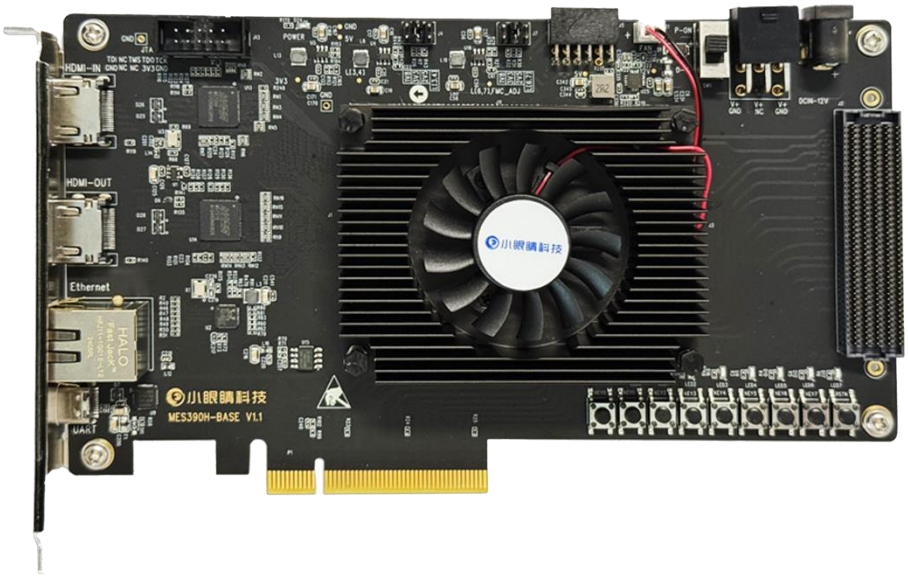
泰坦 390H 开发板
FPGA 通用实验指导手册 V1.1

小眼睛科技 Titan2 系列

泰坦 390H（PG2T390H）开发板实验教程

紫光同创 Titan2 系列 PG2T390H 开发平台



深圳市小眼睛科技有限公司

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2025/8/1	初始版本
V1.1		修订版本

公司名称：深圳市小眼睛科技有限公司

官方淘宝店铺：小眼睛半导体

淘宝店铺链接：https://shop372525434.taobao.com

地址：深圳市宝安区西乡街道 F518 时尚创意园

官方网址：www.meyesemi.com

* 请关注小眼睛科技官方微信公众号“小眼睛 FPGA”

我们将不定期更新 FPGA 学习资料和行业资讯

* 请关注抖音/视频号“小眼睛 FPGA” /B 站 UP 主“小眼睛半导体”

海量 FPGA 教学视频免费学习

* 加入 FPGA 开发者技术交流 QQ 群（442106123）

与 4000+FPGA 开发者实时沟通

* 小眼睛 FPGA 已入驻电子发烧友 FPGA 开发者技术社区

FPGA 相关帖子资源丰富，更有 600W+开发者及小眼睛 FPGA 技术团队

为大家解决技术难题，让您售后无忧

网址：https://bbs.elecfans.com/xfpga

* 微信公众号

* 抖音号

* 视频号

目录

1. FPGA 开发工具使用

1.1. 实验简介

实验目的:

PDS 软件的安装和环境搭建。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

1.2. 实验原理

1.2.1. PDS 软件的安装

1.2.1.1. 软件简介

Pango Design Suite 是紫光同创基于多年 FPGA 开发软件技术攻关与工程实践经验而研发的一款拥有国产自主知识产权的大规模 FPGA 开发软件，可以支持千万门级 FPGA 器件的设计开发该软件支持工业界标准的开发流程，可实现从 RTL 综合到配置数据流生成下载的全套操作。

1.2.1.2. 支持平台

软件工具	Windows 操作系统
ADS 综合工具、OEM 综合工具、PDS 后端工具	Windows 7,10,11、

1.2.1.3. 软件安装

所有版本的安装均一致，文中以 PDS_2022.1 版本为例，将软件安装在 C:\pango\PDS_2022.1（软件默认安装路径），若选择自定义安装路径需注意路径不可出现中文和特殊字符。软件安装完成后，会在桌面以及程序菜单中添加快捷方式 PangoDesign Suite2022.1；在程序菜单 Pango Design Suite2022.1 文件夹中包含 Pango Design Suite、软件卸载的快捷方式 Uninstall、程序附件 Accessories 以及软件文档 Documents。

本章节安装流程以 Pango Design Suite 2022.1 Windows 版本安装进行说明，下面将详细介绍 PDS 安装过程。

1.2.1.4. 安装程序

首先关闭电脑所有杀毒软件，否则杀毒软件有可能会拦截一些组件，造成安装失败或功能缺失等不确定结果；将压缩包解压出来（注意：解压目录的路径名称只能够包含字母、数字、下划线，不要出现中文与特 殊字符，否则安装程序有可能出问题）

双击安装包中的安装程序 Setup.exe，启动安装程序如下图所示：

图 1.2-1

启动安装程序后的界面如下：

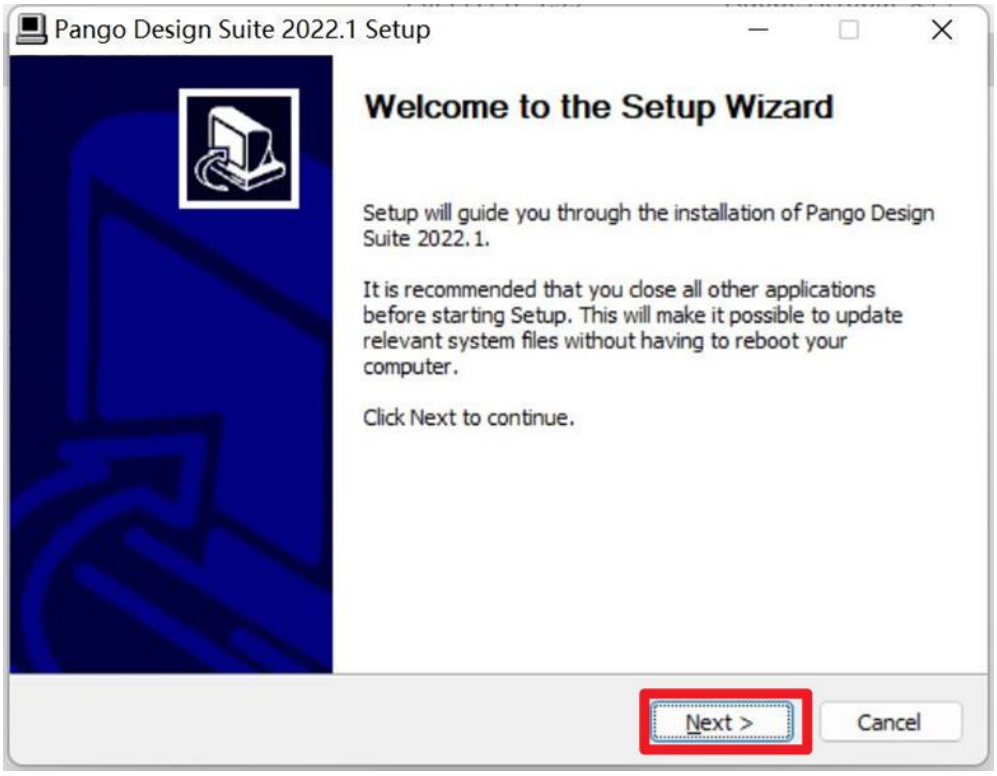


图 1.2-2

点击“Next”，跳转至许可协议对话框：

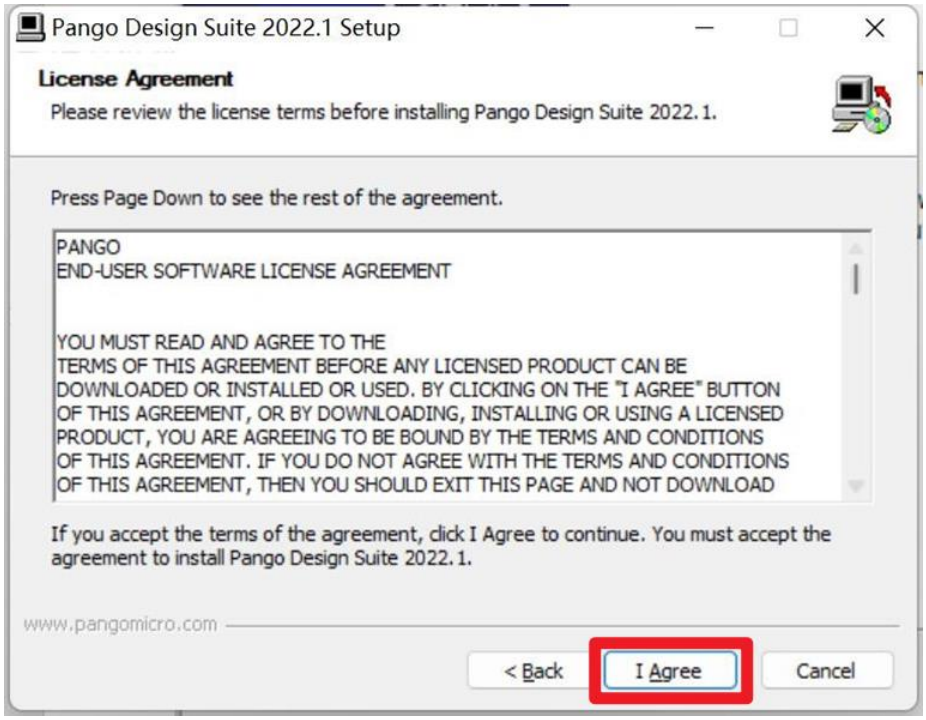


图 1.2-3

选择接受许可协议，点击“I Agree”按钮，进入选择安装路径选择框，如下图所示，默认安装路径为 C:\pango\PDS_2022.1，建议采用默认路径，若使用自定义安装路径需注意路径不要出现中文和特殊字符。

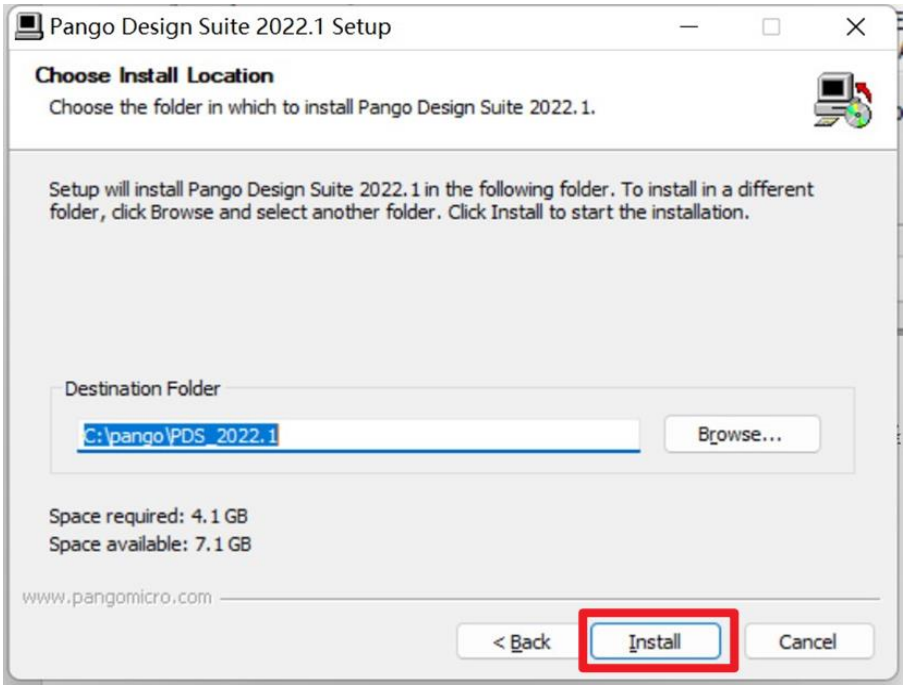


图 1.2-4

直接点击“Install”，则跳转到安装界面。

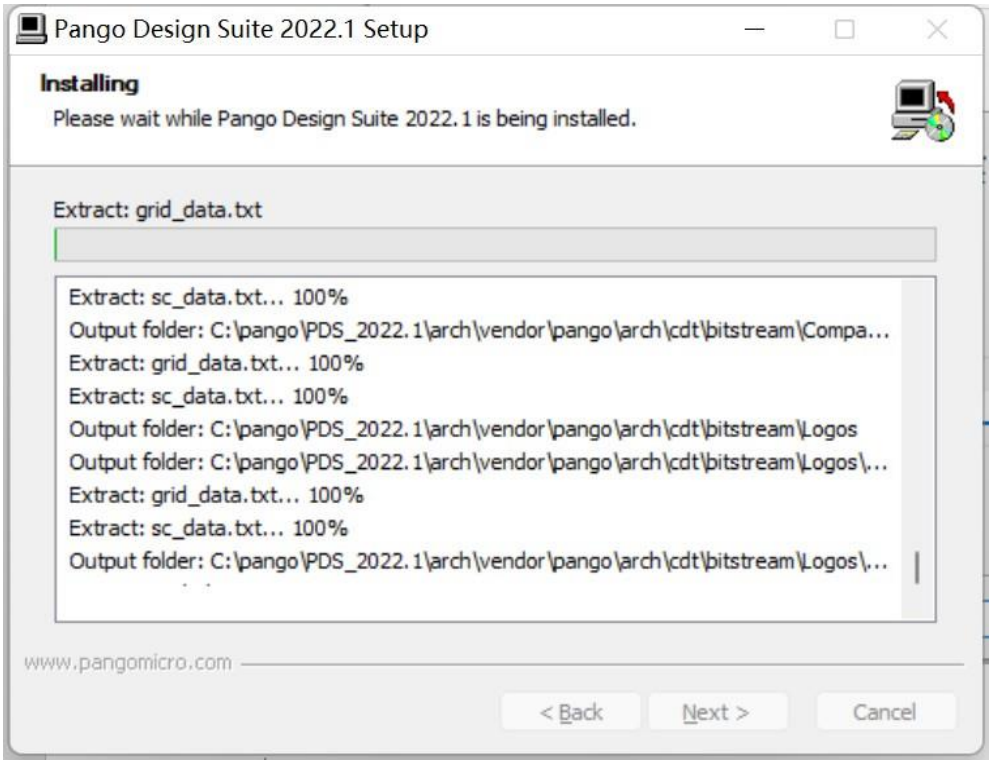


图 1.2-5

耐心等待至完成全部安装过程，点击“Finish”，安装完毕。

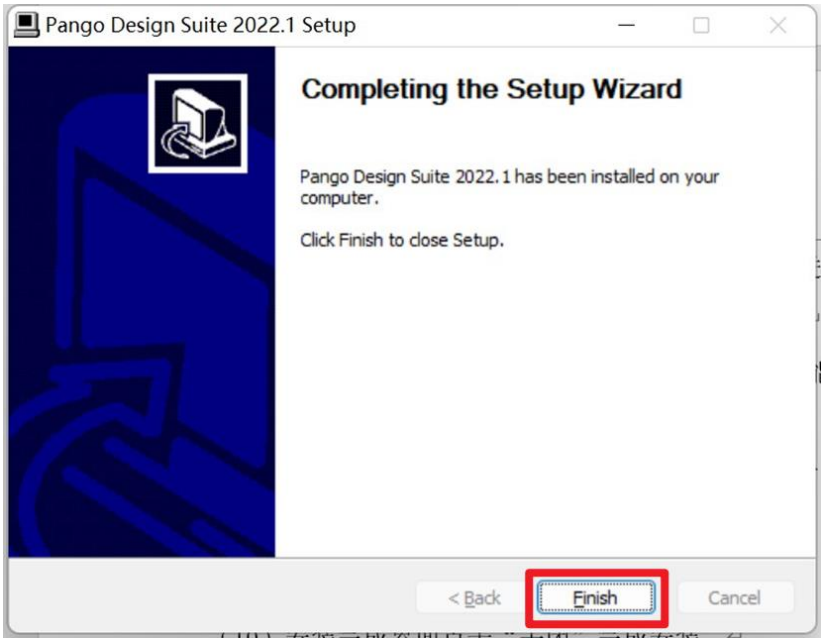


图 1.2-6

安装完成后，会提示是否需要安装运行库 vcredist_VS2017.exe。若电脑之前未安装过则需要安装此运行库后才能运行 PDS，点击“是”按钮进行安装；若电脑之前已安装过此运行库则无需再次安装，点击“否”按钮不进行安装即可。（如果没有这个提示，可以不管）

注：如果不确定，建议点击“是”进行安装，否则可能导致 PDS 无法运行。

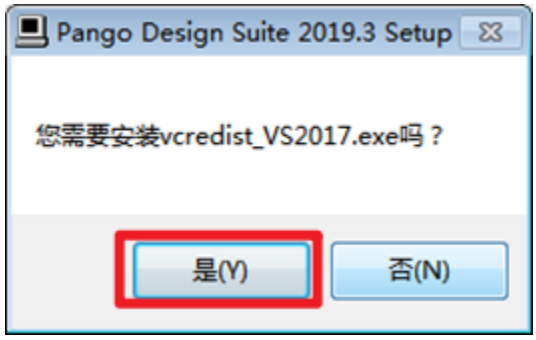


图 1.2-7

点击“是”进入运行库安装界面，选择同意许可条款和条件，点击安装按钮进行安装。

图 1.2-8

安装完成界面点击“关闭”完成安装。

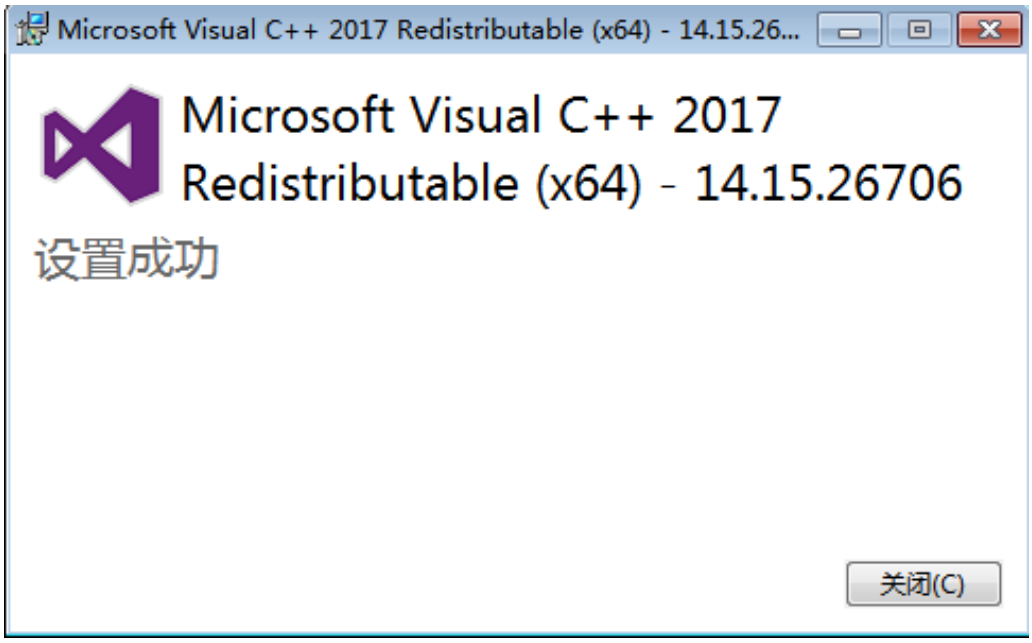


图 1.2-9

完成安装后，会提示是否需要安装 USB Cable Driver。安装点击“是”，否则可能导致下载器无法正常使用。

图 1.2-10

点击“是”进入驱动程序安装界面：

图 1.2-11

点击“下一步”进入安装驱动程序许可协议界面。点击“我接受这个协议”，然后点击”下一步“。

图 1.2-12

点击“完成”，完成驱动程序的安装。

图 1.2-13

完成安装后，会提示是否需要安装 ParellelPortDriver。安装点击“是”。ParellelPortDriver 的安装程序保存地址：PDS 软件安装目录\parellel_driver\Win32\InstallDriver.exe

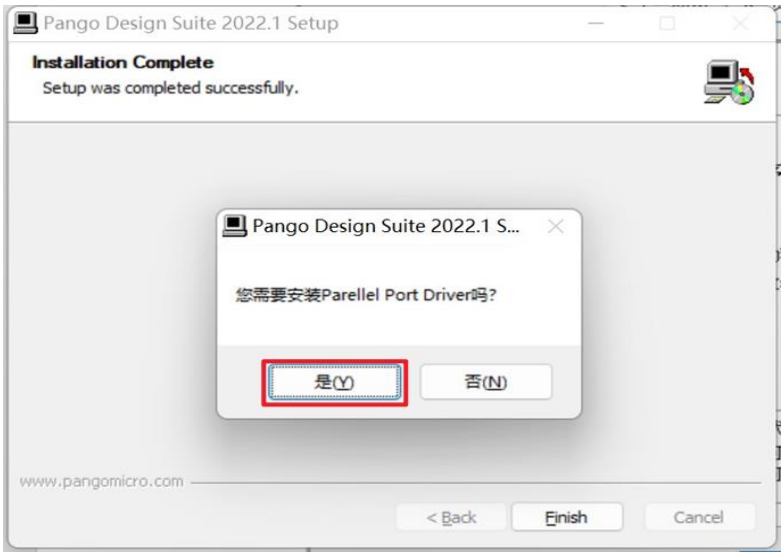


图 1.2-14

点击“是”进行并口驱动程序安装。

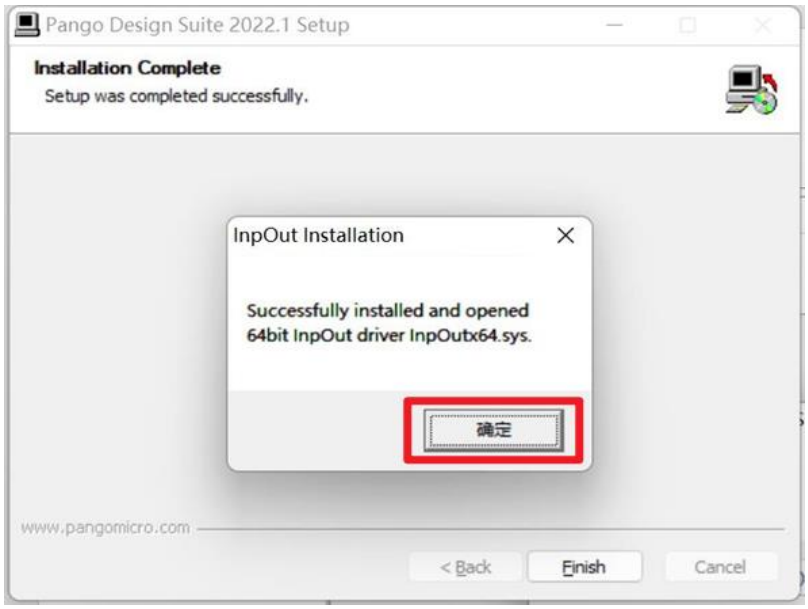


图 1.2-15

完成后点击“确定”，结束安装。在桌面上看到如下图标:

图 1.2-16

1.2.1.5. License 关联，环境变量设置

软件完成安装后，Pango Design Suite 需要 License 文件才能正常使用，若安装 Lite 版本软件无需 License。License 文件可联系相关供应商或客服获取。

本教程使用软件版本开发语言支持 Verilog，若使用 VHDL 需更新支持 VHDL 的软件版本

为方便管理 license 文件，建议在 PDS 软件安装目录下新建一个 license 文件夹存放 license 文件。

首先在电脑上打开运行窗口(win+r)，接着在窗口内输入 sysdm.cpl 然后回车。在系统属性界面内选择高级，然后点击环境变量，进行设置。

图 1.2-17

或者直接打开系统环境变量

图 1.2-18

1.2.1.6. PDS license 环境变量设置

查看下载的资料包内PDS license 文件路径，建议将license文件放置在PDS软件文件夹地址内，这里以：D:\pango\license\pds_node-locked.lic地址为例设置环境变量：

变量名：PANGO_LICENSE_FILE

变量值：D:\pango\license\pds_node-locked.lic

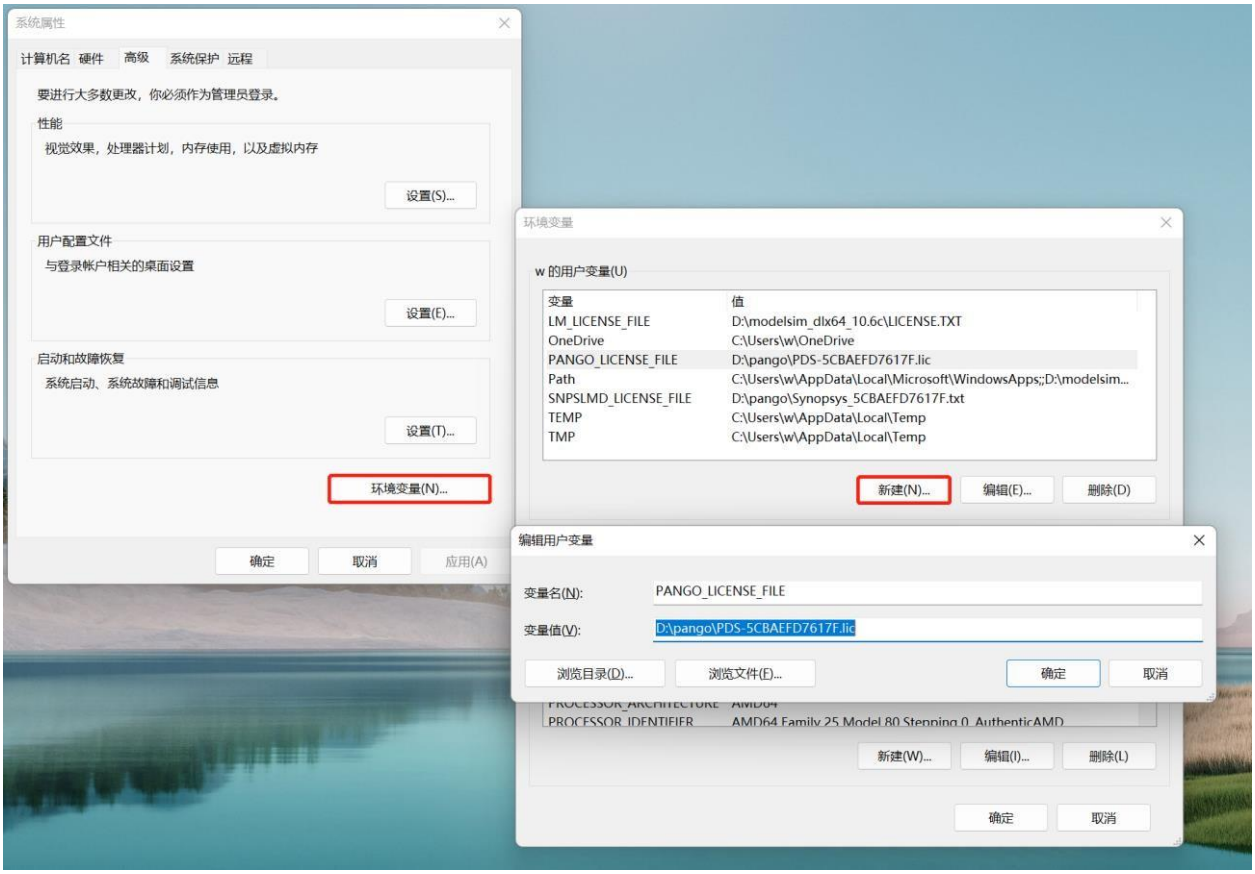


图 1.2-19

如果使用开发板资料提供的通用License，还需要安装tap-windows软件。其主要作用是用来生成虚拟网卡，具体如下所示：

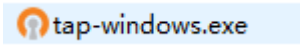


图 1.2-20

双击运行，全部保持默认。

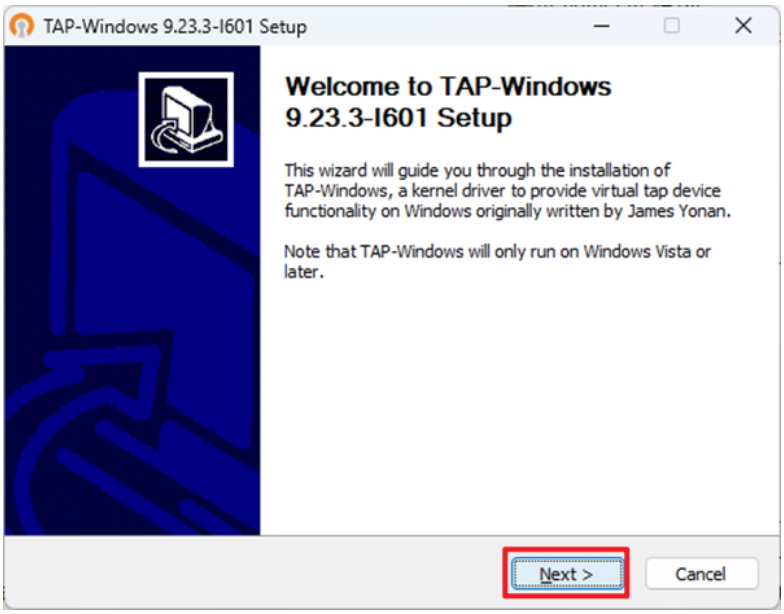


图 1.2-21

点击 Next。

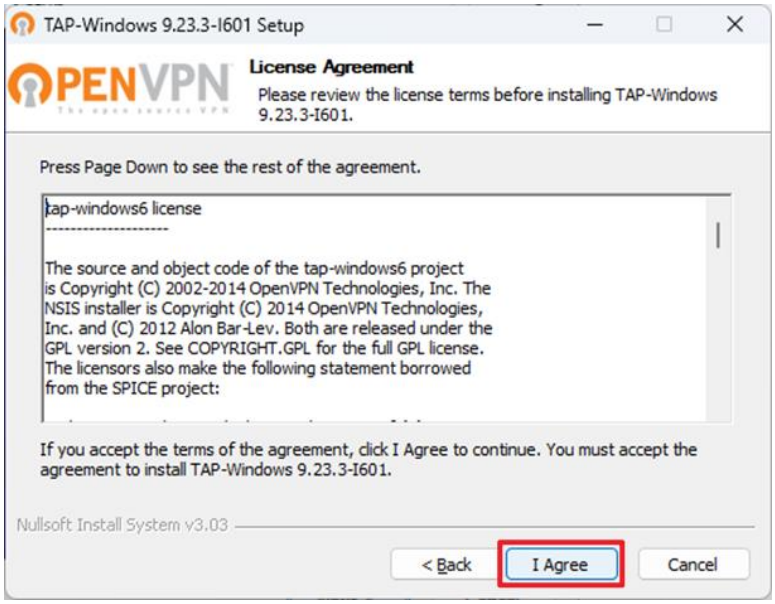


图 1.2-22

点击 I Agree。

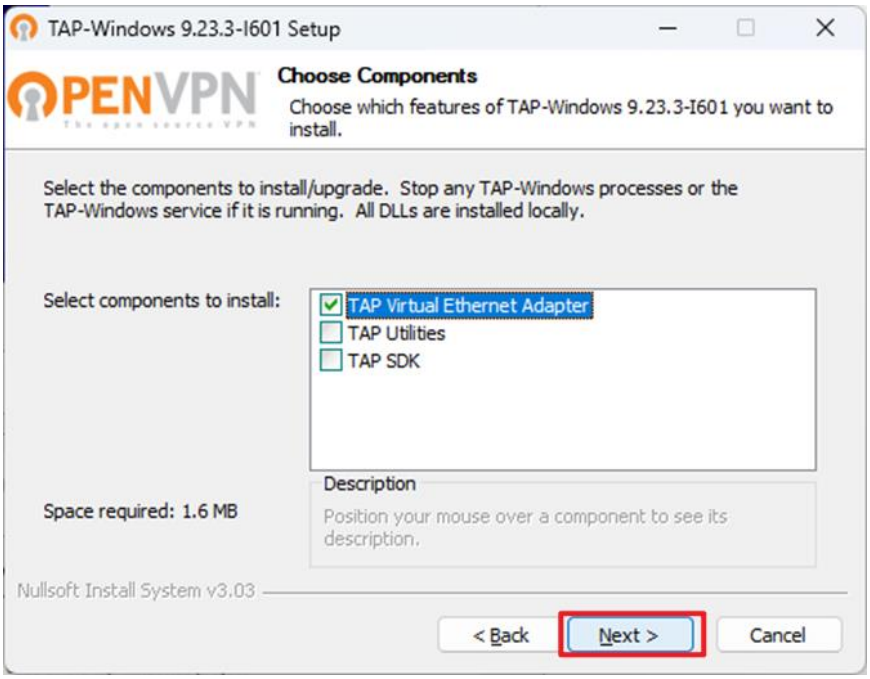


图 1.2-23

保持默认，点击 Next。

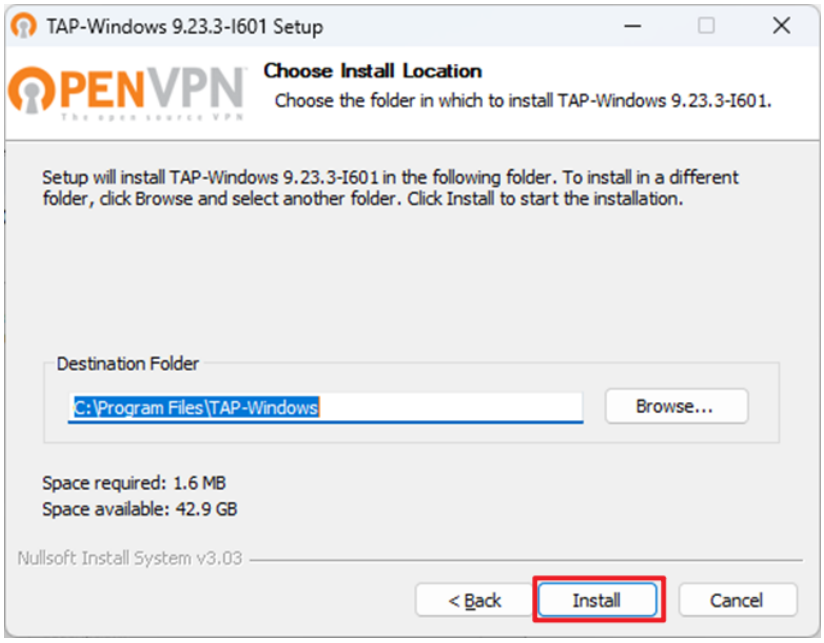


图 1.2-24

安装路径保持默认，点击Install，等待安装完成。

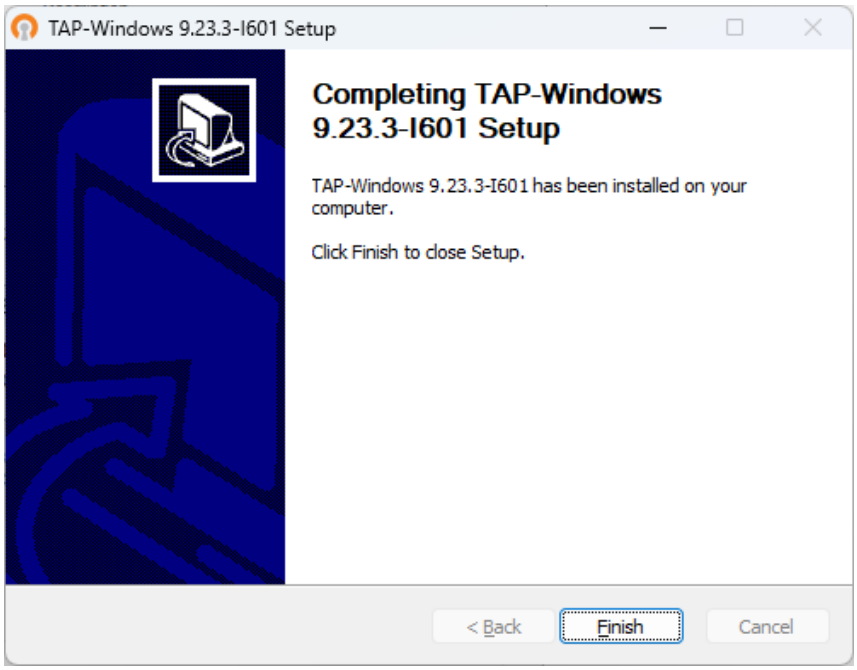


图 1.2-25

点击Finish，至此，安装完成。

打开设备管理器，在网络适配中找到TAP-Windows Adapter V9，并双击。

图 1.2-26

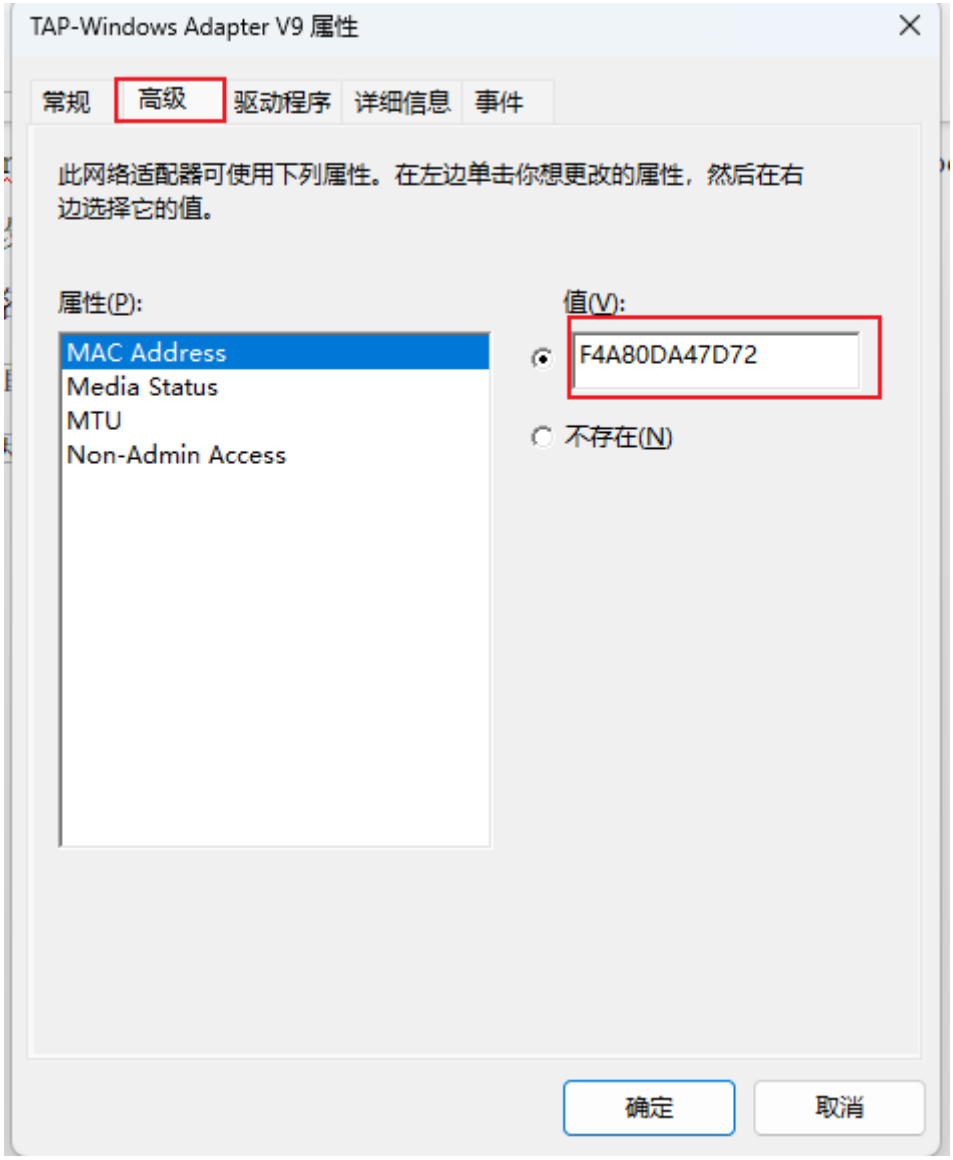
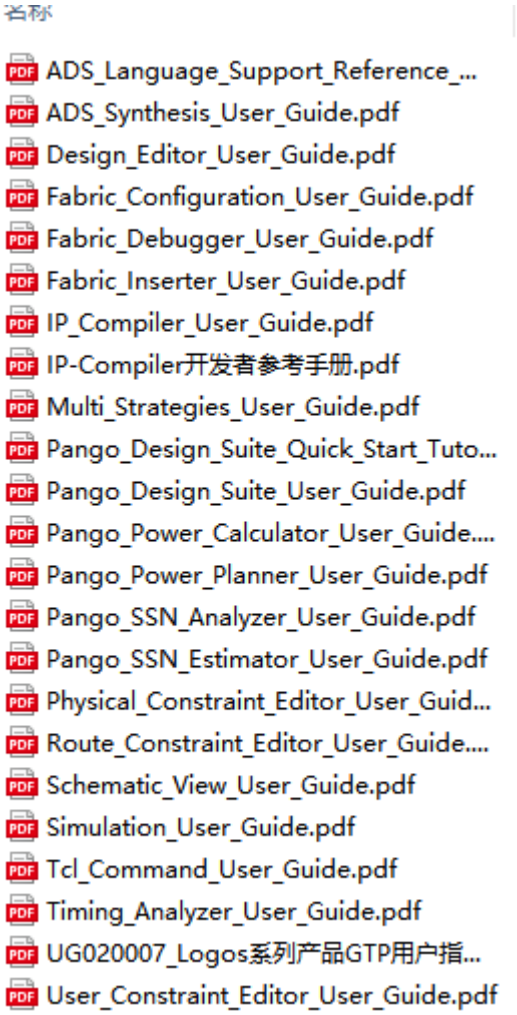


图 1.2-27

选择属性选项卡，并将MAC Address的值改成Lincese文件名后面的字符串。

1.2.2. PDS 工具的使用

详见开发板配套资料《PDS 快速使用手册》文档,或软件安装目录 doc 文件夹下《Pango_Design_Suite_Quick_Start_Tutorial.pdf》和《Pango_Design_Suite_User_Guide.pdf》。



2. Modelsim 的使用和 do 文件编写

2.1. 实验简介

实验目的:

了解 Modelsim 的基本使用方法，完成 do 文件的编写，提高仿真效率。

实验环境:

Window11

PDS2022-SP6.4

Modelsim10.6c

2.2. 实验原理

将 Modelsim 的命令编写到一个 do 文件中，这样每次仿真时，只需运行这个 do 文件脚本即可自动执行其中的所有命令，从而显著提高重复仿真的效率。

2.3. 接口列表

暂无

2.4. Testbench 文件的编写

Testbench 文件其实就是模拟信号的生成，给我们所设计的模块提供输入，以便测试。因为我们上板去生成比特流，尤其是比较复杂的算法，往往是需要耗很多时间的。所以要快速验证我们的设计逻辑是否正常，还得是用仿真来验证，不管是模拟图像的生成还是信号的生成，都可以通过 Testbench 来完成，但是，要注意一点，逻辑前仿真通过了只能说明 80%上板没问题,剩下的可能就要看实际的时序了,毕竟仿真是理想状态，实际总是不太理想。

接下来介绍 Testbench 的基本编写方法:

`timescale 1ns/1ns 该语句 第一个 1ns 表示时间单位为 1ns，第二个 1ns 表示时间精度为 1ns。注意的是，时间单位不能比时间精度还小。时间单位表示运行一次仿真所用的时间。时间精度表示仿真显示的最小刻度。

#10 表示延时 10 个单位时间，比如`timescale 1ns/1ns，#10 表示延时 10ns。

initial 对信号进行初始化，只会执行一次。

{$random}%x，表示随机取[0,x-1]之间的数字。x 为正整数。如果是$random%x，则是[-(x-1),x-1]的数。

$display 打印信息，会自动换行。

$stop 暂停仿真。

$readmemb 读取文件函数。

$monitor 为监测任务，用于变量的持续监测。只要变量发生了变化，$monitor 就会打印显示出对应的信息。

输入信号一般用 reg 定义，方便后续用 always 块生成想要模拟的值，输出一般直接 wire 引出即可。

例如生成时钟，always#10 sys_clk = ~sysclk; 表示每 10 个单位时间就翻转一次，如果时间单位是 ns，那就是每 10ns 翻转一次，就是生成了 50MHZ 的时钟。周期是 20 ns。

接下来给出一个参考的 testbench，如下所示：

```
1.  `define UD #1
2.  module tb_led_test();
3.
4.  reg            clk      ;
5.  reg            rst_n    ;
6.  wire[7:0]      led      ;
7.  reg [7:0]      data     ;
8.
9.  initial begin
10.     rst_n <= 0;
11.     clk   <= 0;
12.     #20;
13.     rst_n <= 1;
14.     #2000
15.     $display("I am stop"); //
16.     $stop;
17. end
18. always#10 clk = ~clk;//20ns   50MHZ
19.
20. led_test
21. #(
22.     .CNT_MAX     (10)
23. )u_led_test(
24.  .clk          (clk     ),// input
25.  .rstn         (rst_n   ),// input
26.  .led          (led     ) // output [7:0]
27. );
28.
29. initial begin
30.     $monitor("led:%b", led);
31. end
32.
33. always@(posedge clk or negedge rst_n)     begin
34.     if(!rst_n)
35.         data     <=     8'd0;
36.     else
37.     begin
38.         data     <=     {$random}%256;
39.         $display("Now data is %d",data);
40.     end
41. end
42.
43. endmodule
```

本 testbench 模块 tb_led_test 用于对 LED 控制模块 led_test 进行仿真验证。

其主要功能包括：产生 50MHz 的时钟信号，提供上电复位信号（20ns 后释放），实例化待测模块并传递参数 CNT_MAX=10，同时在每个时钟周期生成一个随机数 data 并打印输出，以辅助观察模块运行情况。此外，testbench 会实时监控 LED 输出信号的变化，并在 2000ns 时$display 打印提示信息后自动结束仿真。注意$stop 仅仅是暂停仿真，不是完全结束仿真，还可以通过 run 指令继续运行仿真。

在信号设置方面，clk 为 1 位寄存器类型信号，用于产生 50MHz 的时钟；rst_n 为 1 位寄存器类型信号，用于提供低电平有效的复位控制；led 为 8 位线网类型信号，由待测模块输出 LED 状态；data 为 8 位寄存器类型信号，用于在仿真中存储随机数，作为调试参考。

仿真流程如下：首先在初始化阶段，rst_n 被拉低保持复位，同时时钟信号 clk 置为 0。经过 20ns 后释放复位，模块进入正常工作状态。随后，testbench 通过 always 语句产生周期为 20ns 的时钟信号，即 50MHz。此时 led_test 模块被实例化并开始运行，参数 CNT_MAX 被设置为 10。在运行过程中，testbench 在每个时钟上升沿产生新的随机数 data，当复位有效时 data 清零，否则赋值为 $random 取模 256 的结果，并打印输出。同时，使用 $monitor 实时监控 led 信号的变化。当仿真运行至 2000ns 时，打印“I am stop”提示并调用 $stop 命令终止仿真。

在调试过程中，需要重点关注以下几个方面。首先检查复位逻辑，确认 led_test 在 rst_n 拉低时能够正确复位。其次观察 LED 输出，确认其随内部逻辑正常变化。再次检查随机数 data 的变化情况，确保 testbench 在每个时钟周期都能产生不同的值，验证输入驱动的多样性。最后需要关注仿真时间是否满足需求，本例默认运行 2000ns，可根据测试范围进行调整。

2.5. Modelsim 的使用

该部分主要介绍 Modelsim 的基本使用方法。

当我们的设计文件没有使用到任何平台的 IP 核时，我们可以直接打开 Modelsim 新建工程，然后进行仿真，具体步骤如下所示：

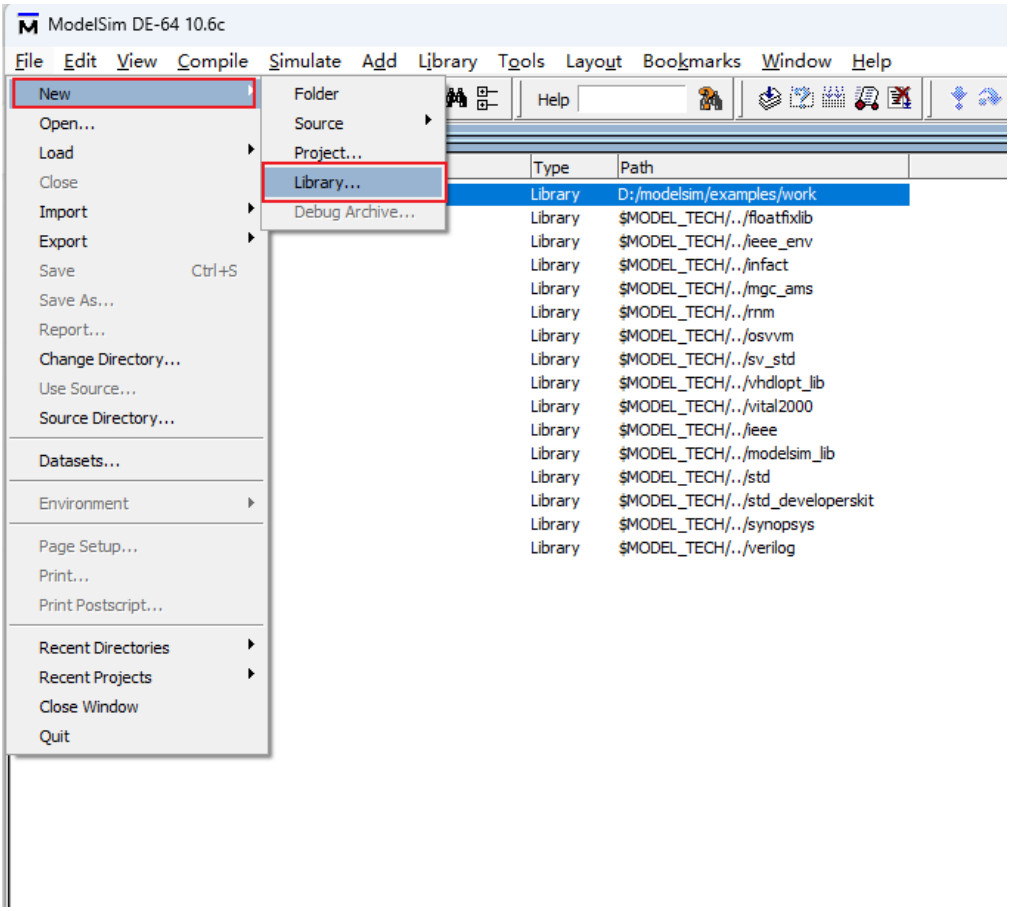


图 2.5-1

点击左上角 File->New->Library，新建一个工作库，一般取名为 work，因为 Modelsim 运行时都会在这个 work 下面工作，所以第一次运行 Modelsim 我们需要新建一个叫 work 的库，如果打开发现已经有 work 的工作库时，则不用新建。

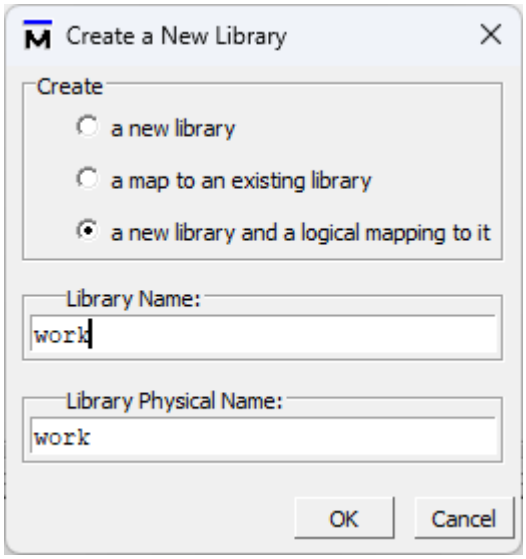


图 2.5-2

输入 work，点击 OK 即可。新建完成后就可以看到有个 work 的库在 Modelsim 里面。接下来新建工程。

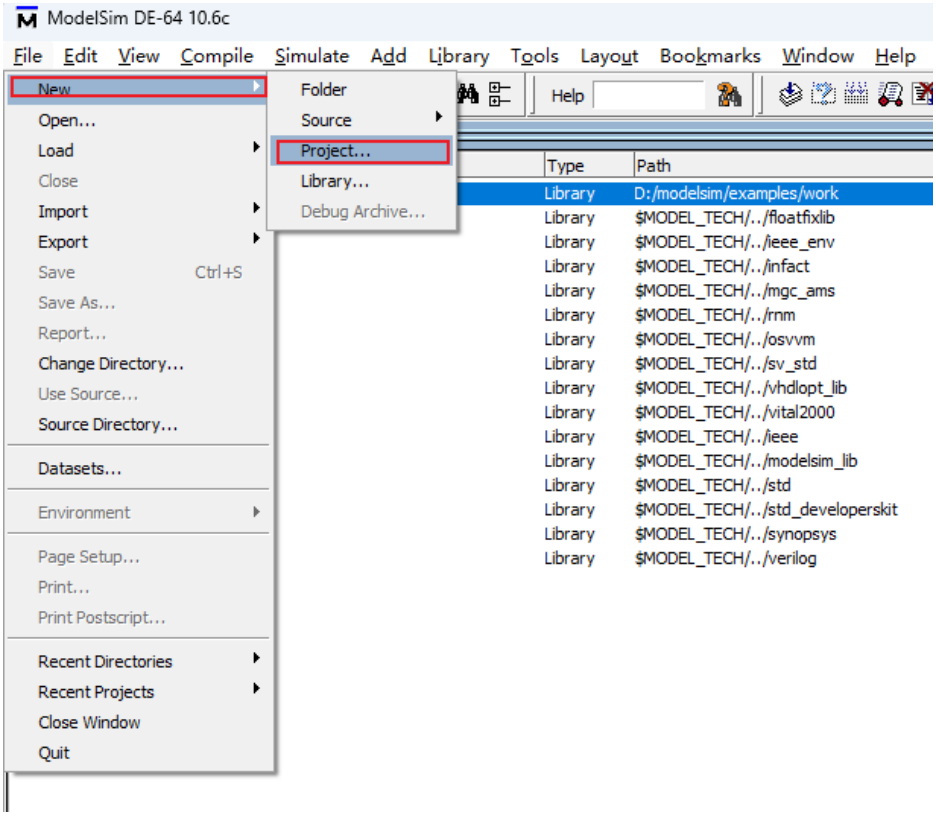


图 2.5-3

左上角 File->New->Project，新建工程。

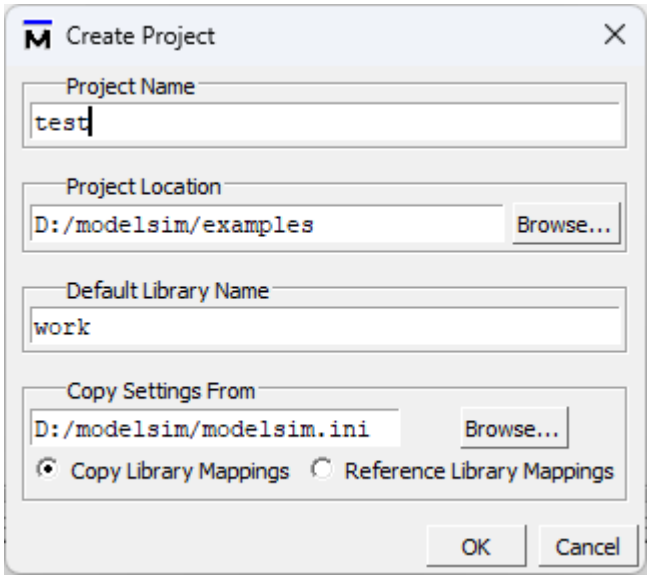


图 2.5-4

工程名注意不要出现中文，其余保持默认即可，可以看到 Default Library Name 其名字默认指向 work。

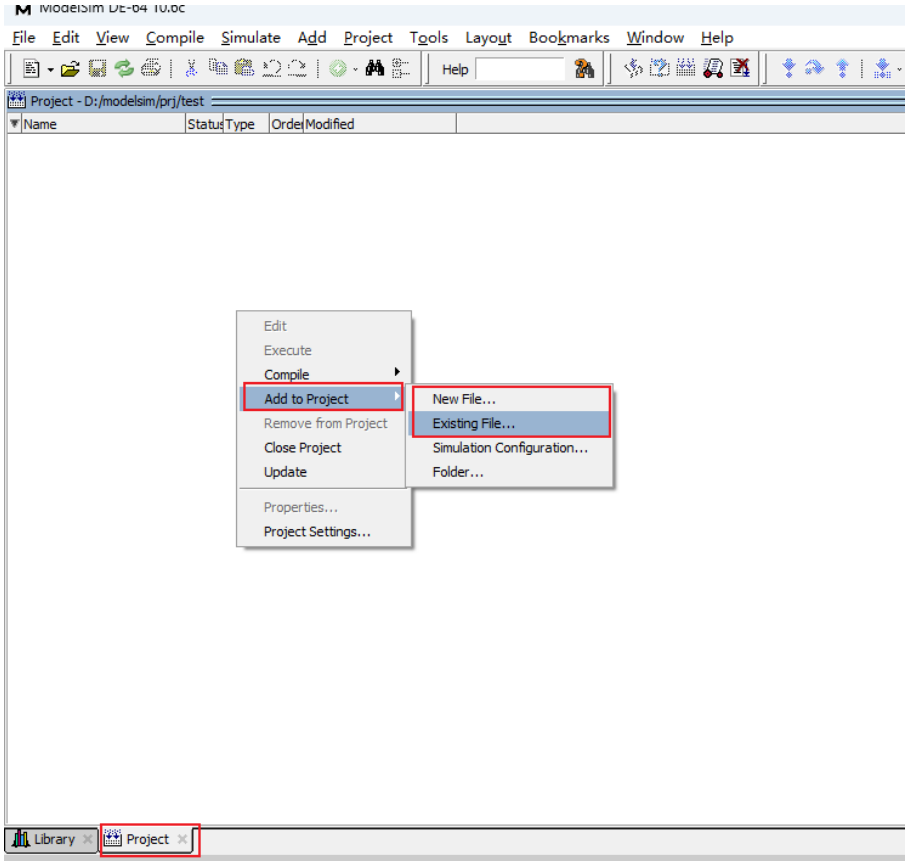


图 2.5-5

新建完后可以看到下方多了一个叫 Project 的选项卡，鼠标右键该界面空白部分，选择 Add to Project->Existing File，或者 Add to Project->New File。添加我们要仿真的文件，这里用一个比较简单的来演示。

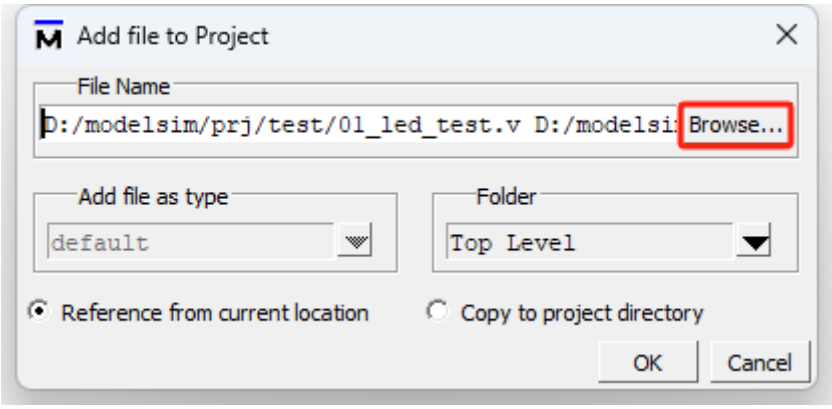


图 2.5-6

点击 Browse，添加要仿真的文件。

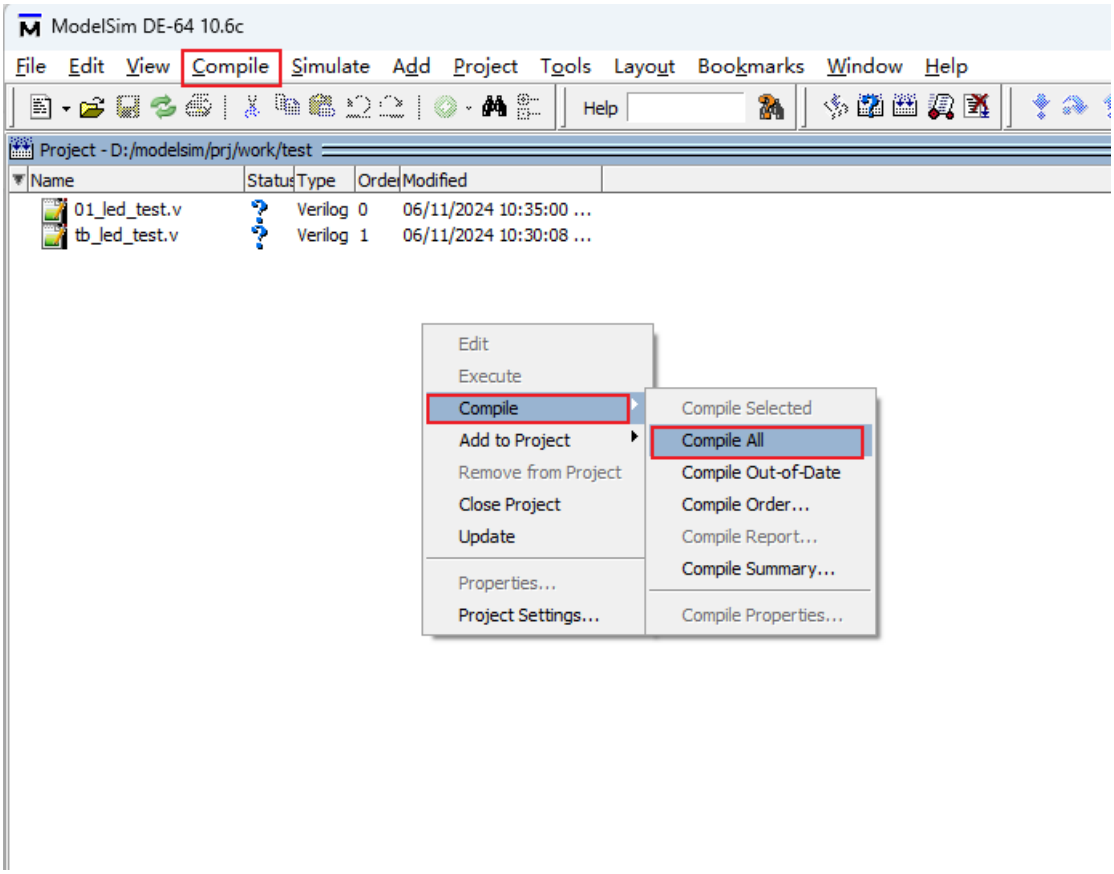


图 2.5-7

选择上方 Compile 或者鼠标右键空白部分，选择 Compile->Compile，该步骤主要对 verilog 文件进行编译，检查是否有语法错误等。

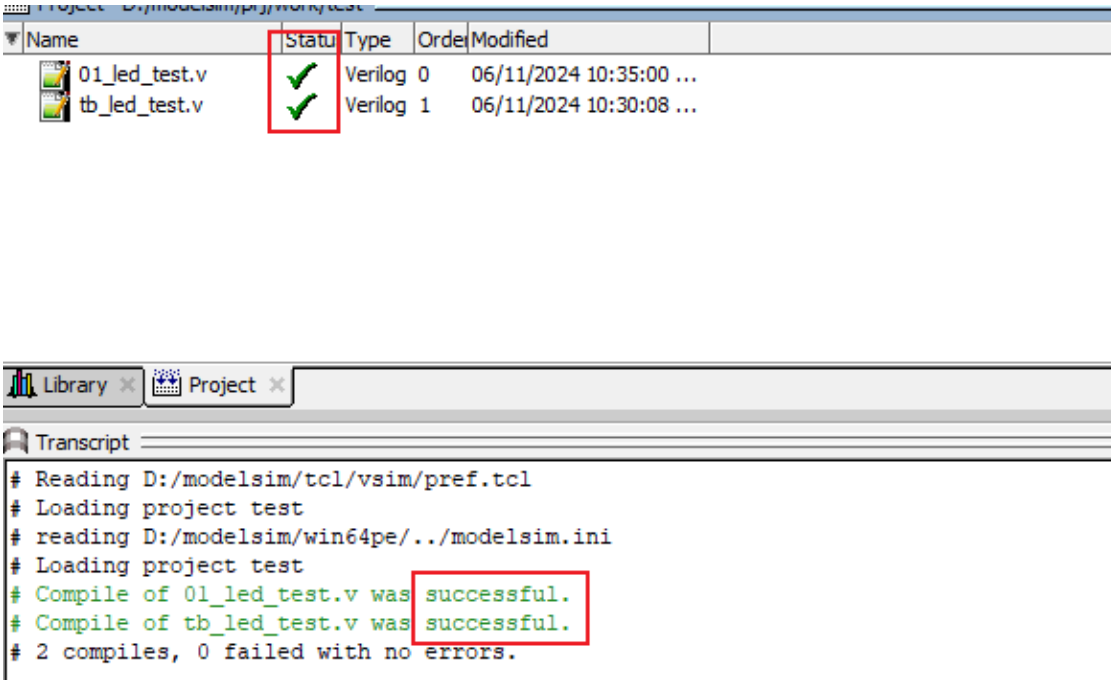


图 2.5-8

当看到 Status 是个绿色的√，或者下方打印输出区间没有任何 errors，显示 successful，表示我们的文件编译通过，可以进行下一步操作了。

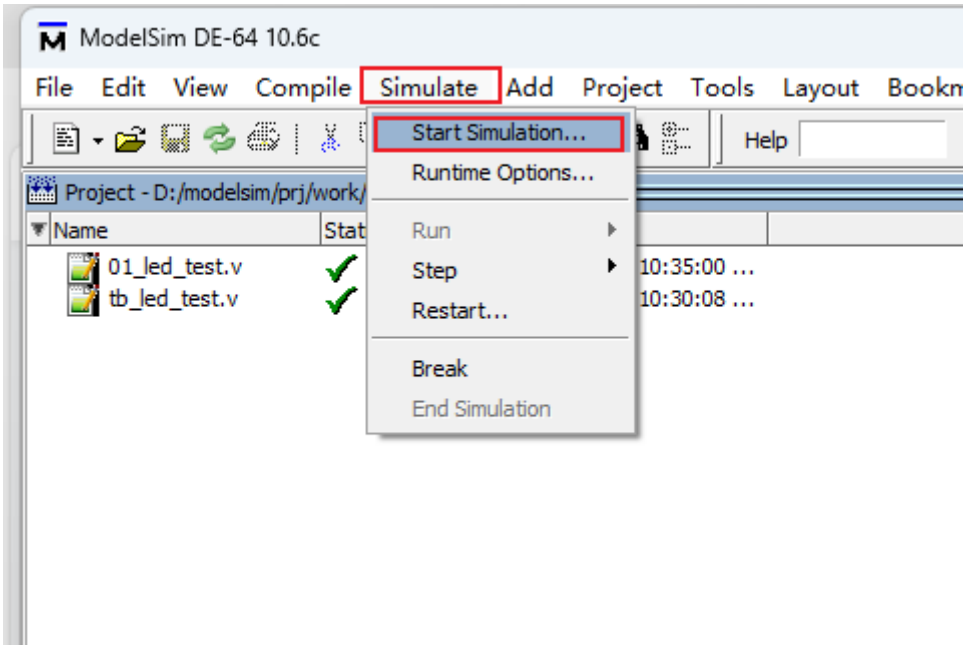


图 2.5-9

选择上方 Simulate->Start Simulation，之后会弹出如图 2.5-10 所示的界面，把 work 展开，选择我们的 testbench 文件，可以看到 Design Unit 显示的是我们的 testbench 文件就没问题了。然后点击 OK，开始仿真。

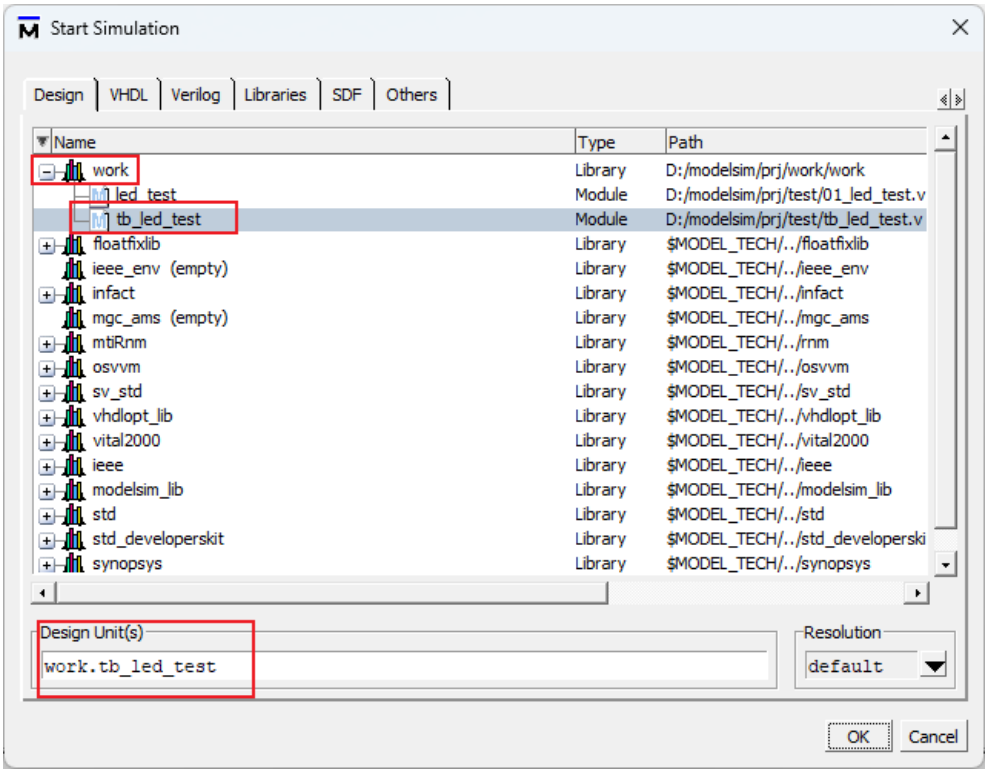


图 2.5-10

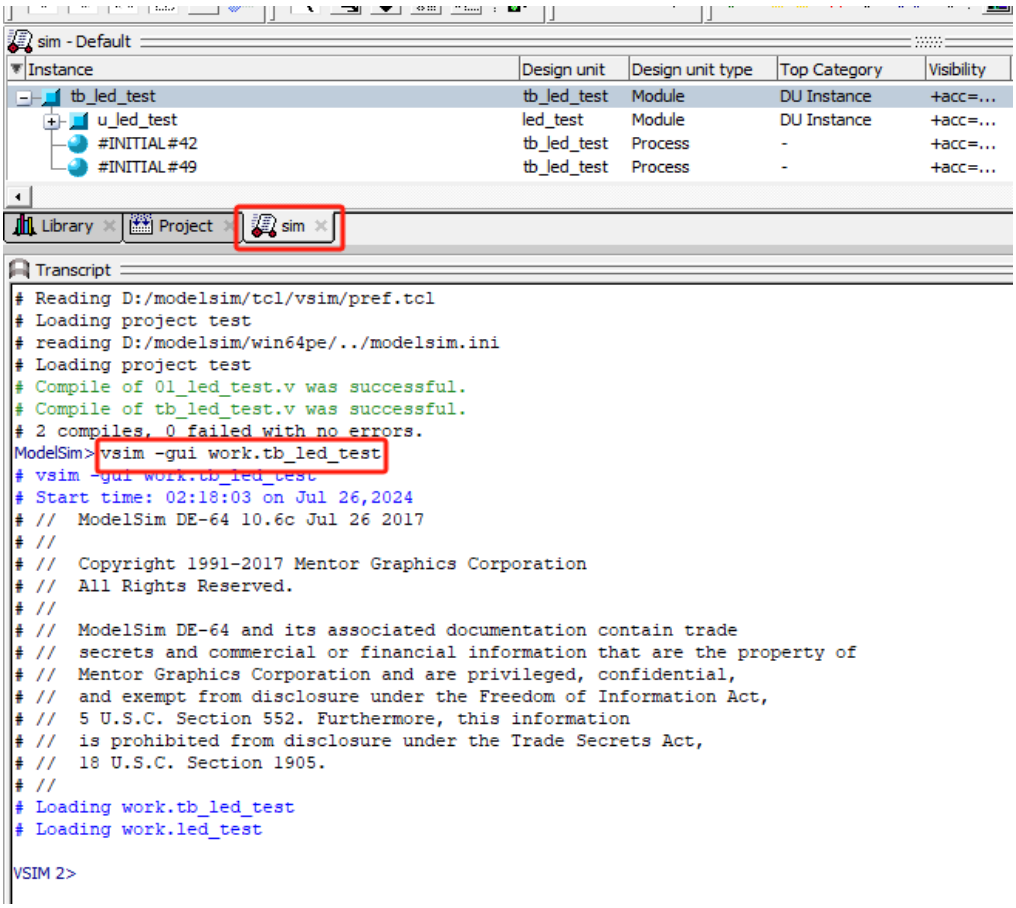


图 2.5-11

可以看到会弹出一个新的选项卡叫”sim”，然后看红框部分，当点击 OK 后，实际

上 Modelsim 自动输入一句命令 vsim -gui work.tb_led_test。这其实和后面我们的 do 文件编写是有联系的，do 文件的编写实际上就是在写这些命令，这里我们先铺垫一下。

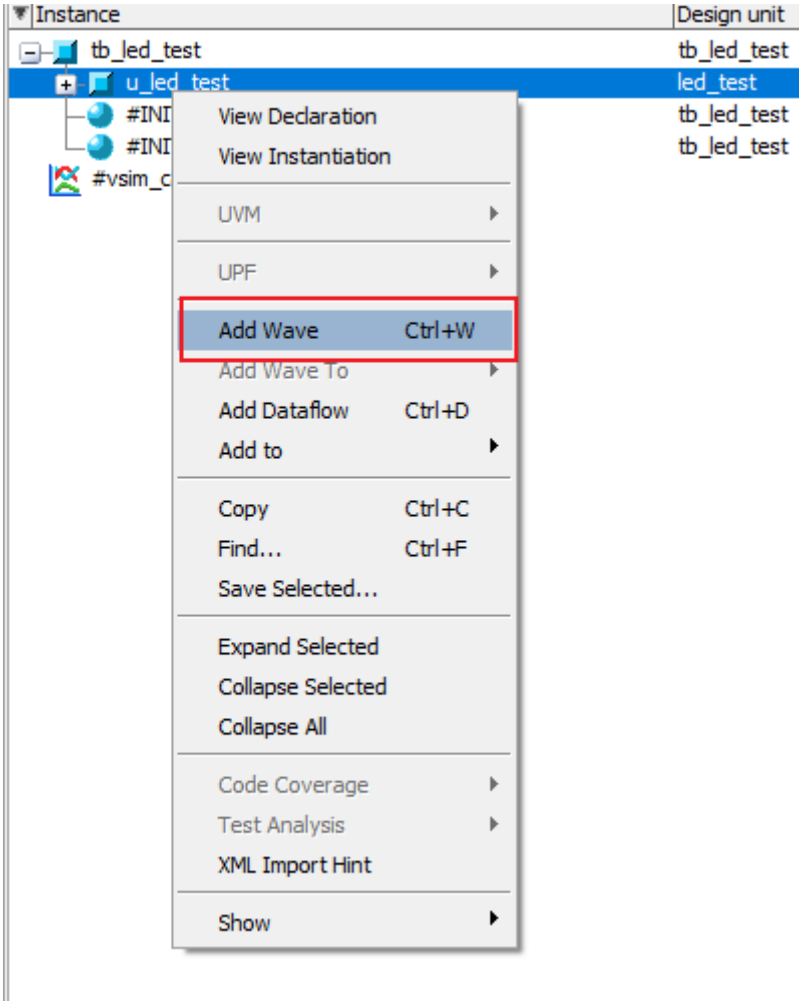


图 2.5-12

接下来添加我们要观察的信号，这里我是直接右键 u_led_test 这个模块，然后选择 Add Wave 或者 crtl+w，即可将该模块的全部信号都加入到波形窗口中。

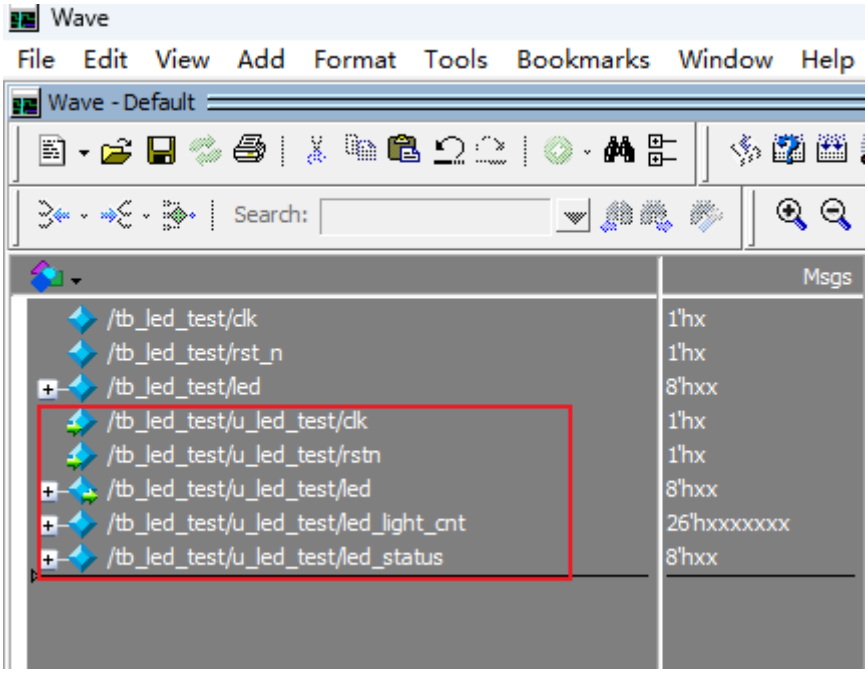


图 2.5-13

可以看到，波形窗口已经添加该模块的全部信号，之后我们按下快捷建，crtl+a 全选全部信号，crtl+g，对信号进行分组，该分组是按照不同模块进行分组，crtl+h 消除信号名称的前缀，如图 2.5-14 所示：

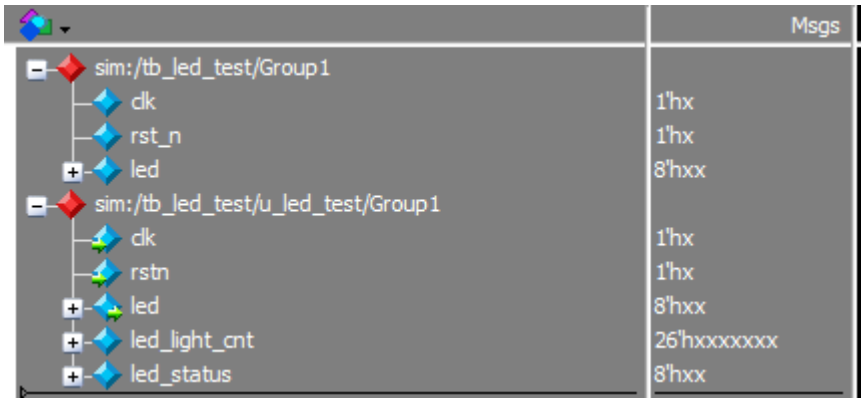


图 2.5-14

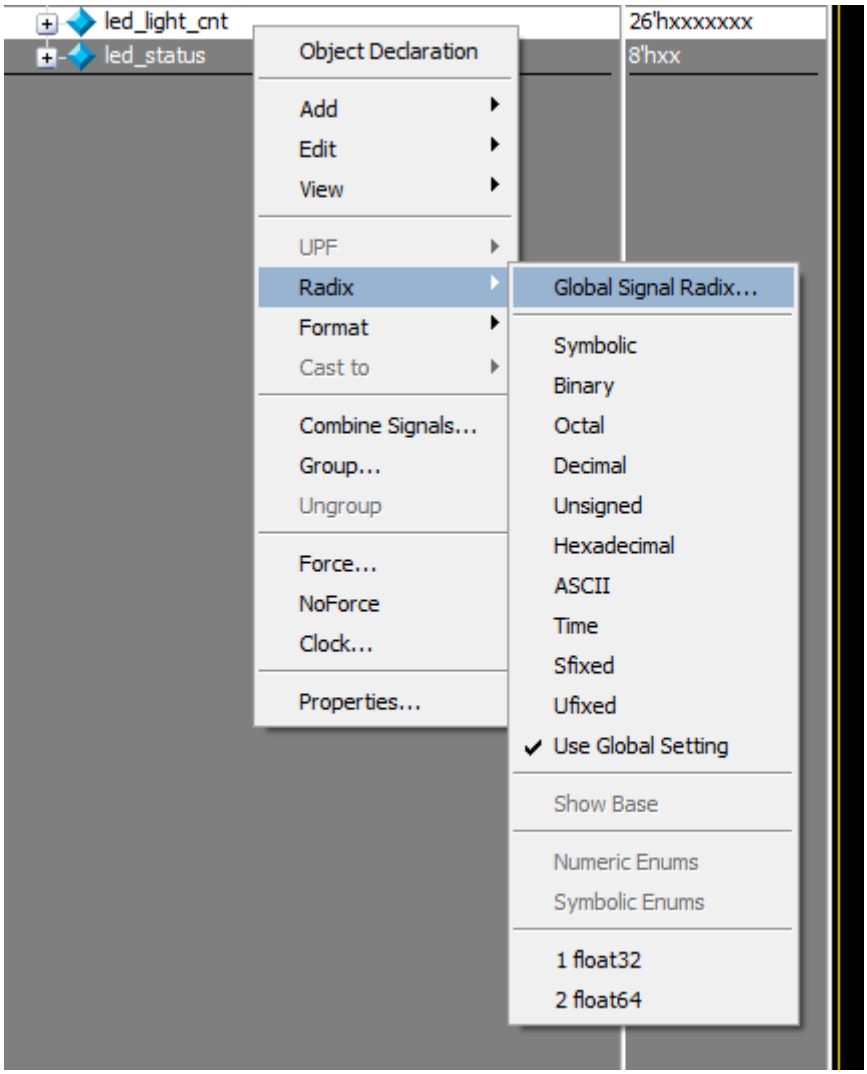


图 2.5-15

鼠标右键信号，Radix 可以修改该信号显示的格式，比如二进制显示，16 进制显示等。Properties 可以修改该信号波形显示的颜色，这两个是比较常用的。接下来开始来运行我们的仿真。

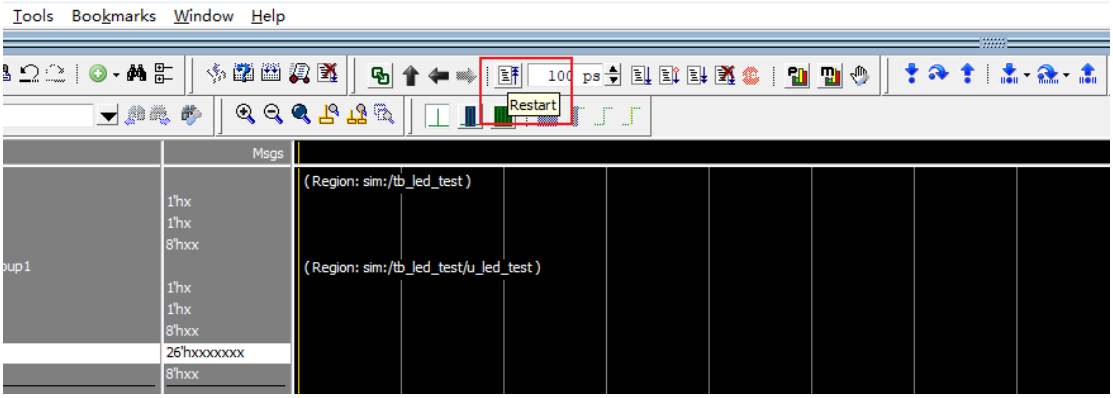


图 2.5-16

点击上方这个地方，对信号全部进行 Restart 复位。

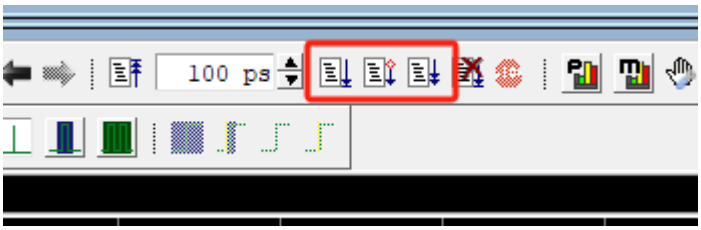


图 2.5-17

红框旁边的 100 ps 是一次仿真运行的时间，红框内从左往右看，第一个是表示运行一次仿真，其时间为 100ps，100ps 并不是固定的，我们可以修改为 1ms,100us 等。第二个基本比较少用。第三个是让仿真不断运行，直到用户点击停止为止，如图 2.5-18 所示：

图 2.5-18

按下后，当用户点击 stop，仿真才会停止。

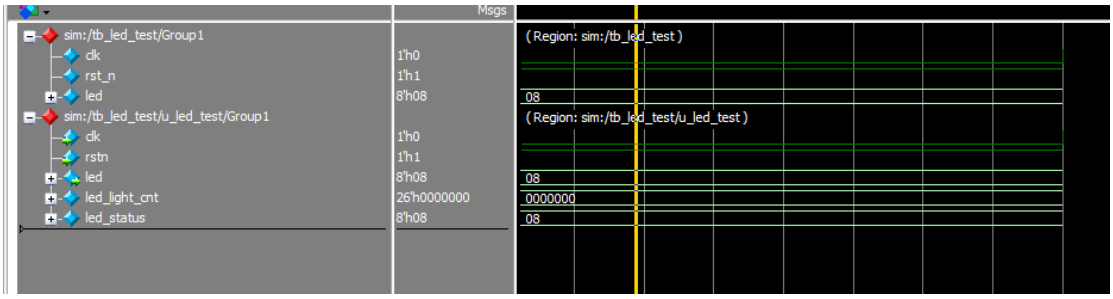


图 2.5-19

图 2.5-19 为操作后显示出来的波形。

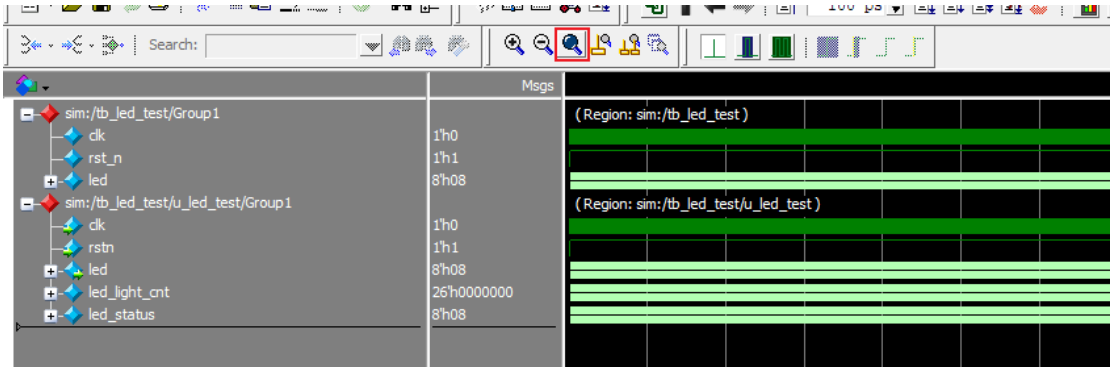


图 2.5-20

点击红框部分那个按钮，将缩小波形。我们也可以按住 crtl 键，然后鼠标滚轮上下，可以对波形进行缩放。到这里，基本的使用方法就结束了，更多操作大家可以去看视频操作，或者网上百度，或者自己摸索一下。

再铺垫一下，完成这些操作后，我们回去打印输出区间观察一下

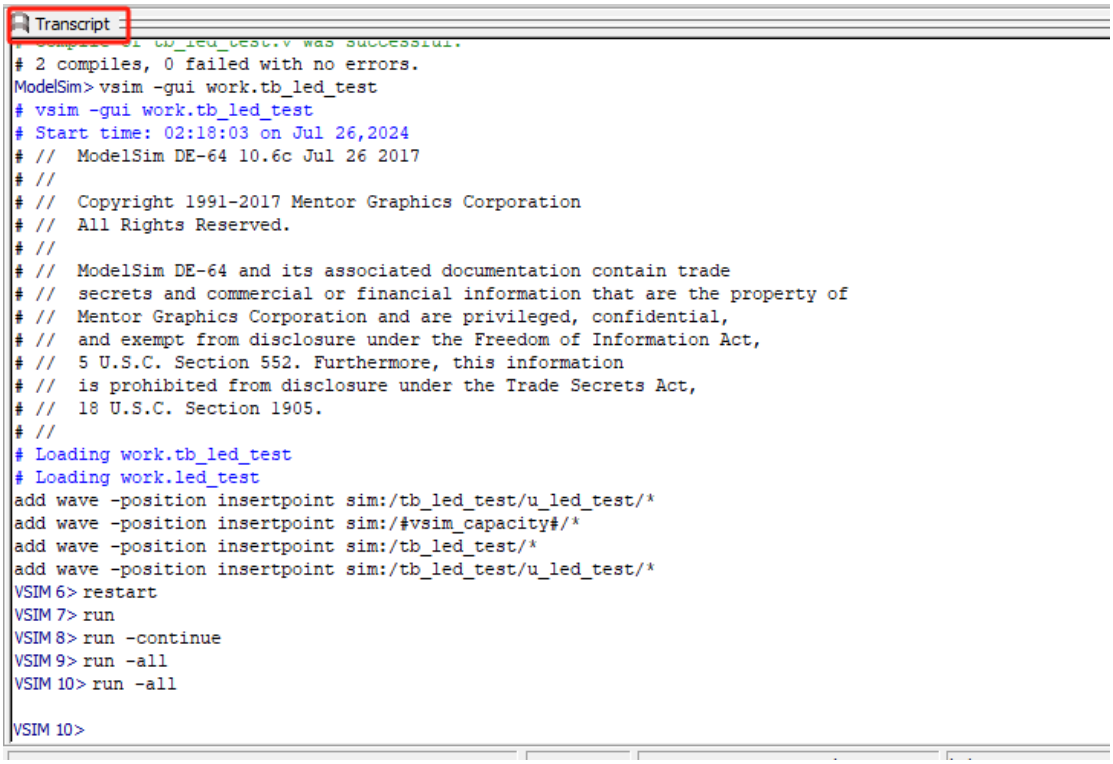


图 2.5-21

可以看当我们添加波形时，Modelsim 自动执行了一句 add wave -position xxxxxx x 的命令，执行了 restart，也就是复位，run 就是运行仿真，这些都和后续 do 文件的编写息息相关。所以其本质就是编写这些命令，我们就不需要用鼠标去点每个功能，每次我们只需要运行 do 文件就可以完成全部操作，大大提高我们的效率。

如果大家不小心把某些选项卡关了，可以在上方 View 选择要查看的窗口，如图 2.5-22 所示：

比如 Library 前面有个√，就是显示 Library 选项卡的意思。大家可以在这里找找需要显示的界面。

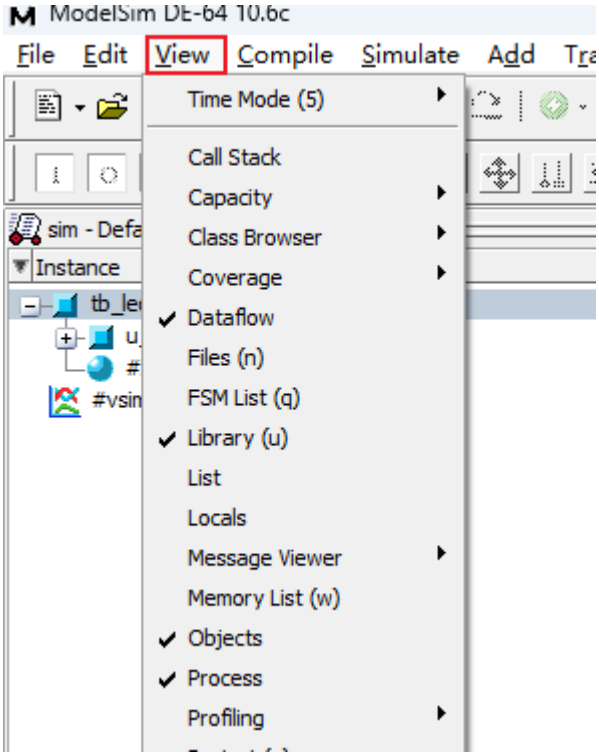


图 2.5-22

2.6. 文件的编写

2.6.1. 基本命令介绍

前文其实也有提到，Modelsim 实际上是通过输入命令来执行相应功能的，比如 add wave xxxxx，就是把信号添加到波形窗口。那接下来就是教大家如何使用这些命令来提高我们的开发效率。

首先先介绍常用的命令。

vlib:该命令为创建一个目录。例如 vlib work。即在当前路径下创建一个名字叫 work 的文件夹。

vmap: 映射逻辑库到物理目录。其格式为 vmap work work 第一个 work 逻辑库名称，第二个 work 是表示在 PC 里实际的库文件的路径。

注意：前面所说的通过 File->New->Library 的方法建立了一个 work 的库，其实就是运行了 vlib 和 vmap，具体可以看教程视频讲解。本质就是 vlib work vmap work work。

vlog:该命令用来编译 verilog 源码。例如 vlog -work work ./src/test.v 第一个 work 表示文件夹的名称 、第二个 work 表示 modelsim 中 library 的库的名称、第三个就是要编译的文件的路径。

vsim:表示启动仿真。

add wave:表示添加波形到波形窗口(add wave -divider 会添加分割线)。

view wave:打开波形窗口。

view structure:打开结构窗口。

view signals:打开信号窗口。

restart:重新仿真，复位仿真时间，并清空之前的仿真数据。(如果修改了 verilog 文件 需要重新编译再仿真才行，restart 只是在当前这个仿真下重新开始仿真而已)。

run x:运行 x 时间。例如 run 1ms run1ns run 1us run 250ms 均可。

quit -sim:退出仿真。

quit:退出 Modelsim。（关闭整个软件）

2.6.2. 文件示例

如果从 0 开始写，相信是比较陌生的，其实当我们使用紫光联合仿真的时候，他会在 sim 的文件夹下生成一个后缀为 tcl 的脚本，每次运行联合仿真，实际就是打开 Modelsim 然后运行该 tcl 脚本，具体路径都在工程目录下的 sim 文件夹下，如图 2-1 所示：

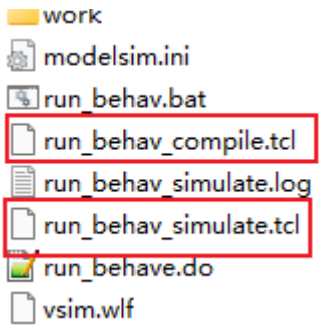


图 2-1

主要是运行 run_behav_compile.tcl 和 run_behav_simulate.tcl 这两个文件。我们可以打开来参考一下。

```
vlib work
vmap work ./work
vmap usim "D:/modelsim/pg_sim_lib/usim"
vmap adc_e2 "D:/modelsim/pg_sim_lib/adc_e2"
vmap ddc_e2 "D:/modelsim/pg_sim_lib/ddc_e2"
vmap dll_e2 "D:/modelsim/pg_sim_lib/dll_e2"
vmap hsstlp_lane "D:/modelsim/pg_sim_lib/hsstlp_lane"
vmap hsstlp_pll "D:/modelsim/pg_sim_lib/hsstlp_pll"
vmap iolhr_dft "D:/modelsim/pg_sim_lib/iolhr_dft"
vmap ipal_e1 "D:/modelsim/pg_sim_lib/ipal_e1"
vmap ipal_e2 "D:/modelsim/pg_sim_lib/ipal_e2"
vmap iserdes_e2 "D:/modelsim/pg_sim_lib/iserdes_e2"
vmap oserdes_e2 "D:/modelsim/pg_sim_lib/oserdes_e2"
vmap pciegen2 "D:/modelsim/pg_sim_lib/pciegen2"
vlog -work work \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/01_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desktop/tb_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/source/Desktop/01_led_test.v"
```

以上是 run_behav_compile.tcl 的内容，大家可以结合视频教程一起分析一下，该文

件主要完成工作区间的建立和一些库的映射以及对代码的编译。

vlib:该命令创建一个文件夹。例如 vlib work。

vmap:映射逻辑库到物理目录。其格式为 vmap work work 第一个 work 逻辑库名称，第二个 work 是表示在 PC 里实际的库文件的路径。

vlog:该命令用来编译 verilog 源码。例如 vlog -work work ./src/test.v

第一个 work 表示文件夹的名称。 第二个 work 表示 Modelsim 中 library 的库的名称。 第三个就是要编译的文件的路径。

所以大家其实可以参考 demo 的脚本来编写我们的 do 文件，我们的 do 文件本质上也是写这些命令，只不过后缀不一样，但其运行方法是一致的，均为 do+空格+文件名。所以到此，大家应该都知道我们的 do 文件是怎么去编写了，其实就是把这些 Modelisim 的运行指令，写成一个脚本，然后用 do 指令直接完成我们想要的所有操作，可以大大提高我们的效率。在展示如何使用 Modelsim 的时候也介绍了，每一步操作实际上都是软件工具自动帮我们输入命令，现在就是把这些命令给拿出来。接下来我们看 run_behav_simulate.tcl 的内容。

```
1.  vsim  -novopt  -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft
    -L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
2.  add wave *
3.  view wave
4.  view structure
5.  view signals
6.  run 1000ns
7.  vsim  -novopt  -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft
    -L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
8.  add wave *
9.  view wave
10. view structure
11. view signals
12. run 1000ns
13.
```

vsim:表示启动仿真。vsim -L +逻辑库的名字。

add wave:表示添加波形到波形窗口。(add wave -divider 会添加分割线)

view wave:打开波形窗口。

view structure:打开结构窗口。

view signals:打开信号窗口。

run x:运行 x 时间。例如 run 1ms run1ns run 1us run 1s run 250ms 均可。

再顺带介绍一下一些常用的。

restart:重新仿真，复位仿真时间，并清空之前的仿真数据。(如果修改了 verilog 文件 需要重新运行 do 文件才生效，restart 只是在当前这个仿真下重新开始仿真而已)

quit -sim:退出仿真。

quit:退出 Modelsim。

该脚本主要是完成仿真，以及一些仿真完成后的操作，比如添加波形，观察波形，

设置运行时间。

所以，其实我们可以把这两个文件合起来，变成一个文件，做成我们自己的 do 文件就行了，如此，以后修改代码重新仿真都不需要去 PDS 软件里面去点联合仿真，我们直接在 Modelsim 里面直接 do 就行了。合并后的 do 文件如下所示：

```
cd D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/sim/behav
vlib  work
vmap  work ./work
vmap  usim "D:/modelsim/pg_sim_lib/usim"
vmap  adc_e2 "D:/modelsim/pg_sim_lib/adc_e2"
vmap  ddc_e2 "D:/modelsim/pg_sim_lib/ddc_e2"
vmap  dll_e2 "D:/modelsim/pg_sim_lib/dll_e2"
vmap  hsstlp_lane "D:/modelsim/pg_sim_lib/hsstlp_lane"
vmap  hsstlp_pll "D:/modelsim/pg_sim_lib/hsstlp_pll"
vmap  iolhr_dft "D:/modelsim/pg_sim_lib/iolhr_dft"
vmap  ipal_e1 "D:/modelsim/pg_sim_lib/ipal_e1"
vmap  ipal_e2 "D:/modelsim/pg_sim_lib/ipal_e2"
vmap  iserdes_e2 "D:/modelsim/pg_sim_lib/iserdes_e2"
vmap  oserdes_e2 "D:/modelsim/pg_sim_lib/oserdes_e2"
vmap  pciegen2 "D:/modelsim/pg_sim_lib/pciegen2"
vlog -work work    \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desk-
top/01_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/Desk-
top/tb_led_test.v" \
"D:/ziguan_demo/MES2L100Hv2/MES2L100Hv2/2_Demo/01_led_test/01_led_test/source/source/Desk-
top/01_led_test.v"

vsim  -novopt  -L work -L usim -L adc_e2 -L ddc_e2 -L dll_e2 -L hsstlp_lane -L hsstlp_pll -L iolhr_dft
-L ipal_e1 -L ipal_e2 -L iserdes_e2 -L oserdes_e2 -L pciegen2 tb_led_test usim.GTP_GRS
add wave *
add wave -position insertpoint sim:/tb_led_test/u_led_test/*
view wave
view structure
view signals

restart
run 1000ns

```

可以看到，基本上就是把两个文件给合并起来，然后多添加了 restart 语句。至于添加波形的语句 add wave -position insertpoint sim:/tb_led_test/u_led_test/* ，如果大家不熟悉这样的格式，可以直接在 Modelsim 里面手动添加，然后看其打印区间，输出的指令格式，复制下来就行了。一开始，大家不熟悉的话可以这么操作，等熟悉了后，就可以完全编写了，因为使用了紫光的联合仿真，所以中间会用 vmap 映射很多的紫光的仿真库。包括 vsim 也调用了很多紫光相关的库。所以，如果大家并没有用到紫光的 IP 核或者原语等，只是单纯的验证逻辑的话，其实没有这么麻烦。与紫光的仿真库有关的都可以删了，所以主要就是一个 vlib work，然后 vmap work work，然后 vlog 我们要仿真的文件的路径，注意需要写好 testbench。然后 vsim，然后添加要查看的波形，然后 restart，然后 run 即可。

3. Pango 与 Modelsim 的联合仿真

3.1. 实验简介

实验目的：完成 PDS 软件和 Modelsim 的联合仿真设置，所有版本设置方法基本一致，这里以 PDS2002.2 版本为例。

实验环境：

Window11

PDS2022.2-SP6.4

Modelsim10.6rc

硬件环境：

暂无

3.2. 实验原理

编写完成 Testbench 文件后，在 PDS 设置好 Modelsim 的路径，即可启动联合仿真。

3.2.1. 编译仿真库

首先打开 PDS 软件，可以不用打开工程，具体图 3.2-1 所示：

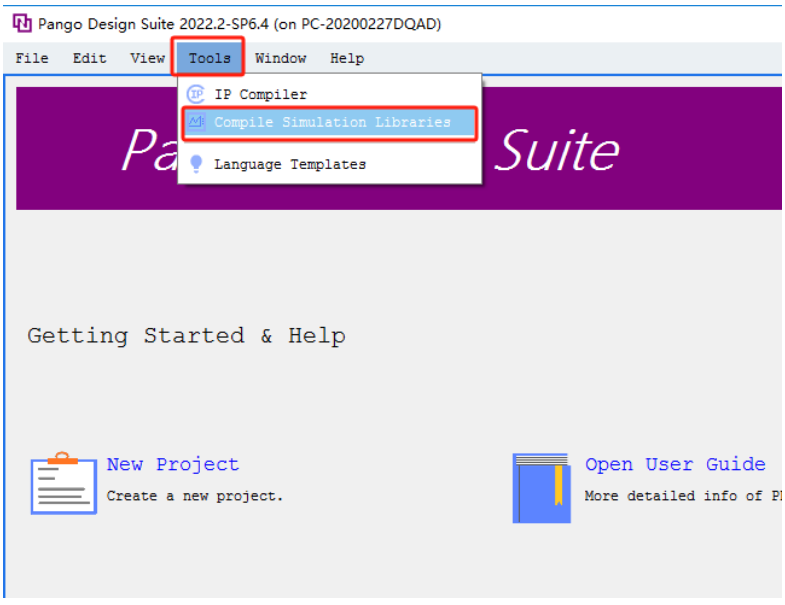


图 3.2-1

不管是否打开工程，均可以在软件上方工具栏中找到 Tools->Compile Simulation Libraries;

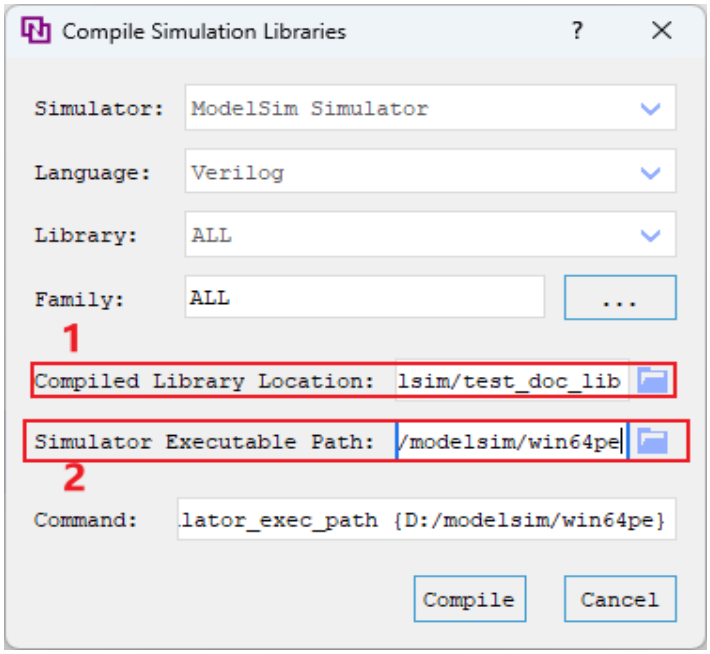


图 3.2-2

打开后可以看到弹出如图 3.2-2 所示的界面，其中红框 1 表示存放生成的仿真库的路径，推荐可以在 Modelsim 的安装目录下新建一个文件夹来存放，笔者是用 pango_sim_lib 来表示，通俗易懂。红框 2 表示 Modelsim 的启动路径，不同版本其存放的文件夹名字可能不一样，有 win64pe/win64/win32 等，都是 win 开头，笔者所用的 10.6c 为 win64pe。之后点击 Compile 即可，等待编译完成。

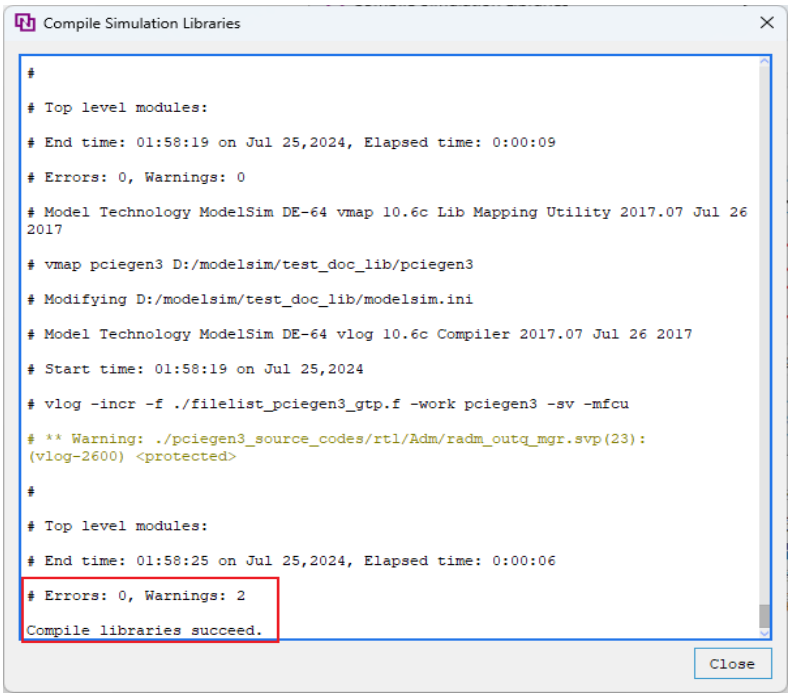


图 3.2-3

当没有任何 Errors 时(Warnings 可以忽略)，表示我们的仿真库已经生成成功了。

3.2.2. 设置仿真路径

编译完成仿真库后，我们需要在 PDS 工程中设置仿真路径，即设置 Modelsim 的路径以及刚才生成的仿真库的路径。

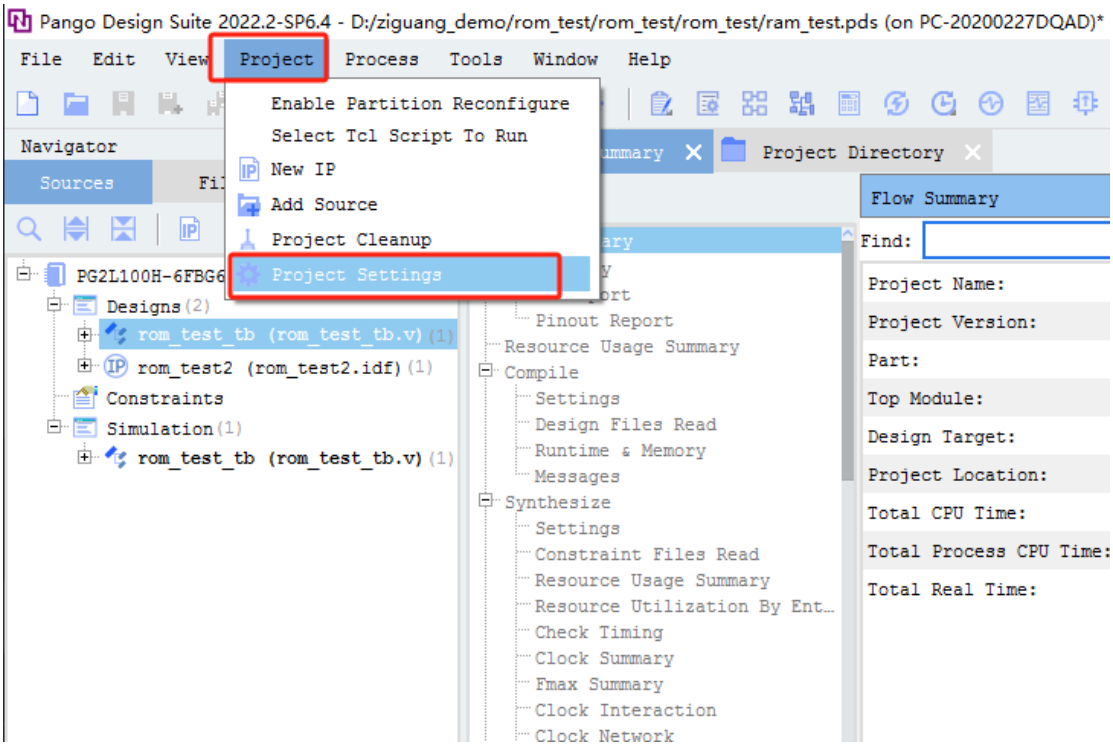


图 3.2-4

打开工程后，选择 Project->Project Settings；

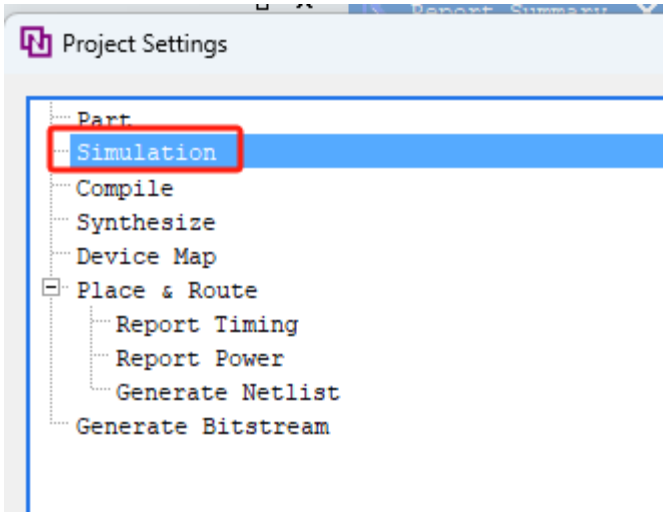


图 3.2-5

选择 Simulation 选项，准备设置我们的仿真路径。

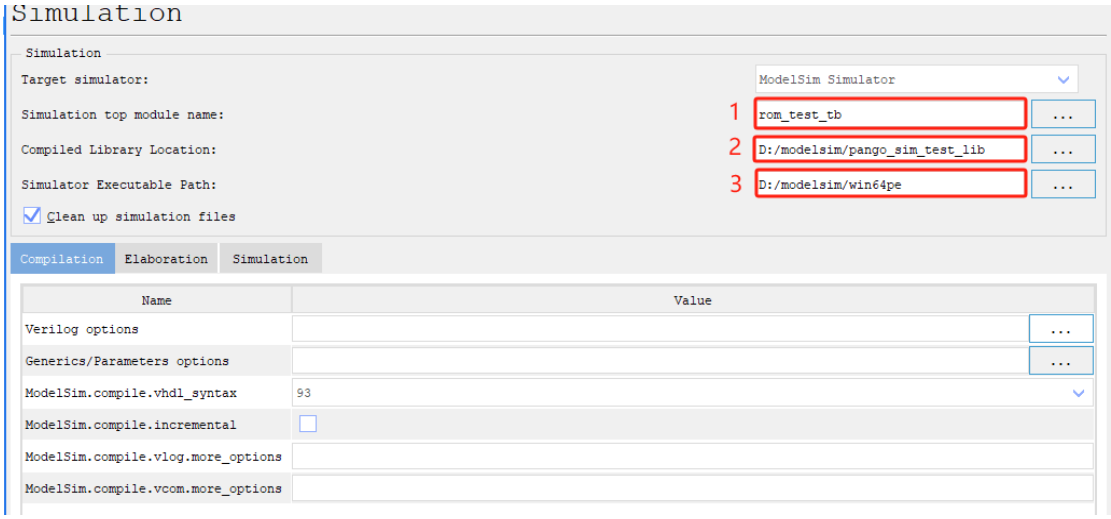


图 3.2-6

接下来开始配置路径，红框 1 表示我们要仿真的顶层文件，PDS 软件会自动识别。红框 2 选择生成的仿真库的路径。红框 3 是 Modelsim 的启动路径，也就是说红框 2 和红框 3 的路径和刚才生成仿真库所设置的路径是一模一样的。之后点击 OK 即可。

3.2.3. 启动联合仿真

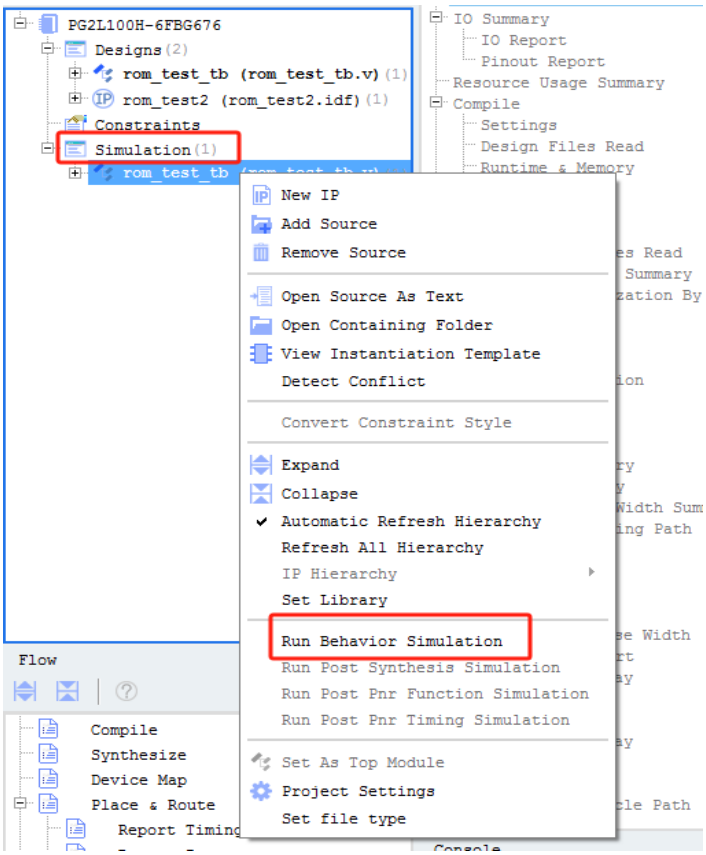


图 3.2-7

接下来在 Simulation 下，右键仿真的顶层文件，可以看到有四种仿真，我们常用的是第一种行为仿真，可以通过查看仿真波形来验证我们设计的逻辑功能是否正确，该仿

真不需要进行任何编译即可直接进行，如果是后面的三种，比如 Post Synthesis Simulation 则需要综合后才能仿真。接下来点击 Run Behavior Simulation，会自动弹出 Modelsim 的界面。如图 3.2-8 所示：

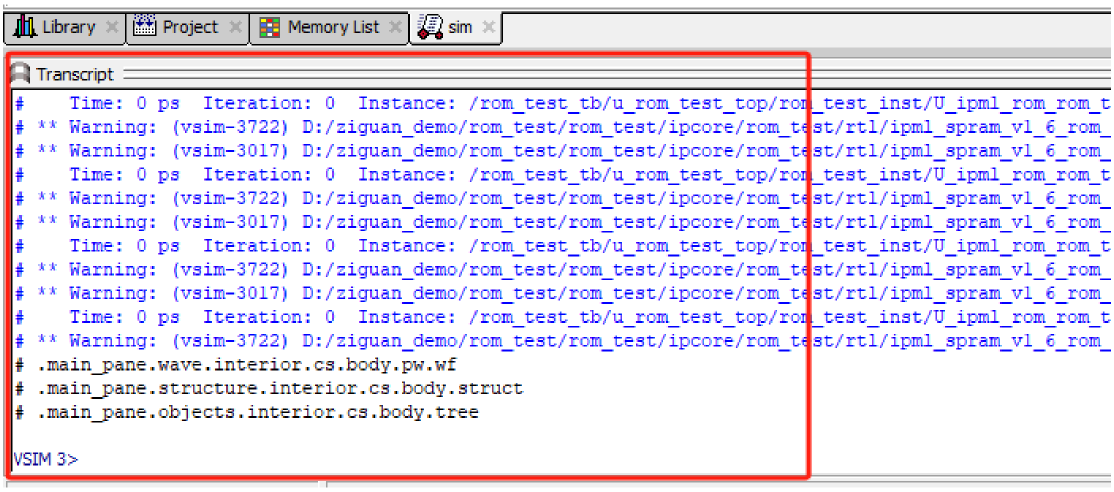


图 3.2-8

打开后 Modelsim 会自动执行仿真脚本，具体在下个章节会介绍，如果观察到打印区间没有显示任何 error，即表示仿真成功，可以开始进行某些操作任何观察波形。

4. 紫光同创 IP core 的使用及添加

4.1. 实验简介

实验目的:

了解 PDS 软件如何安装 IP、使用 IP 以及查看 IP 手册

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

暂无

4.2. 实验原理

4.2.1. IP 的安装

PDS 软件安装完成之后，PDS 自带部分基础 IP，其他 IP 需用户下载 IP 安装包并安装 IP。

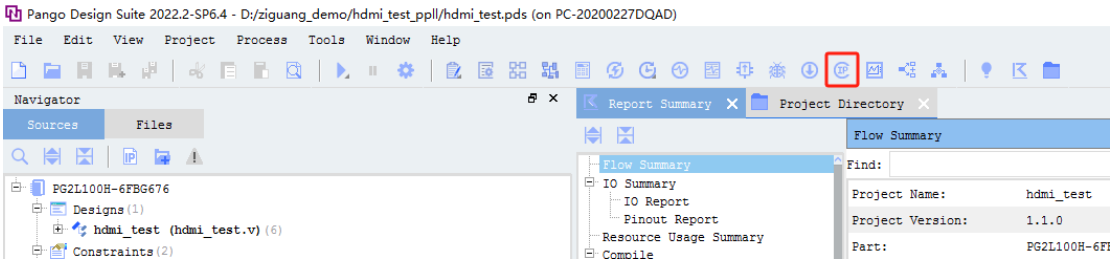


图 4.2-1

打开 PDS 后，点击图 4.2-1 里红框部分的 IP 图标。

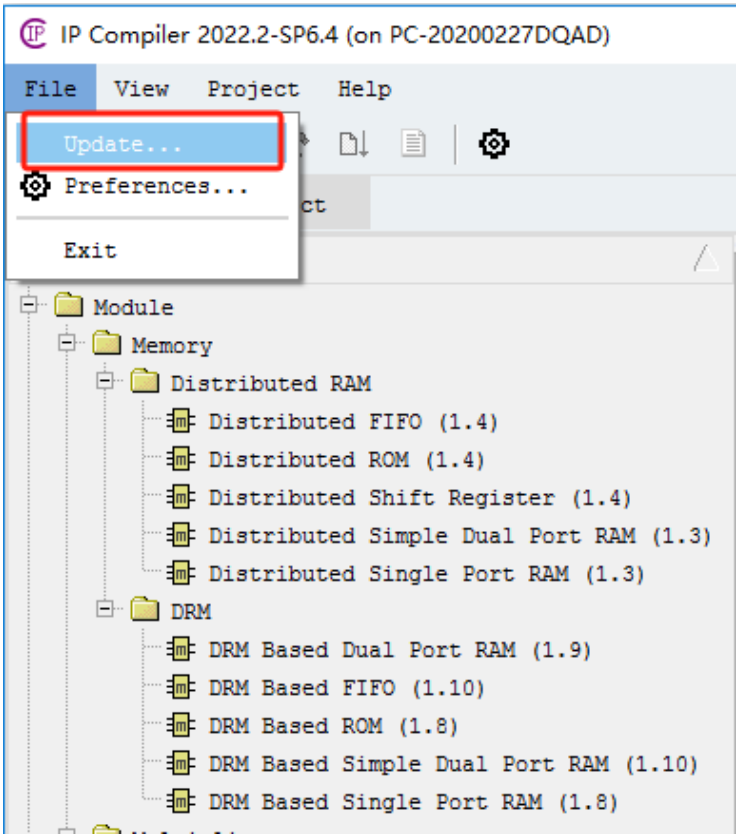


图 4.2-2

之后在弹出的选项卡的左上角点击 File->Update...

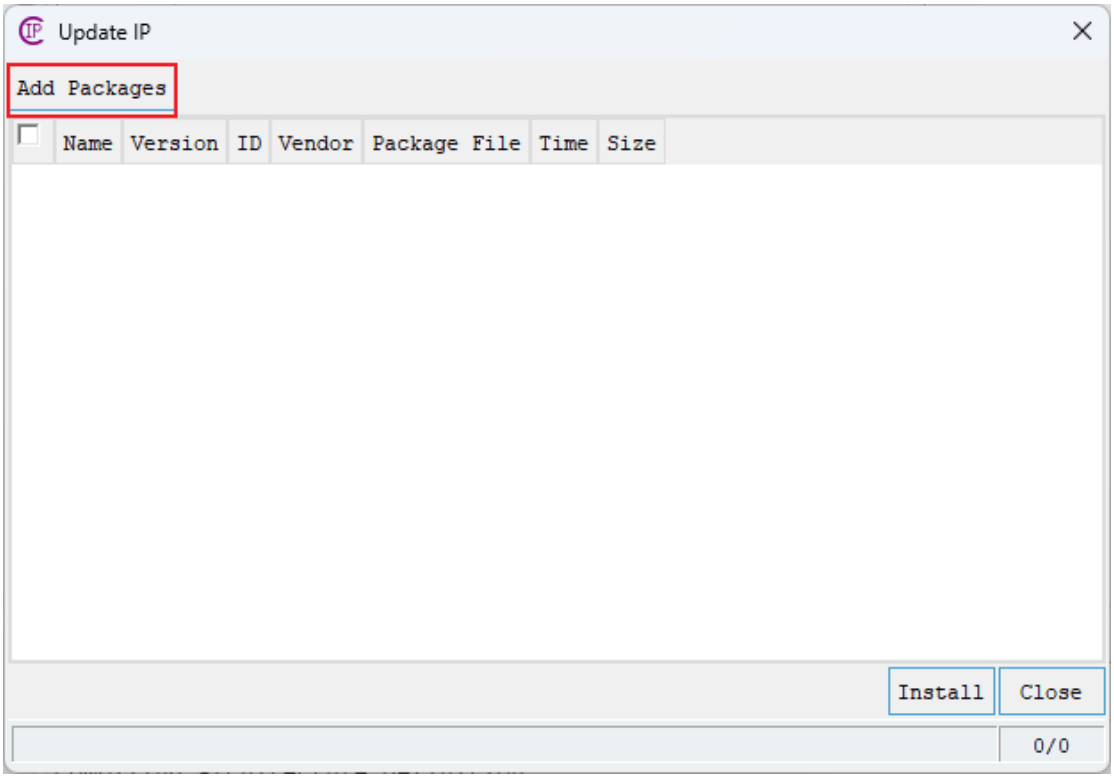


图 4.2-3

点击左上角 Add Package。

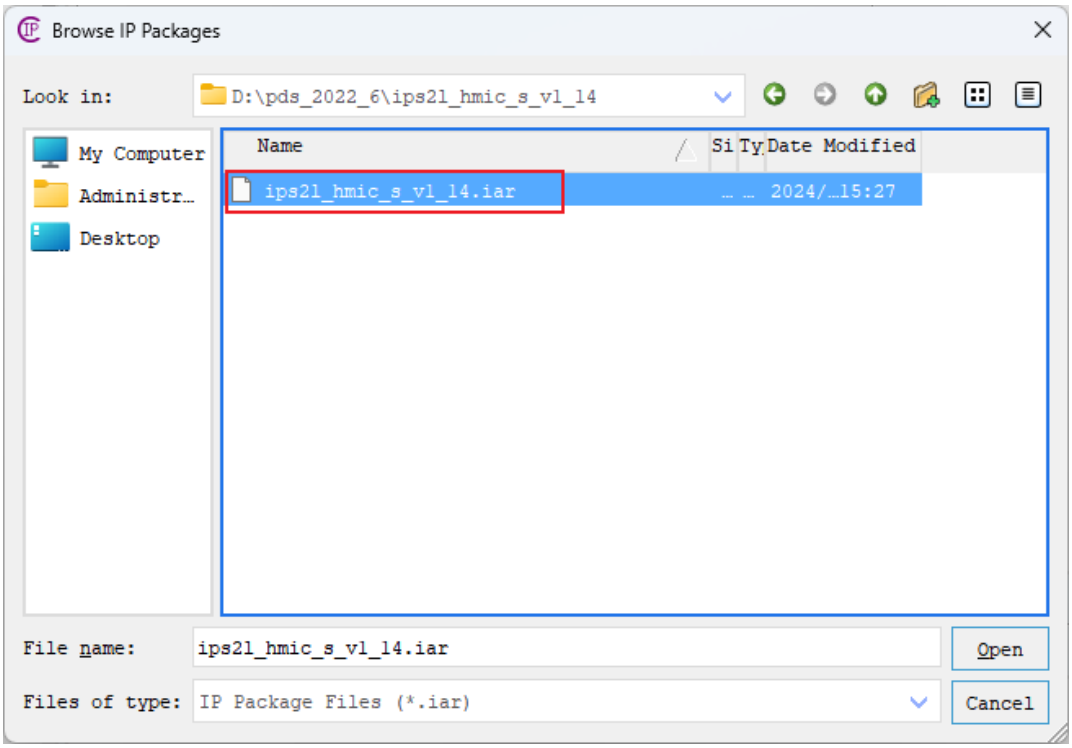


图 4.2-4

如图 4.2-4 是 DDR3 IP 的安装文件，后缀都是.iar。大家选择对应的文件后，点击右下角的 Open 即可。

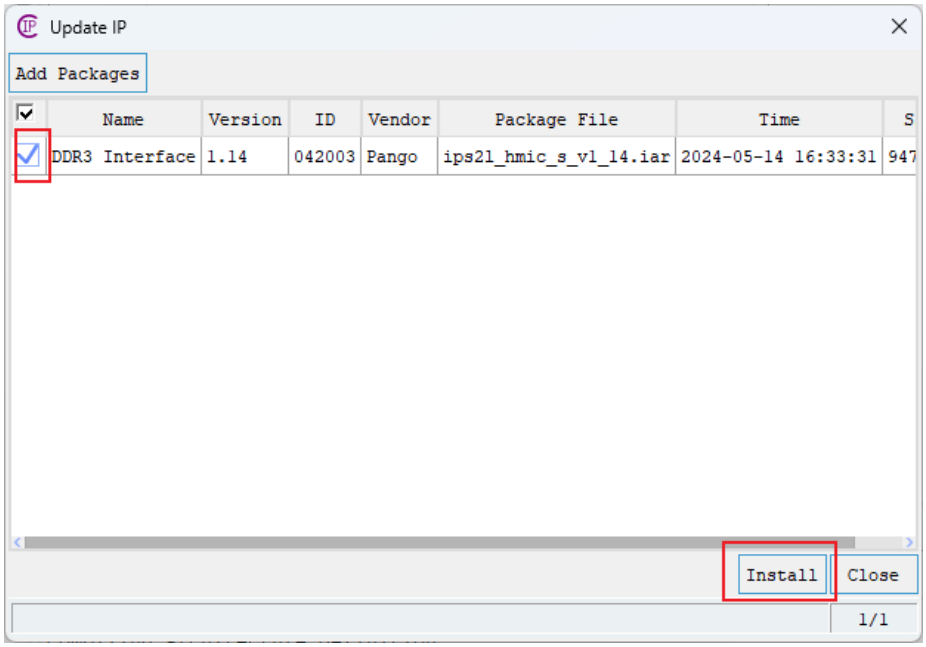


图 4.2-5

之后勾选上前面的√，点击 Install 即可。

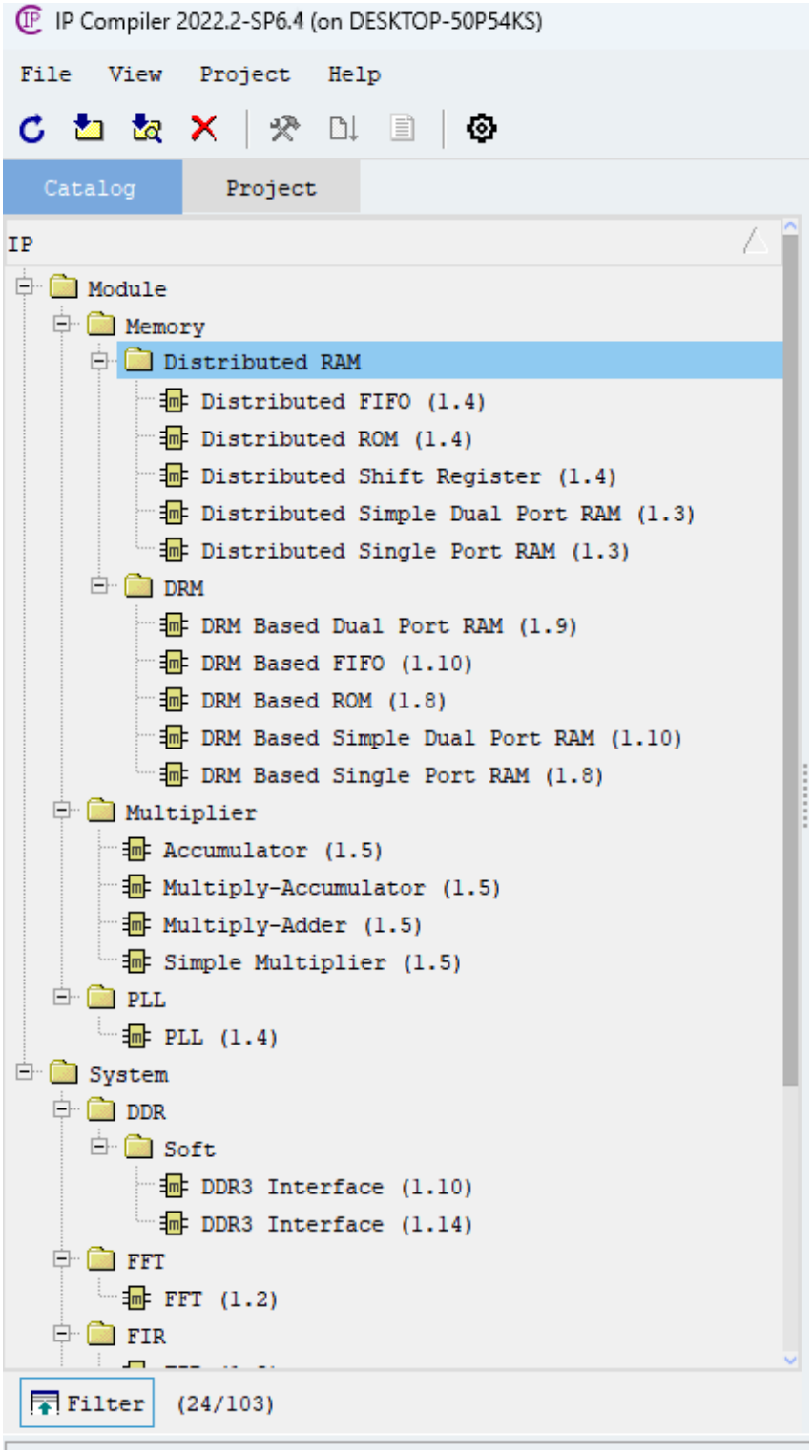


图 4.2-6

之后在左边的界面可以看到刚才安装的 IP 即可。注意如果发现安装后，弹出了警告，并且左边的界面没有任何变化。那就意味着你安装的 IP 该系列的器件不支持。因为你的工程可能是 LOGOS、LOGOS2、或者 Tian2 等系列，不同芯片型号所用的 IP 是不太相同的，所以大家注意这一点。

4.2.2. 例化 IP 及查看 IP 手册

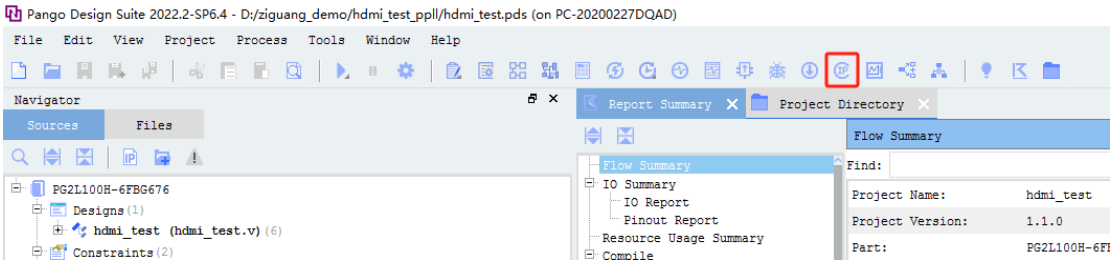


图 4.2-7

继续点击图 4.2-7 所示红框的图标。

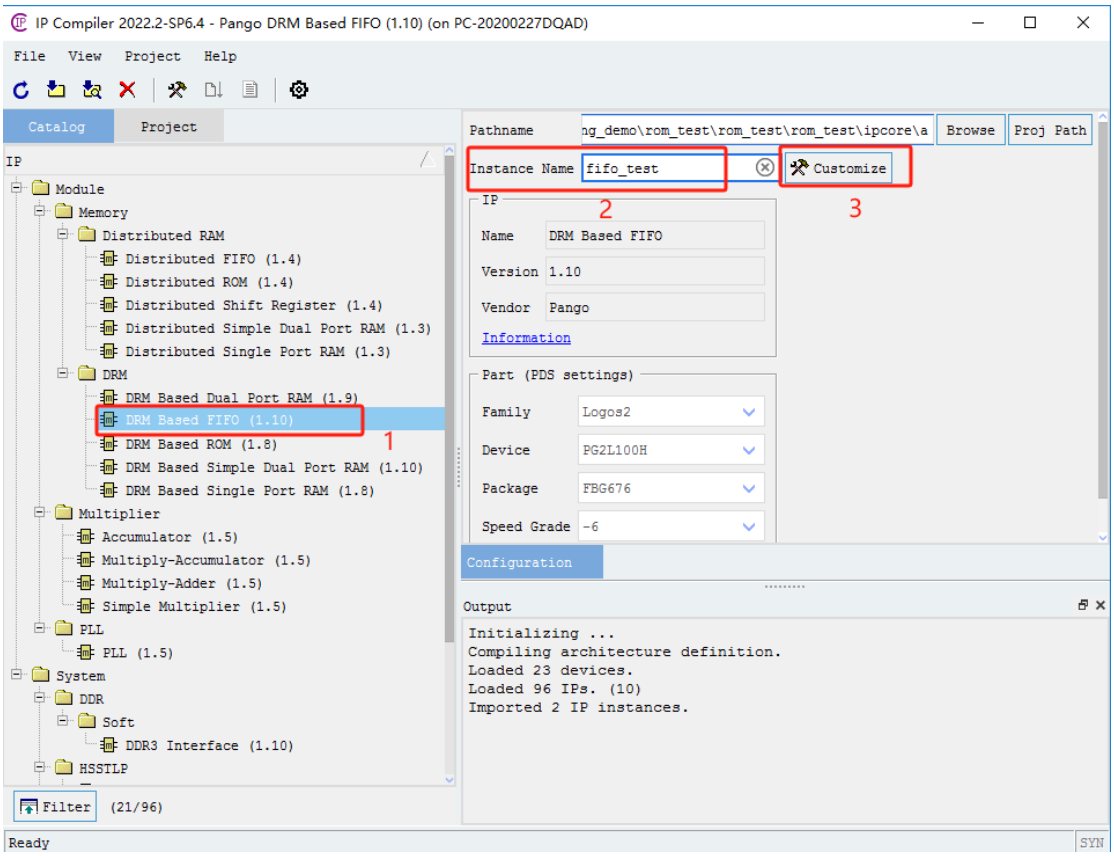


图 4.2-8

选择想要生成的 IP，这里以 FIFO 为例子，即红框 1 所示。红框 2 是用来填写生成的 IP 的名字。点击红框 3 后即可生成 IP，并弹出该 IP 的配置界面。如图 4.2-9 所示：

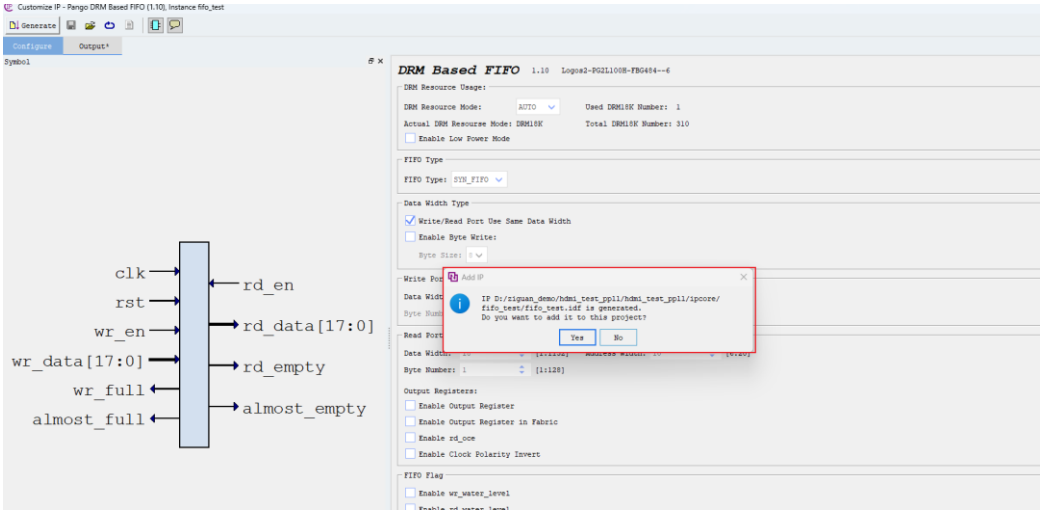


图 4.2-9

其弹出的提示是询问我们是否要把该 IP 添加到工程中，点击 YES 就行。如果我们不知道 IP 如何使用，可以打开官方参考手册查看，如图 4.2-10 所示：

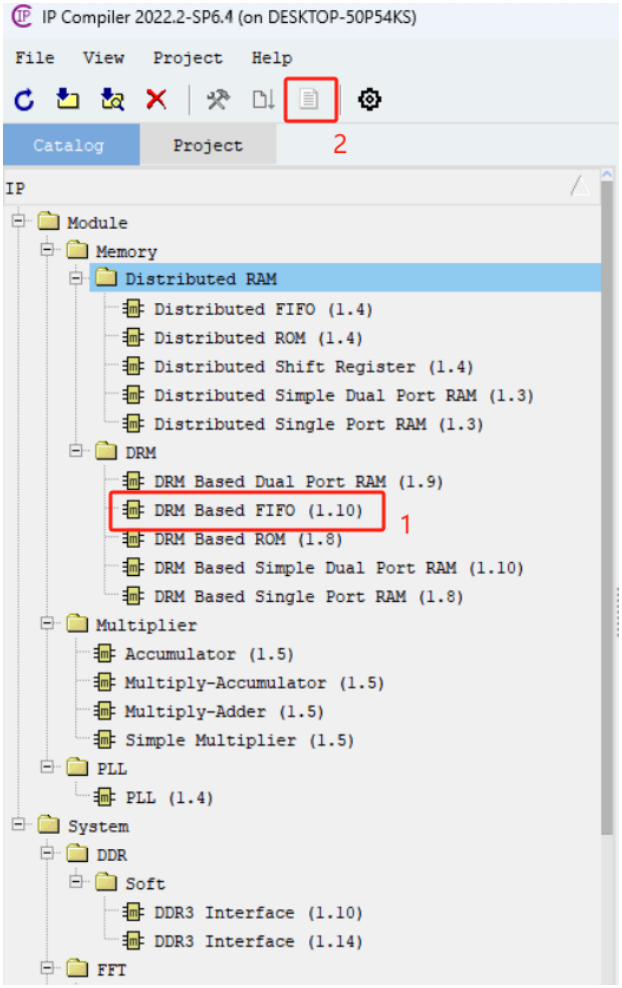


图 4.2-10

选择想要查看的 IP，如何点击红框 2 所示的图标，即可自动弹出官方参考文档。

图 4.2-11

对我们的 IP 配置完成后，点击左上角红框 1 处的 Generate 即可。

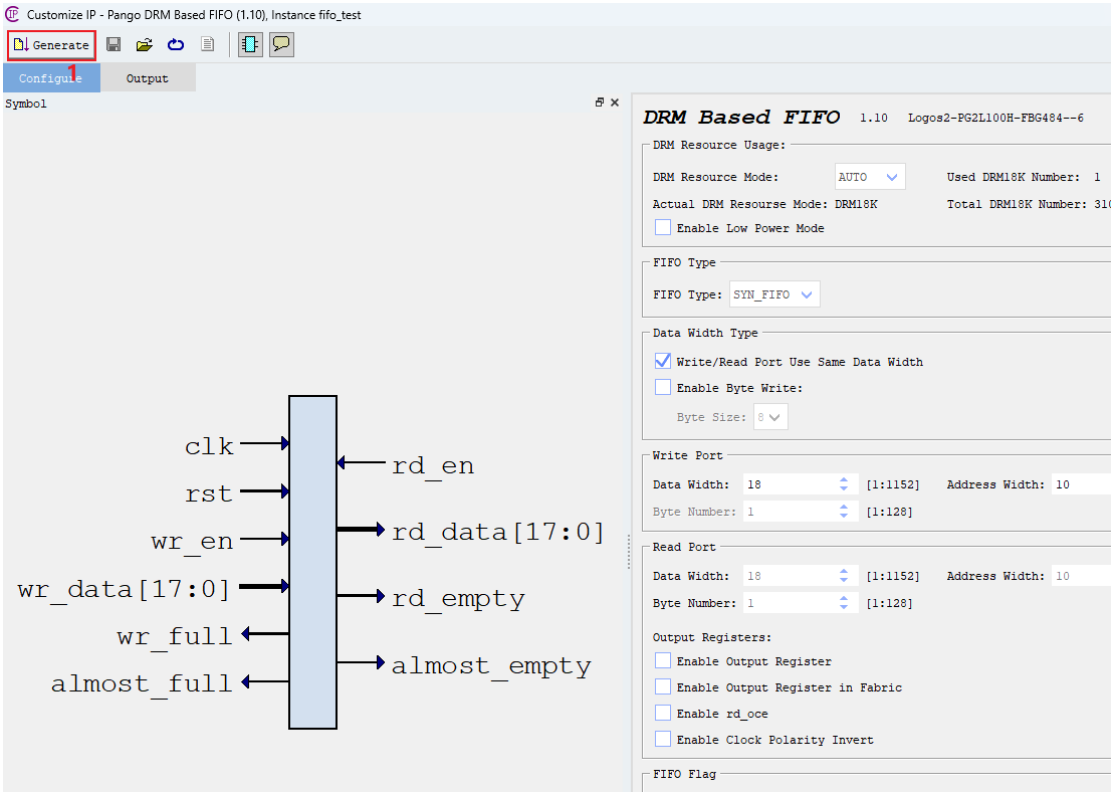


图 4.2-12

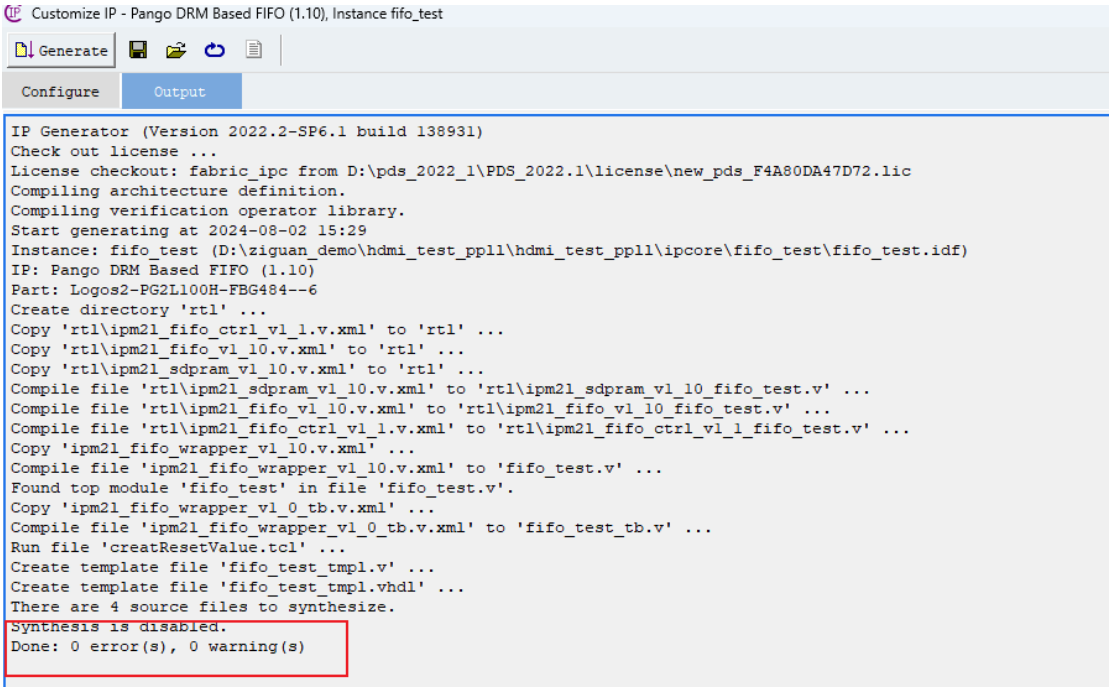


图 4.2-13

没有任何错误测表示生成成功。

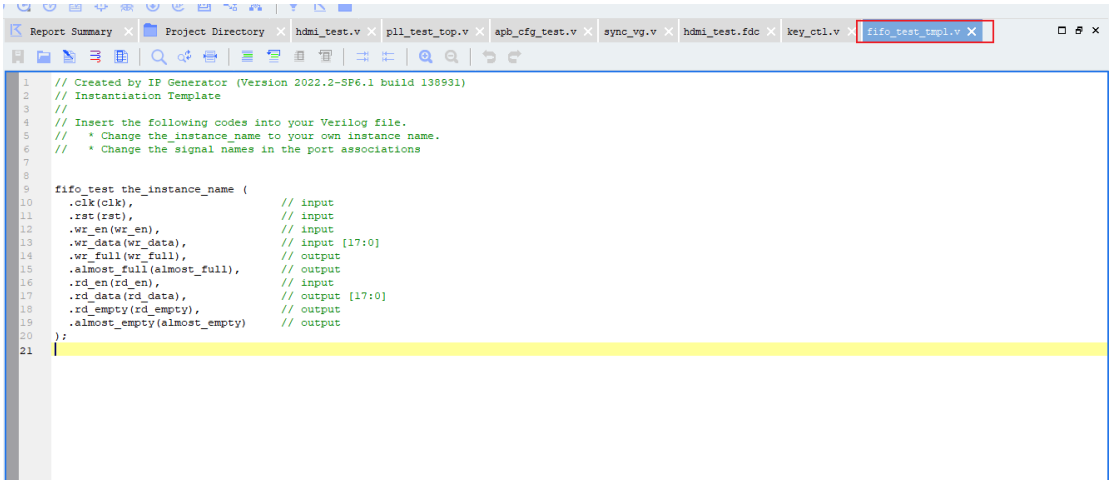


图 4.2-14

同时工具也会自动弹出一个 IP 的例化模板，供我们使用。只需要把该例化模板添加到自己的工程之中，即可使用我们生成的 IP。

5. Pango 的时钟资源--锁相环

5.1. 实验目的

了解 Logos2 系列的 PLL 的使用及配置方法。

5.2. 实验原理

5.2.1. PLL 介绍

锁相环作为一种反馈控制电路，其特点是利用外部输入的参考信号来控制环路内部震荡信号的频率和相位。因为锁相环可以实现输出信号频率对输入信号频率的自动跟踪，所以锁相环通常用于闭环跟踪电路。锁相环在工作的过程中，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，这就是锁相环名称的由来。

锁相环拥有强大的性能，可以对输入到 FPGA 的时钟信号进行任意分频、倍频、相位调整、占空比调整，从而输出一个期望时钟；除此之外，在一些复杂的工程中，哪怕我们不需要修改任何时钟参数，也常常会使用 PLL 来优化时钟抖动，以此得到一个更为稳定的时钟信号。正是因为 PLL 的这些性能都是我们在实际设计中所需要的，并且是通过编写代码无法实现的，所以 PLL IP 核才会成为程序设计中最常用 IP 核之一。

PLL IP 是紫光同创基于 PLL 及时钟网络资源设计的 IP，通过不同的参数配置，可实现时钟信号的调频、调相、同步、频率综合等功能。

5.2.2. IP 配置

首先点击快捷工具栏的“IP”图标，进入 IP 例化设置

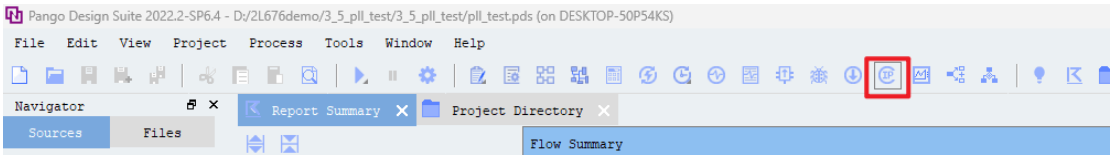


图 5.2-1 “IP”图标示意图

然后在 IP 目录处选择 PLL，在 Instance name 处为本次实例化的 IP 取一个名字，接着点击 Customise 进入 IP 配置页面。操作示意图如下：

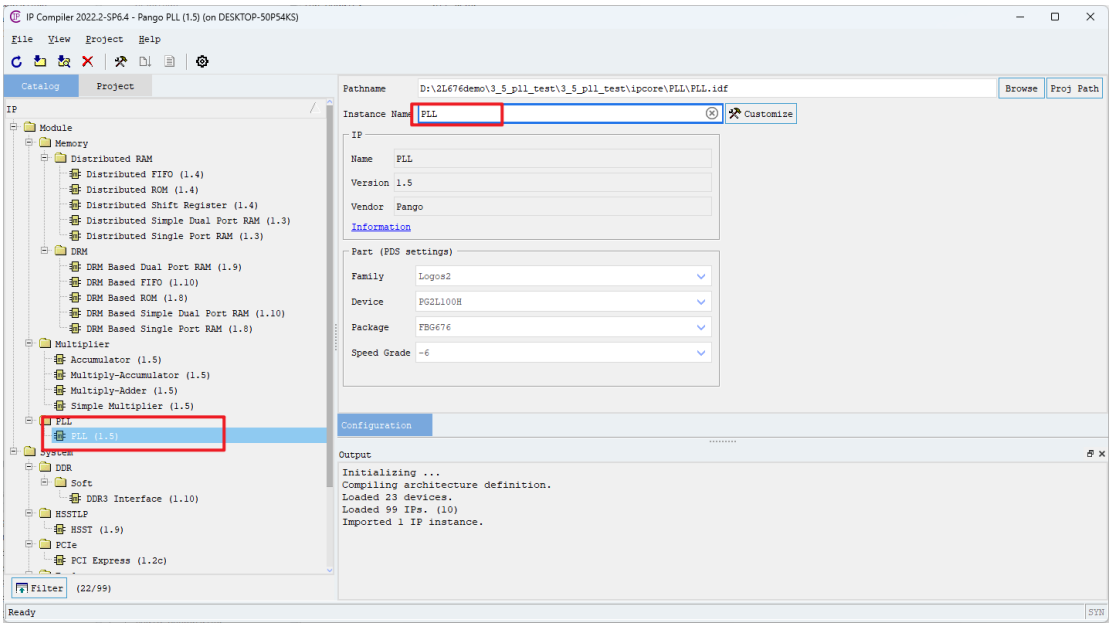


图 5.2-2 IP 配置流程图 1

PLL 的使用可选择 Basic 和 Advanced 两种模式，Advanced 模式下 PLL 的内部参数配置完全开放，需要自己填写输入分频系数、输出分频系数、占空比、相位、反馈分频系数等才能正确配置。Basic 模式下用户无需关心 PLL 的内部参数配置，只需输入期望的频 率值、相位值、占空比等，IP 将自动计算，得到最佳的配置参数。如果没有特殊应用，建议使用 Basic 模式配置 PLL。本次实验我们选择 Basic Configuration。

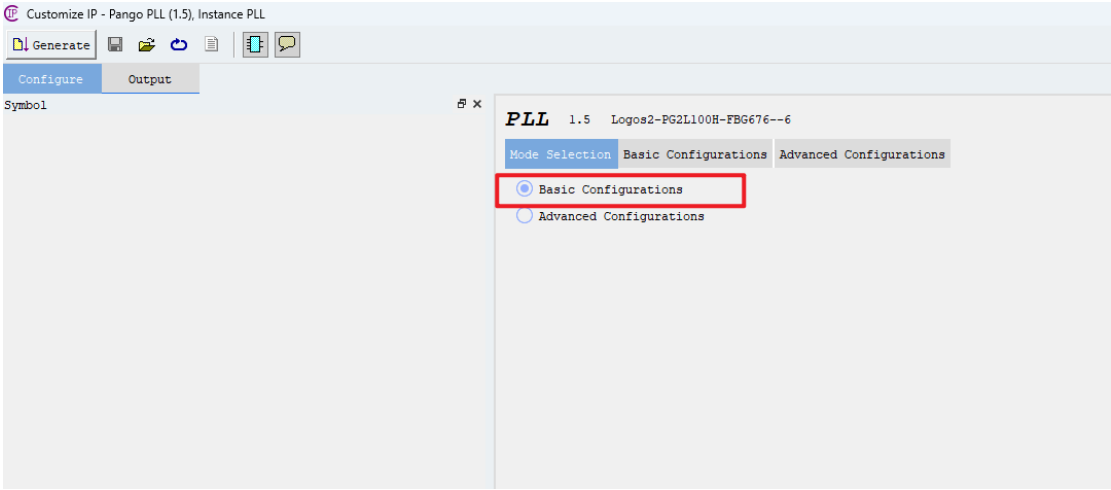


图 5.2-3 IP 配置流程图 2

接下来进行基础配置：

在 Public Configurations 一栏将输入时钟频率设置为 27MHZ。

在 Clockout0 Configurations 选项卡下，勾选使能 clkout0，将输出频率设置为 54MHZ。

在 Clockout1 Configurations 选项卡下，勾选使能 clkout1，将输出频率设置为 81MHZ。

在 Clockout2 Configurations 选项卡下，勾选使能 clkout2，将输出频率设置为 81M HZ，并设置相位偏移为 180 度。

其他选项可以使用默认设置，若有其他需求可以查阅 IP 手册了解，本实验我们暂介绍 IP 基本的使用方法：

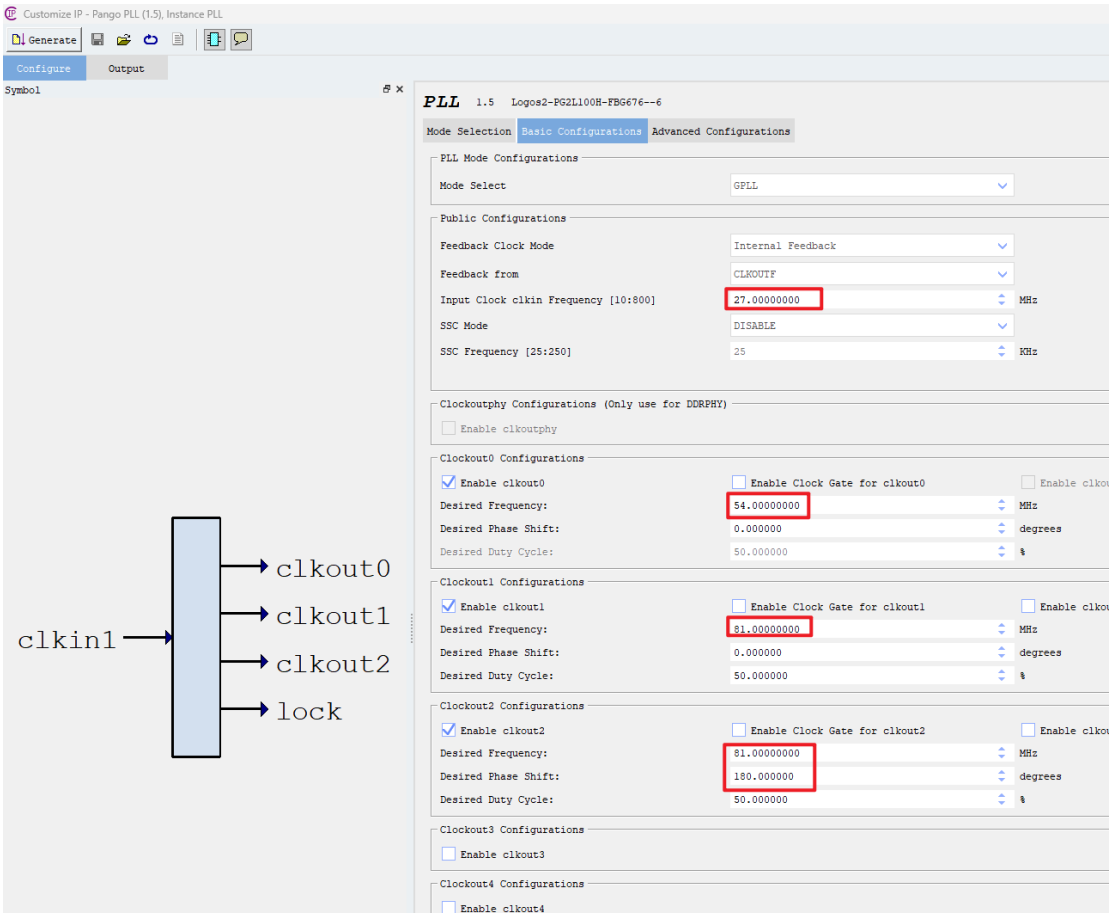


图 5.2-4 IP 配置流程图 3

点击左上角 generate 生成 IP。

5.3. 代码设计

模块接口列表如下所示：

表 5.3-1 PLL IP 使用实验模块接口表

端口	I/O	位宽	描述
sys_clk	input	1	系统时钟
clkout0	output	1	54MHZ 时钟
clkout1	output	1	81MHZ 时钟
clkout2	output	1	81MHZ 时钟，相位偏移 180 度
lock	output	1	时钟锁定信号，当为高电平时，代表 IP 核输出时钟稳定。

PLL_TEST 顶层代码：

```
1.  module PLL_TEST(
2.      input                       sys_clk             ,
3.      output                      clkout0             ,
4.      output                      clkout1             ,
5.      output                      clkout2             ,
6.      output                      lock
7.    );
8.
9.  PLL PLL_U0 (
10.     .clkout0                    (clkout0            ),// output
11.     .clkout1                    (clkout1            ),// output
12.     .clkout2                    (clkout2            ),// output
13.     .lock                       (lock               ),// output
14.     .clkin1                     (sys_clk            ) // input
15. );
16.
17.
18. endmodule
19.
```

该模块的功能是例化 PLL IP 核，功能简单，在此不做说明。

PLL_tb 测试代码：

```
1.  timescale 1ns / 1ps
2.
3.  module    PLL_tb();
4.      reg                         sys_clk             ;
5.      wire                        clkout0             ;
6.      wire                        clkout1             ;
7.      wire                        clkout2             ;
8.      wire                        lock                ;
9.
10.
```

```
11.
12.     initial
13.         begin
14.             #2
15.                     sys_clk <= 0      ;
16.         end
17.
18.     parameter    CLK_FREQ = 27;//Mhz
19.     always # ( 1000/CLK_FREQ/2 ) sys_clk = ~sys_clk ;
20.
21.
22. PLL_TEST u_PLL_TEST(
23.     .sys_clk                    (sys_clk                    ),
24.     .clkout0                    (clkout0                    ),
25.     .clkout1                    (clkout1                    ),
26.     .clkout2                    (clkout2                    ),
27.     .lock                       (lock                      )
28. );
29.
30.
31. endmodule
32.
```

timescale 定义了模块仿真的时间单位和时间精度。时间单位是 1 纳秒，精度是 1 皮秒。

initial 块负责初始化系统时钟。在仿真启动后的 2 纳秒，系统时钟 sys_clk 被设置为 0。这是为了在仿真开始时定义一个已知的初始状态。

代码定义了一个时钟频率参数 CLK_FREQ 为 27 MHz，并使用一个 always 块来翻转系统时钟信号。always 块中的逻辑使得 sys_clk 每 37 纳秒翻转一次，从而生成一个 27 MHz 的方波时钟信号。这种时钟信号用于驱动被测试的 PLL_TEST 模块。

最后，将测试平台的各个信号连接到 PLL_TEST 模块。这包括将生成的系统时钟 sys_clk 连接到 PLL_TEST 的时钟输入端，并将 PLL_TEST 的输出信号 clkout0、clkout1、clkout2 和 lock 使用 wire 引出观察。

5.4. PDS 与 Modelsim 联合仿真

PDS 支持与 Modelsim 或 QuestaSim 等第三方仿真器的联合仿真，而 Modelsim 是较为常用的仿真器，使用 PDS 与 Modelsim 来进行联合仿真。

接下来选择 Project->Project Setting，打开工程设置，准备设置联合仿真。

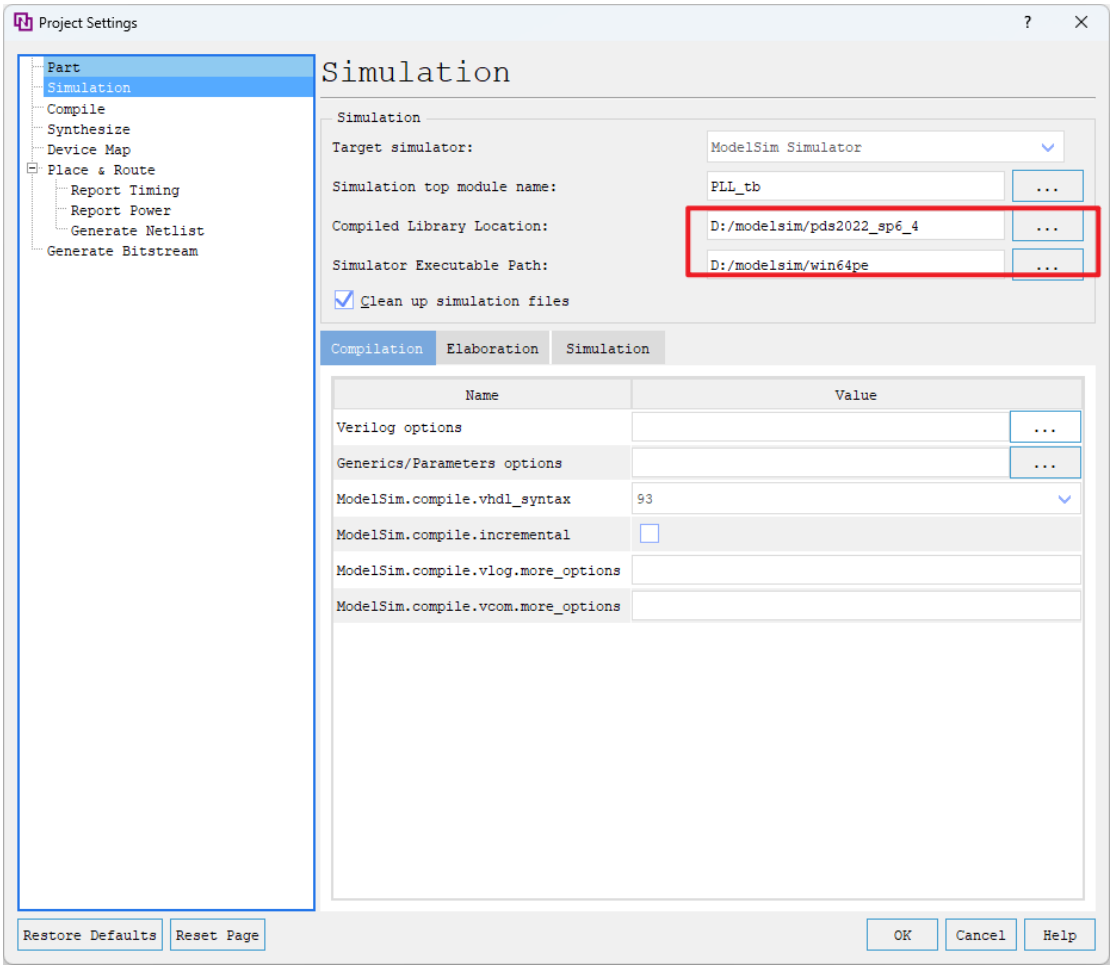


图 5.4-1 PDS 和 Modelsim 联合仿真流程图 4

选择 Simulation 选项卡,红框 1 选择刚才编译生成的仿真库的路径,红框 2 选择 Modelsim 的启动路径,之后点击 OK。

右键仿真的文件,选择 Run Behavior Simulation 开始行为仿真。

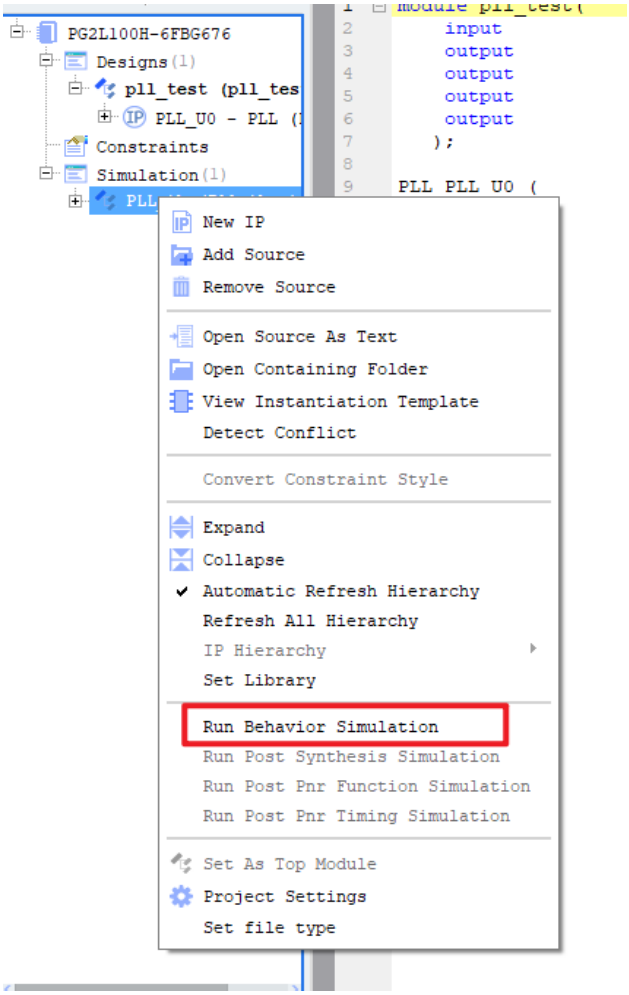


图 5.4-2 PDS 和 Modelsim 联合仿真流程图 5

运行后会自动打开 Modelsim。并执行仿真,如果没有任何报错,则表示成功。如果出现错误，请检测 PDS 与 Modelsim 的配置。

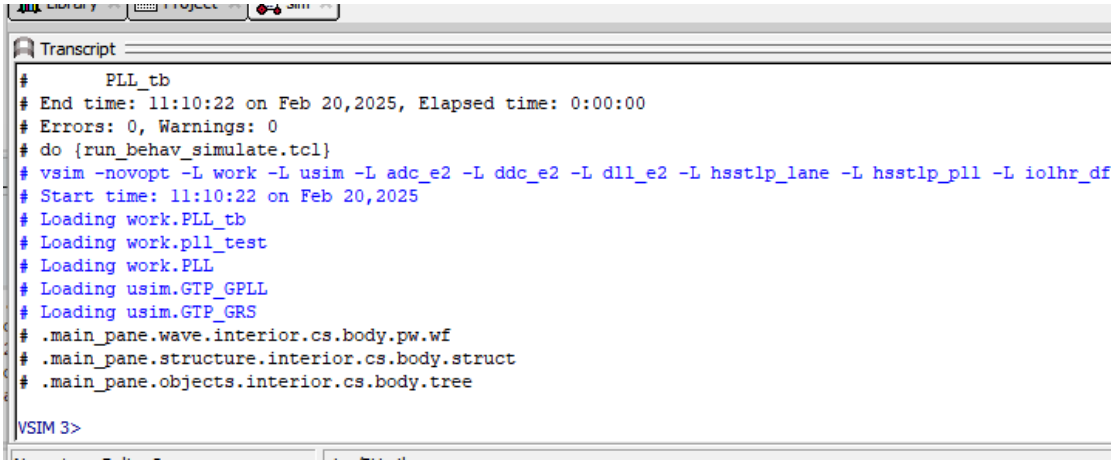


图 5.4-3 PDS 和 Modelsim 联合仿真流程图 6

5.5. 实验现象

点击 Wave 观察 PLL 输出信号：

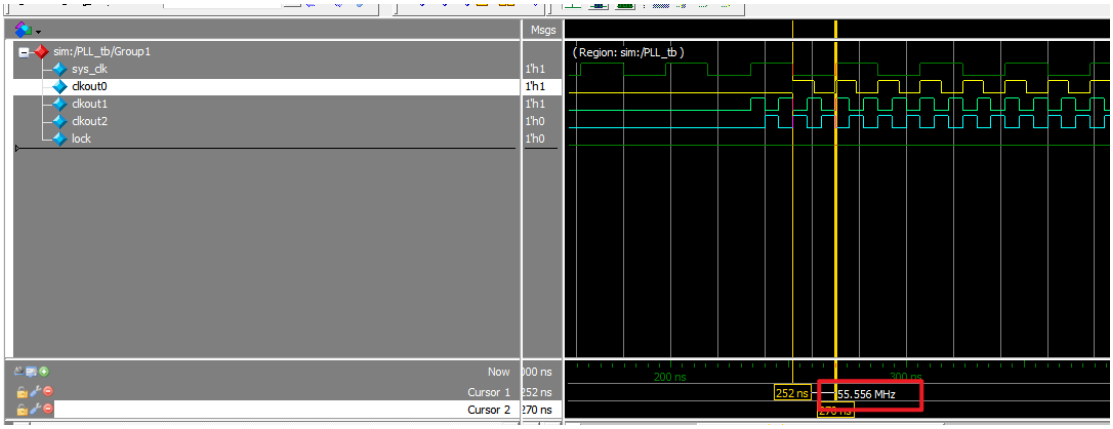


图 5.5-1 PLL IP 使用实验结果波形图 1

使用标尺测量 clkout0，发现其一个时钟周期是 18ns，也就是 55.556MHZ。出现了偏差是因为 tb 生成的 27MHZ 其实并不准确，所以导致误差。

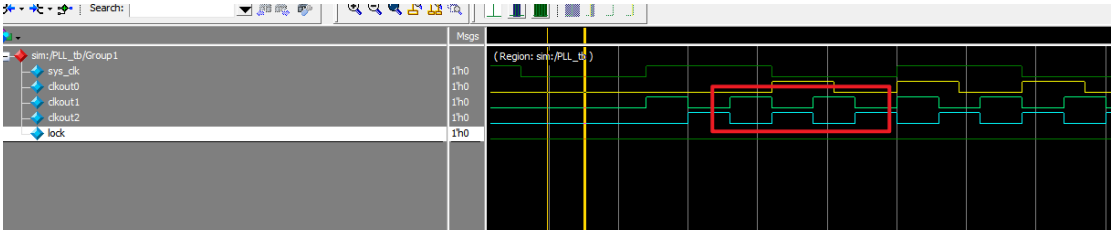


图 5.5-2 PLL IP 使用实验结果波形图 2

使用可以看到 clkout1 和 clkout2 相位偏差 180°，符合设置。需要注意 PLL 的输出时钟应该在时钟锁定信号 lock 有效之后才能使用，lock 信号拉高之前输出的时钟是不确定的。

6. Pango 的 ROM、RAM、FIFO 的使用

6.1. 实验简介

实验目的:

掌握紫光平台的 RAM、ROM、FIFO IP 的使用

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

暂无

6.2. 实验原理

不管是 Logos 系列或者是 Logos2 系列，其 IP 配置以及模式和功能均一致，不会像 PLL 那样有动态配置以及内部反馈选项的选择等之间的差异，所以是 RAM、ROM、FIFO 是通用的。

6.2.1. RAM 介绍

RAM 即随机存取存储器。它可以在运行过程中把数据写进任意地址，也可以把数据从任意地址中读出。其作用可以拿来做数据缓存，也可以跨时钟，也可以存放算法中间的运算结果等。

注意，PDS 的 IP 配置工具中提供两种不同的 RAM，一种是 Distributed RAM(分布式 RAM)另一种是 DRM Based RAM，分布式 RAM 用的是 LUT(查找表)资源去构成的 RAM，这种 RAM 会消耗大量 LUT 资源，因此通常在一些比较小的存储才会用到这种 RAM，以节省 DRM 资源。而 DRM Based RAM 是利用片内的 DRM 资源去构成的 RAM，不占用逻辑资源，而且速度快，通常设计中均使用 DRM Based RAM。

RAM 分为三种，如下表所示:

表 6.2-1

RAM 类型	特点
单端口 RAM	只有一个端口可以读写。只有一个读写口和地址口
伪双端口 RAM	有 wr 和 rd 两个端口，顾名思义，wr 只能写，rd 只能读
真双端口 RAM	提供 A 和 B 两个端口，两个端口均可以独立进行读写

注意，当使用真双端口时，要避免出现同时读写同个地址，这会造成写入失败，在

逻辑设计上需要避开这个情况。

以下给出比较常用的 RAM 的配置作为介绍，通常我们比较常用伪双端口 RAM 来设计，如图 6.2-1 所示：

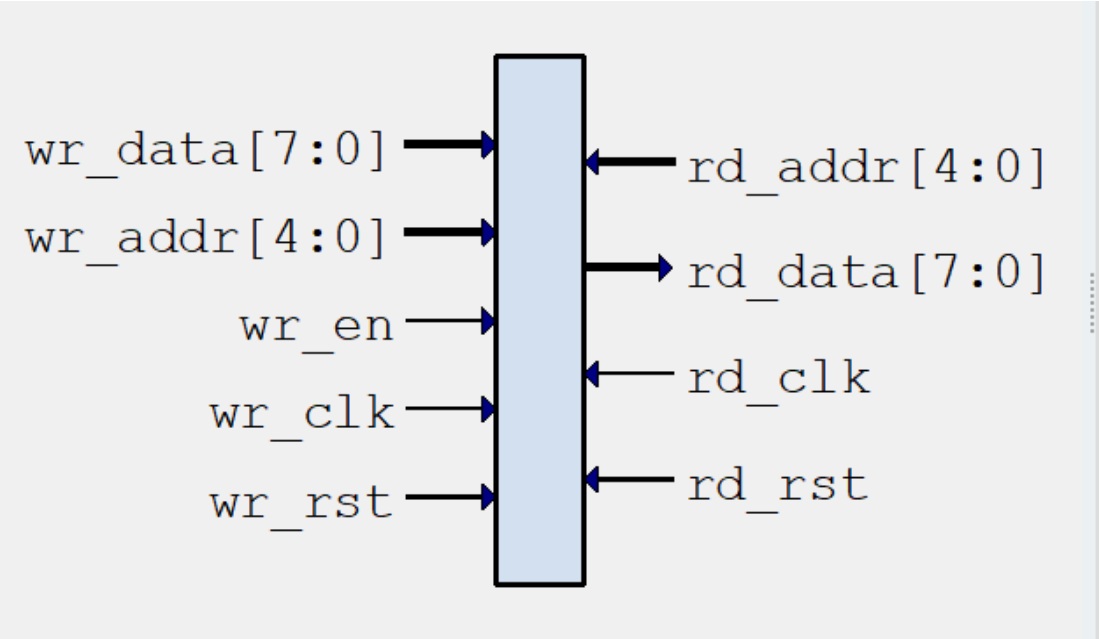


图 6.2-1

图 6.2-2 为 IP 配置：

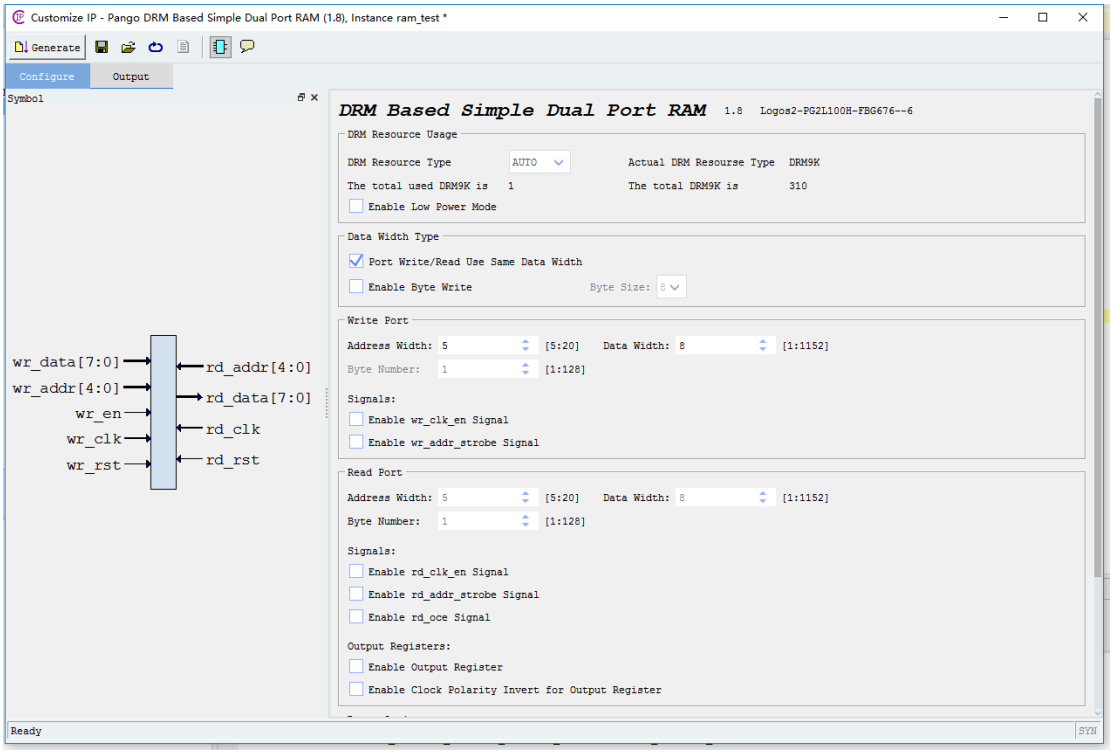


图 6.2-2

注意，如果勾选 Enable Output Register(输出寄存)，输出数据会延迟一个时钟周期。

具体每个端口的含义这里参考官方手册，大家也可以自行查看 IP 手册，如下图所示:

端口名	输入/输出	说明
wr_data	输入	写数据信号，位宽范围1~1152
wr_addr	输入	写地址信号，位宽范围5~20
wr_en	输入	写使能信号 1：写使能 0：读使能
wr_clk	输入	写时钟信号
wr_clk_en	输入	写时钟使能信号 1：对应地址有效 0：对应地址无效
wr_rst	输入	写端口复位信号，高有效
wr_byte_en	输入	Byte Write使能信号，当配置“Enable Byte Write”选项勾选时有效，位宽范围1~128。 1：对应Byte值有效； 0：对应Byte值无效
wr_addr_strobe	输入	写地址锁存信号 1：对应地址无效，上一个地址被保持 0：对应地址有效
rd_data	输出	读数据信号，位宽范围1~1152
rd_addr	输入	读地址信号，位宽范围5~20
rd_clk	输入	读时钟信号
rd_clk_en	输入	读时钟使能信号。 1：对应地址有效； 0：对应地址无效。
rd_rst	输入	读端口复位信号，高有效
rd_oce	输入	读数据输出寄存使能信号 1：对应地址有效，读数据寄存输出 0：对应地址无效，读数据保持
rd_addr_strobe	输入	读地址锁存信号 1：对应地址无效，上一个地址被保持 0：对应地址有效

图 6.2-3

DRM Resource Type：用于配置所建 RAM IP 核用的是哪种资源，不同芯片型号可选资源是不一样的，有的是 9K,有的是 18K,有的是 36K,如果没有特殊情况，直接 AUTO 即可。

6.2.1.1. RAM 的读写时序

配置成不同模式的时候，RAM 的读写时序是不一样的，真双端口和单端口的 RAM

配置均有三种模式，而伪双端口只有一种。由于真双端口和单端口的配置是一样的，这里以真双端口为例子。

分为 NORMAL_WRITE(正常模式)、TRANSPARENT_WRITE(直写)、READ_BEFORE_WRITE(读优先模式)三种模式。

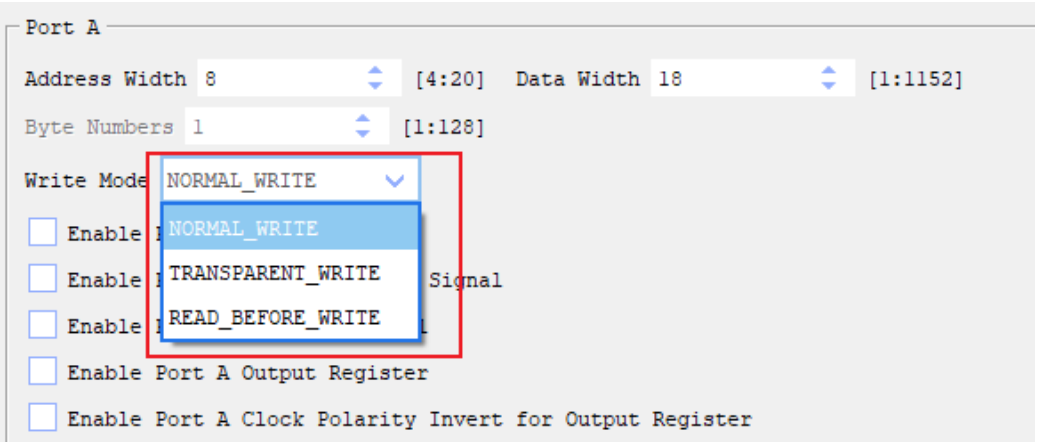


图 6.2-4

而伪双端口不属于上面三种模式，有它独特的模式。这几种模式的差异就在于读写时序的不同，接下来，我们来分析读写时序。

以下时序图均来自官方 IP 手册，并且均未使能输出寄存。注意 wr_en 为 1 时表示写数据，为 0 表示读数据。

6.2.1.1.1. NORMAL_WRITE

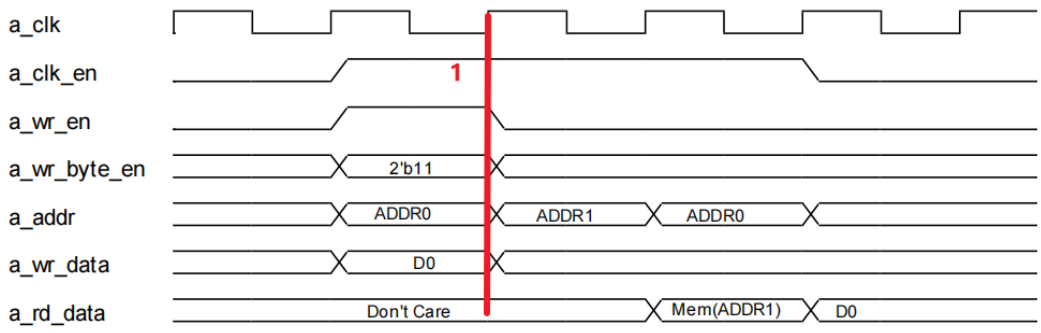


图 6.2-5

在 NORMAL_WRITE 这种模式下，可以看到，当时钟的上升沿到来，且 clk_en 和 wr_en 均为高电平时，就会把数据写到对应的地址里面，如图中的 1 时刻。然后看读数据端口，当 wr_en 不为 0 的时候，a_rd_data 一直为 Don't Care 状态，而当时钟上升沿到来，且 clk_en 为高电平，wr_en 为低电平时，a_rd_data 输出当前 a_addr 里的数据，即 Mem(ADDR1)和 ADDR0 里的 D0。

6.2.1.1.2. READ_BEFORE_WRITE

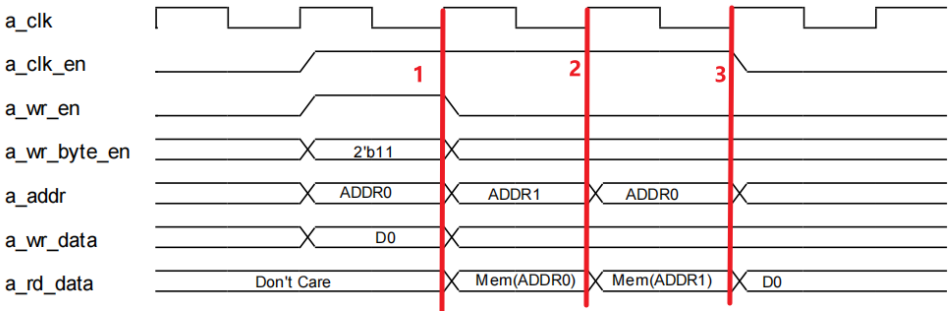


图 6.2-6

在 READ_BEFORE_WRITE 这种模式下，可以看到在 1 的时刻，时钟上升沿到来，且 clk_en 和 wr_en 均为高电平，D0 写进了 ADDR0 里面，但是注意看此时的 a_rd_data 和 a_addr，可以发现，此时 a_wr_en 并不为 0，可 a_rd_data 还是输出了上一刻 ADDR0 的数据(因为不是输出 D0)。之后，a_wr_en 拉低，此时才是读数据，在 3 时刻，把 ADDR0 的数据读出来，a_rd_data 才输出了 D0。

所以总结一下，这个模式其实就是进行写操作时，读端口会把当前写的地址的原始数据输出，因此叫读优先模式很合情合理对吧，顾名思义，就是我优先把原来的数据读出来。

6.2.1.1.3. Transparent_Write

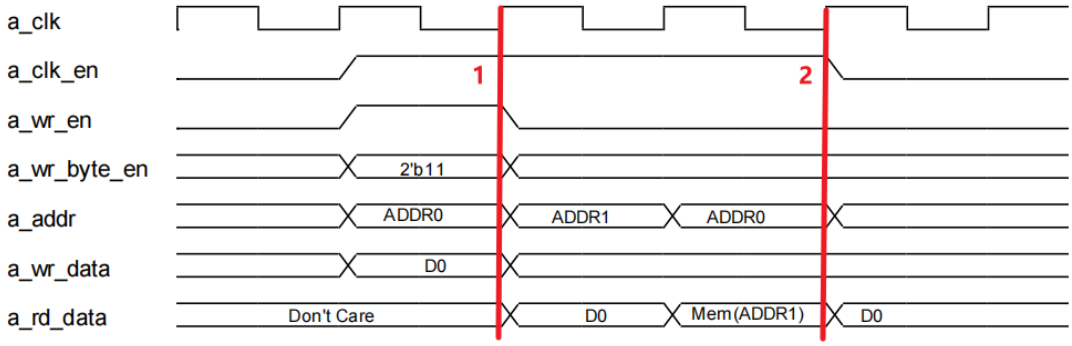


图 6.2-7

在 Transparent_Write 这种模式下，可以看到在 1 的时刻，时钟上升沿到来，且 clk_en 和 wr_en 均为高电平，D0 写进了 ADDR0 里面，但是注意看此时的 a_rd_data 和 a_addr，可以发现，此时 a_wr_en 并不为 0，可 a_rd_data 居然直接输出了 D0，之后 a_wr_en 拉低，进入读状态，在 2 时刻，再一次把 ADDR0 的数据读出来，输出了 D0。

分析总结一下，根据 1 时刻的情况，我们可以得出结论，在这种模式下，当我们进行写操作时，读端口会马上输出我们写入的数据。所以叫直写模式。

6.2.1.1.4. 伪双端口的读写时序

注意：wr_en 为 1 时是写操作，为 0 是读操作。

伪双端口的读写时序与上面三种都不同，我们看图 8 的时序来分析：

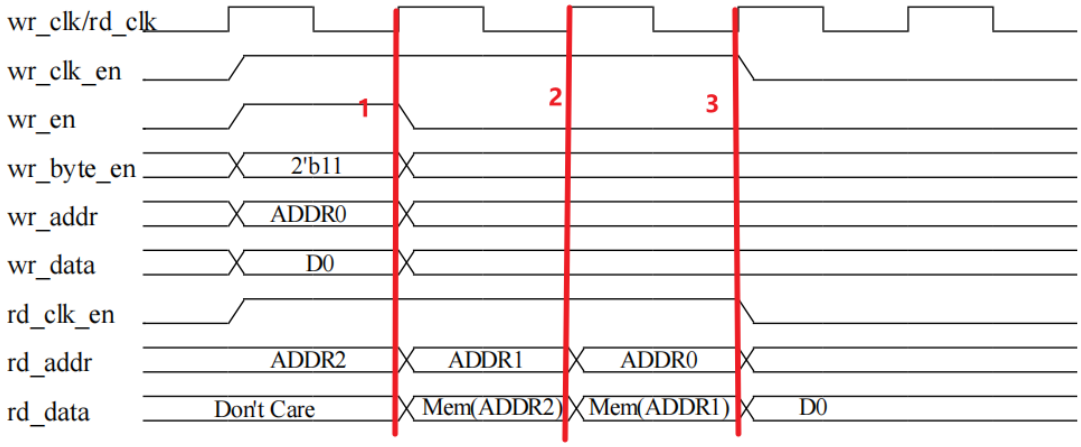


图 6.2-8

注意看 1 时刻，此时 wr_en 和 wr_clk_en 均为高电平，所以是写操作，所以 1 时刻就是往地址 ADDR0 里写入 D0，注意此时的 rd_addr 和 rd_data，可以看到这一时刻 rd_addr 是 ADDR2，然后进行写操作时，rd_data 同样输出了 ADDR2 里的数据，而此时 wr_en 还是高

电平。接下来看 2 和 3 时刻，此时 wr_en 为 0，rd_clk_en 是高电平，所以是读操作，此时分别读出 ADDR1 和 ADDR0 里的数据，之后 rd_clk_en 变成低电平，读时钟无效，可以看到 rd_data 保持 D0 输出。

分析总结一下，主要是 1 时刻，大家可以看到 1 时刻往 ADDR0 写入了 D0，读端口却输出了 ADDR2 中的数据。仔细观察可以得出结论：伪双端口 RAM 在进行写操作的时候，会把当前读端口指向的地址的数据输出。是不是有点像直写？只不过直写是输出写入的数据，而伪双端口是输出读端口指向的地址的数据。

具体大家可以结合视频讲解。

2.1.1.2 ROM 介绍

ROM 即只读存储器，在程序的运行过程中他只能被读取，无法被写入，因此我们应该在初始化的时候就给他配置初值，一般是在生成 IP 的时候通过导入.dat 文件对其进行初值配置。

注意，PDS 的 IP 配置工具中提供两种不同的 ROM，一种是 Distributed ROM(分布式 ROM)另一种是 DRM Based ROM，分布式 ROM 用的是 LUT(查找表)资源去构成的 ROM，这种 ROM 会消耗大量 LUT 资源，因此通常在一些比较小的存储才会用到这种 RAM，以节省 DRM 资源。而 DRM Based ROM 是利用片内的 DRM 资源去构成的 ROM，不占用逻辑资源，而且速度快，通常设计中均使用 DRM Based ROM。

以下给出比较常用的 ROM 的配置作为介绍，由于只能读，因此其均为单端口 ROM 如图 6.2-9 所示：

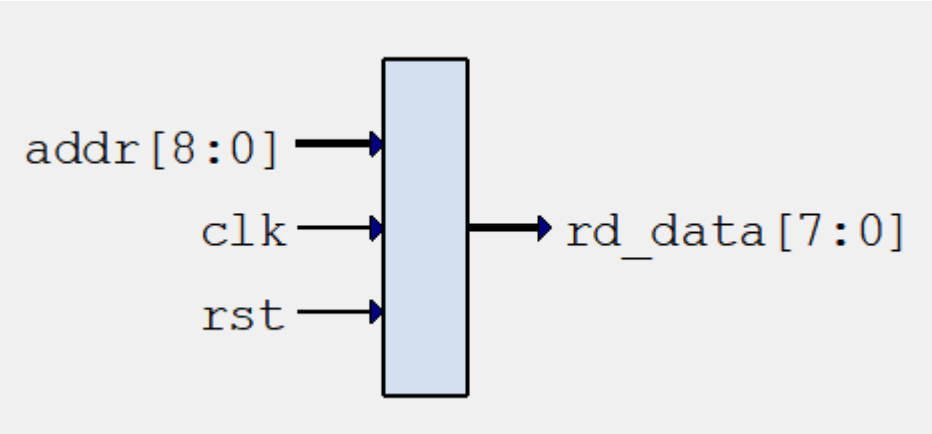


图 6.2-9

下图为 IP 配置:

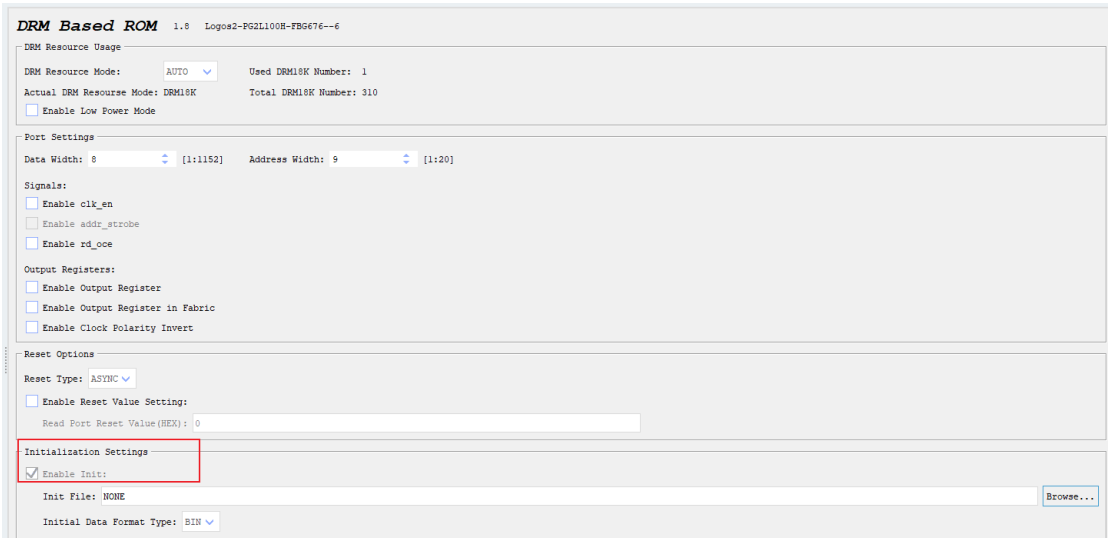


图 6.2-10

注意,如果勾选 Enable Output Register(输出寄存),输出数据会延迟一个时钟周期。同时,可以看到 Enable Init 选项是默认勾选的,并且不可取消。

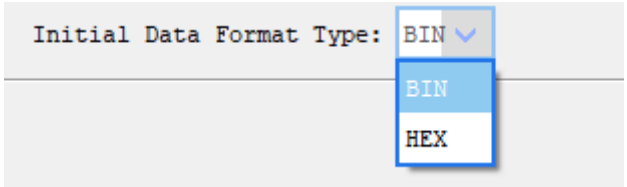


图 6.2-11

导入的数据的格式只能为二进制或者是十六进制。

具体每个端口的含义这里参考官方手册,大家也可以自行查看 IP 手册,如图 6.2-12 所示:

端口	I/O	描述
addr	I	读地址信号。
addr_strobe	I	读地址锁存信号。 1：对应地址无效，地址被保持； 0：对应地址有效。
rd_data	O	读数据信号。
clk	I	时钟信号。
clk_en	I	时钟使能信号。 1：对应地址有效； 0：对应地址无效。
rst	I	复位信号。 1：复位； 0：复位释放。
rd_oce	I	输出寄存使能信号。 1：读数据寄存输出； 0：寄存输出数据保持。

图 6.2-12

可以看到图 6.2-12 给出的是完整的接口列表，一般我们只需要 addr、rd_data、clk、rst 这四个信号即可。

以下时序图均来自官方 IP 手册，并且均未使能输出寄存。

6.2.1.2. ROM 的读时序

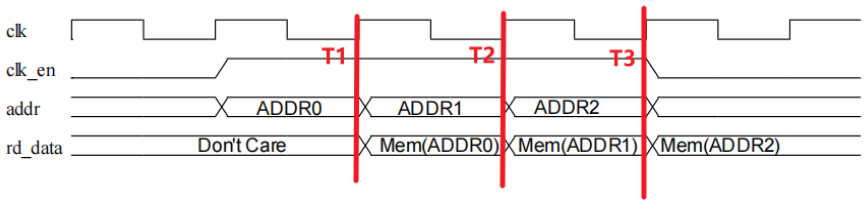


图 6.2-13

可以看到该时序是非常简单的，比如在 TI 时刻，当 clk 上升沿到来时，且 clk_en 为高电平时，给出要读出的地址，rd_data 就会输出数据，在不勾选输出使能寄存的情况下，rd_data 的输出会有延迟，具体时间可以从仿真里看到，所以我们在下个时钟周期的上升沿即 T2 时刻的上升沿才能获取到 ROM 读出的值。

所以整体时序非常简单，如果勾选了 clk_en 信号，就要给 clke_en 高电平才能读数据，如果不勾选 clk_en 信号，就一直根据地址读取 ROM 数据。

6.2.2. FIFO 介绍

FIFO 即先入先出，在 FPGA 中，FIFO 的作用就是对存储进来的数据具有一个先入先出特性的一个缓存器，经常用作数据缓存或者进行数据跨时钟域传输。FIFO 和 RAM

最大的区别就是 FIFO 不需要地址，采用的是顺序写入，顺序读出。

在紫光的 IP 工具中又分为 Distribute FIFO 和 DRM FIFO，其实就是用不同的资源去构成，前者 Distribute FIFO 也就是分布式 FIFO，使用的是片上的 LUT 资源去构成，而 DRM FIFO 使用的是片上的 DRM 资源去构成，DRM 构成的 FIFO 其性能大于 LUT 资源构成的，不仅容量更大，且可配置更多功能。

本章着重介绍 DRM Based FIFO。

注意：FIFO 写满后禁止继续写入数据，否则将会写溢出。

注意：FIFO 读空后禁止继续读数据，否则将会读溢出。

以下给出常用的 FIFO 的配置作为介绍。

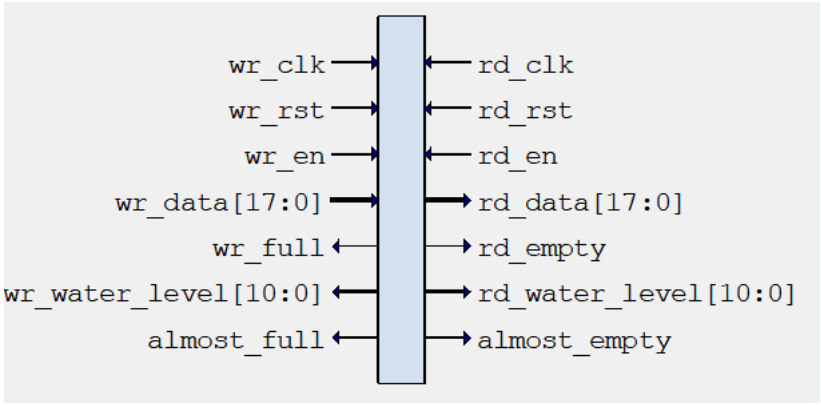


图 6.2-14

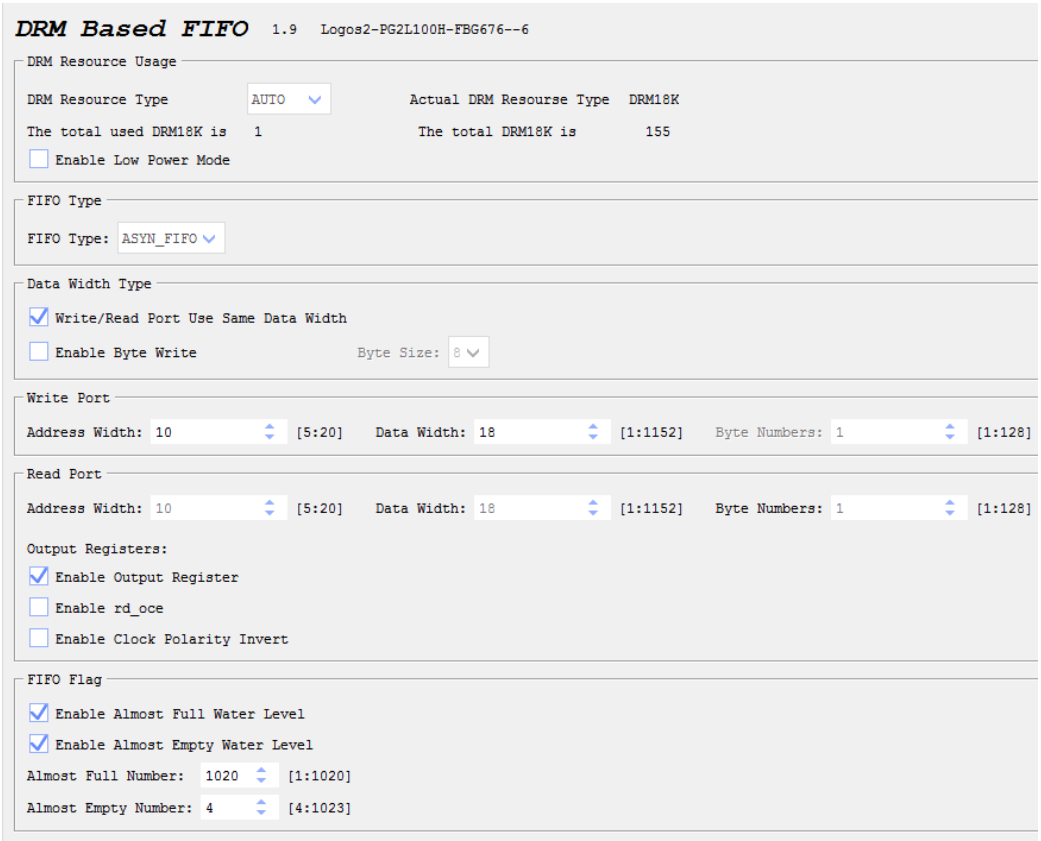


图 6.2-15

注意，如果勾选 Enable Output Register(输出寄存)，输出数据会延迟一个时钟周期。

FIFO Type 有 SYNC 和 ASYNC 两种，第一种是同步 FIFO，读写端口共用一个时钟和复位，另一种是异步 FIFO，读写时钟和复位均独立。在平常设计中，比较常用的是异步 FIFO，因为同步 FIFO 和异步 FIFO 的读写时序一模一样，只有读写端口的时钟复位有差异，当异步 FIFO 的读写端口使用相同的时钟和复位，此时异步 FIFO 和同步 FIFO 基本是一致的。

Reset Type 也可以选择 SYNC 和 ASYNC 两种，SYNC 模式下需要时钟的上升沿采样到复位有效才会复位，而在 ASYNC 模式下，复位一旦有，FIFO 立即复位。

其余端口说明引用官方 IP 手册，如图 6.2-16 所示：

端口名	输入/输出	说明
wr_data	输入	写数据信号，位宽范围1~1152
wr_en	输入	写使能信号，高有效
wr_byte_en	输入	Byte Write使能信号，当配置“Enable Byte Write”选项勾选时有效，位宽范围1~128。 1：对应Byte值有效； 0：对应Byte值无效
clk	输入	同步FIFO时钟信号，仅同步FIFO有效
rst	输入	同步FIFO复位信号，高有效，仅同步FIFO有效
wr_clk	输入	异步FIFO写时钟信号，仅异步FIFO有效
wr_rst	输入	异步FIFO写复位信号，高有效，仅异步FIFO有效
wr_full	输入	FIFO Full信号 1：FIFO满 0：FIFO未满
almost_full	输出	FIFO Almost Full信号 1：FIFO将满 0：FIFO未将满
wr_water_level	输出	写端口water level信号，位宽范围5~20，表示写数据水位
rd_data	输出	读数据信号
rd_en	输出	读使能信号
rd_clk	输入	异步FIFO读时钟信号，仅异步FIFO有效
rd_rst	输入	异步FIFO读复位信号，仅异步FIFO有效
rd_empty	输入	FIFO Empty信号 1：FIFO空 0：FIFO未空
almost_empty	输出	FIFO Almost Empty信号 1：FIFO将空 0：FIFO未将空
rd_water_level	输出	读端口water level信号，位宽范围5~20，表示读数据水位
rd_oce	输入	输出寄存使能信号 1：对应地址有效，读数据寄存输出 0：对应地址无效，读数据保持

图 6.2-16

其中 rd_water_level 和 wr_water_level 分别代表”可读的数据量”和”已写入的数据量”，其含义与 Xilinx 的 FIFO 的 wr_data_count 和 rd_data_count 是一致的。

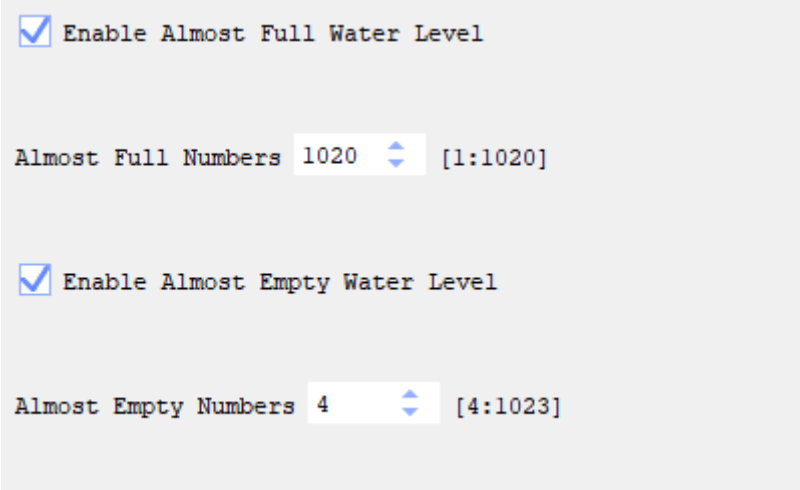


图 6.2-17

当我们将 Enable Almost Full Water Level 和 Enable Almost Empty Water Level 勾选上，才能看到 rd_water_level 和 wr_water_level，而下面的 Almost Full Numbers 的设置是表示当写入 1020 个数据时，Almost Full 信号就会拉高，Almost Empty Numbers 的设置表示当可读数据剩下 4 个时 Almost Empty 信号就会拉高。

同时 FIFO 支持混合位宽，例如写端口 16bit，读端口 8bit。如果写入 16'h0102，那么读出来会是 8'h02,8'h01，会先读出低位。

如果写端口 8bit，读端口 16bit。当写入 8'h01,8'h02 时，读出来是 16'h0201，先写入的数据存放在低位。

6.2.2.1. 的读写时序

因为同步 FIFO 和异步 FIFO 的读写时序一致，这里用异步 FIFO 的读写时序图来做介绍。

注意：复位时高电平有效。读出数据均未勾选 Enable Output Register(输出寄存)。

6.2.2.1.1. FIFO 未满时的写时序

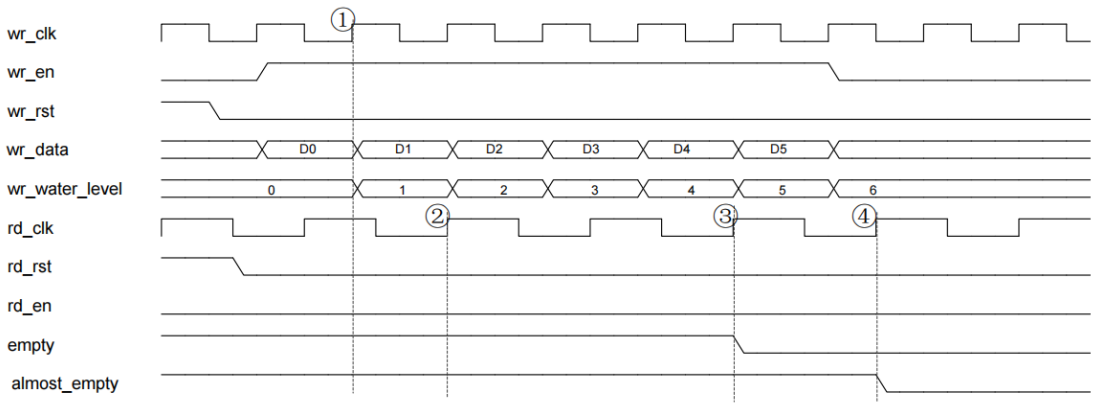


图 6.2-18

可以看到在 1 时刻，复位信号时低电平，处于工作状态，此时在 wr_clk 的上升沿且 wr_en 为高电平时将数据 D0 写入 FIFO，wr_water_level 也从 0 变 1，表示已经写入了一个数据，此时注意看读端口的 empty 信号，在 3 时刻 empty 信号从高变低，意味着读端口已经有数据可以读了，FIFO 不再为空，而注意看，rd_clk 和 wr_clk 是不一样的，从 1 写入到 3 时刻 empty 拉低时，经过了 3 个 rd_clk。

所以这里我们可以得出结论:rd_water_level 要滞后 wr_water_level 三个 rd_clk。

6.2.2.1.2. FIFO 将满时的写时序

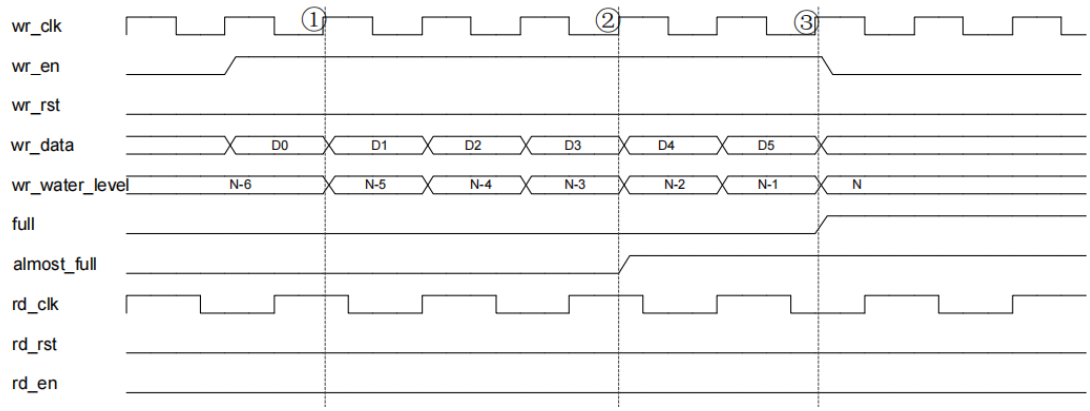


图 6.2-19

将满时主要分析 full 和 almost_full 信号。假设 Almost Full Numbers 设置为 N-2，在 1 时刻，此时已经写入了 N-6 个数据，意味着再写 6 个数据 FIFO 就满了，从 1 时刻到 2 时刻一共写入了 4 个数据，因此当 wr_water_level 变成 N-2 时，满足条件，可以看到 Almost Full 信号拉高，再写两个数据 FIFO 就满了，所以再经过两个时钟周期后，Full 信号拉高。

6.2.2.1.3. FIFO 在满状态下的读时序

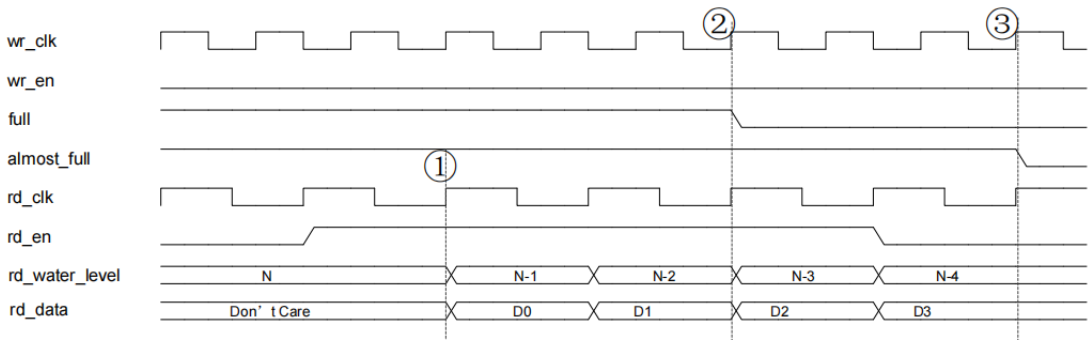


图 6.2-20

在满状态下，FIFO 已经有 N 个数据了，此时在 1 状态下，rd_clk 的上升沿，且 rd_en 为高电平时，此时从 FIFO 里读出数据(数据的输出有延时，仿真中延时 0.2ns)。此

时 rd_water_level 变成 N-1，rd_data 输出 D0。然后看 2 时刻，full 信号拉低，此时可以看以下，在 1 时刻到 2 时刻期间一共经过了 3 个 wr_clk 写端口才能判断到此时数据量已经不为满。 所以我们可以得出结论，wr_water_level 要滞后 rd_water_level 三个 wr_clk。

6.2.2.1.4. FIFO 将空时的读时序

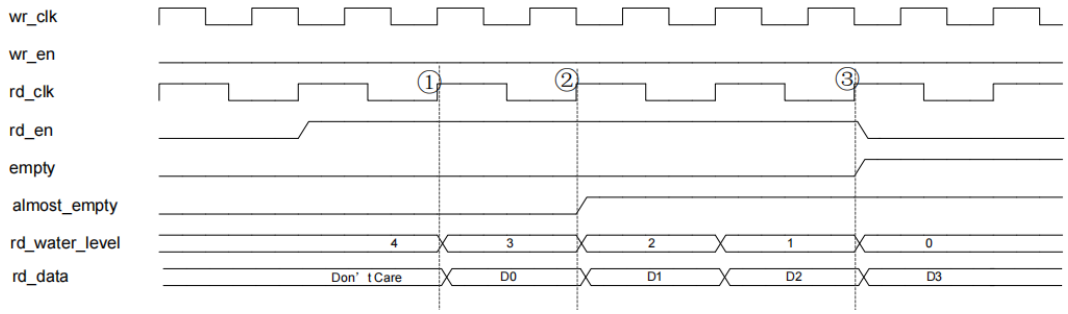


图 6.2-21

在 1 时刻，可读的数据量剩下 4，假设 Almost Empty Number 设为 2，在 1 时刻和 2 时刻分别读出了两个数据，所以在 2 时刻下，可读数据量剩下两个，达到 Almost Empty Number 触发条件，因此 almost_empty 信号拉高，再过两个时钟周期，即再读两个数据，FIFO 将变成空状态，也就是状态 3，此时 empty 信号拉高。

6.3. 接口列表

该部分介绍每个顶层模块的接口。

ram_test_top.v

端口	I/O	位宽	描述
wr_clk	input	1	写时钟
rd_clk	input	1	读时钟
rst_n	input	1	全局复位
rw_en	input	1	1:写操作 0:读操作
wr_addr	input	5	写地址
rd_addr	input	5	读地址
Wr_data	input	8	写入 RAM 的数据
Rd_data	output	8	从 RAM 读出的数据

rom_test_top.v

端口	I/O	位宽	描述
rd_clk	input	1	读时钟
rst_n	input	1	全局复位
rd_addr	input	10	读地址
rd_data	input	64	从 ROM 读出的数据

fifo_test_top.v

端口	I/O	位宽	描述
sys_clk	input	1	写/读时钟
rst_n	input	1	全局复位
wr_data	input	8	写入 FIFO 的数据
wr_en	input	1	写使能
rd_en	input	1	读使能
wr_water_level	output	8	已写入 FIFO 的数据量
rd_water_level	output	8	可从 FIFO 读出的数据量
Rd_data	output	8	从 FIFO 读出的数据

6.4. 工程说明

暂无

6.5. 代码仿真说明

本次的顶层模块实际就是例化 IP，然后把端口引出而已，主要代码都在 testbench 里面，所以我们直接介绍仿真代码。

6.5.1. RAM 仿真测试

```
1.  `timescale 1ns/1ns
2.  module ram_test_tb();
3.  reg     sys_clk;
4.  reg     rd_clk ;
5.  reg     rst_n;
6.  reg     rw_en;     //读写使能信号
7.
8.  reg     [7:0]     wr_data;
```

```
reg     [4:0]     wr_addr;
reg     [4:0]     rd_addr;

wire    [7:0]     rd_data;

reg     [1:0]     state;

initial
begin
    rst_n     <=     1'd0;
    sys_clk   <=     1'd0;
    rd_clk    <=     1'd0;
    #20
    rst_n     <=     1'd1;

end

//读写控制
always@(posedge sys_clk or negedge rst_n)     begin
    if(!rst_n)
    begin
        state     <=     2'd0;
        wr_data   <=     8'd0;
        rw_en     <=     1'd0;
        wr_addr   <=     8'd0;
        rd_addr   <=     8'd0;
    end
    else
    begin
    case(state)
        2'd0:begin
                rw_en     <=     1'd1;
                state     <=     2'd1;
            end

        2'd1:begin
                if(wr_addr == 5'd31)
                begin
                    rw_en     <=     1'd0;
                    state     <=     2'd2;
                    wr_data   <=     8'd0;
                    wr_addr   <=     5'd0;
                    rd_addr   <=     5'd0;
                end
                else
                begin
                    state     <=     2'd1;
                    wr_data   <=     wr_data+1'b1;
                    rd_addr   <=     rd_addr+1'b1;
                    wr_addr   <=     wr_addr+1'b1;
                end
            end
        2'd2:begin
                if(rd_addr == 5'd31)
                begin
                    state     <=     2'd3;
                    rd_addr   <=     5'd0;
                end
```

```
67.                     else
68.                     begin
69.                         state     <=     2'd2;
70.                         rd_addr   <=     rd_addr+1'b1;
71.                     end
72.                 end
73.             2'd3:begin
74.                     state     <=         2'd0;
75.                 end
76.
77.             default:     state     <=     2'd0;
78.             endcase
79.         end
80.     end
81.
82.     //50MHZ
83.     always#10 sys_clk = ~sys_clk;
84.
85.     //
86.     GTP_GRS GRS_INST(
87.         .GRS_N(1'b1)
88.         ) ;
89.
90.     ram_test_top u_ram_test_top(
91.         .wr_clk     ( sys_clk     ),
92.         .rd_clk     ( sys_clk     ),
93.         .rst_n      ( rst_n       ),
94.         .rw_en      ( rw_en       ),
95.         .wr_addr    ( wr_addr     ),
96.         .rd_addr    ( rd_addr     ),
97.         .wr_data    ( wr_data     ),
98.         .rd_data    ( rd_data     )
99.     );
100.    endmodule
```

涉及到 testbench 的一些基础操作这里就不再详细讲解，只关注重点逻辑部分。从代码的 **27 行到 80 行** 是 RAM 的读写控制状态机，主要用来控制读写地址的生成、读写使能信号以及写入数据的过程。

首先看 **38-42 行**，也就是 **state=0** 的时候，这里把 rw_en 拉高，并跳转到状态 1，表示进入写操作阶段（这里没有时钟使能 clk_en，可以不用管）。注意这是时序逻辑，信号的赋值要等到下一个时钟周期才生效，所以虽然在 state=0 时已经拉高了 rw_en，但真正的数据写入是从下一个时钟周期才开始的。

接下来是 **state=1** 的逻辑，对应 **44-60 行**。在这个状态下，仿真一直在往 RAM 里面写数据。可以看到，当 wr_addr 不等于 31 时，每个周期 wr_data 和 wr_addr 都会不断加 1（rd_addr 也跟着加 1，这是为了配合验证伪双口 RAM 的时序）。当 wr_addr 等于 31 的时候，会在下一个时钟周期把数据清零、地址清零并切换到 state=2。但要注意，在当前时钟周期下还是会再往地址 31 写一次数据。这样整个写过程一共写入了 32 个数据，从地址 0 写到地址 31。

然后进入 **state=2**，对应 **61-72 行**，此时开始读数据。在这个状态下，每个周期上

升沿 rd_addr 不断累加,直到等于 31 的时候,在下一个时钟周期才会清零并切换状态,而当前时钟周期还会继续把地址 31 的数据读出来。这样就保证了完整地读出地址 0~31 的 32 个数据。

最后是 **state=3**，对应 **73-75 行**。这里可以看到只是等待一个时钟周期，然后再跳转回 state=0，起到一个延时的作用，相当于让读写过程周而复始地循环执行。

整体来说，这个状态机的功能就是：**先写满 32 个数据到 RAM，再依次读出 32 个数据，然后等待一个周期后重新开始**。如果是初学者，可能会对“在 rd_addr=31 时还会再读一个数据”感到困惑，这其实就是时序逻辑的特性 —— 在时钟上升沿触发的赋值，要等到下一个时钟周期才会生效，所以当 rd_addr=31 时，当前周期的读地址还是有效的，因此会再读出最后一个数据。

一句话总结：**时序逻辑的赋值总在下一个时钟周期才生效，所以写操作和读操作在边界条件时（wr_addr=31、rd_addr=31），依然会在当下周期完成最后一个数据的写入或读出。**

6.5.2. ROM 仿真测试

```
1.  `timescale 1ns/1ns
2.  module rom_test_tb();
3.  reg     sys_clk;
4.  reg     rst_n;
5.  reg     [9:0]     rd_addr;
6.  wire    [63:0]     rd_data;
7.
8.  initial
9.  begin
10.     rst_n     <=     1'd0;
11.     sys_clk   <=     1'd0;
12.     #20
13.     rst_n     <=     1'd1;
14.
15. end
16.
17. //50MHZ
18. always#10 sys_clk = ~sys_clk;
19. //
20. GTP_GRS GRS_INST(
21.     .GRS_N(1'b1)
22.     ) ;
23.
24. always@(posedge sys_clk or negedge rst_n)     begin
25.     if(!rst_n)
26.         rd_addr     <=     10'd0;
27.     else
28.         rd_addr     <=     #2 rd_addr + 1'b1;
29. end
30.
31. rom_test_top u_rom_test_top(
32.     .rd_clk     ( sys_clk   ),
33.     .rst_n      ( rst_n     ),
```

```
34.      .rd_addr  ( rd_addr  ),
35.      .rd_data  ( rd_data  )
36.  );
37.
38.  endmodule
```

首先在 8-15 行，是初始化部分。rst_n 在最开始被拉低，保持 20ns 后释放为高电平，同时 sys_clk 初始化为 0。这样在 20ns 之后，整个电路进入正常工作状态。

接下来 17-18 行，通过 always #10 sys_clk = ~sys_clk; 来生成一个周期 20ns 的时钟信号，即 50MHz 的系统时钟，作为 ROM 的读时钟。

核心逻辑在 24-29 行。在时钟上升沿时，如果复位有效，则 rd_addr 被清零；如果复位释放，则 rd_addr 会在每个时钟周期递增 1。这里在 rd_addr <= #2 rd_addr + 1'b1; 中引入了 #2 的延时，表示在时钟上升沿过后延迟 2ns 再更新地址。这通常用于模拟真实硬件中信号到达的延迟，也可以帮助仿真时观察地址和数据之间的关系。

这样设计后，整个 testbench 的运行过程就是：在复位释放后，ROM 的读地址会从 0 开始，每个时钟周期加 1，不断扫描整个地址空间，同时通过端口把 ROM 中的数据读出到 rd_data。

整体功能 testbench 的主要作用就是 **连续递增读 ROM 的内容**，方便验证 ROM 初始化的数据是否正确，以及 ROM 在仿真中的读延迟和数据稳定性。 **testbench 的逻辑就是通过时钟驱动，让 rd_addr 递增，从而顺序读出 ROM 中的全部数据，借助仿真波形检查 ROM 的读出结果是否符合预期**

6.5.3. FIFO 仿真测试

```
1.  `timescale 1ns/1ns
2.  module fifo_test_tb();
3.
4.  reg sys_clk;
5.  reg rst_n;
6.
7.  reg          [7:0]   wr_data;
8.  reg                  wr_en;
9.  reg                  rd_en;
10.
11. reg                  rd_state;    //读状态
12. reg                  wr_state;
13.
14. wire         [7:0]   rd_data;
15. reg          [7:0]   rd_cnt;
16.
17. wire         [7:0]   rd_water_level;
18. wire         [7:0]   wr_water_level;
19.
20. initial
21. begin
22.     rst_n    <=   1'd0;
23.     sys_clk <=   1'd0;
24.     #20
25.     rst_n    <=   1'd1;
```

```
26.
27.
28.  end
29.
30.  always#10 sys_clk = ~sys_clk;     //50MHZ
31.
32.  always@(posedge sys_clk or negedge rst_n)    begin
33.      if(!rst_n)
34.      begin
35.          wr_state     <=     1'd0;
36.          wr_en    <=    1'd0;
37.          wr_data <=    8'd0;
38.      end
39.      else
40.      begin
41.          case(wr_state)
42.              1'd0:     if(wr_water_level == 127)     //128 个数据
43.                        begin
44.                            wr_en     <=     #2 1'd0;
45.                            wr_data   <=     #2 8'd0;
46.                            wr_state <=     #2 1'd1;
47.                        end
48.                        else
49.                        begin
50.                            wr_en     <=     #2 1'd1;
51.                            wr_data   <=     #2 wr_data+1'b1;
52.                            wr_state <=     #2 1'd0;
53.                        end
54.
55.              1'd1:     if(rd_cnt == 127)
56.                            wr_state     <=     #2 1'd0;
57.
58.
59.              default:     wr_state     <=1'd0;
60.          endcase
61.      end
62.  end
63.
64.  always@(posedge sys_clk or negedge rst_n)    begin
65.      if(!rst_n)
66.      begin
67.          rd_state<=    1'd0;
68.          rd_en    <=    1'd0;
69.          rd_cnt   <=    8'd0;
70.      end
71.      else
72.      begin
73.          case(rd_state)
74.              1'd0:    if(rd_water_level >= 8'd128)     //等待 128 个数据
75.                       begin
76.                           rd_state     <=     #2 1'd1;
77.                           rd_en         <=     #2 1'd1;
78.                       end
79.                       else
80.                       begin
81.                           rd_cnt        <=   #2 8'd0;
82.                           rd_state     <=     #2 1'd0;
83.                       end
```

```
84.
85.                 1'd1:    begin
86.
87.                             rd_cnt  <=     #2 rd_cnt + 1'b1;
88.                             if(rd_cnt == 127)
89.                             begin
90.                                 rd_en         <=    #2 1'd0;
91.                                 rd_state      <=    #2 1'd0;
92.                             end
93.                         end
94.                 default:     rd_state     <=    1'd0;
95.             endcase
96.         end
97.  end
98.
99.  GTP_GRS GRS_INST(
100.     .GRS_N(1'b1)
101.     ) ;
102.
103. fifo_test_top u_fifo_test_top(
104.     .sys_clk            ( sys_clk            ),
105.     .rst_n              ( rst_n              ),
106.     .wr_data            ( wr_data            ),
107.     .wr_en              ( wr_en              ),
108.     .rd_en              ( rd_en              ),
109.     .wr_water_level     ( wr_water_level     ),
110.     .rd_water_level     ( rd_water_level     ),
111.     .rd_data            ( rd_data            )
112. );
113. endmodule
```

涉及到 testbench 的一些基础操作这里就不再重复，我们关注 FIFO 的写入与读取逻辑部分。从代码的 **32 行到 62 行** 和 **64 行到 97 行**，分别对应 FIFO 的写入控制状态机和读取控制状态机。

首先在 **初始化部分（20-28 行）**，rst_n 在 20ns 时释放，同时生成一个 50MHz 的系统时钟（30 行）。这样在仿真启动后，FIFO 可以在复位完成后开始正常工作。

接下来 **写入逻辑（32-62 行）**。在复位时，wr_state、wr_en 和 wr_data 都清零。复位释放后，进入写状态机：

在 **state=0** 时，testbench 会不断往 FIFO 中写入数据，每次写入时 wr_en 拉高，并且 wr_data 自增。当写入数据达到 128 个（wr_water_level == 127）时，拉低 wr_en，清零 wr_data，并将写状态机切换到 state=1。在 **state=1** 时，写状态机会等待读取计数 rd_cnt 到达 127，才会重新跳转回 state=0，继续写数据。这样保证 FIFO 不会无限写入，而是写满 128 个数据后进入等待状态，等待读出操作。

然后是 **读取逻辑（64-97 行）**。在复位时，rd_state、rd_en 和 rd_cnt 被清零。复位释放后，进入读状态机：

在 **state=0** 时，只有当 FIFO 中的读水位计数 rd_water_level >= 128 时，才会启动读取操作，将 rd_state 切换到 state=1，并拉高 rd_en。在 **state=1** 时，开始从 FIFO 中读取数据，每个周期 rd_cnt 自增，直到读满 128 个数据（rd_cnt == 127），再

拉低 rd_en 并回到 state=0。如果 FIFO 数据未达到 128，则继续保持等待，直到数据满足条件再开始读出。

这样，写和读状态机配合起来，构成了 **写满 128 个数据 → 读出 128 个数据 → 再次写入 → 再次读取** 的循环过程，保证 FIFO 的写入和读取可以交替进行。

整体功能实现上，这个 testbench 的作用就是验证 FIFO 的写入和读取机制是否正确，尤其是写水位和读水位信号的变化是否符合预期。通过波形可以直观地看到 FIFO 在写入 128 个数据后进入读过程，随后又重新开始写入，验证了 FIFO 的基本功能。

testbench 通过两个状态机分别控制写和读，每次写满 128 个数据再读出 128 个数据，形成一个完整的 FIFO 功能验证流程。

7. 基于紫光 FPGA 的 LED 流水灯

7.1. 实验简介

实验目的:

通过按键控制 8 个 LED 灯按顺序依次点亮和熄灭。PT2G390H 开发板有 8 个用户 LED 灯（LED1～8），FPGA 输出高电平时对应的 LED 灯亮灯（详情请查看“PT2G390H 开发板硬件使用手册”）。控制 8 个 LED 灯按顺序依次点亮和熄灭。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

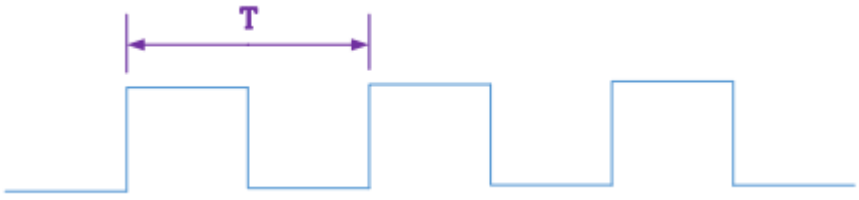
PT2G390H 开发板

7.2. 实验原理

通常的时，分，秒的计时进位大家应该不陌生;

1 小时=60 分钟=3600 秒，当时针转动 1 小时，秒针跳动 3600 次;

在数字电路中的时钟信号也是有固定的节奏的，这种节奏的开始到结束的时间,我们通常称之为周期（T）。



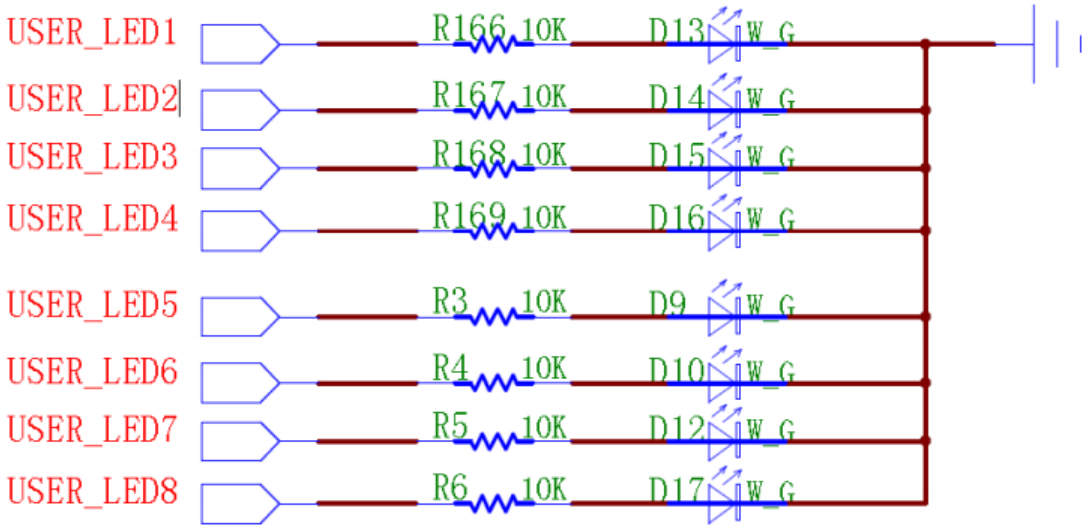
在数字系统中通常关注到时钟的频率,那频率与周期的关系如下:

$$f = \frac{1}{T};$$

PT2G390H 板卡上单端时钟有一个 50MHz 和一个 27MHz 的晶振提供时钟给到 PT2G390H;

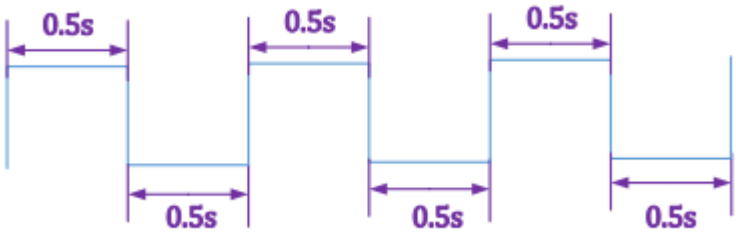
实验分析:

控制 LED 亮灭需要控制 IO 输出的高低电平即可(高电平点亮,低电平熄灭),原理图如下:



控制 LED 依次 0.5s 亮，0.5s 灭,需要控制 IO 依次输出 0.5s 高电平，0.5s 低电平周期变化。

如下图波形:



若使用 50MHz 外部输入时钟，时钟周期为 20ns（在 verilog 设计中的计数器的计时原理基本上是一致的，确认输入时钟周期和目标计时时间后可得到计数器的计数值到达多少后可得到计时宽度）；

$$0.5s = 25000000*20ns = 25000000 \times T_{50MHz};$$

IO 输出状态只有两种：1 或 0；我们可以使用一个计数器，计数满 25000000 个时钟周期时变化不同 LED 点亮。

7.3. 实验源码设计

7.3.1. 文件头设计

在 module 之前添加文件头，文件头中包含信息有：公司，作者，时间，设计名，工程名，模块名，目标器件，EDA 工具(版本)，模块描述，版本描述（修改描述）等信

息；以及仿真时间单位定义；

```
1.  `timescale 1ns / 1ps
2.  //////////////////////////////////////////////////////////////////////////////////
3.  // Company:Meyesemi
4.  // Engineer: Will
5.  //
6.  // Create Date: 2023-01-29 20:31
7.  // Design Name:
8.  // Module Name:
9.  // Project Name:
10. // Target Devices: Pango
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 1.0 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////////////////////////////////////////////////
21.
22. `define UD #1
```

`timescale1ns/1ps 表示仿真精度是 1ns，显示精度是 1ps；

`defineUD#1 定义 UD 表示#1；#1 仅仿真有效，表示延时一个仿真精度，结合上一条语句表示延时 1ns；

7.3.2. 设计 module

```
1.  module led_test(
2.  input           clk,
3.  input           rstn,
4.
5.  output [7:0]    led
6.  );
```

此段代码是标准的 module 创建的模型，module 创建时需要确认输入输出信号并定义好位宽，之后在对 module 进行具体的逻辑设计；管脚与管脚之间用“,”隔开，最后一个管脚不用间隔符号；

创建 module 时需要定义输入输出信号;本实验输入时钟和复位即可，输出是控制 LED 的亮灭，PT2G390H 板卡上共有 8 个 LED，故而输出 8bit 位宽的信号；

单个状态计数 25_000_000，即 24_999_999=25'b1_0111_1101_0111_1000_0011_1111;所以计数器的位宽为 25 位即可；

```
1.  //time counter
2.      always @(posedge clk)
3.      begin
4.          if(!rstn)
5.              led_light_cnt <= `UD 26'd0;
6.          else if(led_light_cnt == 26'd24_999_999)
7.              led_light_cnt <= `UD 26'd0;
8.          else
9.              led_light_cnt <= `UD led_light_cnt + 26'd1;
```

```
10.         end
11.
```

当计数器计数到 25'd24_999_999 时，计数过程包含了从 0～26'd2499_9999 的时钟周期，故而总时长时 25’d25_000_000×Tclk；硬件输入时钟为 50MHz，所以此计数器的计数周期是 0.5s；

在指定的时间刻度上对 LED 的状态进行变更，以达到控制 LED 依次亮灭的目的；led_light_cnt 的计时周期为 0.5s，故在 led_light_cnt 上取一个点来变更 LED 的显示状态即可完成每隔 0.5s 让 LED 显示发生变化；由于 LED 亮和灭只有两个状态，在赋值处理上将寄存器进行移位操作；

```
1.  //led status change
2.      always @(posedge clk)
3.      begin
4.          if(!rstn)
5.              led_status <= `UD 8'b0000_0001;
6.          else if(led_light_cnt == 25'd24_999_999)
7.              led_status <= `UD {led_status[6:0],led_status[7]};
8.      end
9.
10.     assign led = led_status;
```

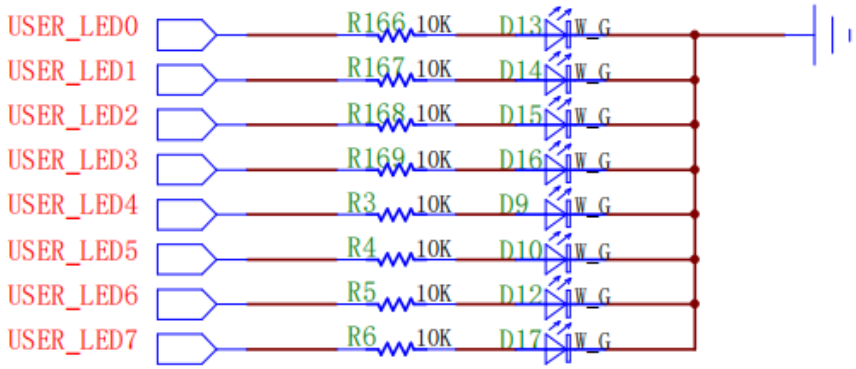
7.3.3. 完整的 Module

```
1.  module led_test(
2.      input           clk,
3.      input           rstn,
4.
5.      output [7:0]    led
6.  );
7.
8.  //============================================================================
9.  //reg and wire
10.
11.     reg [25:0] led_light_cnt    = 26'd0         ;
12.     reg [ 7:0] led_status       = 8'b0000_0001  ;
13.
14. //time counter
15.     always @(posedge clk)
16.     begin
17.         if(!rstn)
18.             led_light_cnt <= `UD 26'd0;
19.         else if(led_light_cnt == 26'd24_999_999)
20.             led_light_cnt <= `UD 26'd0;
21.         else
22.             led_light_cnt <= `UD led_light_cnt + 26'd1;
23.     end
24.
25. //led status change
26.     always @(posedge clk)
27.     begin
28.         if(!rstn)
29.             led_status <= `UD 8'b0000_0001;
30.         else if(led_light_cnt == 25'd24_999_999)
31.             led_status <= `UD {led_status[6:0],led_status[7]};
```

```
32.     end
33.
34.     assign led = led_status;
35.
36. endmodule
```

7.3.4. 硬件管脚分配

PT2G390H 的 LED 和 CLK 与 FPGA 的 IO 连接部分的原理图如下，详情可查看硬件使用手册或原理图：



信号	PT2G390HPin
LED1	V30
LED2	V29
LED3	V20
LED4	V19
LED5	W24
LED6	W23
LED7	V24
LED8	U24

复位设计是低电平有效，PT2G390H 开发板提供了 8 个用户按键（K1～8），按键低电平有效，但按键按下时，IO 上的输入电压为低；当没有按下按键时，IO 上的输入电压为高电平；选择任一个用户按键作为复位输入即可。

7.4. 实验现象

8 颗 LED 灯按照设定的顺序和时间依次点亮和熄灭

8. 基于紫光 FPGA 的键控流水灯实验例程

8.1. 实验简介

实验目的:

PT2G390H 开发板有 8 个用户 LED 灯（LED1 ~ 8），FPGA 输出高电平时对应的 LED 灯亮灯（详情请查看“PT2G390H 开发板硬件使用手册”）。由 USER_BUTTON1 按键输入，切换 USER_LED1~USER_LED8 的输出效果。

实验环境:

Window11

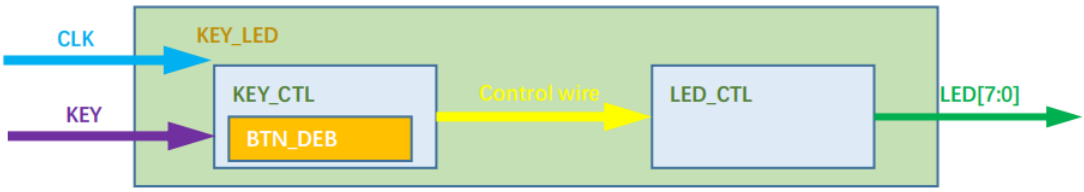
PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

8.2. 实验原理

实现框架如下:



（1）顶层实现按键切换 LED 的流水灯状态;

（2）需要设计一个输入控制模块及一个输出控制模块;

这个实验带大家将多个模块整合成为一个工程，涉及到的知识点有子模块设计、模块例化；子模块的设计主要是依据功能定位，确定输入输出，再做具体的设计;

模块例化方式如下:

```
module_name # (
.PARAM ( PARAM_SET )
// PARAM 为例化模块的常量接口；PARAM_SET 为常量赋值内容
) unint_name(
// module_name 为例化 module 名；unint_name 为例化后单元名称
.port ( signal )
// port 为例化模块中的管脚； signal 为当前模块的信号
);
```

8.2.1. 按键控制模块功能

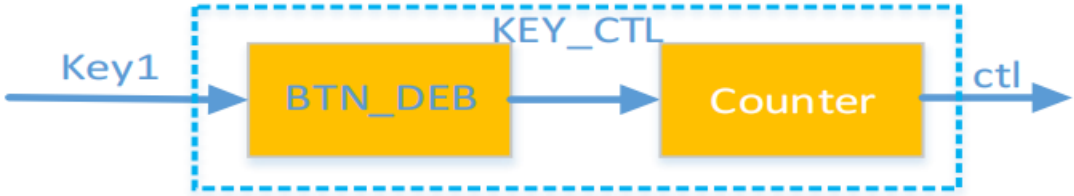
接收按键输入信号。统计按键按下次数，由于流水灯模式是 3 种，计数统计范围是 0 ~ 2 循环，将计数结果传递给 LED 控制模块;

根据需求输入信号有：时钟，按键；输出信号有：流水灯控制信号;

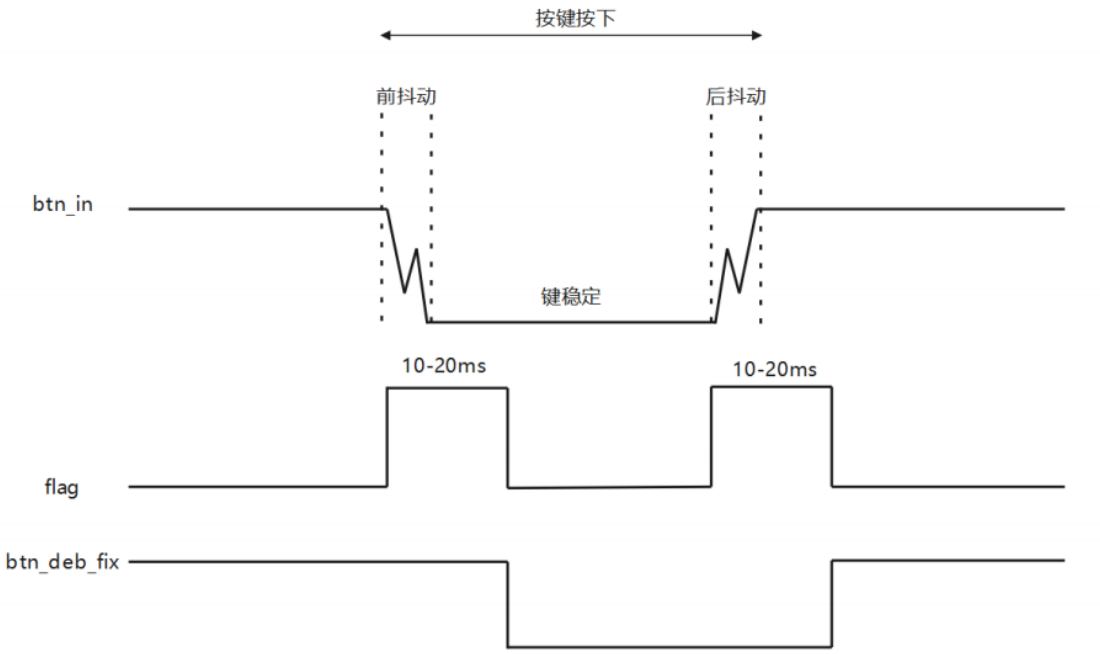
内部功能处理：

<1>内部需要对按键信号做消抖处理；

<2>按键触发计数器（计数值输出）改变继而调整流水灯的状态；



8.2.2. 按键消抖模块



前后抖动时间约为 5 ~ 10ms，取按键抖动区间开始标识，持续 10-20ms 后标识归零，在抖动区间内输出保持，非消抖区间，按键状态输出。

8.2.3. LED 控制模块功能

3 种流水灯模式有按键传递过来的计数控制切换，每一个 LED 的显示状态完整后进入下一模式初始化。根据需求可得到如下信息：

输入信号：时钟，流水灯模式控制信号；出信号：8bit 位宽的 LED 控制信号；功能处理注意事项：流水灯状态切换点，不同状态的切换时如何初始化；

8.3. 实验源码设计

8.3.1. 顶层文件源码

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module key_led_top(
4.      input               clk,       //50MHz
5.      input               key,
6.
7.      output [7:0]     led
8.  );
9.
10.    wire [1:0] ctrl;
11.
12.    key_ctl key_ctl(
13.        .clk          (  clk  ),     //input               clk,
14.        .key          (  key  ),     //input               key,
15.
16.        .ctrl         (  ctrl  )     //output       [1:0] ctrl
17.    );
18.
19.    led u_led(
20.        .clk    (  clk    ),         //input             clk,
21.        .ctrl   (  ctrl   ),         //input   [1:0]   ctrl,
22.
23.        .led    (  led    )         //output [7:0]   led
24.    );
25.
26. endmodule
27.
```

8.3.2. 按键控制模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module key_ctl(
4.      input               clk,
5.      input               key,
6.
7.      output       [1:0] ctrl
8.  );
9.
10.     wire btn_deb;
11.
12.     btn_deb_fix#(
13.         .BTN_WIDTH    (  4'd1          ), //parameter                   BTN_WIDTH = 4'd8
14.         .BTN_DELAY    (20'h7_ffff     )
15.     ) u_btn_deb
16.     (
17.         .clk          (  clk           ),//input                           clk,
18.         .btn_in       (  key           ),//input       [BTN_WIDTH-1:0] btn_in,
19.
20.         .btn_deb_fix (  btn_deb       ) //output reg [BTN_WIDTH-1:0] btn_deb
21.     );
22.
23.     reg btn_deb_1d;
```

```
24.     always @(posedge clk)
25.     begin
26.         btn_deb_1d <= `UD btn_deb;
27.     end
28.
29.     reg [1:0]   key_push_cnt=2'd0;
30.     always @(posedge clk)
31.     begin
32.         if(~btn_deb & btn_deb_1d)
33.         begin
34.             if(key_push_cnt == 2'd2)
35.                 key_push_cnt <= `UD 2'd0;
36.             else
37.                 key_push_cnt <= `UD key_push_cnt + 2'd1;
38.         end
39.     end
40.
41.     assign ctrl = key_push_cnt;
42.
43. endmodule
```

整个模块主要功能是对外部输入的按键信号进行消抖，并通过计数器实现 按键多次按下时的模式切换控制。

在 10-21 行，这里实例化了一个 btn_deb_fix 模块，对输入 key 进行消抖处理。由于机械按键在按下和释放时会产生抖动，所以必须经过消抖电路，输出稳定的 btn_deb 信号。参数设置了单通道按键（BTN_WIDTH=1）和延迟阈值（BTN_DELAY=20'h7_ffff），确保按键信号稳定后才有效。

在 23-27 行，通过一个时序寄存器 btn_deb_1d 保存消抖信号的一个时钟周期延迟。这样可以配合当前周期的 btn_deb，用来检测按键的边沿变化。

重点逻辑在 29-39 行：定义了一个 2 位寄存器 key_push_cnt，用来计数按键的按下次数。在 always 块中，通过条件 if(~btn_deb & btn_deb_1d) 检测到 **按键从按下到释放的上升沿**（即一次有效的按键动作）。每次检测到有效按键动作时，计数器 key_push_cnt +1。计数器达到 2'd2 时，再次按键会将其清零，形成一个 **0 → 1 → 2 → 0** 的循环计数。

最后在 41 行，将 key_push_cnt 的值输出到 ctrl。这样 ctrl 就会随着按键次数循环变化，通常可用来切换工作模式， **按一次进入模式 1，按两次进入模式 2，再按一次回到模式 0**。

总体说 btn_deb_fix 模块解决了机械按键抖动问题；边沿检测逻辑确保每次按键只触发一次计数；2 位计数器实现了多模式循环切换功能；输出 ctrl 作为最终控制信号，直接驱动其他模块。

8.3.3. 按键消抖模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module btn_deb_fix#(
```

```
4.      parameter           BTN_WIDTH = 4'd8,
5.      parameter           BTN_DELAY = 20'h7_ffff
6.  )
7.  (
8.      input                           clk,   //
9.      input       [BTN_WIDTH-1:0] btn_in,
10.
11.     output reg [BTN_WIDTH-1:0] btn_deb_fix
12. );
13.
14.     //16'h3ad43;
15.     reg [19:0]              cnt[BTN_WIDTH-1:0];
16.     reg [BTN_WIDTH-1:0] flag;
17.
18.     reg [BTN_WIDTH-1:0] btn_in_reg;
19.
20.     always @(posedge clk)
21.     begin
22.      btn_in_reg <= `UD btn_in;
23.     end
24.
25.     genvar i;
26.     generate
27.         for(i=0;i<BTN_WIDTH;i=i+1)
28.         begin
29.             always @(posedge clk)
30.             begin
31.              if (btn_in_reg[i] ^ btn_in[i]) //取按键边沿开始抖动区间标识
32.                flag[i] <= `UD 1'b1;
33.              else if (cnt[i]==BTN_DELAY)     //持续 10ms-20ms 后归零
34.                flag[i] <= `UD 1'b0;
35.                  else
36.                      flag[i] <= `UD flag[i];
37.             end
38.
39.             always @(posedge clk)
40.             begin
41.              if(cnt[i]==BTN_DELAY)         //计数 10ms-20ms 时归零
42.                cnt[i] <= `UD 20'd0;
43.              else if(flag[i])              //抖动区间有效时计数
44.                cnt[i] <= `UD cnt[i] + 1'b1;
45.              else                           //非抖动区间保持 0
46.                cnt[i] <= `UD 20'd0;
47.             end
48.
49.             always @(posedge clk)
50.             begin
51.              if(flag[i])                   //抖动区间，消抖输出保持
52.                btn_deb_fix[i] <= `UD btn_deb_fix[i];
53.              else                           //非抖动区间，按键状态传递到消抖输出
54.                btn_deb_fix[i] <= `UD btn_in[i];
55.             end
56.         end
57.     endgenerate
58.
59. endmodule
```

代码的 25 行到 57 行是按键消抖的核心实现，主要用于对多路机械按键信号进行消

抖，输出稳定的按键状态。这里只讲解主要实现的功能。首先代码的 20-23 行，通过一个时钟上升沿触发的 always 块，将输入按键状态寄存到 btn_in_reg，这是为了后续检测按键边沿变化，保证时序逻辑能够正确捕捉按键的跳变。接着在生成块中，对于每一路按键，都存在三个 always 块，分别用于抖动检测、计数器累加和消抖输出。

代码的 31-37 行是抖动检测逻辑。当上一周期的按键状态 btn_in_reg[i] 与当前按键状态 btn_in[i] 不一致时，说明检测到按键边沿变化，于是将抖动标志 flag[i] 拉高，表示当前按键进入抖动阶段；如果计数器 cnt[i] 已经达到设定的消抖时间 BTN_DELAY，则将 flag[i] 清零；否则保持原状态不变。这个逻辑保证了按键在刚触发或释放的瞬间进入抖动阶段，抖动阶段持续时间由计数器控制。

代码的 41-47 行是计数器逻辑。在抖动阶段，计数器 cnt[i] 每个时钟周期累加，用于记录抖动持续时间，当计数器达到设定的消抖时间时自动清零。如果按键不在抖动阶段，计数器保持 0。这一逻辑的作用是为抖动检测提供时间窗口，从而滤除机械按键可能存在的瞬时跳变。

代码的 51-55 行是消抖输出逻辑。当按键处于抖动阶段时，消抖输出 btn_deb_fix[i] 保持原状态不变，避免在抖动过程中出现不稳定输出；而在非抖动阶段，按键的实际状态直接传递到消抖输出，从而生成稳定的按键信号。这里需要注意，由于是时序逻辑，赋值总在下一个时钟周期才生效，所以在抖动开始或结束的边沿，输出信号会在下一个时钟周期才更新。时序逻辑的赋值总在下一个时钟周期生效，所以抖动开始或结束时的操作会在当前时钟周期观察到的输出上有延迟，但不会影响整体的消抖效果。

8.3.4. LED 控制模块

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module led(
4.      input           clk,//50MHz
5.      input   [1:0]   ctrl,
6.
7.      output [7:0]   led
8.  );
9.
10.     reg [24:0] led_light_cnt = 25'd0;
11.     reg [ 7:0] led_status = 8'b1000_0000;
12.
13.     //   time counter
14.     always @(posedge clk)
15.     begin
16.         if(led_light_cnt == 25'd24_999_999)
17.             led_light_cnt <= `UD 25'd0;
18.         else
19.             led_light_cnt <= `UD led_light_cnt + 25'd1;
20.     end
21.
22.     reg [1:0] ctrl_1d=0;     //保存上一个 led 状态周期的 ctrl 值
23.     always @(posedge clk)
```

```
24.     begin
25.         if(led_light_cnt == 25'd0)
26.             ctrl_1d <= ctrl;
27.     end
28.
29.     // led status change
30.     always @(posedge clk)
31.     begin
32.         if(led_light_cnt == 25'd24_999_999)//0.5s 周期
33.         begin
34.             case(ctrl)
35.                 2'd0 :  //从高位到低位的 led 流水灯
36.                 begin
37.                     if(ctrl_1d != ctrl)
38.                         led_status <= `UD 8'b1000_0000;
39.                     else
40.                         led_status <= `UD {led_status[0],led_status[7:1]};
41.                 end
42.                 2'd1 :  //隔一亮一交替
43.                 begin
44.                     if(ctrl_1d != ctrl)
45.                         led_status <= `UD 8'b1010_1010;
46.                     else
47.                         led_status <= `UD ~led_status;
48.                 end
49.                 2'd2 :  //从高位到低位暗灯流水
50.                 begin
51.                     if(ctrl_1d != ctrl )
52.                         led_status <= `UD 8'b0111_1111;
53.                     else
54.                         led_status <= `UD {led_status[0],led_status[7:1]};
55.                 end
56.             endcase
57.         end
58.     end
59.
60.     assign led = led_status;
61.
62. endmodule
```

从代码的 14 行到 58 行是 LED 的闪烁控制逻辑，主要用于根据输入的控制信号 ctrl 实现不同模式的 LED 灯显示。这里只讲解主要实现的功能。首先代码的 14-20 行是时间计数器逻辑，通过一个 50MHz 时钟对 led_light_cnt 计数，每个时钟上升沿计数器加 1，当计数器达到 24,999,999 时归零，这样形成了一个 0.5 秒的周期，用作 LED 模式切换的时间基准。

代码的 23-27 行用于保存上一个 LED 状态周期的控制信号值 ctrl_1d，在每个计数器归零的时刻，将当前的 ctrl 保存到 ctrl_1d 中，以便在模式切换时判断当前周期是否与上一个周期不同，确保在切换模式时能够正确初始化 LED 状态。

代码的 30-58 行是 LED 状态变化逻辑。在每个计数器达到 24,999,999，即 0.5 秒时，进入一个 case 分支，根据 ctrl 信号选择不同的 LED 显示模式。当 ctrl=2'd0 时，实现从高位到低位的 LED 流水灯，如果模式刚刚切换（ctrl_1d != ctrl），则将 LED

状态初始化为 8'b1000_0000，否则每个周期将 LED 状态循环右移一位，形成流水灯效果。当 ctrl=2'd1 时，实现隔一亮一交替的 LED 闪烁，如果模式切换则初始化为 8'b1010_1010，否则每个周期取反 LED 状态，实现交替闪烁效果。当 ctrl=2'd2 时，实现从高位到低位的暗灯流水效果，切换模式时初始化为 8'b0111_1111，否则同样通过右移循环实现流水效果。

assign 语句 assign led = led_status; 将计算得到的 LED 状态输出到模块端口 led，使外部能够看到 LED 灯的显示效果。不同模式下 LED 状态会按 0.5 秒周期循环变化。这个模块通过计数器实现时间基准，通过保存上一个控制周期的信号判断模式切换，并根据不同模式更新 LED 状态，实现从流水灯、交替闪烁到暗灯流水的多种 LED 显示效果

8.4. 实验现象

每按下一次 KEY1，LED 灯状态切换一次，总共三种 LED 模式供循环切换；
LED 模式一：从高位到低位的 LED 流水灯；LED 模式二：隔一亮一交替点亮；
LED 模式三：从高位到低位暗灯流水；

9. 基于紫光 FPGA 的 UART 串口通信

9.1. 实验简介

实验目的:

PT2G390H 开发板集成了一路 USB 转串口模块，采用的 USB-UART 芯片 CP2102,USB 接口采用 USBTypeC 接口，可以用一根 USBTypeC 线连接到 PC 的 USB 口进行串口数据通信（详情请查看“PT2G390H 开发板硬件使用手册”）。通过本实验实现 FPGA 与 PC 之间的串口收发实验。串口通信时波特率设置为 115200bps，数据格式为 1 位起始位、8 位数据位、无校验位、1 位结束位。板子 1s 向串口助手发送一次十进制显示的“www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字（00~FF），LED 以二进制显示亮起。

实验环境:

Window11

PDS2023.2-SP3-ads

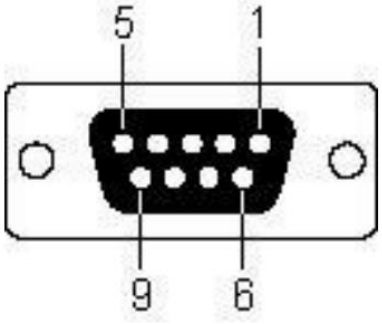
硬件环境:

PT2G390H 开发板

9.2. 实验原理

9.2.1. 串口原理

从图 8.2-1 我们可以看到标准串口接口是 9 根线，具体含义如下:



数据线:

TXD（pin 3）: 串口数据输出(Transmit Data)

RXD（pin 2）: 串口数据输入(Receive Data)

握手:

RTS（pin 7）: 发送数据请求(Request to Send)

CTS（pin 8）: 清除发送(Clear to Send)

DSR（pin 6）: 数据发送就绪(Data Send Ready)

DCD（pin 1）：数据载波检测(Data Carrier Detect)

DTR（pin 4）：数据终端就绪(Data Terminal Ready)

地线：

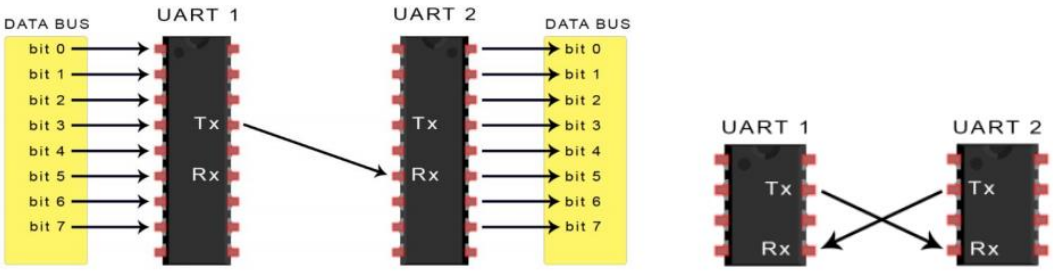
GND（pin 5）：地线

其它：

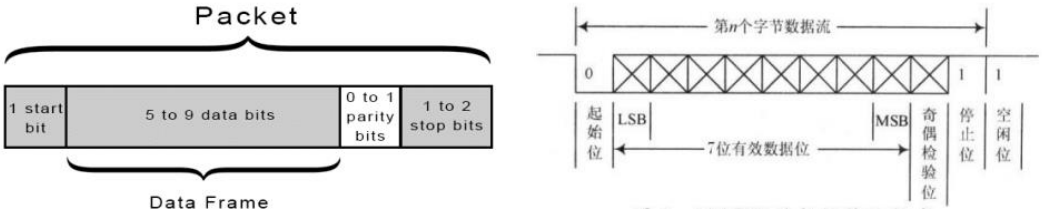
RI（pin 9）：铃声指示

通常我们用 RS232 串口仅用到了 9 根传输线中的三根：TXD，RXD，GND。但是对于数据传 输，双方必须对数据传输采用使用相同的波特率，约定同样的传输模式（传输架构，握手条件 等）。尽管这种方法对于大多数应用已经足够，但是对于接收方过载的情况这种使用受到限制。

RS232 的串口连接方式：



串口传输协议如下：



起始位：先发出一个逻辑”0”信号，表示传输字符的开始。

数据位：可以是 5~8 位逻辑”0”或”1”。如 ASCII 码（7 位），扩展 BCD 码（8 位）。LSB 表示低位，MSB 表示高位，有效数据的传输顺序为低位在前高位在后。

校验位：数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)。

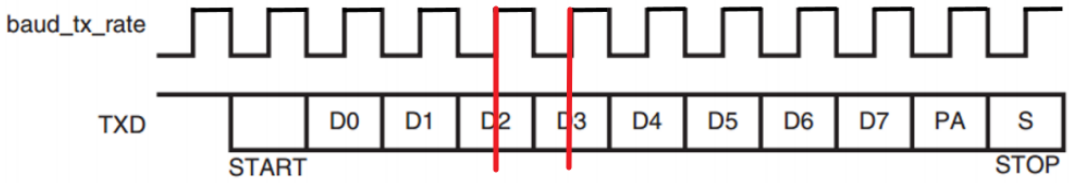
停止位：它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。

空闲位：处于逻辑“1”状态，表示当前线路上没有资料传送。

波特率：uart 中的波特率就可以认为是比特率，即每秒传输的位数(bit)。一般选波特率

都会有 9600,19200,115200 等选项。其实意思就是每秒传输这么多个比特位数(bit)。

引入波特率的概念后可得到串口的传输节奏如下：



细心的话可以发现数据的传输都是在数据稳定后的中心时刻，接收数据其实也是，都是在中心时刻采集数据，此时的数据是最稳定的。

9.2.2. 串口发送字符

从前面串口协议中可以了解到串口每次传输可以以有 5～8bit 数据，在计算机中字符通常

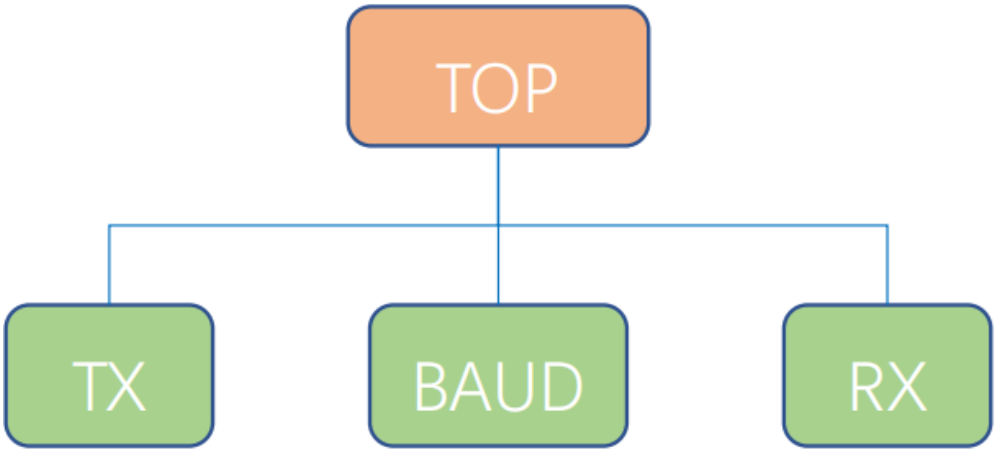
用 ASCII 码（7bit）表示，所以字符的发送可以用 ASCII 码发送。

查询 ASCII 码表格可得到：“www.meyesemi.com”用到的字符对应 ASCII 码；

```
8'h1  : write_data <= `UD 8'h77;// ASCII code is w
8'h2  : write_data <= `UD 8'h77;// ASCII code is w
8'h3  : write_data <= `UD 8'h77;// ASCII code is w
8'h4  : write_data <= `UD 8'h2E;// ASCII code is .
8'h5  : write_data <= `UD 8'h6D;// ASCII code is m
8'h6  : write_data <= `UD 8'h65;// ASCII code is e
8'h7  : write_data <= `UD 8'h79;// ASCII code is y
8'h8  : write_data <= `UD 8'h65;// ASCII code is e
8'h9  : write_data <= `UD 8'h73;// ASCII code is s
8'ha  : write_data <= `UD 8'h65;// ASCII code is e
8'hb  : write_data <= `UD 8'h6D;// ASCII code is m
8'hc  : write_data <= `UD 8'h69;// ASCII code is i
8'hd  : write_data <= `UD 8'h2E;// ASCII code is .
8'he  : write_data <= `UD 8'h63;// ASCII code is c
8'hf  : write_data <= `UD 8'h6F;// ASCII code is o
8'h10 : write_data <= `UD 8'h6D;// ASCII code is m
```

9.3. 实验源码设计

从实验目的分析可将实验做如下划分：



从原理上分析波特率的计算是一个计数器，发射和接收可复用，我们在设计时为保持 TX，或 RX 的完整性，故将波特周期计数器集成在各自模块内部；

上述分析仅仅搭建好 PT2G390H 的与 PC 通信的桥梁 UART，传输的数据没有体现。故而需要增加发送数据模块，与接收数据模块；

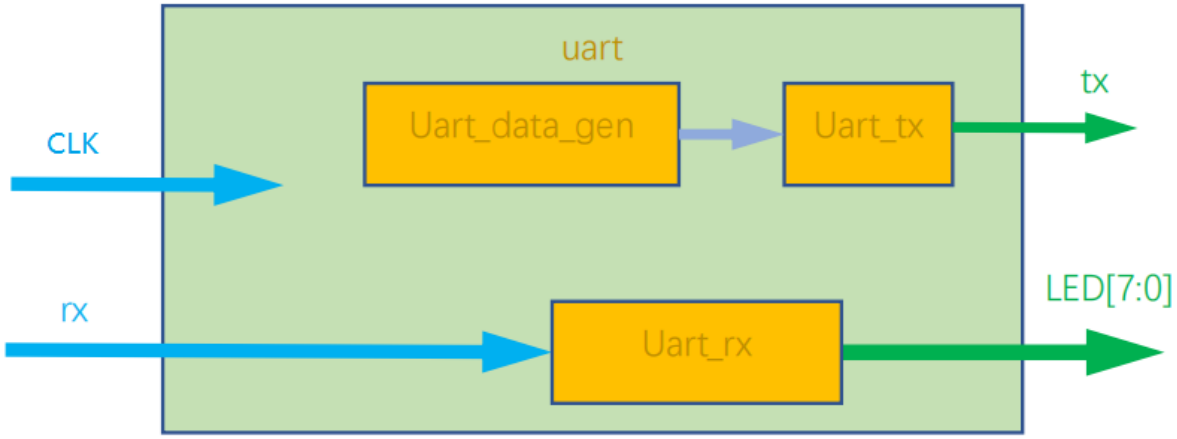


图 8.4-2

9.3.1. 串口发送模块设计

目标：接收到一个发送命令信号时，将 data[7:0]->依次发出{start,data[0:7],stop}共10bit 数据（无校验位，停止位 1bit）；

有两种方法可以将一个并行数据串行化；

方法一：通过 bit 计数与 baud 计数控制移位输出；

```
1.  // transmit bit
2.      always@(posedge clk)
3.      begin
4.          if(!rstn)
5.              txd <= `UD 1'b1;
6.          else
7.          begin
8.              if(trans_en)
9.              Begin
10. //  将开始标志和停止标志以及传输数据集成放到 trans_data  中可用下方语句
11. //              txd <= `UD trans_data[trans_bit];
12. // 单  bit  控制用下方语句
13.                 case(trans_bit)
14.                     4'h0    :txd <= `UD 1'b0;
15.                     4'h1    :txd <= `UD tx_data_reg[0];
16.                     4'h2    :txd <= `UD tx_data_reg[1];
17.                     4'h3    :txd <= `UD tx_data_reg[2];
18.                     4'h4    :txd <= `UD tx_data_reg[3];
19.                     4'h5    :txd <= `UD tx_data_reg[4];
20.                     4'h6    :txd <= `UD tx_data_reg[5];
21.                     4'h7    :txd <= `UD tx_data_reg[6];
22.                     4'h8    :txd <= `UD tx_data_reg[7];
```

```
                    4'h9     :txd <= `UD 1'b1;
                    default :txd <= `UD 1'b1;
                endcase
            end
            else
                txd <= `UD 1'b1;
        end
    end
```

这段代码用于实现 UART 模块中每一位数据的发送逻辑。模块通过一个时钟同步控制信号 txd 的高低电平，从而实现串行数据的输出。

首先看第 2-30 行的 always 块逻辑，这是在每个时钟上升沿触发的时序逻辑。模块首先判断复位信号 rstn，如果复位有效，txd 被置为高电平 1'b1，保持串口空闲状态。因为 UART 总是空闲高电平，复位时也要保证输出正确。

当复位结束后，如果传输使能信号 trans_en 有效，模块进入数据传输状态。代码通过 case 语句根据 trans_bit 的值选择发送哪一位。具体来说，trans_bit 从 0 到 9 分别对应 UART 串口的起始位、8 位数据位和停止位：当 trans_bit=0 时，发送起始位 0，表示数据传输开始；

当 trans_bit=1 到 8 时，依次发送数据寄存器 tx_data_reg[0] 到 tx_data_reg[7] 的数据位，按照从低位到高位顺序传输；当 trans_bit=9 时，发送停止位 1，表示数据传输结束；默认情况下，也保持高电平 1，保证空闲状态输出正确。

如果传输使能 trans_en 为无效状态，模块将 txd 保持高电平，这样可以保证 UART 总是在空闲时输出逻辑 1。

方法二：通过 bit 计数与 baud 计数控制状态跳转，在状态机中输出；

```
// logical ouput 状态机输出
  always @ (posedge clk)
  begin
    if(tx_en)
    begin
        case(tx_state)
            IDLE       :  uart_tx  <= `UD 1'h1;  //空闲状态输出高电平
            SEND_START :  uart_tx <= `UD 1'h0; //start 状态发送一个波特周期的低电平
            SEND_DATA  :                        //发送状态每个波特周期发送一个 bit;
            begin
                case(tx_bit_cnt)
                    3'h0  :  uart_tx  <= `UD trans_data[0];
                    3'h1  :  uart_tx  <= `UD trans_data[1];
                    3'h2  :  uart_tx  <= `UD trans_data[2];
                    3'h3  :  uart_tx  <= `UD trans_data[3];
                    3'h4  :  uart_tx  <= `UD trans_data[4];
                    3'h5  :  uart_tx  <= `UD trans_data[5];
                    3'h6  :  uart_tx  <= `UD trans_data[6];
                    3'h7  :  uart_tx  <= `UD trans_data[7];
                    default: uart_tx  <= `UD 1'h1;
                endcase
            end
            SEND_STOP  :  uart_tx  <= `UD 1'h1; //发送停止状态 输出 1 个波特周期高电平
            default    :  uart_tx  <= `UD 1'h1; // 其他状态默认与空闲状态一致，保持高电平输出
        endcase
```

```
    end
    else
        uart_tx <= `UD 1'h1;
 end
```

这里笔者采用方法二，完整 module 设计如下：

```
`timescale 1ns / 1ps
`define UD #1

module uart_tx #(
    parameter            BPS_NUM   =    16'd434
//  设置波特率为 4800 时，bit 位宽时钟周期个数:50MHz set 10417   40MHz set 8333
//  设置波特率为 9600 时，bit 位宽时钟周期个数:50MHz set 5208    40MHz set 4167
//  设置波特率为 115200 时，bit 位宽时钟周期个数:50MHz set 434   40MHz set 347 12M set 104
)
(
    input           clk,          // clock                                          时钟信号
    input [7:0]     tx_data,      // uart tx data signal byte;                      等待发送的字节数据
    input           tx_pluse,     // uart tx enable signal,rising is active;  发送模块发送触发信号

    output reg      uart_tx,      // uart tx transmit data line                     发送模块串口发送信号线
    output          tx_busy       // uart tx module work states,high is busy;发送模块忙状态指示
);

    //=====================================================================
    //wire and reg in the module
    //=====================================================================
    reg               tx_pluse_reg =0;

    reg [2:0]      tx_bit_cnt=0; //the bits number has transmited.

    reg [2:0]      tx_state=0;   //current state of tx state machine.
    reg [2:0]      tx_state_n=0; //next state of tx state machine.

    reg [3:0]      pluse_delay_cnt=0;
    reg               tx_en = 0;

    // uart tx state machine's state
    localparam   IDLE     = 4'h0; //tx state machine's state.空闲状态
    localparam   SEND_START = 4'h1; //tx state machine's state.发送 start 状态
    localparam   SEND_DATA   = 4'h2; //tx state machine's state.发送数据状态
    localparam   SEND_STOP   = 4'h3; //tx state machine's state.发送 stop 状态
    localparam   SEND_END    = 4'h4; //tx state machine's state.发送结束状态

    // uart bps set   the clk's frequency is 50MHz
    reg [15:0]    clk_div_cnt=0; //count for division the clock.

    //=====================================================================
    //logic
    //=====================================================================
    assign tx_busy = (tx_state != IDLE);
    //some control single.

    always @(posedge clk)
    begin
        tx_pluse_reg <= `UD tx_pluse;
```

```
51.     end
52.
53.     // uart 模块发送工作使能标志信号
54.     always @(posedge clk)
55.     begin
56.         if(~tx_pluse_reg & tx_pluse)
57.             tx_en <= 1'b1;
58.         else if(tx_state == SEND_END)
59.             tx_en <= 1'b0;
60.     end
61.
62.     //division the clock to satisfy baud rate.波特周期计数器
63.     always @ (posedge clk)
64.     begin
65.         if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
66.             clk_div_cnt     <= `UD 16'h0;
67.         else
68.             clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
69.     end
70.
71.     //count the number has transmited.发送数据状态中，发送 bit 位计数，以波特周期累加
72.     always @ (posedge clk)
73.     begin
74.         if(!tx_en)
75.             tx_bit_cnt      <= `UD 3'h0;
76.         else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
77.             tx_bit_cnt      <= `UD 3'h0;
78.         else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
79.             tx_bit_cnt      <= `UD tx_bit_cnt + 3'h1;
80.         else
81.             tx_bit_cnt      <= `UD tx_bit_cnt;
82.     end
83.
84.     //====================================================================
85.     //transmitter state machine
86.     //====================================================================
87.
88.     //    state change  状态跳转
89.     always @(posedge clk)
90.     begin
91.         tx_state <= tx_state_n;
92.     end
93.
94.     //    state change condition  状态跳转条件及规律
95.     always @ (*)
96.     begin
97.       case(tx_state)
98.         IDLE     :
99.         begin
100.        //检测到 tx_pluse 上升沿后，立即进入 SEND_START 状态
101.                if(~tx_pluse_reg & tx_pluse)
102.                 tx_state_n = SEND_START;
103.            else
104.                tx_state_n = tx_state;
105.        end
106.            SEND_START :
107.            Begin
108.                //发送一个波特周期的低电平后进入，发送数据状态
```

```
109.                if(clk_div_cnt == BPS_NUM)
110.                 tx_state_n = SEND_DATA;
111.             else
112.               tx_state_n = tx_state;
113.          end
114.             SEND_DATA :
115.             Begin
116.                //计时 8 个波特周期（发送了 8bit 数据）
117.                if(tx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
118.                 tx_state_n = SEND_STOP;                              //跳转到发送 stop 状态
119.             else
120.               tx_state_n = tx_state;
121.          end
122.             SEND_STOP :
123.             begin
124.                if(clk_div_cnt == BPS_NUM)          //设置停止位宽为 1 个波特周期计数
125.                 //发送一个波特周期的高电平，之后跳转到发送结束状态
126.                 tx_state_n = SEND_END;
127.             else
128.               tx_state_n = tx_state;
129.             end
130.             SEND_END :   tx_state_n = IDLE;
131.             default :   tx_state_n = IDLE;
132.           endcase
133.         end
134.
135.         //    logical ouput   状态机输出
136.         always @ (posedge clk)
137.         begin
138.           if(tx_en)
139.           begin
140.               case(tx_state)
141.                   IDLE        :   uart_tx   <= `UD 1'h1;              //空闲状态输出高电平
142.                   SEND_START :   uart_tx   <= `UD 1'h0;   //start 状态发送一个波特周期的低电平
143.                   SEND_DATA   :           //发送状态每个波特周期发送一个 bit;
144.                   begin
145.                       case(tx_bit_cnt)
146.                           3'h0   :   uart_tx   <= `UD tx_data[0];
147.                           3'h1   :   uart_tx   <= `UD tx_data[1];
148.                           3'h2   :   uart_tx   <= `UD tx_data[2];
149.                           3'h3   :   uart_tx   <= `UD tx_data[3];
150.                           3'h4   :   uart_tx   <= `UD tx_data[4];
151.                           3'h5   :   uart_tx   <= `UD tx_data[5];
152.                           3'h6   :   uart_tx   <= `UD tx_data[6];
153.                           3'h7   :   uart_tx   <= `UD tx_data[7];
154.                           default: uart_tx   <= `UD 1'h1;
155.                       endcase
156.                   end
157.                   //发送停止状态  输出 1 个波特周期高电平
158.                   SEND_STOP   :   uart_tx   <= `UD 1'h1;
159.
160.                   default     :   uart_tx   <= `UD 1'h1;
161.                // 其他状态默认与空闲状态一致，保持高电平输出
162.               endcase
163.           end
164.           else
165.               uart_tx <= `UD 1'h1;
166.         end
```

```
167.
168.        endmodule
```

模块代码使用状态机实现了 UART 串口的字节发送功能，通过波特率控制时钟分频和状态机管理数据传输流程。模块以 50MHz 时钟为基础，通过参数 BPS_NUM 配置波特率的时钟周期数，例如 4800 波特率对应 10417 个时钟周期，9600 波特率对应 5208 个时钟周期，115200 波特率对应 434 个时钟周期。

代码的 48 行到 166 行是 UART 串口发送控制逻辑，主要用于将输入的字节数据 tx_data 按照设定的波特率发送到串口输出 uart_tx，并通过 tx_busy 输出模块忙状态指示。

首先代码的 48-51 行是 tx_pluse 同步逻辑，通过寄存器 tx_pluse_reg 在每个时钟上升沿采样 tx_pluse 信号，用于检测发送触发的上升沿。紧接着在代码的 54-60 行，通过判断 tx_pluse 上升沿设置发送使能 tx_en，当检测到触发信号时拉高 tx_en，表示模块开始发送；当状态机进入发送结束状态 SEND_END 时，将 tx_en 清零，模块返回空闲状态。

代码的 63-69 行是波特率周期计数器 clk_div_cnt，用于将系统时钟分频以满足指定波特率。在每个波特周期计数完成或发送触发信号到来时清零计数器，否则每个时钟上升沿累加 1，实现对波特周期的精确控制。

在代码的 72-82 行，tx_bit_cnt 用于记录当前发送的数据位编号。在发送使能为低时计数器清零；在 SEND_DATA 状态下，每完成一个波特周期累加 1，发送完 8 位数据后清零，为发送停止位做好准备。

状态机逻辑在代码的 89-134 行实现。状态机共包含五个状态：IDLE（空闲）、SEND_START（发送起始位）、SEND_DATA（发送数据位）、SEND_STOP（发送停止位）和 SEND_END（发送结束）。状态跳转逻辑通过组合逻辑 tx_state_n 计算下一状态：在空闲状态检测到发送触发信号时进入起始位状态，发送一个波特周期的低电平后进入数据发送状态，依次发送 8 位数据后进入停止位状态，发送一个波特周期的高电平后进入发送结束状态，最终回到空闲状态。状态更新在每个时钟上升沿通过 tx_state <= tx_state_n 完成。

在代码的 135-168 行是状态机的输出逻辑。在发送使能有效时，根据当前状态控制 uart_tx 输出电平：空闲和结束状态输出高电平，起始位输出低电平，数据发送状态按 tx_bit_cnt 输出对应的数据位，停止位输出高电平；当发送使能无效时，输出保持高电平，保证串口线在空闲时为逻辑 1。

9.3.2. 串口接收模块设计

串口接收模块是发射模块的逆过程，设计思路区别不大，但是有如下几点需要注意：

1.接收开始信号，当 rx 下降沿到来后保持几个时钟周期的低电平，表明进入接收 start；

2.接收数据提取位置，前面讲发射的时候都是在波特周期开始的位置变更数据，接收数据提取时需要在 rx 稳定时刻取数，去波特周期的中间位置取数；

3.最终输出数据锁存，在最后 1bit 存入寄存器后需要对接收数据锁存，并在之后需要给出数据使能信号，表示输出数据有效；

Module 设计如下：

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.
4.  module uart_rx # (
5.      parameter              BPS_NUM      =     16'd434
6.  //  设置波特率为 4800 时，   bit 位宽时钟周期个数:50MHz set 10417   40MHz set 8333
7.  //  设置波特率为 9600 时，   bit 位宽时钟周期个数:50MHz set 5208    40MHz set 4167
8.  //  设置波特率为 115200 时，bit 位宽时钟周期个数:50MHz set 434     40MHz set 347
9.  )
10. (
11.       //input ports
12.       input              clk,
13.       input              uart_rx,
14.
15.       //output ports
16.       output reg [7:0]   rx_data,
17.       output reg         rx_en,
18.       output             rx_finish
19. );
20.
21.     // uart rx state machine's state
22.     localparam   IDLE          = 4'h0;     //空闲状态，等待开始信号到来.
23.     localparam   RECEIV_START = 4'h1;     //接收 Uart 开始信号，低电平一个波特周期.
24. //接收 Uart 传输数据信号，此工程定义传输 8bit，每个波特周期中间位置取值，8 个周期后跳转到 stop 状态
25.     localparam   RECEIV_DATA   = 4'h2;     .
26.  //停止状态数据线是高电平，与空闲状态是一致的按照协议标准需要等待一个停止位周期再做状态跳转.
27.     localparam   RECEIV_STOP   = 4'h3;
28.     localparam   RECEIV_END    = 4'h4;     //结束中转状态.
29.
30.     //=============================================================================
31.     //wire and reg in the module
32.     //=============================================================================
33.     reg     [2:0]         rx_state=0;         //current state of tx state machine 当前状态
34.     reg     [2:0]         rx_state_n=0;       //next state of tx state machine     下一个状态
35.     reg     [7:0]         rx_data_reg;        //                                   接收数据缓冲寄存器
36.     reg                   uart_rx_1d;         //save uart_rx one cycle.       保存 uart_rx 一个时钟周期
37.     reg                   uart_rx_2d;         //save uart_rx one cycle.       保存 uart_rx  前两个时钟周期
38.     wire                  start;             //active when start a byte receive.     检测到 start 信号标志
39.     reg     [15:0]         clk_div_cnt;       //count for division the clock.       波特周期计数器
40.
41.     //=============================================================================
42.     //logic
43.     //=============================================================================
44.
45.     //some control single.
46.     always @ (posedge clk)
47.     begin
48.         uart_rx_1d <= `UD uart_rx;
49.         uart_rx_2d <= `UD   uart_rx_1d;
50.     end
```

```
51.
52.     assign start       = (!uart_rx) && (uart_rx_1d || uart_rx_2d);
53.     assign rx_finish = (rx_state == RECEIV_END);
54.
55.     //division the clock to satisfy baud rate.波特周期计数器
56.     always @ (posedge clk)
57.     begin
58.         if(rx_state == IDLE || clk_div_cnt == BPS_NUM)
59.             clk_div_cnt     <= `UD 16'h0;
60.         else
61.             clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
62.     end
63.
64.     // receive bit data numbers
65.     //在接收数据状态中，接收的 bit 位计数，每一个波特周期计数加 1
66.     reg     [2:0]       rx_bit_cnt=0;     //the bits number has transmited.
67.     always @ (posedge clk)
68.     begin
69.         if(rx_state == IDLE)
70.             rx_bit_cnt <= `UD 3'h0;
71.         else if((rx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
72.             rx_bit_cnt <= `UD 3'h0;
73.         else if((rx_state == RECEIV_DATA) && (clk_div_cnt == BPS_NUM))
74.             rx_bit_cnt <= `UD rx_bit_cnt + 3'h1;
75.         else
76.             rx_bit_cnt <= `UD rx_bit_cnt;
77.     end
78.
79. //=============================================================================
80. //receive state machine
81. //=============================================================================
82.     //状态机状态跳转
83.     always @(posedge clk)
84.     begin
85.         rx_state <= rx_state_n;
86.     end
87.
88.     //状态机状态跳转条件及跳转规律
89.     always @ (*)
90.     begin
91.       case(rx_state)
92.           IDLE         :
93.           begin
94.               if(start)                    //监测到 start 信号到来，下一状态跳转到 start 状态
95.                   rx_state_n = RECEIV_START;
96.               else
97.                   rx_state_n = rx_state;
98.           end
99.           RECEIV_START     :
100.          begin
101.                  if(clk_div_cnt == BPS_NUM)              //已完成接收 start 标志信号
102.                      rx_state_n = RECEIV_DATA;
103.                  else
104.                      rx_state_n = rx_state;
105.              end
106.              RECEIV_DATA     :
107.              begin
```

```
108.                 if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM) //已完成 8bit 数据的传输
109.                     rx_state_n = RECEIV_STOP;
110.                 else
111.                     rx_state_n = rx_state;
112.             end
113.             RECEIV_STOP     :
114.             begin
115.                 if(clk_div_cnt == BPS_NUM)                          //已完成接收 stop 标志信号
116.                     rx_state_n = RECEIV_END;
117.                 else
118.                     rx_state_n = rx_state;
119.             end
120.             RECEIV_END     :
121.             begin
122.                 if(!uart_rx_1d)  //数据线重新被拉低，新数据传输又发送 start 信号，需要跳转到 start 状态
123.                     rx_state_n = RECEIV_START;
124.                 else               //没有其他状况出现时，回到空闲状态，等待 start 信号的到来
125.                     rx_state_n = IDLE;
126.             end
127.             default     :  rx_state_n = IDLE;
128.         endcase
129.     end
130.
131.     // 状态机输出
132.     always @ (posedge clk)
133.     begin
134.         case(rx_state)
135.             IDLE           ,
136.             RECEIV_START :   //在空闲和 start 状态时将接收数据缓冲寄存器和数据使能置位；
137.             begin
138.                 rx_en <= `UD 1'b0;
139.                 rx_data_reg <= `UD 8'h0;
140.             end
141.             RECEIV_DATA   :
142.             begin
143.                 if(clk_div_cnt == BPS_NUM[15:1])     //在波特周期的中间位置取传输的数据；
144.     //以循环右移的方式将 uart_rx 数据填入缓冲寄存器的最高位（Uart 传输低位在前，最后一个 bit 刚好是最高位）
145.                     rx_data_reg   <= `UD {uart_rx , rx_data_reg[7:1]};
146.             end
147.             RECEIV_STOP   :
148.             begin
149.                 rx_en     <= `UD 1'b1;                 // 输出使能信号，表示最新的数据输出有效
150.                 rx_data <= `UD rx_data_reg;        // 将缓冲寄存器的值赋值给输出寄存器
151.             end
152.             RECEIV_END     :
153.             begin
154.                 rx_data_reg <= `UD 8'h0;
155.             end
156.             default:     rx_en <= `UD 1'b0;
157.         endcase
158.     end
159.
160. endmodule
```

串口接收模块代码的 46 行到 158 行是 UART 串口接收控制逻辑，主要用于从串口输入 uart_rx 接收字节数据，并输出接收数据 rx_data 与有效信号 rx_en，同时通过 rx_finish 指示接收过程是否完成。

46-50 行是 uart_rx 信号同步逻辑，通过寄存器 uart_rx_1d 和 uart_rx_2d 在每个时钟上升沿采样 uart_rx 信号，消除异步信号干扰，并用于检测起始位。紧接着在 52 行，通过组合逻辑 start 检测字节起始信号，即串口线从高电平下降到低电平时标记一个字节接收开始。rx_finish 信号在状态机进入 RECEIV_END 状态时拉高，表示接收过程完成。

而 56-62 行是波特率周期计数器 clk_div_cnt，用于将系统时钟分频以满足指定波特率。在空闲状态或计数达到波特周期时计数器清零，否则每个时钟上升沿累加 1。

在代码的 67-77 行，rx_bit_cnt 用于记录当前接收的数据位编号。在空闲状态清零计数器；在 RECEIV_DATA 状态下，每完成一个波特周期累加 1，接收完 8 位数据后清零，为接收停止位做好准备。

状态机逻辑在代码的 83-129 行实现。状态机共包含五个状态：IDLE（空闲）、RECEIV_START（接收起始位）、RECEIV_DATA（接收数据位）、RECEIV_STOP（接收停止位）和 RECEIV_END（接收结束）。状态跳转逻辑通过组合逻辑 rx_state_n 计算下一状态：空闲状态检测到 start 信号时进入起始位状态，接收一个波特周期的低电平后进入数据接收状态，依次接收 8 位数据后进入停止位状态，接收一个波特周期的高电平后进入结束状态，最终回到空闲状态；如果在结束状态检测到串口线被拉低，表示新字节开始接收，则跳转到起始位状态。状态更新在每个时钟上升沿通过 rx_state <= rx_state_n 完成。

132-158 行是状态机的输出逻辑。在空闲和起始位状态，将接收数据缓冲寄存器和数据使能清零；在数据接收状态，通过在波特周期中间采样串口线的电平，将数据按低位先行的顺序填入缓冲寄存器；在停止位状态，将缓冲寄存器的值赋给输出寄存器 rx_data 并拉高 rx_en 表示数据有效；在结束状态清零缓冲寄存器，为下一次接收做好准备。

9.3.3. 串口发送控制模块设计

目标：产生 1S 间隔的触发信号并输出第一个发送字节，busy 的下降沿时输出下一个字节；

Module 如下：

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.  module uart_data_gen(
4.      input                clk,
5.      input       [7:0]    read_data,
6.      input                tx_busy,
7.      input       [7:0]    write_max_num,
8.      output reg  [7:0]    write_data,
9.      output reg           write_en
10.  );
11.
12.          // set every second send a string,"====HELLO WORLD==="
```

```
13.         // 设置约每秒发送一个字符串
14.         reg [25:0] time_cnt=0;
15.         reg [ 7:0] data_num;
16.         always @(posedge clk)
17.         begin
18.             time_cnt <= `UD time_cnt + 26'd1;
19.         end
20.
21.         // 设置串口发射工作区间
22.         reg          work_en=0;
23.         reg          work_en_1d=0;
24.         always @(posedge clk)
25.         begin
26.             if(time_cnt == 26'd2048)
27.                 work_en <= `UD 1'b1;
28.             else if(data_num == write_max_num-1'b1)
29.                 work_en <= `UD 1'b0;
30.         end
31.
32.         always @(posedge clk)
33.         begin
34.             work_en_1d <= `UD work_en;
35.         end
36.
37.         // get the tx_busy‘s falling edge     获取 tx_busy 的下降沿
38.         reg              tx_busy_reg=0;
39.         wire              tx_busy_f;
40.         always @ (posedge clk) tx_busy_reg <= `UD tx_busy;
41.
42.         assign tx_busy_f = (!tx_busy) && (tx_busy_reg);
43.
44.         // 串口发射数据触发信号
45.         reg write_pluse;
46.         always @ (posedge clk)
47.         begin
48.             if(work_en)
49.             begin
50.                 if(~work_en_1d || tx_busy_f)
51.                     write_pluse <= `UD 1'b1;
52.                 else
53.                     write_pluse <= `UD 1'b0;
54.             end
55.             else
56.                 write_pluse <= `UD 1'b0;
57.         end
58.
59.         always @ (posedge clk)
60.         begin
61.             if(~work_en & tx_busy_f)
62.                 data_num     <= 7'h0;
63.             else if(write_pluse)
64.                 data_num     <= data_num + 8'h1;
65.         end
66.
67.         always @(posedge clk)
68.         begin
69.             write_en <= `UD write_pluse;
70.         end
```

```
71.
72.     //  字符的对应 ASCII 码
73.         always @ (posedge clk)
74.         begin
75.             case(data_num)
76.                 8'h0  ,
77.                 8'h1  : write_data <= `UD 8'h77;// ASCII code is w
78.                 8'h2  : write_data <= `UD 8'h77;// ASCII code is w
79.                 8'h3  : write_data <= `UD 8'h77;// ASCII code is w
80.                 8'h4  : write_data <= `UD 8'h2E;// ASCII code is .
81.                 8'h5  : write_data <= `UD 8'h6D;// ASCII code is m
82.                 8'h6  : write_data <= `UD 8'h65;// ASCII code is e
83.                 8'h7  : write_data <= `UD 8'h79;// ASCII code is y
84.                 8'h8  : write_data <= `UD 8'h65;// ASCII code is e
85.                 8'h9  : write_data <= `UD 8'h73;// ASCII code is s
86.                 8'ha  : write_data <= `UD 8'h65;// ASCII code is e
87.                 8'hb  : write_data <= `UD 8'h6D;// ASCII code is m
88.                 8'hc  : write_data <= `UD 8'h69;// ASCII code is i
89.                 8'hd  : write_data <= `UD 8'h2E;// ASCII code is .
90.                 8'he  : write_data <= `UD 8'h63;// ASCII code is c
91.                 8'hf  : write_data <= `UD 8'h6F;// ASCII code is o
92.                 8'h10 : write_data <= `UD 8'h6D;// ASCII code is m
93.                 8'h11 ,
94.                 8'h12 : write_data <= `UD 8'h0d;
95.                 8'h13 : write_data <= `UD 8'h0a;
96.                 default : write_data <= `UD read_data;
97.             endcase
98.         end
99.
100. .   endmodule
```

串口发送控制模块，用于按照固定时间间隔生成指定字符序列并通过串口发送。模块以系统时钟 clk 为基础，通过控制逻辑管理发送触发和数据输出。

代码的 14 行到 98 行是串口数据生成与发送控制逻辑，主要功能是每隔一定时间生成一个指定字符串（如 "www.meyesemi.com\r\n"）并通过 write_data 和 write_en 输出给 UART 发送模块。

首先，代码的 14-19 行是时间计数器 time_cnt，每个时钟上升沿累加 1，用于实现发送周期控制，本实验中是实现功能是每秒触发一次字符串发送。

在 22-30 行，通过 work_en 信号控制串口发送工作区间：当 time_cnt 达到设定值时开始发送，当发送的数据量达到 write_max_num-1 时结束发送。work_en_1d 在 32-35 行用于寄存器同步，实现发送使能的边沿检测。

38-42 行通过寄存器 tx_busy_reg 同步 tx_busy 信号，并生成 tx_busy_f 信号，用于检测 UART 发送模块的忙状态下降沿，作为发送触发条件之一。

代码的 45-57 行生成 write_pluse 发送脉冲信号：在工作使能状态下，如果刚开始发送或检测到 tx_busy 下降沿，则拉高 write_pluse，触发 UART 发送模块发送一个字节。

在 59-65 行，data_num 用于记录当前发送的字节序号：当发送结束且 tx_busy 下降沿出现时清零；当 write_pluse 有效时累加，为依次输出字符做索引。write_en 在 6

7-70 行直接由 write_pluse 同步输出。

在 73-98 行，通过 case 语句将 data_num 对应到具体 ASCII 字符，实现字符映射输出：如索引 0-2 输出 'w'，索引 4 输出 '.'，索引 5-10 输出 "meyese" 等，最后两个索引输出回车换行 0x0d 0x0a，default 情况下输出 read_data。

9.3.4. 串口实验顶层模块设计

目标：板子 1s 向串口助手发送一次十进制显示的“www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字，LED 以二进制显示亮起。

Uart_data_gen 模块产生一个间隔 1S 钟的触发信号，同时输出第一个发送字节，等待 uart_tx 输出的 busy 下降沿到来，获知 uart_tx 进入空闲状态可发送下一个 byte 时，再次给出串口发送的触发脉冲，并输出下一个字节；

Uart_rx 模块接收到数据后输出一个 rx_en 信号（接收数据使能信号）、一组接收数据信号；接收的数据信号是锁存的，可直接点亮 LED 灯；

具体的 module 实现如下:

```
1.  `timescale 1ns / 1ps
2.  `define UD #1
3.
4.  module uart_top(
5.      //input ports
6.      input           clk,
7.      input           uart_rx,
8.
9.      //output ports
10.     output  [7:0] led,
11.     output          uart_tx);
12. );
13.
14.   parameter       BPS_NUM = 16'd434;
15.   // 设置波特率为 4800 时，bit 位宽时钟周期个数:50MHz set 10417   40MHz set 8333
16.   // 设置波特率为 9600 时，bit 位宽时钟周期个数:50MHz set 5208    40MHz set 4167
17.   // 设置波特率为 115200 时，bit 位宽时钟周期个数:50MHz set 434   40MHz set 347 12M set 104
18.
19.
20. //====================
21. //wire and reg in the module
22. //====================
23.
24.     wire            tx_busy;    //transmitter is free.
25.     wire            rx_finish;   //receiver is free.
26.     wire    [7:0] rx_data;      //the data receive from uart_rx.
27.     wire    [7:0] tx_data;
28.     wire            tx_en;       //enable transmit.
29.
30. //======================
31. //logic
32. //======================
33.     wire rx_en;
34. //======================
35. //instance
```

```
36. //======================
37.     reg   [7:0] receive_data;
38.     always @(posedge clk)   receive_data <= led;
39.     uart_data_gen uart_data_gen(
40.         .clk                (clk         ),     //input                clk,
41.         .read_data          (receive_data ),    //input        [7:0]   read_data,
42.         .tx_busy            (tx_busy     ),     //input                tx_busy,
43.         .write_max_num      (8'h14       ),     //input        [7:0]   write_max_num,
44.         .write_data         (tx_data     ),     //output reg [7:0]   write_data
45.         .write_en           (tx_en       )     //output reg           write_en
46.     );
47.
48.     //uart transmit data module.
49.     uart_tx #(
50.         .BPS_NUM          (  BPS_NUM    ) //parameter BPS_NUM   =     16'd434
51.      )
52.      u_uart_tx(
53.         .clk              (  clk        ),// input              clk,
54.         .tx_data          (  tx_data    ),// input [7:0]        tx_data,
55.         .tx_pluse         (  tx_en      ),// input              tx_pluse,
56.         .uart_tx          (  uart_tx    ),// output reg         uart_tx,
57.         .tx_busy          (  tx_busy    ) // output            tx_busy
58.     );
59.
60.     //Uart receive data module.
61.     uart_rx #(
62.         .BPS_NUM          (  BPS_NUM      ) //parameter   BPS_NUM   =     16'd434
63.      )
64.      u_uart_rx (
65.         .clk              (  clk        ),// input              clk,
66.         .uart_rx          (  uart_rx    ),// input              uart_rx,
67.         .rx_data          (  rx_data    ),// output reg [7:0]   rx_data,
68.         .rx_en            (  rx_en      ),// output reg         rx_en,
69.         .rx_finish        (  rx_finish  ) // output            rx_finish
70.     );
71.
72.     assign led = rx_data;
73.
74. endmodule
```

模块进行信号声明与内部连线数据生成模块实例化，实例化 uart_data_gen 模块，用于定时生成串口发送数据。UART 发送模块实例化（uart_tx），负责将 tx_data 按波特率发送到 uart_tx 输出端。UART 接收模块实例化（uart_rx）接收外部串口信号 uart_rx，并将接收到的数据输出到 rx_data，同时生成 rx_en 和 rx_finish 信号。37-38 行使用寄存器 receive_data 缓存 LED 状态。72 行将 rx_data 直接输出到 LED，实现接收到的数据通过 LED 显示。

顶层模块主要是整合 UART 接收、数据生成和 UART 发送模块，实现了串口数据收发与 LED 显示功能。接收到的数据实时显示在 LED 上，同时模块可按照设定周期生成数据并通过 UART 发送，实现串口双向通信。

9.4. 实验现象

用 SSCOM 串口调试工具，波特率设置为 115200bps，数据格式为 1 位起始位、8 位

数据位、无校验位、1 位结束位，用 Type-C 连接开发板与电脑后有如下现象：

实验现象一：在串口工具中每隔 1S 中打印一次：“www.meyesemi.com”；

实验现象二：

在串口工具上以 Hex 格式发送 55；我们可看到 PT2G390H 板卡上的 LED1,LED3,LED5,LED7 为熄灭，LED2,LED4,LED6,LED8 为点亮状态；

发送 F0；我们可看到 PT2G390H 板卡上的 LED1,LED2,LED3,LED4 为熄灭，LED5,LED6,LED7,LED8 为点亮状态。

10. HDMI 实验例程

10.1. 实验简介

实验目的：

实验 1：PT2G390H 开发板通过 HDMI 在屏幕上显示彩条；

实验 2.PT2G390H 开发板 HDMIIN 接收，通过 HDMIOUT 实现环路输出；

实验环境：

Window11

PDS2023.2-SP3-ads

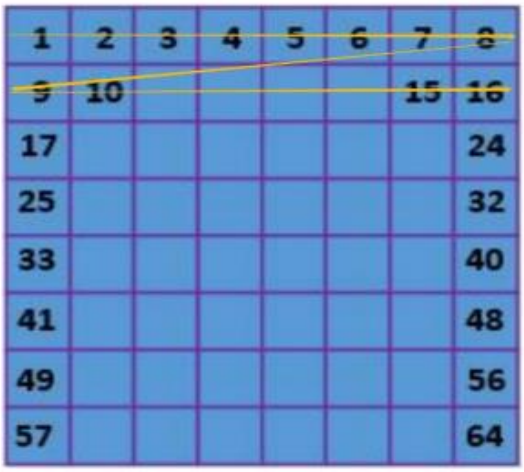
硬件环境：

PT2G390H 开发板

10.2. 实验原理

10.2.1. 显示原理

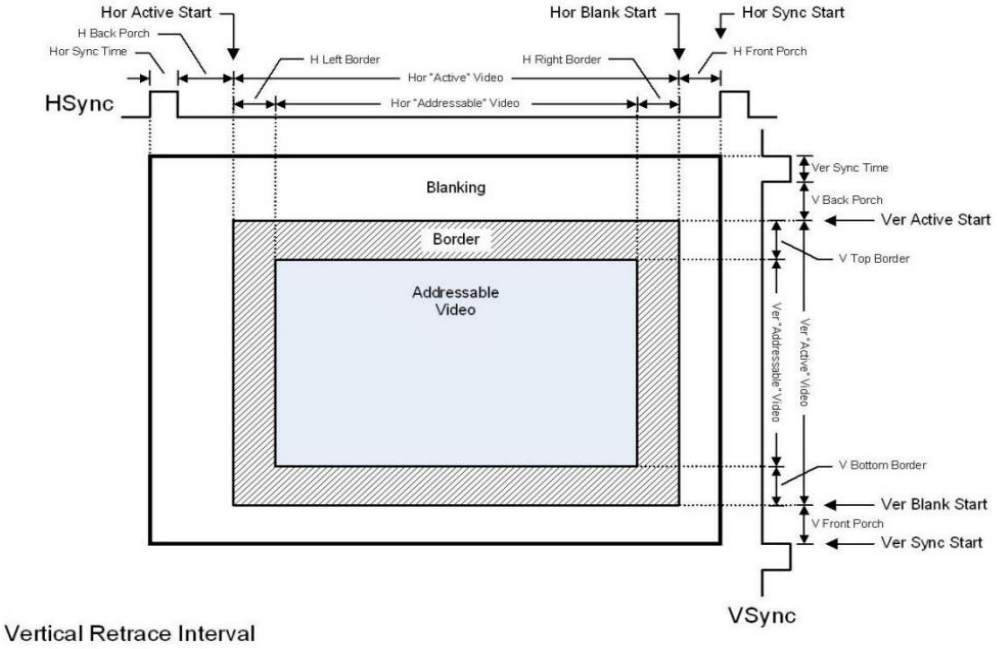
下图表示一个 8*8 像素的画面，图中每个格子表示一个像素点，显示图像时像素点快速点亮的过程按表格中编号的顺序逐个点亮，从左到右，从上到下，按图中箭头方向的“Z”字形顺序。



以上图为例，每行 8 个像素点，每完成一行信号的传输，会转到下一行信号传输，直到完成第 8 行数据的传输，就完成了一个画面的数据传输了，一个画面也称为一场或一帧，显示每秒中刷新的帧数称为帧率。比如 1920*1080P 像素，就是 1 行有效像素点 1920，一场（也就是一帧）有效行为 1080 行。

每个像素点的像素值数据，对应每个像素点的颜色。常见的像素值表示格式比如：RGB888，RGB 分别代表：红 R,绿 G，蓝 B，888 是指 R、G、B 分别有 8bit，也就是 R、G、B 每一色光有 2^8=256 级阶调，通过 RGB 三色光的不同组合，一个像素上最多

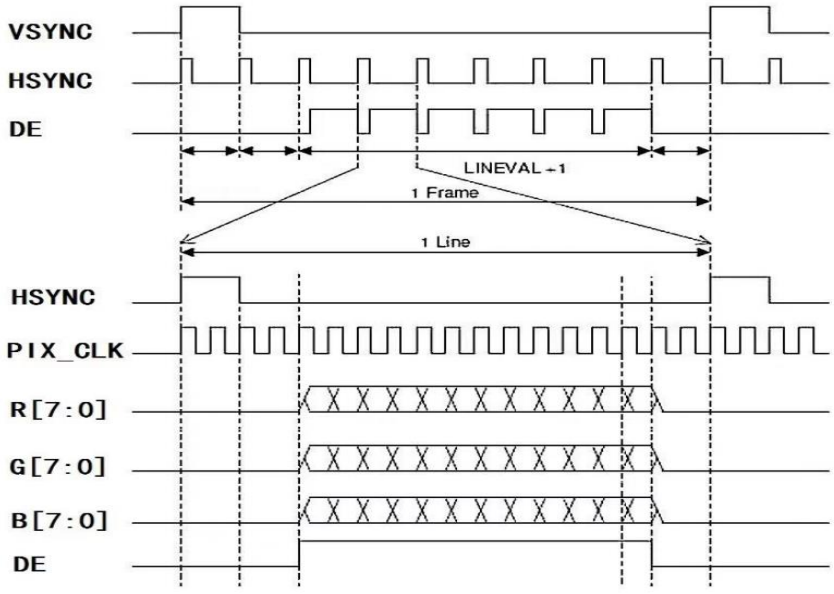
可显示 24 位的 256*256*256=16,777,216 色。



像素数据源源不断输送进来，行、场的切换通过行场同步信号来控制，即 hsync（行同步）和 vsync（场同步信号）。

上图中 Addressable 部分内容是在显示器中可看到的区域，像素点是否有效通过 DE 信号标识；Border 可理解为显示黑边或者显示边框，通常 Border 显示的像素值是 0（黑色）。行、场切换过程都是在用户感受不到的区域进行的，这个区域就是 Blanking 部分，称为消隐区间。同步信号上升沿表示新的一行/一场开始，Hsync 对应行，Vsync 对应场。

彩条产生：



本实验采用 1920*1080@60 的视频规格，详细时序参数如下：

VESA MONITOR TIMING STANDARD

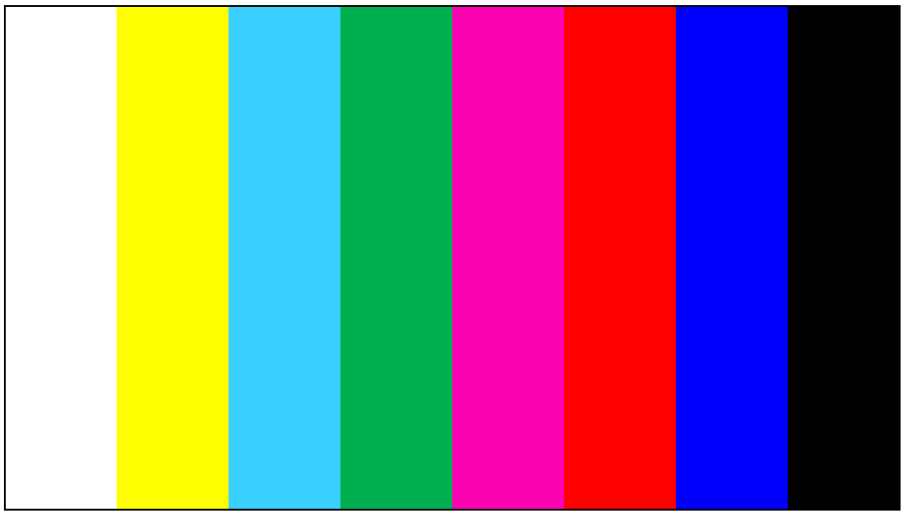
Adopted: 11/17/08
Resolution: 1920 x 1080 at 60 Hz (non-interlaced)
EDID ID: DMT ID: 52h; STD 2 Byte Code: (D1, C0)h; CVT 3 Byte Code: n/a
Method: *** NOT CVT COMPLIANT ***
Per CEA-861 --- 1080p (Code 16) Timing Definition

Detailed Timing Parameters

Timing Name	= **1920 x 1080 @ 60Hz;**					
Hor Pixels	= **1920;**	// Pixels				
Ver Pixels	= **1080;**	// Lines				
Hor Frequency	= 67.500;	// kHz	=	14.8 usec	/	line
Ver Frequency	= 60.000;	// Hz	=	16.7 msec	/	frame
Pixel Clock	= **148.500;**	// MHz	=	6.7 nsec		± 0.5%
Character Width	= **4;**	// Pixels	=	26.9 nsec		
Scan Type	= **NONINTERLACED;**			// H Phase	=	1.4 %
Hor Sync Polarity	= **POSITIVE**	// HBlank	=	12.7% of HTotal		
Ver Sync Polarity	= **POSITIVE**	// VBlank	=	4.0% of VTotal		
Hor Total Time	= 14.815;	// (usec)	=	550 chars	=	2200 Pixels
Hor Addr Time	= 12.929;	// (usec)	=	480 chars	=	1920 Pixels
Hor Blank Start	= 12.929;	// (usec)	=	480 chars	=	1920 Pixels
Hor Blank Time	= 1.886;	// (usec)	=	70 chars	=	280 Pixels
Hor Sync Start	= 13.522;	// (usec)	=	502 chars	=	2008 Pixels
// H Right Border	= 0.000;	// (usec)	=	**0** chars	=	0 Pixels
// H Front Porch	= 0.593;	// (usec)	=	**22** chars	=	88 Pixels
Hor Sync Time	= 0.296;	// (usec)	=	**11** chars	=	44 Pixels
// H Back Porch	= 0.997;	// (usec)	=	**37** chars	=	148 Pixels
// H Left Border	= 0.000;	// (usec)	=	**0** chars	=	0 Pixels
Ver Total Time	= 16.667;	// (msec)	=	1125 lines		HT – (1.06xHA)
Ver Addr Time	= 16.000;	// (msec)	=	1080 lines		= 1.11
Ver Blank Start	= 16.000;	// (msec)	=	1080 lines		
Ver Blank Time	= 0.667;	// (msec)	=	45 lines		
Ver Sync Start	= 16.059;	// (msec)	=	1084 lines		
// V Bottom Border	= 0.000;	// (msec)	=	**0** lines		
// V Front Porch	= 0.059;	// (msec)	=	**4** lines		
Ver Sync Time	= 0.074;	// (msec)	=	**5** lines		
// V Back Porch	= 0.533;	// (msec)	=	**36** lines		
// V Top Border	= 0.000;	// (msec)	=	**0** lines		

HDMI 显示的数据源采用 verilog 编写的显示时序产生模块 sync_vg 实现上图的时序，彩条生成模块 pattern_vg 根据像素点所在位置，即列数和行数确定像素值，实现彩条图案。

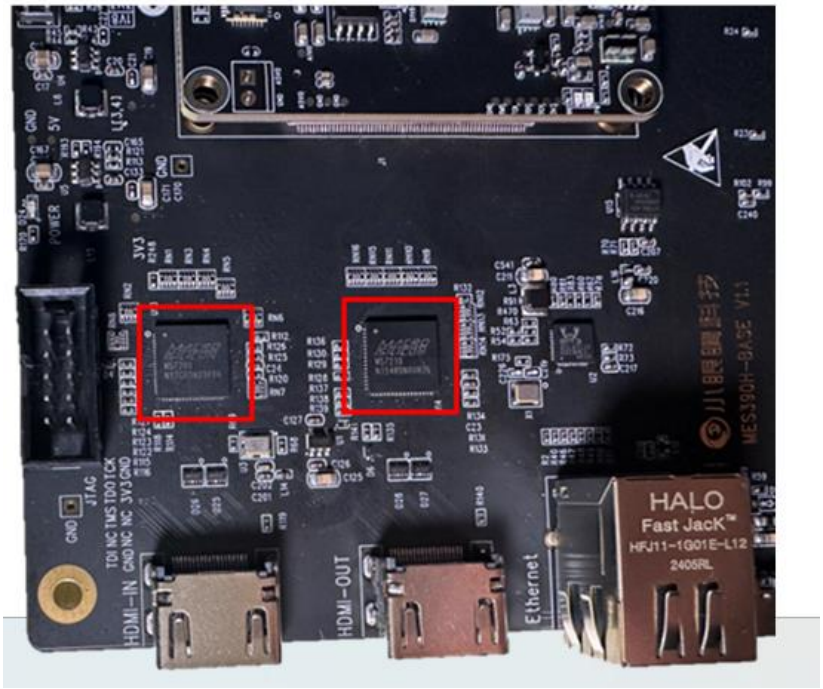
彩条按照每行均匀分成 8 部分，根据每行的像素点数的范围对像素值设置成对应的颜色，实现彩条信号。



10.2.2. HDMI_PHY 配置

HDMI 输入接口采用宏晶微 MS7200HMDI 接收芯片，HDMI 输出接口采用宏晶微 MS7210HMDI 发送芯片。芯片兼容 HDMI1.4b 及以下标准视频的 3D 传输格式，最高分辨率高达 4K@30Hz，最高采样率达到 300MHz，支持 YUV 和 RGB 之间的色彩空间转换，数字接口支持 YUV 及 RGB 格式。

MS7200 和 MS7210 的 IIC 配置接口与 FPGA 的 IO 相连，通过 FPGA 的编程来对芯片进行初始化和配置操作。

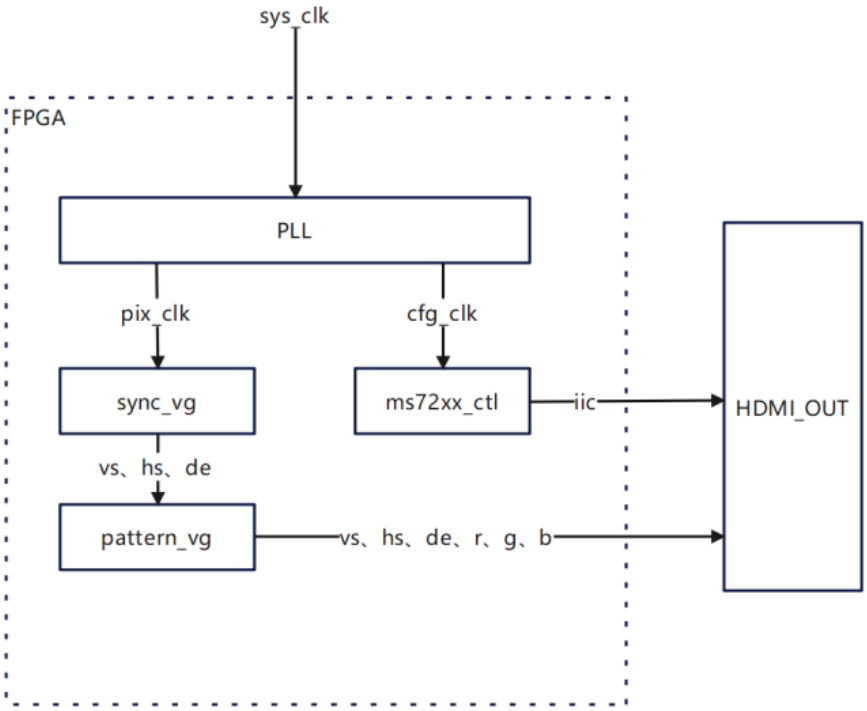


hdmi_loop 例程包含对 MS7200 和 MS7210 的配置模块 ms72xx_ctl，已将 MS7200 和 MS7210 配置成 1920*1080@60RGB888 模式，配置流程参考源码，用户可例化模块 ms72xx_ctl 使用。

10.2.3. 实验源码设计

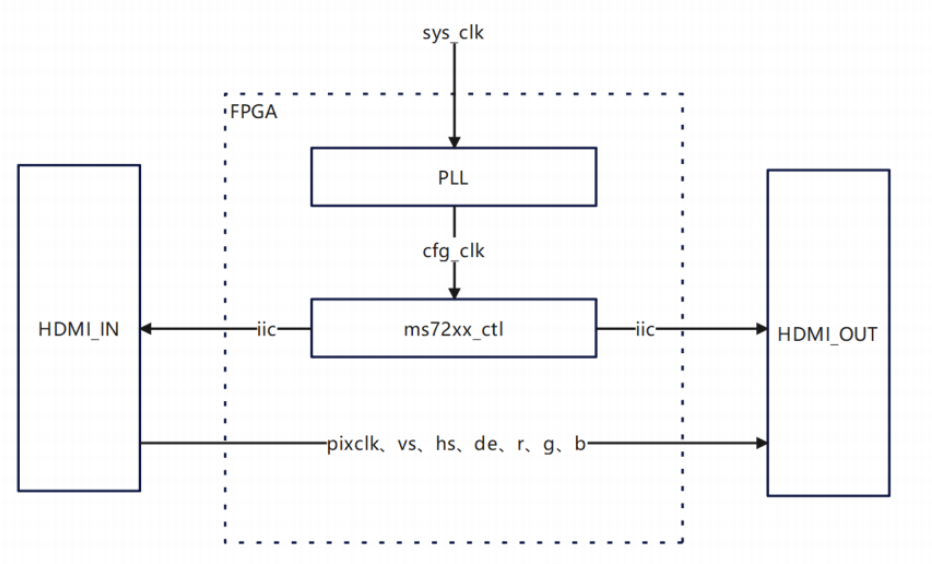
实验 1：hdmi_test

HDMI 输出彩条显示例程，分成 4 个模块，时钟模块 pll、MS7210 配置模块 ms72xx_ctl、显示时序产生模块 sync_vg、彩条生成模块 pattern_vg，以下为模块拓扑图，源码详情请查看 demo。



实验 2：hdmi_loop

HDMI 环路例程，将 MS7200 接收的信号给到 MS7210 即可实现 HDMI 环路输出，以下为模块拓扑图，源码详情请查看 demo。



10.3. 实验现象

实验 1 现象：hdmi_test

连接好 PT2G390H 开发板和显示器，下载程序，可以看到显示器显示 8 条彩条。

实验 2 现象：hdmi_loop

连接好 PT2G390H 开发板、视频源和显示器，注意视频源必须为 1920*1080P@60，下图为设置分辨率步骤，下载程序，可以看到显示器显示与视频源一致的图像。

11. DDR4 读写实验例程

11.1. 实验简介

实验目的:

PT2G390H 开发板集成四颗 4Gbit（512MB）DDR4 芯片,型号为 MT40A512M16LY-062EIT:E。DDR3 的总线宽度共为 32bit。DDR4SDRAM 的最高数据速率 1866Mbps。本实验生成 DDR4 IP 官方例程,实现 DDR4 的读写控制,了解其工作原理和用户接口。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

11.2. DDR4 控制器简介

PT2G390H 为用户提供一套完整的 DDR memory 控制器解决方案，配置方式比较灵活，采用软核实现 DDR memory 的控制，有如下特点:

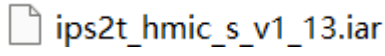
- 支持 DDR3、DDR4、DDR4 乒乓 PHY;
- 支持最大数据位宽 72bit（乒乓 PHY 最大 32bit）;
- 用户接口：AXI4 总线接口、APB 总线接口;
- 支持可配低功耗模式：Self-Refresh 和 Power Down;
- 支持 DDR3 的最高数据速率达到 1866Mbps;
- 支持 DDR4 的最高数据速率达到 2000Mbps;
- Burst Length 8 和单 Rank;
- PHY 可以单独使用。

11.3. 实验设计

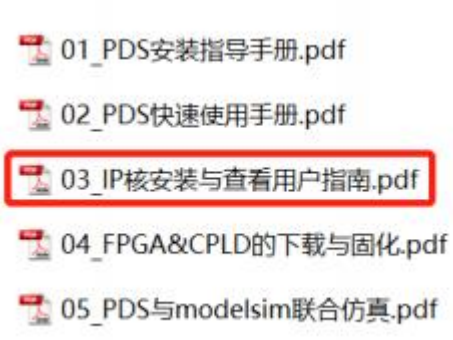
11.3.1. 安装 DDR4 IP 核

PDS 安装后，需手动添加 DDR4IP，请按以下步骤完成:

（1）DDR4IP 文件： ips2t_hmic_s_v1_13.iar

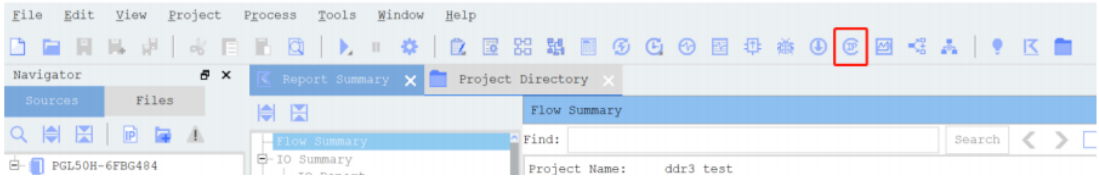


（2）IP 安装步骤：IP 核安装与查看用户指南.pdf

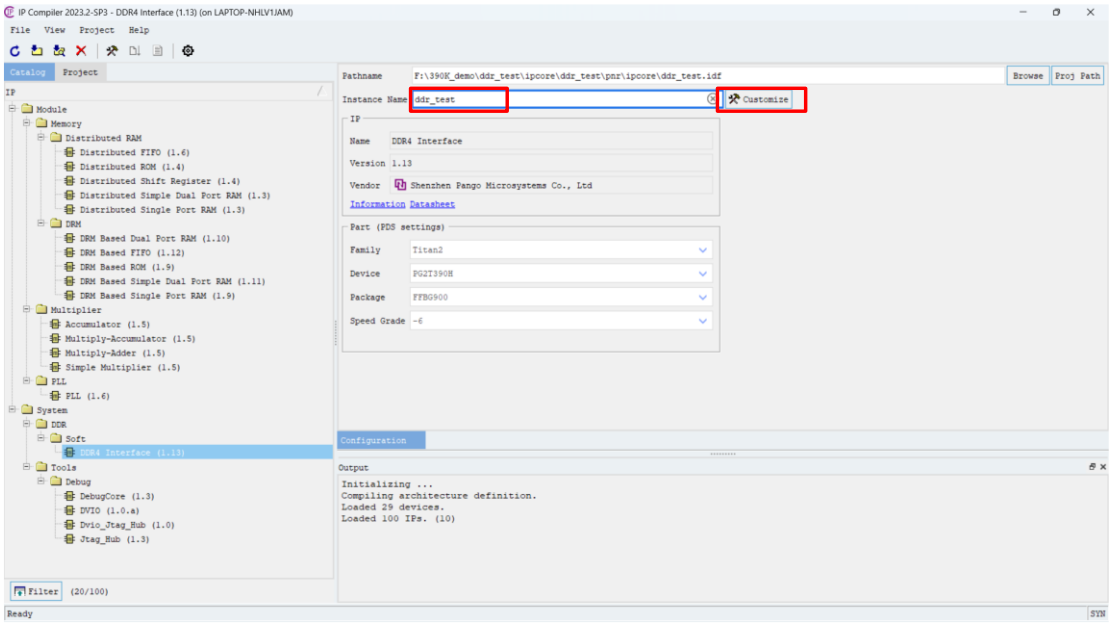


11.3.2. DDR4 读写 Example 工程

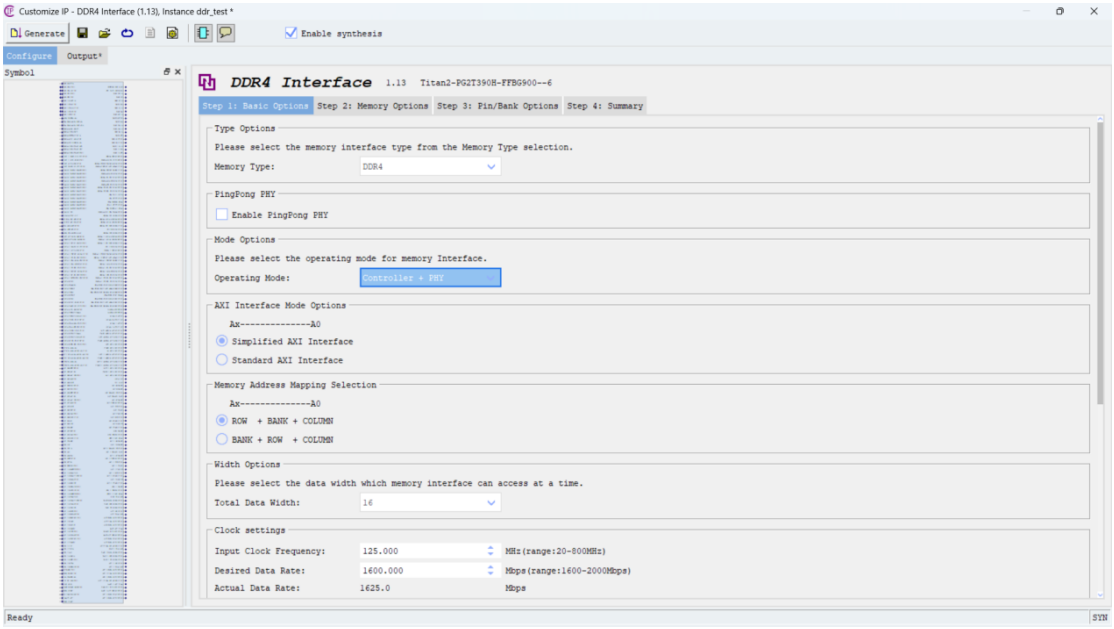
1.打开 PDS 软件，新建工程 ddr_test，点开如下图标，打开 IPCompiler；



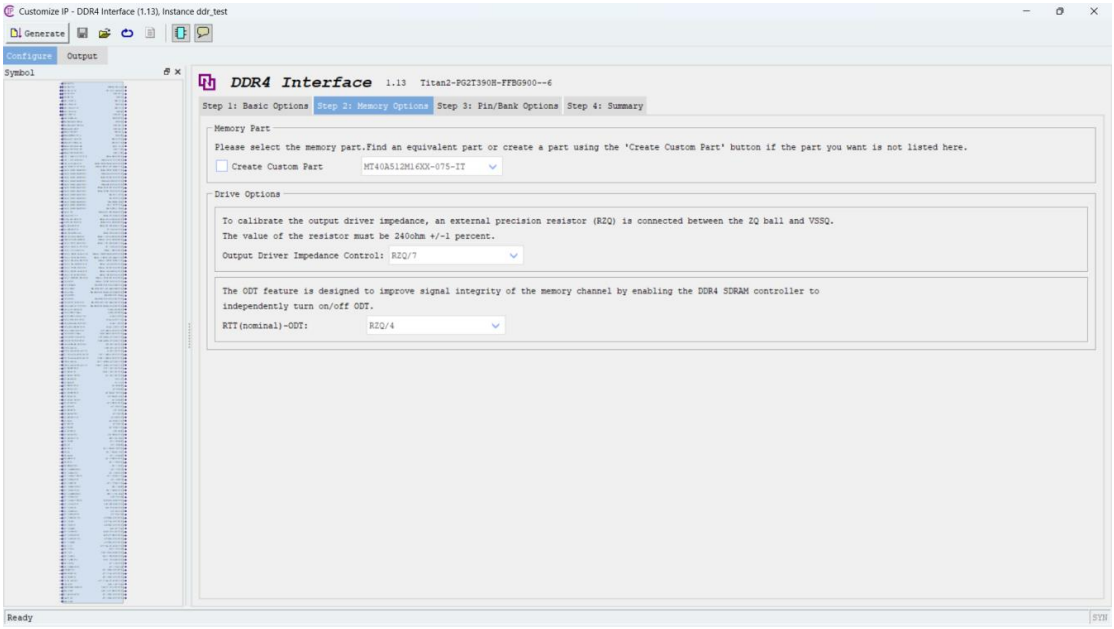
2.选择 DDR4IP，取名，然后点击 Customize；



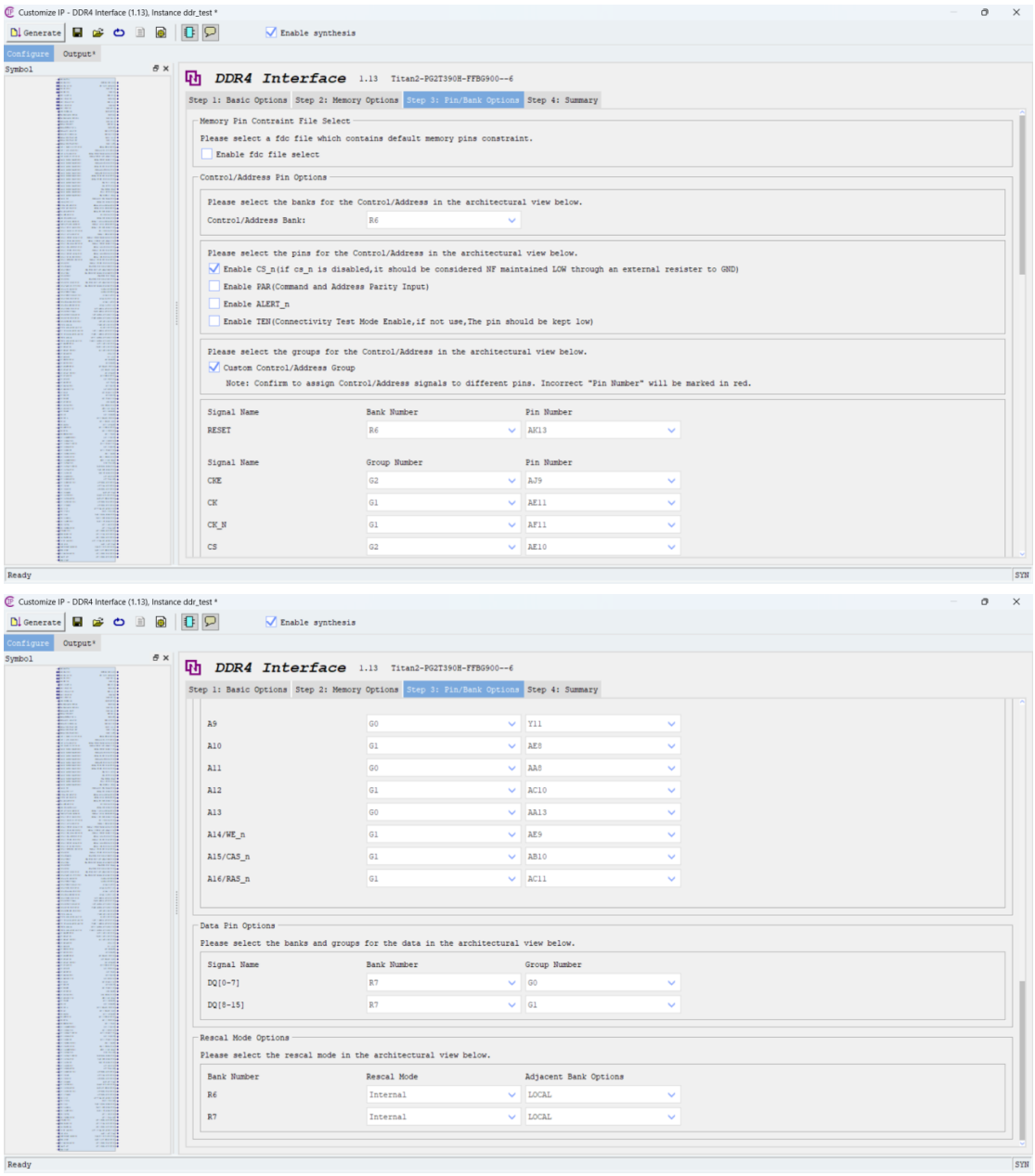
3.在 DDR 设置界面中 Step1 按照如下设置：



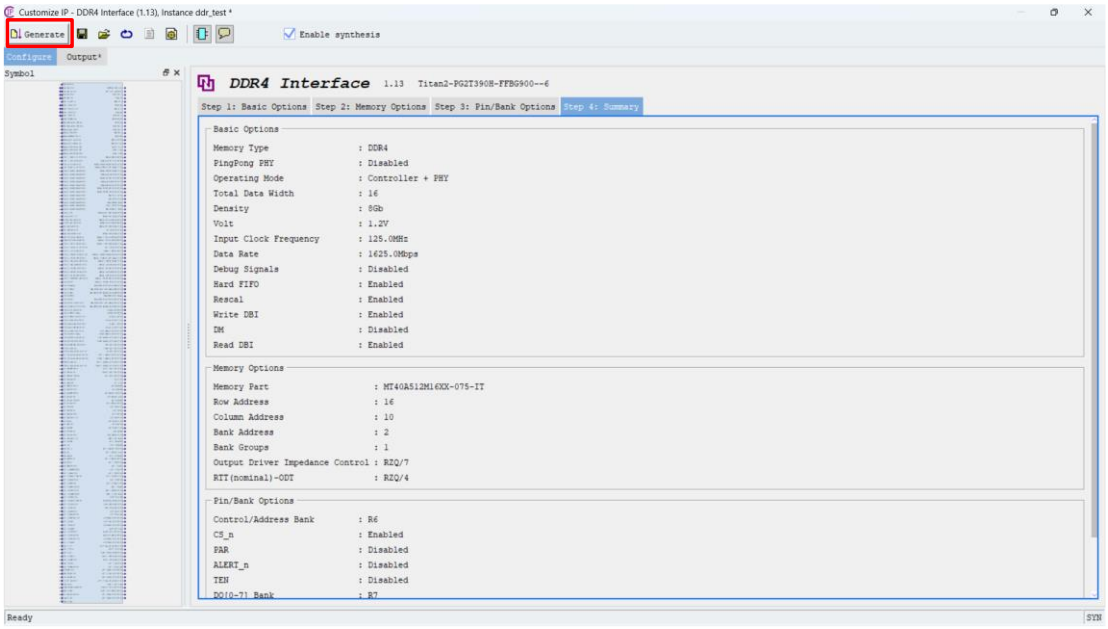
4.Step2 按照如下设置：



5.Step3 按照如下设置，勾选 CustomControl/AddressGroup，管脚约束参考原理图：



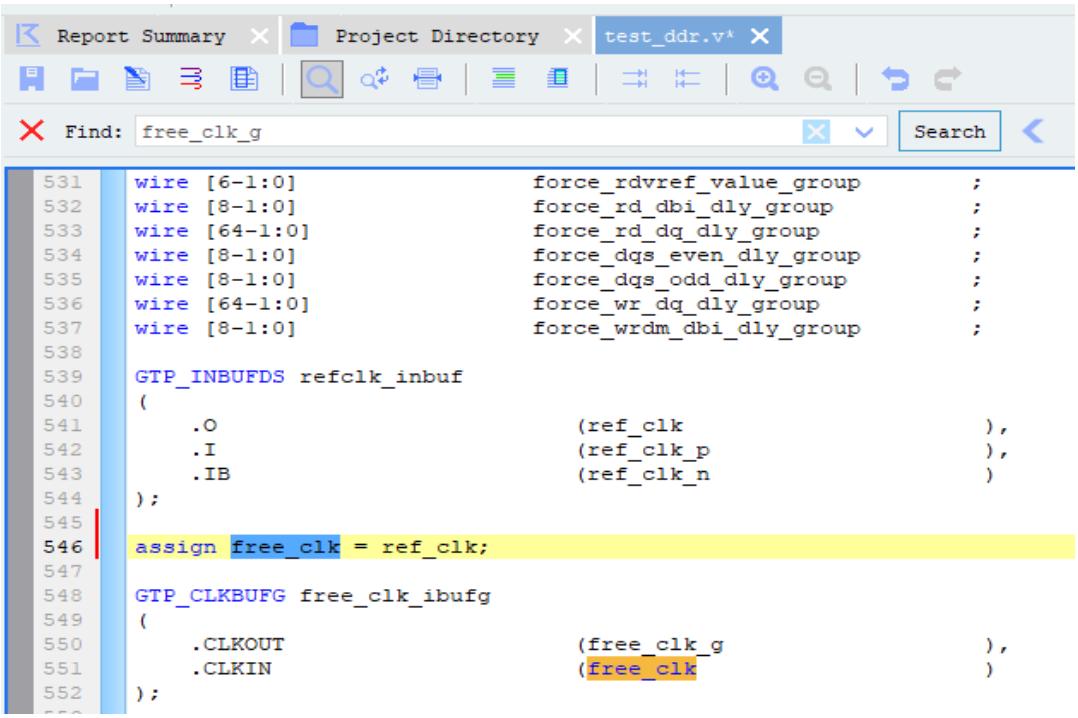
6.Step4 为概要，点击 Generate 可生成 DDR4IP；



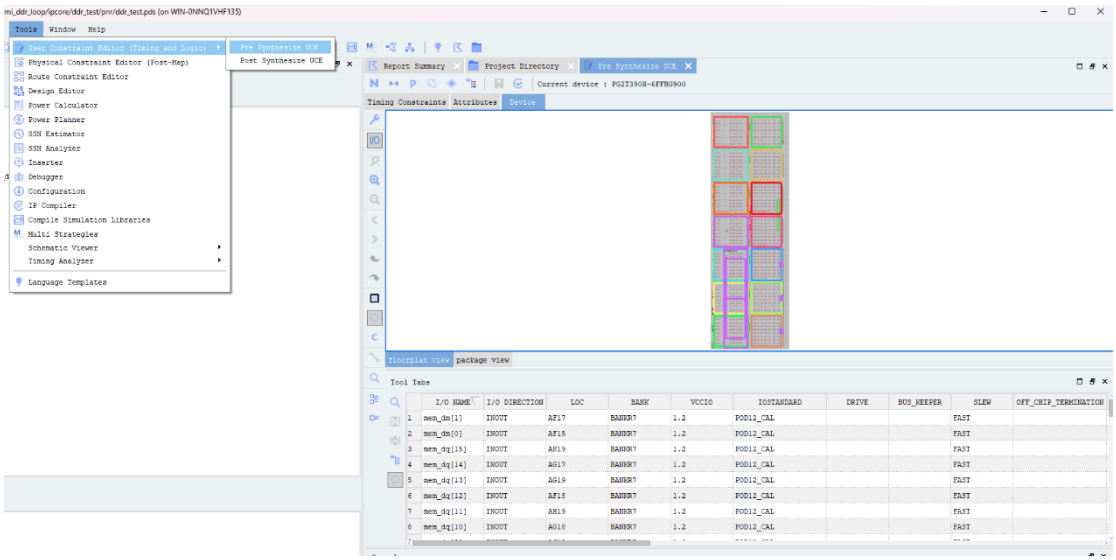
7.关闭本工程，在本工程路径下打开 Example 工程：ddr_test\ipcore\ddr_test\pnr

8.打开顶层文件，free_clk、ref_clk 可使用同一时钟源：

```
33  )(
34  input                                   ref_clk_p                ,
35  input                                   ref_clk_n                ,
36  //input                                   free_clk                 ,
37  input                                   rst_board                ,
38  output                                  pll_lock                 ,
39  output                                  ddrphy_cpd_lock          ,
40  output                                  ddr_init_done            ,
41  //uart
```



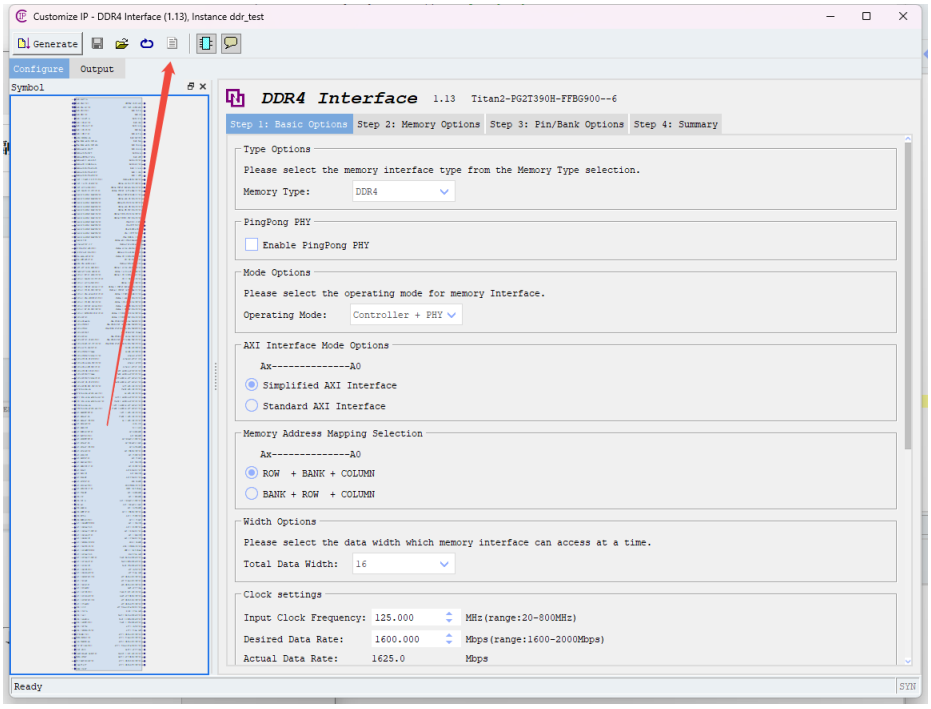
9.对“Step3 已做管脚约束”外的其他管脚，对照原理图使用 UCE 工具进行修改：



10.以下管脚可约束在 LED，方便观察实验现象；

信号	LED	PT2G390HPin
err_flag_led	LED0	V30
heart_beat_led	LED1	V29
ddr_init_done	LED2	V20
pll_lock	LED3	V19

11.在第一个工程时创建 ddr ip 时可按以下方式查看 IP 核的用户指南，了解 Example 模块组成；



11.4. 实验现象

注：例程位置：Demo\07_ddr_test\ipcore\ddr_test\pnr

信号名称	参考说明	LED 编号
ddr_init_done	初始化标志	1
err_flag_led	数据检测错误信号	2
heart_beat_led	心跳信号	3
pll_lock	Pll 锁定指示	4

打开约束文件，修改上面的四个信号名的引脚约束，重新约束到开发板上的 LED 灯。下载程序，可以看到 LED0 常灭，LED1 闪烁，LED2 常亮，LED3 常亮。

12. HDMI 回环实验

12.1. 实验简介

实验目的:

完成 HDMI 回环实验

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

12.2. 实验原理

12.2.1. HDMI 编解码介绍

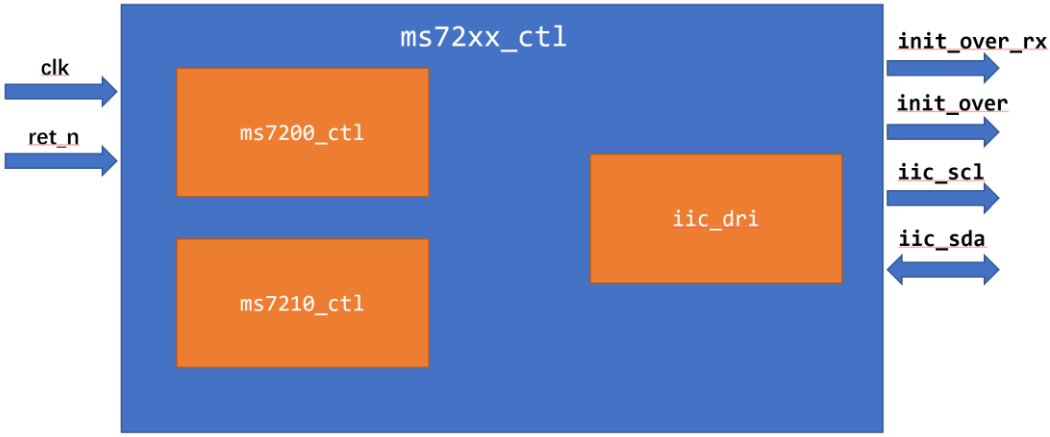
HDMI 是新一代的多媒体接口标准，英文全称是 High-DefinitionMultimediaInterface，即高清多媒体接口。它能够同时传输视频和音频，简化了设备的接口和连线；同时提供了更高的数据传输带宽，可以传输无压缩的数字音频及高分辨率视频信号。HDMI1.0 版本于 2002 年发布，最高数据传输速度为 5Gbps；HDMI2.0 版本于 2013 年推出的，2.0 理论传输速度能达到 18Gbit/s，实际传输速度能达到 14.4Gbit/s；而 2017 年发布的 HDMI2.1 标准的理论带宽可达 48Gbps，实际速度也能达到 42.6Gbit/s。

在 HDMI 接口出现之前，被广泛应用的是 VGA 接口。VGA 的全称是 VideoGraphicsArray，即视频图形阵列，是一个使用模拟信号进行视频传输的标准。相信大家已经很熟悉了。VGA 接口除信号容易受到干扰外，其体积也较大，由于 VGA 接口传输的是模拟信号，其信号容易受到干扰，因此 VGA 在高分辨率下字体容易虚，信号线长的话，图像有拖尾现象。因此 VGA 接口已逐渐退出舞台，一些显示器也不再带有 VGA 接口，在数字设备高度发展的今天，取而代之的是 HDMI 接口和 DP（DisplayPort）接口等。

DVI 和 HDMI 接口协议在物理层使用 TMDS 标准传输音视频数据。TMDS（Transition Minimized Differe-ntial Signaling，最小化传输差分信号）是美国 SiliconImage 公司开发的一项高速数据传输技术，TMDS 广泛应用于 HDMI 和 DVI 接口中。它采用差分信号传输方式，能有效降低电磁干扰，并通过编码减少信号转换次数以提高传输效率。在 HDMI 中，TMDS 由三对数据通道和一对时钟通道组成，分别传输视频的 RGB 分量和同步时钟信号。这种技术支持较高的带宽需求，确保高分辨率视频和音频信号的稳定传输，是数字视频信号传输的重要基础技术。

使用 fpga 实现 TMDS 编码传输比较有难度，不过 PT2G390H 板卡板载了 HDMI 编码解码芯片，其中 HDMI 输入接口采用宏晶微 MS7200HMDI 接收芯片，HDMI 输出接口采用宏晶微 MS7210HMDI 发送芯片。芯片兼容 HDMI1.4b 及以下标准视频的 3D 传输格式，最高分辨率高达 4K@30Hz，最高采样率达到 300MHz，支持 YUV 和 RGB 之间的色彩空间转换，数字接口支持 YUV 及 RGB 格式，MS7200 和 MS7210 的 IIC 配置接口与 FPGA 的 IO 相连，我们只需要配置这两个芯片即可使用它们对输入输出的视频进行解码和编码。

芯片配置模块层次框图如下：



12.2.2. ddr4 介绍

DDR4SDRAM（Double-Data-RateFourSynchronousDynamicRandomAccessMemory）是 DDRSDRAM 的第四代产品，相较于 DDR2、DDR3，DDR4 有更高的运行性能与更低的电压。DDRSDRAM 是在 SDRAM 技术的基础上发展改进而来的，同 SDRAM 相比，DDRSDRAM 的最大特点是双沿触发，即在时钟的上升沿和下降沿都能进行数据采集和发送，同样的工作时钟，DDRSDRAM 的读写速度可以比传统的 SDRAM 快一倍。

本次实验使用的 DDR4 芯片是镁光的 MT40A512M16LY-062EIT:E，由于 DDR4 的时序非常复杂，如果直接编写 DDR4 的控制器代码，那么工作量是非常大的，且性能难以得到保证。我们使用紫光提供的 ddr IP 来生成 ddr4 控制器，以简化 ddr4 的使用，我们只需要通过操控 ddr IP，就可以间接的控制板载 ddr4。

HMIC_SIP 是紫光同创推出的一款 DDR4 IP，基于 FPGA 产品 HPIO 资源实现 SDRAM 读写，兼容 DDR4，可通过公司 PDS(PangoDesignSuite)套件中的 IPC(IPCompiler)工具完成 IP 模块的配置和生成。

HMIC_SIP 产品的主要特性如下：

支持 DDR3、DDR4、DDR4 乒乓 PHY；

支持最大数据位宽 72bit（乒乓 PHY 最大 32bit）；

用户接口：AXI4 总线接口、APB 总线接口；

支持可配低功耗模式：Self-Refresh 和 PowerDown；

支持 DDR3 的最高数据速率达到 1866Mbps；

支持 DDR4 的最高数据速率达到 2000Mbps；

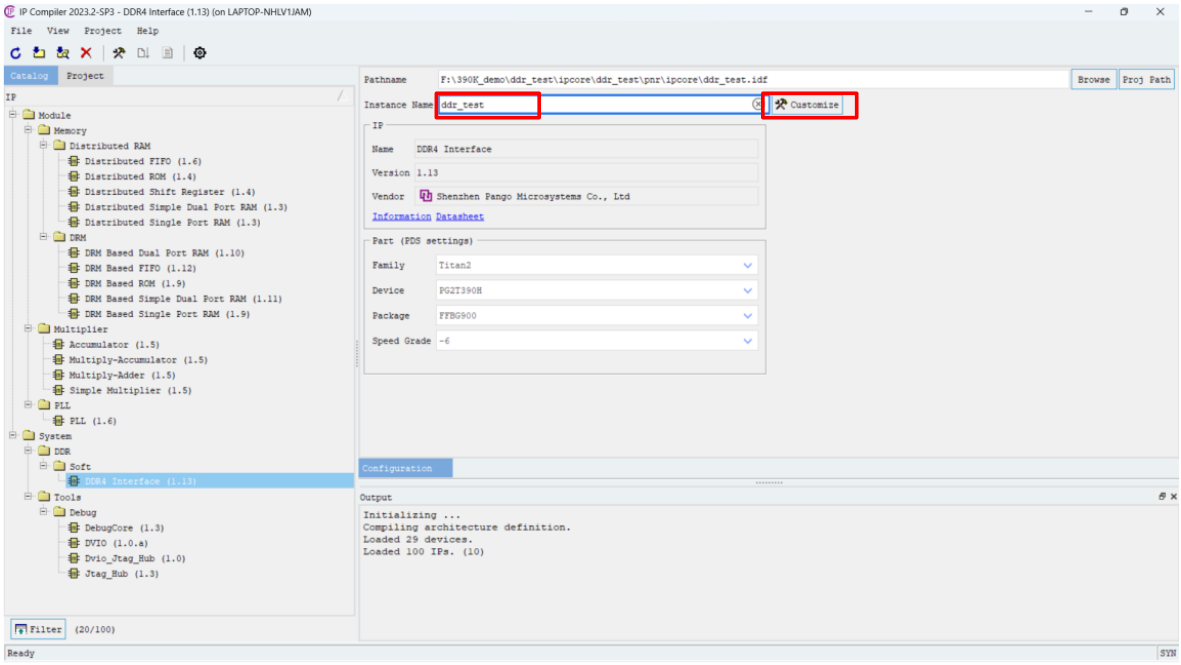
BurstLength8 和单 Rank；

PHY 可以单独使用。

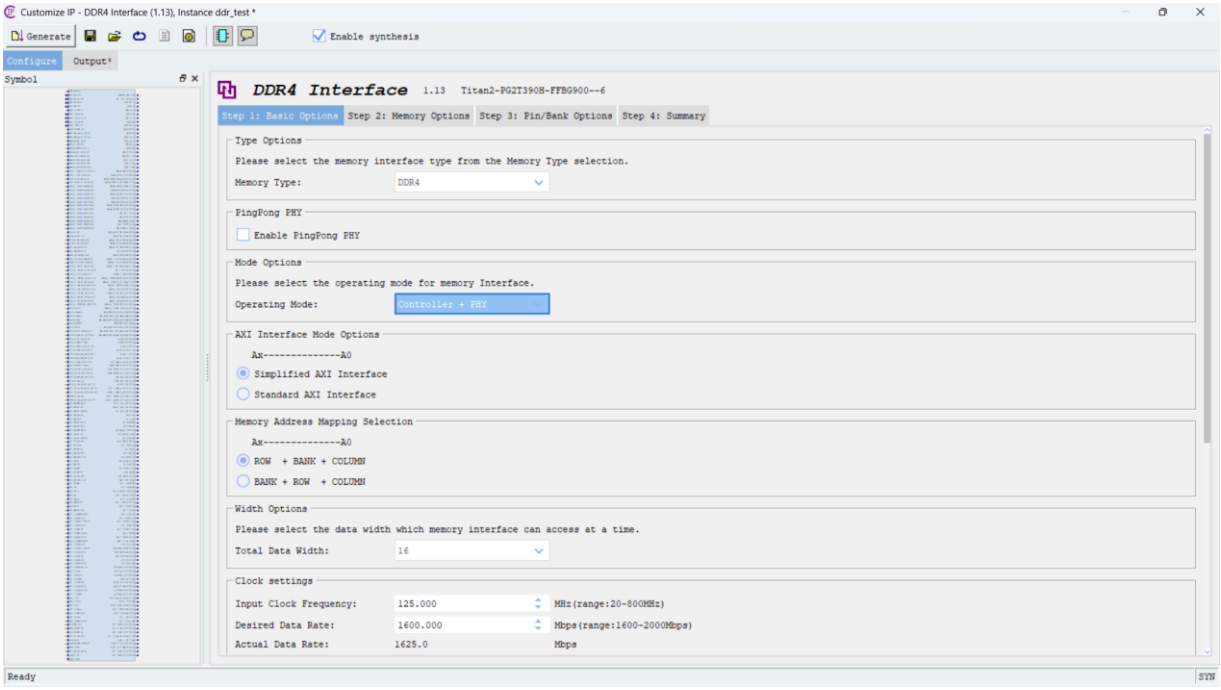
接下来我们着手生成 ddr3IP：

1.打开 PDS 软件，新建工程 ddr_test，点开如下图标，打开 IPCompiler；

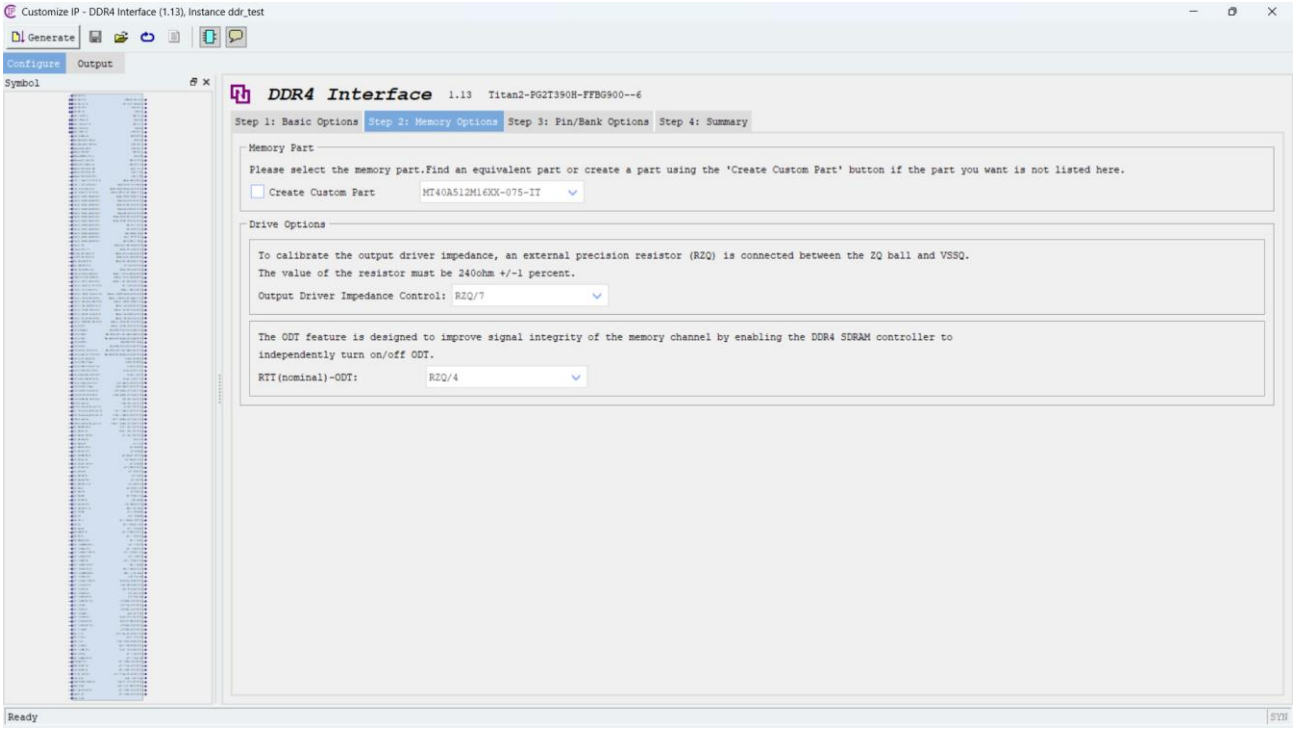
2.选择 DDRIP，取名，然后点击 Customize；



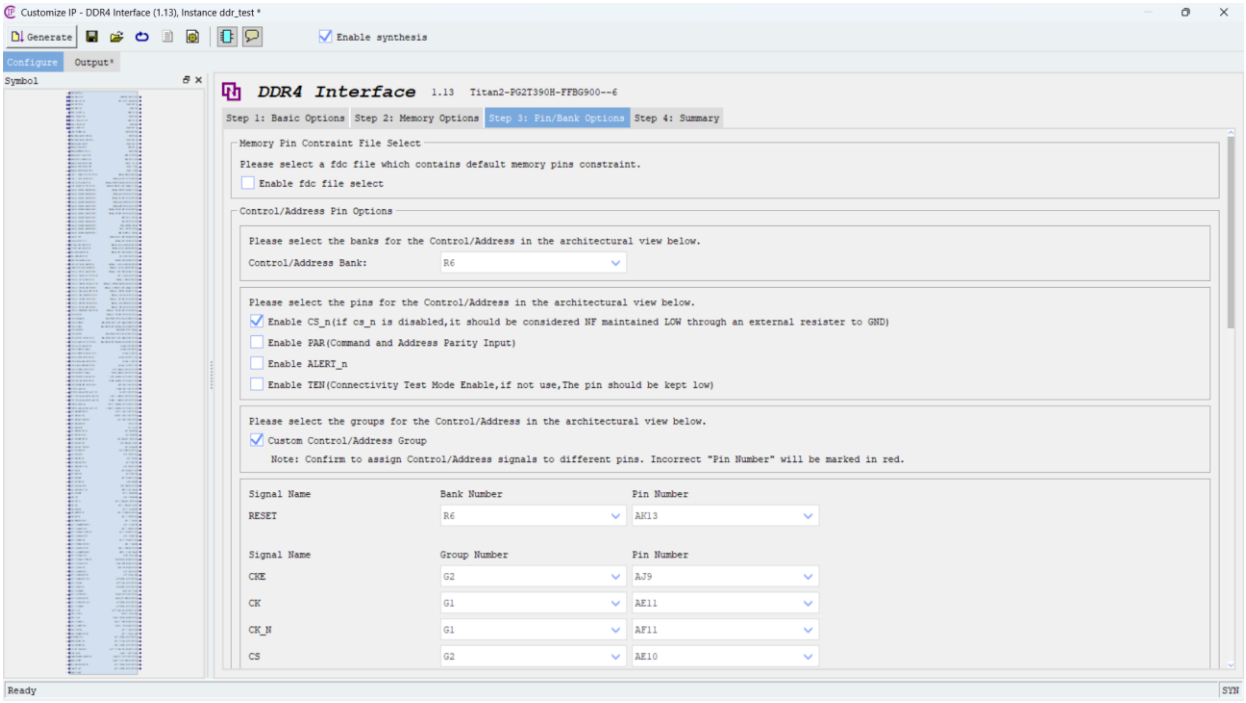
3.在 DDR4 设置界面中 Step1 参考如下设置：

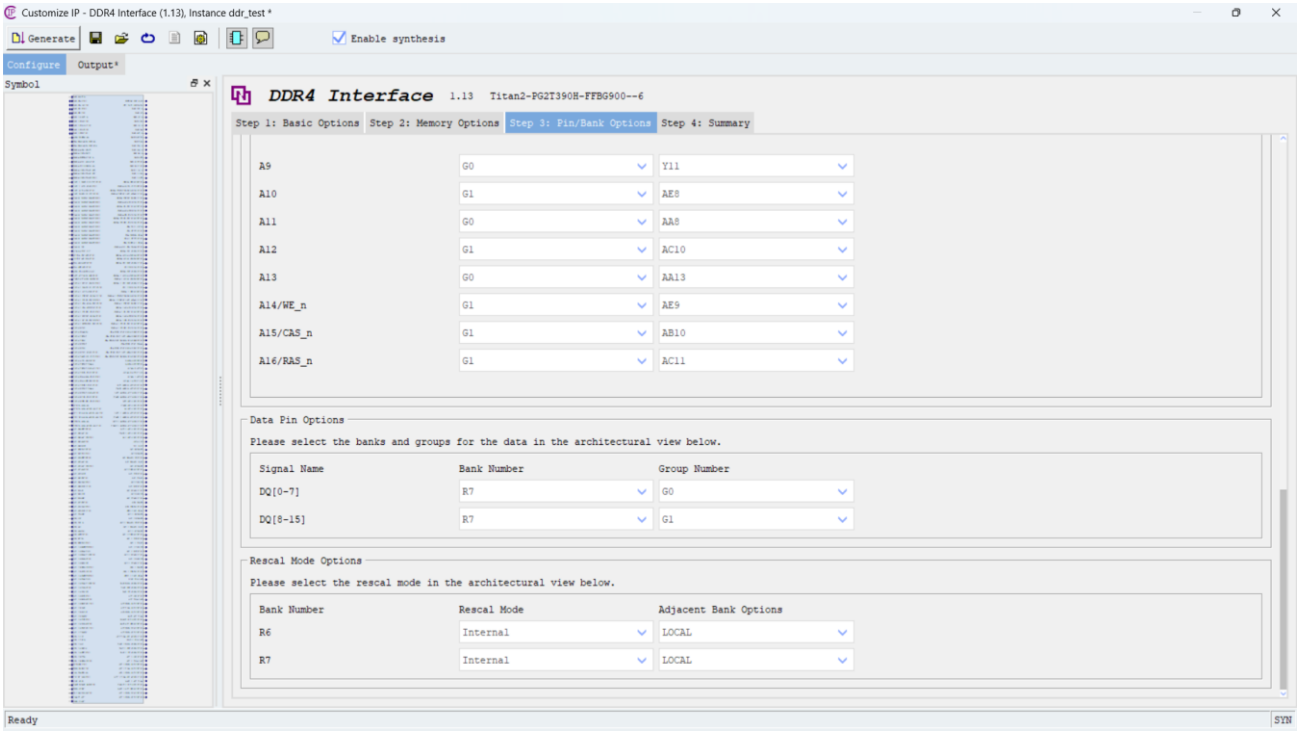


4.Step2 参考如下设置：

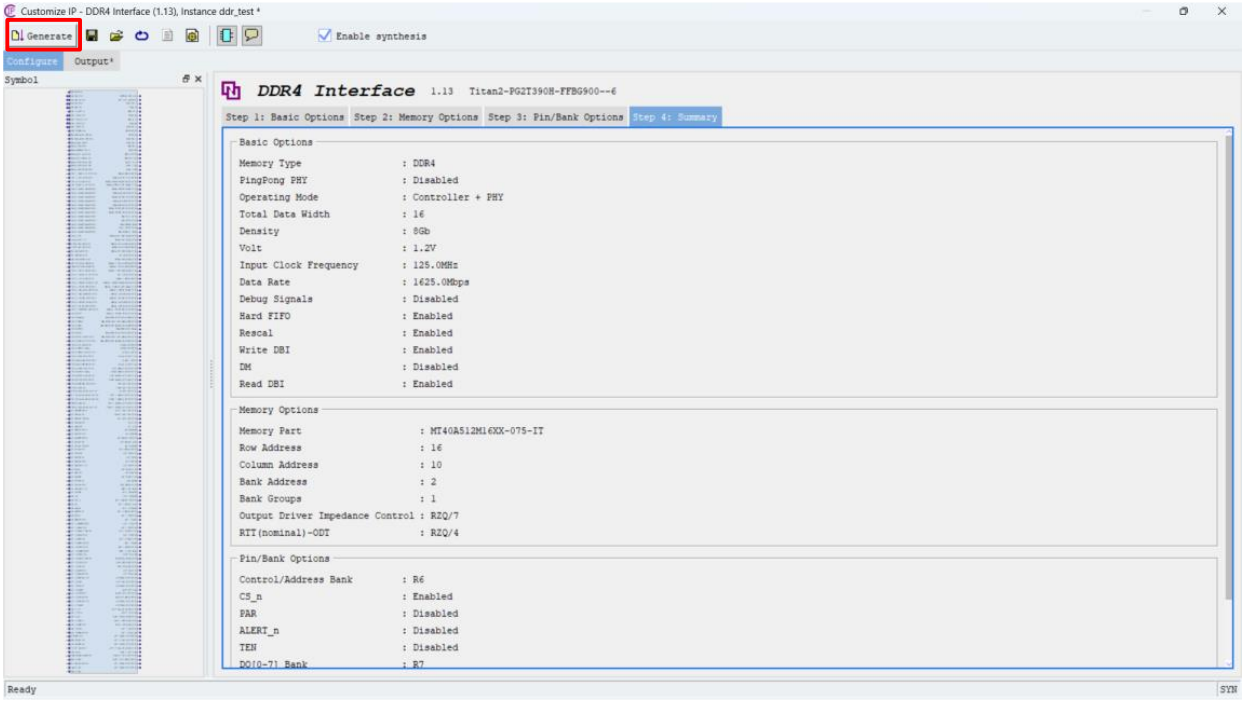


5.Step3 参考如下设置，勾选 CustomControl/AddressGroup，管脚约束参考原理图：





6.Step4 为概要，点击 Generate 可生成 DDR4 IP；



关于 ddrIP 详细信息请参考官方 IP 手册。

12.2.3. axi 接口介绍

AXI 的英文全称是 Advancede Xtensible Interface，即高级可扩展接口，它是 ARM 公司所提出的 AMBA（Advanced Microcontroller Bus Architecture）协议的一部分。紫光 ddrIP 支持使用标准 AXI 接口或者简化的 AXI 接口进行数据交互。本次实验使用的是简化 AXI 接口的模式（SimplifiedAXI4）。其接口列表如下：

（1）写地址通道

端口	I/O	位宽	有效值	描述
axi_awaddr	I	CTRL_ADDR_WIDTH	-	AXI 写地址。
axi_awuser_ap	I	1	高电平	AXI 写并自动 precharge。
axi_awuser_id	I	4	-	AXI 写地址 ID。
axi_awlen	I	4	-	AXI 写突发长度。
axi_awready	O	1	高电平	AXI 写地址 ready。
axi_awvalid	I	1	高电平	AXI 写地址 valid。

注：“-”表示无该项参数。

(2)读地址通道

端口	I/O	位宽	有效值	描述
axi_araddr	I	CTRL_ADDR_WIDTH	-	AXI 读地址。
axi_aruser_ap	I	1	高电平	AXI 读并自动 precharge。
axi_aruser_id	I	4	-	AXI 读地址 ID。
axi_arlen	I	4	-	AXI 读突发长度。
axi_arready	O	1	高电平	AXI 读地址 ready。
axi_arvalid	I	1	高电平	AXI 读地址 valid。

注：“-”表示无该项参数。

（3）写数据通道

端口	I/O	位宽	有效值	描述
axi_wdata	I	DQ_WIDTH*8	-	AXI 写数据。
axi_wstrb	I	DQ_WIDTH*8/8	高电平	AXI 写数据 strobes。
axi_wready	O	1	高电平	AXI 写数据 ready。
axi_wusero_id	O	4	-	AXI 写数据 ID。
axi_wusero_last	O	1	高电平	AXI 写数据 last。

注：“-”表示无该项参数。

(4)读数据通道

端口	I/O	位宽	有效值	描述
axi_rid	O	4	-	AXI 读数据 ID。
axi_rlast	O	1	高电平	AXI 读数据 last 信号。
axi_rvalid	O	1	高电平	AXI 读数据 valid。
axi_rdata	O	DQ_WIDTH*8	-	AXI 读数据。

注：“-”表示无该项参数。

其中各个通道读写时序要求如下：

(1)写地址通道时序

写地址通道包含的信号：axi_awready，axi_awvalid，axi_awaddr，axi_awuser_ap，axi_awuser_id，axi_awlen。典型时序如图 2-24 所示。

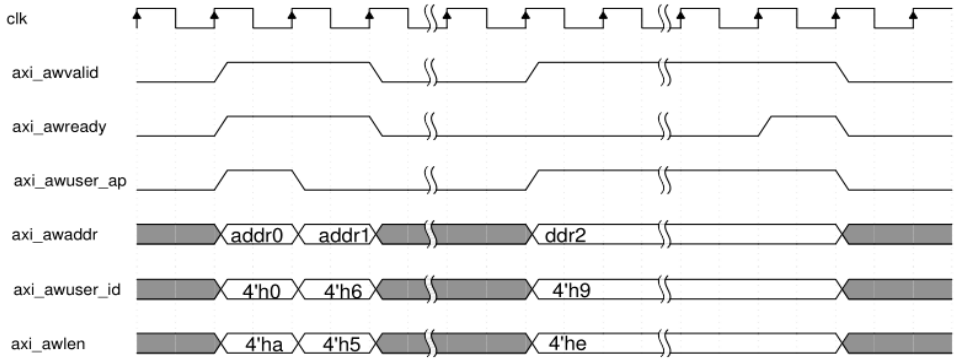


图 2-18 写地址典型时序

- 一次握手完成条件：axi_awready 与 axi_awvalid 同时有效。
- 包长由 axi_awlen 控制，包长为 axi_awlen 的值加 1。
- 握手过程：从 axi_awvalid 有效的时钟上升沿开始，axi_awaddr，axi_awuser_ap，axi_awuser_id，axi_awlen 需保持不变直到握手完成后释放。

(2)读地址通道时序

读地址通道包含的信号：axi_arready，axi_arvalid，axi_araddr，axi_aruser_ap，axi_aruser_id，axi_arlen。

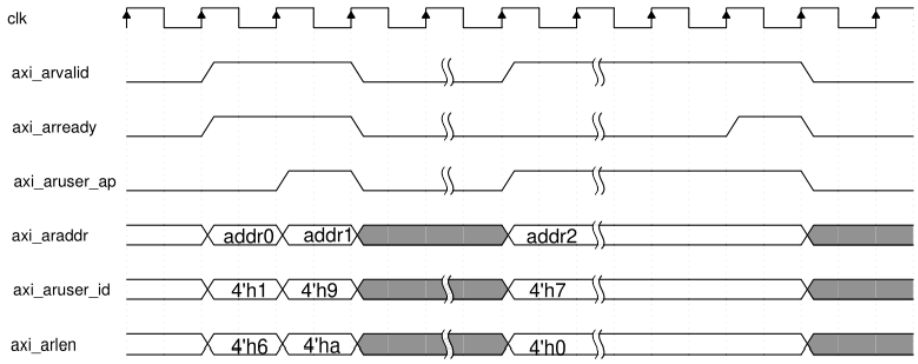


图 2-19 读地址典型时序

- 一次握手完成条件：axi_arready 与 axi_arvalid 同时有效。
- 包长由 axi_arlen 控制，包长为 axi_arlen 的值加 1。
- 握手过程：从 axi_arvalid 有效的时钟上升沿开始，axi_araddr，axi_aruser_ap，axi_aruser_id，axi_arlen 需保持不变直到握手完成后释放。

(3)写数据通道时序

写数据通道包含的信号：axi_wready，axi_wusero_id，axi_wusero_last，axi_wdata 和 axi_wstrb。典型时序如图 2-26 所示。

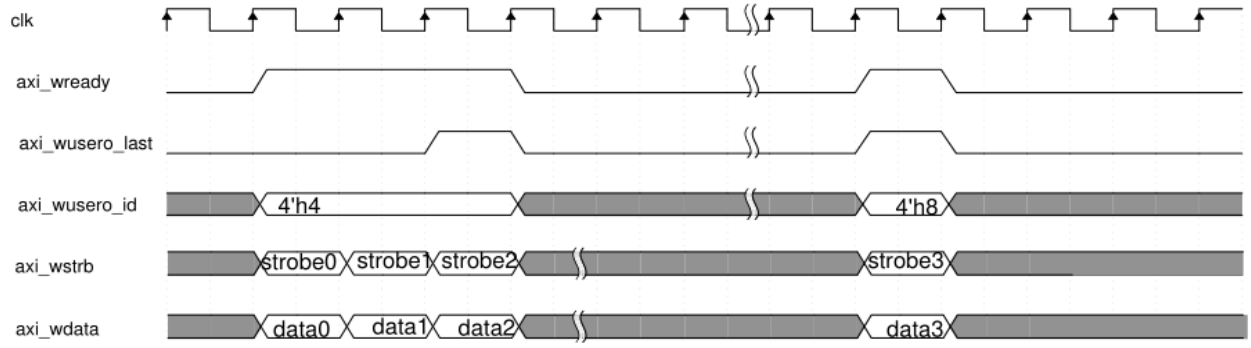


图 2-20 写数据典型时序

- 传输有效指示：axi_wready 有效。
- 一次传输结束指示：axi_wusero_last 有效。
- 传输过程中：axi_wready，axi_wusero_id 和 axi_wusero_last 同步接收，axi_wdata 和 axi_wstrb 同步发送。

(4)读数据通道时序

读数据通道包含的信号：axi_rdata, axi_rid, axi_rlast 和 axi_rvalid。典型时序如图 2-27 所示。

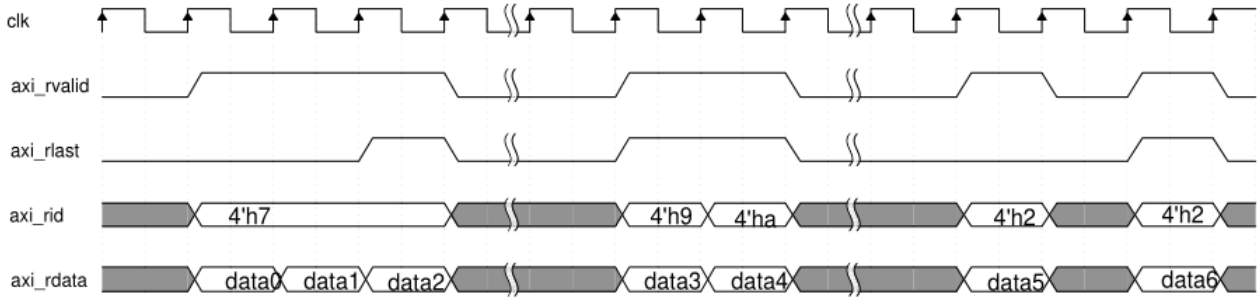


图 2-21 读数据典型时序

- 传输有效指示：axi_rvalid 有效。
- 一次传输结束指示：axi_rlast 有效。

更多详细信息请打开 IP 用户使用手册进行查看。

12.3. 接口列表

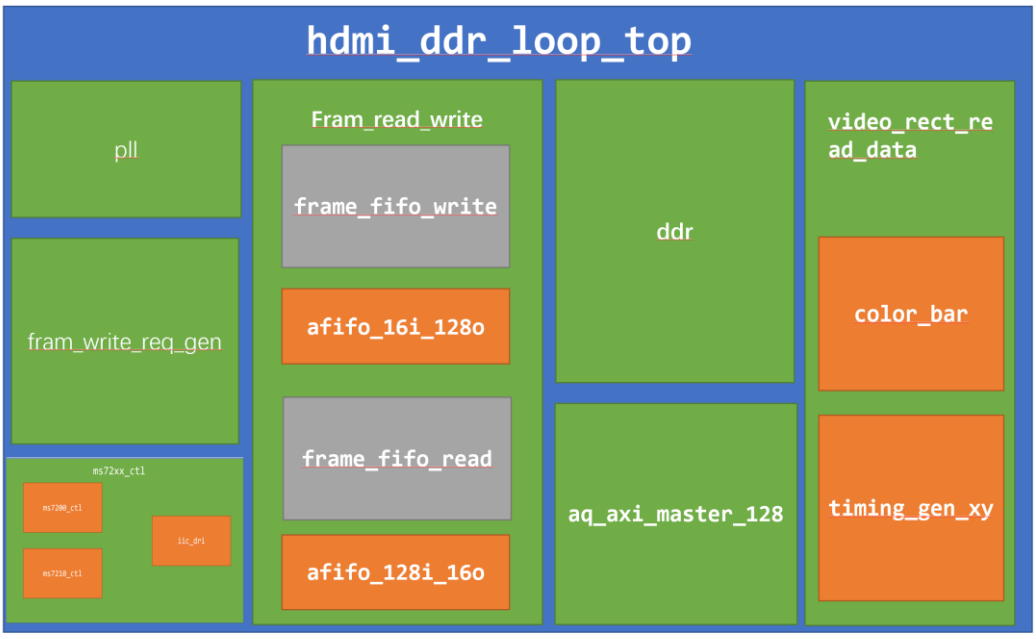
信号名称	输入/输出说明	端口位宽	端口说明
ref_clk_p	输入	1 位	差分参考时钟正端
ref_clk_n	输入	1 位	差分参考时钟负端
sys_clk	输入	1 位	系统时钟（50MHz）
clk_p	输入	1 位	差分时钟正端
clk_n	输入	1 位	差分时钟负端
rst_n	输入	1 位	复位信号（低电平有效）
mem_rst_n	输出	1 位	DDR 内存复位信号
mem_ck	输出	1 位	DDR 内存时钟
mem_ck_n	输出	1 位	DDR 内存时钟反相
mem_cke	输出	1 位	DDR 内存时钟使能
mem_act_n	输出	1 位	DDR 激活命令
mem_odt	输出	1 位	DDR ODT
mem_bg	输出	1 位	DDR bank group
mem_odt	输出	1 位	On-Die 终端使能
mem_a	输出	13 位	行地址总线
mem_ba	输出	3 位	Bank 地址总线
mem_dqs	输入/输出	2 位	数据选通信号
mem_dqs_n	输入/输出	2 位	数据选通信号反相
mem_dq	输入/输出	16 位	数据总线
mem_dm	输出	2 位	数据掩码信号

heart_beat_led	输出	1 位	心跳指示灯
ddr_init_done	输出	1 位	DDR 初始化完成指示
rx_rst_n	输出	1 位	接收复位信号（低电平有效）
rstn_out	输出	1 位	外部复位信号（低电平有效）
iic_tx_scl	输出	1 位	I2C 传输时钟线
iic_tx_sda	输入/输出	1 位	I2C 传输数据线
hdmi_int_led	输出	1 位	HDMI 输出初始化完成指示灯
pll_lock	输出	1 位	PLL 锁定指示
mem_led	输出	1 位	内存指示灯
ddrphy_cpd_lock	输出	1 位	DDR 物理层锁定指示
pixclk_in	输入	1 位	HDMI 输入像素时钟
vs_in	输入	1 位	HDMI 输入垂直同步信号
hs_in	输入	1 位	HDMI 输入水平同步信号
de_in	输入	1 位	HDMI 输入数据使能
r_in	输入	8 位	HDMI 输入红色数据
g_in	输入	8 位	HDMI 输入绿色数据
b_in	输入	8 位	HDMI 输入蓝色数据
rx_init_done	输出	1 位	接收初始化完成指示
pix_clk	输出	1 位	HDMI 输出像素时钟

vs_out	输出	1 位	HDMI 输出垂直同步信号
hs_out	输出	1 位	HDMI 输出水平同步信号
de_out	输出	1 位	HDMI 输出数据使能
r_out	输出	8 位	HDMI 输出红色数据
g_out	输出	8 位	HDMI 输出绿色数据
b_out	输出	8 位	HDMI 输出蓝色数据

12.4. 工程说明

工程整体框图示意图如下：



工程结构大体说明如下：

Ms72xx_ctrl 模块配置完 HDMI 编码解码芯片之后，经过解码后的数据也就是输入的像素时钟信号，行、场同步，数据有效信号和图像 rgb 数据进入 fram_write_req_gen 模块，其功能是产生一帧数据的读请求。Fram_read_write 和 aq_axi_master_128 模块是密不可分的，Fram_read_write 负责处理读写请求，其内部例化了两个缓存 fifo，用来缓存一次突发读或者突发写的数据。本次 AXI 协议封装模式突发大小为 128bit，突发长度为 16。完成一次突发传输地址应该增加 16x16=256。当写 FIFO 数据满足一次突发写长

度后，通过 aq_axi_master_128 写入 ddr，当有读请求进入 Fram_read_write 模块时，通过 aq_axi_master_128 模块从 ddr 中突发读出数据。

Video_rect_read_data 模块负责产生读请求，其中的 color_bar 模块负责产生 vga 时序，输出彩条数据（这样如果 ddr 读写失败就会再屏幕上显示彩条图案），timing_gen_xy 负责产生图像 x、y 坐标像素计数信号，可以用于字符叠加显示等等，这里虽然没有用到，但还是将其保留。

12.4.1. 代码模块说明

由于工程较大，一些简单的模块相信读者能够自行研究，我们在这里介绍重要的模块

Frame_read_write 帧读写缓存控制模块

```
1.  `timescale 1ns/1ps
2.  module frame_read_write
3.  #
4.  (
5.      parameter                       MEM_DATA_BITS           = 16    ,
6.      parameter                       READ_DATA_BITS          = 16    ,
7.      parameter                       WRITE_DATA_BITS         = 16    ,
8.      parameter                       ADDR_BITS               = 24    ,
9.      parameter                       BUSRT_BITS              = 10    ,
10.     parameter                       BURST_SIZE              = 128
11. )
12. (
13.     input                           rst                     ,
14.     // external memory controller user interface clock
15.     input                           mem_clk                 ,
16.     // to external memory controller,send out a burst read request
17.     output                          rd_burst_req            ,
18.     // to external memory controller,data length of the burst read request, not bytes
19.     output          [BUSRT_BITS - 1:0]rd_burst_len          ,
20.     // to external memory controller,base address of the burst read request
21.     output          [ADDR_BITS - 1:0]rd_burst_addr          ,
22.     // from external memory controller,read data valid
23.     input                           rd_burst_data_valid     ,
24.     // from external memory controller,read request data
25.     input           [MEM_DATA_BITS - 1:0]rd_burst_data      ,
26.     // from external memory controller,burst read finish
27.     input                           rd_burst_finish         ,
28.     // data read module clock
29.     input                           read_clk                ,
30.     // data read module read request,keep '1' until read_req_ack = '1'
31.     input                           read_req                ,
32.     // data read module read request response
33.     output                          read_req_ack            ,
34.     // data read module read request finish
35.     output                          read_finish             ,
36.     // data read module read request base address 0, used when read_addr_index = 0
37.     input           [ADDR_BITS - 1:0]read_addr_0            ,
38.     // data read module read request base address 1, used when read_addr_index = 1
39.     input           [ADDR_BITS - 1:0]read_addr_1            ,
40.     // data read module read request base address 1, used when read_addr_index = 2
```

```
41.     input                [ADDR_BITS - 1:0]read_addr_2                  ,
42.     // data read module read request base address 1, used when read_addr_index = 3
43.     input                [ADDR_BITS - 1:0]read_addr_3                  ,
44.      // select valid base address from read_addr_0 read_addr_1 read_addr_2 read_addr_3
45.     input                [     1: 0]      read_addr_index              ,
46.     // data read module read request data length
47.     input                [ADDR_BITS - 1:0]read_len                     ,
48.     // data read module read request for one data, read_data valid next clock
49.     input                                 read_en                      ,
50.     // read data
51.     output               [READ_DATA_BITS   - 1:0]read_data             ,
52.     // to external memory controller,send out a burst write request
53.     output                                wr_burst_req                 ,
54.     // to external memory controller,data length of the burst write request, not bytes
55.     output               [BUSRT_BITS - 1:0]wr_burst_len                ,
56.     // to external memory controller,base address of the burst write request
57.     output               [ADDR_BITS - 1:0]wr_burst_addr                ,
58.     // from external memory controller,write data request ,before data 1 clock
59.     input                                 wr_burst_data_req            ,
60.     // to external memory controller,write data
61.     output               [MEM_DATA_BITS - 1:0]wr_burst_data            ,
62.     // from external memory controller,burst write finish
63.     input                                 wr_burst_finish              ,
64.     // data write module clock
65.     input                                 write_clk                    ,
66.     // data write module write request,keep '1' until read_req_ack = '1'
67.     input                                 write_req                    ,
68.     // data write module write request response
69.     output                                write_req_ack                ,
70.     // data write module write request finish
71.     output                                write_finish                 ,
72.     // data write module write request base address 0, used when write_addr_index = 0
73.     input                [ADDR_BITS - 1:0]write_addr_0                 ,
74.     // data write module write request base address 1, used when write_addr_index = 1
75.     input                [ADDR_BITS - 1:0]write_addr_1                 ,
76.     // data write module write request base address 1, used when write_addr_index = 2
77.     input                [ADDR_BITS - 1:0]write_addr_2                 ,
78.     // data write module write request base address 1, used when write_addr_index = 3
79.     input                [ADDR_BITS - 1:0]write_addr_3                 ,
80.     // select valid base address from write_addr_0 write_addr_1 write_addr_2 write_addr_3
81.     input                [     1: 0]      write_addr_index             ,
82.     // data write module write request data length
83.     input                [ADDR_BITS - 1:0]write_len                    ,
84.     // data write module write request for one data
85.     input                                 write_en                     ,
86.     input                [WRITE_DATA_BITS - 1:0]write_data             // write data
87. );
88.     wire                 [   15: 0]       wrusedw                      ;// write used words
89.     wire                 [   15: 0]       rdusedw                      ;// read used words
90.     wire                                  read_fifo_aclr               ;// fifo Asynchronous clear
91.     wire                                  write_fifo_aclr              ;// fifo Asynchronous clear
92. //instantiate an asynchronous FIFO
93. afifo_16i_128o write_buf(
94.     .wr_clk                               (write_clk                   ),
95.     .wr_rst                               (write_fifo_aclr             ),
96.     .wr_en                                (write_en                    ),
97.     .wr_data                              (write_data                  ),
98.     .wr_full                              (                            ),
```

```
99.     .wr_water_level         (                       ),
100.    .almost_full            (                       ),
101.    .rd_clk                 (mem_clk                ),
102.    .rd_rst                 (write_fifo_aclr        ),
103.    .rd_en                  (wr_burst_data_req      ),
104.    .rd_data                (wr_burst_data          ),
105.    .rd_empty               (                       ),
106.    .rd_water_level         (rdusedw[9:0]           ),
107.    .almost_empty           ()                      );
108.
109.
110. frame_fifo_write
111. #
112. (
113.    .MEM_DATA_BITS              (MEM_DATA_BITS          ),
114.    .ADDR_BITS                  (ADDR_BITS              ),
115.    .BUSRT_BITS                 (BUSRT_BITS             ),
116.    .BURST_SIZE                 (BURST_SIZE             )
117. )
118. frame_fifo_write_m0
119. (
120.    .rst                    (rst                    ),
121.    .mem_clk                (mem_clk                ),
122.    .wr_burst_req           (wr_burst_req           ),
123.    .wr_burst_len           (wr_burst_len           ),
124.    .wr_burst_addr          (wr_burst_addr          ),
125.    .wr_burst_data_req      (wr_burst_data_req      ),
126.    .wr_burst_finish        (wr_burst_finish        ),
127.    .write_req              (write_req              ),
128.    .write_req_ack          (write_req_ack          ),
129.    .write_finish           (write_finish           ),
130.    .write_addr_0           (write_addr_0           ),
131.    .write_addr_1           (write_addr_1           ),
132.    .write_addr_2           (write_addr_2           ),
133.    .write_addr_3           (write_addr_3           ),
134.    .write_addr_index       (write_addr_index       ),
135.    .write_len              (write_len              ),
136.        .fifo_aclr          (write_fifo_aclr        ),
137.        .rdusedw            ({6'b0, rdusedw[9:0]}   )
138.
139.    );
140.
141.    //instantiate an asynchronous FIFO
142.
143.    afifo_128i_16o read_buf (
144.        .wr_clk             (mem_clk                ),
145.        .wr_rst             (read_fifo_aclr         ),
146.        .wr_en              (rd_burst_data_valid    ),
147.        .wr_data            (rd_burst_data          ),
148.        .wr_full            (                       ),
149.        .wr_water_level     (wrusedw[9:0]           ),
150.        .almost_full        (                       ),
151.        .rd_clk             (read_clk               ),
152.        .rd_rst             (read_fifo_aclr         ),
153.        .rd_en              (read_en                ),
154.        .rd_data            (read_data              ),
155.        .rd_empty           (                       ),
156.        .rd_water_level     (                       ),
```

```
157.        .almost_empty                   (                       )
158.        );
159.
160.
161.
162.    frame_fifo_read
163.    #
164.    (
165.        .MEM_DATA_BITS                  (MEM_DATA_BITS          ),
166.        .ADDR_BITS                      (ADDR_BITS              ),
167.        .FIFO_DEPTH                     (128                    ),
168.        .BUSRT_BITS                     (BUSRT_BITS             ),
169.        .BURST_SIZE                     (BURST_SIZE             )
170.    )
171.    frame_fifo_read_m0
172.    (
173.        .rst                        (rst                    ),
174.        .mem_clk                    (mem_clk                ),
175.        .rd_burst_req               (rd_burst_req           ),
176.        .rd_burst_len               (rd_burst_len           ),
177.        .rd_burst_addr              (rd_burst_addr          ),
178.        .rd_burst_data_valid        (rd_burst_data_valid    ),
179.        .rd_burst_finish            (rd_burst_finish        ),
180.        .read_req                   (read_req               ),
181.        .read_req_ack               (read_req_ack           ),
182.        .read_finish                (read_finish            ),
183.        .read_addr_0                (read_addr_0            ),
184.        .read_addr_1                (read_addr_1            ),
185.        .read_addr_2                (read_addr_2            ),
186.        .read_addr_3                (read_addr_3            ),
187.        .read_addr_index            (read_addr_index        ),
188.        .read_len                   (read_len               ),
189.        .fifo_aclr                  (read_fifo_aclr         ),
190.        .wrusedw                    ({6'b0, wrusedw[9:0]}   )
191.    );
192.
193.    endmodule
```

此模块较为简单，其内部例化了两个 FIFO，分别用于数据的写入与读出缓存。frame_fifo_write 是写 FIFO 控制模块，frame_fifo_read 是读 FIFO 控制模块。需要注意的是模块设计之初在 ddr 中划分了四个空间，分别是 0~2073600、2073600~4147200、4147200~6220800、6220800~----；用户可在这些区间存储其他内容。例如缓存不同图像的不同帧等等。read_addr_index 用于指示存储地址从哪个区间开始，read_addr_index 的值取 0、1、2、3 分别对应四个存储区间。

frame_fifo_write 写 FIFO 控制模块：

```
1.  `timescale 1ns/1ps
2.  module frame_fifo_write
3.  #
4.  (
5.      parameter                   MEM_DATA_BITS           = 32    ,
6.      parameter                   ADDR_BITS               = 23    ,
7.      parameter                   BUSRT_BITS              = 10    ,
8.      parameter                   BURST_SIZE              = 128
```

```
9.  )
10. (
11.     input                                rst                          ,
12.     // external memory controller user interface clock
13.     input                                mem_clk                      ,
14.     // to external memory controller,send out a burst write request
15.     output reg                           wr_burst_req                 ,
16.     // to external memory controller,data length of the burst write request, not bytes
17.     output reg          [BUSRT_BITS - 1:0]wr_burst_len                ,
18.     // to external memory controller,base address of the burst write request
19.     output reg          [ADDR_BITS - 1:0]wr_burst_addr                ,
20.     // from external memory controller,write data request ,before data 1 clock
21.     input                                wr_burst_data_req            ,
22.     // from external memory controller,burst write finish
23.     input                                wr_burst_finish              ,
24.     // data write module write request,keep '1' until read_req_ack = '1'
25.     input                                write_req                    ,
26.     // data write module write request response
27.     output reg                           write_req_ack                ,
28.     // data write module write request finish
29.     output                               write_finish                 ,
30.     // data write module write request base address 0, used when write_addr_index = 0
31.     input               [ADDR_BITS - 1:0]write_addr_0                 ,
32.     // data write module write request base address 1, used when write_addr_index = 1
33.     input               [ADDR_BITS - 1:0]write_addr_1                 ,
34.     // data write module write request base address 1, used when write_addr_index = 2
35.     input               [ADDR_BITS - 1:0]write_addr_2                 ,
36.     // data write module write request base address 1, used when write_addr_index = 3
37.     input               [ADDR_BITS - 1:0]write_addr_3                 ,
38.     // select valid base address from write_addr_0 write_addr_1 write_addr_2 write_addr_3
39.     input               [     1: 0]      write_addr_index             ,
40.
41.     input               [ADDR_BITS - 1:0]write_len    ,// data write module write request data length
42.     output reg                           fifo_aclr                    ,// to fifo asynchronous clear
43.     input               [   15: 0]       rdusedw                      // from fifo read used words
44. );
45.     //256 bit '1'    you can use ONE[n-1:0] for n bit '1'
46.     localparam                           ONE                 = 256'd1;
47.     //256 bit '0'
48.     localparam                           ZERO                = 256'd0;
49. //write state machine code
50.     localparam                           S_IDLE              = 0       ; //idle state,waiting for write
51.     localparam                           S_ACK               = 1       ;   //written request response
52.
53.      //check the FIFO status, ensure that there is enough space to burst write
54.     localparam                           S_CHECK_FIFO        = 2       ;
55.     //begin a burst write
56.     localparam                           S_WRITE_BURST       = 3       ;
57.      //a burst write complete
58.     localparam                           S_WRITE_BURST_END   = 4       ;
59.     //a frame of data is written to complete
60.     localparam                           S_END               = 5       ;
61.
62.     //asynchronous write request, synchronize to 'mem_clk' clock domain,first beat
63.     reg                                  write_req_d0                 ;
64.     //the second
65.     reg                                  write_req_d1                 ;
66.     //third,Why do you need 3 ? Here's the design habit
```

```
67.     reg                                 write_req_d2                ;
68.     //asynchronous write_len(write data length), synchronize to 'mem_clk' clock domain first
69.     reg             [ADDR_BITS - 1:0]write_len_d0                ;
70.     reg             [ADDR_BITS - 1:0]write_len_d1                ;//second
71.     reg             [ADDR_BITS - 1:0]write_len_latch             ;//lock write data length
72.     reg             [ADDR_BITS - 1:0]write_cnt                   ;//write data counter
73.     reg             [    1: 0]      write_addr_index_d0         ;
74.     reg             [    1: 0]      write_addr_index_d1         ;
75.     reg             [    3: 0]      state                       ;//state machine
76.
77.     1'b1 : 1'b0;//write finish at state 'S_END'
78.     assign                              write_finish            = (state == S_END) ?
79. always@(posedge mem_clk or posedge rst)
80. begin
81.     if(rst == 1'b1)
82.     begin
83.         write_req_d0    <=  1'b0;
84.         write_req_d1    <=  1'b0;
85.         write_req_d2    <=  1'b0;
86.         write_len_d0    <=  ZERO[ADDR_BITS - 1:0];      //equivalent to write_len_d0 <= 0;
87.         write_len_d1    <=  ZERO[ADDR_BITS - 1:0];      //equivalent to write_len_d1 <= 0;
88.         write_addr_index_d0     <=  2'b00;
89.         write_addr_index_d1     <=  2'b00;
90.     end
91.     else
92.     begin
93.         write_req_d0    <=  write_req;
94.         write_req_d1    <=  write_req_d0;
95.         write_req_d2    <=  write_req_d1;
96.         write_len_d0    <=  write_len;
97.         write_len_d1    <=  write_len_d0;
98.         write_addr_index_d0 <= write_addr_index;
99.         write_addr_index_d1 <= write_addr_index_d0;
100.    end
101.end
102.
103.
104.always@(posedge mem_clk or posedge rst)
105.begin
106.    if(rst == 1'b1)
107.    begin
108.        state <= S_IDLE;
109.        write_len_latch <= ZERO[ADDR_BITS - 1:0];
110.        wr_burst_addr <= ZERO[ADDR_BITS - 1:0];
111.        wr_burst_req <= 1'b0;
112.        write_cnt <= ZERO[ADDR_BITS - 1:0];
113.        fifo_aclr <= 1'b0;
114.        write_req_ack <= 1'b0;
115.        wr_burst_len <= ZERO[BUSRT_BITS - 1:0];
116.    end
117.    else
118.        case(state)
119.            //idle state,waiting for write write_req_d2 == '1' goto the 'S_ACK'
120.            S_IDLE:
121.            begin
122.                if(write_req_d2 == 1'b1)
123.                begin
124.                    state <= S_ACK;
```

```
                end
                write_req_ack <= 1'b0;
                end
 //'S_ACK' state completes the write request response, the FIFO reset, the address latch, and the data length latch
                S_ACK:
                begin
//after write request revocation(write_req_d2 == '0'),goto 'S_CHECK_FIFO',write_req_ack goto '0'
                    if(write_req_d2 == 1'b0)
                    begin
                        state <= S_CHECK_FIFO;
                        fifo_aclr <= 1'b0;
                        write_req_ack <= 1'b0;
                    end
                    else
                    begin
                        //write request response
                        write_req_ack <= 1'b1;
                        //FIFO reset
                        fifo_aclr <= 1'b1;
//select valid base address from write_addr_0 write_addr_1 write_addr_2 write_addr_3
                        if(write_addr_index_d1 == 2'd0)
                            wr_burst_addr <= write_addr_0;
                        else if(write_addr_index_d1 == 2'd1)
                            wr_burst_addr <= write_addr_1;
                        else if(write_addr_index_d1 == 2'd2)
                            wr_burst_addr <= write_addr_2;
                        else if(write_addr_index_d1 == 2'd3)
                            wr_burst_addr <= write_addr_3;
                        //latch data length
                        write_len_latch <= write_len_d1;
                    end
                    //write data counter reset, write_cnt <= 0;
                    write_cnt <= ZERO[ADDR_BITS - 1:0];
                end
                S_CHECK_FIFO:
                begin
                    //if there is a write request at this time, enter the 'S_ACK' state
                    if(write_req_d2 == 1'b1)
                    begin
                        state <= S_ACK;
                    end
                    //if the FIFO space is a burst write request, goto burst write state
                    else if(rdusedw >= BURST_SIZE)
                    begin
                        state <= S_WRITE_BURST;
                        wr_burst_len <= BURST_SIZE[BUSRT_BITS - 1:0];
                        wr_burst_req <= 1'b1;
                    end
                end

                S_WRITE_BURST:
                begin
                    //burst finish
                    if(wr_burst_finish == 1'b1)
                    begin
                        wr_burst_req <= 1'b0;
                        state <= S_WRITE_BURST_END;
                        //write counter + burst length
```

```
183.                        write_cnt <= write_cnt + BURST_SIZE[ADDR_BITS - 1:0];
184.                        //the next burst write address is generated
185.                        wr_burst_addr <= wr_burst_addr + BURST_SIZE[ADDR_BITS - 1:0];
186.                    end
187.                end
188.                S_WRITE_BURST_END:
189.                begin
190.                    //if there is a write request at this time, enter the 'S_ACK' state
191.                    if(write_req_d2 == 1'b1)
192.                    begin
193.                        state <= S_ACK;
194.                    end
195.                    //if the write counter value is less than the frame length, continue writing,
196.                    //otherwise the writing is complete
197.                    else if(write_cnt < write_len_latch)
198.                    begin
199.                        state <= S_CHECK_FIFO;
200.                    end
201.                    else
202.                    begin
203.                        state <= S_END;
204.                    end
205.                end
206.                S_END:
207.                begin
208.                    state <= S_IDLE;
209.                end
210.                default:
211.                    state <= S_IDLE;
212.            endcase
213.    end
214.    endmodule
```

frame_fifo_write 写 FIFO 控制模块的帧写入控制功能，用于将上层模块请求的数据按照突发方式写入 FIFO，并通过 FIFO 状态判断保证写入安全。模块以 mem_clk 时钟为基础，支持异步写请求同步，并可选择四个写起始地址，通过 write_addr_index 控制。

代码整体是写入控制逻辑，主要用于管理写请求、FIFO 空间检查、突发写入以及帧写入完成标志，同时通过 write_req_ack 响应上层写请求，通过 write_finish 指示一帧数据写入完成。

代码的 79-101 行是写请求和相关信号的同步逻辑。异步输入信号 write_req、write_len 与 write_addr_index 通过三级寄存器（write_req_d0/d1/d2 等）同步到 mem_clk 时钟域，防止毛刺或异步信号导致的误动作，并在状态机中使用 write_req_d2 作为触发条件。

代码的 104-214 行实现了状态机控制逻辑。状态机共包含六个状态：

S_IDLE（空闲）：等待同步后的写请求 write_req_d2 为高，进入 S_ACK 状态，同时 write_req_ack 输出低。

S_ACK（写请求应答）：拉高 write_req_ack 响应上层写请求，清空 FIFO (fifo_

aclr) 并锁存写长度 write_len_latch 和写起始地址 wr_burst_addr，等待写请求撤销后进入 S_CHECK_FIFO。

S_CHECK_FIFO（检查 FIFO 空间）：判断 FIFO 已使用字数 rdusedw 是否大于等于突发长度 BURST_SIZE，若满足条件则发起突发写请求 wr_burst_req 并进入 S_WRITE_BURST，否则持续检查，同时优先响应新写请求。

S_WRITE_BURST（突发写入）：等待外部存储器完成突发写入 wr_burst_finish，完成后更新写地址和写计数 write_cnt，进入 S_WRITE_BURST_END。

S_WRITE_BURST_END（突发写入结束）：若有新写请求，返回 S_ACK；否则判断写计数是否达到帧长度 write_len_latch，未完成返回 S_CHECK_FIFO 继续写入，完成则进入 S_END。

S_END（写入完成）：输出 write_finish 表示一帧数据写入完成，然后返回 S_IDLE。

在状态机逻辑中，wr_burst_addr、wr_burst_len 和 wr_burst_req 分别用于控制外部存储器的写地址、突发长度和写请求信号，write_cnt 用于累加已写入数据长度。write_req_ack 用于向上层模块确认写请求已接收，fifo_aclr 用于复位 FIFO，保证写入过程中数据不受干扰。

整体上模块通过 **异步写请求同步 + 六段状态机** 实现了可靠的帧级突发写入控制，能够根据 FIFO 状态自动控制突发写入，同时响应上层写请求并指示写入完成。

frame_fifo_read 读 FIFO 控制模块：

```
1.  `timescale 1ns/1ps
2.  module frame_fifo_read
3.  #
4.  (
5.      parameter                       MEM_DATA_BITS           = 32    ,
6.      parameter                       ADDR_BITS               = 23    ,
7.      parameter                       BUSRT_BITS              = 10    ,
8.      parameter                       FIFO_DEPTH              = 256   ,
9.      parameter                       BURST_SIZE              = 128
10. )
11. (
12.     input                           rst                     ,
13.     // external memory controller user interface clock
14.     input                           mem_clk                 ,
15.     // to external memory controller,send out a burst read request
16.     output reg                      rd_burst_req            ,
17.     // to external memory controller,data length of the burst read request, not bytes
18.     output reg      [BUSRT_BITS - 1:0]rd_burst_len          ,
19.     // to external memory controller,base address of the burst read request
20.     output reg      [ADDR_BITS - 1:0]rd_burst_addr          ,
21.    // from external memory controller,read request data valid
22.     input                           rd_burst_data_valid     ,
23.     // from external memory controller,burst read finish
24.     input                           rd_burst_finish         ,
25.     // data read module read request,keep '1' until read_req_ack = '1'
26.     input                           read_req                ,
27.     output reg                      read_req_ack    ,// data read module read request response
28.     output                          read_finish     ,// data read module read request finish
29.     // data read module read request base address 0, used when read_addr_index = 0
30.     input           [ADDR_BITS - 1:0]read_addr_0            ,
31.     // data read module read request base address 1, used when read_addr_index = 1
32.     input           [ADDR_BITS - 1:0]read_addr_1            ,
33.     // data read module read request base address 1, used when read_addr_index = 2
34.     input           [ADDR_BITS - 1:0]read_addr_2            ,
35.     // data read module read request base address 1, used when read_addr_index = 3
36.     input           [ADDR_BITS - 1:0]read_addr_3            ,
37.     // select valid base address from read_addr_0 read_addr_1 read_addr_2 read_addr_3
38.     input           [    1: 0]      read_addr_index         ,
39.     // data read module read request data length
40.     input           [ADDR_BITS - 1:0]read_len                ,
41.     output reg                      fifo_aclr       ,// to fifo asynchronous clear
42.     input           [   15: 0]      wrusedw         // from fifo write used words
43. );
44.     //256 bit '1'    you can use ONE[n-1:0] for n bit '1'
45.     localparam                      ONE                     = 256'd1;
46.     localparam                      ZERO                    = 256'd0;               //256 bit
    '0'
47. //read state machine code
48.     localparam                      S_IDLE      = 0     ;   //idle state,waiting for frame read
49.     localparam                      S_ACK       = 1     ;   //read request response
50.     //check the FIFO status, ensure that there is enough space to burst read
51.     localparam                      S_CHECK_FIFO            = 2   ;
52.     localparam                      S_READ_BURST            = 3   ;   //begin a burst read
53.     localparam                      S_READ_BURST_END        = 4   ;  //a burst read complete
54.     localparam                      S_END       = 5     ;  //a frame of data is read to complete
```

```
55.
56.     //asynchronous read request, synchronize to 'mem_clk' clock domain,first beat
57.     reg                                 read_req_d0                 ;
58.     reg                                 read_req_d1                 ;//second
59.     //third,Why do you need 3 ? Here's the design habit
60.     reg                                 read_req_d2                 ;
61.     //asynchronous read_len(read data length), synchronize to 'mem_clk' clock domain first
62.     reg                 [ADDR_BITS - 1:0]read_len_d0                    ;
63.     reg                 [ADDR_BITS - 1:0]read_len_d1                    ;//second
64.     reg                 [ADDR_BITS - 1:0]read_len_latch                 ;//lock read data length
65.     reg                 [ADDR_BITS - 1:0]read_cnt                       ;//read data counter
66.     reg                 [    3: 0]      state                       ;//state machine
67.     //synchronize to 'mem_clk' clock domain first
68.     reg                 [    1: 0]      read_addr_index_d0          ;
69.     //synchronize to 'mem_clk' clock domain second
70.     reg                 [    1: 0]      read_addr_index_d1          ;
71.
72.     1'b1 : 1'b0;//read finish at state 'S_END'
73.     assign                              read_finish                 = (state == S_END) ?
74. always@(posedge mem_clk or posedge rst)
75. begin
76.     if(rst == 1'b1)
77.     begin
78.         read_req_d0     <=  1'b0;
79.         read_req_d1     <=  1'b0;
80.         read_req_d2     <=  1'b0;
81.         read_len_d0     <=  ZERO[ADDR_BITS - 1:0];      //equivalent to read_len_d0 <= 0;
82.         read_len_d1     <=  ZERO[ADDR_BITS - 1:0];      //equivalent to read_len_d1 <= 0;
83.         read_addr_index_d0 <= 2'b00;
84.         read_addr_index_d1 <= 2'b00;
85.     end
86.     else
87.     begin
88.         read_req_d0     <=  read_req;
89.         read_req_d1     <=  read_req_d0;
90.         read_req_d2     <=  read_req_d1;
91.         read_len_d0     <=  read_len;
92.         read_len_d1     <=  read_len_d0;
93.         read_addr_index_d0 <= read_addr_index;
94.         read_addr_index_d1 <= read_addr_index_d0;
95.     end
96. end
97.
98.
99. always@(posedge mem_clk or posedge rst)
100.begin
101.    if(rst == 1'b1)
102.    begin
103.        state <= S_IDLE;
104.        read_len_latch <= ZERO[ADDR_BITS - 1:0];
105.        rd_burst_addr <= ZERO[ADDR_BITS - 1:0];
106.        rd_burst_req <= 1'b0;
107.        read_cnt <= ZERO[ADDR_BITS - 1:0];
108.        fifo_aclr <= 1'b0;
109.        rd_burst_len <= ZERO[BUSRT_BITS - 1:0];
110.        read_req_ack <= 1'b0;
111.    end
112.    else
```

```
        case(state)
            //idle state,waiting for read, read_req_d2 == '1' goto the 'S_ACK'
            S_IDLE:
            begin
                if(read_req_d2 == 1'b1)
                begin
                    state <= S_ACK;
                end
                    read_req_ack <= 1'b0;
                end
//'S_ACK' state completes the read request response, the FIFO reset, the address latch, and the data length latch
                S_ACK:
                begin
                    if(read_req_d2 == 1'b0)
                    begin
                        state <= S_CHECK_FIFO;
                        fifo_aclr <= 1'b0;
                        read_req_ack <= 1'b0;
                    end
                    else
                    begin
                        //read request response
                        read_req_ack <= 1'b1;
                        //FIFO reset
                        fifo_aclr <= 1'b1;
//select valid base address from read_addr_0 read_addr_1 read_addr_2 read_addr_3
                        if(read_addr_index_d1 == 2'd0)
                            rd_burst_addr <= read_addr_0;
                        else if(read_addr_index_d1 == 2'd1)
                            rd_burst_addr <= read_addr_1;
                        else if(read_addr_index_d1 == 2'd2)
                            rd_burst_addr <= read_addr_2;
                        else if(read_addr_index_d1 == 2'd3)
                            rd_burst_addr <= read_addr_3;
                        //latch data length
                        read_len_latch <= read_len_d1;
                    end
                    //read data counter reset, read_cnt <= 0;
                    read_cnt <= ZERO[ADDR_BITS - 1:0];
                end
                S_CHECK_FIFO:
                begin
                    //if there is a read request at this time, enter the 'S_ACK' state
                    if(read_req_d2 == 1'b1)
                    begin
                        state <= S_ACK;
                    end
                    //if the FIFO space is a burst read request, goto burst read state
                    else if(wrusedw < (FIFO_DEPTH - BURST_SIZE))
                    begin
                        state <= S_READ_BURST;
                        rd_burst_len <= BURST_SIZE[BUSRT_BITS - 1:0];
                        rd_burst_req <= 1'b1;
                    end
                end

                S_READ_BURST:
                begin
```

```
171.                    if(rd_burst_data_valid)
172.                        rd_burst_req <= 1'b0;
173.                    //burst finish
174.                    if(rd_burst_finish == 1'b1)
175.                    begin
176.                        state <= S_READ_BURST_END;
177.                        //read counter + burst length
178.                        read_cnt <= read_cnt + BURST_SIZE[ADDR_BITS - 1:0];
179.                        //the next burst read address is generated
180.                        rd_burst_addr <= rd_burst_addr + BURST_SIZE[ADDR_BITS - 1:0];
181.                    end
182.                end
183.                S_READ_BURST_END:
184.                begin
185.                    //if there is a read request at this time, enter the 'S_ACK' state
186.                    if(read_req_d2 == 1'b1)
187.                    begin
188.                        state <= S_ACK;
189.                    end
190.                    //if the read counter value is less than the frame length, continue read,
191.                    //otherwise the read is complete
192.                    else if(read_cnt < read_len_latch)
193.                    begin
194.                        state <= S_CHECK_FIFO;
195.                    end
196.                    else
197.                    begin
198.                        state <= S_END;
199.                    end
200.                end
201.                S_END:
202.                begin
203.                    state <= S_IDLE;
204.                end
205.                default:
206.                    state <= S_IDLE;
207.            endcase
208.    end
209.    endmodule
210.
```

rame_fifo_read 读 FIFO 控制模块同样是使用状态机实现了基于 FIFO 的帧读取控制功能，将 FIFO 中的数据按照突发方式读取块，同时通过 FIFO 空间判断保证读取安全。模块以 mem_clk 时钟为基础，支持异步读请求同步，并可选择四个读起始地址，通过 read_addr_index 控制。

代码的 74 行到 209 行是读控制逻辑，主要用于管理读请求、FIFO 空间检查、突发读操作以及帧读取完成标志，同时通过 read_req_ack 响应上层读请求，通过 read_finish 指示一帧数据读取完成。

首先，代码的 74-96 行是读请求及相关信号的同步逻辑。异步输入信号 read_req、read_len 与 read_addr_index 通过三级寄存器（read_req_d0/d1/d2 等）同步到 mem_clk 时钟域，防止异步毛刺或信号漂移导致误动作，并在状态机中使用 read_req_d2 作为

触发条件。

代码的 99-209 行实现了状态机控制逻辑。状态机共包含六个状态：

S_IDLE（空闲）：等待同步后的读请求 read_req_d2 为高，进入 S_ACK 状态，同时 read_req_ack 输出低。

S_ACK（读请求应答）：拉高 read_req_ack 响应上层读请求，清空 FIFO (fifo_aclr) 并锁存读长度 read_len_latch 和读起始地址 rd_burst_addr，等待读请求撤销后进入 S_CHECK_FIFO。

S_CHECK_FIFO（检查 FIFO 空间）：判断 FIFO 可用空间是否大于等于突发长度 BURST_SIZE（通过 wrusedw < FIFO_DEPTH - BURST_SIZE 判断），若满足条件则发起突发读请求 rd_burst_req 并进入 S_READ_BURST，否则持续检查，同时优先响应新读请求。

S_READ_BURST（突发读取）：在 rd_burst_data_valid 有效信号时拉低 rd_burst_req，等待外部存储器完成突发读取 rd_burst_finish，完成后更新读地址和读计数 read_cnt，进入 S_READ_BURST_END。

S_READ_BURST_END（突发读取结束）：若有新读请求，返回 S_ACK；否则判断读计数是否达到帧长度 read_len_latch，未完成返回 S_CHECK_FIFO 继续读取，完成则进入 S_END。

S_END（读取完成）：输出 read_finish 表示一帧数据读取完成，然后返回 S_IDLE。

在状态机逻辑中，rd_burst_addr、rd_burst_len 和 rd_burst_req 分别用于控制外部存储器的读地址、突发长度和读请求信号，read_cnt 用于累加已读取数据长度。read_req_ack 用于向上层模块确认读请求已接收，fifo_aclr 用于复位 FIFO，保证读取过程中数据不受干扰。

aq_axi_master_128AXI 读写主机：

```
1.  module aq_axi_master_128 #(
2.      parameter                       DATA_WIDTH              = 128
3.  )(
4.    // Reset, Clock
5.      input                           ARESETN                 ,
6.      input                           ACLK                    ,
7.
8.    // Master Write Address
9.      output              [   0: 0]   M_AXI_AWID              ,
10.     output              [  31: 0]   M_AXI_AWADDR            ,
11.     output              [   7: 0]   M_AXI_AWLEN     ,// Burst Length: 0-255
12.     output              [   2: 0]   M_AXI_AWSIZE    ,// Burst Size: 100
13.     output              [   1: 0]   M_AXI_AWBURST ,// Burst Type: Fixed 2'b01(Incremental Burst)
14.     output                          M_AXI_AWLOCK            ,// Lock: Fixed 2'b00
15.     output              [   3: 0]   M_AXI_AWCACHE           ,// Cache: Fiex 2'b0011
```

```
    output              [   2: 0]       M_AXI_AWPROT            ,// Protect: Fixed 2'b000
    output              [   3: 0]       M_AXI_AWQOS             ,// QoS: Fixed 2'b0000
    output              [   0: 0]       M_AXI_AWUSER            ,// User: Fixed 32'd0
    output                              M_AXI_AWVALID           ,
    input                               M_AXI_AWREADY           ,

// Master Write Data
    output              [DATA_WIDTH-1: 0]M_AXI_WDATA            ,
    output              [DATA_WIDTH/8-1: 0]M_AXI_WSTRB          ,
    output                              M_AXI_WLAST             ,
    output              [   0: 0]       M_AXI_WUSER             ,
    output                              M_AXI_WVALID            ,
    input                               M_AXI_WREADY            ,

// Master Write Response
    input               [   0: 0]       M_AXI_BID               ,
    input               [   1: 0]       M_AXI_BRESP             ,
    input               [   0: 0]       M_AXI_BUSER             ,
    input                               M_AXI_BVALID            ,
    output                              M_AXI_BREADY            ,

// Master Read Address
    output              [   0: 0]       M_AXI_ARID              ,
    output              [  31: 0]       M_AXI_ARADDR            ,
    output              [   7: 0]       M_AXI_ARLEN             ,
    output              [   2: 0]       M_AXI_ARSIZE            ,
    output              [   1: 0]       M_AXI_ARBURST           ,
    output              [   1: 0]       M_AXI_ARLOCK            ,//
    output              [   3: 0]       M_AXI_ARCACHE           ,
    output              [   2: 0]       M_AXI_ARPROT            ,
    output              [   3: 0]       M_AXI_ARQOS             ,
    output              [   0: 0]       M_AXI_ARUSER            ,
    output                              M_AXI_ARVALID           ,
    input                               M_AXI_ARREADY           ,

// Master Read Data
    input               [   0: 0]       M_AXI_RID               ,
    input               [DATA_WIDTH-1: 0]M_AXI_RDATA            ,//
    input               [   1: 0]       M_AXI_RRESP             ,
    input                               M_AXI_RLAST             ,
    input               [   0: 0]       M_AXI_RUSER             ,
    input                               M_AXI_RVALID            ,
    output                              M_AXI_RREADY            ,

// Local Bus
    input                               MASTER_RST              ,

    input                               WR_START                ,
    input               [  31: 0]       WR_ADRS                 ,
    input               [  31: 0]       WR_LEN                  ,
    output                              WR_READY                ,
    output                              WR_FIFO_RE              ,
    input                               WR_FIFO_EMPTY           ,
    input                               WR_FIFO_AEMPTY          ,
    input               [DATA_WIDTH-1: 0]WR_FIFO_DATA           ,
    output                              WR_DONE                 ,

    input                               RD_START                ,
```

```
74.     input               [  31: 0]       RD_ADRS                     ,
75.     input               [  31: 0]       RD_LEN                     ,
76.     output                              RD_READY                     ,
77.     output                              RD_FIFO_WE                   ,
78.     input                               RD_FIFO_FULL                 ,
79.     input                               RD_FIFO_AFULL                ,
80.     output              [DATA_WIDTH-1: 0]RD_FIFO_DATA                ,
81.     output                              RD_DONE                     ,
82.
83.     output              [  31: 0]       DEBUG
84. );
85.
86.     localparam                          S_WR_IDLE                   = 3'd0  ;
87.     localparam                          S_WA_WAIT                    = 3'd1  ;
88.     localparam                          S_WA_START                   = 3'd2  ;
89.     localparam                          S_WD_WAIT                    = 3'd3  ;
90.     localparam                          S_WD_PROC                    = 3'd4  ;
91.     localparam                          S_WR_WAIT                    = 3'd5  ;
92.     localparam                          S_WR_DONE                    = 3'd6  ;
93.
94.     reg                 [   2: 0]       wr_state                    ;
95.     reg                 [  31: 0]       reg_wr_adrs                  ;
96.     reg                 [  31: 0]       reg_wr_len                   ;
97.     reg                                 reg_awvalid,                reg_w_last;
98.     reg                                 reg_wvalid                   ;
99.     reg                 [   7: 0]       reg_w_len                    ;
100.    reg                 [   7: 0]       reg_w_stb                    ;
101.    reg                 [   1: 0]       reg_wr_status                ;
102.    reg                 [   3: 0]       reg_w_count,                reg_r_count;
103.
104.    reg                 [   7: 0]       rd_chkdata,                 wr_chkdata;
105.    reg                 [   1: 0]       resp                         ;
106.    reg                                 rd_first_data                ;
107.    reg                                 rd_fifo_enable               ;
108.    reg                 [  31: 0]       rd_fifo_cnt                  ;
109.    assign                              WR_DONE                         = (wr_state == S_WR_DONE);
110.
111.
112.
113.    assign WR_FIFO_RE   = rd_first_data | (reg_wvalid & ~WR_FIFO_EMPTY & M_AXI_WREADY & rd_fifo_enable);
114.    //assign WR_FIFO_RE           = reg_wvalid & ~WR_FIFO_EMPTY & M_AXI_WREADY;
115.    always @(posedge ACLK or negedge ARESETN)
116.    begin
117.        if(!ARESETN)
118.            rd_fifo_cnt <= 32'd0;
119.        else if(WR_FIFO_RE)
120.            rd_fifo_cnt <= rd_fifo_cnt + 32'd1;
121.        else if(wr_state == S_WR_IDLE)
122.            rd_fifo_cnt <= 32'd0;
123.    end
124.
125.    always @(posedge ACLK or negedge ARESETN)
126.    begin
127.        if(!ARESETN)
128.            rd_fifo_enable <= 1'b0;
129.        else if(wr_state == S_WR_IDLE && WR_START)
130.            rd_fifo_enable <= 1'b1;
131.        else if(WR_FIFO_RE && (rd_fifo_cnt == RD_LEN[31:4] - 32'd1) )//5
```

```
132.             rd_fifo_enable <= 1'b0;
133.     end
134.       // Write State
135.       always @(posedge ACLK or negedge ARESETN) begin
136.         if(!ARESETN) begin
137.           wr_state              <= S_WR_IDLE;
138.           reg_wr_adrs[31:0]     <= 32'd0;
139.           reg_wr_len[31:0]      <= 32'd0;
140.           reg_awvalid           <= 1'b0;
141.           reg_wvalid            <= 1'b0;
142.           reg_w_last            <= 1'b0;
143.           reg_w_len[7:0]        <= 8'd0;
144.           reg_w_stb[7:0]        <= 8'd0;
145.           reg_wr_status[1:0]    <= 2'd0;
146.           reg_w_count[3:0]      <= 4'd0;
147.           reg_r_count[3:0]    <= 4'd0;
148.           wr_chkdata            <= 8'd0;
149.           rd_chkdata <= 8'd0;
150.           resp <= 2'd0;
151.           rd_first_data <= 1'b0;
152.       end else begin
153.         if(MASTER_RST) begin
154.           wr_state <= S_WR_IDLE;
155.         end else begin
156.           case(wr_state)
157.             S_WR_IDLE: begin
158.               if(WR_START) begin
159.                 wr_state            <= S_WA_WAIT;
160.                 reg_wr_adrs[31:0] <= WR_ADRS[31:0];//突发传输长度
161.                 reg_wr_len[31:0]   <= WR_LEN[31:0] -32'd1;//16
162.                 rd_first_data <= 1'b1;
163.               end
164.               reg_awvalid           <= 1'b0;
165.               reg_wvalid            <= 1'b0;
166.               reg_w_last            <= 1'b0;
167.               reg_w_len[7:0]        <= 8'd0;
168.               reg_w_stb[7:0]        <= 8'd0;
169.               reg_wr_status[1:0]    <= 2'd0;
170.             end
171.             S_WA_WAIT: begin
172.               if(!WR_FIFO_AEMPTY | (reg_wr_len[31:11] == 21'd0)) begi
173.                 wr_state            <= S_WA_START;
174.               end
175.               rd_first_data <= 1'b0;
176.             end
177.             S_WA_START: begin
178.               wr_state              <= S_WD_WAIT;
179.               reg_awvalid           <= 1'b1;
180.               reg_wr_len[31:11]     <= reg_wr_len[31:11] - 21'd1;
181.               if(reg_wr_len[31:11] != 21'd0) begin
182.                 reg_w_len[7:0]    <= 8'hFF;
183.                 reg_w_last        <= 1'b0;
184.                 reg_w_stb[7:0]    <= 8'hFF;
185.               end else begin
186.             reg_w_len[7:0]    <= reg_wr_len[10:4];
187.                 reg_w_last        <= 1'b1;
188.                 reg_w_stb[7:0]    <= 8'hFF;
189.               end
```

```
            end
            S_WD_WAIT: begin
              if(M_AXI_AWREADY) begin
                wr_state          <= S_WD_PROC;
                reg_awvalid       <= 1'b0;
                reg_wvalid        <= 1'b1;
              end
            end
            S_WD_PROC: begin
              if(M_AXI_WREADY & ~WR_FIFO_EMPTY) begin
                if(reg_w_len[7:0] == 8'd0) begin
                  wr_state          <= S_WR_WAIT;
                  reg_wvalid        <= 1'b0;
                  reg_w_stb[7:0]   <= 8'h00;
                end else begin
                  reg_w_len[7:0]   <= reg_w_len[7:0] -8'd1;
                end
              end
            end
            S_WR_WAIT: begin
              if(M_AXI_BVALID) begin
                reg_wr_status[1:0]   <= reg_wr_status[1:0] | M_AXI_BRESP[1:0];
                if(reg_w_last) begin
                  wr_state            <= S_WR_DONE;
                end else begin
                  wr_state            <= S_WA_WAIT;
                  reg_wr_adrs[31:0] <= reg_wr_adrs[31:0] + 32'd2048;
                end
              end
            end
            S_WR_DONE: begin
                wr_state <= S_WR_IDLE;
              end

            default: begin
              wr_state <= S_WR_IDLE;
            end
          endcase

        end
      end
    end

    assign M_AXI_AWID            = 1'b0;
    assign M_AXI_AWADDR[31:0] = reg_wr_adrs[31:0];
    assign M_AXI_AWLEN[7:0]     = reg_w_len[7:0];
    assign M_AXI_AWSIZE[2:0]   = 3'b100;
    assign M_AXI_AWBURST[1:0] = 2'b01;
    assign M_AXI_AWLOCK          = 1'b0;
    assign M_AXI_AWCACHE[3:0] = 4'b0011;
    assign M_AXI_AWPROT[2:0]   = 3'b000;
    assign M_AXI_AWQOS[3:0]     = 4'b0000;
    assign M_AXI_AWUSER[0]      = 1'b1;
    assign M_AXI_AWVALID        = reg_awvalid;

    assign M_AXI_WDATA   = WR_FIFO_DATA;
    assign M_AXI_WSTRB     = (reg_wvalid & ~WR_FIFO_EMPTY)?16'hffff:16'h0000;
    assign M_AXI_WLAST          = (reg_w_len[7:0] == 8'd0)?1'b1:1'b0;
```

```
248.      assign M_AXI_WUSER          = 1;
249.      assign M_AXI_WVALID          = reg_wvalid & ~WR_FIFO_EMPTY;
250.    //  assign M_AXI_WVALID          = (wr_state == S_WD_PROC)?1'b1:1'b0;
251.
252.      assign M_AXI_BREADY          = M_AXI_BVALID;
253.
254.      assign WR_READY              = (wr_state == S_WR_IDLE)?1'b1:1'b0;
255.
256.    //  assign WR_FIFO_RE            = (wr_state == S_WD_PROC)?M_AXI_WREADY:1'b0;
257.
258.      localparam S_RD_IDLE   = 3'd0;
259.      localparam S_RA_WAIT   = 3'd1;
260.      localparam S_RA_START = 3'd2;
261.      localparam S_RD_WAIT   = 3'd3;
262.      localparam S_RD_PROC   = 3'd4;
263.      localparam S_RD_DONE   = 3'd5;
264.
265.      reg [2:0]    rd_state;
266.      reg [31:0]   reg_rd_adrs;
267.      reg [31:0]   reg_rd_len;
268.      reg          reg_arvalid, reg_r_last;
269.      reg [7:0]    reg_r_len;
270.     assign RD_DONE = (rd_state == S_RD_DONE) ;
271.      // Read State
272.      always @(posedge ACLK or negedge ARESETN) begin
273.        if(!ARESETN) begin
274.          rd_state            <= S_RD_IDLE;
275.          reg_rd_adrs[31:0] <= 32'd0;
276.          reg_rd_len[31:0]   <= 32'd0;
277.          reg_arvalid         <= 1'b0;
278.          reg_r_len[7:0]     <= 8'd0;
279.        end else begin
280.          case(rd_state)
281.            S_RD_IDLE: begin
282.              if(RD_START) begin
283.                rd_state            <= S_RA_WAIT;
284.                reg_rd_adrs[31:0] <= RD_ADRS[31:0];
285.                reg_rd_len[31:0]   <= RD_LEN[31:0] -32'd1;
286.              end
287.              reg_arvalid       <= 1'b0;
288.              reg_r_len[7:0]   <= 8'd0;
289.            end
290.            S_RA_WAIT: begin
291.              if(~RD_FIFO_AFULL) begin
292.                rd_state            <= S_RA_START;
293.              end
294.            end
295.            S_RA_START: begin
296.              rd_state            <= S_RD_WAIT;
297.              reg_arvalid         <= 1'b1;
298.              reg_rd_len[31:11] <= reg_rd_len[31:11] -21'd1;
299.              if(reg_rd_len[31:11] != 21'd0) begin
300.                reg_r_last       <= 1'b0;
301.                reg_r_len[7:0]   <= 8'd255;
302.              end else begin
303.                reg_r_last       <= 1'b1;
304.                reg_r_len[7:0]   <= reg_rd_len[10:4];
305.              end
```

```
306.              end
307.              S_RD_WAIT: begin
308.                if(M_AXI_ARREADY) begin
309.                  rd_state          <= S_RD_PROC;
310.                  reg_arvalid       <= 1'b0;
311.                end
312.              end
313.              S_RD_PROC: begin
314.                if(M_AXI_RVALID) begin
315.                  if(M_AXI_RLAST) begin
316.                    if(reg_r_last) begin
317.                      rd_state            <= S_RD_DONE;
318.                    end else begin
319.                      rd_state            <= S_RA_WAIT;
320.                      reg_rd_adrs[31:0] <= reg_rd_adrs[31:0] + 32'd2048;
321.                    end
322.                  end else begin
323.                    reg_r_len[7:0] <= reg_r_len[7:0] -8'd1;
324.                  end
325.                end
326.              end
327.              S_RD_DONE:begin
328.                  rd_state            <= S_RD_IDLE;
329.              end
330.
331.            endcase
332.          end
333.        end
334.
335.        // Master Read Address
336.        assign M_AXI_ARID           = 1'b0;
337.        assign M_AXI_ARADDR[31:0] = reg_rd_adrs[31:0];
338.        assign M_AXI_ARLEN[7:0]     = reg_r_len[7:0];
339.        assign M_AXI_ARSIZE[2:0]   = 3'b100 ;
340.        assign M_AXI_ARBURST[1:0] = 2'b01;
341.        assign M_AXI_ARLOCK         = 1'b0;
342.        assign M_AXI_ARCACHE[3:0] = 4'b0011;
343.        assign M_AXI_ARPROT[2:0]   = 3'b000;
344.        assign M_AXI_ARQOS[3:0]     = 4'b0000;
345.        assign M_AXI_ARUSER[0]      = 1'b1;
346.        assign M_AXI_ARVALID        = reg_arvalid;
347.
348.        assign M_AXI_RREADY         = M_AXI_RVALID & ~RD_FIFO_FULL;
349.
350.        assign RD_READY             = (rd_state == S_RD_IDLE)?1'b1:1'b0;
351.        assign RD_FIFO_WE           = M_AXI_RVALID;
352.        assign RD_FIFO_DATA = M_AXI_RDATA;
353.
354.        assign DEBUG[31:0] = {reg_wr_len[31:8],
355.                              1'd0, wr_state[2:0], 1'd0, rd_state[2:0]};
356.
357.    endmodule
358.
```

aq_axi_master_128 模块是基于 AXI4 协议的 128 位主控接口，用于在 FPGA 中实现高速数据传输。模块支持完整的 AXI 写和读操作，能够与本地 FIFO 对接，实现数据的连续读写，并支持大容量数据的突发传输（burst）。模块接口主要包括三部分：

AXI 写通道接口、AXI 读通道接口和本地总线接口。其中，AXI 写通道包含地址、数据和响应通道；AXI 读通道包含地址和数据通道。本地总线接口用于控制 FPGA 内部的数据 FIFO，并接收上层系统的读写请求和数据。

写操作使用独立的状态机控制，共包含七个状态：空闲（S_WR_IDLE）、等待（S_WA_WAIT）、地址启动（S_WA_START）、写地址等待（S_WD_WAIT）、数据传输（S_WD_PROC）、写响应等待（S_WR_WAIT）以及写完成（S_WR_DONE）。在空闲状态下，模块等待外部发起写请求信号 WR_START，同时接收写地址 WR_ADRS 和写长度 WR_LEN。当请求有效时，状态机进入等待状态，判断 FIFO 中数据是否可用或是否满足传输长度条件。随后，模块通过 AXI 写地址通道发起突发传输请求，AWVALID 信号拉高，突发长度由寄存器 reg_w_len 计算，并根据总长度将数据拆分为多次突发传输。

在数据传输状态，模块从本地 FIFO (WR_FIFO_DATA) 读取数据，并通过 WVALID、WLAST 和 WSTRB 控制信号发送至 AXI 总线。每次突发传输完成后，模块等待写响应通道返回的 BVALID 信号，确认传输成功后，状态机根据剩余数据长度决定是否继续下一突发传输或进入写完成状态。写完成时，WR_DONE 信号有效，同时状态机回到空闲状态，准备接收下一次写请求。

FIFO 控制通过 WR_FIFO_RE 信号实现，数据计数由 rd_fifo_cnt 管理。模块提供 WR_READY 信号，表示当前写通道空闲，可接受新的写事务。

读操作同样采用独立状态机控制，包括六个状态：空闲（S_RD_IDLE）、等待（S_RA_WAIT）、地址启动（S_RA_START）、地址等待（S_RD_WAIT）、数据接收（S_RD_PROC）以及读完成（S_RD_DONE）。在空闲状态下，模块等待外部发起读请求 RD_START，同时接收读地址 RD_ADRS 和读长度 RD_LEN。状态机在 FIFO 有足够空间时，通过 AXI 读地址通道发起突发读请求，ARVALID 信号拉高，突发长度由寄存器 reg_r_len 控制，长度大于 256 时会拆分为多次突发传输。

数据接收过程中，模块通过 AXI 读数据通道接收数据，当 RVALID 有效且 FIFO 未满时，将数据写入本地读 FIFO (RD_FIFO_DATA)，同时拉高 RD_FIFO_WE 信号。每次突发传输完成后，状态机根据剩余数据长度决定是否继续下一突发读，直到整个读事务完成。读完成后，RD_DONE 信号有效，状态机回到空闲状态，准备接受下一次读请求。

模块在写操作和读操作中都支持大于单次 AXI 突发最大值（255 个数据）的分段传输。通过寄存器高位控制 reg_wr_len[31:11] 或 reg_rd_len[31:11]，模块可将大块数据拆分为多次突发，每次突发最多 256 个数据，确保传输符合 AXI 协议限制，同时提高数据总线利用率。

AXI 信号说明

写地址通道（AW）：M_AXI_AWADDR、M_AXI_AWLEN、M_AXI_AWVALID

由模块寄存器控制，突发类型固定为增量突发（INCR），总线大小为 128-bit。

写数据通道（W）：M_AXI_WDATA 连接本地 FIFO 数据，WSTRB 全字节有效，WLAST 指示当前突发传输的最后一个数据。

写响应通道（B）：M_AXI_BREADY 与 BVALID 配合，保证数据传输成功。

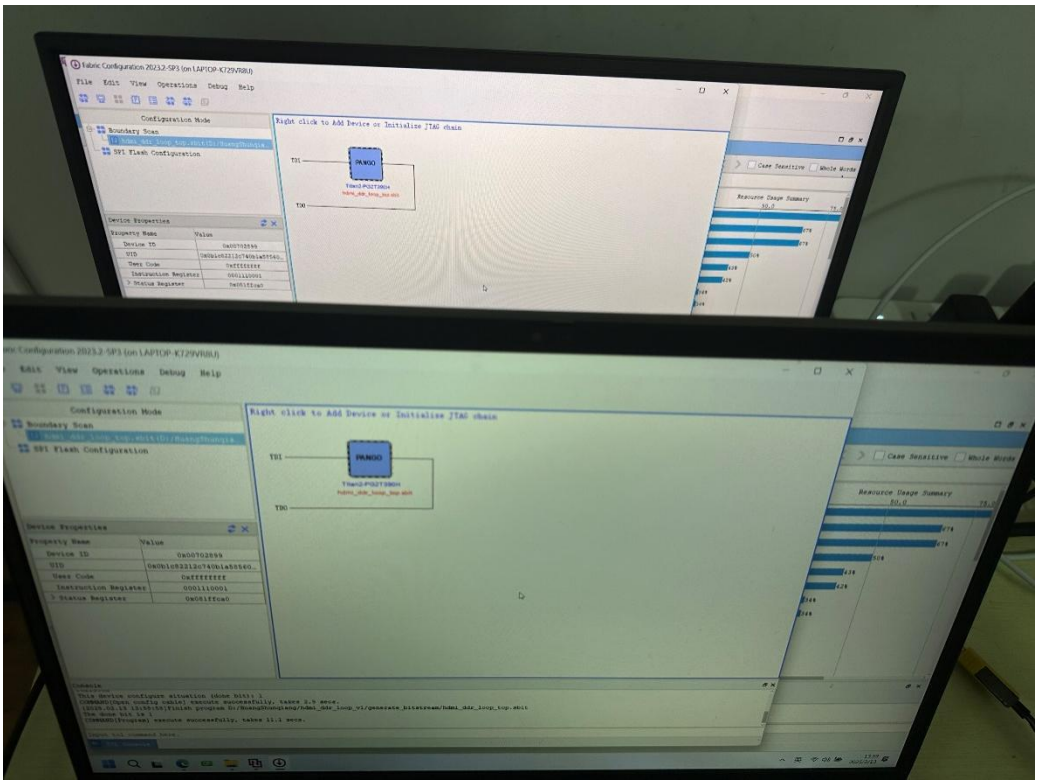
读地址通道（AR）：M_AXI_ARADDR、M_AXI_ARLEN、M_AXI_ARVALID 控制读突发请求。

读数据通道（R）：M_AXI_RDATA 写入 FIFO，RREADY 控制数据接收。

状态与调试接口 ：模块提供 DEBUG[31:0] 输出，用于监控写长度和读写状态，便于仿真和调试。WR_READY 与 RD_READY 信号指示当前通道是否空闲，可接收新的事务请求。

12.5. 实验现象

连接好 PT2G390H 开发板、视频源和显示器，视频源建议设置为 1920*1080P@60，下图为设置分辨率步骤，下载程序，可以看到显示器显示与视频源一致的图像。



13. 基于 UDP 的以太网传输实验例程

13.1. 实验简介

实验目的:

完成基于 UDP 的以太网通信测试。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

13.2. 开发板以太网接口简介

PT2G390H 开发板使用 Realtek RTL8211E PHY 实现了一个 10/100/1000 以太网端口，用于网络连接。该器件工作电压为支持 2.5V、3.3V，通过 RGMII 接口连接到 PT2G390H。RJ-45 连接器是 HFJ11-1G01E-L12RL，具有集成的自动缠绕磁性元件，可提高性能，质量和可靠性。

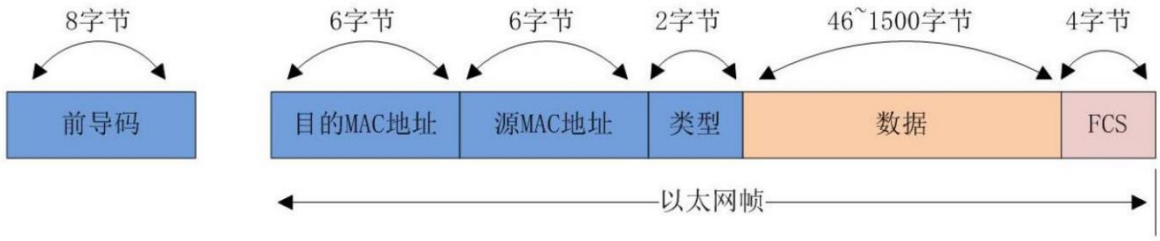
RJ-45 有两个状态指示灯 LED，用于指示流量和有效链路状态（详情请查看“PT2G390H 开发板硬件使用手册”）。

13.3. 实验要求

通过以太网端口实现 PC 端和开发板间通信，实现了 ARP，UDP 功能。

13.4. 以太网协议简介

13.4.1. 以太网帧格式



前导码（Preamble）：8 字节，连续 7 个 8’h55 加 1 个 8’hd5，表示一个帧的开始，用于双方设备数据的同步。

目的 MAC 地址：6 字节，存放目的设备的物理地址，即 MAC 地址；

源 MAC 地址：6 字节，存放发送端设备的物理地址；

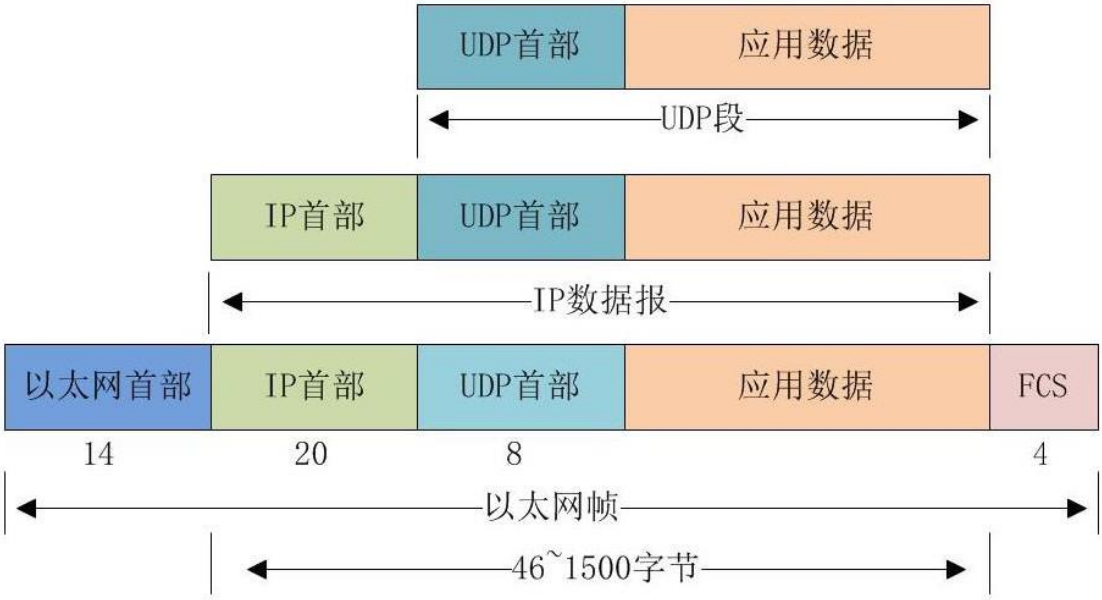
类型：2 字节，用于指定协议类型，常用的有 0800 表示 IP 协议，0806 表示 AR

P 协议，8035 表示 RARP 协议；

数据：46 到 1500 字节，最少 46 字节，不足需要补全 46 字节，例如 IP 协议层就包含在数据部分，包括其 IP 头及数据。

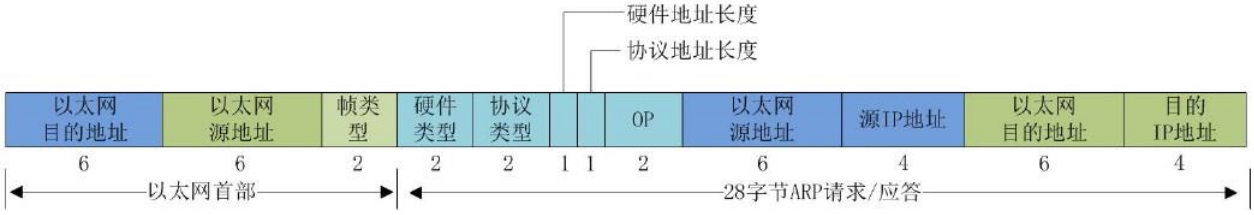
FCS：帧尾，4 字节，称为帧校验序列，采用 32 位 CRC 校验，对目的 MAC 地址字段到数据字段进行校验。

进一步扩展，以 UDP 协议为例，可以看到其结构如下，除了以太网首部的 14 字节，数据部分包含 IP 首部，UDP 首部，应用数据共 46~1500 字节。



13.4.2. ARP 数据报格式

ARP 地址解析协议，即 ARP（Address Resolution Protocol），根据 IP 地址获取物理地址。主机发送包含目的 IP 地址的 ARP 请求广播（MAC 地址为 48'hff_ff_ff_ff_ff_ff）到网络上的主机，并接收返回消息，以此确定目标的物理地址，收到返回消息后将 IP 地址和物理地址保存到缓存中，并保留一段时间，下次请求时直接查询 ARP 缓存以节约资源。下图为 ARP 数据报格式。



帧类型：ARP 帧类型为两字节 0806；

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，采用 0x0800 IP 类型，之后的硬件地址长度和协议地址长度分别对应 6 和 4；

OP 字段中 1 表示 ARP 请求，2 表示 ARP 应答

例如：|ff ff ff ff ff ff|00 0a 35 01 fe c0|08 06|00 01|08 00|06|04|00 01|00 0a 35 01 fe c0|c0 a8 00 02| ff ff ff ff ff ff|c0 a8 00 03|

表示向 192.168.0.3 地址发送 ARP 请求。

|00 0a 35 01 fe c0 | 60 ab c1 a2 d5 15 |08 06|00 01|08 00|06|04|00 02| 60 ab c1 a2 d5 15|c0 a8 00 03|00 0a 35 01 fe c0|c0 a8 00 02|

表示向 192.168.0.2 地址发送 ARP 应答。

13.4.3. IP 数据包格式

因为 UDP 协议包只是 IP 包中的一种，所以我们来介绍一下 IP 包的数据格式。下图为 IP 分组的报文头格式，报文头的前 20 个字节是固定的，后面的可变

版本:占 4 位,指 IP 协议的版本目前的 IP 协议版本号为 4 (即 IPv4);

首部长度:占 4 位,可表示的最大数值是 15 个单位(一个单位为 4 字节)因此 IP 的首部长度的最大值是 60 字节;

区分服务:占 8 位,用来获得更好的服务,在旧标准中叫做服务类型,但实际上一直未被使用过 1998 年这个字段改名为区分服务.只有在使用区分服务(DiffServ)时,这个字段才起作用.一般的情况下都不使用这个字段;

总长度:占 16 位,指首部和数据之和的长度,单位为字节,因此数据报的最大长度为 65535 字节总长度必须不超过最大传送单元 MTU

标识:占 16 位,它是一个计数器,用来产生数据报的标识

标志(flag):

占 3 位,目前只有前两位有意义

MF：标志字段的最低位是 MF (More Fragment)，MF=1 表示后面“还有分片”。MF=0 表示最后一个分片

DF：标志字段中间的一位是 DF (Don't Fragment)，只有当 DF=0 时才允许分片

片偏移:占 12 位,指较长的分组在分片后某片在原分组中的相对位置.片偏移以 8 个

字节为偏移单位;

生存时间:占 8 位,记为 TTL (Time To Live) 数据报在网络中可通过的路由器数的最大值,TTL 字段是由发送端初始设置一个 8 bit 字段.推荐的初始值由分配数字 RFC 指定,当前值为 64.发送 ICMP 回显应答时经常把 TTL 设为最大值 255;

协议:占 8 位,指出此数据报携带的数据使用何种协议以便目的主机的 IP 层将数据部分上交给哪个处理过程, 1 表示为 ICMP 协议, 2 表示为 IGMP 协议, 6 表示为 TCP 协议, 17 表示为 UDP 协议;

首部检验和:占 16 位,只检验数据报的首部不检验数据部分，采用二进制反码求和，即将 16 位数据相加后，再将进位与低 16 位相加，直到进位为 0，最后将 16 位取反;

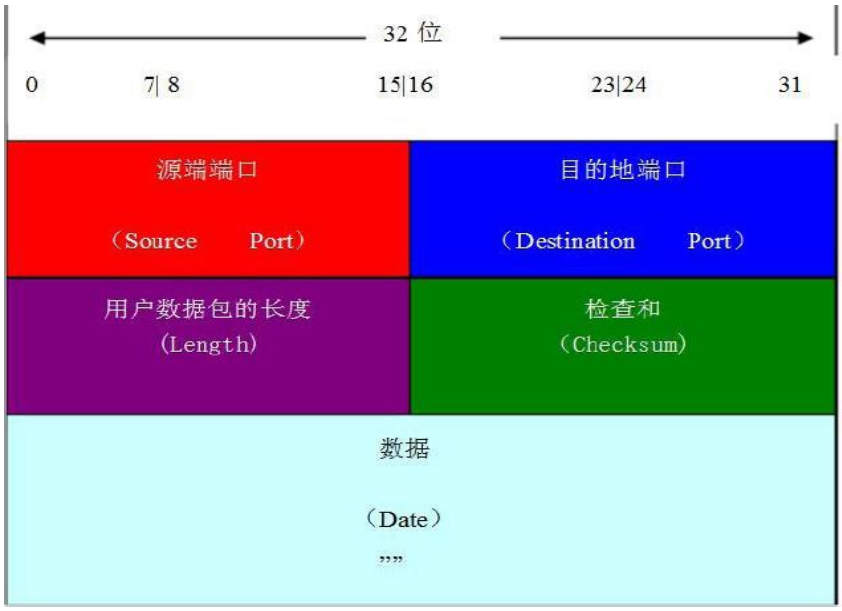
源地址和目的地址:都各占 4 字节,分别记录源地址和目的地址;

13.4.4. UDP 协议

UDP 是 User Datagram Protocol（用户数据报协议）的英文缩写。UDP 只提供一种基本的、低延迟的被称为数据报的通讯。所谓数据报，就是一种自带寻址信息，从发送端走到接收端的数据包。UDP 协议经常用于图像传输、网络监控数据交换等数据传输速度要求比较高的场合。

UDP 协议的报头格式:

UDP 报头由 4 个域组成，其中每个域各占用 2 个字节，具体如下:



① UDP 源端口号

② 目标端口号

③ 数据报长度

④ 校验和

UDP 协议使用端口号为不同的应用保留其各自的数据传输通道。数据发送一方将

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。

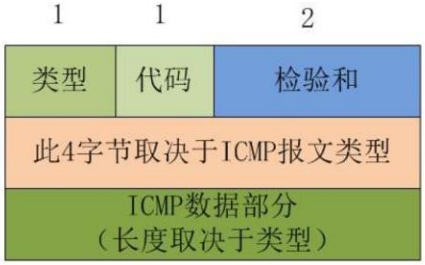
数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些

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。虽然 UDP 提供有错误检测，但检测到错误时，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。

13.4.5. Ping 功能

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。虽然 UDP 提供有错误检测，但检测到错误时，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。



类型值（十进制）	ICMP报文类型
0	回送应答
3	终点不可达
4	源点抑制
5	改变路由
8	回送请求
11	时间超时

13.5. SMI(MDC/MDIO)总线接口

串行管理接口（ Serial Management Interface ），也被称作 MII 管理接口（ MII ManagementInterf-ace），包括 MDC 和 MDIO 两条信号线。MDIO 是一个 PHY 的

管理接口，用来读/写 PHY 的寄存器，以控制 PHY 的行为或获取 PHY 的状态，MDC 为 MDIO 提供时钟，由 MAC 端提供，在本实验中也就是 FPGA 端。在 RTL8211EG 文档里可以看到 MDC 的周期最小为 400ns，也就是最大时钟为 2.5MHz。

Table 61. MDC/MDIO Management Timing Parameters

Symbol	Description	Minimum	Maximum	Unit
t_1	MDC High Pulse Width	160	-	ns
t_2	MDC Low Pulse Width	160	-	ns
t_3	MDC Period	400	-	ns
t_4	MDIO Setup to MDC Rising Edge	10	-	ns
t_5	MDIO Hold Time from MDC Rising Edge	10	-	ns
t_6	MDIO Valid from MDC Rising Edge	0	300	ns

13.5.1. SMI 帧格式

如下图，为 SMI 的读写帧格式：

	Management Frame Fields							
	Preamble	ST	OP	PHYAD	REGAD	TA	DATA	IDLE
Read	1…1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD	Z
Write	1…1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD	Z

名称	说明
Preamble	由 MAC 发送 32 个连续的逻辑“1”，同步于 MDC 信号，用于 MAC 与 PHY 之间的同步；
ST	帧开始位，固定为 01
OP	操作码，10 表示读，01 表示写
PHYAD	PHY 的地址，5 bits
REGAD	寄存器地址，5 bits
TA	Turn Around，MDIO 方向转换，在写状态下，不需要转换方向，值为 10，在读状态下，MAC 输出端为高阻态，在第二个周期，PHY 将 MDIO 拉低
DATA	共 16bits 数据
IDLE	空闲状态，此状态下 MDIO 为高阻态，由外部上拉电阻拉高

13.5.2. 读时序

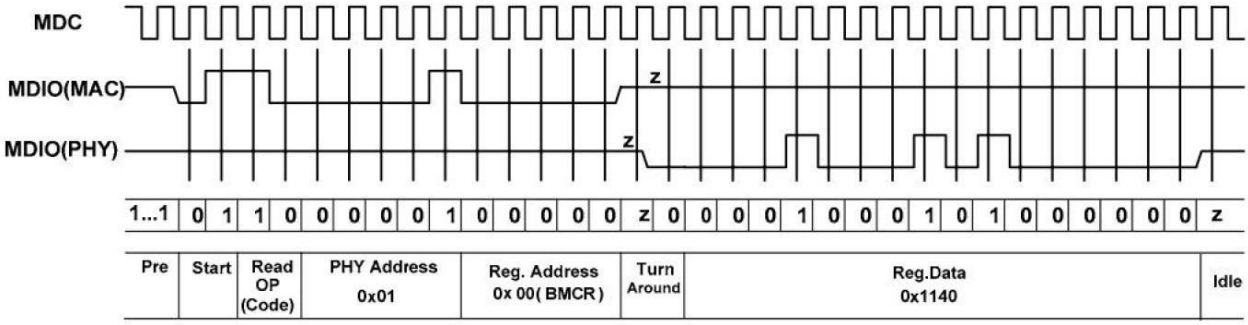


Figure 8. MDC/MDIO Read Timing

可以看到在 Turn Around 状态下，第一个周期 MDIO 为高阻态，第二个周期由 PHY 端拉

低。

13.5.3. 写时序

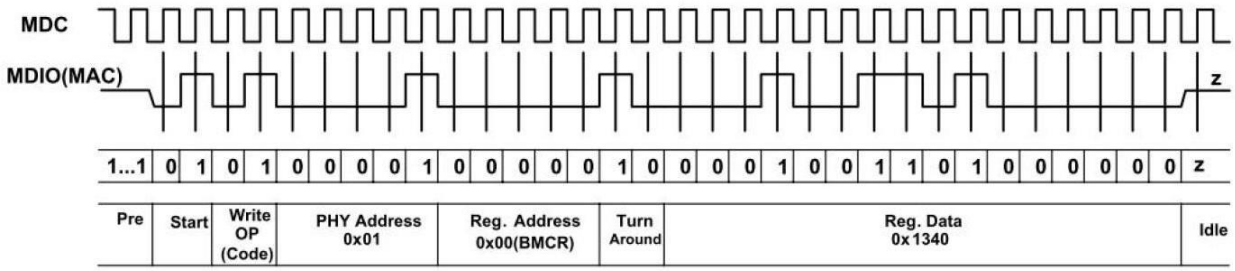


Figure 9. MDC/MDIO Write Timing

为了保证能够正确采集到数据，在 MDC 上升沿之前就把数据准备好，在本实验中为下降沿发送数据，上升沿接收数据。

13.6. 实验设计

本实验以千兆以太网 RGMII 通信为例来设计 verilog 程序，会先发送预设的 UDP 数据到网络，每秒钟发送一次.程序分为两部分，分别为发送和接收，实现了 ARP，UDP 功能。

13.6.1. 发送部分

13.6.1.1. MAC 层发送

发送部分中，mac_tx.v 为 MAC 层发送模块，首先在 SEND_START 状态，等待 mac_tx_ready 信号，如果有效，表明 IP 或 ARP 的数据已经准备好，可以开始发送。再进入发送前导码状态，结束时发送 mac_data_req，请求 IP 或 ARP 的数据，之后进入发送数据状态，最后进入发送 CRC 状态。在发送数据过程中，需要同时进行 CRC 校验。前导码完成后就将上层协议数据发送出去，这个时候同样把这些上层数据放到 C

RC32 模块中做序列生成，上层协议会给一个数据输出完成标志信号，这个时候 mac_tx 知道数据发送完成了，需要结束 CRC32 的序列生成，这个时候就开始提取 FCS，衔接数据之后发送出去。这样就连接了前导码---数据（Mac 帧）----FCS。之后跳转到结束状态，再回到 IDLE 状态，等待下一次的发送请求。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
crc_result	input	32	CRC32 结果
crcen	output	1	CRC 使能信号
crcre	output	1	CRC 复位信号
crc_din	output	8	CRC 模块输入信号
mac_frame_data	input	8	从 IP 或 ARP 来的数据
mac_tx_req	input	1	MAC 的发送请求
mac_tx_ready	input	1	IP 或 ARP 数据已准备好
mac_tx_end	input	1	IP 或 ARP 数据已经传输完毕
mac_tx_data	output	8	向 PHY 发送数据
mac_send_end	output	1	MAC 数据发送结束
mac_data_valid	output	1	MAC 数据有效信号，即 gmii_tx_en
mac_data_req	output	1	MAC 层向 IP 或 ARP 请求数据

13.6.1.2. MAC 发送模式

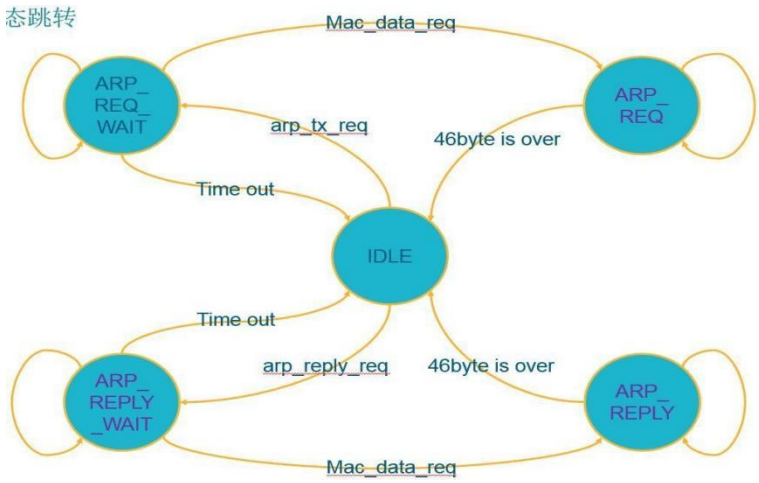
工程中的 mac_tx_mode.v 为发送模式选择，根据发送模式是 IP 或 ARP 选择相应的信号与数据。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位

mac_send_end	input	1	MAC 发送结束
arp_tx_req	input	1	ARP 发送请求
arp_tx_ready	input	1	ARP 数据已准备好
arp_tx_data	input	8	ARP 数据
arp_tx_end	input	1	ARP 数据发送到 MAC 层结束
arp_tx_ack	input	1	ARP 发送响应信号
ip_tx_req	input	1	IP 发送请求
ip_tx_ready	input	1	IP 数据已准备好
ip_tx_data	input	8	IP 数据
ip_tx_end	input	1	IP 数据发送到 MAC 层结束
mac_tx_ready	output	1	MAC 数据已准备好信号
ip_tx_ack	output	1	IP 发送响应信号
mac_tx_ack	input	1	MAC 发送响应信号
mac_tx_req	output	1	MAC 发送请求
mac_tx_data	output	8	MAC 发送数据
mac_tx_end	output	1	MAC 数据发送结束

13.6.1.3. ARP 发送

发送部分中，arp_tx.v 为 ARP 发送模块， 在 IDLE 状态下，等待 ARP 发送请求或 ARP 应答请求信号，之后进入请求或应答等待状态，并通知 MAC 层，数据已经准备好，等待 mac_data_req 信号，之后进入请求或应答数据发送状态。由于数据不足 46 字节，需要补全 46 字节发送。



信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
destination_mac_addr	input	48	发送的目的 MAC 地址
source_mac_addr	input	48	发送的源 MAC 地址
source_ip_addr	input	32	发送的源 IP 地址
destination_ip_addr	input	32	发送的目的 IP 地址
mac_data_req	input	1	MAC 层请求数据信号
arp_request_req	input	1	ARP 请求的请求信号
arp_reply_ack	output	1	ARP 回复的应答信号
arp_reply_req	input	1	ARP 回复的请求信号
arp_rec_source_ip_addr	input	32	ARP 接收的源 IP 地址，回复时放到目的 IP 地址
arp_rec_source_mac_addr	input	48	ARP 接收的源 MAC 地址，回复时放到目的 MAC 地址
mac_send_end	input	1	MAC 发送结束
mac_tx_ack	input	1	MAC 发送应答

arp_tx_ready	output	1	ARP 数据准备好
arp_tx_data	output	8	ARP 发送数据
arp_tx_end	output	1	ARP 数据发送结束
arp_tx_req	output	1	ARP 发送请求信号

13.6.1.4. IP 层发送

在发送部分，ip_tx.v 为 IP 层发送模块，在 IDLE 状态下，如果 ip_tx_req 有效，也就是 UDP 或 ICMP 发送请求信号，进入等待发送数据长度状态，之后进入产生校验和状态，校验和是将 IP 首部所有数据以 16 位相加，最后将进位再与低 16 位相加，直到进入为 0，再将低

16 位取反，得出校验和结果。

在生成校验和之后，等待 MAC 层数据请求，开始发送数据，并在即将结束发送 IP 首部后请求 UDP 或 ICMP 数据。等发送完，进入 IDLE 状态。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rstn	input	1	低电平复位
dest_mac_addr	input	48	发送的目的 MAC 地址
sour_mac_addr	input	48	发送的源 MAC 地址
sour_ip_addr	input	32	发送的源 IP 地址
dest_ip_addr	input	32	发送的目的 IP 地址
ttl	input	8	生存时间
ip_send_type	input	8	上层协议号，如 UDP,ICMP
upper_layer_data	output	8	从UDP 或 ICMP 过来的数据
upper_data_req	input	1	向上层请求数据
mac_tx_ack	input	1	MAC 发送应答
mac_send_end	input	1	MAC 发送结束信号

mac_data_req	input	1	MAC 层请求数据信号
upper_tx_ready	input	1	上层 UDP 或 ICMP 数据准备好
ip_tx_req	input	1	发送请求，从上层过来
ip_send_data_length	input	16	发送数据总长度
ip_tx_ack	output	1	产生 IP 发送应答
ip_tx_ready	output	1	IP 数据已准备好
ip_tx_data	output	8	IP 数据
ip_tx_end	output	1	IP 数据发送到 MAC 层结束

13.6.1.5. IP 发送模式

工程中的 ip_tx_mode.v 为发送模式选择，根据发送模式是 UDP 或 ICMP 选择相应的信号与数据。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
mac_send_end	input		MAC 数据发送结束
udp_tx_req	input	1	UDP 发送请求
udp_tx_ready	input	1	UDP 数据准备好
udp_tx_data	input	8	UDP 发送数据
udp_send_data_length	input	16	UDP 发送数据长度
udp_tx_ack	output	1	输出 UDP 发送应答
icmp_tx_req	input	1	ICMP 发送请求
icmp_tx_ready	input	1	ICMP 数据准备好
icmp_tx_data	input	8	ICMP 发送数据

icmp_send_data_length	input	16	ICMP 发送数据长度
icmp_tx_ack	output	1	ICMP 发送应答
ip_tx_ack	input	1	IP 发送应答
ip_tx_req	input	1	IP 发送请求
ip_tx_ready	output	1	IP 数据已准备好
ip_tx_data	output	8	IP 数据
ip_send_type	output	8	上层协议号，如 UDP,ICMP
ip_send_data_length	output	16	发送数据总长度

13.6.1.6. UDP 发送

发送部分中，udp_tx.v 为 UDP 发送模块。

信号名称	方向	位宽	说明
udp_send_clk	input	1	系统时钟
rstn	input	1	低电平复位
app_data_in_valid	input	1	从外部所接收的数据输出有效信号
app_data_in	input	8	外部所接收的数据
app_data_length	input	16	从外部所接收的当前数据包的长度（不含 udp、ip、mac 首部）
udp_dest_port	input	16	从外部所接收的数据包的源端口号
app_data_request	input	1	用户接口数据发送请求
udp_send_ready	output	1	UDP 数据发送准备
udp_send_ack	output	1	UDP 数据发送应当
ip_send_ready	input	1	IP 数据发送准备
ip_send_ack	input	1	IP 数据发送应当
udp_send_request	output	1	用户接口数据发送请求

udp_data_out_valid	output	1	发送的数据输出有效信号
udp_data_out	output	8	发送的数据输出
udp_packet_length	output	16	当前数据包的长度（不含 udp、 ip、mac 首部）

13.6.2. 接收部分

13.6.2.1. MAC 层接收

在接收部分，其中 mac_rx.v 为 mac 层接收文件，首先在 IDLE 状态下当 rx_en 信号为高，进入 REC_PREAMBLE 前导码状态，接收前导码。之后进入接收 MAC 头部状态，即目的 MAC 地址，源 MAC 地址，类型，将它们缓存起来，并在此状态判断前导码是否正确，错误则进入 REC_ERROR 错误状态，在 REC_IDENTIFY 状态判断类型是 IP（8'h0800）或 ARP(8'h0806)。然后进入接收数据状态，将数据传送到 IP 或 ARP 模块，等待 IP 或 ARP 数据接收完毕，再接收 CRC 数据。并在接收数据的过程中对接收的数据进行 CRC 处理，将结果与接收到的 CRC 数据进行对比，判断数据是否接收正确，正确则结束，错误则进入 ERROR 状态。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rstn	input	1	低电平复位
rx_en	input	1	开始接受使能
mac_rx_datain	input	8	接受的数据
checksum_err	input	1	IP 层校验错误信号
ip_rx_end	input	1	IP 接受结束
arp_rx_end	input	1	ARP 接受结束
ip_rx_req	output	1	IP 接受请求
arp_rx_req	input	1	请求 ARP 接收
mac_rx_dataout	output	8	MAC 层接收数据输出给 IP 或 ARP
mac_rec_error	output	1	MAC 层接收错误

mac_rx_dest_mac_addr	output	48	MAC 接收的目的 IP 地址
mac_rx_sour_mac_addr	output	48	MAC 接收的源 IP 地址

13.6.2.2. ARP 接收

工程中的 arp_rx.v 为 ARP 接收模块，实现 ARP 数据接收，在 IDLE 状态下，接收到从 MAC 层发来的 arp_rx_req 信号，进入 ARP 接收状态，在此状态下，提取出目的 MAC 地址，源 MAC 地址，目的 IP 地址，源 IP 地址，并判断操作码 OP 是请求还是应答。如果是请求，则判断接收到的目的 IP 地址是否为本机地址，如果是，发送应答请求信号 arp_reply_req，如果不是，则忽略。如果 OP 是应答，则判断接收到的目的 IP 地址及目的 MAC 地址是否与本机一致，如果是，则拉高 arp_found 信号，表明接收到了对方的地址。并将对方的 MAC 地址及 IP 地址存入 ARP 缓存中。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rstn	input	1	低电平复位
local_ip_addr	input	32	本地 IP 地址
local_mac_addr	input	48	本地 MAC 地址
arp_rx_data	input	8	ARP 接收数据
arp_rx_req	input	1	ARP 接收请求
arp_rx_end	output	1	ARP 接收完成
arp_reply_ack	input	1	ARP 回复应答
arp_reply_req	output	1	ARP 回复请求
arp_rec_sour_ip_addr	input	32	ARP 接收的源 IP 地址
arp_rec_sour_mac_addr	input	48	ARP 接收的源 MAC 地址
arp_found	output	1	ARP 接收到请求应答正确

13.6.2.3. IP 层接收模块

在工程中，ip_rx 为 IP 层接收模块，实现 IP 层的数据接收，信息提取，并进行校验和检查。首先在 IDLE 状态下，判断从 MAC 层发过来的 ip_rx_req 信号，进入接收 IP 首部状态，先在 REC_HEADER0 提取出首部长度及 IP 总长度，进入 REC_H

EADER1 状态，在此状态提取出目的 IP 地址，源 IP 地址，协议类型，根据协议类型发送 udp_rx_req 或 icmp_rx_req。在接收首部的同时进行校验和的检查，将首部接收的所有数据相加，存入 32 位寄存器，再将高 16 位

与低 16 位相加，直到高 16 位为 0 ，再将低 16 位取反，判断其是否为 0，如果是 0，则检验正确，否则错误，进入 IDLE 状态，丢弃此帧数据，等待下次接收。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
local_ip_addr	input	32	本地 IP 地址
local_mac_addr	input	48	本地 MAC 地址
ip_rx_data	input	8	从 MAC 层接收的数据
ip_rx_req	input	1	MAC 层发送的 IP 接收请求信号
mac_rx_destination_mac_addr	input	48	MAC 层接收的目的 MAC 地址
udp_rx_req	output	1	UDP 接收请求信号
icmp_rx_req	output	1	ICMP 接收请求信号
ip_addr_check_error	output	1	地址检查错误信号
upper_layer_data_length	output	16	上层协议的数据长度
ip_total_data_length	output	16	数据总长度
net_protocol	output	8	网络协议号
ip_rec_source_addr	output	32	IP 层接收的源 IP 地址
ip_rec_destination_addr	output	32	IP 层接收的目的 IP 地址
ip_rx_end	output	1	IP 层接收结束
ip_checksum_error	output	1	IP 层校验和检查错误信号

13.6.2.4. UDP 接收

在工程中，udp_rx.v 为 UDP 接收模块，在此模块首先接收 UDP 首部，再接收数据部分，在接收的同时进行 UDP 校验和检查，如果 UDP 数据是奇数个字节，在计算校验和时，在最后一个字节后加上 8'h00，并进行校验和计算。校验方法与 IP 校验和一样，如果校验正确，将拉高 udp_rec_data_valid 信号，表明接收的 UDP 数据有效，否则无效，等待下次接收。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
udp_rx_data	input	8	UDP 接收数据
udp_rx_req	input	1	UDP 接收请求
mac_rec_error	input	1	MAC 层接收错误
net_protocol	input	8	网络协议号
ip_rec_source_addr	input	32	IP 层接收的源 IP 地址
ip_rec_destination_addr	input	32	IP 层接收的目的 IP 地址
ip_checksum_error	input	1	IP 层校验和检查错误信号
ip_addr_check_error	input	1	地址检查错误信号
upper_layer_data_length	input	16	上层协议的数据长度
udp_rec_ram_rdata	output	8	UDP 接收 RAM 读数据
udp_rec_ram_read_addr	input	11	UDP 接收 RAM 读地址
udp_rec_data_length	output	16	UDP 接收数据长度
udp_rec_data_valid	output	1	UDP 接收数据有效

13.6.3. 其他部分

13.6.3.1. ICMP 应答

在工程中，icmp_reply.v 实现 ping 功能，首先接收其他设备发过来的 icmp 数据，

判断类型是否是回送请求（ECHO REQUEST），如果是，将数据存入 RAM，并计算校验和，判断校验和是否正确，如果正确则进入发送状态，将数据发送出去。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
mac_send_end	input	1	Mac 发送结束信号
ip_tx_ack	input	1	IP 发送应答
icmp_rx_data	input	8	ICMP 接收数据
icmp_rx_req	input	1	ICMP 接收请求
icmp_rev_error	input	1	接收错误信号
upper_layer_data_length	input	16	上层协议长度
icmp_data_req	output	1	发送请求 ICMP 数据
icmp_tx_ready	output	1	ICMP 发送准备好
icmp_tx_data	output	8	ICMP 发送数据
icmp_tx_end	output	1	ICMP 发送结束
icmp_tx_req	output	1	ICMP 发送请求

13.6.3.2. ARP 缓存

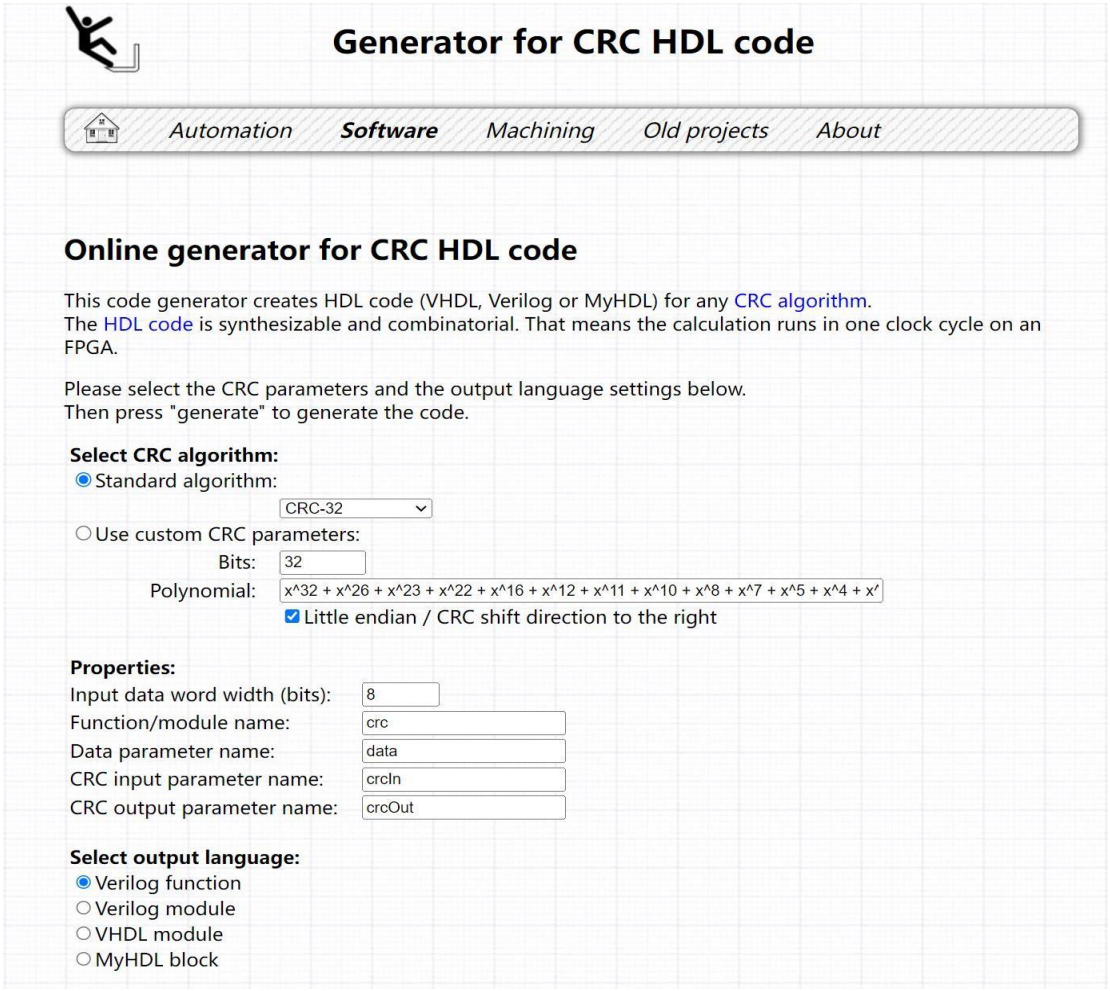
在工程中，arp_cache.v 为 arp 缓存模块，将接收到的其他设备 IP 地址和 MAC 地址缓存，在发送数据之前，查询目的地址是否存在，如果不存在，则向目的地址发送 ARP 请求，等待应答。在设计文件中，只做了一个缓存空间，如果有需要，可扩展。

信号名称	方向	位宽	说明
clk	input	1	系统时钟
rst_n	input	1	低电平复位
arp_found	input	1	ARP 接收到回复正确
arp_rec_source_ip_addr	input	32	ARP 接收的源 IP 地址

arp_rec_source_mac_addr	input	48	ARP 接收的源 MAC 地址
destination_ip_addr	input	32	目的 IP 地址
destination_mac_addr	output	48	目的 MAC 地址
mac_not_exist	output	1	目的地址对应的 MAC 地址不存在

13.6.3.3. CRC 校验模块(crc.v)

CRC32 校验是在目标 MAC 地址开始计算的，一直计算到一个包的最后一个数据为止。一些网站可以自动生成 CRC 算法的 verilog 文件：https://bues.ch/cms/hacking/crcgen.html



13.6.4. 实验现象

用网线连接 MES50HP 开发板网口 1 和 PC 端网口；设置接收端（PC 端）IP 地址为 192.168.1.105，开发板的 IP 地址为 192.168.1.11 如下图：

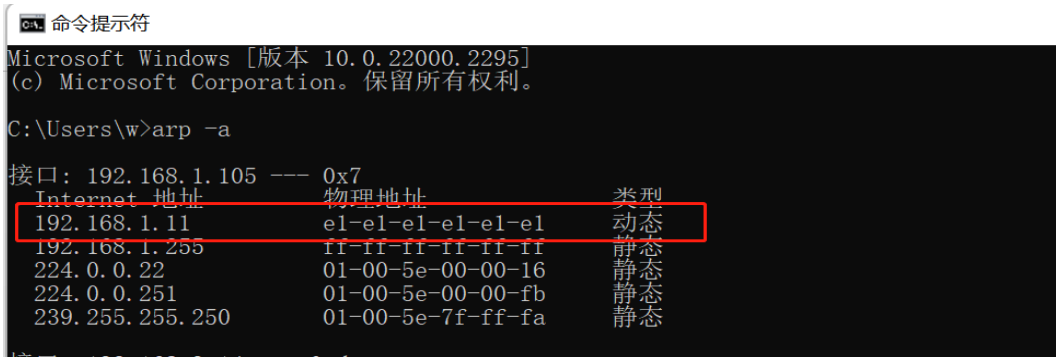
```
module ethernet1_test#(
    parameter       LOCAL_MAC  = 48'he1_e1_e1_e1_e1_e1,
    parameter       LOCAL_IP   = 32'hC0_A8_01_0B,//192.168.1.11
    parameter       LOCL_PORT  = 16'h1F90,

    parameter       DEST_IP    = 32'hC0_A8_01_69,//192.168.1.105
    parameter       DEST_PORT  = 16'h1F90
)(
```

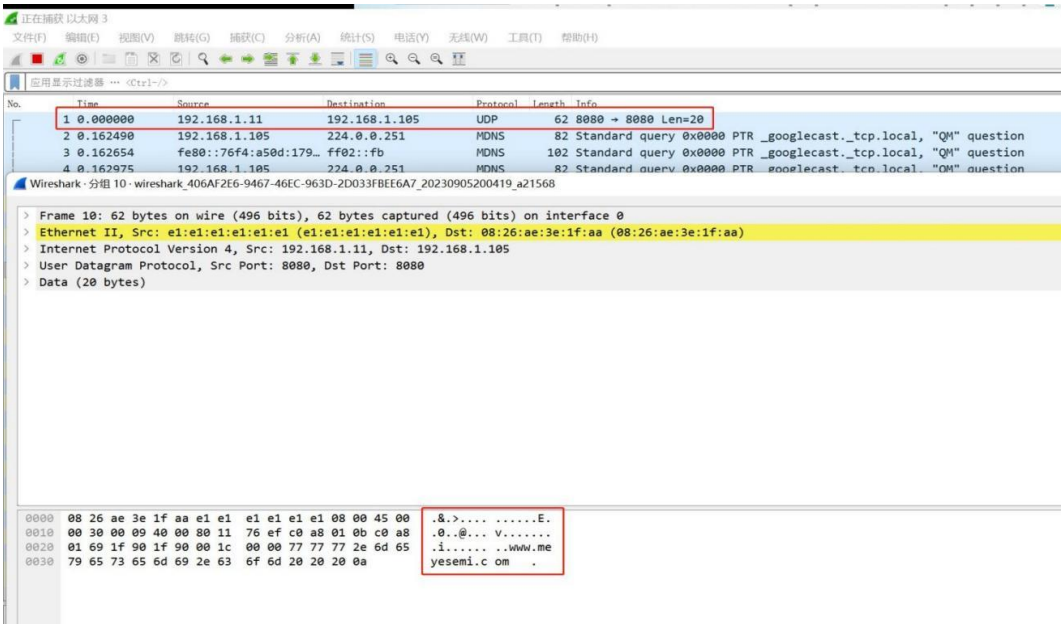
将程序下载到开发板后，便可以看到 LED 灯规律性闪烁：

信号名称	LED 编号	参考说明
led	1	闪烁

通过命令提示符，输入 arp -a，可以查到 IP：192.168.1.11　　MAC：e1_e1_e1_e1_e1_e1;



（注：目前只提供 Ethernet1）



成功建立连接后会持续发送数据报“www.meyesemi.com”

14. 灰度化实验例程

14.1. 实验简介

实验目的:

完成图像 RGB 到 YUV 的转换,输出灰度图像.。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

14.2. 实验原理

14.2.1. 图像格式介绍

1、RGB 格式:RGB 色彩就是常说的光学三原色，R 代表 Red（红色），G 代表 Green（绿色），B 代表 Blue（蓝色）,自然界中肉眼所能看到的任何色彩都可以由这三种色彩混合叠加而成，因此也称为加色模式。我们常用的 RGB 格式有 RGB565，RGB 888。其中 RGB565:每个像素用 16bit 来表示，占 2 个字节，第一字节的前 5 位是 R，后三位+第二字节前三位是 G，第二字节后 5 位是 B。RGB888:每个像素用 24bit 来表示，占 3 个字节，R、G、B、各占 8bit。还有一种是 RGB32 格式，在 RGB888 的基础上增加了透明度 Alpha，用 A 表示。

2、YUV(YCBCR)格式：常用于描述影像的饱和度和色调，也可以简化图像信息、去除冗余信息、提升计算效率。RGB 与 YUV 的转换实际上是色彩空间的转换，即将 RGB 的三原色色彩空间转换为 YUV 所表示的亮度与色度的色彩空间模型。Y 表示亮度、U 和 V 表示色度。而 YCbCr 是通过 YUV 信号的发展而来的，一般来说。这两种信号是用在不同的场景，但目前在一般情况下，两者是等价的。Y 还是表示亮度、Cb 表示蓝色分量、Cr 表示红色分量。一般有 YUV4:4:4,YUV4:2:2。其表示的含义就是各个分量采样率的不同。比如 YUV4:2:2，Y 的采样率是 U 和 V 的两倍。通过显示 Y 分量即可得到灰度图。

总结: RGB 着重于人眼对色彩的感应，YUV 则着重于视觉对于亮度的敏感程度。同时灰度图像只包含了亮度信息其大小在 0-255 之间，在某些情况下，可以减少计算所需的数据量，提高计算效率。除此之外还有很多别的格式，比如还有 RAW 格式显示的图像。

14.2.2. 计算公式介绍

常用的 RGB 转 YUV 的公式如下所示:

Y = 0.299R + 0.587G + 0.114B

Cb =− 0.172R − 0.339G + 0.511B + 128 Cr = 0.511R − 0.428G − 0.083B + 128

最后我们只需要显示 Y 分量，即可得到灰度图，但是我们可以发现，该公式涉及到大量的小数运算。在 Verilog 中，我们是不可能直接定义一个值为 0.299 的常量的。因此我们需要将将小数转为化整数。可以通过将公式整体扩大 256 倍，例如 0.299*256≈77。这样做会损失部分精度，要求高精度的话可以扩大更大的倍数，倍数越大，结果越准确。所以 0.299R 就转为 77R，最后再把计算结果>>8 位即可，相当于除以 256，我们通过移位操作可以大大降低运算难度。

具体公式如下所示:

Y = (77R + 150G + 29B) >> 8

Cb = (− 43R − 85G + 128B + 32768) >> 8 Cr = (128R − 107G − 21B + 32768) >> 8

该公式避免了小数也避免了除法，大大降低了运算的难度。

后续笔者会先通过 Matlab 完成图像的灰度化，先通过 Matlab 自带的灰度化函数来完成图像灰度化，再利用上述公式完成图像灰度化，然后再通过 Moldelsim 仿真得到灰度化后的 txt 数据，Matlab 再将其恢复为图片显示，最后对比灰度化的结果是否准确。

14.3. 接口列表

工程顶层模块，后面的工程均是这个模块为顶层，故后面的章节不再列出。以下为 RGB2YCbCr.v 模块的接口列表。

端口	I/O	位宽	描述
clk	input	1	像素时钟
rst_n	input	1	系统复位
vsync_in	input	1	处理前的视频场信号
hsync_in	input	1	处理前的视频行信号

de_in	input	1	数据有效信号
red	input	5	红色分量
greed	input	6	绿色分量
blue	input	5	蓝色分量
vsync_out	output	1	处理后的视频场信号
hsync_out	output	1	处理后的视频场信号
de_out	output	1	处理后的数据有效信号
y	output	8	亮度数据(灰度数据)
cb	output	8	蓝色色度数据(预留)
cr	output	8	红色色度数据(预留)

14.4. 工程说明

该图为本次工程框架，通过 HDMI 输入，经过 MS7000 芯片解码出 RGB 数据，然后通过灰度化模块输出 Y 分量，然后再经过 DDR3 缓存，最后读出到屏幕上显示。显示的分辨率为 1080P@60HZ。

14.5. 代码模块说明

主要介绍灰度化模块，具体如下所示：

```
1.  module RGB2YCbCr
2.  (
```

```
3.      //module clock
4.      input                   clk             ,
5.      input                   rst_n           ,
6.
7.
8.      input                   vsync_in ,    // vsync 信号
9.      input                   hsync_in ,    // hsync 信号
10.     input                   de_in    ,    //
11.     input       [4:0]    red          ,
12.     input       [5:0]    green        ,
13.     input       [4:0]    blue         ,
14.
15.
16.     output                  vsync_out,    // vsync 信号
17.     output                  hsync_out,    // hsync 信号
18.     output                  de_out   ,    // data enable 信号
19.     output      [7:0]    y            ,
20.     output      [7:0]    cb           ,
21.     output      [7:0]    cr
22. );
23.
24. //reg define
25. reg   [15:0]    rgb_r_m0, rgb_r_m1, rgb_r_m2;
26. reg   [15:0]    rgb_g_m0, rgb_g_m1, rgb_g_m2;
27. reg   [15:0]    rgb_b_m0, rgb_b_m1, rgb_b_m2;
28. reg   [15:0]    y0 ;
29. reg   [15:0]    cb0;
30. reg   [15:0]    cr0;
31. reg   [ 7:0]    y1 ;
32. reg   [ 7:0]    cb1;
33. reg   [ 7:0]    cr1;
34. reg   [ 2:0]    vsync_in_d;
35. reg   [ 2:0]    hsync_in_d;
36. reg   [ 2:0]    de_in_d    ;
37.
38. //wire define
39. wire [ 7:0]    rgb888_r;
40. wire [ 7:0]    rgb888_g;
41. wire [ 7:0]    rgb888_b;
42.
43.
44. //RGB565 to RGB 888
45. assign rgb888_r          = {red   , red[4:2]   };
46. assign rgb888_g          = {green, green[5:4]};
47. assign rgb888_b          = {blue , blue[4:2] };
48.
49.
50. assign vsync_out = vsync_in_d[2]       ;
51. assign hsync_out = hsync_in_d[2]       ;
52. assign de_out     = de_in_d[2]          ;
53. assign y              = hsync_out ? y1 : 8'd0;
54. assign cb             = hsync_out ? cb1: 8'd0;
55. assign cr             = hsync_out ? cr1: 8'd0;
56.
57. //----------------------------------------
58. //RGB 888 to YCbCr
59.
60. /*************************************************************
```

```
61.                 RGB888 to YCbCr
62.   Y   = 0.299R +0.587G + 0.114B
63.   Cb = 0.568(B-Y) + 128 = -0.172R-0.339G + 0.511B + 128
64.   CR = 0.713(R-Y) + 128 = 0.511R-0.428G -0.083B + 128
65.
66.   Y   = (77 *R     +     150*G     +     29 *B)>>8
67.   Cb = (-43*R     -     85 *G     +     128*B)>>8 + 128
68.   Cr = (128*R     -     107*G     -     21 *B)>>8 + 128
69.
70.   Y   = (77 *R     +     150*G     +     29 *B          )>>8
71.   Cb = (-43*R     -     85 *G     +     128*B + 32768)>>8
72.   Cr = (128*R     -     107*G     -     21 *B + 32768)>>8
73.  ************************************************************/
74.
75.  //step1 pipeline mult
76.  always @(posedge clk or negedge rst_n) begin
77.      if(!rst_n) begin
78.          rgb_r_m0 <= 16'd0;
79.          rgb_r_m1 <= 16'd0;
80.          rgb_r_m2 <= 16'd0;
81.          rgb_g_m0 <= 16'd0;
82.          rgb_g_m1 <= 16'd0;
83.          rgb_g_m2 <= 16'd0;
84.          rgb_b_m0 <= 16'd0;
85.          rgb_b_m1 <= 16'd0;
86.          rgb_b_m2 <= 16'd0;
87.      end
88.      else begin
89.          rgb_r_m0 <= rgb888_r * 8'd77 ;
90.          rgb_r_m1 <= rgb888_r * 8'd43 ;
91.          rgb_r_m2 <= rgb888_r << 3'd7 ;
92.          rgb_g_m0 <= rgb888_g * 8'd150;
93.          rgb_g_m1 <= rgb888_g * 8'd85 ;
94.          rgb_g_m2 <= rgb888_g * 8'd107;
95.          rgb_b_m0 <= rgb888_b * 8'd29 ;
96.          rgb_b_m1 <= rgb888_b << 3'd7 ;
97.          rgb_b_m2 <= rgb888_b * 8'd21 ;
98.      end
99.  end
100.
101.     //step2 pipeline add
102.     always @(posedge clk or negedge rst_n) begin
103.         if(!rst_n) begin
104.             y0   <= 16'd0;
105.             cb0 <= 16'd0;
106.             cr0 <= 16'd0;
107.         end
108.         else begin
109.             y0   <= rgb_r_m0 + rgb_g_m0 + rgb_b_m0;
110.             cb0 <= rgb_b_m1 - rgb_r_m1 - rgb_g_m1 + 16'd32768;
111.             cr0 <= rgb_r_m2 - rgb_g_m2 - rgb_b_m2 + 16'd32768;
112.         end
113.
114.     end
115.
116.     //step3 pipeline div
117.     always @(posedge clk or negedge rst_n) begin
118.         if(!rst_n) begin
```

```
119.              y1   <= 8'd0;
120.              cb1 <= 8'd0;
121.              cr1 <= 8'd0;
122.          end
123.          else begin
124.              y1   <= y0 [15:8];
125.              cb1 <= cb0[15:8];
126.              cr1 <= cr0[15:8];
127.          end
128.      end
129.
130.      //
131.      always@(posedge clk or negedge rst_n) begin
132.          if(!rst_n) begin
133.              vsync_in_d <= 3'd0;
134.              hsync_in_d <= 3'd0;
135.              de_in_d     <= 3'd0;
136.          end
137.          else begin
138.              vsync_in_d <= {vsync_in_d[1:0], vsync_in};
139.              hsync_in_d <= {hsync_in_d[1:0], hsync_in};
140.              de_in_d     <= {de_in_d[1:0]    , de_in    };
141.          end
142.      end
143.
144.      endmodule
```

该代码实现了 RGB565 到 YCbCr444 的格式转换，核心通过流水线结构分为乘法、加法和移位运算三个阶段，逐步完成 RGB888 转换到 Y、Cb、Cr 分量的计算。公式其实是非常简单的，部分人可能会直接 assign y = (77*R+150*G+29*B)>>8 来实现，这种写法乍一看好像没问题，实际上会造成非常大的延迟，因为这个组合逻辑需要计算三个乘法最后相加再移位，延迟是非常大的，非常容易造成时序违例，导致工作时钟频率降低。所以我们通过流水线的处理方式来改善该路径的延时，在中间穿插了三级流水线，分成三个 clk 来计算得出灰度化结果，可以有效避免时序违例。模块输入 50MHz 时钟与复位信号，同时接收来自图像接口的同步信号（vsync、hsync、de）和 16 位 RGB565 数据（红 5 位、绿 6 位、蓝 5 位），输出对应的 Y、Cb、Cr 信号以及同步控制信号。

首先在代码的 44-47 行，将输入的 RGB565 数据扩展成 RGB888。具体方法是通过位拼接扩展低位，例如红色通道 red[4:0] 转换成 8 位后为 {red, red[4:2]}，这样就保证了 RGB565 到 RGB888 的精度映射。绿、蓝通道同理处理。

代码的 50-55 行给出了输出同步信号和数据通路的控制。为了避免数据不稳定，模块在内部使用三拍寄存器延迟输入的 vsync、hsync 和 de 信号，最终将延迟后的信号作为 vsync_out、hsync_out、de_out 输出。同时在输出 Y、Cb、Cr 时增加条件判断，当 hsync_out 为高电平时输出对应的数据，否则输出 0，以保证数据只在有效行输出。

核心的 RGB 转 YCbCr 计算逻辑分为三步流水线。第一阶段（75-99 行）是乘法操作，将 RGB 三个分量分别与系数常量相乘，得到中间结果。可以看到该 always 块

的主要内容是完成乘法的计算，把公式里的所有乘法用一个时钟周期来计算得到，因此在数据有效的第一个时钟周期里，会先得到乘法的结果。如果是乘 128，可以用<<7 来代替，其余正常的用*来实现，也可以用乘法器来实现。还有要注意的是，因为整体都扩大了 256 倍，所以乘法得到的结果会扩大 256 倍，也就是<<8，位宽会扩大 8 为，所以定义寄存器的时候应该定义为[15:0]，避免溢出。例如 Y 分量对应的公式 0.299R + 0.587G + 0.114B 被转化为整数运算 (77*R + 150*G + 29*B)>>8，所以这里使用 77、150、29 作为权重。Cb 和 Cr 同理，分别对应不同的权重和偏移。代码中 rgb_r_m0、rgb_g_m0、rgb_b_m0 等寄存器就是保存这些乘法结果。

第二阶段（101-114 行）是加法和偏移操作，将第一阶段的乘法结果组合。该 always 块主要是完成加减法的内容，将每个乘法的结果全部相加，同样的，为了避免溢出，寄存器的位宽也定义为 16bit 即可。实际上为了避免减法得到负数，cb0 和 cr0 的计算应该判断系数之间的大小再进行减法，避免得到负数造成结果不正常，读者需要注意涉及到减法时，需要避免负数，因为我们的计算通常都是无符号的，出现负数会导致结果异常。Y 由 rgb_r_m0 + rgb_g_m0 + rgb_b_m0 得出，Cb 由 rgb_b_m1 - rgb_r_m1 - rgb_g_m1 + 32768 得出，Cr 由 rgb_r_m2 - rgb_g_m2 - rgb_b_m2 + 32768 得出。这里的 32768 相当于在移位前加上偏置值 128，用来保证 Cb 和 Cr 的中心值为 128。

第三阶段（116-128 行）是移位取高 8 位，相当于完成除以 256 的操作。最终得到 8 位的 Y、Cb、Cr 输出，分别存入 y1、cb1、cr1。

在 130-142 行，模块通过三级移位寄存器延迟同步信号 vsync_in、hsync_in 和 de_in，保证与数据通路保持一致的时序。这部分逻辑确保了输出的 YCbCr 信号和同步信号严格对齐，从而保证图像格式转换正确。

整体来看，该模块通过三阶段流水线结构完成 RGB 到 YCbCr 的转换，利用整数系数和移位操作代替浮点运算，保证了硬件实现的高效性。同时在时序上对同步信号和数据信号进行了统一延迟处理，确保输出图像数据与行场同步信号匹配。

14.6. 实验现象

连接好下载器，电源、HDMI_IN 口、HDMD_OUT 口，然后下载程序。如下所示：

上图为测试原图像。

上图为测试

15. 均值滤波实验例程

15.1. 实验简介

实验目的:

完成图像的均值滤波

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

15.2. 实验原理

15.2.1. 均值滤波概念介绍

均值滤波(Mean Filtering)是一种简单的图像处理技术,用于平滑图像和处理噪声。其基本思想是用像素点周围的邻域像素的平均值来代替该像素的值,其原理是使用一个固定大小的窗口(如 3x3 或 5x5)覆盖目标像素及其邻域,计算该区域内所有像素值的均值,并用这个均值替代窗口中心像素的值,再将窗口滑动到图像的每个像素位置重复该操作。

均值滤波简单易实现,计算量小,能够有效平滑图像和减少噪声,但同时也会导致图像细节和边缘模糊,可能会损失某些特征,因此对于需要保留边缘细节的应用,通常会选择更高级的滤波方法。

15.2.2. 均值滤波原理介绍

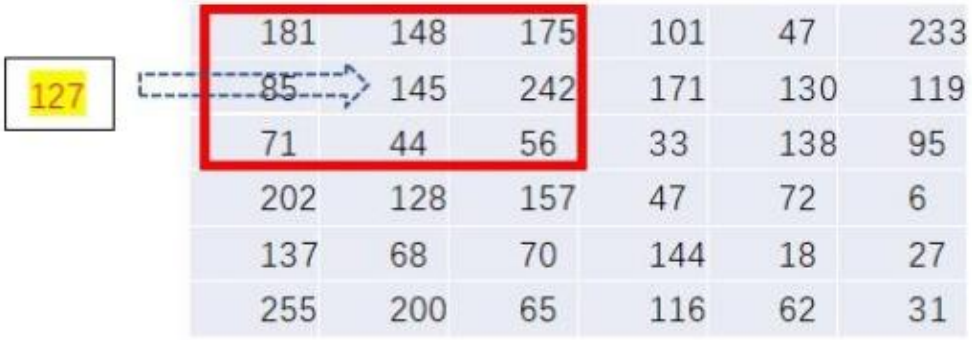
我们以 3x3 大小的区域为例子,它其实是将这个范围内的 9 个值进行求和的平均值,以这个新值来代替这个区域的中心值。我们配合下图进行理解:

可以看到 3x3 区域中中间像素值由 40 被替换为 107,其计算过程为:

$$(197+25+106+107+40+107+163+149+71)/9=107$$

接着移动 3x3 窗口对图片进行逐行逐列进行均值运算，即可完成图片的均值滤波操作具体操作过程如下：

假设有一张 8 位灰度图（其像素值区间在 0~255）进行第一次运算后值 145 替换为 127



第二次运算后 242 被替换为 123：

以此类推逐行逐列进行均值运算（其实就是使用一个 3x3 卷积核对整张图片进行卷积操作再除以 9，只不过卷积核为全 1），最终实现整张图片的滤波操作。

15.3. 接口列表

以下为 aver_filter.v 模块的接口列表

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	复位信号
matrix_de	input	1	矩阵数据输入行信号
matrix_vs	input	1	矩阵数据输入场信号
matrix11	input	8	第一行第一列矩阵数据

matrix12	input	8	第一行第二列矩阵数据
matrix13	input	8	第一行第三列矩阵数据
matrix21	input	8	第二行第一列矩阵数据
matrix22	input	8	第二行第二列矩阵数据
matrix23	input	8	第二行第三列矩阵数据
matrix31	input	8	第三行第一列矩阵数据
matrix32	input	8	第三行第二列矩阵数据
matrix33	input	8	第三行第三列矩阵数据
aver_filter_vs	output	1	均值滤波后数据场信号
aver _filter_de	output	1	均值滤波后数据行信号
aver _filter_data	output	8	均值滤波后数据

15.4. 工程说明

上图为本次工程架构，HDMI 输入后经过 MS7200 芯片解码出 RGB 数据，然后经过灰度化后进行均值滤波，然后将数据存到 DDR3 之中，然后再从 DDR3 中读出到 HDMI 上显示。

15.5. 代码模块说明

```
1.  //均值滤波
```

```
2.  module  aver_filter
3.  (
4.      input   wire            video_clk       /*synthesis PAP_MARK_DEBUG="1"*/,
5.      input   wire            rst_n           ,
6.
7.      //矩阵数据输入
8.      input   wire            matrix_de       /*synthesis PAP_MARK_DEBUG="1"*/,
9.      input   wire            matrix_vs       /*synthesis PAP_MARK_DEBUG="1"*/,
10.     input   wire    [7:0]   matrix11        ,
11.     input   wire    [7:0]   matrix12        ,
12.     input   wire    [7:0]   matrix13        ,
13.
14.     input   wire    [7:0]   matrix21        ,
15.     input   wire    [7:0]   matrix22        ,
16.     input   wire    [7:0]   matrix23        ,
17.
18.     input   wire    [7:0]   matrix31        ,
19.     input   wire    [7:0]   matrix32        ,
20.     input   wire    [7:0]   matrix33        ,
21.
22.     output  wire            aver_filter_vs  ,
23.     output  wire            aver_filter_de  ,
24.     output  wire    [7:0]   aver_filter_data
25.
26. );
27.
28. /*************************************************************
29. step1  每行相加   delay:1clk
30. *************************************************************/
31. reg [18:0]   line1_sum;
32. reg [18:0]   line2_sum;
33. reg [18:0]   line3_sum;
34. always@(posedge video_clk or negedge rst_n) begin
35.     if(!rst_n)
36.     begin
37.         line1_sum   <=  10'd0;
38.         line2_sum   <=  10'd0;
39.         line3_sum   <=  10'd0;
40.     end
41.     else    if(matrix_de)
42.     begin
43.         line1_sum   <=  matrix11 + matrix12 + matrix13  ;
44.         line2_sum   <=  matrix21 + matrix22 + matrix23  ;
45.         line3_sum   <=  matrix31 + matrix32 + matrix33  ;
46.     end
47.     else
48.     begin
49.         line1_sum   <=  10'd0;
50.         line2_sum   <=  10'd0;
51.         line3_sum   <=  10'd0;
52.     end
53. end
54.
55. /*************************************************************
56. step2  矩阵总和  delay:1clk
57. *************************************************************/
58. reg [18:0]   data_sum;
59. always@(posedge video_clk or negedge rst_n) begin
```

```
60.      if(!rst_n)
61.          data_sum     <=    12'd0;
62.      else
63.          data_sum     <=    line1_sum + line2_sum + line3_sum;
64.  end
65.
66.  /*************************************************************
67.  step3  求均值  /9 delay:1clk
68.  *************************************************************/
69.  //除法转乘法    *228>>11
70.  reg [18:0]   aver_filter_mux ;    //均值
71.
72.  always@(posedge video_clk or negedge rst_n) begin
73.      if(!rst_n)
74.          aver_filter_mux <=    19'd0;
75.      else
76.          aver_filter_mux <=    data_sum * 228; //后续要>>11
77.  end
78.
79.
80.
81.  /*************************************************************
82.  step4  中心像素点与阈值进行比较    delay:1clk
83.  *************************************************************/
84.  reg [7:0]    aver_filter_reg;
85.  always@(posedge video_clk or negedge rst_n) begin
86.      if(!rst_n)
87.          aver_filter_reg <=    1'd0;
88.      else
89.          aver_filter_reg <=    aver_filter_mux[18:11];
90.  end
91.
92.
93.
94.
95.  /*************************************************************
96.  时钟延迟  一共延迟  4clk
97.  *************************************************************/
98.  reg [3:0]    video_de_reg;
99.  reg [3:0]    video_vs_reg;
100.
101.     always@(posedge video_clk or negedge rst_n) begin
102.         if(!rst_n)
103.         begin
104.             video_de_reg     <=    4'd0;
105.             video_vs_reg     <=    4'd0;
106.         end
107.         else
108.         begin
109.             video_de_reg     <=    {video_de_reg[2:0],matrix_de};
110.             video_vs_reg     <=    {video_vs_reg[2:0],matrix_vs};
111.         end
112.     end
113.
114.     assign   aver_filter_vs       =    video_vs_reg[3] ;
115.     assign   aver_filter_de       =    video_de_reg[3] ;
116.     assign   aver_filter_data     =    aver_filter_reg ;
117.
```

```
118.     endmodule
119.
```

该模块实现了对 3×3 矩阵图像数据的均值滤波运算，用于图像去噪和平滑处理。输入为逐像素的 3×3 矩阵数据（matrix11 ~ matrix33）以及行场同步信号 matrix_de、matrix_vs，输出为滤波后的像素数据 aver_filter_data，并同步输出行场控制信号 aver_filter_de、aver_filter_vs。模块采用流水线结构，整体处理延迟为 4 个时钟周期。

在代码的 31-53 行，完成了矩阵每一行的加法求和操作。分别将 matrix11~matrix13、matrix21~matrix23、matrix31~matrix33 三组像素点求和，得到 line1_sum、line2_sum 和 line3_sum。该步骤在 matrix_de 有效时进行，否则输出清零，保证无效数据不被处理。

代码的 58-64 行完成矩阵的总和计算，即将三行的求和结果相加，得到 3×3 矩阵内所有像素点的总和 data_sum。这一步延迟 1 个时钟周期。

在 69-77 行，实现了均值计算。由于除法在硬件中开销较大，这里采用乘法加移位的方式代替。公式为 data_sum / 9 = (data_sum * 228) >> 11，其中乘以 228 后右移 11 位等效于除以 9。结果存入寄存器 aver_filter_mux。

接着在 84-90 行，将求得的均值结果取出高 8 位，得到最终的均值像素值 aver_filter_reg，即滤波后的输出像素。该过程延迟 1 个时钟周期。

最后在 95-116 行，模块对输入的行场同步信号 matrix_de、matrix_vs 进行 4 拍移位寄存器延迟，以补偿滤波数据通路中的延迟。最终通过 video_de_reg[3]、video_vs_reg[3] 输出与数据对齐的 aver_filter_de 和 aver_filter_vs 信号，并将滤波结果 aver_filter_reg 输出为 aver_filter_data。

综上所述，该模块通过流水线结构依次完成行求和、总和、均值计算和输出对齐，能够实时对输入的 3×3 矩阵像素块进行均值滤波运算，有效抑制图像噪声并平滑边缘，整体输出相对于输入延迟 4 个时钟周期。

15.6. 实验现象

连接好下载器，电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器，然后下载程序。

上图为添加了椒盐噪声的图像。

可以看到椒盐噪声变的平滑，比原图像有所改善，但效果比中值滤波差，均值滤波主要让图像平滑，无法过滤掉椒盐噪声。

16. Sobel 边缘检测实验例程

16.1. 实验简介

实验目的:

在图像灰度化的基础上完成基于 sobel 算子的边沿检测。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

16.2. 实验原理

sobel 算子是广泛应用的微分算子之一，可以计算图像处理中的边缘检测，计算图像的灰度地图。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量原理就是基于图像的卷积来实现在水平方向与垂直方向检测对于方向上的边缘。

水平方向的梯度算子的计算公式为:

$$G_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix} * A$$

垂直方向的梯度算子的计算公式为:

$$G_Y = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix} * A$$

最后再把水平方向的算子和垂直方向的算子平方相加再开根号即可得到结果。如下所示:

$$G = \sqrt{G_x^2 + G_y^2}$$

平方的操作在 fpga 中还是比较容易实现的，开根号的话一般使用 cordic IP 来完成 sqrt 操作，但紫光的软件中没有提供这种 IP 核，因此需要手写一个 sqrt 的模块。但是我们将该公式变换一下，根据不等式(有条件限制，可以做近似计算)可以换成以下公式:

$$G = |GX + GY|$$

可以大大降低所消耗的资源，当然最后的误差可能会比较大，可以通过调节阈值来弥补，追求完美的效果的话，还是需要使用 cordic 来完成，或者开源的 sart 代码来完

成。

最后计算得到的结果 G 与阈值比较，大于阈值视为边沿，可以将其像素值置为 0 或者 255，取决于想把边沿定位什么颜色，以 8bit 灰度数据为例子，255 就是白色，0 就是黑色

16.3. 接口列表

sobel.v 模块接口列表

端口	I/O	位宽	描述
SOBEL_THRESHOLD	/	8	sobel 阈值
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
matrix_de	input	1	矩阵数据有效信号
matrix_vs	input	1	视频场信号
matrix11	input	8	3X3 矩阵第 1 个 8bit 灰度像素
matrix12	input	8	3X3 矩阵第 2 个 8bit 灰度像素
matrix13	input	8	3X3 矩阵第 3 个 8bit 灰度像素
matrix21	input	8	3X3 矩阵第 4 个 8bit 灰度像素
matrix22	input	8	3X3 矩阵第 5 个 8bit 灰度像素
matrix23	input	8	3X3 矩阵第 6 个 8bit 灰度像素
matrix31	input	8	3X3 矩阵第 7 个 8bit 灰度像素
matrix32	input	8	3X3 矩阵第 8 个 8bit 灰度像素
matrix33	input	8	3X3 矩阵第 9 个 8bit 灰度像素

sobel_vs	output	1	输出 sobel 处理后的场信号
sobel_de	output	1	输出 sobel 处理后的数据有效信号
sobel_data	output	8	输出 sobel 处理后的数据

16.4. 工程说明

该工程架构与之前的工程基本一致，只是在灰度化后，新增了 sobel 处理模块，然后再将数据缓存到 DDR3 之中，然后再把数据从 DDR3 中读出到 HDMI 显示器上显示.

16.5. 代码模块说明

```
1.  //不开方根 3clk
2.  module  sobel
3.  #(
4.      parameter    SOBEL_THRESHOLD = 28
5.  )
6.  (
7.      input        wire              video_clk       ,
8.      input        wire              rst_n           ,
9.
10.     //矩阵数据输入
11.     input        wire              matrix_de       ,
12.     input        wire              matrix_vs       ,
13.     input        wire    [7:0]     matrix11        ,
14.     input        wire    [7:0]     matrix12        ,
15.     input        wire    [7:0]     matrix13        ,
16.
17.     input        wire    [7:0]     matrix21        ,
18.     input        wire    [7:0]     matrix22        ,
19.     input        wire    [7:0]     matrix23        ,
20.
21.     input        wire    [7:0]     matrix31        ,
22.     input        wire    [7:0]     matrix32        ,
23.     input        wire    [7:0]     matrix33        ,
```

```
    //sobel 数据输出
    output      wire                sobel_vs        ,
    output      wire                sobel_de        ,
    output      wire    [7:0]       sobel_data
);

/*****************************************************************
%       -1   0  +1     %       +1   2  +1
% gx = -2    0  +2     % gy =   0    0  0
%       -1   0  +1     %       -1  -2  -1
*****************************************************************/

/*****************************************************************
wire define
*****************************************************************/


/*****************************************************************
reg define
*****************************************************************/
reg [9:0]   gx_temp1;
reg [9:0]   gx_temp2;
reg [9:0]   gy_temp1;
reg [9:0]   gy_temp2;


/*****************************************************************
step1  计算卷积
*****************************************************************/
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        gx_temp1    <=  9'd0;
        gx_temp2    <=  9'd0;
    end
    else    if(matrix_de)
    begin
        gx_temp1    <=  matrix13 + 2*matrix23 + matrix33;
        gx_temp2    <=  matrix11 + 2*matrix21 + matrix31;
    end
    else
    begin
        gx_temp1    <=  9'd0;
        gx_temp2    <=  9'd0;
    end
end


always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        gy_temp1    <=  9'd0;
```

```
82.             gy_temp2     <=    9'd0;
83.         end
84.         else     if(matrix_de)
85.         begin
86.             gy_temp1     <=    matrix13 + 2*matrix23 + matrix33;
87.             gy_temp2     <=    matrix11 + 2*matrix21 + matrix31;
88.         end
89.         else
90.         begin
91.             gy_temp1     <=    9'd0;
92.             gy_temp2     <=    9'd0;
93.         end
94.     end
95.
96.     /*************************************************************************
97.     step2  求卷积和
98.     *************************************************************************/
99.     reg [9:0]    gx_data;
100.        reg [9:0]    gy_data;
101.
102.
103.        always@(posedge video_clk or negedge rst_n) begin
104.            if(!rst_n)
105.                gx_data <=    10'd0;
106.            else     if(gx_temp1 >= gx_temp2)
107.                gx_data <=    gx_temp1 - gx_temp2;
108.            else
109.                gx_data <=    gx_temp2 - gx_temp1;
110.        end
111.
112.        always@(posedge video_clk or negedge rst_n) begin
113.            if(!rst_n)
114.                gy_data <=    10'd0;
115.            else     if(gy_temp1 >= gy_temp2)
116.                gy_data <=    gy_temp1 - gy_temp2;
117.            else
118.                gy_data <=    gy_temp2 - gy_temp1;
119.        end
120.
121.        /*************************************************************************
122.        step3  绝对值相加
123.        *************************************************************************/
124.        reg [10:0]   sobel_data_reg;
125.
126.        always@(posedge video_clk or negedge rst_n) begin
127.            if(!rst_n)
128.                sobel_data_reg   <=    11'd0;
129.            else
130.                sobel_data_reg   <=    gx_data + gy_data;
131.        end
132.
133.
134.        /*************************************************************************
135.        时钟延迟  一共延迟  3clk
136.        *************************************************************************/
137.        reg [2:0]    video_de_reg;
138.        reg [2:0]    video_vs_reg;
139.
```

```
140.    always@(posedge video_clk or negedge rst_n) begin
141.        if(!rst_n)
142.        begin
143.            video_de_reg    <=   3'd0;
144.            video_vs_reg    <=   3'd0;
145.        end
146.        else
147.        begin
148.            video_de_reg    <=   {video_de_reg[1:0],matrix_de};
149.            video_vs_reg    <=   {video_vs_reg[1:0],matrix_vs};
150.        end
151.    end
152.
153.    assign  sobel_vs        =   video_vs_reg[2]          ;
154.    assign  sobel_de        =   video_de_reg[2]          ;
155.    assign  sobel_data      =   (sobel_data_reg>=SOBEL_THRESHOLD)?8'd255:8'd0    ;
156.
157.    endmodule
158.
```

module sobel 模块实现了基于 Sobel 算子的边缘检测功能。输入为 3×3 矩阵窗口内的像素点（matrix11~matrix33）以及有效信号 matrix_de、matrix_vs，输出为经过 Sobel 运算后的边缘检测结果 sobel_data，同时输出与其同步的行场控制信号 sobel_de、sobel_vs。该模块的运算总延迟为 3 个时钟周期。

在代码的 59-75 行，完成了水平方向梯度的计算。Sobel 算子在水平方向上的卷积核为 [-1 0 +1; -2 0 +2; -1 0 +1]，其本质是右边像素加权求和减去左边像素加权求和。实现时，gx_temp1 表示右半部分加权和 (matrix13 + 2×matrix23 + matrix33)，gx_temp2 表示左半部分加权和 (matrix11 + 2×matrix21 + matrix31)。

在 78-94 行，完成了垂直方向梯度的计算。Sobel 算子在垂直方向上的卷积核为 [+1 +2 +1; 0 0 0; -1 -2 -1]，其本质是下方像素加权求和减去上方像素加权求和。实现时，gy_temp1 表示下半部分加权和，gy_temp2 表示上半部分加权和。

在 103-119 行，计算梯度的绝对值。gx_data 为水平方向的梯度幅值，gy_data 为垂直方向的梯度幅值。通过比较大小并取差值的方式，得到两方向的绝对值结果。

在 124-131 行，将水平方向和垂直方向的梯度幅值相加，得到 Sobel 算子的近似梯度强度 sobel_data_reg，即边缘强度。

在 137-150 行，对输入的同步信号 matrix_de 和 matrix_vs 进行移位寄存器延迟补偿，使输出信号与数据结果对齐。模块的总延迟为 3 个时钟周期。

最后在 153-155 行，输出边缘检测结果。sobel_data 通过与设定的阈值 SOBEL_THRESHOLD 比较，若梯度强度大于等于阈值则输出 255（白色像素，表示边缘），否则输出 0（黑色像素，表示非边缘区域）。通过这种方式实现了二值化的边缘图像输出。

整体上模块通过 Sobel 算子在水平和垂直方向上的加权差分计算，实现了图像边缘特征的提取，并结合阈值比较完成边缘检测。模块结构比较简单，运算延迟固定为 3 个时钟周期，适用于图像预处理与特征提取场景。

16.6. 实验现象

上图为测试原图像。

上图为 sobel 算子的边沿检测效果，读者可以调节阈值来改善效果。

17. 高斯滤波实验例程

17.1. 实验简介

实验目的:

生成 3x3 矩阵，完成高斯滤波。

实验环境:

Window11

PDS2023.2-SP3-ads

硬件环境:

PT2G390H 开发板

17.2. 实验原理

17.2.1. 高斯滤波概念介绍

高斯噪声指的是概率密度函数服从高斯分布（即正态分布）的一类噪声。如果一个噪声，它的幅度分布服从高斯分布，而它的功率谱密度又是均匀分布的，则称它为高斯白噪声。在图像处理领域中，由于环境温度，不良光照等等原因可能使得相机 cmos 采集到的信号引入高斯噪声。

左图为正常灰度图，右图为添加了高斯噪声的灰度图

高斯滤波是一种常用的线性平滑滤波，适用于消除高斯噪声。 通俗的讲，高斯滤波就是通过设计卷积核，通过卷积运算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。

对于均值滤波和中值滤波来说，其进行矩阵划分的每个像素的权重是相等的。而在高斯滤波中，会将中心点的权重值加大，远离中心点的权重值减小，在此基础上计算邻域内各个像素值不同权重的和。

高斯滤波卷积核设计，首先介绍二维高斯函数:

$$f(x,y) = A\exp\left(-\left(\frac{(x-x_o)^2}{2\sigma_X^2} + \frac{(y-y_o)^2}{2\sigma_Y^2}\right)\right).$$

这是二维高斯概率密度分布函数，其表示随机变量在二维平面服从高斯分布，G(x, y) 代表二维平面上相应坐标点的概率密度，x 表示二维平面中的 x 轴方向，其中 y 表示二维平面中的 y 轴方向， ux ， uy 分别为 x 方向， y 方向随机变量的数学期望， □x ， □ y 分别为 x 方向，y 方向随机变量的方差。

假定卷积核中心点坐标为（0，0），那么它相邻的八个点坐标如下：

(-1，1)	(0，1)	(1，1)
(-1，0)	(0，0)	(1，0)
(-1，-1)	(0，-1)	(1，-1)

选取随机变量数学期望 u =0，方差□ =0.8 将坐标带入二维高斯概率密度分布函数得：

0.05212607	0.11385381	0.05212607
0.11385381	0.24867959	0.11385381
0.05212607	0.11385381	0.05212607

可以观察出中心概率密度最大，且概率密度沿四周逐渐减小，接下来对其进行归一化（卷积核的所有权重值都需要缩放到相同的范围内，并且总和为 1，以保证卷积操作对图像的影响是平衡的，避免卷积过后改变原来图像的亮度，使之偏亮或者偏暗）：

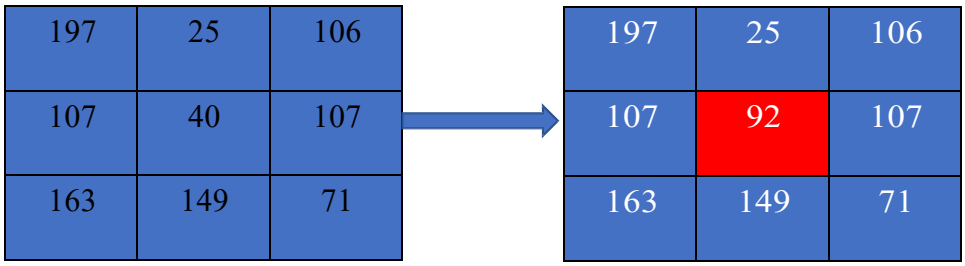
0.05711825	0.12475774	0.05711825
0.12475774	0.27249597	0.12475774
0.05711825	0.12475774	0.05711825

最后由于在 FPGA 中对小数的运算较为复杂，这里将卷积核扩大 16 倍取近似的整数，FPGA 卷积完成后再将数据缩小 16 倍，以达到避开浮点数运算的目的，近似后的卷积核为：

1	2	1
2	4	2
1	2	1

17.2.2. 高斯滤波 FPGA 介绍

与均值滤波相同只是使用了不同的 3x3 卷积核对图片逐行逐列进行卷积操作再除以 16。我们配合下图来理解，假设如下左图是一个 3x3 的像素点区域，使用我们设计的卷积核进行高斯滤波计算后，中间的值被替换为 92：



具体计算过程为：

（197+106+163+71）+（25+107+107+149）*2+40*4□=92

然后使用这个方式对整张图片进行逐行逐列的操作，即可完成对图片的高斯滤波操作。

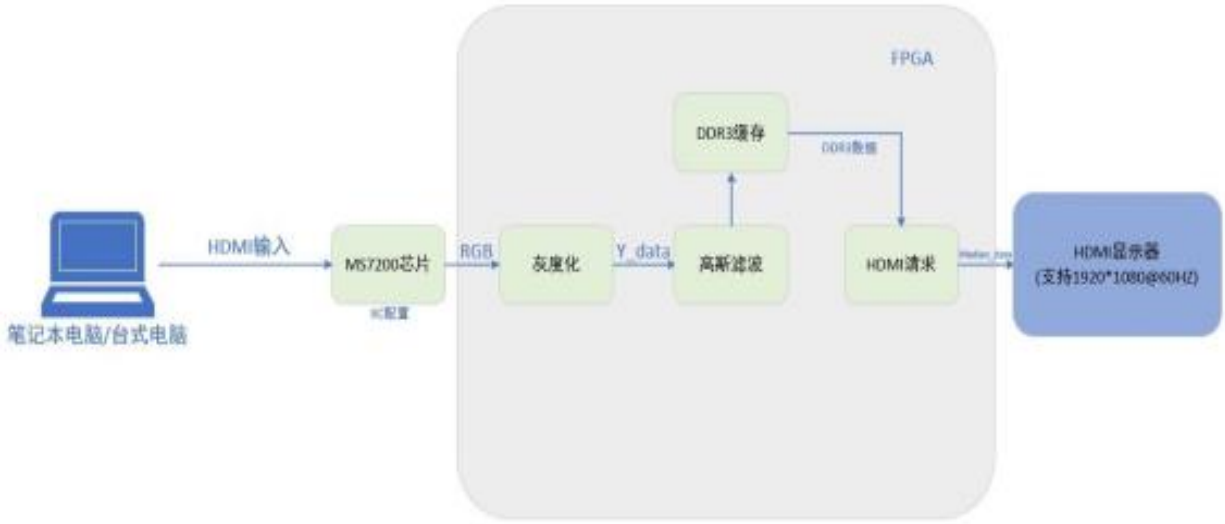
17.3. 接口列表

以下为 gauss_filter.v 模块的接口列表。

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	复位信号
matrix_de	input	1	矩阵数据输入行信号
matrix_vs	input	1	矩阵数据输入场信号
matrix11	input	8	第一行第一列矩阵数据
matrix12	input	8	第一行第二列矩阵数据
matrix13	input	8	第一行第三列矩阵数据
matrix21	input	8	第二行第一列矩阵数据
matrix22	input	8	第二行第二列矩阵数据

matrix23	input	8	第二行第三列矩阵数据
matrix31	input	8	第三行第一列矩阵数据
matrix32	input	8	第三行第二列矩阵数据
matrix33	input	8	第三行第三列矩阵数据
gauss_filter_vs	output	1	高斯滤波后数据场信号
gauss_filter_de	output	1	高斯滤波后数据行信号
gauss_filter_data	output	8	高斯滤波后数据

17.4. 工程说明



17.5. 代码模块说明

```
1.  //高斯滤波
2.  module  gauss_filter
3.  (
4.      input   wire            video_clk       ,
5.      input   wire            rst_n           ,
6.
7.      //矩阵数据输入
8.      input   wire            matrix_de       ,
9.      input   wire            matrix_vs       ,
10.     input   wire    [7:0]   matrix11        ,
11.     input   wire    [7:0]   matrix12        ,
12.     input   wire    [7:0]   matrix13        ,
13.
```

```
    input   wire    [7:0]   matrix21        ,
    input   wire    [7:0]   matrix22        ,
    input   wire    [7:0]   matrix23        ,

    input   wire    [7:0]   matrix31        ,
    input   wire    [7:0]   matrix32        ,
    input   wire    [7:0]   matrix33        ,

    output  wire            gauss_filter_vs ,
    output  wire            gauss_filter_de ,
    output  wire    [7:0]   gauss_filter_data

);
/*
高斯核
| 1 2 1 |
| 2 4 2 |          sum 高斯核  = 16   ->    sum/16=sum>>4
| 1 2 1 |
*/

/*****************************************************************
step1  每行相加   delay:1clk
*****************************************************************/
reg [11:0]  line1_sum;
reg [11:0]  line2_sum;
reg [11:0]  line3_sum;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        line1_sum   <=  12'd0;
        line2_sum   <=  12'd0;
        line3_sum   <=  12'd0;
    end
    else    if(matrix_de)
    begin
        line1_sum   <=  matrix11 + matrix12*2 + matrix13    ;
        line2_sum   <=  matrix21*2 + matrix22*4 + matrix23*2    ;
        line3_sum   <=  matrix31 + matrix32*2 + matrix33    ;
    end
    else
    begin
        line1_sum   <=  12'd0;
        line2_sum   <=  12'd0;
        line3_sum   <=  12'd0;
    end
end

/*****************************************************************
step2  矩阵总和  delay:1clk
*****************************************************************/
reg [11:0]  data_sum;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        data_sum    <=  12'd0;
    else
        data_sum    <=  line1_sum + line2_sum + line3_sum;
end

```

```
72.  /*************************************************************
73.  step3  右移 4 位  delay:1clk
74.  *************************************************************/
75.  //移位实现/16
76.  reg [7:0]    gauss_filter_reg     ;    //均值
77.
78.  always@(posedge video_clk or negedge rst_n) begin
79.      if(!rst_n)
80.          gauss_filter_reg     <=    8'd0;
81.      else
82.          gauss_filter_reg     <=    data_sum[11:4]; //
83.  end
84.
85.
86.  /*************************************************************
87.  时钟延迟  一共延迟  3clk
88.  *************************************************************/
89.  reg [2:0]    video_de_reg;
90.  reg [2:0]    video_vs_reg;
91.
92.  always@(posedge video_clk or negedge rst_n) begin
93.      if(!rst_n)
94.      begin
95.          video_de_reg     <=    3'd0;
96.          video_vs_reg     <=    3'd0;
97.      end
98.      else
99.      begin
100.             video_de_reg     <=    {video_de_reg[1:0],matrix_de};
101.             video_vs_reg     <=    {video_vs_reg[1:0],matrix_vs};
102.         end
103.     end
104.
105.     assign   gauss_filter_vs      =    video_vs_reg[2] ;
106.     assign   gauss_filter_de      =    video_de_reg[2] ;
107.     assign   gauss_filter_data    =    gauss_filter_reg     ;
108.
109.
110.     endmodule
111.
```

module gauss_filter 实现了基于 3×3 高斯核的图像平滑处理。输入为窗口内的 9 个像素点（matrix11~matrix33）以及同步信号 matrix_de、matrix_vs，输出为经过高斯滤波后的像素值 gauss_filter_data，同时输出与其对应的同步信号 gauss_filter_de、gauss_filter_vs。整体延迟为 3 个时钟周期。

在代码的 27~32 行定义了高斯卷积核，其权重矩阵为：

1	2	1
2	4	2
1	2	1

所有权重加和为 16，因此最终结果需要除以 16，相当于右移 4 位（>>4）。这种权重分布能够在平滑图像的同时，保留更多边缘信息，相比均值滤波效果更自然。

在 40~59 行，实现了对每一行像素的加权求和。line1_sum 对应第一行的加权和

(matrix11 + 2×matrix12 + matrix13)，line2_sum 对应第二行的加权和 (2×matrix21 + 4×matrix22 + 2×matrix23)，line3_sum 对应第三行的加权和 (matrix31 + 2×matrix32 + matrix33)。这一部分完成了行级加权累加。

在 64~70 行，将三行的加权结果进行累加，得到整个 3×3 窗口的高斯加权总和 data_sum。

在 76~83 行，通过右移 4 位实现除以 16 的操作，从而得到最终的高斯滤波输出值 gauss_filter_reg。采用移位运算代替除法运算，大大简化了硬件逻辑，提升了运算效率。

在 89~103 行，对输入的同步信号 matrix_de 和 matrix_vs 进行了移位寄存器延迟补偿，使输出结果与对应的控制信号对齐。由于滤波运算包含 3 个时钟周期的流水线延迟，因此同步信号也需延迟 3 个周期。

在 105~107 行，输出高斯滤波后的像素值 gauss_filter_data 以及对应的同步信号 gauss_filter_de、gauss_filter_vs。最终得到的图像是经过高斯卷积平滑处理后的结果，可用于降低噪声、提高后续边缘检测的稳定性。

最后声明了中间寄存器使用移位拼接的方式对矩阵数据的行场同步信号进行延迟操作一共延迟了 3 个时钟周期使得输出的高斯滤波数据与延迟后的行场同步信号相匹配。整个模块代码简洁高效，采用固定的 3×3 高斯核实现滤波操作，逻辑简洁，硬件开销低，能够在视频图像处理中有效去除高频噪声，同时保留主要轮廓信息。

17.6. 实验现象

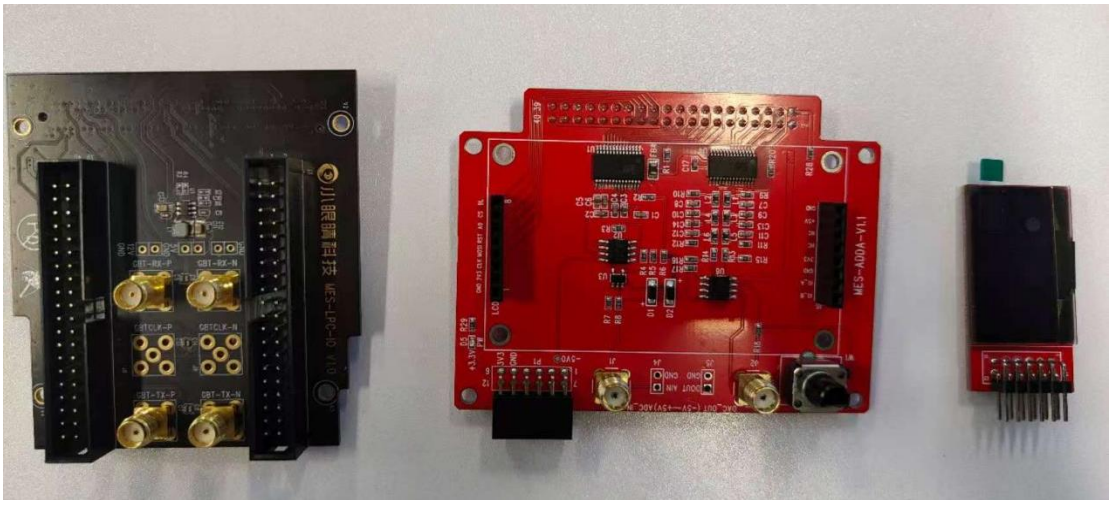
上图为添加了椒盐噪声的图像。

可以看到高斯滤波也让图像平滑、降噪，主要用在边缘检测、模糊处理时的预处理。其也是一种低通滤波，可以除去高频噪声。

18. adda_oled 波形显示实验例程

18.1. 实验简介

实验使用“小眼睛科技”公司的 FPGA 开发板以及 MES-LPC-IO FMC 扩展子卡，MES_ADDA 模块和 1.3 寸 oled 屏 PMOD 模块，扩展模块使用如下图所示。通过按键控制 DAC 输出正弦波、锯齿波、方波、三角波四种波形，并且环路到 ADC 采集，FPGA 将采集的 ADC 数据通过 OLED 屏显示对应波形。



18.2. 实验原理

（1）1.3 寸 OLED 屏使用原理参考《1.3 寸 oled 屏显示实验说明》；

（2）MES_ADDA 模块使用原理参考相关 DAC 信号输出及 ADC 采集模拟信号并显示波形实验；

（3）OLED 屏显示波形

DAC 信号输出时钟信号设置为 120MHz，且 DAC 输出波形的周期为 1024 个 DAC 时钟周期，ADC 采样时钟设置为 5MHz。OLED 屏分辨率为 128*64，若按一行 128 个像素点显示波形对应 128 个 ADC 采样数据。ADC 环路接收 DAC 输出波形，128 个采样点对应约 4 个周期的波形：

ADC 采样一个模拟波形周期所需 ADC 时钟计数为 1024/24：

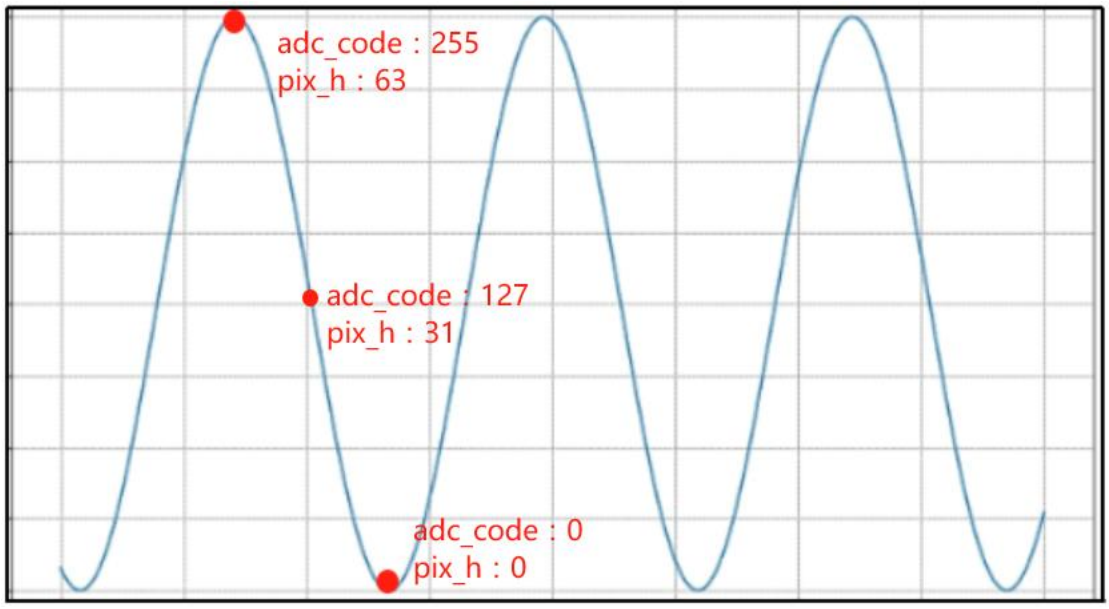
$$\frac{1024}{120\text{MHz}} = \frac{1024/24}{5\text{MHz}}$$

ADC 环路接收 DAC 输出波形，128 个采样点对应 3 个周期的模拟波形：

$$\frac{128}{1024/24} = 3$$

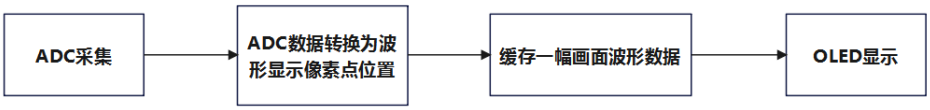
OLED 屏一列共 64 个像素点（0~63），且 MES_ADDA 模块采集的 ADC 数据为 8 bit 位宽对应数值 0~255，列像素点高度与最大数值 8’hFF 的关系为 1:4，因此将 ADC 采样数据右移 2bit 后的数值为该采样点在屏幕显示的高度。如下所示：

$$pix\_h = \frac{adc\_code + 1}{4} - 1$$



18.3. 代码思路

将 ADC 采集数据在 OLED 屏显示波形，需将 ADC 所采集数据转换为 OLED 屏上每个像素点的的像素值，“1”表示点亮该像素点，“0”则相反。缓存一幅画面的 128*64bit 数据后将数据读出并驱动 OLED 显示对应波形。



（1）数据的转换

根据以上推理的 ADC 与像素数据的转换公式，定义 64bit 的 wr_data 表示 1 列 64 个像素点的数据。一个 ADC 数据对应一列像素点，只点亮一列中的一个匹配的像素点。

```
//////////adc data

wire [5:0]bit_num;
wire [7:0]adc_code_1;
assign adc_code_1=adc_code+1'b1;
assign bit_num=adc_code_1[7:2]+1'b1;
reg [63:0]wr_data;
always @(posedge clk ) begin
    wr_data <= 64'd0;
    if (~rst_n)
        wr_data <= 64'd0;
    else
    begin
        wr_data[bit_num] <= 1'b1;
    end
end
```

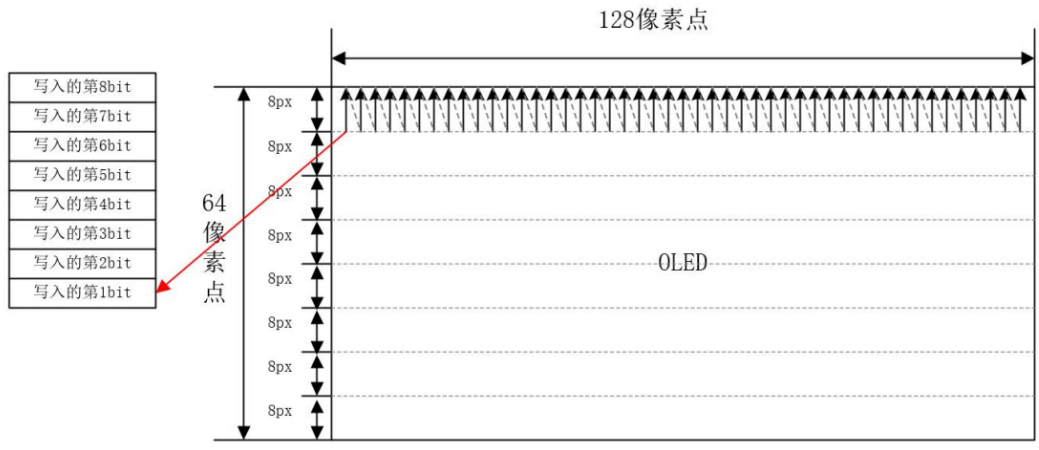
缓存数据与 OLED 的 128*64 像素数据对应关系如下:

	addr0	addr1	addr2	…	…	addr126	addr127
wr_data[63]							
wr_data[62]							
wr_data[61]							
wr_data[60]							
…							
…							
wr_data[3]							
wr_data[2]							
wr_data[1]							
wr_data[0]							

（2）波形数据输出

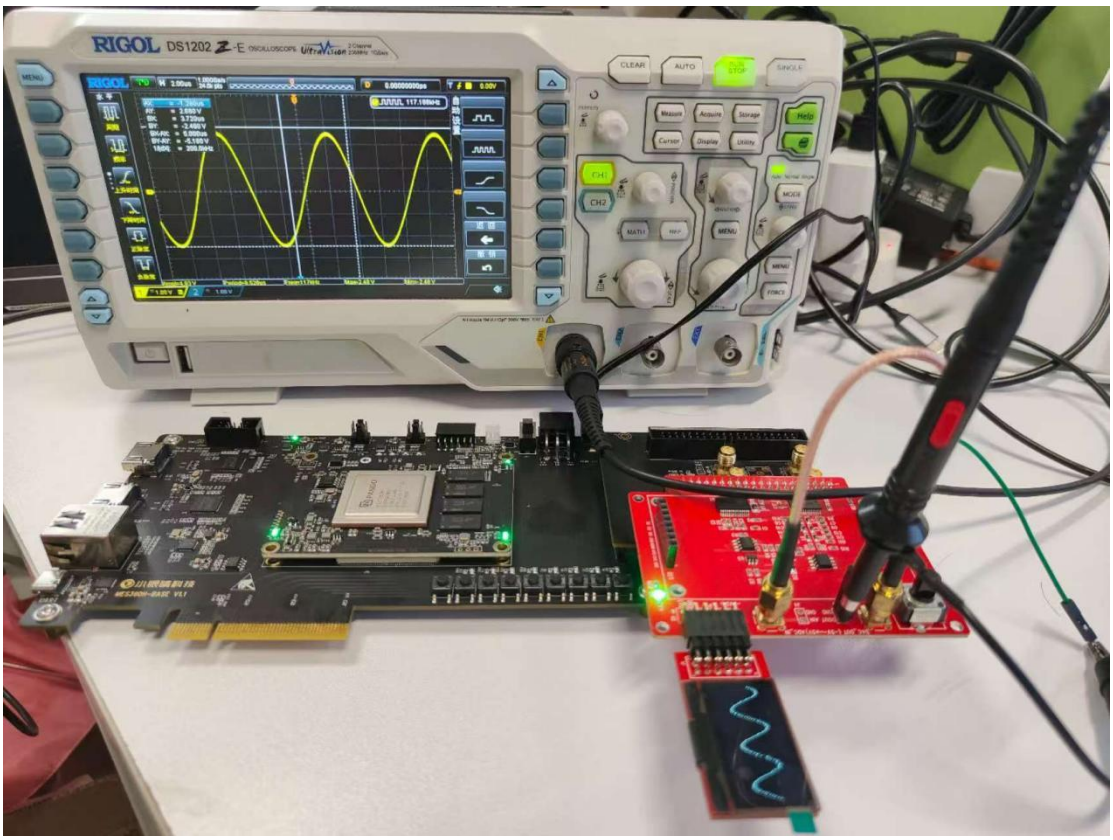
由于 OLED 的显示按每 8 行为 1 页，需要按页进行写操作，因此需 cnt_p 做页计数，根据当前的页数选取对应的 8bit 数据 data_sig。

```
////////////////////////////////////////////////////////////////////////////////
   function [7:0] data_out ;
      input [7:0] data_in ;
      integer i ;
      begin
         for (i = 0 ; i < 8 ; i = i + 1)
            data_out[i] = data_in[7 - i] ;
      end
   endfunction
always @(posedge clk ) begin
    if (~rst_n)
        data_sig <= 9'd0 ;
    else if (state == state_command)
        data_sig <= {1'b0 ,order[cnt] };
    else if (state == state_send)
    begin
        case(cnt_p)
            3'd0:    data_sig <= {1'b1 ,data_out(rd_data[63:56])};
            3'd1:    data_sig <= {1'b1 ,data_out(rd_data[55:48])};
            3'd2:    data_sig <= {1'b1 ,data_out(rd_data[47:40])};
            3'd3:    data_sig <= {1'b1 ,data_out(rd_data[39:32])};
            3'd4:    data_sig <= {1'b1 ,data_out(rd_data[31:24])};
            3'd5:    data_sig <= {1'b1 ,data_out(rd_data[23:16])};
            3'd6:    data_sig <= {1'b1 ,data_out(rd_data[15:8])};
            3'd7:    data_sig <= {1'b1 ,data_out(rd_data[7:0])};
        default:     data_sig <= 9'd0;
        endcase
    end
    else
        data_sig <= data_sig ;
end
```



18.4. 实验现象

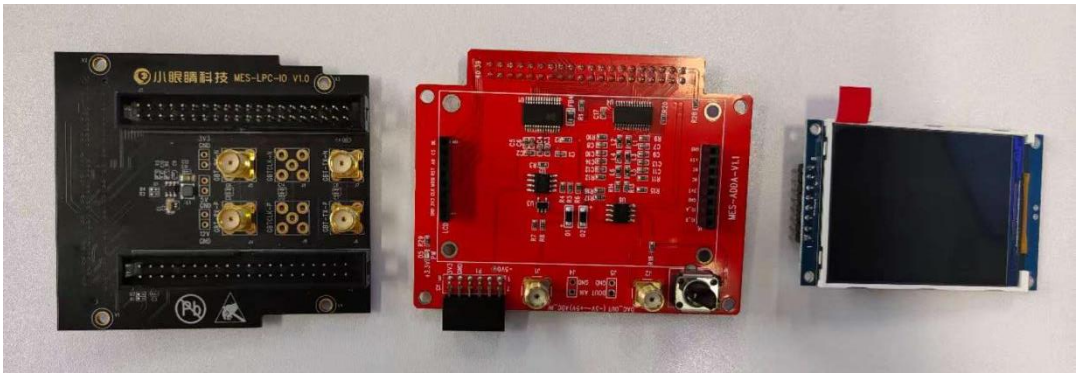
按下 KEY0 可切换 DAC 输出正弦波、锯齿波、方波和三角波四种波形，并且示波器可测量 DAC 输出对应的波形，波形环路到 ADC 采集，可在 1.3 寸 OLED 显示屏上观察到对应波形的显示，且显示画面每间隔 5 秒刷新一次。



19. SPI 屏幕驱动

19.1. 实验简介

实验使用“小眼睛科技”公司的 MES390H 开发板，配合 MES-LPC-IO FMC 扩展子卡，MES_ADDA 模块以及 SPI 屏幕，编写 SPI 屏幕驱动代码，实现屏幕上显示红绿蓝交错的九宫格图案。实验所需的扩展模块如下图所示。



19.2. 实验原理

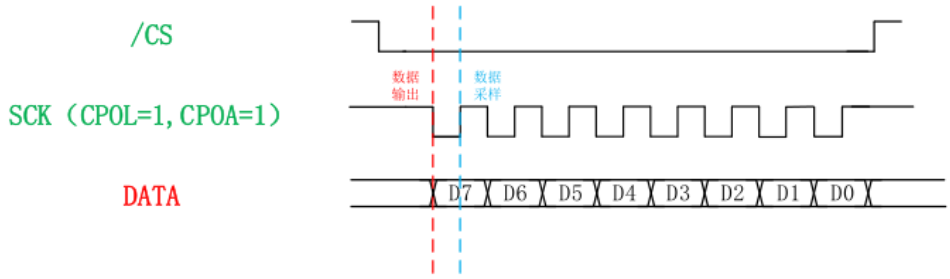
19.2.1. SPI 协议介绍

SPI 协议（Serial Peripheral Interface，串行外围接口）是一种同步串行通信协议，通常用于短距离设备间的高速数据传输。它采用主从架构，由主设备（如微控制器）和一个或多个从设备（如传感器、显示屏等）组成。SPI 通过 4 条主要的信号线进行数据传输：MOSI（主设备输出，从设备输入）、MISO（主设备输入，从设备输出）、SCLK（串行时钟，由主设备提供）和 CS（片选信号，控制与哪个从设备通信）。其中，MOSI 和 MISO 实现双向通信，而 SCLK 负责同步时钟信号。

SPI 的通信特点是全双工，即主设备和从设备可以同时发送和接收数据。通信过程由主设备控制，通过时钟信号 SCLK 同步数据传输。每次传输的数据按位发送，通常一字节一字节地进行，数据传输的顺序可以配置为从最高有效位（MSB）或最低有效位（LSB）开始。片选信号 CS 用于选择具体的从设备，低电平激活。当有多个从设备时，主设备可以通过多个 CS 引脚分别控制它们。SPI 协议广泛应用于短距离、高速通信的场景，如显示屏、存储设备、传感器、音频设备等。

SPI 有四种通讯模式，由时钟极性和时钟相位共同决定。时钟极性全称 ClockPolarity，通常简写成 CPOL。它是指时钟信号在空闲状态下是高电平还是低电平，当时钟空闲时为低电平即 CPOL==0，反之则 CPOL==1；时钟相位全称 ClockPhase，通常简写成 CPHA。它是指时钟信号开始有效的第一个边沿和数据的关系，当 CPHA==0 时，数据

采样发生在时钟信号有效的第一个边沿（也叫奇边沿），当 CPHA==1 时，数据采样发生在时钟信号有效的第二个边沿（也叫偶边沿）；我们以 CPOL=1，CPHA=1 的模式进行分析：



上图分析了时钟相位与时钟极性均设置为 1 时的 SPI 通讯时序。根据时序图可以发现，由于 CPOL 设置为 1，时钟 SCK 空闲状态为高电平，有效状态为低电平。由于 CPHA 设置为 1，数据采样将发生在时钟有效的第二个边沿，也就是图中时钟为低电平后的第二个边沿。根据 Sitronix 公司的 ST7789VM 芯片数据手册，我们本次实验所使用的 SPI 通讯模式即为此模式。

19.2.2. SPI 屏幕介绍

SPI 屏幕是一种通过串行外围接口（SPI, Serial Peripheral Interface）与微控制器或处理器通信的显示设备。其与微控制器的通讯通常包含四条主要的信号线：MOSI（主设备输出，从设备输入）、MISO（主设备输入，从设备输出）、SCLK（串行时钟）和 CS（片选/从设备选择）。有时也会简化为三线结构，不使用 MISO（本次实验使用的通讯方式就是三线结构）。

相比于并行通信方式（直接使用 rgb 格式输入像素数据），SPI 使用的引脚数量较少，通常只需要 4-5 根信号线就可以完成数据传输，因此硬件设计更为简洁，尤其适合引脚资源有限的嵌入式系统。

SPI 屏幕的控制流程一般是由主控制器通过 SPI 发送显示数据到屏幕控制器，屏幕控制器再将这些数据处理后驱动液晶或 OLED 屏幕显示图像。由于 SPI 是一种全双工通信协议，主设备和从设备可以在同一时钟周期内同时发送和接收数据，不过在大多数 SPI 屏幕应用中，通常只有主设备（控制器）负责发送显示数据，从设备（屏幕）接收数据。

虽然 SPI 接口相对于并行接口来说在引脚数和硬件复杂性上有优势，但其缺点是数据传输的带宽相对较小，因此在显示高分辨率图像时速度可能较慢，适合在较小分辨率的屏幕或更新频率要求不高的场景中使用。常见的 SPI 屏幕有 TFT 和 OLED 两种类型，广泛应用于智能手表、物联网设备、传感器终端等对显示要求不高的小型设备上。

本次实验选用深圳金逸晨电子有限公司生成的型号为 GMT024-8pinSPI_LCM 的 SPI 屏幕，GMT024-8pinSPI_LCM 使用 Sitronix 公司的 ST7789VM 芯片来控制 tft 液晶显

示屏幕的显示，其分辨率为 240*320，其引脚说明如下：

引脚序号	引脚名称	功能说明
1	GND	电源负极
2	VCC	电源正极（使用3.3v）
3	SCL	数据时钟线（对应SCLK）
4	SDA	数据线（对应MISO）
5	RST	复位
6	DC	数据，命令选择（高电平发送数据，低电平发送命令）
7	CS	片选信号（低电平选中，高电平不选中）
8	BL	tft背光控制信号（低电平关闭，高电平开启）

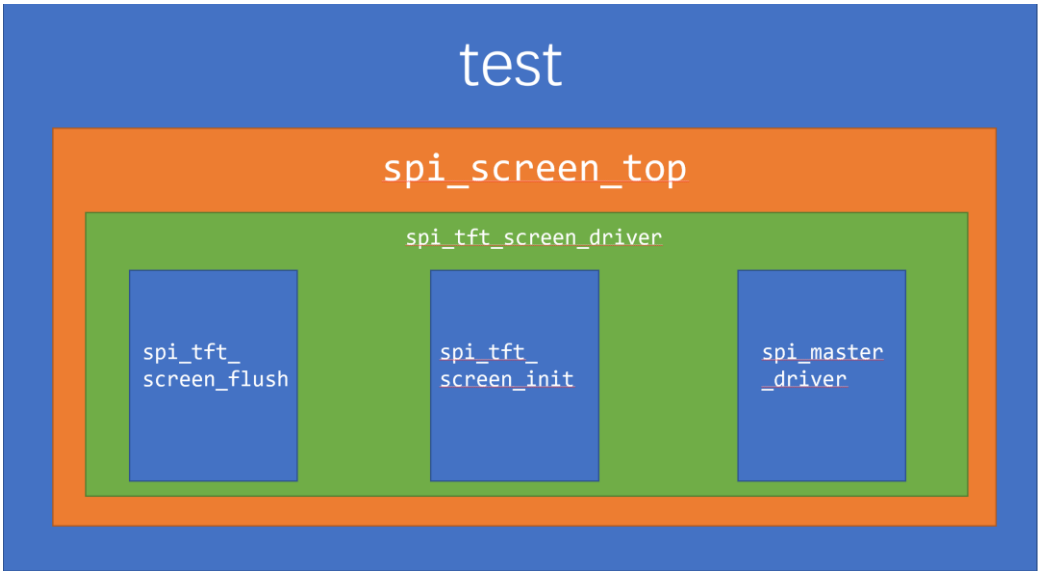
19.3. 代码介绍

19.3.1. 总体介绍

我们将 SPI 屏幕的驱动功能分为以下模块：

- test：产生测试数据，例化整个 spi 屏幕驱动模块，用于板级测试产生图像数据。
- spi_screen_top：对屏幕驱动模块进行封装，使之便于用户操作。
- spi_tft_screen_driver：spi 屏幕驱动模块，例化屏幕初始化模块，屏幕刷新显示模块，spi 发送模块。
- spi_tft_screen_flush：对用户输入的图像数据进行刷新显示。
- spi_tft_screen_init：spi 屏幕初始化模块，对 ST7789VM 进行相关配置。
- spi_master_driver：接收各个模块传入的命令与数据并通过 spi 协议发送给 ST 7789VM。

模块层次框图如下：



19.3.2. 模块介绍

test 模块：

```
1.  `timescale 1ns / 1ps
2.  module test(
3.      input                           sys_clk                 ,// 系统时钟
4.      input                           sys_rst_n               ,// 复位
5.
6.      //spi tft screen    屏幕接口
7.      output                          lcd_spi_sclk             ,// 屏幕 spi 时钟接口
8.      output                          lcd_spi_mosi             ,// 屏幕 spi 数据接口
9.      output                          lcd_spi_cs              ,// 屏幕 spi 使能接口
10.     output                          lcd_dc                  ,// 屏幕 数据/命令 接口
11.     output                          lcd_reset               ,// 屏幕复位接口
12.     output                          lcd_blk                  // 屏幕背光接口
13.
14. );
15.
16.     wire                            flush_data_update        ;//更新当前坐标点显示数据使能
17.     reg             [  15: 0]       flush_data              ;//当前坐标点显示的数据
18.     wire            [  15: 0]       flush_addr_width         ;//当前刷新的 x 坐标
19.     wire            [  15: 0]       flush_addr_height        ;//当前刷新的 y 坐标
20.
21.
22. always @(posedge sys_clk or negedge sys_rst_n) begin          // 显示九宫格测试图片 240*320
23.     if (sys_rst_n == 1'b0)
24.         flush_data <= 16'd0;
25.     else if ((flush_addr_width >= 'd0    && flush_addr_width < 'd106)
26.             && (flush_addr_height >= 'd0 && flush_addr_height < 'd80))
27.         flush_data <= 16'hF800;//红  1111 1000 0000 0000
28.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212)
29.             && (flush_addr_height >= 'd0 && flush_addr_height < 'd80))
30.         flush_data <= 16'h07E0;//绿  0000 0111 1110 0000
31.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320)
32.             && (flush_addr_height >= 'd0 && flush_addr_height < 'd80))
33.         flush_data <= 16'h001F;//蓝  0000 0000 0001 1111
34.
35.     else if ((flush_addr_width >= 'd0    && flush_addr_width < 'd106)
36.             && (flush_addr_height >= 'd80 && flush_addr_height < 'd160))
37.         flush_data <= 16'h07E0;//绿
38.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212)
39.             && (flush_addr_height >= 'd80 && flush_addr_height < 'd160))
40.         flush_data <= 16'h001F;//蓝
41.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320)
42.             && (flush_addr_height >= 'd80 && flush_addr_height < 'd160))
43.         flush_data <= 16'hF800;//红
44.
45.     else if ((flush_addr_width >= 'd0    && flush_addr_width < 'd106)
46.             && (flush_addr_height >= 'd160 && flush_addr_height < 'd240))
47.         flush_data <= 16'h001F;//蓝
48.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212)
49.             && (flush_addr_height >= 'd160 && flush_addr_height < 'd240))
50.         flush_data <= 16'hF800;//红
51.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320)
52.             && (flush_addr_height >= 'd160 && flush_addr_height < 'd240))
53.         flush_data <= 16'h07E0;//绿
```

```
54.       else
55.           flush_data <= 16'h0000;
56.   end
57.   //整个 spi 屏幕控制顶层，用户只需要提供显示数据就可以使用这个顶层进行 spi 屏幕显示
58.   spi_screen_top spi_screen_top_inst(
59.       .sys_clk                    (sys_clk                    ),
60.       .sys_rst_n                  (sys_rst_n                  ),
61.
62.
63.       //用户信号
64.       .flush_data_update_o        (flush_data_update          ),//更新当前坐标点显示数据使能
65.       .flush_data_i               (flush_data                 ),//当前坐标点显示的数据
66.       .flush_addr_width_o         (flush_addr_width           ),//当前刷新的 x 坐标
67.       .flush_addr_height_o        (flush_addr_height          ),//当前刷新的 y 坐标
68.
69.        //spi tft screen    屏幕接口
70.       .lcd_spi_sclk               (lcd_spi_sclk               ),//  屏幕 spi 时钟接口
71.       .lcd_spi_mosi               (lcd_spi_mosi               ),//  屏幕 spi 数据接口
72.       .lcd_spi_cs                 (lcd_spi_cs                 ),//  屏幕 spi 使能接口
73.       .lcd_dc                     (lcd_dc                     ),//  屏幕  数据/命令  接口
74.       .lcd_reset                  (lcd_reset                  ),//  屏幕复位接口
75.       .lcd_blk                    (lcd_blk                    ) //  屏幕背光接口
76.   );
77.
78.   endmodule
```

该模块是基于 SPI 接口驱动 TFT 液晶屏的顶层示例，主要功能是生成九宫格彩色测试图像，并通过 spi_screen_top 子模块显示在屏幕上。模块接口包含系统时钟 sys_clk、复位信号 sys_rst_n，以及屏幕控制所需的 SPI 信号（时钟、数据、片选、数据/命令切换、复位、背光）。模块中例化了 spi_screen_top 模块，根据 spi_screen_top 输出的 flush_addr_width 与 flush_addr_height 信号分配不同的显示数据，产生红绿蓝交错的九宫格测试图案。

在显示逻辑部分，定义了三个主要信号：flush_data_update 表示当前坐标点显示数据是否需要更新，若为高电平则代表一个 16bit 的 rgb565 图像数据通过 spi 协议成功发送给屏幕。flush_data 保存当前坐标点像素的 RGB565 数据值，flush_addr_width 与 flush_addr_height 分别表示当前刷新像素的横纵坐标。通过 always 块对坐标进行判断，将整个 240×320 的屏幕划分为九个矩形区域，并分别赋予红、绿、蓝三种颜色的组合，实现九宫格彩条测试图像。例如，当横坐标处于 0-80 时，像素值被赋为红色（16'hF800）；而在 106-160 的区域则显示蓝色（16'h001F），以此类推，最终形成九个彩色区域的测试画面。

在模块实例化部分，调用了 spi_screen_top 子模块。该子模块封装了 SPI 总线时序控制逻辑，用户只需提供 flush_data 和坐标信息即可完成图像显示。输入的像素数据与更新使能信号由 test 模块内部生成，再通过接口传递给 spi_screen_top，从而驱动屏幕显示。输出的 SPI 接口信号包括 lcd_spi_sclk（时钟）、lcd_spi_mosi（数据）、lcd_spi_cs（片选）、lcd_dc（数据/命令切换）、lcd_reset（硬件复位）和 lcd_blk（背光控制）。

总体代码比较简单，主要是功能是生成图像数据。值得注意的是 flush_addr_width 和 flush_addr_height 由 spi_screen_top 模块输出，他们的值会自动递增，我们只需要使用类似 vga 接口的操作方式在刷新的过程中给出图像数据即可。

spi_screen_top 模块：

```
1.  `timescale 1ns / 1ps
2.    //整个 spi 屏幕控制顶层，用户只需要提供显示数据就可以使用这个顶层进行 spi 屏幕显示
3.  module spi_screen_top(
4.      input                           sys_clk                 ,
5.      input                           sys_rst_n               ,
6.
7.
8.      //用户信号
9.      output                          flush_data_update_o      ,//更新当前坐标点显示数据使能
10.     input               [  15: 0]   flush_data_i            ,//当前坐标点显示的数据
11.     output              [  15: 0]   flush_addr_width_o       ,//当前刷新的 x 坐标
12.     output              [  15: 0]   flush_addr_height_o      ,//当前刷新的 y 坐标
13.
14.
15.       //spi tft screen    屏幕接口
16.     output                          lcd_spi_sclk            ,//  屏幕 spi 时钟接口
17.     output                          lcd_spi_mosi             ,//  屏幕 spi 数据接口
18.     output                          lcd_spi_cs              ,//  屏幕 spi 使能接口
19.     output                          lcd_dc                  ,//  屏幕  数据/命令  接口
20.     output                          lcd_reset               ,//  屏幕复位接口
21.     output                          lcd_blk                  //  屏幕背光接口
22.   );
23. //屏幕尺寸
24.     parameter                        SCREEN_WIDTH               = 32'd320;
25.     parameter                        SCREEN_HEIGHT               = 32'd240;
26.
27. //屏幕用户接口
28.     wire                [   7: 0]   spi_screen_flush_data      ;//屏幕显示数据
29.     wire                            spi_screen_flush_updte     ;//像素点数据刷新
30.     wire                            spi_screen_flush_fsync     ;//屏幕帧同步
31.
32. //长宽计数器
33.     reg                 [  15: 0]   width_cnt                  ;
34.     reg                 [  15: 0]   height_cnt                 ;
35.
36. //数据更新
37.     reg                             data_update_cnt             ;
38.
39.
40. //更新数据寄存器
41.     reg                 [  15: 0]   flush_data_reg              ;
42. //更新数据使能寄存器
43.     reg                             flush_updte_en;
44.
45.     assign   spi_screen_flush_data      = flush_data_reg[15:8];//高位发送
46.
47.     //当发送完 16bit 的图像数据时，坐标点显示数据使能拉高
48.     assign   flush_data_update_o = (spi_screen_flush_updte == 1'b1
49.               && data_update_cnt == 1'b0 && flush_updte_en == 1'b1) ? 1'b1 : 1'b0;
50.
51.     //将长宽计数器连接到输出
```

```
52.     assign  flush_addr_width_o          = width_cnt;
53.     assign  flush_addr_height_o         = height_cnt;
54. always@(posedge sys_clk or negedge sys_rst_n) begin
55.     if( sys_rst_n == 1'b0)
56.         flush_updte_en <= 'd0;
57.     else if( spi_screen_flush_fsync == 1'b1 )    //刷新模块发送完一帧数据,清零
58.         flush_updte_en <= 'd0;
59.     else if( spi_screen_flush_updte == 1'b1)    //发送完 8 位数据后,flush_updte_en 拉高
60.         flush_updte_en <= 'd1;
61.     else
62.         flush_updte_en <= flush_updte_en;
63. end
64. always@(posedge sys_clk or negedge sys_rst_n) begin
65.     if( sys_rst_n == 1'b0)
66.         data_update_cnt <= 'd0;
67.     else if( spi_screen_flush_fsync == 1'b1 )                       //刷新模块发送完一帧数据
68.         data_update_cnt <= 'd0;
69.      //刷新模块通过 spi 发送完一个 8bit 数据，data_update_cnt 加一
70.     else if( spi_screen_flush_updte == 1'b1)
71.          data_update_cnt <= data_update_cnt + 1'b1;
72.     else
73.         data_update_cnt <= data_update_cnt;
74. end
75.
76.
77. always@(posedge sys_clk or negedge sys_rst_n) begin
78.     if( sys_rst_n == 1'b0 )
79.         width_cnt <= 'd0;
80.     else if( spi_screen_flush_fsync == 1'b1 )             //一帧图像数据发送完，计数器清零
81.         width_cnt <= 'd0;
82.     else if( flush_data_update_o)//发送完当前像素点 16bit 图像数据时
83.         if( width_cnt == (SCREEN_WIDTH-1))                  //计数到计数器最大值时清零
84.             width_cnt <= 'd0;
85.         else
86.             width_cnt <= width_cnt + 1'b1;                          //width_cnt 计数器加 1
87.     else
88.         width_cnt <= width_cnt;
89. end
90.
91.
92. always@(posedge sys_clk or negedge sys_rst_n) begin
93.     if( sys_rst_n == 1'b0 )
94.         height_cnt <= 'd0;
95.     else if( spi_screen_flush_fsync == 1'b1)              //一帧图像数据发送完，计数器清零
96.         height_cnt <= 'd0;
97.     else if( width_cnt == (SCREEN_WIDTH-1) && flush_data_update_o)//当图像绘制完一行时
98.         if( height_cnt == (SCREEN_HEIGHT-1))        //计数到计数器最大值时清零
99.             height_cnt <= 'd0;
100.        else
101.            height_cnt <= height_cnt + 1'b1;                    //height_cnt 计数器加一
102.        else
103.            height_cnt <= height_cnt;
104.    end
105.
106.
107.
108.    always@(posedge sys_clk or negedge sys_rst_n) begin
109.        if( sys_rst_n == 1'b0)
```

```
110.            flush_data_reg <= 'd0;
111.        else if( spi_screen_flush_updte == 1'b1)                //刷新模块发送完一个图像数据
112.            if( data_update_cnt == 1'b0 )       //data_update_cnt=0 时,发送完了低八位,寄存新数据
113.                flush_data_reg <= flush_data_i;
114.            else
115.          //data_update_cnt=1 时，发送完了高八位,将数据向左移动 8 位，发送低八位数据
116.                flush_data_reg <= flush_data_reg << 8;
117.        else
118.            flush_data_reg <= flush_data_reg;
119.    end
120.
121.
122.
123.
124.    spi_tft_screen_driver spi_tft_screen_driver_inst(
125.        .sys_clk                        (sys_clk                    ),
126.        .sys_rst_n                      (sys_rst_n                  ),
127.
128.
129.        //用户接口
130.        .spi_screen_flush_data_i        (spi_screen_flush_data      ),//屏幕显示数据
131.        //像素点数据刷新//在进行数据到 tft 屏幕的显示
132.        .spi_screen_flush_updte_o       (spi_screen_flush_updte     ),
133.        .spi_screen_flush_fsync_o       (spi_screen_flush_fsync     ),//屏幕帧同步
134.
135.         //spi tft screen    屏幕接口
136.        .lcd_spi_sclk                   (lcd_spi_sclk               ),// 屏幕 spi 时钟接口
137.        .lcd_spi_mosi                   (lcd_spi_mosi               ),// 屏幕 spi 数据接口
138.        .lcd_spi_cs                     (lcd_spi_cs                 ),// 屏幕 spi 使能接口
139.        .lcd_dc                         (lcd_dc                     ),// 屏幕 数据/命令 接口
140.        .lcd_reset                      (lcd_reset                  ),// 屏幕复位接口
141.        .lcd_blk                        (lcd_blk                    ) // 屏幕背光接口
142.    );
143.    endmodule
```

spi_screen_top 模块是对 spi_tft_screen_driver 模块的封装，简化了 SPI 驱动模块的操作，使用户只需要按照类似 VGA 接口的方式，在行列坐标刷新过程中提供像素数据即可实现显示。该模块对外提供数据更新使能信号、像素坐标输出信号，以及完整的 SPI 屏幕接口，包括时钟、数据、片选、命令选择、复位和背光控制。

代码第 3 行定义了模块 spi_screen_top，这是 SPI 屏幕控制的顶层模块。用户只需提供要显示的像素数据即可使用。代码第 10-13 行定义了用户接口信号，包括 flush_data_update_o（数据刷新使能）、flush_data_i（当前显示数据）、flush_addr_width_o 和 flush_addr_height_o（刷新坐标输出）。代码第 16-21 行定义了 SPI 屏幕接口信号，用于与 TFT 屏幕通信，包括 SPI 时钟、数据、片选、命令选择、复位和背光。

代码第 23-25 行通过参数 SCREEN_WIDTH 和 SCREEN_HEIGHT 定义屏幕的分辨率，分别为 320×240，用于行列像素计数。代码第 27-30 行定义了内部信号 spi_screen_flush_data（8 位像素数据）、spi_screen_flush_updte（数据更新信号）和 spi_screen_flush_fsync（帧同步信号）。其中，spi_screen_flush_updte 每拉高一次表示 SPI 发送完 8 位图像数据，spi_screen_flush_fsync 每拉高一次表示 SPI 已完成一帧图像传输。

代码第 37 行定义了 data_update_cnt，用于区分用户输入的 16 位 RGB565 数据的高 8 位和低 8 位。在数据传输过程中，当 data_update_cnt 为 0 时发送高 8 位数据，为 1 时发送低 8 位数据，从而完成 16 位数据的 SPI 拆分传输。

代码第 75-87 行驱动 width_cnt 计数器，实现 X 方向像素扫描。当帧同步信号 spi_screen_flush_fsync 有效时，width_cnt 清零；当 flush_data_update_o 有效时，width_cnt 自增，直至到达 SCREEN_WIDTH-1，再清零。代码第 90-102 行驱动 height_cnt，Y 方向扫描逻辑类似，当一行刷新完成时自增，直至到达 SCREEN_HEIGHT-1 再清零，实现整帧逐行刷新。

代码第 106-116 行通过时钟驱动 flush_data_reg，根据 data_update_cnt 将输入的 16 位像素数据拆分为高低 8 位并逐字节输出。在高低位传输完成后，flush_data_update_o 信号拉高，通知用户可准备下一个像素点数据。与此同时，flush_addr_width_o 和 flush_addr_height_o 输出当前像素的行列坐标，方便用户逻辑对齐显示内容。

代码第 121-138 行实例化 spi_tft_screen_driver 子模块，将 spi_screen_top 内部生成的像素数据和控制信号转换为符合 TFT SPI 通信协议的时序信号，并输出到硬件接口，包括 lcd_spi_sclk、lcd_spi_mosi、lcd_spi_cs、lcd_dc、lcd_reset 和 lcd_blk。

综上，spi_screen_top 模块实现了像素数据缓存、行列计数、数据拆分传输、帧同步控制以及 SPI 输出的顶层封装，使用户无需关注 SPI 时序细节，仅需提供逐像素显示数据即可完成屏幕刷新操作。

spi_tft_screen_driver 模块：

```
1.  `timescale 1ns / 1ps
2.  module spi_tft_screen_driver(
3.      input                           sys_clk                 ,
4.      input                           sys_rst_n               ,
5.
6.
7.      //用户接口
8.      input           [   7: 0]       spi_screen_flush_data_i     ,//屏幕显示数据
9.      output                          spi_screen_flush_updte_o    ,//像素点数据刷新
10.     output                          spi_screen_flush_fsync_o    ,//屏幕帧同步
11.     //------
12.
13.      //spi tft screen    屏幕接口
14.     output                          lcd_spi_sclk                ,// 屏幕 spi 时钟接口
15.     output                          lcd_spi_mosi                ,// 屏幕 spi 数据接口
16.     output                          lcd_spi_cs                  ,// 屏幕 spi 使能接口
17.     output                          lcd_dc                      ,// 屏幕 数据/命令 接口
18.     output                          lcd_reset                   ,// 屏幕复位接口
19.     output                          lcd_blk                     // 屏幕背光接口
20. );
21.
22. //屏幕尺寸
23.     parameter                       SCREEN_WIDTH            = 32'd320;
24.     parameter                       SCREEN_HEIGHT           = 32'd240;
25.
```

```
26. //总模块信号
27.     reg                             lcd_init_done               ;//初始化完成标志信号
28.     wire                            spi_start                  ;//spi 开始信号
29.     wire                            spi_end                     ;//spi 结束信号
30.     wire                [   7: 0]    spi_send_data               ;//spi 发送数据
31.     wire                            spi_send_ack                ;//spi 数据发送完成响应
32.     wire                            lcd_dc_i                   ;//lcd 数据/命令信号
33.
34. //  初始化模块信号
35.     wire                            tft_screen_init_req        ;//初始化模块请求
36.     wire                            tft_screen_init_ack        ;//初始化模块完成
37.     wire                [   7: 0]    tft_screen_init_data       ;//初始化模块数据
38.     wire                            tft_screen_init_dc         ;//初始化模块 dc
39.     wire                            spi_send_init_req          ;//spi 发送数据请求
40.     wire                            spi_send_init_end          ;//结束 spi 发送
41.     wire                            spi_send_init_ack          ;//spi 数据发送完成响应
42.
43. //刷新模块
44.
45.     wire                            tft_screen_flush_req       ;//刷新模块请求
46.     wire                [   7: 0]    tft_screen_flush_data      ;//刷新模块数据
47.     wire                            tft_screen_flush_dc        ;//刷新模块数据/命令信号
48.
49.     wire                            spi_send_flush_req         ;//刷新模块发送请求
50.     wire                            spi_send_flush_end         ;//刷新模块发送结束
51.     wire                            spi_send_flush_ack         ;//刷新模块数据发送完成响应
52.
53.
54. //屏幕驱动信号，默认
55.     assign                          lcd_reset                  = 1'b1;
56.     assign                          lcd_blk                    = 1'b1;
57.
58.
59.     assign                  tft_screen_flush_req       = ( lcd_init_done == 1'b1) ? 1'b1 : 1'b0;
60.     assign                  spi_send_flush_ack         = ( lcd_init_done == 1'b1) ? spi_send_ack :
    1'b0;
61.
62.
63.     assign                  ft_screen_init_req         =  ~lcd_init_done;
64.     assign                  spi_send_init_ack          = ( lcd_init_done == 1'b0) ? spi_send_ack :
    1'b0;
65.
66.     //像素初始化完成之后进入显示数据刷新模式
67.   assign      spi_start         = ( lcd_init_done == 1'b0) ? spi_send_init_req : spi_send_flush_req;
68.   assign      spi_end           = ( lcd_init_done == 1'b0) ? spi_send_init_end : spi_send_flush_end;
69.   assign      spi_send_data     = ( lcd_init_done == 1'b0) ? tft_screen_init_data : tft_screen_flush_data;
70.   assign      lcd_dc_i          = ( lcd_init_done == 1'b0) ? tft_screen_init_dc : tft_screen_flush_dc;
71.
72.
73. //初始化是否完成
74. always@(posedge sys_clk or negedge sys_rst_n) begin
75.     if( sys_rst_n == 1'b0)
76.         lcd_init_done <= 1'b0;
77.     else if( tft_screen_init_ack == 1'b1)    //初始化模块完成  ,lcd_init_done 拉高
78.         lcd_init_done <= 1'b1;
79.     else
80.         lcd_init_done <= lcd_init_done;
81. end
```

```
82.
83.
84.
85.  //刷新模块
86.  spi_tft_screen_flush #(
87.       .SCREEN_WIDTH                    (SCREEN_WIDTH                ),
88.       .SCREEN_HEIGHT                   (SCREEN_HEIGHT               ),
89.       .Number_Of_Pixels                (SCREEN_WIDTH*SCREEN_HEIGHT*'d2)
90.  )spi_tft_screen_flush_inst(
91.       .sys_clk                    (sys_clk                  ),
92.       .sys_rst_n                  (sys_rst_n                ),
93.
94.        //用户接口
95.       .spi_screen_flush_data_i         (spi_screen_flush_data_i    ),//屏幕显示数据
96.       .spi_screen_flush_updte_o        (spi_screen_flush_updte_o   ),//像素点数据刷新
97.       .spi_screen_flush_fsync_o        (spi_screen_flush_fsync_o   ),//屏幕帧同步
98.
99.
100.          .tft_screen_flush_req_i          (tft_screen_flush_req       ),//刷新请求
101.          .tft_screen_flush_data_o         (tft_screen_flush_data      ),//刷新数据
102.          .tft_screen_flush_dc_o           (tft_screen_flush_dc        ),//刷新 dc
103.
104.
105.          .spi_send_flush_req_o            (spi_send_flush_req         ),//spi 发送数据请求
106.          .spi_send_flush_end_o            (spi_send_flush_end         ),//结束 spi 发送
107.          .spi_send_flush_ack_i            (spi_send_flush_ack         ) //spi 一个数据发送完成
108.     );
109.
110.
111.
112.     //初始化模块
113.     spi_tft_screen_init   #(
114.          .SCREEN_WIDTH                    (SCREEN_WIDTH                ),
115.          .SCREEN_HEIGHT                   (SCREEN_HEIGHT               )
116.     ) spi_tft_screen_init_inst(
117.          .sys_clk                    (sys_clk                  ),
118.          .sys_rst_n                  (sys_rst_n                ),
119.
120.
121.          .tft_screen_init_req_i           (tft_screen_init_req        ),//初始化请求
122.          .tft_screen_init_ack_o           (tft_screen_init_ack        ),//初始化完成
123.          .tft_screen_init_data_o          (tft_screen_init_data       ),//初始化数据
124.          .tft_screen_init_dc_o            (tft_screen_init_dc         ),//初始化 dc
125.
126.
127.          .spi_send_init_req_o             (spi_send_init_req          ),//spi 发送数据请求
128.          .spi_send_init_end_o             (spi_send_init_end          ),//结束 spi 发送
129.          .spi_send_init_ack_i             (spi_send_init_ack          ) //spi 一个数据发送完成
130.     );
131.
132.
133.
134.     //spi 主机驱动模块
135.     spi_master_driver spi_master_driver_inst(
136.          //系统接口
137.          .sys_clk                    (sys_clk                  ),
138.          .sys_rst_n                  (sys_rst_n                ),
139.
```

```
140.        //用户接口
141.        .spi_start_i                (spi_start              ),// spi 开始信号
142.        .spi_end_i                  (spi_end                ),// spi 结束信号
143.        .spi_send_data_i            (spi_send_data          ),// spi 发送数据
144.        .spi_send_ack_o             (spi_send_ack           ),// spi 发送 8bit 数据完成信号
145.
146.        .lcd_dc_i                   (lcd_dc_i               ),//数据还是命令信号输入
147.        .lcd_dc                     (lcd_dc                 ),//数据还是命令信号输出
148.
149.        //spi  端口
150.        .spi_sclk                   (lcd_spi_sclk           ),
151.        .spi_mosi                   (lcd_spi_mosi           ),
152.        .spi_cs                     (lcd_spi_cs             )
153.    );
154.
155.    endmodule
```

spi_tft_screen_driver 模块是 SPI 屏幕驱动的核心模块，此模块是屏幕数据刷新模块（spi_tft_screen_flush）、屏幕初始化模块（spi_tft_screen_init）、spi 发送模块（spi_master_driver）的顶层代码。它负责将像素数据和初始化命令通过 SPI 接口传输给 TFT 屏幕，实现屏幕的显示和刷新控制。模块内部将屏幕初始化和像素数据刷新分为两个子模块，并通过 SPI 主控驱动完成最终的数据传输。

模块第 2 行定义了 spi_tft_screen_driver，这是整个 SPI 屏幕驱动的顶层模块。第 3-4 行是系统时钟和复位输入。第 8-10 行是用户接口，包括屏幕显示数据输入 spi_screen_flush_data_i、像素点数据刷新输出 spi_screen_flush_updte_o 以及帧同步输出 spi_screen_flush_fsync_o。第 14-19 行定义了 SPI 屏幕硬件接口信号，包括 SPI 时钟、数据、片选、数据/命令选择、复位和背光控制。

模块第 23-24 行设置了屏幕参数 SCREEN_WIDTH = 320 和 SCREEN_HEIGHT = 240，用于像素扫描和刷新控制。第 27-32 行定义了总模块使用的内部信号，包括初始化完成标志 lcd_init_done、SPI 开始/结束信号 spi_start、spi_end，以及 SPI 数据和数据/命令选择信号 spi_send_data、lcd_dc_i 等。

第 35-41 行定义了屏幕初始化模块的信号，包括初始化请求 tft_screen_init_req、完成响应 tft_screen_init_ack、初始化数据 tft_screen_init_data 和对应的 DC 信号等，用于初始化阶段的数据传输和控制。第 45-51 行定义了刷新模块的信号，包括刷新请求 tft_screen_flush_req、刷新数据 tft_screen_flush_data、刷新 DC 信号 tft_screen_flush_dc 以及 SPI 数据发送请求/完成/响应信号。

第 55-56 行将 LCD 复位和背光信号默认置高。第 59-70 行根据初始化状态选择数据来源，如果初始化未完成，则 SPI 发送初始化数据；初始化完成后，进入像素数据刷新模式，SPI 发送刷新数据。这样保证屏幕在初始化完成前不会刷新显示数据。

第 74-81 行使用 lcd_init_done 寄存器记录初始化完成状态，当初始化模块完成时置高，标志模块进入刷新模式。

第 86-108 行实例化 spi_tft_screen_flush 模块，负责将用户输入的 8 位像素数据

逐行逐列刷新到屏幕，并输出刷新控制信号。该模块根据屏幕宽高和像素数量自动管理行列扫描，并通过 spi_send_flush_req/end/ack 与 SPI 主控模块交互完成数据传输。

第 113-130 行实例化 spi_tft_screen_init 模块，实现屏幕初始化，包括发送初始化命令和设置显示参数。通过 tft_screen_init_req/ack/data/dc 与 SPI 主控模块交互完成初始化数据传输。

第 135-153 行实例化 spi_master_driver 模块，这是 SPI 主控驱动模块，接收来自初始化模块或刷新模块的 SPI 开始/结束信号和待发送数据，将数据按照 SPI 协议输出到屏幕接口，包括 SPI 时钟、数据和片选信号，同时传递数据/命令选择信号 lcd_dc。

整体流程为：系统复位后，模块先通过初始化子模块完成 TFT 屏幕初始化，初始化完成后进入刷新模式，通过刷新子模块获取用户提供的像素数据，最终通过 SPI 主控模块发送到屏幕，实现完整的显示和刷新功能。用户只需要提供逐像素数据，无需关心 SPI 时序和屏幕初始化流程，即可驱动 TFT 屏幕显示图像。

spi_tft_screen_flush 模块

```
1.  `timescale 1ns / 1ps
2.
3.  //对 spi tft 屏幕进行刷新
4.  module spi_tft_screen_flush(
5.      input                           sys_clk                 ,
6.      input                           sys_rst_n               ,
7.
8.
9.      //用户接口
10.     input           [   7: 0]       spi_screen_flush_data_i     ,//屏幕显示数据
11.     output                          spi_screen_flush_updte_o    ,//像素点数据刷新
12.     output                          spi_screen_flush_fsync_o    ,//屏幕帧同步
13.
14.
15.     //驱动模块
16.     input                   tft_screen_flush_req_i     , //刷新请求//初始化后此信号拉高
17.     output reg  [ 7: 0]     tft_screen_flush_data_o    , //刷新的图像数据和刷新命令通过 spi 模块发送
18.     output reg              tft_screen_flush_dc_o      , //刷新 dc//区分命令与数据
19.
20.     //SPI 主模块
21.     output                  spi_send_flush_req_o        ,//spi 发送数据请求
22.     output                  spi_send_flush_end_o        ,   //结束 spi 发送
23.     input                   spi_send_flush_ack_i            //spi 一个数据发送完成
24. );
25.     parameter               SCREEN_WIDTH        = 16'd320;
26.     parameter               SCREEN_HEIGHT       = 16'd240;
27.     parameter               Number_Of_Pixels    = 32'd240*32'd320*32'd2; //  像素点个数
28.
29.
30.     localparam              S_IDLE                  = 4'b0001;
31.     localparam              S_DATA                  = 4'b0010;  //发送数据
32.     localparam              S_DELAY                 = 4'b0100;  //延时
33.     localparam              S_FRAME_SYNC            = 4'b1000;  //  帧同步
34.
```

```
35.     //命令与数据之间切换等待 5 个时钟周期
36.     localparam                          DELAY_5clk              = 'd5    ;
37.
38.     reg             [  31: 0]       flush_cnt       ;       //刷新模块写命令/数据计数器
39.     reg             [  12: 0]       delay_cnt       ;       //延迟计数
40.     reg             [   3: 0]       state           ,next_state;        //状态寄存器
41.
42.     //为写数据状态时，请求拉高
43.     assign    spi_send_flush_req_o     = (state == S_DATA) ? 1'b1 : 1'b0;
44.     //为延迟状态或帧同步状态时结束信号拉高
45.     assign     spi_send_flush_end_o     = (state == S_DELAY || state == S_FRAME_SYNC) ? 1'b1 : 1'b0;
46.
47.     //当发送完数据时，且写命令/数据计数器 计数到 10 时，像素点数据刷新拉高
48.     assign     spi_screen_flush_updte_o   = ( spi_send_flush_ack_i == 1'b1 && flush_cnt >= 'd10 ) ? 1'b1 : 1'b0;
49.     //为帧同步状态时，屏幕帧同步拉高，产生帧同步信号
50.     assign      spi_screen_flush_fsync_o   = ( state == S_FRAME_SYNC ) ? 1'b1 : 1'b0;
51.
52. always@(posedge sys_clk or negedge sys_rst_n) begin
53.     if( sys_rst_n == 1'b0 )
54.         state <= S_IDLE;
55.     else
56.         state <= next_state;                //将状态赋值为下一状态
57. end
58.
59.
60. always@(*) begin
61.     case(state)
62.     S_IDLE:
63.         if( tft_screen_flush_req_i == 1'b1 )       //初始化完成之后此信号拉高模块进入刷新模式
64.             next_state = S_DATA;
65.         else
66.             next_state = S_IDLE;
67.     S_DATA:
68.         //地址设置需要发送命令和写入四个参数，设置 XY 地址共需要发送 10 个数据
69.         if( spi_send_flush_ack_i == 1'b1 && flush_cnt <= 'd10 )
70.             //加上发送内存写入命令共需要 11 个数据，所以 flush_cnt 计数到 10
71.             next_state = S_DELAY;
72.
73.         //一帧图像数据发送完成，跳转到帧同步状态
74.         else if( spi_send_flush_ack_i == 1'b1 && flush_cnt == (Number_Of_Pixels + 'd10))
75.             next_state = S_FRAME_SYNC;
76.         else
77.             next_state = S_DATA;
78.     S_DELAY:
79.         if( delay_cnt == DELAY_5clk)  //延迟 5 给时钟周期后，跳转到 S_DATA 状态
80.             next_state = S_DATA;
81.         else
82.             next_state = S_DELAY;
83.     S_FRAME_SYNC:
84.         next_state = S_IDLE;
85.     default: next_state = S_IDLE;
86.     endcase
87. end
88.
89.
90. //发送数据计数
91. always@(posedge sys_clk or negedge sys_rst_n) begin
92.     if( sys_rst_n == 1'b0 )
```

```
93.             flush_cnt <= 'd0;
94.                 //一帧图像数据发送完成，flush_cnt 清零
95.         else if( spi_send_flush_ack_i == 1'b1 &&   flush_cnt == (Number_Of_Pixels + 'd10))
96.             flush_cnt <= 'd0;
97.         else if( spi_send_flush_ack_i == 1'b1 )//发送完一个数据，flush_cnt 加 1
98.             flush_cnt <= flush_cnt + 1'b1;
99.         else
100.            flush_cnt <= flush_cnt;
101.    end
102.
103.
104.    //延时计数
105.        always@(posedge sys_clk or negedge sys_rst_n) begin
106.            if( sys_rst_n == 1'b0)
107.                delay_cnt <= 'd0;
108.            else if( state == S_DELAY)//为延迟状态时，delay_cnt 加 1
109.                delay_cnt <= delay_cnt + 1'b1;
110.            else
111.                delay_cnt <= 'd0;
112.        end
113.
114.        //tft_screen_flush_dc_o=0 时写命令
115.        //tft_screen_flush_dc_o=1 时写数据
116.        always @(*) begin
117.            case(flush_cnt)
118.            'd0: begin
119.                tft_screen_flush_data_o = 8'h2A;                        //设置列地址
120.                tft_screen_flush_dc_o    = 1'b0;
121.            end
122.            //写 X
123.            'd1: begin
124.                tft_screen_flush_data_o = 8'h00;                        //列地址开始的高 8 位
125.                tft_screen_flush_dc_o    = 1'b1;
126.            end
127.            'd2: begin
128.                tft_screen_flush_data_o = 8'h00;                        //列地址开始的低 8 位
129.                tft_screen_flush_dc_o    = 1'b1;
130.            end
131.            'd3: begin
132.                tft_screen_flush_data_o = SCREEN_WIDTH[15:8];           //列地址结束的高 8 位
133.                tft_screen_flush_dc_o    = 1'b1;
134.            end
135.            'd4: begin
136.                tft_screen_flush_data_o = SCREEN_WIDTH[7:0] - 1'b1;     //列地址结束的低 8 位
137.                tft_screen_flush_dc_o    = 1'b1;
138.            end
139.            //写 Y
140.            'd5: begin
141.                tft_screen_flush_data_o = 8'h2B;                        //设置行地址
142.                tft_screen_flush_dc_o    = 1'b0;
143.            end
144.            'd6: begin
145.                tft_screen_flush_data_o = 8'h00;                        //行地址开始的高 8 位
146.                tft_screen_flush_dc_o    = 1'b1;
147.            end
148.            'd7: begin
149.                tft_screen_flush_data_o = 8'h00;                        //行地址开始的低 8 位
150.                tft_screen_flush_dc_o    = 1'b1;
```

```
151.          end
152.          'd8: begin
153.              tft_screen_flush_data_o = SCREEN_HEIGHT[15:8];          //行地址结束的高 8 位
154.              tft_screen_flush_dc_o     = 1'b1;
155.          end
156.          'd9: begin
157.              tft_screen_flush_data_o = SCREEN_HEIGHT[7:0] - 1'b1;    //行地址结束的低 8 位
158.              tft_screen_flush_dc_o     = 1'b1;
159.          end
160.          //写数据
161.          'd10: begin
162.              tft_screen_flush_data_o = 8'h2C;                        //发送图像数据
163.              tft_screen_flush_dc_o     = 1'b0;
164.          end
165.          default: begin
166.              tft_screen_flush_data_o = spi_screen_flush_data_i;      //图像显示数据
167.              tft_screen_flush_dc_o     = 1'b1;
168.          end
169.          endcase
170.      end
171.      endmodule
```

spi_tft_screen_flush 模块的作用是将屏幕刷新数据按照 SPI 协议逐帧传输到 TFT 屏幕。模块接收经过 spi_screen_top 模块转换的 8bit 数据（用户给的是 16bit 的 rgb565 格式的），将其不断发送至屏幕进行显示。实现了刷新流程的有限状态机控制，包括地址设置、延时等待、像素数据写入和帧同步，确保一帧数据能够正确显示在屏幕上。

模块第 4 行定义了 spi_tft_screen_flush，输入包括系统时钟 sys_clk 和异步复位 sys_rst_n。第 10-12 行是用户接口，其中 spi_screen_flush_data_i 输入逐像素数据，spi_screen_flush_updte_o 输出像素点刷新信号，spi_screen_flush_fsync_o 输出帧同步信号。第 16-23 行是与 SPI 驱动相关的接口，模块通过 spi_send_flush_req_o 请求 SPI 发送数据，spi_send_flush_end_o 表示传输结束，spi_send_flush_ack_i 表示 SPI 一个字节传输完成。

第 25-27 行定义屏幕参数：屏幕宽 320、屏幕高 240，总像素数量为 320×240×2（每像素 16 位）。第 30-33 行定义状态机的四个状态：空闲（S_IDLE）、数据发送（S_DATA）、延时（S_DELAY）、帧同步（S_FRAME_SYNC）。第 36 行定义命令与数据切换的延时常数 DELAY_5clk。

第 38-40 行定义了刷新计数器 flush_cnt、延迟计数器 delay_cnt 和状态机寄存器 state。第 43-45 行将状态机状态与 SPI 控制信号对应，当进入 S_DATA 时产生发送请求，当处于延时或帧同步状态时拉高结束信号。第 48-50 行定义了用户接口信号：当发送的数据超过 10 个时 spi_screen_flush_updte_o 拉高，表示像素点刷新完成；在帧同步状态下 spi_screen_flush_fsync_o 拉高，产生一帧的同步信号。

第 52-57 行实现状态机的时序逻辑，每个时钟沿更新当前状态。第 60-86 行是状态转移逻辑：

S_IDLE（空闲状态）： 初始状态，等待外部的刷新请求信号 `tft_screen_flush_`

req_i 拉高。当请求有效时，进入 **S_DATA** 状态，开始刷新流程。

S_DATA（数据发送状态）：在该状态下，模块依次发送 **列地址设置命令、行地址设置命令** 和 **写入数据命令**，然后逐像素写入屏幕显示数据。flush_cnt 用于计数当前发送的是哪一个命令或数据：计数 0~10 时，依次完成列、行地址以及写入命令发送。大于 10 时，开始连续输出像素点数据。当 flush_cnt 达到一帧数据长度（Number_Of_Pixels + 10）时，说明一帧像素写入完成，进入 **S_FRAME_SYNC** 状态。在命令与数据之间切换时，会进入 **S_DELAY** 状态，保持几个时钟周期的空隙。

S_DELAY（延时状态）：用于命令和数据之间的过渡。延时 5 个时钟周期（由 delay_cnt 控制），确保屏幕控制器能够稳定识别命令与数据的切换。延时完成后重新进入 **S_DATA** 状态继续数据传输。

S_FRAME_SYNC（帧同步状态）：当一帧数据传输完成后，进入此状态。输出帧同步信号 spi_screen_flush_fsync_o 拉高，通知上层逻辑新的一帧已经完成。随后状态机回到 **S_IDLE**，准备下一次刷新。

整体来说，状态机在空闲状态下等待刷新请求进入数据发送状态；在数据发送状态下先发送地址和写入命令（共 11 个字节），再持续写入像素数据；一帧数据完成后进入帧同步状态；延时状态则用于命令与数据的切换，等待 5 个时钟周期后回到数据发送状态。帧同步状态完成后回到空闲状态，准备下一帧刷新。

第 91-101 行实现数据发送计数器 flush_cnt，在 SPI 确认发送完成后自增，用于区分是发送初始化命令还是像素数据。当计数到达一帧数据总数时清零，重新开始下一帧刷新。第 105-112 行实现延时计数器 delay_cnt，在延时状态下累加计数，用于保证命令和数据之间的时序要求。

第 116-170 行根据计数器值决定输出的数据内容和 DC 信号。当 flush_cnt=0 时，发送设置列地址命令 0x2A（DC=0，表示命令）；随后依次写入列地址的起始高 8 位、起始低 8 位、结束高 8 位、结束低 8 位。接着 flush_cnt=5 时发送设置行地址命令 0x2B，再依次写入行地址的起始高 8 位、起始低 8 位、结束高 8 位和结束低 8 位。到 flush_cnt=10 时发送写入内存命令 0x2C（表示开始写入像素数据）。之后的所有计数（大于 10）都取用户提供的像素数据 spi_screen_flush_data_i，并将 DC 置 1 表示数据。

spi_tft_screen_flush 模块通过状态机依次完成列地址和行地址设置，再发送写入命令，最后连续传输一帧的图像像素数据。模块在每一帧完成时输出帧同步信号，屏幕数据刷新模块在外部信号 tft_screen_flush_req_i（刷新请求）的触发下开始工作，用一个状态机产生需要写入芯片的数据，并向 spi 发送模块发送请求信号，将命令与数据不断的产生并发送给芯片。在芯片初始化完成之后，这个模块是主要的驱动逻辑。

spi_tft_screen_init 模块

```
1.  timescale 1ns / 1ps
```

```
2.
3.  //对 spi tft 屏幕进行初始化
4.  module spi_tft_screen_init(
5.      input                           sys_clk                 ,
6.      input                           sys_rst_n               ,
7.
8.      input                           tft_screen_init_req_i   ,//初始化请求
9.      output                          tft_screen_init_ack_o   ,//初始化完成
10.     output reg          [   7: 0]   tft_screen_init_data_o  ,//初始化数据
11.     output reg                      tft_screen_init_dc_o    ,//初始化 dc
12.
13.     output                          spi_send_init_req_o     ,//spi 发送数据请求
14.     output                          spi_send_init_end_o     ,//结束 spi 发送
15.     input                           spi_send_init_ack_i      //spi 一个数据发送完成
16. );
17.
18.     parameter                       SCREEN_WIDTH            = 16'd320;
19.     parameter                       SCREEN_HEIGHT           = 16'd240;
20.
21.
22.     localparam      DELAY_255ms   = 32'd12_750_000;//255ms 255_000_000 /20 =12_750_000
23.     localparam      DELAY_200us             = 32'd10_000;     //200us 200_000/20=10_000
24.     localparam      S_IDLE                  = 4'b0001;//初始状态
25.     localparam      S_SEND_DATA             = 4'b0010;//发送数据状态
26.     localparam      S_DELAY                 = 4'b0100;//延迟状态
27.     localparam      S_ACK                   = 4'b1000;//响应状态
28.
29.
30.     reg             [   4: 0]   init_cnt                ;//初始化命令/数据计数
31.     reg             [  31: 0]   delay_cnt               ;//延时计数
32.     reg             [   3: 0]   state                   ;//状态寄存器
33.     reg             [   3: 0]   next_state              ;//下一状态寄存器
34.
35.
36.     //为响应状态时，初始化完成信号拉高
37.     assign      tft_screen_init_ack_o       = (state == S_ACK) ? 1'b1 : 1'b0;
38.     //为发送数据状态时，spi 发送数据请求信号拉高
39.     assign      spi_send_init_req_o         = (state == S_SEND_DATA) ? 1'b1 : 1'b0;
40.     //为延迟状态时，结束 spi 发送
41.     assign      spi_send_init_end_o         = (state == S_DELAY) ? 1'b1 : 1'b0;
42.
43. always@(posedge sys_clk or negedge sys_rst_n) begin
44.     if( sys_rst_n == 1'b0)
45.         state <= S_IDLE;
46.     else
47.         state <= next_state;
48. end
49.
50.
51. always@(*) begin
52.     case(state)
53.     S_IDLE:
54.         if( tft_screen_init_req_i == 1'b1)//初始化请求有效时，跳转到发送数据状态
55.             next_state <= S_SEND_DATA;
56.         else
57.             next_state <= S_IDLE;
58.     S_SEND_DATA:
59.         if( spi_send_init_ack_i == 1'b1)//spi 一个数据发送完成，跳转到延迟状态
```

```
60.                 next_state <= S_DELAY;
61.             else
62.                 next_state <= S_SEND_DATA;
63.         S_DELAY:
64.             if( init_cnt == 'd18)//初始化命令和数据发送完成跳转到响应状态
65.                 if( delay_cnt == DELAY_255ms)
66.                     next_state <= S_ACK;
67.                 else
68.                     next_state <= S_DELAY;
69.             else if(init_cnt == 'd1 )//延迟结束后，跳转到发送数据状态
70.                 if(delay_cnt == DELAY_255ms)
71.                     next_state <= S_SEND_DATA;
72.                 else
73.                     next_state <= S_DELAY;
74.             else if(init_cnt == 'd2 )
75.                     if(delay_cnt == DELAY_255ms)
76.                     next_state <= S_SEND_DATA;
77.                 else
78.                     next_state <= S_DELAY;
79.             else if(init_cnt == 'd4 )
80.                     if(delay_cnt == DELAY_255ms)
81.                     next_state <= S_SEND_DATA;
82.                 else
83.                     next_state <= S_DELAY;
84.             else if(init_cnt == 'd17 )
85.                     if(delay_cnt == DELAY_255ms)
86.                     next_state <= S_SEND_DATA;
87.                 else
88.                     next_state <= S_DELAY;
89.             else if( delay_cnt == DELAY_200us)
90.                 next_state <= S_SEND_DATA;
91.             else
92.                 next_state <= S_DELAY;
93.         S_ACK:
94.             next_state <= S_IDLE;
95.         default: next_state <= S_IDLE;
96.         endcase
97.
98.   end
99.
100.
101.
102.
103.    //初始化数据计数//
104.    always@(posedge sys_clk or negedge sys_rst_n) begin
105.        if( sys_rst_n == 1'b0)
106.            init_cnt <= 'd0;
107.        else if( spi_send_init_ack_i == 1'b1)//spi 一个数据发送完成，init_cnt 加 1
108.            init_cnt <= init_cnt + 1'b1;
109.        else
110.            init_cnt <= init_cnt;
111.    end
112.
113.
114.    //延时计数//写命令之间需要间隔的时间
115.    always@(posedge sys_clk or negedge sys_rst_n) begin
116.        if( sys_rst_n == 1'b0)
117.            delay_cnt <= 'd0;
```

```
118.        else if( state == S_DELAY)              //为延迟状态时，delay_cnt 加 1
119.            delay_cnt <= delay_cnt + 1'b1;
120.        else
121.            delay_cnt <= 'd0;
122.    end
123.
124.
125.    //命令数据输出
126.    //0 命令，1 数据
127.    always@(*)begin
128.        case (init_cnt)
129.            'd0: begin
130.                tft_screen_init_data_o  = 8'h01;        //SWRESET//软复位
131.                tft_screen_init_dc_o    = 1'b0;
132.            end
133.            'd1: begin
134.                tft_screen_init_data_o  = 8'h11;        //SLPOUT//唤醒
135.                tft_screen_init_dc_o    = 1'b0;
136.            end
137.            'd2: begin
138.                tft_screen_init_data_o  = 8'h3A;        //COLMOD//像素设置
139.                tft_screen_init_dc_o    = 1'b0;
140.            end
141.            'd3: begin
142.                tft_screen_init_data_o  = 8'h55;        //数据//0_101_0_101//0_65k 像素_0_rgb565(16bit)
143.                tft_screen_init_dc_o    = 1'b1;
144.            end
145.            'd4: begin
146.                tft_screen_init_data_o  = 8'h36;        //MADCTL//帧内存数据的读写扫描方向
147.                tft_screen_init_dc_o    = 1'b0;
148.            end
149.            'd5: begin
150.                tft_screen_init_data_o  = 8'h70;        //数据//01110000
151.                tft_screen_init_dc_o    = 1'b1;
152.            end
153.            'd6: begin
154.                tft_screen_init_data_o  = 8'h2A;        //CASET 列地址设置
155.                tft_screen_init_dc_o    = 1'b0;
156.            end
157.
158.            'd7: begin
159.                tft_screen_init_data_o  = 8'h00;        //列地址开始的高 8 位
160.                tft_screen_init_dc_o    = 1'b1;
161.            end
162.            'd8: begin
163.                tft_screen_init_data_o  = 8'h00;        //列地址开始的低 8 位
164.                tft_screen_init_dc_o    = 1'b1;
165.            end
166.            'd9: begin
167.                tft_screen_init_data_o  = SCREEN_WIDTH[15:8];           //列地址结束的高 8 位
168.                tft_screen_init_dc_o    = 1'b1;
169.            end
170.            'd10: begin
171.                tft_screen_init_data_o  = SCREEN_WIDTH[7:0] - 1'b1;     //列地址结束的低 8 位
172.                tft_screen_init_dc_o    = 1'b1;
173.            end
174.
175.            'd11: begin
```

```
176.                tft_screen_init_data_o  = 8'h2B;                    //RASET//行地址设置
177.                tft_screen_init_dc_o    = 1'b0;
178.            end
179.
180.            'd12: begin
181.                tft_screen_init_data_o  = 8'h00;                    //与列地址设置相似
182.                tft_screen_init_dc_o    = 1'b1;
183.            end
184.            'd13: begin
185.                tft_screen_init_data_o  = 8'h00;                    //数据
186.                tft_screen_init_dc_o    = 1'b1;
187.            end
188.            'd14: begin
189.                tft_screen_init_data_o  = SCREEN_HEIGHT[15:8];          //数据
190.                tft_screen_init_dc_o    = 1'b1;
191.            end
192.            'd15: begin
193.                tft_screen_init_data_o  = SCREEN_HEIGHT[7:0] - 1'b1;    //数据
194.                tft_screen_init_dc_o    = 1'b1;
195.            end
196.            'd16: begin
197.                tft_screen_init_data_o  = 8'h13;                //NORON//转换为正常显示模式
198.                tft_screen_init_dc_o    = 1'b0;
199.            end
200.            'd17: begin
201.                tft_screen_init_data_o  = 8'h29;                //DISPON//开启显示
202.                tft_screen_init_dc_o    = 1'b0;
203.            end
204.            default: begin
205.                tft_screen_init_data_o  = 8'h01;                //SWRESET//软复位
206.                tft_screen_init_dc_o    = 1'b0;
207.            end
208.        endcase
209.    end
210.
211.    endmodule
212.
```

spi_tft_screen_init 模块用于完成 SPI 接口 TFT 屏幕的初始化操作。该模块在接收到初始化请求后，会按照预设的命令序列依次向屏幕发送控制指令和参数数据，同时在必要的指令之间插入延时，以满足屏幕的上电与工作时序要求。最终，当所有初始化命令执行完毕后，模块会拉高初始化完成信号，表示屏幕已经进入可显示状态。

代码第 4 行定义了模块 spi_tft_screen_init，这是专门用于屏幕初始化的控制模块。输入信号 sys_clk 和 sys_rst_n 分别提供系统时钟和复位；tft_screen_init_req_i 用于触发初始化过程，tft_screen_init_ack_o 在初始化完成时输出有效信号。输出端口 tft_screen_init_data_o 与 tft_screen_init_dc_o 用于提供初始化命令或数据以及命令/数据选择控制。spi_send_init_req_o、spi_send_init_end_o 和 spi_send_init_ack_i 构成与 SPI 发送模块的交互接口，分别表示数据发送请求、结束信号和单次数据发送完成反馈。

代码第 18-19 行定义了参数 SCREEN_WIDTH 和 SCREEN_HEIGHT，分别指定屏幕的分辨率为 320×240，为后续地址设置指令提供参考。第 22-23 行定义了延时常量

DELAY_255ms 与 DELAY_200us，用于在特定初始化命令之间提供硬件所需的等待时间。第 24-27 行定义了状态机的 4 个状态：S_IDLE（空闲）、S_SEND_DATA（发送数据）、S_DELAY（延迟）和 S_ACK（完成响应）。

代码第 30-33 行声明了几个关键寄存器：init_cnt 用于记录当前执行到的初始化命令序号，delay_cnt 用于延时计数，state 和 next_state 分别表示当前状态和下一状态。第 37-41 行通过组合逻辑，将状态机状态与对外信号进行绑定，例如在 S_ACK 状态下拉高初始化完成信号，在 S_SEND_DATA 状态下发出 SPI 数据请求，在 S_DELAY 状态下拉高 SPI 发送结束信号。

代码第 43-48 行实现了状态寄存器的时序逻辑，在时钟上升沿更新 state。第 51-96 行是状态转移逻辑：当接收到初始化请求时，状态机从 S_IDLE 跳转到 S_SEND_DATA；当 SPI 完成一次数据传输后，进入 S_DELAY 状态；在延时满足条件并且初始化序列未完成时，根据 init_cnt 的不同值决定是否继续发送数据或保持延时；当所有命令执行完毕且延时完成时，进入 S_ACK 状态，并最终回到 S_IDLE，准备下一次初始化。

代码第 104-111 行是 init_cnt 的控制逻辑，每当 SPI 发送完成一个数据，init_cnt 自增，用于逐步遍历初始化命令序列。代码第 115-122 行实现了延时计数器逻辑，仅在 S_DELAY 状态下计数，否则清零。

代码第 127-209 行是初始化命令与数据的输出逻辑，根据 init_cnt 的值决定当前输出的命令或参数。例如 init_cnt=0 时发送软复位指令 SWRESET，init_cnt=1 时发送 SLPOUT 唤醒屏幕，init_cnt==2 时，发送命令 8'h3A（设置像素格式），紧接着是 8'h55，表示使用 RGB565 格式。init_cnt==4 时，发送命令 8'h36（芯片内存数据的读写扫描方向），紧接着是 8'h78，设置扫描方式。init_cnt=16 时设置正常显示模式 NORON，init_cnt=17 时开启显示 DISPON。每条指令对应的 tft_screen_init_data_o 为命令码或数据，tft_screen_init_dc_o 决定其是命令（0）还是数据（1）。

综上，spi_tft_screen_init 模块通过有限状态机控制 SPI 数据发送和延时逻辑，依次完成 TFT 屏幕的初始化命令序列。其核心作用是将复杂的初始化过程封装为一个可重复触发的过程，用户只需发出一次初始化请求，模块即可自动完成屏幕的启动和配置。

spi_master_driver 模块：

```
1.  `timescale 1ns / 1ps
2.
3.  // //模式 0：CPOL= 0，CPHA=0。SCK 串行时钟线空闲是为低电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
4.  // //模式 1：CPOL= 0，CPHA=1。SCK 串行时钟线空闲是为低电平，数据在 SCK 时钟的下降沿被采样，数据在 SCK 时钟的上升沿切换
5.  // //模式 2：CPOL= 1，CPHA=0。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的下降沿被采样，数据在 SCK 时钟的上升沿切换
6.  // //模式 3：CPOL= 1，CPHA=1。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
7.    //模式 3：CPOL= 1，CPHA=1。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
8.
9.  module spi_master_driver(
10.     //系统接口
11.     input                           sys_clk                 ,
12.     input                           sys_rst_n               ,
```

```
13.
14.     //用户接口
15.     input                           spi_start_i             ,// spi 开始信号
16.     input                           spi_end_i               ,// spi 结束信号
17.     input               [   7: 0]   spi_send_data_i         ,// spi 发送数据
18.     output      reg                 spi_send_ack_o          ,// spi 发送 8bit 数据完成信号
19.
20.     input                           lcd_dc_i                ,//数据还是命令信号输入
21.     output reg                      lcd_dc                  ,//数据还是命令信号输出
22.     //spi   端口
23.     output reg                      spi_sclk                ,//spi 时钟
24.     output reg                      spi_mosi                ,//spi 数据
25.     output                          spi_cs                  //spi 使能
26. );
27. localparam   IDLE = 3'b001,   //空闲状态
28.              DATA = 3'b010,   //发送数据状态
29.              STOP = 3'b100;   //停止状态
30.     reg                 [2:0]       cure_state;//状态寄存器
31.     reg                 [2:0]       next_state;//下一状态寄存器
32.
33.     reg                 [7:0]       spi_send_data_reg       ;//数据寄存器
34.     reg                 [3:0]       spi_send_data_bit_cnt   ;//数据发送 bit 计数器
35.
36. assign spi_cs = (cure_state == IDLE) ?1:0;  //片选信号，低电平有效，为空闲状态时拉高
37. always @(posedge sys_clk or negedge sys_rst_n)
38.     if(sys_rst_n == 1'b0 )
39.         cure_state <= IDLE;
40.     else
41.         cure_state <= next_state;
42. always @(*)
43.     case(cure_state)
44.         IDLE:begin   //空闲
45.             if(spi_start_i) //spi_start 开始信号来临时，开始进行数据发送
46.                 next_state = DATA;
47.             else
48.                 next_state = IDLE;
49.         end
50.         DATA:begin   //发送数据
51.             if(spi_send_data_bit_cnt == 7 &&   spi_sclk == 1'b0) //字节发送完毕
52.                 next_state = STOP;
53.             else
54.                 next_state = DATA;
55.         end
56.         STOP:next_state = IDLE; //停止
57.         default:next_state = IDLE;
58.     endcase
59.
60. //发送数据缓存
61. always@(posedge sys_clk or negedge sys_rst_n) begin
62.     if( sys_rst_n == 1'b0 )
63.         spi_send_data_reg <= 'd0;
64.     else if(spi_send_data_bit_cnt == 'd0)          //8bit 数据发送完成后，缓存新的数据
65.         spi_send_data_reg <= spi_send_data_i;
66.     else
67.         spi_send_data_reg <= spi_send_data_reg;
68. end
69. always@(posedge sys_clk or negedge sys_rst_n) begin
70.     if( sys_rst_n == 1'b0 )
```

```
71.             spi_send_ack_o <= 'd0;
72.             //发送完 8 位数据后，spi 发送 8bit 数据完成信号拉高
73.         else if(spi_send_data_bit_cnt == 'd7 && spi_sclk == 1'b0 && spi_cs == 1'b0)
74.             spi_send_ack_o <= 'd1;
75.         else
76.             spi_send_ack_o <= 'd0;
77.     end
78.     //数据是命令还是数据
79.     always@(posedge sys_clk or negedge sys_rst_n) begin
80.         if( sys_rst_n == 1'b0)
81.             lcd_dc <= 1'b1;
82.         else if( spi_start_i == 1'b1)  //接收到开始信号时，lcd_dc 赋值为 lcd_dc_i
83.             lcd_dc <= lcd_dc_i;
84.         else
85.             lcd_dc <= lcd_dc;
86.     end
87.     //产生 spi 时钟
88.     always@(posedge sys_clk or negedge sys_rst_n) begin
89.         if( sys_rst_n == 1'b0)
90.             spi_sclk <= 1'b1;
91.         else if(cure_state != DATA )
92.             spi_sclk <= 1'b1;
93.         else if( spi_cs == 1'b0 )
94.             spi_sclk <= ~spi_sclk;  //当 spi_cs 低电平时，翻转 spi_sclk，生成 40ns 周期的 sclk 时钟
95.         else
96.             spi_sclk <= 1'b1;
97.     end
98.     //数据发送 bit 数寄存器
99.     always@(posedge sys_clk or negedge sys_rst_n) begin
100.        if( sys_rst_n == 1'b0)
101.            spi_send_data_bit_cnt <= 'd0;
102.        //使用 SPI 模式三，所以数据在下降沿切换，所以当时钟为低电平时，数据计数器加 1
103.        else if( spi_cs == 1'b0 && spi_sclk == 1'b0)
104.            if( spi_send_data_bit_cnt == 'd7)//发送完 8 位数据，清零
105.                spi_send_data_bit_cnt <= 'd0;
106.            else
107.                spi_send_data_bit_cnt <= spi_send_data_bit_cnt + 1'b1;
108.        else if( spi_cs == 1'b0)
109.            spi_send_data_bit_cnt <= spi_send_data_bit_cnt;//在时钟上升沿时，保持不变
110.        else
111.            spi_send_data_bit_cnt <= 'd0;
112.    end
113.
114.    //spi 数据发送
115.    always@(posedge sys_clk or negedge sys_rst_n) begin
116.        if( sys_rst_n == 1'b0)
117.            spi_mosi <= 1'b1;
118.        else if( spi_cs == 1'b0)
119.            //将数据从高位到低位逐位发送
120.            spi_mosi <= spi_send_data_reg['d7 - spi_send_data_bit_cnt];
121.        else
122.            spi_mosi <= spi_mosi;
123.    end
124.
125.    endmodule
```

spi_master_driver 模块是一个 SPI 主机驱动模块，用于按照 SPI 模式 3（CPOL=1，CPHA=1）与外设通信，实现字节数据的发送控制。使用时钟相位时钟极性均设置为

1，也就是时钟低电平有效，数据采样边沿为偶边沿。本模块为其他模块发送配置信息，也负责图像数据的发送。模块主要目的是将复杂的 SPI 时序操作封装起来，用户只需提供开始信号和待发送数据，即可完成 SPI 数据传输，并获取发送完成的反馈信号。模块同时处理 SPI 时钟生成、数据输出以及命令/数据选择信号。

代码第 9 行定义了模块 spi_master_driver，作为 SPI 主控驱动顶层模块。第 11-12 行为系统时钟 sys_clk 与复位 sys_rst_n 输入信号。第 15-21 行定义了用户接口,包括：spi_start_i：触发 SPI 发送开始；spi_end_i：外部 SPI 发送结束信号；spi_send_data_i：待发送 8 位数据；spi_send_ack_o：8 位数据发送完成标志；lcd_dc_i：输入命令/数据选择信号；lcd_dc：输出命令/数据选择信号给 SPI 外设。

第 23-25 行定义 SPI 硬件接口：spi_sclk（SPI 时钟）、spi_mosi（数据输出）、spi_cs（片选信号，低电平有效）。第 36 行将片选信号与状态机绑定：空闲状态下拉高，表示 SPI 不选择外设；其他状态为低电平，表示 SPI 正在通信。

代码第 27-31 行定义了三状态状态机：

IDLE（空闲）状态：在空闲状态下，模块等待 spi_start_i 信号的触发。当上层模块触发这个开始信号时，状态机会切换到数据发送状态，处理发送逻辑。

DATA（数据发送）状态：当进入数据发送状态时，模块通过 SPI 总线逐位发送数据。每次发送一个字节（8 位），在数据传输过程中，信号 spi_send_data_bit_cnt 用于记录当前发送的位数。每当一个字节数据发送完毕时，状态机会进入停止状态。

STOP（停止）状态：数据发送完成后，状态机会进入停止状态，SPI 片选信号被拉高，表示传输结束。之后，状态机重新回到空闲状态，等待上层模块触发下一次数据传输请求。

第 33-34 行定义数据寄存器与位计数器：spi_send_data_reg 缓存当前要发送的字节；spi_send_data_bit_cnt 记录当前发送的 bit 位。

代码第 37-41 行为状态寄存器时序逻辑，根据 sys_clk 更新当前状态。第 42-58 行为状态机组合逻辑，控制状态转换条件，例如数据发送完成后跳转到 STOP 状态。

第 61-68 行为发送数据寄存器逻辑:当位计数器为 0 时,缓存新的 SPI 发送数据；否则保持原值。第 69-77 行生成发送完成信号 spi_send_ack_o，在每个字节发送完成且片选有效时拉高，通知用户下一数据可以准备。

第 79-86 行控制命令/数据输出信号 lcd_dc：当接收到 SPI 开始信号时，更新为输入的命令/数据标志，否则保持原值。

第 88-97 行产生 SPI 时钟 spi_sclk：在空闲或非发送状态保持高电平；数据发送状态下，通过翻转 spi_sclk 生成 SPI 时钟， SPI 时钟（spi_sclk）的生成由状态机控制。在数据发送状态下，当 spi_cs 低电平时，将时钟信号周期性地反转，生成一个周期为 40ns 的时钟，用于同步 MOSI 数据线的变化。模块使用的是 SPI 模式 3，因此时钟在空闲时为高电平，数据在时钟的上升沿被采样，数据在下降沿切换。

第 99-112 行为位计数器逻辑：仅在 SPI 片选低电平且时钟为低电平时自增（因为模式 3 数据在下降沿切换），完成 8 位计数后清零。其他条件下保持或清零。

第 115-123 行实现 SPI 数据输出 spi_mosi，将缓存字节的高位到低位逐位发送到 MOSI 引脚，仅在片选有效时更新数据。

SPI 片选信号（spi_cs）在代码第 37 行对该信号进行了赋值，在 IDLE 状态下，spi_cs 被拉高，表示 SPI 总线处于空闲状态，主设备不与外设通信。而在 DATA 状态下，spi_cs 被拉低，表示 SPI 通信已经启动。

接下来 spi_mosi 信号：在代码第 114~121 行：每当 SPI 时钟为低电平并且 spi_cs 低电平时，待发送的数据将按照 spi_send_data_bit_cnt 索引从高位到低位将数据从数据寄存器 spi_send_data_reg 中取出，并通过 spi_mosi 信号发送。当发送完一个字节的数据后，模块通过 spi_send_ack_o 信号通知上层控制逻辑，表示数据发送已完成。该信号将在每次发送完 8 位数据时拉高，作为上层逻辑知晓一字节数据发送完成的反馈。

此外，本次实验所用到的 SPI 显示屏型号 GMT024-8pinSPI_LCM 使用 Sitronix 公司的 ST7789VM 芯片来控制 tft 液晶显示屏幕的显示，其还使用 DC 信号用于指示当前传输的数据是命令数据还是实际数据，在本模块中 DC 信号由上层逻辑传入，当 spi_start_i 信号触发时，lcd_dc 会被更新为输入信号 lcd_dc_i 的值，配合 spi 的发送过程，指示发送的每个字节数据是图像数据还是控制命令。

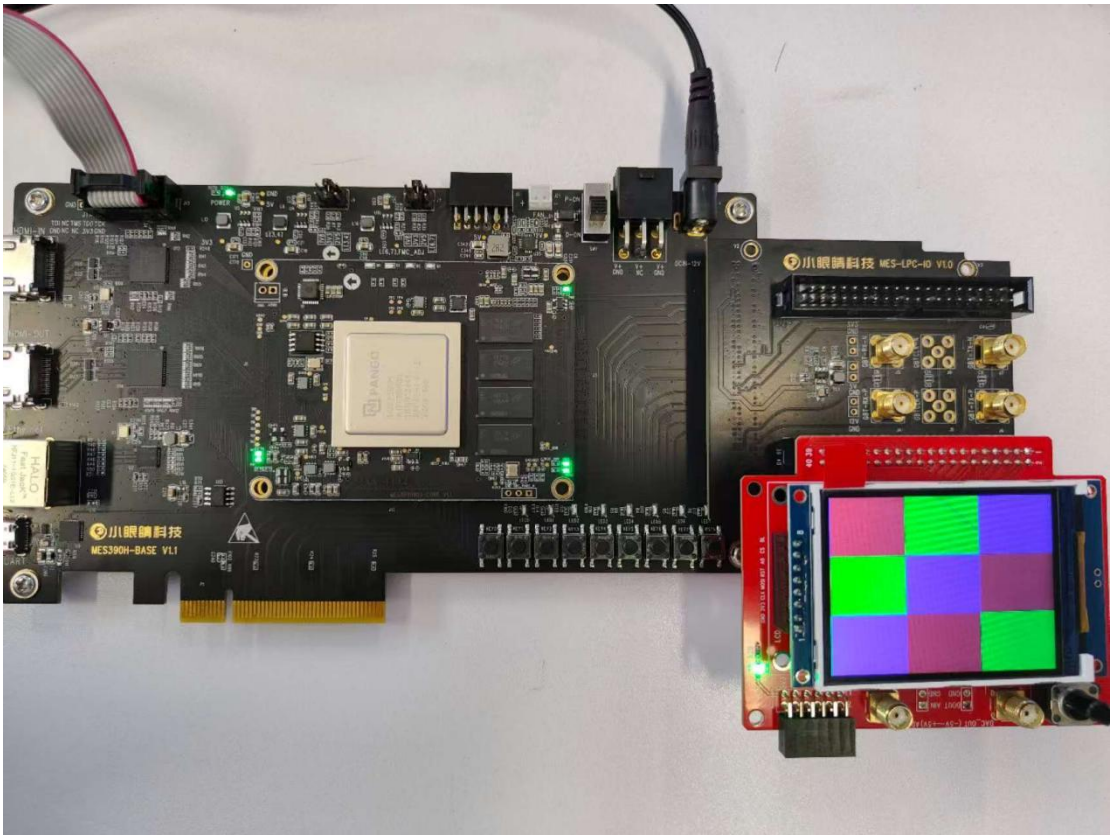
总的来说 spi_master_driver 模块封装了 SPI 模式 3 的数据发送全过程，包括片选、时钟翻转、数据逐位发送、命令/数据控制和发送完成反馈。用户只需触发开始信号并提供 8 位数据，模块即可自动完成 SPI 数据传输，保证时序正确且无需外部手动控制 SPI 时钟和位传输。

19.4. 实验现象

新建工程，添加 rtl 代码，添加 fdc 文件；本实验管脚约束如下：（示例工程中也有仿真代码，读者若不清楚时序也可自行仿真分析）

	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD	DRIVE	BUS_KEEPER
1	lcd_blk	OUTPUT	A25	BANKL3	3.3	LVCMOS33	4	UNUSED
2	lcd_dc	OUTPUT	F26	BANKL3	3.3	LVCMOS33	4	UNUSED
3	lcd_reset	OUTPUT	E26	BANKL3	3.3	LVCMOS33	4	UNUSED
4	lcd_spi_cs	OUTPUT	A26	BANKL3	3.3	LVCMOS33	4	UNUSED
5	lcd_spi_mosi	OUTPUT	E23	BANKL3	3.3	LVCMOS33	4	UNUSED
6	lcd_spi_sclk	OUTPUT	D23	BANKL3	3.3	LVCMOS33	4	UNUSED
7	sys_clk	INPUT	T26	BANKL5	3.3	HSLVCMOS33		UNUSED
8	sys_rst_n	INPUT	P28	BANKL5	3.3	HSLVCMOS33		UNUSED

综合生成比特流，连接好扩展板卡，将程序下载进入板卡。板卡连接图及最终实验现象如下图所示。



20. 光纤通信测试实验例程

20.1. 实验简介

实验使用“小眼睛科技”公司的 MES390H 开发板, MES-HPC-QSFP 扩展子卡完成光纤回环通信测试实验，扩展子卡使用如下图所示。

20.2. 实验原理

HSSTHP 是内置于 Titan2 系列产品的高速串行接口模块,数据速率高达 12.5Gbps。除了 PMA、HSSTHP 集成了丰富的 PCS 功能，可灵活应用于各种串行协议标准。PG2T390H-FFBG900 包含 4 个 HSSTHP，共可支持 16 个全双工收发 LANE。 每个 HSSTHP 支持一至四个全双工收发 Lane。HSSTHP 主要特性包括：

- 支持数据速率：0.6Gbps-12.5Gbps
- 灵活的参考时钟选择方式
- 发送通道和接收通道数据率可独立配置
- 可编程输出摆幅和去加重
- 接收端自适应均衡器
- PMATx/Rx 支持扩频
- 数据通道支持 8bitonly,10bitonly, 8b10b 8bit, 16bit only, 20bit only, 8b10b 16bit, 32bit only, 40bit only, 8b10b 32bit, 64bit only, 80bit only, 8b10b 64bit, 64b66b/64b67b 16bit, 64b66b/64b67b 32bit, 128b130b 等模式。 □
- 可灵活配置的 PCS，可支持 PCIExpressGEN1/2/3，XAUI，千兆以太网，CPRI，SRIO 等协议
- 灵活的字节边界对齐功能
- 支持 RxClockSlip 功能以保证固定的接收延迟

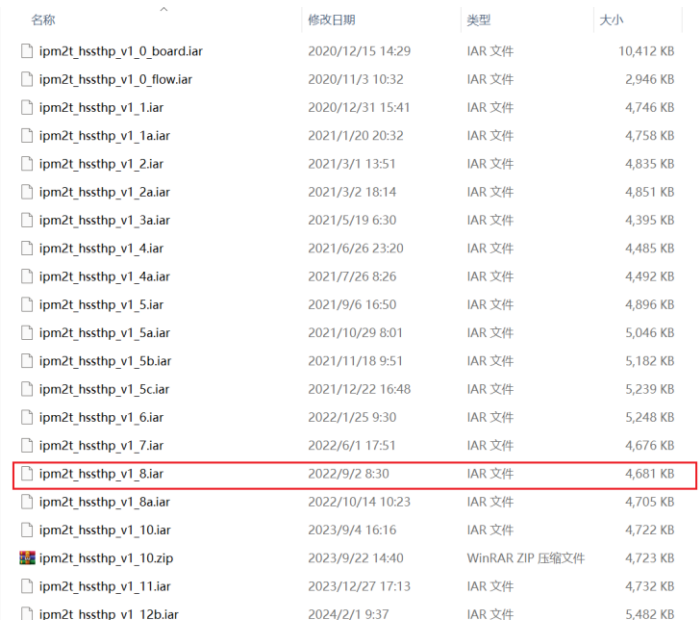
- 支持协议标准 8b10b,64b66b/64b67b,128b130b 编码解码
- 灵活的 CTC 方案
- 支持 x2 和 x4 的通道绑定
- HSSTHP 的配置支持动态修改
- 近端环回和远端环回模式
- 内置 PRBS 功能

20.3. 工程说明

20.3.1. 安装 HSST IP 核

PDS 安装后，需手动添加 HSST IP，请按以下步骤完成：

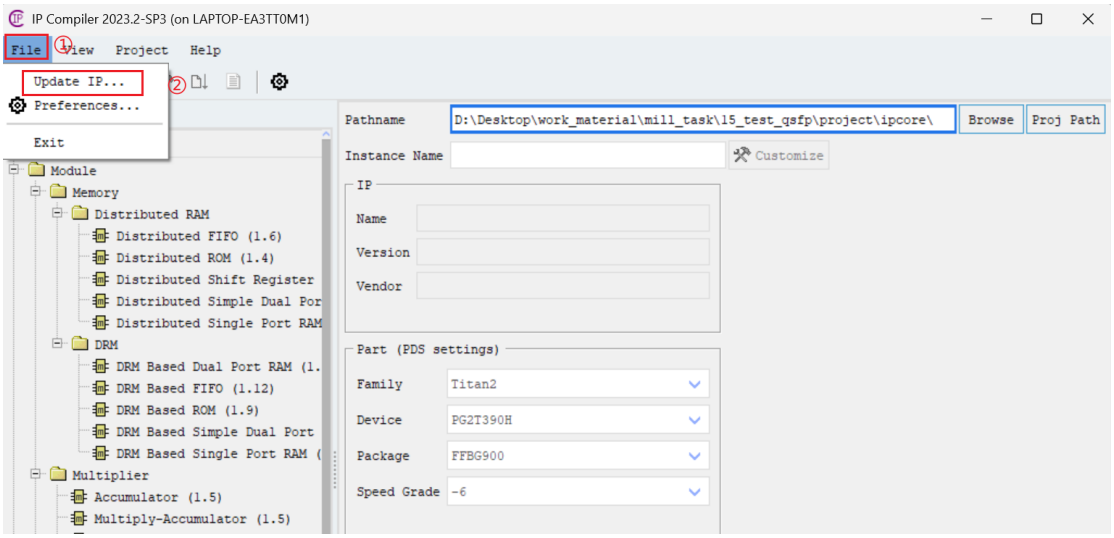
（1）HSST IP 文件：在路径 6_IP_setup_packet\HSST 下选择 ipm2t_hssthp_v1_8.iar。



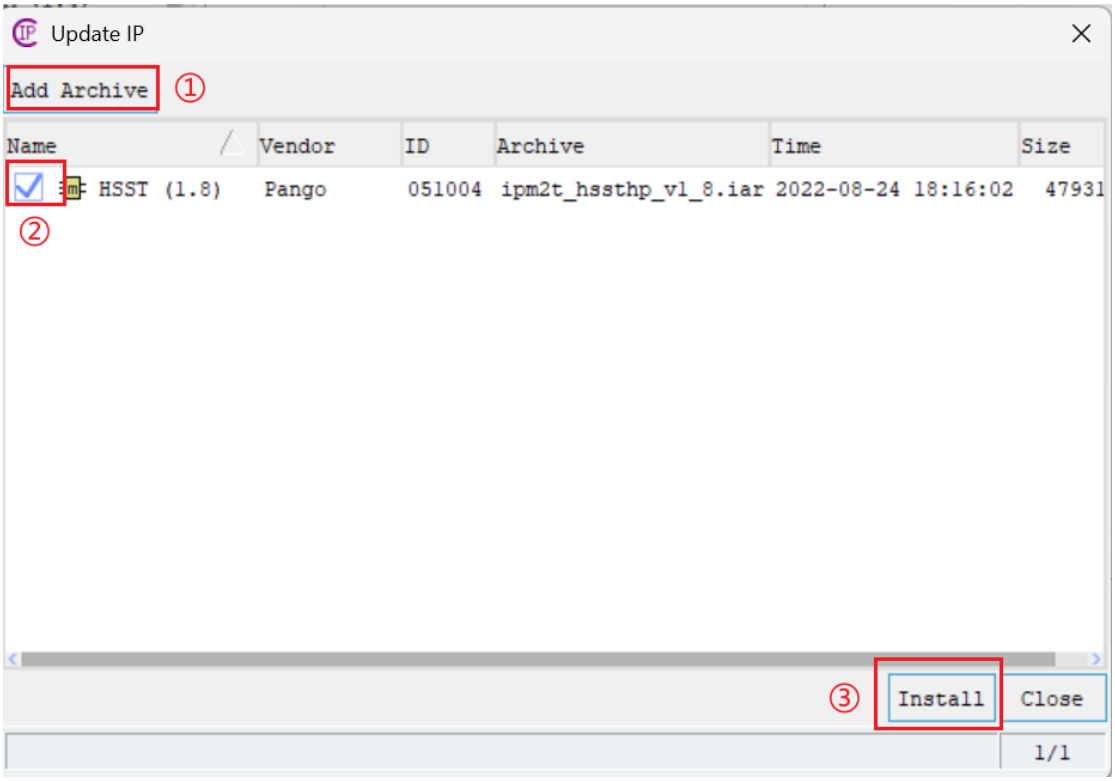
名称	修改日期	类型	大小
ipm2t_hssthp_v1_0_board.iar	2020/12/15 14:29	IAR 文件	10,412 KB
ipm2t_hssthp_v1_0_flow.iar	2020/11/3 10:32	IAR 文件	2,946 KB
ipm2t_hssthp_v1_1.iar	2020/12/31 15:41	IAR 文件	4,746 KB
ipm2t_hssthp_v1_1a.iar	2021/1/20 20:32	IAR 文件	4,758 KB
ipm2t_hssthp_v1_2.iar	2021/3/1 13:51	IAR 文件	4,835 KB
ipm2t_hssthp_v1_2a.iar	2021/3/2 18:14	IAR 文件	4,851 KB
ipm2t_hssthp_v1_3a.iar	2021/5/19 6:30	IAR 文件	4,395 KB
ipm2t_hssthp_v1_4.iar	2021/6/26 23:20	IAR 文件	4,485 KB
ipm2t_hssthp_v1_4a.iar	2021/7/26 8:26	IAR 文件	4,492 KB
ipm2t_hssthp_v1_5.iar	2021/9/6 16:50	IAR 文件	4,896 KB
ipm2t_hssthp_v1_5a.iar	2021/10/29 8:01	IAR 文件	5,046 KB
ipm2t_hssthp_v1_5b.iar	2021/11/18 9:51	IAR 文件	5,182 KB
ipm2t_hssthp_v1_5c.iar	2021/12/22 16:48	IAR 文件	5,239 KB
ipm2t_hssthp_v1_6.iar	2022/1/25 9:30	IAR 文件	5,248 KB
ipm2t_hssthp_v1_7.iar	2022/6/1 17:51	IAR 文件	4,676 KB
ipm2t_hssthp_v1_8.iar	2022/9/2 8:30	IAR 文件	4,681 KB
ipm2t_hssthp_v1_8a.iar	2022/10/14 10:23	IAR 文件	4,705 KB
ipm2t_hssthp_v1_10.iar	2023/9/4 16:16	IAR 文件	4,722 KB
ipm2t_hssthp_v1_10.zip	2023/9/22 14:40	WinRAR ZIP 压缩文件	4,723 KB
ipm2t_hssthp_v1_11.iar	2023/12/27 17:13	IAR 文件	4,732 KB
ipm2t_hssthp_v1_12b.iar	2024/2/1 9:37	IAR 文件	5,482 KB

（2）IP 安装步骤：

在 IP Compiler 界面，选择 File-->Update IP。

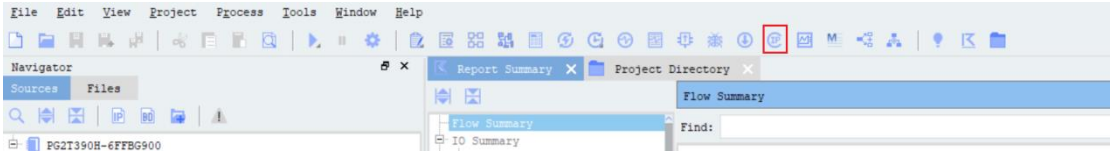


在 Update IP 界面，点击 Add Archive 按键，选择对应的 HSST IP 文件，并勾选住要下载的文件，然后点击 Install 按键下载即完成对 IP 的安装。

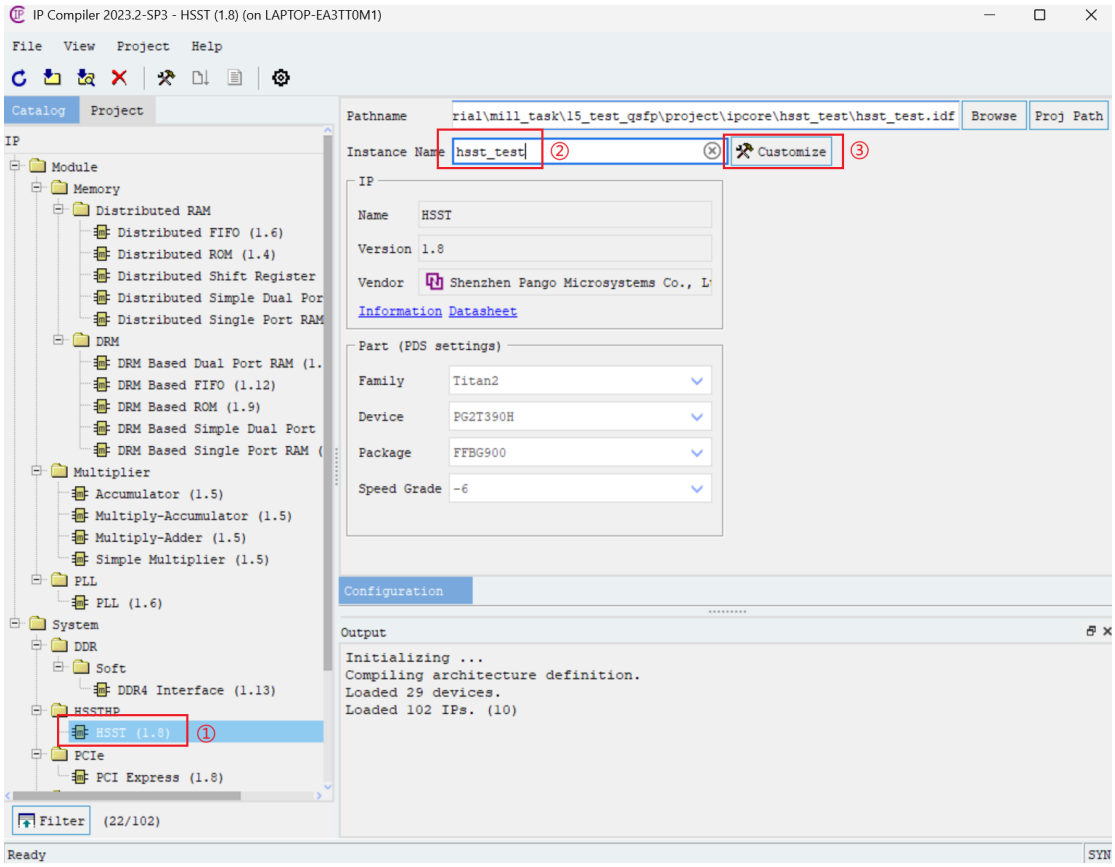


20.3.2. 例程介绍

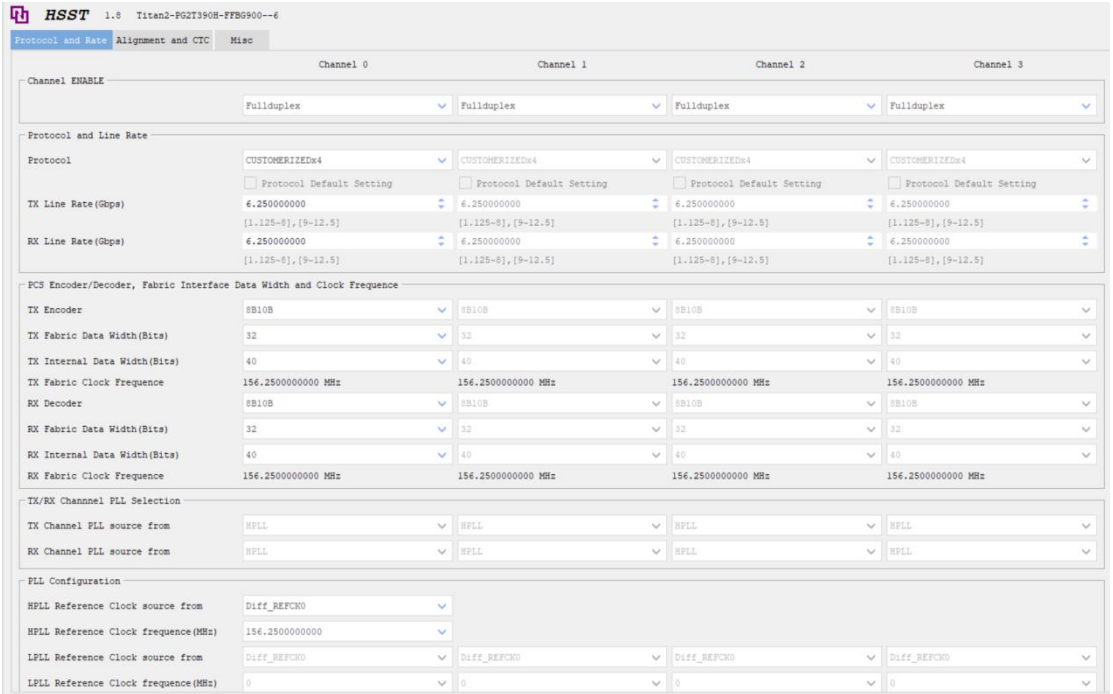
打开 PDS 软件，新建工程，打开 IP Compiler 按键。



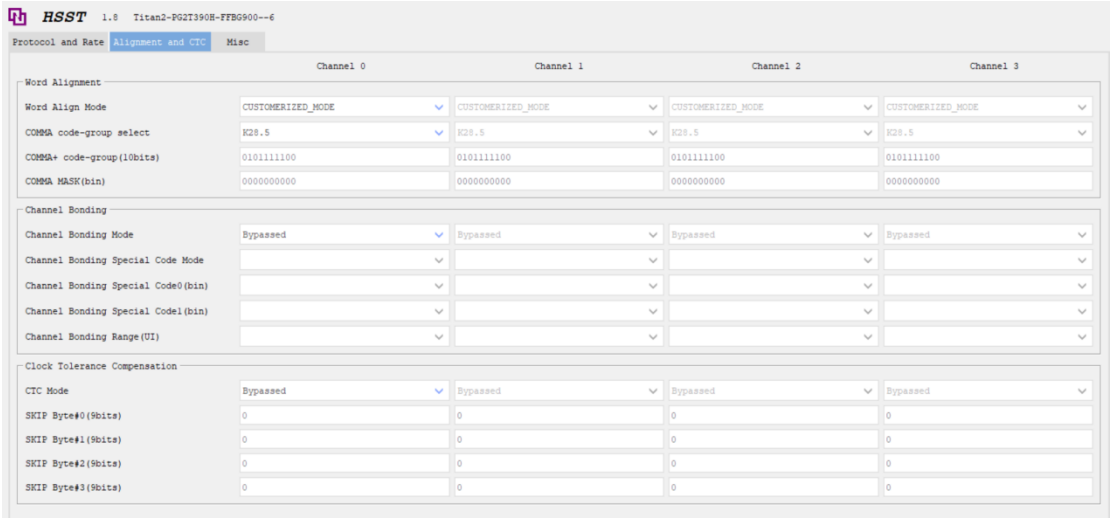
在 IP Cimpiler 界面，选择 HSST IP，为 IP 取名，然后点击 Customize 按键，即可创建对应的 IP。



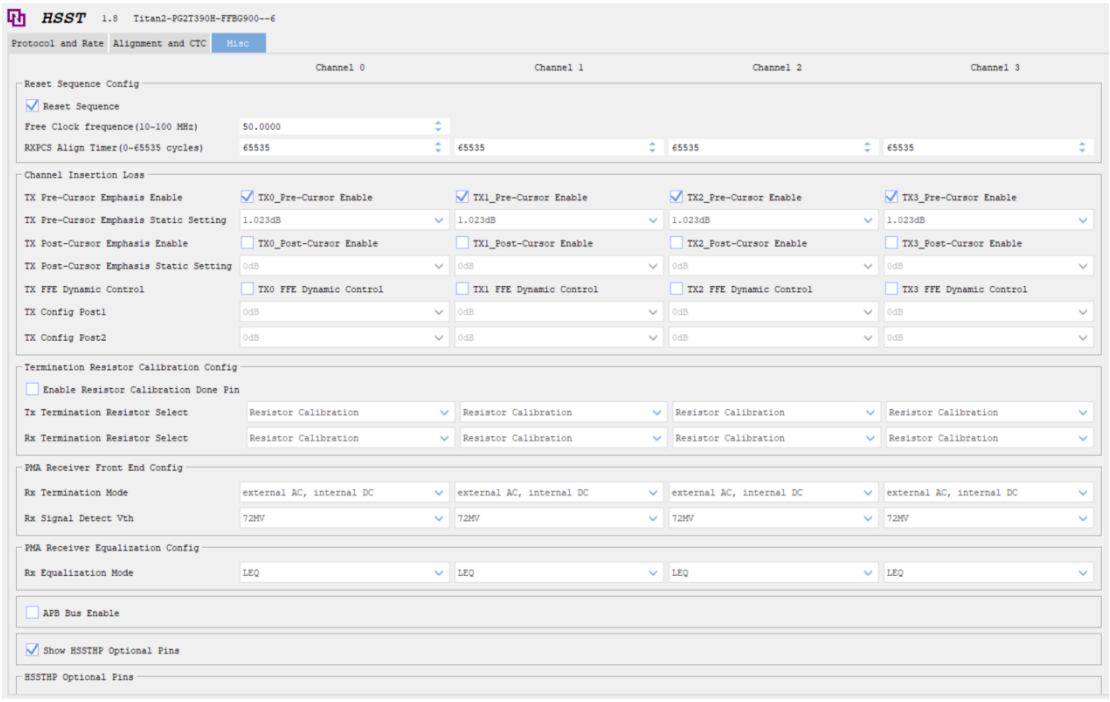
在 HSST 设置界面中 Protocol and Rate 按照如下设置。



Alignment and CTC 按照如下设置：



Misc 按照如下设置，点击 Generate 可生成 HSST IP；



在配置好 hsst IP 后，编写对应的光纤回环通信代码，代码实现简略如下。第 26 行定义代码所有通道固定发送同一组测试数据 0x112233bc 和 k 字符 0x0001，用于仿真和硬件 CHECK。hsst_test_qsfp_quad1 和 hsst_test_qsfp_quad2 分别管理 QSFP1 和 QSFP2 的 SerDes 收发，实现包括数据用户接口等功能。SerDes 解码接收到的串并数据后，需要字对齐，确保高速接收并行数据边界准确，word_align 模块便是实现这部分功能。

```
1.  module test_qsfp_top (
2.      input    wire    sys_clk,            // 系统时钟
3.      input    wire    rst_n,              // 异步复位
4.
5.      // QSFP1 Lane 信号
6.      input    wire    q1_i_refckp_0, q1_i_refckn_0,
7.      input    wire    q1_i_p_l0rxn, q1_i_p_l0rxp, // RX Lane0
8.      // ... Lane1~3，省略若干端口 ...
9.      output   wire    q1_o_p_l0txn, q1_o_p_l0txp, // TX Lane0
```

```
    // ... Lane1~3 省略 ...

    // QSFP2 Lane 信号，结构同 QSFP1，略

    // QSFP 管理与状态信号
    input   wire    qsfp1_intl, qsfp1_modprsl,
    output  wire    qsfp1_modsell, qsfp1_resetl, qsfp1_lpmode,
    input   wire    qsfp2_intl, qsfp2_modprsl,
    output  wire    qsfp2_modsell, qsfp2_resetl, qsfp2_lpmode
);

// 时钟缓冲
    wire    i_free_clk;
GTP_CLKBUFG GTP_CLKBUFG_inst (
      .CLKOUT(i_free_clk),
      .CLKIN(sys_clk)
);

// 固定测试数据定义（每通道循环）
    assign  q1_tx0_data     =   32'h112233bc;
    assign  q1_tx0_kchar    =   4'b0001;
// ... 其余通道同上，略

// QSFP1、2 发送/接收模块例化
hsst_test_qsfp hsst_test_qsfp_quad1 (
      ...
);
hsst_test_qsfp hsst_test_qsfp_quad2 (
      ...
 );

// RX 字对齐功能
word_align word_align_q1ch0(
    .rx_clk(...),
    .gt_rx_data(...),
    .gt_rx_ctrl(...),
    .rx_data_align(...),
    .rx_ctrl_align(...)
);
// ...其余通道同上，略

endmodule
```

test_qsfp_top 是一个基于 QSFP（Quad Small Form-factor Pluggable）高速接口的顶层测试设计，主要功能是完成 QSFP1 与 QSFP2 模块的收发通道建立、测试数据驱动、状态信号控制以及接收侧字对齐处理。模块输入包括系统时钟 sys_clk 和异步复位 rst_n，并定义了 QSFP1、QSFP2 的高速差分收发通道端口及模块管理接口。收发通道包括四个 Lane，每个 Lane 分别包含接收（RX）和发送（TX）差分对信号；管理与状态接口则包括 intl（中断）、modprsl（模块在位检测）、modsell（模块选择）、resetl（复位控制）和 lpmode（低功耗模式控制）等信号。

在时钟管理部分，代码通过实例化 GTP_CLKBUFG 缓冲模块，将外部输入的系统时钟 sys_clk 分配为内部全局时钟 i_free_clk，为后续的高速收发逻辑提供稳定的时钟

源。

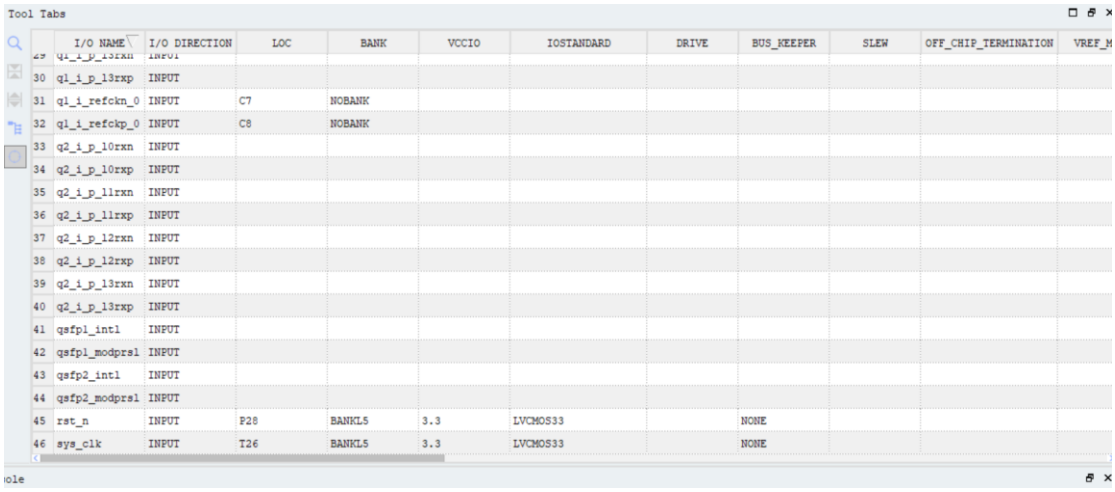
在测试数据部分，模块定义了固定的 32 位发送数据模式与 K 字符控制信号。例如，q1_tx0_data 被设置为 32'h112233bc，对应的 q1_tx0_kchar 为 4'b0001，用于在 Lane0 上输出固定测试码流。这种方式简化了验证过程，有助于确认收发链路的连通性和稳定性。其余通道的数据定义与 Lane0 类似。

在收发逻辑部分，模块分别例化了两个 hsst_test_qsfp 子模块，分别对应 QSFP1 与 QSFP2 的四通道高速收发器。子模块内部实现了高速串并转换、协议封装、发送与接收路径的数据处理等功能，顶层仅需提供测试数据与控制信号即可完成通道验证。

接收链路中，为保证数据对齐，模块在每个接收通道实例化了 word_align 字对齐单元。该单元输入接收时钟、GT 接收数据和控制信号，并在内部完成字边界检测与数据重排，输出对齐后的 rx_data_align 和 rx_ctrl_align，确保后级逻辑能够正确解析数据。

模块通过统一的时钟分配、固定的测试数据源、QSFP 收发子模块的例化以及接收端的字对齐处理，形成了一个完整的 QSFP 高速链路测试平台。它能够在不依赖上层业务逻辑的情况下，快速验证 QSFP 模块收发能力、Lane 链路稳定性以及基本控制接口的正确性，为后续系统集成和协议层开发提供可靠的测试环境。

之后需要对部分管脚进行约束，详情请查看原理图。


Tool Tabs

	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD	DRIVE	BUS_KEEPER	SLEW	OFF_CHIP_TERMINATION	VREF_M
30	q1_i_p_l3rxp	INPUT									
31	q1_i_refckn_0	INPUT	C7	NOBANK							
32	q1_i_refckp_0	INPUT	C8	NOBANK							
33	q2_i_p_l0rxn	INPUT									
34	q2_i_p_l0rxp	INPUT									
35	q2_i_p_l1rxn	INPUT									
36	q2_i_p_l1rxp	INPUT									
37	q2_i_p_l2rxn	INPUT									
38	q2_i_p_l2rxp	INPUT									
39	q2_i_p_l3rxn	INPUT									
40	q2_i_p_l3rxp	INPUT									
41	qsfp1_intl	INPUT									
42	qsfp1_modprsl	INPUT									
43	qsfp2_intl	INPUT									
44	qsfp2_modprsl	INPUT									
45	rst_n	INPUT	P28	BANKL5	3.3	LVCMOS33		NONE			
46	sys_clk	INPUT	T26	BANKL5	3.3	LVCMOS33		NONE			

同时，按照 HSST IP 使用指南规定，在应用中，需要根据实际单板通过 PDS 在.fdc 文件中对 HSSTHP 的 PAD_REFCLKP、 PAD_REFCLKN 进行管脚位置约束。QSFP 扩展子卡使用的是 HSSTHP_1 和 HSSTHP_2，因此需要约束这部分管脚位置。

器件型号/封装	模块名称	约束位置	对应管脚:PAD_REFCLKP/N
PG2T 390H FFBG900	HSSTHP_1	HSSTHP_664_1836	PAD_REFCLKP0/N0: C8/ C7
			PAD_REFCLKP1/N1: E8/E7
	HSSTHP_2	HSSTHP_664_1530	PAD_REFCLKP0/N0: G8/ G7
			PAD_REFCLKP1/N1: J8/J7
	HSSTHP_3	HSSTHP_664_1224	PAD_REFCLKP0/N0: L8/ L7
			PAD_REFCLKP1/N1: N8/N7
	HSSTHP_4	HSSTHP_664_918	PAD_REFCLKP0/N0: R8/ R7
			PAD_REFCLKP1/N1: U8/U7

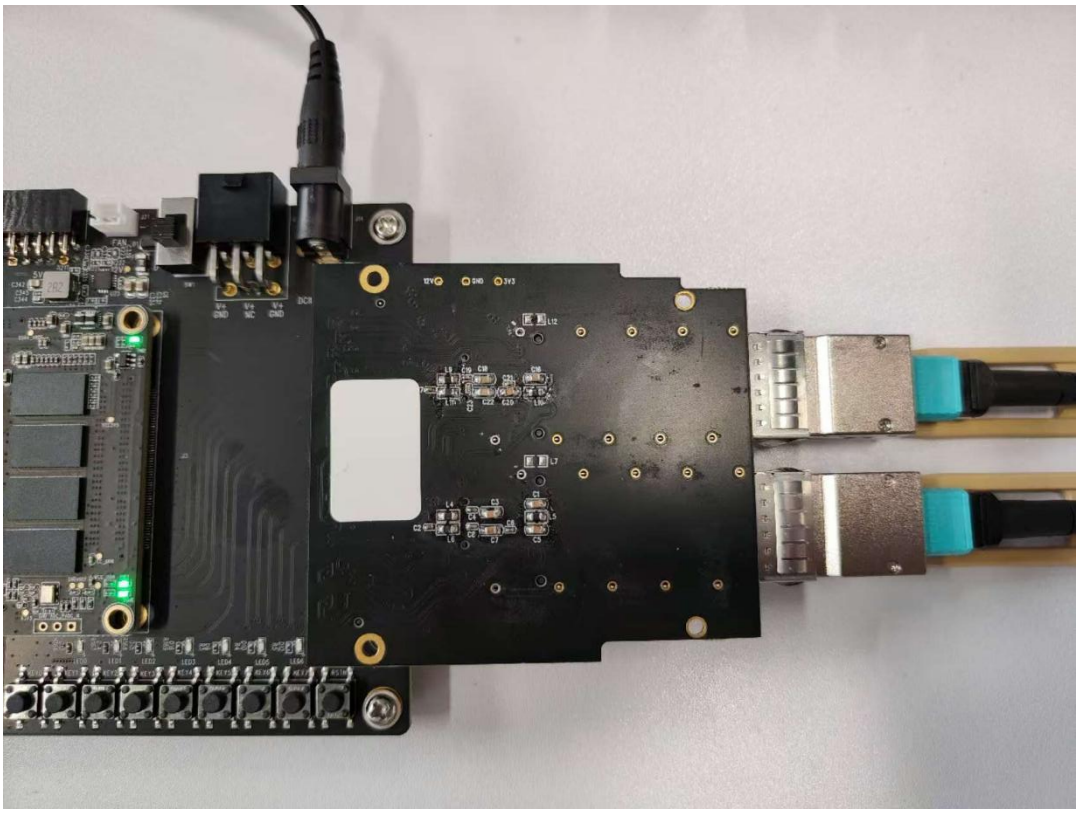
打开生成的 IP 文件夹“pnr\example_design”路径下的 fdc 文件，将 HSSTHP_1 和 HSSTHP_2 的管脚约束复制到自己工程的 fdc 文件中，取消注释，同时将 U_INST 修改为自己例化的 IP 名称。

```
#HSSTHP_1
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL0_ENABLE.U_LANE0_WRAP.U_LANE0} {PAP_LOC} {HSSTHP_664_1836:U0_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.HPLL_ENABLE.U_HPLL_WRAP.U_HPLL} {PAP_LOC} {HSSTHP_664_1836:U_HSSTHP_COMMON}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL1_ENABLE.U_LANE1_WRAP.U_LANE1} {PAP_LOC} {HSSTHP_664_1836:U1_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL2_ENABLE.U_LANE2_WRAP.U_LANE2} {PAP_LOC} {HSSTHP_664_1836:U2_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL3_ENABLE.U_LANE3_WRAP.U_LANE3} {PAP_LOC} {HSSTHP_664_1836:U3_HSSTHP_LANE}
#HSSTHP_2
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL0_ENABLE.U_LANE0_WRAP.U_LANE0} {PAP_LOC} {HSSTHP_664_1530:U0_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.HPLL_ENABLE.U_HPLL_WRAP.U_HPLL} {PAP_LOC} {HSSTHP_664_1530:U_HSSTHP_COMMON}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL1_ENABLE.U_LANE1_WRAP.U_LANE1} {PAP_LOC} {HSSTHP_664_1530:U1_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL2_ENABLE.U_LANE2_WRAP.U_LANE2} {PAP_LOC} {HSSTHP_664_1530:U2_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL3_ENABLE.U_LANE3_WRAP.U_LANE3} {PAP_LOC} {HSSTHP_664_1530:U3_HSSTHP_LANE}
#HSSTHP_3
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL0_ENABLE.U_LANE0_WRAP.U_LANE0} {PAP_LOC} {HSSTHP_664_1224:U0_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.HPLL_ENABLE.U_HPLL_WRAP.U_HPLL} {PAP_LOC} {HSSTHP_664_1224:U_HSSTHP_COMMON}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL1_ENABLE.U_LANE1_WRAP.U_LANE1} {PAP_LOC} {HSSTHP_664_1224:U1_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL2_ENABLE.U_LANE2_WRAP.U_LANE2} {PAP_LOC} {HSSTHP_664_1224:U2_HSSTHP_LANE}
#define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL3_ENABLE.U_LANE3_WRAP.U_LANE3} {PAP_LOC} {HSSTHP_664_1224:U3_HSSTHP_LANE}
#HSSTHP_4 defualt instance quad 4
define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL0_ENABLE.U_LANE0_WRAP.U_LANE0} {PAP_LOC} {HSSTHP_664_918:U0_HSSTHP_LANE}
define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.HPLL_ENABLE.U_HPLL_WRAP.U_HPLL} {PAP_LOC} {HSSTHP_664_918:U_HSSTHP_COMMON}
define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL1_ENABLE.U_LANE1_WRAP.U_LANE1} {PAP_LOC} {HSSTHP_664_918:U1_HSSTHP_LANE}
define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL2_ENABLE.U_LANE2_WRAP.U_LANE2} {PAP_LOC} {HSSTHP_664_918:U2_HSSTHP_LANE}
define_attribute {i:U_INST.U_GTP_HSSTHP_WRAPPER.CHANNEL3_ENABLE.U_LANE3_WRAP.U_LANE3} {PAP_LOC} {HSSTHP_664_918:U3_HSSTHP_LANE}
```

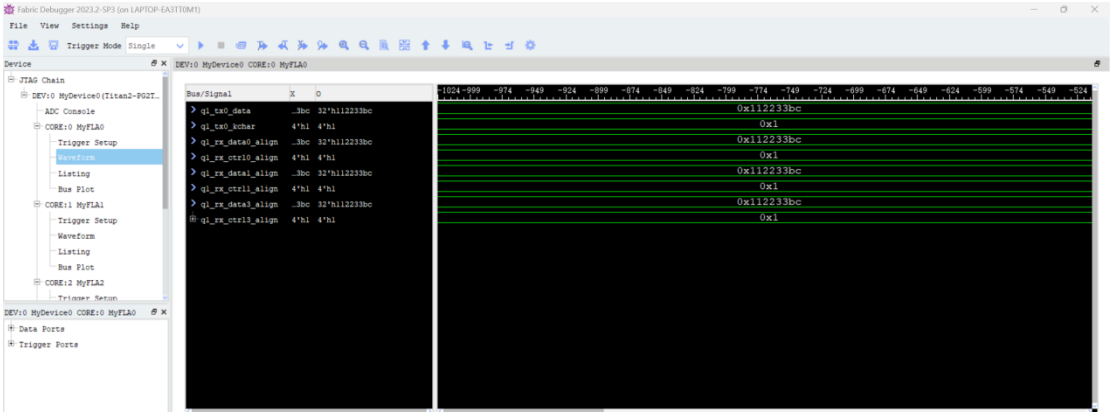
同时，复制时钟约束语句，将 U_INST 修改为自己例化的 IP 名称，对时钟进行约束。具体参考例程的 fdc 文件。

```
###==== BEGIN Clocks
create_clock -name {free_clk} [get_ports {i_free_clk}] -period {10} -waveform {0.000 5.000}
create_clock -name {p_clk2core_tx_0} [get_pins {U_INST.o_p_clk2core_tx_0}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_tx_1} [get_pins {U_INST.o_p_clk2core_tx_1}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_tx_2} [get_pins {U_INST.o_p_clk2core_tx_2}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_tx_3} [get_pins {U_INST.o_p_clk2core_tx_3}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_rx_0} [get_pins {U_INST.o_p_clk2core_rx_0}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_rx_1} [get_pins {U_INST.o_p_clk2core_rx_1}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_rx_2} [get_pins {U_INST.o_p_clk2core_rx_2}] -period {6.4} -waveform {0 3.2}
create_clock -name {p_clk2core_rx_3} [get_pins {U_INST.o_p_clk2core_rx_3}] -period {6.4} -waveform {0 3.2}
###==== END Clocks
###==== BEGIN "set_clock_groups"
set_clock_groups -name free_clk        -asynchronous -group [get_clocks {free_clk}]
set_clock_groups -name p_clk2core_tx_0 -asynchronous -group [get_clocks {p_clk2core_tx_0}]
set_clock_groups -name p_clk2core_tx_1 -asynchronous -group [get_clocks {p_clk2core_tx_1}]
set_clock_groups -name p_clk2core_tx_2 -asynchronous -group [get_clocks {p_clk2core_tx_2}]
set_clock_groups -name p_clk2core_tx_3 -asynchronous -group [get_clocks {p_clk2core_tx_3}]
set_clock_groups -name p_clk2core_rx_0 -asynchronous -group [get_clocks {p_clk2core_rx_0}]
set_clock_groups -name p_clk2core_rx_1 -asynchronous -group [get_clocks {p_clk2core_rx_1}]
set_clock_groups -name p_clk2core_rx_2 -asynchronous -group [get_clocks {p_clk2core_rx_2}]
set_clock_groups -name p_clk2core_rx_3 -asynchronous -group [get_clocks {p_clk2core_rx_3}]
###==== END "set_clock_groups"
###==== BEGIN "Inputs/Outpusts"
###==== END "Inputs/Outpusts"
###==== BEGIN "Delay Paths"
###==== END "Delay Paths"
```

20.4. 实验现象



把光纤两端接入 SFP0 和 SFP1 接口，打开 Fabric Debugger 软件，加载对应代码后抓取数据，可以看到发送的数据和接收的数据是一致的。



说明：

K 码对应的是 bc，当该字节为 bc 时，K 码为 1。所以当 K 码变化规律固定，且数据只出现移位时，数据是正确的。

例如收到的数据是 0xbcc5bcc5，该数据为 32bit，对应 4 字节，与 rxk 相对应，所以此时 bc 码出现在第 2 个字节和第 4 个字节。所以对应的 o_rxk 为 4'b1010 即 16 进制为 4'ha，也就是 Debugger 显示的 0xa。

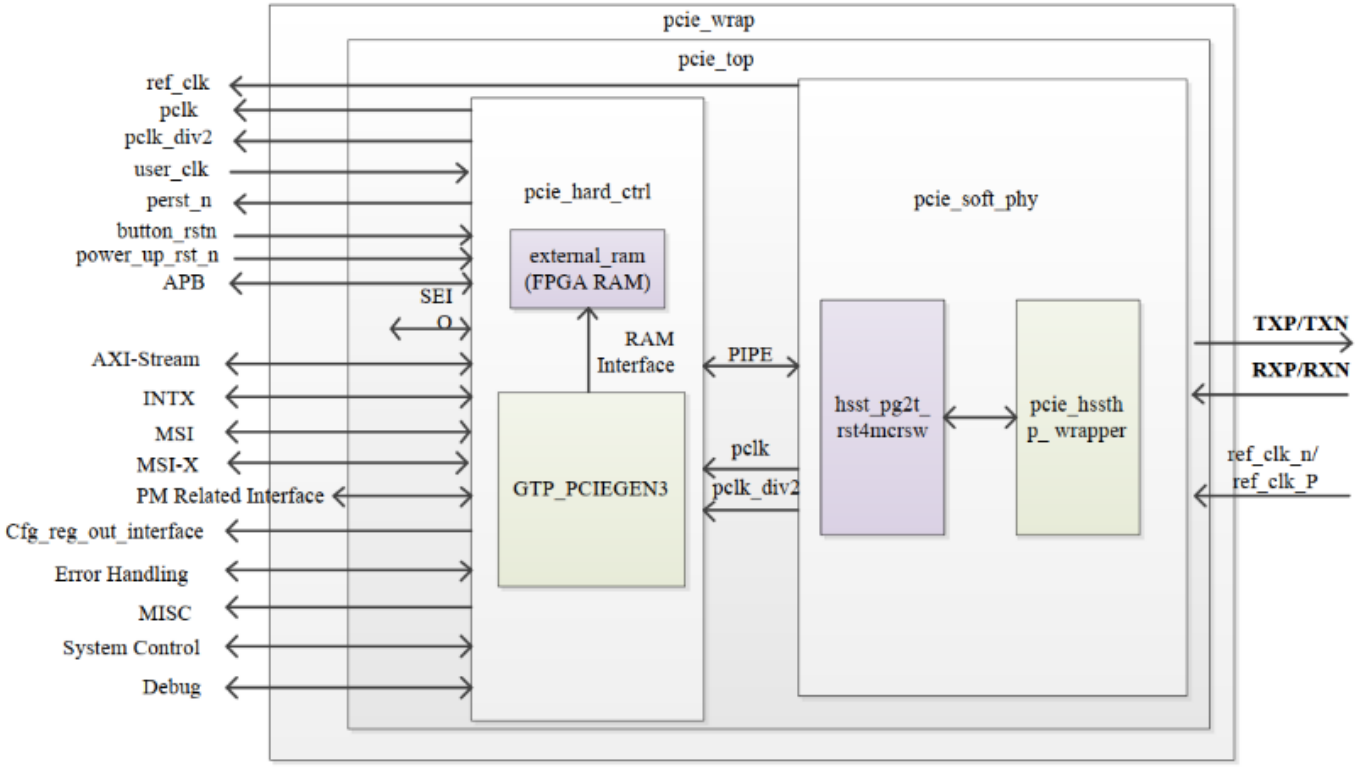
21. PCIE 通信测试实验例程

21.1. 实验简介

实验使用“小眼睛科技”公司的 MES390H 开发板，通过配置 PCIe IP 生成的参考工程（PCIe 回环）以及紫光同创官方提供的 PCIe DMA 验证测试方案，完成对 PCIe 通信的测试。

21.2. 实验原理

MES2T900-390HP 扩展底板上提供一个工业级高速数据传输 PCIex8 接口，PCIE 卡的外形尺寸符合标准 PCIe 卡电气规范要求，可直接在普通 PC 的 x8PCIe 插槽上使用。 PCIe 接口的收发信号直接跟 FPGA 的 HSST 收发器相连接，四通道的 TX 信号和 RX 信号都是以差分信号方式连接到 FPGA，MaxLinkSpeed 可高达 8GT/s。PCIe 的参考时钟由 PC 的 PCIe 插槽提供给开发板，参考时钟频率为 100Mhz。



PCIExpress IP 主要由 pcie_hard_ctrl 和 pcie_soft_phy 两部分组成。

pcie_hard_ctrl 用于实现协议相关的 Transaction Layer、 Data Link Layer 及 Physical Layer (MAC)三层的主要功能。external_ram：用于实现 PCIe 的 data_rcv_ram、 tph_st_ram、 transmit_retry_ram 功能；GTP_PCIEGEN3：用于实现 PCIe 主要功能。

pcie_soft_phy 包含 HSSTHP 及相应的复位序列。hsst_pg2t_rst4mcrsw：HSS

THP 复位序列；pcie_hssthp_wrapper： HSSTHP 顶层。

PCIE IP 符合 PCI Express® Base Specification Revision 3.0 协议和 PHY Interface for the PCI ExpressTM Architecture Version 2.00（数据通路扩展为 32 bits）协议。

功能特性	特性说明
支持配置 Device Type	PCI Express Endpoint
	Legacy PCI Express Endpoint
	Root Port of PCI Express Root Complex
支持配置 Max Link Width	x1
	x2
	x4
	X8
支持配置 Max Link Speed	2.5GT/s
	5GT/s
	8GT/s
支持 Max Link Width 设置为 x1 时可选 LPLL	仅在 Gen1/2 时可选
支持 Max Link Width 设置为 x2 时可配置 Auto-Lane Reversal	-
支持配置两个 Physical Function	-
支持 SRIOV 功能,最大支持 6 个 Virtual Function	-
支持 100MHz Reference Clk	-
支持 Debug 接口	-
支持 PRBS 测试接口	-
支持眼图扫描接口	-
支持通过 Apb 动态配置 PCIe Configuration Space	-
支持 Receive Queue Management	-
支持 Lane Reversal	-
支持 Force No Scrambling	-
支持配置 ID	支持配置 Vendor ID

	支持配置 Device ID
	支持配置 Revision ID
	PCI Express Endpoint、Legacy PCI Express Endpoint 支持配置 Subsystem Vendor ID
	PCI Express Endpoint、Legacy PCI Express Endpoint 支持配置 Subsystem ID
	支持配置 Classcode
支持 BAR 配置	PCI Express Endpoint、Legacy PCI Express Endpoint 支持配置 6 个 BAR
	Root Port of PCI Express Root Complex 仅支持配置 BAR0、BAR1
	PCI Express Endpoint 支持配置为 Memory BAR
	Legacy PCI Express Endpoint、Root Port of PCI Express Root Complex 支持配置为 Memory、IO BAR
	支持 32bit BAR
	32bit BAR 支持配置大小为 256 Byte -2G Byte
	BAR0、BAR2、BAR4 支持 64bit BAR
	64bit BAR 支持 Prefetchable
	64bit BAR 支持配置大小为 256 Byte -8E Byte
	支持 Expansion ROM BAR
	Expansion ROM BAR 支持配置大小为 2K Byte -16M Byte
支持配置 Max Payload Size	128 Byte
	256 Byte
	512 Byte
	1024 Byte
支持配置 Extended Tag Field 与 Extended Tag Default	–
支持 Atomic 事务	–
RC 时支持设置 Read Completion Boundary	–
支持配置 Target Link Speed	–
RC 时支持设置 CRS Software	–

Visibility	
支持设置 ECRC Generation Capable	–
功能特性	特性说明
默认使能 ECRC Check Capable	–
支持 INTX 中断	支持 INTA、INTB、INTC、INTD
支持 MSI 中断	支持 64-bit Address MSI 中断
	支持 Multiple Message Capable：1、2、4、8、16、32 个 Vectors
	支持 Per Vector Masking Capable
支持 MSIx 中断	支持配置 Table Size 、Offset 与 BIR
	支持配置 PBA Offset 与 BIR
支持 SRIOV settings	支持配置 Capability Version
	支持配置 Number of PF VF’s
	支持配置 First VF Offset
	支持配置 VF Device ID
支持配置 6 个 VF BAR	支持 32bit VF BAR
	32bit VF BAR 支持配置大小为 256Byte-2G Byte
	VF BAR0、VF BAR2、VF BAR4 支持 64bit VF BAR
	64bit VF BAR 支持 Prefetchable
	64bit VF BAR 支持配置大小为 256Byte-8E Byte
支持 SRIOV MSI 中断	支持 Multiple Message Capable：1、2、4、8、16、32 个 Vectors

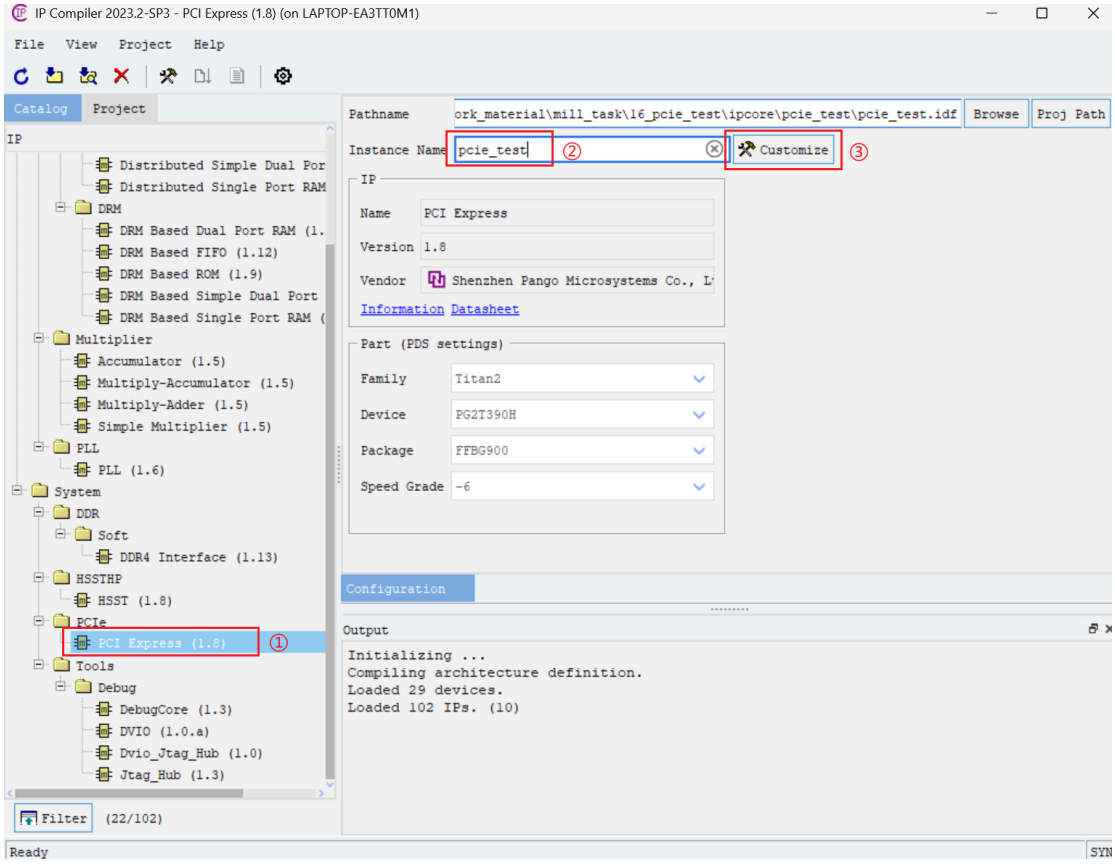
21.3. 工程说明

21.3.1. 安装 PCIe IP 核

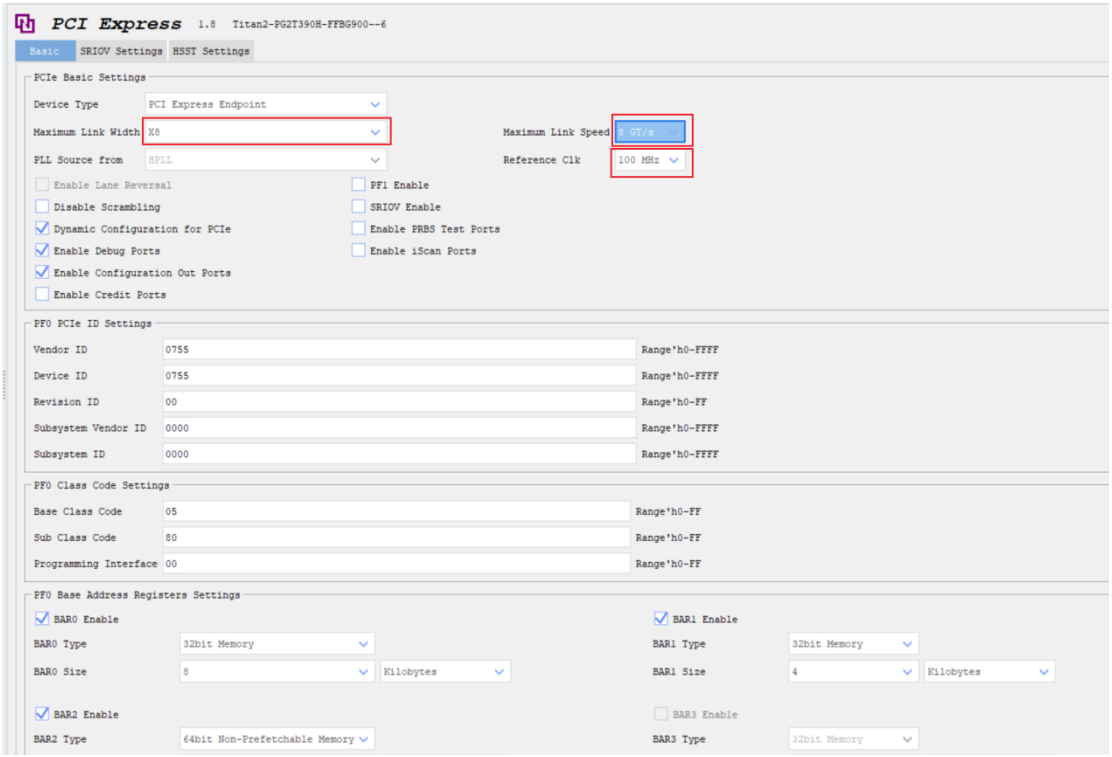
PDS 安装后，需手动添加 PCIE IP，PCIe IP 文件位于 6_IP_setup_packet\PCIE\ ips2t_pcie_v1_8.iar，安装 IP 核的操作步骤如章节 16.3.1 所示。

名称	修改日期	类型	大小
ips2t_pcie_eval.zip	2020/12/31 16:46	WinRAR ZIP 压缩文件	21,997 KB
ips2t_pcie_gen2_v1_0.iar	2024/6/25 13:53	IAR 文件	3,868 KB
ips2t_pcie_v1_0.iar	2021/3/2 8:28	IAR 文件	5,525 KB
ips2t_pcie_v1_1.iar	2021/5/10 4:04	IAR 文件	4,954 KB
ips2t_pcie_v1_1a.iar	2021/5/21 23:52	IAR 文件	4,686 KB
ips2t_pcie_v1_1b.iar	2021/8/23 19:54	IAR 文件	5,087 KB
ips2t_pcie_v1_1c.iar	2021/9/6 20:30	IAR 文件	5,060 KB
ips2t_pcie_v1_3.iar	2022/10/31 8:29	IAR 文件	4,484 KB
ips2t_pcie_v1_6.iar	2023/9/14 13:45	IAR 文件	4,935 KB
ips2t_pcie_v1_7.iar	2024/7/1 19:21	IAR 文件	5,227 KB
ips2t_pcie_v1_8.iar	2024/9/2 15:13	IAR 文件	5,467 KB
UG052002_PCIe_IP.pdf	2024/9/4 16:24	Microsoft Edge PD...	1,668 KB

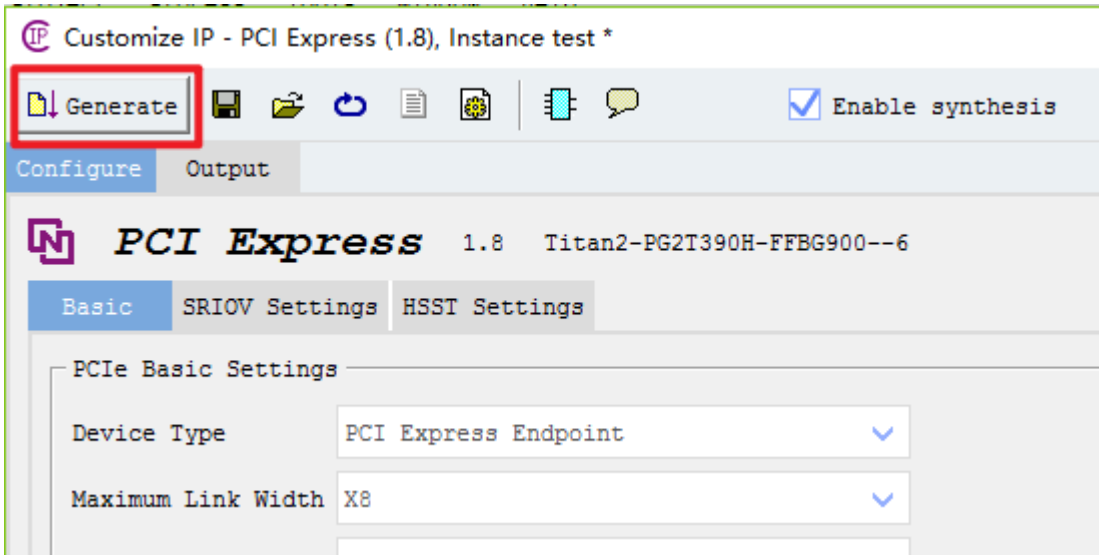
打开 IP Compiler，选取 PCIe IP，取名并点击 Customize。



在 PCIe 配置界面，根据开发板配置 lane 数，可选择 X8，最大线速率可选 8GT/s，参考时钟设置为 100MHz，其他设置保持默认，点击 Generate 生成 PCIe IP。

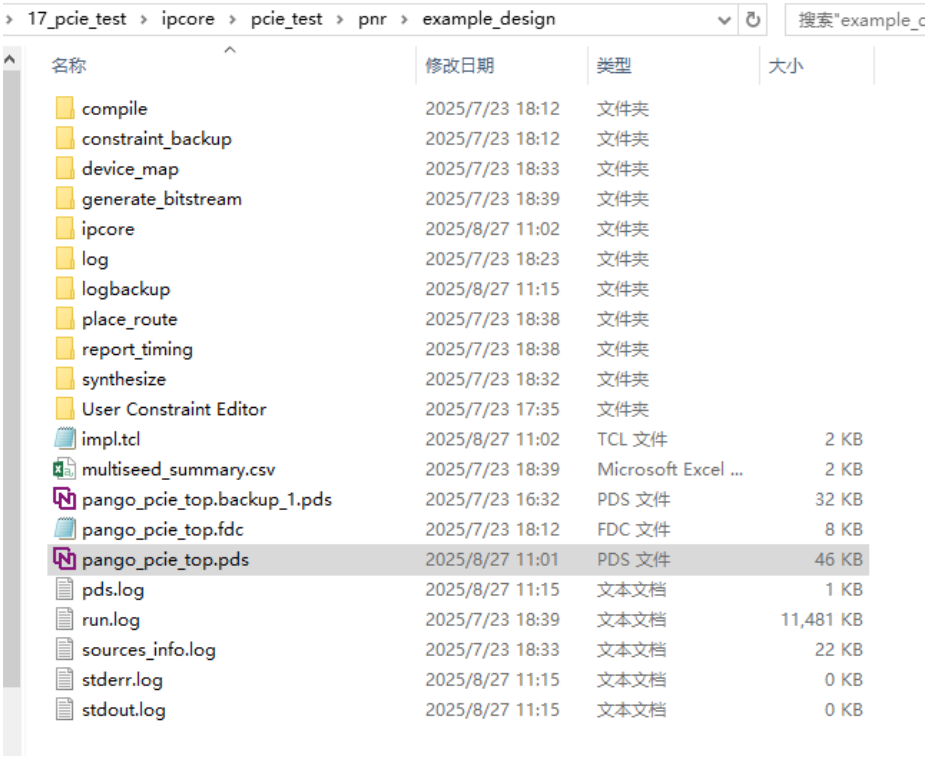


参数配置完成后点击左上角的 <Generate> 按钮，生成 IP。即可生成用户设置的 PCIe IP。



点击生成，后生成 IP 的信息报告界面如下图所示。

关闭本工程，打开 PDS 软件生成的 Example 工程，路径在 “当前工程目录/ipcore/pcie_test/pnr /example”下。（其中 pcie_test 根据 IP 命名而定，并非固定的）

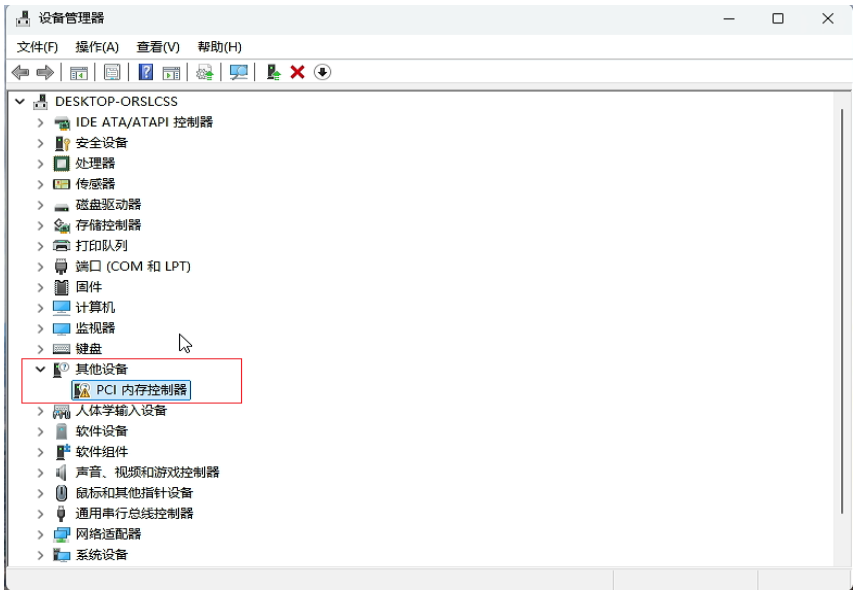


打开 Example 工程后需要对参考工程的管脚约束进行修改，以确保在开发板上正常运行。修改后的管脚约束如下，其中像 txp[0]，txp[1]，rxp[0]，rxp[1]等差分信号都不需要约束。

	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD	DRIVE	BUS_KEEPER	SLEW	OFF
28	rxp[4]	INPUT								
29	rxp[3]	INPUT								
30	rxp[2]	INPUT								
31	rxp[1]	INPUT								
32	rxp[0]	INPUT								
33	pclk_div2_led	OUTPUT	W23	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
34	pclk_led	OUTPUT	V30	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
35	rdlh_link_up	OUTPUT	V29	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
36	ref_led	OUTPUT	V19	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
37	smlh_link_up	OUTPUT	V20	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
38	txd	OUTPUT	L11	BANKL1	3.3	LVCMOS33	12	NONE	FAST	N(d
39	user_led	OUTPUT	W24	BANKL5	3.3	LVCMOS33	12	NONE	FAST	N(d
40	button_rst_n	INPUT	P28	BANKL5	3.3	LVCMOS33		NONE		
41	perst_n	INPUT	L17	BANKL2	3.3	LVCMOS33		NONE		
42	ref_clk_n	INPUT	L7	NOBANK						
43	ref_clk_p	INPUT	L8	NOBANK						
44	rxd	INPUT	K11	BANKL1	3.3	LVCMOS33		NONE		

21.4. 实验现象

测试方法 1：在 Windows 系统下将程序固化到 flash 内，把开发板插入电脑 PCIe 卡槽，开机。打开设备管理器，可识别到 PCIe 设备。表明 PCIe IP 配置正常。



进阶测试方法 2：在 Windows 系统中将实验工程\17_pcie_test 目录下的 pango_pcie_dma_alloc 文件夹拷贝到 Linux 系统下，然后把开发板插入电脑 PCIe 卡槽，在 Linux 系统中按照紫光同创官方提供的 PCIe DMA 验证测试方案的测试文档操作，进行 PCIe DMA 读写测试。

ipcore	2025/7/23 16:30	文件夹	
logbackup	2025/7/23 17:38	文件夹	
pango_pcie_dma_alloc	2025/7/24 11:43	文件夹	
impl.tcl	2025/7/23 17:38	TCL 文件	1 KB
pcie_test.pds	2025/7/23 17:58	PDS 文件	11 KB
pcie_test.pds.lock	2025/7/23 17:38	LOCK 文件	1 KB
pds.log	2025/7/23 17:58	文本文档	3 KB
sources_info.log	2025/7/23 17:58	文本文档	1 KB

Linux 系统中测试的实验现象如下图所示。观察到写数据和读数据完全一致，错误计数为 0，表明系统正常运行。