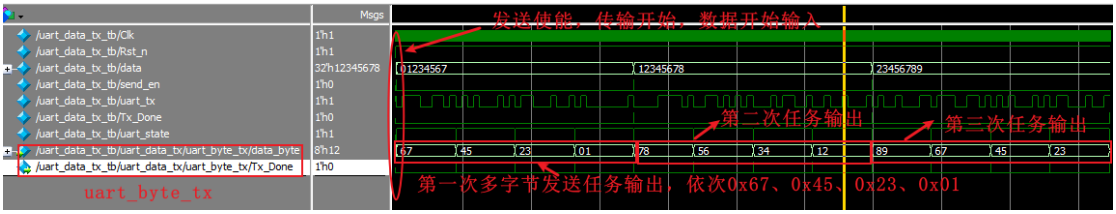


图 9-5 第一次多字节串口发送

可以看到，在 send_en 信号有效后，多字节串口发送模块开始工作，输出值与预期中第一次多字节串口发送的结果一致。后两次传输的结果与预期中同样一致，这里暂不列出详细时序，本次设计成功。

9.2 多字节串口接收原理与设计

串口接收模块在工作时，会接收连续的串行数据，并以字节为单位，对有效数据进行输出。而所谓的多字节串口接收，就是在串口接收模块在接收到数据后，不再以单字节数据输出，而是对数据进行拼接，组合成一个多字节数据

后输出。

结合前面我们设计的多字节串口发送模块，要实现多字节串口接收模块设计，我们可以例化单字节串口接收模块，随后设计一个状态机，对单字节串口接收模块输出的数据进行拼接。同时，基于单字节串口接收模块产生的接收完成信号，计数当前已经接收（拼接）的数据量。当拼接的数据达到指定位宽时，将数据输出。

9.2.1 多字节串口接收原理

当然，仅仅如此是不够的。如果串口在接收过程中，因为某些外部干扰，导致部分数据发生变化无法被识别，那么后续被成功识别的数据将会代替这些数据的位置，最终拼接成错误的数据。

例如，假设在某次串口接收时，串口本应该能正确接收 0x98、0x76、0x54、0x32。但是在传输的过程中由于外部干扰，0x76 的部分位数据发生了改变，没有被串口识别到，最终接收到的数据为 0x98、0x54、0x32。如果这些数据直接交给状态机进行拼接，最终输出的必然是错误数据。

为了防止这种情况，我们就需要对每次接收进行计时，一旦接收超时，便产生接收超时标志信号，同时通过状态机结束本次串口接收。经过以上分析，我们可以得出本次设计系统结构如图 9-6 所示：

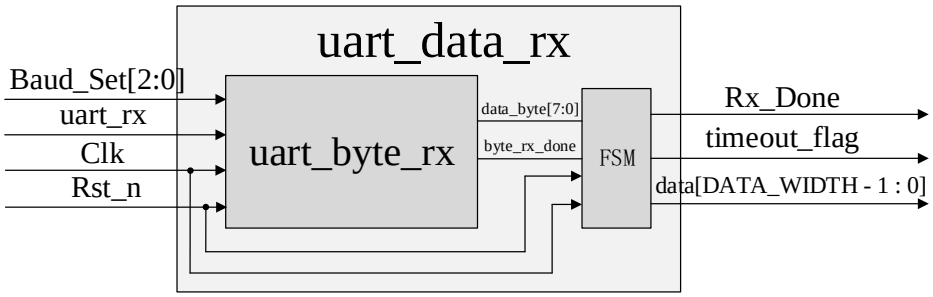


图 9-6 多字节串口接收模块系统框图

9.2.2 多字节串口接收模块设计

在确定了设计整体结构之后，接下来就是设计的代码实现。首先是对单字节串口接收模块的例化，代码如下：

```
input Clk;
input Rst_n;
input uart_rx;
wire [7:0] data_byte;
wire byte_rx_done;
```

```
uart_byte_rx uart_byte_rx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .baud_set(Baud_Set),
    .uart_rx(uart_rx),
    .data_byte(data_byte),
    .Rx_Done(byte_rx_done)
);
```

为了使设计能够适配更多应用场景，我们可以使用 parameter 定义多字节串口接收模块输出位宽为 16。这样我们在调用该模块时，随时都能通过顶层修改模块输出位宽。代码如下：

```
parameter DATA_WIDTH = 16;
```

接下来我们就该考虑如何产生接收超时信号了。对于串口接收来说，在不同应用场景下，会使用不同的波特率，每种波特率下，串口传输数据的频率也各不相同。那么我们该如何判断接收是否是超时的呢？

这里，我们参考了 modbus 对于串口传输超时的定义。在 modbus 中，规定报文间隔要大于 3.5 字符时间，也就是传输一旦超过 3.5 字符时间就视作是新的报文。基于此，本次设计中，我们认为，在进行串口接收数据时，一旦数据间超过 3.5 个字符时间，便是接收超时。

以常用的 115200 波特率为例，我们可以计算出，串口正常接收一字节数据所需的时间为 1/115200*10^9*(1+8+1)=86.805us。那么如果串口超过 86.805*3.5=300.8175us 时间内都未接收到数据，便可以认为是接收超时。

在本次设计中，我们使用的是 50MHz 时钟，即每个时钟周期为 20ns。串口接收一字节数据所花的时间，可以通过计数两个串口接收完成信号之间的时钟上升沿来确定。因此，我们就可以求出，当计数值大于 300.8175*10^3/20-1 = 15190 时，接收超时。一旦接收超时，我们就让它产生超时标志信号。

至于计数器该何时计数，我们可以让计数器在每个字节接收开始时工作，在字节接收完成时被清零。具体就是利用串口传输时的低电平起始信号（！uart_rx），以及单字节串口接收模块在接收完一个字节后产生的 byte_rx_done 信号。因此，判断并产生串口接收超时标志信号的设计代码如下：

```
parameter TIMEOUT = 15190;//115200 下为 15190,9600 下为 182292

reg [31:0]timeout_cnt;
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    timeout_flag <= 1'd0;
else if(timeout_cnt >= TIMEOUT)
```

```
    timeout_flag <= 1'd1;
else if(state == S0)
    timeout_flag <= 1'd0;

reg to_state;
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    to_state <= 0;
else if(!uart_rx)
    to_state <= 1;
else if(byte_rx_done)
    to_state <= 0;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    timeout_cnt <= 32'd0;
else if(to_state)begin
    if(byte_rx_done)
        timeout_cnt <= 32'd0;
    else if(timeout_cnt >= TIMEOUT)
        timeout_cnt <= TIMEOUT;
    else
        timeout_cnt <= timeout_cnt + 1'd1;
end
```

设计完接收超时信号，接下来就是状态机了。结合前面的原理分析和结构图我们可以知道，本次设计中状态机的工作流程如下：

1. 等待单字节接收模块的接收完成信号，缓存第一个字节数据，对缓存的数据进行计数
2. 开始超时计数，如果下一次传输完成信号到来前接收超时，产生 Rx_Done 信号，传输结束。如果未超时则对新一字节数据进行拼接，并对已缓存数据计数。
3. 对比已缓存的数据与输出数据位宽，检查所有数据是否接收完成。如果完成，产生 Rx_Done 信号，传输结束。如果未完成则继续步骤 2

因此，整个状态只需要三个状态就能实现功能，状态定义及初始化如下：

```
localparam S0 = 0;  //等待单字节接收完成信号
localparam S1 = 1;  //判断接收是否超时
localparam S2 = 2;  //检查所有数据是否接收完成
```

接下来首先是 S0 状态的实现，考虑到在不同应用场景下，可能会对数据有不同的字节序需求，我们可以通过使用 parameter 定义一个常量 MSB_FIRST 控制数据的大小端字节序。该状态代码如下：

```
S0:
    begin
        Rx_Done <= 0;
        data_r <= 0;
        if(DATA_WIDTH == 8)begin
            data <= data_byte;
            Rx_Done <= byte_rx_done;
        end
        else if(byte_rx_done)begin
            state <= S1;
            cnt <= cnt + 9'd8;
            if(MSB_FIRST == 1)
                data_r <= {data_r[DATA_WIDTH - 1 - 8 : 0]
                , data_byte};
            else
                data_r <= {data_byte
                , data_r[DATA_WIDTH - 1 : 8]};
        end
    end
```

随后是 S1 状态的实现，代码如下：

```
S1:
    if(timeout_flag)begin
        state <= 0;
        Rx_Done <= 1;
    end
    else if(byte_rx_done)begin
        state <= S2;
        cnt <= cnt + 9'd8;
        if(MSB_FIRST == 1)
            data_r <= {data_r[DATA_WIDTH - 1 - 8 : 0], data_byte};
        else
            data_r <= {data_byte, data_r[DATA_WIDTH - 1 : 8]};
    end
```

最后便是 S2 状态的实现，代码如下：

```
S2:
    if(cnt >= DATA_WIDTH)begin
        state <= S0;
        cnt <= 0;
        data <= data_r;
        Rx_Done <= 1;
    end
    else begin
        state <= S1;
        Rx_Done <= 0;
    end
```

至此，我们便完成了多字节串口接收模块的设计。至于模块是否能够正常

工作，还需要进一步验证，因此，接下来我们还需要创建一个仿真激励文件，对模块进行仿真验证。

9.2.3 激励创建及仿真测试

为了方便测试，在激励文件中，我们可以例化前面设计的多字节串口发送模块，用以产生测试多字节串口接收模块所需的串行输入信号。本次仿真激励文件完整代码如下：

```
module uart_data_rx_tb();

    parameter DATA_WIDTH = 32;
    parameter MSB_FIRST = 0;

    reg Clk;
    reg Rst_n;

    reg [DATA_WIDTH - 1 : 0]data;
    wire [DATA_WIDTH - 1 : 0]rx_data;
    reg send_en;
    wire uart_tx;
    wire uart_rx;
    wire Tx_Done;
    wire uart_state;
    wire Rx_Done;
    wire timeout_flag;

    assign uart_rx = uart_tx;

    //为多字节串口接收模块产生输入数据
    uart_data_tx
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_tx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .data(data),
        .send_en(send_en),
        .Baud_Set(3'd4),
        .uart_tx(uart_tx),
        .Tx_Done(Tx_Done),
        .uart_state(uart_state)
    );
```

```
    uart_data_rx
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_rx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .uart_rx(uart_rx),
        .data(rx_data),
        .Rx_Done(Rx_Done),
        .timeout_flag(timeout_flag),
        .Baud_Set(3'd4)
    );

    initial Clk = 1;
    always #10 Clk = !Clk;

    initial begin
        Rst_n = 0;
        data = 0;
        send_en = 0;
        #201;
        Rst_n = 1;
        #2000;
        data = 32'h12345678;
        send_en = 1;
        #20;
        send_en = 0;
        #20;
        @(posedge Tx_Done);
        #1;
        #1000000;
        data = 32'h87654321;
        send_en = 1;
        #20;
        send_en = 0;
        #20;
        @(posedge Tx_Done);
        #1;
        #1000000;
        data = 32'h24680135;
        send_en = 1;
        #20;
        send_en = 0;
        #20;
        @(posedge Tx_Done);
```

```
        #1;
        #400000;
        $stop;
    end

endmodule
```

激励文件中，波特率被设置为 115200，为了验证模块能否正确判断传输是否超时，将每两次数据输入之间的间隔设置为 1000.001us。

同时，激励文件中，DATA_WIDTH 被设置为 32，字节序被设置为小端字节序。由于是小端字节序，所以多字节串口发送模块会先发低位字节再发高位字节。以第一次数据发送为例，多字节串口发送模块会依次发送数据 0x78、0x56、0x34、0x12。由于多字节串口接收模块也是使用的小端字节序，这些字节数据在被多字节串口接收模块接收拼接时，先接收的字节将会被作为最高位。因此多字节串口接收模块输出的数据为 32'h12345678。

根据理论结果，观察仿真波形与推论是否一致，我们便能知道设计是否正确，本次设计仿真总体波形如图 9-7 所示。

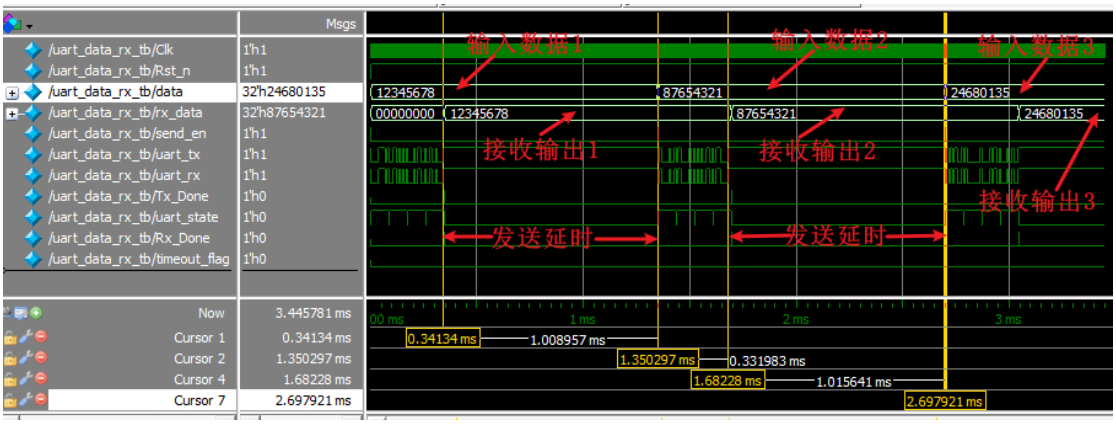


图 9-7 仿真整体波形

可以看到，即使是延时了 1000.001us，timeout_flag 信号也并未置高，且多字节串口接收模块的输出数据与我们预期中的一致。

接下来我们再来观察数据传输的中间过程，观察字节序是否正常工作。为了方便观察，在仿真中添加了 uart_byte_tx 的 data_byte（输出字节数据）信号、uart_byte_rx 的 data_byte（输入字节数据）信号，波形时序如图 9-8 所示：

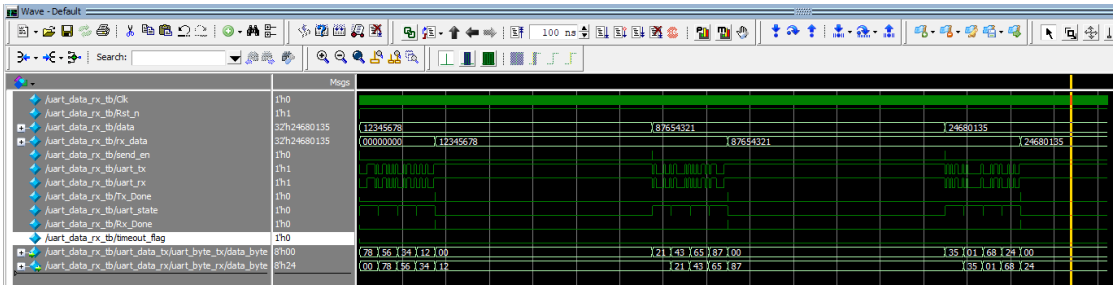


图 9-8 多字节串口接收

可以看到，数据波形与我们推导中一致，说明字节序功能也正常工作。至此，多字节串口接收模块设计完成。

9.3 创建测试用顶层封装

虽然我们已经通过仿真测试，验证了两个模块功能的可行性，但是考虑到仿真终究是理想条件，与实际的应用情况始终会有出入。因此，设计还需要进行板级测试，验证其功能的完整性。

为了方便测试，这里我们新建一个顶层，例化多字节串口收发模块，将接收模块的输出端与发送模块的输入端相连。这样，接收模块接收到的数据就会通过发送模块发送出去，再通过电脑端的串口调试软件，就能实现一个简易的串口回环。通过判断回环的数据是否一致，就能知道模块功能是否实现。

顶层代码如下：

```
module uart_loopback(
    Clk,
    Rst_n,
    uart_rx,

    led,
    uart_tx
 );

    parameter DATA_WIDTH = 32;
    parameter MSB_FIRST = 0;

    input Clk;
    input Rst_n;
    input uart_rx;

    output [2:0]led;
    output uart_tx;
```

```
    wire [DATA_WIDTH-1:0]data;
    wire Rx_Done;
    wire [7:0]data_byte;

    uart_data_rx
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_rx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .uart_rx(uart_rx),

        .data(data),
        .Rx_Done(Rx_Done),
        .timeout_flag(led[0]),

        .Baud_Set(3'd4)
     );

    uart_data_tx
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )uart_data_tx(
        .Clk(Clk),
        .Rst_n(Rst_n),

        .data(data),
        .send_en(Rx_Done),
        .Baud_Set(3'd4),

        .uart_tx(uart_tx),
        .Tx_Done(led[1]),
        .uart_state(led[2])
    );

endmodule
```

顶层中设置 DATA_WIDTH 为 32，波特率为 115200，超时信号、发送完成信号、串口状态信号被连接到 LED 灯上进行显示。

在添加完测试顶层且综合无误后，需要为设计添加管脚分配和电平约束，本次设计管脚分配表如表 9-1 所示：

表 9-1 管脚分配表

Pin Name	Signial Name	Pin NO.	Pin Name	Signial Name	Pin NO.

CLK_G	CLK	T9	LED2	led[2]	B9
KEY0	Rst_n	B16	LED1	led[1]	C14
UART_RXD	uart_rx	U8	LED0	led[0]	D14
UART_TXD	uart_tx	V8			

引脚电平约束为 LVCMOS 33*，设置完成后编译设计并生成比特流，接下来准备对设计板级验证。

9.4 板级验证

本节将在高云开发板上使用设计的多字节串口收发模块，通过串口回环的方式，对来自电脑的数据回环。通过对比回环数据与发送数据是否一致，对设计模块进行验证。

9.4.1 系统所需硬件

1. 高云开发板 x1
2. 高云下载器 x1
3. 串口线 x1
4. 电源线 x1

9.4.2 硬件连接

连接开发板的下载线和电源线，使用串口线一端连接开发板上的 UART 接口，一端连接电脑的 USB 口，如下图 8-6 所示。

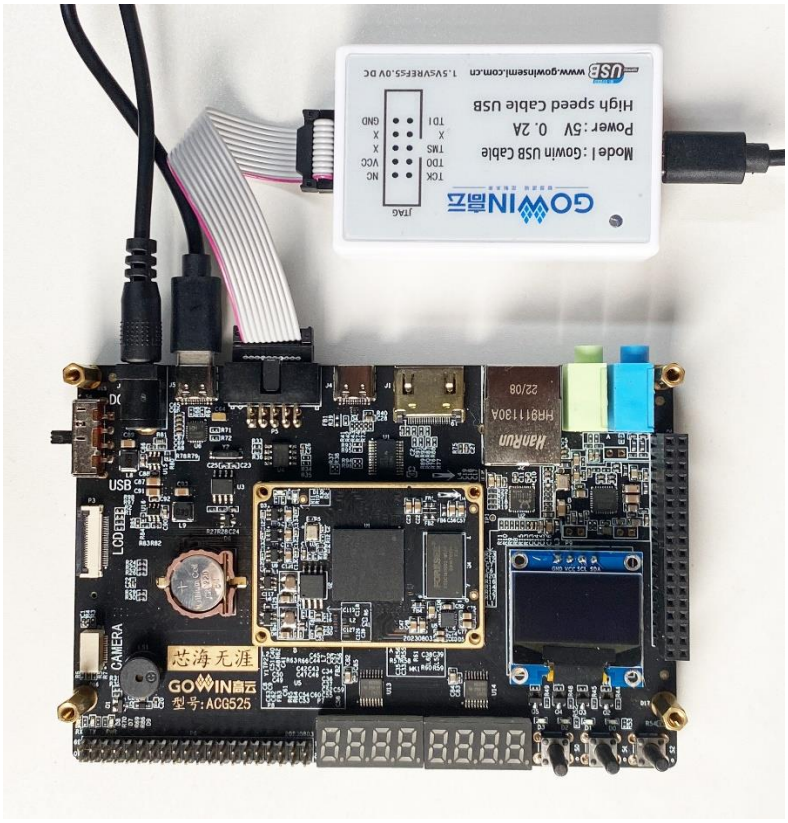


图 9-9 硬件连接

连接完成后，确保开发板电源拨码开关拨到 ON 侧，接下来将生成好的 bit 烧录到开发板中。

烧录完成后打开设备管理器，查询串口端口号，设备管理器中串口的名称如图 9-10：

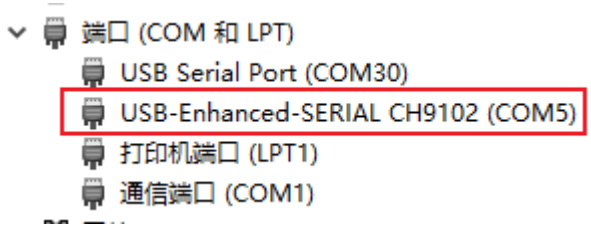


图 9-10 查询端口号

这里笔者电脑串口端口号为 COM5，用户在查询时，以自己电脑中的端口号为准。随后打开串口猎人，在软件中连接上刚刚查询到的端口，设置波特率为 115200。接着输入 4 字节数据，点击发送开始回环，如图 9-11 实验结果所示：

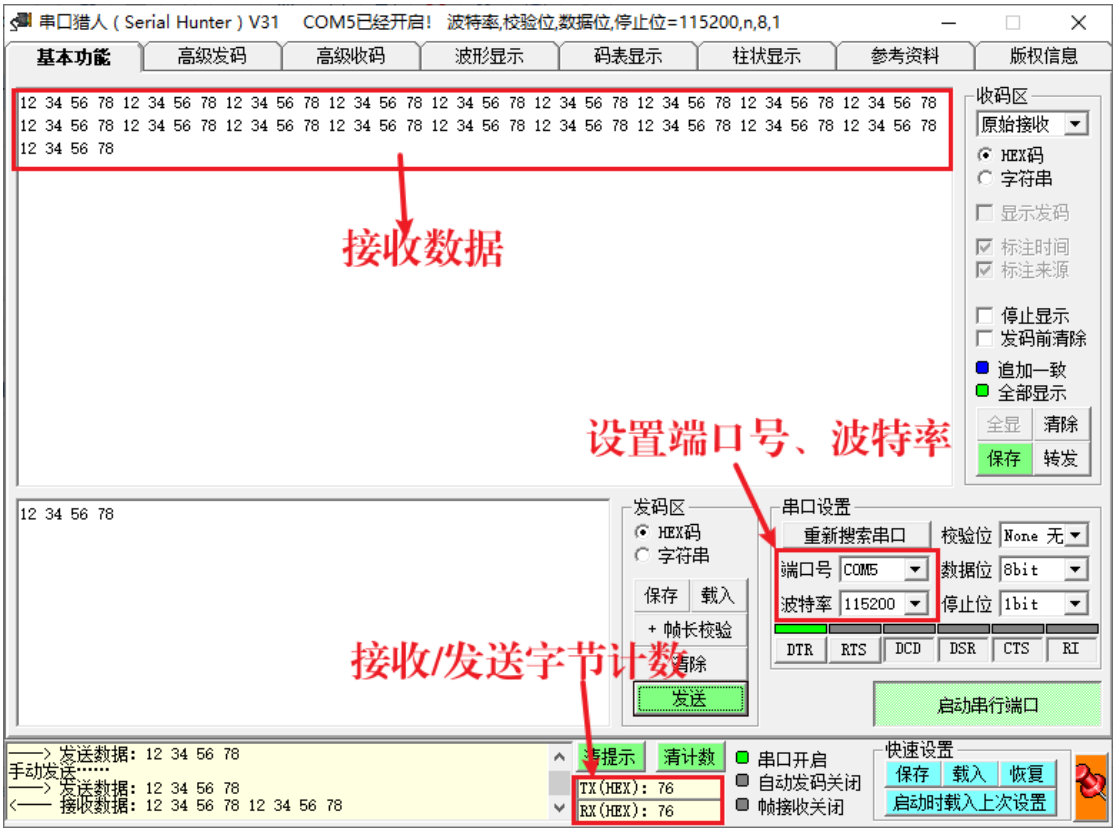


图 9-11 实验结果

这里笔者发送了多次，一共发送 76 字节数据。可以看到，软件接收到的回环数据也为 76 字节，且接收到的数据与发送数据一致。证明功能正常，本次多字节串口收发模块设计成功。

9.5 总结

本章带大家完成了多字节串口接收模块和多字节串口发送模块的设计，并通过仿真测试和板级验证带领大家验证了设计的可行性。为了方便使用，模块中部分信号使用的 parameter 参数定义，用户可以通过顶层参数传递，根据自己的设计需求对这些参数值修改。

10 状态机设计实例

工程源码	|----02_设计实例 |----ch10_fsm_hello
相关视频课程	
说明	

章节导读

状态机全称是有限状态机（Finite State Machine、FSM），是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。本节将学习状态机的相关概念并使用状态机实现特定字符串的检测。理解一段式、两段式以及三段式状态机的区别以及优缺点。

10.1状态机工作原理

状态机分为摩尔（Moore）型有限状态机与米利（Mealy）型有限状态机。摩尔状态机的输出是只由当前状态确定（不直接依赖于输入）。米利有限状态机的输出不止与其当前状态有关还与它的输入相关，这也是与摩尔有限状态机的不同之处。其原理图分别如下图 10-1 摩尔型状态机、如下图 10-2 米利型状态机所示

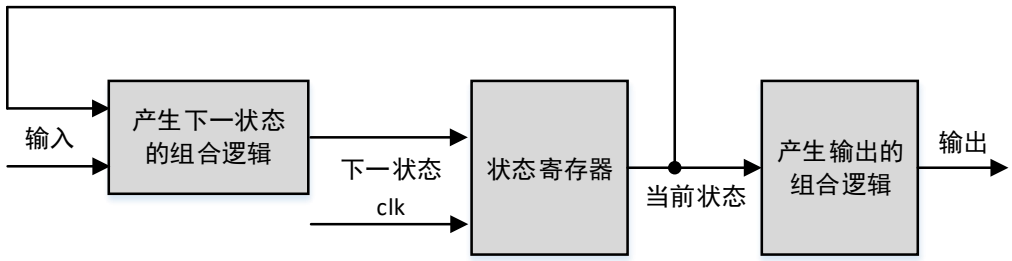


图 10-1 摩尔型状态机

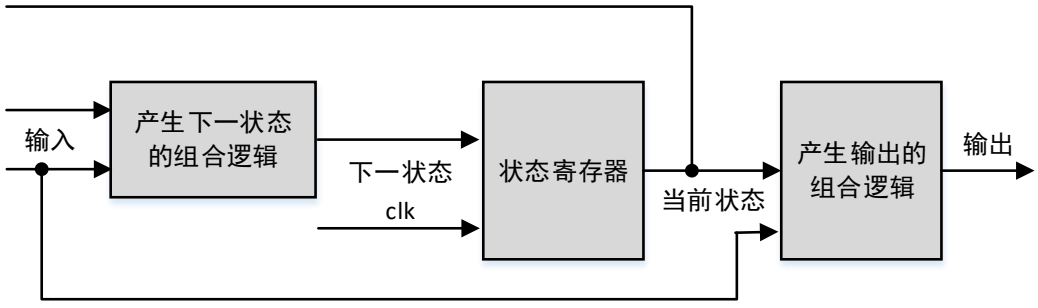


图 10-2 米利型状态机

状态机可根据控制信号按照预先设定的状态进行状态转移，这就出现了如何对状态进行有效编码的问题。编码方式，最简单的就是直接使用二进制编码进行表示，除此之外还有使用格雷码、独热码。假设有八个状态从 A 到 H，利用不同的编码格式分别如下表 10-1 所示。

表 10-1 状态不同的编码格式

编码格式 / 状态	二进制	独热码	格雷码
A	3`b000	8`b0000_0000	4`b0000
B	3`b001	8`b0000_0010	4`b0001
C	3`b010	8`b0000_0100	4`b0011
D	3`b011	8`b0000_1000	4`b0010
E	3`b100	8`b0001_0000	4`b0110
F	3`b101	8`b0010_0000	4`b0111
G	3`b110	8`b0100_0000	4`b0101
H	3`b111	8`b1000_0000	4`b0100

从上表中可以发现：独热码，每一个状态均使用一个寄存器，在与状态比较时仅仅需要比较一位，相比其他译码电路简单；格雷码，所需寄存器数与二进制码一样，译码复杂，但相邻位只跳动一位，一般用于异步多时钟域多 bit 位的转换，如异步 FIFO；二进制码，最为常见的编码方式，所用寄存器少，译码较复杂。

按照 FPGA 厂商给的建议，选择哪一种编码格式是要根据状态机的复杂度、器件类型以及从非法状态中恢复出来的要求来确定。在使用不同的编码格式生成出来的 RTL 视图中可以看出二进制比独热码使用更少的寄存器。二进制用 7 个寄存器就可以实现 100 个状态的状态机，但是独热码就需要 100 个寄存器。但是另一方面，虽然独热码使用更多的寄存器，但是其组合逻辑相对简单。一般在 CPLD 中，由于提供较多的组合逻辑资源而推荐使用二进制，而在 FPGA 中提供较多的时序逻辑而推荐使用独热码。

状态机描述方式，可分为一段式、两段式以及三段式。

一段式，整个状态机写到一个 always 模块里面。在该模块中既描述状态转移，又描述状态的输入和输出。

两段式，用两个 always 模块来描述状态机。其中一个 always 模块采用同步时序描述状态转移，另一个模块采用组合逻辑判断状态转移条件，描述状态转移规律及其输出。

三段式，在两个 always 模块描述方法基础上，使用三个 always 模块。一个 always 模块采用同步时序描述状态转移，一个 always 采用组合逻辑判断状态转移条件，描述状态转移规律，另一个 always 模块描述状态输出(可以用组合电路

输出，也可以时序电路输出)。

可以看出两段式有限状态机与一段式有限状态机的区别是将时序部分（状态转移）和组合部分（判断状态转移条件和产生输出）分开，写为两个 always 语句，即为两段式有限状态机。将组合部分中的判断状态转移条件和产生输出再分开写，则为三段式有限状态机。二段式在组合逻辑特别复杂时，注意需在后面加一个触发器以消除组合逻辑对输出产生的毛刺的影响。三段式则没有这个问题，这是由于第三个 always 会生成触发器。其实现在的器件根本不在乎这一点资源消耗，推荐使用二段式或者三段式以及输出寄存的状态机输出来描述有限状态机。

编写状态机还应主要注意的事项是，为了避免不必要的锁存器生成，需要穷举所有状态对应的输出动作，或者使用 default 来定义未定义状态动作；在定义状态时，推荐使用参数定义 localparam 或 parameter，这样可以在编写时状态更清晰且不容易出错，也方便修改。在复位或者跑飞能回到初始态或者预定态，所以在设计中要有异步或者同步复位来确保状态机上电有个初始态。

10.2字符串检测状态机实现

为了实现对字符串“hello”的检测，首先画出其状态转移图，如下图 10-3 所示。

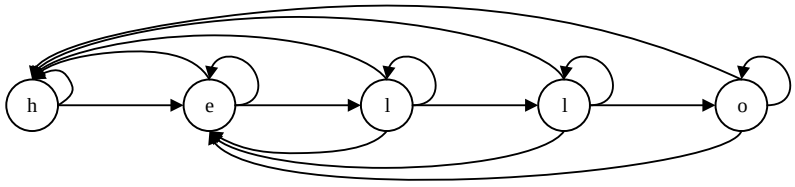


图 10-3 字符串“hello”检测状态转移图

由上图可以看出，如果在状态不符合转换条件，那么状态机要么回到初始态 h，要么回到状态 e（不满足向后转移条件，但是输入字符为‘h’），且每一个状态都有特定的方向，其输出仅由当前状态决定，因此这是一个摩尔型状态机。

新建一个名为hello的工程，并在本工程中添加并新建一个source file文件，并以 hello.v 保存。整个字符串“hello”检测模块示意图如下图 10-4 所示。

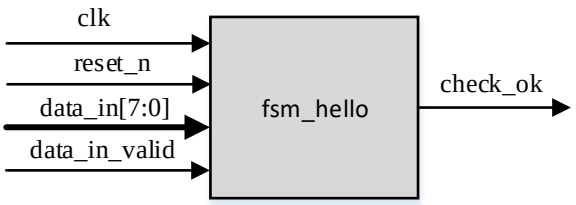


图 10-4 字符串检测模块示意图

表 10-2 状态机模块端口功能描述

端口名称	I/O	端口功能描述
clk	I	模块的工作时钟，频率为 50MHz，
reset_n	I	模块复位，低电平复位
data_in	I	字符数据输入端，位宽 8bit
data_in_valid	I	字符数据输入有效标志，为高表示当前有字符数据输入，为低表示无字符数据输入。
check_ok	O	字符串“hello”检测成功标志位，检测成功输出 1 个时钟周期高电平。

首先，使用独热码对各个状态进行定义。

```
localparam
    CHECK_h   = 5'b0_0001,
    CHECK_e   = 5'b0_0010,
    CHECK_l1  = 5'b0_0100,
    CHECK_l2  = 5'b0_1000,
    CHECK_o   = 5'b1_0000;
```

然后，设置好初始状态。

```
always@(posedge clk or posedge reset)
if(reset)begin
    check_ok <= 1'b0;
    state <= CHECK_h;
end
else begin
    case(state)
    //此部分代码略，后面会详细描述
    default:state <= CHECK_h;
    endcase
end
```

依照图 10-3 开始状态转移部分的编写。当在初始态 CHECK_h 时如果检测到有字符‘h’输入，则跳入 CHECK_e 状态开始字符‘e’的检测，否则保留在 CHECK_h 状态等待字符‘h’输入。

```
CHECK_h:
    begin
        check_ok <= 1'b0;
    if(data_valid && data_in == "h")
```

```
        state <= CHECK_e;
    else
        state <= CHECK_h;
    end
```

当在状态 CHECK_e 时，如果检测到有字符‘e’输入，则进入 CHECK_l1 状态开始第一个字符‘l’的检测；如果检测到有字符‘h’输入，则仍然停留在 CHECK_e 状态等待字符‘e’输入；如果检测到有其他字符输入时，跳回初始状态等待新的一轮检测；否则一直在状态 CHECK_e 等待字符的输入。

```
CHECK_e:
    if(data_valid && data_in == "e")
        state <= CHECK_l1;
    else if(data_valid && data_in == "h")
        state <= CHECK_e;
    else if(data_valid)
        state <= CHECK_h;
    else
        state <= CHECK_e;
```

当在状态 CHECK_l1 时，如果检测到有字符‘l’输入，则进入 CHECK_l2 状态开始第二个字符‘l’的检测；如果检测到有字符‘h’输入，则跳转到 CHECK_e 状态等待字符‘e’输入；如果检测到有其他字符输入时，跳回初始状态等待新的一轮检测；否则一直在状态 CHECK_l1 等待字符的输入。

```
CHECK_l1:
if(data_valid && data_in == "l")
    state <= CHECK_l2;
else if(data_valid && data_in == "h")
    state <= CHECK_e;
else if(data_valid)
    state <= CHECK_h;
else
    state <= CHECK_l1;
```

当在状态 CHECK_l2 时，如果检测到有字符‘l’输入，则进入 CHECK_o 状态开始字符‘o’的检测；如果检测到有字符‘h’输入，则跳转到 CHECK_e 状态等待字符‘e’输入；如果检测到有其他字符输入时，跳回初始状态等待新的一轮检测；否则一直在状态 CHECK_l2 等待字符的输入。

```
CHECK_l2:
if(data_valid && data_in == "l")
    state <= CHECK_o;
else if(data_valid && data_in == "h")
    state <= CHECK_e;
else if(data_valid)
    state <= CHECK_h;
```

```
else
    state <= CHECK_l2;
```

当在状态 CHECK_o 时，如果检测到有字符‘o’输入，则说明完成一次“hello”字符串的检测，回到初始状态等待新的一轮检测，同时控制 check_ok 产生一个时钟脉冲；如果检测到有字符‘h’输入，则跳转到 CHECK_e 状态等待字符‘e’输入；否如果检测到有其他字符输入时，跳回初始状态等待新的一轮检测；否则一直在状态 CHECK_o 等待字符的输入。

```
CHECK_o:
    if(data_valid && data_in == "h")
    state <= CHECK_e;
    else if(data_valid)begin
        state <= CHECK_h;
        if(data_in == "o")
            check_ok <= 1'b1;
        else
            check_ok <= 1'b0;
    end
    else
        state <= CHECK_o;
```

对设计好的代码进行分析和综合直至没有错误以及关键警告。

10.3激励创建及仿真测试

为了测试仿真编写测试激励文件，在工程下新建 Verilog File 并命名为 fsm_hello_tb.v。本激励文件除产生正常的时钟以及复位信号外，还模拟了不同字符输入情况。Testbench 设计中对单个字符产生以一个任务的形式设计，通过调用任务产生不同字符串的产生，验证设计的正确性，具体代码如下。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module fsm_hello_tb;

    reg clk;
    reg reset_n;

    reg data_valid;
    reg [7:0]data_in;

    wire check_ok;

    fsm_hello fsm_hello(
        .clk(clk),
```

```
        .reset_n(reset_n),
        .data_valid(data_valid),
        .data_in(data_in),
        .check_ok(check_ok)
    );

    initial clk = 1;
    always#(`CLK_PERIOD/2) clk = ~clk;

    initial begin
        reset_n = 0;
        data_valid = 0;
        data_in = 0;
        #(`CLK_PERIOD*20);
        reset_n = 1;
        #(`CLK_PERIOD*20 + 1);

        repeat(2) begin
            gen_char("I");
            #(`CLK_PERIOD);
            gen_char("A");
            #(`CLK_PERIOD);
            gen_char("h");
            #(`CLK_PERIOD);
            gen_char("e");
            #(`CLK_PERIOD);
            gen_char("l");
            #(`CLK_PERIOD);
            gen_char("l");
            #(`CLK_PERIOD);
            gen_char("l");
            #(`CLK_PERIOD);
            gen_char("h");
            #(`CLK_PERIOD);
            gen_char("e");
            #(`CLK_PERIOD);
            gen_char("l");
            #(`CLK_PERIOD);
            gen_char("l");
            #(`CLK_PERIOD);
            gen_char("o");
            #(`CLK_PERIOD);
            gen_char("e");
            #(`CLK_PERIOD);
            gen_char("h");
            #(`CLK_PERIOD);
            gen_char("h");
```

```
            #(`CLK_PERIOD);
            gen_char("o");
            #(`CLK_PERIOD);
        end

            #200;
            $stop;
    end

    task gen_char;
        input [7:0]char;
        begin
            data_in = char;
            data_valid = 1'b1;
            #(`CLK_PERIOD);
            data_valid = 1'b0;
        end
    endtask
endmodule
```

打开 Modelsim 新建一个工程，命名为“fsm_hello”，并添加文件“fsm_hello.v”和“fsm_hello_tb.v”，然后进行仿真配置，配置好之后运行仿真，可以看到如下图 10-5 所示的波形文件，可以看出在复位信号置高之前状态均不发生转移。在复位有效后，只有当输入字符（data_valid 为高）时，状态才会根据设计进行转换，且没有出现转移错误，在正确检测到字符串“hello”时，check_ok 产生一个周期脉冲表示一次检测成功。

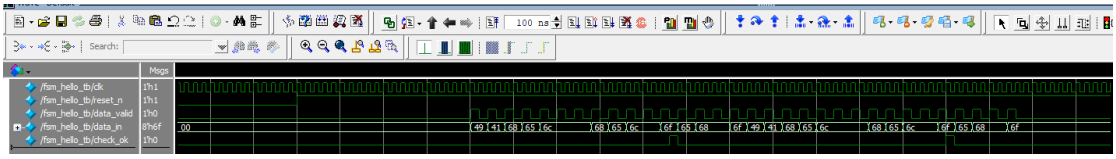


图 10-5 仿真波形文件

至此就完成了一个简单的状态机设计，在后面的章节中会经常用到状态机设计思想。这里也就不再对二段式、三段式状态机展开，将在具体实例中进行详细说明。

10.4字符串检测上板测试

通过上面字符串检测状态机的学习，结合前面章节学习的串口接收模块，设计一个串口接收字符串检测系统。整个系统的结构框图如下图 10-6 所示。系统功能为 FPGA 通过串口接收模块接收 PC 通过串口发送的字符串数据流，然后

通过字符串检测模块对字符串进行检测，每次完整接收到“hello”字符串就让 LED 灯翻转。

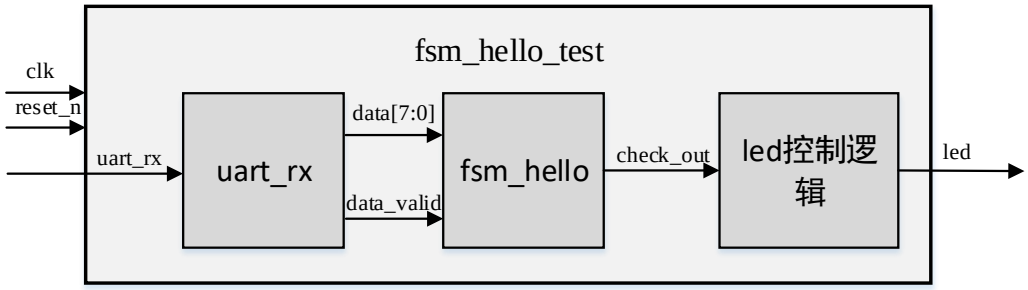


图 10-6 串口接收字符串检测系统框图

系统设计的顶层为 fsm_hello_test，具体设计的代码如下。

```
module fsm_hello_test(
    clk,
    reset_n,

    uart_rx,
    led
);
    assign reset=~reset_n;
    input clk;   //系统时钟输入__50M
    input reset_n;   //复位信号输入，低有效

    input uart_rx;   //串口输入信号

    output led;  //LED 指示灯
    reg led;

    wire [7:0]data;         //串口接收数据,作为字符数据输入
    wire data_valid;  //1byte 数据接收完成标志

    wire check_ok;    //字符串检测成功标诿

  //接收串口数据作为字符串检测的输入数据
    uart_byte_rx uart_byte_rx(
        .clk(clk),
        .reset_n(reset_n),

        .baud_set(3'd0),
        .uart_rx(uart_rx),

        .data_byte(data),
        .rx_done(data_valid)
    );
```

```
    fsm_hello fsm_hello(
        .clk(clk),
        .reset_n(reset_n),

        .data_valid(data_valid),
        .data_in(data),

        .check_ok(check_ok)
    );

    always@(posedge clk or posedge reset)
    if(reset)
        led <= 1'b0;
    else if(check_ok)
        led <= ~led;
    else
        led <= led;

endmodule
```

顶层模块的仿真可以仿照串口接收模块来做，这里就不额外讲解顶层的仿真代码的设计，读者自己进行仿真验证顶层设计功能的正确性。在仿真确定顶层设计功能正确情况下，对顶层输入/输出管脚进行约束，led 绑定的是板子上 LED0 的管脚。具体管脚约束情况如下图 10-7 所示。

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	clk	input		T9	4	False	LVCMOS33	OFF
2	led	output		D14	1	False	LVCMOS33	8
3	reset_n	input		B16	1	False	LVCMOS33	OFF
4	uart_rx	input		U8	5	False	LVCMOS33	OFF

图 10-7 管脚约束

管脚约束完成后，对工程进行全编译并生成 bit 文件，将 bit 文件下载到开发板上，在 bit 文件下载到 FPGA 后，可以看到绑定的管脚的 LED0 是熄灭的。通过数据线将电脑与 FPGA 串口相连接，在电脑上打开一个串口软件（以串口猎人为例），选择与 FPGA 相连串口端口号，设置串口波特率为 9600，发送数据格式设置为字符串。在串口软件上发送几组字符串数据，观察板子 LED0 的变化。下面依次发送 3 次字符串的串口软件窗口。

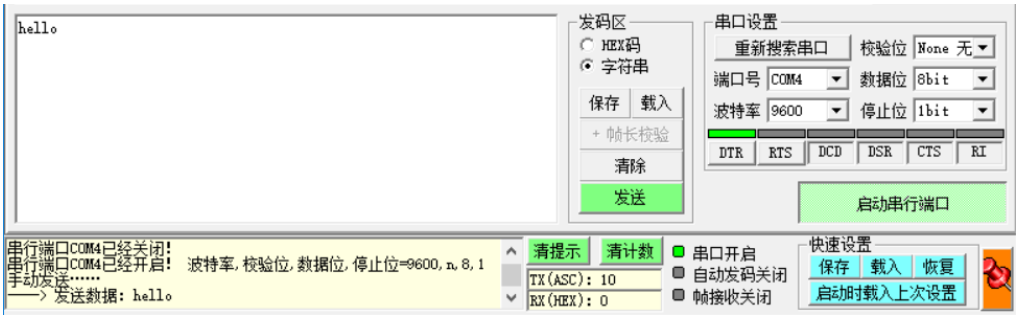


图 10-8 发送测试字符串 1

图 10-9 发送测试字符串 2

图 10-10 发送测试字符串 3

（1）当发送字符串“hello”时，板子上 LED0 由灭变亮，测试结果正确。

（2）当发送字符串“helllo”时，板子上 LED0 没有变化，测试结果正确。

（3）当发送字符串“lhoolehello” 板子上 LED0 由亮变灭，测试结果正确。

10.5总结

本章实验带领读者了解了状态机的工作原理以及字符串检测状态机的实现。读者可以通过电脑发送其他不同字符串数据进行测试，观察板子上 LED0 变化，当发送字符串有完整的“hello”字符串时，LED0 就会翻转一次（亮变灭或灭变亮），通过上板测试可以验证设计的正确性，整个系统设计是到达预期效果的。

11 独立按键消抖设计与验证

工程源码	|----02_设计实例 |----ch11_key_filter
相关视频课程	
说明	

章节导读

按键作为一种基本的人机交互元件，在电子设计中广泛使用，从系统复位到控制设置均可以看到其身影。当前按键的种类也很多，常见的如多向按键、自锁按键、薄膜按键等。

本节将单 Bit 数据的异步信号同步以及边沿检测的方法引入到 FPGA 中常用的按键消抖设计。并在仿真测试激励文件中引入随机数发生函数以及激励模型。最后再以一个实际的例子来演示模块化设计的方式。

11.1按键物理结构及电路设计

普通按键的硬件示意图如下图 11-1 所示。

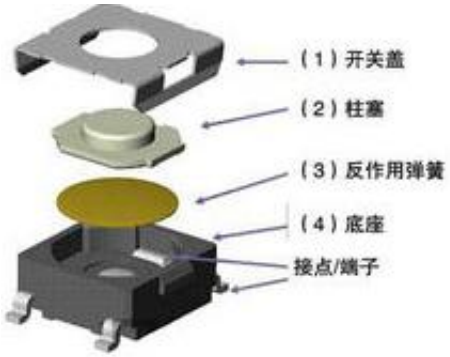


图 11-1 按键硬件结构示意图

高云开发板所载的为两脚贴片按键，分别位于学习板正面的右下角，原理图如下图所示。由原理图可以看出，按键未按下时 IO 口为高电平，按键按下时则变为低电平，因此系统即可通过检测 IO 的电平来判断按键的状态。

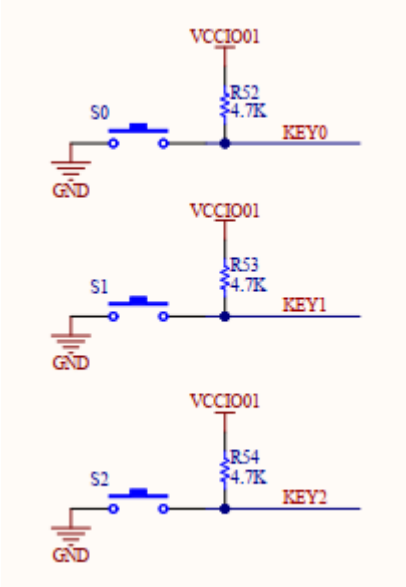


图 11-2 按键原理图

按键结构示意图中可以看到按键存在一个反作用弹簧，因此当按下或者松开时均会产生额外的物理抖动，物理抖动便会产生电平的抖动。在按键从按下再到松开的过程中，其电平变化如下图 11-3 所示，上为理想波形输出，下为实际波形输出。

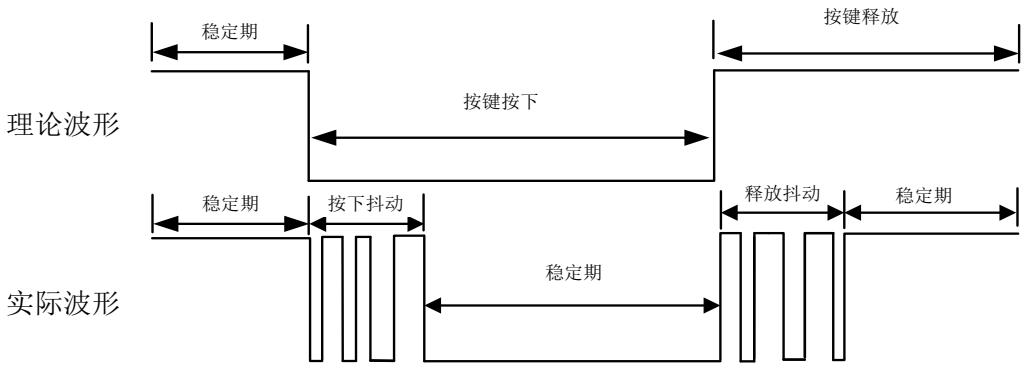


图 11-3 按键从按下到松开的电平变化

上图中，产生的抖动次数以及间隔时间均是不可预期的，这就需要通过滤波来消除抖动可能对外部其他设备造成的影响。一般情况下抖动的总时间会持续 20ms 以内。这种抖动，可以通过硬件电路或者逻辑设计的方式来消除，也可以通过软件的方式完成。其中硬件电路消除抖动适用于按键数目较少的场合。

11.2硬件电路实现按键消抖

使用基于与非门的 RS 触发器来消除抖动的电路图，如下图 11-4 所示。假

设初始状态开关与 B 接通，此时$\overline{S}$=1，$\overline{R}$=0，触发器置 0 即 Q=0；当 B 切换到 A 的过程中，存在开关既没有与 A 也没有与 B 接触，此时$\overline{S}$=1，$\overline{R}$=1，触发器保持即 Q=0；当切换到 A 瞬间时有$\overline{R}$=1，$\overline{S}$=0，触发器置 1 即 Q=1。此时即使当 A 出现抖动即$\overline{S}$=1，触发器的状态仍能保持当前状态即 Q=1 不变。整个过程波形图如图所示$\overline{Q}$。同理，可以分析出开关由 A 切换到 B 时触发器的状态变化。

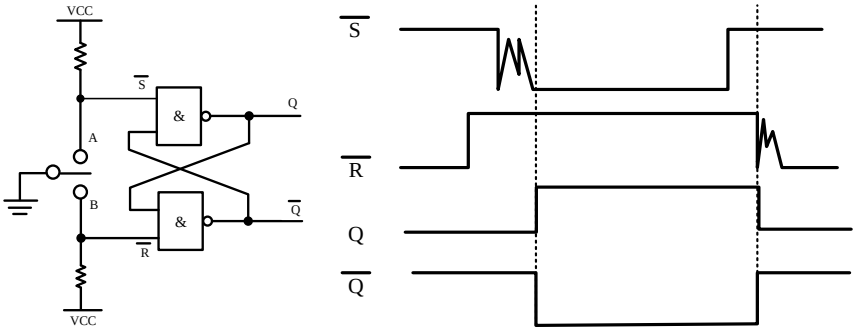


图 11-4 基于 NAND 的 RS 触发器消抖电路及波形分析

也可以采用或非门构成的 RS 触发器来实现功能。其电路如图 11-5 所示，可对照基于以上分析过程自行分析电路工作状态。

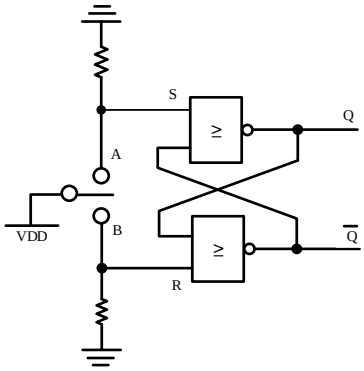


图 11-5 基于 NOR 的 RS 触发器消抖电路

可以看出，以上两种接法只适用于单刀双掷类开关按键，并不适用于目前常见的两脚或者四脚按键。针对此类按键目前常用如下图 11-6 以及如下图 11-7 所示的 RC 电路。

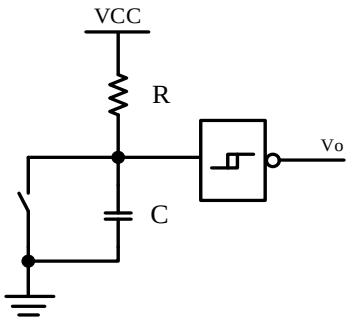


图 11-6 高电平输出

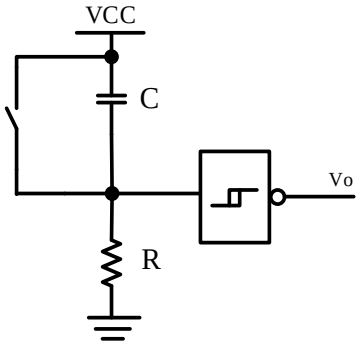


图 11-7 低电平输出

其原理是利用电容以及电阻对波形进行积分，然后通过施密特触发器（比较器）来进行波形整形进而得到理想的输出波形。对于高电平输出电路，当开关断开时，电容器充电其 Vc=5V，Vo 输出为 0；当开关闭合时，电容放电至 Vc=0，Vo 输出为 5V；按键抖动时，如果充电时间常数 RC 较小，电容将一直进行充放电过程，充电到 5V 放电到 0V。如果将充电时间常数 RC 设置足够大，当开关从断开状态向闭合状态抖动时，电容没有充电到施密特触发器的阈值电压以上，则输出 Vo 保持 5V。当开关断开时，电容器重新充电到 5V，超过阈值电压，则输出变化为低电平。整个过程如图 11-8 所示，在闭合或释放时出现抖动，但一旦真正稳定了，输出端只会变化一次。

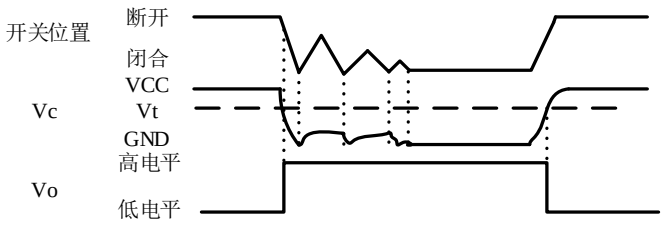


图 11-8 高电平输出的 RC 电路

也可以用 555 定时器组成的单稳态触发器，同样可以消除按键的抖动。输出端 Vo 平时为低电平，当有脉冲 Vi 下降沿到来时，反转为高电平，此时电源

通过 R 对 C 充电，当电容 C 上电压达到 VCC*2/3 时，555 定时器内部放电管导通，C 通过其快速放电，输出端翻转为低电平。在电容电压从 0 升至 2/3VCC 过程中，输出端一直为高，此时即使有抖动输出也不会发生变化。通过合理的选择暂稳态时间 Td，就能达到防抖动的效果。

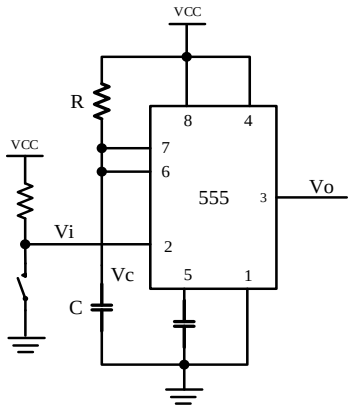


图 11-9 基于单稳态触发器的消抖电路

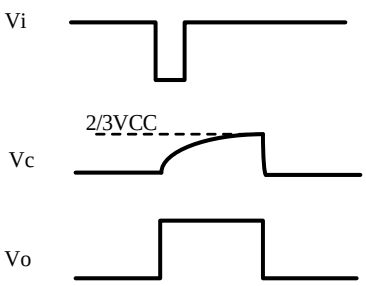


图 11-10 基于单稳态触发器的消抖电路波形分析

11.3状态机实现按键消抖

11.3.1模块接口设计

对于单按键的消抖模块，其接口如下图 11-11 所示，接口声明功能描述如下表 11-1：

图 11-11 按键消抖模块接口

表 11-1 按键消抖模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块工作时钟，50M 时钟

reset_n	I	模块复位信号
key_in	I	按键输入
key_flag	O	按键状态切换的标志
key_state	O	按键状态标志

11.3.2状态机转移状态以及条件设计

对于 FPGA 设计而言，通常使用状态机来进行按键消抖处理。在如下图中可看出，若按照状态机概念对其进行状态编码分析得到以下状态：未按下时空闲状态 IDLE、按下抖动滤除状态 FILTER0、按下稳定状态 DOWN 以及释放抖动滤除状态 FILTER1。其独热码编码分别为 4'b0001，4'b 0010，4'b 0100 以及 4'b 1000。其状态转换图如下图 11-12 所示，可以看出其为米利型状态机。

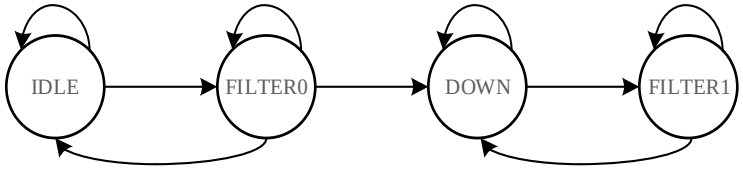


图 11-12 状态转移图

其状态转移条件如下表 11-2 所示。

表 11-2 按键消抖状态转移条件

当前状态	下一状态	转移条件
IDLE	FILTER0	检测到按键输入的下降沿，则启动按下消抖；
FILTER0	IDLE	在按下消抖中如果检测到上升沿且未超过 20ms 计时，则认为按键还在抖动中，仍未稳定因此回到初始态；
FILTER0	DOWN	检测到下降沿即启动计时，若超过 20ms 计时则认为确实按下；
DOWN	FILTER1	按下状态稳定后，如果检测到上降沿，则启动释放消抖；
FILTER1	DOWN	在按键松开恢复消抖过程中，检测到上升沿即启动计时，检测到下降沿且未超过 20ms 计时，则认为还在抖动中，继续消抖；
FILTER1	IDLE	检测到上升沿即启动计时，若超过 20ms 计时则认为确实松开释放完成。

11.3.3按键输入信号边沿检测

从状态机的设计来看，需要对按键输入信号 key_in 的边沿进行检测，关于边沿检测的理论知识可参考串口接收模块中讲解，这里不重复讲解，具体按键数据信号的边沿检测代码如下。

```
always@(posedge clk or posedge reset)
if(reset)begin
    key_in_sync1 <= 1'b0;
    key_in_sync2 <= 1'b0;
end
```

```
else begin
    key_in_sync1 <= key_in;
    key_in_sync2 <= key_in_sync1;
end

//使用 D 触发器存储两个相邻时钟上升沿时外部输入信号（已经同步到系统时钟域中）的电平状态
always@(posedge clk or posedge reset)
if(reset)begin
    key_in_reg1 <= 1'b0;
    key_in_reg2 <= 1'b0;
end
else begin
    key_in_reg1 <= key_in_sync2;
    key_in_reg2 <= key_in_reg1;
end

//产生跳变沿信号
assign key_in_nedge = !key_in_reg1 & key_in_reg2;
assign key_in_pedge = key_in_reg1 & (!key_in_reg2);
```

11.3.4计数器设计

在状态转移表中还应有 20ms 计数器模块以及计数器使能模块，这里也可以合并成一个 always 模块。一般推荐一个 always 块只对一个信号进行操作。

```
always@(posedge clk or posedge reset)
if(reset)
    cnt <= 20'd0;
else if(en_cnt)
    cnt <= cnt + 1'b1;
else
    cnt <= 20'd0;

always@(posedge clk or posedge reset)
if(reset)
    cnt_full <= 1'b0;
else if(cnt == 20'd999_999)
    cnt_full <= 1'b1;
else
    cnt_full <= 1'b0;
```

11.3.5状态机设计

现在开始状态机设计，首先用本地参数化对状态机的状态进行定义。

```
localparam
```

```
    IDLE= 4'b0001,
    FILTER0= 4'b0010,
    DOWN= 4'b0100,
    FILTER1= 4'b1000;
```

由于状态以及判断条件较少，此处先用一段式状态机来进行描述。当复位时候将计数器清零，状态回到 IDLE，key_flag 与 key_state 也回到初始态。

```
always@(posedge clk or posedge reset)
if(reset)begin
    en_cnt <= 1'b0;
    state <= IDLE;
    key_flag <= 1'b0;
    key_state <= 1'b1;
end
else begin
    case(state)
        default:begin
            state <= IDLE;
            en_cnt <= 1'b0;
            key_flag <= 1'b0;
            key_state <= 1'b1;
        end
    endcase
end
```

在未按下时状态为 IDLE 时，如果检测到下降沿则状态进入按下抖动滤除状态 FILTER0，并使能计数器，否则继续保持 IDLE 状态。

```
IDLE :begin
    key_flag <= 1'b0;
    if(key_in_nedge)begin
        state <= FILTER0;
        en_cnt <= 1'b1;
    end
    else
        state <= IDLE;
end
```

当在 FILTER0 状态时，如果 20ms 尚未计时结束就有上升沿到来，则认为此时还是按键按下抖动过程，状态回到 IDLE 并清 0 计数器。按下过程中当最后一次抖动后，不会存在上升沿，计数器则可以一直计数，计数满后则将 key_flag 置 1、key_state 置 0，状态进入按下稳定状态 DOWN 并将计数器清 0。这样就可以通过判断 key_flag && !key_state 来确定按键的状态，为 1 则按下。

```
FILTER0:begin
    if(cnt_full)begin
        key_flag <= 1'b1;
        key_state <= 1'b0;
```

```
        en_cnt <= 1'b0;
        state <= DOWN;
    end
    else if(key_in_pedge)begin
        state <= IDLE;
        en_cnt <= 1'b0;
    end
    else
        state <= FILTER0;
end
```

进入按键稳定状态 DOWM 后，将 key_flag 清 0。如果检测到上升沿则进入释放抖动滤除状态 FILTER1，否则保持当前态。

```
DOWN:begin
        key_flag <= 1'b0;
        if(key_in_pedge)begin
            state <= FILTER1;
            en_cnt <= 1'b1;
        end
        else
            state <= DOWN;
end
```

进入 FILTER1 状态后，如果 20ms 计数尚未结束就检测到下降沿，则认为此时还是按键释放抖动过程，状态回到 DOWN 并清 0 计数器。释放过程中当最后一次抖动后，不会存在下降沿，计数器则可以一直计数，计数满后则将 key_flag 与 key_state 均置 1，状态进入 IDLE 并将计数器清 0，等待下一次按键被按下。

```
FILTER1:begin
    if(cnt_full)begin
        key_flag <= 1'b1;
        key_state <= 1'b1;
        state <= IDLE;
        en_cnt <= 1'b0;
    end
    else if(key_in_nedge)begin
        en_cnt <= 1'b0;
        state <= DOWN;
    end
    else
        state <= FILTER1;
end
```

这里如果改写为两段式状态机，则如下所示，状态转移部分省略。

```
always@(posedge clk or posedge reset)
if(reset)
    curr_state <= IDLE;
else
```

```
    curr_state <= next_state;

always@(*)
begin
    case(curr_state)
    //此部分如一段式状态机
    endcase
end
```

11.4testbench 设计及仿真测试

testbench 文件中激励除产生正常的时钟以及复位信号外，还模拟了按键从按下到松手释放的过程，本次仿真激励对按键采用单独设计仿真模型的方式进行设计，这样整个激励文件的内部结构即为如下图 11-13 所示：

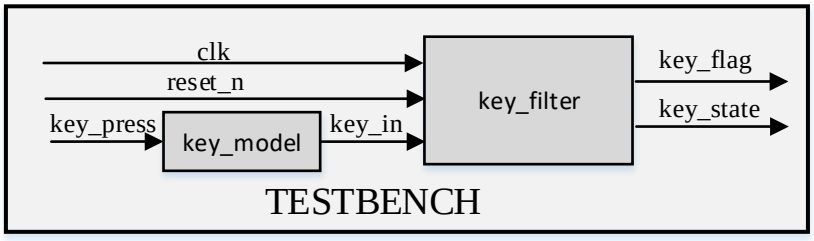


图 11-13 激励文件内部结构

11.4.1按键模型 key_model 的设计

按键模型的功能是模拟真实按键过程中信号的输出，包括按键过程中的抖动。考虑到真实按键过程中的抖动时间是不确定，这里引入 verilog 中的随机数发生函数$random 来产生抖动。

$random 这一系统函数可以产生一个有符号的 32bit 随机整数。一般的用法是$random%b，其中 b>0。这样就会生成一个范围在(-b+1):(b-1)中的随机数。如果只得到正数的随机数，可采用{$random}%b 来产生。在本节中需要产生在 20ms 以内的按下抖动与松手抖动，实际上应该产生 25'd20_000_000 以内随机数的抖动，这里为了节约仿真时间，只产生一个 16 位的随机数也就是 0 到 16'd65535。

11.4.2任务的使用

任务 task 是 testbench 中常用的语法，其语法如下：

```
task <任务名>;
    <端口及数据类型声明语句>
```

```
    <语句 1>
    <语句 2>
    <语句 n>
endtask
```

任务的调用的语法如下：

```
<任务名>（端口 1,端口 2,...端口 n）;
```

对于一次按键按下和松手整个过程以任务的形式进行设计，这里实现了五十次的 0~65535ns 按下抖动，然后 key_in 赋固定值 0 且延时 25ms（大于 20ms 即为稳定），同时也实现了释放抖动后，key_in 赋固定值 1 且延时 25ms。

```
task key_gen;
begin
key_out = 1'b1;
repeat(50)begin
    myrand = {$random}%65536;//0~65535;
    #myrand key_out = ~key_out;
end
key_out = 0;
#25000000;

repeat(50)begin
    myrand = {$random}%65536;//0~65535;
    #myrand key_out = ~key_out;
end
key_out = 1;
#25000000;
end
endtask
```

在上面的基础上，为了实现按键模型 key_model 模块可以通过外部 key_press 的控制，即给 key_press 一个脉冲，ke_model 输出一次按键信号，这样在 key_model 模块中通过 key_press 上升沿的到来为条件去调用上面那个任务，具体代码如下。

```
initial begin
key_out = 1'b1;
while(1)
    begin
        @(posedge key_press);
        key_gen;
    end
end
```

设计好按键模型 key_model 模块后，对整个仿真 teshbench 的设计就相对比较简单了。添加并新建 key_filter_tb.v 文件，将按键消抖模块和按键模型例化在其中并产生时钟和复位激励信号，具体代码如下。仿真模拟产生了两次按键按

下和松开的动作。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module key_filter_tb;

    reg  clk;
    reg  reset_n;
    reg  key_press;
    wire key;

    wire key_flag;
    wire key_state;

    key_model key_model_inst(
        .key_press(key_press),
        .key_out(key)
    );

    key_filter key_filter_inst(
        .clk(clk),
        .reset_n(reset_n),

        .key_in(key),
        .key_flag(key_flag),
        .key_state(key_state)
    );

    initial clk= 1;
    always#(`CLK_PERIOD/2) clk = ~clk;

    initial begin
        reset_n = 1'b0;
        key_press = 1'b0;
        #(`CLK_PERIOD*10);
        reset_n = 1'b1;
        #(`CLK_PERIOD*10 + 1);

        key_press = 1'b1;
        #(`CLK_PERIOD);
        key_press = 1'b0;
        #60000000;

        key_press = 1'b1;
        #(`CLK_PERIOD);
        key_press = 1'b0;
        #60000000;
```

```
        $stop;
    end

endmodule
```

设计完仿真 testbech 后，打开 Modelsim 新建工程进行仿真，可以得到下方图 11-14 与下方图 11-15 所示的仿真波形图，按下稳定以及释放稳定均会产生一个时钟周期的 key_flag 高电平信号，key_state 也会正常变化。这里放大产生的抖动过程可以看到每一个抖动时间均不一样，这样就成功模拟了随机抖动过程。

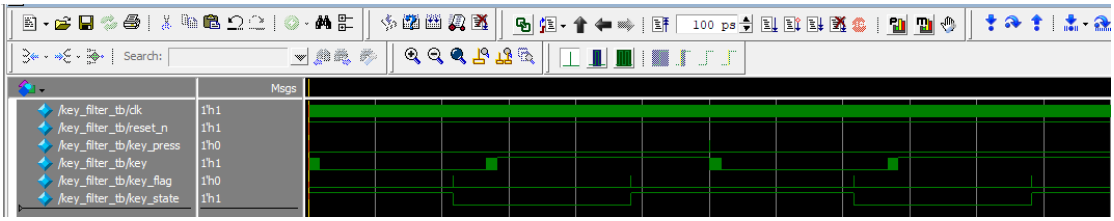


图 11-14 按键消抖模型仿真波形图

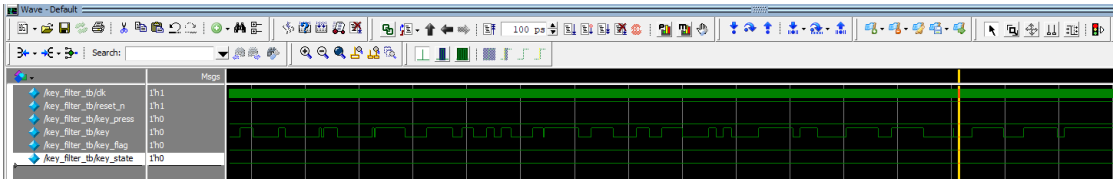


图 11-15 随机抖动过程

从以上的波形分析可以看出本章设计的独立按键消抖功能仿真正常。

11.5总结

本程序在全部学习章节中占有承上启下的作用，由于以后章节对本程序多采用直接引用的方式，所以本程序在此处未进行管脚约束，无法综合完成是正常现象。本程序使用仿真工具能得到正确波形即可完成本章设计任务。

本章在设计过程中复习了单 bit 异步信号同步化以及边沿检测和状态机相关设计。在仿真过程中引入了任务（task）以及随机函数（$random）并使用了仿真模型来简化激励文件的编写。具体的板级验证会在下一章中详细介绍。

12 模块化设计基础之加减法计数器

工程源码	|----02_设计实例 |----ch12_key_led
相关视频课程	
说明	

章节导读

在相对大一点的工程设计过程中，设计内容通常不会只写在一个设计文件中，而是会针对不同的功能设计出不同的子文件，最后在顶层文件中再进行例化调用。

在前一章节中设计并验证了独立按键的消抖，这里基于上一讲的按键消抖模块来实现一个加减法计数器，实现每次按下按键 0，4 个 LED 显示状态以二进制加法格式加 1，每次按下按键 1，4 个 LED 显示状态以二进制加法格式减 1，并以此学习模块化的设计方式。在第五章中将利用第四章编写好的独立模块，组合起来设计一些应用性较强的项目，进一步理解模块化设计方式的优点。

12.1模块功能划分

为了实现两个按键控制 LED 灯按照二进制计数方式的加减，可以将顶层模块 key_led_top 划分为两个按键消抖模块以及一个 LED 控制模块，其模块间的连线如图 12-1 所示。

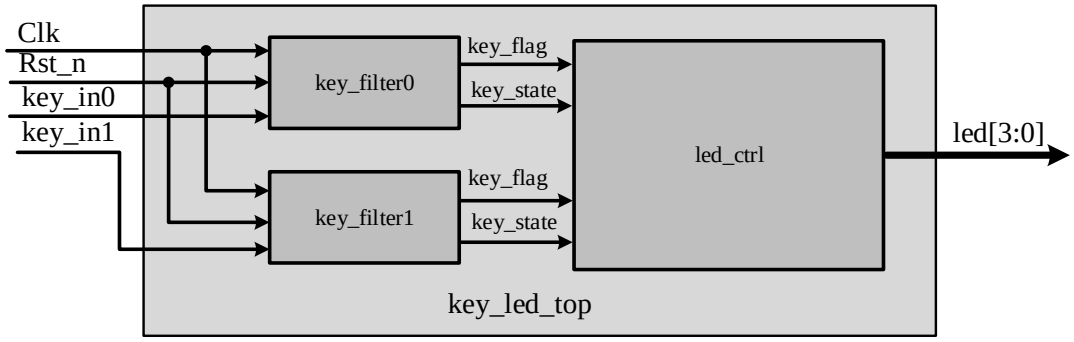


图 12-1 顶层模块端口图

12.1.1模块功能设计

这里的 key_filter 在前面章节中已经编写完成，只需在 key_led_top 中调用即可。现编写 led_ctrl，从下图中可得出其端口列表如下。

```
input Clk;
input Rst_n;
input key_flag0,key_flag1;
input key_state0,key_state1;

output [3:0]led;
```

本模块需要根据两个按键的状态来进行计数器的加减，且由上一节可以看出当 key_flag && key_state 为 1 时，即为按键按下。

```
reg [3:0]led_r;
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    led_r <= 4'b0000;
else if(key_flag0 && !key_state0)
    led_r <= led_r + 1'b1;
else if(key_flag1 && !key_state1)
    led_r <= led_r - 1'b1;
else
    led_r <= led_r;
```

计数器的 led_r 初值为 4`b0000，这里当按键 0 按下即计数器加一，计数器变为 4`b0001，由开发板上的 led 灯电路图可知，led 灯为高电平点亮，直接将 led_r 的值输出给到 led，便可以看到 led 灯的亮灭效果为暗暗暗亮。

```
assign led = led_r;
```

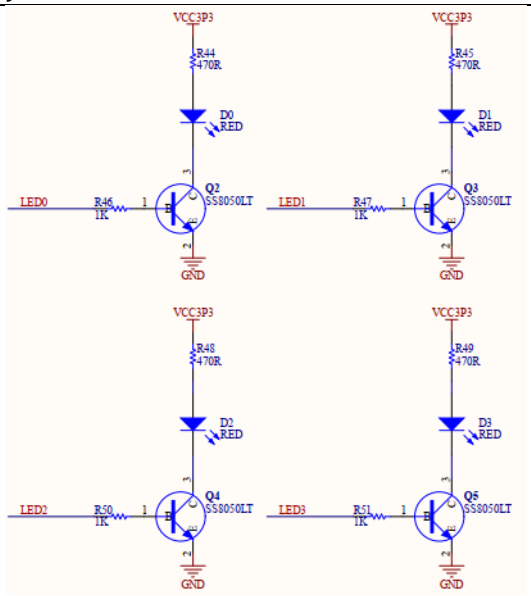


图 12-2 LED 灯电路图

这样各个独立的模块即编写完成，下面开始顶层文件的设计，在新建好的 key_led_top 中，例化三个模块并将内部信号类型设置为 wire。

```
module key_led_top(Clk,Rst_n,key_in0,key_in1,led);
```

```
    input Clk;
    input Rst_n;
    input key_in0;
    input key_in1;

    output [3:0]led;

    wire key_flag0,key_flag1;
    wire key_state0,key_state1;

    key_filter key_filter0(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .key_in(key_in0),
        .key_flag(key_flag0),
        .key_state(key_state0)
    );

    key_filter key_filter1(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .key_in(key_in1),
        .key_flag(key_flag1),
        .key_state(key_state1)
    );

    led_ctrl led_ctrl0(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .key_flag0(key_flag0),
        .key_flag1(key_flag1),
        .key_state0(key_state0),
        .key_state1(key_state1),
        .led(led)
    );

endmodule
```

进行分析和综合直至没有错误以及警告。

12.2激励创建及仿真测试

为了测试仿真编写测试激励文件，这里由于调用了两个按键进行仿真，因此需要将前一讲设计的按键仿真模型进行改写，加入使能信号 press，即 press 上升沿时就执行一次输出 key，其中任务 task press_key 部分是不变的。这里如果

不改写调用的两个仿真模型会同时执行，会导致出错。

新建 key_led_tb.v 文件输入以下内容，并保存到 testbench 文件夹下，再次进行分析和综合直至没有错误以及警告。以下内容除了生成了时钟以及复位信号，还模拟了按键 0 按下释放两次，以及按键 1 按下释放两次的过程。这样整体的代码就如下所示。

```
`timescale 1ns/1ns

`define clk_period 20

module key_led_tb;

    reg Clk;
    reg Rst_n;

    wire key_in0;
    wire key_in1;
    reg press0,press1;

    wire [3:0]led;

    key_led_top key_led_top(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .key_in0(key_in0),
        .key_in1(key_in1),
        .led(led)
    );

    key_model key_model0(
        .press(press0),
        .key(key_in0)
    );

    key_model key_model1(
        .press(press1),
        .key(key_in1)
    );

    initial Clk= 1;
    always#(`clk_period/2) Clk = ~Clk;

    initial begin
        Rst_n = 1'b0;
        press0 = 0;
        press1 = 0;
```

```
        #(`clk_period*10) Rst_n = 1'b1;
        #(`clk_period*10 + 1);

        press0 = 1;
        #(`clk_period*3)
        press0 = 0;

        #80_000_000;

        press0 = 1;
        #(`clk_period*3)
        press0 = 0;

        #80_000_000;

        press1 = 1;
        #(`clk_period*3)
        press1 = 0;

        #80_000_000;

        press1 = 1;
        #(`clk_period*3)
        press1 = 0;

        #80_000_000;
        $stop;
    end

endmodule
```

设置好仿真脚本后进行功能仿真，可以看到如图 12-3 所示的波形文件。每当按键 0 按下时，计数器 led_r 则会加 1，按键 1 按下后计数器 led_r 则会减 1。

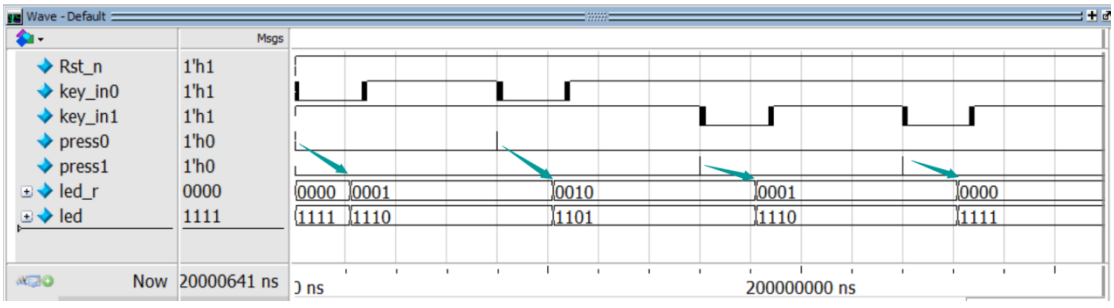


图 12-3 功能仿真波形图

至此，功能仿真验证工作就完成了。

12.3板级验证

本实验的板级验证环节，主要验证以下目标：

1. 能否正确烧写和下载程序。
2. 在高云开发板上，相应按键按下后，led 能否按设计意图点亮。

系统所需硬件如下：

1. 高云开发板。
2. 电源电缆一根
3. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台

12.3.1I/O 约束

我们需要点击 FloorPlanner 进行 I/O 约束，具体约束如下。

表 12-1 I/O 约束表

	功能信号	高云开发板对应引脚
基本管脚	clk	T9
	reset_n	C15
	key_in0	B16
	key_in1	A15
	led[0]	D14
	led[1]	C14
	led[2]	B9
	led[3]	A9

管脚绑定完成之后，如下图 12-4 所示。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	Clk	input		T9	4	False	LVCMOS33	OFF
2	Rst_n	input		C15	1	False	LVCMOS33	OFF
3	key_in0	input		B16	1	False	LVCMOS33	OFF
4	key_in1	input		A15	1	False	LVCMOS33	OFF
5	led[0]	output		D14	1	False	LVCMOS33	8
6	led[1]	output		C14	1	False	LVCMOS33	8
7	led[2]	output		B9	0	False	LVCMOS33	8
8	led[3]	output		A9	0	False	LVCMOS33	8

图 12-4 管脚绑定后效果图

12.3.2布局布线

重新点击“Place & Route”，报错消失，成功生成 Bit 数据流，如下图 12-5 所示。

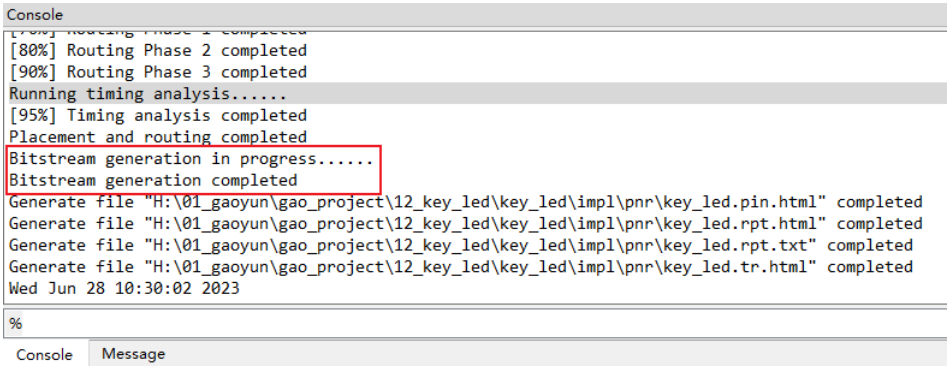


图 12-5 成功生成数据流

12.3.3下载数据流并观察现象

点击“Program Device”进入下载界面，弹出“Cable Setting”界面，如下图 12-6 所示，点击 Save。

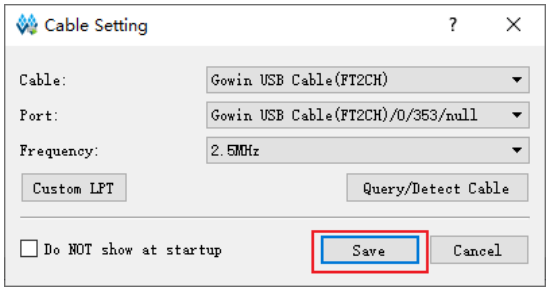


图 12-6 检测到器件示意图

然后点击  下载数据流，Programming 之后，进行如下图 12-7 按键测试后符合设计思想，即为设计无误。

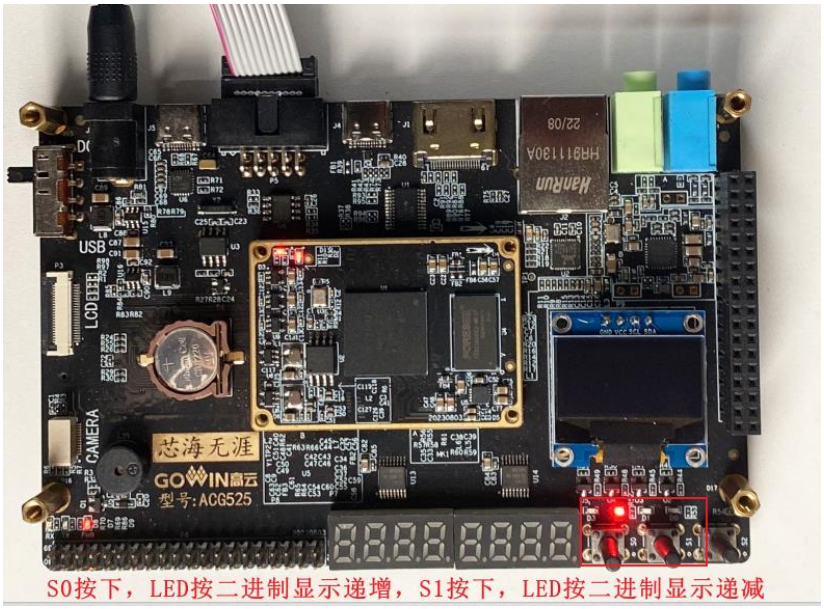


图 12-7 加减法计数器板级验证

看到与之前描述对应的现象，即为设计无误。至此，通过仿真以及板级验证说明该加减法计数器设计正确。

12.4常见问题说明

按键按下，计数器计数增减，必须要通过按键消抖进行处理，否则由于按键的机械特性，将会产生不确定的电平变化次数跳转。

12.5总结

本章通过模块化设计实现了加减法的设计并进行了行为仿真和板级验证。模块化的设计方式在后面章节中的使用尤为广泛，特别是在综合实验中，更能体会到这种设计方式的优点。建议读者能够跟随本实验内容，完整的进行整个实验。

13 8 位 7 段数码管驱动设计与验证

工程源码	|----02_设计实例 |----ch13_hex8_hc595
相关视频课程	
说明	

章节导读

电子系统中常用的显示设备有数码管、LCD 液晶以及 VGA 显示器等。其中数码管又可分为段式显示（7 段、米字型等）以及点阵显示（8*8、16*16 等），LCD 液晶的应用可以分为字符式液晶（1602、12864 等）以及真彩液晶屏，VGA 显示器一般是现在的电脑显示器。芯航线开发板对以上三种设备均提供了硬件接口。

本章将实现 FPGA 驱动数码管动态显示并提取出实现的电路结构，从电路结构入手编写代码，仿真对设计进行验证。本节课核心不再是代码，而是电路结构，电路结构确定后编写代码只是照图施工的过程。这也是越来越接近 FPGA 设计本质的硬件思维。

13.1数码管驱动原理

其中 8 段数码管的结构图如下图 13-1 所示。

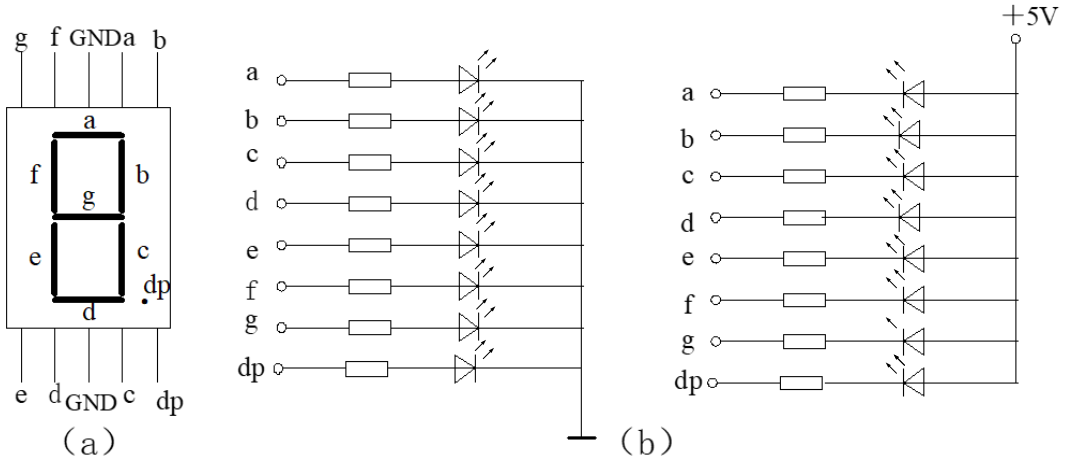


图 13-1 8 段数码管结构图

由上图可以看出数码管有两种结构：共阴极与共阳极。这两者的区别在于，公共端是连接到地还是高电平，对于共阴数码管需要给对应段以高电平才会使其点亮，而对于共阳极数码管则需要给低电平才会点亮。开发板上板载的是共

阳数码管。同时为了显示数字或字符，必须对数字或字符进行编码译码。这里先不考虑小数点也就是简化为 7 段数码管，其编码译码格式如下表所示：

表 13-1 数码管编码译码表

待显示内容 Data_disp	段码（二进制格式）								段码（十六进制格式）
	a	b	c	d	e	f	g	h	
0	0	0	0	0	0	0	1	1	8`hc0
1	1	0	0	1	1	1	1	1	8`hf9
2	0	0	1	0	0	1	0	1	8`ha4
3	0	0	0	0	1	1	0	1	8`hb0
4	1	0	0	1	1	0	0	1	8`h99
5	0	1	0	0	1	0	0	1	8`h92
6	0	1	0	0	0	0	0	1	8`h82
7	0	0	0	1	1	1	1	1	8`hf8
8	0	0	0	0	0	0	0	1	8`h80
9	0	0	0	0	1	0	0	1	8`h90
a	0	0	0	1	0	0	0	1	8`h88
b	1	1	0	0	0	0	0	1	8`h83
c	0	1	1	0	0	0	1	1	8`hc6
d	1	0	0	0	0	1	0	1	8`ha1
e	0	1	1	0	0	0	0	1	8`h86
f	0	1	1	1	0	0	0	1	8`h8e

段式数码管工作方式有两种：静态显示方式和动态显示方式。静态显示的特点是每个数码管的段选必须接一个 8 位数据线来保持显示的字形码。当送入一次字形码后，显示字形可一直保持，直到送入新字形码为止。这种方法由于每一个数码管均需要独立的数据线因此硬件电路比较复杂，成本较高，很少使用。

为了节约 IO 以及成本一般采用如下图 13-2 所示的电路结构，这样 3 个数码管接在一起就比静态的少了 7*2 个 I/O。

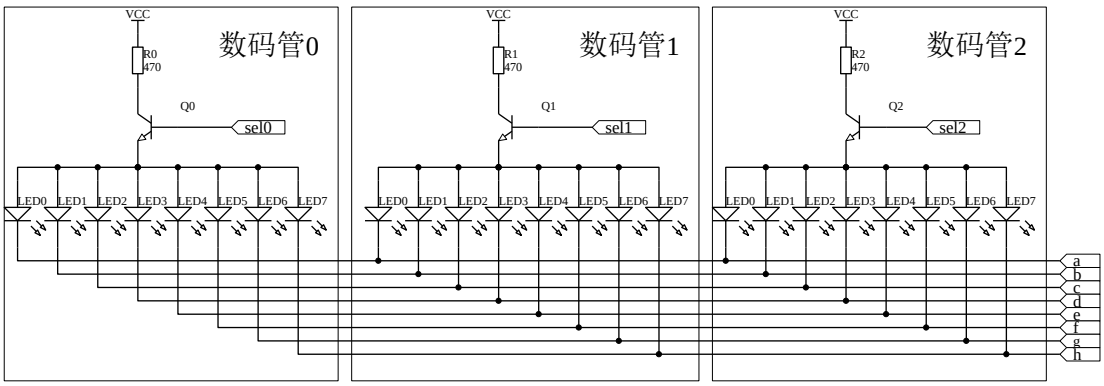


图 13-2 三位数码管等效电路图

这样就实现了另一种显示模式，动态显示。动态显示的特点是将所有位数码管的段选线并联在一起，由位选线控制是哪一位数码管有效。选亮数码管采

用动态扫描显示。所谓动态扫描显示即轮流向各位数码管送出字形码和相应的位选，利用发光管的余辉和人眼视觉暂留作用，使人的感觉好像各位数码管同时都在显示。

现在举例假设将扫描时间定为 1S，这三个数码管分成 3s，第 1 秒时 sel 数据线上为`b100，这时数码管 0 被选中，这时 a=0，数码管 0 的 LED0 就可以点亮；第 2 秒时 sel 数据线上为`b010，这时数码管 1 被选中，这时 b=0，数码管 1 的 LED1 就可以点亮；第 3 秒时 sel 数据线上为`b001，这时数码管 2 被选中，这时 c=0，数码管 2 的 LED2 就可以点亮。这时的效果就会是数码管 0 的 LED0 亮一秒后数码管 1 的 LED1 亮一秒最后是数码管 2 的 LED2 亮一秒，这样再次循环。

这样如果使用 1ms 刷新时间的话由于数码管的余辉效应以及人的视觉暂留这样就会出现数码管 0 的 LED0、数码管 1 的 LED1 以及数码管 2 的 LED2 “同时”亮，并不会有闪烁感。

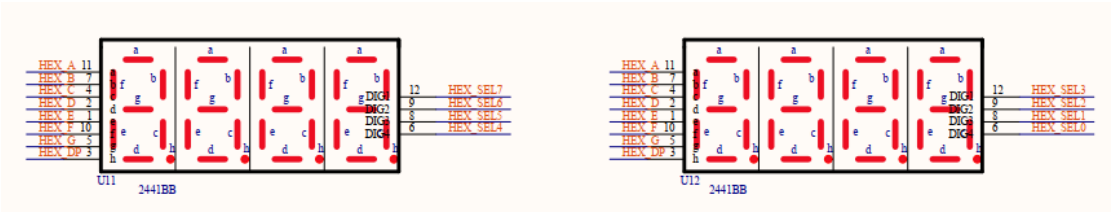


图 13-3 两个 7 段 4 位数码管级联原理图

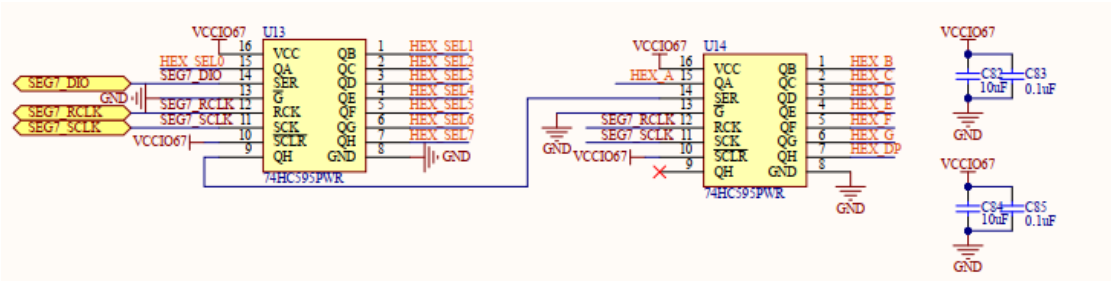


图 13-4 7 段 8 位数码管采用 74HC595 驱动方案原理图

13.2数码管动态扫描驱动设计

13.2.1模块接口设计及内部功能划分

由上面的分析可以得出如下图 13-5 的输入输出框图，其接口列表如下表所示：

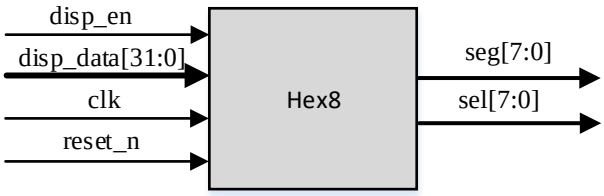


图 13-5 数码管模块框图

表 13-2 模块接口列表

信号名称	I/O	功能描述
clk	I	模块工作时钟，50MHz
reset_n	I	模块复位信号
disp_en	I	数码管使能信号 1 使能，0 关闭
disp_data[31:0]	I	8 个数码管待显示数据，每四位组成一个 BCD 码
sel[7:0]	O	数码管位选，选择当前要显示的数码管
seg[7:0]	O	数码管段选，当前要显示的内容

根据以上的分析可知，首先要有一个周期为 1ms 的驱动时钟，因此需要一个分频电路；在进行数码管的位选时，需要一个循环移位；在选择位后，需要选择器来选通数据输入位；要实现数码管编码译码表的功能需要一个译码器。

数码管驱动模块逻辑电路图可以简化成如下图 13-6 所示的，其中每一部分的作用如下表 13-3 所示。

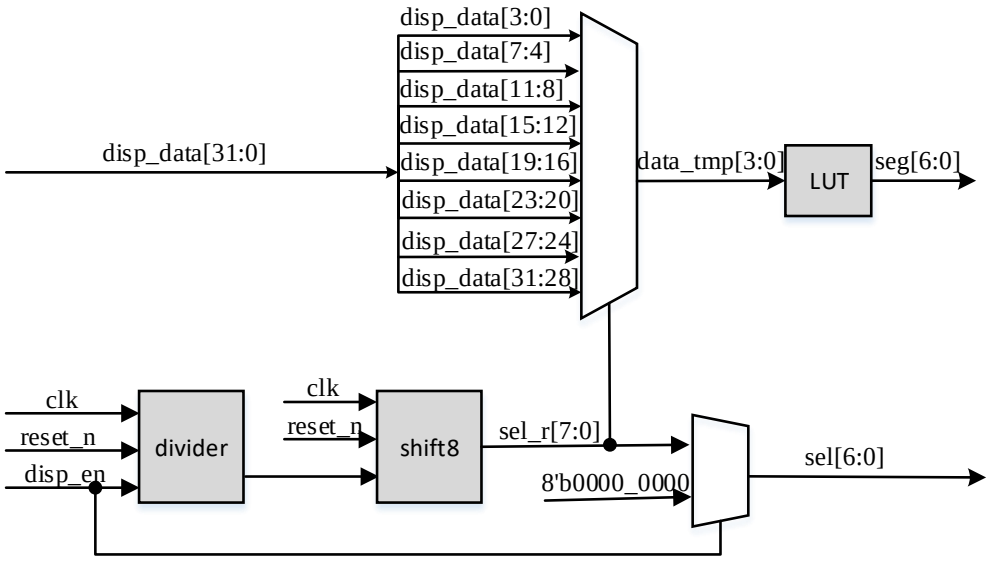


图 13-6 数码管驱动模块逻辑电路图

表 13-3 子模块功能描述

名称	功能描述
divder	分频产生 1KHz 的扫描时钟
shift8	8 位循环移位寄存器

MUX8	数据输入选择
MUX2	使能选择
LUT	数据译码器

13.2.2扫描时钟模块设计

从系统时钟 50M 分频得到 1KHz 的扫描时钟，总的计数器值为 25000d，从 0 开始计数，所以计数值最大值为 24999，这样计数器的位宽定义为 15 位即可。

```
always@(posedge clk or posedge reset)
if(reset)
    divider_cnt <= 15'd0;
else if(!en)
    divider_cnt <= 15'd0;
else if(divider_cnt == 24999)
    divider_cnt <= 15'd0;
else
    divider_cnt <= divider_cnt + 1'b1;

//1K 扫描时钟生成模块
always@(posedge clk or posedge reset)
if(reset)
    clk_1K <= 1'b0;
else if(divider_cnt == 24999)
    clk_1K <= ~clk_1K;
else
    clk_1K <= clk_1K;
```

13.2.3数码管位选模块设计

接下来编写 8 位循环移位寄存器，这里利用循环移位寄存器实现 0000_0001b→1000_0000b 的变化，进而实现数码管的位选，即实现每个扫描时钟周期选择一个数码管。移位寄存器输出值与数码管选通的对应关系如下表所示，其中 sel7 为高位。

表 13-4 移位寄存器与数码管对应关系

sel0	sel1	sel2	sel3	sel4	sel5	sel6	sel7	被选通数码管
1	0	0	0	0	0	0	0	数码管 0
0	1	0	0	0	0	0	0	数码管 1
0	0	1	0	0	0	0	0	数码管 2
……								
0	0	0	0	0	0	0	1	数码管 7

代码如下：

```
always@(posedge clk_1K or posedge reset)
if(reset)
```

```
    sel_r <= 8'b0000_0001;
else if(sel_r == 8'b1000_0000)
    sel_r <= 8'b0000_0001;
else
    sel_r <=  sel_r << 1;
```

13.2.4数码管数据显示设计

利用 8 选 1 多路器，选择端为当前扫描到的数码管也就是循环移位寄存器的输出端，利用多路器将待显示数据输送到对应到数码管上。

```
reg [3:0]data_tmp;//数据缓存
    always@(*)
        case(sel_r)
            8'b0000_0001:data_tmp = disp_data[3:0];
            8'b0000_0010:data_tmp = disp_data[7:4];
            8'b0000_0100:data_tmp = disp_data[11:8];
            8'b0000_1000:data_tmp = disp_data[15:12];
            8'b0001_0000:data_tmp = disp_data[19:16];
            8'b0010_0000:data_tmp = disp_data[23:20];
            8'b0100_0000:data_tmp = disp_data[27:24];
            8'b1000_0000:data_tmp = disp_data[31:28];
            default:data_tmp = 4'b0000;
        endcase
```

13.2.5显示数据译码设计

前面所说如果要使数码管显示数字或字符，须对数字或字符进行编码译码。这里利用一个 4 输入查找表，来实现 8 位的输出显示译码。

```
always@(*)
    case(data_tmp)
        4'h0:seg = 7'b1000000;
        4'h1:seg = 7'b1111001;
        4'h2:seg = 7'b0100100;
        4'h3:seg = 7'b0110000;
        4'h4:seg = 7'b0011001;
        4'h5:seg = 7'b0010010;
        4'h6:seg = 7'b0000010;
        4'h7:seg = 7'b1111000;
        4'h8:seg = 7'b0000000;
        4'h9:seg = 7'b0010000;
        4'ha:seg = 7'b0001000;
        4'hb:seg = 7'b0000011;
        4'hc:seg = 7'b1000110;
        4'hd:seg = 7'b0100001;
        4'he:seg = 7'b0000110;
```

```
        4'hf:seg = 7'b0001110;
    endcase
```

13.2.6模块使能设计

模块化的设计理念是，使得每个模块独立化，其端口设计要便于以后被调用与控制。基于这种理念，这里需要加入使能信号。关于使能子模块，直接利用一个二选一多路器即可实现。

```
assign sel = (en)?sel_r:8'b0000_0000;
```

13.374HC595 驱动模块设计

开发板上数码管设计用到了芯片 74HC595，该芯片的作用是移位寄存器，通过移位的方式，节省 FPGA 的管脚。FPGA 只需要输出 3 个管脚，即可达到发送数码管数据的目的，与传统段选、位选方式相比，大大节省了 IO 设计资源。74HC595 的具体使用方法和技术参数如下：在数据手册中可以看出，不同工作温度和工作电压下 74HC595 的芯片工作频率值不相同，分别如表 13-5 所示。由于在学习板中芯片采用 3.3V 供电，这样在设计其工作频率时，直接使用 50M 晶振四分频后的时钟作为其工作时钟，使其在 3.3V 状态下工作于 12.5M 频率。

表 13-5 74HC595 芯片设计频率参数下限

SYMBOL	PARAMETER	Vcc(V)	MIN	TYP	MAX	UNIT
f_{max}	maximum clock frequency SH_CP and ST_CP	2.0	4.8	-	-	MHz
		4.5	24	-	-	MHz
		6.0	28	-	-	MHz
T_{amb} = -40to + 125℃						

74HC595 的驱动模块端口如图 13-7 所示，其接口列表如表 13-6 所示。

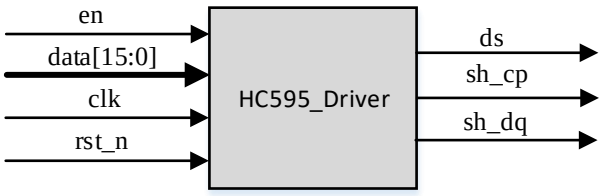


图 13-7 74HC595 驱动输入输出端口

表 13-6 74HC595 驱动输入输出端口功能描述

信号名称	I/O	功能描述
clk	I	50M 时钟
rst_n	I	复位信号

en	I	数码管使能信号 1 使能，0 关闭
data[15:0]	I	8 个数码管数码管的 SEG 和 SEL 数据
ds	O	串行数据输出
sh_cp	O	移位寄存器的时钟输出
st_cp	O	存储寄存器的时钟输出

在 74HC595 驱动模块设计中，首先设计工作时钟产生，对 50M 时钟进行 2 分频。

```
parameter CNT_MAX = 2;

reg [15:0]r_data;
always@(posedge clk)
if(s_en)
    r_data <= data;

reg [7:0]divider_cnt;//分频计数器;

always@(posedge clk or posedge reset)
if(reset)
    divider_cnt <= 0;
else if(divider_cnt == CNT_MAX - 1'b1)
    divider_cnt <= 0;
else
    divider_cnt <= divider_cnt + 1'b1;

wire sck_plus;
assign sck_plus = (divider_cnt == CNT_MAX - 1'b1);
```

对 sck_pluse 进行计数，用于查找表实现数据的串行输入以及移位时钟 sh_cp 与存储时钟 st_cp 的产生。

```
reg [5:0]SHCP_EDGE_CNT;

always@(posedge clk or posedge reset)
if(reset)
    SHCP_EDGE_CNT <= 0;
else if(sck_plus)begin
    if(SHCP_EDGE_CNT == 6'd32)
        SHCP_EDGE_CNT <= 0;
    else
        SHCP_EDGE_CNT <= SHCP_EDGE_CNT + 1'b1;
end
else
    SHCP_EDGE_CNT <= SHCP_EDGE_CNT;
```

查找表实现状态输出。

```
always@(posedge clk or posedge reset)
```

```
if(reset)begin
    st_cp <= 1'b0;
    ds <= 1'b0;
    sh_cp <= 1'd0;
end
else begin
    case(SHCP_EDGE_CNT)
        0: begin sh_cp <= 0; st_cp <= 1'd0;ds <= r_data[15];end
        1: begin sh_cp <= 1; st_cp <= 1'd0;end
        2: begin sh_cp <= 0; ds <= r_data[14];end
        3: begin sh_cp <= 1; end
        4: begin sh_cp <= 0; ds <= r_data[13];end
        5: begin sh_cp <= 1; end
        6: begin sh_cp <= 0; ds <= r_data[12];end
        7: begin sh_cp <= 1; end
        8: begin sh_cp <= 0; ds <= r_data[11];end
        9: begin sh_cp <= 1; end
        10: begin sh_cp <= 0; ds <= r_data[10];end
        11: begin sh_cp <= 1; end
        12: begin sh_cp <= 0; ds <= r_data[9];end
        13: begin sh_cp <= 1; end
        14: begin sh_cp <= 0; ds <= r_data[8];end
        15: begin sh_cp <= 1; end
        16: begin sh_cp <= 0; ds <= r_data[7];end
        17: begin sh_cp <= 1; end
        18: begin sh_cp <= 0; ds <= r_data[6];end
        19: begin sh_cp <= 1; end
        20: begin sh_cp <= 0; ds <= r_data[5];end
        21: begin sh_cp <= 1; end
        22: begin sh_cp <= 0; ds <= r_data[4];end
        23: begin sh_cp <= 1; end
        24: begin sh_cp <= 0; ds <= r_data[3];end
        25: begin sh_cp <= 1; end
        26: begin sh_cp <= 0; ds <= r_data[2];end
        27: begin sh_cp <= 1; end
        28: begin sh_cp <= 0; ds <= r_data[1];end
        29: begin sh_cp <= 1; end
        30: begin sh_cp <= 0; ds <= r_data[0];end
        31: begin sh_cp <= 1; end
        32: st_cp <= 1'd1;
        default:
            begin
                st_cp <= 1'b0;
                ds <= 1'b0;
                sh_cp <= 1'd0;
            end
    endcase
```

```
end
```

13.4数码管显示模块仿真测试

以下生成了复位信号以及使能信号、待显示数据的初始化以及切换，分别在数码管上显示“12345678”、“87654321”以及“89abcdef”。

```
initial begin
    reset_n = 1'b0;
    //en = 1;
    disp_data = 32'h12345678;
    #(`clk_period*20);
    reset_n = 1;
    #(`clk_period*20);
    #20000000;
    disp_data = 32'h87654321;
    #20000000;
    disp_data = 32'h89abcdef;
    #20000000;
    $stop;
end
```

设计好数码管驱动和对应的仿真文件之后，打开 Modelsim 新建一个仿真工程进行功能仿真，可以看到如下图 13-8 所示的仿真测试波形全局图，放大局部波形如下图 13-9 所示（切换成 2 进制显示更好观测），可以看出在复位信号置高之前数码管均显示 0，在复位结束后数码管才开始正常显示，且当待显示数据为 89ABCDEFh（MSB）后，数码管从 1 到 8 依次被选通且分别显示为 FEDCBA98h（LSB）。即仿真通过。

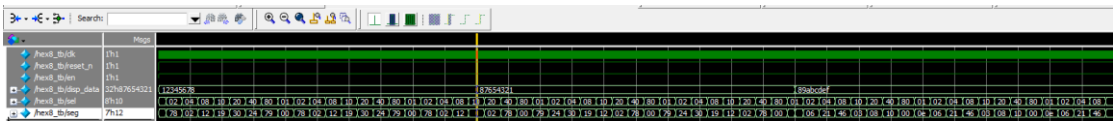


图 13-8 仿真测试波形全局图

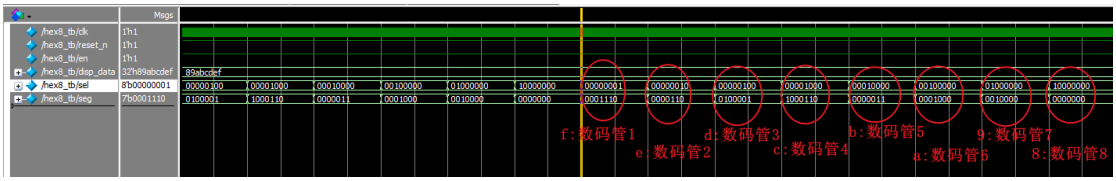


图 13-9 局部波形图

13.5板级调试与验证

添加并新建顶层文件 hex_data.v，将其设置为顶层设计，并将并将编写好的

数码管驱动模块例化到顶层文件中，使数码管显示“01234567”，代码如下所示：

```
module hex_top(
    clk,
    reset_n,
    sh_cp,
    st_cp,
    ds
);

    input clk;  //50M
    input reset_n;

    output sh_cp;
    output st_cp;
    output ds;

    wire [31:0]disp_data;
    wire [7:0] sel;//数码管位选（选择当前要显示的数码管）
    wire [6:0] seg;//数码管段选（当前要显示的内容)

    assign disp_data = 32'h01234567;

    hc595_driver hc595_driver(
        .clk(clk),
        .reset_n(reset_n),
        .data({1'd0,seg,sel}),
        .s_en(1'b1),
        .sh_cp(sh_cp),
        .st_cp(st_cp),
        .ds(ds)
    );

    hex8 hex8(
        .clk(clk),
        .reset_n(reset_n),
        .en(1'b1),
        .disp_data(disp_data),
        .sel(sel),
        .seg(seg)
    );

endmodule
```

对顶层进行分析和综合直至没有错误和警告。然后点击 FloorPlanner 进行 I/O 约束，具体的 I/O 约束如下图 13-10 所示。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	clk	input		T9	4	False	LVCMOS33
2	ds	output		H4	7	False	LVCMOS33
3	reset_n	input		B16	1	False	LVCMOS33
4	sh_cp	output		F4	7	False	LVCMOS33
5	st_cp	output		F3	7	False	LVCMOS33

图 13-10 I/O 约束图

然后点击“Place & Route”进行布局布线，没有错误的情况下，将会生成本次实验的 bit 数据流文件，点击 Program Device 进行下载，下载完成之后，数码管将会显示如下图 13-11。

图 13-11 数码管显示数据

从上图可以看出，每一个数字后面的“.”都被点亮了，这是因为顶层例化“hc595_driver”模块时，data 的最高位为 0，如果想要其不被点亮，将该位至 1 即可，此时数码管将会显示如下图 13-12。

图 13-12 数码管显示数据（“.”不被点亮）

至此完成了数码管的动态显示。

13.1常见问题说明

1. 在本实验中， 开发板采用的是板载 74HC595 三线驱动模式，使用该模式能够大大节约 IO 口资源的使用量。
2. 本案例在 HEX8 模块中，使用计数器分频，将 50MHZ 的时钟，分频生成 1KHz 的低频时钟，然后再使用该分频得到的时钟信号直接去驱动 D 触发器，这种时钟信号因为默认不走全局时钟路径，会有很大的时钟抖动和偏斜。该时钟在使用上还有另外一个缺点，即到达各个 D 触发器的延迟差别较大，会影响系统的稳定性。这种时钟信号称为门控时钟，后续在串口发送模块里，我们会以串口的波特率时钟信号为例，

为大家展示在 FPGA 系统中高质量的时钟信号设计和使用方法，（提前剧透下这时钟的名字叫使能时钟）。

13.2总结

本章中介绍了数码管有如下驱动方式：静态扫描、动态扫描和 SPI 驱动模式。得益于模块化设计的优势，如果需要实现 FPGA 直接驱动数码管，只需直接调用 hex8,驱动 hc595 文件即可，既达到了移位寄存的省管脚的功效，又保留了随时可以使用段选和位选并行输出的架构。

建议读者能够跟随本实验内容，完整的进行整个实验。

14 IP 核使用之 ROM

工程源码	|----02_设计实例 |----ch14_rom_ip
相关视频课程	
说明	

章节导读

本章将实现一组固定的数据（正弦波形表）存储在 FPGA 中使用 IP 核构建的 ROM 中，开发板上电后，系统开始从 ROM 中读出数据，并将数据直接通过并口输出。通过使用在线调试工具 gao 抓取并口上的数据，显示得到的正弦波形。

14.1ROM IP 的创建

新建 Gowin 工程，单击  进入 IP Core Generator 界面，在窗口的左上角“Filter”白色方框内输入“ROM”进行查找。可以看到在 Memory 下有两个和 ROM 有关的选项 pROM 和 ROM16，如下图 14-1 所示。

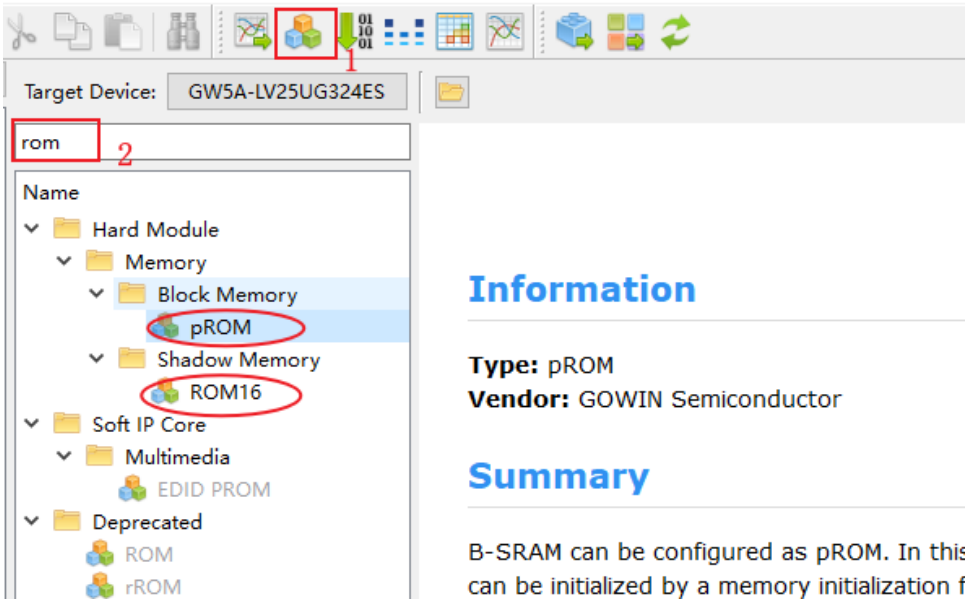


图 14-1 ROM IP 显示

下面将对上图所示的 ROM 相关的选项做一下简单介绍。

14.1.1pROM

pROM 为 16K 块状只读存储器，双击即可进入 pROM 配置界面，

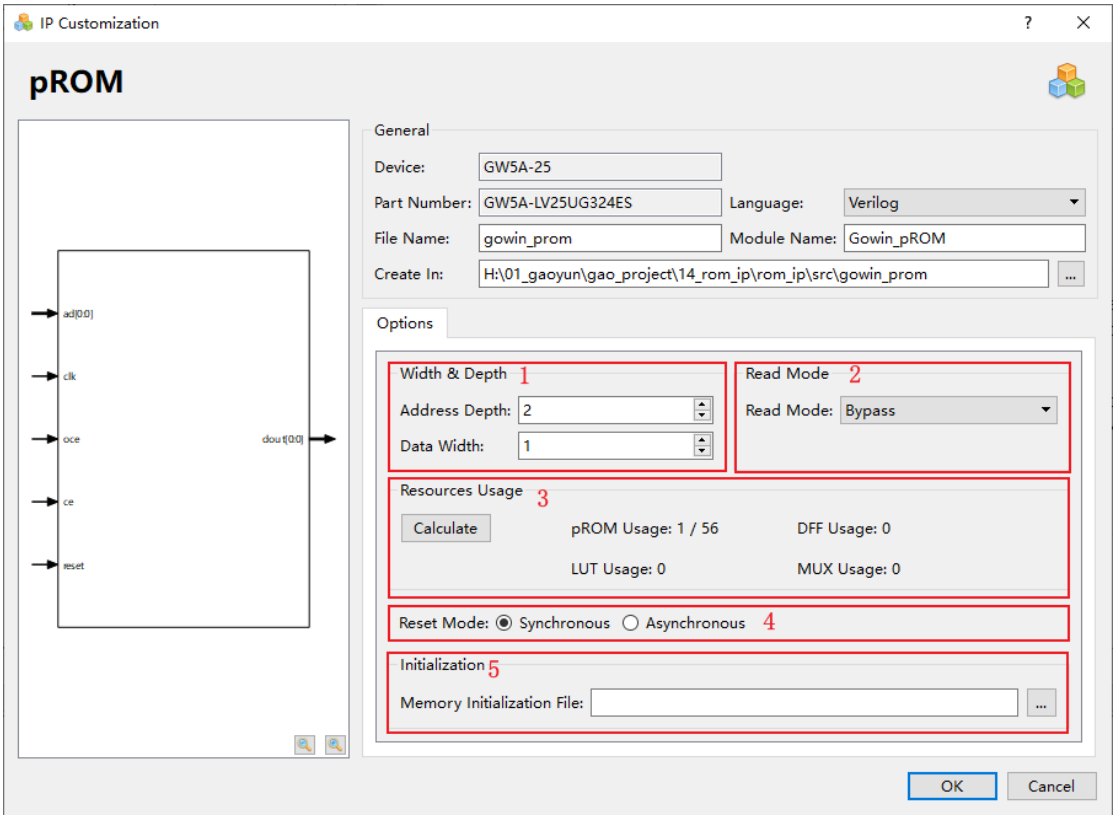


图 14-2 pROM 配置界面

下面对上图所示的几个部分进行简要介绍：

1. 数据深度和地址位宽配置，数据宽度和地址宽度的配置关系如下表 14-1 所示。

表 14-1 pROM 数据宽度和地址宽度配置关系

只读模式	BSRAM 容量	数据宽度	地址宽度
pROM	16 Kbits	1	14
		2	16
		4	12
		8	11
		16	10
		32	9

2. 读模式配置，可支持 2 种读模式：bypass 模式（旁路）和 pipeline 模式（寄存器），两种模式下的波形图如下图 14-3 和图 14-4 所示。

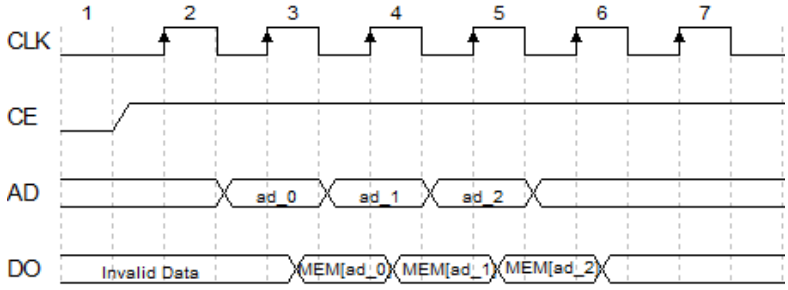


图 14-3 ROM 时序波形图（Bypass 模式）

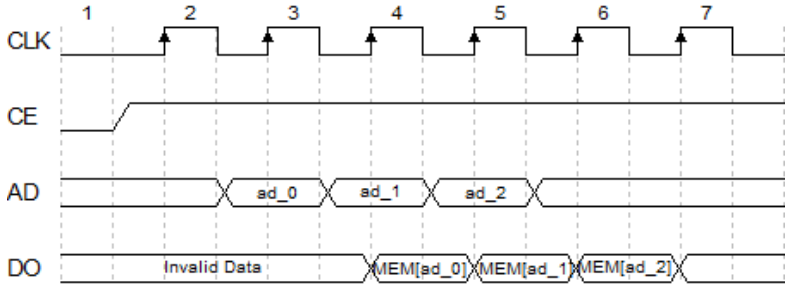


图 14-4 ROM 时序波形图（Pipeline 模式）

3. 资源利用情况，当配置完成的时候，可以点击 Calculate 计算 LUT、DFF 等利用情况。

4. 复位模式配置，支持同步复位（Synchronous）和异步复位（Asynchronous），reset 信号复位锁存器和输出寄存器，因此当设置 reset 信号有效时，不管用户使用的是寄存器输出模式还是旁路输出模式，端口都输出 0。同步复位有效时，DO 在 CLK 上升沿复位为 0，异步复位有效时，DO 随之复位为 0，不需要等到 CLK 上升沿。

5. 配置初始值，初始值以二进制、十六进制或带地址十六进制的格式写在初始化文中。“Memory Initialization File”选取的初始化文件可通过手写或者 IDE 菜单栏“File->New->Memory Initialization File”

14.1.2ROM16

ROM16 是数据位宽为 1 的只读存储器，由地址确定输出存储在 ROM 对应位置的数据。双击即可进行配置，其配置界面如下图 14-5 所示：

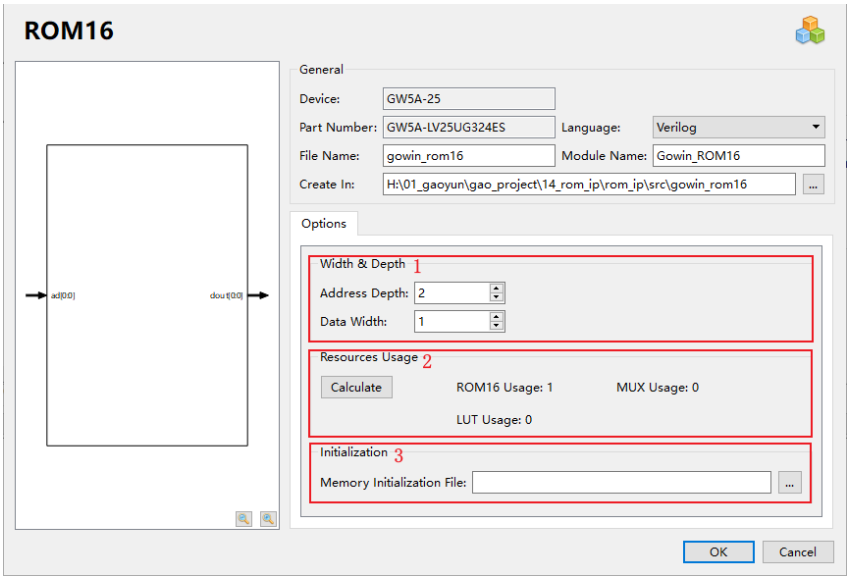


图 14-5 ROM16 配置界面

上图所示的 1、2、3 和图 14-2 所示的 1、3、5 功能是一样的，这里将不再继续进行说明，该 IP 只有两个信号，一个输，一个输出，其中输入 AD 为地址信号，DO 对应的是其地址对应的数据信号，其时序波形图如下图 14-6 所示。

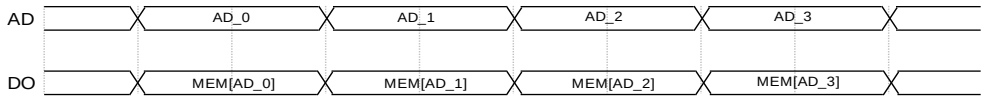


图 14-6 ROM16 时序波形图

本章实验将以 pROM 为例进行说明。

14.1.3配置 pROM

我们双击 pROM，进入 ROM 配置界面，设置数据位宽为 8，数据深度为 256 用于存储我们正弦波波形表，读模式设置为 Bypass，复位模式设置为 Synchronous。最后就需要添加初始化文件了，点击 Memory Initialization File 选项后面的... 进行添加，点击之后，可以看到需要添加的数据文件为 mi 格式的，其支持的文件格式有二进制格式（Bin File）、十六进制格式（Hex File）和带地址的十六进制格式（Address-Hex File）。下面我们将讲解十六进制格式的初始化文件配置。

1. 新建一个 mi 格式的文件

首先选择文件类型，依次点击 -> Memory Initialization File，操作如下所

示。

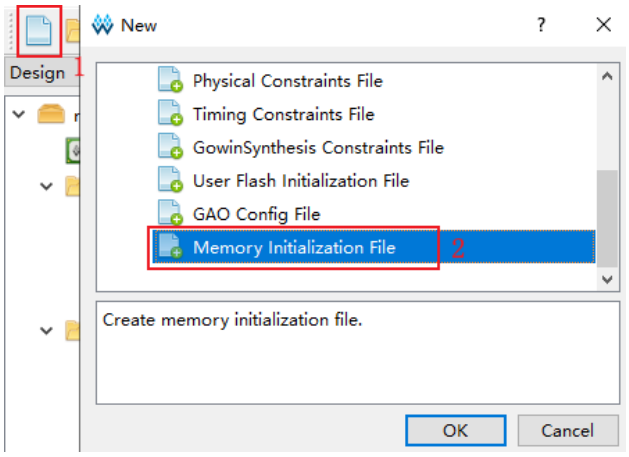


图 14-7 选择文件类型

弹出 New File 对话框，给文件命名为 sine_init，文件默认保存至工程目录的 src 文件夹下，然后点击 OK，如下所示。

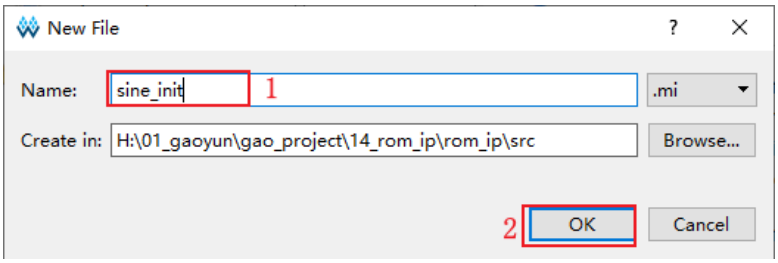


图 14-8 给.mi 文件命名

2. 配置文件格式

点击 OK 之后，会弹出 sine_init.mi 文件配置界面，将 File Format 选择为 HEX 格式，点击 Update 进行文件更新，如下图 14-9 所示。

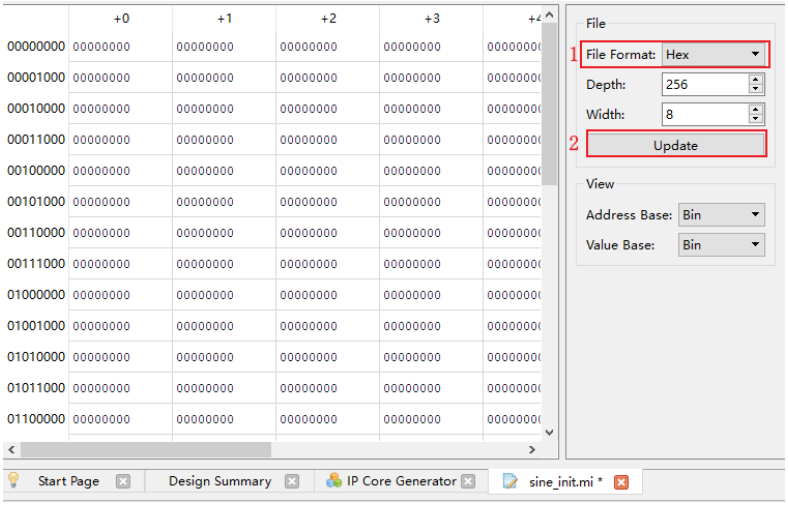


图 14-9 配置文件格式

3. Ctrl+S 保存文件。

此时我们使用 Notepad++打开 src 文件夹下的 sine_init.mi 文件，可以看到如下图 14-10 所示的内容。

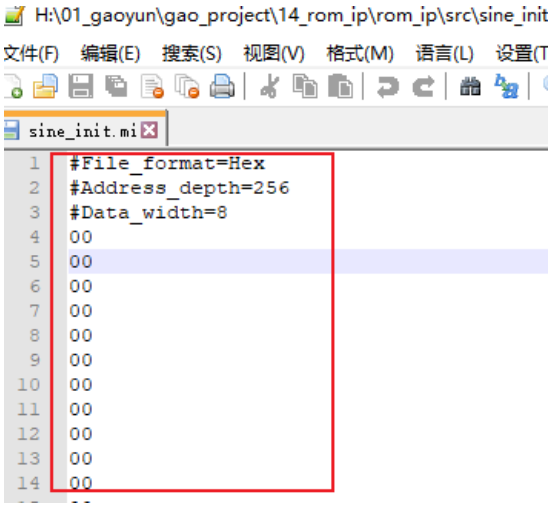


图 14-10 sine_init 文件内容

从上图可以看出文件的最开始显示的就是文件格式、地址深度以及数据位宽。

4. Mif 精灵生成数据文件

首先我们找到我们提供的 Mif 精灵（论坛搜索 mif），点击依次选择为 Altera，数据位宽设置为 8，Radix 设置为 hex，Depth 和 Maxi 设置为 256，type 设置为 sine，如下所示。

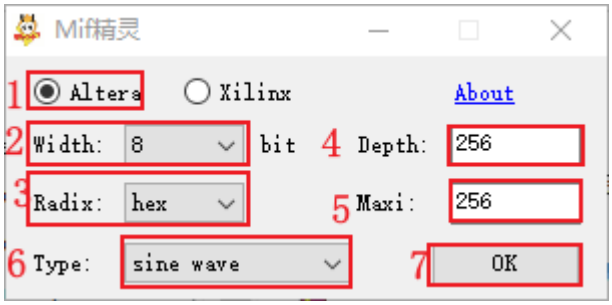


图 14-11 Mif 精灵配置界面

点击 OK 之后，将会在软件目录下，生成.coe 文件，右击使用 Notepad++打开，将 memory_initialization_vector 后面的数据进行复制替换 sine_init.mi 文件第 4 行之后的数据，如下所示。

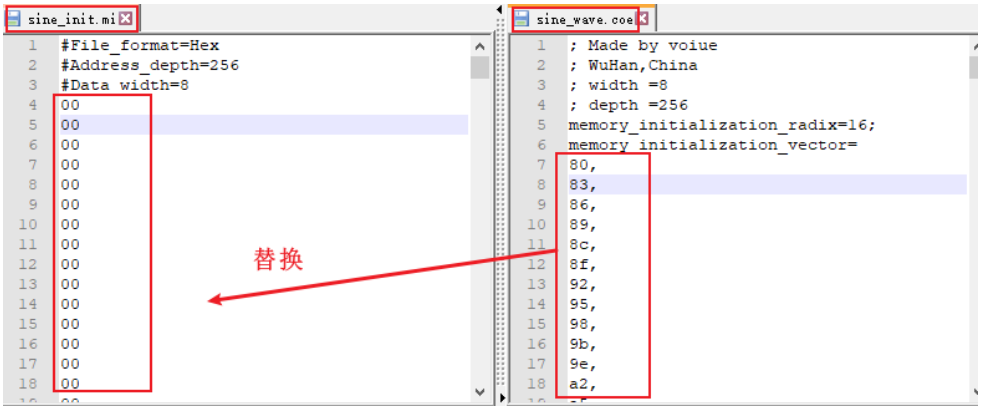


图 14-12 替换 sine_init.mi 中的数据内容

然后使用列编辑，将数据之后“,”和“;”删除，这样就得到了正弦波的.mi 文件。

然后点击 Memory Initialization File 一栏后面的[...]，选择我们刚刚生成的 sine_init.mi 文件，pROM 的最终配置界面如下图 14-13 所示。

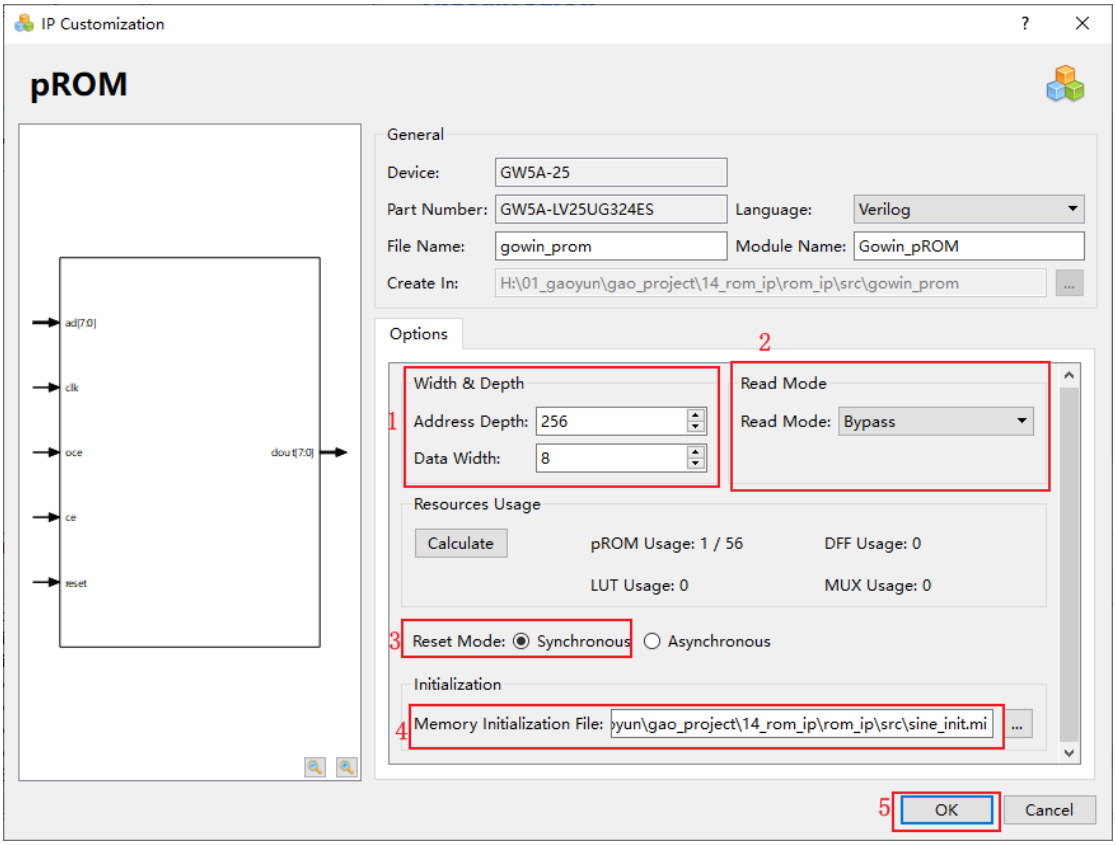


图 14-13 pROM 配置界面

此时我们点击 OK 之后，会提示如下所示的界面。

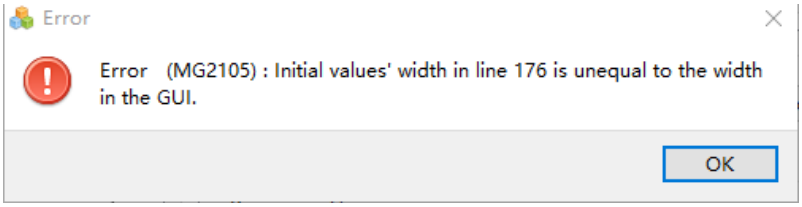


图 14-14 生成 pROM 报错

上图显示我们的初始化文件的 176 行数据位宽不匹配，打开我们的初始化文件 sine_init.mi 查看 176 行，可以看到 176 行显示的 f，是一个 4 位的数据，这里我们需要将其修改为一个 8 位的数据，也就是将高位补 0，所有的 4 位数据，都需要修改，操作如下所示。

图 14-15 修改 sine_init.m 文件

然后点击 OK，出现如下所示的提示框，表明 IP 生成成功，点击 OK 即可。

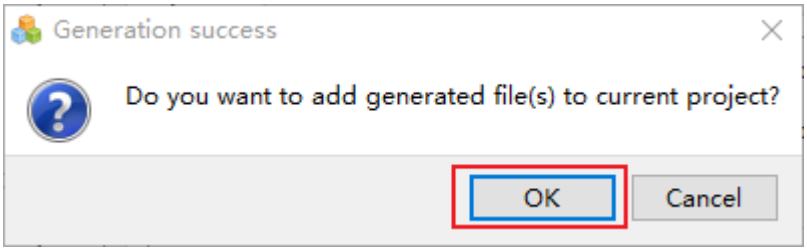


图 14-16 IP 生成成功提示框

点击 OK 之后，就会直接打开例化文件 gowin_prom_tmp.v，如下图 14-17 所示，这样我们在例化 IP 的时候，直接将其复制即可，如果想要更改 IP 中的配置，就点击  ，然后点击上方的 Target Device 后面的  ，选中 IP 对应的.ipc 文件，操作如下图 14-18 所示。

```
1  //Copyright (C)2014-2023 Gowin Semiconductor Corporation.
2  //All rights reserved.
3  //File Title: Template file for instantiation
4  //GOWIN Version: V1.9.9 Beta-1
5  //Part Number: GW5A-LV25UG324ES
6  //Device: GW5A-25
7  //Created Time: Thu Jun 29 16:02:28 2023
8
9  //Change the instance name and port connections to the signal
10 //--------Copy here to design--------
11
12     Gowin_pROM your_instance_name(
13         .dout(dout_o), //output [7:0] dout
14         .clk(clk_i), //input clk
15         .oce(oce_i), //input oce
16         .ce(ce_i), //input ce
17         .reset(reset_i), //input reset
18         .ad(ad_i) //input [7:0] ad
19     );
20
21 //--------Copy end-------------------
22
```

enerator　sine_init.mi　gowin_prom.v　gowin_prom_tmp.v

图 14-17 例化文件

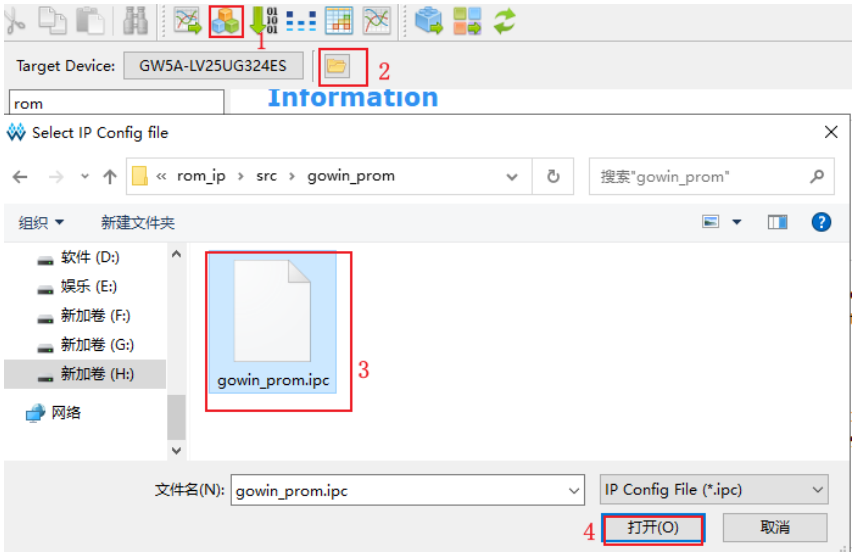


图 14-18 更改 IP 配置操作示意图

14.2激励创建及仿真测试

新建一个 rom_tb.v 的文件用于存放测试激励文件，保存至 src 文件的 tb 文件夹下。这里除了实现例化需要仿真的文件以及时钟创建，还实现了地址数从 0 自加到 2559d，但是由于本次实验配置的 pROM 的最大数据个数为 255d，因此在 address 第一次加满 255 后会重新从 0 开始自加，这样就有十个地址数循环。rom_tb.v 文件代码如下：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module rom_tb;

    reg clk;
    reg [7:0]addr;
    reg reset_n;

    integer i = 0;

    initial clk = 1;
    always #(`CLK_PERIOD/2) clk = ~clk;


    wire [7:0]dout;

    GSR GSR(.GSRI(1'b1));

   Gowin_pROM Gowin_pROM(
        .dout(dout), //output [7:0] dout
        .clk(clk), //input clk
        .oce(1'b1), //input oce
        .ce(1'b1), //input ce
        .reset(~reset_n), //input reset
        .ad(addr) //input [7:0] ad
    );

    initial begin
        reset_n = 0;
        #201
        reset_n = 1;
        addr = 0;
        #21;
        for(i=0;i<2560;i=i+1)begin
            #`CLK_PERIOD;
        addr = addr + 1'b1;
        end
        #(`CLK_PERIOD * 50);
        $stop;
    end
endmodule
```

需要注意的是，由于 pROM IP 中用到了全局变量 GSR，由于仿真不具有这个变量，所以需要在仿真文件中加上 GSR GSR(.GSRI(1'b1))这样一句话，不然仿真会报错。

然后打开 Modelsim 新建一个仿真工程，将 rom_tb 文件和 gowin_prom.v 文

件添加至工程中，可以看到如下所示的数据文件，发现数据发生变化了，但是不能直观看到正弦波。

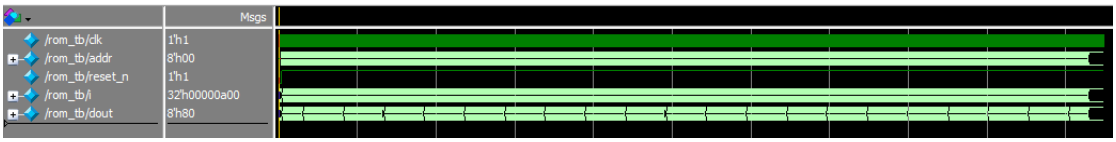


图 14-19 仿真输出波形图

可通过在信号 dout 右键依次选择 Format->Analog（automatic），如下图 14-20 所示。

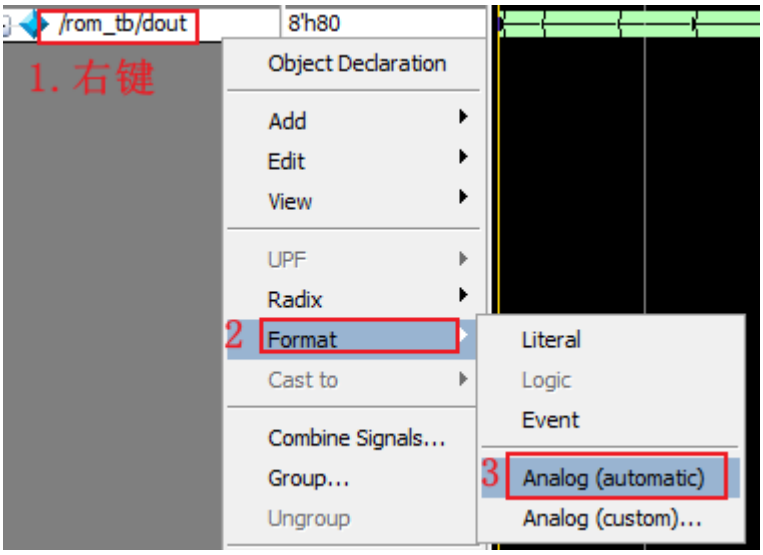


图 14-20 将波形转换为模拟量形式

然后还需要设置 Radix 为无符号的数据，如下图 14-21 所示。

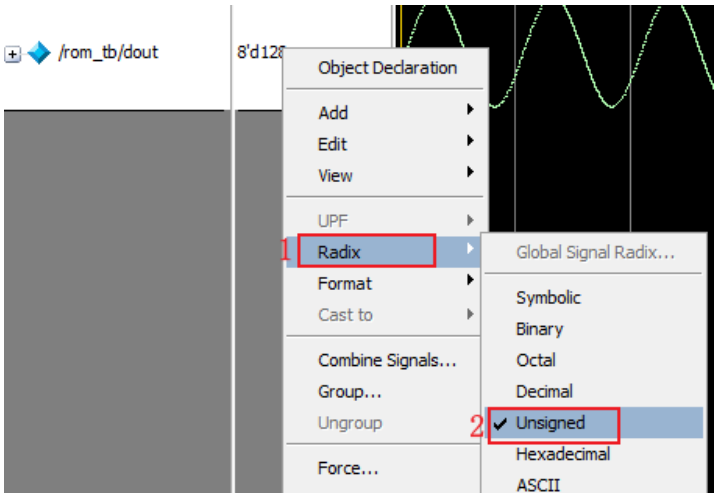


图 14-21 修改输出数据显示格式

设置完成之后，我们就可以看到如下图 14-22 所示的正弦波。说明我们仿真

正确。

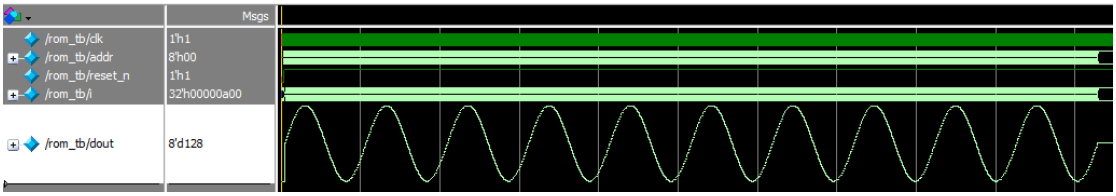


图 14-22 正弦波仿真波形

14.3板级验证

14.3.1创建顶层测试文件代码

添加并创建一个顶层文件并命名为 rom_top，例化 ROM IP，并使地址递增，rom_top.v 文件代码如下所示：

```
module rom_top(
    clk,
    reset_n,
    dout
);
    input clk;
    input reset_n;
    output wire [7:0]dout;

    reg [9:0]addr;
    always@(posedge clk or negedge reset_n)
    if(!reset_n)
        addr<=0;
    else
        addr<=addr+1'b1;

    Gowin_pROM Gowin_pROM(
        .dout(dout), //output [7:0] dout
        .clk(clk), //input clk
        .oce(1'b1), //input oce
        .ce(1'b1), //input ce
        .reset(~reset_n), //input reset
        .ad(addr) //input [7:0] ad
    );

    endmodule
```

然后进行分析综合直至没有错误。

14.3.2创建 GAO 文件

创建 gao 文件用于在线抓取波形，设置触发条件为 dout 不等于 0，如下图 14-23 所示。

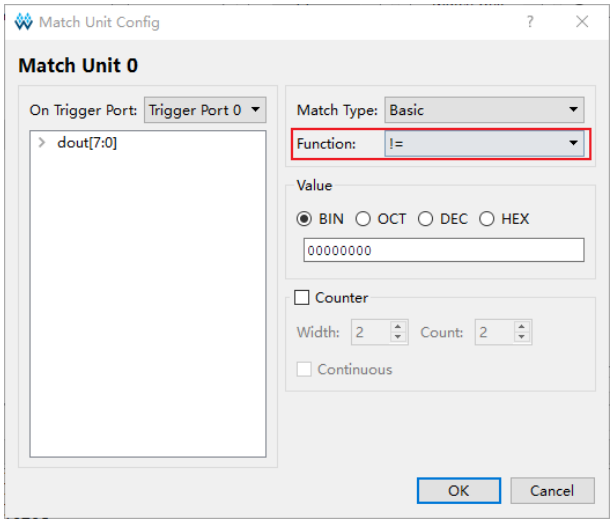


图 14-23 设置触发条件

采样时钟为 clk，捕获信号为 dout[7:0]，如下所示。

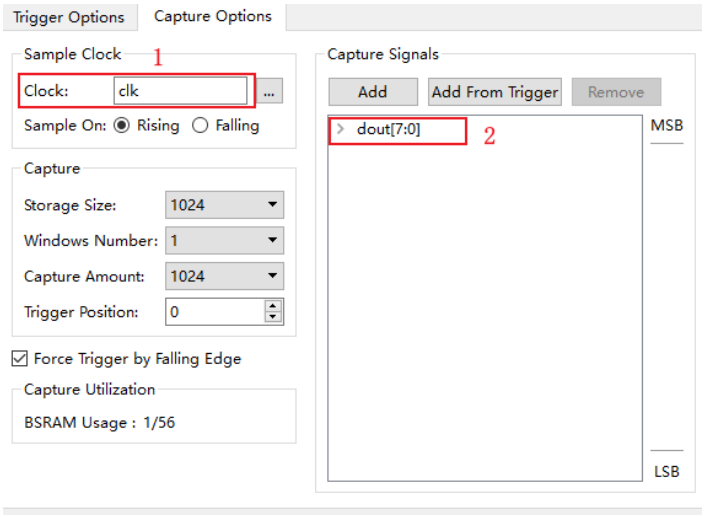


图 14-24 采样配置

其余操作参看高云在线逻辑分析仪一章的内容。

14.3.3I/O 约束

进入 FloorPlanner 界面进行 I/O 约束，这里只需要对 clk 与 reset_n 进行约束即可，如下图 14-25 所示。

Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
clk	input		T9	4	False	LVCMOS33
reset_n	input		B16	1	False	LVCMOS33

图 14-25 I/O 约束

14.3.4下载数据流

我们首先点击“Place & Route”布局布线没有错误之后，生成 bit 数据流文件，然后点击 Program Device 下载数据流文件，如下图 14-26 所示。

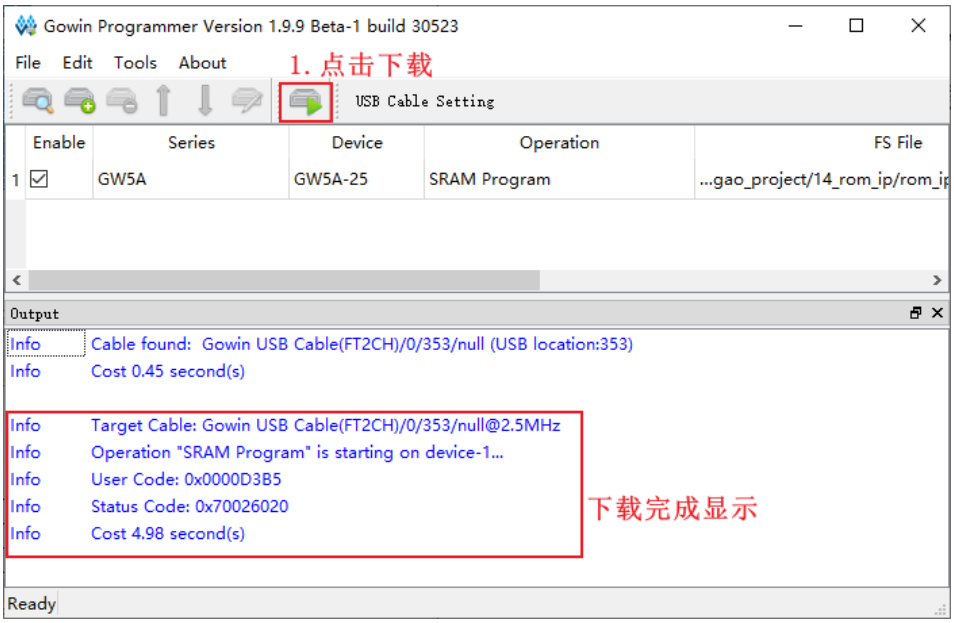


图 14-26 下载数据流文件

14.3.5在线逻辑分析仪抓取波形

点击  进入在线逻辑分析仪界面，然后点击  运行，抓取波形如下图 14-27 所示。

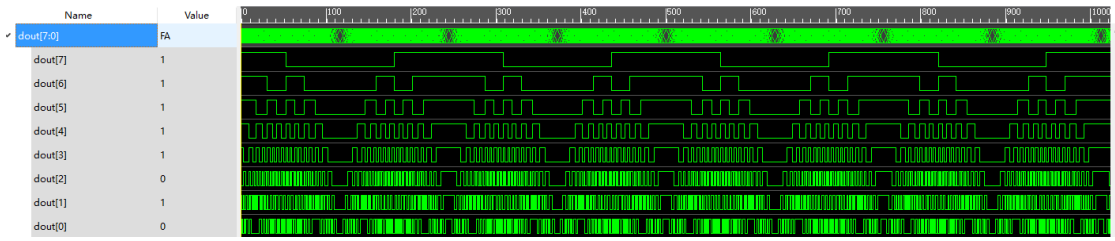


图 14-27 在线逻辑分析仪抓取波形图

从上图可以看出 dout[7:0]数据在发生变化，并且是周期性变化，但是高云的在线逻辑分析仪暂时没有将数据转换成模拟量进行分析的功能，这里我们只

能自己进行分析，比如我们截取一端从数据 00 变化开始观察，与我们的波形文件里的数据进行对比，如下图 14-28 所示，从图中可以看出数据一致，说明实验成功。

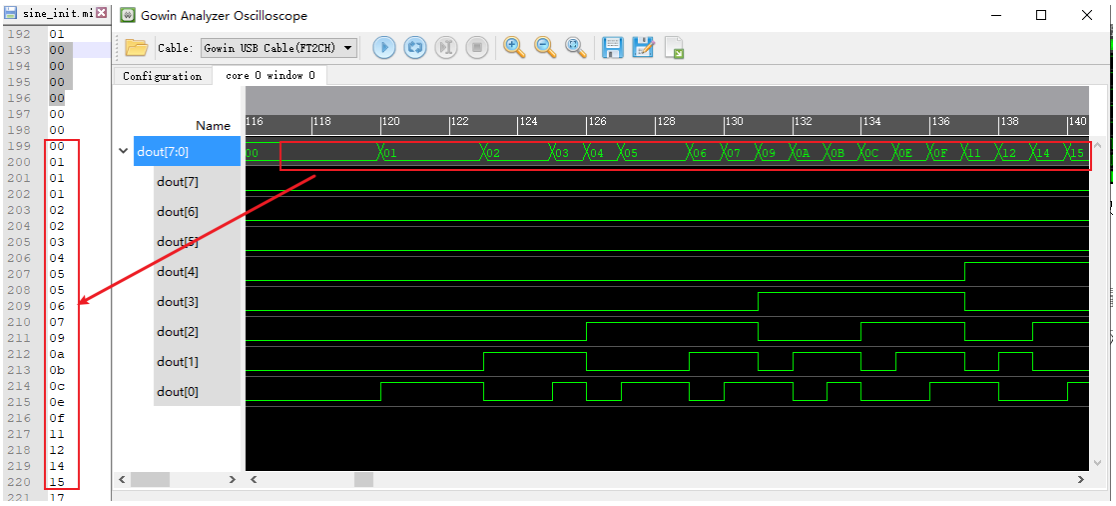


图 14-28 对比数据

14.4总结

本章实验带领大家学习如何创建 IP、修改 IP 配置，并通过向 ROM IP 中写入正弦波配置内容，最终通过仿真以及在线逻辑仪两种方式，最终输出正弦波波形数据，读者可以更换用于 ROM 初始化的 mi 文件，更改为其他波形的数据，可再次尝试用 mif 精灵生成三角波等其他波形，重新导入 IP 的 mi 文件，即可重复上面的操作。

15 IP 核使用之 RAM

工程源码	|----02_设计实例 |----ch15_ram_ip
相关视频课程	
说明	

章节导读

本节带领读者熟悉 RAM IP 的生成及相关设置，并将对其进行仿真验证。在此基础上，我们在后续的章节中，将搭建串口收发与存储双口 RAM 简易应用系统实现 PC 通过串口发送数据到 RAM 进行存储，按键控制 RAM 里数据通过串口发送到 PC 端显示。本章的内容，就是后续章节创建 RAM IP 核部分的一个理论铺垫。

15.1RAM IP 创建

新建 Gowin 工程，点击  进入 IP Core Generator 界面，在该界面下找到 Memory 一栏并将其展开，如下图 15-1 所示。

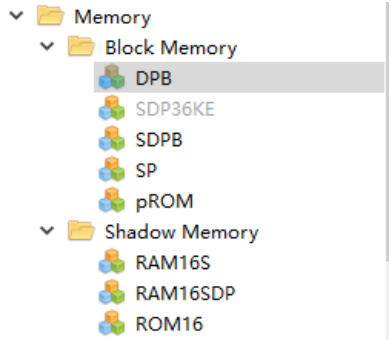


图 15-1 Memory IP

如上图所示，pROM 和 ROM16 在上一章节中已经做过介绍了。剩下的 DPB、SDPB、SP、RAM16S 和 RAM16SDP 都是 RAM 相关的 IP，其中 Block Memory 下的是块状静态随机存储器，具有静态存取功能，根据 BSRAM 的特性建立软件模型，可分为双端口模式（DPB）、伪双端口模式（SDPB）、单端口模式（SP）；Shadow Memory 下的是分布式静态存储器，可配置单端口模式（RAM16S）和伪双端口模式（RAM16SDP）。

本章将以 Shadow Memory 下的 RAM16SDP 模式为例进行说明。RAM16SDP 时地址深度为 16，数据位宽为 1 的伪双端口 SSRAM，具有两个地址，写地址

WAD 和读地址 RAD，这两个地址端口是异步的。WRE 为高电平时进行写操作，此时会在 CLK 的上升沿讲数据加载到存储器对应写地址。读操作则由读地址确定输出 RAM 对应位置的数据，其时序波形图如下所示。

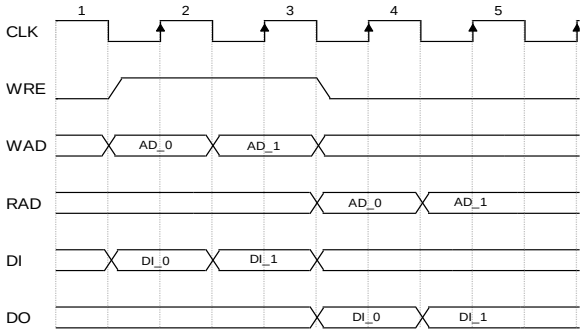


图 15-2 RAM16SDP 时序波形图

双击 RAM16SDP 进入配置界面，如下图 15-3 所示。

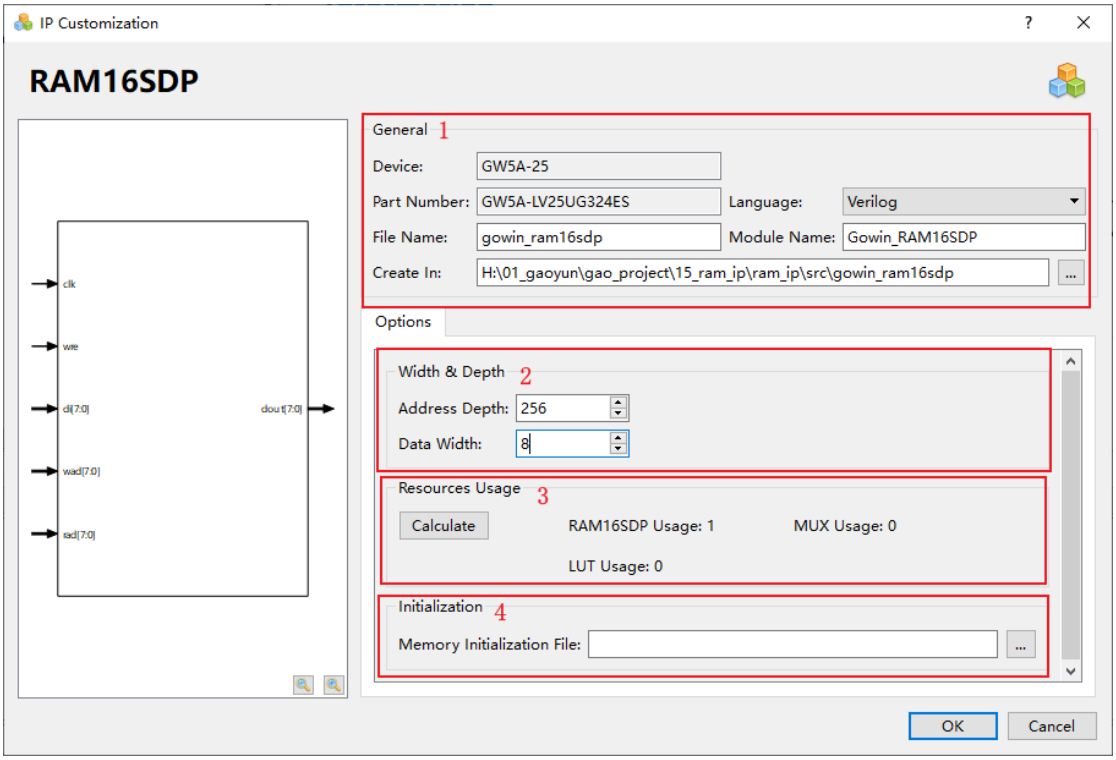


图 15-3 RAM16SDP 配置界面

1. General 配置框

General 配置框用于配置产生 IP 设计文件的相关信息。

- Device：显示已配置的 Device 信息。
- Part Number：显示已配置的 Part Number 信息。
- Language：配置产生的 IP 设计文件的硬件描述语言，选择右侧下拉列

表框，选择目标语言，支持 Verilog 和 VHDL。

- Module Name：配置产生的 IP 设计文件的 module name。在右侧文本框可重新编辑模块名称。Module Name 不能与原语名称相同，若相同，则报出 Error 提示。
- File Name：配置产生的 IP 设计文件的文件名。在右侧文本框可重新编辑文件名称。
- Create In：配置产生的 IP 设计文件的目标路径。可在右侧文本框中重新编辑目标路径，也可通过文本框右侧选择按钮选择目标路径。

2. 端口配置

配置选项：Data Width & Address Depth：配置地址深度（Address Depth）和数据宽度（Data Width）。当配置的地址深度和数据宽度无法通过单个模块时，IP Core 会实例化多个模块组合实现。

3. Resource Usage

计算并显示当前容量配置上占用的 Block Ram、 DFF、LUT、MUX 的资源情况。

4. 配置初始值

初始值以二进制、十六进制或带地址十六进制的格式写在初始化文件中，这个可以参考上一个章节的配置 pROM 相关内容，生成初始化需要的文件。

我们这里只配置端口 A 和端口 B 的地址深度和数据位宽，将地址深度设置为 256，数据位宽设置为 8，SDPB 的 IP 配置如下图 15-4 所示。

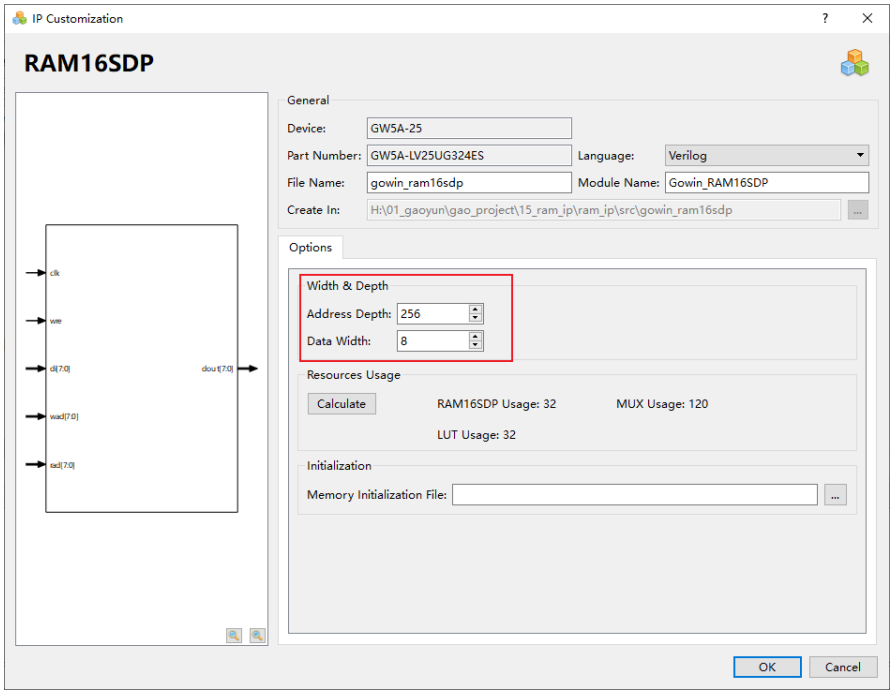


图 15-4 RAM16SDP 配置界面

点击 OK 之后，将会弹出 gowin_sdpb_tmp.v 文件，该文件夹放置的就是该 IP 的例化代码，如下所示，读者在例化的时候可以直接复制该内容进行修改。

```
Gowin_RAM16SDP your_instance_name(
    .dout(dout_o), //output [7:0] dout
    .wre(wre_i), //input wre
    .wad(wad_i), //input [7:0] wad
    .di(di_i), //input [7:0] di
    .rad(rad_i), //input [7:0] rad
    .clk(clk_i) //input clk
);

//--------Copy end-------------------
```

图 15-5 RAM16SDP 例化代码

15.2激励创建与仿真验证

为了测试简单伪双端口 RAM16SDP，可以通过实际写入一些数据再读取部分数据的方式来验证 RAM16SDP 读写是否正常。添加并新建 tb 文件命名为 ram_sdpd_tb.v。编写 tb 代码，具体 tb 代码实现的是在地址从 0~16 上写入数据为从 255 减至 240。延时一段时间后读地址为 0~16 上的数据。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20
```

```
module ram_tb();
    reg clk;
    reg wea;
    reg [7:0]addra;
    reg [7:0]dina;
    reg clkb;
    reg [7:0]addrb;
    wire[7:0]doutb;
    integer i;

    GSR GSR(.GSRI(1'b1));

    Gowin_RAM16SDP your_instance_name(
        .dout(doutb), //output [7:0] dout
        .wre(wea), //input wre
        .wad(addra), //input [7:0] wad
        .di(dina), //input [7:0] di
        .rad(addrb), //input [7:0] rad
        .clk(clk) //input clk
    );
    initial clk = 1'b1;
    always #(`CLK_PERIOD/2) clk = ~clk;


initial begin
    wea=0;
    addra=0;
    dina=0;
    addrb=255;

    #(`CLK_PERIOD*10 +1 );
    wea=1;
    for (i=0;i<=15;i=i+1)begin
        dina=255-i;
        addra = i;
    #`CLK_PERIOD;
    end
        wea=0;
    #1;
    for (i=0;i<=15;i=i+1)begin
        addrb=i;
        #`CLK_PERIOD;
    end
    #200;
    $stop;
end
```

```
endmodule
```

编写好 tb 文件后，新建 modelsim 工程进行功能仿真，可以看到如下图 15-6 所示的波形。

图 15-6 工程仿真波形全貌

对写数据部分进行放大后的波形如下图 15-7 所示。

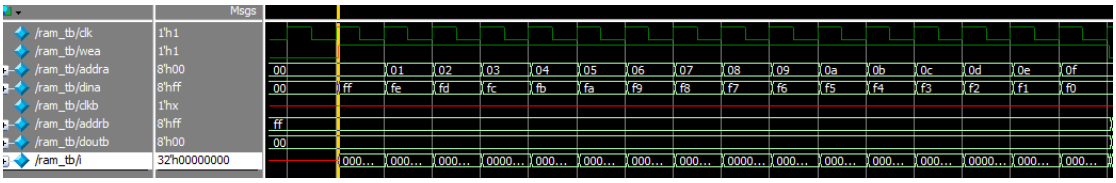


图 15-7 数据写入时放大后波形

在写入使能有效后，上升沿到来之后地址 0 写如 0xff，下一个上升沿在地址 1 处写入数据 0xfe，以此类推。可以看出写入数据正常。

放大读取部分后的波形如下图 15-8 所示。

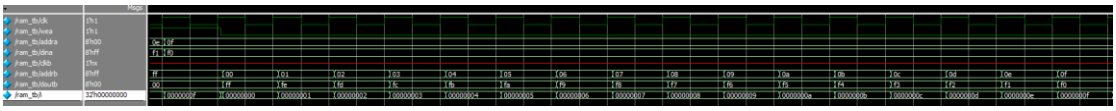


图 15-8 读数据波形文件

给出地址 00 之后，讲地址 00 写入的数据 0xff 读出，仿真波形和时序波形图 15-2 一致，说明 RAM16SDP 工作正常。

15.3总结

本章通过调用 IP 核的方式实现了一个伪双端口的 RAM16SDP IP。本章属于纯软件界面和理论学习，并通过仿真观察实验现象，没有上板调试任务。下一讲将以此为基础，进行串口收发、按键以及双口 RAM 组成的简易系统来进行更为系统化的板级测试。

16 搭建串口收发与存储双口 RAM 简易应用系统

工程源码	|----02_设计实例 |----ch16_uart_ram
相关视频课程	
说明	

章节导读

本节将以模块化设计为基础，利用已编写的串口收发模块、按键模块以及 RAM 的 IP 模块来设计一个简易应用系统，再次加深模块化设计的思想。

16.1系统模块功能划分及接口设计

为了实现通过串口发送数据到 FPGA 中，FPGA 接收到数据后将数据存储在伪双端口 RAM16SDP 中，当需要时，按下按键 S0，则 FPGA 将 RAM 中存储的数据通过串口发送出去。先进行功能划分：串口接收模块、按键消抖模块、RAM 模块、串口发送模块以及控制模块。前几讲除了控制模块均已经详细介绍，得益于当时的设计，这里就不用再次编写这些模块，可以直接在本工程下进行调用，整个设计的模块框图如下图 16-1 所示。

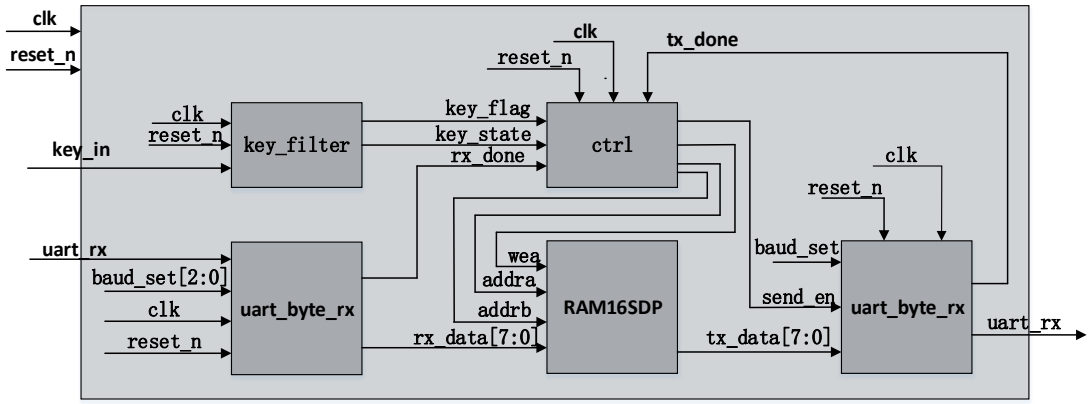


图 16-1 系统模块划分

16.1.1接口设计

新建工程后，先将已编写好的模块设计文件添加进工程中，并新建一个以名为 uart_ram.v 的设计文件保存在默认文件下，并将该文件设置为顶层文件。

根据前面设计的系统结构图用 Top-Down 的设计方式就可以先把顶层文件写出。结构图左边的端口为 input 类型，右边的为 output 类型，内部连线均为 wire 型。可以看出在进行一个系统设计时，良好的模块划分以及设计的重要性。然

后例化各个模块，这里将波特率设置为 9600bps。

表 16-1 模块接口列表

信号名称	I/O	功能描述
clk	I	系统时钟 50Mhz
reset_n	I	系统复位信号，低电平有效
uart_rx	I	串行数据输入
key_in	I	按键输入
uart_tx	O	串行数据输出

16.1.2控制模块设计

现在编写本系统的控制模块 ctrl.v，模块的接口可参照系统结构图写出。为了实现 FPGA 接收到数据后将数据存储在双口 ram 的一段连续空间中，这样就需要设计一个写地址自加的控制部分，且其控制写使能信号为串口接收模块输出的 rx_done 信号。每来一个 rx_done 也就是每接收成功一字节数，地址数进行加一。

```
assign wea = rx_done;

always@(posedge clk or posedge reset)
if(reset)
    addra <= 8'd0;
else if(rx_done)
    addra <= addra + 1'b1;
else
    addra <= addra;
```

为了实现当按下按键 0，FPGA 将 RAM 中存储的数据通过串口发送出去。也就是实现按键按下即启动连续读操作，再次按下即可暂停读操作。

```
always@(posedge clk or posedge reset)
if(reset)
    send_state <= 1'b0;
else if(key_flag && !key_state)
    send_state <= ~send_state;
else
    send_state <= send_state;

always@(posedge clk or posedge reset)
if(reset)
    addrb <= 8'd0;
else if(tx_done && send_state == 1'b1)
    addrb <= addrb + 8'd1;
else
    addrb <= addrb;
```

在切换到数据发送状态后，第一个数据的发送使能就根据按键产生，之后的发送使能就根据串口发送完成信号产生。

```
assign send_en_pre = send_1st_en | (tx_done && send_state == 1'b1);
always@(posedge clk or posedge reset)
if(reset)
    send_en <= 1'b0;
else
    send_en <= send_en_pre;
```

16.2激励创建及仿真测试

为了测试仿真编写测试激励文件，这里由于使用了按键消抖模块，因此必须将之前创建的 key_modle.v 仿真模块加入到工程中。添加并创建仿真文件 uart_ram_tb.v。除了例化顶层模块 uart_ram，还需要例化 key_model 和串口发送模块 uart_byte_tx 作为产生激励的模块。仿真激励代码如下所示：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module uart_ram_tb();

    reg clk;
    reg reset_n;
    reg key_press;
    reg [7:0]tx_data;
    reg send_en;

    wire key_in;
    wire uart_tx;
    wire tx_done;

    initial clk = 1;
    always#(`CLK_PERIOD/2) clk = ~clk;

    initial begin
        reset_n   = 1'b0;
        key_press = 1'b0;
        tx_data   = 8'h00;
        send_en   = 1'b0;
        #(`CLK_PERIOD*20);
        reset_n = 1;

    //1st byte
        tx_data = 8'haa;
        send_en = 1'b1;
```

```
        #(`CLK_PERIOD);
        send_en = 1'b0;

        @(posedge tx_done);
        #(`CLK_PERIOD*5);

    //2nd byte
        tx_data = 8'hbb;
        send_en = 1'b1;
        #(`CLK_PERIOD);
        send_en = 1'b0;

        @(posedge tx_done);
        #(`CLK_PERIOD*5);

    //3rd byte
        tx_data = 8'h55;
        send_en = 1'b1;
        #(`CLK_PERIOD);
        send_en = 1'b0;

        @(posedge tx_done);
        #(`CLK_PERIOD*5);

    //change state
        key_press = 1'b1;
        #(`CLK_PERIOD);
        key_press = 1'b0;

        #35000000;
        $stop;
    end

    key_model key_model_inst(
        .key_press(key_press),
        .key_out(key_in)
    );

    uart_byte_tx uart_byte_tx_inst(
        .clk(clk),
        .reset_n(reset_n),

        .data_byte(tx_data),
        .send_en(send_en),
        .baud_set(3'd0),

        .uart_tx(uart_tx),
```

```
        .tx_done(tx_done),
        .uart_state()
    );

    uart_ram uart_ram_inist(
        .clk(clk),
        .reset_n(reset_n),

        .key_in(key_in),
        .uart_rx(uart_tx),
        .uart_tx()
    );
endmodule
```

然后新建 Modelsim 工程，将本次实验所需的文件进行添加，然后将 uart_ram 模块的所有端口信号添加进行观察，波形图如下图 16-2 所示，可以从波形上看到，每当写入地址加一时数据均可以有效的写入，按键按下后每当一次输出结束后读地址也进行加一，实现数据输出。

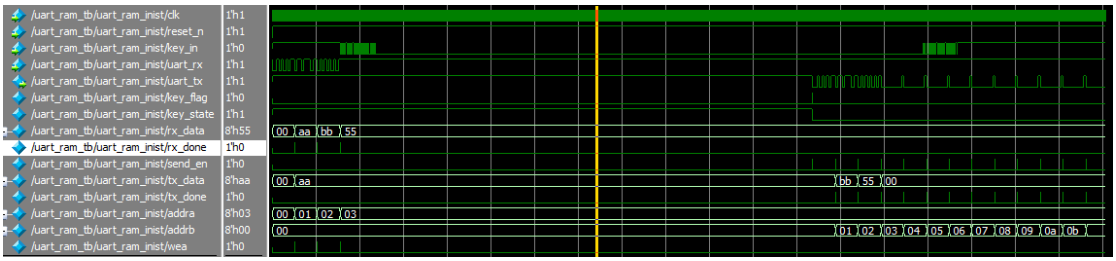


图 16-2 功能仿真波形文件

放大波形文件的数据接收和写入部分，如下所示。可以看出在第一次给出写使能信号，也就是串口接收完成产生 rx_done 信号之后，将数据 aa 写入地址 0，写入成功后写入地址加一。从图中还可以看出，一共产生了三次 rx_done 信号，分别向地址 0 中写入 aa，地址 1 中写入 bb，地址 2 中写入 55。

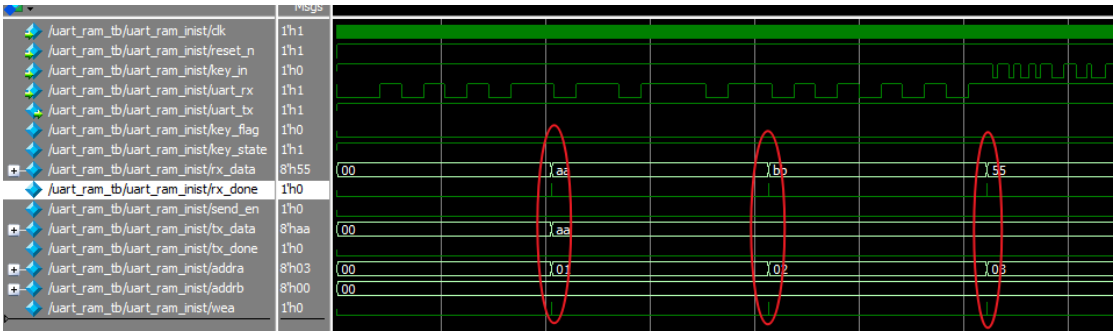


图 16-3 数据接收及写入部分波形

完整的数据读取及发送部分波形如下图 16-4 所示，从图中可以看出每当 send_en 有效后均会进行一字节数据的发送。传输完地址 2 中的数据后，后续将

会输出 0，这是由于这里将 RAM 定为 256 宽度，只有前 3 个写入了数据，后面地址空间没有写入数据，即输出全为 0。

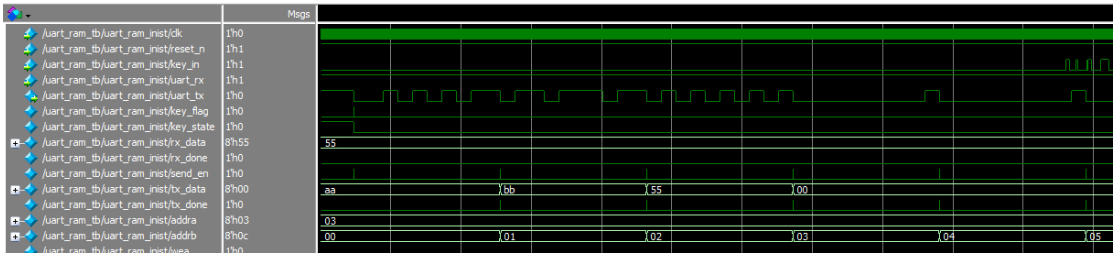


图 16-4 完整数据读取及发送部分波形

通过上述对仿真波形的分析，可以看出符合我们预期想要实现的功能，接下来就进行板级验证。

16.3板级验证

本次实验的板级验证环节，主要验证以下几个目标：

3. 能否正确将生成的数据流文件下载至开发板。
4. 打开电脑上的串口调试助手，输入数据，然后按下按键，串口助手源源不断的收到数据，再次按下按键，串口助手接收数据停止。

系统所需硬件：

10. 高云开发板。
11. 高云下载器及下载线。
12. 12V 的供电电源。
13. Type-C 线一根。
14. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

16.3.1I/O 约束

虽然我们完成了设计的顶层封装，但是要想在高云硬件平台上进行测试，我们还需要对设计综合，确认无误后进行管脚分配并约束电平。本次设计管脚分配如表 8-1：

表 16-2 管脚约束表

Pin Name	Signial Name	Pin NO.	Pin Name	Signial Name	Pin NO.
CLK_G	CLK	T9	UART_RXD	uart_rx	U8
KEY1	Reset_n	A15	UART_TXD	uart_tx	V8

KEY0	key_in	B16			

配置完成后将对应引脚约束为 LVCMOS33，接下就可以生成比特流，在比特流生成完成后连接硬件准备烧录验证了。

16.3.2硬件连接

连接开发板的下载线和电源线，使用串口线一端连接开发板上的 UART 接口，一端连接电脑的 USB 口，如下图 16-5 所示。

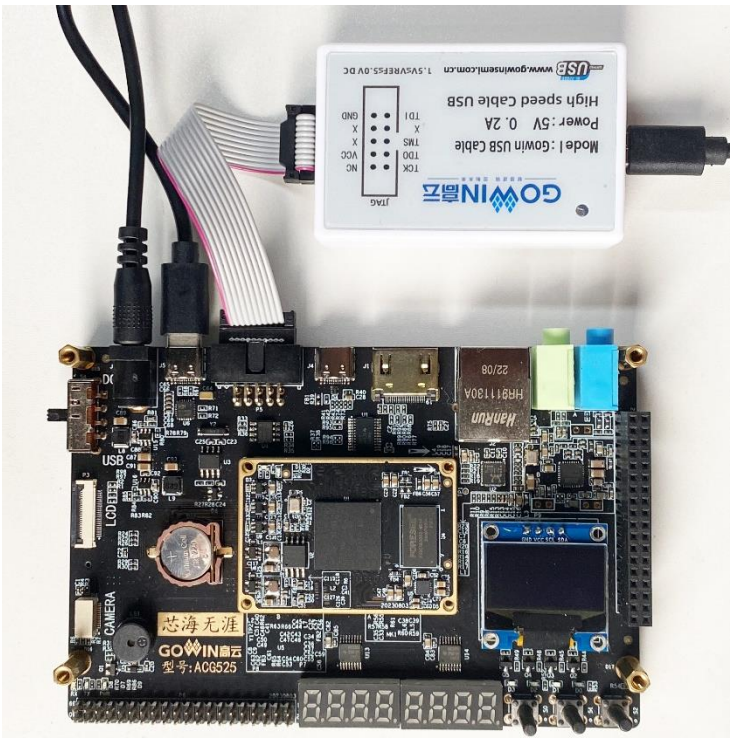


图 16-5 硬件连接

连接完成后，确保开发板电源拨码开关拨到 ON 侧，接下来将生成好的 bit 烧录到开发板中。

烧录完成后打开设备管理器，查询串口端口号，设备管理器中串口的名称如图 16-6 所示：

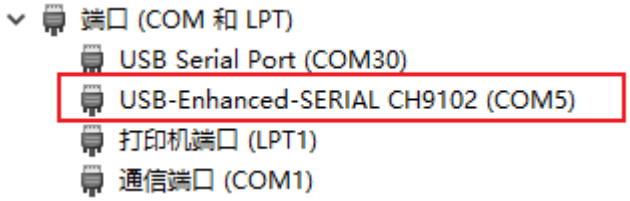


图 16-6 查询端口号

这里笔者电脑串口端口号为 COM5，用户在查询时，以自己电脑中的端口

号为准。随后打开串口猎人，在软件中连接上刚刚查询到的端口，设置波特率为 9600。然后在发码区依次输入 11、22、aa、dd、34、67 发送给 FPGA。按下按键 S0 之后即可看见串口助手源源不断的收到数据，再次按下按键 S0，串口助手接收数据停止。且可以计算处每一个循环的数据位宽均为 256 个，与设计的 RAM 宽度相符合。

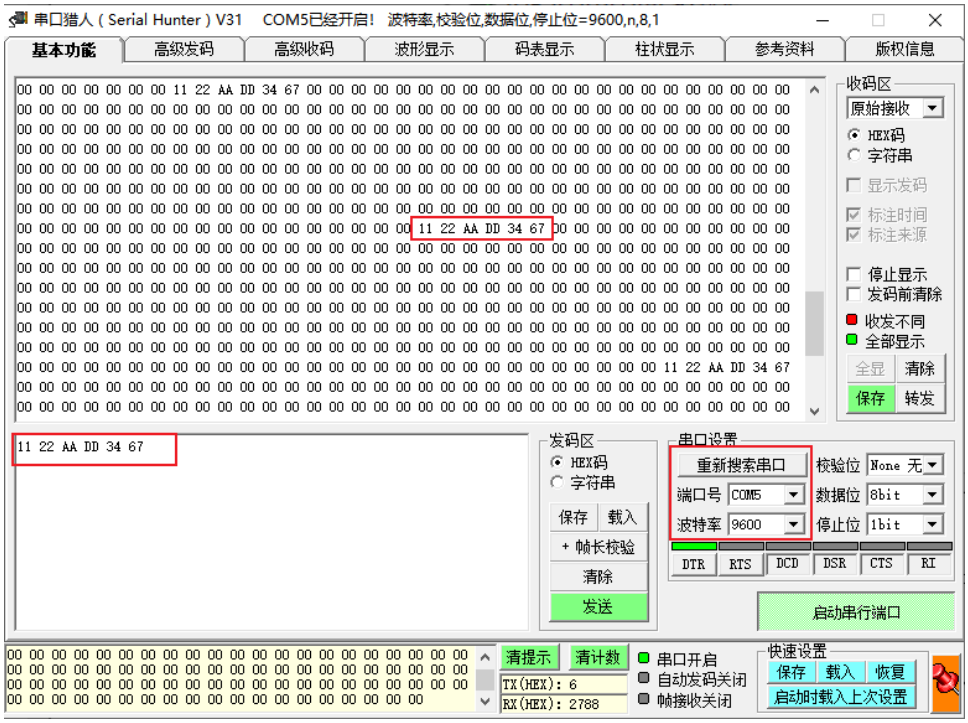


图 16-7 串口助手接收数据显示 1

现在即使按下复位再次发送 68、76、ff 可以看出，RAM 已有的数据并不会消除，只是会用新的数据依次覆盖上一次保存的数据。

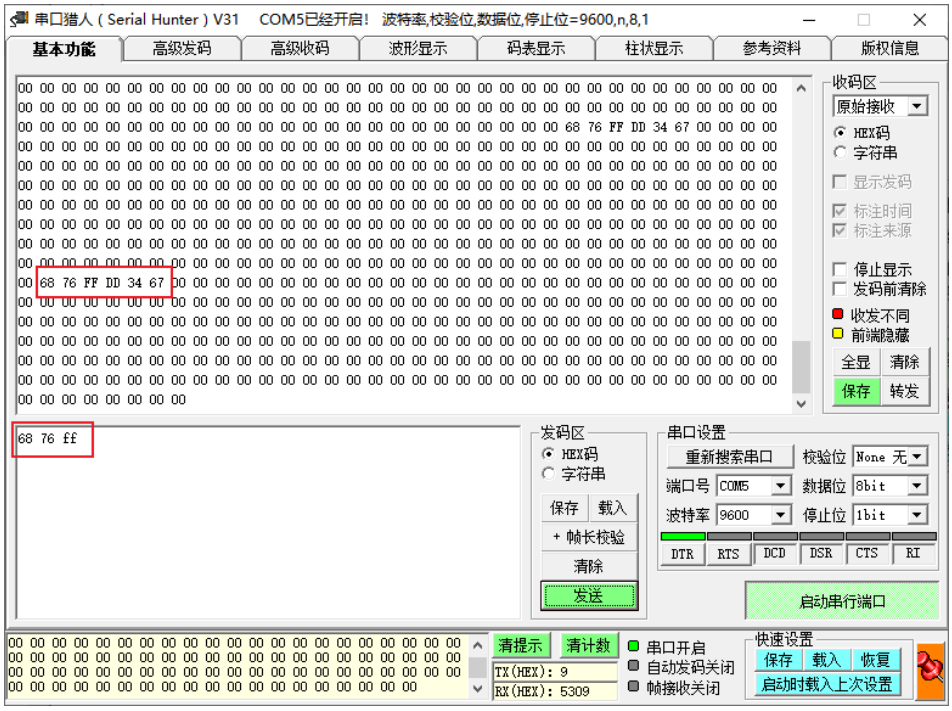


图 16-8 串口助手接收数据显示 2

16.4总结

本章利用前面相关章节编写好的按键消抖、串口发送与接收以及双端口 RAM 模块实现了串口发送数据到 FPGA 中，FPGA 将接收到的数据存储在双口 RAM 中，当按下按键时 FPGA 将 RAM 中存储的数据再通过串口发送出去。

读者可以尝试改变 IP 设置相关参数，观察波形。得益于之前编写模块的可移植性本章实际编写的内容并不多，再次显示了模块化设计的优点，在今后的设计中希望多加体会。

17 IP 核使用之 FIFO

工程源码	|----02_设计实例 |----ch17_fifo_ip
相关视频课程	
说明	

章节导读

FIFO（First In First Out），即先进先出。FPGA 或者 ASIC 中使用到的 FIFO 一般指的是对数据的存储具有先进先出特性的一个缓存器，常被用于数据的缓存或者高速异步数据的交互。它与普通存储器的区别是没有外部读写地址线，这样使用起来相对简单，但缺点就是只能顺序写入数据，顺序的读出数据，其数据地址由内部读写指针自动加 1 完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

本节将侧重介绍 FIFO 的基础知识和以双时钟 FIFO 为例讲解 FIFO IP 的生成和基本的使用。

17.1FIFO 相关知识

17.1.1FIFO 结构

FIFO 从读写时钟上来分有两类结构：单时钟 FIFO（同步 FIFO）和双时钟 FIFO（异步 FIFO）。单时钟 FIFO 具有一个时钟（读写共用一个时钟）输入，因此所有输入信号的读取都是在这个时钟的上升沿进行的，所有输出信号的变化也是在这个时钟信号的上升沿的控制下进行的，即单时钟 FIFO 的所有输入输出信号都是同步这个时钟信号的。而在双时钟 FIFO 结构中，写端口和读端口分别有独立的时钟，所有与写相关的信号都是同步于写时钟 wr_clk 的，所有与读相关的信号都是同步于读时钟 rd_clk 的。下面图 17-1 是双时钟 FIFO 的结构示意图。

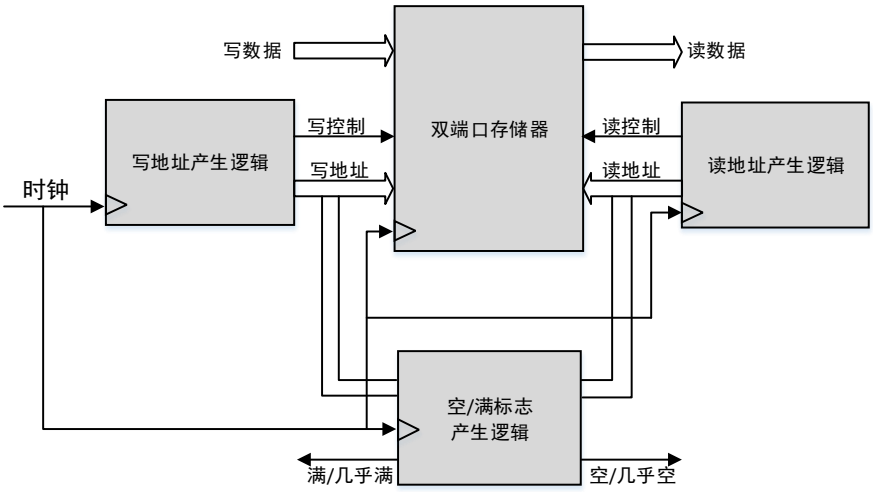


图 17-1 双时钟 FIFO 结构示意图

17.1.2FIFO 应用场景

1. 单时钟 FIFO

单时钟 FIFO 常用于片内数据交互。例如，在 FPGA 的控制下从外部传感器读取到的一连串传感器数据，首先被写入 FIFO 中，然后再以 UART 串口波特率将数据依次发送出去。由于传感器的单次读取数据可能很快，但并不是时刻都需要采集数据，例如某传感器使用 SPI 接口的协议，FPGA 以 2M 的 SPI 数据速率从该传感器中读取 20 个数据，然后以 9600 的波特率通过串口发送出去。因为 2M 的数据速率远高于串口 9600 的波特率，因此需要将从传感器中采集到的数据首先用 FIFO 缓存起来，然后再以串口的数据速率缓慢发送出去。这里，由于传感器数据的读取和串口数据的发送都是可以同步于同一个时钟的，因此可以使用单时钟结构的 FIFO 来实现此功能。

2. 双时钟 FIFO

双时钟 FIFO 的一个典型应用就是异步数据的收发，所谓异步数据是指数据的发送端和接收端分别使用不同的时钟域。使用双时钟 FIFO 能够将不同时钟域中的数据同步到所需的时钟域系统中。例如，在一个高速数据采集系统中，实现将高速 ADC 采集的数据通过千兆以太网发送到 PC 机。ADC 的采样时钟(CLK1)由外部专用锁相环芯片产生，则高速 ADC 采样得到的数据就是同步于该 CLK1 时钟信号，在 FPGA 内部，如果 FPGA 工作时钟(CLK2)是由独立的时钟芯片加片上锁相环产生的，则 CLK1 和 CLK2 就是两个不同域的时钟，他们的频率和相位没有必然的联系，假如 CLK1 为 65M，CLK2 为 125M，那么就不能使

用 125M 的数据来直接采集 65M 速率的数据，因为两者数据速率不匹配，在采集过程中会出现包括亚稳态问题在内的一系列问题，所以这里就可以使用一个具备双时钟结构的 FIFO 来进行异步数据的收发。如下图 17-2 为使用 FIFO 进行异步数据收发的简易系统框图。

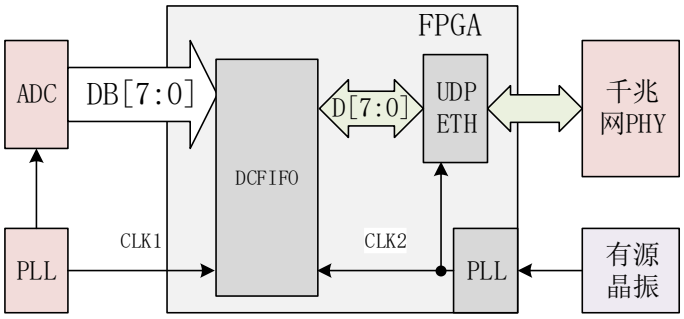


图 17-2 基于千兆以太网传输的高速数据采集（8bit）系统

在此系统中，由于 ADC 的数据位宽为 8 位，基于 UDP 协议的以太网发送模块所需的数据也是 8 位，因此使用的是读写数据宽度相同的双时钟 FIFO 结构。假如 CLK1 的频率为 20M，ADC 的数据位宽为 16 位，则可以使用读写数据位宽不同的双时钟 FIFO，在实现异步时钟域数据收发的同时，实现数据位宽的转换。通过设置双时钟 FIFO 的写入位宽为 16 位，读取位宽为 8 位，则可以实现将 16 位的 ADC 数据转换为以太网支持的 8 位发送数据，然后通过以太网发送到 PC 机。

17.1.3FIFO 常见参数

1. FIFO 的宽度：即 FIFO 一次读写操作的数据位。
2. FIFO 的深度：指的是 FIFO 可以存储多少个 N 位的数据（如果宽度为 N）。
3. 满标志：FIFO 已满或将要满时由 FIFO 的状态电路发送出的一个信号，以阻止 FIFO 的写操作继续向 FIFO 中写数据而造成溢出。
4. 空标志：FIFO 已空或将要空时由 FIFO 的状态电路送出的一个信号，以阻止 FIFO 的读操作继续从 FIFO 中读出数据而造成无效数据的读出。
5. 读时钟：读操作所遵循的时钟，在每个时钟沿来临时读数据。
6. 写时钟：写操作所遵循的时钟，在每个时钟沿来临时写数据。

17.1.4实现 FIFO 的方法

使用 FIFO 实现用户功能设计主要有三种方式。

第一种为用户根据需求自己编写 FIFO 逻辑，当对于 FIFO 的功能有特殊需求时，可以使用此种方式实现。

第二种方式为使用第三方提供的开源 IP 核，此种 IP 核以源码的形式提供，能够快速的应用到用户系统中，当用户对 FIFO 功能有特殊需求时，可以在此源码的基础上进行修改，以适应自己的系统需求。

第三种方式为使用 Gowin 软件提供的免费 FIFO IP 核，此种方式下，Gowin 软件为用户提供了友好的图形化界面方便用户对 FIFO 的各种参数核结构进行配置，生成 FIFO IP 核针对 Gowin 不同系列的器件，还可以实现结构上的优化。由于该 FIFO IP 核已经提供了大部分设计所需的所有功能，因此在系统设计中，推荐使用该 FIFO IP 核进行系统设计。

17.2双时钟 FIFO IP 的配置

打开 Gowin 软件，新建一个名为 fifo_ip 的工程，然后在 Gowin 软件上方的工具栏中点击  进入 IP Core Generator 界面，然后在收索栏中输入 fifo，可以看到在 Memory Control 下有两个 FIFO IP 供我们使用，一个是双时钟 FIFO（FIFO），一个是单时钟 FIFO（FIFO SC），如下图 17-3 所示。

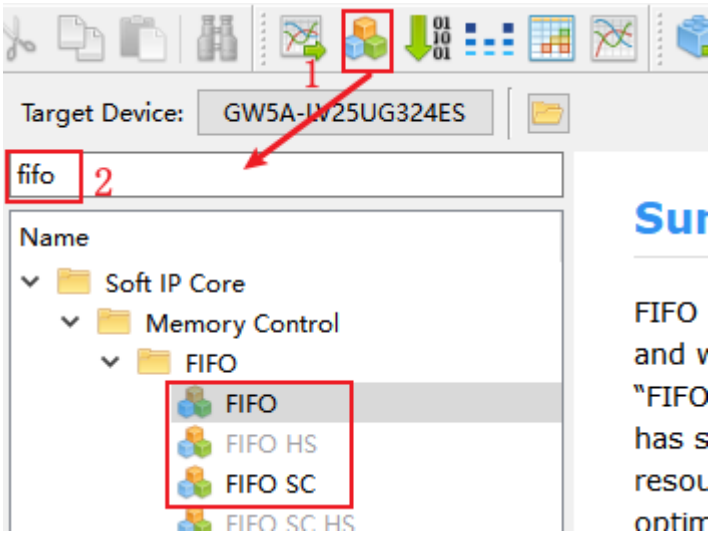


图 17-3 搜索 FIFO IP

本章实验将以双时钟 FIFO 为例进行说明，双击 FIFO 进入配置界面，如下图 17-4 所示。

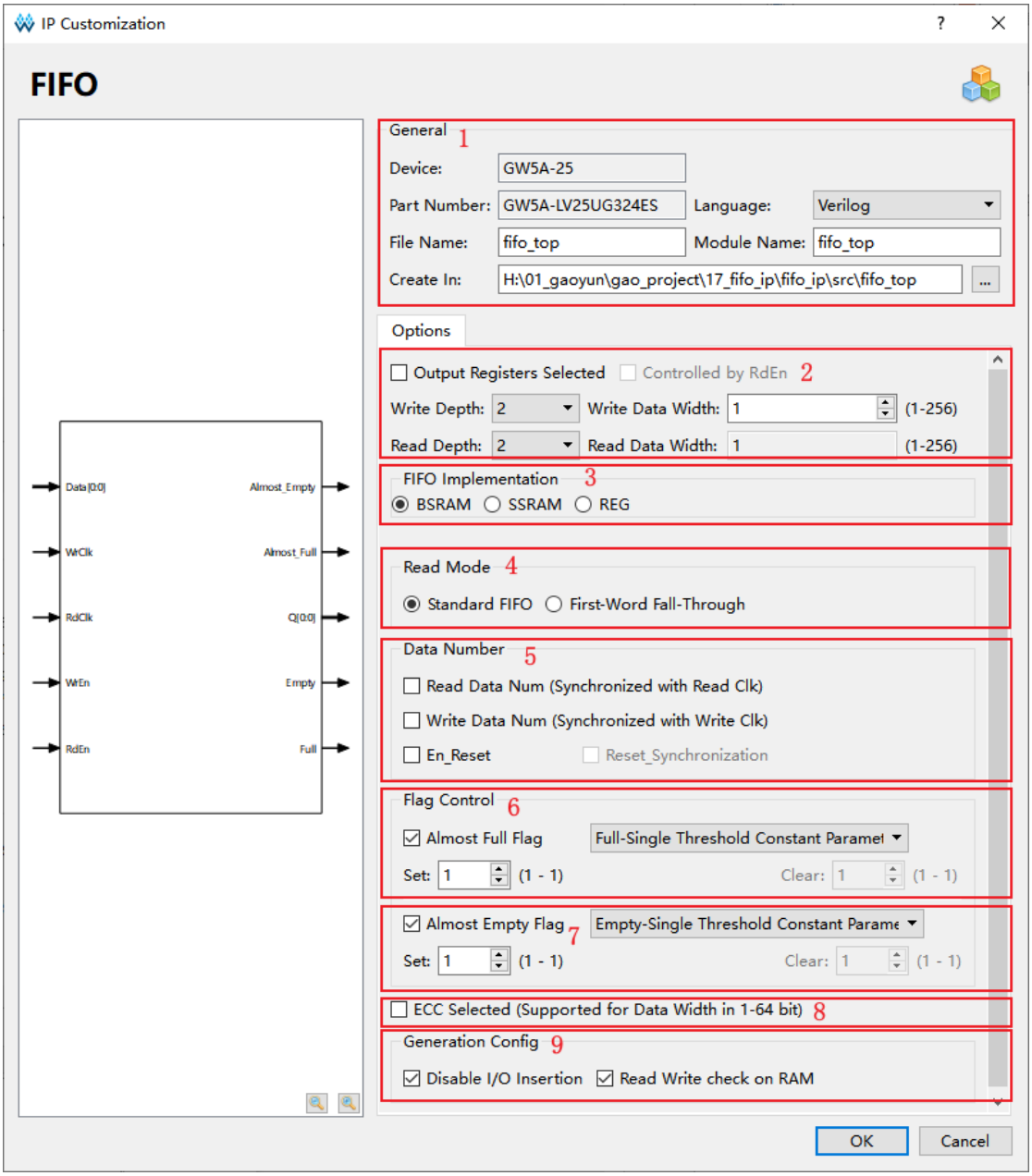


图 17-4 FIFO 配置界面

对于上述图中的各项配置说明如下表 17-1 所示。

表 17-1 FIFO 配置说明表

编号值	选项说明	参数名称	描述	备注
0	General	Device	显示已配置的 Device 信息	
		Part Number	显示已配置的 Part Number 信息	
		Language	配置产生的 IP 设计文件的硬件描述语言，选择右侧下拉列表框，选择目标语言，支持 Verilog 和 VHDL	
		Module Name	配置产生的 IP 设计文件的	

			module name。在右侧文本框可重新编辑模块名称。Module Name 不能与原语名称相同，若相同，则报出 Error 提示	
		File Name	配置产生的 IP 设计文件的文件名。在右侧文本框可重新编辑文件名称	
		Create In	配置产生的 IP 设计文件的目标路径。可在右侧文本框中重新编辑目标路径，也可通过文本框右侧选择按钮选择目标路径	
2	FIFO 读写深度以输出寄存器配置	Output Registers Selected	输出寄存器可选，有效时，数据延迟一个周期输出。BSRAM 输出数据需 要满足时钟到输出延 时。当 Output Register 使能打开时，这个时间 自动满足；当 Output Register 使能关闭时，用户需满足时钟到输 出延时，避免采样到亚稳态的数据。	异步 FIFO IP 满足写入数据宽度不同于读出数据 宽 度 ， Read Data Width 自 动 计 算 等 于 Write Depth x Write Data Width / Read Depth。
		Write Depth	写数据深度	
		Write Data Width	写数据位宽	
		Read Depth	读数据深度	
		Read Data Width	读数据位宽	
3	FIFO Implementation	Block SRAM	配 置 使 用 Block SRAM 、Shadow SRAM、LUT	
		Shadow SRAM		
		LUT		
4	Read Mode	Standard FIFO	标准模式	标准 FIFO 按照时序图标准读写时序，FWFT FIFO 不管是否有读使能信号，都会将写入的第一个数马上放在输 出数据总线上，读使能 拉高后会按顺序输出 写入的其他数据
		First-Word Fall Through	FWFT 模式	
5	Data Num	Read Data Num	写数据数目	有 效 时 ， 增 加 输 出 Rnum
		Write Data Num	读数据数目	有 效 时 ， 增 加 输 出 Wnum
		En_Reset	使能复位	使能复位，使能时，读写各自复位，增加输入 WrReset、RdReset
6	Almost Full Flag	Full-Single Threshold Constant Parameter	静态半满单常量阈值，有效时，Set 有效	半满标志使能，有效 时 ， 增 加 输 出 Almost_Full。
		Full-Dual Threshold Constant Parameters	静态半满双常量阈值， 有效时，Set、Clear 有效。	

		Full-Single Threshold Input Parameter	动态半满单输入阈值， 有效时，增加输入 AlmostFullTh	
		Full-Dual Threshold Input Parameters	动态半满双输入阈值， 有效，增加输入 AlmostFullSetTh、AlmostFullClrTh	
		Set	半满置 1 阈值大小	选择单常量阈值，Set 有效 选择双常量阈值，Set、 Clear 有效
		Clear	半满清 0 阈值大小	
7	Almost Empty Flag	Empty-Single Threshold Constant Parameter	静态半空单常量阈值， 有效时，Set 有效。	半空标志使能，有效时，增加输出 Almost_Empty。
		Empty-Dual Threshold Constant Parameters	静态半空双常量阈值， 有效时，Set、Clear 有效。	
		Empty-Single Threshold Input Parameter	动态半空单输入阈值， 有效时，增加输入 AlmostEmptyTh。	
		Empty-Dual Threshold Input Parameter	动态半空双输入阈值， 有效，增加输入 AlmostEmptySetTh、AlmostEmptyClrTh。	
		Set	半满置 1 阈值大小	选择单常量阈值，Set 有效 选择双常量阈值，Set、 Clear 有效
		Clear	半满清 0 阈值大小	
8	ECC Selected		ECC 功能	Data width 1-64bit 时 有效，增加输出 ERROR
9	Generation Config	Disable I/O Insertion	禁用 I/O 插入	
		Read Write check on RAM	RAM 读写检查	

首先设置读写深度为 256，最大可设置读写深度为 65536，写数据位宽为 8，读数据位宽等于 256*8/256，界面将会自动识别为 8，FIFO Implementation 按照默认选项设置为 BSRAM，如下图 17-5 所示。

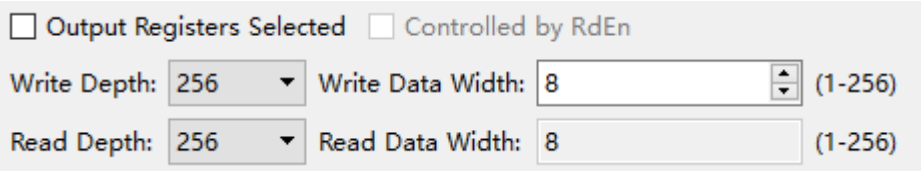


图 17-5 FIFO 读写深度设置

设置读模式，有两种方式可供选择：Standard FIFO 和 First-Word Fall Through，两种工作方式的时序图如下图 17-6、图 17-7 所示（都未勾选输出寄存器，读写位宽和深度都为 8）。

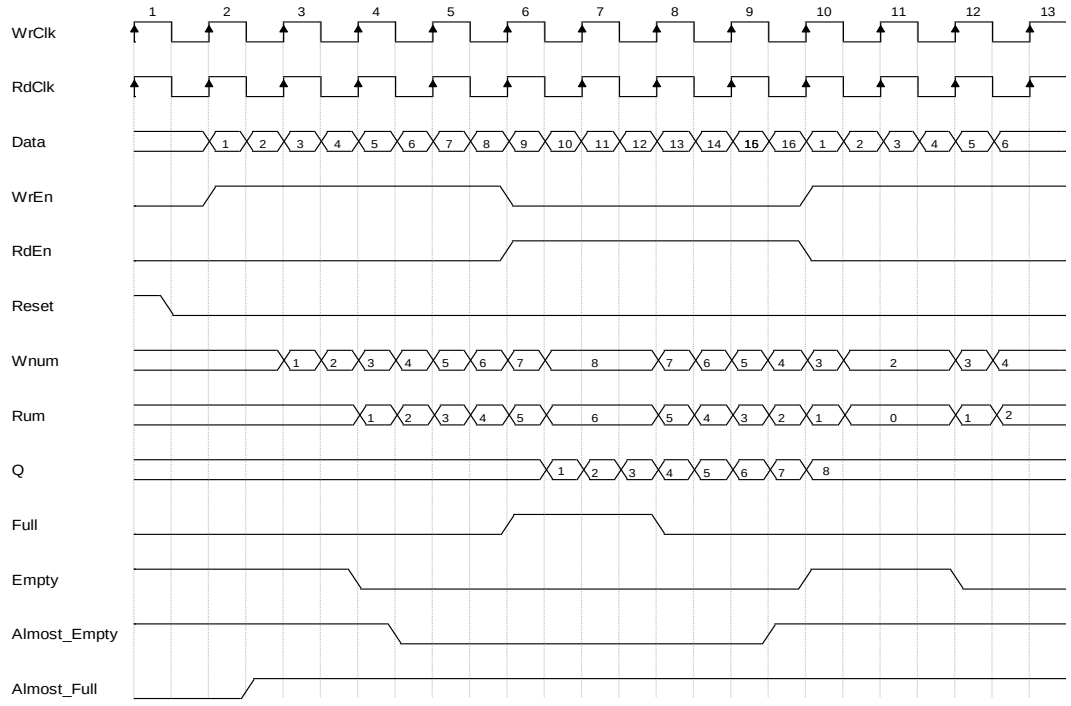


图 17-6 FIFO 工作时序图（Standard FIFO 模式）

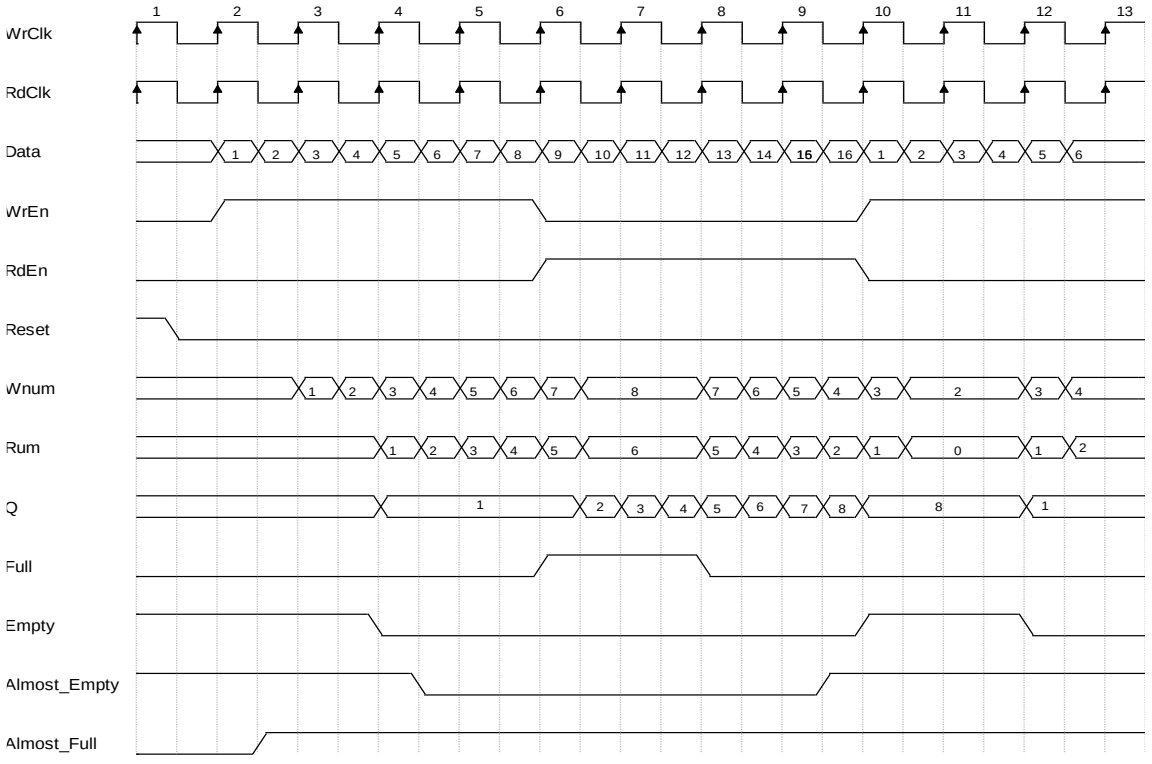


图 17-7 FIFO 工作时序图（FWFT 模式）

对比两种模式的读数据的不同，当 FIFO 工作在 Standard FIFO 模式的时候，读使能拉高之后，将写入 FIFO 的数据依次读出至 Q，选择 FIFO IP 的 FWFT 模式时，在 FIFO 为空状态时，不管是否有读使能信号，都会将写入的第一个数马

上放在输出数据总线上，当读使能拉高后会按顺序输出写入的其他数据。本章实验选择 FWFT 模式。

勾选读写计数并使能复位，当勾选 En_Reset 之后，出现读写复位两个独立信号，如果同时勾选 En_Reset 和 Reset_Synchronization，读写使用同一个复位信号，这里两个信号同时勾选，Almost Full Flag 设置为 Full-Single Threshold Constant Parameter，FIFO 的其它配置按照默认即可，FIFO 最终的配置如下所示。

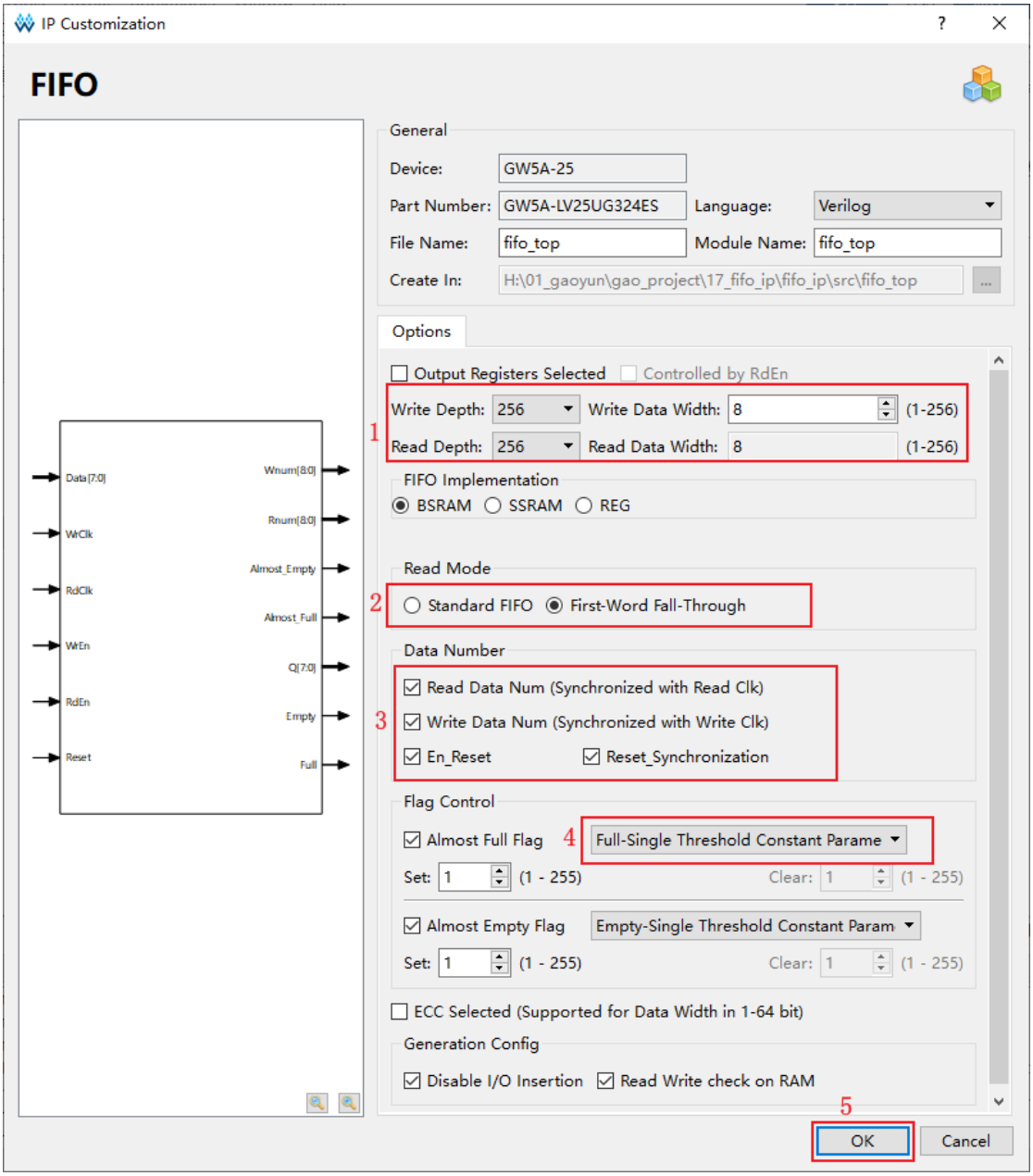


图 17-8 FIFO 最终配置

点击 OK 之后，弹出是否创建文件的对话框，点击 OK，出现 FIFO 的例化

代码，如下图 17-9 所示，此时说明 FIFO IP 添加至工程成功。

```
fifo_top your_instance_name(
    .Data(Data_i), //input [7:0] Data
    .Reset(Reset_i), //input Reset
    .WrClk(WrClk_i), //input WrClk
    .RdClk(RdClk_i), //input RdClk
    .WrEn(WrEn_i), //input WrEn
    .RdEn(RdEn_i), //input RdEn
    .Wnum(Wnum_o), //output [8:0] Wnum
    .Rnum(Rnum_o), //output [8:0] Rnum
    .Almost_Empty(Almost_Empty_o), //output Almost_Empty
    .Almost_Full(Almost_Full_o), //output Almost_Full
    .Q(Q_o), //output [7:0] Q
    .Empty(Empty_o), //output Empty
    .Full(Full_o) //output Full
);
```

图 17-9 FIFO 例化代码

17.3仿真验证

为了测试仿真编写测试测试激励文件，添加并新建仿真文件命名为 fifo_ip_tb.v。上面创建的 FIFO 是独立时钟，首先分别产生写时钟信号 WrClk 和读时钟信号 RdClk，周期时间值通过 define 分别设置为 10ns 和 30ns。仿真中首先是产生一段时间的复位信号（高电平复位），然后通过一个 while 循环在 FIFO 没有满的情况下，一直往 FIFO 里写入数据，数据值从 FF 开始递加，为了让仿真波形更加贴合实际，写使能信号在时钟上升沿之后延时 1ns 给高电平，这样能更加清晰看到 FIFO 各信号之间的关系。在 FIFO 写满之后将写使能置 0，停止写入数据。读操作类似，在 FIFO 为非空的情况下进行读数据操作，直到 FIFO 读空就停止读操作。仿真代码如下所示：

```
`timescale 1ns / 1ns
`define WR_CLK_PERIOD 10
`define RD_CLK_PERIOD 30

module fifo_ip_tb();
    reg [7:0]Data;
    reg Reset;
    reg WrClk;
    reg RdClk;
    reg WrEn;
    reg RdEn;
    wire [8:0] Wnum;
    wire [8:0] Rnum;
    wire Almost_Empty;
```

```
    wire Almost_Full;
    wire [7:0] Q;
    wire Empty;
    wire Full;

    initial WrClk = 1;
    always #(`WR_CLK_PERIOD/2) WrClk = ~WrClk;

    initial RdClk = 1;
    always #(`RD_CLK_PERIOD/2) RdClk = ~RdClk;

    initial begin
        Reset   = 1'b1;
        WrEn = 1'b0;
        RdEn = 1'b0;
        Data   = 8'hFF;
        #(`WR_CLK_PERIOD*8+1);
        Reset = 1'b0;

        //write data
        while(Full == 1'b0)
        begin
            @(posedge WrClk);
            #1;
            WrEn = 1'b1;
            Data   = Data + 1'b1;
        end
        WrEn = 1'b0;

        while(Empty == 1'b0)
        begin
            @(posedge RdClk);
            #1;
            RdEn = 1'b1;
        end
        RdEn = 1'b0;

        //reset
        #200;
        Reset   = 1'b1;
        #(`WR_CLK_PERIOD*3+1);
        Reset = 1'b0;

        #2000;
        $stop;
    end
```

```
    fifo_top fifo_top(
        .Data(Data), //input [7:0] Data
        .Reset(Reset), //input Reset
        .WrClk(WrClk), //input WrClk
        .RdClk(RdClk), //input RdClk
        .WrEn(WrEn), //input WrEn
        .RdEn(RdEn), //input RdEn
        .Wnum(Wnum), //output [8:0] Wnum
        .Rnum(Rnum), //output [8:0] Rnum
        .Almost_Empty(Almost_Empty), //output Almost_Empty
        .Almost_Full(Almost_Full), //output Almost_Full
        .Q(Q), //output [7:0] Q
        .Empty(Empty), //output Empty
        .Full(Full) //output Full
    );
endmodule
```

然后打开 Modelsim 新建仿真工程，将 fifo_ip_tb.v 文件和 fifo_top.vo 文件添加至工程中，配置仿真环境，将 fifo_ip_tb 的端口信号添加至波形窗口进行观察，然后点击观察全局波形，如下图 17-10 所示。

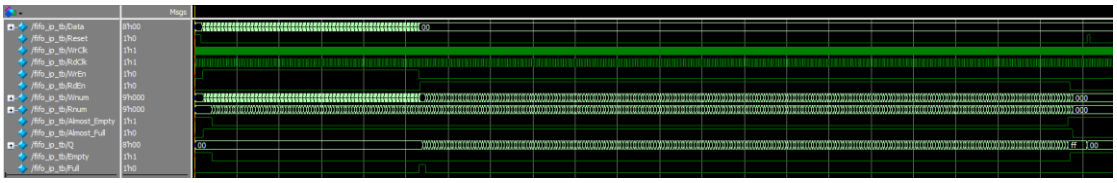


图 17-10 仿真全局波形

复位完成后，在非满情况下进行写数据操作，直到 FIFO 的 full 信号被拉高就停止写操作，非满情况下的写数据波形图如下图 17-11 所示。

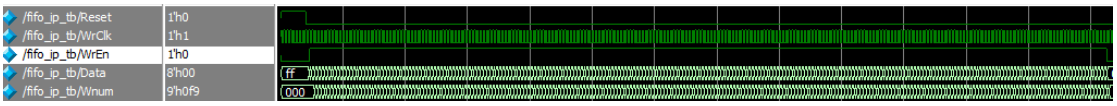


图 17-11 非满情况下写数据全图

对写操作的波形头尾进行放大，放大后的波形图如下所示。

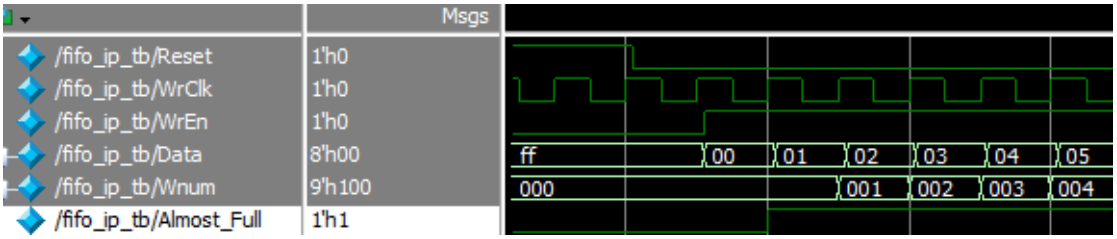


图 17-12 一组数据写操作波形头部进行放大后的波形图

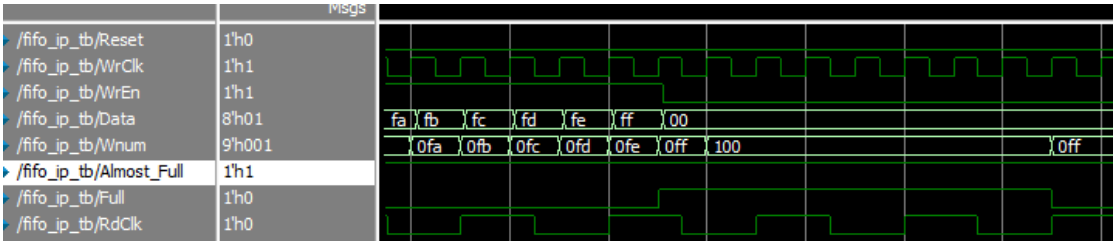


图 17-13 一组数据写操作波形尾部进行放大后的波形

从波形上可以看出写入的第一个数据为 0x00，也就是产生写使能之后 0xff 加 1 得到的，Wnum 是对写入数据的计数，可以看到计数值变化延迟写操作两个时钟周期出现，Almost_Full 在写入一个数据之后被拉高，这是因为我们在配置的时候按照默认的，Almost_Full Flag 信号的 set 的值为 1。从最后写尾部可以看到写入的最后一个数据为 0xFF，写完之后，Full 信号被拉高，表明 FIFO 被写满了，从 0x00~0xFF 写入的数据为 256 个，此时延迟两个时钟周期，可以看到 Whum 的值为 0x100，也就是 256，此时表明 FIFO 的写入数据正常。

然后观察 FIFO 的读操作部分，读数据波形全貌如下图 17-14 所示。

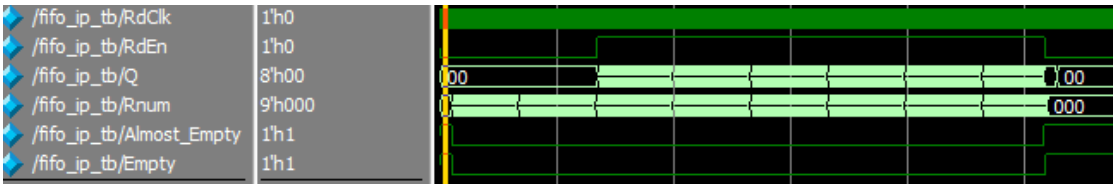


图 17-14 一组数据读数据波形全貌

对读操作波形头尾进行放大后波形如下所图 17-15、图 17-16 示。

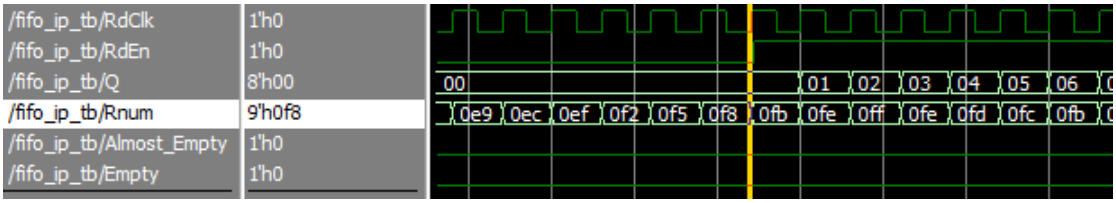


图 17-15 一组数据读数据波形头部

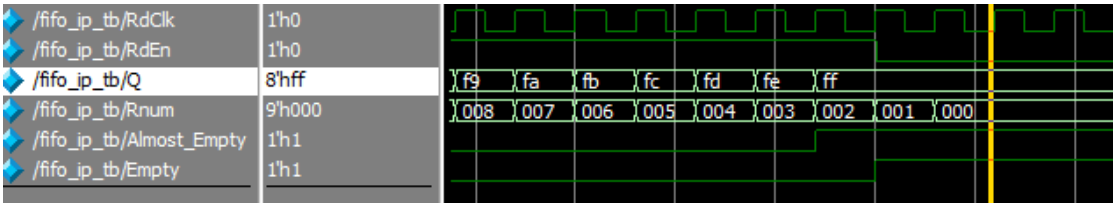


图 17-16 一组数据读数据波形尾部

从上述波形图可以看出，读出的第一个数据为 0x00，读出的最后一个数据为 0xFF，并且数据是连续递增的，和写入的数据一致。Rnum 并不是读出的数

据个数，而是 FIFO 中可读数据个数，同步与读端口时钟，延迟读使能信号两个时钟周期。Empty 在 Rnum 为 0x001 时拉高，表示 FIFO 为空，Almost_Empty 提前一个时钟周期出现，这是因为我们在配置 FIFO IP 的时候，默认的 Almost Empty Flag 为 1，如果为 2 的话，Almost Empty Flag 将会在 Rnum 为 0x003（0x001+2）时拉高。

通过对比以上波形可以看到我们设置的 FIFO IP 处于正常工作状态。

17.4总结

本章介绍了 FIFO 的分类、区别以及相关概念，并实现了双时钟 FIFO 的配置且进行了仿真，读者可以通过修改 FIFO 的一些配置进行进一步的仿真验证，以便更好的理解 FIFO 中的各项标志信号，当然，我们也会在后面的一些工程实验会进一步学习 FIFO 的使用。

18 IP 核使用之 PLL

工程源码	|----02_设计实例 |----ch18_pll_ip
相关视频课程	
说明	

章节导读

在 verilog 设计中，程序的运行往往都是围绕着时钟展开，越是复杂的设计往往会涉及越多不同的时钟。而对于开发板来说，通常都只设计有一个晶振，高云开发板上就板载了一个 50MHz 的有源晶振。通过开发板内部逻辑，虽然能够基于该时钟分频倍频，产生不同频率的时钟，但是这些时钟往往质量较差，并不适合应用。

FPGA 厂商为了解决这个问题，会在器件内部加入专用的时钟电路，也就是我们常说的锁相环（PLL）。通过该专用时钟电路分频倍频产生的时钟，不仅质量好，精度也会更高。本章我们将带大家学习锁相环的工作机理，并结合 Gowin 提供的 PLL 软核，通过一个简单的应用来带领大家熟悉锁相环的基础使用方法。

18.1PLL 电路原理

锁相环（PLL，Phase-Locked Loop），是一种反馈控制电路，常常用于利用外部输入的参考信号控制环路内部振荡信号的频率和相位。锁相环在工作时，当输出信号的频率与输入信号的频率相等时，输出电压与输入电压保持固定的相位差值，即输出电压与输入电压的相位被锁住，因此得名锁相环。

锁相环通常由图 18-1 所示的架构组成：

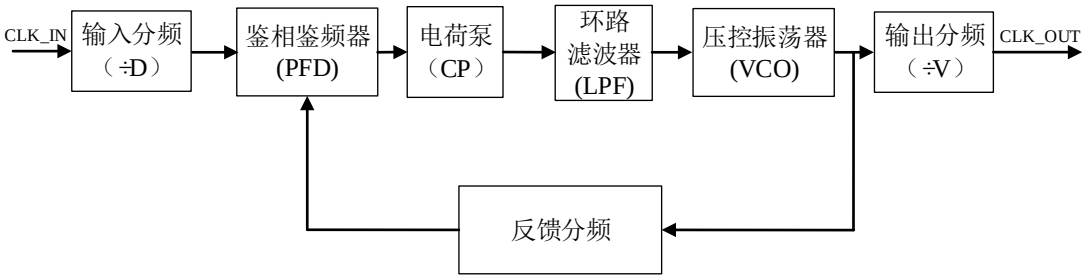


图 18-1 PLL 结构框图

其中，输入分频、输出分频、反馈分频为三个分频计数器，对时钟分频以满足需求。其余四个模块的功能分别如下：

- 鉴相鉴频器 PFD（Phase Frequency Detector）：对输入的基准信号（通常是来自频率稳定的晶振）和反馈回路的信号进行频率的比较，输出一个代表两者相位差异的信号。
- 电荷泵（CP）：根据 PFD 输出的信号，产生对应电压。
- 环路滤波器 LF（Loop Filter）：用于控制噪声的带宽，滤掉高频噪声，保留直流部分。
- 压控振荡器 VCO（Voltage Controlled Oscillator）：根据滤波器输入的电压，输出对应频率的周期信号。环路滤波器输入的电压越大 VCO 输出的频率越高，进而产生 N 倍于输入时钟的新时钟。

其中，VCO 输出的时钟经过反馈分频后传回 PFD 这一电路我们称之为反馈回路。PLL 在工作时，压控振荡器输出的时钟信号在经过反馈回路后输入到 PFD 中，PFD 会将其与输入的基准时钟比较，从而得到二者间的频率和相位差。频率和相位差会以信号的方式输出，驱动 CP 产生电压，经过低通滤波后转换为直流脉冲电压，作为 VCO 的控制电压，驱动 VCO 改变输出时钟。输出时钟又会经由反馈回路，输入到 PFD 与基准时钟对比，如此往复，最终输出稳定的满足需求的时钟。

因此，PLL 输出的时钟并不是由输入的基准时钟直接分频倍频得来，而是基于基准时钟，通过内部的震荡电路生成新的时钟，再经由反馈电路将时钟环回给 PFD，通过不断将新产生的时钟与基准时钟作比较，最终输出频率和相位稳定的时钟。

也正是因为如此，在使用 PLL 时，当基准时钟输入进 PLL 之后，我们并不能立马得到输出时钟，即使得到也不能立马使用。因为此时的时钟还并不稳定，需要等待一段时间之后，才能得到精确且稳定的时钟。

18.2Arora V FPGA 时钟资源简介

时钟资源对 FPGA 高性能的应用至关重要。Arora V FPGA 产品提供了专用全局时钟网络 GCLK（包括 PCLK 和 LW），直接驱动器件全局。除了 GCLK 资源，还提供了锁相环（PLL）、高速时钟 HCLK 和 DDR 存储器接口数据脉冲时钟 DQS 等时钟资源。

1. 全局时钟

GCLK 又分为 PCLK 和 LW。PCLK 可实现对全局的驱动。LW 一方面可以

作为控制线，给 DFF 提供时钟使能（CE）、置复位（SET/RESET）信号；另一方面，还可以用作逻辑绕线，作为普通数据信号使用。

2. 高速时钟

Arora V FPGA 产品的高速时钟 HCLK，具有低抖动和低偏差性能，可以支持 I/O 完成高性能数据传输，是专门针对源时钟同步的数据传输接口而设计的。一个 Bank 支持四路 HCLK。

3. PLL

Arora V FPGA 提供了锁相环 PLL，支持 7 路时钟输出，每路时钟可独立基于给定的参考输入时钟进行时钟频率、相位和占空比调整。PLL 可基于给定的参考输入时钟进行时钟频率调整、相位调整、占空比调整来产生不同频率、相位和占空比的输出时钟。CLKOUT0 和 CLKFBOUT 支持 1/8 小数频率调整，CLKOUT0~CLKOUT3 支持动静态微调占空比。同时 PLL 支持 CLKOUT6 到 CLKOUT4 的内部级联，支持 SSC 功能，支持时钟去歪斜以实现 CLKIN 和 CLKOUT 的对齐。

本节只是对 Arora V FPGA 时钟资源简要介绍，如果想要进一步了解，可以查看高云官方给的资料，快速查看方式，就是进入 Gowin 软件，进入 IP Core Generator 界面，输入 PLL，然后选择 PLL_ADV，在右边的串口中点击“UG306 – Gowin Clock User Guide(CN)”，最后就会进入官方给的中文资料，操作如下图 18-2 所示：

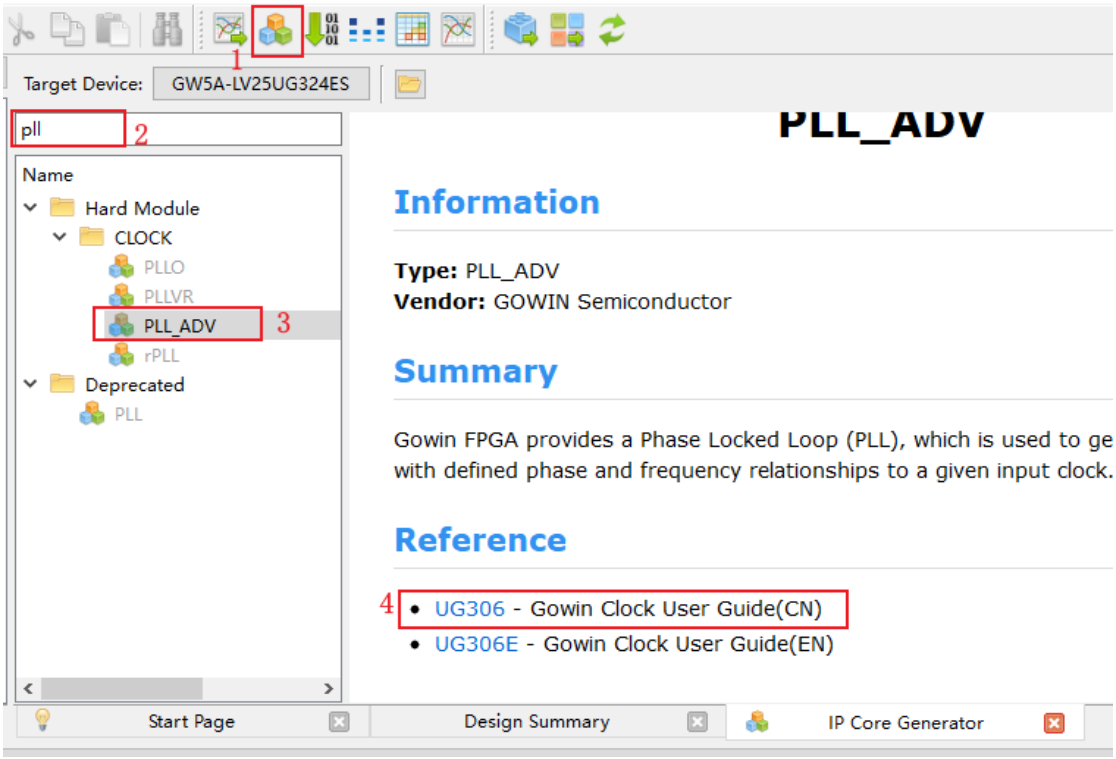


图 18-2 查看官方资料快速方法

18.3 PLL IP 配置

在 IP Core Generator 界面，输入 PLL，找到 PLL_ADV，双击进入配置界面，如下图 18-3 所示。

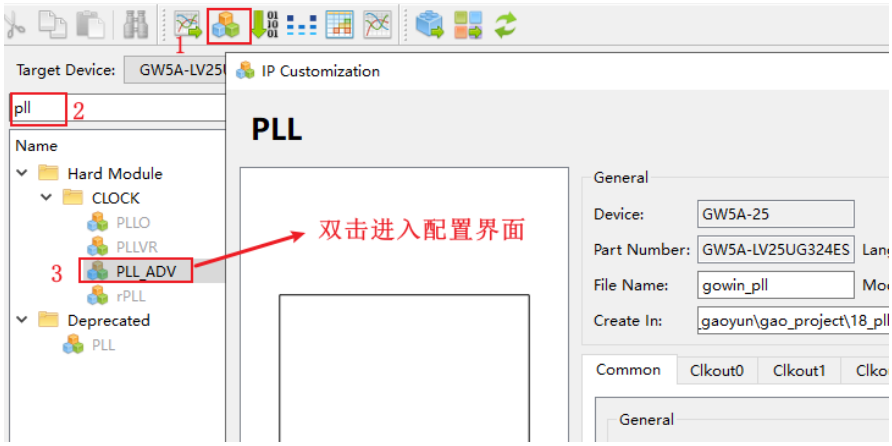


图 18-3 进入 PLL 配置界面操作示意图

下面对 PLL IP 各项配置进行说明。

18.3.1PLL Common 配置

PLL Common 配置界面如下图 18-4 所示。

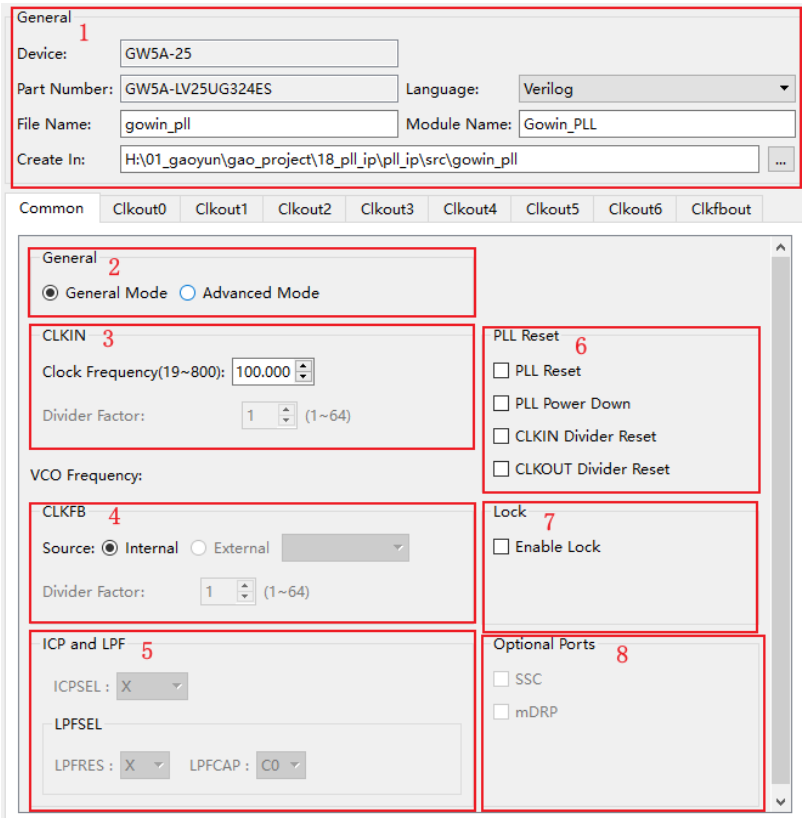


图 18-4 PLL Common 配置界面

1. General 配置框 1

General 配置框用于配置产生的 IP 设计文件的相关信息，读者可以在该对话框中文件名称，模块名称，以及文件保存路径。

2. General 配置框 2

配置一般模式和高级模式。一般模式下输入输入时钟频率和输出时钟频率，软件会自动计算不同的分频系数；高级模式使用与高级用户，允许输入输入频率和不同分频系数得到预期的输出频率。我们按照默认的一般模式（General Mode）即可。

3. CLKIN

配置 PLL 输入时钟频率，分频参数设置。

- Clock Frequency：配置输入时钟频率，这里选择开发板上 50M 的时钟晶振作为时钟输入源。

- Divide Factor：可在高级模式下配置分频参数，范围 1~64。

4. CLKFB

配置 PLL 反馈时钟的源和倍频参数。

- Source：配置反馈时钟源，可以选择“Internal”和“External”；如果选择 Internal，则反馈来自内部；如果选择 External，则反馈来自 CLKOUT0~6 和 CLKFBOUT 中的一个，用户可自行选择，External 只在 Advanced Mode 模式有效，前面我们选择的一般模式，所以这里只能按照默认选择 Internal。
- Divide Factor：可在高级模式下配置倍频参数，范围为 1~64。

5. ICP and LPF

动态控制 ICP 电流大小，电流随着取值的增大而增大，值为 0 时电流最小。

- LPFSEL：动态控制 LPFSEL 大小，RES 取值范围由小到大为 R0~R7，R0 对应的带宽最大，R7 对应的带宽最小。
- LPFCAP：动态控制 LPFCAP 大小，CAP 取值范围由小到大为 C0~C2。

6. PLL Reset

- PLL Reset：PLL 全部复位信号，复位数字电路，配置 PLLO 的 RESET 使能模式。
- PLL Power Down：则对模拟电路 power down 复位。
- CLKIN Divider Reset：配置使能 RESET_I。
- CLKOUT Divider Reset：配置使能 RESET_O。

我们这里选择“PLL Reset”。

7. Lock

PLL 锁定指示信号，勾选。

8. Optional Ports

可选端口设置，只在 Advanced Mode 模式下有效。

最终的 Common 配置界面如下图 18-5 所示。

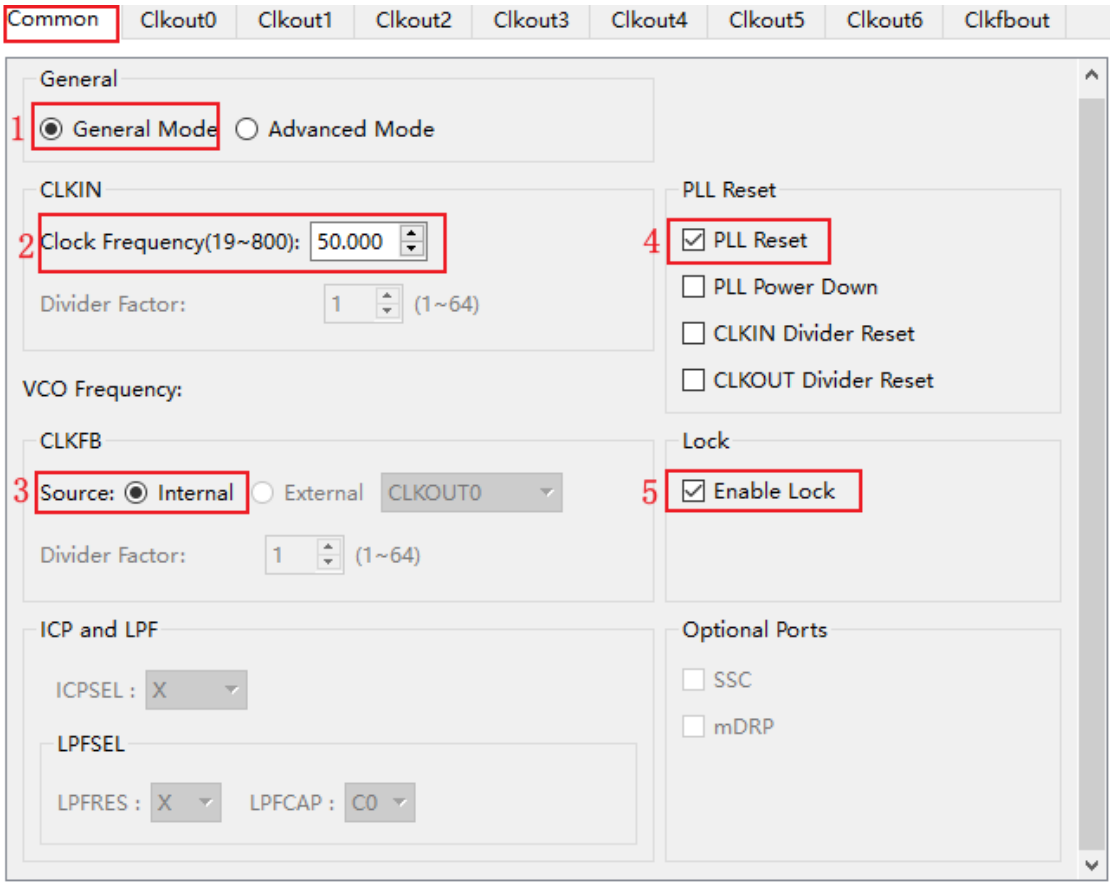


图 18-5 Common 最终配置界面

18.3.2Clkout 配置

这里以 Clkout 0 的配置为例进行说明，Clkout 0 的配置界面如下所示。

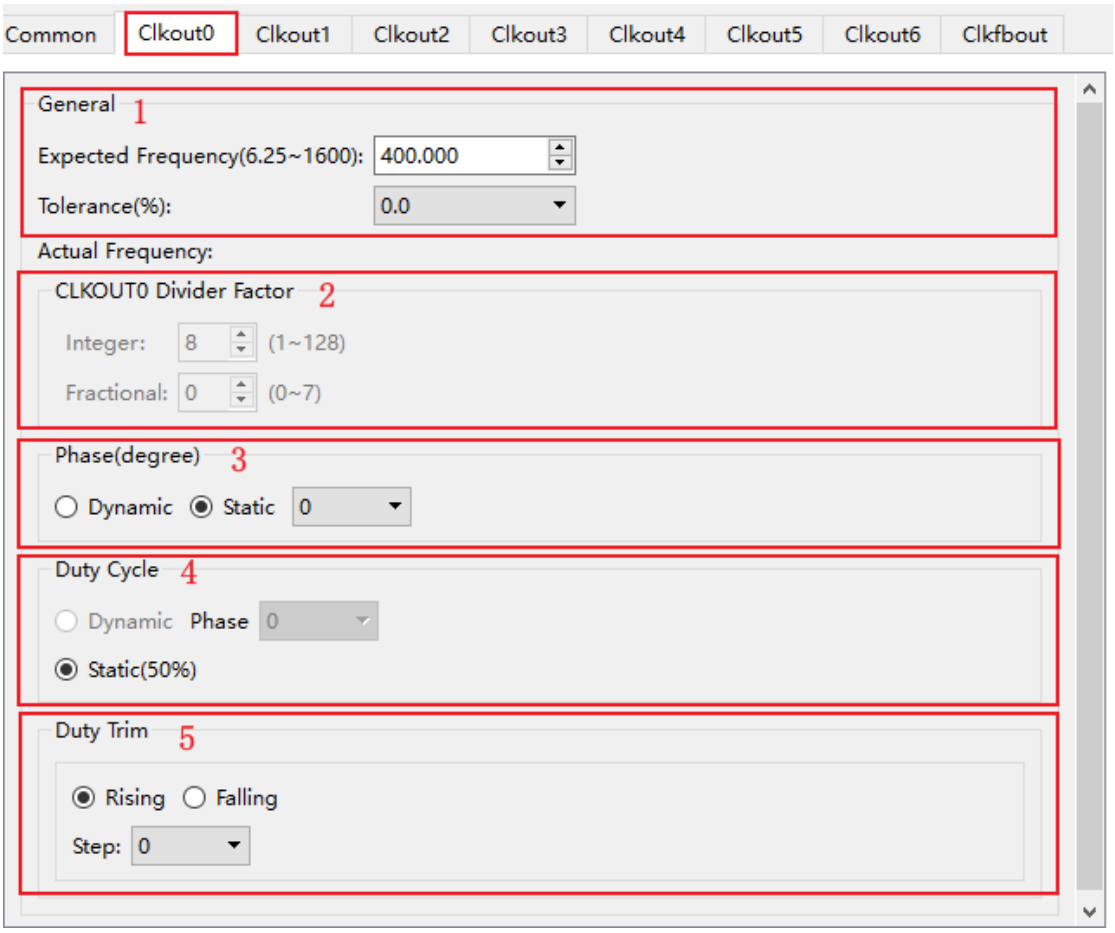


图 18-6 Clkout 0 配置界面

1. General 配置

- Expected Frequency：频率范围配置界面，范围为 6.25M~1600M，这里设置 Clkout0 的输出频率为 100M。
- Tolerance：配置 Clkout0 期望频率和计算出的实际 频率的允许误差，这里按照默认选择的 0。

2. CLKOUT0 Divider Factor

对 Clkout0 的输出分频器进行设置，该选项只在 Advanced Mode 模式下有效。

3. Phase 设置

PLL 相位调整支持动态（Dynamic）和静态（static）两种方式，这里按照默认选择即可。

4. Duty Cycle

PLL 占空比调整只支持静态调整，占空比 50%。

5. Duty Trim

PLL 占空比微调，可控制选择调整上升沿还是下降沿，以及要调整的 step，该选项只支持 Clkout0~3。占空比微调对照表如下表 18-1 所示。

表 18-1 占空比微调对照表

调整方向	调整值	占空比微调延时值
Rising（1'b0）	0	0
	1	-50ps
	2	-100ps
	4	-200ps
Falling（1'b1）	0	0
	1	+50ps
	2	+100ps
	4	+200ps

Clkout0 的最终配置如下图 18-7 所示。

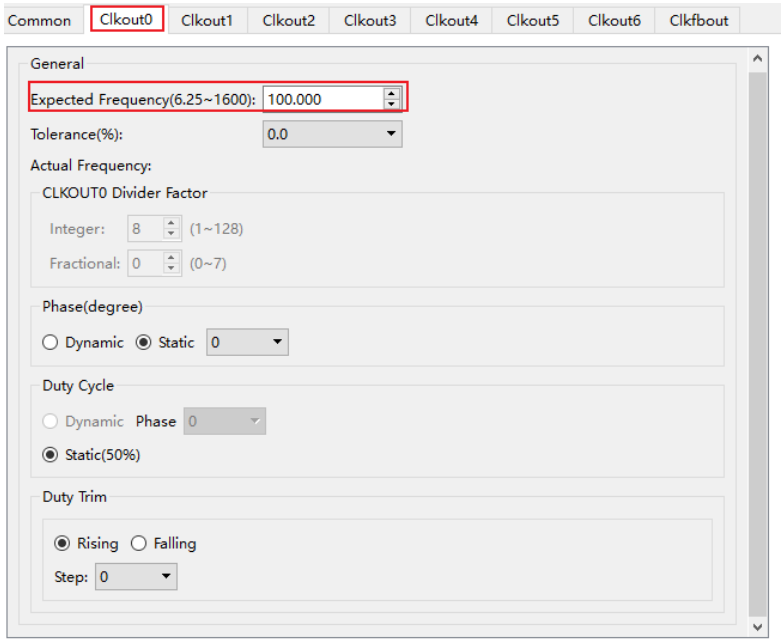


图 18-7 配置 Clkout0 的输出频率

我们一共配置 4 路输出，剩下的 Clkout1~3 都可以参考 Clkout0 的方式进行配置，有一点不同就是 General 配置，如下图 18-8 所示。

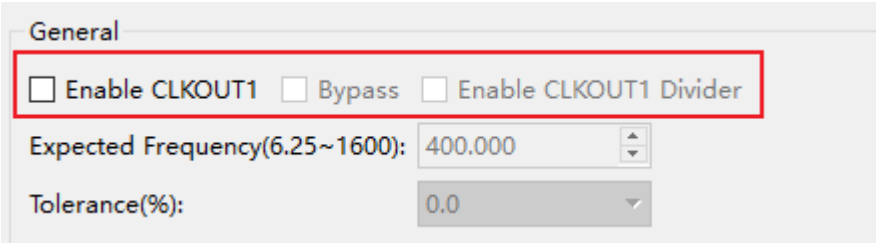


图 18-8 Clkout1~3 增加的选项配置

使能选项 Enable，这里我们需要勾选 Clkout1~3 的 Enable 选项，使能之后，对应页面的所有选项才可以选择，否则不能选择。选择“Bypass”，且选择“Enable CLKOUTx Divider”，则为 Bypass in 模式，反之不选 Enable CLKOUTx Divider，则为 Bypass out 模式。

我们依次配置 Clkout1 输出频率为 100M，相位偏移 90，Clkout2 输出频率 200M，Clkout3 输出频率为 200M，占空比微调 2 Step，配置界面如下图 18-9、图 18-10、图 18-11 所示。

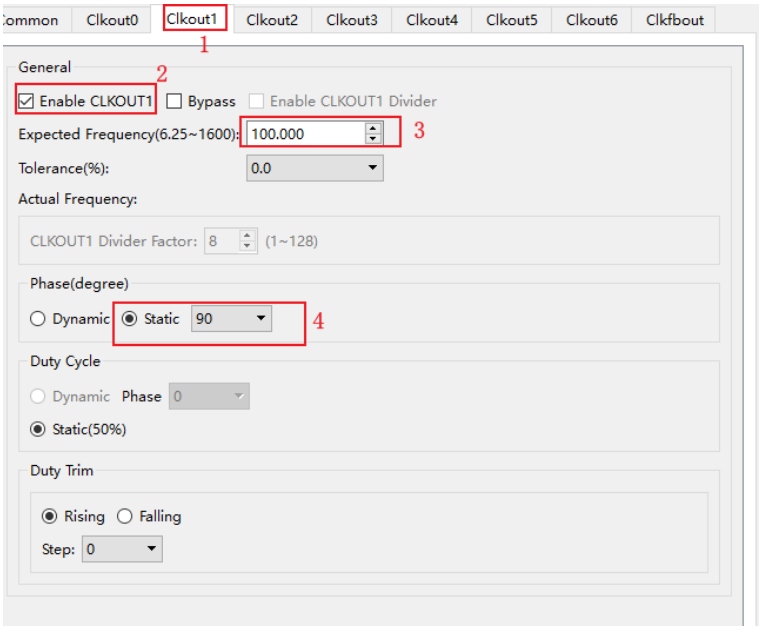


图 18-9 Clkout1 输出配置

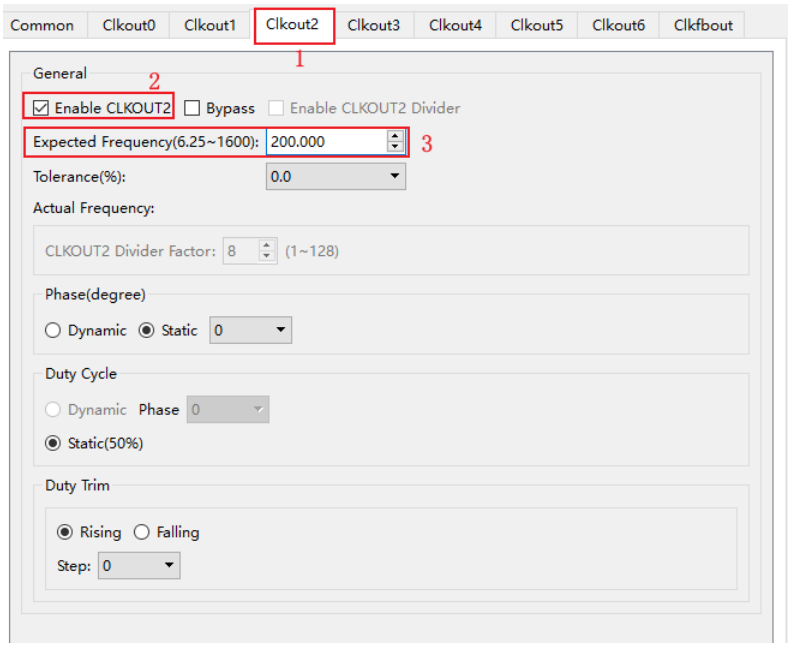


图 18-10 Clkout2 输出配置

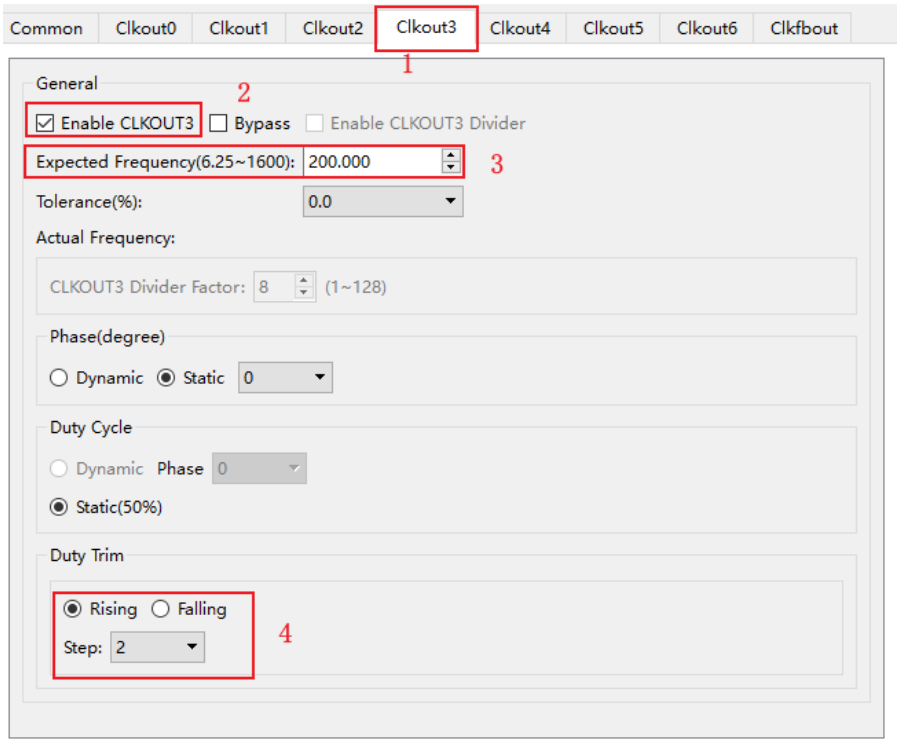


图 18-11 Clkout3 输出配置

按照上述配置完 PLL 之后，点击 OK，弹出如下图 18-12 所示的界面。点击 OK 之后，就完成了 PLL IP 生成成功。

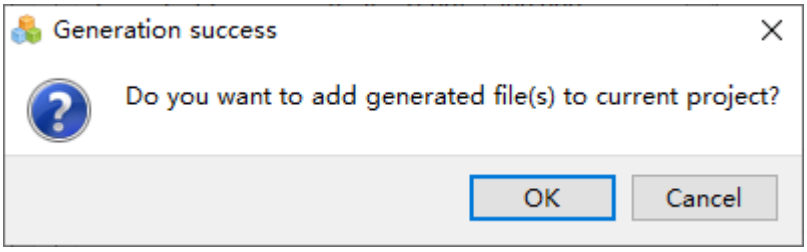


图 18-12 IP 生成成功示意图

随后弹出 PLL 的例化文件 gowin_pll_tmp.v，如下图 18-13 所示，读者使用时，将下面所示代码放置至自己文件中即可。

```
Gowin_PLL your_instance_name(
    .lock(lock_o), //output lock
    .clkout0(clkout0_o), //output clkout0
    .clkout1(clkout1_o), //output clkout1
    .clkout2(clkout2_o), //output clkout2
    .clkout3(clkout3_o), //output clkout3
    .clkfbout(clkfbout_o), //output clkfbout
    .clkin(clkin_i), //input clkin
    .reset(reset_i) //input reset
);
```

图 18-13 PLL 例化代码

对上图所示 PLL 的端口说明如下表 18-2 所示。

表 18-2 pll 端口说明表

端口名称	I/O	端口含义
lock_o	输出	PLL 锁定指示：1：锁定；0：失锁
clkout0	输出	0 通道时钟输出
clkout1	输出	1 通道时钟输出
clkout2	输出	2 通道时钟输出
clkout3	输出	3 通道时钟输出
clkfbout	输入	反馈时钟输入
clkin	输入	参考时钟输入
reset	输入	PLL 全部复位信号，复位数字电路，高电平有效

如果想要修改已建 IP 的参数，操作如下图 18-14 所示，然后就会弹出 IP 配置窗口，重新修改之后，保存，这样就修改成功了。

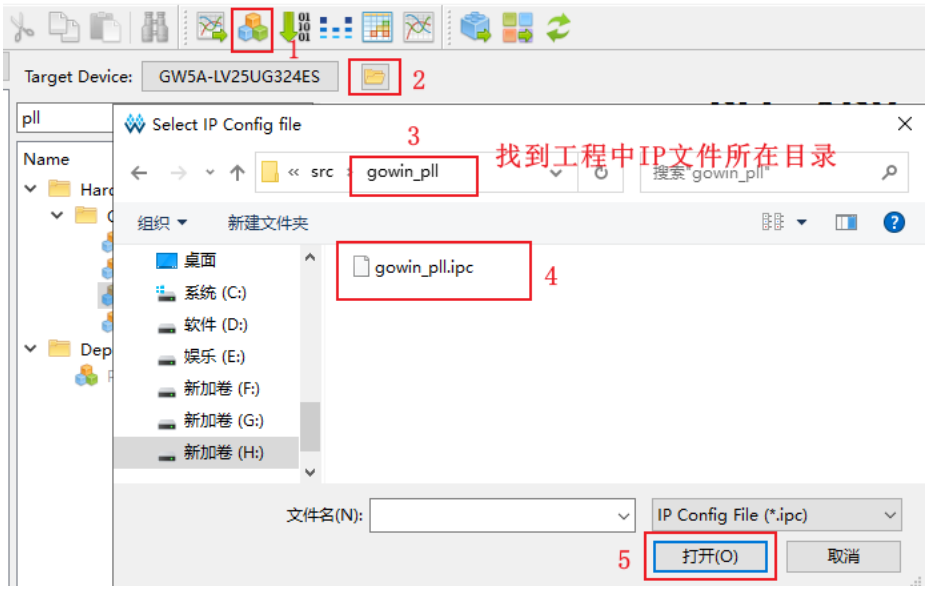


图 18-14 修改已建 IP 的参数操作示意图

通过上述操作，我们成功配置了 PLL IP 核，接下来我们便需要通过一个具体的设计，通过观察最终输出的时钟是否与预期一致，来验证 IP 的正确性。

18.4基于 PLL 的多时钟 LED 驱动设计

本次设计我们将 PLL 输出的 4 个不同的时钟，通过仿真，对比输出时钟波形间的关系，验证 PLL 的基础功能。同时，为了验证 PLL 输出的时钟能都稳定于其他模块，PLL 输出的时钟还将倍用于驱动 LED。

18.4.1LED 闪烁控制

基于“视觉暂留”现象，当人眼被中等强度的光刺激以后，人眼看到的图像会短暂停留 0.1~0.4 秒。而如果我们直接使用生成的时钟驱动 led 闪烁，其变化速率便会远远超过人眼的识别速度，因此，我们需要设计一个分频计数模块，对输入的时钟分频，控制 LED 的闪烁频率。

模块代码设计如下：

```
module led_flip_ctrl#(
    parameter CNT_MAX = 24'd10_000_000
)
(
    clk,
    rst_n,
    led
);

    input clk;
    input rst_n;
    output reg led;

    reg [27:0]cnt;

    always@(posedge clk or negedge rst_n)
    if(!rst_n)begin
        cnt <= 'b0;
        led <= 'b0;
    end
    else if(cnt >= CNT_MAX - 1'b1)begin
        cnt <= 'b0;
        led <= ~led;
    end
    else
        cnt <= cnt + 1'b1;

endmodule
```

可以看到，代码内容并不复杂，根据输入时钟 clk 的不同，我们可以通过修改计数最大值 CNT_MAX，来控制 LED 翻转速率，从而达到人眼可识别的变化频率。

18.4.2顶层设计

完成个模块设计后，在顶层中对模块以及 IP 例化，并连接端口，代码如下：

```
module pll_led(
```

```
    clk,
    rst_n,
    clk_100m,
    clk_100m_phase,
    clk_200m,
    clk_200m_duty,
    led
);

    input clk;
    input rst_n;

    output clk_100m;
    output clk_100m_phase;
    output clk_200m;
    output clk_200m_duty;
    output [3:0]led;

    wire locked;

    Gowin_PLL Gowin_PLL(
        .lock(locked), //output lock
        .clkout0(clk_100m), //output clkout0
        .clkout1(clk_100m_phase), //output clkout1
        .clkout2(clk_200m), //output clkout2
        .clkout3(clk_200m_duty), //output clkout3
        .clkfbout(), //output clkfbout
        .clkin(clk), //input clkin
        .reset(~rst_n) //input reset
    );

        led_flip_ctrl #(
        .CNT_MAX(26'd50_000_000)
    )
    led0(
        .clk(clk_100m),
        .rst_n(locked),
        .led(led[0])
    );

    led_flip_ctrl #(
        .CNT_MAX(26'd50_000_000)
    )
    led1(
        .clk(clk_100m_phase),
        .rst_n(locked),
        .led(led[1])
```

```
    );

    led_flip_ctrl #(
        .CNT_MAX(26'd50_000_000)
    )
    led2(
        .clk(clk_200m),
        .rst_n(locked),
        .led(led[2])
    );

    led_flip_ctrl #(
        .CNT_MAX(26'd50_000_000)
    )
    led3(
        .clk(clk_200m_duty),
        .rst_n(locked),
        .led(led[3])
    );

endmodule
```

顶层中，PLL 产生的四个时钟倍分别作为四个例化的 led_flip_ctrl 模块的输入时钟。PLL 输出的 locked 信号被作为 led_flip_ctrl 模块的复位信号，以确保模块在输出时钟稳定后才开始工作。

每个被例化的 led_flip_ctrl 模块，都通过顶层传参的方式将 CNT_MAX 的值设置为了 50_000_000。通过计算我们可以知道，在 100M 时钟作为驱动时钟时，LED 每 1S 完成一次亮灭；而当 200M 时钟作为驱动时钟时，LED 每 0.5S 完成一次亮灭。

18.4.3仿真测试

为了验证上述设计，接下来我们需要创建一个激励文件，为设计提供激励信号，观察波形。考虑到设计中控制 led 闪烁的计数器值过大，为了节省仿真时间，这里我们可以在激励文件中适当将 CNT_MAX 缩小为 26'd500。激励文件如下：

```
`timescale 1ns / 1ps
`define CLK_PERIOD 20

module pll_led_tb;
    reg clk;
    reg rst_n;
    wire [3:0]led;
```

```
    wire clk_100m;
    wire clk_100m_phase;
    wire clk_200m;
    wire clk_200m_duty;

    defparam pll_led.led0.CNT_MAX = 26'd500;
    defparam pll_led.led1.CNT_MAX = 26'd500;
    defparam pll_led.led2.CNT_MAX = 26'd500;
    defparam pll_led.led3.CNT_MAX = 26'd500;

    pll_led pll_led(
        .clk(clk),
        .rst_n(rst_n),
        .clk_100m(clk_100m),
        .clk_100m_phase(clk_100m_phase),
        .clk_200m(clk_200m),
        .clk_200m_duty(clk_200m_duty),
        .led(led)
    );

    initial clk = 1'b1;
    always #(`CLK_PERIOD/2) clk = ~clk;

    initial
        begin
        rst_n = 0;
        #201;
        rst_n = 1;
        #20000;
        @(posedge pll_led.locked)
       #20000;
        $stop;
    end
endmodule
```

然后打开 Modelsim，新建一个仿真工程，将工程所需的文件 gowin_pll、led_filp_ctrl、pll_led、pll_led_tb 添加至工程中，然后配置仿真环境，将 pll_led 的信号添加至 Wave 窗口观察，然后点击按钮，得到仿真完整波形图如下图 18-15 所示。

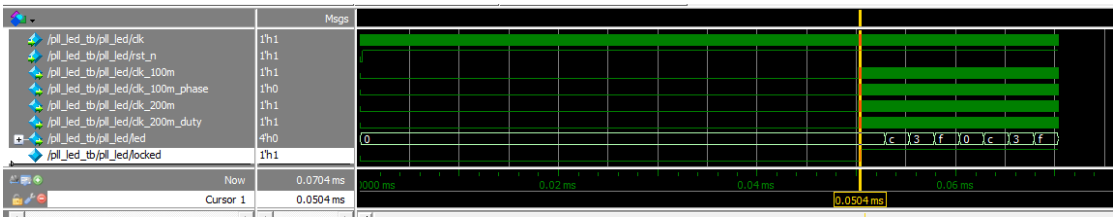


图 18-15 仿真完整波形图

设计中，我们使用 locked 作为分频计数模块复位，那么根据预期，系统的工作顺序应该为 rst_n 复位→pll 开始工作→pll 产生稳定时钟，locked 拉高→分频计数器开始工作→led 开始变化。

这里我们将 led0~led1 模块中的 cnt 计数添加进 Wave 窗口，重新跑一下仿真波形，然后观察波形图，通过观察信号分析模块工作的先后顺序是否如预期中一致，如下图 18-16 所示。

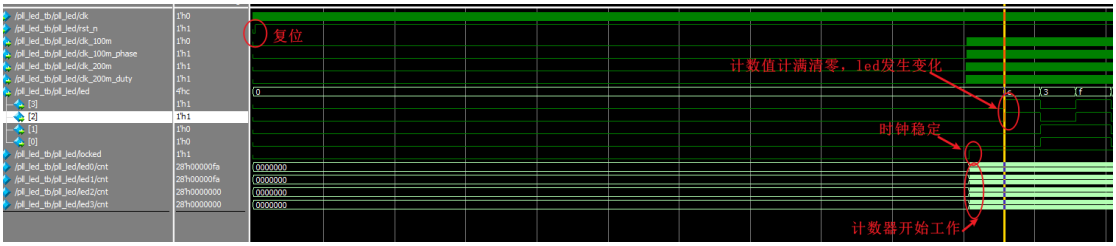


图 18-16 信号先后关系

可以看到，随着复位结束后，PLL 开始工作，经过一段时间开始输出时钟。此时的 locked 信号仍为低电平，因此 led_flip_ctrl 模块处于复位状态，计数器为 0。lock 监视器在对时钟监视一段时间后，确认时钟稳定且不再变化，locked 信号被置高，此时 led_flip_ctrl 模块复位结束，计数器开始工作。再经过一段时间后，计数器达到最大计数值被清零，led 状态发生改变。

因此，模块的工作顺序与预期中一致。同时，通过波形我们也可以很清楚的看到，locked 是在 PLL 输出时钟信号稳定一段时间后才被拉高。因此，在很多设计中，该信号被用于其他模块的复位信号，以确保系统能够在时钟稳定后再开始工作。

接下来观察输出时钟波形，首先是 clk_100m 和 clk_100_phase，其波形变化如下图 18-17 所示。

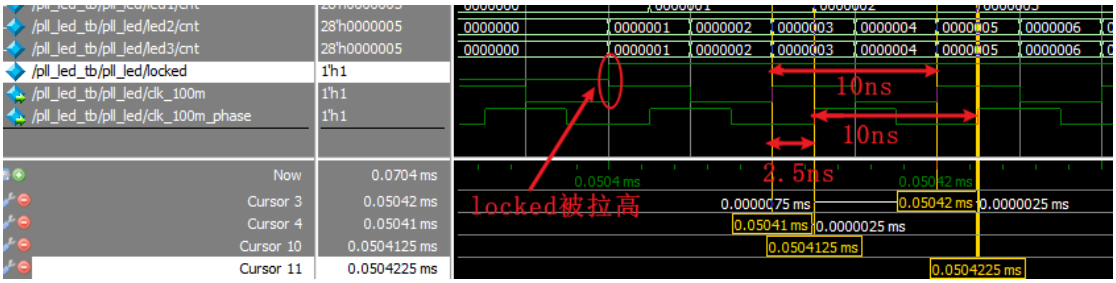


图 18-17 clk_100m 和 clk_100_phase 波形变化关系

可以看到两个时钟的时钟周期为 10ns，即说明这两个时钟的频率都是 100MHz。clk_100m_phase 波形整体要比 clk_100m 滞后 2.5ns，对应的刚好是 1/4

个时钟周期，也就是 90 度相位。这两个时钟波形与预期中一致。

接下来是 clk_200m 和 clk_200_duty 两个时钟，其波形如图 18-18 所示。

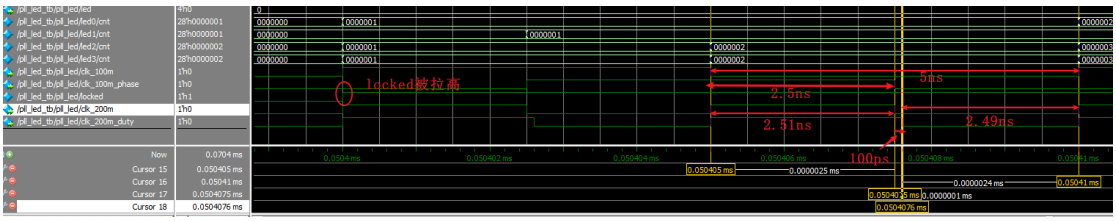


图 18-18 clk_200m 和 clk_200_duty 波形变化关系

可以看到，clk200m 和 clkk_200m_duty 的时钟周期为 5ns，即对应时钟频率为 200MHz。clk_200m 时钟在单个时钟周期内，高电平与低电平的持续时间都为 2.5ns，即占空比为 50。clk_200m_duty 时钟在单个时钟周期内高电平持续时间为 2.51ns，低电平持续时间为 2.49ns，及占空比下降沿微调 2 step，也就是高电平持续时间增加 100ps，与预期一致。

最后是 led 的翻转，led[0]和 led[1]的翻转过程波形如图 18-19 和图 18-20 所示。

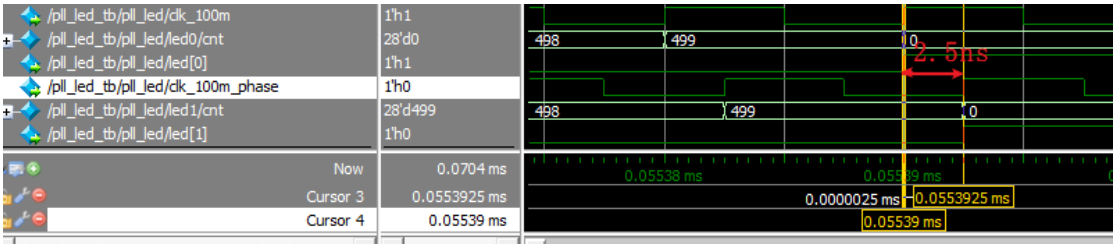


图 18-19 led[0]和 led[1]上升沿变化过程

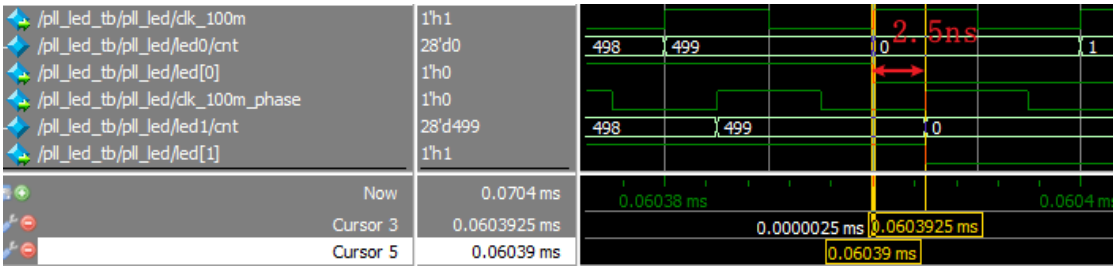


图 18-20 led[0]和 led[1]下降沿变化过程

可以看到，计数器每计数到 499 就会清零，并翻转 led。由于相位差，led[0]会比 led[1]早 2.5ns 变化（这里是驱动 led[0]的 clk_100m 时钟在 locked 有效时，上升沿刚好过来，控制 led0 的计数器开始计数，而控制 led0 的计数器将经过 2.5ns 之后才开始计数），然而，对于人眼来说，7.5ns 的变化实在是过于迅速，根本不可能看得见，所以，如果人以人眼观测只能看到 led[0]和 led[1]同时亮灭。

led[2]和 led[3]的翻转过程波形如下图 18-21 和图 18-22 所示。

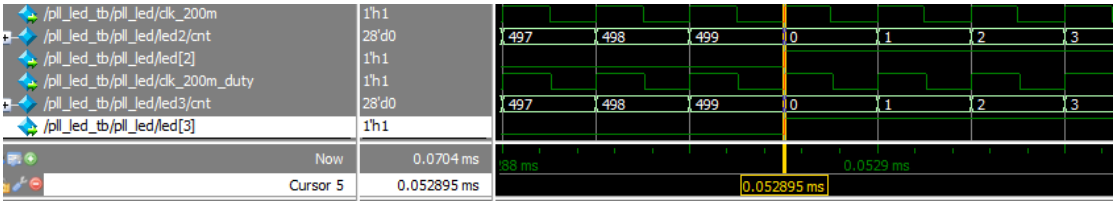


图 18-21 led[2]和 led[3]上升沿变化过程

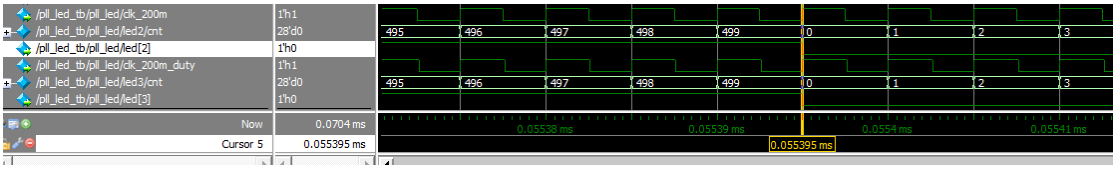


图 18-22 led[2]和 led[3]下降沿变化过程

可以看到，计数器计数到 499 后清零并翻转 led，led[2]和 led[3]同时翻转变化。由于 led[2]和 led[3]的驱动时钟是 led[0]和 led[1]的两倍，因此，对应的 led[2]和 led[3]的变化频率也是 led[0]和 led[1]的两倍。

至此，仿真波形分析完毕，PLL 能够成功输出与需求的频率、相位、占空比一致的时钟。且该时钟能够被正常用于其他模块的驱动。

18.4.4板级验证

虽然通过仿真我们已经验证了 PLL 的基础功能，但是在实际的应用场合中，PLL 所生成的时钟能否正常用于其他模块，仍需要我们烧录到实际的开发板中验证。为此，我们需要先为设计分配管脚。

18.4.4.1管脚约束

本次验证所选择的硬件平台为高云开发板。本次设计的引脚约束表如表 18-3：

表 18-3 引脚分配表

Pin Name	Signial Name	Pin NO.	Pin Name	Signial Name	Pin NO.
CLK_G	clk	T9	LED1	led[1]	C14
KEY0	rst_n	B16	LED2	led[2]	B9
LED0	led[0]	D14	LED3	led[3]	A9

分配完引脚后约束电平为 LVCMOS33，随后保存设计，生成 bit 流，等待 bit 生成完成，便可以连接硬件准备烧录验证了。

18.4.4.2系统所需硬件

1. 高云开发板 x1

2. 高云下载器 x1
3. DC 电源线 x1

18.4.4.3硬件连接

本次设计硬件连接如图 18-23 所示：

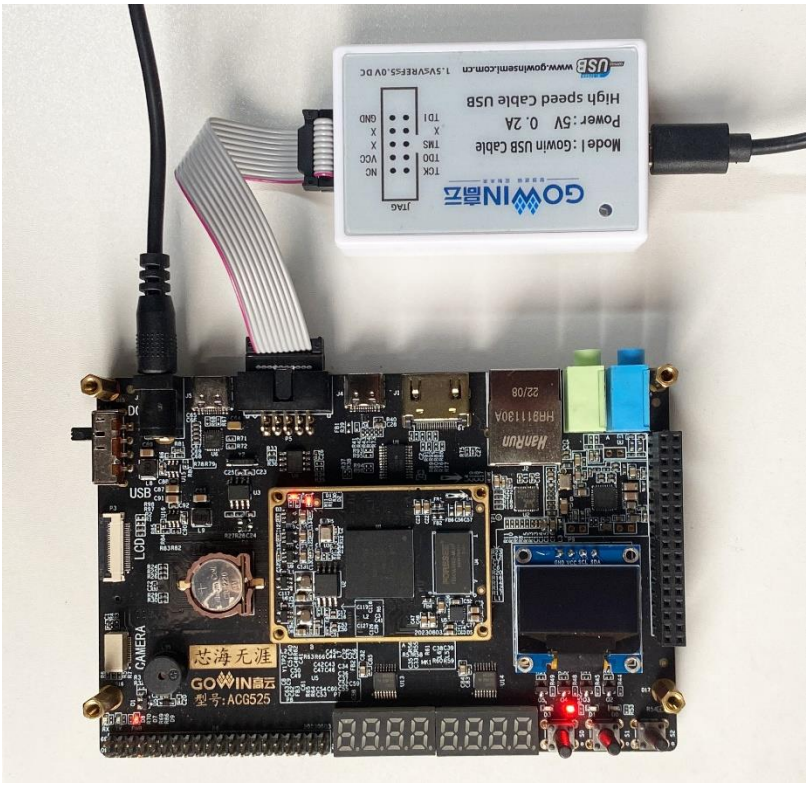


图 18-23 硬件连接图

连接完成后，将电源拨码开关拨到对应供电侧，接下来将生成好的 bit 烧录到开发板中。

18.4.4.4板级现象

烧录到开发板后，led 很快便开始变化，D2 和 D3 按照 0.5s 的频率闪烁，D1 和 D0 按照 1s 的频率闪烁。

由于板级验证的结果与我们仿真时推测的一致，说明 PLL 产生的时钟稳定，且能被用于其他模块。

18.5总结

本章我们学习了 Gowin PLL IP 的配置，并通过具体的设计，带大家了解了

PLL 的一些基本使用。在 FPGA 设计中，PLL 属于十分常用的资源，常常被用来产生不同频率的时钟。而在后续章节的学习过程中，也会经常使用到 PLL，因此，建议读者能够跟随教程完成本章设计，了解并熟悉 PLL 的配置和使用。

19 基于 AD9767 高速 DAC 的 DDS 信号发生器设计

工程源码	|----02_设计实例 |---- ch19_dds_ad9767
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本节将介绍基于 ACM9767 模块的 DDS 数字信号发生器原理及设计方法，通过本节课程的学习，读者能够对基于高速 DA 转换器的数字信号发生器设计有更深入的理解。

19.1DDS 概述

DDS(Direct Digital Synthesizer)即数字合成器，是近年来发展起来的一种新的频率合成技术，其主要优点是相对带宽很大，频率转换时间极短（可小于 20 ns)，频率分辨率很高，全数字化结构便于集成，输出相位连续可调，且频率、相位和幅度均可实现程控。随着 FPGA 技术的不断发展，该技术得到愈加成熟的应用。借助 FPGA 平台开发的高性能的 DDS 发生器与基于 DDS 芯片设计的信号发生器相比，具有成本更低，操作更加灵活，而且还能根据要求在线更新配置等诸多优点。同时，整个系统开发流程更趋于软件化、自定义化，开发周期更短，调试过程更加简单。

正是由于其便捷的频率、相位以及幅度的数控优势，基于 FPGA 控制的 DDS 信号发生器有其广泛的应用前景。

19.2ACM9767 模块概述

提到基于 FPGA 的 ACM9767 模块波形信号发生器设计，就不得不先对 ACM9767 模块进行简单的介绍。

ACM9767 模块使用的是 ADI 公司 AD9767 型 DAC 芯片。该芯片支持 I、 Q 输出模式（该模式常用于数字通信领域）。 输出形式为差分电流输出，输出电流满量程范围为可设置为 2~20mA。芯片本身自带 1.2V 的参考电压，无需外部提供参考源。

继承了 AD9767 芯片的优异性能，ACM9767 模块是一款高性能高速双通道

DAC 模块。 模块具有单电源 5V 供电输入，双通道数字转模拟信号输出，每个通道数据分辨率为 14 位， 输出电压范围为+-5V， 且转换速率高达 125Msps，非常适合诸如信号发生器、数字调制通信系统的开发等应用。

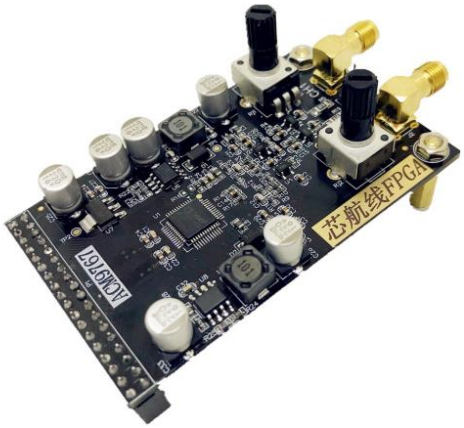


图 19-1 ACM9767 模块样图

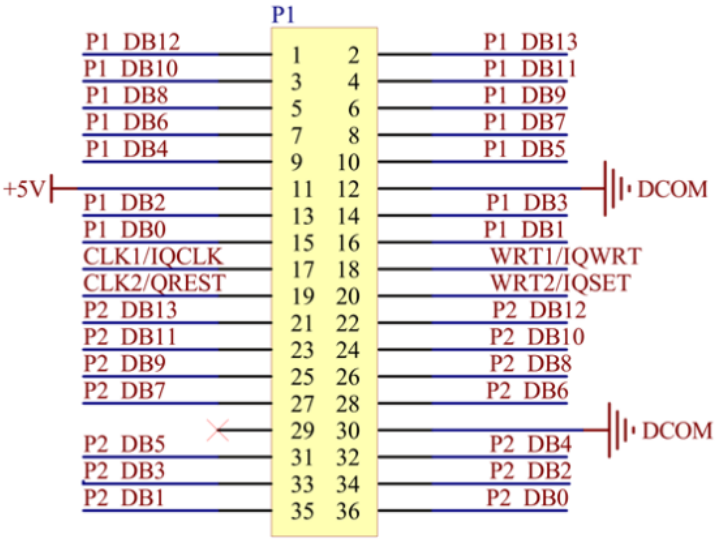


图 19-2 ACM9767 模块插接管脚信号定义

模块对用户提供一个 36 针的排母接口，可直接插接到市面上各种常见的 FPGA 开发板（Terasic 公司 DE2、DE1、 DE0，芯路恒科技（小梅哥 FPGA） 所有的 FPGA 开发板，包含 ACX720、AC620、 AC6102、 AC601、AC609 等）上进行使用，而无需使用任何转接线转接。

本次系统设计将在小梅哥团队出品的高云开发板上进行。在使用时，将 ACM9767 模块插接到高云开发板的通用 GPIO 接口上，即可实现 FPGA 和 ACM9767 模块的数字接口互通。由于在 ACM9767 模块模拟量输出峰值范围内，输出的模拟量电压值可以由输入的数字量信号完全控制，这样，就可以通过

FPGA 完全控制数字量的输出值，实现满足预定指标的模拟信号波形输出。

基于此，借助 ACM9767 模块和高云开发板实现：输出波形可以是不同频率、幅度的正弦波、三角波、方波信号，同时用户可以根据实际需求通过按键对波形的频率和相位值进行调整切换这一设计功能，是完全可行的。

接下来，我们将从典型的 DDS 信号发生器信号发生流程图，着手以上设计功能的实现。

19.3系统原理及整体设计

19.3.1DDS 信号发生器的系统设计原理

介绍完 DDS 信号发生的应用背景和 ACM9767 的模块硬件性能，接下来我们来共同分析和探讨基于 FPGA 的 DDS 信号发生器的实现过程。

DDS 是如何实现控制发生信号的呢？如果想生成一个质量较好的 DDS 发生信号，无非是对波形、频率、相位这几个关键要素进行控制和搭配组合。下面是其中一种常用的 DDS 控制信号发生器的实现方案原理图。

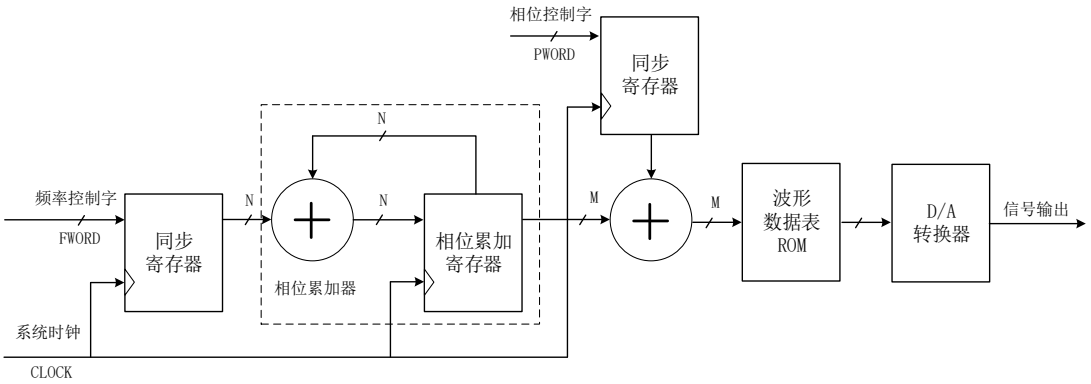


图 19-3 DDS 控制信号发生原理图

由图可以看出，DDS 主要由相位累加器、相位调制器、波形数据表以及 D/A 转换器构成。其中相位累加器由 N 位加法器与 N 位寄存器构成。每个时钟周期的时钟上升沿，加法器就将频率控制字与累加寄存器输出的相位数据相加，相加的结果又反馈至累加寄存器的数据输入端，以使加法器在下一个时钟脉冲的作用下继续与频率控制字相加。这样，相位累加器在时钟作用下，不断对频率控制字进行线性相位累加。即在每一个时钟脉冲输入时，相位累加器便把频率控制字累加一次。

相位累加器输出的数据就是合成信号的相位。相位累加器的溢出频率，就

是 DDS 输出的信号频率，相位累加器输出的数据，作为波形存储器的相位采样地址，这样就可以把存储在波形存储器里的波形采样值经查表找出，完成相位到幅度的转换。

波形存储器的输出数据送到 D/A 转换器，由 D/A 转换器将数字信号转换成模拟信号输出。

DDS 信号流程示意图如下图 19-4 所示：

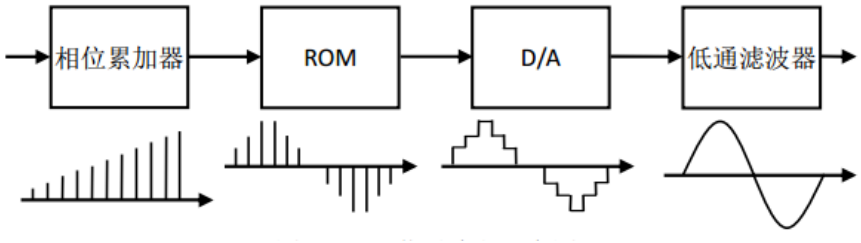


图 19-4 DDS 信号流程示意图

这里相位累加器位数为 N 位（N 的取值范围实际应用中一般为 24~32），相当于把正弦信号在相位上的精度定义为 N 位，所以其分辨率为$1/2^N$。

若 DDS 的时钟频率为F_{clk}，频率控制字 fword 为 1，则输出频率为$F_{out}=\frac{F_{clk}}{2^N}$，这个频率相当于“基频”。若 fword 为 B，则输出频率为$F_{out}=B\times\frac{F_{clk}}{2^N}$。

因此理论上由以上三个参数就可以得出任意的f_o输出频率。且可得出频率分辨率由时钟频率和累加器的位数决定的结论。当参考时钟频率越高，累加器位数越高，输出频率分辨率就越高。

从上式分析可得，当系统输入时钟频率F_{clk}不变时，输出信号频率由频率控制字 B 所决定，由上式可得：$B=2^N\times\frac{F_{out}}{F_{clk}}$。其中 B 为频率字且只能取整数。为了合理控制 ROM 的容量，此处选取 ROM 查询的地址时，可以采用截断式，即只取 32 位累加器的高 M 位。这里相位寄存器输出的位数一般取 10~16 位。

19.3.2DDS 信号发生器原理讲解举例

以上通过理论计算加数据变换的形式对 DDS 原理进行了较为严谨的公式推导解释，但是如果从采样点选取的角度分析，DDS 究竟是怎么实现频率和相位的控制的呢？以下通过一个简化的实例来描述 DDS 实现频率和相位控制的过程。

如下图 19-5 为一个完整周期的正弦信号的波形，总共有 33 个采样点，其中第 1 点和第 33 点的值相同，第 33 点为下一个周期的起始点，因此，实际一个周

期为 32 个采样点（1~32）。

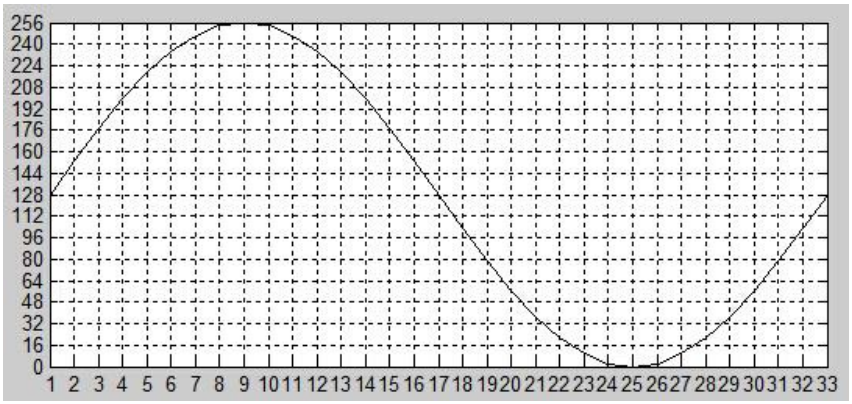


图 19-5 完整周期的正弦信号波形示例

例如：当使用 FPGA 控制 DAC 输出一个周期的正弦信号时，每 1ms 输出一个数值。如果每个点都输出，则总共输出这一个完整的周期信号需要输出 32 个点，因此输出一个完整的信号需要 32ms，可知输出信号的频率为 1000/32Hz。

如果需要用这一组数据来输出一个 2*（1000/32）Hz 的正弦信号，因为输出信号频率为 2*（1000/32）Hz，那么输出一个完整的周期的正弦波所需要的时间为 32/2，即 16ms。为了保证输出信号的周期为 16ms，我们需要对我们的输出策略进行更改，上面输出周期为 32ms 的信号时，我们采用的为逐点输出的方式，以 32 个点来输出一个完整的正弦信号，而我们 FPGA 控制 DAC 输出信号的频率固定为 1ms。因此，我们要输出周期为 16ms 的信号，只能输出 16 个点来表示一个完整的周期。我们就选择以每隔一个点输出一个数据的方式来输出即可。我们可以选择输出（1、3、5、7……29、31）这些点，因为采用这些点，我们还是能够组成一个完整周期的正弦信号，而输出时间缩短为一半，即频率提高了一倍。最终结果如下图 19-6 所示：

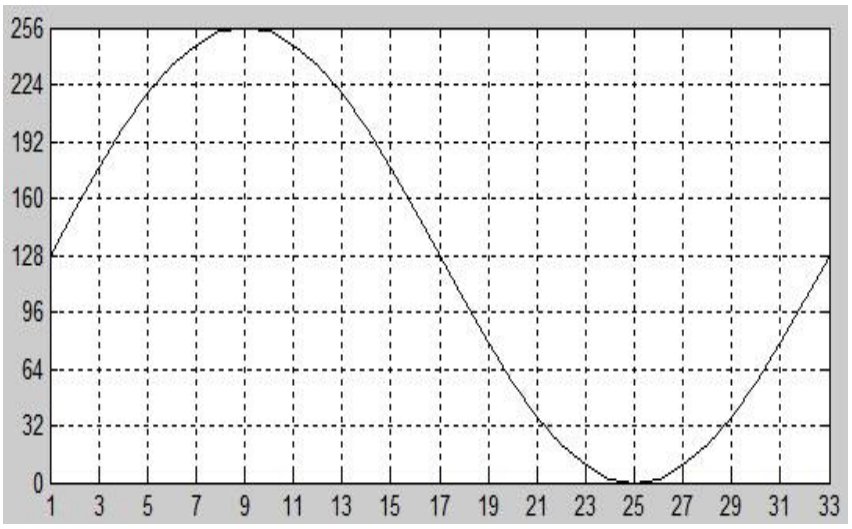


图 19-6 频率提高 1 倍后的正弦输出信号

如果需要使用该组波形数据输出频率为(1/2)*（1000/32）Hz 的信号，即周期为 64ms，则只需要以此组数据为基础，每 2ms 输出一个数据即可，例如第 1ms 和第 2ms 输出第一个点，第 3ms 和第 4ms 输出第二个点，以此类推，第 63ms 和第 64ms 输出第 32 个点，即可实现周期加倍，即频率减半的效果。

概括来说，一个周期的离散点数值表格一定，如果想提高输出频率，则采用跳过若干点位（1,3,5.....）的方法，重构原波形；如果想降低输出频率，则采用拉长每一个离散点的输出时间(1,1,2,2,3,3.....)，来完成输出波形不变，而扩大时钟周期的输出效果。时钟周期扩大，则频率降低。

对于相位的调整，则更加简单。只需要在每个取样点的序号上加上一个偏移量，便可实现相位的控制。例如，上述一个周期 32 个点均匀的将 360 度等分，上面默认的是第 1ms 时输出第一个点的数据，假如我们现在在第 1ms 时从第 9 个点开始输出，则将相位左移了 90 度，这就是控制相位的原理。

在本次设计中，对于一个周期的离散点数值表格，我们实现 DDS 输出时，将横坐标上的数据作为 ROM 的地址，纵坐标上的数据作为 ROM 的输出，那么指定不同的地址就可实现对应值的输出。而我们 DDS 输出控制频率和相位，归结到底就是控制 ROM 的读出地址。

在理解了 DDS 的实现原理之后，接下来便可以开始本次系统的设计实现了。

19.3.3信号发生控制方案设计框图

介绍完 DDS 信号发生的原理，并进行了举例后，接下来将进行 DDS 信号发生器的整体设计。

根据以上原理分析，基于高云开发板，我们可以作如下架构设计

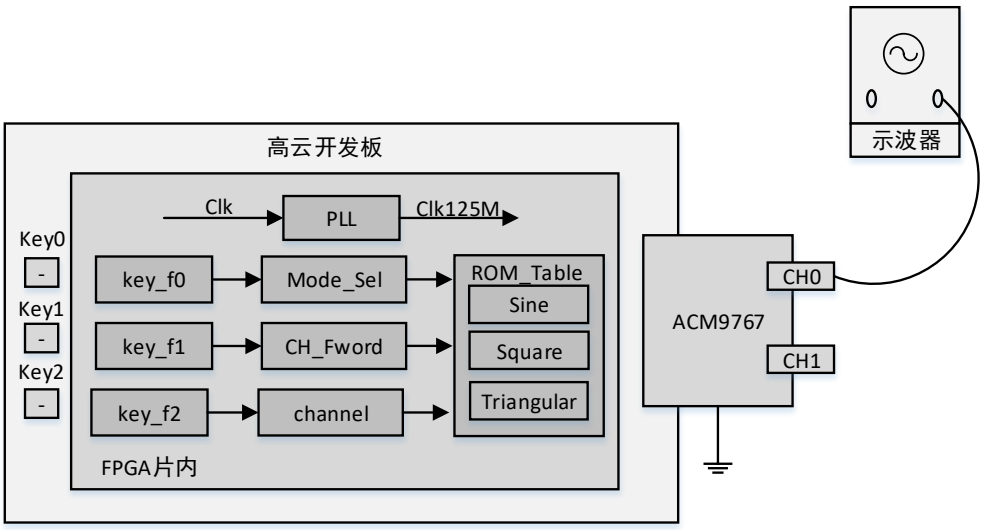


图 19-7 基于高云开发板的 ACM9767 模块系统框图

为了验证频率和相位可以在一段区间内实现变化这一特性，我们可以通过开发板上的按键按下切换档位的方式，设定几个可以跳变切换的典型值以验证频率相位的实时变化效果。但是，由于高云开发板上的按键有限，所以本次实验，我们设计时可以先按如下规律实现设计：

1. 程序上电后给出默认频率和相位初始值。
2. 每当按下一次按键，切换一种波形输出。
3. 每当按下一次按键扩大频率 10 倍。
4. 每当按下一次按键切换一个通道，然后针对该通道设置其输出波形类型和频率。

开发板上的按键用途说明如下表 19-1 所示。由于我们开发板上只有 3 个按键，按键个数比较少，所以本次实验我们将不会通过按键控制输出波形的相位，将会在程序中固定相位值。

表 19-1 按键用途设计说明表

按键名称	按键用途
S0	输出信号波形切换
S1	输出信号频率切换
S2	通道切换

在曾经的学习内容中，我们也介绍了按键消抖的相关理论和使用场景。在本工程中，由于同样涉及到人工按下按键属于随机触发的外部信号的情况，我们同样需要加入按键消抖的相关处理模块来解决按键抖动的问题。

明确了频率和相位控制字切换的设计方案后，还需要明确波形信号的存储

和读取策略。为了便于波形数据取用的统一管理，对于 3 种不同类型的波形数据存储，我们可以为每个信号发生通道开辟 3 个 ROM 空间。这三个 ROM 空间分别存储正弦波、方波和三角波的离散信号值作信号发生时查表使用。

在设计之初，设计人员也许有过这样的思考：我能否只开辟一块 ROM 空间，然后输出哪种波形，就装载哪种波形的 ROM 表？这样做可以节省 FPGA 片上的 ROM 空间资源，让更多 ROM 空间资源预留给其他需要进行缓存的设计单元。

但是，这样做会产生如下问题：熟悉 FPGA 的 ROM IP 核的用户都知道，切换 ROM 表的装载文件，即 mi 文件，不是一件容易的事情，每切换一次 mi 文件，都需要对 ROM IP 核进行 mi 文件的重新装载，并重新编译 IP 核和工程。这样看来，要想让已经完成编译并下载到 FPGA 开发板内的工程进行实时的发生信号波形切换，就变成了难以完成的任务。所以，从波形实时切换和选择的角度，只有并行开辟三块 ROM 空间并加载 mi 文件，然后同时读取波形数据，并在最终输出端进行输出波形数据选择，才能实现信号的实时切换功能。

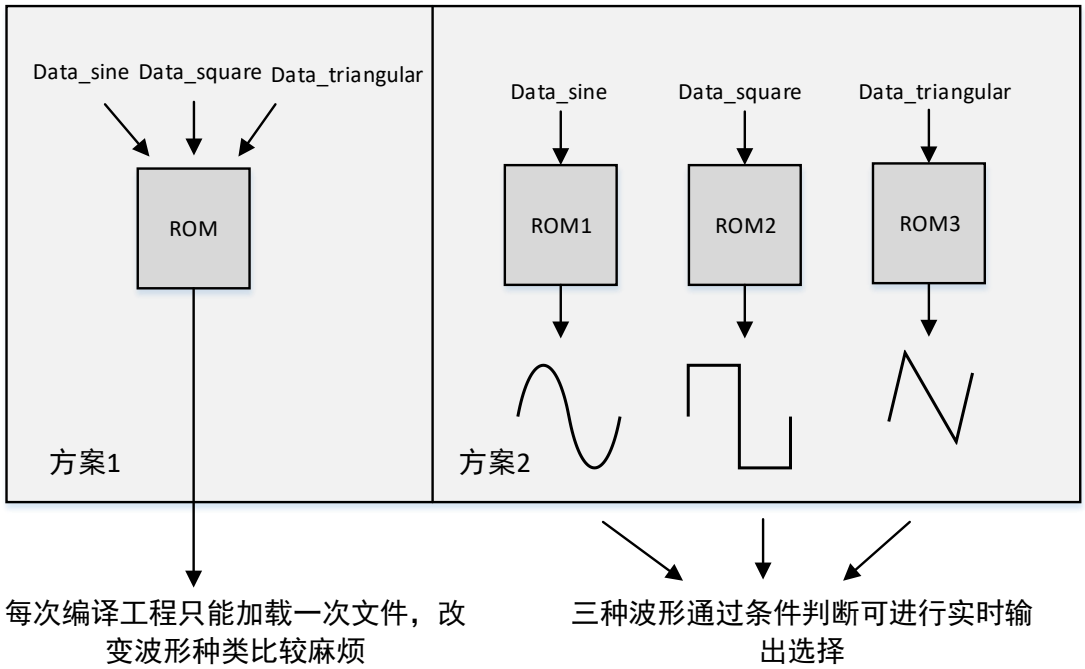


图 19-8 方案 1 和方案 2 模块架构对比分析

通过以上分析比较发现，使用方案 2 比使用方案 1 虽然占用更多的 ROM 资源，但输出波形的实时响应性能更优秀，所以在 ROM 资源充沛的情况下，优先选用方案 2。

19.4系统各模块设计

在有了系统总体框架思路后，接下来即可着手对模块各个击破。经过上述分析，我们在本次设计中需要利用按键消抖模块解决按键按下抖动的问题，同时，我们需要配置锁相环模块、信号离散值ROM表、按键控制模块以及对DDS驱动模块进行设计。由于按键消抖模块已经有专门的章节进行讲解，在这里，受篇幅所限，我们也不再赘述，而将讲解重点放在如何设计 ACM9767 驱动模块及几个重要 IP 核的配置之中。

19.4.1锁相环模块

由于 AD9767 的工作时钟频率为 125MHz，所以这里我们需要利用 FPGA 内部的锁相环为 ACM9767 模块以及其 FPGA 配套的驱动模块提供相同的工作频率。这样，在数据交互时，能确保时钟域相同。

锁相环的配置方法我们也讲过多次，我们也就不再进一步介绍。这里，我们给出配置界面供参考即可。

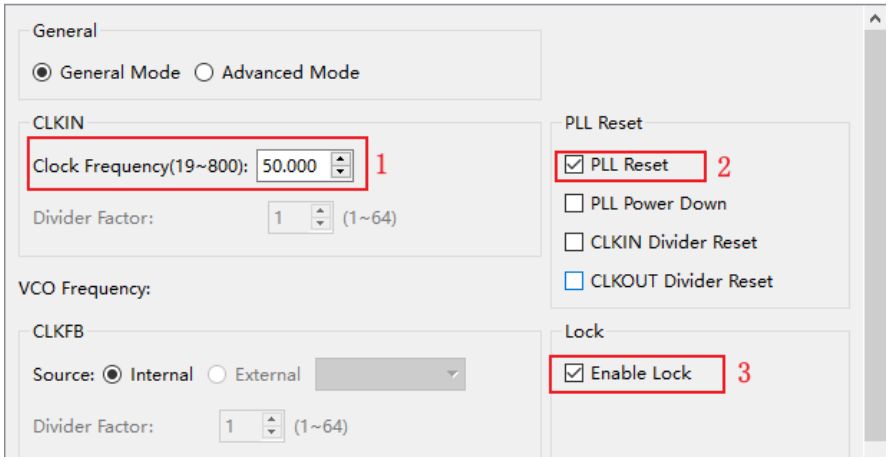


图 19-9 锁相环配置界面 1

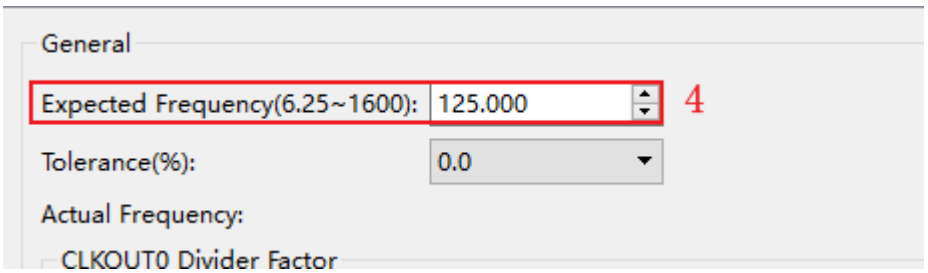


图 19-10 锁相环配置界面 2

完成配置后，即可利用锁相环将 50M 主时钟频率，通过分频倍频而获得我

们需要的 125MHz 时钟。

19.4.2波形数据存储单元

为了能产生相应的波形，我们需要借助 ROM 来分别存储正弦波、方波、三角波的波形数据。其中单端口的 ROM 主要配置数据如下图 19-11 所示，其初始化文件选为已经生成的相应波形的 mi 文件。此处 mi 的位宽为 14，深度为 4096。

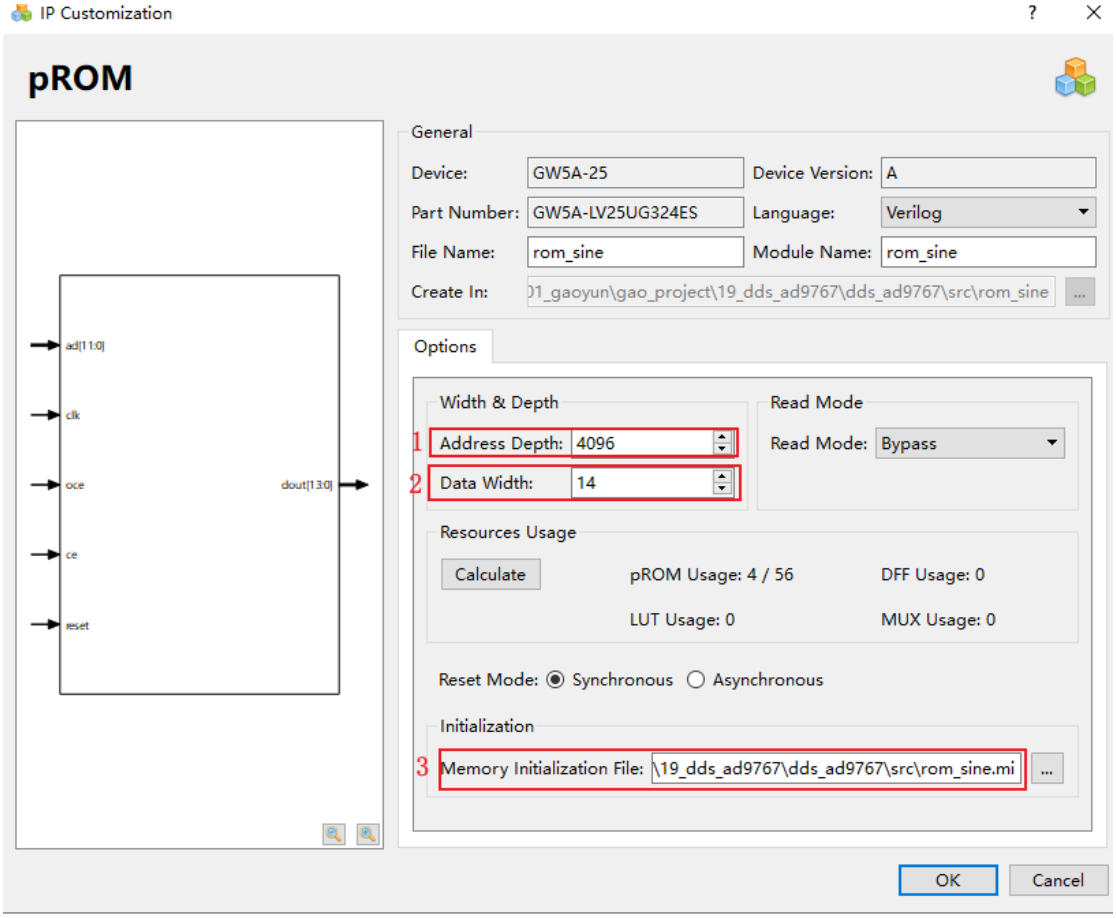


图 19-11 波形数据存储 ROM 配置表

关于 mi 文件的生成可以参看“IP 核使用之 ROM”一节的内容。同时，由于 AD 芯片的数字位宽为 14 位，所以 maxi 中数据改为 16383（即 2^14 - 1）可以确保 FPGA 输出的二进制值，都能反映到输出模拟量实时输出值的变化之中，其余设置如下图所示，然后剩下的按照“IP 核使用之 ROM”一节的内容进行修改，使其匹配高云的 ROM IP。

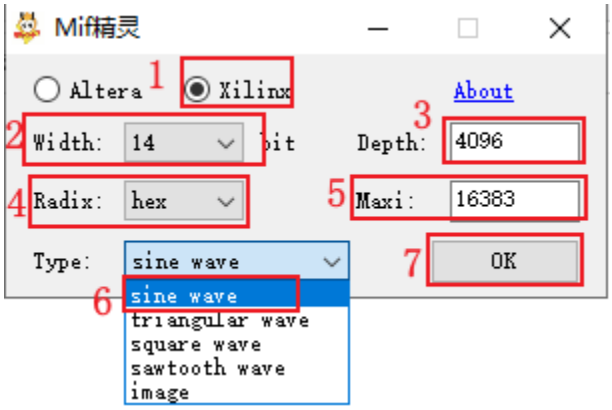


图 19-12 使用 Mif 精灵生成波形信号参考图示

19.4.3ACM9767 驱动模块设计

在本设计中参考时钟F_{clk}频率为 125MHz，相位累加器位数 N 取 32 位，频率控制字位数 M 取 12 位。经过以上的分析，可以得出 DDS 模块的端口示意图如下图 19-13 所示。

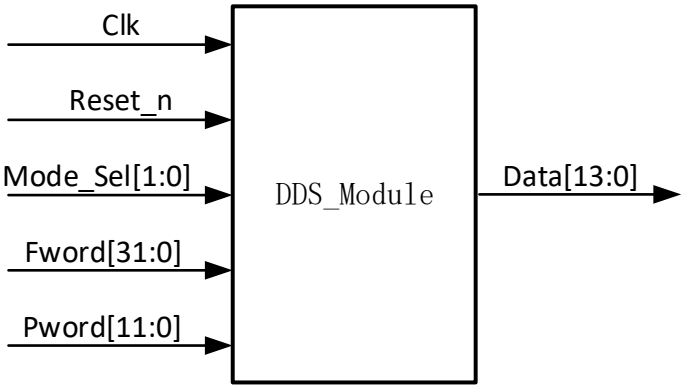


图 19-13 ACM9767 驱动模块端口

其中，每个端口的功能描述如下所示。

表 19-2 ACM9767 驱动模块端口列表

端口名称	I/O	功能描述
Clk	I	为本模块的工作时钟，频率为锁相环输出的 125MHz
Rst_n	I	控制器复位，低电平复位
Model_Sel	I	波形选择
FWord	I	频率控制字
PWord	I	相位控制字
Data	O	数据输出

为了研究简便，我们以双通道信号发生器的其中一个通道作为研究对象，而另一个通道与之完全相同。实际在 FPGA 驱动 ACM9767 工作过程中，FPGA 设计上只需重复例化驱动模块一次，在管脚绑定时只需按双通道对应的数字信

号驱动管脚驱动外部 ACM9767 模块即可。

接下来，开始驱动模块的代码设计讲解。

为了便于对频率和相位控制字的后期处理，在驱动模块之中必须先对频率和相位值进行缓存。其代码如下：

```
//频率控制字同步寄存器
reg [31:0]Fword_r;
always@(posedge Clk)
Fword_r <= Fword;

//相位控制字同步寄存器
reg [11:0]Pword_r;
always@(posedge Clk)
Pword_r <= Pword;
```

实现波形数据的还原，其实质就是按频率控制字每个时钟周期进行累加，进而决定下一个时钟周期应该便宜多少个周期切分的单元，从而作为下一次还原的数据。

```
//相位累加器
reg [31:0]Freq_ACC;
always@(posedge Clk or negedge Reset_n)
if(!Reset_n)
    Freq_ACC <= 0;
else
    Freq_ACC <= Fword_r + Freq_ACC;
```

需要注意的是，在该模块设计中我们直接截取 32 位累加器结果中的高 12 位作为 ROM 的查询地址，这样，查询地址也是 4096 个点，和 ROM 表保存的单周期的离散点存储深度是一致的。这样产生的误差虽然会对频谱纯度有影响，但是对波形的精度的影响是可以忽略的。

```
//波形数据表地址
reg [11:0]Rom_Addr;
always@(posedge Clk)
        Rom_Addr <= Freq_ACC[31:20] + Pword_r;
```

由于本设计有两个输出通道，所以将该模块例化两次，以实现双通道的输出。

本工程设计共创建了 3 种典型的发生信号模型，如果希望输出哪一种波形，则进行对应的模式选择即可。从这个角度来看，每个通道的 3 块 ROM 空间，实际上都在同时读取数据和输出数据，至于最终呈现在 FPGA 输出端的到底是何种波形的数据，则由模式选择信号 Mode_Sel 决定。

```
always@(*)
    case(Mode_Sel)
```

```
        0:Data = Data_sine;
        1:Data = Data_square;
        2:Data = Data_triangular;
        3:Data = 8192;
endcase
```

这样，完成上述设计后，我们以一个通道为例的 ACM9767 驱动模块设计就完成了。

19.4.4按键控制模块设计

按键控制模块的主要功能：通过按键 S0 控制输出波形类型，按键 S1 控制输出波形频率，按键 S2 控制通道切换，通过按键 S2 切换到对应通道上时，然后通过 S0 和 S1 去切换输出波形相关参数，并通过不同的 LED 灯闪烁代表不同切换到哪个通道上，LED0 闪烁代表切换到通道 CH1，LED1 闪烁代表通道 CH2。该模块的基本结构如下图 19-14 所示：

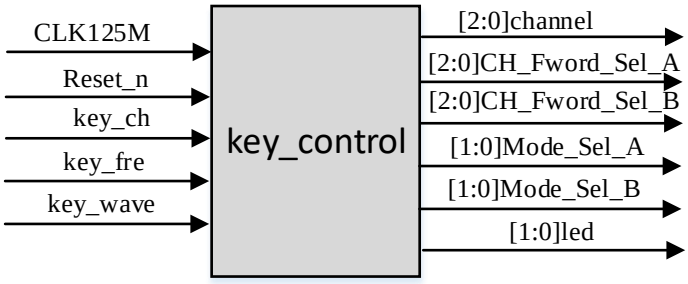


图 19-14 按键控制模块结构框图

其中，每个端口的信号说明如下表 19-3 所示。

表 19-3 按键控制模块端口列表

端口名称	I/O	功能描述
CLK125M	I	为本模块的工作时钟，频率为锁相环输出的 125MHz
Reset_n	I	控制器复位，低电平复位
key_ch	I	通道切换按键，对应开发板上的 S2
key_fre	I	频率切换按键，对应开发板上的 S1
key_wave	I	波形切换按键，对应开发板上的 S0
channel	O	设置的通道：0：代表 CH1；1：代表 CH2
CH_Fword_Sel_A	O	通道 1 的频率控制参数
CH_Fword_Sel_B	O	通道 2 的频率控制参数
Mode_Sel_A	O	通道 1 的波形选择参数
Mode_Sel_B	O	通道 2 的波形选择参数
led	O	LED 灯：LED0 闪烁：设置通道 1；LED1 闪烁：设置通道 2

首先例化三个按键控制模块，这个模块在前面章节已经介绍了，本章将不再进行说明，直接使用即可，代码如下所示：

```
wire [2:0]Key_Flag;
wire [2:0]Key_State;

//按键 S0
key_filter key_filter0(
    .Clk(CLK125M),
    .Reset_n(Reset_n),
    .Key(key_wave),
    .Key_Flag(Key_Flag[0]),
    .Key_State(Key_State[0])
);

//按键 S1
key_filter key_filter1(
    .Clk(CLK125M),
    .Reset_n(Reset_n),
    .Key(key_fre),
    .Key_Flag(Key_Flag[1]),
    .Key_State(Key_State[1])
);

//按键 S2
key_filter key_filter2(
    .Clk(CLK125M),
    .Reset_n(Reset_n),
    .Key(key_ch),
    .Key_Flag(Key_Flag[2]),
    .Key_State(Key_State[2])
);
```

然后设置按键 S0 的功能，当 channel 为 0 时，设置的是通道 1 的输出波形类型，由于本次设计只有正弦波、方波和三角波三种类型可以选择，所以当按键 S0 按下之后，Mode_Sel_A 依次递增，当 Mode_Sel_A 大于等于 2 时，从 0 开始计数，这样我们就可以通道通道 1 依次输出正弦波、方波和三角波，当 channel 为 1 时，此时控制的就是通道 2 的输出波形，代码如下所示：

```
//按键 S0 控制波形
always@(posedge CLK125M or negedge Reset_n)
if(!Reset_n) begin
  Mode_Sel_A <= 2'd0;
  Mode_Sel_B <= 2'd0;
end
else if(Key_Flag[0] && (Key_State[0] == 0)) begin
  if(channel == 3'd0) begin
      Mode_Sel_A <= Mode_Sel_A + 1'd1;
      if(Mode_Sel_A >= 2'd2)
          Mode_Sel_A <= 2'd0;
```

```
  end
  else if(channel == 3'd1) begin
      Mode_Sel_B <= Mode_Sel_B + 1'd1;
      if(Mode_Sel_B >= 2'd2)
          Mode_Sel_B <= 2'd0;
  end
end
else begin
  Mode_Sel_A <= Mode_Sel_A;
  Mode_Sel_B <= Mode_Sel_B;
end
```

随后设置按键 S1 的功能，该按键设置的输出波形的频率，实现方式 S0 按键控制功能一致，当 channel 为 0 时，控制 CH_Fword_Sel_A 的值依次递增，由于 CH_Fword_Sel_A 的位宽为[2:0]，所以本次设计我们准备 8 种输出频率，这 8 种输出频率具体是多少将在后面的内容进行介绍，当 channel 为 1 时，控制 CH_Fword_Sel_B 的输出，代码如下所示：

```
//按键 S1 控制频率
always@(posedge CLK125M or negedge Reset_n)
if(!Reset_n) begin
    CH_Fword_Sel_A <= 3'd0;
    CH_Fword_Sel_B <= 3'd0;
end
else if(Key_Flag[1] && (Key_State[1] == 0)) begin
    if(channel == 3'd0)
        CH_Fword_Sel_A <= CH_Fword_Sel_A + 1'd1;
    else if(channel == 3'd1)
        CH_Fword_Sel_B <= CH_Fword_Sel_B + 1'd1;
end
else begin
    CH_Fword_Sel_A <= CH_Fword_Sel_A;
    CH_Fword_Sel_B <= CH_Fword_Sel_B;
end
```

最后是设置按键 S2 的功能，该按键控制的按键的切换，一共有 2 个通道可以选择，所以当 channel 大于 2 时，清零重新开始计数，这样就实现了通道 0 和通道 1 切换，代码如下所示：

```
//按键 S2 控制通道 0 还是通道 1
always@(posedge CLK125M or negedge Reset_n)
if(!Reset_n)
    channel <= 3'd0;
else if(Key_Flag[2] && (Key_State[2] == 0))
    channel <= channel + 1'd1;
else if(channel >= 3'd2)
    channel <= 3'd0;
```

按键功能介绍完毕之后，就剩下 LED 灯的控制，这是为了便于读者可以了解到当前控制的是哪个通道，LED0 闪烁，代表此时控制的是 CH1，LED1 闪烁，代表此时控制的是 CH2，代码如下所示：

```
always @ (posedge CLK125M or negedge Reset_n)
begin
if (!Reset_n)
    led <= 2'd0;
else if (cnt == 32'd62499999) begin
    if(channel == 2'd0) begin
        led[0] <= ~led[0];
        led[1] <= 1'd1;
    end
    else if(channel == 2'd1) begin
        led[1] <= ~led[1];
        led[0] <= 1'd1;
    end
end
else
    led <= led;
end
```

至此按键控制模块设计完成。

19.4.5工程顶层模块设计

在工程顶层，我们实现了各子模块的例化，我们在前面讲解设计原理分析的内容时，已经讲解了频率控制字的理论推导计算方法。这里我们总共输出 8 种频率，分别是 1hz、10hz、100hz、1khz、10khz、100khz、1Mhz、10Mhz，根据 CH_Fword_Sel_A 和 CH_Fword_Sel_B 的值，输出不同的频率控制字，代码如下所示：

```
//频率控制字
//如果把周期完整的一个波形等分成 2 的 32 次方份，在时钟频率为 125M 次/秒的条件下，
always@(posedge CLK125M)
    case(CH_Fword_Sel_A)
        0:Fword_A = 34;//2**32 / 1250000000;     34.35
        1:Fword_A = 344;//2**32 / 125000000;
        2:Fword_A = 3436;//2**32 / 12500000;
        3:Fword_A = 34360;//2**32 / 1250000;
        4:Fword_A = 343597;//2**32 / 125000;
        5:Fword_A = 3435974;//2**32 / 12500;
        6:Fword_A = 34359738;//2**32 / 1250;
        7:Fword_A = 343597384;//2**32 / 125;
    endcase
```

```
always@(posedge CLK125M)
    case(CH_Fword_Sel_B)
        0:Fword_B = 34;//2**32 / 1250000000;     34.35
        1:Fword_B = 344;//2**32 / 125000000;
        2:Fword_B = 3436;//2**32 / 12500000;
        3:Fword_B = 34360;//2**32 / 1250000;
        4:Fword_B = 343597;//2**32 / 125000;
        5:Fword_B = 3435974;//2**32 / 12500;
        6:Fword_B = 34359738;//2**32 / 1250;
        7:Fword_B = 343597384;//2**32 / 125;
       endcase
```

为了尽可能将波形细分，我们将单周期波形细分的份数，设定为 2 的 32 次方即 4,294,967,296 份。而时钟频率是 125MHz，则说明每一个时钟周期上升沿到来，得递进累加（4,294,967,296/125000000）份数，即算出得到一个基准的频率控制字。以此，如果频率控制字扩大 10 倍，则相同时间内出现的波形数量，也扩大 10 倍，以此，从代码的角度，就形成了以上频率控制字的查找表。

相位控制我们本次实验没有按键可以控制其切换，都是固定的相位，但是我们也将对相位控制进行说明，相对而言，相位控制字的设计更加简单。我们单周期波形的存储深度为 4096，将 4096 等分后与 360 度角度等分对应，即可得到每个等分角度的变化值，代码如下所示：

```
always@(*)
    case(CH_Pword_Sel)
        0:Pword = 0;   //0
        1:Pword = 341; //30
        2:Pword = 683; //60
        3:Pword = 1024;    //90
        4:Pword = 1707;    //150
        5:Pword = 2048;    //180
        6:Pword = 3072;    //270
        7:Pword = 3641;    //320
     endcase
```

最后是关于数码管的显示，使数码管上显示正在设置的通道的输出波形频率，如下所示：

```
//数码管显示频率
always@(posedge CLK125M)
    case(channel)
        0: begin
                case(CH_Fword_Sel_A)
                    0: disp_data = 32'h0000_0001;
                    1: disp_data = 32'h0000_0010;
                    2: disp_data = 32'h0000_0100;
```

```
                    3: disp_data = 32'h0000_1000;
                    4: disp_data = 32'h0001_0000;
                    5: disp_data = 32'h0010_0000;
                    6: disp_data = 32'h0100_0000;
                    7: disp_data = 32'h1000_0000;
                endcase
            end
        1: begin
                case(CH_Fword_Sel_B)
                    0: disp_data = 32'h0000_0001;
                    1: disp_data = 32'h0000_0010;
                    2: disp_data = 32'h0000_0100;
                    3: disp_data = 32'h0000_1000;
                    4: disp_data = 32'h0001_0000;
                    5: disp_data = 32'h0010_0000;
                    6: disp_data = 32'h0100_0000;
                    7: disp_data = 32'h1000_0000;
                endcase
            end
    endcase

hc595_driver hc595_driver(
  .clk(clk_50M),
  .reset_n(Reset_n),
  .data({1'd1,seg,sel}),
  .s_en(1'b1),
  .sh_cp(sh_cp),
  .st_cp(st_cp),
  .ds(ds)
);

hex8 hex8(
  .clk(clk_50M),
  .reset_n(Reset_n),
  .en(1'b1),
  .disp_data(disp_data),
  .sel(sel),
  .seg(seg)
);
```

hc595_driver 和 hex8 模块在数码管章节已经进行过说明了，本章实验将直接调用这两个模块，如果读者对这两个模块的设计不是很清除，请参看数码管相关章节的内容。

到这里，我们的顶层模块也设计完成。

19.5激励创建及仿真测试

19.5.1激励信号设计

由于在板级验证时，需保证每一次按下按键都能切换通道、频率以及波形类型，而我们在仿真时，更关注波形的整体输出效果，所以本次仿真将忽略按键的功能，只看最终输出的波形，所以仿真时将 key_control 模块的 channel、CH_Fword_Sel_A、CH_Fword_Sel_B、Mode_Sel_A、Mode_Sel_B 相关信号注释掉，在顶层通过 assign 语句直接赋值，如下所示：

```
key_control key_control
(
    .CLK125M(CLK125M),
    .Reset_n(Reset_n),
    .key_ch(key_ch),
    .key_fre(key_fre),
    .key_wave(key_wave),
//        .channel(channel),
//        .CH_Fword_Sel_A(CH_Fword_Sel_A),      //仿真时注释
//        .CH_Fword_Sel_B(CH_Fword_Sel_B),
//        .Mode_Sel_A(Mode_Sel_A),
//        .Mode_Sel_B(Mode_Sel_B),
    .led(led)
);
    //仿真时使用
assign CH_Fword_Sel_A = 3'd4;
assign Mode_Sel_A = 2'd0;
assign CH_Fword_Sel_B = 3'd4;
assign Mode_Sel_B = 2'd1;
assign channel = 3'd0;
```

我们可以直接通过 PLL 的锁定信号，控制两个通道的复位状态。很多读者就会考虑为什么不用 tb 仿真文件去控制复位信号呢，这里需要注意的是，因为锁相环也是受复位信号的控制，当处于复位的时候，将会没有时钟输出，这样会导致 DDS_Module 模块中的 Rom_Addr 信号输出为未知态，从而导致最终输出的数据 Data 为未知态，当复位完成后，Rom_Addr 也会一直处于未知态，波形如下图 19-15 所示。从图中可以看出，当 Fword_r 更新之后，Freq_ACC 的值为未知态加上 Fword_r 的值，也就一直为未知态，从而导致最终无输出数据为未知态。并且实际使用时按键不够，所以直接使用锁相环的锁定信号作为复位。

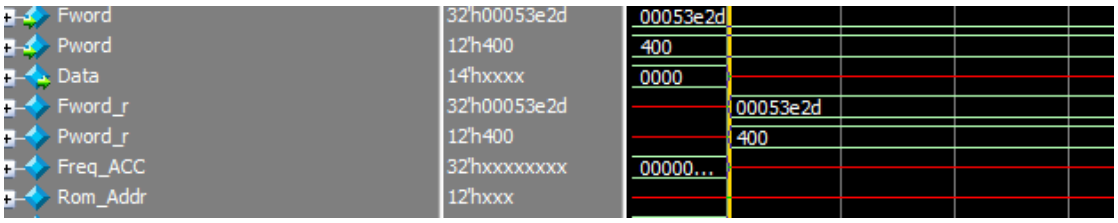


图 19-15 仿真输出为高阻态

由于默认的频率控制字给出的控制频率是 1Hz，这就表明如果想看到一个周期的完整波形，就需要让 Modelsim 绘制出 1 秒的完整波形。而对于 Modelsim 软件来说，绘制 1 秒钟的仿真波形，对于这个涉及到 ROM 数据读取的设计工程来说，是非常缓慢的。所以我们顶层直接设置 CH_Fword_Sel_A 和 CH_Fword_Sel_B 的值为 3'd4，方便我们观测波形变化。并且使 Mode_Sel_A 为 0，Mode_Sel_B 为 1，也就是通道 1 输出正弦波，通道 2 输出方波。

仿真 tb 文件可以作如下设计：

```
`timescale 1ns / 1ps

module DDS_AD9767_tb;

  reg Clk;
  reg  Reset_n;
  reg  key_ch;
  reg  key_fre;
  reg  key_wave;

wire [1:0] led;
wire sh_cp;
wire st_cp;
wire ds;
wire  [13:0]DataA;
wire  [13:0]DataB;
wire  ClkA;
wire  WRTA;
wire  ClkB;
wire  WRTB;

GSR GSR(.GSRI(1'b1));

  dds_ad9767 dds_ad9767(
      .clk_50M(Clk),
      .key_ch(key_ch),
      .key_fre(key_fre),
      .key_wave(key_wave),
      .led(led),
```

```
        .sh_cp(sh_cp),
        .st_cp(st_cp),
        .ds(ds),
        .DataA(DataA),
        .ClkA(ClkA),
        .WRTA(WRTA),
        .DataB(DataB),
        .WRTB(WRTB),
        .ClkB(ClkB)
    );
    initial Clk = 1;
    always #10 Clk = ~Clk;

    initial begin
        key_ch = 0;
        key_fre = 0;
        key_wave = 0;
        #1000000;
        $stop;
    end

endmodule
```

19.5.2仿真结果分析

根据以上仿真代码，可以得到如下图 19-16 仿真波形，这里我们对波形进行了分组，同时我们对关键信号进行监视，以有利于观察波形数据输出结果。

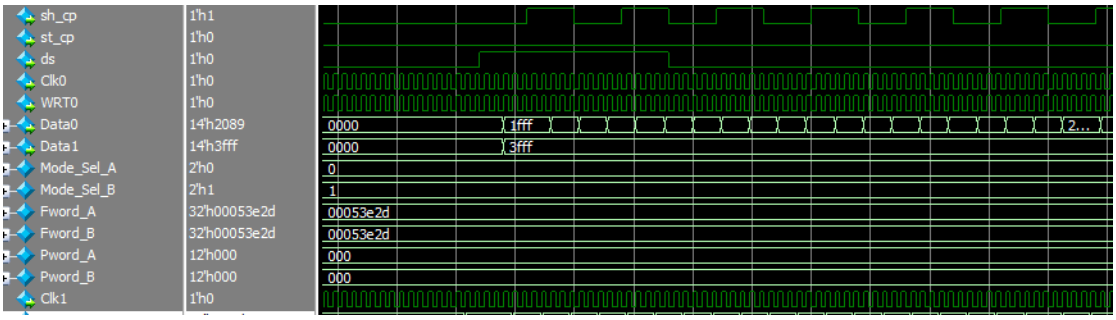


图 19-16 仿真波形总览

默认来说，软件给出的输出结果都是数字量，而对本实验来说，则是让仿真软件输出模拟量波形会更加直观展现我们的设计意图。Modelsim 仿真工具具备将数字量转换成模拟量的能力。这里我们只需要鼠标右键点击信号名称，然后切换波形输出类型即可。

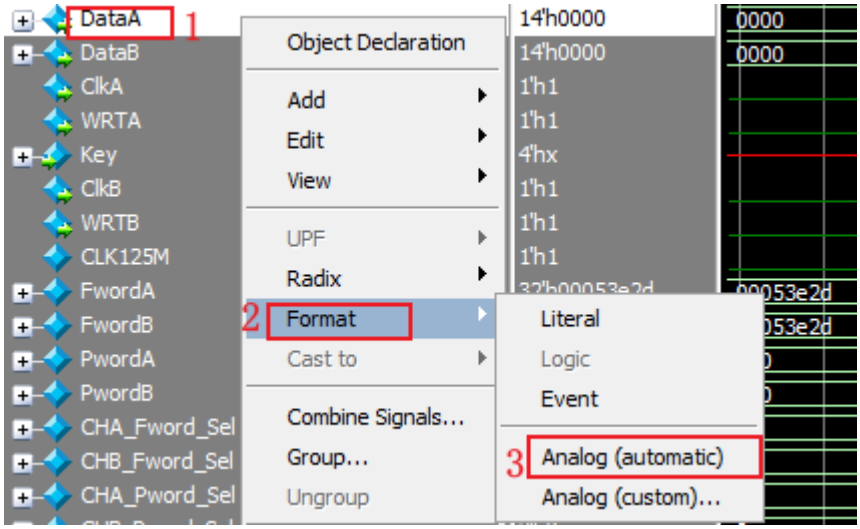


图 19-17 波形的数字量输出转换成模拟量输出

对于通道 1，经过转换后，可以看到，输出周期为 0.1ms 的正弦波，对于通道 2 输出 0.1ms 的方波，波形图如下图 19-18 所示。

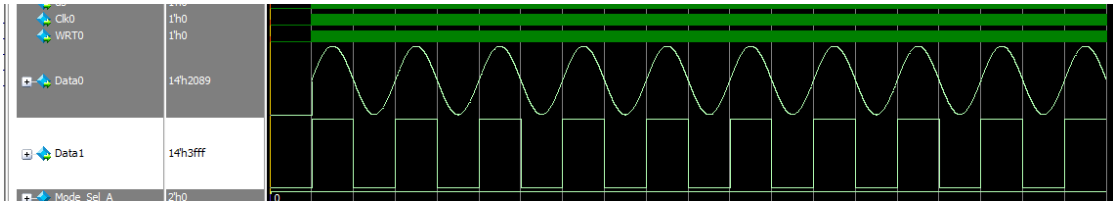


图 19-18 通道输出波形效果

19.6板级验证

19.6.1实验目标

本实验的板级验证环节，主要验证以下三个目标：

1. 能否正确的烧写和下载程序。
2. 下载并配置好输出波形后，能否在示波器上观察到期望的波形。
3. 按下设计好的频率、通道、波形类型相关按键，示波器上是否能发生频率、通道、波形类型等相关变化。

19.6.2系统所需硬件

系统所需硬件如下：

1. 高云开发板
2. 电源线一根

3. 下载器及其配套 xianlan 一套
4. ACM9767 模块
5. 示波器一台

19.6.3硬件连接

本次实验系统硬件方面连接十分简单，只需给高云开发板连接好电源线并将 ACM9767 模块连接到高云开发板的扩展接口上即可。ACM9767 模块的 CH1 与 CH2 通道则会按照用户的设置输出相应的波形，将通道与示波器相连即可看到产生的波形。当所有连接工作完成后将开发板的电源开关拨到 DC 侧，本次系统设计开发板连接方法如图 19-19 所示。

图 19-19 硬件连接图

19.6.4管脚绑定

前面我们也在整体设计阶段规划了按键控制功能。这里，在经过一系列设计与验证后，我们可以进一步深化确定硬件功能和相关参数。按设计，我们利用开发板上按键 S0 来控制输出波形类型，S1 控制输出频率，S2 控制通道切换，具体功能如下表 19-4 所示。

表 19-4 按键控制表

按键/拨码开关	功能描述
S0	控制通道 1 和 2 输出的波形，00 为正弦波，01 为方波，10 为三角波

S1	调节 ACM9767 通道 1 和 2 的输出频率，通过按键可依次在 1Hz，10Hz，100Hz，1kHz，10kHz，100kHz，1MHz，10MHz 频率中切换
S2	控制通道切换，按下之后切换到不同的通道进行频率和波形的切换

在高云开发板上挂载 ACM9767 模块，其管脚绑定对应关系如下表 19-5 所示：

表 19-5 管脚绑定表

功能信号	高云开发板	功能信号	高云开发板
Data0 [13]	C18	Data1[13]	L12
Data0 [12]	C17	Data1 [12]	L13
Data0 [11]	G14	Data1 [11]	H17
Data0 [10]	F14	Data1 [10]	H18
Data0 [9]	D18	Data1 [9]	K17
Data0 [8]	D17	Data1 [8]	K18
Data0 [7]	G18	Data1 [7]	L17
Data0 [6]	G16	Data1 [6]	L18
Data0 [5]	F18	Data1 [5]	N17
Data0 [4]	F17	Data1 [4]	N18
Data0 [3]	H14	Data1 [3]	P17
Data0 [2]	H13	Data1 [2]	P18
Data0 [1]	K16	Data1 [1]	P15
Data0 [0]	K15	Data1 [0]	H18
Clk0	J13	Clk1	K12
WRT0	K14	WRT1	K13
key_ch	C15	key_fre	A15
key_wave	B16	sh_cp	F4
st_cp	F3	ds	H4
led[0]	D14	led[1]	C14
clk_50M	T9		

19.6.5下载与验证

连接好硬件之后将开发板上的拨码开关拨到 DC 侧，将我们提供好的工程源码下载至开发板中即可从示波器中看到相应的波形。

下载完成后在示波器上显示的默认波形如下图 19-20 所示。

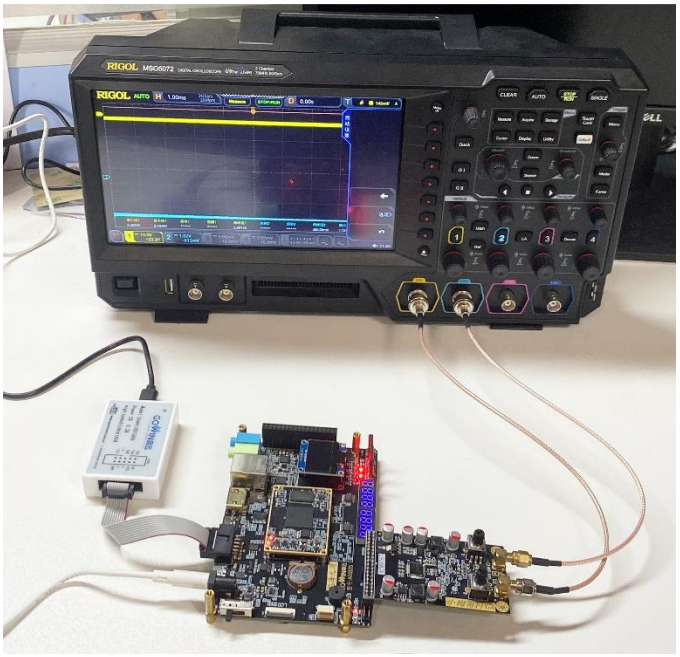


图 19-20 输出信号波形

由于我们默认的基准波形为 1Hz 的正弦波，而示波器对于如此低频的信号很难识别观察，所以品目暂时显示一条横线也是正常现象。

在随后的实验中，我们可以根据设计按下不同的按键，先按下 S1 切换 CH1 输出的频率，然后按下 S2 切换到 CH2，然后按下 S1 切换 CH2 输出的频率，改变两个通道的输出频率之后，示波器上显示波形如下图 19-21 所示。

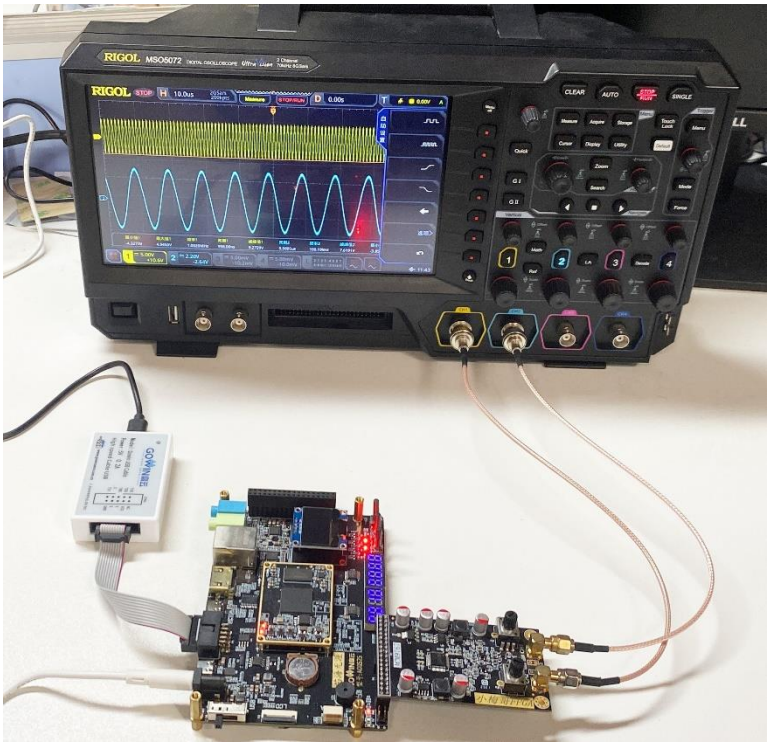


图 19-21 按下 S1、S2 切换 CH1 和 CH2 输出频率

然后我们按下按键 S0，切换输出信号波形，如下图 19-22 所示。

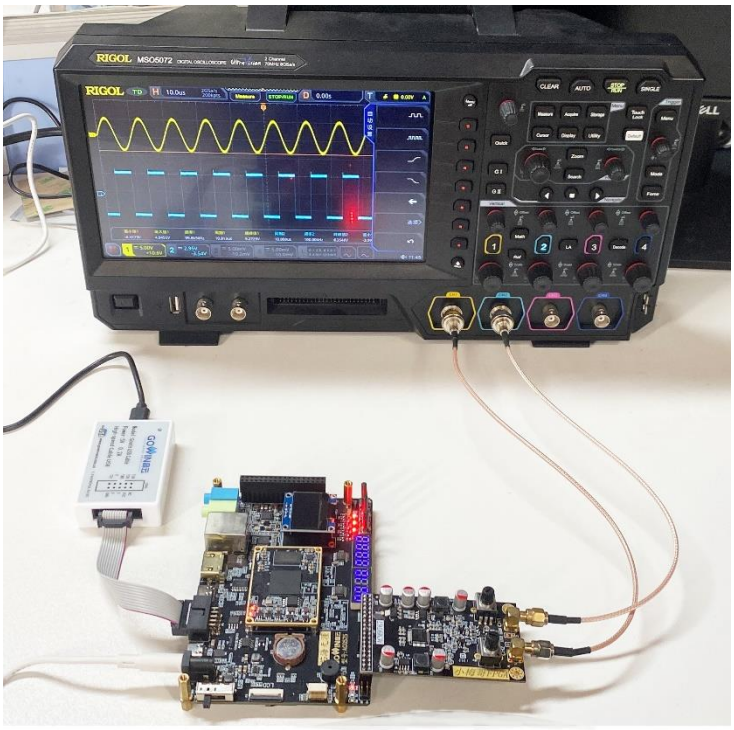


图 19-22 切换输出信号波形

通过上述板级验证，可以看到最终输出结果与我们设计需求一致，说明本次实验成功。

19.7常见问题与思考总结

1. 输出波形的类型，是通过 Mode_Sel 信号实现的。为了实现输出波形的类型可选择，必须为每个通道开辟 3 个 rom 空间，用于存放代表 3 种不同类型的波形数值。
2. 本实验以高云开发板搭载 ACM9767 模块作为开发模型，以按键的跳变实现频率、波形类型和通道的档位切换。而真正在实现以此实验为基础的信号发生器开发时，可以考虑进行无极调节的硬件设计。借助带滑动变阻器的旋钮，频率和相位的控制将有更加丰富的选择，也更接近于一个真实的信号发生器。

20 无源蜂鸣器驱动设计与验证

工程源码	|----02_设计实例 |---- ch20_pwm_gen
相关视频课程	
说明	

章节导读

蜂鸣器是一种产生声音的器件，广泛应用于报警器、电子玩具、汽车电子设备、定时器等电子产品中作发声器件。

蜂鸣器按照构造方式的不同，可分为压电式蜂鸣器和电磁式蜂鸣器两种类型。压电式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器、共鸣箱以及外壳等组成。电磁式蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。由于两种蜂鸣器发音原理不同，压电式结构简单耐用但音调单一音色差，适用于报警器等设备。而电磁式由于音色好，所以多用于语音、音乐等设备。

蜂鸣器按照驱动电路的不同可以分为有源蜂鸣器与无源蜂鸣器。有源蜂鸣器内部带震荡源，所以只要通电就会鸣叫；而无源蜂鸣器内部不带震荡源，因此如果用直流信号无法令其鸣叫，这就需要用 2K-5K 的方波（声音频率）去驱动。

本章将介绍高云开发板上使用的蜂鸣器电路，并使用 FPGA 来实现驱动无源蜂鸣器按照“哆来咪发梭拉西”7个音调发声。并在此基础上实现“天空之城”的音乐发生器

20.1无源蜂鸣器电路设计

高云开发板上使用了一枚 3.3V 直流电压驱动的电磁式无源蜂鸣器，其电路如下图 20-1 所示：

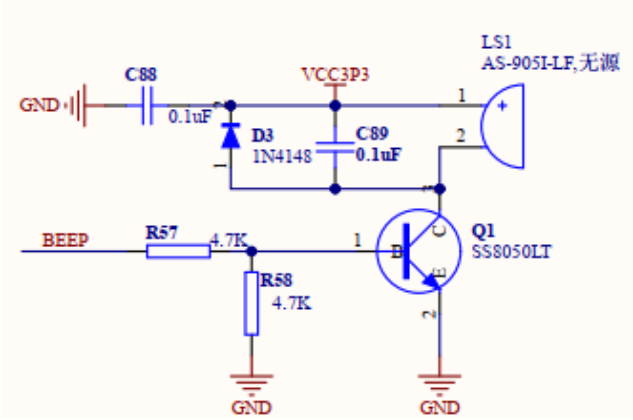


图 20-1 无源蜂鸣器驱动电路

在上述电路中，电容 C89 用于提高电路抗干扰性能；D3 起保护三极管的作用，当三极管突然截止时，无源蜂鸣器两端产生的瞬时感应电动势可以通过 D3 迅速释放掉，避免叠加到三极管集电极上从而击穿三极管；BEEP 端口接 FPGA 输出管脚，使用时只需要在 BEEP 信号上输入 2~5KHz 的 PWM 波，就能驱动蜂鸣器按照既定的频率产生振动信号。

20.2无源蜂鸣器驱动原理

通过前面对无源蜂鸣器的特点介绍可知，要使无源蜂鸣器能够正常发声，需要在控制端 BEEP 给出相应频率的 PWM 波。因此，对于无源蜂鸣器的控制，就转化为了设计一个 PWM 波发生电路。因此，接下来将介绍 PWM 波发生部分相关设计。

PWM 波即脉冲宽度调制，英文全称 Pulse Width Modulation。PWM 控制技术广泛应用在测量、通信、功率控制与变换的等众多领域中，应用的逆变电路绝大部分是 PWM 型。以下图 20-2 为周期为 1KHz，脉冲宽度（占空比）分别为 20%、50%、90%的波形图：

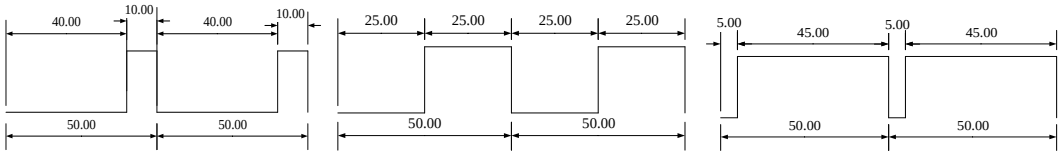


图 20-2 占空比示意图

由上图可知，当信号周期一定，信号高电平时间所占信号周期的百分比不一样，即为不同占空比的 PWM 波。在逆变电路中，当使用这样的波形去驱动 MOS 管的导通时，由于一个周期内不同占空比的 PWM 信号其高电平持续长度

不一样，因此使得 MOS 管的开通时间也不一样，从而使得电路中的平均电流也不一样。因此，通过调整PWM波的占空比即可调整被控制电路中的平均电流。

而除了调整 PWM 信号的占空比，PWM 信号的周期也是可以调整的，例如，在逆变电路中，使用 IGBT 作为开关器件，常见开关频率为几 K 到几十 K，而使用 MOS 管作为开关器件，其开关频率则可高达几百 K。因此，对于不同的器件，对驱动信号的频率要求也不一样，还需要能够对PWM波的频率进行调整。

通过以上分析，可以看出，要设计一个 PWM 发生电路，需要能够实现对信号的频率和占空比的调节。在单片机或者 DSP 中，产生 PWM 波的方法就是使用片上定时器进行循环计数，通过设定定时器的一个定时周期时长来确定对应输出 PWM 信号的频率，同时还有一个比较器，该比较器比较定时器的实时计数值与用户设定的比较值的大小，根据比较结果来控制输出信号的电平高低。通过设定不同的比较值，即可实现不同占空比的 PWM 信号输出。

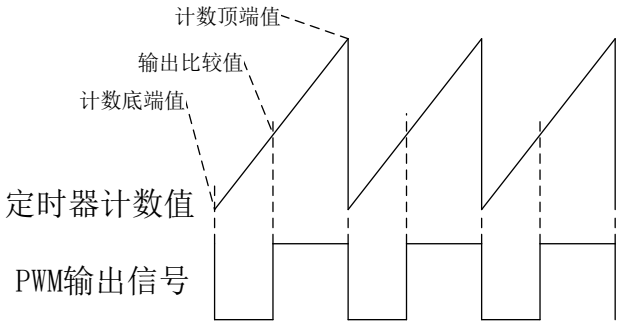


图 20-3 使用定时器产生 PWM 原理

20.3PWM 发生器模块设计

对于 FPGA 来说，要产生 PWM 波，也可以借鉴单片机或 DSP 使用定时器产生 PWM 波的思路。依据上一小节的 PWM 原理图可提取出如下两个主要电路：定时器/计数器电路以及输出比较电路。可以设计出如下图 20-4 所示的 PWM 发生器模块。

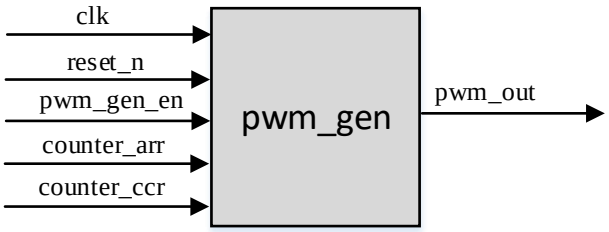


图 20-4 PWM 发生器模块接口

表 20-1 PWM 发生器模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块工作时钟，50MHz
reset_n	I	模块复位信号
pwm_gen_en	I	PWM 产生模块使能信号
counter_arr	I	输入 32 位预重装值，确定频率
counter_ccr	I	输入 32 位输出比较值，确定占空比
pwm_out	O	PWM 输出信号

最终输出 PWM 波的频率计算公式为：

$$f_{pwm} = \frac{f_{clk}}{counter\_arr}$$

其中，这里的 counter_arr 是自减计数器的预重装值，计数器是从 counter_arr 开始递减到 1。因此，当输出频率确定时，可计算得到预重装值，计算公式为：

$$counter\_arr = \frac{f_{clk}}{f_{pwm}}$$

例如，当希望设置输出信号频率为 5KHz 时

$$counter\_arr = \frac{f_{clk}}{f_{pwm}} - 1 = \frac{50000000}{5000} = 10000$$

因此，只需要设置 counter_arr 值为 10000 即可使得最终输出信号频率为 5KHz。

当输出 PWM 频率确定后，其输出占空比计算则为输出比较值与预重装值之商。计算公式为：

$$\text{PW} = \frac{counter_{ccr}}{counter_{arr}}$$

因此，当输出占空比确定时，可计算得到输出比较值，计算公式为：

$$counter_{ccr} = \text{PW} \times counter_{acr}$$

例如，当输出频率为 5KHz，输出占空比为 70%时：

$$counter_{ccr} = \text{PW} \times counter_{arr} = 0.7 * 10000 = 7000$$

在运行过程中，修改预重装值可以设置输出 PWM 信号的频率，并将同时影响输出占空比，而在预重装值确定的情况下，修改输出比较值，则可以设置输出占空比。

这里先写一个 32 位的计数器，来实现频率控制。

```
always@(posedge clk or posedge reset)
if(reset)
    pwm_gen_cnt <= 32'd1;
else if(pwm_gen_en)
begin
    if(pwm_gen_cnt <= 32'd1)
        pwm_gen_cnt <= counter_arr;     //计数减到 1，加载预重装寄存器
```

```
    else
        pwm_gen_cnt <= pwm_gen_cnt - 1'b1;//计数器自减 1
end
else
    pwm_gen_cnt <= counter_arr; //未使能时，计数器值等于预重装寄存器
```

再写一个比较器，比较器的输出即为 PWM 的输出，来实现占空比控制。

```
always@(posedge clk or posedge reset)
if(reset)                          //复位时，PWM 输出低电平
    pwm_out <= 1'b0;
else if(pwm_gen_cnt <= counter_ccr)//计数值小于比较值，PWM 输出高电平
    pwm_out <= 1'b1;
else
    pwm_out <= 1'b0;       //计数值大于比较值，PWM 输出低电平
```

20.4激励创建仿真验证

在完成上面的设计并分析综合直到没有错误。然后添加并新建仿真文件命名为 pwm_gen_tb.v，在仿真文件中只需要产生 50MHz 基准计数时钟源，然后给出预重装值和输出比较值，在最后使能计数来模拟 PWM 输出。部分激励文件如下：

```
initial begin
    reset_n = 0;
    pwm_gen_en = 0;
    counter_arr = 0;
    counter_ccr = 0;
    #(`CLK_PERIOD*20 +1);
    reset_n = 1;
    #(`CLK_PERIOD*10 +1);

    counter_arr = 1000; //设置输出信号频率为 50KHz
    counter_ccr = 400; //设置输出 PWM 波占空比为 40%
    #(`CLK_PERIOD*10);
    pwm_gen_en = 1;     //启动计数以产生 PWM 输出
    #100050;

    counter_ccr = 700; //设置输出 PWM 波占空比为 70%
    #100050;
    pwm_gen_en = 0;     //停止计数以关闭 PWM 输出

    counter_arr = 500; //设置输出信号频率为 100KHz
    counter_ccr = 250; //设置输出 PWM 波占空比为 50%
    #(`CLK_PERIOD*10);
    pwm_gen_en = 1;     //启动计数以产生 PWM 输出
    #50050;
```

```
    counter_ccr = 100; //设置输出 PWM 波占空比为 20%
    #50050;
    pwm_gen_en = 0;     //停止计数以关闭 PWM 输出

    #200;
    $stop;
end
```

设计好仿真文件之后，打开 Modelsim 工程，添加所需文件，然后将 pwm_gen_tb 的端口信号添加至 Wave 窗口观察，点击观察全局波形图如下图 20-5 所示。

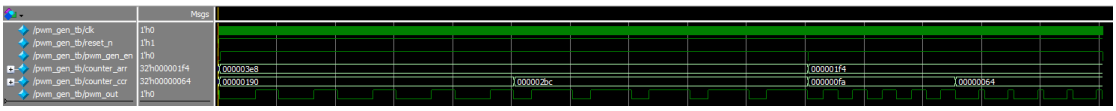


图 20-5 全局波形图

放大局部波形，如下图。此时为设置输出 PWM 波频率为 50KHz（counter_arr 为 1000）、占空比为 40%（counter_ccr 为 400）时的仿真波形。可通过窗口中工具添加标准线，让标准线分别对准 PWM 输出的一个周期的边沿。由图可知，低电平周期为 12us，高电平周期为 8us，整个信号周期为 20us，即频率为 50KHz。占空比为 8/20 = 0.4。符合既定设计。同理可以分析其他波形图。

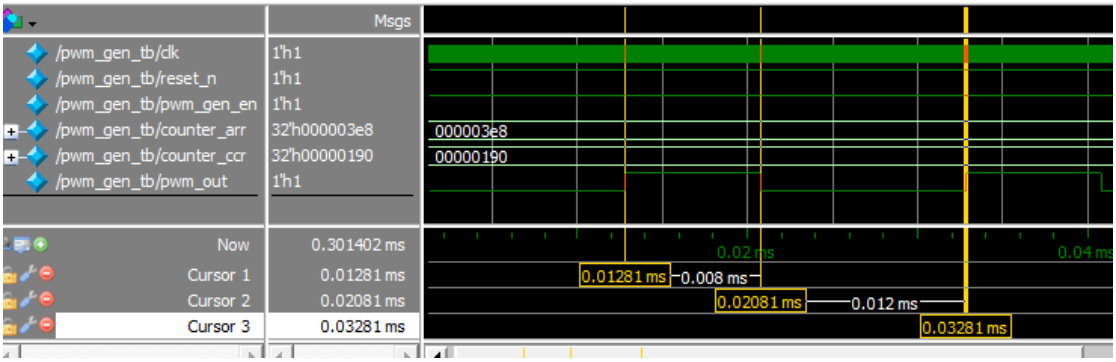


图 20-6 50Khz 占空比 0.4 仿真波形图

下图为设置输出 PWM 波频率为 50KHz（counter_arr 为 1000）、占空比为 70%（counter_ccr 为 700）时的仿真波形。由图可知，低电平周期为 6us，高电平周期为 14us，整个信号周期为 20us，即频率为 50KHz。占空比为 14/20 = 0.7。

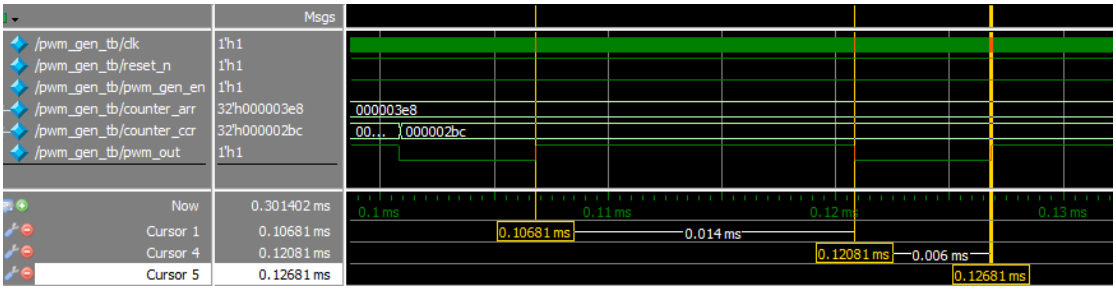


图 20-7 50Khz 占空比 0.7 仿真波形图

下图为设置输出 PWM 波频率为 100KHz（counter_arr 为 500）、占空比为 50%（counter_ccr 为 250）时的仿真波形。由图可知，低电平周期为 5us，高电平周期为 5us，整个信号周期为 10us，即频率为 100KHz。占空比为 5/10 = 0.5。

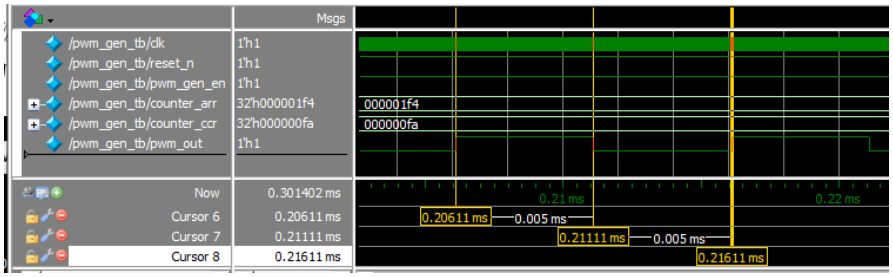


图 20-8 100Khz 占空比 0.5 仿真波形图

下图为设置输出 PWM 波频率为 100KHz（counter_arr 为 500）、占空比为 20%（counter_ccr 为 100）时的仿真波形。由图可知，低电平周期为 8us，高电平周期为 2us，整个信号周期为 10us，即频率为 100KHz。占空比为 2/10 = 0.2。

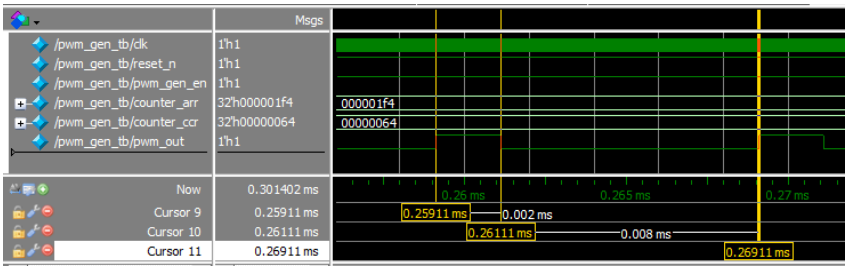


图 20-9 100Khz 占空比 0.2 仿真波形图

20.5板级调试与验证

本实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的 bit 文件下载到高云开发板。
2. 下载完成后开发板上蜂鸣器能否播放音阶。

系统所需硬件

1. 高云开发板。
2. 电源电缆一根。
3. 高云下载器一个。
4. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

20.5.1顶层文件设计

该部分电路比较简单，因此本板级验证部分将不再讲解代码的详细设计思路，只给出具体代码。

蜂鸣器播放音调电路的代码部分如下所示：

```
module pwm_gen_test(
  clk,
  reset_n,

  beep
);
  assign reset=~reset_n;
  input clk;
  input reset_n;

  output beep;

  reg [31:0]counter_arr;    //预重装值寄存器
  wire [31:0]counter_ccr;    //输出比较值

  reg [24:0]t500ms_dly_cnt; //500ms 延时计数器
  reg [2:0]pitch_num;       //音调编号

  parameter T500ms_MAX_CNT  = 25'd25000000;

  localparam
    D1  = 170068, //D 调音 1
    D2  = 151515, //D 调音 2
    D3  = 142857, //D 调音 3
    D4  = 127227, //D 调音 4
    D5  = 113379, //D 调音 5
    D6  = 101010, //D 调音 6
    D7  = 89928 ; //D 调音 7

  //输出比较值为预重装值一半
    assign counter_ccr = counter_arr >> 1;

  pwm_gen pwm_gen(
```

```
    .clk(clk),
    .reset_n(reset_n),
    .pwm_gen_en(1'b1),
    .counter_arr(counter_arr),
    .counter_ccr(counter_ccr),
    .pwm_out(beep)
    );

  //500ms 延时计数器计数
  always@(posedge clk or posedge reset)
  if(reset)
    t500ms_dly_cnt <= 1'd0;
  else if(t500ms_dly_cnt == T500ms_MAX_CNT - 1'b1)
    t500ms_dly_cnt <= 1'd0;
  else
    t500ms_dly_cnt <= t500ms_dly_cnt + 1'b1;

  //每 500ms 切换一次音调
  always@(posedge clk or posedge reset)
  if(reset)
    pitch_num <= 3'd0;
  else if(t500ms_dly_cnt == T500ms_MAX_CNT - 1'b1)
    pitch_num <= pitch_num + 1'd1;
  else
    pitch_num <= pitch_num;

  //根据音调编号给预重装值给相应的值
  always@(posedge clk or posedge reset)
  if(reset)
    counter_arr  = 32'd1;
  else
  begin
    case(pitch_num)
      0 :counter_arr  = 32'd1;
      1 :counter_arr  = D1;
      2 :counter_arr  = D2;
      3 :counter_arr  = D3;
      4 :counter_arr  = D4;
      5 :counter_arr  = D5;
      6 :counter_arr  = D6;
      7 :counter_arr  = D7;
    endcase
  end

endmodule
```

20.5.2添加 I/O 约束

打开管脚约束窗口，其中 Location 和 I/O Type 是我们要约束的内容。这里，参考以下表 20-2 管脚约束表即可完成管脚绑定。

表 20-2 管脚绑定表

	功能信号	接口编号	高云
基本管脚	clk	CLK_G	T9
	beep	BEEP	A16
	reset_n	KEY0	B16

具体管脚约束如下图 20-10 所示。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	beep	output		A16	1	False	LVCMOS33
2	clk	input		T9	4	False	LVCMOS33
3	reset_n	input		B16	1	False	LVCMOS33

图 20-10 管脚绑定效果图

20.5.3预重装值的选择和音阶下载验证

通过前面仿真验证，确认该 PWM 发生电路理论设计正确，接下来使用该 PWM 发生模块来驱动高云 开发板上的无源蜂鸣器，让无源蜂鸣器产生“哆来咪发梭拉西”的音调。首先对“哆来咪发梭拉西”7 个音调的频率找到对应值，音调的频率，具体见下表（分别为低音、中音和高音）。

表 20-3 不同音调、音符与频率对应表

音调音符	1̣	2̣	3̣	4̣	5̣	6̣	7̣
A	221	248	278	294	330	371	416
B	248	278	294	330	371	416	467
C	131	147	165	175	196	221	248
D	147	165	175	196	221	248	278
E	165	175	196	221	248	278	312
F	175	196	221	234	262	294	330
G	196	221	234	262	294	330	371
音调音符	1	2	3	4	5	6	7
A	441	495	556	589	661	742	833
B	495	556	624	661	742	833	935
C	262	294	330	350	393	441	495
D	294	330	350	393	441	495	556
E	330	350	393	441	495	556	624
F	350	393	441	495	556	624	661
G	393	441	495	556	624	661	742
音调音符	1̇	2̇	3̇	4̇	5̇	6̇	7̇

A	882	990	1112	1178	1322	1484	1665
B	990	1112	1178	1322	1484	1665	1869
C	525	589	661	700	786	882	990
D	589	661	700	786	882	990	1112
E	661	700	786	882	990	1112	1248
F	700	786	882	935	1049	1178	1322
G	786	882	990	1049	1178	1322	1484

下面以 D 调举例，根据每个音符的频率值，可以计算得出 PWM 发送模块的预重装值，以下表 20-4 为计算得出的音调频率与对应 PWM 发送模块输出相应频率的预重装值。

表 20-4 频率预装值对应表

频率/Hz	预重装值	频率/Hz	预重装值	频率/Hz	预重装值
147	340136	294	170068	589	84889
165	303030	330	151515	661	75643
175	285714	350	142857	700	71429
196	255102	393	127227	786	63613
221	226244	441	113379	882	56689
248	201613	495	101010	990	50505
278	179856	556	89928	1112	44964

本例中，保持 PWM 波的占空比为 50%即可，而通过前面仿真验证可知，占空比为 50%时，输出比较值刚好为预重装值的一半，即只需要将预重装值除以2（右移一位）的结果直接赋值给输出比较值即可，这样可以避免再重复计算输出比较值。

另外，为了保证音调的切换能够容易分辨，因此设计一个 500ms 的定时器，每 500ms 切换一次音调。这里只对低音的“哆来咪发梭拉西”7 个音符进行测试，后面章节将会有更全的测试验证。

然后点击 Place & Route 进行布局布线，生成 bit 数据流文件，然后连接硬件，点击 Programmer 下载文件，下载完成后蜂鸣器即开始循环播放“哆来咪发梭拉西”7 个音。

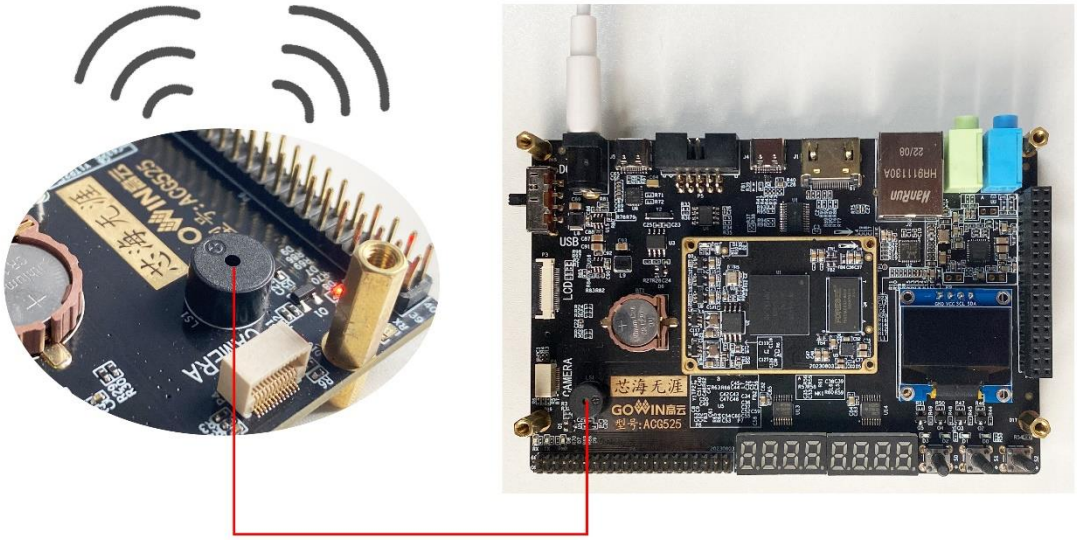


图 20-11 蜂鸣器发出音阶声音

20.6常见问题说明

1. 蜂鸣器有有源和无源之分。有源蜂鸣器内部带震荡源，所以只要通电就会鸣叫；而无源蜂鸣器内部不带震荡源，需要用 2K-5K 的方波（声音频率）去驱动发声。
2. 不同硬件可能会有不同方波占空比的应用需求，通过改变计数器的高低电平持续时间的计数最大值，可以控制和调整占空比的大小。

20.7总结

本节讲解了蜂鸣器的发声原理、PWM 波的含义，并使用 FPGA 设计生成 PWM 波从而控制芯片管脚发出音阶。下一节，也正是基于本节的内容，讲述如何使用蜂鸣器生成音乐的原理。

21 基于 PWM 波的音乐发生器

工程源码	|----02_设计实例 |---- ch21_music_gen
相关视频课程	
说明	

章节导读

前面章节已经实现并上板验证了使用 PWM 产生模块实现对“哆来咪发梭拉西”音调的产生，在此基础上并结合模块化设计思想，利用蜂鸣器实现播放一段音乐的功能。

21.1蜂鸣器音乐实例设计

在本章中，我们将选取乐谱相对简单点的一首谱子“天空之城”，在高云开发板上实现音乐播放。以下是“天空之城”的乐谱，这里主要是讲解 FPGA 实现的一个过程，关于乐谱的细节不作详细讲解。

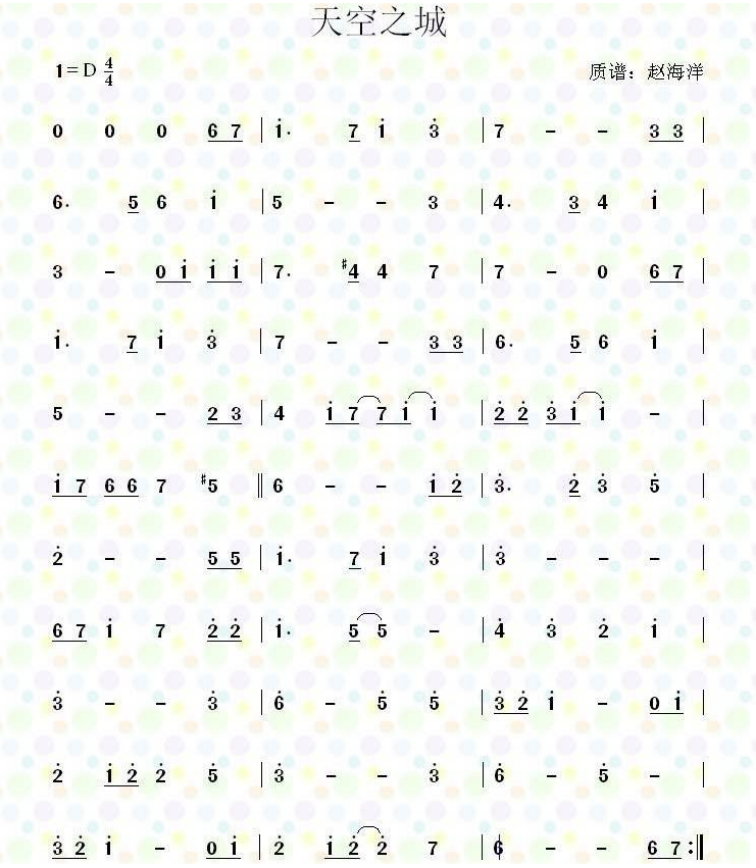


图 21-1 天空之城简谱乐谱

从简谱看，该音乐是 D 调的，这里的各音符对应的频率对应的是上一节蜂鸣器驱动频率表中 D 调的部分。另外，该音乐为四分之四拍，每个音符数字对应为 1 拍。（音符节奏分为一拍、半拍、1/4 拍、1/8 拍，）几个特殊音符说明如下（我们规定一拍音符的时间为 1；半拍为 0.5；1/4 拍为 0.25；1/8 拍为 0.125……）。

（1）普通音符。如音符 7，对应频率 556，占一拍。

（2）带下划线音符，表示 0.5 拍；两个下划线是四分之一拍（0.25）。

（3）有的音符后带一个点，表示多加 0.5 拍，即 1+0.5。

（4）有的音符后带一个“—”，表示多加 1 拍，即 1+1。

（5）有的两个连续的音符上面带弧线，表示连音，可以稍微改下连音后面那个音的频率，比如减少或增加一些数值（需自己调试），这样表现会更流畅，其实不做处理，影响也不大。

这里我们采用 ROM 来存储乐谱，然后通过依次读取 ROM 的数据，然后产生对应的音调。存储格式采用 5bit 数据对音调进行编码，高 2bit 数据数值 3，2，1 作为高中低音编码，低 3bit 数据数值 1~7 作为“哆来咪发梭拉西”的编码。例如存储的数据 5’b01_011 表示的是低音“3”。通过这种形式将乐谱进行编码存储在 ROM 中（注，需要对乐谱有一定了解才能对乐谱进行正确的编码，具体编码后的 ROM 初始化 mi 文件可参考工程）。

整个音乐发生器的设计结构如下图 21-2 所示：

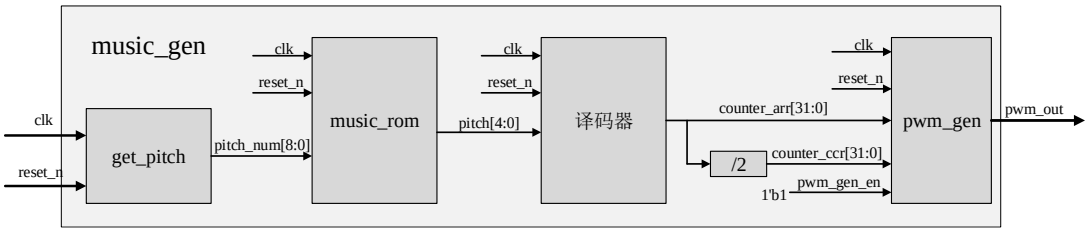


图 21-2 音乐发生器设计框图

在节拍的设计上，以 250ms 时间为一拍的时间，半拍的时间是 125ms，通过上面的乐谱来看，没有低于半拍的，那设计时就以 125ms 为基本单位时间，则 1 拍就是 2 个单位时间，半拍是一个单位时间。get_pitch 模块一方面产生这 125ms 的单位时间，另一方面在一个个单位时间的控制下产生获取 rom 里音符的编码地址。下面是 get_pitch 模块的代码。

```
module get_pitch(
    clk,
```

```
    reset_n,

    pitch_num
);
    assign reset=~reset_n;
    input clk;    //系统时钟输入__50MHz
    input reset_n;   //复位信号输入，低有效

    output reg[8:0] pitch_num; //获取音符 rom 的地址编号

    parameter T125ms_MAX_CNT  = 24'd6250000;

    reg [23:0] t125ms_dly_cnt; //125ms 延时计数器,每个节拍的时间

    //125ms 延时计数器计数
    always@(posedge clk or posedge reset)
    if(reset)
        t125ms_dly_cnt <= 1'd0;
    else if(t125ms_dly_cnt == T125ms_MAX_CNT - 1'b1)
        t125ms_dly_cnt <= 1'd0;
    else
        t125ms_dly_cnt <= t125ms_dly_cnt + 1'b1;

    //每 125ms 切换一次音符
    always@(posedge clk or posedge reset)
    if(reset)
        pitch_num <= 9'd0;
    else if(t125ms_dly_cnt == T125ms_MAX_CNT - 1'b1)
        pitch_num <= pitch_num + 1'd1;
    else
        pitch_num <= pitch_num;

endmodule
```

乐谱的存储上，保持与上面单位时间是半拍规定一致，乐谱存放也以半拍为一个存储数据进行存放。ROM 初始化文件存放目录为 music_gen\src\music_rom.mi。存放数据是以“天空之城”的乐谱实际不超过 256 个音符，也就是说初始化文件的数据不超过 512 个数据（数据以半拍为单位进行存储，一个节拍用两个数据存储），这里 ROM 的深度设置为 512，没有初始化的 ROM 数据选择使用 0 来进行填充。这样正好让两次音乐播放之间还能留有一定的时间间隔，ROM 的配置界面如下图 21-3 所示。

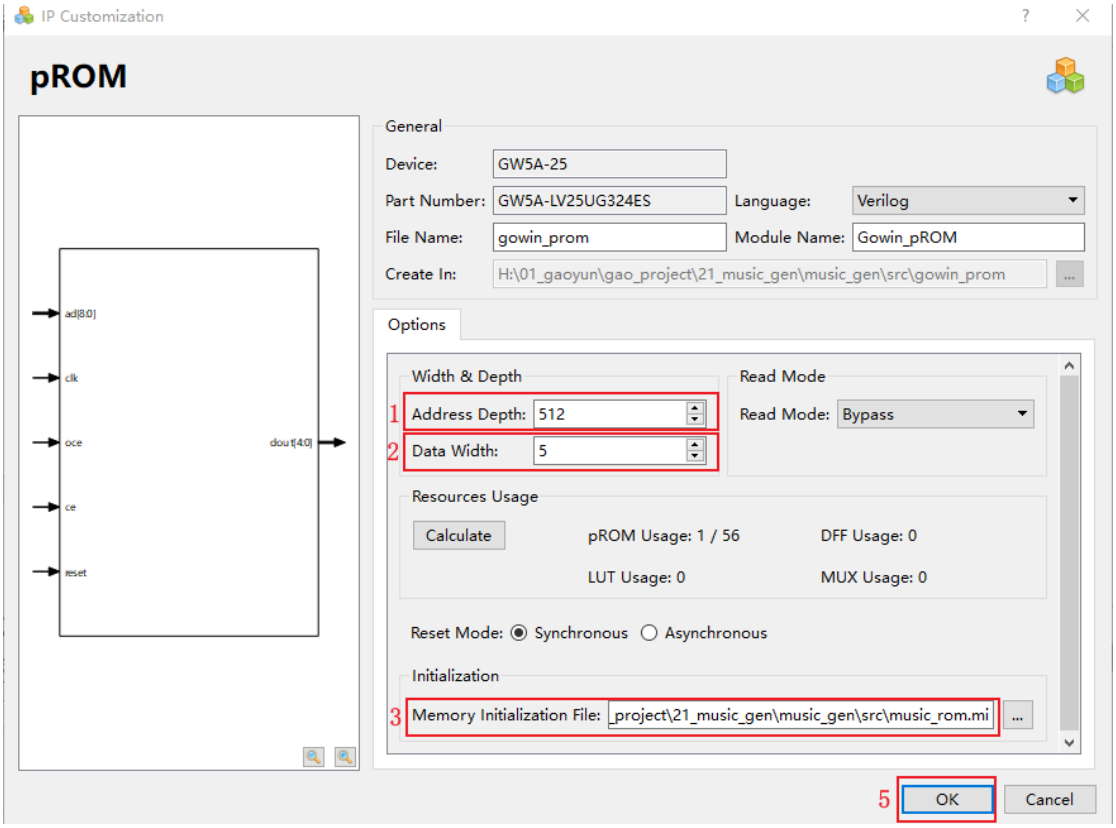


图 21-3 ROM 配置界面

乐谱的存储是经过编码后存储在 ROM 中的，从 ROM 中获取的音符数据还需要经过译码器转换成 PWM 产生器产生音符对应的预重装值寄存器 counter_arr。具体代码见顶层设计代码，下面给出顶层设计代码如下。

```
module music_gen(
  clk,
  reset_n,

  beep
);
  wire reset;
  input clk;   //系统时钟输入，50MHz
  input reset_n;   //复位信号输入，低有效
  output beep;     //蜂鸣器控制信号

  reg  [31:0]counter_arr;     //预重装值寄存器
  wire [31:0]counter_ccr;     //输出比较值


  wire [8:0] pitch_num;       //音乐的音调编号，0——>最大值 循环递增
  wire [4:0] pitch;           //音乐的音调

  localparam
```

```
    DL1 = 340136, //D 调低音 1
    DL2 = 303030, //D 调低音 2
    DL3 = 285714, //D 调低音 3
    DL4 = 255102, //D 调低音 4
    DL5 = 226244, //D 调低音 5
    DL6 = 201613, //D 调低音 6
    DL7 = 179856, //D 调低音 7

    D1  = 170068, //D 调音 1
    D2  = 151515, //D 调音 2
    D3  = 142857, //D 调音 3
    D4  = 127227, //D 调音 4
    D5  = 113379, //D 调音 5
    D6  = 101010, //D 调音 6
    D7  = 89928 , //D 调音 7

    DH1 = 84889,  //D 调高音 1
    DH2 = 75643,  //D 调高音 2
    DH3 = 71429,  //D 调高音 3
    DH4 = 63613,  //D 调高音 4
    DH5 = 56689,  //D 调高音 5
    DH6 = 50505,  //D 调高音 6
    DH7 = 44964;  //D 调高音 7

    assign reset=~reset_n;

  //乐谱存放 rom
    Gowin_pROM Gowin_pROM(
        .dout(pitch), //output [4:0] dout
        .clk(clk), //input clk
        .oce(1'b1), //input oce
        .ce(1'b1), //input ce
        .reset(reset), //input reset
        .ad(pitch_num) //input [8:0] ad
    );

  get_pitch get_pitch(
    .clk(clk),
    .reset_n(reset_n),

    .pitch_num(pitch_num)
  );

  //根据 rom 存储输出不同的音调输出不同的预置数
  always@(posedge clk or posedge reset)
  if(reset)
    counter_arr  = 32'd1;
```

```
  else
  begin
    case(pitch)
      5'b01_001:counter_arr  = DL1;  //D 调低音 1
      5'b01_010:counter_arr  = DL2;  //D 调低音 2
      5'b01_011:counter_arr  = DL3;  //D 调低音 3
      5'b01_100:counter_arr  = DL4;  //D 调低音 4
      5'b01_101:counter_arr  = DL5;  //D 调低音 5
      5'b01_110:counter_arr  = DL6;  //D 调低音 6
      5'b01_111:counter_arr  = DL7;  //D 调低音 7

      5'b10_001:counter_arr  = D1;   //D 调音 1
      5'b10_010:counter_arr  = D2;   //D 调音 2
      5'b10_011:counter_arr  = D3;   //D 调音 3
      5'b10_100:counter_arr  = D4;   //D 调音 4
      5'b10_101:counter_arr  = D5;   //D 调音 5
      5'b10_110:counter_arr  = D6;   //D 调音 6
      5'b10_111:counter_arr  = D7;   //D 调音 7

      5'b11_001:counter_arr  = DH1;  //D 调高音 1
      5'b11_010:counter_arr  = DH2;  //D 调高音 2
      5'b11_011:counter_arr  = DH3;  //D 调高音 3
      5'b11_100:counter_arr  = DH4;  //D 调高音 4
      5'b11_101:counter_arr  = DH5;  //D 调高音 5
      5'b11_110:counter_arr  = DH6;  //D 调高音 6
      5'b11_111:counter_arr  = DH7;  //D 调高音 7

      default:  counter_arr  = 32'd1;//休止符
    endcase
  end

  //设置输出比较值为预重装值一半
  assign counter_ccr = counter_arr >> 1;

  pwm_gen pwm_gen(
    .clk(clk),
    .reset_n(reset_n),
    .pwm_gen_en(1'b1),
    .counter_arr(counter_arr),
    .counter_ccr(counter_ccr),
    .pwm_out(beep)
  );

endmodule
```

21.2仿真验证

在完成上面的设计并分析综合直到没有错误。然后添加并新建仿真文件命名为 music_gen_tb.v，在仿真文件中除了实现例化需要仿真的文件以及时钟创建，顶层仿真文件的设计相对比较简单，只要给出时钟和复位激励就可以，为了减少仿真时间，在仿真代码中使用 defparam music_gen.get_pitch.T125ms_MAX_CNT = 24'd62500;将 125ms 节拍间隔减小至 1.25ms，这样既能对功能进行验证，又能减少仿真时间，具体代码如下。

```
module music_gen_tb();

    reg clk;  //系统时钟输入，50M
    reg reset_n; //复位信号输入，低有效

    wire beep;      //pwm 输出信号

    defparam music_gen.get_pitch.T125ms_MAX_CNT = 24'd62500;
    GSR GSR(.GSRI(1'b1));
    music_gen music_gen(
        .clk(clk),
        .reset_n(reset_n),

        .beep(beep)
    );

    initial clk = 0;
    always #(`CLK_PERIOD/2) clk = ~clk;

    initial begin
        reset_n = 0;
        #(`CLK_PERIOD*20 +1);
        reset_n = 1;

        wait(music_gen.pitch_num == 512);
        #200;

        $stop;
    end
    endmodule
```

设计好仿真文件之后，打开 Modelsim，新建仿真工程，添加仿真所需文件，配置好仿真环境，将 music_gen 模块中的 clk、reset_n、beep、pitch_num、counter_ccr、counter_arr 信号添加至 Wave 窗口观察，然后点击全速运行，然后运

行一会儿，手动点击停止，可以看到如下图 21-4 所示的部分波形文件。

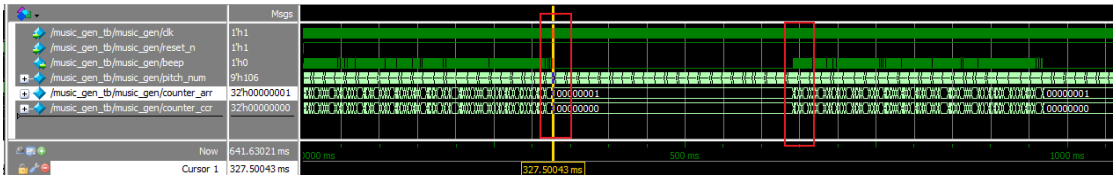


图 21-4 仿真波形文件

放大上图红框中的波形图，其中 pitch_num 转换为 Unsigned 显示，如下图 21-5 所示，波形文件如下图 21-6 所示。

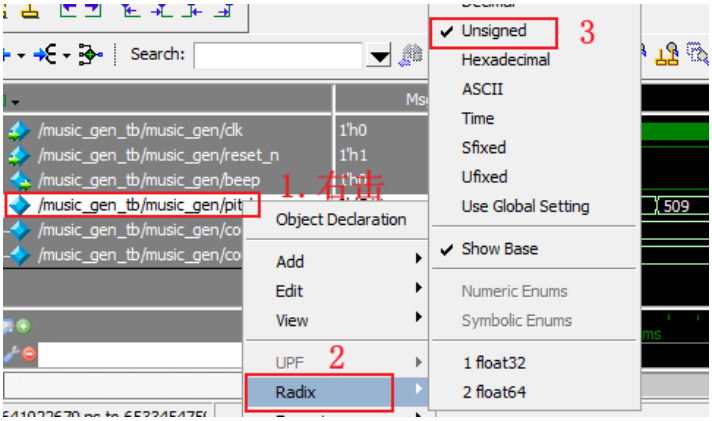


图 21-5 转换信号显示格式

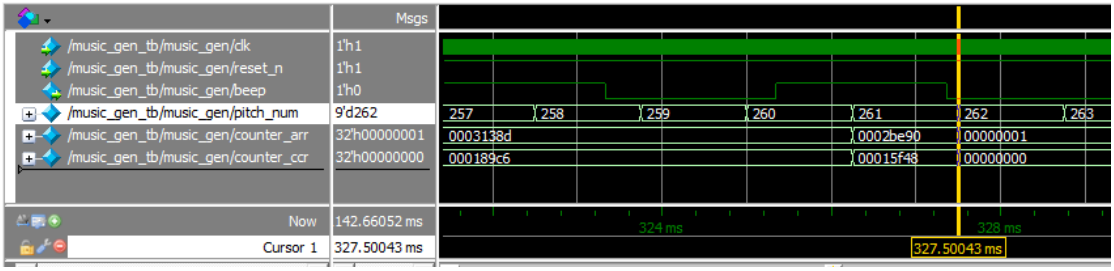


图 21-6 放大波形文件局部（左红框）

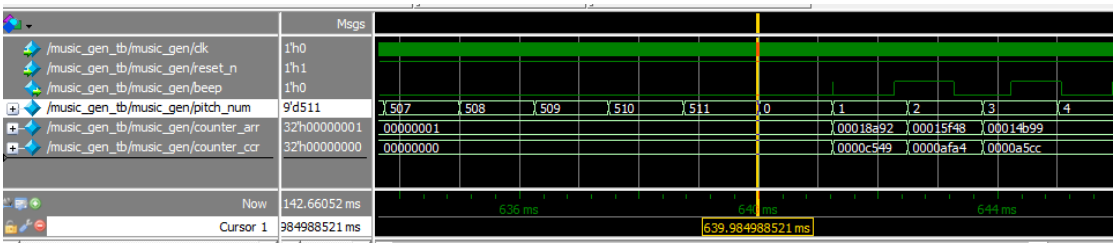


图 21-7 放大波形文件局部（右红框）

从波形上可以看出，循环音乐循环播放之间有一定的时间间隔，这个与设计预期是一致的。可以放大波形后看，其波形也是与预期设计是一致的。

21.3板级验证

本实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的 bit 文件下载到高云发板。
2. 下载完成后蜂鸣器能否播放音乐：天空之城。

21.3.1系统所需硬件

1. 高云开发板。
2. 电源电缆一根
3. 高云下载器一个
4. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台。

21.3.2添加 I/O 约束

打开管脚约束窗口，其中 Location 和 I/O Type 是我们要约束的内容。这里，参考以下表 21-1 管脚约束表即可完成管脚绑定。

表 21-1 管脚绑定表

	功能信号	接口编号	高云开发板
基本管脚	clk	CLK_G	T9
	beep	BEEP	A16
	reset_n	KEY0	B16

具体管脚约束如下图 21-8 所示。

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	beep	output		A16	1	False	LVCMOS33
2	clk	input		T9	4	False	LVCMOS33
3	reset_n	input		B16	1	False	LVCMOS33

图 21-8 管脚绑定效果图

然后点击“Place & Route”进行布局布线，成功之后，点击“Program Device”下载文件，下载完成后蜂鸣器即开始循环播放音乐。

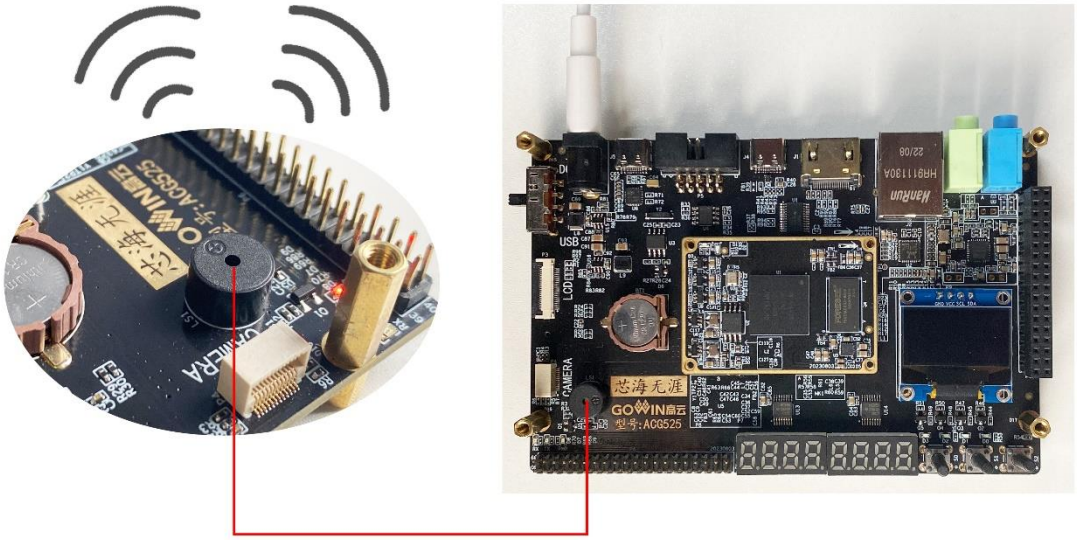


图 21-9 下载好 bit 文件后蜂鸣器发声

读者可以在此基础上做更多的尝试进行熟悉和实验。

21.4总结

本节在上一节使用蜂鸣器奏出音阶的基础上，进一步结合前面的 ROM 存储器章节进行综合运用，将音乐“天空之城”乐谱翻译成 PWM 脉冲控制信息而从蜂鸣器输出。

22 矩阵键盘驱动设计与验证

工程源码	|----02_设计实例 |---- ch22_keyboard_4x4
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本章我们主要是来实现用 verilog 语言实现矩阵键盘的驱动。提到矩阵键盘，许多读者应该不会陌生，因为学过单片机、ARM 的人都应该知道并用 C 语言做过矩阵键盘的驱动。本章作者将用 verilog 语言来写一个 4x4 矩阵键盘的驱动，并用 LED 灯的状态指示按键的状态。课后会留有和本章内容有关的作业，希望所有读者能独立完成作业。接下来就让我们一块走进今天的学习内容“矩阵键盘学习之旅”。

22.1矩阵键盘诞生的背景

相信许多人经常用到矩阵键盘但是却不知道“它”的由来。由于最早的 MCU（即单片机）其 IO 口相对较少，而且用到按键过多的话，就会占用过多的 IO。人们为了解决这个问题就引入了“矩阵键盘”。在矩阵键盘中，每条行线和列线在交叉处都不是直接连同，而是通过一个按键直接相连，这样以来一个 4x4 的矩阵键盘只需要 8 根控制线就可以完成 16 个按键的控制。下面为常见的两类 4x4 矩阵键盘。下方第一幅图片为机械式矩阵键盘，下方第二幅图片为薄膜型矩阵键盘。两种矩阵键盘虽然形式不同，但是原理和功能完全相同，使用方式也完全相同。从使用手感来说，机械式矩阵键盘优于薄膜式矩阵键盘。

图 22-1 机械矩阵键盘

图 22-2 薄膜型矩阵键盘

22.2矩阵键盘工作原理

要说起矩阵键盘是如何工作的，就得从它的硬件原理图来理解它工作机理。如下图 22-3 为一个典型的矩阵键盘应用电路。

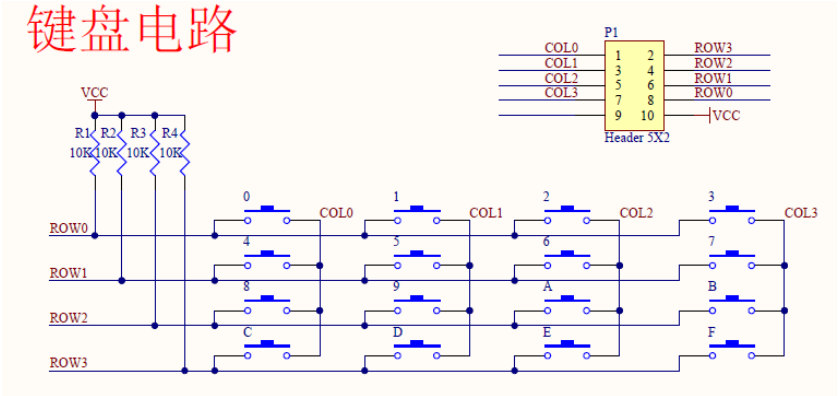


图 22-3 矩阵键盘应用电路

矩阵键盘有两组信号，共 8 根信号线，其中 COL 将每一列的四个按键的一端连接起来，而 ROW 则将每一行的 4 个按键的一端连接起来，通过 4 行 4 列的 8 根信号线，总共能够管理 16 个按键，这也是矩阵键盘相较于普通独立按键的优势所在，即用较少的 IO 得到较多的按键输入。

那我们如何去检测某一个按键被按下，并且还能找到其具体的位置呢？思路其实并不复杂，过程分为两步，第一步检测到有按键按下，第二步确定被按下的按键的具体位置。FPGA 与矩阵键盘的硬件连接示意图如下图 22-4，图中将 4 根行控制线（ROW）通过上拉电阻拉到了高电平未画出，实际是需要上拉到高电平的，这里讲解原理，图中上拉过程省略。

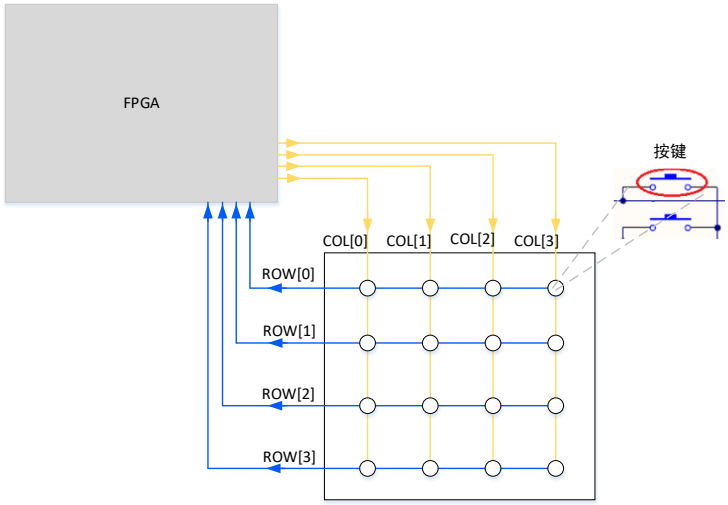


图 22-4 矩阵键盘按键模型原理图

矩阵键盘的 4 条列控制线（COL）由 FPGA 控制输出，4 根行控制线（ROW）为 FPGA 的输入信号，总的思路是 FPGA 通过控制输出 COL 信号，通过检测 ROW 信号状态判断按键的按下以及按下按键的具体位置。

在初始状态下，FPGA 输出 4 条列控制线（COL）为低电平。在没有按键按下时，ROW 信号由于上拉至高电平均保持高电平状态。FPGA 的输入的行信号全为 1。如下图 22-5 所示。

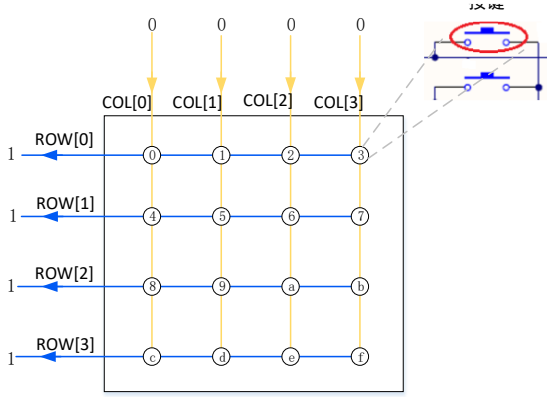


图 22-5 矩阵键盘按键图

当某个按键按下时，比如图 22-6 中按键“9”被按下时，COL[1]和 ROW[2]就相连接了，则输入到 FPGA 的 ROW[2]信号会变为 0。此时就识别到有按键按下。具体是哪个按键按下这个时候还无法判断，只能确定按键是 ROW[2]上的某个按键。因为此时的 COL 全为 0，按键“8”、“9”、“a”、“b”任意一个被按下均会能使 ROW[2]信号变为 0。考虑到按键过程会有抖动的，采用延时方式进行消抖。所谓延时方式消抖是在检测到按键按下后，延时一段时间（一般采用延时 20ms，因为按键抖动时间的长短由按键的机械特性决定，一般为 5ms～10ms）

后再次检测按键是否按下，如果按下，则认为按键确实已经按下，如果没有按下，则认为仅仅是抖动，并未有真的有按键按下。

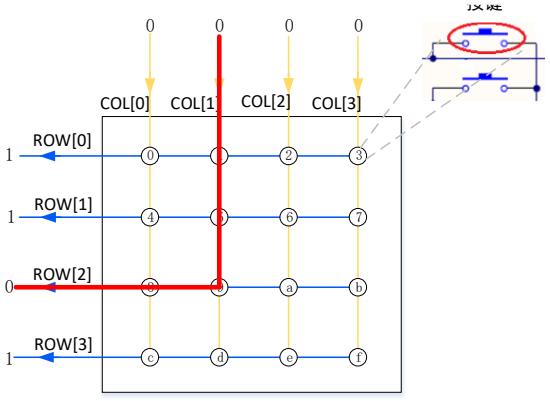


图 22-6 按键“9”按下示意图

在上面过程中检测到按键按下后，接下来是确定按下按键的位置，按键所在的行已经确定，接下来需要对按键所在的列进行判断。采用的方法是列扫描法，所谓列扫描就是依次将列信号 COL[0]、COL[1]、COL[2]、COL[3]中一个信号给低电平 0，其余 3 个信号给高电平 1。具体可参见下面示意图。

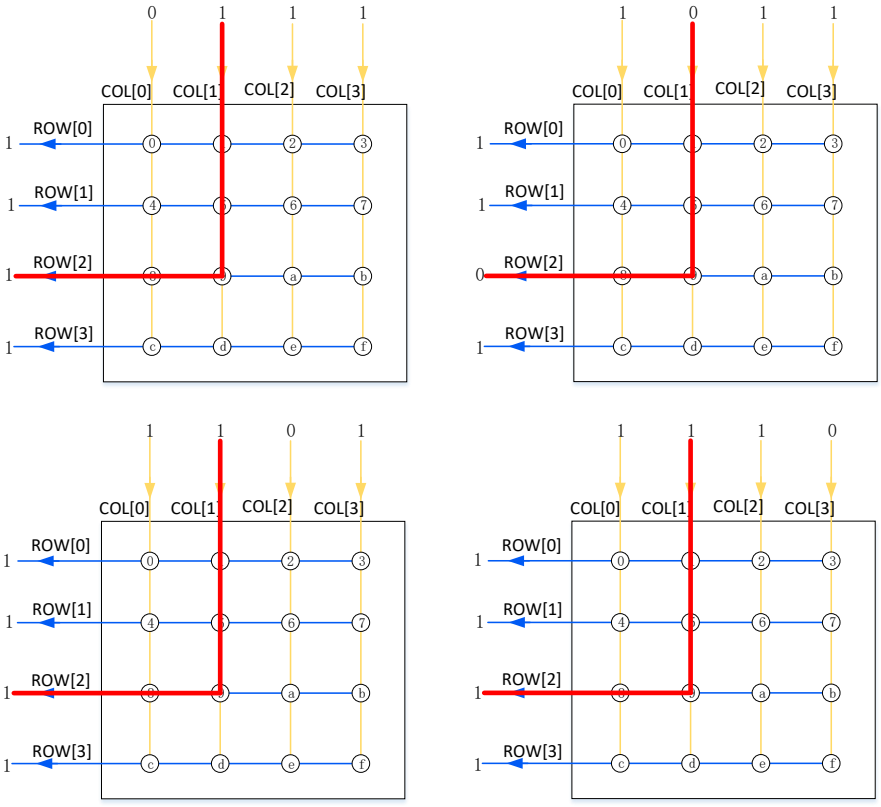


图 22-7 矩阵键盘 col 扫描示意图

从上面列扫描示意图中可以看到，当列扫描在 COL[1]为低电平 0 时，FPGA

再次检测到 ROW[2]为低电平 0。此时可以判断出当前按下的按键所在的列。按键所在行和列均确定的情况下，就可以确定当前按键的具体位置了。

按键按下检测的过程就是上面介绍的那样，在检测到具体哪个按键按下后，还需要有一个检测按键松开的过程，在按键松开后，一次完整的按键检测才算完成。按键松开的检测相对简单，同样的以上面介绍的按键“9”为例，在确定按下的按键位置后，将列信号 COL 全部给低电平 0。通过检测 ROW 信号是否全部变为 1，若全部变为 1，表示按键松手了，如果 ROW[2]信号仍为 0，则表示按键处于按下状态。

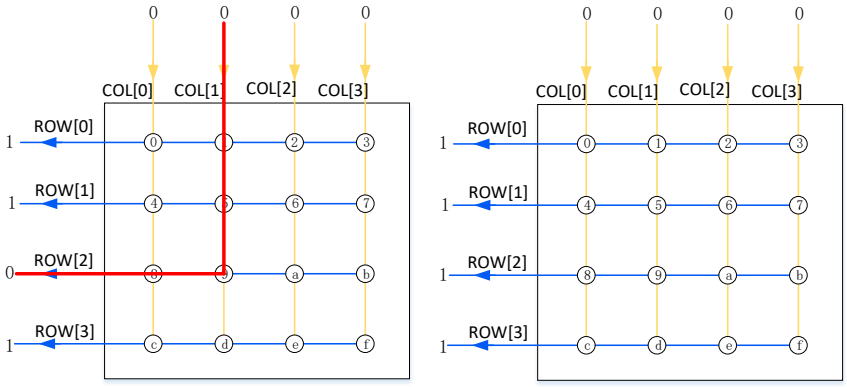


图 22-8 通过 ROW 信号全拉高来检测按键是否被按下

22.3矩阵键盘扫描逻辑设计

按照该思路设计模块如下图 22-9 所示

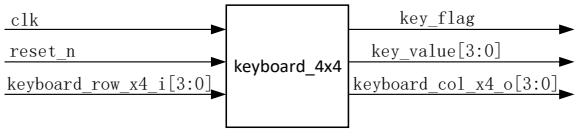


图 22-9 4X4 矩阵键盘总体框架图

表 22-1 输入输出端口列表及说明

端口类型	端口名	描述
Input	clk	系统时钟 50MHz
Input	reset_n	系统复位，低有效
Input	keyboard_row_x4_i	矩阵键盘行控制线，按键未按下为高电平
Output	keyboard_col_x4_o	矩阵键盘列控制线，驱动列信号为低，实现按键状态扫描，
Output	key_value	输出键盘键值
Output	key_flag	按键检查成功标志信号，每当按键检测成功，产生一个时钟周期的高脉冲

22.4矩阵键盘检测状态转移图

现在就来给大家讲解今天的重点，状态转移图的设计。可能会有读者说，这个东西有什么用处？给大家举个例子，“如何去建筑一座楼房”。在建造前肯定是工程师设计建筑的建造图纸，然后工人们按照图纸进行分工建造房屋。今天讲的状态转移图就像是工程师设计图纸没有图纸，直接编代码，肯定会来回修改多次。下面进入正题。矩阵键盘的状态转移图如下图 22-10 所示。

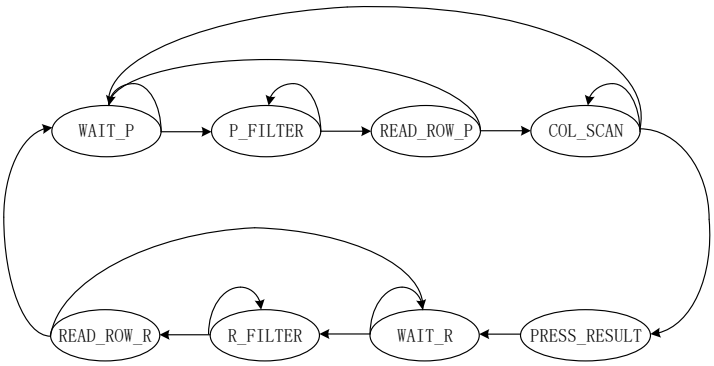


图 22-10 矩阵键盘状态转移图

状态转移的流程描述如下：

- WAIT_P：上电后的初始状态，主要是用来检测是否有按键被按下，该状态下，FPGA 的 4 根 COL 信号输出为 4’b0000，通过输入的 ROW 信号的状态判断有无按键按下。无按键按下时，行控制线 keyboard_row_x4_i 为 4‘b1111，有按键按下时 keyboard_row_x4_i 不等于 4‘b1111，则进入 P_FILTER 状态。
- P_FILTER：主要是进行按键消抖检测，判断是否有按键真实被按下。采用延时方式进行消抖，通过延时 20ms 后，再去判断是否有按键真实被按下。该状态主要是完成 20ms 的延时，20ms 延时到达进入下一状态，否则停留在本状态。
- READ_ROW_P：判断是否真的有按键按下，如果 keyboard_row_x4_i 不等于 4‘b1111，表明真的有按键按下，跳转到 COL_SCAN 状态进列扫描，读取并记录此时行状态 keyboard_row_x4_i 信号，记录结果使用 keyboard_row_reg 存储。如果 keyboard_row_x4_i 等于 4‘b1111，表明实际并未有按键按下，跳转到初始状态 WAIT_P。
- COL_SCAN：进行列扫描，进一步确认前面被按下按键所在的列，通过依次对 COL 信号给值 4‘b1110、4‘b1101、4‘b1011、4‘b0111，然后通

过检测行信号输入是否等于 READ_ROW_P 状态中存储的 keyboard_row_reg（注：这里通过与 keyboard_row_reg 进行比较可以起到同一时刻只对单个按键按下给出检测结果，如果同一时刻按下两个按键将忽略，或者如果一个按键按下的情况下，按下第二个按键，第二个按键的按下将忽略，不起作用），如果等于则确认按下按键所在列，结合前面按下按键的行数据就可以确认按下按键的具体位置了，跳转到 PRESS_RESULT 状态，读取并记录此时行状态 keyboard_col_x4_o 信号，记录结果使用 keyboard_col_reg 存储。否则在列扫描结束后仍未确定按下按键的列，则认为是一次错误的检测回到 WAIT_P 状态重新开始下一次的检测。

- PRESS_RESULT：将输出按下按键所在行和列的位置结果记录。状态跳转到 WAIT_R 等待按键释放状态，同时将列控制信号 COL 给值 4’b0000，用于 WAIT_R 状态检测按键的释放；
- WAIT_R: 等待按键释放，如果 keyboard_row_x4_i 等于 4'b1111，则说明按键被释放，状态跳转到释放消抖状态 R_FILTER
- R_FILTER：与 P_FILTER 状态类似，这里是判断是否按键真实被释放。采用延时方式进行消抖，通过延时 20ms 后，再去判断是否有按键真实被释放。该状态主要是完成 20ms 的延时，20ms 延时到达进入下一状态，否则停留在本状态。
- READ_ROW_R：判断是否真的有按键释放，如果 keyboard_row_x4_i 等于 4‘b1111，表明按键真的被释放，一次按键的检测完成，跳转到 WAIT_P 状态进行下一次的按键检测。如果 keyboard_row_x4_i 不等于 4‘b1111，表明实际按键并未真的释放，跳转到 WAIT_R 重新检测按键的释放。

通过这些状态，我们把一个按键按下，到释放的过程完美的进行了一次检测，最终将输出结果用 LED 灯的状态表示。接下来，就是照着状态转移图和状态转移的流程条件，敲写程序。这下读者们可以自由选择语言编程，Verilog 或者 VHDL 随意选，然后去盖起自己的矩阵键盘“大厦”。

下面是整个矩阵键盘驱动状态部分代码，完成代码可参考提供的例程：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

always@(posedge clk or posedge reset)
    if(reset)begin
```

```
        state <= WAIT_P;
        delay20ms_en <= 1'b0;
        keyboard_col_x4_o <= 4'b0000;
        keyboard_row_reg <= 4'b1111;
        keyboard_col_reg <= 4'b0000;
        col_scan_cnt <= 2'd0;
        key_flag_pre <= 1'b0;
        key_value_pre <= 8'd0;
    end
    else begin
        case(state)
            WAIT_P: begin
                delay20ms_en <= 1'b0;
                keyboard_col_x4_o <= 4'b0000;
                keyboard_row_reg <= 4'b1111;
                keyboard_col_reg <= 4'b0000;
                col_scan_cnt <= 2'd0;
                key_flag_pre <= 1'b0;
                key_value_pre <= 8'd0;
                //检测到有按键按下，至少有一行的信号为低
                if(keyboard_row_x4_i != 4'b1111)begin
                    state <= P_FILTER;           //进入到消抖状态
                    delay20ms_en <= 1'b1;         //使能 20ms 延时
                    end
                else begin
                    state <= WAIT_P;
                    delay20ms_en <= 1'b0;
                    end
                end

            P_FILTER:
                //20ms 延时到，进入到再次确认是否有按键按下状态
                if(delay20ms_flag) begin
                    state <= READ_ROW_P;
                    delay20ms_en <= 1'b0;
                    end
                else begin
                    state <= P_FILTER;
                    delay20ms_en <= 1'b1;    //20ms 延时未到，继续计数
                end

            READ_ROW_P:
                //至少有一行的信号为低，说明确实检测到有按键按下
                if(keyboard_row_x4_i != 4'b1111)begin
                //进入到列扫描状态，进一步确认是哪个按键按下
                    state <= COL_SCAN;
                    //寄存此时的行信号
```

```
                keyboard_row_reg  <= keyboard_row_x4_i;
                keyboard_col_x4_o <= 4'b1110;
                end
            //未有行信号为 0，说明只是抖动，并未真的有按键按下
            else begin
                state <= WAIT_P;
            end

        COL_SCAN:
            //找到是哪个按键按下
            if(keyboard_row_x4_i == keyboard_row_reg) begin
                state <= PRESS_RESULT;
                keyboard_col_reg <= keyboard_col_x4_o;
                end
            //扫描结束，未找到哪个按键按下，认为异常回到初始态
            else if(col_scan_cnt == 2'd3) begin
                state <= WAIT_P;
                col_scan_cnt <= 2'd0;
                end
            else begin
                keyboard_col_x4_o<={keyboard_col_x4_o[2:0],1'b1};
                col_scan_cnt <= col_scan_cnt + 1'b1;
                end

        PRESS_RESULT: begin //记录扫描确认按键后的结果
            state <= WAIT_R;
            keyboard_col_x4_o <= 4'b0000;
            key_flag_pre <= 1'b1;
            key_value_pre <= {keyboard_row_reg,keyboard_col_reg};
            end

        WAIT_R: begin                   //等待按键松开
            key_flag_pre <= 1'b0;
            //行信号全为高，说明检测到按键松开
            if(keyboard_row_x4_i == 4'b1111)begin
                state <= R_FILTER;
                delay20ms_en <= 1'b1;
                end
            else begin
                state <= WAIT_R;
                delay20ms_en <= 1'b0;
                end
            end

        R_FILTER:
            //延时 20ms 到,进入再次确认按键松开状态
            if(delay20ms_flag) begin
```

```
                    state <= READ_ROW_R;
                    delay20ms_en <= 1'b0;
                end
                else begin
                    state <= R_FILTER;
                    delay20ms_en <= 1'b1;
                end

            READ_ROW_R:
                //行信号全为高，确认按键已经松开
                if(keyboard_row_x4_i == 4'b1111)
                    state <= WAIT_P;
                else            //按键并未真的松开，是抖动
                    state <= WAIT_R;
            default:state <= WAIT_P;
        endcase
end
```

以上状态机部分程序的代码是按照状态机设计部分的状态转移图来实现的。可能刚开始让大家，先设计状态转移图后敲代码比较难，但是还是希望大家从刚开始就养成一个好的习惯，这样日积月累自己独立思考能力也就上去了。

在使用状态机将按下按键所在行和列的位置确认后，具体每个按键表示的含义和数值可以根据实际需求定义，这里就将按键以十六进制0x0~0xf的键值表示输出。具体代码如下。

```
always@(posedge clk or posedge reset)
if(reset)
    key_value <= 4'd0;
else if(key_flag_pre)begin
    case(key_value_pre)  //{keyboard_row_reg,keyboard_col_reg}
        8'b1110_1110 : key_value <= 4'h0;
        8'b1110_1101 : key_value <= 4'h1;
        8'b1110_1011 : key_value <= 4'h2;
        8'b1110_0111 : key_value <= 4'h3;
        8'b1101_1110 : key_value <= 4'h4;
        8'b1101_1101 : key_value <= 4'h5;
        8'b1101_1011 : key_value <= 4'h6;
        8'b1101_0111 : key_value <= 4'h7;
        8'b1011_1110 : key_value <= 4'h8;
        8'b1011_1101 : key_value <= 4'h9;
        8'b1011_1011 : key_value <= 4'ha;
        8'b1011_0111 : key_value <= 4'hb;
        8'b0111_1110 : key_value <= 4'hc;
        8'b0111_1101 : key_value <= 4'hd;
        8'b0111_1011 : key_value <= 4'he;
        8'b0111_0111 : key_value <= 4'hf;
```

```
        default : key_value <= key_value;
    endcase
end

always@(posedge clk)
    key_flag <= key_flag_pre;
```

整个矩阵键盘的驱动设计基本完成，以上只是部分主要代码，完成代码请参见提供的例程。

22.5仿真验证

本例中，我们将花费一定的篇幅讲解测试脚本的编写，对于矩阵键盘，我们采用 testbench +矩阵键盘仿真模型的方式进行仿真。如图 22-11 所示为仿真和被测试设计的关系图。

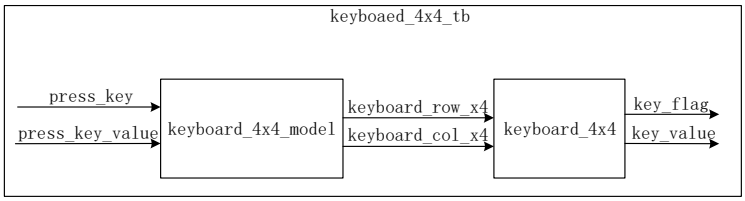


图 22-11 测试系统框架

由上图知道本节内容的测试脚本由两部分组成，即矩阵键盘驱动设计模块和矩阵键盘仿真模型；被测试部分我们已将前面已经讲完了，下面给大家讲解仿真模型如何编写。仿真模型是模拟真实的硬件以及按键按下的过程。为了方便方便比较实际按下按键的位置和矩阵键盘驱动模块检测的按键位置结果数值，在模型设计中加入了按键位置的设置信号 press_key_value。验证矩阵键盘驱动设计的正确性除了看波形外，还可以比较直观的通过比较设计模块 key_value 与 press_key_value 是否相等进行验证。行信号在硬件上是通过上拉电阻接高电平的，模型中使用“pullup();”语句实现上拉的操作。仿真模型的另一个重要的点是，模拟实际按下和释放的过程以及抖动。按键按下实际就是连通按键对应的行与列，对应代码 61~64 行。抖动的过程的产生与前面章节“独立按键消抖设计与验证”中的仿真类似，使用$random 产生。

```
`timescale 1ns/1ns

module keyboard_4x4_model
(
    press_key,
    press_key_value,
```

```
    keyboard_col_x4_i,
    keyboard_row_x4_o
);
    input press_key;
    input [3:0]press_key_value;
    input [3:0]keyboard_col_x4_i;
    output [3:0]keyboard_row_x4_o;

    reg [3:0]keyboard_row_x4_o_tmp;

    pullup(keyboard_row_x4_o[0]);
    pullup(keyboard_row_x4_o[1]);
    pullup(keyboard_row_x4_o[2]);
    pullup(keyboard_row_x4_o[3]);

    assign keyboard_row_x4_o[0] = ~keyboard_row_x4_o_tmp[0] ? 1'b0 : 1'bz;
    assign keyboard_row_x4_o[1] = ~keyboard_row_x4_o_tmp[1] ? 1'b0 : 1'bz;
    assign keyboard_row_x4_o[2] = ~keyboard_row_x4_o_tmp[2] ? 1'b0 : 1'bz;
    assign keyboard_row_x4_o[3] = ~keyboard_row_x4_o_tmp[3] ? 1'b0 : 1'bz;

    reg [15:0]myrand;

    initial begin
    while(1) begin
        keyboard_row_x4_o_tmp = 4'b1111;
        @(posedge press_key)
            case(press_key_value)
                4'h0: press_key_task(0,0);
                4'h1: press_key_task(0,1);
                4'h2: press_key_task(0,2);
                4'h3: press_key_task(0,3);
                4'h4: press_key_task(1,0);
                4'h5: press_key_task(1,1);
                4'h6: press_key_task(1,2);
                4'h7: press_key_task(1,3);
                4'h8: press_key_task(2,0);
                4'h9: press_key_task(2,1);
                4'ha: press_key_task(2,2);
                4'hb: press_key_task(2,3);
                4'hc: press_key_task(3,0);
                4'hd: press_key_task(3,1);
                4'he: press_key_task(3,2);
                4'hf: press_key_task(3,3);
            endcase
        end
    end
```

```
    task press_key_task;
        input [1:0] row;
        input [1:0] col;
        begin
            keyboard_row_x4_o_tmp = 4'b1111;
            //模拟按键按下
            keyboard_row_x4_o_tmp[row] = keyboard_col_x4_i[col];
            //模拟按键按下抖动
            repeat(20)begin
            myrand = {$random} % 65536;
            #myrand keyboard_row_x4_o_tmp[row] = ~keyboard_row_x4_o_tmp[row];
            end
         //按键已经按下并稳定 22ms(20ms 以内认为是抖动，仿真中大于 20ms 即可)
            repeat(22000000)begin
            keyboard_row_x4_o_tmp[row] = keyboard_col_x4_i[col];
            #1;
            end
            //模拟按键松开过程的抖动
            repeat(20)begin
            myrand = {$random} % 65536;
            #myrand keyboard_row_x4_o_tmp[row] = ~keyboard_row_x4_o_tmp[row];
            end
            //按键松开
            keyboard_row_x4_o_tmp = 4'b1111;
            #22000000;
        end
    endtask

endmodule
```

以上就是本节内容的仿真模型代码部分，可能刚开始大家不习惯这么做觉得太繁琐了；但是这样写的结果大大提高了你的仿真正确性和模块设计和仿真的可扩展性。下面贴上整个仿真的测试顶层代码。整个仿真过程模拟按下了 3 次不同的按键，键值分别为 0xa、0x8、0xd。

```
`timescale 1ns/1ns

module keyboaed_4x4_tb;

    reg clk;
    reg reset_n;
    reg press_key;
    reg [3:0]press_key_value;
    wire [3:0]keyboard_row_x4;
    wire [3:0]keyboard_col_x4;

    wire key_flag;
```

```
    wire [3:0]key_value;

    initial clk = 1;
    always #10 clk = ~clk;

    initial begin
        reset_n = 1'b0;
        press_key = 1'b0;
        press_key_value = 8'h0;
        #201;
        reset_n = 1'b1;

        press_key = 1'b1;
        press_key_value = 8'ha;
        #20;
        press_key = 1'b0;

        #50000000;
        press_key = 1'b1;
        press_key_value = 8'h8;
        #20;
        press_key = 1'b0;

        #50000000;
        press_key = 1'b1;
        press_key_value = 8'hd;
        #20;
        press_key = 1'b0;

        #50000000;
        $stop;
    end

    keyboard_4x4_model
    keyboard_4x4_model
    (
        .press_key(press_key),
        .press_key_value(press_key_value),
        .keyboard_col_x4_i(keyboard_col_x4),
        .keyboard_row_x4_o(keyboard_row_x4)
    );

    keyboard_4x4
    keyboard_4x4
    (
        .clk(clk),
        .reset_n(reset_n),
```

```
        .keyboard_row_x4_i(keyboard_row_x4),
        .keyboard_col_x4_o(keyboard_col_x4),
        .key_flag(key_flag),
        .key_value(key_value)
    );

endmodule
```

经过上面的编写整个工程的所有代码都给大家呈现出来了，接下来我们可以进行仿真了，打开 Modelsim，新建仿真工程，将 keyboaed_4x4_tb 的信号添加至 Wave 窗口观察，如下图 22-12 所示就是仿真结果图。

图 22-12 矩阵键盘扫描逻辑仿真波形

通过观察可以见到矩阵键盘驱动模块检测到的按键次数与按键值与仿真模拟按下的设置是完全匹配的，仿真模拟按下了 3 次，按下的按键分别为 0xa、0x8、0xd。被验证的设计模块正确的检测到按键按下并识别按键具体位置，可以进行更多次的验证确保设计模块功能的正确性。

22.6板级调试

22.6.1硬件连接介绍

小梅哥团队出品的 AC620 开发板、Starter 开发板、AC601 评估板、AC6102、ACX720 和 ACX735 开发板以及后续新推出的开发板都支持通过外接的方式连接矩阵键盘，我们也提供了能够很好的兼容这些板卡的矩阵键盘，如图 22-13 所示。

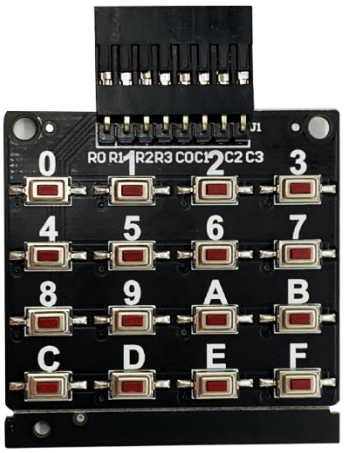


图 22-13 板级测试与 FPGA 相接的矩阵键盘

如下图 22-14 所示为本次实验的硬件连接图。

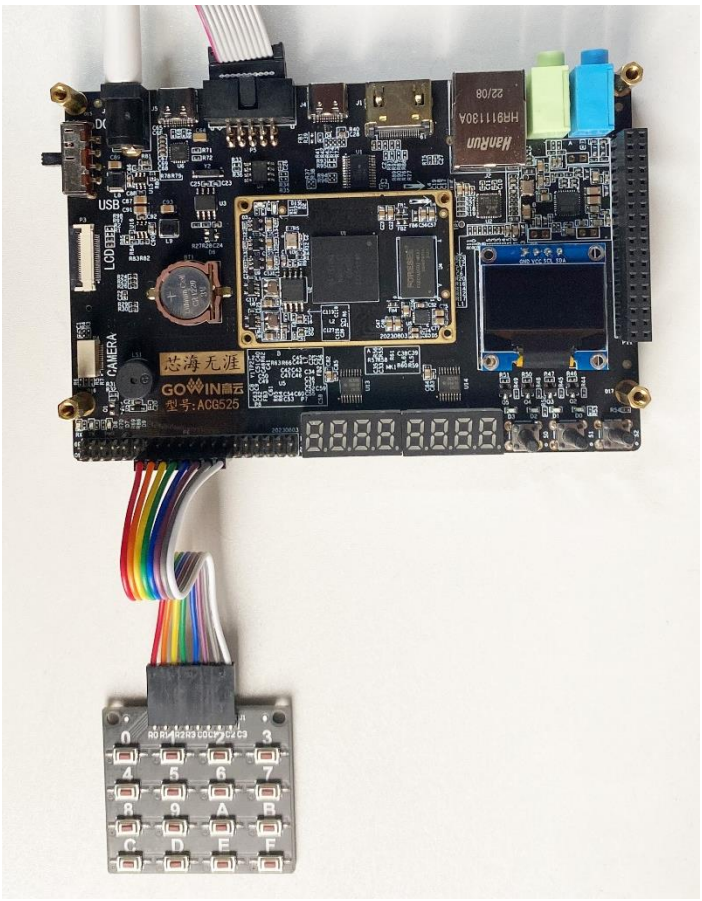


图 22-14 硬件连接图

以上为开发板矩阵键盘插接图，注意：插接方式为矩阵键盘排线插接 GPIO 口靠开发板内侧，也就是通用 GPIO 排针 1 脚那一侧，GPIO 口插接处左右两边均留 6 针。

另外，在之前介绍矩阵键盘检测原理的时候讲解过，对于 ROW 信号，其与

FPGA 的连接管脚上默认是连接了有上拉电阻的，但是市面上很多矩阵键盘默认都是没有提供上拉电阻的，我们开发板选配的矩阵键盘默认也是没有提供上拉电阻的，那么是不是没有上拉电阻就不能完成实验了呢？实际上，我们可以使用 FPGA 片上的 IO 弱上拉电阻来代替外部的上拉电阻。

22.6.2FPGA 的片上弱上拉电阻设置

高云开发板的通用输入输出管脚都支持内部弱上拉电阻，当需要上拉电阻的信号（如本例中的矩阵键盘 ROW 信号和 IIC 协议中的 SDA、SCL 信号）连接到了 FPGA 的通用输入输出管脚上，在一些要求不高的场合，就可以使用片上上拉电阻来为这些信号设置上拉，操作方式就是，依次点击 FloorPlanner->I/O Constraints，进入 I/O 约束界面，然后找到需要添加上拉信号，信号后面对应的 Pull Mode 选项，点击选择 UP，即给对应的信号设置了片上弱上拉，如下图 22-15 所示。

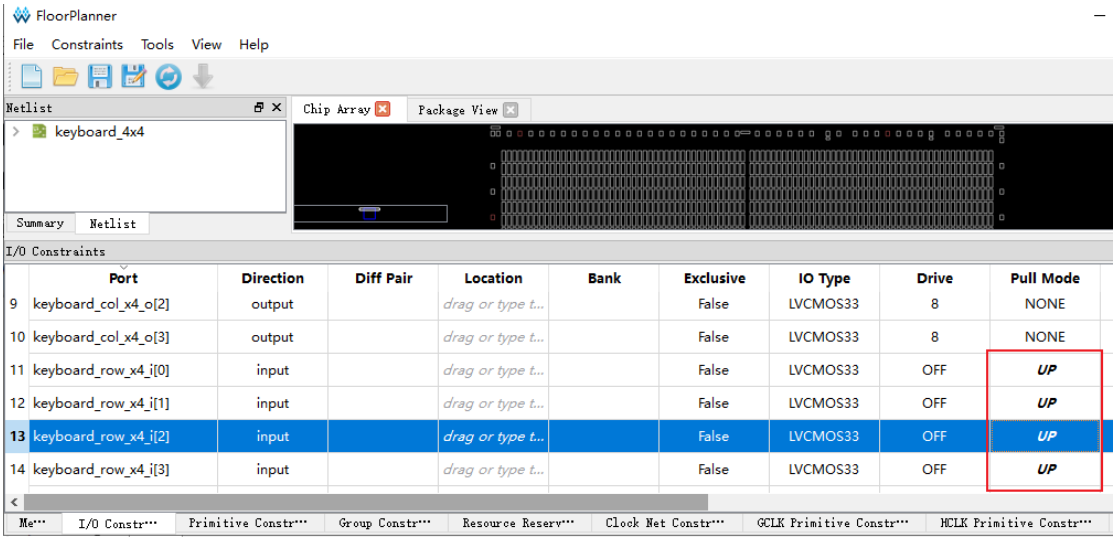


图 22-15 给信号添加弱上拉

设置完上拉之后，我们需要对本次设计分配引脚并约束电平。这里以矩阵键盘连接开发板最靠边外侧的拓展接口为例，其引脚分配如表 22-2 所示。

表 22-2 引脚分配表

Pin Name	Signial Name	Pin NO.	Pin Name	Signial Name	Pin NO.
CLK_G	clk	T9	KEY0	reset_n	B16
GPIO11	keyboard_col_x4_o[3]	H14	GPIO19	keyboard_row_x4_i[3]	L13
GPIO13	keyboard_col_x4_o[2]	K16	GPIO21	keyboard_row_x4_i[2]	H18
GPIO15	keyboard_col_x4_o[1]	K14	GPIO23	keyboard_row_x4_i[1]	K18
GPIO17	keyboard_col_x4_o[0]	K13	GPIO25	keyboard_row_x4_i[0]	L18

完成后如图 22-16 所示。key_flag 和 key_value[0]~key_value[3]信号不需要分

配。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive	Pull Mode
1	clk	input		T9	4	False	LVCMOS33	OFF	NONE
2	key_flag	output		drag or type t...		False	LVCMOS33	8	NONE
3	key_value[0]	output		drag or type t...		False	LVCMOS33	8	NONE
4	key_value[1]	output		drag or type t...		False	LVCMOS33	8	NONE
5	key_value[2]	output		drag or type t...		False	LVCMOS33	8	NONE
6	key_value[3]	output		drag or type t...		False	LVCMOS33	8	NONE
7	keyboard_col_x4_o[0]	output		L15	3	False	LVCMOS33	8	NONE
8	keyboard_col_x4_o[1]	output		H17	3	False	LVCMOS33	8	NONE
9	keyboard_col_x4_o[2]	output		K18	3	False	LVCMOS33	8	NONE
10	keyboard_col_x4_o[3]	output		L18	3	False	LVCMOS33	8	NONE
11	keyboard_row_x4_i[0]	input		T3	5	False	LVCMOS33	OFF	UP
12	keyboard_row_x4_i[1]	input		T7	5	False	LVCMOS33	OFF	UP
13	keyboard_row_x4_i[2]	input		P6	5	False	LVCMOS33	OFF	UP
14	keyboard_row_x4_i[3]	input		T4	5	False	LVCMOS33	OFF	UP
15	reset_n	input		V9	4	False	LVCMOS33	OFF	NONE

图 22-16 引脚约束及分配

22.6.3目标板验证

为了方便我们进行板级验证，这里通过设置 GAO 文件，通过 Gowin 在线逻辑分析仪的功能，验证本次实验是否成功。

在线逻辑分析仪的具体使用方式参看对应章节的内容，本章将不再进行说明，只对部分细节内容进行展示。

首先是触发信号，选择 key_flag_pre 信号，也就是按键被按下标志信号，如图 22-17 下所示。

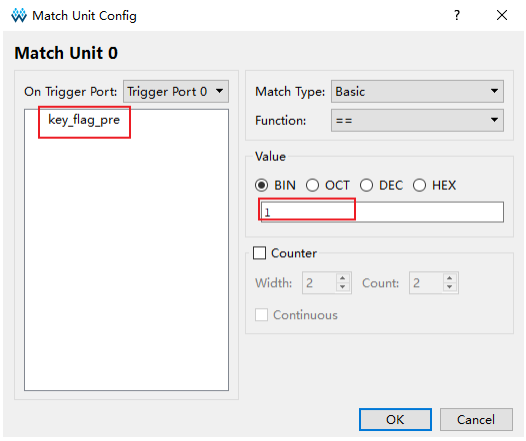


图 22-17 设置触发信号

其次是设置捕获信号，采样时钟为 clk，需要观察的信号为 key_flag_pre 信号和按键值信号 key_value_pre[7:0]，设置如下图 22-18 所示。

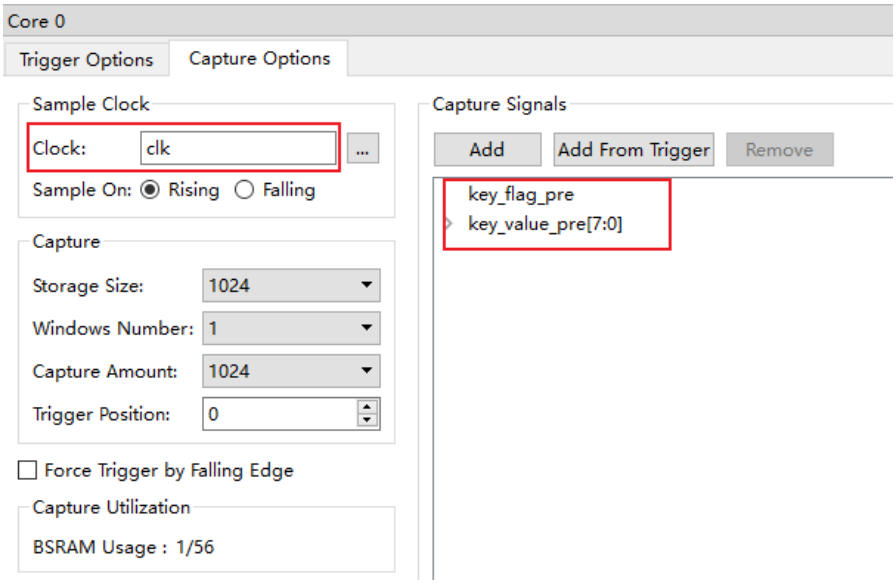


图 22-18 设置捕获信号

其它操作本章将不再重复说明，设置完成之后，保存文件，点击“Place & Route”进行布局布线，成功之后，生成 bit 数据流，然后点击 Program Device 将数据流文件下载至开发板。

下载完成之后，点击进入 Gowin Analyzer Oscilloscope 界面，点击开始捕获信号，按下按键之后，将会捕获到信号，比如我们这里按下按键 A，捕获信号如下图 22-19 所示。

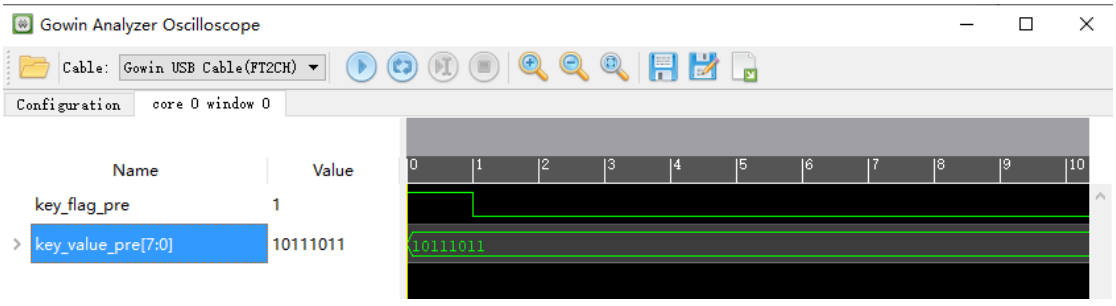


图 22-19 按键‘A’按下捕获波形图

从上图可以看出，key_flag_pre 从高电平变成了低电平，表明按键被按下，key_value_pre[7:0]对应的值为 8’b1011_1011，其对应的按键值就是 A，符合设计预期。读者可以点击再次捕获，按下不同的按键，key_value_pre[7:0]的值会变成对应的按键值。

至此，本次板级验证成功，设计的矩阵键盘驱动模块能够稳定且高效地达到预期效果。

22.7总结

本设计实例通过状态机的设计思想实现了矩阵键盘的扫描以及消抖工作，同时介绍了在 FPGA 上设置 IO 口的片上弱上拉电阻的方法，该方法尤其适用于做 IIC 接口时，在总线上没有上拉电阻的情况下临时顶替。本矩阵键盘消抖模块具有一定的实用性，用户可自行用于自己设计的系统中。

23 VGA 显示驱动设计与验证

工程源码	|----02_设计实例 |---- ch23_VGA_CTRL_GM7123
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

在本章节，我们将重点介绍基于 FPGA 的 VGA 信号输出驱动开发。关于本章节内容，我们将依次进行如下内容学习：

（1）以 VGA 为切入点，着重讲解 VGA 的成像原理。

（2）对 VGA 核心模块进行仿真验证。

（3）以 VGA 彩条实验为目标进行板级验证。

23.1VGA 开发概述

在基于 FPGA 的数字系统设计中，VGA 显示实验是一个大家绕不开的话题，毕竟使用 FPGA 就能驱动平常只有电脑才能驱动得了的大型显示器，相较于普通的 MCU 一般只能点亮普通的低分辨率小尺寸 LCD 液晶屏，驱动 VGA 显示器显得更加的令人感兴趣。

然而事实上，使用 FPGA 驱动 VGA 显示器，从底层和应用两个方向来说，都与普通的 MCU 驱动小尺寸 LCD 液晶屏有着较大的差异。

首先从底层驱动来说，使用 FPGA 驱动 VGA 显示器在指定位置显示某一指定颜色并不难，一个具有基本的 FPGA 逻辑设计能力的人在了解了 VGA 的驱动时序之后，大概半小时就能写出一个易用的 VGA 控制器，相较于使用小尺寸 MCU 接口的 LCD 液晶屏需要查阅一大堆的寄存器来配置其工作模式，显然 VGA 控制器的开发难度要小很多。

其次从功能应用来说，使用 FPGA 驱动 VGA 显示器显示复杂多变的图案就显得异常的麻烦了。不仅开发周期长，而且灵活性很低。不适合用来显示人机交互类的图案（所谓人机交互类图案就是指各种多变的文字信息，以及按钮信息等）。

使用 FPGA 驱动 VGA 虽然不适合用来显示复杂多变的图像内容，但是却常

用于显示实时图像内容，例如显示图像传感器实时采集到的图像。此种方式数据流由图像传感器实时提供，FPGA 只需要控制好图像数据流的存储和传输即可，无需主动生成需要绘制的图像数据，所以实现相对简单。

鉴于当前基于 FPGA 的图像处理是一个非常热门的应用方向，而在图像处理中，通过显示器实时查看显示内容属于必备功能，因此本节将详细介绍 VGA 时序及在 FPGA 中实现 VGA 控制器的方法。

23.2VGA 显示器成像原理

在 VGA 标准刚兴起的时候，常见的 VGA 接口彩色显示器一般基于 CRT（阴极射线管）实现，色彩由 RGB 三基色组成，显示是用逐行扫描的方式。下图 23-1 为基于 CRT 的显示器实物图，相信现在很多人已经没有见过这种显示器了。

图 23-1 CRT 显示器样式图

阴极射线枪发出的电子束打在涂有荧光粉的荧光屏上，产生 RGB 三基色，合成一个彩色像素，扫描从屏幕的左上方开始，从左到右，从上到下进行扫描，每扫完一行，电子束都回到屏幕的下一行左边的起始位置。

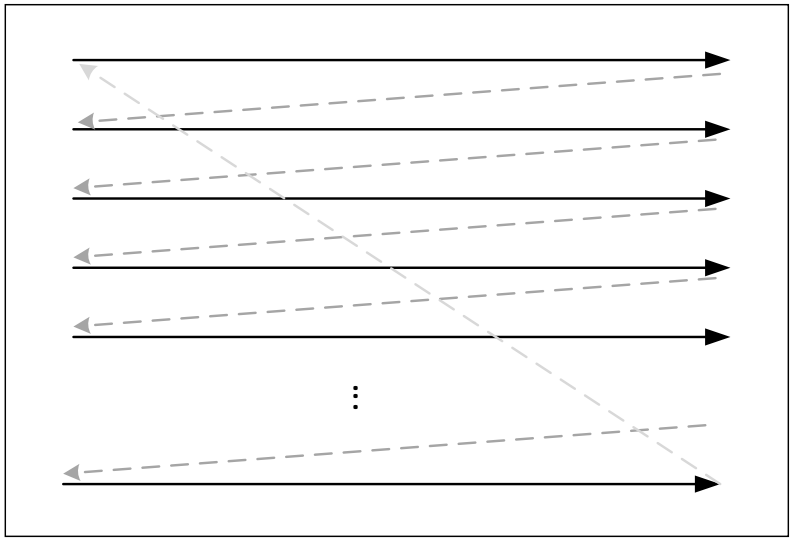


图 23-2 屏幕扫描形式

在回扫的过程中，电子枪不能发射电子，否则会影响荧光屏上既有图像的颜色，所以回扫期间，需要进行行消隐，简单来说就是关闭电子枪。每行结束时，用行同步信号进行行同步，图中从右上方向左下方的斜向虚线就是其回行扫示意图。

当整个屏幕的所有行都扫描完后，使用场同步信号进行场同步，并使扫描回到屏幕的左上方。同样的，为了避免电子枪在回到左上方的过程中发出的电子破坏荧光屏上既有的图像内容，这个回扫的过程也需要关闭电子枪，即场消隐。

随着显示技术的发展，出现了液晶显示器，液晶显示器让显示设备彻底摆脱了厚重的机身，也为便携式计算机的出现创造了可能。

液晶显示器的成像原理与 CRT 不同。液晶显示器是通过改变对液晶像素点单元施加电压的电压大小，来改变液晶单元的透光性。在液晶单元背后发射白光，并添加三色滤光片，分别使 R、G、B 这 3 种光线透过滤光片，最后通过 3 个像素点合成一个彩色像素点，从而实现彩色显示。

由于液晶技术晚于 CRT 显示技术诞生，在液晶显示器出现的时候，计算机显示接口已经确定，很难再突然改变。所以为了能够兼容传统的显示接口，液晶显示器通过内部电路实现了对 VGA 接口的完全兼容。因此，在使用显示器时，只要该显示器带有标准的 VGA 接口，就不用去关注其成像原理，直接使用标准的 VGA 时序即可驱动。

当使用 VGA 接口传输图像时，显示驱动芯片（如显卡）输出的 RGB 数据先要经过 DAC 转换为 3 路分别代表 R、G、B 颜色分量的模拟信号，送到 VGA

接口，这些模拟信号经由 VGA 线缆到达显示器的 VGA 接口，对于模拟的 CRT 显示器，这些信号会直接被放大后用于驱动电子枪发射电子，而对于液晶显示器，则需要显示器使用专门的模拟数字转换芯片将模拟信号再转换为数字信号后，去驱动 RGB 接口的液晶显示屏显示图像。

23.3VGA 时序详解

上一小节的内容中，我们介绍了基于阴极射线管（CRT）的 VGA 显示器的显示原理，那么具体到每一个像素的图像，相关信号的时序究竟是怎样的呢？本节内容就针对该时序进行详细介绍。

由于 VGA 时序的诞生，最早就是为了驱动阴极射线管（CRT）。也因为 CRT 的物理特性，才有了多个与之相关的物理时间参数，所以为了让大家彻底理解 VGA 时序中各个参数的物理意义，这里以一张图像在基于 CRT 结构的显示器上显示为例，介绍显示器完整展现这幅图的各个阶段和每个阶段的物理意义。

下图 23-3 中，白底的卡通人物图像就是我们希望显示的图像内容。我们的目的就是要让该图像恰好完全显示在显示器上。而 CRT 显示器是基于电子枪的，通过电子轰击荧光粉来产生明暗不同的亮度，从而实现图像线索。所以这一幅图像，可以理解为横向的很多行图像向下依次平铺构成的，而每一行图像，又可以理解为由多个像素点从左向右依次平铺构成的。电子枪每次只能点亮一个点上的荧光粉，所以需要像我们人眼看文章一样，从左向右一个字一个字的看，看完一行内容后，再把视线回到左侧，另起一行开始看，这也就是 CRT 的成像方式。

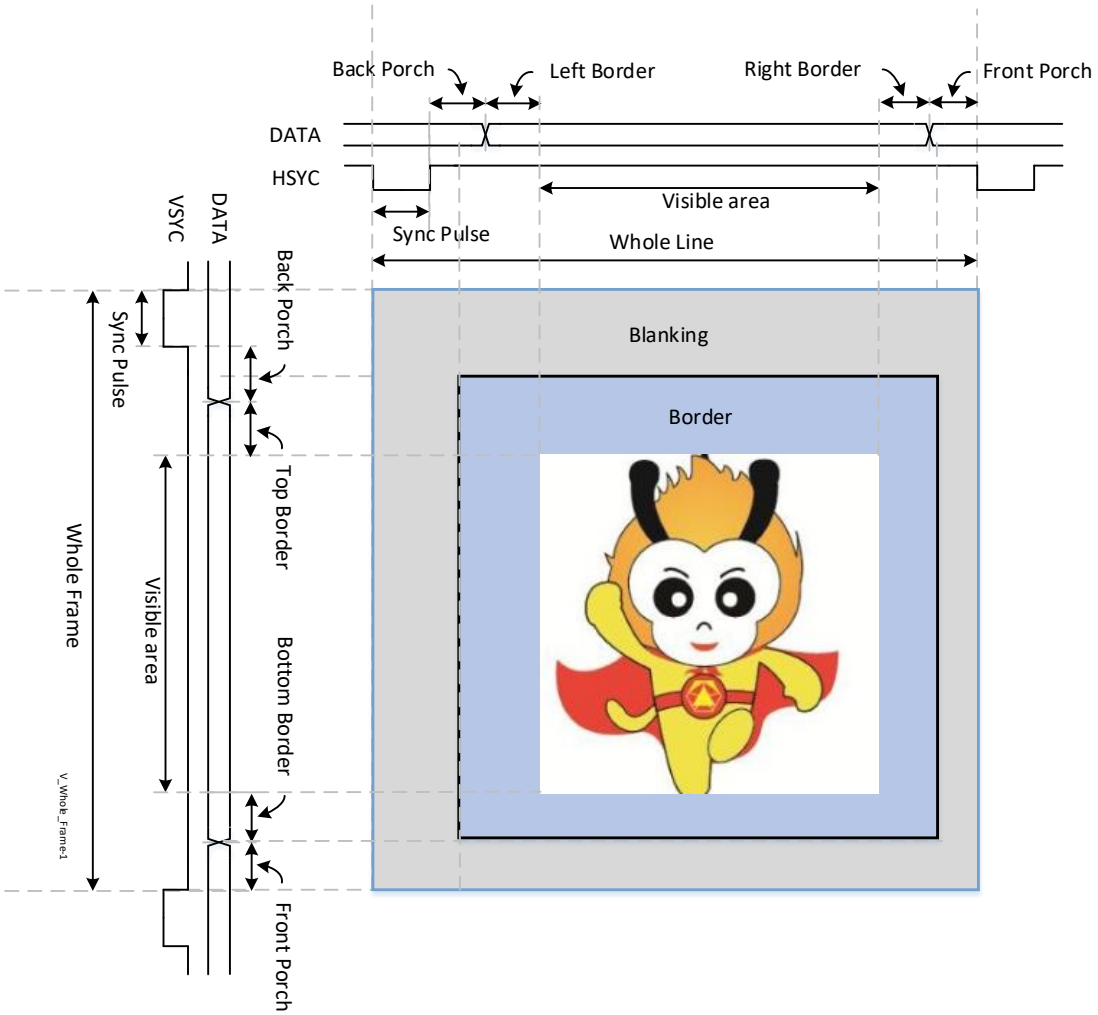


图 23-3 CRT 成像方式

23.3.1CRT 行扫描过程

对于 CRT 显示器，虽然扫描的时候是按照一行一行的方式进行的，但不是扫描完一行有效数据段之后就立马返回，而是会继续向右扫描一段区域，这个区域称为右边界区域（horizontal right border），该区域已经不在有效的显示范围内，如果从物理结构的角度来说，这一段对应的荧光屏玻璃上就不再有荧光粉了，但是电子枪还在继续向右走，大家可以形象理解为显示器右边的黑边。同样的，显示器左边也有这样一段黑边，在开始显示有效数据之前，电子枪扫描到的这段区域同样也是没有荧光粉的，不会显示图像, 这个区域称为左边界区域（horizontal left border）。

那么，电子枪什么时候会到最左侧准备开始新一行图像的扫描呢？当电子枪扫描一行图像到达荧光屏的最右端后，其并不会自动回到最左边准备下一行，

而是需要有一个通知信号，通知其回去，这个通知信号就是行同步信号脉冲（horizontal sync pulse）。行同步信号是一个脉冲，当该脉冲出现后，电子枪的指向会在一定时间内从最右侧回到显示屏的最左侧。而这个回去的过程需要耗费一定的时间，这个时间就称为 horizontal back porch。这也是这个名词中 back 的意义所在，即出现行同步信号后，电子枪从显示屏最右侧回到最左侧的时间。

当电子枪扫描过了右侧没有荧光粉的区域后，还没有收到回到最左侧的命令（行同步信号脉冲）之前，电子枪需要关闭以实现消隐，这个消隐的时间段就称为 horizontal front porch，直观一点理解就是完成了一行图像的扫描，但还没收到回到最左侧命令之前的一段时间。这也是这个名词中 front 的意义所在。

23.3.2CRT 场扫描过程

一幅完整的图像可以看作是多行图像平铺构成的，所以理解了行扫描的过程中每个时间段对应的时间参数名称之后，再来理解场扫描中的名词就非常简单了。

首先来讲，CRT 在扫描一行图像的时候，电子枪的水平位置是保持稳定不变的，而当一行图像扫描完成，开始扫描下一行图像的时候，电子枪的水平位置会向下调整一定的值。因此，我们可以认为，场时序就是在垂直方向上从上往下依次扫描。

其次来说，对于 CRT 显示器来说，其不是扫描完所有行的图像后就立马返回最上方，而是会继续向下扫描一段区域，这个区域称为下边界区域（vertical bottom border），该区域已经不在有效的显示范围内，如果从物理结构的角度来说，这一段对应的荧光屏玻璃上就不再有荧光粉了，但是电子枪还在继续向下走，大家可以形象理解为显示器下边的黑边。同样的，显示器上边也有这样一段黑边，在开始显示有效数据之前，电子枪扫描到的这段区域同样也是没有荧光粉的，不会显示图像，这个区域称为上边界区域（vertical top border）。

再来说说，电子枪什么时候会到最上方准备开始新一场图像的扫描。当电子枪扫描一场图像到达荧光屏的最下方后，其并不会自动回到最上边准备下一场，而是需要有一个通知信号，通知其回去，这个通知信号就是场同步信号脉冲（vertical sync pulse）。场同步信号是一个脉冲，当该脉冲出现后，电子枪的指向会在一定时间内从最下方回到显示屏的最上方。而这个回去的过程需要耗费一定的时间，这个时间就称为 vertical back porch。这也是这个名词中 back 的意义所在，即出现场同步信号后，电子枪从显示屏最下方回到最上方的时间。

当电子枪扫描过了下方没有荧光粉的区域后，还没有收到回到最上方的命令（场同步信号脉冲）之前，电子枪需要关闭以实现消隐，这个消隐的时间段就称为 vertical front porch，直观一点理解就是完成了一场图像的扫描，但还没收到回到最上方命令之前的一段时间。这也是这个名词中 front 的意义所在。

23.3.3VGA 时序标准

了解了扫描一场完整的图像的过程和其中各个阶段的物理意义之后，再看上面图中的各个参数和所处的位置，以及其作用，就非常的直观了。

在上面介绍行、场同步脉冲信号的时候，我们只说了是脉冲信号，但是并未定义脉冲信号的极性，VGA 时序标准支持四种极性，但是并不是所有的显示设备都支持这四种极性，本文默认以应用最为广泛的行、场同步信号都为低脉冲进行介绍。所谓行、场同步信号都是低电平，就是说在产生同步脉冲信号的时候，行、场同步信号变为低电平，其他时刻为高电平。下图 23-4、图 23-5 分别为行扫描时序和场扫描时序的示意图。

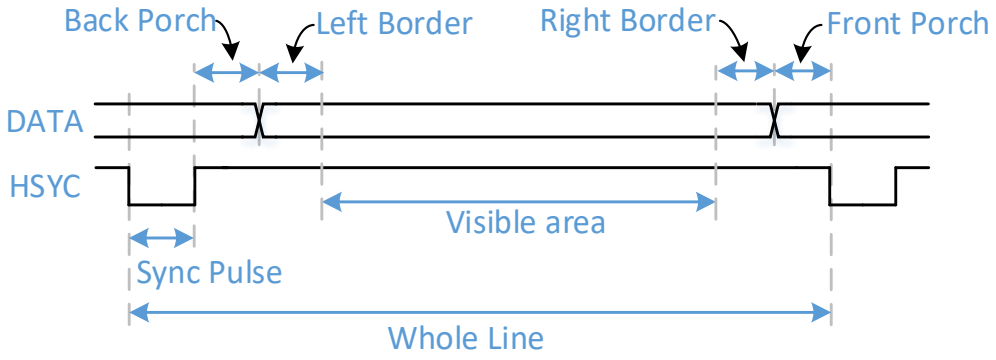


图 23-4 行扫描时序图

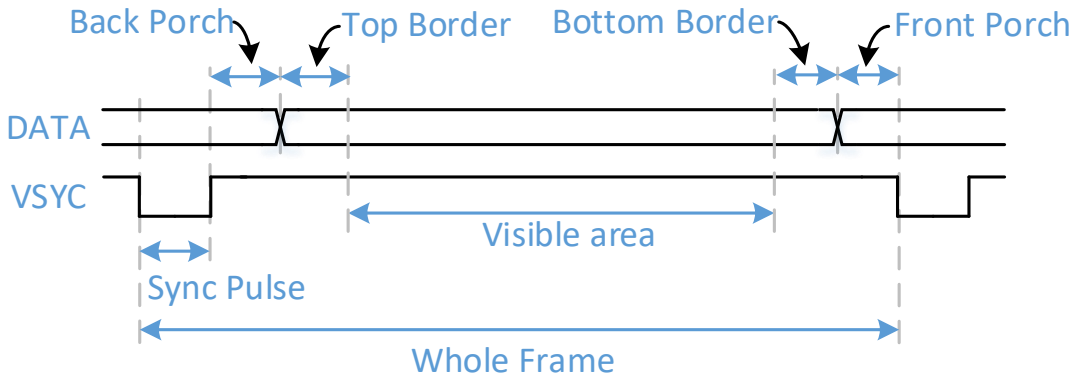


图 23-5 场扫描时序图

上述两幅图中，都只给出了时序参数的名称，并没有给出每个参数具体的

值是多少。而每个参数具体的值是多少，并不是固定的，而是根据需要扫描的有效图像区域的大小确定的。需要扫描的有效图像区域的大小，一般用分辨率来表示。例如标准 VGA 时序的分辨率就是 640*480 个像素点。很显然，这个分辨率放在 2K、4K 大行其道的今天，已经是非常非常低的了，所以除了 VGA 标准，还有很多表示更大或更小分辨率的标准，如 QVGA、SVGA、XVGA 等等，下表 23-1 为几个常见的分辨率名词和与之对应的像素矩阵大小。

表 23-1 各分辨率标准对应像素矩阵表

分辨率标准	像素矩阵大小
SQCIF	12*96
QCIF	176*144
CIF	352*288
QVGA	320*240
VGA	640*480
SVGA	800*600
WSVGA	1024*600
XGA	1024*768
XVGA	1280*960
UXGA	1600*1200

虽然上表列出了众多的分辨率标准的名词，但是记这些名词貌似在大多数场合下并没有太大的意义，反不如直接使用像素矩阵大小的数据来的直观。

23.4各常见分辨率时序参数

下表 23-2 给出了若干个常见分辨率对应的行场时序中各个参数的具体数值，注意，这些参数值中，行相关的参数都是以像素的更新频率，也就是像素时钟作为单位，而场相关的参数，则是以行作为单位。

表 23-2 行场时序参数表

	480 272	640 480	800 480	800 600	1024 600	1024 768	1280 720	1920 1080
H Right Borde	0	8	0	0	0	0	0	0
H Front Porch	2	8	40	40	24	24	110	88
H Sync Time	41	96	128	128	136	136	40	44
H Back Porch	2	40	88	88	160	160	220	148
H Left Border	0	8	0	0	0	0	0	0
H Data Time	480	640	800	800	1024	1024	1280	1920
H Total Time	525	800	1056	1056	1344	1344	1650	2200
V Bottom Borde	0	8	8	0	0	0	0	0
V Front Porch	2	2	2	1	1	3	5	4
V Sync Time	10	2	2	4	4	6	5	5
V Back Porch	2	25	25	23	23	29	20	36
V Top Border	0	8	8	0	0	0	0	0
V Data Time	272	480	480	600	600	768	720	1080

V Total Time	286	525	525	628	628	806	750	1125

上表中，大部分参数都来源于视频电子标准协会（Video Electronics Standards Association）制定的显示器时序标准（Monitor Timing Standard）。当然，也有部分分辨率是该协会没有制定，由显示设备厂家自己定义并世界范围内流通的标准，例如 480*272 分辨率就没有在这个标准里面，但是 sony 的 psp 设备使用了该分辨率的显示屏，后续引发其他厂家也生产相同规格的显示屏并最终大量应用到各种多媒体终端设备，如当年风靡一时的 MP4 播放器、车载导航设备等。在我们前期提供的 4.3 寸显示屏中，就使用了该分辨率。

这里再补充介绍下上表列出的各分辨率的常见显示屏设备。

- 480*272：这个刚刚已经说过了，最早用于 sony 的 psp 设备中，后来被大量应用到各种 MP4 播放器和车载导航设备中。我们前期也提供了一款该分辨率，尺寸为 4.3 寸，支持电阻触摸功能的显示屏模组。
- 640*480：标准的 VGA 分辨率，是所有显示器都必须支持的标准，所有的计算机设备都要求支持该分辨率图像的输出，计算机设备在刚开机的时候，BISO 阶段，显卡设备没有开始工作，由 CPU 直接驱动显示器，使用的就是这个分辨率。另外，该分辨率在 FPGA 相关的应用中还有一个更为大家熟知的存在——OV7670、OV7725 等众多 30W 像素的图像传感器输出的图像大小就是这个分辨率，所以为了显示 30W 像素的摄像头图像，也会按照此分辨率驱动 VGA 显示器。
- 800*480：该分辨率的显示屏在 2010 年~2013 年的智能手机中经常用到。如当年的中兴 V880、华为 U8800、酷派 8017、联想乐 phone（对，上述列举的就是当年名噪一时的中华酷联，由于笔者当时从事相关工作，所以对这些型号比较熟悉）。针对 FPGA 开发板，我们也提供了一款该分辨率，尺寸为 4.3 或 5 寸，支持电容触摸功能的显示屏模组。
- 800*600：很多老式的 CRT 显示器最高就是支持这个分辨率。
- 1024*600：这个分辨率大概是并行 RGB 接口中分辨率最高的一款了吧，目前网络上销售的非常多的 7 寸触摸屏模组，使用的就是这个分辨率的显示屏。
- 1024*768：很经典的一个型号，早期（2000 年~2005 年）的方屏笔记本电脑，使用的就是这个分辨率，笔者的第一台笔记本（高三时候在网上买的二手洋垃圾）就是这个分辨率的。
- 1280*720：也就是我们常说的 720P 视频的分辨率。但是这个分辨率的

屏幕其实并不太常用，用的是 1280*800 的分辨率。1280*800 分辨率的屏幕广泛应用于各种 7 寸的安卓或 win8 平板电脑中。但是 1280*720 这个分辨率，在 FPGA 系统中实现 720p 的实时图像显示时候有用到，所以这里也列出了其时序参数。

- 1920*1080：这个分辨率，就不用多说了，1080p 标准，相信大家都熟悉。现在的笔记本电脑，显示屏的分辨率最低标准貌似就是这个分辨率了，而经典的如 iphone plus 系列手机，使用的也都是这个分辨率的显示屏。

23.5VGA 控制器设计思路

了解了整个 VGA 扫描时序的详细参数之后，设计一个能够驱动 VGA 显示器显示正常图像内容的控制器就非常的简单了。

首先确定，我们的设计目的是设计一个能够产生符合 VGA 显示时序的逻辑控制器，为了实现这个目的，我们需要产生用于行同步的行同步信号脉冲（horizontal sync pulse）、用于场同步的场同步信号脉冲（vertical sync pulse）、并在一行图像的显示有效区间内将需要显示的图像内容的对应像素点的颜色数据输出，在一场图像的显示有效区间内使能将需要显示的图像内容输出。据此，整个设计就变得非常简单了。如果以场同步脉冲的下降沿作为一场图像的起始时刻，以行同步信号的同步脉冲的下降沿作为一行图像的起始时刻，那么每行图像的扫描时序都可以看成是一个依线性序列的操作，设计时只需要在序列的指定时刻产生制定的操作即可。例如对于 640*480 分辨率的时序，其完整的一行包括 800 个像素时钟周期，因此我们设计时，只需要使用一个计数器循环计数 800 个时钟周期，并在对应的计数值时候产生相应的动作，例如：

在计数值为0时刻，拉低行同步信号（HSYNC），以示产生行同步低脉冲，然后让行同步信号（HSYNC）保持 H Sync Time 个时钟周期低电平，此阶段就是 H Sync Time 时序。此阶段处于消隐阶段，消隐信号保持为有效电平。

拉高行同步信号（HSYNC）并保持 H Bach Porch 个时钟周期的高电平，此阶段就是回扫阶段。此时数据总线应该保持全 0 状态。且消隐信号也保持为有效电平。

让行同步信号（HSYNC）继续保持 H Left Border 个时钟周期的高电平，该阶段就是左廊阶段。此时数据总线应该保持全 0 状态。消隐信号被关闭。

正式进入输出图像数据的阶段后，在 H Data Time 阶段，行同步信号

（HSYNC）继续保持高电平，并在 RGB 数据总线上每个时钟周期输出一个颜色数据。

当 H Data Time 个数据输出完成后，进入 H Right Border 阶段，此时，继续保持行同步信号（HSYNC）为高电平。但是数据总线不再输出颜色数据。

H Right Border 阶段过后，进入 H Front Porch 阶段，此阶段继续保持行同步信号（HSYNC）为高电平，数据线不再输出颜色数据，且将消隐信号开启。至此，一行图像的扫描过程结束。

表 23-3 行扫描的一行数据区间构成

H Sync Time	H Bach Porch	H Left Border	H Data Time	H Right Border	H Front Porch
96 vclk	40 vclk	8 vclk	640 vclk	8 vclk	8 vclk

分析完了行扫描时序的实现方案，再来设计场扫描时序的实现方案就非常简单了，思路和行扫描时序的实现方案完全一致，区别仅在于，场扫描时序中的时序参数都是以完成一行扫描所耗费的时间为基本时间单位的。

表 23-4 场扫描的一行数据区间构成

V Sync Time	V Bach Porch	V Top Border	V Data Time	V Bottom Border	V Front Porch
2 lines	25 lines	8 lines	480 lines	8 lines	2 lines

了解了整体的设计思路后，设计该控制器时需要编写的逻辑内容就非常的清晰。设计时，可用两个计数器进行计数（行、场扫描计数器），行扫描计数器的驱动时钟就是基本的像素时钟信号，场计数器的驱动自加使能使用行计数器的溢出信号。根据计数器的实时计数值控制行、场同步信号输出，并在适当的时候送出数据，就能让图像在 VGA 显示器上正确的显示。

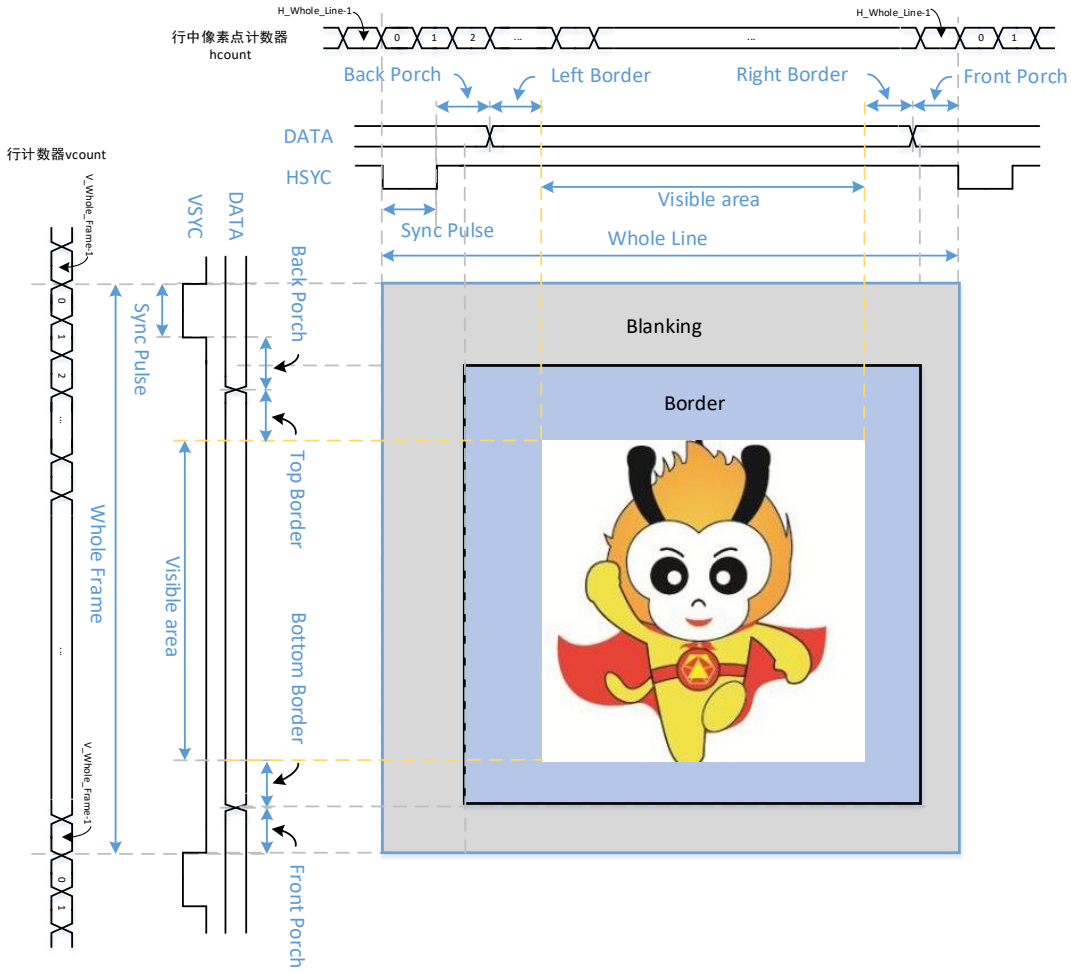


图 23-6 VGA 成像示意图

这样，VGA 成像的基本时序，就介绍完成了，接下来即开始着手设计 640*480 分辨率 VGA 控制器。

23.6640*480 分辨率 VGA 控制器设计

基于上述对 VGA 时序的详细分析，设计一个 VGA 控制器主要也就包括行计数器、场计数器，以及行、场同步信号在合适的时刻产生低电平脉冲，以及在显示有效区域将图像数据输出。本节内容将以 640*480 这个标准的 VGA 分辨率为例，介绍 VGA 控制器的设计方法。

23.6.1640*480 分辨率 VGA 控制器时序分析

通过上述对于 VGA 时序的分析，我们已经了解了 VGA 时序的基本特征，而对于一个特定分辨率的时序标准来说，在什么时候产生 HS、VS 的脉冲信号，

在什么时候输出有效图像数据都是确定的，所以，我们可以根据该分辨率的时序特征，使用 2 个计数器分别来标记行和场整个周期中的每一个时间点，然后在相应的时间点拉高或拉低 HS、VS 信号，输出 RGB 数据，即可实现 VGA 时序的生成。

对于行扫描来说，主要关心的就是在什么时刻产生 HS 脉冲信号，以及在一行中的什么位置输出图像内容。如果以 HS 信号的下降沿作为一行扫描的起点时刻（此时刻为 0 时刻），那么我们只需要知道在第几个时刻 HS 脉冲信号结束（VGA_HS_end），并知道图像数据在一行中什么时候开始输出（hdat_begin）、什么时刻结束输出（hdat_end），还需要知道一行扫描到什么时刻停止（hpixel_end）。所以，对于 640*480 分辨率的时序，据此可以归纳得到如下 4 个与行图像扫描相关的参数：

表 23-5 图像行扫描以每行最左侧为同一原点的绝对参数

时间节点参数	VGA_HS_end	hdat_begin	hdat_end	hpixel_end
时间节点值（clk）	95	143	783	799
时间节点参数构成	H Sync Time - 1	VGA_HS_end + H Bach Porch + H Left Border	hdat_begin + H Data Time	hdat_end + H Right Border + H Front Porch

同样的，对于场扫描来说，主要关心的是在什么时刻产生 VS 脉冲信号，以及在一场中的什么位置输出图像内容。如果以 VS 信号的下降沿作为一行扫描的起点时刻（此时对应 0 行），那么我们只需要知道在第几个行时 VS 脉冲信号结束（VGA_VS_end），并知道图像数据在一行中什么时刻开始输出（vdat_begin）、什么时刻结束输出（vdat_end），还需要知道一行扫描到什么时刻停止（vline_end）。所以，对于 640*480 分辨率的时序，据此可以归纳得到如下 4 个与场图像扫描相关的参数：

表 23-6 图像场扫描的绝对参数

时间节点参数	VGA_VS_end	vdat_begin	vdat_end	vline_end
时间节点值（clk）	1	34	514	524
时间节点参数构成	V Sync Time - 1	VGA_VS_end + V Bach Porch + V Top Border	vdat_begin + V Data Time	vdat_end + V Bottom Border + V Front Porch

有了以上参数之后，编写逻辑就变得非常简单了，以下分别实现。

23.6.2行扫描计数器设计

行扫描计数器即每个像素时钟自加 1，一旦加满到 799（刚好 800 个时钟周期），也就是 H Total Time 个时钟周期后，计数器清零并重新计数，该部分代码

可如下设计：

```
reg [9:0] hcount_r;      //VGA 行扫描计数器

//**********************VGA 驱动部分**********************
//行扫描计数器
always@(posedge Clk25M or negedge Rst_n)
if(!Rst_n)  //复位时，让行扫描计数器清零
    hcount_r<=10'd0;
else if(hcount_r==10'd799)  //当一行数据扫完后，再次清零行扫描计数器
    hcount_r<=10'd0;
else    //0~799 之间，每个像素时钟时行扫描计数器自加 1
    hcount_r<=hcount_r+10'd1;
```

23.6.3场扫描计数器设计

由于场扫描计数器是在每次一行扫描完成后加 1 的，即场扫描计数器的自加条件是行扫描计数器溢出。所以，场扫描计数器的自加条件为行扫描完成，即“hcount_r == 10'd799”，场扫描计数器代码如下所示：

```
reg [9:0] vcount_r;      //VGA 场扫描计数器

//场扫描
always@(posedge Clk25M or negedge Rst_n)
if(!Rst_n)  //复位时让场计数器清零
    vcount_r<=10'd0;
else if(hcount_r==10'd799) begin    //每次一行扫描完成
    if(vcount_r==10'd524)   //每次一场扫描结束，清零计数器
        vcount_r<=10'd0;
    else
        vcount_r<=vcount_r+10'd1;//场计数器在 0~524，满足条件自加 1
end
else //不满足行扫描结束条件器件，让场扫描计数器保持不变
    vcount_r<=vcount_r;
```

23.6.4行场同步信号设计

根据 VGA 工业标准时序，我们知道每一个完整的 VGA 帧都包含了数据段和消隐段，在消隐段期间，行同步信号和场同步信号分别有一段行同步头和场同步头。在同步期间，对应行同步信号或者场同步信号为低电平。因此我们可以根据行、场计数器的值来确定行、场同步信号的电平状态。对于行同步信号，其行同步头为一行扫描的前 96 个像素时钟周期，因此行同步信号可用如下的简单方式控制：

```
assign VGA_HS=(hcount_r>10’d95);
```

对于场同步信号，其场同步头为一行扫描的前 2 个像素时钟周期，因此行同步信号可用如下的简单方式控制：

```
assign VGA_VS=(vcount_r>10'd1);
```

23.6.5输出数据

VGA 控制器的设计目的是为了驱动 VGA 显示器显示需求的图像内容，因此需要设计数据输出部分，这里，数据来源可以为其它部分产生的图像信号，如摄像头数据、BMP 图片数据。我们在驱动 VGA 时，只需要保证在扫描正确的像素点时，其它部分产生的图像信号能够与该像素点位置对应上，则不需要对图像数据再进行二次处理，但是，在行、场消隐期间，需要保证输出到 VGA 的 RGB 数据线上的数据全部为 0，因此可以设置一个二选一多路器，只有在非消隐期间，VGA 控制器才直接输出其他部分输入的图像数据，而消隐期间则强制输出全 0。

我们可以首先产生一个图像数据有效标志信号，然后使用该标志信号控制 VGA 输出数据的内容，即切换二选一多路器的通道，从而实现消隐期间数据全 0 的功能。

图像数据有效标志信号产生代码如下所示：

```
//数据、同步信号输出
assign dat_act=((hcount_r>=10'd143)&&(hcount_r<10'd783))
            &&((vcount_r>=10'd34)&&(vcount_r<10'd514));
```

dat_act 即为图像数据有效标志信号。

VGA_BLANK 信号和 dat_act 信号功能一致，因此直接将 dat_act 信号赋值给 VGA_BLANK 即可。

```
assign VGA_BLK = dat_act;
```

另外，也可以在消隐期间，设置 RGB 输出端口上数据为 0 来强制产生消隐效果。

消隐强制输出 0 二选一多路器代码如下所示：

```
assign VGA_RGB=(dat_act)?data_in:24'h000000;
```

其中，VGA_RGB 是输出到 VGA 接口上的数据，而 data_in 则是其他模块传递过来的正确的图像数据。

23.6.6输出行列扫描位置

为了使其他模块能够根据当前扫描位置正确的输出图像数据，因此需要将 VGA 控制器的实时扫描位置输出，以供其他模块使用。

```
assign hcount=dat_act?(hcount_r- 10'd143):10'd0;
assign vcount=dat_act?(vcount_r- 10'd34):10'd0;
```

23.6.7输出数据锁存时钟信号

对于 FPGA 上实现的 VGA 控制器，在时钟信号的上升沿输出数据，而对于 DAC 芯片来说，需要在数据的中点采集数据，根据实际板级调试效果，将 VGA 控制器的时钟取反后输出作为 DAC 的数据锁存时钟信号，能够确保 DAC 芯片在锁存数据时数据刚好是稳定的，如下所示：

```
//将 VGA 控制器时钟信号取反输出，作为 DAC 数据锁存信号
assign VGA_CLK = ~Clk25M;
```

23.6.8VGA 控制器设计实例

以上为我们根据直观思维设计的驱动电路，在代码中，直接使用了数字作为运算和比较的内容，这样的编写方式，虽然很直观，但是并不十分利于修改。因此，为了实现易于修改的控制器设计，方便后期简单修改后兼容其他分辨率，对代码进行优化，使用参数化设计。将代码中使用到的一些与时序相关的数字直接使用 parameter 这样的参数进行定义，这样在以后需要修改时间参数时，只需要修改 parameter 定义的内容即可，不需要再深入到代码中一个一个修改。这里不再一一介绍如何修改，只贴出最终设计修改完成的代码，请用户自行比对领悟。

```
module VGA_CTRL(
    Clk25M, //系统输入时钟 25MHZ
    Rst_n,  //复位输入，低电平复位
    data_in,    //待显示数据
    hcount,     //VGA 行扫描计数器
    vcount,     //VGA 场扫描计数器
    VGA_RGB,    //VGA 数据输出
    VGA_HS,     //VGA 行同步信号
    VGA_VS,     //VGA 场同步信号
    VGA_BLK,        //VGA 场消隐信号
    VGA_CLK     //VGA DAC 输出时钟
);

    //----------------模块输入端口----------------
    input  Clk25M;          //系统输入时钟 25MHZ
    input  Rst_n;
    input  [23:0]data_in;     //待显示数据

    //----------------模块输出端口----------------
    output [9:0]hcount;
```

```
    output [9:0]vcount;
    output [23:0]VGA_RGB;  //VGA 数据输出
    output VGA_HS;           //VGA 行同步信号
    output VGA_VS;           //VGA 场同步信号
    output VGA_BLK;     //VGA 场消隐信号
    output VGA_CLK;     //VGA DAC 输出时钟

    //将 VGA 控制器时钟信号取反输出，作为 DAC 数据锁存信号
    assign VGA_CLK = ~Clk25M;

    //---------------内部寄存器定义---------------
    reg [9:0] hcount_r;     //VGA 行扫描计数器
    reg [9:0] vcount_r;     //VGA 场扫描计数器
    //---------------内部连线定义---------------
    wire hcount_ov;
    wire vcount_ov;
    wire dat_act;//有效显示区标定

    //VGA 行、场扫描时序参数表
    parameter VGA_HS_end=10'd95,
                hdat_begin=10'd143,
                hdat_end=10'd783,
                hpixel_end=10'd799,
                VGA_VS_end=10'd1,
                vdat_begin=10'd34,
                vdat_end=10'd514,
                vline_end=10'd524;

    assign hcount=dat_act?(hcount_r-hdat_begin):10'd0;
    assign vcount=dat_act?(vcount_r-vdat_begin):10'd0;

    //**********************VGA 驱动部分**********************
    //行扫描
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)
        hcount_r<=10'd0;
    else if(hcount_ov)
        hcount_r<=10'd0;
    else
        hcount_r<=hcount_r+10'd1;

    assign hcount_ov=(hcount_r==hpixel_end);

    //场扫描
    always@(posedge Clk25M or negedge Rst_n)
    if(!Rst_n)
        vcount_r<=10'd0;
```

```
    else if(hcount_ov) begin
        if(vcount_ov)
            vcount_r<=10'd0;
        else
            vcount_r<=vcount_r+10'd1;
    end
    else
        vcount_r<=vcount_r;

    assign  vcount_ov=(vcount_r==vline_end);

    //数据、同步信号输出
    assign dat_act=((hcount_r>=hdat_begin)&&(hcount_r<hdat_end))
                    &&((vcount_r>=vdat_begin)&&(vcount_r<vdat_end));

    assign VGA_BLK = dat_act;

    assign VGA_HS=(hcount_r>VGA_HS_end);
    assign VGA_VS=(vcount_r>VGA_VS_end);
    assign VGA_RGB=(dat_act)?data_in:24'h000000;

endmodule
```

综上所述，得到该模块的接口图如下图 23-7 所示。

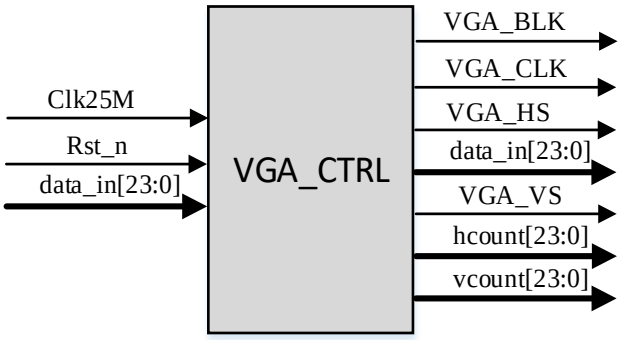


图 23-7 VGA 控制模块接口框图

表 23-7 端口功能表

端口名	端口功能
Clk25M	VGA 像素时钟，25MHz
Rst_n	系统复位，低电平复位
data_in[23:0]	待显示数据输入端口
VGA_HS	VGA 接口行同步信号
VGA_VS	VGA 接口场同步信号
VGA_RGB[23:0]	VGA 三元色数据输出
hcount[9:0]	图像区行扫描地址
vcount[9:0]	图像区场扫描地址

完成了 640*480 分辨率控制器模块设计后，接下来即可进行该模块控制器的仿真验证。

23.7640*480 分辨率 VGA 控制器仿真验证

本小节对设计的 VGA 控制器进行仿真验证，通过仿真查看行场同步信号是否满足设计需求。

23.7.1Testbench 设计

Testbench 的设计思路非常简单，只需要产生一个 25MHz 的时钟信号，然后在 data_in 端口上给一个固定的数据编码，为了与消隐时候的强制输出全 0 相区分，因此只需要是 data_in 上的数据不为 0 即可。testbench 内容如下所示：

```
`timescale 1ns/1ns

`define clk_period 40

module VGA_CTRL_tb;
    //----------------模块输入端口----------------
    reg  Clk25M;           //系统输入时钟 25MHZ
    reg  Rst_n;
    reg  [23:0]data_in;     //待显示数据

    //----------------模块输出端口----------------
    wire [9:0]hcount;
    wire [9:0]vcount;
    wire [23:0]VGA_RGB;  //VGA 数据输出
    wire VGA_HS;           //VGA 行同步信号
    wire VGA_VS;           //VGA 场同步信号

    reg [11:0]V_cnt = 0;//扫描行数统计计数器

    VGA_CTRL VGA_CTRL(
            .Clk25M(Clk25M),     //系统输入时钟 25MHZ
            .Rst_n(Rst_n),
            .data_in(data_in),  //待显示数据
            .hcount(hcount),        //VGA 行扫描计数器
            .vcount(vcount),        //VGA 场扫描计数器
            .VGA_RGB(VGA_RGB),  //VGA 数据输出
            .VGA_HS(VGA_HS),        //VGA 行同步信号
            .VGA_VS(VGA_VS),        //VGA 场同步信号
            .VGA_BLK(VGA_BLK),      //VGA 场消隐信号
            .VGA_CLK(VGA_CLK)   //VGA DAC 输出时钟
```

```
        );

    initial Clk25M = 0;
    always #(`clk_period/2) Clk25M = ~Clk25M;

    initial begin
        Rst_n = 0;
        data_in = 8'd0;
        #(`clk_period *20 +1);
        Rst_n = 1;
        data_in = 24'hffffff;
    end

    initial begin
        wait(V_cnt == 3);//等待扫描 2 帧后结束仿真
        $stop;
    end

    always @(posedge VGA_VS)//统计总扫描帧数
        V_cnt <= V_cnt + 1'b1;

endmodule
```

23.7.2仿真结果分析

23.7.2.1VGA_HS 信号

以下为 VGA_HS 信号的仿真波形。

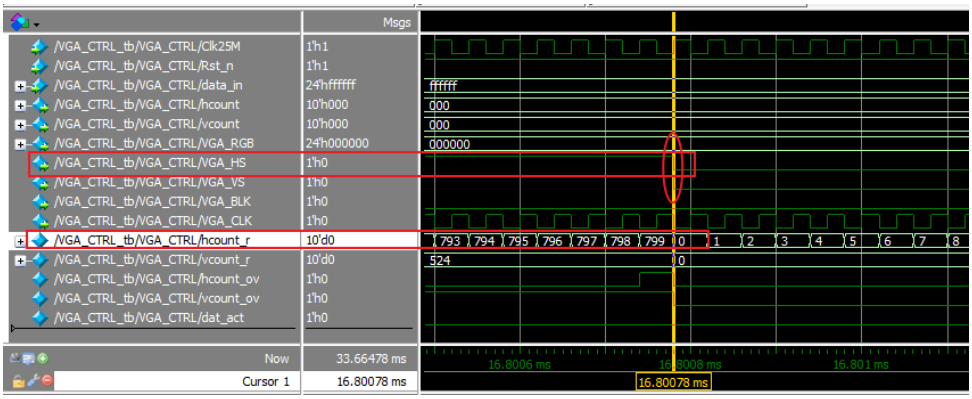


图 23-8 VGA_HS 信号电平转换图 1

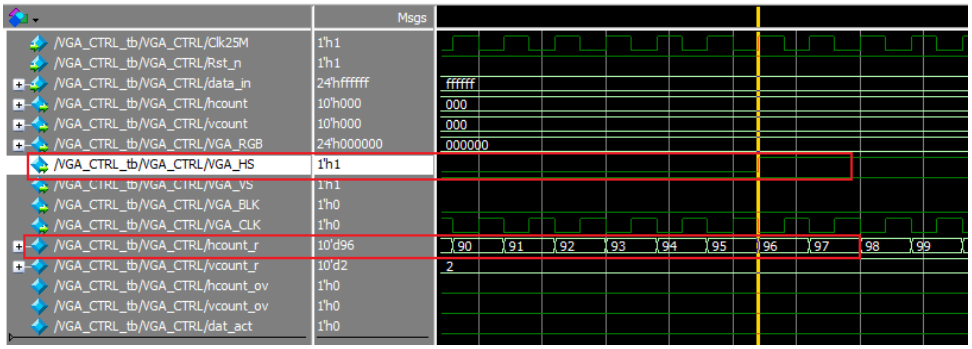


图 23-9 VGA_HS 信号电平转换图 2

由图可见，VGA_HS 在 0~95 这一行扫描段内为低电平，即行同步头，其他时间为高电平，行扫描依次，行扫描计数器计数最大值为 799，即刚好 800 个像素时钟周期，与设计一致，因此可知行扫描信号满足时序设计要求。

23.7.2.2VGA_VS 信号

以下为 VGA_VS 仿真波形图。

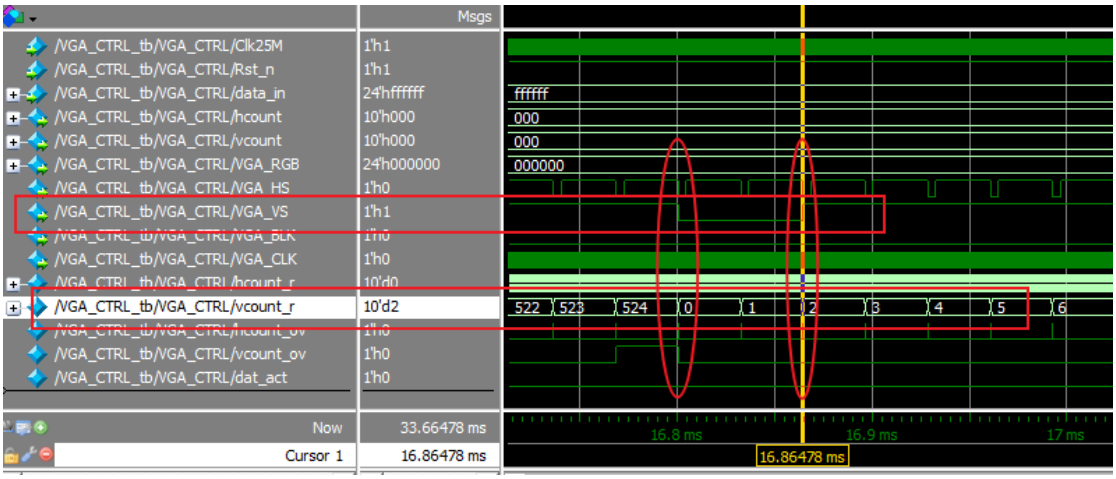


图 23-10 VGA_VS 信号电平转换图

由图可见，VGA_VS 信号在 0~1 这一段场扫描时间内为低电平，即场同步头，其他时间为高电平。场扫描一次，场扫描计数器计数最大值为 524，即刚好 525 个行扫描周期，与设计一致，满足 VGA 工业标准，因此可知场扫描信号满足时序设计要求。

其他信号本文不再进行详细分析比对，在进行板级调试中，如果发现显示效果不对，则可根据实际显示效果，判断错误位置，如行同步信号错误、场同步信号错误等。

23.8640*480 分辨率 VGA 控制器板级验证

前面章节，我们简述了 VGA 控制器的设计思路并给出了具体的 VGA 控制器设计过程，同时通过仿真验证了设计的合理性。本节，我们将对该 VGA 控制器使用彩条实验来进行板级验证，通过板级验证来进一步确定我们设计的正确性。

23.8.1板级验证功能设计

我们设计一个测试工程，该工程中我们测试上述提到的 8 种颜色，通过颜色的位置，不但能确定是否能够正确输出指定颜色的图像，还能间接确定是否能够精确指定像素位置。

因此，我们对屏幕进行划分，将屏幕划分成 4 行 2 列总共八个像素阵列，每个阵列分别显示一种颜色。据此，我们可以首先定义每种颜色的具体数据编码，然后再定义每个像素阵列的基本显示颜色，这里首先使用 localparam 定义每种颜色的具体数据编码：

```
//定义颜色编码
localparam
  BLACK = 24'h000000,//黑色
  BLUE  = 24'h00000F,//蓝色
  RED   = 24'hFF0000,//红色
  PURPPLE =24'hFF00FF,//紫色
  GREEN = 24'h00FF00, //绿色
  CYAN  = 24'h00FFFF, //青色
  YELLOW  = 24'hFFFF00,//黄色
  WHITE   = 24'hFFFFFF;//白色
```

```
//定义每个像素块的默认显示颜色值
localparam
R0_C0 = BLACK,  //第 0 行 0 列像素块
R0_C1 = BLUE,   //第 0 行 1 列像素块
R1_C0 = RED,    //第 1 行 0 列像素块
R1_C1 = PURPPLE,//第 1 行 1 列像素块
R2_C0 = GREEN,  //第 2 行 0 列像素块
R2_C1 = CYAN,   //第 2 行 1 列像素块
R3_C0 = YELLOW, //第 3 行 0 列像素块
R3_C1 = WHITE;  //第 3 行 1 列像素块
```

紧接着，我们需要知道 VGA 当前扫描的位置是在哪一个位置区间，换一种说法，我们需要通过 VGA 当前的扫描位置得到当前扫描的是哪一个像素阵列，然后给待显示数据赋予对应的颜色值即可。这里我们先定义每个像素块处于扫描中的条件。

1. 产生每一列的处于扫描状态标志信号，屏幕每行总共 640 个像素点，我们将屏幕划分成了 2 列，因此，当行扫描范围在 0~319 这一段像素内时，第 0 列处于活跃阶段；当行扫描范围在 320~639 这一段像素内，第 1 列处于活跃阶段，因此可得：

```
wire C0_act = hcount >= 0 && hcount < 320; //正在扫描第 0 列
wire C1_act = hcount >= 320 && hcount < 640;//正在扫描第 1 列
```

2. 产生每一行的处于扫描状态标志信号，屏幕每列总共 480 个像素点，我

们将屏幕划分成了 4 列，因此，当行扫描范围在 0~119 这一段像素内时，第 0 行处于活跃阶段；当行扫描范围在 120~239 这一段像素内时，第 1 行处于活跃阶段；当行扫描范围在 240~359 这一段像素内时，第 2 行处于活跃阶段；当行扫描范围在 360~479 这一段像素内时，第 3 行处于活跃阶段。因此可得：

```
wire R0_act = vcount >= 0 && vcount < 120;  //正在扫描第 0 行
wire R1_act = vcount >= 120 && vcount < 240;//正在扫描第 1 行
wire R2_act = vcount >= 240 && vcount < 360;//正在扫描第 2 行
wire R3_act = vcount >= 360 && vcount < 480;//正在扫描第 3 行
```

3. 产生扫描每一个像素块的标志信号：

```
wire R0_C0_act = R0_act & C0_act;   //第 0 行 0 列像素块被扫描中
wire R0_C1_act = R0_act & C1_act;   //第 0 行 1 列像素块被扫描中
wire R1_C0_act = R1_act & C0_act;   //第 1 行 0 列像素块被扫描中
wire R1_C1_act = R1_act & C1_act;   //第 1 行 1 列像素块被扫描中
wire R2_C0_act = R2_act & C0_act;   //第 2 行 0 列像素块被扫描中
wire R2_C1_act = R2_act & C1_act;   //第 2 行 1 列像素块被扫描中
wire R3_C0_act = R3_act & C0_act;   //第 3 行 0 列像素块被扫描中
wire R3_C1_act = R3_act & C1_act;   //第 3 行 1 列像素块被扫描中
```

然后，我们就可以根据当前被扫描的像素块范围来确定需要给 VGA 输出什么颜色，这里采用一个多路器即可实现：

```
always@(*)
    case({R3_C1_act,R3_C0_act,R2_C1_act,R2_C0_act,
            R1_C1_act,R1_C0_act,R0_C1_act,R0_C0_act})
        8'b0000_0001:disp_data = R0_C0;
        8'b0000_0010:disp_data = R0_C1;
        8'b0000_0100:disp_data = R1_C0;
        8'b0000_1000:disp_data = R1_C1;
        8'b0001_0000:disp_data = R2_C0;
        8'b0010_0000:disp_data = R2_C1;
        8'b0100_0000:disp_data = R3_C0;
        8'b1000_0000:disp_data = R3_C1;
        default:disp_data = R0_C0;
endcase
```

23.8.2添加 PLL 时钟分频单元

通过以上步骤，我们就完成了简易 VGA 控制器测试电路的主要电路设计。在前面我们曾经提到，VGA 控制器的像素时钟为 25Mhz，而我们芯路恒 FPGA 开发板设计的是 50Mhz 的晶振，因此需要使用锁相环对时钟进行分频得到 25M 的时钟，以供 VGA 控制器使用。注意，虽然我们直接使用寄存器二分频也能从 50M 直接分频得到 25M 时钟，但是这样分频出来的时钟驱动能力是非常差的，

抖动也非常大，不能再作为时序电路的时钟使用，因此这里必须使用 pll 来得到 25MHz 时钟。

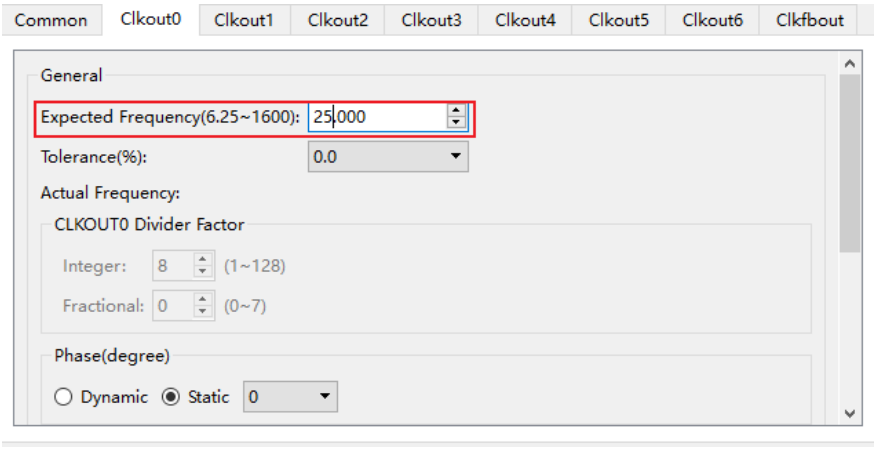


图 23-11 VGA 驱动频率锁相环配置图

23.8.3完整的彩条实验测试电路源码

实现完整的彩条实验测试电路源码如下所示：

```
module VGA_CTRL_test(
    Clk,    //50MHZ 时钟
    Rst_n,
    VGA_RGB,//TFT 数据输出
    VGA_HS, //TFT 行同步信号
    VGA_VS, //TFT 场同步信号
    VGA_BLK,        //VGA 场消隐信号
    VGA_CLK     //VGA DAC 输出时钟
);

    input Clk;
    input Rst_n;
    output [23:0]VGA_RGB;
    output VGA_HS;
    output VGA_VS;
    output VGA_BLK;     //VGA 场消隐信号
    output VGA_CLK;     //VGA DAC 输出时钟

    reg [23:0]disp_data;
    wire [9:0]hcount;
    wire [9:0]vcount;
    wire Clk25M;

    Gowin_PLL Gowin_PLL(
        .clkout0(Clk25M), //output clkout0
        .clkin(Clk) //input clkin
```

```
    );

    VGA_CTRL VGA_CTRL(
        .Clk25M(Clk25M),     //系统输入时钟 25MHZ
        .Rst_n(Rst_n),
        .data_in(disp_data),     //待显示数据
        .hcount(hcount),         //VGA 行扫描计数器
        .vcount(vcount),         //VGA 场扫描计数器
        .VGA_RGB(VGA_RGB),   //VGA 数据输出
        .VGA_HS(VGA_HS),         //VGA 行同步信号
        .VGA_VS(VGA_VS),         //VGA 场同步信号
        .VGA_BLK(VGA_BLK),       //VGA 场消隐信号
        .VGA_CLK(VGA_CLK)    //VGA DAC 输出时钟
    );


//定义颜色编码
localparam
    BLACK       = 24'h000000, //黑色
    BLUE        = 24'h0000FF, //蓝色
    RED     = 24'hFF0000, //红色
    PURPPLE = 24'hFF00FF, //紫色
    GREEN       = 24'h00FF00, //绿色
    CYAN        = 24'h00FFFF, //青色
    YELLOW  = 24'hFFFF00, //黄色
    WHITE       = 24'hFFFFFF; //白色

//定义每个像素块的默认显示颜色值
localparam
    R0_C0 = BLACK,  //第 0 行 0 列像素块
    R0_C1 = BLUE,   //第 0 行 1 列像素块
    R1_C0 = RED,    //第 1 行 0 列像素块
    R1_C1 = PURPPLE,//第 1 行 1 列像素块
    R2_C0 = GREEN,  //第 2 行 0 列像素块
    R2_C1 = CYAN,   //第 2 行 1 列像素块
    R3_C0 = YELLOW, //第 3 行 0 列像素块
    R3_C1 = WHITE;  //第 3 行 1 列像素块

    wire R0_act = vcount >= 0 && vcount < 120;  //正在扫描第 0 行
    wire R1_act = vcount >= 120 && vcount < 240;//正在扫描第 1 行
    wire R2_act = vcount >= 240 && vcount < 360;//正在扫描第 2 行
    wire R3_act = vcount >= 360 && vcount < 480;//正在扫描第 3 行

    wire C0_act = hcount >= 0 && hcount < 320; //正在扫描第 0 列
    wire C1_act = hcount >= 320 && hcount < 640;//正在扫描第 1 列

    wire R0_C0_act=R0_act & C0_act;//第 0 行 0 列像素块处于被扫描中标志信号
```

```
    wire R0_C1_act=R0_act & C1_act;//第 0 行 1 列像素块处于被扫描中标志信号
    wire R1_C0_act=R1_act&C0_act; //第 1 行 0 列像素块处于被扫描中标志信号
    wire R1_C1_act=R1_act & C1_act;//第 1 行 1 列像素块处于被扫描中标志信号
    wire R2_C0_act=R2_act & C0_act;//第 2 行 0 列像素块处于被扫描中标志信号
    wire R2_C1_act=R2_act & C1_act;//第 2 行 1 列像素块处于被扫描中标志信号
    wire R3_C0_act=R3_act & C0_act;//第 3 行 0 列像素块处于被扫描中标志信号
    wire R3_C1_act=R3_act & C1_act;//第 3 行 1 列像素块处于被扫描中标志信号

    always@(*)
        case({R3_C1_act,R3_C0_act,R2_C1_act,R2_C0_act,
                R1_C1_act,R1_C0_act,R0_C1_act,R0_C0_act})
            8'b0000_0001:disp_data = R0_C0;
            8'b0000_0010:disp_data = R0_C1;
            8'b0000_0100:disp_data = R1_C0;
            8'b0000_1000:disp_data = R1_C1;
            8'b0001_0000:disp_data = R2_C0;
            8'b0010_0000:disp_data = R2_C1;
            8'b0100_0000:disp_data = R3_C0;
            8'b1000_0000:disp_data = R3_C1;
            default:disp_data = R0_C0;
        endcase

endmodule
```

基于以上测试源码，即可完成对 VGA 彩条实验的功能测试任务。接下来，我们继续完成板级验证的其他条件准备。

23.8.4系统所需硬件

1. 高云开发板
2. 电源线一根
3. 高云下载器一套
4. VGA 模块 GM7123 一个
5. VGA 电缆一根
6. 支持 VGA 接口的液晶显示器一台(可以和编程 PC 选用同一个液晶显示器，但观察实验现象时需能进行模式切换和分辨率设置)

23.8.5板级验证需求

VGA 的板级验证，主要验证以下三个方面：

1. 能够正确的全屏点亮屏幕，显示稳定。

2. 能否正确的显示颜色，即按照需求定时需要显示的颜色。

3. 能否正确的定位坐标，即实现在指定的位置显示对应的数据。

23.8.6GM7123 模块硬件连接

由于一个 24 位色的 VGA 电路至少需要占用 28 个 I/O，直接集成在开发板上会浪费较多 I/O，所以对于小梅哥 FPGA 的各个 FPGA 开发板，都统一使用一个 36Pin 的通用显示扩展接口，再加上扩展的 VGA 输出模块来实现相应的功能。下图 23-12 所示为提供的 VGA 输出模块。

图 23-12 24 位色 VGA 输出模块

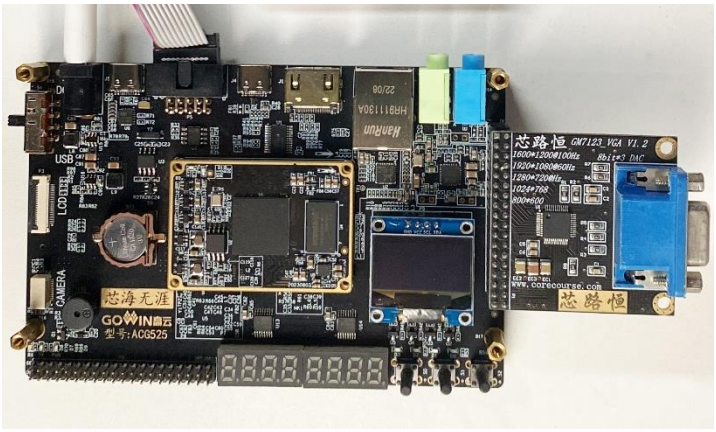


图 23-13 高云开发板连接 VGA 接口的效果图

图中的 VGA 模块使用兼容 ADI 公司 ADV7123 芯片的国产 GM7123 芯片设计，能够实现 24 位色，分辨率高达 1920*1080 的 VGA 图像输出，关于该模块的进一步介绍，可查看对应的“GM7123 模块使用说明.pdf”文档。

连接好 VGA 模块后，使用 VGA 通信电缆将模块输出端与 VGA 显示器的对应接口连接，并将显示器设定为 VGA（640*480）模式，再按开发板的正常供电电缆和下载电缆连接方式连接好开发板和 PC 机。

23.8.7下载与验证

经过上述工作，所有代码都已经设计完毕，硬件环境也已搭建完成。接下来介绍板级测试和验证方法：

1. 按照开发板的引脚分配表分配正确的引脚，本设计实例针对各开发板的管脚分配如下表 23-8 所示。

表 23-8 VGA 驱动模块管脚绑定表

Pin Name	Pin NO.	Pin Name	Pin NO.
Clk	T9	VGA_RGB[13]	M18
Rst_n	V9	VGA_RGB[14]	L15
VGA_RGB[0]	U15	VGA_RGB[15]	L16
VGA_RGB[1]	V15	VGA_RGB[16]	F16
VGA_RGB[2]	U16	VGA_RGB[17]	M13
VGA_RGB[3]	V16	VGA_RGB[18]	J16
VGA_RGB[4]	U17	VGA_RGB[19]	H15
VGA_RGB[5]	U18	VGA_RGB[20]	G13
VGA_RGB[6]	T17	VGA_RGB[21]	H12
VGA_RGB[7]	T18	VGA_RGB[22]	F15
VGA_RGB[8]	M14	VGA_RGB[23]	E18
VGA_RGB[9]	N14	VGA_HS	T11
VGA_RGB[10]	N15	VGA_VS	R11
VGA_RGB[11]	N16	VGA_BLK	L14
VGA_RGB[12]	M16	VGA_CLK	E16

2. 连接好开发板的硬件电路，并全编译工程直至没有错误后下载 bit 文件，最终测试效果如下图 23-14 所示。

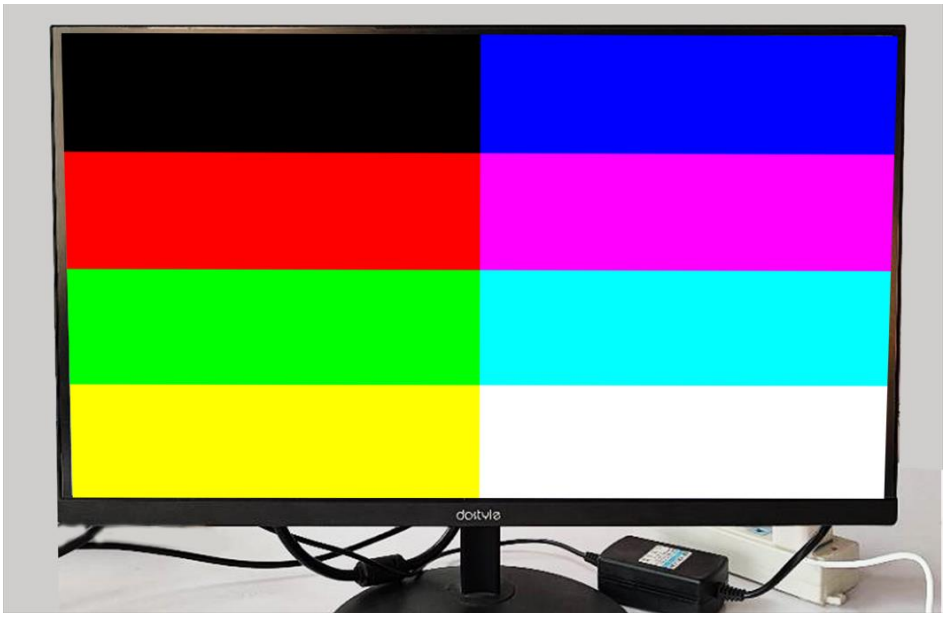


图 23-14 VGA 显示效果图

通过图片可知，VGA 控制器设计能够稳定正确的刷新 VGA 显示器并控制正确的显示位置，因此设计无误。

后续，我们就可以使用该控制器再结合一定的图像信号产生电路实现更多更复杂的显示系统设计。当然，也可能根据具体的使用环境，再对本控制器进行设计微调。

24 多分辨率适配型 VGA 控制器设计

工程源码	无
相关视频课程	
说明	

章节导读

上一节，我们详细讲解了 VGA 时序标准的由来、发展，并在 FPGA 中设计实现了一个针对标准 VGA 分辨率（640*480）的 VGA 控制器，且通过仿真和板级验证的方式验证了其正确性。然而，相信很多读者也都知道，在当今这个 4K 分辨率都快要普及的时代，640*480 这种低分辨率的显示屏，不仅技术上显得落后，而且显示效果也很差，很难满足日益苛刻的人眼审美。

鉴于 TFT 屏显示系统在数字系统中应用实在是非常广泛，掌握 TFT 屏幕的驱动和应用，是非常有实用意义的，所以本节将学习如何使用 FPGA 来进行 RGB 接口的 TFT 屏幕驱动设计。

24.1多分辨率适配的 VGA 控制器设计目的

要实现 4K 分辨率的输出，使用传统的 VGA 驱动和传输方式已经几乎不可能实现，而对于稍微低一点的 1080p(1920*1080 分辨率)、720p(1280*720 分辨率)来说，使用 VGA 接口和传输方式还是能够满足其应用需求的。而这两个分辨率当前在多媒体和工控行业也都还有非常广泛的应用。所以本节，我们将设计一个能够支持从 480*272 分辨率直到 1920*1080 分辨率的可配置型 VGA 控制器。

本节所要设计的 VGA 控制器能够通过简单的代码配置，实现对多种分辨率的驱动时序的支持。本节内容的目的主要有两个，一是在上一节内容的基础上，为大家设计并展示一个支持多种分辨率的实用型 VGA 控制器，二是通过本节内容，让大家进一步熟悉 Verilog 设计中参数化设计的方法。

本节内容将在上一节内容的基础上通过修改得到。在上一节“各常见分辨率时序参数”中，给出了若干个常见分辨率对应的时序参数的数值。同时，在设计 640*480 分辨率 VGA 控制器的时候，也分析了各个时序参数在控制器逻辑中的作用。所以，在实现时，只需要根据不同的分辨率来修改这些时序参数的 parameter 的值即可。

这里再把各个常见分辨率的时序参数值列出来一遍，方便大家进一步熟悉。行相关的参数都是都是以像素的更新频率，也就是像素时钟作为单位，而场相

关的参数，则是以行作为单位。

表 24-1 常见分辨率时序参数表

	480 272	640 480	800 480	800 600	1024 600	1024 768	1280 720	1920 1080
H Right Border	0	8	0	0	0	0	0	0
H Front Porch	2	8	40	40	24	24	110	88
H Sync Time	41	96	128	128	136	136	40	44
H Back Porch	2	40	88	88	160	160	220	148
H Left Border	0	8	0	0	0	0	0	0
H Data Time	480	640	800	800	1024	1024	1280	1920
H Total Time	525	800	1056	1056	1344	1344	1650	2200
V Bottom Border	0	8	8	0	0	0	0	0
V Front Porch	2	2	2	1	1	3	5	4
V Sync Time	10	2	2	4	4	6	5	5
V Back Porch	2	25	25	23	23	29	20	36
V Top Border	0	8	8	0	0	0	0	0
V Data Time	272	480	480	600	600	768	720	1080
V Total Time	286	525	525	628	628	806	750	1125

更多其他分辨率的时序参数值，大家可以查询视频电子标准协会（Video Electronics Standards Association）制定的显示器时序标准（Monitor Timing Standard）的文档，该文档在 www.corecourse.cn 论坛上搜索 VGA 关键词即可下载。

在“640*480 分辨率 VGA 控制器时序分析”小节中，我们通过定义 VGA_HS_end、hdat_begin、hdat_end、hpixel_end、VGA_VS_end、vdat_begin、vdat_end、vline_end 这些时间节点参数来实现了 VGA 控制器的各个信号的产生和数据的准确输出。而这些时间节点参数，又是通过 VGA 分辨率中的各种 Border、Porch、time 参数通过简单的相加运算得到的，所以，可配置参数化设计的第一步，就是要将各个时间节点参数值用 VGA 时序标准中的时序参数值表示。以下为几个时间节点参数与 VGA 时序标准参数的关系。

```
hdat_begin = H_Sync_Time + H_Back_Porch + H_Left_Border - 1'b1;
hdat_end = H_Total_Time - H_Right_Border - H_Front_Porch - 1'b1;
vdat_begin = V_Sync_Time + V_Back_Porch + V_Top_Border - 1'b1;
vdat_end = V_Total_Time - V_Bottom_Border - V_Front_Porch - 1'b1;
```

由于 VGA_HS_end = H_Sync_Time – 1，VGA_VS_end = V_Sync_Time – 1，hpixel_end = H_Total_Time – 1，vline_end = V_Total_Time – 1，所以这几个时间节点参数值没有再单独列出，而是直接使用时序参数值。

有了上述关系表达式，我们只需在使用不同的分辨率时让各个 Border、Porch、time 参数值为对应分辨率的数值即可。为了最简单的实现这种设计，我们可以通过预定义加条件编译的方式来实现。

24.2条件编译原理

所谓条件编译，就是当设计中满足某个条件时，将该条件下的一段代码编译进设计中。因此，我们只需要合理设置编译条件，并在对应的编译条件下编写正确的代码，就可以实现条件编译。以一个最简单的例子来说明。

对于一个最简单的 16 位计数器，我们可以根据不同的需求，设置其计数的最大值，从而来设置其计数一周所占用的时间。假设这个计数最大值用 CNT_MAX 表示，如果定义了工作在条件一（CASE_A）的情况下，该计数最大值为 2000，如果定义了工作在条件二（CASE_B）的情况下，该计数最大值为 2500，则该代码可以按照如下方式编写。

```
`define CASE_A
//`define CASE_B

`ifdef CASE_A
    `define CNT_MAX 2000
`elsif CASE_B
    `define CNT_MAX 2500
`endif
```

实际使用时，只需要在代码中定义工作在条件一还是条件二，就能选择让 CNT_MAX 为对应的值。在 Verilog 代码中，只需要直接使用 CNT_MAX 这个参数即可，如下所示：

```
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    cnt <= 0;
else if(cnt == `CNT_MAX)
    cnt <= 0;
else
    cnt <= cnt + 1'b1;
```

24.3基于条件编译的 VGA 时序参数

同样的，我们只需要定义各个分辨率的标志作为条件编译中的条件，并在每个条件下编写该分辨率的各个时序参数的值，那么使用时，只要让代码中某一个分辨率的标志生效，就能让该条件下的所有时序参数值生效，实现“一键”修改分辨率了。以下为实现该参数内容定义的代码：

```
//以下 7 行预定义根据实际使用的分辨率，选择一个使能，另外 6 个使用注释的方式屏蔽

//`define Resolution_480x272 1  //时钟为 9MHz
```

```
`define Resolution_640x480 1    //时钟为 25.175MHz
//`define Resolution_800x480 1  //时钟为 33MHz
//`define Resolution_800x600 1  //时钟为 40MHz
//`define Resolution_1024x768 1 //时钟为 65MHz
//`define Resolution_1280x720 1 //时钟为 74.25MHz
//`define Resolution_1920x1080 1    //时钟为 148.5MHz

//定义不同分辨率的时序参数
`ifdef Resolution_480x272
    `define H_Total_Time  12'd525
    `define H_Right_Border  12'd0
    `define H_Front_Porch  12'd2
    `define H_Sync_Time  12'd41
    `define H_Back_Porch  12'd2
    `define H_Left_Border  12'd0

    `define V_Total_Time  12'd286
    `define V_Bottom_Border  12'd0
    `define V_Front_Porch  12'd2
    `define V_Sync_Time  12'd10
    `define V_Back_Porch  12'd2
    `define V_Top_Border  12'd0

`elsif Resolution_640x480
    `define H_Total_Time  12'd800
    `define H_Right_Border  12'd8
    `define H_Front_Porch  12'd8
    `define H_Sync_Time  12'd96
    `define H_Back_Porch  12'd40
    `define H_Left_Border  12'd8

    `define V_Total_Time  12'd525
    `define V_Bottom_Border  12'd8
    `define V_Front_Porch  12'd2
    `define V_Sync_Time  12'd2
    `define V_Back_Porch  12'd25
    `define V_Top_Border  12'd8

`elsif Resolution_800x480
    `define H_Total_Time 12'd1056
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd40
    `define H_Sync_Time 12'd128
    `define H_Back_Porch 12'd88
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd525
```

```
    `define V_Bottom_Border 12'd8
    `define V_Front_Porch 12'd2
    `define V_Sync_Time 12'd2
    `define V_Back_Porch 12'd25
    `define V_Top_Border 12'd8

`elsif Resolution_800x600
    `define H_Total_Time 12'd1056
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd40
    `define H_Sync_Time 12'd128
    `define H_Back_Porch 12'd88
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd628
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd1
    `define V_Sync_Time 12'd4
    `define V_Back_Porch 12'd23
    `define V_Top_Border 12'd0

`elsif Resolution_1024x768
    `define H_Total_Time 12'd1344
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd24
    `define H_Sync_Time 12'd136
    `define H_Back_Porch 12'd160
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd806
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd3
    `define V_Sync_Time 12'd6
    `define V_Back_Porch 12'd29
    `define V_Top_Border 12'd0

`elsif Resolution_1280x720
    `define H_Total_Time 12'd1650
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd110
    `define H_Sync_Time 12'd40
    `define H_Back_Porch 12'd220
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd750
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd5
```

```
    `define V_Sync_Time 12'd5
    `define V_Back_Porch 12'd20
    `define V_Top_Border 12'd0

`elsif Resolution_1920x1080
    `define H_Total_Time 12'd2200
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd88
    `define H_Sync_Time 12'd44
    `define H_Back_Porch 12'd148
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd1125
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd4
    `define V_Sync_Time 12'd5
    `define V_Back_Porch 12'd36
    `define V_Top_Border 12'd0

`endif
```

在该代码中，定义了 7 种支持的分辨率，分别用 Resolution_480x272、Resolution_640x480、Resolution_800x480、Resolution_800x600、Resolution_1024x768、Resolution_1280x720、Resolution_1920x1080 表示，每个标识后面的数字就是该标识对应的分辨率。使用时，一次在这 7 个标识中选择一个生效，其他 6 个通过注释的方式屏蔽掉，就能唯一确定一个分辨率。然后在后续代码中，在根据是否定义了这些标识来分别定义时序参数中每个物理量的具体数值。

24.4基于条件编译的多分辨率支持 VGA 控制器设计

通过这种方式，在 VGA 控制器的代码中实际使用时，只需要再定义 VGA_HS_end、hdat_begin、hdat_end、hpixel_end、VGA_VS_end、vdat_begin、vdat_end、vline_end 与这些时序参数的关系，就能实现对不同分辨率的支持了。完整的控制器代码如下所示：

```
`include "disp_parameter_cfg.v"

module disp_driver(
    ClkDisp,
    Rst_n,

    Data,
    DataReq,
```

```
    H_Addr,
    V_Addr,

    Disp_HS,
    Disp_VS,
    Disp_Red,
    Disp_Green,
    Disp_Blue,
    Disp_DE,
    Disp_PCLK
);

    input ClkDisp;
    input Rst_n;
    input [`Red_Bits + `Green_Bits + `Blue_Bits - 1:0] Data;
    output DataReq;

    output [11:0] H_Addr;
    output [11:0] V_Addr;

    output reg Disp_HS;
    output reg Disp_VS;

    output reg [`Red_Bits - 1 :0]Disp_Red;
    output reg [`Green_Bits - 1 :0]Disp_Green;
    output reg [`Blue_Bits - 1 :0]Disp_Blue;

    output reg Disp_DE;
    output Disp_PCLK;

    wire hcount_ov;
    wire vcount_ov;

    //----------------内部寄存器定义----------------
    reg [11:0] hcount_r;     //行扫描计数器
    reg [11:0] vcount_r;     //场扫描计数器

    assign Disp_PCLK = ~ClkDisp;
    assign DataReq = Disp_DE;

    parameter hdat_begin = `H_Sync_Time + `H_Back_Porch +
`H_Left_Border - 1'b1;
    parameter hdat_end = `H_Total_Time - `H_Right_Border -
`H_Front_Porch - 1'b1;

    parameter vdat_begin    = `V_Sync_Time + `V_Back_Porch +
`V_Top_Border - 1'b1;
```

```
    parameter vdat_end = `V_Total_Time - `V_Bottom_Border -
`V_Front_Porch - 1'b1;

    assign H_Addr = Disp_DE?(hcount_r - hdat_begin):12'd0;
    assign V_Addr = Disp_DE?(vcount_r - vdat_begin):12'd0;

    //行扫描
    assign hcount_ov = (hcount_r >= `H_Total_Time - 1);

    always@(posedge ClkDisp or negedge Rst_n)
    if(!Rst_n)
        hcount_r <= 0;
    else if(hcount_ov)
        hcount_r <= 0;
    else
        hcount_r <= hcount_r + 1'b1;

    //场扫描
    assign vcount_ov = (vcount_r >= `V_Total_Time - 1);

    always@(posedge ClkDisp or negedge Rst_n)
    if(!Rst_n)
        vcount_r <= 0;
    else if(hcount_ov) begin
        if(vcount_ov)
            vcount_r <= 0;
        else
            vcount_r <= vcount_r + 1'd1;
    end
    else
        vcount_r <= vcount_r;

    always@(posedge ClkDisp)
        Disp_DE <= ((hcount_r >= hdat_begin)&&(hcount_r <
hdat_end))&&((vcount_r >= vdat_begin)&&(vcount_r < vdat_end));

    always@(posedge ClkDisp)    begin
        Disp_HS <= (hcount_r > `H_Sync_Time - 1);
        Disp_VS <= (vcount_r > `V_Sync_Time - 1);
        {Disp_Red,Disp_Green,Disp_Blue} <= (Disp_DE)?Data:1'd0;
    end

endmodule
```

为了让代码的可读性较强，对于各个分辨率的时序参数值，单独写在了名为“disp_parameter_cfg.v”的文件中，然后在使用到这些参数的文件首行使用`include 来包括该文件即可。

为了让该控制器能适用于不同的场合，不仅仅是 VGA 控制器，还可以用来驱动 RGB TFT 屏、HDMI 转换芯片，所以对信号的命名也做了修改，统一将 VGA 改为了 Disp，以让该控制器的应用场景不再局限于传统的 VGA 显示器。

24.5完整的条件编译配置文件

同时，为了让该控制器支持 RGB565 和 RGB888 两种颜色模式，在条件编译文件中还使用了“MODE_RGB888”和“MODE_RGB565”两个条件来指定 RED、GREEN、BLUE 颜色的数据位宽。完整的参数化文件内容如下所示：

```
/*
使用时根据实际工作需求选择几个预定义参数就可以

MODE_RGB888 和 MODE_RGB565 两个参数二选一，用来决定驱动工作在 16 位模式还是
24 位模式
针对小梅哥提供的一系列显示设备，各个设备参数如下所述
    4.3 寸屏：16 位色 RGB565 模式
    5 寸屏：16 位色 RGB565 模式
    GM7123 模块使用 24 位色 RGB888 模式，

Resolution_xxxx 预定义用来决定显示设备分辨率，常见设备分辨率如下所述

4.3 寸 TFT 显示屏：Resolution_480x272
5 寸 TFT 显示屏：Resolution_800x480

VGA 常见分辨率：
    Resolution_640x480
    Resolution_800x600
    Resolution_1024x768
    Resolution_1280x720
    Resolution_1920x1080
*/

//以下两行预定义根据实际使用的模式，选择一个使能，另外一个使用注释的方式屏蔽

`define MODE_RGB888
//`define MODE_RGB565

//以下 7 行预定义根据实际使用的分辨率，选择一个使能，另外 6 个使用注释的方式屏
蔽

//`define Resolution_480x272 1  //时钟为 9MHz
`define Resolution_640x480 1    //时钟为 25.175MHz
//`define Resolution_800x480 1  //时钟为 33MHz
//`define Resolution_800x600 1  //时钟为 40MHz
```

```
//`define Resolution_1024x768 1 //时钟为 65MHz
//`define Resolution_1280x720 1 //时钟为 74.25MHz
//`define Resolution_1920x1080 1    //时钟为 148.5MHz

//定义不同的颜色深度
`ifdef MODE_RGB888
    `define Red_Bits 8
    `define Green_Bits 8
    `define Blue_Bits 8

`elsif MODE_RGB565
    `define Red_Bits 5
    `define Green_Bits 6
    `define Blue_Bits 5
`endif

//定义不同分辨率的时序参数
`ifdef Resolution_480x272
    `define H_Total_Time  12'd525
    `define H_Right_Border  12'd0
    `define H_Front_Porch  12'd2
    `define H_Sync_Time  12'd41
    `define H_Back_Porch  12'd2
    `define H_Left_Border  12'd0

    `define V_Total_Time  12'd286
    `define V_Bottom_Border  12'd0
    `define V_Front_Porch  12'd2
    `define V_Sync_Time  12'd10
    `define V_Back_Porch  12'd2
    `define V_Top_Border  12'd0

`elsif Resolution_640x480
    `define H_Total_Time  12'd800
    `define H_Right_Border  12'd8
    `define H_Front_Porch  12'd8
    `define H_Sync_Time  12'd96
    `define H_Back_Porch  12'd40
    `define H_Left_Border  12'd8

    `define V_Total_Time  12'd525
    `define V_Bottom_Border  12'd8
    `define V_Front_Porch  12'd2
    `define V_Sync_Time  12'd2
    `define V_Back_Porch  12'd25
    `define V_Top_Border  12'd8
```

```
`elsif Resolution_800x480
    `define H_Total_Time 12'd1056
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd40
    `define H_Sync_Time 12'd128
    `define H_Back_Porch 12'd88
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd525
    `define V_Bottom_Border 12'd8
    `define V_Front_Porch 12'd2
    `define V_Sync_Time 12'd2
    `define V_Back_Porch 12'd25
    `define V_Top_Border 12'd8

`elsif Resolution_800x600
    `define H_Total_Time 12'd1056
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd40
    `define H_Sync_Time 12'd128
    `define H_Back_Porch 12'd88
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd628
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd1
    `define V_Sync_Time 12'd4
    `define V_Back_Porch 12'd23
    `define V_Top_Border 12'd0

`elsif Resolution_1024x768
    `define H_Total_Time 12'd1344
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd24
    `define H_Sync_Time 12'd136
    `define H_Back_Porch 12'd160
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd806
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd3
    `define V_Sync_Time 12'd6
    `define V_Back_Porch 12'd29
    `define V_Top_Border 12'd0

`elsif Resolution_1280x720
    `define H_Total_Time 12'd1650
```

```
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd110
    `define H_Sync_Time 12'd40
    `define H_Back_Porch 12'd220
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd750
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd5
    `define V_Sync_Time 12'd5
    `define V_Back_Porch 12'd20
    `define V_Top_Border 12'd0

`elsif Resolution_1920x1080
    `define H_Total_Time 12'd2200
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd88
    `define H_Sync_Time 12'd44
    `define H_Back_Porch 12'd148
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd1125
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd4
    `define V_Sync_Time 12'd5
    `define V_Back_Porch 12'd36
    `define V_Top_Border 12'd0

`endif
```

经过上述分析，使用多分辨率适配型控制器，即可较好的解决不同分辨率条件下各显示器的像素问题。使用 VGA 章节学到的原理，也可以较好的迁移到后续 TFT 和 HDMI 章节中。

25 TFT 显示屏驱动设计与验证

工程源码	|----02_设计实例 |---- ch25_tft_ctrl
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本章节将介绍 TFT 显示屏的驱动相关理论知识，并基于介绍的内容，完成 TFT 显示屏的彩条显示实验。

25.1TFT 显示屏的应用背景

在电子系统中，经常会使用到相关设备来显示相关信息。从数码管、LCD1602、LCD12864、小尺寸 TFT 彩屏，到现在的高达 4K 分辨率显示屏，显示技术经历了巨大的变革。从单色到真彩，从低分辨率到高清分辨率。不同的显示设备根据应用领域不同，有着不同的屏幕接口，如 8080 接口、RGB 接口、MIPI 接口等。每种显示原理以及屏幕接口都有其各自的优缺点，在不同场合使用时可根据实际需求合理选择合适的显示设备。

相信大部分同学开始学习单片机时，接触到的就是 51 单片机。在 51 单片机上，使用最广泛的显示设备就是 1602、12864 字符点阵液晶屏。这两种液晶显示设备均支持 Intel 8080 的总线，51 单片机使用外扩 8080 总线或者 IO 模拟该总线对液晶屏进行读写，以实现数据显示。

后来，当学习了较为高级的单片机，如 MSP430、STM32 后，开始接触到了彩色液晶显示屏。单片机经常使用的彩色液晶显示屏通常都是内部带有显示缓存存储器，这样只要把待显示的内容写入到显存中，显示屏就会自动的依次读取每个像素点对应的显存数据并驱动液晶像素点显示对应颜色。彩色液晶显示屏模型和 12864 液晶屏较为类似，只不过能够显示彩色图像，而 12864 只能显示黑白内容。此类彩色液晶屏驱动方式之一为使用 STM32 的 FSMC 总线，将液晶显示屏映射为一块存储区，直接使用 FSMC 总线对该存储区进行读写即可。

随着学习的进一步深入，部分同学接触到了嵌入式 ARM 处理器，如 ARM9、ARM11、Cortex-A8 等等，这些处理器往往运行成熟的操作系统，如 WindowsCE、Linux 等等。而且它们都能驱动很多的大屏幕，如市面上最早流行的 2440 的开发板，基本标配都有一块 4.3 寸 480*272 分辨率的 RGB（Red、

Green、Blue）显示屏，一些配置较高的板甚至支持到了 7 寸 800*480 分辨率。这些屏幕，使用普通的 MCU 是无法驱动的，因为这些屏幕本身不含显存，需要驱动器能够带有显存，并按照 RGB 时序准确的将显存中的数据送到屏幕上显示。一般的单片机由于工作速度、总线带宽、存储容量有限，很难支持如此高的数据刷新速率。而 ARM 处理器由于性能较高，而且使用外部大容量 DDR 存储器作为运行内存，内建专门针对实时图像显示的 framebuffer 硬件逻辑，因此能够支持这样一类的屏幕。

MCU 接口的液晶屏，多使用 8080 接口，每个屏幕内部都使用了一个独立的显示控制器，如大家熟知的 ILI9320、ILI9341 等，这些控制器都包含了大量的寄存器，要控制这些屏幕工作在正确的模式，需要先由主控对这些寄存器写入初始值。例如，ILI9341 的寄存器数量超过 100 个，使用时需要对其中近 100 个进行初始化写入操作。对于 fpga 来说，实现复杂的时序编程效率和灵活性都非常低，所以不是很适合驱动这类屏幕。

而 RGB 接口的 TFT 屏幕，虽然其对数据的实时性要求很高，但是接口时序却非常简单，与 VGA 显示器时序呈 100%兼容的特性，几乎不需要任何初始化操作就能开始显示，非常适合 FPGA 这种头脑简单，四肢发达的器件进行驱动。

鉴于 TFT 屏显示系统在数字系统中应用实在是非常广泛，掌握 TFT 屏幕的驱动和应用，是非常有实用意义的，所以本节将学习如何使用 FPGA 来进行 RGB 接口的 TFT 屏幕驱动设计。

25.2TFT 触摸显示屏模块介绍

芯路恒 TFT5.0’’_V3.0 / TFT4.3’’_V3.0 显示屏模块是武汉芯路恒科技有限公司，小梅哥 FPGA 团队专为广大 FPGA 学习和应用群体开发的一款分辨率为 800*480，支持 5 点触控的通用型电容触摸屏模块。该显示屏不仅能够用于本公司生产销售的各个型号的 FPGA 开发板，而且还可以直接用于众多的 Linux 开发板。同时，由于模块提供了有 2.54mm 间距的排针连接端子，对于部分有 DIY 需求的客户，也可以使用优质杜邦线将显示屏连接到各种硬件板卡上。

图 25-1 TFT 屏显示效果

该显示屏共分为 2 个版本，4.3 寸版本的 TFT4.3’’_V3.0 和 5.0 寸版本的 TFT5.0’’_V3.0。两者 PCB 背板电路完全相同，接口脚位定义完全相同，接口时序完全相同，仅使用的显示屏模组尺寸不同。设计两个尺寸的主要目的是适配不同的开发板使用，以获得较好的物理结构兼容性。下图 25-2 为 4.3 寸电容触摸显示屏背板，下图 25-3 为 5.0 寸电容触摸显示屏背板，在背板的右侧，使用文字标注了版本型号。

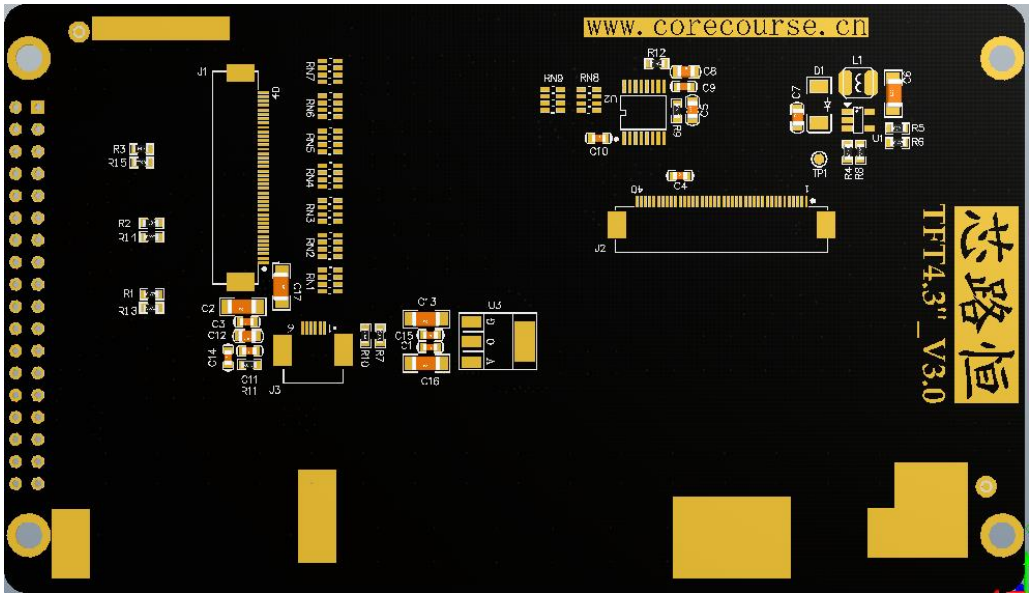


图 25-2 4.3 寸电容触摸显示屏 TFT4.3’’_V3.0 背板

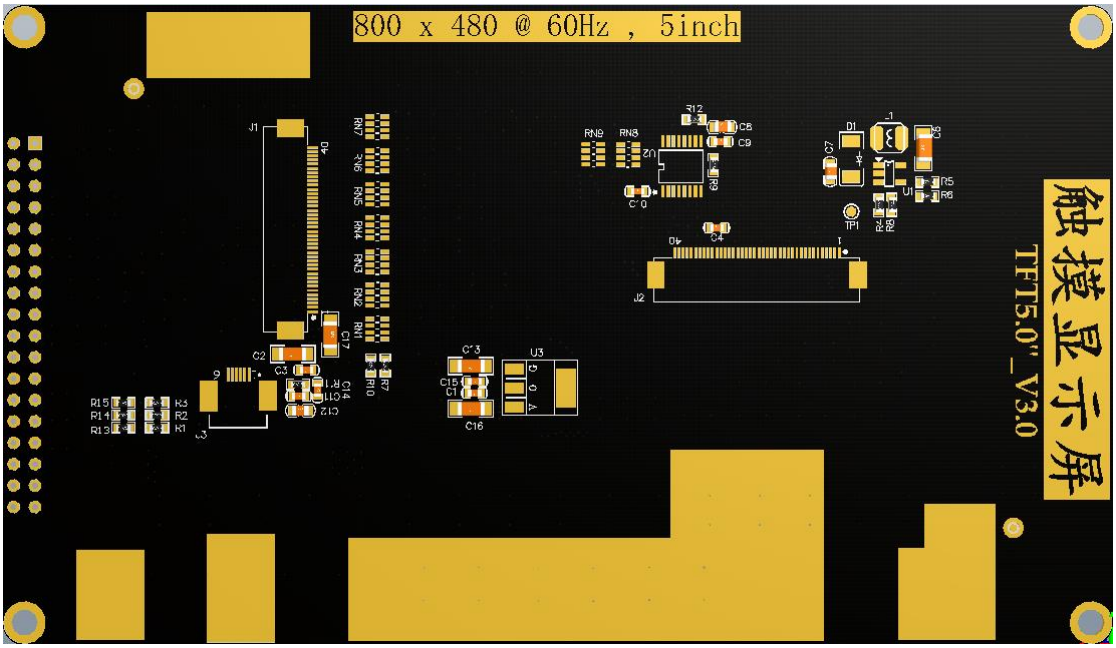


图 25-3 5.0 寸电容触摸显示屏 TFT5.0’’_V3.0 背板

在介绍该模块功能时，所有内容对 4.3 寸和 5 寸屏版本均完全适配。以下不再区分尺寸说明，全部以 TFT_V3.0 为名进行讲解。

显示屏由我们设计的专用的 5 寸或 4.3 寸 PCB 背板和显示屏模组厂家提供的显示屏模组经贴合而成。总结就是：芯路恒 TFT5.0’’_V3.0 显示屏模块 = 通用 TFT 显示模组+芯路恒定制的屏幕 PCB 背板。如下图 25-4 所示：

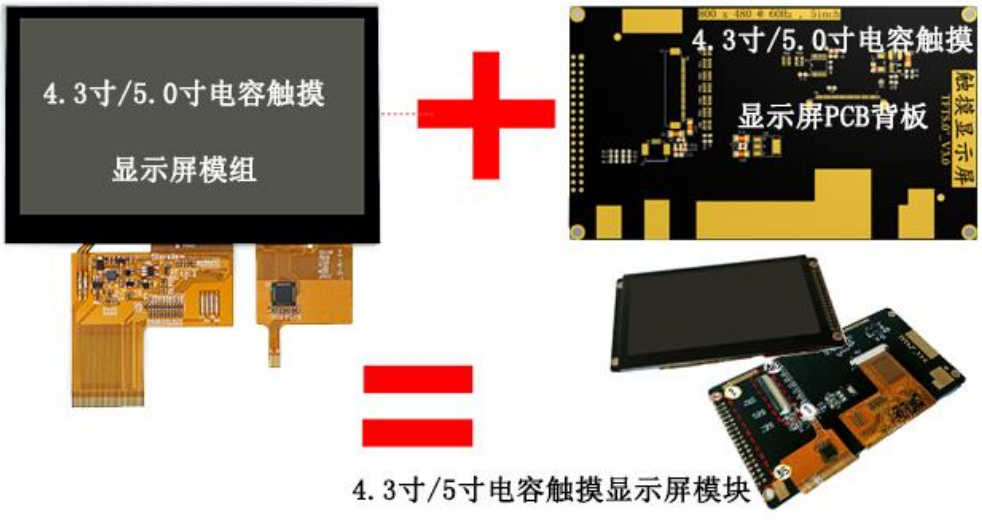


图 25-4 电容触摸显示屏模块

25.2.1屏幕 PCB 背板设计

4.3 寸和 5 寸显示屏 PCB 背板电路设计相同。设计时，PCB 背板兼顾考虑了

对支持电阻触摸屏和电容触摸屏的支持，在该背板上集成了电阻触摸屏转换器 XPT2046 和 19.2V 的背光升压电路。以提供显示屏组件正常工作的必备条件。以下介绍 PCB 背板上的各个功能电路：

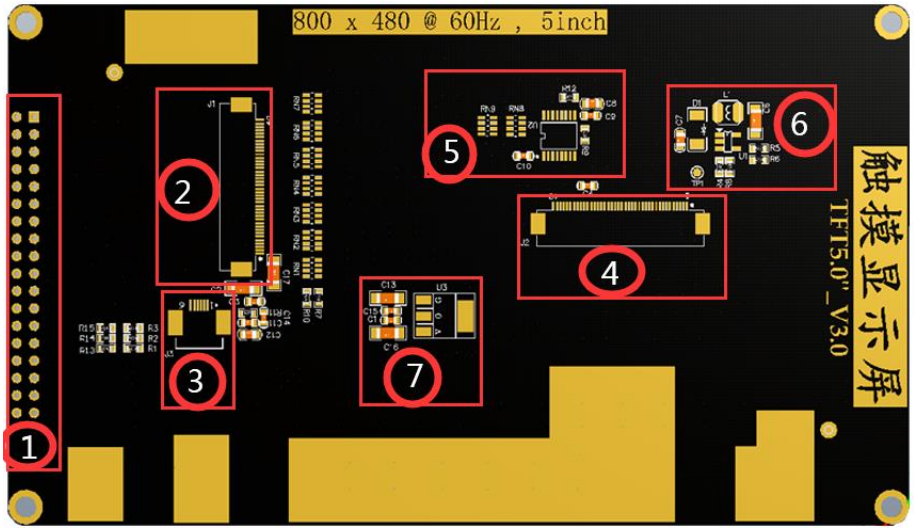


图 25-5 显示屏模组背面功能分区标号

表 25-1 显示屏模组背面功能分区介绍

功能单元	序号	功能说明	备注
用户接口	1	2*18 排针，接开发板显示扩展接口	两个接口二选一即可
	2	FPC 接口，可用 FPC 排线连接主板	
模组接口	3	电容触摸屏连接器，接入电容触摸屏信号线	
	4	显示屏连接器，接入显示屏信号线	
功能电路	5	电阻触摸控制器电路	基于 XPT2046 芯片
	6	背光升压电路	5V 升 19.2V
	7	电源降压电路	5V 降压到 3.3V

25.2.1.1屏用 2*18 排针接口

该接口主要用来连接小梅哥各个开发板上的通用显示扩展接口，支持但不限于 AC620、AC6102、AC501、AC609、Starter、ACX735、ACX720、ACZ702 等开发板，高云开发板支持 2*18 排针接口，本章实验就使用排针接口。

显示扩展接口一般位于开发板右侧部位，为 36pin 排母连接器。插接时，显示屏模块覆盖在开发板上方，如下图 25-6 所示：

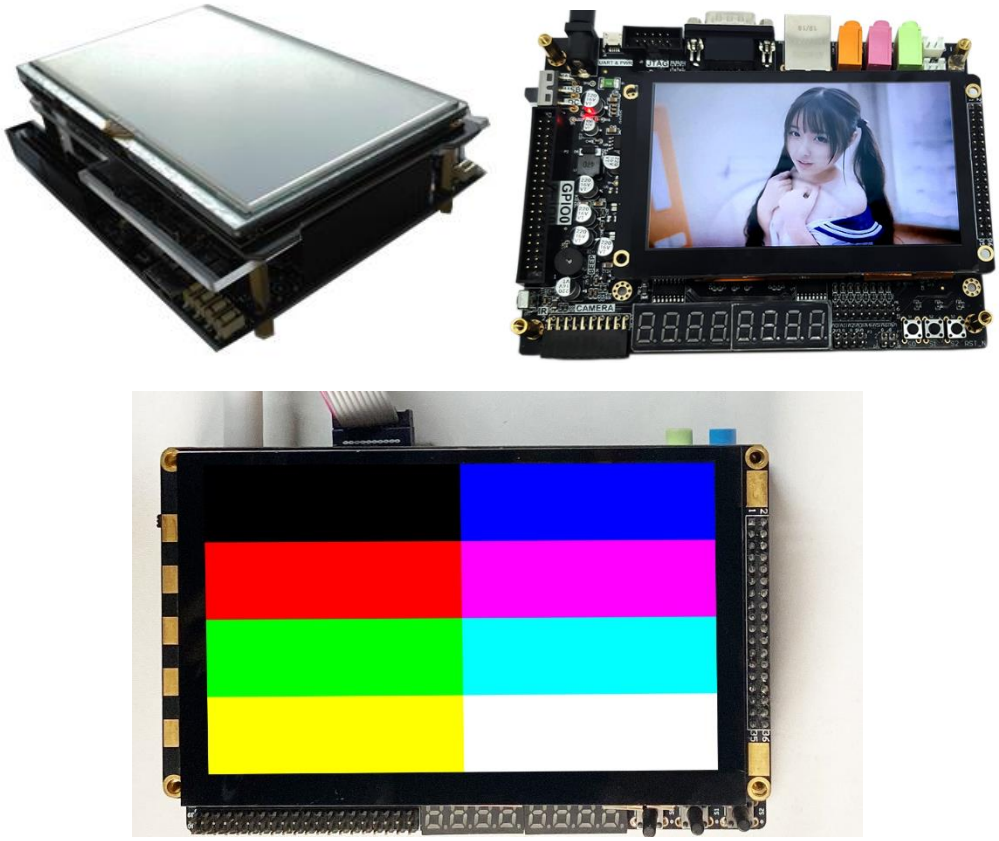


图 25-6 AC6102、AC620 和高云开发板 2*18P 排针接线示意图

显示屏模块提供两个用户接口，一个为 2*18 的 36 针排针接口，使用该接口可以直接插接到小梅哥 ACX720、AC620、AC609、AC6102、高云等 FPGA 开发板以及部分 FPGA 核心板上。LCD 显示频对接 2*18 排针的接口信号顺序如下图 25-7 所示：

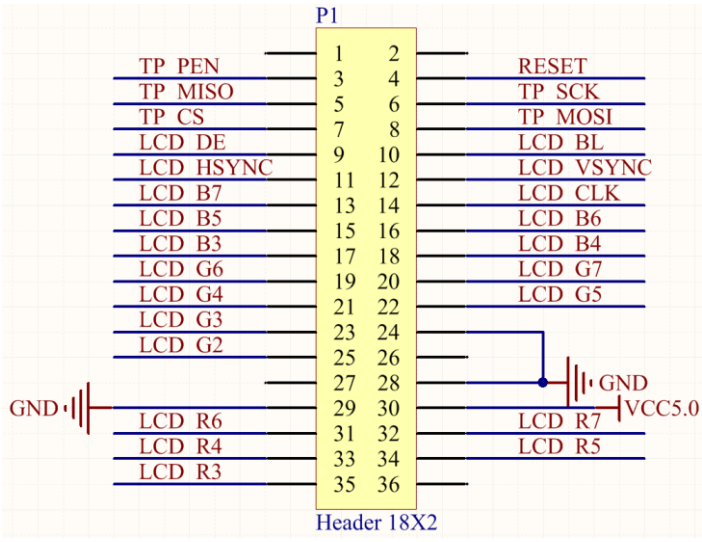


图 25-7 2*18 排针接口顺序图

这些信号中，以 TP 开头的信号为触摸板的数字接口（基于 SPI 或 I2C 协议），以 LCD 开头的代表显示屏的数字接口。RESET 信号实际没有任何作用，LCD_BL 为背光控制脚，为高电平时点亮背光。

以下表 25-2 为该显示屏的所有信号功能介绍：

表 25-2 信号的分类和功能

信号分类	信号名称	信号功能
电阻触摸控制器接口 使用时根据具体的模块使用的触摸类型，与电容屏二选一	TP_PEN=> RT_PEN	触摸屏笔触中断信号，当检测到触摸屏被按下时，可以变低，从而向外产生触摸按下通知信号，作为中断信号
	TP_MISO	SPI 数字接口的从机数据输入接口，由主机（MCU、FPGA）向触摸控制芯片 XPT2046 传输控制命令。
	TP_MOSI=> RT_MOSI	SPI 数字接口的从机数据输出接口，由触摸控制芯片 XTP2046 向主机（MCU、FPGA）传输采样到的坐标点。
	TP_CS=> RT_CS	SPI 接口的片选信号
	TP_SCK=> RT_SCLK	SPI 接口的串行时钟接口
电容触摸控制接口 使用时根据具体的模块使用的触摸类型，与电阻屏二选一	TP_PEN=> CT_INT	电容触摸屏的中断信号
	TP_MOSI=> CT_SDA	电容触摸屏的 I2C 接口的数据线
	TP_SCK=> CT_SCL	电容触摸屏的 I2C 接口的时钟线
	TP_CS=> CT_RST	电容触摸屏的复位控制信号
TFT 显示接口	LCD_DE	LCD 驱动时序的数据使能信号，即在显示有效区域，打开该信号（高电平）以使能信号输入，在非有效区域，关闭该信号以禁止像素数据输入，以免影响到消隐。
	LCD_HSYNC	LCD 驱动的行同步信号
	LCD_VSYNC	LCD 驱动的场同步信号
	LCD_R[7:0]	LCD 颜色数据的红色分量部分，在 2*18 的排针接口上仅用了 LCD_R[7:3]共 5 位，在 40 FPC 排线接口上全部使用
	LCD_G[7:0]	LCD 颜色数据的绿色分量部分，在 2*18 的排针接口上仅用了 LCD_G[7:2]共 6 位，在 40 FPC 排线接口上全部使用
	LCD_B[7:0]	LCD 颜色数据的蓝色分量部分, 在 2*18 的排针接口上仅用了 LCD_B[7:3]共 5 位，在 40 FPC 排线接口上全部使用
其他控制信号	LCD_BL	LCD 背光控制信号，高电平点亮背光，可以使用 PWM 控制
	RESET	虽名为 RESET 信号，实际未做任何功能，因为 TFT 屏无无非接口，也无需复位，此脚可以不关心。

需要说明的是，无论是 4.3 寸还是 5 寸的显示屏模组，都是支持 RGB888 的 24 位色模式，但是在实际使用时，为了节省存储器、节省 IO 用量，或是为了提升存储器可用带宽，往往会采用 RGB565 的模式来进行显示。既将 LCD_R[7:3]、

LCD_G[7:2]、LCD_B[7:3]取出，用来传递图像数据，而将 LCD_R[2:0]、LCD_G[1:0]、LCD_B[2:0]直接接地或者接高电平。这样就能够使用 16 位的数据来驱动 24 位色的显示屏且保证颜色基本不失真了。

该显示屏除了支持 2*18 的排针接口，还支持通过 40P FPC 接口进行连接，高云板子上直接使用排针接口即可。

25.2.1.2屏用背光升压电路

除基于 LED 技术的显示屏外，其他投射式显示屏模组的每个像素点都是不能主动发光的。例如 TFT 显示屏，使用的是液晶材质在不同的电压下其对光线的透过性不同的特性，在其背面施加以 LED 白光源。不同透光性的液晶单元，对白光的透过性不同，从而进入人眼的光线强度也就不同，使得人眼能够看到不同的亮度，再通过对每个液晶单元加盖不同颜色的滤光片，仅允许红、绿、蓝三种基本光纤中的一种透过液晶单元，从而最终实现彩色显示的效果。

所以，TFT 显示屏要正常显示颜色，背光光源也是必不可少的。常见的 TFT 显示屏模组，都集成了能够发出白光的 LED 灯，这些灯通过串联加并联的方式连接在一起构成了 LED 光带。正常工作时，需要对这些 LED 组成的灯带提供高达 19.2V 的电压，才能让其达到合适的亮度。

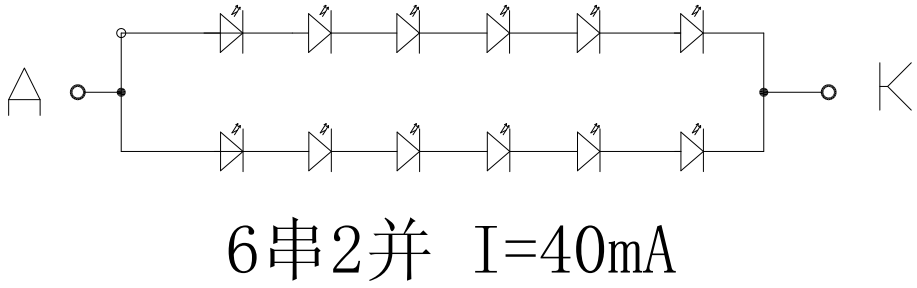


图 25-8 屏用背光升压电路原理图

由于 TFT 屏背光灯需要高达 19.2V 的供电电压，因此本显示屏模块在 PCB 背板上设计了一个从 5V 升压到 19.2V 的背光升压电路。该电路带一个控制信号，名为 LCD_BL，当该信号为高电平时，背光被点亮，否则背光熄灭。如果需要调整 TFT 背光的亮度，可以通过给该信号连接 1KHz 左右的 PWM 波，通过调整 PWM 波的占空比来调整背光的亮度。

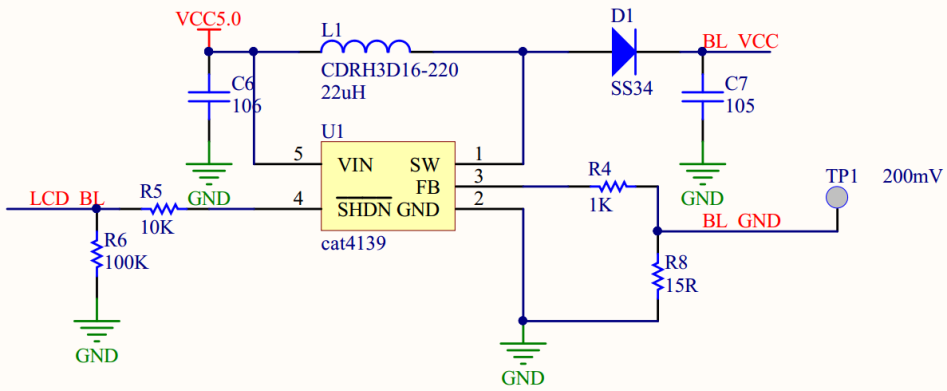


图 25-9 屏用背光升压电路原理图

25.2.1.3电阻屏触摸控制器

电阻触摸控制屏以其灵敏的触控能力和强大的抗干扰能力，常用于工业设备中。因此本屏幕背板上设计了一个触摸控制器，用来控制和处理电阻触摸屏。因为电阻触摸屏本质是电阻，其只能通过改变自身的电阻值来改变流经自生的电流大小，在两侧施加一定电压的情况下，其产生的是模拟信号。而 FPGA 只能传输和处理数字信号，所以需要使用一个能够实现对电阻触摸屏施加电压并将其阻值大小转化为数字信号的触摸控制器。本 PCB 背板上触摸控制器使用 XPT2046，该控制器能够读取电阻触摸屏的坐标值并通过 SPI 接口传递给控制器（如单片机、FPGA），实现触摸读取功能。

25.2.1.4屏幕模组连接口

正常市面销售的 TFT 模组仅仅是将背光 LED 灯、显示屏液晶和触摸屏组装到一起的模组，如下图 25-10 所示。

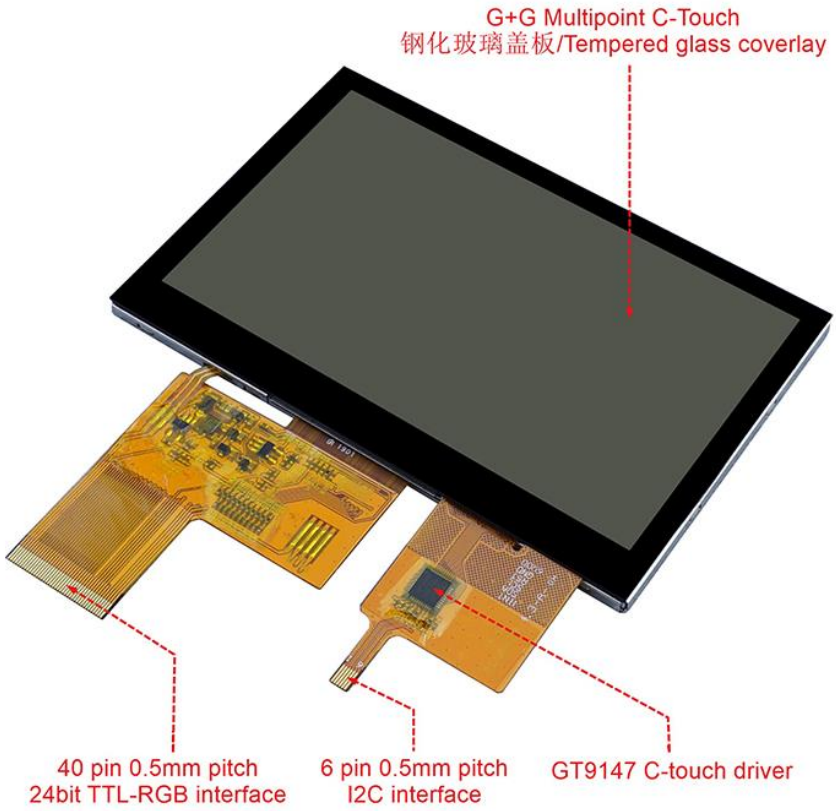


图 25-10 模组接口功能图

这些模组一般以 40pin 的 FPC 排线连接点提供对外连接。TFT 模组本身是无法直接正常工作的，需要给其提供工作电压和背光驱动电压。所以在本 PCB 背板上使用 1 个 40pin 的 FPC 母座来连接该 TFT 模组。该接口将 TFT 屏线接口上的所有信号引入到 PCB 板，然后再分别接到与开发板连接的排针接口、FPC 连接口、触摸控制器以及背光供电。

25.2.1.5电容触摸屏连接口

相较于电阻触摸屏，电容触摸屏的触摸坐标定位实现更加的复杂，一般都需要使用专用的电容触摸控制器来完成多点触摸信号的感应。所以大部分电容触摸模组都集成好了该电容触摸控制器，对外提供标准的 I2C 总线接口，使用时，只需要主机通过 I2C 总线读取该触摸控制器芯片中存储的实时坐标值即可。

也因如此，在电容触摸屏模组对外的接口上，实现反而比电阻触摸屏简单，由于 I2C 控制器直接传输的就是数字信号，因此不需要再使用像 XPT2046 这种模数转换器来转换坐标位置，所以在 PCB 背板上，只需要使用一个简单的连接座将触摸屏模组的 I2C 接口映射到与主机控制器连接的排针或 FPC 排座上即可。

虽然本屏幕模块的 PCB 背板上设计了支持电容触摸屏和电容触摸屏，但是

两者是不能，也不会同时使用的。为了降低对控制器的占用，电阻触摸屏和电容触摸屏通过跳线的方式选择具体连接哪种接口的触摸屏到控制器。

在 PCB 背板上，使用两个 0R 的排阻作为跳线，选择是将电阻触摸屏控制器的 SPI 接口的控制信号还是电容触摸屏的 I2C 接口的控制信号连接到用户接口的对应脚上。如下图 25-11 所示：

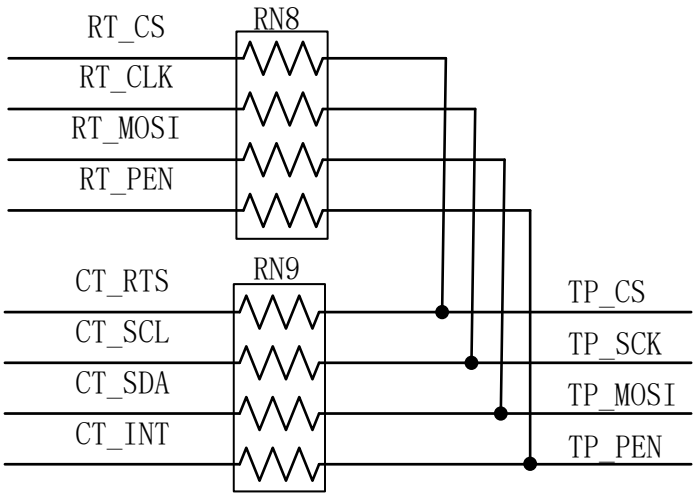


图 25-11 模组控制信号选择

图中，TP_XXX 信号连接到了 2*18 排针和 40P FPC 连接器的用户接口上，具体该引脚连接到 PCB 背板上的哪种触摸屏接口，根据 RN8 和 RN9 是否焊接来决定，如果 RN8 焊接，而 RN9 不焊接，则将电阻触摸屏的控制信号接到用户接口，进而连接到处理器上，如果 RN9 焊接，而 RN8 不焊接，则将电容触摸屏的控制信号接到用户接口，进而连接到处理器上。

25.2.1.6通用 TFT 显示模组

通用显示屏模组采用久经市场检验的 4.3 寸或 5 寸显示屏模组。这两种模组功能相同，接口相同，时序参数也都完全相同，仅在 FPC 排线的物理位置上有所区别，使用时驱动和程序可以完全互通。

对于 4.3 寸和 5 寸显示模组来说，无论是 480*272 还是 800*480 分辨率，其接口信号定义和功能都是一样的。例如对于一个经典的 4.3 寸触摸显示屏，其接口定义如下所示：

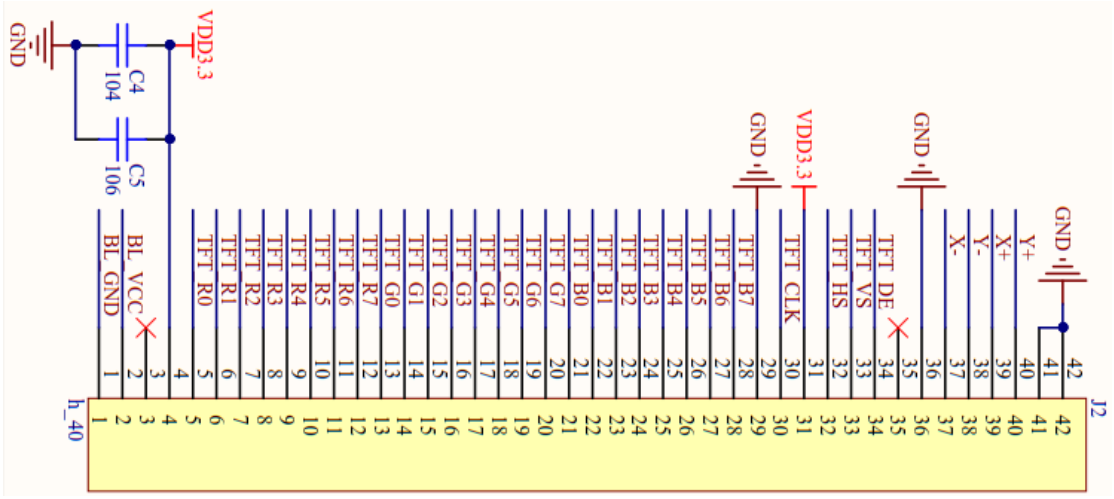


图 25-12 模组接口

表 25-3 AT043TN24 模组 IO 说明

引脚编号	引脚名	引脚功能说明
1	VLED-	LED 背光灯供电阴极
2	VLED+	LED 背光灯供电阳极
3	GND	TFT 工作电源地
4	VDD	TFT 工作电源 VCC
5~12	R0~R7	红色分量数据 0~7 位
13~20	G0~G7	绿色分量数据 0~7 位
21~28	B0~B7	蓝色分量数据 0~7 位
29	GND	TFT 工作电源地
30	PCLK	TFT 像素时钟，驱动器与 TFT 屏的数据和控制信号全部需要同步于该像素时钟信号。
31	DISP	显示开/关
32	HSYNC	行同步信号
33	VSYNC	场同步信号
34	DE	数据使能信号，即在显示有效区域，打开该信号以使能信号输入，在非有效区域，关闭该信号以禁止像素数据输入，以免影响到消隐。
35	NC	悬空
36	GND	TFT 工作电源地
37	X1	差分模拟触摸接口右侧电极
38	Y1	差分模拟触摸接口底侧电极
39	X2	差分模拟触摸接口左侧电极
40	Y2	差分模拟触摸接口上方电极

25.2.1.7关于 PPI

说一句题外话，经常有网友问我们为什么只提供 4.3 寸、5 寸的屏幕，没有提供 7 寸的。因为他们觉得屏幕尺寸越大，效果就越清晰。事实上这种看法是

不对的，最起码是不准确的。一个屏幕清晰度如何，既不是单单看物理像素数量，也不是看物理尺寸大小，而是看屏幕像素密度，也就是 PPI（每英寸屏幕所拥有的像素数），PPI 的计算公式为：

$$\mathrm{PPI} = \frac{\sqrt{横向\ Pixel^2 + 纵向\ Pixel^2}}{屏幕尺寸\ inch}$$

由此可知，相同物理像素数量的情况下，屏幕尺寸越小，像素密度越高，显示效果就越细腻。所以，同样都是 800*480 分辨率的显示屏，5 寸屏的 PPI 肯定比 7 寸的高，显示效果更加细腻，4.3 寸屏的 PPI 又比 5 寸屏高，显示效果更加细腻。

常见的 7 寸分辨率有 800*480 和 1024*600 两种。因此，在对显示屏物理尺寸没有特别要求的情况下，如果仅需要 800*480 的分辨率，使用 5 寸或 4.3 寸的屏幕，不仅显示效果更加细腻，而且从便携性，成本各方面都具有更大的优势。

25.2.2 RGB 接口的颜色模式兼容性

在前面介绍中提到，该屏幕的颜色数据支持 24 位输入，即每种颜色（RGB）有 8 位表示。但是在很多对颜色效果要求不高的系统中，为了节约存储器带宽和控制器的引脚数量，会使用 16 位色（RGB565）进行图像显示。下表 25-4 为 RGB565 模式下三种颜色对应的数据位。

表 25-4 RGB565 数据线格式

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D1
R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

为了让 RGB 接口的 TFT 屏适配 RGB565 颜色模式，需要对其提供的 24 位数据线进行取舍。模组 24 位输入数据线和 RGB565 颜色模式下 16 位数据的对应关系如下表所示。

表 25-5 显示数据对应表

TFT24 位数据线	R[7:3]	R[2:0]	G[7:2]	G[1:0]	B[7:3]	B[2:0]
模组 16 位数据线	TFT_RGB [15:11]	NC	TFT_RGB [10:5]	NC	TFT_RGB [4:0]	NC

RGB565 颜色模式理论上可以显示 2^16=65536 种颜色。根据三基色原理，可以总结出常见的几种颜色对应的数据编码。下表 25-6 位总结的几种基本色的对应 RGB565 模式下的数据编码值。

表 25-6 颜色数据编码

颜色	黑	蓝	红	紫	绿	青	黄	白

R	0	0	31	31	0	0	31	31
G	0	0	0	0	63	63	63	63
B	0	31	0	31	0	31	0	31
数据编码	0x0000	0x001F	0xF800	0xF81F	0x07E0	0x07FF	0xFFE0	0xFFFF

25.2.3RGB TFT 屏时序参数

RGB 接口的 TFT 屏扫描方式和 VGA (Video Graphics Array)标准兼容，也是使用行列扫描的方式。本节，学习完本显示屏的结构和物理参数值后，就能直接使用多分辨率适配的控制器直接驱动本显示屏。

25.2.3.14.3 寸普清分辨率显示屏时序参数

4.3 寸普清分辨率显示屏采用 480*272 分辨率，其时序参数可以从显示模组厂家提供的规格书中查到。

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Clock cycle	1/tc	5	9.00	12	MHz	
Hsync cycle	1/fH	-	17.14	-	KHz	
Vsync cycle	1/fv	59.94	-	-	Hz	
Horizontal signal	th	-	525	-	CLK	Note 1
Horizontal display period	thd	-	480	-	CLK	
Horizontal Front porch	thf	2	-	-	CLK	Note 2
Horizontal Pulse width	thp	2	41	-	CLK	Note 2
Horizontal Back porch	thb	2	-	-	CLK	Note 2
Vertical cycle	tv	-	286	-	H	
Vertical display period	tvd	-	272	-	H	
Vertical Front porch	tvf	2	2	-	H	
Vertical Pulse width	tvp	2	10	-	H	
Vertical Back porch	tvb	2	2	-	H	
DISP Setup Time	tdiss	10	-	-	ns	
DISP Hold Time	tdish	10	-	-	ns	
Clock Period	PW CLK	66.7	-	-	ns	
Clock Pulse High Period	PWH	26.7	-	-	ns	
Clock Pulse Low Period	PWL	26.7	-	-	ns	
Hsync Setup Time	ths	10	-	-	ns	
Hsync Hold Time	thh	10	-	-	ns	

图 25-13 480*272 普清分辨率屏时序参数

4.3 寸普清显示屏的分辨率为 480*272，对应的 60Hz 刷新率时的像素时钟频率为 9MHz。

25.2.3.24.3 寸/5 寸高清分辨率显示屏时序参数

4.3/5 寸高清分辨率显示屏采用 800*480 分辨率，其时序参数也可以从显示模组厂家提供的规格书中查到。在该规格书的第 9 页详细列出了各个时序参数的值。

4.3/5 寸高清显示屏的分辨率为 800*480，对应的 60Hz 刷新率时的像素时钟频率为 33.3MHz。

下表为 480*272 分辨率和 800*480 分辨率的各个时序参数值。如果用户需要自己设计对应的控制器，可以根据下表 25-7 的参数来编写。

表 25-7 两种高清分辨率屏各时序参数值

	480 * 272	800 * 480
H Right Border	0	0
H Front Porch	2	40
H Sync Time	41	128
H Back Porch	2	88
H Left Border	0	0
H Data Time	480	800
H Total Time	525	1056
V Bottom Border	0	8
V Front Porch	2	2
V Sync Time	10	2
V Back Porch	2	25
V Top Border	0	8
V Data Time	272	480
V Total Time	286	525

由于在上一节多分辨率视频的 VGA 控制器中我们已经包含了这两个分辨率的参数，所以无需再编写修改代码。只需要在配置文件中选择使用 Resolution_480x272 或 Resolution_800x480 即可。

25.3RGB 接口 TFT 控制器板级验证

前面章节，我们简述了 RGB 接口 TFT 控制器的设计思路并给出了具体的 RGB 接口 TFT 控制器设计过程，同时通过仿真验证了设计的合理性。本节，我们将对该 RGB 接口 TFT 控制器使用彩条实验来进行板级验证，通过板级验证来进一步确定我们设计的正确性。

25.3.1板级验证功能设计

完成 TFT 的驱动模块设计后，接下来需思考如何通过板级验证来证实 TFT

显示频驱动模块的正确性。这里，板级验证的设计思想仍然可以和 VGA 控制器章节保持一致，使用彩条实验来验证 TFT 控制器控制器的设计正确性。

关于彩条实验的设计思想，可以参考前面 VGA 控制器文档对应章节设计部分，这里也不再重复讲解。接下来，还需要生成 TFT 工作的像素时钟。

25.3.2添加 PLL 时钟分频单元

本小节讲解如何获得像素时钟。值得注意的是，VGA 的典型工作时钟频率和 TFT 的典型工作时钟频率是不同的。前面章节介绍过，VGA 控制器的典型工作时钟频率为 25MHz，而 TFT 控制器的典型工作时钟频率为 33MHz。

高云开发板上为 50MHz 的晶振，因此需要使用锁相环对时钟进行分频得到 33MHz 的时钟，以供 TFT 控制器使用。具体 PLL 配置主要参数如图所示。

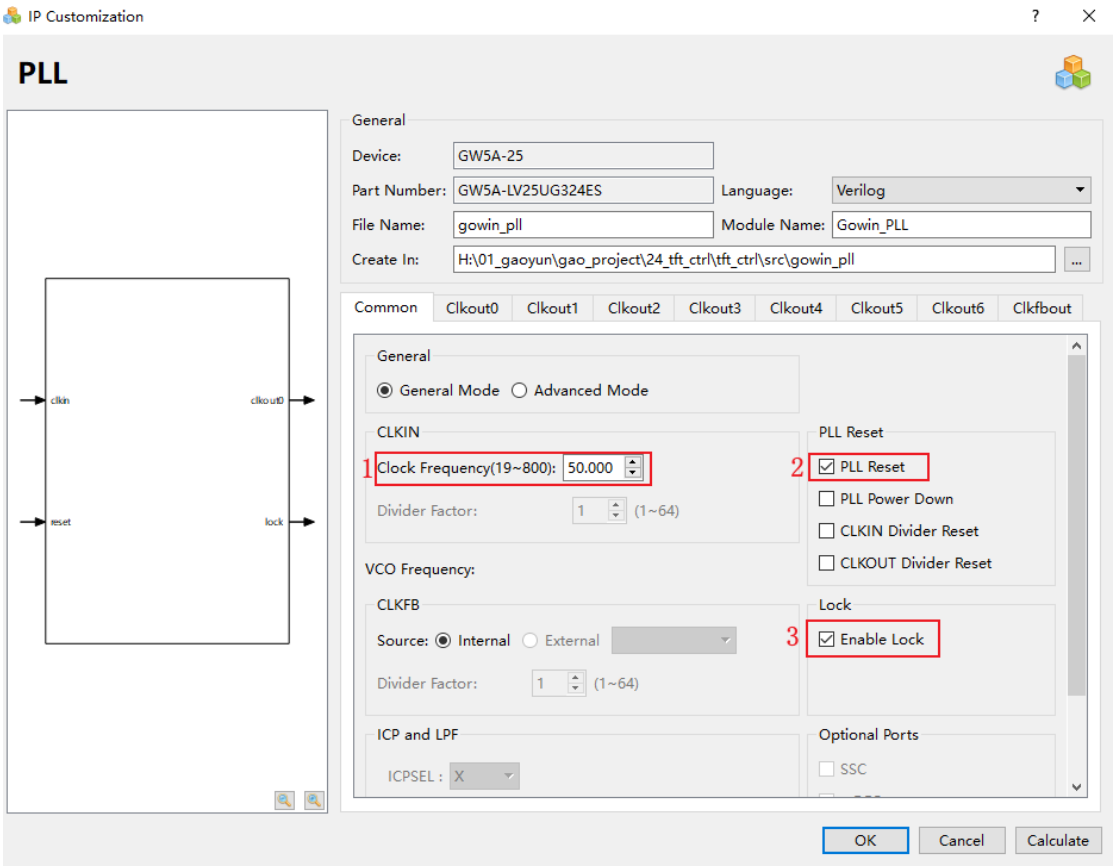


图 25-14 TFT 控制器锁相环配置图 1

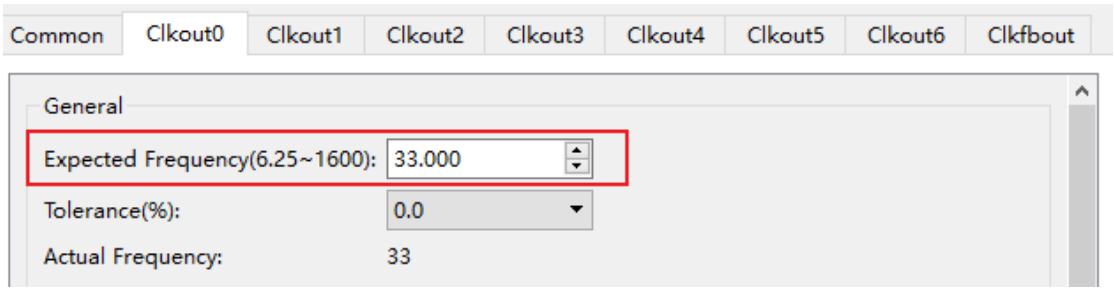


图 25-15 TFT 控制器锁相环配置图 2

输入时钟 50M，输出时钟 33M，由于高云开发板的按键按下是低电平，这里将 PLL 复位设置成低电平有效。PLL 配置好后，再将 PLL 例化到板级测试的顶层文件中。

25.4板级调试与验证

经过上述工作，所有代码都已经设计完毕。现对该 TFT 控制器进行板级验证，通过板级验证来进一步验证设计的正确性。TFT 的板级验证，主要验证以下三个方面：

1. 能否正确的全屏点亮屏幕，显示稳定。
2. 能否正确的显示颜色，即按照需求定制需要显示的颜色。
3. 能否正确的定位坐标，即实现在指定的位置显示对应的数据。

25.4.1完整的彩条实验测试电路代码

由于在 VGA 彩条部分已经对相关代码进行了罗列并附有详细的注释，这里 RGB 接口 TFT 屏的设计原理和 VGA 部分完全相同，对于不同的分辨率，只需选择相应的代码，并按前面多分辨率适配的设计方案执行条件编译即可。基于以上实际情况、这里暂不将 TFT 彩条实验测试电路代码列入篇幅，有需要的同学可以参考配套工程代码的对应内容。

25.4.2系统所需硬件

1. 高云开发板一块
2. DC 电源电缆一根
3. 高云下载器一个
4. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台

5. 5 寸 TFT 显示屏一块

25.4.3液晶屏模组硬件连接

本次系统的硬件连接入如图 25-16 所示。

图 25-16 硬件连接

在插接 5 寸 TFT 屏时，需要注意显示屏的引脚要与开发板上拓展接口一一对应。连接完硬件后，将电源拨码开关拨到对应供电侧，接下来便可以进行板级验证了。

25.4.4下载与验证

明确验证目标并完成硬件连接后，按照前面小节介绍的内容，创建工程，并按照高云开发板的引脚分配表分配正确的引脚。

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	TFT_clk	output		M14	3	False	LVCMOS33
2	TFT_de	output		U18	3	False	LVCMOS33
3	TFT_hs	output		T18	3	False	LVCMOS33
4	TFT_pwm	output		U17	3	False	LVCMOS33
5	TFT_rgb[0]	output		M18	3	False	LVCMOS33
6	TFT_rgb[10]	output		L15	3	False	LVCMOS33
7	TFT_rgb[11]	output		E18	2	False	LVCMOS33
8	TFT_rgb[12]	output		F16	2	False	LVCMOS33
9	TFT_rgb[13]	output		F15	2	False	LVCMOS33
10	TFT_rgb[14]	output		G13	2	False	LVCMOS33
11	TFT_rgb[15]	output		H12	2	False	LVCMOS33
12	TFT_rgb[1]	output		M16	3	False	LVCMOS33
13	TFT_rgb[2]	output		N16	3	False	LVCMOS33
14	TFT_rgb[3]	output		N15	3	False	LVCMOS33
15	TFT_rgb[4]	output		N14	3	False	LVCMOS33
16	TFT_rgb[5]	output		H15	2	False	LVCMOS33
17	TFT_rgb[6]	output		J16	3	False	LVCMOS33
18	TFT_rgb[7]	output		M13	3	False	LVCMOS33
19	TFT_rgb[8]	output		L14	3	False	LVCMOS33
20	TFT_rgb[9]	output		L16	3	False	LVCMOS33
21	TFT_vs	output		T17	3	False	LVCMOS33
22	clk50M	input		T9	4	False	LVCMOS33
23	reset_n	input		B16	1	False	LVCMOS33

图 25-17 TFT 彩条实验管脚绑定表

对于时钟约束，本实验的输入时钟为 50MHz，该时钟进来就直接作为 PLL 的输入，PLL 输出 33M 作为 TFT 屏幕驱动。管脚分配完成之后，生成本次实验所需的 bit 文件，将 bit 文件下载至开发板中，最终测试结果如下图 25-18 所示。

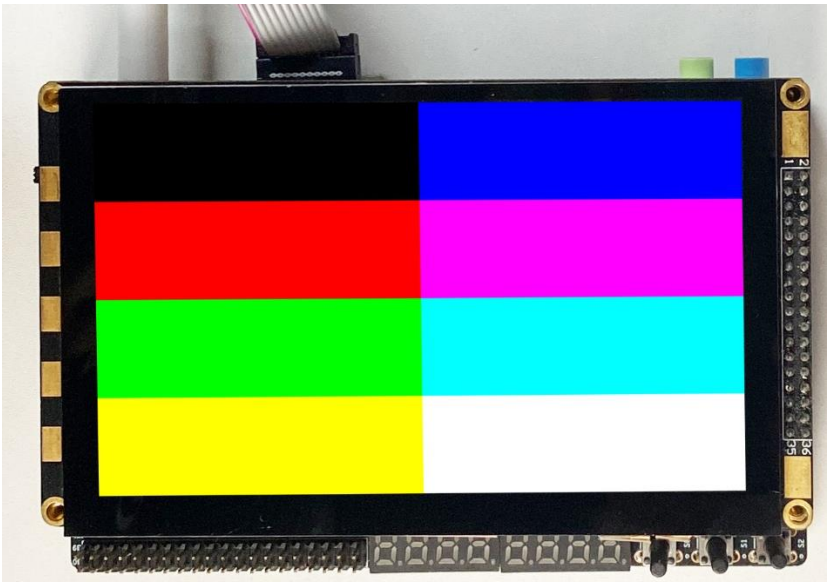


图 25-18TFT 屏幕驱动测试

通过板级测试可知，TFT 屏驱动设计能够稳定正确的刷新 TFT 显示屏并控制正确的显示位置，因此设计无误。

后续，就可以使用该控制器再结合一定的图像信号产生电路实现更多更复杂的显示系统设计，并且可以稍微修改相关参数直接应用于 VGA 接口显示系统。

26 基于 FPGA 的 HDMI/DVI 显示

工程源码	|----02_设计实例 |----ch26_hdmi_color
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

随着网络带宽、存储器容量、处理器解码能力的不断发展，与我们日息相关的图像显示领域也出现了非常大的进步。以显示器显示原理来说，从最早期的 CRT 阴极射线管扫描成像开始，到等离子电视，再到现在随处可见的液晶电视、以及现在高端领域里面使用的 OLED 显示器，整个显示器领域从最早期的黑白显示效果慢慢进化到现在的 OLED 自发光真彩色显示，视觉效果大大提升了。再从显示器的物理分辨率来说，早期的 CRT 显示器，其最大分辨率为 800*600 像素，而现在的液晶显示器，分辨率已经一路从 720p 发展到 1080p、2K、4K 以及最新的 8K 分辨率，显示效果更加细腻，色彩还原效果也不断提升。

随着显示技术的发展，与显示技术密切相关的数据传输方式和能力也在不断的提升。传输方式从最早期的 VGA 模拟信号传输方式发展到 DVI 接口、HDMI 接口、DisplayPort 接口等，传输的数据质量，数据容量也都有了大的提升。传输方式的变化主要与成像原理和数据速率相关。

本节对 DVI 电路接口和 DVI 数据链路原理做简单的介绍，并在 FPGA 上实现基于 DVI 接口的显示器彩条显示实验。

26.1基于 FPGA 的 DVI 电路设计

在高云开发板上，设计了一路简易的 HDMI 发送接口，使用该接口，可以使 FPGA 能够仅使用 8 个 I/O 就完成高达 1280*720@60Hz 的图像输出。

在早期基于 FPGA 的开发板上，实现 HDMI 一般使用的是 HDMI 发送芯片，典型的例如 ADV7513、sil9022、CH7301，使用这些芯片实现 HDMI 发送，本质上还是将 FPGA 输出的 24 位像素数据+3 位的控制信号（HSYNC、VSYNC、DE）接入这些芯片，然后由这些芯片完成数据的编码和串行发送。

下图 26-1 为 DE10_NANO_SOC 开发板上使用 ADV7513 实现的 HDMI 发送电路，可以看到，除去用于音频传输的 I2S 相关信号，仅与视频数据传输相关

的信号就高达 28 个。这在一些 FPGA IO 相对紧张的应用中，属于较大的浪费了。

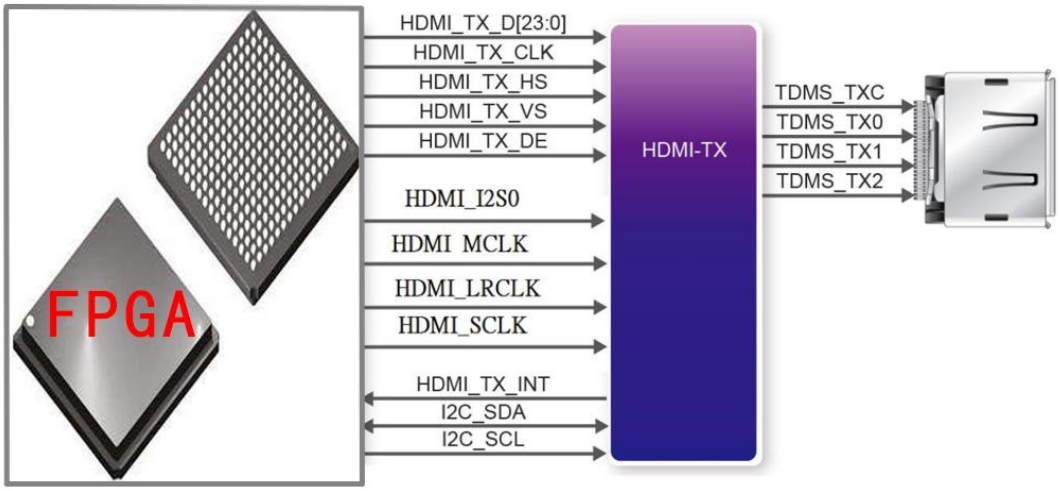


图 26-1 ADV7513 实现的 HDMI 发送电路

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，例如 Cyclone IV E 或者 Spartan-6，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 26-2 为使用 FPGA 直接编程实现 HDMI 接口的示意图：

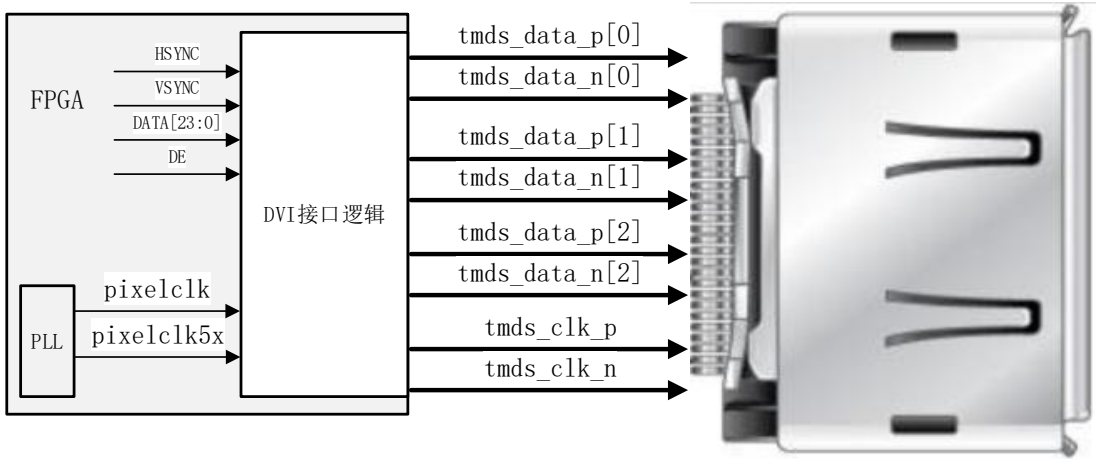


图 26-2 TMDS 编码和串行发送器

高云开发板上板载的 HDMI 接口示意图如下图 26-3 所示，其中 TPD12S016 芯片为 TVS 器件，保护线路，防止 HDMI 上差分线之间的压差过高。整个电路成本低廉，却能达到理想的传输效果。

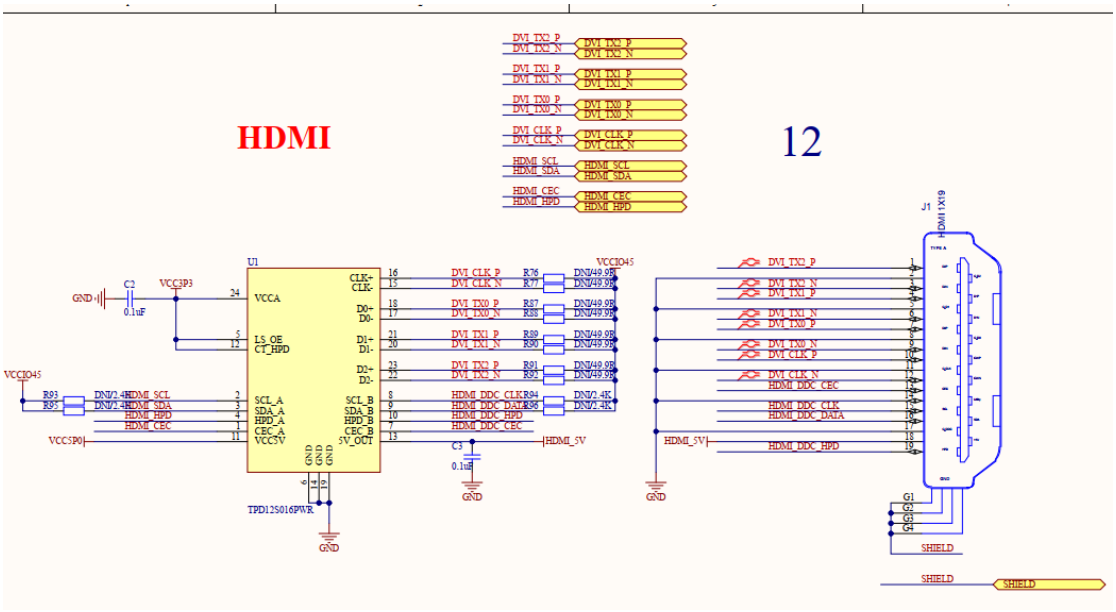


图 26-3 差分线路 HDMI 传输电路设计

下图为高云开发板上 HDMI 接口的硬件实物图。

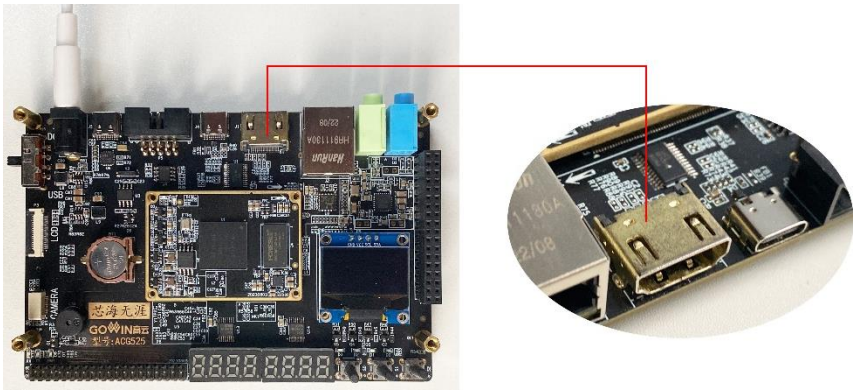


图 26-4 高云开发板 HDMI 接口硬件实物图

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。

26.2HDMI 与 DVI 的区别和联系

26.2.1DVI 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、数字投影机等显示设备上。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。

26.2.2HDMI 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。

26.2.3HDMI 与 DVI 的区别

1. 接口外观不同，下图 26-5 中，左侧为 HDMI 接口，右侧为 DVI 接口，可以看出 HDMI 接口体积比 DVI 小。
2. 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。
3. HDMI 最高传输速率比 DVI 高，HDMI 可以传输更高清数字视频信号。
4. 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 15 米，DVI 只能达到 8 米。

图 26-5 DVI 接口和 HDMI 接口

26.2.4HDMI 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。

26.2.5HDMI 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。

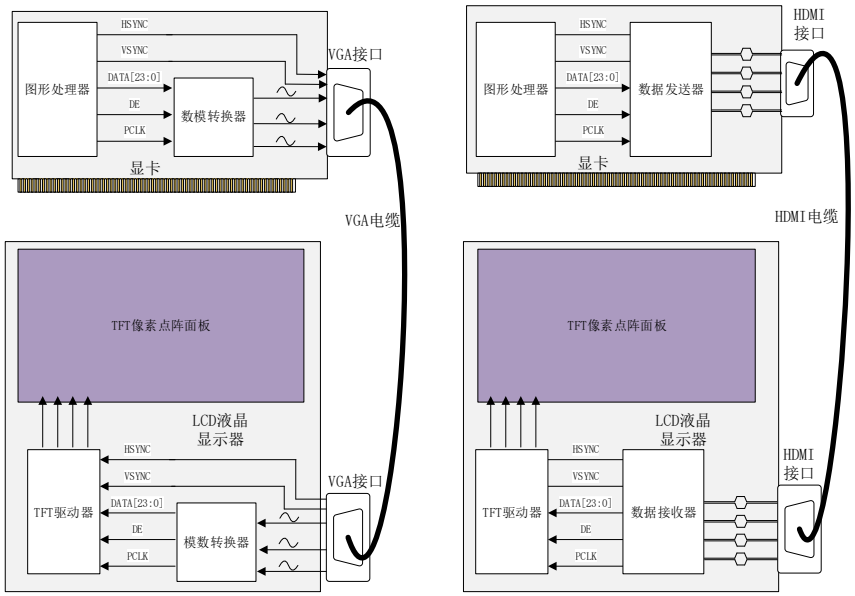


图 26-6 VGA 和 HDMI 接口与显卡传输图像数据流变换图

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。

26.3DVI 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。在输出传输过程中，还包括了输入接口层、TMDS 发送器、TMDS 接收器和输出接口层。

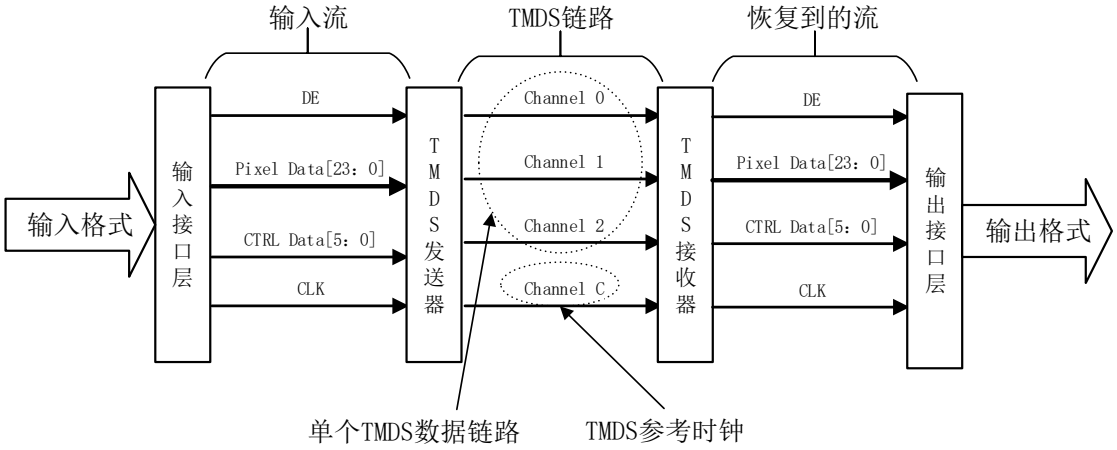


图 26-7 DVI 数据链路

26.3.1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。

26.3.2TMDS 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。

26.3.3TMDS 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。

26.3.4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。

26.4TMDS 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。

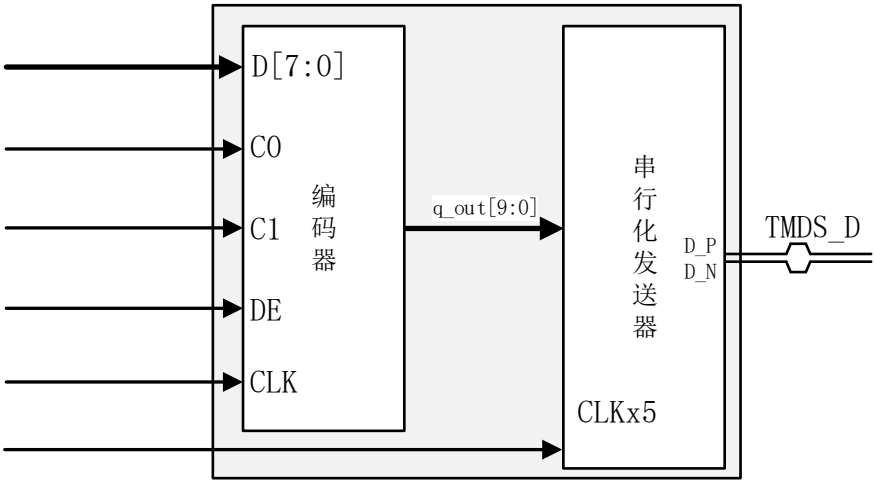


图 26-8 TMDS 原理与实现输入输出示意图

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。

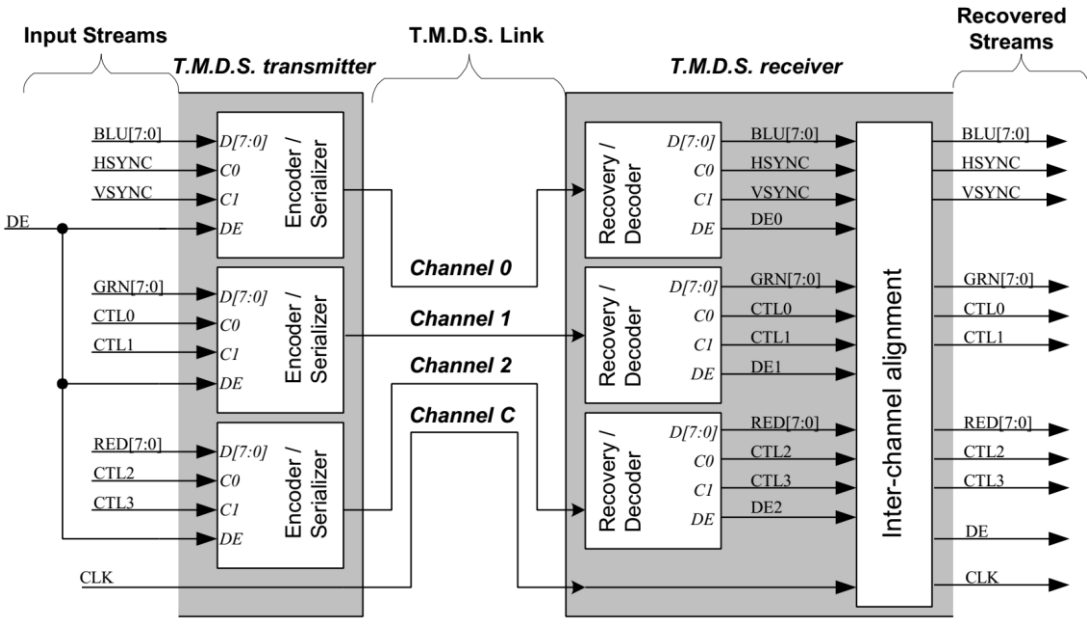


图 26-9 TMDS 传输通道编码解码概况图

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。

26.5TMDS 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。

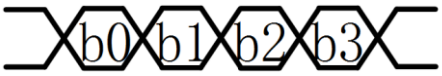


图 26-10 数据流示意图

26.6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。

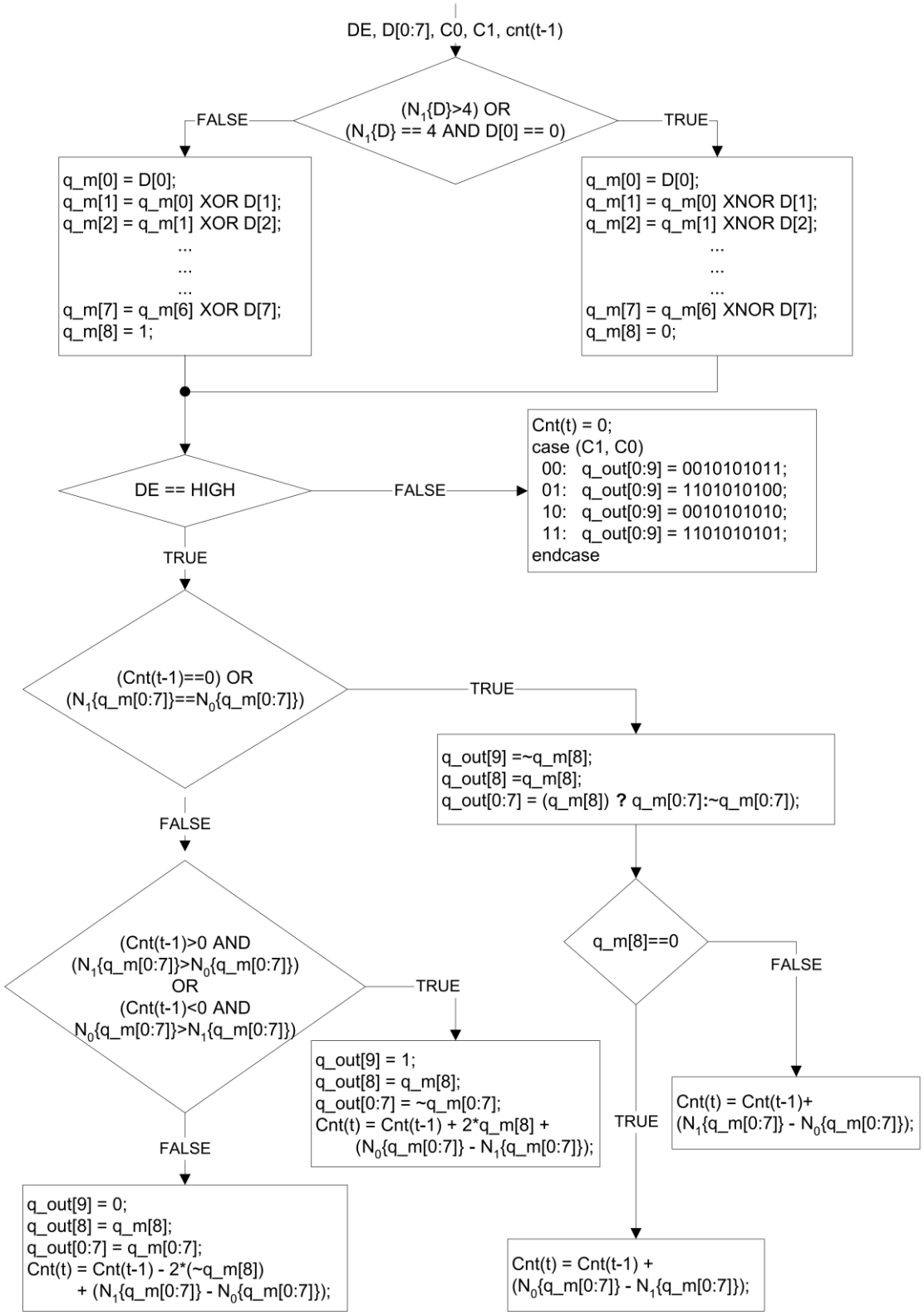


图 26-11 最小化传输编码

表 26-1 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

26.6.1数据编码算法流程

1. 先统计输入的数据中有多少个 1。
2. 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。
3. 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。)

26.6.2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。

26.6.3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。

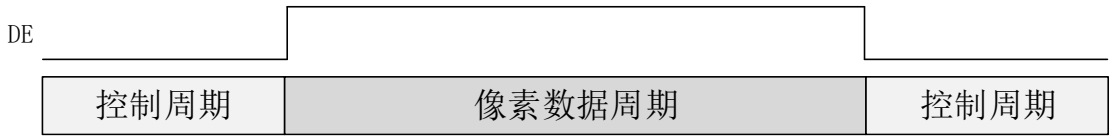


图 26-12 像素数据周期和控制周期切换控制信号

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 26-2 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 26-2 3 个通道控制信号内容

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

下表 26-3 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 26-3 2 位控制信号对应状态编码值

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

26.7TMDS 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

    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 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
```

26.8串行发送模块

26.8.1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 TMDS 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 26-13 所示。

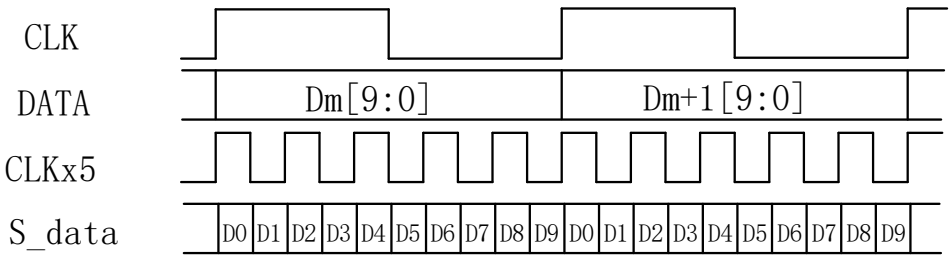


图 26-13 串行发送时序图

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。

26.8.2FPGA 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，Gowin 的 FPGA 也不例外，在使用的时候我们可以在 Gowin 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。

本次实验需要使用到 ODDR（Dual Data Rate Output），实现双倍数据速率输出。ODDR 模式，用于从 FPGA 器件传输双倍数据速率信号。Q0 为双倍速率数据输出，Q1 用于 Q0 所连的 IOBUF/TBUF 的 OEN 信号，ODDR 的逻辑框图如图 26-14 所示，时序图如下图 26-15 所示。

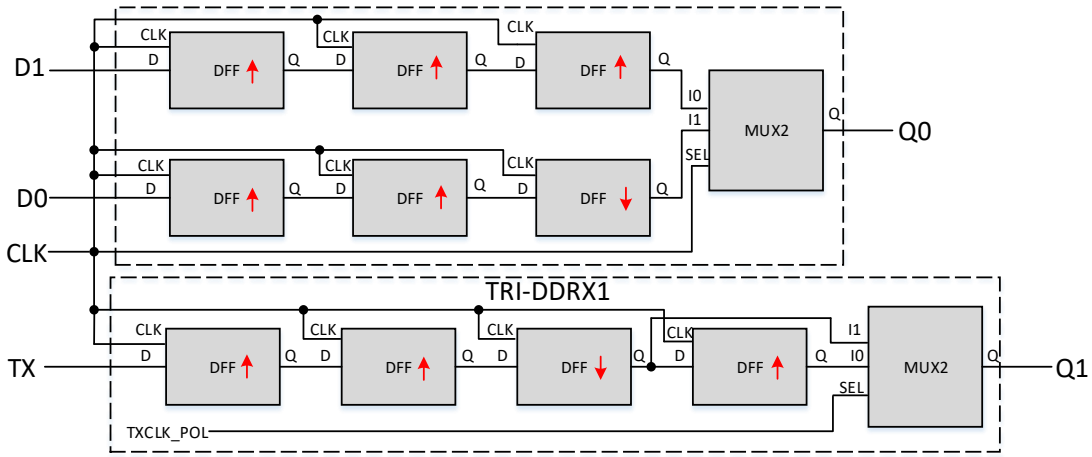


图 26-14 ODDR 逻辑框图

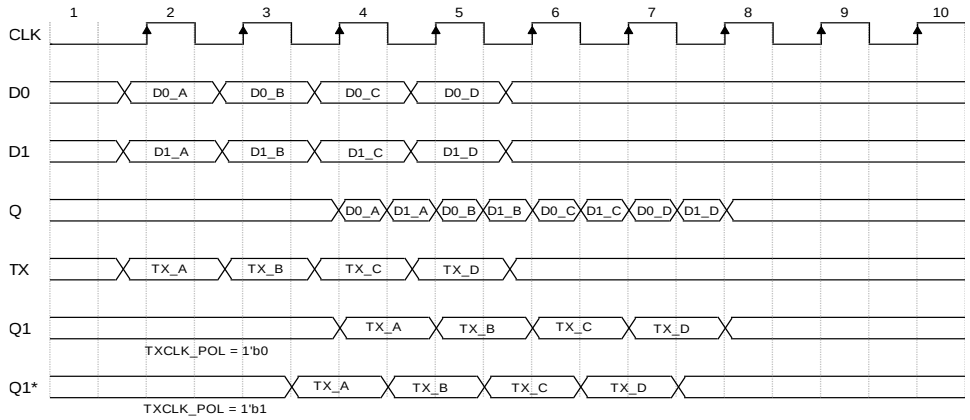


图 26-15 ODDR 时序图

ODDR 端口示意图如下图 26-16 所示，端口信号说明如下表 26-4 所示。

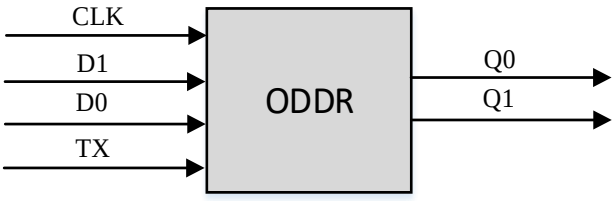


图 26-16 ODDR 端口示意图

表 26-4 ODDR 端口说明表

端口名	I/O	描述
D0,D1	Input	ODDR 数据输入信号
TX	Input	通过 TRI-DDRX1 产生 Q1
CLK	Input	时钟输入信号
Q0	Output	ODDR 数据输出信号
Q1	Output	ODDR 三态使能控制输出信号，可连接 Q0 所连的 IOBUF/TBUF 的 OEN 信号，或悬空。

ODDR 使用的时候可以直接实例化原语，例化代码如下所示：

```
ODDR uut(
    .Q0(Q0),
    .Q1(Q1),
    .D0(D0),
    .D1(D1),
    .TX(TX),
    .CLK(CLK)
);
defparam uut.TXCLK_POL=1'b0;
```

上述代码中的 TXCLK_POL 是用来控制 Q1 输出极性的，1’b0 代表 Q1 上升沿输出，1’b1 代表 Q1 下降沿输出。

当然我们可以通过调用 IP 的方式使用 ODDR，在 IP Core Generator 界面搜索 DDR，选择 IO 下面的 DDR，进入 IP 配置界面，如下图 26-17 所示：

图 26-17 进入 IP 配置界面示意图

IP 配置界面如下图 26-18 所示。

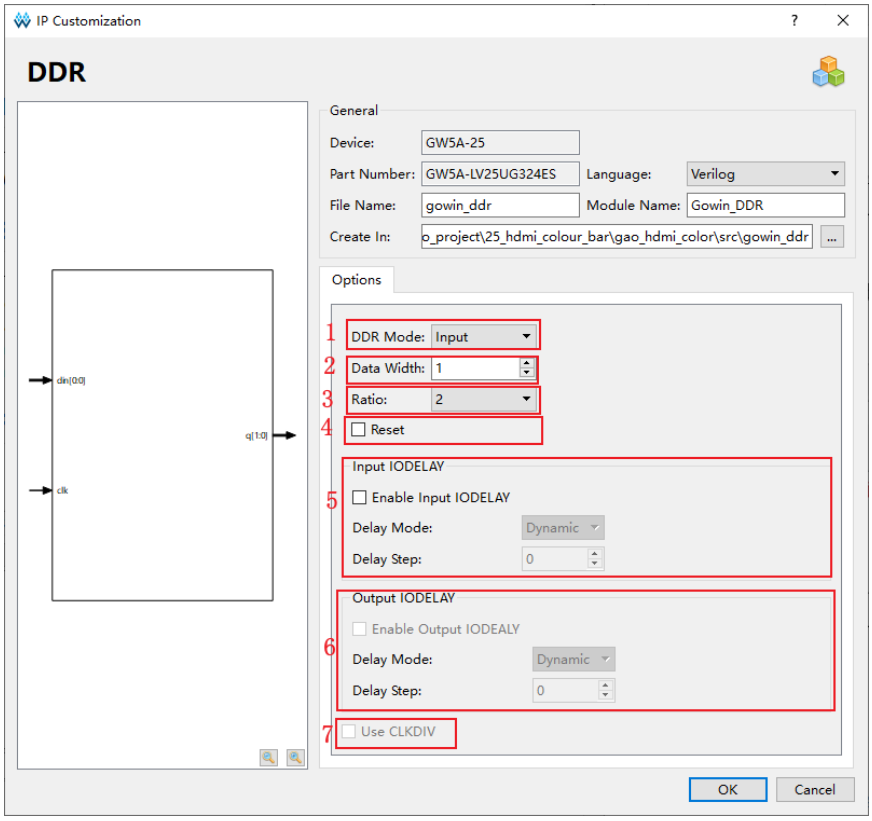


图 26-18 IO_DDR 配置界面

下面对上述图中的各项配置进行简要说明。

1. DDR Mode：配置 DDR 模式，包括输入“Input”、输出“Output”、三态“Tristate”和双向“Bidirectional”。
2. Data Width：配置 DDR 的数据宽度，支持 1~64。
3. Ratio：配置 DDR 数据转换的比值，包括 2，4，7，8，10，14，16，32。
4. Reset：Radio 选择 2 时，可选择使能或不使能此选项，使能时将实例化 IDDRC 或 ODDRC。
5. Input IODELAY：

- Enable Input IODELAY：配置 DDR 是否使用输入延时模块。
- Delay Mode：配置 Delay 模式，“Dynamic”表示使用 IODELAY 并动态调整延时步数，“Static”表示使用 IODELAY 并静态调整延时步数，“Adaptive”表示使用 IODELAY 并自适应调整延时步数。
- Delay Step：选择静态调整延时步数，范围 0~255。

6. Output IODELAY：

- Enable Output IODELAY：配置 DDR 是否使用输出延时模块。
- Delay Mode：配置 Delay 模式，“Dynamic”表示使用 IODELAY 并静态调整延时步数，“Static”表示使用 IODELAY 并静态调整步数，“Adaptive”表示使用 IODELAY 并自适应调整延时步数。
- Delay Step：选择静态调整延时步数，范围 0~255。

7. Use CLKDIV：使能时将实例化 CLKDIV，对时钟信号 fclk 进行分配，Ratio 为 2 时不勾选。

26.8.3TMDS 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 ODDR 即可人，下图 26-19 为 1 位 ODDR 示意图。

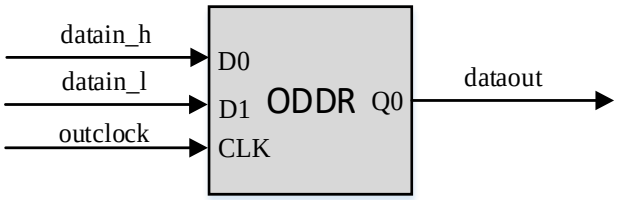


图 26-19 1 位的 ODDR 使用方式

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，所以需要每个时钟周期切换一次送给 datain_h 和 data_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 26-5 每个时钟周期对应的 datain_h 和 datain_l 连接关系

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

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既清零，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图 26-20 所示的逻辑电路图：

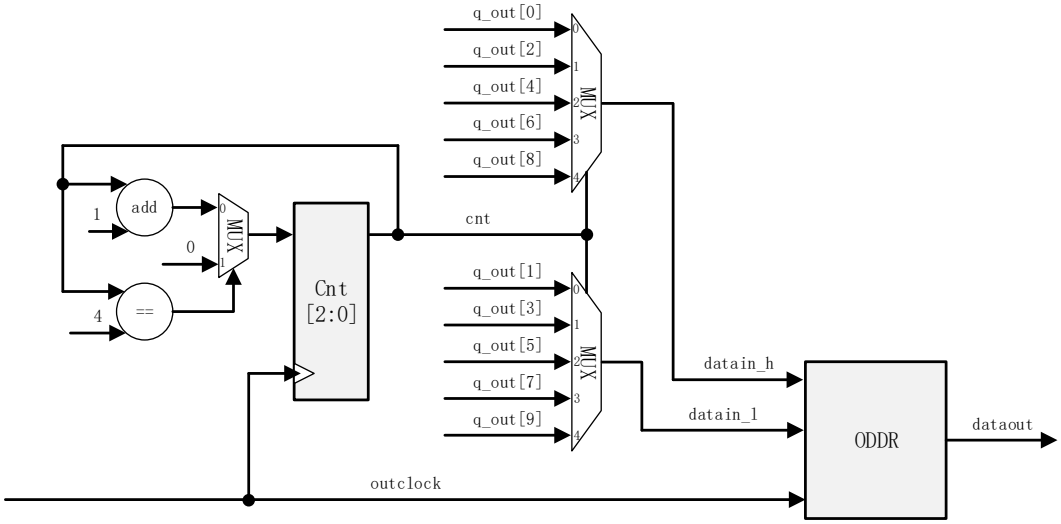


图 26-20 多路器思路切换输入的位

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。

26.8.4串行发送模块编码实现

通过上述分析可以知道，送给 DDR 的 datain_l端口的数据为 q_out 的 1、3、5、7、9 位，送给 DDR 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。

1. 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_I 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_3[1]};
wire [4:0]TMDS_2_h={datain_2[8],datain_2[6],datain_2[4],datain_2[2],datain_3[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]};
```

2. 用 5 倍的速率时钟将数据输入移位寄存器，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
```

3. 移位完成后，只需要将每个通道的高低位移位寄存器的值分辨送往 ODDR 的 datain_h 和 datain_l 端口即可，这里通过调用一个 IO_DDR IP 实现，IO_DDR IP 配置如下图 26-21 所示。

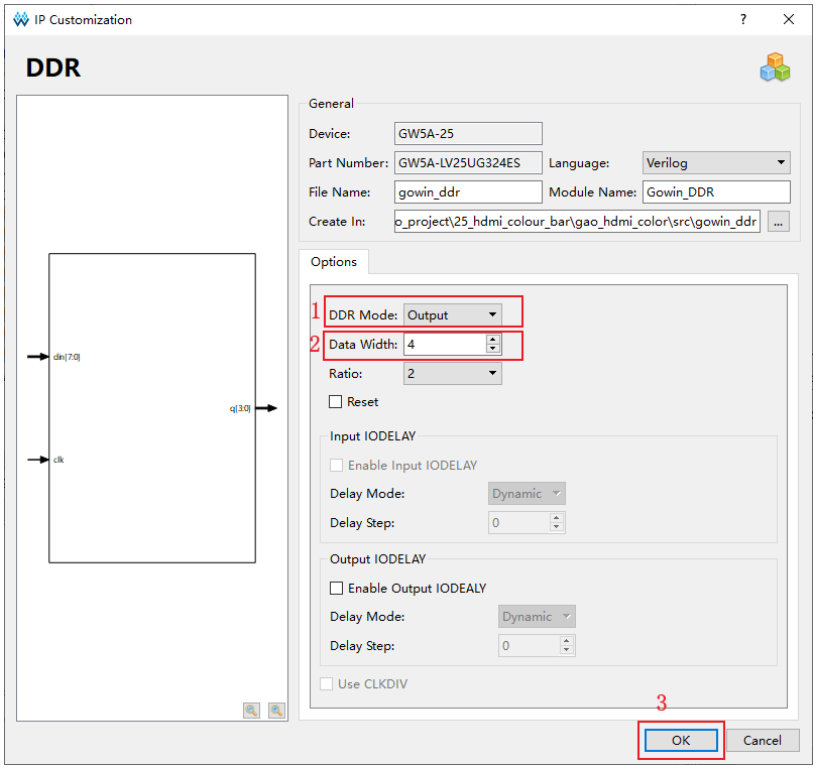


图 26-21 IO_DDR 配置界面示意图

上图所示的配置中，调用了 4 个输出的 ODDR，完成所有通道数据的输出，调用代码如下所示：

```
Gowin_DDR Gowin_DDR_1(
    .din({TMDS_shift_3l[0],TMDS_shift_2l[0],TMDS_shift_1l[0],TMDS_shift_0l[0],
    TMDS_shift_3h[0],TMDS_shift_2h[0],TMDS_shift_1h[0],TMDS_shift_0h[0]}),
    .clk(clkx5), //input clk
    .q({dataout_3_p,dataout_2_p,dataout_1_p,dataout_0_p}) //output [3:0] q
);


Gowin_DDR Gowin_DDR_2(
    .din(~{TMDS_shift_3l[0],TMDS_shift_2l[0],TMDS_shift_1l[0],TMDS_shift_0l[0],
    TMDS_shift_3h[0],TMDS_shift_2h[0],TMDS_shift_1h[0],TMDS_shift_0h[0]}),
    .clk(clkx5), //input clk
    .q({dataout_3_n,dataout_2_n,dataout_1_n,dataout_0_n}) //output [4:0] q
    );
```

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 代码如下所示：

```
module serdes_4b_10to1(
    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
);

    reg [2:0] TMDS_mod5 = 0;  // 模 5 计数器

    reg [4:0] TMDS_shift_0h = 0, TMDS_shift_0l = 0;
    reg [4:0] TMDS_shift_1h = 0, TMDS_shift_1l = 0;
    reg [4:0] TMDS_shift_2h = 0, TMDS_shift_2l = 0;
    reg [4:0] TMDS_shift_3h = 0, TMDS_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_3[1]};
    wire [4:0] TMDS_2_h =
{datain_2[8],datain_2[6],datain_2[4],datain_2[2],datain_3[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]};


    // 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

    Gowin_DDR Gowin_DDR_1(
        .din({TMDS_shift_3l[0],TMDS_shift_2l[0],TMDS_shift_1l[0],TMDS_shift_0l[0]
,TMDS_shift_3h[0],TMDS_shift_2h[0],TMDS_shift_1h[0],TMDS_shift_0h[0]}),
        .clk(clkx5), //input clk
        .q({dataout_3_p,dataout_2_p,dataout_1_p,dataout_0_p}) //output [3:0] q
    );

    Gowin_DDR Gowin_DDR_2(
        .din(~{TMDS_shift_3l[0],TMDS_shift_2l[0],TMDS_shift_1l[0],TMDS_shift_0l[0
],TMDS_shift_3h[0],TMDS_shift_2h[0],TMDS_shift_1h[0],TMDS_shift_0h[0]}),
        .clk(clkx5), //input clk
        .q({dataout_3_n,dataout_2_n,dataout_1_n,dataout_0_n}) //output [4:0] q
    );

endmodule
```

26.9DVI 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：

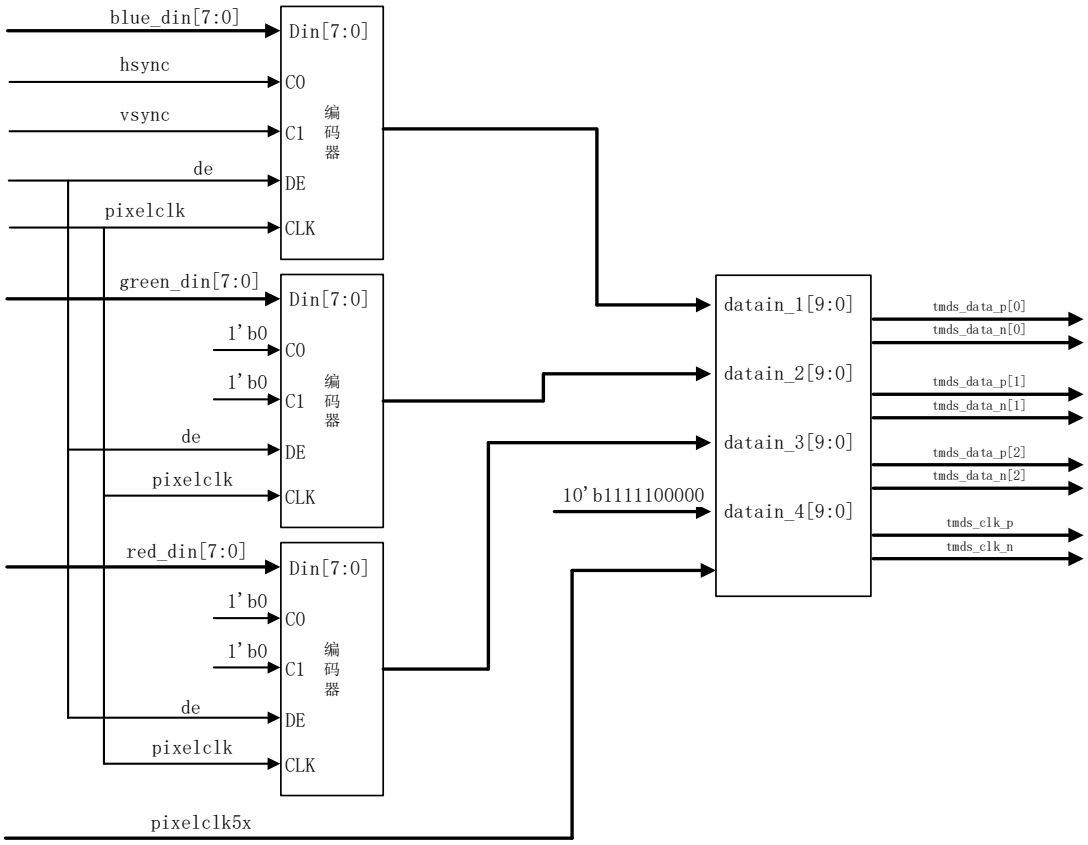


图 26-22 DVI 发送原理图

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 26-23 所示。

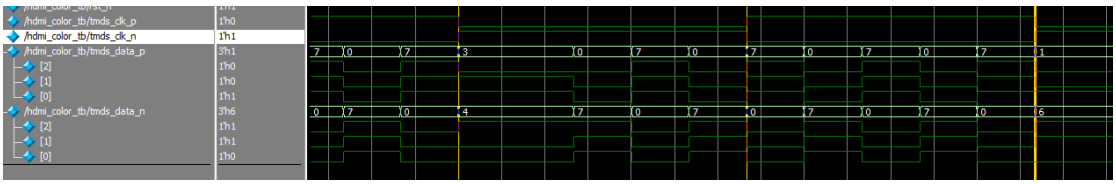


图 26-23 tmds_clk 和 data 的关系图

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(
    input       pixelclk,       // system clock
    input       pixelclk5x,     // system clock x5
    input       rst_n,          // 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,
    output      tmds_clk_n,
    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_n(rst_n),
        .din(blue_din),
        .c0(hsync),
        .c1(vsync),
        .de(de),
        .dout(blue)
    );

    encode encg(
        .clk(pixelclk),
        .rst_n(rst_n),
        .din(green_din),
        .c0(1'b0),
        .c1(1'b0),
        .de(de),
```

```
        .dout(green)
    );

    encode encr(
        .clk(pixelclk),
        .rst_n(rst_n),
        .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
```

26.10基于 DVI 接口的显示器彩条显示实验

设计完成该控制器之后，在使用时，只需要基于基本的 VGA 或者 TFT 显示系统，将 VGA 或 TFT 控制器逻辑的相关信号接到 dvi_encoder 的对应端口上即可。如下图 26-24 所示：

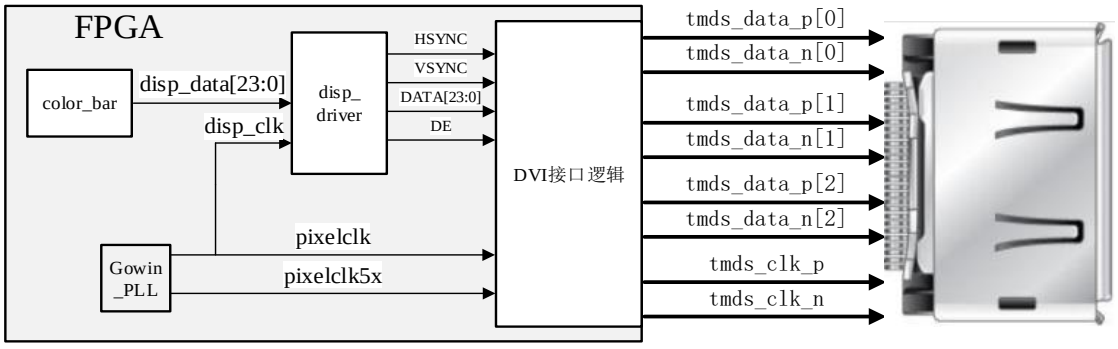


图 26-24 DVI 输出逻辑接口图

可以看到，在上述框图中，只是将传统的直接输出到 FPGA 管脚的 VGA/TFT 控制器的输出信号接到了 DVI 接口逻辑的对应输入接口，即可实现 HDMI 接口的图像发送。所以，使用该 DVI 接口逻辑不会改变原有图像显示系统的框架结构，只是对最终输出的信号多加入了一级变换处理而已。由于在 VGA 控制器设计与验证章节，我们已经完成了 VGA 控制器的设计，因此我们就可以将本节所设计的 DVI 发送器连接到 VGA 控制器上，实现 VGA 控制器设计章节里在显示屏上显示彩条的功能。

本节实验，配置 VGA 控制器输出 800*480 分辨率的时序信号，在“disp_parameter_cfg.v”文件中设置使能“Resolution_800x480”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 显示器需要使用 PLL 产生相应频率的时钟信号（480p 为 33MHz），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 480p 分辨率，也就是 165MHz。

其中 colour_bar 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。

pll 产生 VGA 控制器输出 800*480 分辨率所需的 375MHz 时钟和 DVI 编码器进行串行编码时所需的 165MHz 时钟。至此，整个验证工程就设计完成了。

至此，整个验证工程就设计完成了，接下来开始硬件连接。

26.10.1系统所需硬件

1. 高云开发板
2. 电源线一根（可选）
3. 下载器一个
4. HDMI 电缆一根
5. 支持 HDMI 接口的液晶显示器一台

26.10.2硬件连接

本次实验的硬件连接如下图 26-25 所示。

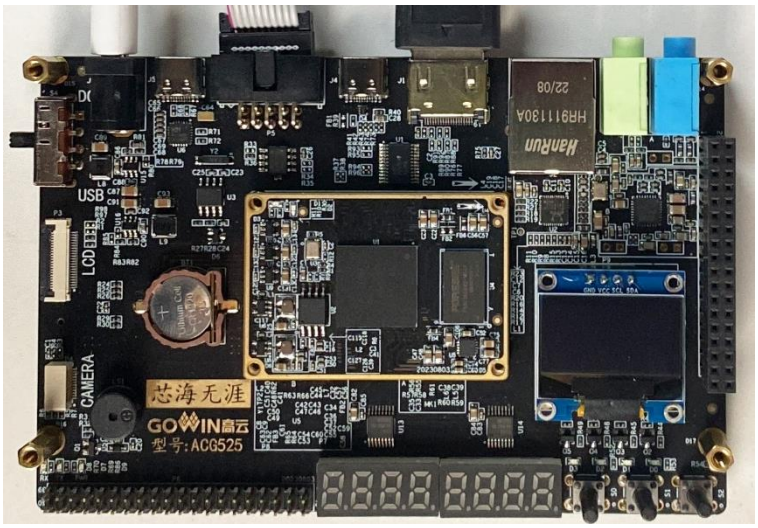


图 26-25 硬件连接图

图像数据通过开发板上的 HDMI 接口输出，在连接 HDMI 线缆时请确保线缆一端连接在开发板的 HDMI 接口，一端连接在显示器的 HDMI 接口上。

26.10.3下载与验证

连接完硬件后，我们需要为设计分配引脚。除了时钟和复位引脚外，还需要分配 HDMI 引脚信号。本次设计的引脚分配如下表 26-6 所示。

表 26-6 管脚分配表

Port	Location.	Port	Location..
clk	T9	rst_n	B16
led	D14		
tmds_data_p[2]	R5	tmds_data_n[2]	T5
tmds_data_p[1]	T6	tmds_data_n[1]	V6
tmds_data_p[0]	R7	tmds_data_n[0]	T7
tmds_clk_p	M10	tmds_clk_n	N9

打开 FloorPlanner 对引脚进行分配，如下图 26-26 所示。

I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	clk	input		T9	4	False	LVCMOS33
2	led	output		D14	1	False	LVCMOS33
3	rst_n	input		B16	1	False	LVCMOS33
4	tmds_clk_n	output		N9	5	False	LVCMOS33
5	tmds_clk_p	output		M10	5	False	LVCMOS33
6	tmds_data_n[0]	output		T7	5	False	LVCMOS33
7	tmds_data_n[1]	output		V6	5	False	LVCMOS33
8	tmds_data_n[2]	output		T5	5	False	LVCMOS33
9	tmds_data_p[0]	output		R7	5	False	LVCMOS33
10	tmds_data_p[1]	output		T6	5	False	LVCMOS33
11	tmds_data_p[2]	output		R5	5	False	LVCMOS33

图 26-26 管脚约束

26.10.4专用管脚报错

完成管脚绑定后，直接将工程进行全编译，编译的时候会提示如下错误，如图 26-27 所示。

```
Processing netlist completed
Reading constraint file: "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\src\gao_hdmi_color.cst"
Physical Constraint parsed completed
Running placement.....
ERROR  (PR2028) : The constrained location is useless in current package
ERROR  (PR2017) : 'tmds_data_p[2]' cannot be placed according to constraint, for the location is a dedicated pin (CPU/SSPI)
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\pnr\gao_hdmi_color.pin.html" completed
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\pnr\gao_hdmi_color.rpt.html" completed
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\pnr\gao_hdmi_color.rpt.txt" completed
Thu Sep 14 14:44:48 2023
```

图 26-27 编译提示错误

出现上述所示的错误，是因为 tmds_data_p[2]分配的引脚是一个专用引脚，是给 CPU/SSPI 使用的，解决方式就是将该管脚设置为通用管脚，操作方式如下所示：

1. 右击 Place & Route，选择 Configuration，进入设置界面，如下所示。

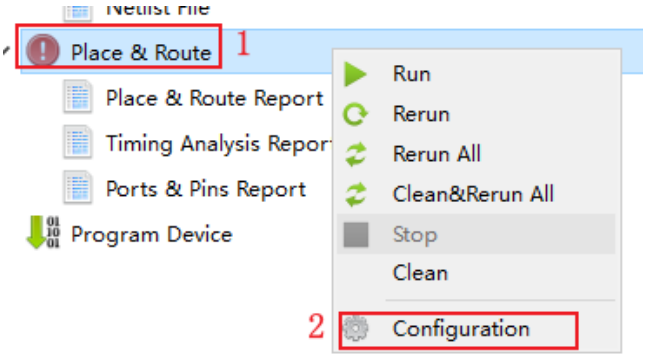


图 26-28 进入设置界面

2. 然后点击 Dual-Puroose Pin，根据错误提示信息，选择“Use SSPI as regular IO”和“Use CPU as regular IO”，将其设置为通用管脚，如下所示。

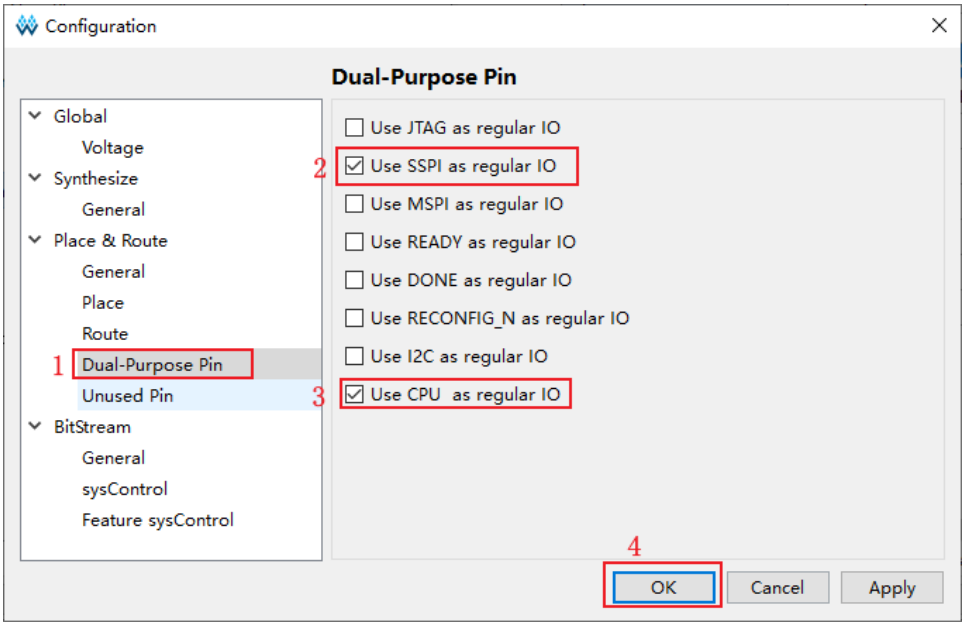


图 26-29 将专用引脚设置为通用管脚

设置完成之后，重新布局布线，可以看到报错消失，如下所示，成功生成了 bit 文件，将 bit 文件下载至开发板中。

```
[10%] Placement Phase 0 completed
[20%] Placement Phase 1 completed
[30%] Placement Phase 2 completed
[50%] Placement Phase 3 completed
Running routing......
[60%] Routing Phase 0 completed
[70%] Routing Phase 1 completed
[80%] Routing Phase 2 completed
[90%] Routing Phase 3 completed
Running timing analysis......
[95%] Timing analysis completed
Placement and routing completed
Bitstream generation in progress......
Bitstream generation completed
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\
Generate file "H:\01_gaoyun\gao_project\26_hdmi_colour_bar\gao_hdmi_color\impl\
Thu Sep 14 14:54:15 2023
```

图 26-30 成功生成 bit 文件

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 26-31 为该工程运行时在 HDMI 显示器上的实际显示效果。

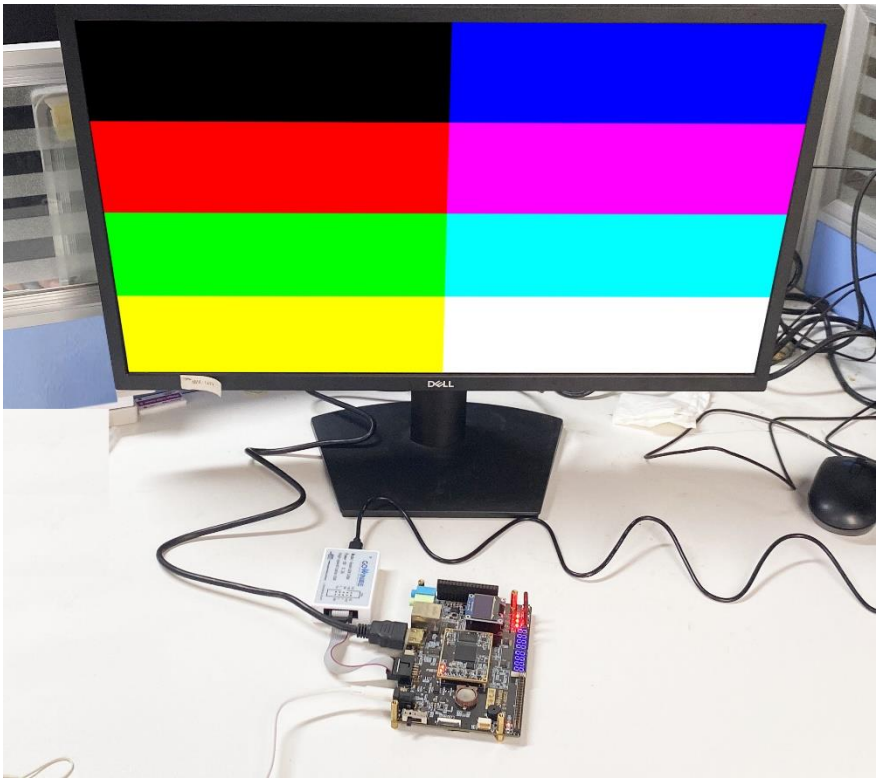


图 26-31 高云开发板彩条实验效果图

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 800*480，说明本次设计的 DVI 发送模块功能正常，本次设计成功。

26.11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。在后续的各种图像相关的应用中，只需要按照本节内容最后采取的方法，将 VGA 控制信号送入 dvi_encoder 编码器，就能实现 HDMI/DVI 接口显示器的驱动了。

本方案不仅结构简单，占用 FPGA 逻辑资源少，而且经济实惠，不需要使用专用的 RGB 转 HDMI 芯片，也大大缩减了对 FPGA 管脚占用的数量，适合在众多成本敏感的场合应用。

27 基于 HDMI 和 TFT 显示屏的图片显示

工程源码	|----02_设计实例 |----ch27_rom_image_tft_hdmi
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

在前面的章节中通过显示彩条的形式对 HDMI 显示器和 TFT 屏显示进行了板级测试。在此基础上，本节将实现在 TFT 屏上显示一张图片。待显示的图片存放于 FPGA 的 ROM 存储器中，由于存储图片数据的 ROM 消耗 FPGA 中 RAM 资源，而 FPGA 中 RAM 资源有限，所以显示的图片尺寸不能过大，过大将消耗过多的 RAM 资源，而且可能由于图片数据的过大导致超过 FPGA 的最大 RAM 资源量无法实现功能。本实验将选取图片尺寸为 200*200 大小的作为显示对象在 TFT 屏上进行显示。

27.1系统整体设计

系统整体设计结构框图如下图 27-1 所示。

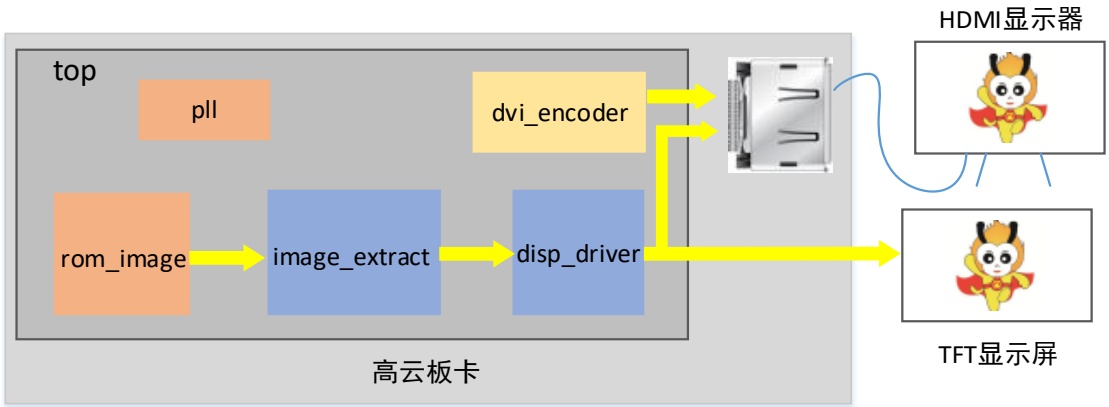


图 27-1 基于 TFT/HDMI 显示屏显示图像的工程架构

整个系统主要包括 4 个模块部分，模块功能描述如下：

1. PLL 模块：产生系统内各模块工作所需的时钟，直接使用 IP。系统内模块均工作在 33MHz（模块工作时钟均采用与 TFT 屏驱动时钟）下，高云开发板 FPGA 管脚输入时钟为 50MHz，这样 PLL 的输入时钟为 50MHz，输出时钟为 33MHz。具体配置与上节 TFT 屏驱动板级测试工程中 PLL 设置一样。（在本节后端会为适配 HDMI 设计而添加 165M 时

钟，这里也可以先预留出 165M 时钟方便后续使用）

2. disp_driver 模块：TFT 屏显示驱动，上一节已经详细讲解了实现过程，这里就不多说。
3. rom_image 模块：用来存储待显示的图片数据，直接使用 ROM IP。具体配置和图片数据的生成后面详细介绍。
4. image_extract 模块：作为 rom_image 模块和 disp_drive 模块之间的桥梁。用来根据 TFT 屏驱动显示控制器的行列扫描位置信息去存储图片数据 ROM 中提取相应的图片数据在 TFT 屏上显示，具体设计后面详细介绍。
5. dvi_encoder 模块：HDMI 驱动模块，用于驱动 HDMI 显示屏显示图像数据。

27.2图片数据的产生及 ROM 的配置

本实验是将图片数据存储在 ROM 中，关于 ROM IP 的配置使用前面已经有章节讲过了。将数据存放在 ROM 首先需要产生 ROM 初始化的数据文件。这里借助如下图 27-2 所示的工具软件将图片转成 ROM 初始化数据文件。相关软件可在提供资料的网盘进行下载。

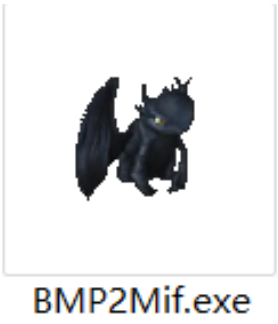


图 27-2 ROM 图片数据生成工具

双击打开软件，界面如下图 27-3 所示。

图 27-3 ROM 图片生成工具主界面

可通过点击界面的帮助按钮会弹出相关使用说明。该工具只支持 8 位或 24 位 BMP 格式图片的转换。

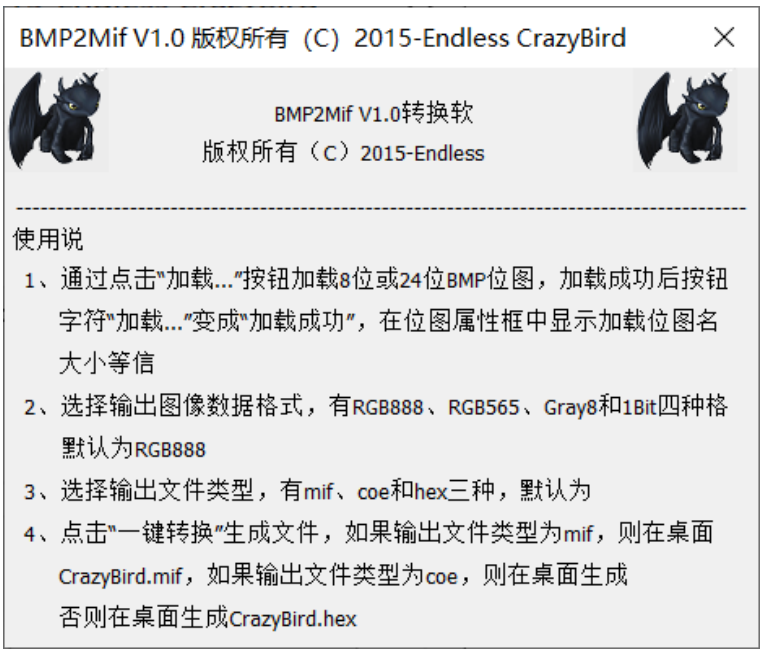


图 27-4 图片转换工具使用说明

点击加载图片，选择图片路径，在工程目录下的 srcs\image_init 文件夹中有提供一张供用户使用的测试图片及转换后的数据文件。这里的图片尺寸不能过大，避免超过 FPGA 的 RAM 资源使用量。

图 27-5 加载图片界面配置

图片加载完成后，可以在右侧位图属性看到加载图片的大小信息。然后选择输出图像格式 RGB565，输出文件类型 Coe，点击一键转换按钮。

图 27-6 一键转换

转换完成会出现如下弹框，点击确认，完成图片转换成 ROM 初始化数据文件，转换后的文件会自动存放在桌面。

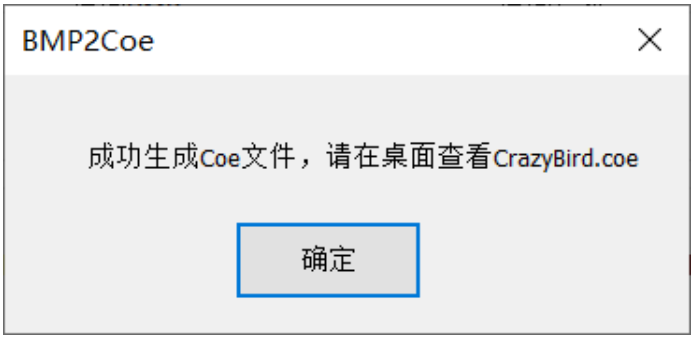


图 27-7 生成 coe 文件确认窗口

转换后的图片数据文件名为“CrazyBird.coe”，如下图所示为存放在桌面上的转换后的图片数据文件。可以将复制粘贴到工程目录下，根据自己的需要可更改文件名。

图 27-8 生成的图片用 coe 文件格式存储

有关 ROM IP 配置可参考前面“IP 核使用之 ROM”文档的讲解，具体配置如下图 27-9 所示。

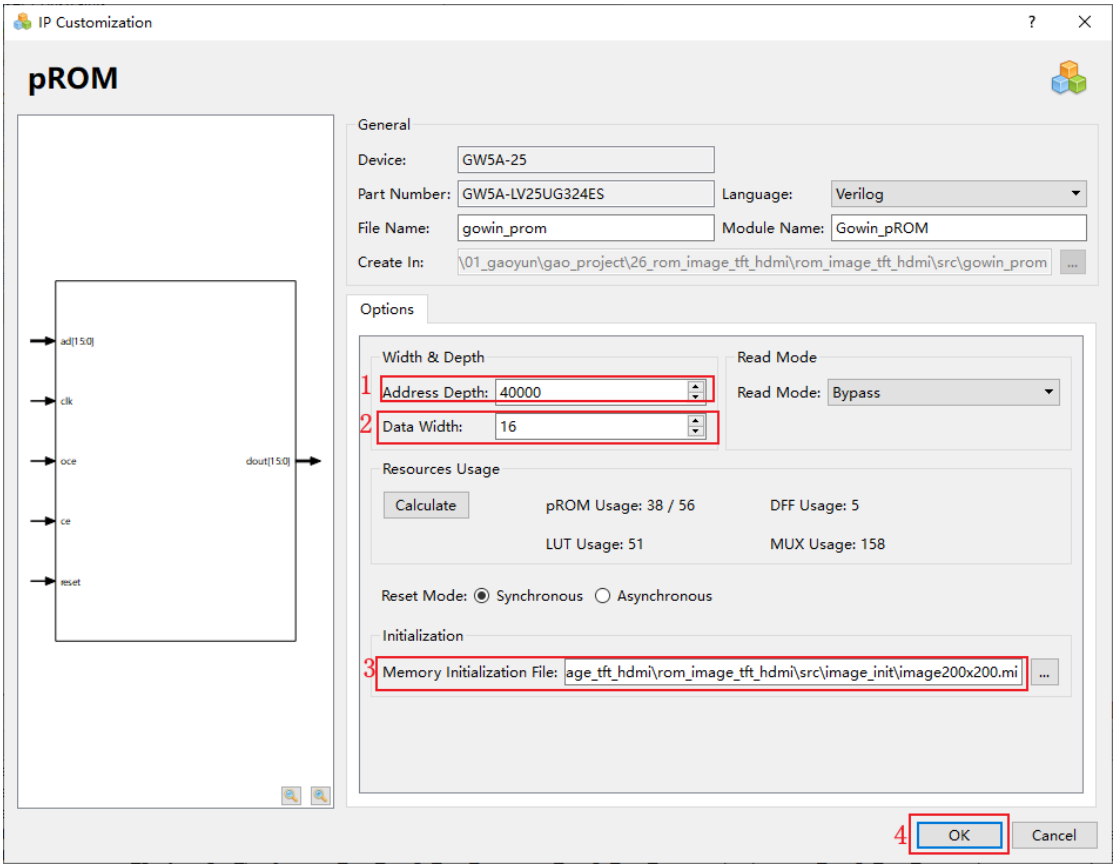


图 27-9 ROM IP 配置

生成图片数据是 RGB565 格式的，图片大小是 200*200，所以配置 ROM 端口的位宽是 16，深度是 40000。

这里需要注意的是，通过前面的得到的 coe 文件格式不能直接初始化 Gowin 软件中的 ROM IP，必须转换为 mi 格式的文件，操作如下所示：

1. 打开 Gowin 软件，新建一个.mi 格式的文件。
2. 设置.mi 格式的文件类型，设置数据格式为 Hex，Depth 为 40000，Width 为 16，点击保存得到文件头如下所示，复制文件头。

```
#File_format=Hex
#Address_depth=40000
#Data_width=16
```

3. 将 coe 文件复制至.mi 格式的文件中，修改文件头，如下图 27-10 所示。

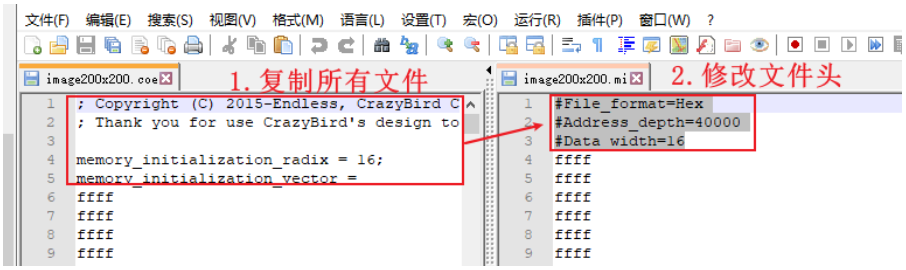


图 27-10 得到 ROM 初始化文件

27.3图片提取 image_extract 模块的设计

Image_extract 模块的功能是根据 TFT 屏驱动模块当前显示行列位置等信号控制产生读取 ROM 数据的地址，从 ROM 中读出需要显示的图像数据传给 TFT 屏显示驱动模块进行图片的显示。模块接口及接口功能描述如下所示：

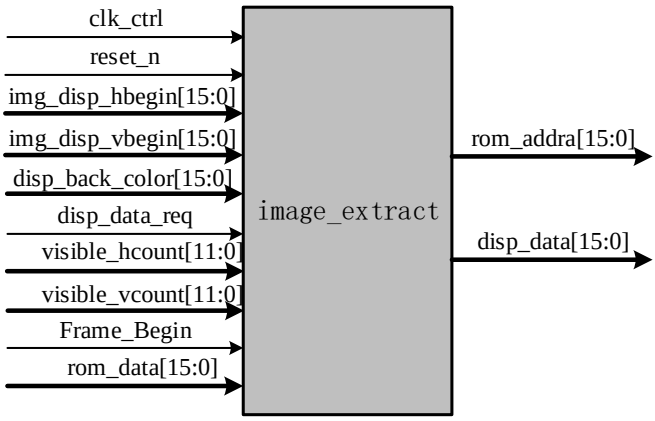


图 27-11 图片提取模块输入输出接口信息

表 27-1 模块接口功能描述

端口名	I/O	端口功能
clk_ctrl	I	模块工作时钟
reset_n	I	模块复位信号，低电平复位
img_disp_hbegin[15:0]	I	待显示图片起始像素点显示在屏幕行位置设置
img_disp_vbegin[15:0]	I	待显示图片起始像素点显示在屏幕列位置设置
disp_back_color[15:0]	I	屏幕显示的背景颜色，
disp_data_req	I	屏幕可见显示区标识信号，用于作为模块请求显示数据的标识信号
visible_hcount[15:0]	I	图像区行扫描地址
visible_vcount[15:0]	I	图像区场扫描地址
Frame_Begin	I	场信号的开始标志信号，用来控制模块即将显示新一帧图像
rom_data[15:0]	I	从 ROM 读取的图片数据
disp_data[15:0]	O	传给 TFT 屏显示的数据
rom_addra[15:0]	O	从 ROM 读取的图片数据的地址

关于 img_disp_hbegin、img_disp_vbegin 和 disp_back_color 等输入信号对应

控制显示的的示意图如下。

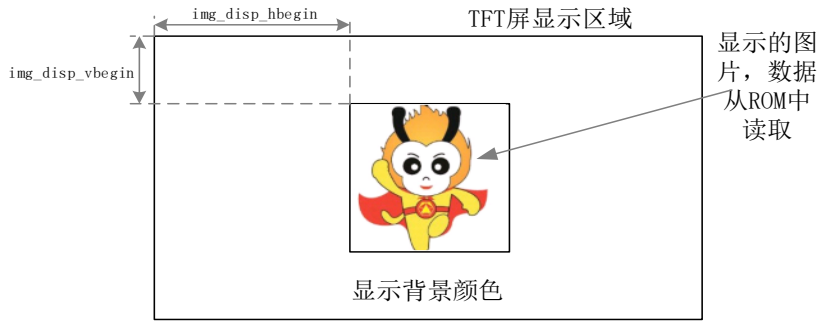


图 27-12 两个起始信号的图片对应信息区间

接下来就开始对模块代码进行设计，考虑到模块的通用性和移植性（比如移植到 4.3 寸屏幕或更改图片尺寸大小等），将 TFT 屏显示显示区域的大小，显示图片的大小、显示图片数据的位宽（这样就可以方便移植到 RGB888 图像数据格式的显示）等使用 parameter 进行参数化定义。这样将其移植到其他系统上时，只需在例化模块的时候将参数进行相应的修改即可。参数化定义的代码如下。

```
parameter H_Visible_area = 800, //整个屏幕显示区域宽度
parameter V_Visible_area = 480, //整个屏幕显示区域高度
parameter IMG_WIDTH      = 160, //图片宽度
parameter IMG_HEIGHT     = 120, //图片高度
parameter IMG_DATA_WIDTH = 16,  //图片像素点位宽
parameter ROM_ADDR_WIDTH = 16   //存储图片 ROM 的地址位宽
```

这样在输入输出端口或内部变量信号进行定义时就可以使用上面参数，具体代码如下。

```
input   clk_ctrl        ; //时钟输入，与 TFT 屏时钟保持一致
input   reset_n         ; //复位信号，低电平有效
input  [15:0] img_disp_hbegin; //待显示图片左上角第一个像素点在 TFT 屏的行向坐标
input  [15:0] img_disp_vbegin; //待显示图片左上角第一个像素点在 TFT 屏的场向坐标
input  [IMG_DATA_WIDTH-1:0] disp_back_color; //显示的背景颜色
output [ROM_ADDR_WIDTH-1:0] rom_addra; //读图片数据 ROM 地址
input  [IMG_DATA_WIDTH-1:0] rom_data; //读图片数据 ROM 数据
input  frame_begin; //一帧图像起始标识信号，clk_ctrl 时钟域
input  disp_data_req; //
input  [11:0] visible_hcount; //TFT 可见区域行扫描计数器
input  [11:0] visible_vcount; //TFT 可见区域场扫描计数器
output [IMG_DATA_WIDTH-1:0] disp_data; //待显示图片数据
reg   [ROM_ADDR_WIDTH-1:0]  rom_addra; //读图片数据 rom 地址
```

从上面的显示的示意图可以看出，图片显示起始位置 img_disp_hbegin 和 img_disp_vbegin 输入设置数值如果设置较大会导致实际显示区域比图片尺寸小，

仅能显示图片部分内容。为了区分正常显示完整图片和显示部分图片情况，首先对 img_disp_hbegin 和 img_disp_vbegin 设置值进行判断，判断当前设置值是属于哪种情况。

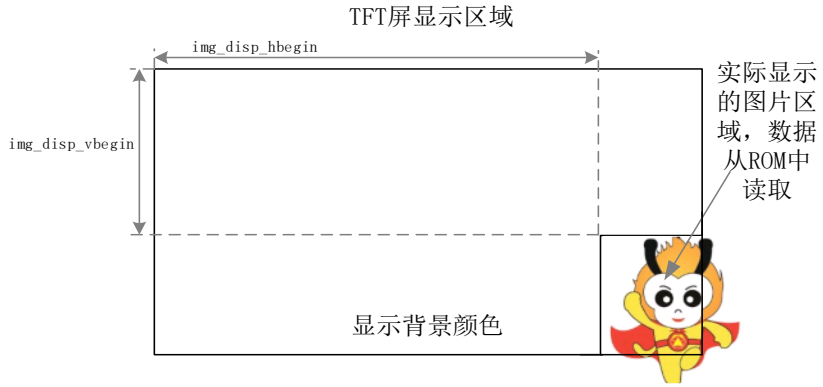


图 27-13 行起始和场起始信号设置时间过长导致的问题

判断条件分别为行起始设置值 img_disp_hbegin 加上图片的宽度是否大于 TFT 屏显示区域的宽度和列起始设置值 img_disp_vbegin 加上图片的高度是否大于 TFT 屏显示区域的高度，具体代码如下。

```
assign h_exceed = img_disp_hbegin + IMG_WIDTH > H_Visible_area -1'b1;
assign v_exceed = img_disp_vbegin + IMG_HEIGHT > V_Visible_area-1'b1;
```

其中，h_exceed 为高电平就是表示设置的行起始数值会导致图片行显示不全，同样的，v_exceed 为高电平就是表示设置的列起始数值会导致图片列显示不全。

当显示图片不正常（残缺）情况下，也就是 h_exceed 为高电平时，显示的行计数需满足(visible_hcount >= img_disp_hbegin && visible_hcount < H_Visible_area)，当正常显示图片情况下，显示的行计数满足(visible_hcount >= img_disp_hbegin && visible_hcount < img_disp_hbegin + IMG_WIDTH)，同理，显示的列计数也需要满足一定的条件，同时满足行计数显示区域和列计数显示区域的地方就是图片显示的区域。具体代码如下。img_disp 表示的就是最终图片显示的区域。

```
assign img_h_disp = h_exceed ? (visible_hcount >= img_disp_hbegin &&
visible_hcount < H_Visible_area):(visible_hcount >= img_disp_hbegin &&
visible_hcount < img_disp_hbegin + IMG_WIDTH);

assign img_v_disp = v_exceed ? (visible_vcount >= img_disp_vbegin &&
visible_vcount < V_Visible_area): (visible_vcount >= img_disp_vbegin
&& visible_vcount < img_disp_vbegin + IMG_HEIGHT);

assign img_disp = disp_data_req && img_h_disp && img_v_disp;
```

对 ROM 中图片数据的读的关键是产生读取 ROM 的地址。图片数据在 ROM 中存储的顺序如下图 27-14 所示。地址 0 存储的是图片数据的第 0 行的第 0 个像素点数据，地址 1 存储的是图片数据的第 0 行的第 1 个像素点数据，依次类推（注，这里的计数均是从 0 开始）。

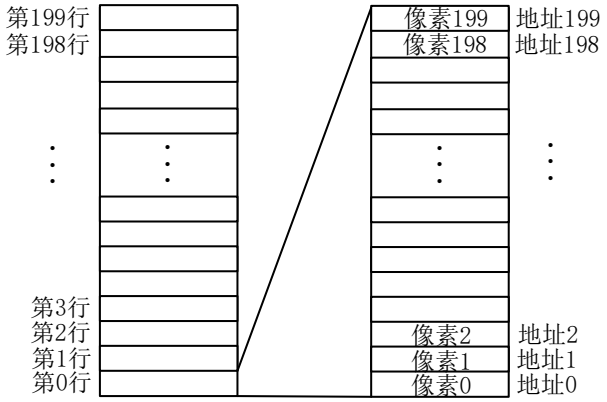


图 27-14 图片地址信息读取对应存储位置

将 ROM 中存储的图片数据在 TFT 屏上显示，就是将 ROM 中的图片数据从第 0 行开始一个一个读出来传给 TFT 屏驱动模块显示。图片显示区域的行计数最大值在 h_exceed 为高电平时，行计数最大值为 TFT 屏显示行计数最大值 H_Visible_area - 1'b1，其他情况计数最大值为 img_disp_hbegin + IMG_WIDTH - 1'b1，具体代码如下。

```
assign hcount_max h_exceed?(H_Visible_area-1'b1):(img_disp_hbegin+IMG_WIDTH-
1'b1);
```

ROM 的地址产生在显示区域内每一行内加 1，当到达显示区域行计数最大值后，地址换到 ROM 的下一行地址的起始，也就是加上 img_disp_hbegin + IMG_WIDTH - hcount_max，具体代码如下。

```
always@(posedge clk_ctrl or negedge reset_n)
 begin
   if(!reset_n)
     rom_addra <= 15'd0;
   else if(frame_begin)
     rom_addra <= 15'd0;
   else if(img_disp)
   begin
     if(visible_hcount == hcount_max)
       rom_addra<=rom_addra+(img_disp_hbegin+IMG_WIDTH-hcount_max);
     else
       rom_addra <= rom_addra + 1'b1;
   end
```

```
    else
      rom_addra <= rom_addra;
  end
```

读取 ROM 的地址产生后，就可以从 ROM 中读取到待显示的图片数据，采用二选一通过显示区域的标识信号 disp_data，将 ROM 读出数据送到显示区域，其他显示区域给设置的背景颜色。具体代码如下。

```
assign disp_data = img_disp ? rom_data : disp_back_color;
```

至此，image_extract 模块的设计初步完成，接下来进行模块的功能仿真验证模块功能的正确性。系统中的 TFT 屏显示驱动模块已经在前面进行了仿真和板级验证，系统中 PLL 和 ROM 模块采用的是 IP，不需单独进行仿真。这样就只剩下 image_extract 模块需要仿真，这里，由于系统并不复杂，仿真就直接在整个系统的仿真中进行验证，系统仿真中也能看到该模块的各个信号和验证功能的正确性。

27.4激励创建及仿真验证

在激励中，我们将直接例化顶层设计，为减少大家设计量，这里我们仅连接 TFT 相关端口。顶层设计的端口以及显示图片尺寸等参数设置的代码如下，这里设置图片显示区域在 TFT 屏的中间区域（起始位置是行计数 300，列计数 140），顶层里面详细代码参考提供工程源码。

```
module rom_image_tft_hdmi(
    clk50M,
    reset_n,
    TFT_rgb,
    TFT_hs,
    TFT_vs,
    TFT_clk,
    TFT_de,
    TFT_pwm,

  //hdmi1 interface
    hdmi1_clk_p   ,
    hdmi1_clk_n   ,
    hdmi1_dat_p   ,
    hdmi1_dat_n   ,
    hdmi1_oe
);
    input          clk50M;   //系统时钟输入，50M
    input          reset_n;  //复位信号输入，低有效
    output [15:0] TFT_rgb;   //TFT 数据输出
```

```
    output          TFT_hs;   //TFT 行同步信号
    output          TFT_vs;   //TFT 场同步信号
    output          TFT_clk;  //TFT 像素时钟
    output          TFT_de;   //TFT 数据使能
    output          TFT_pwm;  //TFT 背光控制

    //hdmi1 interface
    output           hdmi1_clk_p   ;
    output           hdmi1_clk_n   ;
    output [2:0]     hdmi1_dat_p   ;
    output [2:0]     hdmi1_dat_n   ;
    output           hdmi1_oe      ;

    //设置待显示图片尺寸，和存储图片 ROM 的地址位宽，显示背景颜色
    parameter DISP_IMAGE_W    = 200;
    parameter DISP_IMAGE_H    = 200;
    parameter ROM_ADDR_WIDTH  = 16;
    parameter DISP_BACK_COLOR = 16'hFFFF; //白色
    //设置屏幕尺寸
    parameter TFT_WIDTH  = 800;
    parameter TFT_HEIGHT = 480;
    //图片显示在屏幕中间位置
    parameter DISP_HBEGIN = (TFT_WIDTH  - DISP_IMAGE_W)/2;
    parameter DISP_VBEGIN = (TFT_HEIGHT - DISP_IMAGE_H)/2;
```

对于系统的仿真，也比较简单，只需要产生时钟和复位激励就行。在第 2 场图片显示开始的时候停止仿真，这样就可以看到完整的一场图片显示的仿真波形。具体 testbench 代码如下：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module rom_image_tft_tb();

    reg         clk50M;          //模块全局时钟输入，50M
    reg         reset_n;         //复位信号输入，低有效

    wire[15:0] TFT_rgb;          //TFT 数据输出
    wire       TFT_hs;           //TFT 行同步信号
    wire       TFT_vs;           //TFT 场同步信号
    wire       TFT_clk;          //TFT 像素时钟
    wire       TFT_de;           //TFT 数据使能
    wire       TFT_pwm;          //TFT 背光控制

    initial clk50M = 1;
    always#(`CLK_PERIOD/2) clk50M = ~clk50M;

    GSR GSR(.GSRI(1'b1));
```

```
    rom_image_tft_hdmi rom_image_tft_hdmi(
        .clk50M    (clk50M   ),
        .reset_n   (reset_n  ),

        .TFT_rgb   (TFT_rgb  ),
        .TFT_hs    (TFT_hs   ),
        .TFT_vs    (TFT_vs   ),
        .TFT_clk   (TFT_clk  ),
        .TFT_de    (TFT_de   ),
        .TFT_pwm   (TFT_pwm  ),
        .hdmi1_clk_p()  ,
        .hdmi1_clk_n()  ,
        .hdmi1_dat_p()  ,
        .hdmi1_dat_n()  ,
        .hdmi1_oe()
    );

    initial begin
        reset_n = 0;
        #(`CLK_PERIOD *20 +1);
        reset_n = 1;

    @(posedge rom_image_tft_hdmi.disp_driver.Frame_Begin);
    @(posedge rom_image_tft_hdmi.disp_driver.Frame_Begin);
    #200;
    $stop;
    end

endmodule
```

仿真 testbench 文件设计好后，打开 Modelsim 软件，新建仿真工程，配置仿真文件，首先查看 image_extract 模块信号，通过观察波形对 image_extract 模块功能的正确性进行验证和完善，如所示为 image_extract 模块信号仿真波形，这是 TFT 屏显示的一场图片的波形。

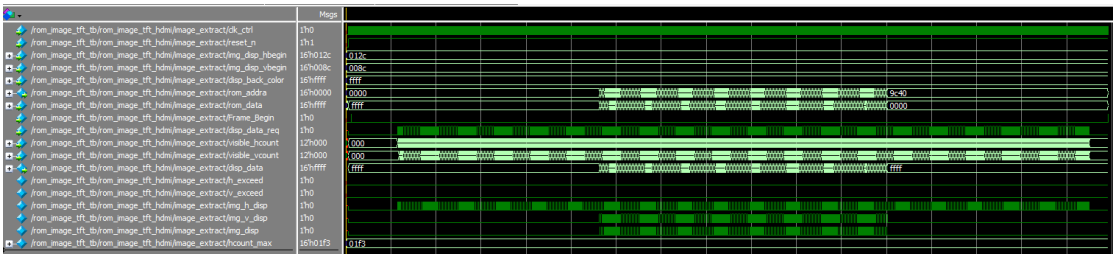


图 27-15 TFT 显示一场图片的仿真波形

对波形放大，如下图所示，可以看到图片显示区域在列计数为 140~339 范围（共 200 行）和行计数为 300~499 范围（1 行 200 个像素点），与图片尺寸一

致。ROM 地址在这个区域内每个像素时钟加 1，ROM 地址也是从 0 加到了 39999，也正好是图片数据存储的地址范围。仿真功能正常。

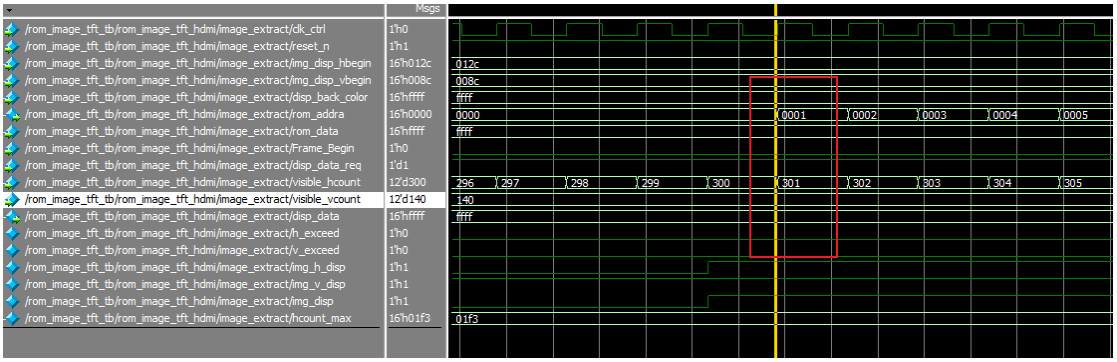


图 27-16 仿真中开始读 ROM 数据

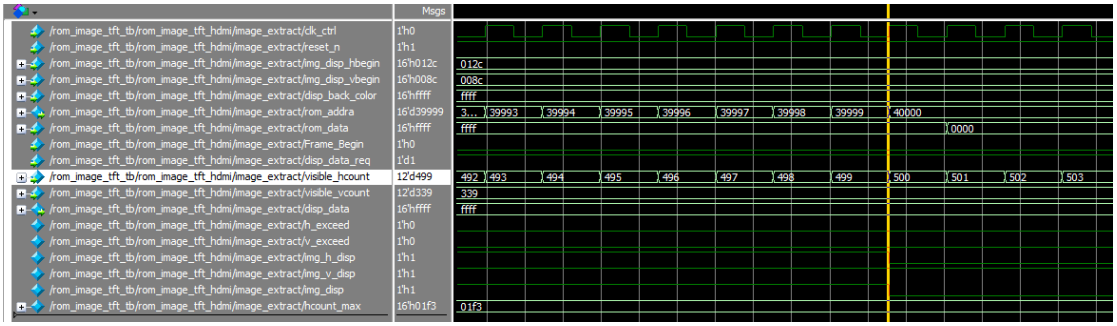


图 27-17 第 40000 个地址内数据读完仿真时序

在仿真没有问题后，对设计顶层进行板级验证。

27.5板级调试与验证

经过以上工作，代码设计部分的任务已经全部完成并完成了仿真测试。接下来进行板级验证。

本实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的 bit 文件下载到高云开发板。
2. 下载完成后能否实现让 TFT 显示屏和 HDMI 显示器显示储存在片上 ROM 中的芯路恒公司标识。
3. 标识的位置，是否在屏幕中央。

27.5.1系统所需硬件

1. 高云开发板。
2. 电源电缆一根。

3. 高云下载器一个。
4. 5 寸 TFT 显示屏一块
5. HDMI 线缆一根
6. 支持 HDMI 接口的显示器一台

27.5.2添加 I/O 约束

管脚分配表如下表 27-2 所示。约束完成之后如下

Pin Name	Pin NO.	Pin Name	Pin NO.
clk50M	T9	TFT_rgb[15]	H12
reset_n	B16	TFT_rgb[14]	G13
hdmi1_clk_n	M10	TFT_rgb[13]	F15
hdmi1_clk_p	N9	TFT_rgb[12]	F16
hdmi1_dat_n[0]	T7	TFT_rgb[11]	E18
hdmi1_dat_n[1]	V6	TFT_rgb[10]	L15
hdmi1_dat_n[2]	T5	TFT_rgb[9]	L16
hdmi1_dat_p[0]	R7	TFT_rgb[8]	L14
hdmi1_dat_p[1]	T6	TFT_rgb[7]	M13
hdmi1_dat_p[2]	R5	TFT_rgb[6]	J16
TFT_clk	M14	TFT_rgb[5]	H15
TFT_de	U18	TFT_rgb[4]	N14
TFT_pwm	U17	TFT_rgb[3]	N15
TFT_hs	T18	TFT_rgb[2]	N16
TFT_vs	T17	TFT_rgb[1]	M16
		TFT_rgb[0]	M18

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
10	TFT_rgb[14]	output		G13	2	False	LVCMOS33
11	TFT_rgb[15]	output		H12	2	False	LVCMOS33
12	TFT_rgb[1]	output		M16	3	False	LVCMOS33
13	TFT_rgb[2]	output		N16	3	False	LVCMOS33
14	TFT_rgb[3]	output		N15	3	False	LVCMOS33
15	TFT_rgb[4]	output		N14	3	False	LVCMOS33
16	TFT_rgb[5]	output		H15	2	False	LVCMOS33
17	TFT_rgb[6]	output		J16	3	False	LVCMOS33
18	TFT_rgb[7]	output		M13	3	False	LVCMOS33
19	TFT_rgb[8]	output		L14	3	False	LVCMOS33
20	TFT_rgb[9]	output		L16	3	False	LVCMOS33
21	TFT_vs	output		T17	3	False	LVCMOS33
22	clk50M	input		T9	4	False	LVCMOS33
23	hdmi1_clk_n	output		N9	5	False	LVCMOS33
24	hdmi1_clk_p	output		M10	5	False	LVCMOS33
25	hdmi1_dat_n[0]	output		T7	5	False	LVCMOS33
26	hdmi1_dat_n[1]	output		V6	5	False	LVCMOS33
27	hdmi1_dat_n[2]	output		T5	5	False	LVCMOS33
28	hdmi1_dat_p[0]	output		R7	5	False	LVCMOS33
29	hdmi1_dat_p[1]	output		T6	5	False	LVCMOS33
30	hdmi1_dat_p[2]	output		R5	5	False	LVCMOS33
31	reset_n	input		B16	1	False	LVCMOS33

图 27-18 所示。

表 27-2 管脚绑定表

Pin Name	Pin NO.	Pin Name	Pin NO.
clk50M	T9	TFT_rgb[15]	H12
reset_n	B16	TFT_rgb[14]	G13
hdmi1_clk_n	M10	TFT_rgb[13]	F15
hdmi1_clk_p	N9	TFT_rgb[12]	F16
hdmi1_dat_n[0]	T7	TFT_rgb[11]	E18
hdmi1_dat_n[1]	V6	TFT_rgb[10]	L15
hdmi1_dat_n[2]	T5	TFT_rgb[9]	L16
hdmi1_dat_p[0]	R7	TFT_rgb[8]	L14
hdmi1_dat_p[1]	T6	TFT_rgb[7]	M13
hdmi1_dat_p[2]	R5	TFT_rgb[6]	J16
TFT_clk	M14	TFT_rgb[5]	H15
TFT_de	U18	TFT_rgb[4]	N14
TFT_pwm	U17	TFT_rgb[3]	N15
TFT_hs	T18	TFT_rgb[2]	N16
TFT_vs	T17	TFT_rgb[1]	M16
		TFT_rgb[0]	M18

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
10	TFT_rgb[14]	output		G13	2	False	LVCMOS33
11	TFT_rgb[15]	output		H12	2	False	LVCMOS33
12	TFT_rgb[1]	output		M16	3	False	LVCMOS33
13	TFT_rgb[2]	output		N16	3	False	LVCMOS33
14	TFT_rgb[3]	output		N15	3	False	LVCMOS33
15	TFT_rgb[4]	output		N14	3	False	LVCMOS33
16	TFT_rgb[5]	output		H15	2	False	LVCMOS33
17	TFT_rgb[6]	output		J16	3	False	LVCMOS33
18	TFT_rgb[7]	output		M13	3	False	LVCMOS33
19	TFT_rgb[8]	output		L14	3	False	LVCMOS33
20	TFT_rgb[9]	output		L16	3	False	LVCMOS33
21	TFT_vs	output		T17	3	False	LVCMOS33
22	clk50M	input		T9	4	False	LVCMOS33
23	hdmi1_clk_n	output		N9	5	False	LVCMOS33
24	hdmi1_clk_p	output		M10	5	False	LVCMOS33
25	hdmi1_dat_n[0]	output		T7	5	False	LVCMOS33
26	hdmi1_dat_n[1]	output		V6	5	False	LVCMOS33
27	hdmi1_dat_n[2]	output		T5	5	False	LVCMOS33
28	hdmi1_dat_p[0]	output		R7	5	False	LVCMOS33
29	hdmi1_dat_p[1]	output		T6	5	False	LVCMOS33
30	hdmi1_dat_p[2]	output		R5	5	False	LVCMOS33
31	reset_n	input		B16	1	False	LVCMOS33

图 27-18 管脚绑定效果图

管脚绑定完成后，将程序进行综合与实现，生成 bit 文件。确认 bit 生成完成并无误后，接下来我们开始连接硬件。

27.5.3硬件连接

本次实验的硬件连接图如下图 27-19 所示。

图 27-19 硬件连接图

27.5.4板级验证

将前面我们生成好的 bit 烧录到开发板中，可以看到，如下图 27-20、图 27-21 所示的显示效果。

图 27-20 TFT 显示效果

图 27-21 HDMI 显示屏显示效果

可以看到，bit 被成功下载，logo 成功显示在了 TFT 显示屏和 HDMI 显示器的正中央。所显示的 logo 整体正常，没有出现数据丢失导致的断层等现象，且颜色层次鲜明清晰。至此，本次设计成功，设计能够正确且稳定的实现预期功能。

27.6常见问题分析

对于本节工程需要注意的有以下几点：

1. 使用 BMP2Mif 软件，选择的输入图片数据格式必须为 BMP 格式，否则无法转换成功，输出为 RGB565 格式。
2. 本实验会充分运用并考验 FPGA 的片上存储资源大小。前面我们反复强调在本节的实验中，图片的存储尺寸不能过大，否则无法正常显示。与本节实验形成鲜明对比的是：在 AC620 开发上，因为使用相同的存储图片会使 FPGA 的片上存储资源使用量超标，所以在 AC620 开发板使用 Quartus II 生成的相关工程中，我们均是采用处理后缩小的图片进行的演示。希望读者能够通过本实验，深入理解并关注片上存储资源消耗量。

27.7总结

本节实验在前面章节学习了 HDMI 和 TFT 显示屏的显示原理基础上，进一步学习了使用 FPGA 内部存储器存储一幅小型图片的实验案例，以此来理解 FPGA 片上存储器存储图片并通过 TFT 显示图片的原理。

28 基于 HDMI 和 TFT 显示屏的静态字符显示

工程源码	|----02_设计实例 |----ch28_rom_char_tft_hdmi
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

TFT 显示屏作为显示设备，不仅可以显示图片，还可以显示字符文字。上一节讲解实现了基于 TFT 显示屏的图片显示系统，本节将在此基础上对系统中部分模块做更换修改，完成基于 TFT 显示屏和 HDMI 显示器的字符显示系统的实现。

28.1系统整体设计

系统整体设计框图如下图 28-1 所示。

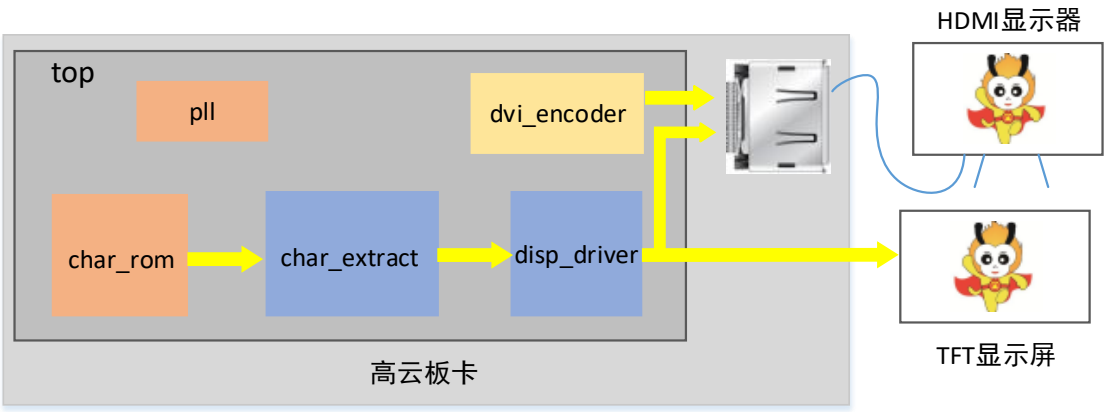


图 28-1 TFT/HDMI 显示屏静态字符显示系统整体设计结构框图

整个系统框架与“基于 HDMI 和 TFT 显示屏的图片显示”类似，主要包括 5 个模块部分，模块功能描述如下：

1. PLL 模块：产生系统内各模块工作所需的时钟，直接使用 IP。系统内模块均工作在 33MHz（模块工作时钟均采用与 TFT 屏驱动时钟）下，高云开发板 FPGA 管脚输入时钟为 50MHz，这样 PLL 的输入时钟为 50MHz，输出时钟为 33MHz。
2. disp_driver 模块：TFT 屏显示驱动，上一节已经详细讲解了实现过程，这里就不多说。
3. char_rom 模块：用来存储待显示的字符数据，直接使用 ROM IP。具体

配置和字符数据的生成后面详细介绍。

4. char_extract 模块：作为 char_rom 模块和 disp_driver 模块之间的桥梁。用来根据 TFT 屏驱动显示控制器的行列扫描位置信息去存储图片数据 ROM 中提取相应的字符数据在 TFT 屏上显示，具体设计后面详细介绍。
5. dvi_encoder 模块：HDMI 驱动模块，用于驱动 HDMI 显示屏显示图像数据。

28.2字符数据的产生及 ROM 的配置

本实验是将字符点阵数据存储在 ROM 中，关于 ROM IP 的配置使用前面已经有章节讲过了。将数据存放在 ROM 首先需要产生 ROM 初始化的数据文件。这里借助如下图 28-2 所示的工具软件将字符转成点阵数据然后产生 ROM 初始化数据文件。相关软件可在提供资料的网盘进行下载。

图 28-2 字模软件工具文件夹图标

在文件夹内找到“PCtoLCD.exe”，双击打开软件，软件如下图 28-3 所示。

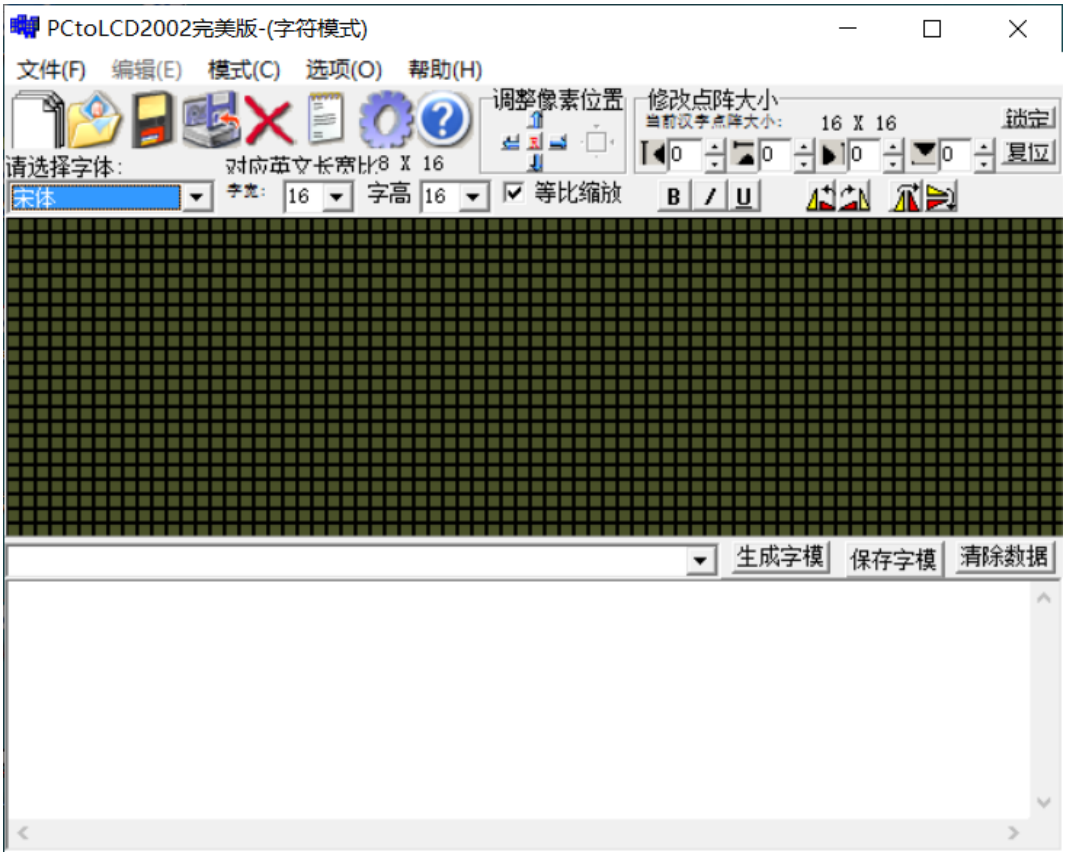


图 28-3 字模软件界面

可以在界面工具栏上模式选择字符模式或图形模式，默认是字符模式，对于本实验是实现字符的显示，因此这里保持默认即可。

图 28-4 字模软件字符模式选择

下面以“Hello FPGA!”字符串为例生成用于 TFT 屏显示的数据。

1. 点击界面工具栏上选项进行设置，具体设置如下图 28-5 所示。这里设置中文文字的点阵大小为 32*32，对应的英文字符点阵大小为 16*32。这样英文字符的点阵中 1 行就是 16 个点，也就是 2 个字节数据，为了便于产生 ROM 初始化数据文件，将每行显示数据的点阵设置为 2。如果是中文文字就设置为 4。

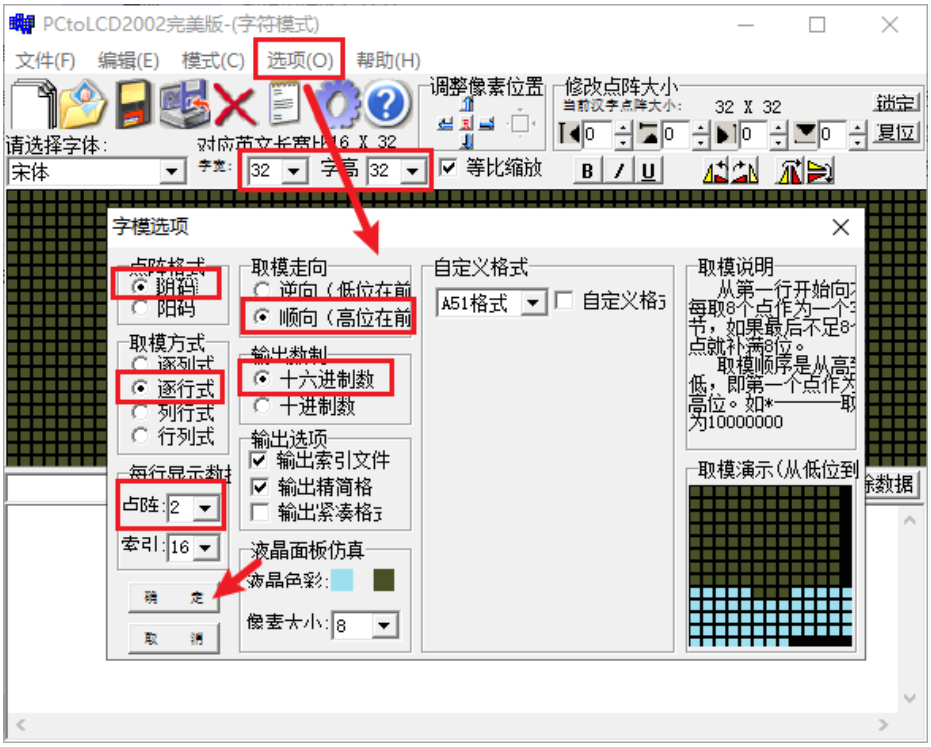


图 28-5 取模软件输入本章内容字模配置

2. 设置完后，在界面下方文件输出地方输入“Hello FPGA!”，然后点击生成字模。这样在软件下面窗口就会生成对应的字模数据。

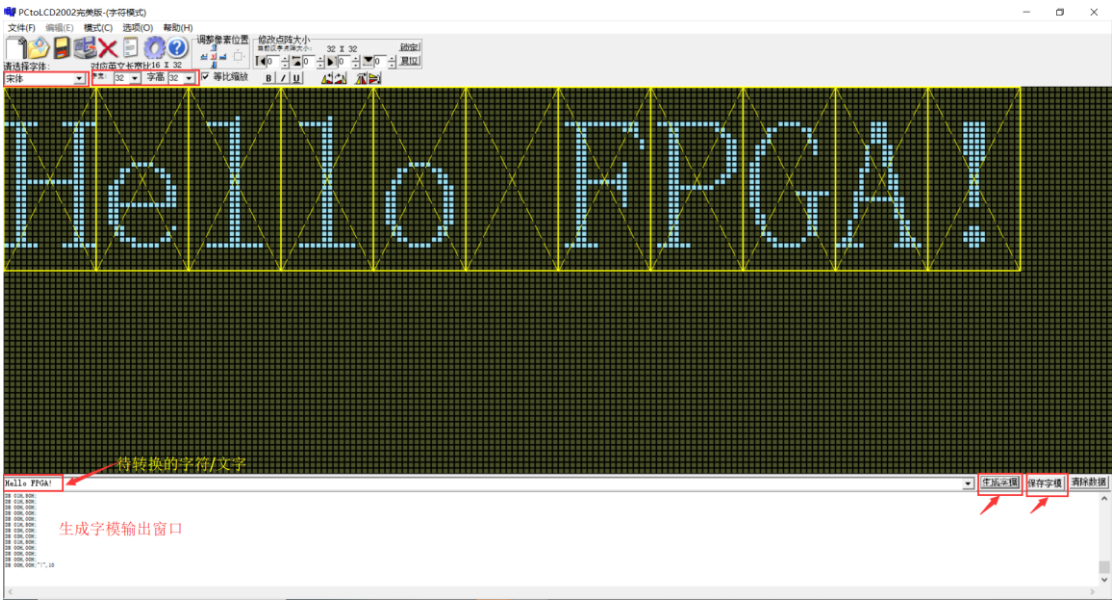


图 28-6 生成字模操作

3. 点击保存字模，可将生成字模存放在指定路径和设置文件名。

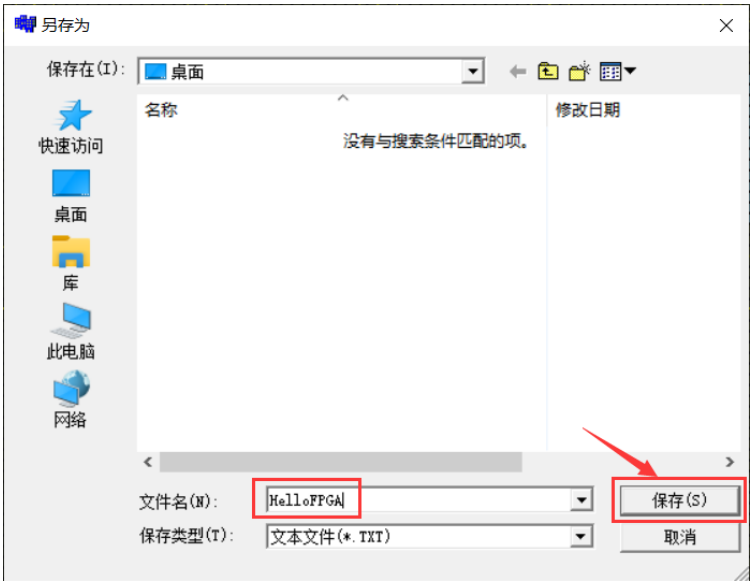


图 28-7 指定字模存放路径

4. 保存完后，将会在指定文件夹路径下生成两个文本文件，分别为“HelloFPGA_index.TXT”和“HelloFPGA.TXT”。

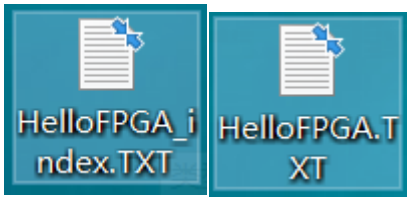


图 28-8 生成字模的文本文件图标

5. 其中 HelloFPGA.TXT 文件是生成的字符点阵数据，该文件还不能直接作为 ROM 初始化数据文件，还需要使用编辑器做一定的处理。

```
 H(0) e(1) l(2) l(3) o(4)  (5) F(6) P(7) G(8) A(9) !(10)

DB 00H,00H;
DB 00H,00H;
DB 00H,00H;
DB 00H,00H;
DB 00H,00H;
DB 00H,00H;
DB FCH,3FH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 3FH,FCH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB 30H,0CH;
DB FCH,3FH;
DB 00H,00H;
DB 00H,00H;
DB 00H,00H;
DB 00H,00H;"H",0

DB 00H,00H;
```

图 28-9 需处理的字符点阵数据文件内容

6. 对数据文件头部添加 ROM 数据初始化文件的关键字指明数据的进制和数据部分。采用编辑器的列处理将数据文件处理成如下。

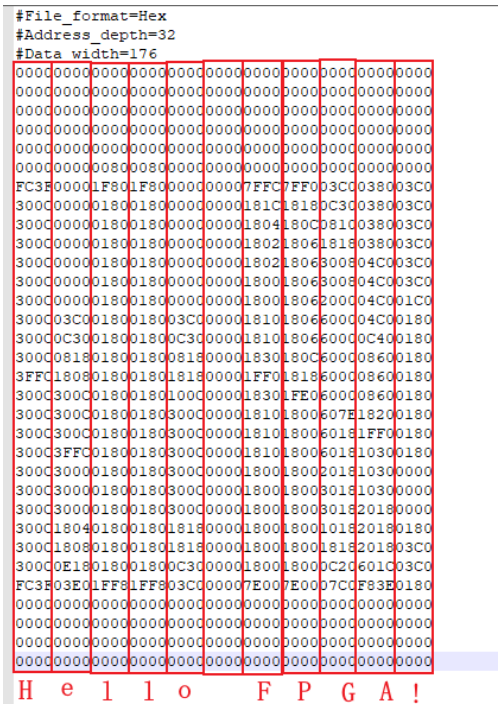


图 28-10 处理后效果图

7. 处理完后的数据文件还需要将文件的后缀更改为.mi 格式，如下图 28-11 是更改后的文件，该文件在工程目录下的 src\ char_init 有提供，

可直接用 notepad++等编辑器直接打开查看。

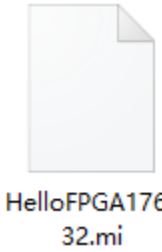


图 28-11 更改处理后文件格式为 mi

8. 从处理后的数据可以看到 1 个数据的位宽是 176 位，共有 32 个数据，这样从后面的 ROM IP 设置数据位宽就设置位 176，深度设置为 32。在 ROM 初始化数据生成完后，就对 ROM IP 进行配置，具体 ROM 的配置如下图 28-12 所示。

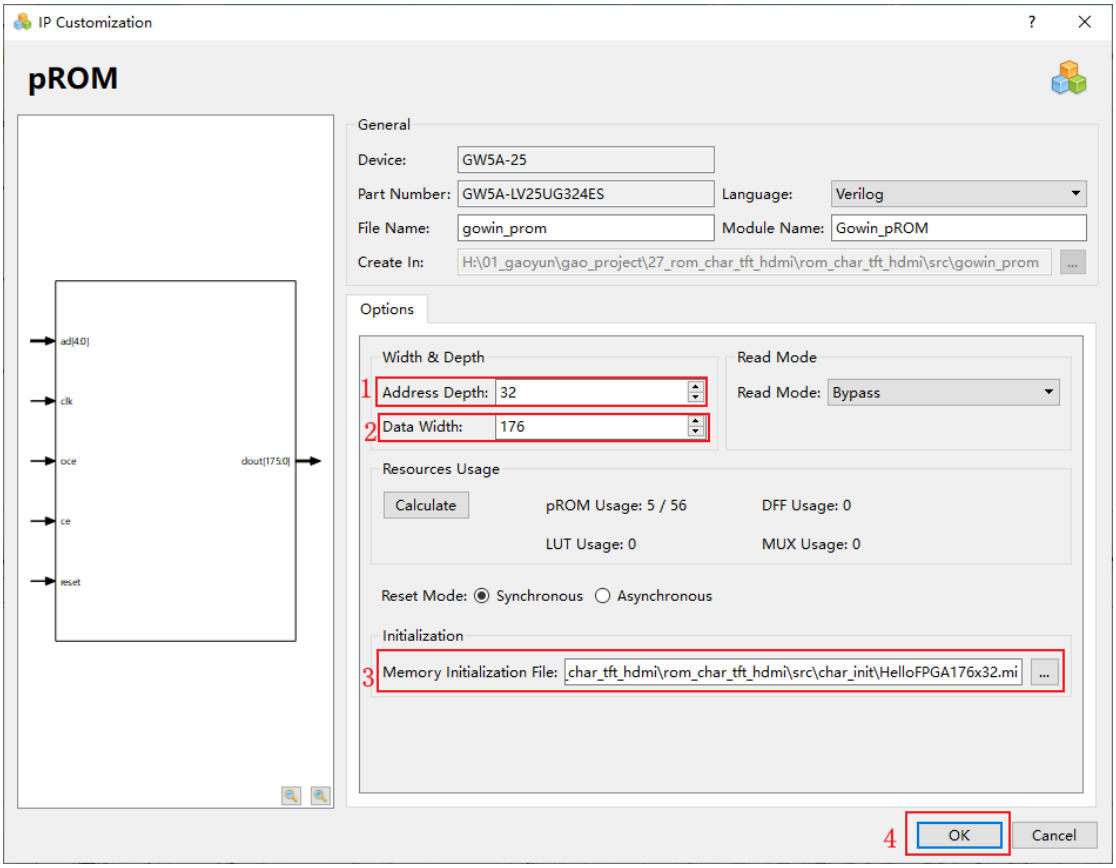


图 28-12 ROM IP 配置

28.3字符提取 char_extract 模块的设计

char_extract 模块的功能是根据 TFT 屏驱动模块当前显示行列位置等信号控制产生读取 ROM 数据的地址，从 ROM 中读出需要显示的字符数据传给 TFT 屏

显示驱动模块进行字符的显示。模块接口及接口功能描述如下所示：

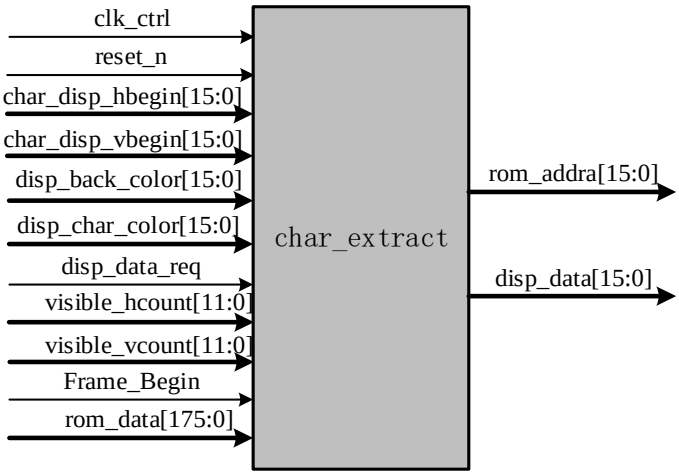


图 28-13 字符提取模块端口列表

表 28-1 字符提取模块接口功能描述

端口名	I/O	端口功能
clk_ctrl	I	模块工作时钟
reset_n	I	模块复位信号，低电平复位
char_disp_hbegin[15:0]	I	待显示图片起始像素点显示在屏幕行位置设置
char_disp_vbegin[15:0]	I	待显示图片起始像素点显示在屏幕列位置设置
disp_back_color[15:0]	I	屏幕显示的背景颜色，
disp_char_color[15:0]		
disp_data_req	I	屏幕可见显示区标识信号，用于作为模块请求显示数据的标识信号
visible_hcount[15:0]	I	图像区行扫描地址
visible_vcount[15:0]	I	图像区场扫描地址
Frame_Begin	I	场信号的开始标志信号，用来控制模块即将显示新一帧图像
rom_data[15:0]	I	从 ROM 读取的字符信息数据
disp_data[15:0]	O	传给 TFT 屏显示的数据
rom_addra[175:0]	O	从 ROM 读取的字符数据的地址

关于 char_disp_hbegin、char_disp_vbegin、disp_back_color 和 disp_char_color 等输入信号对应控制显示的的示意图如下图 28-14 所示。

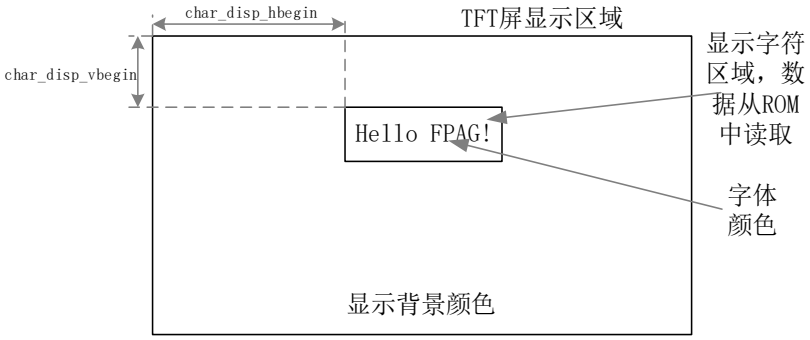


图 28-14 输入信号对应显示示意图

接下来就开始对模块代码进行设计，考虑到模块的通用性和移植性（比如移植到 4.3 寸屏幕或更改图片尺寸大小等），将 TFT 屏显示显示区域的大小、字符显示区域显示的大小、TFT 屏显示数据的位宽（这样就可以方便移植到 RGB888 等其他图像数据格式的显示）等使用 parameter 进行参数化定义。这样将其移植到其他系统上时，只需在例化模块的时候将参数进行相应的修改即可。参数化定义的代码如下。

```
//设置待显示字符尺寸，和存储字符 ROM 的地址位宽
parameter CHAR_WIDTH        = 16; //单个字符显示宽度
parameter CHAR_HEIGHT       = 32; //单个字符显示高度
parameter ROW_DISP_CHAR_NUM = 11; //一行显示字符个数
parameter COL_DISP_CHAR_NUM = 1 ; //显示字符行数
parameter CHAR_ROM_ADDR_W   = 5 ; //存储字符 ROM 地址位宽，
log2(CHAR_HEIGHT * COL_DISP_CHAR_NUM)
parameter DISP_BACK_COLOR = 16'hFFFF; //背景白色
parameter DISP_CHAR_COLOR = 16'h0000; //显示字符黑色
//设置 TFT 屏幕尺寸
parameter TFT_WIDTH  = 800;
parameter TFT_HEIGHT = 480;
//显示字符串的总的像素点宽度和高度
localparam DISP_CHAR_TOTAL_W = CHAR_WIDTH  * ROW_DISP_CHAR_NUM;
localparam DISP_CHAR_TOTAL_H = CHAR_HEIGHT * COL_DISP_CHAR_NUM;
//字符显示在屏幕中间位置
localparam DISP_HBEGIN = (TFT_WIDTH  - DISP_CHAR_TOTAL_W)/2;
localparam DISP_VBEGIN = (TFT_HEIGHT - DISP_CHAR_TOTAL_H)/2;
```

这样在输入输出端口或内部变量信号进行定义时就可以使用上面参数，具体代码如下。

```
//时钟输入，与 TFT 屏时钟保持一致
input                           clk_ctrl;
//复位信号，低电平有效
input                           reset_n;
//待显示图片左上角第一个像素点在 TFT 屏的行向坐标
input  [15:0]                   char_disp_hbegin;
//待显示图片左上角第一个像素点在 TFT 屏的场向坐标
input  [15:0]                   char_disp_vbegin;
input  [DISP_DATA_W-1:0]        disp_back_color;  //显示的背景颜色
input  [DISP_DATA_W-1:0]        disp_char_color;  //显示字符的颜色
input                 Frame_Begin;//一帧图像起始标识信号，clk_ctrl 时钟域
output[CHAR_ROM_ADDR_W-1:0]     rom_addra; //读字符数据 rom 地址
input [DISP_CHAR_TOTAL_W-1:0] rom_data;         //读出字符数据
input                           disp_data_req;    //
input  [11:0]               visible_hcount;  //TFT 可见区域行扫描计数器
input  [11:0]               visible_vcount;  //TFT 可见区域场扫描计数器
output [DISP_DATA_W-1:0]        disp_data;        //待显示数据
```

从上面的显示的示意图可以看出，图片显示起始位置 char_disp_hbegin 和 char_disp_vbegin 输入设置数值如果设置较大会导致实际显示区域比待显示字符尺寸小，仅能显示字符部分内容。为了区分正常显示完整字符和显示部分字符情况，首先对 char_disp_hbegin 和 char_disp_vbegin 设置值进行判断，判断当前设置值是属于哪种情况。

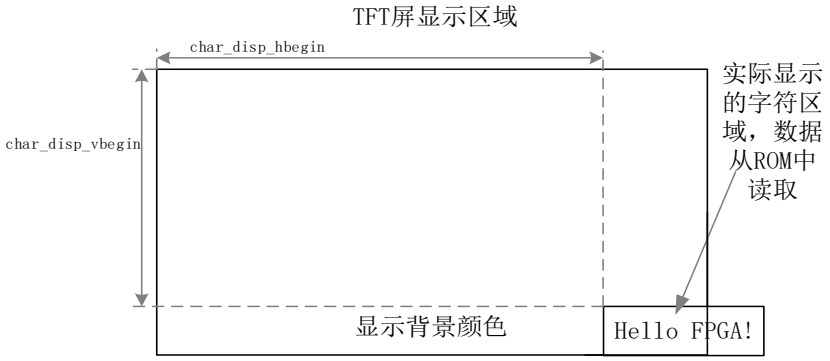


图 28-15 行起始和场起始相对于实际字符显示区域关系

判断条件分别为行起始设置值 char_disp_hbegin 加上图片的宽度是否大于 TFT 屏显示区域的宽度和列起始设置值 char_disp_vbegin 加上图片的高度是否大于 TFT 屏显示区域的高度，具体代码如下。

```
//判断设置的显示的起始位置是否会导致显示超出范围
assign h_exceed = char_disp_hbegin + DISP_CHAR_TOTAL_W > H_Visible_area - 1'b1;
assign v_exceed = char_disp_vbegin + DISP_CHAR_TOTAL_H > V_Visible_area - 1'b1;
```

其中，h_exceed 为高电平就是表示设置的行起始数值会导致字符右边部分显示不全，同样的，v_exceed 为高电平就是表示设置的列起始数值会导致字符下边部分显示不全。

当显示字符不正常（残缺）情况下，也就是 h_exceed 为高电平时，显示的行计数需满足(visible_hcount >= img_disp_hbegin && visible_hcount < H_Visible_area)，当正常显示图片情况下，显示的行计数满足(visible_hcount >= img_disp_hbegin && visible_hcount < img_disp_hbegin + IMG_WIDTH)，同理，显示的列计数也需要满足一定的条件，同时满足行计数显示区域和列计数显示区域的地方就是实际字符显示的区域。具体代码如下。char_disp 表示的就是最终图片显示的区域。

```
//判断设置的显示的起始位置是否会导致显示超出范围
assign h_exceed = char_disp_hbegin + DISP_CHAR_TOTAL_W >
H_Visible_area - 1'b1;
```

```
assign v_exceed = char_disp_vbegin + DISP_CHAR_TOTAL_H >
V_Visible_area - 1'b1;//不同的设置情况，显示区域做不同的处理
assign char_h_disp = h_exceed ?
(visible_hcount >=char_disp_hbegin&&
visible_hcount<H_Visible_area):
(visible_hcount >= char_disp_hbegin && visible_hcount <
char_disp_hbegin + DISP_CHAR_TOTAL_W);

assign char_v_disp = v_exceed ?
(visible_vcount >= char_disp_vbegin && visible_vcount<V_Visible_area):
(visible_vcount >= char_disp_vbegin && visible_vcount <
char_disp_vbegin + DISP_CHAR_TOTAL_H);

assign char_disp = disp_data_req && char_h_disp && char_v_disp;
```

对 ROM 中字符数据的读的关键是产生读取 ROM 的地址。字符数据在 ROM 中存储的顺序如下图 28-16 所示。地址 0 存储的是字符点阵的第 0 行数据，地址 1 存储的是字符点阵的第 1 行数据，依次类推（注，这里的计数均是从 0 开始）。

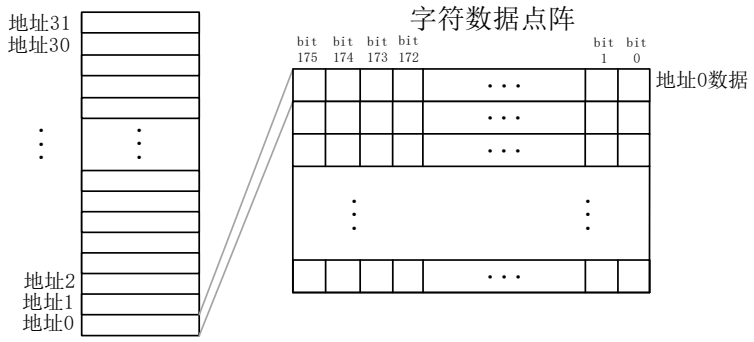


图 28-16 字符数据点阵和地址映射关系

将 ROM 中存储的图片数据在 TFT 屏上显示，就是将 ROM 中的字符点阵数据读出来传给 TFT 屏驱动模块显示。图片显示区域的行计数最大值在 h_exceed 为高电平时，行计数最大值为 TFT 屏显示行计数最大值 H_Visible_area - 1'b1，其他情况计数最大值为 img_disp_hbegin + IMG_WIDTH - 1'b1，具体代码如下。

```
assign hcount_max = h_exceed ?
(H_Visible_area-1'b1):(char_disp_hbegin + DISP_CHAR_TOTAL_W -1'b1);
```

ROM 的地址产生在字符显示区域内当到达显示区域行计数最大值后，地址加 1，具体代码如下。

```
always@(posedge clk_ctrl or negedge reset_n)
begin
if(!reset_n)
    rom_addra <= 'd0;
else if(Frame_Begin)
    rom_addra <= 'd0;
```

```
else if((visible_hcount == hcount_max) && char_disp)
    rom_addra <= rom_addra + 1'b1;
else
    rom_addra <= rom_addra;
end
```

读取 ROM 的地址产生后，就可以从 ROM 中读取到待显示的字符一行的点阵数据，再得到字符一行的点阵数据后，在字符显示区域内，每一行中的每个像素点数据根据点阵对应 bit 位为 1 还是为 0，采用二选一通过显示区域的标识信号 disp_data，输出对应的像素数据。如果为显示字符颜色，为 0 显示背景颜色。显示区域中每行中第 0 个像素点的点阵的数值是读出 ROM 数据的 bit175，第 1 个像素点的点阵数值是读出 ROM 数据的 bit174，依次类推。这样就采用移位的方式，在显示区域内，每个像素时钟将读出的数据左移 1 位。具体代码如下。

```
always@(posedge clk_ctrl or negedge reset_n)
  begin
    if(!reset_n)
      char_flag <= 'd0;
    else if(char_disp)
      char_flag <= {char_flag[DISP_CHAR_TOTAL_W-2:0],1'b0};
    else
      char_flag <= rom_data;
  end

assign disp_data = (char_disp && char_flag[DISP_CHAR_TOTAL_W-1]) ?
disp_char_color : disp_back_color;
```

至此，char_extract 模块的设计初步完成，接下来进行模块的功能仿真验证模块功能的正确性。系统中的 TFT 屏显示驱动模块已经在前面进行了仿真和板级验证，HDMI 初始化模块也在前面章节中得到了验证。系统中 PLL 和 ROM 模块采用的是 IP，不需单独进行仿真。这样就只剩下 char_extract 模块需要仿真，这里，由于系统并不复杂，仿真就直接在整个系统的仿真中进行验证，系统仿真中也能看到该模块的各个信号和验证功能的正确性。

28.4激励创建和仿真测试

同上一章一样，本章的激励文件中，我们直接例化顶层，然后只连接其中的 TFT 相关接口即可。顶层设计的端口以及显示图片尺寸等参数设置的代码如下，这里设置图片显示区域在 TFT 屏的中间区域（起始位置是行计数 312，列计数 224），顶层里面详细代码参考提供工程源码。

```
module rom_char_tft_hdmi(
    clk50M,
    reset_n,

    TFT_rgb,
    TFT_hs,
    TFT_vs,
    TFT_clk,
    TFT_de,
    TFT_pwm,

      //hdmi1 interface
    hdmi1_clk_p   ,
    hdmi1_clk_n   ,
    hdmi1_dat_p   ,
    hdmi1_dat_n   ,
    hdmi1_oe
);

    input          clk50M;   //系统时钟输入，50M
    input          reset_n;  //复位信号输入，低有效

    output [15:0] TFT_rgb;  //TFT 数据输出
    output         TFT_hs;   //TFT 行同步信号
    output         TFT_vs;   //TFT 场同步信号
    output         TFT_clk;  //TFT 像素时钟
    output         TFT_de;   //TFT 数据使能
    output         TFT_pwm;  //TFT 背光控制

   //hdmi1 interface
    output           hdmi1_clk_p   ;
    output           hdmi1_clk_n   ;
    output [2:0]     hdmi1_dat_p   ;
    output [2:0]     hdmi1_dat_n   ;
    output           hdmi1_oe      ;

    //设置待显示字符尺寸，和存储字符 ROM 的地址位宽
    parameter CHAR_WIDTH        = 16; //单个字符显示宽度
    parameter CHAR_HEIGHT       = 32; //单个字符显示高度
    parameter ROW_DISP_CHAR_NUM = 11; //一行显示字符个数
    parameter COL_DISP_CHAR_NUM = 1 ; //显示字符行数
    parameter CHAR_ROM_ADDR_W   = 5 ;
    parameter DISP_BACK_COLOR = 16'hFFFF; //背景白色
    parameter DISP_CHAR_COLOR = 16'h0000; //显示字符黑色
    //设置 TFT 屏幕尺寸
    parameter TFT_WIDTH  = 800;
    parameter TFT_HEIGHT = 480;
```

```
    //显示字符串的总的像素点宽度和高度
    localparam DISP_CHAR_TOTAL_W = CHAR_WIDTH  * ROW_DISP_CHAR_NUM;
    localparam DISP_CHAR_TOTAL_H = CHAR_HEIGHT * COL_DISP_CHAR_NUM;
    //字符显示在屏幕中间位置
    localparam DISP_HBEGIN = (TFT_WIDTH  - DISP_CHAR_TOTAL_W)/2;
    localparam DISP_VBEGIN = (TFT_HEIGHT - DISP_CHAR_TOTAL_H)/2;
```

对于系统的仿真，也比较简单，只需要产生时钟和复位激励就行。在 TFT 屏第 2 场图片显示开始的时候停止仿真，这样就可以看到完整的一场图片显示的仿真波形。具体 testbench 代码如下。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module rom_char_tft_tb();

    reg         clk50M;         //模块全局时钟输入，50M
    reg         reset_n;        //复位信号输入，低有效

    wire[15:0] TFT_rgb;         //TFT 数据输出
    wire       TFT_hs;          //TFT 行同步信号
    wire       TFT_vs;          //TFT 场同步信号
    wire       TFT_clk;         //TFT 像素时钟
    wire       TFT_de;          //TFT 数据使能
    wire       TFT_pwm;         //TFT 背光控制

    initial clk50M = 1;
    always#(`CLK_PERIOD/2) clk50M = ~clk50M;

    GSR GSR(.GSRI(1'b1));

    rom_char_tft_hdmi rom_char_tft_hdmi(
        .clk50M    (clk50M   ),
        .reset_n   (reset_n  ),

        .TFT_rgb   (TFT_rgb  ),
        .TFT_hs    (TFT_hs   ),
        .TFT_vs    (TFT_vs   ),
        .TFT_clk   (TFT_clk  ),
        .TFT_de    (TFT_de   ),
        .TFT_pwm   (TFT_pwm  ),
        .hdmi1_clk_p()  ,
        .hdmi1_clk_n()  ,
        .hdmi1_dat_p()  ,
        .hdmi1_dat_n()  ,
        .hdmi1_oe()
    );
```

```
    initial begin
        reset_n = 0;
        #(`CLK_PERIOD *20 +1);
        reset_n = 1;

    @(posedge rom_char_tft_hdmi.disp_driver.Frame_Begin);
    @(posedge rom_char_tft_hdmi.disp_driver.Frame_Begin);
    #200;
    $stop;
    end

endmodule
```

仿真 testbench 文件设计好后，打开 Modelsim 新建仿真工程，配置仿真环境，将 char_extract 模块信号，通过观察波形对 char_extract 模块功能的正确性进行验证和完善。如图 28-17 所示为 char_extract 模块信号仿真波形，这时 TFT 屏显示的一场图片的波形。红色圈主的地方就是字符显示区域的地方。

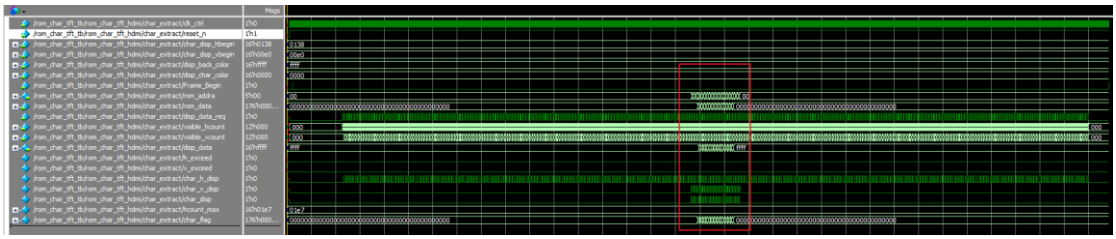


图 28-17 TFT 显示屏的静态字符显示仿真全图

对波形放大，如下所示。可以看到字符显示区域在列计数为 224~255 范围（共 32 行）和行计数为 312~487 范围（字符显示区域内 1 行 176 个像素点），与字符尺寸一致。

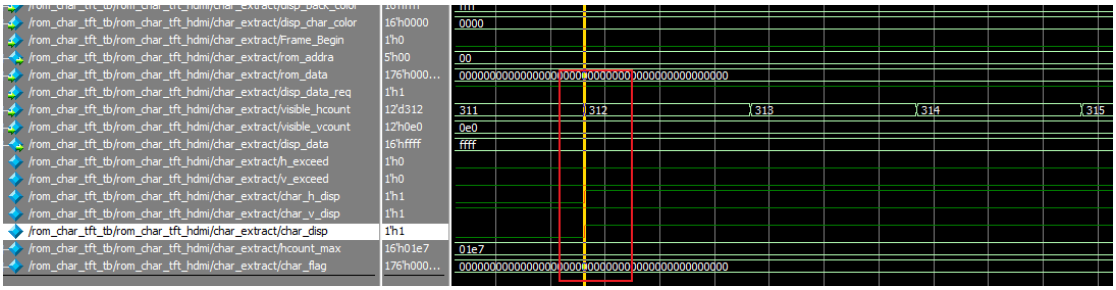


图 28-18 TFT 显示屏的静态字符显示仿真局部放大图 1（字符显示起始）

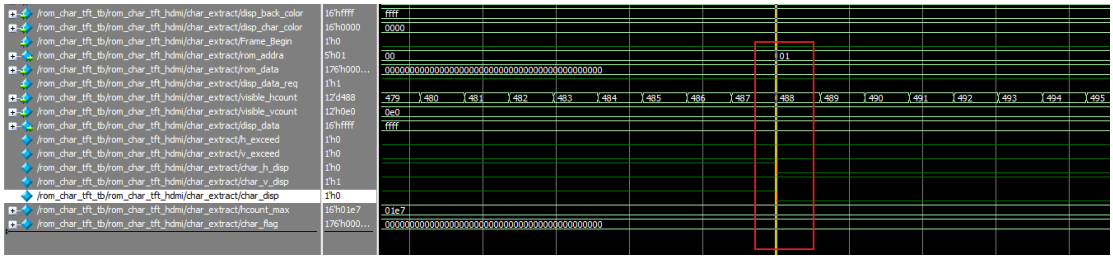


图 28-19 TFT 显示屏的静态字符显示仿真局部放大图 2（字符显示截止）

ROM 地址在字符显示区域每到达显示的最大行计数加 1，ROM 地址也是从 0 加到了 31，也正好是图片数据存储的地址范围。仿真功能正常。

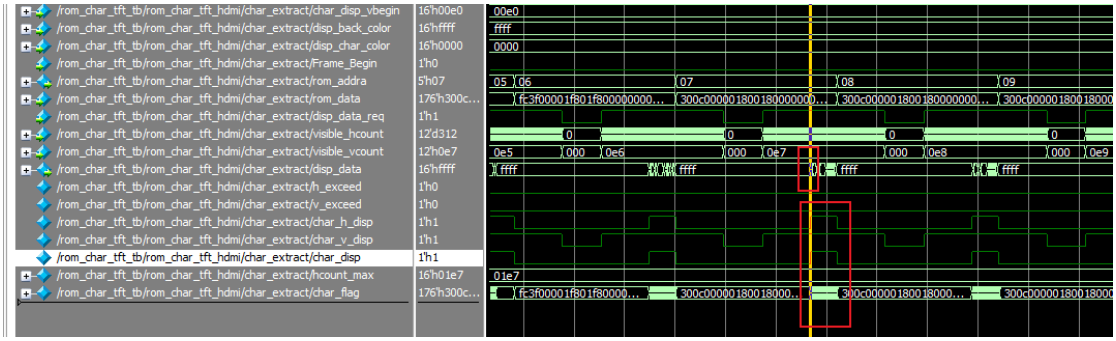


图 28-20 字符显示其中 4 行数据局部仿真效果图

在仿真没有问题后，对设计顶层进行板级验证。

28.5板级调试与验证

经过以上工作，代码设计部分的任务已经全部完成并完成了仿真测试。接下来进行板级验证。

本实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的 bit 文件下载到高云开发板。
2. 下载完成后能否实现让 TFT 显示屏和 HDMI 显示器显示储存在片上 ROM 中的字符 “Hello FPGA”字样。
3. 显示的字符位置，是否位于屏幕中央，颜色是否如程序设计。

28.5.1系统所需硬件

1. 高云开发板
2. 电源电缆一根
3. 高云下载器一个

4. 5 寸 TFT 显示屏一块
5. HDMI 线缆一根
6. 支持 HDMI 接口显示器一台

28.5.2添加 I/O 约束

管脚约束表如下表 28-2 所示，绑定成功之后效果图如下

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
10	TFT_rgb[14]	output		G13	2	False	LVCMOS33
11	TFT_rgb[15]	output		H12	2	False	LVCMOS33
12	TFT_rgb[1]	output		M16	3	False	LVCMOS33
13	TFT_rgb[2]	output		N16	3	False	LVCMOS33
14	TFT_rgb[3]	output		N15	3	False	LVCMOS33
15	TFT_rgb[4]	output		N14	3	False	LVCMOS33
16	TFT_rgb[5]	output		H15	2	False	LVCMOS33
17	TFT_rgb[6]	output		J16	3	False	LVCMOS33
18	TFT_rgb[7]	output		M13	3	False	LVCMOS33
19	TFT_rgb[8]	output		L14	3	False	LVCMOS33
20	TFT_rgb[9]	output		L16	3	False	LVCMOS33
21	TFT_vs	output		T17	3	False	LVCMOS33
22	clk50M	input		T9	4	False	LVCMOS33
23	hdmi1_clk_n	output		N9	5	False	LVCMOS33
24	hdmi1_clk_p	output		M10	5	False	LVCMOS33
25	hdmi1_dat_n[0]	output		T7	5	False	LVCMOS33
26	hdmi1_dat_n[1]	output		V6	5	False	LVCMOS33
27	hdmi1_dat_n[2]	output		T5	5	False	LVCMOS33
28	hdmi1_dat_p[0]	output		R7	5	False	LVCMOS33
29	hdmi1_dat_p[1]	output		T6	5	False	LVCMOS33
30	hdmi1_dat_p[2]	output		R5	5	False	LVCMOS33
31	reset_n	input		B16	1	False	LVCMOS33

图 28-21 所示

表 28-2 管脚绑定表

Pin Name	Pin NO.	Pin Name	Pin NO.
clk50M	T9	TFT_rgb[15]	H12
reset_n	B16	TFT_rgb[14]	G13
hdmi1_clk_n	M10	TFT_rgb[13]	F15
hdmi1_clk_p	N9	TFT_rgb[12]	F16
hdmi1_dat_n[0]	T7	TFT_rgb[11]	E18
hdmi1_dat_n[1]	V6	TFT_rgb[10]	L15
hdmi1_dat_n[2]	T5	TFT_rgb[9]	L16
hdmi1_dat_p[0]	R7	TFT_rgb[8]	L14
hdmi1_dat_p[1]	T6	TFT_rgb[7]	M13
hdmi1_dat_p[2]	R5	TFT_rgb[6]	J16
TFT_clk	M14	TFT_rgb[5]	H15
TFT_de	U18	TFT_rgb[4]	N14
TFT_pwm	U17	TFT_rgb[3]	N15
TFT_hs	T18	TFT_rgb[2]	N16
TFT_vs	T17	TFT_rgb[1]	M16
		TFT_rgb[0]	M18

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
10	TFT_rgb[14]	output		G13	2	False	LVCMOS33
11	TFT_rgb[15]	output		H12	2	False	LVCMOS33
12	TFT_rgb[1]	output		M16	3	False	LVCMOS33
13	TFT_rgb[2]	output		N16	3	False	LVCMOS33
14	TFT_rgb[3]	output		N15	3	False	LVCMOS33
15	TFT_rgb[4]	output		N14	3	False	LVCMOS33
16	TFT_rgb[5]	output		H15	2	False	LVCMOS33
17	TFT_rgb[6]	output		J16	3	False	LVCMOS33
18	TFT_rgb[7]	output		M13	3	False	LVCMOS33
19	TFT_rgb[8]	output		L14	3	False	LVCMOS33
20	TFT_rgb[9]	output		L16	3	False	LVCMOS33
21	TFT_vs	output		T17	3	False	LVCMOS33
22	clk50M	input		T9	4	False	LVCMOS33
23	hdmi1_clk_n	output		N9	5	False	LVCMOS33
24	hdmi1_clk_p	output		M10	5	False	LVCMOS33
25	hdmi1_dat_n[0]	output		T7	5	False	LVCMOS33
26	hdmi1_dat_n[1]	output		V6	5	False	LVCMOS33
27	hdmi1_dat_n[2]	output		T5	5	False	LVCMOS33
28	hdmi1_dat_p[0]	output		R7	5	False	LVCMOS33
29	hdmi1_dat_p[1]	output		T6	5	False	LVCMOS33
30	hdmi1_dat_p[2]	output		R5	5	False	LVCMOS33
31	reset_n	input		B16	1	False	LVCMOS33

图 28-21 管脚绑定完成之后效果图

板级验证的显示效果图如下。

图 28-22 TFT 屏显示效果图

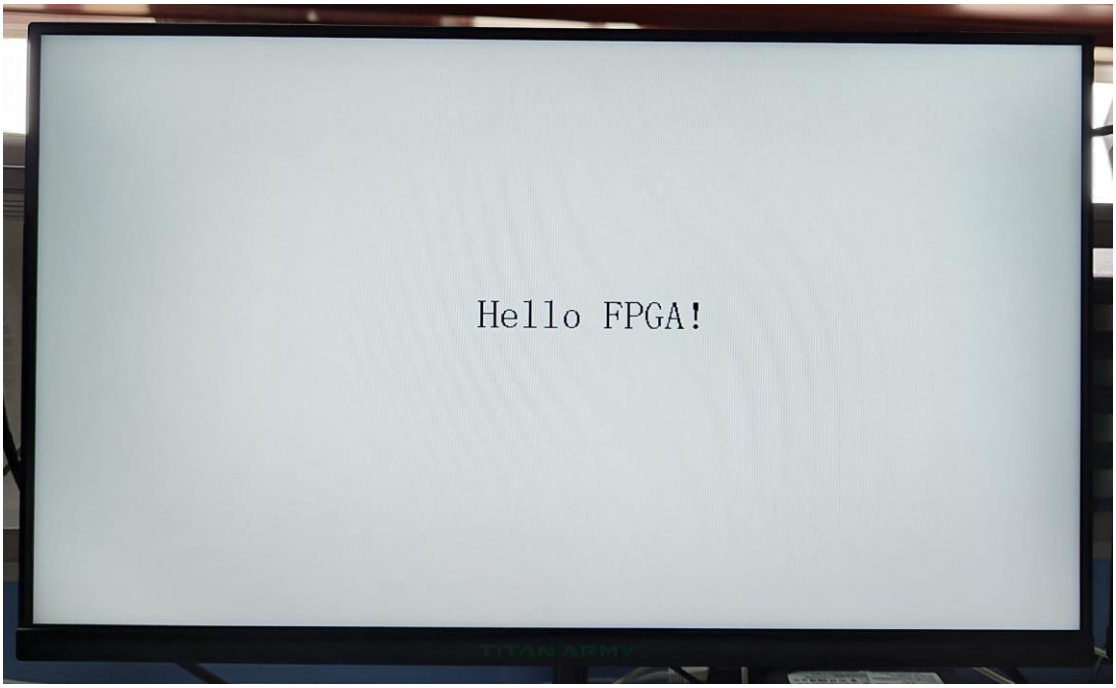


图 28-23 HDMI 显示效果图

可以看到，字符成功在 HDMI 和 TFT 上显示。至此，就完成了 TFT 屏显示字符的功能。上面在 TFT 上显示的是英文字符，同样也是可以显示中文的，在工程目录下 src\char_init 提供有 ROM 初始化数据文件“HelloFPGA224x64.mi”，这里的数据文件是包括英文和中文。使用的时候需要对设计顶层代码的部分参数进行修改。对于提供的“HelloFPGA224x64.mi”是显示两行字符，一行最多是 14 个英文字符（1 个中文算 2 个英文字符）。具体参数修改如下。

```
//设置待显示字符尺寸，和存储字符 ROM 的地址位宽
parameter CHAR_WIDTH        = 16; //单个字符显示宽度
parameter CHAR_HEIGHT       = 32; //单个字符显示高度
parameter ROW_DISP_CHAR_NUM = 14; //一行显示字符个数
parameter COL_DISP_CHAR_NUM = 2 ; //显示字符行数
parameter CHAR_ROM_ADDR_W   = 6 ; //存储字符 ROM 地址位宽
```

除此之外，还需要对 ROM IP 配置中的数据位宽和深度做一定的修改。在进行中文字模转换的时候，为了方便使用编辑器修改转换成 ROM 初始化文件，将每行显示数据下的点阵参数改为 4。

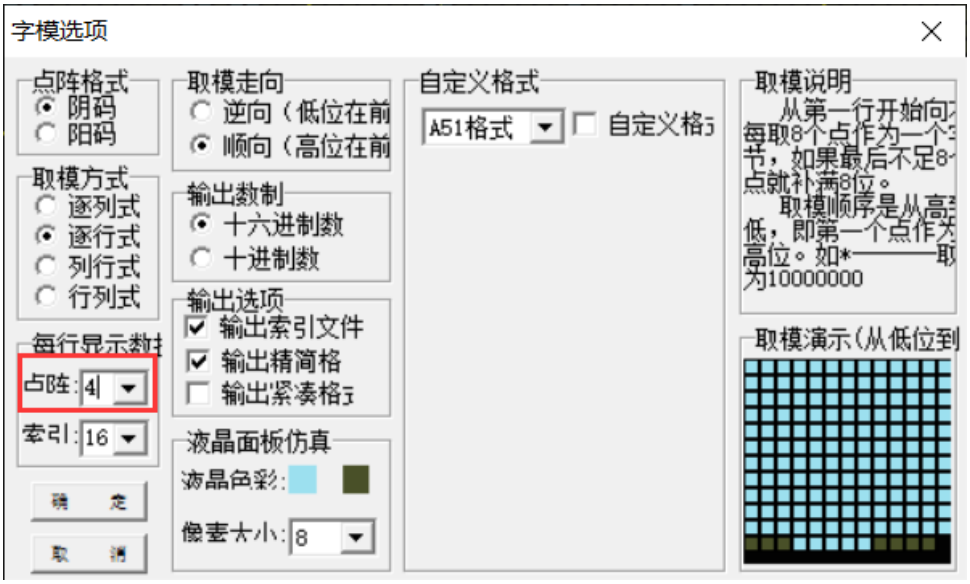


图 28-24 修改点阵参数

下图是“小梅哥”和“芯路恒”取模的界面。

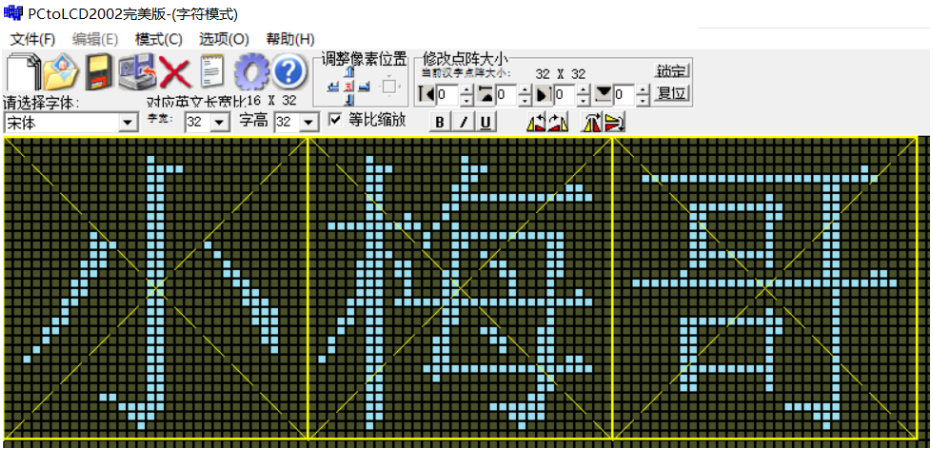


图 28-25 输入中文汉字“小梅哥”测试

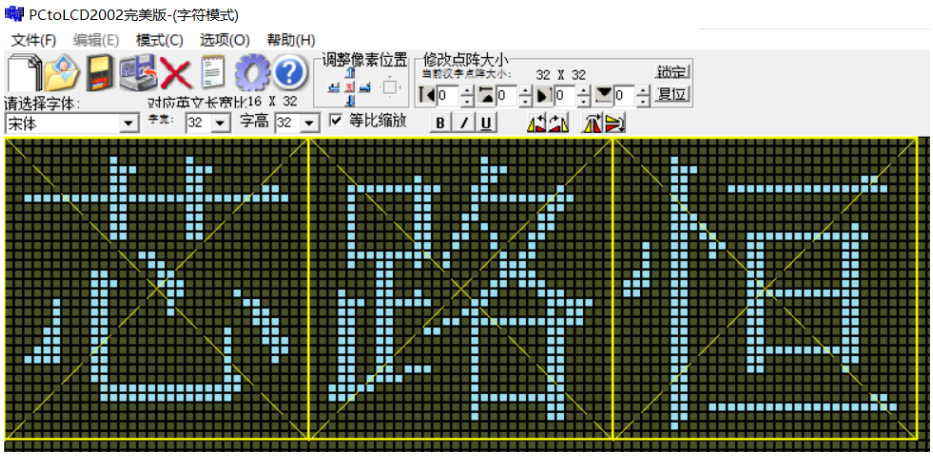


图 28-26 输入中文汉字“芯路恒”测试

图 28-27 和图 28-28 是中文和英文混合形式的显示效果图。

图 28-27 TFT 屏显示中文和英文混合字符效果图

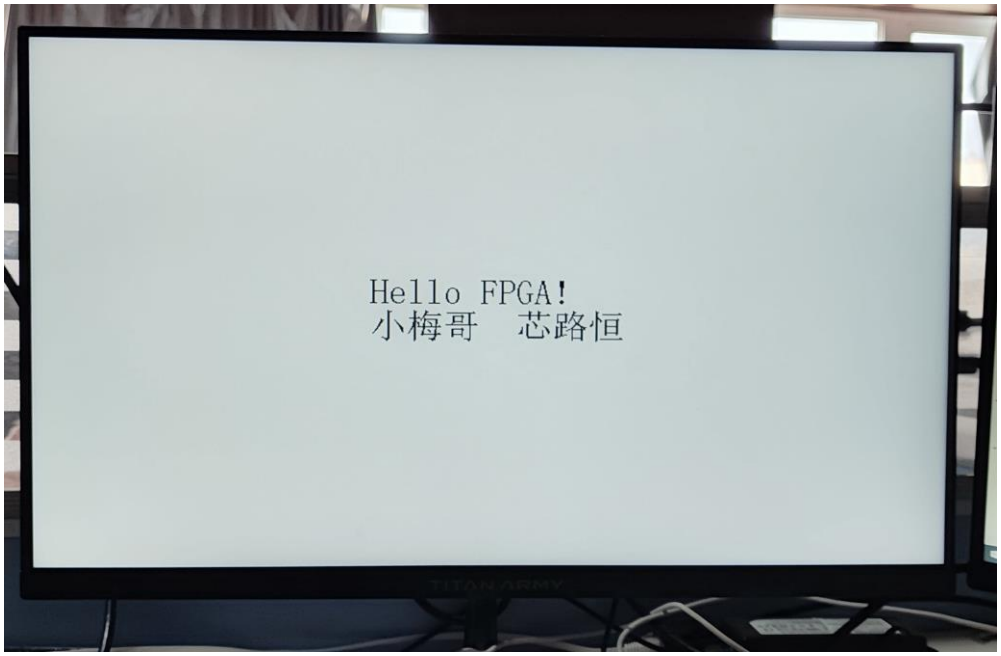


图 28-28 HDMI 显示效果图

28.6常见问题分析

对于本工程需要注意的有以下几点：

1. 在更换显示内容后，需要分别在顶层文件和 IP 参数选择项目中，修改和显示区域对应的有关参数以及存储信息。

2. 使用字模软件生成的 TXT 文件，并不可以被 ROM 的 IP 核直接识别，需要通过文本编辑软件将该 TXT 文件调整到位宽和地址信息与 mi 文件的标准格式一致。

28.7总结

本节实验在上一节学习了 HDMI 和 TFT 显示屏的图像显示原理基础上，进一步学习了使用 FPGA 内部存储器存储字符数据的实验案例，以此来理解 FPGA 片上存储器存储字符并通过 TFT 显示字符的原理。建议读者能够跟随本实验内容，完整的进行整个实验。

29 I²C 接口控制器设计与验证

工程源码	|----02_设计实例 |----ch29_i2c_control
相关视频课程	
说明	

章节导读

I2C(Inter-Integrated Circuit BUS) 集成电路总线，它是一种串行通信总线，使用多主从架构，由飞利浦公司在 1980 年代为了让主板、嵌入式系统或手机用以连接低速周边设备而发展。一般用在小数据量场合使用，传输距离短。

29.1I²C 协议解析

在物理层上，I²C 接口只需要两条总线线路，即 SCL（串行时钟线）、SDA（串行数据线），I²C 总线是半双工，所以任意时刻只能有一个主机。每个连接在总线上的 I²C 器件都有一个唯一的器件地址，在通信的时候就是靠这个地址来通信的。传输速率标准模式下可以达到 100kb/s，快速模式下可以达到 400kb/s，高速模式下可达 3.4Mbit/s。总线上的主设备与从设备之间以字节(8 位)为单位进行双向的数据传输。

总线上的每一个设备都可以作为主设备或者从设备，而且每一个设备都会对应一个唯一的地址(可以从 I²C 器件数据手册得知)，主从设备之间就是通过这个地址来确定与哪个器件进行通信。

I²C 协议规定：

- 在时钟（SCL）为高电平的时候，数据总线（SDA）必须保持稳定，所以数据总线（SDA）在时钟（SCL）为低电平的时候才能改变。

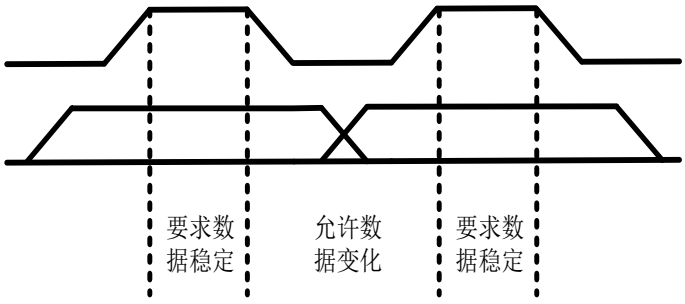


图 29-1 I²C 协议数据变化时序要求

- 在时钟（SCL）为高电平的时候，数据总线（SDA）由高到低的跳变为总线**起始信号**，在时钟（SCL）为高电平的时候，数据总线（SDA）由低到高的跳变为总线**停止信号**。

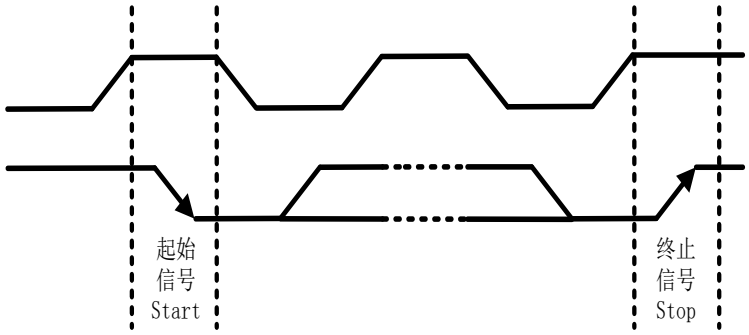


图 29-2 I²C 协议起始信号和终止信号时序

- 应答，当 IIC 主机（不一定是发送端还是接受端）将 8 位数据或命令传出后，会将数据总线（SDA）释放，即设置为输入，然后等待从机应答（低电平 0 表示应答，1 表示非应答），此时的时钟仍然是主机提供的。

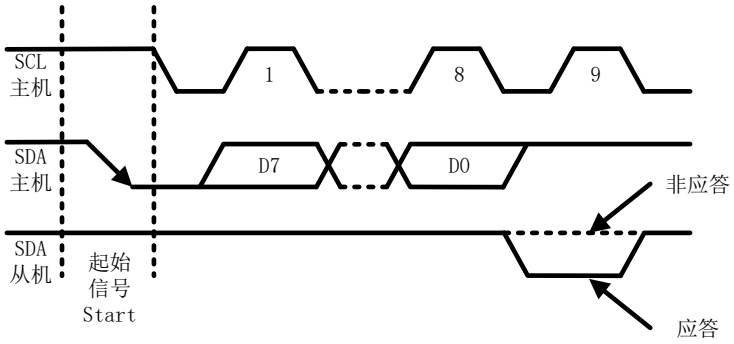


图 29-3 从机应答信号时序

数据帧格式，I²C 器件通讯的时候首先是要发送“起始信号”，紧跟着就是七位器件地址，第八位是数据传送方向位（0：代表写，1：代表读），再后面就是等待从机的应答。当然传送结束后，“终止信号”也是由主机来产生的。发送数据的时候是高位先发送。

29.2I²C 器件地址

每个 I²C 器件都有一个器件地址，有的器件地址在出厂时地址就设置好了，用户不可以更改（例如 OV7670 器件地址为固定的 0x42），有的确定了几位，剩下几位由硬件确定（比如常见的 I²C 接口的 EEPROM 存储器，留有 3 个控地址

的引脚，由用户自己在硬件设计时确定）。

严格讲，主机不是直接向从机发送地址，而是主机往总线上发送地址，所有的从机都能接收到主机发出的地址，然后每个从机都将主机发出的地址与自己的地址比较，如果匹配上了，这个从机就会向总线发出一个响应信号。主机收到响应信号后，开始向总线上发送数据，与这个从机的通讯就建立起来了。如果主机没有收到响应信号，则表示寻址失败。

通常情况下，主从器件的角色是确定的，也就是说从机一直工作在从机模式。不同器件定义地址的方式是不同的，有的是软件定义，有的是硬件定义。例如某些单片机的 I²C 接口作为从机时，其器件地址是可以通过软件修改从机地址寄存器确定的。而对于一些其他器件，如 CMOS 图像传感器、EEPROM 存储器，其器件地址在出厂时就已经完全或部分设定好了，具体情况可以在对应器件的数据手册中查到。

对于 AT24C64 这样一颗 EEPROM 器件，其器件地址为 1010 加 3 位的片选信号。3 位片选信号由硬件连接设定。例如 SOIC 封装的该芯片 PIN1、PIN2、PIN3 为片选地址引脚。当硬件电路上分别将这三个引脚连接到 GND 或 VCC 时，就可以设置不同的片选地址。

I²C 协议在进行数据传输时，主机需要首先向总线上发出控制命令，其中控制命令就包含了从机器件地址和读写控制，然后等待从机响应。如图 29-4 所示为 I²C 控制命令传输的数据格式。

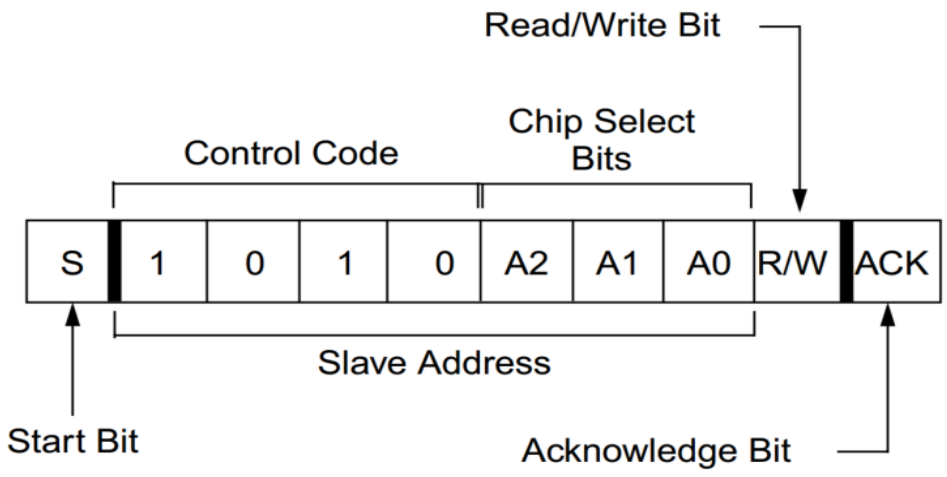


图 29-4 I²C 协议控制命令传输的数据格式

I²C 传输时，按照从高到低的位序进行传输。控制字节的最低位为读写控制位，当该位为 0 时表示主机对从机进行写操作，当该位为 1 时表示主机对从机进行读操作。例如，当需要对片选地址为 3’b100 的 AT24LC64 发起写操作，

则控制字节应该为 1010_100_0。若进行读操作，则控制字节应该为 1010_100_1。

29.3I²C 存储器地址

每个支持 I²C 协议的器件，内部总会有一些可供读写的寄存器或存储器，例如，EEPROM 存储器，内部就是顺序编址的一系列存储单元；型号为 OV7670 的 CMOS 摄像头（OV7670 的该接口叫 SCCB 接口，其实质也是一种特殊的 I²C 协议，可以直接兼容 I²C 协议），其内部就是一系列编址的可供读写的寄存器。因此，我们要对一个器件中的存储单元（寄存器和存储器，以下简称存储单元）进行读写，就必须要能够指定存储单元的地址。I²C 协议设计了有从机存储单元寻址地址段，该地址段为一个字节或两个字节长度，在主机确认收到从机返回的控制字节响应后由主机发出。地址段长度视不同的器件类型，长度不同，例如同是 EEPROM 存储器，AT24C04 的址段长度为一个字节，而 AT24C64 的地址段长度为两个字节。具体是一个字节还是两个字节，与器件的存储单元数量有关。如下图 29-5、图 29-6 分别为 1 字节地址和 2 字节地址器件的地址分布图，其中 1 字节地址的器件是以内存为 1kbit 的 EEPROM 存储器 AT24C01 举例，2 字节地址的器件是以内存为 64kbit 的 EEPROM 存储器 AT24C64 举例的。

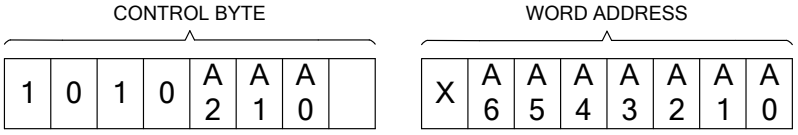


图 29-5 1 字节地址器件的地址分布

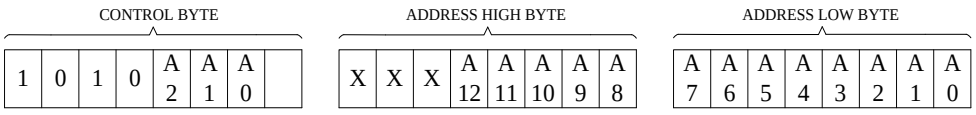


图 29-6 2 字节地址器件的地址分布

29.4I²C 写时序

29.4.1单字节写时序

根据前面讲的，不同器件，I²C 器件地址字节不同，这样对于 I²C 单字节写时序就会有所差别，下面两幅图分别为 1 字节地址段器件和 2 字节地址段器件单字节写时序图。

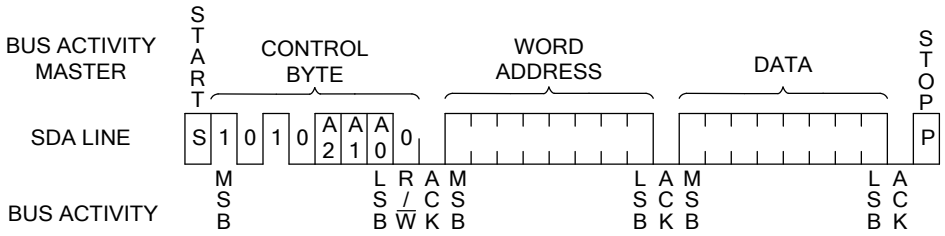


图 29-7 1 字节地址器件单字节写操作时序

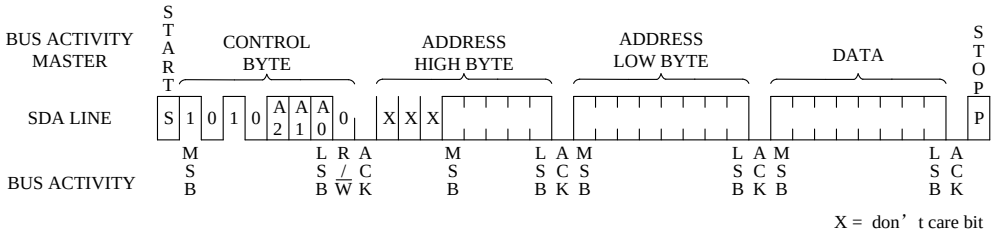


图 29-8 2 字节地址器件单字节写操作时序

根据时序图，从主机角度来描述一次写入单字节数据过程如下：

单字节地址写单字节数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输 1 字节地址数据；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输待写入的数据；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功，主机产生 STOP 位，终止传输。

双字节地址写单字节数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据高字节；

6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据低字节；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功，主机设置 SDA 为输出，传输待写入的数据；
10. 设置 SDA 为三态门输入，读取从机应答信号；
11. 读取应答信号成功，主机产生 STOP 位，终止传输。

29.4.2连续写时序（页写时序）

注：I^2C 连续写时序仅部分器件支持。

连续写（也称页写，需要注意的是 I^2C 连续写时序仅部分器件支持）是主机连续写多个字节数据到从机，这个和单字节写操作类似，连续多字节写操作也是分为 1 字节地址段器件和 2 字节地址段器件的写操作，如下图 29-9、图 29-10 分别为 1 字节地址段器件和 2 字节地址段器件连续多字节写时序图。

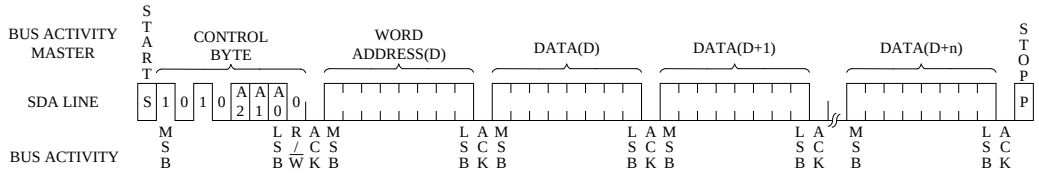


图 29-9 1 字节器件连续多字节写操作时序

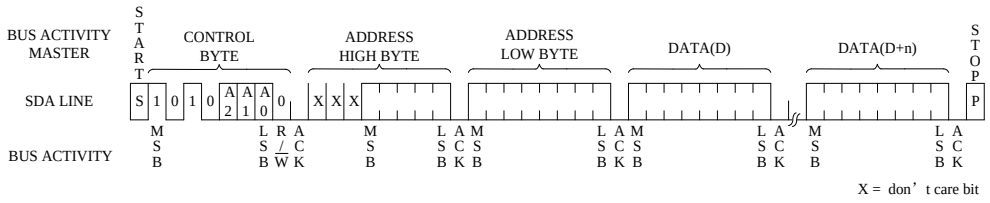


图 29-10 2 字节地址器件连续多字节写操作时序

根据时序图，从主机角度来描述一次写入多字节数据过程如下：

单字节地址器件连续多字节写数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输 1 字节地址数据；

6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输待写入的第 1 个数据；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功后，主机设置 SDA 为输出，传输待写入的下一个数据；
10. 设置 SDA 为三态门输入，读取从机应答信号；n 个数据被写完，转到步骤 11，若数据未被写完，转到步骤 9；
11. 读取应答信号成功，主机产生 STOP 位，终止传输。

双字节地址器件连续多字节写数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据高字节；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据低字节；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功，主机设置 SDA 为输出，传输待写入的第 1 个数据；
10. 设置 SDA 为三态门输入，读取从机应答信号；
11. 读取应答信号成功后，主机设置 SDA 为输出，传输待写入的下一个数据；
12. 设置 SDA 为三态门输入，读取从机应答信号；n 个数据被写完，转到步骤 13，若数据未被写完，转到步骤 11；
13. 读取应答信号成功，主机产生 STOP 位，终止传输。

注：对于 AT24Cxx 系列的 EEPROM 存储器，一次可写入的最大长度为 32 字节。

29.5I²C 读时序

29.5.1单字节读时序

同样的，I²C 读操作时序根据不同 I²C 器件具有不同的器件地址字节数，单字节读操作分为 1 字节地址段器件单字节数据读操作和 2 字节地址段器件单字节数据读操作。下面两幅图分别为不同情况的时序图。

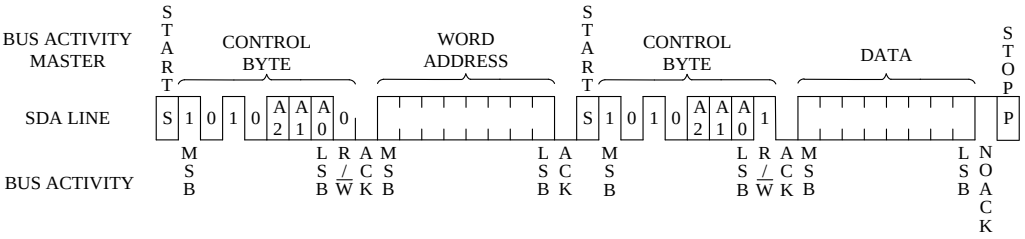


图 29-11 1 字节地址器件单字节读操作时序

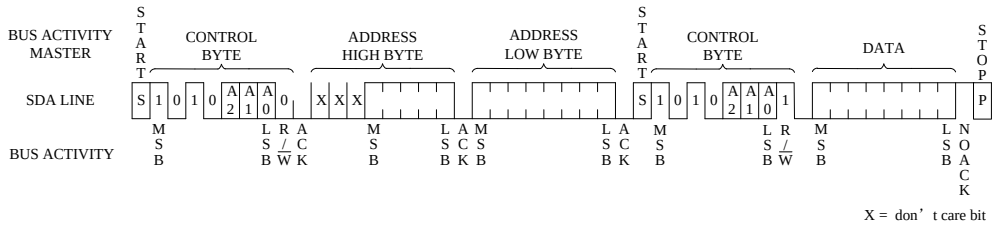


图 29-12 2 字节地址器件单字节读操作时序

根据时序图，从主机角度来描述一次读取单字节数据过程如下：

单字节地址读取单字节数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输 1 字节地址数据；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机发起起始信号；
8. 主机传输器件地址字节，其中最低位为 1，表明为读操作；
9. 设置 SDA 为三态门输入，读取从机应答信号；
10. 读取应答信号成功，主机设置 SDA 为三态门输入，读取 SDA 总线上的

一个字节的数据；

11. 产生无应答信号（高电平）（无需设置为输出高电平，因为总线会被自动拉高）；
12. 主机产生 STOP 位，终止传输。

双字节地址读取单字节数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据高字节；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据低字节；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功，主机发起起始信号；
10. 主机传输器件地址字节，其中最低位为 1，表明为读操作；
11. 设置 SDA 为三态门输入，读取从机应答信号；
12. 读取应答信号成功，主机设置 SDA 为三态门输入，读取 SDA 总线上的一个字节的数据；
13. 主机设置 SDA 输出，产生无应答信号（高电平）（无需设置为输出高电平，因为总线会被自动拉高）；
14. 主机产生 STOP 位，终止传输。

29.5.2连续读时序（页读取）

连续读是主机连续从从机读取多个字节数据，这个和单字节读操作类似，连续多字节读操作也是分为 1 字节地址段器件和 2 字节地址段器件的读操作，如下图分别为 1 字节地址段器件和 2 字节地址段器件连续多字节读时序图。

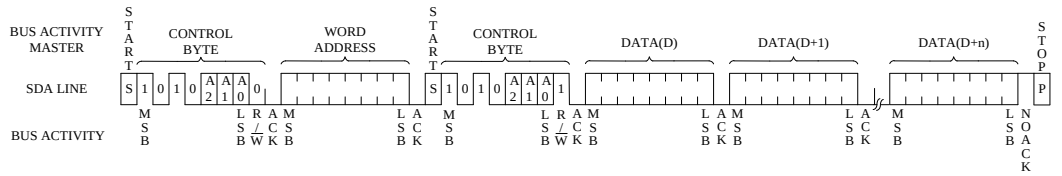


图 29-13 1 字节地址器件连续多字节读操作时序

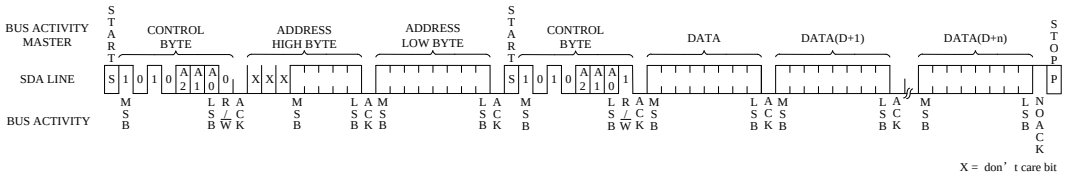


图 29-14 2 字节地址器件连续多字节读操作时序

根据时序图，从主机角度来描述一次读取多字节数据过程如下：

单字节地址器件连续多字节读取数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输 1 字节地址数据；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机发起起始信号；
8. 主机传输器件地址字节，其中最低位为 1，表明为读操作；
9. 设置 SDA 为三态门输入，读取从机应答信号；
10. 读取应答信号成功，主机设置 SDA 为三态门输入，读取 SDA 总线上的第 1 个字节的数据；
11. 主机设置 SDA 输出，发送一位应答信号；
12. 设置 SDA 为三态门输入，读取 SDA 总线上的下一个字节的数据；若 n 个字节数据读完成，跳转到步骤 13，若数据未读完，跳转到步骤 11；（对于 AT24Cxx，一次读取长度最大为 32 字节，即 n 不大于 32）
13. 主机设置 SDA 输出，产生无应答信号（高电平）（无需设置为输出高电平，因为总线会被自动拉高）；
14. 主机产生 STOP 位，终止传输。

双字节地址器件连续多字节读取数据过程：

1. 主机设置 SDA 为输出；
2. 主机发起起始信号；
3. 主机传输器件地址字节，其中最低位为 0，表明为写操作；
4. 主机设置 SDA 为三态门输入，读取从机应答信号；
5. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据高字节；
6. 主机设置 SDA 为三态门输入，读取从机应答信号；
7. 读取应答信号成功，主机设置 SDA 为输出，传输地址数据低字节；
8. 设置 SDA 为三态门输入，读取从机应答信号；
9. 读取应答信号成功，主机发起起始信号；
10. 主机传输器件地址字节，其中最低位为 1，表明为读操作；
11. 设置 SDA 为三态门输入，读取从机应答信号；
12. 读取应答信号成功，主机设置 SDA 为三态门输入，读取 SDA 总线上的第 1 个字节的数据；
13. 主机设置 SDA 输出，发送一位应答信号；
14. 设置 SDA 为三态门输入，读取 SDA 总线上的下一个字节的数据；若 n 个字节数据读完成，跳转到步骤 15，若数据未读完，跳转到步骤 13；（对于 AT24Cxx，一次读取长度最大为 32 字节，即 n 不大于 32）
15. 主机设置 SDA 输出，产生无应答信号（高电平）（无需设置为输出高电平，因为总线会被自动拉高）；
16. 主机产生 STOP 位，终止传输。

29.6 I^2C 控制器实现思路解析

经过 I^2C 写时序和 I^2C 读时序两个小节对 I^2C 读写时序的深入解析，想必大家已经知道了 I^2C 总线是如何完成一次存储器读写操作的完整过程了。也一定是跃跃欲试，在脑海中都已经想好了各种实现的方案。比如：

1. 使用线性序列机

根据 I^2C 传输的时序特点，是很容易分析总结出来，I^2C 单纯的读或者写时

序就像是时间轴上的一段连续的操作，我们只需要在指定的时间将 SDA 或者 SCL 信号拉高、拉低、或者设置为三态就可以了。

优点：代码编写方便，思路简单。

缺点：读写操作要合并到一个序列机中实现比较麻烦，如果分开实现读写序列，那么就需要制作读写序列的切换逻辑，虽然本身也不难，但是让整个设计的结构性失去了清爽。而且编写时候调试非常痛苦，一个完整的序列大概有一百多个时间节点，一旦写错一个节点，修改时候工作量非常大。

2. 使用状态机

序列机实现方便，但是编写过程中调试麻烦，而使用状态机则能够很方便的解决这个问题，比如，对整个的 I^2C 控制器从大的角度分为 3 个状态：空闲态（IDLE）、完整的写操作状态（WRITE）、完整的读操作状态（READ）。然后再对每个单独的写和读状态进行进一步划分，得到分别与读和写相关联的小状态，比如对于完整的写操作，又可分为细小的：

【发起始位】->【写器件地址】->【应答位】->【写存储器地址】->【应答位】->【写数据】->【应答位】->【停止位】。

而对于完整的读操作，又可分为细小的：

【发起始位】->【写器件地址】->【应答位】->【写存储器地址】->【应答位】->【发起始位】->【写器件地址-】>【应答位】->【读数据】->【应答位】->【停止位】。

当然了，上述状态划分，实际上有些状态是可以合并的。但是如果当你按照这样的思路来写的时候，你会发现，很多小状态里面又要分成更小的状态，或者使用序列机。比如写数据、写器件地址、读数据状态都是要执行 8 位数据的传输，这样的话，在小状态里面又需要使用 CASE 加计数器的方式来实现 8 位数据的传输，如果坚持写下去，到最终实现的效果就是整个的代码写完大概有超 500 行，状态交错，可读性差。

3. 通用传输逻辑建模

当然了，上述两种思路，笔者也曾都尝试过，其中编写和调试的难度，笔者也是深有体会。在对学员进行辅导时候，有很多学员分别按照上述思路，最终也都完成了系统设计。所以，上面的思路，是一定能行的通的，不可否认。

一个小小的 I^2C 控制器，就真的有那么难吗？我们能不能想到办法来简化这种时序？如果有做过单片机平台的 C 语言实现 I^2C 控制器的同学不妨回顾一下，

我们使用 C 语言来实现 I²C 控制器来对某器件读写操作时候是个怎么样的流程呢？

一般情况下，该驱动一定会有一个最基本的 8 位数据传输函数，假设为 write_byte(char data),该函数不关心需要传输的数据内容究竟是器件地址、存储器地址、还是要传输的数据，其只管把上层传过来的 8 位数据变成 1 位 1 位的发出去就完事了。

然后，该驱动会有一个写操作函数，假设为 write_reg(charaddr, char data)，该函数中会多次调用单字节写操作函数，从而实现一个完整的写操作时序。

所以，我们能不能借用该思路，将完整的读写操作时序进行分析归纳，得到一个通用的底层传输模块，使用该模块就能够完成 I²C 协议中每一段的操作，比如带起始位的写、带停止位的写、带停止位的读。然后再在上层模块多次使用该底层模块来实现一次完整的数据传输呢？

假设我们有下图这样一个模块，这个模块就像是一个搬运工，他的职责就是按照命令端口（CMD）的指示，将 Data 端口上的 8 位数据按照一定的格式发出去。CMD 本身有 6 位的位宽，所以理论上支持多达 64 种命令，但是实际上对于我们的 I²C 传输来说，根本用不着这么多命令。那么具体是那些命令呢？这就要结合 I²C 的传输时序进行分析了。

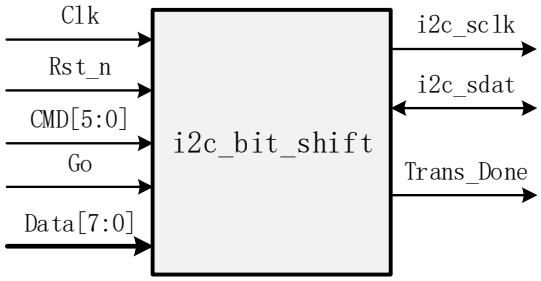


图 29-15 I²C 位传输模块接口图

再回顾一下 I²C 完整的读和写时序，以完整的读操作为例，如下图。

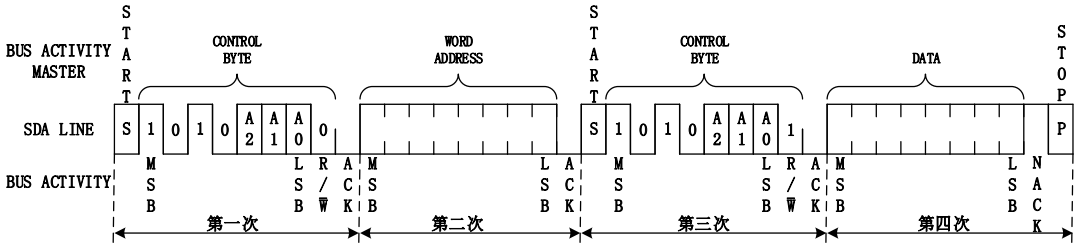


图 29-16 I²C 完整读操作时序

如果我们将起始位、停止位、应答位分别和就近的一次 8 位数据传输组合

到一起，就可以发现对于一次完整的读操作，实际可以分成以下三种小的传输：

- 第一次和第三次，起始位+写数据（7 位器件 ID + 1 位读写控制位），然后检查从机应答位，也就是带起始位的单字节写操作。
- 第二次，写数据（8 位 EEPROM 的寄存器地址），然后检查从机应答位。
- 第四次，读数据（从 EEPROM 中读出 8 位数据）+ 应答位（根据需要给出应答（0）或者无应答（1））+ 停止位，也就是带停止位的读操作。

而对于写操作，最后一次则是写 8 位数据，然后检查应答位，然后再产生停止位，也就是带停止位的写操作。

好了经过上述分析，我们基本可以总结出 I2C 传输协议中包含的各种情况，为以下五种：

- 带起始位的写操作，不带停止位，主要对应每次开始传输时候的一段。
- 不带起始位的写操作，也不带停止位，主要对应传输器件地址和连续写多个字节数据到器件中的操作，比如 EEPROM 的写写入操作。
- 不带起始位的写操作，但是带停止位，主要对应写入数据时候的一段，也就是完整写的最后一段。
- 不带起始位的读操作，也不带停止位，主要对应从器件中连续读出多个字节数据的操作，比如 EEPROM 的页读取操作。
- 不带起始位的读操作，但是带停止位，主要对应从器件中读出最后一个数据的那一段。

如果用图示的形式表达，就是下面的样子：

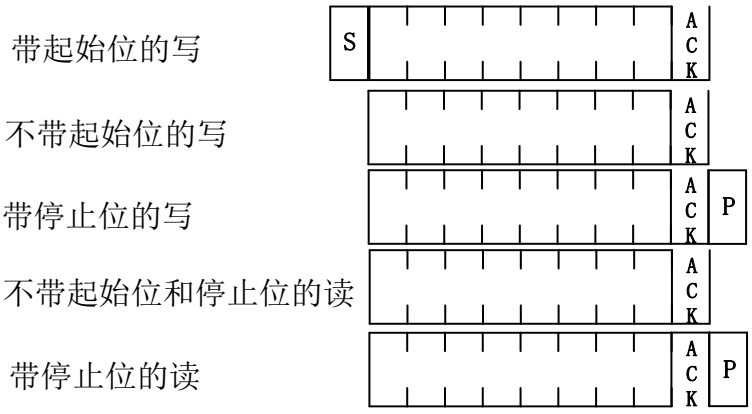


图 29-17 I²C 协议各种传输情况

总结下来，发现其实非常的简单，每段在传输的时候，只需要确定当前这

个字节的传输之前是否需要加入起始位，以及当前这个字节的传输结束后是否需要加入停止位就结束了。因此，我们完全可以编写一个能够根据指令的灵活的决定是否在每个字节的传输之前加入起始位，在每个字节之后是否加入停止位的逻辑，来应对这多种传输情况。

29.7i2c_bit_shift 模块设计

上一个小节，我们已经分析出了如何根据完整的 I^2C 传输时序，提取出通用的底层传输单元。本节我们将根据该思路，设计出完整的传输逻辑。

29.7.1位传输单元接口设计

按照通用底层传输单元的分析思路，可以设计出如下图 29-18 完整的底层通用传输逻辑单元：

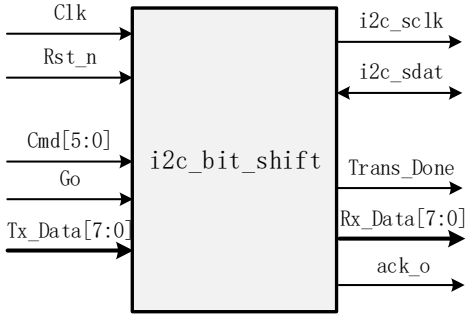


图 29-18 底层通用传输逻辑单元接口

先解释下模块里面的几个用到的输入输出端口信号的功能，如下表 29-1 所示：

表 29-1 i2c_bit_shift 模块接口功能描述

接口名称	I/O	功能描述
Clk	I	模块工作时钟，50M 时钟
Rst_n	I	模块复位信号
Cmd	I	控制总线实现各种传输操作的各种命令的组合（写、读、起始、停止、应答、无应答）
Go	I	整个模块的启动使能信号，为了接口使用方便，使用时希望只需要对该端口产生一个单时钟周期的脉冲即可启动一个字节完整的传输（含可能的起始位、停止位、应答位）
Rx_DATA	O	总线收到的 8 位数据，读操作时读到的数据由此端口输出
Tx_DATA	I	总线要发送的 8 位数据，需要传输的数据经此端口传入该模块
Trans_Done	O	发送或接收 8 位数据完成标志信号，每次传输完成都会产生一个单周期的高脉冲信号

ack_o	O	从机是否应答标志，在此底层逻辑中，我们暂不对总线上是否给出正确的应答信号做出任何的处理，仅将接收到的应答信号状态存储并输出，如果没有收到正确的应答信号，该如何执行下一步，由外部其他逻辑去决定。
i2c_sclk	O	i2c 时钟总线
i2c_sdat	I /O	i2c 数据总线

需要注意的是，在上述端口中，Cmd 端口的解释是控制总线实现各种传输操作的各种命令的组合。组合二字表明，该端口每次可能不止传输一个命令。什么意思呢，其实很好理解，以写操作来说，要实现 8 位数据的写，肯定要传输写命令，不能传读命令。但是同时在 8 位数据传输之前还有可能需要产生起始位，因此还要同时给该逻辑提供产生起始位的命令，或者要写完产生停止位，也要给该逻辑在提供写命令的同时还要提供产生停止位的命令。所以说每一次传输，命令端口输入的都应该是好几个命令的组合。传输逻辑会根据输入的命令组合依次去执行相应的操作，最终实现指定的传输序列。

分析完成端口之后，接下来就可以分析，要实现该传输，需要如何实现了。

29.7.2位传输单元状态机设计

由于序列机一般只能完成固定的序列传输，而此逻辑需要根据不同的命令产生可能需要的起始位和应答位，因此使用序列机的方法比较难控制，因此采用状态机的方式。

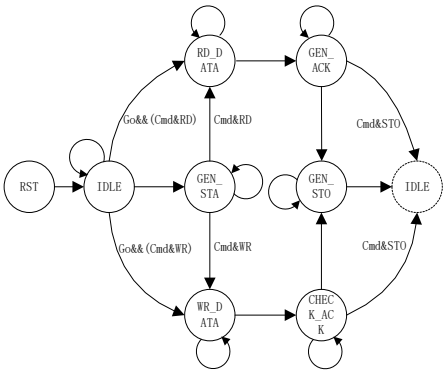


图 29-19 I²C 位传输状态转移图

从上面的状态转移图可以看出，这里总共分成了 7 个状态，刚开始复位（RST）后的默认状态（IDLE）、产生起始信号状态（GEN_STA）、写数据状态（WR_DATA）、读数据状态（RD_DATA）、检测从机是否应答状态（CHECK_ACK）、给从机应答状态（GEN_ACK）、产生停止位状态（GEN_STO），通过这些状态我们就能组合成基本的 I2C 读写时序了。

于是就可以定义如下几个状态：

```
localparam
     IDLE      = 7'b0000001,   //空闲状态
     GEN_STA   = 7'b0000010,   //产生起始信号
     WR_DATA   = 7'b0000100,   //写数据状态
     RD_DATA   = 7'b0001000,   //读数据状态
     CHECK_ACK = 7'b0010000,   //检测应答状态
     GEN_ACK   = 7'b0100000,   //产生应答状态
     GEN_STO   = 7'b1000000;   //产生停止信号
```

当然为了方便组合成对应传输情况，我们在这里定义了组合成 Cmd 里面出现的 6 个命令（写请求<WR>、起始位请求<STA>、读请求<RD>、停止位请求<STO>、应答位请求<ACK>、无应答请求<NACK>），同样为了更直观我们和在代码里面处理，我们也将其定义成了参数：

```
localparam
     WR   =6'b000001,//写请求
     STA  =6'b000010,//起始位请求
     RD   =6'b000100,//读请求
     STO  =6'b001000,//停止位请求
     ACK  =6'b010000,//应答位请求
     NACK =6'b100000;//无应答请求
```

例如：我们来一次写器件地址操作，那么我们根据写操作时序来就是，起始位、器件地址、最低位为 0（表示写）、停止位，如果我们简化一下就可以用 Cmd 加上器件地址就可以了，那么此时的 Cmd 命令里面就应该含有写请求<WR>、起始位请求<STA>、停止位请求<STO>，这时的 Cmd 就可以用个小技巧（Cmd=WR | STA | STO）那么这个 Cmd 到时候怎么来用呢，这就是下面要说的上面定义的 7 种状态的 IDLE 状态。

在 Go 脉冲控制信号的控制下，使能 en_div_cnt 来让上面的计数器模块运行，同时将 Cmd 命令和上面定义的几种命令进行按位与运算，最终来确定状态的跳转。当然这个是有优先级的，因为只有产生了起始信号（STA），才能进行写操作（WR）和读操作（RD），所以空闲状态代码里面可以按照这个优先级来将 Cmd 和上面的几种命令进行按位与运算，如果运算结果为 1，就可以跳转到下面对应的状态，如果运算结果为 0 就接着往下来判断，代码如下：

```
IDLE:
     begin
          Trans_Done <= 1'b0;
          i2c_sdat_oe <= 1'd1;
          if (Go) begin
               en_div_cnt <= 1'b1;
               if (Cmd & STA)
                    state <= GEN_STA;
               else if (Cmd & WR)
```

```
                    state <= WR_DATA;
                else if (Cmd & RD)
                    state <= RD_DATA;
                else
                    state <= IDLE;
            end else begin
                en_div_cnt <= 1'b0;
                state <= IDLE;
            end
        end
```

根据上面的代码，显然写器件地址操作的 Cmd 命令里面的条件会首先跳转到 GEN_STA 这个状态，根据 i2c 总线协议规定，在时钟（SCL）为高电平的时候，数据总线（SDA）由高到低的跳变为总线起始信号，所以这里刚开始第一步将 i2c_sdat_o 设置为 1，i2c_sdat_oe 使能，第二步将总线时钟（SCL，代码里面 i2c_sclk）拉高，第三步将第一步已经被上拉电阻拉高的 i2c_sdat 再拉低（代码里面是通过拉低 i2c_sdat_o 来间接拉低 i2c_sdat），此时的 i2c_sclk 应该还是维持高电平，第四步将时钟总线 i2c_sclk 拉为低电平。这里将一个操作分成了 4 步来完成，这也是为什么下面在计算这个 SCL_CNT_M 的时候除以 4 的原因。产生起始信号状态代码如下：

```
GEN_STA:
    begin
        if (sclk_plus) begin
            if (cnt == 3)
                cnt <= 0;
            else
                cnt <= cnt + 1'b1;
            case (cnt)
                0:begin i2c_sdat_o <= 1; i2c_sdat_oe <= 1'd1;end
                1:begin i2c_sclk <= 1;end
                2:begin i2c_sdat_o <= 0; i2c_sclk <= 1;end
                3:begin i2c_sclk <= 0;end
                default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
            endcase
            if (cnt == 3) begin
                if (Cmd & WR)
                    state <= WR_DATA;
                else if (Cmd & RD)
                    state <= RD_DATA;
            end
        end
    end
```

上面的代码中产生了起始信号后，接着就是发送 7 位器件地址和 1 位方向

位（0：写，1：读），这里例举的是写器件地址操作，所以当然最低位为 0。那么起始信号发送完后同时就要在里面判断这个 Cmd 中是否含有写数据（WR）命令，再次判断如果 Cmd 和写数据（WR）命令进行按位与运算，如果运算结果为1，说明命令里面有写数据请求，此时就可以跳转到写数据状态，如果运算结果为 0 就接着往下来判断，还是一样的判断方法。这里当然是跳转到写数据状态。写数据状态代码如下：

```
WR_DATA:
    if (sclk_plus) begin
        if (cnt == 31)
            cnt <= 0;
        else
            cnt <= cnt + 1'b1;
        case (cnt)
            0,4,8,12,16,20,24,28:
                begin
                    i2c_sdat_o <= Tx_DATA[7-cnt[4:2]];
                    i2c_sdat_oe <= 1'd1;
                end //set data;
            1,5,9,13,17,21,25,29:begin i2c_sclk<=1;end//sclk posedge
            2,6,10,14,18,22,26,30:begin i2c_sclk<=1;end//sclkkeep high
            3,7,11,15,19,23,27,31:begin i2c_sclk <=0;end//sclk negedge
/*
            0 :begin i2c_sdat_o <= Tx_DATA[7];end
            1 :begin i2c_sclk <= 1;end  //sclk posedge
            2 :begin i2c_sclk <= 1;end  //sclk keep high
            3 :begin i2c_sclk <= 0;end  //sclk negedge

            4 :begin i2c_sdat_o <= Tx_DATA[6];end
            5 :begin i2c_sclk <= 1;end  //sclk posedge
            6 :begin i2c_sclk <= 1;end  //sclk keep high
            7 :begin i2c_sclk <= 0;end  //sclk negedge

            8 :begin i2c_sdat_o <= Tx_DATA[5];end
            9 :begin i2c_sclk <= 1;end  //sclk posedge
            10:begin i2c_sclk <= 1;end  //sclk keep high
            11:begin i2c_sclk <= 0;end  //sclk negedge

            12:begin i2c_sdat_o <= Tx_DATA[4];end
            13:begin i2c_sclk <= 1;end  //sclk posedge
            14:begin i2c_sclk <= 1;end  //sclk keep high
            15:begin i2c_sclk <= 0;end  //sclk negedge

            16:begin i2c_sdat_o <= Tx_DATA[3];end
            17:begin i2c_sclk <= 1;end  //sclk posedge
```

```
            18:begin i2c_sclk <= 1;end  //sclk keep high
            19:begin i2c_sclk <= 0;end  //sclk negedge

            20:begin i2c_sdat_o <= Tx_DATA[2];end
            21:begin i2c_sclk <= 1;end  //sclk posedge
            22:begin i2c_sclk <= 1;end  //sclk keep high
            23:begin i2c_sclk <= 0;end  //sclk negedge

            24:begin i2c_sdat_o <= Tx_DATA[1];end
            25:begin i2c_sclk <= 1;end  //sclk posedge
            26:begin i2c_sclk <= 1;end  //sclk keep high
            27:begin i2c_sclk <= 0;end  //sclk negedge

            28:begin i2c_sdat_o <= Tx_DATA[0];end
            29:begin i2c_sclk <= 1;end  //sclk posedge
            30:begin i2c_sclk <= 1;end  //sclk keep high
            31:begin i2c_sclk <= 0;end  //sclk negedge
*/
            default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
        endcase
        if (cnt == 31) begin
            state <= CHECK_ACK;
        end
end
```

为了让写数据这个状态写的数据更灵活，这里我们把要写入的 8 位数据定义成输入端口 Tx_DATA，那么我们把 7 位器件地址和最低位（方向位）的 0 直接赋值给 Tx_DATA，之后就可以直接统一发送 Tx_DATA 里面的数据就行了。同样我们还是按照产生起始位一样的规律，写数据的时候每一个 bit 也是分成 4 步来完成，第一步将要发送的数据准备好（i2c_sdat_o<=Tx_DATA[7]），i2c_sdat_oe 使能，第二步将总线时钟（SCL，代码里面 i2c_sclk）拉高，第三步将总线时钟继续保持为高电平，第四步将时钟总线 i2c_sclk 拉为低电平。这样就将最高位的数据通过总线发送出去了，接着还是一样的分成 4 步来发送 Tx_DATA[6]...直到最低位 Tx_DATA[0]发送完成。这里也用了个小技巧，根据 cnt 的值来动态的给数据总线（代码里面是 i2c_sdat_o）赋值，如果用线性序列机来写代码就很长，这里依然保留线性序列机的写法（绿色的注释部分）。

写完数据后我们就要判断是否真的写成功了，判断依据就是对方有没有产生应答，所以写完数据后就要跳转到检测应答状态（CHECK_ACK）。

检测应答状态一样的分为 4 步，第一步将总线设置为输入，即 i2c_sdat_oe 设置为 0，i2c_sclk 拉为低电平，第二步将总线时钟 i2c_sclk 拉高，第三步总线时钟继续保持为高电平，读取数据总线 i2c_sdat 的值到 ack_o，此时判断对方产

生是否产生应答只需要判断 ack_o 的值是 0（应答）还是 1（无应答）了，第四步将时钟总线 i2c_sclk 拉为低电平。检测应答状态代码如下：

```
CHECK_ACK:
    begin
        if(sclk_plus)begin
            if (cnt == 3)
                cnt <= 0;
            else
                cnt <= cnt + 1'b1;
            case(cnt)
                0:begin i2c_sdat_oe <= 1'd0; i2c_sclk <= 0;end
                1:begin i2c_sclk <= 1;end
                2:begin ack_o <= i2c_sdat; i2c_sclk <= 1;end
                3:begin i2c_sclk <= 0;end
                default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
            endcase
            if(cnt == 3)begin
                if(Cmd & STO)
                    state <= GEN_STO;
                else begin
                    state <= IDLE;
                    Trans_Done <= 1'b1;
                end
            end
        end
end
```

检测完应答后要判断 Cmd 里面是否含有要产生停止信号的命令，如果就跳转到产生停止信号（GEN_STO）状态，没有就跳转到空闲状态（IDLE）。根据这个举的写器件地址例子，Cmd 里面是有产生停止位命令请求的，所以接下来就是跳转到产生停止信号（GEN_STO）状态

根据 i2c 总线协议规定，在时钟（SCL，代码里面是 i2c_sclk）为高电平的时候，数据总线（SDA）由低电平到高电平的跳变就一个停止信号，所以这里刚开始第一步将 i2c_sdat_o 设置为 0，i2c_sdat_oe 使能，第二步将总线时钟 i2c_sclk 拉高，第三步将 i2c_sdat_o 设置为 1，让外部上拉电阻将数据总线 i2c_sdat 拉成高电平，此时的 i2c_sclk 应该还是维持高电平，第四步将时钟总线 i2c_sclk 拉为低电平。产生停止信号代码如下：

```
GEN_STO:
    begin
        if(sclk_plus)begin
            if(cnt == 3)
                cnt <= 0;
```

```
            else
                cnt <= cnt + 1'b1;
            case(cnt)
                0:begin i2c_sdat_o <= 0; i2c_sdat_oe <= 1'd1;end
                1:begin i2c_sclk <= 1;end
                2:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
                3:begin i2c_sclk <= 1;end
                default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
            endcase
            if(cnt == 3)begin
                Trans_Done <= 1'b1;
                state <= IDLE;
            end
        end
end
```

到这里一个基本的写器件地址操作就完成了，当然作为一个基本的 i2c 操作，读操作肯定是少不了的，我们可以按照写操作和应答检测的思路来，第一步将总线设置为输入，即 i2c_sdat_oe 设置为 0，i2c_sclk 拉为低电平，第二步将总线时钟 i2c_sclk 拉高，第三步将总线时钟继续保持为高电平，读取数据总线 i2c_sdat 的值到 Rx_DATA[7]，第四步将时钟总线 i2c_sclk 拉为低电平。这样就将最高位的数据通过总线读取出来了，接着还是一样的分成 4 步来读取 Rx_DATA[6]...直到最低位 Rx_DATA[0]读取完成。这里也用了个小技巧，每次将读取到的数据放到 Rx_DATA 的最低位，读取一次，左移一次，读完 8 次数据后，第一次读取的最高位数据也就通过移位的方式放到了最高位。读数据状态代码如下：

```
RD_DATA:
    if (sclk_plus) begin
        if (cnt == 31)
            cnt <= 0;
        else
            cnt <= cnt + 1'b1;
        case (cnt)
            0,4,8,12,16,20,24,28:
                begin
                    i2c_sdat_oe <= 1'd0;
                    i2c_sclk <= 0;
                end //set data;
            1,5,9,13,17,21,25,29:begin i2c_sclk<=1;end//sclk posedge
            2,6,10,14,18,22,26,30:
                begin
                    i2c_sclk <= 1;
                    Rx_DATA <= {Rx_DATA[6:0],i2c_sdat};
```

```
                end //sclk keep high
            3,7,11,15,19,23,27,31:begin i2c_sclk<=0;end//sclk negedge
            default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
        endcase
        if (cnt == 31) begin
            state <= GEN_ACK;
        end
end
```

读完数据后跳转到产生应答状态（GEN_ACK），这个状态里面会根据 Cmd 里面是否含有应答位请求（ACK）或者无应答请求（NACK）来给对方做出一个应答，产生应答信号跟写数据状态一样的思路，分为 4 步，第一步将 Cmd 与 ACK 进行与运算，如果结果为 1 说明需要产生应答，此时将 i2c_sdat_o 赋值为 0，如果不满足，接着往下判断，将 Cmd 与 NACK 进行与运算，如果结果为 1 说明需要产生非应答信号，此时将 i2c_sdat_o 赋值为 1，让外部上拉电阻将数据总线拉成高电平。将 i2c_sdat_oe 使能，将时钟总线 i2c_sclk 拉为低电平，第二步将总线时钟 i2c_sclk 拉高，第三步将总线时钟继续保持为高电平，第四步将时钟总线 i2c_sclk 拉为低电平。这样就就会根据 Cmd 给对方做出了应答，此时还要判断 Cmd 里面是否含有跳转到产生停止信号（STO）的命令（通过判断 Cmd & STO 是否为 1 就能判断），如果有就要跳转到产生停止信号（GEN_STO）状态了，代码如下：

```
GEN_ACK:
    begin
        if(sclk_plus)begin
            if(cnt == 3)
                cnt <= 0;
            else
                cnt <= cnt + 1'b1;
            case(cnt)
                0:  begin
                        i2c_sdat_oe <= 1'd1;
                        i2c_sclk <= 0;
                        if(Cmd & ACK)
                            i2c_sdat_o <= 1'b0;
                        else if(Cmd & NACK)
                            i2c_sdat_o <= 1'b1;
                    end
                1:begin i2c_sclk <= 1;end
                2:begin i2c_sclk <= 1;end
                3:begin i2c_sclk <= 0;end
                default:begin i2c_sdat_o <= 1; i2c_sclk <= 1;end
            endcase
```

```
            if(cnt == 3)begin
                if(Cmd & STO)
                    state <= GEN_STO;
                else begin
                    state <= IDLE;
                    Trans_Done <= 1'b1;
                end
            end
        end
end
```

29.7.3位传输单元关键逻辑设计

我们根据 I^2C 的协议标准，这里采用的是快速模式（400kb/s）来作为总线的工作时钟，当然为了让这个模块更具有通用性和灵活性，这里就将系统时钟和产生的总线工作时钟频率都写成了参数化，这样就方便随时修改。

这里系统用的是 50MHz 的时钟，所以我们将 SYS_CLOCK 设置成 50_000_000，实际也可以根据输入的时钟来更改这个值，SCL_CLOCK 是用来配置时钟（SCL）总线的频率，通过这两个参数就可以计算出参数 SCL_CNT_M 要计数到多少就能产生期望的总线工作时钟（SCL）频率，为什么要在计算中除以 4，在产生起始信号状态部分已经有说明。

```
//系统时钟采用 50MHz
parameter SYS_CLOCK =50_000_000;
//SCL 总线时钟采用 400kHz
parameter SCL_CLOCK =400_000;
//产生时钟 SCL 计数器最大值
localparam SCL_CNT_M = SYS_CLOCK/SCL_CLOCK/4-1;
```

有了上面的 SCL_CNT_M，接下来就可以根据这个值来产生状态机部分的工作时钟 sclk_plus。这个部分的逻辑运行是通过 en_div_cnt 的有效来控制的，当然这个信号的有效与否是受脉冲信号 Go 控制的，在状态机部分可以体现出来。

```
reg [19:0]div_cnt;
reg en_div_cnt;
always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
     div_cnt <= 20'd0;
else if(en_div_cnt)begin
     if(div_cnt < SCL_CNT_M)
          div_cnt <= div_cnt + 1'b1;
     else
          div_cnt <= 0;
end else
     div_cnt <= 0;
```

```
wire sclk_plus = div_cnt == SCL_CNT_M;
```

对于 I²C 总线，要求连接到总线上的输出端必须是开漏输出结构，给不了高电平，所以总线上所有的高电平应该是由上拉电阻上拉达到效果的，而不是由主机直接给总线赋值 1 就能实现，所以我们在写这个逻辑的时候也应该遵循这个标准，当总线上要输出低电平的时候，我们就直接给总线赋值 0，要输出高电平的时候，只能将总线设置成高阻态，这样再由外部上拉电阻来上拉成高电平，这里为了方便用户理解，就把我们给赋值给总线的值先赋值给 i2c_sdat_o，然后再控制使能信号 i2c_sdat_oe，通过这两个信号间接的来给数据总线 SDA（代码里面是 i2c_sdat）赋值。代码中是通过下面一行代码来实现的。

```
assign i2c_sdat =!i2c_sdat_o && i2c_sdat_oe ?1'b0:1'bz;
```

上面的 Cmd 部分的命令因为采用的是独热码编码，所以 Cmd 部分的命令组合可以通过小技巧按位或操作来实现，解析这个 Cmd 命令的时候就可以通过按位与操作来判断是否含有对应的命令，例如：

带起始位和停止位的写地址操作，这时的 Cmd 里面就应该有起始位请求<STA>、停止位请求<STO>、写请求<WR>，此时按照上面说的小技巧那就可以这样来给 Cmd 赋值了，Cmd=WR | STA | STO，在状态机里面的部分要来提取这 Cmd 里面的命令，也可以按照上面说的小技巧按位与操作来判断，此时的判断就有优先级，当然这部分在 IDLE 状态也有体现，具体如下：

```
if (Cmd & STA)
     state <= GEN_STA;
else if (Cmd & WR)
     state <= WR_DATA;
else if (Cmd & RD)
     state <= RD_DATA;
else
     state <= IDLE;
```

当然跳转到写请求或者读请求状态里面也会对 Cmd 里面的命令做进一步的判断，这个在对应状态里面可以看出来。

写请求里面的数据发送过程其实也是有规律的，按照常规的写法一般就是一个线性序列机，当然这里肯定也是有技巧的，根据 cnt 的值来动态的给数据总线（代码里面是 i2c_sdat_o）赋值，这就大大简化了发送时候的代码量，当然这里线性序列机写法的代码依然还保留着。

29.7.44 位传输单元仿真验证

到此上面关于位传输单元的接口设计就讲完了，这个模块是否真的能实现

所有基本出现的几种命令组合都能实现，我们就来仿真验证一下，

在按键消抖实验章节，我们曾经使用了成熟的随机抖动仿真模型，来模拟随机抖动的发生过程。事实上，如果想对某硬件器件进行仿真，可以到该硬件的官方平台上寻找相应的仿真模型。作为该器件硬件的生产商，我们从器件官网获得的仿真模型是最具权威和代表性的。这里，我们对 EEPROM 的仿真模型进行仿真，从而能更好的检验 I2C 控制器是否存在问题，以便后面进行优化和改进。

本实验采用的是镁光官网提供的 EEPROM 仿真模型，读者可以从我们提供的例程代码中搜索：仿真模型 24LC04B.v 文件。

或者在芯路恒论坛搜索：

I2C 协议读写 EEPROM 存储器完整设计加仿真文件，并从工程中获取仿真模型。

http://www.corecourse.cn/forum.php?mod=viewthread&tid=27812

在这里，我们选择具有代表性的器件 24LC04B 进行模型下载和仿真。

仿真代码如下：

```
`timescale 1ns / 1ps
module i2c_bit_shift_tb;
    reg Clk;
    reg Rst_n;
    reg [5:0] Cmd;
    reg Go;
    wire [7:0] Rx_DATA;
    reg [7:0] Tx_DATA;
    wire Trans_Done;
    wire ack_o;
    wire i2c_sclk;
    wire i2c_sdat;

    pullup PUP(i2c_sdat); //模拟外部上拉电阻

    localparam
        WR   = 6'b000001,   //写请求
        STA  = 6'b000010,   //起始位请求
        RD   = 6'b000100,   //读请求
        STO  = 6'b001000,   //停止位请求
        ACK  = 6'b010000,   //应答位请求
        NACK = 6'b100000;   //无应答请求

    i2c_bit_shift DUT(
        .Clk(Clk),
```

```
        .Rst_n(Rst_n),
        .Cmd(Cmd),
        .Go(Go),
        .Rx_DATA(Rx_DATA),
        .Tx_DATA(Tx_DATA),
        .Trans_Done(Trans_Done),
        .ack_o(ack_o),
        .i2c_sclk(i2c_sclk),
        .i2c_sdat(i2c_sdat)
    );
    M24LC04B M24LC04B(
        .A0(0),
        .A1(0),
        .A2(0),
        .WP(0),
        .SDA(i2c_sdat),
        .SCL(i2c_sclk),
        .RESET(~Rst_n)
    );
    always #10 Clk = ~Clk;
    initial begin
        Clk = 1;
        Rst_n = 0;
        Cmd = 6'b000000;
        Go = 0;
        Tx_DATA = 8'd0;
        #2001;
        Rst_n = 1;
        #2000;

        //写数据操作，往 EEPROM 器件的 B1 地址写数据 DA
        //第一次：起始位+EEPROM 器件地址（7 位）+写方向（1 位）
        Cmd = STA | WR;
        Go = 1;
        Tx_DATA = 8'hA0 | 8'd0;//写方向
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
        #200;

        //第二次：写 EEPROM 的 8 位寄存器地址
        Cmd = WR;
        Go = 1;
        Tx_DATA = 8'hB1;//写地址 B1
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
```

```
        #200;

        //第三次：写 8 位数据 + 停止位
        Cmd = WR | STO;
        Go = 1;
        Tx_DATA = 8'hda;//写数据 DA
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
        #200;

        #5000000; //仿真模型的两次操作时间间隔

        //读数据操作，从 EEPROM 器件的 B1 地址读数据
        //第一次：起始位+EEPROM 器件地址（7 位）+写方向（1 位）
        Cmd = STA | WR;
        Go = 1;
        Tx_DATA = 8'hA0 | 8'd0;//写方向
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
        #200;

        //第二次：写 EEPROM 的 8 位寄存器地址
        Cmd = WR;
        Go = 1;
        Tx_DATA = 8'hB1;//写地址 B1
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
        #200;

        //第三次：起始位+EEPROM 器件地址（7 位）+读方向（1 位）
        Cmd = STA | WR;
        Go = 1;
        Tx_DATA = 8'hA0 | 8'd1;//读方向
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
        #200;

        //第四次：读 8 位数据 + 停止位
        Cmd = RD | STO;
        Go = 1;
        @ (posedge Clk);
        Go = 0;
        @ (posedge Trans_Done);
```

```
        #200;

        #2000;
        $stop;
    end

endmodule
```

在上面的代码中，按照拆分的思路分成三次将数据写进去了，之后有一个#5000000;的延时，这个是仿真模型的两次操作时间间隔，小于这个延时的，操作就会有问题，后面就是再把前面写的数据再读取出来，按照拆分思路分成可四次来完成，仿真波形如下图 29-20 所示：

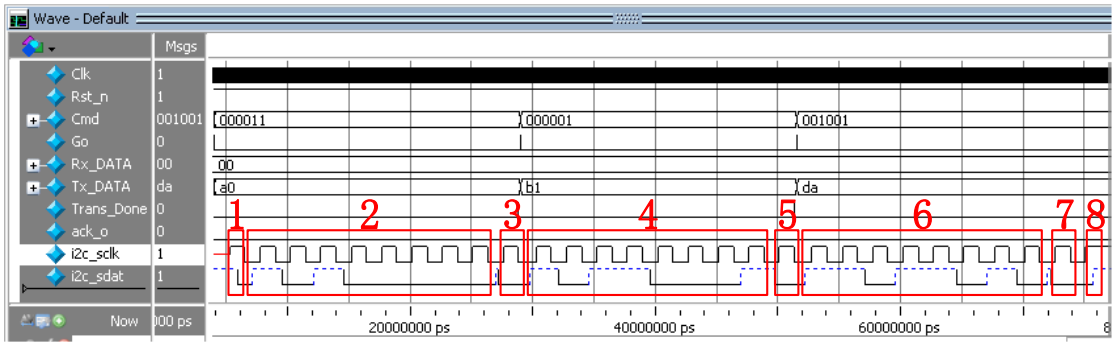


图 29-20 发送数据波形仿真图

第一次，波形图的 1、2，1 是起始位，2 是 EEPROM 器件地址（7 位）和最低位 0（写方向），后面的 3 是 EEPROM 器件给的应答位。

第二次，波形图的 4，是写 EEPROM 的 8 位寄存器地址（B1），后面的 5 是 EEPROM 器件给的应答位。

第三次，波形图的 6、7、8，6 是写入的数据（DA），7 是 EEPROM 器件给的应答位，8 是停止位。

从波形图可以看出，写入的数据，命令完全对应，接下来我们再来看一下读数据的波形图：

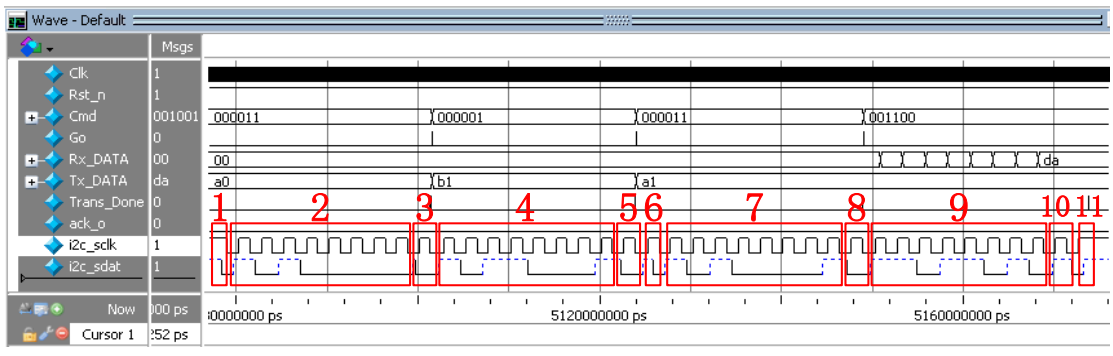


图 29-21 读数据波形图

第一次，波形图的 1、2，1 是起始位，2 是 EEPROM 器件地址（7 位）和最低位 0（写方向），后面的 3 是 EEPROM 器件给的应答位。

第二次，波形图的 4，是写 EEPROM 的 8 位寄存器地址（B1），后面的 5 是 EEPROM 器件给的应答位。

第三次，波形图的 6、7、8，6 是起始位，7 是 EEPROM 器件地址（7 位）和最低位 1（读方向），后面的 8 是 EEPROM 器件给的应答位。

第四次，波形图的 9、10、11，9 是读出的数据（DA），10 是主机给 EEPROM 器件的无应答位，11 是停止位。

从波形图可以看出，读出的数据与命令完全对应，与写入的数据也是一致的说明我们的设计是可以实现这些情况的组合的，通过这些组合就能实现 I²C 的各种操作了。

29.8i2c_control 模块设计

上面的 i2c_bit_shift 模块说完了，我们发现实现一个字节的写操作还是可以实现的，实际的应用中我们不可能只写一个字节的数据，那么此时这个 i2c_bit_shift 模块用来连续写就有些不方便了，从上面的仿真代码就能看出来，这时就需要一个上层模块来控制这个 i2c_bit_shift 模块去连续写，这样就方便一些。

还是用一个实际的例子来说明。例如我们要从一个 1 字节器件地址的 EEPROM 中的 0C 地址读取数据。我们来看一下上面说的单字节读时序图。

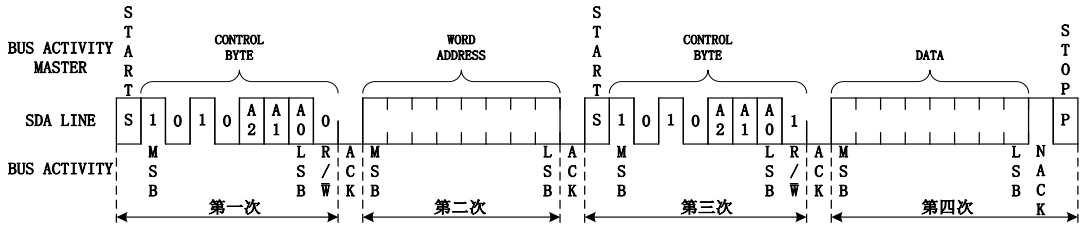


图 29-22 I²C 控制器模块设计位传输图

首先我们按照 i2c_bit_shift 模块的操作来拆分一下，这里面可以分成四次操作：

第一次，起始位+写数据（7 位器件 ID + 1 位读写控制位（这里是 0，表示写）），等待从机应答。

第二次，写数据（8 位 EEPROM 的寄存器地址），等待从机应答。

第三次，起始位+写数据（7 位器件 ID + 1 位读写控制位（这里是 1，表示

读)),等待从机应答。

第四次,读数据(从 EEPROM 中读出 8 位数据)+ 应答位(根据需要给出应答(0)或者无应答(1))+ 停止位。

总的来说就是写了三次 1 字节的数据,读了一次 1 字节的数据。为了让代码更简洁这里就用到了两个 task 一个是用来写字节(write_byte),另一个是读字节(read_byte)。写字节的 task 里面就将要写的字节数据准备好(赋值给 i2c_bit_shift 模块的 Tx_DATA 端口),同时涉及到的传给 i2c_bit_shift 模块的 Cmd 命令也准备好,同时触发 Go 信号,这样就可以通过 i2c_bit_shift 模块将要写的数据发送到总线上了。同理,读字节的 task 里面将涉及到的传给 i2c_bit_shift 模块的 Cmd 命令也准备好,同时触发 Go 信号,这样就可以通过 i2c_bit_shift 模块将总线上的数据读取出来了。代码如下:

```
task read_byte;
    input [5:0]Ctrl_Cmd;
    begin
        Cmd <= Ctrl_Cmd;
        Go <= 1'b1;
    end
endtask

task write_byte;
    input [5:0]Ctrl_Cmd;
    input [7:0]Wr_Byte_Data;
    begin
        Cmd <= Ctrl_Cmd;
        Tx_DATA <= Wr_Byte_Data;
        Go <= 1'b1;
    end
endtask
```

这样的话代码就可以简化成下面这样了,第三行的 device_id | 8'd1 是因为后面要进行读操作,所以用这个小技巧就可以把原本的写改成读了。

```
write_byte(WR | STA, device_id);
write_byte(WR | STO,8'h0C);
write_byte(WR | STA, device_id |8'd1);
read_byte(RD | ACK | STO);
```

那么上面的代码还是有问题,怎么就判断第一行的这个 write_byte 的 task 把数据写成功了呢,这个就需要判断 i2c_bit_shift 模块返回的 Trans_Done 的握手信号,这样就可以接着发送第二行的 write_byte 的 task,以此类推的执行完这四行的 task 就可以实现从 EEPROM 中的 0C 地址读取数据。

这里我们就分成 3 个状态来控制这个过程,第一个是读寄存器状态

（RD_REG），这里面有这个读数据操作的四个 task，第二个是等待读寄存器完成状态（WAIT_RD_DONE），这里面是等待每一次 task 执行完成，然后控制状态跳转回去继续执行下一行的 task，第三个是读寄存器完成状态（RD_REG_DONE），这个状态是向外反馈一个读写操作完成（RW_Done）状态，同时会在这个状态将读取到的值取出来赋值到 rddata。代码如下：

```
RD_REG:
    begin
        state <= WAIT_RD_DONE;
        case(cnt)
            0:write_byte(WR | STA, device_id);
            1:write_byte(WR, reg_addr[15:8]);
            2:write_byte(WR, reg_addr[7:0]);
            3:write_byte(WR | STA, device_id | 8'd1);
            4:read_byte(RD | NACK | STO);
            default:;
        endcase
end
```

每写入一次数据就会跳转到等待读完成状态，同时在这个状态里面会根据 addr_mode 来确定写的是 8 位地址，还是 16 位的地址，从而来控制 cnt 的值，后面回到读寄存器状态（RD_REG）里面的时候就可以选择性跳过发送高 8 位全 0 来代替的地址了。代码如下：

```
WAIT_RD_DONE:
    begin
        Go <= 1'b0;
        if(Trans_Done)begin
            if(cnt <= 3)
                ack <= ack | ack_o;
            case(cnt)
                0: begin cnt <= 1; state <= RD_REG;end
                1: begin
                        state <= RD_REG;
                        if(addr_mode)
                            cnt <= 2;
                        else
                            cnt <= 3;
                    end
                2: begin
                        cnt <= 3;
                        state <= RD_REG;
                    end
                3:begin
                        cnt <= 4;
                        state <= RD_REG;
```

```
                    end
                4:state <= RD_REG_DONE;
                default:state <= IDLE;
            endcase
        end
end
```

当读完数据后就会跳转到读完成状态，将 RW_Done 拉高，同时取出读到的数据，跳转到空闲状态，代码如下：

```
RD_REG_DONE:
    begin
        RW_Done <= 1'b1;
        rddata <= Rx_DATA;
        state <= IDLE;
end
```

这里同样是为了兼容 2 字节地址器件，所以在第二行的写 task 里面的具体的 8'h0C 的寄存器地址就用一个 16 位的变量（reg_addr）来代替，这时的高 8 位就用全 0 来代替，这时会引入一个新的控制信号 addr_mode（当 addr_mode 为 0，表示的是 8 位的地址，为 1，表示的是 16 位地址），因为是从高位先发送，所以为了兼容 8 位的地址，也就是当 addr_mode 为 0 的时候将 16 位的变量（reg_addr）的高字节和低字节替换：

```
assign reg_addr = addr_mode?addr:{addr[7:0],addr[15:8]};
```

那么在等待读寄存器完成状态（WAIT_RD_DONE）通过控制 cnt 的值，从而在读寄存器状态（RD_REG）里面直接跳过发送高 8 位全 0 来代替的地址了。

同理，如果往 1 字节器件地址的 EEPROM 中的 0C 地址写数据 0A 呢？我们来看一下上面说的单字节写时序图。

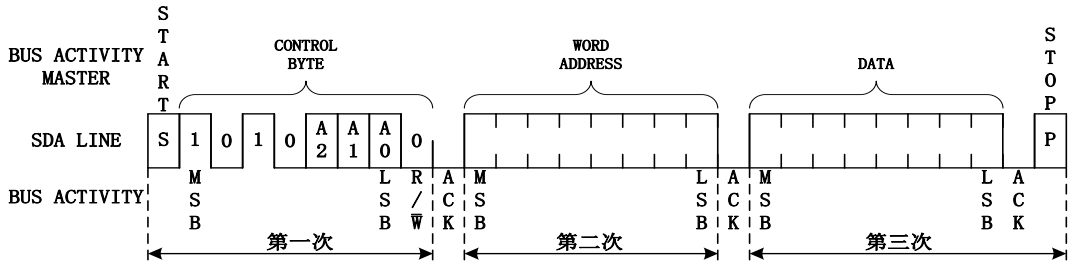


图 29-23 单字节传输时序图

首先我们按照 i2c_bit_shift 模块的操作来拆分一下，这里面可以分成三次操作：

第一次，起始位+写数据（7 位器件 ID + 1 位读写控制位（这里是 0，表示写）），等待从机应答。

第二次，写数据（8 位 EEPROM 的寄存器地址），等待从机应答。

第三次，写数据（写 8 位数据到 EEPROM 中），等待从机应答，然后给出停止位信号。

简单来说就是用了 write_byte 这个 task 写了三次 1 字节的数据，按照上面的时序图，三次 task 应该这样来给参数。

```
write_byte(WR | STA, device_id);
write_byte(WR, reg_addr[15:8]);
write_byte(WR, reg_addr[7:0]);
write_byte(WR | STO, wrdata);
```

上面的代码还是一样可以分成 3 个状态来控制这个过程，第一个是写寄存器状态（WR_REG），这里面有这个写数据操作的四个 task（第三行的 task 是为了兼容 2 字节的地址，实际执行的时候在等待写寄存器完成状态（WAIT_WR_DONE）里面通过 cnt 的值，在这个状态里面跳过了），第二个是等待写寄存器完成状态（WAIT_WR_DONE），这里面是等待每一次 task 执行完成，然后控制状态跳转回去继续执行下一行的 task，第三个是写寄存器完成状态（WR_REG_DONE），这个状态是向外反馈一个读写操作完成（RW_Done）状态。代码如下：

```
WR_REG:
    begin
        state <= WAIT_WR_DONE;
        case(cnt)
            0:write_byte(WR | STA, device_id);
            1:write_byte(WR, reg_addr[15:8]);
            2:write_byte(WR, reg_addr[7:0]);
            3:write_byte(WR | STO, wrdata);
            default:;
        endcase
end
```

每写入一次数据就会跳转到等待写完成状态，同时在这个状态里面会根据 addr_mode 来确定写的是 8 位地址，还是 16 位的地址，从而来控制 cnt 的值，后面回到写寄存器状态（WR_REG）里面的时候就可以选择性跳过发送高 8 位全 0 来代替的地址了。代码如下：

```
WAIT_WR_DONE:
    begin
        Go <= 1'b0;
        if(Trans_Done)begin
            ack <= ack | ack_o;
            case(cnt)
                0: begin cnt <= 1; state <= WR_REG;end
                1: begin
                        state <= WR_REG;
```

```
                        if(addr_mode)
                            cnt <= 2;
                        else
                            cnt <= 3;
                    end
                2: begin
                        cnt <= 3;
                        state <= WR_REG;
                    end
                3:state <= WR_REG_DONE;
                default:state <= IDLE;
            endcase
        end
    end
```

当写完数据后就会跳转到写完成状态，将 RW_Done 拉高，同时跳转到空闲状态，代码如下：

```
WR_REG_DONE:
    begin
        RW_Done <= 1'b1;
        state <= IDLE;
end
```

读和写操作都有了，那么我们再定义个空闲状态，这里面再引入两个信号写请求（wrreg_req）、读请求（rdreg_req），在空闲状态下根据来的是读请求还是写请求从而跳转到对应的状态下来执行，同时在这里面将一些标志信号（读写操作完成的标志信号）恢复成默认状态。

在空闲状态下收到写数据（wrreg_req）请求或者读数据（rdreg_req）请求后会根据这两个请求跳转到对应的状态。空闲状态代码如下：

```
IDLE:
    begin
        cnt <= 0;
        ack <= 0;
        RW_Done <= 1'b0;
        if(wrreg_req)
            state <= WR_REG;
        else if(rdreg_req)
            state <= RD_REG;
        else
            state <= IDLE;
end
```

这里把上面用到的几个状态都定义成参数，状态如下：

```
localparam
    IDLE            = 7'b0000001,   //空闲状态
    WR_REG          = 7'b0000010,   //写状态
```

```
    WAIT_WR_DONE     = 7'b0000100,   //等待写完成状态
    WR_REG_DONE      = 7'b0001000,   //写完成状态
    RD_REG           = 7'b0010000,   //读状态
    WAIT_RD_DONE     = 7'b0100000,   //等待读完成状态
    RD_REG_DONE      = 7'b1000000;   //读完成状态
```

好了到了这里 i2c_control 模块就基本完成了，我们来总结一下这里面的几个用到的输入输出端口信号的功能，如下表 29-2 所示：

表 29-2 i2c_comtrol 模块接口功能描述

接口名称	I/O	功能描述
Clk	I	模块工作时钟，50M 时钟
Rst_n	I	模块复位信号
wrreg_req	I	写请求信号
rdreg_req	I	读请求信号
addr	I	16 位地址输入
addr_mode	I	输入地址模式，0：8 位的地址，1：16 位地址
wrdata	I	总线发送的 8 位数据
rddata	O	总线收到的 8 位数据
device_id	I	i2c 器件的 ID
RW_Done	O	读/写完成标志
ack	O	从机是否应答标志
i2c_sclk	O	i2c 时钟总线
i2c_sdat	I /O	i2c 数据总线

同时也来捋一下这里面的几个状态之间的关系，画出对应的状态转移图，如下图 29-24 所示。

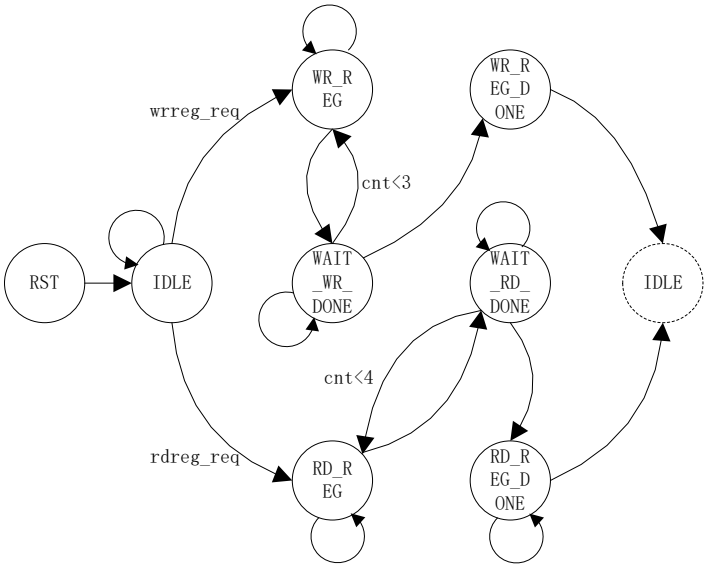


图 29-24 i2c_control 状态转移图

总的来说这个模块用起来还是灵活的，例如我们还是往 1 字节器件地址的

EEPROM 中的 0C 地址写数据 0A，这个时候我们只需要将 EEPROM 的器件地址给到 device_id，000C 给到 addr（1 字节的地址，高字节就用全 0 来代替），要写的数据 0A 给到 wrdata，地址输入模式 addr_mode 给 0，这个时候给个写请求信号（wrreg_req），在时钟的驱动下就可以将数据 0A 写到 1 字节器件地址的 EEPROM 中的 0C 地址的内存中路了。读数据的话也是一样的操作，此时无非就是将写请求（wrreg_req）换成读请求（rdreg_req）最后读出来的数据会在 rddata 端口上了。

29.9接口仿真验证

到此为止，整个 I2C 控制器的设计就完成了，上面说了这么多关于这个控制器的，接下来我们就拿一个 EEPROM 的仿真模型来测试一下这个控制器是否存在问题。本实验采用的是镁光官网提供的 EEPROM 仿真模型，读者可以从我们提供的例程代码中搜索：仿真模型 24LC04B.v 文件。

或者在芯路恒论坛搜索：

I2C 协议读写 EEPROM 存储器完整设计加仿真文件，并从工程中获取仿真模型。

http://www.corecourse.cn/forum.php?mod=viewthread&tid=27812

这里我们选用的是 1 字节寄存器地址段的 24LC04B 仿真模型，下图为仿真验证的结构图。

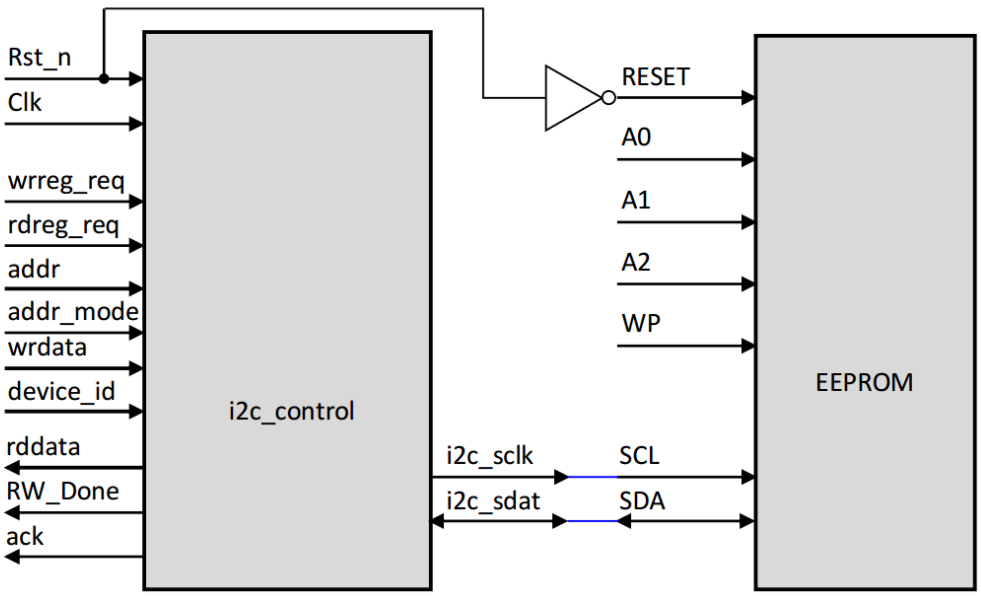


图 29-25 I²C 仿真控制验证结构图

仿真代码如下：

```
module i2c_control_tb;

reg Clk;
reg Rst_n;
reg wrreg_req;
reg rdreg_req;
reg [15:0]addr;
reg addr_mode;
reg [7:0]wrdata;
wire [7:0]rddata;
reg [7:0]device_id;
wire RW_Done;
wire ack;
wire i2c_sclk;
wire i2c_sdat;

pullup PUP(i2c_sdat);

i2c_control DUT(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .wrreg_req(wrreg_req),
    .rdreg_req(rdreg_req),
    .addr(addr),
    .addr_mode(addr_mode),
    .wrdata(wrdata),
    .rddata(rddata),
    .device_id(device_id),
    .RW_Done(RW_Done),
    .ack(ack),
    .i2c_sclk(i2c_sclk),
    .i2c_sdat(i2c_sdat)
);

M24LC04B M24LC04B(
    .A0(0),
    .A1(0),
    .A2(0),
    .WP(0),
    .SDA(i2c_sdat),
    .SCL(i2c_sclk),
    .RESET(~Rst_n)
);

initial Clk = 1;
always #10 Clk = ~Clk;
```

```
initial begin
    Rst_n = 0;
    rdreg_req = 0;
    wrreg_req = 0;
    #2001;
    Rst_n = 1;
    #2000;

    write_one_byte(8'hA0,8'h0A,8'hd1);
    write_one_byte(8'hA0,8'h0B,8'hd2);
    write_one_byte(8'hA0,8'h0C,8'hd3);

    read_one_byte(8'hA0,8'h0A);
    read_one_byte(8'hA0,8'h0B);
    read_one_byte(8'hA0,8'h0C);
    $stop;
end

task write_one_byte;
    input [7:0]id;
    input [7:0]mem_address;
    input [7:0]data;
    begin
        addr = {8'd0,mem_address};
        device_id = id;
        addr_mode = 0;
        wrdata = data;
        wrreg_req = 1;
        #20;
        wrreg_req = 0;
        @(posedge RW_Done);
        #20000;
    end
endtask

task read_one_byte;
    input [7:0]id;
    input [7:0]mem_address;
    begin
        addr = {8'd0,mem_address};
        device_id = id;
        addr_mode = 0;
        rdreg_req = 1;
        #20;
        rdreg_req = 0;
        @(posedge RW_Done);
```

```
        #20000;
    end
endtask

endmodule
```

这里面因为用到的仿真模型是 i2c 接口，是开漏结构不能直接输出高电平，所以也要和实际的一样，要接外部上拉电阻，前面也提到了，所以这里也要将 i2c_sdat 进行弱上拉处理，这里就用到了一个新的关键字 pullup，代码如下：

```
pullup PUP(i2c_sdat);
```

下图是对 24LC04B 型号 EEPROM 模型进行 3 次写操作，然后再 3 次读操作。

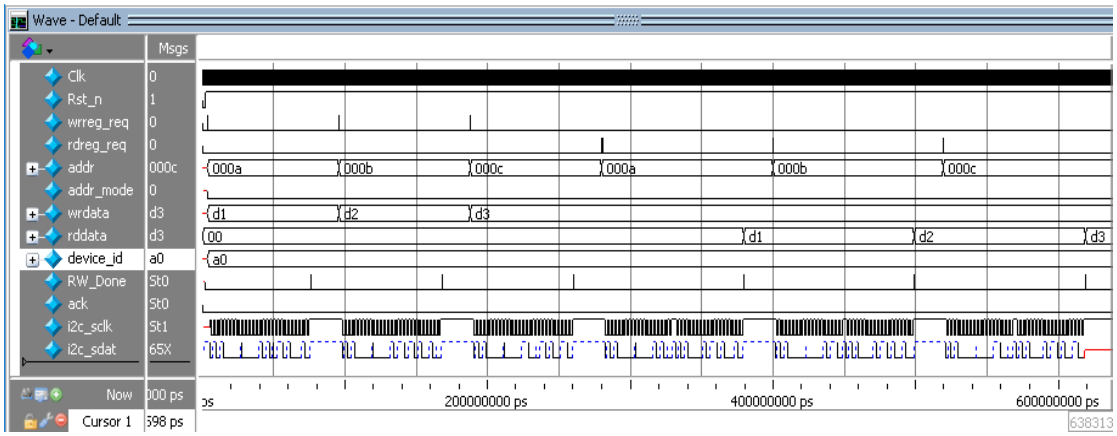


图 29-26 3 次写操作和 3 次读操作仿真波形图

写操作时序放大波形图如下：

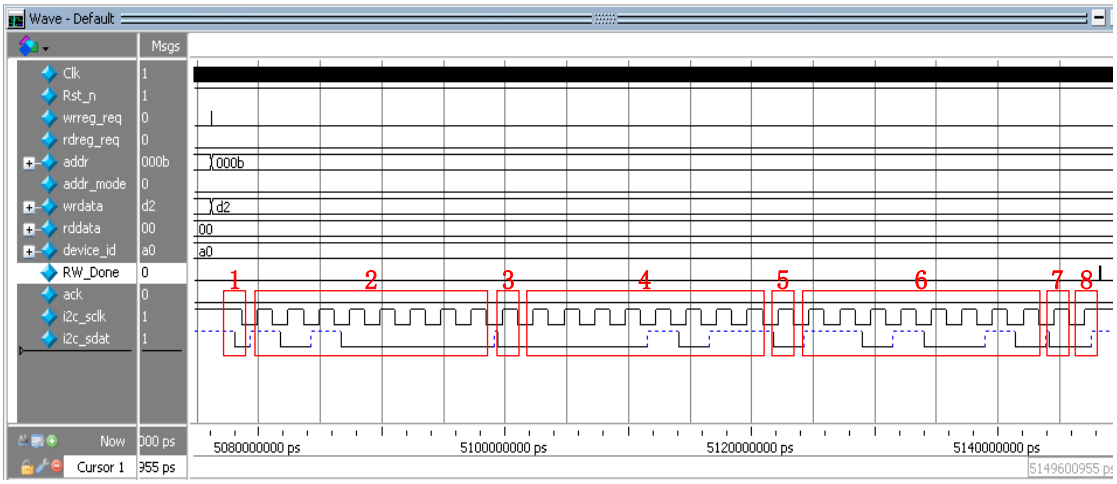


图 29-27 写操作时序放大波形图

从上图的波形中我们可以看出首先来一个写请求 wrreg_req 脉冲，然后 1 的位置就是个起始信号，2 的位置是器件地址 A0，后面 3 是 EEPROM 给的应答，4 是将要往 EEPROM 里面写数据的地址 0B，5 是是 EEPROM 给的应答，6 是写

的数据 D2，后面的 7 是 EEPROM 给的应答，8 是停止信号。完全符合协议标准。

读操作时序放大波形图如下：

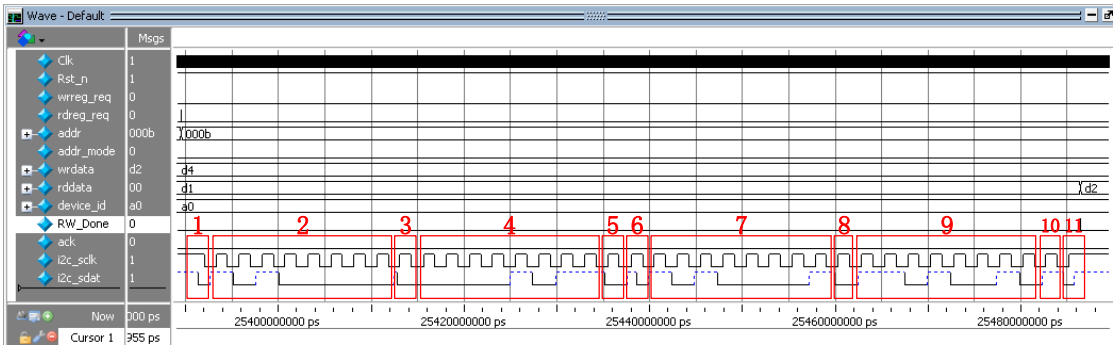


图 29-28 读操作时序放大波形图

从上图的波形中我们可以看出首先来一个读请求 rdreg_req 脉冲，然后 1 的位置就是个起始信号，2 的位置是器件地址 A0，后面 3 是 EEPROM 给的应答，4 是将要从 EEPROM 里面读数据的地址 0B，5 是 EEPROM 给的应答，6 是个起始信号，7 的位置是器件地址 A0，因为这次是要读操作，所以最低位是 1，所以最终就是 10100001（A1），8 是 EEPROM 给的应答，9 是读出来的数据 D2，后面的 10 是主机给 EEPROM 的无应答信号，11 是停止信号。此时后面的 RW_Done 有一个高脉冲信号指示，读数据操作完成，完全符合协议标准。

通过观察上面的读写时序波形发现，有一部分的波形是断断续续的蓝色虚线，这些位置正好是要输出高电平，这里是模拟外部电阻弱上拉，这里这个 EEPROM 的仿真模型与实际的器件是完全一样的。

通过仿真比较读写数据是一致的，验证了我们设计的 i2c 控制器是没有问题的，这样有关 i2c 控制器就完成了。

需要注意的是，根据器件手册的性能描述，EEPROM 的读操作和写操作之间，必须有至少 5ms 以上的时间间隔，否则有可能会造成读写失败。

29.10总结

本节课程无实际的 bit 文件生成和下载，仅需对我们的 verilog 代码进行相关的设计验证即可，同时我们在以前的章节中学习了如何让 vivado 和 modelsim 交互使用的基础上，进一步学习了如何在 modelsim 中调用相关器件模型 verilog 代码和我们的代码进行交互验证的方法。后续章节，我们将结合具体应用，利用本章学习到的 i2c 器件操作知识，进行数字钟显示和校准实验。

30 基于 I²C 接口的串口读写 EEPROM

工程源码	|----02_设计实例 |----ch30_uart_eeprom
相关视频课程	
说明	

章节导读

EEPROM（Electrically Erasable Programmable read only memory），即带电可擦可编程只读存储器，是一种掉电后数据不丢失的存储芯片。用户可以通过高于普通电压的作用来擦除和重编程（重写）。

在高云开发板上设计有一个型号为 24LC64 的 EEPROM 芯片，该芯片是合泰公司推出的一款 64k 存储大小的非易失性 I²C 存储器设备。本章将使用上一章节设计的 I²C 控制器，结合前面设计的串口收发模块实现串口对 EEPROM 存储器的读写功能，并进行相应的板级验证。

30.124LC64 功能描述

24LC64 是一种两线制串行读/写非易失性 I²C 存储器设备，器件名称中的 64 代表了其存储大小为 64Kbit。在器件内部中，每个存储单元能存储 8 位，因此器件内部共有 8k 个存储单元。要想为这 8K 个寄存器分配地址，就需要使用 13 位数据表示，因此该器件为 2 字节地址段器件。

器件内部将每 32 个字节划分为一页，数据在进行连续多字节读写时无法超出当前页地址。因此在使用 IIC 的连续多字节读写时，一次最多只能读写 32 个字节数据，而这种读写方式在器件中又被称为页写入和页读取。

要了解这种读写方式，首先需要了解 24LC64 的内部结构，其结构框图如图 30-1 所示。

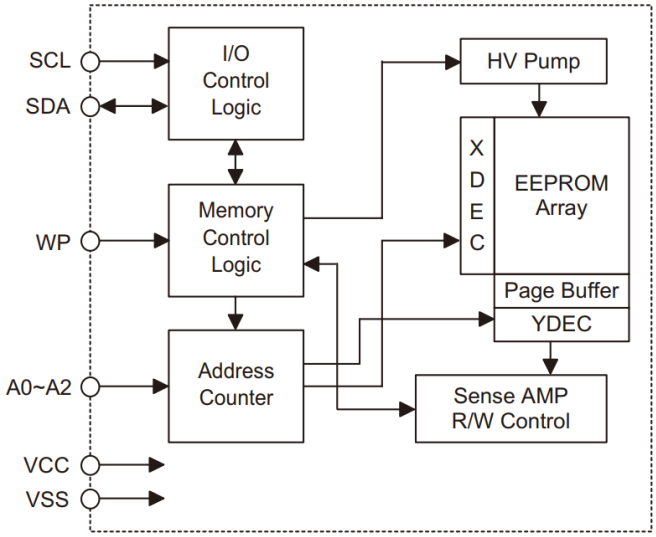


图 30-1 24LC64 结构框图

该器件使用 I²C 接口，用户通过 SCL 与 SDA 总线对器件进行读写。其中的寄存器控制逻辑（Memory Control Logic）负责将数据写入/读出 EEPROM，待写入的数据通过 HV Pump 的高压作用写入 EEPROM 阵列的指定存储单元。为了实现页读写功能，内部的 Page Buffer 会对存储器地址的高 8 位进行缓存，用以确定当前操作所读写的页。器件内部的地址计数器（Address Counter）会记录上一次操作的地址，用以在页读写模式下的地址自加等操作。当页读写地址超过当前页时，由于 Page Buffer 中的数据不会变化，地址只能在当前页范围内变化，因此读写操作会从当前页的起始地址继续执行。为了降低读出延时，读操作由锁存型读出放大器读/写控制逻辑（Sense AMP R/W Control）负责，在数据读出后送给寄存器控制逻辑最后通过 I²C 总线输出。

30.2器件地址

每个 I²C 器件都有自己的器件地址，对于 24LC64 这样一颗 EEPROM 器件，其器件地址由 1010 +3 位片选信号（对应图 30-1 中 A0~A2）组成，如图 30-2 所示：

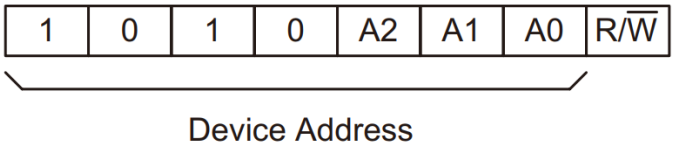


图 30-2 24LC64 器件地址

这三位片选信号由硬件上 A2、A1、A0 三个引脚的连接设定，也就是可以

通过对 A2、 A1、 A0 三个引脚物理接地或者上拉控制器件地址。在高云开发板上，24LC64 的相关电路如图 30-3 所示：

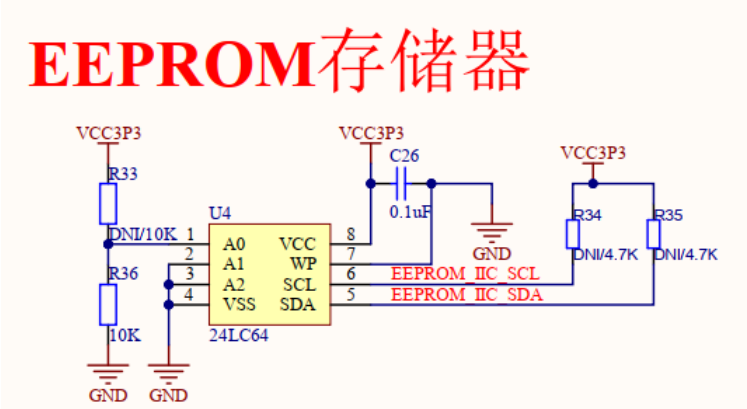


图 30-3 24LC64 硬件电路

可以看到， A2、A1、A0 引脚物理接地，因此开发板上 EEPROM 芯片的器件地址为 7 位的 1010000。但是在大多数情况下我们习惯加上一个读写位 0 来表示器件地址，也就是 8 位的 10100000，即 0xA0。

30.3串口读写 EEPROM 系统设计

在了解了 24LC64 的器件地址以及结构原理后，接下来我们就可以针对 24LC64 进行本次的串口读写 EEPROM 设计了。要达成设计目的，我们就需要使用到串口，利用 PC 机的串口向 FPGA 开发板发送读或写指令来对 EEPROM 器件进行数据的读或写。首先，我们规定读写 EEPROM 的串口指令如图 30-4 所示：

起始地址	功能码	数据0	.....	数据15

图 30-4 读写 EEPROM 器件指令

指令各部分所代表的功能含义如下：

1. 起始地址：读写数据的第一个地址。
2. 功能码：控制读写操作以及读写字节数。功能码的 bit[7]用于区分是写数据操作还是读数据操作，当值为 0 时代表写操作，值为 1 时代表读操作；bit[3:0]用于控制读写操作时，写入\读出 EEPROM 的有效字节数。
3. 数据 0~15：写操作时待写入数据，写操作时该部分必须填充 16 字节数据，但是用户可以通过功能码的 bit[3:0]控制实际写入 EEPROM 的字节

数。读操作时该部分可以填充任意值。

如图 30-5 为本次设计的整体框图。

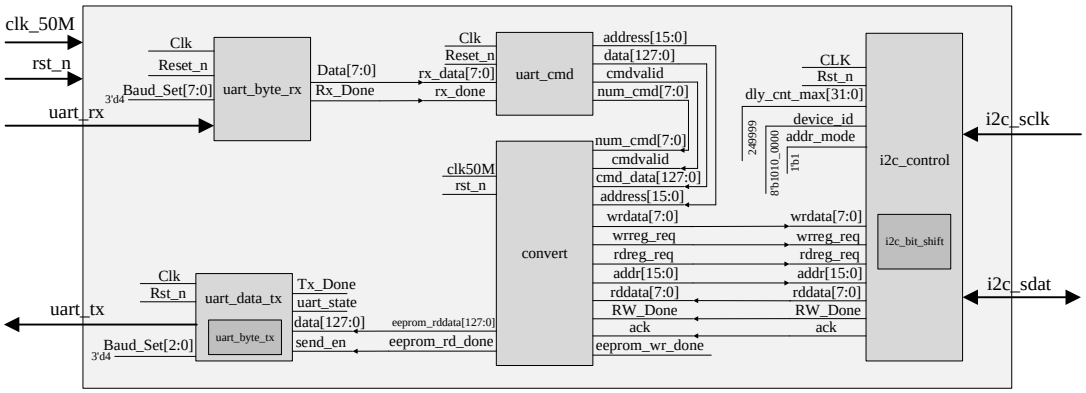


图 30-5 串口读写 EEPROM 整体设计框图

在前面章节中我们已经带大家完成过 uart_byte_rx、i2c_control、uart_byte_tx 模块的设计，因此这里主要介绍 uart_cmd、convert、uart_data_tx 三个模块，它们的功能如下:

1. uart_cmd：串口指令模块。该模块会判断从串口接收模块中接收的数据。当数据满足串口指令时，模块会从中提取出对应的起始地址、功能码以及数据指令，并传输给 convert 模块。

2. convert：指令转读写操作模块。该模块会接收从 uart_cmd 传输来的指令，并根据指令中的功能码判断当前的读/写操作类型以及实际读/写字节数。随后通过内部状态机产生相关读写请求信号，与起始地址和数据一起传输给 i2c_control 模块，实现对 EEPROM 的读/写操作。

3. uart_data_tx：多字节串口发送模块。在前面章节中，我们设计的串口发送模块单次只支持 1 字节的数据发送，而本次设计中我们一次可能需要发送至多 16 字节数据，为此我们为串口发送模块设计了一个上层封装模块，通过状态机使得串口发送能够实现 1~16 字节数据的发送、大小端选择等等。

系统在工作时，串口接收模块（uart_byte_rx）会接收用户传输的数据，并交由串口指令模块（uart_cmd）。串口指令模块会根据我们定义的串口指令对内容进行提取，得到对应的起始地址、功能码和 26 字节数据，随后将它们送到指令转读写操作模块（convert）进行解析判断，依据判断得到的操作模式，产生对应的读写请求信号，并将地址与数据等信息交由 I2C 控制器（i2c_control），实现对指定地址的读写操作。如果是读操作，读出的数据会被交给多字节串口

发送模块（uart_data_tx），该模块能够支持最多一次传输 16 字节的数据，最终将读出的数据发送给电脑的串口。

接下来就对各个模块进行详细介绍，并带大家完成对应模块的设计。

30.3.1uart_cmd 模块

该模块负责串口指令的解析，依据前面我们定义的串口指令，正确的格式是起始地址+功能码+16 字节数据。而在前面介绍 24LC64 时我们分析过，该器件为 2 字节地址器件，因此起始地址需要占 2 字节，功能占 1 字节，数据占 16 字节。所以，正确的串口指令需要占 19 字节。为了保证所接收到的 19 字节内容就是串口指令，我们可以再为其加上 2 字节的帧头 0x55 和 0xA5，以及 1 字节的帧尾 0xF0，作为我们的串口指令协议。因此，协议的完整组成如下图 30-6 所示。

0x55	0xA5	起始地址	功能码	数据0~15	0xF0

图 30-6 串口指令协议

串口指令协议的长度为 22 字节，我们所需做的就是对串口接收的数据进行 22 字节的缓存（打拍），随后对这 22 字节数据进行判断。只要满足串口指令协议要求（即帧头 0x55+帧头 0xA5+19 字节内容+帧尾 0xF0），便代表是正确的串口指令。随后从指令中提取出对应的起始地址、功能码、数据，并产生 cmd 有效信号，一起发送给指令转读写操作模块。因此，uart_cmd 的设计代码如下：

```
`timescale 1ns/1ns

module uart_cmd(
    Clk,
    Reset_n,
    rx_data,
    rx_done,
    address,
    data,
    num_cmd,
    cmdvalid
);

    input Clk;
    input Reset_n;
    input [7:0]rx_data;
    input rx_done;
```

```
    output reg[7:0]address;
    output reg[127:0]data;
    output reg cmdvalid;
    output reg [7:0]num_cmd;

    reg [7:0] data_str [20:0];
    always@(posedge Clk)
    if(rx_done)begin
        data_str[20] <= #1 rx_data;
        data_str[19] <= #1 data_str[20];
        data_str[18] <= #1 data_str[19];
        data_str[17] <= #1 data_str[18];
        data_str[16] <= #1 data_str[17];
        data_str[15] <= #1 data_str[16];
        data_str[14] <= #1 data_str[15];
        data_str[13] <= #1 data_str[14];
        data_str[12] <= #1 data_str[13];
        data_str[11] <= #1 data_str[12];
        data_str[10] <= #1 data_str[11];
        data_str[9] <= #1 data_str[10];
        data_str[8] <= #1 data_str[9];
        data_str[7] <= #1 data_str[8];
        data_str[6] <= #1 data_str[7];
        data_str[5] <= #1 data_str[6];
        data_str[4] <= #1 data_str[5];
        data_str[3] <= #1 data_str[4];
        data_str[2] <= #1 data_str[3];
        data_str[1] <= #1 data_str[2];
        data_str[0] <= #1 data_str[1];
    end

    reg r_rx_done;
    always@(posedge Clk)
        r_rx_done <= rx_done;

    always@(posedge Clk or negedge Reset_n)
    if(!Reset_n) begin
        address <= #1 0;
        data <= #1 0;
        cmdvalid <= #1 0;
        num_cmd <= #1 0;
    end else if(r_rx_done)begin
        if((data_str[0] == 8'h55) && (data_str[1] == 8'hA5) &&
(data_str[20] == 8'hF0))begin
            data <= #1
{data_str[4],data_str[5],data_str[6],data_str[7],data_str[8],
data_str[9],data_str[10],data_str[11],data_str[12],data_str[13],
```

```
data_str[14],data_str[15],data_str[16],data_str[17],data_str[18],
                        data_str[19]};
            num_cmd <= #1 data_str[3];
            address <= #1 data_str[2];
            cmdvalid <= #1 1;
        end
    end
    else
        cmdvalid <= #1 0;

endmodule
```

30.3.2convert 模块

convert 模块负责指令到读写操作的转换，当来自 uart_cmd 的 cmdvalid 信号有效的时候，convert 模块会根据功能码判断当前的操作模式以及实际读写的数据量。如果是写操作，则将地址和待写入数据发送给前面章节中我们设计好的 I^2C 控制器。

为了让数据能够成功被 I^2C 控制器接收，需要在传输每字节数据时，产生一个写请求信号给 I^2C 控制器。同时，因为 I^2C 控制器每次只支持单个字节的收发，所以我们需要在每写完 1 字节数据后等待 I^2C 控制器的应答信号，以确保数据成功写入。

如果是读操作，除了将地址传输给 I^2C 控制器外，在读取每 1 字节数据之前，都需要产生一次读请求信号。在 I^2C 控制器对读请求信号进行响应后才能读去这 1 字节的数据，如此循环，直到读取完所需的数据量，并最终输出给多字节串口发送模块。

很显然，要实现以上这些功能，最适合的方式无疑是状态机。这里我们可以定义 IDLE、DO_WR、WAIT_WR_DONE、DO_RD、WAIT_RD_DONE 五个状态。让 IDLE 态等待 cmdvalid 信号，判断读写操作； DO_WR 态产生写请求信号、数据和地址； WAIT_WR_DONE 态判断写传输是否完成； DO_RD 态产生读请求信号与地址； WAIT_RD_DONE 态读取数据，并判断当前读操作是否成功。来实现以上设计目的。因此，convert 模块的完整代码如下：

```
module convert(
    clk50M,
    rst_n,
    rddata,
    RW_Done,
    ack,
    address,
```

```
    cmd_data,
    num_cmd,
    cmdvalid,

    eeprom_rddata,
    wrdata,
    wrreg_req,
    rdreg_req,
    addr,
    eeprom_rd_done,
    eeprom_wr_done
    );

     input clk50M;
     input rst_n;
     input [7:0]rddata;
     input RW_Done;
     input ack;
     input [15:0]address;
     input [127:0]cmd_data;
     input [7:0]num_cmd;
     input cmdvalid;

     output reg[127:0]eeprom_rddata;
     output reg[7:0]wrdata;
     output reg wrreg_req;
     output reg rdreg_req;
     output reg[15:0]addr;
     output reg eeprom_rd_done;
     output reg eeprom_wr_done;

    reg eeprom_rd_err,eeprom_wr_err;

     reg [2:0]state;
    reg [6:0]data_cnt;

    localparam
        IDLE = 0,
        DO_WR = 1,
        WAIT_WR_DONE  = 2,
        DO_RD = 3,
        WAIT_RD_DONE = 4;

    always@(posedge clk50M or negedge rst_n)
    if(!rst_n)begin
        state <= IDLE;
        wrreg_req <= 1'd0;
```

```
        addr <= 16'd0;
        wrdata <= 8'd0;
        data_cnt <= 7'd0;
        eeprom_wr_err <= 1'd0;
        eeprom_wr_done <= 1'd0;
        eeprom_rd_done <= 1'd0;
        eeprom_rd_err <= 1'd0;
        rdreg_req <= 1'd0;
        eeprom_rddata <= 128'd0;
    end
    else begin
    case(state)
        IDLE:
            begin
                wrreg_req <= 1'd0;
                addr <= 16'd0;
                wrdata <= 8'd0;
                data_cnt <= 7'd0;
                eeprom_wr_err <= 1'd0;
                eeprom_rd_done <= 1'd0;
                eeprom_rd_err <= 1'd0;
                rdreg_req <= 1'd0;
                if(cmdvalid && !num_cmd[7])
                    state <= DO_WR;
                else if(cmdvalid && num_cmd[7])begin
                    state <= DO_RD;
                    eeprom_rddata <= 128'd0;
                end
                else
                    state <= IDLE;
            end

        DO_WR:
            begin
                wrreg_req <= 1'd1;
                addr <= data_cnt + address;
                state <= WAIT_WR_DONE;
                case(data_cnt)
                    0:wrdata <= cmd_data[127:120];
                    1:wrdata <= cmd_data[119:112];
                    2:wrdata <= cmd_data[111:104];
                    3:wrdata <= cmd_data[103:96];
                    4:wrdata <= cmd_data[95:88];
                    5:wrdata <= cmd_data[87:80];
                    6:wrdata <= cmd_data[79:72];
                    7:wrdata <= cmd_data[71:64];
                    8:wrdata <= cmd_data[63:56];
```

```
                9:wrdata <= cmd_data[55:48];
                10:wrdata <= cmd_data[47:40];
                11:wrdata <= cmd_data[39:32];
                12:wrdata <= cmd_data[31:24];
                13:wrdata <= cmd_data[23:16];
                14:wrdata <= cmd_data[15:8];
                15:wrdata <= cmd_data[7:0];
                default:wrdata <= 0;
            endcase
        end

    WAIT_WR_DONE:
        begin
            wrreg_req <= 1'd0;
            if(RW_Done)begin
                if(!ack)begin   //ACK 为 0 表示有正常应答
                    if(data_cnt >= num_cmd - 1)begin
                        state <= IDLE;
                        eeprom_wr_done <= 1'd1;
                        data_cnt <= 0;
                    end
                    else begin
                        state <= DO_WR;
                        data_cnt <= data_cnt + 1'd1;
                    end
                end
                else begin  //ACK 为 1 表示无应答
                    state <= IDLE;
                    eeprom_wr_err <= 1'd1;
                end
            end
            else begin
                state <= WAIT_WR_DONE;
                data_cnt <= data_cnt;
            end
        end

    DO_RD:
        begin
            rdreg_req <= 1'd1;
            state <= WAIT_RD_DONE;
            addr <= data_cnt + address;
        end

    WAIT_RD_DONE:
        begin
            rdreg_req <= 1'd0;
```

```
                if(RW_Done)begin
                    if(!ack)begin   //ACK 为 0 表示有正常应答
                        case(data_cnt)
                            0:eeprom_rddata[127:120] <= rddata;
                            1:eeprom_rddata[119:112] <= rddata;
                            2:eeprom_rddata[111:104] <= rddata;
                            3:eeprom_rddata[103:96] <= rddata;
                            4:eeprom_rddata[95:88] <= rddata;
                            5:eeprom_rddata[87:80] <= rddata;
                            6:eeprom_rddata[79:72] <= rddata;
                            7:eeprom_rddata[71:64] <= rddata;
                            8:eeprom_rddata[63:56] <= rddata;
                            9:eeprom_rddata[55:48] <= rddata;
                            10:eeprom_rddata[47:40] <= rddata;
                            11:eeprom_rddata[39:32] <= rddata;
                            12:eeprom_rddata[31:24] <= rddata;
                            13:eeprom_rddata[23:16] <= rddata;
                            14:eeprom_rddata[15:8] <= rddata;
                            15:eeprom_rddata[7:0] <= rddata;
                            default:;
                        endcase
                        if(data_cnt >= (num_cmd & 8'h7F) - 1)begin
                            state <= IDLE;
                            eeprom_rd_done <= 1'd1;
                            data_cnt <= 0;
                        end
                        else begin
                            state <= DO_RD;
                            data_cnt <= data_cnt + 1'd1;
                        end
                    end
                    else begin  //ACK 为 1 表示无应答
                        state <= IDLE;
                        eeprom_rd_err <= 1'd1;
                    end
                end
                else begin
                    state <= WAIT_RD_DONE;
                    data_cnt <= data_cnt;
                end
            end

            default:state <= IDLE;
        endcase
    end

endmodule
```

30.3.3uart_data_tx 模块

uart_data_tx 模块是单字节串口发送模块（uart_byte_tx）的上层封装模块，用以实现多字节的串口收发。对于设计来说，我们所需要做的就是将来自 convert 模块的多字节数据，单个字节地传输给单字节串口发送模块即可，因此，该模块的代码如下：

```
module uart_data_tx(
    Clk,
    Rst_n,

    data,
    send_en,
    Baud_Set,

    uart_tx,
    Tx_Done,
    uart_state
);

    parameter DATA_WIDTH = 8;
    parameter MSB_FIRST = 1;

    input Clk;
    input Rst_n;

    input [DATA_WIDTH - 1 : 0]data;
    input send_en;
    input [2:0]Baud_Set;
    output uart_tx;
    output reg Tx_Done;
    output uart_state;

    reg [DATA_WIDTH - 1 : 0]data_r;

    reg [7:0] data_byte;
    reg byte_send_en;
    wire byte_tx_done;

    uart_byte_tx uart_byte_tx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .data_byte(data_byte),
        .send_en(byte_send_en),
        .Baud_Set(Baud_Set),
        .uart_tx(uart_tx),
```

```
        .Tx_Done(byte_tx_done),
        .uart_state(uart_state)
    );

    reg [8:0]cnt;
    reg [1:0]state;

    localparam S0 = 0;  //等待发送请求
    localparam S1 = 1;  //发起单字节数据发送
    localparam S2 = 2;  //等待单字节数据发送完成
    localparam S3 = 3;  //检查所有数据是否发送完成

    always@(posedge Clk or negedge Rst_n)
    if(!Rst_n)begin
        data_byte <= 0;
        byte_send_en <= 0;
        state <= S0;
        cnt <= 0;
    end
    else begin
        case(state)
            S0:
                begin
                    data_byte <= 0;
                    cnt <= 0;
                    Tx_Done <= 0;
                    if(send_en)begin
                        state <= S1;
                        data_r <= data;
                    end
                    else begin
                        state <= S0;
                        data_r <= data_r;
                    end
                end

            S1:
                begin
                    byte_send_en <= 1;
                    if(MSB_FIRST == 1)begin
                        data_byte<= data_r[DATA_WIDTH-1:DATA_WIDTH-8];
                        data_r <= data_r << 8;
                    end
                    else begin
                        data_byte <= data_r[7:0];
                        data_r <= data_r >> 8;
                    end
```

```
                    state <= S2;
                end

            S2:
                begin
                    byte_send_en <= 0;
                    if(byte_tx_done)begin
                        state <= S3;
                        cnt <= cnt + 9'd8;
                    end
                    else
                        state <= S2;
                end

            S3:
                if(cnt >= DATA_WIDTH)begin
                    state <= S0;
                    cnt <= 0;
                    Tx_Done <= 1;
                end
                else begin
                    state <= S1;
                    Tx_Done <= 0;
                end
            default:state <= S0;
        endcase
    end

endmodule
```

代码中例化了前面章节设计的单字节发送模块，通过状态机将数据单字节地传输给单字节发送模块，实现多字节的串口发送。除此之外，模块还支持自定义位接口位宽以满足不同位宽大小的数据输入、支持大小端转换。

30.3.4顶层封装设计

在设计完各个模块后，最后一步便是顶层封装。在对各个模块进行例化时，对于部分端口，我们可以直接赋定值。例如，串口收发模块的波特率设置端口，可以设置为常用的 115200；I^2C 控制器的器件 id 端口，可以直接设置为 EEPROM 的器件地址 8'b10100000。

以下便是本次设计的顶层封装设计：

```
//板级运行时请将下属代码注释以屏蔽，仿真时将下属代码取消注释以生效
`define DO_SIM 1
module uart_eeprom(
```

```
    clk50M,
    rst_n,
    i2c_sclk,
    i2c_sdat,
    uart_rx,
    uart_tx
);
    input clk50M;
    input rst_n;
    inout i2c_sdat;
    output i2c_sclk;
    input uart_rx;
    output uart_tx;

    wire [15:0] addr;
    wire ack;
    wire RW_Done;

    wire eeprom_rd_done;
    wire eeprom_wr_done;

    wire [127:0]eeprom_rddata;
    wire wrreg_req;
    wire rdreg_req;
    wire [7:0]wrdata;
    wire [7:0]rddata;

    wire [15:0]address;
    wire [127:0]cmd_data;
    wire [7:0]num_cmd;
    wire cmdvalid;


     convert convert(
          .clk50M(clk50M),
          .rst_n(rst_n),
          .rddata(rddata),
          .RW_Done(RW_Done),
          .ack(ack),
          .address(address),
          .cmd_data(cmd_data),
          .num_cmd(num_cmd),
          .cmdvalid(cmdvalid),
          .eeprom_rddata(eeprom_rddata),
          .wrdata(wrdata),
          .wrreg_req(wrreg_req),
          .rdreg_req(rdreg_req),
```

```
        .addr(addr),
        .eeprom_rd_done(eeprom_rd_done),
        .eeprom_wr_done(eeprom_wr_done)
    );


    i2c_control i2c_control(
        .Clk(clk50M),
        .Rst_n(rst_n),

        .wrreg_req(wrreg_req),
        .rdreg_req(rdreg_req),
        .addr(addr),
        .addr_mode(1'b1),
        .wrdata(wrdata),
        .rddata(rddata),
        .device_id(8'b1010_0000),
        .RW_Done(RW_Done),
        .ack(ack),
    `ifdef DO_SIM
        .dly_cnt_max(250-1),
    `else
        .dly_cnt_max(250000-1),
    `endif
        .i2c_sclk(i2c_sclk),
        .i2c_sdat(i2c_sdat)
    );

    wire rx_done;
    wire [7:0]rx_data;
    uart_byte_rx uart_byte_rx(
        .Clk(clk50M),
        .Reset_n(rst_n),
        .Baud_Set(3'd4),
        .uart_rx(uart_rx),
        .Data(rx_data),
        .Rx_Done(rx_done)
    );

    uart_cmd uart_cmd(
        .Clk(clk50M),
        .Reset_n(rst_n),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .address(address),
        .data(cmd_data),
        .num_cmd(num_cmd),
```

```
        .cmdvalid(cmdvalid)
    );

    uart_data_tx
    #(
        .DATA_WIDTH(128),
        .MSB_FIRST(1)
    )
    uart_data_tx(
        .Clk(clk50M),
        .Rst_n(rst_n),
        .data(eeprom_rddata),
        .send_en(eeprom_rd_done),
        .Baud_Set(3'd4),
        .uart_tx(uart_tx),
        .Tx_Done(),
        .uart_state()
    );
endmodule
```

考虑到串口最多一次需要发送 16 字节数据，在例化多字节串口发送模块时，数据位宽被设置为了 128（16*8）。即无论用户输入的功能码中无论读操作的有效位是多少，多字节串口发送模块都会发送 16 字节数据，多余的字节将会以 00 表示。

同时，需要注意的是，一般情况下 eeprom 存储硬件在两次操作间都需要有 5ms 的间隔（这也是为什么在 I^2C 控制器模块的状态机中会有一个 5ms 的延时状态）。顶层中该值通过 I^2C 控制器的 dly_cnt_max 端口设置，而为了节省仿真所消耗的时间，代码中通过条件编译 DO_SIM，为仿真专门设置了一个较小的值。实际使用时，用户需要根据具体应用场景考虑是否注释 DO_SIM。

30.4串口读写 EEPROM 仿真验证

在完成了串口读写 EEPROM 系统设计后，接下来编写 testbench 测试文件对设计的整个系统进行仿真验证， 整个仿真主要是通过串口发送模块模拟对该系统发送指令进行仿真验证，这里就只选用 M24LC64（与开发板的 EEPROM 器件相对应）这个仿真模型进行仿真， M24LC04 的仿真模型类似（读者可尝试对该模型进行仿真），编写的 testbench 文件具体代码如下：

```
`timescale 1ns/1ns
module uart_eeprom_tb();

    wire i2c_sclk;
```

```
    wire i2c_sdat;
    reg clk50M;
    reg rst_n;
    wire uart_tx;
    wire uart_rx;
    wire Tx_Done;
    wire uart_tx_t;
    assign uart_rx = uart_tx_t;

    reg send_en;

    pullup PUP(i2c_sdat);

    uart_eeprom uart_eeprom(
        .clk50M(clk50M),
        .rst_n(rst_n),
        .i2c_sclk(i2c_sclk),
        .i2c_sdat(i2c_sdat),
        .uart_rx(uart_rx),
        .uart_tx(uart_tx)
    );
    M24LC64 M24LC64(
        .A0(0),
        .A1(0),
        .A2(0),
        .WP(0),
        .SDA(i2c_sdat),
        .SCL(i2c_sclk),
        .RESET(~rst_n)
    );

    initial clk50M = 1;
    always #10 clk50M = ~clk50M;

    reg [175:0] test_data;


    initial begin
       rst_n = 0;
       #201;
       rst_n = 1;
       #200;
       test_data=176'h55_A5_00_03_04_12_34_56_78_9A_BC_DE_F0_23_45_67_
89_AB_CD_EF_34_F0;
       send_en = 1;
       #20;
       send_en = 0;
```

```
        #200;
        @(posedge uart_eeprom.eeprom_wr_done);

        #201;
        test_data=176'h55_A5_00_04_83_12_34_56_78_9A_BC_DE_F0_23_45_67_
89_AB_CD_EF_34_F0;
        send_en = 1;
        #20;
        send_en = 0;
        #200;
        @(posedge uart_eeprom.eeprom_rd_done);
        #2000;
        @(posedge uart_eeprom.uart_data_tx.Tx_Done);
        #100000;
        $stop;
    end

    uart_data_tx
    #(
        .DATA_WIDTH(176),
        .MSB_FIRST(1)
    )
    uart_data_tx(
        .Clk(clk50M),
        .Rst_n(rst_n),
        .data(test_data),
        .send_en(send_en),
        .Baud_Set(3'd4),
        .uart_tx(uart_tx_t),
        .Tx_Done(Tx_Done),
        .uart_state()
    );

endmodule
```

仿真过程主要是通过串口发送模块模拟对该系统发送读写指令进行仿真验证，依次进行了 1 次写和读指令的发送。写数据指令中，起始地址为 0x0003，功能码为 0x04，因此虽然输入了 16 个写数据，但是有效写入的只有前四个数据，即 0x12、0x34、0x56、0x78。

读数据指令中，起始地址为 0x0004，功能码为 0x83，因此只会从 EEPROM 中读取 3 字节数据，即 0x34、0x56、0x78。串口一共会输出 16 字节数据，因此其余位全部发 0。

图 30-7 为系统仿真波形时序图，为了方便观察，添加了源设计工程中，uart_byte_tx 模块、uart_byte_rx 模块和 convert 模块的部分中间信号

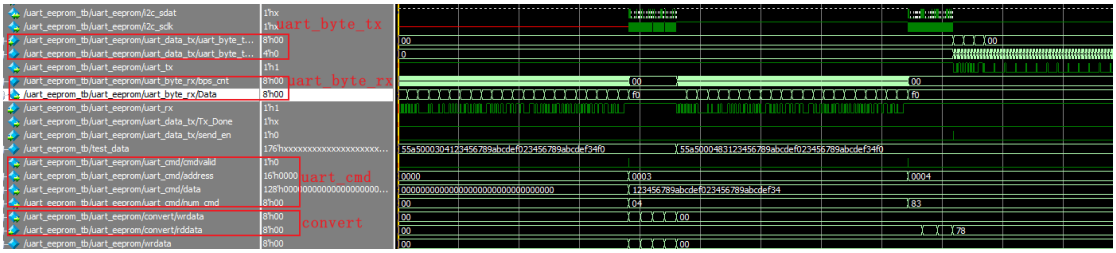


图 30-7 仿真总体波形图

接下来我们将波形放大，第一条指令的仿真波形如下图 30-8 所示。

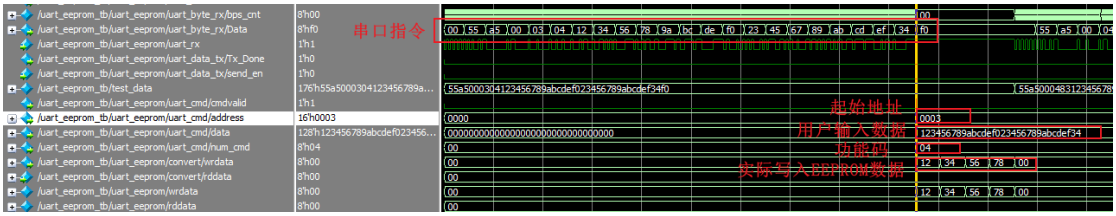


图 30-8 写指令仿真波形

从上述两幅图中可以看出，写指令被窜口完全接收，在进行帧头帧尾检验正确后，指令被拆分为起始地址、功能码以及用户数据。功能码 bit[6:0]的值为 4，因此写入的字节数为四个，最终写入的是 0x12、0x34、0x56、0x78，与预计中的一致，说明写指令功能成功。

第二条指令的仿真波形时序如下图 30-9、图 30-10 所示。

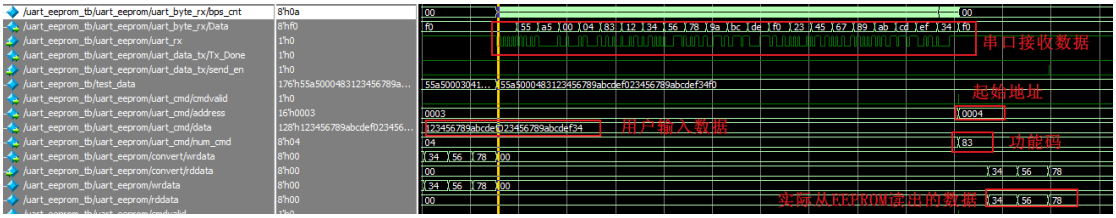


图 30-9 读指令仿真时序（串口接收到写入 EEPROM）

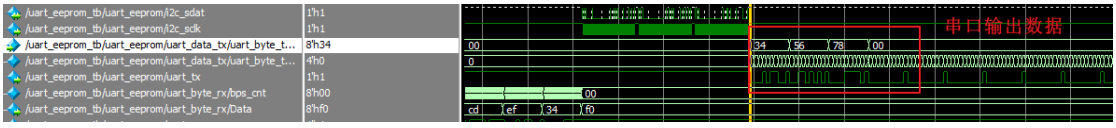


图 30-10 读指令仿真时序（数据读出 EEPROM 到串口发出）

同样的，我们可以看到第二条指令被成功识别，并根据功能码识别为读操作。功能码的 bit[6:0]值为 3，因此从地址 0x004 开始连续读取 3 字节数据，即 0x34、0x56、0x78。这些数据会被送到多字节串口发送模块，最终发送的数据是 0x34、0x56、0x78 以及 13 字节的 0 。可以看到与预期中的一致，说明读指令功能也成功。

综上所述，整个设计的仿真功能验证成功，设计在仿真中能够很好的实现

预期的各个功能。在仿真验证确定串口读写 EEPROM 系统没有问题后，接下来就是进行板级验证了。

30.5串口读写 EEPROM 板级验证

本节介绍在高云开发板上使用串口进行“串口读写 EEPROM”实验的验证。在进行该验证之前请先确保顶层设计中条件编译 DO_SIM 已被注释。

30.5.1引脚分配与电平约束

本次实验板级验证平台是 高云开发板，这里我们再来看看 开发板上 EEPROM 部分的硬件原理图，如图 30-11 所示：

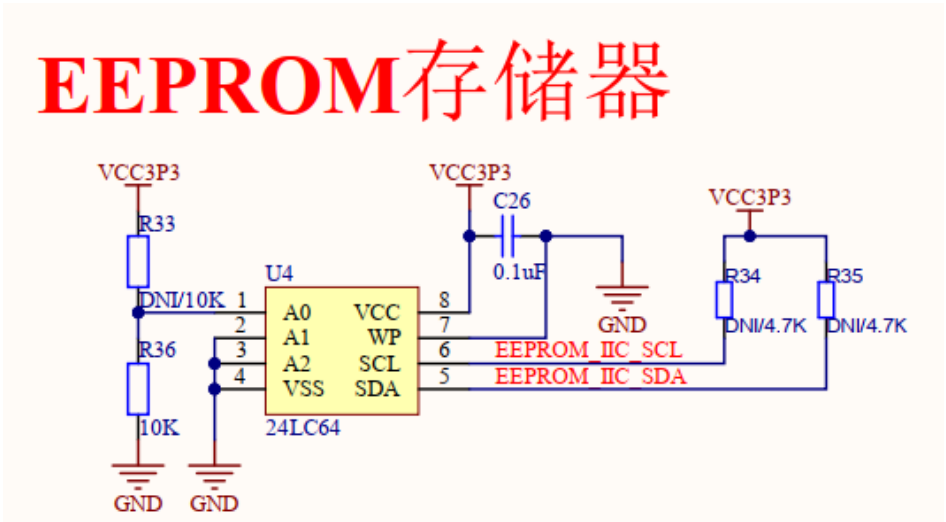
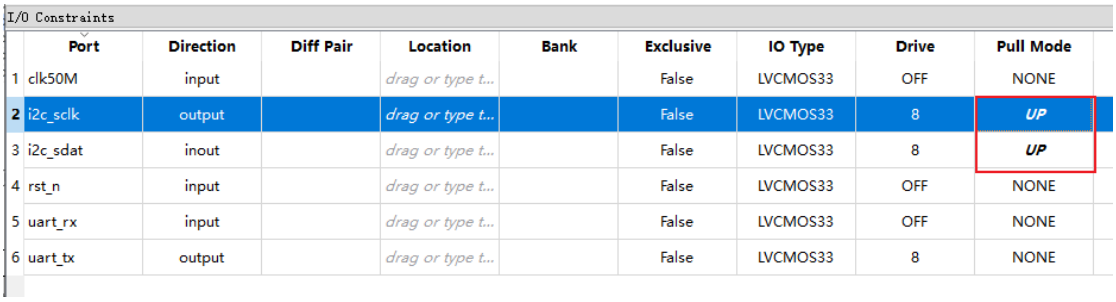


图 30-11 开发板 EEPROM 部分的硬件原理图

细心的读者会发现， I2C 时钟线 SCL 和 I2C 数据线没有进行硬件上拉处理，与前面讲解的需要上拉处理不一样，可能会猜想是硬件设计的问题。这里说明一下，硬件设计是没有问题的，因为对于 FPGA 是可以通过软件对引脚进行上拉处理的，这个也是本实验包含的一个知识点。我们在前面矩阵键盘的实验中，也曾提到过：通过 Gowin 的管脚分配界面就可以将管脚设置为上拉电阻（弱上拉），如下图 30-12 所示。



I/O Constraints

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive	Pull Mode
1	clk50M	input		drag or type t...		False	LVCMOS33	OFF	NONE
2	i2c_sclk	output		drag or type t...		False	LVCMOS33	8	UP
3	i2c_sdat	inout		drag or type t...		False	LVCMOS33	8	UP
4	rst_n	input		drag or type t...		False	LVCMOS33	OFF	NONE
5	uart_rx	input		drag or type t...		False	LVCMOS33	OFF	NONE
6	uart_tx	output		drag or type t...		False	LVCMOS33	8	NONE

图 30-12 设置管脚弱上拉

设置完毕后再为本次设计添加电平约束与管脚分配，本次设计管脚分配表如表 30-1 所示：

表 30-1 引脚分配表

Signial Name	Pin NO.	Signial Name	Pin NO.
clk50M	T9	rst_n	B16
i2c_sclk	R3	uart_rx	U8
i2c_sdat	T3	uart_tx	V8

分配并约束完成后，引脚信息如下图 30-13 所示。

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive	Pull Mode
1	clk50M	input		T9	4	False	LVCMOS33	OFF	NONE
2	i2c_sclk	output		R3	5	False	LVCMOS33	8	UP
3	i2c_sdat	inout		T3	5	False	LVCMOS33	8	UP
4	rst_n	input		B16	1	False	LVCMOS33	OFF	NONE
5	uart_rx	input		U8	5	False	LVCMOS33	OFF	NONE
6	uart_tx	output		V8	5	False	LVCMOS33	8	NONE

图 30-13 管脚分配并约束完成

上述工作完成后，接下来对我们的设计生成比特流，然后便可以开始连接硬件，准备烧录比特流到 高云开发板上进行板级验证了。

30.5.2系统所需硬件

1. 高云开发板 x1
2. 高云下载器 x1
3. 串口线 x1
4. 电源线 x1

30.5.3硬件连接

将下载器、电源线依次连接至开发板上，将串口一端连接开发板，一端连接主机，连接串口之后，可以用 USB 直接给开发板供电也可以用电源供电，本

次实验采用电源供电，如下图 30-14 所示。

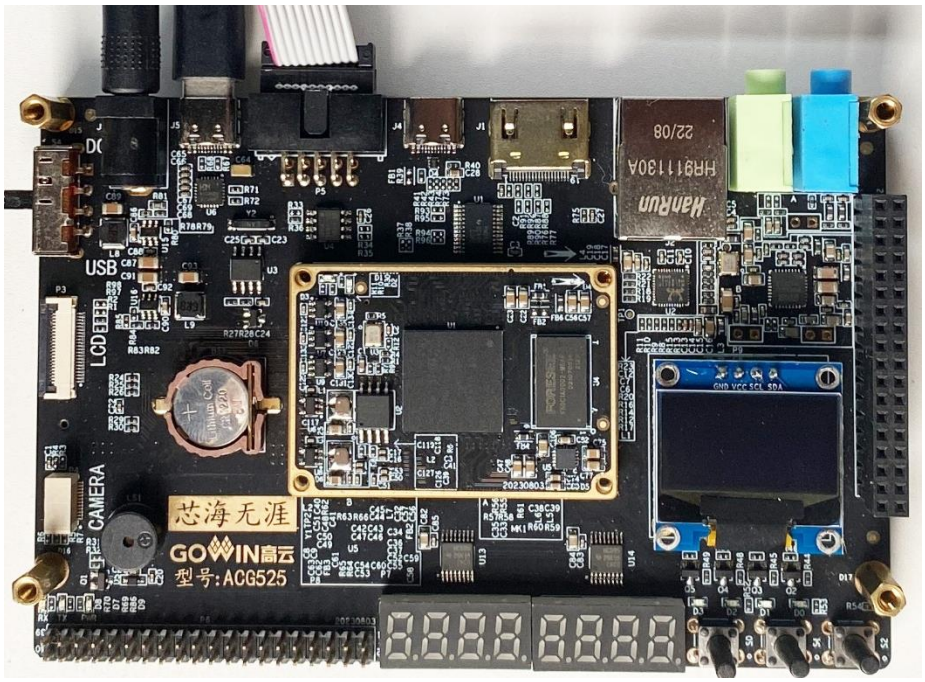


图 30-14 硬件连接图

连接完毕后拨动开关为开发板上电（拨到 DC 侧），接下来将生成好的 bit 烧录到开发板中开始板级验证。

30.5.4板级验证

板级验证需要用到串口软件工具，这里使用的是名叫串口猎人的串口工具（该工具软件在开发板资料的常用软件中有提供，用户也可自行选择其他串口软件）。

首先在设备管理器中找到串口对应的端口号，然后在串口软件中选择对应端口号，并设置波特率为 115200（该值可以在设计的顶层中修改），如图 30-15 所示。

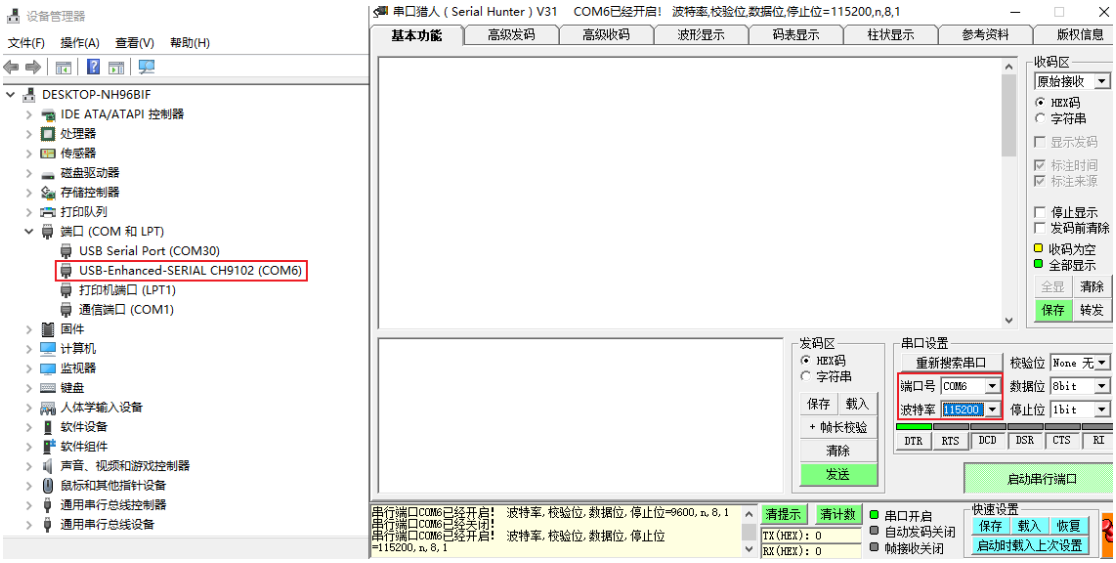


图 30-15 连接串口

连接完成后，我们分别发送串口写指令和串口读指令，对串口写入 16 字节数据并读出，观察读写数据是否一致，对应结果如图 30-16 和图 30-17 所示：

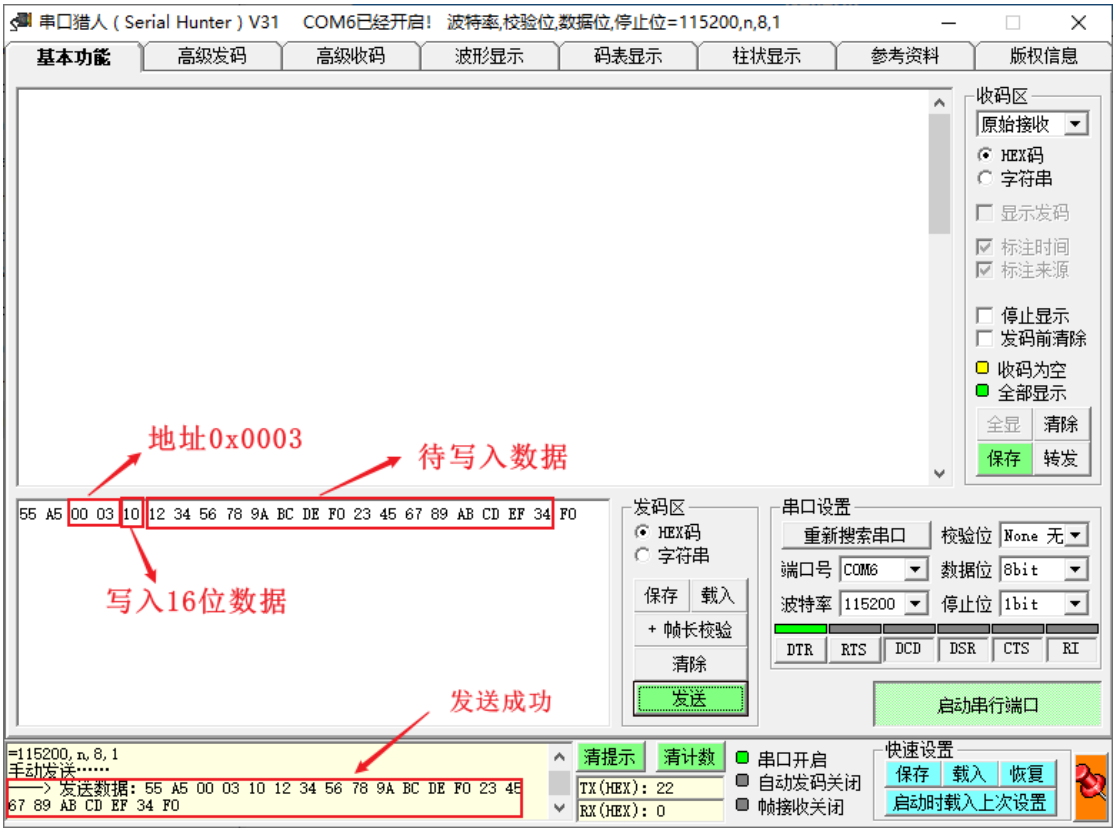


图 30-16 串口写指令

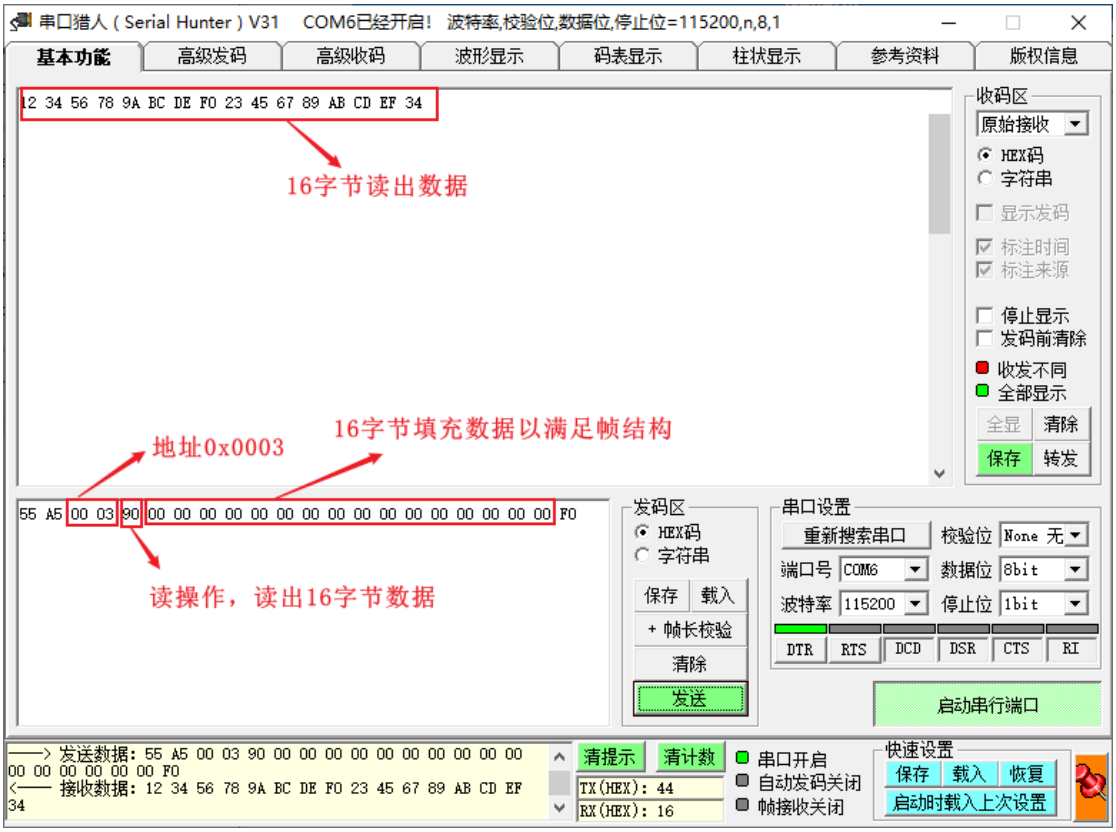


图 30-17 串口读指令

可以看到，读出的数据与写入数据一致，说明设计成功。如果这里串口接收到的数据仅第一个字节正确，用户请确认设计的顶层文件中是否注释掉了条件编译信号 DO_SIM。

30.6总结

本节基于设计好的 I2C 控制器，结合前面设计的串口收发模块设计了一个简易应用，实现了串口对 EEPROM 存储器的读写功能，并进行相应的板级验证。读者可以在此基础上进行一定的优化和拓展，实验开发板上还有音频模块和时钟模块挂载在 I2C 总线上，资源比较丰富，读者可以利用这些模块实现更丰富的应用。

31 基于 I²C 接口的 PCF8563 数字时钟显示

工程源码	|----02_设计实例 |----ch31_uart_rtc8563_tft
相关视频课程	
说明	

章节导读

PCF8563 是 PHILIPS 公司推出的一款工业级内含 I2C 总线接口功能的具有极低功耗的 CMOS 多功能时钟/日历芯片。PCF8563 的多种报警功能、定时器功能、时钟输出功能以及中断输出功能能完成各种复杂的定时服务，甚至可为单片机提供看门狗功能。是一款性价比极高的时钟芯片，它已被广泛用于电表、水表、气表、电话、传真机、便携式仪器以及电池供电的仪器仪表等产品领域。

本章将通过 I2C 总线接口读取 PCF8563 时钟芯片的时间和日期，并通过串口每秒钟实时将时间发送出来，同时也能通过串口来校准 PCF8563 时钟芯片的时间和日期，并且通过数码管显示日期和时间。

31.1功能描述

PCF8563 芯片结构框图如图 31-1 所示。

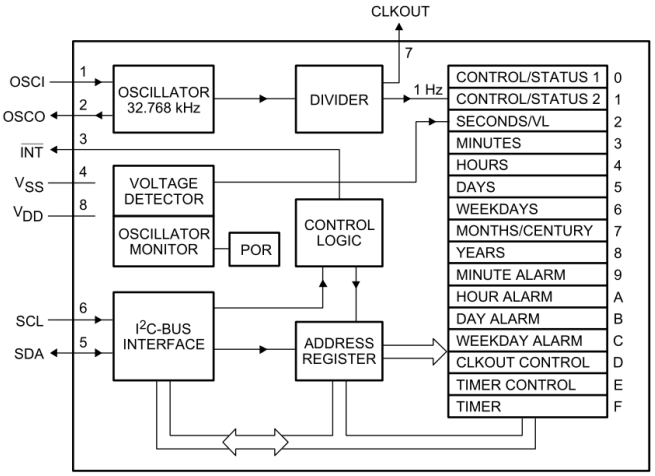


图 31-1 PCF8563 芯片结构框图

PCF8563 包含 16 个 8 位寄存器，一个可自动增量的地址寄存器，一个内置 32.768KHz 的振荡器（带有一个内部集成的电容），一个分频器（用于给实时时钟 RTC 提供源时钟），一个可编程时钟输出，一个定时器，一个报警器，一个

掉电检测器和一个 400KHz I²C 总线接口。

PCF8563 包含的 16 个寄存器被设计成可寻址地址为 8 位的寄存器，但 8 位寻址地址不是所有位都有用。前两个寄存器（地址为 0x00 和 0x01 寄存器）用于控制寄存器和状态寄存器，内存地址 0x02~0x08 的寄存器用于时钟计数器（秒~年计数器），地址 0x09~0x0C 用于报警寄存器（定义报警条件），地址 0x0D 控制 CLKOUT 管脚的输出频率，地址 0x0E 和 0x0F 分别用于定时器控制寄存器和定时器寄存器。秒、分、时、日、月、年、分钟报警、小时报警、日报警寄存器中数据编码格式为 BCD 码，星期和星期报警寄存器不以 BCD 码格式编码。

当一个 RTC 寄存器被读时，所有计数器的内容被锁存，因此，在传送条件下，可以禁止对时钟\日历芯片的错读。

关于具体的寄存器详细功能信息可以查看配套的芯片数据手册。

31.2串行接口

PCF8563 寄存器读写控制接口为串行的 I²C 接口。具体 I²C 接口的读写数据控制器设计在“I²C 接口控制器设计与验证”一节中已经做了讲解。本节就不重复讲解。

31.3器件地址

根据上一节 I²C 总线协议可以知道，主机在使用 I²C 总线对从机进行读写操作时，在 I2C 总线启动后，主机首先需要传送一个字节的控制字，该控制器高 7bit 为从机的器件地址，最低 1bit 为读写控制命令。开发板上 PCF8563 的器件地址为 0xa3（读操作）或 0xa2（写操作）。其器件地址各个位的数据如下所示。

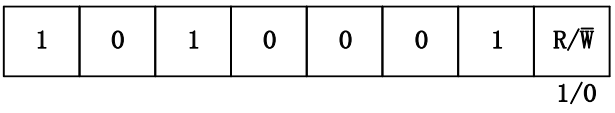


图 31-2 器件地址各个位的数据

31.4总线协议

时钟/日历芯片读/写操作包括 3 种形式。

（1）主机向从机某个寄存器写数据（写模式）。

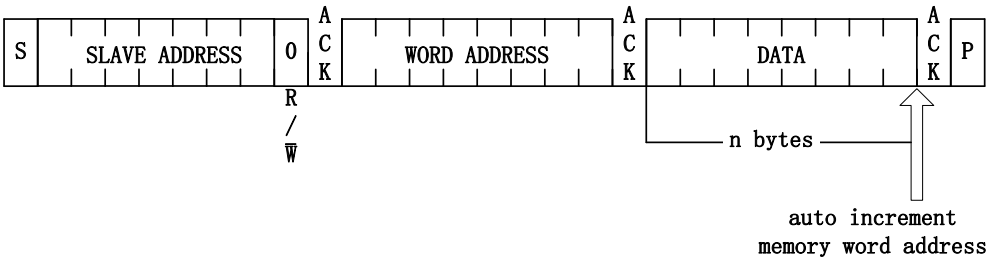


图 31-3 PCF8563 芯片读写操作主机向从机某寄存器写数据

（2）主机从设置的寄存器地址开始往后读从机的寄存器数据（读模式）。

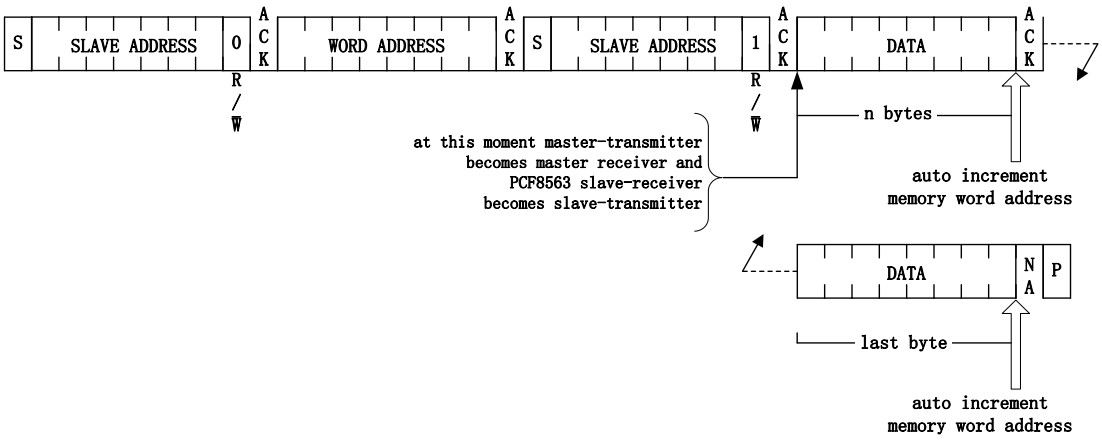


图 31-4 PCF8563 芯片读写操作主机从从机某寄存器读数据

（3）主机从从机的第 1 个地址（0x00）开始往后读从机寄存器数据（读模式）。

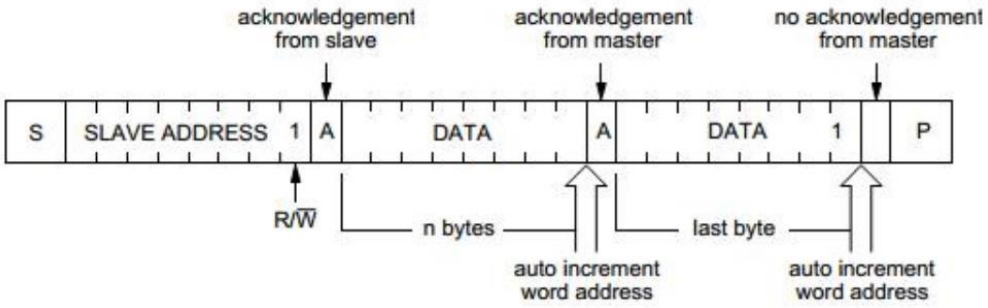


图 31-5 从机从从机第 1 个地址开始往后读从机寄存器数据

三种读/写操作形式中的地址是 8 位的数，用于指定访问寄存器的起始地址，其中这 8bit 地址中的高 4 位无用，低 4 位 0x00~0x0F 对应 PCF8563 的 16 个寄存器地址。

31.5系统整体设计

基于以上原理，本章要实现通过 I²C 总线接口读取 PCF8563 时钟芯片的时间和日期，并通过串口每秒钟，实时将时间发送出来，同时也能通过串口来校准 PCF8563 时钟芯片的时间和日期。搭配数码管，也能显示当前的时间，数码管由 HC595 芯片级联驱动，系统整体设计框图如下图 31-6 所示。

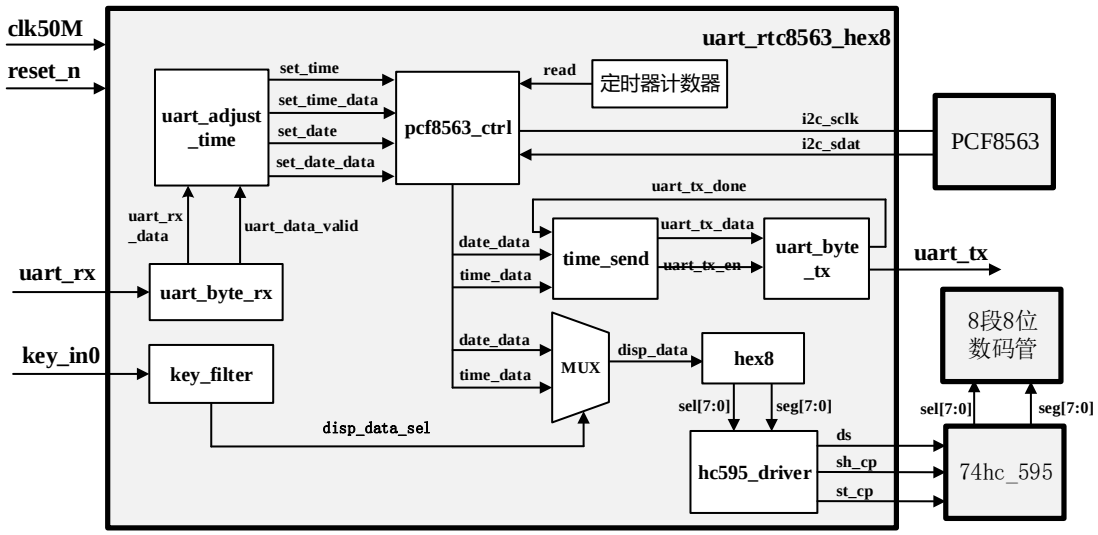


图 31-6 系统框图

顶层 uart_rtc8563_hex8 接口功能描述如下表 31-1 所示。

表 31-1 uart_rtc8563_hex8 接口功能描述表

接口名称	方向	功能描述
clk50M	I	模块工作时钟，50MHz 时钟
reset_n	I	模块复位信号
key_in0	I	数码管显示内容切换按键，按下显示日期，释放显示时间
uart_rx	I	串口输入信号
uart_tx	O	串口输出信号
i2c_sclk	O	i2c 时钟总线
i2c_sdat	I/O	i2c 数据总线
ds	O	串行数据输出
sh_cp	O	移位寄存器的时钟输出
st_cp	O	存储寄存器的时钟输出

子模块功能描述如下表 31-2 所示。

表 31-2 子模块功能描述表

接口名称	功能描述
pcf8563_ctrl	PCF8563 驱动模块
uart_adjust_time	时间校准模块，通过串口接收的时间校准指令去控制 pcf8563_ctrl 模块去调整时间
uart_byte_rx	串口接收模块，用于接收调整时间的指令

time_send_uart	时间发送模块，通过控制 uart_byte_tx 模块，将时间按照规定格式发送出去
uart_byte_tx	串口发送模块
hex8	数码管显示模块，显示当前时间
key_filter	按键消抖模块
定时计数器	通过计数器产生固定时间间隔去读取 PCF8563 时间日期的读信号，逻辑比较简单，不以单独模块形式存在，直接在顶层通过逻辑产生
MUX	2 选 1 模块，通过按键输入信号选择不同的数据显示在数码管上
hc595_driver	hc595 的驱动模块，可以将数码管的段选和位选信号转换成 SPI 接口信号以节省向外传输的位宽。

31.6PCF8563 驱动模块设计

根据功能需求，我们把整个模块设计成了 4 个分支，也就是 9 个状态，从下面的状态转移图可以很直观的看出来，如下图 31-7 所示。

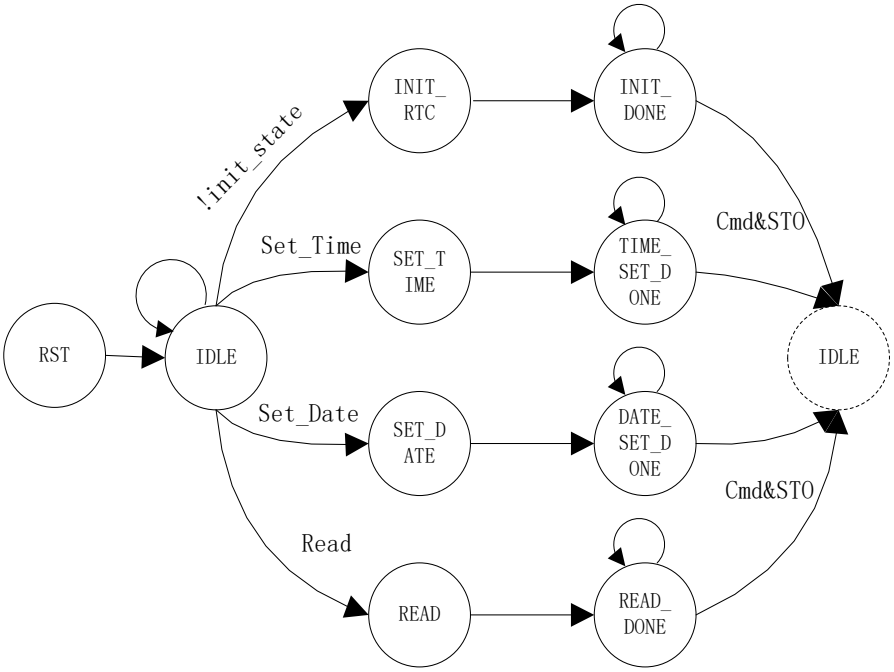


图 31-7 PCF8563 驱动模块设计图

第一条分支：在空闲状态如果 init_state 为 0，说明时钟芯片还没有初始化，就进入初始化时钟芯片状态（INIT_RTC），在这个状态里面将相关的命令写入到时钟芯片后，就跳转到等待时钟芯片初始化完成状态（INIT_DONE），初始化完成了就会回到空闲状态（IDLE）。代码如下：

```
IDLE:
    begin
    Set_Done <= 1'b0;
    Read_Done <= 1'b0;
    wrreg_req <= 1'b0;
```

```
    rdreg_req <= 1'b0;
    cnt <= 0;
    if(!init_state)
        state <= INIT_RTC;
    else if(Set_Date)
        state <= SET_DATE;
    else if(Set_Time)
        state <= SET_TIME;
    else if(Read)
        state <= READ;
    end

INIT_RTC:
    begin
    write_reg(PCF8563_Address_Control_Status_1, 0);
    state <= INIT_DONE;

end

INIT_DONE:
    begin
    wrreg_req <= 1'b0;
    if(RW_Done)begin
        state <= IDLE;
        init_state <= 1;
    end
    else
        state <= INIT_DONE;
end
```

第二条分支：在空闲状态如果时钟芯片已经初始化，如果有设置时间请求（Set_Time），就进入设置时间状态（SET_TIME），在这个状态里面将设置时间写入到时钟芯片后，就跳转到等待设置时间完成状态（TIME_SET_DONE），设置时间完成了就会回到空闲状态（IDLE）。代码如下：

```
SET_TIME:
begin
    state <= TIME_SET_DONE;
    case(cnt)
    0:write_reg(PCF8563_Address_Seconds, Time_to_Set[7:0]);
    1:write_reg(PCF8563_Address_Minutes, Time_to_Set[15:8]);
    2:write_reg(PCF8563_Address_Hours, Time_to_Set[23:16]);
    default:;
    endcase
end

TIME_SET_DONE:
```

```
begin
    wrreg_req <= 1'b0;
    if(RW_Done)begin
    if(cnt < 2)begin
        cnt <= cnt + 1'b1;
        state <= SET_TIME;
    end
    else begin
        cnt <= 0;
        Set_Done <= 1'b1;
        state <= IDLE;
    end
    end
    else
    state <= TIME_SET_DONE;
end
```

第三条分支：在空闲状态如果时钟芯片已经初始化，并且没有设置时间请求（Set_Time），此时有设置日期请求（Set_Date），就进入设置日期状态（SET_DATE），在这个状态里面将设置日期写入到时钟芯片后，就跳转到等待设置日期完成状态（DATE_SET_DONE），设置日期完成了就会回到空闲状态（IDLE）。代码如下：

```
SET_DATE:
    begin
        state <= DATE_SET_DONE;
        case(cnt)
        0:write_reg(PCF8563_Address_Days, Date_to_Set[7:0]);
        1:write_reg(PCF8563_Address_WeekDays, Date_to_Set[15:8]);
        2:write_reg(PCF8563_Address_Months, Date_to_Set[23:16]);
        3:write_reg(PCF8563_Address_Years, Date_to_Set[31:24]);
        default:;
        endcase
    end

    DATE_SET_DONE:
    begin
        wrreg_req <= 1'b0;
        if(RW_Done)begin
        if(cnt < 3)begin
            cnt <= cnt + 1'b1;
            state <= SET_DATE;
        end
        else begin
            cnt <= 0;
            Set_Done <= 1'b1;
            state <= IDLE;
```

```
        end
        end
        else
        state <= DATE_SET_DONE;
    end
```

第四条分支：在空闲状态如果时钟芯片已经初始化，并且没有设置时间请求（Set_Time）、没有设置日期请求（Set_Date），此时有读日期时间请求（Read），就进入读取日期时间状态（READ），在这个状态里面将对应的寄存器地址写入到时钟芯片后，就跳转到等待日期时间读取完成状态（READ_DONE）里面读取日期时间，日期时间读取完成了就会回到空闲状态（IDLE）。这也是芯片在时间日期读写稳定后，日常运行的状态。其代码如下：

```
SET_DATE:
begin
    state <= DATE_SET_DONE;
    case(cnt)
    0:write_reg(PCF8563_Address_Days, Date_to_Set[7:0]);
    1:write_reg(PCF8563_Address_WeekDays, Date_to_Set[15:8]);
    2:write_reg(PCF8563_Address_Months, Date_to_Set[23:16]);
    3:write_reg(PCF8563_Address_Years, Date_to_Set[31:24]);
    default:;
    endcase
end

DATE_SET_DONE:
begin
    wrreg_req <= 1'b0;
    if(RW_Done)begin
    if(cnt < 3)begin
        cnt <= cnt + 1'b1;
        state <= SET_DATE;
    end
    else begin
        cnt <= 0;
        Set_Done <= 1'b1;
        state <= IDLE;
    end
    end
    else
    state <= DATE_SET_DONE;
end
```

注意，读出的 RTC 芯片时钟数据已经按照器件的使用说明，严格的作了按位与操作掩码处理，确保器件手册提到的有关数据位生效，无关数据位强制置 0。通过该操作，这样可以避免对器件无关位的理解不同导致读出的数据不同，从

而产生错误的时间日期显示结果。

当然上面为了让代码更简洁，这里就用到了两个 task，一个是用来写字节（write_reg），另一个是读字节（read_reg）。写字节的 task 里面将要写的寄存器地址赋值给 addr，要写的数据赋值给 wrdata，同时将写请求（wrreg_req）拉高，这样就可以通过控制 i2c_control 模块将数据写入到时钟芯片里面了（FPGA 指挥外部时钟芯片）。同理，读字节的 task 里面将你要从那个寄存器地址读数据，就把地址赋值给 addr，同时将读请求（rdreg_req）拉高，这样就可以通过控制 i2c_control 模块将数据从时钟芯片里面读取出来了。

```
task write_reg;
    input [7:0]reg_addr;
    input [7:0]reg_data;
    begin
        wrreg_req <= 1'b1;
        addr = reg_addr;
        wrdata = reg_data;
    end
endtask

task read_reg;
    input [7:0]reg_addr;
    begin
        rdreg_req <= 1'b1;
        addr = reg_addr;
    end
endtask
```

31.7时间校准模块设计

这个模块就是通过把串口接收的数据，先存入到一个移位寄存 rx_cmd_data 里面，然后实时判断移位寄里面的数据帧头部分是否跟自己定义匹配，如果匹配就将数据部分（日期、时间）写入到时钟芯片，这样就实现了对时钟芯片的时间校准，这里自己定义了一个协议帧头（自己也可以定义），模块接口示意图如下：

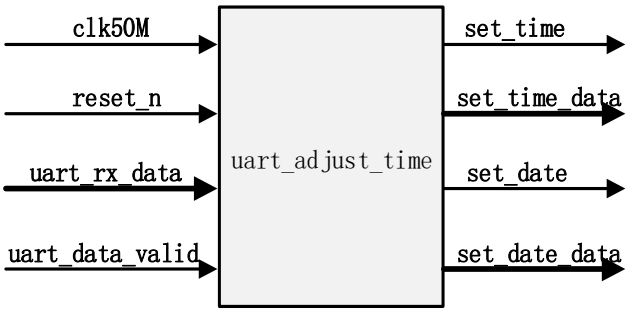


图 31-8 时间校准模块设计

表 31-3 时间校准模块接口功能描述

接口名称	方向	功能描述
clk50M	I	模块工作时钟，50M 时钟
reset_n	I	模块复位信号，低电平有效
uart_data_valid	I	数据有效标志，为 1 输入的 8 位数据 uart_data 有效
uart_rx_data	I	串口输入的 8 位数据
set_time	O	设置时间触发信号，为 1，set_time_data 的值将被写入到时钟芯片
set_time_data	O	24 位的时间数据
set_date	O	设置日期触发信号，为 1，set_date_data 的值将被写入到时钟芯片
set_date_data	O	32 位的日期数据

时间校准数据帧，格式如下表 31-4。

表 31-4 时间校准格式表

byte10	byte9	byte8	byte7	byte6	byte5	byte4	byte3	byte2	byte1	byte0
帧头 0	帧头 1	帧头 2	帧头 3	年	月	日	周	时	分	秒
0x01	0x03	0x02	0x04	0x19	0x11	0x11	0x01	0x23	0x59	0x30

对协议帧的解释：

前面的帧头 0~3 是属于自己定义的部分（这里定义的帧头是 0x01030204），可长可短（可自行定义），后面年里面的 0x19 表示 2019 年，以此类推，月里面 0x11 表示 11 月，日里面的 0x11 表示 11 日，周里面的 0x01 表示周一（0x07 表示周日），后面的时里面的 0x23 表示 23 时，分里面的 0x59 表示 59 分，秒里面的 0x30 表示 30 秒。帧头检测移位寄存器部分代码如下：

```
always @ (posedge clk or negedge reset_n)
begin
if (!reset_n)
    rx_cmd_data <= 'd0;
else if (uart_data_valid)
    rx_cmd_data <= {rx_cmd_data[79:0], uart_rx_data};
else
    rx_cmd_data <= rx_cmd_data;
end

assign cmd_checked = (rx_cmd_data[87:56] == 32'h01030204);
```

当然这里的 checked 信号是一个脉冲信号，所以我们要把这个信号进行边沿检测，检测到了后就通过 set_time 这个信号来控制上面的时钟芯片驱动模块，将时间进行校准，同时通过 set_date 这个信号来控制上面的时钟芯片驱动模块，将日期进行校准，代码如下：

```
//接收到正确的时间调整命令后，先对时间进行调整，时间设置完成后再对日期进行调整更新
always @ (posedge clk or negedge reset_n)
begin
if (!reset_n)
    cmd_checked_dly1 <= 1'b0;
else
    cmd_checked_dly1 <= cmd_checked;
end

//第 1 步设置时间，确保设置成功
always @ (posedge clk or negedge reset_n)
begin
if (!reset_n)
    set_time <= 1'b0;
else if(set_done)
    set_time <= 1'b0;
else if(cmd_checked && (!cmd_checked_dly1))
    set_time <= 1'b1;
end

//时间调整数据，00:23:20  //0.1s 时间计数器
assign set_time_data = rx_cmd_data[23:0];
always @ (posedge clk or negedge reset_n)
begin
if (!reset_n)
    set_state <= 1'b0;
else if(set_time)
    set_state <= 1'b1;
else if(set_date)
    set_state <= 1'b0;
end

//第 2 步，时间调整完成后使能设置日期
assign set_date = (set_done & (set_state == 1)) ? 1'b1 : 1'b0;
assign set_date_data = {rx_cmd_data[55:40], rx_cmd_data[31:24],
rx_cmd_data[39:32]};//日期调整数据，19-05-01
```

时间校准模块设计完成后进行模块的功能仿真。仿真 testbench 设计中将串口接收和发送模块例化到其中，串口接收模块用来实现接收串口传来的数据，

串口发送模块用来模拟实际上板测试过程中 PC 发送的时间调整命令数据。具体代码如下，仿真过程中模拟发送了 2 次时间调整数据命令。分别为 88'h01030204_191111_01_235930（调整时间到 19 年 11 月 11 日 23 点 59 分 30 秒）和 88'h01030204_191113_03_124956（调整时间到 19 年 11 月 11 日 23 点 59 分 30 秒）。指令的发送使用任务的方式进行封装，这样可以方便多次进行时间调整的仿真。为了减少仿真中调整时间和调整日期之间的由于 0.1S 导致的仿真时间过长，在 testbench 中使用参数重定义 defparam uart_adjust_time.DLY_CNT_MAX = 500;将延时计数器的最大值改小。仿真设计的具体代码如下：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module uart_adjust_time_tb();

reg         clk50M         ;
reg         reset_n        ;
reg  [7:0]  uart_tx_data   ;
reg         uart_tx_en     ;
reg         set_done       ;
wire        uart_tx_done   ;
wire        uart_rx        ;
wire [7:0]  uart_rx_data   ; //串口接收数据
wire        uart_data_valid; //串口接收数据有效标识
wire        set_time       ; //设置时间使能
wire [23:0] set_time_data  ; //设置时间数据
wire        set_date       ; //设置日期使能
wire [31:0] set_date_data  ; //设置日期数据
reg  [87:0] adjust_cmd_data_reg;

integer     i;

//defparam uart_adjust_time.DLY_CNT_MAX = 500;

  initial clk50M = 1;
  always#(`CLK_PERIOD/2)clk50M = ~clk50M;

  initial begin
    reset_n = 1'b0;
    uart_tx_data = 8'd0;
    uart_tx_en = 1'b0;
    adjust_cmd_data_reg = 88'd0;
    set_done = 1'b0;
    #(`CLK_PERIOD*20 + 1 );
    reset_n = 1'b1;
```

```
    #(`CLK_PERIOD*50);

    send_adjust_cmd(88'h01030204_191111_01_235930);
    #1000000;
    set_done = 1;
    #(`CLK_PERIOD  );
    set_done = 0;
    #1000000;
    set_done = 1;
    #(`CLK_PERIOD  );
    set_done = 0;

    send_adjust_cmd(88'h01030204_191113_03_124956);
    #1000000;
    set_done = 1;
    #(`CLK_PERIOD  );
    set_done = 0;
    #1000000;
    set_done = 1;
    #(`CLK_PERIOD  );
    set_done = 0;
    #4000000;
    $stop;
  end

task send_adjust_cmd;
  input [87:0] adjust_cmd_data;
  begin
    adjust_cmd_data_reg = adjust_cmd_data;
    for(i=11;i>0;i=i-1) begin
    uart_tx_data = adjust_cmd_data_reg[87:80];
    uart_tx_en = 1'b1;
    #(`CLK_PERIOD);
    uart_tx_en = 1'b0;
    @(posedge uart_tx_done);
    adjust_cmd_data_reg = adjust_cmd_data_reg << 8;
    #40;
    end
  end
endtask

  uart_byte_tx uart_byte_tx
  (
    .clk    (clk50M       ),
    .reset_n   (reset_n      ),
    .data_byte (uart_tx_data ),
    .send_en   (uart_tx_en   ),
```

```
    .baud_set  (3'd0         ),
    .uart_tx   (uart_rx      ),
    .tx_done   (uart_tx_done ),
    .uart_state(             )
  );

  uart_byte_rx uart_byte_rx
  (
    .clk   (clk50M          ),
    .reset_n  (reset_n         ),
    .baud_set (3'd0            ),
    .uart_rx  (uart_rx         ),
    .data_byte(uart_rx_data    ),
    .rx_done  (uart_data_valid )
  );

    uart_adjust_time uart_adjust_time
  (
    //clock reset
    .clk            (clk50M          ), //系统时钟输入__50M
    .reset_n        (reset_n         ), //复位信号输入，低有效
    .uart_rx_data   (uart_rx_data    ), //串口接收数据
    .uart_data_valid(uart_data_valid ), //串口接收数据有效标识
    .set_done       (set_done        ),
    .set_time       (set_time        ), //设置时间使能
    .set_time_data  (set_time_data   ), //设置时间数据
    .set_date       (set_date        ), //设置日期使能
    .set_date_data  (set_date_data   )  //设置日期数据
  );
endmodule
```

仿真波形如下图 31-9 所示。

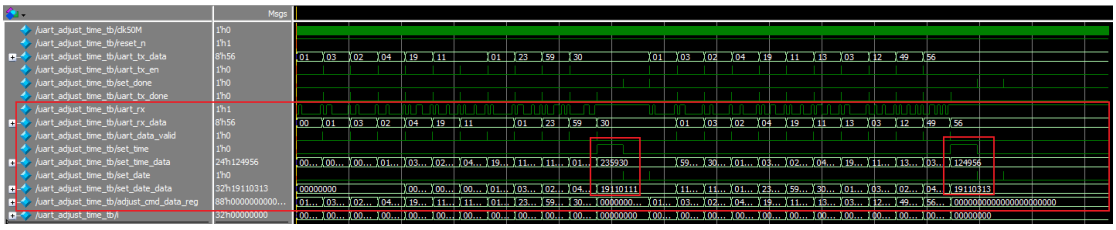


图 31-9 时间校准模块仿真波形图

从波形可以看出，仿真进行两次时间的调整，一次是将时间调整为 23 点 59 分 30 秒，日期调整为 19 年 11 月 11 日周一；二次是将时间调整为 12 点 49 分 56 秒，日期调整为 19 年 11 月 13 日周三。与预期相符，说明时间调整模块设计是符合预期的。

31.8时间发送控制模块设计

发送控制模块的思路其实很简单，总体思路其实就是先利用系统时钟，通过一个计数器，计时到 1 秒钟，然后产生一个脉冲信号，通过这个脉冲信号来更新当前要发送的时间，模块接口示意图如下图 31-10 所示：

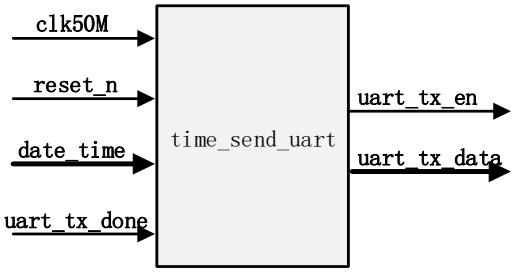


图 31-10 时间发送控制模块设计

表 31-5 时间发送模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块工作时钟，50M 时钟
reset_n	I	模块复位信号，低电平有效
uart_tx_done	I	触发脉冲信号，为高电平，将 1 字节的数据 date_time 送到 uart_tx_data 端口，使用串口发送 1 字节完成标识信号作为该信号
date_time	I	待发送的 48 位数据（含日期、时间）
uart_tx_en	O	串口发送的 1 字节数据
uart_tx_data	O	串口发送使能信号，触发脉冲信号

当待发送的 48 位日期时间发生变化时产生一个标识信号，该信号用来作为向串口发送一次日期时间数据的使能信号。具体代码如下：

```
//数据发生变化就通过串口发送一次数据
always@(posedge clk or posedge reset)
begin
    if(reset)
    date_time_latch <= 'd0;
    else if(date_time_en)
    date_time_latch <= date_time;
end

always@(posedge clk or posedge reset)
begin
    if(reset)
    time_data_change <= 1'b0;
    else if(date_time_en & (date_time_latch != date_time))
    time_data_change <= 1'b1;
    else
    time_data_change <= 1'b0;
```

```
end
```

对于通过串口发送时间模块的设计，首先自定义通过串口发送的日期时间格式如下。其中以日期时间 2019 年 5 月 12 日 10 点 5 分 57 秒举例说明格式组成。

表 31-6 时间发送模块数据格式

格式	'2'	'0'	年		'-'	月		'-'	日		空格	时		':'	分		':'	秒		回车
举例	2	0	1	9	-	0	5	-	1	2	空格	1	0	:	0	5	:	5	7	回车

定义的格式中数据是以字符（ASCII）显示（为了让用户看起来更直观些），所以在通过串口发送模块发送之前需要将 16 进制数据转换成字符处理，这里也用了个小技巧，因为日期和时间都是数字，而 0 的 ASCII 是 8'h30，所以我们将对应的数据加上这个 8'h30 就能将这个数字直接换成字符了，代码如下：

```
//bcd translate ascii
always @ (posedge clk)
if (time_data_change) begin
    data_time_ascii[111:104] <= 4'd2 + 8'h30;//"2"
    data_time_ascii[103:96 ] <= 4'd0 + 8'h30;//"0"
    data_time_ascii[95 :88 ] <= date_time_latch[47:44] + 8'h30;//YY
    data_time_ascii[87 :80 ] <= date_time_latch[43:40] + 8'h30;
    data_time_ascii[79 :72 ] <= date_time_latch[39:36] + 8'h30;//MM
    data_time_ascii[71 :64 ] <= date_time_latch[35:32] + 8'h30;
    data_time_ascii[63 :56 ] <= date_time_latch[31:28] + 8'h30;//DD
    data_time_ascii[55 :48 ] <= date_time_latch[27:24] + 8'h30;
    data_time_ascii[47 :40 ] <= date_time_latch[23:20] + 8'h30;//hh
    data_time_ascii[39 :32 ] <= date_time_latch[19:16] + 8'h30;
    data_time_ascii[31 :24 ] <= date_time_latch[15:12] + 8'h30;//mm
    data_time_ascii[23 :16 ] <= date_time_latch[11:8]  + 8'h30;
    data_time_ascii[15 :8  ] <= date_time_latch[7:4]   + 8'h30;//ss
    data_time_ascii[7  :0  ] <= date_time_latch[3:0]   + 8'h30;
end
```

串口一次发送 1 字节的数据，根据上面自定义的格式，日期时间数据总共是 20 个字节，这样串口发送需要产生一个串口发送字节个数计数器，当计数器计数达到 20 就清零，然后每当使能脉冲信号到来时或者每发送完一个字节计数器加 1。有关计数器的时序图如下图 31-11 所示。

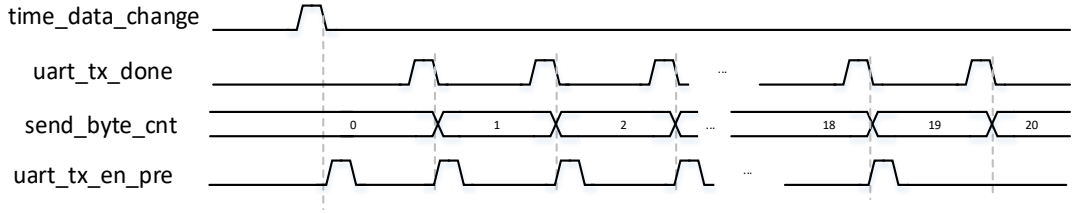


图 31-11 时间发送模块计数器时序图

使用线性序列机方式根据计数器数值发送对应的字符数据和使能串口发送使能信号。具体实现代码如下：

```
always@(posedge clk or posedge reset)
      begin
        if(reset)
          send_byte_cnt <= 'd0;
        else if(time_data_change)
          send_byte_cnt <= 'd0;
        else if(uart_tx_done)
          send_byte_cnt <= send_byte_cnt + 1'b1;
        else
          send_byte_cnt <= send_byte_cnt;
      end

      always@(posedge clk or posedge reset)
      begin
        if(reset)
          uart_tx_en_pre <= 1'b0;
         else if(time_data_change || (uart_tx_done && send_byte_cnt <=
'd18))
          uart_tx_en_pre <= 1'b1;
        else
          uart_tx_en_pre <= 1'b0;
      end

      always@(posedge clk or posedge reset)
      begin
        if(reset)
        begin
          uart_tx_data <= 'd0;
          uart_tx_en <= 1'b0;
        end
        else if(uart_tx_en_pre)
        begin
          case(send_byte_cnt)
            5'd0 : begin uart_tx_data <= data_time_ascii[111:104];
                         uart_tx_en <= 1'b1; end  //2 YYYY
            5'd1 : begin uart_tx_data <= data_time_ascii[103:96 ];
                         uart_tx_en <= 1'b1; end  //0
            5'd2 : begin uart_tx_data <= data_time_ascii[95 :88 ];
                         uart_tx_en <= 1'b1; end  //1
            5'd3 : begin uart_tx_data <= data_time_ascii[87 :80 ];
                         uart_tx_en <= 1'b1; end  //9
            5'd4 : begin uart_tx_data <= 8'h2D;
```

```
                        uart_tx_en <= 1'b1; end  //-
            5'd5 : begin uart_tx_data <= data_time_ascii[79 :72 ];
                        uart_tx_en <= 1'b1; end  //0 MM
            5'd6 : begin uart_tx_data <= data_time_ascii[71 :64 ];
                        uart_tx_en <= 1'b1; end  //1
            5'd7 : begin uart_tx_data <= 8'h2D;
                        uart_tx_en <= 1'b1; end  //-
            5'd8 : begin uart_tx_data <= data_time_ascii[63 :56 ];
                        uart_tx_en <= 1'b1; end  //0 DD
            5'd9 : begin uart_tx_data <= data_time_ascii[55 :48 ];
                        uart_tx_en <= 1'b1; end  //1
            5'd10: begin uart_tx_data <= 8'h20;
                        uart_tx_en <= 1'b1; end  //space
            5'd11: begin uart_tx_data <= data_time_ascii[47 :40 ];
                        uart_tx_en <= 1'b1; end  //1 hh
            5'd12: begin uart_tx_data <= data_time_ascii[39 :32 ];
                        uart_tx_en <= 1'b1; end  //2
            5'd13: begin uart_tx_data <= 8'h3A;
                        uart_tx_en <= 1'b1; end  //:
            5'd14: begin uart_tx_data <= data_time_ascii[31 :24 ];
                        uart_tx_en <= 1'b1; end  //0 mm
            5'd15: begin uart_tx_data <= data_time_ascii[23 :16 ];
                        uart_tx_en <= 1'b1; end  //0
            5'd16: begin uart_tx_data <= 8'h3A;
                        uart_tx_en <= 1'b1; end  //:
            5'd17: begin uart_tx_data <= data_time_ascii[15 :8  ];
                        uart_tx_en <= 1'b1; end  //0 ss
            5'd18: begin uart_tx_data <= data_time_ascii[7  :0  ];
                        uart_tx_en <= 1'b1; end  //0
            5'd19: begin uart_tx_data <= 8'h0A;
                        uart_tx_en <= 1'b1; end  // LF(换行)
              default:begin uart_tx_data <= 'd0;
                            uart_tx_en <= 1'b0; end
          endcase
        end
        else
        begin
          uart_tx_data <= uart_tx_data;
          uart_tx_en <= 1'b0;
        end
    end

    endmodule
```

模块功能仿真，在仿真文件设计中添加串口发送模块 uart_byte_tx，这样就可以达到与实际整体功能一致的效果。出此之外还需提供时钟复位激励以及模拟需要发送的时间数据即可。具体仿真设计代码如下。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module time_send_uart_tb();

  reg         clk50M       ; //系统时钟输入，50M
  reg         reset_n      ; //复位信号输入，低有效
  reg  [47:0] date_time    ;
  reg         read_done    ;
  wire        uart_tx_done ;
  wire [7:0]  uart_tx_data ;
  wire        uart_tx_en   ;
  wire        uart_tx      ;

    initial clk50M = 1;
    always#(`CLK_PERIOD/2)clk50M = ~clk50M;

  initial begin
    reset_n = 1'b0;
    date_time = 48'd0;
    read_done = 1'b0;
    #(`CLK_PERIOD*20 + 1 );
    reset_n = 1'b1;
    #(`CLK_PERIOD*50);
    date_time = 48'h19_05_01_10_23_56;
    #30000000;
    read_done = 1;
    #(`CLK_PERIOD  );
    read_done = 0;
    #1000000;
    read_done = 1;
    #(`CLK_PERIOD  );
    read_done = 0;

    date_time = 48'h19_06_10_12_45_12;
    #30000000;
    read_done = 1;
    #(`CLK_PERIOD  );
    read_done = 0;
    #1000000;
    read_done = 1;
    #(`CLK_PERIOD  );
    read_done = 0;

    $stop;
  end
```

```
  time_send_uart time_send_uart(
    .clk(clk50M),
    .reset(~reset_n),
    .date_time_en(read_done),
    .date_time(date_time),
    .uart_tx_done(uart_tx_done),
    .uart_tx_data(uart_tx_data),
    .uart_tx_en(uart_tx_en)
  );

  uart_byte_tx uart_byte_tx
  (
    .clk       (clk50M       ),
    .reset_n   (reset_n      ),
    .data_byte (uart_tx_data ),
    .send_en   (uart_tx_en   ),
    .baud_set  (3'd0         ),
    .uart_tx   (uart_tx      ),
    .tx_done   (uart_tx_done ),
    .uart_state(             )
  );

endmodule
```

31.9板级调试与验证

本实验的板级验证环节，主要验证以下几个目标：

1. 能否正确将生成的 bit 文件下载到开发板。
2. 下载完成后数码管能否实现时间的显示格式。
3. 按下 S0 按键后数码管能否切换显示格式为日期。
4. 能否使用串口调试工具进行时间校准，将时间设定为希望显示的时间。
5. 能否使用提供的时间设置软件小工具，将时间设置为实时时间，并可以通过按下按键 S0 在当前的时间和当天日期之间切换。
6. 能否使用 TFT 屏替换数码管，复现上述步骤。

31.9.1系统所需硬件

1. 开发板
2. 电源电缆一根

3. USB-typeC 口电缆一根

4. 硬件条件符合实验要求，具有完全开发功能的 PC 机一台

完成各子模块的设计和仿真后，对顶层进行设计，顶层的设计可依照系统整体设计框图对各模块进行例化。由于数码管显示的设计上，开发板上用的是板载数码管加 HC595 串转并模式，代码如下所示：

```
module uart_rtc8563_hex8(
  //external clock, reset
  input clk50M,
  input reset_n,
  //key
  input key_in0,
  //LED
  output [1:0]led,
  //uart interface
  input uart_rx,
  output uart_tx,
  //hex8 interface
  output sh_cp,
  output st_cp,
  output ds,
  //pcf8563 interface
  output i2c_sclk,
  inout i2c_sdat
);

  wire disp_data_sel;
  wire [7:0]uart_rx_data; //串口接收数据
  wire uart_data_valid; //串口接收数据有效标识
  wire uart_tx_done;
  wire [7:0]uart_tx_data;
  wire uart_tx_en;
  wire set_done;
  wire set_time; //设置时间使能
  wire [23:0]set_time_data; //设置时间数据
  wire set_date; //设置日期使能
  wire [31:0]set_date_data; //设置日期数据
  reg  [25:0]read_tcnt;
  wire read;
  wire [23:0]time_data;
  wire [31:0]date_data;
  wire read_done;
  wire [31:0]disp_data;
  wire [47:0]date_time;
```

```
  assign led[0] = ~disp_data_sel;
  assign led[1] = read_done;

  key_filter key_filter
  (
    .clk(clk50M),
    .reset_n(reset_n),
    .key_in(key_in0),
    .key_flag(),
    .key_state(disp_data_sel)
  );

  uart_byte_rx uart_byte_rx
  (
    .clk(clk50M),
    .reset_n(reset_n),
    .baud_set (3'd0),
    .uart_rx(uart_rx),
    .data_byte(uart_rx_data),
    .rx_done(uart_data_valid )
  );

  uart_byte_tx uart_byte_tx
  (
    .clk(clk50M),
    .reset_n(reset_n),
    .data_byte(uart_tx_data),
    .send_en(uart_tx_en),
    .baud_set(3'd0),
    .uart_tx(uart_tx),
    .tx_done(uart_tx_done),
    .uart_state()
  );

  uart_adjust_time uart_adjust_time
  (
    //clock reset
    .clk            (clk50M          ), //系统时钟输入__50M
    .reset_n        (reset_n         ), //复位信号输入，低有效
    .uart_rx_data   (uart_rx_data    ), //串口接收数据
    .uart_data_valid(uart_data_valid ), //串口接收数据有效标识
    .set_done       (set_done        ),
    .set_time       (set_time        ), //设置时间使能
    .set_time_data  (set_time_data   ), //设置时间数据
    .set_date       (set_date        ), //设置日期使能
    .set_date_data  (set_date_data   )  //设置日期数据
  );
```

```
  assign date_time = {date_data[31:16],date_data[7:0],time_data};

  time_send_uart time_send_uart(
    .clk(clk50M),
    .reset(~reset_n),
    .date_time_en(read_done),
    .date_time(date_time),
    .uart_tx_done(uart_tx_done),
    .uart_tx_data(uart_tx_data),
    .uart_tx_en(uart_tx_en)
  );

  always@(posedge clk50M or negedge reset_n)
  begin
    if(!reset_n)
    read_tcnt <= 26'd0;
    else if(read_tcnt == 26'd14_999_999)
    read_tcnt <= 26'd0;
    else
    read_tcnt <= read_tcnt + 1'd1;
  end

  assign read = (read_tcnt == 26'd14_999_999);

  pcf8563_ctrl pcf8563_ctrl(
    .Clk(clk50M),
    .Rst_n(reset_n),
    .Set_Time(set_time),
    .Time_to_Set(set_time_data),
    .Set_Date(set_date),
    .Date_to_Set(set_date_data),
    .Read(read),
    .Time_Read(time_data),
    .Date_Read(date_data),
    .Set_Done(set_done),
    .Read_Done(read_done),
    .i2c_sclk(i2c_sclk),
    .i2c_sdat(i2c_sdat)
  );

  assign disp_data = disp_data_sel ?
{time_data[23:16],4'hf,time_data[15:8] ,4'hf,time_data[7:0]} :
                                      {8'h20,date_data[31:24],date_data
[23:16],date_data[7:0]};

  hex_top_special_for_rtc hex_top_special_for_rtc(
```

```
    .clk (clk50M),
    .reset_n (reset_n),
    .disp_data (disp_data),
    .sh_cp (sh_cp),
    .st_cp (st_cp),
    .ds (ds)
  );

endmodule
```

这里使用 set_done 信号，作为时间日期信号设定完成标志，避免在时间正常读取的同时同步进行设定时间的操作导致时间显示紊乱。

顶层设计中除了例化各个子模块外，通过计数器产生了读取日期时间的读使能信号 read。完成顶层设计后，进行管脚约束，然后进行全编译无误后，下载进开发板。

31.9.2添加 I/O 约束

打开管脚约束表，按下如下表 31-7 所示的内容，对引脚进行约束。

表 31-7 管脚绑定表

	功能信号	FPGA 管脚编号
基本管脚	clk	T9
	key_in	B16
	uart_rx	U8
	uart_tx	V8
	led[1]	C14
	led[0]	D14
	ds	H4
	sh_cp	F3
	st_cp	F4
	reset_n	A15
I^2C 管脚	i2c_sclk	R3
	i2c_sdat	T3

我们将生成的配置文件配置到 FPGA 后，我们打开串口猎人，将收码区切换成字符串显示，之后我们就能看到收码区每一秒钟都会收到一次日期和时间，说明时间发送部分没有问题。

图 31-12 串口接收工具接收时间效果图

此时我们来看数码管上的显示，左边是按下按键 S0 的显示日期，右边是释放按键的显示时间

图 31-13 时间日期数码管显示效果图

接下来我们来测试一下时间校准功能是否正常。例如：现在想将时钟芯片设置为 2019 年 11 月 7 日，12 时 30 分 24 秒，我们来看一下这一天是周四，那么此时我们可以通过串口猎人输入帧头+数据的格式来校准时间，输入数据为 01 03 02 04 19 11 07 04 12 30 24，如图 31-14 所示，我们通过串口猎人已经将时间校准到了刚刚设定的时间了。

图 31-14 通过串口助手工具设置时间

此时我们来看数码管上的显示，左边是按下按键的显示日期，右边是释放按键的显示时间：

图 31-15 时间日期数码管显示效果图（时间设定后）

当然如果你觉得这些还不够方便的话，这里也提供了一个上位机软件来将电脑时间同步校准到时钟芯片。输入正确的端口号点一下上位机软件上的“更新时间”按钮就能实现时间的同步，软件界面如图 31-16 所示：

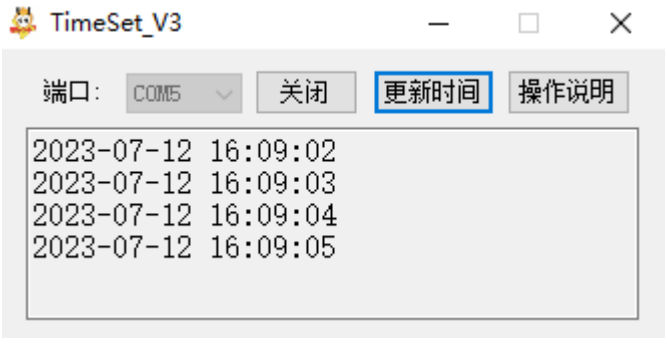


图 31-16 时间更新上位机设置界面

经过测试，时间校准功能也是没有问题的，到了这里，我们的设计经过板级验证也是没有问题了。

31.10扩展实验

学习完本节实验后，可结合“基于 TFT 显示屏的动态数字显示”的实验，将本节中的数码管显示模块替换成 TFT 屏动态数字显示模块，就可以实现时钟日期在 TFT 显示屏上显示的效果。其管脚绑定表如下表 31-8 所示：

表 31-8 管脚绑定表

	功能信号	高云开发板
led 基本管脚	clk50M	T9
	key_in	B16
	uart_rx	U8
	uart_tx	V8
	led[2]	D14
	led[1]	C14
	led[0]	G17
	reset_n	A15
I²C 管脚	i2c_sclk	R3
	i2c_sdat	T3
5 寸 TFT 屏模块	TFT_clk	M14
	TFT_de	U18
	TFT_pwm	U17
	TFT_hs	T18
	TFT_vs	T17
	TFT_rgb[15]	H12
	TFT_rgb[14]	G13
	TFT_rgb[13]	F15
	TFT_rgb[12]	F16
	TFT_rgb[11]	E18
	TFT_rgb[10]	L15
	TFT_rgb[9]	L16
	TFT_rgb[8]	L14
	TFT_rgb[7]	M13
	TFT_rgb[6]	J16
	TFT_rgb[5]	H15
	TFT_rgb[4]	N14
	TFT_rgb[3]	N15
	TFT_rgb[2]	N16
	TFT_rgb[1]	M16
	TFT_rgb[0]	M18

替换显示模块后的系统框图如下

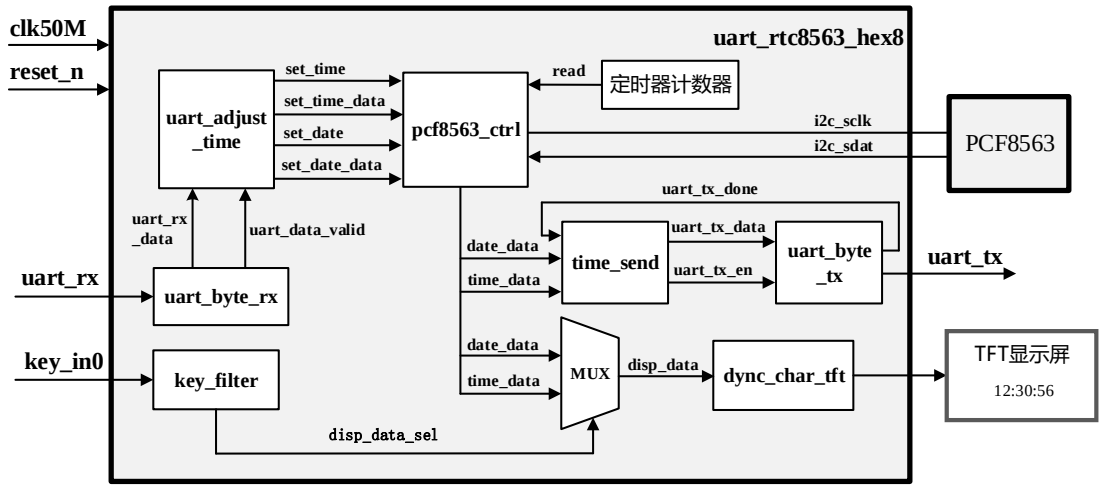


图 31-17 所示。

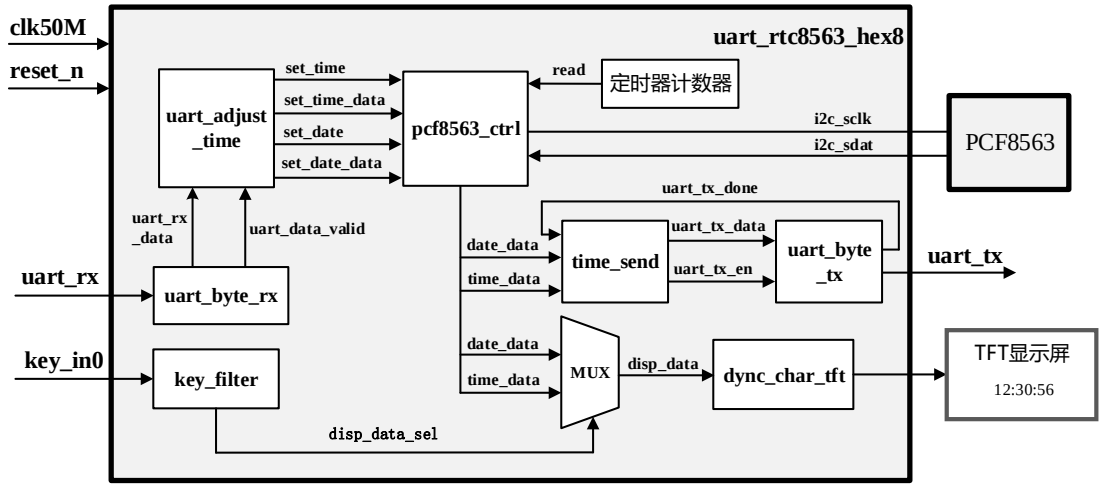


图 31-17 系统框图

本实验的中各模块在前面都已经介绍过了。具体实现直接参见提供的工程代码例程。

本实验板级测试与前面使用数码管显示类似，时间的调整设置与是一样的，有差别的是前面是板子上数码管显示时间日期，本实验是 TFT 显示屏上显示时间日期。具体效果如下。

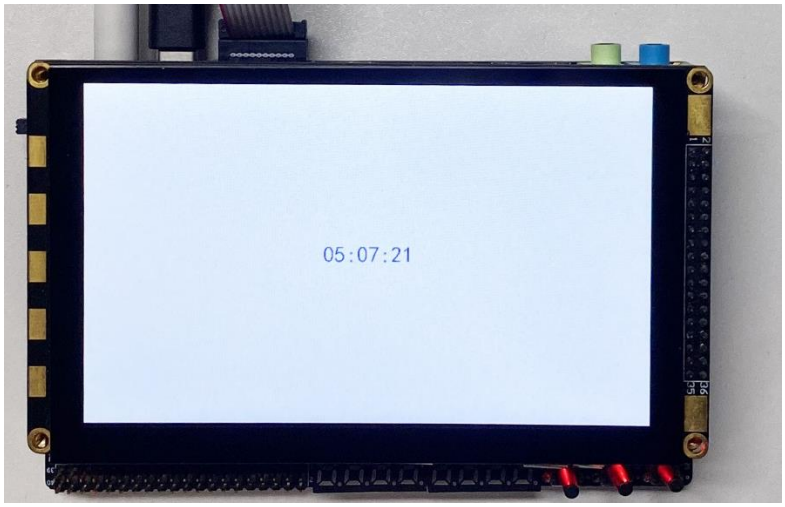


图 31-18 TFT 时间显示效果

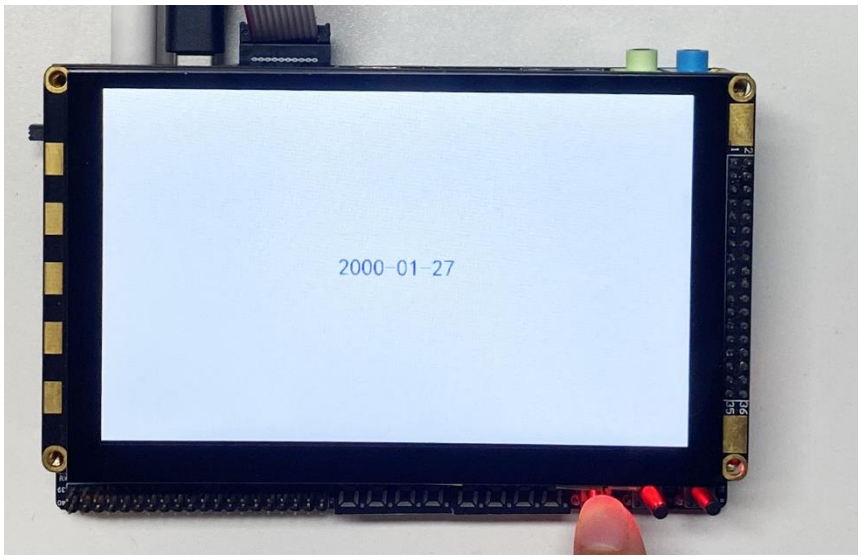


图 31-19 TFT 日期显示效果

我们在例程资料中，准备了液晶屏像素固定模式的程序和液晶屏像素参数模式的程序，供大家参考。大家也可以自己尝试动手利用本节学到的知识和前述章节的模块进行例化，达到上述实验效果。

31.11常见问题分析

本章节需在前述字符的动态显示的基础上，添加支持时间分隔的“冒号”和支持日期分隔的“横线”字库。

当字模库发生变化的时候，需修改字模库头部存储深度，ROM_IP 的存储深度，data_scan_ctrl 对 ROM 的基地址寻址相关语句，顶层的含显示格式的位拼接语句等，同时需要添加每秒读一次时钟的读使能信号。

如果发现本节新增的标点符号无法输出，则逐项排查上述 TFT 显示的修改部分。

31.12总结

本节实验在上一节学习了 I^2C 接口控制器原理与设计基础上，进一步学习了使用 FPGA 内部存储器接收 PCF8563 传递过来的时间信息，显示时间和日期并可以通过串口通信修改时间和日期的案例，以此来将时间信息生成与显示进行一次综合应用。建议读者能够跟随本实验内容，完整的进行整个实验。

32 DDR 控制器的使用

工程源码	|----02_设计实例 |----ch32_DDR3_MC_PHY_1vs4
相关视频课程	
说明	

章节导读

本节将主要介绍 DDR3 的基本工作原理和用途，同时介绍高云 DDR 控制器 IP 的用户使用接口以及官方给的例程的仿真验证。

32.1RAM 存储器发展

DDR 存储器发展到今天，无非是在两个方向上不断探索前进。一是追求更高的存储容量，而是追求更快的读写速度。虽然 DDR3 已不是当今主流的 DDR 存储器，市场上的 DDR4 和 DDR5 也已经层出不穷。但是 DDR3 存储器作为 RAM 存储器家族发展历程中的一个重要里程牌，其学习价值也是非常丰厚的。通过对本小节内容的学习，读者既可以把握 RAM 存储器的发展路线，又可以明确当代主流 DDR 存储器的使用场合，同时，这些知识和内容，还可以作为读者了解更高等级 DDR 存储器的一块敲门砖。

话说 DDR 存储器的发展革沿，得从追溯到早期的 SRAM 存储器开始。我们在数电的课程中也了解过，使用 6 个晶体管的组合，可以实现 1 位数据的存储。

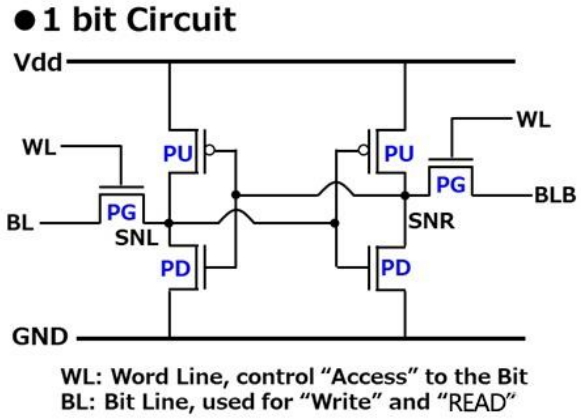


图 32-1 SRAM 使用 6 个晶体管实现 1 位数据的存储示意图

因此，早期 SRAM 芯片对于晶体管的消耗量是巨大的。而影响芯片的成本最大的一个因素就是芯片内部集成的晶体管数量。正是受到 RAM 存储器单位存储容量过高成本的制约，工程师们便想到通过优化芯片内部存储器件的方法，

来降低芯片制造成本，只有通过降低单位 RAM 存储器的制造成本，才能让市场接受更大存储容量的存储器价格。而工程师们最终寻找到的突破口，就是使用 1 个电容+1 个晶体管的组合，实现 1 位数据的存储。

如果给电容两端施加电压，电容两端就会形成一个电势差。如果电容里面储存有电荷，那么这两端就会有电压差，此时，我们就可以认为值存储为 1；如果电荷电容器里面没有电压差，则说明该电容内没有储存电荷，那么这个时候我们就认为它存储的数据为 0。这样，原来的 SRAM 从依靠 6 个晶体管存储 1 位数据，到现在仅需要 1 个晶体管配合 1 个电容就可以存储 1 位数据，单位 RAM 存储容量的制造成本就大幅下降了。而 1 个晶体管配合 1 个电容就是 SDRAM 的 1bit 数据存储硬件开销。现如今，SDRAM 仍然在当前部分缓存容量和缓存数据要求不高的场合，凭借其成本优势具有一定实用价值。

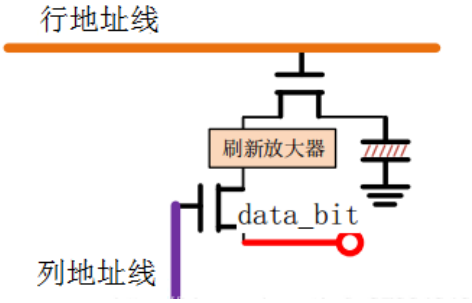


图 32-2 SDRAM 存储 1bit 数据示意图

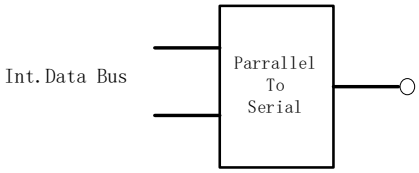
正是存储结构上的创新，存储数据原理的变化，解决了 SRAM 无法做到大容量经济性这样一个问题。因此 SDRAM 最大的特点就是容量大。虽然 SDRAM 相对于 SRAM 的存储容量有大幅提升，但这个提升也仅仅是相对的，从发展的眼光来看，还是无法满足当今日新月异的电子设备数据缓存需求。况且，虽然 SDRAM 相比于 SRAM 在容量得到了提升，但是这一轮进化在器件工作频率的提升效果上，并不明显。这样，RAM 存储器的发展，就来到了 DDR SDRAM 设计阶段。

相较于 SDRAM，DDR SDRAM 的核心技术在于存储单元的读写速率和存储器接口的读写速率分离。

在 SDRAM 阶段，存储单元的读写速率和存储接口的读写速率近乎相同，且存储单元的读写速率也遇到读写速率的瓶颈，很难再有提升空间。下一步，只能通过在存储单元的存储周期不变的情况下，通过改变整个存储器的接口设计策略，让存储器接口在同一个存储单元的存储周期读写 2bit 的数据，同时，重新设计存储器接口，让存储器接口的读写速率为存储单元的读写速率的 2 倍。

这就是 DDR SDRAM 的核心设计思想。

从时钟的角度来分析，DDR SDRAM 在发挥时钟信号的工作效率上作文章，虽然存储单元的读写速率很难提升，但经过优化后的存储器接口，可以让数据的交互既发生在时钟的上升沿，又发生在时钟的下降沿，同时，将存储单元内的 2bit 数据作为操作对象。通过这样一种改进，一个存储器它的吞吐速率就变成了原来的两倍。



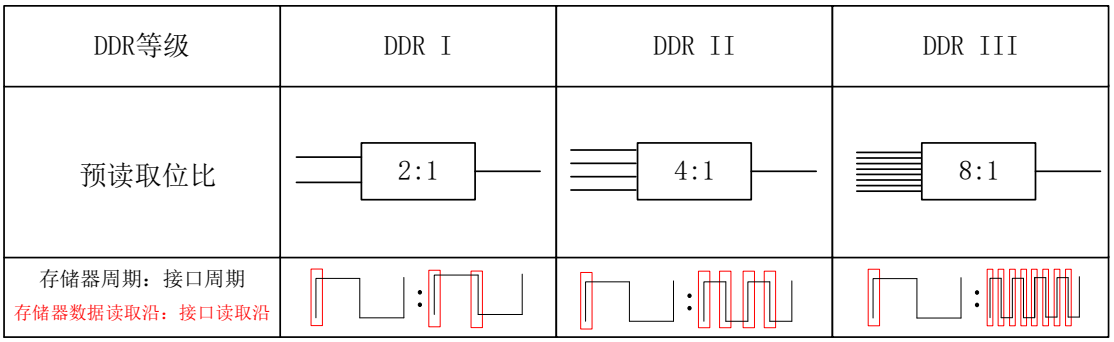


图 32-3 DDR 存储器迭代与数据预读取量、接口读取周期对应关系

这里，有读者就有疑问了。如果不提高 DDR 存储器的接口制造工艺，而单纯的给以更高的接口读写时钟频率，这样是否可行呢？回答是否定的。当存储器件的接口工艺没有换代升级，存储器接收到更高频率的读写时钟信号后，很有可能会出现时序的紊乱，即数据存储时序无法收敛。

这样，介绍完 RAM 存储器的历史革沿，不断发展的 DDR 存储器通过迭代升级，既实现了存储容量的增长，又实现了存储速率的增长。

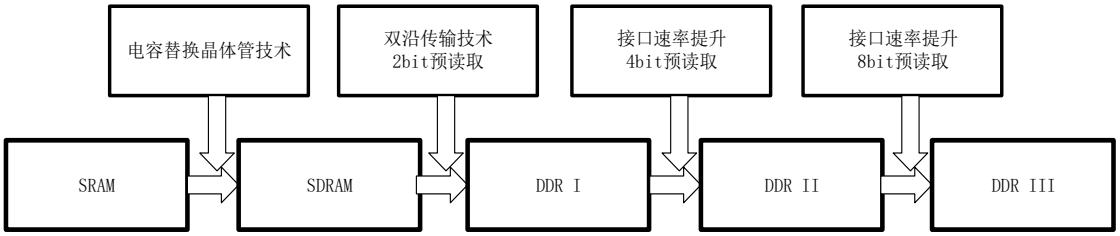


图 32-4 RAM 存储器发展革沿概略图

讲到这里，我们可以一起来举例分析一下，在综合成本、存储容量和读写速率的各因素条件下，如何选择缓存器件呢？

举例来说，我们想进行一次 1080P 的 30FPS 图像数据 5 秒钟的存储。那么

需要多大的存储容量呢？经过计算，一帧 1080P 图像需要 5MByte 的存储空间，那么完成上述任务，就需要 150MByte 的存储空间。而这样一个存储空间的选择，从存储容量，存储速率和经济成本各个角度来权衡，使用 DDR3 存储器是合理可行的。

在我们后续的课程讲解中，基于 FPGA 的 DDR3 应用案例，主要有两大类。第一类是图像传输缓存及图像处理系统，第二类是高速数据采集系统的应用。

第一类应用案例，主要是利用 DDR3 的大容量存储空间用于存放临时数据，在工程设计中，这些临时数据会被用来二次加工和提取关键信息，而加工和提取的过程，又有可能涉及到存储容量的消耗。第二类应用案例，主要是解决数据采集与发送的速率差异。由于数据收发速率的不同，如果不加控制处理，数据的读出必然会存在溢出（写的快）或间歇（读的快）的现象，DDR3 存储器通过将采集到的数据缓存起来，数据能做到集中采集，集中发送，这样，数据采集的可靠性就会获得大大提高。数据采集之所以要用到 DDR3，第一因为它数据存储量大，第二它的数据读写速率快。

随着技术的迭代升级，DDR 存储器的控制信号越来越多，管脚控制也越来越复杂。然而作为用户，我们并不需要自己去设计 DDR 存储器的控制器。Gowin 官方已经给我们提供了成熟的 DDR3 控制器 IP 核，我们可以通过创建 DDR3 控制器 IP 核，实现对 DDR3 控制器的存储控制。用户在实际应用中，只需要关心对存储器写入和读出的数据，关心一些基本的时钟、复位和使能信号，即可实现对数据存储的操作。

32.2高云 DDR3 IP 简介

Gowin DDR3 Memory Interface IP 是一个通用的 DDR3 内存接口 IP，符合 JEDS79-3F 标准协议。该 IP 包含 DDR3 内存控制器（Memory Controller，MC）与对应的物理层接口（Physical Interface，PHY）设计。Gowin DDR3 Memory Interface IP 为用户提供一个通用的命令接口，使其与内存芯片进行互连，完成用户的访存需求。其整体结构如下图 32-5 所示。

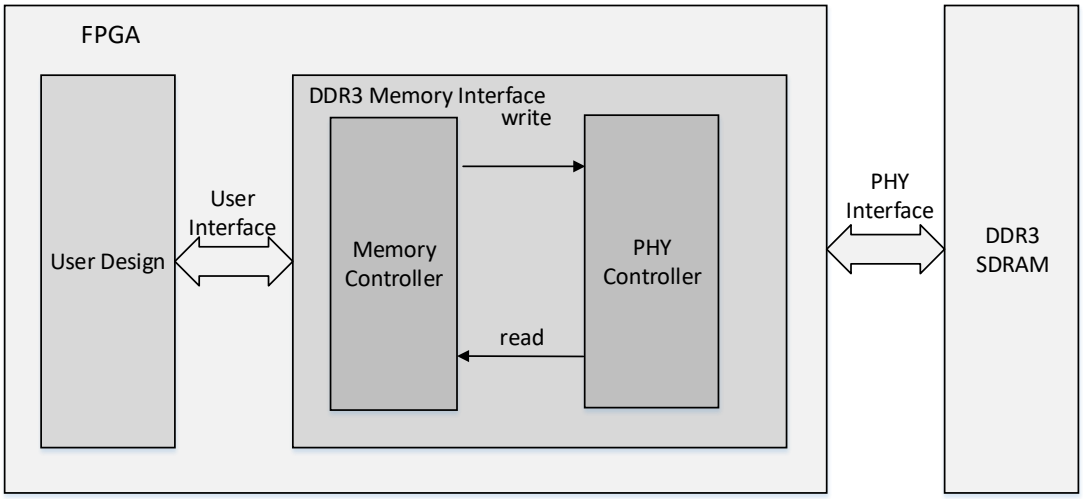


图 32-5 Gowin DDR3 Memory Interface IP 结构图

上述图中 DDR3 Memory Interface 内部包含两个模块 Memory Controller 和 PHY Controller，下面对这两个模块进行简要介绍：

1. Memory Controller

Memory Controller 属 MC 层，实现协议层功能，内部状态机进行 BANK、ROW、COL 及刷新控制。Memory Controller 接收用户侧读写命令，内部以 FIFO 逻辑存储，将读写命令转换为 PHY 侧可识别的接口时序，输入到 PHY 侧。

2. PHY Controller

PHY 提供了 MC 与外部 DDR3 SDRAM 之间的物理层定义与接口，接收来自 MC 层内存控制器的命令，并向 DDR3 SDRAM 颗粒提供接口时序。

PHY 的基本结构如下图 32-6 所示，主要包括四个模块，分别为初始化模块、数据通路、命令地址控制通路和 I/O 逻辑模块。

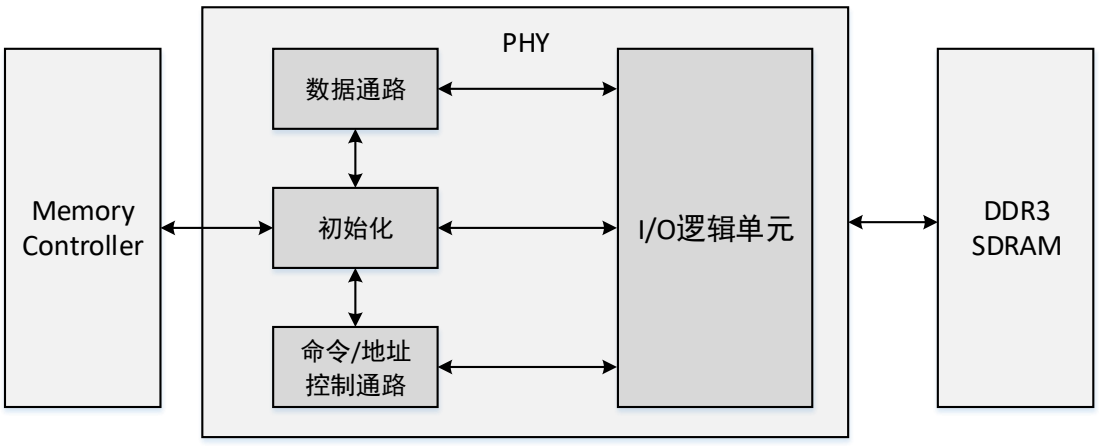


图 32-6 DDR3 PHY 基本结构图

初始化模块主要完成 DDR3 SDRAM 上电后的初始化和读校准。在完成所有初始化和读校准之后，信号“init_calib_complete”会变高，代表整个初始化完成，需要注意的时候，在“init_calib_complete”信号被拉高之前，不允许执行读/写操作。数据通路单元包括写数据和读数据过程。控制通路单元接收 MC 发送的命令和地址信号，并与数据通路配合，处理写、读数据时延参数，并将命令发送到 I/O 逻辑模块。I/O 逻辑模块主要是对数据通路和命令/地址通路传递过来的数据、命令、地址信号进行时钟域的转换。

32.2.1DDR3 IP 配置

首先，我们找到 Gowin DDR3 Memory Interface 这个 IP，进入 Gowin 软件，点击  界面，如果没有打开工程，直接打开的该软件，在 Target Device 中找到芯片类型，比如我们这里选择 GW5A-LV25UG324ES，然后在搜索栏中输入 DDR3，可以看到 Gowin DDR3 Memory Interface 这个 IP，如下图 32-7 所示。

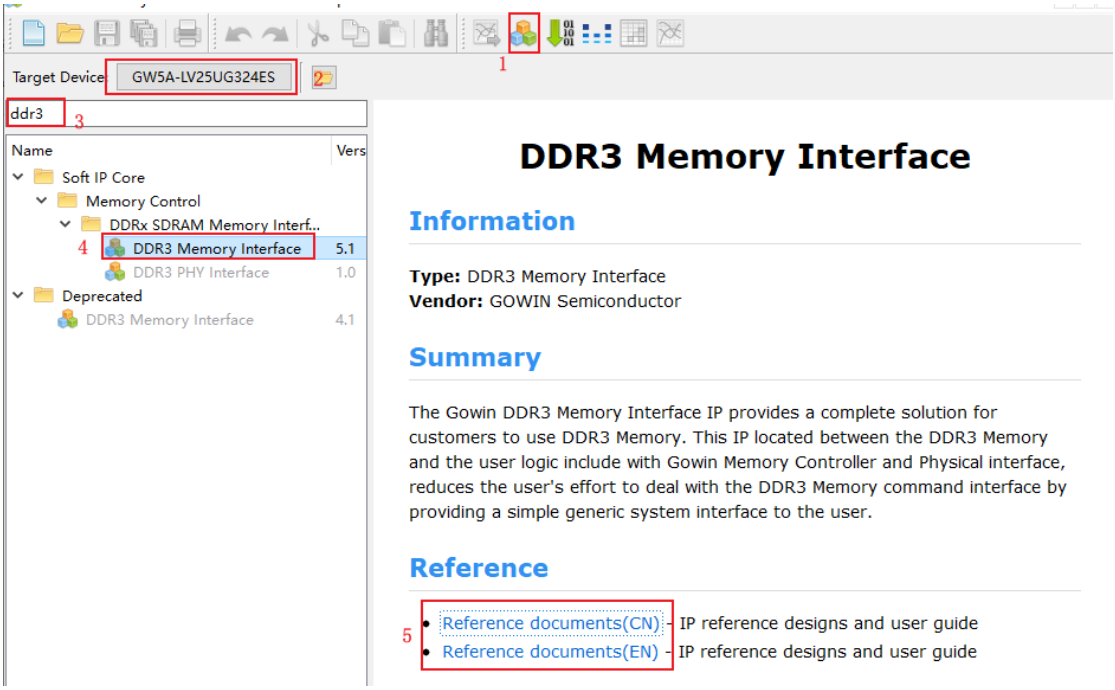


图 32-7 找到 DDR3 IP

上述图中的 Reference 中就有对这个 IP 的介绍，分为中文版本（CN）和英文版本（EN），点击可进入 Gowin 官网对该 IP 的介绍和例程，后面在仿真章节中将会对官方给的例程进行说明，这里双击进入 DDR3 IP 配置界面，如下图 32-8 所示。

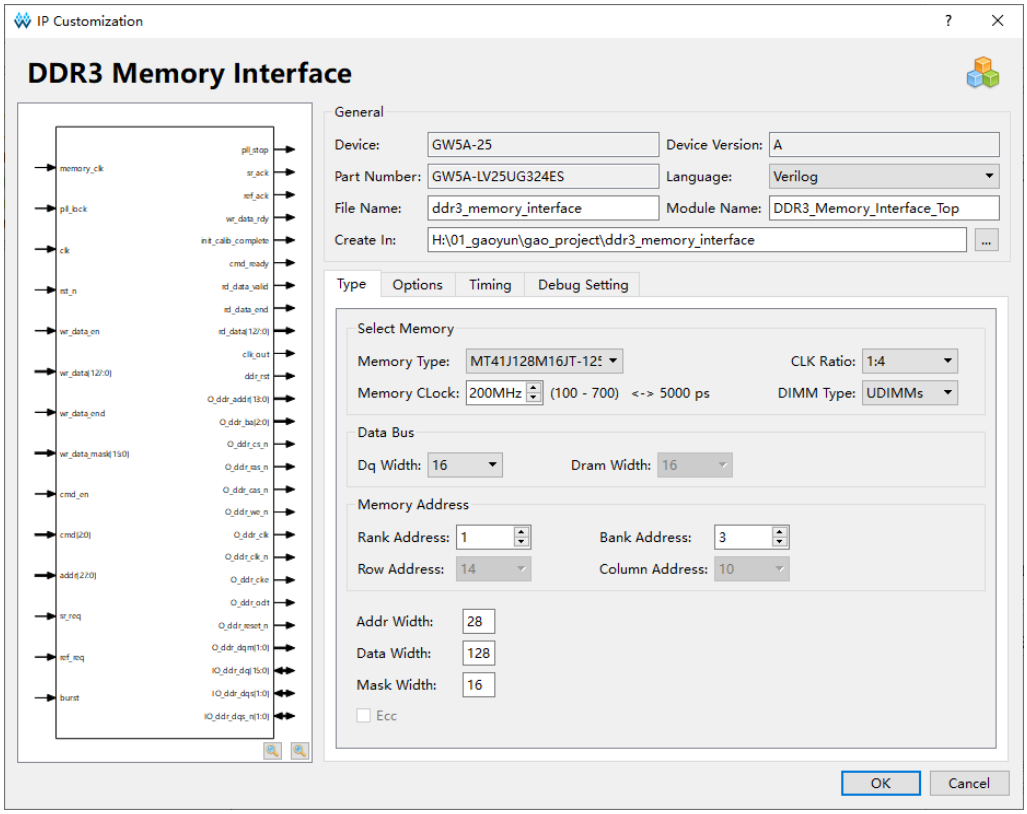


图 32-8 DDR3 IP 配置界面

下面对 IP 各项配置进行说明。

1. Type

Type 配置框如下图 32-9 所示。

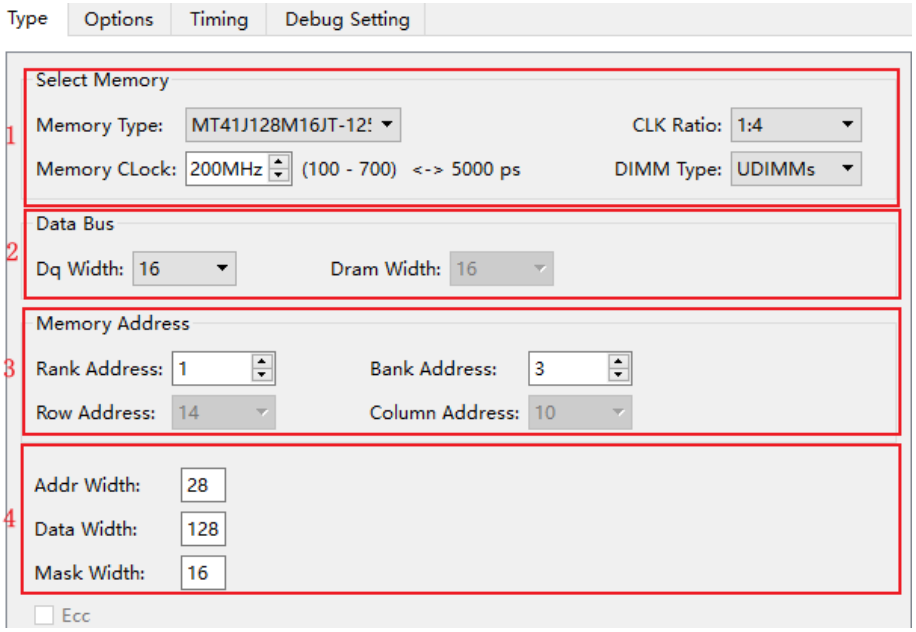


图 32-9 Type 配置框

（1）选择内存（Select Memory）

- Memory Type：存储器类型，有两个选项可供选择，一个是 MT41J128M16JT-125k，一个是 Custom，这个和板载的 DDR 存储芯片有关，如果使用的是 MT41J128M16JT-125k 一致或者兼容这个型号的 DDR3 都可以按照默认选择 MT41J128M16JT-125k，如果使用的 DDR 型号不一致，可以点击 Custom 进行配置，我们高云开发板上使用的 DDR3 的存储芯片为 F60C1A0002-M6AR，该芯片可以兼容 MT41J128M16JT-125k，所以我们这里默认选择 MT41J128M16JT-125k。
- Memory Clock：颗粒接口时钟频率，根据颗粒工作时钟及需求写入，使用 GW2A 器件时，该时钟可以是 pll 的输出时钟或其他时钟；使用 GW5A 器件时，此时钟必须由 PLL 的 clkout2 输出。
- CLK Ratio：用户接口时钟频率与颗粒接口时钟频率比值，两种方式 1：4 或者 1：2，5A 器件只支持 1：4。
- DIMM Type：颗粒 DIMM 类型，支持 Components、RDIMMs，UDIMMs，SODIMMs。这里按照默认选择 UDIMMs 即可。

（2）数据总线（Data Bus）

- Dq Width：Dq 数据位宽，支持 8，16，24，32，40，48，56，64，72，当选择 Memory Type 为 MT41J128M16JT-125k 时，此选项为 16。
- Dram Width：单颗粒的数据位宽，支持 8、16，当选择 Memory Type 为 MT41J128M16JT-125k 时，此选项不可选。

（3）内存地址（Memory Address）

- Rand Address：Rank 地址，对于 Single 与 Dual rank 器件，此选择为 1。
- Bank Address：内存 BANK 地址宽度，根据 DDR3 SDRAM 芯片选择。
- Row Address：内存行地址宽度，根据 DDR3 SDRAM 芯片选择。
- Column Address：内存列地址宽度，根据 DDR3 SDRAM 芯片选择。

（4）其他配置

- Addr Width：地址位宽，28 位，根据前面配置有关
- Data Width：数据位宽，CLK Ratio 为 1：2 时，数据位宽为 64，CLK Ratio 为 1：2 时，数据位宽为 128。
- Mask Width：CLK Ratio 为 1：2 时，位宽为 8，CLK Ratio 为 1：2 时，

位宽为 16。

2. Options

Options 配置界面如下图 32-10 所示。

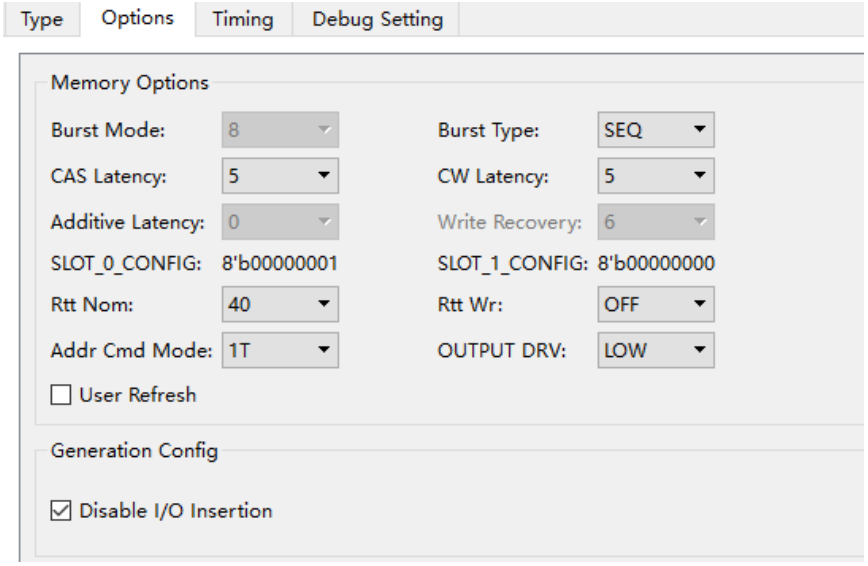


图 32-10 Options 配置界面

下面对 Options 界面比较重要的几个选项进行说明。

- Burst Mode：颗粒突发模式，支持“4”、“8”、“OTF”；时钟比例 1：2 时，支持 4/OTF，时钟比例 1：4 时只支持 8，所以 5A 器件该选项不可选，默认就是 8。
- Burst Type：颗粒突发类型，支持 SEQ（Sequential）和 INT（Interleaved）。
- CAS Latency：CAS 延迟时间，可选 5、6、7、8。
- Additive Latency：附加延迟时间。
- CW Latency：CWL 延迟时间，根据实际选择。
- RTT NOM：Nominal ODT 数值，OFF”: OFF；“20”: 20；“30”: 30；“40”: 40；“60”: 60；“120”: 120。
- RTT WR：Multiple-RANK 中用于写端口的 Dynamic ODT 的数值，对于 Single-Component 设计 RTT_WR 无效，“OFF”: RTT_WR disabled、“120”: RZQ/2、“60”: RZQ/4。
- USER_REFRESH：是否由用户自己控制刷新操作，勾选就代表由用户自己控制，本次实验我们不使用该功能，读者如果想要是应用该功能，

请自行查看官方文档给出的说明。

3. Timing

Timing 的配置界面与实际使用的 DDR 芯片有关，根据自己使用的 DDR3 芯片型号的数据手册进行修改，使用我们的开发板进行实验时，按照默认的配置即可。如下图 32-11 所示。

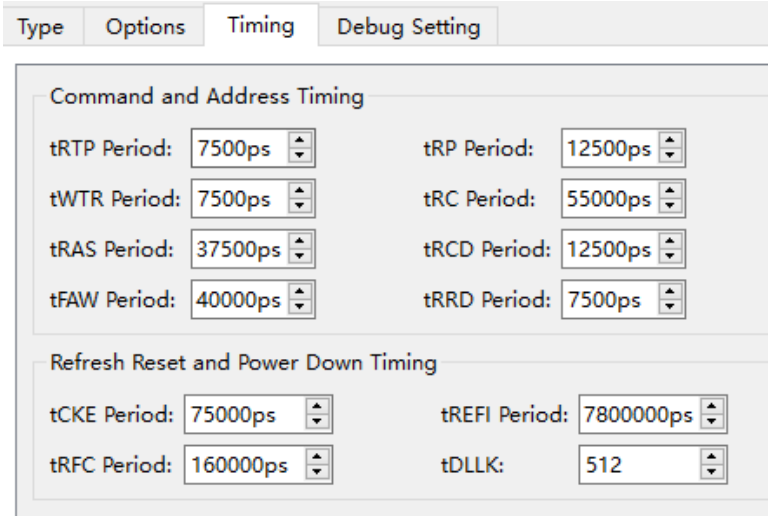


图 32-11 Timing 配置界面

4. Debug Setting

此选项是使能 debug 接口和调试参数，读者根据实际需要选择使用，本次实验不使用该功能。

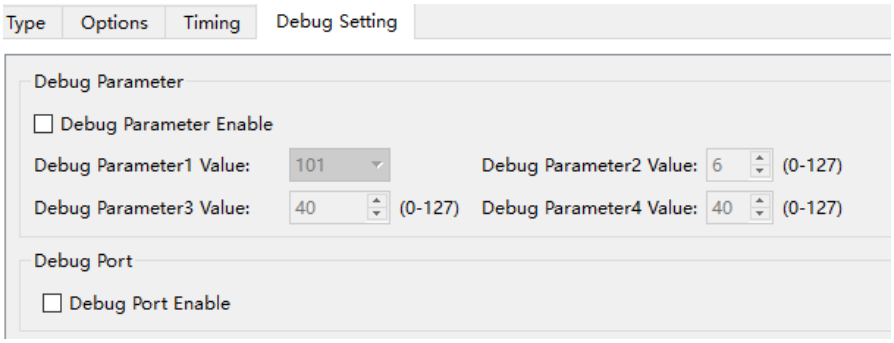


图 32-12 Debug Setting 配置界面

最后点击 OK，将会生成 DDR3_Memory_Interface_Top IP 的例化代码，如下所示。

```
DDR3_Memory_Interface_Top your_instance_name(
    .clk(clk_i), //input clk
    .pll_stop(pll_stop_o), //output pll_stop
    .memory_clk(memory_clk_i), //input memory_clk
```

```
    .pll_lock(pll_lock_i), //input pll_lock
    .rst_n(rst_n_i), //input rst_n
    .cmd_ready(cmd_ready_o), //output cmd_ready
    .cmd(cmd_i), //input [2:0] cmd
    .cmd_en(cmd_en_i), //input cmd_en
    .addr(addr_i), //input [27:0] addr
    .wr_data_rdy(wr_data_rdy_o), //output wr_data_rdy
    .wr_data(wr_data_i), //input [127:0] wr_data
    .wr_data_en(wr_data_en_i), //input wr_data_en
    .wr_data_end(wr_data_end_i), //input wr_data_end
    .wr_data_mask(wr_data_mask_i), //input [15:0] wr_data_mask
    .rd_data(rd_data_o), //output [127:0] rd_data
    .rd_data_valid(rd_data_valid_o), //output rd_data_valid
    .rd_data_end(rd_data_end_o), //output rd_data_end
    .sr_req(sr_req_i), //input sr_req
    .ref_req(ref_req_i), //input ref_req
    .sr_ack(sr_ack_o), //output sr_ack
    .ref_ack(ref_ack_o), //output ref_ack
    .init_calib_complete(init_calib_complete_o),
    .clk_out(clk_out_o), //output clk_out
    .ddr_rst(ddr_rst_o), //output ddr_rst
    .burst(burst_i), //input burst
    .O_ddr_addr(O_ddr_addr_o), //output [13:0] O_ddr_addr
    .O_ddr_ba(O_ddr_ba_o), //output [2:0] O_ddr_ba
    .O_ddr_cs_n(O_ddr_cs_n_o), //output O_ddr_cs_n
    .O_ddr_ras_n(O_ddr_ras_n_o), //output O_ddr_ras_n
    .O_ddr_cas_n(O_ddr_cas_n_o), //output O_ddr_cas_n
    .O_ddr_we_n(O_ddr_we_n_o), //output O_ddr_we_n
    .O_ddr_clk(O_ddr_clk_o), //output O_ddr_clk
    .O_ddr_clk_n(O_ddr_clk_n_o), //output O_ddr_clk_n
    .O_ddr_cke(O_ddr_cke_o), //output O_ddr_cke
    .O_ddr_odt(O_ddr_odt_o), //output O_ddr_odt
    .O_ddr_reset_n(O_ddr_reset_n_o), //output O_ddr_reset_n
    .O_ddr_dqm(O_ddr_dqm_o), //output [1:0] O_ddr_dqm
    .IO_ddr_dq(IO_ddr_dq_io), //inout [15:0] IO_ddr_dq
    .IO_ddr_dqs(IO_ddr_dqs_io), //inout [1:0] IO_ddr_dqs
    .IO_ddr_dqs_n(IO_ddr_dqs_n_io) //inout [1:0] IO_ddr_dqs_n
   );
```

下面将对该 IP 的接口信号进行说明。

32.2.2用户接口信号说明

32.2.2.1初始化接口

init_calib_complete：DDR3 SDRAM 必须经过校准操作才能进行正常的写、

读操作。因此上电后 PHY 会对 DDR3 SDRAM 进行初始化校准操作，初始化完成后 init_calib_complete 拉高，时序图如下图 32-13 所示。

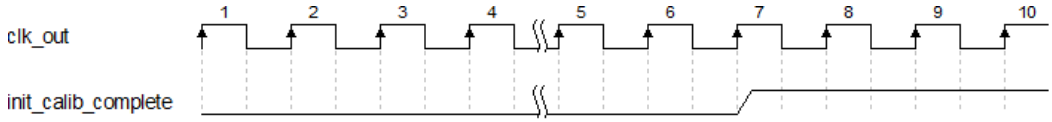


图 32-13 初始化完成信号时序图

32.2.2.2命令和地址接口

cmd：命令端口， cmd 信号的位宽为 3，当 cmd 等于 3’b001 代表此时为读操作，当 cmd 为 3’b000 代表此时为写操作；

cmd_en：命令使能信号，高电平时 cmd 有效；

addr：用户侧地址总线，与 cmd 一同写入控制器中，当 cmd_en 有效时，addr 有效，addr 为 DDR 地址，即 addr 直接反映 DDR 内存地址。当 DDR3 burst_mode 配置 BC4 时，一次写入\读取 4 个 dq 数据，因此一次 DDR 写/读占用 4 个地址；当 DDR3 burst_mode 配置 BL8 时，一次写入\读取 8 个 dq 数据，因此一次 DDR 写/读占用 8 个地址。用户在使用过程中，应注意对地址的控制。

cmd_ready：当 cmd_ready 信号为高电平时，代表此时 DDR 控制器可以接收用户命令。

命令、地址以及使能信号之间的时序如下图 32-14 所示。

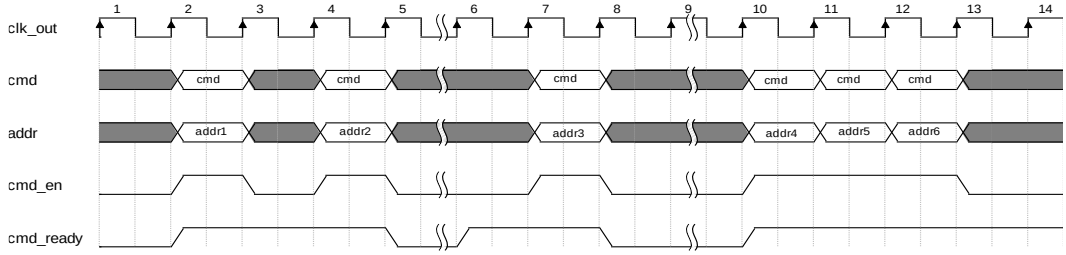


图 32-14 命令、地址以及使能信号之间的时序图

32.2.2.3写数据接口

wr_data：数据总线接口，用户可通过此接口写入需要存储 DDR 内的数据。

wr_data_en：数据写入使能接口、高电平时 wr_data 有效。

wr_data_end：表明当前周期 wr_data 总线上的数据是当前写入的最后一个数据。

wr_data_rdy：当 wr_data_rdy 为高电平时，表示控制器可以接收 user 数据，用户可通过接口 wr_data、wr_data_en 和 wr_data_end 将数据写入控制器。

按照我们的配置，时钟比例为 1：4，Burst_Mode 为 BL8，写数据时序图如下图 32-15 所示。

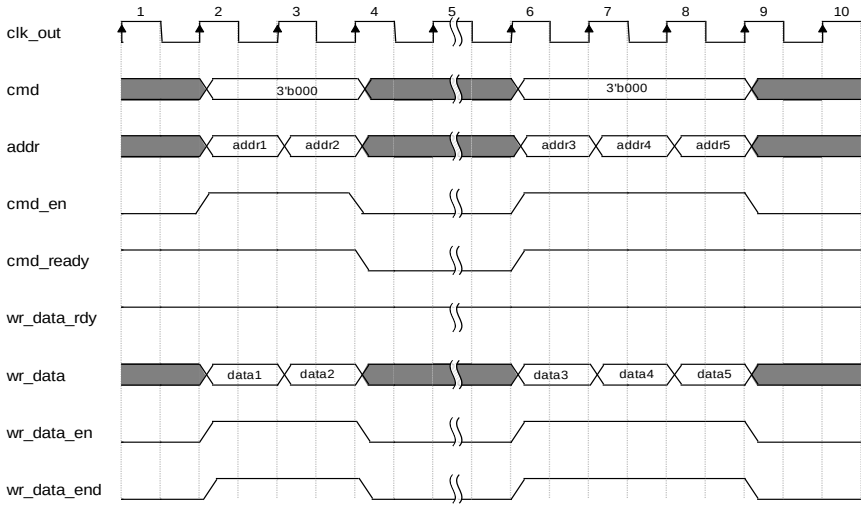


图 32-15 写数据接口时序（1:4，burst mode = BL8）

从上述图中可以看出，当时钟比例为 1：4，burst_mode 配置为 BL8 时，wr_data 与 dq 数据位宽比为 1:4，此时一个 wr_data 可满足 DDR 一次突发写，wr_data_en 与 wr_data_end 的行为相同，所以用户在写数据时将 wr_data_en 与 wr_data_end 同时写 1 即可。其他模式的写数据接口时序请读者自行查看官方 DDR3 IP 用户指南。

32.2.2.4读数据接口

用户可以通过用户接口 rd_data、rd_data_valid 与 rd_data_end 读取 DDR3 SDRAM 返回的数据。

rd_data：从 DDR3 中读取数据的端口。

rd_data_valid：读数据有效端口，当其为高电平时，指示此时返回的 rd_data 有效。

rd_data_end：指示在当前 burst_mode 下所返回的最后一组数据，高电平时有效。

当时钟比例为 1：4 时，读数据依照读命令顺序依次返回数据，时序图如下图 32-16 所示。从图中可以看出，在 cmd_read 信号为高时，代表此时可以传输命令，我们此时将 cmd_en 信号拉高，cmd 信号设置为 3’001，代表此时为读操作，给出命令的同时，我们也必须给出需要读取的数据的地址，当给出以上信号之后，等待一段时间之后，将会从 DDR 中将数据读取出来，此时

rd_data_valid 信号将被拉高，代表此时读取数据有效，rd_data 为读取的有效数据。

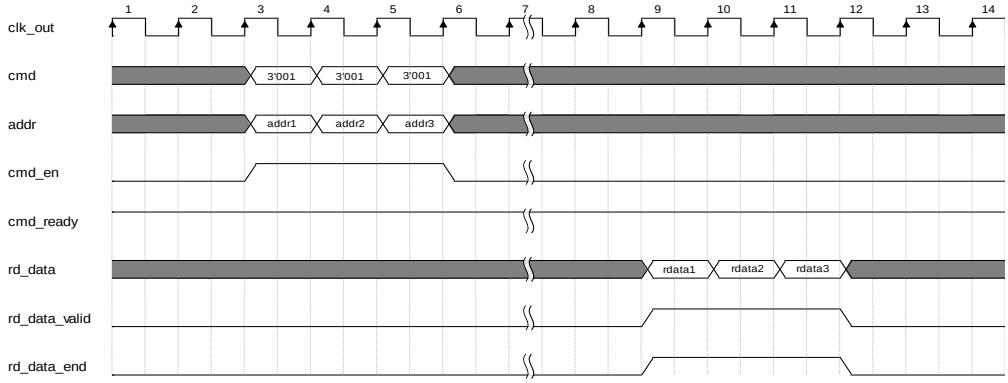


图 32-16 时钟比例 1：4，读数据时序图

32.2.3IP 使用注意事项

32.2.3.1时钟与复位

1. 时钟

IP 有三个时钟，两个输入时钟 clk 和 memory_clk，一个输出时钟 clk_out，如下图 32-17 所示。

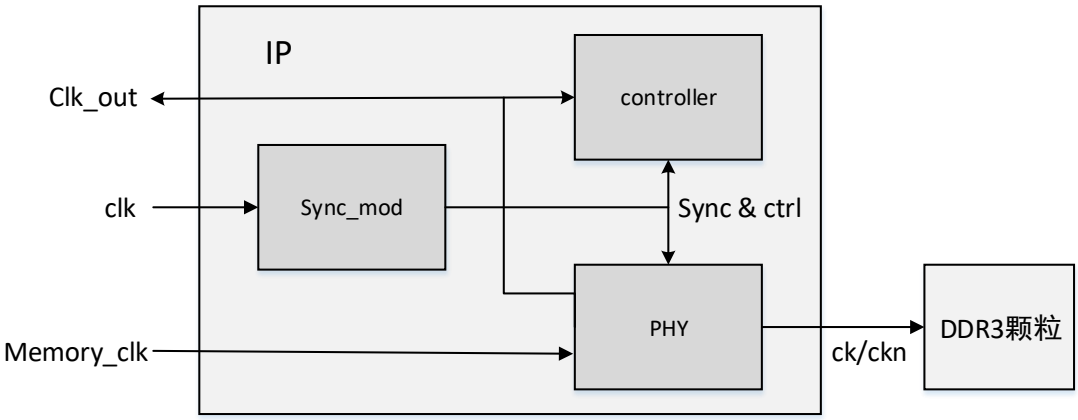


图 32-17 时钟说明框图

上述图中的 clk 用于产生一些同步及控制信号，这些同步及控制信号作用于 IP 的主体逻辑（PHY 层和 Memory controller）。clk 要求是低速的连续时钟，推荐值 50MHz，可将板载仅晶振的输入连接到 clk；memory_clk 是高速时钟，使用 HCLK 资源，驱动 PHY 并向外输出送到 DDR3 颗粒；clk_out 是 memory_clk 的分频时钟，当 clk_ratio=4:1 时，clk_out 是 Memory_clk 的四分频时钟，使用

PCLK 资源，clk_out 作为 IP 的逻辑处理时钟，并向外输出到 user 逻辑，user 对 IP 的接口操作应该与 clk_out 同步。

2. 复位

IP 有输入信号 rst_n 和 pll_lock，输出信号 ddr_rst，如下图 32-18 所示。

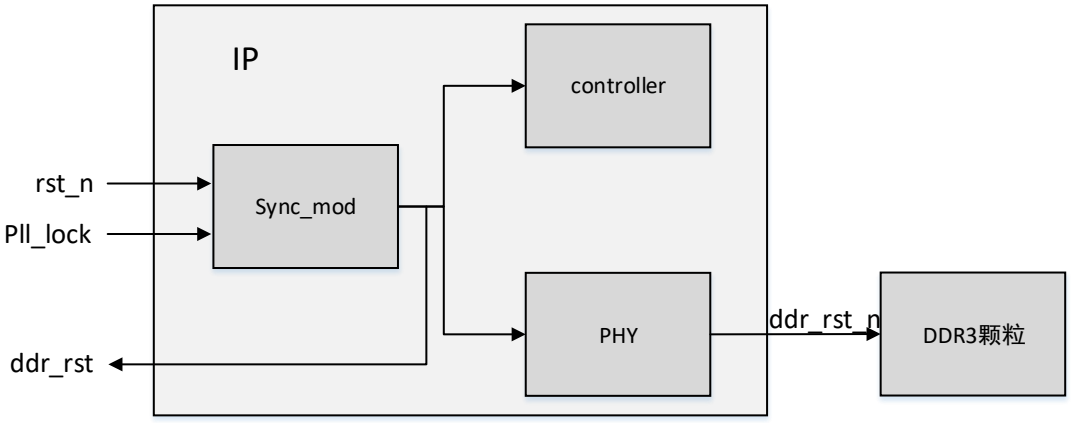


图 32-18 复位信号说明框图

如上图所示，rst_n 和 pll_lock 进行或运算产生 ddr_rst，作为 IP 的全局复位，并向外送到 user。任何复位逻辑都可以接入 rst_n，pll_lock 只能接入 PLL 的 LOCK 信号，如果 pll_lock 没有接入 PLL 的 LOCK 信号，IP 将不能检测时钟是否稳定，此时容易出现 DDR 初始化失败。

32.2.3.2pll_stop 信号

pll_stop 是在 5A（S）（T）器件环境下存在的控制信号，是控制 memory_clk 的开关，低有效，如图 32-19 和图 32-20 所示。

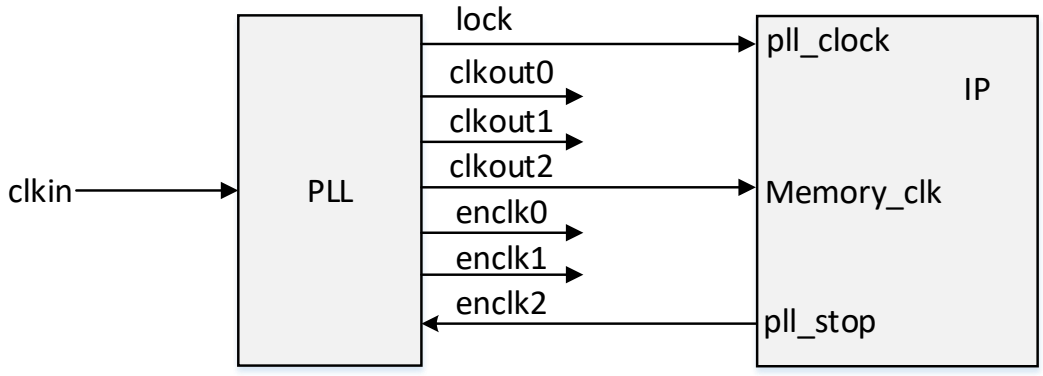


图 32-19 138K pll_stop

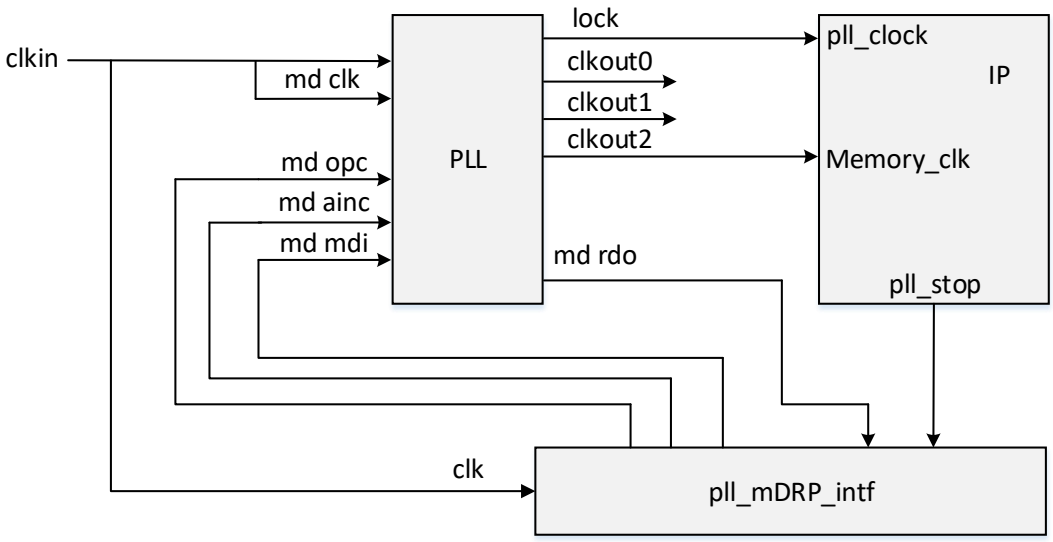


图 32-20 25K pll_stop

如上图所示，使用 138K 器件时，pll_stop 直接接入 PLL 的 enclk2。使用 25K 器件时，pll_stop 需要接入转接模块 pll_mDRP_intf，间接控制 PLL 的 clkout2 输出，pll_mDRP_intf 的 clk 与 PLL 的 mdclk 和 clkin 相同。pll_mDRP_intf 模块在 Gowin_DDR3_Memory_Interface_RefDesign 参考设计中，请从官网下载

32.2.4DDR3 IP 端口说明

在 DDR3 IP 配置界面左边显示了该 IP 的各个端口信号，下面将对这些端口信号进行说明。如下表 32-1 所示为 DDR3 IP 端口说明表。

表 32-1 DDR3 IP 端口说明表

	信号	位宽	方向	描述
User Interface	addr	ADDR_WIDTH	Input	地址输入，信号宽度可以设置为 parameter。
	cmd	3	Input	命令通道
	cmd_en	1	Input	命令与地址使能信号： 0：无效 1：有效
	cmd_ready	1	Output	高电平时指示 Memory Interface 可接收命令与地址
	rd_data	APP_DATA_WIDTH	Output	读数据通道
	rd_data_end	1	Output	高电平时指示当前输出的一组 rd_data 的结束周期
	rd_data_valid	1	Output	rd_data 有效信号： 0：无效 1：有效
	burst	1	Input	OTF 控制端口，为 1'b1 时是 BL8 模式；为 1'b0 时是 BL4 模式，仅 OTF 模式下有效

	wr_data	APP_DATA_WIDTH	Input	写数据通道
	wr_data_end	1	Input	高电平指示当前时钟周期是此组数据 wr_data 的最后一个周期
	wr_data_mask	APP_MASK_WIDTH	Input	wr_data 掩码 0：对应 wr_data 字节无效 1：对应 wr_data 字节有效
	wr_data_rdy	1	Output	高电平时表示 MC 可以接收用户数据
	wr_data_en	1	Input	wr_data 写使能信号： 0：无效； 1：有效
	sr_req	1	Input	自刷新请求
	sr_ack	1	Output	自刷新应答信号
	ref_req	1	Input	用户刷新请求
	ref_ack	1	Output	用户刷新应答信号
	clk	1	Input	参考输入时钟，一般为 PCB 晶振输入，推荐 50M 晶振
	memory_clk	1	Input	用户输入颗粒接口频率，使用 GW2A 器件时，该时钟可以是 pll 的输出时钟或其他时钟；使用 GW5A 器件时，此时钟必须由 PLL 的 clkout2 输出
	pll_stop	1	Output	此端口用法参看前面介绍的内容
	pll_lock	1	Input	如果 memory_clk 为 PLL 倍频输入，此接口接 PLL 的 pll_lock 管脚，如果用户不使用 PLL，此接口接高电平
	rst_n	1	Input	系统复位输入信号： 0：有效 1：无效
	init_calib_complete	1	Output	初始化完成信号
	clk_out	1	Output	用户设计时钟
	ecc_err	APP_DATA_WIDTH/32	Output	ECC 指示信号输出
	ddr_rst	1	Output	经过 IP 处理过的复位信号，供用户设计使用，高复位
DDR3 SDRAM Interface	O_ddr_addr	ROW_WIDTH	Output	Row 地址（激活命令）、Column 地址（读、写命令）
	O_ddr_bank	BANK_WIDTH	Output	Bank 地址
	O_ddr_cs_n	CS_WIDTH	Output	片选，低有效
	O_ddr_ras_n	1	Output	Row 地址选通信号
	O_ddr_cas_n	1	Output	Column 地址选通信号
	O_ddr_we_n	1	Output	Row 写使能
	O_ddr_ck	CK_WIDTH	Output	提供给 DDR3 SDRAM 的时钟信号
	O_ddr_ck_n	CK_WIDTH	Output	与 ddr_ck 组成差分信号
	O_ddr_cke	CKE_WIDTH	Output	DDR3 SDRAM 时钟使能信号
	O_ddr_odt	ODT_WIDTH	Output	内存信号端接电阻控制
	O_ddr_reset_n	1	Output	DDR3 SDRAM 复位信号

	O_ddr_dm	DM_WIDTH	Output	DDR3 SDRAM 数据屏蔽信号
	IO_ddr_dq	DQ_WIDTH	Bidirection	DDR3 SDRAM 数据
	IO_ddr_dqs	DQS_WIDTH	Bidirection	DDR3 SDRAM 数据选通信号
	IO_ddr_dqs_n	DQS_WIDTH	Bidirection	与 ddr_dqs 组成差分信号

32.3高云 DDR3 例程说明

32.3.1下载官方提供的例程

我们在前面的内容中，提到过如何找到高云的 DDR3 IP，找到之后，会看到 IP 简介界面下有 Reference 选项，点击 Reference documents(CN)进入官网下载相关例程，操作如下图 32-21 所示。

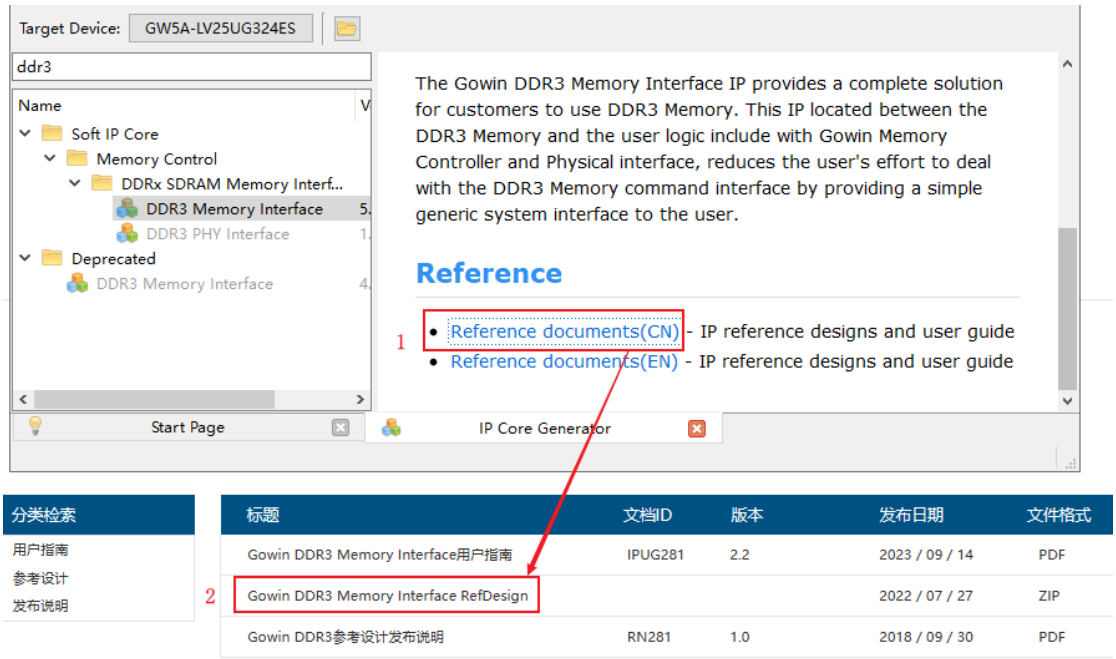


图 32-21 下载官方例程示意图

下载完成之后，将压缩包进行解压，得到两个文件夹：“DDR3_MC_PHY_1vs2”、“DDR3_MC_PHY_1vs4”，一个时钟比例为 1：2，一个时钟比例为 1：4，这里以 1：4 的进行说明，也就是“DDR3_MC_PHY_1vs4”这个文件夹下的工程，如下图 32-22 所示。

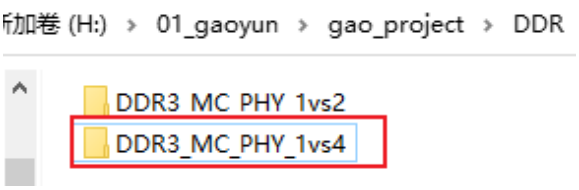


图 32-22 官方提供例程文件

32.3.2DDR3_MC_PHY_1vs4 工程文件说明

打开 DDR3_MC_PHY_1vs4 文件夹，可以看到如下图 32-23 所示的文件夹。

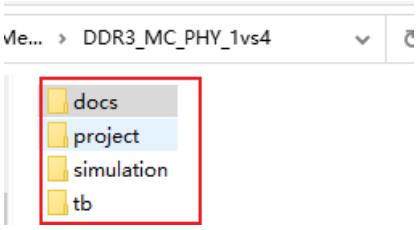


图 32-23 DDR3_MC_PHY_1vs4 文件夹

1. docs：存放一个.txt 文本，里面的写的关于工程的说明。
2. project：存放的 Gowin 关于 DDR3 的例程工程。
3. simulation：存放该工程的仿真工程文件。
4. tb：存放的 tb 和 DDR3 模型文件，需要注意的 tb 文件夹的 sim_model 本来应该存放的关于 DDR3 的仿真模型，但是官方给的例程中，少了这个文件，在仿真的时候将会报错，这里可以在我们提供的例程中提取，添加完成之后，如下图 32-24 所示。

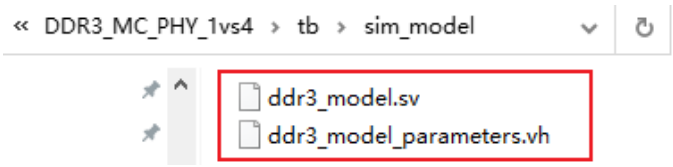


图 32-24 添加 DDR3 仿真模型

32.3.3Gowin 工程说明

点击 Project 文件下存放的工程 ddr3_1v4_hs，点开之后如下图 32-25 所示。

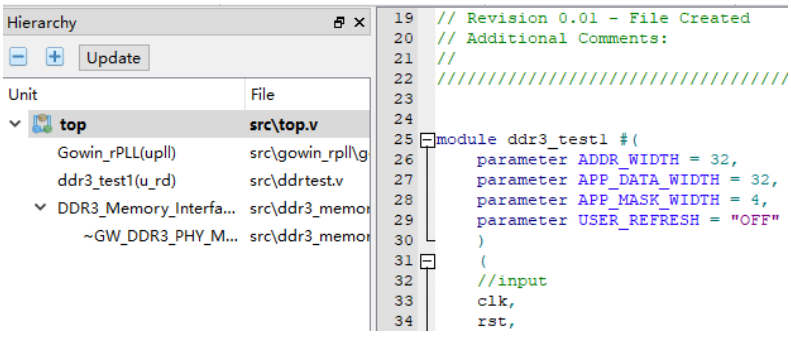


图 32-25 点开 ddr3_1v4_hs 工程

从上述图中可以看出，工程中主要包含 3 个模块：

Gowin_rPLL：PLL IP，主要用于生成 DDR3 IP 的 memory_clk，提供的例程

中 DDR3 的 memory_clk 配置的为 400M，所以这里需要通过 pll 输出一个 400M 的时钟，其输入时钟为 50M，由我们开发板上的晶振提供。我们在介绍 DDR3 IP 的时候，知道 DDR3 IP 有个端口 pll_lock 专门用于连接 PLL IP 的锁定信号，所以生成 PLL IP 的时候，需要使能 LOCK。

DDR3_Memory_Interface_Top：DDR3 IP，配置时钟比例为 1：4，memory_clk 为 400M，具体配置我们可以点开工程的 IP 进行查看，操作如下图 32-26 所示，DDR3 配置界面如下图 32-27 所示。

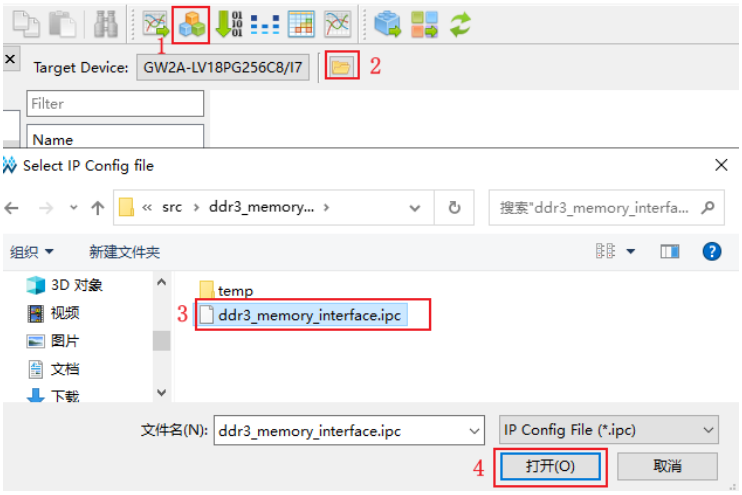


图 32-26 查看 DDR3 IP 配置操作方法

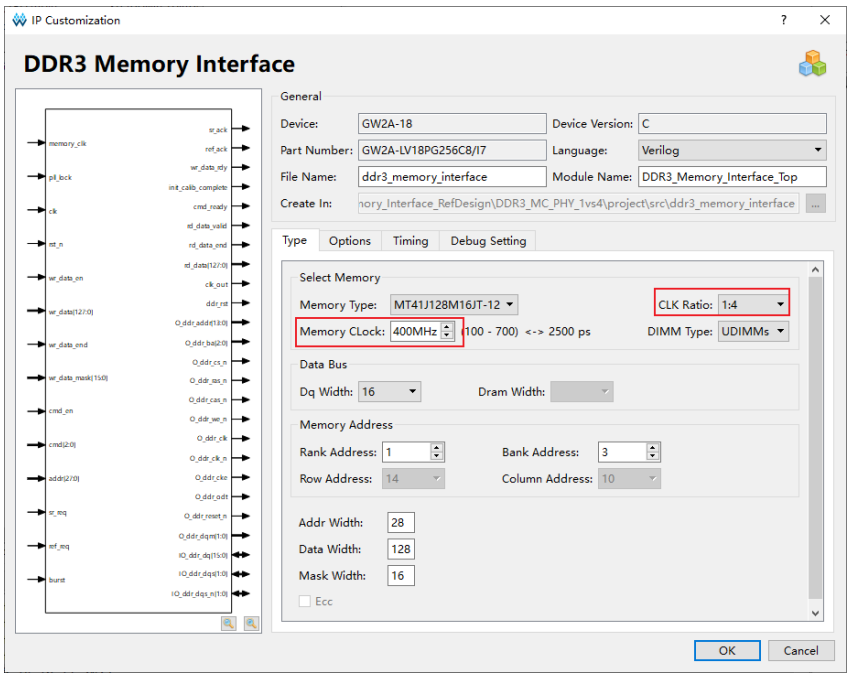


图 32-27 DDR3 IP 配置界面

ddr3_test1：主要生成 DDR3 IP 用户端口的相关信号。

这三个模块之间的结构框图如下图 32-28 所示。

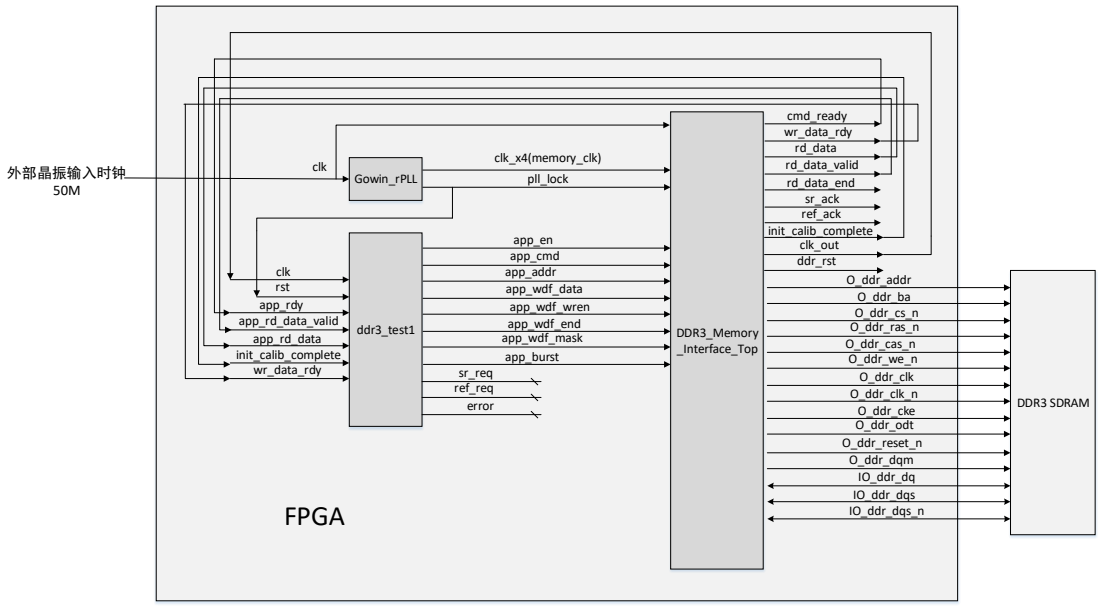


图 32-28 ddr3_1v4_hs 工程内部模块结构框图

上述模块中，需要我们弄清楚的就是 ddr3_test 模块，其内部实现的功能就是对 DDR3 IP 实现读写操作，具体代码我们将不做详细介绍，接下来我们将结合仿真来对该工程实现的功能进行分析。

32.3.4Modelsim 仿真波形说明

32.3.4.1打开 Modelsim 工程

首先我们需要打开 Modelsim 工程，其操作步骤如下所示：

1. 打开 Modelsim 软件，这里 10.7 版本的软件，用户如果电脑上没有安装 Modelsim 软件，可以去我们的论坛上进行下载安装，论坛链接如下所示：

   http://www.corecourse.cn/forum.php?mod=viewthread&tid=28768

2. 依次点击 File->Change Directory...，改变工程目录。

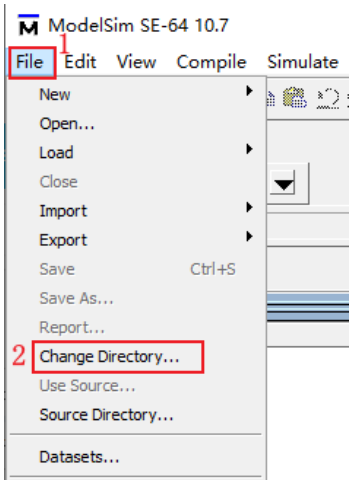


图 32-29 改变工程目录

3. 在弹出的选择文件夹对话框中找到 modesim_sim 文件夹，路径为：DDR3_MC_PHY_1vs4\simulation\modesim_sim，如下图 32-30 所示。

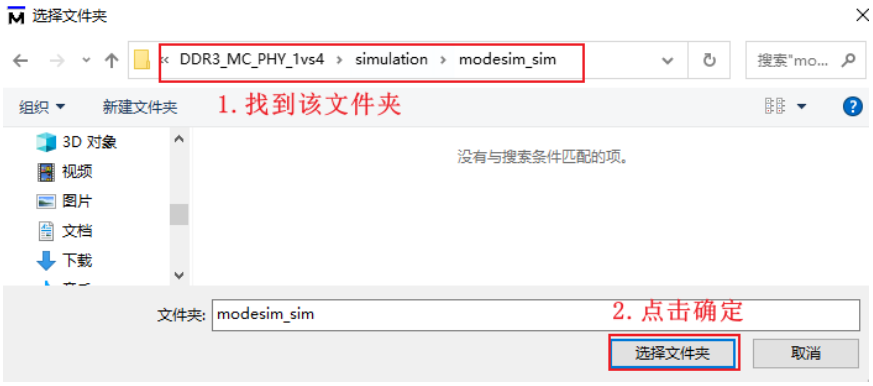


图 32-30 找到仿真文件夹所在位置

4. 输入命令 do cmd.do，操作如下图 32-31 所示。

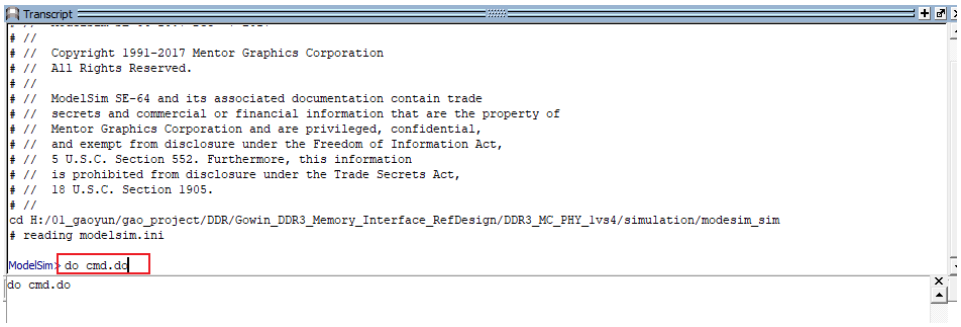


图 32-31 输入运行命令

5. 命令输入完成之后，按下键盘上的回车按键，开始运行工程。运行完成之后，会提示如下图 32-32 错误。

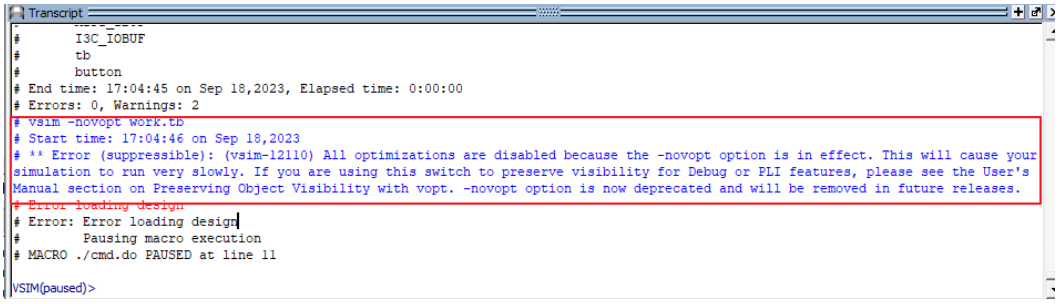


图 32-32 Modelsim 提示错误

6. 出现第 5 步错误的原因是 modelsim 10.7 版本之后都不再使用-novopt，所以只要不适用-novopt 就不会报错，修改方式就是找到 modelsim_sim 文件夹下的 cmd.do 文件，用 notepad++或者记事本打开，将 11 行原先的 vsim -novopt work.tb 修改为 vsim -voptargs="+acc" work.tb 然后保存文件，操作如下图 32-33 所示

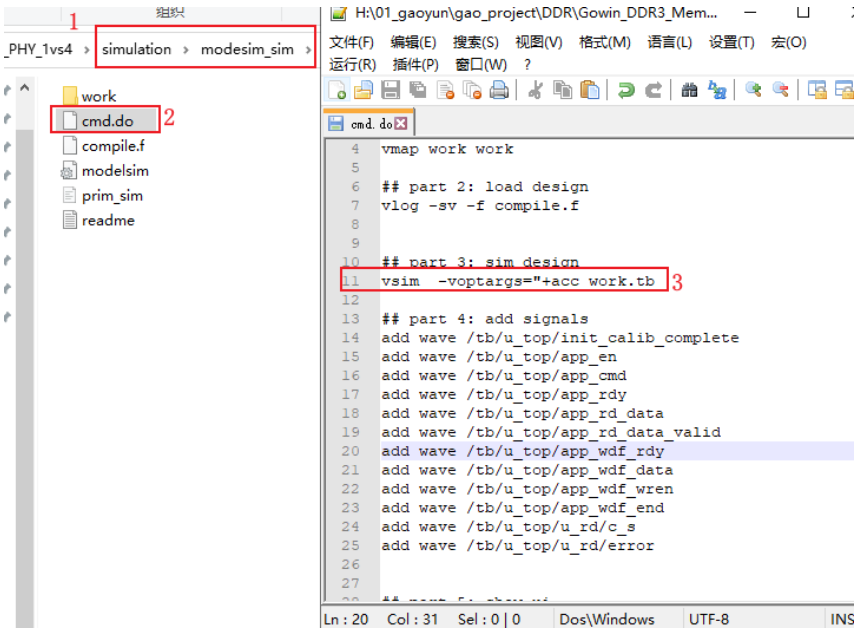


图 32-33 修改 cmd.do 文件内容

7. 然后重新输入命名然后运行，报错消失，弹出波形框，开始运行仿真，如下图 32-34 所示。

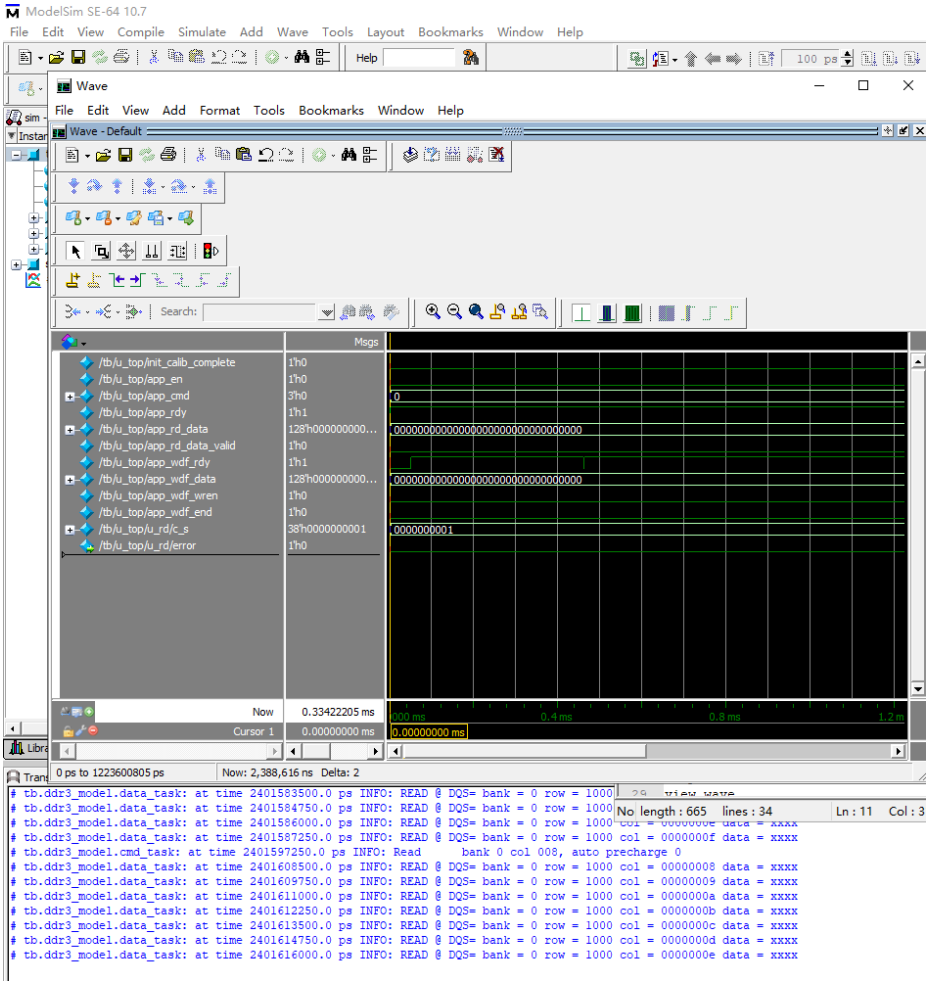


图 32-34 仿真运行示意图

32.3.4.2波形分析

等待仿真运行完成，整个的仿真波形如下图 32-35 所示。

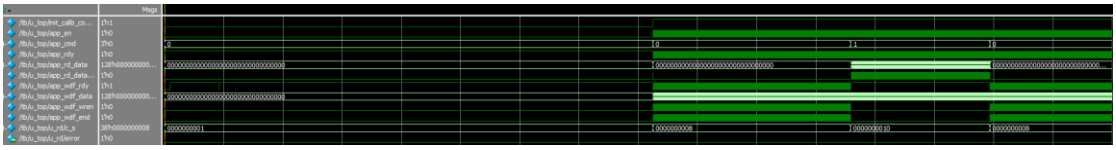


图 32-35 仿真波形图

我们可以看到，当运行了 4.12ms 之后，init_calib_complete 信号被置高，如下图 32-36 所示，说明这个时候对 DDR3 存储器初始化和校准已经完成，之后是往 DDR3 里写入和读出数据，通过比较同一地址写入和读出的数据是否相等验证其功能的正常。

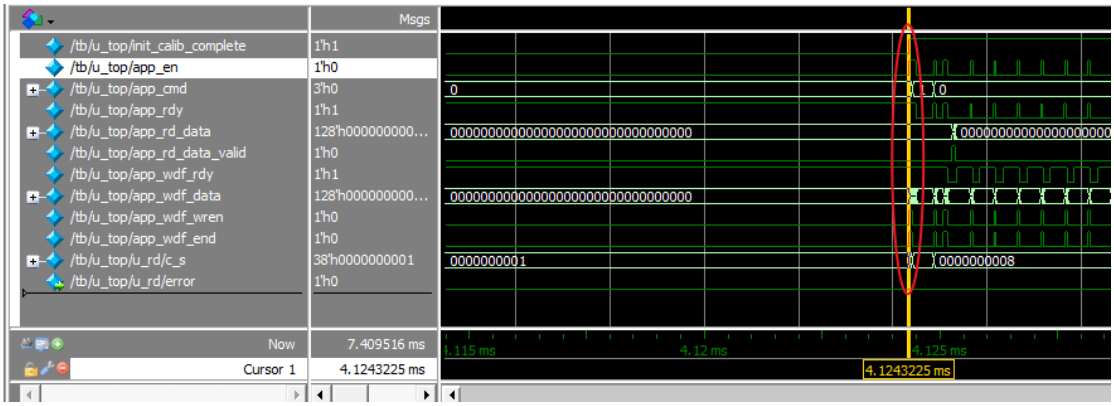


图 32-36 DDR3 初始化信号被拉高

为了能观察到用户侧的命令接口的控制信号波形，将命令接口的波形添加到波形窗口，操作如下图 32-37 所示。

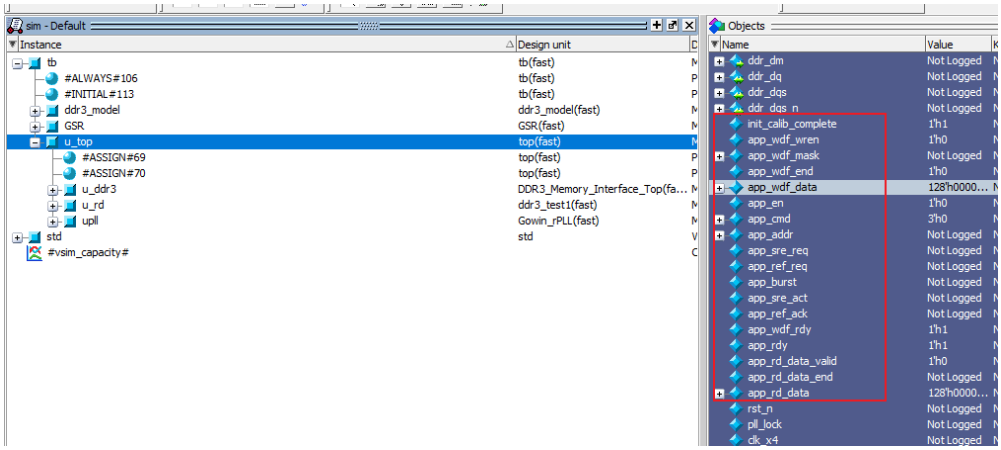


图 32-37 添加用户命令接口

然后点击 Restart 重新开始工程的仿真，操作如下图 32-38 所示，然后等待仿真完成。

图 32-38 重新运行仿真

首先，我们先分析写操作，放大波形，看到波形图如下图 32-39 所示。

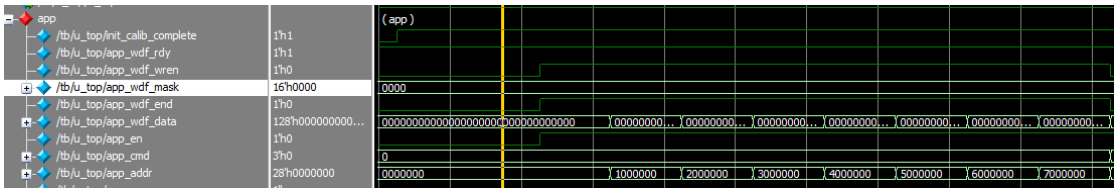


图 32-39 写操作波形图

从上图可以看出，读写操作必须在 DDR 初始化完成之后进行操作，当 app_wdf_rdy 信号为高电平时，拉高 app_en 信号，并将 app_cmd 设置为 0，说明此时为写操作，同时使能 app_wdf_wren 信号，给出地址 app_addr，并向该地址中写入对应的数据，数据和地址是对齐的，上图所示的就是依次向地址 28'h0000000 写入 128'h00000000000000000000000000000000、28'h1000000 写入 128'h00000000000010000001000000100000、28'h2000000 写入 128'h00000000000020000002000000200000、28'h3000000 写入 128'h00000000000030000003000000300000.....。

随后我们便可以看到立即切换了命令，进行读操作，读操作波形图如下图 32-40 所示。

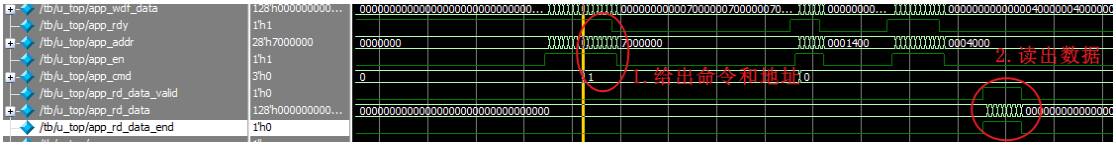


图 32-40 读操作示意图

从上图中可以看出，在进行读操作的时候，首先给出命令和地址之后，数据不会马上读出，间隔一段时间之后，数据才会被读出，首先放大读命令和地址处的波形图，如下图 32-41 所示。

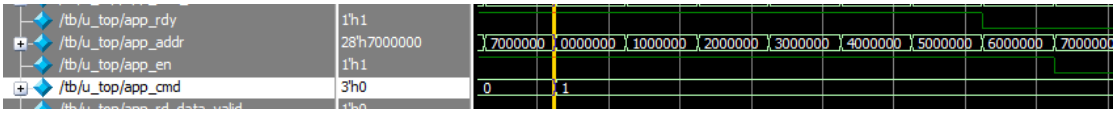


图 32-41 给出读地址和读命令

从上述图中可以看出，app_en 拉高，app_cmd 命令给 1，给出读操作指令，并同时给出地址，可以可能到给出的地址为 28'h0000000~28'h7000000，但是给出 28'h7000000 地址的时候，app_en 信号被拉低，但是我们往后看，可以看到在下一次写之前，给出了最后一个地址 28'h7000000，如下图 32-42 所示，所以最终需要读出 28'h0000000~28'h7000000 地址存放的数据。

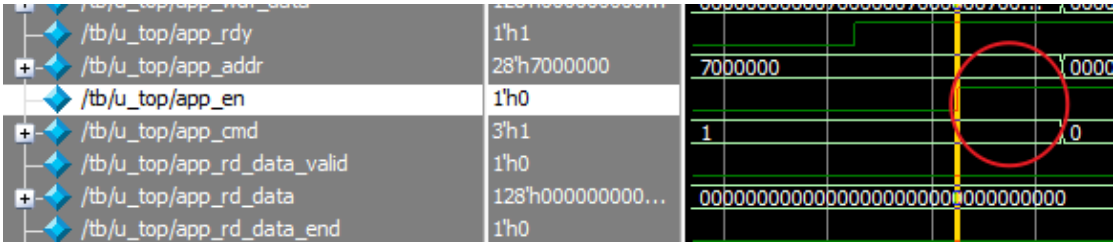


图 32-42 给出最后一个地址命令

最终读出的数据如下图 32-43 所示，这里由于放大看不清，读者可以自行仿

真放大观察，最终可以看到读出的数据依次是 128'h00000000000000000000000000000000、128'h00000000001000000100000010000000、128'h00000000002000000020000000200000...，与我们写入的一致，说明对 DDR3 读写操作成功。

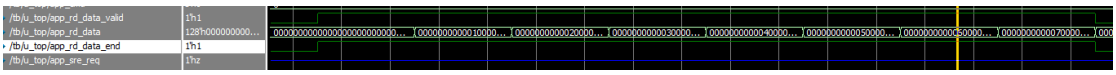


图 32-43 读出的数据波形图

我们上面只对波形的前面读写操作进行了分析，可以看到前面进行读写操作之后，后面将会持续更长时间的读写操作，如下图 32-44 所示。

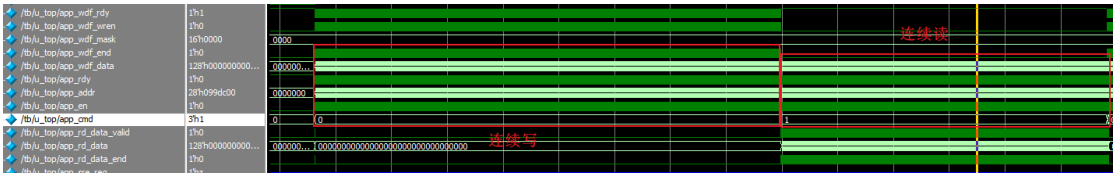


图 32-44 连续读写操作

上图所示的读写操作，涉及数据量比较大，我们这里将不再一个地址一个地址的进行对比分析，读者可以自行查看对应的波形细节，经过分析，最终我们可以看到读出和写入 DDR 中的数据一致。

通过上述仿真，应该让我们对如何通过命令控制 DDR 读写操作有了进一步的了解，方便后续我们自己通过命令去控制读写操作。

32.4思考与总结

本章实验主要讲解了高云 DDR3 IP 的配置以及对用户接口信号进行了说明，并结合官方给出的例程，进行了仿真分析，让我们对 DDR3 的读写操作有了部分的了解，在之后的章节中我们将以本章为基础，对 DDR3 的读写操作进行进一步的了解，并开发一些关于 DDR3 的相关例程。

33 二端口 DDR3 控制器（ddr3_ctrl_2port）设计

工程源码	|----02_设计实例 |----ch33_ddr3_ctrl_2port
相关视频课程	
说明	

章节导读

本节将主要介绍 DDR3 二端口控制器的一种设计方法。通过完成本次 DDR3 二端口控制器设计，读者能够简化大容量缓存的操作控制，从而实现仅需犹如 FIFO 般简便的操作，就可以享受 DDR 的巨大存储容量的这样一个现实应用。

33.1DDR 存储器的应用局限

在前面的内容中，我们对高云 DDR3 IP 的配置以及时序都进行的了说明。虽然借助 IP 核可以成功的实现 FPGA 和 DDR3 存储器之间的数据通信，但是该 IP 核在使用上因有较大的局限而在实际应用上并不方便。

我们知道，数据的存储核读出需满足如下基本原则：对于一个存储器来说，在进行写操作是，如果存储器已满或将满时，则不应再向其中写入数据，即使写入数据也是无效的。如果在进行读操作时，存储器已空或将空，则不应再从其中读出数据，否则读出的数据也是无效的。

在某些典型条件下，针对上述原则，我们如果不加额外的处理手段而单单只使用 DDR3 控制器 IP 核，则数据的读写有效性就有可能无法得到保证，具体如下：

首先，虽然 DDR3 控制器提供给用户的 clk_out 时钟频率是固定的，用户只需要按照该固定频率写入和读出数据即可，但是大部分硬件也会有自身的固定工作频率和数据读写频率，甚至有的器件固定工作频率、数据读写频率和 DDR3 的 clk_out 时钟频率相差极大。存储器和外设硬件的读写频率差异，很有可能将导致 DDR3 的 clk_out 时钟无法满足写入侧或读出侧的硬件读写速率需求，从而读写两端速率不匹配。如果读写速率不匹配，则不但 DDR3 与外设的数据交互会存在跨时钟域的问题，还会导致 DDR3 读写出错。

其次，即使 DDR3 的读写速率刚好也可以满足硬件外设的读写速率，还会存在数据读写连续的问题。在有的外设发送或接收有效数据并不是连续的前提下，我们如果设置 DDR3 控制器 IP 核时，默认外设每一拍发送的数据都是有效

数据，而不针对数据读写不连续现象作其他收发控制，则也会导致数据读写出错。

此外，为了保证数据的交互速率高效，DDR3 IP 核典型的应用数据交互位宽为 128 位，而我们的外设又多以 8 位核 16 位等低位宽读写居多，这样，位宽的不匹配也会成为数据读写核缓存的一大障碍。

凡此种种，直接使用 DDR3 控制器，会存在的问题可以总结如下图 33-1 所示。

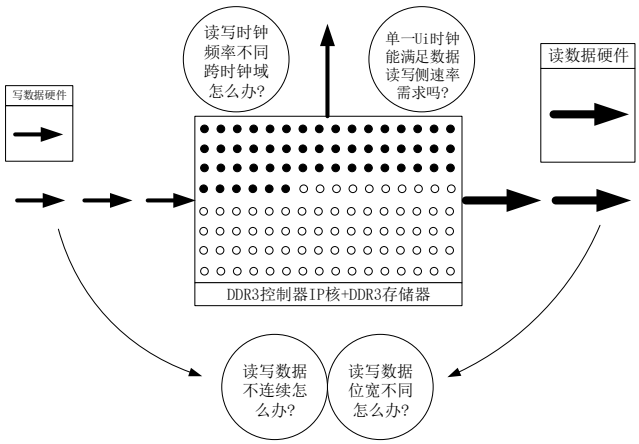


图 33-1 直接使用 DDR 控制器 IP 核会遇到实际问题

值得庆幸的是，以上种种 DDR3 控制器应用上的障碍，都能够通过在读写端各添加一个 FIFO 来获得较好解决。如果读写端各添加一个 FIFO 对写入和读出数据进行缓存，既可以解决跨时钟域的读写问题，又可以适配多种多样的外设读写速率需求，同时，还可以满足数据在固定频率下非连续写和非连续读、位宽不匹配等种种问题。

因此，我们有必要借助前面讲解的内容，对 DDR3 控制器 IP 和进行深入设计。

33.2模块整体结构框图

经过对交互数据的端口分析，可以绘制出 DDR3 二端口控制器设计框图。

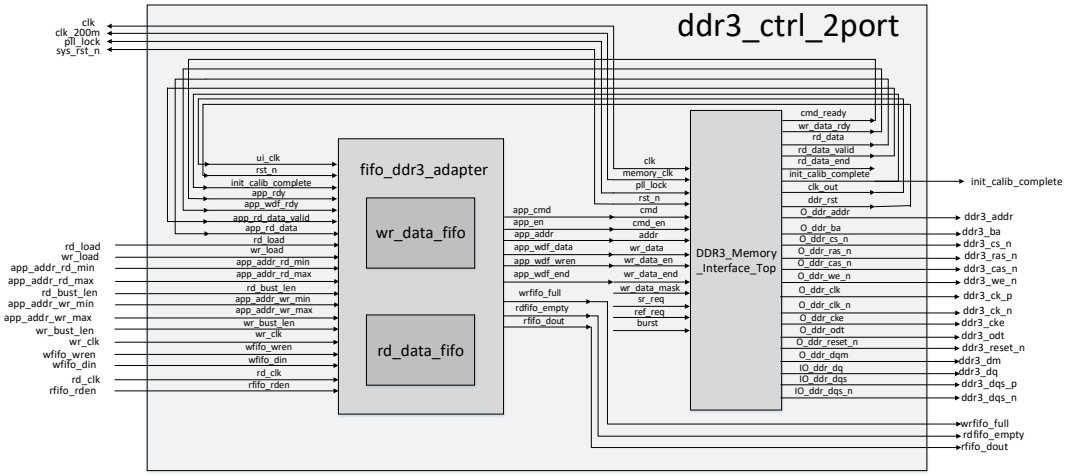


图 33-2 DDR3 二端口控制器模块组成框图

该控制器由如下几个部分构成：

1. DDR3 读写控制的 IP 核（DDR3_Memory_Interface_Top）。
2. 读写 FIFO 和 DDR3 IP 核数据交互控制模块（fifo_ddr3_adapter）。
3. 写 FIFO 模块（wr_ddr3_fifo）：负责将上游外设传递而来的数据写入 DDR3 控制器。
4. 读 FIFO 模块（rd_ddr3_fifo）：负责将 DDR3 控制器内部数据读出到下游。
5. 将模块组进行整合的顶层模块（ddr3_ctrl_2port）。

从以上模块组成框图中我们不难发现，该模块组设计完成后，实际应用时，只需要在工程中合理描述写 FIFO 的写端口核读 FIFO 的读端口对应关系，就可以有效完成 DDR3 的读写控制。至于 DDR3 控制器和 DDR3 器件的数据交互、FIFO 和 DDR3 控制器 IP 核的数据交互，我们则不必过多关心。

33.3模块端口描述

按照上面的模块组设计方案，我们实际再对 DDR3 控制器进行读写控制时，只需要重点对写 FIFO 的写侧端口和读 FIFO 的读侧端口进行输入输出信号对应即可。

表 33-1 ddr3_ctrl_2port 模块端口说明

端口名	方向	描述
时钟、复位、初始化等接口		
clk	I	DDR3 参考输入时钟，50M

pll_lock	I	如果 memory_clk 为 PLL 倍频输入，此接口接 PLL 的 pll_lock 管脚，如果用户不使用 PLL，此接口接高电平
clk_200m	I	提供给 DDR3 控制器的基本工作时钟，要求 200MHz
sys_rst_n	I	外部复位信号
init_calib_complete	O	DDR3 初始化完成标志信号
用户接口		
rd_load	I	输出源更新信号
wr_load	I	输入源更新信号
app_addr_rd_min	I	读 DDR3 的起始地址
app_addr_rd_max	I	读 DDR3 的结束地址
rd_bust_len	I	从 DDR3 中读数据时的突发长度
app_addr_wr_min	I	写 DDR3 的起始地址
app_addr_wr_max	I	写 DDR3 的结束地址
wr_bust_len	I	向 DDR3 中写数据时的突发长度
wr_clk	I	写 FIFO 的写时钟信号
wfifo_wren	I	写 FIFO 的写使能信号，给高电平表示往 FIFO 中写入数据
wfifo_din	I	写入到写 FIFO 中的数据信号
wrfifo_full	O	写 FIFO 的写满标识信号，用于标识当前 FIFO 是否有被写满
rd_clk	I	读 FIFO 的读操作工作时钟
rfifo_rden	I	读 FIFO 的读数据使能控制信号，给高电平表示从 FIFO 中读出数据
rdfifo_empty	O	读 FIFO 的读空标识信号，用于标识当前 FIFO 是否为空（即 FIFO 内有无数据）
rfifo_dout	O	读 FIFO 的读数据输出
接 DDR3 管脚接口（仿真中与 ddr3_model 对接）		
ddr3_dq	IO	数据输入、输出：双向数据总线
ddr3_dqs_n ddr3_dqs_p	IO	差分信号对，DDR3 SDRAM 数据选通信号
ddr3_addr	O	Row 地址（激活命令）、Column 地址（读、写命令）
ddr3_ba	O	Bank 地址
ddr3_ras_n	O	Row 地址选通信号
ddr3_cas_n	O	Column 地址选通信号
ddr3_we_n	O	Row 写使能
ddr3_reset_n	O	DDR3 SDRAM 复位信号
ddr3_ck_p ddr3_ck_n	O	差分信号对，提供给 DDR3 SDRAM 的时钟信号
ddr3_cke	O	DDR3 SDRAM 时钟使能信号
ddr3_cs_n	O	片选，低有效
ddr3_dm	O	DDR3 SDRAM 数据屏蔽信号
ddr3_odt	O	内存信号端接电阻控制

下方的代码给出的模块例化模板。至于参数的设置和端口信号的连接方法，可查看上述端口列表描述。

```
ddr3_ctrl_2port ddr3_ctrl_2port(
    .clk(clk)                   ,       //50M 时钟信号
```

```
    .pll_lock(pll_lock)            ,
    .clk_200m(ddr3_clk200m)          ,      //DDR3 参考时钟信号
    .sys_rst_n(ddr3_rst_n)          ,      //外部复位信号
    .init_calib_complete(ddr3_init_done) ,    //DDR 初始化完成信号

    //用户接口
    .rd_load(rdfifo_clr)            ,   //输出源更新信号
    .wr_load(wrfifo_clr)            ,   //输入源更新信号
    .app_addr_rd_min(28'd0)     ,   //读 DDR3 的起始地址
    .app_addr_rd_max(app_addr_max)     ,   //读 DDR3 的结束地址
    .rd_bust_len(burst_len)         ,   //从 DDR3 中读数据时的突发长度
    .app_addr_wr_min(28'd0)     ,   //写 DD3 的起始地址
    .app_addr_wr_max(app_addr_max)     ,   //写 DDR 的结束地址
    .wr_bust_len(burst_len)         ,   //向 DDR3 中写数据时的突发长度

    .wr_clk(wrfifo_clk)             ,//wr_fifo 的写时钟信号
    .wfifo_wren(wrfifo_wren)          , //wr_fifo 的写使能信号
    .wfifo_din(wrfifo_din)           , //写入到 wr_fifo 中的数据
    .wrfifo_full(),
    .rd_clk(rdfifo_clk)             , //rd_fifo 的读时钟信号
    .rfifo_rden(rdfifo_rden)          , //rd_fifo 的读使能信号
    .rdfifo_empty(),
    .rfifo_dout(rdfifo_dout)          , //rd_fifo 读出的数据信号

    //DDR3
    .ddr3_dq(ddr_dq)              ,   //DDR3 数据
    .ddr3_dqs_n(ddr_dqs_n)          ,   //DDR3 dqs 负
    .ddr3_dqs_p(ddr_dqs)          ,   //DDR3 dqs 正
    .ddr3_addr(ddr_addr)          ,   //DDR3 地址
    .ddr3_ba(ddr_bank)            ,   //DDR3 banck 选择
    .ddr3_ras_n(ddr_ras)          ,   //DDR3 行选择
    .ddr3_cas_n(ddr_cas)          ,   //DDR3 列选择
    .ddr3_we_n(ddr_we)            ,   //DDR3 读写选择
    .ddr3_reset_n(ddr_reset_n)        ,   //DDR3 复位
    .ddr3_ck_p(ddr_ck)          ,   //DDR3 时钟正
    .ddr3_ck_n(ddr_ck_n)          ,   //DDR3 时钟负
    .ddr3_cke(ddr_cke)            ,   //DDR3 时钟使能
    .ddr3_cs_n(ddr_cs)            ,   //DDR3 片选
    .ddr3_dm(ddr_dm)              ,   //DDR3_dm
    .ddr3_odt(ddr_odt)                //DDR3_odt
);
```

33.4模块使用说明

从模块的整体结构框图可以看出，里面包含 4 各子模块，其中 rd_data_fifo 和 wr_data_fifo 为双时钟 FIFO IP，DDR3_Memory_Interface_Top 是 DDR3 控制

器 IP，fifo_ddr3_adapter 模块是我们设计接口转换模块。

33.4.1DDR3_Memory_Interface_Top 模块

该 IP 的详细配置参略前面 “DDR 控制器的使用”一节的内容。

33.4.2wr_data_fifo

wr_data_fifo 是双时钟 FIFO IP，其具体配置如下图 33-3 所示。

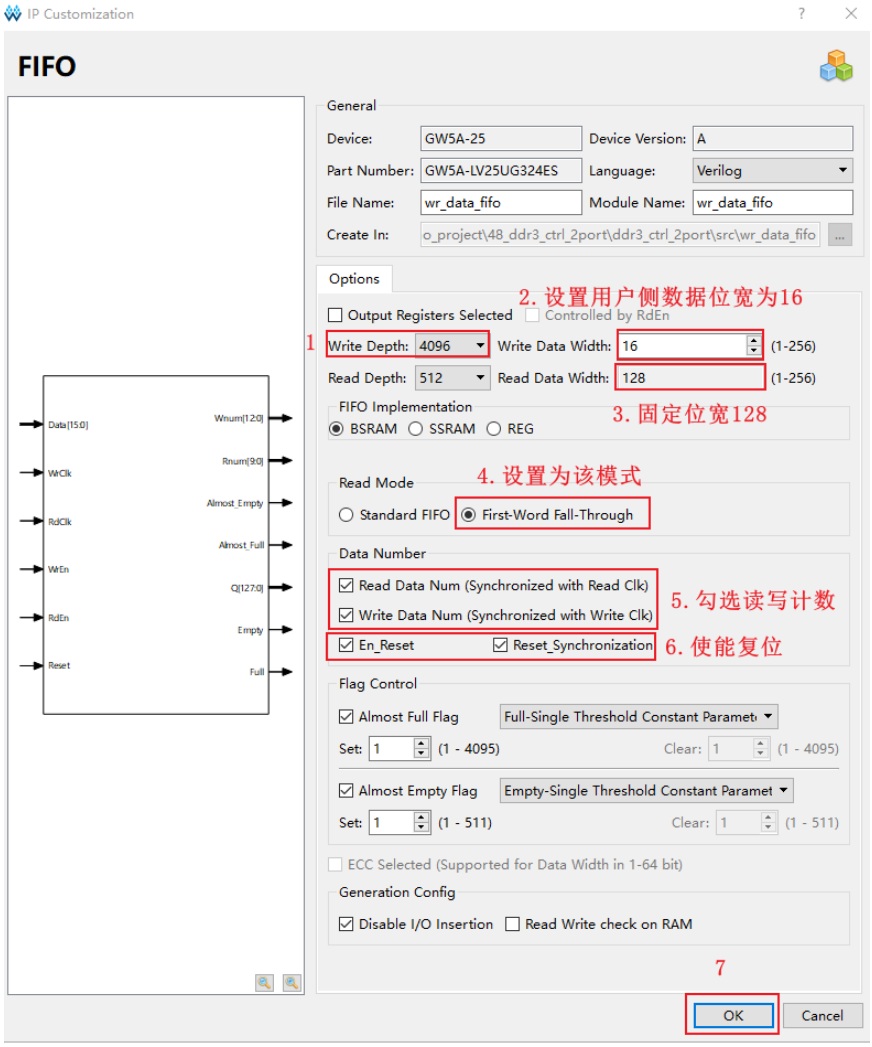


图 33-3 写 FIFO 配置

33.4.3rd_data_fifo

rd_data_fifo 也是双时钟 FIFO IP，具体配置如下图 33-4 所示。

图 33-4 读 FIFO 配置

这样，我们的读写 FIFO，就配置完成了。

33.4.4fifo_ddr3_adapter 模块

fifo_ddr3_adapter 模块内部除去例化了上述两个 FIFO 模块以外，主要就是实现 FIFO 和 DDR3 之间的数据交互。

我们来梳理一下整个数据的流向，用户首先将数据写入到 wr_data_fifo 中，然后将 wr_data_fifo 中的数据读出放至 DDR3 中，最后将 DDR3 中的数据读出放至 rd_data_fifo 中，由用户去读取。wr_data_fifo 的读时钟由用户给，写时钟是 DDR 输出的用户时钟 clk_out，rd_data_fifo 的写时钟为 clk_out，读时钟交由用户去给，整个的数据流向如下图 33-5 所示。

图 33-5 fifo_ddr3_adapter 模块的数据流向

接下来我们对 fifo_ddr3_adapter 模块的代码设计进行说明。

33.4.4.1读写 FIFO 代码设计

1. 写 FIFO 代码设计

写 FIFO 的输入数据信号 Data、写时钟 WrClk、写使能信号 wfifo_wren 都由外部用户侧控制，在该模块中我们不需要做任何处理，只需要将其引出即可，并把写 FIFO 的写满信号引出，方便用户端判断是否能向 FIFO 中写入数据，当 FIFO 被写满之后，应该停止向 FIFO 中数据，否则，则继续向 FIFO 中写入数据。

写 FIFO 的复位控制，一个是由外部的复位信号 rst_n 信号控制，还有一个就是根据输入源更新信号 wr_load 进行控制，将 wr_load 信号移位寄存后，产生一段复位电平，充分复位 FIFO。写 FIFO 的读使能信号由 DDR 控制器的用户端的写数据使能信号 app_wdf_wren 控制，读出的数据信号给到 DDR 控制器的写数据信号 app_wdf_data。写 FIFO 的实现代码如下所示：

```
input           ui_clk              ,    //DDR 用户时钟信号
input           rst_n               ,    //外部按键复位信号
input   [15:0]  wfifo_din           , //写入到 wr_fifo 中的数据
input           wr_load             ,   //输入源更新信号
input           wr_clk              ,//wr_fifo 的写时钟信号
input           wfifo_wren          , //wr_fifo 的写使能信号
output          wrfifo_full         , //写 FIFO 为满信号
output          app_wdf_wren        ,   //DDR3 用户写使能信号
output  [127:0] app_wdf_data        ,   //写入进 DDR 的数据

reg          wfifo_rst_h       ;  //wfifo 复位信号，高有效
wire [9:0]  wfifo_rcount;
wire [127:0] wfifo_dout        ; //从 wr_fifo 中读出的数据，需要写入进
DDR 中
reg  [15:0]  wr_load_d         ;  //由输入源场信号移位拼接得到
reg          wrfifo_load_d0    ;
wire         wfifo_rden        ;

assign wfifo_rden =  app_wdf_wren;
assign app_wdf_data = wfifo_dout;

//对输入源场信号进行移位寄存
always @(posedge wr_clk or negedge rst_n) begin
    if(!rst_n)begin
        wrfifo_load_d0 <= 1'b0;
        wr_load_d  <= 16'b0;
    end
```

```
    else begin
        wrfifo_load_d0 <= wr_load;
        wr_load_d <= {wr_load_d[14:0],wrfifo_load_d0};
    end
end

//产生一段复位电平，满足 fifo 复位时序
always @(posedge wr_clk or negedge rst_n) begin
    if(!rst_n)
        wfifo_rst_h <= 1'b0;
    else if(wr_load_d[0] && !wr_load_d[15])
        wfifo_rst_h <= 1'b1;
    else
        wfifo_rst_h <= 1'b0;
end

wr_data_fifo wr_data_fifo(
    .Data(wfifo_din), //input [15:0] Data
    .Reset(~rst_n|wfifo_rst_h), //input Reset
    .WrClk(wr_clk), //input WrClk
    .RdClk(ui_clk), //input RdClk
    .WrEn(wfifo_wren), //input WrEn
    .RdEn(wfifo_rden), //input RdEn
    .Wnum(), //output [12:0] Wnum
    .Rnum(wfifo_rcount), //output [9:0] Rnum
    .Almost_Empty(), //output Almost_Empty
    .Almost_Full(), //output Almost_Full
    .Q(wfifo_dout), //output [127:0] Q
    .Empty(), //output Empty
    .Full(wrfifo_full) //output Full
);
```

2. 读 FIFO 代码设计

读 FIFO 的写使能信号是 DRR3 控制器输出的读出数据有效信号 app_rd_data_valid，写数据信号是从 DDR3 控制器读取出来的数据 app_rd_data，写时钟信号是 DDR3 控制器输出的用户时钟信号 ui_clk。写 FIFO 的复位控制，一个是由外部的复位信号 rst_n 信号控制，还有一个就是根据输入源更新信号 rd_load 进行控制，将 rd_load 信号移位寄存后，产生一段复位电平，充分复位 FIFO。

读 FIFO 的读使能信号、读时钟信号、读出的数据信号我们都不需要在 fifo_ddr3_adapter模块内实现，只需要将其引出，交由用户端口去使用，读FIFO 的 rdfifo_empty 信号也需引出，引出该信号之后，方便在读取数据的时候进行判

断，只有当读 FIFO 中有数据时，也就是 rdfifo_empty 信号为低时，才有数据需要我们去读取，读 FIFO 的实现代码如下所示：

```
input           app_rd_data_valid   ,   //DDR 读数据有效信号
input   [127:0] app_rd_data         ,
input           rd_clk              , //rd_fifo 的读时钟信号
input           rfifo_rden          , //rd_fifo 的读使能信号
output          rdfifo_empty        , //rd_fifo 的为空信号
output  [15:0]  rfifo_dout           //rd_fifo 读出的数据信号

reg  [15:0]  rd_load_d         ;  //由输出源场信号移位拼接得到
wire [127:0] rfifo_din         ; //写入 rd_fifo 中的数据
reg          rdfifo_rst_h      ;  //rfifo 复位信号，高有效
wire [9:0]   rfifo_wcount;

assign rfifo_wren =  app_rd_data_valid;
assign rfifo_din = app_rd_data;

//对输出源场信号进行移位寄存
always @(posedge ui_clk or negedge rst_n) begin
    if(!rst_n) begin
        rd_load_d <= 1'b0;
        rdfifo_load_d0 <= 1'b0;
    end
    else begin
        rd_load_d <= {rd_load_d[14:0],rdfifo_load_d0};
        rdfifo_load_d0 <= rd_load;
    end
end

//产生一段复位电平，满足 fifo 复位时序
always @(posedge ui_clk or negedge rst_n) begin
    if(!rst_n)
        rdfifo_rst_h <= 1'b0;
    else if(rd_load_d[0] && !rd_load_d[14])
        rdfifo_rst_h <= 1'b1;
    else
        rdfifo_rst_h <= 1'b0;
end

rd_data_fifo rd_data_fifo(
    .Data(rfifo_din), //input [127:0] Data
    .Reset(~rst_n|rdfifo_rst_h), //input Reset
    .WrClk(ui_clk), //input WrClk
    .RdClk(rd_clk), //input RdClk
    .WrEn(rfifo_wren), //input WrEn
    .RdEn(rfifo_rden), //input RdEn
```

```
    .Wnum(rfifo_wcount), //output [9:0] Wnum
    .Rnum(), //output [12:0] Rnum
    .Almost_Empty(), //output Almost_Empty
    .Almost_Full(), //output Almost_Full
    .Q(rfifo_dout), //output [15:0] Q
    .Empty(rdfifo_empty), //output Empty
    .Full() //output Full
);
```

通过上述描述完成了读写 FIFO 侧的代码设计，接下来就需要完成最重要的 FIFO 和 DDR3 控制器之间数据交互的状态机的设计。

33.4.4.2FIFO 与 DDR3 控制器数据交互代码设计

FIFO 与 DDR3 控制器进行数据交互，我们可以通过一段式状态机去实现，首先上电一开始处于 IDLE（空闲状态），然后 DDR3 初始化完成之后跳转到 DDR3_DONE（DDR3 初始化完成状态），然后根据不同的跳转早读\写状态，最后从读写状态再跳转到 DDR3_DONE 状态，状态转移图如下所示，具体的跳转条件将会在后面的内容中进行说明。

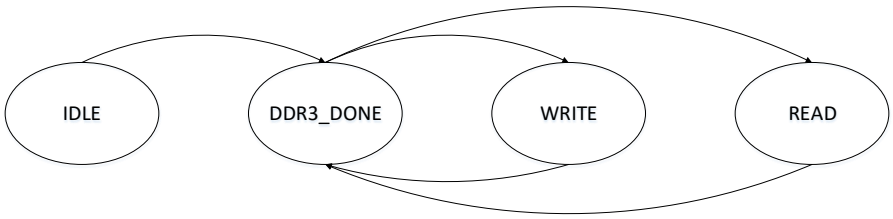


图 33-6 状态转移图

下面将讲解每个状态如何去实现。

1. IDLE（空闲状态）

在前面讲解 DDR3 控制器使用的时候，我们就说过，DDR3 控制器支持读写的最大一个条件就是必须等到其初始化完成之后，所以 IDLE 状态实现代码非常简单，就是等到 DDR3 初始化完成之后，跳转到 DDR3_DONE 状态，实现代码如下所示：

```
IDLE:begin
    if(init_calib_complete)
        state <= DDR3_DONE;
    else
        state <= IDLE;
end
```

2. DDR3_DONE（DDR3 初始化完成状态）

当处于 DDR3_DONE 状态的时候，有如下几种情况需要考虑。

（1）当检测到写入更新标志之后，对地址进行复位计数，并且将写地址设置为写起始地址信号。实现代码如下所示：

```
if(wr_rst) begin    //当检测到写入更新标志之后，对地址进行复位计数
    state <= DDR3_DONE;
    wr_addr_cnt <= 24'd0;
    app_addr_wr <= app_addr_wr_min_a;
end
```

上述代码中的 wr_rst 信号就是检测到的写入更新标志脉冲信号，也就是将 wr_load 信号进行打拍处理，解决跨时钟域问题，然后检测其上升沿信号，就得到了 wr_rst 信号，实现代码如下所示：

```
always @(posedge ui_clk or negedge rst_n)  begin
    if(~rst_n)begin
        wr_load_d0 <= 0;
        wr_load_d1 <= 0;
    end
    else begin
        wr_load_d0 <= wr_load;
        wr_load_d1 <= wr_load_d0;
    end
end

//输入源复位标志信号
always@(posedge ui_clk or negedge rst_n) begin
    if(~rst_n)
        wr_rst <= 0;
    else if(wr_load_d0 && !wr_load_d1)
        wr_rst <= 1;
    else
        wr_rst <=0;
end
```

（2）当读写地址达到设定的最大读写地址之后，将地址信号设置为读写的起始地址并且清除读写地址计数信号，实现代码如下所示：

```
else if(app_addr_rd >= app_addr_rd_max_a - 8) begin//读到读地址结束
    state <= DDR3_DONE;
    rd_addr_cnt <= 24'd0;
    app_addr_rd <= app_addr_rd_min_a;
end
else if(app_addr_wr >= app_addr_wr_max_a - 8) begin //写结束
    state <=DDR3_DONE;
    wr_addr_cnt <= 24'd0;
    app_addr_wr <= app_addr_wr_min_a;
```

```
end
```

（3）FIFO 控制模块优先处理 DDR3 写请求，以免写 FIFO 溢出时，用于写入 DDR3 的数据丢失，当写 FIFO 中的数据量大于写突发长度时，也就是写 FIFO 中的可读数据个数 wfifo_rcount 大于一次突发长度的时候，执行写 DDR3 操作。实现代码如下所示：

```
else if(wfifo_rcount >= wr_bust_len_a) begin
    state <= WRITE; //跳至写操作
    wr_addr_cnt <= 24'd0;
    app_addr_wr <= app_addr_wr;
end
```

（4）当帧复位 raddr_rst_h 信号到来时，将 raddr_rst_h 信号拉高一段时间之后，进入读状态，此时读地址读起始地址，实现代码如下所示：

```
else if(raddr_rst_h) begin
    if(raddr_rst_h_cnt >= 11'd201) begin
        state <= READ;
        rd_addr_cnt <= 24'd0;
        app_addr_rd <= app_addr_rd_min_a;
    end
    else begin
        state <= READ;
        rd_addr_cnt <= 24'd0;
        app_addr_rd <= app_addr_rd;
    end
end
```

上述代码中的 raddr_rst_h 信号就是对输出源更新信号进行寄存，然后检测其上升沿信号，并将 raddr_rst_h 信号拉高，当读地址变成设定的读地址的起始值，将 raddr_rst_h 信号拉低。raddr_rst_h_cnt 信号是对输出源的帧复位脉冲信号进行计数，这样就防止了 FIFO 还没有复位完全就写入数据的情况，实现代码如下所示：

```
//对输入的更新信号进行打拍处理
always @(posedge ui_clk or negedge rst_n)  begin
    if(~rst_n)begin
        rd_load_d0 <= 0;
        rd_load_d1 <= 0;
    end
    else begin
        rd_load_d0 <= rd_load;
        rd_load_d1 <= rd_load_d0;
    end
end
```

```
//对输出源的读地址做个帧复位脉冲
always @(posedge ui_clk or negedge rst_n)  begin
    if(~rst_n)
        raddr_rst_h <= 1'b0;
    else if(rd_load_d0 && !rd_load_d1)
        raddr_rst_h <= 1'b1;
    else if(app_addr_rd == app_addr_rd_min_a)
        raddr_rst_h <= 1'b0;
    else
        raddr_rst_h <= raddr_rst_h;
end

    //对输出源的帧复位脉冲进行计数
always @(posedge ui_clk or negedge rst_n)  begin
    if(~rst_n)
        raddr_rst_h_cnt <= 11'b0;
    else if(raddr_rst_h)
        raddr_rst_h_cnt <= raddr_rst_h_cnt + 1'b1;
    else
        raddr_rst_h_cnt <= 11'b0;
end
```

（5）当读 FIFO 中的数据量小于读突发长度时，执行读 DDR3 操作。代码如下所示：

```
else if(rfifo_wcount <= rd_bust_len_a) begin
    state <= READ;      //跳到读操作
    rd_addr_cnt <= 24'd0;
    app_addr_rd <= app_addr_rd;  //读地址不变
end
```

DDR3_DONE 状态的完整实现代码如下所示：

```
DDR3_DONE:begin
    if(wr_rst) begin   //当检测到写入更新标志之后，对地址进行复位计数
        state <= DDR3_DONE;
        wr_addr_cnt <= 24'd0;
        app_addr_wr <= app_addr_wr_min_a;
    end
    else if(app_addr_rd >= app_addr_rd_max_a - 8) begin//读到读地址结束
        state <= DDR3_DONE;
        rd_addr_cnt <= 24'd0;
        app_addr_rd <= app_addr_rd_min_a;
    end
    else if(app_addr_wr >= app_addr_wr_max_a -8) begin //写结束
        state <=DDR3_DONE;
        wr_addr_cnt <= 24'd0;
        app_addr_wr <= app_addr_wr_min_a;
    end
```

```
    else if(wfifo_rcount >= wr_bust_len_a) begin
        state <= WRITE; //跳至写操作
        wr_addr_cnt <= 24'd0;
        app_addr_wr <= app_addr_wr;
    end
    else if(raddr_rst_h) begin
        if(raddr_rst_h_cnt >= 11'd201) begin
            state <= READ;
            rd_addr_cnt <= 24'd0;
            app_addr_rd <= app_addr_rd_min_a;
        end
        else begin
            state <= READ;
            rd_addr_cnt <= 24'd0;
            app_addr_rd <= app_addr_rd;
        end
    end
    else if(rfifo_wcount <= rd_bust_len_a) begin
        state <= READ;      //跳到读操作
        rd_addr_cnt <= 24'd0;
        app_addr_rd <= app_addr_rd;  //读地址不变
    end
    else begin
        state <= state;
        wr_addr_cnt <= 24'd0;
        rd_addr_cnt <= 24'd0;
    end
end
```

3. WRITE（写状态）

在写状态时，当写地址计数器达到一次写操作的时候并且握手信号使能 app_rdy 和写数据有效使能信号 app_wdf_rdy 信号同时为高的情况下，代表此时我们将 FIFO 中存储的数据写入到 DDR 中，满足写条件的时候，设置写计数进行 wr_addr_cnt 增加 1，并且 DDR3 用户的侧的地址加 8，为什么要加 8 呢？这是因为用户端在每一个用户时钟进行一个 128bit 的数据的传输，在 DDR3 物理芯片需要分 8 此传输，每次传输一个位宽 16bit，8 次就需要 8 个地址，当 wr_addr_cnt 的值等于一次突发长度时，代表一次突发写入完成回到 DDR3_DONE 状态，等待下一次突发传送，实现代码如下所示：

```
WRITE:  begin
    if(wr_addr_cnt == (wr_bust_len_a - 1) && app_rdy && app_wdf_rdy)
begin
        state <= DDR3_DONE;
        app_addr_wr<=app_addr_wr + 8;//DDR 突发长度为 8，一次写入 8 个数据
```

```
    end
    else if((app_rdy) && app_wdf_rdy) begin //写条件满足
        wr_addr_cnt <= wr_addr_cnt + 1'd1;
        app_addr_wr <= app_addr_wr + 8;
    end
    else begin //写条件不满足
        wr_addr_cnt <= wr_addr_cnt;
        app_addr_wr <= app_addr_wr;
    end
end
```

4. READ（读状态）

当 app_rdy 信号为高时，表示此时可以从 DDR 中读出数据，并设置读计数 rd_addr_cnt 加 1，地址加 8，这里加 8 的原因在和写操作地址加 8 的原因一样。当 rd_addr_cnt 等于一次读突发长度时，返回 DDR3_DONE 状态，等待下一次读操作，实现代码如下所示：

```
READ: begin
    if(rd_addr_cnt == (rd_bust_len_a - 1) && app_rdy) begin
        state <= DDR3_DONE;
        app_addr_rd <= app_addr_rd + 8;
    end
    else if(app_rdy) begin
        rd_addr_cnt <= rd_addr_cnt + 1'd1; //用户地址计数器每次加一
        app_addr_rd <= app_addr_rd + 8;
    end
    else begin
        rd_addr_cnt <= rd_addr_cnt;
        app_addr_rd <= app_addr_rd;
    end
end
```

通过上述讲解，完成最核心的状态机的设计。

但是在对 DDR3 控制进行操作的时候，还有关于读写命令的控制，在写状态且写有效，或者在读状态时，将命令使能信号 app_en 拉高，否则 app_en 信号为低，代码如下所示：

```
assign app_en = ((state == WRITE && (app_rdy && app_wdf_rdy))
                    ||(state == READ && app_rdy)) ? 1'b1:1'b0;
```

在写状态其写有效的情况下，拉高 DDR3 控制的写使能信号 app_wdf_wren，否则为低，代码如下所示：

```
assign app_wdf_wren=(state==WRITE&&(app_rdy&&app_wdf_rdy))? 1'b1:1'b0;
```

由于我们 DDR3 芯片时钟和用户时钟分频选择 4：1，突发长度为 8，所以 app_wdf_end 信号 app_wdf_wren 信号一致。

```
assign app_wdf_end = app_wdf_wren;
```

当状态处于读操作的时候，将 app_cmd 设置为 1，否则为 0，代表需要向 DDR3 中写入数据。

```
assign app_cmd = (state == READ) ? 3'd1 :3'd0;
```

最后当状态处于读状态时，将 DDR3 控制器的地址信号 app_addr 为读地址信号 app_addr_rd，否则为写地址信号 app_addr_wr。

```
always @(*)  begin
    if(~rst_n)
        app_addr <= 0;
    else if(state == READ )
        app_addr <= app_addr_rd;
    else
        app_addr <= app_addr_wr;
end
```

讲解至此，fifo_ddr3_adapter 模块的设计就算完成了，完整的代码请自行查看工程源代码。最后在 ddr3_ctrl_2port 模块中，例化 fifo_ddr3_adapter 模块和 DDR3_Memory_Interface_Top 模块，将需要的端口引出，ddr3_ctrl_2port 模块就设计完成，接下来我们将会对设计的 ddr3_ctrl_2port 模块进行仿真分析。

33.5模块仿真分析

33.5.1仿真文件设计

对上述设计的模块设计内容进行分析和综合直至没有错误和警告。新建名称为 ddr3_ctrl_2port_tb 的仿真文件，在仿真文件中我们需要添加 DDR3 仿真模型，用于模拟向 DDR 芯片中写入和读出数据，具体仿真的 testbench 设计的结构框图如下图 33-7 所示。

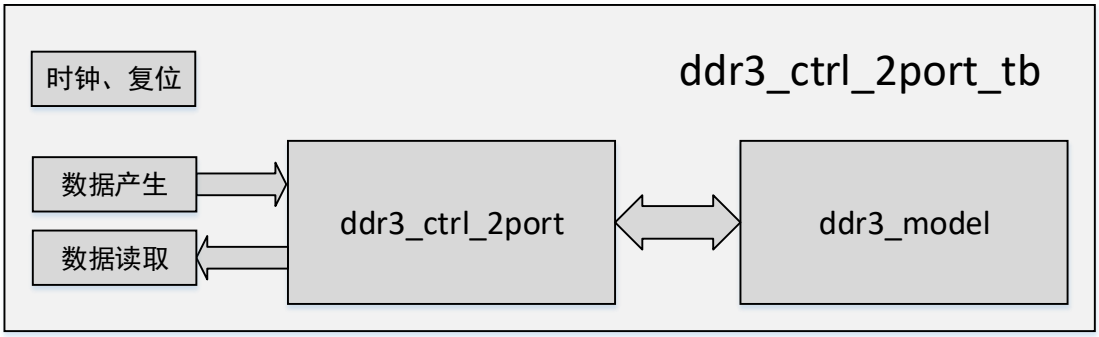


图 33-7ddr3_ctrl_2port_tb 仿真架构

从上述图中可以看出，在仿真设计除去 ddr3_ctrl_2port 和 ddr3_model 这两

个模块以外，还需要产生时钟和复位、数据产生和数据读取的激励。为了简单处理，仿真时钟 DDR 控制器时钟采用产生 200MHz 时钟 ddr3_clk200m，写侧 FIFO 的写时钟和读侧 FIFO 的读时钟均采用 100Mhz 时钟，DDR3 控制器的参考时钟采用 50MHz 的时钟。200MHz 的时钟周期 5ns，即每 2.5ns 翻转一次，100MHz 时钟的时钟周期为 10ns，即每 5ns 翻转一次，50MHz 时钟的时钟周期为 20ns，即每 10ns 翻转一次，时钟产生代码如下。注意这里仿真的时间单位和精度分别使用 1ns 和 100ps，即使用`timescale 1ns/100ps。

```
initial ddr3_clk200m = 1'b1;
always #2.5 ddr3_clk200m = ~ddr3_clk200m;

initial wrfifo_clk = 1'b1;
always #5 wrfifo_clk = ~wrfifo_clk;

initial rdfifo_clk = 1'b1;
always #5 rdfifo_clk = ~rdfifo_clk;

initial clk = 1'b1;
always #10 clk = ~clk;
```

数据的产生的激励设计上将其封装成任务 task 形式，方便调用。具体代码如下，该任务的具体功能是在调用这个任务和给定的输入参数 data_begin 和 wr_data_cnt 时，向 wr_ddr3_fifo 中写入以起始数据 data_begin 开始递增的 wr_data_cnt 个数据。

```
task wr_data;
    input [15:0]data_begin;
    input [15:0]wr_data_cnt;
    begin
    wrfifo_wren = 1'b0;
    wrfifo_din  = data_begin;
    @(posedge wrfifo_clk);
    #1 wrfifo_wren = 1'b1;
    repeat(wr_data_cnt)
    begin
        @(posedge wrfifo_clk);
        wrfifo_din = wrfifo_din + 1'b1;
    end
    #1 wrfifo_wren = 1'b0;
    end
endtask
```

数据读取的激励设计上采用与数据产生类似的方式，封装成任务 task 形式，具体代码如下。任务的具体功能是在调用这个任务和给定的输入参数 rd_data_cnt 时，向 rd_ddr3_fifo 中读出 rd_data_cnt 个数据。

```
task rd_data;
    input [15:0]rd_data_cnt;
    begin
    rdfifo_rden = 1'b0;
    @(posedge rdfifo_clk);
        #1 rdfifo_rden = 1'b1;
        repeat(rd_data_cnt)
        begin
            @(posedge rdfifo_clk);
        end
    #1 rdfifo_rden = 1'b0;
    end
endtask
```

仿真中加入了 DDR 控制器 IP 模块，在写入数据前需要产生 pll_clock 信号并且产生 wr_load 信号，初始化完成之后，通过调用 wr_data 事件写入数据，写入的初始值为 1，写入 1024 个数据，数据写完之后，产生 rd_load 信号，然后调用 rd_data 事件读取 1024 个数据，具体代码如下所示：

```
initial begin
    ddr3_rst_n = 1'b0;
    wr_load  = 1'b1;
    wrfifo_wren = 1'b0;
    wrfifo_din  = 16'd0;
    rd_load  = 1'b1;
    rdfifo_rden = 1'b0;
    pll_lock   = 1'b0;
    #201;
    ddr3_rst_n = 1'b1;
    pll_lock   = 1'b1;
    #200;
    wr_load  = 1'b0;
    rd_load  = 1'b0;
    @(posedge ddr3_init_done);
    #200;
    wr_data(16'd1,16'd1024);
    #2000;
    rd_load  = 1'b1;
    #20;
    rd_load  = 1'b0;
    #20000;
    rd_data(16'd1024);
    #15000;
    $stop;
end
```

最后就是例化 ddr3_ctrl_2port 模块和 ddr3_model 模块，进行端口定义和连

接，这样整体的仿真 tb 文件就设计完成了，完整的仿真模块代码，请自行查看 src 文件夹下的 tb 文件夹下的源文件。

33.5.2仿真波形分析

建立 Modelsim 仿真工程，然后将本次实验需要的文件添加至工程中，建立仿真工程的步骤请读者前面“Gowin 联合 Modelsim 仿真实验”一节内容，然后将 fifo_ddr3_adapter 模块的信号添加进行观察，操作如下所示。

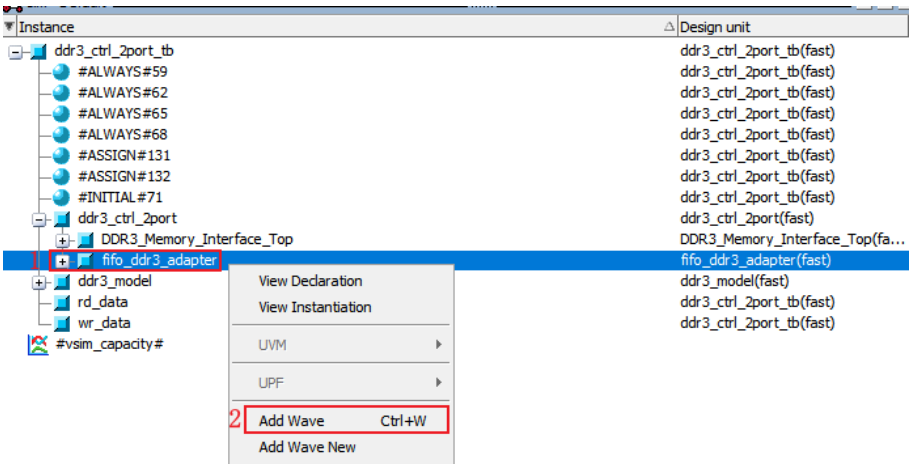


图 33-8 添加 fifo_ddr3_adapter 模块的信号进行观察

然后点击开始运行仿真，仿真时间比较长，需要我们等待大概 3ms 左右，整个仿真就运行完成，整个波形图如下图 33-9 所示。

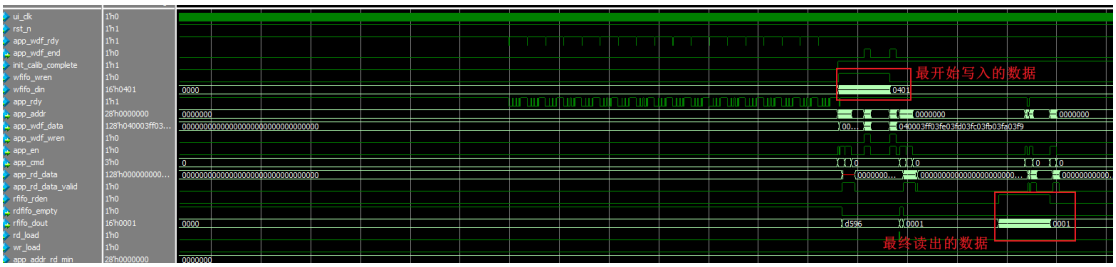


图 33-9 仿真整体效果图

我们对读写的具体波形分析。

首先是读操作，在本次仿真中，我们写入的数据是 1024 个 16 位的数据，一次突发长度 burst_len 为 64，这个突发长度是从 FIFO 中每次读取到 DDR 中的突发长度，1024 个 16 位的数据对应的就是需要向 DDR 控制器中写入 128 个（1024*16/128）个 128 位的数据，每次突发长度为 64，那么我们就需要 2（128/64）次突发，将 1024 个 16 位的数据最终写入到 DDR 中，具体波形如下图 33-10 所示。

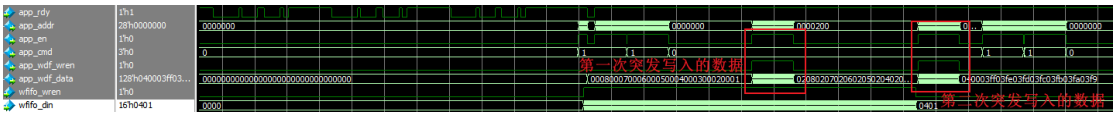


图 33-10 突发写 2 次的仿真体现

对第一次突发写 DDR 操作部分的波形进行放大，其波形图如下图 33-11 所示。

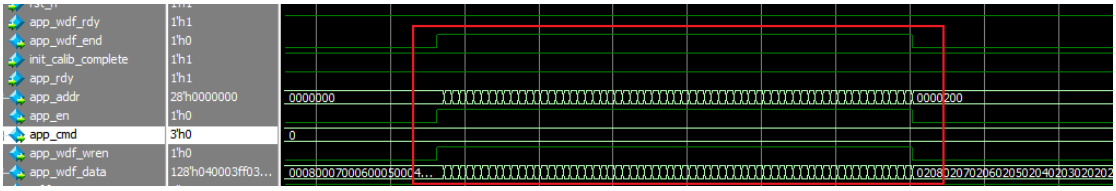


图 33-11 第一次突发写的放大观察

从波形图可以看到一次写数据操作流程如下：

（1）首先所有的读写操作都必须在初始化信号被拉高之后执行。

（2）在 app_wdf_rdy 和 app_rdy 信号同时位高时，方可发送命令进行写操作。

（3）发送的时候，首先需要拉高 app_en 信号，然后给出读命令（app_cmd 为 0），同时给出地址和需要写入的数据。

一次写操作过程中，数据写入流程与我们设计预期是一致的，然后我们对写入的数据波形的开头进行放大，波形图如下图 33-12 所示。写入的每个数据是由写 FIFO 写入 8 个 16bit 数据拼接而成的 128bit 数据，具体拼接过程是由写 FIFO 完成，app_addr 地址每次增加 8，这里加 8 的原因在前面的代码设计中我们已经做过介绍，是因为用户端在每一个用户时钟进行一个 128bit 的数据的传输，在 DDR3 物理芯片需要分 8 此传输，每次传输一个位宽 16bit，8 次就需要 8 个地址，这与我们预期一致。然后放大第一次突发写入的最后的数据，波形图有如下图 33-13 所示，从写入的数据内容可以判断，写入的 16 位的数据从 0x001~0x200，也就是一共写入了 512 个 16 位的数据，也就是 64 个 128 位的数据，与一次突发需要写入的数据个数一致，

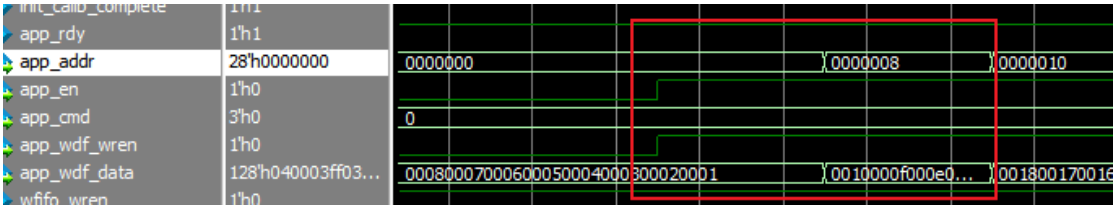


图 33-12 第一次突发写入的最开始数据

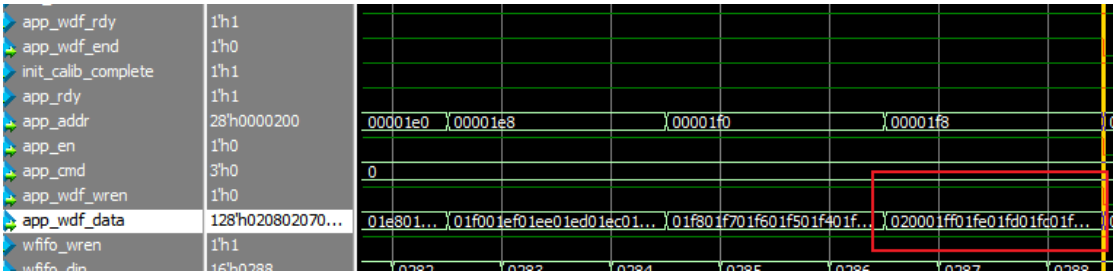


图 33-13 第一次突发写入的最后数据

然后是读操作，读操作有效是在产生 rd_load 信号，之后，波形图如下图 33-14 所示。

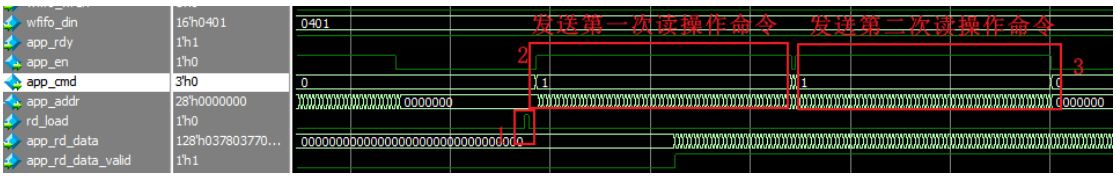


图 33-14 突发读操作波形图

读操作的一般流程如下所示：

（1）在 app_rdy 为高电平的时候，将 app_en 信号拉高，并给出读命令（app_cmd 为 1），同时给出需要的读取的数据对应的地址，这些将会被传输给 DDR 控制器进行处理；

（2）间隔一段时间之后，对应地址存储的数据将会被读出，读出的数据并非都是有效的，只有当 app_rd_data_valid 信号为高电平的时候，代表读出的数据有效，是我们所需要的数据

从整体上看，写 FIFO 中写入的数据是 0x1~0x4000 一共 1024 个连续递增的数据，波形与仿真代码是一致的，下面是整个写入数据的写入第一个数据与最后一个数据的波形图。

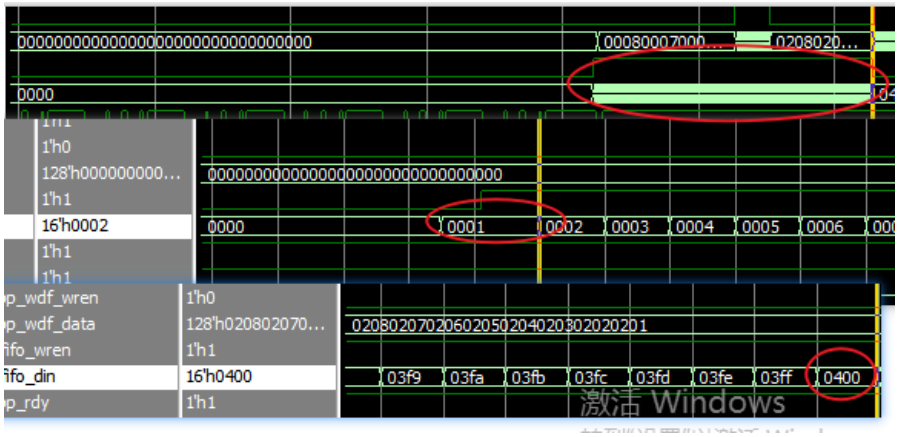


图 33-15 写入数据整体效果与关键信息

读 FIFO 读出的数据同样是 0x1~0x4000 一共 1024 个连续递增的数据。

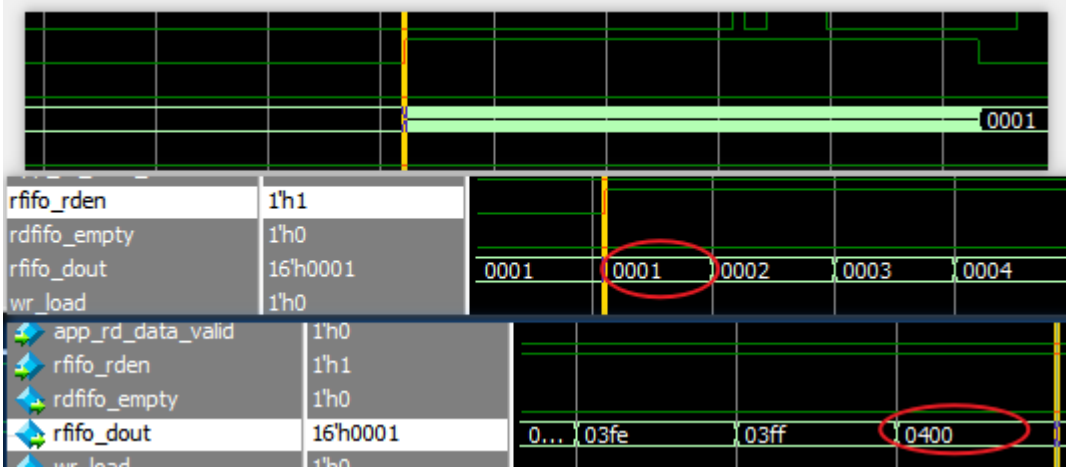


图 33-16 读出数据整体效果与关键信息

从写入数据和最后读出数据的一致性说明 ddr3_ctrl_2port 模块工作正常。

33.6思考与总结

本章实验主要讲解了 ddr3_ctrl_2port 模块的具体设计与仿真分析，ddr3_ctrl_2port 模块将会方便用户去使用 DDR3 控制器存储数据，不仅解决了跨时钟域的问题，还解决了数据位宽的转换以及数据读写连续的问题。通过本节内容让我们对 DDR3 控制器的读写操作有了进一步的理解，在后面的实验中，我们使用 ddr3_ctrl_2port 模块进行一些实际的应用，比如图像处理或者数据采集相关的一些案例。

34 基于 DDR3 的串口传图帧缓存系统设计实现（HDMI 和 TFT 显示）

工程源码	|----02_设计实例 |----ch34_uart_ddr3_tft_hdmi
相关视频课程	
说明	

章节导读

本章将基于前面讲解的设计内容，继续讲解串口传图的基本内容。本章在 FPGA 学习内容中，处于立交桥的地位。本章的内容全面涵盖输入、缓存、输出，同时本章的内容又是基于 FPGA 片上图片缓存的内容深化。学完本章以后，既可以基于本章内容实现各种类型千变万化输入输出接口的替换设计，又可以在本章的数据接收与处理环节作文章，进行图像处理内容的理解与学习。

在学习本章之前，建议读者先复习串口接收模块、disp_driver 模块、HDMI 驱动模块 dvi_encoder 设计、基于 HDMI 和 TFT 显示屏的图片显示相关内容，同时，对二端口 DDR3 控制器模块章节的端口设计进行理解强化。

34.1系统整体设计

通过前面 DDR 相关章节的内容，让我们对 DDR 控制器的 IP 配置及使用有了一定的了解。我们趁热打铁，结合前面章节学习的串口接收模块和 TFT 显示屏控制、HDMI 控制模块，设计一个基于 DDR3 的串口传图帧缓存系统。该系统的整体设计框图如下。

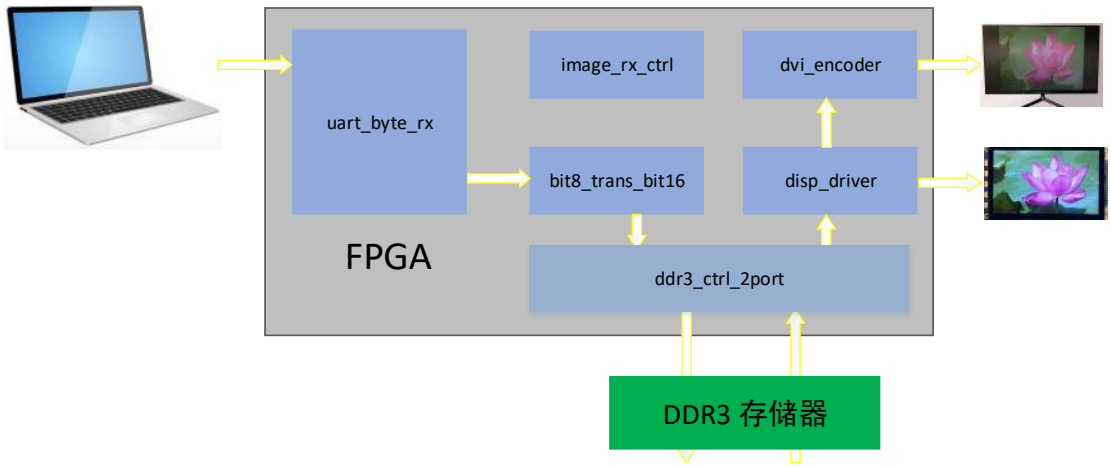


图 34-1 串口顶层系统设计框图

其中，

（1）uart_byte_rx 模块：负责串口图像数据的接收，该模块的设计前面有讲，但是需要注意的是，前面在讲解串口接收时，为了避免出现非常强的电磁干扰，导致采集的信号被干扰而使得结果出错，我们就将每一位数据平均分成 16 小段，取中间比较稳定的部分作为最终的结果输出，但是在串口传图相关实验中，不必考虑这种干扰，直接将每一位接收到的数据输出即可，这样将会减少传输数据的时间，考虑到传输时间，本次设计我们将波特率设置为 2000000bps。

（2）bit8_trans_bit16 模块：将串口接收的每两个 8bit 数据转换成一个 16bit 数据（图像数据是 16bit 的 RGB565 的数据，电脑是通过串口将一个像素点数据分两次发送到 FPGA，FPGA 需将串口接收数据重组成 16bit 的图像数据），实现过程相对比较简单。

（3）disp_driver 模块：tft 屏显示驱动控制，对缓存在 DDR3 中的图像数据进行显示。

（4）ddr3_ctrl_2port 模块组：包含 wr_ddr3_fifo、rd_ddr3_fifo、fifo_ddr3_adapter 以及 DDR3_Memory_Interface_Top 模块。完成采集的图像数据缓存。

（5）pll 模块：上述各个模块所需时钟的产生，使用 PLL IP。

（6）dvi_encoder 模块：用于将生成信号转换成 HDMI 输出信号作 HDMI 显示。

除去使用 IP 和前面章节讲过的模块（组）外，还需要设计 bit8_trans_bit16 模块和 image_rx_ctrl 模块。

34.2接收控制模块设计（image_rx_ctrl）

为了对接收的信号进行分析和整理，我们设计了接收控制模块，主要内容是描述图像的行场同步信号生成。

```
module image_rx_ctrl # (
  parameter DISP_WIDTH  = 800 ,
  parameter DISP_HEIGHT = 480
)
(
  input             clk               ,
  input             reset_p           ,
```

```
  input             image_data_valid   ,
  output reg [15:0] image_data_hcnt    ,
  output reg [15:0] image_data_vcnt    ,
  output reg        image_data_hs      ,
  output reg        image_data_vs      ,
  output reg        frame_rx_done_flip
);

//generate image data hs or vs
always@(posedge clk or posedge reset_p)
    if(reset_p)
    image_data_hcnt <= 'd0;
    else if(image_data_valid) begin
    if(image_data_hcnt == (DISP_WIDTH - 1'b1))
        image_data_hcnt <= 'd0;
    else
        image_data_hcnt <= image_data_hcnt + 1'b1;
    end

always@(posedge clk or posedge reset_p)
    if(reset_p)
    image_data_vcnt <= 'd0;
    else if(image_data_valid) begin
    if(image_data_hcnt == (DISP_WIDTH - 1'b1)) begin
        if(image_data_vcnt == (DISP_HEIGHT - 1'b1))
        image_data_vcnt <= 'd0;
        else
        image_data_vcnt <= image_data_vcnt + 1'b1;
    end
    end
//hs
always@(posedge clk or posedge reset_p)
    if(reset_p)
    image_data_hs <= 1'b0;
    else if(image_data_valid&&image_data_hcnt == (DISP_WIDTH - 1'b1))
    image_data_hs <= 1'b0;
    else
    image_data_hs <= 1'b1;
//vs
always@(posedge clk or posedge reset_p)
    if(reset_p)
    image_data_vs <= 1'b0;
    else if(image_data_valid&&image_data_hcnt==(DISP_WIDTH - 1'b1) &&
            image_data_vcnt == (DISP_HEIGHT - 1'b1))
    image_data_vs <= 1'b0;
    else
    image_data_vs <= 1'b1;
```

```
always@(posedge clk or posedge reset_p)
    if(reset_p)
    frame_rx_done_flip <= 1'b0;
    else if(image_data_valid&&image_data_hcnt==(DISP_WIDTH - 1'b1) &&
            image_data_vcnt == (DISP_HEIGHT - 1'b1))
    frame_rx_done_flip <= ~frame_rx_done_flip;

endmodule
```

在该模块之中，完成了图像行同步信号描述及计数，图像场同步信号描述及计数，图像接收完成信号描述，通过这些信息，就可以定位图像传输的进度。同时，如果图像传输完成，则给出传输完成的 LED 点亮信号。

34.3位宽转换模块设计（bit8_trans_bit16）

bit8_trans_bit16 该模块主要功能是将串口传输过来（一次传 8bit）的每两个 8bit 数据拼接成一个 16bit 图像数据（RGB565）。功能相对比较简单。该模块的接口图如下图 34-2 所示，接口功能描述如下表 34-1 所示。

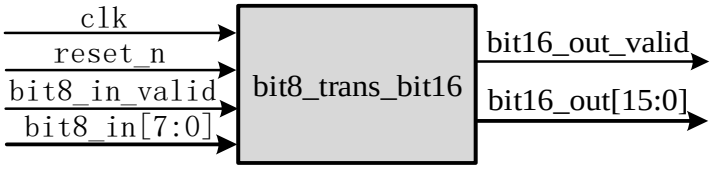


图 34-2 bit8_trans_bit16 模块接口图

表 34-1 bit8_trans_bit16 模块接口功能描述表

接口名称	I/O	功能描述
clk	I	模块工作时钟
reset_n	I	模块复位，低有效
bit8_in[7:0]	I	8bit 数据输入
bit8_in_valid	I	8bit 数据有效标识
bit16_out[15:0]	O	转换后 16bit 数据输输出
bit16_out_valid	O	转换后 16bit 数据有效标识

模块主要信号设计的时序要求如下。对输入的每两个 8bit 数据进行拼接，并产生一个拼接后数据输出有效标识信号。拼接后的数据是前一个数据在高字节位置，后一个数据在低字节位置。这个主要是串口传图上位机软件先发送 16bit 图像数据的高字节，后发低字节，FPGA 上这样处理为了保证拼接后图像数据的正确性。

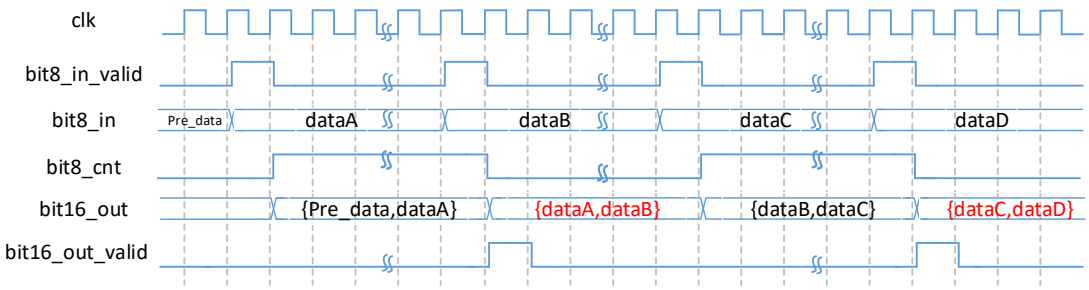


图 34-3 bit8_trans_bit16 数据转换时序图

有了时序图，代码设计就比较简单，具体代码如下。

```
module bit8_trans_bit16(
  input             clk,
  input             reset_p,

  input      [7:0] bit8_in,
  input            bit8_in_valid,

  output reg [15:0]bit16_out,
  output reg       bit16_out_valid
);

reg bit8_cnt;

always@(posedge clk or posedge reset_p)
begin
if(reset_p)
    bit8_cnt <= 1'b0;
else if(bit8_in_valid)
    bit8_cnt <= bit8_cnt + 1'b1;
else
    bit8_cnt <= bit8_cnt;
end

always@(posedge clk or posedge reset_p)
begin
if(reset_p)
    bit16_out <= 16'h0000;
else if(bit8_in_valid)
    bit16_out <= {bit16_out[7:0],bit8_in};
else
    bit16_out <= bit16_out;
end

always@(posedge clk or posedge reset_p)
begin
```

```
if(reset_p)
    bit16_out_valid <= 1'b0;
else if(bit8_in_valid && bit8_cnt)
    bit16_out_valid <= 1'b1;
else
    bit16_out_valid <= 1'b0;
end
endmodule
```

通过以上设计，位宽转换模块就设计完成了，其他模块和 IP 的配置请参看前面涉及的相关章节的内容。

34.4系统仿真验证与板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可，具体代码参见本章工程源码。顶层的仿真和 ddr3_ctrl_2port 模块仿真类似，这里就不做详细讲解，读者可以参靠本章工程源码完成顶层的仿真。

34.4.1管脚绑定

根据高云开发板提供的管脚绑定表，我们对 TFT 显示屏和串口的管脚，直接进行绑定即可。

由于涉及到 DDR3 的工程，管脚数量都不少，受制于篇幅，读者可以自己参考教程配套的工程代码完成管脚绑定。本工程涉及到绑定的部分，主要为时钟、复位基础管脚，有 DDR 相关管脚，串口相关管脚，TFT 相关管脚等几个部分。

34.4.2串口传图工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，准备进行上板验证。由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成数据流文件。上板调试硬件平台基于高云开发板，使用一根 Type-C 线，一端接入高云开发板上的 UART 接口，另一端接入 PC 机的 USB 口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接如下所示。

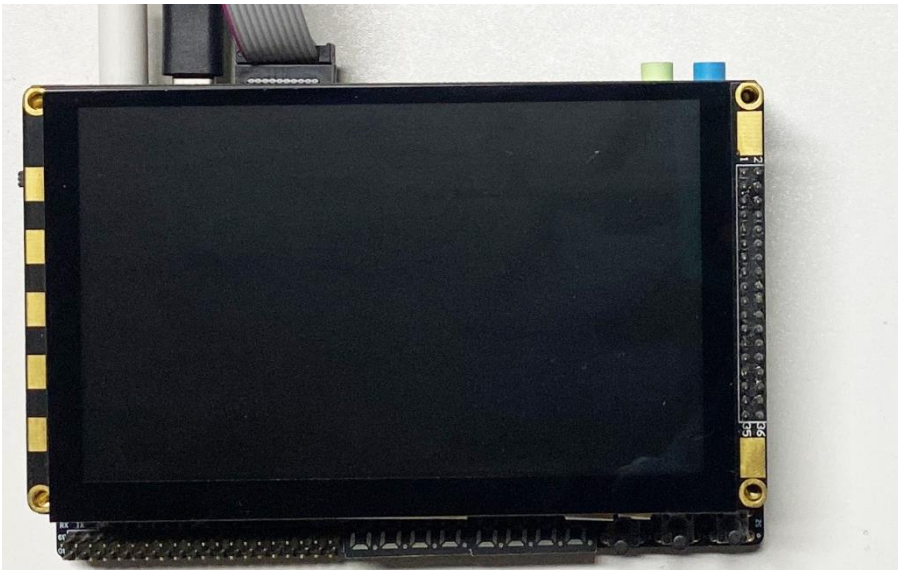


图 34-4 串口传图开发板连接图

连接完成之后，给开发板上电，然后下载数据流文件，下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态，如下图 34-5 所示。

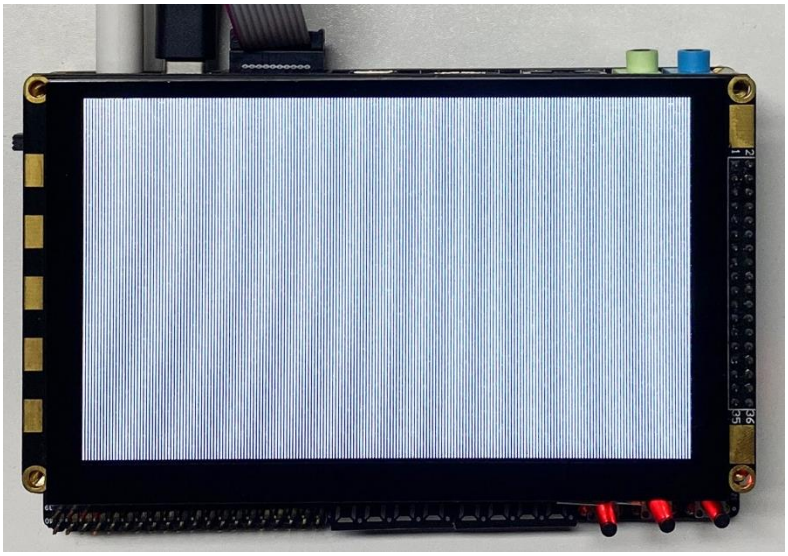


图 34-5 下载完程序后产生的碎花屏效果

出现这种花屏是因为这个时候，DDR3 中并未写入数据，显示的数据是不可知的一些数据。LED0 和 LED1 分别表示是 DDR 控制器内时钟锁相环的 locked 信号和 DDR 初始化校准完成信号的状态，亮表示这两个信号均变为高电平，说明 DDR 已经正常完成初始化和校准操作。接下来通过小梅哥串口传图工具向 FPGA 传输图片数据。小梅哥串口传图工具可在论坛下载。

图 34-6 启动串口传图工具

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径：图片宽度和高度设置成与显示屏分辨率一致（TFT5.0 寸屏是 800*480）。下图 34-7 是待传的图片。

图 34-7 需上传的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24 或 16 的 bmp 格式图片，图片的宽度和高度需要为 800*480（插 5 寸屏情况下）或 480*272（插 4.3 寸屏情况下）。

图 34-8 需传图片的属性信息

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置，设置好传图工具后，点击“连接设备”。

图 34-9 软件连接图片

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

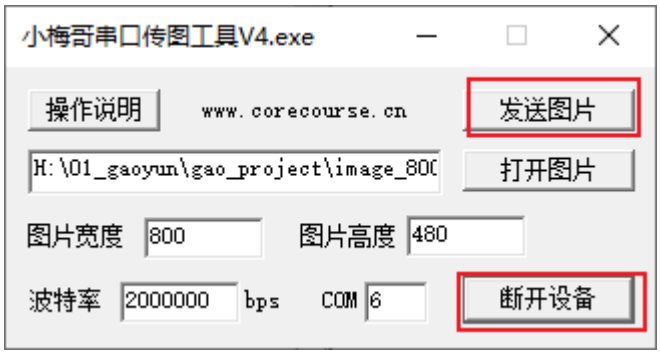


图 34-10 设置完图片信息后向对应的 com 端口号发送图片

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下图 34-11 所示。

图 34-11 串口传图完成后的图片效果

以上就完成了串口传图 TFT 显示的工程设计。

34.4.3串口传图工程中添加 HDMI 显示

结合前面讲解的 HDMI 章节知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器，当然，添加 HDMI 显示后，TFT 显示也可以保留。

为了实现以上目标，我们可以对工程作如下改进。

1. 新添加 pll 锁相环，添加 165M 时钟信号输出。

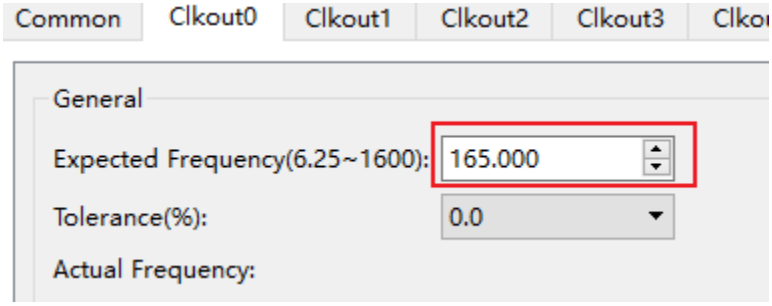


图 34-12 锁相环添加 165MHz 输出频率

2. 使用分频原语，将 165Mhz 进行 5 分频得到 33Mhz 的时钟，如下所示

```
CLKDIV u_clkdiv
(.RESETN(pll_locked)
,.HCLKIN(loc_clk165m) //clk  x5
,.CLKOUT(loc_clk33m)    //clk  x1
```

```
,.CALIB (1'b1)
);
    defparam u_clkdiv.DIV_MODE="5";
```

3. 在端口出添加 HDMI 输出管脚，在 CST 文件或者管脚配置界面进行 HDMI 管脚绑定。

```
//hdmi interface
output          hdmi_clk_p   ,
output          hdmi_clk_n   ,
output [2:0]    hdmi_dat_p   ,
output [2:0]    hdmi_dat_n   ,
```

4. 添加 HDMI 接口例化：

```
dvi_encoder u_dvi_encoder(
    .pixelclk(pixelclk),// system clock
    .pixelclk5x(pixelclk5x),// system clock x5
    .rst_n(~g_rst_p),// reset
    .blue_din(disp_blue),// Blue data in
    .green_din(disp_green),// Green data in
    .red_din(disp_red),// Red data in
    .hsync(disp_hs),// hsync data
    .vsync(disp_vs),// vsync data
    .de(disp_de),// data enable
    .tmds_clk_p(hdmi_clk_p),
    .tmds_clk_n(hdmi_clk_n),
    .tmds_data_p(hdmi_dat_p),//rgb
    .tmds_data_n(hdmi_dat_n) //rgb
);
```

5. RGB565 和 RGB888 的转换

```
assign pixelclk   = loc_clk33m;
assign pixelclk5x = loc_clk165m;
assign disp_red   = {TFT_rgb[15:11],3'b0};
assign disp_green = {TFT_rgb[10:5],2'b0};
assign disp_blue  = {TFT_rgb[4:0],3'b0};
assign disp_hs    = TFT_hs;
assign disp_vs    = TFT_vs;
assign disp_de    = TFT_de;
```

6. 添加 HDMI 管脚绑定

68	hdmi_clk_n	output		N9	5	False	LVCMOS33	8
69	hdmi_clk_p	output		M10	5	False	LVCMOS33	8
70	hdmi_dat_n[0]	output		T7	5	False	LVCMOS33	8
71	hdmi_dat_n[1]	output		V6	5	False	LVCMOS33	8
72	hdmi_dat_n[2]	output		T5	5	False	LVCMOS33	8
73	hdmi_dat_p[0]	output		R7	5	False	LVCMOS33	8
74	hdmi_dat_p[1]	output		T6	5	False	LVCMOS33	8
75	hdmi_dat_p[2]	output		R5	5	False	LVCMOS33	8

图 34-13 添加 HDMI 管脚绑定

7. 完成以上配置后，在前面给出的线缆连接图中，添加开发板 HDMI 接口和液晶显示器的连接。

图 34-14 串口传图 HDMI 显示硬件搭建

8. 将改造好的工程下载到 FPGA 开发板，得到如下实验现象。

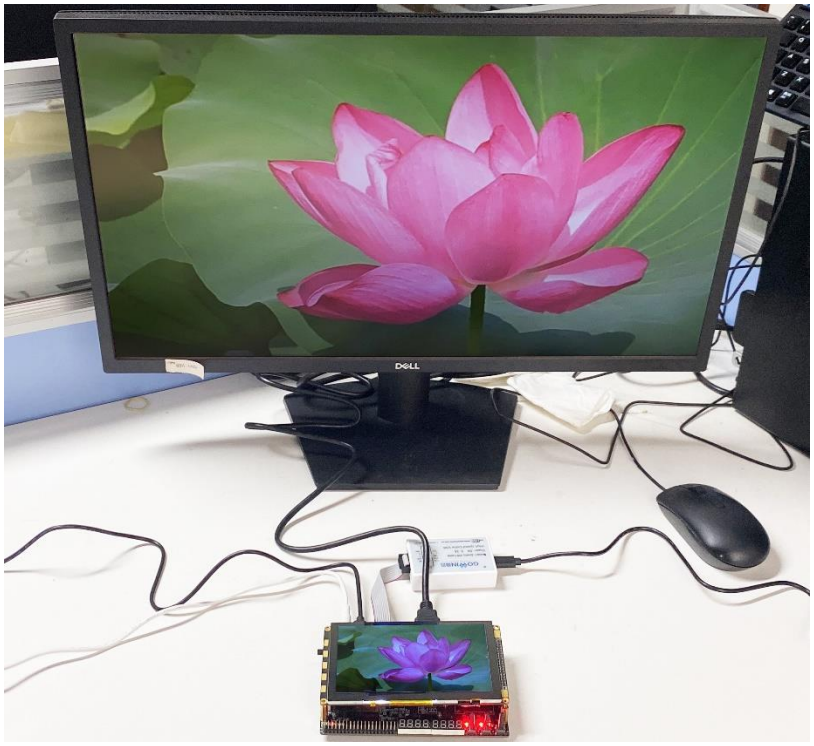


图 34-15 串口传图 TFT 和 HDMI 联合显示实验效果

至此，串口传图帧缓存系统在高云开发板上的设计及验证就成功完成了。

34.5总结

本章讲解了基于 DDR3 的串口传图帧缓存系统的设计与实现方法。通过本章的学习，各位读者可以更深入的对输入控制器，缓存控制器，输出控制器的设计有一个更为全面宏观的认识，在这些控制器协调工作时，既需要关注数据在流动中形态的不同，又需要关注数据收发控制信号和不同时钟频率信号的协调。

35 彩色图像灰度化的设计实现(HDMI 和 TFT 显示)

工程源码	|----02_设计实例 |----ch35_uart_ddr3_tft_hdmi_rgb2gray
相关视频课程	
说明	

章节导读

本章节主要是在前述串口传图章节的基础上，加入图像处理模块，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，然后将原始彩色图像和处理后的图像数据并进行实时彩色图像灰度化处理，然后将原始彩色图像和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏上同时显示处理前的彩色图像和处理后的灰度图像。

35.1图像处理基础知识

在上一节，我们讲解了基于 DDR3 的串口传图帧缓存系统设计实现方法，而从本章节开始，我们将延续上一章节内容，正式开启图像处理这一部分的内容学习。

35.1.1数字图像处理技术的诞生背景

数字图像处理是指将图像信号转换成数字信号并利用计算机对其进行处理的过程。图像处理技术最早出现于 20 世纪 50 年代，当时的点子计算机技术已经发展到一定水平，人们已经开始利用计算机来处理图形和图像信息。数字图像处理作为一门学科大约形成于 20 世纪 60 年代初期。早期的图像处理的目的是改善图像的质量，它以人为对象，以改善人的视觉效果为目的。图像处理中，输入的是质量低的图像，输出的是改善质量后的图像，常用的图像处理方法有图像增强、复原、编码、压缩等。

数字图像处理常用方法有以下几种：图像变换、图像编码压缩、图像增强和复原、图像分割、图像描述、图像分类（识别）。接下来，我们对这几种常用方法的应用场合进行逐一介绍。

35.1.2图像变换

由于图像分辨率越来越高，导致图像阵列占用的存储空间越来越大。如果

直接在空间域中对图像阵列进行处理，涉及计算量很大。因此，往往采用各种图像变换的方法，如傅立叶变换、沃尔什变换、离散余弦变换等间接处理技术，将空间域的处理转换为变换域处理。这样做的好处是不仅可减少计算量，而且可获得更有效的处理效果（如傅立叶变换可在频域中进行数字滤波处理）。目前新兴研究的小波变换在时域和频域中都具有良好的局部优化特性，它在图像处理中也有着广泛而有效的应用。

35.1.3图像编码压缩

图像编码压缩技术可减少描述图像的数据量（即比特数），以便节省图像传输、处理时间和减少所占用的存储器容量。压缩可以在不失真的前提下获得，也可以在允许的失真条件下进行。编码是压缩技术中最重要的方法，它在图像处理技术中是发展最早且比较成熟的技术。

35.1.4图像增强和复原

图像增强和复原的目的是为了提高图像的质量，如去除噪声，提高图像的清晰度等。图像增强不考虑图像降质的原因，只突出图像中所感兴趣的部分。如强化图像高频分量，可使图像中物体轮廓清晰，细节明显；如强化低频分量可减少图像中噪声影响。图像复原要求对图像降质的原因有一定的了解，一般讲应根据降质过程建立“降质模型”，再采用某种滤波方法，恢复或重建原来的图像。

35.1.5图像分割

图像分割是数字图像处理中的关键技术之一。图像分割的目标，是将图像中有意义的特征部分提取出来。这些有意义的特征包括图像中的边缘、区域等。利用这些特征，可以进一步进行图像识别、分析和理解。虽然目前已研究出不少边缘提取和区域分割的方法，但目前还没有一种普遍适用于分割提取各种图像的有效方法。因此，学界对图像分割的研究还在不断深入之中。同时，图像分割也是目前图像处理中研究的热点之一。

35.1.6图像描述

图像描述是图像识别和理解的必要前提。作为最简单的二值图像可采用其几何特性描述物体的特性，一般图像的描述方法采用二维形状描述，它有边界

描述和区域描述两类方法。对于特殊的纹理图像可采用二维纹理特征描述。随着图像处理研究的深入发展，学界已经开始进行三维物体描述的研究，提出了体积描述、表面描述、广义圆柱体描述等方法。

35.1.7图像分类（识别）

图像分类与识别属于模式识别的范畴，其主要内容是图像经过某些预处理（增强、复原、压缩）后，进行图像分割和特征提取，从而进行判决分类。图像分类常采用经典的模式识别方法，有统计模式分类和句法（结构）模式分类，近年来新发展起来的模糊模式识别和人工神经网络模式分类在图像识别中也越来越受到重视。

随着计算机技术的发展，图像处理技术已经深入到我们生活中的方方面面，其中，在娱乐休闲上的应用已经深入人心。图像处理技术在娱乐中的应用主要包括：电影特效制作、电脑电子游戏、数码相机、视频播放、数字电视等。

电影特效制作：自从 20 世纪 60 年代以来，随着电影中逐渐运用了计算机技术，一个全新的电影世界展现在人们面前，这也是一次电影的革命。越来越多的计算机制作的图像被运用到了电影作品的制作中。其视觉效果的魅力有时已经大大超过了电影故事的本身。如今，我们已经很难发现一部没有利用任何计算机数码元素的新电影。

电脑电子游戏：电脑电子游戏的画面，是近年来电子游戏发展最快的部分之一。从 1996 年到现在，游戏画面的进步简直可以用突飞猛进来形容，随着图像处理技术的发展，众多在几年前无法想象的画面在今天已经成为了平平常常的东西。

数码相机：所谓数码相机，是一种能够进行拍摄，并通过内部处理把拍摄到的景物转换成以数字格式存放图像的特殊照相机。于普通相机不同，数码相机并不使用胶片，而是使用固定的或者是可拆卸的半导体存储器来保存获取的图像。数码相机可以直接连接到计算机、电视机或者打印机上。在一定条件下，数码相机还可以直接接到移动式电话机或者手持 PC 机上。由于图像是内部处理的，所以使用者可以马上检查图像是否正确，而且可以立刻打印出来或是通过电子邮件传送出去。

视频播放与数字电视：家庭影院中的 VCD ， DVD 播放器和数字电视中，大量使用了视频编码解码等图像处理技术，而视频编码解码等图像处理技术的发展，也推动了视频播放与数字电视向高清晰，高画质方向发展。

后续章节，我们将依次介绍基于 FPGA 实现数字图像处理的相关内容。在硬件上，后续图像处理章节将基于前述串口传图到 DDR3 然后显示在 TFT/HDMI 显示器实验的程序框架，对需要送往显示设备显示的图像数据进行常见数字图像处理的相关运算。这些图像处理算法包括：RGB 彩色图像转灰度图像、图像中值滤波算法、图像均值滤波算法、图像高斯滤波算法以及基于 Sobel 算子的边缘检测算法等。

学习数字图像处理相关章节的内容，核心在于对目标任务的实现算法的理解，在对算法的理解基础上，利用 Verilog HDL 语言实现这些算法。

数字图像处理算法理论的建立和推导是一个严谨且科学的数学过程，当前广泛采用的数字图像处理算法，其理论基础已经由图像处理相关领域的专家进行了深度的分析和论证。本章内容，重点在于介绍各种数字图像处理算法在 FPGA 平台上的具体实现方法，而对于各种数字图像处理算法的理论推导和验证，本章内容不会过多涉及。各位读者有兴趣的，可自行根据专业的数字图像处理教材进行学习和验证。

那么接下来，我们就正式开启图像处理章节相关课程。首先，我们先来分析彩色图像灰度化的设计实现。

35.2灰度图像的相关概念

在正式入题之前先给大家讲解一些灰度（gray）图像，YUV 图像以及 Ycbcr 图像。

Gray 图像：灰度（gray）图像就是我们常说的黑白图像，由黑到白为灰阶，其值域为 0~255（8bit）。

YUV 图像：YUV 是被欧洲电视系统所采用的一种颜色编码方法（属于 PAL），是 PAL 和 SECAM 模拟彩色电视制式采用的颜色空间。在现代彩色电视系统中，通常采用三管彩色摄像机或彩色 CCD 摄像机进行取像，然后把取得的彩色图像信号经分色、分别放大校正后得到 RGB 图像，RGB 图像经过矩阵变换电路得到亮度辛哈 Y 和两个色差信号 B-Y（即 U）、R-Y（即 V），最后发送端讲亮度和色差三个信号分别进行编码，用同一信道发送出去。这种色彩的表示方法就是所谓的 YUV 色彩空间表示。YUV 色彩空间的特点式它的亮度信号 Y 和色度信号 U、V 是分离的。YUV 主要用于优化彩色视频信号的传输，使其向后兼容老式黑白电视。与 RGB 视频信号传输相比，它最大的优点在于只需占用极少的频宽（RGB 要求三个独立的视频信号同时传输）就能完成图像传输。

对于 YUV 三种图像传输信号分量，每一种分量起始也都是有其物理意义的。其中“Y”表示明亮度（Luminance 或 Luma），也就是灰阶值；而“U”和“V”表示的则是色度（Chrominance 或 Chroma），作用是描述影像色彩及饱和度。如果 U 和 V 明确，那么像素的颜色就是可以指定和还原的。

由于计算机的普及，一种适合于在计算机显示系统中得到应用的 YUV 制式，即 YCbCr 得到更加广泛的运用。

如果将 RGB 信号的特定部分叠加到一起，则可以明确“亮度”值。“色度”U 和 V 则定义了颜色的两个方面—色调和饱和度，分别用 Cr 和 Cb 来表示。其中，Cr 反映了 RGB 输入信号红色部分与 RGB 信号亮度值之间的差异。

在计算机数字信号的绝对统治力下，YCbCr 也逐渐成为了 YUV 制式的典型代表。但是各位读者需明白，YCbCr 和 YUV 是两个不同的概念，YCbCr 只是 YUV 的其中一种最适宜用于计算机显示的图像信号格式。

使用灰度图像可以实现：在保留图像显示的内容信息的同时，弱化图像的颜色信息。虽然人眼观察经过灰度化处理后的图像信息，体验感没有彩色图像逼真，但相较于彩色图像而言，灰白图像的后续分析会大大减轻硬件负担。这样，不太关心图像体验感的用户可以更高效的利用存储资源和总线资源，尽快获得图像中期望的内容。

用一句俗话来说：颜色信息，是非常“吃硬件”的，于是有人就想，能不能剥离颜色信息，而进行图像数据的关键信息判断呢？打个不恰当的比方如下：如果你是汽车驾驶员，对于驾驶汽车这个任务，只关心前方是否有人，有坑，有障碍物，前车距离是多远，而并不关心道路是什么颜色，道路两旁的房屋漂不漂亮。驾驶员的大脑，并不需要对过多无关的颜色信息进行分析（当然，红绿灯还是要看一看的），而只需要简单的判断前方是否安全即可。

35.3系统整体设计

介绍了灰度图像的实现意义以后，我们来看 RGB 彩色图像灰度化实现的整体设计方案。最终 TFT 显示的要求如下图 35-1 所示。

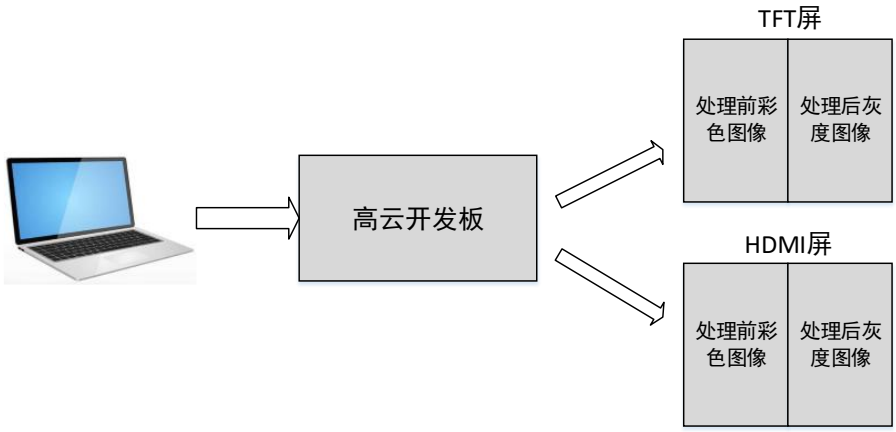


图 35-1 彩色图像灰度化的实验目标

系统整体设计框图如下图 35-2。从 FPGA 设计架构来说，彩色图像灰度化与“基于 DDR3 的串口传图帧缓存系统设计实现”架构基本一致，增加了彩色图像灰度化处理模块 rgb2grag 模块和图像合并模块 image_stitche_x 模块。

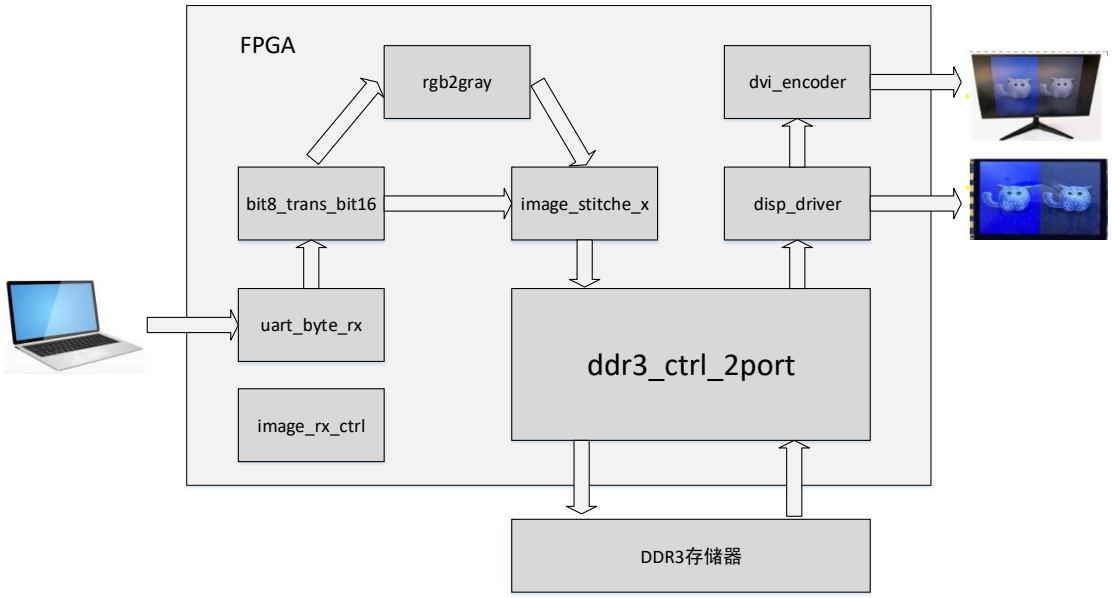


图 35-2 系统设计框图

其中，

（1）uart_byte_rx 模块：负责串口图像数据的接收，该模块的设计前面有讲，但是需要注意的是，前面在讲解串口接收时，为了避免出现非常强的电磁干扰，导致采集的信号被干扰而使得结果出错，我们就将每一位数据平均分成 16 小段，取中间比较稳定的部分作为最终的结果输出，但是在串口传图相关实验中，不必考虑这种干扰，直接将每一位接收到的数据输出即可，这样将会减少传输数据的时间，考虑到传输时间，本次设计我们将波特率设置为 2000000bps。

（2）bit8_trans_bit16 模块：将串口接收的每两个 8bit 数据转换成一个 16bit 数据（图像数据式 16bit 的 RGB565 的数据，电脑式通过串口将一个像素点数据分两次发送到 FPGA，FPGA 需要将串口接收数据重组成 16bit 的图像数据），实现过程相对简单。

（3）rgb2gray 模块：彩色图像灰度化处理，对串口接收的彩色图像数据实时进行灰度化处理。

（4）image_stitche_x 模块：将串口接收的尺寸为 400*480 大小的彩色图像与灰度化处理后的 400*480 大小的图像数据以左右形式合并成一张 800*480 的图像。

（5）disp_driver 模块：tft 屏显示驱动控制，对缓存在 DDR3 中的图像数据进行显示。

（6）ddr3_ctrl_2port 模块组：包含 wr_ddr3_fifo、rd_ddr3_fifo、fifo_ddr3_adapter 以及 DDR3_Memory_Interface_Top 模块，完成采集的图像数据缓存。

（7）pll 模块：上述各个模块所需时钟的产生，使用 PLL IP。

（8）image_rx_ctrl 模块。用于显示接收进度，给出接收完成标志。

（9）dvi_encoder 模块：用于将生成信号转换成 HDMI 输出信号作 HDMI 显示。

本系统就是在上一系统基础上添加图像处理模块搭建系统。除去使用 IP 和前面章节讲过的模块外，还需要设计的模块包括 rgb2gray 模块和 image_stitche_x 模块。

35.4彩色图像灰度化处理模块的设计

35.4.1基本原理

将彩色图像转换为灰度图像的过程称为图像灰度化处理。常见的 24 位深度彩色图像 RGB888 中的每个像素的颜色由 R、G、B 三个分量决定，并且三个分量各占 1 个字节，每个分量可以取值 0~255，这样一个像素点可以有 1600 多万（255*255*255）的颜色的变化范围。而灰度图像是 R、G、B 三个分量相同的一种特殊的彩色图像，其中一个像素点的变化范围为 0~255。对于一幅彩色图来说，其对应的灰度图则是只有 8 位的图像深度，这也说明了用灰度图做图像

处理所需的计算量确实要少。不过需要注意的是，虽然丢失了一些颜色等级，但是从整幅图像的整体和局部的色彩以及亮度等级分布特征来看，灰度图描述与彩色图的描述是一致的。一般有分量法、最大值法、平均值法、加权平均法四种方法对彩色图像进行灰度化。

35.4.2彩色图像灰度化处理方法介绍

35.4.2.1方法 1：分量法

将彩色图像中的三分量的亮度作为三个灰度图像的灰度值，可根据应用需要选取一种灰度图像。具体表达式如下。

$$\text{gray}_1(i,j) = R(i,j)$$
$$\text{gray}_2(i,j) = G(i,j)$$
$$\text{gray}_3(i,j) = B(i,j)$$

其中，$\text{gray}_1(i,j), \text{gray}_2(i,j), \text{gray}_3(i,j)$为转换后的灰度图像在（i，j）处的灰度值，*R(i,j)*，*G(i,j)*，*B(i,j)*分别为转换前的彩色图像在（i，j）处 R、G、B 三个分量的值。

35.4.2.2方法 2：最大值法

将彩色图像中的三分量亮度 R，G，B 的最大值作为灰度图的灰度值。具体表达式如下。

$$\text{gray(i,j)} = \max[R(i,j), G(i,j), B(i,j)]$$

35.4.2.3方法 3：平均值法

将彩色图像中的三分量亮度求平均得到一个灰度值。如下:

$$\text{gray(i,j)} = \frac{R(i,j) + G(i,j) + B(i,j)}{3}$$

上式中有除法，考虑到在 FPGA 中实现除法比较的消耗资源，这里在实现前可以先做如下的近似处理。可以将上面公式乘以 3/256，这样就需要同时乘以 256/3 保证公式的正确性。公式处理过程如下：

$$\text{gray(i,j)} = \frac{[R(i,j) + G(i,j) + B(i,j)]}{3} * \frac{3}{256} * \frac{256}{3}$$
$$= \frac{[R(i,j) + G(i,j) + B(i,j)]}{256} * \frac{256}{3}$$

对 256/3 做近似取整处理，将 256/3 替换成 85，则公式变为如下。

$$\text{gray(i,j)} \approx \frac{[R(i,j)+G(i,j)+B(i,j)]}{256} * 85$$

这样式子中除以 256 就可以采用移位方法来处理，式子变为如下：

$$\text{gray(i,j)} = \{[R(i,j)+G(i,j)+B(i,j)] * 85\} \gg 8$$

上面处理过程中使用是对 256/3 的近似处理，当然这里可以采用其他数据，比如 512/3、1024/3、2048/3 等等，基本的原则是将平均公式法中分母的 3 替换成 2 的幂次的数，这样除法就可以使用移位的方式实现，减小 FPGA 中由于存在除法带来的资源消耗。

35.4.2.4方法 4：加权平均法

根据重要性及其它指标，将三个分量以不同的权值进行加权平均。有一个很著名的心理学公式：

$$\text{gray(i,j)} = 0.299 * R(i,j) + 0.587 * G(i,j) + 0.114 * B(i,j)$$

这里 0.299+0.587+0.114=1，刚好是满偏，这是通过不同的敏感度以及经验总结出来的公式，一般可以直接用这个。在实际应用时，为了能避免低速的浮点运算以及除法运算，可以先将式子缩放 1024 倍来实现运算算法，如下：

$$\text{gray(i,j)} = [0.299 * R(i,j) + 0.587 * G(i,j) + 0.114 * B(i,j)] * 1024/1024$$

通过近似取整处理后得到近似公式如下。

$$\text{gray(i,j)} \approx [306 * R(i,j) + 601 * G(i,j) + 117 * B(i,j)]/1024$$

式子中除以 1024（这里是 2 的 n 次方就可以，n 不同，结果会略微有差别）可以采用移位方法来处理，式子变为如下：

$$\text{gray(i,j)} \approx [306 * R(i,j) + 601 * G(i,j) + 117 * B(i,j)] \gg 10$$

也可以压缩到 8 位以内，式子变为如下。具体压缩到多少位可以根据实际需求。

$$\text{gray(i,j)} = [77 * R(i,j) + 150 * G(i,j) + 29 * B(i,j)] \gg 8$$

35.4.3彩色图像灰度化处理模块设计验证

上述中方法 1、2 实现起来相对容易，这里就不多说。接下俩主要是针对方法 3、4 在 FPGA 上的实现做讲解。

35.4.3.1方法 3 平均值法的实现

该方法实现起来并不复杂，通过上面的计算公式可以知道，计算公式里只有加法、乘法和移位计算，这里的乘法通过移位相加的方式进行计算，计算具体实现见下面代码。

```
//求平均法 GRAY = (R+B+G)/3=（(R+B+G)*85）>>8
wire [9:0]sum;
reg [15:0]gray_r;

assign sum = red_8b_i + green_8b_i + blue_8b_i;

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        gray_r <= 16'd0;
    else if(rgb_valid)
        gray_r <= (sum << 6)+(sum << 4)+(sum << 2)+ sum;
    else
        gray_r <= 16'd0;
end

assign gray_8b_o = gray_r[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid;
    gray_hs    <= rgb_hs;
    gray_vs    <= rgb_vs;
end
```

对该模块的仿真也相对比较简单，只需要在 testbech 中给时钟复位激励以及 RGB 三通道的图像数据即可，具体代码如下，仿真中分别给 R、G、B 通道不同的是起始数据值，然后通过递增加 1 的形式改变 R、G、B 通道数据，验证设计的正确性。

```
initial begin
    reset_p    = 1;
    rgb_valid  = 0;
    red_8b_i   = 0;
    green_8b_i = 0;
    blue_8b_i  = 0;
    #(`CLK_PERIOD*200+1);
    reset_p    = 0;
    red_8b_i   = 56;
    green_8b_i = 124;
```

```
    blue_8b_i  = 203;
    #2000;

    rgb_valid = 1;
    repeat(256)begin
    #(`CLK_PERIOD)
    red_8b_i   = red_8b_i   + 1;
    green_8b_i = green_8b_i + 1;
    blue_8b_i  = blue_8b_i  + 1;
    end
    rgb_valid = 0;

    #2000;
    $stop;
end
```

为了能验证采用平均值法实现彩色图像灰度化计算的正确性，在仿真代码中加入了如下代码。直接通过平均法的原始公式产生一组对比数据。

```
always@(posedge clk)
begin
    if(rgb_valid == 1'b1)
        comp1_gray <= (red_8b_i + green_8b_i + blue_8b_i)/3;
    else
        comp1_gray <= 0;
end
```

这样可以比较容易的通过对比 comp1_gray 和 gray_8b_o 的值来验证设计模块的正确性。其仿真波形图如下：

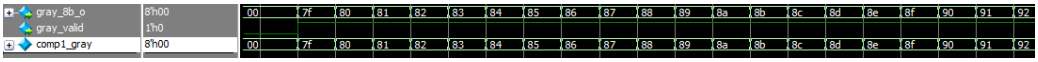


图 35-3 仿真波形图 1

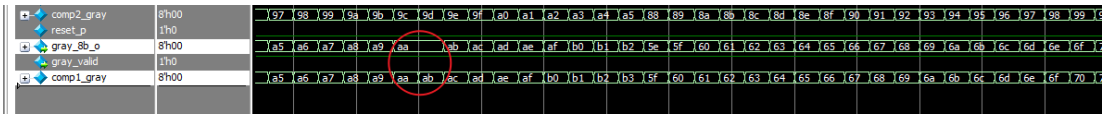


图 35-4 仿真波形图 2

从上述图中可以看出，从一个地方开始 comp1_gray 和 gray_8b_o 的值相差 1，可以分析下出现这个问题的原因，其实 gray_8b_o 和 comp1_gray 的计算公式并非完全一样，gray_8b_o 的计算公式是为了避免除法做了一定的近似，而在仿真文件中 comp1_gray 是直接求的平均。可以通过 red、green、blue 这 3 个数据分别对 gray_8b_o 和 comp1_gray 进行计算，计算结果与仿真波形结果是一致的。这也说明了，避免除法做近似处理计算的 gray 和直接通过除法求的平均值 comp1_gray 是稍存在偏差的。这个是可以接受的误差范围。

35.4.3.2方法 4 加权平均法的实现

对于加权平均法可通过两种方式实现，公式直接计算法和查找表法。

35.4.3.2.1 公式直接计算法

公式直接计算法与方法 3 实现类似，通过转换公式直接进行计算，只是具体计算数值发生了变化，同样乘法采用移位相加的方式实现。具体代码如下：

```
//典型灰度转换公式 Gray = R*0.299+G*0.587+B*0.114=(R*77 + G*150 +
B*29) >>8
wire [15:0]red_x77;
wire [15:0]green_x150;
wire [15:0]blue_x29;
reg  [15:0]sum;

//乘法转换成移位相加方式
assign red_x77 = (red_8b_i  << 6) + (red_8b_i  << 3) + (red_8b_i  <<
2) + red_8b_i;
assign green_x150 = (green_8b_i<< 7) + (green_8b_i<< 4) +
(green_8b_i<< 2) + (green_8b_i<<1);
assign blue_x29   = (blue_8b_i << 4) + (blue_8b_i << 3) + (blue_8b_i
<< 2) + blue_8b_i;

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
    sum <= 16'd0;
    else if(rgb_valid)
    sum <= red_x77 + green_x150 + blue_x29;
    else
    sum <= 16'd0;
end

assign gray_8b_o = sum[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid;
    gray_hs    <= rgb_hs;
    gray_vs    <= rgb_vs;
end
```

35.4.3.2.2 公查找表法

通过观察计算公式发现，R、G、B 数据值均乘以了一个定值，然后对乘法之后的结果相加，最后右移 8 位，上面采用直接计算法实现是对常数乘法采用

的移位相加方法计算，对于这类固定范围内的数值，同时可取数据不多情况下（这里 R、G、B 数值范围在 0~255，可取的数据有限）乘以一个常数，可以采用查找表方法实现，该方法主要的优势是直接通过访问 ROM 内的数据，相对使用移位相加实现乘法使用的 ALU 资源会少点，但占用的存储器会更多。因为需要将 R、G、B 乘以系数之后的数值存储在 ROM 中，然后通过读取 ROM 方式来得到计算之后的数值。这里使用 Gowin 软件添加 3 个 ROM IP 核，分别通过 R*75、G*147、B*36（0≤R≤255，0≤G≤255，0≤B≤255）的计算值建立 3 个初始化 mi 文件，然后在 ROM IP 核中分别添加 mi 文件进行初始化。具体代码如下：代码中 rom_red_x77、rom_green_x150、rom_blue_x29 分别存储着 R* 75、G*147、B*36（0≤R≤255，0≤G≤255，0≤B≤255）256 个数值。在提供的工程目录下的\uart_ddr3_tft_rgb2gray \src \mi 文件夹中有提供这 3 个 mi 文件。使用查找表法实现彩色图像灰度化的代码如下。

```
//查找表方式，可以省去公式法中乘法运算 Gray =(R*77 + G*150 + B*29) >>8，将
3 个分量乘以系数后的数值存储在 ROM 中
wire [14:0]red_x77;
wire [15:0]green_x150;
wire [13:0]blue_x29;
reg  [15:0]sum;
reg        rgb_valid_dly1;
reg        rgb_hs_dly1;
reg        rgb_vs_dly1;

rom_red_x77 rom_red_x77(
    .dout(red_x77 ), //output [14:0] dout
    .clk(clk), //input clk
    .oce(rgb_valid), //input oce
    .ce(rgb_valid), //input ce
    .reset(reset_p), //input reset
    .ad(red_8b_i) //input [7:0] ad
);

rom_green_x150 rom_green_x150(
    .dout(green_x150), //output [15:0] dout
    .clk(clk), //input clk
    .oce(rgb_valid), //input oce
    .ce(rgb_valid), //input ce
    .reset(reset_p), //input reset
    .ad(green_8b_i) //input [7:0] ad
);

rom_blue_x29 rom_blue_x29(
    .dout(blue_x29), //output [13:0] dout
```

```
    .clk(clk), //input clk
    .oce(rgb_valid), //input oce
    .ce(rgb_valid), //input ce
    .reset(reset_p), //input reset
    .ad(blue_8b_i) //input [7:0] ad
);

always@(posedge clk)
begin
    rgb_valid_dly1 <= rgb_valid;
    rgb_hs_dly1    <= rgb_hs;
    rgb_vs_dly1    <= rgb_vs;
end

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        sum <= 16'd0;
    else if(rgb_valid_dly1)
        sum <= red_x77 + green_x150 + blue_x29;
    else
        sum <= 16'd0;
end

assign gray_8b_o = sum[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid_dly1;
    gray_hs    <= rgb_hs_dly1;
    gray_vs    <= rgb_vs_dly1;
end
```

仿真过程与方法 3 类似，这里就不重复描述。对于方法 4 的两种不同实现方式，可以对比下综合后消耗的资源情况。下图 35-5 和下图 35-6 分别为使用公式直接计算法和查找表法实现使用资源情况。

File	Register	LUT	ALU	DSP	BSRAM	SSRAM
uart_ddr3_tft_hd... src\uart_ddr3_tft_hdmi_rgb2gray.v	3976 (0)	4059 (1)	479 (0)	0 (0)	23 (0)	12 (0)
ddr3_pll(ddr3_pll) src\ddr3_pll\ddr3_pll.v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
pll(pll) src\pll\pll.v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
uart_byte_rx(use_... src\uart_byte_rx\uart_byte_rx.v	38 (38)	43 (43)	0 (0)	0 (0)	0 (0)	0 (0)
bit8_trans_bit16(u... src\bit8_trans_bit16\bit8_trans_bit16.v	18 (18)	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)
image_rx_ctrl(ima... src\image_rx_ctrl\image_rx_ctrl.v	33 (33)	56 (56)	0 (0)	0 (0)	0 (0)	0 (0)
rgb2gray(rgb2gr... src\rgb2gray\rgb2gray.v	7 (7)	6 (6)	89 (89)	0 (0)	0 (0)	0 (0)
image_stitche_x(i... src\image_stitche\image_stitche_x.v	182 (36)	248 (52)	20 (0)	0 (0)	2 (0)	6 (0)
image_buffer(... src\image_buffer\image_buffer.v	73 (73)	98 (98)	10 (10)	0 (0)	1 (1)	3 (3)
~fifo.imag... src\image_buffer\image_buffer.v						

图 35-5 公式直接计算法使用资源情况

	File	Register	LUT	ALU	DSP	BSRAM	SSRAM
uart_ddr3_tft_hd...	**src\uart_ddr3_tft_hdmi_rgb2gray.v**	3977 (0)	4056 (1)	421 (0)	0 (0)	26 (0)	12 (0)
ddr3_pll(ddr3_pll)	src\ddr3_pll\ddr3_pll.v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
pll(pll)	src\pll\pll.v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
uart_byte_rx(use_...	src\uart_byte_rx\uart_byte_rx.v	38 (38)	43 (43)	0 (0)	0 (0)	0 (0)	0 (0)
bit8_trans_bit16(u...	src\bit8_trans_bit16\bit8_trans_bit16.v	18 (18)	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)
image_rx_ctrl(ima...	src\image_rx_ctrl\image_rx_ctrl.v	33 (33)	55 (55)	0 (0)	0 (0)	0 (0)	0 (0)
rgb2gray(rgb2gr...	src\rgb2gray\rgb2gray.v	8 (8)	6 (6)	31 (31)	0 (0)	3 (0)	0 (0)
rom_red_x77(...	src\gowin_prom\rom_red_x77.v	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
rom_green_x1...	src\rom_green_x150\rom_green_x150.v	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
rom_blue_x29(...	src\rom_blue_x29\rom_blue_x29.v	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
image_stitche_x(i...	src\image_stitche\image_stitche_x.v	182 (36)	248 (52)	20 (0)	0 (0)	2 (0)	6 (0)

图 35-6 查找表法使用资源情况

比较两种方法使用资源情况，可以很明显的看出查找表方式消耗的 ALU 相对少很多，但是消耗了 3 个 BSRAM 资源。

对于同一功能多种不同实现方法的模块代码如何整合到一起呢？当然每种方法作为一个单独的模块使用一个.v 文件保存肯定是没有问题的，这个就不太便于后期的维护和使用。如果能将多种实现方法整合到一个模块保存在一个.v 文件，使用起来就更加的方便。方法肯定是有的，而且还不只一种。下面提供两种方式，宏定义法，和使用 generate -if 方法。宏定义法相对比较好理解，通过不同的宏定义条件编译方式进行选择某种实现方式，实现的部分代码如下。该种方法可通过修改模块代码的宏定义选择不同的方法，还算是比较方便的。

```
//`define AVERAGE    //求平均法
//`define FORMULA    //直接公式法
`define LUT        //查找表法

module rgb2gray (
  clk,
  rst_n,
  rgb_valid,
  red_8b_i,
  green_8b_i,
  blue_8b_i,
  gray_8b_o,
  gray_valid
);
`ifdef AVERAGE  //求平均法 GRAY=(R+B+G)/3=（(R+B+G)*85）>>8
...//方法 1
`endif

`ifdef FORMULA //灰度转换公式 Gray = R*0.299+G*0.587+B*0.114
...//方法 2
`endif
```

```
`ifdef LUT//查找表方式
...//方法 3
`endif

endmodule
```

使用 generate-if 方法也是比较好的办法。先看该种方法的代码如下。

```
module rgb2gray
#(
  parameter PROC_METHOD = "AVERAGE" //"AVERAGE"      :求平均法
                                    //or "FORMULA"   :直接公式法
                                    //or "LUT"       :查找表法
)
(
  clk,
  rst_n,
  rgb_valid,
  red_8b_i,
  green_8b_i,
  blue_8b_i,
  gray_8b_o,
  gray_valid
);

generate
  if (PROC_METHOD == "AVERAGE") begin: PROC_AVERAGE
//-------------------------------------------
//求平均法 GRAY = (R+B+G)/3=（(R+B+G)*85）>>8
      //方法 1
//-------------------------------------------
  end
  else if (PROC_METHOD == "FORMULA") begin: PROC_FORMULA
//-------------------------------------------
//典型灰度转换公式 Gray = R*0.299+G*0.587+B*0.114=(R*77 + G*150 +
B*29) >>8
      //方法 2
//-------------------------------------------
  end
  else if(PROC_METHOD == "LUT") begin: PROC_LUT
//-------------------------------------------
//查找表方式，可以省去公式法中乘法运算 Gray =(R*77 + G*150 + B*29) >>8，将
3 个分量乘以系数后的数值存储在 ROM 中
      //方法 3
//-------------------------------------------
  end

endgenerate
```

```
endmodule
```

该种方式相对于宏定义条件编译来说更加方便，就是在模块被例化调用时，不管需要使用哪种方式都不需要去通过修改模块源代码方式去改变具体实现方法，直接在例化模块时就可通过重定义参数PROC_METHOD实现不同方法的设置。具体代码如下。

```
rgb2gray
#(
    .PROC_METHOD("FORMULA")  //"AVERAGE"      :求平均法
                             //or "FORMULA"   :直接公式法
                             //or "LUT"       :查找表法
)rgb2gray_average
(
    .clk       (clk       ),
    .reset_p   (reset_p   ),
    .rgb_valid (rgb_valid ),
    .red_8b_i  (red_8b_i  ),
    .green_8b_i(green_8b_i),
    .blue_8b_i (blue_8b_i ),
    .gray_8b_o (gray_8b_o ),
    .gray_valid(gray_valid)
);
```

而且这种方式可以实现多次例化时还可以让每个被例化的模块采用指定的实现方法，具体可参看提供例程的工程目录下\src\rgb2gray 中的仿真文件 rgb2gray_tb 中的应用。

35.5图像合并模块的设计

要实现图像以左右形式合并，首先要分析下“基于 DDR3 的串口传图帧缓存系统”是如何实现在 TFT 上显示一张图片的过程。这个过程是外部输入（串口传过来）图像数据经过数据位宽的处理后写入 DDR3 固定地址区间中，TFT 显示驱动模块根据 TFT 屏驱动时序从这个固定的地址区间取出数据实现一个个图像像素点数据在 TFT 屏上的显示，目前这个 DDR3 中固定的地址区间的范围是存储TFT显示一帧数据的内存空间。可以看作是一个深度为384000（5寸TFT的长*宽为 800*480），数据位宽为 16bit 的“RAM”（假想的一个 RAM）。写入和读出端口的地址范围均为 0~383999。每次写入（或读取）一个数据，写入端口地址（或读端口地址）加 1，地址达到最大值 383999 后又重新回到 0 开始往上加 1，循环往复。也就说 TFT 显示控制模块总是从读端的地址 0 开始读取数

据，每次地址加 1，在读地址达到最大值 383999 后就完成了一帧图像的显示，然后读地址又重新回到0开始下一帧图像的显示。地址总是顺序的，这样在TFT屏显示的图像就取决于这个“RAM”地址从0~383999内存存储的数据。那么这样 TFT 显示的图像数据取决于写入端写入的图像数据，如果写入的图像数据是一张完整的 800*480 图像数据，那么显示的就是一张 800*480 的图像，如果写入的图像数据是两张 400*480 的图像数据，那么显示的就是一张混合后拼接成的 800*480 的图像。

那么如何实现左右形式的图像拼接显示呢？先写入第一张 400*480 的图像数据，然后再写入第二张 400*480 的图像数据？并不是，要知道如何去写入这两张图片数据，需要知道 TFT 显示图片的扫描方式，这个在前面 TFT 显示驱动章节已经讲过，扫描方式是一行一行的，也就是这个缓存图片数据的“RAM”存储图片数据的内容结构如下图所示，地址 0~799 存储的是第 1 行图像数据，地址 800~1599 存储的是第 2 行图像数据……，地址 383200~383999 存储的是第 480 行图像数据。

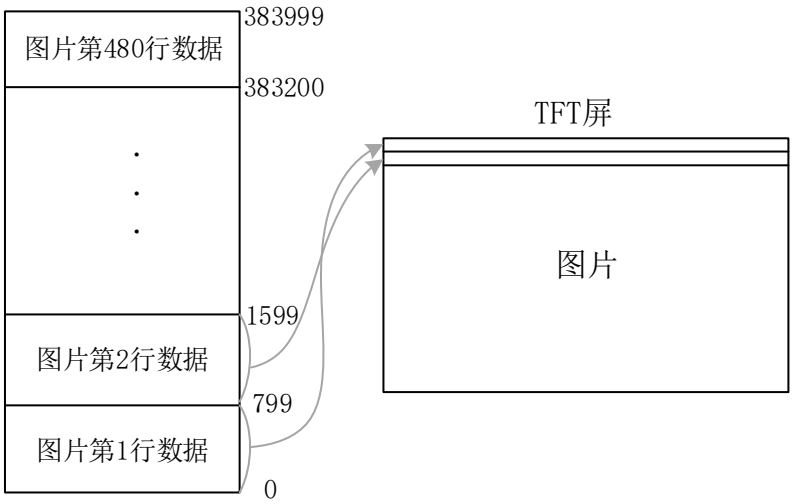


图 35-7 图片数据存储和显示对应关系 1

如果要实现左右形式的图片拼接显示，就需要往“RAM”中交错写的写入存储两张 400*480 的图片数据，左右形式拼接的“RAM”中存储的数据与 TFT 屏显示的对应关系如下图所示。从图中可以看出，存储两张 400*480 图片的方式是，先存储左边图像（图片 1）的一行数据，然后再存储右边图像（图片 2）的一行数据，直到两张图像的 480 行数据均存储完成。

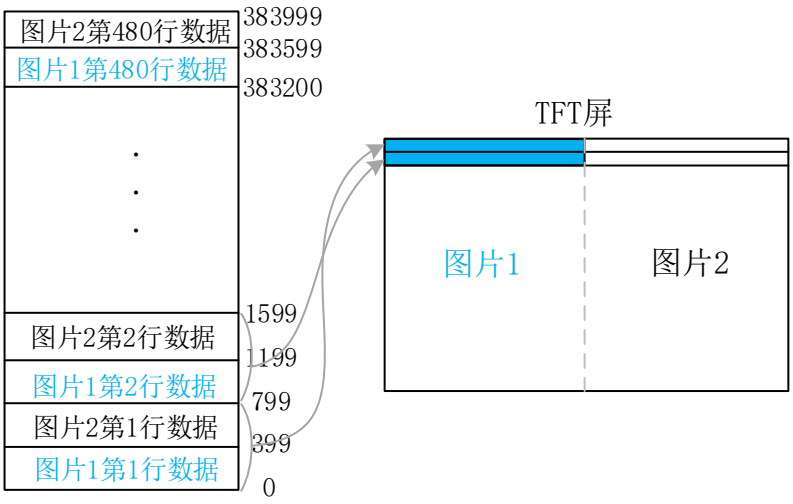


图 35-8 图片存储和显示对应关系 2

想要实现上面存储方式，考虑到两张图片可能是同时从其他模块进入，但由于图片 1 和图片 2 的存储到“RAM”有先后顺序，避免在存储其中一张图片时，另一张图片数据的丢失，需要在存储到“RAM”前为图片 1 和图片 2 的数据各加一个 fifo 就行少量数据的缓存。图像合并模块的整体设计框架如下图所示。图像合并模块两个图片输入端口，分别作为为图片 1 和图片 2 数据流的输入。通过逻辑控制交替输出两张图片的行数据给到下一级存储模块，到达实现图片合并的显示效果。

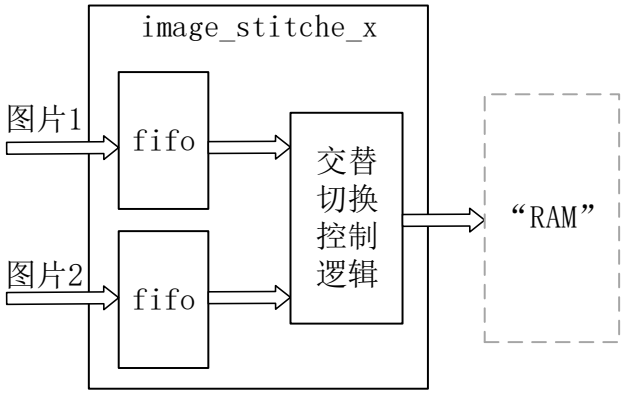


图 35-9 图片合并模块的用途示意

所谓的交替，就是先读取图片 1 的 FIFO 的 400 个数据进行输出，读取完之后，切换到读取图片 2 的 FIFO 的 400 个数据数据进行输出，同样读取完之后，切换到读取图片 1 的 FIFO 的 400 个数据数据进行输出，如此反复进行这样的切换操作。（这里需要注意的是每次读取的数据个数是图片 1/图片 2 的一行数据的像素点个数，这里图片 1/图片 2 均为 400*480 大小图片，所以每次读取 400 个数据）

根据上面的设计思路，采用状态机进行设计，画出状态转移图如下图 35-10 所示。

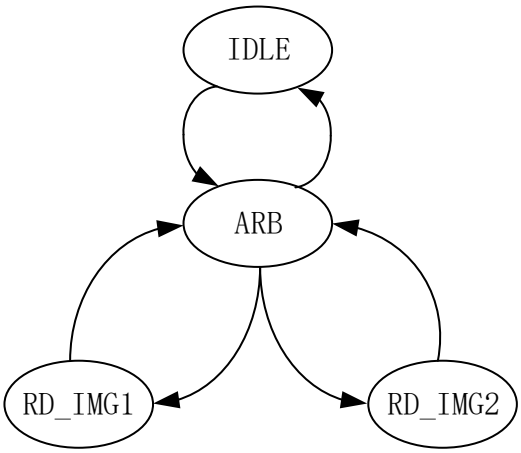


图 35-10 图像合并模块的状态转移图

- IDLE：初始状态，上电或复位后处于该状态。
- ARB：仲裁状态，根据当前的情况决定下一状态跳转到哪个状态。如果跳转到 ARB 的上一状态是读取图片 1 的 FIFO 数据状态 RD_IMG1，则下一状态跳转到读取图片 2 的 FIFO 数据状态 RD_IMG2；否则如果跳转到 ARB 的上一状态是读取图片 2 的 FIFO 数据状态 RD_IMG2，则下一状态跳转到读取图片 1 的 FIFO 数据状态 RD_IMG1，读取 FIFO 的交替切换就是本状态实现。
- RD_IMG1：读取图片 1 的 FIFO 数据状态，在一行图像数据（400 个数据）读取完成后，状态跳转到 ARB 状态。
- RD_IMG2：读取图片 2 的 FIFO 数据状态，在一行图像数据（400 个数据）读取完成后，状态跳转到 ARB 状态。

状态机具体实现代码如下：

```
//*********************************
//State Machine
//*********************************
always@(posedge clk_image_out or posedge reset_p)
begin
if(reset_p)
    curr_state <= S_IDLE;
else
    curr_state <= next_state;
end
```

```
always@(*)
begin
case(curr_state)
    S_IDLE:
    begin
        next_state = S_ARB;

    S_ARB:
    begin
    if((rd_image_sel == 1'b0) && (image1_buf_empty == 1'b0))
        next_state = S_RD_IMG1;
    else if((rd_image_sel == 1'b1) && (image2_buf_empty == 1'b0))
        next_state = S_RD_IMG2;
    else
        next_state = S_ARB;
    end

    S_RD_IMG1:
    begin
    if(image1_buf_rden && (rd_data_cnt == IMAGE1_WIDTH_IN - 1'b1))
        next_state = S_ARB;
    else
        next_state = S_RD_IMG1;
    end

    S_RD_IMG2:
    begin
    if(image2_buf_rden && (rd_data_cnt == IMAGE2_WIDTH_IN - 1'b1))
        next_state = S_ARB;
    else
        next_state = S_RD_IMG2;
    end

    default: next_state = S_IDLE;
endcase
end
```

RD_IMG1 和 RD_IMG2 状态中的 rd_image_sel 为读 FIFO 的切换选择标识信号，每读完图片 1（或图片 2）FIFO 的一行图片数据（400 个），信号 rd_image_sel 电平就变化一次。具体代码如下：

```
always@(posedge clk_image_out or posedge reset_p)
begin
 if(reset_p)
    rd_image_sel <= 1'b0;
 else if((curr_state==S_RD_IMG1)&&(rd_data_cnt==IMAGE1_WIDTH_IN-1'b1))
    rd_image_sel <= 1'b1;
 else if((curr_state==S_RD_IMG2)&&(rd_data_cnt==IMAGE2_WIDTH_IN-1'b1))
```

```
    rd_image_sel <= 1'b0;
 else
    rd_image_sel <= rd_image_sel;
end
```

不同的 rd_image_sel 信号电平标识的是需要读取哪个 FIFO 的图片数据，rd_image_sel 为 0 表示需要读取图片 1 的 FIFO 的数据，rd_image_sel 为 1 表示需要读取图片 2 的 FIFO 的数据。

代码 rd_data_cnt 表示读取 FIFO 的数据个数计数，在读取 FIFO 状态（RD_IMG1 或 RD_IMG2 状态），每读取一个 FIFO 数据，计数器加 1，具体代码如下：

```
always@(posedge clk_image_out or posedge reset_p)
begin
if(reset_p)
    rd_data_cnt <= 'd0;
else if(curr_state == S_RD_IMG1)
begin
    if(image1_buf_rden == 1'b1)
    rd_data_cnt <= rd_data_cnt + 1'b1;
    else
    rd_data_cnt <= rd_data_cnt;
end
else if(curr_state == S_RD_IMG2)
begin
    if(image2_buf_rden == 1'b1)
    rd_data_cnt <= rd_data_cnt + 1'b1;
    else
    rd_data_cnt <= rd_data_cnt;
end
else
    rd_data_cnt <= 'd0;
end
```

读 FIFO 信号 image1_buf_rden 和 image2_buf_rden 分别根据当前所处状态和 FIFO 的空满标识控制产生，直接使用组合逻辑产生，具体代码如下：

```
assign image1_buf_rden = (curr_state == S_RD_IMG1) & (image1_buf_empty
== 1'b0) & (data_out_ready_i == 1'b0);
assign image2_buf_rden = (curr_state == S_RD_IMG2) & (image2_buf_empty
== 1'b0) & (data_out_ready_i == 1'b0);
```

data_out_ready_i 信号表示的是下游其他模块的入口 FIFO 的将满信号，主要是考虑到下游模块的入口 FIFO 如果要满的情况下，是不去读本模块的 FIFO 数据的，因为读取的本模块的 FIFO 数据是要输出给下游的，如果下游 FIFO 满了，还往下游传数据会导致数据的丢失。

合并后的输出信号就相对简单，将读取的 FIFO 的数据直接输出即可，考虑读 FIFO 的读使能与读取的数据有效之间有一个周期的延时，所以读出的数据需要在读使能延迟一拍后获取数据，具体代码如下：

```
always@(posedge clk_image_out or posedge reset_p)
begin
if(reset_p)
begin
    image1_buf_rden_dly1 <= 1'b0;
    image2_buf_rden_dly1 <= 1'b0;
end
else
begin
    image1_buf_rden_dly1 <= image1_buf_rden;
    image2_buf_rden_dly1 <= image2_buf_rden;
end
end
always@(posedge clk_image_out or posedge reset_p)
begin
if(reset_p)
    data_valid_o <= 1'b0;
else if(image1_buf_rden_dly1 | image2_buf_rden_dly1)
    data_valid_o <= 1'b1;
else
    data_valid_o <= 1'b0;
end

always@(posedge clk_image_out or posedge reset_p)
begin
if(reset_p)
    data_pixel_o <= 'd0;
else if(image1_buf_rden_dly1)
    data_pixel_o <= image1_buf_dout;
else if(image2_buf_rden_dly1)
    data_pixel_o <= image2_buf_dout;
else
    data_pixel_o <= 'd0;
end
```

考虑到模块的可移植性和可扩展性将图片的尺寸或数据位宽使用参数化表示，输入的两张图片的大小和合并输出的图片的大小尺寸均用参数化表示，具体代码如下：

```
module image_stitche_x
#(
  parameter DATA_WIDTH        = 16,  //16 or 24
  //image1_in: 400*480
```

```
  parameter IMAGE1_WIDTH_IN   = 400,
  parameter IMAGE1_HEIGHT_IN  = 480,
  //image2_in: 400*480
  parameter IMAGE2_WIDTH_IN   = 400,
  parameter IMAGE2_HEIGHT_IN  = 480,
  //image_out: 800*480
  parameter IMAGE_WIDTH_OUT   = 800,
  parameter IMAGE_HEIGHT_OUT  = 480
)
(
  input                       clk_image1_in      ,
  input                       clk_image2_in      ,
  input                       clk_image_out      ,
  input                       reset_p            ,

  output                      rst_busy_o         ,
  output                      image_in_ready_o   ,

  input     [DATA_WIDTH-1:0] image1_data_pixel_i,
  input                       image1_data_valid_i,

  input     [DATA_WIDTH-1:0] image2_data_pixel_i,
  input                       image2_data_valid_i,

  input                       data_out_ready_i   ,
  output reg[DATA_WIDTH-1:0] data_pixel_o       ,
  output reg                  data_valid_o
);
```

考虑到用户在例化使用模块时对参数进行重定义时可能对出现由于笔误或其他不小心的错误使得图片的尺寸设置出现，输入的两张图片大小的尺寸参数的宽度之和不等于输出图片大小尺寸，或者输入图片的高度与输出图片的高度不一致问题，导致最终无法达到模块的功能而不便于定位错误。本模块与“彩色图像灰度化处理模块”设计采用类似的方式，使用 generate-if 的形式对错误参数的设置输出固定的数据，或者采取某种固定输出，这样便于根据实际仿真和上板的现象快速定位是否为参数配置错误导致。具体代码如下。

```
generate
  if ((IMAGE1_WIDTH_IN + IMAGE2_WIDTH_IN != IMAGE_WIDTH_OUT) ||
(IMAGE1_HEIGHT_IN != IMAGE_HEIGHT_OUT) || (IMAGE2_HEIGHT_IN !=
IMAGE_HEIGHT_OUT))
  begin: error_set_pro   //错误设置输入/输出图片参数情况的处理
//-------------------------------------------------------
    always@(posedge clk_image_out or posedge reset_p)
    begin
      if(reset_p)
```

```
        data_pixel_o <= 'd0;
      else
        data_pixel_o <= {DATA_WIDTH{1'b1}};
    end

    always@(posedge clk_image_out or posedge reset_p)
    begin
      if(reset_p)
        data_valid_o <= 'd0;
      else
        data_valid_o <= 1'b1;
    end
//------------------------------------------------------------
  end
  else
  begin: correct_set_pro    //正确设置输入/输出图片参数情况的处理
 //------------------------------------------------------------
........//正确参数配置情况下的代码，也就文档前面分析设计的代码
//------------------------------------------------------------
  end
endgenerate
```

至此，图像左右合并模块的设计基本完成，完整代码和仿真的 tb 文件参见提供的工程文件。仿真波形如下，图片 1 通道写入了两行（每行 50 个）数据，数据分别为 0~49 和 100~149。

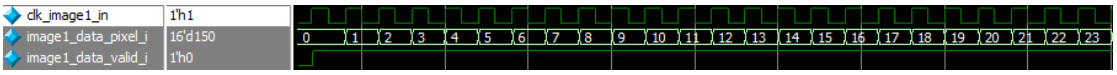


图 35-11 图片 1 通道写入数据仿真波形 1

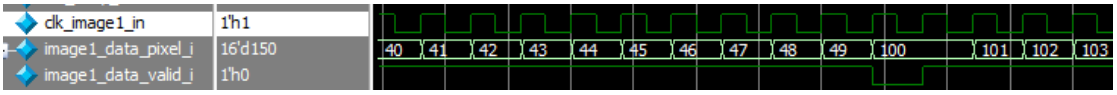


图 35-12 图片 1 通道写入数据仿真波形 2

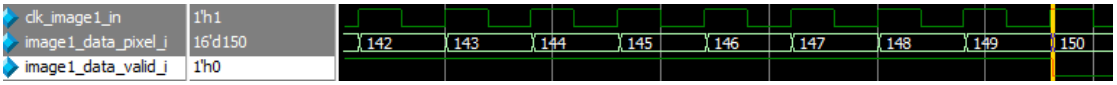


图 35-13 图片 1 通道写入数据仿真波形 3

之后，图片 2 通道写入两行（每行 50 个数据），数据分别为 50~99 和 150~199；

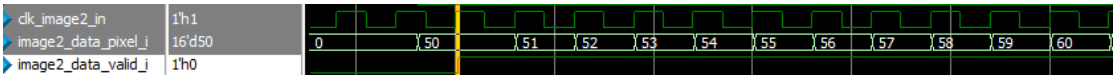


图 35-14 图片 2 通道写入数据仿真波形 1

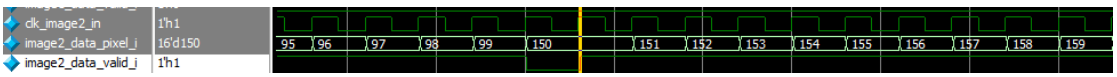


图 35-15 图片 2 通道写入数据仿真波形 2

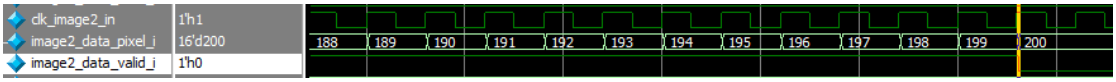


图 35-16 图片 2 通道写入数据仿真波形 3

理论上，合并之后的输出数据应该为 0~199 的 200 个数据，可以观察仿真波形，仿真结果与期望一致。

图 35-17 输出数据仿真波形 1

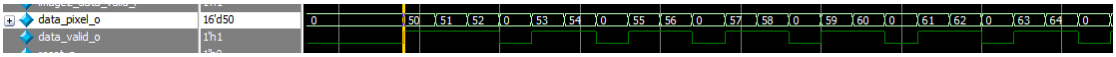


图 35-18 输出数据仿真波形 2

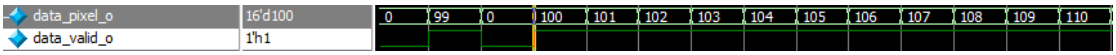


图 35-19 输出数据仿真波形 3

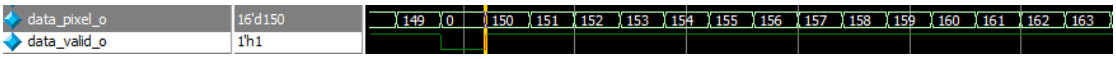


图 35-20 输出数据仿真波形 4

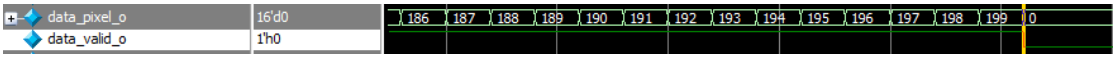


图 35-21 输出数据仿真波形 5

35.6系统板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。由于串口发送的图片数据以及在 TFT 屏上显示的图片数据是 16bit 的 RGB565 数据，而彩色图像灰度化处理是对 24bit 的 RGB888 数据进行转换，这里就涉及到 RGB565->RGB888 的转换和 RGB888->RGB565 的转换。以下转换关系是参考于网上。

```
24bit RGB888 -> 16bit RGB565 的转换
24ibt RGB888 R7 R6 R5 R4 R3 R2 R1 R0 G7 G6 G5 G4 G3 G2 G1 G0 B7 B6 B5
B4 B3 B2 B1 B0
16bit RGB656 R7 R6 R5 R4 R3 G7 G6 G5 G4 G3 G2 B7 B6 B5 B4 B3
从 8bit 到 5bit 或 6bit，取原 8bit 的高位，数据位上做了压缩，却损失了精度。
16bit RGB565 -> 24bit RGB888 的转换
16bit RGB656 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B4 B3 B2 B1 B0
24ibt RGB888 R4 R3 R2 R1 R0 R2 R1 R0 G5 G4 G3 G2 G1 G0 G1 G0 B4 B3 B2
B1 B0 B2 B1 B0
从 5bit 或 6bit 到 8bit，取原 5bit 或 6bit 的低 3 位或低 2 位做补全成 8 位
```

两种转换关系在顶层中均有用到，具体代码参见提供的工程源码文件。

顶层的仿真与“基于 DDR3 的串口传图帧缓存系统”类似，这里就不做详细讲解，读者自己建立仿真文件完成顶层的仿真。

35.6.1彩色图像灰度化工程的 TFT 显示

在顶层涉及分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。

由于本工程设计管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成数据流文件。上板调试硬件平台基于高云开发板，使用一根数据线，一端接入高云开发板的 UART 接口，另一端接 PC 机的 USB 接口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接示意图与上一章节一样。

硬件连接好并上电后然后下载生成的 bit 文件。下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态，如下所示。

图 35-22 下载程序完成后出现碎花屏效果

出现这种花屏是因为这个时候，DDR3 中并未写入数据，显示的数据是不可知的一些数据。LED0 和 LED1 分别表示时钟锁相环的 locked 信号和 DDR 初始化校准完成信号的状态，亮表示这个两个信号均变为高电平，说明 DDR 已经正常完成初始化和校准操作。接下来通过小梅哥串口传图工具向 FPGA 传输图片数据。小梅哥串口传图工具可在论坛下载。

图 35-23 启动串口传图工具

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径；图片宽度设置成与显示屏分辨率宽度一半，高度设置成与显示屏分辨率高度一致（TFT5.0 寸屏是 800*480，上位机上宽度设置为 400，高度设置为 480；TFT4.3 寸屏是 480*472，上位机上宽度设置为 240，高度设置为 272）。下图 35-24 是待传的图片。

图 35-24 准备下载的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24 的 bmp 格式图片，图片的宽度和高度需要 400*480（插 5 寸屏情况下）或 240*272（插 4.3 寸屏情况下）。

图 35-25 准备下载的图片像素信息

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用 COM6），设置好传图工具后，点击“连接设备”。

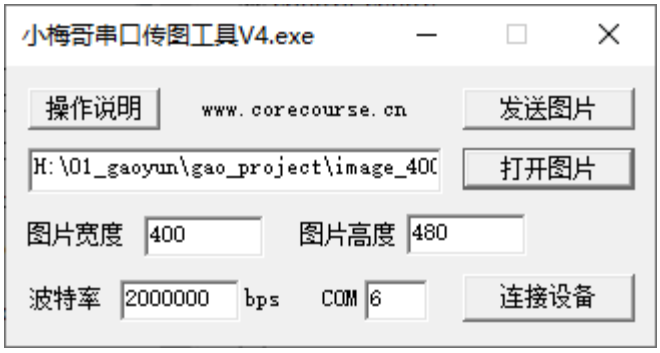


图 35-26 连接开发板和软件

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

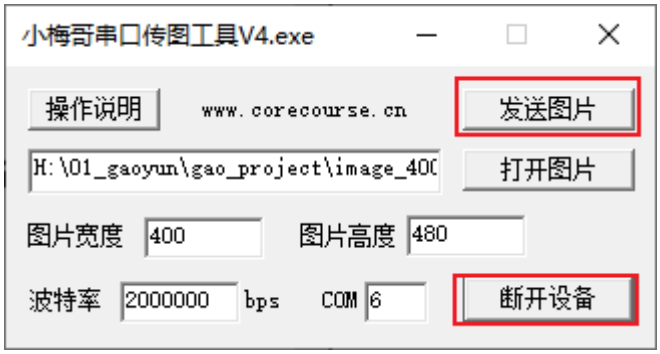


图 35-27 发送图片

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下图 35-28。

图 35-28 图片灰度化处理前后效果（TFT）

通过对比可以看出，TFT 屏上左右两边分别显示的是原始的彩色图像和灰度化之后的灰度图像。可通过改变在顶层例化彩色图像灰度化模块使用的方式观察不同的实现方方法下的实现效果。

35.6.2彩色图像灰度化工程添加 HDMI 显示

结合前面讲解的 HDMI 章节的知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。关于本小节对工程输出端口改造的方法，可参考串口传图章节对应：串口传图工程添加 HDMI 显示的内容，这里就不再重复讲解了。

最终，添加 HDMI 工程显示效果如下：

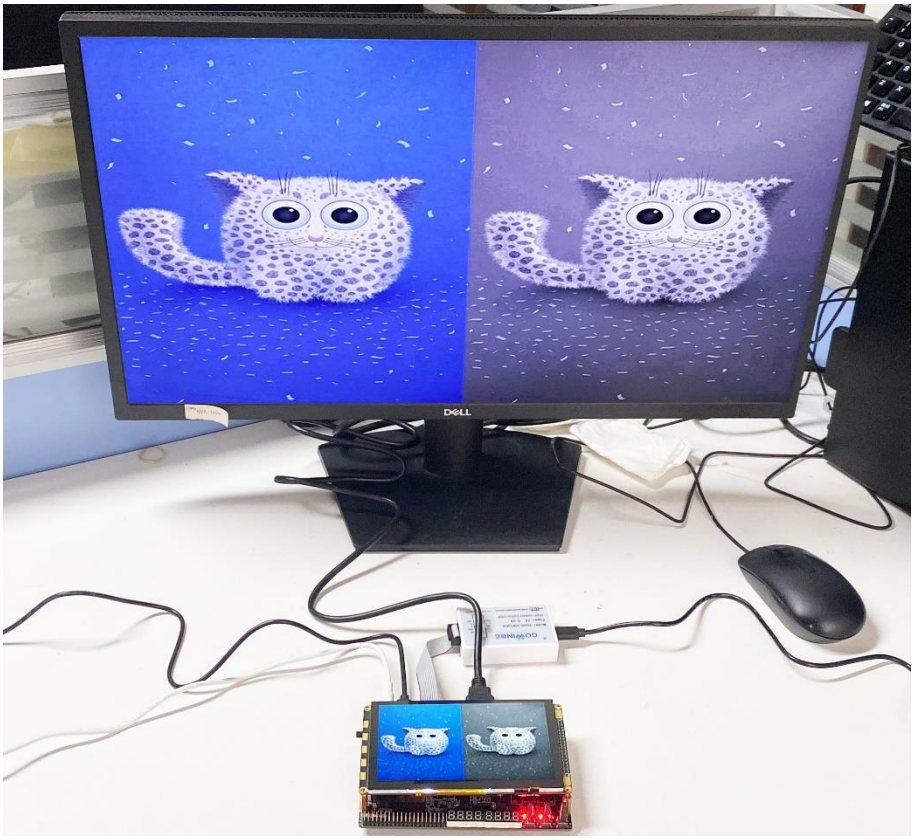


图 35-29 图片灰度化处理前后效果（HDMI）

至此，彩色图像灰度化系统在高云开发板上的设计及验证就成功完成了。

35.7总结

实现彩色图像灰度化，是实现后续图像滤波的基础。经过灰度化处理后的图像，能够有效减少图像上的非关注信息存储容量。这对于后期图像的处理和识别，是有很大帮助的。

36 灰度图像中值滤波设计实现(HDMI 和 TFT 显示)

工程源码	|----02_设计实例 |----ch36_uart_ddr3_tft_hdmi_median_filter
相关视频课程	
说明	

章节导读

本章节将基于上一章节的内容，讲述灰度图像的中值滤波算法实现。

将灰度图像进行中值滤波处理的功能是实现图像的增强和复原。中值滤波可以有效将图像中孤立噪声点、椒盐噪声点杂色滤除，是图像的内容得到修复和完善。中值滤波算法的核心在于实现像素矩阵数据的大小排序。如何快速而准确的实现像素数值大小排序是评判该中值滤波算法模块优劣的重要标准。

本章节主要是在上一节的基础上，加入中值滤波图像处理模块，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，同时进行中值滤波的处理，然后将中值滤波处理前和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏和 HDMI 显示器上同时显示中值滤波处理前的灰度图像和处理后的灰度图像。

36.1系统整体设计

在本节内容中，基于灰度图像的中值滤波，我们希望得到如下实验效果。最终在 TFT 屏和 HDMI 显示器显示的要求如下。

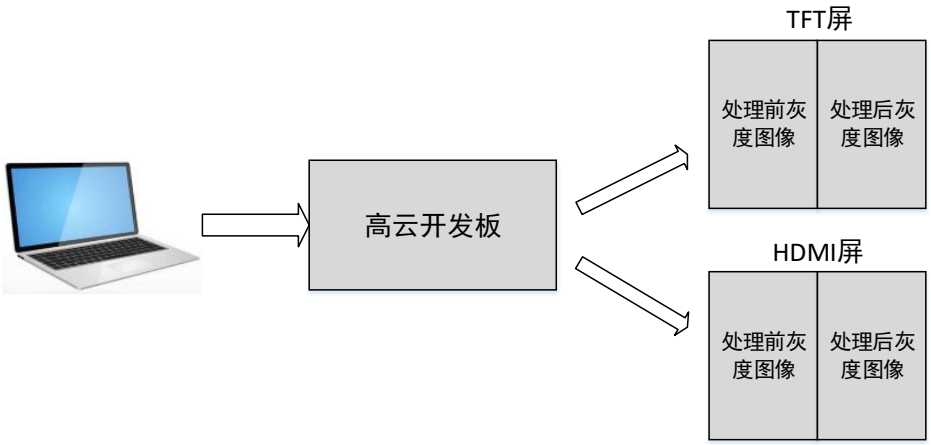


图 36-1 灰度图像中值滤波实验目标

系统整体设计框图如下图 36-2。大体结构与“彩色图像灰度化的设计实现基本一致”，增加了中值滤波处理模块。

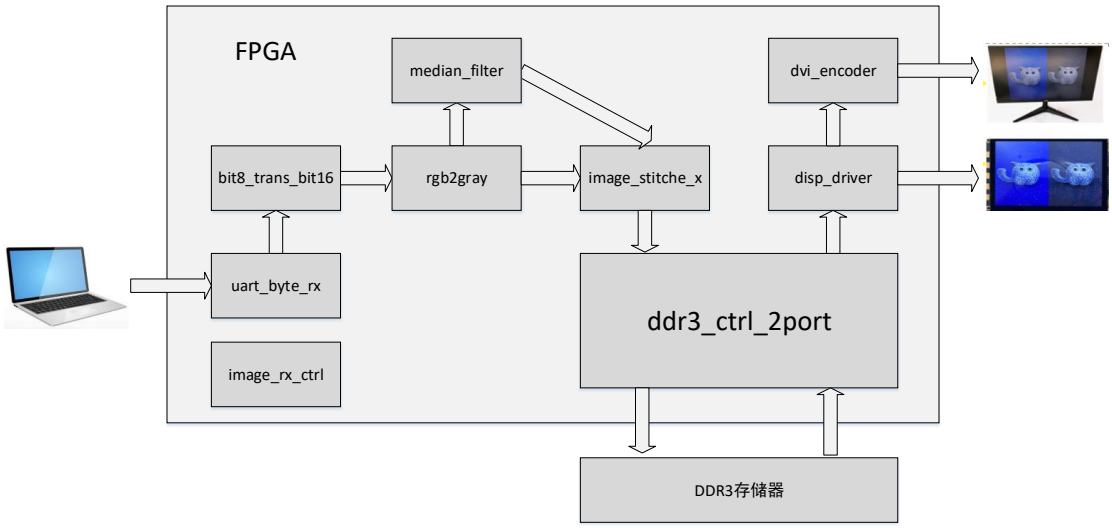


图 36-2 系统设计框图

其中，

（1）uart_byte_rx 模块：负责串口图像数据的接收，该模块的设计前面有讲，但是需要注意的是，前面在讲解串口接收时，为了避免出现非常强的电磁干扰，导致采集的信号被干扰而使得结果出错，我们就将每一位数据平均分成 16 小段，取中间比较稳定的部分作为最终的结果输出，但是在串口传图相关实验中，不必考虑这种干扰，直接将每一位接收到的数据输出即可，这样将会减少传输数据的时间，考虑到传输时间，本次设计我们将波特率设置为 2000000bps。

（2）bit8_trans_bit16 模块：将串口接收的每两个 8bit 数据转换成一个 16bit 数据（图像数据是 16bit 的 RGB565 的数据，电脑是通过串口将一个像素点数据分两次发送到 FPGA，FPGA 需将串口接收数据重组成 16bit 的图像数据），实现过程相对比较简单。

（3）rgb2gray 模块：彩色图像灰度化处理，对串口接收的彩色图像数据实时进行灰度化处理。

（4）median_filter 模块：对彩色图像进行灰度化处理后的数据进行中值滤波处理。

（5）image_stitche_x 模块：将串口接收的尺寸为 400*480 大小的彩色图像与灰度化处理后的 400*480 大小的图像数据以左右形式合并成一张 800*480 的图像。

（6）disp_driver 模块：TFT 屏显示驱动控制，对缓存在 DDR3 中的图像数据进行显示。

（7）ddr3_ctrl_2port 模块组：包含 wr_ddr3_fifo、rd_ddr3_fifo、fifo_ddr3_adapter 以及 DDR3_Memory_Interface_Top 模块，该模块用于控制 ddr3 存储器完成采集的图像数据缓存。

（8）pll 模块：上述各个模块所需时钟的产生，使用 PLL IP。

（9）image_rx_ctrl 模块。用于显示接收进度，给出接收完成标志。

（10）dvi_encoder 模块：用于将生成信号转换成 HDMI 输出信号作 HDMI 显示。

本系统就是在上一系统基础上添加图像处理模块搭建系统。除去使用 IP 和前面章节讲过的模块外，只需要设计中值滤波模块。

36.2灰度图像中值滤波处理模块的设计

36.2.1基本原理

中值滤波法是一种非线性平滑技术，它将每一像素点的灰度值设置为该点某邻域窗口内的所有像素点灰度值的中值。

中值滤波是基于排序统计理论的一种能有效抑制噪声的非线性信号处理技术，中值滤波的基本原理是把数字图像或数字序列中一点的值用该点的一个邻域中各点值的中值代替，让周围的像素值接近真实值，从而消除孤立的噪声点。方法是用某种结构的二维滑动模板，将板内像素按照像素值的大小进行排序，生成单调上升（或下降）的二维数据序列。二维中值滤波输出为 g（x,y）=mid{f(x-k,y-l),(k,l∈W)}，其中 f(x,y)，g(x,y)分别为原始图像和处理后图像。W 为二维模板，通常为 3*3，5*5 区域，也可以是不同的的形状，如线状，圆形，十字形，圆环形等。

中值滤波法对消除椒盐噪声非常有效，在光学测量条纹图像的相位分析处理方法中有特殊作用，但在条纹中心分析方法中作用不大。中值滤波在图像处理中，常用于保护边缘信息，是经典的平滑噪声的方法。

要得到模板中数据的之间值，首先要将数据按大小排序，然后根据有序的数字序列来找中间值。中值滤波排序的过程中有很多成熟的算法。如冒泡排序、二分排序等，大多基于微机平台的软件算法，而适合硬件平台的排序算法则比

较少。

36.2.2FPGA 中值滤波实现算法简介

下面介绍的即使通过硬件平台实现 FPGA 中值滤波排序算法：

为便于说明实现步骤，滤波的像素矩阵模板取 3*3 大小的九宫格，矩阵第二行第二列的值 L（2，2）即为目标处理像素点，其邻域的 8 个点，即为其周围的像素值，用以辅助 L（2，2）实现中值滤波运算。

表 36-1FPGA 中值滤波算法实现

L(1,1)	L(1,2)	L(1,3)
L(2,1)	L(2,2)	L(2,3)
L(3,1)	L(3,2)	L(3,3)

如上表 36-1 所示，为 3*3 的图像模板。

第一步：

分别对三行像素进行排序：

由 L11，L12，L13 得到 L1max，L1mid，L1min；

由 L21，L22，L23 得到 L2max，L2mid，L2min；

由 L31，L32，L33 得到 L3max，L3mid，L3min。

第二步：

分别对三行像素中的 3 个最大，3 个中间和 3 个最小分别进行排序：

由 L1max，L2max，L3max 得到 Lmax_max，Lmax_mid，Lmax_min；

由 L1mid，L2mid，L3mid 得到 Lmid_max，Lmid_mid，Lmid_min；

由 Lmin，L2min，L3min 得到 Lmin_max，Lmin_mid，Lmin_min；

第三步：

对最大的最小(Lmax_min)，中间的中间(Lmid_mid)以及最小的最大(Lmin_max)进行排序（例：由 Lmax_min，Lmid_mid，Lmin_max 得到 median）。

FPGA 的算法实现步骤基本如此。

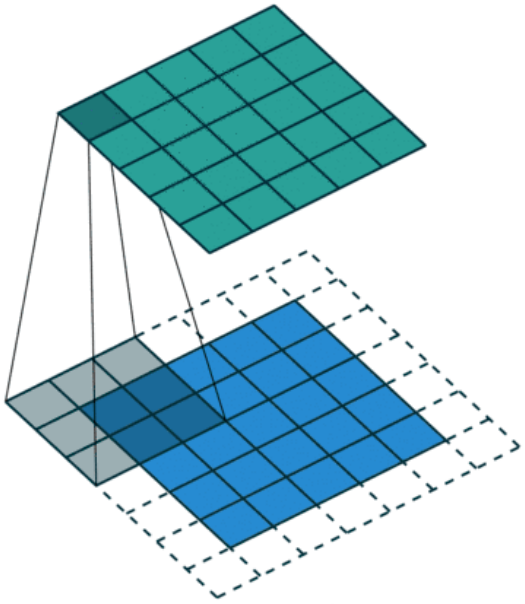


图 36-3 需计算的像素点及提取的滤波像素矩阵

为进一步了解中值滤波算法的实现步骤，我们以数值进行举例演示。假设数值的大小即为像素点亮度大小。

36.2.3FPGA 中值滤波实现算法演示举例

举例来说，某图像数据的局部，存在如下 3x3 的亮度数据矩阵如下：

表 36-2 3*3 图像数据矩阵举例

2	4	80
3	5	4
6	3	2

从上面 3x3 的数据矩阵中，不难发现，以上像素矩阵以九宫格中心数字 5 为需要处理的像素点。选取的像素矩阵中有数据 80 和周边的点有明显的颜色显示数值差异。反映到图像中，应该是一个比周边值更亮的噪声孤点。

如果采用中值滤波算法，对以上原始数据矩阵进行处理：

第一步： 首先将 9 个图像数据按行分成 3 组， 分别对三组数据进行排序。

（1）对 L11， L12， L13 进行排序得到：L1max（即 80）， L1mid（即 4），L1min（即 2）；

（2）对 L21， L22， L23 进行排序得到：L2max（即 5）， L2mid（即 4），

L2min（即 3）；

（3）对 L31， L32， L33 进行排序得到：L3max（即 6）， L3mid（即 3），L3min（即 2）。

处理后，得到一个临时过渡的矩阵，内容为：

表 36-3 图像矩阵举例第一步运算结果

80	4	2
5	4	3
6	3	2

第二步：分别对三组像素数据中的 3 个最大， 3 个中间和 3 个最小分别进行排序。

（1）对 L1max，L2max，L3max 进行排序得到：Lmaxmax（即 80），Lmaxmid（即 6），Lmaxmin（即 5）；

（2）对 L1mid， L2mid， L3mid 进行排序得到：Lmidmax（即 4），Lmidmid（即 4），Lmidmin（即 3）；

（3）对 L1min， L2min， L3min 进行排序得到：Lminmax（即 3），Lminmid（即 2），Lminmin（即 2）；

处理后，得到这样第二个临时的过渡矩阵，内容为：

表 36-4 图像矩阵举例第二步运算结果

80	6	5
4	4	3
3	2	2

第三步：对最大的最小(Lmaxmin)，中间的中间(Lmidmid)以及最小的最大(Lminmax)进行排序，排序得到的中间数据为最终的中值。

表 36-5 最终中值筛选

5（第一行末，最大的最小）	4（第二行中，中间的中间）	3（第三行首，最小的最大）

最终，得到该图像矩阵中值为 4。

由于 FPGA 的图像显示数据采用的是逐个像素点的行场扫描输出方案，因此，当有噪声信号点（如图中数据 80）参与中值滤波点阵处理后，该点是不会对像素矩阵最终输出值产生影响的，即实现了图片噪声信号滤波。

这个中值，最终作为我们滤波像素矩阵计算像素输出点安置在新的像素矩阵中，该值就是原像素矩阵中待处理的中心点数字 5 的对应行场位置输出结果。如果对整幅图像都作同样的处理，则将会得到一幅新的图像。这幅新的图像每个像素，都和原图像像素形成中值滤波运算后一一对应的关系。从存储空间的角度分析，完成中值滤波，必须开辟至少两幅图像的存储空间，第一块存储空间作为原图像的存储空间，第二块存储空间作为图像处理后缓存数据的存储空间。否则，如果将处理后的数据安置回原始的存储空间，则下一个运算点进行中值滤波计算时，取用的像素矩阵将不是原始图像的像素矩阵数值，而是带有一个已经滤波完成的像素点参与运算的像素矩阵，这样将导致所有的运算，都受到其他已运算点位的影响，从而使最终输出结果不符合设计预期。

36.2.4模块接口设计

经过分析，中值滤波算法只需要将待处理的数据输入中值滤波模块，即可得到滤波后对应的输出像素值。

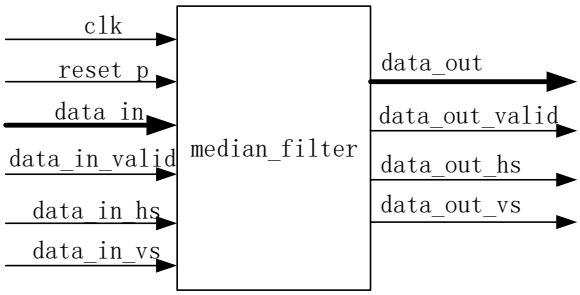


图 36-4 中值处理模块输入输出接口

表 36-6 模块端口描述表

端口名称	I/O	端口说明
clk	I	图像像素时钟
reset_p	I	模块复位信号，高电平有效
data_in	I	图像像素数据，数据位宽可自定义，对于灰度图像，位宽为 8
data_in_valid	I	图像像素数据有效标识，为 1 表示当前 data_in 有效
data_in_hs	I	图像行信号
data_in_vs	I	图像场信号
data_out	O	图像处理后数据输出，数据位宽与 data_in 相同
data_out_valid	O	图像处理后像素数据有效标识，为 1 表示当前 data_out 有效
data_out_hs	O	图像处理后行信号
data_out_vs	O	图像处理后场信号

36.2.5实现过程

通过上面的学习，我们知道图像数据是按照时钟节拍一个像数一个像数传入到 FPGA，要实现中值滤波，首先是要产生上述的 3*3 的模板，然后再按照算法步骤进行排序计算。

模块设计之初，必须对原始图像每一个像素点数据的读取和存储进行分析。要想稳定的实现滤波模板的滑动，必须保证滤波模板的输入和输出数据平衡，即：实现模块的端口为 1 各像素点的滤波前灰度值输入，1 个像素点的滤波后灰度值输出。而图像的显示扫描过程是按完整的逐行扫描行来执行，这样就导致必须至少缓存有完整的两行数据，另外加上第三行的至少 3 个原始像素值，才能运算获得第一个像素点的值输出。

换句话说：由于图像数据是一个一个输入进来的，貌似要实现 3*3 的模板，就首先必须要保证能对 3 行图像数据进行获取，这样就必须要对图像数据进行行缓存。乍一看，3*3 模板需要缓存 3 行，其实不然，缓存 2 行后，接下来输入进来的数据就是第 3 行的数据了，这样就实现了 3 行数据同时存在的情况了，对行缓存区的要求是左端进入一个数据，右端出来一个数据，这个要求与移位寄存器有些类似。

另外，对于数据缓存，还有这样一个要求：除了这些计算第一个像素点的9个数值必须保留并参与运算外，第一、第二行的其他像素点的数值也不能抛弃。这是因为，在像素矩阵滑动取值时，每个点只有一次被少扫描读取的机会，如果需要利用第一行后续点位像素值计算第二行相同位置的像素值时，那么这些值就再也读不回来了。

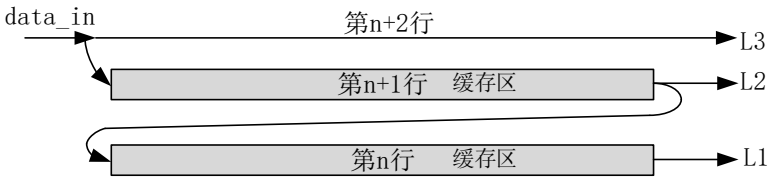


图 36-5 逐行中值滤波矩阵处理

接下来，看看如何生成移位寄存器 IP。

在 Gowin 里有一个 IP 正好可以满足我们的需求，IP 名叫 RAM Based Shift Register，可以在 IP Catalog 进行搜索。这里是对灰度图像数据进行处理，数据位宽设置为 8。由于待处理的图像数据尺寸为 400*480，所以深度设置为 400。IP 设置界面如下图 36-6。

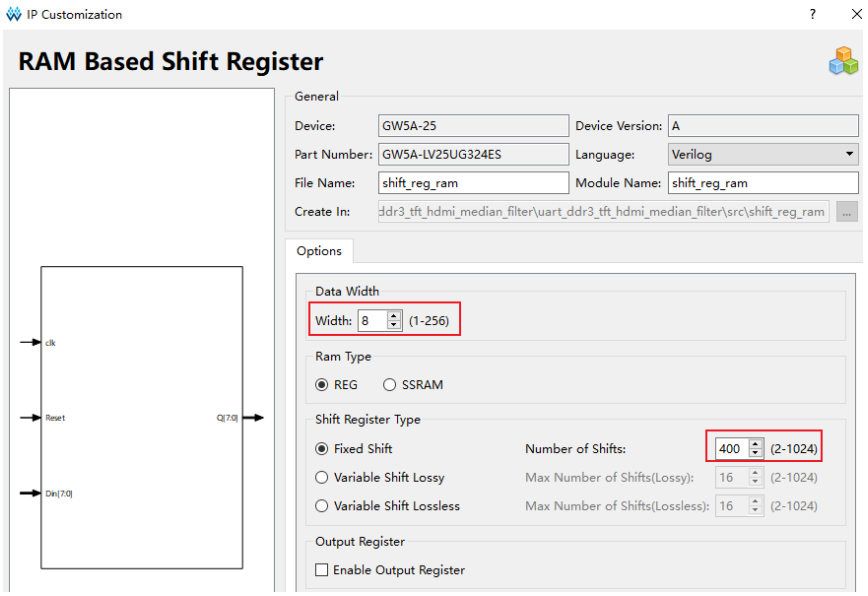


图 36-6 中值滤波处理 IP

考虑对其移位控制，对其时钟进行使能控制，使其在时钟使能位高的情况下，在进行移位，这里调用原语如下：

```
DCE DCE(
    .CLKIN(clk),
    .CE(shiftin_valid),
    .CLKOUT(clk_ce)
);
```

上述代码 clk 为时钟输入，shiftin_valid 为使能信号，只有在 shiftin_valid 有效的情况下，才会输出时钟 clk_ce，将输出的时钟 clk_ce 作为 shift_reg_ram 的输入时钟，这样最终对 IP 的工作原理如下所示。D[7:0]为输入数据，Q[7:0]输出数据，在时钟使能 CE 为高的情况下，输入端 D，在每个时钟周期输入一个数据，输出端 Q 输出一个数据，缓存在 RAM 内的数据向右移动一个位置。

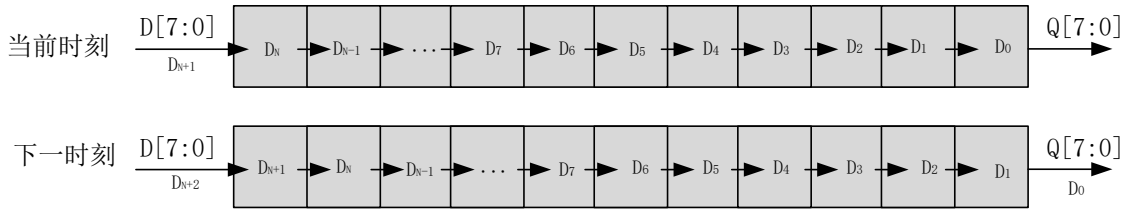


图 36-7 图像数据处理流程

要实现 2 行数据的缓存，只需要例化两个移位寄存器 IP，然后将其中一个 IP 的输出接到另一个 IP 的输入。具体代码如下。

```
module shift_register_2taps
#(
```

```
  parameter DATA_WIDTH = 8
)
(
  input                   clk,
  input                   Reset,
  input  [DATA_WIDTH-1:0] shiftin,
  input                   shiftin_valid,

  output [DATA_WIDTH-1:0] shiftout,
  output [DATA_WIDTH-1:0] taps1x,
  output [DATA_WIDTH-1:0] taps0x
);
wire clk_ce;

assign shiftout = taps0x;

DCE DCE(
    .CLKIN(clk),
    .CE(shiftin_valid),
    .CLKOUT(clk_ce)
);
shift_reg_ram shift_reg_ram_inst1(
    .clk(clk_ce), //input clk
    .Reset(Reset), //input Reset
    .Din(shiftin), //input [7:0] Din
    .Q(taps1x) //output [7:0] Q
);
shift_reg_ram shift_reg_ram_inst2(
    .clk(clk_ce), //input clk
    .Reset(Reset), //input Reset
    .Din(taps1x), //input [7:0] Din
    .Q(taps0x) //output [7:0] Q
);

endmodule
```

代码中 taps0x 对应下图中 L1 输出，taps1x 对应下图中 L2 输出，shiftin 对应图中 L3 输出。

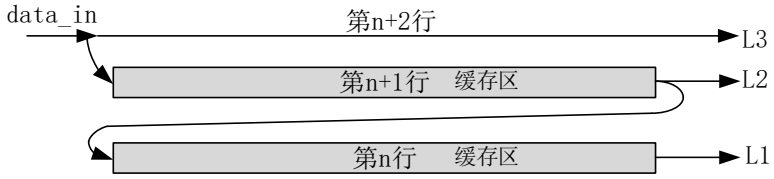


图 36-8 数据缓存示意图

编写模块仿真测试代码如下。仿真中产生了 3 组相同的数据，每组数据从 1

开始，每个周期数据加 1。

```
`timescale 1ns/1ns
`define CLK_PERIOD 40

module shift_register_2taps_tb();
  reg        clk;
  reg  [7:0] shiftin;
  reg        shiftin_valid;

  wire [7:0] shiftout;
  wire [7:0] taps1x;
  wire [7:0] taps0x;

initial clk = 1'b1;
always #(`CLK_PERIOD/2) clk = ~clk;

initial begin
  shiftin_valid = 1'b0;
  shiftin = 'd0;
  #201;
  repeat(3) begin
    shiftin = 'd0;
    repeat(400) begin
      shiftin_valid = 1'b1;
      shiftin = shiftin + 1'b1;
      #(`CLK_PERIOD);
    end
  end
  #200;
  $stop;
end

shift_register_2taps
shift_register_2taps_inst
(
  .clk           (clk          ),
  .shiftin       (shiftin      ),
  .shiftin_valid (shiftin_valid),

  .shiftout      (shiftout     ),
  .taps1x        (taps1x       ),
  .taps0x        (taps0x       )
);

endmodule
```

仿真波形如下图 36-9 所示。

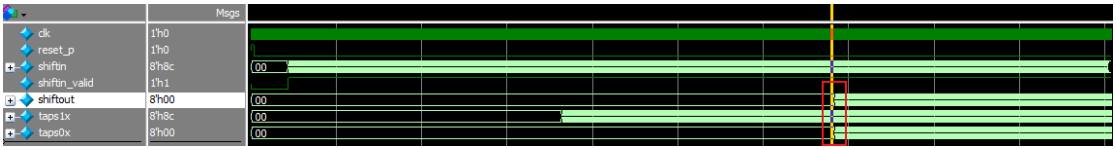


图 36-9 移位寄存模块仿真波形

对上图中红色圈注的地方进行波形放大查看如下图 36-10 所示。

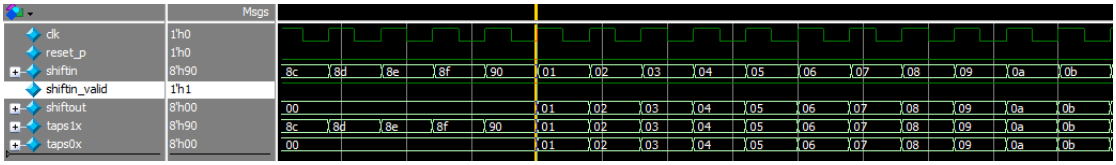


图 36-10 移位寄存模块放大信息图

从仿真波形可以看出从这个时候开始，taps0x、taps1x 和 shiftin 输出数据都是从 1 开始递增，并且数据一样，这与预期一致，因为仿真输入的是 3 行完全相同的数据，该模块实现的功能是缓存 2 行数据，这样在开始输入第 3 行数据的时候，就可以在输出同时出现 3 行相同的数据了。

上面仅仅实现了 3 行数据的同时存在，即 3*1 的模板，要实现 3*3 的模板，还需要进一步处理。输入的数据是一个接着一个的，只需要 3 组采用寄存器将数据存储就可以实现 3 列数据同时出现，如下图所示是 3*3 模板数据与寄存器之间对应关系。

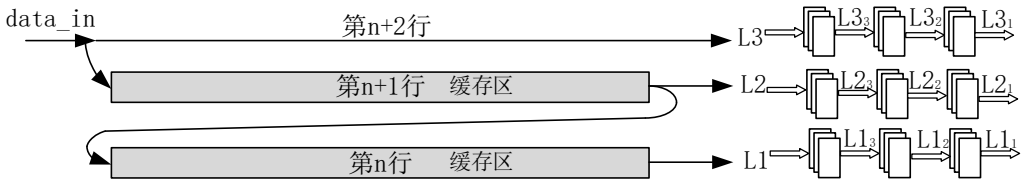


图 36-11 3*3 模板数据与寄存器之间对应关系

具体实现代码如下。

第 1 部分：定义滤波像素矩阵模板的数据缓存空间

代码中 row0_col0、row0_col1、row0_col2、row1_col0、row1_col1、row1_col2、row2_col0、row2_col1、row2_col2 分别对应 3*3 模板中的 9 个数据缓存空间。

```
//----------------------------------------------------
// matrix 3x3 data
// row0_col0   row0_col1   row0_col2
// row1_col0   row1_col1   row1_col2
// row2_col0   row2_col1   row2_col2
//----------------------------------------------------
```

```
always @(posedge clk or posedge reset_p) begin
  if(reset_p) begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
  else if(data_in_hs && data_in_vs)
    if(data_in_valid) begin
      row0_col2 <= line0_data;
      row0_col1 <= row0_col2;
      row0_col0 <= row0_col1;

      row1_col2 <= line1_data;
      row1_col1 <= row1_col2;
      row1_col0 <= row1_col1;

      row2_col2 <= line2_data;
      row2_col1 <= row2_col2;
      row2_col0 <= row2_col1;
    end
    else begin
      row0_col2 <= row0_col2;
      row0_col1 <= row0_col1;
      row0_col0 <= row0_col0;

      row1_col2 <= row1_col2;
      row1_col1 <= row1_col1;
      row1_col0 <= row1_col0;

      row2_col2 <= row2_col2;
      row2_col1 <= row2_col1;
      row2_col0 <= row2_col0;
    end
  else begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
```

```
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
end
```

第 2 部分：实现 3 个数据的排序处理。

3*3 模板中的 9 个数据准备好之后，就是按照中值滤波算法的处理步骤进行，从中值滤波的步骤中可以总结出一个通用的模块，就是对 3 个数据的排序处理。这个处理可以单独作为一个子模块来实现。实现方法比较简单，就是 3 个数据两两进行比较即可，具体实现代码如下。代码中数据位宽采用参数指定，便于移植。

```
module sort
#(
  parameter DATA_WIDTH = 8
)
(
  input                       clk,    //pixel clk
  input                       reset_p,
  input                       data_in_valid,
  input     [DATA_WIDTH-1:0] data0_in,
  input     [DATA_WIDTH-1:0] data1_in,
  input     [DATA_WIDTH-1:0] data2_in,

  output reg[DATA_WIDTH-1:0] data_max_out,
  output reg[DATA_WIDTH-1:0] data_mid_out,
  output reg[DATA_WIDTH-1:0] data_min_out,
  output reg                 data_out_valid
);

always @(posedge clk or posedge reset_p) begin
  if(reset_p) begin
    data_max_out <= 'd0;
    data_mid_out <= 'd0;
    data_min_out <= 'd0;
  end
  else if(data_in_valid) begin
    if((data0_in >= data1_in) && (data0_in >= data2_in)) begin
      data_max_out <= data0_in;

      if(data1_in >= data2_in) begin
        data_mid_out <= data1_in;
```

```
        data_min_out <= data2_in;
      end
      else begin
        data_mid_out <= data2_in;
        data_min_out <= data1_in;
      end
    end
    else if((data1_in > data0_in) && (data1_in >= data2_in)) begin
      data_max_out <= data1_in;

      if(data0_in >= data2_in) begin
        data_mid_out <= data0_in;
        data_min_out <= data2_in;
      end
      else begin
        data_mid_out <= data2_in;
        data_min_out <= data0_in;
      end
    end
    else if((data2_in > data0_in) && (data2_in > data1_in)) begin
      data_max_out <= data2_in;

      if(data0_in >= data1_in) begin
        data_mid_out <= data0_in;
        data_min_out <= data1_in;
      end
      else begin
        data_mid_out <= data1_in;
        data_min_out <= data0_in;
      end
    end
  end
end

always @(posedge clk or posedge reset_p)
  if(reset_p)
    data_out_valid <= 1'b0;
  else if(data_in_valid)
    data_out_valid <= 1'b1;
  else
    data_out_valid <= 1'b0;

endmodule
```

第 3 部分：3x3 像素矩阵排序的实现

排序模块就不再单独进行仿真，后面直接放在中值滤波模块里一起进行仿真。目前 3*3 模板数据已经有了，3 个数据的排序也有了，整个中值滤波的实现

就比较简单了，按照算法的 3 个步骤进行实现就可以比较轻松的完成，具体代码如下，实现过程中注意 data_in_valid 与 data_in 的对应关系。

```
always @(posedge clk)
begin
  data_in_valid_dly1 <= data_in_valid;
  data_in_valid_dly2 <= data_in_valid_dly1;
  data_in_valid_dly3 <= data_in_valid_dly2;

  data_in_hs_dly1    <= data_in_hs;
  data_in_hs_dly2    <= data_in_hs_dly1;
  data_in_hs_dly3    <= data_in_hs_dly2;

  data_in_vs_dly1    <= data_in_vs;
  data_in_vs_dly2    <= data_in_vs_dly1;
  data_in_vs_dly3    <= data_in_vs_dly2;
end

//----------------------------------------------------
// line0 of (max mid min)
//----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_line0
(
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly1 ),
  .data0_in      (row0_col0          ),
  .data1_in      (row0_col1          ),
  .data2_in      (row0_col2          ),

  .data_max_out  (line0_max          ),
  .data_mid_out  (line0_mid          ),
  .data_min_out  (line0_min          ),
  .data_out_valid(                   )
);

//----------------------------------------------------
// line1 of (max mid min)
//----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_line1
(
```

```
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly1 ),
  .data0_in      (row1_col0          ),
  .data1_in      (row1_col1          ),
  .data2_in      (row1_col2          ),

  .data_max_out  (line1_max          ),
  .data_mid_out  (line1_mid          ),
  .data_min_out  (line1_min          ),
  .data_out_valid(                   )
);

//-----------------------------------------------------
// line2 of (max mid min)
//-----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_line2
(
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly1 ),
  .data0_in      (row2_col0          ),
  .data1_in      (row2_col1          ),
  .data2_in      (row2_col2          ),

  .data_max_out  (line2_max          ),
  .data_mid_out  (line2_mid          ),
  .data_min_out  (line2_min          ),
  .data_out_valid(                   )
);

//-----------------------------------------------------
// max of (max mid min)
//-----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_max
(
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly2 ),
  .data0_in      (line0_max          ),
  .data1_in      (line1_max          ),
```

```
  .data2_in      (line2_max          ),

  .data_max_out  (max_max            ),
  .data_mid_out  (max_mid            ),
  .data_min_out  (max_min            ),
  .data_out_valid(                   )
);

//-----------------------------------------------------
// mid of (max mid min)
//-----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_mid
(
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly2 ),
  .data0_in      (line0_mid          ),
  .data1_in      (line1_mid          ),
  .data2_in      (line2_mid          ),

  .data_max_out  (mid_max            ),
  .data_mid_out  (mid_mid            ),
  .data_min_out  (mid_min            ),
  .data_out_valid(                   )
);

//-----------------------------------------------------
// min of (max mid min)
//-----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_min
(
  .clk           (clk                ),    //pixel clk
  .reset_p       (reset_p            ),
  .data_in_valid (data_in_valid_dly2 ),
  .data0_in      (line0_min          ),
  .data1_in      (line1_min          ),
  .data2_in      (line2_min          ),

  .data_max_out  (min_max            ),
  .data_mid_out  (min_mid            ),
  .data_min_out  (min_min            ),
```

```
  .data_out_valid(                    )
);

//----------------------------------------------------
// matrix 3x3 of mid
//----------------------------------------------------
sort
#(
  .DATA_WIDTH (DATA_WIDTH)
)sort_matrix_mid
(
  .clk           (clk                 ),    //pixel clk
  .reset_p       (reset_p             ),
  .data_in_valid (data_in_valid_dly3 ),
  .data0_in      (max_min             ),
  .data1_in      (mid_mid             ),
  .data2_in      (min_max             ),

  .data_max_out  (                    ),
  .data_mid_out  (matrix_mid          ),
  .data_min_out  (                    ),
  .data_out_valid(                    )
);

//----------------------------------------------------
//result
//----------------------------------------------------
assign data_out = matrix_mid;
```

36.2.6仿真验证

中值滤波处理模块设计完成后，编写仿真验证 testbench 文件，代码如下

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module median_filter_tb();

  reg       clk;    //pixel clk
  reg       reset_p;
  reg [7:0] data_in;
  reg       data_in_valid;
  reg       data_in_hs;
  reg       data_in_vs;
  wire[7:0] data_out;
  wire      data_out_valid;
  wire      data_out_hs;
  wire      data_out_vs;
```

```
initial clk = 1'b1;
always #(`CLK_PERIOD/2) clk = ~clk;

initial begin
  reset_p = 1'b1;
  data_in = 8'd0;
  data_in_valid = 1'b0;
  data_in_hs = 1'b0;
  data_in_vs = 1'b0;
  #201;
  reset_p = 1'b0;
  #200;

  data_in_vs = 1'b1;
  repeat(480) begin
    #500;
    data_in_hs = 1'b1;
    #500;
    repeat(400*2) begin
      data_in_valid = ~data_in_valid;
      data_in = $random % 256;
      #(`CLK_PERIOD);
    end
    #500;
    data_in_hs = 1'b0;
  end
  data_in_vs = 1'b0;
  #(`CLK_PERIOD);
  data_in_vs = 1'b1;

  #2000;
  $stop;
end

median_filter
#(
  .DATA_WIDTH ( 8 )
)median_filter
(
  .clk            (clk           ),    //pixel clk
  .reset_p        (reset_p       ),
  .data_in        (data_in       ),
  .data_in_valid  (data_in_valid ),
  .data_in_hs     (data_in_hs    ),
  .data_in_vs     (data_in_vs    ),
```

```
  .data_out       (data_out      ),
  .data_out_valid (data_out_valid),
  .data_out_hs    (data_out_hs   ),
  .data_out_vs    (data_out_vs   )
);

endmodule
```

在仿真波形中任意找几个数据查看算法计算的对错，下面波形分别是对每行 3 个数据进行排序计算最大值，中间值，最小值。可以看出算法计算符合预期。同理可以查看波形中计算的 max_max、max_mid、max_min、mid_max、mid_mid、mid_min、min_max、min_mid、min_min 计算结果是否符合预期。最后看计算出来的中间值是否符合预期。

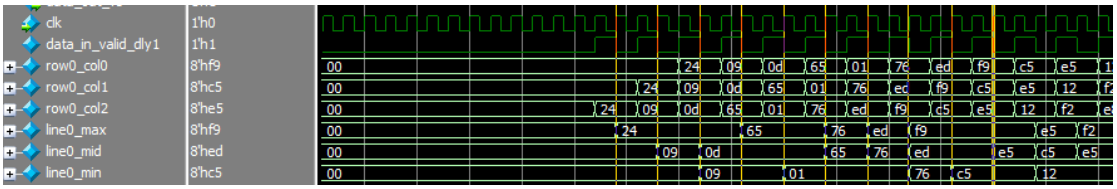


图 36-12 仿真波形图 1

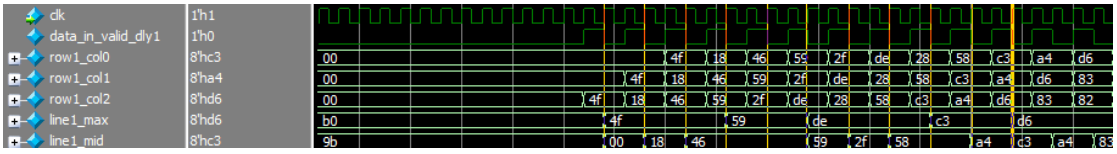


图 36-13 仿真波形图 2

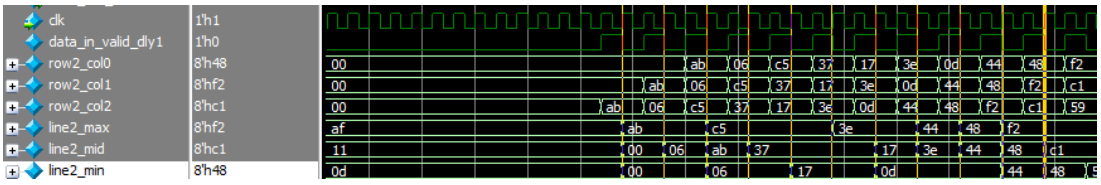


图 36-14 仿真波形图 3

（1）任意取一处位置的 3*3 模板数据

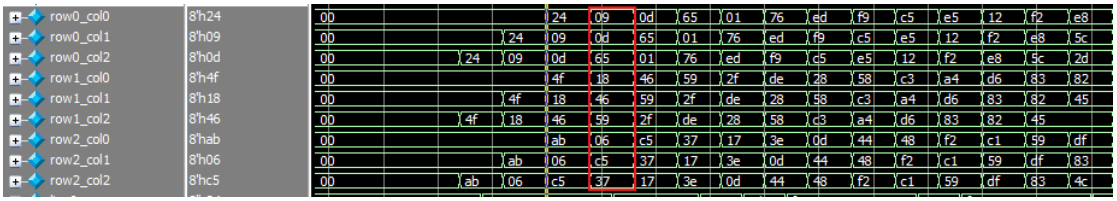


图 36-15 仿真 3*3 模板数据

（2）计算的每行 3 个数据的排序后的结果

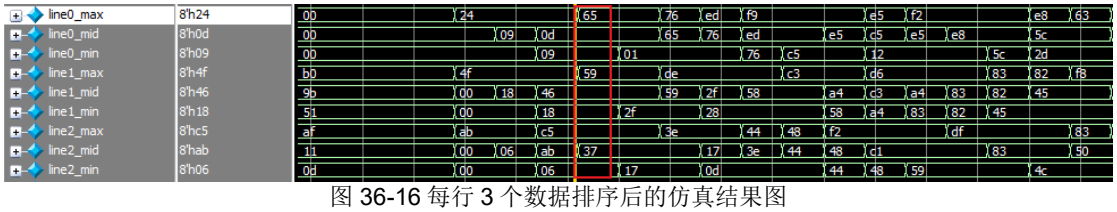


图 36-16 每行 3 个数据排序后的仿真结果图

（3）分别对 3 行数据中最大值、中间值、最小值的排序后的结果

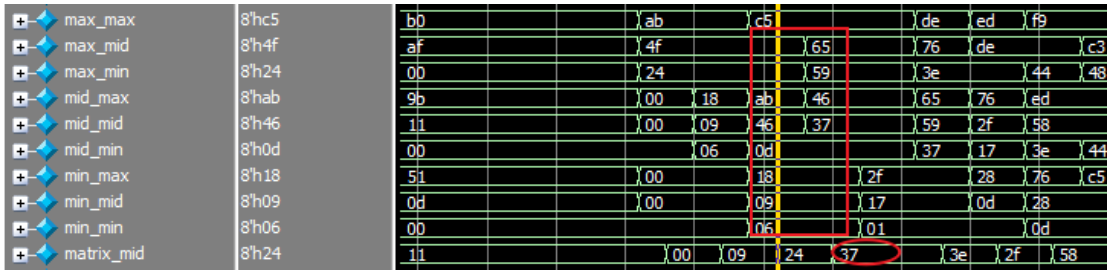


图 36-17 最大、中间、最小值的排序结果

（4）从上面波形数据可以看出，max_min 为 0x59，mid_mid 为 0x37，min_max 为 0x18，根据算法的最后异步得到最后的中值为 0x37，与上面波形最下面的 matrix_mid 的数据是匹配的，验证了设计的正确性。

36.3系统板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。顶层的仿真与“基于 DDR3 的串口传图帧缓存系统”类似，这里就不做详细讲解，读者可以参考本章例程或自己建立仿真文件完成顶层的仿真。

36.3.1灰度图像中值滤波工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。

由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成 Bit 文件。上板调试硬件平台基于高云开发板，使用一根数据线，一端接入高云开发板的 UART 接口，另一端接入 PC 机的 USB 口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接示意图与“基于 DDR3 的串口传图帧缓存系统”一样。

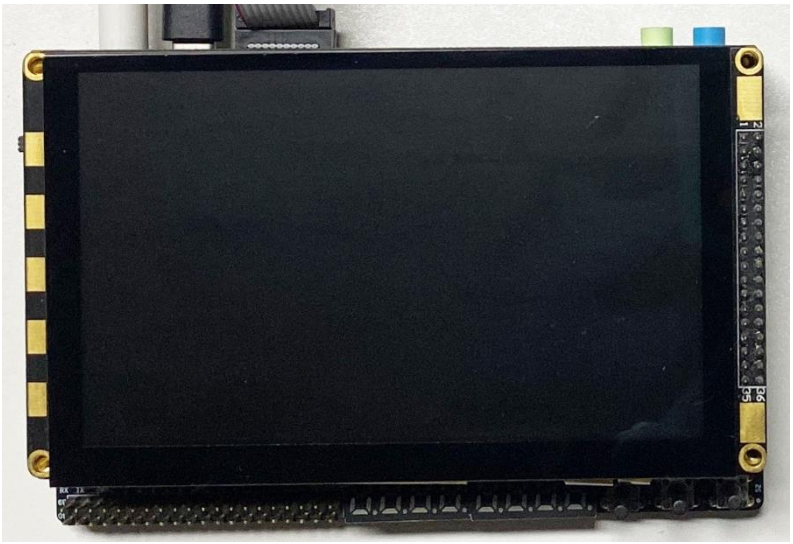


图 36-18 灰度图像中值滤波硬件连接

硬件连接好并上电后然后下载生成的 Bit 文件。下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态。

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径；图片宽度设置成与显示屏分辨率宽度一半，高度设置成与显示屏分辨率高度一致（TFT5.0 寸屏是 800*480，上位机上宽度设置为 400，高度设置为 480；TFT4.3 寸屏是 480*272，上位机上宽度设置为 240，高度设置为 272）。下图是待传的图片（带有椒盐噪声）。

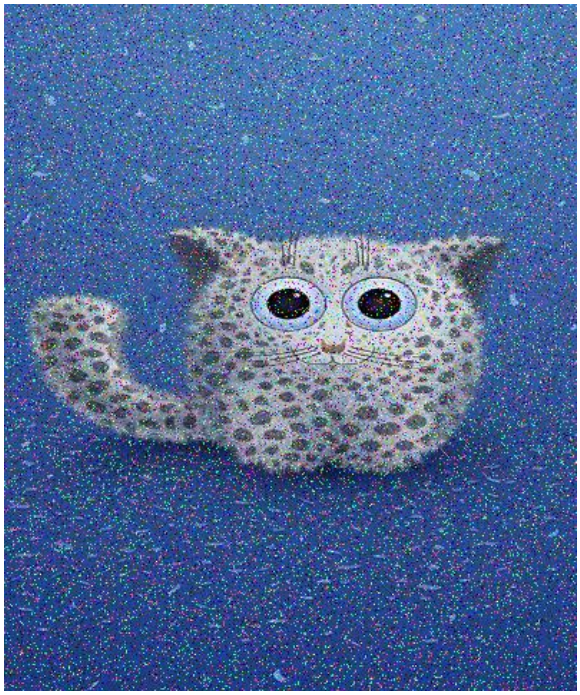


图 36-19 需进行中值滤波处理的图片（椒盐噪声）

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24 的 bmp 格式图片，图片的宽度和高度需要为 400*480（插 5 寸屏情况下）或 240*272（插 4.3 寸屏情况下）。

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用的是 COM6），设置好传图工具后，点击“连接设备”。

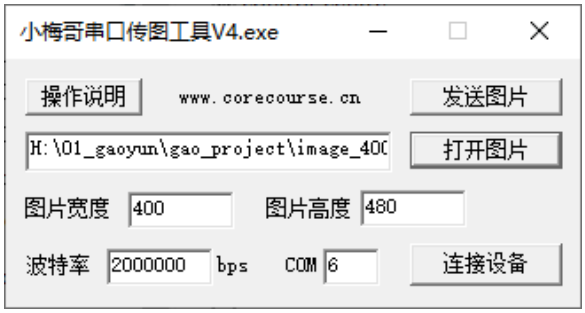


图 36-20 连接开发板和软件

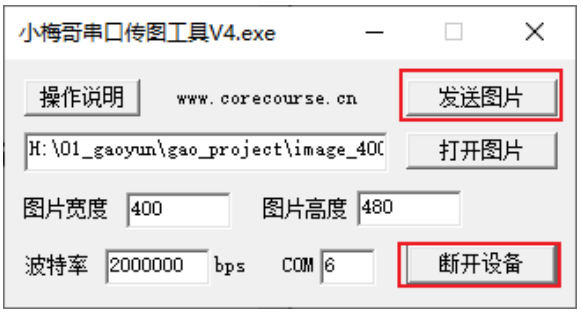


图 36-21 发送图片

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下。

图 36-22 中值滤波处理前（左）后（右）效果对比图（TFT）

通过对比可以看出，TFT 屏上左右两边分别显示的是彩色图像灰度化图像数据和经过中值滤波处理之后的灰度图像。可以明显的看出，中值滤波处理可以有效的去除图像中的噪声数据。

36.3.2灰度图像中值滤波工程添加 HDMI 显示

结合前面讲解的 HDMI 章节知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。关于本小节对工程输出端口改造的方法，可参考串口传图章节对应：基于 DDR3 的串口传图帧缓存系统设计实现（HDMI 和 TFT 显示），这里就不再重复讲解了。

最终，添加双 HDMI 工程显示的效果如下：

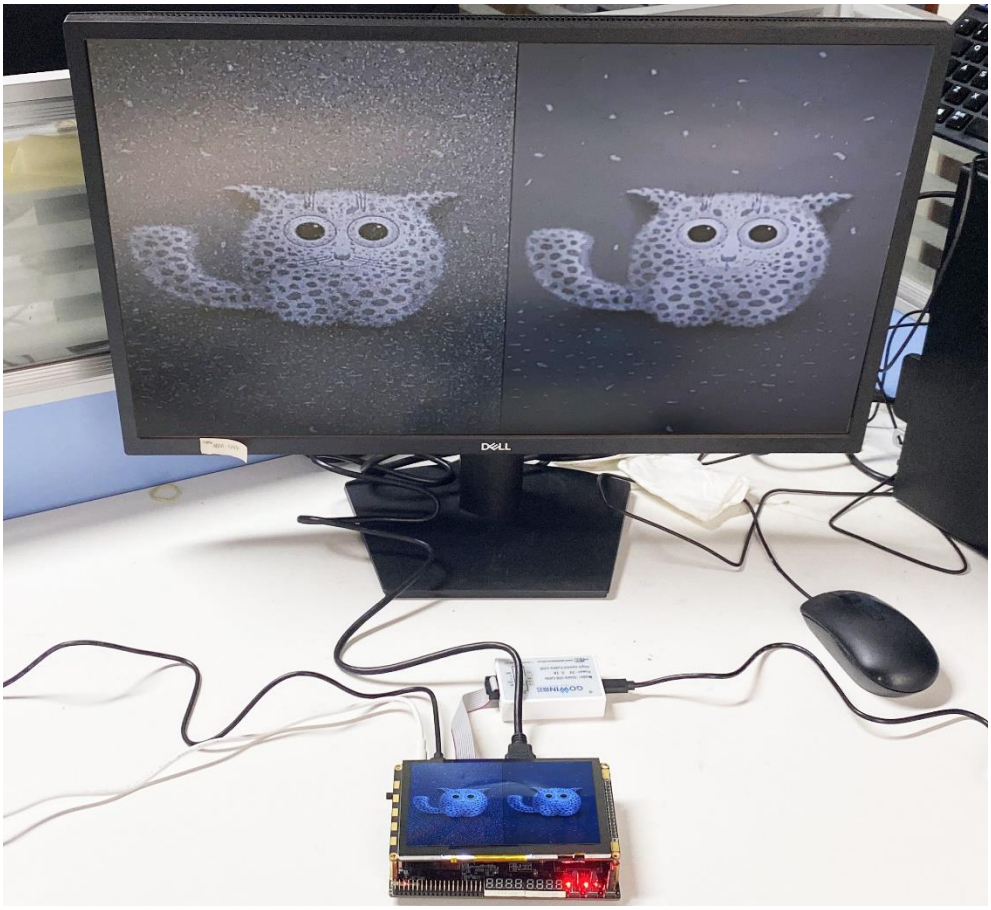


图 36-23 中值滤波处理前（左）后（右）效果对比图（HDMI）

至此，灰度图像中值滤波在高云开发板上的设计及验证就成功完成了。

36.4总结

将彩色模块的数据流经过灰度处理，得到灰度图像，将灰度图像的数据流经过中值滤波模板的处理，得到中值滤波后的图像。从设计来看，灰度图像的中值滤波可以有效祛除图像上的孤立噪声点。

37 灰度图像均值滤波设计实现(HDMI 和 TFT 显示)

工程源码	|----02_设计实例 |----ch37_uart_ddr3_tft_hdmi_mean_filter
相关视频课程	
说明	

章节导读

本节主要是在上一节的基础上，将中值滤波图像处理模块更换成均值滤波处理，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，同时进行均值滤波的处理，然后将均值滤波处理前和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏和 HDMI 显示器上同时显示均值滤波处理前的灰度图像和处理后的灰度图像。

37.1系统整体设计

在本节内容中，基于灰度图像的均值滤波，我们希望得到如下实验效果。最终 TFT 和 HDMI 的显示要求如下图 37-1。

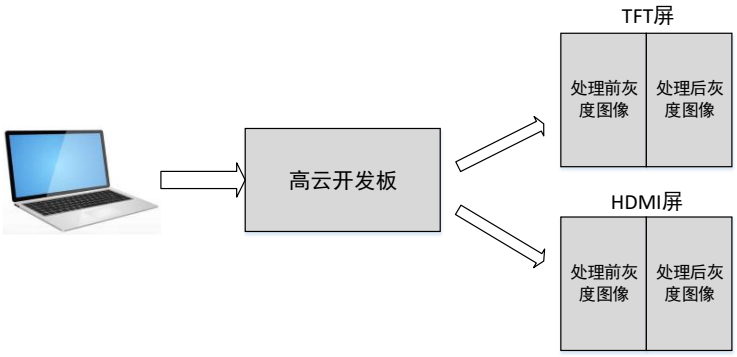


图 37-1 均值滤波实验目标

系统的整体设计框图如下。大体结构与“基于 FPGA 的灰度图像中值滤波的设计实现”基本一致，将中值滤波模块更换成了均值滤波处理模块。

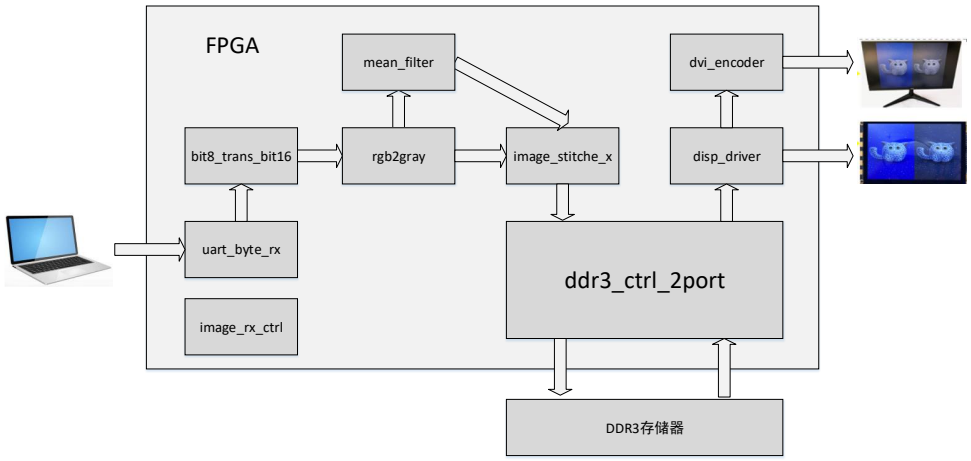


图 37-2 系统设计框图

除了均值滤波模块，其他各模块介绍与前面一样，这里不再重复介绍。

37.2灰度图像均值滤波处理模块的设计

37.2.1基本原理

均值滤波是典型的线性滤波算法。和中值滤波相似的是，它同样是通过让待处理目标像素，被一个邻域模板施加影响，达到图像处理的效果。如 3*3 的模板就是以目标像素为中心的周围 8 个像素，构成一个滤波模板，即去掉目标像素本身。实际使用时，该算法通过求取模板中的全体像素的平均值来代替原来像素值构成一幅新的图像以实现滤波。而这种算法，在专业领域被命名为邻域平均法。

例如，对于希望处理的当前像素点（x，y），选择一个模板，该模板由其近邻的若干像素组成，先求出模板中所有像素的均值，再把均值赋予当前像素点（x，y），作为处理后图像在该点的灰度 g（x，y），即 g（x，y）=1/m ∑f（x，y） m 为该模板中包含当前像素在内的像素总个数。

虽然均值滤波可以在一定程度上去除图像上的噪点，但专家的理论分析结合前人的实践经验还告诉我们，均值滤波本身存在着固有的缺陷，即它不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊，不能很好地去除噪声点。

37.2.2实现方法

如下图 37-3 所示，为一个 3*3 的图像模板。

(x-1, y-1)	(x, y-1)	(x+1, y-1)
(x-1, y)	(x, y)	(x+1, y)
(x11, y+1)	(x, y+1)	(x+1, y+1)

图 37-3 均值滤波图像模板

中心点（x，y）为均值滤波将要处理的位置，f(x,y)表示（x,y）点的像素值，g(x,y)表示（x,y）点经过均值处理后的值。均值滤波公式表示如下：

```
g(x,y)=1/9*(f(x-1,y-1)+f(x,y-1)+f(x+1,y-1)+f(x-1,y)+
f(x,y)+f(x+1,y)+f(x-1,y+1)+f(x,y+1)+f(x+1,y+1))
```

由上式我们看出(x,y)点的 3x3 像素点的均值等于 3*3 模板的 9 个点的像素值之和除以 9。由于 FPGA 不擅长算乘除法，为了简便运算只将目标像素周围八个点求和然后除以 8（即右移三位），取代中间像素点。

这样，公式可以化简如下：

```
g(x,y)=1/8*(f(x-1,y-1)+f(x,y-1)+f(x+1,y-1)+f(x-1,y)+f(x+1,y)+f(x-
1,y+1)+f(x,y+1)+f(x+1,y+1))
```

FPGA 实现该算法步骤如下：

第一步：形成 3*3 举证像素，这个在中值滤波设计中已经有讲过，这节可以直接使用。

第二步：求周围邻域八个点的像素值之和。

第三步：将第二步结果右移三位（相当于除以 8）得到结果。

37.2.3模块接口设计

了解了均值滤波算法后，我们同样可以明确均值滤波模块接口如下图 37-4，模块端口描述如下表 37-1 所示。

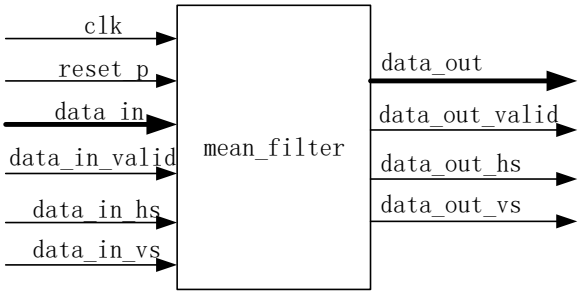


图 37-4 均值滤波模块输入输出接口图

表 37-1 模块端口描述表

端口名称	I/O	端口说明
clk	I	图像像素时钟
reset_p	I	模块复位信号，高电平有效
data_in	I	图像像素数据，数据位宽可自定义，对于灰度图像，位宽为 8
data_in_valid	I	图像像素数据有效标识，为 1 表示当前 data_in 有效
data_in_hs	I	图像行信号
data_in_vs	I	图像场信号
data_out	O	图像处理后数据输出，数据位宽与 data_in 相同
data_out_valid	O	图像处理后像素数据有效标识，为 1 表示当前 data_out 有效
data_out_hs	O	图像处理后行信号
data_out_vs	O	图像处理后场信号

37.2.4实现过程

3*3 的模板数据的产生在上节中值滤波处理已经讲解，根据算法的步骤，只需要计算邻域八个点的像素值求均值即可。具体代码实现如下。

第 1 部分：端口定义

```
module mean_filter
#(
  parameter DATA_WIDTH = 8
)
(
  input                       clk,    //pixel clk
  input                       reset_p,
  input     [DATA_WIDTH-1:0] data_in,
  input                       data_in_valid,
  input                       data_in_hs,
  input                       data_in_vs,

  output    [DATA_WIDTH-1:0] data_out,
  output reg                  data_out_valid,
  output reg                  data_out_hs,
  output reg                  data_out_vs
);
//line data
wire [DATA_WIDTH-1:0] line0_data;
wire [DATA_WIDTH-1:0] line1_data;
wire [DATA_WIDTH-1:0] line2_data;
//matrix 3x3 data
reg  [DATA_WIDTH-1:0] row0_col0;
reg  [DATA_WIDTH-1:0] row0_col1;
reg  [DATA_WIDTH-1:0] row0_col2;
```

```
reg  [DATA_WIDTH-1:0] row1_col0;
reg  [DATA_WIDTH-1:0] row1_col1;
reg  [DATA_WIDTH-1:0] row1_col2;

reg  [DATA_WIDTH-1:0] row2_col0;
reg  [DATA_WIDTH-1:0] row2_col1;
reg  [DATA_WIDTH-1:0] row2_col2;
```

第 3 部分：建立滤波像素矩阵

```
//----------------------------------------------------
// matrix 3x3 data
// row0_col0   row0_col1   row0_col2
// row1_col0   row1_col1   row1_col2
// row2_col0   row2_col1   row2_col2
//----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p) begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
  else if(data_in_hs && data_in_vs)
    if(data_in_valid) begin
      row0_col2 <= line0_data;
      row0_col1 <= row0_col2;
      row0_col0 <= row0_col1;

      row1_col2 <= line1_data;
      row1_col1 <= row1_col2;
      row1_col0 <= row1_col1;

      row2_col2 <= line2_data;
      row2_col1 <= row2_col2;
      row2_col0 <= row2_col1;
    end
    else begin
      row0_col2 <= row0_col2;
      row0_col1 <= row0_col1;
      row0_col0 <= row0_col0;
```

```
      row1_col2 <= row1_col2;
      row1_col1 <= row1_col1;
      row1_col0 <= row1_col0;

      row2_col2 <= row2_col2;
      row2_col1 <= row2_col1;
      row2_col0 <= row2_col0;
    end
  else begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
end
```

第 4 部分：均值滤波运算

```
always @(posedge clk)
begin
  data_in_valid_dly1 <= data_in_valid;
  data_in_hs_dly1    <= data_in_hs;
  data_in_vs_dly1    <= data_in_vs;
end

//----------------------------------------------------
//result
//----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p)
    data_out_tmp <= 'd0;
  else if(data_in_valid_dly1)
    data_out_tmp <= (row0_col0 + row0_col1 + row0_col2 +
                     row1_col0 +             row1_col2 +
                     row2_col0 + row2_col1 + row2_col2 );
end

assign data_out = data_out_tmp>>3;

always @(posedge clk)
```

```
begin
  data_out_valid <= data_in_valid_dly1;
  data_out_hs    <= data_in_hs_dly1;
  data_out_vs    <= data_in_vs_dly1;
end

endmodule
```

37.2.5仿真验证

均值滤波处理模块设计完成后，编写仿真验证 testbench 文件，仿真验证代码与中值滤波基本是一样的，将例化的中值滤波模块替换成均值滤波模块即可，具体代码如下：

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module mean_filter_tb();

  reg       clk;    //pixel clk
  reg       reset_p;
  reg [7:0] data_in;
  reg       data_in_valid;
  reg       data_in_hs;
  reg       data_in_vs;
  wire[7:0] data_out;
  wire      data_out_valid;
  wire      data_out_hs;
  wire      data_out_vs;

initial clk = 1'b1;
always #(`CLK_PERIOD/2) clk = ~clk;

initial begin
  reset_p = 1'b1;
  data_in = 8'd0;
  data_in_valid = 1'b0;
  data_in_hs = 1'b0;
  data_in_vs = 1'b0;
  #201;
  reset_p = 1'b0;
  #200;

  data_in_vs = 1'b1;
  repeat(480) begin
    #500;
    data_in_hs = 1'b1;
```

```
    #500;
    repeat(400*2) begin
      data_in_valid = ~data_in_valid;
      data_in = $random % 256;
      #(`CLK_PERIOD);
    end
    #500;
    data_in_hs = 1'b0;
  end
  data_in_vs = 1'b0;
  #(`CLK_PERIOD);
  data_in_vs = 1'b1;

  #2000;
  $stop;
end

mean_filter
#(
  .DATA_WIDTH ( 8 )
)mean_filter
(
  .clk            (clk           ),    //pixel clk
  .reset_p        (reset_p       ),
  .data_in        (data_in       ),
  .data_in_valid  (data_in_valid ),
  .data_in_hs     (data_in_hs    ),
  .data_in_vs     (data_in_vs    ),

  .data_out       (data_out      ),
  .data_out_valid (data_out_valid),
  .data_out_hs    (data_out_hs   ),
  .data_out_vs    (data_out_vs   )
);
endmodule
```

运行仿真，任意找到一个仿真地方进行放大观察，仿真波形如下图 37-5 所示。

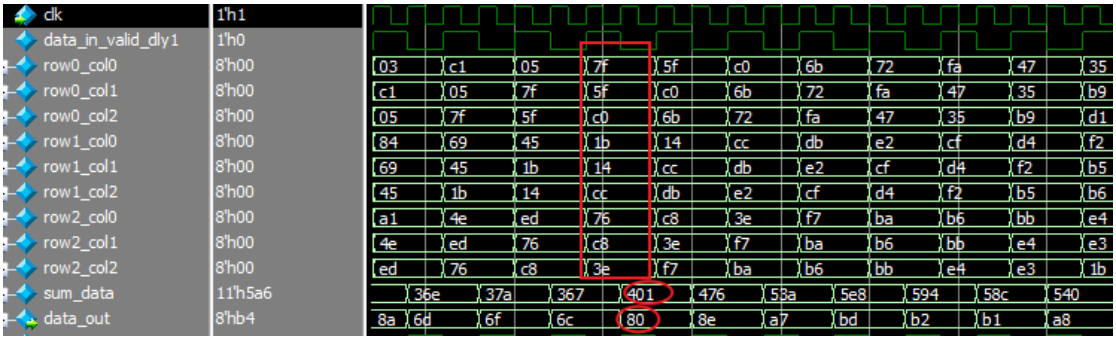


图 37-5 仿真波形任意处核验设计思路

从上面波形数据可以看出，当前 3*3 的模板数据如下图 37-6 所示。

7f	5f	c0
1b	14	cc
76	c8	3e

图 37-6 被观察的像素点及其邻域

根据 FPGA 计算均值的公式可得，周围邻域八个点的像素值之和为（0x7f+0x5f+0xc0+0x1b+0xcc+0x76+0xc8+0x3e）= 0x401，然后右移 3 位得到 0x80，仿真中 sum_data 和 data_out 与该计算结果一致，验证了设计的正确性。

37.2.6系统板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。顶层的仿真与“基于 FPGA 的灰度图像中值滤波的设计实现”类似，这里就不做详细讲解，读者自己建立仿真文件完成顶层仿真。

37.2.7灰度图像均值滤波工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。

由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成 bit 文件。上班调试硬件平台基于高云开发板，使用一根数据线，一端接入高云开发板的 UART 接口，一端接入 PC 机的 USB 口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接示意图与“基于 DDR3 的串口传图帧缓存系统”一样。

图 37-7 灰度图像均值滤波硬件

硬件连接好并上电后然后下载生成的 bit 文件。下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态。

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径：图片宽度设置成与显示屏分辨率宽度一半，高度设置成与显示屏分辨率高度一致（TFT5.0 寸屏是 800*480，上位机上宽度设置为 400，高度设置为 480；TFT4.3 寸屏是 480*272，上位机上宽度设置为 240，高度设置为 272）。下图 37-8 是待传的图片（带有椒盐噪声）。

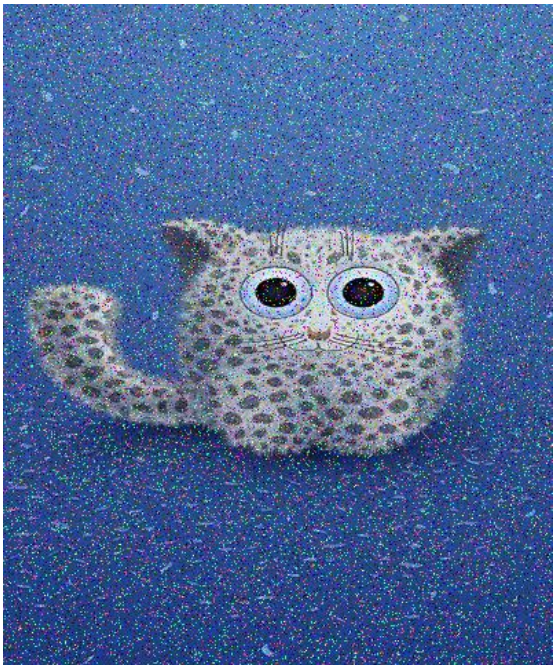


图 37-8 带椒盐噪声的待传图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片深度为 24 的 bmp 格式图片，图片的宽度和高度需要为 400*480（插 5 寸屏情况下）或 240*272（插 4.3 寸屏情况下）。

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用的是 COM6），设置好传图工具后，点击“连接设备”。

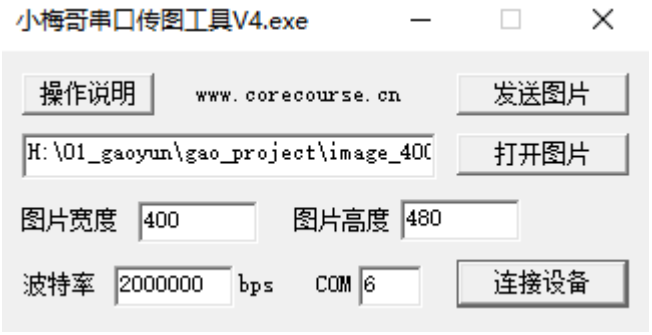


图 37-9 配置传图工具

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

图 37-10 发送图片

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下图 37-11。

图 37-11 均值滤波处理前（左）后（右）效果对比图（TFT）

通过对比可以看出，TFT 屏上左右两边分别显示的是彩色图像灰度化图像数据和经过均值滤波处理之后的灰度图像。均值滤波略去了部分噪声，像素值高的会被拉低，像素值低的会被拉高，趋向于一个平均值。

37.2.8灰度图像均值滤波工程添加 HDMI 显示

结合前面讲解的 HDMI 章节知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。关于本小节对工程输出端口改造方法，可参考串口传图章节对应；串口传图工程添加 HDMI 显示的内容，这里就不再重复讲解了。

最终，添加 HDMI 工程显示的效果如下图 37-12：

图 37-12 均值滤波处理前（左）后（右）效果对比图

至此，灰度图像中值滤波在高云开发板上的设计及验证就成功完成了。

37.3总结

将彩色模块的数据流经过灰度处理，得到灰度图像，将灰度图像的数据流经过均值滤波模板的处理，得到均值滤波后的图像。从设计来看，灰度图像的均值滤波不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊。

38 灰度图像高斯滤波设计实现(HDMI 和 TFT 显示)

工程源码	|----02_设计实例 |----ch38_uart_ddr3_tft_hdmi_gaussian_filter
相关视频课程	
说明	

章节导读

本节主要是在上一节的基础上，更换图像处理模块，更换为高斯滤波处理模块，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，同时进行高斯滤波的处理，然后将高斯滤波处理前和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏和 HDMI 显示器上同时显示高斯滤波处理前的灰度图像和处理后的灰度图像。

38.1系统整体设计

在本节内容中，基于灰度图像的均值滤波，我们希望得到如下实验效果。最终 TFT 和 HDMI 的显示要求如下。

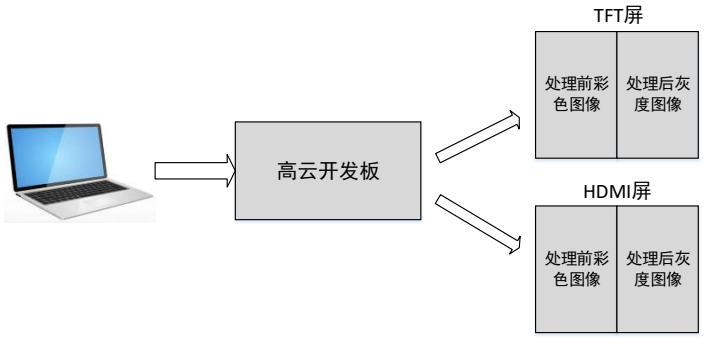


图 38-1 高斯滤波实验目标

系统整体设计框图如下图 38-2。大体结构与“基于 FPGA 的灰度图像中值滤波的设计实现”基本一致，将中值滤波模块更换成了高斯滤波处理模块。

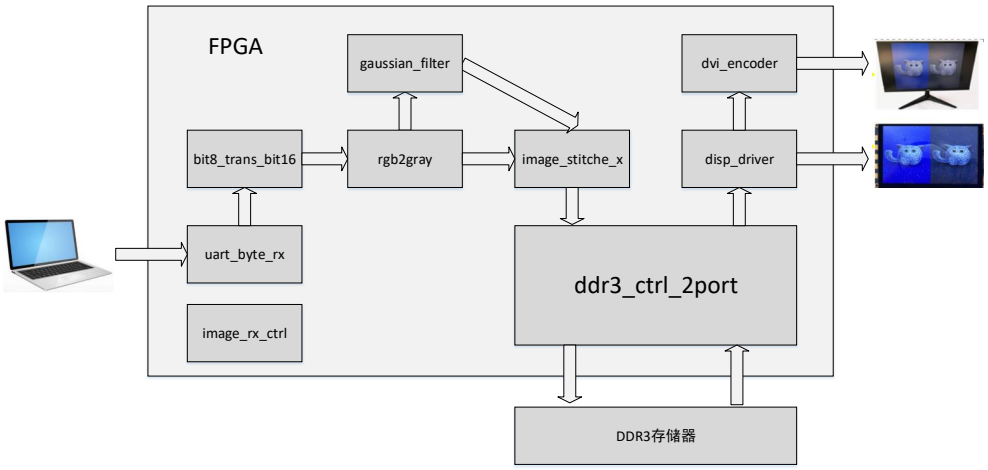


图 38-2 系统设计框图

除了高斯滤波模块，其他各模块介绍与前面一样，这里不再重复介绍。

38.2灰度图像高斯滤波处理模块的设计

38.2.1基本原理

高斯滤波是一种线性平滑滤波，适用于消除高斯噪声，广泛应用于图像处理的减噪过程。通俗的来讲，高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。高斯滤波的具体操作是：用一个模板（或称卷积、掩模）扫描图像中的每一个像素，用模板确定的邻域内像素的加权平均灰度值去替代模板中心像素点的值。

一维高斯分布：

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$

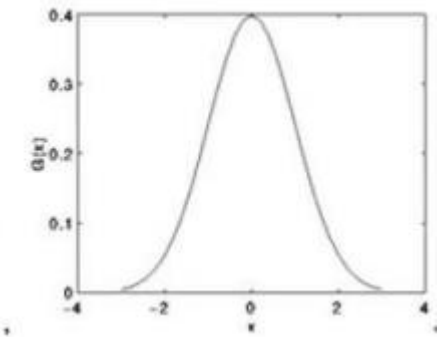


图 38-3 一维高斯分布图

二维高斯分布：

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

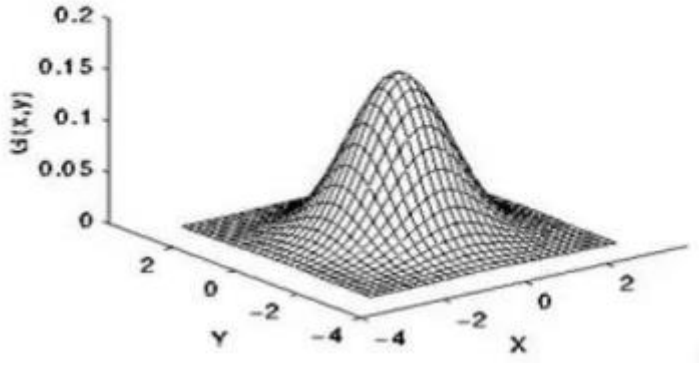


图 38-4 二维高斯分布图

高斯滤波后图像被平滑的程度取决于标准差。它的输出是邻域像素的加权平均，同时离中心越近的像素权重越高。因此，相对于均值滤波（mean filter）它的平滑效果更柔和，而且边缘保留的也更好。

高斯滤波被用作平滑滤波器的本质原因是因为它是一个低通滤波器，而且大部分基于卷积平滑滤波器都是低通滤波器。

GAUSS 滤波算法克服了边界效应，因而滤波后的图像较好。

38.2.2实现方法

均值滤波算法中所有参与运算像素点权重一致，而高斯滤波算法则不同。高斯滤波以中心像素点为圆心，按距离中心像素点的距离，给邻域点赋予权重值。距离越近则权重值越高，距离越远则权重值越低。根据权重公式，5*5 高斯滤波算子权重表格如下表 38-1：

表 38-1 高斯滤波 5*5 算子

(1/273) *	1	4	7	4	1
	4	16	26	16	4
	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1

而 3*3 高斯滤波算子权重表格如下表 38-2：

表 38-2 高斯滤波 3*3 算子

	1	2	1
(1/16)*	2	4	2
	1	2	1

表 38-3 串行像素形成 3*3 矩阵

(x-1,y-1)	(x,y-1)	(x+1,y-1)
(x-1,y)	(x,y)	(x+1,y)
(x-1,y+1)	(x,y+1)	(x+1,y+1)

如果用 f(x,y)表示待处理像素(x,y)点的像素值；而用 g(x,y)表示(x,y)点经过高斯滤波处理后的值；

用模板确定的邻域内像素的加权平均灰度值去替代模板中心像素点的值计算公式可以如下表示：

```
g(x,y)=（1/16）* (  f(x-1,y-1)+2f(x,y-1)+f(x+1,y-1)+2f(x-1,y)+4f(x,y)
+ 2f(x+1,y)+ f(x-1,y+1)+2f(x,y+1)+f(x+1,y+1)  )
```

如果用模板扫描图像中的每一个像素，就可以完成整幅图像的高斯滤波。

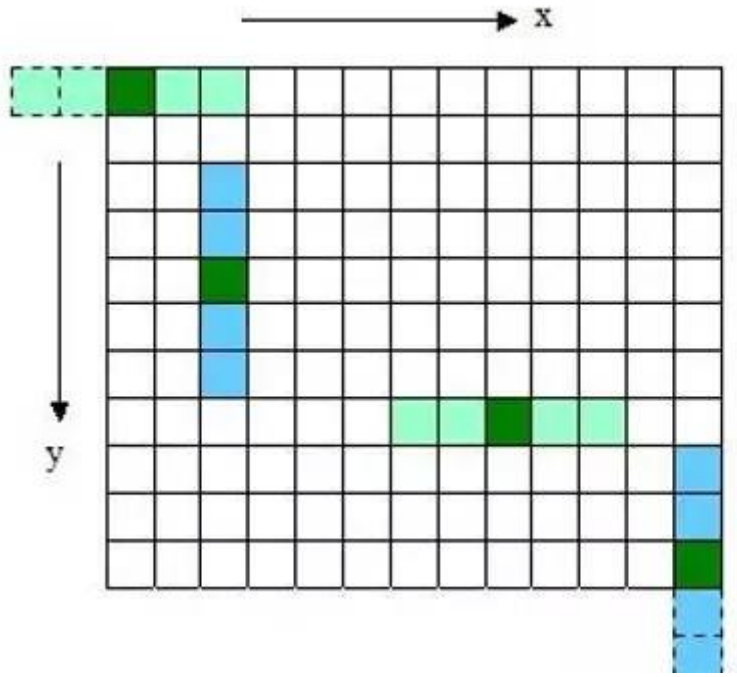


图 38-5 模板扫描像素点示意图

如图 38-5 所示，如果使用高斯滤波模板依次（从左到右，从上到下）划过每一个屏幕图像待输出点的对应存储空间，则就最终可以完成一帧图像的高斯滤波。因为原始图像，是按逐行扫描，每扫完一行换下一行的规律进行的图像输出，所以高斯滤波模板的滑动，也可以按这一输出规则，同步对这些数据进行处理而实现。

FPGA 实现该算法的步骤如下：

第一步：形成 3*3 矩阵像素，这个在中值滤波设计中已经有讲过，这节可以直接使用。

第二步：根据公司计算乘加和

第三步：将第二部结果右移 4 位（即除以 16）得到结果

38.2.3模块接口设计

和前面的章节讲解内容一致，高斯滤波数据输入接口为 16 位灰度值，输出接口位滤波后得到的 data_out 值。

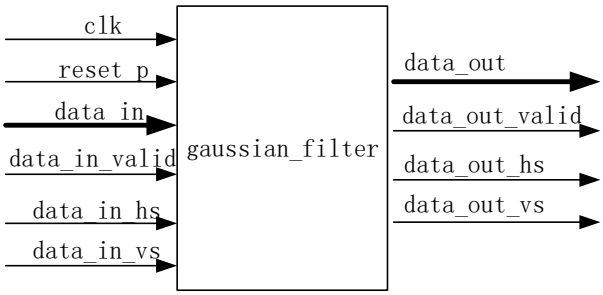


图 38-6 高斯滤波输入输出接口

表 38-4 模块端口描述如下表

端口名称	I/O	端口说明
clk	I	图像像素时钟
reset_p	I	模块复位信号，高电平有效
data_in	I	图像像素数据，数据位宽可自定义，对于灰度图像，位宽为 8
data_in_valid	I	图像像素数据有效标识，为 1 表示当前 data_in 有效
data_in_hs	I	图像行信号
data_in_vs	I	图像场信号
data_out	O	图像处理后数据输出，数据位宽与 data_in 相同
data_out_valid	O	图像处理后像素数据有效标识，为 1 表示当前 data_out 有效
data_out_hs	O	图像处理后行信号
data_out_vs	O	图像处理后场信号

38.2.4实现过程

3*3 模板数据的产生在上节中值滤波处理已经讲解，根据算法的步骤，只需要计算乘加和，然后除以 16 即可。具体代码实现如下。

```
module gaussian_filter
#(
  parameter DATA_WIDTH = 8
)
(
  input                       clk,    //pixel clk
  input                       reset_p,
  input     [DATA_WIDTH-1:0] data_in,
  input                       data_in_valid,
  input                       data_in_hs,
  input                       data_in_vs,
```

```
  output reg[DATA_WIDTH-1:0] data_out,
  output reg                 data_out_valid,
  output reg                 data_out_hs,
  output reg                 data_out_vs
);
//line data
wire [DATA_WIDTH-1:0] line0_data;
wire [DATA_WIDTH-1:0] line1_data;
wire [DATA_WIDTH-1:0] line2_data;
//matrix 3x3 data
reg  [DATA_WIDTH-1:0] row0_col0;
reg  [DATA_WIDTH-1:0] row0_col1;
reg  [DATA_WIDTH-1:0] row0_col2;

reg  [DATA_WIDTH-1:0] row1_col0;
reg  [DATA_WIDTH-1:0] row1_col1;
reg  [DATA_WIDTH-1:0] row1_col2;

reg  [DATA_WIDTH-1:0] row2_col0;
reg  [DATA_WIDTH-1:0] row2_col1;
reg  [DATA_WIDTH-1:0] row2_col2;
//
reg   data_in_valid_dly1;
reg   data_in_hs_dly1;
reg   data_in_vs_dly1;
reg [DATA_WIDTH+3:0] sum_data;

//3xline data
shift_register_2taps
#(
  .DATA_WIDTH ( DATA_WIDTH )
)shift_register_2taps(
  .clk           (clk           ),
  .shiftin       (data_in       ),
  .shiftin_valid (data_in_valid ),

  .shiftout      (              ),
  .taps0x        (line0_data    ),
  .taps1x        (line1_data    )
);

assign line2_data = data_in;

//-----------------------------------------------------
// matrix 3x3 data
// row0_col0   row0_col1   row0_col2
// row1_col0   row1_col1   row1_col2
```

```
// row2_col0   row2_col1   row2_col2
//-----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p) begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
  else if(data_in_hs && data_in_vs)
    if(data_in_valid) begin
      row0_col2 <= line0_data;
      row0_col1 <= row0_col2;
      row0_col0 <= row0_col1;

      row1_col2 <= line1_data;
      row1_col1 <= row1_col2;
      row1_col0 <= row1_col1;

      row2_col2 <= line2_data;
      row2_col1 <= row2_col2;
      row2_col0 <= row2_col1;
    end
    else begin
      row0_col2 <= row0_col2;
      row0_col1 <= row0_col1;
      row0_col0 <= row0_col0;

      row1_col2 <= row1_col2;
      row1_col1 <= row1_col1;
      row1_col0 <= row1_col0;

      row2_col2 <= row2_col2;
      row2_col1 <= row2_col1;
      row2_col0 <= row2_col0;
    end
  else begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;
```

```
    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
end

always @(posedge clk)
begin
  data_in_valid_dly1 <= data_in_valid;
  data_in_hs_dly1    <= data_in_hs;
  data_in_vs_dly1    <= data_in_vs;
end

//----------------------------------------------------
//result
//----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p)
    sum_data <= 'd0;
  else if(data_in_valid_dly1)
    sum_data <= (row0_col0   + row0_col1*2 + row0_col2 +
                 row1_col0*2 + row1_col1*4 + row1_col2*2 +
                 row2_col0   + row2_col1*2 + row2_col2);
end

assign data_out = sum_data>>4;

always @(posedge clk)
begin
  data_out_valid <= data_in_valid_dly1;
  data_out_hs    <= data_in_hs_dly1;
  data_out_vs    <= data_in_vs_dly1;
end

endmodule
```

38.2.5仿真验证

高斯滤波处理模块设计完成后，编写仿真验证 testbench 文件，仿真验证代码与中值滤波基本是一样的，将例化的中值滤波模块替换成均值滤波模块即可，具体代码如下。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module gaussian_filter_tb();

  reg       clk;    //pixel clk
  reg       reset_p;
  reg [7:0] data_in;
  reg       data_in_valid;
  reg       data_in_hs;
  reg       data_in_vs;
  wire[7:0] data_out;
  wire      data_out_valid;
  wire      data_out_hs;
  wire      data_out_vs;

initial clk = 1'b1;
always #(`CLK_PERIOD/2) clk = ~clk;

initial begin
  reset_p = 1'b1;
  data_in = 8'd0;
  data_in_valid = 1'b0;
  data_in_hs = 1'b0;
  data_in_vs = 1'b0;
  #201;
  reset_p = 1'b0;
  #200;

  data_in_vs = 1'b1;
  repeat(480) begin
    #500;
    data_in_hs = 1'b1;
    #500;
    repeat(400*2) begin
      data_in_valid = ~data_in_valid;
      data_in = $random % 256;
      #(`CLK_PERIOD);
    end
    #500;
    data_in_hs = 1'b0;
  end
  data_in_vs = 1'b0;
  #(`CLK_PERIOD);
  data_in_vs = 1'b1;

  #2000;
```

```
  $stop;
end

gaussian_filter
#(
  .DATA_WIDTH ( 8 )
)gaussian_filter
(
  .clk            (clk           ),    //pixel clk
  .reset_p        (reset_p       ),
  .data_in        (data_in       ),
  .data_in_valid  (data_in_valid ),
  .data_in_hs     (data_in_hs    ),
  .data_in_vs     (data_in_vs    ),

  .data_out       (data_out      ),
  .data_out_valid (data_out_valid),
  .data_out_hs    (data_out_hs   ),
  .data_out_vs    (data_out_vs   )
);

endmodule
```

运行仿真，任意找到一个仿真地方进行放大观察，仿真波形如下图 38-7。

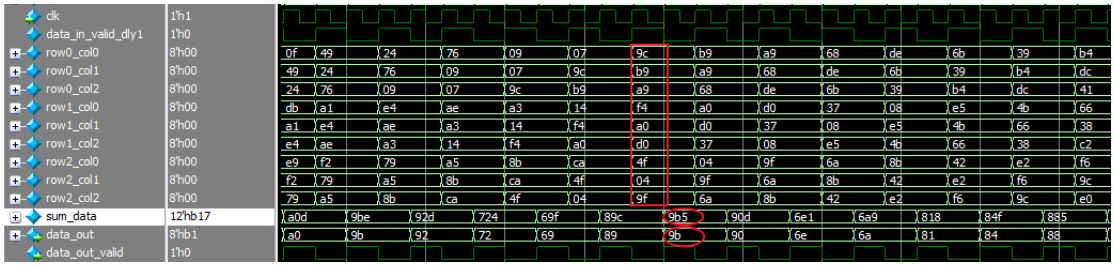


图 38-7 仿真波形图

从上面波形数据可以看出，当前 3*3 的模板图像数据如下图 38-8。

9c	b9	a9
f4	a0	d0
4f	04	9f

图 38-8 3*3 的模板图像数据

根据 FPGA 计算公式可得，高斯滤波处理后的数据计算结果为（1*0x9c+2*0xb9+1*0xa9+2*0xf4+4*0xa0+2*0xd0+1*0x4f+2*0x04+1*0x9f）=0x9B5，然后右移 4 位得到 0x9B，仿真中 sum_data 和 data_out 与该计算结果一致。验证了设计的正确性。

38.2.6系统板级测试

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。顶层的仿真与“基于 FPGA 的灰度图像中值滤波的设计实现”类似，这里就不做详细讲解，读者自己建立仿真文件完成顶层的仿真。

38.2.7灰度图像高斯滤波工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。

由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成 bit 文件。上板调试硬件平台基于高云开发板，使用一个数据线，一端接入高云开发板的 UART 接口，另一端接入 PC 机的 USB 口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接示意图与“基于 DDR3 的串口传图帧缓存系统”一样。

硬件连接好并上电后，然后下载 bit 文件。下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态。

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径；图片宽度设置成与显示屏分辨率宽度一半，高度设置成与显示屏分辨率高度一致（TFT5.0 寸屏是 800*480，上位机上宽度设置为 400，高度设置为 480；TFT4.3 寸屏是 480*272，上位机上宽度设置为 240，高度设置为 272）。

下图 38-9 是待传的图片（带有高斯噪声）。

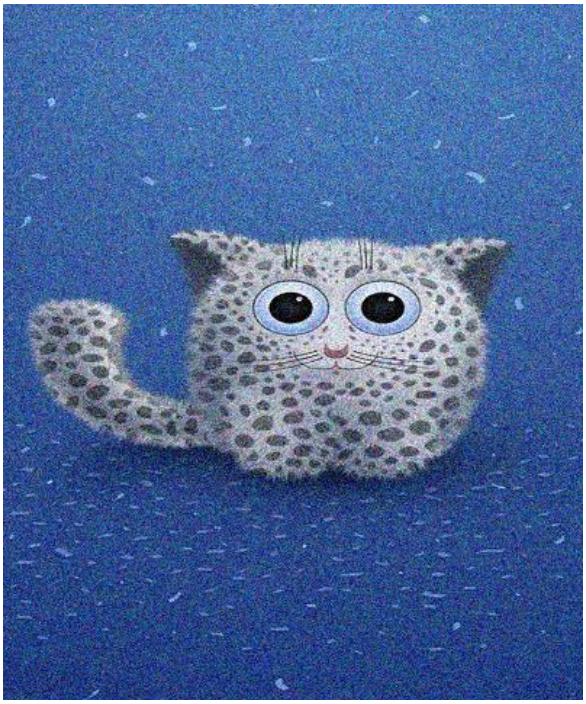


图 38-9 待传的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24 的 bmp 格式图片，图片的宽度和高度需要为 400*480（插 5 寸屏情况下）或 240*272（插 4.3 寸屏情况下）。

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用的是 COM6），设置好传图工具后，点击“连接设备”。

图 38-10 串口传图工具连接开发板

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

图 38-11 启动传图

传图过程中，可以看到 TFT 屏上开始显示发送的图片。图片传送完成后，TFT 屏显示效果如下。

图 38-12 高斯滤波处理前（左）后（右）效果对比图（TFT）

通过对比可以看出，TFT 屏上左右两边分别显示的是彩色图像灰度化图像数据和经过高斯滤波处理之后的灰度图像。采用高斯滤波处理后有一定的滤除噪声的效果。

38.2.8灰度图像高斯滤波工程添加 HDMI 显示

结合前面讲解的 HDMI 章节知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。关于本小节对工程输出端口改造的方法，可参考串口传图章节对应：串口传图工程添加双 HDMI 显示的内容，这里就不再重复讲解了。

最终，添加双 HDMI 工程显示的效果如下图 38-13 所示：

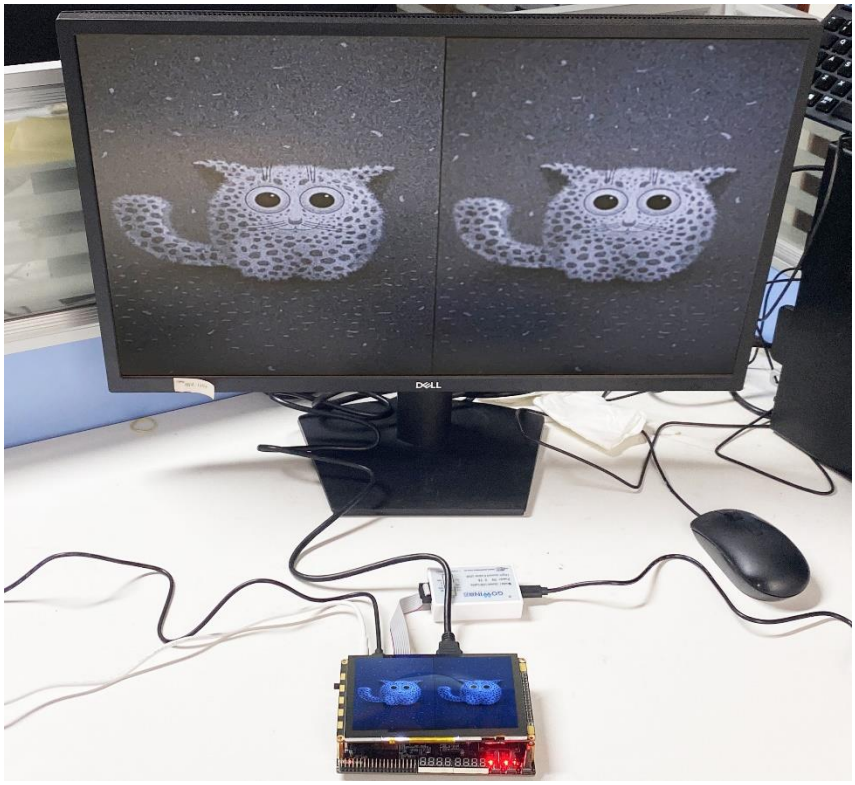


图 38-13 高斯滤波处理前（左）后（右）效果对比图（HDMI）

38.3总结

将彩色模块的数据流经过灰度处理，得到灰度图像，将灰度图像的数据流经过高斯滤波模板的处理，得到高斯滤波后的图像。从设计来看，高斯滤波相对于均值滤波（mean filter）它的平滑效果更柔和，而且边缘保留的也更好。高斯滤波算法克服了边界效应，因而滤波后的图像较好。

39 Sobel 算子边缘检测设计实现(HDMI 和 TFT 显示)

工程源码	|----02_设计实例 |----ch39_uart_ddr3_tft_hdmi_sobel_filter
相关视频课程	
说明	

章节导读

本节主要是在上一节的基础上，更换图像处理模块，更换为 sobel 算子边缘检测处理模块，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，同时进行 sobel 算子边缘检测的处理，然后将高斯滤波处理前和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏上同时显示 sobel 算子边缘检测处理前的灰度图像和处理后的灰度图像。

39.1系统整体设计

在本节内容中，基于灰度图像的均值滤波，我们希望得到如下实验效果。最终 TFT 和 HDMI 的显示要求如下。

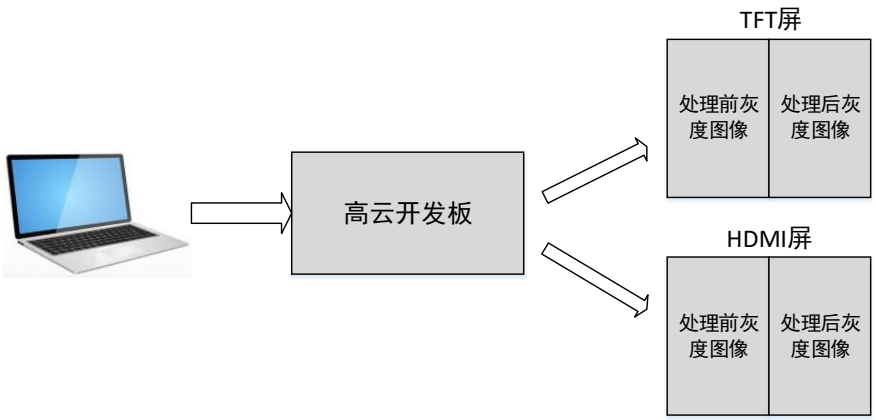


图 39-1 边缘检测实验目标

系统整体设计框图如下图 39-2。本设计的工程架构与“基于 FPGA 的灰度图像中值滤波的设计实现”基本一致，将中值滤波模块更换成了 sobel 算子边缘检测处理模块。

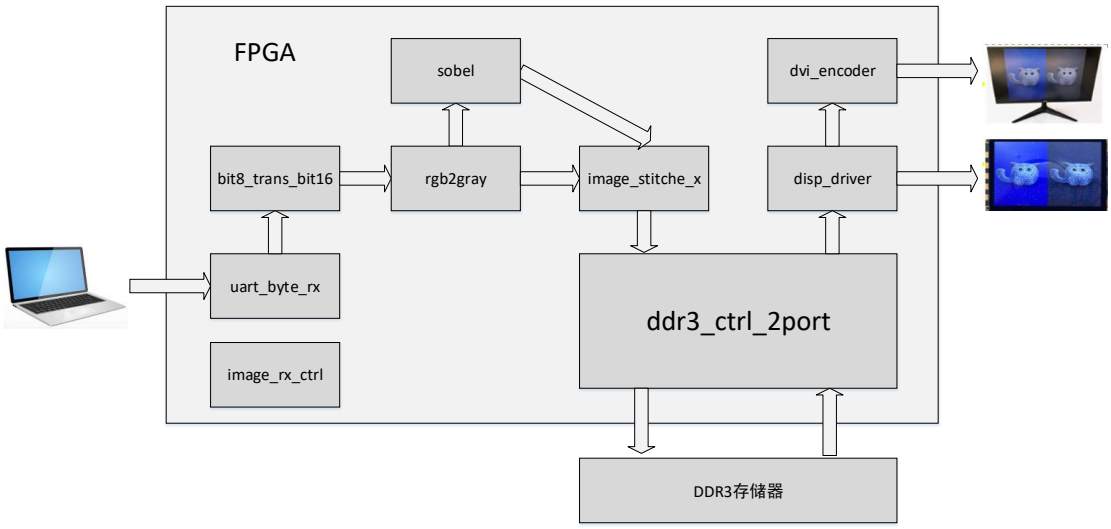


图 39-2 系统设计框图

除了 sobel 算子边缘检测模块，其他各模块介绍与前面一样，这里不再重复介绍。

39.2Sobel 算子边缘检测模块的设计

39.2.1基本原理

边缘检测是图像处理和计算机视觉中的基本问题，边缘检测的目的是标识数字图像中亮度变化明显的点。图像属性中亮度的显著变化通常反映了属性的重要事件和变化。这些属性包括：

- 深度上的不连续
- 表面方向不连续
- 物质属性变化
- 场景照明变化

边缘检测是图像处理和计算机视觉中，尤其是特征提取中的一个研究领域。

常用的边缘检测算子如下：

一阶：Roberts Cross 算子，Prewitt 算子，Sobel 算子，Kirsch 算子，罗盘算子；

二阶：Marr-Hildreth，在梯度方向的二阶导数过零点，Canny 算子，Laplacian 算子。今天我们要介绍基于 Sobel 算子边缘检测的 FPGA 算法的实现方法。

39.2.2Sobel 实现方法

Sobel 算法是像素图像边缘检测中最重要的算子之一，在机器学习、数字媒体、计算机视觉等信息科技领域起着举足轻重的作用。在技术上，它是一个离散的一阶差分算子，用来计算图像亮度函数的一阶梯度之近似值。在图像的任何一点使用此算子，将会产生该点对应的梯度矢量或是其法矢。

Sobel 边缘检测算法比较简单，实际应用中比 Canny 边缘检测效率要高，但是边缘不如 Canny 检测的准确。在很多对效率要求较高，而对细纹理不太关心的实际应用的场合，Sobel 边缘检测是首选方案。

Sobel 边缘检测通常带有方向性，可以只检测竖直边缘、垂直边缘、或都检测。Sobel 算子继承了高斯滤波算法的观念，认为邻域的像素对当前像素产生的影响不是等价的，所以距离不同的像素按距离远近被赋予不同的权值。这样一来，邻域不同点位对需计算点位的影响力不同。

表 39-1 Sobel 算子 x 方向

-1		+1
-2		+2
-1		+1

表 39-2 Sobel 算子 y 方向

+1	+2	+1
0	0	0
-1	-2	-1

表 39-3 原始图像 f

（i-1，j-1）	(i,j-1)	(i+1,j-1)
(i-1,j)	(i,j)	(i+1,j)
(i-1,j+1)	(i,j+1)	(i+1,j+1)

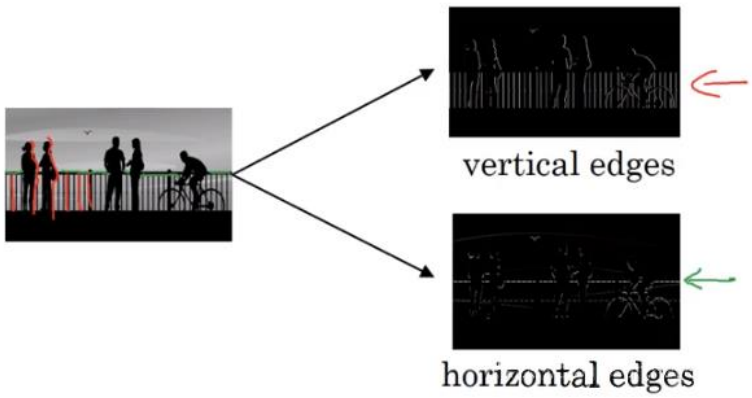


图 39-3 水平边缘检测和垂直边缘检测的不同效果

实现步骤：

第一步：Sobel 提供了水平方向和垂直方向两个方向的滤波模板。设 x 方向和 y 方向的卷积因子分别为 Gx 和 Gy，模板如下所示，A 为原图像。

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

第二步：矩阵运算后，就得到横向灰度值 Gx 和纵向灰度值 Gy，然后通过如下公式进行计算出该点的灰度值：

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

这个运算比较复杂，涉及平方和开根（FPGA 不擅长），可以采用取近似值计算方法，对于最终结果影响不大。

$$|G| = |G_x| + |G_y|$$

第三步：设置一个阈值 threshold，对数据进行比较然后输出二值图像。

39.2.2.1Sobel 算子卷积运算的原理举例

以如下 6*6 矩阵为例：

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

图 39-4 举例的图像数据

从各研究文献来看，关于卷积核的形式，有不同的观点。有的采用了邻域像素点权重相同的策略，有的借用了高斯滤波的观点，认为和中心点的距离会和中心点产生正相关的效果。而加权计算卷积核算子的形式也并不唯一，甚至

还有筛选 45° 角边缘的卷积核（如下方最后一幅图），最终也派生出如下几种卷积核的形式如下：

1	0	-1
2	0	-2
1	0	-1

-1	0	1
-2	0	2
-1	0	1

1	0	-1
1	0	-1
1	0	-1

-1	-2	-1
0	0	0
1	2	1

2	1	0
1	0	-1
0	-1	2

图 39-5 几种卷积核形式

为了计算方便，这里借用高斯滤波的观点，假设邻域的像素对当前像素产生的影响是不等价的，即引用卷积核为：

1	0	-1
2	0	-2
1	0	-1

图 39-6 举例的卷积核

如果我们需要实现对左上角 3*3 的方框进行卷积运算。

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

图 39-7 从图像数据中提取的卷积矩阵

即实现：

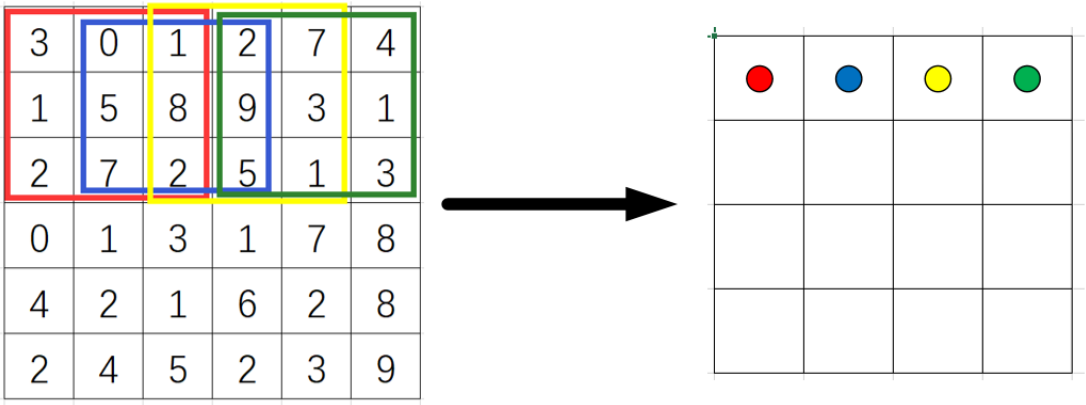
3	0	1		1	0	-1
1	5	8	★	2	0	-2
2	7	2		1	0	-1

图 39-8 举例的被卷积矩阵和卷积核运算示意

卷积运算规则为：将两个矩阵相同位置的数值相乘，然后将这些积相加。例如：采用数学运算得到如上矩阵卷积的计算过程和像素值如下：

```
3×1+0×0+1×(-1)+1×2+5×0+8×(-2)+2×1+7×0+2×(-1)=-12
```

一个 6*6 的像素矩阵中，如果提取 3*3 的矩阵进行 3*3 算子的卷积计算，则通过向右滑动提取框，一行可以提取到 4 个 3*3 的矩阵。同理，提取框向下滑动，也可以提取到 4 列 3*3 的矩阵。因此，6*6 的像素矩阵经过卷积计算完成后，会得到一个 4*4 的新矩阵。



39.2.2.2Sobel 算子卷积运算实现边缘检测的直观认识

首先说明，要想将卷积运算和边缘检测两个概念联系起来，是需要严谨的数学论证和理论分析的。卷积计算只是一种数字处理方法，而边缘检测是经过一种算法需要最终达到的目标。那么利用 Sobel 卷积核进行的卷积计算是否能有针对性的实现边缘检测呢？

好在这些广泛采用的数字图像处理算法，其理论基础已经由图像处理相关领域的专家进行了深度的分析和论证，各位有兴趣的读者可以查阅相关资料自行研究。而我们在这里，为了增强读者利用卷积算法实现边缘检测的认识，可

以对一种极端情况进行举例演示。

假定图像灰度值如下：

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

图 39-9 像素值按区域左右均匀排布

按上面介绍的卷积计算规则，得到计算结果 4*4 矩阵如下：

0	40	40	0
0	40	40	0
0	40	40	0
0	40	40	0

图 39-10 卷积计算后得到的像素值

从直观感受上来认识，原始图像的两侧数据区域，变成了一个中心不变的线状数据阵列。反映到大画幅图像上，这个线状数据阵列就是一条线状边界。

交换原始图像左右区域的位置而卷积核不变，可以发现边界值全部变为了原值的相反数，可以感受到，卷积运算还可以表征亮度变化的方向。

0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

0	-40	-40	0
0	-40	-40	0
0	-40	-40	0
0	-40	-40	0

图 39-11 交换原始图像位置区域求卷积结果

39.2.3模块接口设计

基于以上思想，结合前面的章节对于移位寄存器设计的启示，sobel 边缘检测模块有哪些信号输入输出，也就基本明确了，这里，给出如下 sobel 边缘检测模块端口设计。

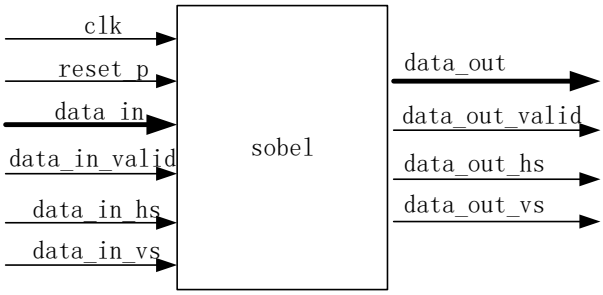


图 39-12 边缘检测模块输入输出接口图

表 39-4 模块端口描述表

端口名称	I/O	端口说明
clk	I	图像像素时钟
reset_p	I	模块复位信号，高电平有效
data_in	I	图像像素数据，数据位宽可自定义，对于灰度图像，位宽为 8
data_in_valid	I	图像像素数据有效标识，为 1 表示当前 data_in 有效
data_in_hs	I	图像行信号
data_in_vs	I	图像场信号
data_out	O	图像处理后数据输出，数据位宽与 data_in 相同
data_out_valid	O	图像处理后像素数据有效标识，为 1 表示当前 data_out 有效
data_out_hs	O	图像处理后行信号
data_out_vs	O	图像处理后场信号

39.2.4实现过程

3*3 模板数据的产生在上节中值滤波处理已经讲解，根据算法 d 饿步骤，只需要计算$|G_x|$，$|G_y|$，然后根据$|G_x|$和$|G_y|$计算出 G，最后将 G 与设置的阈值进行比较得到二值图像结果。具体代码实现如下。

第 1 部分：模块端口定义

```
module sobel
#(
  parameter DATA_WIDTH = 8
)
(
  input                       clk,    //pixel clk
  input                       reset_p,
  input     [DATA_WIDTH-1:0] data_in,
```

```
  input                       data_in_valid,
  input                       data_in_hs,
  input                       data_in_vs,
  input     [DATA_WIDTH-1:0] threshold,

  output reg                  data_out,
  output reg                  data_out_valid,
  output reg                  data_out_hs,
  output reg                  data_out_vs
);
//line data
wire [DATA_WIDTH-1:0] line0_data;
wire [DATA_WIDTH-1:0] line1_data;
wire [DATA_WIDTH-1:0] line2_data;
//matrix 3x3 data
reg  [DATA_WIDTH-1:0] row0_col0;
reg  [DATA_WIDTH-1:0] row0_col1;
reg  [DATA_WIDTH-1:0] row0_col2;

reg  [DATA_WIDTH-1:0] row1_col0;
reg  [DATA_WIDTH-1:0] row1_col1;
reg  [DATA_WIDTH-1:0] row1_col2;

reg  [DATA_WIDTH-1:0] row2_col0;
reg  [DATA_WIDTH-1:0] row2_col1;
reg  [DATA_WIDTH-1:0] row2_col2;
//
reg                   data_in_valid_dly1;
reg                   data_in_valid_dly2;
reg                   data_in_hs_dly1;
reg                   data_in_hs_dly2;
reg                   data_in_vs_dly1;
reg                   data_in_vs_dly2;

wire                  Gx_is_positive;
wire                  Gy_is_positive;

reg  [DATA_WIDTH+1:0] Gx_absolute; //high bit expansion 2bit
reg  [DATA_WIDTH+1:0] Gy_absolute; //high bit expansion 2bit
```

第 2 部分：移位寄存器建立

```
//3xline data
shift_register_2taps
#(
  .DATA_WIDTH ( DATA_WIDTH )
)shift_register_2taps(
  .clk           (clk           ),
```

```
  .shiftin       (data_in       ),
  .shiftin_valid (data_in_valid ),

  .shiftout      (              ),
  .taps0x        (line0_data    ),
  .taps1x        (line1_data    )
);
assign line2_data = data_in;
```

第 3 部分：建立图像数据缓存空间（被卷积矩阵）

```
//----------------------------------------------------
// matrix 3x3 data
// row0_col0   row0_col1   row0_col2
// row1_col0   row1_col1   row1_col2
// row2_col0   row2_col1   row2_col2
//----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p) begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
  else if(data_in_hs && data_in_vs)
    if(data_in_valid) begin
      row0_col2 <= line0_data;
      row0_col1 <= row0_col2;
      row0_col0 <= row0_col1;

      row1_col2 <= line1_data;
      row1_col1 <= row1_col2;
      row1_col0 <= row1_col1;

      row2_col2 <= line2_data;
      row2_col1 <= row2_col2;
      row2_col0 <= row2_col1;
    end
    else begin
      row0_col2 <= row0_col2;
      row0_col1 <= row0_col1;
```

```
      row0_col0 <= row0_col0;

      row1_col2 <= row1_col2;
      row1_col1 <= row1_col1;
      row1_col0 <= row1_col0;

      row2_col2 <= row2_col2;
      row2_col1 <= row2_col1;
      row2_col0 <= row2_col0;
    end
  else begin
    row0_col0 <= 'd0;
    row0_col1 <= 'd0;
    row0_col2 <= 'd0;

    row1_col0 <= 'd0;
    row1_col1 <= 'd0;
    row1_col2 <= 'd0;

    row2_col0 <= 'd0;
    row2_col1 <= 'd0;
    row2_col2 <= 'd0;
  end
end
```

第 4 部分：卷积运算（被卷积矩阵和卷积核进行运算）

```
always @(posedge clk)
begin
  data_in_valid_dly1 <= data_in_valid;
  data_in_valid_dly2 <= data_in_valid_dly1;

  data_in_hs_dly1    <= data_in_hs;
  data_in_hs_dly2    <= data_in_hs_dly1;

  data_in_vs_dly1    <= data_in_vs;
  data_in_vs_dly2    <= data_in_vs_dly1;
end

//-----------------------------------------------------
// mask x          mask y
//[-1,0,1]       [ 1, 2, 1]
//[-2,0,2]       [ 0, 0, 0]
//[-1,0,1]       [-1,-2,-1]
//-----------------------------------------------------
assign Gx_is_positive = (row0_col2 + row1_col2*2 + row2_col2) >=
(row0_col0 + row1_col0*2 + row2_col0);
```

```
assign Gy_is_positive = (row0_col0 + row0_col1*2 + row0_col2) >=
(row2_col0 + row2_col1*2 + row2_col2);

always @(posedge clk or posedge reset_p) begin
  if(reset_p)
    Gx_absolute <= 'd0;
  else if(data_in_valid_dly1) begin
    if(Gx_is_positive)
      Gx_absolute <= (row0_col2 + row1_col2*2 + row2_col2) -
(row0_col0 + row1_col0*2 + row2_col0);
    else
      Gx_absolute <= (row0_col0 + row1_col0*2 + row2_col0) -
(row0_col2 + row1_col2*2 + row2_col2);
  end
end

always @(posedge clk or posedge reset_p) begin
  if(reset_p)
    Gy_absolute <= 'd0;
  else if(data_in_valid_dly1) begin
    if(Gy_is_positive)
      Gy_absolute <= (row0_col0 + row0_col1*2 + row0_col2) -
(row2_col0 + row2_col1*2 + row2_col2);
    else
      Gy_absolute <= (row2_col0 + row2_col1*2 + row2_col2) -
(row0_col0 + row0_col1*2 + row0_col2);
  end
end

//-----------------------------------------------------
//result
//-----------------------------------------------------
always @(posedge clk or posedge reset_p) begin
  if(reset_p)
    data_out <= 1'b0;
  else if(data_in_valid_dly2) begin
    data_out <= ((Gx_absolute+Gy_absolute)>threshold) ? 1'b0 : 1'b1;
  end
end

always @(posedge clk)
begin
  data_out_valid <= data_in_valid_dly2;
  data_out_hs    <= data_in_hs_dly2;
  data_out_vs    <= data_in_vs_dly2;
end
```

```
endmodule
```

完成以上模块设计后，和前面的章节类似，将其嵌入工程设计的图像处理环节，即可实现图像的边缘检测。

39.2.5仿真验证

模块设计完成后，编写仿真验证 testbench 文件，仿真验证代码与中值滤波基本是一样的，将例化的中值滤波模块替换成均值滤波模块即可，阈值 threshold 在模块例化给固定常数 128，具体代码如下。

```
`timescale 1ns/1ns
`define CLK_PERIOD 20

module sobel_tb();

  reg       clk;    //pixel clk
  reg       reset_p;
  reg [7:0] data_in;
  reg       data_in_valid;
  reg       data_in_hs;
  reg       data_in_vs;
  wire      data_out;
  wire      data_out_valid;
  wire      data_out_hs;
  wire      data_out_vs;

initial clk = 1'b1;
always #(`CLK_PERIOD/2) clk = ~clk;

initial begin
  reset_p = 1'b1;
  data_in = 8'd0;
  data_in_valid = 1'b0;
  data_in_hs = 1'b0;
  data_in_vs = 1'b0;
  #201;
  reset_p = 1'b0;
  #200;

  data_in_vs = 1'b1;
  repeat(480) begin
    #500;
    data_in_hs = 1'b1;
    #500;
    repeat(400*2) begin
      data_in_valid = ~data_in_valid;
```

```
      data_in = $random % 256;
      #(`CLK_PERIOD);
    end
    #500;
    data_in_hs = 1'b0;
  end
  data_in_vs = 1'b0;
  #(`CLK_PERIOD);
  data_in_vs = 1'b1;

  #2000;
  $stop;
end

sobel
#(
  .DATA_WIDTH ( 8 )
)sobel
(
  .clk            (clk           ),    //pixel clk
  .reset_p        (reset_p       ),
  .data_in        (data_in       ),
  .data_in_valid  (data_in_valid ),
  .data_in_hs     (data_in_hs    ),
  .data_in_vs     (data_in_vs    ),
  .threshold      (8'd128        ),

  .data_out       (data_out      ),
  .data_out_valid (data_out_valid),
  .data_out_hs    (data_out_hs   ),
  .data_out_vs    (data_out_vs   )
);

endmodule
```

运行仿真，任意找到一个仿真地方进行放大观察，仿真波形如下图 39-13。

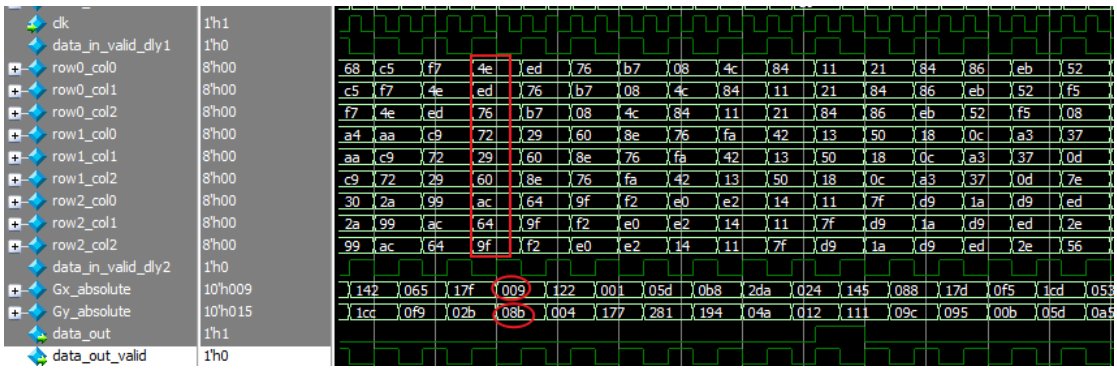


图 39-13 仿真波形图

从上面波形数据可以看出，当前 3*3 的模板图像数据如下图 39-14。

4e	ed	76
72	29	60
ac	64	9f

图 39-14 3*3 模板图像数据

根据计算公式可得，

$|G_x|$=(-1*0x4e+0*0xed+1*0x76-2*0x72+0*0x29+2*0x60-1*0xac+0*0x64+1*0x9f) = 0x9,

$|G_y|$=(1*0x4e +2*0xed+1*0x76+0*0x72+0*0x29+0*0x60-1*0xac-2*0x64-1*0x9f) = 0x8B,

则|Gx| + |Gy| = 0x94，大于设置的固定阈值’d128，最终处理后输出数据为二值图像数据的 0。仿真中 data_out 与该计算结果一致。验证了设计的正确性。

39.3顶层模块边缘检测阈值设定

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。Sobel 模块涉及到阈值的设定，为了能动态的改变阈值，通过一个按键（S1）控制其累加。初值设置为’d128，然后按键每按下一次，阈值加 8。具体代码在顶层中实现如下：

```
always@(posedge loc_clk50m or posedge g_rst_p)
  if(g_rst_p)
    sobel_threshold <= 'd128;
  else if(~key0_state && key0_flag)
    sobel_threshold <= sobel_threshold + 'd8;
```

顶层的仿真与“基于 FPGA 的灰度图像中值滤波的设计实现”类似，这里就不做详细讲解，读者自己建立仿真文件完成顶层的仿真。

39.4系统板级测试

39.4.1Sobel 算子边缘检测工程的 TFT 显示

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。

由于本工程涉及管脚数量较多，我们也就不以列表的形式展示本章例程的管脚绑定了。如需参考，读者可以直接从例程中复制本工程的管脚绑定 cst 文件到自己设计的工程中对自己的设计进行验证。

对工程的管脚和时钟进行约束后，生成 Bit 文件。上板调试硬件平台基于高云开发板，使用一根数据线，一端接入高云开发板的的 UART 接口，另一端接入 PC 机的 USB 口，显示屏使用的是 TFT5.0 寸屏幕，开发板连接示意图与“基于 DDR3 的串口传图帧缓存系统”一样。

硬件连接好并上电后然后下载生成的 Bit 文件。下载完成后，开发板上 LED0~LED2 会亮，TFT5.0 寸屏上显示花屏状态。

双击打开串口传图工具，通过点击“打开图片”按钮设置图片存放路径；图片宽度设置成与显示屏分辨率宽度一半，高度设置成与显示屏分辨率高度一致（TFT5.0 寸屏是 800*480，上位机上宽度设置为 400，高度设置为 480；TFT4.3 寸屏是 480*272，上位机上宽度设置为 240，高度设置为 272）。

下图是待传的图片。

图 39-15 待传的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。注：这里提供的上位机要求图片为位深度为 24 的 bmp 格式图片，图片的宽度和高度需要为 400*480（插 5 寸屏情况下）或 240*272（插 4.3 寸屏情况下）。

串口波特率设置与 FPGA 串口接收波特率一致（FPGA 串口接收波特率是 2000000bps），串口端口号根据实际连接电脑的串口号进行设置（这里使用的是 COM6），设置好传图工具后，点击“连接设备”。

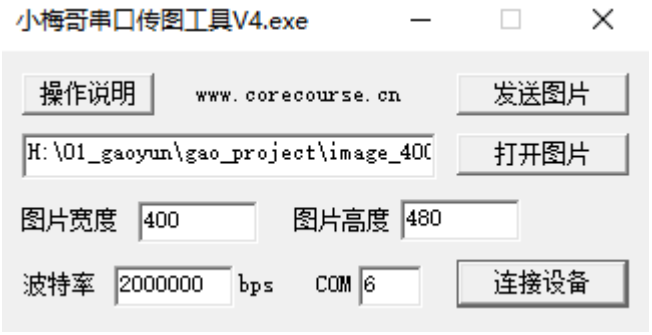


图 39-16 串口传图工具连接开发板配置

连接成功后，“连接设备”会变成“断开设备”，通过点击“发送图片”按钮开始发送图片数据。

图 39-17 启动传图

传图过程中，可以看到 TFT 屏上开始显示发送的图片，图片传送完成后，TFT 屏显示效果如下。下面图片分别是不同阈值下的效果图。

图 39-18 阈值 128 效果图（TFT）

图 39-19 阈值 64 效果图（TFT）

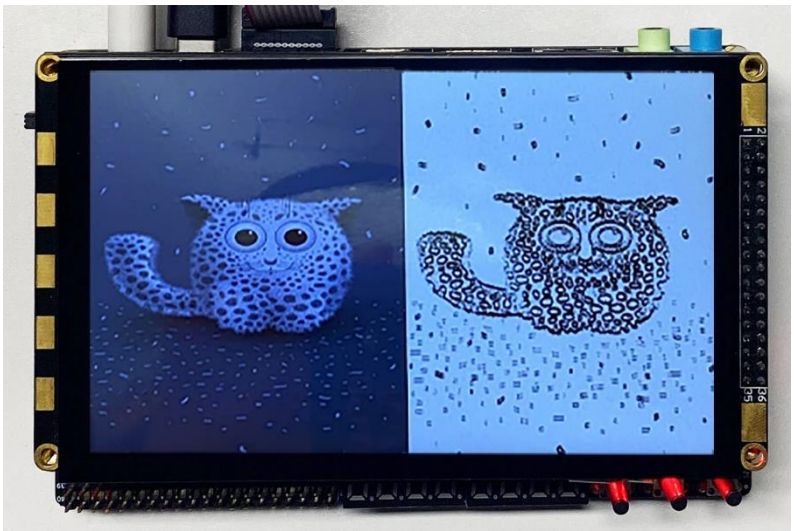


图 39-20 阈值 192 效果图（TFT）

通过对比可以看出，TFT 屏上左右两边分别显示的是彩色图像灰度化图像数据和经过 sobel 算子边缘检测处理之后的二值图像。随着阈值的增大边缘检测的细节在不断减少，大家可以自己改变阈值进行实验对比。阈值修改可通过按键 S1 去增加，每按下一次，阈值增加 8。每次在修改阈值后需要重新通过串口传一次图片数据。

39.4.2Sobel 算子灰度图像边缘检测工程添加 HDMI 显示

结合前面讲解的 HDMI 章节知识点，前面讲解的 HDMI 章节知识点，该显示内容还可以通过 HDMI 接口输出到 HDMI 显示器上。关于本小节对工程输出端口改造的方法，可参考串口传图章节对应：串口传图工程添加 HDMI 显示的内容，这里就不再重复讲解了。

最终，添加 HDMI 工程显示的效果如下：

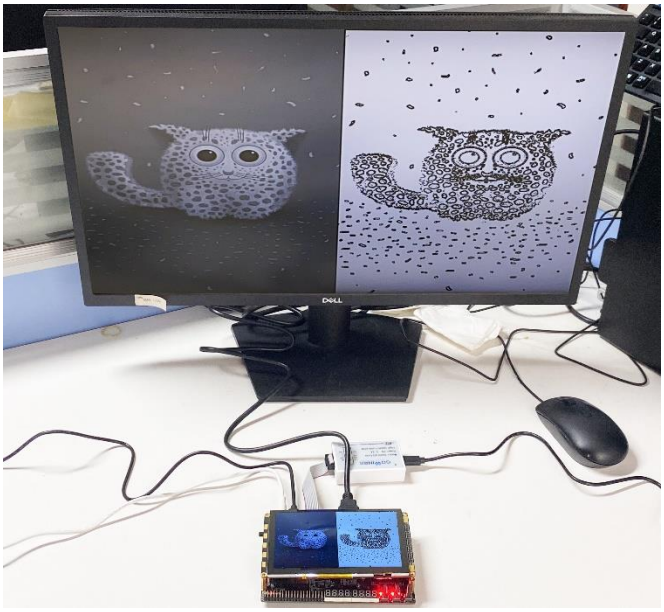


图 39-21 阈值 128 效果图（HDMI）

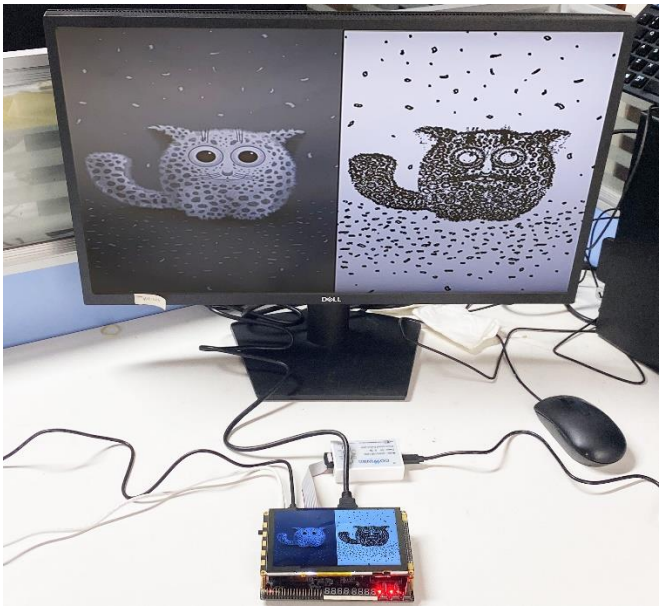


图 39-22 阈值 64 效果图（HDMI）

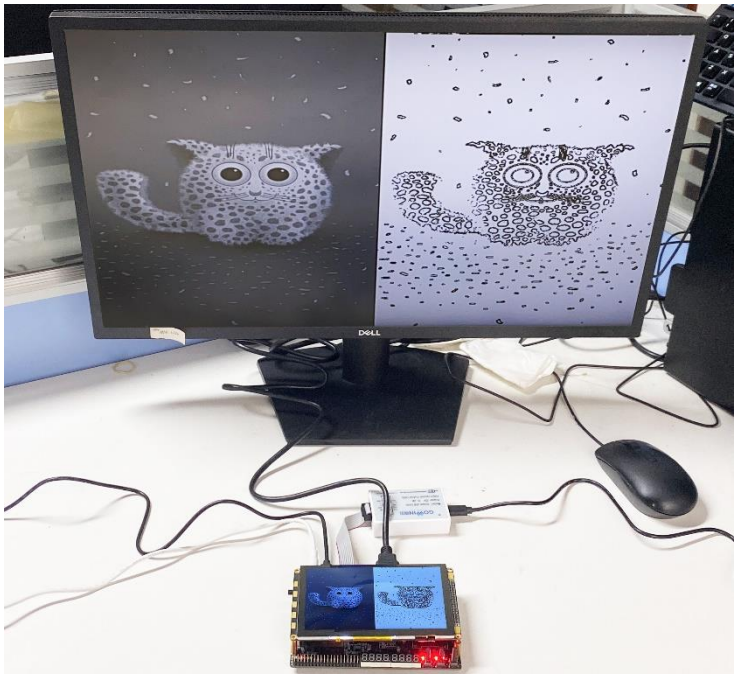


图 39-23 阈值 192 效果图（HDMI）

至此，Sobel 算子边缘检测在高云开发板上的设计及验证就成功完成了。

到这里，几种典型的图像处理方案，我们就讲解完成了。希望基于以上讲解的内容，各位读者加以认真学习和领会。

39.5总结

边缘检测可以帮助用户识别图像的边界信息，捕捉图像的关键数据。Sobel

边缘检测是通过卷积计算实现的。通过将 Sobel 算子模块插入灰度图像 FPGA 显示的中间环节，就可以实现 Sobel 边缘检测计算的图像显示。希望读者对 Sobel 算法的 FPGA 实现方法加以理解。

40 基于 OV5640 摄像头理论知识讲解

工程源码	|----02_设计实例 |----ch40_ov5640_init_DVP
相关视频课程	
说明	

40.1OV5640 摄像头介绍

OV5640_V5（V5 是版本号，下面均以 OV5640 表示该产品）是芯路恒科技推出的一款高性能 500W 像素高清摄像头模块。该模块采用 OmniVision 公司生产的一颗 1/4 英寸 CMOS QSXGA（2592*1944）图像传感器 OV5640，配合高质量的光学镜头及为实现更高性能而精心设计的 PCBA，使该模块拥有了尽可能高的成像质量。

OV5640 模块的特点如下：

- 1.4μm*1.4μm 像素大小，并且使用 OmniBSI 技术以达到更高性能（高灵敏度、低串扰和低噪声）
- 自动图像控制功能：自动曝光（AEC）、自动白平衡（AWB）、自动消除灯光条纹
- 自动黑电平校准（ABLC）和自动带通滤波器（ABF）等
- 支持图像质量控制：色饱和度调节、色调调节、gamma 校准、锐度和镜头校准等
- 标准的 SCCB 接口，兼容 I^2C 接口
- 支持 RawRGB、RGB(RGB565/RGB555/RGB444)、CCIR656、YUV(422/420)、YCbCr（422）和压缩图像（JPEG）输出格式
- 支持 QSXGA（500W）图像尺寸输出，以及按比例缩小到其他任何尺寸
- 支持图像缩放、平移和窗口设置
- 支持图像压缩，即可输出 JPEG 图像数据
- 支持数字视频接口（DVP）
- 自带嵌入式微处理器
- 集成 LDO，仅需提供 3.3V 电源即可正常工作

40.2硬件电路说明

OV5640 模块对用户提供一个 20 针的排针接口，通过该接口可直接插接到芯路恒科技所有的 FPGA 开发板（包含 AC620、AC6102、AC601、Starter 开发板等）上, 并提供有相应的应用例程，用户可以直接在这些开发板上，使用提供的例程对模块进行测试，同时，用户也可以在理解清楚我们提供的设计例程的基础上，进行二次开发。

模块的 2*10 接口信号图如下图 40-1 所示：

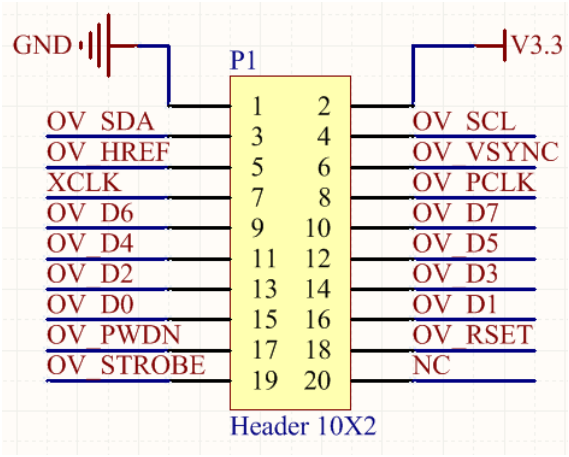


图 40-1 OV5640 摄像头模块接口图

下表 40-1 为该接口上各个信号的功能介绍。

表 40-1 摄像头模块各个信号功能介绍

信号	作用描述
V3.3	模块供电脚，接 3.3V 电源，为了尽可能的保证成像质量，该电源需要拥有较小的纹波。
GND	模块地线
OV_SCL	SCCB 通信时钟信号，对应 I2C 总线的时钟 SCLK 信号，该引脚需要额外提供上拉电阻
OV_SDA	SCCB 通信数据信号，对应 I2C 总线的时钟 SDAT 信号，该引脚需要额外提供上拉电阻
OV_D[7:0]	8 位数据输出，数据的数据内容根据不同的模式设置有所不同，需要具体情况具体分析
XCLK	OV5640 基本工作时钟输入脚，默认 24MHz，OV5640 片上的 PLL 会将该时钟倍频到一定频率后供片上的各个功能电路工作
OV_PCLK	像素时钟输出，根据输出模式和分辨率不同，该时钟频率值也不一样
OV_PWDN	掉电使能(高有效)，正常工作情况下需要将该引脚拉为低电平
OV_VSYNC	帧同步信号输出，每幅图像开始输出前该信号会产生一个高脉冲，以用作帧同步功能
OV_HREF	行同步信号输出，该信号在 OV_D[7:0]数据线输出一行图像数据的过程中一直保持高电平。
OV_RESET	复位信号(低有效)
OV_STROBE	预留

注意：OV5640 芯片 DVP 接口本身拥有 10 位的数据线，可以输出 10 位的 RAW 数据，但是在大多数情况下，使用高 8 位数据即可，因此模组在设计时，仅使用了 OV5640 芯片的 D9~D2 高 8 位，映射到模组上的 OV_D7~OV_D0。

虽然 OV5640 模块设计时主要针对芯路恒科技出产的 FPGA 开发板的 CMOS 摄像头接口，但是从理论上来说，任意一个开发板，只要是带有符合摄像头模块接口，都能连接本模块并直接使用。即使 CAMERA_SCLK 和 CAMERA_SDA 脚上没有连接物理上拉电阻，在 FPGA 的管脚中，也可以通过设置开启 FPGA 的 IO 片上上拉电阻实现该功能。所以物理电路上没 R28 和 R29 这两个电阻也是可以解决的。

另外，建议非极端情况下不要尝试使用杜邦线飞线与 FPGA 连接，由于 OV5640 的输出数据速率较高，使用杜邦线飞线的方式无法保证信号质量，容易影响成像效果。

高云开发板需要通过 40pin 的接口接 OV5640 摄像头，可以通过转接板接，也可以直接使用我们的双目摄像头，下图为 OV5640 双目摄像头模块连接小梅哥高云开发板的示意图：

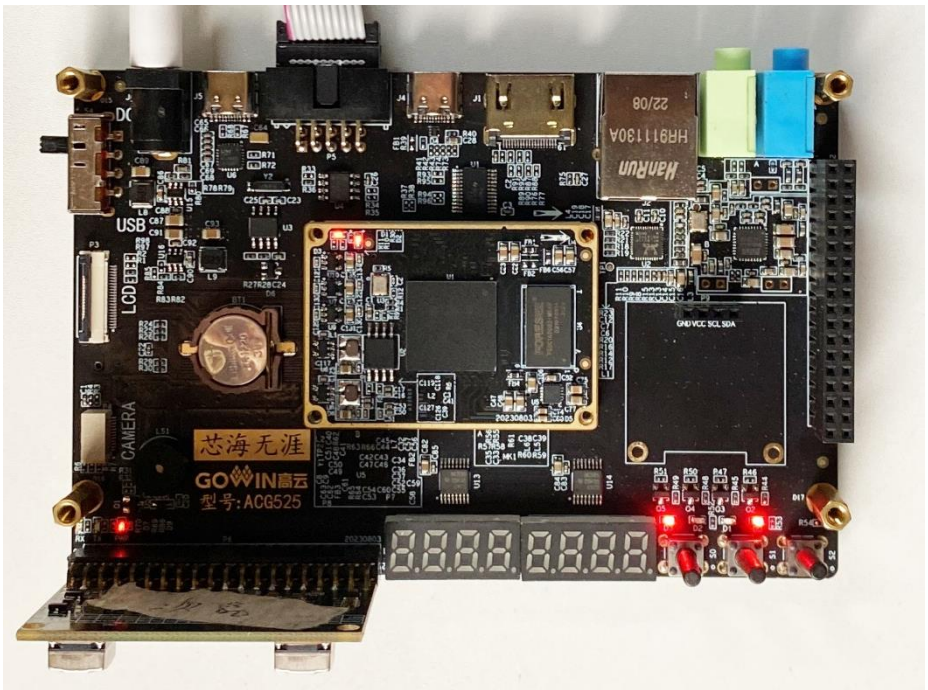


图 40-2 OV5640 双目摄像头连接高云开发板示意图

40.3OV5640 工作原理

40.3.1COMS 图像传感器成像原理

OV5640 是一个典型的 CMOS 图像传感器，作为一个图像传感器，其主要作用就是将现实中的各种光线转换为数字系统能够识别的数字信号，光线中三元色各个颜色的强度本身是模拟信号，所以图像传感器的最基本的原理就是进行模数转换，将光线这个模拟量转换为数字信号。

仅仅有模数转换功能还不够，我们还得先搞清楚另一个问题——光线是一种怎样的模拟量。

光线是一种怎样的模拟量呢，这个在初中物理中已经有过详细的介绍。我们都知道，自然界中的光，实际上是三种基本单色光的组合，这三种基本单色光为红（RED）、绿（GREEN）、蓝（BLUE），我们称之为三原色。通过将这三种基本颜色按照不同的比例混合，就可以得到其他的任意颜色。例如纯黄色是由红色和绿色按照一比一的比例混合得到的，蓝色量为 0。下图 40-3 为三种颜色混合得到几种常见颜色的示意图。

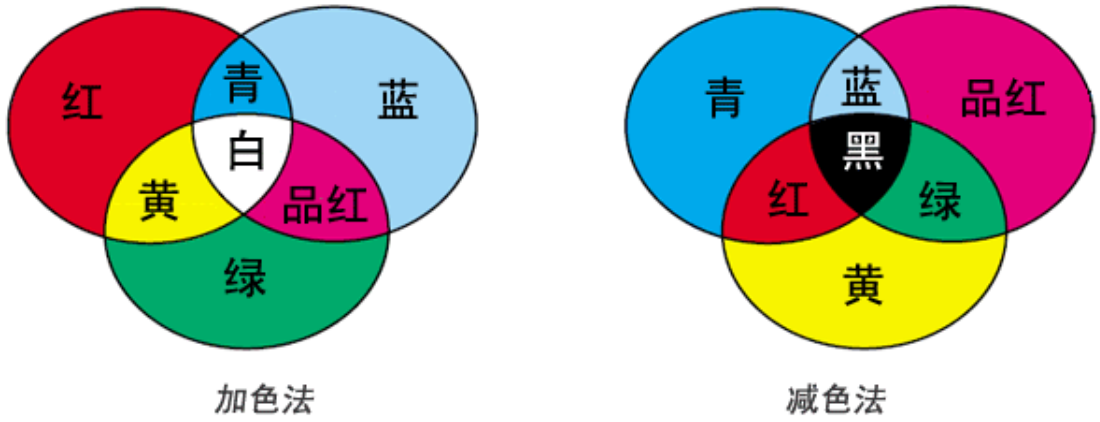


图 40-3 加色法和减色法得到常见颜色示意图

既然已经知道，每一束光线都可以理解为三原色按照不同比例混合得到的效果，那么我们只要想办法知道该束光线的三原色的比例，然后用颜色加比例的表示方法，就能唯一确定这束光线的最终颜色了。所以图像传感器里面所谓的模数转换，实质就是对一束光线的三原色的强度进行转换，将三原色中每一种颜色的强度转化为数字信号。再用颜色加数字的方式来表示该束光线的真实颜色。

这里，假如我们对三原色中的每一种基本颜色的强度都分为 256 级，那么每一种基本颜色的强度就可以用一个 8 位的数字来表示，0 表示该颜色强度最弱，

或者说无该颜色分量，255 表示该颜色强度最强。这样一来，我就可以用一个 24 位的数字来唯一表示该光线的颜色了。下表 40-2 为上图中几种颜色的对应三原色的数值。

表 40-2 常见颜色数值表

	红	绿	蓝	黄	品红	青	白	黑
红	255	0	0	255	0	255	255	0
绿	0	255	0	255	255	0	255	0
蓝	0	0	255	0	255	255	255	0

这种表示颜色的方法就是最常见的 RGB888 格式。所谓 RGB888 就是使用 3 个 8 位的数据表示一种颜色，其中高八位表示红色分量，中八位表示绿色分量，低八位表示蓝色分量，上述 8 种颜色用 RGB888 格式表示就如下表 40-3 所示：

表 40-3 RGB888 格式颜色数值对应表

	红	绿	蓝	黄	品红	青	白	黑
RGB888	0xff0000	0x00ff00	0x0000ff	0xffff00	0x00ffff	0xff00ff	0xffffff	0x000000

通过上面的分析我们就可以知道，只要模数转换器能够对每一束光的三原色的强度进行转换得到数字信号，就能唯一表示该颜色了，但是接下来，另一个问题又出现了。如何才能对一束自然光中的三原色的强度分别进行模数转换呢？这就涉及到对一束光的三原色分离。

所谓对光的三原色分离就是通过某种手段，将该束光线中的三种颜色分别独立提取出来，当三种颜色都独立的提取到之后，就能使用模数转换器对该颜色的强度进行转换了。

三原色分离的原理其实非常简单，就是使用单色滤光片。

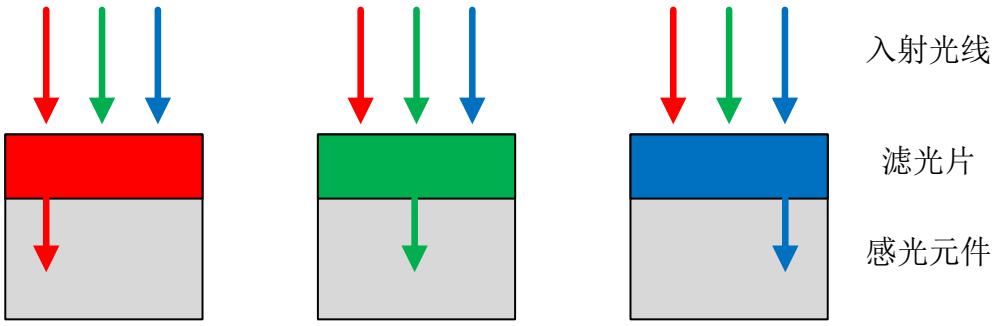


图 40-4 三色分离原理实现

如上图，通过加入滤光片，就能让对应颜色的光线通过滤光片到达感光元件，通过这种方式，只需要三个不同颜色的滤光片，就能对一束入射的复合光线进行分离，得到三原色，然后使用模数转换器对感光元件感应到的单色光的光照强度进行转换，就能得到该单色光的强度数字值了。

但是，通过上述分析我们也发现一个事实，那就是一个感光元件只能对一种颜色的光进行感应，如果要对一束光线中的三种颜色分别感应，就需要三个感光元件。如果对应到 CMOS 图像传感器里面的概念来说，这每一个滤光片加感光元件都是一个像素。注意这个概念，每一个滤光片加感光元件都是一个像素，那么从反面来说就是，每个像素都只能感应一种颜色的光线。所以，下次看到某某图像传感器宣传其有多少多少万像素大小时，作为半专业人士的我们，要在心里默念，这只是物理像素个数，实际上每个像素只能感应一种颜色。

既然每个像素只能感应一种颜色，那么新一个问题又来了——如何才能得到某束光线的三种颜色分量的值呢？答案就是使用至少 3 个像素，每个像素感应该束光的一种颜色，然后三个像素就能把该束光的三种颜色分量全提取出来了。但是，实际上使用 3 个像素来提取一束光的三种颜色分量，理论上可行，但是实际上却存在 3 个像素间摆放位置的物理限制，首先是如何让该束光线能够均匀的照射到感应三种颜色分量光线的像素上的问题，其次还要知道，一个图像传感器，并不是只感应一束光，是要感应成千上万束细小的光束，每一束光线都需要有对应的像素组来提取其三种颜色分量。因此，像素和像素之间的物理摆放问题也是需要认真考虑的问题。同时，还得兼顾考虑这种摆放模式下图像传感器的可生产性问题。

正是为了解决这些问题，诞生了著名的拜尔（以下简称 Bayer）矩阵。Bayer 矩阵定义，感光像素矩阵中，奇数行间隔放置绿色和红色感应像素，偶数行间隔放置绿色和蓝色感应像素，奇数列间隔放置绿色和蓝色感应像素，偶数列间隔放置红色和绿色感应像素，其输出数据格式如下图 40-4 所示。

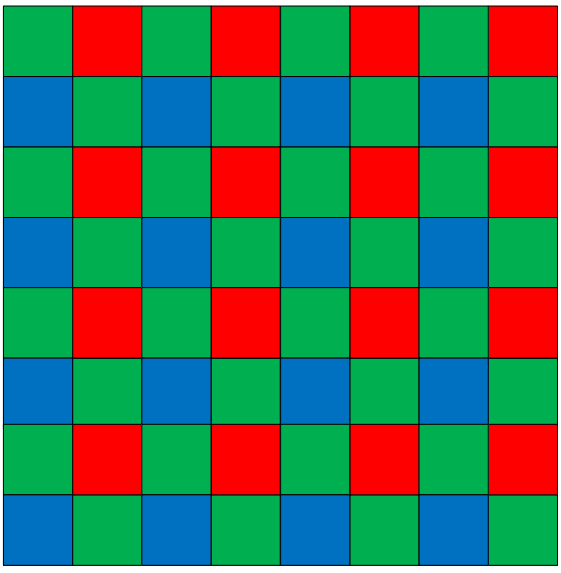


图 40-5 bayer 颜色矩阵

根据这种分布规律，在Bayer矩阵中，以相邻的四个像素作为一组，在该组中，有两个感应绿色分量的像素呈对角分布，另外两个像素则分别对应感应红色分量和感应蓝色分量。通过这样一种方式，我们可以发现，任取一个像素，其与相邻的 3 个像素组成的矩阵中，总符合这样的规律。不同的只是三种颜色的像素所在的位置的差异。下图 40-6 为图像传感器手册中给出的像素物理分布图。

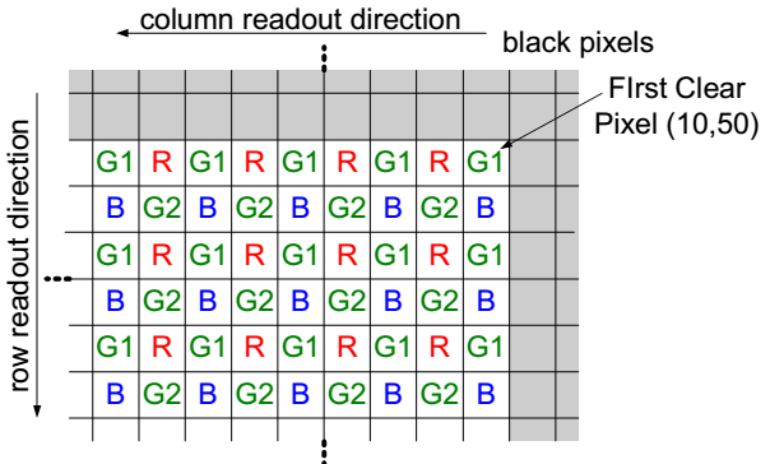


图 40-6 像素物理分布图

而且通过对上述 8*8 的矩阵进行分析发现，在整个矩阵中，像素的位置关系有且只有下面四种情况：

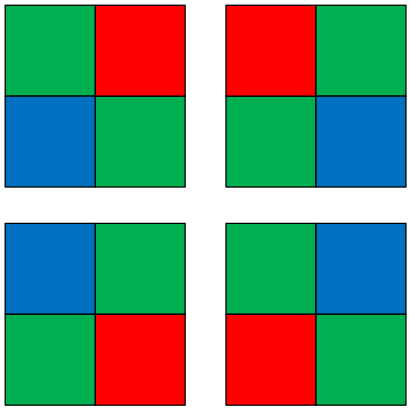


图 40-7 8*8 矩阵像素位置关系

所以，在实际颜色提取时，需要通过数据转换，将相邻四个像素的数据通过插值算法合并为一个 RGB 像素颜色，此种转换算法名为 RAW2RGB，这里取左上角四个像素点的数据为例，具体颜色转换算法如下图 40-8 所示：

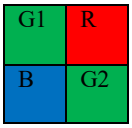


图 40-8 RAW2RGB 算法

保留相邻四个像素中的红色和蓝色分量，而对两个绿色分量求平均，得到新的绿色分量，此三种颜色分量组成一个新的 RGB 格式的像素，新的像素色彩组成如下图 40-9 所示：

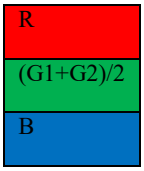


图 40-9 新的像素组成

按照这样的思路，整个图像传感器中的每一个像素都可以以不同的角色参与 4 次运算，并最终得到 4 个 RGB 颜色值，所以理论来说，还是可以认为一个图像传感器有多少个物理像素，就能得到多少个 RGB 格式的像素值，虽然每个物理像素都只能感应一种颜色。但是进过插值运算后，其能输出 RGB 的数据格式的像素数量还是等于其物理像素个数的。

有了 Bayer 像素矩阵后，只需要使用一个模数转换器，依次对每一个像素的感光元件感应到的模拟量进行转换并输出，就能得到一幅完整图像的原始像素信息了。下图 40-10 为 OV5640 图像传感器的功能框图。

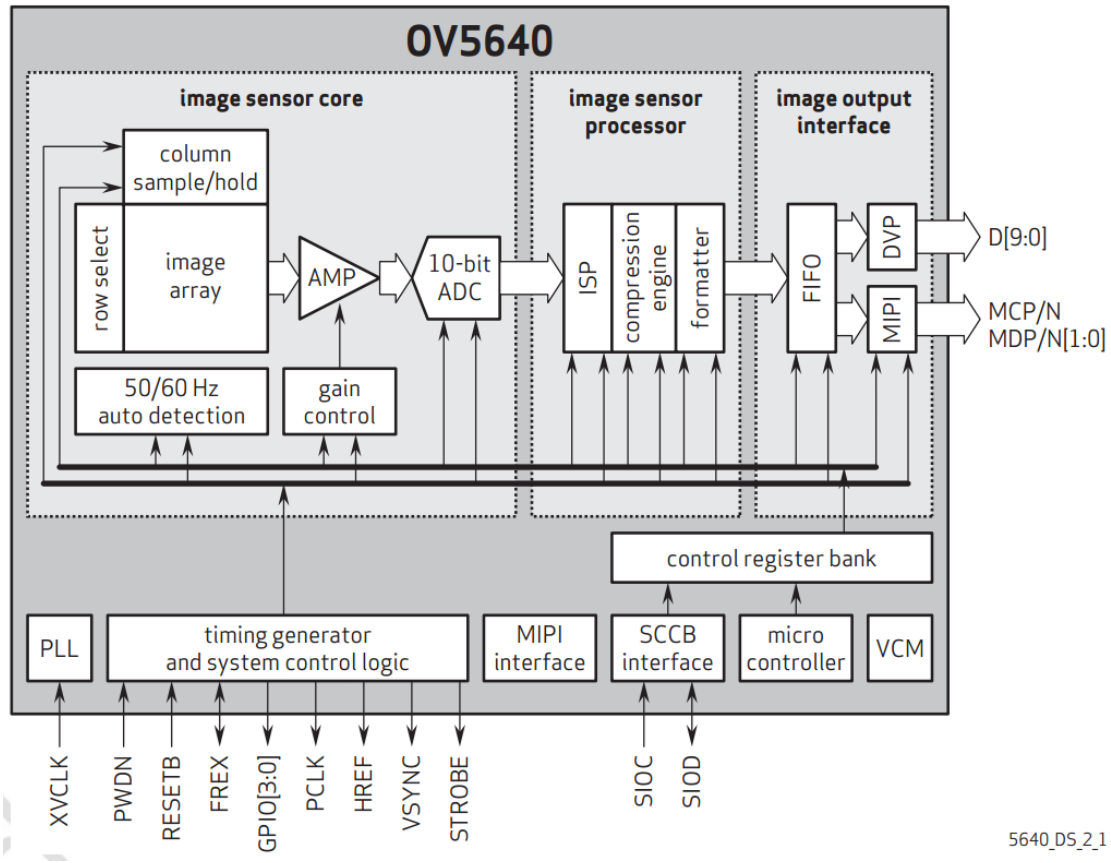


图 40-10 OV5640 传感器功能框图

通过该图可以看到，该图像传感有一个基本的像素矩阵，在图中名为 image array，该像素矩阵共有 2592*1944 个物理像素，2592*1944=5,038,848，这也就是我们常说的 500 万像素的由来。在像素矩阵外围，有一个 row select 功能模块来选择当前输出哪一行的像素，和一个 column sample/hold 电路来依次采样每行像素中的每一个像素的感光元件感应结果（模拟信号）并输出到信号放大器（AMP），经由 AMP 对该信号放大之后，送给 10 位的模数转换器（10-bit ADC）进行模数转换。

至此，关于 OV5640 的第一大基本功能——成像和采样原理，就介绍清楚了。

40.3.2OV5640 数字信号处理流程

40.3.2.1影像处理器（ISP）

在经过前一阶段的图像感应核心（image sensor core）完成图像的模数转换之后，就得到了该图像的原始图像数字信息。但是实际我们在用相机的时候都知道，拍摄时我们可以根据不同的环境亮度调整对比图，调整亮度，等等，这些都是由一种名为影像处理器（Image Signal Processor，简称 ISP）的功能电路实现的。ISP 的功能包括但不限于以下内容：

扣暗电流（去掉底电流噪声）、线性化（解决数据非线性问题）、shading（解决镜头带来的亮度衰减与颜色变化）、去坏点（去掉 sensor 中坏点数据）、去噪（去除噪声）、demosaic（raw 数据转为 RGB 数据）、3A（自动白平衡，自动对焦，自动曝光）、gamma（亮度映射曲线、优化局部与整体对比度）、旋转（角度变化）、锐化（调整锐度）、缩放（放大缩小）、色彩空间转换（转换到不同色彩空间进处理）、颜色增强（可选，调整颜色）等。

实际上，在不同的应用系统中，ISP 的功能差异也比较大，CMOS 图像传感器中也提供了有 ISP，对采样的图像数据进行一定的处理。以弥补图像传感器部分本身可能存在的某些误差和缺陷，同时也能对图像进行一定程度的增强。在很多专业的图像应用领域，都会使用独立的 ISP 对图像进行二次处理，以得到需要的效果。

40.3.2.2压缩引擎（Compression Engine）

对于 500W 像素的图像传感器，其一幅图像总共有 500 万个像素，即使每个

像素只用 8 位数据，表示一种基本色，一幅图像的数据量也有 500 万个字节，也就是 5MB，更何况我们日常使用时，大多使用的是 RGB888 的图像格式，每个像素的颜色用 3 个字节表示，所以如果要按照这种数据量来计算，一幅 500 万像素的 RGB888 格式的图像，其数据量将高达 15MB，这么大的数据量无论是在存储到存储器中时对存储器容量的占用还是使用各种通信接口进行传输时对通信带宽的占用，都是非常大的。因此，为了解决这个问题，在很多应用场合中，例如相机拍照、视频录制、网络摄像头等应用中，大多使用压缩算法对原始数据进行压缩，压缩后的图片数据量会明显小于原始图像数据量，能够有效的降低对存储器或通信带宽的占用。对于图片压缩，使用的最普遍的算法就是 JPEG 压缩。JPEG 压缩算法具有压缩率高，图像失真小等特点。目前大部分数码相机、手机等拍摄得到的照片都是 jpeg 压缩后的图片。

由于 jpeg 压缩算法对应用处理器的运算能力较高，为了满足更多应用场合对图片压缩功能的需求，OV5640 提供了一个内置的图像压缩引擎，能够将图像传感器采集到的图像数据按照 JPEG 标准进行压缩，然后再将压缩之后的图像数据输出。通过这种片上图像压缩引擎，明显降低了基于 OV5640 的应用系统中对应用处理器的算力需求。如果设置 OV5640 输出使用压缩引擎压缩后的图像，那么应用处理器只需要将 OV5640 输出的每帧图像中的所有数据接收并按顺序保存到一个.jpg 文件中，得到的就是一张能够直接在电脑上直接打开的 jpg 图像。

图像压缩引擎作为 OV5640 中一个标配功能，我们在实际应用时可以根据需求选择是否需要使用。例如对于网络传输，图像存储类应用，压缩相对来说更有益处，而对于实时图像处理类应用来说，则使用未经压缩的图像数据将更加方便高效。

40.3.2.3格式化处理器（Formatter）

在经过图像传感器，模数转换器和 ISP 之后，得到的数据依旧是原始的 RAW 格式，这样的图像数据是无法直接在数字显示屏上显示的，需要进行 RAW 转 RGB 算法，但是 RAW 转 RGB 算法又会占用较高的应用处理器的计算能力。另外，在很多应用中，不仅对数据格式有要求，对数据量也有较高的要求，例如在某些对图像质量要求不高的场合，无需使用 RGB888 这种真彩色，使用 RGB565（用 16 位数据表示一个颜色，高 5 位表示红色，中 6 位表示绿色，低 5 位表示蓝色）这种伪真彩色表示方法就能满足需求。而使用 RGB565 模式就能比 RGB888 模式降低 1/3 的数据量。同时，在一些应用领域，如图像处理、

图像识别，一般使用亮度和色度分离的 YUV 格式更加方便，而 YUV 格式有包括无损的 YUV444 格式和有损的 YUV422 以及 YUV420，所以为了满足不同应用场景的需求，OV5640 内置了一个图像格式转换器，能够将原始 RAW 数据转换为 RGB888、RGB666、RGB565、RGB555、YUV444、YUV422、YUV420 等。这样，当开发应用系统时，就能根据应用处理器的运算能力和对图像格式的实际需求灵活选择合适的图像格式输出。

40.4OV5640 数字接口

OV5640 是一个图像传感器，作为一个图像传感器，其最基本的工作就是采集图像数据并输出，所以就存在输出数据接口。而同时，在前面介绍格式转换功能模块的时候也说过，为了支持不同的应用场景，OV5640 可以输出不同格式的图像数据，这就意味着 OV5640 的一些工作模式是需要在实际使用的过程中去现场定义的。因此必然存在着 OV5640 和应用处理器交互一些控制信息的情况，事实上，不仅仅是格式转换模块，OV5640 中的几乎所有模块都有众多的寄存器来确定其工作参数。所以，对于一个图像传感器，一般都存在两类接口，一个是用于图像数据流输出的数据流接口，一个是用于接收应用处理器各种设置参数的控制接口。

40.4.1工作时钟

首先来说，OV5640 是一个典型的模数混合器件，既然有数字电路，就一定需要有基本的工作时钟。OV5640 的时钟从 XCLK 引脚输入，该时钟默认固定频率为 24MHz。要想 OV5640 能够开始工作，提供该 XCLK 时钟必不可少。无论是数据流接口还是控制接口，正常工作的前提就是 XCLK 已经稳定的供应。

40.4.2数据流接口

对于图像传感器来说，常见的数据流接口包括 DVP 接口、LVDS 接口、CSI（MIPI）接口。

40.4.2.1DVP 接口

DVP 接口是最为经典的图像数据流接口，其本质是一个包含控制信号和数据信号的并行数据端口，使用单端 TTL 电平进行通信。DVP 接口包括时钟信号（PCLK）、行有效信号（HREF）、场同步信号（VSYNC）、8 位或更多位的数据

信号。图像传感器使用 DVP 接口与应用处理器连接示意图如下图 40-11 所示：

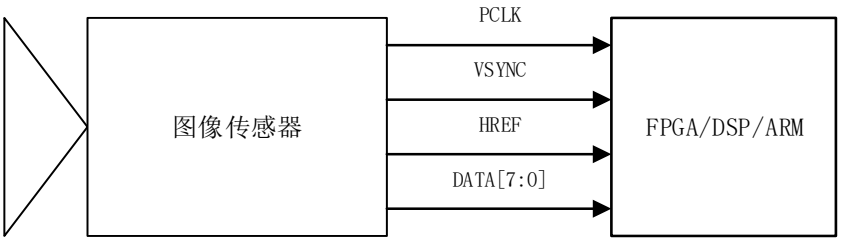


图 40-11 DVP 接口与应用处理器示意图

VSYNC 信号每个高脉冲标志着新一帧图像的开始，在 HREF 为高电平期间，每个 PCLK，8 位 DATA 数据线输出一个数据。每个数据的意义根据当前设置的输出图像格式不同而有所差异。下图为OV7670输出标准VGA分辨率（640*480）图像时候的时序图。（由于 OV5640 手册中没有提供这个时序图，因此截取了 OV7670 中的图示，两者从原理上是完全一致的）。

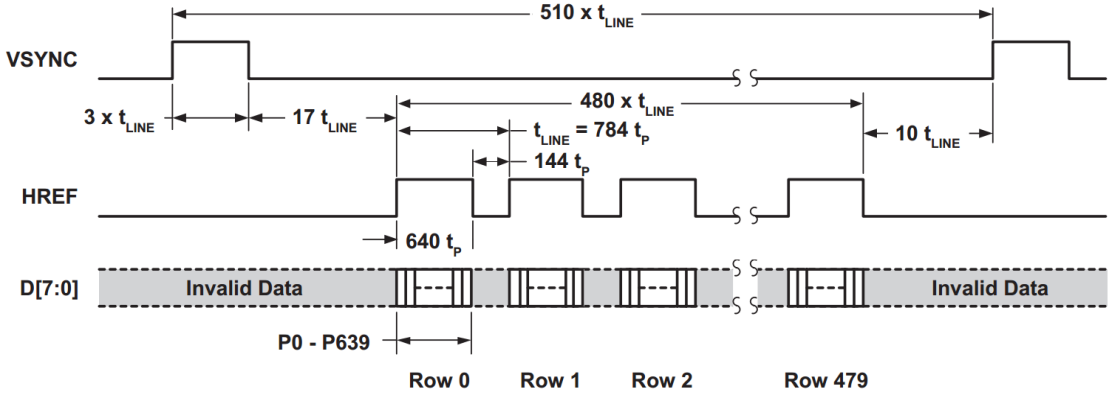


图 40-12 OV7670 输出标准 VGA 分辨率（640*480）

上图中，tp 为每一个像素所占用的时钟周期个数，如果输出的是原始的RAW格式，则tp＝1*PCLK，如果输出的是RGB或者YUV格式，则tp=2*PCLK。也就是说，如果输出的是 RGB 或者 YUV 格式，则每行（Row）有 640*2 个字节的数据输出。

40.4.2.2LVDS 接口

随着图像传感器的像素密度越来越高，其输出数据量也大幅增加，由于传统的 DVP 接口采用的是单端 TTL 并行接口，这种接口模式 IO 翻转速率上限受限，当数据量超过一定的值之后，DVP 接口就无法再满足全部数据的传输。为此，能够显著提升数据传输速率的低压差分串行传输接口（LVDS）就应用到了图像传感器的数据流接口上。LVDS接口采用差分电平传输，每个信号都使用两

根信号线传输数据，接收端通过这两根信号线之间的电压差来确定接收到的数据是 0 还是 1，采用差分传输方式能够显著提升信号线的抗干扰能力，所以 LVDS 的传输速率相对较高。图像传感器使用 LVDS 接口传输图像信息，不仅能满足高带宽的通信需求，还能提高信号传输的抗干扰能力。目前 LVDS 接口的图像传感器主要用于工业相机等场合，索尼的 imx290、imx327 等图像传感器都支持使用 LVDS 作为数据流传输接口。LVDS 接口传输的数据内容实质也就是将 DVP 接口的各个信号编码到 LVDS 传输协议中传输的，传输的数据本质还是 DVP 接口的那些信号和时序。

40.4.2.3CSI 接口

CSI 接口属于 MIPI 接口的一种，全称为摄像机串行接口（Camera Serial Interface），CSI 接口物理层使用 LVDS 接口传输，但是在数据内容层发生了变化。CSI 接口不仅能够传输数据流内容，还能传输控制数据，目前 CSI 接口在多媒体数字终端（手机、平板电脑）等设备中应用非常广泛。

无论是 DVP 接口，还是 LVDS 接口、CSI 接口，其最终的目的都是将数据输出给应用处理器。具体使用什么接口需要看应用处理器的连接能力。这其中，DVP 接口对应用处理器的 IO 能力要求较低，只需要能支持 TTL 电平即可。而 LVDS 接口则需要 IO 口能够提供相应的 LVDS 电平以及数据传输能力。CSI 接口，不仅需要应用处理器能够接受 LVDS 电平，还需要能够解码 CSI 协议。现代的各种多媒体处理器大多自带 CSI 功能接口硬件，因此能够直接连接 CSI 接口的图像传感器。而 FPGA 一般支持 DVP 接口和 LVDS 接口，对于 CSI 接口，要支持的话需要有对应的 CSI 协议 IP。所以除非是必须要使用 CSI 接口，否则目前的 FPGA 在连接摄像头时，大多还算使用 DVP 接口和 LVDS 接口。本节内容仅概念性简单介绍 CSI 接口，不对 CSI 接口及其协议做系统讲解。

40.4.3控制接口

目前来说，几乎所有的图像传感器都是使用了一种标准的串行摄像头控制总线（Serial Camera Control Bus，简称 SCCB）来作为控制接口。所谓串行摄像头控制总线，虽然看名字貌似是摄像头的一个专属总线，但是实质上，其就是 I2C 总线的一个小变种而已。OmniVision（OV5640 生产厂家）提供了一个名为《OmniVision Serial Camera Control Bus (SCCB) Functional Specification》的文档介绍 SCCB 接口。但是该文档介绍该协议时使用的两相写、三相写、两相读的

思路，反而不易让人理解（据说厂家这么做是为了规避 I2C 总线的版权）。毕竟我们在实际实现的时候，大多都是使用的现有的 I2C 控制器直接连接的。所以这里，我就抛开厂家字节的解释思路，直接拿 SCCB 总线和 I2C 总线的读写操作进行对比，通过对比找到差异，然后使用 I2C 控制器来连接图像传感器时，通过编程手段解决这个差异即可。

SCCB 总线本身分为三线制和两线制两种工作模式，三线制模式是为了支持在一个控制总线上连接多个图像传感器而设计，两线制模式则与典型的 I2C 总线物理结构上完全一致。下图 40-13、图 40-14 分别为三线制和两线制应用的示意图。

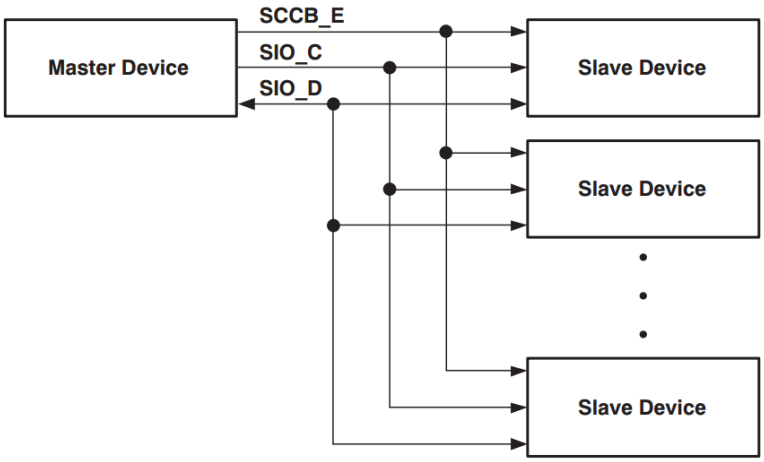


图 40-13 三线制应用模式

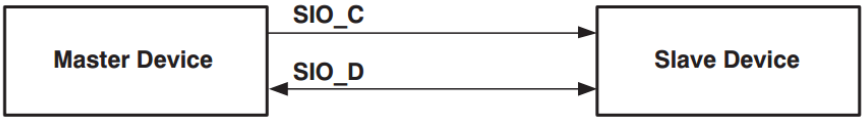


图 40-14 两线制应用模式

目前应用较多的大多为两线制应用模式，所以本节仅介绍两线制模式的应用。

首先来说，作为 SCCB 控制器，其存在的目的就是两个，第一是应用处理器向图像传感器中的指定寄存器写入指定数据，也就是写寄存器操作；第二是应用处理器从图像传感器中读取某些控制和状态数据，用来判断图像处理器的工作状态。所以，SCCB 的功能就是让应用处理器能够读写图像传感器中的寄存器数据。

而 I^2C 协议中，两个最基本的功能也是向器件指定的存储地址写入数据（写操作）和从器件的指定存储地址读出数据。所以首先从功能上来说，SCCB 和

I²C 一致。如果有不了解 I²C 协议的具体内容的，可以先学习下小梅哥的 I²C 相关教程。

其次，从协议时序上来说，I²C 协议有起始位（SCL 为高电平时，SDA 信号出现下降沿）、应答位、停止位（SCL 为高电平时 SDA 信号出现上升沿），而 SCCB 协议也同样遵循这三个定义。

再者，从写寄存器操作来说。SCCB 和 I²C 的写寄存器时序完全一致。直接使用原有的 I²C 总线协议的写寄存器操作就能完成对图像传感器的寄存器写操作。

最后，当从图像传感器中的寄存器读取一个数据时，SCCB 协议仅比 I²C 协议多一个停止位，其他部分完全一样。为了解释这个停止位差别，以下用图示说明。

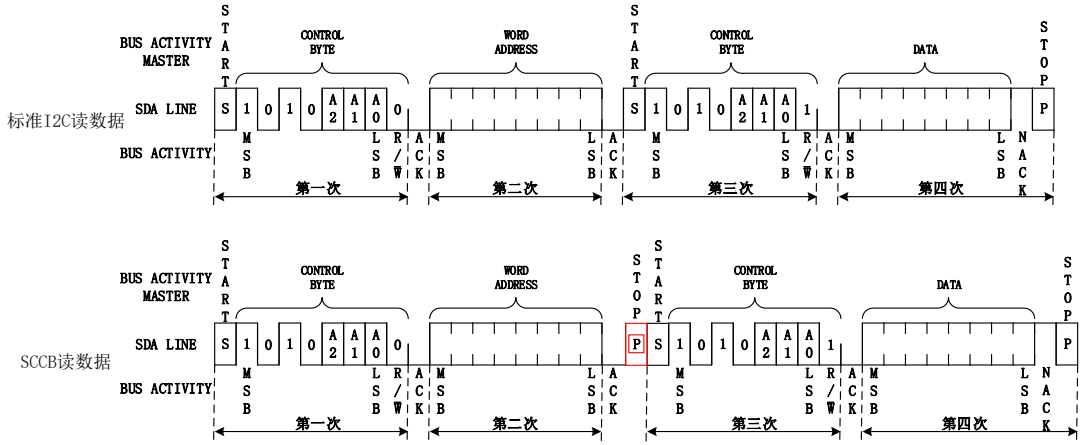


图 40-15 SCCB 协议读数据格式

图中上方为标准 I²C 协议读取一个地址中数据的时序图，下方为 SCCB 协议读取一个地址中数据的时序图，两者的差异以红色线条的形式标记出来。可以看到：对于 I²C 协议，其在执行读的过程中，当第二次传输（word address）传输完成之后，直接产生了新的起始位并开始再一次传输控制字节；对于 SCCB 协议，其在执行读的过程中，当第二次传输（word address）传输完成之后，是先产生了一个停止位，然后再产生新的起始位并开始再一次传输控制字节的。两者仅有此差别。

所以，我们在使用的时候，基本只需要使用标准的 I²C 控制器就可以实现对摄像头的控制了。不过需要补充一点的是，上图中为了控制画幅大小，是以 1 字节存储器地址的器件为例对比的，这样的模式对应 OV7670、OV7725 等摄图像传感器，但是对于 OV5640、OV5640 等图像传感器，都是使用的 2 字节存储器地址的形式。

40.4.4复位控制

复位控制，这个貌似是在很多应用中不太关注的一个问题，但是事实上，在笔者所协助调试的很多系统程序中，因为复位时序不符合要求导致器件不正常工作的案例常有发生，典型的就有某以太网 PHY 芯片和本节内容讨论的 OV5640。没有正确的复位时序，则以太网 PHY 芯片往往无法通过 MDIO 接口配置其工作模式。没有正确的复位时序，OV5640 会无法工作在正确的模式，导致输出的图像效果异常或者直接无法输出数据。

一般的器件都会在其 datasheet 中提供其上电和复位时序要求，OV5640 也不例外，下图为 OV5640 器件手册中提供的 OV5640 的上电复位时序图。

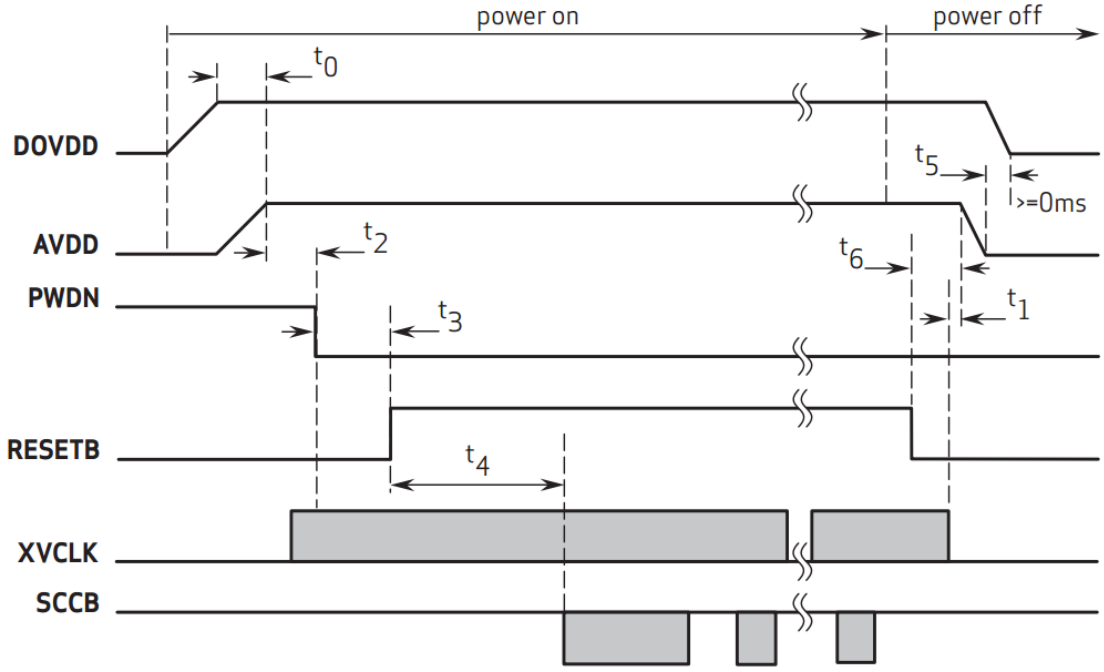


图 40-16 OV5640 的上电复位时序图

根据手册描述：

- PWDN 信号为掉电控制信号，当该引脚为高电平时整个图像传感器的模拟和数字部分都处于掉电不工作状态，为低电平时整个器件才可以开始工作；
- 如果不希望控制 PWDN 信号，则该信号可以直接接到低电平；
- 如果需要控制 PWDN 信号，则需要在上电至少 5ms 之后再拉低 PWDN。
- RESET 信号为复位信号，低电平复位。
- RESET 引脚必须在上电稳定后保持 t3（1ms）时间后才能变为高电平让

整个系统开始工作。

- SCCB 接口工作需要在复位信号拉高 t4（20ms）时间之后才能开始工作。
- XCLK 信号必须在 SCCB 接口正常工作之前 1ms 就就绪，换句话说，如果 XCLK 没有正常提供，OV5640 的 SCCB 接口也将无法正常的读写寄存器。

据此可以总结出对于 OV5640 的从上电到开始读写寄存器的一个基本的控制顺序：

1. PWDN 和 RESET 都为低电平。
2. 等待 PWDN。
3. 等待至少 1ms 的时间，此期间 RESET 一直为低电平，然后拉高 RESET 信号。
4. RESET 拉高之后，再等待至少 20ms 的时间。
5. 开始通过 SCCB 读写 OV5640 的寄存器。
6. XVCLK 提前操作 SCCB 的 1ms 时间一般都能满足，所以一般情况下可以不用关心。

在后面编程实战的章节，我们将结合工程设计，更加深入的对硬件复位和软件复位进行讲解。

40.5OV5640 应用指南

40.5.1OV5640 输出图像尺寸

OV5640 是一个 500W 像素的图像传感器，其拥有 2592*1944 个物理像素。但是相信大家在实际应用的时候也都见到过使用 OV5640 输出的图像大小小于这个最大物理像素尺寸的，比如常见的 VGA 分辨率（640*480）、800*480、800*600、1024*720、1280*720 等。那么这些分辨率的图像是怎么得到的呢。这就要说到 OV5640 的图像输出设置三部曲了。

OV5640 最终输出的图像是经历了开窗、平移、尺寸压缩之后的得到的。什么是开窗，什么是平移，什么是尺寸压缩呢？下图 40-17 为 OV5640 输出图像的示意图。

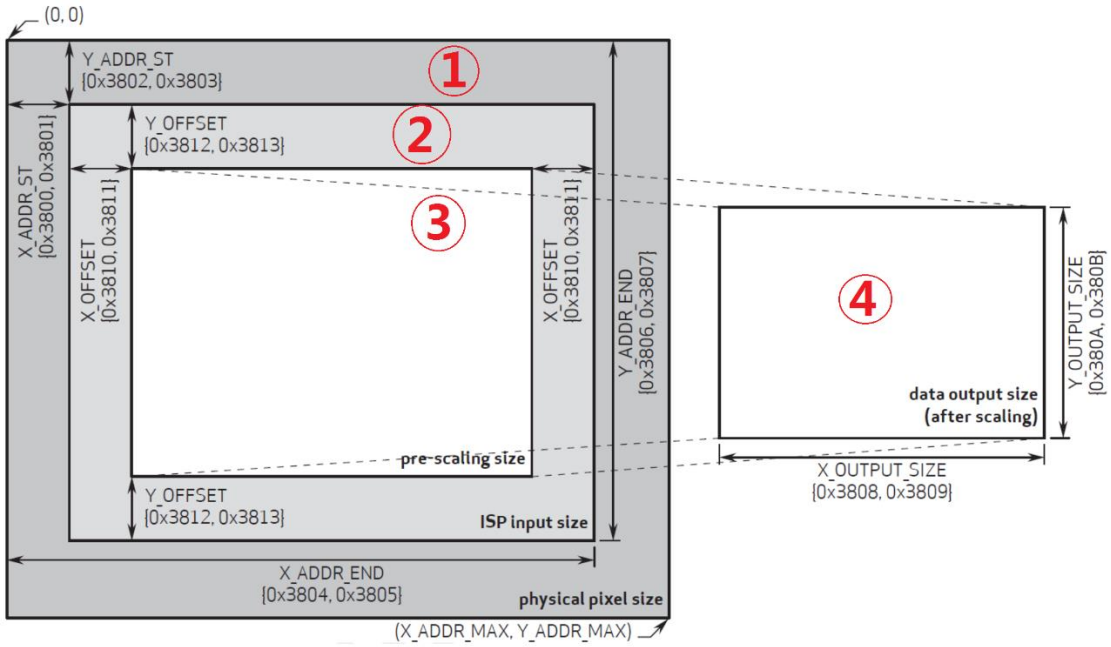


图 40-17 OV5640 输出图像的示意图

OV5640 使用 0x3800~0x3814 寄存器来设置图像窗口，上图显示了寄存器与窗口尺寸定义的关系。物理像素尺寸是整个图像传感器拥有的总的像素矩阵尺寸。ISP 输入尺寸是从像素矩阵中截取的需要读取的像素矩阵部分，一般来讲，ISP 输入尺寸越大，能够达到的最大帧率也就越低。数据输出尺寸是最终从 OV5640 输出的图像尺寸，其大小是在 ISP 输入尺寸的基础上再次经过平移（pre scaling size）和缩放（after scaling）得到的。接下来分别介绍这三个操作。

40.5.2ISP 输入窗口设置（ISP input size）

ISP 输入窗口设置，简称开窗，该设置允许用户设置整个传感器区域（physical pixel size）感兴趣部分，也就是在传感器里面开窗（X_ADDR_ST、Y_ADDR_ST、X_ADDR_END 和 Y_ADDR_END），开窗范围从 0*0~2591*1943 都可以设置，该窗口所设置的范围，将输入 ISP 进行处理。在左侧最大的一圈（标记为①）是整个图像传感器的物理像素矩阵，大小为 2592*1944；而次外围（标记为②）则是通过开窗的方式，从整个物理像素矩阵中截取出的一块图像像素，整个截取的图像范围是可以通过（0x3800~0x3807 寄存器设置的）。下表为 0x3800~0x3807 寄存器的功能介绍。

表 40-4 ISP 输入窗口设置寄存器功能介绍

寄存器地址	名称	默认值	功能描述
0x3800	TIMING HS	0x00	bit[3:0]: X 方向开始位置地址高 4 位
0x3801	TIMING HS	0x00	bit[7:0]: X 方向开始位置地址低 8 位

0x3802	TIMING VS	0x00	bit[2:0]: Y 方向开始位置地址高 3 位
0x3803	TIMING VS	0x00	bit[7:0]: Y 方向开始位置地址低 8 位
0x3804	TIMING HW	0x0A	bit[3:0]: X 方向结束位置地址高 4 位
0x3805	TIMING HW	0x3F	bit[7:0]: X 方向结束位置地址低 8 位
0x3806	TIMING VH	0x07	bit[2:0]: Y 方向结束位置地址高 3 位
0x3807	TIMING HH	0x9F	bit[7:0]: Y 方向结束位置地址低 8 位

40.5.3预缩放窗口设置（pre-scaling size）

该设置允许用户在 ISP 输入窗口的基础上，再次设置将要用于缩放的窗口大小。该设置通过在 ISP 输入窗口内将矩阵的起始位置较 ISP 输入窗口的起始位置进行 X（X_OFFSET）和 Y（Y_OFFSET）方向的平移。这两个偏移量是通过 0X3810~0X3813 等 4 个寄存器进行设置。至于平移的目的，一种可能就是在镜头机械位置等已经固定的情况下，针对特定应用调整采集到的图像输出的中心位置。注意，该设置仅仅调整的是有效输出区间的左上角的起始位置，没有定义区间的结束位置。

下表 40-5 为 0X3810~0X3813 寄存器的功能说明。

表 40-5 预缩放窗口设置寄存器功能介绍

寄存器地址	名称	默认值	功能描述
0x3810	TIMING HOFFSET	0x00	bit[3:0]: 预缩放窗口 X 方向起始位置相较于 ISP 输入窗口 X 方向的偏移量的高 4 位
0x3811	TIMING HOFFSET	0x10	bit[7:0]: 预缩放窗口 X 方向起始位置相较于 ISP 输入窗口 X 方向的偏移量的低 8 位
0x3812	TIMING VOFFSET	0x00	bit[2:0]: 预缩放窗口 Y 方向起始位置相较于 ISP 输入窗口 Y 方向的偏移量的高 3 位
0x3813	TIMING VOFFSET	0x04	bit[7:0]: 预缩放窗口 Y 方向起始位置相较于 ISP 输入窗口 Y 方向的偏移量的低 8 位

注意，这里的偏移量的大小值是基于 ISP 输入窗口的起始地址的增量，例如，假设 ISP 输入窗口的 X 方向起始位置为 0x0F1、而 X 方向的偏移量值为 0x005，那么最终输出图像的 X 方向的起始地址就是 0x0F6。Y 方向也是相同的计算方法。

40.5.4输出大小窗口设置（data output size）

该窗口是以预缩放窗口为原始大小，经过内部 DSP 进行缩放处理后，输出给外部的图像窗口大小。它控制最终的图像输出尺寸（X_OUTPUT_SIZE/Y_OUTPUT_SIZE）。例如当希望将 OV5640 的图像通过 800*480 的显示屏显示时，就应该设置图像的输出大小窗口尺寸为 800*480。输

出窗口尺寸大小是通过 0X3808~0X380B 等 4 个寄存器进行设置。

下表 40-6 为 0X3808~0X380B 寄存器的功能说明。

表 40-6 输出大小窗口设置寄存器功能说明

寄存器地址	名称	默认值	功能描述
0x3808	TIMING DVPHO	0x00	bit[3:0]: 输出图像 X 方向尺寸大小的高 4 位
0x3809	TIMING DVPHO	0x10	bit[7:0]: 输出图像 X 方向尺寸大小的低 8 位
0x380A	TIMING DVPVO	0x00	bit[2:0]: 输出图像 Y 方向尺寸大小的高 3 位
0x380B	TIMING DVPVO	0x04	bit[7:0]: 输出图像 Y 方向尺寸大小的低 8 位

注意：当输出大小窗口与预缩放窗口比例不一致时，图像将进行缩放处理（会变形），仅当两者比例一致时，输出比例才是 1:1（正常）。

40.5.5OV5640 输出图像时序

在了解了 OV5640 的输出图像尺寸设置原理之后，接下来我们就可以讨论其输出图像的时序了。在图像显示领域，会将一些特定的分辨率命一个名词，下表 40-7 为常见的分辨率定义和其对应的实际像素矩阵大小：

表 40-7 常见分辨率定义表

定义	分辨率	定义	分辨率
QSXGA	2592*1944	XGA	1024*768
QXGA	2048*1536	SVGA	800*600
UXGA	1600*1200	VGA	640*480
SXGA	1280*1024	QVGA	320*240
WXGA+	1440*900	QQVGA	160*120
WXGA	1280*800		

40.5.5.1行时序

下图 40-18 为 OV5640 在 DVP 接口模式下输出一行图像数据的时序图。

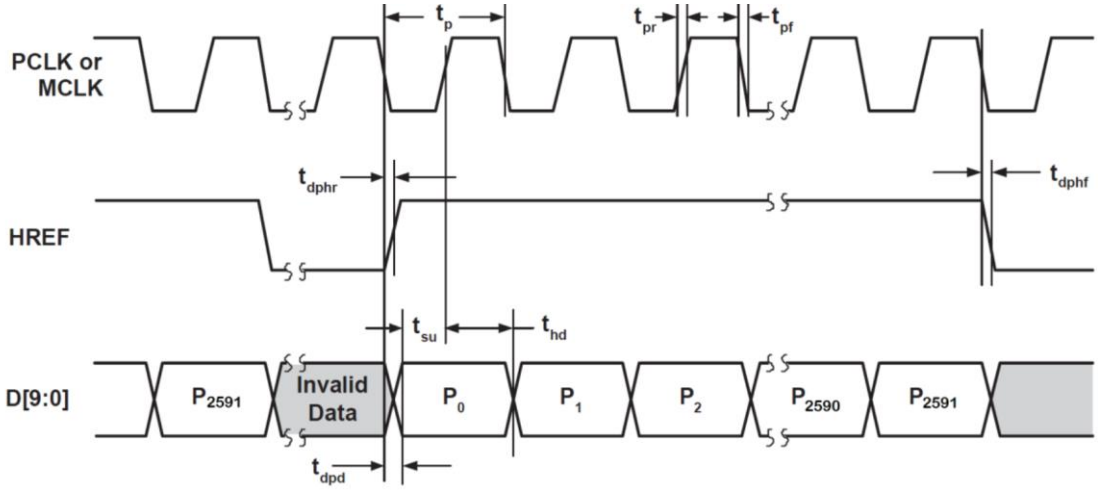


图 40-18 OV5640 在 DVP 接口模式下一行图像行时序

再来回顾一下图中的一些信号和功能意义。

- PCLK，即像素时钟，一个 PCLK 时钟，输出一个像素（RAW 模式）或半个像素（RGB564、YUV422）或 1/3 个像素（RGB888、YUV444 模式）数据。
- VSYNC，即帧同步信号。当设置 VSYNC 信号极性为高有效时，VSYNC 信号的高电平期间输出一帧图像。
- HREF/HSYNC，即行同步信号，当设置 HSYNC 信号极性为高有效时，高电平期间输出一行图像。

从上图可以看出，图像数据在 HREF 为高的时候输出，当 HREF 变高后，每一个 PCLK 时钟，输出一个 8 位/10 位数据。我们采用 8 位接口，所以每个 PCLK 输出 1 个字节，且在 RGB565/YUV422 输出格式下，每个 tp=2 个 Tpclk，如果是 Raw 格式，则一个 tp=1 个 Tpclk。比如我们采用 QSXGA 时序， RGB565 格式输出，每 2 个字节组成一个像素的颜色（低字节在前，高字节在后），这样每行输出总共有 2592*2 个 PCLK 周期，输出 2592*2 个字节。

40.5.5.2场时序

下图 40-19 为 OV5640 在 DVP 接口模式下输出一帧图像（QSXGA）数据的时序图。

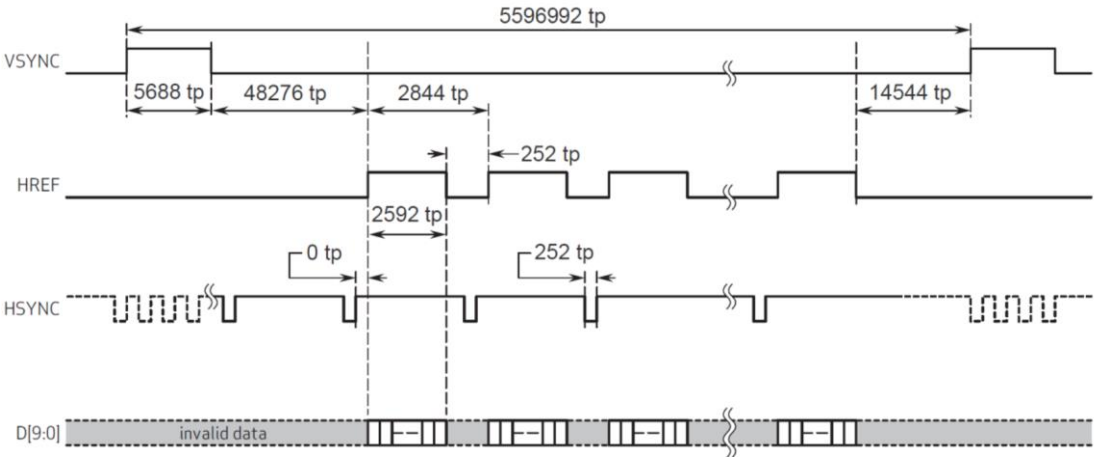


图 40-19 OV5640 在 DVP 接口模式下一场图像场时序

上图清楚的表示了 OV5640 在 QSXGA 模式下的数据输出。我们按照整个时序去读取 OV5640 的数据，就可以得到图像数据。

总结几个与我们使用时密切相关的知识点。

1. 所有信号（HREF、VSYNC、DATA）都是在 PCLK 的下降沿变化，应用处理器应该在 PCLK 的上升沿取用数据。
2. VSYNC 作为帧同步信号，默认是高电平有效，也就是说，每当新一帧图像开始输出之前，VSYNC 都会呈现一段时间的高电平。而在正常输出图像数据期间，VSYNC 信号保持低电平。在 VSYNC 变为低电平后的第一个 HREF 高电平期间输出的数据就是新一帧图像的第一行数据。
3. HREF 作为行数据有效信号，在其为高电平时，DATA 数据线上每个时钟周期输出一个新的数据。
4. 关于 DATA 数据线上每个数据所对应的意义，将在下一节介绍。

40.5.6OV5640 输出图像格式

对于 OV5640 的图像输出数据格式，在不同的应用领域有所不同。在嵌入式系统如单片机、ARM 处理器系统中，一般常用 2 种输出方式：RGB565 和 JPEG。在图像处理系统中，一般常用 RGB565 模式和 YUV422 格式。而在图像采集系统中，为了充分保留图像传感器采集到的原始图像信息，会采用 RAW 格式输出。

当输出 RGB565、YUV422、RAW 格式数据的时候，时序完全就是上面两幅图介绍的关系。而当输出数据是 JPEG 数据的时候，同样也是这种方式输出（所以数据读取方法一模一样），不过 PCLK 数目大大减少了，输出的数据是压缩后的 JPEG 数据，输出的 JPEG 数据以 0XFF，0XD8 开头，以 0XF，0XD9 结尾，且在 0XFF，0XD8 之前，或者 0XFF，0XD9 之后，会有不定数量的其他数据存在（一般是 0），这些数据我们直接忽略即可，将得到的 0XFF，0XD8~0XFF，0XD9 之间的数据，保存为.jpg/.jpeg 文件，就可以直接在电脑上打开看到图像了。

在基于 FPGA 的系统中，使用 OV5640 做图像采集实时显示，图像处理的应用较多。在图像处理算法中，针对不同的算法和应用目的，常使用 RGB565 模式和 YUV422 格式，JPEG 格式一般很少用到。在对颜色敏感的应用中，常用 RGB888 模式，如仓库物品分拣系统中通过不同的颜色值区分不同的产品。而其他更多的图像处理或图像识别系统中，一般是使用灰度图像进行处理，此时常用的图像模式为 YUV422 模式。RGB565 模式使用 16 位数据来表示一个像素点代表的红绿蓝分量，与我们最熟悉的颜色表示方式一致。而 YUV 模式则是采用色度和亮度分离的模式来表示一个像素点，当仅使用图像中的亮度值来表示

一个像素的颜色时，就是传统的灰度图像，老式的黑白电视机就是仅能显示亮度值而无法显示色度值。

关于输出图像格式的具体寄存器设置，请参考后续章节中“典型工作模式配置”下的“调整图像输出模式”部分。

40.6OV5640 典型工作模式配置

接下来介绍 OV5640 基于 FPGA 的图像采集系统中各种常见的典型参数配置。

40.6.1基本初始化配置

基本初始化配置表以 800*480@30FPS 的 RGB565 输出模式为例。

40.6.2修改信号极性

为得到正确的图像，OV5640 传感器的视频信号极性与基带芯片或 ISP 必须设置成一致。例如，默认状态下，OV5640 的 VSYNC 是低电平有效，即 VSYNC 信号在输出图像时为低电平。而我们常见的图像捕获系统中，往往以 VSYNC 信号为高电平代表图像数据有效，此时就需要设置 VSYNC 信号的极性为高电平有效。OV5640 可以通过设置地址为 0x4740 号寄存器的值来设置信号的极性，具体设置方法如下表 40-8 所示。

表 40-8 0x4740 寄存器极性设置

信号名称	寄存器中数据位	功能描述
VSYNC	0x4740.bit0	1 – Vsync 为高时输出数据有效 0 – Vsync 为低时输出数据有效
HREF	0x4740.bit1	0 – Href 为高时输出数据有效 1 – Href 为低时输出数据有效
PCLK	0x4740.bit5	1 – 数据在下降沿输出 0 – 数据在上升沿输出

40.6.3修改帧率

OV5640 的图像输出帧率可以通过修改地址为 0x3035、0x3036、0x3037 的寄存器的值来修改，该寄存器实际上是设置了 OV5640 片上的各种分频和倍频系数，例如在典型配置模式下，当输入时钟 XCLK 的信号频率为 24MHz 时，0x3035 寄存器的值为 0x21 可设置输出帧率为 30fps，设置 0x41 可输出帧率为 15fps，设置为 0x81 可输出帧率为 7.5fps。

40.6.4图像镜像翻转

因为 OV5640 是一款 BSI 图像传感器, 成像光线是从芯片背面射入的，所以原始生成的图像看起来是左右相反的，故此需要对图像做镜像处理使其显示正常。设置镜像和翻转功能是通过设置寄存器 0x3820 和 0x3821 的值实现的。上电时，0x3820 的值默认为 0x40，0x3821 的值默认为 0x00。0x3820 寄存器的 bit2 和 bit1 分别设置 ISP 和传感器的翻转，0x3821 寄存器的 bit2 和 bit1 分别设置 ISP 和传感器的镜像。

一幅正放的图片，人眼看到的图像如下图 40-20 所示，可以看到，文字都是正常的。

图 40-20 正放图片的人眼视觉效果

当使用 OV5640 采集之后，如果不设置翻转和镜像功能，即寄存器的值为上电默认值，0x3820 = 0x40，0x3821 = 0x00。其输出图像默认如下图 40-21 所示：

图 40-21 摄像头寄存器采用上电默认值

以图像下方的一排文字为参考，可以看到，图像相较于原始图像，在水平方向发生了镜像。

如果设置图像不镜像，仅翻转，即寄存器的值为 0x3820=0x40|0x06;0x3821=0x00&0xf9。则是下图所示的样式。

图 40-22 摄像头寄存器设置为上下翻转模式

以原始图像下方的一排文字为参考，可以看到，图像相较于原始图像，在垂直方向发生了翻转，下方的文件，翻转到了上方。

如果设置图像既镜像，又翻转，即寄存器的值为 0x3820 = 0x40|0x06;0x3821 = 0x00|0x06。则是下图所示的样式。

图 40-23 摄像头寄存器设置为翻转加镜像模式

以原始图像下方的一排文字为参考，可以看到，图像左右镜像，且上下翻转了。

因此，当希望图像正常显示时，设置镜像和翻转都关闭即可。设置 0x3820 = 0x40 & 0xf9;0x3821 = 0x00 & 0xf9。

40.6.5调整图像尺寸

前面提到，设置输出图像尺寸可以通过设置 ISP 在传感器上的开窗，预缩放偏移和输出图像大小窗口。最常用的是设置输出图像大小窗口设置，地址为 0x3808 和 0x3809 的寄存器设置输出图像的高度，地址为 0x380a 和 0x380b 的寄存器设置输出图像的宽度。例如，需要设置输出图像大小为 800*480 分辨率，则设置输出图像高度寄存器的值为 0x0320（800d），设置输出图像宽度寄存器的值为 0x01e0（480d）。即设置：

0x3808 = 0x03; // DVPHO800　　0x3809 = 0x20; // DVPHO

0x380a = 0x01; // DVPVO480　　0x380b = 0xe0; // DVPVO

40.6.6调整图像输出模式

OV5640 输出图像制式支持多种制式，如 RGB、YUV、RAW。而 RGB 制式

又包括 RGB888、RGB565、RGB555、RGB444 模式等，YUV 制式包括 YUV444、YUV422、YUV420 模式等，而每个模式下，又根据输出像素的各个字节代表的不同意义，又分为多种模式，例如对于 RGB565 模式，连续的两个字节代表一个像素的颜色值，在代表一个像素点的 2 个字节数据中，哪几位代表红色分量，哪几位代表绿色分量，哪几位代表蓝色分量，也是通过寄存器可以设置的。具体模式设置是通过 0x4300 这个寄存器设置的。这里仅介绍两种常用的模式设置，RGB565 和 YUV422。其他模式，用户可以通过查看 OV5640_CSP3_DS_2.01_Ruisipusheng.pdf 中相关描述。

0x4300 寄存器共 8 位，其中 bit[7:4]设置图像输出模式，bit[3:0]设置每个模式下输出像素内容的顺序。下表为 bit[7:4]的值和对应的输出模式的关系。

表 40-9 调整图像输出格式寄存器

值	输出模式	值	输出模式	值	输出模式
0x0	RAW	0x4	YUV420	0x8	RGB555 format2
0x1	Y8	0x5	YUV420(MIPI)	0x9	RGB444 format1
0x2	YUV444/RGB888	0x6	RGB565	0xa	RGB444 format2
0x3	YUV422	0x7	RGB555 format1	0xf	Bypass

说明：关于很多读者所关心的 JPEG 模式，并不是在这个寄存器中设置的，JPEG 输出是另外有单独的 JPEG 压缩引擎，对 YUV422 或 YUV420 格式的图像编码得到的。如果需要输出 JPEG 格式，则本寄存器应该设置高 4 位的值为 0x3 或 0x4，即选择 YUV422 或 YUV420 格式，然后再设置 JPEG 相关的寄存器。对于 JPEG 模式本手册不做讨论。

当设置了图像输出模式后，可以通过设置该寄存器的 bit[3:0]来设置输出图像的顺序。例如，对于 RGB565 模式(bit[7:4]=0x6)，设置 bit[3:0]为不同的值则可以实现不同的输出序列，如下表所示。

表 40-10 RGB565 模式下输出图像顺寻配置表

值	高字节内容定义	低字节内容定义
0x0	{b[4:0],g[5:3]}	{g[2:0],r[4:0]}
0x1	{r[4:0],g[5:3]}	{g[2:0],b[4:0]}
0x2	{g[4:0],r[5:3]}	{r[2:0],b[4:0]}
0x3	{b[4:0],r[5:3]}	{r[2:0],g[4:0]}
0x4	{g[4:0],b[5:3]}	{b[2:0],r[4:0]}
0x5	{r[4:0],b[5:3]}	{b[2:0],g[4:0]}
0x6~0xe	Not allowed	Not allowed
0xf	{g[2:0],b[4:0]}	{r[4:0],g[5:3]}

对于我们常见的应用来说，一般是使用 RGB 或 BGR 序列，即设置寄存器的 bit[3:0]的值为 0x0 或 0x1。

下表 40-11 为 YUV422 模式（bit[7:4]=0x3）时 bit[3:0]的值与对应的输出序

列的关系：

表 40-11 不同 bit 值与输出序列关系

值	输出序列	值	输出序列
0x0	YUYV	0x3	VYUY
0x1	YVYU	0x4~0xe	Not Allowed
0x2	UYVY	0xf	UYVY

在提供的基于高云开发板的 OV5640 实例中，我们大都设置的是 RGB565 模式。

40.6.7彩条测试模式

通过设置地址为 0x503d 寄存器中相应位的值，可以设置 OV5640 输出的图像内容是传感器采集到的图像还是测试信号发生器模块产生的测试信号。

0x503d 寄存器的最高位 bit7 为测试信号发生器的开关控制位，设置为 1 即可开启测试信号发生器。bit[1:0]为测试信号类型选择寄存器不同的值可以选择不同的测试模式，如彩条（Color Bar）、棋盘格（Color square）。

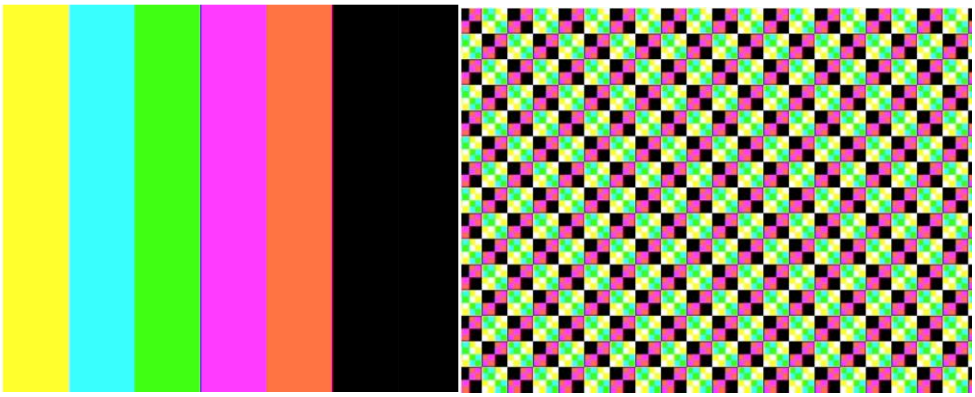


图 40-24 彩条测试地址寄存器配置后输出效果

例如要设置彩条测试模式，设置 0x503d 寄存器的值为 0x80 即可。

到这里，摄像头的基本理论知识和一些基础的测试方法就讲解完成了，希望读者根据讲解的内容，认真领会摄像头的基本控制思想。

40.7总结

本章讲解了 OV5640 摄像头的基本理论知识和典型工作模式配置。摄像头通过配置初始化寄存器，可以明确其工作模式，数据数据格式，复位、翻转、画幅大小、帧率等一系列关键信息。

41 OV5640 基于 FPGA 的编程实战

工程源码	|----02_设计实例 |----ch41_ov5640_ddr3_tft
相关视频课程	
说明	

章节导读

前面介绍了这么多内容，其主要目的是为了让大家了解 OV5640 的详细工作原理，方便大家在日后应用时能够根据原理举一反三，拥有开发和调试基于 OV5640 的数字系统的能力。而使用 OV5640，无外乎两个方面的工作——正确设置其工作模式和正确接收其输出的图像数据流。

由于 CSI 接口需要专门的解码 IP 或者专用集成电路，所以这里不做讨论。在我们开发的 OV5640_V5 摄像头模组中，使用的是 DVP 接口。同时，为了让 OV5640_V5 模组能够正常的接受 FPGA 对其进行的各种工作模式设置，模组上也提供了 2 线制的 SCCB 接口（等价于 I2C）。所以对于 OV5640_V5 模块来说，只需要使用 I2C 控制器完成其各种模式寄存器设置，然后设计一个能够正确的接收 OV5640 的 DVP 接口数据流输出的数据的逻辑功能电路即可。

41.1基于 OV5640 的图像采集显示系统

在前面的章节，我们已经介绍过来 OV5640 的 SCCB 接口和 DVP 接口时序。SCCB 接口实现很简单，由于兼容 I2C 总线的缘故，只需要使用成熟的 I2C 控制器配合简单的控制逻辑即可实现。而 DVP 接口的时序是很简单的，只需要 FPGA 被动的按照 DVP 接口的数据流输出时序把数据流渠道 FPGA 内部即可。

DVP 接口很简单，要想成功接收 DVP 接口的数据流非常容易，而关键点在于，接收到的数据流该去干什么。个人认为，脱离了实际应用场景来谈 DVP 接口数据流接收没有太大的意义。得出此结论是基于笔者使用 OV5640 开发的多个应用系统的经验。例如，如果需要把 OV5640 采集到的图像数据通过千兆以太网发送出去，由于千兆以太网本身就是 8 位的数据位宽，而 OV5640 的 DVP 接口也是 8 位位宽，所以几乎可以直接将 DVP 接口的数据线直连到千兆以太网发送逻辑端口上。而如果要将 OV5640 的图像数据通过 USB3.0 传输到 PC 机，由于 USB3.0 芯片与 FPGA 采用的是 32 位位宽的总线相连，那么将 DVP 接口的数据最好是能够接收到后每 4 个数据组合成一个 32 位的数据在送给 USB3.0 的

接口逻辑。而最常见的将图像存储到 SDRAM 存储器中的应用，因为 SDRAM 提供的是 16 位的数据端口，所以将 DVP 接口输出的数据按照每 2 个数据组合成 16 位数据后再写入 SDRAM 比较合适。另外，从数据格式来说，输出的数据格式为 RGB888、RGB565、YUV422、YUV420、JPEG 等都有可能，不同的数据格式，在接收和使用时也有相应的要求。

为了让大家能够真正理解并掌握 OV5640 的应用开发规律。这里采用基于实际应用的思路，先提出一个应用需求，然后根据应用需求来设计相应的接口逻辑。

41.2图像采集显示系统定义

从实验现象的角度来说，将摄像头采集的图像直接显示在显示屏上是最简单直观的。所以本节将在实验“基于 DDR3 的串口传图帧缓存系统设计”实现一个图像采集显示应用系统，来设计 OV5640 应用系统中所需的基本应用逻辑电路。

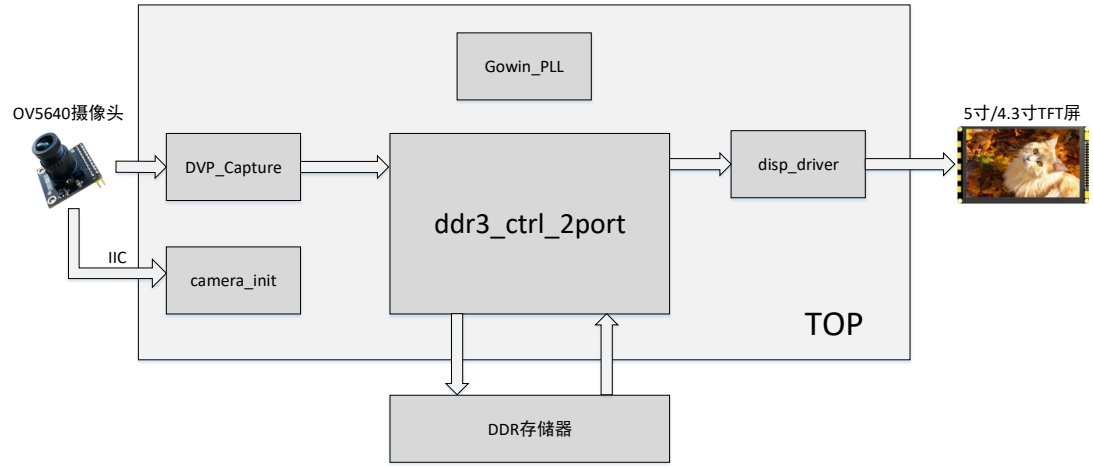


图 41-1 图像采集显示系统基本框图

上图为 OV5640 图像采集显示系统的框图，在该图中，所需的基本硬件只有 OV5640 摄像头+转接板（或者 ACM5640_Dual 双目摄像头），TFT 显示屏，高云开发板。对于这样的系统，分析理解是应按照数据流的流向进行。

按照数据流方向进行分析时，如果是为了理解现有系统的功能，可采用正推的方法进行，既按照数据流的方向顺序分析。而如果是为了根据系统模型来确定系统中的各个接口规范，则应该按照数据流的方向逆向分析，既从数据流的最终使用者一步一步倒推数据源的需求。此所谓由需求决定设计。

整个系统最终的目的是要在 TFT 显示屏上显示图像数据，由于 TFT 显示屏

采用的是 RGB565 接口，为了让图像正常的显示在显示屏的指定位置，需要有相应的时序发生逻辑，这就是上图中的 Disp_Driver。该模块会按照 TFT 显示屏的接口时序产生对应的控制时序（HS、VS、DE、RGB data），实质上就是我们已经学习过的 TFT/VGA 控制器。

由于 TFT 显示屏显示时候，是按照 60 帧率的速率进行刷新，也就是每秒刷新 60 张图像，为了保证图像的正常显示，一般需要有一个图像缓冲区，TFT 控制器实时从该图像缓冲区中读取数据并送往 TFT 屏上显示。所以在上述系统中，设计了一个 DDR3 用来存储需要显示的图像数据。

从设计的框图可以看出，整个系统与基于 DDR3 的串口传图帧缓存系统的差异在于串口传图是电脑通过串口将数据传输给 FPGA，而本实验是通过 OV5640 摄像头采集数据传给 FPGA 处理。本节重点是如何驱动 OV5640，让 FPGA 正确采集到图像数据。

DVP Capture 模块负责将 DVP 接口输出的数据按照每两个一组合，得到符合 RGB565 图像格式的 16 位数据，之后数据处理过程与串口传图是一样，所以需要使用 SCCB 接口对 OV5640 进行设置，让其图像输出格式为 RGB565。

OV5640 要想能够正常的输出图像数据，必须经 SCCB 接口对其寄存器进行配置，所以整个系统首要的工作是使用控制逻辑经由 SCCB 接口对其进行初始化，在上图中，对 OV5640 进行初始化的功能模块为 Camera_Init。该模块会在系统开始工作时，对 OV5640 中各个寄存器写入指定值以实现初始化操作。接下来，就首先针对 Camera_Init 模块进行介绍。

41.3摄像头初始化模块设计思路

所谓的 OV5640 摄像头初始化模块，其工作就是完成对 OV5640 中众多模式设置寄存器的写入操作。本质上就是使用 I2C 控制器将一些预先定义好的数据值写入到 OV5640 对应的寄存器中。所以我们完全可以使用之前设计好的 I2C 控制器加上一个存储好所有 OV5640 所需设置的寄存器参数的查找表，配合一定的控制逻辑实现，该设计实际比较简单，本节介绍时就不再采用思维引导的方式进行，而是介绍一个笔者设计好的功能模块的设计框架和应用方法。下图为我们设计的一个比较通用的 OV5640 初始化逻辑电路。

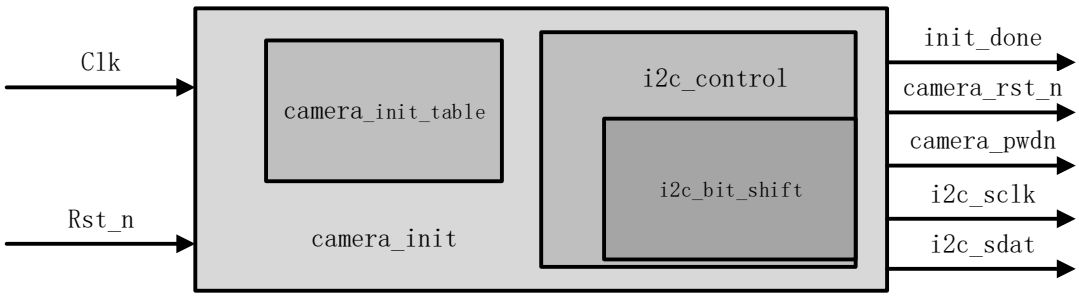


图 41-2 摄像头初始化模块系统框图

图中例化了设计好的通用 I2C 控制器（i2c_control）和一个 ROM 查找表（camera_init_table），在 camera_init 顶层逻辑中，通过简单的状态机循环读取 camera_init_table 中的数据值并通过 i2c_control 即可写入到 OV5640 的各个寄存器中。该代码已经提供在了我们的各个开发板配套资料包中，大家也可以通过我们的网站：www.corecourse.cn，搜索“camera_init”关键词找到。

41.4摄像头初始化模块典型用法

在大多情况下，直接基于成熟的摄像头的初始化控制方案而稍加修改，就能将其应用到实战开发中。在应用上，仅需在整个系统顶层例化摄像头初始化模块的代码，即可正常使用该模块。比如，这里我们需要输出 160x128 的画幅，则顶层例化方法如下：

```
localparam RGB = 0;
localparam JPEG = 1;

parameter IMAGE_WIDTH  = 160; //图片宽度
parameter IMAGE_HEIGHT = 128; //图片高度(≤720)
parameter IMAGE_FLIP_EN   = 0;   //0：不翻转，1：上下翻转
parameter IMAGE_MIRROR_EN = 0;   //0：不镜像，1：左右镜像

//摄像头初始化配置
wire Init_Done;

camera_init
#(
  .CAMERA_TYPE ( "ov5640"    ),//"ov5640" or "ov7725"
  .IMAGE_TYPE  ( 0           ),// 0: RGB; 1: JPEG
  .IMAGE_WIDTH ( IMAGE_WIDTH  ),// 图片宽度
  .IMAGE_HEIGHT( IMAGE_HEIGHT ),// 图片高度
  .IMAGE_FLIP_EN ( 0           ),// 0: 不翻转，1: 上下翻转
  .IMAGE_MIRROR_EN  ( 0           ) // 0: 不镜像，1: 左右镜像
)camera_init
```

```
(
  .Clk          (Clk              ),
  .Rst_n        (locked           ),
  .Init_Done    (Init_Done        ),
  .camera_rst_n(                  ),
  .camera_pwdn (                  ),
  .i2c_sclk     (Camera_sclk      ),
  .i2c_sdat     (Camera_sdat      )
);
```

在代码中，加入了几个参数定义，每个参数对应了 OV5640 使用时的一个常用模式。现就这几个模式参数分别介绍如下。

CAMERA_TYPE	定义选定的摄像头是 ov5640 还是 ov7725。
RGB	定义 RGB 输出模式，是用来指示 IMAGE_TYPE 图片类型的，（0 表示为 RGB 类型，1 表示为 JPEG 类型），为了用户更直观，所以用了两个 localparam 来定义了，这样直接在 IMAGE_TYPE 的参数中直接填写 RGB 或者 JPEG，就能将 0 或者 1 传到被例化的模块里面。
JPEG	定义 JPEG 输出模式，是用来指示 IMAGE_TYPE 图片类型的，（0 表示为 RGB 类型，1 表示为 JPEG 类型），为了用户更直观，所以用了两个 localparam 来定义了，这样直接在 IMAGE_TYPE 的参数中直接填写 RGB 或者 JPEG，就能将 0 或者 1 传到被例化的模块里面。
IMAGE_WIDTH	输出图像宽度设置，当前寄存器初始化表中的设置，宽度最大可支持到 1280 像素
IMAGE_HEIGHT	输出图像高度设置，当前寄存器初始化表中的设置，宽度最大可支持到 720 像素
IMAGE_FLIP	图像翻转设置，0 则不翻转，1 则上下翻转
IMAGE_MIRROR	图像镜像设置，0 则不镜像，1 则左右镜像

使用时，关于控制图像的类型、分辨率、翻转、镜像，可以通过修改 localparam 和 parameter 的值，也可以直接在例化时候修改#后面括号里面的对应内容。我们上一个小节分析的摄像头配置信息，也是通过这里定义的参数，来传递给底层模块，从而实现对摄像头类型和配置信息的识别。

i2c_control 模块通过控制 i2c_bit_shift 模块顺序从 camera_init_table 里面（这里面有两个表 ov5640_init_table_jpeg、ov5640_init_table_rgb），读取预先存储好的摄像头的寄存器值，通过 i2c 总线（也就是顶层里面的 camera_sclk、camera_sdat 这两根线）来配置 ov5640 摄像头寄存器。

在接下来的内容中，我们将讲解几个典型的摄像头初始化 寄存器配置方法。如果读者希望了解更多初始化寄存器的相关内容，可以通过两个初始化寄存器的查找表文件 ov5640_init_table_jpeg 和 ov5640_init_table_rgb 中的寄存器的值更多信息，还可以参考 OV5640 的《OV5640_自动对焦照相模组应用指南(DVP_接口)__R2.13C.pdf》文档。

41.5摄像头初始化重点代码分析

41.5.1摄像头初始化之寄存器控制

OV5640 支持多种输出格式，包括 JPEG、RGB、YUV、灰度、RAW 等。这些格式中，RGB、YUV、灰度、RAW 格式的寄存器初始化内容大体相同，仅有个别寄存器设置差异，可以使用同一个寄存器初始化列表，然后修改个别寄存器内容即可，而 JPEG 模式中，有很多寄存器的设置内容，相较于 RGB 格式差别比较大，所以我们设计了基于 generate 条件判断语法的选择性生成逻辑针对选定的图像模式是 RGB 还是 JPEG，选择 RGB 或 JPEG 模式的寄存器初始化表，作为 OV5640 的寄存器初始化信息。

同时，由于 OV5640 和 OV7725 这一类 CMOS 摄像头的 DVP 接口都是一模一样的，差别仅在于 SCCB 接口以及寄存器初始化表，所以设计中同时还使用 generate 条件判断语法，对 OV7725 以及 OV5640 的初始化逻辑做了选择性生成。这样，我们在使用的时候，只需要在顶层定义好使用的摄像头型号以及图像模式，就能指导 FPGA 编译软件编译时只对于模式匹配的逻辑模块进行编译，该部分代码如下所示。

代码所在模块 ： camera_init

```
generate
 ////////////////////////////ov5640////////////////////////////////
 if(CAMERA_TYPE == "ov5640")
 begin
   assign device_id = 8'h78;
   assign addr_mode = 1'b1;
   assign addr = lut[23:8];
   assign wrdata = lut[7:0];

   if(IMAGE_TYPE == RGB) //------RGB------
   begin
     assign lut_size = 252;

     ov5640_init_table_rgb #(
       .IMAGE_WIDTH     (IMAGE_WIDTH     ),
       .IMAGE_HEIGHT    (IMAGE_HEIGHT    ),
       .IMAGE_FLIP_EN   (IMAGE_FLIP_EN   ),
       .IMAGE_MIRROR_EN (IMAGE_MIRROR_EN )
     )ov5640_init_table_rgb_inst
     (
       .addr (cnt ),
       .clk  (Clk ),
```

```
        .q    (lut )
      );
    end
    else  //------JPEG------
    begin
      assign lut_size = 250;

      ov5640_init_table_jpeg #(
        .IMAGE_WIDTH     (IMAGE_WIDTH     ),
        .IMAGE_HEIGHT    (IMAGE_HEIGHT    ),
        .IMAGE_FLIP_EN   (IMAGE_FLIP_EN   ),
        .IMAGE_MIRROR_EN (IMAGE_MIRROR_EN )
      )ov5640_init_table_jpeg_inst
      (
        .addr (cnt ),
        .clk  (Clk ),
        .q    (lut )
      );
    end
  end
  ////////////////////////////ov7725////////////////////////////////
  else if(CAMERA_TYPE == "ov7725")
  begin
    assign device_id = 8'h42;
    assign addr_mode = 1'b0;
    assign addr = lut[15:8];
    assign wrdata = lut[7:0];

    if(IMAGE_TYPE == RGB) //------RGB------
    begin
      assign lut_size  = 68;

      ov7725_init_table_rgb #(
        .IMAGE_WIDTH     (IMAGE_WIDTH     ),
        .IMAGE_HEIGHT    (IMAGE_HEIGHT    ),
        .IMAGE_FLIP_EN   (IMAGE_FLIP_EN   ),
        .IMAGE_MIRROR_EN (IMAGE_MIRROR_EN )
      )ov7725_init_table_rgb_inst
      (
        .addr (cnt ),
        .clk  (Clk ),
        .q    (lut )
      );
    end
  end
endgenerate
```

综合设计需求来看，IIC 控制器必须能够识别摄像头的类型为 OV5640，

OV7725，也能够识别 OV5640 摄像头的数据格式为 RGB 或 JPEG。因此，如果要保证 IIC 控制器的兼容性，由于 OV5640 和 OV7725 的器件地址、地址长度、需要写入的寄存器个数都不一样，所以必须在 camera_init 层明确 IIC 控制器需要初始化的类型是 OV5640-RGB，OV5640-JPEG，OV7725-RGB 或 OV7725-JPEG。

如果摄像头的类型明确了，device_id，addr_mode，写入寄存器 rom 内的值的位宽、值的含义就明确了。进一步通过条件判断明确是 RGB 格式还是 JPEG 格式后，需填写的寄存器数量也明确了。

在上面的源码语句中，通过 generate——endgenerate 语句，实现了对摄像头类型、像素格式的选择。上述源码先判断了摄像头类型是 OV5640 还是 OV7725，确认后，则对变量 device_id，addr_mode，addr，wrdata 四个变量进行赋值，以明确器件 ID，摄像头类型定义，摄像头寄存器地址和写入数据设定值。由于摄像头在 rgb 模式和 jpeg 模式下寄存器个数不同，所以必须先通过工程源码的顶层定义识别出是 RGB 模式还是 jpeg 模式，才能给出摄像头寄存器个数的准确定义数值。

例如，顶层代码给出了摄像头的型号为 ov5640，通过查阅 ov5640 的技术手册，可以明确 ov5640 的 IIC 器件 ID 号为 device_id = 8’h78。此时定义 addr_mode = 1，addr = lut[23:8]，wrdata = lut[7:0]。addr_mode、addr 和 wrdata 共同配合，以便于给 ov5640 的寄存器地址赋初值。

代码所在模块 ： i2c_control
`assign reg_addr = addr_mode?addr:{addr[7:0],addr[15:8]};`

在 i2c_control 模块中，通过给变量 reg_addr 的赋值，就能够准确的告知 IIC 控制器向 ov5640 寄存器写入的数据格式，包括有效地址是低地址还是高地址在前。

由于 ov5640 摄像头配置寄存器位宽为 24 位，ov7725 摄像头配置寄存器位宽为 16 位，地址信息通过 addr_mode 和 addr 两个变量传递给 IIC 控制模块，即可在 IIC 模块中解析出摄像头类型和寄存器位宽。

又比如，以 ov5640_rgb 格式图像翻转控制器进行讲解。OV5640 摄像头数据手册定义寄存器地址号为 0x3820 控制成像是否翻转。官方手册指导：该寄存器默认值为 0x40，如果成像需要翻转，则将该寄存器默认值设为 0x47 或 0x46（表中未设定该寄存器最低位 Bit[0]的含义，所以该位为无关位）。

0x3819	HSYNC WIDTH	0x00	RW	Bit[7:4]: Debug mode Bit[3:0]: HSYNC width[7:8]
0x3820	TIMING TC REG20	0x40	RW	Timing Control Bit[7:3]: Debug mode Bit[2]: ISP vflip Bit[1]: Sensor vflip
0x3821	TIMING TC REG21	0x00	RW	Timing Control Bit[7:6]: Debug mode Bit[5]: JPEG enable Bit[4:3]: Debug mode Bit[2]: ISP mirror Bit[1]: Sensor mirror Bit[0]: Horizontal binning enable

图 41-3 OV5640 数据手册关于摄像头翻转控制的表格说明

代码所在模块 ： ov5640_init_table_rgb

```
localparam IMAGE_FLIP_DAT    = IMAGE_FLIP_EN ? 8'h47 : 8'h40;
```

上方使用条件判断语句及使用本地参数化的方法明确 IMAGE_FLIP_DAT 的值，随后 rom[211]的内容即可明确。

代码所在模块 ： ov5640_init_table_rgb

```
rom[211] = {16'h3820, IMAGE_FLIP_DAT}; // flip
```

从顶层传递而来的 IMAGE_FLIP_EN 信号，决定了写入的寄存器初始值是 8’h47 或 8’h40。

然后在摄像头初始化寄存器参数表中，以位拼接的格式，对 rom[211]寄存器进行赋值。lut 值的位宽决定了 rom 地址的位宽，一个 rom 地址的位宽为 24 位，高 16 位为地址号，低 8 位为初值。

理解了上面的内容，读者可以尝试结合代码，自行尝试修改 rom[211]，即 16’h3820 寄存器的值。读者可以分别尝试直接将 IMAGE_FLIP_DAT 替换为 8’h47，8’h46，8’h40，以直接观察输出效果获得图像翻转的直观体验。

41.5.2RGB 和 JPEG 两种模式寄存器差异地方

对于 RGB 和 JPEG 两种模式时寄存器的差异，可以采用文件对比工具的方式筛选出来，下面为筛选好的两者之间的差异内容。其实主要区别如下：

41.5.2.1JPEG 模式

```
rom[56] = 24'h4300_60;
...

rom[208] = 24'h4300_30; // YUV 422, YUYV
rom[209] = 24'h501f_00; // YUV 422

// 12824'h720, 30fps
// input clock 24Mhz, PCLK 42Mhz
```

```
rom[210] = 24'h3035_11; // PLL JPEG mode 11->15fps;21->7.5fps;
41->3.75fps;
rom[211] = 24'h3036_69; // PLL
rom[212] = 24'h3c07_07; // lightmeter 1 threshold[7:0]
rom[213] = {16'h3820, IMAGE_FLIP};    // flip
rom[214] = {20'h38212, IMAGE_MIRROR}; // no mirror
rom[215] = 24'h3814_11; // timing X inc
rom[216] = 24'h3815_11; // timing Y inc
rom[217] = 24'h3800_00; // HS
rom[218] = 24'h3801_00; // HS
rom[219] = 24'h3802_00; // VS
rom[220] = 24'h3803_00; // VS
rom[221] = 24'h3804_0a; // HW SET_OV5640 +  HE}
rom[222] = 24'h3805_3f; // HW SET_OV5640 +  HE}
rom[223] = 24'h3806_07; // VH SET_OV5640 +  VE}
rom[224] = 24'h3807_9f; // VH SET_OV5640 +  VE}
rom[225] = {16'h3808, IMAGE_WIDTH[15:8]}; // DVPHO  (<1280>500)
(<640>280)IMAGE_WIDTH
rom[226] = {16'h3809, IMAGE_WIDTH[ 7:0]}; // DVPHO
rom[227] = {16'h380a, IMAGE_HEIGHT[15:8]}; // DVPVO
(<720>2d0)  (<480>1e0)IMAGE_HEIGHT
rom[228] = {16'h380b, IMAGE_HEIGHT[ 7:0]}; // DVPHO
rom[229] = 24'h380c_0b; // HTS
rom[230] = 24'h380d_1c; // HTS
rom[231] = 24'h380e_07; // VTS
rom[232] = 24'h380f_b0; // VTS
rom[233] = 24'h3813_04; // timing V offset
rom[234] = 24'h3618_04;
rom[235] = 24'h3612_2b;
rom[236] = 24'h3709_12;
rom[237] = 24'h370c_00;
rom[238] = 24'h4004_06; // BLC line number
rom[239] = 24'h3002_00; // reset JFIFO, SFIFO, JPG
rom[240] = 24'h3006_ff; // disable clock of JPEG2x, JPEG
rom[241] = 24'h4713_03; // JPEG mode 3
rom[242] = 24'h4407_01; // Quantization scale
rom[243] = 24'h460b_35;
rom[244] = 24'h460c_22;
rom[245] = 24'h4837_16; // MIPI global timing
rom[246] = 24'h3824_02; // PCLK manual divider
rom[247] = 24'h5001_a3; // SDE on, CMX on, AWB on
rom[248] = 24'h3503_00; // AEC/AGC on
rom[249] = 24'h4740_20; // VS 1   //
```

41.5.2.2RGB 模式

```
rom[56] = 24'h4300_61; // RGB565   //h4300_6f(千兆网模式)
```

```
...

// 12824'h720, 30fps
// input clock 24Mhz, PCLK 42Mhz
rom[208] = 24'h3035_21; // PLL  21:30fps  41:15fps  81:7.5fps
rom[209] = 24'h3036_69; // PLL
rom[210] = 24'h3c07_07; // lightmeter 1 threshold[7:0]
rom[211] = {16'h3820, IMAGE_FLIP}; // flip
rom[212] = {20'h38210, IMAGE_MIRROR}; // no mirror
rom[213] = 24'h3814_31; // timing X inc
rom[214] = 24'h3815_31; // timing Y inc
rom[215] = 24'h3800_00; // HS
rom[216] = 24'h3801_00; // HS
rom[217] = 24'h3802_00; // VS
rom[218] = 24'h3803_fa; // VS
rom[219] = 24'h3804_0a; // HW SET_OV5640 +  HE}
rom[220] = 24'h3805_3f; // HW SET_OV5640 +  HE}
rom[221] = 24'h3806_06; // VH SET_OV5640 +  VE}
rom[222] = 24'h3807_a9; // VH SET_OV5640 +  VE}
rom[223] = {16'h3808, IMAGE_WIDTH[15:8]}; // DVPHO  (<1280>500)
(<640>280)IMAGE_WIDTH
rom[224] = {16'h3809, IMAGE_WIDTH[ 7:0]}; // DVPHO
rom[225] = {16'h380a, IMAGE_HEIGHT[15:8]}; // DVPVO
(<720>2d0)  (<480>1e0)IMAGE_HEIGHT
rom[226] = {16'h380b, IMAGE_HEIGHT[ 7:0]}; // DVPHO
rom[227] = 24'h380c_07; // HTS
rom[228] = 24'h380d_64; // HTS
rom[229] = 24'h380e_02; // VTS
rom[230] = 24'h380f_e4; // VTS
rom[231] = 24'h3813_04; // timing V offset
rom[232] = 24'h3618_00;
rom[233] = 24'h3612_29;
rom[234] = 24'h3709_52;
rom[235] = 24'h370c_03;
rom[236] = 24'h3a02_02; // 60Hz max exposure
rom[237] = 24'h3a03_e0; // 60Hz max exposure
rom[238] = 24'h3a14_02; // 50Hz max exposure
rom[239] = 24'h3a15_e0; // 50Hz max exposure
rom[240] = 24'h4004_02; // BLC line number
rom[241] = 24'h3002_1c; // reset JFIFO, SFIFO, JPG
rom[242] = 24'h3006_c3; // disable clock of JPEG2x, JPEG
rom[243] = 24'h4713_03; // JPEG mode 3
rom[244] = 24'h4407_04; // Quantization scale
rom[245] = 24'h460b_37;
rom[246] = 24'h460c_20;
rom[247] = 24'h4837_16; // MIPI global timing
rom[248] = 24'h3824_04; // PCLK manual divider
```

```
rom[249] = 24'h5001_83; // SDE on, CMX on, AWB on
rom[250] = 24'h3503_00; // AEC/AGC on
rom[251] = 24'h4740_20; // VS 1
```

41.5.3摄像头初始化之复位延时控制

OV5640 摄像头提供了专门的硬件复位管脚 camera_rst_n，通过拉低该管脚一定的时间，能够让 OV5640 的所有寄存器恢复到未经过配置的初始化状态。设计时，也推荐将该信号接到 FPGA 的管脚上，通过 FPGA 来控制。

但是在实际使用中发现，用户所使用的 FPGA 板不一样，有的摄像头接口能够给摄像头模块提供复位信号的控制，而有的则无对应管脚连接，所以无法对摄像头进行硬件复位。同时由于摄像头的供应商不一样，部分厂家的摄像头模块并未将这个硬件复位管脚引出，导致 FPGA 板即使有管脚，也无法控制到摄像头的硬件复位管脚。

为了解决这个问题，我们可以使用软件复位的方式。所谓软件复位，就是对摄像头的复位寄存器写入特定值，以设置摄像头进入复位状态。这样，无论 FPGA 和摄像头模式是否有条件进行复位，都能确保使用软件复位的方式，让摄像头进入正确的复位状态。

对于 OV5640 摄像头，其 0x3008 寄存器的 bit7 位就是软件复位位，只需要将该位置 1，然后等待一定时间（2~5ms）后再设置该位为 0，即可将 OV5640 的所有寄存器恢复到初始状态。

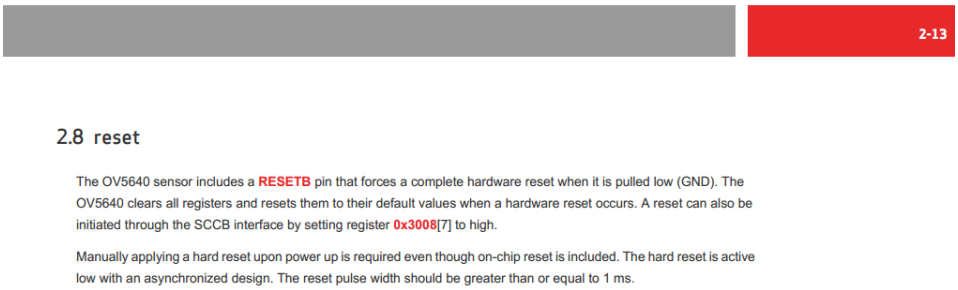


图 41-4 OV5640 摄像头技术手册对复位方式的官方描述

table 7-1　system and IO pad control registers (sheet 3 of 7)

address	register name	default value	R/W	description
0x3007	CLOCK ENABLE03	0xFF	RW	Clock Enable Control (0: disable clock; 1: enable clock) Bit[7]: Enable digital gain compensation clock Bit[6]: Enable SYNC FIFO clock Bit[5]: Enable ISPFC SCLK clock Bit[4]: Enable MIPI PCLK clock Bit[3]: Enable MIPI clock Bit[2]: Enable DVP PCLK clock Bit[1]: Enable VFIFO PCLK clock Bit[0]: Enable VFIFO SCLK clock
0x3008	SYSTEM CTROL0	0x02	RW	System Control Bit[7]: Software reset Bit[6]: Software power down Bit[5:0]: Debug mode
0x3009	DEBUG MODE	–	–	Debug Mode

图 41-5 OV5640 摄像头数据手册对于复位寄存器 0c3008 的赋值描述

具体来说，OV5640 摄像头提供了硬件复位管脚复位和软件写复位寄存器来复位两种复位方式。

硬件复位：通过对 OV5640 芯片的复位管脚拉低 1ms 左右实现完成复位

软件复位：通过 0x3008 寄存器的 bit[7]位写入 1 来进入复位，写入 0 来退出复位。

从官方的技术手册要求来看，采用软件复位而放弃对 OV5640 复位管脚的控制，并不是无条件的。官方手册要求：OV5640 如果进行软件复位，需在配置复位寄存器后，产生至少 2ms 至 5ms 的等待延时。下方即为官方手册对该要求的原文：

SCCB address is 0x42/0x43 for VGA sensors

SCCB address is 0x60/0x61 for 1.3M and 2.0M sensors

SCCB address is 0x78/0x79 for 3.0M ， 5.0M sensors

f. If SCCB soft reset is used, please wait at least 2~5ms after SCCB soft rest.

7.4 Check Camera Interface

a. Check polarity of HREF(HSYCN), VSYNC, PCLK, make sure the polarity of camera module matches with backend or baseband side.

图 41-6 OV5640 Camera Module Hardware Application Notes 关于复位延时的描述原文

合理的复位延时，能保证 rom[1]后续寄存器值的正确写入，否则，如果设置软件复位后立即退出复位，继续写后续寄存器，会因为当前摄像头还处在复位逻辑中，继而导致此期间写寄存器的值不会生效。这样就会导致最后紧跟在复位操作后的若干个寄存器没有正确初始化，从而影响正确的成像。我们的初始化表的第二个寄存器，就是对复位寄存器进行设置，以让 OV5640 进入复位状态。

代码所在模块 ： ov5640_init_table_rgb

```
rom[1 ] = 24'h3008_82; // software reset, bit[7] //delay 5ms
```

综合以上分析，可知上方代码中 rom[1]的意义，以及正确实现软件复位需要的 5ms 延时要求。

如果能够通过程序正确进行上述操作，那么 OV5640 的硬件复位管脚 camera_rst_n 即使不连接 FPGA 管脚，而是以硬件的方式设置为永久的高电平，也可以正常工作。

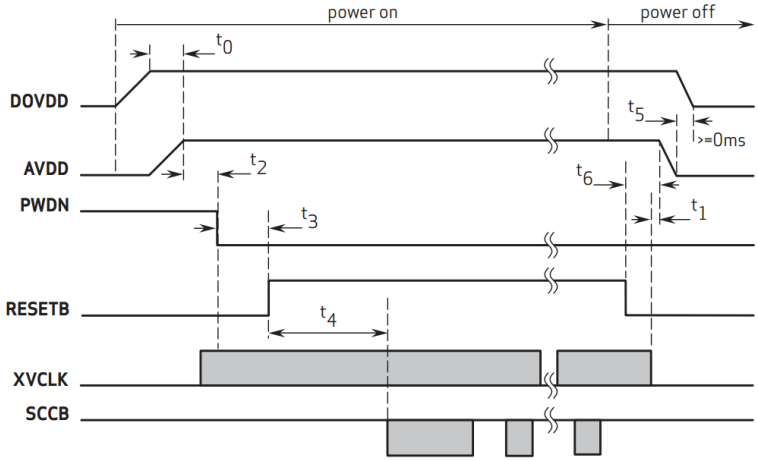


图 41-7 OV5640 摄像头稳定采集数据时 PWDN 和 RESETB 为固定的初始值即可

接下来，我们解决如何在模块内部给出 camera_pwdn 和 camera_rst_n 的初值定值，同时通过设定延时时长的方法，保证复位控制寄存器 rom[1]设定后能有至少 5ms 的确认时间的问题。

代码所在模块 ： camera_init

```
assign camera_pwdn = 0;
```

代码所在模块 ： camera_init

```
assign camera_rst_n = 1;
```

给定初值后，即需开始解决写入 rom[1]后延时 5ms 的问题。首先，我们看到 i2c_control 模块中，设计的状态机有控制数据写入完成后延时的功能。

代码所在模块 ： i2c_control

```
WAIT_DLY:
begin
  if(dly_cnt <= dly_cnt_max) begin
    dly_cnt <= dly_cnt + 1'b1;
    state <= WAIT_DLY;
  end
  else begin
    dly_cnt <= 0;
    RW_Done <= 1'b1;
```

```
    state <= IDLE;
  end
end
```

在 i2c_control 模块中，我们设置了等待延迟状态，并通过 dly_cnt 计数器来完成时钟周期的计数等待。当 dly_cnt 计数器完成延时，即计数到和 dly_cnt_max 相等时，i2c 状态机才会回到写 i2c 操作的起始状态进而开始下一次的数据读写。这样看来，我们可以通过设定 dly_cnt_max 的值，来控制延迟计数器的计数次数，从而设定一次 i2c 读写完成后的等待延迟时间。

从 ov5640 的寄存器配置手册来看，大部分寄存器配置都没有延迟要求。或者一次 i2c 的读写和存储，并不会影响到其他后续寄存器的写入效果，而唯独复位寄存器较为特殊。而 rom 地址，在 camera_init 模块中，又是和变量 cnt 对应的。这样，在 camera_init 模块的设计中，我们就可以将 rom 地址和延迟总时长对应起来，实现专门对 rom[1]进行一个较长时间的延迟。

代码所在模块 ： camera_init

```
always@(posedge Clk or negedge Rst_n)
if(~Rst_n)
  cnt <= 0;
else if(Go)
  cnt <= 0;
else if(cnt < lut_size)begin
  if(RW_Done && (!ack))
    cnt <= cnt + 1'b1;
  else
    cnt <= cnt;
end
else
  cnt <= 0;

......     ......
reg [1:0]state;

always@(posedge Clk or negedge Rst_n)
if(~Rst_n)begin
  state <= 0;
  wrreg_req <= 1'b0;
  i2c_dly_cnt_max <= 32'd0;
end
else if(cnt < lut_size)begin
  case(state)
    0:
      if(Go)
        state <= 1;
```

```
      else
        state <= 0;
    1:
      begin
        wrreg_req <= 1'b1;
        state <= 2;
        if(cnt == 1)
//////给出满足延时不小于 5ms 的 16 进制计数器值//////
          i2c_dly_cnt_max <= 32'h40000;
        else
          i2c_dly_cnt_max <= 32'd0;
      end
    2:
      begin
        wrreg_req <= 1'b0;
        if(RW_Done)
          state <= 1;
        else
          state <= 2;
      end
    default:state <= 0;
  endcase
end
else
  state <= 0;
```

我们在进行 rom[1]的寄存器赋值操作时，即当 cnt 为 1 时，专门对于 cnt 延迟控制计数器进行特殊赋值，即对 i2c_dly_cnt_max 赋值为 32’h40000，而对其他 rom 地址保持 i2c_dly_cnt_max 为默认值 0。由于主时钟的时钟周期为 20ns，这样，计数 32’h40000,就是计数十进制的 262144 拍，就可以满足大于 5ms 的延迟。

通过例化过程，camera_init 模块的 i2c_dly_cnt_max，和 i2c_control 模块的 dly_cnt_max 信号对应，完成计数器最大值设置的传递。

代码所在模块 ： camera_init

```
i2c_control i2c_control(
  .Clk         (Clk             ),
  .Rst_n       (Rst_n           ),
  ......  ......
  .dly_cnt_max (i2c_dly_cnt_max ),
  .i2c_sclk    (i2c_sclk        ),
  .i2c_sdat    (i2c_sdat        )
);
```

这样，i2c_control 模块就可以直接通过已有的延时状态，来实现 rom[1]寄存器写完后产生不低于 5ms 的延时这一任务。随之就不会发生 rom[1]复位配置过

程中，rom[2]，rom[3].....又在同时写入的现象。

41.6硬件管脚复位和 Init_done 信号的生成

上一小节我们讲解 ov5640 摄像头的寄存器控制复位实现方法。在本节中，我们将讲解 ov5640 硬件复位控制方法，并通过硬件复位结束的延时信号，将 Init_done 信号强制拉低，直到所有摄像头初始化寄存器写入完成后，再拉高。

OV5640 官方技术手册要求，摄像头模组需上电复位（注意，这里的复位和寄存器配置上的复位不同，是指硬件管脚复位）完成 20ms 后再配置摄像头。原始文档的时序图如下：

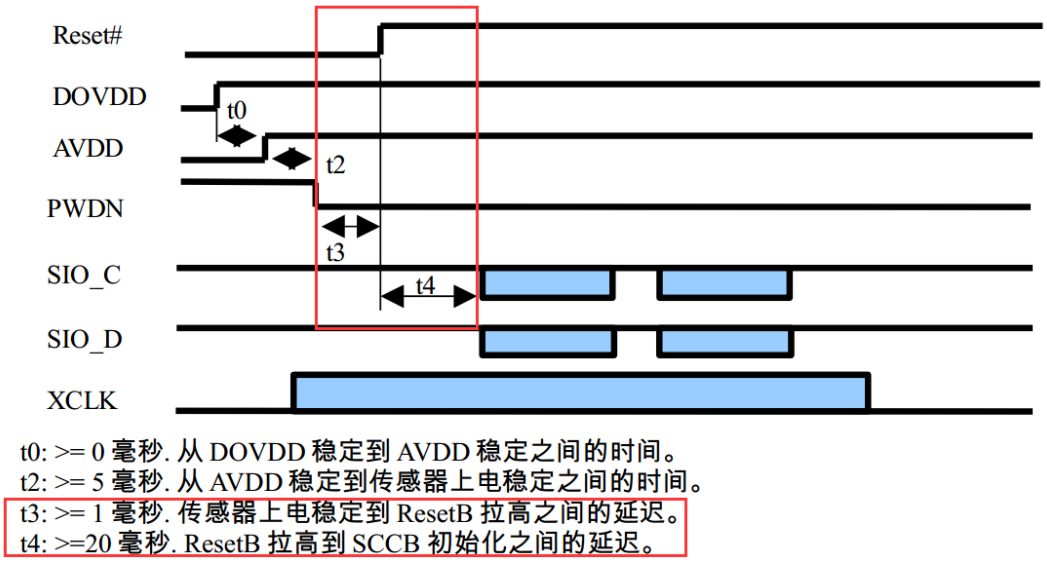


图 41-8 ov5640_自动对焦照相模组应用指南对上电、复位和 SCCB 初始化的延时要求

在实际硬件复位设计时，只需要使用一个计数器作延时使用，即可实现上述时序图中的延时要求。以下为 camera_init 代码中设计的复位时序的相关代码。

针对以上要求，我们作出如下设计：

代码所在模块 ： camera_init

```
//上电并复位完成 20ms 后再配置摄像头，所以从上电到开始配置应该是 1.0034 + 20
= 21.0034ms
//这里为了优化逻辑，简化比较器逻辑，直接使延迟比较值为 24'h100800，是
21.0125ms
always@(posedge Clk or negedge Rst_n)
if(~Rst_n)
  delay_cnt <= 21'd0;
else if (delay_cnt == 21'h100800)
  delay_cnt <= 21'h100800;
else
  delay_cnt <= delay_cnt + 1'd1;
```

```
//当延时时间到，开始使能初始化模块对 OV5640 的寄存器进行写入
assign Go = (delay_cnt == 21'h1007ff) ? 1'b1 : 1'b0;
```

上方代码注释很清晰的告诉了我们设定延时最大值的来历。每个时钟周期，delay_cnt 自加 1，直到 delay_cnt 值达到 21'h100800，如果 delay_cnt == 21'h1007ff，说明在下一个时钟周期，delay_cnt == 21'h100800 条件即将满足，将 Go 拉高一个时钟周期。

代码所在模块 ： camera_init

```
always@(posedge Clk or negedge Rst_n)
if(~Rst_n)
  Init_Done <= 0;
else if(Go)
  Init_Done <= 0;
else if(cnt == lut_size)
  Init_Done <= 1;
```

Init_Done 信号值的判断，优先判断 Go 的值，Init_Done 初始值为 0，在下一时钟周期，delay_cnt == 21'h100800 条件得到满足，而 Go 立即从 1 变为 0，开始等待 rom 寄存器配置完成。当满足条件 cnt == lut_size 表明所有寄存器配置完成，Init_Done 信号拉高输出。

41.7OV5640 DVP 接口时序逻辑设计

上一节，我们已经完成了 OV5640 初始化逻辑的介绍。接下来，将要开始完成 DVP 接口的时序设计。

为了便于分析 DVP 接口。这里以 OV5640 输出 3*3 像素的图像大小矩阵为例绘制时序图，介绍 DVP 接口的时序，并分析接口逻辑设计方法。

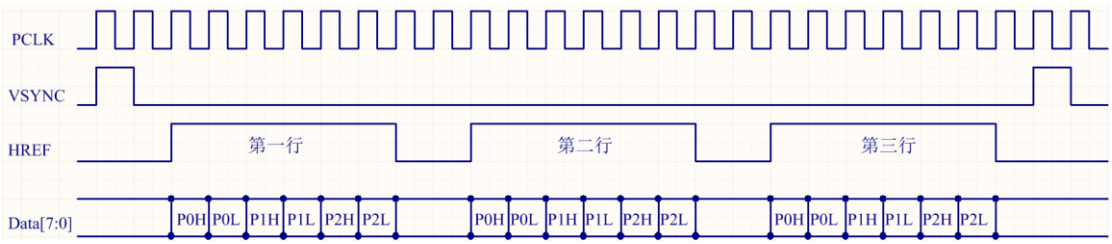


图 41-9 OV5640 DVP 接口时序逻辑图

特别说明：上图仅为时序示意图，其中 VSYNC 的高电平脉冲宽度，VSYNC 下降沿到 HREF 上升沿之间的间隔时钟个数，HREF 下降沿到下一个 HREF 的上升沿之间的高电平之间的时钟个数，以及最后一个 HREF 下降沿到 VSYNC 上升沿之间的间隔时钟个数实际上远大于图中所绘制，绘制时为了保证

画幅大小和进行了精简，这些差异不影响我们分析 DVP 接口时序。

- VSYNC 的高脉冲标志着新一帧图像数据的即将到来。所以当 VSYNC 高脉冲出现之后，第一个 HREF 高电平期间 DATA 端口上传输的就是整幅图像的第一行数据，紧接着第二行，直到最后一行输出完成，再产生 VSYNC 的高脉冲开始新一帧图像数据的输出。
- HREF 上升沿后的第一个时钟时刻的数据为该行图像的第一个像素数据的高字节（P0H），第二个时钟时刻的数据为第一个像素数据的低字节（P0L），以此类推，直到第 2N 个数据为第 N 个像素数据的低字节，然后一行数据输出完成，HREF 变为低电平。间隔一定时钟周期后再开始下一行图像数据的输出。
- Data 数据线上，PxH 和 PxL 两个 8 位的数据拼接为一个 RGB565 像素的图像数据。

41.7.1基本数据流接收

根据应用需求，整个 DVP Capture 模块的设计目的就是要实现每两个数据拼接为 1 个 16 位的数据并按照写 RAM 或 FIFO 的接口形式输出，模块设计框图如下图所示：

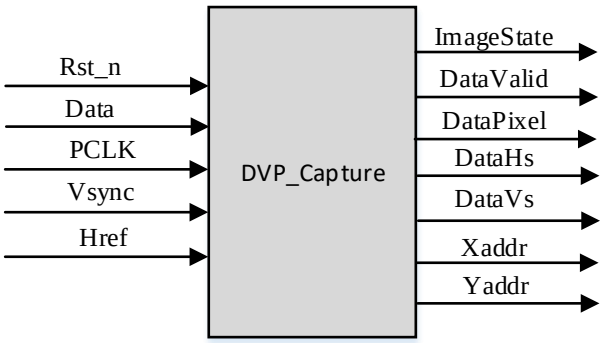


图 41-10 DVP 接口数据流接收模块输入输出接口

模块中，DataPixel 为 16 位的 RGB565 格式的像素数据，由连续的 2 个 Data 数据拼接而来。DataValid 为 DataPixel 数据有效标志信号，由于 DATA 端口需要 2 个时钟才能传输一个像素所需的 16 位数据，所以 DataPixel 端口上的数据理论来说应该是每 2 个时钟周期只有一个时钟周期是真正有效的，所以 DataValid 在连续的两个时钟周期中，只有一个时钟周期为高电平，一个时钟周期为低电平，来确保下一级在使用 DataPixel 时，每两个时钟周期内只使用一次，使用时，如果是数据直接写入 FIFO 或者 RAM，可以直接将 DataValid 信号当做 wrreq 信号

使用。下图 41-11 为该模块的时序图：

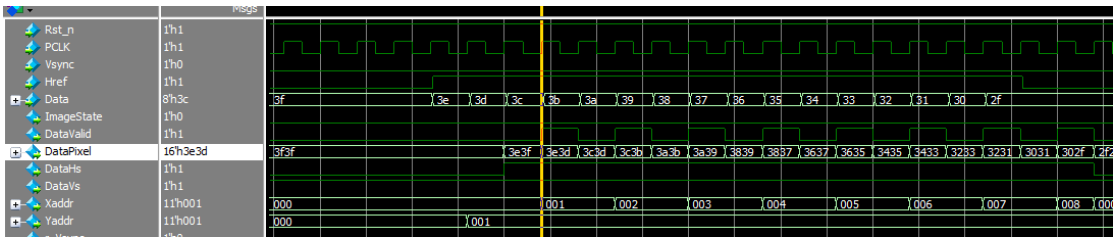


图 41-11 DVP 接口模块时序图

从图中可以看到，DataPixel 的数据在 DataValid 信号为高电平的时候，输出的是前两个时钟周期的值拼接成的。如 C1 和 C2 组成一个 16 位的数据，在 C2 之后延迟两个数据输出在 DataPixel 端口上。这样就完成了 DVP 接口数据流转 RGB565 数据流的功能。

41.7.2像素位置输出

在有些应用中，需要实时知道当前输出的图像数据在输出像素矩阵中的绝对位置，以便于根据不同的位置进行不同的处理。最典型的如使用图像处理技术定位图像中某个点的具体位置。则需要用到该信息，因此，该 DVP Capture 模块也将每个像素对应的位置通过 Xaddr 和 Yaddr 两个端口输出。在上图中以 C1C2 这个数据为例，可以看到，在 DataValid 为高电平的时候，数据为 C1C2，而此时的 Xaddr 也为 1。每当 DataValid 高电平信号到来时，Xaddr 的值加 1。当一行图像数据输出完成之后，Xaddr 清零。

Xaddr 和 Yaddr 的生成非常简单。对于 Xaddr 来说，只要设计一个 Hcount 计数器，在 HREF 为高电平期间持续计数即可，然后舍去 Hcount 的最低位得到的值就刚好可以与像素输出时刻一一对应。而对于 Yaddr，则只需要使用一个 Vcount 计数器对 HREF 的上升沿进行计数即可。关于该部分实现的具体细节详见提供的设计代码的完整内容。

需要关注的是，在模块中，Xaddr 和 Yaddr 都是从 1 开始算的，并非从 0 开始，也就是说，输出的每行最开头的一个像素对应的 Xaddr 的值为 1，输出的每帧图像的第一行数据，其 Yaddr 的值也为 1。

41.7.3舍弃前 N 张图像

在某些工程师的应用说明中有提到，一般 CMOS 摄像头开始工作后，其起始的几张图像是不稳定的。关于这种不稳定的情况，有两种解读，一是输出的

图像数据量不稳定，例如本来应该每行输出 800 个像素点的数据，但是实际上只输出了不到 800 个或者超过 800 个数据。另一种解读是刚开始输出的这几幅图像数据量稳定，但其颜色失真较大，所以不建议取用。在笔者实际设计调试的过程中，即使直接使用起始时候的图像，也暂未发现输出数据量异常的情况，图像数据量应该是稳定的。另外对于颜色失真较大的可能，由于没有专门采集和分析起始几张图像的质量，也暂未得到认证。为了兼顾这种可能出现的情况。在 DVP Capture 中还是对起始的 10 帧图像做了舍弃处理。使用一个 FrameCnt 计数器计数 VSYNC 的上升沿，每个上升沿就意味着一帧图像即将开始，当该计数值达到 10 之后，在使能 DataValid 信号，从而确保了前 10 帧图像的数据不会输出。关于该部分实现的具体细节详见提供的设计代码的完整内容。

41.7.4系统异常状态恢复控制

在实际应用的过程中，笔者还遇到了这样一个问题。那是在将图像数据写入 fifo 的一个应用中，由于系统运行过程中突然被复位，然后重新开始运行，此时 fifo 中已经有写入了一部分数据还没来得及读走，如果 fifo 没有在系统复位后也执行相应的清零操作，这部分数据会依旧保留在 fifo 内，读取端在复位之后，默认会认为 fifo 中的第一个数据就是 Xaddr 和 Yaddr 都为 1 的那个数据，如果读取端不借助 Xaddr 和 Yaddr 的值，就会导致真正的第一个数据的位置判断错误。为了简化逻辑设计并解决这个问题，可以通过合理的设计，让系统从复位中恢复之后，该 FIFO 在摄像头的第一个数据（这里指摄像头开始工作后 DVP Capture 模块输出的一个数据，非每帧或每行图像的第一个数据）开始向其写入之前，先执行一次清零操作，让 FIFO 中的残留数据先清零，然后再向其中写入时，就能与 FIFO 的读取端对位置的判定方式一致了。所以在逻辑中设计了一个图像状态信号 ImageState，当系统进入复位状态后该信号变为高电平，只有当系统从复位中恢复且 DVP 接口的第一个 VSYNC 高电平出现时，再将该信号拉低，实际在应用过程中，可以直接将 ImageState 信号连接到 FIFO 的清零端口（aclr），就能实现对 FIFO 的合理清零了，关于该部分实现的具体细节详见提供的设计代码的完整内容。

整个设计逻辑的代码如下所示：

```
module DVP_Capture(
  input           Rst_n,
  input           PCLK,
  input           Vsync,
  input           Href,
```

```
  input     [7:0]  Data,

  output reg       ImageState,
  output           DataValid,
  output    [15:0] DataPixel,
  output           DataHs,
  output           DataVs,
  output    [10:0] Xaddr,
  output    [10:0] Yaddr
);

  reg       r_Vsync;
  reg       r_Href;
  reg [7:0] r_Data;

  reg [15:0]r_DataPixel;
  reg       r_DataValid;
  reg       r_DataHs;
  reg       r_DataVs;
  reg [11:0]Hcount;
  reg [10:0]Vcount;
  reg [3:0] FrameCnt;

  reg       dump_frame;

  //等到初始化摄像完成且头场同步信号出现，释放清零信号，开始写入数据
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    ImageState <= 1'b1;
  else if(r_Vsync)
    ImageState <= 1'b0;

  //对 DVP 接口的数据使用寄存器打一拍，以用信号边沿检测功能
  always@(posedge PCLK)
  begin
    r_Vsync <= Vsync;
    r_Href  <= Href;
    r_Data  <= Data;
  end

  //在 HREF 为高电平时，计数输出数据个数
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    Hcount <= 0;
  else if(r_Href)
    Hcount <= Hcount + 1'd1;
  else
```

```
    Hcount <= 0;

  /*根据计数器的计数值奇数和偶数的区别，在计数器为偶数时，
  将 DVP 接口数据端口上的数据存到输出像素数据的高字节，在计
  数器为奇数时，将 DVP 接口数据端口上的数据存到输出像素数据
  的低字节*/
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    r_DataPixel <= 0;
  else if(!Hcount[0])
    r_DataPixel[15:8] <= r_Data;
  else
    r_DataPixel[7:0] <= r_Data;

  /*在行计数器计数值为奇数，且 HREF 高电平期间，产生输出
   数据有效信号*/
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    r_DataValid <= 0;
  else if(Hcount[0] && r_Href)
    r_DataValid <= 1;
  else
    r_DataValid <= 0;

  always@(posedge PCLK)
  begin
    r_DataHs <= r_Href;
    r_DataVs <= ~r_Vsync;
  end

  /*使用 Vcount 计数器对 HREF 信号的高电平进行计数，统计
  一帧图像中的每一行图像的行号*/
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    Vcount <= 0;
  else if(r_Vsync)
    Vcount <= 0;
  else if({r_Href,Href} == 2'b01)
    Vcount <= Vcount + 1'd1;
  else
    Vcount <= Vcount;

  /*输出 X 地址*/
  assign Yaddr = Vcount;

  /*由于一行 N 个像素的图像输出 2N 个数据，所以 Hcount 计数
  值为 N 的 2 倍，将该计数值除以 2 后即可作为 Xaddr 输出*/
```

```
  assign Xaddr = Hcount[11:1];

  /*帧计数器，对每次系统开始运行后的前 10 帧图像进行计数*/
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    FrameCnt <= 0;
  else if({r_Vsync,Vsync}== 2'b01)begin
    if(FrameCnt >= 10)
      FrameCnt <= 4'd10;
    else
      FrameCnt <= FrameCnt + 1'd1;
  end
  else
    FrameCnt <= FrameCnt;

  /*舍弃每次系统开始运行后的前 10 帧图像的数据，以确保输出图像稳定*/
  always@(posedge PCLK or negedge Rst_n)
  if(!Rst_n)
    dump_frame <= 0;
  else if(FrameCnt >= 10)
    dump_frame <= 1'd1;
  else
    dump_frame <= 0;

  assign DataPixel = r_DataPixel;
  assign DataValid = r_DataValid & dump_frame;
  assign DataHs = r_DataHs & dump_frame;
  assign DataVs = r_DataVs & dump_frame;

endmodule
```

同时，设计一个 DVP 信号发生测试逻辑，用来对设计的 DVP 接口逻辑进行仿真测试，该部分设计代码思路很简单，只要按照 DVP 接口的时序要求产生 PCLK、HREF、VSYNC、DATA 端口的数据即可，该测试文件代码如下所示：

```
`timescale 1ns/1ns

module DVP_Capture_tb;

  reg         Rst_n;
  reg         PCLK;
  reg         Vsync;
  reg         Href;
  reg  [7:0] Data;

  wire        ImageState;
  wire        DataValid;
```

```
wire [15:0]DataPixel;
wire       DataHs;
wire       DataVs;
wire [11:0]Xaddr;
wire [11:0]Yaddr;

DVP_Capture DVP_Capture(
  .Rst_n      (Rst_n      ),//input
  .PCLK       (PCLK       ),//input
  .Vsync      (Vsync      ),//input
  .Href       (Href       ),//input
  .Data       (Data       ),//input     [7:0]

  .ImageState (ImageState ),//output reg
  .DataValid  (DataValid  ),//output
  .DataPixel  (DataPixel  ),//output    [15:0]
  .DataHs     (DataHs     ),//output
  .DataVs     (DataVs     ),//output
  .Xaddr      (Xaddr      ),//output    [11:0]
  .Yaddr      (Yaddr      ) //output    [11:0]
);

initial PCLK = 1;
always#40 PCLK = ~PCLK;

parameter WIDTH = 16;
parameter HIGHT = 12;

integer i,j;

initial begin
  Rst_n = 0;
  Vsync = 0;
  Href  = 0;
  Data  = 8'hff;
  #805;
  Rst_n = 1;
  #400;

  repeat(15)begin
    Vsync = 1;
    #320;
    Vsync = 0;
    #800;
    for(i=0;i<HIGHT;i=i+1)
    begin
      for(j=0;j<WIDTH;j=j+1)
```

```
        begin
          Href = 1;
          Data = Data - 1;
          #80;
        end
        Href = 0;
        #800;
      end
    end
    $stop;
  end

endmodule
```

41.8系统板级测试

在本系统中，共用到 6 路时钟，clk50m、loc_clk50m、loc_clk200m、camera_xclk、clk_disp、camera_pclk。

- clk50m：第一路为由外部有源晶体振荡器提供的基本工作时钟，为 50MHz，名为 clk50m。该时钟主要有两个作用，第一个作用是作为 FPGA 片上锁相环 PLL 的基本时钟。
- loc_clk50m：作为 camera_init 模块的工作时钟，用来对 OV5640 的寄存器进行初始化操作。
- clk_200M：作为 DDR3 控制器工作时钟
- camera_xclk：由 FPGA 片上 PLL 提供，时钟频率为 24MHz，通过 IO 口输出，连接到 OV5640 的基本工作时钟脚上。作为 OV5640 的基本工作时钟。
- clk_disp：由 PLL 提供，作为显示屏控制器基本工作时钟。由于 800*480 分辨率的 TFT 显示屏，其工作在 60Hz 的刷新率的情况下要求的像素时钟频率为 33MHz。（如果是 480*272 分辨率的 TFT 显示屏，其工作在 60Hz 的刷新率的情况下要求的像素时钟频率为 9MHz）
- camera_pclk：摄像头输出的像素时钟，该时钟频率会因不同的输出分辨率和帧率而有所差别，该时钟由 OV5640 通过管脚直接输入到 FPGA 内部，本次设计作为 DVP_Capture 模块和数据写入缓存的工作时钟。

系统顶层代码根据设计框图进行连接即可，与串口传图顶层类似。详见提供的例程。

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。对工程的管脚和时钟进行约束后，生成 Bit 文件。上板调试硬件平台基于高云开发板。高云开发板连接示意图如下：

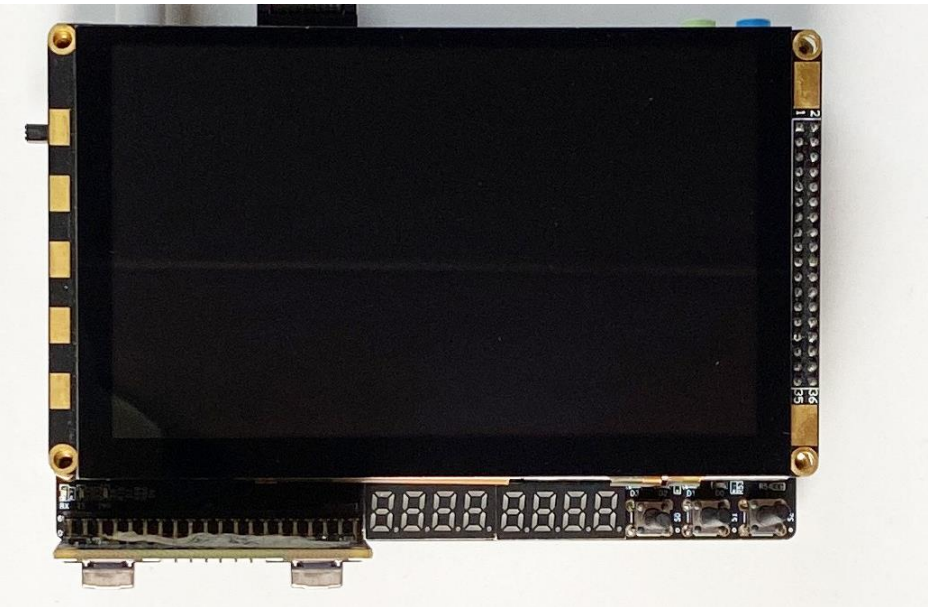


图 41-12 板级验证开发板连接图

连接好开发板后，下载生成的 Bit 文件。下载完成之后，TFT 屏上会显示摄像头采集到的图像数据，如下图所示。

图 41-13 板级验证实验效果

42 OV5640 摄像头采集 VGA 显示屏显示

工程源码	|----02_设计实例 |----ch42_ov5640_ddr3_vga
相关视频课程	
说明	

章节导读

通过上一章节的学习，我们完成了 OV5640 图像采集的 TFT 屏显示系统的设计，但是市面上的 TFT 屏大都只能支持一些低分辨率的图像显示。例如上一章节中我们使用的 5 寸屏，其分辨率为 800*480，因此最高就只能显示 800*480 分辨率的图像。受 TFT 屏本身分辨率的影响，这个系统通常只能被应用于一些低分辨率图像显示的场景。

而 FPGA 除了能够驱动 TFT 屏外，还能够驱动我们的电脑显示器，例如 VGA 接口显示器、HDMI 接口显示器等等。因而对于一些高分辨率的图像显示设计，开发者大都倾向于使用能够提供更高分辨率的电脑显示器，本章将以“OV5640 图像采集的 TFT 屏显示系统”为基础，带大家学习如何以较高的分辨率将 OV5640 的图像采集并显示在 VGA 显示器上。

42.1基于 OV5640 的 VGA 显示屏显示系统

常见的高分辨率有很多，例如 720p(1280*720)、1080p(1920*1080)、2k(2560*1440)、4k(4096*2160)等，考虑到 OV5640 输出图像的帧率会随着分辨率的增高而变小。例如，在输出 640*480 分辨率图像时可以达到 90 帧，而输出 2591*1944 分辨率图像时只能达到 15 帧。本章将通过摄像头产生 1280*720 的图像数据，基于“OV5640 图像采集的 TFT 屏显示系统”，实现数据在 VGA 显示屏上 1280*720 分辨率的显示设计。

首先是本次设计的系统框图，如图 42-1 所示。

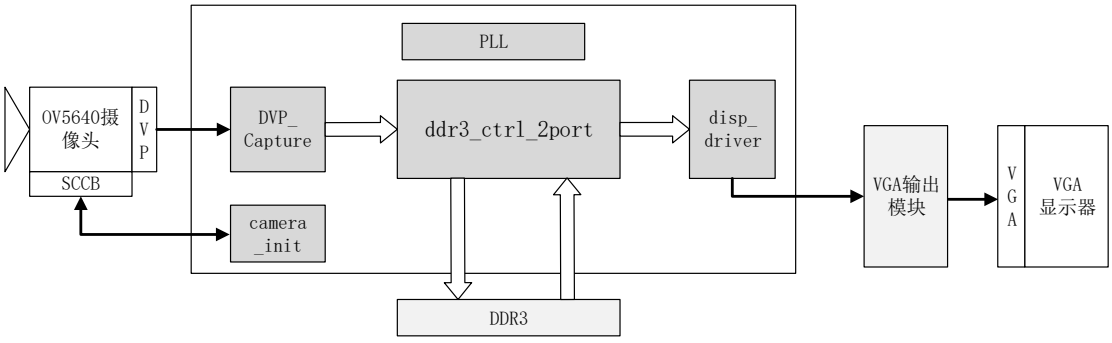


图 42-1 系统框图

本次设计基于“OV5640 的图像采集显示系统”（对应工程 ov5640_ddr3_tft），在该系统上进行了一定的添加与修改，具体如下：

1. 取消了 TFT 屏显示，改用 VGA 输出模块，将数据传输给 VGA 显示屏显示。
2. 对不部分模块参数进行了修改。

在讲解 RGB 接口 TFT 屏扫描方式时，我们曾讲过，RGB 接口的 TFT 屏其扫描方式与 VGA 一致，因此，TFT 驱动模块除了用来驱动 TFT 屏，也能用来驱动 VGA 显示屏。但是，值得注意的一点是，TFT 屏直接使用的数字信号，而 VGA 使用的是模拟信号，因此，在驱动模块输出图像时序和数据后，还需要使用专门的硬件将这些数字信号转变为模拟信号，也就是我们刚刚说的 VGA 输出模块。

为了方便理解设计，接下来我们按照数据流向对设计模块进行分析，一起完成本次系统设计。

42.2OV5640 初始化配置

首先是数据产生模块，也就是我们的 OV5640 摄像头。由于待显示的图像分辨率由 5 寸 TFT 屏的 800*480 变为了 VGA 的 1280*720，所以我们需要修改 OV5640 的初始化配置，使其输出相应分辨率的图像数据。OV5640 中控制控制输出图像分辨率大小的寄存器为 3808~380b，4 个寄存器功能如下：

表 42-1 输出大小窗口设置寄存器功能说明

寄存器地址	名称	默认值	功能描述
0x3808	TIMING DVPHO	0x00	bit[3:0]: 输出图像 X 方向尺寸大小的高 4 位
0x3809	TIMING DVPHO	0x10	bit[7:0]: 输出图像 X 方向尺寸大小的低 8 位
0x380a	TIMING DVPVO	0x00	bit[2:0]: 输出图像 Y 方向尺寸大小的高 3 位
0x380b	TIMING DVPVO	0x04	bit[7:0]: 输出图像 Y 方向尺寸大小的低 8 位

而在 OV5640 的配置表中，这几个寄存器的值通过全局传参进行设置：

```
18  module ov5640_init_table_rgb #(
19    parameter DATA_WIDTH      = 24,
20    parameter ADDR_WIDTH      = 8,
21    parameter IMAGE_WIDTH     = 16'd640,
22    parameter IMAGE_HEIGHT    = 16'd480,
```

```
23    parameter IMAGE_FLIP_EN   = 1'b0,
24    parameter IMAGE_MIRROR_EN = 1'b0
25  )
       // DVPHO  (<1280>500) (<640>280)IMAGE_WIDTH
227    rom[223] = {16'h3808, IMAGE_WIDTH[15:8]};
278    rom[224] = {16'h3809, IMAGE_WIDTH[ 7:0]}; // DVPHO
       // DVPVO  (<720>2d0)  (<480>1e0)IMAGE_HEIGHT
279    rom[225] = {16'h380a, IMAGE_HEIGHT[15:8]};
280    rom[226] = {16'h380b, IMAGE_HEIGHT[ 7:0]}; // DVPHO
```

因此我们需要在顶层的 100~101 行（这里的行号仅用于参考，方便用户定位代码所在位置，实际所在位置以用户代码中为准，后述文中所涉及的行号也同理），将 IMAGE_WIDTH 和 IMAGE_HEIGHT 这两个参数的值设置为 720P 对应的参数：

```
100    parameter IMAGE_WIDTH  = 1280;
101    parameter IMAGE_HEIGHT = 720;
```

在摄像头被正确初始化后，会按照配置的参数，将采集的图像数据按照 1280*720 的分辨率以 RGB565 格式输出，同时输出的还有 PCLK 和行场同步信号。数据会被输出给 DVP_Capture 模块，在 DVP_Capture 模块中数据被拼接为 16 位后通过 ddr3_ctrl_2port 模块写入 DDR3，并在需要时读出，随后送入到显示驱动模块。

42.3显示驱动模块配置

数据在被送到显示驱动模块后，显示驱动模块会根据用户设置的参数，产生显示所需的时序信号。为了方便用户修改，显示驱动模块的配置参数全部以条件编译的方式保存于 disp_parameter_cfg.v 文件中。涉及所需的显示分辨率为 1280*720，因此，只需要通过行注释的方式将文件中 48~83 行代码改为如下：

```
49  //使用 4.3 寸 480*272 分辨率显示屏
50  //`define HW_TFT43
51
52  //使用 5 寸 800*480 分辨率显示屏
53  //`define HW_TFT50
54
55  //使用 VGA 显示器，默认为 640*480 分辨率，24 位模式，其他分辨率或需 16 位
模式可在代码 63 行至 75 行进行重配置
56  `define HW_VGA
57
58  //=====================================
59  //以下宏定义选择用于根据显示设备进行位模式和分辨率 2 个参数的设置
60  //=====================================
```

```
61  `ifdef HW_TFT43  //使用 4.3 寸 480*272 分辨率显示屏
62  `define MODE_RGB565
63  `define Resolution_480x272 1 //时钟为 9MHz
64
65  `elsif HW_TFT50  //使用 5 寸 800*480 分辨率显示屏
66  `define MODE_RGB565
67  `define Resolution_800x480 1 //时钟为 33MHz
68
69  `elsif HW_VGA    //使用 VGA 显示器，默认为 640*480 分辨率，24 位模式
70  //====================================
71  //可选择其他分辨率和 16 位模式，需用户根据实际需求设置
72  //代码 70~71 行设置位模式
73  //代码 73~78 行设置分辨率
74  //====================================
75  `define MODE_RGB888
76
77  //  `define Resolution_640x480   1 //时钟为 25.175MHz
78  //`define Resolution_800x600   1 //时钟为 40MHz
79  //`define Resolution_1024x600  1 //时钟为 51MHz
80  //`define Resolution_1024x768  1 //时钟为 65MHz
81  `define Resolution_1280x720  1 //时钟为 74.25MHz
82  //`define Resolution_1920x1080 1 //时钟为 148.5MHz
83  `endif
```

可以看到这里通过条件编译将输出图像格式由 RGB565 设置为了 RGB888，将分辨率设置为了 1280*720。而根据条件编译出的结果，在 191~204 行中，可以看到此时输出图像的时序参数如下：

```
191 `elsif Resolution_1280x720
192  `define H_Total_Time    12'd1650
193  `define H_Right_Border  12'd0
194  `define H_Front_Porch   12'd110
195  `define H_Sync_Time     12'd40
196  `define H_Back_Porch    12'd220
197  `define H_Left_Border   12'd0
198
199  `define V_Total_Time    12'd750
200  `define V_Bottom_Border 12'd0
201  `define V_Front_Porch   12'd5
202  `define V_Sync_Time     12'd5
203  `define V_Back_Porch    12'd20
204  `define V_Top_Border    12'd0
```

原设计中，disp_driver 模块的输入输出图像数据都是 RGB565 格式。而在我们修改显示驱动配置后，disp_driver 模块输入输出 RGB888 格式数据，因此我们需要修改顶层中输入输出图像数据信号的位宽以及对 disp_driver 模块例化的端口连接。同时为了方便区分，我们还需要将顶层中的所有名为 TFT 的信号修改

为 VGA，代码如下：

```
25  output [23:0]   VGA_rgb       , //TFT 数据输出
26  output          VGA_hs        , //TFT 行同步信号
27  output          VGA_vs        , //TFT 场同步信号
28  output          VGA_clk       , //TFT 像素时钟
29  output          VGA_de        , //TFT 数据使能

209  wire [23:0] Data_in;
210  assign Data_in =
{rdfifo_dout[15:11],3'd0,rdfifo_dout[10:5],2'd0,rdfifo_dout[4:0],3'd0}
;
211  disp_driver disp_driver
212  (
213    .ClkDisp     (clk_disp       ),
214    .Rst_p       (g_rst_p        ),
215
216    .Data        (Data_in        ),
217    .DataReq     (rdfifo_rden    ),
218
219    .H_Addr      (               ),
220    .V_Addr      (               ),
221
222    .Disp_HS     (VGA_hs         ),
223    .Disp_VS     (VGA_vs         ),
224    .Disp_Red    (VGA_rgb[23:16] ),
225    .Disp_Green  (VGA_rgb[15:8]  ),
226    .Disp_Blue   (VGA_rgb[7:0]   ),
227    .Frame_Begin (frame_begin    ),
228    .Disp_DE     (VGA_de         ),
229    .Disp_PCLK   (VGA_clk        )
230  );
```

42.4产生 VGA 驱动时钟

修改完时序参数后的驱动模块并不能正常工作，因为此时该模块所使用的时钟仍是原来的 33MHz。根据前面我们得到的时序参数可以知道 1280*720 分辨率下扫描一行需要 1650 个时钟周期，一帧图像共有 750 行，进而我们能求出扫描一帧图像所需的时钟周期。而对于 VGA 显示器，其要求输入的图像信号刷新率不低于 50Hz，推荐采用 60Hz。也就是要求 1 秒钟内，显示屏上的图像更新 60 次。因此我们可以计算出满足该分辨率和帧率要求时，像素时钟频率为：

*1650*750*60=74.25MHz*

接下来使用设计中已有的 PLL 产生我们所需的 74.25MHz 时钟（由于

GOWIN 软件中 PLL 无法生成准确的 74.25MHz 的时钟，所以我们直接产生 75MHz 的时钟，不影响最终输出的结果），并取消掉其中多余的 33MHz 时钟。如图 42-2 所示。

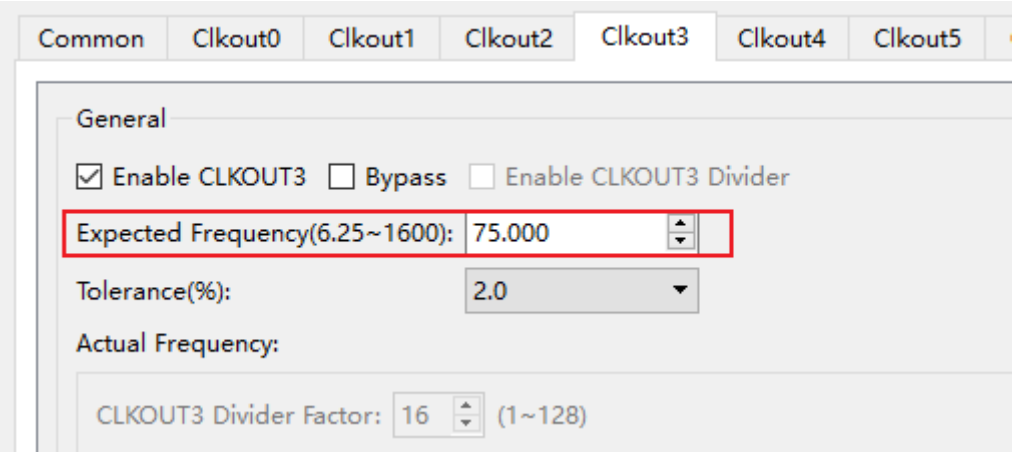


图 42-2 生成显示驱动时钟

配置完 PLL 后我们需要重新对其例化，随后定义用来连接 75MHz 时钟的信号 loc_clk75m，并将时钟连接到显示驱动模块的工作时钟接口和读 fifo 的写时钟接口，使用 assign 将 PLL 输出的 75Mhz 时钟赋值给 clk_disp，最后修改后的 PLL 端口例化完整代码如下：

```
wire loc_clk75m;
assign clk_disp = loc_clk75m;
Gowin_PLL Gowin_PLL(
    .lock(pll_lock), //output lock
    .clkout0(loc_clk50m), //output clkout0
    .clkout1(loc_clk24m), //output clkout1
    .clkout2(clk_200M), //output clkout2
    .clkout3(loc_clk75m), //output clkout3
    .clkin(clk50m), //input clkin
    .reset(~reset_n) //input reset
)
```

至此，显示驱动模块的配置完成，模块会根据设置好的参数，产生 1280*720@60Hz 分辨率图像显示所需的时序参数，将数据以及时序等信号输出给 VGA 输出模块。

42.5VGA 输出模块

前面说到，VGA 传输使用模拟信号，而 FPGA 输出的是纯数字信号，因此我们需要使用专用的硬件将 FPGA 输出的数字信号转换为符合 VGA 接口要求的模拟信号。这类转换器，最常用的芯片就是 ADI 公司的 ADV7123。

为了实现 VGA 显示驱动器的驱动，芯路恒公司提供了一个扩展模块，该模块使用兼容 ADI 公司的 GM7123 芯片。芯片内部包含 3 个 10 位的 DAC ，在设计时，模块仅使用了芯片内每个 DAC 的高 8 位，因此支持 24 位数字像素输入，输入后的数据经过 DAC 转换为模拟量，并支持高达 1920*1080 的 VGA 图像输出。GM7123 模块如下所示。

图 42-3 GM7123 VGA 输出模块

由于一个 24 位色的 VGA 电路至少需要占用 28 个 I/O，直接集成在开发板上会浪费较多的 I/O，所以对于小梅哥 FPGA 的各个 FPGA 开发板，都统一使用一个 36Pin 的通用显示扩展接口，再加上扩展的 VGA 输出模块来实现相应的功能。TFT 显示屏与 VGA 输出模块都要使用 36pin 拓展口，所以 TFT 显示屏和 VGA 显示器无法同时显示。

VGA 并不需要 TFT 的背光信号，因此，需要将顶层中 30 行和 230 行 TFT_pwm 信号相关的语句删除掉。

删除完多余信号，要想 VGA 输出模块能够正常工作，我们还需要对其引脚进行分配，本次设计 VGA 输出模块的引脚分配表如所示。

表 42-2 VGA 输出模块管脚约束表

Pin Name	Signial Name	Pin NO.		Pin Name	Signial Name	Pin NO.
VGA_HS	VGA_hs	T11		VGA_G5	VGA _rgb[13]	M18
VGA_VS	VGA_vs	R11		VGA_G4	VGA _rgb[12]	M16
VGA_CLOCK	VGA _clk	E16		VGA_G3	VGA _rgb[11]	N16
VGA_BLK	VGA _de	L14		VGA_G2	VGA _rgb[10]	N15
VGA_R7	VGA _rgb[23]	E18		VGA_G1	VGA _rgb[9]	N4
VGA_R6	VGA _rgb[22]	F15		VGA_G0	VGA _rgb[8]	M14
VGA_R5	VGA _rgb[21]	H12		VGA_B7	VGA _rgb[7]	T18
VGA_R4	VGA _rgb[20]	G13		VGA_B6	VGA _rgb[6]	T17

VGA_R3	VGA _rgb[19]	H15		VGA_B5	VGA _rgb[5]	U18
VGA_R2	VGA _rgb[18]	J16		VGA_B4	VGA _rgb[4]	U17
VGA_R1	VGA _rgb[17]	M13		VGA_B3	VGA _rgb[3]	V16
VGA_R0	VGA _rgb[16]	F16		VGA_B2	VGA _rgb[2]	U16
VGA_G7	VGA _rgb[15]	L16		VGA_B1	VGA _rgb[1]	V15
VGA_G6	VGA _rgb[14]	L15		VGA_B0	VGA _rgb[0]	U15

分配完成后，VGA 输出模块就能正常进行工作了。数据在经由 VGA 线缆到达显示器的 VGA 接口后，对于模拟的 CRT 显示器，这些信号会直接被放大后用于驱动电子枪发射电子进而产生相应图像，而对于液晶显示器，则需要显示器使用专门的模拟数字转换芯片将模拟信号再转换为数字信号后，去驱动 RGB 接口的液晶显示屏显示图像。

至此整个设计梳理完成，接下来就可以生成比特流并将其烧录到开发板中进行验证了。

42.6板级验证

本节介绍在高云开发板上使用 OV5640 模块和 VGA 输出模块进行“OV5640 摄像头采集 VGA 显示屏显示”实验的验证。

42.6.1所需硬件

1. 高云开发板 x1
2. OV5640 摄像头 x1
3. 高云下载器 x1
4. VGA 输出模块 GM7123 x1
5. VGA 显示器
6. 电源线 x1
7. VGA 线缆 x1

42.6.2硬件连接

将 OV5640、VGA 输出模块、下载器、电源线依次连接开发板，将 VGA 线缆一端连接到 VGA 输出模块的 VGA 接口上，另一端连接 VGA 显示器。连接完成后将开发板电源开关拨到对应侧为开发板上电，硬件连接图如下图 42-4 所示。

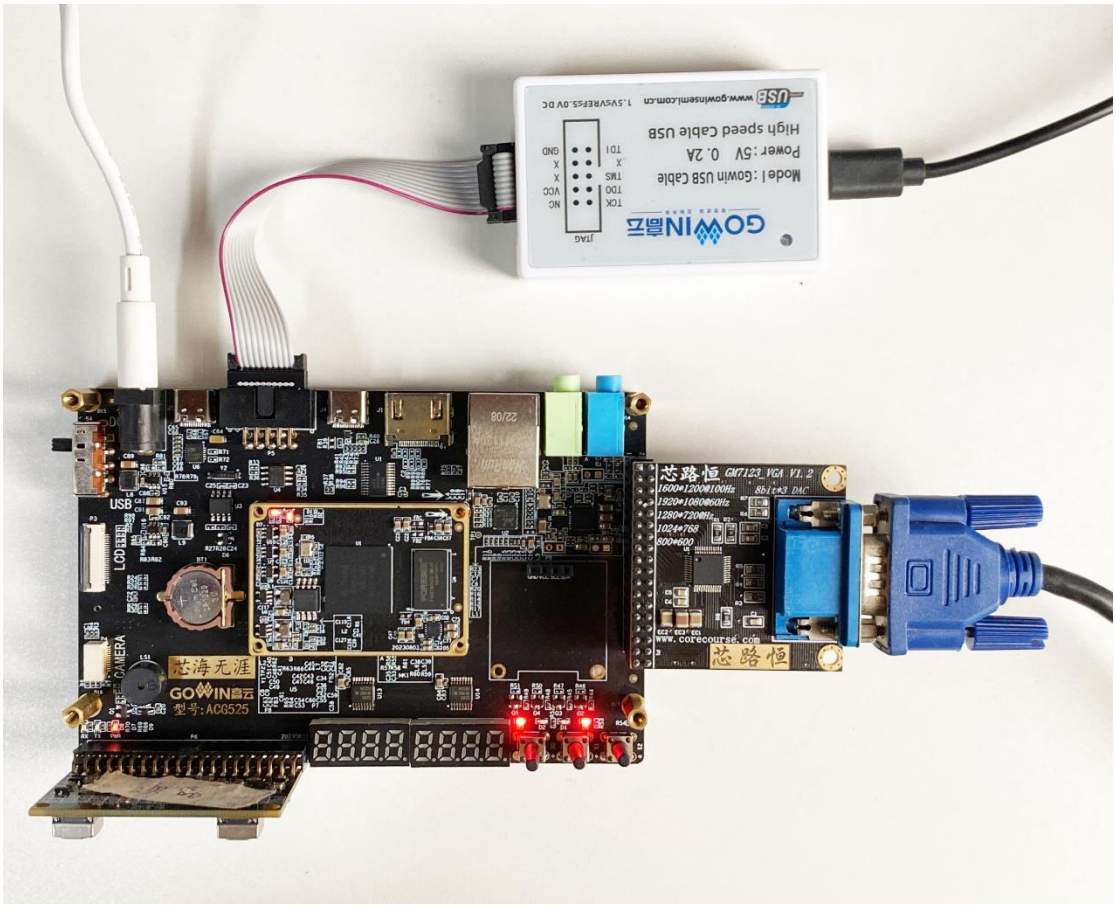


图 42-4 硬件连接图

在连接 VGA 接口时，需要注意：VGA 线缆必须一端连接 GM7123 的 VGA 接口，另一端连接显示器或电视机的 VGA 接口。不可以将线缆一端连接 GM7123 的 VGA 接口，另一端连接到笔记本或者主机的 VGA 接口上。因为笔记本和主机的 VGA 接口为输出接口，而 GM7123 的 VGA 接口也为输出接口，这些输出接口互相连接无法实现图像的显示。而显示器和电视机上的 VGA 接口为输入接口，GM7123 的 VGA 输出接口只有与这些输入接口相连，才能实现图像的显示。

最后将生成好的文件烧录到开发板中。接着是对显示效果的验证，分别图 42-5 所示：

图 42-5 VGA 显示器显示效果

从上述图中可以看到图像显示正常，没有出现任何撕裂重叠现象，颜色层次清晰，说明设计成功。

42.7总结

本章带大家实现了 OV5640 摄像头采集 VGA 显示屏显示系统的设计与验证，设计基于已有工程实现，虽然 TFT 屏的驱动原理与 VGA 驱动原理一致，但是使用时需要注意二者的接口位宽。关于 VGA 扫描原理大家可以参考“VGA 显示驱动设计与验证”小节。

43 OV5640 摄像头采集 HDMI 显示屏显示

工程源码	|----02_设计实例 |----ch43_ov5640_ddr3_hdmi
相关视频课程	
说明	

章节导读

通过前面章节，我们已经完成了高云开发板上的 OV5640 采集系统的 TFT 显示以及 VGA 显示。除了这两种显示方式之外，开发板还支持 HDMI 显示，板上自带有两个完全一样的 HDMI 接口，本章将带大家通过这两个接口完成 OV5640 图像采集的 HDMI 显示设计。

43.1VGA 与 HDMI 显示对比

虽然使用 VGA 显示器已经能够实现较高分辨率的图像显示，但是 VGA 显示有个致命缺点，那就是在显示高分辨率的图像时，可能会出现失真、图像出现虚影的现象。而 HDMI 显示同样能够支持较高分辨率，甚至所能支持的最大分辨率要高于 VGA，但是 HDMI 在进行高分辨率显示时，极少甚至不会出现失真、虚影等现象。而这一切则是因为这两种显示设备接口在传输数据时，使用的是不同类型数据。如下图 43-1 为现今常见的 VGA 显示器和 HDMI 显示器与显卡传输图像数据的数据流变换示意图。

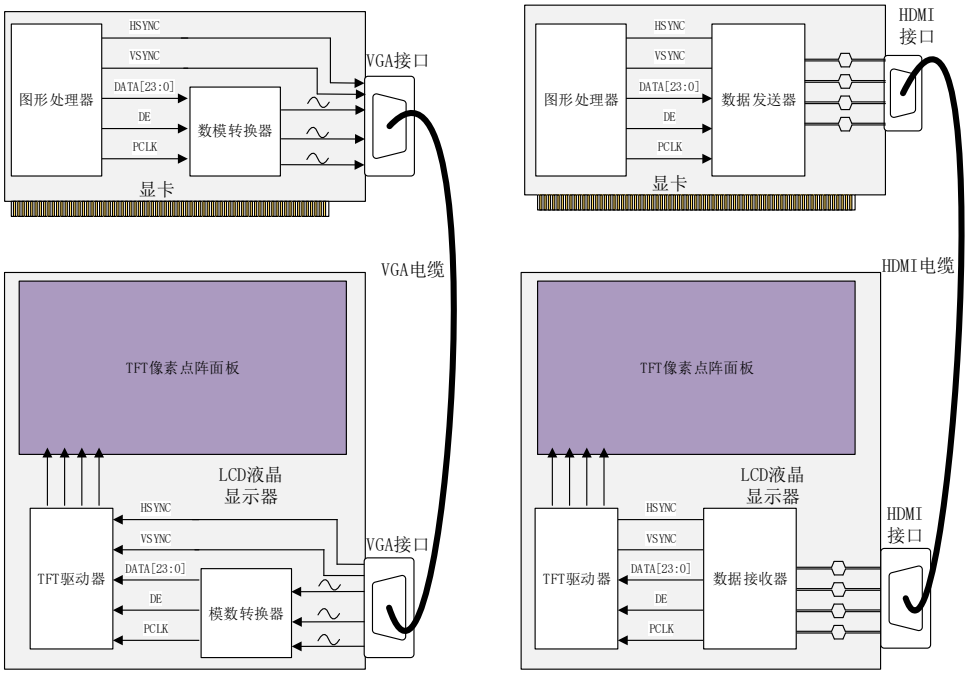


图 43-1 VGA 和 HDMI 接口与显卡传输图像数据流变换图

现今常见的无论是 VGA 接口显示器还是 HDMI 接口显示器，大多使用的液晶屏，这类屏幕采用的是液晶材质，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。这些液晶会根据加载其两端的电压大小而改变透光性，进而实现被动的颜色显示。

对于一个液晶屏而言，其所能显示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，所以液晶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。

而 VGA 接口传输的是模拟信号，因此显卡输出的数字图像信号必须转换成模拟信号后才能通过 VGA 接口输出到 VGA 显示器。而图像信号在送到 VGA 显示器的 VGA 接口后还需要再经过模数转换，转换为数字信号后才能驱动液晶显示。

数据在模数转换和数模转换的过程容易发生变化，从而导致显示的图像存在一些细微的差别。如果在转化时数据变化速度过快，就可能造成拖影现象。并且由于 VGA 线缆传输的是模拟信号，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。因此在较高分辨率的图像显示时，VGA 模拟信号传输的这些缺点就越发的明显了。

而 HDMI 接口，使用的是数字信号，显卡输出的数字图像信号全程不需要任何转变就能直接用于 HDMI 显示器的显示，减少了信号转换过程，能够有效消除拖影现象。同时，HDMI 显示使用的是差分传输，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，因此即使是进行高分辨率的图像显示，HDMI 显示也能保证最终显示的图像质量，相较于 VGA 显示更具优势。

而本章也将基于“OV5640 摄像头采集 VGA 显示屏显示”实验，通过实现 1280*720@60Hz 图像的 HDMI 显示，带领大家学习如何将 OV5640 采集到的图像数据通过 HDMI 接口显示到 HDMI 显示器上。

43.2基于 OV5640 的 HDMI 显示屏显示系统

下图 43-2 为本次设计的系统框图。

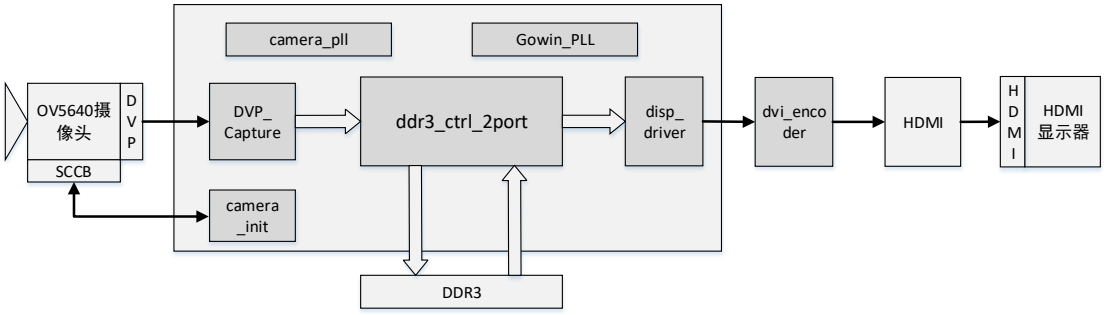


图 43-2 系统框图

本次设计基于“OV5640 摄像头采集 VGA 显示屏显示”实验，对于设计来说，要想实现 HDMI 显示，只需要对 disp_driver 模块输出的 1280*720@30Hz 图像数字信号以及对应行场同步信号进行编码，转换为符合 HDMI 接口的编码格式即可。因此在设计中添加了以下模块：

1. 新增加一个 PLL IP 核，用于产生 dvi 编码模块工作所需的时钟。
2. dvi 编码模块（dvi_encoder），用于将 disp_driver 模块输出的图像数据与时序，转换为符合 HDMI 接口发送时序的编码格式。

高云开发板自带 HDMI 接口，对于 HDMI 显示来说，从 disp_driver 模块输出的图像数据以及时序信号需要经由 dvi_encoder 核进行 TMDS 编码，并转换为串行信号后才能通过 HDMI 接口输出。输出的信号在通过 HDMI 线缆传输给 HDMI 显示器的 HDMI 接口后，显示器内部的 TMDS 解码模块会对信号解码，然后进行串并转换，将信号转变为并行信号后依据时序信号，将图像显示出来，实现 1280*720@60Hz 的 HDMI 显示。

43.3产生 DVI 驱动时钟

对于 dvi_encoder 模块来说，所需的时钟有两个，一个是进行 TMDS 编码的时钟，一个是编码后对信号进行并串转换的时钟。

dvi_encoder 模块会对接收的 RGB888 数据和行场同步信号进行 TMDS 编码，这些信号会被分成三组，每组由两位控制信号和对应的 8 位颜色分量组成（其中两组的控制信号为空）也就是每组一共 10 位数据。在设计中这些信号由 disp_driver 提供，因此 TMDS 编码时使用的也是 disp_driver 输出的工作时钟，即 74.25MHz。

而在进行并串转换时，dvi_encoder 模块会在每个时钟的上升沿和下降沿对每组信号进行并行到串行的转换，为了能够将编码完成的数据立马转换，就需

要一个 5 倍于 TMDS 编码时的工作时钟，也就是 74.25x5=371.25MHz 的时钟。

实际使用的时候，无法准确生成 371.25MHz 的时钟，在实验中我们将通过 PLL 生成 375MHz 的时钟，然后通过时钟分频源码，进行 5 分频，生成 75MHz 的时钟，给到 disp_driver 模块使用，最终 PLL 模块的例化代码和时钟分频代码如下所示：

```
wire          loc_clk24m;
wire          clk_vga;//75m
wire          clk_vgax5;//375m
wire          clk_200M;
wire          pll_lock;
wire          loc_clk50m;

camera_pll camera_pll(
    .clkout0(clk_vgax5), //output clkout0
    .clkout1(loc_clk24m), //output clkout1
    .clkin(loc_clk50m) //input clkin
);

Gowin_PLL Gowin_PLL(
    .lock(pll_lock), //output lock
    .clkout0(clk_200M), //output clkout0
    .clkout1(loc_clk50m), //output clkout1
    .clkin(clk50M), //input clkin
    .reset(~reset_n) //input reset
);

CLKDIV u_clkdiv
(.RESETN(pll_lock)
,.HCLKIN(clk_vgax5) //clk  x5
,.CLKOUT(clk_vga)    //clk  x1
,.CALIB (1'b1)
);
defparam u_clkdiv.DIV_MODE="5";
```

例化完成后，我们便完成了 dvi_encoder 模块时钟的生成，接下来就是 dvi_encoder 模块的设计和例化工作了。

43.4DVI 编码模块

DVI 编码模块负责对信号进行 TMDS 编码以及并行到串行的转换，对于该模块的详细描述，可以参考“基于 FPGA 的 HDMI/DVI 显示”章节。这里我们直接使用设计好的模块即可，使用时只需将该模块以及相关子模块（可以直接从例程中提取）添加进工程中，在顶层中添加相关输出接口并例化 DVI 编码模

块。

顶层中对 dvi_encoder 模块接口的添加以及模块例化代码如下：

```
output          tmds_clk_p ,
output          tmds_clk_n ,
output  [2:0]   tmds_data_p,    //rgb
output  [2:0]   tmds_data_n,     //rgb

dvi_encoder u_dvi_encoder(
    .pixelclk(clk_vga),// system clock
    .pixelclk5x(clk_vgax5),// system clock x5
    .rst_n(reset_n),// reset
    .blue_din(video_b),// Blue data in
    .green_din(video_g),// Green data in
    .red_din(video_r),// Red data in
    .hsync(video_hs),// hsync data
    .vsync(video_vs),// vsync data
    .de(video_de),// data enable
    .tmds_clk_p(tmds_clk_p),
    .tmds_clk_n(tmds_clk_n),
    .tmds_data_p(tmds_data_p),//rgb
    .tmds_data_n(tmds_data_n) //rgb
   );
```

至此，我们便完成了所有相关模块的设计，由于涉及中我们在顶层中添加了 HDMI 相关接口，因此还需要为这些引脚进行分配与电平约束。

43.5引脚分配

本次涉及 HDMI 相关引脚分配表如下表 43-1 所示。

表 43-1 HDMI 接口引脚分配表

Pin Name	Signial Name	Pin NO.
HDMI_CLK_P	tmds_clk_p	M10
HDMI_CLK_N	tmds_clk_n	N9
HDMI_D0_P	tmds_data_p[0]	R7
HDMI_D0_N	tmds_data_n[0]	T7
HDMI_D1_P	tmds_data_p[1]	T6
HDMI_D1_N	tmds_data_n[1]	V6
HDMI_D2_P	tmds_data_p[2]	R5
HDMI_D2_N	tmds_data_n[2]	T5

根据上述表中的内容，对 HDMI 进行引脚分配和时钟约束之后，HDMI 接口便能够被正常驱动并工作，我们也完成了对整个系统的设计工作。从 disp_driver 输出的信号会送到 dvi_encoder 模块中，在经过了编码和串并转换后，被送到 HDMI 接口输出。为了验证整个系统设计能否正常工作，接下来就可以

生成系统设计的比特流并将其烧录到开发板中进行验证。

43.6板级验证

本节介绍在高云开发板上使用 OV5640 摄像头模块和 HDMI 显示器进行“OV5640 摄像头采集 HDMI 显示屏显示”实验的验证。

43.6.1所需硬件

1. 高云开发板 x1
2. OV5640 摄像头 x1
3. 高云下载器 x1
4. HDMI 显示器 x1
5. 电源线 x1
6. HDMI 线缆

43.6.2硬件连接

将 OV5640、下载器、电源线依次连接开发板，将 HDMI 线缆一端连接开发板上的 HDMI 接口，另一端连接到 HDMI 显示器，如下图 43-3 所示。

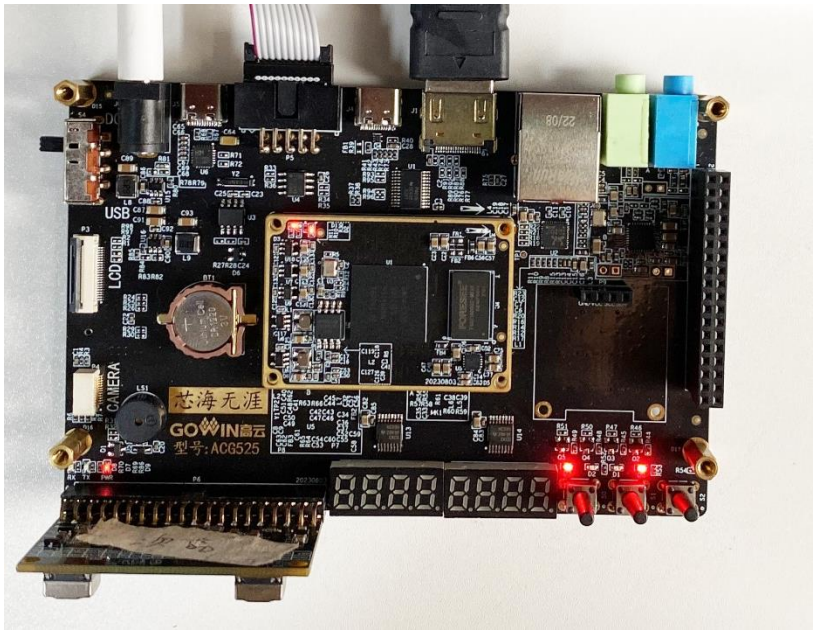


图 43-3 硬件连接

在连接 HDMI 接口时，需要注意：HDMI 线缆必须一端连接 FPGA 的 HDMI

接口，另一端连接显示器或电视机的 HDMI 接口。不可以将线缆一端连接开发板 HDMI 接口，另一端连接到笔记本或者主机的 HDMI 接口上。因为笔记本和主机的 HDMI 接口为输出接口，而开发板的 HDMI 接口大多数在硬件上只支持输出，这些输出接口互相连接无法实现图像的显示。而显示器和电视机上的 HDMI 接口为输入接口，FPGA 的 HDMI 输出接口只有与这些输入接口相连，才能实现图像的显示。

连接完毕后，给开发板上电，将生成好的比特流文件烧录到开发板中，系统开始工作，显示效果如下图 43-4 所示。

图 43-4 HDMI 显示效果

从上图中可以看到图像显示正常，没有出现任何撕裂重叠现象，颜色层次清晰，说明设计成功。

43.7总结

本章带大家实现了 OV5640 摄像头采集 HDMI 显示屏显示系统的设计与验证，设计整体上只是在“OV5640 摄像头采集 VGA 显示屏显示”系统的基础上对信号进行了 TMDS 编码和并串转换，从而以符合 HDMI 编码的格式输出。至于 TMDS 编码以及并串转换的具体实现，大家可以参考“基于 FPGA 的 HDMI/DVI 显示”章节。

44 OV5640 实时 RAW2RGB 采集显示系统

工程源码	|----02_设计实例 |----ch44_ov5640_ddr3_raw2rgb_tft
相关视频课程	
说明	

章节导读

RAW 格式数据作为数字图像传感器输出的原始数据，由于没有经过各种转换处理，图像数据失真较小，十分适合进行图像处理，在许多场景下都有广泛的用途。本节以 RAW 格式数据作为实验对象，以 RAW2RGB 模块为核心，实现了 RAW 格式数据到 RGB888 格式数据的转换。

本节首先对 RAW 数据及其数据格式进行讲解，随后讲述了 RAW2RGB 模块的工作原理，包括如何在节省硬件资源的情况下获得相邻的四个 RAW 数据像素，如何确定四个像素点的排列顺序，以及如何根据排列顺序计算出相应的 RGB 数据值，并在图像采集显示系统中应用此模块，完成摄像头 RAW 数据的采集转换和显示。

44.1原始数据 RAW

在基于 FPGA 的图像采集显示系统中，我们逃不开的一点就是对图像数据的采集与处理。图像数据在被摄像头的感光单元捕捉后会形成 Bayer 格式的原始图像数据，也就是图像的电子底片。而我们平常所接触到的数据格式，如 RGB565、RGB888、YUV422 等，都是在原始 RAW 数据基础上经过摄像头内置的 ISP 等模块加工处理后得到的。由于数据经过了加工处理，往往会有部分数据会被舍弃，因而很难获得最佳画质。

作为摄头捕捉到的最原始的数据，RAW 格式数据与其说是图像文件，不如说是一个数据包。这个数据包由于未经过相机内的影像生成器转换，所以除了曝光量是提前设置好的外，其余的如对比度，色温，饱和度等都可以按自己需求在后期调整，因为是直接对原数据进行处理，所以对图像的损失会非常小。

同时，RAW 格式数据对黑白之间影调的范围也非常的大。JPEG 格式的文件都是 8 位的数据文件，只包含 256 级影调变化。通常 250 级左右的影调便能表达出柔和、自然的画面，所以一般 250 级左右的影调图像文件便能满足大部分的需求。考虑到摄像头输出接口位宽为 8，本次实验次采用的是 8 位位宽的

RAW 格式数据。

RAW 格式本身的优点非常多，而其适合图像处理的这一点也非常符合本次实验的需求，通过将 RAW 数据与其相邻的三个数据进行插值计算，我们可以转换出该点对应的 RGB888 像素值。每个点都会与其相邻的像素点进行四次运算，最终每个像素点都能实现从 RAW 到 RGB888 的转变，同时经由相应的模块输送到显示设备上进行显示。

44.2Bayer 色彩矩阵空间

44.2.1色彩感光原理

要想实现对 RAW 格式数据到 RGB888 格式数据的转换，我们首先要了解 RAW 格式数据的结构。这里以 OV5640 这款 500W 像素的 CMOS 摄像头为例，介绍其成像原理。

OV5640 是一个典型的 CMOS 图像传感器，作为一个图像传感器，其主要作用就是将现实中的各种光线转换为数字系统能够识别的数字信号。光线中三元色各个颜色的强度本身是模拟信号，所以图像传感器的最基本的原理就是进行模数转换，将光线这个模拟量转换为数字信号。

仅仅有模数转换功能还不够，我们还得先搞清楚光线是一种怎样的模拟量。在初中物理中已经有过详细的介绍：自然界中的光，实际上是三种基本单色光的组合，这三种基本单色光为红（RED）、绿（GREEN）、蓝（BLUE），我们称之为三原色。通过将这三种基本颜色按照不同的比例混合，就可以得到其他的任意颜色。例如纯黄色是由红色和绿色按照一比一的比例混合得到的，蓝色量为 0。下图 44-1 为三种颜色混合得到几种常见颜色的示意图。

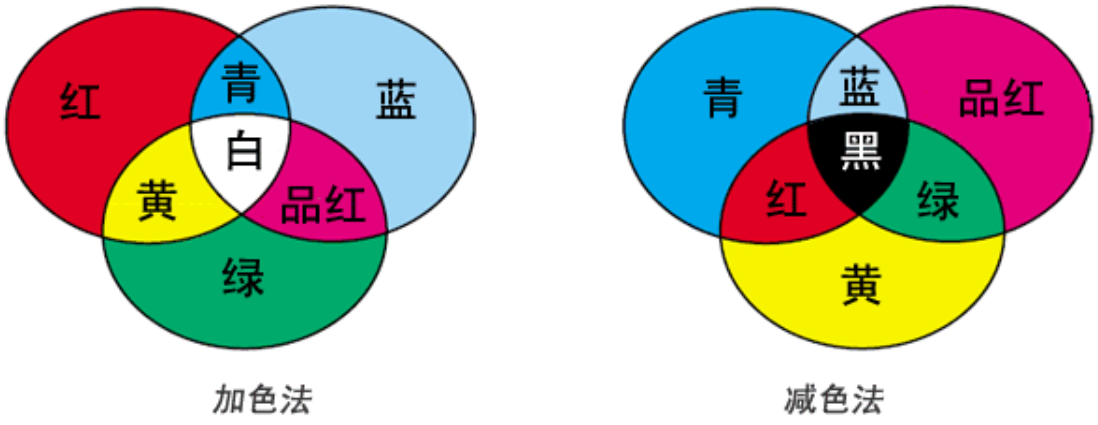


图 44-1 几种常见颜色示意图

既然已经知道，每一束光线都可以理解为三原色按照不同比例混合得到的效果，那么我们只要想办法知道该束光线的三原色的比例，然后用颜色加比例的表示方法，就能唯一确定这束光线的最终颜色了。所以图像传感器里面所谓的模数转换，实质就是对一束光线的三原色的强度进行转换，将三原色中每一种颜色的强度转化为数字信号。再用颜色加数字的方式来表示该束光线的真实颜色。

这里，假如我们对三原色中的每一种基本颜色的强度都分为 256 级，那么每一种基本颜色的强度就可以用一个 8 位的数字来表示，0 表示该颜色强度最弱，或者说无该颜色分量，255 表示该颜色强度最强。这样一来，就可以用一个 24 位二进制的数字来唯一表示该光线的颜色了。上图中几种颜色的对应三原色的数值如表 44-1 所示：

表 44-1 几种常见颜色的三原色表示

	红	绿	蓝	黄	品红	青	白	黑
红	255	0	0	255	0	255	255	0
绿	0	255	0	255	255	0	255	0
蓝	0	0	255	0	255	255	255	0

这种表示颜色的方法就是最常见的 RGB888 格式。所谓 RGB888 就是使用 3 个 8 位的数据表示一种颜色，其中高八位表示红色分量，中八位表示绿色分量，低八位表示蓝色分量，上述 8 种颜色用 RGB888 格式表示就如下表 44-2 所示：

表 44-2 几种常见颜色的 RGB888 表示

	红	绿	蓝	黄	品红	青	白	黑
RGB888	0xff0000	0x00ff00	0x0000ff	0xffff00	0x00ffff	0xff00ff	0xffffff	0x000000

通过上面的分析我们就可以知道，只要模数转换器能够对每一束光的三原色的强度进行转换得到数字信号，就能唯一表示该颜色了。但是接下来，另一个问题又出现了。如何才能对一束自然光中的三原色的强度分别进行模数转换呢？这就涉及到对一束光的三原色分离。

所谓对光的三原色分离就是通过某种手段，将该束光线中的三种颜色分别独立提取出来，当三种颜色都独立的提取到之后，就能使用模数转换器对该颜色的强度进行转换了。三原色分离的原理其实非常简单，就是使用单色滤光片，如图 44-2 所示：

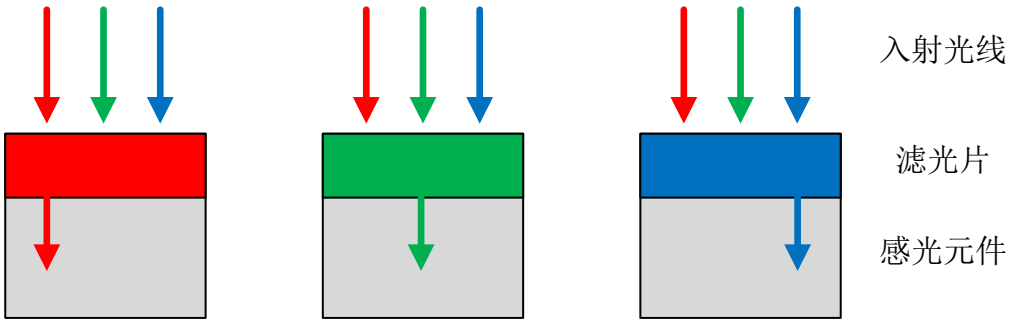


图 44-2 单色滤光片

通过加入滤光片，就能让对应颜色的光线通过滤光片到达感光元件，通过这种方式，只需要三个不同颜色的滤光片，就能对一束入射的复合光线进行分离，得到三原色，然后使用模数转换器对感光元件感应到的单色光的光照强度进行转换，就能得到该单色光的强度数字值了。

通过上述分析我们也发现，一个感光元件只能对一种颜色的光进行感应，如果要对一束光线中的三种颜色分别感应，就需要三个感光元件。如果对应到 CMOS 图像传感器里面的概念来说，这每一个滤光片加感光元件都是一个像素。注意这个概念，每一个滤光片加感光元件都是一个像素，那么从反面来说就是，每个像素都只能感应一种颜色的光线。

既然每个像素只能感应一种颜色，那么新一个问题又来了——如何才能得到某束光线的三种颜色分量的值呢？答案就是使用至少 3 个像素，每个像素感应该束光的一种颜色，然后三个像素就能把该束光的三种颜色分量全提取出来了。

但是，使用 3 个像素来提取一束光的三种颜色分量，虽然理论上可行，实际上却存在 3 个像素间摆放位置的物理限制。首先是如何让该束光线能够均匀的照射到感应三种颜色分量光线的像素上的问题，其次还要知道，一个图像传感器，并不是只感应一束光，是要感应成千上万束细小的光束，每一束光线都需要有对应的像素组来提取其三种颜色分量。因此，像素和像素之间的物理摆放问题也是需要认真考虑的。同时，还得兼顾考虑这种摆放模式下图像传感器的可生产性问题。

正是为了解决这些问题，诞生了著名的拜尔（以下简称 Bayer）矩阵。

44.2.1.1Bayer 矩阵定义

感光像素矩阵中，每个奇数行间隔放置绿色和红色感应像素，每个偶数行

间隔放置绿色和蓝色感应像素，每个奇数列间隔放置绿色和蓝色感应像素，每个偶数列间隔放置红色和绿色感应像素，其输出数据格式如图 44-3 所示：

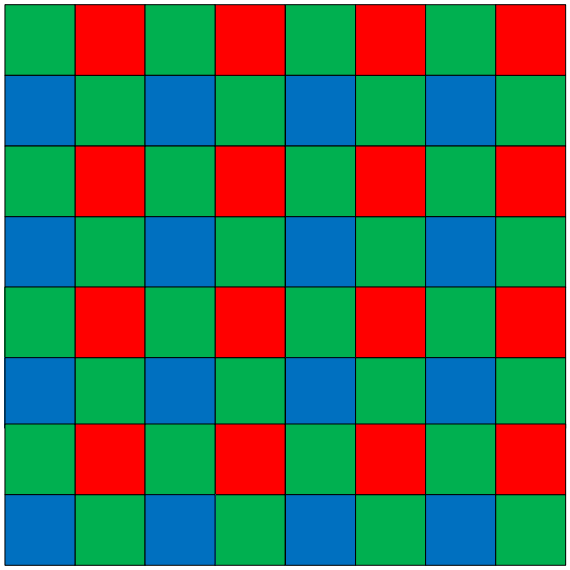


图 44-3 Bayer 矩阵

根据这种分布规律，在 Bayer 矩阵中，以相邻的四个像素作为一组，在该组中，有两个感应绿色分量的像素呈对角分布，另外两个像素则分别对应感应红色分量和感应蓝色分量。任取一个像素，其与相邻的 3 个像素组成的矩阵中，总符合该规律。不同的只是三种颜色的像素所在的位置的差异。图 44-4 为图像传感器手册中给出的像素物理分布图。

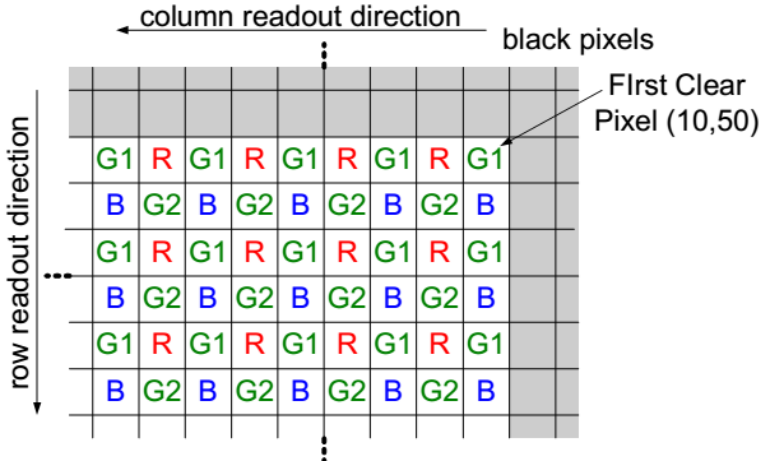


图 44-4 像素物理分布图

通过对上述 8*8 的矩阵进行分析发现，在整个矩阵中，像素的位置关系有且只有所示四种情况：

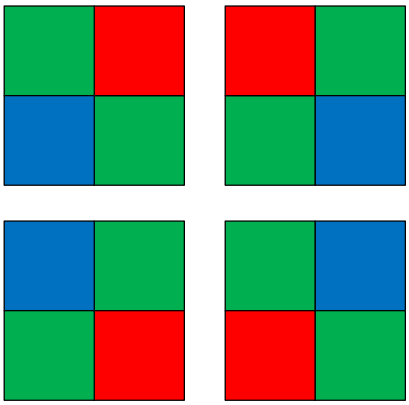


图 44-5 像素的四种位置关系

所以，在实际颜色提取时，需要通过数据转换，将相邻四个像素的数据通过插值算法合并为一个 RGB 像素颜色，此种转换算法名为 RAW2RGB，这里取左上角四个像素点的数据为例，具体颜色转换算法如图 44-6 所示：

图 44-6 相邻四种颜色分量所代表的含义

保留相邻四个像素中的红色和蓝色分量，而对两个绿色分量求平均，得到新的绿色分量，此三种颜色分量组成一个新的 RGB 格式的像素，新的像素色彩组成如图 44-7 所示：

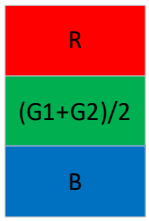


图 44-7 像素色彩计算

按照这样的思路，整个图像传感器中的每一个像素都可以以不同的角色参与 4 次运算，并最终得到 4 个 RGB 颜色值，所以理论来说，还是可以认为一个图像传感器有多少个物理像素，就能得到多少个 RGB 格式的像素值，虽然每个物理像素都只能感应一种颜色。但是经过插值运算后，其能输出 RGB 的数据格式的像素数量还是等于其物理像素个数的。

有了 Bayer 像素矩阵后，只需要使用一个模数转换器，依次对每一个像素的感光元件感应到的模拟量进行转换并输出，就能得到一幅完整图像的原始像素信息了。

44.3RAW 转 RGB 算法的 FPGA 实现

在有了一幅完整图像的 RAW 数据后我们便能开始着手对其进行插值运算了，但是直接将整幅图像存储下来再进行运算将会消耗大量的存储资源。观察 Bayer 矩阵我们可以发现对任意一个像素点进行差值运算时，只要知道其右边点的数据及下一行与这两个点相邻的数据，也就是只要知道下一个数据以及下一行该点正下方和右下方的数据即可。至于如何获取这些数据，就涉及到我们接下来要讲的 2x2 插值矩阵了。

44.3.12x2 插值矩阵的获取

在基于 FPGA 的图像处理中，我们一般通过使用 RAM-based Shift Register IP 核的方式来实现插值矩阵的获取。该 IP 核为移位寄存器，是一种存储器模型，用户可以设置移位寄存器的位宽和深度来控制对数据的延迟。

如果让移位寄存器的位宽与 RAW 格式数据一致，深度等于一行图像的像素个数，那么当一行图像的像素点依次移入到移位寄存器中后，接下来再输入进来的两个数据必然为下一行图像的前两位数据。取移位寄存器输出端（最先移入移位寄存器的数据会最先出现再输出端）的连续两个数据，与最新从图像传感器输出的连续的两个像素的数据，就能刚好组合成一个 2x2 的插值运算矩阵，以图 44-8 为例。

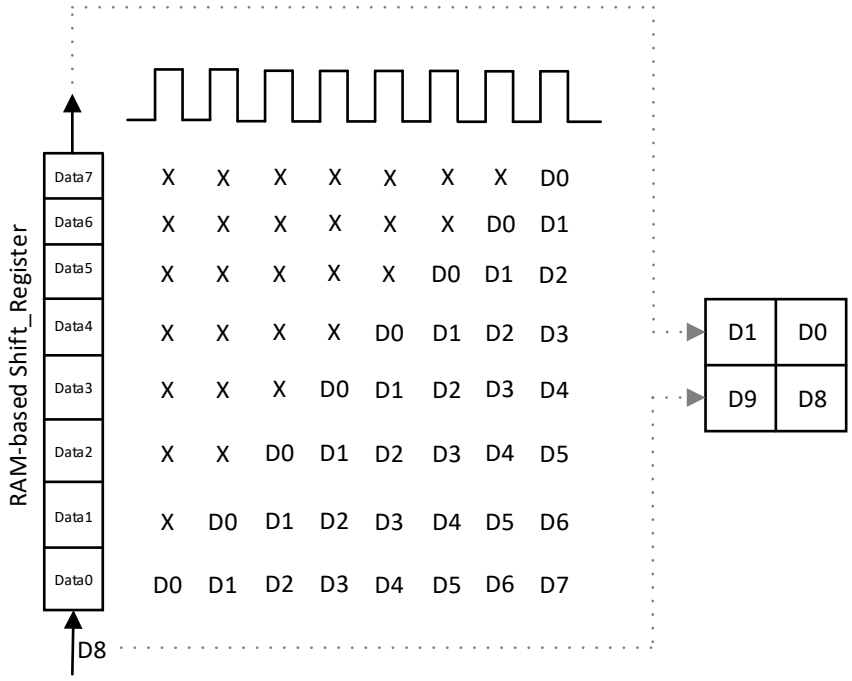


图 44-8 RAM-based shift 寄存器与两级缓存

假设我们一行只有 8 个数据，第一行第一个数据为 D0，第二个数据为 D1，依次累加，那么第二行第一个数据，第二个数据分别为 D8、D9。D0、D1 为第一行输入的前两个数，D8、D9 刚好为第二行输入的前两个数，知道了这四个值，我们就能按照其排列顺序计算出 D8 的 RGB 像素值。注意图中的 2x2 矩阵式寄存器中的排列关系，转换到 2x2 插值运算矩阵中 D0 与 D1，D8 与 D9 的顺序是相反的。

44.3.1.1RAM based Shift Register

上文提到的 RAM-based Shift Register 为移位寄存器，是一种存储器模型，Gowin 中提供了实现该存储器的 IP 核，在 IP Core Generator 中搜索其名字便能找到该 IP。

改 IP 的配置界面如下图 44-9 所示。

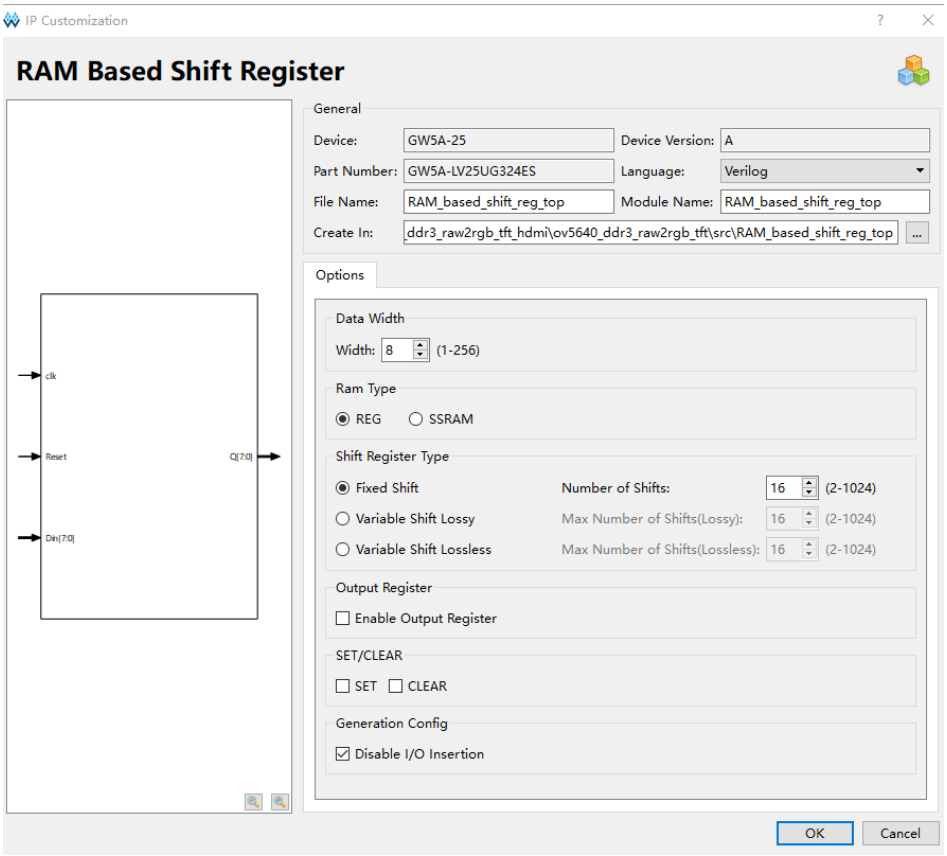


图 44-9 RAM-based Shift Register 配置界面

对上述配置选项进行简要说明：

1. Data Width：配置数据位宽（1~256），本次实验和 RGB888 单个颜色的数据位宽一致，为 8。

2. Ram Type：有两个选项可供选择，一个 REG，一个是 SSRAM。

3. Shift Register Type：移位寄存器选项，有三个选项可供选择：第一个是 Fixed shift：固定移位，我们可以自己设定需要移位的寄存器的个数，比如本次实验，我们就需要使用该选项，固定移位一行的数据寄存器个数，也就是 800，这里可以选择的范围在 2~1024；第二个是：Variable shift lossy：有损的可变模式，在该模式下，数据从进入寄存器到从寄存器中输出所经历的延迟是可变的，会新增 ARRD 端口，由该端口输入的值决定；第三个是：Variable shift lossless：无损的可变模式，与前一个模式类似，只不过是无损的。

4. Output Register：指定寄存器用于输出数据，这将使输出产生额外的时钟周期延迟。

5. SET/CLEAR：配置 SET 时，生成 SSET 输入端口，SSET 有效时置 1，且 SSET 优先级高于 SCLR；配置 CLEAR 时，生成 SCLR 输入端口，SCLR 有效时置 0，且 SCLR 优先级低于 SSET，本次实验配置 CLEAR。

本次实验关于 RAM Based Shift Register 的最终配置界面如下图 44-10 所示。

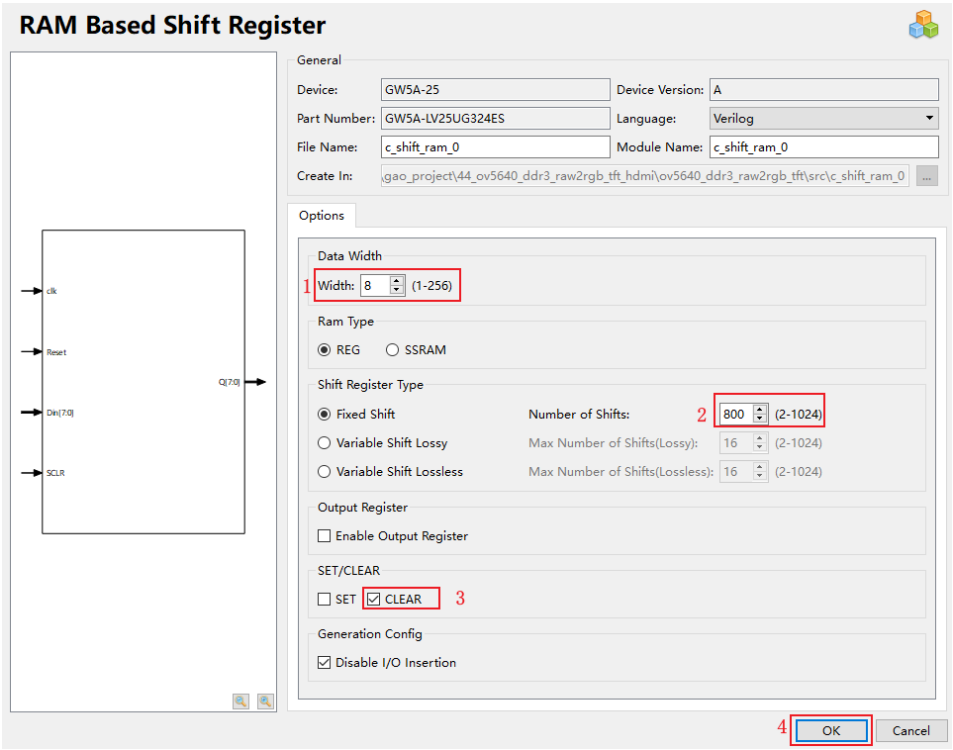


图 44-10 本次实验最终配置界面

当我们设置好移位寄存器对一行图像数据缓存后，接下来我们就需要想办法得到相邻的四个数据。

44.3.1.2得到 2*2 矩阵

在上面 RAW2RGB 的原理中有说到，当移位寄存器中一次存储了一行数据的时候，只需要对其输入端和输出端的前两位进行差值计算就能得到输出端第一位数据的 RGB 像素值。所以我们只需在数据输入输出时对其进行两级缓存即可，该部分代码如下所示：

```
wire [7:0]RAW_Data;
reg [7:0]r0_RAW_Data;
reg [7:0]r1_RAW_Data;
wire [7:0]RAW_Data_s;
reg [7:0]r0_RAW_Data_s;
reg [7:0]r1_RAW_Data_s;

always@(posedge Clk)begin
    r0_RAW_Data <= #1  r1_RAW_Data;
    r1_RAW_Data <= #1  RAW_Data;

    r0_RAW_Data_s <= #1  r1_RAW_Data_s;
    r1_RAW_Data_s <= #1  RAW_Data_s;
end
```

在这段代码中，RAW_Data 为移位寄存器的输入数据，RAW_Data_s 为移位寄存器的输出数据，这两个数据经过两级寄存后便能得到四个数据，构成一个 2x2 插值矩阵，如图 44-11 所示。

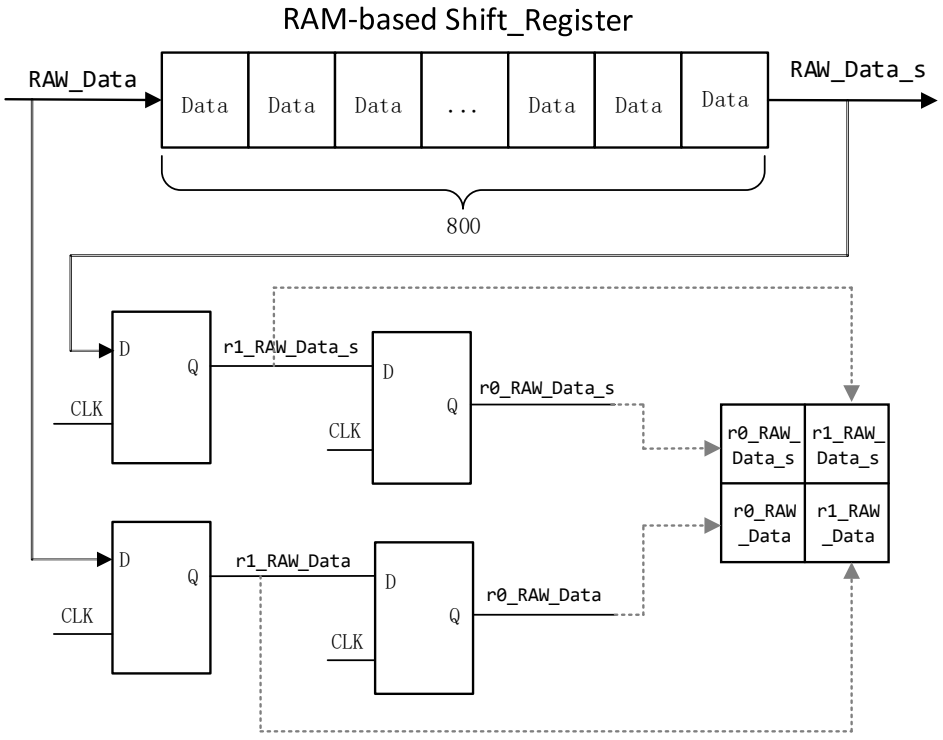


图 44-11 插值矩阵获取

这里需要注意的一点是没数据在移位寄存器中的排列顺序与在像素矩阵（或 2x2 矩阵）中的排列顺序是不一样的。例如数据在写入移位寄存器中时，新写入的数据总在左侧，即 r1_RAW_Data 应该在 r0_RAW_Data 的左侧。而这两个数据在经过两级寄存后，先被寄存的 r0_RAW_Data 会先输出到 2x2 矩阵中，并存放于相对左侧的位置，后寄存的 r1_RAW_Data 则位于右侧。数据在从移位寄存器中输出，经过两级寄存器后在 2x2 矩阵中的位置关系也同理。

至此，我们便获得了所需的 2x2 矩阵，接下来只需对该 2x2 矩阵进行插值运算就可以得到相应点的 RGB 像素值。但这也产生了一个新的问题，那就是我们该如何确定 2x2 矩阵中颜色分量的排布关系。

44.3.2像素位置的确定和颜色转换

在 OV5640 的官方数据手册中给出了如图 44-12 所示的像素分布图：

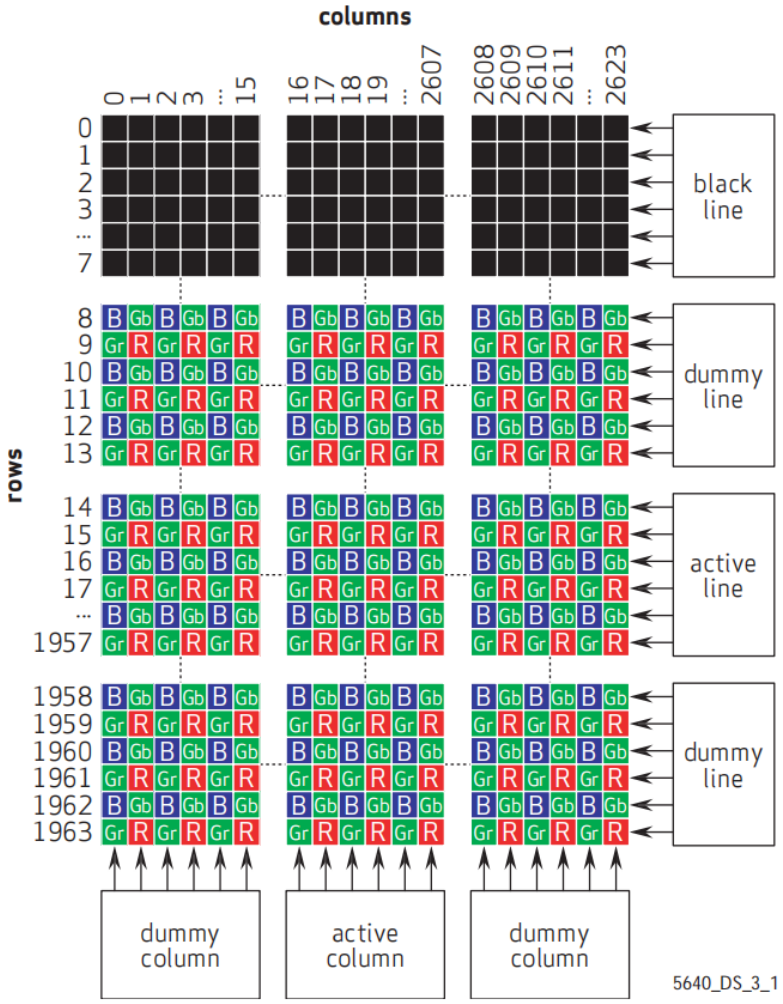


图 44-12 像素物理分布

由该分布图可看到，OV5640 内部有一个 2624*1964 的像素矩阵，但是实际能够输出的部分仅仅为 active line 与 active column 交汇的像素点，也就是中间的像素矩阵，其余部分用于校准和差值。由于感光像素矩阵中，每个奇数行间隔防止绿色（Gr）和红色（R）感应像素，每个偶数行间隔放置绿色(Gb)和蓝色（B）感应像素，每个奇数列间隔放置绿色和蓝色感应像素，每个偶数列间隔放置红色和绿色感应像素观察像素的行和列。我们很容易就可以发现只要知道了像素点坐标的奇偶值就能判断出它对应的颜色分量，又由于在 2X2 矩阵中红色与蓝色为对角关系，两个绿色为对角关系，我们用 0 和 1 代表像素点行与列的奇偶值，0 代表偶数，1 代表奇数，按图 44-3 所得到的规律，从图 44-12 中截取了四个像素矩阵得到图 44-13 的位置关系

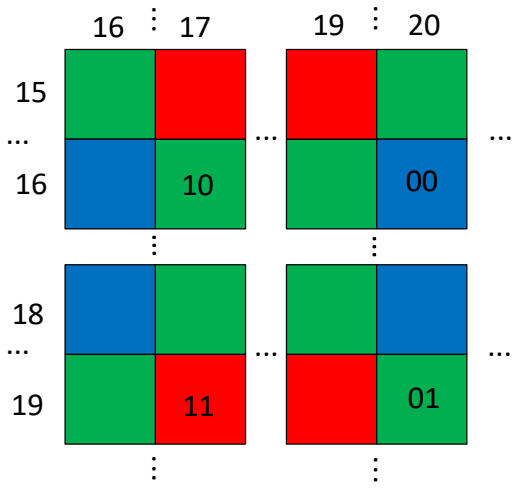


图 44-13 像素位置关系

可以看到，当坐标像素点为第 16 行第 17 列时，其相应的坐标则为（17,16），取坐标的奇偶值便能得出该点为 10，此时红色分量与蓝色分量分别位于 2X2 像素矩阵的右上角和左下角。同理，也可推出其余三种情况。

在知道了像素的排列位置后，我们便能按照相应的插值计算求出相应点的 RGB 像素值。至于像素点的位置坐标，我们只需通过取 disp_driver 模块中场同步扫描计数器与列同步扫描计数器的最低位即可得到。为了方便设计，我们取 2X2 矩阵中最新输入的数据，即右下角的像素位置作为定位，也就是两级缓存的四个数据中的 r1_RAW_Data。

为了方便理解，我们对 2X2 矩阵中四个位置进行命名，左上角为 DLU，D 为 data，L 为 left，U 为 up；右上角为 DRU，R 为 right；左下角为 DLD，第二个 D 为 down；右下角为 DRD，可得出图 44-14 所示的关系：

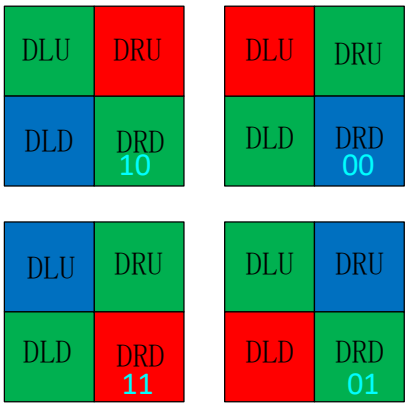


图 44-14 2*2 插值矩阵中像素四种对应关系

当右下角行与列奇偶值为 10 时，绿色像素分量值为（DLU+DRD）/2，红色像素分量值为 DRU，蓝色像素分量值为 DLD。同理其他情况下三种颜色分量的值也能一一求出。该部分代码设计如下：

```
wire [7:0]DLU = r0_RAW_Data_s;
wire [7:0]DRU = r1_RAW_Data_s;
wire [7:0]DLD = r0_RAW_Data;
wire [7:0]DRD = r1_RAW_Data;
always@(posedge Clk)begin
    r_Xaddr <= #1 Xaddr;
    r_Yaddr <= #1 Yaddr;
end

always@(posedge Clk)
    case({r_Xaddr,r_Yaddr})
        // R  Gr
        // Gb B
        2'b00:
            begin
                RED <= #1  DLU;
                GREEN_r <= #1  DLD + DRU;
                BLUE <= #1  DRD;
            end

        // Gb B
        // R  Gr
        2'b01:
            begin
                RED <= #1  DLD;
                GREEN_r <= #1  DLU + DRD;
                BLUE <= #1  DRU;
            end
```

```
        // Gb R
        // B  Gr
        2'b10:
            begin
                RED <= #1  DRU;
                GREEN_r <= #1  DLU + DRD;
                BLUE <= #1  DLD;
            end

        // B  Gb
        // Gr R
        2'b11:
            begin
                RED <= #1  DRD;
                GREEN_r <= #1  DLD + DRU;
                BLUE <= #1  DLU;
            end
    endcase
```

通过对 Xaddr，Yaddr 像素在矩阵中行列坐标的最后一位，进行一拍寄存，我们可以得到 2x2 矩阵右下角，也就是 r1_RAW_Data 信号的行场奇偶坐标。利用该坐标，我们就能够判定出 2X2 矩阵中各像素分量的位置排列进而计算该点颜色分量。

在完成了像素位置定位后，该模块的设计也已经基本完成，接下来即可对模块进行仿真验证了。

44.4RAW 转 RGB 模块仿真验证

为了模拟 RAW2RGB 模块的应用场景，我们单独设计了一个名为 RAW2RGB 的工程，工程中添加了 RAW2RGB 模块和 disp_driver 模块，其中 RAW2RGB 模块包含了上文我们提到的 RAM-based Shift Register 核，用于获取相邻的四个数据；RAW2RGB 模块用于数据从 RAW 格式到 RGB888 格式转换；disp_driver 模块用于产生对图像数据的行场同步计数，进而对像素定位。

44.4.1仿真测试激励

为了方便验证，我们将输入 RAW2RGB 模块的数据设定为固定值，当场同步信号为奇数值，输入数据为 00f7；当场同步信号为偶数值，输入数据为 f700，即最终得到的值为 R=F7，G=00，B=7F，观察各个信号的数据变化。本例提供的工程的仿真激励代码如下：

```
`timescale 1ns/1ns
```

```
module RAW2RGB_tb;

    reg Clk;
    reg Rst_n;
    wire DinReq;
    wire [7:0]RAW_Data;

    wire [11:0]H_Addr;
    wire [11:0]V_Addr;

    reg DoutReq;
    wire [7:0]RED;
    wire [7:0]GREEN;
    wire [7:0]BLUE;
    wire [7:0]Disp_Red;
    wire [7:0]Disp_Green;
    wire [7:0]Disp_Blue;
    wire      DataReq;
    wire      Disp_HS;
    wire      Disp_VS;
    wire      Disp_DE;
    wire      Disp_PCLK;

    wire [15:0]data;

    RAW2RGB RAW2RGB(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .DinReq(DinReq),
        .RAW_Data(RAW_Data),
        .Xaddr(H_Addr[0]),
        .Yaddr(V_Addr[0]),
        .RED(RED),
        .GREEN(GREEN),
        .BLUE(BLUE),
        .DoutReq(DataReq)
    );

    disp_driver disp_driver(
        .ClkDisp(Clk),
        .Rst_n(Rst_n),
        .Data({RED,GREEN,BLUE}),
        .DataReq(DataReq),
        .H_Addr(H_Addr),
        .V_Addr(V_Addr),
        .Disp_HS(Disp_HS),
```

```
        .Disp_VS(Disp_VS),
        .Disp_Red(Disp_Red),
        .Disp_Green(Disp_Green),
        .Disp_Blue(Disp_Blue),
        .Disp_DE(Disp_DE),
        .Disp_PCLK(Disp_PCLK)
    );

    initial Clk = 1;
    always#10 Clk = ~Clk;

    initial begin
        Rst_n = 0;
        DoutReq = 0;
        #201;
        Rst_n = 1;
        #20000000;
        $stop;
    end

    reg flag;
    always@(posedge Clk)
    if(DinReq)
        flag <= #1 ~flag;
    else
        flag <= 0;

    assign data = V_Addr[0]?16'h00f7:16'h7f00;

    assign RAW_Data = DinReq?flag?data[7:0]:data[15:8]:0;

endmodule
```

由于数据在输入时有 16 位，而 RAW2RGB 模块一次只能处理 8 位数据，因此增加了一个 flag 信号，让其在数据进来时每时钟翻转一次，从而依次读取输入数据的高 8 位、低 8 位。

44.4.2仿真结果分析

本次设计仿真工程位于 RAW2RGB 文件夹下，仿真波形如下图 44-15 所示。

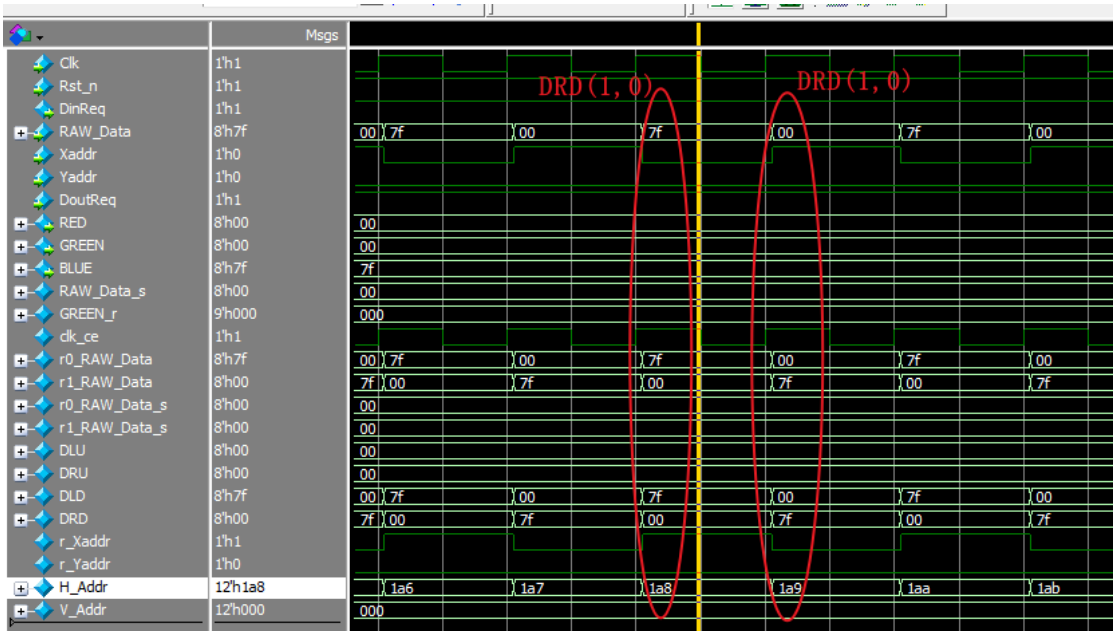


图 44-15 DRD（1,0）与 DRD（0,0）仿真波形

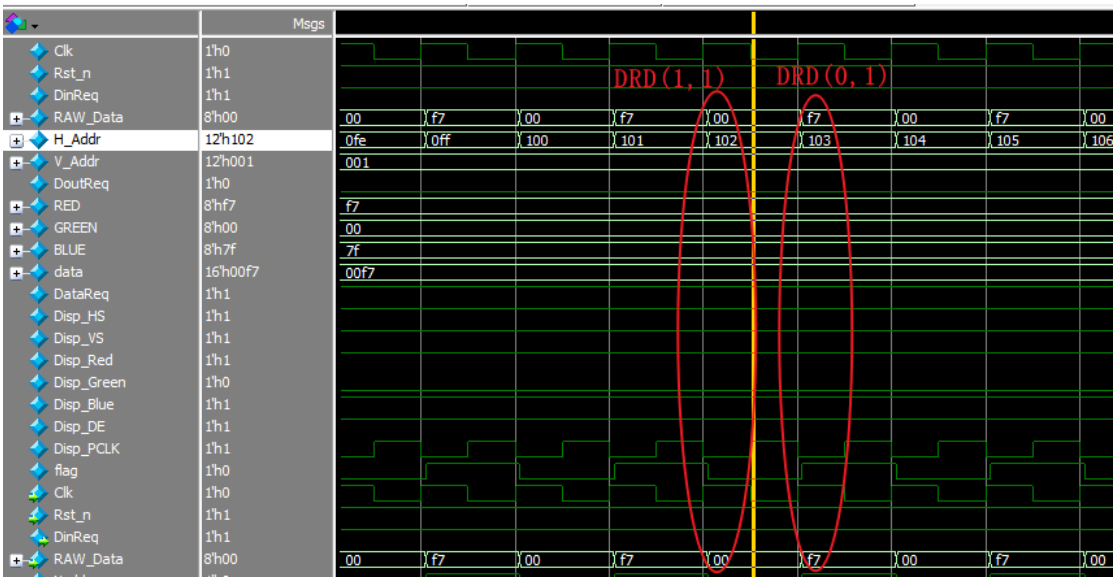


图 44-16 DRD（1,1）与 DRD（0,1）仿真波形

这里的 r_Xaddr 信号和 r_Yaddr 信号分别为行、列有效数据扫描计数器最低位的打拍信号（打拍是为了与数据同步），用于计算 RGB888 格式数据。

图 44-15 和

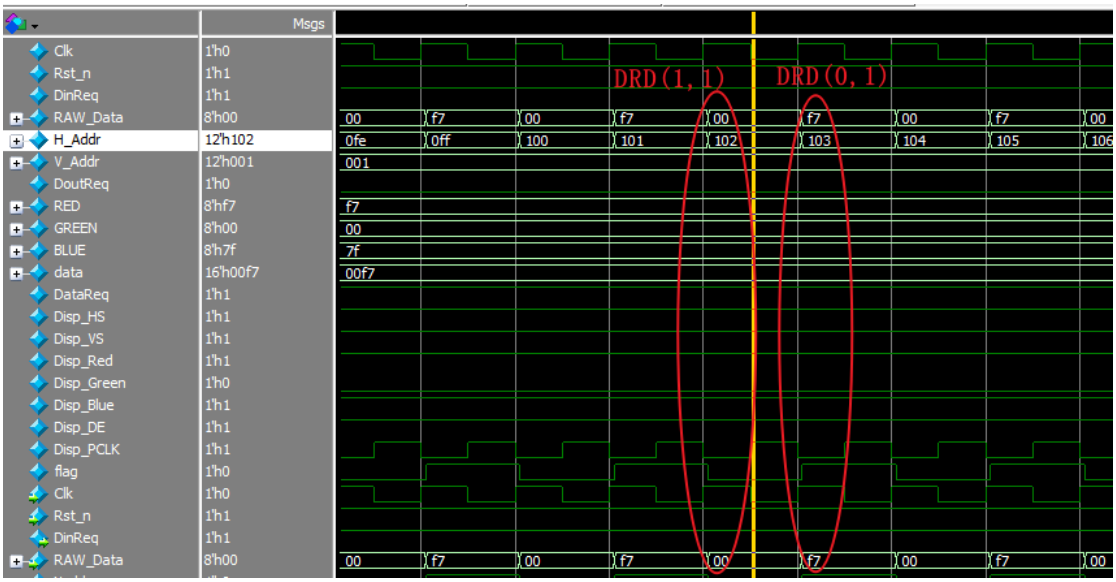


图 44-16 中圈出了 RGB 颜色分量的四种排列情况下的仿真波形，以图中 DRD（1,1）时刻为例，可以看到，此时 V_Addr[0]为 1，所以 data 为 00f7，由于 flag 值为 0，所以 RAW_Data 值为高 8 位，即 00，证明数据输入无误。

由于 r_Xaddr 与 r_Yaddr 值都为 1，所以该点的行场奇偶坐标为（1,1），因此该点的颜色计算方式如图 44-17 中左下角的情况：

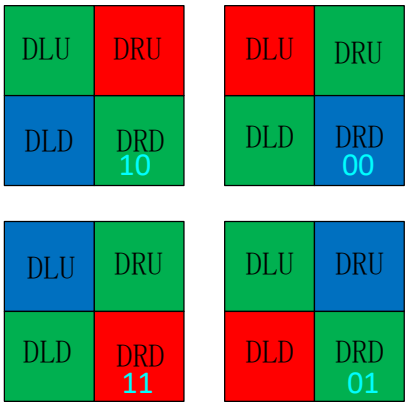


图 44-17 2x2 插值矩阵中像素四种对应关系

DLU，DRU，DLD，DRD 的含义在上文代码中已经讲过，所以 RED 的值为 DRD 的值，即 f7；GREEN 的值为（DLD+DRUD）/2，即 00；BLUE 的值为 DLU，即 7f，证明数据的输出无误。

同样的，其他三种情况下的各颜色分量计算同理。

通过对各信号数据的对比计算，验证了我们设计的 RAW2RGB 模块在仿真方面是没有问题的。设计 RAW 转 RGB 算法模块的最终目的，是希望将其应用到实际的案例中，将真实的图像数据转换为 RGB 图像，以进行进一步的显示或处理。因此我们可以将其应用到具体的工程实例中，对其进行板级调试和验证。

44.5基于 OV5640 的实时 RAW2RGB 采集显示系统

要想直观地看到该模块是否能正常工作，我们需要一个采集显示系统。因而我们将 RAW2RGB 模块与前面章节设计完成的 ov5640_ddr3_tft 采集显示系统相结合，设计得到一个能够将摄像头采集到的 RAW 格式图像数据，在 FPGA 内部完成 RAW 转 RGB 转换后，再将转换得到的 RGB 图像输出到显示设备上的工程。考虑到显示效果展示，我们将工程中的 VGA 显示修改为 5 寸 TFT 显示，用于表明设计能够实现开发板配套 TFT 屏的显示。

44.5.1图像采集转换显示系统介绍

对于设计而言，我们可以在 RAW 格式数据被存入 DDR3 前使用 RAW2RGB 模块进行 RGB888 的转换，也可以在 RAW 格式数据被从 DDR3 中读出后进行 RGB888 的转换。考虑前一种方式中，RAW 在转换为 RGB888 后，会由原来的 8 位数据变为 24 位数据，因此存储 RGB888 格式数据到 DDR 时将需要 3 倍于存储 RAW 格式数据时所需的带宽。出于节约性能的考虑，设计选择了后者，所以整个系统结构如图 44-18 所示。

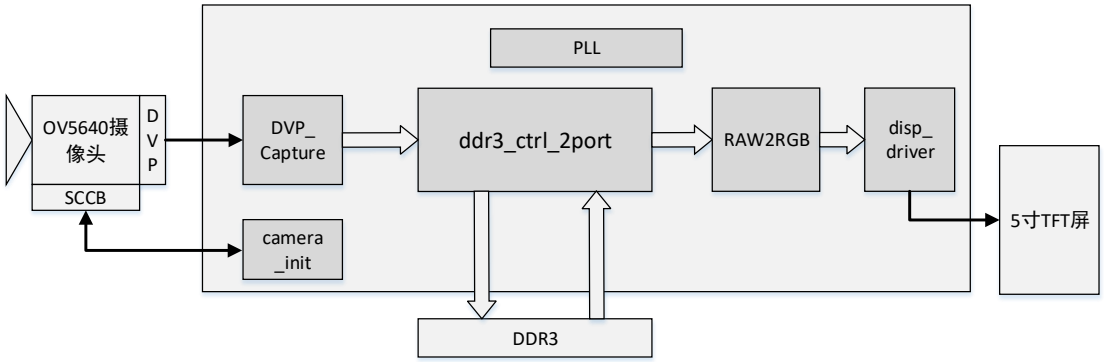


图 44-18 系统结构

考虑 5 寸 TFT 屏的所支持的最大分辨率为 800*480，设计中摄像头输出的图像分辨率不能大于 800*480。整个系统在工作时，FPGA 首先会通过 SCCB 接口对 OV5640 配置，使其将采集到的图像数据以 RAW 格式输出 800*480 分辨率的原始图像数据。随后，摄像头通过 DVP 接口将 8 位的 RAW 格式数据输出给 FPGA。FPGA 内部再通过 DVP_Capture 模块将连续的两个 8 位数据拼接为 16 位写入到片外存储器 DDR3 中。数据在需要时被读出，送入到 RAW2RGB 模块中，并被转换为 RGB888 格式数据后输出给 disp_driver 模块。disp_driver 模块会根据设定的时序参数结合数据产生相应的时序信号，与数据一起输出。此时的

RGB888 图像数据和时序信号会被送给 TFT 屏和 dvi_encoder 模块。TFT 屏只能显示 RGB565 格式数据，因此需要对数据的各个颜色分量低位进行舍弃。

44.5.2配置 OV5640 输出 RAW 格式数据

设计使用原工程中的 OV5640 初始化寄存器表配置 OV5640 摄像头，该配置表中将 OV5640 摄像头相关寄存器地址以及地址对应的配置值存于二维数组 ROM 中。ROM 的位宽为 24 位，其中高 16 位用于存放寄存器地址，低 8 位用于存放对应的配置值。程序在运行时，摄像头初始化模块会读取该配置表，根据其中的地址以及配置值使用 IIC 初始化 OV5640 摄像头

该配置表默认配置 OV5640 输出的为 RGB565 格式数据，设计需要的是 RAW 格式数据。这里我们需要修改 OV5640 中 0x4300 寄存器的值为 00，0x501f 寄存器的值为 03。前者用于控制输出数据的格式，后者用于控制摄像头 mux 的控制格式。这两个寄存器各个位的具体功能较为复杂，这里暂不详细介绍，用户可以参考 OV5640 手册“OV5640_CSP3_DS_2.01_Ruisipusheng”。

这两个寄存器存的地址以及配置值存放于配置表的 ROM[56]和 ROM[57]数组中，因此我们需要将 rom[56]与 rom[57]中的值分别修改为 24'h4300_00 和 24'h501f_03，如下：

```
rom[56] = 24'h4300_00; // RAW
rom[57] = 24'h501f_03; // RAW
```

除此之外，还需要设置摄像头的输出图像分辨率为 800*480，OV5640 摄像头的输出分辨率由 0x3808~380b 寄存器控制。配置表中使用 parameter 对相关参数进行了全局修饰，因此，只需在顶层，将 IMAGE_WIDTH 与 IMAGE_HEIGHT 信号的值分别改为 800*480 即可，代码如下：

```
parameter IMAGE_WIDTH  = 800;
parameter IMAGE_HEIGHT = 480;
```

配置修改完成后，OV5640 摄像头便能按照需求，产生我们所需要的 800*480 分辨率的 RAW 格式数据。这些数据最终倍存储进 DDR 中，在需要时，由 RAW2RGB 模块读出。

44.5.3添加 RAW2RGB 模块

RAW2RGB 模块包含一个 RAM based Shift Register IP 核，在添加 RAW2RGB 模块时，需要检查设计中是否包含该 IP。本次设计中，该 IP 核的配置界面如所示。

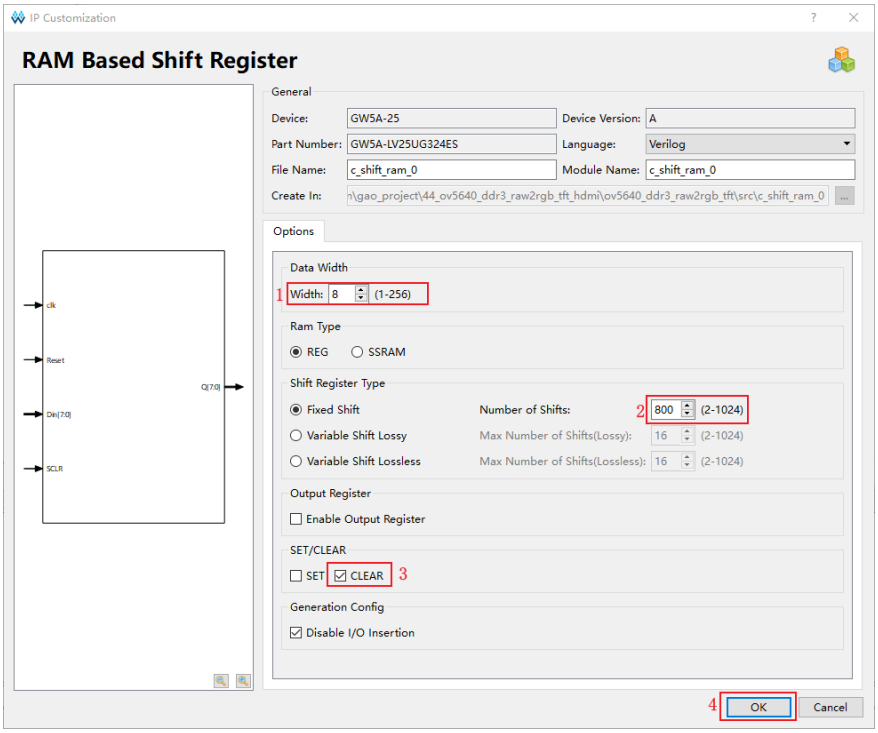


图 44-19 配置 RAM based Shift Register

然后设置时钟使能，调用 DCE 原语，代码如下所示：

```
wire clk_ce;
DCE DCE(
    .CLKIN(Clk),
    .CE(DinReq),
    .CLKOUT(clk_ce)
);
```

由于 RAW2RGB 模块一次只能接收 8 位数据，而 rd_ddr3_fifo 模块每次输出 16 位数据，因此我们需要在添加一个标志信号，用来控制 RAW2RGB 模块依次接收这 16 位数据的高 8 位与低 8 位，相关代码以及 RAW2RGB 模块的例化如下：

```
//RAW2RGB
wire [7:0]    RAW_Data;
reg           flag;
wire          DinReq;
wire [7:0]    RED;
wire [7:0]    GREEN;
wire [7:0]    BLUE;
wire          DataReq;

assign RAW_Data = (~flag)?rdfifo_dout[15:8]:rdfifo_dout[7:0];

always@(posedge clk_disp)
```

```
if(DinReq)
    flag <= ~flag;
else
    flag <= 0;

RAW2RGB RAW2RGB(
    .Clk(clk_disp),
    .Rst_n(~g_rst_p),
    .DinReq(DinReq),
    .RAW_Data(RAW_Data),
    .Xaddr(H_Addr[0]),
    .Yaddr(V_Addr[0]),
    .RED(RED),
    .GREEN(GREEN),
    .BLUE(BLUE),
    .DoutReq(DataReq)
);
```

44.5.4产生读请求信号

完成了 RAW2RGB 模块的添加和例化后，我们还需要产生一个读请求信号，用于通过 rd_ddr3_fifo 从 ddr3 中读取存储的 RAW 图像数据。RAW2RGB 模块每次会读取 8 位数据，rd_ddr3_fifo 每次传输 16 位数据，因此可以将 flag 信号与 DinReq 信号相与，产生对 rd_ddr3_fifo 的请求信号，而 rd_ddr3_fifo 由 ddr3_ctrl_2port 模块控制，因此修改 ddr3_ctrl_2port 模块的顶层例化如下：

```
ddr3_ctrl_2port ddr3_ctrl_2port(
    .clk(loc_clk50m)                 ,       //50M 时钟信号
    .pll_lock(pll_lock)             ,
    .clk_200m(clk_200M)              ,       //DDR3 参考时钟信号
    .sys_rst_n(reset_n)             ,       //外部复位信号
    .init_calib_complete(init_calib_complete) ,     //DDR 初始化完成信号

    //用户接口
    .rd_load(Frame_Begin)              ,    //输出源更新信号
    .wr_load(~camera_init_done)              ,    //输入源更新信号
    .app_addr_rd_min(28'd0)      ,    //读 DDR3 的起始地址
    .app_addr_rd_max(app_addr_max)      ,    //读 DDR3 的结束地址
    .rd_bust_len(burst_len)          ,    //从 DDR3 中读数据时的突发长度
    .app_addr_wr_min(28'd0)      ,    //写 DD3 的起始地址
    .app_addr_wr_max(app_addr_max)      ,    //写 DDR 的结束地址
    .wr_bust_len(burst_len)          ,    //向 DDR3 中写数据时的突发长度

    .wr_clk(wrfifo_clk)              ,//wr_fifo 的写时钟信号
    .wfifo_wren(wrfifo_wren)           , //wr_fifo 的写使能信号
    .wfifo_din(wrfifo_din)            , //写入到 wr_fifo 中的数据
```

```
    .wrfifo_full(),
    .rd_clk(rdfifo_clk)                , //rd_fifo 的读时钟信号
    .rfifo_rden(flag & DinReq)           , //rd_fifo 的读使能信号
    .rdfifo_empty(),
    .rfifo_dout(rdfifo_dout)           , //rd_fifo 读出的数据信号

    //DDR3
    .ddr3_dq(IO_ddr_dq)              ,   //DDR3 数据
    .ddr3_dqs_n(IO_ddr_dqs_n)          ,   //DDR3 dqs 负
    .ddr3_dqs_p(IO_ddr_dqs)          ,   //DDR3 dqs 正
    .ddr3_addr(O_ddr_addr)           ,   //DDR3 地址
    .ddr3_ba(O_ddr_ba)               ,   //DDR3 banck 选择
    .ddr3_ras_n(O_ddr_ras_n)           ,   //DDR3 行选择
    .ddr3_cas_n(O_ddr_cas_n)           ,   //DDR3 列选择
    .ddr3_we_n(O_ddr_we_n)           ,   //DDR3 读写选择
    .ddr3_reset_n(O_ddr_reset_n)         ,   //DDR3 复位
    .ddr3_ck_p(O_ddr_clk)            ,   //DDR3 时钟正
    .ddr3_ck_n(O_ddr_clk_n)            ,   //DDR3 时钟负
    .ddr3_cke(O_ddr_cke)             ,   //DDR3 时钟使能
    .ddr3_cs_n(O_ddr_cs_n)           ,   //DDR3 片选
    .ddr3_dm(O_ddr_dqm)              ,   //DDR3_dm
    .ddr3_odt(O_ddr_odt)                 //DDR3_odt
);
```

44.5.5修改 disp_driver 配置参数

在 RAW2RGB 成功读取数据并转换为 RGB888 后，数据会送到 disp_driver 模块，disp_driver 模块会根据设定好的时序阐述产生相关时序信号。为了方便使用，我们将常用的参数以条件编译的方式先存放在 disp_parameter_cfg 中。而设计中此时使用的是 1280*720 分辨率下 RGB565 的时序参数，我们需要将其修改为我们所需要的 800*480 分辨率下 RGB888 的时序参数。这里我们将其中 49~83 行附近代码修改为如下：

```
//使用 4.3 寸 480*272 分辨率显示屏
//`define HW_TFT43

//使用 5 寸 800*480 分辨率显示屏
 `define HW_TFT50

//使用 VGA 显示器，默认为 640*480 分辨率，24 位模式，其他分辨率或需 16 位模式
可在代码 63 行至 75 行进行重配置
//`define HW_VGA

//=====================================
//以下宏定义选择用于根据显示设备进行位模式和分辨率 2 个参数的设置
```

```
//=====================================
`ifdef HW_TFT43  //使用 4.3 寸 480*272 分辨率显示屏
  `define MODE_RGB565
  `define Resolution_480x272 1 //时钟为 9MHz

`elsif HW_TFT50  //使用 5 寸 800*480 分辨率显示屏
//  `define MODE_RGB565
  `define MODE_RGB888
  `define Resolution_800x480 1 //时钟为 33MHz

`elsif HW_VGA    //使用 VGA 显示器，默认为 640*480 分辨率，24 位模式
//=====================================
//可选择其他分辨率和 16 位模式，需用户根据实际需求设置
//代码 70~71 行设置位模式
//代码 73~78 行设置分辨率
//=====================================
  `define MODE_RGB565
  // `define MODE_RGB888
  // `define Resolution_640x480   1 //时钟为 25.175MHz
  //`define Resolution_800x600   1 //时钟为 40MHz
  //`define Resolution_1024x600  1 //时钟为 51MHz
  //`define Resolution_1024x768  1 //时钟为 65MHz
  `define Resolution_1280x720  1 //时钟为 74.25MHz
  //`define Resolution_1920x1080 1 //时钟为 148.5MHz
`endif
```

这里将输出图像格式设置为了 RGB888 格式，将分辨率设置为了 800*480。而根据条件编译的结果，在 132~145 行附近，可以看到此时输出图像的时序参数如下：

```
`elsif Resolution_800x480
  `define H_Total_Time    12'd1056
  `define H_Right_Border  12'd0
  `define H_Front_Porch   12'd40
  `define H_Sync_Time     12'd128
  `define H_Back_Porch    12'd88
  `define H_Left_Border   12'd0

  `define V_Total_Time    12'd525
  `define V_Bottom_Border 12'd8
  `define V_Front_Porch   12'd2
  `define V_Sync_Time     12'd2
  `define V_Back_Porch    12'd25
  `define V_Top_Border    12'd8
```

至此，disp_driver 模块便能按照正确的时序参数产生 800*480 分辨率所需的时序信号了。

44.5.6修改顶层模块连接

完成了 disp_driver 的参数配置后，接下来还需要需改 disp_driver 模块与 TFT 屏接口的连接。由于设计结构的变化，disp_driver 模块不再直接从 rd_ddr3_fifo 读取数据，而是通过 DataReq 信号来获取 RAW2RGB 模块转换好的 RGB888 格式数据。并结合设定好的时序参数，产生时序信号。数据按照各个颜色分量从 disp_driver 模块输出，同时交由 TFT 屏核 dvi_encoder 模块。考虑到 TFT 屏使用的 RGB565 格式数据，因此需要舍弃对应颜色分量的低位，以满足信号格式需求，因此，disp_driver 模块修改后的顶层例化如下：

```
 //TFT 显示
disp_driver disp_driver(
    .ClkDisp(clk_disp),
    .Rst_n(reset_n),
    .Data({RED,GREEN,BLUE}),
    .DataReq(DataReq),
    .H_Addr(hcount),
    .V_Addr(vcount),
    .Disp_HS(TFT_hs),
    .Disp_VS(TFT_vs),
    .Disp_Red(video_r),
    .Disp_Green(video_g),
    .Disp_Blue(video_b),
    .Frame_Begin(Frame_Begin),
    .Disp_DE(TFT_de),
    .Disp_PCLK(TFT_clk)
);
assign TFT_rgb = {video_r[7:3],video_g[7:2],video_b[7:3]};
```

完成了模块间的连接后，我们还需要在顶层添加一个输出信号，用于去当 TFT 的背光显示，该信号高电平有效，代码如下：

```
//TFT Interface
output          TFT_pwm       , //TFT 背光控制

assign TFT_pwm = 1'b1;
```

至此便完成了整个设计代码层面的修改，引脚分配和“ov5640_ddr3_tft”的工程一致，我们这里不需要修改，直接生成数据流文件进行板级验证即可。

44.6板级验证

本节介绍在高云开发板上使用 OV5640 摄像头和 5 寸 TFT 屏进行“基于 OV5640 的实时 RAW2RGB 采集显示”实验的验证。

44.6.1系统所需硬件

1. 高云开发板 x1
2. OV5640 摄像头 x1
3. 高云下载器 x1
4. 5 寸 TFT 屏 x1
5. 电源线 x1

44.6.2硬件连接

将 OV5640、5 寸 TFT 屏、下载器、电源线依次连接开发板，如图 44-20 所示。

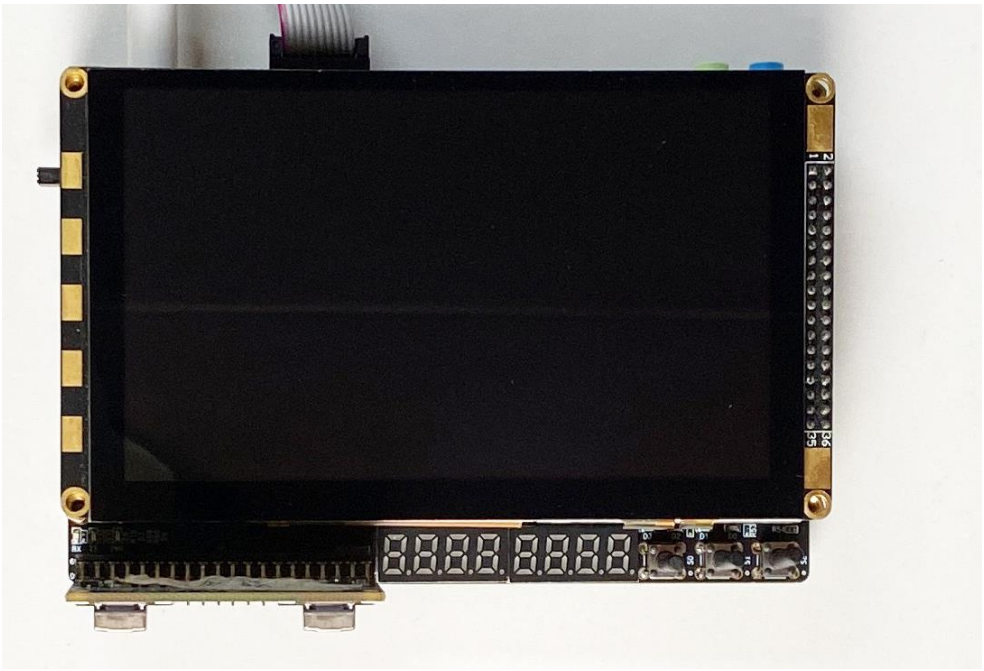


图 44-20 整体硬件连接图

连接完毕后拨动开关，给开发板上电，接下来将生成的数据流文件烧录到开发板中，系统开始工作，TFT 显示效果如下图 44-21 所示。

图 44-21 TFT 屏显示效果图

从上图可以看出到图像显示正常，没有出现任何撕裂重叠现象，颜色层次清晰，说明设计成功。

44.7常见问题说明

对于本节工程需要注意的有以下几点：

1. 数据时移位写入寄存器的，而在像素矩阵中是从左到右依次写入的，因而其左右顺序与寄存器中是相反的。
2. 摄像头默认输出为 RGB565 数据格式的数据，所以读者在进行板级验证前需要找到摄像头的 init_table 表，将 rom[56]与 rom[57]寄存器的值分别改为 24'h4300_00 和 24'h501f_03 即可，这两个寄存器控制摄像头的输出格式，详细可以参考 OV5640 的数据手册。

44.8总结与思考

本节设计实现了从 RAW 数据到 RGB 格式数据的转换，通过调用一个移位

寄存器，实现了对相邻行图像数据的提取，本次设计的重点是对数据的定位，以及相应排列顺序时不同的计算方式。学习本节时读者可以参考相应的视频教程相互理解印证，建议读者能够跟随本实验内容，完整的进行整个实验。

45 以太网硬件 UDP 协议栈设计和开发

工程源码	|----02_设计实例 |----ch45_udp_rgmii
相关视频课程	
说明	

章节导读

本章我们将开始学习以太网通信相关内容，由于以太网通信涉及到的内容和知识点非常的庞杂，学习需要花费一定的时间和精力。为了让大家快速上手 FPGA 的应用，本节将先介绍我们提供的 2 个基于 FPGA 的应用实验的操作方法，让大家先从最直观的角度体验 FPGA 实现以太网通信的魅力。

45.1前言

以太网（Ethernet）是一种计算机局域网技术。IEEE 组织的 IEEE 802.3 标准制定了以太网的技术标准，它规定了包括物理层的连线、电子信号和介质访问层协议的内容。

以太网是在 20 世纪 70 年代研制开发的一种基带局域网技术，使用同轴电缆作为网络媒体，采用载波多路访问和冲突检测(CSMA/CD)机制，数据传输速率达到 10MBPS。随着技术的发展，当前以太网已经能够做到 100Gbps 的超高传输速率了。

自打问世以来，以太网就以其灵活的互联特性和强大的数据传输能力，成为了几乎所有电脑设备的标配。通过一根网线和一个网络访问账号，用户就可以在支持以太网的设备上访问世界各地服务器上的数据。从最开始的电话线传输，发展到 10M、100M、1Gbps 以太网，以及现在的 10G、40G、100G 以太网。从最早期只能传输简单文本信息，到现在可以实时传输超高清视频数据流，以太网一直是我们通过 Internet 进行工作、学习、生活、娱乐的强力支撑。

虽然目前以太网的传输速率已经可以达到 10Gbps、40Gbps、100Gbps 的速率，但是它们主要用于服务器内部进行高速数据交换，或者电信骨干网络。对于一般的消费级和工业应用，大部分都还是以百兆和千兆以太网为主。

上面提到，以太网主要规定的是物理层和介质访问层协议的内容，没有对更上层的协议进行定义。而我们常用的以太网通信应用，例如网络聊天，视频聊天，远程文件下载等，都是基于以太网物理层，通过定义上层的具体数据传

输协议实现的。例如 QQ 聊天消息收发是基于 UDP 协议加腾讯自定义上层协议，实时视频数据的传输也多采用 UDP 协议，而文件传输则使用 FTP 传输协议。除开这些协议，还有应用非常广泛的网页传输协议 HTTP 和可靠数据传输使用的 TCP 协议等。

如今，工业以太网非常火热，使用较多的有 5 大主流协议，包括 Ethernet/IP、PROFINET、POWERLINK、EtherCAT、SERCOSIII。笔者了解较多的是 EtherCAT 协议，该协议主要用作于运动控制系统，很多的自动化应用系统都选择采用 EtherCAT 协议进行通信，完成多轴机械臂的协同控制。而使用 FPGA 实现 EtherCAT 从站，使用 SoC FPGA 实现 EtherCAT 主站，能够带来更高的实时效率。

除了工业控制，以太网在视频监控领域也使用较多，比较典型的一类就是工业相机。采用以太网接口的工业相机，能够使用网线连接到接收端，相较于使用 USB 传输方案的工业相机，采用以太网传输能够显著增加数据传输距离，一般的千兆工业相机在无中继情况下传输距离能够达到 100 米，而 USB 的可靠传输距离大概在 5M 左右。

对于 FPGA 来说，网络通信是其最经典的应用之一。无论是移动通信基站，还是大型数据交换中心，都大量的使用了 FPGA 作为通信设备的核心器件。掌握以太网通信基本原理和使用 FPGA 实现网络通信的方法，不仅具有极强的学习价值，还拥有非常明确的实用、商用价值。

45.1.1FPGA 的以太网应用

在 FPGA 中，以太网通信也有着非常广泛的应用案例。除了上述提到的大型数据交换中心，另一个非常基础的应用就是大型 LED 显示屏。例如各种广场上的 LED 广告屏，大厦外墙的 LED 广告屏，这些显示屏一般安置在户外，其显示的内容则由放置在室内的计算机主机传输，计算机主机和 LED 显示屏之间，由于需要实时传输大量的图像数据，一般都使用以太网作为传输方式，在 LED 显示屏中使用 FPGA 实现以太网，完成实时图像数据的接收并驱动 LED 点阵显示。

由于 FPGA 实现以太网具有高效，低延时的特点，目前也有众多的安全厂商开发基于 FPGA 的硬件网络防火墙， 来防御对服务器的恶意网络攻击。

对于很多专用定制型场合， 为了进行高速数据采集和传输，也有众多使用 FPGA 实现的以太网来进行实时高效数据传输的应用，例如笔者就曾参与设计

过一个基于 FPGA 和以太网接口的高速数据采集传输系统， 用来解决数据的采集和实时无丢失传输问题。

45.1.2开发板硬件概述

为了让大家能够学习使用 FPGA 进行以太网应用开发的思路和方法，我们在芯路恒出品的众多 FPGA 开发板上，都设提供了以太网通信接口，通过该以太网电路，用户可以将 FPGA 采集或运算得到的数据传递给其他设备如 PC 或服务器，或者接收其他设备传输过来的数据并进行处理。

45.1.3FPGA 实现以太网通信协议

接触过以太网协议的用户，应该最常听说的就是 TCP/IP 协议。 确实，在计算机和嵌入式系统中，TCP/IP 协议应用非常广泛。 因此，当大家看到 FPGA 上带有以太网接口时，可能第一个想到的也是实现 TCP/IP 协议。这里，首先可以很肯定的告诉大家，使用 FPGA 实现 TCP/IP 协议是完全没有问题的，但是，实现的方式却不是大家最期望的直接使用 Verilog 编写协议层代码来实现。FPGA 发展到现在，三十多年了，却鲜见有成功商用的 RTL 级的 TCP/IP 的设计，而大部分使用 Verilog 或者 VHDL 实现的以太网传输，都是基于非常简单的 UDP 议的。当然，探索或者实现其中部分功能的人还是有的，只是，很难做到像 PC 那样灵活应用。

个人理解，TCP/IP 协议设计之初就是根据软件灵活性设计的，因此在很多设计考虑上，并不适合使用硬件逻辑实现。TCP/IP 协议非常的复杂，如果使用硬件逻辑实现，工程量必然十分浩大，而且功能和易用性都无法得到保证。

上面说过，在 FPGA 上，可以使用 Verilog 实现 UDP 协议来进行数据的传输。 UDP 协议是一种不可靠传输，发送方只负责将数据发送出去，而不管接收方是否正确的接收， 非常类似于 UART 串口传输。 这种传输方式从理论上讲是无法保证发送方发送的每一个数据包接收方都能正确接收的， 但是在很多场合，又是可以接受这种潜在的不可靠性的，例如视频实时传输显示。在这类系统中，由于数据并不需要进行运算并得到非常精确的结果用于其他功能，而仅仅是显示在屏幕上，因此可以接受一定程度的丢包或者误码。此类应用在 LED 大屏显示系统中非常广泛， 所以本节内容所讨论的基于 FPGA 的以太网通信就是围绕着实现高效的 UDP 数据传输这种目的展开的。

45.2基于 FPGA 的以太网通信实验实操

45.2.1以太网回环收发测试

在测试以太网通信速率和通信质量的应用中，最常用的测试方式就是进行回环测试。所谓回环测试，就是由网络连接中的一个节点作为主机，运行测试程序，该测试程序会向被测试的设备发送以太网数据包，被测试设备收到数据后，原封不动的将接收到的数据内容通过以太网发送给主机，主机收到被测设备发回的数据后，与自己发送的数据进行比对，或者以文本的形式显示，供测试者人工对比分析。

45.2.1.1UDP RGMII 回环测试工程介绍

为了完成基本的以太网回环测试，我们提供了一个基于 FPGA 的以太网回环测试 DEMO。该 DEMO 使用 UDP 协议， 接收 PC 发送的 UDP 数据包，提取出其中的数据部分并使用 UDP 协议发回给 PC。整个系统框图如下图 45-1 所示：

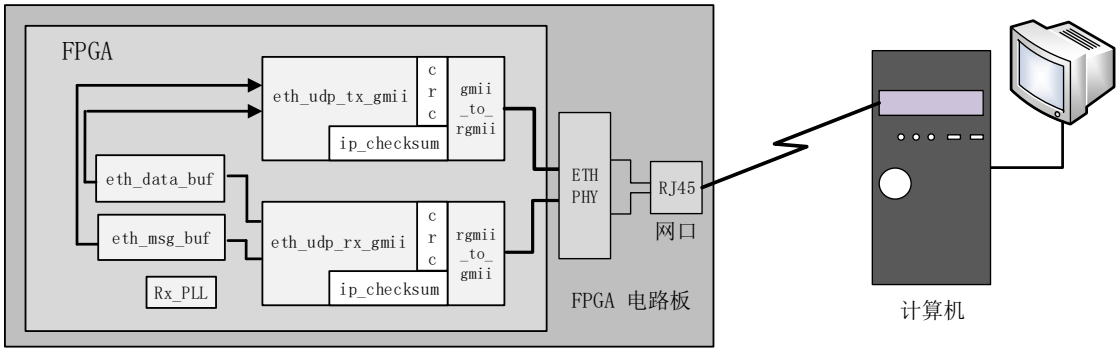


图 45-1 UDP RGMII 回环系统框图

本例程在小梅哥团队出品的高云开发板上使用 verilog 实现以太网 UDP 协议通信。FPGA 程序接收到上位机发来的 UDP 数据包，通过解析目标 MAC address 来确定是否是发给 FPGA 的数据包。如果是的话，把数据包中的数据部分保存到 fifo 中。 然后 FPGA 的发送程序再把 fifo 的数据包发送回上位机。

本例对应的例程工程压缩包名为 eth_udp_loopback_rgmii.rar， 该文件可在我们提供的配套资料中找到。解压工程到不含中文或者空格的目录中，打开工程后工程结构如图 45-2 所示：

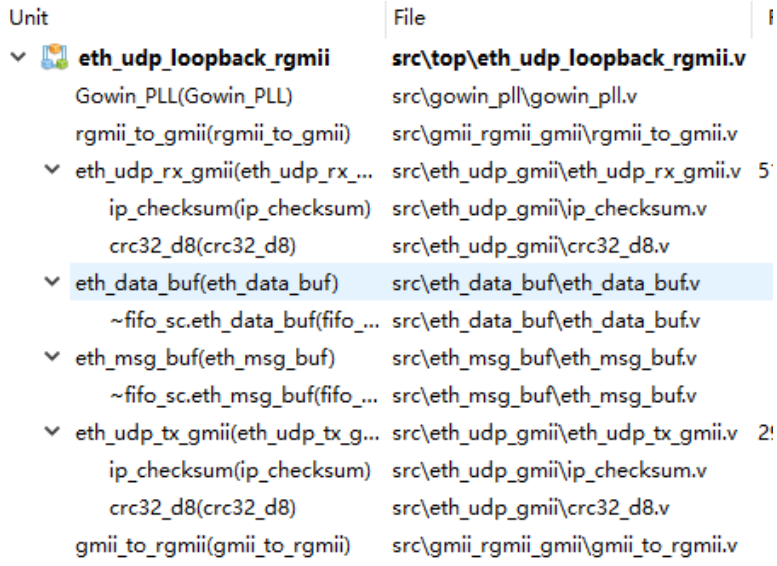


图 45-2 工程代码结构

这里简单为大家介绍下各个模块的功能，如下表 45-1 所示。

表 45-1 模块功能介绍

模块名	功能描述
Gowin_PLL	接收 pll 核，用于产生与 PHY 提供给 FPGA 的 rgmii_rx_clk 时钟有 90 度相移的时钟，保证 FPGA 在接收数据时，数据与时钟中心对齐。
eth_data_buf	数据缓存 fifo，用于缓存接收模块解析出的数据包中的数据
eth_msg_buf	信息缓存 fifo，用于缓存目的 MAC 地址、目的 IP 等信息
eth_udp_rx_gmii	UDP 协议接收模块，该模块能够完整的解析从以太网 PHY 的 GMII 接口收到的数据并解析得到发送方的 MAC 地址、IP 地址、UDP 报文的数据部分等。
rgmii_to_gmii	rgmii 接口到 gmii 接口转换模块，实现 rgmii 到 gmii 的转换
crc32_d8	CRC32 校验逻辑，实现对 MAC 层数据的 CRC32 校验功能
ip_checksum	IP 报头校验和计算模块，该模块通过 IP 首部校验和算法对 IP 的头部进行计算，并得出一个数值，该值用来检测数据是否出错，如果计算的结果与 IP 数据报本身包含的报头校验和不相等，则数据在传输过程中发生了错误，当被舍弃。
eth_udp_tx_gmii	UDP 协议发送模块，该模块将从 FIFO 中读取到的内容经过 UDP、IP、MAC 层协议层层打包后，通过 GMII 接口输出给以太网 PHY 芯片，以完成数据的发送。
gmii_to_rgmii	gmii 接口到 rgmii 接口转换模块，实现 gmii 到 rgmii 的转换
eth_udp_loopback_gmii	整个工程的设计顶层文件，例化了 UDP 接收和 UDP 发送模块, 实现完整的回环测试功能。

45.2.1.2UDP RGMII 回环测试工程实操

1. 使用网线将高云开发板上的以太网接口和当前调试测试工程用的 PC 机的网口连接起来。使用下载线连接开发板调试接口与电脑主机的 USB 接口，如下图 45-3 所示。连接完成后为开发板供电。

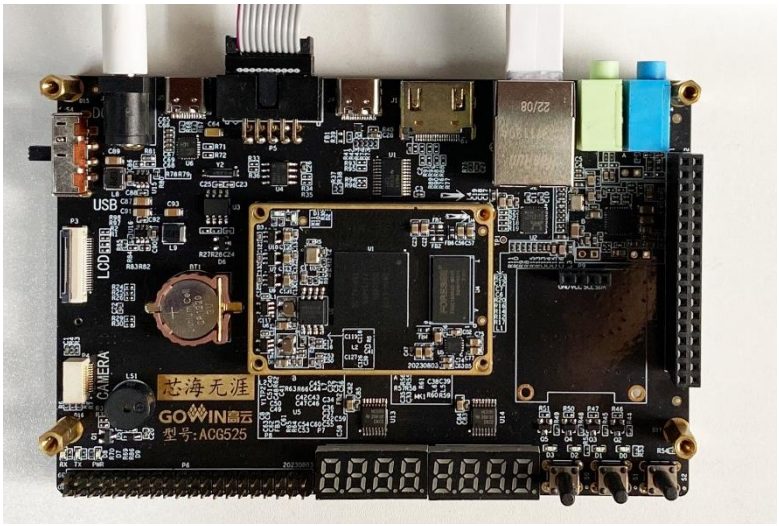


图 45-3 硬件连接

2. 双击 eth_udp_lookback_rgmii. gprj 以打开工程。

3. 下载数据流文件到开发板中，注意这一步一定要先于下面的步骤执行，否则以下操作无法正常进行。

4. 在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态。需要看到在活动网络中有以太网连接存在，才表明开发板和电脑的网络才已经连通。此时如果重新下载数据流文件到开发板中，会发现此本地连接会先消失，然后再重新出现。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意，如下图 45-4 所示。

图 45-4 查看网络连接状态

5. 点击“以太网”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如图 45-5 所示。

图 45-5 查看网络连接状态

6. 在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址（默认网关可以不设置）。如图 45-6 所示。

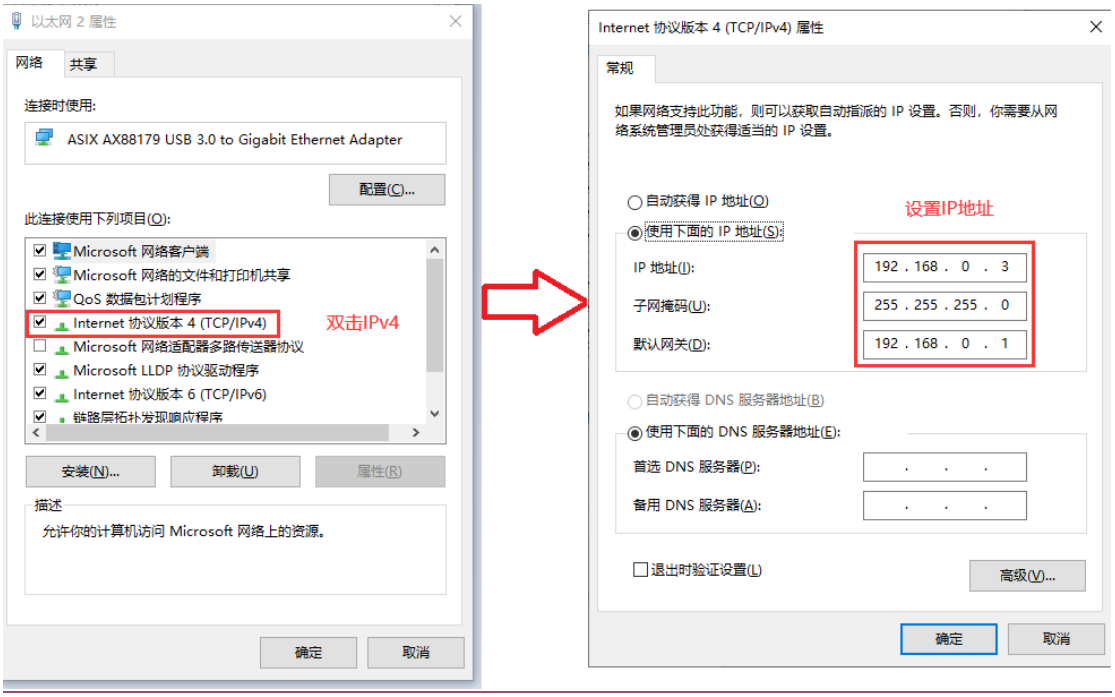


图 45-6 修改 PC 的 IP 地址

如果用户电脑中相关驱动缺失文件，会出现弹窗报错“出现了一个意外的情况，不能完成所有你在设置中所要求的更改。”导致修改失败，对于该情况用户可以参考下帖：

设置静态 IP 时提示“出现了一个意外的情况，不能完成所有你在设置中所要求的更改”

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29229

(出处: 芯路恒电子技术论坛)

7. 由于本测试工程不支持 ARP 协议，因此我们还需要通过 netsh 方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

操作时先以管理员身份运行 cmd.exe 程序（该文件在 C:\Windows\System32 路径下），也就是我们常说的命令行窗口。由于有用户反应在使用时无法成功绑定 arp，经过分析就是操作权限不够，所以这里强调要以管理员身份运行 cmd.exe。

打开 cmd 窗口之后，依次执行如下命令：

（1）查看电脑网卡名称，有线网卡的大概率为“以太网”

```
netsh interface IPV4 show interface
```

图 45-7 查看电脑的网卡名称

（2）根据自己的网卡名称，笔者电脑网卡名称为“以太网”，用下述命令执行静态绑定，如果网卡名称不同，需要自行修改命令然后执行。

```
netsh -c interface ipv4 add neighbors "以太网" "192.168.0.2" "00-0a-35-01-fe-c0"
```

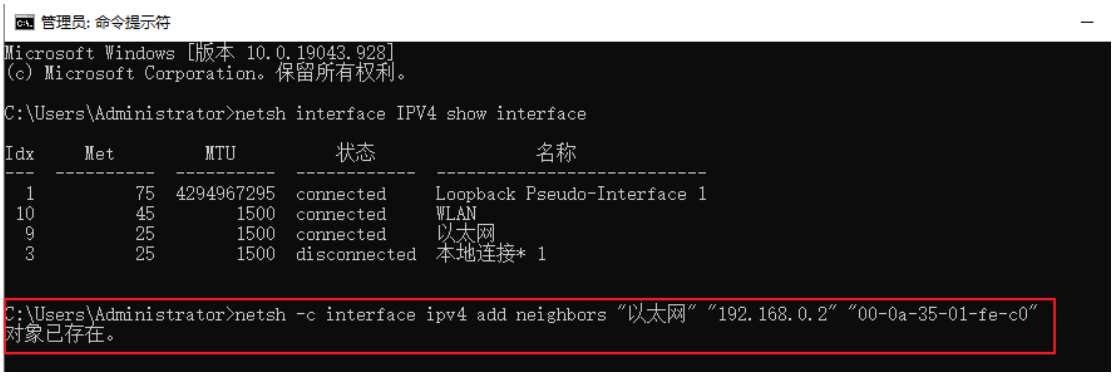


图 45-8 静态绑定

（3）最后用查看绑定结果，执行如下命令。

```
arp -a
```

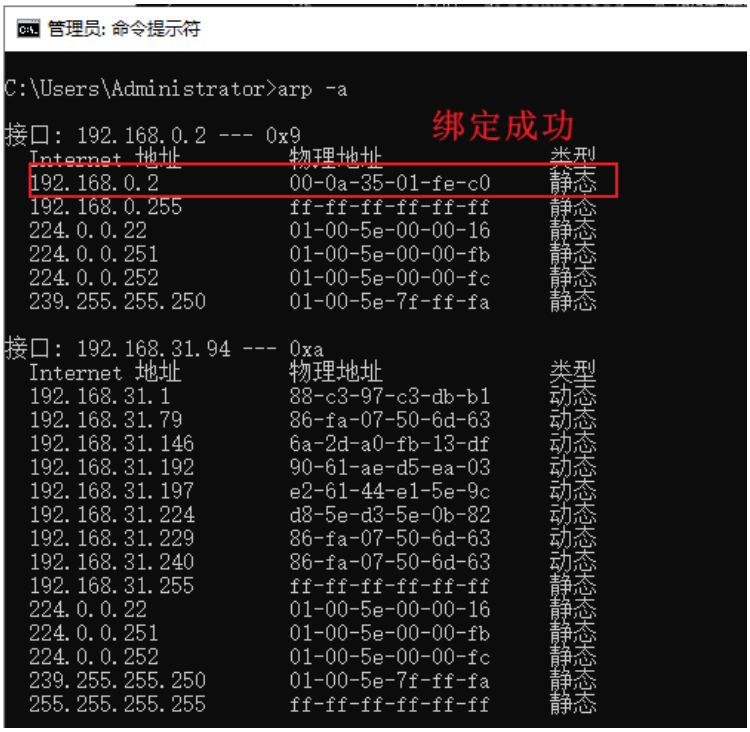


图 45-9 查看绑定结果

8. 打开网络调试助手（NetAssist.exe）并按照如下所述设置各项参数，参考如下图 45-10 所示。

- 选择协议类型为 UDP。
- 设置本地 IP 地址为 192.168.0.3。
- 设置本地端口号为 6102（这里可以设置任意值，推荐 6102）。
- 点击【连接】按钮以创建连接，连接上后该按钮为红色“断开”字样。

- 连接上后，设置目标主机为 192.168.0.2，目标端口 5000。
- 在文本框中输入希望发送的文本内容。
- 点击发送按钮，即可在上方的接收窗口中显示 FPGA 发回的数据，与发送的内容一致。
- 在下方的计数器中，检查发送的数据个数和接收的数据个数是否一致。

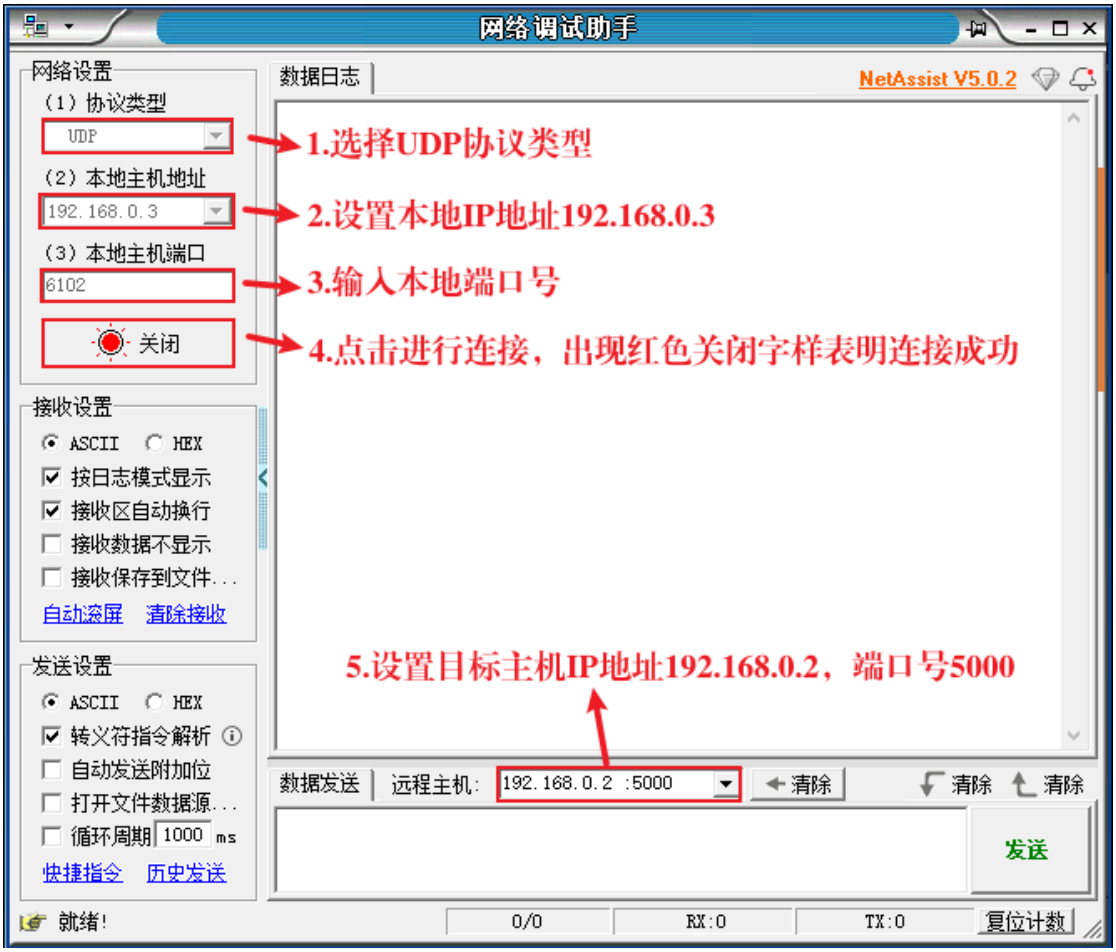


图 45-10 设置相关参数

9. 在发送窗口中输入你希望发送的数据，然后点击发送，则上方的接收窗口回接收到与发送框完全一样的内容，每点击一次发送，就会收到一次数据。在下方的计数器中可以查看发送和接收数据个数是否一致，如下图 45-11 所示。

图 45-11 以太网回环测试

10. 安装网络抓包工具 Wireshark，我们在实验的时候可以用这工具来查看 PC 网口发送的数据和接送的数据。

11. 打开安装好的 Wireshark 抓包工具。在软件界面选择您 PC 的有线网卡。按开始按钮开始抓包，如图 45-12 所示。（不同的版本界面有所差异，我们提供的是 Wireshark_win64_V1.12.4_setup.1427187922.exe）。

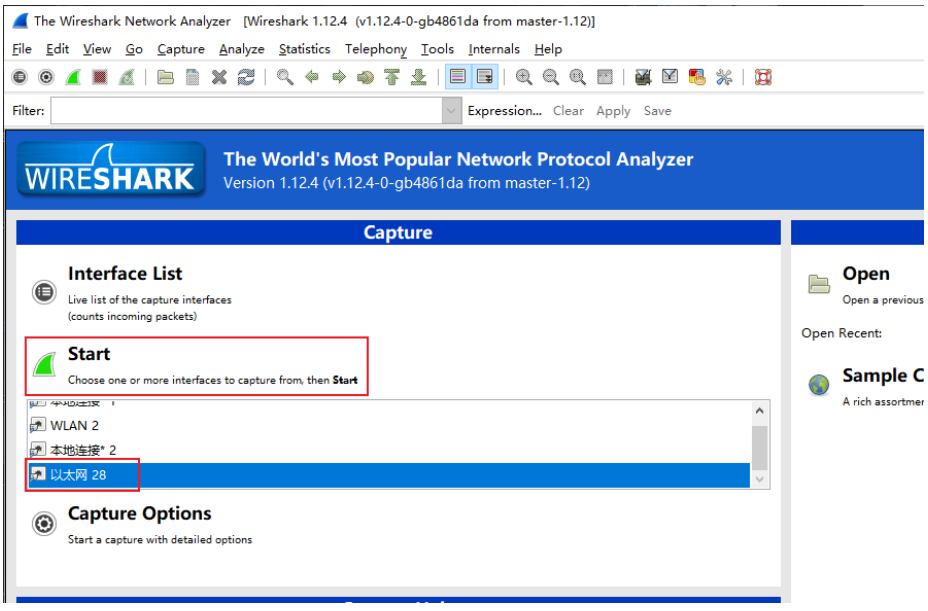


图 45-12 wireshark 主界面

在 wireshark 抓包窗口我们可以看到开发板（192.168.0.2）与 PC 网口（192.168.0.3）间传输的数据包，如图 45-13 所示。

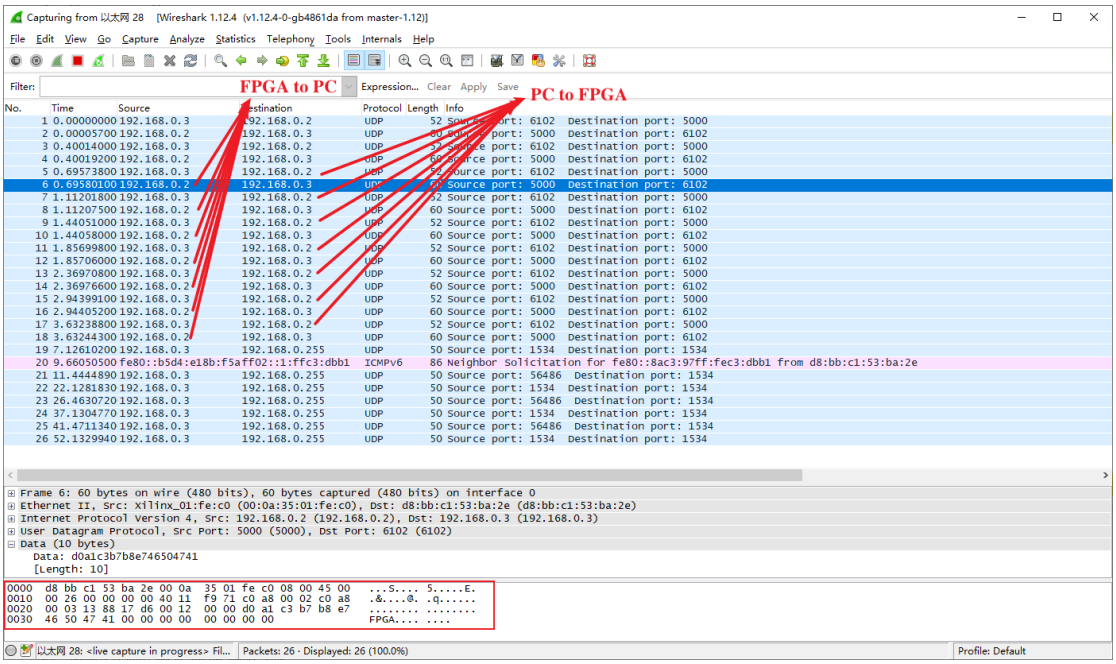


图 45-13 wireshark 抓包

可以看到每次开发板在接收到来自电脑的 UDP 协议数据后，马上会通过 UDP 协议将数据回传给 UDP，对应的网络调试助手中就是每次电脑发送“小梅哥 FPGA”后，立马就能接收来自 FPGA 的回传。至此，本次设计完成。在本例中介绍的设置 IP 地址、绑定 ARP 地址、wireshark 抓包、以太网调试软件的使用等内容，在后续的实验中都有可能需要，请大家留意。

45.2.2以太网图像发送 PC 接收显示实验测试

通过 FPGA 实现以太网的一个主要目的就是使用以太网将 FPGA 采集到的各种数据发送到 PC，如高速 ADC 实时采集数据， 或者图像传感器采集的图像数据。此类应用传输过程中数据量大，实时性要求较高。 使用基于 CPU 架构实现的软件以太网协议栈，受限于 CPU 计算性能和数据组包的规律，一般效率都无法做到很高。而使用 FPGA 实现以太网传输，则可以提供稳定且高效的传输能力。

无论是 ADC 实时采集的模拟数据，还是图像传感器或其他传感器采集的数据，其数据特点都是高速且高实时性。因此本节将以一个以太网图像传输实例为例，介绍使用以太网传输图像到 PC 并显示的方式。

45.2.2.1以太网图像传输工程介绍

基于以太网的图像传输工程，使用 UDP 协议，将 OV5640 摄像头采集到的图像数据经由以太网传输到 PC 端，再由 PC 端的图像显示软件接收图像数据并绘制在屏幕上，以还原图像内容。整个系统框图如下图 45-14 所示：

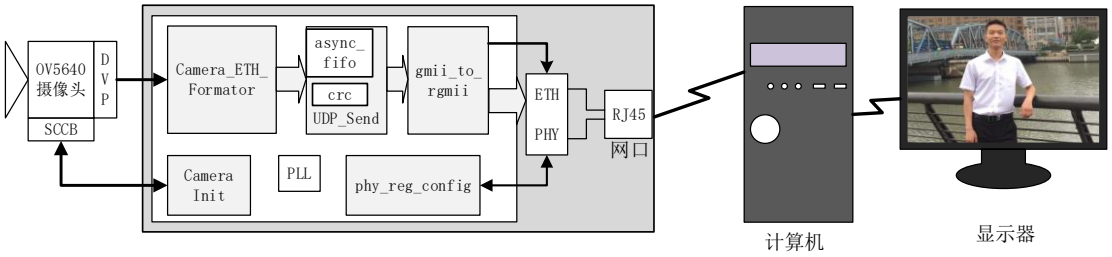


图 45-14 以太网图像传输系统

本例程在小梅哥团队出品的高云开发板上使用 verilog 实现以太网 UDP 协议通信。FPGA 采集 OV5640 输出的 RGB565 格式的图像数据，并将采集到的图像数据使用一定的编号方式编号后以行为单位，使用 UDP 协议经由以太网发送到 PC 机。PC 机上使用我们专门开发的显示软件（小梅哥 UDP 摄像头 V3.exe）接收 FPGA 发送的编号后的图像数据，解析后还原为图像内容绘制在 PC 机显示屏上。

本例对应的例程工程压缩包名为 ov5640_udp_rgmii_8_8.rar，该文件可在我们提供的配套资料中找到。

解压工程到不含中文或者空格的目录中，打开工程后结构如图 45-15 所示。

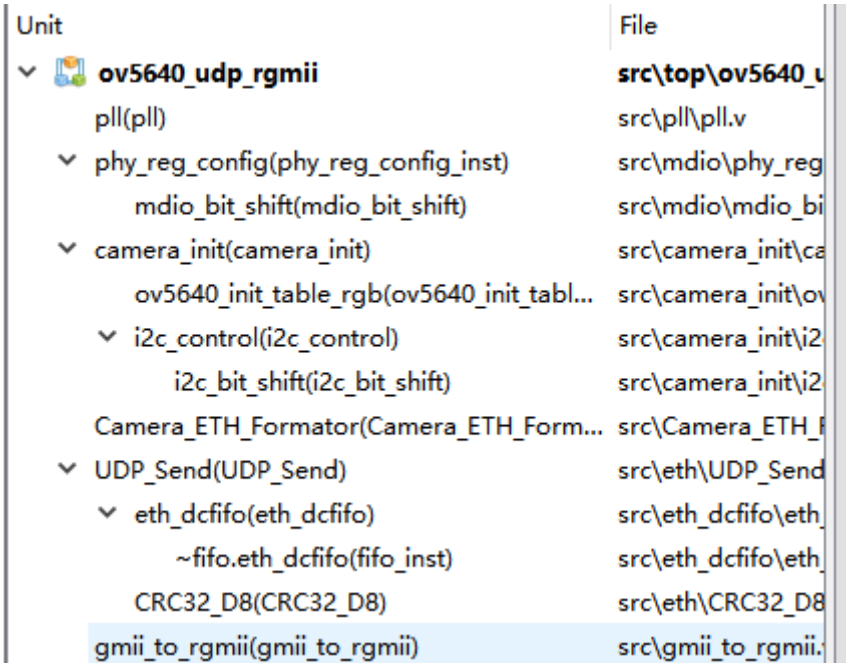


图 45-15 工程结构

这里简单为大家介绍下部分模块的功能：

表 45-2 模块功能介绍

模块名	功能描述
ov5640_udp_rgmii	工程顶层，例化了 UDP 发送和摄像头 DVP 接口图像数据捕获编码模块, 实现完整的图像采集传输功能
pll	锁相环，用于产生各个模块所需工作时钟
phy_reg_config	以太网 PHY 芯片的寄存器配置模块，控制 mdio_bit_shift 模块对以太网 PHY 的指定寄存器进行读写操作以完成以太网 PHY 的工作模式设置。
mdio_bit_shift	MDIO 底层传输协议实现，完成基于 MDIO 协议对 PHY 寄存器的一次读或者写操作
Camera_ETH_Formator	编号模块，对摄像头输出的数据以行为单位进行编号，以便于以太网上位机正确接收图像
UDP_Send	UDP 协议发送模块， 该模块将用户输入的需要发送的数据内容经过 UDP、 IP、 MAC 层协议层层打包后，通过 RGMII 接口输出给以太网 PHY 芯片， 以完成数据的发送。
async_fifo	UDP 发送数据缓存 FIFO，用户将需要发送的数据写入该 FIFO，然后由 UDP_Send 中的发送逻辑将数据读出并最终以 UDP 协议发送。
CRC32_D8	CRC32 校验逻辑， 计算经由 MAC 层发送的数据的 CRC 值并附加在 MAC 结尾发送。
gmii_to_rgmii	gmii 接口到 rgmii 接口转换模块，实现 gmii 到 rgmii 的转换

45.2.2.2以太网图像传输工程实操

1. 使用网线将高云开发板上的以太网接口和您当前调试测试工程用的 PC 机的网口连接起来。连接好下载线缆并给开发板上电。
2. 将 OV5640 摄像头插入开发板的摄像头接口中。为开发板上电，如图 45-16 所示。

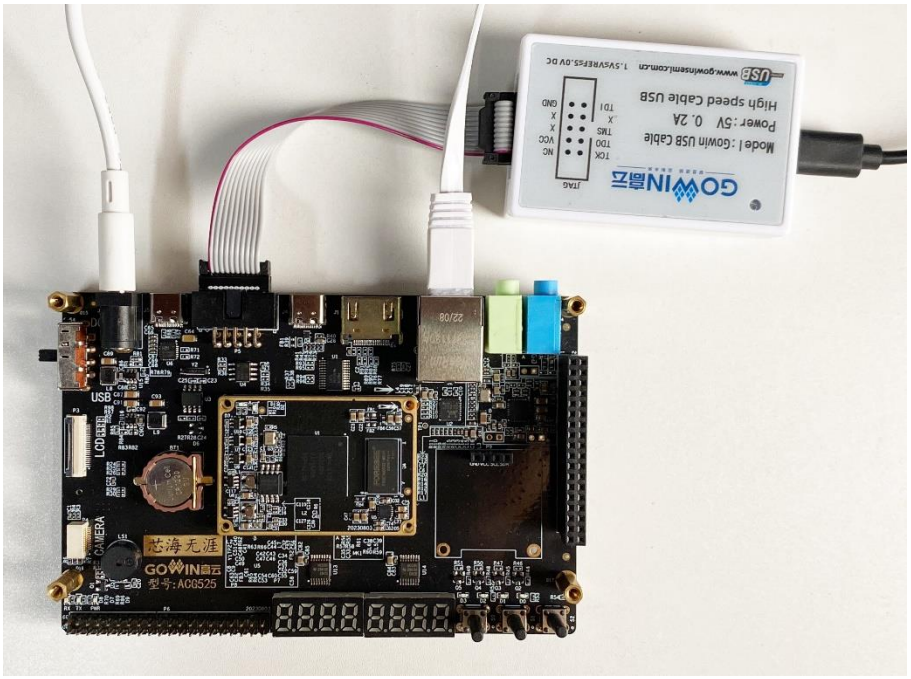


图 45-16 硬件连接图

3. 打开工程，并将 bit 文件烧录到开发板中。注意这一步一定要先于下面的步骤执行，否则以下操作无法正常进行。

4. 在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态。需要看到在活动网络中有以太网连接存在，才表明开发板和电脑的网络才已经连通。此时如果重新下载文件到开发板中，会发现此本地连接会先消失，然后再重新出现。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意，如图 45-17 所示。

图 45-17 查看网络连接状态

5. 点击“以太网”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000Mbps），如图 45-18 所示。

图 45-18 查看网络连接速度

在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4（TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址（默认网关可以不设置）。如图 45-19 所示。

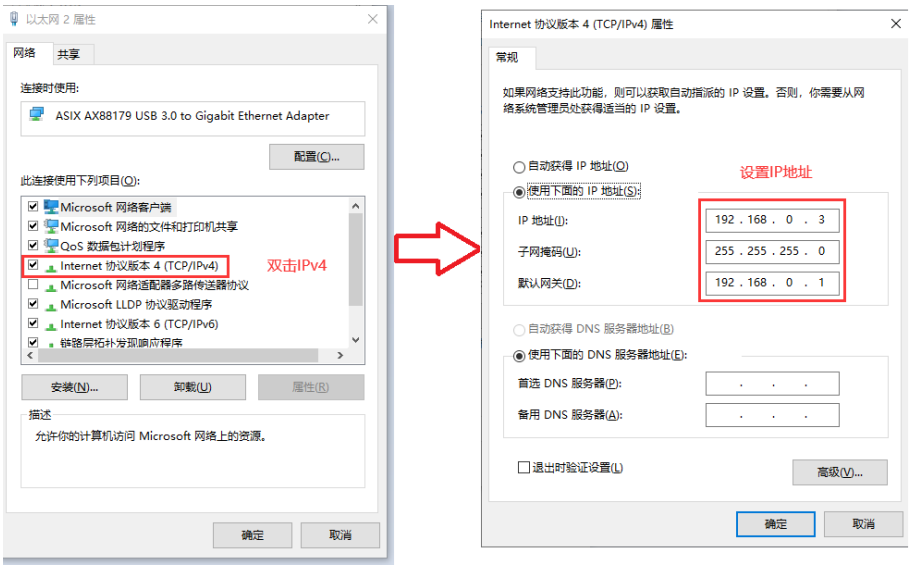


图 45-19 修改 PC 的 IP 地址

如果用户电脑中相关驱动缺失文件，会出现弹窗报错“出现了一个意外的情况，不能完成所有你在设置中所要求的更改。”导致修改失败，对于该情况用户可以参考下帖：

设置静态 IP 时提示“出现了一个意外的情况，不能完成所有你在设置中所要求的更改”

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29229

(出处: 芯路恒电子技术论坛)

6. 开启电脑巨型帧。在“设备管理器\网络适配器\Realtek PCIe GbE Family Controller 属性\高级\巨型帧”，如图 45-20 所示。

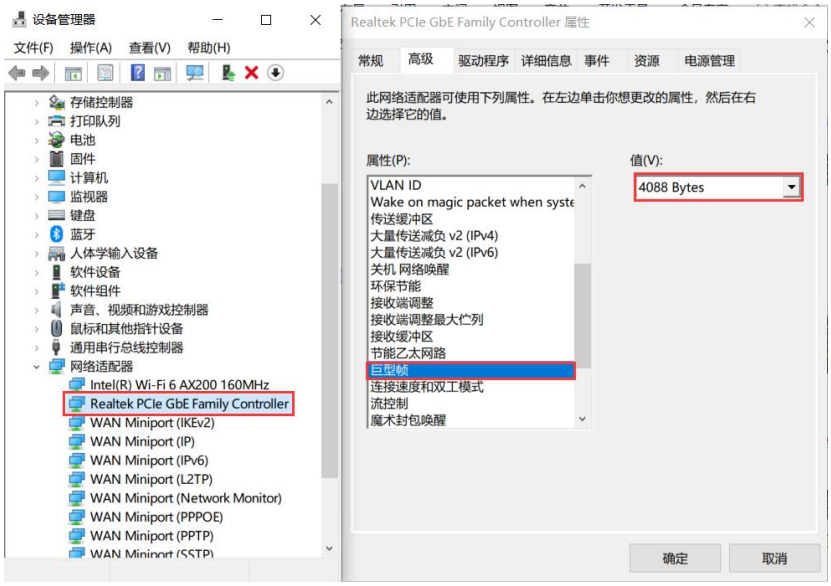


图 45-20 开启巨型帧

7. 由于本测试工程不支持 ARP 协议，因此我们还需要通过 netsh 方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

操作时先以管理员身份运行 cmd.exe 程序（该文件在 C:\Windows\System32 路径下），也就是我们常说的命令行窗口。由于有用户反应在使用时无法成功绑定 arp，经过分析就是操作权限不够，所以这里强调要以管理员身份运行 cmd.exe。

打开 cmd 窗口之后，依次执行如下命令：

（1）查看电脑网卡名称，有线网卡的大概率为“以太网”

```
netsh interface IPV4 show interface
```

图 45-21 查看电脑的网卡名称

（2）根据自己的网卡名称，笔者电脑网卡名称为“以太网”，用下述命令执行静态绑定，如果网卡名称不同，需要自行修改命令然后执行。

```
netsh -c interface ipv4 add neighbors "以太网" "192.168.0.2" "00-0a-35-01-fe-c0"
```

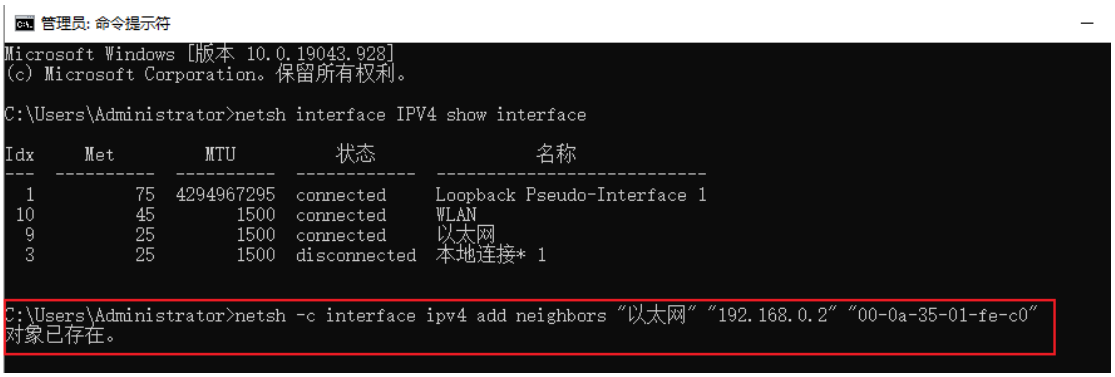


图 45-22 静态绑定

（3）最后用查看绑定结果，执行如下命令。

```
arp -a
```

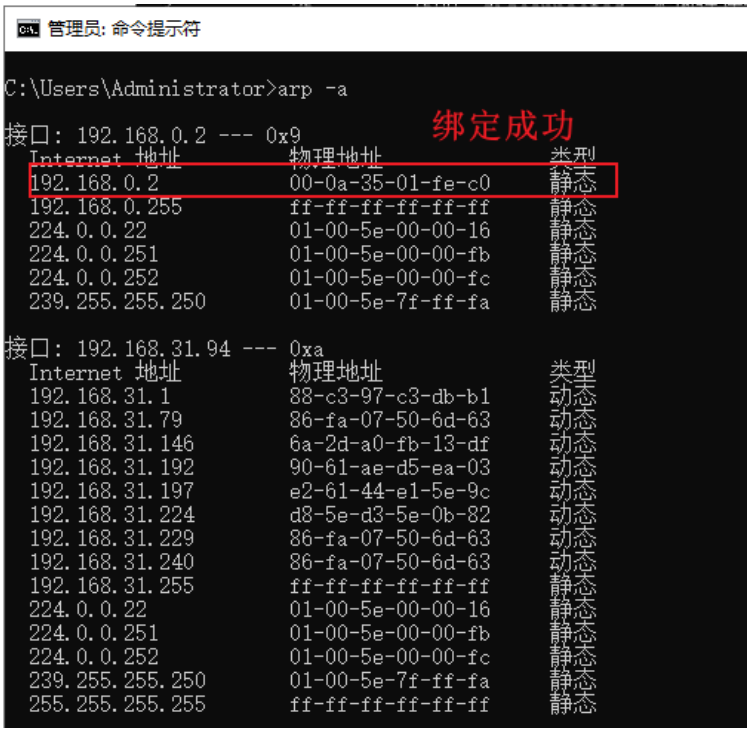


图 45-23 查看绑定结果

8. 打开网络调试助手（小梅哥 UDP 摄像头.exe）并按照如下所述设置各项参数。

图 45-24 连接网络调试助手

- 本机 IP，对应当前实验所用的电脑 IP 地址，前面已经按照要求设置了 192.168.0.3，所以这里直接填写该地址。
- 本机端口，对应当前实验所用电脑接收图像时使用的网络端口，该端口在 FPGA 程序中定义了 6102，所以这里也要设置为 6102 才能正确显

示图像。需要解释的时，早期我们默认使用的是 6000 这个端口号，但是发现该端口号与 National Instruments 的软件有冲突，装有 NI 软件的客户无法连接上，所以特修改为 6102 端口号。

- 远程 IP，对应 FPGA 开发板使用的 IP 地址，也就是代码中 local_ip 值。例程使用的为 192.168.0.2。
- 远程端口，对应 FPGA 开发板使用的端口号，也就是代码中的 local_port 值，例程使用的为 5000。
- 分辨率，对应了摄像头输出的图像分辨率大小，本实验中，千兆 RGMII 接口使用的分辨率为 1280*720。
- 保存图片，该按钮在软件停止接收图像后，点击可以保存最后界面上显示的图像内容到文件。
- 连接/停止，该按钮可以开始和停止接收并显示图像，当所有参数设置好之后，点击按钮即可开始接收并显示图像。

9. 设置好参数后，点击连接按钮即可开始接收并显示图像内容。

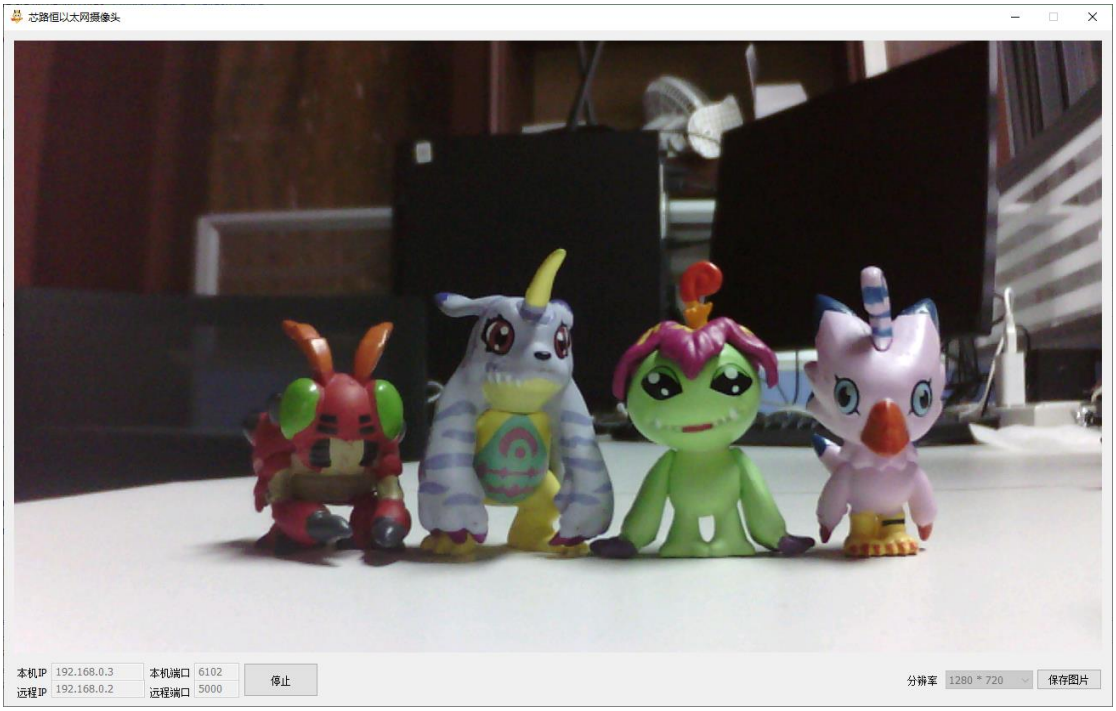


图 45-25 显示效果

10. 该软件运行时需要关闭防火墙，软件自带关闭防火墙功能，点击连接时会提示是否关闭防火墙，勾选专用网络和公共网络两个选项，然后点击【允许访问按钮】即可。如果还是无法关闭，考虑软件权限不够，

以管理员身份运行本软件即可。

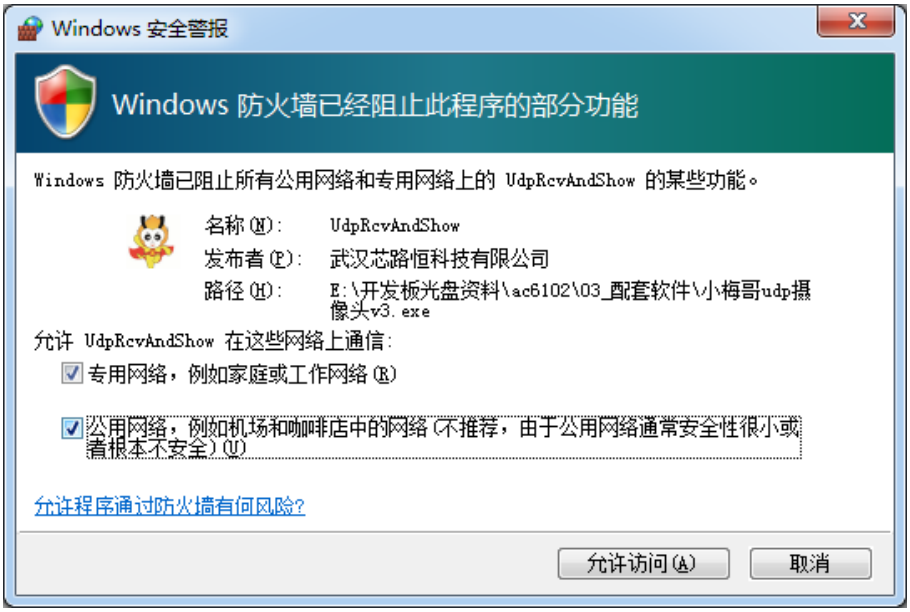


图 45-26 关闭防火墙

至此，则完成了基于 FPGA 的千兆以太网 RGMII 接口的 UDP 协议图像发送到 PC 显示功能实验。本实验操作简单，趣味性和实用性都较强，适合大家用来学习练习。

45.3总结

本章通过两个以太网应用实验，以最直观的角度带大家了解以太网通信。由于以太网通信所包含内容过于繁杂，所以建议读者不必急于了解相关原理。读者可以先跟随本章内容，体验完整应用，再随后续章节由浅入深，逐步下挖底层原理实现。

46 以太网相关通信接口详解

工程源码	本章节无工程
相关视频课程	
说明	

章节导读

对于以太网来说，其存在的目的是实现不同设备间的联网通信，而实现以太网通信，对于大多数人来说，所熟知的就是一个插接网线的接口，那么这个插接网线的接口背后，又包含着怎样的电路结构呢？一般的数字芯片，只能输出 TTL 电平的信号，那么他们又是怎么最终实现了能够插接网线的以太网接口了，中间用到了怎样的转换电路，这些转换电路又是使用的怎样的数字接口进行数据交互的呢，本章内容将通过详细的图文说明，介绍这些相关的知识点。

46.1以太网 MAC 层接口介绍

对于网口大家相信都不陌生，无论是笔记本、台式机，还是交换机、路由器，上面都有网口，事实上，从专业名词的角度来说，这些网口都应称之为 RJ45 接口。

图 46-1 RJ45 接口

但从功能来说，这些网口只是起到一个信号连接的作用，本身无主动通信的能力，一个典型的网络通信电路如下图 46-2 所示。

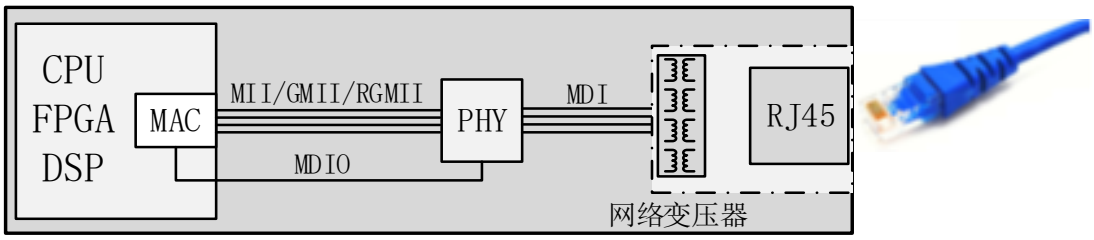


图 46-2 典型网络通信电路

最终送到 RJ45 接口上的电信号要求为高速串行信号，数据是通过串行方式

发送的，千兆以太网的位速率为 1.25Gbps、百兆以太网的位速率为 125Mbps。而典型的处理器如 CPU、FPGA、DSP 一般都不具备直接产生符合以太网物理链路层传输的电信号的能力（高端器件往往集成了高速串行收发器，可以支持，这里不做为讨论），只能产生 TTL 电平标准的并行数据，所以在 RJ45 接口和处理器之间，需要使用一个能够实现 TTL 电平和高速串行数据收发模式之间的转换器，这个转换器就是我们常说的 PHY 芯片。

PHY 芯片的最基本功能就是实现并行以太网数据到符合以太网物理层链路数据传输格式的电平信号之间的转换。在于 RJ45 接口连接的一侧，传输的是经过了 8b/10b(千兆)或 4b/5b（百兆）编码后，由高速串行发送器驱动的高速串行数据信号，而在连接处理器 MAC 的一侧则根据具体的速率和接口模式，提供 GMII、RGMII、MII、RMII 等多种并行接口。

无论是哪种接口，其传输的信号意义都是一样的，都至少包括接收数据信号（RXD[n:0]）、接收数据有效信号(RX_DV)、接收数据时钟信号（RX_CLK）、发送数据信号（TXD[n:0]）、发送数据使能信号（TX_EN）、发送数据时钟信号（TX_CLK）。区别在于接收数据信号和发送数据信号的位宽。以下将分别介绍 MII、GMII、RGMII 这 3 种接口。

46.1.1MII 接口

MII 接口应用于 100Mbps 和 10Mbps 以太网模式下，其接口信号连接关系及各信号的介绍如下图 46-3 所示。

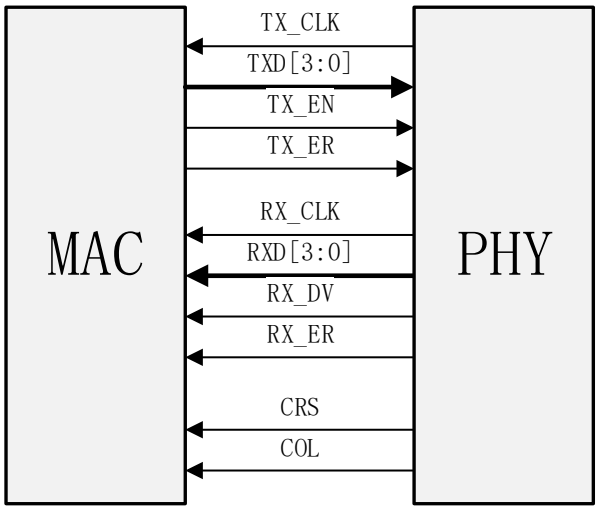


图 46-3 MII 接口连接

表 46-1 MII 接口信号

信号	方向	位宽	说明
TX_ER	O	1	即 Transmit Error，发送数据错误提示信号，同步于 TX_CLK，高电平有效，表示 TX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，TX_ER 不起作用。
TX_EN	O	1	即 Transmit Enable，发送使能信号，只有在 TX_EN 有效期内 TXD 上的数据才会传入 PHY 芯片并进行编码传输。
TX_CLK	I	1	发送参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz。需要注意的是，TX_CLK 时钟的方向是从 PHY 指向 MAC 的，此时钟是由 PHY 提供的。
TXD	O	4	即 Transmit Data，数据发送信号，共 4 根信号线，TX_EN 为高电平的情况下，每个时钟周期传输 4 位数据。
RX_ER	I	1	即 Receive Error，接收数据错误提示信号，同步于 RX_CLK，高电平有效，表示 RX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，RX_ER 不起作用。
RX_DV	I	1	即 Reveive Data Valid，接收数据有效信号，作用类似于发送通道的 TX_EN。
RXD	I	4	即 ReceiveData，数据接收信号，共 4 根信号线。
RX_CLK	I	1	接收数据参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz。RX_CLK 也是由 PHY 侧提供的。
CRS	I	1	即 Carrier Sense，载波侦测信号，不需要同步于参考时钟，只要有数据传输，CRS 就有效。需要注意的是 CRS 只有 PHY 在半双工模式下有效。
COL	I	1	即 Collision Detectd，冲突检测信号，不需要同步于参考时钟。需要注意的是 CRS 只有 PHY 在半双工模式下有效。

其中，MAC 侧向 PHY 侧传输数据的时序图如下图 46-4，参数 t1~t5 是 PHY 芯片能正常接收到数据需要满足的时序参数，VIL 和 VIH 是高低电平的电压标准，电压低于 VIL 表示低电平，电压高于 VIH 表示高电平，详细参数参见具体的 PHY 芯片手册。从波形图可以看出，发送数据信号 TXD 和发送使能信号需要在时钟上升沿保持稳定。

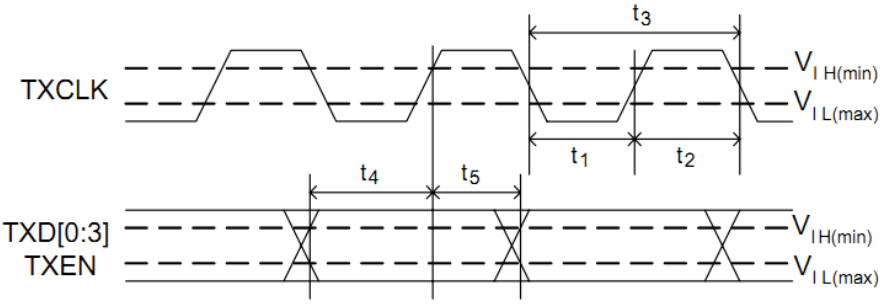


图 46-4 MII 接口 MAC 侧向 PHY 侧传输数据时序

PHY 侧向 MAC 侧传输数据（也就是 MAC 侧接收 PHY 侧传过来的数据）的时序图如下图 46-5，参数 t1~t5 是 PHY 芯片输出数据和与时钟之间的时序参数，VIL 和 VIH 是高低电平的电压标准，电压低于 VIL 表示低电平，电压高于

VIH 表示高电平，详细参数参见具体的 PHY 芯片手册。从波形图可以看出，PHY 芯片传出（也就是 MAC 接收）数据信号 RXD 和数据有效信号在时钟上升沿保持稳定。在 MAC 接收数据时，需要根据传过来数据信号的时序特点进行正确的接收。

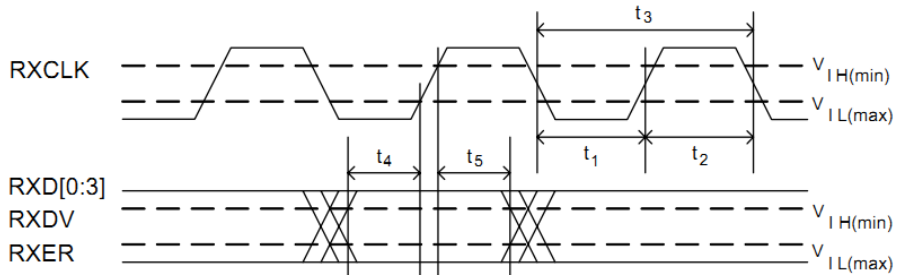


图 46-5 MII 接口 PHY 侧向 MAC 侧传输数据时序

46.1.2GMII 接口

GMII 接口应用于 1000Mbps 以太网模式下，其接口信号连接关系及各信号的介绍如下。

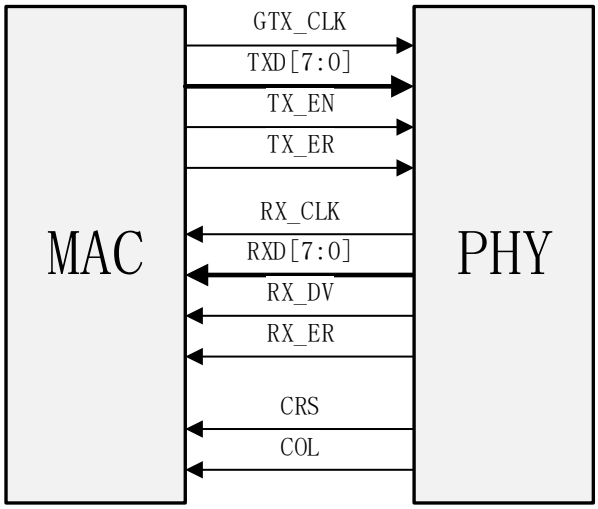


图 46-6 GMII 接口连接

表 46-2 GMII 接口信号

信号	方向	位宽	说明
TX_ER	O	1	即 Transmit Error，发送数据错误提示信号，同步于 GTX_CLK，高电平有效，表示 TX_ER 有效期内传输的数据无效。
TX_EN	O	1	即 Transmit Enable，发送使能信号，只有在 TX_EN 有效期内传的数据才有效。
GTX_CLK	O	1	发送参考时钟，时钟频率为 125MHz。需要注意的是，GTX_CLK 时钟的方向是从 MAC 侧指向 PHY 侧的，此时钟是由 MAC 提供的，这里与 MII 接口有所差别的地方。

TXD	O	8	即 Transmit Data，数据发送信号，共 8 根信号线
RX_ER	I	1	即 Receive Error，接收数据错误提示信号，同步于 RX_CLK，高电平有效，表示 RX_ER 有效期内传输的数据无效。
RX_DV	I	1	即 Reveive Data Valid，接收数据有效信号，作用类似于发送通道的 TX_EN。
RXD	I	8	即 ReceiveData，数据接收信号，共 8 根信号线。
RX_CLK	I	1	接收数据参考时钟，时钟频率为 125MHz。RX_CLK 是由 PHY 侧提供的。
CRS	I	1	即 Carrier Sense，载波侦测信号，不需要同步于参考时钟，只要有数据传输，CRS 就有效。需要注意的是 CRS 只有 PHY 在半双工模式下有效。
COL	I	1	即 Collision Detectd，冲突检测信号，不需要同步于参考时钟。需要注意的是 CRS 只有 PHY 在半双工模式下有效。

与 MII 接口类似，MAC 侧与 PHY 侧之间传输数据的时序图如下图 46-7 所示，图中的参数是 PHY 芯片能正常接收到数据和 PHY 发出数据需要满足的时序参数，VIL 和 VIH 是高低电平的电压标准，电压低于 VIL 表示低电平，电压高于 VIH 表示高电平，详细参数参见具体的 PHY 芯片手册。从波形图可以看出，发送数据信号 TXD 和发送使能信号需要在时钟 GTX_CLK 的上升沿保持稳定；同样的 PHY 芯片传出（也就是 MAC 接收）数据信号 RXD 和数据有效信号在时钟 RX_CLK 上升沿保持稳定。在 MAC 发送/接收数据时，需要满足这些时序要求才能让 PHY 正确收到数据和正确接收到 PHY 传过来数据。

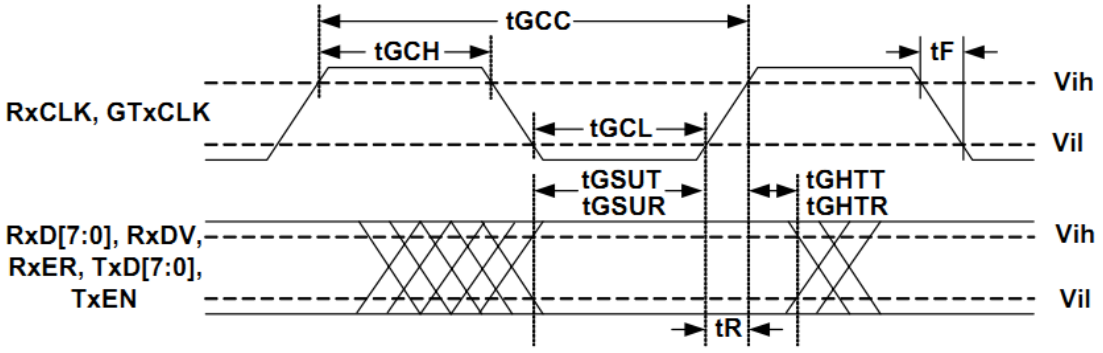


图 46-7 GMII 接口 MAC 侧与 PHY 侧数据传输时序

46.1.3RGMII 接口

RGMII 即 ReducedGMII，是 GMII 的简化版本，将接口信号线数量从 24 根减少到 14 根，时钟频率仍旧为 125MHz，TX/RX 数据宽度从 8 位为变为 4 位。RGMII 接口信号连接关系及各信号的介绍如下。

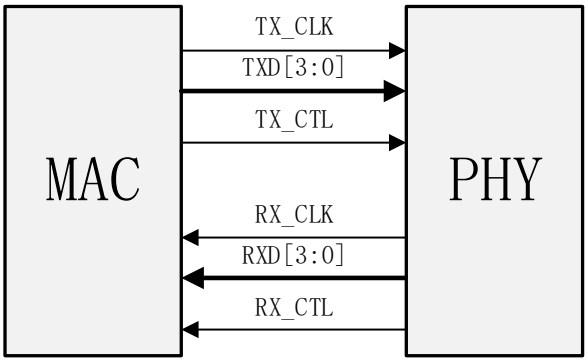


图 46-8 RGMII 接口连接

表 46-3 RGMII 接口信号

信号	方向	位宽	说明
TX_CTL	O	1	该信号线上传送 GMII 接口中的 TX_EN 和 TX_ER 两种信息，在 TX_CLK 的上升沿发送 TX_EN，下降沿发送 TX_ER。
TX_CLK	O	1	发送参考时钟，1000Mbps 速率下，时钟频率为 125MHz，TX_CLK 时钟是由 MAC 提供的，与 GMII 接口中方向一致。
TXD	O	4	在时钟 TX_CLK 的上升沿发送 GMII 接口中的 TXD[3:0]，在时钟 TX_CLK 的下降沿发送 GMII 接口中的 TXD[7:4]。
RX_CTL	I	1	该信号线上传送 GMII 接口中的 RX_DV 和 RX_ER 两种信息，在 RX_CLK 的上升沿传输 RX_DV，下降沿传输 RX_ER。
RXD	I	4	时钟 RX_CLK 的上升沿传输 GMII 接口中的 RXD[3:0]，在时钟 RX_CLK 的下降沿发送 GMII 接口中的 RXD[7:4]。
RX_CLK	I	1	接收数据参考时钟，1000Mbps 速率下，时钟频率为 125MHz。RX_CLK 由 PHY 侧提供的。

RGMII 接口为了保持 1000Mbps 的传输速率不变，RGMII 接口在时钟的上升沿和下降沿都采样数据。在参考时钟的上升沿发送 GMII 接口中的 TXD[3:0]/RXD[3:0]，在参考时钟的下降沿发送 GMII 接口中的 TXD[7:4]/RXD[7:4]。RGMII 同时也兼容 100Mbps 和 10Mbps 两种速率，此时参考时钟速率分别为 25MHz 和 2.5MHz。TX_CTL 信号线上传送 TX_EN 和 TX_ER 两种信息，在 TX_CLK 的上升沿，下降沿发送 TX_ER；同样的，RX_CTL 信号线上也传送 RX_DV 和 RX_ER 两种信息，在 RX_CLK 的上升沿传输 RX_DV，下降沿传输 RX_ER。具体时序如下图 46-9 所示。关于 RTL8211 PHY 芯片具体时序参数与配置的 TXDLY 和 RXDLY 管脚的电平有关，详细可参见 RTL8211 芯片手册。

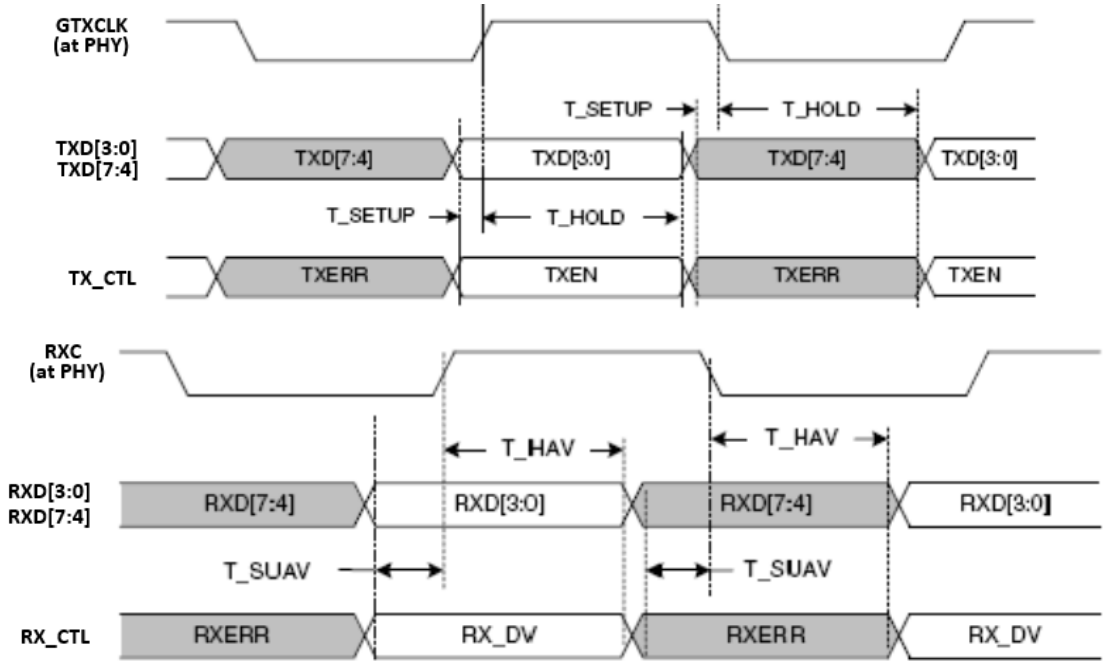


图 46-9 RGMII 接口时序

46.1.4MII、GMII、RGMII 接口对比

上述分别介绍了 MII、GMII、RGMII 接口，接下来对比分析这三种接口的特点。

表 46-4 MII、GMII、RGMII 接口对比

	MII	GMII	RGMII
以太网速率	100Mbps/10Mbps	1000Mbps	1000M/bps
数据线	收发各 4 位	收发各 8 位	收发各 4 位
信号是否时钟双沿传输	单沿传输（SDR）	单沿传输（SDR）	双沿传输（DDR）
TX_CLK 方向	PHY -> MAC	MAC -> PHY	MAC -> PHY
TX_ER	有	有	编码进了 TX_EN
RX_ER	有	有	编码进了 RX_EN
COL、CRS 信号	有	有	无

几个要点说明：

1. MII 接口时，TX_CLK 由 PHY 芯片提供给 MAC，其他接口模式下均由 MAC 提供给 PHY 芯片。
2. 除 GMII 接口外，其他接口均使用 4bit 数据线用来收发数据。
3. RGMII 接口的信号传输采用 DDR 接口，也就是在 CLK 的上升沿和下降沿各传输一次数据。
4. RGMII 接口将 TX_ER 和 RX_ER 信号编码进了 TX_CTL 和 RX_CTL 信号中，不再使用独立的信号线。

46.2FPGA 以太网电路介绍

高云开发板通过一片 Realtek 的 RTL8211F-CG 以太网 PHY 提供对以太网连接的支持，RTL8211F-CG 是一片 10M/100M/1000Mbps 自适应以太网收发器，提供 RGMII 接口的 MAC 连接。在芯路恒科技出品的众多 FPGA 板卡中，都用到了该芯片作为以太网物理层。这里，先介绍完整的 RTL8211F-CG 应用模式的电路设计，再根据具体的板卡应用情况分别介绍。

下图为开发板上的网口示意图。

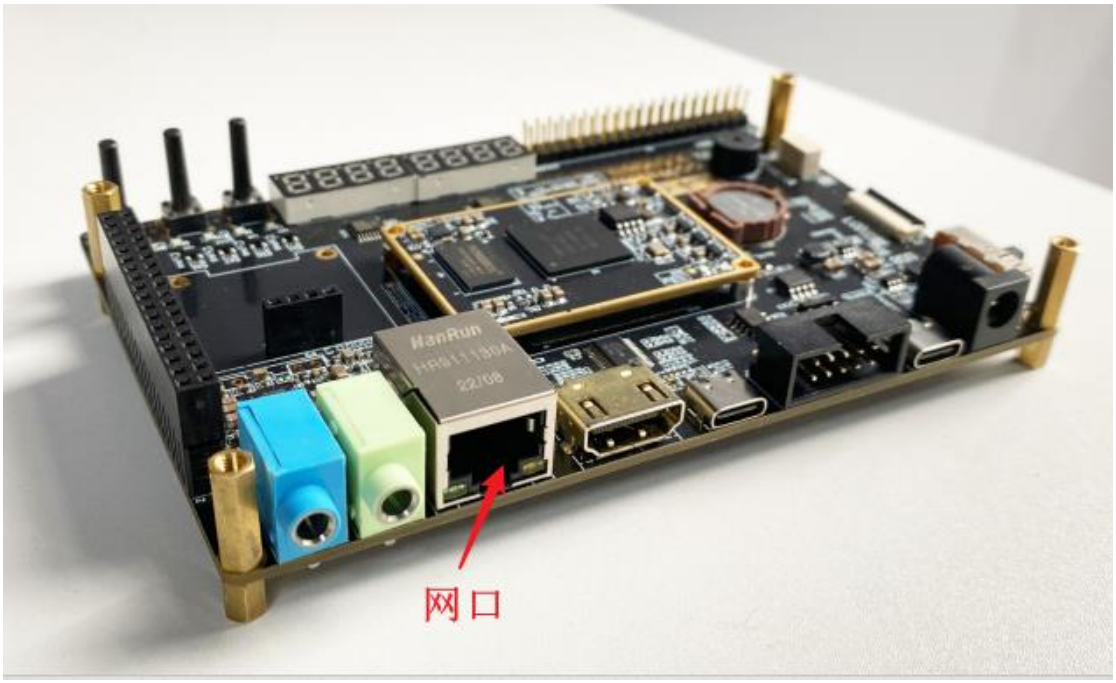


图 46-10 开发板上网口示意图

开发板上提供了一个千兆以太网接口，该接口由千兆以太网 PHY 和网络变压器接口组成。当需要发送以太网数据时，FPGA 把数据发送给 PHY 芯片，PHY 芯片将数据编码后，通过网络变压器将数据加载到网线上。数据经由网络传递到接收方。而远端发送过来的数据，经由网线传递给网络变压器，网络变压器的输出连接到 PHY 芯片上，PHY 芯片对信号进行解码后，得到实际的数据，然后将数据传递给 FPGA 芯片。FPGA 实现千兆以太网数据传输的功能框图如下图 46-11 所示：

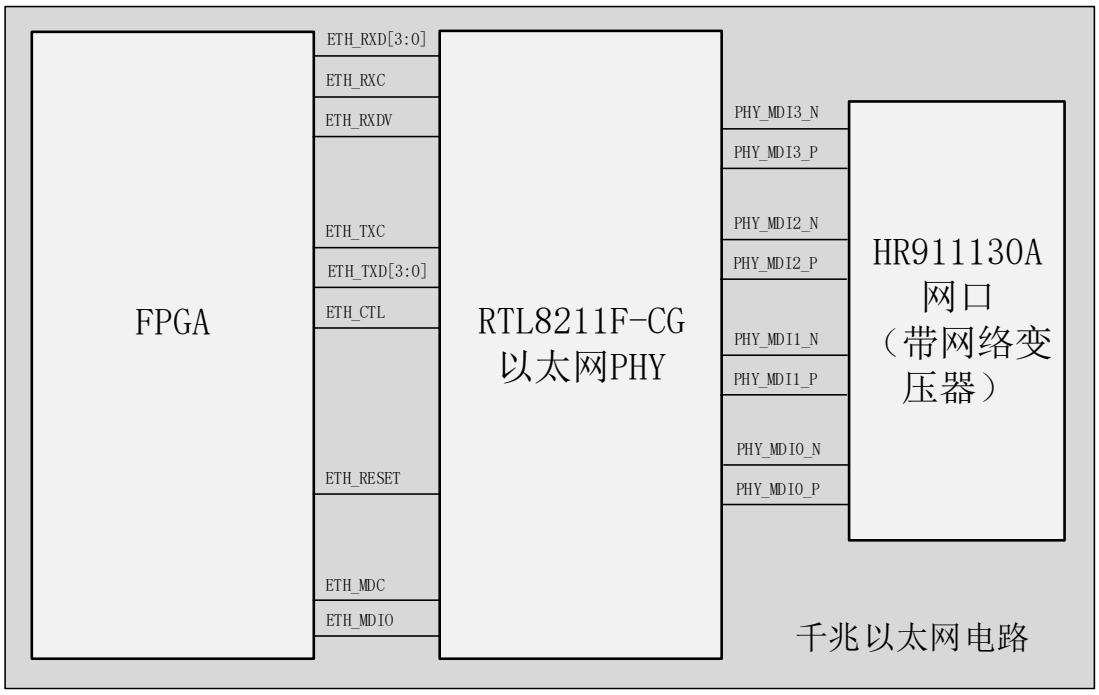


图 46-11 千兆以太网数据传输功能框图

46.2.1PHY 芯片 RTL8211F-CG

高云开发板上使用的是一款功能齐全，性能优异，性价比高的 10/100/1000Mbps 以太网 PHY 芯片——RTL8211F-CG。RTL8211F-CG 是一款支持 RGMII 接口的以太网物理层收发器，能够工作在 10M、100M 或 1000M Base 模式，对 MAC 层可提供 RGMII 接口模式，并提供了标准的 MDIO 管理接口与处理器相连。

注意，大家自己在设计基于这个芯片的电路时需要留意 RTL8211F-CG 芯片的后缀比较重要，该芯片有很多型号，根据尾缀的不同其封装和接口功能也有差异。例如 RTL8211EG 是 64Pin 封装，支持 GMII 接口，而 RTL8211F-CG 则是 40pin 引脚，仅支持 RGMII，不支持 GMII 接口。因此，开发板上只支持 RGMII 接口。

46.2.2RTL8211F-CG 芯片器件地址

以太网物理层芯片都有一个器件地址，该器件地址就是在介绍 PHY 管理接口 MDIO 时所说的 PHY 器件地址，该器件地址用来由 MDIO 主机（如 MAC 或处理器）寻址 MDIO 总线上连接的指定的 PHY 芯片。

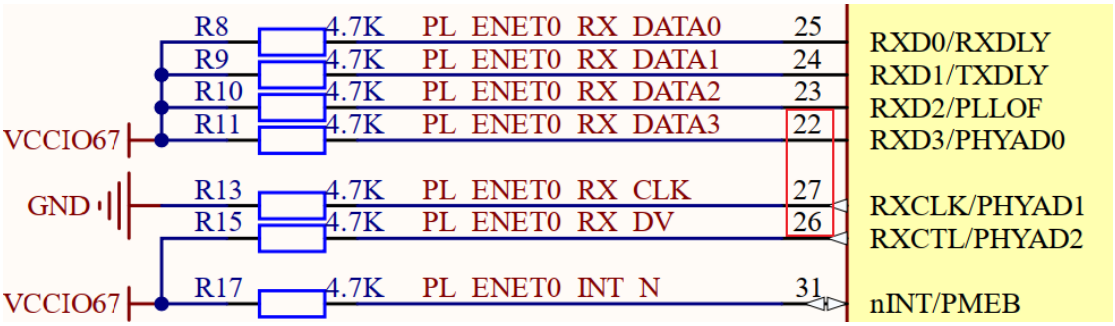
对于 RTL8211F-CG 芯片，该地址分为固定部分和硬件可设定两部分。器件地址共有 5 位，其中高 2 位为固定的 00，低三位通过三个地址设置脚在芯片上

电或复位时候设置， 这三位分别对应 RTL8211F 芯片的 22、27、26 脚。下表 46-5 为这三个脚功能说明。

表 46-5 RTL8211F-CG 芯片地址脚说明表

Pin No.	Pin Name	Description
22	PHYAD0	PHTAD[2:0]，PHY 地址配置引脚
27	PHYAD1	
26	PHYAD2	

在开发板上，这三个脚通过三个电阻分别上/下拉到了指定电平，具体来说，PHY_ADDR0 和 PHY_ADDR2 通过电阻下拉到了高电平，PHY_ADDR1 通过电阻上拉到了低电平，所以高云开发板上的 PHY 器件地址为 00101。



46.2.3PHY 与 FPGA 连接管脚说明

开发板上与 RTL8211F-CG 芯片的引脚连接如下表 46-6 所示：

表 46-6 RTL8211F-CG 芯片的引脚连接

功能标号	FPGA 管脚	功能标号	FPGA 管脚
ETH_TXEN	C2	ETH_RXDV	E1
ETH_TXD3	F1	ETH_RXD3	F5
ETH_TXD2	F2	ETH_RXD2	F6
ETH_TXD1	D1	ETH_RXD1	G1
ETH_TXD0	D2	ETH_RXD0	G3
ETH_GTXC	C1	ETH_RXC	L5
ETH_RESET	G6	ETH_MDIO	E4
		ETH_MDC	D3

其他未列出管脚均为无必要与 FPGA 连接的信号。

47 以太网（MAC）帧协议介绍

工程源码	本章节无工程
相关视频课程	
说明	

章节导读

经过上一章节，我们已经对高云开发板实现以太网通信的相关硬件基础有了一定的了解。本章我们将进一步深入学习，了解以太网协议的底层原理，并基于该原理，带领大家完成以太网收发逻辑的设计。

47.1以太网帧概述

以太网是目前最流行的一种局域网组网技术（其他常见局域网组网技术还有令牌环局域网、无线局域网、ATM 局域网），以太网技术的正式标准是 IEEE 802.3 标准，它规定了在以太网中传输的数据帧结构，如下表 47-1 所示。

表 47-1 以太网数据帧结构说明表

前同步码 Preamble	分隔符 SFD	目的地址 DES_MAC	源地址 SRC_MAC	长度/类型 length/type	数据和填充 Data and Pad	校验字段 FCS
7 字节	1 字节	6 字节	6 字节	2 字节	46~1500 字节	4 字节

每个字段的功能和说明如下表 47-2 所示。

表 47-2 每个字段说明表

字段	字段长度（字节）	说明
前导码（Preamble）	7	7 个字节的 0x55，用于帧的同步
帧开始符（SFD）	1	值为 0xd5，标明下一个字节为目的 MAC 字段
目的 MAC 地址	6	指明该以太网帧的接收方网卡地址
源 MAC 地址	6	标记该以太网帧的发送方网卡的地址
长度/类型 （Length/ Type）	2	当这两个字节的值小于 1518 时，那么它就代表其后数据字段的长度；如果这两个字节的值大于 1518，则表示该以太网帧中的数据属于哪个上层协议（例如 0x800，代表 IP 数据包；0x806，代表 ARP 数据包等。
数据和填充 （Data and Pad）	46~1500	高层的数据，通常为 3 层协议数据单元。对于 UDP、TCP/IP 协议，该部分为 IP 数据包
帧校验序列（FCS）	4	对接收网卡提供判断是否传输错误的一种方法，如果发现错误，丢弃此帧，使用 CRC32 计算方法。

47.2前导码和分隔符

从物理层上看，一个完整的以太网帧有 7 个字段。事实上，前两个字段前导码和分隔符并不能算是真正意义上的以太网数据帧，这 8 个字节的值在任何以太网帧都是一样的，7 字节的前导码值为 0x55，1 字节的分隔符值为 0xD5。前导码存在的意义是为了标记一帧以太网帧即将开始传输，分隔符存在的意义是指示前导码传输完毕，接下来将要传输的就是正式的以太网帧有用的部分。

前导码的值为 7 字节的 0x55，0x55 用二进制表示就是 01010101b，也就是说前导码的意义就是驱动网络链路电平开始交替翻转，以标记帧的开始。而分隔符的值为 0xD5，0xD5 用二进制表示就是 11010101b，由于以太网 PHY 在将数据编码为以太网物理链路层的位传输信号是从低位开始编码的，所以从信号链路的电平来看，就是先连续的产生 62 位 0 和 1 交替变化的电平，最后再以 2 位连续的高电平来标记以太网数据帧部分的到来。

47.3MAC 地址

按照规定，每个连接到互联网的网卡都必须要有一个唯一的物理地址，该地址称为该网卡的 MAC 地址，这是一个 48 位的数据，如同人的身份证一样唯一。数据在网卡之间传递时，就是依靠这个 MAC 值来确定网络上的数据是否是传给该网卡的，如果数据中的 MAC 地址值与该网卡的 MAC 地址值相同，则该网卡会接收网络上的数据内容，否则就会忽略。

事实上，MAC 地址的对应并不是唯一的，在网络传输中，总共有三种对应方式，分别为单播地址、多播地址和广播地址。单播地址通常与一个具体网卡的 MAC 地址相对应，也就是一对一，唯一指定一个数据帧的接收方，它要求第一个字节的 bit0（即最先发出去的位）必须是 0；多播地址要求以太网帧中目的 MAC 地址第一个字节的 bit0 强制设置为 1，这样，在网络中多播地址不会与任何网卡的 MAC 相同，此时网卡依旧会接收该数据帧，然后交由上层 IP 协议来解多播地址，也就是说多播数据可以被很多个网卡同时接收；而广播地址的所有 48 位全为 1（即 FF-FF-FF-FF-FF-FF），同一局域网中的所有网卡可以接收广播数据包。

每个电脑的 MAC 地址可以通过在 cmd 窗口中使用 ipconfig －all 命令查看，其中物理地址值就是该网卡的 MAC 地址，如下图 47-1 所示：

图 47-1 查看网卡 MAC 地址

每个联网设备的 MAC 地址必须是全世界唯一的，为了实现这种唯一性，所有的设备厂商要想让自家产的设备作为标准设备并能够联网，都需要向美国的 IEEE 申请，设备厂商申请的不是一个个具体的 MAC 地址值，而是一个 MAC 地址段。48 位的 MAC 地址由两部分组成，其中前 24 位由 IEEE 批复提供，后 24 位则由设备厂商自行定义。

在我们基于 FPGA 的以太网实验中，关于开发板的 MAC 地址，经常有读者会问这个地址是怎么确定的，这里简单做个解释：

对于需要公开销售连接到互联网上的设备，例如计算机，路由器，必须有全球唯一的 MAC 地址，这个地址需要设备制造商向 IEEE 组织申请并购买。

Altera、Xilinx 等制造官方 FPGA 板卡时，由于板载网口，也都会向 IEEE 组织购买 MAC 地址段。

MAC 地址主要面向直接连接互联网的设备，要求网络上所有设备的 MAC 地址唯一，目的是为了防止冲突。如果设备不接入互联网，只接入局域网，则只要局域网内所有的设备 MAC 地址不一样即可避免冲突。

我们在设计基于 FPGA 的简易系统时，不需要设备接入公共互联网，一般只直接对电脑传输数据，所以从防止冲突的角度考虑，只要不与接入的电脑的 MAC 地址冲突即可，所以我们可以任意找一个 MAC 地址使用。

如果用户使用 FPGA 设计了能够接入公共互联网的设备，则需要自行购买

MAC 地址段并设置。

47.4数据和填充字段

本小节介绍数据和填充字段的内容和其长度限制的由来。

47.4.1数据和填充字段内容

以太网帧的数据和填充字段内容就是希望通过以太网帧传输的“数据”部分，当然，这个数据并不仅仅是用户数据，所谓用户数据，就是我们希望发送的数据内容，如聊天时发送的“你好，很高兴认识你”这句话就是用户数据。以太网帧中的数据部分一般都是另一个上层协议，如 TCP/IP 协议。用户数据都是包含在该上层协议中的。

以太网帧的数据和填充字段要求该段数据长度至少为 46 个字节，而最多不超过 1500 个字节。也就是整个以太网帧的数据长度最小为 64 字节。那么该长度需求的原因是什么呢？为什么至少要 46 个字节，最多不超过 1500 个字节呢？要理解这个要求的由来，还要从以太网数据帧在网络上传输的原理说起。

47.4.246 字节最少数据长度限制

首先来讲，以太网传输是不可靠的，这意味着发送方并不知道接收方有没有收到自己发出的数据包，如果他发出的数据包发生错误，就需要重传。以太网的错误主要是发生碰撞，碰撞是指两台机器同时监听到网络是空闲的，同时发送数据，就会发生碰撞，碰撞对于以太网来说是正常但需要合理应对的。

假设一个局域网内有 A，B，C，D 四台网络设备，示意图如下图 47-2 所示。

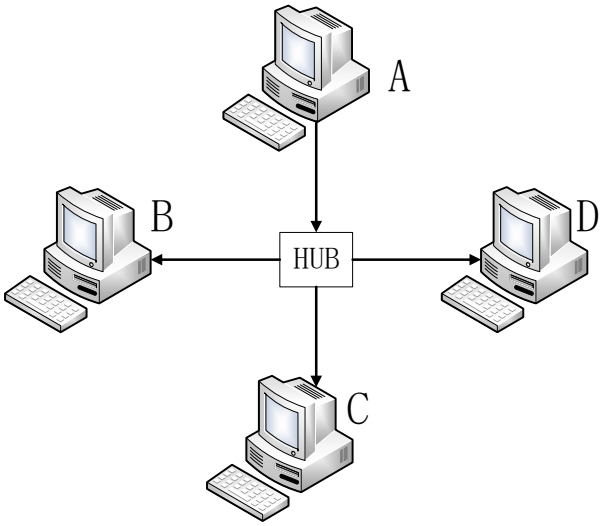


图 47-2 局域网示意图

其中，集线器（HUB）是一种网络共享媒体，所谓网络共享媒体，即在单一时间点内，仅能被一部主机所使用。

以上图中设备 A 向设备 D 发送数据为例说明以太网中网卡之间的数据传输过程。CSMA/CD 搭配上述的环境，它的传输情况需要有以下的流程。

监听媒体使用情况（Carrier Sense）： A 主机在发送网络封包前，需要先对网络媒体进行监听，确认没有其他设备在使用后，才能够发送出讯框。

多点传输 (Multiple Access)：A 主机所送出的数据会被集线器复制多份，然后传送给所有连接到此集线器的其他网络设备。也就是说，A 主机送出的数据，B、C、D 三个计算机都能够接收的到。但由于目标是 D 设备，因此 B 与 C 会将此讯框数据丢弃，而 D 设备则会抓下来处理；

碰撞侦测 (Collision Detection)：该讯框数据附有检测能力，在 A 主机发送给数据过程中，若其他主机例如 B 计算机也刚好在同时间发送讯框数据时，那么 A 与 B 送出的数据碰撞在一块（出车祸），此时这些讯框就会损毁，那么 A 与 B 就会各自随机等待一个时间，然后重新通过第一步再传送一次该讯框数据。

某一时刻，假设 A 检测到网络是空闲的，开始发数据包，尽力传输，当数据包还没有到达 B 时，B 也监测到网络是空闲的，开始发数据包，这时就会发生碰撞，B 首先发现发生碰撞，开始发送碰撞信号，所谓碰撞信号，就是连续的 01010101 或者 10101010，十六进制就是 55 或 AA。这个碰撞信号会返回到 A，如果碰撞信号到达 A 时，A 还没有发完这个数据包，A 就知道这个数据包发生了错误，就会重传这个数据包。但如果碰撞信号返回到 A 时，数据包已经发完，则 A 不会重传这个数据包。

要保证以太网的重传，必须保证 A 收到碰撞信号的时候，数据包没有传完，要实现这一要求，A 和 B 之间的距离很关键，也就是说信号在 A 和 B 之间传输的来回时间必须控制在一定范围之内。

根据 IEEE 定义的这个标准，最小长度是对最初的 10Mb/s 以太网设计的标准。该标准是将一个以太网的最大长度限制在 2500m 情况下定义的（通过 4 个中继器连接的 5 个 500m 的电缆段）。

根据电子在铜缆中传播速度约为 0.77c（231M m/s），可得到 64 字节采用 10Mb/s 时的传输时间为 64*8/10,000,000=51.2us，那么最小尺寸的帧（64 字节）能在电缆中传输约为 231*51.2 ≈ 12000m。如果采用一条最长为 2500m 的电缆，从一个站到另一个站之间的最大往返距离为 5000m。以太网设计者确定最小帧

长度考虑到安全因素（大于 2 倍的 2500m 长度限制），在完全兼容（和很多不兼容）的情况下，一个输出帧的最后位在所需时间后仍处于传输过程中，这个时间是信号到达位于最大距离的接收器并返回的时间。如果这时检测到一个冲突，传输中的站能知道哪个帧发生冲突，即当前正在传输中的那个帧。在这种情况下，该站发送一个干扰信号（高电压）提醒其他站，然后启动一个随机的二进制指数退避过程。

总结：最小数据帧的设计原因和以太网电缆长度（以太网的最大长度限制为 2500m）有关，为的是让两个相距最远的站点能够感知到双方的数据发生了碰撞；最远两端数据的往返时间就是争用期，以太网设计者规定的争用期是 51.2us（10M 以太网传输 64byte 数据的时间，该时间是有考虑到充足的裕量）。

47.4.31500 字节最大数据帧长度限制

要理解以太网帧的数据段长度要求在 1500 字节以内，需要先了解 IP 数据包的传输机制。

以太网 MAC 帧中传输的数据段中，出现频次最高的就是 IP 协议。IP 协议定义一帧 IP 数据帧最大可以为 65536 字节，加上 Ethernet Frame 头和尾，整个以太网帧的长度可以有 65535+14+4=65553 byte。如果在 10Mbps 以太网（数据位速率为 100ns）上，一次性传输 65553 字节（65553*8= 524424bit）的数据，将会占用共享链路超过 50ms（65553*8*100ns=52.4ms）,这将严重影响其它主机的通信，特别是对延迟敏感的应用是无法接受的。由于线路质量差而引起的丢包，发生在大包的概率也比小包概率大得多。

传输大包在丢包率较高的线路上是不明智的。但是如果选择一个比较小的长度，传输效率又不高，拿 TCP 应用来说，如果选择以太网长度为 218byte，TCP payload = 218 - Ethernet Header -IP Header - TCP Header=218-18 - 20 -20= 160 byte 那有效传输效率=160/218= 73% ，而如果以太网长度为 1518，那有效传输效率=1460/1518=96% 通过比较，选择较大的帧长度，有效传输效率更高。而更大的帧长度同时也会造成上述的问题，可能是基于这些考虑，选择了一个折中的 1518 字节。对应的 IP packet 就是 1500 字节。

上述分析是基于最早的以太网是通过 Hub 或集线器来工作的事实，集线器在任意时刻只能有一台主机发送，这种共享方式发送效率很低。现代高速交换机则让每个连接交换机的主机工作在独占模式，带宽独享，可以同时收发。而且现在早已不是早期的 10Mbps 的带宽，而是 1000M、10000M，所以即使发送

大包也不会影响别的主机，影响的只是交换机的接收和发送队列，既然发送大包效率要比小包效率高，而且特定的应用也有发大包的需求，比如 NFS 文件系统，既然物理限定已经一定程度上减弱或消失了，所以增大一次传输的数据帧长度也是合理的了。这是一个好主意，所以现代的网卡、交换机、路由器网络接口等都可以实现更大的 MTU，可以达到>9000 字节的大小，这种远大于标准以太帧尺寸的帧就是我们常说的巨型帧（Jumbo Frame）。

47.5校验序列

上面已经从各种物理角度考虑了以太网帧的各个字段的物理意义，就剩下最后的校验序列了。校验序列又是个什么数据，为什么有这样的一个字段存在呢？

首先来说，以太网通信的时候数据最终都是通过网线从发送方传输到接收方，在这个传输路径中，网线会经过各种情况，遭受各种可能的电磁干扰。万一数据在网线上传输的过程中受到干扰，数据中的某一位或多个数据位因为干扰其电平状态发生了变化，例如 0 变 1、1 变 0，那么接收方在接收到该数据包时，收到的内容和发送方发送的内容就不再一致了，但是接收方本身是无法知道这些数据在传输过程中到底有没有受到干扰，有没有发生变化。这就存在接收数据错误的风险。

为了解决这个问题，以太网 MAC 帧发送方会对该帧发送的所有内容，包括 MAC 帧头（包括目的 MAC 地址、源地址、长度/类型，不含前导码和分隔符）、数据和填充字段采用一种特殊的算法（CRC32）进行计算，得到一个 32 位的数，附加在数据和填充字段的结尾发送。而接收方接收到一帧 MAC 帧时，也会对该帧的 MAC 帧头和数据填充字段使用完全相同的算法进行计算，得到一个 32 位的数，并将这计算得到的 32 位的数据与接收到的数据帧的最后 32 位数据进行比较，若相同则表明接收和发送的数据内容完全一致，若不同则意味着该帧数据中肯定有至少 1 位数据的值在传输过程中发生了变化，整个这一帧内容都会被直接舍弃。

47.6总结

至此，对于以太网 MAC 帧的介绍就结束了。可以看到，以太网 MAC 帧的结构还是非常简单的，重点是理解整个以太网 MAC 帧中各个字段的含义。接下来，就可以编写相应的网络实现协议逻辑了。

在实现网络协议时，最重要的一点就是要了解和掌握使用的接口方式。因为不管编写的协议逻辑如何，最终都是要通过对应的接口输出或者输入。目前，千兆以太网接口大多采用 RGMII 接口实现。下节内容，我们将先介绍 RGMII 接口的实现，为后续实现以太网数据的接收和发送打下基础。

48 RGMII 与 GMII 转换电路设计

工程源码	|----02_设计实例 |----ch48_gmii_rgmii_gmii
相关视频课程	
说明	

章节导读

在前面章节的内容中，我们了解了 MAC 帧的基本内容。在了解了以太网 MAC 帧的结构之后，相信读者都有一定的兴趣和自信，能够编写出对应的以太网帧发送和接收逻辑，但是在这之前，还需要先介绍下以太网 PHY 与 MAC（PHY）的连接实现方法，解决了接口问题，才能编写对应的网络协议实现逻辑。

48.1RGMII 接口信号与时序

RGMII 是 IEEE802.3z 标准中定义的千兆媒体独立接口（Gigabit Medium Independent Interface）GMII 的一个替代品。相较于 GMII 接口，RGMII 接口可以在保证传输速率不变的情况下减少管脚数。RGMII 接口通过过在时钟的上升和下降沿分别传输 1 次数据，并对控制信号采用多路复用的方式来降低管脚数量的。

RGMII 数据在时钟的上升沿和下降沿均进行采样。通常，来自 RGMII PHY 的时钟和数据同时生成，即边缘对齐，因此必须在 PCB 上对时钟信号加入布线延迟来使得数据在时钟的上升沿时处于稳定状态。下表 48-1 列出了 RGMII 接口的信号说明（以下信号方向皆是从以太网 MAC 层角度来看的）。

表 48-1 RGMII 接口信号

信号	I/O 方向	信号描述
TX_CLK	output	由 MAC 控制器输出的发送时钟
TXD[3:0]	output	TX_CLK 的上升沿传输 8 位数据中的低 4 位（bit 3:0） TX_CLK 的下降沿传输 8 位数据中的高 4 位（bit 7:4）
TX_CTL	output	TX_CLK 的上升沿传输控制信号中的 TX_EN 信号 TX_CLK 的下降沿沿传输控制信号中的 TX_EN^TX_ERR
RX_CLK	input	接收数据的参考时钟，由 PHY 输出
RXD[3:0]	input	RX_CLK 的上升沿传输 8 位数据中的低 4 位（bit 3:0） RX_CLK 的下降沿传输 8 位数据中的高 4 位（bit 7:4）
RX_CTL	input	RX_CLK 的上升沿传输控制信号中的 RX_DV 信号 RX_CLK 的下降沿沿传输控制信号中的 RX_DV^RX_ERR

图 48-1 展示了时钟和数据边沿对齐情况下通过板级 PCB 对时钟加入延迟的

方式实现目的端时钟和数据中心对齐的形式。

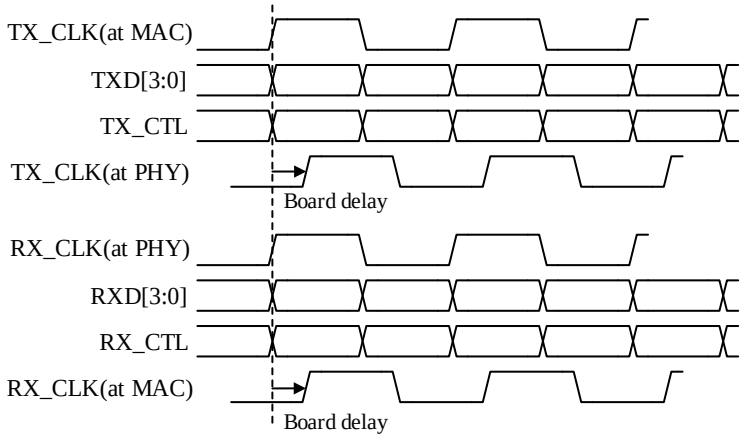


图 48-1 布线延迟

数据边沿对齐的形式使得 PCB 设计变得更加的复杂，所以，后来的 PHY 芯片都针对 RGMII 接口提供了可选的内部延迟电路。这些 PHY 芯片支持通过引脚管脚（例如 RTL8211）或配置寄存器（例如 KSZ9031、88E1512）的方式对时钟加入延迟，这样就可以降低 PCB 的布线要求。

48.2RGMII 发送的 FPGA 实现方案

对于 FPGA 来说，实现 RGMII 接口的发送是一个非常直接的过程，整个发送逻辑框图如图 48-2 所示。

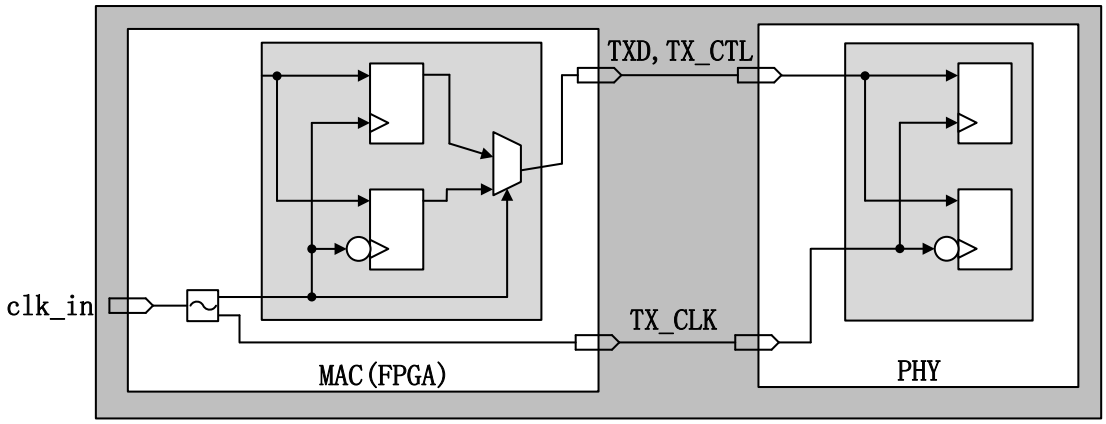


图 48-2 RGMII 发送设计逻辑框图

设计实现时，我们需要使用高云的 ODDR（Output Double Data Rate，输出双倍数据速率）原语，用于从 FPGA 器件传输双倍数据速率信号。Q0 为双倍速率数据输出，Q1 用于 Q0 所连的 IOBUF/TBUF 的 OEN 信号，ODDR 的逻辑框图如图 48-3 所示，时序图如下图 48-4 所示。

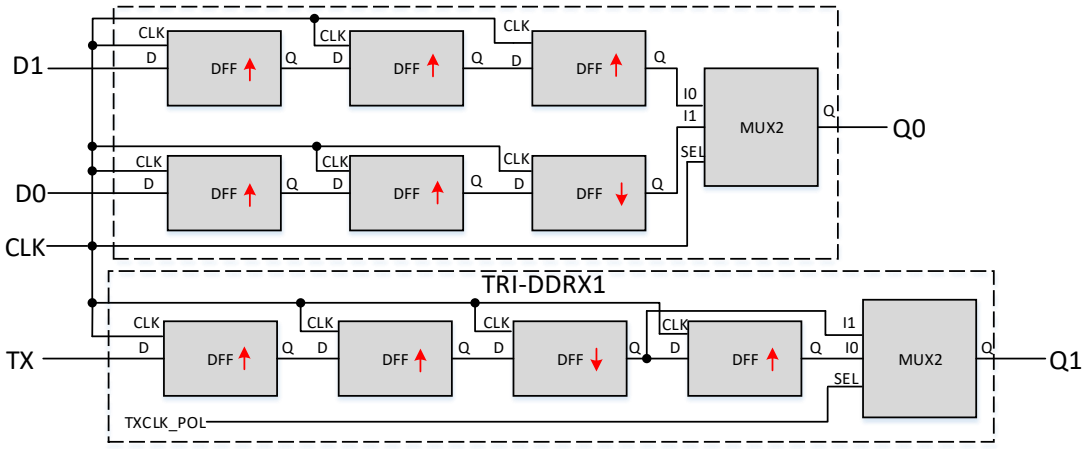


图 48-3 ODDR 逻辑框图

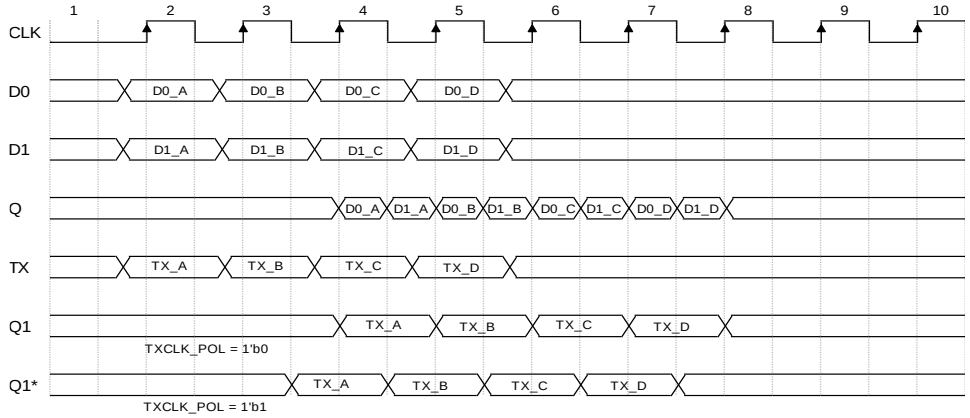


图 48-4 ODDR 时序图

ODDR 端口示意图如下图 26-16 所示，端口信号说明如下表 26-4 所示。

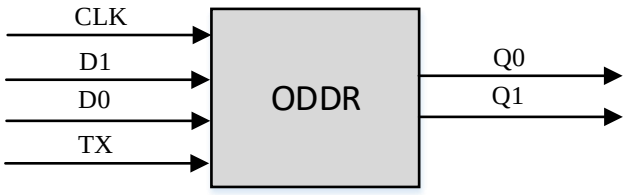


图 48-5 ODDR 端口示意图

表 48-2 ODDR 端口说明表

端口名	I/O	描述
D0,D1	Input	ODDR 数据输入信号
TX	Input	通过 TRI-DDRX1 产生 Q1
CLK	Input	时钟输入信号
Q0	Output	ODDR 数据输出信号
Q1	Output	ODDR 三态使能控制输出信号，可连接 Q0 所连的 IOBUF/TBUF 的 OEN 信号，或悬空。

ODDR 使用的时候可以直接实例化原语，例化代码如下所示：

```
ODDR uut(
    .Q0(Q0),
    .Q1(Q1),
    .D0(D0),
    .D1(D1),
    .TX(TX),
    .CLK(CLK)
);
defparam uut.TXCLK_POL=1'b0;
```

上述代码中的 TXCLK_POL 是用来控制 Q1 输出极性的，1’b0 代表 Q1 上升沿输出，1’b1 代表 Q1 下降沿输出。

在例化 ODDR 原语时，值得注意的一点是，在一个 8 位的 GMII 数据在转换为 RGMII 后，在支持 RGMII 的 PHY 芯片会在时钟上升沿发送数据的低 4 位，在时钟的下降沿发送数据的高 4 位。因此，在例化 ODDR 原语时，我们需要将数据的高 4 位数据连接到 D1 端口，将低 4 位连接到 D0 端口，以满足 PHY 对数据格式的需求。

对于数据和时钟传输，大多数支持 RGMII 的 PHY 芯片都提供了一个对发送时钟 TX_CLK 添加延迟选项。可以根据设计要求启用或禁用此选项。当启用延迟 PHY 设备内 TX_CLK 的选项时，FPGA 必须生成与数据和波形边缘对齐的时钟，如所示。

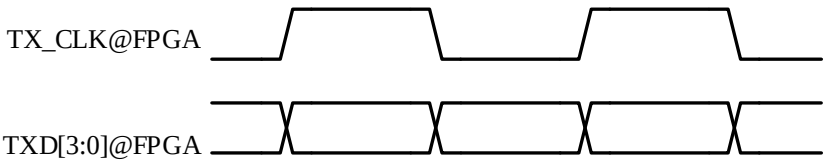


图 48-6 时钟与数据对齐关系（FPGA，启用延迟）

如果 PHY 的该功能被关闭，则 FPGA 必须对时钟信号进行一定的相位调整后再作为 TX_CLK 输出，以使 TXD 和 TX_CLK 成中心对齐关系。当然，如果 PCB 上的 TX_CLK 信号线路本身就引入了 一定的延迟，则需要将这一部分延迟考虑在内，再计算 FPGA 需要对 TX_CLK 调整的相位值，其传输波形如图 48-7 所示。

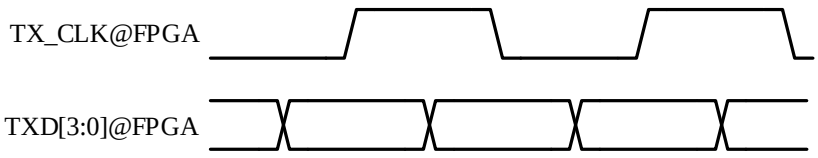


图 48-7 时钟与数据对齐关系（PHY，禁用延迟）

将源时钟和数据对齐的方法有很多，例如，我们可以使用 PLL 产生 2 路相

位相差 90 度的时钟信号，一路作为 ODDR 的工作时钟，一路作为 TX_CLK 输出。 当然，如果考虑到 PCB 板上的 TX_CLK 相对于 TXD 的延迟值，该相位可以进一步调整以确保 TX_CLK 到达 PHY 时与 TXD 呈中心对齐关系。

使用 PLL 能够实现任意相位差的两路时钟信号，可以随意调整时钟和数据传输的对齐关系，但是这种方式会占用更多 FPGA 资源。因此，在使用时，用户可以根据实际情况选择不同的方案。

48.3RGMII 接收的 FPGA 实现方案

对于 FPGA 来说，实现 RGMII 接口的接收同样是一个非常直接的过程，整个接收逻辑框图如图 48-8 所示。

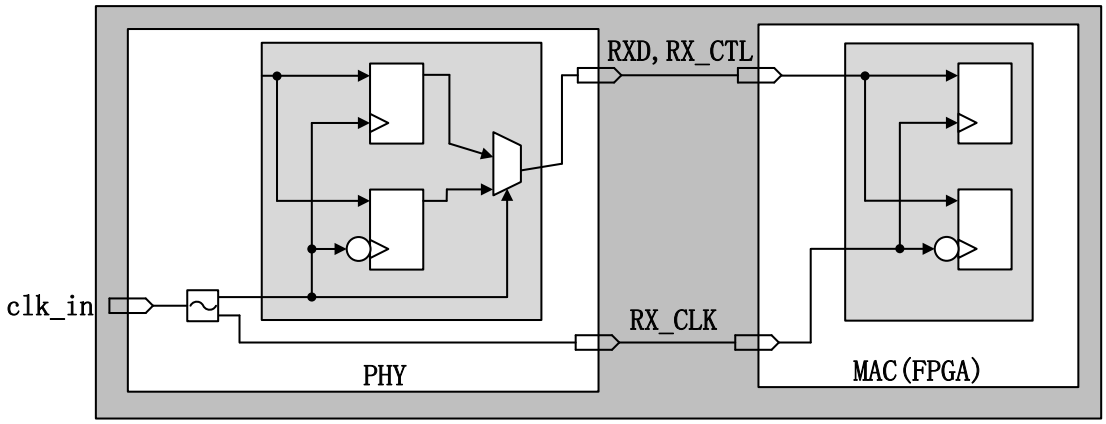


图 48-8 RGMII 接收设计逻辑框图

同样时，设计实现时，可以通过高云的 IDDR 原语实现双倍速率输入。IDDR 模式，输出数据在同一时钟边沿提供给 FPGA 逻辑。IDDR 逻辑框图如下图 48-9 所示，时序图如下图 48-10 所示。

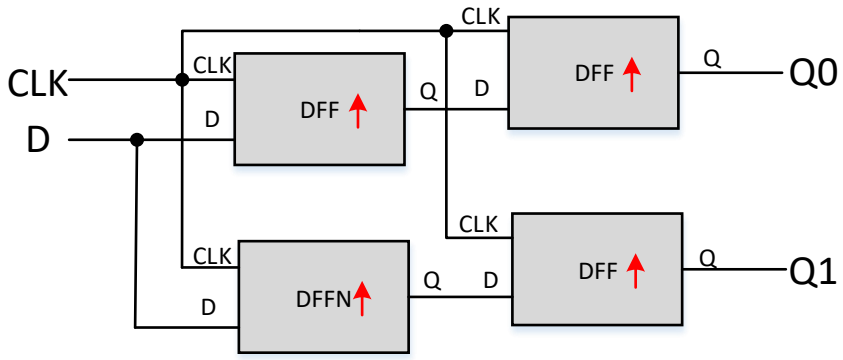


图 48-9 IDDR 逻辑框图

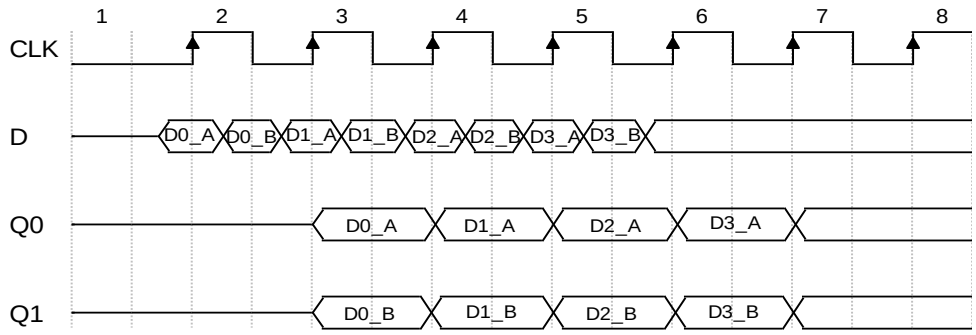


图 48-10 IDDR 时序图

IDDR 端口示意图如下图 48-11 所示，端口信号说明如下表 48-3 所示。

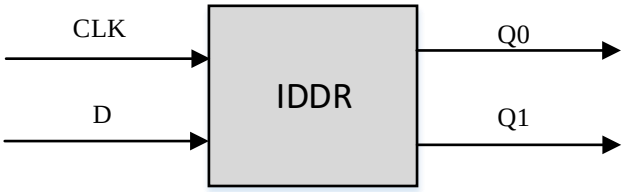


图 48-11 IDDR 端口示意图

表 48-3 IDDR 端口介绍表

端口名	I/O	描述
D	Input	IDDR 数据输入信号
CLK	Input	时钟输入信号
Q0，Q1	Output	IDDR 数据输出信号

IDDR 使用的时候可以直接实例化原语，例化代码如下所示：

```
IDDR uut(
  .Q0(Q0),
  .Q1(Q1),
  .D(D),
  .CLK(CLK)
);
```

为了让接收的数据能够更加容易的被 MAC 捕获，大多数支持 RGMII 的 PHY 芯片都提供了一个对接收时钟 RX_CLK 添加延迟的选项。可以根据设计要求启用或禁用此选项。当启用延迟 PHY 设备内容 RX_CLK 的选项时，PHY 输出的 RX_CLK 与 RXD 会呈中心对齐的关系，如图 48-12 所示。在这种情况下，FPGA 就可以直接使用 RX_CLK 来捕获输入的数据，而不需要再对 RX_CLK 添加 PCB 板级延迟，或者是在 FPGA 内部对 RX_CLK 添加延迟。

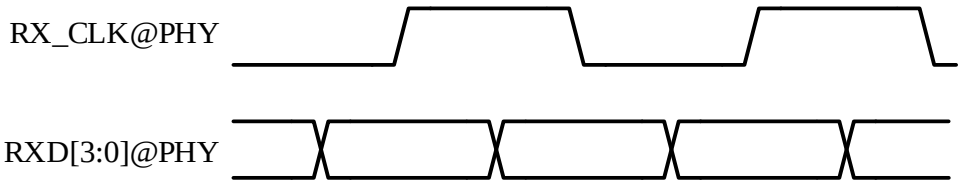


图 48-12 时钟和数据对齐关系（PHY，启用延迟）

如果 PHY 对 RX_CLK 添加延迟的功能被关闭（波形图如图 48-13 所示），则 FPGA 必须对 RX_CLK 时钟信号进行一定的相位调整后再用来捕获数据，这种情况，可以使用 FPGA 内的 PLL 将该信号进行一定的相位移动后再用来捕获数据。当然，让 RX_CLK 能够进入 PLL，必须要求改信号是从 FPGA 器件的专用时钟输入管脚或 DQS 脚输入。

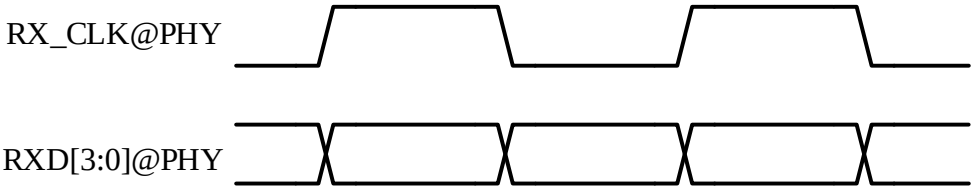


图 48-13 时钟与数据对齐关系（PHY，禁用延迟）

可以看到，通过 FPGA 侧实现 RX_CLK 和 RXD 的中心对齐需要使用到的 PLL 资源，因此，使用 PHY 内部的 RX_CLK 延迟功能将降低对 FPGA 的资源占用。推荐有条件的情况下尽量选择 PHY 内部增加延迟的方案。

当然，对于本次设计而言，我们可以直接使用 IDDR 原语实现。

48.3.1使用 FPGA 实现 RGMII 接口

了解了 RGMII 在 FPGA 内部的实现原理和方法后， 实现就非常的方便了。

RGMII 接口实现分为接收和发送两部分，而接收和发送中，又分为时钟、控制信号与数据两部分。在前面的小节，本文已经介绍了实现发送和接收以及数据与时钟对齐的方法， 这里就以图文的形式展示一个具体的实现过程。

48.3.1.1RGMII 发送接口实现

对于 RGMII 发送，只需要例化 6 个 ODDR 就可以实现，其中 4 个用来发送 4 位的 TXD 信号；一个用来发送 TXEN 和 TXER 信号。另一个则用来输出 TX_CLK 信号。图 48-14 为使用 ODDR 原语实现 RGMII 发送的系统框图：

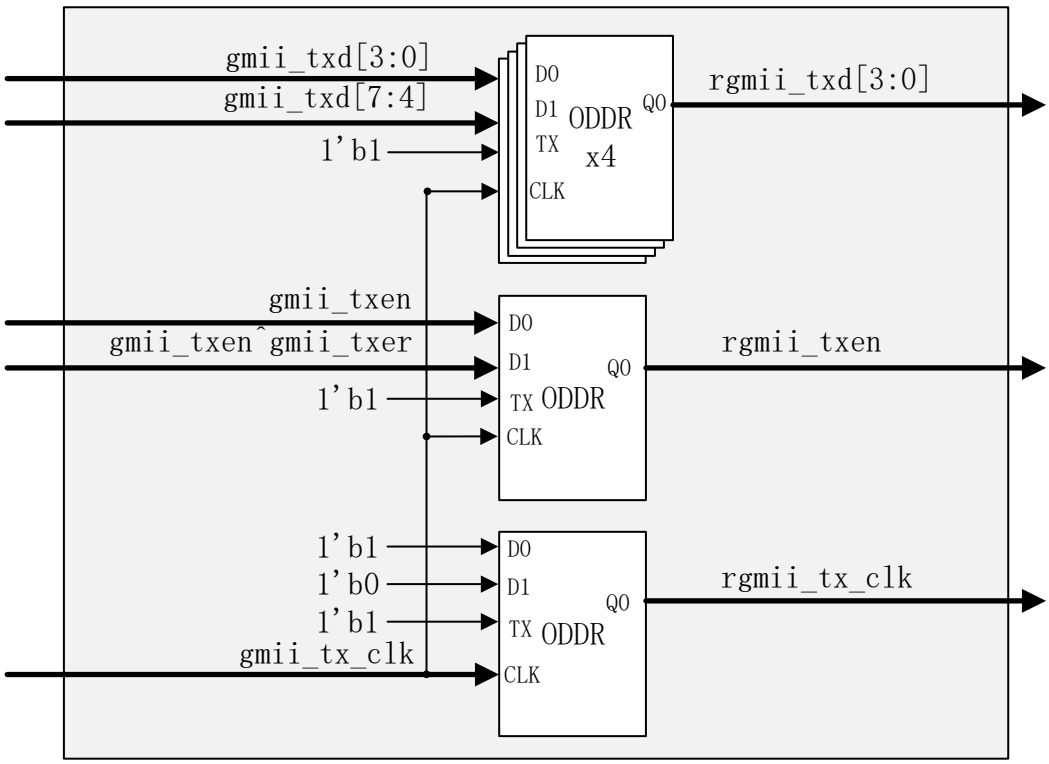


图 48-14 ODDR 原语的 RGMII 发送实现

据图可知，实现 GMII 转 RGMII 接口，甚至不需要我们自己编写任何逻辑代码，只需要例化 6 个 ODDR，然后将信号连接到对应的端口即可。

48.3.1.2rgmii_tx_clk 的输出

在上面的系统框图中，相信已经有读者发现了一个差异，即对于 rgmii_tx_clk 的输出，图中也是使用 ODDR 输出的，而非使用 PLL 产生并输出。

使用 ODDR 输出 rgmii_tx_clk 信号的最大优势在于，可以最大程度上保证输出的 rgmii_tx_clk 与 rgmii_txd[3:0]保持边沿对齐关系，因为从实现原理上来讲，rgmii_tx_clk 和 rgmii_txd[3:0]都是 ODDR 在 gmii_tx_clk 的驱动下输出其高位和低位的数据，这样能够保证两者的传输路径的时序模型完全一致，从而使得 rgmii_tx_clk 和 rgmii_txd[3:0]都能在同一时刻发生变化，实现更加可靠的时钟和数据边沿对齐特性。只是这种情况下，在 PHY 侧，要么通过设置，在 PHY 内部对 rgmii_tx_clk 加入时间延迟，要么通过控制 PCB 走线来为 rgmii_tx_clk 增加一定时间的延迟。如果对于某些设计时序没有严格控制好的平台板卡，在 FPGA 侧使用边沿对齐和中心都无法保证数据和时钟到达 PHY 时恰好呈中心对齐或边沿对齐模式，则需要使用 PLL 直接输出 rgmii_tx_clk，并通过实验调整其与 FPGA 内部使用的 gmii_tx_clk 的相位来最终达到稳定输出数据的目的。在我

们设计的开发板上，一般都是通过设置 PHY 内部对 rgmii_tx_clk 加入时间延迟来实现的。无需使用 PLL 来调整发送时钟 rgmii_tx_clk 的相位。

48.3.1.3GMII 转 RGMII 顶层代码

在上面，我们已经完整了介绍了实现 GMII 转 RGMII 的方案和各个模块的细节，这里，只需要通过一个顶层模块将上述内容例化到一起，就能最终实现前面框图所介绍的系统了。因此，完整的 GMII 转 RGMII 代码如下：

```
module gmii_to_rgmii(
  reset_n,

  gmii_tx_clk,
  gmii_txd,
  gmii_txen,
  gmii_txer,

  rgmii_tx_clk,
  rgmii_txd,
  rgmii_txen
);


  input        reset_n;

  input        gmii_tx_clk;
  input  [7:0] gmii_txd;
  input        gmii_txen;
  input        gmii_txer;

  output       rgmii_tx_clk;
  output [3:0] rgmii_txd;
  output       rgmii_txen;

  generate
    genvar i;
        for(i=0;i<4;i=i+1)
        begin
        ODDR    U_ODDR_dq1
                (.Q0(rgmii_txd[i]),
                 .Q1(),
                 .D0(gmii_txd[i]),
                 .D1(gmii_txd[i+4]),
```

```
                .TX(1),
                .CLK(gmii_tx_clk)
               );
        end
  endgenerate

  ODDR    U_ODDR_en1
      (.Q0(rgmii_txen),
             .Q1(),
             .D0(gmii_txen),
             .D1(gmii_txen^gmii_txer),
             .TX(1),
             .CLK(gmii_tx_clk)
      );

  ODDR    U_ODDR_clk1
      (.Q0(rgmii_tx_clk),
             .Q1(),
             .D0(1),
             .D1(0),
             .TX(1),
             .CLK(gmii_tx_clk)
      );

endmodule
```

至此，GMII 转 RGMII 的逻辑功能就设计完成了。接下来我们需要设计一个激励文件，来验证设计能否正常工作。

48.3.1.4仿真验证

本次仿真代码如下所示：

```
`timescale 1ns / 1ps
`define CLK_PERIOD 8

module gmii_to_rgmii_tb();
    reg  reset_n;
    reg gmii_tx_clk;
    reg [3:0] tx_byte_cnt;
    reg [7:0] gmii_txd;
    reg gmii_txen;
    reg gmii_txer;
    wire rgmii_tx_clk;
```

```
    wire [3:0] rgmii_txd;
    wire rgmii_txen;

  GSR GSR(.GSRI(1'b1));

    gmii_to_rgmii gmii_to_rgmii(
      .reset_n(reset_n),
      .gmii_tx_clk(gmii_tx_clk),
      .gmii_txd(gmii_txd),
      .gmii_txen(gmii_txen),
      .gmii_txer(gmii_txer),

      .rgmii_tx_clk(rgmii_tx_clk),
      .rgmii_txd(rgmii_txd),
      .rgmii_txen(rgmii_txen)
    );

        //clock generate
    initial gmii_tx_clk = 1'b1;
    always #(`CLK_PERIOD/2)gmii_tx_clk = ~gmii_tx_clk;

    always@(posedge gmii_tx_clk or negedge reset_n)
    if(!reset_n)
        tx_byte_cnt <= 4'd0;
    else if(gmii_txen)
        tx_byte_cnt <= tx_byte_cnt + 1'b1;
    else
        tx_byte_cnt <= 4'd0;

    always@(*)
    begin
        case(tx_byte_cnt)
        16'd0  : gmii_txd = 25;
        16'd1  : gmii_txd = 34;
        16'd2  : gmii_txd = 18;
        16'd3  : gmii_txd = 96;
        16'd4  : gmii_txd = 78;
        16'd5  : gmii_txd = 29;
        16'd6  : gmii_txd = 63;
        16'd7  : gmii_txd = 42;
        16'd8  : gmii_txd = 51;
        16'd9  : gmii_txd = 74;
        16'd10 : gmii_txd = 39;
        16'd11 : gmii_txd = 60;
```

```
        16'd12 : gmii_txd = 27;
        16'd13 : gmii_txd = 36;
        16'd14 : gmii_txd = 145;
        16'd15 : gmii_txd = 231;
        endcase
    end

  initial
    begin
    reset_n = 0;
    gmii_txen = 0;
    gmii_txer = 0;
    #201;
    reset_n = 1;
    gmii_txen = 1;
    #2000;
    $stop;
  end

endmodule
```

仿真文件内容较为简单，通过 gmii_txen 信号控制计数器 tx_byte_cnt 在 0~15 间不断自加循环，进而控制输入输出 gmii_txd 在几个给定的值中循环。随后观察 rgmii 输出的波形数据，其仿真结果如下图 48-15 所示。

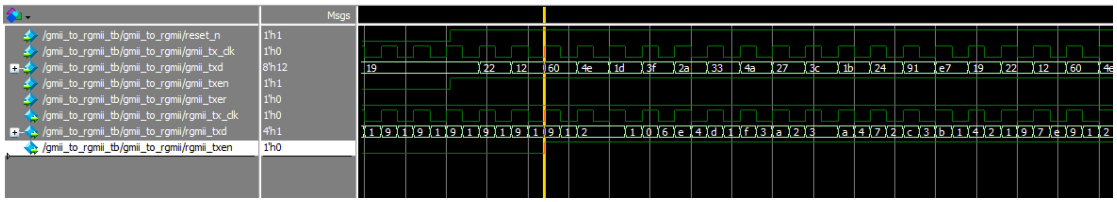


图 48-15 GMII 转 RGMII 仿真时序

可以看到，输入数据 gmii_txd 依次为 0x19、0x22、0x12、0x60、0x4e...。输出端口 rgmii_txd 在 rgmii_tx_clk 时钟的双沿上依次输出 0x9、0x1、0x2、0x2、0x2、0x1、0、0x6、0xe、0x4...，即在时钟上升沿输出数据低 4 位，在时钟下降沿输出数据高 4 位。满足 PHY 芯片对数据的时序要求，说明该模块设计成功。

在后续设计千兆发送逻辑时，只需要先完全按照 GMII 接口设计逻辑代码，然后在输出到 FPGA 管脚时，加入该转换逻辑，即可快速实现 RGMII 接口的千兆以太网发送逻辑。

48.3.2RGMII 接收接口实现

对于 RGMII 发送，只需要例化 5 个 IDDR 就可以实现，其中 4 个用来接收 4 位的 RXD 信号；一个用来接收 RXEN 和 RXER 信号。图 48-16 为使用 IDDR 原语实现 RGMII 发送的系统框图。

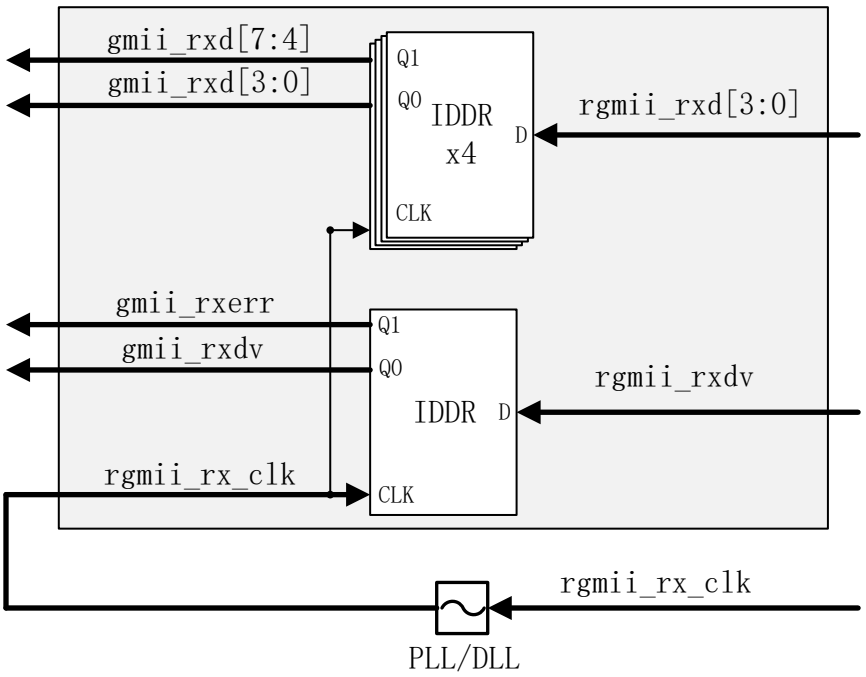


图 48-16 IDDR 的 RGMII 接收实现

据图可知，实现 RGMII 转 GMII 接口，也不需要我们自己编写任何逻辑代码，只需要例化 5 个 IDDR，然后将信号连接到对应的端口即可。

48.3.2.1rgmii_rx_clk 的接收

对于 RGMII 接口来说，rgmii_rx_clk 时钟是由 PHY 产生，提供给 FPGA 的。而该时钟在到达 FPGA 内部接收 rgmii_rxd 和 rgmii_rxdv 的 IDDR 时需要与 rgmii_rxd 和 rgmii_rxdv 信号呈中心对齐的关系。而 rgmii_rxdv 时钟不仅需要作为 IDDR 的接口时钟，在 FPGA 内部还会作为以太网数据的解包逻辑时钟，是一个典型的时钟信号。所以如果直接使用普通的 FPGA IO 接入该时钟，势必会导致时钟质量下降，从而引起逻辑运行的不稳定。而且，一旦该时钟信号和 rgmii_rxd 不满足中心对齐的关系，要调整会非常的麻烦。所以，综合考虑，将该时钟信号接入了 FPGA 的专用时钟管脚，使其能够直接进入 FPGA 内部的 PLL，也能够直接上 FPGA 的全局时钟资源。在保证时钟质量的同时，还能确保万一该时钟相位无法满足 rgmii_rxd 的捕获要求，还能通过 FPGA 内部的 PLL 进行相

位调制，以确保最终 rgmii_rxd 能够被正确接收。所以在上述框图中，rgmii_rx_clk 接入 FPGA 后并没有直接接到 IDDR 的时钟脚，而是先进入了 PLL，由 PLL 调整到合适的相位之后，再用来捕获 rgmii_rxd。

48.3.2.2RGMII 转 GMII 顶层代码

在上面，我们已经完整的介绍了实现 RGMII 转 GMII 的方案和各个模块的细节，这里，只需要通过一个顶层模块将上述内容例化到一起，就能最终实现前面框图所介绍的系统了。完整的 RGMII 转 GMII 代码如下：

```
module rgmii_to_gmii(
  reset,

  rgmii_rx_clk,
  rgmii_rxd,
  rgmii_rxdv,

  gmii_rx_clk,
  gmii_rxdv,
  gmii_rxd,
  gmii_rxer
);


  input         reset;

  input         rgmii_rx_clk;
  input  [3:0]  rgmii_rxd;
  input         rgmii_rxdv;

  output        gmii_rx_clk;
  output [7:0]  gmii_rxd;
  output        gmii_rxdv;
  output        gmii_rxer;

  assign gmii_rx_clk = rgmii_rx_clk;

  genvar i;
  generate
    for(i=0;i<4;i=i+1)
    begin: rgmii_rxd_i
      IDDR U_IDDR_dq1 (
        .Q0   (gmii_rxd[i]   ),
```

```
        .Q1   (gmii_rxd[i+4] ),
        .D    (rgmii_rxd[i]  ), // 1-bit DDR data input
        .CLK  (rgmii_rx_clk  ) // 1-bit clock input
      );
    end
  endgenerate

  IDDR U_IDDR_dv1 (
    .Q0   (gmii_rxer    ), // 1-bit output for positive edge of clock
    .Q1   (gmii_rxdv    ), // 1-bit output for negative edge of clock
    .D    (rgmii_rxdv   ), // 1-bit DDR data input
    .CLK  (rgmii_rx_clk ) // 1-bit clock input
  );

endmodule
```

至此， RGMII 转 GMII 的逻辑功能就设计完成了，接下来我们需要设计一个激励文件，来验证设计是否能够正常工作。

48.3.2.3仿真验证

本次仿真代码如下：

```
`timescale 1ns / 1ps
`define CLK_PERIOD 8

module rgmii_to_gmii_tb();
    reg  reset_n;
    wire gmii_rx_clk;
    reg [3:0] rx_byte_cnt;
    wire [7:0] gmii_rxd;
    wire gmii_rxdv;
    wire gmii_rxerr;
    wire rgmii_rx_clk;
    reg [3:0] rgmii_rxd;
    reg rgmii_rxdv;
    reg rx_clk;
    wire locked;

    GSR GSR(.GSRI(1'b1));
    rgmii_to_gmii rgmii_to_gmii(
        .reset(!reset_n),
        .gmii_rx_clk(gmii_rx_clk),
        .gmii_rxdv(gmii_rxdv),
```

```
        .gmii_rxd(gmii_rxd),
        .gmii_rxer(gmii_rxerr),

        .rgmii_rx_clk(rgmii_rx_clk),
        .rgmii_rxd(rgmii_rxd),
        .rgmii_rxdv(rgmii_rxdv)
    );

    Gowin_PLL Gowin_PLL(
        .lock(locked), //output lock
        .clkout0(rgmii_rx_clk), //output clkout0
        .clkin(rx_clk), //input clkin
        .reset(!reset_n) //input reset
    );

    //clock generate
    initial rx_clk = 1'b1;
    always #(`CLK_PERIOD/2)rx_clk = ~rx_clk;

    always@(rx_clk or negedge reset_n)
    if(!reset_n)
        rx_byte_cnt <= 4'd0;
    else if(rgmii_rxdv && locked)
        rx_byte_cnt <= rx_byte_cnt + 1'b1;
    else
        rx_byte_cnt <= 4'd0;

    always@(*)
    begin
        case(rx_byte_cnt)
        16'd0  : rgmii_rxd = 12;
        16'd1  : rgmii_rxd = 7;
        16'd2  : rgmii_rxd = 9;
        16'd3  : rgmii_rxd = 6;
        16'd4  : rgmii_rxd = 11;
        16'd5  : rgmii_rxd = 15;
        16'd6  : rgmii_rxd = 0;
        16'd7  : rgmii_rxd = 8;
        16'd8  : rgmii_rxd = 4;
        16'd9  : rgmii_rxd = 2;
        16'd10 : rgmii_rxd = 5;
        16'd11 : rgmii_rxd = 1;
        16'd12 : rgmii_rxd = 3;
        16'd13 : rgmii_rxd = 10;
```

```
        16'd14 : rgmii_rxd = 14;
        16'd15 : rgmii_rxd = 13;
        endcase
    end

  initial
    begin
    reset_n = 0;
    rgmii_rxdv = 0;
    #201;
    reset_n = 1;
    rgmii_rxdv = 1;
    @(posedge locked)
    #2000;
    $stop;
  end

endmodule
```

仿真中添加了一个 PLL，用于对 rgmii_rx_clk 时钟进行 90 度的相位调制，以保证该时钟到达 IDDR 时，与 rgmii_rxd 和 rgmii_rxdv 信号呈中心对齐的关系。（但是在实际使用过程中发现，90 度的相位调制并不能稳定输出数据，270 度的相位调制才能稳定输出数据）通过将 PLL 的 locked 信号与 rgmii_rxdv 信号逻辑与，确保只有在 PLL 输出稳定时钟时，控制计数器 rx_byte_cnt 的值能够在 0~15 间不断自加循环。随后根据计数器的值，控制输入数据 rgmii_rxd 在几个易于观测的给定值中循环。观察 gmii 输出的波形数据，其仿真结果如图 48-17 所示。

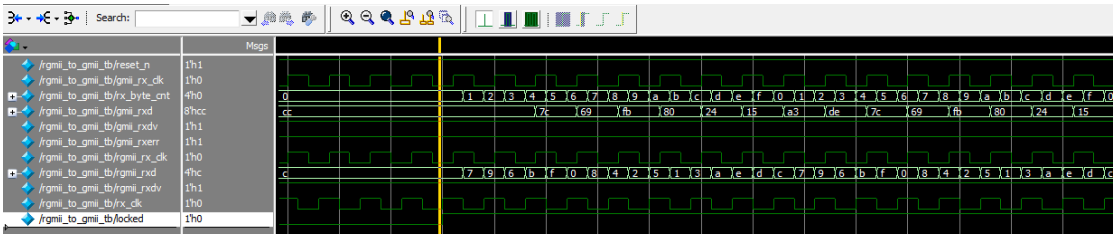


图 48-17 RGMII 转 GMII 仿真时序

可以看到，当 PLL 输出时钟稳定后，输入数据 rgmii_rxd 开始变化，输入值依次为 0xc、0x7、0x9、0x6、0xb、0xf、0...。此时输入到 IDDR 的时钟也刚好与 rgmii_rxd 和 rgmii_rxdv 信号呈中心对齐，数据在时钟的双沿被采集进 IDDR 中，上升沿采集到的数据作为低四位，下降沿采集到数据作为高四位。因此，输出数据 gmii_rxd 依次为 0xcc、0x7c、0x69、0xfb...。满足设计所需要求，说明

该模块设计成功。

在后续设计千兆以太网接收逻辑时，只需要完全按照 GMII 接口设计逻辑代码，然后再连接到 FPGA 管脚时，加入该转换逻辑，即可快速实现 RGMII 接口的千兆以太网接收逻辑。

48.4总结

至此，关于 RGMII 接口与 GMII 接口的互转逻辑设计就已经介绍完成了，有了这个互转逻辑，在 FPGA 中设计以太网的接收和发送逻辑时，只需要按照 GMII 接口的形式，先设计出对应的发送和接收逻辑，再将对应的端口连接到 RGMII 与 GMII 接口转换逻辑上，就能够完成基于 RGMII 接口的以太网接收和发送了。

49 ARP 协议的原理与 FPGA 实现

工程源码	|----02_设计实例 |----ch49_arp_send_rgmii
相关视频课程	
说明	

章节导读

前面章节中，我们介绍了整个以太网传输中最底层的 MAC 帧格式，也提到，MAC 帧并不是直接最终交付给用户的数据帧，一段用户信息要想通过以太网传输还需要经过各种传输层协议的层层打包才能最终送入 MAC 帧的数据字段进行传输。无论是传输用户数据还是一些网络通信辅助相关的信息，都需要将数据编码为指定协议后再打包进 MAC 层传输。所以本节将选择一个最简单的协议 ARP 协议，来介绍以太网 MAC 帧的组包方式，并使用 FPGA 编写 MAC 帧发送器，来发送预先设定好的一段 ARP 协议数据内容，并通过以太网抓包查看数据内容的方式来直观验证 MAC 帧发送器的设计成果。

之所以选择 ARP 协议来辅助进行以太网 MAC 帧发送器的设计，是因为 ARP 协议是所有 MAC 帧上层协议中最简单的协议，使用该协议，我们在生成以太网帧的数据和填充字段的内容时会非常的简单方便，而 ARP 协议又能够在 PC 的以太网抓包工具中正确抓取并显示，非常适合我们用来进行最直观的连通性测试。

49.1ARP 帧存在的目的

在网络通讯时，源主机的应用程序知道目的主机的 IP 地址和端口号，却不知道目的主机的硬件(MAC)地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP 协议就起到这个作用。

49.2ARP 帧工作原理

源主机发出 ARP 请求，询问“IP 地址是 192.168.0.1 的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填 FF:FF:FF:FF:FF:FF 表示广播）。目的主机接收到广播的 ARP 请求，发现其中的 IP 地址与本机相符，则发送一个 ARP 应答数据包给源主机，将自己的硬件地址填写在应答包中。然后源主机收到该应答报文，就能从报文中解析得知目的主

机的硬件地址是多少，然后就能正确的发送数据了。

每台主机都维护一个 ARP 缓存表，可以用 arp -a 命令查看。缓存表中的表项有过期时间（一般为 20 分钟），如果 20 分钟内没有再次使用某个表项，则该表项失效，下次还要发 ARP 请求来获得目的主机的硬件地址。由于网络上的 IP 地址本身是动态分配的（Ipv4 资源严重不足，所以采用动态分配机制），上网时每个设备的 IP 地址在不同的时间点其地址都有可能不同，所以某个网络设备在不同的时间，其 IP 地址可能不一样，所以需要用这种缓存表项定期失效的机制来避免 MAC 地址和不同时刻 IP 地址绑定的冲突。下表 49-1 为完整的 ARP 帧每个字段的数据意义（包含了以太网帧头部字段）。

表 49-1 ARP 帧每个字段的数据意义

14 字节以太网帧首部			28 字节 ARP 请求/应答帧内容								
以太网目的地址	以太网源地址	帧类型	硬件类型	协议类型	硬件地址长度	协议地址长度	操作码（op）	发送端以太网地址	发送端 IP 地址	目的以太网地址	目的 IP 地址
6	6	2	2	2	1	1	2	6	4	6	4

注意到源 MAC 地址、目的 MAC 地址在以太网首部和 ARP 请求中各出现一次，这两个字段的内容值是不一样的。在发送 ARP 请求帧时，以太网首部中的目的 MAC 地址为广播地址，也就是 6 字节的 0xFF。而 ARP 数据报中当发送请求报文时由于目的 MAC 地址暂时无法知晓，所以将其值填充为 0。ARP 返回报文中的值由于已经知晓，所以以太网首部和 ARP 数据报中的值用实际值填充。

硬件类型指链路层网络类型，1 为以太网，协议类型指要转换的地址类型，0x0800 为 IP 地址。后面两个地址长度对于以太网地址和 IP 地址分别为 6 和 4（字节）。op 字段为 1 表示 ARP 请求，op 字段为 2 表示 ARP 应答。每个字段具体的功能描述如下表 49-2 所示。

表 49-2 以太网 ARP 报文各字段功能含义

	字段	度	默认值	备注
以太网首部	接收方 MAC	6		发送 ARP 请求报文时，该值为广播地址（FF-FF-FF-FF-FF-FF） 发送 ARP 应答报文时，该值为对应的发送请求报文设备的实际 MAC 地址。
	发送方 MAC	6		数据帧发送方 MAC 地址
	Ethertype	2	0x0806	0x0806 是 ARP 帧的类型值
ARP 字段	硬件类型	2	0x0001	以太网类型值
	上层协议类型	2	0x0800	上层协议为 IP 协议

	MAC 地址长度	1	0x6	以太网 MAC 地址长度为 6
	IP 地址长度	1	0x4	IP 地址长度为 4
	操作码	2	0x1/ 0x2	0x1 表示 ARP 请求报文 0x2 表示 ARP 应答报文
	发送方 MAC	6		源 MAC 地址
	发送方 IP	4		源 IP 地址
	接收方 MAC	6		发送 ARP 请求报文时，该值由于当前未知，用全 0 填充。 发送 ARP 应答报文时，该值为对应的发送请求报文设备的实际 MAC 地址。
	接收方 IP	4		目的 IP 地址
	填充数据	18		因为物理帧最小长度为 64 字节,前面的 42 字节再加上 4 个 CRC 校验字节,还差 18 个字节
以太网校验	校验字	4		CRC32 校验值，以上所有数据参与校验的结果

49.3手工组建 ARP 数据帧

在了解了以上以太网帧和 ARP 帧的具体格式之后，只要我们确定一个发送方和一个接收方，就能手工组装一个数据帧了。以下以一个具体的场景为例来进行介绍。

假设现在有两台设备，分别命名为 A 和 B。

A 设备 MAC 地址为 00-0a-35-01-fe-c0，IP 地址为 192.168.0.2（C0-A8-00-02）

B 设备 MAC 地址为 84-7b-eb-48-94-13，IP 地址为 192.168.0.3（C0-A8-00-03）

注意，B 设备实际就是笔者实验用的电脑，其 MAC 地址就是笔者实验电脑的实际 MAC 地址，IP 地址则是我们以太网实验中统一使用的 PC 的 IP 地址，需要提前设置好，方能在后续进行实验时不遇到问题。

A 设备发起 ARP 请求包，请求 IP 地址为 192.168.0.3 的设备的 MAC 地址，B 设备收到该 ARP 请求包，且经过匹配，发现请求的 IP 地址与自己一致，然后发出 ARP 应答包。

MAC:00-0a-35-01-fe-c0
IP:　192.168.0.2
设备A

MAC: 84-7b-eb-48-94-13
IP:　192.168.0.3
设备B

据此，除 CRC32 字段需要根据数据内容计算得出以外，ARP 请求报文和 ARP 应答报文的每个字段的准确值都可以手动组建出。而一旦这些值确定了，对应的 CRC32 校验值也就可以通过计算得到唯一值。下表 49-3 为根据上述参数手动组建出的 ARP 请求包和 ARP 应答包的所有字段的值。

表 49-3 ARP 请求包和 ARP 应答包所有字段的值

数据帧字段	ARP 请求帧	ARP 应答帧
接收方 MAC	FF-FF-FF-FF-FF-FF	00_0a_35_01_fe_c0
发送方 MAC	00_0a_35_01_fe_c0	84-7b-eb-48-94-13
类型	08-06	08-06
硬件类型	00-01	00-01
协议类型	08-00	08-00
MAC 地址长度	06	06
IP 地址长度	04	04
操作代码	01	02
发送方 MAC	00_0a_35_01_fe_c0	84-7b-eb-48-94-13
发送方 IP	C0-A8-00-02	C0-A8-00-03
接收方 MAC	00-00-00-00-00-00	00_0a_35_01_fe_c0
接收方 IP	C0-A8-00-03	C0-A8-00-02
填充字段	18 个 00	18 个 00
校验字段	由实际帧内容计算	由实际帧内容计算

49.4使用以太网测试工具生成数据帧

上述组包方式是在我们充分掌握了以太网 MAC 帧和 ARP 帧的每一个字段的物理意义后手动组建出来的，若对于这两个帧结构掌握不是十分牢固，则有可能在组建过程中出错，所以这里再给大家介绍一种使用工具组包的方法。该工具名为“小兵以太网测试仪”，软件安装包名为“小兵以太网测试仪.rar”或者“以太网帧生成器.rar”或者“xb_ether_tester_v3.2.8.rar”。

该软件是基于 wincap 包开发的以太网数据帧发送和接收工具。安装并打开该软件，在界面中点击新建流按钮以打开新报文编辑界面，如下图 49-1 所示。

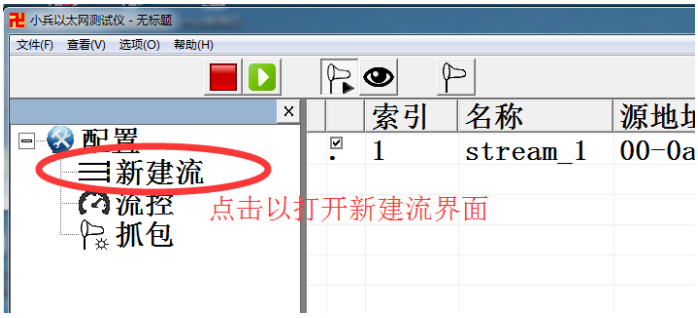


图 49-1 打开新建流界面

在常用报文中选择 arp 请求格式报文，自动更新栏中的所有选项在本报文生成中开启和关闭都可以，因为不涉及到一些动态数据的生成。

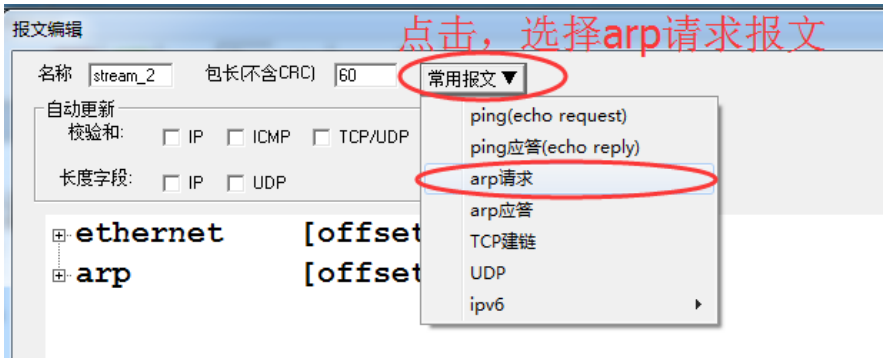


图 49-2 选择报文格式

修改以太网包和 arp 包中各字段内容如下图 49-3 所示：

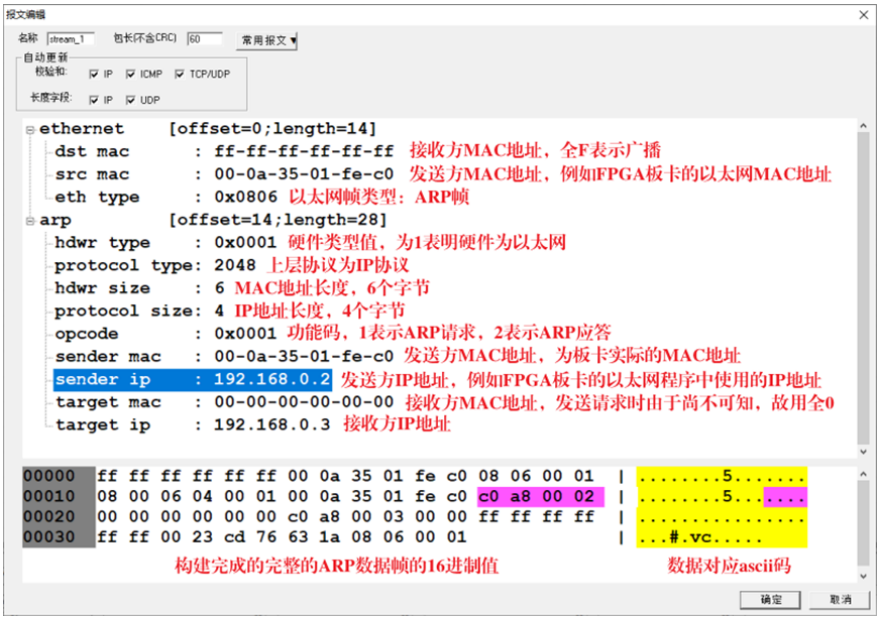


图 49-3 修改报文各字段内容

配置好之后点击确定，然后将构建好的数据包导出为 16 进制格式文本文件，如下图 49-4 所示。

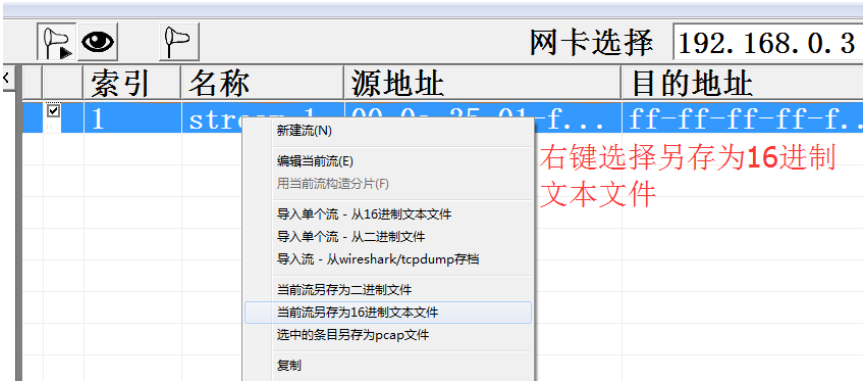


图 49-4 保存文件为 16 进制格式的文件文件

得到的文本中数据内容如下所示：

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x0a, 0x35, 0x01, 0xfe, 0xc0, 0x08, 0x06, 0x00, 0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01, 0x00, 0x0a, 0x35, 0x01, 0xfe, 0xc0, 0xc0, 0xa8, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0xa8, 0x00, 0x03, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x23, 0xcd, 0x76, 0x63, 0x1a, 0x08, 0x06, 0x00, 0x01,

49.5使用 Wireshark 抓包验证

上述组建好的报文，可以直接使用小兵以太网测试仪通过网卡发包，然后我们可以使用 wireshark 抓取网卡上的该报文。

要想 wireshark 能够正常抓有线网卡上的数据包，首先需要让你的电脑的有线网卡处于连接的活动状态，在做 FPGA 实验时，例如下载我们提供的以太网回环测试例程，就能让电脑的网卡连接上，处于活动状态。也可以将电脑网卡连接到交换机，路由器等能够让网卡检测到连接的设备上。

其次，在小兵以太网测试仪的右上角切换网卡为你的有线网卡，如下图 49-5 所示：

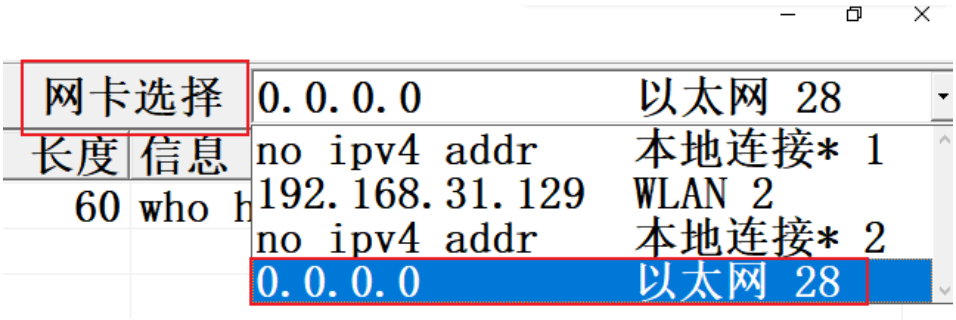


图 49-5 网卡选择

选中刚刚组建好的 ARP 数据包，然后点击开始发包按钮执行发包。

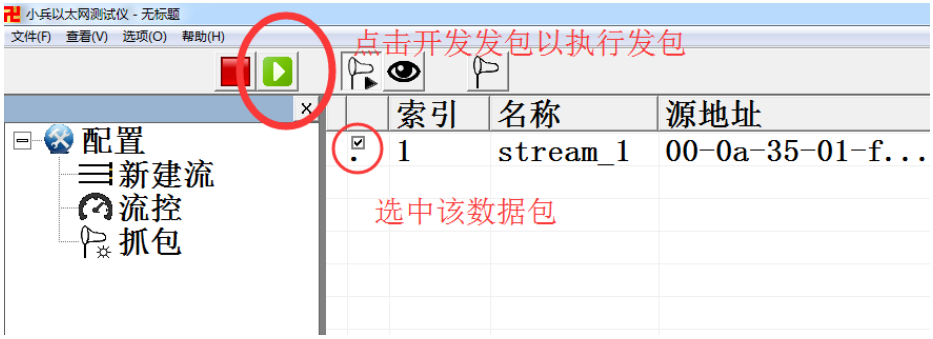


图 49-6 ARP 发包

打开 Wireshark 软件并抓取有线网卡的连接，则可以抓取到大量的 ARP 请求报文和应答报文，如下图 49-7 所示。

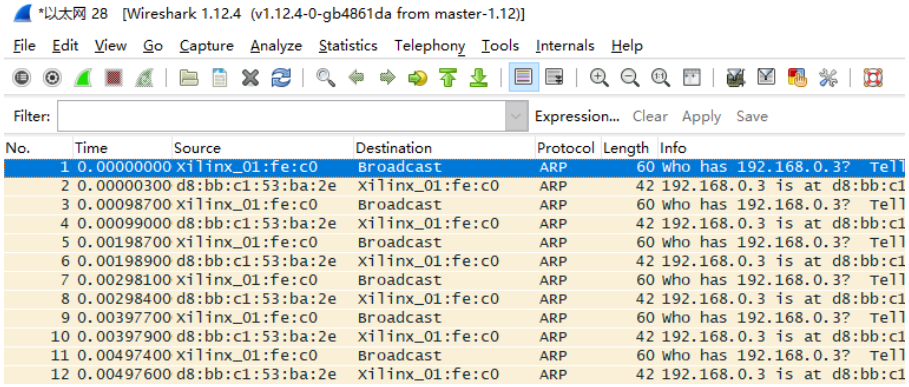


图 49-7 ARP 请求与应答报文

其中，Destination 项为 Broadcast 的包就是 ARP 请求包，紧随其后的一个包就是该 ARP 请求包的应答包。

注意，在整个实验中，B 设备实际就是笔者实验用的电脑，其 MAC 地址就是笔者实验电脑的实际 MAC 地址，IP 地址则是我们以太网实验中统一使用的 PC 的 IP 地址，所以在上述抓包过程中，会有 ARP 应答包存在。读者自己在实验时，也需要先将自己的实验电脑有线网卡 IP 地址设置为 192.168.0.3。

选择一个 ARP 请求包，查看其详细的帧内容，如下图 49-8 所示：

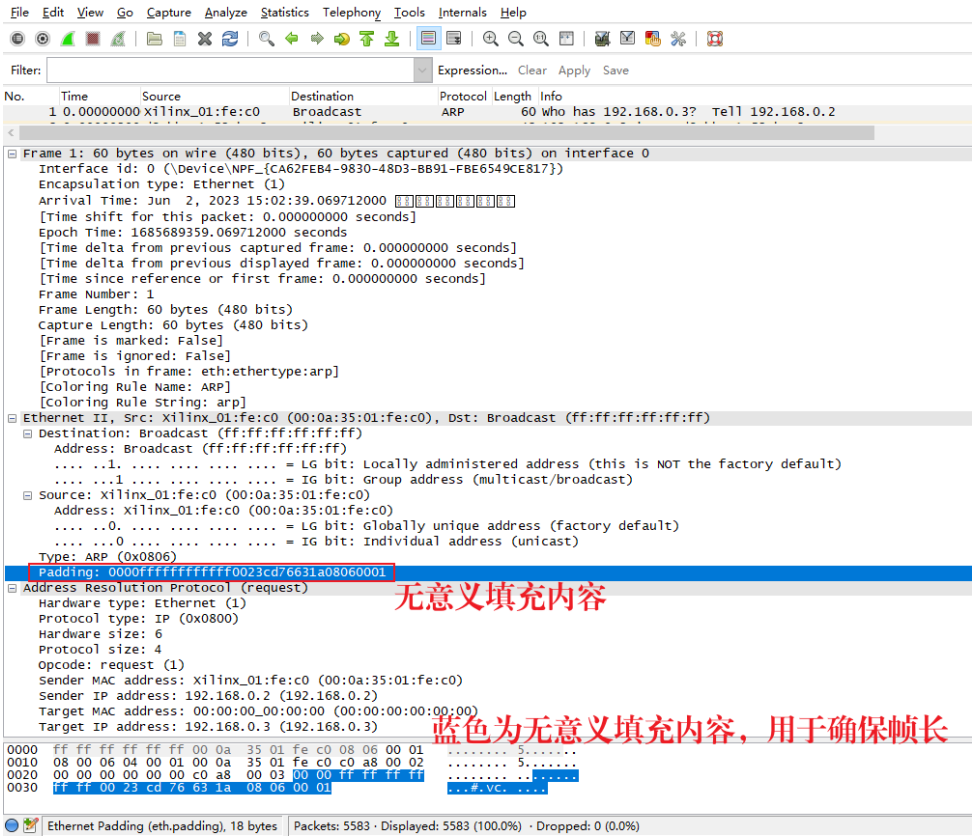


图 49-8 ARP 请求包帧内容

图中最后的蓝色字样部分为软件为保证整个数据帧长度（不含 FCS 字段）不小于 64 个字节而填充的无意义内容。

49.6引入 CRC 校验机制

现代数据通信要求信息传输具有高度可靠性，即误码率要足够低。然而，数据信号在传输过程中不可避免地会受到噪声干扰，或者信道不理想，从而造成的码间干扰而产生差错，即出现误码。我们通常是通过一个校验码去判断接收的消息是否出现差错。信息在发送时，发送端会通过某个算法得到一个值，这个值被称为校验码。通信时发送端在发送消息时将校验码加在信息末端，接收端以相同的算法得到一个校验码，将发送端和接收端的校验码进行对比来判断是否正确接收到了信息。我们在学习串口通信时，可能对奇校验位有所了解。我们以串口通信中的奇校验为例，如果传输的数据中 1 的个数为奇数则为奇校验，校验码为 0；传输的数据中 1 的个数为偶数则为偶校验，校验码为 1。串口通信时发送端传输的数据为：00100011，传输数据中的 1 各位相加结果为 3，所以奇校验为 0。

```
原始消息 ：00100011
发送端   ：00100011  0  （0 为校验码）
接收端   ：11001000  0  （0 为校验码）
```

奇校验这种校验方法虽然很简单，但是这种校验方法却有很大的误码率。接收端可能在接收的过程中受到某些干扰，接收端接收的数据为 11001000。数据中 1 的个数仍然为奇数，得到的奇校验位仍为 0。虽然校验通过，但是数据已经发生了改变。

假设，我们将收发两端的消息以字节的形式进行传输，信息的组成方式均为原始信息+校验码。将原始信息按字节顺序依次对 256（十进制）进行求模运算（当除数被除数都为正数时，求模运算就是取余运算），然后将得到的值再相加最终得到校验码。收发两端的信息以十进制形式传输。

```
原始消息 ：  6 23 4
发送端   ：  6 23 4  33  （33 为校验码）
接收端   ：  6 23 4  33  （33 为校验码）
```

由上面的例子可以看出发送端和接收端传输的信息不一致，但是校验码没有发生改变，仅凭上述简单的方式去校验信息是不可靠的。上文中提到传输的数据均以字节的方式，一个字节也就是八比特能够表达的最大十进制为255，对 256 取余运算后值的范围为 0-255。我们对上述传输数据发生变化，校验码不变的事件进行概率运算。传输数据发生变化，校验码不变为事件 A，三个字节对 256 求模后的可能数值的范围为 0-255 即 256 种，校验码的范围也为一个字节即 0-255 即 256 种。

$$P(A)=\frac{256*256*256}{256*256*256*256} \tag{1}$$

上述中分子 256*256*256 为三个字节数据传输时出错的情况，由于校验码总是由错误数据算出来的，而且为一个字节所以总的可能性为 256*256*256*256，所以事件 A 的概率为$\frac{1}{256}$。由上面两个例子可见，一个好的校验方法配合信号的编码方式可以大大提高通信的稳定性。下面我们将为大家介绍 CRC 校验，CRC 校验又被称为循环冗余校验（Cyclic Redundancy Check,CRC）。CRC 校验计算速度快，检错能力强，易用于编码器等硬件电路，以太网实验中使用的就是 CRC32 校验。

49.6.1CRC 校验原理

CRC 校验原理总的来说非常简单。它的根本思想就是利用线性编码原理，

在数据发送端将要传输的 K 位信息码后面根据一定的规则产生一个校验码即 CRC 校验码 R 位，从而形成一个长度为（K+R）数据帧。在接收端根据信息码和校验码之间遵循的规则进行检验，从而确定数据在传输过程中是否出现差错。

数据发送端和接收端事先约定一个多项式 G(x)作为除数多项式，G(x)的最高次幂为 r。将发送端将要发送 K 位信息码作为多项式 M(x)的系数，将待编码的 K 位数据编码成 M(x)，多项式 M(x)表达式为：

$$\boldsymbol{M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x^1 + m_0x^0} \quad (2)$$

对于 R 位的 CRC 校验，校验的产生过程为：

将 M(x)左移 R 位然后除以一个 G(x)多项式（模 2 除法），所得的余数就是 CRC 校验码，即

$$\frac{M(x)*x^R}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \quad (3)$$

上式中 R(x)就是 CRC 校验码，其中 G(x)为生成多项式。

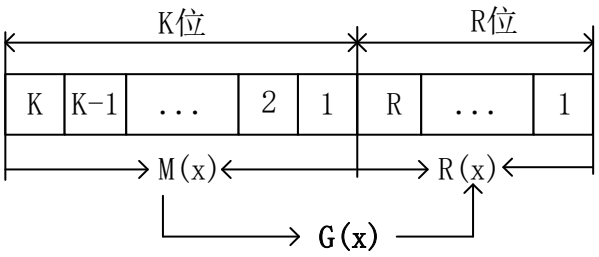


图 49-9 带 CRC 的数据帧

对于数据接收端在接收到信息序列 M(x)后，M(x)模二整除 G(x)后所得的余数等于接收的 R(x)后就没有出现误码。

G(x)的通用表达式为

$$\boldsymbol{G(x) = x^R + g_{R-1}x^{R-1} + g_{R-2}x^{R-2} + \cdots + g_2x^2 + g_1x + 1} \quad (4)$$

R(x)多项式的二进制系数就是 CRC 校验码，R(x)系数的计算方法为：在 K 位信息字段后面添加 R 个 0（就是将数据位左移 R 位），再模二除 G(x)多项式系数对应的二进制序列，得到的余数就是 R(x) 系数对应的二进制序列。R(x) 系数对应的二进制序列共有 R-1 位，若不足则在高位补 0。

举个例子，假设 M(x)多项式对应的二进制序列为 1010001101，G(x)多项式系数对应的二进制序列为 11010。为 5，在 M(x)多项对应的二进制序列后面添加 5 个 0，再对 G(x)多项式系数对应的二进制序列做模 2 除法，得到余数 R(x)多项式的二进制系数为 01110，所以实际发送的二进制序列为 101000110101110。计算方式如图 49-10 所示。

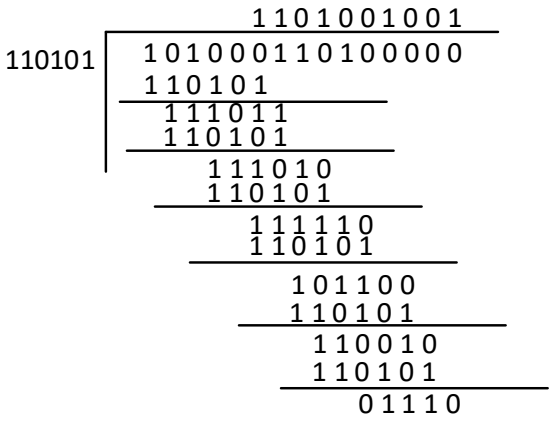


图 49-10 模 2 计算方式

用公式讲解 CRC 编码过程如下：

设待校验的信息码有 K 位，即：M(x)=m_(k-1)，m_(k-2)，…m_1，m_0，多项式 M(x)可表示为：

$$\boldsymbol{M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x^1 + m_0x^0} \tag{5}$$

用多项式 G(x)的最高次幂 R 对应的 x^R 乘以 M(x)将得到

$$\boldsymbol{x^R M(x) = m_{k-1}x^{k-1+R} + m_{k-2}x^{k-2+R} + \cdots + m_1x^{1+R} + m_0x^{0+R}} \tag{6}$$

将$\boldsymbol{x^R M(x)}$模 2 除以 G(x)，得到多项式商为 A(x)，余数为 R(x)即

$$\boldsymbol{A(x)G(x) = x^R M(x) + R(x)} \tag{7}$$

余数多项式 R(x)可表示为

$$\boldsymbol{R(x) = r_{R-1}x^{R-1} + r_{R-2}x^{R-2} + \cdots r_1x^1 + r_0x^0} \tag{8}$$

将式（6）和式（8）代入式（7）得

$$\boldsymbol{A(x)G(x) = m_{k-1}x^{k-1+R} + m_{k-2}x^{k-2+R} + \cdots + m_1x^{1+R} + m_0x^{0+R} + r_{R-1}x^{R-1} + r_{R-2}x^{R-2} + \cdots r_1x^1 + r_0x^0} \tag{9}$$

式（9）对应得码组就是 K+R 位，即：

$$\mathrm{N}=(\boldsymbol{m_{k-1} + m_{k-2} + \cdots + m_1 + m_0 + r_{R-1} + \cdots r_1 + r_0}) \tag{10}$$

上述公式给了一个较为清晰得数学公式推导过程，$\boldsymbol{m_{k-1}}$，$\boldsymbol{m_{k-2}}$，…$\boldsymbol{m_1}$，$\boldsymbol{m_0}$的二进制系数为 K 位信息码；$\boldsymbol{r_{R-1} + \cdots r_1 + r_0}$的二进制系数为 R 位校验码。在数据接收端，将接收到的 K+R 位码模二除以相同的多项式 G(x)。根据式（7）计算如果产生的余数为0，则接收端接收的数据正确无误，否则就认为数据在传输的过程中出现差错。以太网 802.3 协议对 CRC 校验的多项式 G(x)进行了规定，规定在以太网的收发中校验方式只能用 CRC32，规定了 G(x)的表达式为

$$\boldsymbol{G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 +}$$

$$x^2 + x + 1 \quad (11)$$

将上述表达式转换成二进制序列为：1_0000_0100_1100_0001_0001_1101_1011_0111。

以上我们讲解了 CRC 公式的推导过程，下面我们通过 CRC 参数模型来为大家展示如何通过模 2 计算方式得到一个正确的 CRC 校验值。

计算得到一个正确的 CRC 校验值需要知道 CRC 模型参数，不同的 CRC 模型参数也不同。一个完整的 CRC 模型包含以下信息：WIDTH，POLY，INIT，XOROUT，REFIN，REFOUT 等等如图 49-11 所示。

CRC算法名称	多项式公式	WIDTH	POLY	INIT	XOROUT	REFIN	REFOUT
CRC-4/ITU	x4 + x + 1	4	03	00	00	TRUE	TRUE
CRC-5/EPC	x5 + x3 + 1	5	09	09	00	FALSE	FALSE
CRC-5/ITU	x5 + x4 + x2 + 1	5	15	00	00	TRUE	TRUE
CRC-5/USB	x5 + x2 + 1	5	05	1F	1F	TRUE	TRUE
CRC-6/ITU	x6 + x + 1	6	03	00	00	TRUE	TRUE
CRC-7/MMC	x7 + x3 + 1	7	09	00	00	FALSE	FALSE
CRC-8	x8 + x2 + x + 1	8	07	00	00	FALSE	FALSE
CRC-8/ITU	x8 + x2 + x + 1	8	07	00	55	FALSE	FALSE
CRC-8/ROHC	x8 + x2 + x + 1	8	07	FF	00	TRUE	TRUE
CRC-8/MAXIM	x8 + x5 + x4 + 1	8	31	00	00	TRUE	TRUE
CRC-16/IBM	x16 + x15 + x2 + 1	16	8005	0000	0000	TRUE	TRUE
CRC-16/MAXIM	x16 + x15 + x2 + 1	16	8005	0000	FFFF	TRUE	TRUE
CRC-16/USB	x16 + x15 + x2 + 1	16	8005	FFFF	FFFF	TRUE	TRUE
CRC-16/MODBUS	x16 + x15 + x2 + 1	16	8005	FFFF	0000	TRUE	TRUE
CRC-16/CCITT	x16 + x12 + x5 + 1	16	1021	0000	0000	TRUE	TRUE
CRC-16/CCITT-FALSE	x16 + x12 + x5 + 1	16	1021	FFFF	0000	FALSE	FALSE
CRC-16/X25	x16 + x12 + x5 + 1	16	1021	FFFF	FFFF	TRUE	TRUE
CRC-16/XMODEM	x16 + x12 + x5 + 1	16	1021	0000	0000	FALSE	FALSE
CRC-16/DNP	x16 + x13 + x12 + x11 + x10 + x8 + x6 + x5 + x2 + 1	16	3D65	0000	FFFF	TRUE	TRUE
CRC-32	x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1	32	04C11DB7	FFFFFFFF	FFFFFFFF	TRUE	TRUE

图 49-11 CRC 参数模型

（1）WIDTH:表示宽度，即生成的 CRC 数据位宽，如 CRC32，生成的 CRC 为 32 位。

（2）POLY：表示极性，省略最高位 1 后，如 CRC32 的多项式表示为$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$，二进制系数为 1_0000_0100_1100_0001_0001_1101_1011_0111，省略最高位后多项式的值转换成十六进制为 0x04C11DB7。

（3）INIT：表示的是 CRC 初值。

（4）REFIN:该属性只有两个值 TRUE 或者 FALSE，如果是 TRUE 就将原始数据按字节进行翻转。如原始数据为 0x44332211，对应的二进制数据为 0100_0100_0011_0011_0010_0010_0001_0001 翻转后的二进制数据为 1000_1000_0100_0100_1100_1100_0010_0010。若该属性为 FALSE 则不必对数据进行翻转。

（5）REFOUT: 该属性只有两个值 TRUE 或者 FALSE，该属性是决定模 2 计算后得到的 CRC 余数值是否需要翻转，如果该值为 TRUE 则需要对

CRC 余数值进行翻转，否则不必翻转。

（6）XOROUT：该属性是将计算得到的 CRC 余数值与该属性对应的值做异或运算。

49.6.2CRC8 计算

行文至此，大家已经对 CRC 模型各个参数有了初步认识。下面我们来验证一下手动计算得到的 CRC 的值与我们提供的软件计算出的值是否一致。

例：原始数据：0x11，使用 CRC8 算法模型，求 CRC8 的值。

（1）INIT=0x00

（2）REFIN=FALSE

（3）REFOUT= FALSE

（4）XOROUT=0x00

有了上面的参数计算步骤为：

（1）原始数据为 0x11，对应的二进制为 0001_0001。

（2）INIT=0x00，将原始数据 0x11 的高 8 位与 0x00 对应的二进制做异或运算得到的值为 0001_0001。

（3）REFIN= FALSE，对原始数据和 INIT 的值做异或运算后的值不用再按字节翻转，仍为 0001_0001。

（4）将翻转得到的值左移 8 位，其实就是将（3）步骤得到的值后面添加 8 个 0。

（5）经过以上步骤对原始数据处理完毕，将原始数据经过处理后得到的数据，对 CRC8 多项式对应的二进制系数进行模 2 相除，求得余数对应的值。经过处理后的原始数据：0001_0001_0000_0000；CRC8 多项式对应的二进制数为： 1_0000_0111，模 2 相除后取低 8 位为：0111_0111。

```
                    10001
100000111 | 0001000100000000
            100000111
            0000101110000
                 100000111
                 001110111
```

图 49-12 CRC8 手动计算

（6）XOROUT=0x00，将模 2 得到的余数值为 0111_0111，将该值与 0x00 异或，得到值 0111_0111。

（7）REFOUT= FALSE，则（6）步骤得到的值 0111_0111 不必进行翻转，所以最终 CRC 的值仍为 0111_0111，该值对应的十六进制为 0x77。

上面是我们手动计算 CRC8 的结果，我们使用软件来计算一下 CRC8 的值。

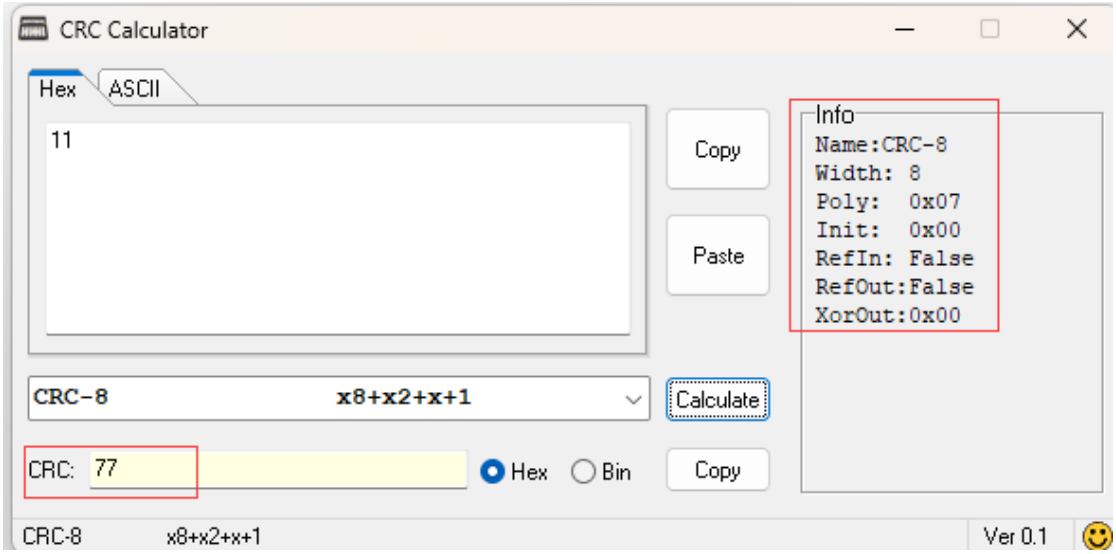


图 49-13 CRC 软件计算

上图左侧信息栏中的信息与图 49-13 中信息一致且 CRC8 软件计算的值与我们手动计算的值一致。

49.6.3CRC32 计算

下面我们来验证一下手动计算得到的 CRC32 的值与我们提供的软件计算出的值是否一致。

例：原始数据：0x44332211，使用 CRC32 算法模型，求 CRC32 的值。

CRC 参数模型表可知：

（1）INIT=0xFFFFFFFF

（2）REFIN=TRUE

（3）REFOUT=TRUE

（4）XOROUT=0xFFFFFFFF

有了上面的参数，计算的步骤为：

（1）原始数据为 0x44332211，对应的二进制为 0100_0100_0011_0011_0010_0010_0001_0001。

（2）INIT=0xFFFFFFFF，将原始数据 0x44332211 的高 32 位与 0xFFFFFFFF 对应的二进制做异或运算得到的值为 1011_1011_1100_1100_1101_1101_1110_1110。

（3）REFIN= TRUE，对原始数据和 INIT 的值做异或运算后的值再按字节进行翻转，得到的值为 1101_1101_0011_0011_1011_1011_0111_0111。

（4）将翻转得到的值左移 32 位，其实就是将（3）步骤得到的值后面添加 32 个 0。

经过以上步骤对原始数据处理完毕，将原始数据经过处理后得到的数据，对 CRC32 多项式对应的二进制系数进行模 2 相除，求得余数对应的值。经过处理后的原始数据：

1101_1101_0011_0011_1011_1011_0111_0111_0000_0000_0000_0000_0000_0000_0000_0000_00000

CRC32 多项式对应的二进制数为：

11_0000_0100_1100_0001_0001_1101_1011_0111

模 2 相除后取低 32 位为：0100_1001_0100_1011_0100_0111_1000_0100

下图 49-14 为 CRC32 手动计算演示：

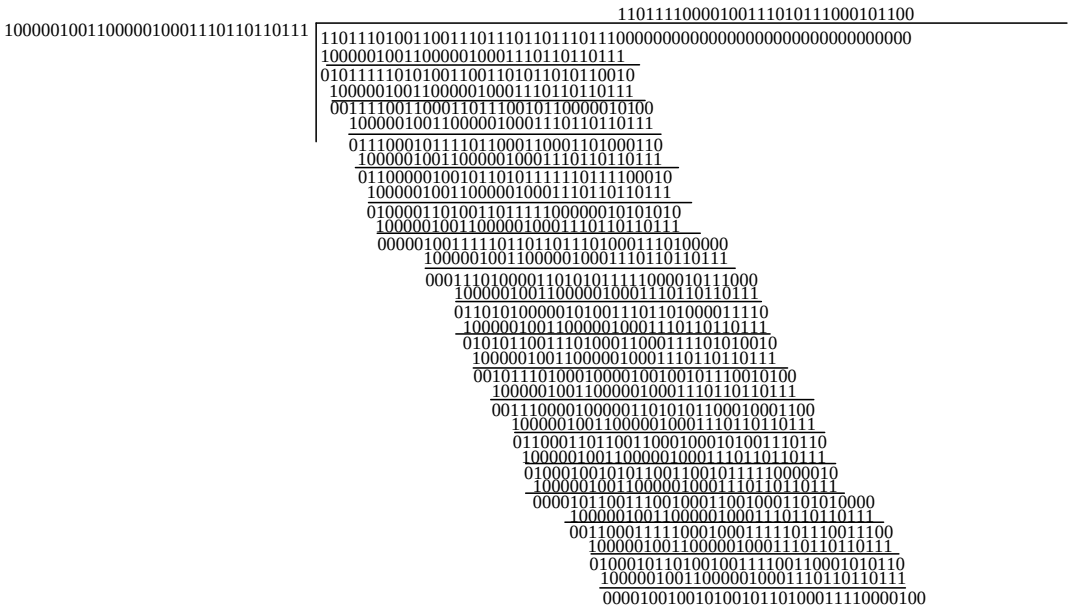


图 49-14 CRC32 手动计算

（5）XOROUT=0xFFFFFFFF，将模 2 得到的余数值为 0100_1001_0100_1011_0100_0111_1000_0100，将该值与 0xFFFFFFFF 异或，得到值 1011_0110_1011_0100_1011_1000_0111_1011。

（6）REFOUT=TRUE，则将（6）步骤得到的值 1011_0110_1011_0100_1011_1000_0111_1011 翻转后，就会得到最终的 CRC 的值 1101_1110_0001_1101_0010_1101_0110_1101，该值对应的十六进制为 0XDE1D2D6D。

上面是我们手动计算 CRC32 的结果，我们使用软件来计算一下 CRC32 的值。

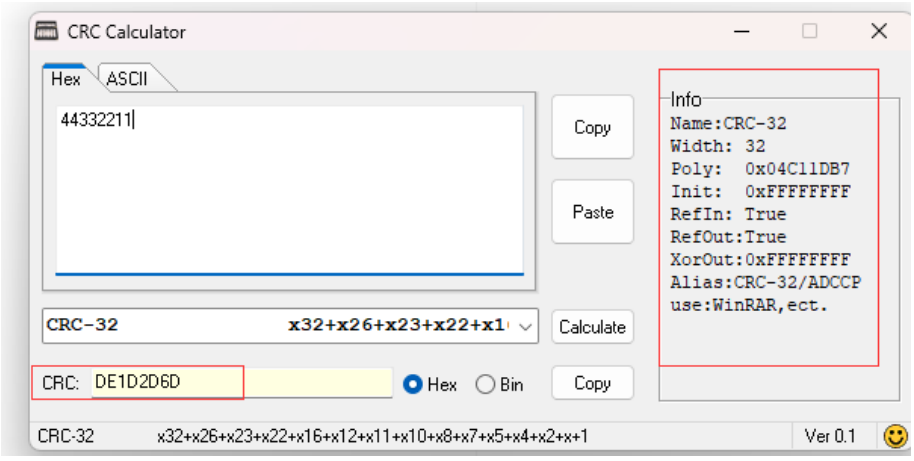


图 49-15 CRC32 软件计算

上图左侧信息栏中的信息与中信息一致且 CRC32 软件计算的值与我们手动计算的值一致。

CRC32 是一种循环冗余校验算法，通过对数据进行处理生成一个 32 位的校验值，这个校验值可以用于检测数据传输过程中的错误。在 CRC32 算法中，生成的校验值是与输入数据相关的，即输入不同的数据会有不同的 CRC32 的值。当使用 CRC32 校验时任意值经过校验后将得到的四个字节的 CRC32 的值按字节高低位放反添加在数据后面，相当于对数据进行了去余运算。CRC32 校验值在附加到数据中时，按照字节顺序放反是因为 CRC32 算法使用的是逆序处理方式，CRC32 算法以比特位为单位处理数据，并且在计算过程中按照逆序对每一个比特位进行操作。这意味着输入数据的每一个比特位都会对 CRC32 值产生影响。为了保持与 CRC32 算法的处理方式一致，将 CRC32 校验值附加到数据中时，需要按照字节顺序进行反转。这样可以确保 CRC32 算法在验证数据时能够按照正确的顺序处理校验值。请注意，字节顺序反转是针对 CRC32 算法而言的，其他类型的校验值可能不需要进行反转操作。因此，在附加 CRC 校验值时，确保按照相应的规则处理数据和校验值是十分重要的。最后将这个重新拼接的数据进行 CRC32 校验时，相当于对 0x0000 做 CRC32 校验得到了定值 0x 2144DF1C。

49.7使用 CRC 计算软件计算 CRC 校验值

完整的以太网数据包结尾是 4 个字节的帧校验（FCS）字段，即 CRC32 值。而在上述的整个组包和抓包过程中，都没有出现 CRC 校验字段的内容。而我们使用 FPGA 发包需要计算 CRC 校验字段，CRC 校验字段的产生可以使用软件或硬件的形式产生，由于 CRC32 的计算本身又涉及到了较多的理论知识，实现也有一定的难度。本节实验为了降低难度，先暂时回避该算法，使用 CRC 计算软件手动计算出数据报的 CRC32 值，然后使用该计算得到的 CRC32 值来让 FPGA 的网卡先体验一波发包，先看到直观的现象。相信这也是大家学习的最爱。

计算以太网数据的 CRC 字段，可以使用专用 CRC 校验工具，该工具名为“CRC_Calc+v0.1.exe”。

打开 CRC_Calc+v0.1.exe”校验软件，在计算多项式中选择倒数第二个多项式 CRC-32 作为进行 CRC32 计算的多项式，该多项式一般用于 WinRAR 压缩软件和以太网传输中。

当选定一个计算多项式后，右侧会展示出该计算方法对应的一些参数的具体值，对于这些参数的意义介绍如下：

- Name：参数模型名称。
- Width：宽度，即 CRC 比特数。
- Poly：生成项的简写，以 16 进制表示。例如：CRC-32 即是 0x04C11DB7，忽略了最高位的"1"，即完整的生成项是 0x104C11DB7。
- Init：这是算法开始时 CRC 的初始化预置值，十六进制表示。
- Refin：待测数据的每个字节是否按位反转，True 或 False。
- Refout：在计算后之后，异或输出之前，整个数据是否按位反转，True 或 False。
- Xorout：计算结果与此参数异或后得到最终的 CRC 值。

将之前保存的记事本中的数据内容复制粘贴到计算软件的文本框中，然后点击 Calculate 即可计算出这串数据的 CRC32 值，如下图 49-16 所示。

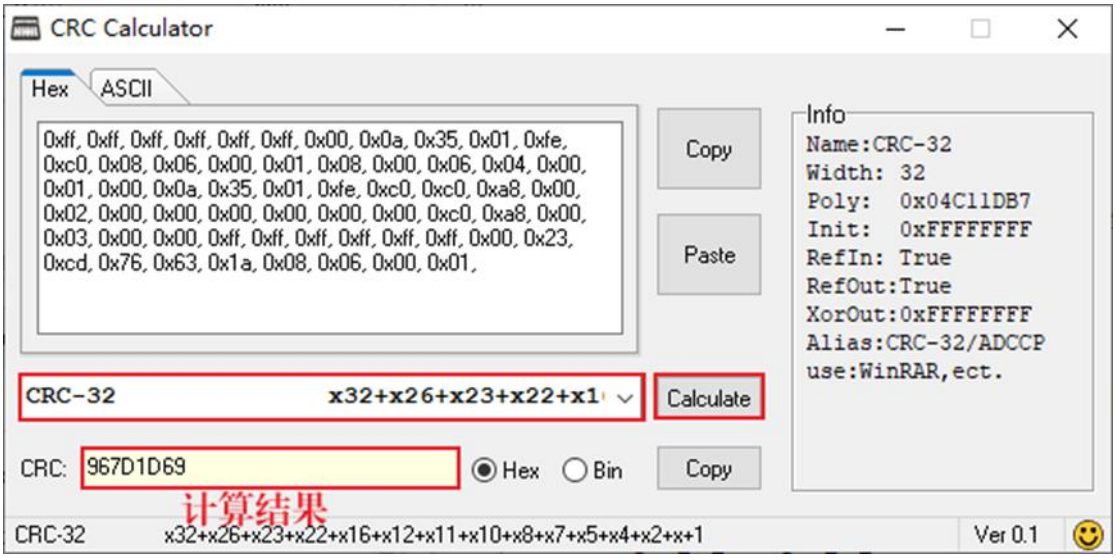


图 49-16 CRC 值

注意，小兵以太网测试仪在保存得到文本文件时，是以 16 个数据为一行，然后换行继续输出的，所以直接从该文本中赋值内容会因为换行符号的存在而导致本计算软件计算结果异常，结果全为 0。所以如果你计算出来的结果为 0，可以采取下列方式解决：

- 安装有 notepad++软件的，使用 notepad++软件打开该文件，能自动显示换行的位置，将文件中所有换行符删除后再复制到 CRC 计算软件中。
- 如果没有安装 notepad++软件，可以在 Gowin 中新建一个 verilog 文件，然后将该文件中的内容复制到该 Gowin 文件中，也会自动显示换行，

将文件中所有换行符删除后再复制到 CRC 计算软件中。

需要注意的是，CRC 软件计算结果与以太网使用时的关系为：

CRC 计算软件结果：低字节在前，高字节在后；

以太网使用时：高字节在前，低字节在后，即应以字节位为单位调换 4 个字节的高低顺序，如 BE7C1CBB 应该调换为 BB1C7CBE。

```
//整个数据包 CRC 校验值，本例中使用 CRC 计算软件计算得出。
parameter CRC = 32'h967D1D69;

wire [31:0]CRC_result;

//调整 CRC 校验字字节顺序以符合以太网数据包校验格式
assign CRC_result = {CRC[7:0],CRC[15:8],CRC[23:16],CRC[31:24]};
```

至此，ARP 以太网数据报的所有字段的内容就都得到了。接下来，就可以使用该数据报内容来进行 FPGA 设计以太网的初探了。

49.8以太网 ARP 帧发包实例（手动 CRC）

本节实验将基于高云开发板上的以太网接口，设计一个能够发送 ARP 帧的 FPGA 系统。其中以太网包和 ARP 包采用分层组包的形式。即底层为以太网帧（MAC 帧）生成层，上层为 ARP 包生成层。工程中包含了两个逻辑设计文件，其中：

eth_send.v 为以太网 MAC 帧组包逻辑，完成以太网 MAC 帧的建立和发送。

eth_send_test.v 为 ARP 包组包工具，并调用 eth_send 完成完整的 ARP 报文的发送。

49.8.1MAC 帧生成和发送

eth_send 模块实现的是 MAC 帧发送的核心内容。其实该内容也非常简单，就是一个序列机，按照顺序将 MAC 帧中的前导码、分隔符、目的 MAC 地址、源 MAC 地址、长度/类型发出，然后开始根据上层传入的数据长度（data_length）值，从上层 fifo 中读取对应个字节的数据发出，最后再发出 4 字节的 CRC 字段即可。

下述代码为采用序列机方式实现 GMII/RGMII 接口以太网数据发送的代码内容。可以看到，每个计数值输出当前阶段需要输出数据的其中 8 位数据，前导码占用 7 个计数值，分隔符占用 1 个计数值。

```
/*
    序列机部分，根据不同的时刻，切换 MII 接口数据线上的内容，
    包含前导码、分隔符、目的地址、源地址、以太网帧类型/长度、
    数据段数据、结尾 CRC 校验值
*/

always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
    gmii_tx_data <= #1  8'd0;
else begin
    case(cnt)
        1, 2, 3, 4, 5, 6, 7:
            gmii_tx_data <= #1  8'h55;  //前导码

        8: gmii_tx_data <= #1  8'hd5;   //分隔符

        //目的 MAC 地址
        9: gmii_tx_data <= #1  r_des_mac[47:40];
        10: gmii_tx_data <= #1  r_des_mac[39:32];
        11: gmii_tx_data <= #1  r_des_mac[31:24];
        12: gmii_tx_data <= #1  r_des_mac[23:16];
        13: gmii_tx_data <= #1  r_des_mac[15:8];
        14: gmii_tx_data <= #1  r_des_mac[7:0];

        //源 MAC 地址
        15: gmii_tx_data <= #1  src_mac[47:40];
        16: gmii_tx_data <= #1  src_mac[39:32];
        17: gmii_tx_data <= #1  src_mac[31:24];
        18: gmii_tx_data <= #1  src_mac[23:16];
        19: gmii_tx_data <= #1  src_mac[15:8];
        20: gmii_tx_data <= #1  src_mac[7:0];

        //以太网帧类型/长度
        21: gmii_tx_data <= #1  r_type_length[15:8];
        22: gmii_tx_data <= #1  r_type_length[7:0];

        //发送数据
        23: gmii_tx_data <= #1  fifo_rddata;

        //发送 CRC 校验结果
        24: gmii_tx_data <= #1  CRC_Result[31:24];
        25: gmii_tx_data <= #1  CRC_Result[23:16];
```

```
        26: gmii_tx_data <= #1  CRC_Result[15:8];
        27: gmii_tx_data <= #1  CRC_Result[7:0];

        28: gmii_tx_data <= #1  8'd0;
        default: gmii_tx_data <= #1  8'd0;
    endcase
end
```

在代码中可以看到，对于 GMII/RGMII 接口，在序列计数器计数值为 23 的情况下，便开始发送 MAC 层的数据和填充字段了，所以根据此计数值来产生对上层 FIFO 的读请求信号。如下所示：

```
//帧中数据发送使能信号
assign en_tx_data = (cnt == 23) && (data_num > 1);

//待发送数据计数器，每发送一个数据段中的数据，本计数器减 1.
always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
    data_num <= #1  0;
else if(tx_go)
    data_num <= #1  data_length;
else if(en_tx_data)
    data_num <= #1  data_num - 1'b1;
else
    data_num <= #1  data_num;
```

由于在发送整个数据和填充字段的过程中序列计数器需要保持在 23 不变，所以使用上述产生的 en_tx_data 信号来控制计数器，让其在发送数据字段的时候停止变化，该部分内容如下所示。

```
//根据发送启动信号产生内部发送使能信号
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    en_tx <= #1  1'd0;
else if(tx_go)
    en_tx <= #1  1'd1;
else if(cnt >= 53)//一帧数据发送完成，清零发送使能信号
    en_tx <= #1  1'd0;

//主序列机计数器
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    cnt <= #1  6'd0;
else if(en_tx)begin
    if(!en_tx_data)
```

```
        cnt <= #1  cnt + 6'd1;
    else    //在发送整个帧中的数据部分时，计数器暂停
        cnt <= #1  cnt;
end
else
    cnt <= #1  6'd0;
```

49.8.2ARP 数据包生成和发送

在 eth_send_test 模块中例化调用以太网帧发送模块 eth_send，并将目的地址设置为广播地址 ffffffff、源地址为板卡地址。

数据长度为 ARP 数据包的长度，这里为 46 字节。协议类型为 ARP(0806）。

```
eth_send eth_send(
    .rst_n(rst_n),
    .tx_go(tx_go),
    .data_length(12'd46),
    .des_mac(48'hff_ff_ff_ff_ff_ff),
    .src_mac(48'h00_0a_35_01_fe_c0),
    .type_length(16'h08_06),
    .CRC_Result(CRC_result),
    .fifo_rdreq(fifo_rdreq),
    .fifo_rddata(fifo_rddata),
    .fifo_rdclk(fifo_rdclk),
    .mii_tx_clk(mii_tx_clk),
    .mii_tx_en(mii_tx_en),
    .mii_tx_er(mii_tx_er),
    .mii_tx_data(mii_tx_data)
);
```

tx_go 为单次发送使能信号，每个 tx_go 信号的高脉冲启动一次发送。这里使用一个 24 位计数器进行计数，每计数一轮，tx_go 有效一次，即启动一次发送。整个计数周期在 MII 模式下，由于时钟频率是 25M，所以发送周期为 671ms 左右（40ns *2^24），在 RGMII 模式下，由于时钟频率为 125M，所以发送周期为 134ms 左右（8ns *2^24）。以下以千兆速率下 RGMII 接口模式为例。

```
//发送间隔计数器
reg [23:0]cnt;

always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
    cnt <= #1 0;
else //计数器自增，不考虑溢出，接受溢出自动清零
    cnt <= #1 cnt + 1'b1;
```

```
//24 位 cnt 计满一次启动一次发送，该时间大约为 134ms
assign tx_go = (cnt == 24'd1);
```

设计这个发送速率控制逻辑的目的在于降低 FPGA 每秒发包的次数，避免发包太快给 PC 网卡带来负担。发包太多太快也没有任何意义。

以太网帧发送模块发送的数据段部分采用 fifo 接口，本例中仅为测试作用，因此使用查找表加自动地址累加的方式代替 fifo 输出 arp 数据包给以太网帧发送模块。在实际使用 fifo 时，需要将 fifo 设置为“First-Word Fall-Through”模式。

```
//计数发送数据个数，用于产生对应的数据
always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
  data_cnt <= #1 12'd0;
else if(fifo_rdreq)
  data_cnt <= #1 data_cnt + 1'b1;
else
data_cnt <= #1 12'd0;
```

以下使用查找表形式依次输出 arp 协议的报文内容。

```
//ARP 包数据段
always@(*)
begin
  case(data_cnt)
    //hdwr type
    00: fifo_rddata = 8'h00;
    01: fifo_rddata = 8'h01;

    //protocol type
    02: fifo_rddata = 8'h08;
    03: fifo_rddata = 8'h00;

    //hdwr size
    04: fifo_rddata = 8'h06;

    //protocol size
    05: fifo_rddata = 8'h04;

    //opcode
    06: fifo_rddata = 8'h00;
    07: fifo_rddata = 8'h01;

    //sender mac
    08: fifo_rddata = 8'h00;
```

```
    09: fifo_rddata = 8'h0a;
    10: fifo_rddata = 8'h35;
    11: fifo_rddata = 8'h01;
    12: fifo_rddata = 8'hfe;
    13: fifo_rddata = 8'hc0;

    //sender ip : 192.168.0.2
    14: fifo_rddata = 8'hc0;//192
    15: fifo_rddata = 8'ha8;//168
    16: fifo_rddata = 8'h00;//0
    17: fifo_rddata = 8'h02;//2

    //target mac
    18: fifo_rddata = 8'h00;
    19: fifo_rddata = 8'h00;
    20: fifo_rddata = 8'h00;
    21: fifo_rddata = 8'h00;
    22: fifo_rddata = 8'h00;
    23: fifo_rddata = 8'h00;

    //target ip : 192.168.0.3
    24: fifo_rddata = 8'hc0;//192
    25: fifo_rddata = 8'ha8;//168
    26: fifo_rddata = 8'h00;//0
    27: fifo_rddata = 8'h03;//3

    //填充字段，以使整个数据帧长度达到 64 字节
    28: fifo_rddata = 8'h00;
    29: fifo_rddata = 8'h00;
    30: fifo_rddata = 8'hff;
    31: fifo_rddata = 8'hff;
    32: fifo_rddata = 8'hff;
    33: fifo_rddata = 8'hff;
    34: fifo_rddata = 8'hff;
    35: fifo_rddata = 8'hff;
    36: fifo_rddata = 8'h00;
    37: fifo_rddata = 8'h23;
    38: fifo_rddata = 8'hcd;
    39: fifo_rddata = 8'h76;
    40: fifo_rddata = 8'h63;
    41: fifo_rddata = 8'h1a;
    42: fifo_rddata = 8'h08;
    43: fifo_rddata = 8'h06;
```

```
    44: fifo_rddata = 8'h00;
    45: fifo_rddata = 8'h01;
    default:fifo_rddata = 8'h0;
  endcase
end
```

用户实验时，只需要先将自己的电脑 IP 地址设置为程序中询问的 192.168.0.3，然后将工程编译结果 bit 文件下载到开发板中，在 wireshark 工具中就能连续不断的收到由开发板发出的 arp 请求了。如下图 49-17 所示：

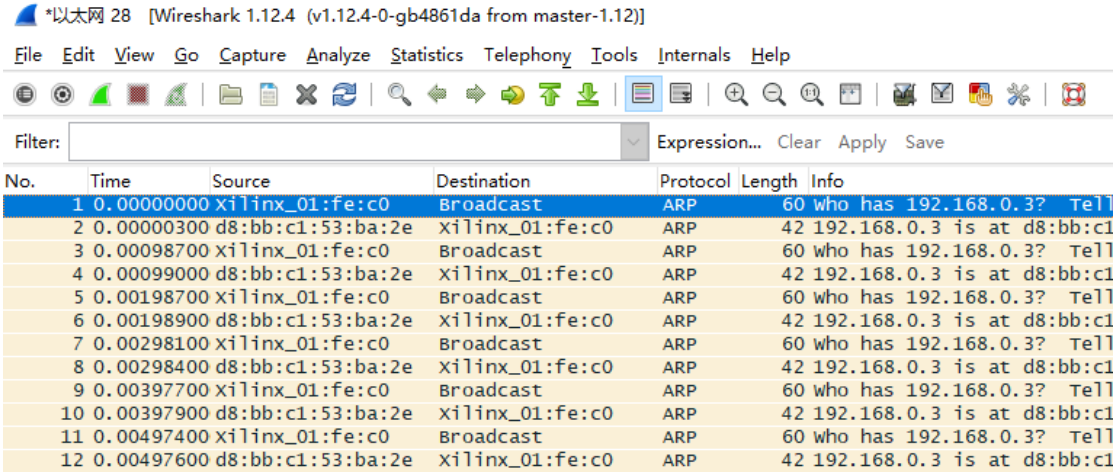


图 49-17 wireshark 抓包

本节实验从应用上来讲，并没有任何实际意义，因为 CRC 校验值都是用软件生成的，数据内容也都是确定的，但是从学习角度，仍旧具有非常重要的价值。

本节实验介绍了 ARP 数据报的格式，并向着使用 FPGA 成功发送数据报的目的出发，过程中介绍了小兵以太网测试仪软件的使用、CRC 计算软件的使用，并提供了 RGMII 千兆接口的 ARP 发包实例，然后介绍使用 Wireshark 软件抓取以太网包和分析的基本流程。为后续继续学习打下了坚实的基础。

49.9以太网 ARP 帧发包实例（自动 CRC）

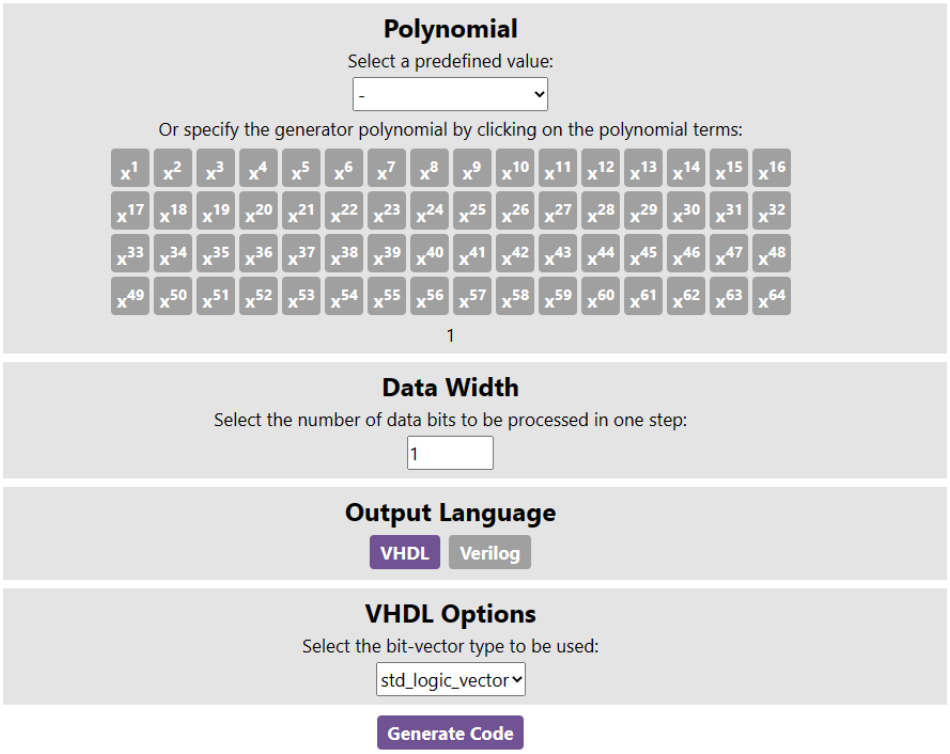
在上一小节中，以太网帧的 FCS 字段使用 PC 端 CRC 校验工具生成，该种方式仅限于以太网帧发送模块设计初期检查协议的数据流控制，当数据报中任何一个字节的内容更改之后，CRC 的值都需要重新计算，这在实际应用中是不现实的。因此本节将引入自动 CRC 校验值的生成。

49.9.1以太网报文的校验字段 FCS 的计算

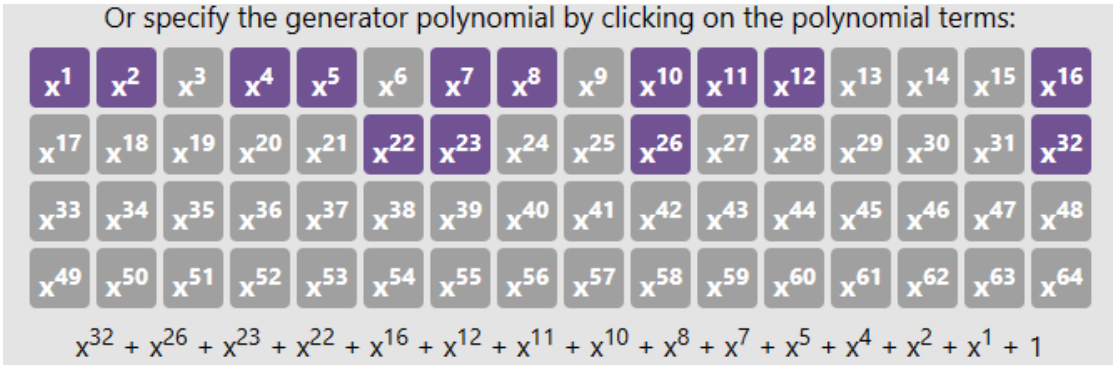
以太网报文校验字段 FCS 采用的是 CRC32 计算，关于 CRC 计算的 Verilog 代码实现已经做的很成熟，网上也有可直接生成 CRC 计算 Verilog 代码的网站。关于 CRC 计算原理感兴趣的可自行网上查找相关资料。这里主要是通过一个网站在线生成 CRC32 的 Verilog 代码，然后通过仿真对代码进行仿真验证。本例以千兆以太网的 RGMII 接口的 8 位数据模式为例进行介绍。

在线生成 CRC32 的 Verilog 代码的网站如下，点击进入网站是如下界面。

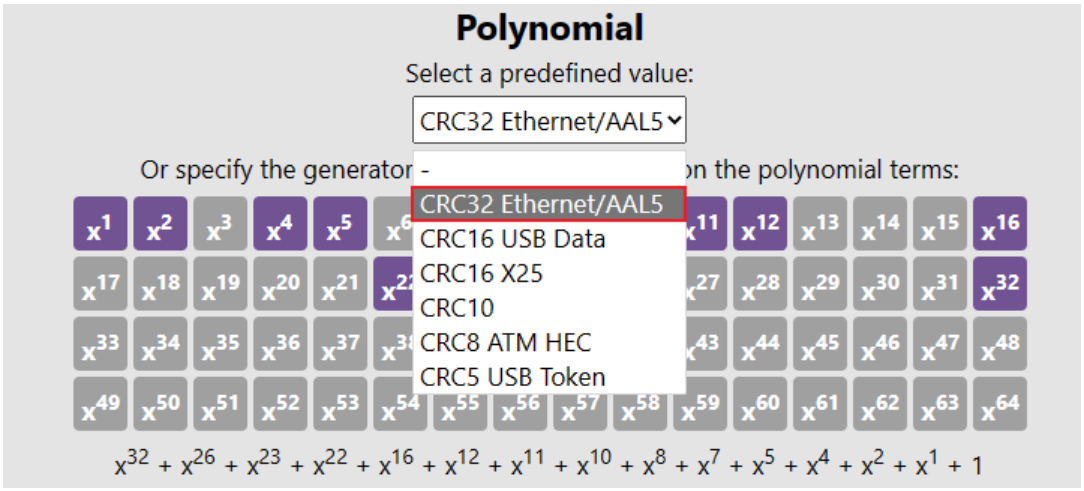
http://www.easics.com/webtools/crctool



（1）CRC 生成多项式的设置，可根据多项式在窗口选择点击多项式中系数为 1 的 x 的幂次方。比如这里的以太网的 CRC32 的多项式为 x32 + x26 + x23 + x22 + x16 + x12 + x11+ x10 + x8 + x7 + x5 + x4 + x2 + x + 1，则只需将 x32、x26 、x23、x22、x16、x12、x11、x10、x8、x7、x5、x4、x2、x 、1 点击选择上，在窗口下方会根据你点击选择的 x 的幂次方会将 CRC 生成公式呈现出来，可与以太网的 CRC 生成多项式进行比较核对，确保多项式的设置没有问题。



（2）这里同样是 CRC 生成多项式的设置，与步骤①不同的地方是这里提供了几个常用的 CRC 生成多项式供选择，对于以太网的 CRC 就直接在下拉框中选择 CRC32 Ethernet/AAL5 即可，如下图所示。这里的不同的选择，同样会在步骤 1 中的窗口中同步更新显示，步骤 2 与步骤 1 都是 CRC 生成多项式的设置，步骤 1 更加灵活，可设置任意的 CRC 生成多项式，而步骤 2 就只能选择几种常见多项式的中的一种。两个地方只要选择一个地方进行设置即可。



（3）设置输入数据流的位宽，如果是 MII 接口，则设置为 4，本次用于 GMII 接口（RGMII 是 GMII 转换得到的，本质还是 GMII）则设置为“8”。

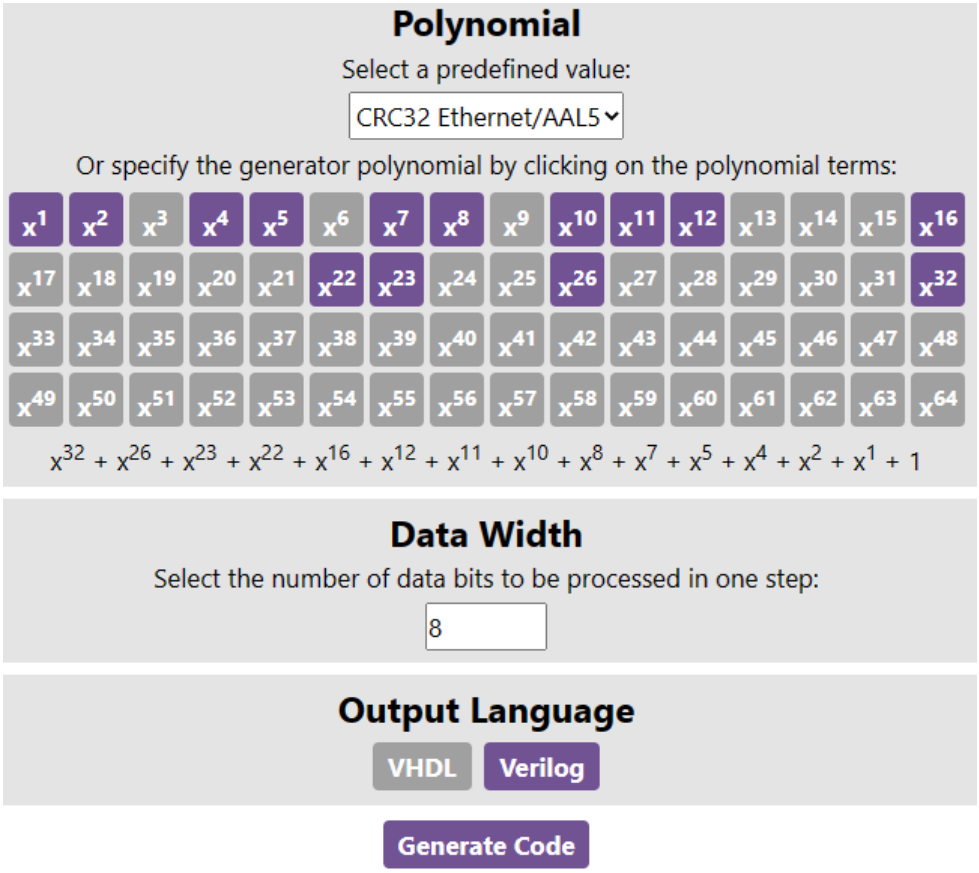
（4）选择生成 HDL 代码的语言，选择“Verilog”。

（5）设置 VHDL 语言下要使用的位矢量类型，步骤 4 选择的是 Verilog 语言，就没有需要设置这个的地方。

（6）下载生成 Verilog 代码的入口，点击可进行下载。

配置完后的界面如下，这个网站设置后有时候更新很慢，需要等待一会时

间。



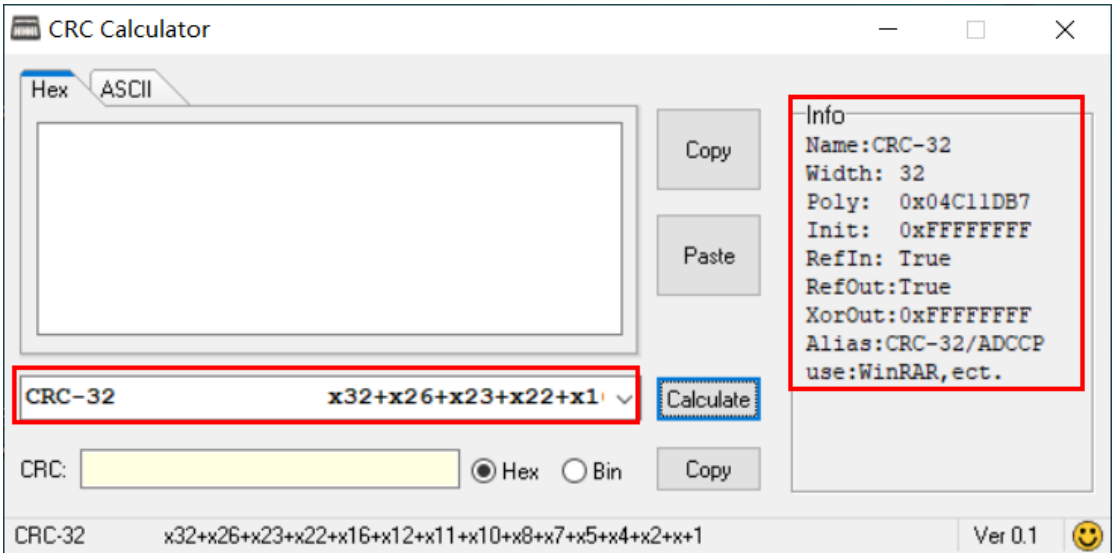
点击 Generate Code，网页会生成并帮我们下载一个命名为 CRC32_D8 的代码文件，生成的具体代码如下。

```
module CRC32_D8;
  // polynomial: x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7
+ x^5 + x^4 + x^2 + x^1 + 1
  // data width: 8
  // convention: the first serial bit is D[7]
  function [31:0] nextCRC32_D8;

    input [7:0] Data;
    input [31:0] crc;
    reg [7:0] d;
    reg [31:0] c;
    reg [31:0] newcrc;
  begin
    d = Data;
    c = crc;

    newcrc[0] = d[6] ^ d[0] ^ c[24] ^ c[30];
    newcrc[1] = d[7] ^ d[6] ^ d[1] ^ d[0] ^ c[24] ^ c[25] ^ c[30] ^ c[31];
    newcrc[2] = d[7] ^ d[6] ^ d[2] ^ d[1] ^ d[0] ^ c[24] ^ c[25] ^ c[26] ^ c[30] ^
c[31];
    newcrc[3] = d[7] ^ d[3] ^ d[2] ^ d[1] ^ c[25] ^ c[26] ^ c[27] ^ c[31];
```

```
    newcrc[4] = d[6] ^ d[4] ^ d[3] ^ d[2] ^ d[0] ^ c[24] ^ c[26] ^ c[27] ^ c[28] ^
c[30];
    newcrc[5] = d[7] ^ d[6] ^ d[5] ^ d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[24] ^ c[25] ^
c[27] ^ c[28] ^ c[29] ^ c[30] ^ c[31];
    newcrc[6] = d[7] ^ d[6] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ c[25] ^ c[26] ^ c[28] ^
c[29] ^ c[30] ^ c[31];
    newcrc[7] = d[7] ^ d[5] ^ d[3] ^ d[2] ^ d[0] ^ c[24] ^ c[26] ^ c[27] ^ c[29] ^
c[31];
    newcrc[8] = d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[0] ^ c[24] ^ c[25] ^ c[27] ^ c[28];
    newcrc[9] = d[5] ^ d[4] ^ d[2] ^ d[1] ^ c[1] ^ c[25] ^ c[26] ^ c[28] ^ c[29];
    newcrc[10] = d[5] ^ d[3] ^ d[2] ^ d[0] ^ c[2] ^ c[24] ^ c[26] ^ c[27] ^ c[29];
    newcrc[11] = d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[3] ^ c[24] ^ c[25] ^ c[27] ^ c[28];
    newcrc[12] = d[6] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ d[0] ^ c[4] ^ c[24] ^ c[25] ^
c[26] ^ c[28] ^ c[29] ^ c[30];
    newcrc[13] = d[7] ^ d[6] ^ d[5] ^ d[3] ^ d[2] ^ d[1] ^ c[5] ^ c[25] ^ c[26] ^
c[27] ^ c[29] ^ c[30] ^ c[31];
    newcrc[14] = d[7] ^ d[6] ^ d[4] ^ d[3] ^ d[2] ^ c[6] ^ c[26] ^ c[27] ^ c[28] ^
c[30] ^ c[31];
    newcrc[15] = d[7] ^ d[5] ^ d[4] ^ d[3] ^ c[7] ^ c[27] ^ c[28] ^ c[29] ^ c[31];
    newcrc[16] = d[5] ^ d[4] ^ d[0] ^ c[8] ^ c[24] ^ c[28] ^ c[29];
    newcrc[17] = d[6] ^ d[5] ^ d[1] ^ c[9] ^ c[25] ^ c[29] ^ c[30];
    newcrc[18] = d[7] ^ d[6] ^ d[2] ^ c[10] ^ c[26] ^ c[30] ^ c[31];
    newcrc[19] = d[7] ^ d[3] ^ c[11] ^ c[27] ^ c[31];
    newcrc[20] = d[4] ^ c[12] ^ c[28];
    newcrc[21] = d[5] ^ c[13] ^ c[29];
    newcrc[22] = d[0] ^ c[14] ^ c[24];
    newcrc[23] = d[6] ^ d[1] ^ d[0] ^ c[15] ^ c[24] ^ c[25] ^ c[30];
    newcrc[24] = d[7] ^ d[2] ^ d[1] ^ c[16] ^ c[25] ^ c[26] ^ c[31];
    newcrc[25] = d[3] ^ d[2] ^ c[17] ^ c[26] ^ c[27];
    newcrc[26] = d[6] ^ d[4] ^ d[3] ^ d[0] ^ c[18] ^ c[24] ^ c[27] ^ c[28] ^ c[30];
    newcrc[27] = d[7] ^ d[5] ^ d[4] ^ d[1] ^ c[19] ^ c[25] ^ c[28] ^ c[29] ^ c[31];
    newcrc[28] = d[6] ^ d[5] ^ d[2] ^ c[20] ^ c[26] ^ c[29] ^ c[30];
    newcrc[29] = d[7] ^ d[6] ^ d[3] ^ c[21] ^ c[27] ^ c[30] ^ c[31];
    newcrc[30] = d[7] ^ d[4] ^ c[22] ^ c[28] ^ c[31];
    newcrc[31] = d[5] ^ c[23] ^ c[29];
    nextCRC32_D8 = newcrc;
  end
  endfunction
endmodule
```

生成的代码里面仅为一个计算 CRC32 的一个函数 function。CRC 的计算与除了与生成多项式相关外，还与 CRC 的初始值，输入输出数据是否位反向以及计算结果是否取反等操作相关。上面生成的 CRC 计算的 Verilog 代码仅仅是与多项式有关的计算过程，未涉及到初始值等因素，所以要实现符合以太网要求的 CRC32 的完整计算需要添加一些计算操作。

再通过 CRC 计算工具回顾下以太网 CRC32 的计算的各种条件，打开 CRC 计算工具 CRC_Calculator，软件的界面如下，多项式选择 CRC-32。



从软件右边的 Info 可以看到以太网 CRC32 计算的相关的信息，具体含义如下。

- Name：参数模型名称。
- Width：宽度，即 CRC 比特数。
- Poly：生成项的简写，以 16 进制表示。例如：CRC-32 即是 0x04C11DB7，忽略了最高位的"1"，即完整的生成项是 0x104C11DB7。
- Init：这是算法开始时 CRC 的初始化预置值，十六进制表示。
- Refin：待测数据的每个字节是否按位反转，True 或 False。
- Refout：在计算后之后，异或输出之前，整个数据是否按位反转，True 或 False。
- Xorout：计算结果与此参数异或后得到最终的 CRC 值。

根据上面的信息，可以了解到计算 CRC32 与初始值，输入输出数据的操作有关，所以要将生成的 CRC 计算的 verilog 代码用在以太网上还需要做一定的添加修改，首先需要对该模块添加输入输出端口。代码如下。

```
module crc32_d8
(
  input           clk        ,
  input           reset_n    ,

  input    [7:0]  data       ,
  input           crc_init   ,
  input           crc_en     ,
```

```
  output   [31:0] crc_result
);
```

端口信号含义如下表 49-4 所示：

表 49-4 CRC 校验端口说明表

端口名	方向	描述
clk	input	模块工作时钟
reset_n	input	模块复位，低电平有效
data[7:0]	input	需要计算 CRC 校验值的输入数据，位宽为 8bit
crc_init	input	设置 CRC 计算的的初始值的使能信号，为高时，将 crc_result 为初始值 32 ‘hffffffff
crc_en	input	计算 CRC 的使能信号，当在该信号为 1 时，计算输入数据的 CRC 值
crc_result[31:0]	output	计算的 CRC 结果

CRC32 的 CRC 的初始值为 0xFFFFFFFF，在每次计算 CRC 前需将 CRC 的值回归到初始值，然后在需要的时候开始计算 CRC，初始值 crc_init 为高电平的时候回到初始值，具体实现代码如下：

```
always @ (posedge Clk)
if (!Reset)
  Crc <={32{1'b1}};
else if(Initialize)
  Crc <={32{1'b1}};
else if (Enable)
  Crc <= #1 CrcNext;
```

CRC 计算前对输入的数据有按位反转的要求，也就是需要高低 bit 调换顺序。具体代码如下：

```
assign
Data={Data_in[0],Data_in[1],Data_in[2],Data_in[3],Data_in[4],Data_in[5
],Data_in[6],Data_in[7]};
```

CRC 计算后的结果也需要做一定的处理，根据要求需要进行先进行位反转，然后与数据 0xFFFFFFFF 进行位异或操作（也就是取反，0 与 1 异或为 1，1 与 1 异或为 0）。这两个操作可同时进行，具体代码如下。

```
assign Crc_eth = ~{
    CrcNext[24], CrcNext[25], CrcNext[26], CrcNext[27],CrcNext[28],
    CrcNext[29], CrcNext[30], CrcNext[31],Crc[16], Crc[17], Crc[18],
    Crc[19],Crc[20], Crc[21], Crc[22], Crc[23],Crc[ 8], Crc[ 9],
    Crc[10], Crc[11],Crc[12], Crc[13], Crc[14], Crc[15],Crc[ 0],
    Crc[ 1], Crc[ 2],Crc[ 3],Crc[ 4], Crc[ 5], Crc[ 6], Crc[ 7]};
```

关于 CRC 校验能够降低数据链路通信出错概率的理论推导，这里不过多介绍，请有兴趣的自行通过网络学习。本节仅使用一个公版的 CRC32 校验程序，作为以太网帧校验字段的运算单元。

49.9.2自动 CRC 校验实验测试

crc32_d8 为千兆 GMII/RGMII 接口的 8 位数据位宽的并行 32 位 CRC 计算单元。

eth_send.v 为以太网 MAC 帧组包逻辑，完成以太网 MAC 帧的建立和发送。

eth_send_test.v 为 ARP 包组包工具，并调用 eth_send 和 crc32_d8.v 完成完整的 ARP 报文的发送。

例化以太网帧发送模块，并将目的地址设置为广播地址 ffffffff、源地址为板卡地址。这里继续使用 00_0a_35_01_fe_c0 地址，以避免与其他实际地址冲突。协议类型为 ARP（0806）。数据长度是整个发送的 arp 协议报文的长度 46。

由于使用了自动 CRC 校验，因此用户可以手动修改源 MAC 地址，并使用以太网发包工具重新组建对应的 ARP 包，替换查找表中的值，然后发送数据。组包完成后无需再用 PC 端 CRC 计算工具计算 CRC 值。

以下简单做个测试，分别使用 00_0a_35_01_fe_c0 和 00_0a_35_01_fe_c1 作为源 MAC 地址，发送 ARP 包，由于整个以太网帧中已经有数据内容发生了变化，因此 CRC 校验值不会相同，以此来检查是否不同的数据包 CRC 计算都能通过。修改时仅修改 eth_send_test 模块中的以下内容：

```
eth_send eth_send(
    .rst_n(rst_n),
    .tx_go(tx_go),
    .data_length(16'd46),//需要发送的数据长度，以字节为单
    .des_mac(48'hff_ff_ff_ff_ff_ff),
    .src_mac(48'h00_0a_35_01_fe_c0),
    .type_length(16'h08_06),
    .CRC_Result(CRC_result),
    .CRC_EN(CRC_EN),
    .fifo_rdreq(fifo_rdreq),
    .fifo_rddata(fifo_rddata),
    .fifo_rdclk(fifo_rdclk),
    .gmii_tx_clk(gmii_tx_clk),
    .gmii_tx_en(gmii_tx_en),
    .gmii_tx_er(gmii_tx_er),
    .gmii_tx_data(gmii_tx_data)
);

//sender mac
08: fifo_rddata =   8'h00;
09: fifo_rddata =   8'h0a;
10: fifo_rddata =   8'h35;
11: fifo_rddata =   8'h01;
12: fifo_rddata =   8'hfe;
13: fifo_rddata =   8'hc0;
```

```
eth_send eth_send(
    .rst_n(rst_n),
    .tx_go(tx_go),
    .data_length(16'd46),//需要发送的数据长度，以字节为
    .des_mac(48'hff_ff_ff_ff_ff_ff),
    .src_mac(48'h00_0a_35_01_fe_c1),
    .type_length(16'h08_06),
    .CRC_Result(CRC_result),
    .CRC_EN(CRC_EN),
    .fifo_rdreq(fifo_rdreq),
    .fifo_rddata(fifo_rddata),
    .fifo_rdclk(fifo_rdclk),
    .gmii_tx_clk(gmii_tx_clk),
    .gmii_tx_en(gmii_tx_en),
    .gmii_tx_er(gmii_tx_er),
    .gmii_tx_data(gmii_tx_data)
);

//sender mac
08: fifo_rddata =   8'h00;
09: fifo_rddata =   8'h0a;
10: fifo_rddata =   8'h35;
11: fifo_rddata =   8'h01;
12: fifo_rddata =   8'hfe;
13: fifo_rddata =   8'hc1;
```

图 49-18 修改 eth_send_test 模块内容

下图为分别设置源 MAC 为 00_0a_35_01_fe_c0 和 00_0a_35_01_fe_c1 时，PC 端以太网抓包工具抓到的数据。

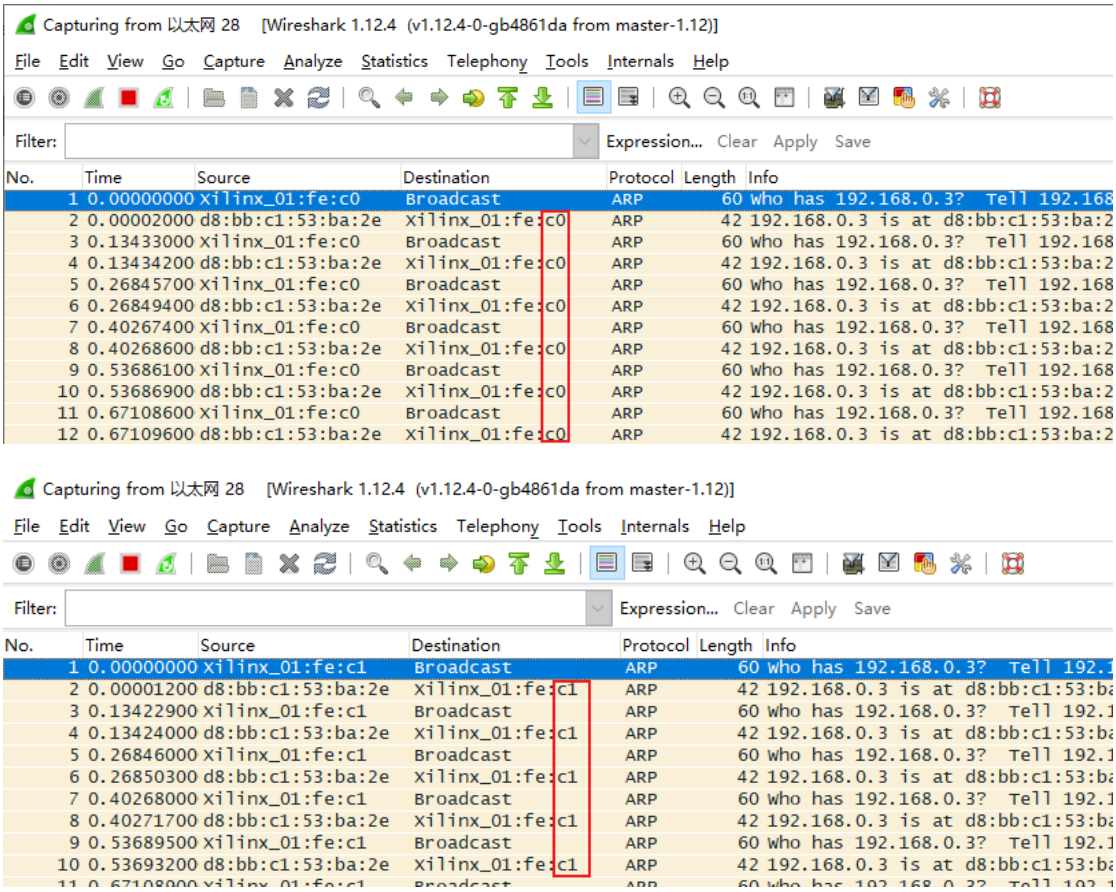


图 49-19 不同源 MAC 地址下 ARP 请求帧抓包结果

可以看到，修改了数据报内容后整个发送模块也能自动得到对应的正确的 CRC 校验字并发送到 PC 端。

49.10总结

至此，我们完成了以太网首发过程中最底层，也是最重要的部分——MAC 帧的发送工作，并实现了 RGMII 接口的 ARP 协议报文的发送实验。本实验的最终目的不在于完成 ARP 协议的发送。事实上，对于 ARP 协议的内容，到此为止，后续也不会再应用，本节内容通过最简单的应用协议，让大家掌握了以太网开发调试过程中各种软件的操作方法，并解决了以太网帧中的难点——CRC 校验算法。在后续的内容中，我们还会多次用到本实验中各种软件的使用和开发思路。

50 IP 协议介绍与 IP 校验和算法实现

工程源码	|----02_设计实例 |----ch50_IP_checksum
相关视频课程	
说明	

章节导读

本章将讲解千兆以太网网络层 IP 协议的内容及算法实现。IP 层的实质是在 MAC 层的基础上将原有协议进行一个更加细化的定义而得。它最核心的意义，就是让 IP 地址参与到网络数据传输中，让硬件和硬件的通信传输，从网卡号作为唯一的身份识别标志，变为以 IP 地址作为身份识别标志。而 IP 层最核心的内容，就是 IP 协议数据字段的格式。

50.1IP 协议数据字段格式

IP 是 TCP/IP 协议族中最核心的协议，所有的 TCP、UDP、ICMP、IGMP 数据都以 IP 数据报的格式传输。IP 仅提供尽力而为的传输服务，如果发生某种错误，IP 会丢失该数据，然后发送 ICMP 消息给信源端。另外，IP 数据报可以不按发送顺序接收。

IP 数据报的格式如图 50-1 所示，IP 数据报的长度 / 类型段的数值为 0x0800，数据和填充段包括 IP 头部数据和 IP 数据两个部分。

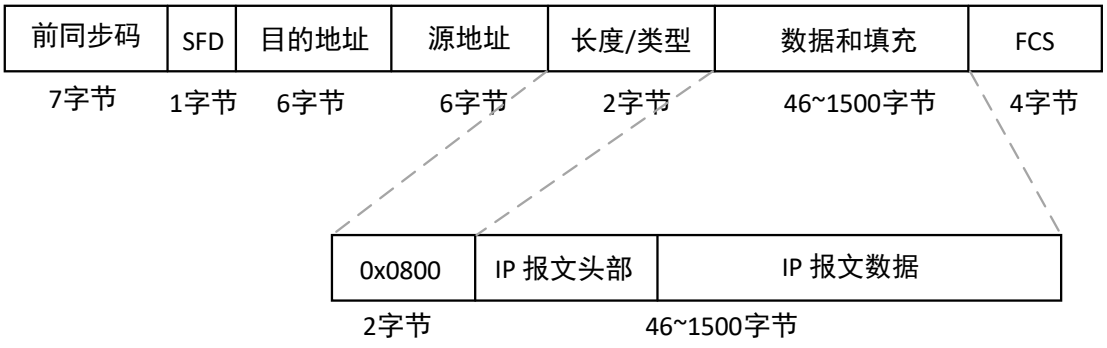


图 50-1 IP 数据报格式

其中，和以太网帧具有帧头一样，IP 数据报也包含了一个 IP 报头部分，与 IP 协议相关的一些信息如 IP 地址，数据包长度等会被打包进 IP 报头中，然后再与需要传输的 IP 报文数据一起，作为 MAC 帧的数据和填充字段送往 MAC 层发送，接下来，将着重介绍 IP 协议首部的详细信息。

50.2IP 协议首部详解

IP 数据报的格式如下表 50-1 所示。

表 50-1 IP 数据报格式说明表

0~3	4~7	7~15	16~18			19~31
版本	首部长度	服务类型	总长度			
分段标识，标识属于同一数据报			NC	DF	MF	段偏移
生存周期 TTL(秒)		上层协议	报头校验和			
源 IP 地址						
目的 IP 地址						
可选字段（可无）						
IP 数据报数据部分，N 个字节						

其中每个字段的功能简介如下表 50-2 所示。

表 50-2 每个字段功能简介说明表

IP 字段	版本度	4 位	IP 协议版本	版本就是 IPV4 或者 IPV6，一般选择 IPV4，即版本值为 4。
	首部长	4 位	IP 首部长度	IP 首部长度即有多少个 32 位数，当 IP 首部长度为 20 时（即无可选字段），该值为 5。（5*4 =20）
	服务类型	8 位	指示了报文的优先权等	简单的发送可以全部置 0 即可
	总长度	16 位	IP 报文长度	IP 报文（报头+数据）长度
	分段标识	16 位	是否属于同一数据段	IP 报文的分段 ID
	段标识和段偏移	16 位	可设为 0	简单的发送可以全部置 0 即可
	生存周期（TTL）	8 位	可以经过的最大路由数	生存时间字段设置了数据报可以经过的最多路由器数,表示数据包在网络上生存多久。TTL 的初始值由源主机设置(通常为 32、64 或 128)，一旦经过一个处理它的路由器,它的值就减去 1。当该字段的值为 0 时,数据报就被丢弃,并发送 ICMP 消息通知源主机。这样当封包在传递过程中由于某些原因而未能抵达目的地的时候就可以避免其一直充斥在网路上面。
	上层协议类型	8 位	指该 IP 包中数据段内容所使用的网络协议类型，如 ICMP、DNS 等。	常用协议号： 00：IP 01：ICMP 06：TCP 17：UDP
	报头校验和	2	IP 报头的校验和	对 IP 报头计算得出
	源 IP 地址	4	发送端的 IP 地址	如 192.168.0.2
	目的 IP 地址	4	接收端的 IP 地址	如 192.168.0.3

	可选字段		可选	没有的时候长度可以为 0

下表 50-3 为 IP 数据报中上层协议字段常见协议与其对应的协议号。

表 50-3 IP 数据报协议字段与其对应的协议号

协议号	协议	协议号	协议
00	IP	22	XNS-IDP
01	ICMP	27	RDP
02	IGMP	29	ISO-TP4
03	GGP	36	XTP
04	IP-ENCAP	37	DDP
05	ST	39	IDPR-CMTP
06	TCP	73	RSPF
08	EGP	81	VMTP
12	PUP	89	OSPFIGP
17	UDP	94	IPIP
20	HMP	98	ENCAP

在 IP 报头中，前 20 字节和紧接其后的选项部分是 IP 数据报的首部。前 20 个字节是固定的，可选字段则可有可无。首部的每一行是一个 32 位字的单位，最高位在左边，为 0bit，最低位在右边，为 31bit。4 字节的 32bit 值按照以下次序传输：首先 0-7bit，其次 8-15 bit，然后 16-23bit，最后是 24-31bit，这种传输次序称为 big endian 字节序(我们在 C 语言写位操作的算法时常用到该词)。TCP/IP 首部中的所有二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序，其他形式存储的二进制数据，如 little endian 格式，则必须在传输数据之前把首部转化成网络字节序。以下对 IP 报头中各个字段的内容详细介绍。

- 版本为当前使用的 IP 协议版本号，大家最熟悉，用的最多的是 IPv4 版本，当然现在 IPv6 版本用的也越来越多了。本实验使用的是 IPv4 版本，所以在版本字段为 4 就表示版本号为 IPv4。
- 首部长度是指首部占 32bit 字的数目，因为 4 位的最大值为 15，因此首部最长为 60 字节，也即是说选项部分的最大值为 40 字节，不够 4 的倍数，要用 0 填充，使数据部分的起始地址为 4 的倍数。
- 服务类型：略。
- 总长度指整个 IP 数据报的长度，包括首部和数据部分，16bit，最长可达 65535 字节。尽管理论上可以传送一个长达 65535 的 IP 数据报，但实际上还要考虑网络的最大承载能力等因素。
- 3 个标志位主要用来标识分片的 IP 数据报，片位移为分片的数据报的首个字节偏离整个原始数据报的位置。

- 源 IP 地址和目的 IP 地址则是该 IP 报文的发送方和接收方的网络地址，例如在我们前面的实验中，说 FPGA 的 IP 地址为 192.168.0.2，电脑的 IP 地址为 192.168.0.3，那么 FPGA 在向电脑发送数据包时，源 IP 地址就是 192.168.0.2，目的 IP 地址为 192.168.0.3。

50.3IP 首部校验和算法介绍

首部检验和字段是根据 IP 首部计算的检验和码，不对首部后面的数据进行计算。具体的计算和实现方法将在后文详细介绍。其计算方法为：

将校验和字段置为 0，然后将 IP 包头按 16 比特分成多个单元，如包头长度不是 16 比特的倍数，则用 0 比特填充到 16 比特的倍数；

对各个单元采用反码加法运算(即高位溢出位会加到低位，通常的补码运算是直接丢掉溢出的高位)，将得到的和的反码填入校验和字段；

例如，我们使用 IP 协议发送一个数据长度为 30 个字节的数据包，发送端 IP 为 192.168.0.2，接收端 IP 为 192.168.0.3。则 IP 报头可如下图 50-2 中阴影部分标记内容所示：

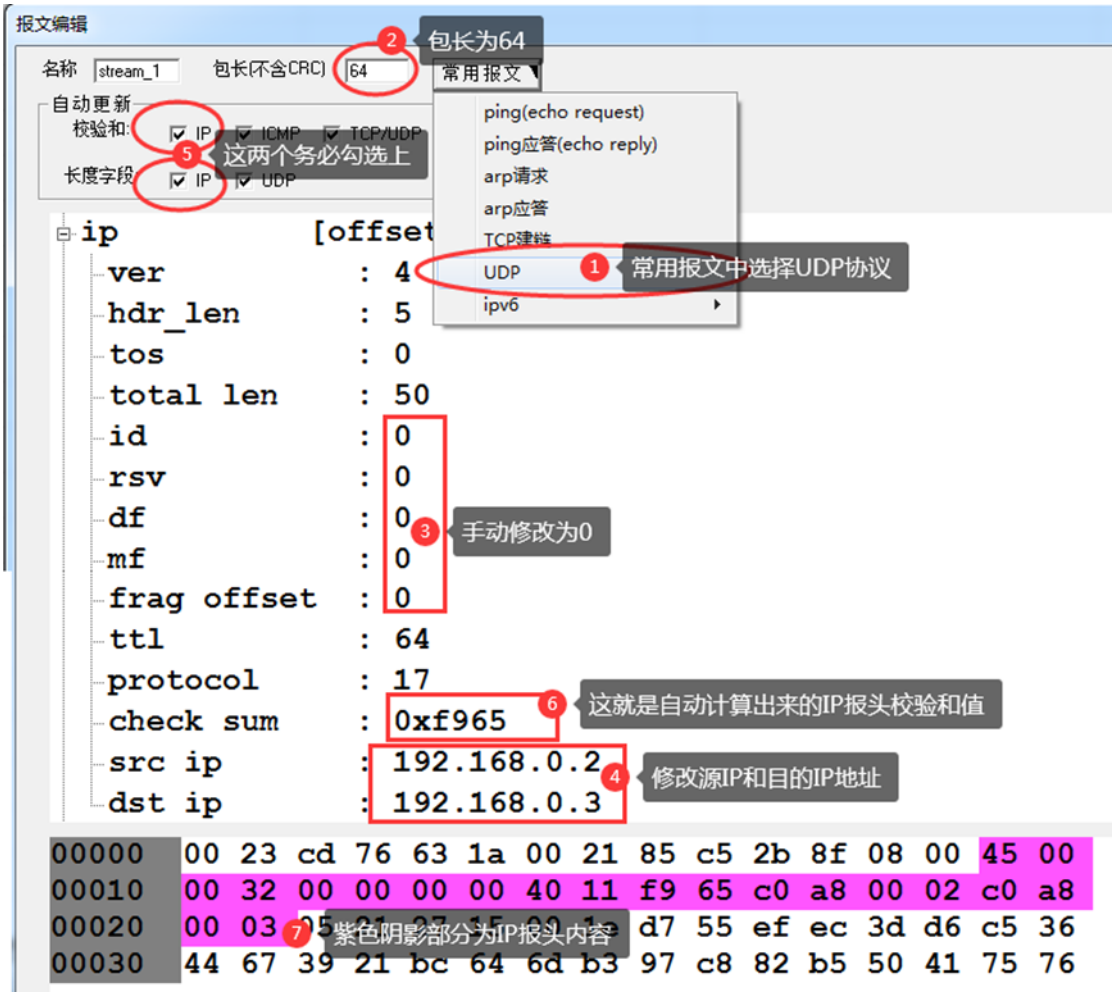


图 50-2 IP 报头内容

IP 报头中每个字段的内容详细如下表 50-4 所示。

表 50-4 IP 报头每个字段内容详细说明表

协议类型 报头长度	服务类型 tos	IP 报文长度 total_len		分段标识 id		分段和偏移 offset	
0x45	0x00	0x00	0x32	0x00	0x00	0x00	0x00
v4; 20B	普通	50 字节					
生存周期 ttl	上层协议 protocol	报头校验和 checksum		源地址 src_ip			
0x40	0x11	0xf9	0x65	0xc0	0xa8	0x00	0x02
64 级	17 UDP			192	168	0	2
目标地址 dst_ip							
0xc0	0xa8	0x00	0x03				
192	168	0	3				

按照上述提到的 IP 首部校验和的方法计算 IP 首部校验和，即：

0x4500 + 0x0032 + 0x0000 + 0x0000 + 0x4011 + 0x0000(计算时强制置 0) + 0xc0a8 + 0x0002 + 0xc0a8 + 0x0003 = 0x20698。

0x0002 + 0x0698 = 0x069a

checksum = ~0x069a = 0xf965

计算结果与实际发包软件计算的一致。

该部分使用 Verilog 编写计算单元非常的方便，可以用纯组合逻辑实现，也可以用时序逻辑实现。以下为使用 Verilog 编写的 checksum 的逻辑代码。

```
module checksum(
    input [3:0]ver, //版本
    input [3:0]hdr_len,//首部长度
    input [7:0]tos, //服务类型
    input [15:0]total_len,  //IP 报文总长
    input [15:0]id, //分段标识
    input [15:0]offset, //偏移
    input [7:0]ttl, //生存周期
    input [7:0]protocol, //上层协议类型
    input [31:0]src_ip, //源 IP 地址
    input [31:0]dst_ip, //目的 IP 地址

    output [15:0]checksum_result //校验和
);

    wire [31:0]sum;

    assign sum = {ver,hdr_len,tos} + total_len + id
                 + offset + {ttl,protocol} + src_ip[31:16]
                 + src_ip[15:0] + dst_ip[31:16] + dst_ip[15:0];

    assign checksum_result = ~(sum[31:16] + sum[15:0]);

endmodule
```

该实现方法看似简单且合理，但是在实际使用过程中，有读者反映在某些情况下会发生错误，该读者举了一个例子，假设某帧数据的 IP 报头如下表 50-5 所示：

表 50-5 某帧的 IP 报头说明表

协议类型 报头长度	服务类型 tos	IP 报文长度 total_len		分段标识 id		分段和偏移 offset	
0x45	0x00	0x05	0xd6	0x70	0xf6	0x40	0x00
v4; 20B	普通	50 字节					
生存周期 ttl	上层协议 protocol	报头校验和 checksum		源地址 src_ip			
0x80	0x11			0xc0	0xa8	0x01	0x6e

64 级	17 UDP			192	168	0	2
目标地址 dst_ip							
0xc0	0xa8	0x01	0x64				
192	168	0	3				

该数据帧中，如果按照上述方法计算：

第一步：4500+05D6+70F6+4000+8011+C0A8+016E+C0A8+0164=2FFFF

第二步：ffff + 2 = 1

第三步：~1 = fffe

既此算法得到的结果为 0xfffe，而使用以太网发送时（或者使用小兵以太网测试仪），实际的值为 0xfffd。

出错的原因在于 ffff + 2 会发生溢出，而上述设计没有考虑到该情况。解决方案很简单，对第二步的计算结果再根据高 16 位的值判断是否需要再执行一次加法操作。

因此可将设计改进如下：

```
module checksum(
    input [3:0]ver, //版本
    input [3:0]hdr_len,//首部长度
    input [7:0]tos, //服务类型
    input [15:0]total_len,  //IP 报文总长
    input [15:0]id, //分段标识
    input [15:0]offset, //偏移
    input [7:0]ttl, //生存周期
    input [7:0]protocol, //上层协议类型
    input [31:0]src_ip, //源 IP 地址
    input [31:0]dst_ip, //目的 IP 地址

    output [15:0]checksum_result //校验和
);


    wire [31:0]suma,sumb;

    assign suma = {ver,hdr_len,tos} + total_len + id
                        + offset + {ttl,protocol} + src_ip[31:16]
                        + src_ip[15:0] + dst_ip[31:16] + dst_ip[15:0];

    assign sumb = (suma[31:16] + suma[15:0]);

    assign checksum_result = sumb[31:16]?
```

```
                                ~(sumb[31:16] + sumb[15:0]):
                                ~sumb[15:0];
endmodule
```

根据 IP 头部校验计算步骤，代码中 suma 就是经过步骤 1 计算的结果，sumb 是步骤 2 计算的和。checksum_result 为根据步骤 3 需要根据高 16bit 是否为零来判断是否需要继续相加求和并取反输出的值。

部分网友在看到这里后产生了疑问，表示既然在第二步的时候出现了相加溢出的情况，那么现在加了一步，在步骤三中再根据步骤 2 结果的高 16 位是否不为 0 执行了再一次的相加操作，那么这一次的相加操作有没有可能继续溢出呢，是否还需要再循环加下去，直到高 16 位不为 0 了再输出呢，事实上这种担心是多余的，该结果完全可以使用理论计算的方式验证。

对于 sumb = suma[31:16]+suma[15:0]来说，该公式包含了高 16bit 不为 0 继续相加求和，高 16bit 为 0 情况下，低 16bit 数据加 0。不管哪种情况 2 个 16bit 数据相加的结果取值范围为 17’h00001~17’h1fffe（当 16’hffff+16’hffff 时，求和值最大），也就是说在第三步再次进行求和时，出现的结果最大时候的情况就是 ffffe+1，其值为 ffff，高 16 位不可能再大于 0，所以无需再继续求和。

设计完成后，可以编写 testbench 对该算法进行计算，testbench 如下所示。

```
`timescale 1ns/1ns
module checksum_tb;

    reg [3:0]ver;
    reg [3:0]hdr_len;
    reg [7:0]tos;
    reg [15:0]total_len;
    reg [15:0]id;
    reg [15:0]offset;
    reg [7:0]ttl;
    reg [7:0]protocol;
    reg [31:0]src_ip;
    reg [31:0]dst_ip;

    wire [15:0]checksum_result;

    checksum checksum
    (
        .ver(ver),
        .hdr_len(hdr_len),
        .tos(tos),
        .total_len(total_len),
```

```
        .id(id),
        .offset(offset),
        .ttl(ttl),
        .protocol(protocol),
        .src_ip(src_ip),
        .dst_ip(dst_ip),
        .checksum_result(checksum_result)
    );

    initial begin
        ver = 0;
        hdr_len = 0;
        tos = 0;
        total_len = 0;
        id = 0;
        offset = 0;
        ttl = 0;
        protocol = 0;
        src_ip = 0;
        dst_ip = 0;
        #200;

        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80002;
        dst_ip = 32'hc0a80003;
        #300;

        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80005;
        dst_ip = 32'hc0a80008;
```

```
        #300;

        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80105;
        dst_ip = 32'hc0a80108;
        #300;
        $stop;
    end

endmodule
```

使用 Modelsim 仿真，整体的仿真波形如下图 50-3 所示。

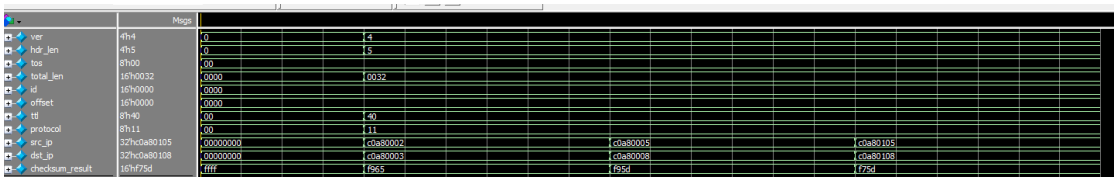


图 50-3 Modelsim 仿真波形图

通过上图可以看出，逻辑计算的结果与使用小兵以太网测试仪计算出的 IP 首部校验和一致，所以证明该算法实现是正确的。

另外，有读者曾经表示担忧，整个算法实现全部使用的是组合逻辑，这样的情况会不会影响整个设计的时序性能。能否影响设计的时序性能，主要看该计算产生到该结果被使用时的时间差。通过时序仿真可知，从输入数据变化到输出结果稳定，至少需要 10.263ns。从 IP 首部所有数据确定，到开始发送校验和字段，千兆以太网 RGMII 接口是 10 个时钟周期(10*8ns = 80ns)，即完全满足时序要求。

50.4总结

至此，我们就完成了 IP 数据报中最重要的部分，IP 报头校验和的计算逻辑设计，同时，相信大家已经了解了 IP 报头的具体内容，IP 数据包就是紧跟再 IP 报头后面的。然后，IP 的数据部分也还并不是单纯的用户数据，我们在网络应用时，还需要将我们的用户数据进一步打包到比 IP 协议更上一层的协议中，再

通过 IP 协议发送，该内容我们将在下一节内容讲到。

51 UDP 协议原理与 FPGA 实现

工程源码	|----02_设计实例 |----ch51_udp_send_rgmii
相关视频课程	
说明	

章节导读

本章将讲解千兆以太网传输层 UDP 协议的相关内容。学习 UDP 层协议的内容，核心也是明确该协议的数据字段格式。在此基础上，理解其“不可靠、无连接”的传输特性。同时，结合前面章节的内容，进一步深化理解用户数据、UDP、IP、MAC 层的层层打包嵌套关系。

51.1UDP 协议介绍

UDP 是 User Datagram Protocol 的简称，中文名是用户数据包协议，是 OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。IETF RFC 768 是 UDP 的正式规范。UDP 在 IP 报文的协议号是 17（即 0x11）。

UDP 协议全称是用户数据报协议，在网络中它与 TCP 协议一样用于处理数据包，是一种无连接的协议。在 OSI 模型中，在第四层——传输层，处于 IP 协议的上一层。UDP 有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP 用来支持那些需要在计算机之间传输数据的网络应用。包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都需要使用 UDP 协议。UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是即使是在今天 UDP 任然不失为一项非常实用和可行的网络传输层协议。

与所熟知的 TCP（传输控制协议）协议一样，UDP 协议直接位于 IP（网络协议）协议的顶层。根据 OSI（开放系统互连）参考模型，UDP 和 TCP 都属于传输层协议。UDP 协议的主要作用是将网络数据流量压缩成数据包的形式。一个典型的数据包就是一个二进制数据的传输单位。每一个数据包的前 8 个字节用来包含报头信息，剩余字节则用来包含具体的传输数据。

以太网 UDP 帧的用户数据是打包在 UDP 协议中，而 UDP 协议又是基于 IP 协议之上的，IP 协议又是走 MAC 层发送的，即从包含关系来说：MAC 帧中的

数据段为 IP 数据报文，IP 报文中的数据段为 UDP 报文，UDP 报文中的数据段为用户希望传输的数据内容，如“Hello，welcome to FPGA!”。下图 51-1 为使用 UDP 协议发送“Hello，welcome to FPGA!”数据的层层打包示意图。

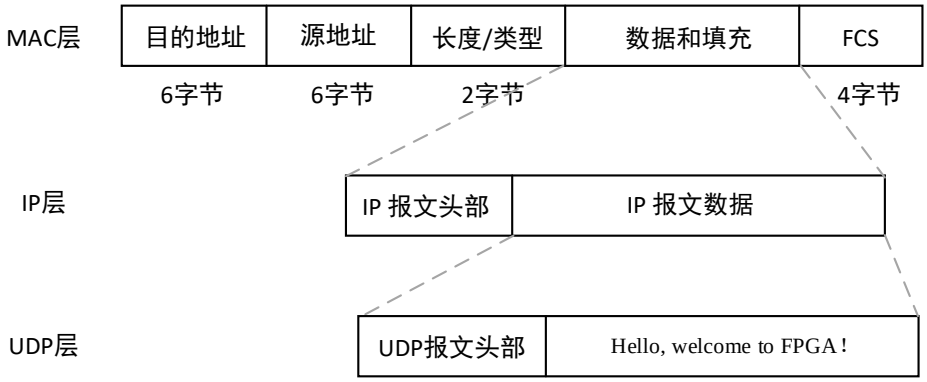


图 51-1 UDP 数据包（Hello，welcome to FPGA!）

其中，和以太网帧、IP 报文具有帧头一样，UDP 数据报也包含了一个 UDP 报头部分，与 UDP 协议相关的一些如端口号，数据包长度等会被打包进 UDP 报文中，然后再与需要传输的 UDP 报文数据一起，作为 IP 报文的数据段送往 IP 层发送。

51.2UDP 数据报格式

UDP 协议实用端口号为不同的应用保留其各自的数据传输通道。UDP 和 TCP 协议正是采用这一机制实现对同一时刻内多项应用同时发送和接收数据的支持。数据发送一方（可以是客户端或服务器端）将 UDP 数据报通过源端口发送出去，而数据接收一方则通过目标端口接收数据。有的网络应用只能使用预先为其预留或注册的静态端口；而另外一些网络应用则可以使用未被注册的动态端口。因为 UDP 报文使用两个字节存放端口号，所以端口号的有效范围是从 0 到 65535。一般来说，大于 49151 的端口号都代表动态端口。

数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上来说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到 8192 字节。

UDP 协议使用报头中的校验值来保证数据的安全。校验值首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果

某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算值将不会相符，由此 UDP 协议可以检测是否出错。这与 TCP 协议是不同的，后者要求必须具有校验值。

许多链路层都提供错误检查，包括流行的以太网协议，也许你想知道为什么 UDP 也要提供检查和校验。其原因是链路层以下的协议在源端和终端之间的某些通道可能不提供错误检测。虽然 UDP 提供有错误检测，但检测到错误时，UDP 不做错误校正，只是简单地把损坏的消息段扔掉，或者给应用程序提供警告信息。下表 51-1 为 UDP 报文的格式。

表 51-1 UDP 报文格式表

	字段名称	长度	功能说明	解释
UDP 字段	源端口号	2 字节	例如 5000	发送方的网络端口地址
	目的端口号	2 字节	例如 6000	接收方的网络端口地址
	UDP 长度	2 字节	UDP 报文长度	UDP 报文（含数据）长度
	UDP 校验和	2 字节	UDP 报头校验和	UDP 协议的报头和数据的和校验
	UDP 数据	n 字节		

由于每次需要发送的数据都不相同，而且校验和内容在发送数据段之前就需要计算出来，不像 MAC 层是在所有数据都发送完成之后才发送 CRC 校验值，因此在 UDP 组包时，校验和值的计算是一个不太好处理的地方，即使通过专用的计算单元计算出校验和，在待发送数据输入完成到数据开始通过 UDP 包发送之间，也有一个计算校验和的时间段。所幸，UDP 校验和在一些要求不太严格的地方，是可以忽略的。因此，在使用 Verilog 实现时，为了提升效率并节约 FPGA 资源，将校验和字段忽略。这样一来，UDP 数据包的组包就变得非常简单了。例如本机端口号为 5000(0x1388)，要给端口号为 6102(0x17D6)的目标主机发送“Hello, welcom to FPGA!!!”，数据为 24 个字节，UDP 首部 8 个字节，合计 32 个字节，则 UDP 数据段可以组包如下表 51-2 所示：

表 51-2 UDP 数据段组包方式

0x13	0x88	0x17	0xD6
0x00	0x20	0x00	0x00
‘H’(0x48)	‘e’(0x65)	‘l’(0x6c)	‘l’(0x6c)
‘o’(0x6f)	‘,’(0x2c)	‘ ’(0x20)	‘w’(0x77)
‘e’(0x65)	‘l’(0x6c)	‘c’(0x63)	‘o’(0x6f)
‘m’(0x6d)	‘ ’(0x20)	‘t’(0x74)	‘o’(0x6f)
‘ ’(0x20)	‘F’(0x46)	‘P’(0x50)	‘G’(0x47)
‘A’(0x41)	‘!’(0x21)	‘!’(0x21)	‘!’(0x21)

以下图 51-2 为通过以太网工具组包出来的该报文 UDP 部分内容：

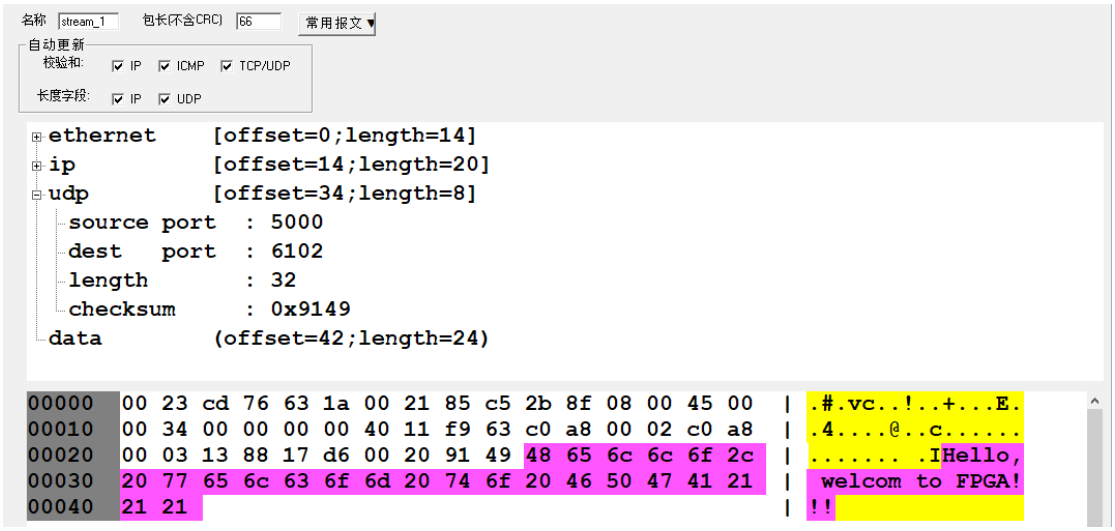


图 51-2 太网工具组包出的该报文 UDP 部分内容

51.3以太网 UDP 帧发包实例（手动 IP checksum）

本节实验我们使用以太网发包工具构建一个 UDP 包，然后将该包使用我们上一节设计的带 CRC 校验的 ARP 发送工程来发送出去。设计时，只需替换查找表中的内容即可。本设计实例提供的工程名称为 udp_send_rgmii.rar（千兆 RGMII 接口）。

51.3.1使用以太网测试仪构建 UDP 数据包

打开小兵以太网测试仪，新建一个数据流，如下图 51-3 所示。

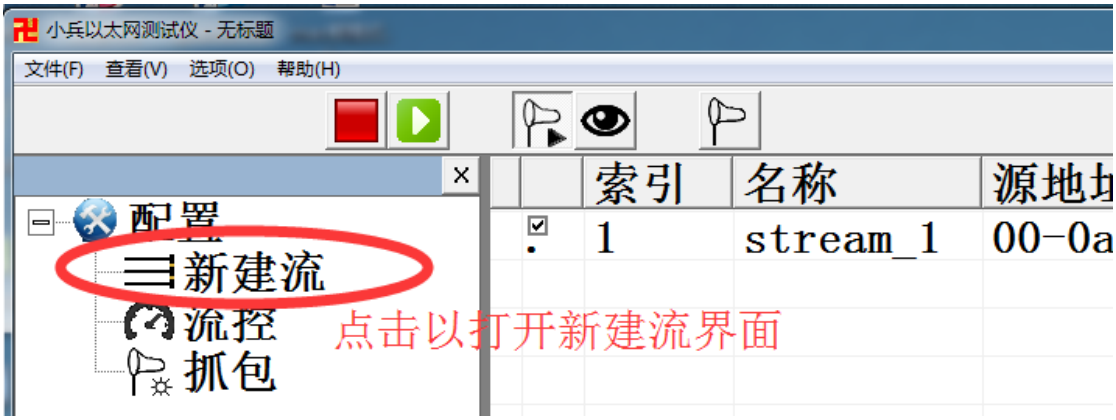


图 51-3 新建流

在常用报文中选择 UDP 格式报文，设置包长为 66，自动更新的所有项全部勾选上，如下图 51-4 所示。

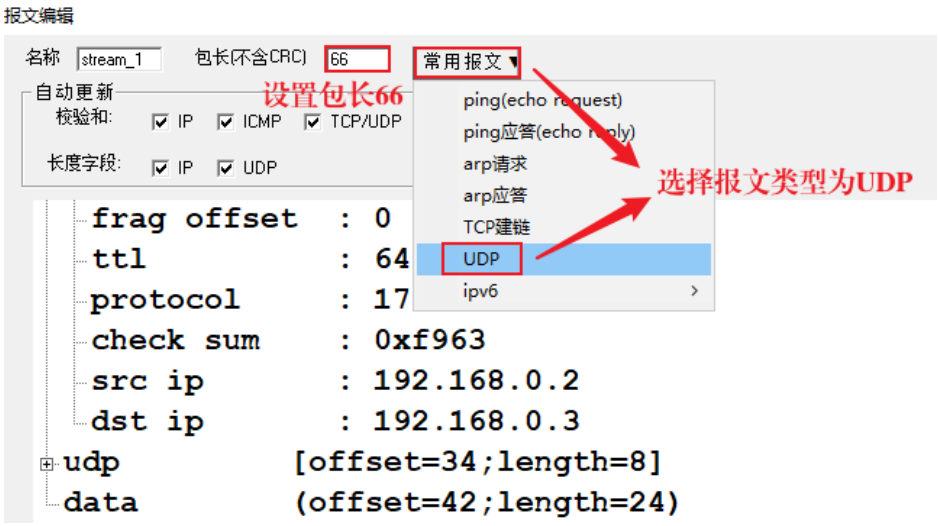


图 51-4 设置 UDP 报文格式

修改以太网包和 IP、UDP 包中各自段内容如下图 51-5 所示。

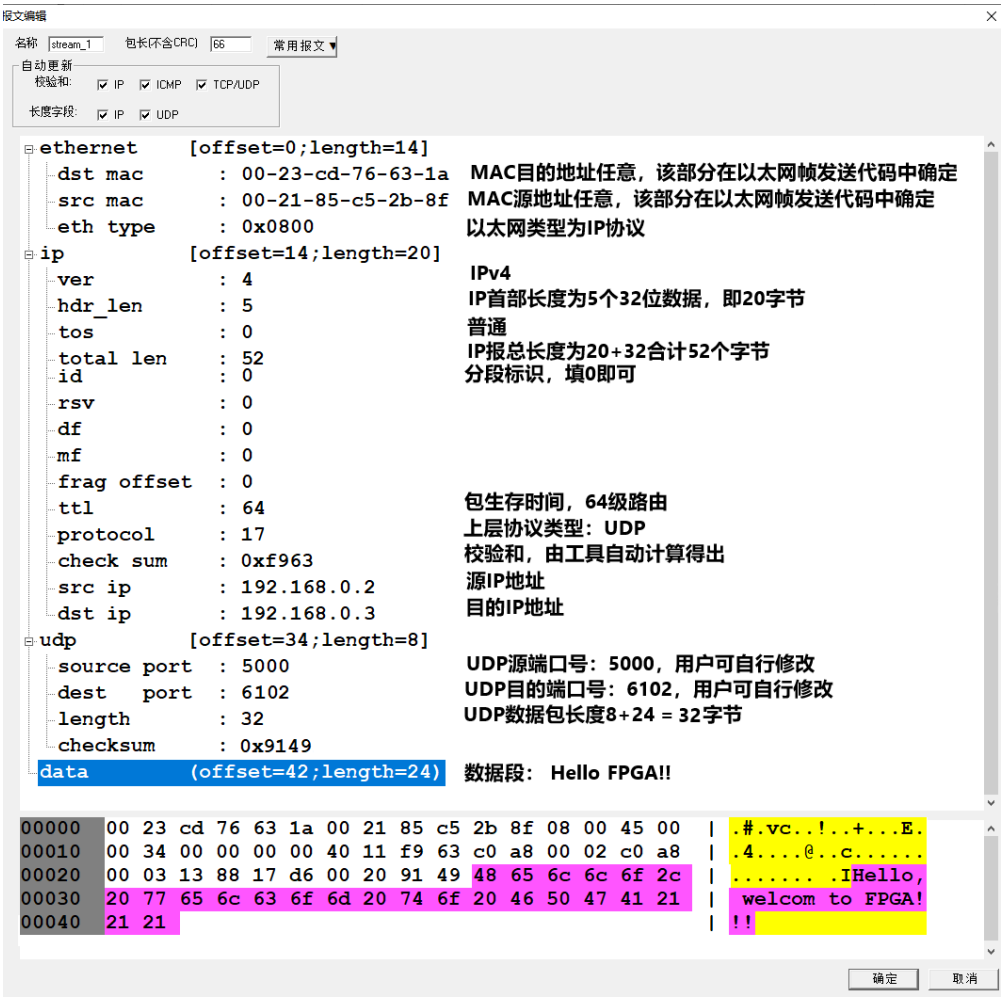


图 51-5 修改报文中各字段内容

配置好之后点击确定，将设置好的数据包导出为 16 进制格式文本文件，如

下图 51-6 所示。

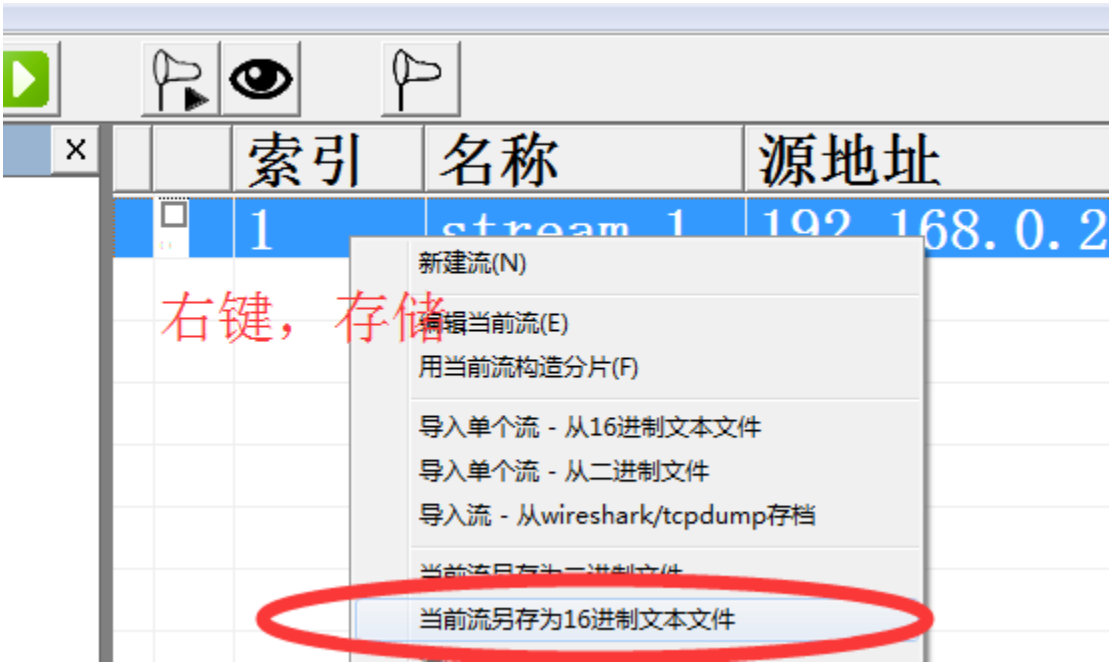


图 51-6 另存为 16 进制文本文件

得到的文本中数据内容如下所示：

0x00, 0x23, 0xcd, 0x76, 0x63, 0x1a, 0x00, 0x21, 0x85, 0xc5, 0x2b, 0x8f, 0x08, 0x00, 0x45, 0x00, 0x00, 0x34, 0x00, 0x00, 0x00, 0x00, 0x40, 0x11, 0xf9, 0x63, 0xc0, 0xa8, 0x00, 0x02, 0xc0, 0xa8, 0x00, 0x03, 0x13, 0x88, 0x17, 0xd6, 0x00, 0x20, 0x91, 0x49, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x65, 0x6c, 0x63, 0x6f, 0x6d, 0x20, 0x74, 0x6f, 0x20, 0x46, 0x50, 0x47, 0x41, 0x21, 0x21, 0x21

其中我们只需要 IP 层以及 UDP 层，MAC 层的数据直接丢弃，因为我们以太网帧发送逻辑会自动组建 MAC 数据。所以我们只需要保留深色背景的内容即可。

51.3.2使用 FPGA 发送 UDP 数据

修改 eth_send 的例化内容，并将目的地址设置广播地址：48'hFF_FF_FF_FF_FF_FF，源地址为板卡地址。这里将板卡地址为 48'h00_0a_35_01_fe_c0。协议类型为 IP（0800）。数据长度为 50。

由于使用了自动 CRC 校验，因此用户可以手动修改源 MAC 地址，并使用以太网发包工具重新组建对饮的 UDP 包，替换查找表中的值。然后发送数据。组包完成后无需再用 PC 端 CRC 计算工具计算 CRC 值。以下为 eth_send_test.v 中部分代码内容：

```
wire fifo_rdreq;
wire tx_go;

reg [7:0] fifo_rddata;
wire fifo_rdclk;
reg [11:0]data_cnt;
wire CRC_EN;

wire [31:0]CRC_result;

CRC32_D8 CRC32_D8(
  .Clk(gmii_tx_clk),
  .Reset(rst_n),
  .Data_in(gmii_tx_data),
  .Enable(CRC_EN),
  .Initialize(~gmii_tx_en),
  .Crc(),
  .CrcNext(),
  .Crc_eth(CRC_result)
);

eth_send eth_send(
  .rst_n(rst_n),
  .tx_go(tx_go),
  .data_length(16'd52), //需要发送的数据长度，以字节为单位
  .des_mac(48'hff_ff_ff_ff_ff_ff),
  .src_mac(48'h00_0a_35_01_fe_c0),
  .type_length(16'h08_00),
  .CRC_Result(CRC_result),
  .CRC_EN(CRC_EN),
  .fifo_rdreq(fifo_rdreq),
  .fifo_rddata(fifo_rddata),
  .fifo_rdclk(fifo_rdclk),
  .gmii_tx_clk(gmii_tx_clk),
  .gmii_tx_en(gmii_tx_en),
  .gmii_tx_er(gmii_tx_er),
  .gmii_tx_data(gmii_tx_data)
);

//计数发送数据个数，用于产生对应的数据
always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
  data_cnt <= #1 12'd0;
```

```
else if(fifo_rdreq)
  data_cnt <= #1 data_cnt + 1'b1;
else
  data_cnt <= #1 12'd0;

//UDP 包
always@(*)
begin
  case(data_cnt)
    0   : fifo_rddata = 8'h45;//协议版本,首部长度
    //服务类型
    1   : fifo_rddata = 8'h00;
    //IP 数据报总长度（IP 报头+数据）
    2   : fifo_rddata = 8'h00;
    ...
    50  : fifo_rddata = 8'h21;//!
    51  : fifo_rddata = 8'h21;//!

    default:fifo_rddata = 8'h0;
  endcase
end

//发送间隔计数器
reg [23:0]cnt;

always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
  cnt <= #1 0;
  else //计数器自增，不考虑溢出，接受溢出自动清零
  cnt <= #1 cnt + 1'b1;

//24 位 cnt 计满一次启动一次发送，该时间大约为 134ms
assign tx_go = (cnt == 24'd1);
```

修改完成后为设计分配管脚并约束电平。本次设计的引脚约束如下表 51-3 所示。

表 51-3 引脚约束表

Pin Name	Signial Name	Pin NO.		Pin Name	Signial Name	Pin NO.
PL_ENET0_TX_DATA3	rgmii_tx_data[3]	F1		PL_ENET0_GTX_CLK	rgmii_gtx_clk	C1
PL_ENET0_TX_DATA2	rgmii_tx_data[2]	F2		PL_ENET0_TX_EN	rgmii_tx_en	C2
PL_ENET0_TX_DATA1	rgmii_tx_data[1]	D1		FPGA_GCLK1	clk	T9
PL_ENET0_TX_DATA0	rgmii_tx_data[0]	D2		FPGA_KEY0	rst_n	C15

PL_ENET0_RESET	phy_rst_n	G6				

分配完引脚后为其约束电平，随后全编辑工程，烧写数据流到开发板上，打开 wireshark 工具，对本地连接进行抓包，抓取到的数据如下图 51-7 所示：

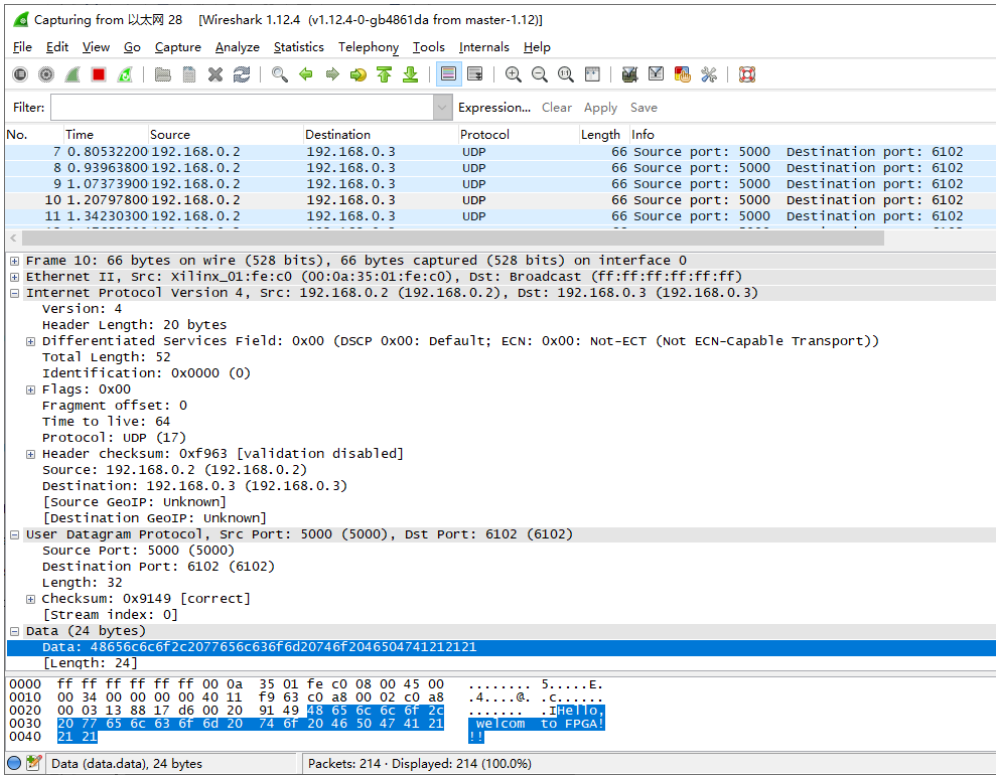


图 51-7 使用 wireshark 抓包

可以看到，IP 源地址为 192.168.0.2、目标地址为 192.168.0.3、协议类型为 UDP、源端口号为 5000、目的端口号为 6102。数据内容为：“Hello, welcome to FPGA!!!”。

51.3.3使用以太网助手接收数据

打开网络调试工具，设置协议类型为 UDP，本地 IP 地址为 192.168.0.3，本地端口号 6102，然后点击连接，接着在目的 IP 地址和目的端口号处分别填入 192.168.0.2 和 5000，然后点击一次发送按钮（此处点击发送按钮的目的是为了通过这种方式激活刚刚设置的目的 IP 和端口号，并不是真的为了发数据），则软件上就会源源不断的开始收到 FPGA 板卡发出的数据内容了，如下图 51-8 所示。

图 51-8 使用网络调试助手接收数据

51.3.4忽略 UDP 校验和测试

接下来我们测试 UDP 协议中忽略校验和，我们在代码中将 UDP 校验和部分数据修改为 0x0000，然后重新编译并烧写至开发板中。

```
129                 //UDP报头校验和
130 //              26   : fifo_rddata = 8'h91;
131 //              27   : fifo_rddata = 8'h49;
132
133 //              //UDP报头校验和  忽略
134                 26   : fifo_rddata = 8'h00;
135                 27   : fifo_rddata = 8'h00;
136
137 //              //UDP报头校验和  错误校验和
138 //              26   : fifo_rddata = 8'h12;
139 //              27   : fifo_rddata = 8'h34;
140
```

图 51-9 修改代码，忽略校验和

在 wireshark 中抓取数据包，可以看到，依旧能够正常抓取到数据，在 UDP

的 checksum 一栏，显示的状态为 none，即无 UDP 校验和。

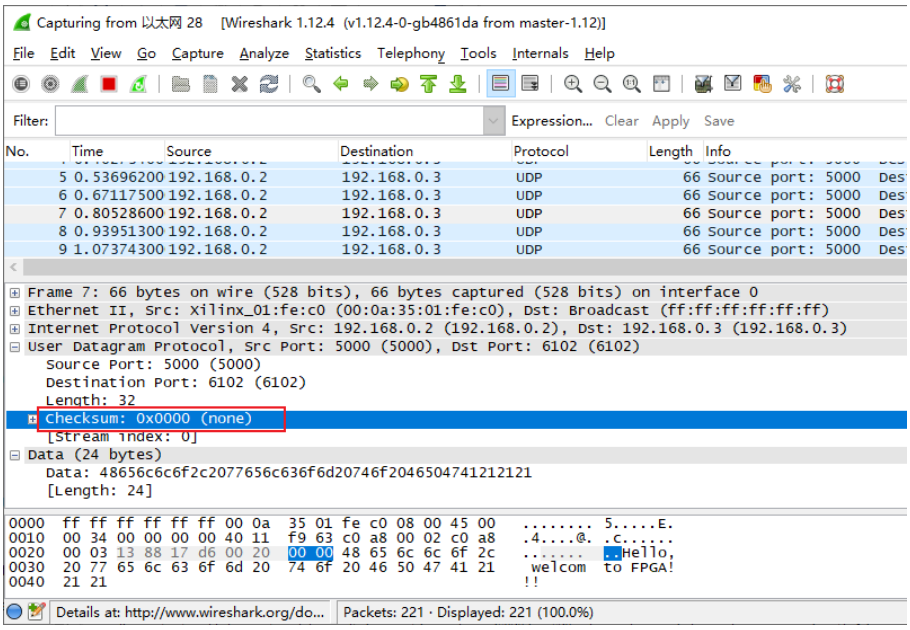


图 51-10 使用 wireshark 抓包

在网络调试工具中，依然可以正确的接收到数据。

图 51-11 使用网络调试助手接收数据（忽略校验和）

51.3.5错误 UDP 校验和测试

接下来测试 UDP 校验和错误的情况。我们在代码中将 UDP 校验和部分数据修改为非 0 但又非正确的值，如 0x1234，然后重新编译并烧写至开发板中。

```
129                 //UDP报头校验和
130 //              26  : fifo_rddata = 8'h91;
131 //              27  : fifo_rddata = 8'h49;
132 //
133 //              //UDP报头校验和  忽略
134 //              26  : fifo_rddata = 8'h00;
135 //              27  : fifo_rddata = 8'h00;
136
137 //              //UDP报头校验和  错误校验和
138                 26  : fifo_rddata = 8'h12;
139                 27  : fifo_rddata = 8'h34;
```

图 51-12 修改代码，错误校验和

在 wireshark 中抓取数据包，可以看到，依旧能够正常抓取到数据，在 UDP 的 checksum 一栏，显示的状态为 validation disabled，即软件关闭了 UDP 校验和的验证。

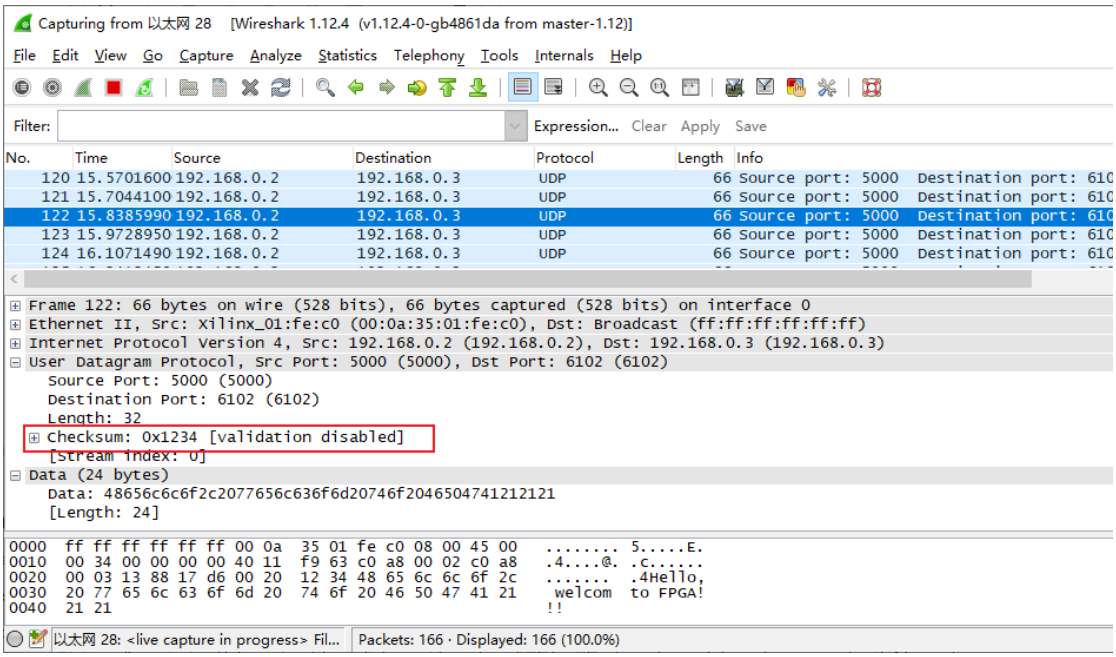


图 51-13 使用 wireshark 抓包（错误 UDP 校验和）

我们可以手动打开 wireshark 的 UDP 校验和验证，方法为选中任意一个抓取到的 UDP 数据包，右键，选择【Protocol Preferences】-> 【Data Preferences】，如下图 51-14 所示，左侧找到 UDP 协议，勾选上 Validate the UDP checksum if

possible。然后点击 apply，再 OK，如下所示。

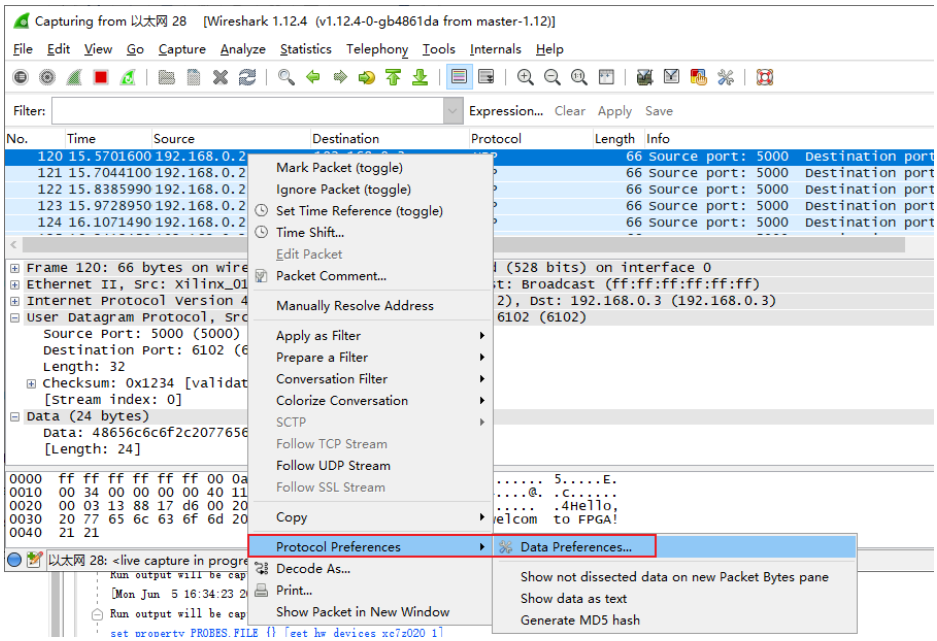


图 51-14 进入协议数据设置界面

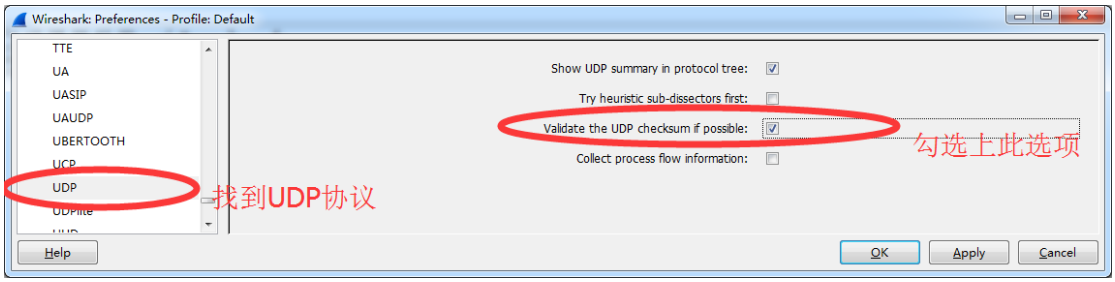


图 51-15 打开开 wireshark 的 UDP 校验和

可以看到，所有 UDP 数据包的颜色变成了深黑色，并在下方提示 checksum 错误，并给出了正确的值，如下图 51-16 所示。

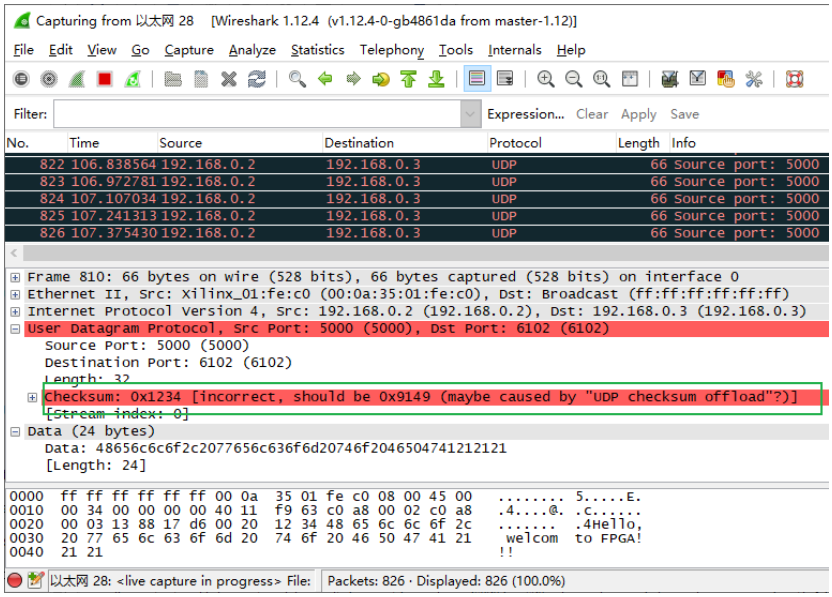


图 51-16 手动开始 UDP 校验和检测

回到以太网调试工具中，此时，以太网工具也无法收到数据了。

总结，UDP 报文的 checksum 字段可以为正确，也可以忽略（0x0000），但是不能为错，为错误一些标准的以太网工具将会将这样的数据包直接丢弃。

51.4以太网 UDP 帧发包实例（自动 IPchecksum）

在上一个实验中，我们使用了以太网组包工具组建了 UDP 数据包并发送到 PC 端。并测试了 UDP 校验和正确、忽略和错误时网络抓包工具 wireshark 的处理以及常用的网络调试工具对待这三种情况的处理，可以知道，UDP 校验和正确或者忽略（0x0000），网络调试工具都能够收到数据。而 UDP 校验和错误时，则无法收到。在这个实例中，我们没有关心 IP 首部校验和的正确与否，默认使用发包工具计算提供的值，该值是正确的。但是这种方式只适用于 IP 首部中的所有值都是确定的情况下，如确定的目标 IP 和源 IP、确定的数据长度。那么当我们需要每次发送的数据长度都可能不同，或者 IP 地址不同时，就需要 FPGA 来自动生成 IP 校验和了。

51.4.1加入自动 IP 报头校验逻辑

关于 IP 首部校验和的计算方法和计算 Verilog 代码，我们在介绍 IP 协议内容时就已经给出，本实验将使用这个代码，对 IP 首部进行实时自动校验。

实现该功能非常简单，只需要将 checksum 模块代码例化到 eth_send_test 模块中，然后连接对应端口即可，如下所示：

```
checksum checksum(
    .ver(4'h4),
    .hdr_len(4'h5),
    .tos(8'h0),
    .total_len(ip_total_len),
    .id(16'h0),
    .offset(16'h0),
    .ttl(8'h40),
    .protocol(8'h11),
    .src_ip(src_ip),
    .dst_ip(dst_ip),
    .checksum_result(ip_checksum)
);
```

由于 IP 报文中，报头很多参数在实际使用时都是固定的，因此在模块例化时，就直接将这些参数固定，例如 IP 版本（ver），首部长度（hdr_len）、tos、id、offset、ttl、上层协议类型（protocol）。最终可能每次发送数据时需要变化的，也就是数据长度（ip_total_len）、源 IP 地址（src_ip）、目的 IP 地址（dst_ip）。IP_checksum 为实时校验和的值。

目的 IP 地址和源 IP 地址一般在某个具体应用中是固定的，因此，直接使用一个定值，当然，也可以通过寄存器的方式，使用其他控制逻辑来修改。在具体应用时，可以通过修改这个值来确定系统的 IP 地址。

```
wire [31:0]src_ip;
wire [31:0]dst_ip;

assign src_ip = 32'hc0_a8_00_02;
assign dst_ip = 32'hc0_a8_00_03;
```

每次发送数据时，用户数据长度定义为 data_total_len，该值在每次发送数据前确定。

每个 UDP 报文的长度为用户数据长度（data_total_len）+UDP 报头长度（8）代码如下所示：

```
wire [15:0]udp_total_len;
assign udp_total_len = data_total_len + 8'd8;
```

每个 IP 报文的长度为用户数据长度（data_total_len）+ UDP 首部长度（8）+ IP 首部长度。一般取 IP 首部长度为 20，则 IP 数据包长度。

```
wire [15:0]ip_total_len;
assign ip_total_len = data_total_len + 8'd28;
```

51.4.2发送可变内容长度的数据

为了测试 IP 首部校验和代码是否确实能够完成 IP 首部的校验和计算，在这里设计一个可变长的数据发送序列。即使每次发送数据的长度都不一样，用户数据长度从 22 到 28 个字节循环变化，发送的内容为：

Hello, welcom to FPGA!

Hello, welcom to FPGA!t

Hello, welcom to FPGA!th

Hello, welcom to FPGA!tha

Hello, welcom to FPGA!than

Hello, welcom to FPGA!thank

Hello, welcom to FPGA!thanks

这样，就能模拟出不同数据包长度时 IP 首部校验和是否正确。控制每次发送数据包长度不一样的方法很简单，即每发完一次数据，就将 data_total_len 的值加 1，这样在 22 到 28 个字节长度之间循环变化。

为了确定上一帧数据发送完成的时间，对以太网帧发送模块 eth_send 模块代码进行一定改写，改写的内容包括：

1. 添加一个发送完成标志信号输出端口 send_done
2. 每当一帧数据发送完成，send_done 信号产生一个高脉冲

```
27      input rst_n; //复位输入
28      input tx_go; //发送启动信号，单时钟周期高脉冲使能一次发送
29      output reg send_done;
30
```

图 51-17 定义输出信号 send_done

```
165     //每次发送完成，产生发送完成标志信号
166     always@(posedge gmii_tx_clk or negedge rst_n)
167     if(!rst_n)
168         send_done <= #1  1'b0;
169     else if(cnt == 27)
170         send_done <= #1  1'b1;
171     else
172         send_done <= #1  1'b0;
```

图 51-18 产生 send_done 信号

然后，每次 send_done 信号产生高电平，则 data_total_len 自加 1，该部分代码在 eth_send_test 中如下所示：

```
//每次发送完成，将发送的数据长度累加 1
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    data_total_len <= 22;
else if(send_done)begin
    if(data_total_len >= 28)
        data_total_len <= 22;
    else
        data_total_len <= data_total_len + 1'b1;
end
else
    data_total_len <= data_total_len;
```

最后，在数据输出查找表中，将 UDP 和 IP 报头中可变部分替换为代码中定义的寄存器或者线网。

```
//UDP 包
always@(*)
begin
    case(data_cnt)
        0   : fifo_rddata = 8'h45;//协议版本,首部长度

        //服务类型
        1   : fifo_rddata = 8'h00;

        //IP 数据报总长度（IP 报头+数据）
        2   : fifo_rddata = ip_total_len[15:8];
        3   : fifo_rddata = ip_total_len[7:0];

        //数据包标识
        4   : fifo_rddata = 8'h00;
        5   : fifo_rddata = 8'h00;

        //标识+分段偏移
        6  : fifo_rddata = 8'h00;
        7  : fifo_rddata = 8'h00;

        //生存时间 64
        8  : fifo_rddata = 8'h40;

        //数据报类型 17： UDP
        9  : fifo_rddata = 8'h11;

        //IP 报头校验和
```

```
        10  : fifo_rddata = ip_checksum[15:8];
        11  : fifo_rddata = ip_checksum[7:0];

        //源地址 192.168.0.2
        12  : fifo_rddata = src_ip[31:24];
        13  : fifo_rddata = src_ip[23:16];
        14  : fifo_rddata = src_ip[15:8];
        15  : fifo_rddata = src_ip[7:0];

        //目的地址 192.168.0.3
        16  : fifo_rddata = dst_ip[31:24];
        17  : fifo_rddata = dst_ip[23:16];
        18  : fifo_rddata = dst_ip[15:8];
        19  : fifo_rddata = dst_ip[7:0];

        //源端口号 5000(0x1388)
        20  : fifo_rddata = src_port[15:8];
        21  : fifo_rddata = src_port[7:0];

        //目的端口号 6102(0x17d6)
        22  : fifo_rddata = dst_port[15:8];
        23  : fifo_rddata = dst_port[7:0];

        //UDP 数据报总长度（UDP 报头+数据）
        24  : fifo_rddata = udp_total_len[15:8];
        25  : fifo_rddata = udp_total_len[7:0];

        //UDP 报头校验和  忽略
        26  : fifo_rddata = 8'h00;
        27  : fifo_rddata = 8'h00;
//
//          //UDP 报头校验和  错误校验和
//          26  : fifo_rddata = 8'h12;
//          27  : fifo_rddata = 8'h34;
        //用户数据：Hello, welcom to FPGA!
        28  : fifo_rddata = 8'h48;//H
        29  : fifo_rddata = 8'h65;//e
        30  : fifo_rddata = 8'h6c;//l
        31  : fifo_rddata = 8'h6c;//l
        32  : fifo_rddata = 8'h6f;//0
        33  : fifo_rddata = 8'h2c;//,
        34  : fifo_rddata = 8'h20;//
        35  : fifo_rddata = 8'h77;//w
```

```
        36  : fifo_rddata = 8'h65;//e
        37  : fifo_rddata = 8'h6c;//l
        38  : fifo_rddata = 8'h63;//c
        39  : fifo_rddata = 8'h6f;//o
        40  : fifo_rddata = 8'h6d;//m
        41  : fifo_rddata = 8'h20;//
        42  : fifo_rddata = 8'h74;//t
        43  : fifo_rddata = 8'h6f;//o
        44  : fifo_rddata = 8'h20;//
        45  : fifo_rddata = 8'h46;//F
        46  : fifo_rddata = 8'h50;//P
        47  : fifo_rddata = 8'h47;//G
        48  : fifo_rddata = 8'h41;//A
        49  : fifo_rddata = 8'h21;//!

        50  : fifo_rddata = 8'h74;//t
        51  : fifo_rddata = 8'h68;//h
        52  : fifo_rddata = 8'h61;//a
        53  : fifo_rddata = 8'h6e;//n
        54  : fifo_rddata = 8'h6b;//k
        55  : fifo_rddata = 8'h73;//s
        default:fifo_rddata = 8'h00;
    endcase
end
```

Data_cnt 从 100 到 111 是在上一个实验的内容上增加的 6 个字节，其内容为“thanks”，每次发送时，根据设定的 data_total_len 的值，在“Hello, welcome to FPGA!”的末尾依次多发送 thanks 的一个到 6 个字符。

51.4.3功能验证

按照上述说明修改好后，全编译工程并下载 bit 文件到开发板，使用 wireshark 工具抓取 PC 网卡，按照之前打开 UDP 校验和检查的方式，打开 wireshark 中对 ip 校验和的验证。对抓取的报文分析如下图 51-19 所示：

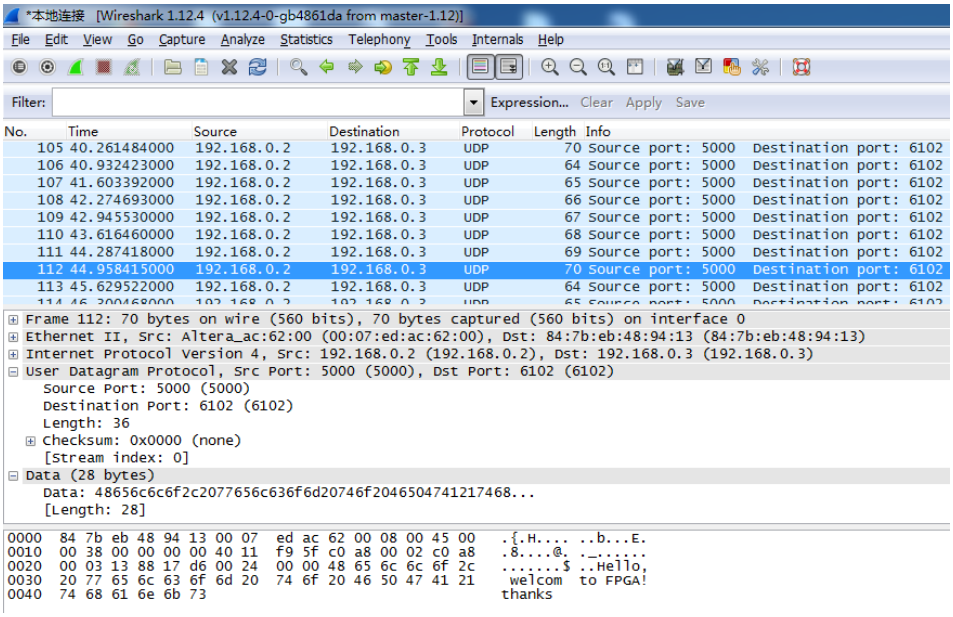


图 51-19 wireshark 抓取报文分析

可以看到，开发板循环发出 64 到 70 个字节长度的数据包，且 IP 校验和都是正确的。表明本设计能够应对各种数据长度的 UDP 报文发送。

打开网络调试工具，按图设置好各项参数，然后点击一次发送按钮，即可接收到数据，接收到的数据内容如下图 51-20 所示：

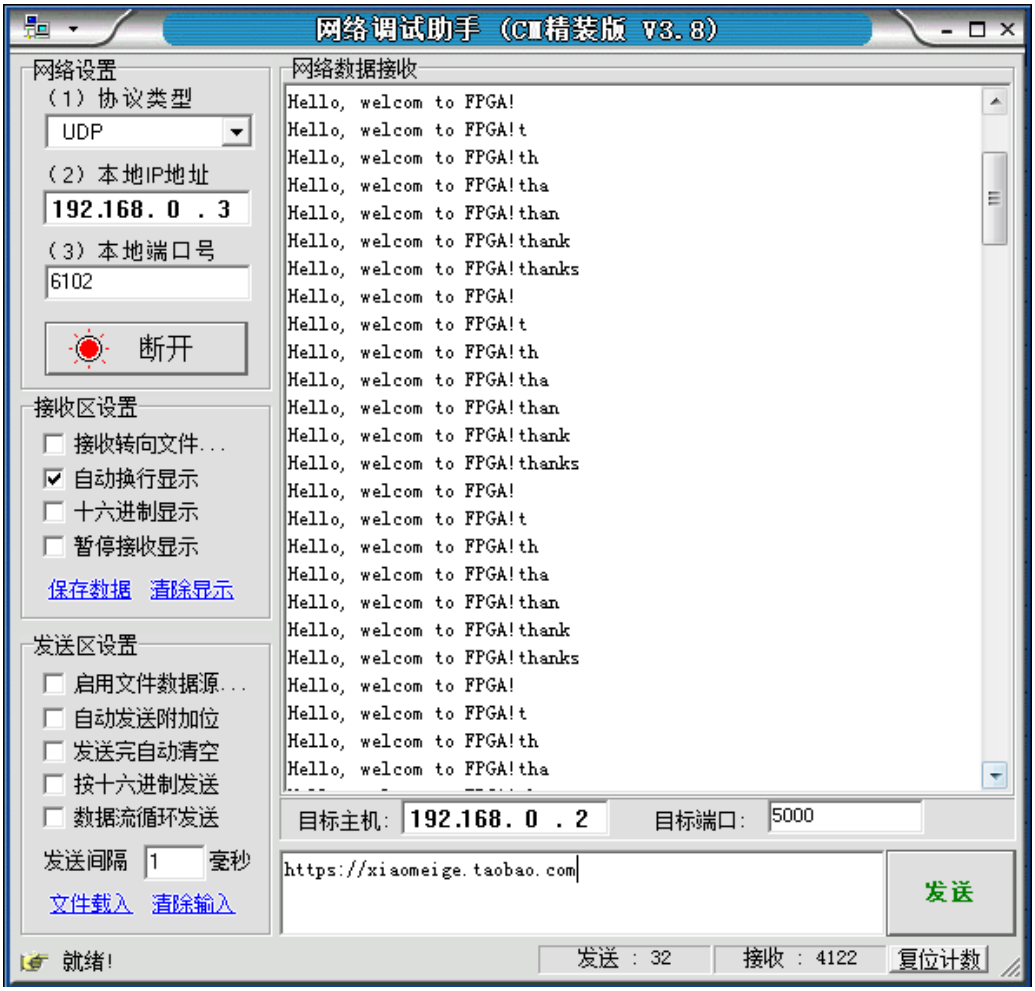


图 51-20 以太网调试助手接收到的数据

至此，基于 FPGA 的以太网 UDP 数据发送协议就设计完成了。

51.5总结

虽然本设计完成了以太网的 UDP 发送逻辑，但是设计发送的数据内容都是预先写好的确定值。而我们实际在应用以太网发送数据时，更多时候发送的是实时在变，且无法预先预知值的内容，例如摄像头采集的图像数据，ADC 采样的数据。这种情况下，需要发送的应该是从某个数据源或者存储器直接取到的数据。所以要想本发送逻辑拥有实用价值，我们应该为其设计一个 FIFO 接口，让数据源产生的数据直接写入该 FIFO，而本发送逻辑则将数据部分修改为从 FIFO 中取数，这样就具备了相当的实用价值。下一节，我们将实现该实用性的 UDP 发送逻辑。

52 实用性 UDP 发送逻辑设计与实现

工程源码	|----02_设计实例 |----ch52_udp_send
相关视频课程	
说明	

章节导读

本章将延续前面几章讲解的以太网各层级数据字段的内容，给出两种基于 FPGA 的以太网发送逻辑设计方案，同时重点对第二种以太网发送逻辑设计方案的状态机进行分析。通过发送模块的设计状态机与数据字段的对应关系，让读者对数据字段和 FPGA 状态机的关联有更加深层次的认识。

52.1以太网报文发送模块实现（第一种设计方案）

本小节的内容，适配于工程 eth_udp_tx_gmii。

52.1.1以太网发送整体设计

在讲解完以太网层层打包的主体知识点后，我们就可以开始尝试设计以太网发送控制器。

以太网 UDP 帧的格式在前面已经做了详解，并且帧内各种校验字段的产生也做了实现，接下来是对整个以太网 UDP 帧的发送进行实现。UDP 帧格式包括前导码+帧界定符、以太网头部数据、IP 头部数据、UDP 头部数据、UDP 数据、FCS 数据。根据格式对 UDP 帧发送的设计的状态机转移图如下图 52-1 所示。

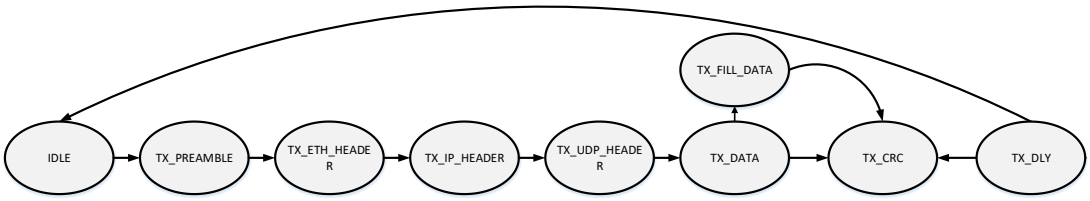


图 52-1 以太网报文发送模块状态机

- IDLE：空闲状态
- TX_PREAMBLE：发送以太网的 7 个字节的前导码和 1 字节的帧首界定符状态
- TX_ETH_HEADER：发送以太网头部数据状态

- TX_IP_HEADER：发送 IP 头部数据状态
- TX_UDP_HEADER：发送 UDP 头部数据状态
- TX_UDP_DATA：发送 UDP 数据状态
- TX_FILL_DATA：发送填充数据状态，当需要发送的有效数据的长度比较小的时候，导致以太网帧的数据段小于 46 字节就需要用 0 来填充到 46 字节，该状态就是用来应对这种情况下，发送填充 0 相应个数的 0 来满足以太网帧数据段长度要求。
- TX_CRC：发送 FCS 校验数据状态
- TX_DLY：以太网帧间隙，千兆以太网每一帧之间都需要间隔至少 96ns 的时间。

在上面的各个状态中只需要按照协议在各个状态发送指定的数据即可。状态机使用 2 段式状态机实现。FPGA 上电后进入 IDLE 状态，在输入 tx_en_pulse 为 1 时就进入到以太网 UDP 帧的发送，根据以太网 UDP 帧的协议，首先进入到发送前导码和帧首界定符的状态 TX_PREAMBLE。在 7 个字节前导码和 1 个字节帧首定界符发送完成后（通过计数器 cnt_preamble 对发送数据字节数进行计数，即 cnt_preamble 计数达到 7）进入到 TX_ETH_HEADER 状态。该状态下发送以太网头部数据，当以太网头部的 14 个字节数据发送完成后进入到 TX_IP_HEADER 状态。依次类推，在不同的状态下发送指定个数字节的数据。发送完成后就进入到下一状态。这里需要说明的是，考虑到发送的有效数据字节个数可能会小于 46 字节（以太网帧数据个数最小值），在状态机设计上增加了 TX_FILL_DATA 状态。当发送的以太网帧数据段字节个数小于 46（对应以太网 UDP 帧数据段个数为 46-20 个 IP 头部数据-8 个 UDP 头部数据 = 18）时，就进入到 TX_FILL_DATA 状态继续发送无效数据 0 凑足到以太网数据段最小字节个数。状态机实现的代码如下。

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    curr_state <= IDLE;
else
    curr_state <= next_state;

always@(*)
begin
    case(curr_state)
    IDLE:
```

```
            if(tx_en_pulse)
            next_state = TX_PREAMBLE;
            else
            next_state = IDLE;

        TX_PREAMBLE:
            if(cnt_preamble == 4'd7)
            next_state = TX_ETH_HEADER;
            else
            next_state = TX_PREAMBLE;

        TX_ETH_HEADER:
            if(cnt_eth_header == 4'd13)
            next_state = TX_IP_HEADER;
            else
            next_state = TX_ETH_HEADER;

        TX_IP_HEADER:
            if(cnt_ip_header == 5'd19)
            next_state = TX_UDP_HEADER;
            else
            next_state = TX_IP_HEADER;

        TX_UDP_HEADER:
            if(cnt_udp_header == 4'd7)
            next_state = TX_DATA;
            else
            next_state = TX_UDP_HEADER;

        TX_DATA:
            if(data_length_reg<5'd18&&cnt_data == data_length_reg - 1'b1)
            next_state = TX_FILL_DATA;
            else if(cnt_data == data_length_reg - 1'b1)
            next_state = TX_CRC;
            else
            next_state = TX_DATA;

        TX_FILL_DATA:
            if(cnt_fill_data == 5'd17 - data_length_reg)
            next_state = TX_CRC;
            else
            next_state = TX_FILL_DATA;

        TX_CRC:
```

```
            if(cnt_crc == 2'd3)
            next_state = TX_DLY;
            else
            next_state = TX_CRC;

        TX_DLY:
            if(dly_cnt == 4'd11)
            next_state = IDLE;
            else
            next_state = TX_DLY;

        default:next_state = IDLE;

        endcase
    end
```

上面各状态的跳转均是在各状态发送字节计数器的控制下进行跳转，每个状态均有一个单独的计数器对本状态发送的数据字节数进行计数。计数达到本状态需要发送字节个数时，状态进行跳转，各状态发送数据字节的计数器代码如下：

```
//cnt_preamble
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_preamble <= 4'd0;
else if(curr_state == TX_PREAMBLE)
    cnt_preamble <= cnt_preamble + 1'b1;
else
    cnt_preamble <= 4'd0;

//cnt_eth_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_eth_header <= 4'd0;
else if(curr_state == TX_ETH_HEADER)
    cnt_eth_header <= cnt_eth_header + 1'b1;
else
    cnt_eth_header <= 4'd0;

//cnt_ip_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_ip_header <= 5'd0;
else if(curr_state == TX_IP_HEADER)
```

```
    cnt_ip_header <= cnt_ip_header + 1'b1;
else
    cnt_ip_header <= 5'd0;

//cnt_udp_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_udp_header <= 4'd0;
else if(curr_state == TX_UDP_HEADER)
    cnt_udp_header <= cnt_udp_header + 1'b1;
else
    cnt_udp_header <= 4'd0;

//cnt_data
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_data <= 16'd0;
else if(curr_state == TX_DATA)
    cnt_data <= cnt_data + 1'b1;
else
    cnt_data <= 16'd0;

//cnt_fill_data
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_fill_data <= 5'd0;
else if(curr_state == TX_FILL_DATA)
    cnt_fill_data <= cnt_fill_data + 1'b1;
else
    cnt_fill_data <= 5'd0;

//cnt_crc
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_crc <= 5'd0;
else if(curr_state == TX_CRC)
    cnt_crc <= cnt_crc + 1'b1;
else
    cnt_crc <= 5'd0;

//dly_cnt
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    dly_cnt <= 5'd0;
```

```
else if(curr_state == TX_DLY)
    dly_cnt <= dly_cnt + 1'b1;
else
 dly_cnt <= 5'd0;
```

状态机的设计就基本完成了，接下来就需要在各个状态发送指定的以太网帧的数据。为了防止在一次以太网数据发送过程中，发送以太网报文源/目的 MAC、IP 和端口号以及报文中各种长度字段发送变化，在一次报文开始发送就将这些数据进行寄存。具体代码如下：

```
//将以太网报文源/目的参数寄存
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    dst_mac_reg  <= 48'd0;
    src_mac_reg  <= 48'd0;
    dst_ip_reg   <= 32'd0;
    src_ip_reg   <= 32'd0;
    dst_port_reg <= 16'd0;
    src_port_reg <= 16'd0;
end
else if(tx_en_pulse)
begin
    dst_mac_reg  <= dst_mac;
    src_mac_reg  <= src_mac;
    dst_port_reg <= dst_port;
    src_port_reg <= src_port;
    dst_ip_reg   <= dst_ip;
    src_ip_reg   <= src_ip;
end

    //将以太网报文数据部分长度参数寄存
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    data_length_reg  <= 16'h0;
    IP_total_len_reg <= 16'h0;
    udp_length_reg   <= 16'h0;
end
else if(tx_en_pulse)
begin
    data_length_reg  <= data_length;
    IP_total_len_reg <= IP_total_len;
    udp_length_reg   <= udp_length;
```

```
end
```

上面 data_length 表示的是以太网中 UDP 帧中数据段的个数，根据以太网 UDP 帧协议可知，UDP 头部长度字段的数据等于 data_length + 8'd8，IP 头部长度字段的数据就为 data_length + 8'd8 + 8'd20，这 3 个长度数据之间的关系用代码表示如下。

```
assign udp_length = data_length + 8'd8;   //udp_header: 8byte
assign IP_total_len = udp_length + 8'd20; //ip_header: 20byte
```

以太网 UDP 帧数据的发送只需要在对应的状态发送对应的数据即可。具体代码如下：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    tx_en   <= 1'b0;
    tx_data <= 8'h00;
end
else
begin
    case(curr_state)
    IDLE:
    begin
        tx_en   <= 1'b0;
        tx_data <= 8'h00;
    end

    TX_PREAMBLE:
    begin
        tx_en <= 1'b1;
        if(cnt_preamble <= 4'd6)
        tx_data <= 8'h55;
        else
        tx_data <= 8'hd5;
    end

    TX_ETH_HEADER:
    begin
        tx_en <= 1'b1;
        case(cnt_eth_header)
        4'd0:   tx_data <= dst_mac_reg[47:40];
        4'd1:   tx_data <= dst_mac_reg[39:32];
        4'd2:   tx_data <= dst_mac_reg[31:24];
        4'd3:   tx_data <= dst_mac_reg[23:16];
```

```
        4'd4:   tx_data <= dst_mac_reg[15:8];
        4'd5:   tx_data <= dst_mac_reg[7:0];
        4'd6:   tx_data <= src_mac_reg[47:40];
        4'd7:   tx_data <= src_mac_reg[39:32];
        4'd8:   tx_data <= src_mac_reg[31:24];
        4'd9:   tx_data <= src_mac_reg[23:16];
        4'd10:  tx_data <= src_mac_reg[15:8];
        4'd11:  tx_data <= src_mac_reg[7:0];
        4'd12:  tx_data <= ETH_type[15:8];
        4'd13:  tx_data <= ETH_type[7:0];
        default:tx_data <= 8'h00;
        endcase
    end

    TX_IP_HEADER:
    begin
        tx_en <= 1'b1;
        case(cnt_ip_header)
        5'd0:   tx_data <= {IP_ver,IP_hdr_len};
        5'd1:   tx_data <= IP_tos;
        5'd2:   tx_data <= IP_total_len_reg[15:8];
        5'd3:   tx_data <= IP_total_len_reg[7:0];
        5'd4:   tx_data <= IP_id[15:8];
        5'd5:   tx_data <= IP_id[7:0];
        5'd6:   tx_data <= {IP_rsv,IP_df,IP_mf,IP_frag_offset[12:8]};
        5'd7:   tx_data <= IP_frag_offset[7:0];
        5'd8:   tx_data <= IP_ttl;
        5'd9:   tx_data <= IP_protocol;
        5'd10:  tx_data <= IP_check_sum[15:8];
        5'd11:  tx_data <= IP_check_sum[7:0];
        5'd12:  tx_data <= src_ip_reg[31:24];
        5'd13:  tx_data <= src_ip_reg[23:16];
        5'd14:  tx_data <= src_ip_reg[15:8];
        5'd15:  tx_data <= src_ip_reg[7:0];
        5'd16:  tx_data <= dst_ip_reg[31:24];
        5'd17:  tx_data <= dst_ip_reg[23:16];
        5'd18:  tx_data <= dst_ip_reg[15:8];
        5'd19:  tx_data <= dst_ip_reg[7:0];
        default:tx_data <= 8'h00;
        endcase
    end

    TX_UDP_HEADER:
    begin
```

```
        tx_en <= 1'b1;
        case(cnt_udp_header)
        4'd0:   tx_data <= src_port_reg[15:8];
        4'd1:   tx_data <= src_port_reg[7:0];
        4'd2:   tx_data <= dst_port_reg[15:8];
        4'd3:   tx_data <= dst_port_reg[7:0];
        4'd4:   tx_data <= udp_length_reg[15:8];
        4'd5:   tx_data <= udp_length_reg[7:0];
        4'd6:   tx_data <= udp_check_sum[15:8];
        4'd7:   tx_data <= udp_check_sum[7:0];
        default:tx_data <= 8'h00;
        endcase
    end

    TX_DATA:
    begin
        tx_en <= 1'b1;
        tx_data <= payload_dat_i;
    end

    TX_FILL_DATA:
    begin
        tx_en <= 1'b1;
        tx_data <= 8'h00;
    end

    TX_CRC:
    begin
        tx_en <= 1'b1;
        tx_data <= 8'h00;
    end

    default:
    begin
        tx_en   <= 1'b0;
        tx_data <= 8'h00;
    end
    endcase
end
```

从代码中可以看到，在发送 UDP 报文数据段数据（也就是需要通过以太网发送数据的有效的数据）时，是需要从外部模块（可以看作是从外部 FIFO 中获取数据）获取数据的（需要满足数据已经提前存放在外部模块中，并且在获取

这些数据时，数据是提前一拍出来的，类似与 FIFO 的读数据模式是 First Word Fall Through 模式）。这样产生一个获取数据的请求信号 payload_req_o 输出，外部模块根据这个请求信号提供数据给以太网发送模块。产生 payload_req_o 的代码如下所示：

```
assign payload_req_o = (curr_state == TX_DATA) ? 1'b1 : 1'b0;
```

需要注意的是上面数据发送的代码中在 CRC 数据发送状态 TX_CRC 发送的数据是 0。因为 CRC 的计算是通过 tx_data 计算得出，这个时候 CRC 还未计算出来，先用 0 代替，并且代码中 tx_data 也并非是立马就要由管脚发送出去的数据。需要将 tx_data 延迟几个时钟周期，使得在发送 CRC 数据的时候 CRC 已经计算出结果。下图 52-2 是 CRC 计算的时序波形图。

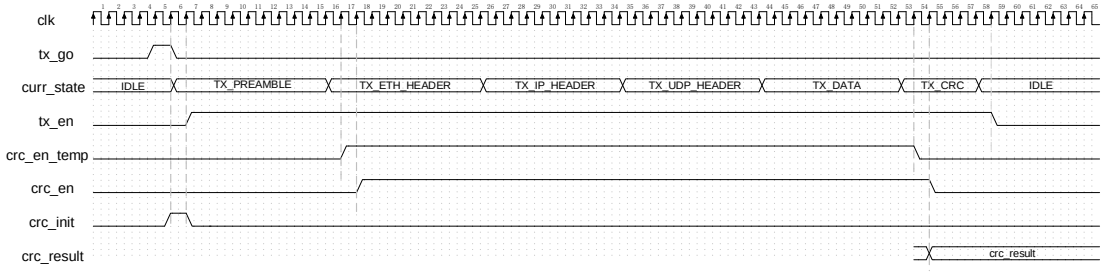


图 52-2 CRC 计算时序波形图

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    crc_init <= 1'b0;
else if (tx_en_pulse && (curr_state == IDLE))
    crc_init <= 1'b1;
else
    crc_init <= 1'b0;

always@(posedge clk125m or posedge reset_p)
if(reset_p)
    rc_en_temp <= 1'b0;
else if (curr_state == TX_ETH_HEADER || curr_state == TX_IP_HEADER ||
     curr_state  ==  TX_UDP_HEADER  ||  curr_state  ==  TX_FILL_DATA  ||
curr_state == TX_DATA)
    crc_en_temp <= 1'b1;
else
    crc_en_temp <= 1'b0;

always@(posedge clk125m or posedge reset_p)
if(reset_p)
    crc_en <= 1'b0;
```

```
else if (crc_en_temp)
    crc_en <= 1'b1;
else
    crc_en <= 1'b0;

always@(posedge clk125m or posedge reset_p)
if(reset_p)
    crc_in <= 8'h00;
else if(crc_en_temp)
    crc_in <= tx_data;
else
    crc_in <= crc_in;

crc32_d8 crc32_d8
(
    .clk         (clk125m    ),
    .reset_p     (reset_p    ),

    .data        (crc_in     ),
    .crc_init    (crc_init   ),
    .crc_en      (crc_en     ),
    .crc_result  (crc_result )//latency=1
);
```

直接将数据 tx_data 作为以太网报文发送出去是不行的，因为该数据包中 CRC 字段全是 0，还未将里面 CRC 字段更新成计算后的 CRC 数据。由于计算的 CRC 结果可被使用的时刻相对 tx_data 延迟 2 个时钟周期，在更新发送数据包中 CRC 数据的时序波形图如下所示。

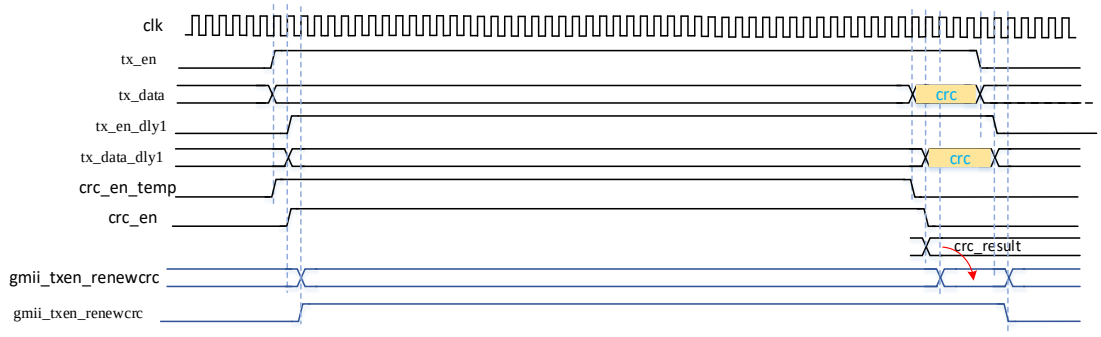


图 52-3 更新发送数据包中 CRC 数据时序波形图

实现代码如下：

```
assign crc_state = curr_state == TX_CRC;
```

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    tx_en_dly1   <= 1'b0;
    tx_data_dly1 <= 8'h00;
end
else
begin
    tx_en_dly1   <= tx_en;
    tx_data_dly1 <= tx_data;
end

always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    crc_state_dly1 <= 1'b0;
    crc_state_dly2 <= 1'b0;
    cnt_crc_dly1   <= 2'd0;
    cnt_crc_dly2   <= 2'd0;
end
else
begin
    crc_state_dly1 <= crc_state;
    crc_state_dly2 <= crc_state_dly1;
    cnt_crc_dly1   <= cnt_crc;
    cnt_crc_dly2   <= cnt_crc_dly1;
end

always@(posedge clk125m or posedge reset_p)
if(reset_p)
    gmii_txd_renewcrc <= 8'h00;
else if(crc_state_dly2)
begin
    case(cnt_crc_dly2)
    2'd0:gmii_txd_renewcrc <= crc_result[7:0];
    2'd1:gmii_txd_renewcrc <= crc_result[15:8];
    2'd2:gmii_txd_renewcrc <= crc_result[23:16];
    2'd3:gmii_txd_renewcrc <= crc_result[31:24];
    endcase
end
else
    gmii_txd_renewcrc <= tx_data_dly1;

always@(posedge clk125m or posedge reset_p)
```

```
if(reset_p)
    gmii_txen_renewcrc <= 1'b0;
else if(tx_en_dly1)
    gmii_txen_renewcrc <= 1'b1;
else
gmii_txen_renewcrc <= 1'b0;
```

gmii_txen_renewcrc 和 gmii_txd_renewcrc 分别打一拍作为输出信号 gmii_txen 和 gmii_txd。具体代码如下：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    gmii_txen <= 1'b0;
    gmii_txd  <= 8'h00;
end
else
begin
    gmii_txen <= gmii_txen_renewcrc;
gmii_txd  <= gmii_txd_renewcrc;
end
```

至此，以太网 UDP 帧发包接口的设计就完成了，接下来是进行设计模块的仿真验证。仿真验证发送的数据模拟发送 23 个字节数据报“Hello, welcome to FPGA!”。仿真代码设计如下。仿真中设置的源/目的 MAC、IP 和端口均与前面使用通过“小兵以太网测试仪”软件封装报文一致。

```
`timescale 1ns / 1ps
`define CLK_PERIOD 8

module eth_udp_tx_gmii_tb();
    reg         clk125M;
    reg         reset_n;

    reg         tx_en_pulse;
    wire        tx_done;

    wire        payload_req_o;
    reg [7:0]   payload_dat_i;

    wire        gmii_tx_clk;
    wire        gmii_txen;
    wire[7:0]   gmii_txd;
    reg [15:0] tx_byte_cnt;
```

```
    eth_udp_tx_gmii eth_udp_tx_gmii
    (
    .clk125m        (clk125M                ),
    .reset_p        (~reset_n                ),

    .tx_en_pulse    (tx_en_pulse            ),
    .tx_done        (tx_done                ),

    .dst_mac        (48'hC8_5B_76_DD_0B_38 ),
    .src_mac        (48'h00_0a_35_01_fe_c0 ),
    .dst_ip         (32'hc0_a8_00_03        ),
    .src_ip         (32'hc0_a8_00_02        ),
    .dst_port       (16'd6000               ),
    .src_port       (16'd5000               ),

    .data_length    (23                     ),

    .payload_req_o (payload_req_o          ),
    .payload_dat_i (payload_dat_i          ),

    .gmii_tx_clk    (gmii_tx_clk            ),
    .gmii_txen      (gmii_txen              ),
    .gmii_txd       (gmii_txd               )
    );

    //clock generate
    initial clk125M = 1'b1;
    always #(`CLK_PERIOD/2)clk125M = ~clk125M;

    always@(posedge clk125M or negedge reset_n)
    if(!reset_n)
        tx_byte_cnt <= 16'd0;
    else if(payload_req_o)
        tx_byte_cnt <= tx_byte_cnt + 1'b1;
    else
        tx_byte_cnt <= 16'd0;

    always@(*)
    begin
        case(tx_byte_cnt)
        16'd0  : payload_dat_i = "H";
        16'd1  : payload_dat_i = "e";
        16'd2  : payload_dat_i = "l";
        16'd3  : payload_dat_i = "l";
```

```
        16'd4  : payload_dat_i = "o";
        16'd5  : payload_dat_i = ",";
        16'd6  : payload_dat_i = " ";
        16'd7  : payload_dat_i = "w";
        16'd8  : payload_dat_i = "e";
        16'd9  : payload_dat_i = "l";
        16'd10 : payload_dat_i = "c";
        16'd11 : payload_dat_i = "o";
        16'd12 : payload_dat_i = "m";
        16'd13 : payload_dat_i = "e";
        16'd14 : payload_dat_i = " ";
        16'd15 : payload_dat_i = "t";
        16'd16 : payload_dat_i = "o";
        16'd17 : payload_dat_i = " ";
        16'd18 : payload_dat_i = "F";
        16'd19 : payload_dat_i = "P";
        16'd20 : payload_dat_i = "G";
        16'd21 : payload_dat_i = "A";
        16'd22 : payload_dat_i = "!";
        default: payload_dat_i = 8'd0;

    endcase
  end

  initial
    begin
    reset_n = 0;
    tx_en_pulse = 0;
    #201;
    reset_n = 1;
    #200;

    tx_en_pulse = 1;
    #(`CLK_PERIOD);
    tx_en_pulse = 0;
    @(posedge tx_done);
    #200;
    $stop;
  end

endmodule
```

仿真波形如下图 52-4 所示。

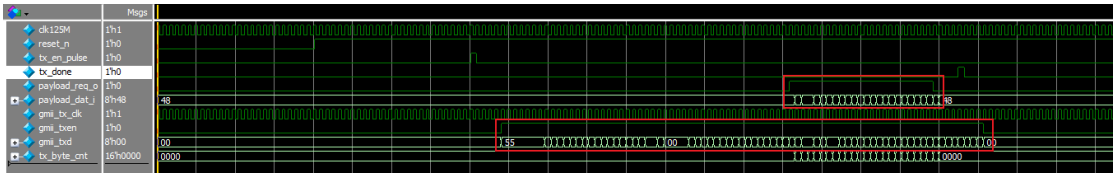


图 52-4 仿真波形图

对仿真波形中信号 payload_dat_i 和 gmii_txd 数据显示类型设置为 ASCII。可以看到发送的数据已经在报文中指定字段出现，如下图 52-5 所示。

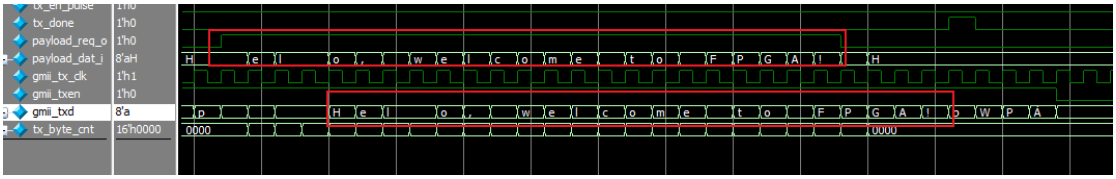


图 52-5 发送的数据字段

对仿真报文中数据与前面使用“小兵以太网测试仪”软件封装报文进行对比，可以看到除了 UDP 头部校验字段数据（仿真波形中是 0x0000，小兵以太网测试仪上是 0x9a85）不一样，其余数据完全一致。UDP 头部校验字段数据在设计中采用的是将其忽略（该字段可忽略，填全零表示忽略），与期望完全匹配的。

```
00000   c8 5b 76 dd 0b 38 00 0a 35 01 fe c0 08 00 45 00  | .[v..8..5.....E.
00010   00 33 00 00 00 00 40 11 f9 64 c0 a8 00 02 c0 a8  | .3....@..d......
00020   00 03 13 88 17 70 00 1f 9a 85 48 65 6c 6c 6f 2c  | .....p....Hello,
00030   20 77 65 6c 63 6f 6d 65 20 74 6f 20 46 50 47 41  |  welcome to FPGA
00040   21                                               | !
```

图 52-6“小兵以太网测试仪”软件封装报文

由于小兵以太网测试仪上并未显示出 CRC 字段数据，所以需要借助 CRC 计算软件计算出 CRC 值来对报文的 CRC 数据进行对比验证。仿真中以太网数据如下：

```
0xc8,0x5b,0x76,0xdd,0x0b,0x38,0x00,0x0a,0x35,0x01,0xfe,0xc0,0x08,0x00,
0x45,0x00,0x00,0x33,0x00,0x00,0x00,0x00,0x40,0x11,0xf9,0x64,0xc0,0xa8,
0x00,0x02,0xc0,0xa8,0x00,0x03,0x13,0x88,0x17,0x70,0x00,0x1f,0x00,0x00,
0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x77,0x65,0x6c,0x63,0x6f,0x6d,0x65,
0x20,0x74,0x6f,0x20,0x46,0x50,0x47,0x41,0x21
```

将数据复制粘贴到 CRC Calculator 软件中，如下图 52-7 所示，计算的 CRC 结果为 0xC0505770。可与仿真中的数据进行对比，如下图 52-8 所示，可以看出是完全一致的，以太网帧中 CRC 字段是先发送 CRC 计算结果的低字节，后发高字节。

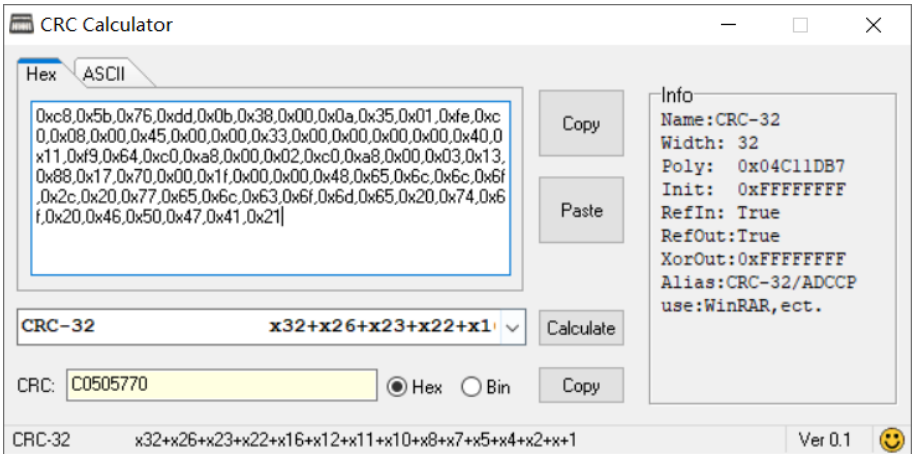


图 52-7 CRC Calculator 软件计算结果

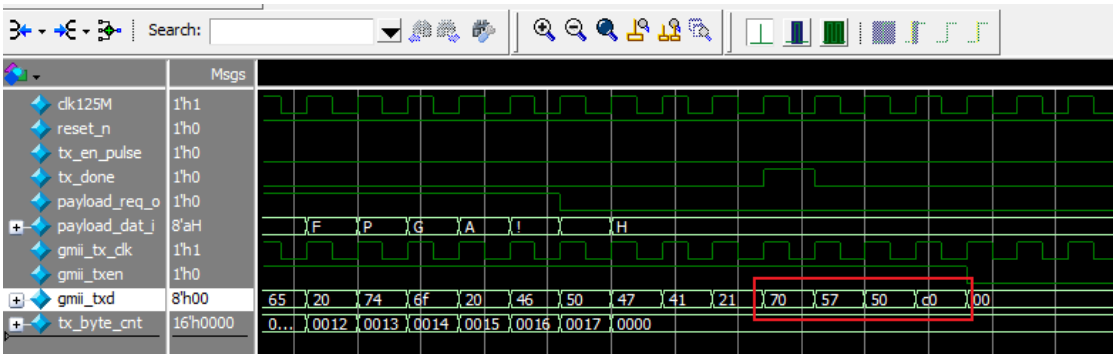


图 52-8 仿真 CRC 波形图

至此，在以太网报文发送模块的设计与验证已经完成。

52.1.2系统板级测试

接下来是设计上板验证的顶层设计文件，上板验证采取固定时间间隔发送发同一字符串“Hello, welcome to FPGA!”具体设计代码如下，固定时间间隔直接采用计数器实现。

```
module eth_udp_tx_rgmii_test(
    clk50M,
    reset_n,
    led,
    eth_reset_n,
    rgmii_tx_clk,
    rgmii_txd,
    rgmii_txen
);
    input          clk50M;
    input          reset_n;
    output         led;
```

```
    output        eth_reset_n;
    output        rgmii_tx_clk;
    output  [3:0] rgmii_txd;
    output        rgmii_txen;

    wire gmii_tx_clk;
    wire [7:0] gmii_txd;
    wire gmii_txen;

    wire       clk125M;
    wire       udp_gmii_rst_n;
    wire       pll_locked;
    reg [27:0] cnt_dly_time;
    wire       tx_en_pulse;
    wire       payload_req;
    reg [7:0]  payload_dat;
    reg [15:0] tx_byte_cnt;

    assign led            = pll_locked;
    assign eth_reset_n    = pll_locked;
    assign udp_gmii_rst_n = pll_locked;

    Gowin_PLL Gowin_PLL(
        .lock(pll_locked), //output lock
        .clkout0(clk125M), //output clkout0
        .clkin(clk50M), //input clkin
        .reset(~reset_n) //input reset
    );

    eth_udp_tx_gmii eth_udp_tx_gmii
    (
        .clk125m       (clk125M               ),
        .reset_p       (~udp_gmii_rst_n        ),

        .tx_en_pulse   (tx_en_pulse           ),
        .tx_done       (tx_done               ),

        .dst_mac       (48'hff_ff_ff_ff_ff_ff ),
        .src_mac       (48'h00_0a_35_01_fe_c0 ),
        .dst_ip        (32'hc0_a8_00_03       ),
        .src_ip        (32'hc0_a8_00_02       ),
        .dst_port      (16'd6102              ),
        .src_port      (16'd5000              ),
```

```
        .data_length    (23                     ),

        .payload_req_o (payload_req             ),
        .payload_dat_i (payload_dat             ),

        .gmii_tx_clk    (gmii_tx_clk            ),
        .gmii_txen      (gmii_txen              ),
        .gmii_txd       (gmii_txd               )
    );

     gmii_to_rgmii gmii_to_rgmii(
      .reset_n(udp_gmii_rst_n),

      .gmii_tx_clk(gmii_tx_clk),
      .gmii_txd(gmii_txd),
      .gmii_txen(gmii_txen),
      .gmii_txer(1'b0),

      .rgmii_tx_clk(rgmii_tx_clk),
      .rgmii_txd(rgmii_txd),
      .rgmii_txen(rgmii_txen)
    );

    always@(posedge clk125M or negedge udp_gmii_rst_n)
    if(!udp_gmii_rst_n)
        cnt_dly_time <= 16'd0;
    else
        cnt_dly_time <= cnt_dly_time + 1'b1;

    assign tx_en_pulse = &cnt_dly_time;

    always@(posedge clk125M or negedge reset_n)
    if(!reset_n)
        tx_byte_cnt <= 16'd0;
    else if(payload_req)
        tx_byte_cnt <= tx_byte_cnt + 1'b1;
    else
        tx_byte_cnt <= 16'd0;

    always@(*)
    begin
    case(tx_byte_cnt)
      16'd0  : payload_dat = "H";
      16'd1  : payload_dat = "e";
```

```
        16'd2  : payload_dat = "l";
        16'd3  : payload_dat = "l";
        16'd4  : payload_dat = "o";
        16'd5  : payload_dat = ",";
        16'd6  : payload_dat = " ";
        16'd7  : payload_dat = "w";
        16'd8  : payload_dat = "e";
        16'd9  : payload_dat = "l";
        16'd10 : payload_dat = "c";
        16'd11 : payload_dat = "o";
        16'd12 : payload_dat = "m";
        16'd13 : payload_dat = "e";
        16'd14 : payload_dat = " ";
        16'd15 : payload_dat = "t";
        16'd16 : payload_dat = "o";
        16'd17 : payload_dat = " ";
        16'd18 : payload_dat = "F";
        16'd19 : payload_dat = "P";
        16'd20 : payload_dat = "G";
        16'd21 : payload_dat = "A";
        16'd22 : payload_dat = "!";
        default: payload_dat = 8'd0;

    endcase
    end

endmodule
```

本次实验上板验证时，目的 IP 地址设置为全 F，也就是广播地址，适应于不同的电脑，当然，读者也可以根据自己的电脑 MAC 地址进行修改，如果不知道自己电脑网卡的 MAC 地址，就在 DOS 命令窗口，用 ipconfig –all 命令看一下。

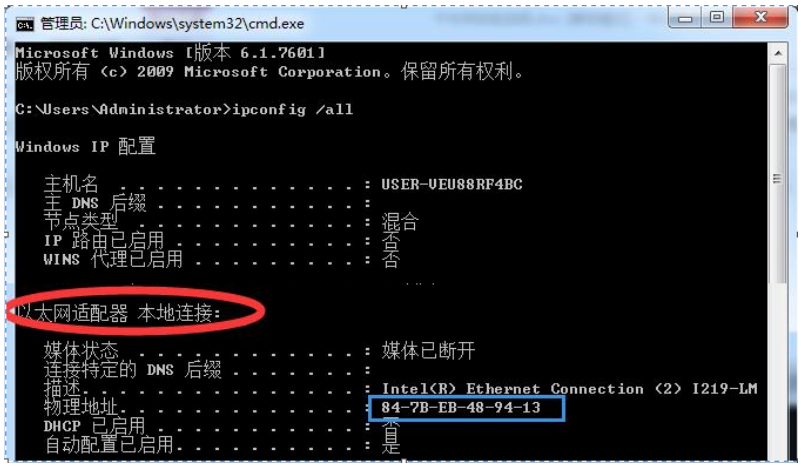


图 52-9 查看电脑 MAC 地址

上板验证的目的 IP 固定设置的 192.168.0.3，端口号固定设置为 6102。这些可根据实际情况更改，需与 FPGA 网口相连接的电脑网口的配置保持一致。

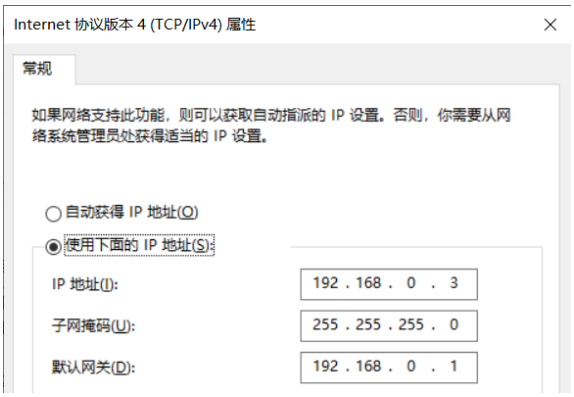


图 52-10 设置计算机静态 IP

在顶层文件设计好之后进行分析综合没有错误并且顶层仿真确认设计功能没有问题后，进行上板验证。对工程的管脚约束后，生成 Bit 文件。

下方图片为使用高云开发板的硬件连接。

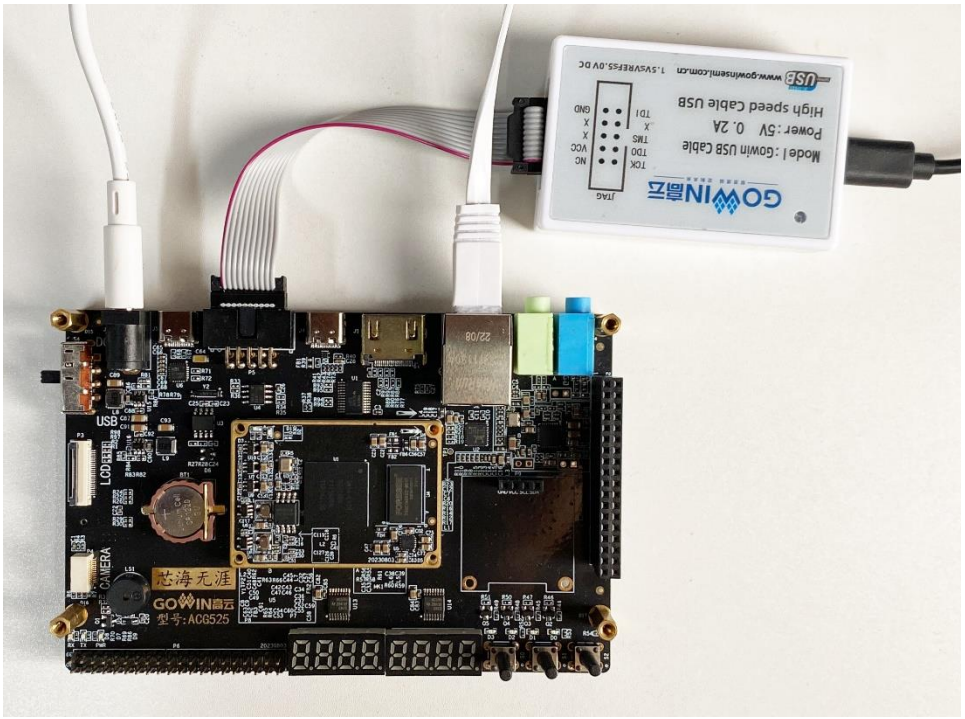


图 52-11 计算机开发板网口连接图

在电脑段打开网络调试助手软件，设置协议类型为“UDP”，本地主机地址为“192.168.0.3”，本地主机端口为“6102”。设置好后点击“打开”按钮。这个时候在工具打印窗口就会出现打印“Hello, welcome to FPGA!”” 的数据。

图 52-12 网络调试助手答应输出信息

这样，我们利用以太网帧发包设计方案实现的字符串发送功能，就得到了验证。

52.2高性能 UDP 发送逻辑设计（第二种方案）

在前面章节的内容中，我们从以太网 MAC 帧结构开始，图文并茂的介绍了以太网 MAC 帧、IP 协议、UDP 协议的详细结构，并针对每个协议都提供了对应的实验内容，带领大家一步一步的掌握了各个协议中的每个细节。相信经过前一节内容的认真学习，读者对于每个协议的每个字段的内容都有了非常清晰的认识。

随着实验的进行，我们编写了 MAC 层、IP 层和 UDP 层的逻辑并最终实现了带 FIFO 的通用 UDP 发送接口。只是，在设计中，我们采用的是分布分层设计的方法，尤其是在 MAC 层与 UDP 层的交互数据时，大量使用了组合逻辑的多路器的方式来在不同的时刻输出不同的数值。这样的写法虽然实现功能没问题，但是从时序性能的角度分析，必然不是一个优秀的设计。在 FPGA 中布线资源充足的情况下，也许能够满足时序要求，但是一旦系统中还存在其他逻辑，导致留给以太网的部分布局布线资源紧张的情况下，就有可能出现时序不满足

的情况。为此，我们有必要重新设计一个性能优异的 UDP 发送逻辑。

提升该设计的性能有两种途径，一是将其中的组合逻辑多路器部分改为时序逻辑。这种方案理论来说可行，但是在实际调试时，各层之间的数据交互要协调好有一定的困难，往往最终要通过凑时序的方式才能实现。而另一种方案则是将三层协议当做一个整体来对待，依次编码，这样就不存在多层之间的数据交互问题。

本节，将不再延续“以太网通信协议解析与实现”章节的分层介绍，分层实现的思路，而是直接将 MAC 层、IP 层、UDP 层看做一个整体，当做同一层来实现，通俗一点讲，就是将MAC帧头、IP报头、UDP报头等看成一个整体，按照顺序依次发送这些报头后，再发送用户数据，这样就能够非常直观的构建以太网帧了。

52.2.1发送状态机

以太网 UDP 帧的格式在前面已经做了讲解，并且帧内各种校验字段的产生也做了实现，接下来是对整个以太网 UDP 帧的发送进行实现。UDP 帧根据格式包括前导码+帧界定符、以太网头部数据、IP 头部数据、UDP 头部数据、UDP 数据、FCS 数据。根据格式对 UDP 帧发送的设计的状态机转移图如下图 52-13 所示。

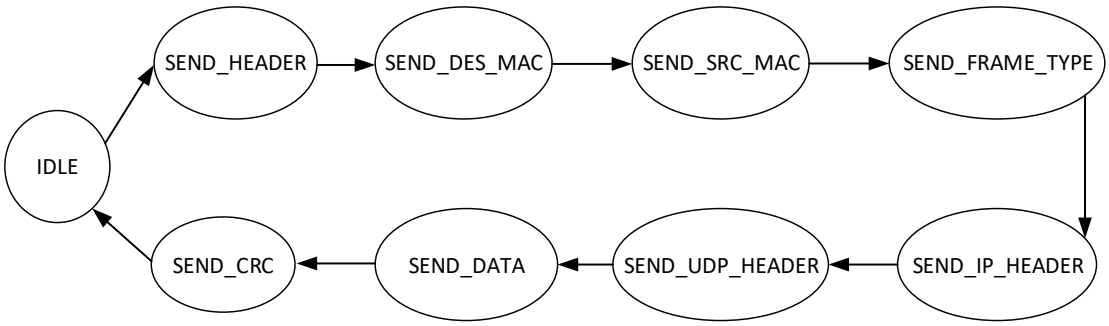


图 52-13 UDP 帧发送状态图

通过状态图可以看到，整个的发送过程就是一个顺序执行的过程，不存在状态的分支，以下为每个状态的功能说明。

- IDLE：空闲状态
- SEND_HEADER：发送以太网的 7 个字节的前导码和 1 字节的帧首界定符状态
- SEND_DES_MAC：发送 MAC 帧的目的网卡地址字段

- SEND_SRC_MAC：发送 MAC 帧的本机网卡地址字段
- SEND_FRAME_TYPE：发送 MAC 帧的长度类型字段
- SEND_IP_HEADER：发送 IP 报文报头部数据字段
- SEND_UDP_HEADER：发送 UDP 报文报头部数据字段
- SEND_DATA：发送 UDP 数据字段
- SEND_CRC：发送 FCS 校验数据字段

每一个状态中，实际又分成了很多的小的状态。以发送 MAC 帧的目的网卡地址字段（SEND_DES_MAC）为例，该状态中需要发送 48 位，也就是 6 个字节的数据。这对于 RGMII 接口来说，需要在该状态执行 6 个时钟周期，每个周期发送目的 MAC 值中的 8 位数据。对于其他状态也是一样，该状态需要发送 N 个字节的数据，就需要在该状态执行 N 个时钟周期。由于在一个确定状态中的不同时刻，需要发送的数据内容也是不一样的，所以每个状态中需要使用序列机的思想来决定在不同的时刻发送不同是数据。

52.2.2协议与序列编码

根据前面内容学习的关于 MAC 层、IP 层、UDP 层的帧结构，我们甚至可以用相同的方法，通过一张表的形式来确定每个状态的每个时刻应该发送哪个数值。下表为笔者整理 RGMII 发送接口实现时每个状态的每个时刻应该发送的数据值。

表 52-1 RGMII 发送接口每个时刻应该发送的数据值

状态	状态中时刻	发送值
SEND_HEADER	0~6	0x55
	7	0xd5
SEND_DES_MAC	0	des_mac[47:40]
	1	des_mac[39:32]
	2	des_mac[31:24]
	3	des_mac[23:16]
	4	des_mac[15:8]
	5	des_mac[7:0]
SEND_SRC_MAC	0	src_mac[47:40]
	1	src_mac[39:32]
	2	src_mac[31:24]
	3	src_mac[23:16]
	4	src_mac[15:8]
	5	src_mac[7:0]
SEND_FRAME_TYPE	0	frame_type[15:8]
	1	frame_type[7:0]

SEND_IP_HEADER	0	ip_header[0][31:24]
	1	ip_header[0][23:16]
	2	ip_header[0][15:8]
	3	ip_header[0][7:0]
	4	ip_header[1][31:24]
	5	ip_header[1][23:16]
	6	ip_header[1][15:8]
	7	ip_header[1][7:0]
	8	ip_header[2][31:24]
	9	ip_header[2][23:16]
	10	ip_header[2][15:8]
	11	ip_header[2][7:0]
	12	ip_header[3][31:24]
	13	ip_header[3][23:16]
	14	ip_header[3][15:8]
	15	ip_header[3][7:0]
	16	ip_header[4][31:24]
	17	ip_header[4][23:16]
	18	ip_header[4][15:8]
	19	ip_header[4][7:0]
SEND_UDP_HEADER	0	udp_header[0][31:24]
	1	udp_header[0][23:16]
	2	udp_header[0][15:8]
	3	udp_header[0][7:0]
	4	udp_header[1][31:24]
	5	udp_header[1][23:16]
	6	udp_header[1][15:8]
	7	udp_header[1][7:0]
SEND_DATA	-	fifo_rddata[7:0]
SEND_CRC	0	crc_result[31:24]
	1	crc_result[23:16]
	2	crc_result[15:8]
	3	crc_result[7:0]

52.2.3发送逻辑端口设计

有了上述列表，实现时只需要先给定各个状态中需要发送的值，然后启动发送，整个状态机就会按照顺序依次发送每一个字段的内容。各个状态需要发送的值，例如 MAC 地址，IP 报头、UDP 报头的值，既可以根据实际的特定应用场合设置为一个固定值，也可以通过端口的方式，由其他模块提供。

事实上，由端口的形式输入更加灵活，当需要使用固定值时，在例化时直接填入固定值即可，当需要由其他模块传递时，则连接到其他模块的对应信号即可。以下表 52-2 为笔者设计的高性能 UDP 发送模块的端口描述：

表 52-2 高性能 UDP 发送模块端口描述

端口名和属性	端口功能介绍

input Rst_n	复位信号
input Go	启动发送信号，当该信号高脉冲出现时，启动一次完整的数据帧发送
output reg Tx_Done	发送完成标志信号，当该信号出现一个时钟周期的高脉冲时标志着一次数据帧发送完成
input [47:0**]**des_mac	数据帧接收方的网卡地址
input [47:0**]**src_mac	数据帧发送方，也就是本机（这里指 FPGA 板卡）的网卡地址
input [15:0**]**des_port	UDP 协议中接收方接收数据使用的端口号
input [15:0**]**src_port	UDP 协议中发送方（本机，FPGA 板卡）发送数据使用的端口号
input [31:0**]**des_ip	IP 协议中接收方的 IP 地址
input [31:0**]**src_ip	IP 协议中发送方（本机，FPGA 板卡）的 IP 地址
input [15:0**]**data_length	发送的数据长度，IP 报头和 UDP 报头中的数据长度值都是由该值计算得到
input GMII_GTXC	GMII 接口的发送时钟，也是整个发送逻辑的工作时钟
output reg[7:0**]**GMII_TXD	GMII 接口的发送数据
output reg GMII_TXEN	GMII 接口的数据发送使能信号
input wrreq	以太网数据发送缓冲 FIFO 写请求信号
input [7:0**]**wrdata	以太网数据发送缓冲 FIFO 写数据内容
input wrclk	以太网数据发送缓冲 FIFO 写入侧时钟信号
input aclr	以太网数据发送缓冲 FIFO 清零信号。
output [12:0**]**wrusedw	以太网数据发送缓冲 FIFO 写空间已使用值

52.2.4用户数据输入

为了让该发送逻辑有更加实际的应用价值，设计时在该逻辑中加入了一个发送 FIFO，发送 FIFO 中存储需要发送的数据内容，在以太网帧发送过程中由发送逻辑在对应的时刻依次读取指定长度的值并发送。所以，使用该发送逻辑发送用户数据时，只需要将需要发送的数据写入 FIFO，给出需要发送的数据的长度值（data_length），然后再产生一个发送启动信号（Go）即可。

52.2.5性能优化

设计该逻辑的一个最大原则就是让整个逻辑的时序性能尽可能优异，真正的 FPGA 设计师都明白优异的时序是设计出来的，而不是靠约束出来的。所以要想得到优异的时序性能，必须在设计时就考虑好很多细节，并按照最优的设计方式进行。本节将重点介绍笔者在设计高性能的 UDP 发送逻辑时采用了哪些能够确实提升设计的逻辑的时序性能的方法。

52.2.5.1序列计数优化

设计时，对于整个发送逻辑中的每个状态所需要的计数，可以使用同一个计数器，在每次进入该状态之前清零，然后计数到最大值后让状态跳转到写一

个状态，但是这种写法，由于每个状态都有对其执行计数，限制计数最大值，以及清零计数值的操作，会导致该计数器的计数和清零逻辑变的异常复杂，降低整个设计的时序性能。

另外也有读者表示可以使用一个计数器从头计到尾的方式，即中间不加清零逻辑，和之前分层设计以太网协议逻辑时候的方法一样。这样的方式，确实可以降低计数器计数本身的控制逻辑，但是这样会导致整计数器的计数值比较多，需要判断的值也比较多，反而又会影响整个系统的运行频率。

所以，为了尽可能的提升系统的时序性能，设计中可以对每个状态使用一个独立的计数器进行计数，这样每个状态对应的计数器的计数和清零条件就变得固定且简单了，能够有效提升设计的时序性能。

设计中，对于每个状态，都提供了一个对应的计数器来控制该状态中每个时刻应该发送的数据内容，下表为每个状态和其对应的计数器的名称以及计数最大值关系。

表 52-3 每个状态对应的计数器名称

状态	序列计数器名称	功能描述	计数最大值
SEND_HEADER	cnt_header	前导码计数器	7
SEND_DES_MAC	cnt_des_mac	目标 MAC 地址计数器	5
SEND_SRC_MAC	cnt_src_mac	源 MAC 地址计数器	5
SEND_FRAME_TYPE	cnt_frame_type	帧类型计数器	1
SEND_IP_HEADER	cnt_ip_header	IP 报头计数器	19
SEND_UDP_HEADER	cnt_udp_header	UDP 报头计数器	7
SEND_DATA	cnt_data	用户数据计数器	data_length_reg - 1
SEND_CRC	cnt_crc	crc 计数器	3

当然了，上述分法，其实也存在继续优化的情况，比如均匀的控制每个计数器的计数最大值，保证每个计数器的计数最大值不超过 7。这样，计数器的位宽不超过 3 位，加上其他可能存在的控制逻辑，能够保证该计数器的清零判断逻辑尽可能的在一个 LUT 中完成，从而进一步降低组合逻辑链路延迟。当然了，这些建议只是方向，具体的实现情况还需要根据 EDA 软件最后的布局布线情况来确定。

52.2.5.2输入输出端口优化

由于整个设计存在多个输入和输出端口，这些端口的值随时都有可能发生变化。为了提升逻辑的时序性能，对于输入和输出端口，均采用先寄存再使用的设计方法。这种方法能够有效缩短输入端口到逻辑电路和逻辑电路到输出端

口的数据链路长度，对于 EDA 软件对设计进行布局布线是非常有利的。

对于输入端口，可以先使用寄存器寄存数据，内部的发送逻辑直接使用寄存后的数据。对于输出端口，先将需要输出的信号通过寄存器寄存一拍之后再输出。图 52-14 为不添加输入输出寄存器时可能的逻辑电路布局情况，图 52-15 为添加可输入输出寄存器后可能的逻辑电路布局情况。

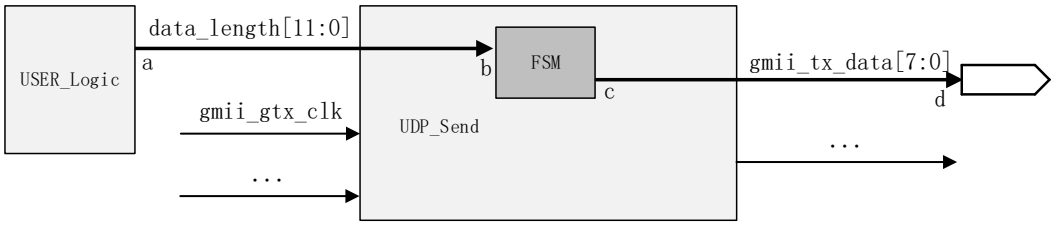


图 52-14 未添加寄存器时逻辑电路可能的布局

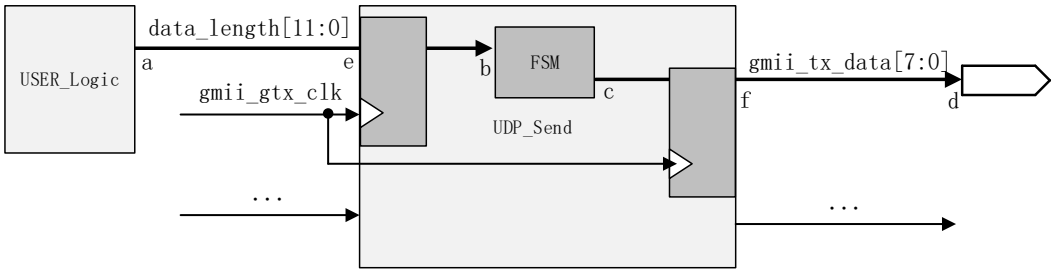


图 52-15 添加寄存器后逻辑电路可能的布局

在图中，a 点到 b 点为发送逻辑的输入端口，以 data_length 这个端口为例；c 点到 d 点为发送逻辑的输出端口，以 gmii_tx_data 端口为例。图 52-13 中，用户逻辑产生的 data_length 信号直接输入到发送逻辑内部的状态机中作为状态机的跳转条件。而状态机输出的 gmii_tx_data 信号也直接输出到 FPGA 的物理管脚。这样的实现方式存在两个问题：

1. 对于输入端口，由于输入端口的数据是由其他逻辑产生，而其他逻辑本身也较为复杂。在布局布线资源已经十分紧缺的 FPGA 系统中，这个其他逻辑到 UDP 发送逻辑的布线距离如果太远，则有可能会引入相当大的数据传输延迟。在 EDA 软件开了优化的情况下，为了降低这个延迟，EDA 软件可能会将其他逻辑与 UDP 发送逻辑拆散打乱后混合布局布线，从而对整体系统时序性能带来不可预料的影响。

2. 对于输出端口，如果由状态机产生的信号直接输出到 FPGA 管脚，这之间的信号传输路径就比较长了，而且受 EDA 软件布局布线的影响，这个信号即使是由寄存器输出的，也有可能无法将该寄存器安置到 FPGA 管脚的快速输出到寄存器上，从而降低系统时序性能。

而在输入和输出端口上插入寄存器先寄存一拍，则可以较好的解决这些问题。对于输入端口，该寄存器相当于在两个独立的逻辑块之间架起了一座桥梁，两个模块在布局布线时以该寄存器作为分界，能够有效缩短传输路径，也能够避免为了优化该路径传输延迟而对两个逻辑块混合布局布线的影响。对于输出端口，加入该寄存器后，该寄存器首先能够将发送逻辑和 FPGA 管脚的这条路径切分为 2 条较短路径，降低传输延迟，而且由于该寄存器没有其他复杂的功能，可以很容易的安置到 FPGA 管脚的快速输出寄存器，从而进一步提升系统时序性能。

以下为对输入端口进行寄存的逻辑代码，需要注意的是，这些端口上的数据在一次以太网帧发送的过程中必须保持稳定，避免在发送过程中这些参数发生变化导致发送失败。所以这些端口仅在有发送使能信号（Go）出现的时候更新一次，其他时刻全部保持自身值不变。

```
//将所有输入信号存储进入寄存器
always@(posedge GMII_GTXC)
if(Go)begin
    des_mac_reg <= #1 des_mac;
    src_mac_reg <= #1 src_mac;
    des_port_reg <= #1 des_port;
    src_port_reg <= #1 src_port;
    des_ip_reg <= #1 des_ip;
    src_ip_reg <= #1 src_ip;
    data_length_reg <= #1 data_length;
end
```

以下为对输出端口进行寄存的逻辑代码，对于输出端口，加上寄存器的目的是为了使其能够更加方便的放置到 FPGA 管脚上的快速输出寄存器，以使数据的输出延迟最小，也能保证多位宽数据的各个信号输出到 FPGA 物理 I/O 管脚上能够尽量的对齐。

```
//对输出端口加一级寄存器，以方便使用 IO 输出寄存器
always@(posedge GMII_GTXC)begin
    GMII_TXD <= #1 GMII_TXD_reg;
    GMII_TXEN <= #1 GMII_TXEN_reg;
end
```

52.2.5.3内部信号寄存器

对于 IP、UDP 报头数据，在组织时，也尽量采用寄存器的方式存储数据。对于 IP 报头数据，设计时直接建立了一个深度为 5，位宽为 32 的二维寄存器，

将对应的字段直接存入该寄存器中对应的位即可。对于 UDP 报头，也是按照相同的思路，建立了深度为 2，位宽为 32 的二维寄存器来存储报头中的各个字段，如下所示。

```
always@(posedge GMII_GTXC)
begin
    ip_header[0][31:24] <= #1 8'h45;     //协议版本+首部长度
    ip_header[0][23:16] <= #1 8'h00;     //服务类型

    //IP 数据报总长度（IP 报头+数据）
    ip_header[0][15:0]  <= #1 data_length_reg + 8'd28;
    ip_header[1][31:0] <= #1 32'd0; //数据包标识 + 标识+分段偏移
    ip_header[2][31:24] <= #1 8'd64;     //生存时间 64
    ip_header[2][23:16] <= #1 8'd17;     //UDP 协议
    ip_header[2][15:0] <= #1 ip_checksum;    //IP 校验和
    ip_header[3][31:0] <= #1 src_ip_reg;     //源 IP 地址
    ip_header[4][31:0] <= #1 des_ip_reg;     //目的 IP 地址
end

always@(posedge GMII_GTXC)
begin
    udp_header[0][31:16] <= #1 src_port_reg;     //源端口号
    udp_header[0][15:0]  <= #1 des_port_reg;     //目的端口号

    //UDP 数据报总长度（UDP 报头+数据）
    udp_header[1][31:16] <= #1 data_length_reg + 8'd8;
    udp_header[1][15:0] <= #1 16'h00;        //UDP 报头校验和忽略
end
```

52.2.6实用型 UDP 收发器 FIFO 缓存配置

本次实验中由于输入输出数据位宽为 8 位，所以 FIFO 的输入输出位宽设置为 8。因为以太网一次发送需要从 FIFO 中读取最少 2562 字节的数据，所以这里我们设置一个大于该值的存储深度 4096 即可。

为了方便实验，勾选输入输出端口的 Read Data Num 和 Write Data Num 信号，其余配置保持默认，这样可以方便知道当前 FIFO 中数据写入情况，FIFO 的配置界面如下图 52-16 所示。特别注意 Read Mode 选择 First World Fall Through 模式。

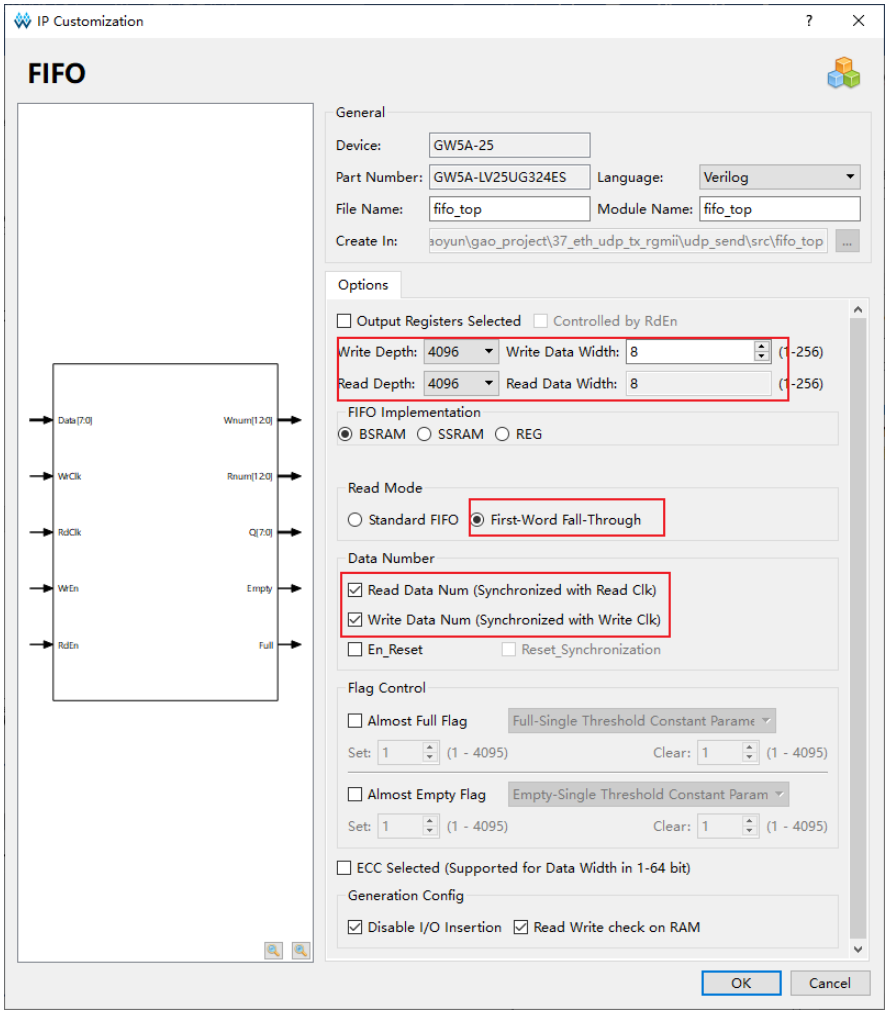


图 52-16 FIFO 配置界面

52.2.7UDP 发送模块设计（核心内容）

本次设计的 UDP 发送模块，由前面的章节实用型 UDP 收发逻辑设计与实现的理论框架设计而来。该模块将 FIFO IP，以及 CRC32_D8 校验模块例化其中，并添加设计一些发送控制信号。

我们在前面讲解 UDP 协议原理与 FPGA 实现以及实用型 UDP 收发逻辑的理论知识的时候，也专门讲解了校验文件的两种设计方法，我们也就不再进一步对校验文件进行分析了，而是直接将其例化引用。

这里，我们给设计文件添加一个顶层 UDP_Send 模块，在顶层中同时例化刚刚讲解的 FIFO IP 以及 CRC32_D8 两个模块。

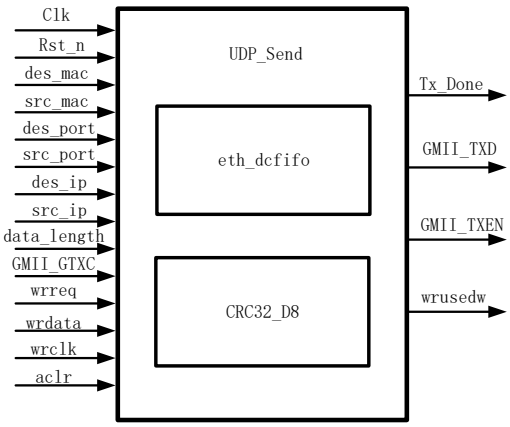


图 52-17 UDP_Send 模块框图

表 52-4 UDP_Send 模块接口功能描述

接口名称	I/O	位宽	功能描述
Clk	I	1	模块工作时钟
Rst_n	I	1	模块复位
des_mac	I	48	目标 mac 地址，相对于本工程 FPGA 而言，是 PC 机地址
src_mac	I	48	源 mac 地址，相对于本工程而言，是 FPGA 地址
des_port	I	16	目标端口号
src_port	I	16	源端口号
des_ip	I	32	目标 IP 地址
src_ip	I	32	源 IP 地址
data_length	I	16	数据长度
GMII_GTXC	I	1	GMII 发送时钟
wrreq	I	1	写请求信号
wrdata	I	8	写数据
wrclk	I	1	fifo 写时钟
aclr	I	1	fifo 复位信号
Tx_Done	O	1	发送完成信号
GMII_TXD	O	8	GMII 端口形式的发送数据
GMII_TXEN	O	1	GMII 发送使能
wrusedw	O	12	写数据计数器

下面给出 UDP 发送层的模块设计。受篇幅所限，我们将代码中描述本模块输入输出接口的代码篇幅省略。特定需要参考的同学，既可以拿本章节配套源码进行学习，又可以根据刚刚给出的端口列表进行复现。

在模块开始，给出本次发送的”长度/类型”字段。

第 1 部分：模块头部

```
`timescale 1 ns/ 1 ns
module UDP_Send(
......
);
```

```
......
    localparam FRAME_TYPE = 16'h0800;
```

对于本帧发送的一些关键信息，在本帧数据发送前作出缓存，便于后续操作。

第 2 部分：将所有输入信号储存进入寄存器

```
always@(posedge GMII_GTXC)
if(Go)begin
    des_mac_reg <= #1 des_mac;
    src_mac_reg <= #1 src_mac;
    des_port_reg <= #1 des_port;
    src_port_reg <= #1 src_port;
    des_ip_reg <= #1 des_ip;
    src_ip_reg <= #1 src_ip;
    data_length_reg <= #1 data_length;
end
```

输出端口添加一级寄存器，作为输出缓存。

第 3 部分：对输出端口加一级寄存器，方便 IO 输出寄存器

```
reg [7:0]GMII_TXD_reg;
reg GMII_TXEN_reg;

always@(posedge GMII_GTXC)begin
    GMII_TXD <= #1 GMII_TXD_reg;
    GMII_TXEN <= #1 GMII_TXEN_reg;
end
```

将上一小节讲解的输出缓存 fifo，例化到本小节内容中。

第 4 部分：例化以太网输出缓存 fifo

```
wire [7:0]fifo_rddata;
reg fifo_rdreq;

wire [11:0]rdusedw;
fifo_top fifo_top(
    .Data(wrdata), //input [7:0] Data
    .WrClk(wrclk), //input WrClk
    .RdClk(GMII_GTXC), //input RdClk
    .WrEn(wrreq), //input WrEn
    .RdEn(fifo_rdreq), //input RdEn
    .Wnum(wrusedw), //output [12:0] Wnum
    .Rnum(rdusedw), //output [12:0] Rnum
    .Q(fifo_rddata), //output [7:0] Q
    .Empty(), //output Empty
    .Full() //output Full
);
```

接下来，描述出一个专门针对 GO 信号的辅助状态机。由于 GO 信号既有可能在本工程中通过数据和状态的变化而获得拉高条件，又有可能在别的工程中，和别的模块进行数据交互而获得拉高条件，这样，我们通过单独设计 GO 信号的状态机变化，将其从主状态机中剥离，保持主状态机设计的独立性和完整性，避免 GO 信号在切换来源时，修改状态机主体。

在状态 0，如果从 fifo 中读出的数据大于等于数据长度（本例中对应图像传输的行像素数），则进入状态 1 开始发送。状态机通过判断 Tx_done 是否拉高，判断发送是否完成，如果发送完成，则进入等待延时状态 2，完成等待延时状态 2 后，进入过渡状态 3，随后跳转回归到状态 0。

第 5 部分：UDP 发送控制状态机以及延时

```
reg [1:0]ctrl_state;
reg [7:0]delay_cnt;

always@(posedge GMII_GTXC or negedge Rst_n)
if(!Rst_n)begin
    ctrl_state <= #1 0;
    Go <= #1 1'b0;
    delay_cnt <= #1 0;
end
else begin
    case(ctrl_state)
        0:
            if(rdusedw >= data_length)begin
                ctrl_state <= #1 2'd1;
                Go <= #1 1'b1;
            end
            else begin
                ctrl_state <= 0;
                Go <= #1 1'b0;
            end

        1:
            begin
                Go <= #1 1'b0;
                if(Tx_Done)
                    ctrl_state <= #1 2'd2;
                else
                    ctrl_state <= #1 2'd1;
            end
```

```
        2:
            if(delay_cnt == 8'd255)begin
                ctrl_state <= #1 2'd3;
                delay_cnt <= #1 0;
            end
            else begin
                ctrl_state <= #1 2'd2;
                delay_cnt <= #1 delay_cnt + 1'b1;
            end

        3: ctrl_state <= #1 2'd0;
    endcase
end
```

接下来，可以进行 IP 层的数据报报头排列。

第 6 部分：IP 数据报发送设计

```
reg [31:0]ip_header[4:0];
reg [31:0]des_ip_reg;
reg [31:0]src_ip_reg;
reg [15:0]data_length_reg;
wire [15:0]ip_checksum;

always@(posedge GMII_GTXC)
begin
    ip_header[0][31:24] <= #1 8'h45;     //协议版本+首部长度
    ip_header[0][23:16] <= #1 8'h00;     //服务类型
    ip_header[0][15:0]  <= #1 data_length_reg + 8'd28; //IP 数据报总长度
（IP 报头+数据）
    ip_header[1][31:0] <= #1 32'd0; //数据包标识 + 标识+分段偏移
    ip_header[2][31:24] <= #1 8'd64;     //生存时间 64
    ip_header[2][23:16] <= #1 8'd17;     //UDP 协议
    ip_header[2][15:0] <= #1 ip_checksum;    //IP 校验和
    ip_header[3][31:0] <= #1 src_ip_reg;     //源 IP 地址
    ip_header[4][31:0] <= #1 des_ip_reg;     //目的 IP 地址
end
```

完成 IP 层的数据报报头设计后，接下来完成 UDP 层数据报的报头设计。

第 7 部分：UDP 报头设计

```
reg [31:0]udp_header[1:0];
reg [15:0]des_port_reg;
reg [15:0]src_port_reg;

always@(posedge GMII_GTXC)
```

```
begin
    udp_header[0][31:16] <= #1 src_port_reg;     //源端口号
    udp_header[0][15:0]  <= #1 des_port_reg;     //目的端口号
    udp_header[1][31:16] <= #1 data_length_reg + 8'd8;
    udp_header[1][15:0] <= #1 16'h00;        //UDP 报头校验和   忽略
end
```

接下来就是 IP 数据报的各字段求和，例化校验模块。

第 8 部分：IP 数据报求和及校验模块例化

```
wire [31:0]suma,sumb;
assign suma = ip_header[0][31:16] + ip_header[0][15:0]
                + ip_header[1][31:16] + ip_header[1][15:0]
                + ip_header[2][31:16] + ip_header[3][31:16]
                + ip_header[3][15:0]  + ip_header[4][31:16] +
ip_header[4][15:0];

assign sumb = (suma[31:16] + suma[15:0]);
assign ip_checksum = sumb[31:16]? ~(sumb[31:16] + sumb[15:0]):~sumb;

wire CRC_Reset;
reg CRC_EN;

assign CRC_Reset = (state == IDLE);

CRC32_D8 CRC32_D8(
    .Clk(GMII_GTXC),
    .Reset(CRC_Reset),
    .Data_in(GMII_TXD_reg),
    .Enable(CRC_EN),
    .Crc(),
    .CrcNext(),
    .Crc_eth(crc_result)
);
```

在下一步，就是描述 MAC、IP、UDP 层层打包的主状态机。状态机的每一个状态，依次就是一个数据帧发送的每一个数据字段。为了讲解清晰，各位读者可以把状态机和 MAC 层数据组包图进行对比分析。

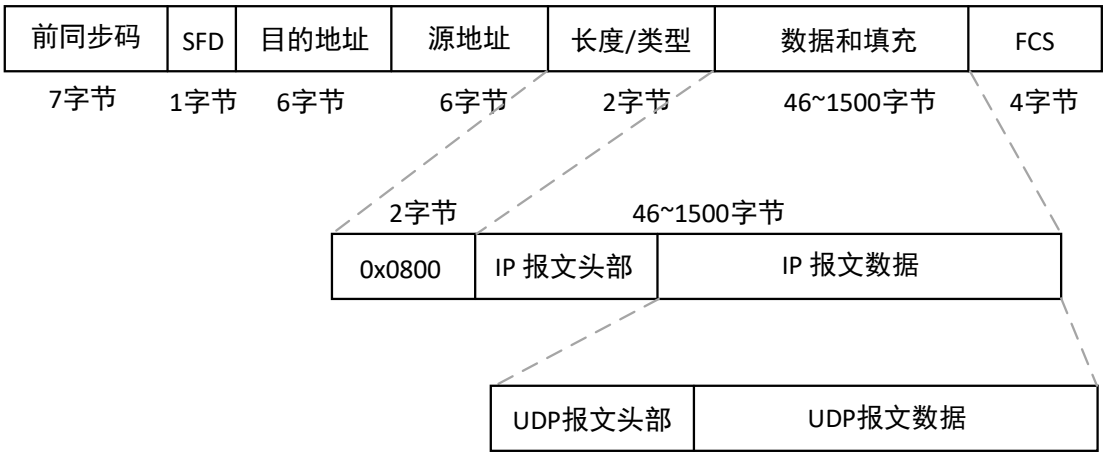


图 52-18 MAC 层打包示意图回顾

第 9 部分：MAC 层发送状态机（关于层层打包的状态机描述）

```
reg [47:0]des_mac_reg;
reg [47:0]src_mac_reg;

reg [4:0]cnt_header;     //前导码计数器
reg [4:0]cnt_des_mac;    //目标 MAC 地址计数器
reg [4:0]cnt_src_mac;    //源 MAC 地址计数器
reg [1:0]cnt_frame_type;     //帧类型计数器

reg [4:0]cnt_ip_header; //IP 报头计数器

reg [4:0]cnt_udp_header;     //UDP 报头计数器
reg [11:0]cnt_data; //数据计数器
reg [4:0]cnt_crc;    //crc 计数器

wire [31:0]crc_result;
reg [9:0]state;

localparam
    IDLE =              10'b0000000001,
    SEND_HEADER =       10'b0000000010,
    SEND_DES_MAC =  10'b0000000100,
    SEND_SRC_MAC =  10'b0000001000,
    SEND_FRAME_TYPE = 10'b0000010000,
    SEND_IP_HEADER =    10'b0000100000,
    SEND_UDP_HEADER = 10'b0001000000,
    SEND_DATA =         10'b0010000000,
    SEND_CRC =      10'b0100000000;
```

```
always@(posedge GMII_GTXC or negedge Rst_n)
if(!Rst_n)begin
    state <= #1 IDLE;
    cnt_header <= #1 0; //前导码计数器
    cnt_des_mac <= #1 0;    //目标 MAC 地址计数器
    cnt_src_mac <= #1 0;    //源 MAC 地址计数器
    cnt_frame_type <= #1 0; //帧类型计数器
    cnt_ip_header <= #1 0;  //IP 报头计数器
    cnt_udp_header <= #1 0; //UDP 报头计数器
    cnt_data <= #1 0;   //数据计数器
    cnt_crc <= #1 0;    //crc 计数器
    GMII_TXD_reg <= #1 0;
    GMII_TXEN_reg <= #1 0;
    fifo_rdreq <= #1 1'b0;
    Tx_Done <= #1 1'b0;
    CRC_EN <= #1 1'b0;
end
else begin
    case(state)
        IDLE:
            begin
                GMII_TXEN_reg <= #1 0;
                Tx_Done <= #1 1'b0;
                if(Go)
                    state <= #1 SEND_HEADER;    // 发送前导码
                else
                    state <= #1 IDLE;
            end

        SEND_HEADER:
            begin
                GMII_TXEN_reg <= #1 1;
                if(cnt_header >= 7)begin
                    cnt_header <= #1 0;
                    state <= #1 SEND_DES_MAC;
                end
                else begin
                    cnt_header <= #1 cnt_header + 1'b1;
                    state <= #1 SEND_HEADER;
                end
                case(cnt_header)
                    0,1,2,3,4,5,6:GMII_TXD_reg <= #1 8'h55;
                    7:GMII_TXD_reg <= #1 8'hd5;
```

```
                default:GMII_TXD_reg <= #1 8'h55;
            endcase
        end

    SEND_DES_MAC:
        begin
            CRC_EN <= #1 1'b1;
            if(cnt_des_mac >= 5)begin
                cnt_des_mac <= #1 0;
                state <= #1 SEND_SRC_MAC;
            end
            else begin
                cnt_des_mac <= #1 cnt_des_mac + 1'b1;
                state <= #1 SEND_DES_MAC;
            end
            case(cnt_des_mac)
                0:GMII_TXD_reg <= #1 des_mac_reg[47:40];
                1:GMII_TXD_reg <= #1 des_mac_reg[39:32];
                2:GMII_TXD_reg <= #1 des_mac_reg[31:24];
                3:GMII_TXD_reg <= #1 des_mac_reg[23:16];
                4:GMII_TXD_reg <= #1 des_mac_reg[15:8];
                5:GMII_TXD_reg <= #1 des_mac_reg[7:0];
                default:GMII_TXD_reg <= #1 8'hff;
            endcase
        end

    SEND_SRC_MAC:
        begin
            if(cnt_src_mac >= 5)begin
                cnt_src_mac <= #1 0;
                state <= #1 SEND_FRAME_TYPE;
            end
            else begin
                cnt_src_mac <= #1 cnt_src_mac + 1'b1;
                state <= #1 SEND_SRC_MAC;
            end
            case(cnt_src_mac)
                0:GMII_TXD_reg <= #1 src_mac_reg[47:40];
                1:GMII_TXD_reg <= #1 src_mac_reg[39:32];
                2:GMII_TXD_reg <= #1 src_mac_reg[31:24];
                3:GMII_TXD_reg <= #1 src_mac_reg[23:16];
                4:GMII_TXD_reg <= #1 src_mac_reg[15:8];
                5:GMII_TXD_reg <= #1 src_mac_reg[7:0];
                default:GMII_TXD_reg <= #1 8'hff;
```

```
                endcase
            end

        SEND_FRAME_TYPE:
            begin
                if(cnt_frame_type >= 1)begin
                    cnt_frame_type <= #1 0;
                    state <= #1 SEND_IP_HEADER;
                end
                else begin
                    cnt_frame_type <= #1 cnt_frame_type + 1'b1;
                    state <= #1 SEND_FRAME_TYPE;
                end
                case(cnt_frame_type)
                    0:GMII_TXD_reg <= #1 FRAME_TYPE[15:8];
                    1:GMII_TXD_reg <= #1 FRAME_TYPE[7:0];
                    default:GMII_TXD_reg <= #1 8'hff;
                endcase
            end

        SEND_IP_HEADER:
            begin
                if(cnt_ip_header >= 19)begin
                    cnt_ip_header <= #1 0;
                    state <= #1 SEND_UDP_HEADER;
                end
                else begin
                    cnt_ip_header <= #1 cnt_ip_header + 1'b1;
                    state <= #1 SEND_IP_HEADER;
                end
                case(cnt_ip_header)
                    0 :GMII_TXD_reg <= #1 ip_header[0][31:24];
                    1 :GMII_TXD_reg <= #1 ip_header[0][23:16];
                    2 :GMII_TXD_reg <= #1 ip_header[0][15:8];
                    3 :GMII_TXD_reg <= #1 ip_header[0][7:0];
                    4 :GMII_TXD_reg <= #1 ip_header[1][31:24];
                    5 :GMII_TXD_reg <= #1 ip_header[1][23:16];
                    6 :GMII_TXD_reg <= #1 ip_header[1][15:8];
                    7 :GMII_TXD_reg <= #1 ip_header[1][7:0];
                    8 :GMII_TXD_reg <= #1 ip_header[2][31:24];
                    9 :GMII_TXD_reg <= #1 ip_header[2][23:16];
                    10:GMII_TXD_reg <= #1 ip_header[2][15:8];
                    11:GMII_TXD_reg <= #1 ip_header[2][7:0];
                    12:GMII_TXD_reg <= #1 ip_header[3][31:24];
```

```
                    13:GMII_TXD_reg <= #1 ip_header[3][23:16];
                    14:GMII_TXD_reg <= #1 ip_header[3][15:8];
                    15:GMII_TXD_reg <= #1 ip_header[3][7:0];
                    16:GMII_TXD_reg <= #1 ip_header[4][31:24];
                    17:GMII_TXD_reg <= #1 ip_header[4][23:16];
                    18:GMII_TXD_reg <= #1 ip_header[4][15:8];
                    19:GMII_TXD_reg <= #1 ip_header[4][7:0];
                    default:GMII_TXD_reg <= #1 8'hff;
                endcase
            end

        SEND_UDP_HEADER:
            begin
                if(cnt_udp_header >= 7)begin
                    cnt_udp_header <= #1 0;
                    fifo_rdreq <= #1 1'b1;
                    state <= #1 SEND_DATA;
                end
                else begin
                    cnt_udp_header <= #1 cnt_udp_header + 1'b1;
                    state <= #1 SEND_UDP_HEADER;
                end
                case(cnt_udp_header)
                    0 :GMII_TXD_reg<=#1 udp_header[0][31:24];
                    1 :GMII_TXD_reg<=#1 udp_header[0][23:16];
                    2 :GMII_TXD_reg<=#1 udp_header[0][15:8];
                    3 :GMII_TXD_reg<=#1 udp_header[0][7:0];
                    4 :GMII_TXD_reg<=#1 udp_header[1][31:24];
                    5 :GMII_TXD_reg<=#1 udp_header[1][23:16];
                    6 :GMII_TXD_reg<=#1 udp_header[1][15:8];
                    7 :GMII_TXD_reg<=#1 udp_header[1][7:0];
                    default:GMII_TXD_reg <= #1 8'hff;
                endcase
            end

        SEND_DATA:
            begin
                if(cnt_data >= data_length_reg - 1)begin
                    cnt_data <= #1 0;
                    state <= #1 SEND_CRC;
                    fifo_rdreq <= #1 1'b0;
                    GMII_TXD_reg <= #1 fifo_rddata;
                end
                else begin
```

```
                    cnt_data <= #1 cnt_data + 1'b1;
                    state <= #1 SEND_DATA;
                    GMII_TXD_reg <= #1 fifo_rddata;
                end
            end

        SEND_CRC:
            begin
                CRC_EN <= #1 1'b0;
                if(cnt_crc >= 3)begin
                    cnt_crc <= #1 0;
                    state <= #1 IDLE;
                    Tx_Done <= #1 1'b1;
                end
                else begin
                    cnt_crc <= #1 cnt_crc + 1'b1;
                    state <= #1 SEND_CRC;
                end
                case(cnt_crc)
                    0 :GMII_TXD_reg <= #1 crc_result[31:24];
                    1 :GMII_TXD_reg <= #1 crc_result[23:16];
                    2 :GMII_TXD_reg <= #1 crc_result[15:8];
                    3 :GMII_TXD_reg <= #1 crc_result[7:0];
                    default:GMII_TXD_reg <= #1 8'hff;
                endcase
            end
        default:begin state <= #1 IDLE;end
    endcase
end
```

至此，关于高性能 UDP 发送逻辑的设计要点就介绍完毕了。

52.2.8UDP 发送逻辑仿真验证

首先，需要编写激励文件，将 UDP_Send 模块例化至仿真文件中，然后使用 task 事件向 FIFO 中写入两次数据，整个的仿真激励文件如下所示：

```
`timescale 1ns / 1ps

module UDP_Send_tb;
    reg Clk;
    reg Rst_n;
    wire Tx_Done;
    reg GMII_GTXC;
    reg wrreq;
```

```
    reg [7:0]wrdata;
    reg wrclk;
    reg aclr;

    wire [7:0]GMII_TXD;
    wire GMII_TXEN;
    wire [12:0]wrusedw;

    parameter data_length = 16'd16;
    parameter des_mac = 48'hFF_FF_FF_FF_FF_FF;
    parameter src_mac = 48'h00_0a_35_01_fe_c0;
    parameter des_port = 16'd6102;
    parameter src_port = 16'd5000;
    parameter des_ip = 32'hc0_a8_00_03;
    parameter src_ip = 32'hc0_a8_00_02;

    UDP_Send UDP_Send(
        .Clk(Clk),
        .Rst_n(Rst_n),

        //  Go,
        .Tx_Done(Tx_Done),

        .des_mac(des_mac),
        .src_mac(src_mac),

        .des_port(des_port),
        .src_port(src_port),

        .des_ip(des_ip),
        .src_ip(src_ip),

        .data_length(data_length),

        .GMII_GTXC(GMII_GTXC),
        .GMII_TXD(GMII_TXD),
        .GMII_TXEN(GMII_TXEN),

        .wrreq(wrreq),
        .wrdata(wrdata),
        .wrclk(wrclk),
        .aclr(aclr),
        .wrusedw(wrusedw)
    );
```

```
    initial Clk = 1;
    always #10 Clk = ~Clk;

    initial wrclk = 1;
    always #10 wrclk = ~wrclk;

    initial GMII_GTXC = 1;
    always #4 GMII_GTXC = ~GMII_GTXC;

    initial begin
        Rst_n = 0;
        #201;
        Rst_n = 1;
        #201;
        wr_data(8'h00,data_length);
        #2000;
        wr_data(8'h10,data_length);
        #2000;
        $stop;
    end

    task wr_data;
    input [7:0]data_begin;
    input [7:0]data_length;
    begin
        wrdata = data_begin;
        wrreq = 0;
        aclr = 1;
        #601;
        #1 aclr = 0;
        #401;
        repeat(data_length)
        begin
            @(posedge wrclk);
            wrdata = wrdata +1'b1;
            wrreq = 1;
        end
        @(posedge wrclk);
        wrreq = 0;
    end
  endtask
endmodule
```

仿真波形如下图 52-19 所示。

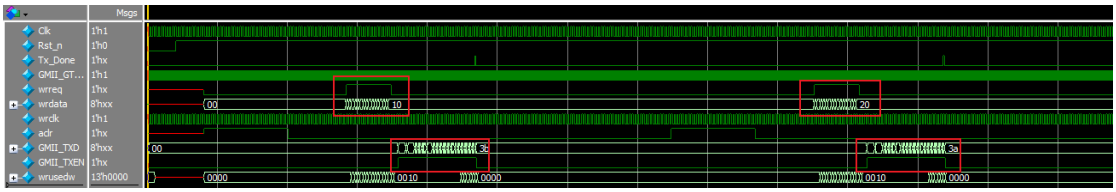


图 52-19 UDP_Send 仿真波形图

放大第一次写入 FIFO 中的数据，以及以太网第一次发送出去的数据，如下图 52-20 和图 52-21 所示。

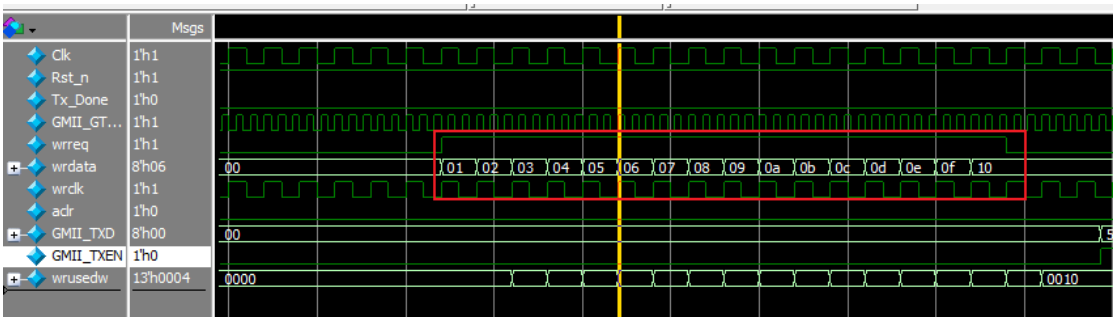


图 52-20 第一次写入 FIFO 中的数据

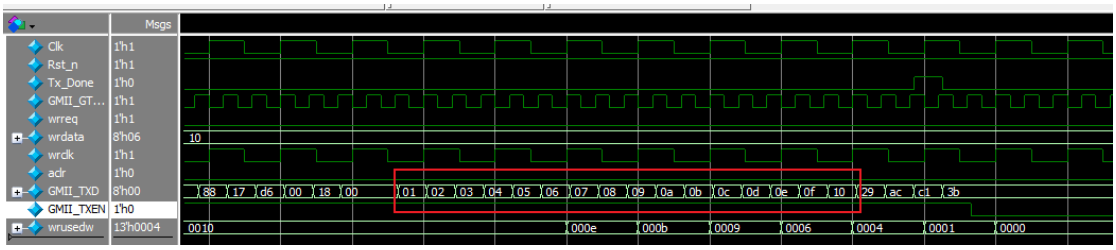


图 52-21 以太网第一次发送的数据

从上述图中可以看出，第一次写入 FIFO 中的数据和以太网第一次发送的数据完全一致。至此，通过仿真验证了 UDP_Send 模块的设计符合预期。

52.3总结

本次实验通过两种方式实现了以太网发送模块的设计，同时对两种设计方案的状态机都进行了详细介绍，在后续的实验中，我们一般都使用第一种方式实现以太网发送模块的功能。读者可以依照本节内容，尝试自行编写以太网发送模块，以此对以太网协议各个字段有进一步的了解。

53 实用性 UDP 接收逻辑设计与实现

工程源码	|----02_设计实例 |----ch53_eth_udp_rx_rgmii
相关视频课程	
说明	

章节导读

上一章中，我们完成了高性能 UDP 发送模块的设计。而完整的 UDP 应用，除了 UDP 发送外，必然还包含 UDP 接收。故此，本章将带大家实现高性能 UDP 接收模块的设计。

53.1UDP 接收模块代码设计

以太网 udp 接收是以太网 udp 发送的逆过程，因此这里不再对 udp 发送原理讲解，用户可以参考前述章节。参考上一章节，我们可以设计出如图 53-1 所示接收模块：

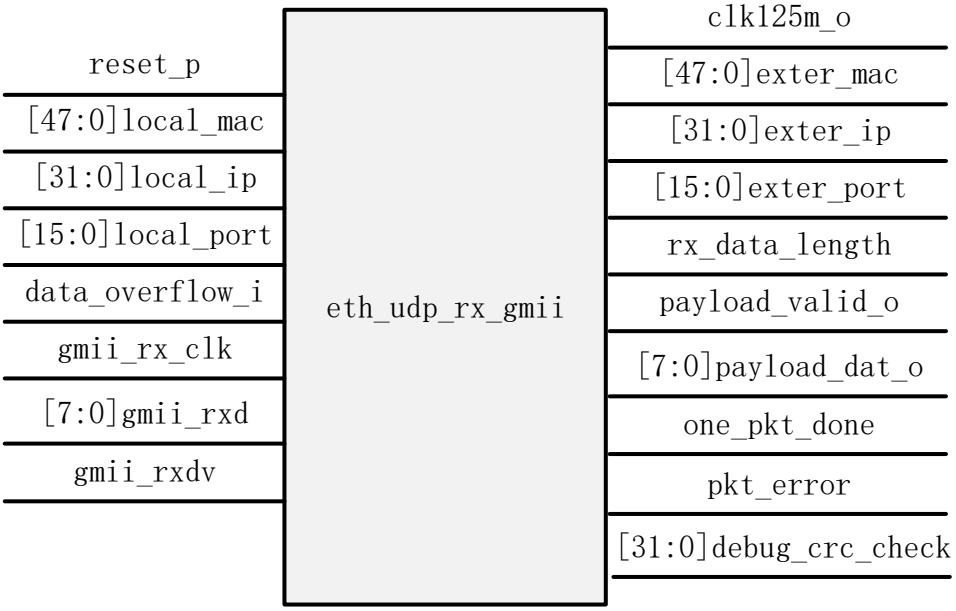


图 53-1 以太网接收模块的基本结构框图

该模块的信号说明如下表 53-1 所示。

表 53-1 以太网接收模块信号说明表

接口名称	I/O	信号意义
reset_p	I	模块复位信号，高有效
local_mac[47:0]	I	本地 mac 地址

local_ip [31:0]	I	本地 ip 地址
local_port[15:0]	I	本地端口号
data_overflow_i	I	输入数据溢出标志信号
gmii_rx_clk	I	接收数据参考时钟，时钟频率为 125MHz。RX_CLK 是由 PHY 侧提供的
gmii_rxdv	I	即 Reveive Data Valid，接收数据有效信号，作用类似于发送通道的 TX_EN
gmii_rxd[7:0]	I	即 ReceiveData，数据接收信号，共 8 根信号线
exter_mac[47:0]	O	目的 mac 地址
exter_ip[31:0]	O	目的 IP 地址
exter_port[15:0]	O	目的端口号
rx_data_length[15:0]	O	接收数据长度信号
payload_valid_o	O	输出数据有效信号
payload_dat_o[7:0]	O	8 位数据输出信号
one_pkt_done	O	以太网包传输完成信号
pkt_error	O	接收数据错误标志信号
debug_crc_check	O	调试 CRC 检验信号

在前面我们对以太网 UDP 帧格式做了讲解，UDP 帧格式包括前导码+帧界定符、以太网头部数据、IP 头部数据、UDP 头部数据、UDP 数据、FCS 数据，以太网接收模块同样是按照该格式接收数据。这里，我们主要通过状态机的方式实现以太网接收模块的功能，该模块的状态转移图如下图 53-2 所示。

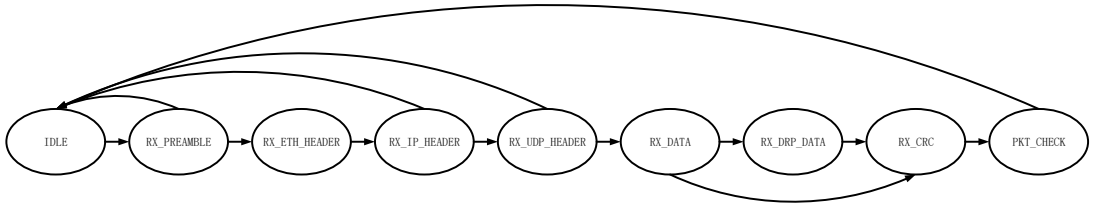


图 53-2 以太网接收模块状态转移图

下面将对各个状态的实现及功能进行简要介绍。

1. IDLE

空闲状态，当产生接收数据有效信号时，进入 RX_PREAMBLE 状态，否则处于 IDLE 状态，代码如下所示：

```
IDLE:
if(!rx_datav_dly2 && rx_datav_dly1)
    next_state = RX_PREAMBLE;
else
    next_state = IDLE;
```

上述代码中的 rx_datav_dly1 信号是将接收数据有效信号 gmii_rxdv 寄存之后打一拍得到的，rx_datav_dly2 信号是将 rx_datav_dly1 信号打一拍得到的，将 rx_datav_dly2 信号取反与 rx_datav_dly1 相与得到接收数据有效脉冲，得到该信号之后，进入到 RX_PREAMBLE 状态，rx_datav_dly1 信号和 rx_datav_dly2 信号

的实现代码如下所示，代码中对 gmii_rxd 信号也进行了寄存和打拍操作，该信号的使用将会在后面进行说明。

```
//将以太网输入的接收信号寄存
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    reg_gmii_rxd  <= 8'h00;
    reg_gmii_rxdv <= 1'b0;
end
else
begin
    reg_gmii_rxd  <= gmii_rxd;
    reg_gmii_rxdv <= gmii_rxdv;
end

//将以太网输入的接收信号寄存后打拍
always@(posedge clk125m)
begin
    rx_data_dly1  <= reg_gmii_rxd;
    rx_data_dly2  <= rx_data_dly1;
    rx_datav_dly1 <= reg_gmii_rxdv;
    rx_datav_dly2 <= rx_datav_dly1;
end
```

2. RX_PREAMBLE 状态

处于 RX_PREAMBLE 状态的时候，当以太网接收到帧界定符（D5）和 5 个前导码（55）时，进入到 RX_ETH_HEADER 状态，如果接收超过 7 个前导码，则表明此时数据接收错误，进入 IDLE 状态，代码如下所示：

```
RX_PREAMBLE:
if(rx_data_dly2 == 8'hd5 && cnt_preamble > 4'd5)
    next_state = RX_ETH_HEADER;
else if(cnt_preamble > 4'd7)
    next_state = IDLE;
else
    next_state = RX_PREAMBLE;
```

上述代码中的 rx_data_dly2 信号就是将 gmii_rxd 信号寄存之后打两拍得到的，cnt_preamble 信号是用来计数的，也就是处于 RX_PREAMBLE 状态时，得到的前导码的数据个数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
```

```
    cnt_preamble <= 4'd0;
else if(curr_state == RX_PREAMBLE && rx_data_dly2 == 8'h55)
    cnt_preamble <= cnt_preamble + 1'b1;
else
    cnt_preamble <= 4'd0;
```

3. RX_ETH_HEADER

处于 RX_ETH_HEADER 状态时，接收以太网头部数据，当接收完 14 个以太网头部数据之后，进入到 RX_IP_HEADER 状态，代码如下所示：

```
RX_ETH_HEADER:
if(cnt_eth_header == 4'd13)
    next_state = RX_IP_HEADER;
else
    next_state = RX_ETH_HEADER;
```

cnt_eth_header 信号在状态处于 RX_ETH_HEADER 时，进入计数，否则清零，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_eth_header <= 4'd0;
else if(curr_state == RX_ETH_HEADER)
    cnt_eth_header <= cnt_eth_header + 1'b1;
else
    cnt_eth_header <= 4'd0;
```

然后当处于该状态的时候，根据 cnt_eth_header 的值，依次得到 14 个字节的以太网头部数据，分别是 MAC 目的地址（6 个字节）、MAC 源地址（6 个字节）和以太网类型（2 个字节），代码如下所示：

```
//eth_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  rx_dst_mac  <= 48'h00_00_00_00_00_00;
  rx_src_mac  <= 48'h00_00_00_00_00_00;
  rx_eth_type <= 16'h0000;
end
else if(curr_state == RX_ETH_HEADER)
begin
  case(cnt_eth_header)
    4'd0 :rx_dst_mac[47:40] <= rx_data_dly2;
    4'd1 :rx_dst_mac[39:32] <= rx_data_dly2;
    4'd2 :rx_dst_mac[31:24] <= rx_data_dly2;
    4'd3 :rx_dst_mac[23:16] <= rx_data_dly2;
```

```
    4'd4 :rx_dst_mac[15:8]  <= rx_data_dly2;
    4'd5 :rx_dst_mac[7:0]   <= rx_data_dly2;

    4'd6 :rx_src_mac[47:40] <= rx_data_dly2;
    4'd7 :rx_src_mac[39:32] <= rx_data_dly2;
    4'd8 :rx_src_mac[31:24] <= rx_data_dly2;
    4'd9 :rx_src_mac[23:16] <= rx_data_dly2;
    4'd10:rx_src_mac[15:8]  <= rx_data_dly2;
    4'd11:rx_src_mac[7:0]   <= rx_data_dly2;

    4'd12:rx_eth_type[15:8] <= rx_data_dly2;
    4'd13:rx_eth_type[7:0]  <= rx_data_dly2;
    default: ;
  endcase
end
else
begin
  rx_dst_mac  <= rx_dst_mac;
  rx_src_mac  <= rx_src_mac;
  rx_eth_type <= rx_eth_type;
end
```

4. RX_IP_HEADER

接收以太网 IP 头部数据状态 RX_IP_HEADER，首先得对接收的以太网 IP 头部数据进行计数，定义一个计数器 cnt_ip_header，当处于该状态的时候进行计数，否则清零，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_ip_header <= 5'd0;
else if(curr_state == RX_IP_HEADER)
    cnt_ip_header <= cnt_ip_header + 1'b1;
else
    cnt_ip_header <= 5'd0;
```

然后当处于 RX_IP_HEADER 状态时，获取以太网 IP 头部数据，根据 cnt_ip_header 的值，一共需要获取 20 个字节的数据，分别为 IP 版本（rx_ip_ver）、首部长度（rx_ip_hdr_len）、服务类型（rx_ip_tos）、数据报总长度（rx_total_len）、标识主机发送的每一份数据报（rx_ip_id）、标志位（rx_ip_rsv、rx_ip_df、rx_ip_mf）、段偏移量（rx_ip_frag_offset）、生存期（rx_ip_ttl）、IP 的协议封装类型（rx_ip_protocol）、头部校验和（rx_ip_check_sum）、源 IP 地址（rx_src_ip）和目的 IP 地址（rx_dst_ip），代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  {rx_ip_ver,rx_ip_hdr_len}     <= 8'h0;
  rx_ip_tos                     <= 8'h0;
  rx_total_len                  <= 16'h0;
  rx_ip_id                      <= 16'h0;
  {rx_ip_rsv,rx_ip_df,rx_ip_mf} <= 3'h0;
  rx_ip_frag_offset             <= 13'h0;
  rx_ip_ttl                     <= 8'h0;
  rx_ip_protocol                <= 8'h0;
  rx_ip_check_sum               <= 16'h0;
  rx_src_ip                     <= 32'h0;
  rx_dst_ip                     <= 32'h0;
end
else if(curr_state == RX_IP_HEADER)
begin
  case(cnt_ip_header)
    5'd0:{rx_ip_ver,rx_ip_hdr_len}          <= rx_data_dly2;
    5'd1:rx_ip_tos                          <= rx_data_dly2;
    5'd2:rx_total_len[15:8]                 <= rx_data_dly2;
    5'd3:rx_total_len[7:0]                  <= rx_data_dly2;
    5'd4:rx_ip_id[15:8]                     <= rx_data_dly2;
    5'd5:rx_ip_id[7:0]                      <= rx_data_dly2;
    5'd6:{rx_ip_rsv,rx_ip_df,rx_ip_mf,rx_ip_frag_offset[12:8]}<=rx_dat
a_dly2;
    5'd7:rx_ip_frag_offset[7:0]             <= rx_data_dly2;
    5'd8:rx_ip_ttl                          <= rx_data_dly2;
    5'd9:rx_ip_protocol                     <= rx_data_dly2;
    5'd10:rx_ip_check_sum[15:8]             <= rx_data_dly2;
    5'd11:rx_ip_check_sum[7:0]              <= rx_data_dly2;
    5'd12:rx_src_ip[31:24]                  <= rx_data_dly2;
    5'd13:rx_src_ip[23:16]                  <= rx_data_dly2;
    5'd14:rx_src_ip[15:8]                   <= rx_data_dly2;
    5'd15:rx_src_ip[7:0]                    <= rx_data_dly2;
    5'd16:rx_dst_ip[31:24]                  <= rx_data_dly2;
    5'd17:rx_dst_ip[23:16]                  <= rx_data_dly2;
    5'd18:rx_dst_ip[15:8]                   <= rx_data_dly2;
    5'd19:rx_dst_ip[7:0]                    <= rx_data_dly2;
    default: ;
  endcase
end
```

在 RX_ETH_HEADER 状态时，我们获取了以太网头部数据，进入本状态

RX_IP_HEADER 之后，需要判断之前获取的以太网头部数据是否正确，当获取的数据类型等于 ETH_type（0x0800），得到的 MAC 地址等于代码中设定的值 local_mac_reg 或者等于广播地址 FF_FF_FF_FF_FF_FF 时，则代表以太网头部数据接收成功将 eth_header_check_ok 信号拉高，否则为低，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  eth_header_check_ok <= 1'b0;
else if(rx_eth_type == ETH_type && (rx_dst_mac == local_mac_reg ||
rx_dst_mac == 48'hFF_FF_FF_FF_FF_FF))
  eth_header_check_ok <= 1'b1;
else
  eth_header_check_ok <= 1'b0;
```

在 RX_IP_HEADER 状态时，当 cnt_ip_header 计数等于 2（这里主要是为了确保 eth_header_chexk_ok 信号已经更新，由于以太网接收数据再处理前打了 2 拍，所以判断点也要往后延 2 拍，也就是要到计数为 2 去判断，判断点大于等于 2 都可以），并且 eth_header_check_ok 为低时，则代表数据接收错误，进入 IDLE 状态，当 cnt_ip_header 计数到 19，也就是接收完 20 个 IP 头部书数据之后，进入 RX_UDP_HEADER 状态，代码如下所示：

```
RX_IP_HEADER:
  if(cnt_ip_header == 5'd2 && eth_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_ip_header == 5'd19)
    next_state = RX_UDP_HEADER;
  else
    next_state = RX_IP_HEADER;
```

5. RX_UDP_HEADER

接收 UDP 头部数据状态 RX_UDP_HEADER，进入该状态之后，首先设置一个计数信号 cnt_udp_header 用来计数得到的 UDP 头部数据的个数，代码如下所示：

```
//cnt_udp_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_udp_header <= 4'd0;
else if(curr_state == RX_UDP_HEADER)
  cnt_udp_header <= cnt_udp_header + 1'b1;
else
  cnt_udp_header <= 4'd0;
```

在 RX_UDP_HEADER 状态时，需要判断前一个状态获取的以太网 IP 头部

数据是否正确，当获取的数据正确时，将 ip_header_check_ok 信号拉高，否则为低，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  ip_header_check_ok <= 1'b0;
else if({IP_ver,IP_hdr_len,IP_protocol,cal_check_sum,local_ip_reg} ==
        {rx_ip_ver,rx_ip_hdr_len,rx_ip_protocol,rx_ip_check_sum,rx_dst
_ip})
  ip_header_check_ok <= 1'b1;
else
  ip_header_check_ok <= 1'b0;
```

当 cnt_udp_header 计数到 2 并且 ip_header_check_ok 信号为 0 时，则代表接收数据出错，返回 IDLE 状态；当 cnt_udp_header 计数到 7 并且 udp_header_check_ok 信号为 0 时，也代表接收的数据是错误的，返回 IDLE 状态；当 cnt_udp_header 计数到 7，则代表 UDP 头部数据接收完成，进入 RX_DATA 状态，一共接收了 8 个字节数据分别是源端口号（rx_src_port）、目的端口号（rx_dst_port）、接收的 16 位包括首部在内的 UDP 报文段长度（rx_udp_length）、16 位的 UDP 报头校验和。综上所述，得到 RX_UDP_HEADER 状态机的代码，如下所示：

```
RX_UDP_HEADER:
  if(cnt_udp_header == 4'd2 && ip_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_udp_header == 4'd7 && udp_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_udp_header == 4'd7)
    next_state = RX_DATA;
  else
    next_state = RX_UDP_HEADER;
```

6. RX_DATA

接收数据状态 RX_DATA，进入该状态之后，对接收的数据个数进行计数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_data <= 16'd0;
else if(curr_state == RX_DATA)
  cnt_data <= cnt_data + 1'b1;
else
  cnt_data <= 16'd0;
```

以太网 UDP 的帧数据段个数=以太网接收的帧长度（rx_total_len）-20 个 IP 头部数据-8 个 UDP 头部数据，当处于 RX_IP_HEADER 状态时并且接收完 20 个以太网头部 IP 数据时，计算出以太网 UDP 的数据段个数 rx_data_length，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  rx_data_length <= 16'd0;
else if(curr_state == RX_IP_HEADER && cnt_ip_header == 5'd19)
  rx_data_length <= rx_total_len - 8'd20 - 8'd8;
else
  rx_data_length <= rx_data_length;
```

处于 RX_DATA 状态时，当接收的帧长度 rx_total_len 小于 46，也就是接收的数据段个数小于 18 时，说明此时接收的个数较少，进入到 RX_DRP_DATA 状态，当接收的数据个数等于以太网 UDP 的帧数据段个数时，进入到 RX_CRC 状态，否则处于 RX_DATA 状态，代码如下所示：

```
RX_DATA:
  if((rx_data_length < 5'd18) && (cnt_data == rx_data_length - 1'b1))
    next_state = RX_DRP_DATA;
  else if(cnt_data == rx_data_length - 1'b1)
    next_state = RX_CRC;
  else
    next_state = RX_DATA;
```

7. RX_DRP_DATA

接收填充数据状态 RX_DRP_DATA，首先对接收到的数据个数进行计数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_drp_data <= 5'd0;
else if(curr_state == RX_DRP_DATA)
  cnt_drp_data <= cnt_drp_data + 1'b1;
else
  cnt_drp_data <= 5'd0;
```

当接收到的数据长度 rx_data_length 小于 18 的时候，此时表明我们主机发送的数据长度小于 18，此时为了满足以太网的最小帧长度要求，会在需要发送的数据后面补 0，此时补 0 的数据个数就应该等于 18 减去接收到的数据长度，当 cnt_drp_data 等于补 0 的数据个数时（18 - rx_data_length -1=17- rx_data_length），进入到 RX_CRC 状态，代码如下所示：

```
RX_DRP_DATA:
  if(cnt_drp_data == 5'd17 - rx_data_length)
    next_state = RX_CRC;
  else
    next_state = RX_DRP_DATA;
```

8. RX_CRC

接收FCS校验数据状态，如果接收的数据为0，则进入PKT_CHECK状态，否则一直处于 RX_CRC 状态，代码如下所示：

```
RX_CRC:
  if(rx_datav_dly2 == 1'b0)
    next_state = PKT_CHECK;
  else
    next_state = RX_CRC;
```

9. PKT_CHECK

进入 PKT_CHECK 状态之后，直接返回 IDLE 状态，重新进入一轮数据的接收，代码如下所示：

```
PKT_CHECK:
  next_state = IDLE;
```

通过以上对状态机的描述，以太网接收模块的基本功能就已经完成了，接下来对该模块比较常用的信号进行说明。

首先是 payload_valid_o 信号和 payload_dat_o 信号，当状态处于接收数据状态 RX_DATA 时，将输出数据有效信号 payload_valid_o 拉高并且将此时接收到的数据输出给 payload_dat_o 信号，代码如下所示：

```
//payload output
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  payload_valid_o <= 1'b0;
  payload_dat_o   <= 8'h0;
end
else if(curr_state == RX_DATA)
begin
  payload_valid_o <= 1'b1;
  payload_dat_o   <= rx_data_dly2;
end
else
begin
  payload_valid_o <= 1'b0;
```

```
  payload_dat_o   <= 8'h0;
end
```

然后就是 one_pkt_done 信号和 pkt_error 信号，当状态处于 PKT_CHECK 时，将传输完成标志信号 one_pkt_done 拉高，当 CRC 校验的结果 crc_check 等于 32'h2144DF1C 并且 reg_data_overflow 信号为低电平的时候，说明接收数据没错，让 pkt_error 信号为低电平，否则接收数据出错，将 pkt_error 信号拉高，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  one_pkt_done <= 1'b0;
  pkt_error    <= 1'b0;
end
else if(curr_state == PKT_CHECK)
begin
  one_pkt_done <= 1'b1;
  if(crc_check == 32'h2144DF1C && reg_data_overflow == 1'b0)
    pkt_error  <= 1'b0;
  else
    pkt_error  <= 1'b1;
end
else
begin
  one_pkt_done <= 1'b0;
  pkt_error    <= 1'b0;
end
```

上述代码中的 reg_data_overflow 信号，代表输入数据溢出，也就是当状态处于接收数据状态时并且输入数据溢出信号 data_overflow_i 为高电平时，将 reg_data_overflow 信号拉高，代表数据溢出，否则为保持不变，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  reg_data_overflow <= 1'b0;
else if(curr_state == RX_DATA && data_overflow_i == 1'b1)
  reg_data_overflow <= 1'b1;
else
  reg_data_overflow <= reg_data_overflow;
```

综上所述，以太网接收模块的代码就基本设计完成。再结合前面章节中的 RGMII 与 GMII 转换电路设计，我们便能实现 RGMII 接口的千兆以太网接收。关于该模块的完整代码请自行查看工程中的源代码文件。

53.2串口发送控制模块

为了验证以太网接收模块功能是否正常，我们计划使用串口打印出以太网接收到的数据，通过对比打印出的数据判断模块能否正常工作。

考虑到以太网发送模块的工作时钟为 125M，串口的工作时钟为 50M，两者时钟速率不匹配，设计首先需要添加一个 FIFO 来解决跨时钟域的问题。随后还需要一个串口发送控制模块，用于控制 FIFO 的读出以及串口发送模块的工作。

首先，添加一个双时钟 FIFO IP，设置输入输出位宽为 8 位，深度为 1024，配置界面如下图 53-3 所示。

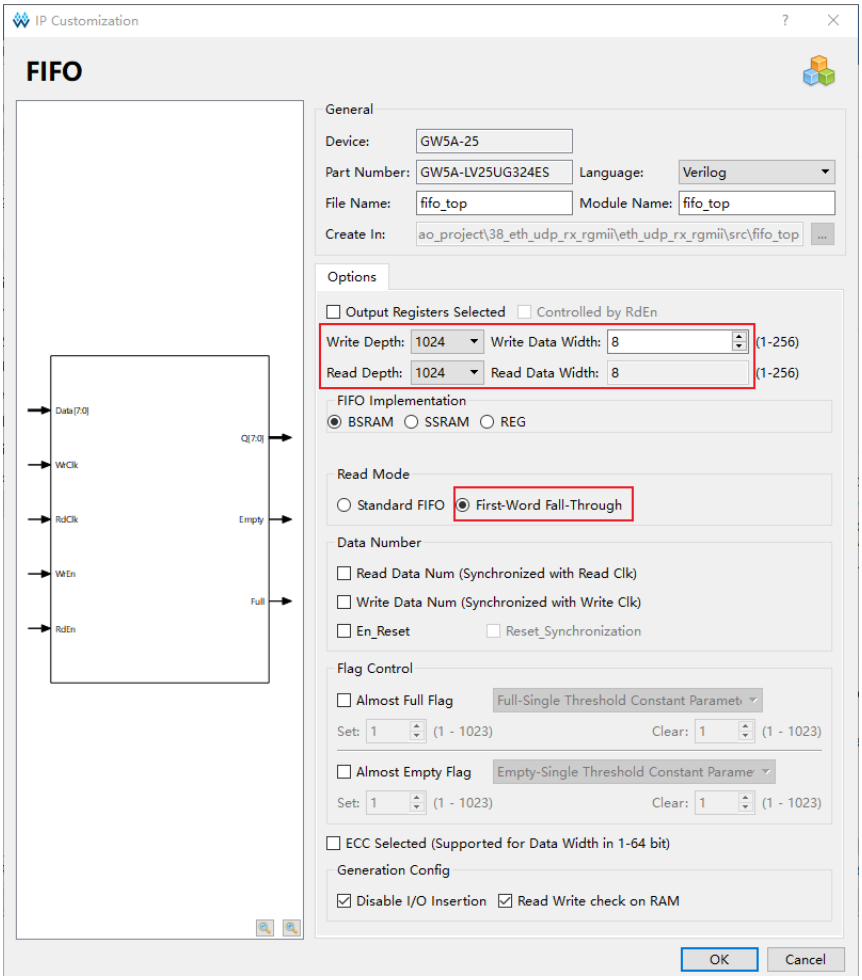


图 53-3 FIFO 配置界面

FIFO IP 的输入时钟为 125M，输入的数据由以太网接收模块输出的数据信号 rx_payload_dat，写使能为以太网接收的输出数据有效信号 rx_payload_valid，输出时钟为串口发送模块的 50M 时钟，读使能由串口发送控制模块产生，FIFO IP 的例化代码如下所示：

```
fifo_top fifo_top(
    .Data(rx_payload_dat), //input [7:0] Data
    .WrClk(clk125m), //input WrClk
    .RdClk(Clk), //input RdClk
    .WrEn(rx_payload_valid), //input WrEn
    .RdEn(fifo_rd_req), //input RdEn
    .Q(dout), //output [7:0] Q
    .Empty(rx_empty), //output Empty
    .Full() //output Full
);
```

然后就是设计一个串口发送控制模块，该模块我们通过一个状态机实现，当处于状态 0 时，当 fifo_empty 为低电平，也就是 FIFO 中被写入数据之后，产生 FIFO 的读请求信号 fifo_rd_req，然后进入到状态 1；当处于状态 1 时，将 fifo_rd_req 信号拉低，并将串口发送使能信号 uart_send_en 拉高，同时把 FIFO 读出来的数据 fifo_rd_data 交由串口发送，然后进入状态 2；处于状态 2 时，将 uart_send_en 拉低，当得到串口发送完成信号 uart_tx_done 时，进入状态 0 重新开始读取数据然后发送，串口发送控制模块中状态机代码如下所示：

```
always@(posedge clk or posedge reset_p)
if(reset_p) begin
    state <= 1'd0;
    uart_tx_data <= 8'd0;
    fifo_rd_req <= 1'd0;
    uart_send_en <= 1'd0;
end
else begin
    case(state)
        0:
        begin
            if(!fifo_empty) begin       //如果 FIFO 不为空，可以开始发送
                fifo_rd_req <= 1'd1;
                state <= 2'd1;
            end
            else begin
                fifo_rd_req <= 1'd0;
                state <= 2'd0;
            end
        end

        1:
        begin
            fifo_rd_req <= 1'd0;
```

```
            uart_send_en <= 1'd1;
            uart_tx_data <= fifo_rd_data;
            state <= 2'd2;
        end

        2:
        begin
            uart_send_en <= 1'd0;
            if(uart_tx_done)
                state <= 2'd0;
            else
                state <= 2'd2;
        end

        default:;
    endcase
end
```

至此，串口发送控制模块就设计完成了，在本次实验中，还需要添加一个串口发送模块 uart_byte_tx，这个模块在前面的实验中已经介绍过了，本次实验就不再做讲解，只需要将该模块的源文件添加至本工程中，然后在顶层模块中完成例化即可。

53.3系统板级验证

通过上述步骤，代码部分的设计就完成了，随后分配管脚并约束电平。本次设计的引脚分配表如下表 53-2 所示：

表 53-2 引脚分配表

Signial Name	Pin NO.		Signial Name	Pin NO.
rgmii_rxd[3]	F5		rgmii_rxdv	E1
rgmii_rxd[2]	F6		eth_reset_n	G6
rgmii_rxd[1]	G1		uart_tx	V8
rgmii_rxd[0]	G3		reset_n	C15
rgmii_rx_clk	L5		Clk	T9

完成以上步骤后，生成 bit 并连接硬件准备板级验证。本次设计硬件连接如下图 53-4 所示：

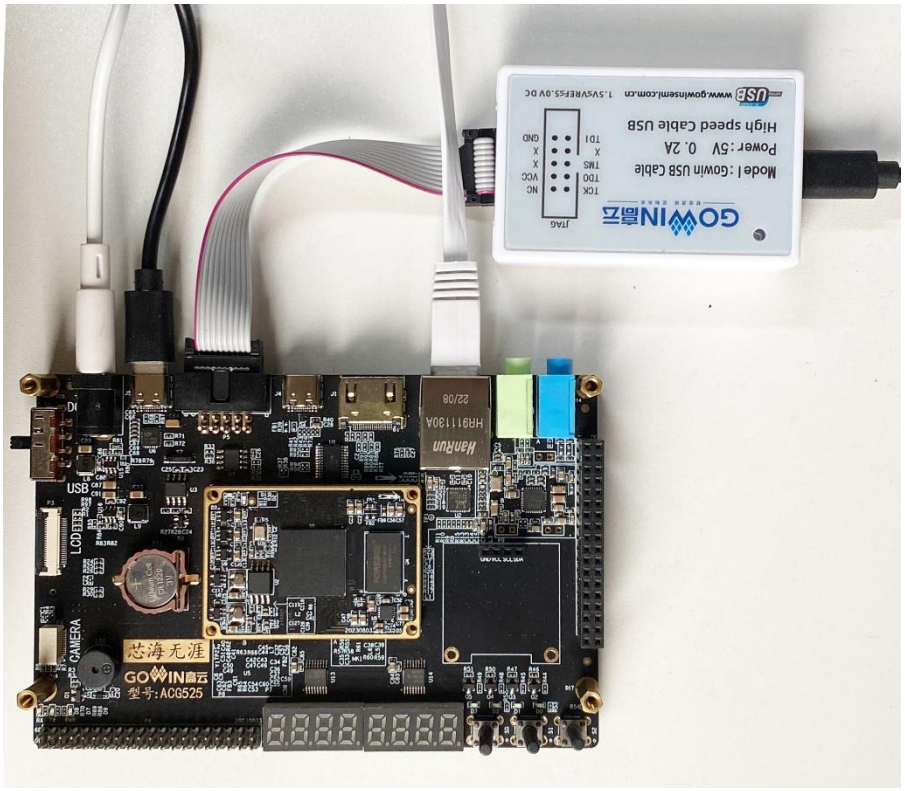


图 53-4 硬件连接图

网线一端插入开发板的网口，另一端插入到电脑网口。串口线的一端接在开发板的串口上，另一端插入电脑的 USB 口。连接完成之后，给开发板上电，并下载 bit 文件。

然后同时打开串口调试助手和网络调试助手，分别配置如下图 53-5、图 53-6 所示。

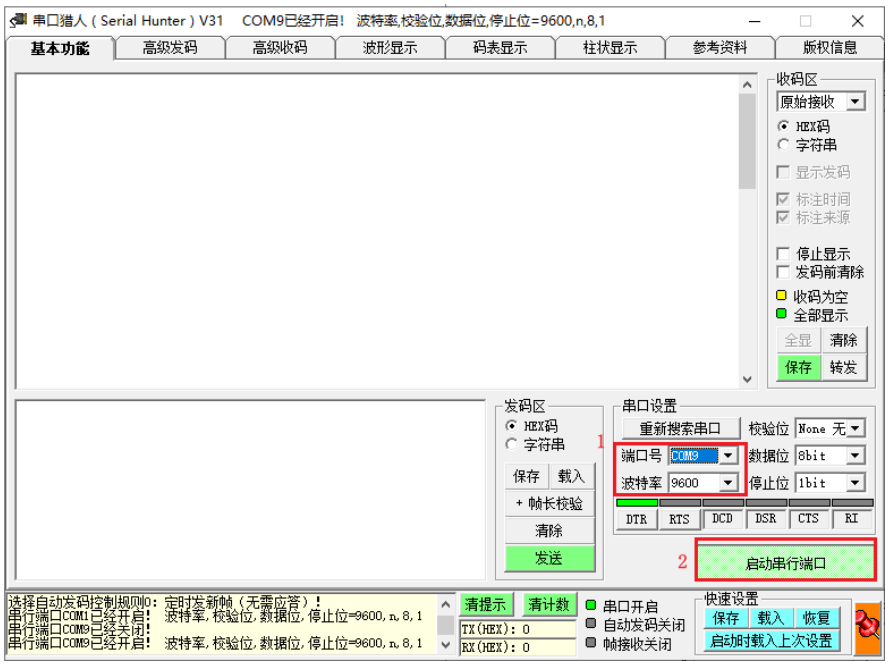


图 53-5 串口助手配置界面

图 53-6 网口配置界面

点击网络调试助手中的发送之后，可以看到网络调试助手界面和串口调试助手界面如下图 53-7、图 53-8 所示。

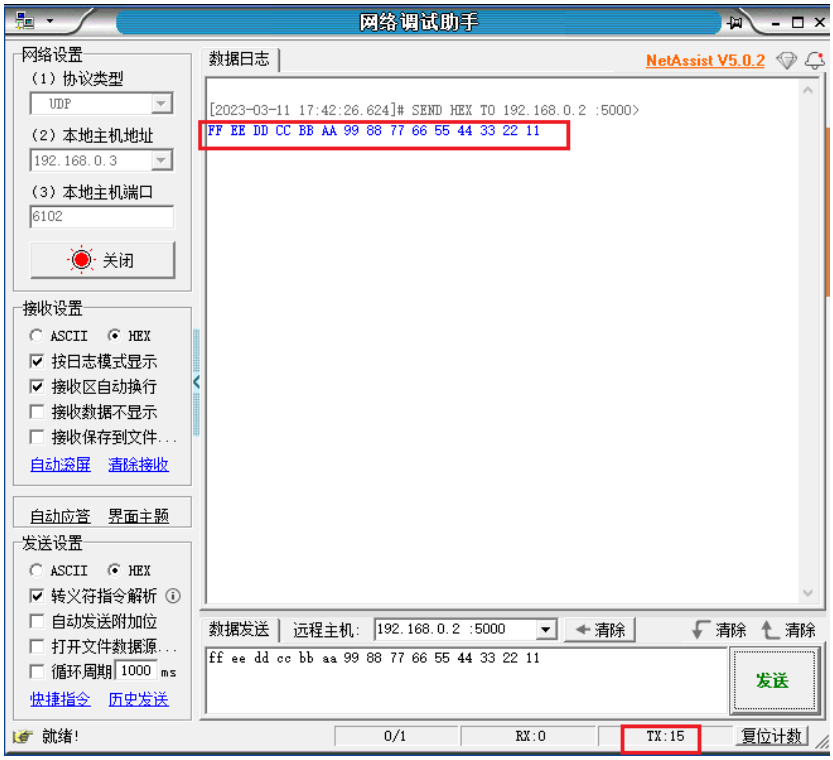


图 53-7 网络调试助手界面

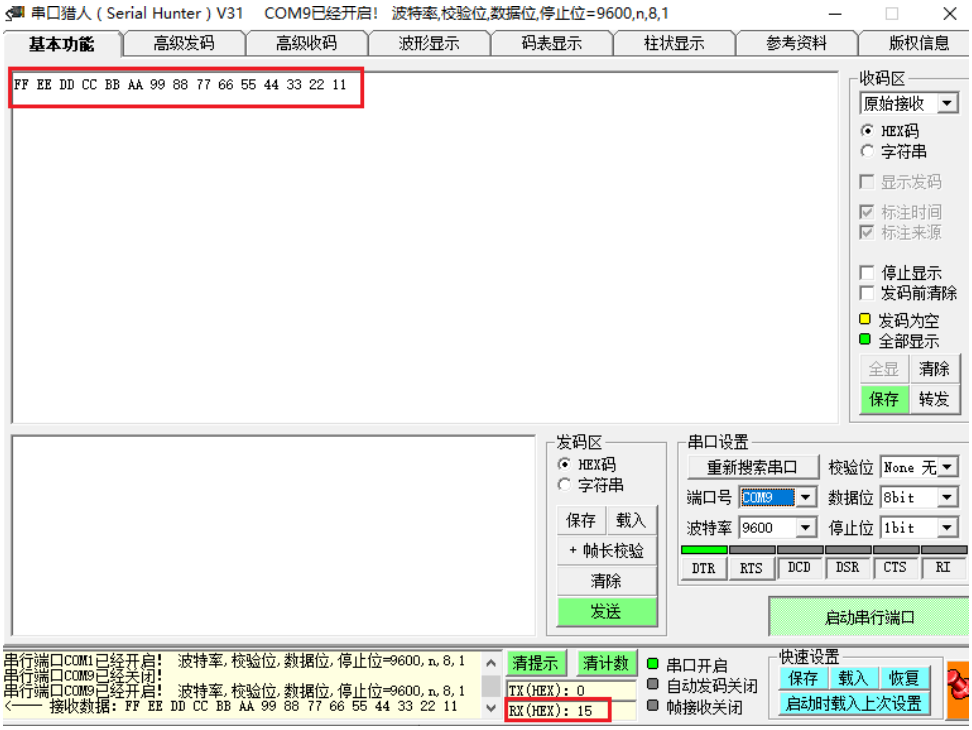


图 53-8 串口调试助手界面

从上述图中可以看出串口接收的数据和网口发送的数据一致，接收的数据个数也一致，这说明以太网接收模块的功能正常。

54 MDIO 控制器设计

工程源码	|----02_设计实例 |----ch54_phy_config_mdio
相关视频课程	
说明	

章节导读

PHY 芯片根据不同的需求，会有不同的工作模式，例如需要设置其以太网通信速率为千兆、百兆、十兆或自动协商，而处理器也需要知道其连接状态，例如需要知道网络是否已经连接通，以及当前的链接速率是多少。为了实现这些功能，IEEE 在以太网标准 IEEE 802.3 中使用若干条款定义了用处理器来管理 PHY 芯片的标准控制接口——MDIO。

本章，我们将学习 MDIO 接口相关知识，并设计相应的驱动模块，通过 MDIO 接口带领大家学习如何配置 PHY 芯片。

54.1PHY 管理接口 MDIO 介绍

54.1.1MDIO 接口介绍

MDIO 是一种简单的双线串行接口，将管理器件(如 MAC 控制器、微处理器)与具备管理功能的 PHY 芯片相连接，从而控制收发器并从收发器收集状态信息。可收集的信息包括链接状态、传输速度与选择、断电、低功率休眠状态、TX/RX 模式选择、自动协商控制、环回模式控制等。除了拥有 IEEE 要求的功能之外，收发器厂商还可添加更多的信息收集功能。

MDIO 总线包含两个信号：MDC 和 MDIO。MDC 是管理数据的时钟信号，由管理器（如处理器或 MAC）输出给 PHY 芯片，最高速率可超过 5MHz。MDIO 则是管理数据的输入输出双向接口，其中的数据与 MDC 时钟同步。

和大家熟悉的 I2C 接口相似，MDIO 总线也支持多设备模式，即一个 MDIO 总线上最多可连接 32 个 PHY 芯片的 MDIO 接口，通过在传输过程中的 PHY 器件地址字段传输不同的器件地址值来指定具体操作哪个 PHY 芯片。

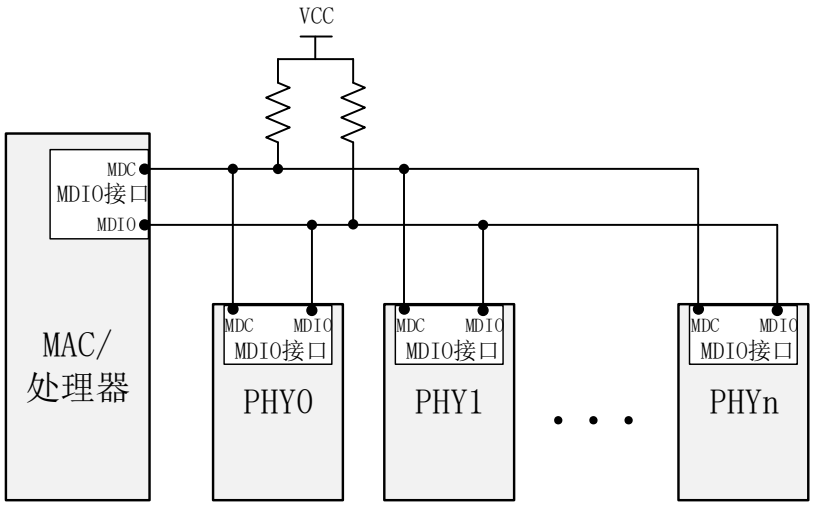


图 54-1 MDIO 与 PHY 芯片连接结构

每个 PHY 芯片都有一个硬件地址，该地址为 5 位，由固定部分和可变部分组成，可变部分可以在芯片应用电路中通过对地址脚设置上下拉到电源或 GND。若多个 PHY 芯片同时连接在同一个 MDIO 总线上，需要设置不同的 PHY 器件地址以示区分。

每个 PHY 芯片中都有最多 32 个寄存器可供读写（32 个寄存器，需要 5 位寄存器地址），包括由 IEEE 802.3 规范定义的前 16 个标准功能寄存器和后续最多 16 个由 PHY 芯片厂家自定义的功能寄存器。所有 PHY 芯片的前 16 个寄存器功能定义都必须相同，后续 0~16 个寄存器的功能由 PHY 芯片厂家根据实际需求自定义。

开发板上使用的 PHY 芯片型号为 RTL8211F-CG，通过查询手册可以知道其器件地址由固定的 00+由引脚控制的 PHYAD[2:0]组成，如图 54-2 所示。

7.8. Hardware Configuration

The I/O pad voltage, interface mode, and PHY address can be set by the CONFIG pins. The respective value mapping of CONFIG with the configurable vector is listed in Table 10. To set the CONFIG pins, an external pull-high or pull-low via resistor is required.

Table 10. CONFIG Pins vs. Configuration Register

CONFIG Pin	Configuration
RXD3	PHYAD[0]
RXC	PHYAD[1]
RXCTL	PHYAD[2]
RXD2	PLLOFF
RXD1	TXDLY
RXD0	RXDLY
LED0	CFG_EXT
LED1	CFG_LDO[0]
LED2	CFG_LDO[1]

Table 11. Configuration Register Definitions

Configuration	Description
PHYAD[2:0]	PHY Address. PHYAD sets the PHY address for the device. The RTL8211F(I)/ RTL8211FD(I) supports PHY addresses from 00001 to 00111. *Note 1: An MDIO command with PHY address=0 is a broadcast from the MAC; each PHY device should respond. This function can be disabled by setting Reg24.13=0 (See Table 37).* *Note 2: The RTL8211F(I)/ RTL8211FD(I) with PHYAD[2:0]=000 can automatically remember the first non-zero PHY address. This function can be enabled by setting Reg24.6 = 1 (See Table 37).*

图 54-2 PHY 地址

而在开发板上，这三个引脚分别被上拉和接地，如图 54-3 所示。

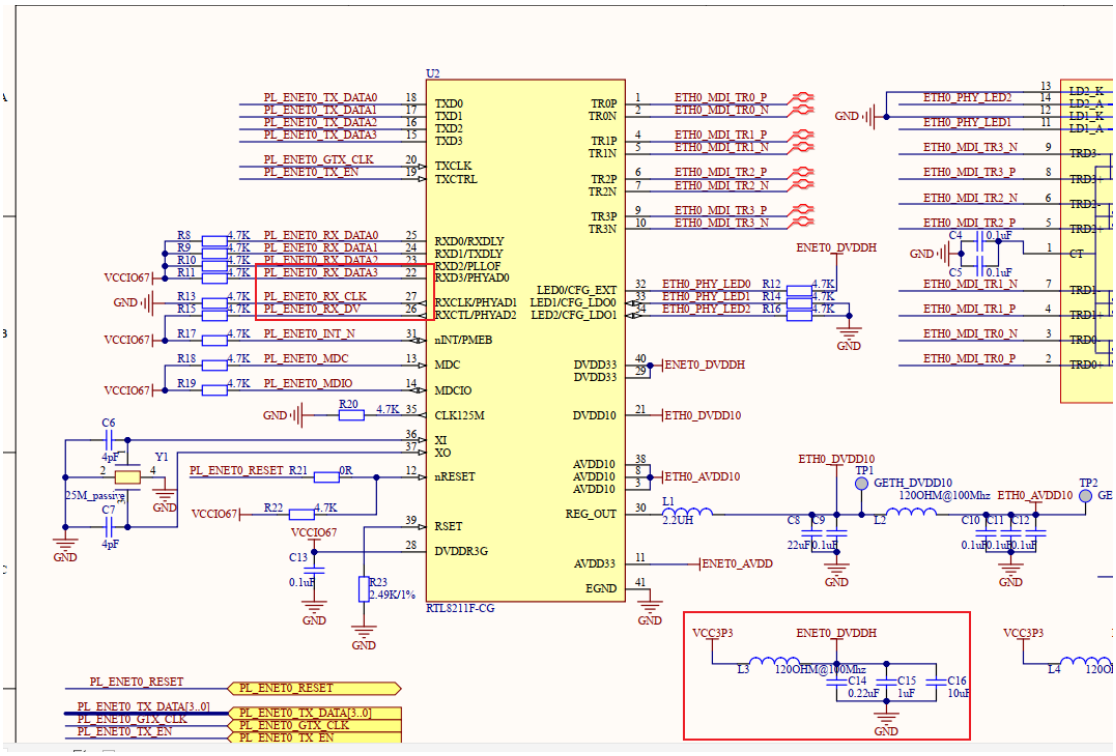


图 54-3 开发板上 PHY 芯片引脚电路

据此，我们便可以得出开发板上 PHY 芯片的地址为 0x05(5’b00101)。而有了地址后，后续对 MDIO 的时序介绍也就方便了不少。

54.1.2MDIO 时序介绍

介绍了 MDIO 接口之后，要想正确的使用 MDIO 接口来管理 PHY 芯片，还需要了解 MDIO 接口的时序流程。MDIO 的工作流程为：

1. 空闲：MDIO 接口在没有传输数据的空闲状态（IDLE）数据线 MDIO 处于高阻态。
2. 开始：MAC 或处理器驱动 MDIO 产生一个 2bit 的开始标识码(01)，标志着一次读、写传输的开始。
3. 读写标记：MAC 或处理器驱动 MDIO 产生一个 2bit 数据来标识本次操作是读操作(10)还是写操作(01)。
4. PHY 器件地址：MAC 或处理器驱动 MDIO 产生 5 位的 PHY 器件地址数据，用来指明此次操作是针对的具体那一个 PHY 芯片。

5. PHY 寄存器地址：MAC 或处理器驱动 MDIO 产生 5 位的 PHY 寄存器地址数据，用来指明此次操作是针对的 PHY 中具体哪一个寄存器。

6. MDIO 控制权切换：在读操作中，由于在 PHY 移出寄存器中的数据时，MDIO 信号将由 PHY 芯片接管驱动，既 MDIO 信号存在一个驱动切换的情形，为了避免切换过程中 MAC 或处理器驱动和 PHY 芯片同时驱动 MDIO 信号以出现冲突干扰，在传输完 PHY 寄存器地址后，需要传输 2bit 的 TA（Turnaround）信号，传输该信号时，MAC 或处理器 MDIO 信号为高阻状态，即不驱动 MDIO 信号至低电平，对于读操作，PHY 芯片将在这 2bit 信号的第 2 个 bit 时将 MDIO 信号拉低以标记读出数据传输。同时，这第 2 个 bit 还将作为类似于 I2C 总线中的应答位，如果在读操作中，该 bit 没有被拉低，则表明读 PHY 芯片操作失败。对于写操作，第一个 bit 时 MAC 或处理器和 PHY 芯片都保持为高阻状态，MDIO 被上拉电阻拉为高电平，在第 2 个 bit 时由 MAC 或处理器将该位拉低。

7. 读写数据：MDIO 串行读出/写入 16bit 的寄存器数据。

8. IDLE 状态： MDIO 再次进入高阻状态，对于部分 PHY 芯片，MDC 将继续产生至少 7 个时钟周期之后方可停止。

注意，在空闲和开始状态之间，MDIO 需要保持高阻态至少 32 个 MDC 时钟周期，该段称为前导信号（preamble）。

下表 54-1 为一次读写操作的数据帧格式：

表 54-1 MDIO 读写操作数据帧格式

	Preamble（32bit）	ST (2bit)	OP (2bit)	PHYAD (5bit)	REGAD (5bit)	TA (2bit)	DATA (16bit)	IDLE
Read	1...1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD	Z
Write	1...1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD	Z

其中各段的意义和功能如下表 54-2 所示。

表 54-2 各段功能意义

名称	功能和意义描述
Preamble	mac 端在 mdc 时钟下，通过 mdio，发送 32 个连续的逻辑 1
ST	帧起始，用 01 来表示
OP	操作符，用 01 来表示写操作，用 10 来表示读操作
PHYAD	物理地址，最多可以将 8 个物理连接到一个 MAC，高 2bit 为 0
REGAD	寄存器地址，5 位宽度，用于设置当前操作 PHY 的 32 个寄存器中的哪一个
TA	这两位是用来指示方向的，Z 是高阻态
DATA	总线上传输的 16 位数据

IDLE	无传输时的状态，该状态时候 PHY 的 MDIO 设置为高阻态，其电平将由 MDIO 引脚上的上拉电阻上拉到高电平

MDIO 会在 MDC 的下降沿变化，变化后的数据会在 MDC 上升沿被采样，进而组成上述所说的数据帧。图 54-4 和图 54-5 便是 MDIO 接口进行读/写时的时序关系：

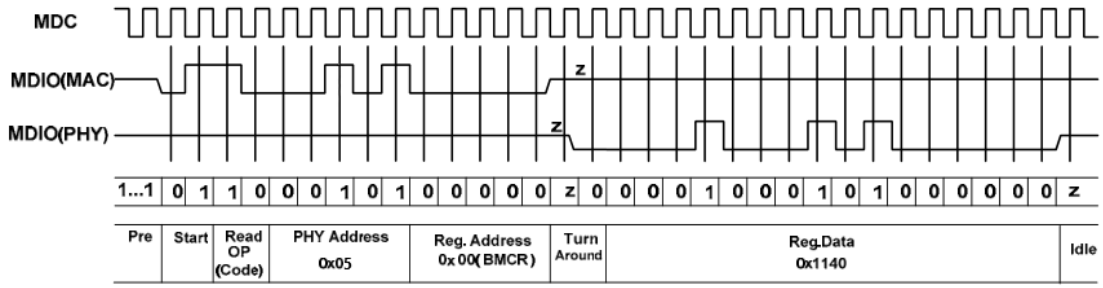


图 54-4 MDC/MDIO 读时序

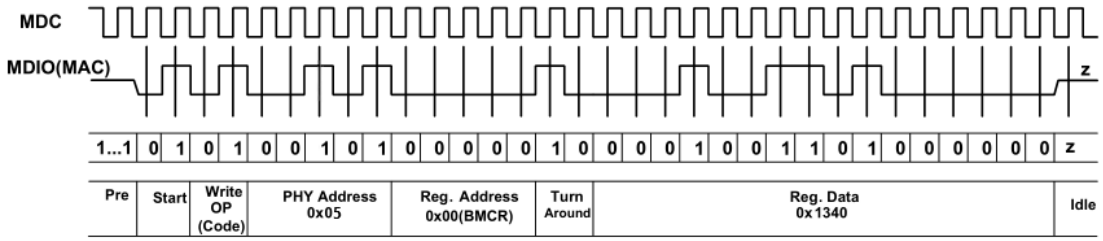


图 54-5 MDC/MDIO 写时序

基于该时序图，我们便能很方便的写出对应的驱动模块，对 PHY 芯片进行读写操作。至于具体在什么模式下需要对什么寄存器进行操作，则需要读者查询 PHY 芯片手册中对应寄存器的说明。

54.1.3MDIO 接口注意事项

部分 PHY 芯片（如 RTL8211）要求在一次读写操作完成，转为 IDLE 状态之后，还需要 MDC 信号持续至少 7 个时钟周期以确保 PHY 芯片正常完成该操作。

PHY 芯片的 MDIO 引脚需要连接至少 1.5K Ohm 的上拉电阻。

在 PHY 芯片成功 Link 之前，不得向 PHY 芯片提供发送使能信号，即 TX_EN 信号在 PHY 芯片没有成功 Link 之前不得出现高电平，否则可能会影响 PHY 芯片的自动协商和工作模式设置。

PHY 芯片一般都有一个硬件复位引脚 reset，一般 PHY 芯片都会做出说明，在该复位信号释放后需要等待至少多久的时间（例如，对于 RTL8211，这个值为 30ms）才能开始使用 MDIO 接口对 PHY 芯片进行读写操作。

54.2mdio_bit_shift 模块设计

根据协议帧的格式和上面的读写时序图，我们发现这个协议帧里面的变化内容有 OP、PYHAD、REGAD、以及传输的数据（DATA）等字段，在这里我们可以把这些作为端口，同时加入传输开始 start 和传输完成 done 等握手信号来指示当前的模块工作状态，因此设计出图 54-6 所示模块单元。

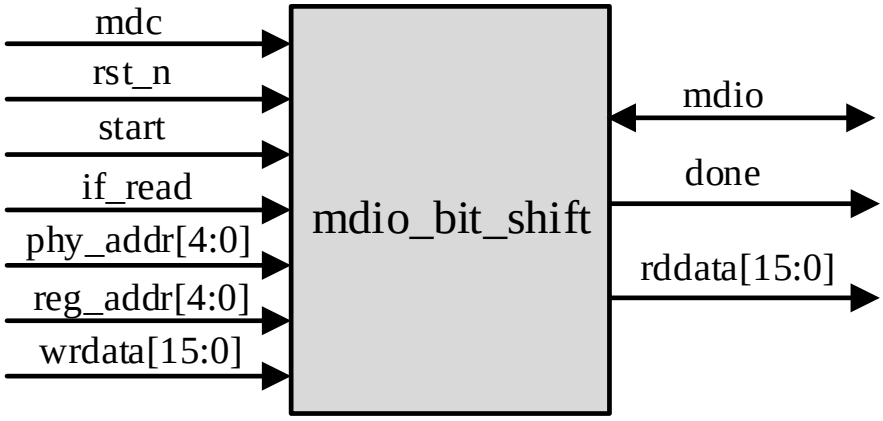


图 54-6 mdio_bit_shift 模块框图

其中，该模块的各个接口信号如表 54-3 所示。

表 54-3 mdio_bit_shift 接口功能描述

接口名称	I/O	功能描述
mdc	I	时钟接口
rst_n	I	模块复位，低电平有效
start	I	开始传输标志
if_read	I	读写方向控制 1:读，0:写
phy_addr	I	5 位的 phy 地址输入信号，最高 2 位为 0
reg_addr	I	5 位的 reg 地址输入信号
wrdata	I	要写入 phy 寄存器的 16 位数据
mdio	I/O	数据接口
done	O	操作完成标志
rddata	O	从 phy 寄存器读出的 16 位数据

模块使用状态机来描述 MDIO 接口的读写时序，对于时序图中的每个字段都采样独热码定义了一个状态，如下：

```
localparam IDLE  = 8'h01,
           PRE   = 8'h02,
           ST    = 8'h04,
           OP    = 8'h08,
           PHYAD = 8'h10,
           REGAD = 8'h20,
           TA    = 8'h40,
```

```
            DATA  = 8'h80;
```

在不同状态下，让状态机按照时序图对差异部分进行描述，从而实现对 phy 寄存器的配置和读取，该部分代码如下：

```
case (state)
    IDLE:
        begin
            mdio_o <= 1'b1;
            mdio_oe <= 1'b0;
            done <= 1'b0;
            rddata <= 'd0;
            if (start)
            begin
                cnt <= 'd0;
                state <= PRE;
            end
        end
    PRE://PRE 32'hffff_ffff  32bit
        begin
            mdio_oe <= 1'b1;
            mdio_o <= 1'b1;
            cnt <= cnt + 1'b1;
            if (cnt > 'd30)
            begin
                cnt <= 'd0;
                state <= ST;
                mdio_o <= 1'b0;
            end
        end
    ST: //ST 01  2bit
        begin
            mdio_o <= 1'b1;
            cnt <= cnt + 1'b1;
            if (cnt >= 'd1)
            begin
                cnt <= 'd0;
                state <= OP;
                mdio_o <= if_read;
            end
        end
    OP: //OP 01:write,10:read  2bit
        begin
            mdio_o <= !if_read;
            cnt <= cnt + 1'b1;
```

```
            if (cnt >= 'd1)
            begin
                cnt <= 'd0;
                state <= PHYAD;
                mdio_o <= phy_addr[4];
            end
        end
    PHYAD:  //PHYAD  5bit
        begin
                cnt <= cnt + 1'b1;

            if (cnt >= 'd4)
            begin
                cnt <= 'd0;
                state <= REGAD;
                mdio_o <= reg_addr[4];
            end
            else
                mdio_o <= phy_addr[4 - cnt[2:0]-1];
        end
    REGAD:  //REGAD  5bit
        begin
            cnt <= cnt + 1'b1;
            if (cnt >= 'd4)
            begin
                cnt <= 'd0;
                state <= TA;
                mdio_o <= !if_read;
                mdio_oe <= !if_read;
            end
            else
                    mdio_o <= reg_addr[4 - cnt[2:0]-1];
        end
    TA:     //TA     2bit
        begin
            mdio_o <= 1'b0;
            cnt <= cnt + 1'b1;
            if (cnt >= 'd1)
            begin
                cnt <= 'd0;
                state <= DATA;
                mdio_o <= wrdata[15];
            end
```

```
        end
    DATA:   //DATA   16bit
        begin
            cnt <= cnt + 1'b1;
            if(cnt < 'd15)begin
                if (if_read)
                    rddata <= {rddata[14:0], mdio};
                else
                    mdio_o <= wrdata[15 - cnt[3:0]-1];
            end
            else begin
                cnt <= 'd0;
                state <= IDLE;
                mdio_o <= 1'b1;
                mdio_oe <= 1'b0;
                done <= 1'b1;
            end
        end
endcase
```

这里我们的状态机并没有完全按照时序图中的顺序来写，这是因为一段式状态机在工作时，输出总是会比状态慢一拍，这是一段式状态机本身的特性。虽然输出滞后一拍并不会影响功能的实现（输出数据是正确的），但是为了使后续仿真波形数据更符合时序图，这里我们在状态机每个状态结束时，提前产生下一拍的数据，以此实现输出数据与状态的同步。

54.3mdio_bit_shift 模块仿真验证

上一节我们已经对模块主要代码进行了分析，但是模块能否正常工作并达到我们所想要的效果，仍需要进行仿真验证。为此，我们设计了如下激励文件：

```
`timescale 1ns/1ps
module mdio_bit_shift_tb;
    reg             mdc;        //时钟接口
    wire            mdio;       //数据接口
    reg             rst_n;      //模块复位，低电平有效
    reg             if_read;    //读写方向控制 1:读，0:写
    reg      [4:0]  phy_addr;   //phy 地址
    reg      [4:0]  reg_addr;   //reg 地址
    reg      [15:0] wrdata;     //写入 phy 寄存器的数据
    reg             start;      //开始传输标志
    wire            done;       //操作完成标志
```

```
    wire     [15:0] rddata;       //phy 寄存器读出的数据

    pullup PUP(mdio);  //模拟外部上拉电阻
    mdio_bit_shift DUT(
        .mdc      (mdc      ),
        .mdio     (mdio     ),
        .rst_n    (rst_n    ),
        .if_read  (if_read  ),
        .phy_addr (phy_addr ),
        .reg_addr (reg_addr ),
        .wrdata   (wrdata   ),
        .start    (start    ),
        .done     (done     ),
        .rddata   (rddata   )
    );
    always #10 mdc = ~mdc;
    initial begin
        mdc = 1;
        rst_n = 0;
        if_read = 0;
        phy_addr = 5'b00000;
        reg_addr = 5'b00000;
        wrdata = 16'd0;
        start = 0;
        #201;

        rst_n = 1;
        if_read = 0;
        phy_addr = 5'b00101;
        reg_addr = 5'b00000;
        wrdata = 16'h2100;
        @(posedge mdc);
        start = 1;
        @(posedge done);
        #200;
        $stop;
    end
endmodule
```

在上面代码中，我们是对物理地址 5'b00101 的寄存器 0（BMCR）进行一次写操作，写入的 16 位数据位 16'h2100，仿真波形如下图 54-7 所示：

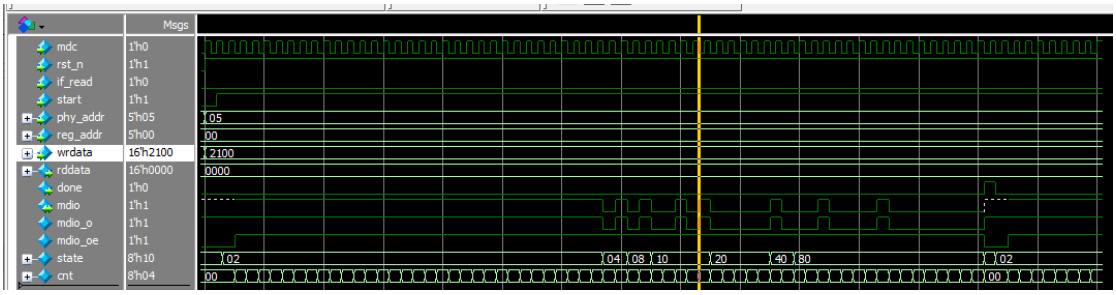


图 54-7 仿真波形

在上述波形中当 state 为 02（也就是 PRE 状态），mdio 发送了连续的 32 个逻辑 1，当 state 为 04（也就是 ST 状态），mdio 先后产生了 01，当 state 为 08（也就是 OP 状态），mdio 依次产生了 01 说明是写操作，当 state 为 10（也就是 PHYAD 状态），mdio 依次产生了 00101 表示操作的 phy 地址，当 state 为 20（也就是 REGAD 状态），mdio 依次产生了 00000 表示操作的 phy 的寄存器 0（BMCR），当 state 为 40（也就是 TA 状态），mdio 依次产生了 10，当 state 为 80（也就是 DATA 状态），mdio 依次产生了 0010_0001_0000_0000(16'h2100)表示写入寄存器 0 的值为 16'h2100，符合我们预期的结果。

54.4phy_reg_config 模块设计

通过上一节的仿真波形，我们可发现，mdio_bit_shift 模块虽然能实现基本的 phy 寄存器的读写操作，但是是一次性的，不能连续配置。为了使设计更贴合实际应用场景，我们需要设计一个上层模块来控制 mdio_bit_shift 模块实现多次读写操作。根据功能我们加入 phy 芯片寄存器配置完成标志信号 phy_init_done，从 phy 寄存器读出的数据 rddata，以及 phy 芯片的复位信号 phy_rst_n，因此设计出如下模块单元：

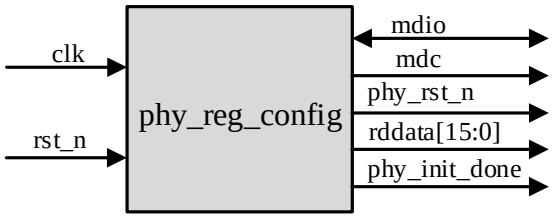


图 54-8 phy_reg_config 模块结构框图

模块对应的接口说明如表 54-4 所示。

表 54-4 phy_reg_config 接口说明

接口名称	I/O	功能描述
clk	I	模块时钟 50MHz

rst_n	I	模块复位，低电平有效
mdio	I/O	管理接口数据总线
mdc	O	管理接口时钟总线
phy_rst_n	O	phy 芯片复位，低电平有效
rddata	O	从 phy 寄存器读出的 16 位数据
phy_init_done	O	phy 寄存器初始化操作完成标志

那么我们该如何使用该模块控制mdio_bit_shift 工作呢？还是通过实际的应用来说明，开发板使用的是 RGMII 模式，是千兆模式。如果此时我们希望它工作在百兆模式可行么，答案是肯定的，我们通过查询数据手册，寄存器 0（BMCR 寄存器）里面有对应的速率控制，前提是要将自动协商给关闭了，当然为了能正确的配置，我们首先要让芯片先重新上电。

这里给出寄存器 0 的位描述，更多寄存器表用户可以查看芯片手册：

Table 24. BMCR (Basic Mode Control Register, Address 0x00)

Bit	Name	Type	Default	Description
0.15	Reset	RW, SC	0	Reset. 1: PHY reset 0: Normal operation Register 0 (BMCR) and register 1 (BMSR) will return to default values after a software reset (set Bit15 to 1). This action may change the internal PHY state and the state of the physical link associated with the PHY.
0.14	Loopback	RW	0	Loopback Mode. 1: Enable PCS loopback mode 0: Disable PCS loopback mode
0.13	Speed[0]	RW	0	Speed Select Bit 0. In forced mode, i.e., when Auto-Negotiation is disabled, bits 6 and 13 determine device speed selection.
0.12	ANE	RW	1	Auto-Negotiation Enable. 1: Enable Auto-Negotiation 0: Disable Auto-Negotiation
0.11	PWD	RW	0	Power Down. 1: Power down (only Management Interface and logic are active; link is down) 0: Normal operation
0.10	Isolate	RW	0	Isolate. 1: RGMII/GMII interface is isolated; the serial management interface (MDC, MDIO) is still active. When this bit is asserted, the RTL8211E-VB(VL)/RTL8211EG-VB ignores TXD[7:0], and TXCTL inputs, and presents a high impedance on TXC, RXC, RXCTL, RXD[7:0]. 0: Normal operation
0.9	Restart_AN	RW, SC	0	Restart Auto-Negotiation. 1: Restart Auto-Negotiation 0: Normal operation
0.8	Duplex	RW	1	Duplex Mode. 1: Full Duplex operation 0: Half Duplex operation This bit is valid only in force mode, i.e., NWay is disabled.
0.7	Collision Test	RW	0	Collision Test. 1: Collision test enabled 0: Normal operation
0.6	Speed[1]	RW	1	Speed Select Bit 1. Refer to bit 0.13.
0.5:0	RSVD	RO	000000	Reserved.

Speed[1]	Speed[0]	Speed Enabled
1	1	Reserved
1	0	Reserved
0	1	100Mbps
0	0	10Mbps

第一步，通过寄存器表我们可知道要让 phy 芯片先掉电，可以通过配置寄存器 0 的 bit11 位来实现，同时这个时候我们也顺便可以配置让芯片工作在双工

模式下, 通过配置寄存器 0 的 bit8 位来实现，通过上表我们可以得出此时这个寄存器应该配置成 0000_1001_0000_0000（16'h0900）。

第二步，通过寄存器表我们可知道要让 phy 芯片工作在百兆模式，那么可以通过配置寄存器 0 的 bit13 和 bit6 来实现，此时 bit13 里面的注释也说了，要先把自动协商禁用了，我们知道寄存器 0 的 bit12 就可以控制自动协商，此时我们把寄存器 0 的 bit12 置为 0 就可以了，自动协商禁用了，我们就可以配置速度了，要工作在百兆模式，那么 bit13 置为 1，bit6 置为 0，通过上表我们可以得出此时这个寄存器应该配置成 0010_0001_0000_0000（16'h2100）。

知道了要配置的寄存器值，那我们就可以通过序列完成对 phy 的初始化，代码如下：

```
parameter speed = 2'b01, //10 为千兆，01 为百兆，00 为十兆

reg [2:0] reg_cnt;
//配置寄存器
always @ (reg_cnt)
begin
    case (reg_cnt)
        0 : reg_data <= {1'b0, 5'd0, 16'h0900}; //掉电
        1 : reg_data <= {1'b0, 5'd0,16'h2140&
{2'h3,speed[0],6'h3f,speed[1],6'h3f}}; //修改网卡速率，设置全双工
    endcase
end
```

其中，reg_data 的高 5bit 是 phy 的寄存器地址，低 16 位是配置进去的值。为了方便修改，这里我们使用 parameter 声明了一个 speed，用户只需修改 speed 的值便能完成对网卡速率的修改。

代码中只是对 PHY 的寄存器 0 进行了两次配置，如果需要配置其他的寄存器，只需添加新的序列即可。

有了待配置的寄存器值，这个时候就需要一个模块来控制每一次的寄存器操作，这里我们通过状态机实现，代码如下：

```
parameter MAX_CNT = 2 //要配置的寄存器个数

//配置 PHY 寄存器状态控制
reg [1:0] state;

always @ (negedge mdc)
begin
    if (!Go) begin
```

```
        state <= 1'b0;
        start <= 1'b0;
        reg_cnt <= 'd0;
    end
    else if (reg_cnt < MAX_CNT) begin
        case (state)
            0:begin  //开始写寄存器
                start <= 1'b1;
                state <= 1'b1;
                mdio_data <= reg_data;
            end
            1:begin //写寄存器完成
                if (done) begin
                    start <= 1'b0;
                    state <= 1'b0;
                    reg_cnt <= reg_cnt + 1'b1;
                end
            end
        endcase
    end
end
```

状态机在开始工作时，首先将 start 信号拉高，以开始写 phy 寄存器。写操作完成后，done 信号便会被拉高，此时我们需要将 start 信号拉低，准备开始下一次写操作。通过 reg_cnt 和 MAX_CNT，完成对所需个数寄存器的配置。

整个设计依赖于 mdc 时钟，为了方便修改，这里我们使用 parameter 对时钟信号进行定义。设计中 mdc 时钟用的是 50KHz，产生 mdc 时钟部分代码如下

```
parameter SYS_CLOCK = 50_000_000,        //系统时钟采用 50MHz
parameter MDC_CLOCK = 2_000,     //MDC 总线时钟采用 2kHz
//产生时钟 mdc 计数器最大值
localparam MDC_CNT_M = SYS_CLOCK/MDC_CLOCK/4 - 1;

reg [19:0] div_cnt;
always@(posedge clk or negedge rst_n)
if (!rst_n)
    div_cnt <= 'd0;
else if(div_cnt < MDC_CNT_M)
    div_cnt <= div_cnt + 1'b1;
else
    div_cnt <= 'd0;

wire mdc_pulse = div_cnt == MDC_CNT_M;
```

```
reg mdc_clk;
always@(posedge clk or negedge rst_n)
if (!rst_n)
    mdc_clk <= 1'b0;
else if(mdc_pulse)
    mdc_clk <= ~mdc_clk;
```

54.5phy_reg_config 模块仿真验证

介绍完 phy_reg_config 模块后，同样的，我们需要对模块仿真，以验证模块功能。仿真代码如下：

```
`timescale 1ns/1ns
module mdio_tb;
    reg         clk;
    reg         rst_n;
    wire        eth_rst_n;
    wire        eth_mdc;
    wire        eth_mdio;
    wire [15:0] rddata;
    wire        init_done;

    pullup PUP(eth_mdio);  //模拟外部上拉电阻

    phy_reg_config
    #(
        .speed(2'b01)
    )
    u_phy_config(
        .clk          (clk       ),
        .rst_n        (rst_n     ),
        .phy_rst_n    (eth_rst_n ),
        .mdc          (eth_mdc   ),
        .mdio         (eth_mdio  ),
        .rddata       (rddata    ),
        .phy_init_done(init_done )
    );

    always #10 clk = ~clk;
    initial begin
        clk = 1;
        rst_n = 0;
```

```
        #201;
        rst_n = 1;

        @(posedge init_done);
        #2000;
        $stop;
    end
endmodule
```

在上面代码中，我们是对物理地址为 5'b00101 的寄存器 0（BMCR）进行两次写操作，首先写入的 16 位数据位 16'h0900, 之后写入的 16 位数据位 16'h2100, 仿真波形如下：

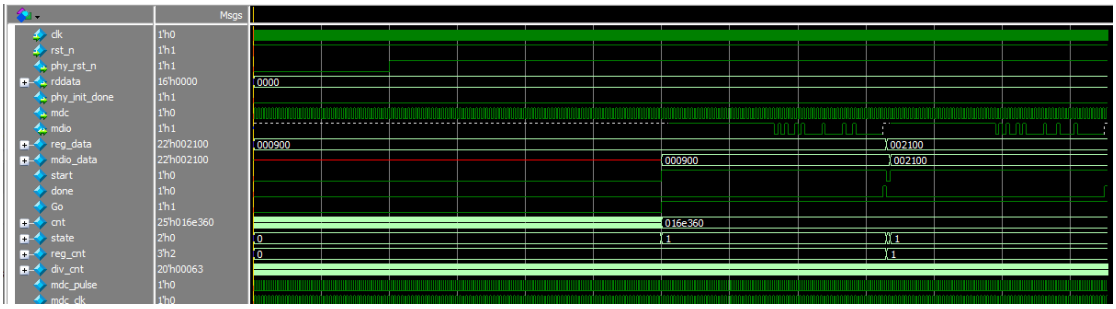


图 54-9 仿真波形

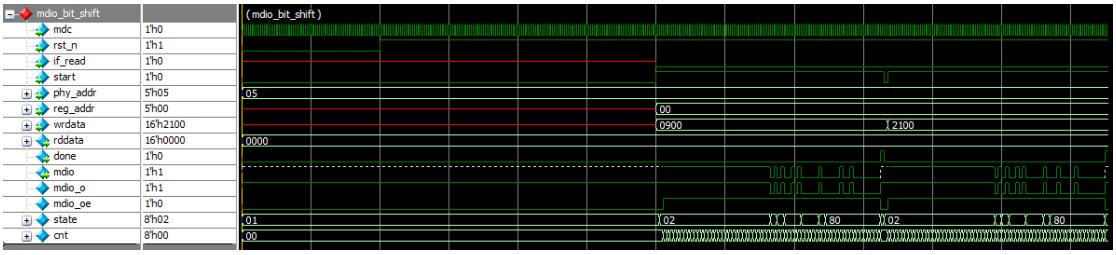


图 54-10 仿真波形（续）

从上面的波形可以看得出来，start_pos 将操作分成了两次，第一次将 16'h0900 写入了 phy 的寄存器 0，第二次将 16'h2100 写入了 phy 的寄存器 0，每次完成后 done 信号都会被拉高，用来跟上层的 phy_reg_config 模块握手，两个寄存器都初始化完成后，phy_init 信号就被拉高指示寄存器初始化完成。

因此，仿真结果与我们预期中一致，说明模块功能正常。而为了进一步确保设计能够运行在实际的硬件中工作，我们还需要为设计分配引脚，以进行板级验证。

54.6引脚分配与约束

本次验证使用的硬件平台为高云开发板，设计的目的只是为了将数据写入

到 PHY 芯片中，观察能否实现对 PHY 的配置。因此，我们只需分配其中的部分引脚即可，对应的引脚分配表如表 54-5 所示。

表 54-5 引脚分配表

Signial Name	Pin NO.	Signial Name	Pin NO.
clk	T9	phy_rst_n	G6
mdc	D3	rst_n	C15
mdio	E4		

54.7板级验证

在连接好硬件将 bit 文件下载到开发板之前，我们首先打开网络连接查看当前开发板网卡的速度模式，笔者电脑此时以太网速度如图 54-11 所示。

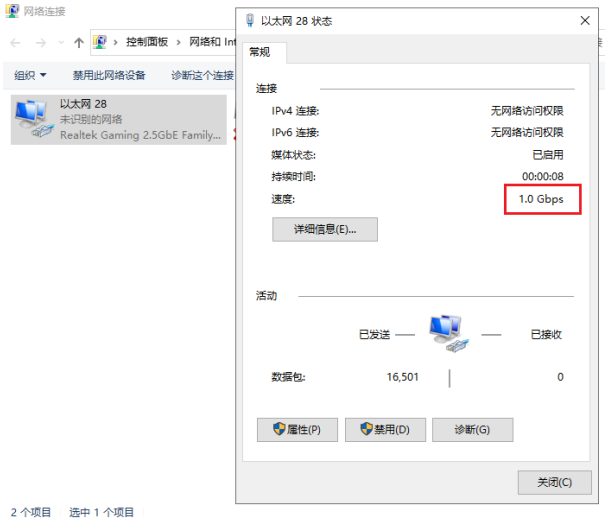


图 54-11 网卡速度

可以看到，此时速度为 1.0Gbps，也就是千兆网模式。接下来我们将生成的 bit 下载到开发板中，此时笔者电脑中以太网速度如下图 54-12：

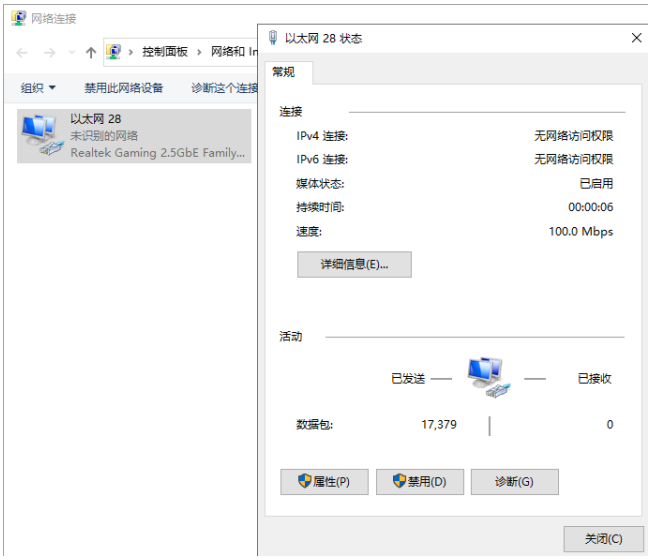


图 54-12 百兆模式

可以看到此时网络速度为 100.0Mbps，也就是百兆模式，说明 PHY 芯片被配置成功，设计能够很好的实现预期功能。

54.8总结

本章我们学习了 MDIO 接口的时序及工作原理，并设计了基于 MDIO 接口的 PHY 驱动模块，最后通过仿真和板级验证的方式验证了设计模块功能的正确性。建议读者能够跟随本实验内容，完整的进行整个实验。

55 千兆以太网 UDP 回环测试

工程源码	|----02_设计实例 |----ch55_eth_udp_lookback_rgmii
相关视频课程	
说明	

章节导读

了解完以太网各个协议以及相关模块设计，接下来我们再回过头，分析和理解前面给出的实操工程：千兆以太网 UDP 回环测试。

55.1工程梳理

经过上述章节，现在我们再来回顾以太网回环收发测试。该 demo 的系统整体结构如图 55-1 所示。

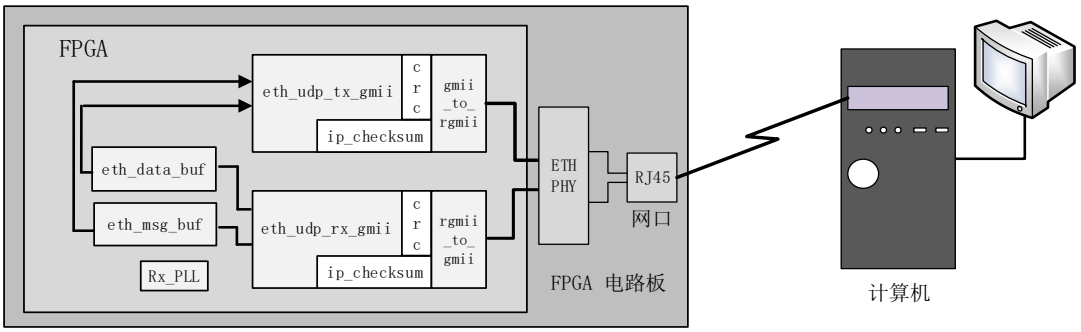


图 55-1 系统整体框图

相信经过前面一系列章节的学习，读者现在再来回顾系统原理图，已经能够清楚的知道各个模块的功能和作用了。

在整个系统中，计算机下发的 UDP 数据包，会由开发板上的 RJ45 网口接收，交给 PHY 芯片。由于开发板使用的 PHY 芯片为 RTL8211，仅支持 RGMII 接口，因此接收的数据包首先要经过 RGMII 和 GMII 的转换后，才能被接收模块校验和解包。解包后的数据和 MAC 地址等信息会被分开放在不同的 FIFO 中，交由发送模块组包。组包完成后的 UDP 数据经由 GMII 到 RGMII 转换后，由 PHY 芯片通过 RJ45 网口发送回电脑，电脑再对数据包进行校验和解包，从而实现 UDP 回环收发测试。

由于本章设计中所使用的各个模块，其所涉及到的原理以及对应设计思路在前面各个章节中都有单独的说明，设计只是对这些模块进行了简单的组合应用。因此这里就不再重复分析与解读设计，设计中各个相关部分读者可以参考

前面对应章节。

55.2总结

本章，我们简单回顾了基于 FPGA 的千兆以太网 RGMII 接口的 UDP 协议数据收发回环测试，设计中所设计到的模块，在前面章节中我们已经进行过讲解。建议读者能够基于本实验重新回顾 UDP 协议中各个章节的内容，完整的进行整个实验。

56 基于 OV5640 的以太网 RGMII 图像传输系统设计

工程源码	|----02_设计实例 |----ch56_ov5640_udp_rgmii
相关视频课程	
说明	

章节导读

FPGA 实现以太网传输的一个主要应用就是使用以太网将 FPGA 采集到的各种数据发送到 PC，继而实现如高速 ADC 实时采集数据，或者图像传感器采集图像数据等功能。这些功能在传输过程中具有数据量大、实时性要求较高的特点。如果使用基于 CPU 架构实现软件以太网协议栈，受限于 CPU 计算性能和数据组包的规律，无法达到很高的效率。而使用 FPGA 实现以太网传输，则该方案拥有稳定且高效的传输能力。

本节主要讲述了一种对数据以行为单位的编码方法。该方法采用摄像头采集数据后经由 FPGA 的 RGMII 接口通过 UDP 协议实现以太网图像传输。以该方法进行 UDP 协议下的以太网图像传输，可以有效缓解 UDP 协议的差错控制缺陷带来的传输质量不佳问题。

56.1UDP 协议的特点

在前面的学习内容中，我们对 UDP 协议的特点和适用场景进行过分析。我们通过以太网进行数据传输时，通常对于一些对数据正确性要求较低（允许少量丢失）但是不能有较大时延的场景，都会使用 UDP 协议。UDP 协议是传输层的一种协议，该协议具有以下特点：

1. 提供不可靠传输。在进行数据传输时 UDP 提供最大努力的交付，但不保证可靠交付。
2. 高效而无连接性。UDP 协议在传输数据前不建立连接，不对数据报进行检查和修改，无须等待对方的应答，所以会出现分组丢失、重复、乱序等问题，如果因为网络原因没有传送到对端，UDP 也不会给应用层返回错误信息。但正因为 UDP 在进行发送时不需要建立连接，所以其在网络开销较小的同时工作时效也相当高。
3. UDP 没有拥塞控制。应用层能够更好的控制要发送的数据和发送时间，网络中的拥塞控制也不会影响主机的发送速率，因此 UDP 协议常常被

用于某些对数据发送速率有一定要求，能容忍一些数据的丢失，但是不能允许有较大的延时的场景，如直播、视频等。

4. UDP 在现场测控领域，往往面向的是分布化的控制器、检测器等应用。现场测控领域的电磁环境往往较为复杂，基于此，现场通信中，若某一应用要将一组数据传送给网络中的另一个节点，可由 UDP 进程将数据加上报头后传送给 IP 进程。由于 UDP 协议省去了建立连接和拆除连接的过程，取消了重发检验机制，所以能够达到较高的通信速率。

正因为 UDP 协议有如上种种优点，所以可以对 UDP 协议图像传输的策略进行改进，而降低干扰的影响后，UDP 协议仍然可以作为图像传输的一种优质方案。

56.2图像数据编码原理

我们前面说过：以太网图像数据的传输通常采用 UDP 协议，图像数据在传输时一旦受到外界干扰发生数据丢失，而数据的丢失会导致图像发生断层，割裂，压缩等现象。

正因为如此，我们可以进行如下有针对性的改进后，仍保留使用 UDP 协议：如果我们对图像数据进行处理，将摄像头采集到的图像数据按照一定的编号方式例如以行为单位编号后，再通过 UDP 协议经由以太网传输到电脑，电脑接收 FPGA 发送的图像数据解析后按照编号将相应的图像内容绘制到电脑显示屏的对应位置上，就能有效解决该问题。

经过该种编码方法改进后，当数据在通过 UDP 协议传输的时候，理论上即使发生了少量数据丢失也不会出现断层割裂等效果。而对于图像传输显示这类场景，对于图像数据的准确性要求较低，即使一幅图像中有两行数据的丢失，只要不是每幅图像都有该丢失情况，在每秒几十上百帧的刷新率下，其影响也极小。

为了让上述理论能够更加通俗易懂，我们结合图像，来对该理论进行讲解。

本次设计原理如下图 56-1 所示。

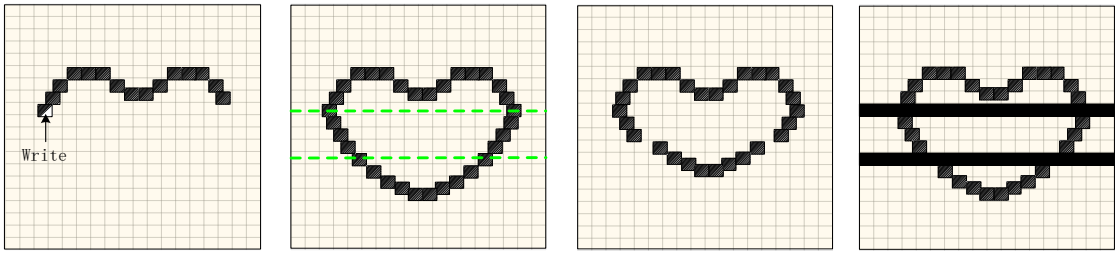


图 56-1 数据丢失后的不同结果

左边第一张为图像的写入过程，图像数据从左至右被依次写入。

第二张图为正常写入后所应展现出的画面，假设被标绿线的行在传输过程中丢失了，在未对行号进行编码的情况下，将会得到第三张图中被压缩的结果，而后续如果仍有图像传输过来，甚至可能出现两张图像重叠的情况，会影响后续图像数据的显示效果。

如果我们对每行数据进行编号，即使数据在传输过程中有部分行的数据丢失了，但因为序号的存在，它们仍会排列在自己所应处的位置。

举例来说：上图 56-1 中，假设绿色行发生丢失，数据在显示屏上的显示画面，如第四张图，被丢失的行由于没有数据，会显示黑色，即使数据丢失，图像也不会发生压缩等现象，只会在当前帧有影响，不会影响后续帧的图像。在每秒几十帧的刷新率下，一两行数据的丢失对人眼而言甚至无法识别到，或者只是看到黑线一闪而过。

在理解了本次设计最终目的实现原理后，接下来可以开始进行工程的设计。

56.3系统总体设计

本次实验我们将对前面 “以太网图像发送 PC 接收显示实验测试”一节中设计的模块进行验证。

本次板级验证的工程名为 ov5640_rgmii_udp，整个工程的系统框图如下图 56-2 所示。

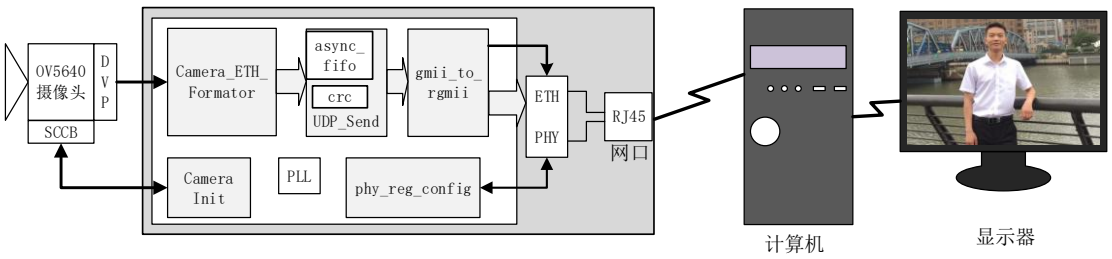


图 56-2 系统框图

结合板级验证的硬件搭建环境，本章节的原理可以以硬件环境为背景作如下对应的深化讲解：

整个系统在开始工作前，首先按照初始化 table 对摄像头进行初始化。

随后摄像头将采集到的数据经由 DVP 接口输送给 Camera_ETH_Formator 模块，也就是本次设计的模块，该模块会根据输入数据行场同步信号的高低电平，

以行为单位，对其进行编码。

随后数据进入一个双口 FIFO 中，当 FIFO 中的值足够进行一次以太网数据传输时，数据就会被发送模块读出。

使用 UDP 协议将读出的数据经由 CRC 校验后，经由以太网发送到 PC 机。

PC 端使用我们专门开发的显示软件（小梅哥 UDP 摄像头）接收 FPGA 发送的包含行编号的图像数据，解析后还原为图像内容绘制在 PC 机显示屏上。

根据前面学习内容的基础，在本章，我们着重讲解图像编码模块的作用，并讲解如何为前面讲解的实用型 UDP 收发器添加 FIFO 缓存。

56.4图像编码模块介绍

56.4.1图像编码模块作用

根据前面介绍的设计思想，本次模块设计的目的是实现对图像数据以行为单位进行编号。如果实现数据以行为单位编号，就需要知道当前输入的数据位于每帧图像数据的哪一行，这一需求是根据定义行同步信号和场同步信号并对其进行计数而获得的。

模块 Camera_ETH_Formator 在整个系统内，可以起到防止在图像显示的过程中由于某一行数据的丢失，导致的图像压缩断层现象发生作用。在加入了该模块后，数据会按照其排列序号一行行进行排列，即使有一行或多行数据丢失，其余行数据也会排列在自己所处的位置。

综合以上信息，设计 Camera_ETH_Formator 的结构如图 56-3 所示。

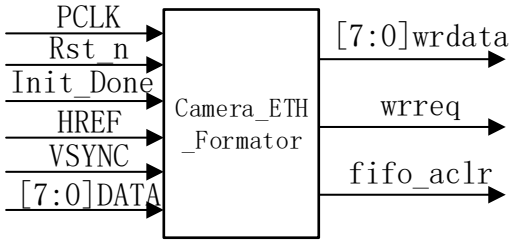


图 56-3 图像编码模块

其中各个信号的含义如表 56-1 所示。

表 56-1 Camera_ETH_Formator 模块端口列表

信号名称	I/O	位宽	信号功能
PCLK	I	1	像素时钟
Rst_n	I	1	模块复位
Init_Done	I	1	摄像头初始化完成信号

HREF	I	1	行同步信号
VSYNC	I	1	场同步信号
DATA	I	8	写入数据
wrdata	O	8	读出数据
wrreq	I	1	数据输出使能

当 HREF 由低电平转为高电平时，DATA 开始输入到图像数据编码模块，当 HREF 从高电平转为低电平时代表一行的数据输出完成，当模块中的数据处理完成后便会产生 wrreq 信号，并输出 8 位的 wrdata 数据。

我们以 1280*720 的显示屏为例，在摄像头将采集到的 RGB565 格式数据通过以太网发送给 PC 的显示屏显示的过程中，每一行有两倍 1280 个字节即 2560 字节有效数据。依照前文我们介绍的设计思想：为了让部分数据即使在丢失的情况下其余行数据也能显示在自己应处的位置，我们可以在每一行的数据前加上 2 个字节的行号。也就是说，以太网每一帧，需要发送一行的数据外加两个字节的行号。

56.4.2图像编码模块功能实现

为了方便理解和设计，我们绘制了本次模块各个信号之间的时序图，如图 56-4 所示。

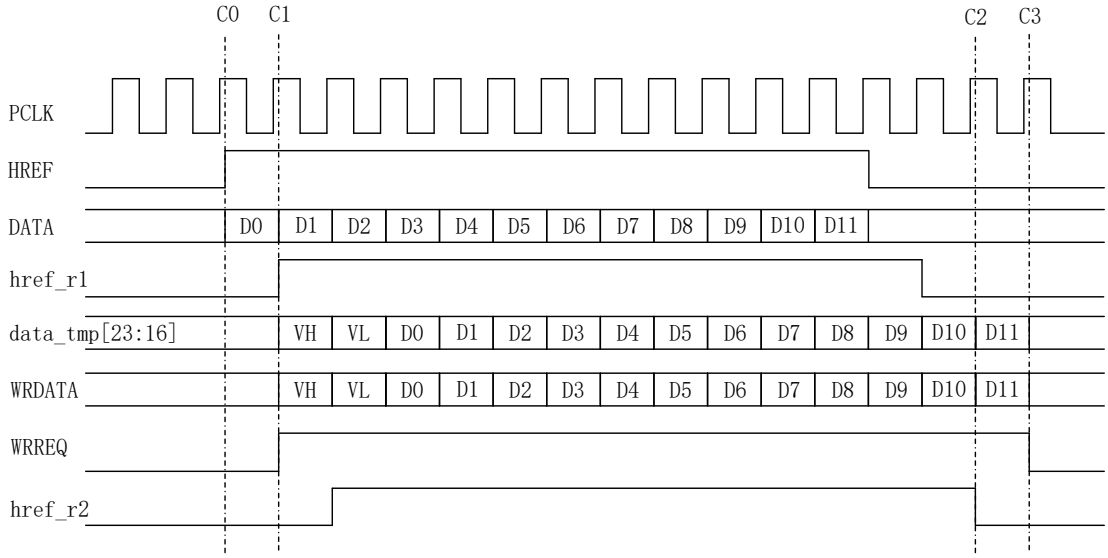


图 56-4 图像编码模块时序图

上图中：href_r1 和 href_r2 为摄像头行同步 HREF 寄存后得到的信号，data_tmp 为 24 位的寄存器，用来完成对数据的移位转换。

如：时序图中 C0 时刻，当 HREF 出现上升沿时，代表数据开始输入。此时我们可以用 HREF 与 href_r1 位拼接后的值表示，当{href_r1，HREF}的值位

2'b01 时，HREF 的电平由低到高，数据开始输入；当{href_r1，HREF}的值为 2'b10 时，HREF 的电平由高到低，一行数据输入完毕，数据停止输入。

本次模块设计的目的是以行为单位，对数据进行编号，根据 HREF 的电平变化，我们可以利用计数器对行信号进行下降沿次数的累加生成行号，该部分代码实现如下：

```
reg [15:0]Vcnt;

always@(posedge PCLK or negedge Rst_n)
if(!Rst_n)
    Vcnt <= #1 0;
else if(VSYNC)
    Vcnt <= #1 0;
else if({href_r1,HREF} == 2'b10)
    Vcnt <= #1 Vcnt + 1'd1;
```

代码中：Vcnt 为行号计数器，每当 HREF 电平状态由高变低时，代表一行数据输出完毕，此时我们只需使计数器自加一即可，当VSYNC信号为高电平，即代表一幅图像数据输入完成，此时将计数器清零。

完成了行号的产生，接下来就是将行号写入数据中。如时序图 56-4 中所示，C0 处检测到 HREF 信号的上升沿，由于该上升沿在时钟上升沿之后才到来，所以实际上该信号是在下一个时钟上升沿到来时才被采集到。该信号被采集后，随即将行号写入 data_tmp 寄存器中。当 2 字节的行号输出完成后，再开始输出图像数据，相应的代码如下：

```
assign wrdata = data_tmp[23:16];

always@(posedge PCLK)begin
    href_r1 <=  #1 HREF;
    href_r2 <=  #1 href_r1;
end

always@(posedge PCLK or negedge Rst_n)
if(!Rst_n)
    data_tmp <= #1 1'b0;
else if({href_r1,HREF} == 2'b01)
    data_tmp <= #1 {Vcnt[7:0],Vcnt[15:8],DATA};
else
    data_tmp <= #1 {data_tmp[15:0],DATA};
```

这里需要注意的一点是，网络在传输数据时，使用的是大端字节序，即先存放高位再存放低位，而计算机在存储数据时使用的时小端字节序，即先存放

数据的低位再存放数据的高位，所以再输出行号时，为了防止数据错误，需要将其高 8 位与低八位的位置互换，这也就是为什么代码中 Vcnt[7:0]先输出而 Vcnt[15:8]后输出。

确定了数据的输出顺序后，接下来便是确定数据的输出使能信号。

观察时序图，每当 data_tmp 中被写入数据时，即 C1 时刻，输出使能信号 wrreq 转为高电平。此时 wrreq 信号有效，wrdata 开始输出数据，其相应的表达形式{href_r1，HREF}=2'b01；当 data_tmp 中数据输出完毕时，即 C3 时刻，wrreq 转为低电平。此时 wrreq 信号无效，wrdata 停止输出数据，相应的表达形式{href_r2 | href_r1}=1。

该部分代码实现如下：

```
always@(posedge PCLK)
if({href_r1,HREF} == 2'b01)
    wrreq <= #1 1'b1;
else if(href_r2 | href_r1)
    wrreq <= #1 1'b1;
else
    wrreq <= #1 1'b0;
```

另外，我们在设计中需要给出缓存 FIFO 的清零时机。当摄像头初始化完成，且场同步信号拉高时，给出清零条件。具体实现代码如下：

```
always @ (posedge PCLK or negedge Rst_n)
if (!Rst_n)
    fifo_aclr <= #1 1'b1;
else if (Init_Done && VSYNC )
    fifo_aclr <= #1 1'b0;
else
    fifo_aclr <= #1 fifo_aclr;
```

至此该模块的设计便完成了，接下来就是对本次模块设计的仿真验证。

56.4.3图像编码模块仿真验证

56.4.3.1图像编码模块仿真文件设计

在进行仿真验证时，仿真程序验证的激励部分可以作如下设计：

1. 令 DATA 在 HREF 信号为高电平时从零开始自加。
2. 令场同步信号 VSYNC 为低电平，将行同步信号 HREF 设为高电平。
3. 延时一段时间后将行同步信号 HREF 置为低电平。

4. 重复数次，观察各数据关系是否正确。
5. 令场同步信号变为高电平，延时一端时间后拉低，再将 HREF 信号设为高电平，延时一段时间后变为低电平，观察行号是否重新计数。

该部分代码设计如下：

```
`timescale 1ns/1ns

module Camera_ETH_Formator_tb;

    reg Rst_n;
    reg PCLK;
    reg HREF;
    reg VSYNC;
    reg [7:0]DATA;

    wire [7:0]wrdata;
    wire wrreq;

    initial PCLK = 1;
    always#10 PCLK = ~PCLK;

    Camera_ETH_Formator Camera_ETH_Formator(
    .Rst_n(Rst_n),
    .PCLK(PCLK),
    .HREF(HREF),
    .VSYNC(VSYNC),
    .DATA(DATA),

    .wrdata(wrdata),
    .wrreq(wrreq)
);

    initial begin
        Rst_n = 0;
        HREF = 0;
        VSYNC = 0;
        #201;
        Rst_n = 1;
        repeat (10)begin
            HREF = 1;
            #201;
            HREF = 0;
            #201;
```

```
        end
        #201;
        VSYNC = 1;
        #201;
        VSYNC = 0;
        repeat (10)begin
            HREF = 1;
            #201;
            HREF = 0;
            #201;
        end
        $stop;
    end

    always@(posedge PCLK)
    if(!Rst_n)
        DATA <= #1 0;
    else if(HREF)
        DATA <= #1 DATA + 1'b1;
    else
        DATA <= #1 DATA;
endmodule
```

设计完激励测试文件后，接下来进行仿真，进而得到仿真输出结果。

56.4.3.2仿真结果分析

本次仿真文件为 Camera_ETH_Formator_tb，仿真后得到的波形如图 56-5 所示。

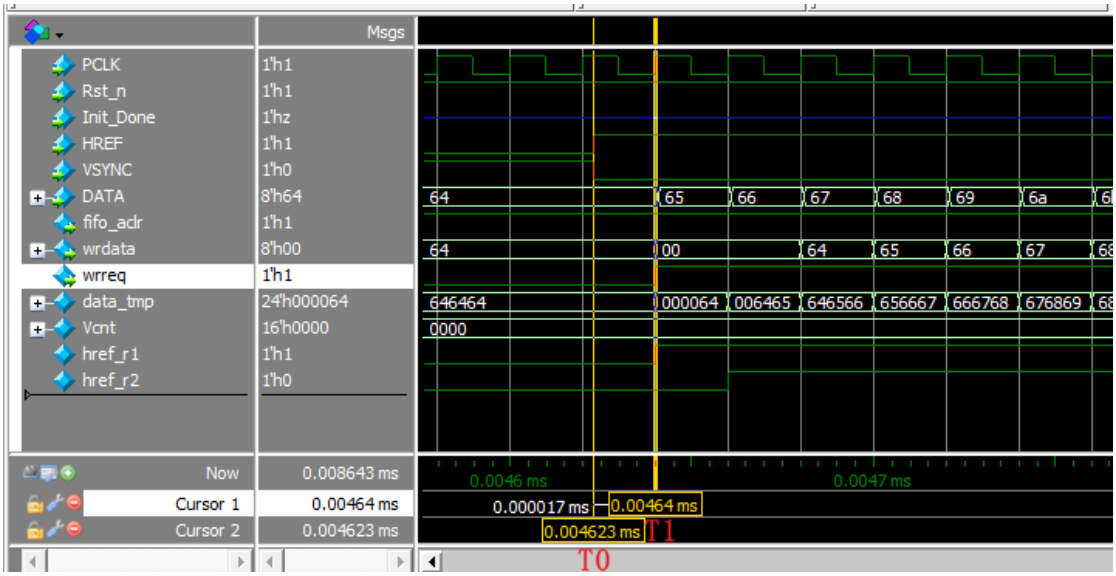


图 56-5 仿真波形图

为了方便描述不同时刻的波形，我们将图 56-5 中左右两个时刻分别命名为 T0 和 T1。可以看到 T0 时刻场同步信号 VSYNC 由高电平转为低电平，行同步信号 HREF 由低电平转为高电平，模拟了一幅图像的第一行数据开始传输。接下来对信号波形进行分析。

1. 行计数器 Vcnt 变化值变化

T0 时刻行场同步信号发生变化，由于该变化发生在时钟上升沿之后，所以数据在下一时钟沿到来才会变化。根据 D 触发器的特性，数据 DATA 会经过一小段的延迟即 T1 时刻后再发生改变。T1 时刻，按设计，行计数器 Vcnt 值应该为 0，仿真中行计数器 Vcnt 的值确实为 0，证明该部分逻辑设计正确。

2. 数据移位寄存器 data_tmp 变化值分析

T1 时刻，Vcnt 的值被写入 data_tmp[23:8]中，输入数据 DATA 被写入 data_tmp[7:0]中。由于此时 Vcnt 为 0，DATA 为 16'h64，所以 T1 时刻 data_tmp 的值为 24'h000064。

在下一个时钟上升沿 data_tmp[23:16]将会被输出，data_tmp 中的数据左移 8 位，新输入的 DATA 的值被写入到 data_tmp[7:0]中。可以看到仿真中得到的结果与理论上应该得到的一致，因此这部分仿真逻辑正确。

3. 输出数据 wrdata 变化值分析

wrdata 在每个时钟，上升沿都会输出 data_tmp 中的高 8 位。由于 T1 时刻 data_tmp 的值为 24'h000064，所以从 T1 时刻开始的三个时钟周期，wrdata 输出的值依次应该为 16'h0，16'h0，16'h64。仿真波形中的结果与该数据一致，因此该部分逻辑设计正确。

需要注意的一点是，wrdata 在输出 data 数据时是按顺序输出的。而在输出行号数据时，为了匹配大端字节序和小端字节序的顺序，我们将行号的高 8 位与低 8 位数据进行了互换。所以 wrdata 输出的 16'h0，16'h0，16'h64 分别对应 T1 时刻的 Vcnt[7:0]，Vcnt[15:8]，DATA。

4. 输出使能信号 wrreq 变化值分析

wrreq 为 wrdata 的使能信号，每当检测到有数据输入时有效，变为高电平；当检测到没有数据输入后将该信号将变为低电平。仿真结果正确，因而该部分逻辑代码设计正确。

56.5phy_reg_config 控制器模块例化

我们在前面 phy_reg_config 控制器设计的章节，也专门着重讲解了 phy_reg_config 控制器的用途和实现方法。它的任务主要是向网卡控制芯片写入配置寄存器初值。在这里，我们直接将该模块在顶层进行例化即可。

```
wire phy_init;
phy_reg_config phy_reg_config_inst(
    .Clk(clk_50m),
    .Rst_n(global_rst_n),
    .Phy_rst_n(eth_rst_n),
    .mdc(eth_mdc),
    .mdio(eth_mdio),
    .Phy_init_done(phy_init)
);
```

56.6实用型 UDP 收发器例化

数据在实现了编号后即可进行输出，所以我们需要一个输出信号，由于 UDP 协议需要大数据量的传输，在信号输出之前我们可以使用一个 FIFO 将编序号的行输出数据进行缓存。数据的存储速率和读出速率不同，所以我们这里需要使用异步时钟 FIFO。设计了 FIFO 以后，还可以通过设计一个写请求信号接口实现数据能够有序而受控写入。

我们在前面的内容中，已经讲解了含 FIFO 的 UDP 发送模块设计方法，这里，我们直接对该收发器进行例化即可。

56.7GMII 转 RGMII 模块调用例化

我们在前面的章节中，已经讲解了 GMII 转 RGMII 和 RGMII 转 GMII 的方法。在本节中，由于我们需要将 UDP_Send 模块生成的 GMII 发送信号转换为本工程指定的 RGMII 发送接口信号。所以，我们在工程顶层，直接例化 GMII 转 RGMII 模块即可。

```
gmii_to_rgmii gmii_to_rgmii(
    .reset_n(rst_n),
    .gmii_tx_clk(GMII_GTXC),
    .gmii_txd(GMII_TXD),
    .gmii_txen(GMII_TXEN),
    .gmii_txer(0),
    .rgmii_tx_clk(eth_gtxc),
```

```
    .rgmii_txd(eth_txd),
    .rgmii_txen(eth_txen)
);
```

这样，整个工程的数据发送链路，就设计完成了。

56.8其他涉及模块说明

由于本章节的讲授重点放在以太网发送部分的设计之上，所以本章中涉及到的锁相环和摄像头初始化部分的内容，就不作过多分析讲解。实际工程应用时，只需将其按工作需求进行配置，并在顶层例化使用即可。

56.9板级验证

56.9.1系统所需硬件

1. 开发板 x1
2. OV5640 摄像头 x1
3. 电源线 x1
4. 下载器 x1
5. 千兆网线 x1

56.9.2硬件连接

本次系统设计硬件方面连接如下图 56-6 所示。

1. 将摄像头连接到 40pin 扩展口上。
2. 用网线将开发板网口与电脑网口连接。
3. 插接好下载器。
4. 连接好电源线，电源线连接电源适配器为开发板供电。

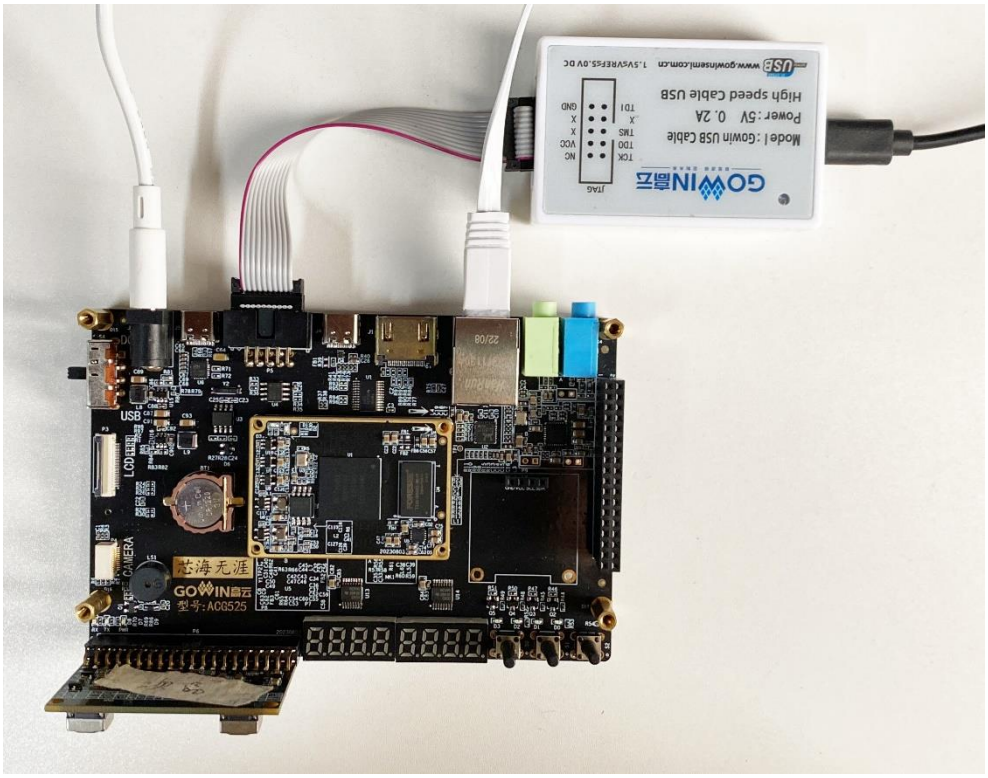


图 56-6 硬件连接图

56.9.3管脚绑定

对工程进行分析和综合，确认无误后，为设计分配管脚并约束电平，本次设计引脚约束表如下表 56-2 所示。

表 56-2 引脚分配表

Signial Name	Pin NO.		Signial Name	Pin NO.
camera_data[7]	F18		camera_sclk	K14
camera_data[6]	F17		camera_sdat	J13
camera_data[5]	G18		eth_txd[3]	F1
camera_data[4]	G16		eth_txd[2]	F2
camera_data[3]	D18		eth_txd[1]	D1
camera_data[2]	D17		eth_txd[0]	D2
camera_data[1]	G14		eth_gtxc	C1
camera_data[0]	F14		eth_mdc	D3
camera_href	K15		eth_mdio	E4
camera_vsync	K16		eth_rst_n	G6
camera_pclk	H14		eth_txen	C2
camera_xclk	H13		clk	T9
			rst_n	C15

56.9.4下载与验证

我们在前面的以太网数据传输实验体验的内容，和本小节的板级验证环节

内容完全相同。相信各位读者已经抢先对本工程的实际效果进行了体验，这里，我们也就不再重复讲解实操的步骤了，如需进行实际验证，可以回到“以太网图像传输工程实操”的内容进行回顾学习。至此，本次设计完成。

56.10总结与思考

本节设计实现了对摄像头采集的数据以行为单位进行编码排序，以防止数据在通过以太网进行传输显示时由于数据丢失而导致的图像压缩和错位现象，建议读者能够跟随本实验内容，完整的进行整个实验。

57 基于双目 OV5640 的以太网 RGMII 图像传输设计

工程源码	|----02_设计实例 |----ch57_ov5640x2_udp_rgmii
相关视频课程	
说明	

章节导读

双目摄像头常常被用作暗光增强、3D 建模以及测距上，相较于单目摄像头仅仅只能用于图像采集，双目摄像头的应用场景更加广泛，所能实现的功能更为全面。本节将以双目摄像头为实验对象，实现对数据的采集显示。

本节主要实现了一种对双目摄像头采集数据的图像拼接设计，并结合了具体的工程，完成了双目摄像头图像采集的以太网传输显示。本节设计基于上节内容为基础，在对不同源数据进行相应处理后，将相同行的数据组合成新的一行图像数据。组合后的图像数据通过以太网传输到 PC 机上。PC 机通过我们开发的软件，将数据解析还原后实现对图像的拼接显示。

57.1双目摄像头与图像数据

在前面我们曾做过有关单目摄像头以太网传图显示系统的介绍。双目摄像头相对于单目而言虽然多了一个传感器输入，但是其原理与单目以太网传图显示系统基本一致。系统工作时首先对摄像头初始化，随后对数据进行行编号后存入 FIFO 中，通过 UDP 发送模块发送给 PC 端显示。不同的是，双目摄像头的图像输入源有两个，为了让最终能将图像同时显示在 PC 的显示设备上，我们在发送到 PC 端前先要对其进行数据拼接操作，随后将数据存储进 FIFO 中等待数据足够后再进行发送。因为双目摄像头的以太网发送部分原理与单目摄像头一致，所以本节仅对数据处理部分进行讲解。

57.2系统整体设计

本次工程设计，大部分结构和单目摄像头以太网图像传输的总体结构图相似，不同的是，在设计中，新增了一个对双目摄像头采集数据图像合并的模块 controller，同时在模块中，例化了两个 FIFO 作为两个半行图像数据的缓存。缓存的数据经过图像合并模块的处理后输出一行图像数据，交付给 UDP_Send 模块。这样，在 UDP_Send 模块看来，接收的数据和单目摄像头工程一样，也是

一整行带行号的数据。

本次板级验证的工程名为 ov5640x2_udp_rgmii，整个工程的系统框图如下所示。

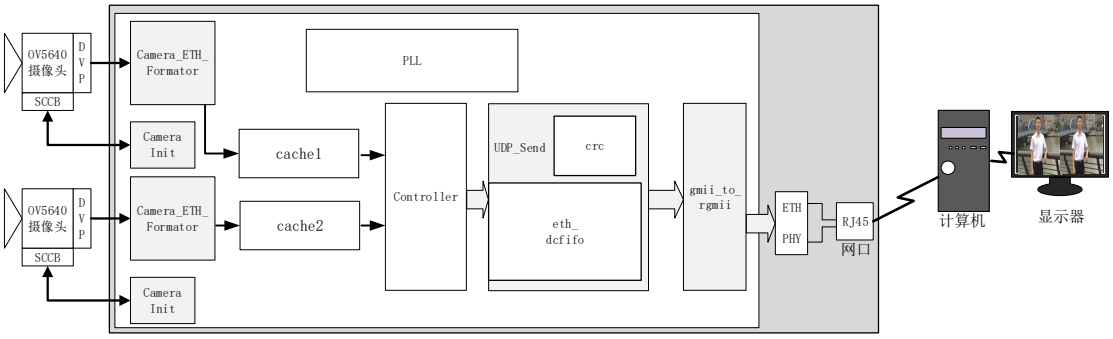


图 57-1 系统框图

结合板级验证的硬件搭建环境，本次板级验证的原理可以以硬件环境为背景作如下对应的深化讲解：

1. 整个系统在开始工作前，首先按照 camera_init 模块的设计对摄像头进行初始化。
2. 随后摄像头将采集到的数据经由 DVP 接口输送给 Camera_ETH_Formator 模块，该模块会根据输入数据行场同步信号的高低电平，以行为单位，对其进行编码。
3. 数据进入一个双口 FIFO 中，当 FIFO 中的值足够进行一次以太网数据传输时，数据就会被 controller 模块读出。
4. controller 模块会先对 cache1 中的数据读取，在读取了一行包含有行号的图像数据后，再对 cache2 中的数据读取。最终输出给以太网发送模块的 fifo。
5. 使用 UDP 协议将以太网发送模块 async_dcfifo 中的数据经过 crc 校验后，经由以太网发送到 PC 机。
6. PC 端使用我们专门开发的显示软件（小梅哥 UDP 摄像头.exe）接收 FPGA 发送的包含行编号的图像数据，解析后还原为图像内容绘制在 PC 机显示屏上。

57.3各子模块设计

57.3.1摄像头初始化模块组（camera_init）

和单目摄像头工程一致，要想让摄像头按照设计需求工作，我们首先需要对摄像头进行初始化配置。为了方便用户操作，我们提供了一份 initial_table 文件，摄像头在初始化时会根据该文件对相关寄存器进行配置。用户若想修改摄像头的工作模式和输出格式，只需根据摄像头数据手册上的指示，对 initial_table 文件中相关寄存器的值进行修改即可。

57.3.2行号插入模块（Camera_ETH_Formator）

本次设计使用双目摄像头，通过双目转接板，将两个摄像头采集到的数据通过拓展接口输送给开发板。开发板接收到数据后根据输送过来的行场同步信号判断当前行数据在整幅图像中的位置。根据行信号判断是否为一行数据的开始，每当一行数据开始发送时，便会将行号计数器的值插入其中。当一行数据发送完毕后其内部的行号计数器便会加一，等到下一行数据到来，再将加一后的行号计数器的值插入该行图像数据中。根据场信号判断一帧图像是否结束，一帧图像发送完毕后便会对行号清零，并重新开始计数。

图像数据经过了行号插入后，接下来便是对图像数据进行数据拼接。

57.3.3数据拼接模块（controller）

因为本次设计使用的双目摄像头，所以对应的图像数据源有两个。要想将两个图像源的图像数据显示在一个显示屏上，我们便需要对其进行拼接处理。经过了行号的插入，我们已经实现了对每行图像数据在对应整幅图像中的定位，而想对数据拼接处理只需要根据行号将相同行号的数据排列在同一行即可。对此我们可以设计一个数据拼接模块，该模块工作时会轮流读取两个信号源行号插入后的数据，每当读取完其中一个信号源一行带行号数据后便跳转到另一个数据源读取一行不带行号的数据。这样既实现了对图像数据的拼接，又不会因为行号对拼接后的图像产生影响。

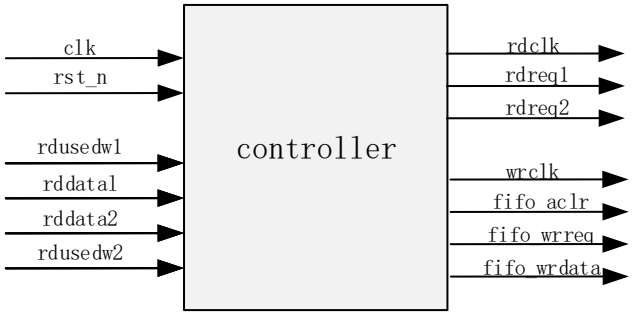


图 57-2 controller 模块端口图

模块各个信号所代表的含义如表 57-1 所示。

表 57-1 信号功能表

信号名称	位宽	I/O	信号功能
clk	1	I	模块时钟
rst_n	1	I	复位信号
rddata1	8	I	左侧读数据
rdusedw1	14	I	左侧图像读数据量
rddata2	8	I	右侧图像数据
rdusedw2	14	I	右侧图像读数据量
rdclk	1	o	读出图像时钟
rdreql	1	o	左侧图像读使能信号 1
rdreq2	1	o	右侧图像读使能信号 2
wrclk	1	o	写合并后 fifo 使能信号
fifo_aclr	1	o	清合并后 fifo 信号
fifo_wrreq	8	o	合并后的 fifo 输出使能
fifo_wrdata	1	o	合并后的 fifo 输出数据

cache1 和 cache2 是两个配置相同，功能也一致的 dcfifo。通道 1 和通道 2 的图像数据经过行号处理后会分别存储进 cache1 和 cache2 中，等待 controller 模块进行读取。这里以 cache1 的图像数据比 cache2 先显示为例，对整个 controller 模块的工作流程进行说明：

- 每当 wrreq1 信号高电平有效时，8 位的图像数据 wrdata1 便会写入 cache1 中。
- cache1 根据输入的数据量产生 14 位的 rdusedw1 信号，用来表示当前 dcfifo 中所能对数据进行读取的次数。同时将该信号传递给 controller 模块
- controller 模块对 rdusedw1 信号进行判断，当其大于摄像头输出图像的一行长度与行号长度的和。这里以摄像头输出的图像分辨率为 640*720 为例，即大于 640*2+2=1282 时，controller 模块便会使 rdreq1 信号为高

电平，读取 cache1 中的图像数据，即 rddata1。每次都会从中读取 1282 个字节的数据。

- 由于 cache1 的图像数据先显示，所以不需要进行处理，controller 会发出写请求信号给以太网的 dcfifo，即 fifo_wrreq。同时，将读取进 controller 的数据按照每次 8 位的数据量进行输出。
- controller 模块采集完便会采集 cache2 中的数据。cache2 的工作流程同理，但不同的是 controller 模块会将 cache2 中的行号剔除，将当行的图像数据紧随 cache1 中当前行的数据输出。最终图像数据在显示设备上显示时便会达到拼接的效果，两个摄像头采集到的数据同时显示在显示屏中。

本次 cache1 和 cache2 中 dcfifo 的配置除了深度不同其余部分配置一致。考虑到 controller 模块在读取其中一个 cache 中的数据时，另一个 cache 中仍可能在存储数据，所以 dcfifo 的深度应该大于两行摄像头采集的数据量加上两个行号。因此 dcfifo 的深度应该大于(1080*2+2)*2=4324。因此该项我们将其设置为 8192。其相关配置如下图 57-3 所示，其余部分保持默认。

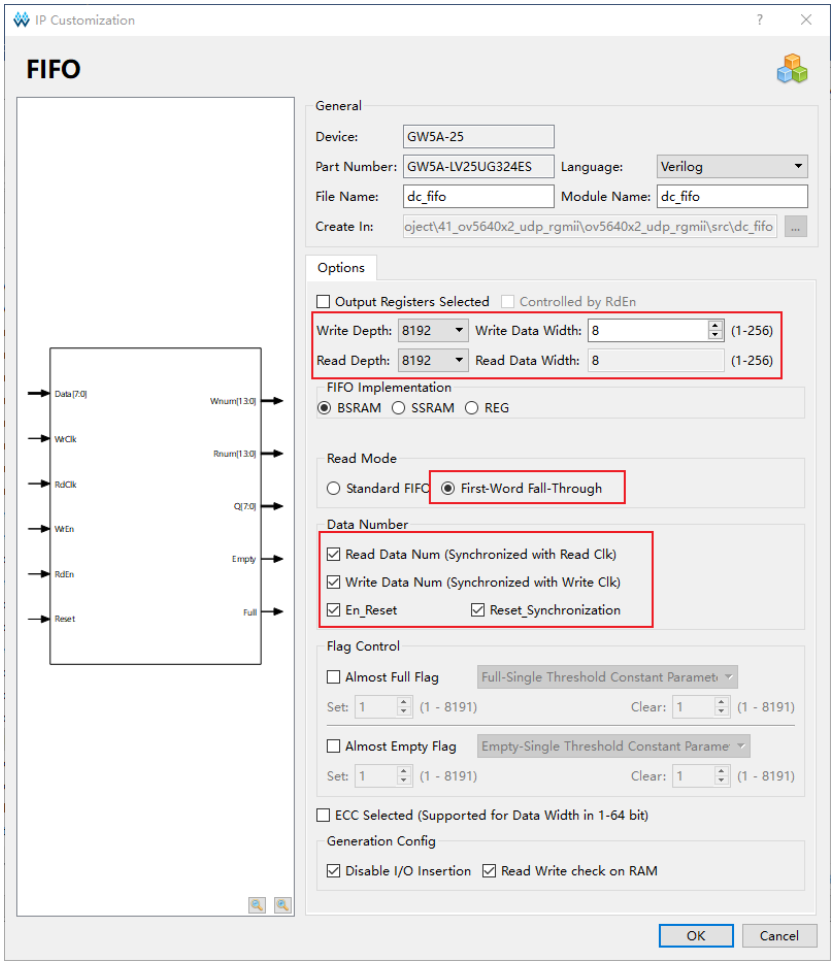


图 57-3 dcfifo 配置

在清楚了 controller 模块的工作原理并配置完了 dcfifo 之后接下来即可开始本次模块的功能设计。

57.4功能设计

本次模块设计主要实现的是对两个摄像头通道输入的图像数据分别处理，而该模块对数据的处理又可以分为三个状态。根据这三个状态，我们可以设计一个状态机，通过控制其状态的跳转，来达到分别处理的目的。本次设计的状态机为二段式状态机，其状态转移图如所示。

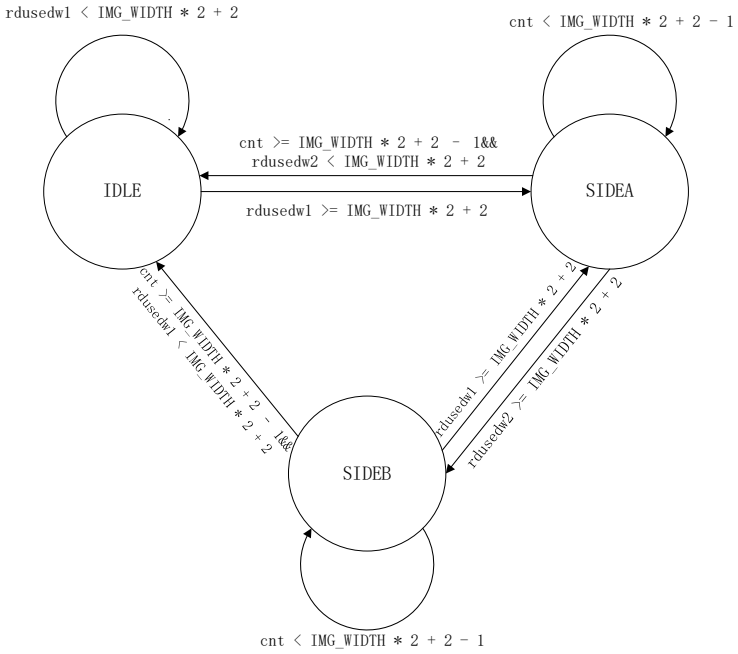


图 57-4 二段状态机状态转移图

IMG_WIDTH 为摄像头输出的图像像素宽度，本次设计我们以该值为 640 为例进行设计。IDLE，SIDEA，SIDEB 为三个不同的状态，在各个状态下实现的功能如下：

- IDLE：空闲状态。状态机处于该状态时，会对 cache1 中可读数据的数据量进行判断。当满足摄像头输出的图像数据和行号之和，即 640*2+2=1282 时，让状态跳转到 SIDEA，否则将一直处于该状态。
- SIDEA：通道 1 读取状态。状态机处于该状态时，会读取 cache1 中的数据，并对数据进行计数。当计数值小于 1281 时，会一直在该状态循环读取，并将读取的数据存储进以太网的发送 FIFO 中。当计数值大于

1281 时，判断此时 cache2 中可读取的数据量是否满足 1282.如果满足便跳转到 SIDEB，否则跳转到 IDLE。

- SIDEB：通道 B 读取状态。状态机处于该状态时，会读取 cache2 中的数据，并对数据进行计数。当计数值小于 1281 时，会一直在该状态循环读取，并舍弃最开始输入的行号数据后存储进以太网的发送 FIFO 中。当计数值大于设定好的图像数据量时，判断此时 cache1 中可读取的数据量是否满足 1282，如果满足便跳转到 SIDEA 状态，否则跳转到 IDLE 状态。

这里可能有读者会问，为什么在 IDLE 状态时要先判断 cache1 中的数据量而不是 cache2 中的数据量。这就涉及到了二者的优先级问题。本次涉及我们以 cache1 的数据作为最终显示图像的起始数据，则 cache1 数据相较于 cache2 数据为高优先级。只有先对 cache1 的行数据处理输出后再对 cache2 的行数据处理输出，不断地循环处理才能使最终的显示图像达到预期的效果。

该部分相应的代码如下：

```
case (state)
    IDLE:
        begin
            if (rdusedw1 >= IMG_WIDTH * 2 + 2) begin
                cnt <= 'd0;
                state <= SIDEA;
            end else
                state <= IDLE;
        end
    SIDEA:
        begin
            if (cnt < IMG_WIDTH * 2 + 2 - 1) begin
                cnt <= cnt + 1'b1;
                state <= SIDEA;
            end else if (rdusedw2 >= IMG_WIDTH * 2 + 2) begin
                cnt <= 'd0;
                state <= SIDEB;
            end else begin
                cnt <= 'd0;
                state <= IDLE;
            end
        end
    SIDEB:
        begin
            if (cnt < IMG_WIDTH * 2 + 2 - 1) begin
```

```
                cnt <= cnt + 1'b1;
                state <= SIDEB;
            end else if (rdusedw1 >= IMG_WIDTH * 2 + 2) begin
                cnt <= 'd0;
                state <= SIDEA;
            end else begin
                cnt <= 'd0;
                state <= IDLE;
            end
        end
    default :   state <= IDLE;
endcase
```

当然，这里我们只实现了状态的跳转，使能信号的转变以及数据的数据还未实现，对应代码如下：

```
always @ (*) begin
case (state)
    IDLE:
        begin
            rdreq1 = 1'b0;
            rdreq2 = 1'b0;
            fifo_wrreq = 1'b0;
            fifo_wrdata = 'd0;
        end
    SIDEA:
        begin
            rdreq1 = 1'b1;
            rdreq2 = 1'b0;
            fifo_wrreq = 1'b1;
            fifo_wrdata = rddata1;
        end
    SIDEB:
        begin
            rdreq1 = 1'b0;
            rdreq2 = 1'b1;
            fifo_wrreq = 1'b1;
            fifo_wrdata = rddata2;
        end
    default:
        begin
            rdreq1 = 1'b0;
            rdreq2 = 1'b0;
            fifo_wrreq = 1'b0;
            fifo_wrdata = 'd0;
```

```
        end
endcase
```

在空闲态 IDLE 时，不进行任何数据处理工作，所以该部分相应信号全部为高电平，输出数据为 0。

在 SIDEA 状态时，系统需要读取 cache1 中的数据并输出，所以相应 cache1 的读请求信号变为高电平，controller 模块的输出请求信号为高电平，输出数据为 cache1 中的数据。

在 SIDEB 状态时，系统读取 cache2 中的数据将行号去除掉后输出，所以相应 cache2 的读请求信号变为高电平，controller 模块的输出请求信号为高电平，输出数据为 cache2 中的数据。

至此，该模块的设计便已完成，接下来即可对其进行仿真验证，以确定其功能是否能够正常运行。

57.5仿真验证

1. 在进行仿真验证时，我们可以对数据输入和计数器部分作如下设计：
2. 设计两个 8 位的数 wrdata1 和 wrdata2 分别作为 cache1 和 cache2 的输入数据。当 wrreq1 有效时，wrdata1 自加，当 wrreq2 有效时，wrdata2 自加。数据超出 8 位后便会溢出清零。
3. 设计两个 4 位的计数器，cnt1 和 cnt2，分别 wrdata1 和 wrdata2 的溢出计数器。每当 wrdata1 和 wrdata2 溢出时，相应计数器的值便会加 1。当 wrdata1 和 wrdata2 的写使能信号 wrreq1 和 wrreq2 为低电平时，对应计数器便会被清零。
4. 完成了输入设计后接下来便是激励的产生，该部分设计如下：
5. 对系统进行复位
6. 让 wrreq1 和 wrreq2 变为高电平，此时数据开始写入 cache1 和 cache2
7. 延时一段时间后将 wrreq1 信号变为低电平，此时 cache1 中数据停止写入
8. 延时一段时间后将 wrreq2 信号变为低电平，wrreq1 信号变为高电平。此时 cache2 停止写入数据，cache1 开始写入数据
9. 延时一段时间后结束仿真，观察波形

本次仿真验证的代码如下：

```
`timescale 1ns/1ps

module controller_top_tb;

    reg clk;
    reg rst_n;
    wire wrclk1;
    reg wrreq1;
    reg [7:0]wrdata1;

    reg [7:0]wrdata2;
    reg wrreq2;
    reg fifo1_aclr;
    reg fifo2_aclr;
    wire wrclk2;

    wire wrclk;
    wire fifo_aclr;
    wire fifo_wrreq;
    wire [7:0] fifo_wrdata;

    assign wrclk1 = clk;
    assign wrclk2 = clk;

    initial clk = 1;
    always#10 clk = ~clk;

    controller_top controller_top(
        .clk(clk),
        .rst_n(rst_n),

        .wrclk1(clk),
        .fifo1_aclr(fifo1_aclr),
        .wrdata1(wrdata1),
        .wrreq1(wrreq1),

        .wrclk2(clk),
        .fifo2_aclr(fifo2_aclr),
        .wrdata2(wrdata2),
        .wrreq2(wrreq2),

        .wrclk(wrclk),
        .fifo_aclr(fifo_aclr),
        .fifo_wrreq(fifo_wrreq),
```

```
        .fifo_wrdata(fifo_wrdata)
    );

    initial begin
        rst_n = 0;
        wrreq1 = 0;
        wrreq2 = 0;
        fifo1_aclr = 1;
        fifo2_aclr = 1;
        #201;
        rst_n = 1;
        fifo1_aclr = 0;
        #3;
        fifo2_aclr = 0;
        #201;
        wrreq1 = 1;
        wrreq2 = 1;
        #30000;
        wrreq1 = 0;
        wrreq2 = 1;
        #30000;
        wrreq1 = 1;
        wrreq2 = 0;
        #30000;
        $stop;
    end

    always@(posedge clk)
    if(!rst_n)
        wrdata1 <=  0;
    else if(wrreq1)
        wrdata1 <= wrdata1 + 1'b1;
    else
        wrdata1 <= 0;

    always@(posedge clk)
    if(!rst_n)
        wrdata2 <=  0;
    else if(wrreq2)
        wrdata2 <= wrdata2 + 1'b1;
    else
        wrdata2 <= 0;

endmodule
```

设计完了激励文件后接下来即可运行仿真，分析波形数据了。

57.5.1仿真波形分析

本次仿真文件为 control_top_tb，仿真得到的波形如图 57-5 所示。

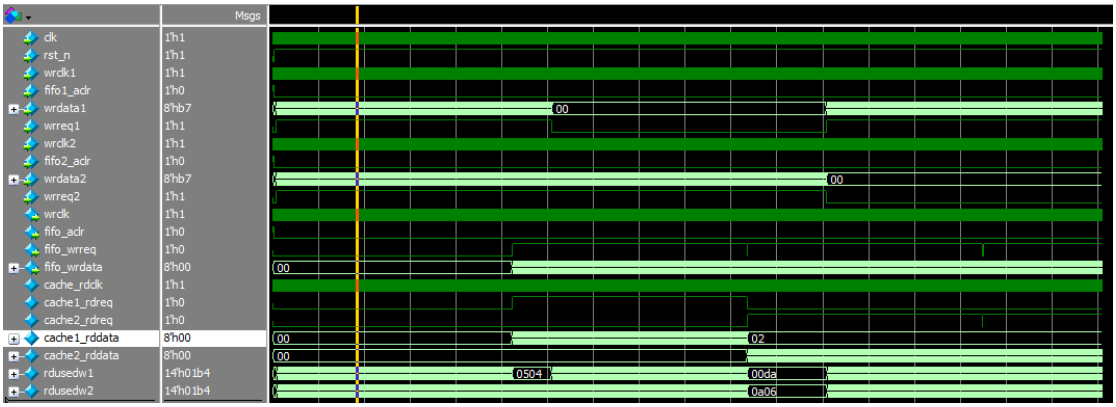


图 57-5 仿真波形图

因为这次仿真涉及的信号较多，所以后面我们将一一需要验证的信号进行梳理，检验本次设计的正确性。

57.5.1.1输入数据

我们对本次仿真的波形图放大，观察输入数据其图像如图 57-6 所示。

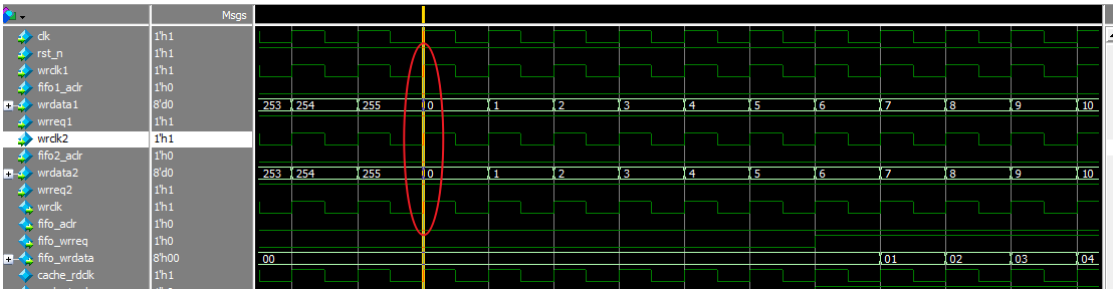


图 57-6 输入数据波形图

从图中我们可以看到 wrreq1 和 wrreq2 有效时，wrdata1 和 wrdata2 自加，当计数值达到 16’hff（即十进制 255）时，输入数据溢出清零。该部分波形结果与设计预想的一致，证明输入数据的逻辑设计正确。

57.5.1.2dcfifo 读出数据量及计数器 cnt

根据功能设计我们知道，当 cache1 的 rdusedw1 信号值达到 1282 时，其读请求信号 cache1_rdreq 有效，controller 模块便会读取 cache1 中 1282 个数据。随

后判断 cache2 的 rdusedw2 信号值是否达到 1282，如果达到便会读取 cache2 中 1282 个数据。仿真中对 cache2 和 cache2 数据的读取如图 57-7 中①②两个红框所示。

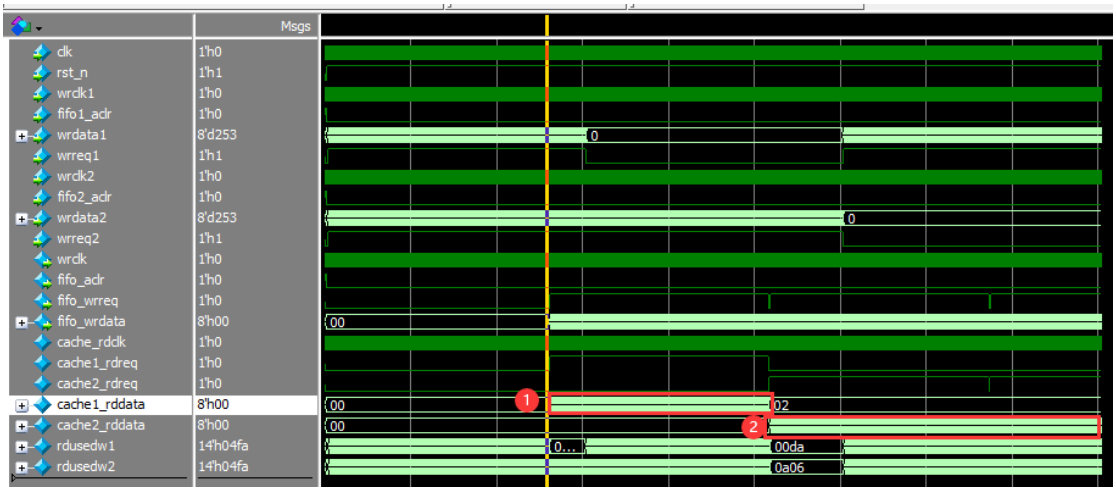


图 57-7 数据读取波形

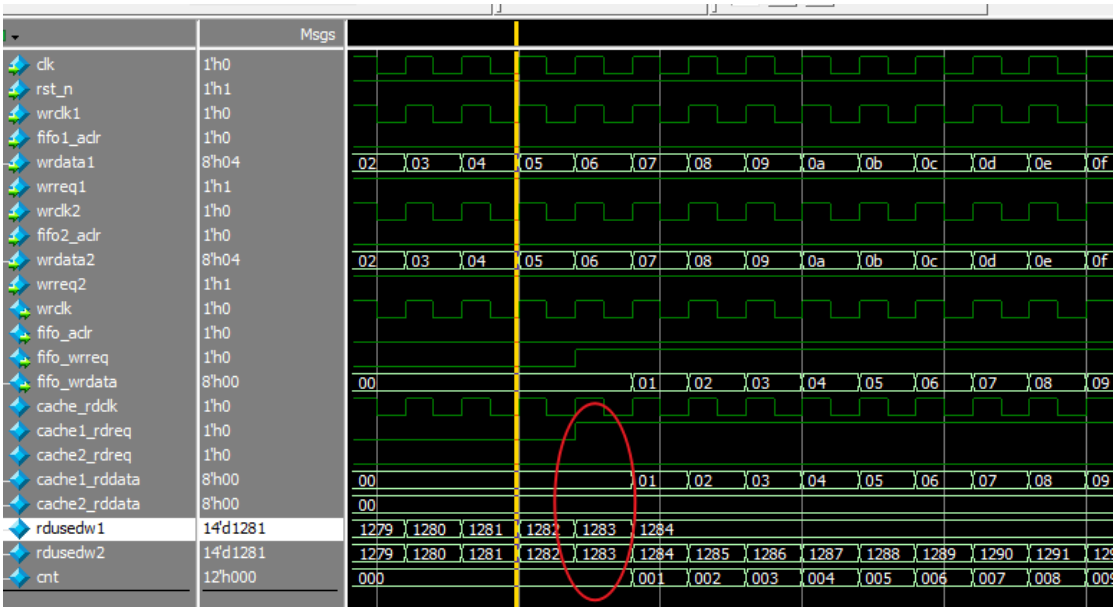


图 57-8 开始数据读取时使能信号与计数器变化

可以看到图 57-8 中 cache1_rdreq 是在 rdusedw1 值为 1283 时才开始跳变为上升沿，出现该现象并不是说明我们设计出现了错误。在上个时钟上升沿时，rdusedw1 的值已经变为了 1282，对 rdusedw1 值的判断语句则是在当前时钟上升沿执行，而此时 rdusedw1 的值已经被更新为 1283，因此该点对应的值正确。此时，cache1 中的数据开始读出。

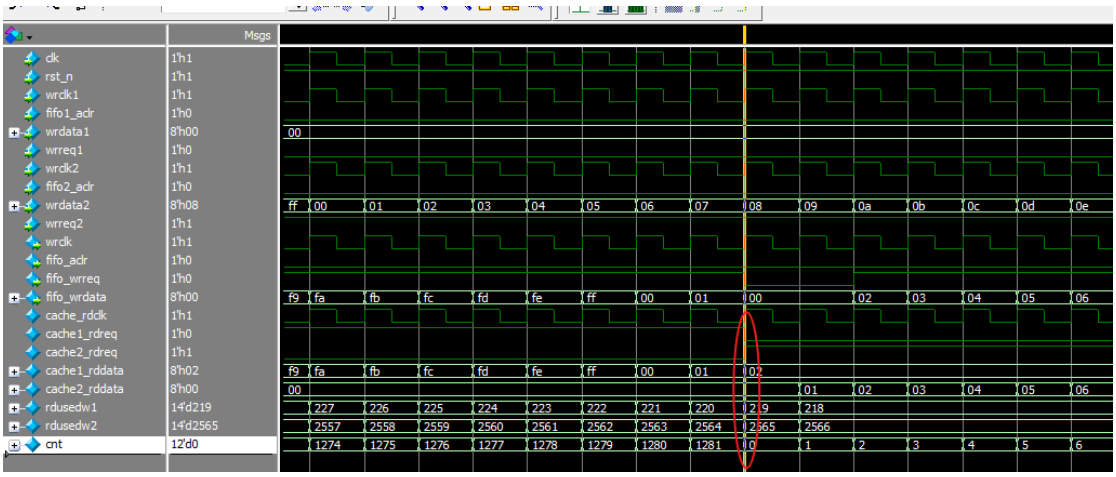


图 57-9 结束数据读取时使能信号与计数器变化

可以看到图 57-9 中 cnt 的值大于 1281 时，cache1_rdreq 变为下降沿，此时停止从 cache1 中读取数据。

57.5.1.3输出使能信号及输出数据

因为当 cache1 中的数据读取到 controller 模块时，输出使能信号 fifo_wrreq 信号便会由低电平跳变为高电平。当 controller 模块停止对 cache 数据读取时，输出使能信号 fifo_wrreq 信号便会由高电平跳变为低电平。所以，当读 cache1 的数据时，即 cache1_rdreq 信号变为高电平时，fifo_wrreq 为高电平；而在读 cache2 的数据时，会舍弃掉前两个数据，controller 在读取数据时一个时钟上升沿只会读取一个数据，所以 cache2_rdreq 信号为高电平两个时钟上升沿后，fifo_wrreq 才会变为高电平。

图 57-10 为 controller 模块输出一行拼接的图像数据，输出使能信号和输出数据与 cache1_rdreq、cache2_rdreq 的关系。将 controller 模块开始输出 cache1 数据时的图像放大，得到图 57-11。

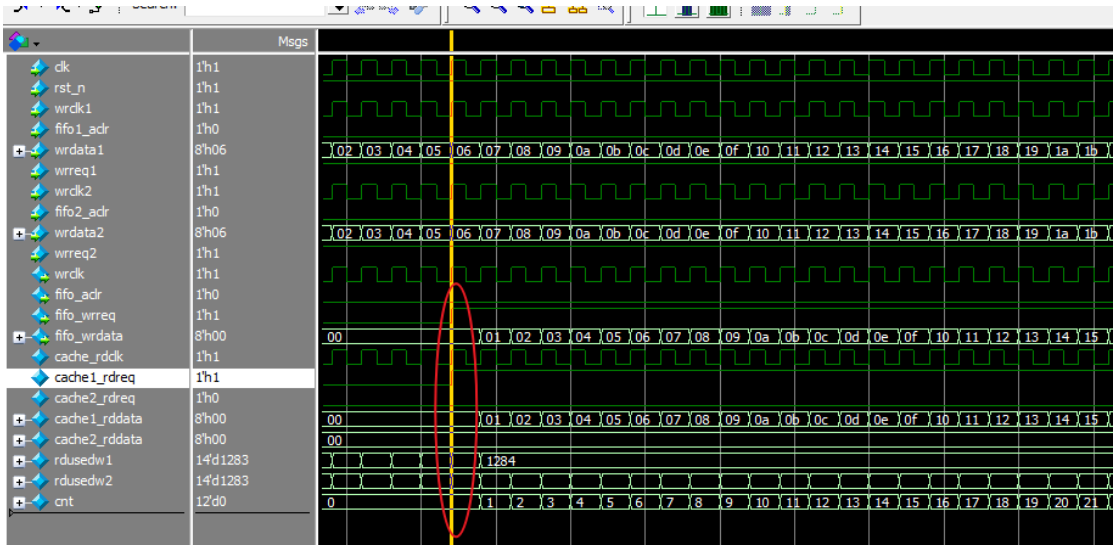


图 57-10 数据输出仿真波形图

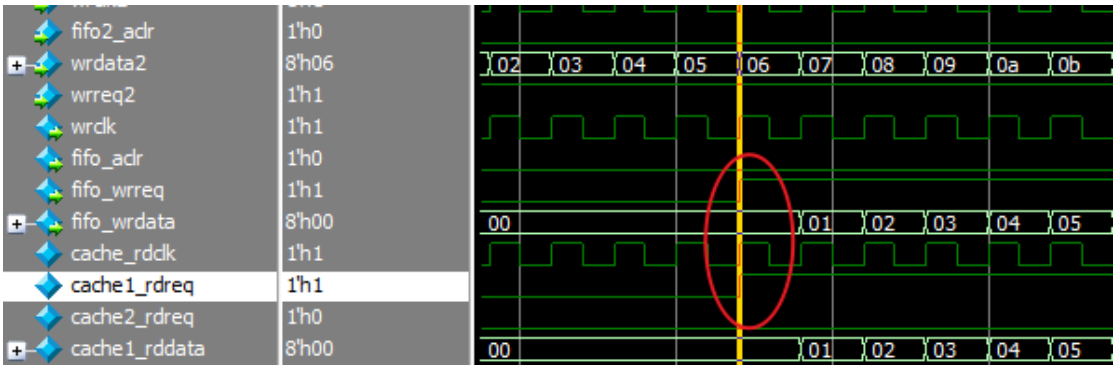


图 57-11 数据输出开始时相应信号变化

可以看到图 57-11 中，当 cache1_rdreq 转变为高电平时，fifo_wrreq 信号也转变为高电平，此时 controller 模块的输出使能有效，数据开始输出，证明该部分设计正确。

将 controller 开始输出 cache2 数据时的图像放大，得到图 57-12。

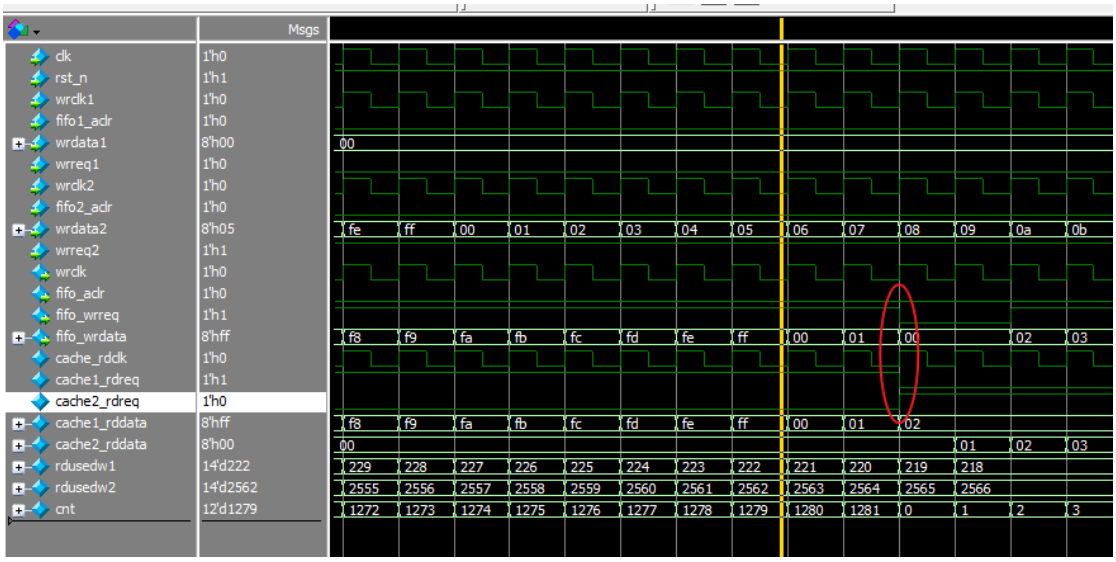


图 57-12 数据输出结束时相应信号变化

可以看到图 57-12 中，当 cache1_rdreq 变为低电平时，control 模块停止读取 cache1 中的数据，fifo_wrreq 信号转变为低电平。此时 cache2_rdreq 转变为高电平，fifo_wrreq 信号在经过两个时钟上升沿后跳变为上升沿，该部分与理论结果一致，证明该部分设计正确。

综上所诉，输出使能信号和输出数据部分设计正确。

通过对上述信号分析比对，我们对本次模块设计的功能进行了验证，确定了本次设计的正确性。考虑到仿真结果具有参考价值，接下来我们还需要为设计分配引脚以进行板级验证。

57.6引脚约束

本次设计引脚分配表如所示。

表 57-2 引脚分配表

Signial Name	Pin NO.		Signial Name	Pin NO.
camera1_data[7]	F18		camera2_data[7]	L18
camera1_data[6]	F17		camera2_data[6]	L17
camera1_data[5]	G18		camera2_data[5]	K18
camera1_data[4]	G16		camera2_data[4]	K17
camera1_data[3]	D18		camera2_data[3]	H18
camera1_data[2]	D17		camera2_data[2]	H17
camera1_data[1]	G14		camera2_data[1]	L13
camera1_data[0]	F14		camera2_data[0]	L12
camera1_rst_n	C18		camera2_rst_n	K13
camera1_pwdn	C17		camera2_pwdn	K12
camera1_xclk	H13		camera2_xclk	N17
camera1_pclk	H14		camera2_pclk	N18

camera1_href	K15		camera2_href	P17
camera1_vsync	K16		camera2_vsync	P18
camera1_sdat	J13		camera2_sdat	P15
camera1_sclk	K14		camera2_sclk	P16
eth_txd[3]	F1		eth_mdio	E4
eth_txd[2]	F2		eth_rst_n	G6
eth_txd[1]	D1		eth_txen	C2
eth_txd[0]	D2		clk	T9
eth_gtxc	C1		rst_n	C15
eth_mdc	D3			

分配完成后，随后生成数据流，等待生成完成后便可以连接硬件准备板级验证了。

57.7板级验证

因为高云开发板本身自带有 40pin 的拓展接口和一个千兆网口接口，满足本次系统设计的需求。所以本次板级验证将使用高云开发板来进行本项目的板级验证。

57.7.1系统所需硬件

本次设计所需硬件如下：

1. 高云开发板 x1
2. 双目摄像头 x1
3. DC 电源线 x1
4. 高云下载器 x1
5. 千兆网线 x1

57.7.2硬件连接

本次系统设计硬件方面连接如

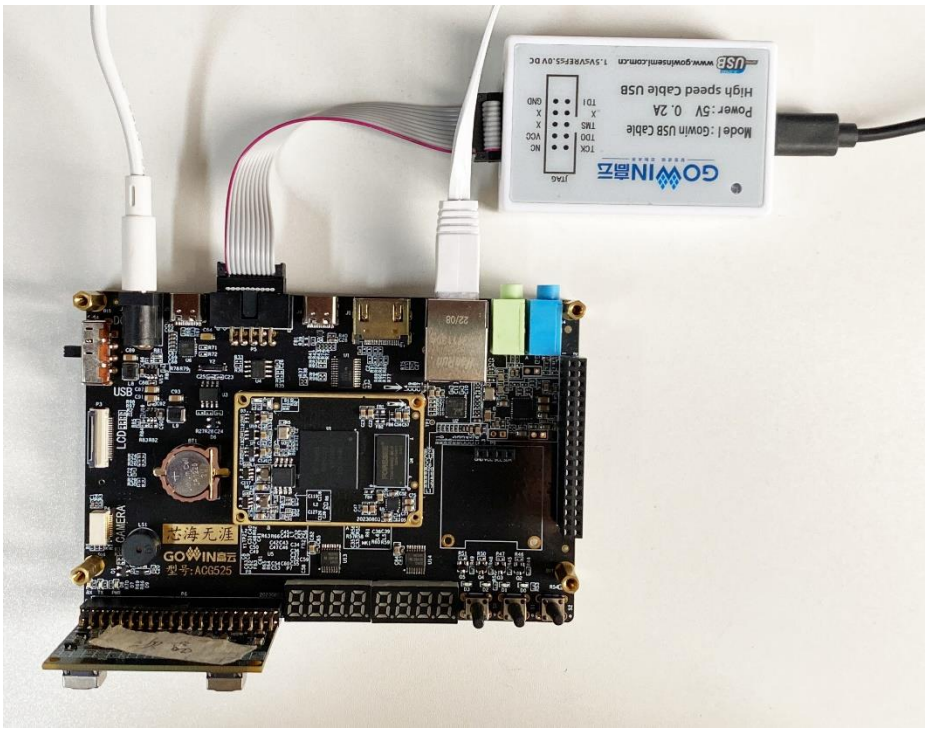


图 57-13 所示。

1. 将双目摄像头与开发板 40pin 扩展接口相连（如果是用的双目转接板，则是将转接板在开发板 40pin 扩展接口上，然后将 OV5640 接在转接板上）。
2. 用网线将开发板网口和电脑网口连接。
3. 连接好下载器。
4. 连接好电源，电源线最好连接电源适配器。

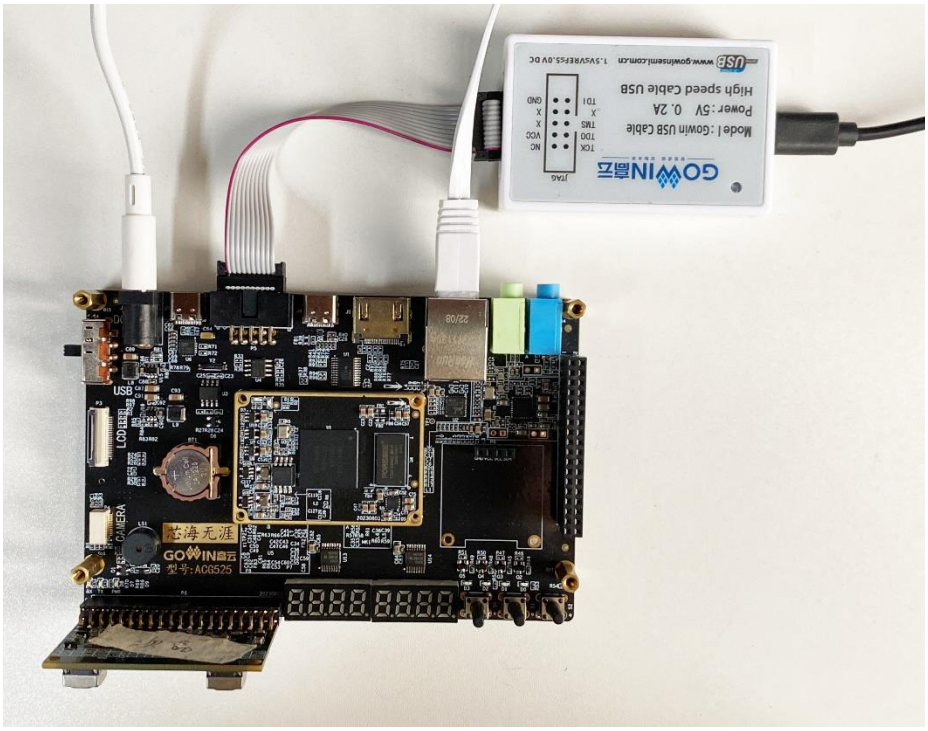


图 57-13 硬件连接

连接完毕后接下来即可进行程序下载和验证了。

57.7.3下载与验证

本次板级验证的工程名为 ov5640x2_udp_rgmii，连接好硬件后对应的下载与验证操作如下：

1. 将开发板左侧的电源拨码开关拨到 ON 侧。
2. 将生成的数据流文件下载至开发板中，注意这一步一定要先于下面的步骤执行。
3. 在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态，如图 57-14 所示。当看到在活动网络中有本地连接存在时，表明开发板和电脑的网络已经连通。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意。

图 57-14 网路连接状态

4. 点击“本地连接”文字，查看该网络状态，确认当前连接速度为千兆速率（1000.0Mbps），如图 57-15 所示。

图 57-15 以太网速率

在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4（TCP/Ipv4）】选项，然后在弹出的属性对话框中 设置静态

IP 地址，如图 57-16 所示。

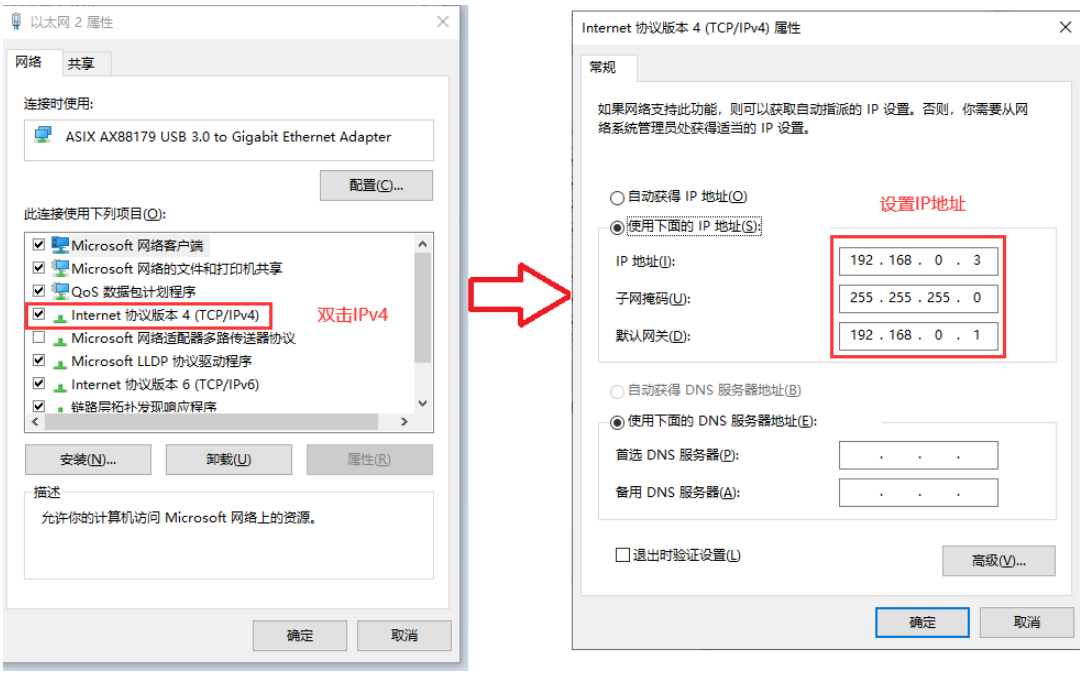


图 57-16 静态 IP 设置

5. 开启电脑巨型帧。在“设备管理器\网络适配器\Realtek PCIe GbE Family Controller 属性\高级\巨型帧”，如图 57-17 所示。

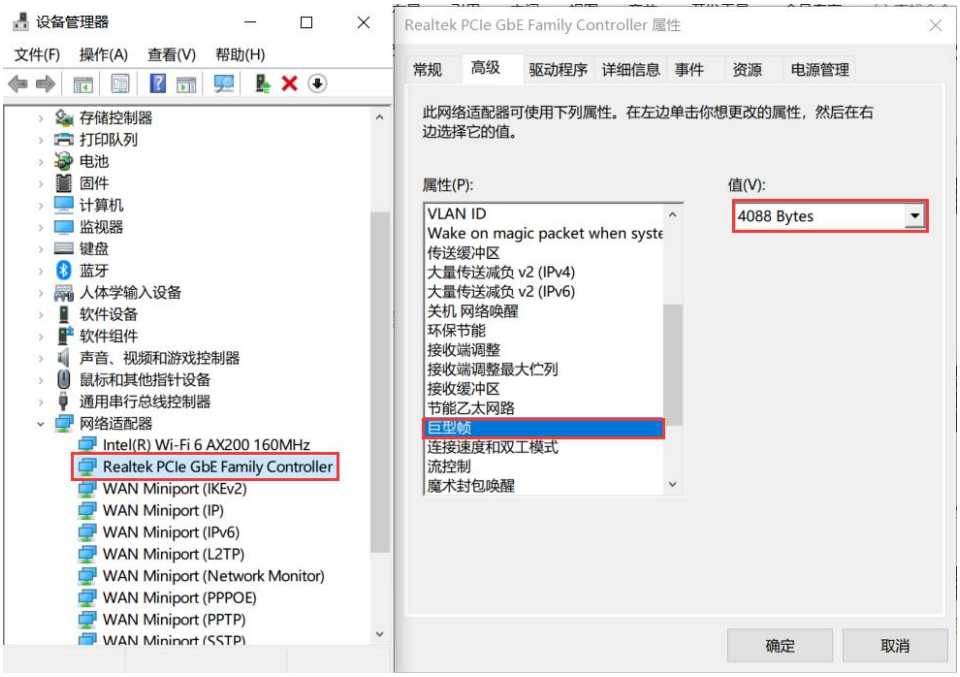


图 57-17 巨型帧设置

6. 由于本测试工程不支持 ARP 协议，因此我们还需要通过 netsh 方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给

192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

操作时先以管理员身份运行 cmd.exe 程序（该文件在 C:\Windows\System32 路径下），也就是我们常说的命令行窗口。由于有用户反应在使用时无法成功绑定 arp，经过分析就是操作权限不够，所以这里强调要以管理员身份运行 cmd.exe。

打开 cmd 窗口之后，依次执行如下命令：

（1）查看电脑网卡名称，有线网卡的大概率为“以太网”

```
netsh interface IPV4 show interface
```

图 57-18 查看电脑的网卡名称

（2）根据自己的网卡名称，笔者电脑网卡名称为“以太网”，用下述命令执行静态绑定，如果网卡名称不同，需要自行修改命令然后执行。

```
netsh -c interface ipv4 add neighbors "以太网" "192.168.0.2" "00-0a-35-01-fe-c0"
```

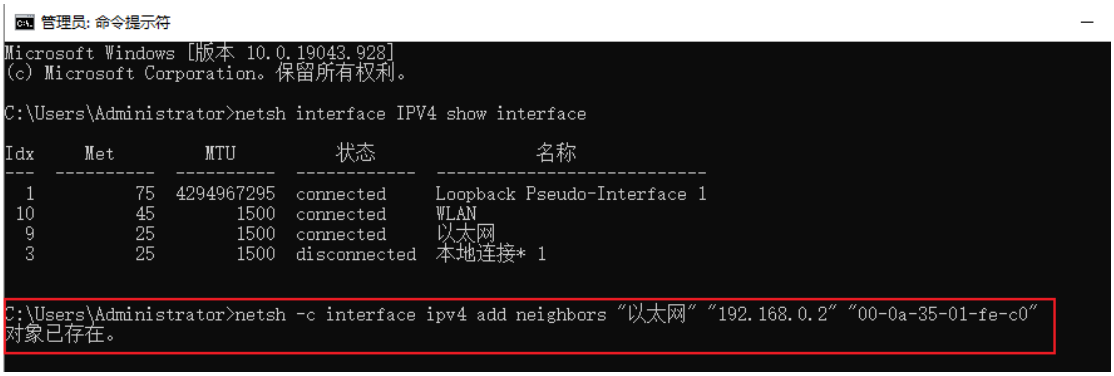


图 57-19 静态绑定

（3）最后用查看绑定结果，执行如下命令。

```
arp -a
```

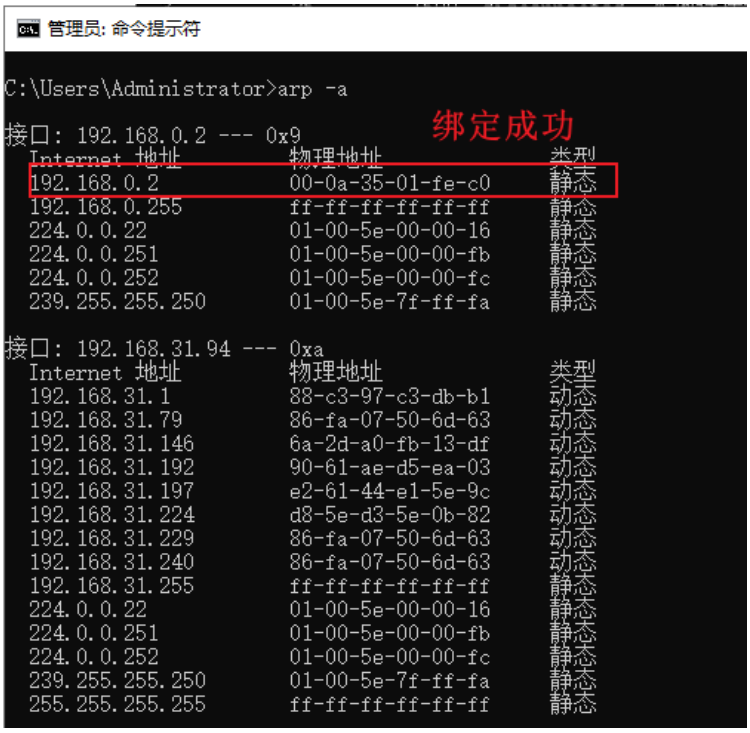


图 57-20 查看绑定结果

7. 打开小梅哥 UDP 摄像头上位机，并按照如图 57-21 所述设置各项参数。

图 57-21 小梅哥 UDP 摄像头界面

- 本机 IP，对应当前实验所用电脑的 IP 地址，前面已经按照要求设置为

了 192.168.0.3，所以这里直接填写该地址。

- 本机端口，对应当前实验所用电脑接收图像时使用的网络端口，该端口在 FPGA 程序中定义为了 6102，所以这里也要设置为 6102 才能正确显示图像。
- 远程 IP，对应 FPGA 开发板使用的 IP 地址，也就是代码中的 src_ip 值。例程使用的为 192.168.0.2。
- 远程端口，对应 FPGA 开发板使用的端口号，也就是代码中的 src_port 值，例程使用的为 5000。
- 分辨率，对应了摄像头输出的图像分辨率大小，本实验中，每个摄像头输出的图像分辨率为 640*720，因此经过数据拼接后的图像分辨率为 1280*720。
- 保存图片，该按钮在软件停止接收图像后，点击可以保存最后界面上显示的图像内容到文件。
- 连接/停止，该按钮可以开始和停止接收并显示图像，当所有参数设置好之后，点击该按钮即可开始接收并显示图像。

8. 设置好参数后，点击链接按钮即可开始接收并显示图像内容。
9. 该软件运行时需要关闭防火墙，软件自带关闭防火墙功能，点击连接时会提示是否关闭防火墙，勾选专用网络和公共网络两个选项，然后点击【允许访问按钮】即可。如果还是无法关闭，考虑软件权限不够，以管理员身份运行本软件即可，如图 57-22 所示。

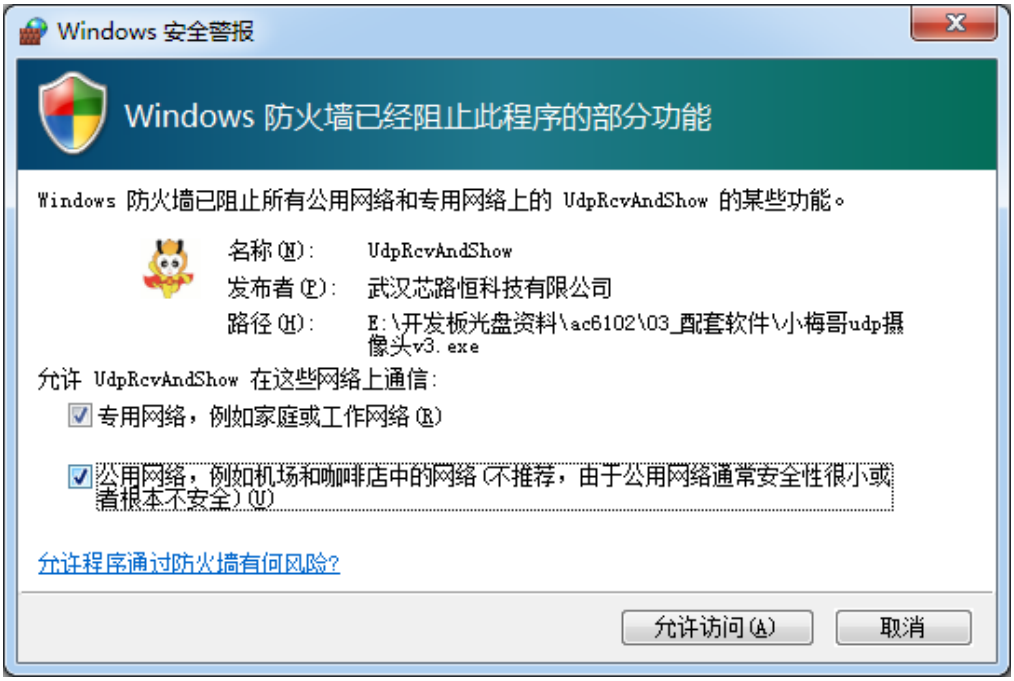


图 57-22 关闭防火墙

连接成功后，可以看到软件显示图像如图 57-23 所示

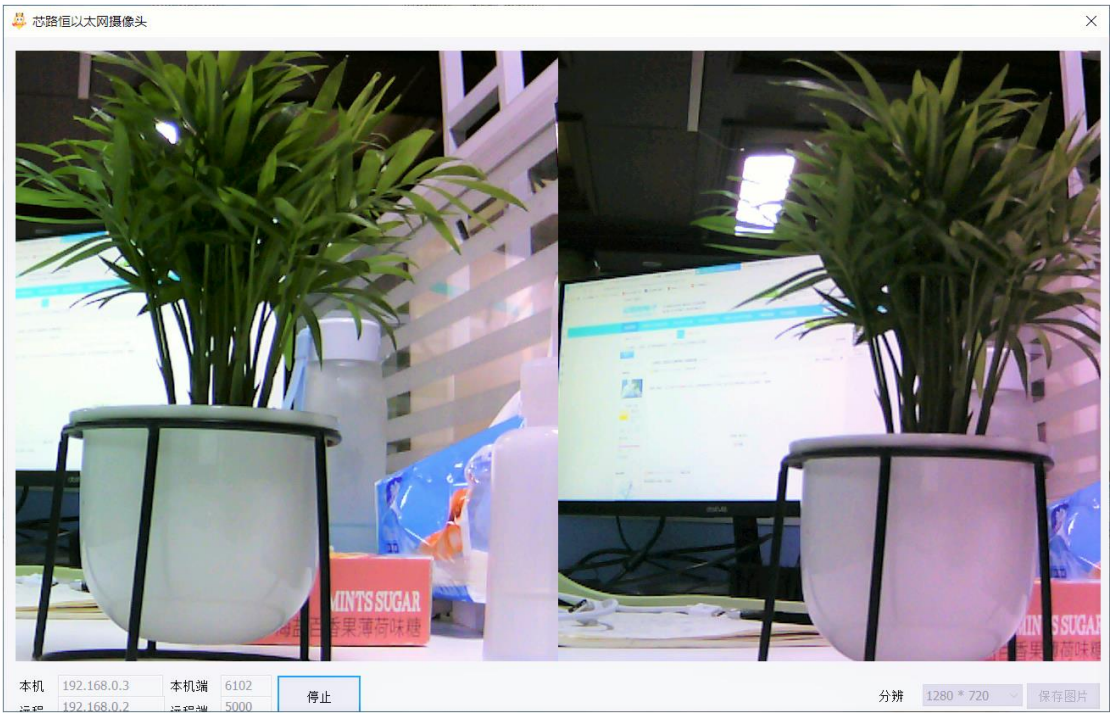


图 57-23 采集显示图像

可以看到显示出来的画面色彩鲜艳亮丽，图像整体饱和度高，图像比例正常，没有任何压缩错位现象，本次双目摄像头数据拼接模块的设计成功。

57.8常见问题说明

对于本节工程需要注意的有以下两点。

1. 在进行以太网传输前，需要确认电脑是否支持千兆传输，同事还需打开巨型帧，否则数据将无法传输显示。
2. 特别需要注意：如果是通过其他工程而来的摄像头工程，要确认摄像头初始化管脚的片上上拉电阻（2 组 SCCB 信号）已经开启。否则摄像头有可能不会初始化成功，进而不会有图像输出。

57.8.1总结与思考

本节设计实现了一个数据拼接模块，通过对两个摄像头采集的数据分开处理并传输，实现了对双目摄像头图像数据的拼接显示。学习本节时，读者可以参考相应的视频教程相互理解印证，建议读者能够跟随本实验内容，完整的进行整个实验。

58 千兆以太网传图接收 TFT 显示系统设计

工程源码	|----02_设计实例 |----ch58_eth_udp_rgmii_ddr3_tft800x480
相关视频课程	
说明	

章节导读

本节实验通过以太网传图工具将图像数据传输到高云开发板上的 DDR3 进行存储，最终图像数据由 TFT 控制器主动从 DDR3 中读取出来，送往 TFT 显示屏进行显示。

58.1系统整体设计

系统的整体设计如下图 58-1 所示。

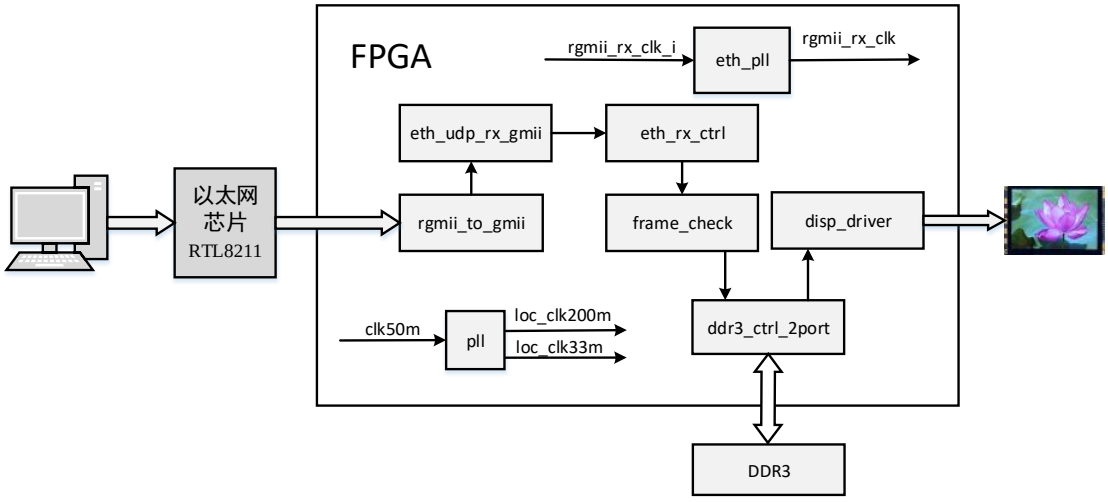


图 58-1 千兆以太网传图 TFT 显示系统整体设计框图

下面对上述图中的各个模块进行简要介绍：

1. pll 模块：锁相环模块，生成本次实验每个模块所需要的工作时钟，输入时钟 50M，由开发板上的晶振提供；输出 200M 的时钟给到 DDR3 控制器使用，输出 33M 的时钟给到 TFT 控制器（disp_driver）使用。
2. eth_pll 模块：锁相环模块，将 rgmii 接口时钟信号 rgmii_rx_clk_i 偏移 270°得到 rgmii_rx_clk 时钟信号，这样做是为了在时钟上升沿/下降沿取数据时，取得的数据是最准确和稳定的，具体的偏移角度是实际使用时，不断调试得到的，在我们的高云的开发板上，偏移 270°是最合适的。

3. rgmii_to_gmii 模块：以太网接口转换模块，将 rgmii 接口转换成 gmii 接口。
4. eth_udp_rx_gmii 模块：以太网接收模块，接收上位机传输过来的图像数据，详细设计请参看前面以太网相关章节的内容。
5. eth_rx_ctrl 模块：以太网接收控制模块，将以太网接收的报文数据存入 FIFO 中。
6. frame_check 模块：判断每个包数据的序号是否连续，如果不连续将整帧图像数据丢弃。
7. disp_driver 模块：产生符合 TFT 显示屏的驱动时序，并从 DDR3 控制器中取出图像数据送到 TFT 显示屏，详细设计请参看“TFT 显示驱动设计与验证”一节的内容。
8. ddr3_ctrl_2port：二端口 DDR3 控制器设计，详细设计请参看“二端口 DDR3 控制器（ddr3_ctrl_2port）设计”一节的内容。

本章需要设计的模块包括 eth_rx_ctrl 模块和 frame_check 模块。

58.2模块设计

58.2.1以太网接收控制模块

本节将介绍以太网控制模块的代码设计及其仿真分析。

58.2.1.1以太网控制模块设计

以太网接收控制模块，主要是将以太网接收的报文数据存入 FIFO 中，其模块的结构框图如下图 58-2 所示。从图中可以看出输入数据为 8 位，输出数据为 16 位，这样做的目的是因为以太网接收模块输出数据是 8 位的，而传输过来的图像数据是 16 位的，所以这里需要将两个连续的 8 位数据还原成 16 位的图像数据。

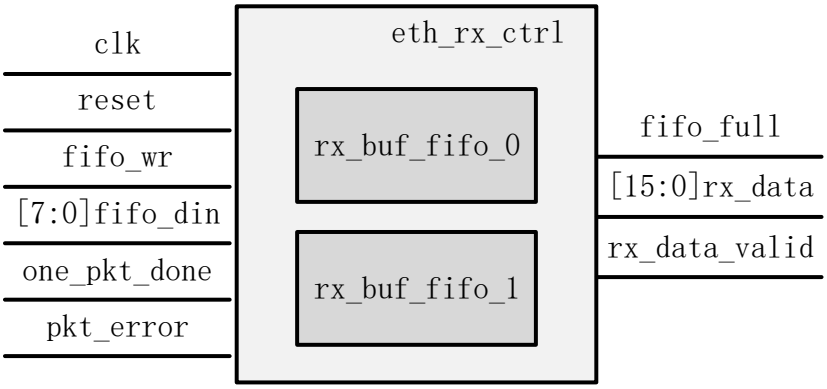


图 58-2 以太网接收控制框图

对于每个信号说明如下表 58-1 所示。

表 58-1 以太网接收控制模块说明表

接口名称	I/O	信号意义
clk	I	模块工作时钟信号
reset	I	模块复位信号，高有效
fifo_wr	I	FIFO 的写使能信号
fifo_din[7:0]	I	需要向 FIFO 中写入的 8 位数据
one_pkt_done	I	以太网单包数据接收完成标志信号
pkt_error	I	以太网包接收错误标志信号
fifo_full	I	FIFO 写满标志信号
rx_data[15:0]	O	从 FIFO 读出的 16 位的数据信号
rx_data_valid	O	FIFO 读出数据有效信号

在该模块中，使用了两个 FIFO IP 模块存储以太网接收过来的数据。这里使用两个 FIFO 实现双缓存的目的是在 CRC 校验出现问题时把这包数据丢掉。以太网在传输数据的过程中只有一包的数据接收完成之后才能判断数据是否出现错误，出现问题之后再清空 FIFO 要保证不能影响下一包数据的接收，当只有一个 FIFO 进行缓存时，有可能存在当前一包数据接收完正在判断是否丢弃过程中，下一包数据正在往 FIFO 里写，如果判断当前包有问题清空过程中会把下一包部分数据清掉，这样下一包数据就不完整了，而双缓存可以避免这个问题。下面将介绍该模块中部分代码的实现。

FIFO 的写使能信号的产生如下所示：

```
assign fifo0_wr_en = fifo_wr && (wrfifo_sel == 1'b0);
assign fifo1_wr_en = fifo_wr && (wrfifo_sel == 1'b1);

always@(posedge clk or posedge reset)
begin
  if(reset)
    wrfifo_sel <= 1'b0;
```

```
  else if(one_pkt_done)
    wrfifo_sel <= ~wrfifo_sel;
  else
    wrfifo_sel <= wrfifo_sel;
end
```

上述代码中，可以看到当 wrfifo_sel 为 0 并且 fifo_wr 有效时，向 FIFO0 中写入数据，当 wrfifo_sel 为 1 并且 fifo_wr 有效时，向 FIFO1 中写入数据。当产生以太网包传输完成标志信号 one_pkt_done 之后，对 wrfifo_sel 信号进行取反，也就是一个包传输完成之后，就切换一个 FIFO 进行数据的存储。本次实验中 fifo_wr 信号连接的是以太网接收模块的 payload_valid_o 信号，也就是当以太网开始接收数据之后，fifo_wr 就有效。

FIFO 的清除代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo0_srst <= 1'b1;
  else if(one_pkt_done && pkt_error && (wrfifo_sel == 1'b0))
    fifo0_srst <= 1'b1;
  else
    fifo0_srst <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo1_srst <= 1'b1;
  else if(one_pkt_done && pkt_error && (wrfifo_sel == 1'b1))
    fifo1_srst <= 1'b1;
  else
    fifo1_srst <= 1'b0;
end
```

从上述代码中可以看出，当产生一个包接收完成标志信号 one_pkt_done、接收数据出现错误标志信号 pkt_error，wrfifo_sel 信号为 0 的时候，fifo0_srst 为 1，清除 FIFO0；wrfifo_sel 信号为 1 的时候，fifo1_srst 为 1，清除 FIFO1。只有数据没有出现错误，对应的 FIFO 复位信号 fifo0_srst 和 fifo1_srst 为 0，保留 FIFO 中的数据。

FIFO 的读使能信号如下所示：

```
always@(posedge clk or posedge reset)
begin
```

```
    if(reset)
        fifo0_rd_en <= 1'b0;
    else if((fifo0_rd_data_cnt > 11'd2) && (rdfifo_sel == 1'b0))
        fifo0_rd_en <= 1'b1;
    else
        fifo0_rd_en <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
    if(reset)
        fifo1_rd_en <= 1'b0;
    else if((fifo1_rd_data_cnt > 11'd2) && (rdfifo_sel == 1'b1))
        fifo1_rd_en <= 1'b1;
    else
        fifo1_rd_en <= 1'b0;
end
```

当 FIFO 中的可读数据大于 2 时，代表 FIFO 中的数据还未读完，也就是当 ifo0_rd_data_cnt > 1'b2 或者 fifo1_rd_data_cnt > 1'b2 时，代表此时 FIFO 中还有未读出的数据，FIFO 中的数据是有效的，此时若 rdfifo_sel 为 0 则读取 FIFO0 中的数据，若 rdfifo_sel 为 1 则读取 FIFO1 中的数据。

产生对应 FIFO 的读使能之后，则将对应的 FIFO 中的数据读取出来，并产生数据有效信号 rx_data_valid，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    rx_data <= 'd0;
  else if(fifo0_rd_en_dly1)
    rx_data <= fifo0_dout;
  else if(fifo1_rd_en_dly1)
    rx_data <= fifo1_dout;
  else
    rx_data <= 'd0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    rx_data_valid <= 1'b0;
  else if(fifo0_rd_en_dly1)
    rx_data_valid <= 1'b1;
```

```
  else if(fifo1_rd_en_dly1)
    rx_data_valid <= 1'b1;
  else
    rx_data_valid <= 1'b0;
end
```

58.2.1.2以太网控制模块仿真

以太网控制模块设计完成之后，需要通过仿真对其进行测试，通过仿真波形，查看是否实现功能：每个包传输过来的时候，会交替写入到两个 FIFO 中，当数据是正确的时，将数据输出，数据出现错误时，将数据清除不输出。如果满足该功能，则认为仿真正确。

首先在 tb 文件中编写两个 task 事件，分别代表写入的是正确的数据和错误的数据，写入错误数据时设置 pkt_error 为高电平，最终通过仿真查看最终输出的数据是否为正确数据，两个 task 事件代码如下所示：

```
task write_data;
    input [7:0]data_begin;
    input [10:0]data_cnt;
    begin
    fifo_wr = 0;
    fifo_din = data_begin;
    one_pkt_done = 0;
    pkt_error = 0;

    @(posedge clk);
    #1 fifo_wr = 1;
    repeat(data_cnt) begin
        @(posedge clk);
        #1 fifo_din = fifo_din + 1;
    end
    fifo_wr = 0;
    @(posedge clk);
    #1 one_pkt_done = 1;
    @(posedge clk);
    #1 one_pkt_done = 0;
    end
endtask

task write_data_error;
    input [7:0]data_begin;
    input [10:0]data_cnt;
```

```
    begin
    fifo_wr = 0;
    fifo_din = data_begin;
    one_pkt_done = 0;
    pkt_error = 0;

    @(posedge clk);
    #1 fifo_wr = 1;
    repeat(data_cnt) begin
        @(posedge clk);
        #1 fifo_din = fifo_din + 1;
    end
    fifo_wr = 0;
    @(posedge clk);#1;
    one_pkt_done = 1;
    pkt_error = 1;
    @(posedge clk);#1;
    one_pkt_done = 0;
    pkt_error = 0;
    end
endtask
```

然后再 tb 文件中例化 eth_rx_ctrl 模块，并且写入正确或错误的数据，代码如下所示：

```
eth_rx_ctrl
#(
    .O_DATA_WIDTH (16  )
)eth_rx_ctrl
(
    .clk          (clk          ),
    .reset        (~reset_n      ),

    .fifo_full    (fifo_full    ),
    .fifo_wr      (fifo_wr      ),
    .fifo_din     (fifo_din     ),

    .one_pkt_done (one_pkt_done ),
    .pkt_error    (pkt_error    ),

    .rx_data      (rx_data      ),
    .rx_data_valid(rx_data_valid)
);

initial begin
```

```
    reset_n = 1'b0;
    fifo_wr = 0;
    fifo_din = 0;
    one_pkt_done = 0;
    pkt_error = 0;
    #201;
    reset_n = 1'b1;
    #200;

    write_data(0,50);
    #2000;
    write_data(100,50);
    #2000;
    write_data_error(200,50);
    #2000;
    write_data(300,50);
    #2000;
    write_data_error(400,50);
    #2000;
    write_data(500,50);
    #2000;
    $stop;
end
```

完整 tb 文件请自行查看工程中的源文件（eth_udp_rgmii_ddr3_tft800x480\src\eth_rx_ctrl），下面对仿真波形进行分析。

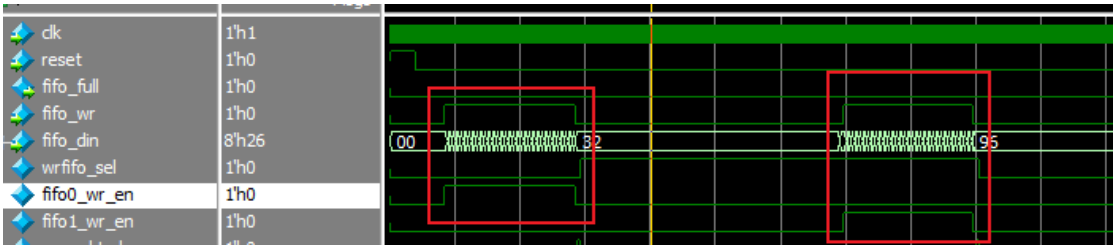


图 58-3 交互写入两个 FIFO 中

前两个包的数据是正确的，所以会产生对应的读使能信号，将得到的数据进行输出，并产生数据有效信号，波形图如下图 58-4 所示。

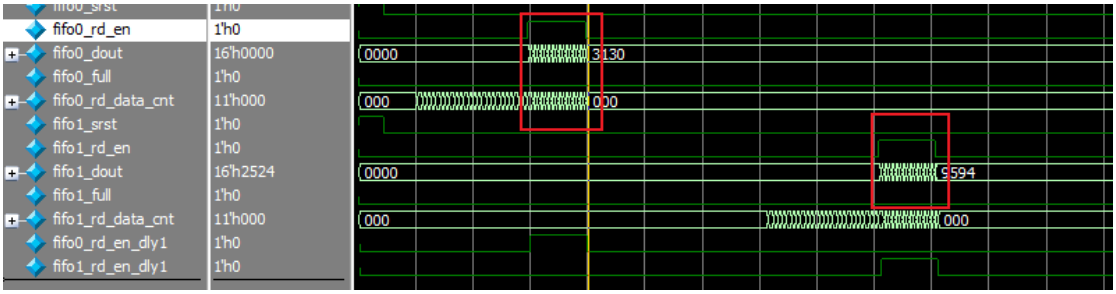


图 58-4 正确数据输出波形图

以第一包为例，写入 0~49 一共 50 个数据，最终输出到的波形图如下图 58-5 所示，可以看到我们写入的数据依次是 0x00、0x01、0x02、0x03....依次递增的 8 位数据，但是最终输出的 16 位数据是 0x0100、0x0302....，并不是 0x0001、0x0203....，这和高云提供的 FIFO 特性有关，这里我们需要知道是，本次实验提供的传图上位机是先传的低字节数据，后传的高字节数据，比如传输的数据为 0x12、0x34，实际的 16 位的图像数据就是 0x3412，通过该模块之后，就将传输的 8 位数据转换成实际的 16 位图像数据，后续我们直接将该数据进行显示即可，不需要再对数据进行任何处理。

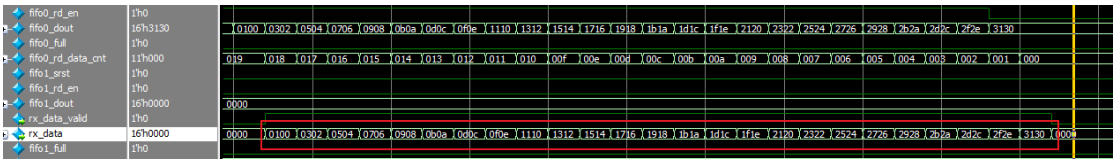


图 58-5 输出数据波形图

传输第三个包为错误的包，此时应该想 FIFO0 写入，但是由于是错误的，写入之后，通过判断后会清除 FIFO0 中的数据（产生 fifo0_srst 信号），最终也就不会输出，波形图如下图 58-6 所示。

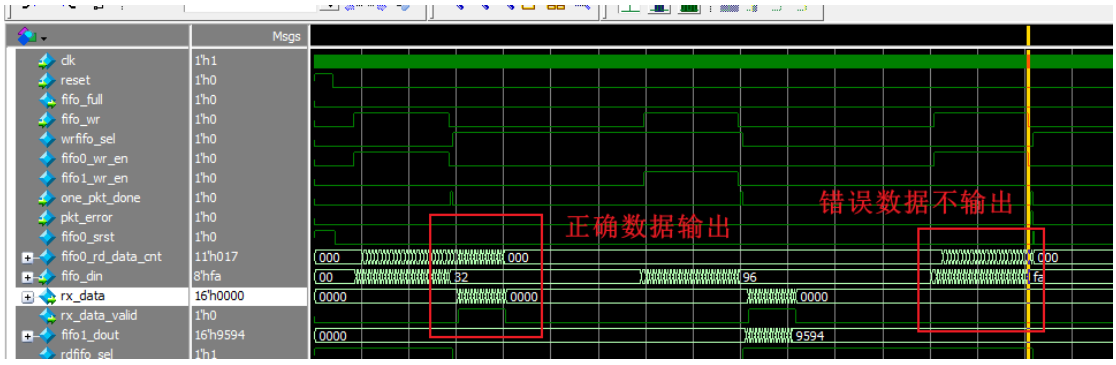


图 58-6 错误数据清除波形图

通过上述分析波形图可以看出，每个包传输过来的时候，会交替写入到两个 FIFO 中，当数据是正确的时，将数据输出，数据出现错误时，将数据清除不输出，符合设计预期的功能。

58.2.2数据包检测模块

本节将介绍数据包检测模块的代码设计及其仿真分析。

58.2.2.1数据包检测模块设计

数据包检测模块（frame_check 模块）判断每包数据的序号是否连续，如果不连续将整帧图像数据丢弃。其模块的结构框图如下图 58-7 所示。

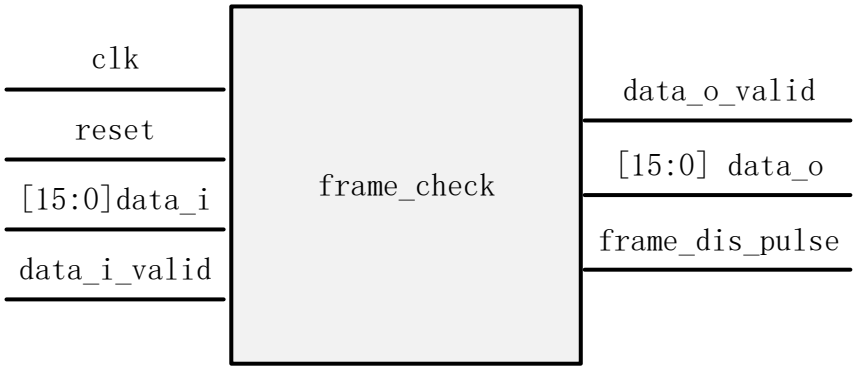


图 58-7 数据包检测模块结构框图

对每个信号的说明如下表 58-2 所示：

表 58-2 数据包检测模块信号说明表

信号名称	I/O	信号意义
clk	I	模块工作时钟信号
reset	I	模块复位信号，高有效
data_i[15:0]	I	从 FIFO 中读取出来的数据
data_i_valid	I	读取数据有效信号
data_o_valid	O	输出数据有效信号
data_o[15:0]	O	模块最终数据的数据
frame_dis_pulse	O	接收数据包错误标志脉冲信号

在我们所设计的以太网上位机软件中，每一个包传输的前两个字节代表的是需要传输图像的序号。通过检测每个包的数据的前两个字节的序号是否连续，来判断是否需要将数据丢弃。下面将介绍一下该模块部分代码实现。

首先，需要将传输图像数据的序号提取出来，通过检测到数据有效信号的上升沿来得到。将数据有效信号 data_i_valid “打两拍”得到 data_i_valid_dly1 信号和 data_i_valid_dly2 信号，当 data_i_valid_dly1 为高电平，data_i_valid_dly2 为低电平时，得到数据有效 data_i_valid_rising 脉冲信号，代码如下所示：

```
always@(posedge clk)
begin
  data_i_valid_dly1 <= data_i_valid;
  data_i_valid_dly2 <= data_i_valid_dly1;
  data_i_dly1 <= data_i;
end
```

```
assign data_i_valid_rising=data_i_valid_dly1 && (~data_i_valid_dly2);
```

其次，将每个包的序号提取出来之后，还得判断每个包序号是否连续。每次检测到 data_i_valid_rising 信号时，将 check_row_num 值加 1，然后判断 check_row_num 的值和 data_row_num 的值来判断序号是否一致。其中每个包的序号代表的就是所需要传输图像图像的行像素值，也就是 frame_check 模块的 IMAGE_HEIGHT 的值，所以 check_row_num 最大的值不超过 IMAGE_HEIGHT-1'b1。check_row_num 的变化代码如下：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    check_flag <= 1'b0;
  else if(data_i_valid_rising)
    check_flag <= 1'b1;
  else
    check_flag <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    check_row_num <= 'd0;
  else if(check_flag == 1'b1)
  begin
    if(check_row_num == IMAGE_HEIGHT - 1'b1)
      check_row_num <= 'd0;
    else
      check_row_num <= check_row_num + 1'd1;
  end
  else if(data_i_valid_rising && (data_i_dly1 == 'd0))
    check_row_num <= 'd0;
  else
    check_row_num <= check_row_num;
end
```

当 check_row_num 的值和 data_row_num 的值不一致时，也就是说以太网采集得到的图像数据序号不一致，此时产生数据错误标志信号 frame_dis_flag，当重新开始传输一幅图像时将 frame_dis_flag 信号拉低，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    frame_dis_flag <= 1'b0;
```

```
  else if(data_i_valid_rising && (data_i_dly1 == 'd0))
    frame_dis_flag <= 1'b0;
  else if(check_flag && (check_row_num != data_row_num))
    frame_dis_flag <= 1'b1;
  else
    frame_dis_flag <= frame_dis_flag;
end
```

通过将 frame_dis_flag 信号“打两拍”得到 frame_dis_flag_dly1 和 frame_dis_flag_dly2 信号，当 frame_dis_flag 信号为高电平，frame_dis_flag_dly2 为低电平时，得到检测数据错误脉冲 frame_dis_pulse，最终将该信号从模块中进行输出提供给其他模块使用，代码如下所示：

```
always@(posedge clk)
begin
  frame_dis_flag_dly1 <= frame_dis_flag;
  frame_dis_flag_dly2 <= frame_dis_flag_dly1;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    frame_dis_pulse <= 1'b0;
  else if(frame_dis_flag && (~frame_dis_flag_dly2))
    frame_dis_pulse <= 1'b1;
  else
    frame_dis_pulse <= 1'b0;
end
```

最后，当数据有效并且得到的图像的数据信号没有出现错误的情况下，将数据进行输出，并且产生输出数据有效信号 data_o_valid，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
  begin
    data_o       <= 'd0;
    data_o_valid <= 'd0;
  end
  else if(data_i_valid_dly1 && data_i_valid_dly2 && (~frame_dis_flag))
  begin
    data_o       <= data_i_dly1;
    data_o_valid <= 1'd1;
  end
  else
```

```
  begin
    data_o       <= 'd0;
    data_o_valid <= 'd0;
  end
end
```

58.2.2.2数据包检测模块仿真

数据包检测模块设计完成之后，需要通过仿真对其进行测试，通过仿真波形，查看是否实现功能：每包数据的序号不连续将整帧图像数据丢弃，序号连续则保留输出。如果满足该功能，则认为仿真正确。

在“frame_check_tb”文件中例化 frame_check 模块，然后设置分别写入 50 个 16'h0505、16'hffff 和 16'h0a0a，在 tb 文件中设置 IMAGE_HEIGHT 为 50，则每次写 50 个数据之后，行像素序号加 1，代码如下所示：

```
`timescale 1ns/1ns
`define PERIOD_CLK 8

module frame_check_tb();
reg       clk;
reg       reset_n;
reg [15:0]data_i;
reg       data_i_valid;
wire      data_o_valid;
wire[15:0]data_o;
wire      frame_dis_pulse;
integer i;
initial clk = 1'b1;
always #(`PERIOD_CLK/2)clk = ~clk;
initial begin
  reset_n = 1'b0;
  data_i = 16'd0;
  data_i_valid = 1'b0;
  #(`PERIOD_CLK*200+1)
  reset_n = 1'b1;
  #200;

  for(i=0;i<50;i=i+1) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
    data_i = i;
    repeat(50) begin
```

```
    @(posedge clk)
    #1 data_i = 16'h0505;
    end
    data_i_valid = 1'b0;
    #200;
  end

  for(i=0;i<100;i=i+2) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
    data_i = i;
    repeat(50) begin
    @(posedge clk)
    #1 data_i = 16'hffff;
    end
    data_i_valid = 1'b0;
    #200;
  end

  for(i=0;i<50;i=i+1) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
    data_i = i;
    repeat(50) begin
    @(posedge clk)
    #1 data_i = 16'h0a0a;
    end
    data_i_valid = 1'b0;
    #200;
  end
  #200;
  $stop;
end
frame_check
#(
  .IMAGE_WIDTH  (  50 ),
  .IMAGE_HEIGHT (  50 ),
  .DATA_WIDTH   (  16 )
)frame_check
(
  .clk            (clk            ),
  .reset          (~reset_n        ),
  .data_i         (data_i         ),
  .data_i_valid   (data_i_valid   ),
```

```
  .data_o_valid   (data_o_valid    ),
  .data_o         (data_o          ),
  .frame_dis_pulse(frame_dis_pulse )
);
endmodule
```

下面将对仿真波形图进行分析。

首先是获取数据序号，波形图如下图 58-8 所示。从图中可以看出，当检测到数据有效信号之后，会产生 data_i_vaild_rising 信号，此时将 data_i 中的数据提取出来，得到了数据序号并赋值给了 data_row_num。

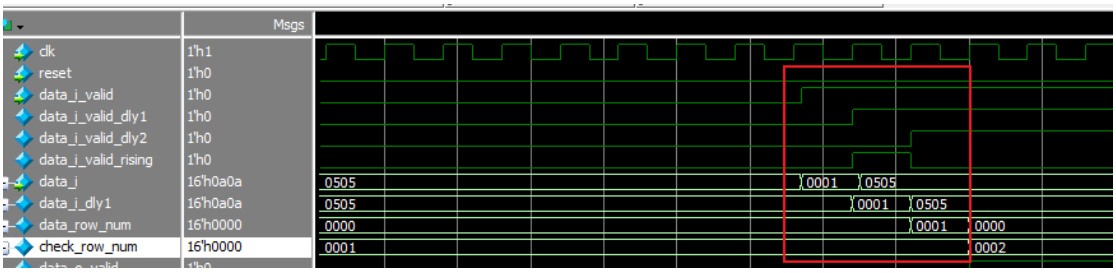


图 58-8 获取数据序号波形图

随后与 check_row_num 的值进行对比，判断序号是否一致，波形图如下图 58-9 所示。从波形图中可以看出，此时序号一致，代表数据正确，最终将数据进行输出并产生数据有效信号 data_o_valid。

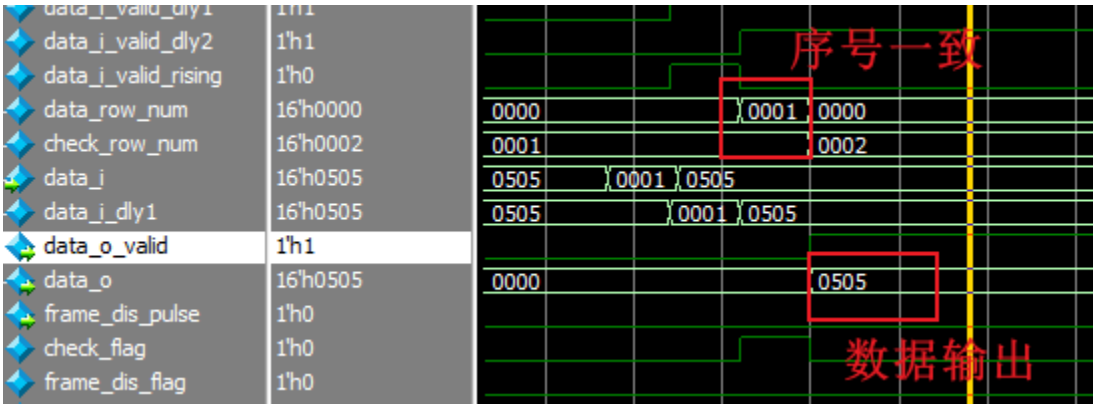


图 58-9 正确数据输出波形图

当传输 16'hffff 的时候，波形图如下图 58-10 所示。从图中可以看出，当出现序号不一致的情况，此时说明这帧数据出现错误，会出现 frame_dis_pulse 信号，数据不会输出。

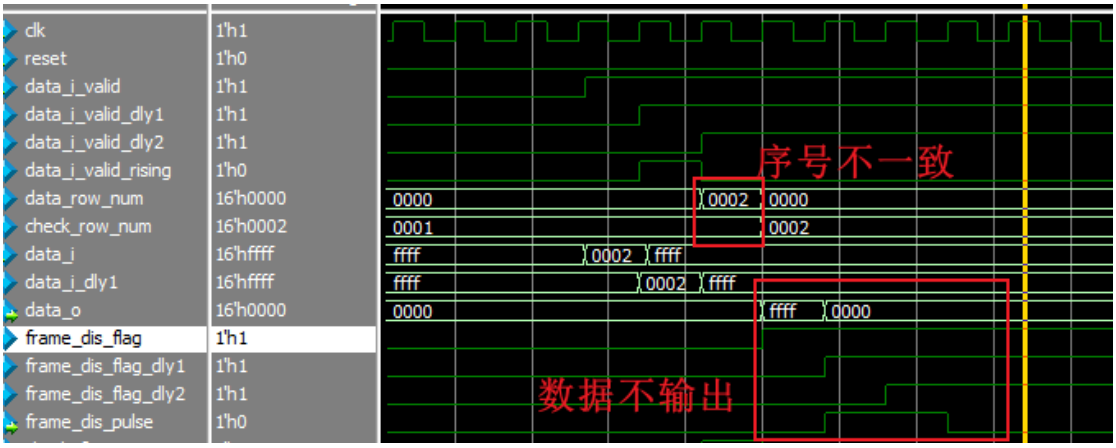


图 58-10 序号不一致数据不输出波形图

通过上述对仿真波形的分析看出，只有每包数据序号连续的情况下，数据才会输出，否则数据不输出，符合设计预期的功能。

58.2.3DDR 控制器模块

在前面的章节中，我们详细介绍过双端口 DDR3 控制器 ddr3_ctrl_2port 的设计，本次实验将使用该控制器，将以太网采集得到的数据存入 DDR 中。

首先是 DDR 控制器的写入控制，写入的数据是从以太网接收而来，所以其写入 FIFO 时钟为 125M，当数据出现错误或者产生复位信号时，更新写入 DDR 中的数据。frame_check 模块最终输出的信号 image_data 为待写入的数据。FIFO 的写使能是 frame_check 模块产生的输出数据有效信号 image_data_valid，代码如下所示：

```
assign wr_load = g_rst_p | frame_dis_pulse;
assign wr_clk = eth_clk125m;
assign wfifo_wren = image_data_valid;
assign wfifo_din = image_data;
```

其次是 DDR 控制器的读出控制，最终读出的数据是提供给 TFT 屏进行显示的，所以读出时钟为 TFT 屏的工作时钟 33M，FIFO 的读使能信号为 disp_driver 模块的屏幕可见显示区标识信号 DataReq，ddr3_ctrl_2port 模块的输出源更新信号为 disp_driver 模块的一帧图像起始标识信号 Disp_Sof，最终读出的数据提供给 disp_driver 模块的输入数据信号 disp_data，代码如下所示：

```
assign rd_load = disp_sof;
assign rd_clk = loc_clk33m;
assign rfifo_rden = disp_data_req;
assign disp_data = rfifo_dout;
```

在前面介绍中，DDR 控制器的读出控制是由 TFT 屏显示驱动模块

（disp_driver）控制的，本章实验将不会介绍该模块，该模块的具体实现请看前面章节“TFT 显示屏驱动设计与验证”一节的内容。

模块设计完成之后，只需要在顶层文件中对各个模块之间的接口信号进行连接，完整的顶层文件代码请自行查看例程文件。

58.3系统板级测试

经过以上工作，代码设计部分的任务已经全部完成，接下来将进行板级验证。本次实验的板级验证环节，主要验证：通过以太网传图工具将图像数据传输到高云开发板上的DDR3进行存储，然后图像数据由TFT控制器主动从DDR3中读取出来，送往 TFT 显示屏进行显示，最终查看 TFT 屏上的图像数据是否显示正常。

58.3.1硬件连接

将 TFT 屏、网线、下载器、电源线依次连接在开发板上，整体的硬件连接图如下图 58-11 所示。

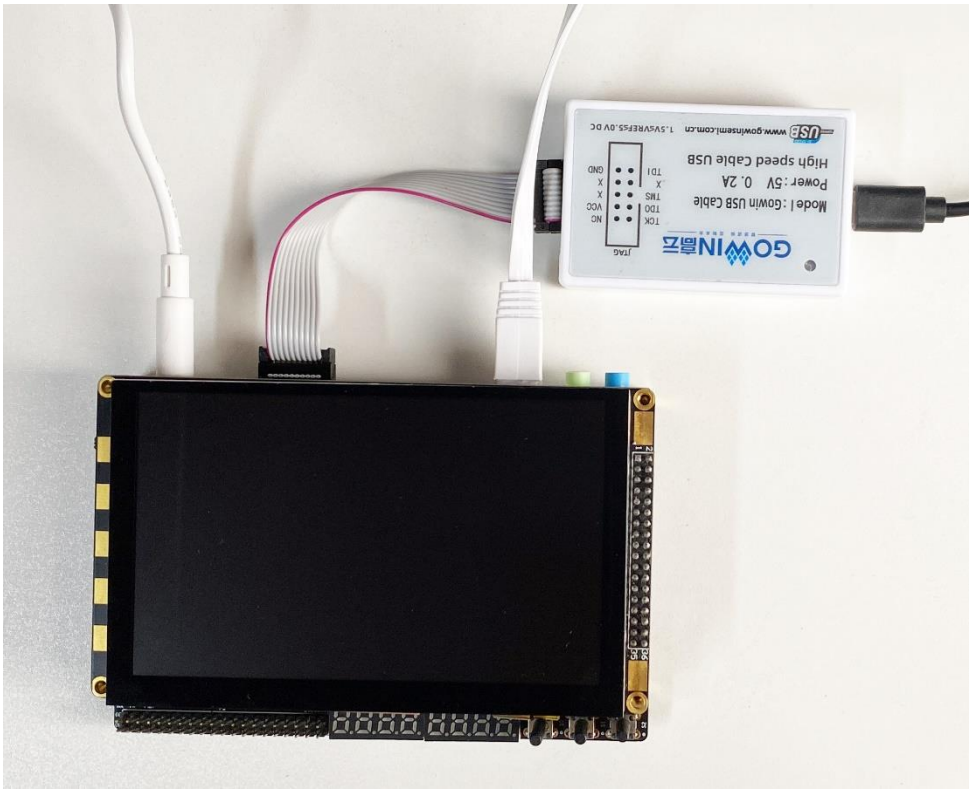


图 58-11 系统硬件连接图

连接完成之后，编译工程，将生成的数据流文件下载至开发板中。

58.3.2修改电脑 IP 地址

本次实验的源 IP 地址为 192.168.0.2，目的 IP 地址(PC 端)需要与源 IP 地址保持在同一网段，用户可以修改为 192.168.0.3。在本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址。如下图 58-12 所示。

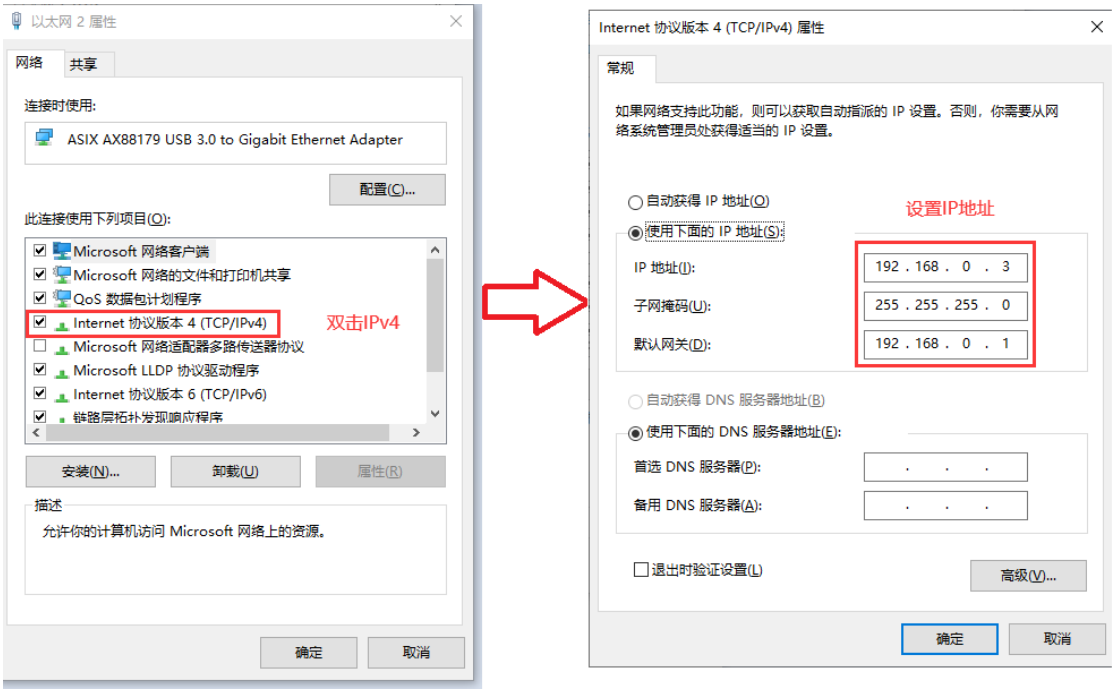


图 58-12 修改电脑 IP 地址

58.3.3绑定 ARP

本工程不支持 ARP 协议，只能通过静态绑定的方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

关于 ARP 的绑定请查看我们论坛上的帖子以太网通信静态 ARP 绑定方法与常见问题解决方案。

58.3.4Picture2Hex 生成图片数据文件

1. 读者在工程文件夹中找到我们提供的 Picture2Hex.exe 软件，软件打开之后，配置界面如下图 58-13 所示，将图像数据的 width 和 high 设置为 800*480。

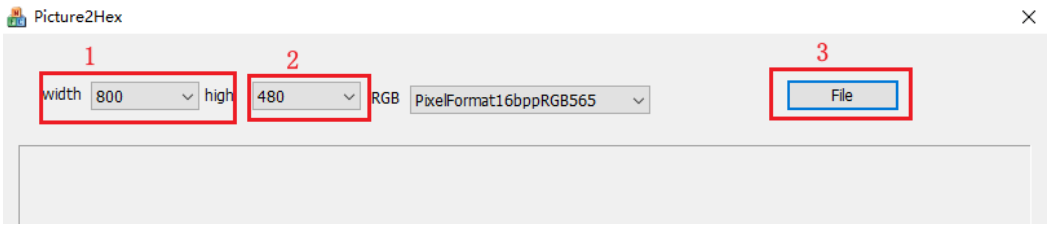


图 58-13 Picture2Hex 软件配置界面

2. 点击 File 之后，找到待显示的图片，图片支持 bmp，png，jpg 格式，选择完图片后，在 work 目录下会生成一个 logo.c 文件，这个文件就是图片转换的数据文件，如下图 58-14 所示。

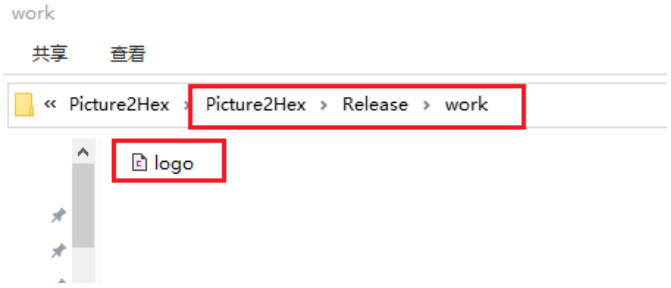


图 58-14 生成的图像.c 文件

3. 将 logo.c 改名为 test800x480.c。

4. 将 test800x480.c 拷贝到以太网传图上位机的 img_data 目录下，如下所示。

图 58-15 拷贝文件只以太网上位机文件夹下

58.3.5运行结果显示

运行以太网传图上位机“以太网传图.exe”，这样可以看到开发板 TFT 屏上显示 PC 下传的图片。显示效果如下图 58-16 所示。上位机软件在我们提供的工程目录下。

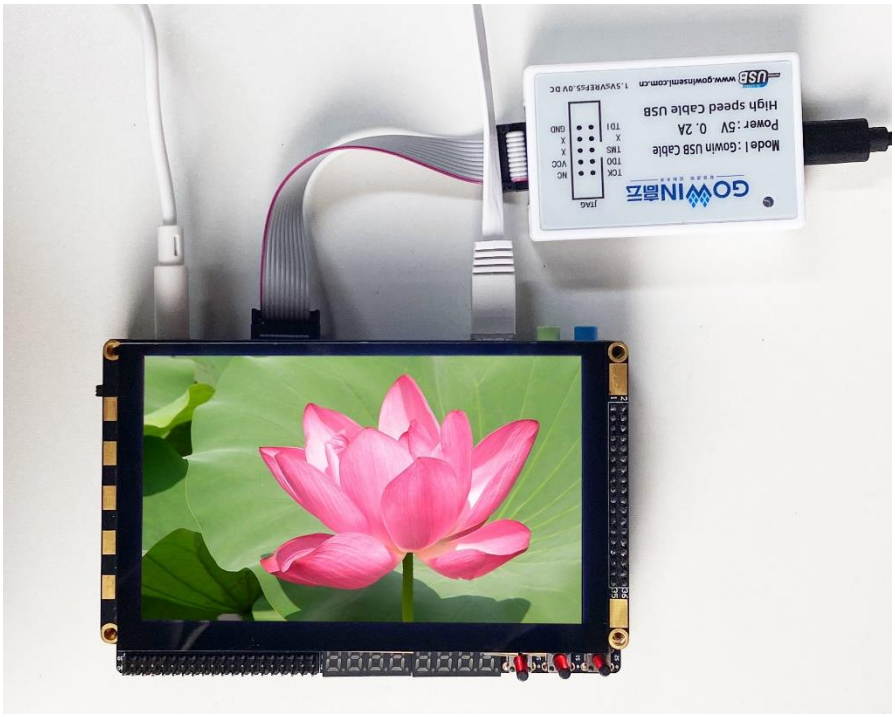


图 58-16 最终效果显示

58.4常见问题说明

当实验现象无法出现时，可以考虑从如下方面查看问题。

1. 检查你的电脑有线网络本地连接是否连接上了。检查你的电脑 IP 地址是否已经设置为了 192.168.0.3.
2. 接收不到图像的情况下，手动确认防火墙是否已经被关闭。
3. 确保开启巨型帧。开启巨型帧的方式如下图 58-17 所示。

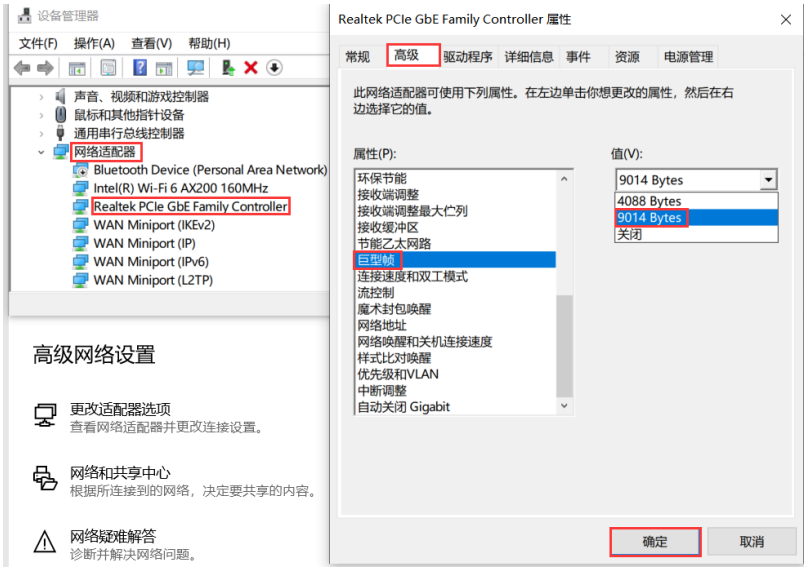


图 58-17 电脑开启巨型帧方式

4. 点击“本地连接 ”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如下图 58-18 所示。

图 58-18 确保连接速度为千兆速率

58.5思考与总结

本章实验通过以太网上传图工具，将需要传输的图像的数据通过以太网传输至 DDR3 中进行存储，最终由 TFT 控制器主动读取出来，送给 TFT 屏进行显示。

本章实验主要讲解了以太网控制模块和数据包检测模块的设计与仿真，这两个模块主要都是用来检测传输的数据是否正确，在介绍以太网控制模块时，提到了使用两个 FIFO 进行双缓存，这样做是为了在 CRC 校验出现问题时把这包数据丢掉，并且确保不影响下一包数据的传输。使用双 FIFO 缓存就是一个“乒乓”操作，是一个用于数据流控制的处理技巧，也是本次实验一个比较核心的内容，希望读者可以好好理解，方便在之后自己的工程中可以学以致用。

59 千兆以太网传输 ACM7606 数据采集

工程源码	|----02_设计实例 |----ch59_ad7606_ddr3_rgmii
相关视频课程	
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本节实验结合 ADI 公司的 16 位 8 通道并行采样 ADC 芯片 AD7606，并利用高云开发板上一片 Realtek 的 RTL8211 以太网收发器，实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出到电脑。FPGA 采用 RGMII 接口与以太网 PHY 芯片通信，接收以太网数据包，并提取出以太网数据包中 UDP 协议报文的数据内容，然后将数据转化成控制命令，最终实现对 AD7606 的采样频率、数据采样个数以及采样通道的合理配置，采集完成后的数据存放至 DDR3 中，然后通过网口以 UDP 协议传输到电脑。用户可以在电脑上通过网口调试工具进行指令的下发，并对采样到的数据进行导出，然后使用 MATLAB 软件进行进一步的数据处理分析。

59.1系统整体设计

在整个设计中，首先通过电脑上的网络调试助手，将命令帧进行发送，然后通过高云开发板上的以太网芯片接收，随后将接收到的数据转换成命令，最终实现对 AD7606 采样频率、数据采样个数以及采样通道的配置。配置完成之后，AD7606 开始采集数据，将 AD7606 采集的数据存储至 DDR3 中，最后数据通过网口传输至电脑，电脑端将采集到的数据通过 MATLAB 进行进一步的分析，系统的整体设计框图如下所示。

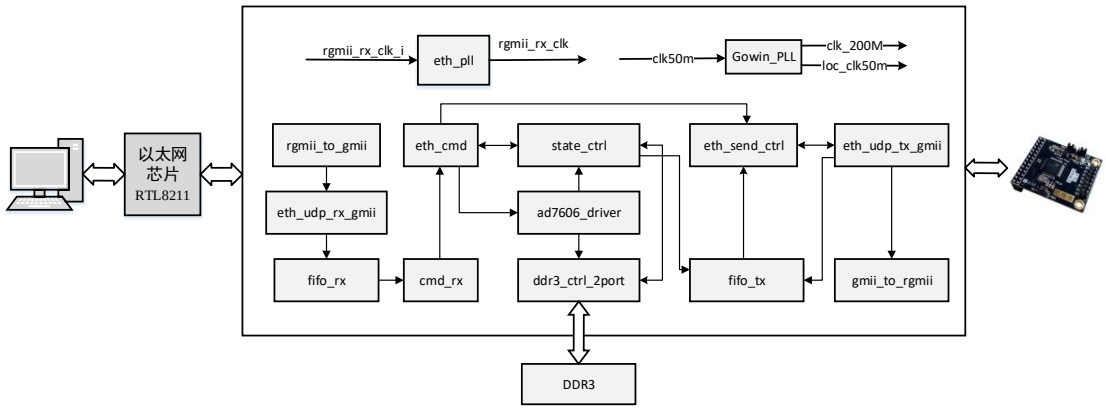


图 59-1 千兆以太网传输 ACM7606 数据采集整体设计框图

下面对上述图中的各个模块的功能进行简要说明：

1. Gowin_PLL 模块：锁相环模块，输入时钟 50M，由开发板上的晶振提供；输出 200M 的时钟给到 DDR3 控制器使用；输出 50M 的时钟给其他模块使用。
2. eth_pll 模块：锁相环模块，将 rgmii 接口时钟信号 rgmii_rx_clk_i 偏移 270°得到 rgmii_rx_clk 时钟信号，这样做是为了在时钟上升沿/下降沿取数据时，取得的数据是最准确和稳定的，具体的偏移角度是实际使用时，不断调试得到的，在我们的高云的开发板上，偏移 270°是最合适的。
3. eth_udp_rx_gmii 模块：GMII 以太网接收模块，接收用户在电脑上通过网络助手或者上位机发送的包含指令数据的数据帧。
4. fifo_rx 模块：FIFO IP 核，用来缓存通过以太网下发的指令，解决跨时钟域的问题。
5. eth_cmd 模块：接收转命令模块，对接收到的以太网数据进行分析，提取出每个控制命令帧。
6. cmd_rx 模块：指令转控制模块，将从接收转命令模块接收到的数据转换为相应的控制数据并分别输出到对应的模块。
7. ad7606_driver 模块：AD7606 控制器驱动模块，该控制器实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出。使用该控制器时，用户无需关心 AD7606 的具体控制时序，一切都在控制器内部完成，用户只需要像使用并行 ADC 一样取用数据即可。
8. state_ctrl 模块：ADC 采集数据 DDR3 缓存以太网发送状态控制模块，协调各个模块的信号控制，程序状态的总控制模块。
9. ddr3_ctrl_2port 模块：DDR3 双端口模块，用来缓存 ACM7606 采集到的数据。
10. fifo_tx 模块：以太网发送 FIFO，从 DDR3 中读取数据进入该 FIFO 中，然后从该模块中读出交由以太网发送模块发出，主要是解决跨时钟的问题。
11. eth_send_ctrl 模块：以太网发送控制模块，该模块主要控制网口发送模块的使能控制信号以及对以太网数据帧数据长度的控制。
12. eth_udp_tx_gmii 模块：网口发送模块，将采集到的数据以以太网帧的

形式进行发送。

13. gmii_to_rgmii 模块：以太网发送 gmii 转 rgmii，将 gmii 接口信号转换成 rgmii 接口。

除去使用 IP 和前面章节讲过的模块外，还需要设计的模块包括eth_cmd 模块、cmd_rx 模块、state_ctrl 模块、ad7606_driver 模块和 eth_send_ctrl 模块。

59.2ACM7606 模块介绍

ACM7606 数据采集模块使用的是 ADI 公司的 16 位 8 通道同步采样模数转换器 AD7606，模块图如下图 59-2 所示。

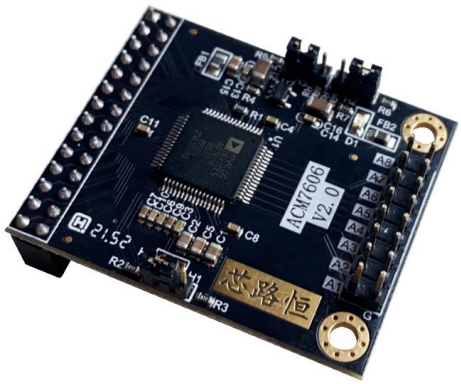


图 59-2 ACM7606 模块图

AD7606 是 16 位 8 通道同步采样模数数据采集系统（DAS）。内置模拟输入箝位保护、二阶抗混叠滤波器、跟踪保持放大器、16 位电荷再分配逐次逼近型模数转换器（ADC）、灵活的数字滤波器、2.5V 基准电压源、基准电压缓冲以及高速串行和并行接口。AD7606 采用 5V 单电源供电，可以处理±10V 和±5V 真双极性输入信号。同时所有通道均能以高达 200kSPS 的吞吐速率采样。输入箝位保护电路可以耐受最高达±16.5V 的电压。无论以何种采样频率工作，其模拟输入阻抗均为 1MΩ。采用单电源工作方式，具有片内滤波和高输入阻抗，因此无需驱动运算放大器和外部双极性电源。AD7606 抗混叠滤波器的 3dB 截止频率为 22kHz；当采样频率为 200Ksps 时，它具有 40dB 的抗混叠抑制特性。

芯片对外提供 SPI 和并行的数字接口。当 AD7606 的 8 个通道全部以 200KPS 的最高速率进行转换时，数据输出速率达到 25.6Mbps，需要使用高性能 MCU 的 SPI 外设才能勉强该速率要求。因此可以使用 16 位并口来进行数据的传输，提高数据传输速率。当 AD7606 应用在 FPGA 系统中的时候，使用 SPI 串行接口和并行接口都能够轻松的满足数据传输的速率需求。当在 FPGA 系统上应用

AD7606 时，可以通过在 FPGA 上设计 AD7606 控制转换逻辑，将转换结果数据直接存储到片上的存储器如 FIFO 或者 RAM 中，也可以存储到 FPGA 片外的存储器如 SRAM 或 SDRAM 中，然后由其他主控芯片如 MCU 或 DSP 读出，或者直接在 FPGA 内部进行数据的运算和处理。当然，由于 FPGA 片上可以设计软核控制器，也可以直接使用软核控制器完成数据的处理和传输工作。

59.2.1功能框图

AD7606 的功能框图如下所示。

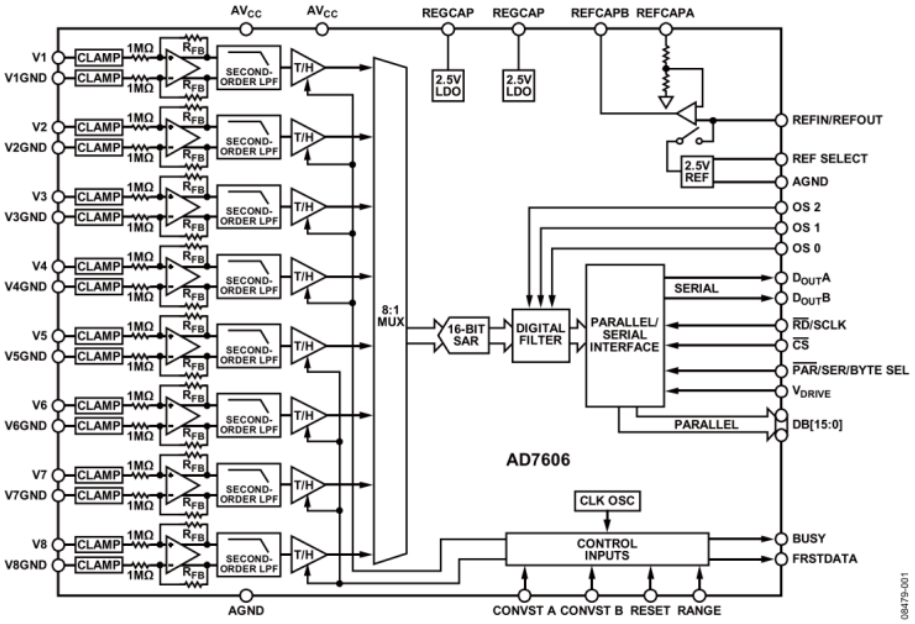


图 59-3 AD7606 功能框图

如上图所示，采集到的数据在经过稳压滤波和采样保持后通过 8 选 1 多路选择器被分别送入到 16 位逐次逼近型 ADC 芯片 AD7606 中进行转换，最后经由数字滤波后输出，当数据以串行模式输出时，数据会从 DoutA、DoutB 中输出，如果数据是以并行方式输出，那么数据将从 DB[15:0]中输出。

59.2.2模拟输入

AD7606 可处理真双极性、单端输入电压。RANGE 引脚的逻辑电平决定所有模拟输入通道的模拟输入范围。如果此引脚与逻辑高电平相连，则所有通道的模拟输入范围为±10V。如果此引脚与逻辑低电平相连，则所有通道的模拟输入范围为±5V。AD7606 的模拟输入阻抗为 1MΩ。这是固定输入阻抗，不随 AD7606 采样频率而变化。AD7606 的输入结构如下图 59-4 所示，其各路模拟输

入均含有箝位保护电路。虽然采用 5V 单电源供电，但此模拟输入箝位保护允许输入过压达到±16.5V。

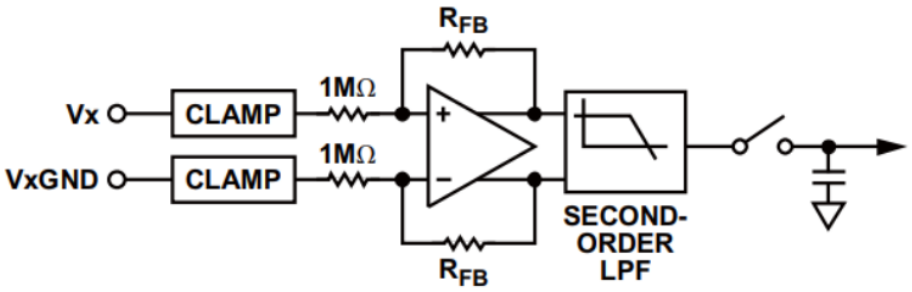


图 59-4 AD7606 模拟输入结构

59.2.3数字滤波器与过采样

AD7606 内置一个可选的数字一阶 sinc 滤波器，在使用较低吞吐率或需要更高信噪比或更宽动态范围的应用中，应使用该滤波器。数字滤波器的过采样引脚 OS[2:0]控制（具体参考 AD7606 数据手册）。OS2 为 MSB 控制位。OS0 为 LSB 控制位，下表 59-1 提供了用来选择不同过采样倍率的过采样位解码。

表 59-1 不同过采样倍率的过采样位解码

OS[2:0]	过采样倍率	5V 范围 SNR(dB)	10V 范围 SNR(dB)	5V 范围 3dB 带宽(kHz)	10V 范围 3dB 带宽(kHz)	最大吞吐量 CONVST 频率(kHz)
000	No OS	89	90	15	22	200
001	2	91.2	92	15	22	100
010	4	92.6	93.6	13.7	18.5	50
011	8	94.2	95	10.3	11.9	25
100	16	95.5	96	6	6	12.5
101	32	96.4	96.7	3	3	6.25
110	64	96.9	97	1.5	1.5	3.125
111	无效					

OS 引脚在 BUSY 下降沿锁存，从而设置下一个转换的过采样倍率，如下图 59-5 所示，如果 OS 引脚选择过采样倍率 8，则下一个 CONVST x 上升沿采集各通道的第一个采样点，一个内部产生的采样信号采集所有通道的其余 7 个样点，然后对这些样点求平均值，以改进 SNR 性能。开启过采样时，CONVST A 和 CONVST B 引脚必须连在一起驱动，转换过程中 BUSY 保持高电平的时间会延长。BUSY 保持高电平的实际时间取决于所选的过采样倍率，过采样倍率越高，则 BUSY 保持高电平的时间或总转换时间越长。

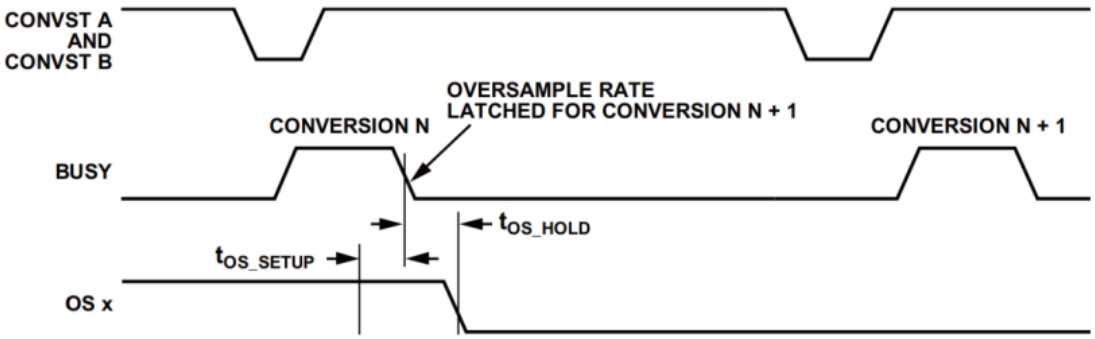


图 59-5 OS 引脚在 BUSY 下降沿锁存时序示意图

59.2.4工作时序

AD7606 根据采样方式不同具有多种驱动时序，本次实验采用的为并行输出（即 8 个 16 位的数据通过 16 根并行线一个接着一个输出），转换后读取模式。其时序图由两部分组成：完成 AD 转换和读取 AD 数据。其中的时间可以参考 ADI 公司的手册。当 CONVST A 和 CONVST B 通道都变为上升沿时，BUSY 信号转变为高电平，代表转换开始，直到 BUSY 的下降沿到来，代表数据已经转换完成，正在锁存至输出数据寄存器中，当$\overline{CS}$变为下降沿时，数据将会被输送到总线上。并行工作模式下，当$\overline{CS}$和$\overline{RD}$都为低电平时，会使能总线，将转换结果输出到并行数据总线上，当 V1 转换结果开始输出之后，FRSTDATA 会随后转变为高电平，表示输出数据总线可以提供 V1 的结果。并行模式下，每次数据的输出为 16 位，对应一个通道。

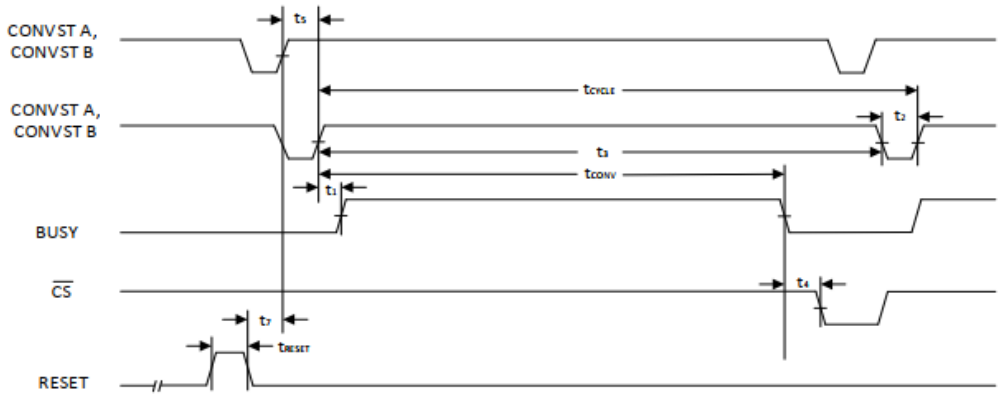


图 59-6 CONVST 时序—转换之后读取

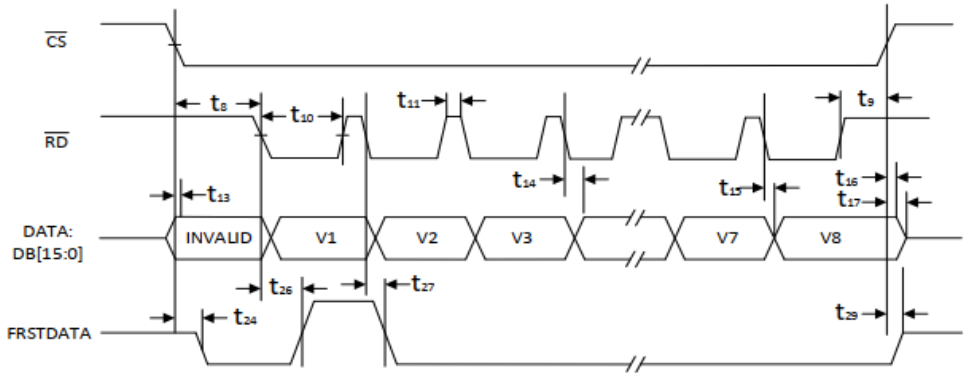


图 59-7 并行模式，独立的CS和RD脉冲

59.3模块设计

59.3.1fifo_rx 模块

FIFO IP 核，以太网接收模块的工作时钟为 125Mhz，命令解析模块以及后续的 AD7606 控制器的工作时钟为 50Mhz，两者速率不匹配，使用该 IP 核解决采集过程中会出现的跨时钟域数据交互问题。

我们这里配置读写位宽为 8 位（与以太网接收模块的数据位宽保持一致），写深度为 256，配置模式选择“First-Word Fall-Through”，使用同步复位，整体的配置界面如下所示。

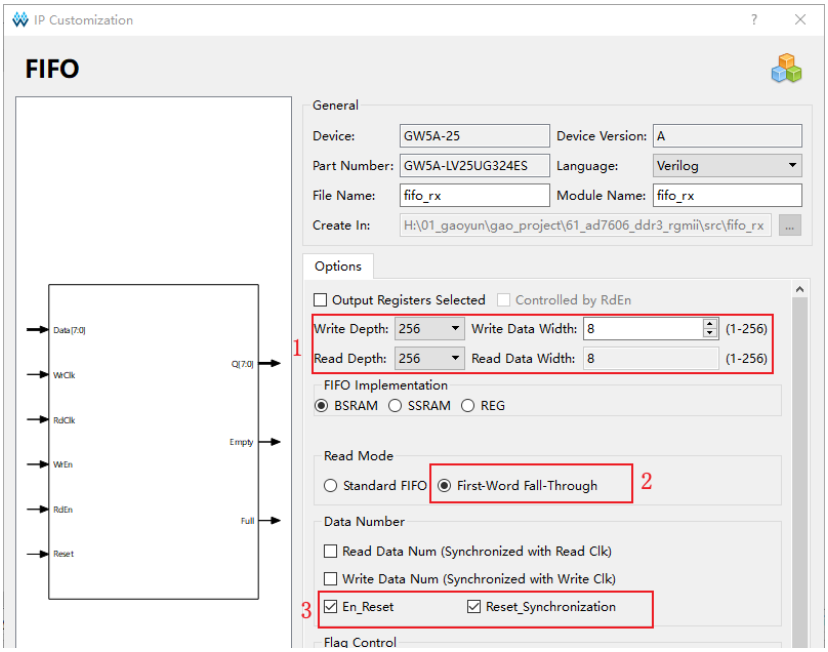


图 59-8 fifo_rx 配置界面

59.3.2eth_cmd 模块

接收转命令模块 eth_cmd，将以太网传输过来的指令数据帧进行拆解，得到需要的指令数据传送给别的模块进行处理，该模块的结构框图如下图 59-9 所示。

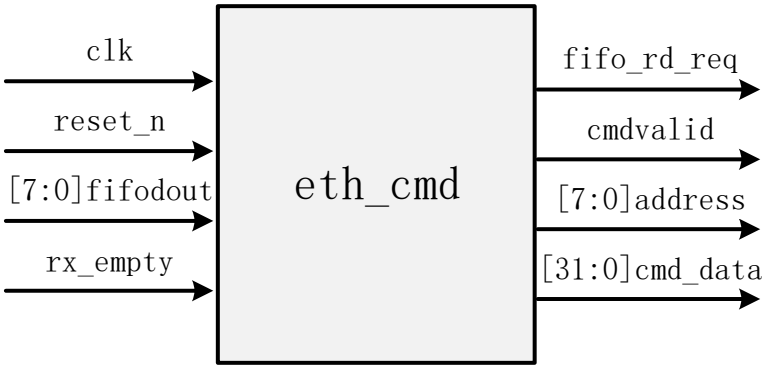


图 59-9 接收转命令模块框图

模块信号说明如下表 59-2 所示。

表 59-2 接收转命令模块信号说明表

信号名称	I/O	信号意义
clk	I	模块工作时钟
reset_n	I	模块复位信号，低电平复位
fifodout[7:0]	I	从 FIFO 中读出的 8 位数据
rx_empty	I	FIFO 为空标志信号
fifo_rd_req	O	FIFO 的读请求信号
cmdvalid	O	输出命令有效标志信号
address[7:0]	O	配置 AD7606 的寄存器地址信号
cmd_data[31:0]	O	写入到寄存器中的数据

网口一次发送的命令数据内容为 32 个字节，为了实现通过网口修改这些寄存器的值，需要对发送一次的数据进行拆解才能实现，对于设计的数据帧，一帧数据一共 8 个字节，包含帧头、帧尾、地址段、数据段。帧格式如下表 59-3 所示。

表 59-3 帧格式说明表

数据	D0	D1	D2	D3	D4	D5	D6	D7
功能	帧头 0	帧头 1	地址 address	data[31:24]	data[23：16]	data[15:8]	data[7:0]	帧尾
值	0x55	0xA5	XX	XX	XX	XX	XX	0xF0

从上表中可以看出，每帧数据一共 8 个字节，分别用 D0~D7 表示，其中，D0 和 D1 两个数据作为帧头，其值固定为 0x55、0xA5，D7 作为帧尾，其值固定为 0xF0。帧头和帧尾的作用是为了准确识别数据帧，确保接收的数据是我们需要分析的。D2 代表的是要操作的寄存器地址，D3 为要写入寄存器的数据的

24~31 位，D4 为要写入寄存器的数据的 16~24 位，D5 为要写入寄存器的数据的 8~15 位，D6 为要写入寄存器的数据的 0~7 位。

该模块的作用就是将网口接收的数据拆解成上述帧格式，将 D2 作为地址 address 输出，指定修改哪个寄存器，D3~D6 共 32 位作为数据 data 输出，控制 AD7606 进行相应的配置。下面将对模块中的部分代码进行说明：

首先，产生 FIFO 的读请求信号。当检测到 FIFO 非空的时候，产生 FIFO 读请求信号，代码如下所示：

```
always@(posedge clk or negedge reset_n)
if(!reset_n)
  fifo_rd_req <= 1'b0;
else if(!rx_empty)
  fifo_rd_req <= 1'b1;
else
  fifo_rd_req <= 1'b0;
```

然后得到帧命令数据，每产生一次读请求，将会从 FIFO 中读取一个 8 位的数据，连续存储 8 个字节的数据就得到一帧命令数据。代码如下所示：

```
always@(posedge clk)
if(fifo_rd_req)begin
  data_0[7] <= #1 fifodout;
  data_0[6] <= #1 data_0[7];
  data_0[5] <= #1 data_0[6];
  data_0[4] <= #1 data_0[5];
  data_0[3] <= #1 data_0[4];
  data_0[2] <= #1 data_0[3];
  data_0[1] <= #1 data_0[2];
  data_0[0] <= #1 data_0[1];
end
```

最后是判断得到的帧命令数据是否正确，当数据符合 D0 为 8'h55，D1 为 8'hA5，D7 为 8'hF0，则代表该数据格式正确，会生成一个指令正确信号 cmdvalid 输出到指令转控制模块，并将数据进行输出，代码如下所示：

```
always@(posedge clk or negedge reset_n)
if(!reset_n)begin
    address <= 0;
    cmd_data <= 32'd0;
    cmdvalid <= 1'b0;
end
else if(fifo_rx_done)begin
if((data_0[0] == 8'h55)&&(data_0[1]==8'hA5)&&(data_0[7]==8'hF0))
begin
        cmd_data[7:0] <= #1 data_0[6];
        cmd_data[15:8] <= #1 data_0[5];
```

```
        cmd_data[23:16] <= #1 data_0[4];
        cmd_data[31:24] <= #1 data_0[3];
        address <= #1 data_0[2];
        cmdvalid <= #1 1;
    end
else
        cmdvalid <= #1 0;
end
else
    cmdvalid <= #1 0;
```

59.3.3cmd_rx 模块

指令转控制模块（cmd_rx）将从接收转命令模块接收到的数据转换为相应的控制数据，首先将对寄存器进行说明，其功能和地址分别如下所示。

表 59-4 寄存器说明表

名称	地址	位宽	功能简介
RestartReq	0	1	重新开始采集请求寄存器，向该寄存器写入任意值即可启动新一轮的采样存储传输
ChannelSel	1	8	通道设置寄存器，共 8 位， 对应了 8 个通道的数据存储开关，如果某通道对应的设置为 1，则该通道的采样结果就会被存入 FIFO 并通过网口发送，注意对应的 2 进制的位，不是 10 进制，比如设置通道 3，对应 01 地址需要写入的值为 04(100)，而不是 03(011)。
DataNum	2	32	数据个数寄存器，设定总共采集传输多少个数据。注意，该寄存器设置的是总共采集的数据个数， 假设设置采集 100 个数据，ChannelSel 为 0000_0011b， 则实际每个通道采样的次数就是 50， 2 个通道的数据加起来是 100 个。假设设置采集 100 个数据，而且设置了 ChannelSel 为 0011_0011b，则实际每个通道采样的次数就是 25，4 个通道加起来采集 100 个数据
ADC_Speed_Set	3	32	ADC 采样速率设置寄存器。该寄存器用来设置 ADC 每多久执行一次转换。 由于 ADC 的最大采样速率为 200Ksps，所以可以通过设置该寄存器的值来让 ADC 的采样速率在 1~200Ksps 范围内调整，以适应不同的应用场景。ADC_Speed_Set=1000000000/20/speed-1，其中 speed 就是实际要设置的采样速率。

指令转控制模块结构框图如下图 59-10 所示。

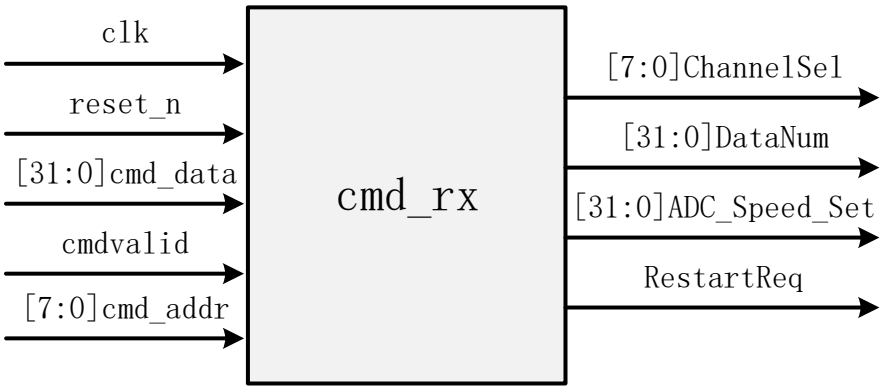


图 59-10 指令转控制模块结构框图

模块信号说明如下所示。

表 59-5 指令转控制模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号
reset_n	I	模块复位信号，低电平复位
cmd_data[31:0]	I	写入到寄存器中的值
cmdvalid	I	命令有效标志信号
cmd_addr[7:0]	I	寄存器地址信号
ChannelSel[7:0]	O	通道设置寄存器
DataNum[31:0]	O	数据个数寄存器
ADC_Speed_Set[31:0]	O	ADC 采样速率控制寄存器
RestartReq	O	重新开始采集请求信号

根据表 59-4 中的内容，地址 cmd_addr 为 0 时，产生 RestartReq；cmd_addr 为 1 时，得到通道设置数据 cmd_data[7:0]；cmd_addr 为 2 时，得到需要采样的数量 cmd_data[31:0]；cmd_addr 为 3 时，得到设置的采样速率的值，代码如下所示：

```
always@(posedge clk or negedge reset_n)
if(!reset_n)begin
    ChannelSel <= 8'b1111_1111;
    DataNum <= 16'd32;
    ADC_Speed_Set <= 32'd9999;
    RestartReq <= 1'b0;
end
else if(cmdvalid)begin
  case(cmd_addr)
      0: RestartReq <= 1'b1;
      1: ChannelSel <= cmd_data[7:0];
      2: DataNum <= cmd_data[31:0];
      3: ADC_Speed_Set <= cmd_data[31:0];
  default:;
  endcase
```

```
end
else
  RestartReq <= 1'b0;
```

59.3.4ad7606_driver 模块

AD7606 控制器驱动模块（ad7606_driver），该控制器实现了对 AD7606 型 8 通道 16 位 ADC 的数据转换控制并输出。使用该控制器时，用户无需关心 AD7606 的具体控制时序，一切都在控制器内部完成，用户只需要像使用并行 ADC 一样取用数据即可。该模块的内部结构框图如下图 59-11 所示。

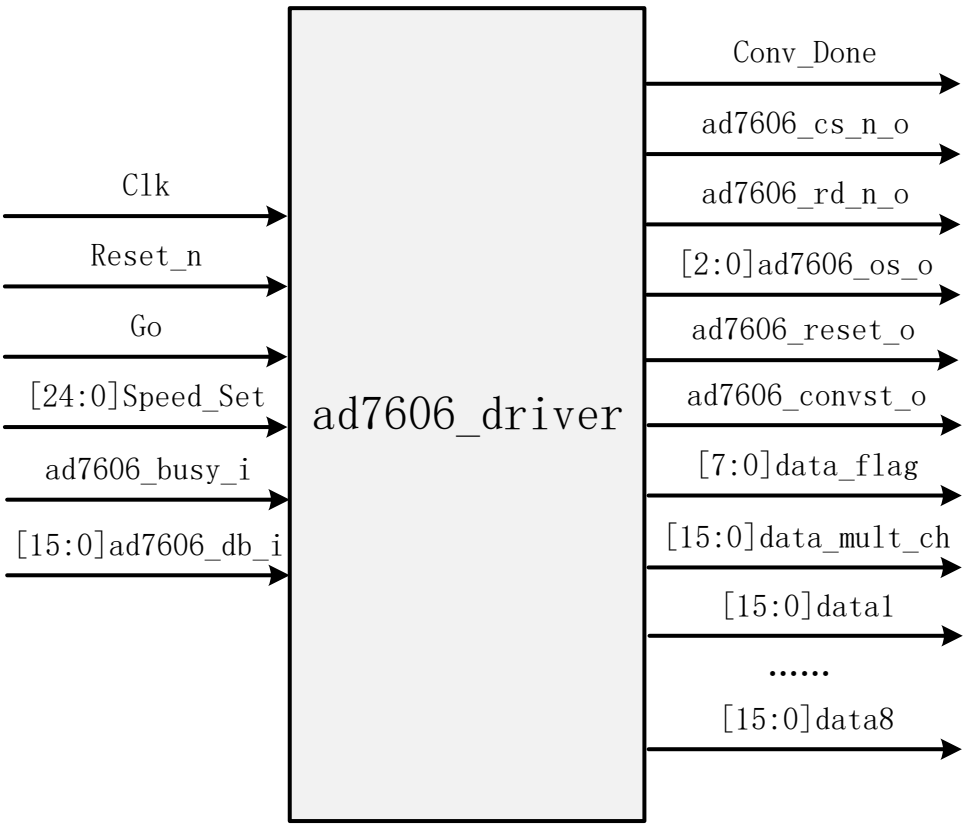


图 59-11 AD7606 控制器驱动模块结构框图

该控制器接口分为四个类，第一类为时序逻辑工作所必须的时钟和复位信号（Clk、Reset_n），第二类是 AD7606 控制器与 AD7606 芯片管脚相连的各种功能控制和数据信号，第三类是设置控制器工作状态/工作数据的用户控制接口，第四类是控制器的结果输出接口。对于用户来说，只需要关注第三类和第四类接口的使用，即可快速高效的使用该控制器来控制 AD7606 芯片完成数据转换。对于该模块的信号说明如下所示。

表 59-6 AD7606 控制器模块信号说明表

类	信号名称	I/O	信号意义
系统信号	Clk	I	模块系统时钟， 为了让采样速率准确，要求为 50MHz
	Reset_n	I	模块复位信号，低电平复位
控制信号	Go	I	采样使能信号，为高电平就使能采样，低电平则在已经开始的一轮采样结束后，停止下一次采样
	Speed_Set[24:0]	I	采样速率控制端口，Speed_Set = 1000000000/20/speed - 1
	Conv_Done	O	一次采样完成标志信号，单时钟周期脉冲信号。每次 8 个通道结果都输出后，产生一个高脉冲信号
数据结果输出端口和标志信号	data_flag[7:0]	O	转换结果有效标志信号，因为 AD7606 有 8 个通道，转换结果是依次输出，并非同时的，所以设置 8 个 Flag 信号，每个通道的结果就绪之后，就产生一个 Flag 信号，通知外部可以取用。另外，如果只关心其中的部分通道， 则只需要关心 data_flag 中对应的位即可
	data_mult_ch[15:0]	O	多通道数据输出端口，该通道 16 位，在不同的时刻，输出不同通道的转换结果，使用时，与 data_flag 信号配合， data_flag 的哪一位出现高脉冲，则代表当前 data_mult_ch 的值为该通道的转换结果。该端口设计的目的是用于往 FIFO、RAM 等存储器中存储结果时使用。
	data1[15:0]…data8[15:0]	O	8 个通道的采样结果输出端口，每个端口分别对应一个模拟通道的采样结果， 使用时与 data_flag 信号配合，每当 data_flag 中的某一位为 1 时，则对应的通道上的 16 位采样结果已经就绪，可以使用。这些端口主要用于每个通道的数据需要分别应用的场合。
ADC 芯片控制信号	ad7606_busy_i	I	ad7606 转换状态标志信号，为高电平则表明 ad7606 当前仍处于转换状态，结果没有更新，如果此时读取，读取的结果就还是前一次的采样转换结果。需要待该信号变为低电平之后，再读取 ad7606 中的数据
	ad7606_db_i[15:0]	I	ad7606 的 16 位数据线，读取时，输出对应通道的转换结果
	ad7606_cs_n_o	O	ad7606 芯片选中控制信号，可以从 AD7606 中读取转换结果时，需要使该信号为低电平
	ad7606_rd_n_o	O	ad7606 转换结果读取信号，该信号的下降沿，AD7606 将特定通道的采样结果送到 16 位数据线上，供外部读取。外部可以在 rd_n 信号的上升沿读取 16 位数据线上的结果
	ad7606_os_o[2:0]	O	ad7606 过采样控制信号，使用过采样可以进一步提高 ad7606 的采样精度，使用过采样会降低 ad7606 的有效转换速率，关于过采样的功能和使用方法，可以参考 ad7606 的 datasheet，默认为 0，则表示不使用过采样。能够运行在最高的转换速率（200Ksps）
	ad7606_reset_o	O	ad7606 的复位信号，复位 ad7606 内部各个功能单元的工作状态
	ad7606_convst_o	O	ad7606 转换开始信号，该信号的上升沿启动 ad7606 内部的采样转换逻辑开始新一轮的采样转换

该控制器在工作时会根据主机的指令对采样频率进行修改，当信号转换完成后便对 ADC 写控制器发出 data_flag 信号，控制其将转换完成数据

data_mult_ch 写入 FIFO 或者 RAM 的同时，也指出了该信号来自哪个通道。通过这两个信号，我们可以实现让 ADC 以特定的采样速率多次采样一个或多个通道的数据。每当 data_flag 中任意一位为 1，则将 data_mult_ch 中的值写入 FIFO 或者 RAM 中。由于 FIFO 和 RAM 等存储器，只有一个数据输入接口，所以这个时候用一个 data_mult_ch 端口分时输出不同通道的采样结果，就比使用 data1~data8 这 8 个 16 位的数据端口分别输出各自通道的采样结果要方便。

例如，将通道 1、2、5、8 的采样结果存入 FIFO。就可以使用下面的写法：

```
module adc_wr_fifo(
  input Clk,
  input Reset_n,
  input [7:0]data_flag;
  input [15:0]data_mult_ch;
  output reg fifo_wrreq,
  output reg [15:0]fifo_data
);
    always@(posedge Clk or negedge Reset_n)
    if(!Reset_n)begin
        fifo_wrreq <= 0;
        fifo_data <= 0;
    else if(data_flag == 8'b1001_0011)begin
        fifo_wrreq <= 1'd1;
        fifo_data <= data_mult_ch;
    end
    else begin
        fifo_wrreq <= 0;
        fifo_data <= fifo_data;
    end
endmodule
```

59.3.5state_ctrl 模块

DDR3 双端口转换模块控制信号的产生以及 ADC 何时启动数据传输都是通过 state_ctrl 模块控制的，该模块的结构框图如下所示：

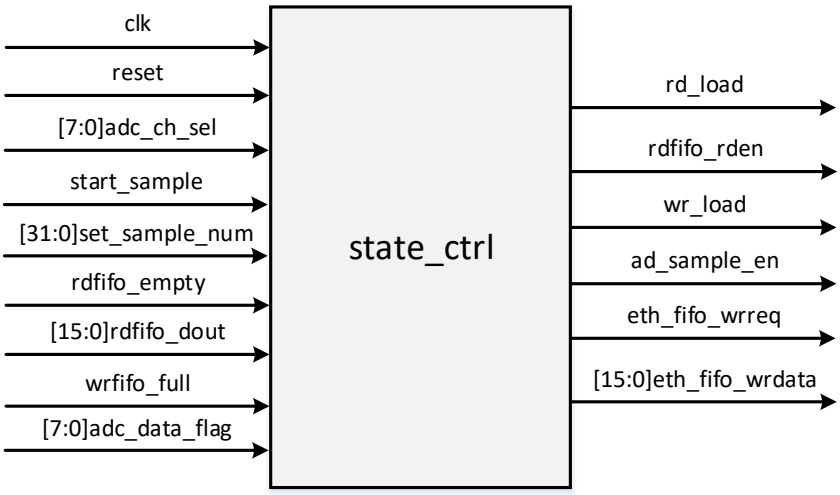


图 59-12 state_ctrl 模块基本结构框图

对于该模块的信号说明如下表 59-7 所示。

表 59-7 state_ctrl 模块信号说明表

信号名称	I/O	信号意义
clk	I	模块时钟信号，与 ADC 采集模块的时钟信号保持一致（50M）
reset	I	模块复位信号，高电平复位
start_sample	I	ADC 模块开始采样标志信号
set_sample_num [31:0]	I	采样数量命令信号
rdfifo_empty	I	读 FIFO 为空标识信号，用于标识当前 FIFO 是否为空（即 FIFO 内有无数据）
rdfifo_dout[15:0]	I	读 FIFO 的数据输出，数据位宽为 16 位
wrfifo_full	I	写 FIFO 的写满标识信号，用于标识当前 FIFO 是否有被写满
adc_ch_sel[7:0]	I	采样通道命令信号
adc_data_flag[7:0]	I	AD7606 控制器输出的转换结果有效标志信号
rd_load	O	DDR3 双端口模块读出源更新信号
rdfifo_rden	O	读 FIFO 的读数据使能控制信号，给高电平表示往 FIFO 读数据，为避免读数据的丢失，确保在 FIFO 非空（rdfifo_empty =0）情况下读数据
wr_load	O	DDR3 双端口模块写入源更新信号
ad_sample_en	O	ADC 采样使能标志信号
eth_fifo_wrreq	O	以太网发送缓存 FIFO 的写请求信号
eth_fifo_wrdata [15:0]	O	写入以太网发送缓存 FIFO 的 16 位数据

DDR3 双端口模块需要的控制信号有：rd_load、wr_load、wrfifo_clk、wrfifo_wren、wrfifo_din、rdfifo_clk、rdfifo_rden。

上述信号中 wrfifo_clk、rdfifo_clk 应该与 state_ctrl 模块的工作时钟保持一致，也就是和 ADC 数据采集模块（ad7606_driver）工作时钟一样为 50M；wrfifo_wren 信号由 AD7606 控制器模块输出的 adc_data_flag 信号与通道命令 ChannelSel 按位与之后，再和采样使能信号 ad_sample_en 相与得到；除去上述

已经得到的控制信号外，state_ctrl 模块还需要产生的控制信号包括：rd_load、wr_load、rdfifo_rden。

根据上述描述，我们可以通过状态机实现该模块的功能。定义状态如下所示：

```
localparam IDLE                 = 4'd0;   //空闲状态
localparam DDR_WR_LOAD          = 4'd1;   //DDR 写入数据更新状态
localparam ADC_SAMPLE           = 4'd2;   //ADC 采样数据状态
localparam DDR_RD_LOAD          = 4'd3;   //DDR 读出数据更新状态
localparam DATA_SEND_START      = 4'd4;   //数据发送状态
localparam DATA_SEND_WORKING    = 4'd5;   //数据发送完成状态
```

下面编写每个状态对应的代码。

1. IDLE 状态

当处于空闲状态时，对 start_sample 采样起始位进行寄存，同时限定其只工作在状态 IDLE，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    start_sample_rm <= 1'b0;
else if(state==IDLE && ddr3_init_done==1'b1)
    start_sample_rm <= start_sample;
else
    start_sample_rm <= 1'b0;
end
```

当产生 start_sample_rm 信号之后跳转到 DDR_WR_FIFO_CLEAR 状态，代码如下所示：

```
begin
    if(start_sample_rm) begin   //DDR 初始化完成并且产生启动采样信号
        state <= DDR_WR_LOAD; //进入写 FIFO 清除状态
    end
    else begin
        state <= state;
    end
end
```

2. DDR_WR_LOAD

进入 DDR 写入数据更新状态之后，开始清除写入 DDR 内的原始数据。设置清除 DDR 内数据的计数器，经过验证，进行 10 拍的延时之后，能够完全清除之前存储在 DDR 内的数据，不影响下一次的传输，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    wr_load_cnt<=0;
```

```
else if(state==DDR_WR_LOAD)
begin
    if(wr_load_cnt==9)
        wr_load_cnt<=4'd9;
    else
        wr_load_cnt<=wr_load_cnt+1'b1;
end
else
    wr_load_cnt<=1'b0;
end
```

我们向 DDR 写入数据时，首先是经过了一个写端的 FIFO 存储的，所以我们写的时候，需要等待 wrfifo_full（写端 FIFO 满信号）的信号拉低，拉低之后，表示可以往 FIFO 里写入数据。DDR 写入数据更新状态代码如下所示：

```
begin
    if(!wrfifo_full && (wr_load_cnt==9))
        state<=ADC_SAMPLE;
    else
        state<=state;
end
```

当处于 DDR_WR_LOAD 状态时，我们需要产生 wr_load 信号，由三拍延时信号拉高提供，之所以提供延迟信号时间为 3 拍，是为了能够准确清除已经写入的数据，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if (reset)
    wr_load<=0;
else if(ddr3_init_done==1'b0)
    wr_load<=1'b1;
else if(state==DDR_WR_LOAD)
    begin
        if(wr_load_cnt==0||wr_load_cnt==1||wr_load_cnt==2)
            wr_load<=1'b1;
        else
            wr_load<=1'b0;
    end
else
    wr_load<=1'b0;
end
```

3. ADC_SAMPLE 状态

进入 ADC 采样数据状态之后，首先设置 ADC 采样个数计数器 adc_sample_cnt，当对应采样通道有效时，也就是 adc_data_flag 信号与 adc_ch_sel 信号相与为高电平时，我们就将 adc_sample_cnt 计数值加 1，对 ADC

采集的数据进行计数。代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    adc_sample_cnt<=1'b0;
else if(state==ADC_SAMPLE)begin
    if(adc_data_flag & adc_ch_sel)
        adc_sample_cnt<=adc_sample_cnt+1'b1;
    else
        adc_sample_cnt<=adc_sample_cnt;
end
else
    adc_sample_cnt<=1'b0;
end
```

当 adc_sample_cnt 达到设定的采样数据个数，ADC 模块数据采集完成，跳转到读出数据更新状态：

```
begin
    if(adc_sample_cnt>=set_sample_num) begin
        state<=DDR_RD_LOAD;
    end
    else
        state<=state;
end
```

当处于 ADC_SAMPLE 状态时，我们还需要产生采样使能信号 ad_sample_en 给到其他模块使用，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    ad_sample_en<=0;
else if(state==ADC_SAMPLE)
    ad_sample_en<=1;
else
    ad_sample_en<=0;
end
```

4. DDR_RD_LOAD 状态

进入 DDR 读出数据更新状态之后，首先设置读取数据更新状态计数器 rd_load_cnt，设置 10 拍的延时，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if(reset)
    rd_load_cnt<=0;
else if(state==DDR_RD_LOAD)
begin
    if(rd_load_cnt==9)
        rd_load_cnt<=4'd9;
```

```
    else
        rd_load_cnt<=rd_load_cnt+1'b1;
end
else
    rd_load_cnt<=1'b0;
end
```

然后等待 rdfifo_empty（读端 FIFO 的空信号）信号拉低，拉低后，表示 FIFO 里已经有被写入数据，此时进入下一状态。在清空（复位）FIFO 的时候，FIFO 的 empty 信号会变高，可以认为在复位 FIFO 时是不允许对 FIFO 进行读操作的，即使读也是不可靠的，等 FIFO 的复位结束后，FIFO 被写入数据后，empty 信号会变低，就允许对 FIFO 进行读操作，然后跳转到 DATA_SEND_START 状态，DDR_RD_LOAD 状态对应的代码如下所示：

```
begin
    if(!rdfifo_empty && (rd_load_cnt==9))begin
        state<=DATA_SEND_START;
    end
    else
        state<=state;
end
```

当处于 DDR_RD_LOAD 状态时，我们需要产生 rd_load 信号，有三拍的延时信号拉高提供，之所以提供的延迟信号时间为 3 拍，是为了保证 rd_load 指令的可靠，代码如下所示：

```
always@(posedge clk or posedge reset)begin
if (reset)
    rd_load<=0;
else if(ddr3_init_done==1'b0)
    rd_load<=1'b1;
else if(state==DDR_RD_LOAD)
begin
    if(rd_load_cnt==0||rd_load_cnt==1||rd_load_cnt==2)
        rd_load<=1'b1;
    else
        rd_load<=1'b0;
end
else
    rd_load<=1'b0;
end
```

5. DATA_SEND_START 状态

进入 DATA_SEND_START 状态之后，state 直接跳转到数据发送状态，代码如下所示；

```
begin
    state <= DATA_SEND_WORKING;
end
```

6. DATA_SEND_WORKING 状态

进入数据发送状态之后，产生 rdfifo_rden 信号，将 DDR 中的数据读取出来，当读出的数据达到需要采集的数据信号时，跳转到 IDLE 状态，完成一次数据传输，代码如下所示：

```
begin
    if(send_data_cnt >= set_sample_num-1'b1) begin
        rdfifo_rden <= 1'b0;
        state <= IDLE;
    end
    else begin
        rdfifo_rden <= 1'b1;
        state <= DATA_SEND_WORKING;
    end
end
```

send_data_cnt 在 rdfifo_rden 有效的情况下进行计数，从而统计从 DDR 中读取的数据个数，代码如下所示：

```
always@(posedge clk or posedge reset)begin
    if(reset)
        send_data_cnt<=32'd0;
    else if(state==IDLE)
        send_data_cnt<=32'd0;
    else if(rdfifo_rden)
        send_data_cnt<=send_data_cnt+1;
    else
        send_data_cnt<=send_data_cnt;
end
```

最后，当 rdfifo_rden 信号为高电平的时候，将 DDR 读出的数据存放至以太网发送缓存 FIFO 中，代码如下所示：

```
always@(posedge clk or posedge reset)
if(reset) begin
    eth_fifo_wrreq <= 1'b0;
    eth_fifo_wrdata <= 'd0;
end
else if(rdfifo_rden) begin
    eth_fifo_wrreq <= 1'b1;
    eth_fifo_wrdata <= rdfifo_dout;
end
else begin
    eth_fifo_wrreq <= 1'b0;
```

```
    eth_fifo_wrdata <= 'd0;
end
```

59.3.6fifo_tx 模块

FIFO 双端口 IP 核（fifo_tx），该模块需要接收从 DDR 读出的 16 位数据，数据经由 fifo_tx 缓存后转换成 8 位数据由以太网发送模块发送出去。ddr3_ctrl_2port 模块中 FIFO IP 的工作时钟为 50M，以太网发送模块的工作时钟 125M，两者数据速率不匹配，使用 FIFO IP 核进行数据存储可以解决过程中会出现的跨时钟域数据交互问题。

在 GOWIN 软件中点击  ，然后在搜索栏中输入 FIFO，在下面搜索结果中找到 FIFO 并双击，操作如下图 59-13 所示。

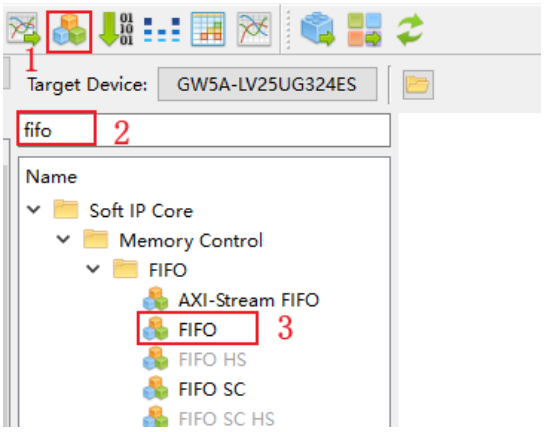


图 59-13 搜索 FIFO IP

双击 FIFO Generator 之后，进入 FIFO 配置界面。

首先将 File Name 和 Module Name 的名称修改为 fifo_tx，然后按照如下步骤进行配置：

1. 写数据位宽设置为 16，写入深度设置为 1024。因为 ADC 模块输出的数据是 16 位的，所以位宽设置为 16。写入深度用户可以修改，理论上只要大于以太网最大帧长度 1472（写入深度大于 1472/2）就可以。
2. 读出位宽设置为 8。设置为 8 是因为以太网发送模块的数据位宽是 8 位的。
3. 读模式选择为“First-Word Fall-Through（FWFT）”，FWFT 模式可以不需要读命令，自动将最新的数据放在 Q 上。
4. 勾选“Read Data Num”和“Write Data Num”。FIFO 数据量计数信号输

出，Read Data Num 和 Write Data Num 分别同步于写时钟和读时钟。

5. 勾选“En_Reset”和“Reset_Synchronization”，使能同步复位。

最终配置界面如下图 59-14 所示：

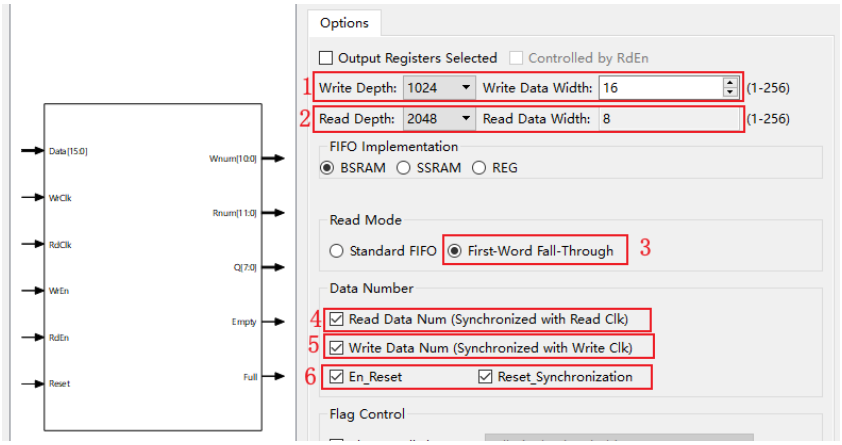


图 59-14 fifo_tx 配置界面

59.3.7eth_send_ctrl 模块

网口发送控制模块（eth_send_ctrl）主要负责配置控制网口发送模块的使能控制信号 pkt_tx_en，并通过 pkt_length 信号对以太网数据帧长度进行控制，该模块的结构框图如下所示。

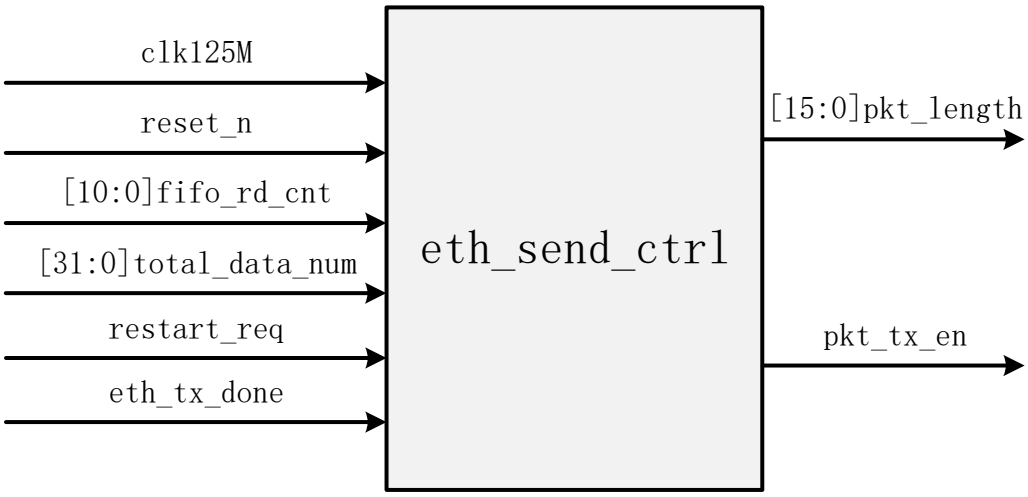


图 59-15 网口发送控制模块

对该模块的信号说明如下所示：

表 59-8 网口发送控制模块信号说明表

信号名称	I/O	信号意义
clk125M	I	模块时钟信号信号，以太网工作时钟 125M

reset_n	I	模块复位信号，低电平复位
fifo_rd_cnt [10:0]	I	FIFO 读数据计数
total_data_num [31:0]	I	需要采集的数据个数
restart_req	I	开始采样请求信号
eth_tx_done	I	以太网一个包传输完成标志信号
pkt_length [15:0]	O	以太网需要传输的数据长度
pkt_tx_en	O	以太网传输使能信号

以太网帧最大长度 1518 字节（数据段 1500 字节），其中数据段 1500 字节，还包括 20 字节 IP 报文头部和 8 字节 UDP 报文头部，所以以太网帧发送的 ACM7606 采集的数据最大长度为 1472 字节（1500-20-8）。

该模块可以通过编写状态机代码方式实现功能，下面将对每个状态的代码进行介绍。

状态 0，得到 pkt_length 信号的初始值。这里需要注意的是经过数据位扩展模块输出的数据是 16 位的，每个数据占据 2 个字节，所以发送 N 个采样数据，则以太网需要发送 2*N 个字节数据。当产生开始采样请求 restart_req 之后，系统开始采样，同时，将需要采集的数据个数 total_data_num 左移一位（相当于乘以 2），得到实际需要以太网传输的数据，当数据大于 16'd1472 时，设置 pkt_length 为最大传输长度 1472；当数据大于 0 时，pkt_length 等于 total_data_num 乘以 2。给 pkt_length 赋初值之后，跳转到状态 1，代码如下所示：

```
begin
    if(restart_req)begin
        data_num <= total_data_num;
        if((total_data_num << 1) >= 16'd1472)begin
            pkt_length <= 16'd1472; //一个数据 2 个字节
            state <= 1;
        end
        else if((total_data_num << 1) > 0)begin
            pkt_length <= total_data_num << 1; //一个数据 2 个字节
            state <= 1;
        end
        else begin
            state <= 0;
        end
    end
end
```

状态 1，当 FIFO 计数信号 fifo_rd_cnt 的数值满足一帧数据帧长度时，产生 pkt_tx_en 信号，以太网发送模块开始读取 FIFO 中的数据，代码如下所示：

```
begin
    if(fifo_rd_cnt >= (pkt_length -2)) begin
        pkt_tx_en <= 1'd1;
```

```
        state <= 2;
    end
    else begin
        state <= 1;
        pkt_tx_en <= 1'd0;
    end
end
```

状态 2，当以太网一个包传输完成之后，产生 eth_tx_done 信号，此时剩下需要传输的数据 data_num 应该减去 pkt_length 的一半，这是因为 ADC 采集的数据是 16 位的，但是以太网每次只传送 8 位数据，所以以太网实际传输的数据应该是 ADC 采集到的数据的一半，代码如下所示：

```
begin
    pkt_tx_en <= 1'd0;
    if(eth_tx_done)begin
        data_num <= data_num - pkt_length/2;
        state <= 3;
    end
end
```

状态3，设置以太网的帧间隙时间间隔。本次实验使用的千兆以太网，其相邻的两帧之间的最小间隔时间为 96ns，在设置的时候只要大于 96ns 就行，本次实验设置 128 个时钟周期，也就是 1024ns。当设置的时间比较小的时候，虽然以太网传输速率会加快，但是会增加电脑端解析数据包的压力，当时间过大，又会影响以太网船速速率，所以在设置的时候需要设定一个比较合理的值。

```
if(cnt_dly_time >= cnt_dly_min)begin
    state <= 4;
    cnt_dly_time <= 28'd0;
end
else begin
    cnt_dly_time <= cnt_dly_time + 1'b1;
    state <= 3;
end
```

状态 4，进行连续的包传输。当剩下的需要以太网传输的数据（data_num*2）大于包传输最大数据 1472 时，使 pkt_length 为 1472，使 pkt_length 为 1472，然后回到状态 1 继续传输；当剩下需要传输的数据不足包传输最大数据 1472 时，pkt_length 为剩下的需要传输的数据 data_num*2，然后回到状态 1 传输剩下的数据，代码如下所示：

```
begin
    if(data_num * 2 >= 16'd1472)begin
        pkt_length <= 16'd1472;
        state <= 1;
```

```
    end
    else if(data_num * 2 > 0)begin
        pkt_length <= data_num * 2;
        state <= 1;
    end
    else begin
        state <= 0;
    end
end
```

模块设计完成之后，只需要在顶层文件中对各个模块之间的接口信号进行连接，完整的顶层文件代码请自行查看例程文件。

59.4板级验证

经过以上工作，代码设计部分的任务已经全部完成，接下来就可以进行板级验证了。本次实验的板级验证环节，主要验证：通过电脑上的网络调试助手，将命令帧进行发送， 然后通过高云开发板上的以太网芯片接收，随后将接收到的数据转换命令，从而实现对 AD7606 采样频率、数据采样个数以及采样通道的配置。配置完成之后，AD7606 开始采集数据，将 AD7606 采集的数据通过网口传输到电脑。电脑端将接收到的数据进行保存，然后通过 MATLAB 进行进一步的分析。针对本次实验，我们也提供有专门的上位机软件，用户只需要在软件界面进行参数配置，便可以实时观察到数据波形变化，使用起来非常方便。

59.4.1硬件连接

将 ACM7606 模块、网线、下载器、电源线依次连接在开发板上，需要注意 ACM7606 模块连接正确后右边会多出 6 组排针，由于视觉缘故，这里很容易连接错误，，还需要给 AD7606 连接信号源，这里我们连接的是 AD7606 的通道 1，信号源给的是 200hz，vpp=5V 的正弦波，整体的硬件连接图如下图 59-16 所示。

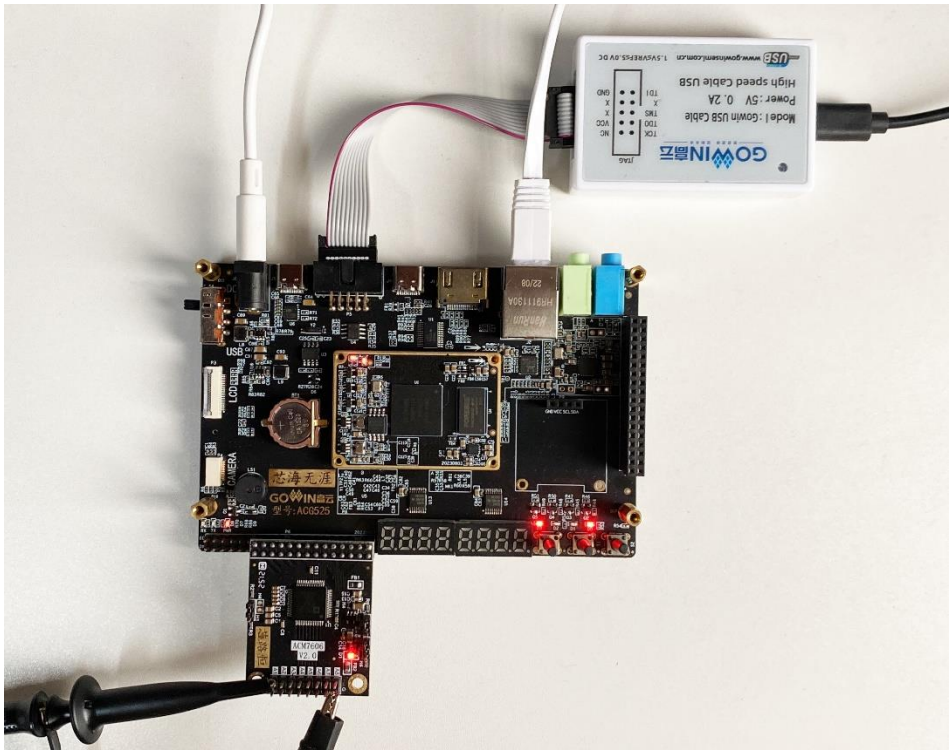


图 59-16 硬件连接图

连接完成之后，下载生成的数据流文件。

59.4.2修改电脑 IP 地址

本次实验设定了目标 IP 地址（PC 端）为 192.168.0.3，用户需要将自己电脑上的以太网 IP 地址修改为该地址，在本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址。如下图 59-17 所示。

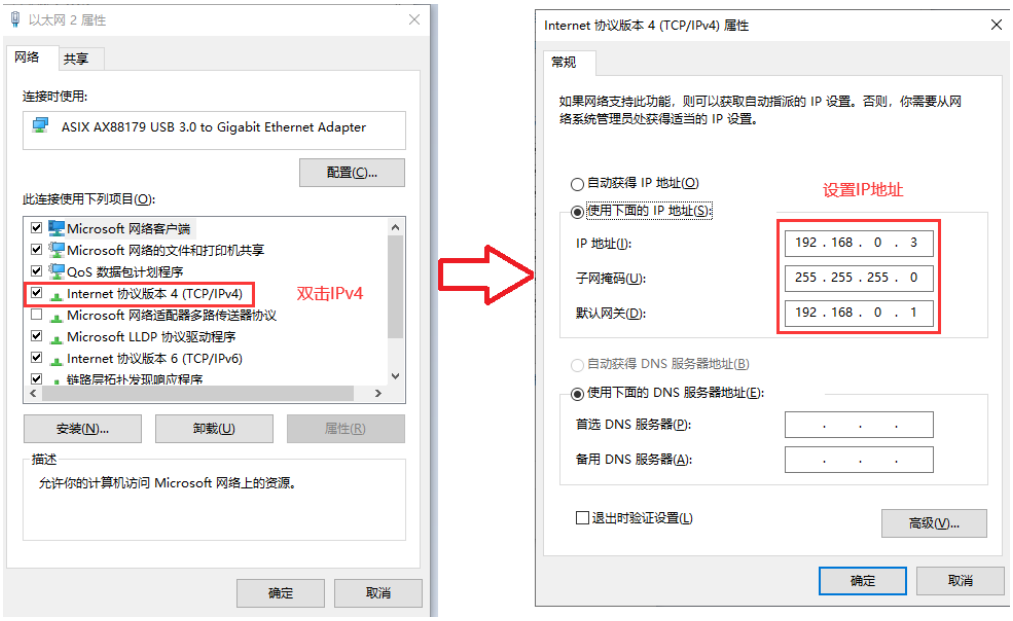


图 59-17 修改电脑 IP 地址

59.4.3绑定 ARP

本工程不支持 ARP 协议，只能通过静态绑定的方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

关于 ARP 的绑定请查看以下帖子内容：

以太网通信静态 ARP 绑定方法与常见问题解决方案

http://www.corecourse.cn/forum.php?mod=viewthread&tid=28645

(出处: 芯路恒电子技术论坛)

59.4.4网络调试助手通信

完成上述操作之后，首先需要打开网络调试助手发送指令去配置，按照如下所述设置各项参数。网络调试助手软件读者可以在常用软件文件夹下找到。

1. 选择协议类型为 UDP。
2. 设置本地 IP 地址为 192.168.0.3。
3. 设置本地端口号为 6102。
4. 点击【连接】按钮以创建连接，连接上后该按钮为红色“断开”字样。
5. 连接上后，设置目标主机为 192.168.0.2，目标端口为 5000。
6. 发送设置中数据类型设置为 hex 格式

7. 点击“接收保存到文件”这几个字，在弹出的界面中设置文件路径、文件名称，如下图 59-18 所示。这样在数据接收完成之后会保存一个数据文件。方便后面进行分析。

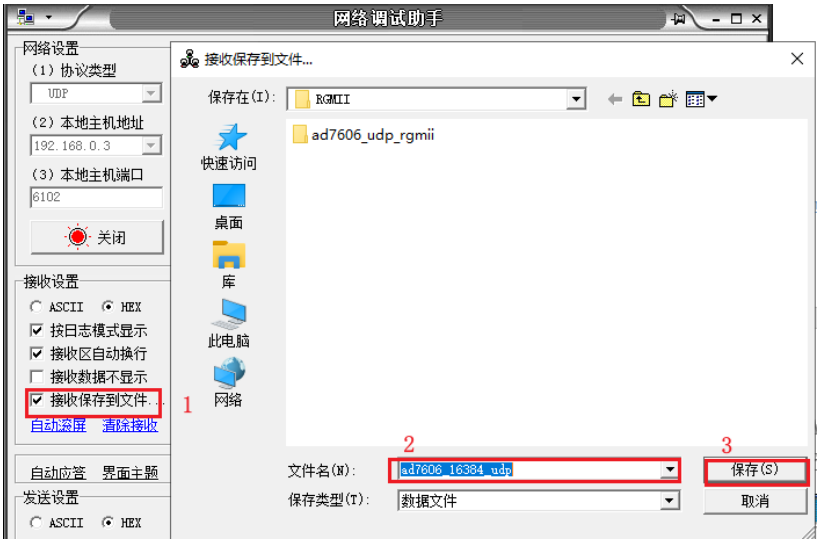


图 59-18 设置保存文件

在前面接收转命令模块中介绍到数据帧格式对 AD7606 的四个寄存器进行配置。

例如，PC 端要设置采样数据个数(DataNum 寄存器，地址为 2)为 16384(0x4000)个，发送数据帧内容：0x55 0xA5 0x02 0x00 0x00 0x40 0x00 0xF0。

PC 端要设置采样速率(ADC_Speed_Set 寄存器，地址为 3)为 200K，则对应的 ADC_Speed_Set 值为 1000000000/20/200000 - 1 = 249（0x000000F9）个， 则发送的数据帧内容为：0x55 0xA5 0x03 0x00 0x00 0x00 0xF9 0xF0。

当上述设置都设置完成后，就可以向 0 号寄存器写入任意值，来开始一次采样传输了。数据帧内容可以为 0x55 0xA5 0x00 0x00 0x00 0x00 0x00 0xF0，这里需要注意的是 0 号寄存器必须放在最后，因为 0 号寄存器负责启动 ADC，ADC 在未配置完全的情况下开始启动，数据很容易输出错误值。

开始传输数据帧命令发送完成之后，AD7606 就能实现以 200K 的采样速率，对 1 个通道进行采样（本次实验以通道 1 为例），共采集 16384 个数据。四个寄存器对应的配置如下表 59-9 所示。

表 59-9 AD7606 数据帧格式配置表

寄存器名称	数据帧数据
DataNum	55 A5 02 00 00 40 00 F0
ChannelSel	55 A5 01 00 00 00 01 F0
ADC_Speed_Set	55 A5 03 00 00 00 F9 F0

RestartReq	55 A5 00 00 00 00 00 F0

配置成网络调试助手发送的数据格式如下：

```
55A50200004000F055A50100000001F055A503000000F9F055A50000000000F0
```

最终的网络助手配置界面如下图 59-19 所示：

图 59-19 网络调试助手配置界面

点击发送之后可以看到网络调试助手在不断的接收数据，如下图 59-20 所示。从图中可以看出一共接收到 32768 个数据，这是因为设置的 ADC 采样数量为 16384，ADC 采样数据是 16 位的，以太网是以字节（8 位）为单位进行发送的，所以通过以太网接收到字节数应该是 16384*2 个数据，这也就是说明接收到的数据的个数没错。

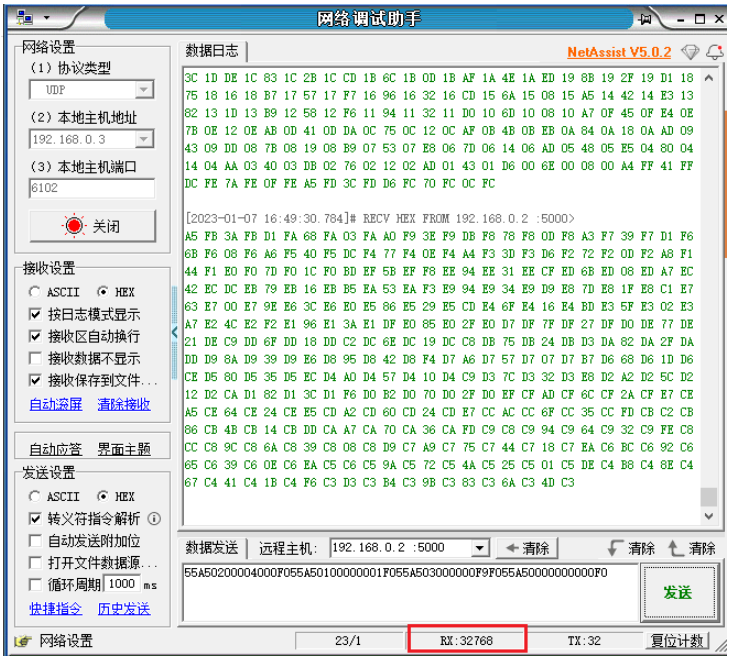


图 59-20 网络调试助手接收数据

59.4.5MATLAB 图像绘制

前面通过网络调试助手得到了 ADC 采集到的数据文件，然后我们就需要对采集到的数据进行分析，本次实验使用 MATLAB 软件进行分析。使用 MATLAB 软件需要读者电脑安装了 MTALAB，如果已经安装好了 MTALAB 软件，则可以双击我们提供的 ADCdata_to_wave_v2_2.m 文件，在打开方式里选择以 MATLAB 打开，该文件的路径： 02_设计实例/ ad7606_udp_rgmii.rar，将压缩包解压便可以看到该文件。文件打开之后，读者需要将代码中文件路径修改为你保存的数据文件路径，随后点击运行便可以直观的看到数据是否正确，MATLAB 操作如下图 59-21 所示，得到的波形图如下图 59-22 所示。

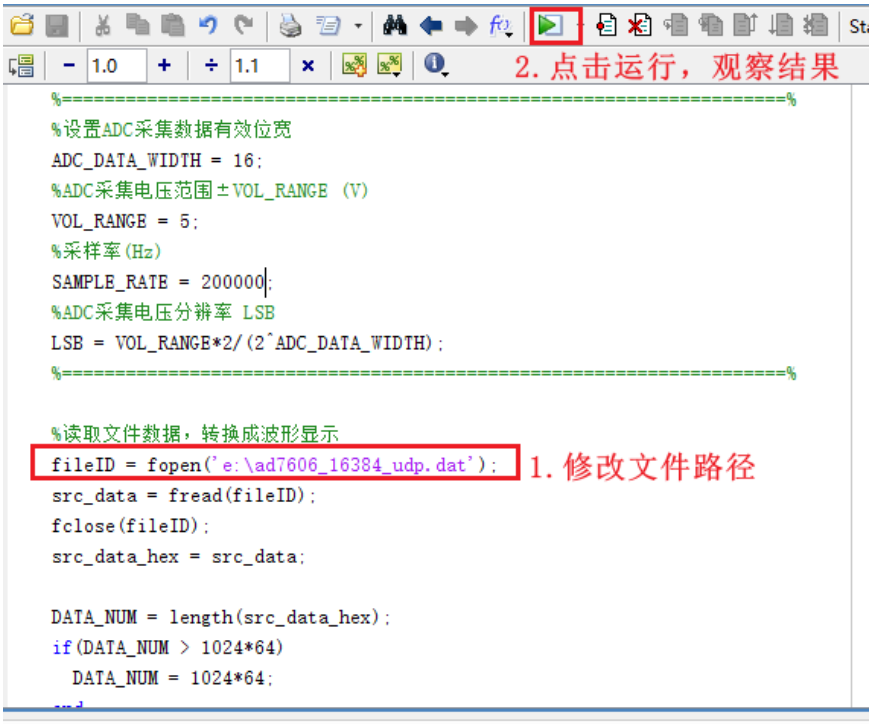


图 59-21 修改文件路径并运行

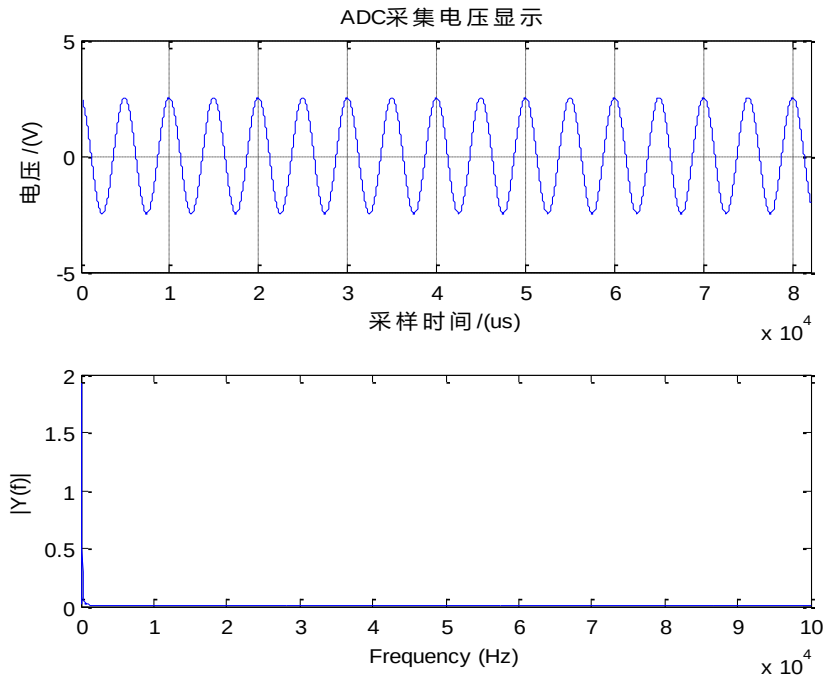


图 59-22 MATLAB 分析波形图

前面我们提到过本次实验提供的信号源为 200hz，Vpp 为 5V 的正弦波（正负 2.5V），与 MATLAB 分析出来的波形一致，说明我们本次实验成功。

59.4.6数据采集上位机通信

前面通过网络调试助手采集数据时，每次保存数据都需要重新点击“接收保存文件”一栏，修改寄存器参数的时候，都需要重新计算，然后发送命令，修改之后也不能直接实时观察到数据波形，使用起来不是很方便。基于上述问题，我们设计了上位机软件“小梅哥控制台 For ADC 采集”进行数据采集，上位机内部直接对命令进行了构建，用户只需要在界面上对采样参数进行设置，便可以实时观测到数据变化，读者可以在论坛上下载最新版本的软件，链接如下：

http:/www.corecourse.cn/forum.php?mod=viewthread&tid=29224(出处:%20 芯路恒电子技术论坛)

双击上位机软件，初始界面如下图 59-23 所示。

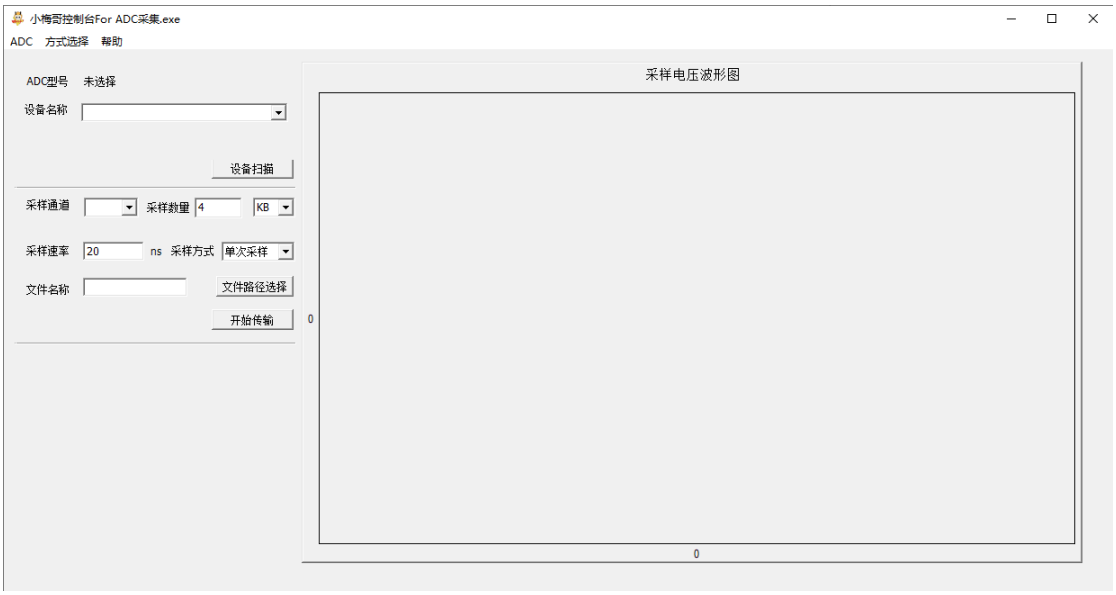


图 59-23 上位机软件初始界面显示

本次实验使用该软件的方式如下所示：

1. 点击 ADC，选择 ACM7606。
2. 点击方式，选择网口，可以看到主机 IP（PC 端）和目的 IP（FPGA）以及对应的端口号。主机 IP：192.168.0.2，主机端口号：6102；目的 IP：192.168.0.3，目的端口号：5000。
3. 选择完成之后，可以看到对采样通道、采样数量等都已经设置了初始值（默认设置的采样率为 ADC 模块的最大采样率），用户可以根据自己的需求进行修改。

4. 点击网络连接。

5. 点击开始传输之后，可以看到在右边采样电压波形图界面可以直观看到波形图，如下图 59-24 所示。需要注意的是波形图的横坐标对应的不是频率，而是采样数量。

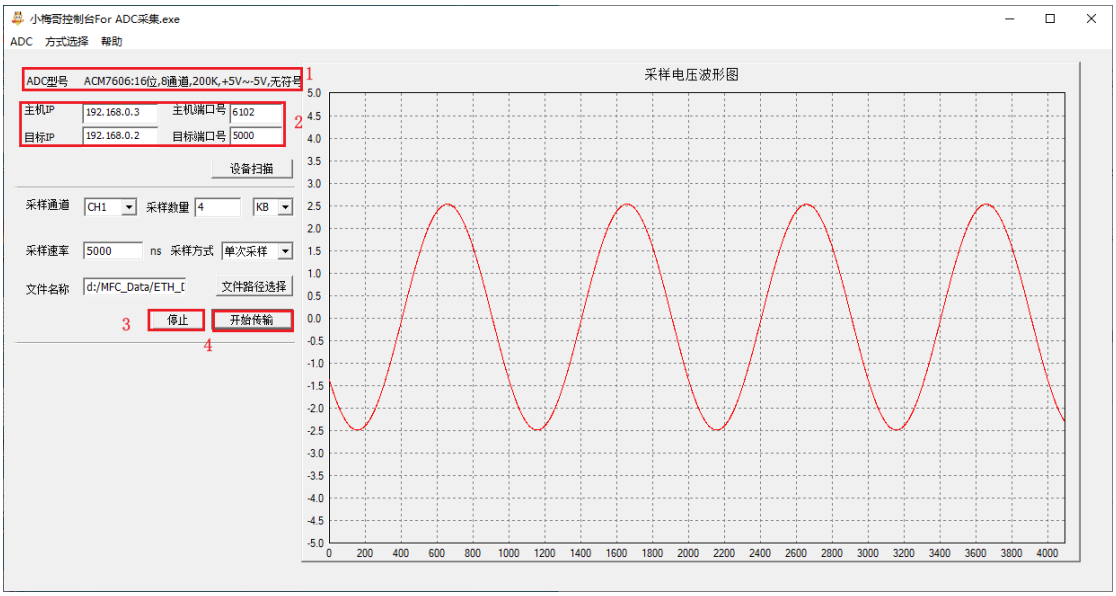


图 59-24 数据采集上位机显示图

通过上位机采集到的数据文件保存在 d:/ MFC_Data 文件夹下，后续可以通过 MATLAB 软件进行进一步的分析，通过 MATLAB 分析的波形图如下图 59-25 所示。从图中可以看出，采集到的数据的频率为 200hz，电压在正负 2.5V 左右，与我们输入的信号一致。

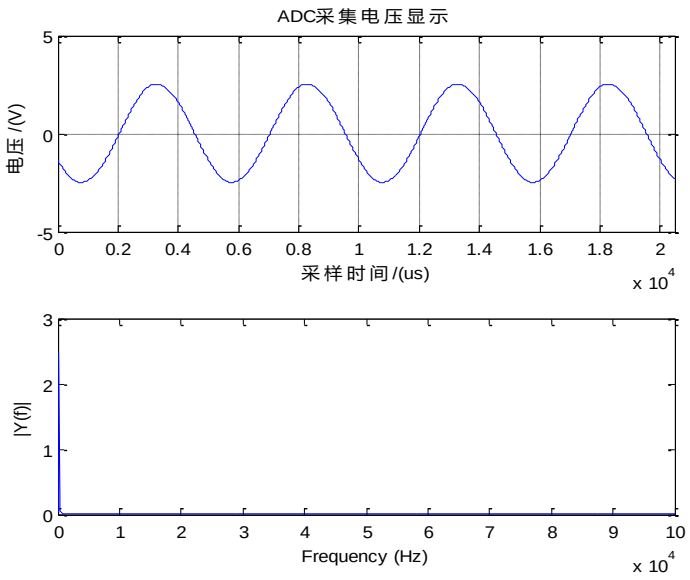


图 59-25 MATLAB 进一步波形分析图

59.5思考与总结

本次实验介绍了基于 AD7606 的千兆以太网收发，用户通过网口调试助手以太网帧向开发板发送指令数据配置 AD7606 的四个寄存器，以此控制 ADC 进行采样，并将数据缓存后再组成以太网帧，经由网口发送至电脑，借由网口调试工具对数据进行查看。如果使用我们提供的上位机软件，则不需要自己设置命令，只需要在界面上修改相关参数，便可以在右边的波形显示界面实时观察到波形变化。

本次实验涉及时钟域的转换，以及采集数据位宽的转换，完成这些转换需要对 FIFO 有一定的了解，想要对 FIFO 进一步了解可以参看前面“IP 核使用之 FIFO”一章的内容。本次实验也要求读者对以太网协议有足够的认识，想要进一步理解以太网功能原理可参看前面以太网相关章节的内容。

60 基于 MIPI OV5647 摄像头的 HDMI 显示设计

61 基于 MIPI OV5647 摄像头的以太网图像传输设计

62 USB2.0 转 UART 数据回环设计实现