小眼睛科技

提供一站式 FPGA&嵌入式解决方案

盘古 100K MINI 开发板

FPGA 通用实验指导手册

小眼睛科技 盘古 676 系列

盘古 100K MINI (MES2L676-100HP-MINI-MINI-MINI)开发板实验教程

紫光同创 Logos2 系列 PG2L100H 开发平台

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2025/10/20	初始版本

公司名称: 深圳市小眼睛科技有限公司

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

官方淘宝店铺: 小眼睛半导体

B 站: 小眼睛半导体 (视频教程免费学)

* 加入 FPGA 开发者技术交流与 5000+FPGA 开发者实时沟通

QQ2 群: 442106123 QQ3 群: 882634519)

*配套资料下载、技术答疑请登录逻辑矩阵技术论坛

LOGIC MATRIX 逻辑矩阵

逻辑矩阵技术论坛欢迎各位发烧友加入

让我们共建开源生态, 持续赋能行业发展

https://www.szlogicmatrix.com/

*扫码注册开源技术论坛

*扫一扫关注官微

* 淘宝官方旗舰店

目录

1. FPGA 开发工具使用

1.1. 实验简介

实验目的:

了解 PDS 软件的使用, 在线 Debugger 工具的使用请看 uart 实验章节的工程说明部分章节。

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

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, 一般地, 例如PDS_2022.1版本可将软件安装在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;

双击安装包中的安装程序 Setup.exe, 启动安装程序:

图 1.2-1

启动安装程序后的界面如下:

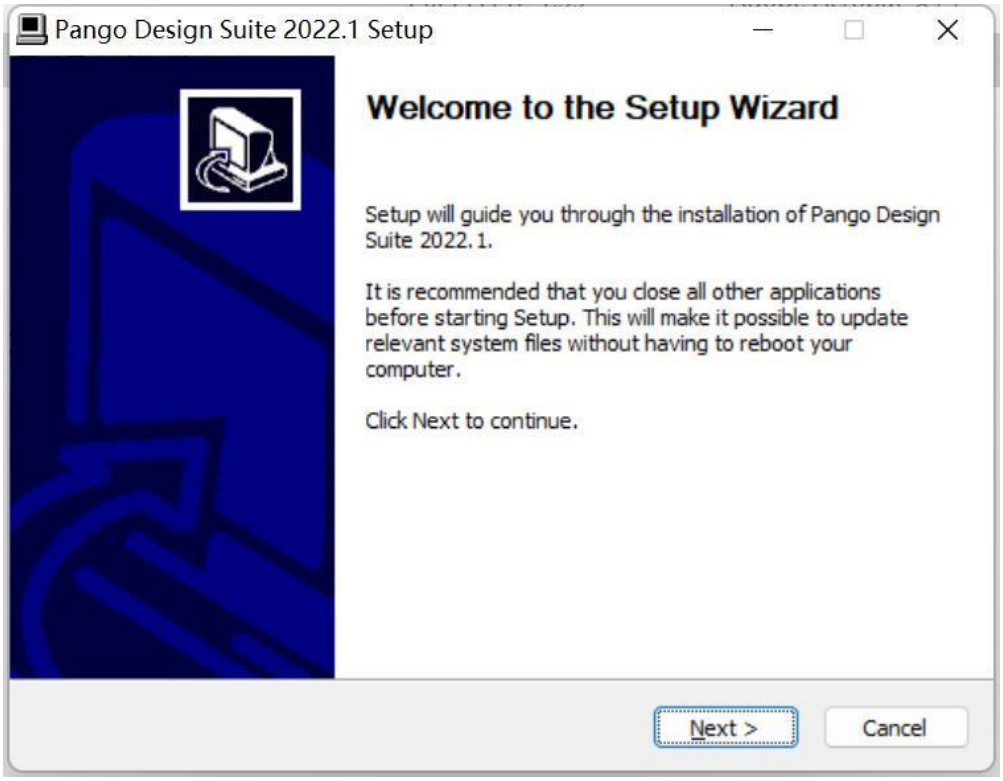


图 1.2-2

点击“Next”, 跳转至许可协议对话框:

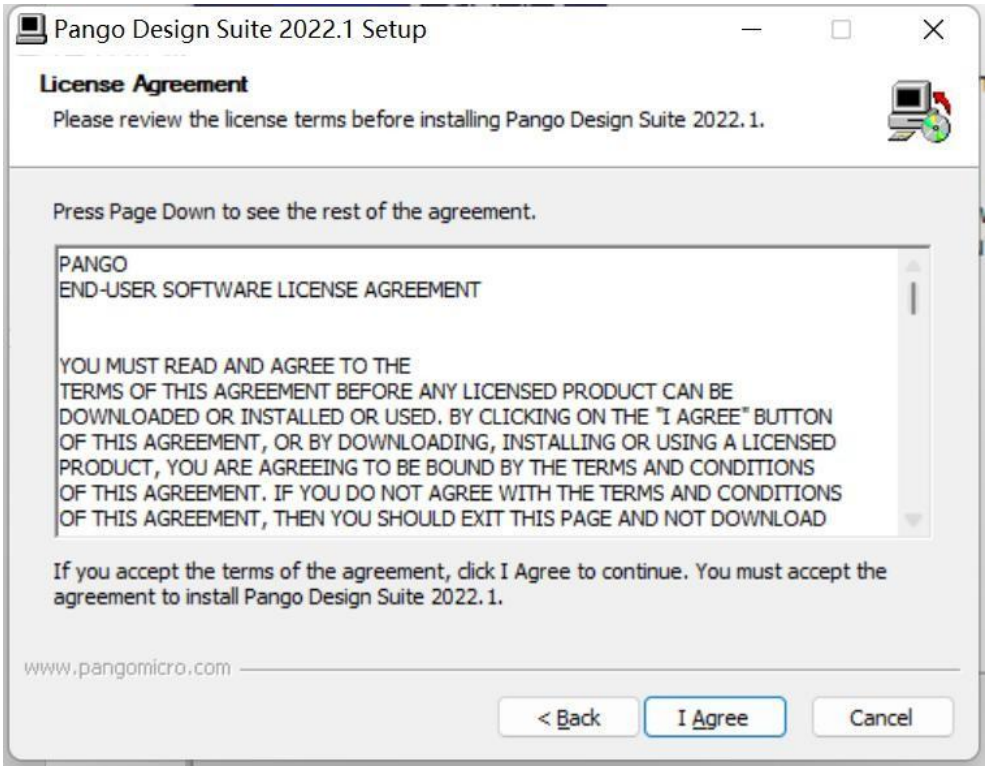


图 1.2-3

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。

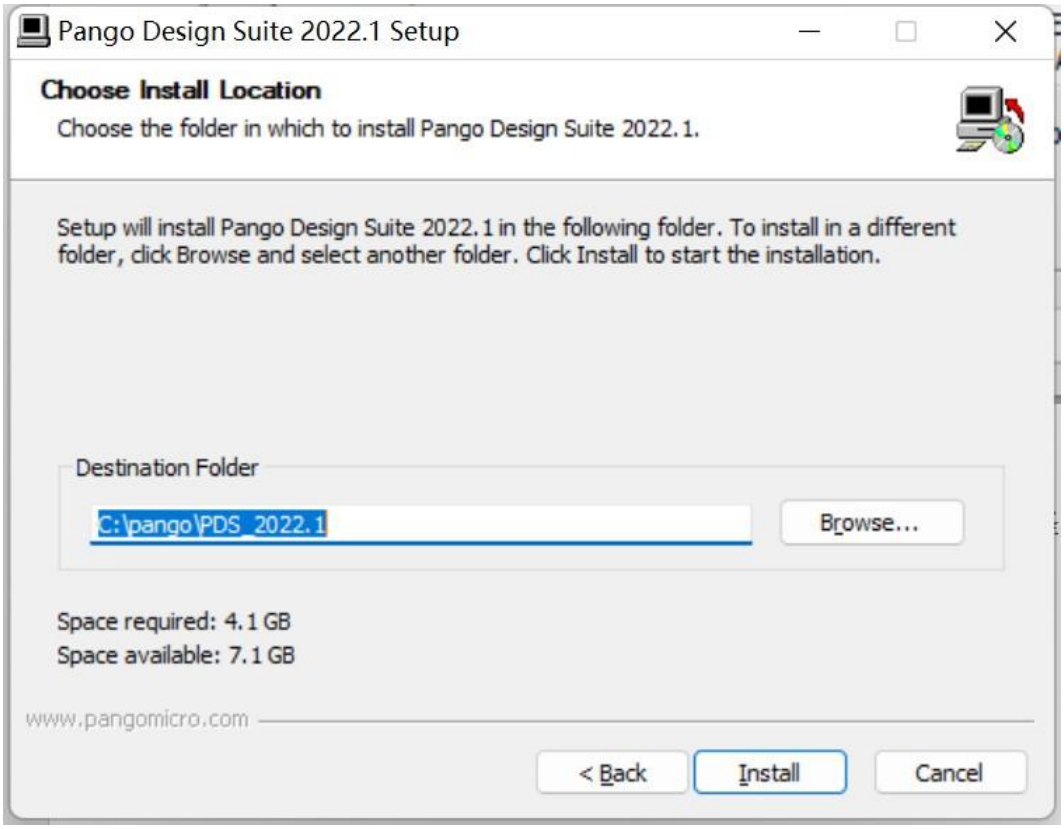


图 1.2-4

直接点击“Install”，则跳转到安装界面。

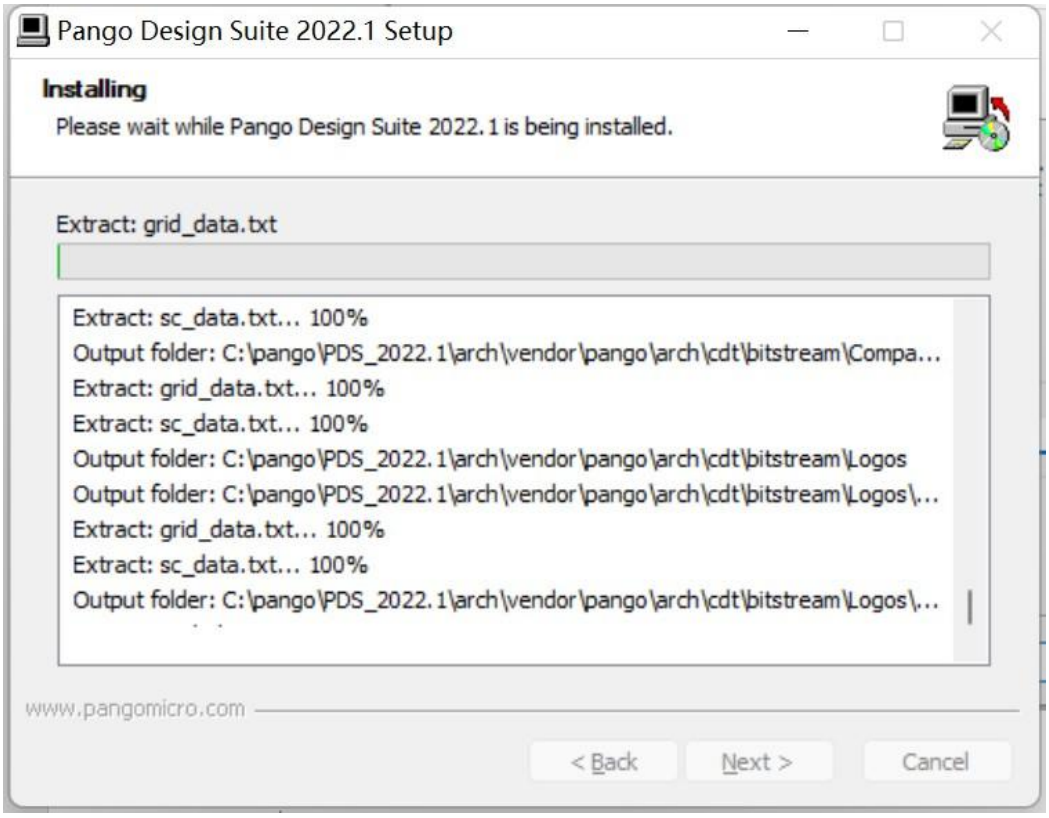


图 1.2-5

耐心等待至完成全部安装过程, 点击“Finish”,安装完毕。

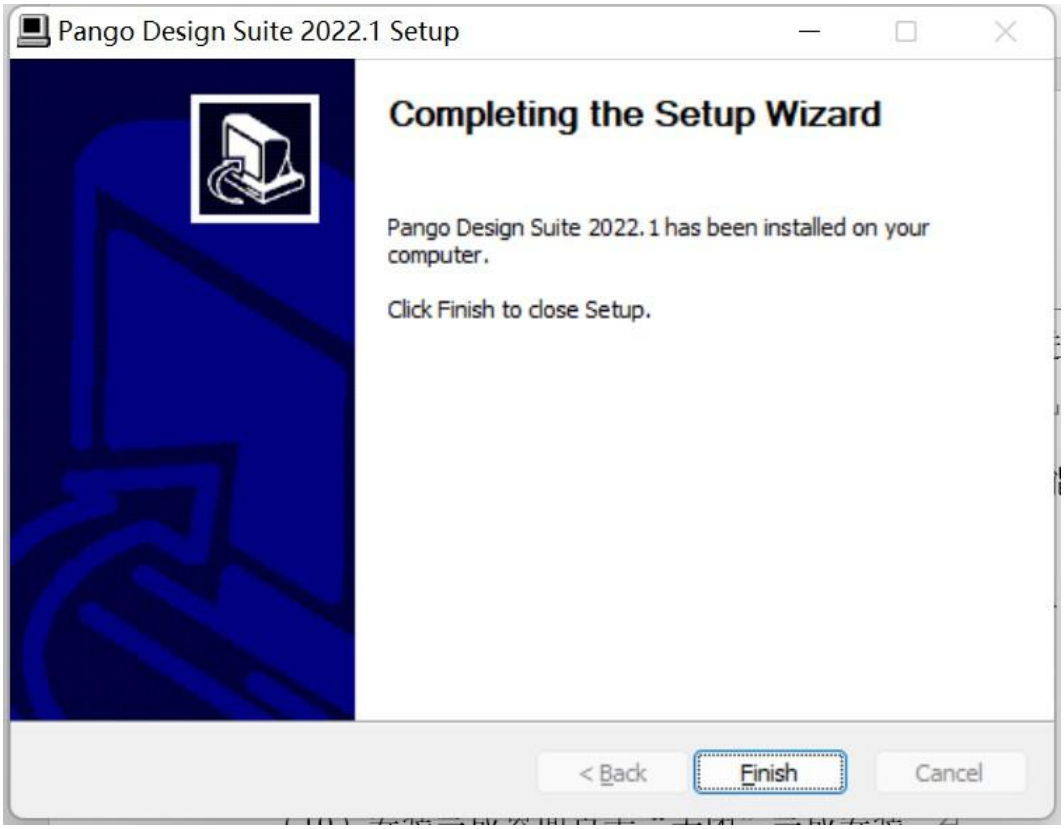


图 1.2-6

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。

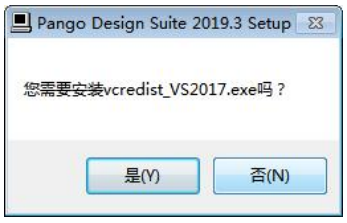


图 1.2-7

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。

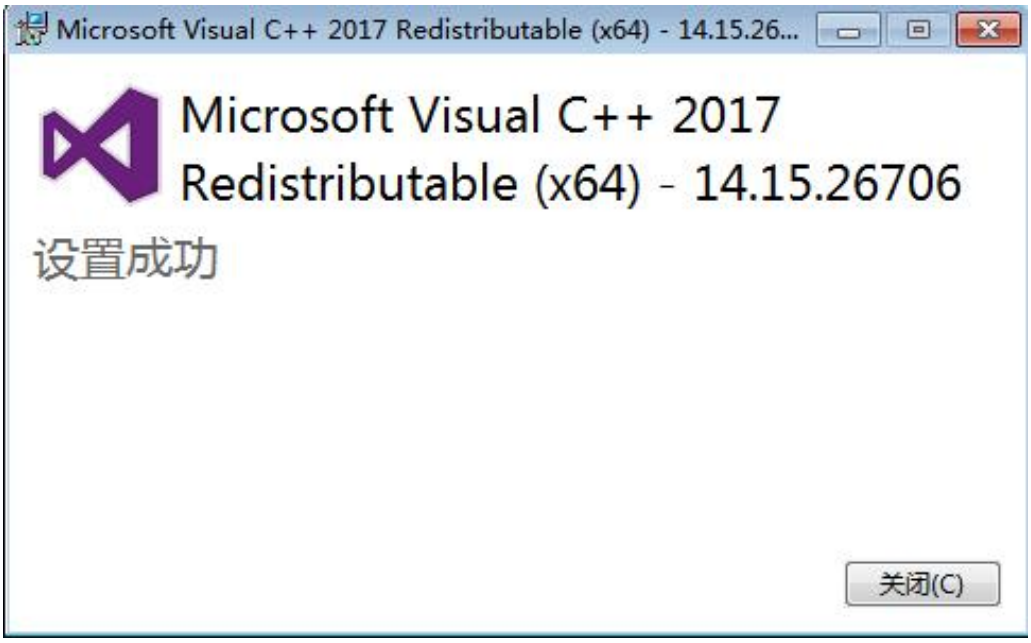


图 1.2-9

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。

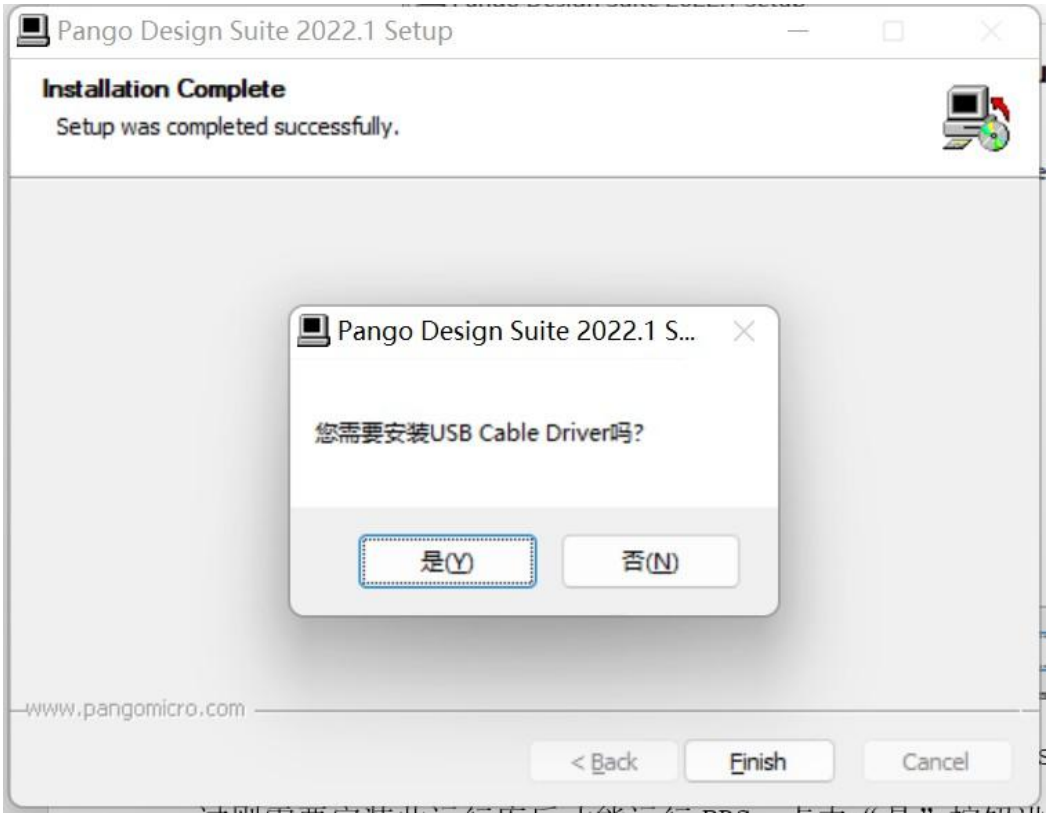


图 1.2-10

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”。

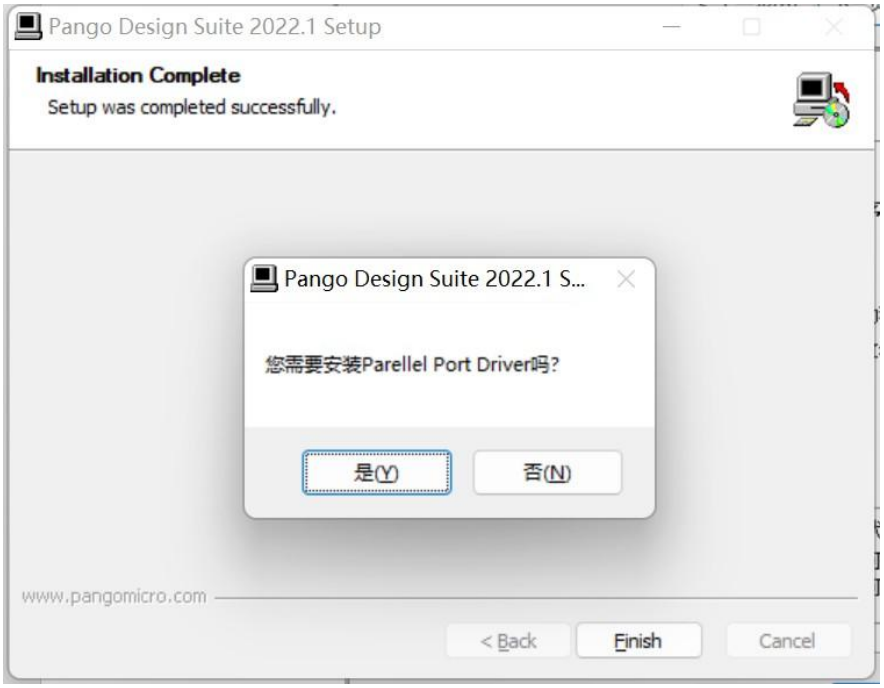


图 1.2-14

点击“是”进行并口驱动程序安装。

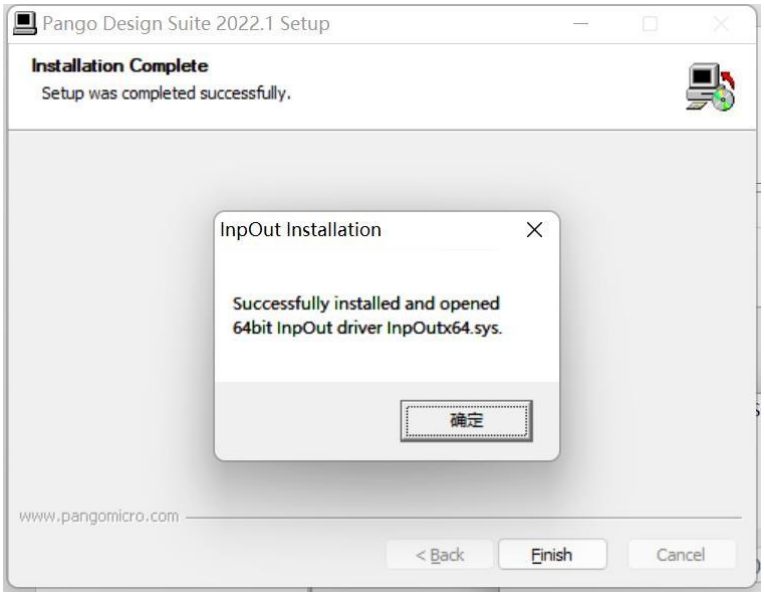


图 1.2-15

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

若PDS license 文件路径为: D:\pango\license\pds_node-locked.lic直接设置环境变量:

变量名: PANGO_LICENSE_FILE

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

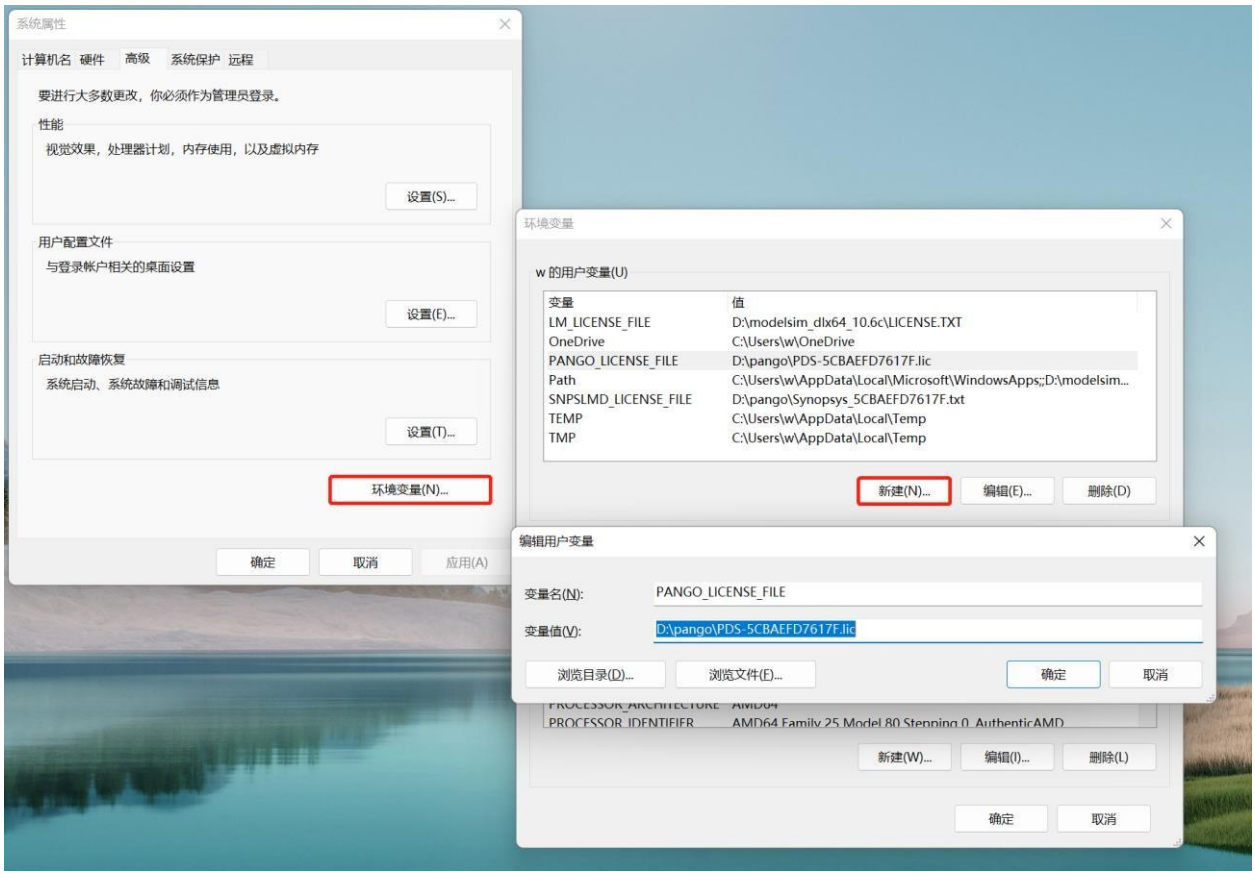


图 1.2-19

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:

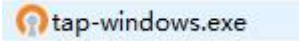


图 1.2-20

双击运行, 全部保持默认。

图 1.2-21

点击 Next。

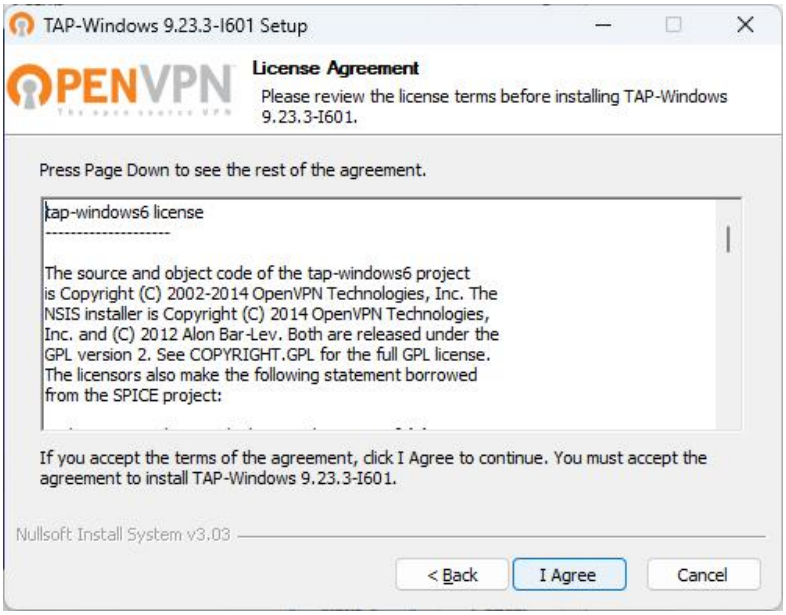


图 1.2-22

点击 I Agree。

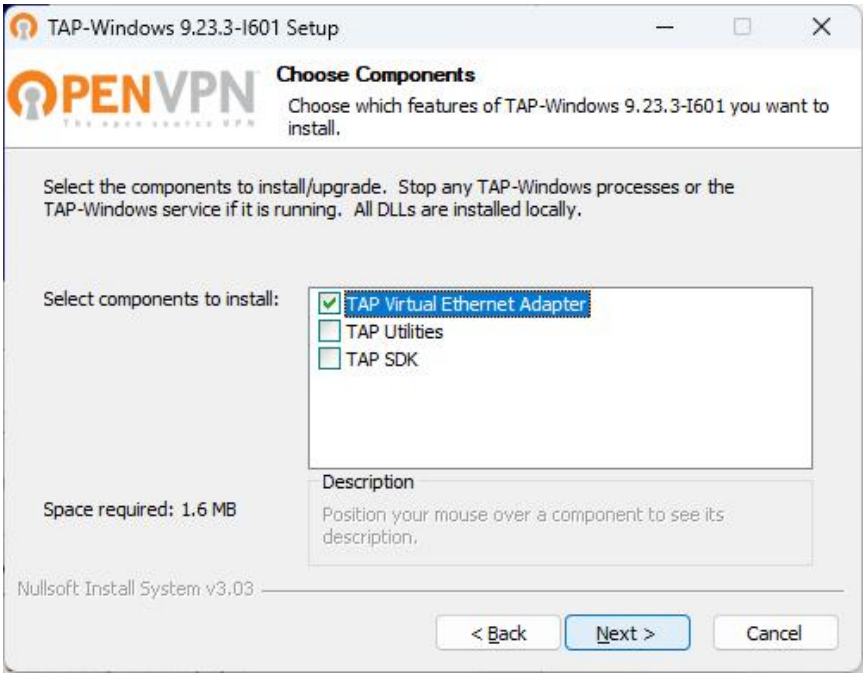


图 1.2-23

保持默认, 点击 Next。

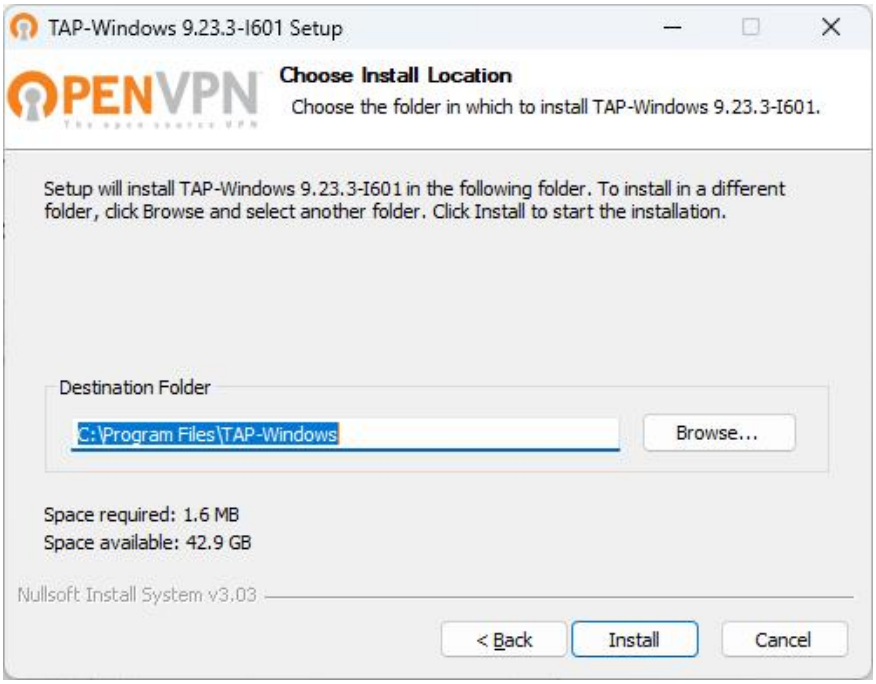


图 1.2-24

安装路径保持默认, 点击 Install, 等待安装完成。

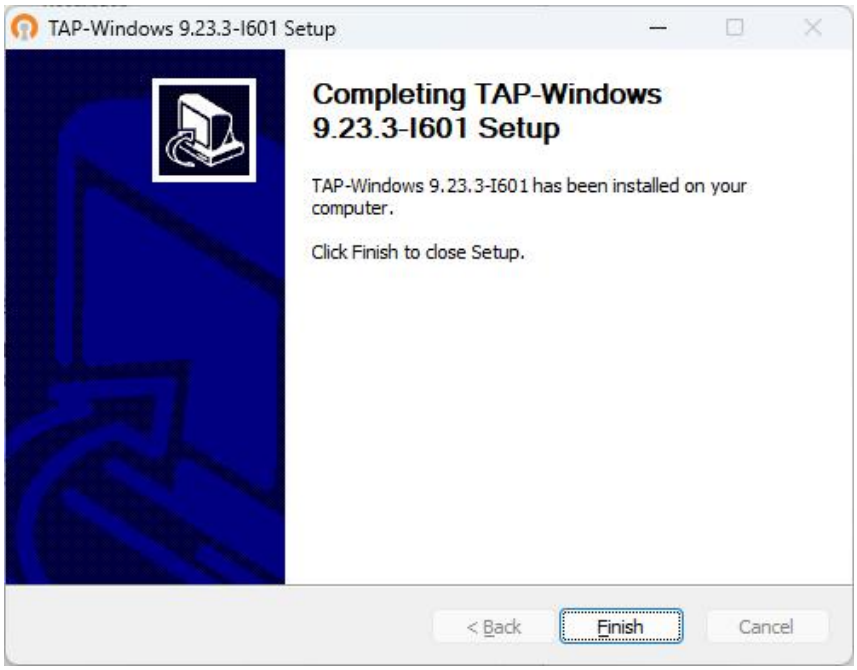


图 1.2-25

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

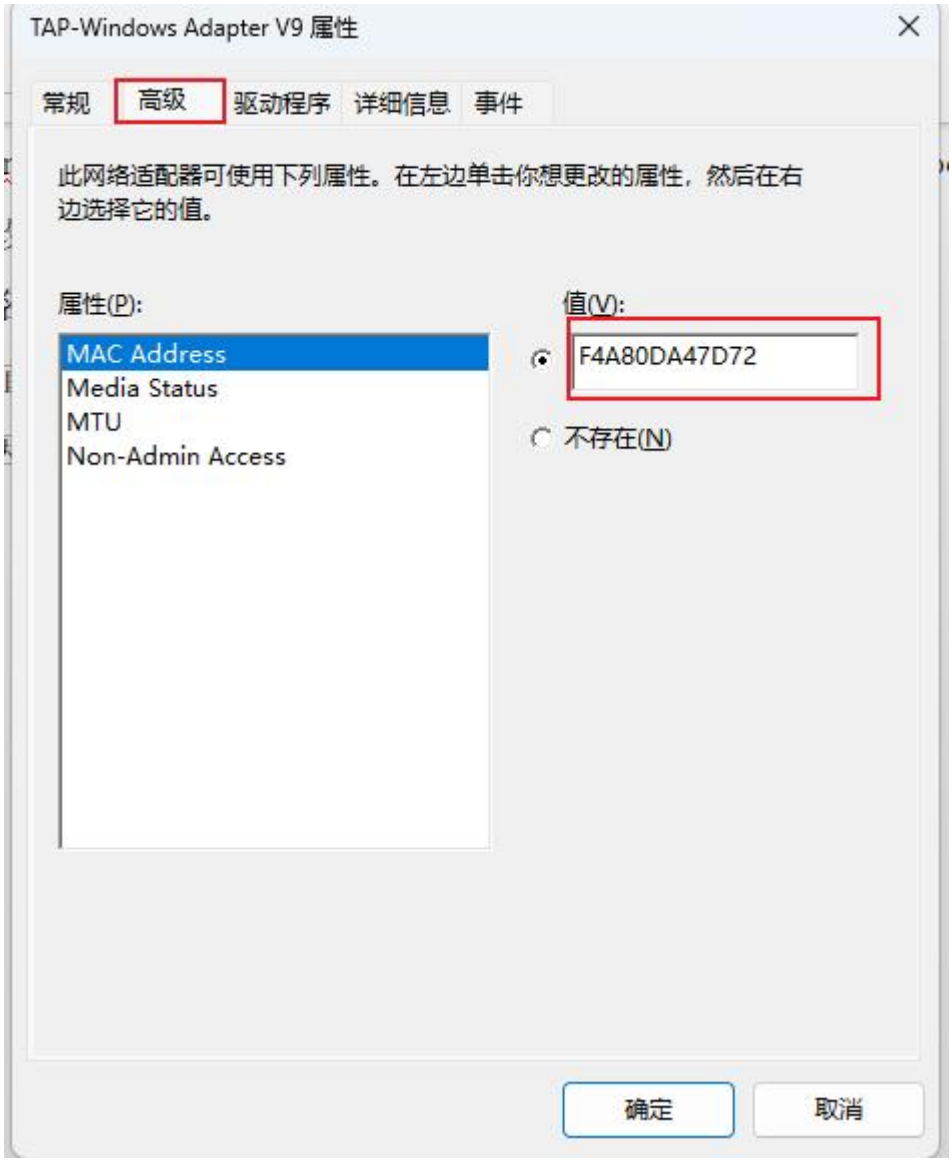


图 1.2-27

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

PDS2022.2-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 20ns。

接下来给出一个参考的 testbench, 如下所示:

```
`timescale 1ns / 1ns
`define UD #1
module tb_led_test();

reg       clk    ;
reg       rst_n  ;
wire[7:0]  led    ;
reg [7:0]  data   ;

initial begin
   rst_n <= 0;
   clk  <= 0;
   #20;
   rst_n <= 1;
   #2000
   $display("I am stop"); //
   $stop;
end
always#10 clk = ~clk;//20ns  50MHZ

led_test
#(
   .CNT_MAX   (10)
)u_led_test(
 .clk      (clk   ),// input
 .rstn     (rst_n ),// input
 .led      (led   ) // output [7:0]
);

initial begin
   $monitor("led:%b", led);
end

always@(posedge clk or negedge rst_n)   begin
   if(!rst_n)
```

```
        data   <=   8'd0;
    else
    begin
        data   <=   {$random}%256;
        $display("Now data is %d",data);
    end
end

endmodule
```

代码第一行定义了仿真的时间单位和精度, 均为 1ns; 第二行宏定义了延时的时间 UD, 可根据个人代码习惯使用。第 10-18 行对复位和时钟进行初始化, 并且延时 2020ns 后用$display 打印出暂停仿真的字样, 之后停止仿真。注意$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。第 19 行每 10ns 翻转一次, 完成了 50MHZ 时钟的生成。21-28 行例化了流水灯模块, 用来做仿真测试, 该代码主要完成每隔一段时间对输出的数据完成一次移位。进而实现了流水灯。30-32 行用$monitor 监测 led 变量, 一旦改变就打印出来。第 34-42 行主要是产生测试数据并打印出来, 该数据仅仅用来观察 random 函数的测试。

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:

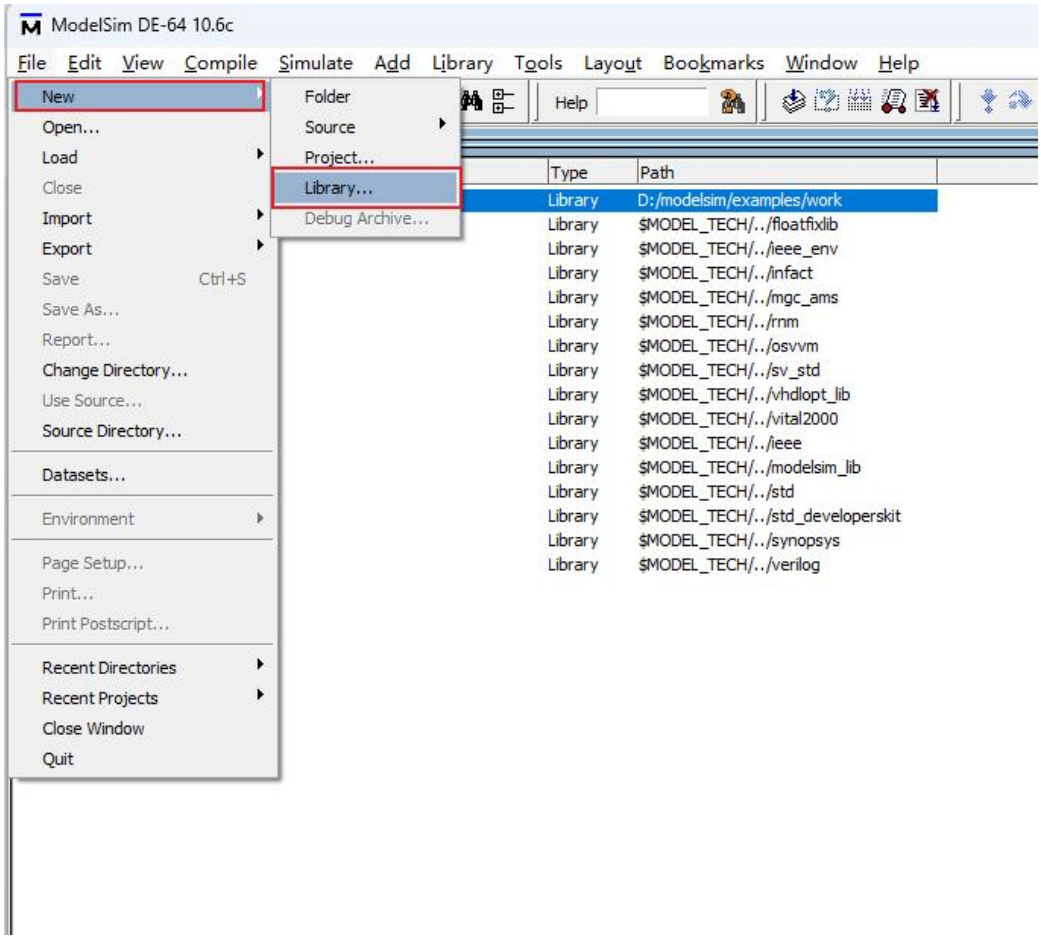


图 2.5-1

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。

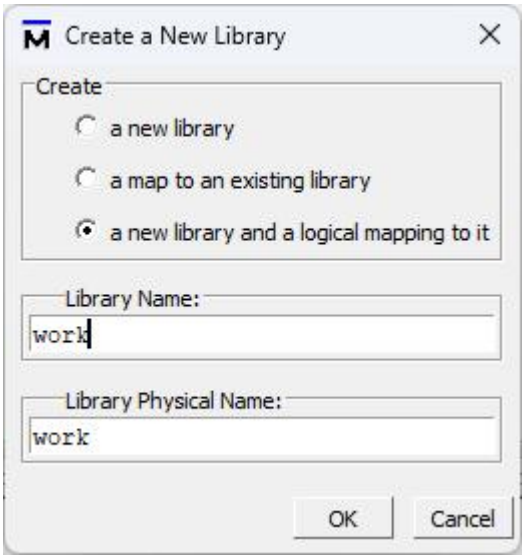


图 2.5-2

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。

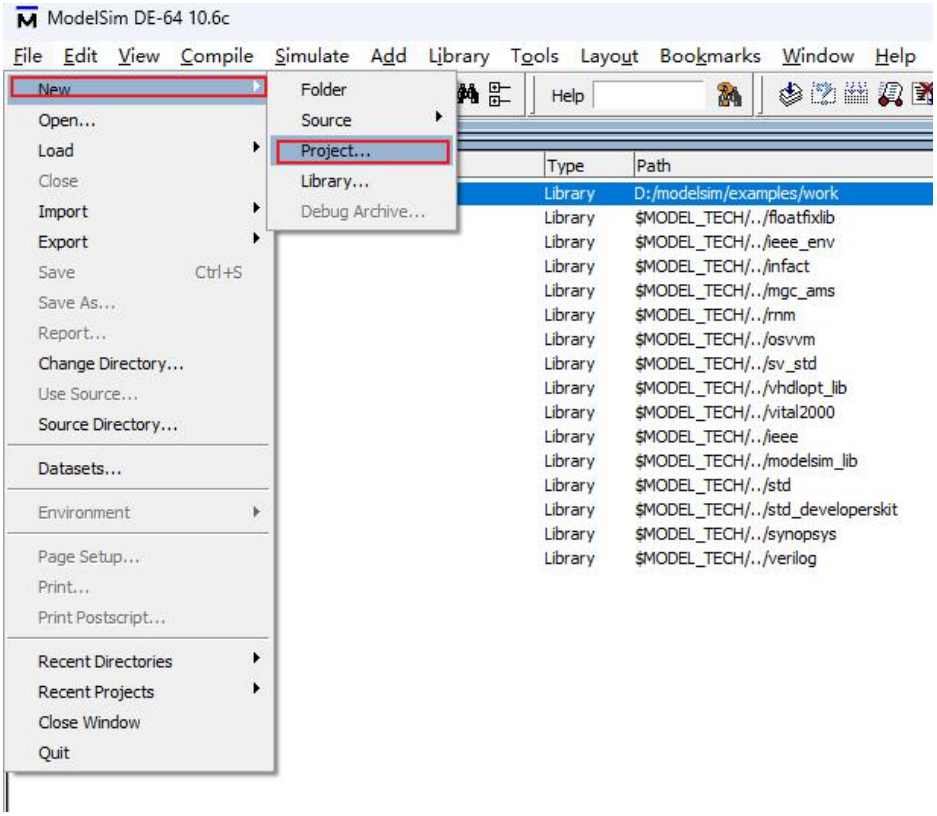


图 2.5-3

左上角 File->New->Project, 新建工程。

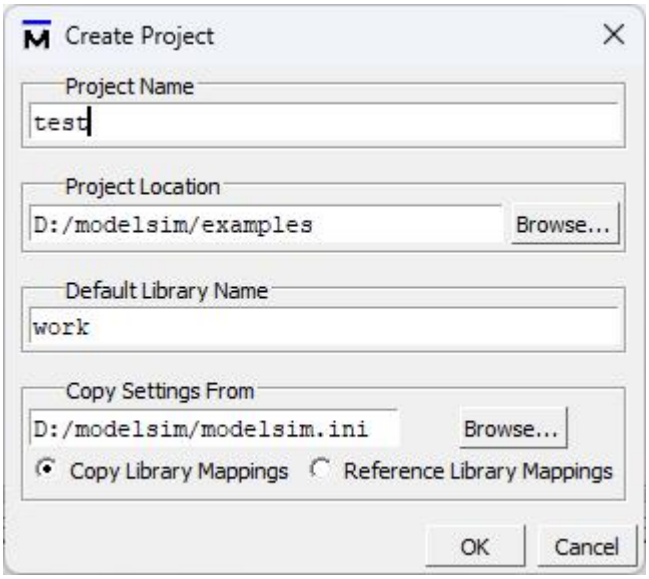


图 2.5-4

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。

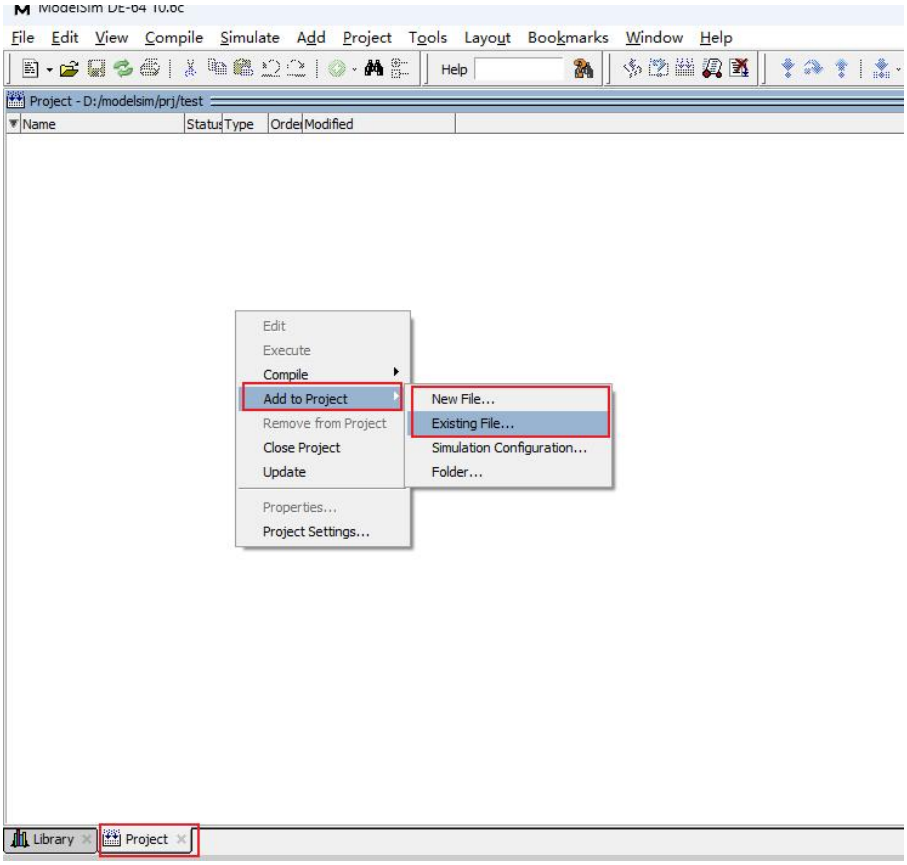


图 2.5-5

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。

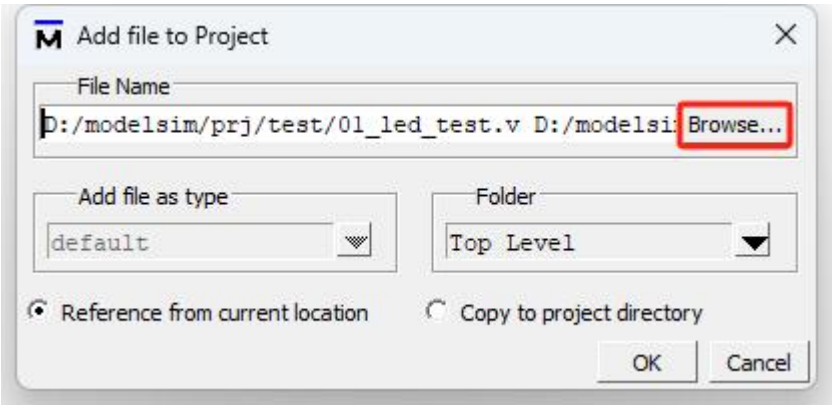


图 2.5-6

点击 Browse, 添加要仿真的文件。

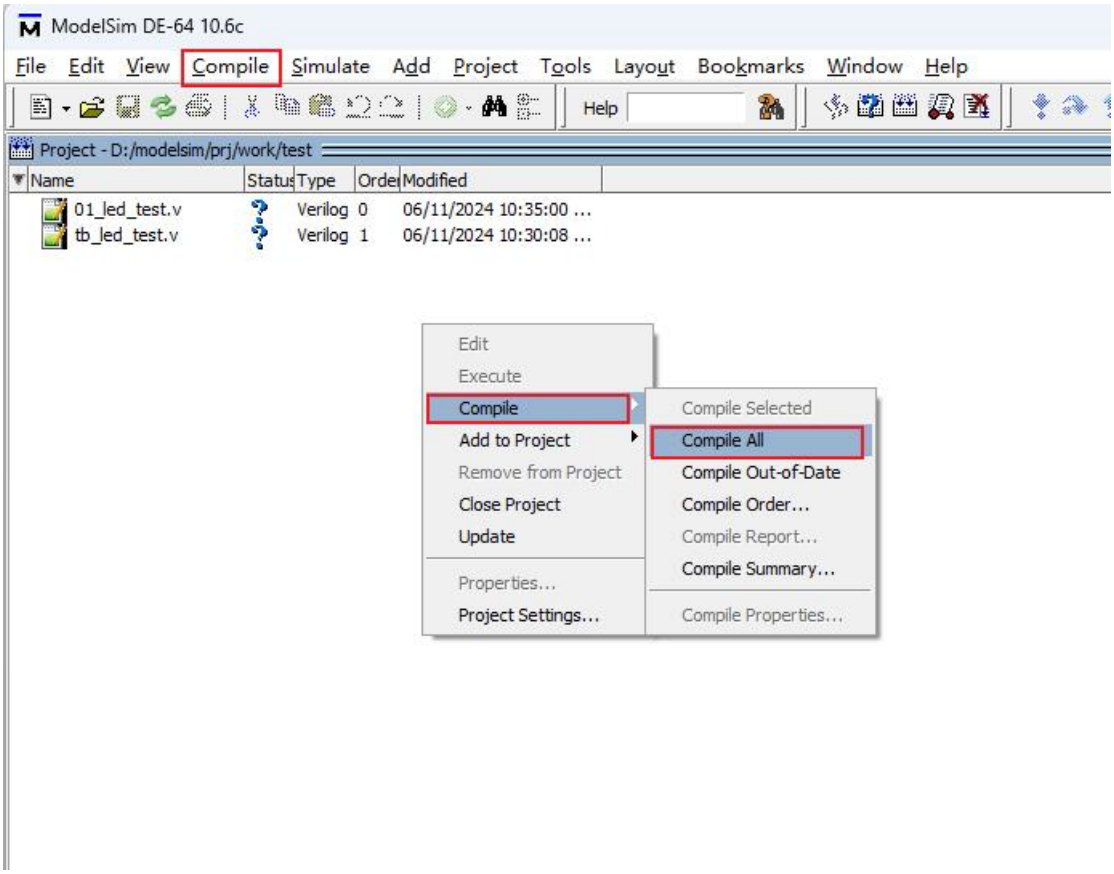
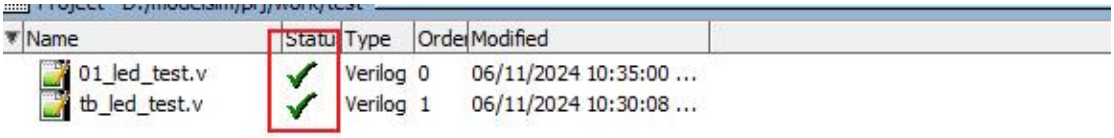


图 2.5-7

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。



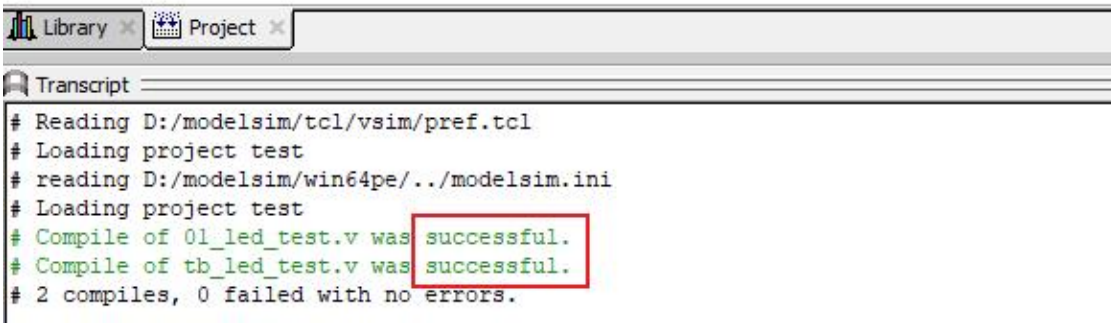


图 2.5-8

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。

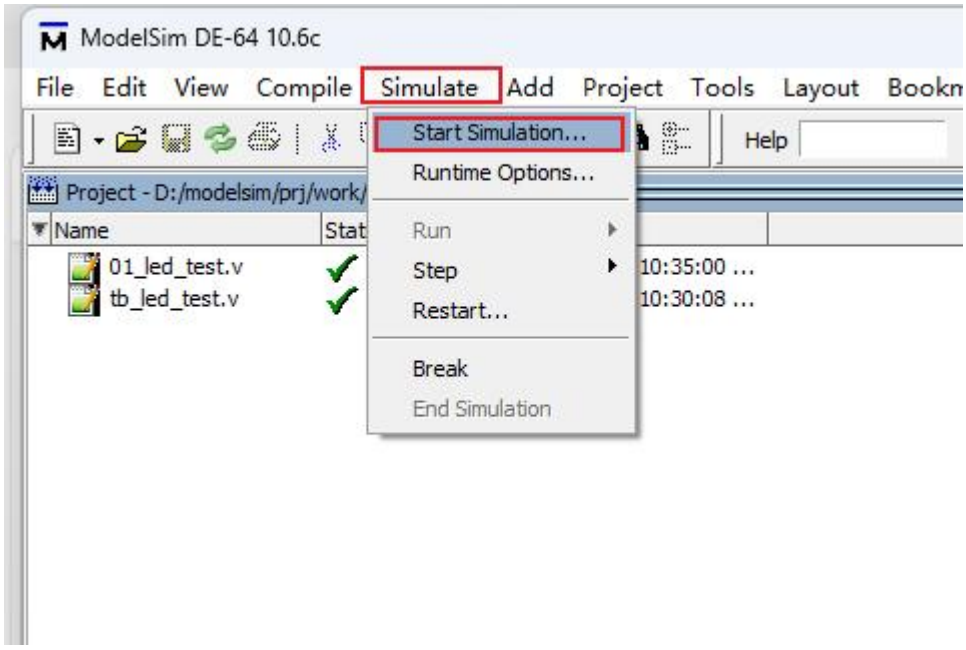


图 2.5-9

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。

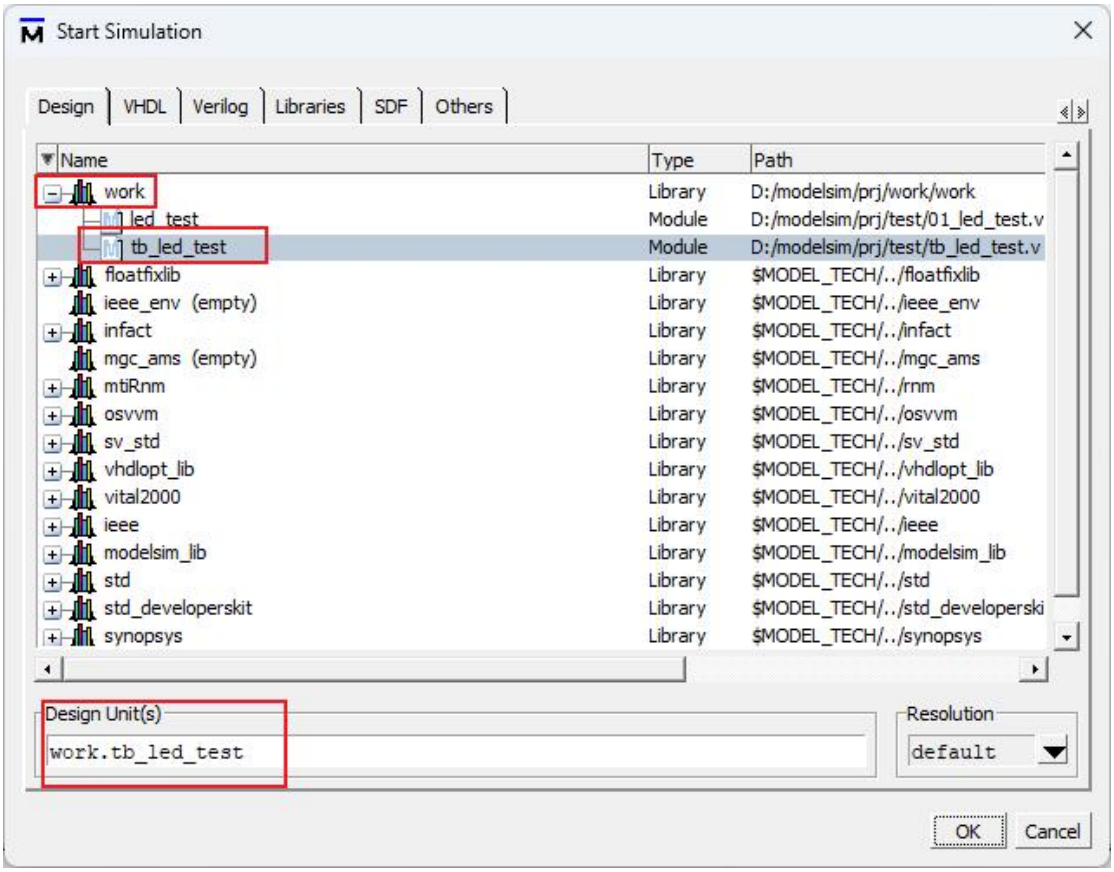


图 2.5-10

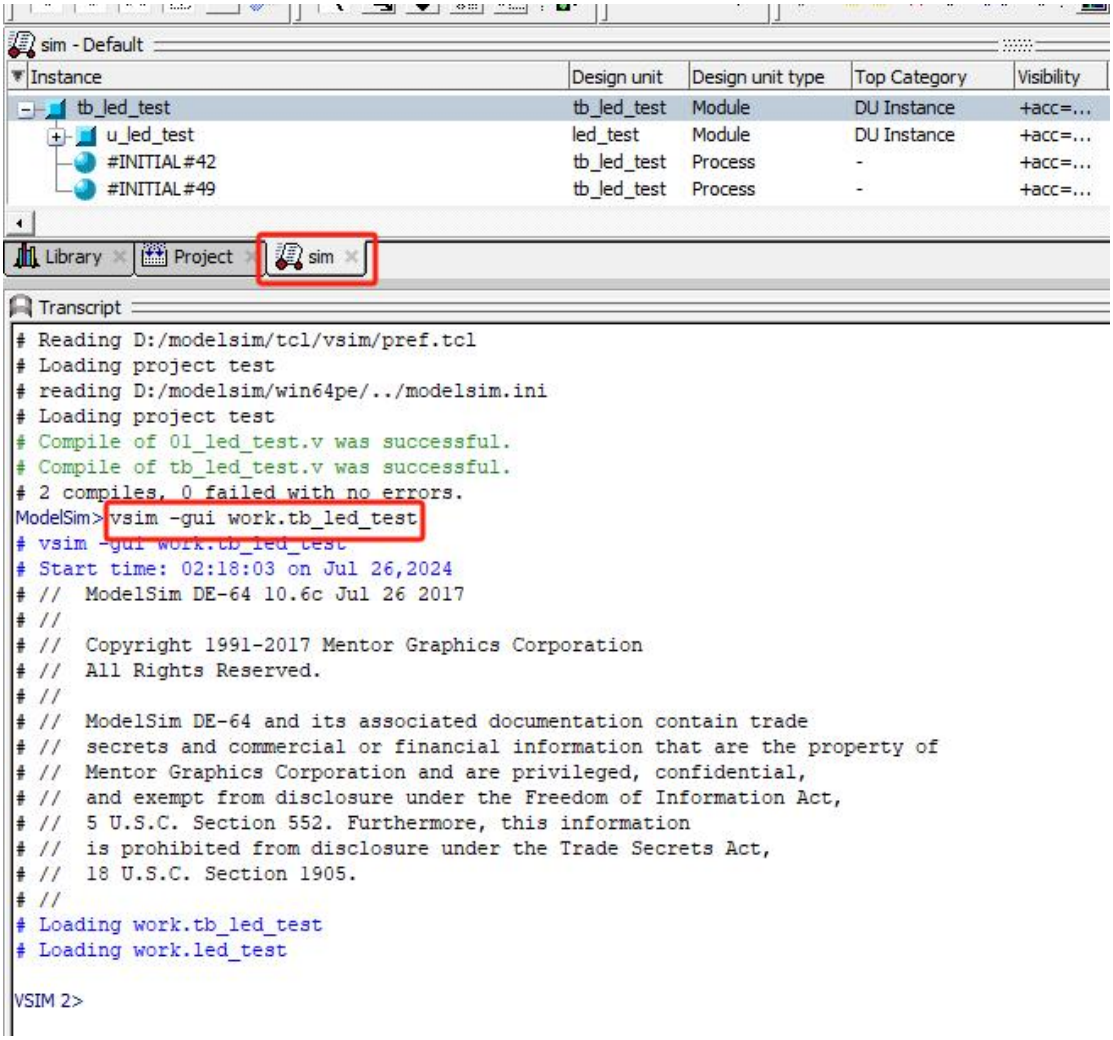


图 2.5-11

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。

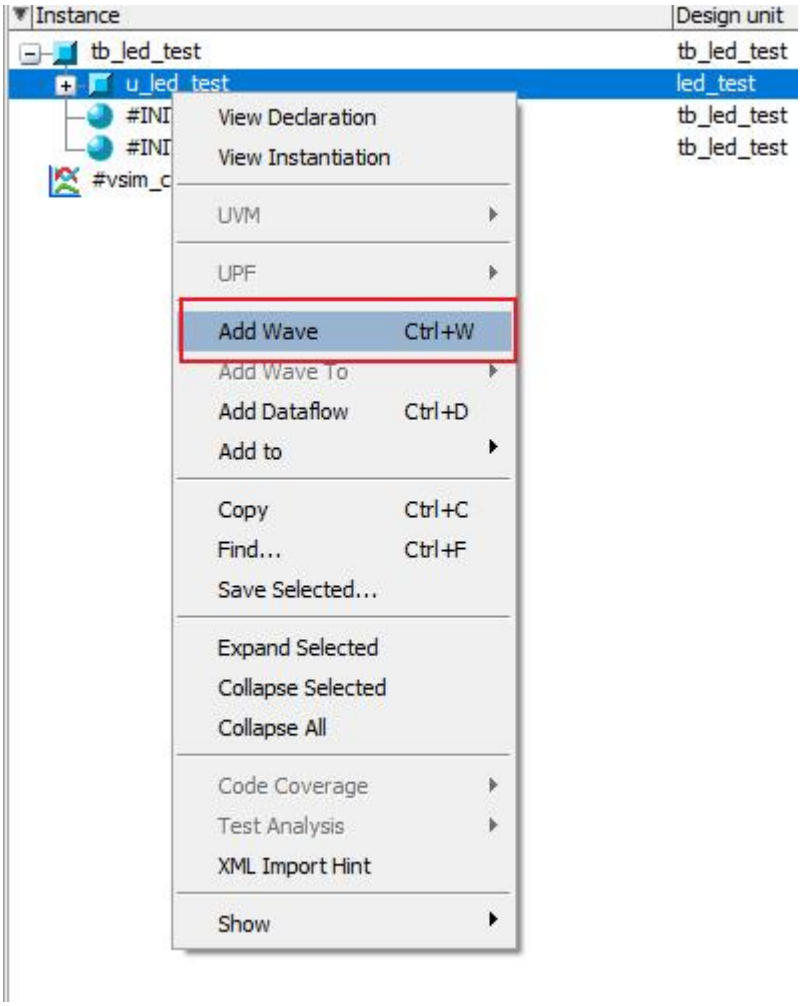


图 2.5-12

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。

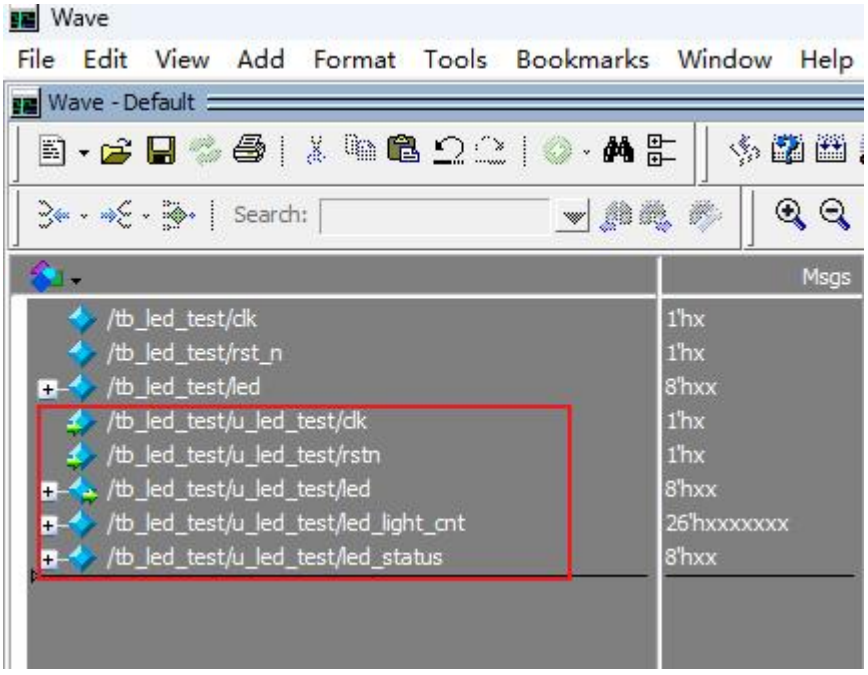


图 2.5-13

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

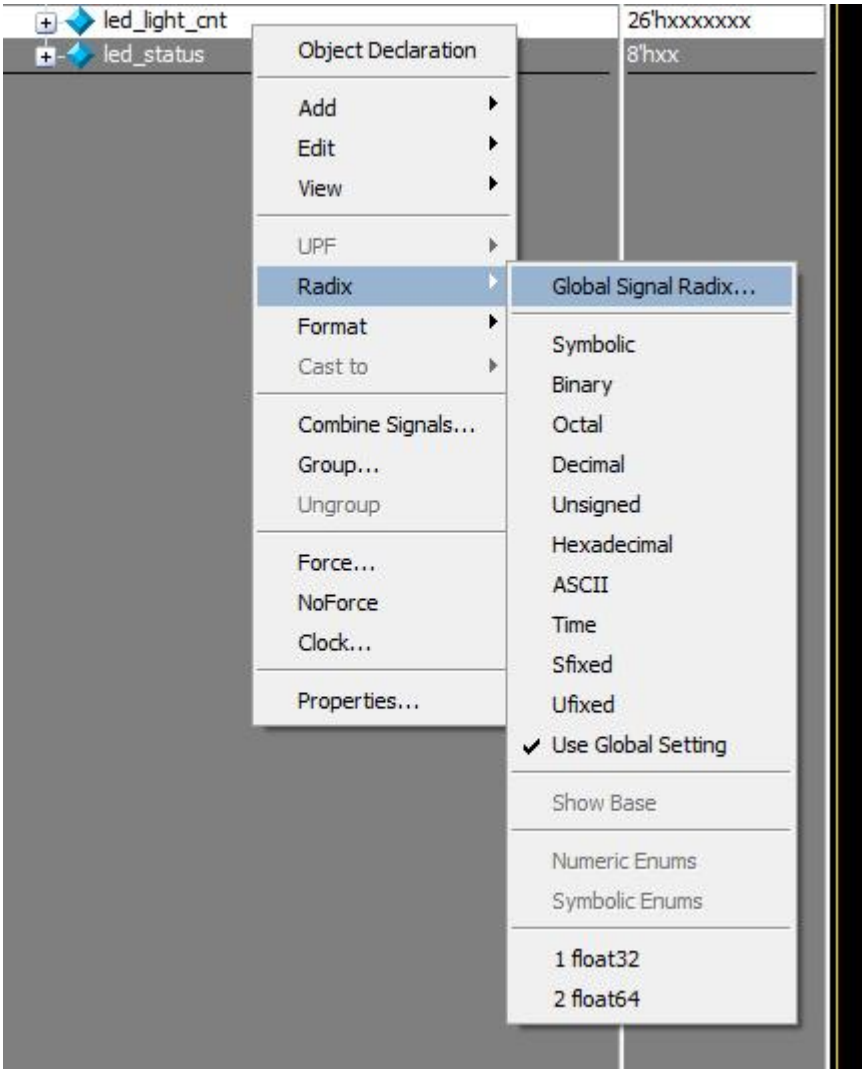


图 2.5-15

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。

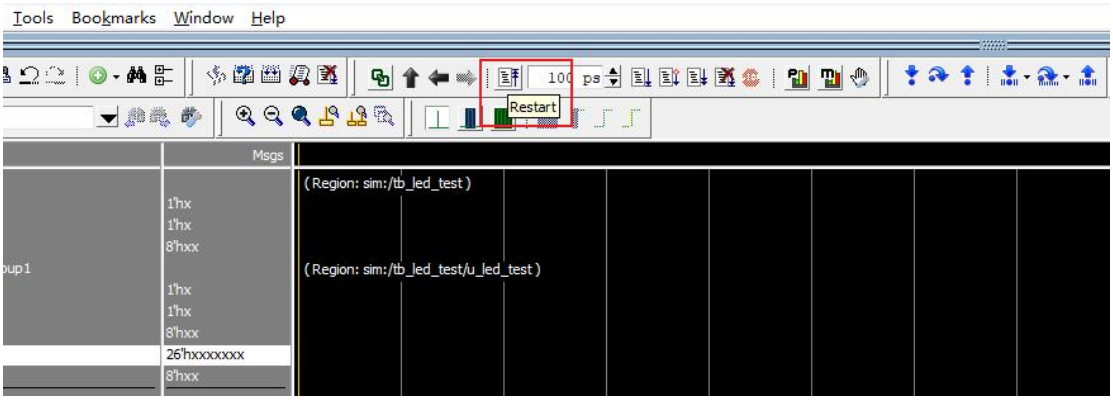


图 2.5-16

点击上方这个地方, 对信号全部进行 Restart 复位。

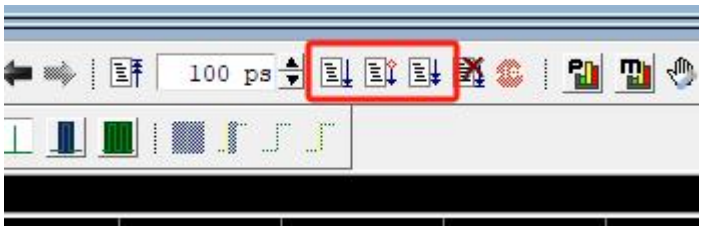


图 2.5-17

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:

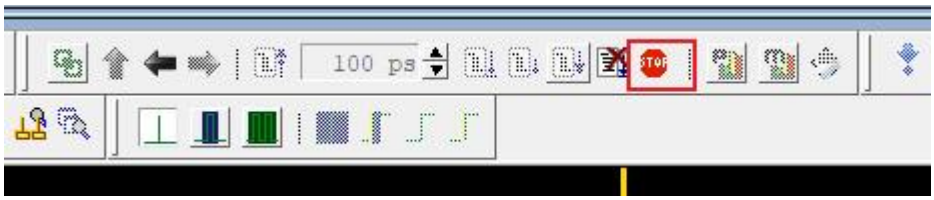


图 2.5-18

按下后, 当用户点击 stop, 仿真才会停止。

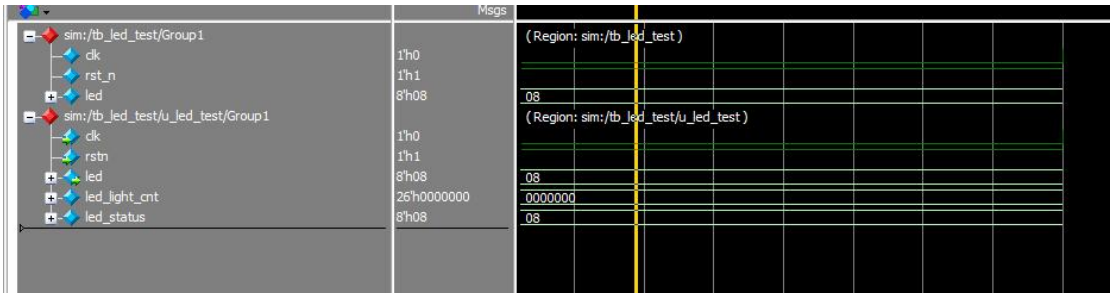


图 2.5-19

图 2.5-19 为操作后显示出来的波形。

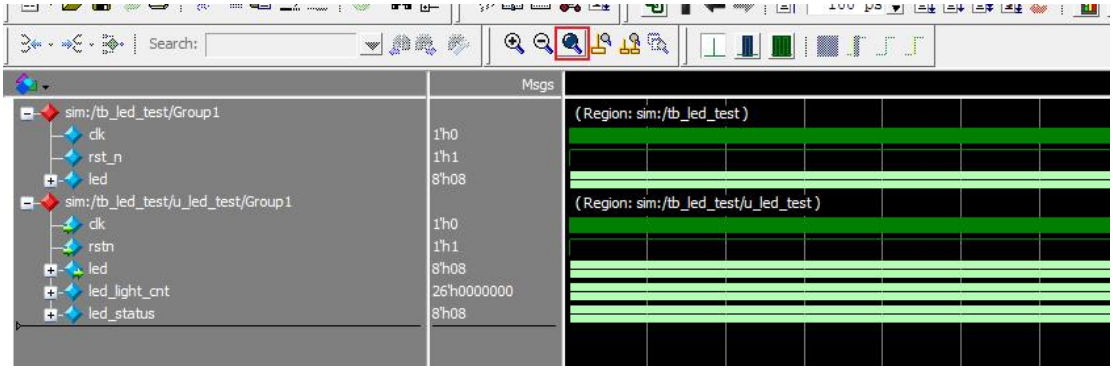


图 2.5-20

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

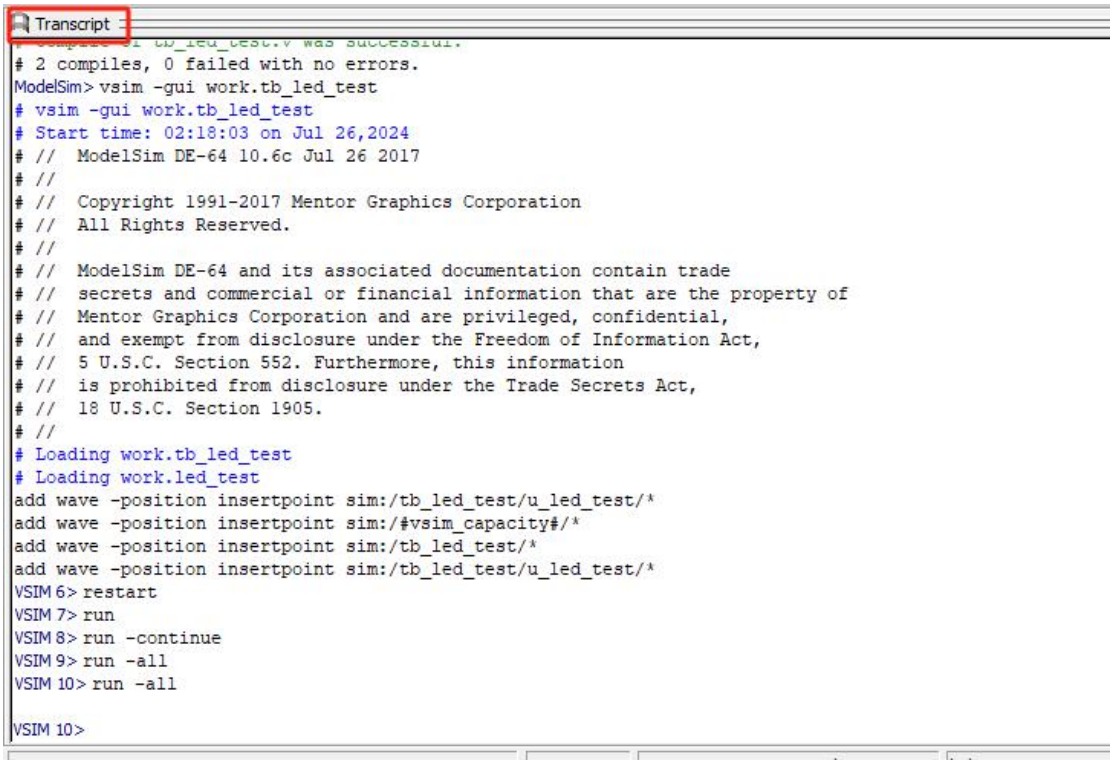


图 2.5-21

可以看当我们添加波形时,Modelsim 自动执行了一句 add wave -position xxxxxxx 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。

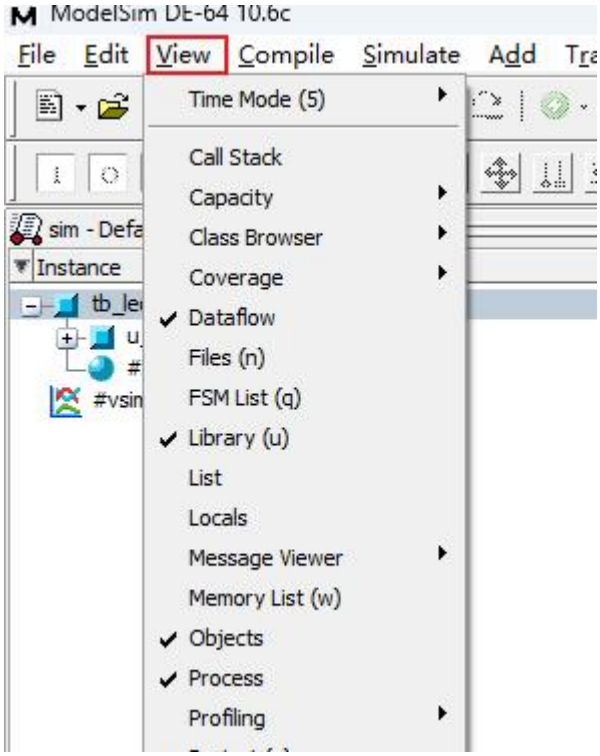


图 2.5-22

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:

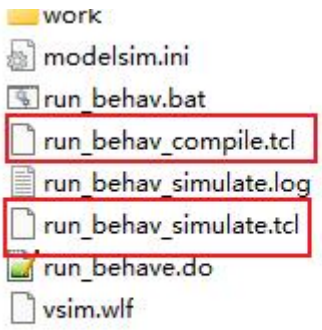


图 2-1

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。

```
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 *
view wave
view structure
view signals
run 1000ns
```

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/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"

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。

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:

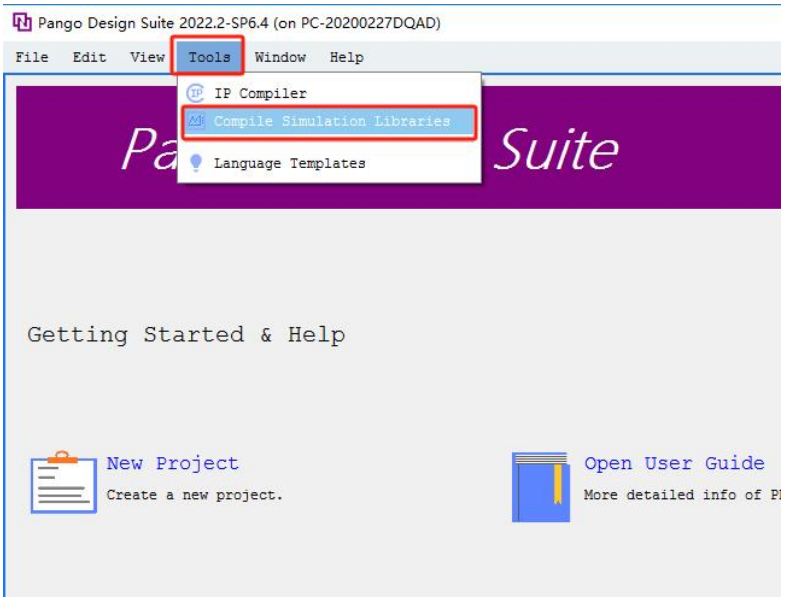


图 3.2-1

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;

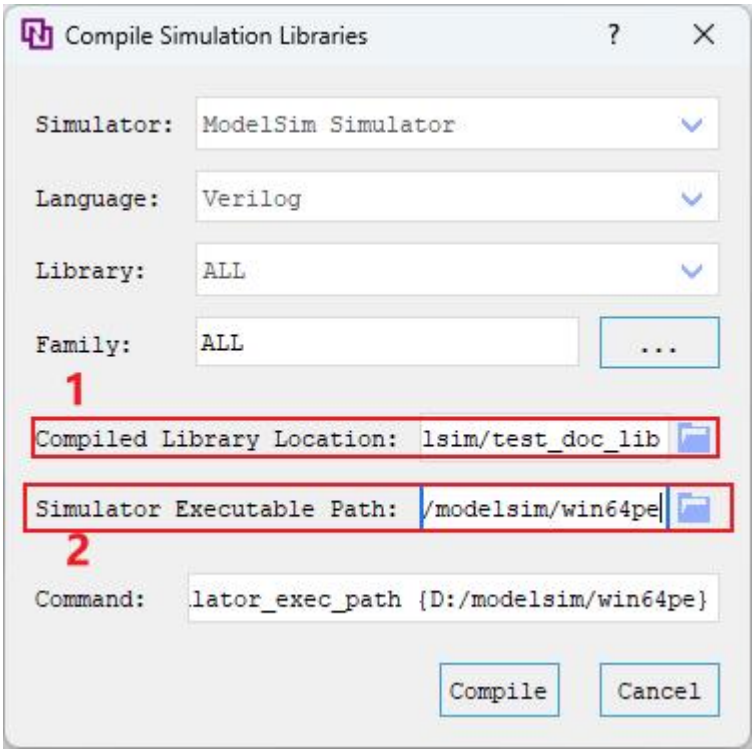


图 3.2-2

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。

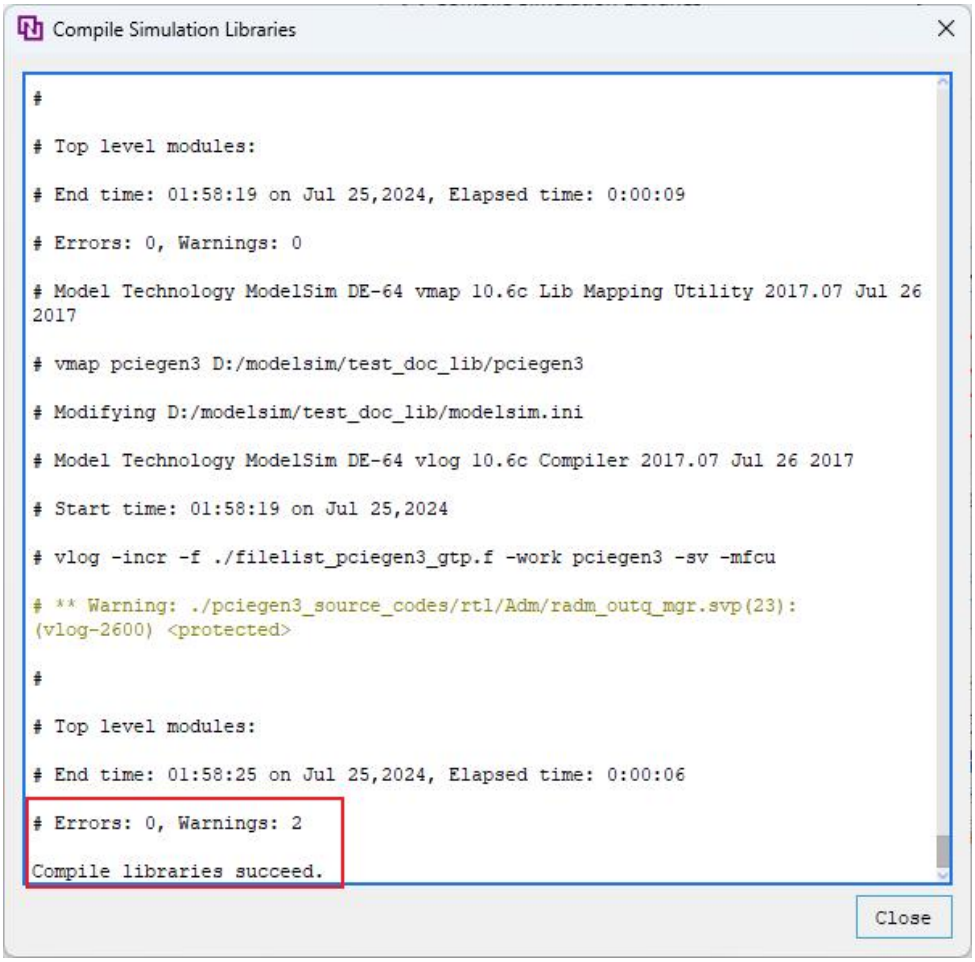


图 3.2-3

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。

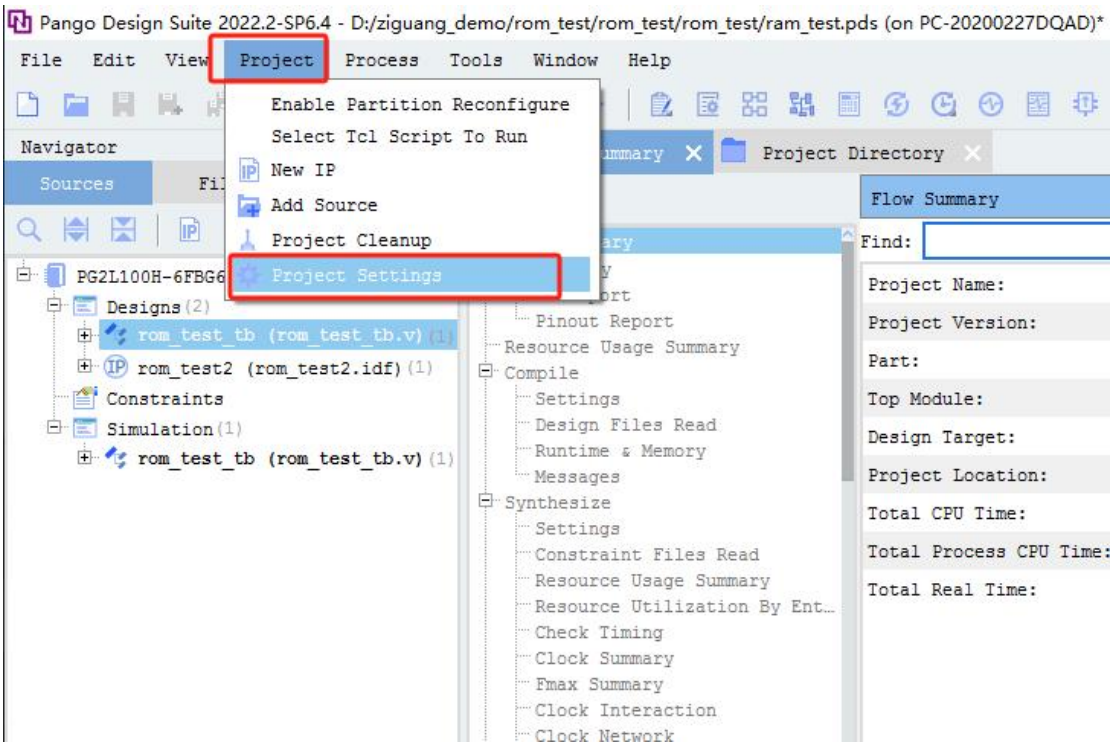


图 3.2-4

打开工程后, 选择 Project->Project Settings;

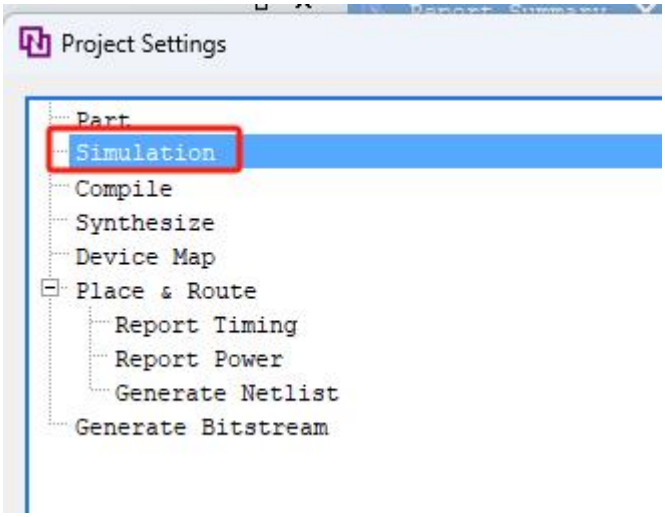


图 3.2-5

选择 Simulation 选项, 准备设置我们的仿真路径。

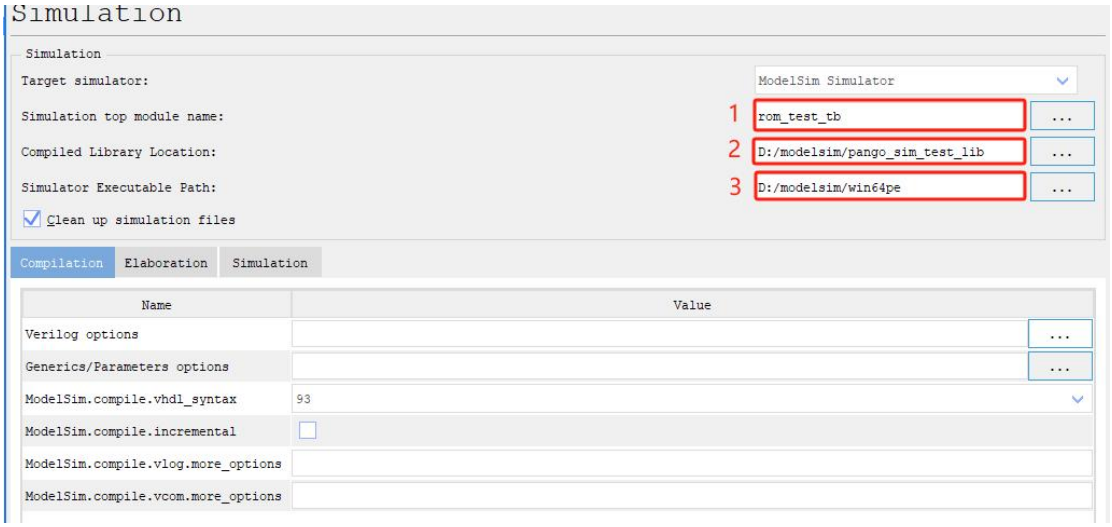


图 3.2-6

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

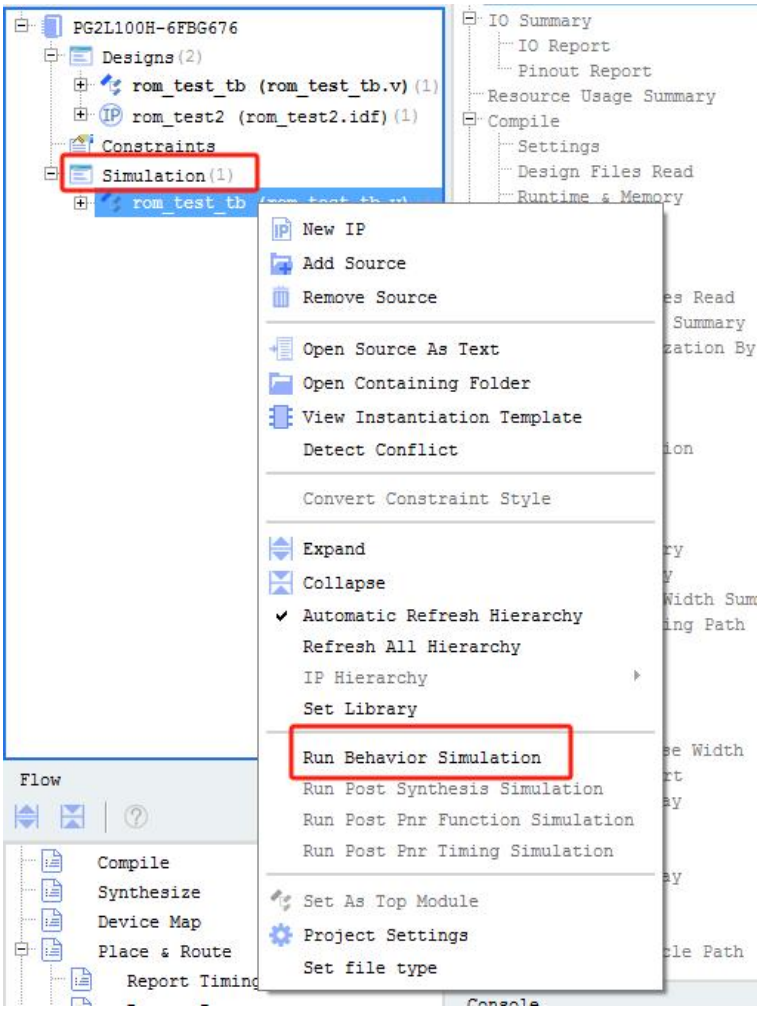


图 3.2-7

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:

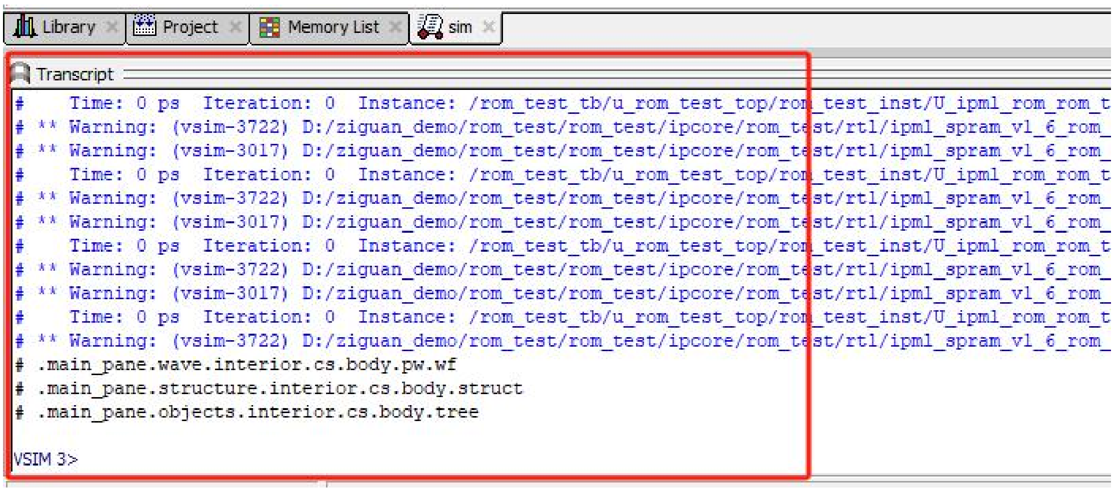


图 3.2-8

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(具体请看下一章节内容)。

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:

MES2L676-100HP-MINI

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。

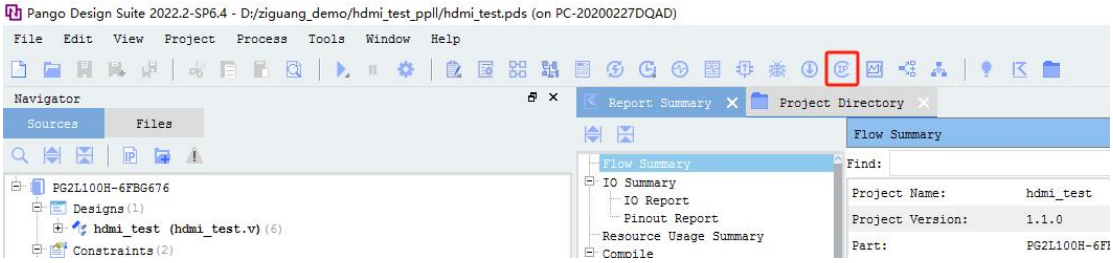


图 4.2-1

打开 PDS 后, 点击图 4.2-1 里红框部分的 IP 图标。

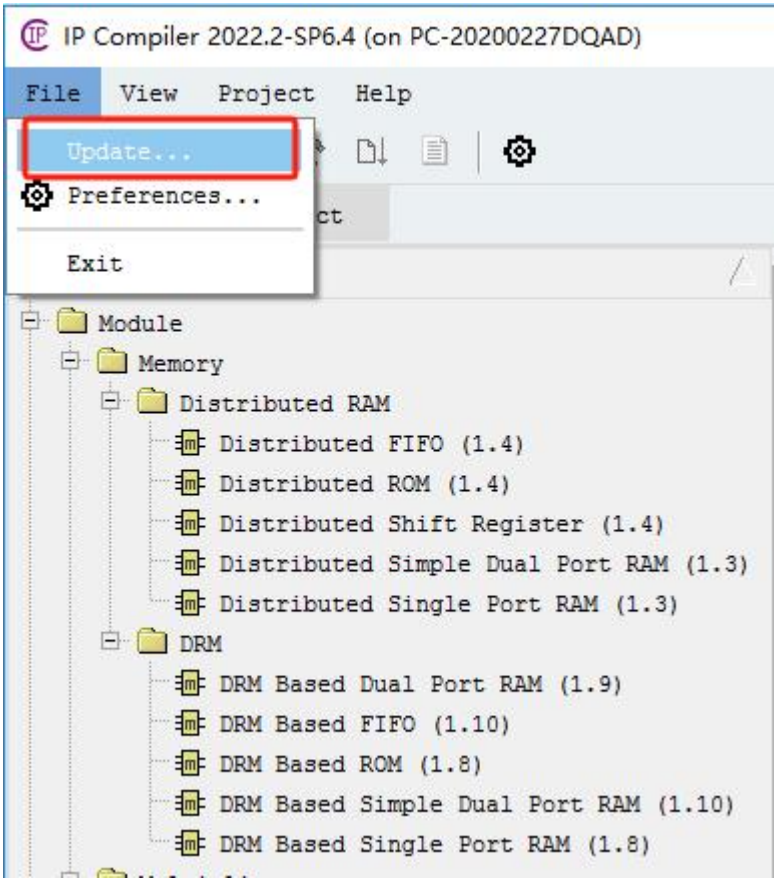


图 4.2-2

之后在弹出的选项卡的左上角点击 File->Update...

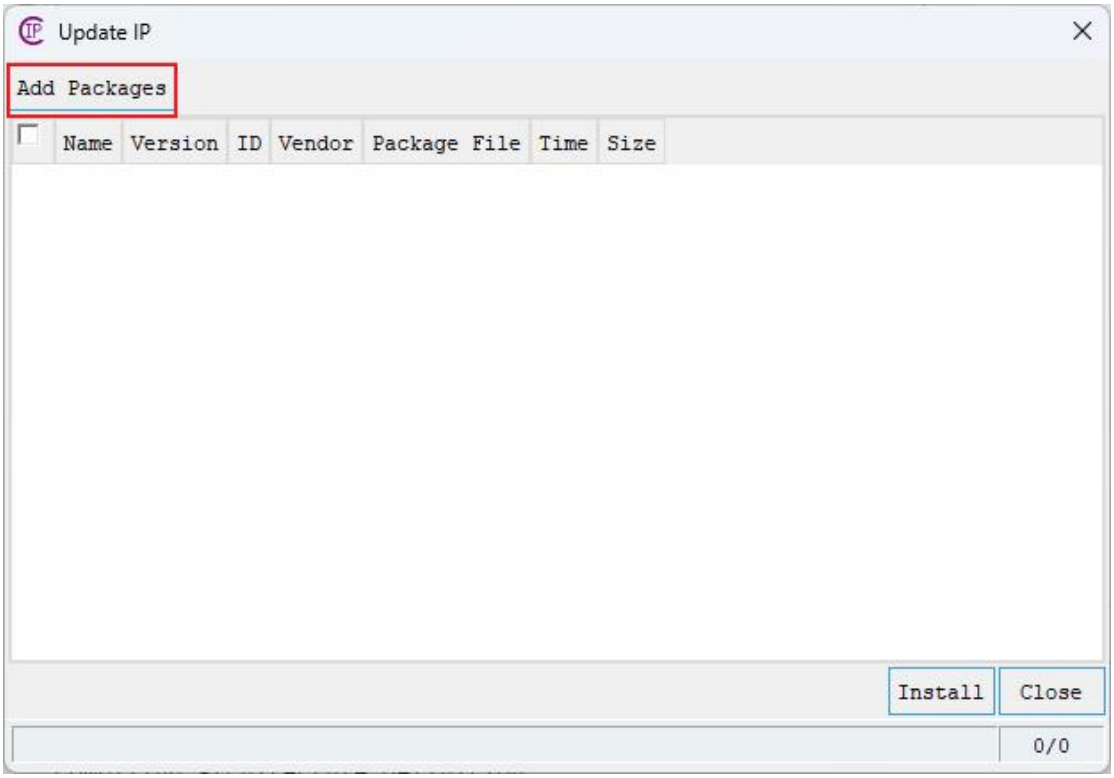


图 4.2-3

点击左上角 Add Package。

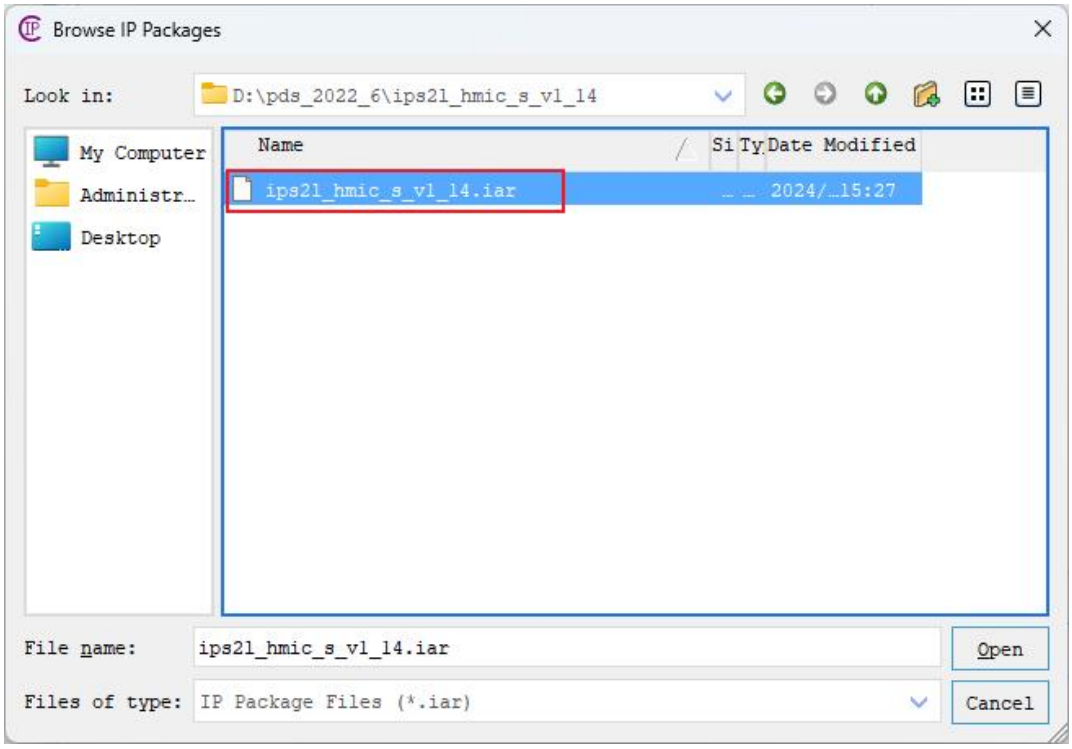


图 4.2-4

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。

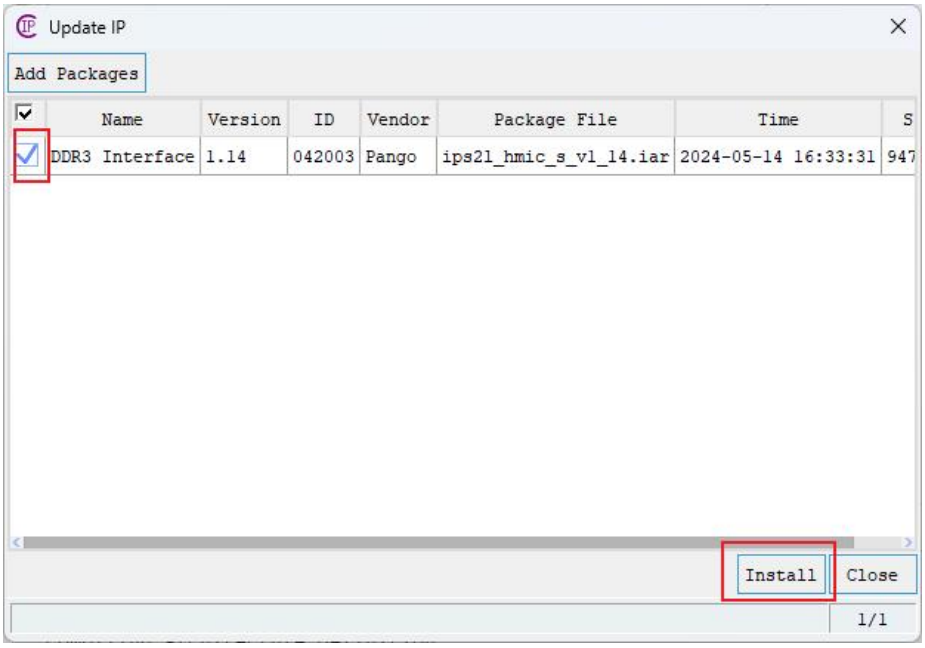


图 4.2-5

之后勾选上前面的√, 点击 Install 即可。

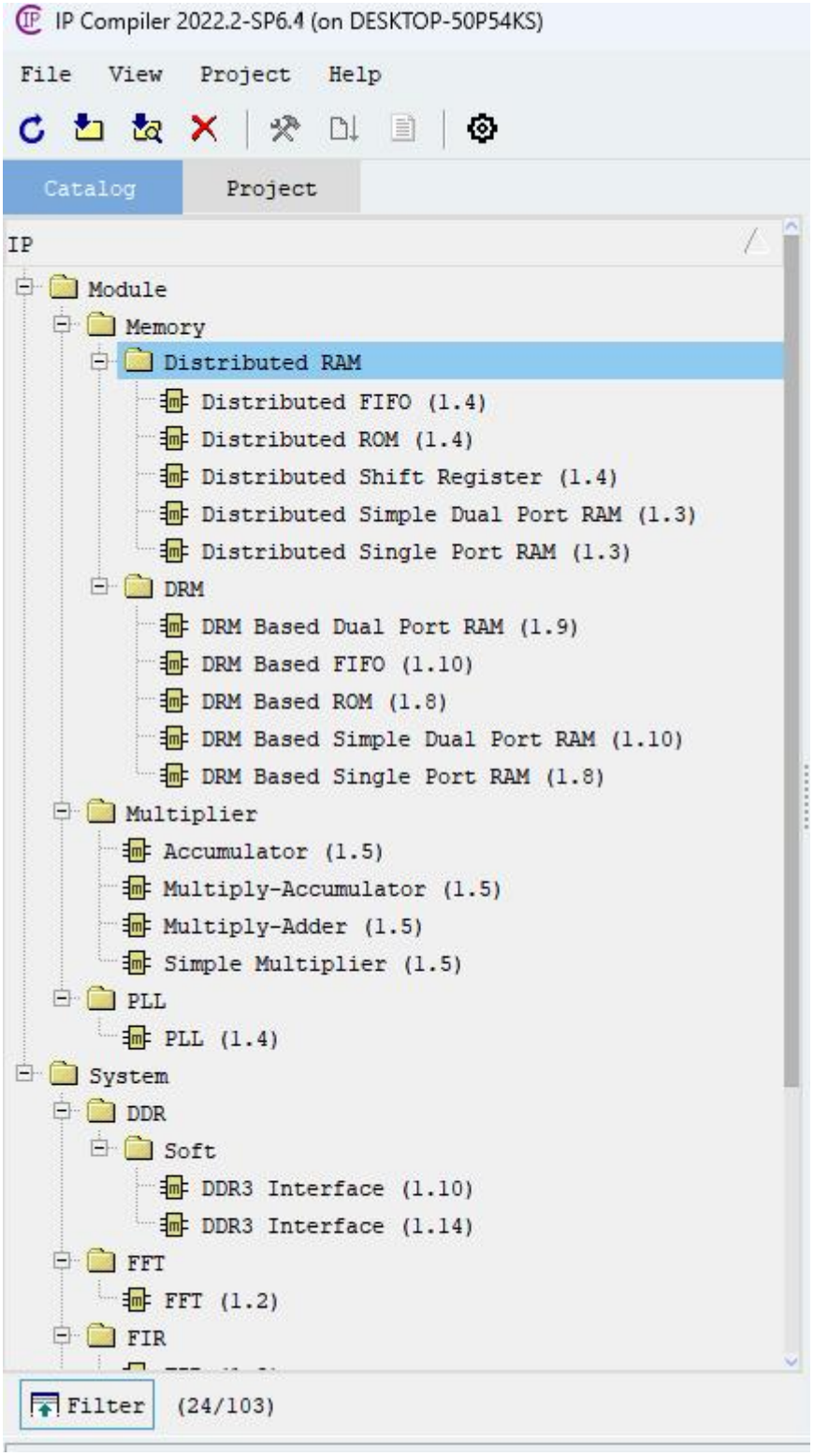


图 4.2-6

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

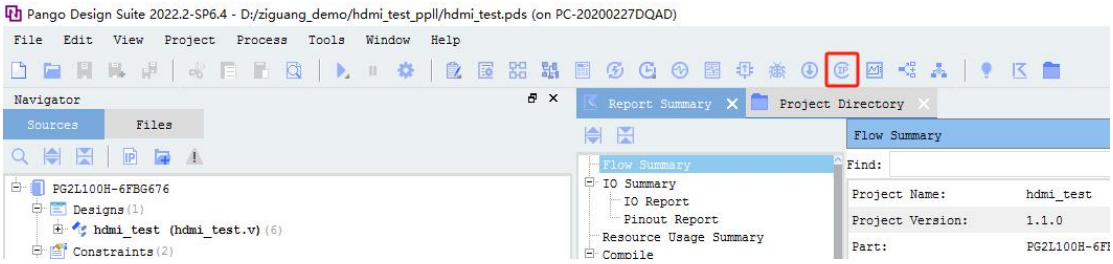


图 4.2-7

继续点击图 4.2-7 所示红框的图标。

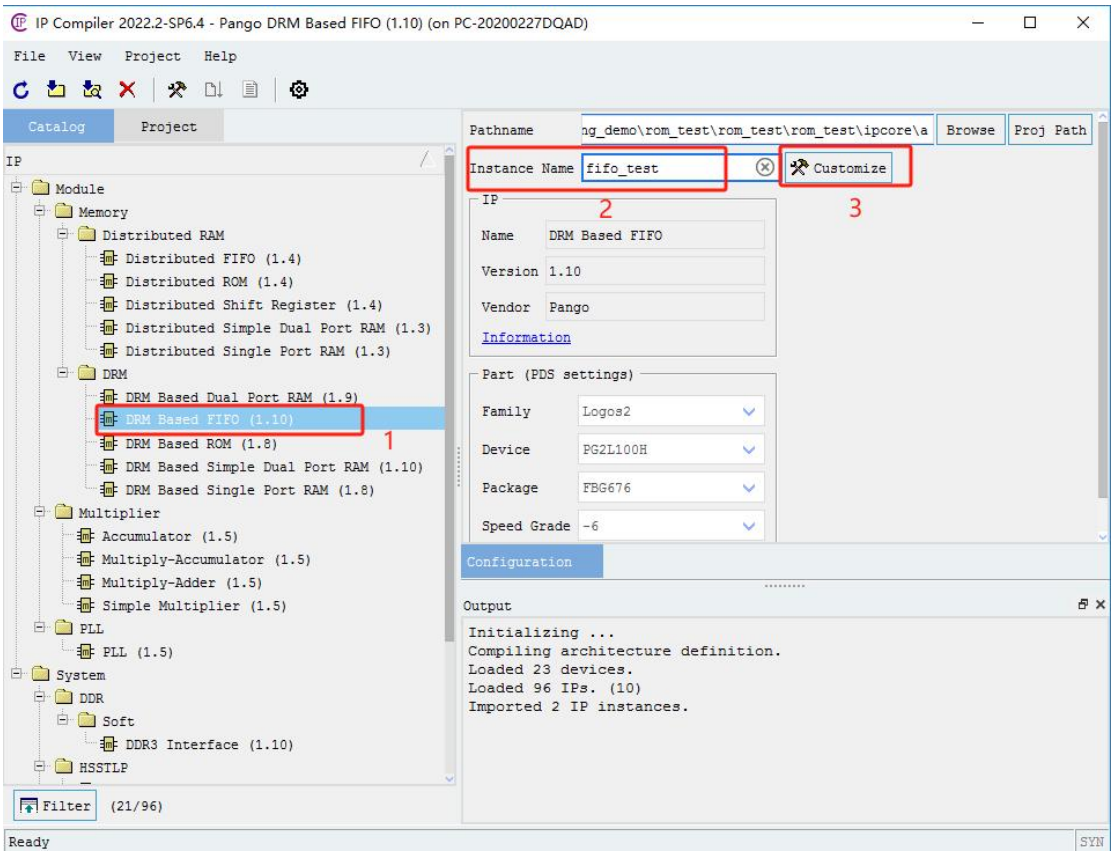


图 4.2-8

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:

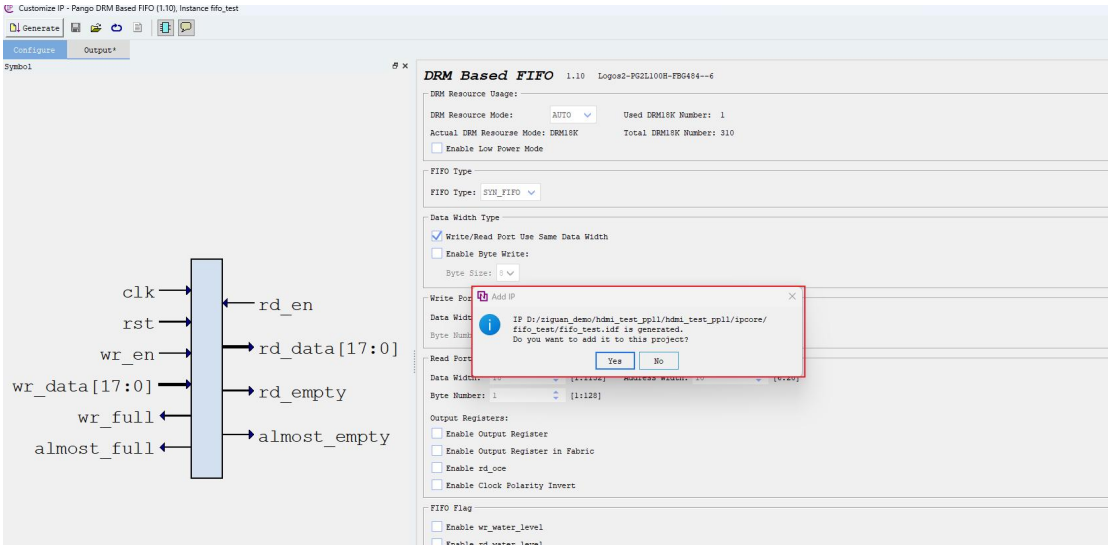


图 4.2-9

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：

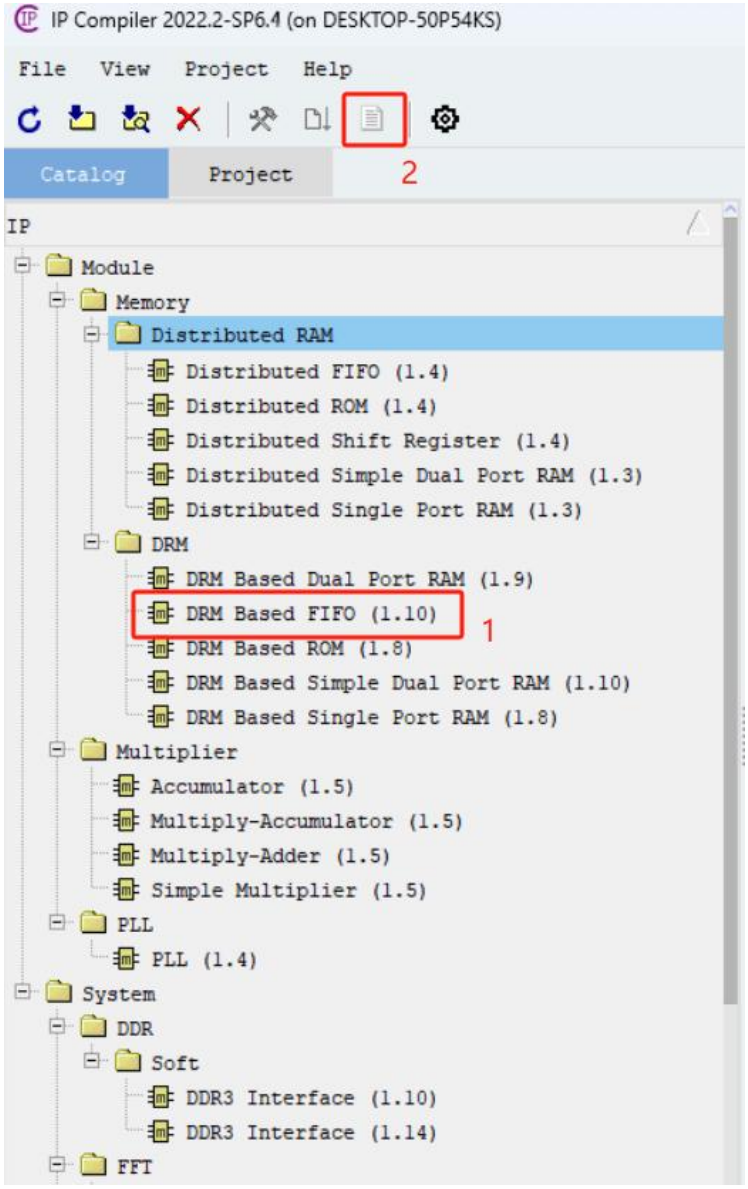


图 4.2-10

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。

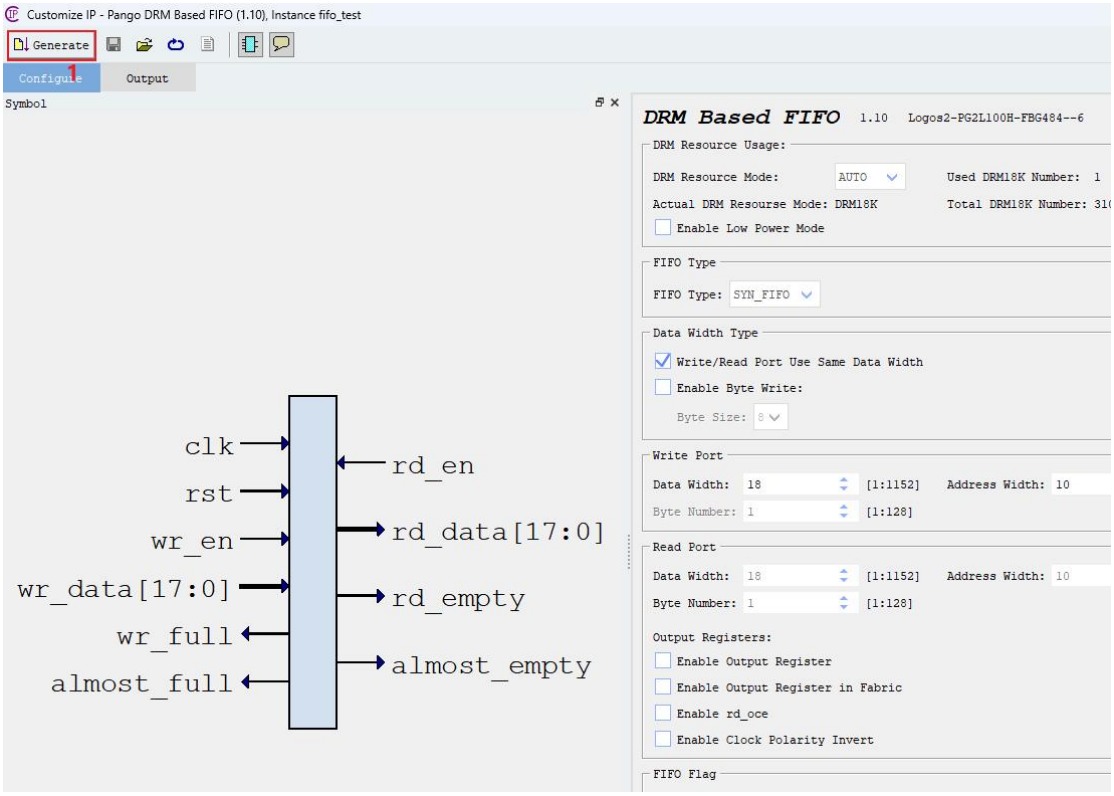


图 4.2-12

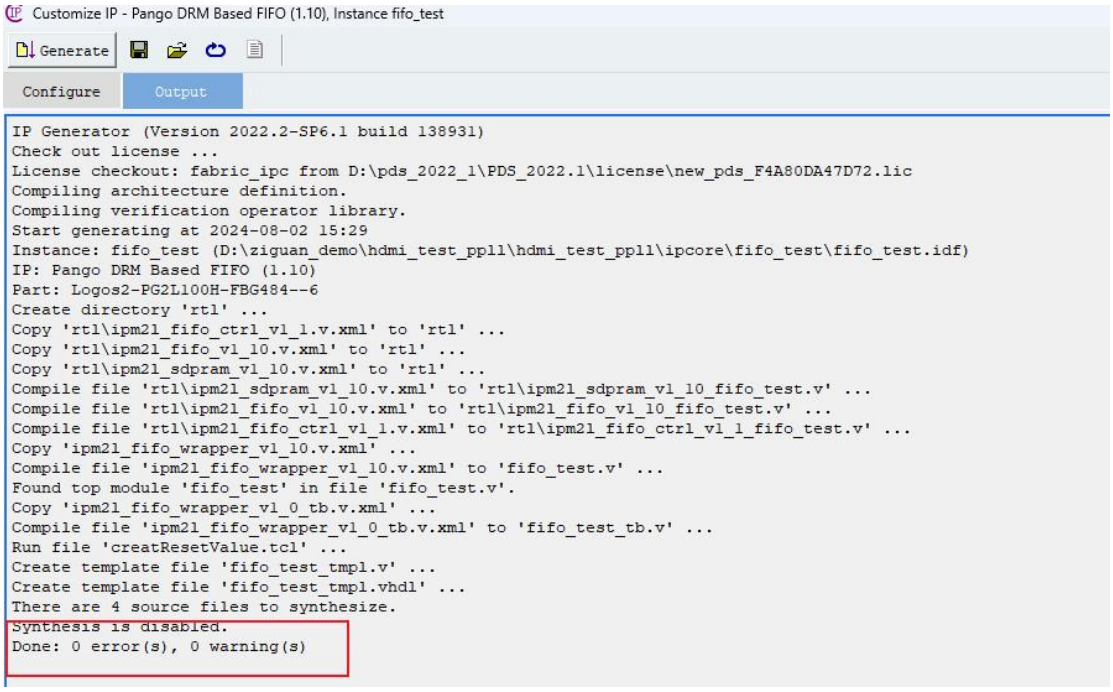


图 4.2-13

没有任何错误测表示生成成功。

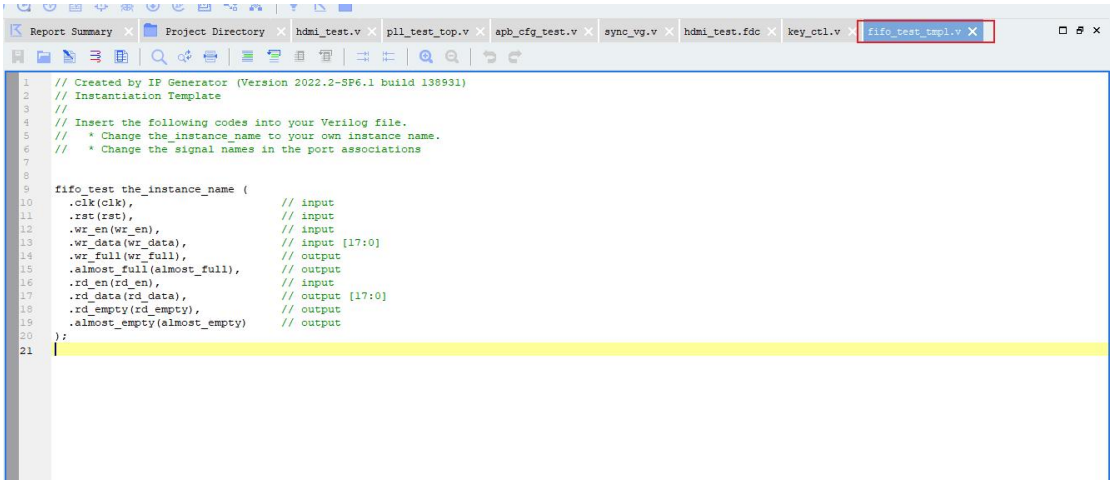


图 4.2-14

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

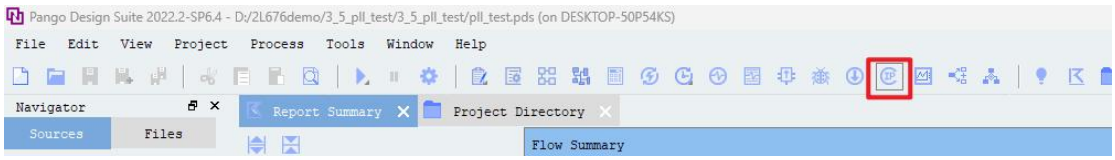


图 5.2-1 “IP”图标示意图

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：

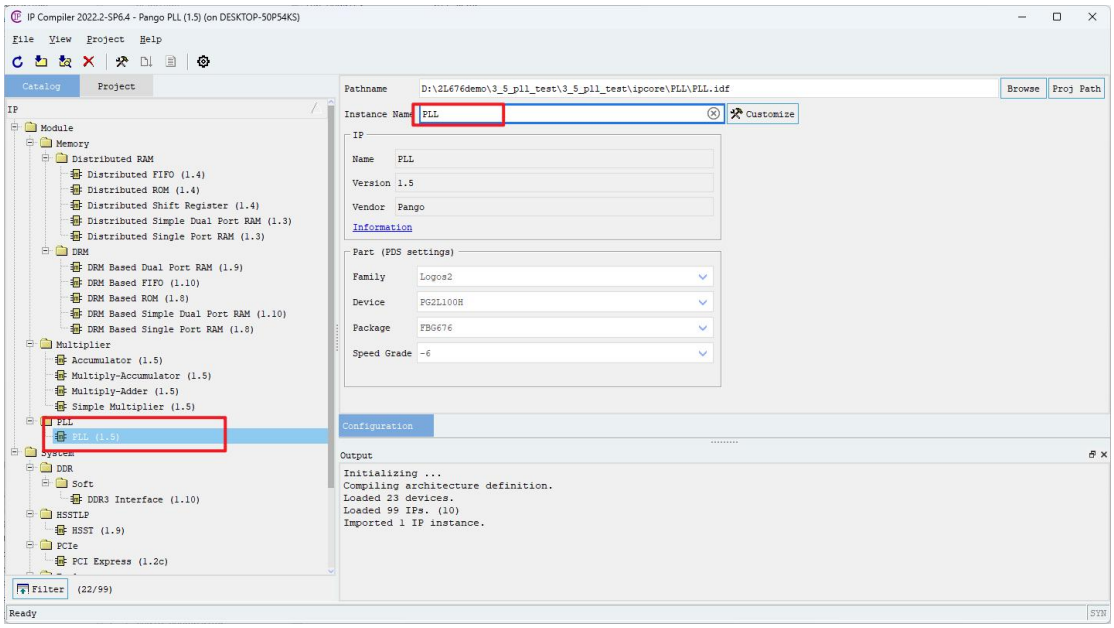


图 5.2-2 IP 配置流程图 1

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。

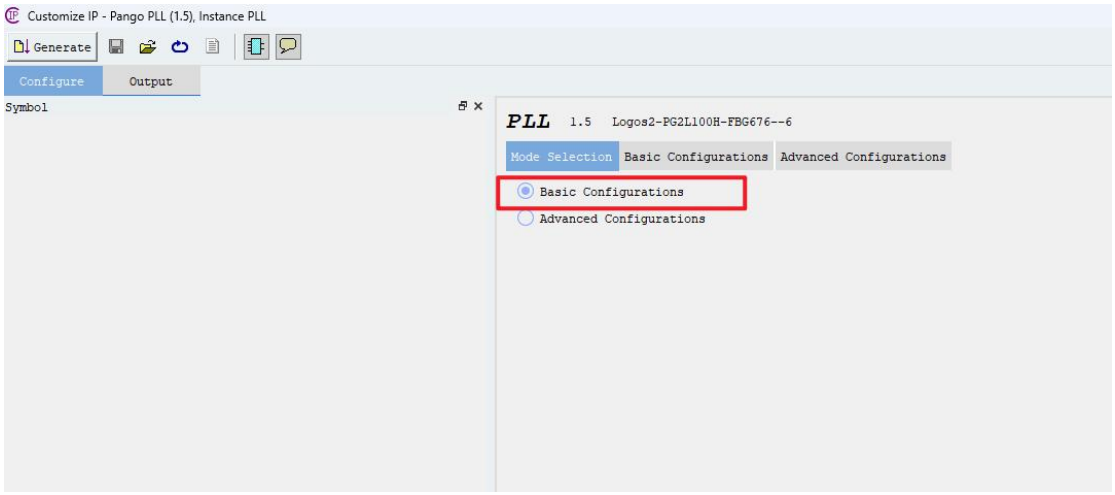


图 5.2-3 IP 配置流程图 2

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 81MHZ, 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:

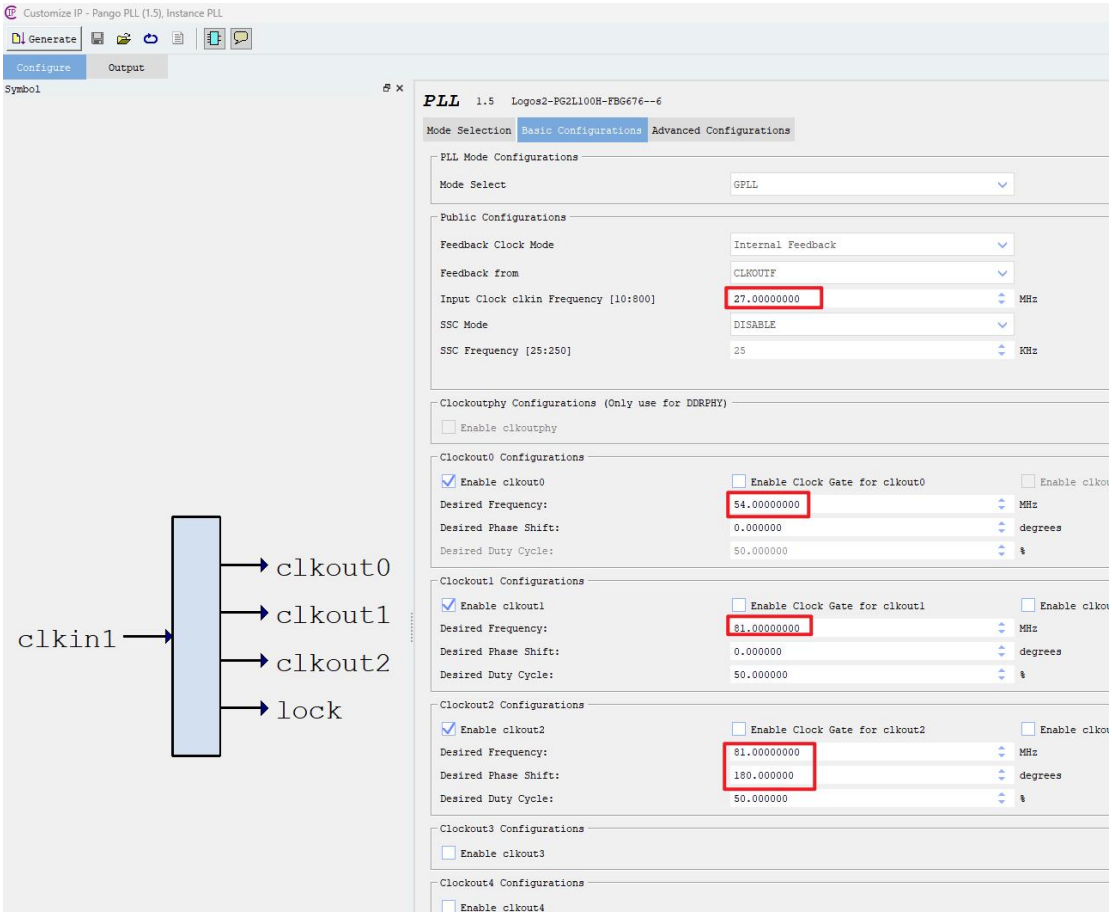


图 5.2-4 IP 配置流程图 3

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
```

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
```

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。

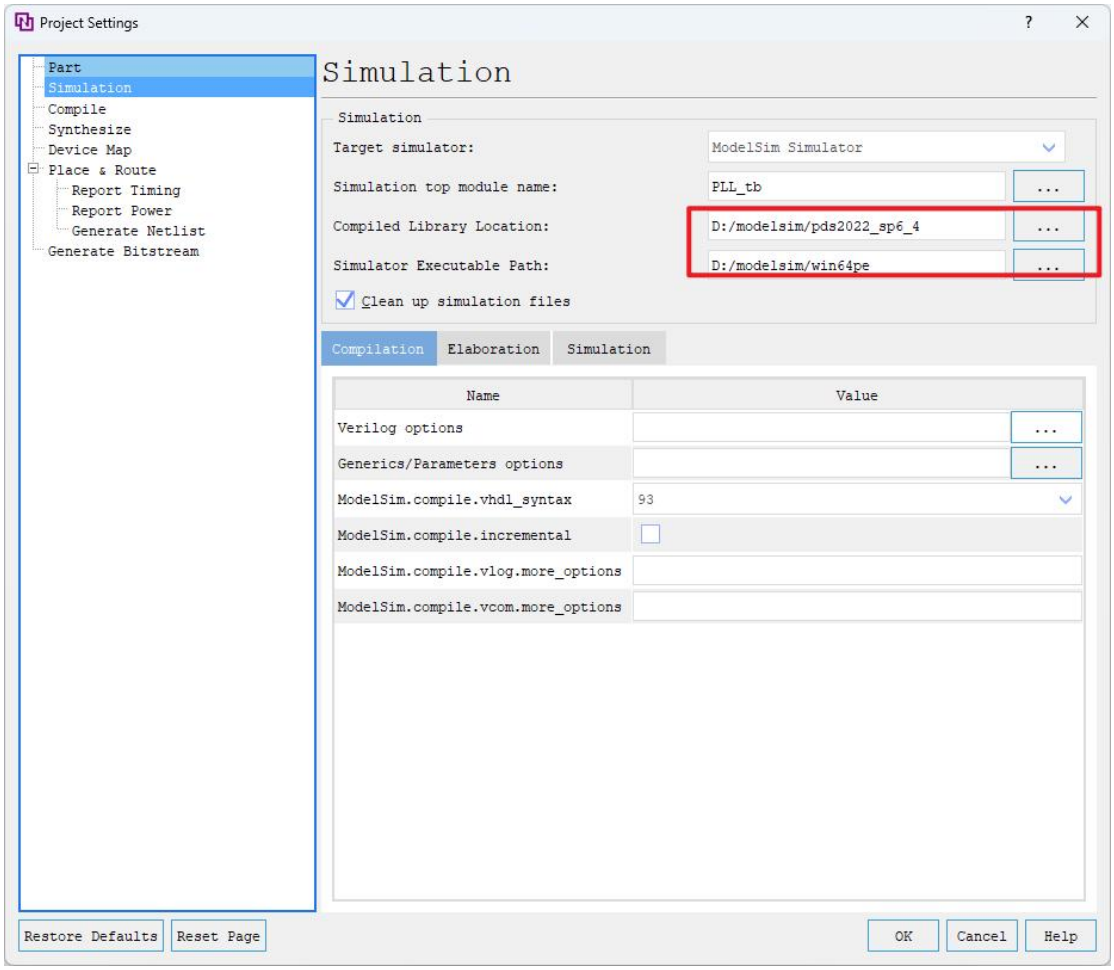


图 5.4-1 PDS 和 Modelsim 联合仿真流程图 4

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。

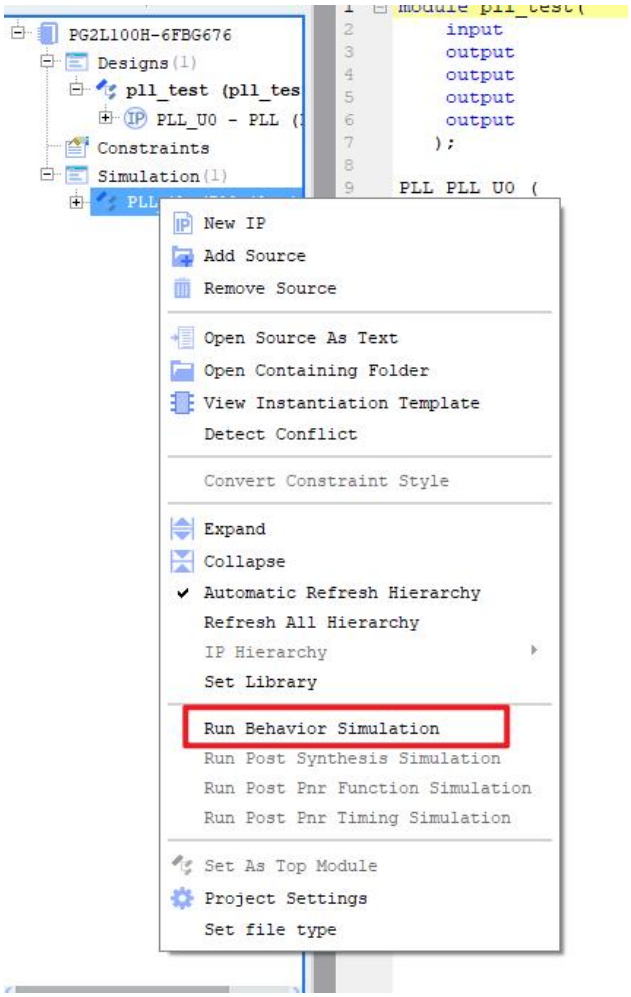


图 5.4-2 PDS 和 Modelsim 联合仿真流程图 5

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。

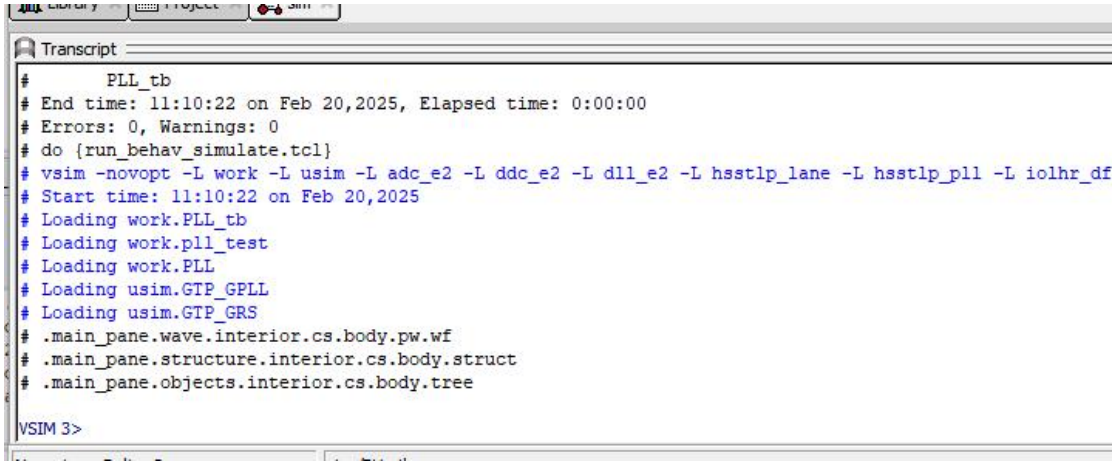


图 5.4-3 PDS 和 Modelsim 联合仿真流程图 6

5.5. 实验现象

点击 Wave 观察 PLL 输出信号:

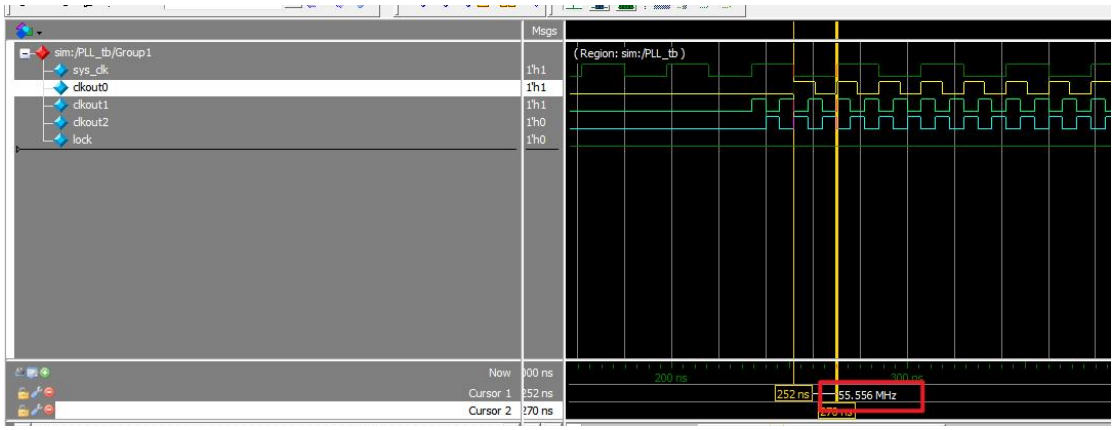


图 5.5-1 PLL IP 使用实验结果波形图 1

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。

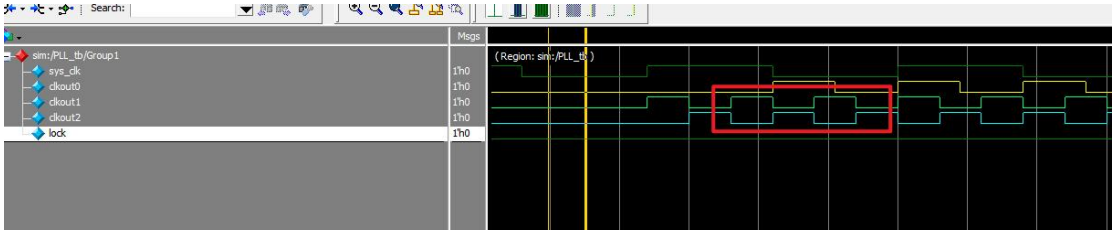


图 5.5-2 PLL IP 使用实验结果波形图 2

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:

MES2L676-100HP-MINI

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:

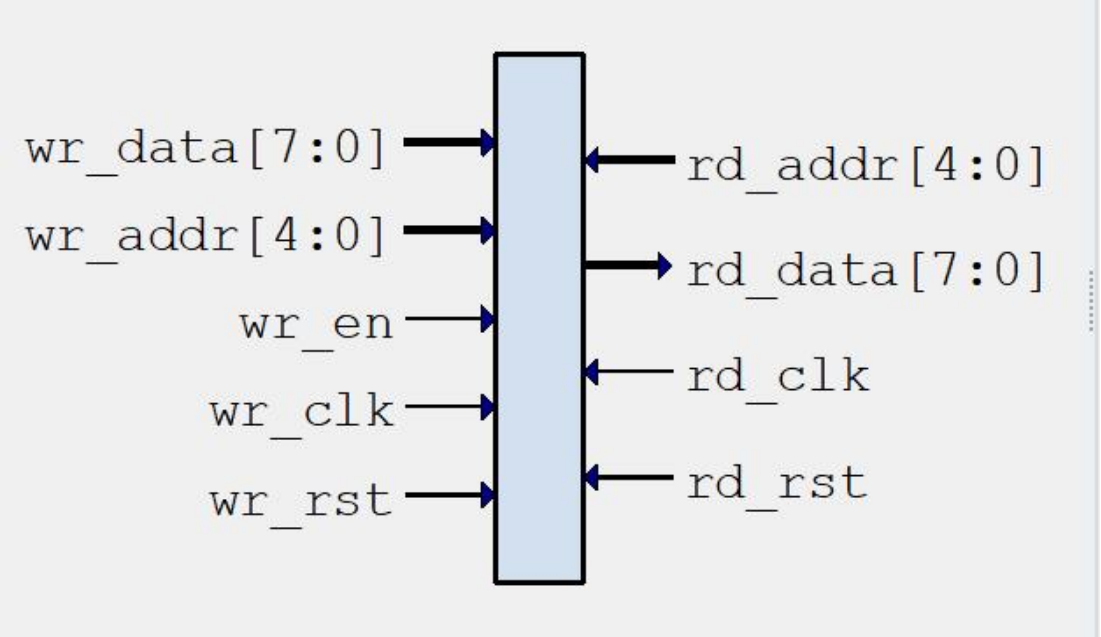


图 6.2-1

图 6.2-2 为 IP 配置:

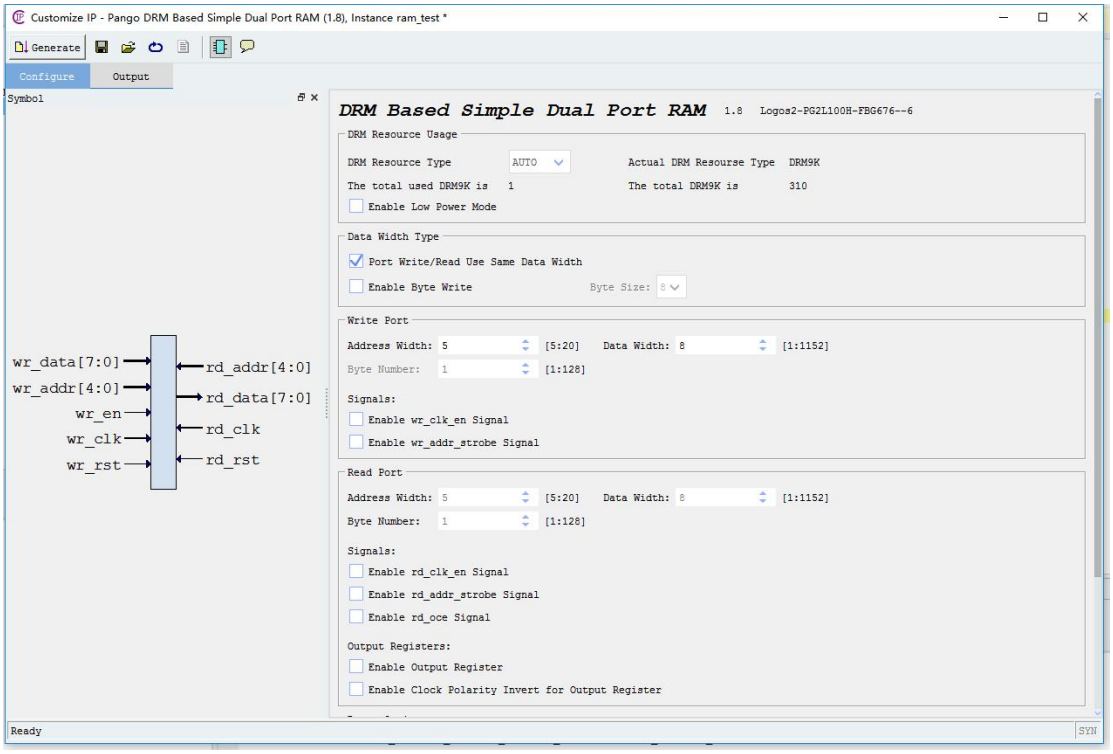


图 6.2-2

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。

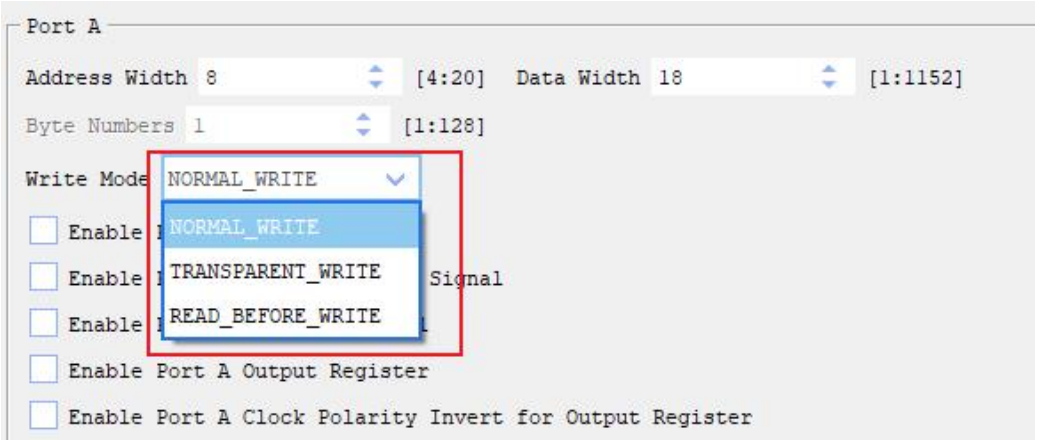


图 6.2-4

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

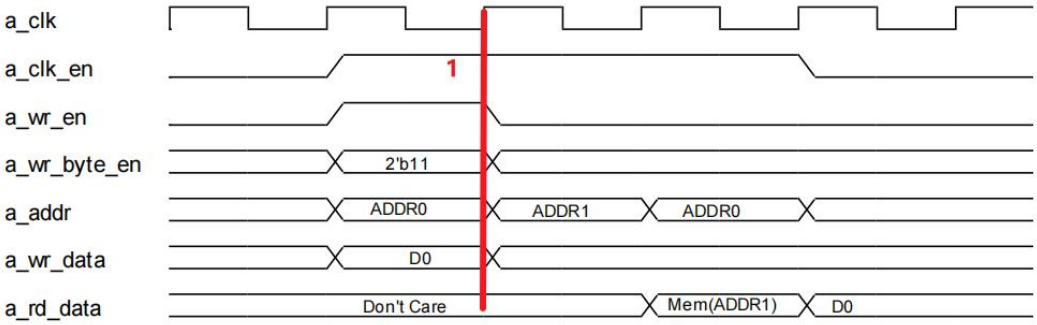


图 6.2-5

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

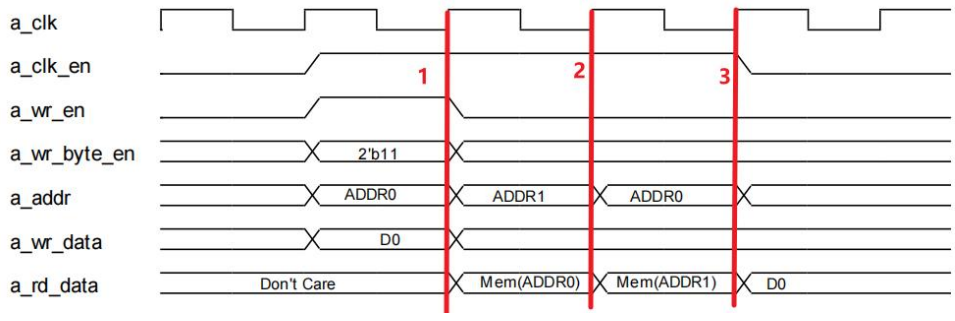


图 6.2-6

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

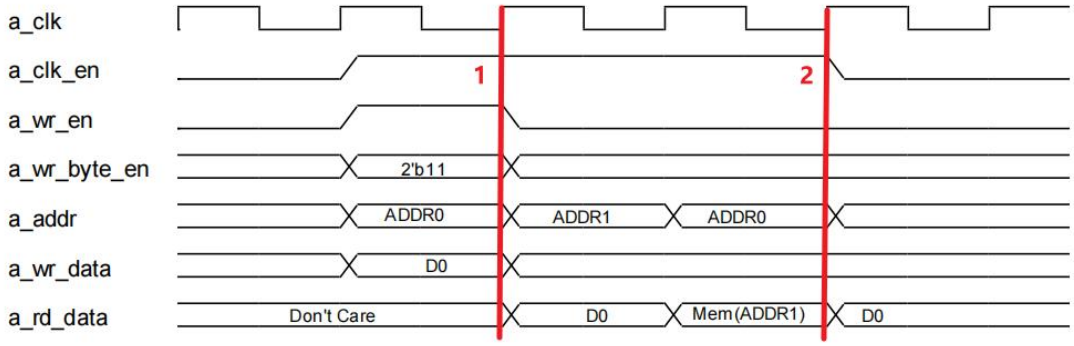


图 6.2-7

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:

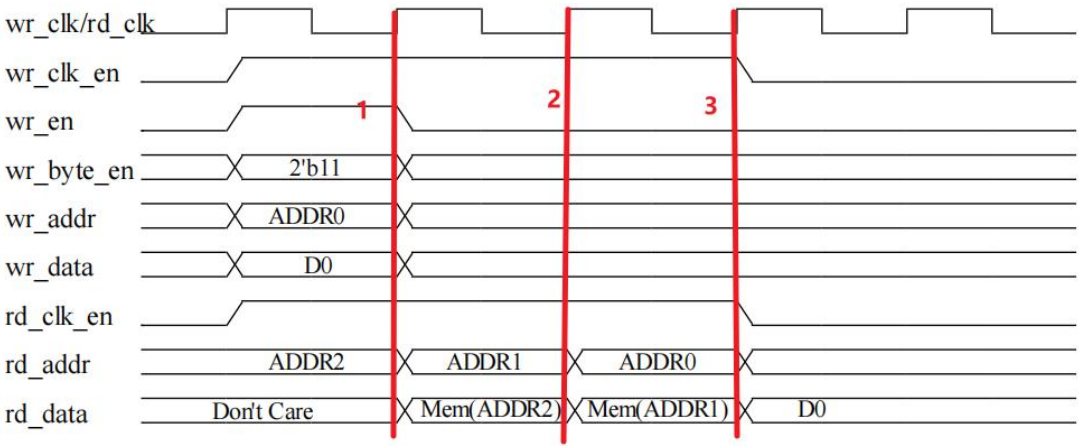


图 6.2-8

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:

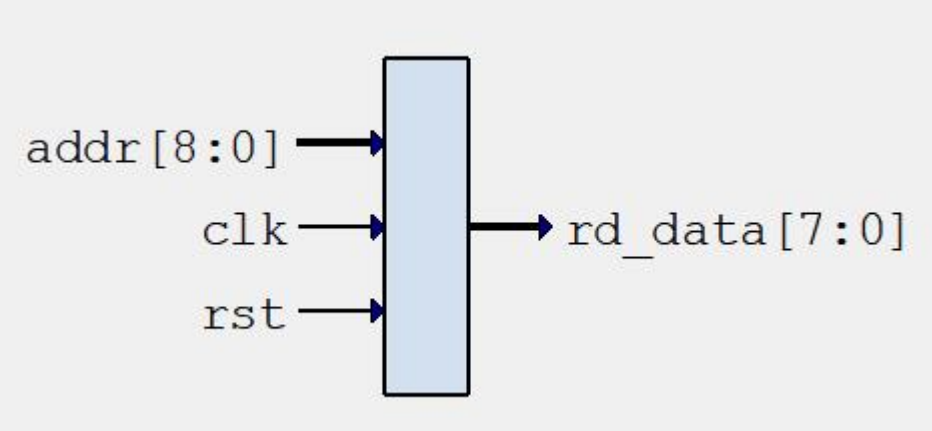


图 6.2-9

下图为 IP 配置:

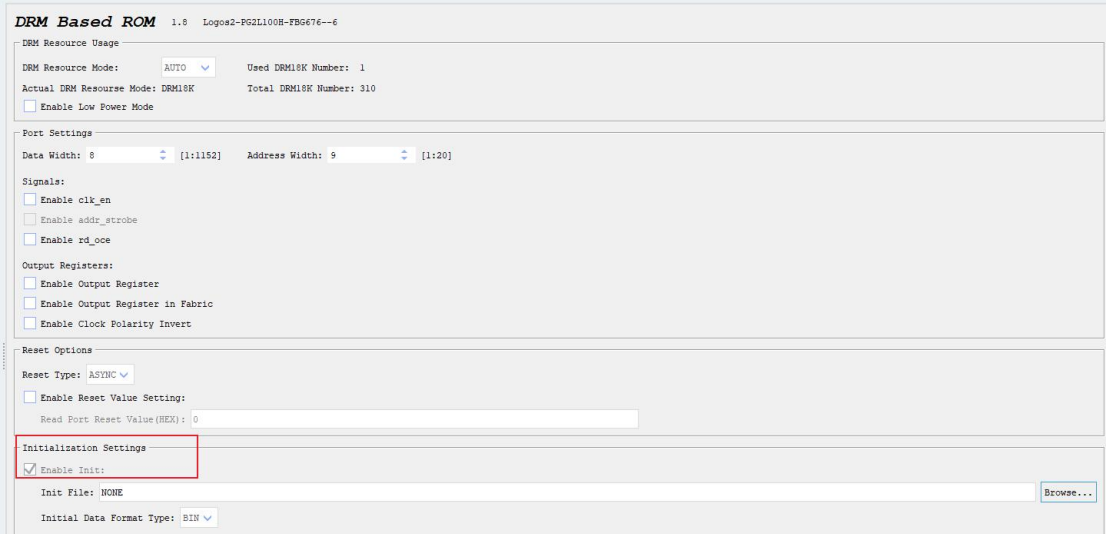


图 6.2-10

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。

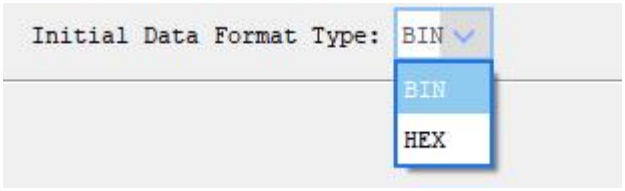


图 6.2-11

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

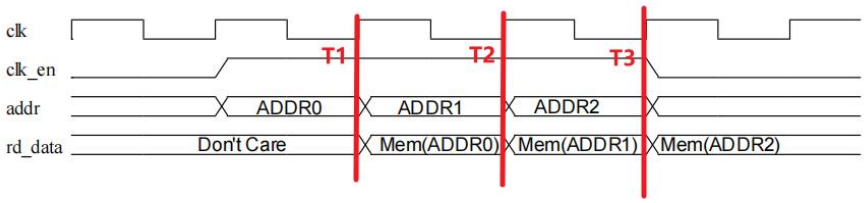


图 6.2-13

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。

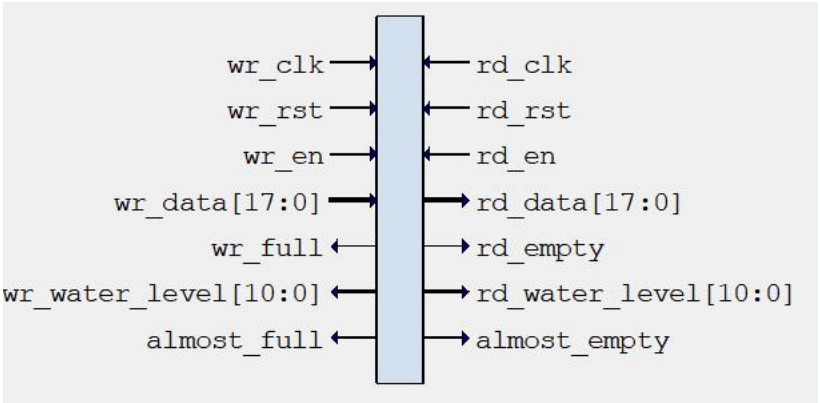


图 6.2-14

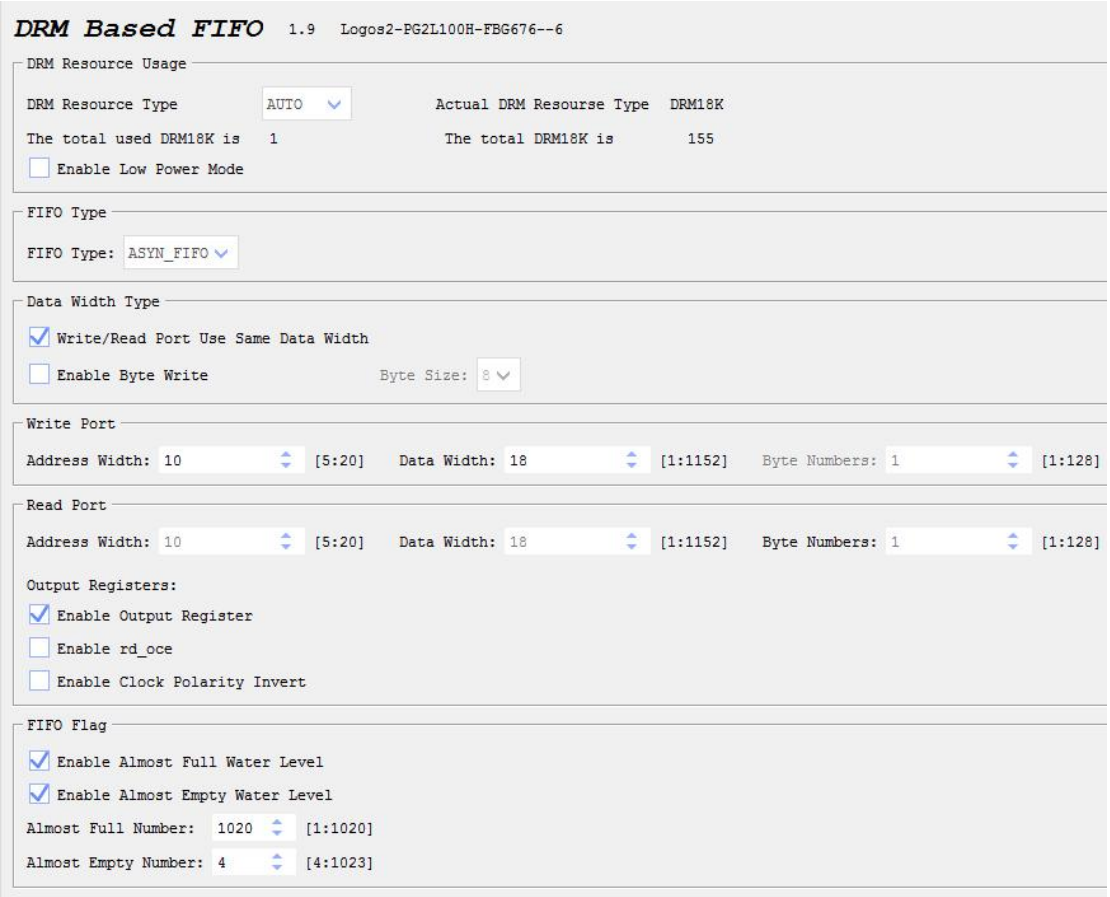


图 6.2-15

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。

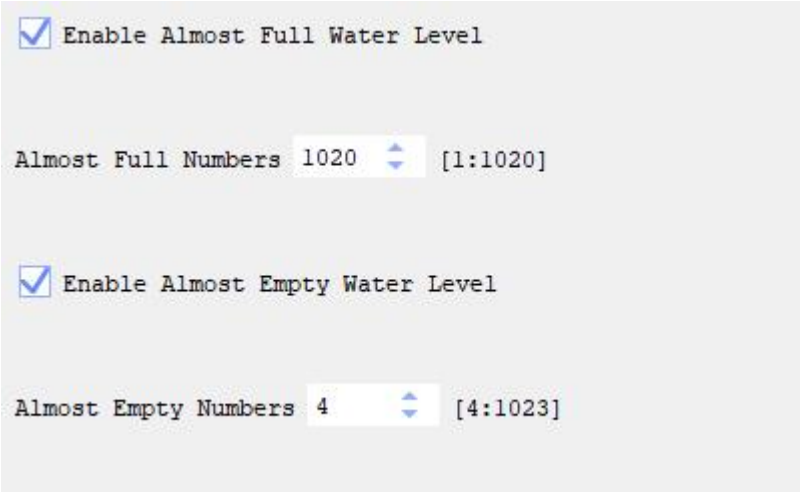


图 6.2-17

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

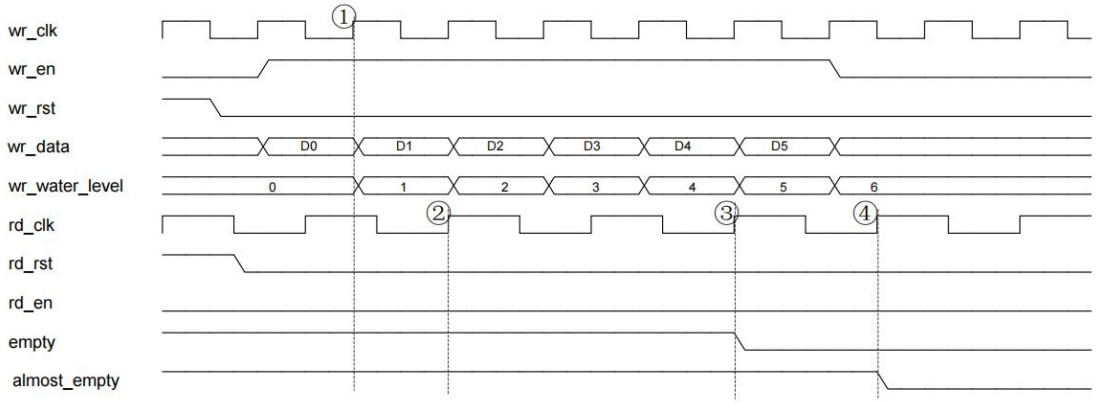


图 6.2-18

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

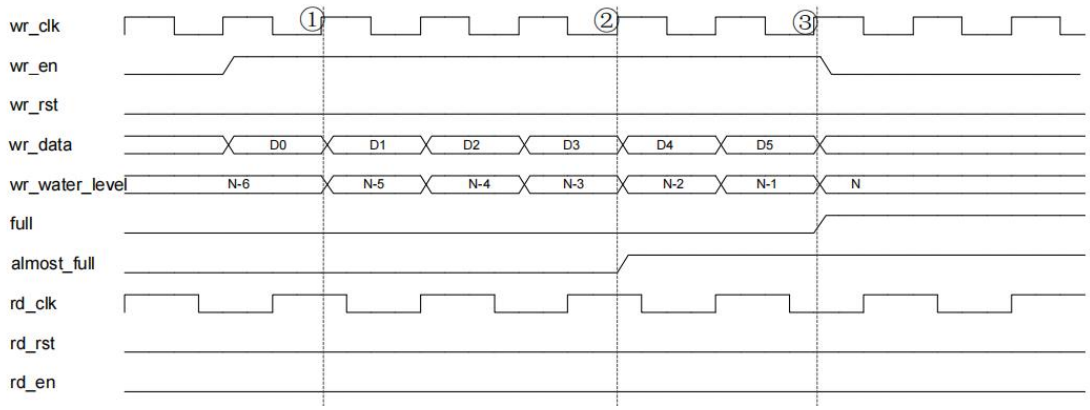


图 6.2-19

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

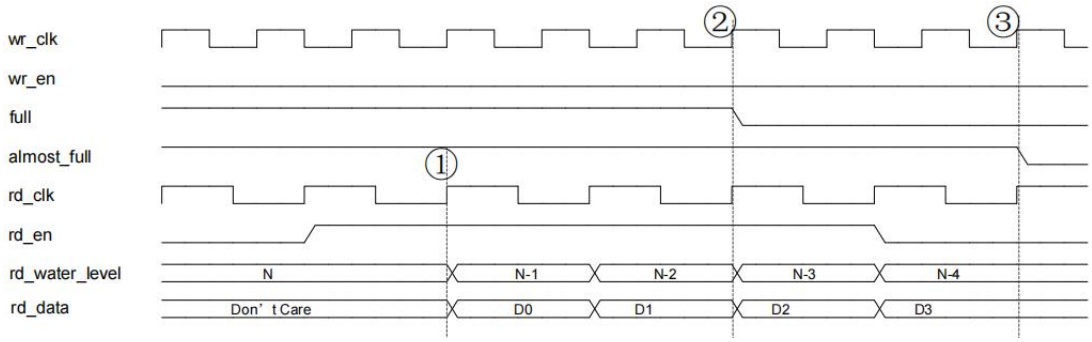


图 6.2-20

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

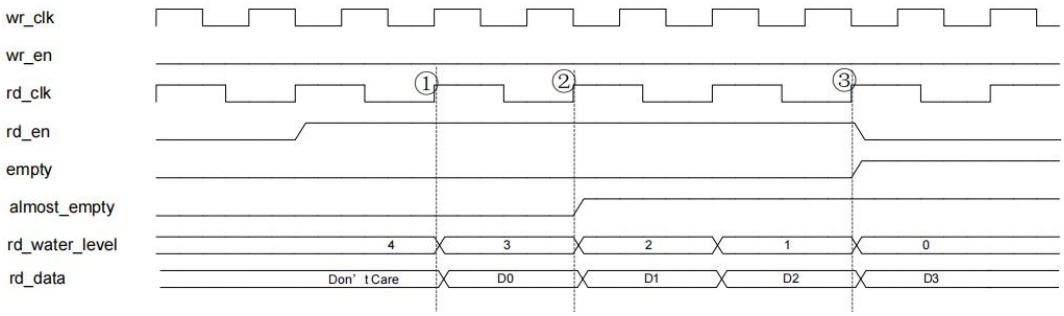


图 6.2-21

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

```
`timescale 1ns/1ns
module ram_test_tb();
reg   sys_clk;
reg   rd_clk ;
reg   rst_n;
reg   rw_en;   //读写使能信号

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
            else
            begin
              state   <=   2'd2;
              rd_addr <=   rd_addr+1'b1;
            end
          end
      2'd3:begin
            state   <=     2'd0;
          end

     default:   state   <=   2'd0;
     endcase
   end
end

//50MHZ
always#10 sys_clk = ~sys_clk;

//
GTP_GRS GRS_INST(
   .GRS_N(1'b1)
   ) ;
```

```
ram_test_top u_ram_test_top(
   .wr_clk  ( sys_clk  ),
   .rd_clk  ( sys_clk  ),
   .rst_n   ( rst_n    ),
   .rw_en   ( rw_en    ),
   .wr_addr ( wr_addr  ),
   .rd_addr ( rd_addr  ),
   .wr_data ( wr_data  ),
   .rd_data ( rd_data  )
);
endmodule
```

涉及到 tb 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和使能以及写入的数据。这里只讲解主要实现的功能, 首先代码的 38-42 行, 也就是 state=0 的时候, 拉高 rw_en, 并跳转到状态 1, 此时进入写操作(没有使能 clk_en, 可以不管), 下个时钟周期开始写入数据(注意是时序逻辑, 边沿采样, 所以是下个时钟周期才开始写数据), 即 state=1 的时候是一直在往 ram 里面写数据,在代码的 44 到 60 行就是写操作了, 可以看到, 当 wr_addr 不等于 31 的时候, wr_data 和 wr_addr 不断加 1(rd_addr 这里+1, 可以看视频讲解, 主要为了验证伪双端口的时序), 当 wr_addr 等于 31 的时候, 在下个时钟周期把数据清 0, 状态跳转, 在当前时钟周期下还会再往地址 31 里面写入数据, 所以在该时钟周期, 一共写入了 32 个数据(从地址 0 写到地址 31)。即状态 1 完成写入 32 个数据后跳转到 state=2 的逻辑。代码的 61-72 行, 也就是 state=2 的时候, 在每个周期的上升沿让 rd_addr 不断累加, 直到 rd_addr=31 的时候, 在下个时钟周期清空地址并让状态跳转的操作, 而在当前时钟周期会继续把地址 31 的数据读出来, 完成读取地址 0-31 的数据, 一共 32 个数据, 所以该状态主要完成读取 32 个数据, 然后在下个时钟周期就跳转到 state=3。state=3 可以看到其主要作用就是等待一个时钟周期, 然后跳转回去 state=0 下, 起到一个延时作用。

具体波形大家可以看视频仿真, 或者自己尝试仿真, 根据波形来看代码。因为这里是时序逻辑, 所以如果是初学者, 纯看文字可能会对 rd_addr=31 这一时刻还会再读一个数据感到疑惑, 建议直接仿真, 或者观看视频讲解的仿真部分, 可以帮助快速理解。

可以总结出一句话就是**时序逻辑的赋值总在下一个时钟周期才生效**。所以在 rd_addr=31 时执行的操作要在下一个时钟周期才会被采样生效。所以当前时钟还是会再从 RAM 读出一个数据。

仿真代码的讲解到此结束, 大家要注意时序逻辑的特点, 具体的内容请看视频讲解。

6.5.2.ROM 仿真测试

```
`timescale 1ns/1ns
module rom_test_tb();
reg   sys_clk;
reg   rst_n;
reg   [9:0]   rd_addr;
```

```
wire   [63:0]    rd_data;

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

end

//50MHZ
always#10 sys_clk = ~sys_clk;
//
GTP_GRS GRS_INST(
   .GRS_N(1'b1)
   ) ;

always@(posedge sys_clk or negedge rst_n)    begin
   if(!rst_n)
      rd_addr    <=    10'd0;
   else
      rd_addr    <=    #2 rd_addr + 1'b1;
end

rom_test_top u_rom_test_top(
   .rd_clk   ( sys_clk  ),
   .rst_n    ( rst_n    ),
   .rd_addr  ( rd_addr  ),
   .rd_data  ( rd_data  )
);

endmodule
```

代码 31-36 行例化了 ROM 的顶层模块, 该模块里面其实就是调用了 ROM IP, 然后把信号引出端口, 没有任何逻辑操作。

代码 24-29 行通过一个 always 块不断生成地址，任何给到 ROM IP，将数据读出，由于没勾选 clk_en 信号，所以数据在 ROM 复位完成后就会不断读出。所以并没有复杂的逻辑，就是让地址从 0 不断累加，把数据读出。

仿真代码的讲解到此结束，大家要注意时序逻辑的特点，具体的内容请看视频讲解。

6.5.3. FIFO 仿真测试

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

reg sys_clk;
reg rst_n;

reg       [7:0]   wr_data;
reg               wr_en;
reg               rd_en;

reg               rd_state;    //读状态
reg               wr_state;

wire      [7:0]   rd_data;
reg       [7:0]   rd_cnt;

wire      [7:0]   rd_water_level;
wire      [7:0]   wr_water_level;

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


end

always#10 sys_clk = ~sys_clk;    //50MHZ
```

```
always@(posedge sys_clk or negedge rst_n)   begin
   if(!rst_n)
   begin
      wr_state    <=    1'd0;
      wr_en   <=   1'd0;
      wr_data <=   8'd0;
   end
   else
   begin
      case(wr_state)
         1'd0:    if(wr_water_level == 127)    //128 个数据
                  begin
                    wr_en    <=    #2 1'd0;
                    wr_data  <=    #2 8'd0;
                    wr_state <=    #2 1'd1;
                  end
                  else
                  begin
                    wr_en    <=    #2 1'd1;
                    wr_data  <=    #2 wr_data+1'b1;
                    wr_state <=    #2 1'd0;
                  end

         1'd1:    if(rd_cnt == 127)
                   wr_state    <=    #2 1'd0;


         default:    wr_state    <=1'd0;
      endcase
   end
end

always@(posedge sys_clk or negedge rst_n)   begin
   if(!rst_n)
   begin
      rd_state<=   1'd0;
```

```
        rd_en  <=  1'd0;
        rd_cnt <=  8'd0;
    end
    else
    begin
        case(rd_state)
            1'd0:  if(rd_water_level >= 8'd128)   //等待 128 个数据
                   begin
                       rd_state   <=   #2 1'd1;
                       rd_en      <=   #2 1'd1;
                   end
                   else
                   begin
                       rd_cnt     <= #2 8'd0;
                       rd_state   <=  #2 1'd0;
                   end

            1'd1:  begin

                       rd_cnt <=   #2 rd_cnt + 1'b1;
                       if(rd_cnt == 127)
                       begin
                           rd_en      <=  #2 1'd0;
                           rd_state   <=  #2 1'd0;
                       end
                   end
            default:   rd_state   <=  1'd0;
        endcase
    end
end

GTP_GRS GRS_INST(
    .GRS_N(1'b1)
    ) ;

fifo_test_top u_fifo_test_top(
```

```
    .sys_clk       ( sys_clk       ),
    .rst_n         ( rst_n         ),
    .wr_data       ( wr_data       ),
    .wr_en         ( wr_en         ),
    .rd_en         ( rd_en         ),
    .wr_water_level  ( wr_water_level  ),
    .rd_water_level  ( rd_water_level  ),
    .rd_data       ( rd_data       )
);
endmodule
```

涉及到 tb 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。整个设计分为读写两个状态的控制。分别完成了写入 128 个数据, 和读出 128 个数据, 由于 FIFO 不需要地址, 所以只需要产生使能信号即可。

首先看写状态, 在 wr_state=0 时, 拉高写使能, 并让 wr_data 不断累加, 往 FIFO 里面写数据, 当 wr_water_level=127 的时候, 拉低写使能, 写数据置 0, 写状态跳转到 1, 注意此时还会再写入一个数据, 所以到此一个写入了 128 个数据。至于拉低写使能, 写数据置 0, 写状态跳转到 1 这些操作将在下一个时钟周期才会被采样生效。之后, 在 wr_state=1 时, 不断等待 rd_cnt, 该条件就是判断当读出 128 个数据的时候, wr_state 跳转到 0 状态。

接下来看读状态, 在 rd_state=0 的时候, 一旦可读的数据量超过 128 个(包括 128), 状态跳转到 rd_state=1 下, 然后开始读出数据, 同时在 rd_state=1 下用变量 rd_cnt 对我们的读出数据也进行计数, rd_cnt 从 0 开始计数,当 rd_cnt=127 的时候会再往 FIFO 读出一个数据,所以此时就一共读出了 128 个数据,下一个时钟周期 rd_en 和 rd_state 都将置 0。

具体波形大家可以看视频仿真, 或者自己尝试仿真, 根据波形来看代码。因为这里是时序逻辑, 所以如果是初学者, 纯看文字可能会对 rd_cnt=127 这一时刻还会再读一个数据感到疑惑, 建议直接仿真, 或者观看视频讲解的仿真部分, 可以帮助快速理解。

可以总结出一句话就是时序逻辑的赋值总在下一个时钟周期才生效。所以在 rd_cnt=127 时执行的操作要在下一个时钟周期才会被采样生效。所以当前时钟 rd_en 还是为 1, 会再从 FIFO 读出一个数据。

仿真代码的讲解到此结束。

7. 基于紫光 FPGA 的键控 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。

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

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)。

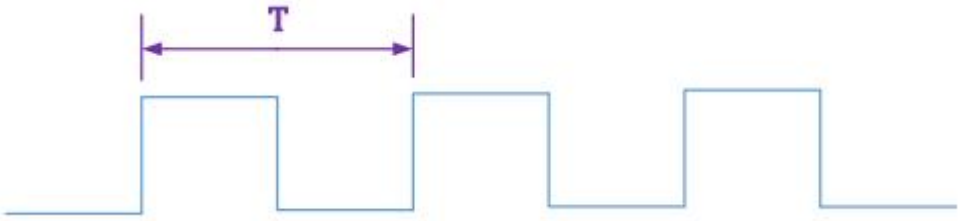


图 7.2-1

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:

$$f = 1/T$$

在 100K 板卡上单端时钟有一个 27MHZ 的时钟。

所以其周期约为 37.037ns。而在我们 FPGA 的设计中,我们的 always 块通常都是在时钟的上升沿时对数据进行赋值, 因此我们可以定义一个变量, 每到时钟的上升沿就让该变量+1, 让其变成一个计数器, 该变量每加 1 就表示经过了 37.037ns, 那如果要定时 1s 的话, 只需要让其计数到 26999999 即可, 因为从 0 开始计数, 所以计数到 26999999 即可, 此时就是一秒了。以此类推, 13499999 就是 0.5s。

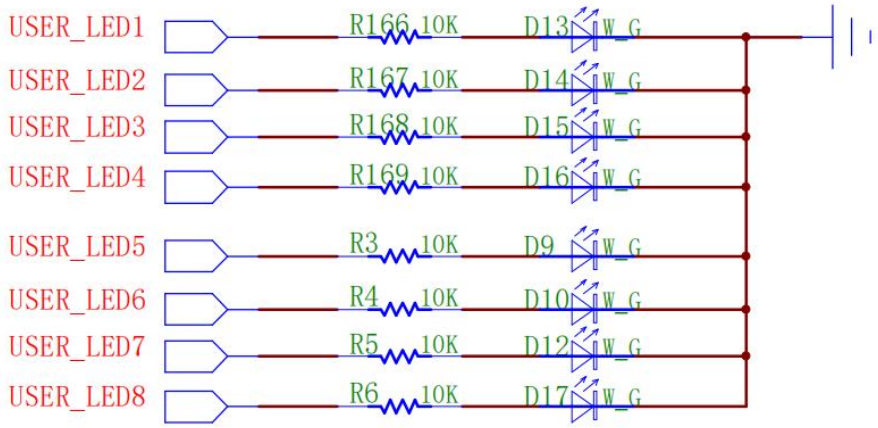


图 7.2-2

图 7.2-2 为开发板上 8 个 LED 灯的原理图。

控制 0.5s 更换一次 LED 灯状态, 使 LED 灯呈现流水灯现象, 可以每 0.5s 依次点亮一个 LED 灯。(高电平用 1 表示, 低电平用 0 表示)

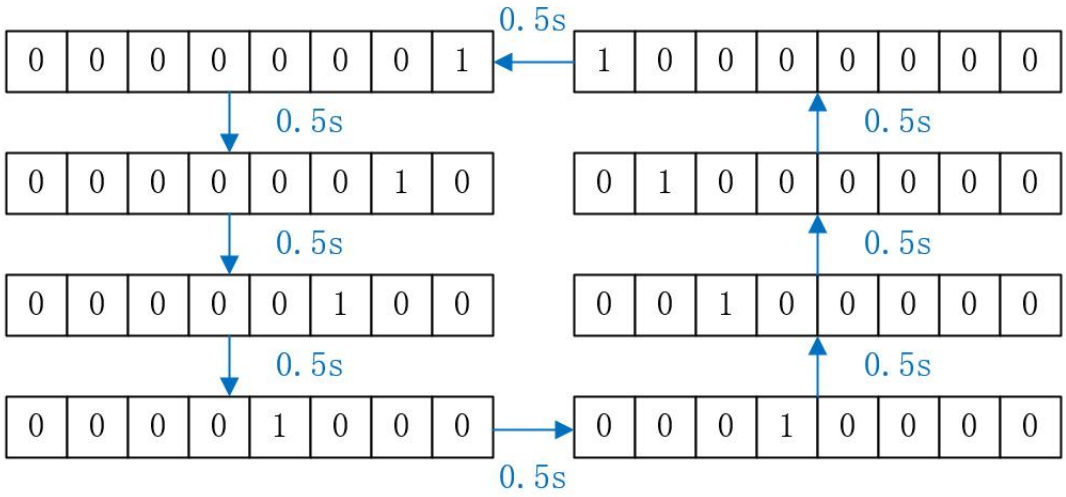


图 7.2-3

故只需要定义一个变量, 让其在时钟上升沿达到时就+1, 计数到 13500000-1 即可, 此时就是 0.5S。

7.3. 接口列表

端口	I/O	位宽	描述
CNT_MAX	**Parameter**	**26**	**计数的最大值, 修改定时的时间。**
clk	input	1	系统时钟, 27MHZ
key	input	1	按键信号
rst_n	input	1	复位信号, 低有效
led[7:0]	output	8	led 灯控制信号

7.4. 工程说明

7.5. 代码模块说明

```
1. `timescale 1ns/1ns
2. `define UD #1
3. module    key_led_test
4. #(
5.     parameter    CNT_MAX = 26'd13_500_000
6. )
7. (
8.     input            clk         ,
9.     input            rstn        ,
10.    input            key         ,
11.
12.    output   [7:0]   led
13.
14.);
15.
16.
17.//============================================================================
18.//reg and wire
19.
20.reg    [25:0]    led_light_cnt = 26'd0;
21.reg    [7:0]     led_status = 8'b0000_0001;
22.
23.//time counter
24.always@(posedge clk)
25.begin
26.    if(!rstn)
27.        led_light_cnt <= `UD 26'd0;
28.    else    if(led_light_cnt == CNT_MAX-1)
29.        led_light_cnt <= `UD 26'd0;
30.    else
31.        led_light_cnt <= `UD led_light_cnt + 26'd1;
32.end
33.
34.//led status change
35.always@(posedge clk)
```

```
36.begin
37.    if(!rstn)
38.        led_status <= `UD 8'b0000_0001;
39.    else    if(led_light_cnt == CNT_MAX-1 && key)
40.        led_status <= `UD {led_status[6:0],led_status[7]};
41.    else    if(led_light_cnt == CNT_MAX-1 && !key)
42.        led_status <= `UD {led_status[0],led_status[7:1]};
43.end
44.
45.assign led = led_status;
46.
47.
48.endmodule
```

代码的第 5 行所定义的参数 CNT_MAX 是用来设定计数的最大值, 默认是 13500000, 也就是定时 0.5S, 可以通过修改该值来改变定时的时间。

代码的 24-32 行, 用变量 led_light_cnt 实现了定时器的功能, 每到时钟的上升沿就让其加 1, 不断计数。由于从 0 开始计数, 所以计数到 CNT_MAX-1 即把它清 0。

代码的 35-43 行, 实现了 LED 灯的状态控制, led_status 是个 8bit 的变量, 当 key 没按下时, 也就是 key 的值为 1 时, 且 Led_light_cnt=CNT_MAX-1 时, 就让 led_status 向左移 1 位。实现 LED0->LED7。当按键按下时, key 为 0, 实现 LED7->LED0。

代码 45 行, 就通过组合逻辑, 将 led_status 的值赋值给 led。

7.6. 代码仿真

```
1.    `timescale    1ns/1ns
2.    module tb_key_led_test();
3.
4.    reg           clk         ;
5.    reg           rst_n       ;
6.    wire    [7:0] led         ;
7.    reg           key         ;
8.
9.    reg     [7:0] data        ;
10.
11.   initial
12.   begin
13.       rst_n    <=    0;
14.       clk      <=    0;
15.       key      <=    0;
16.       #20
```

```
17.         rst_n     <=    1;
18.         key       <=    1;
19.         #2000
20.         key       <=    0;
21.         #2000
22.         $display("I am stop");
23.         $stop;
24.     end
25.
26.     always#10 clk = ~clk;//20ns  50MHZ
27.
28.     key_led_test#(
29.         .CNT_MAX (10 )
30.     )u_key_led_test(
31.         .clk   ( clk   ),
32.         .rstn  ( rst_n ),
33.         .key   ( key   ),
34.         .led   ( led   )
35.     );
36.
37.     initial
38.     begin
39.         $monitor("led:%b",led);
40.     end
41.
42.
43.     always@(posedge clk or negedge rst_n)    begin
44.         if(!rst_n)
45.             data    <=    8'd0;
46.         else
47.         begin
48.             data    <=    {$random}%256;
49.             $display("Now data is %d",data);
50.         end
51.     end
52.
53.     endmodule
```

该 testbench 部分代码是用于测试一些仿真函数使用,在前面的 Modelsim 的使用章节已经做了介绍,所以本次只关注代码的 28-35 行, 例化了我们的流水灯模块。可以看到代码的 29 行, CNT_MAX 传入的参数给的是

10, 这是为了减少仿真的时间, 如果还是 13500000 的话, 我们需要仿真 0.5s 才能看到结果, 这非常的久。

代码的 11-26 行依旧是对复位和时钟赋初值, 延时 20ns 后, 复位结束。时钟依然是每隔 10ns 取反一次, 来生成 50MHZ 的时钟。

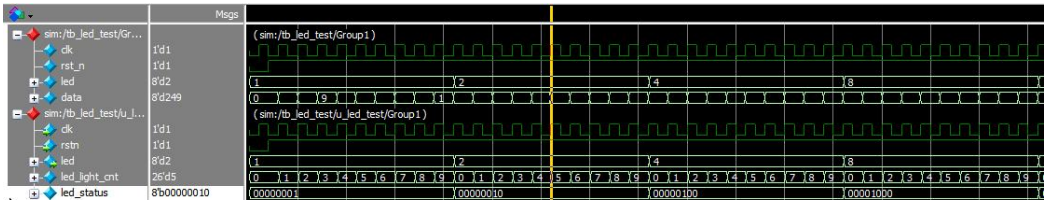


图 7.6-1

图 7.6-1 为仿真结果, 可以看到 led_light_cnt 每次计数 10 个数据后, led_status 会向左移一位, 符合实验结果。

7. 7. 实验步骤

这里将会详细介绍从新建工程到下载程序的具体步骤, 后续的工程将不再详细解释。

7.7.1.打开 PDS 软件, 创建工程

Step1: 打开 PDS 软件, 点击 NEW Project, 然后对其设置完成新建工程;

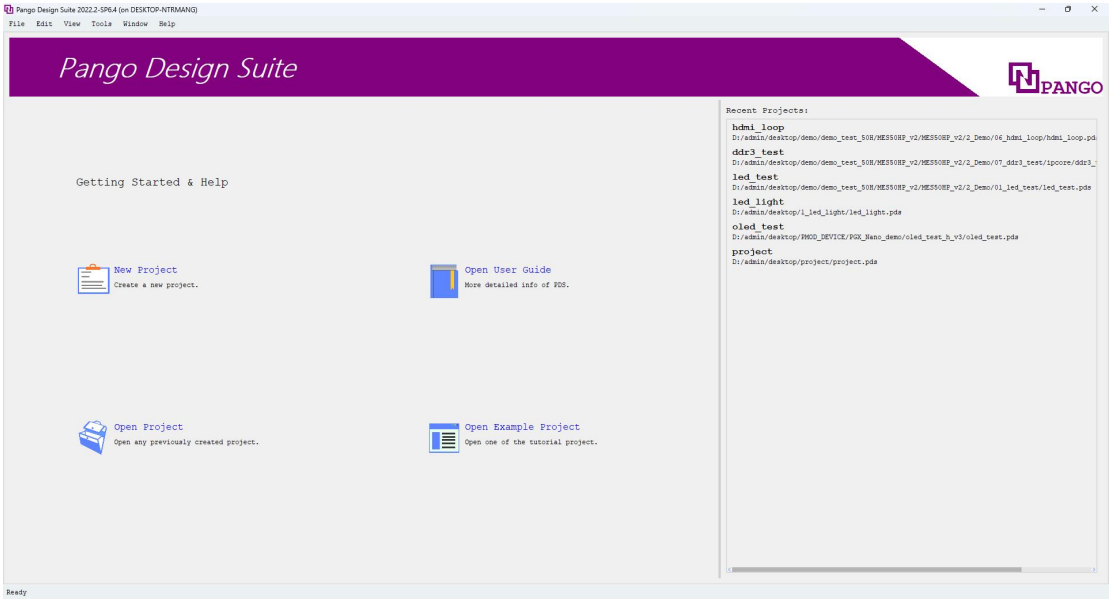


图 7.7-1

Step2: 单击 NEXT;

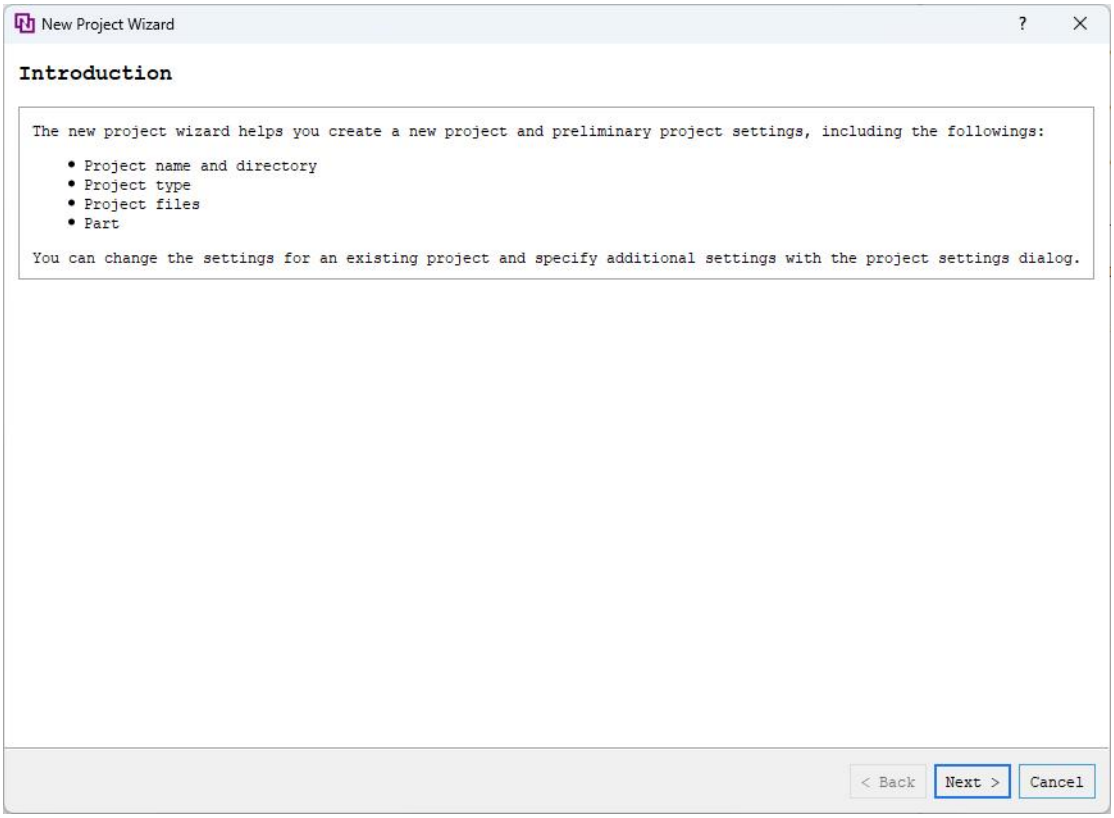


图 7.7-2

Step3: 创建名为 led_water 的工程到对应的文件目录, 之后单击 Next;

新建工程大致包括设置工程名和工程路径、工程类型、工程文件及器件信息。

【Project Name】是工程文件名称, 默认为 project。(只允许字母、数字、下划线 (_)、杠 (-)、点 (.))。

【Project Location】用于选择新工程的工作路径, 文件夹名只允许字母、数字、下划线 (_)、杠 (-)、点 (.)、@、~、,、、+、=、#、空格 ( ), 但空格不能出现在路径名首尾, 即工程文件放置的路径。

【Create Preject Subdirectory】将工程文件名作为工作目录的一部分。

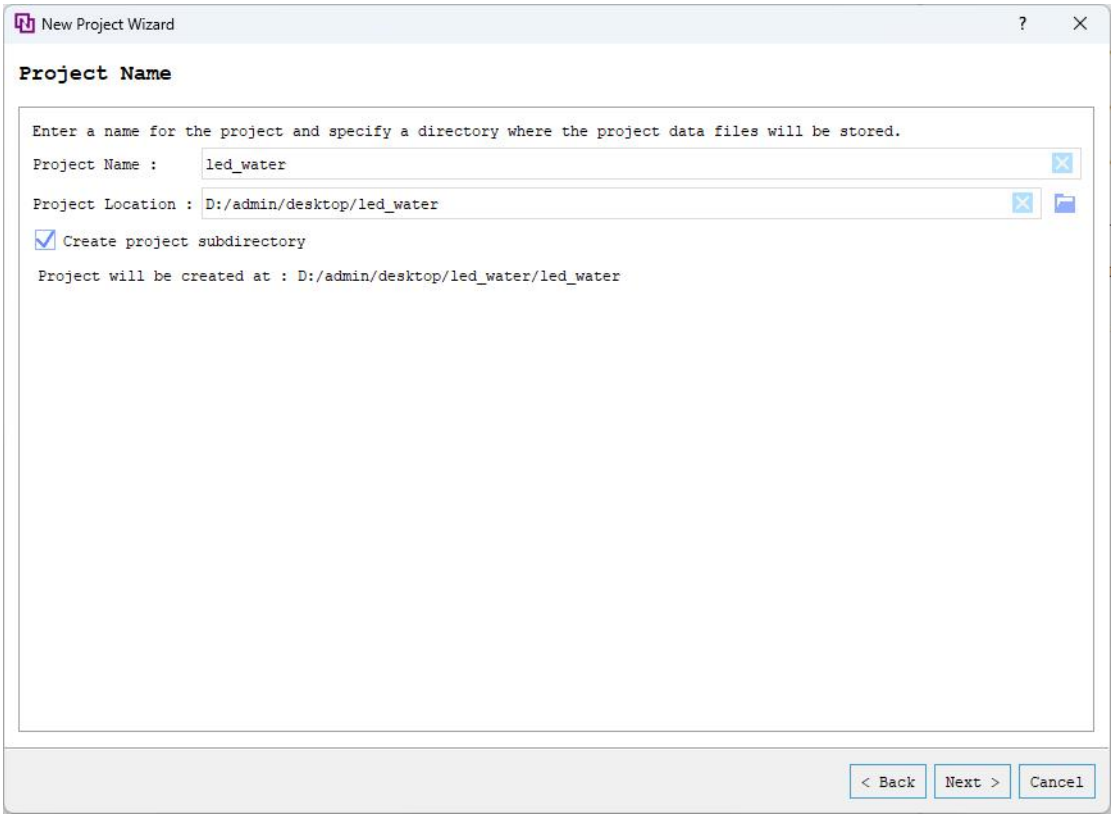


图 7.7-3

Step4: 选择 RTL project, 点击 Next;

【RTL Project】用于创建 RTL 工程。新建的工程可以执行 synthesize, device map, place& route, report timing, report power, generate netlist 及 generate bitstream 等。

【Post-Synthesize Project】用于创建综合后工程。新建的工程可以执行 device map,place& route,report timing, report power, generate netlist 及 generate bitstream 等。

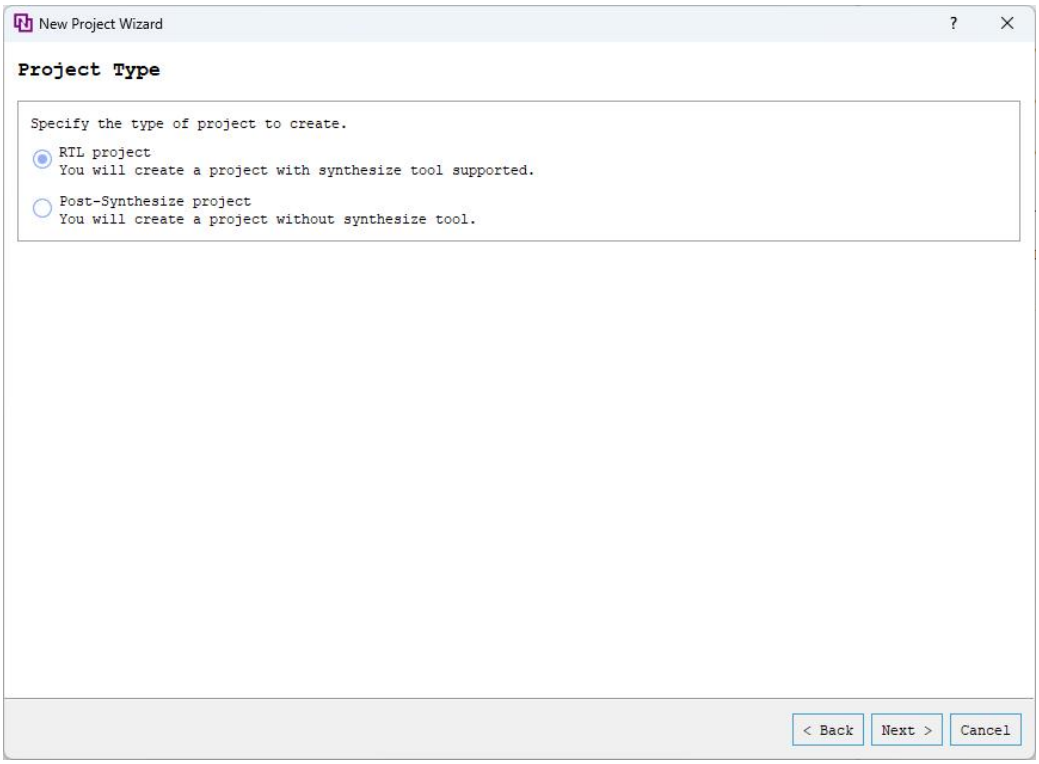


图 7.7-4

Step5: 单击 Next;

该界面可以 Add Files 和 Add Directories 来添加 rtl 源文件及新建 rtl 源文件, 以及调整 rtl 文件编译顺序,Add Files 添加选中的文件,Add Directories 添加选中的文件夹下所有合适的文件,若勾选了下方的 Add source from subdirecotires 则添加所有的子目录下合适的文件, 也可直接 NEXT 跳过添加文件。

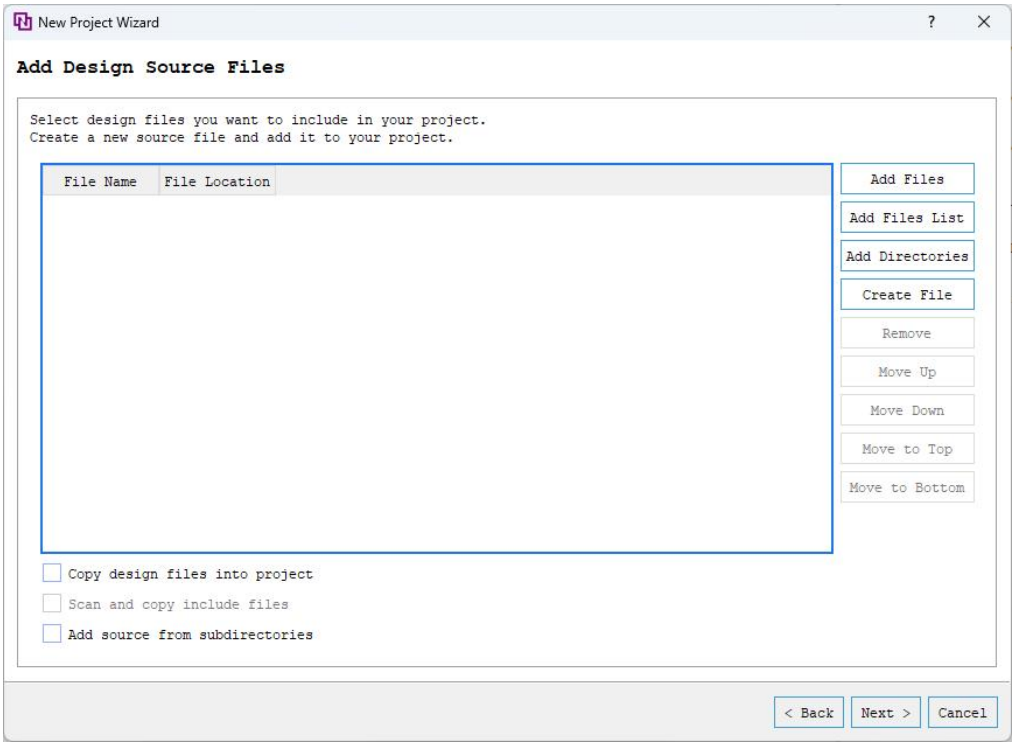


图 7.7-5

Step6: 单击 Next;

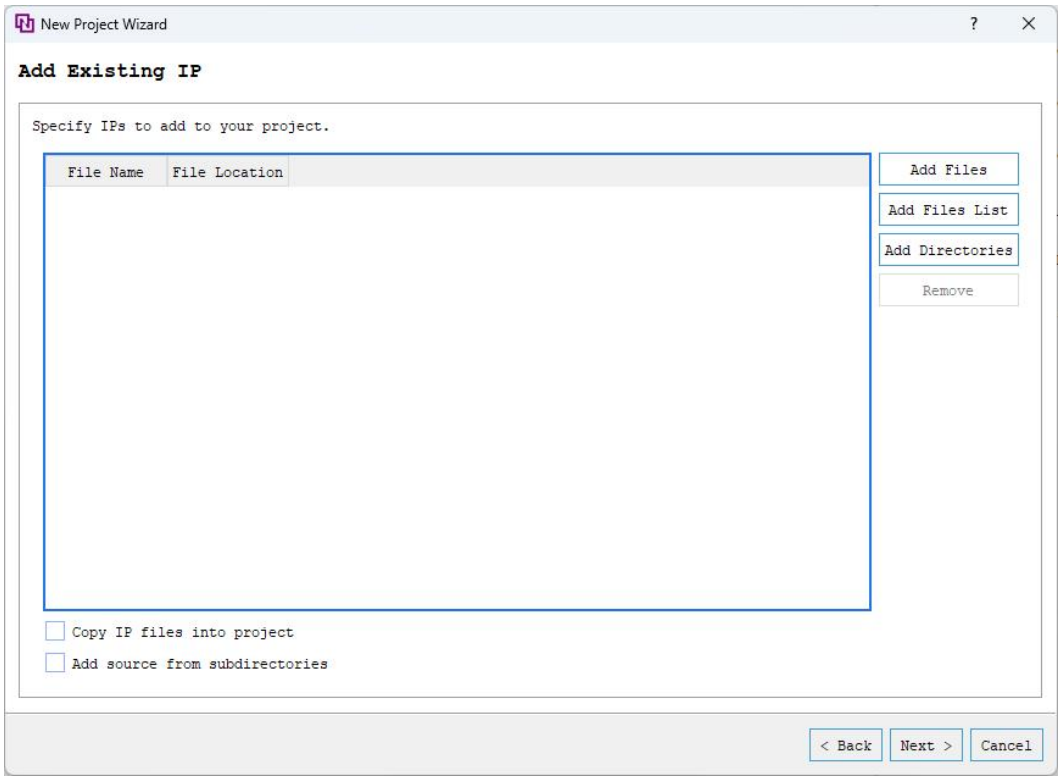


图 7.7-6

Step7: 单击 Next;

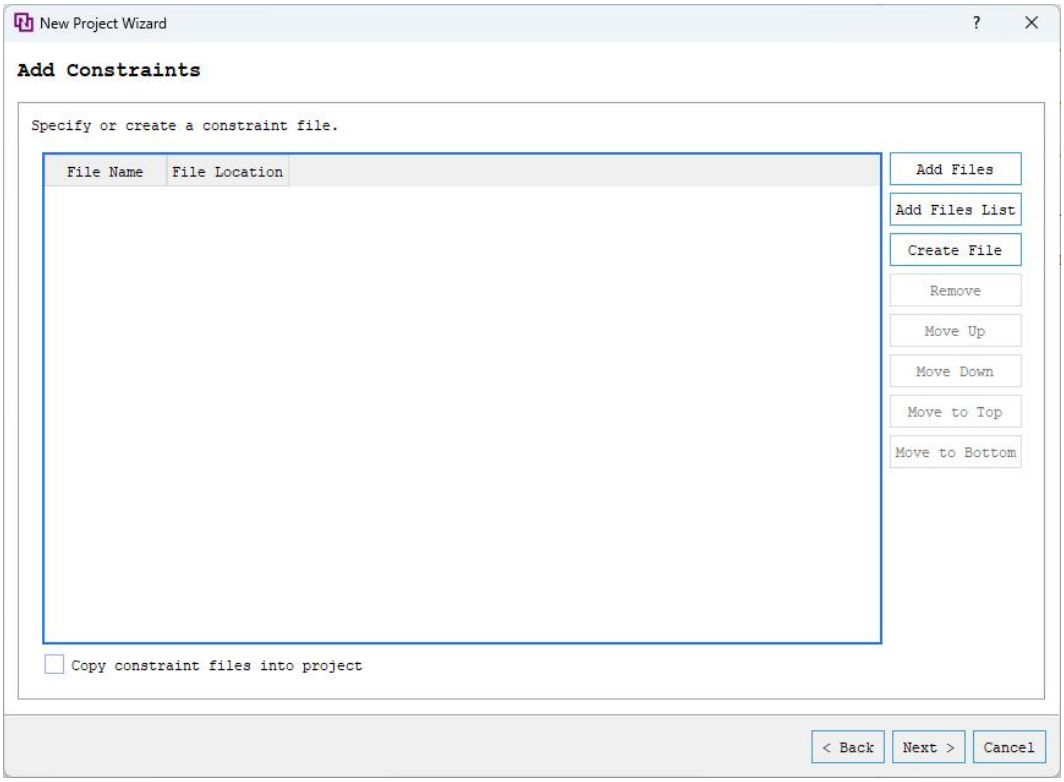


图 7.7-7

Step8: 选择器件系列、型号、封装、速率、综合工具, 之后单击 Next;

synthesize tool 中可以选择综合工具为 Synplify Pro 或 ADS, 在实验中使用 ADS 综合工具。

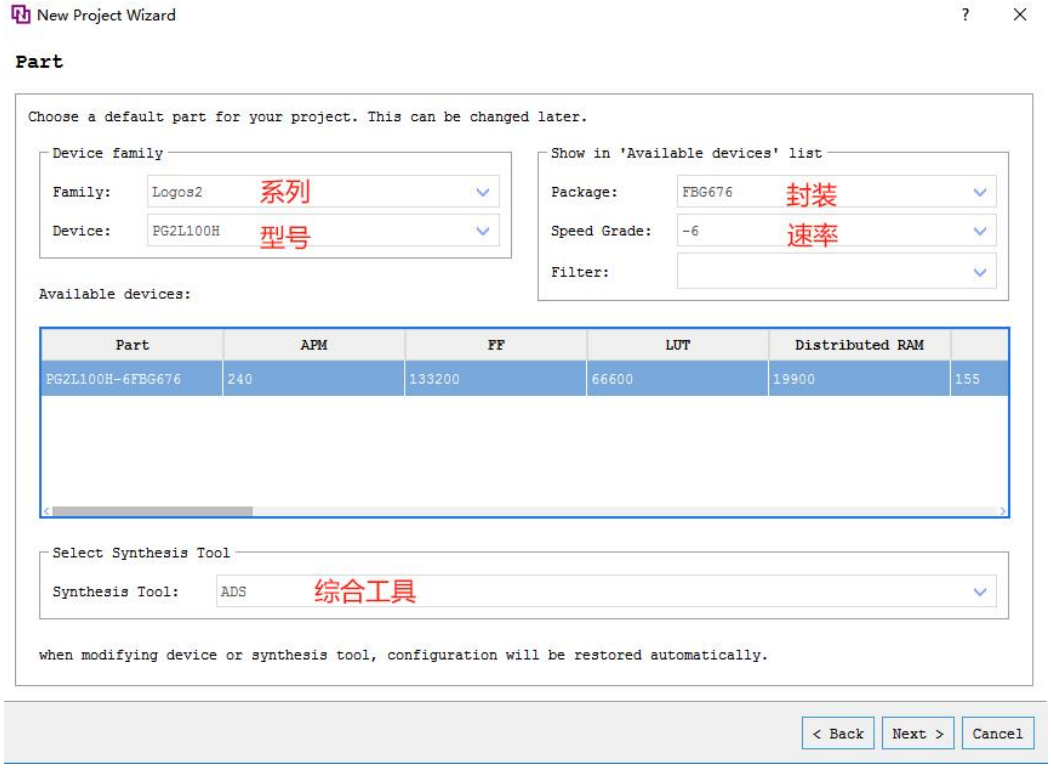


图 7.7-8

Step9: 在 summary 单击 Finish, 完成工程的创建;

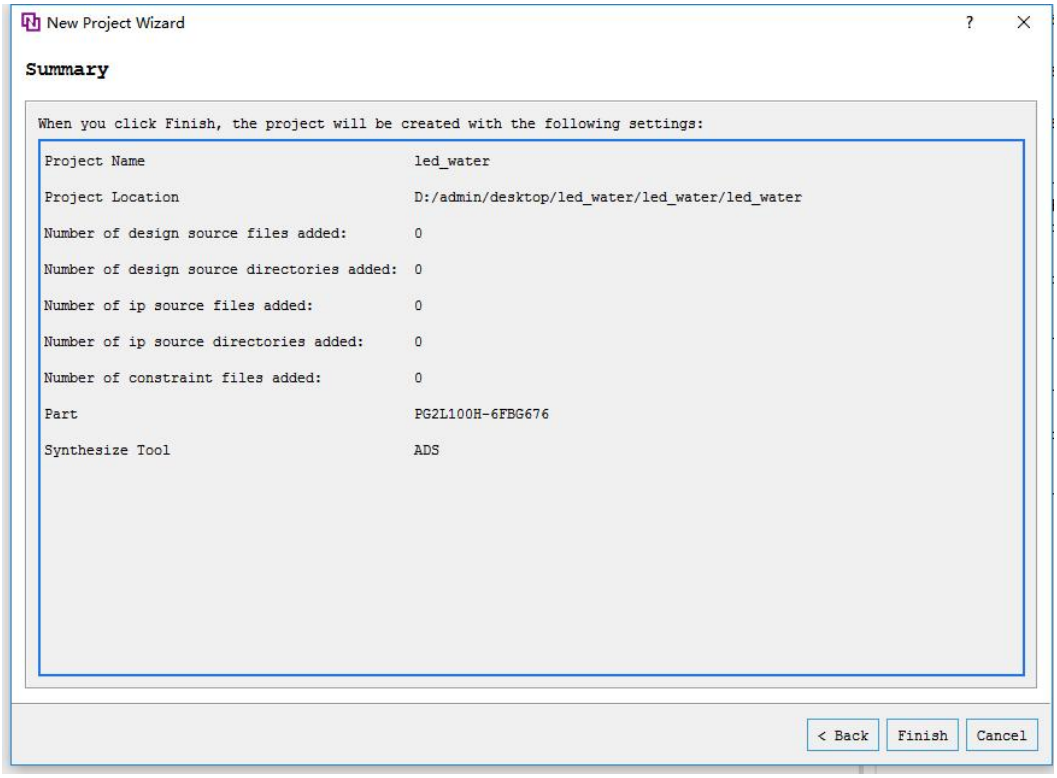


图 7.7-9

7.7.2.添加设计文件

PDS 软件界面如下图:

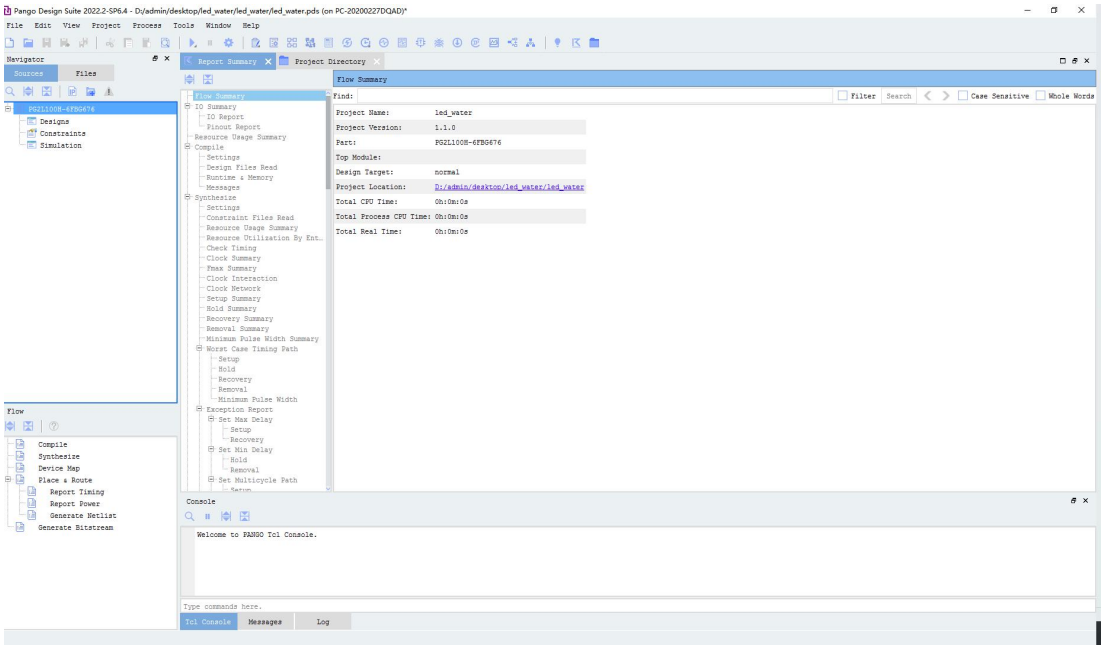


图 7.7-10

双击 Designs, 将前面设计的 module 新建到文件中, 或者将前面编辑好的 verilog 文件添加到工程中;

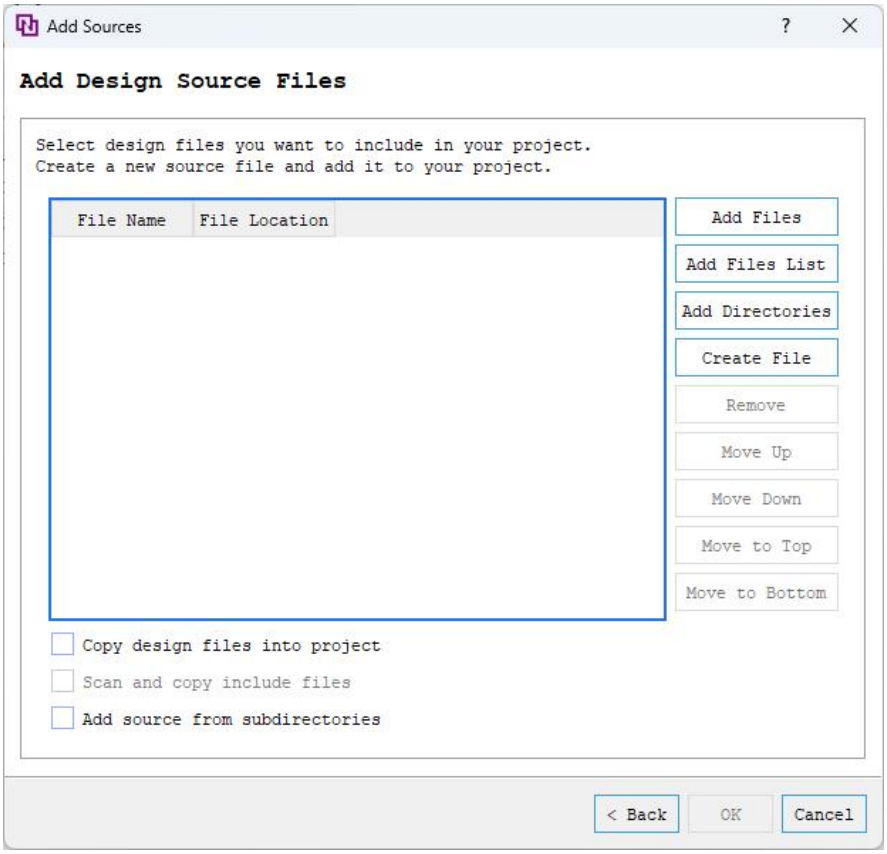


图 7.7-11

添加文件到工程:

在窗口中点击 Add Files, 选择添加文件到工程;

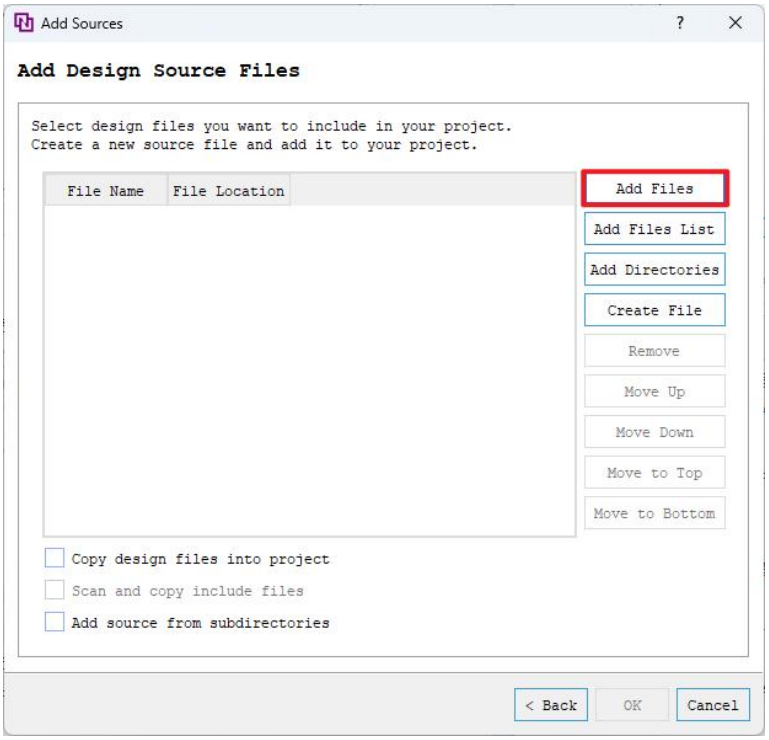


图 7.7-12

新建文件到工程:

1)在窗口中点击 Create File;

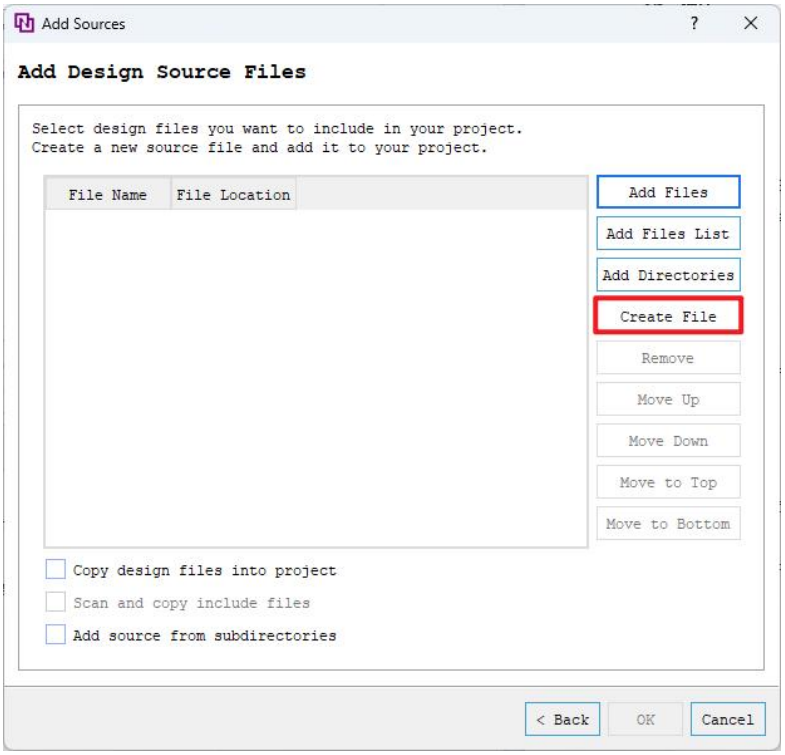


图 7.7-13

2)选择 Verilog Design File, 文件名和 module 名一致, 默认路径, 点击 OK;

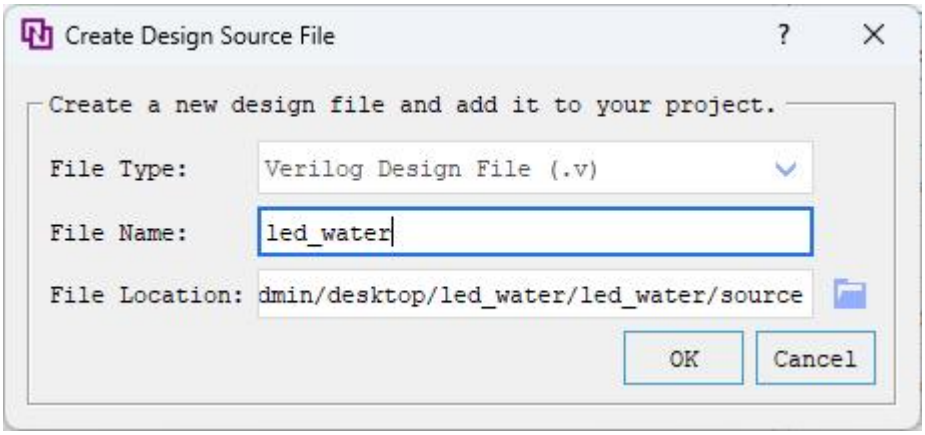


图 7.7-14

点击 OK;

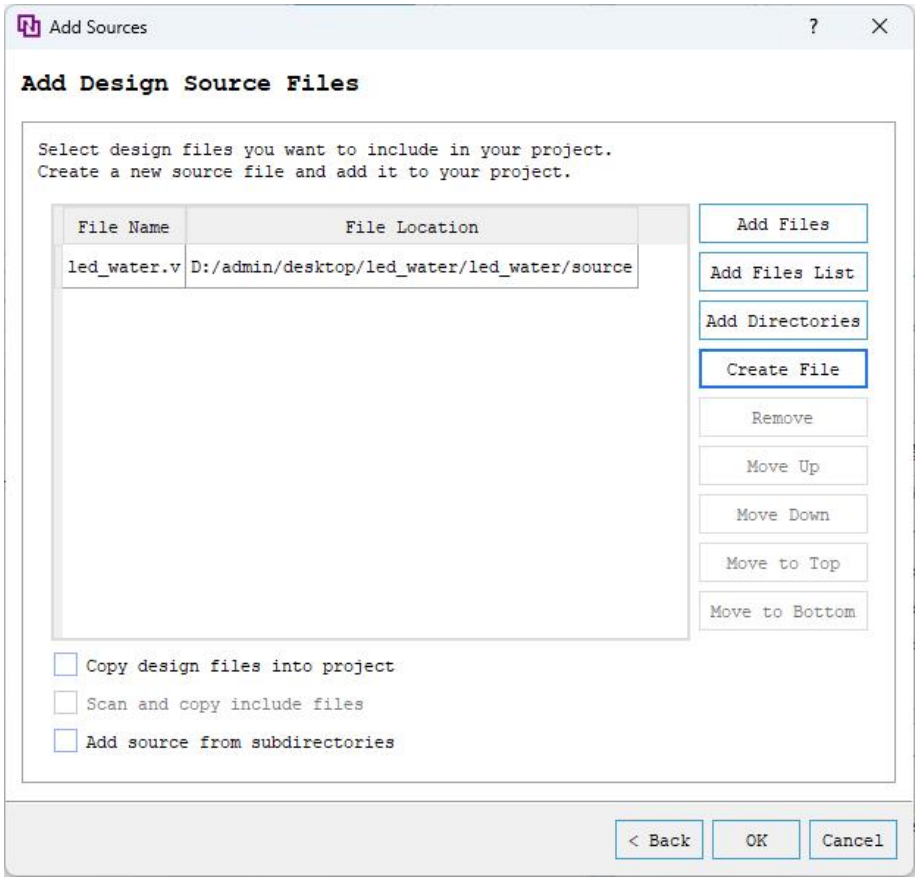


图 7.7-15

4)点击 Cancel;

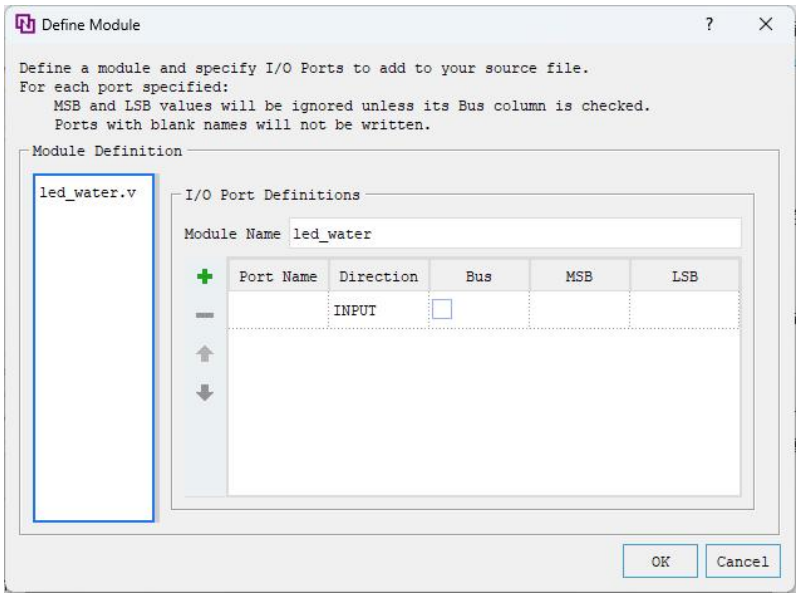


图 7.7-16

5)默认打开新建文件, 将前面设计的 module 复制进去;

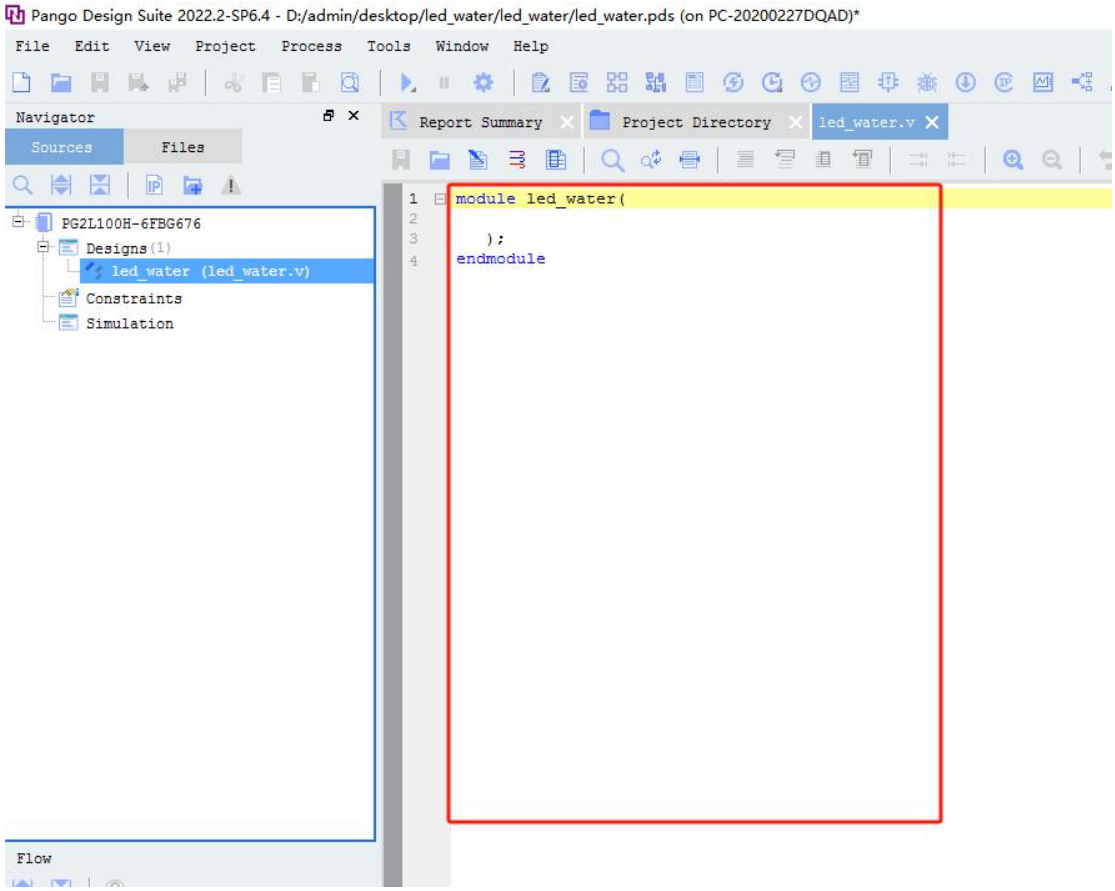


图 7.7-17

点击保存, 新建文件完成。

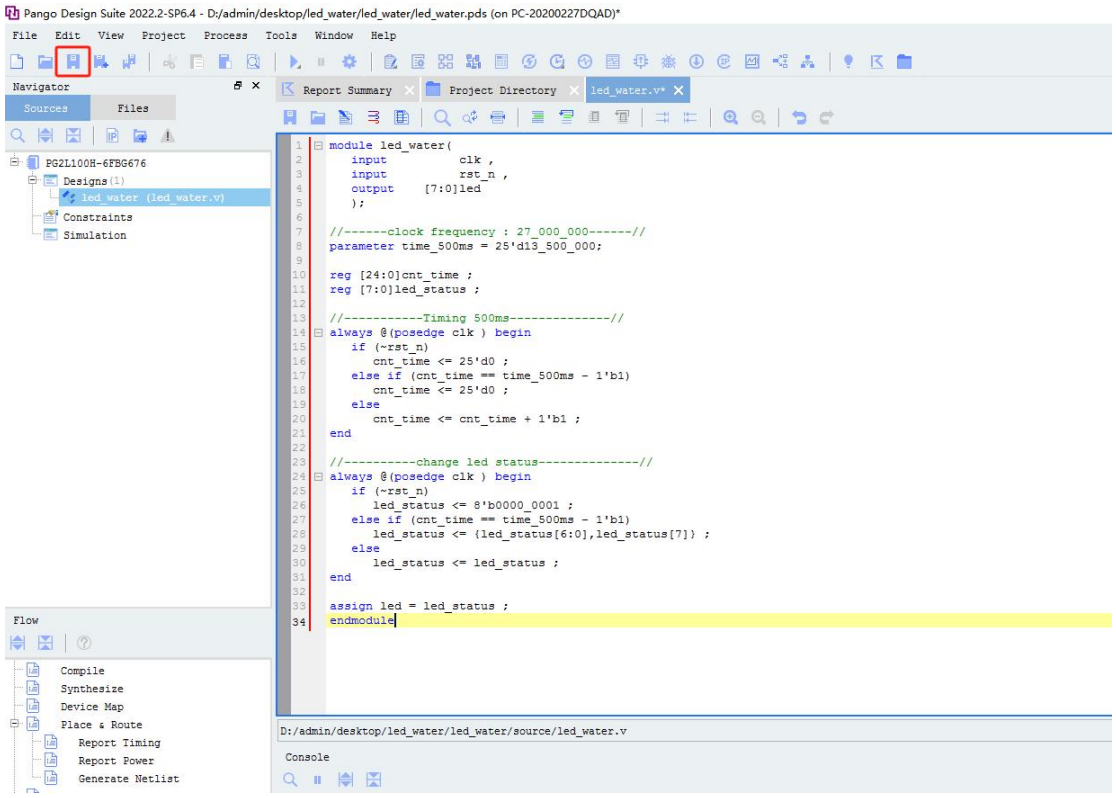


图 7.7-18

7.7.3.编译

可采用以下方式运行 Compile 流程:

(1)双击 Flow 中的 Compile 进行综合;

(2)右击 Compile 点击 Run 进行综合;

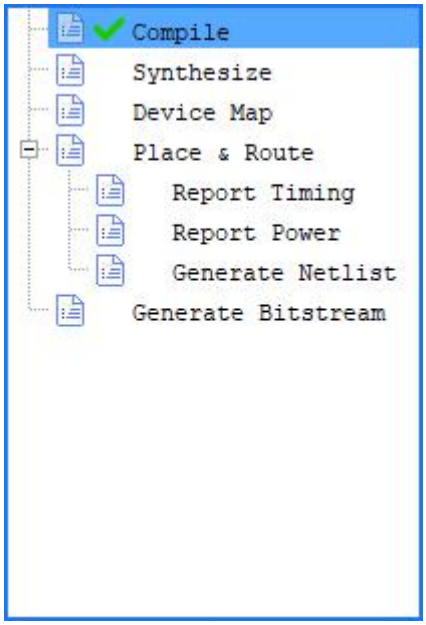


图 7.7-19

7.7.4.工程约束

点击 Tools 选择 User Constraint Editor(Timing and Logic)或者点击工具栏图标 , User Constraint Editor(Timing and Logic) 选择 Pre Synthesize UCE, 如下图所示:

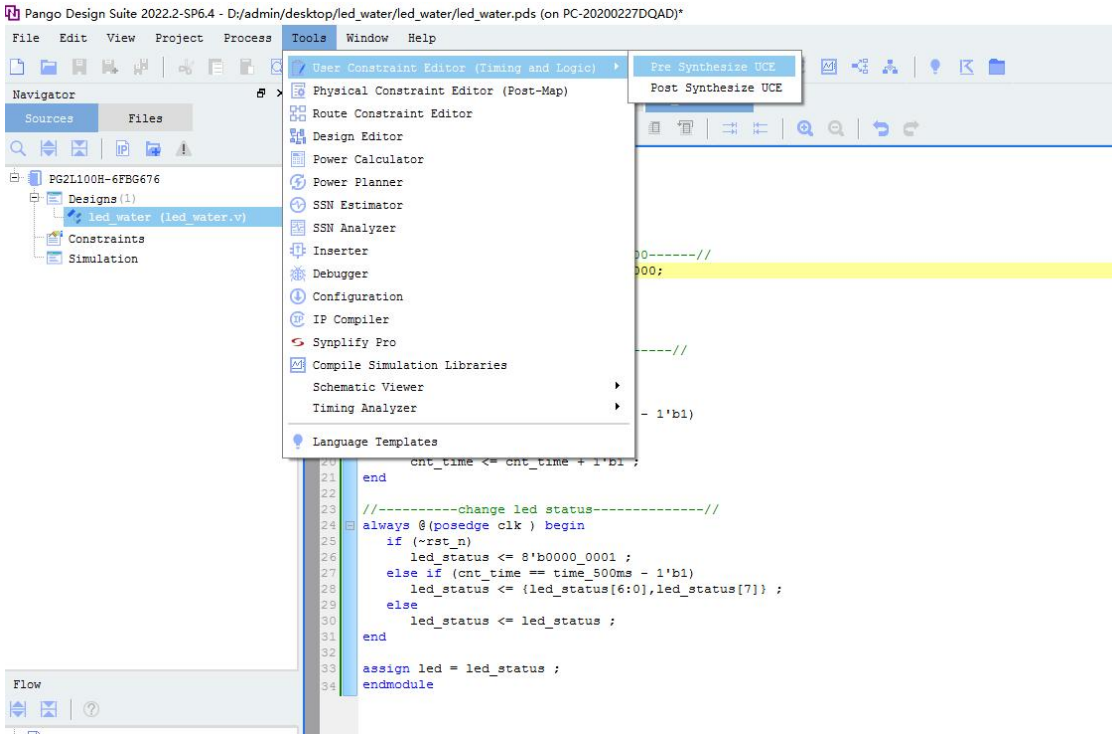


图 7.7-20

Tools 下的 User Constraint Editor(Timing and Logic)。

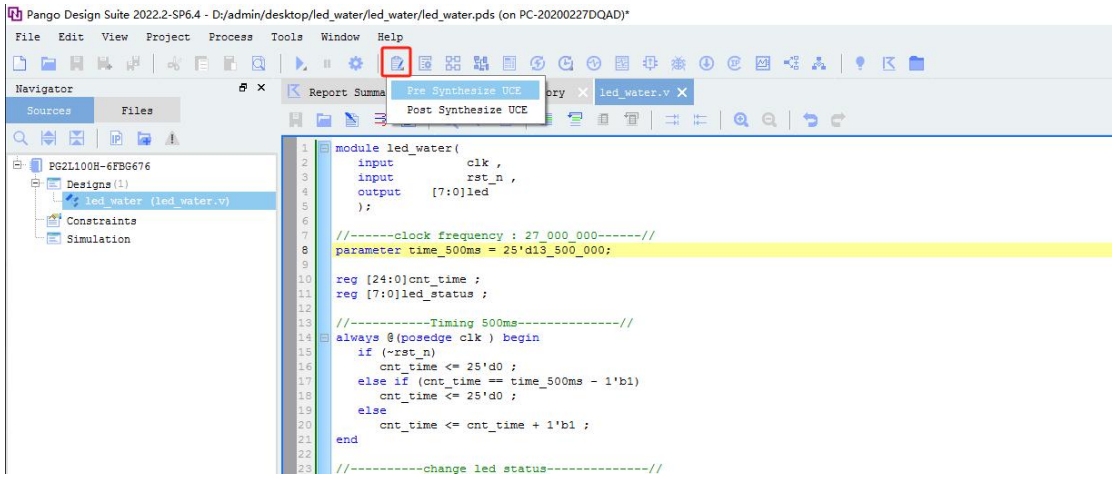


图 7.7-21

工具栏 User Constraint Editor(Timing and Logic)图标。

7.7.4.1. 时钟约束

打开 UCE 后, 选择 Timing Constraints 后选择 Create Clock 添加基准时钟, 基准时钟一般是通过输入

port 输入用户所使用的板上时钟。

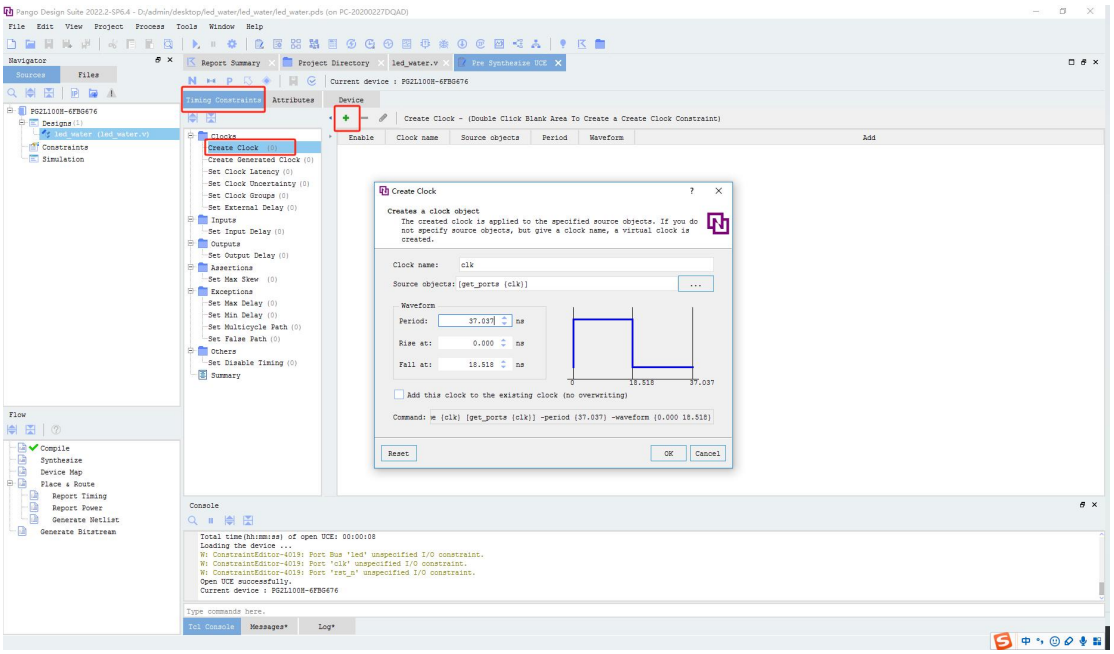


图 7.7-22

在弹窗中对时钟命名, 关联时钟管脚, 添加时钟参数, 点击 OK 会创建一条时钟约束, Reset 重置该页面。创建完成如下图所示:

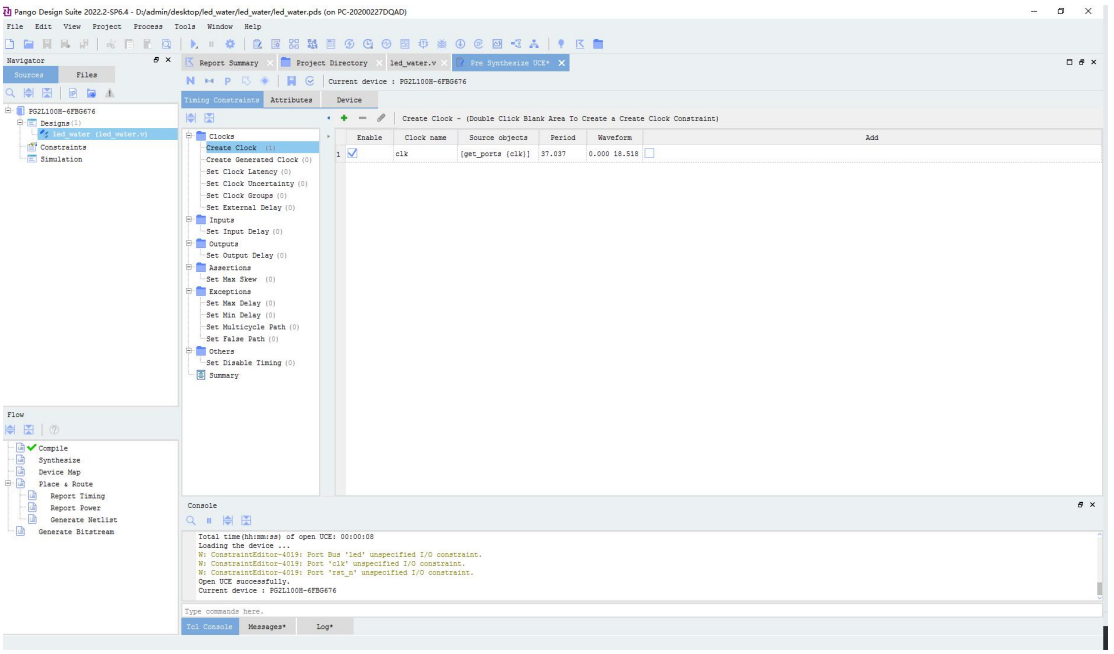


图 7.7-23

7.7.4.2. 物理约束

打开 UCE 后, 选择 Device 后选择 I/O, 根据原理图编辑 IO 的分配。

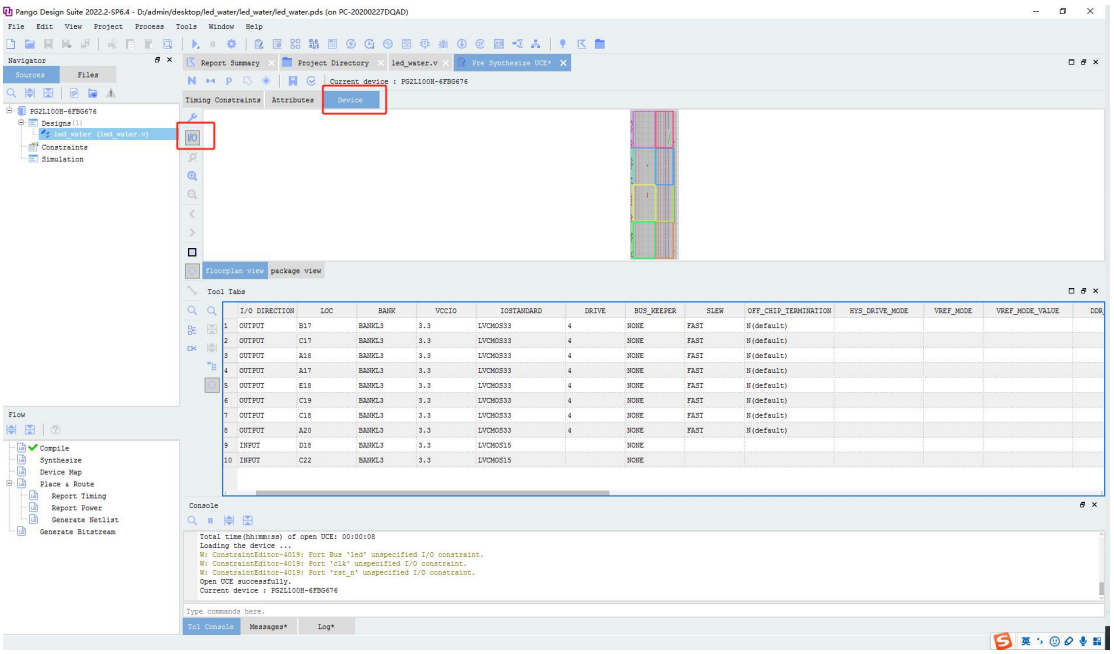


图 7.7-24

按照原理图编辑好 IO 分配后, 点击保存, 会生成.fdc 文件, 完成约束。

7.7.5.综合

运行 Synthesize 流程有以下四种方式可以实现:

(1)双击 Flow 中的 Synthesize 进行综合;

(2)右击 Synthesize 点击 Run 进行综合;

完成 Synthesize 操作后, 会看到下图所示:

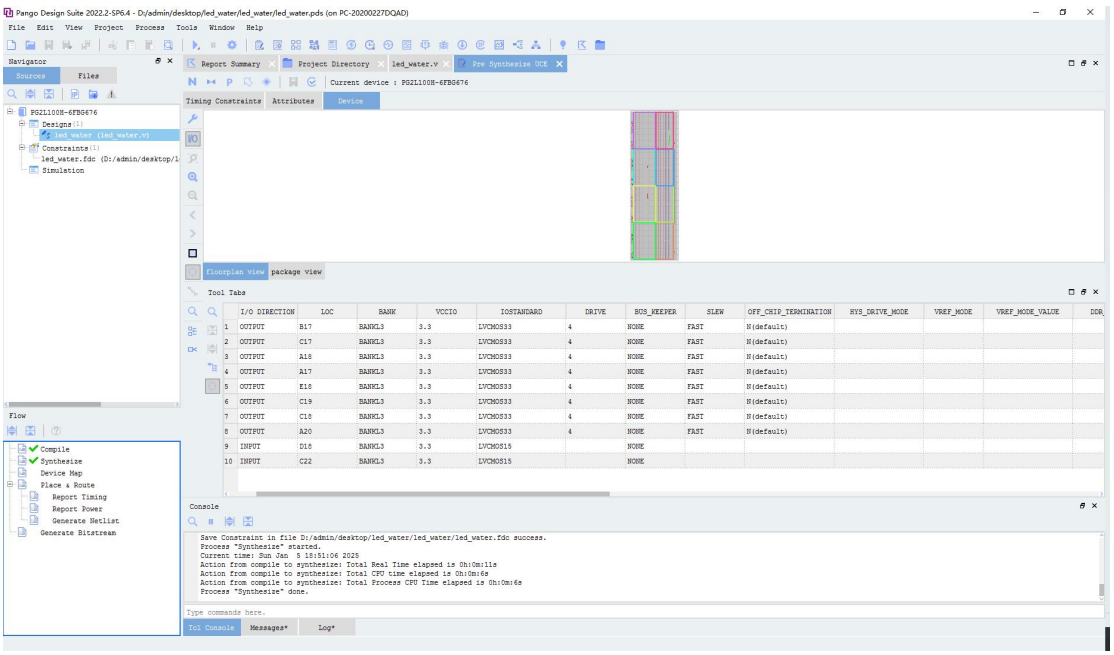


图 7.7-25

7.7.6.Device Map

Device Map 的主要作用是将设计映射到具体型号的子单元上（LUT、FF、Carry 等）。运行 Device Map 流程有以下方式可以实现:

（1）直接双击 Device Map;

（2）右击 Device Map 点击 Run;

完成 Device Map 操作后, 会看到下图所示:

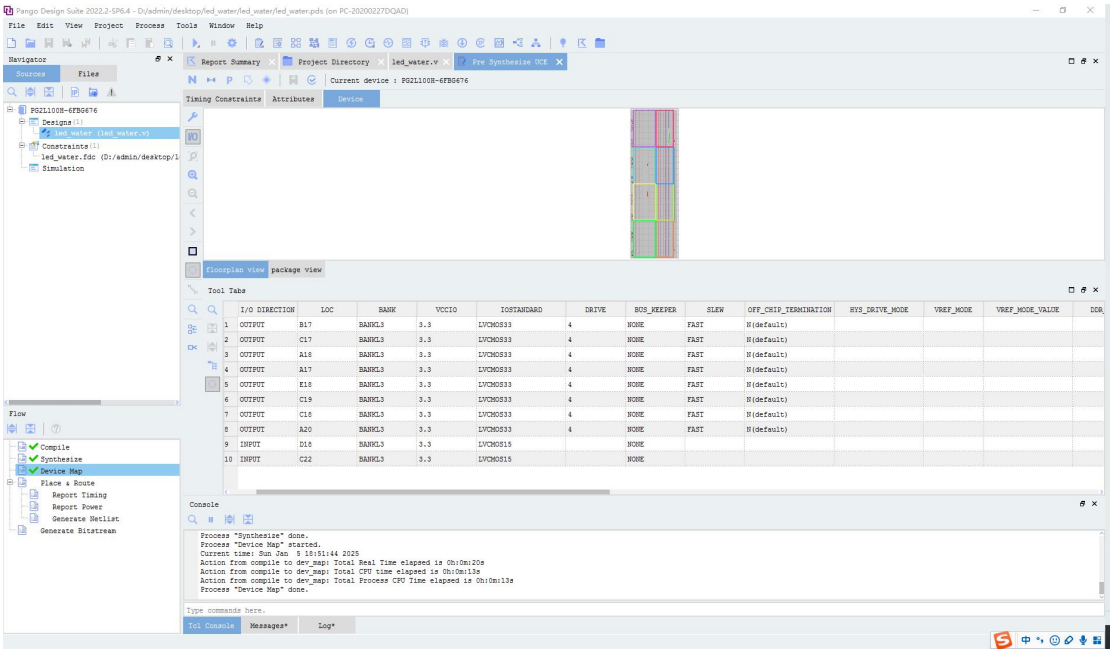


图 7.7-26

7.7.7.Place & Route

布局布线（Place & Route）根据用户约束和物理约束, 对设计模块进行实际的布局及布线。运行 Place & Route 流程有以下方式可以实现:

（1）直接双击 Place & Route;

（2）右击 Place & Route 点击 Run;

完成 Device Map 操作后, 会看到下图所示:

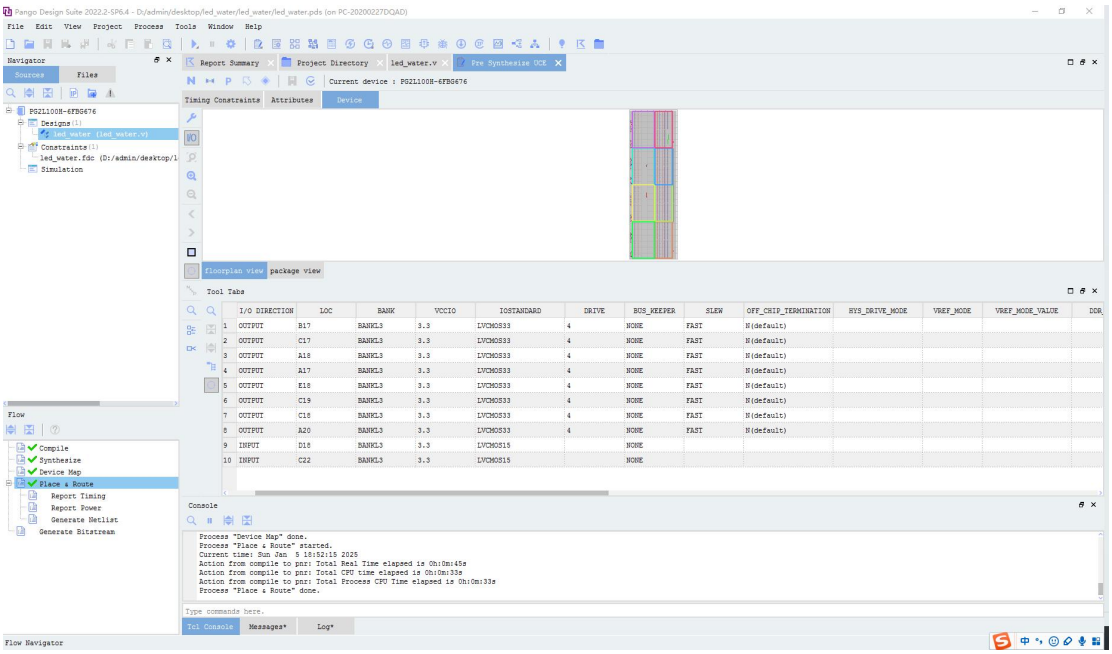


图 7.7-27

7.7.8.Generate Bitstream

Generate Bitstream 生成二进制位流文件。运行 Generate Bitstream 流程有以下方式可以实现:

(1)直接双击 Generate Bitstream;

(2)右击 Generate Bitstream 点击 Run;

完成以上操作, 将会产生位流文件。运行 Generate Bitstream, 可以看到界面如下图所示:

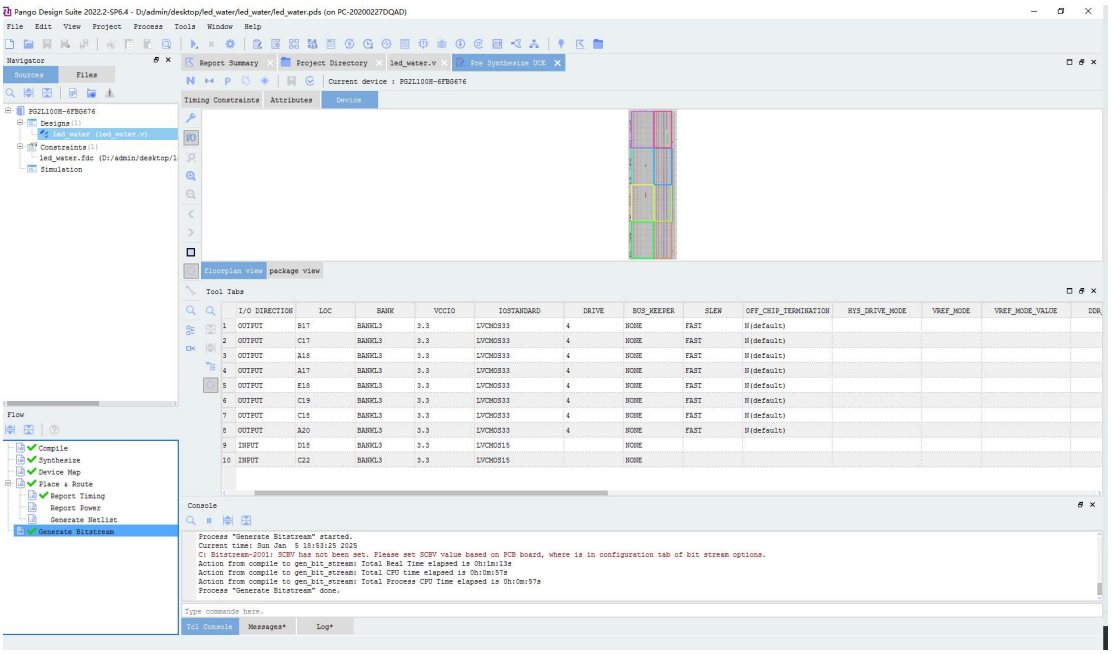


图 7.7-28

7.7.9.下载生成的位流文件

点击 Tools 选择 Configuration 或者点击工具栏图标 Configuration, 如下图所示:

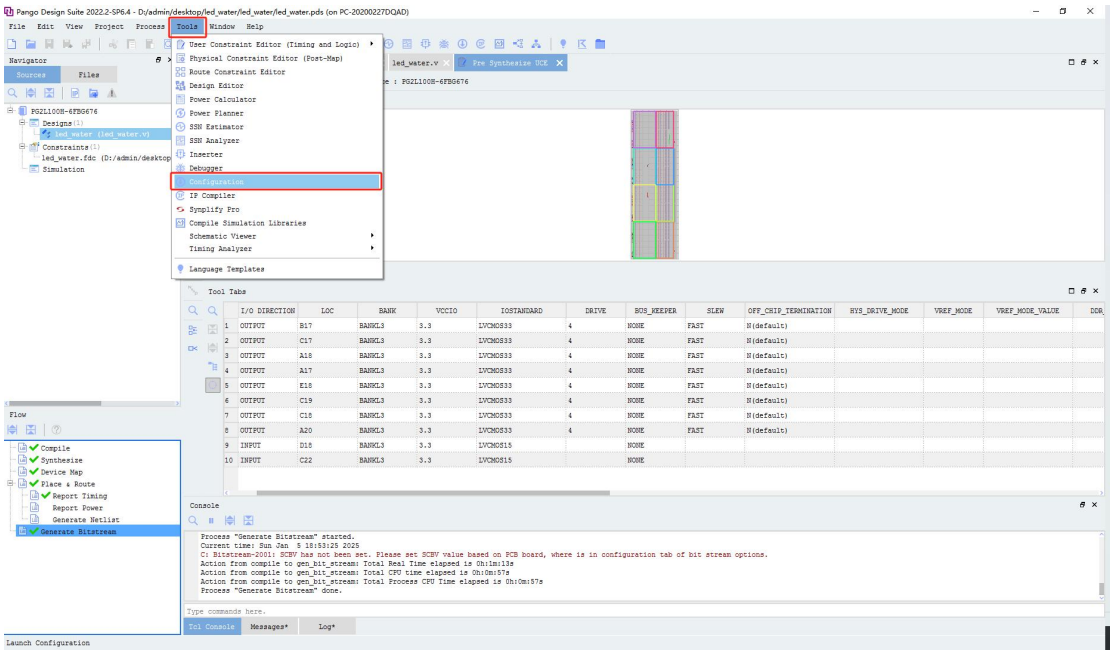


图 7.7-29

上图为 Tools 下的 Configuration 选项。

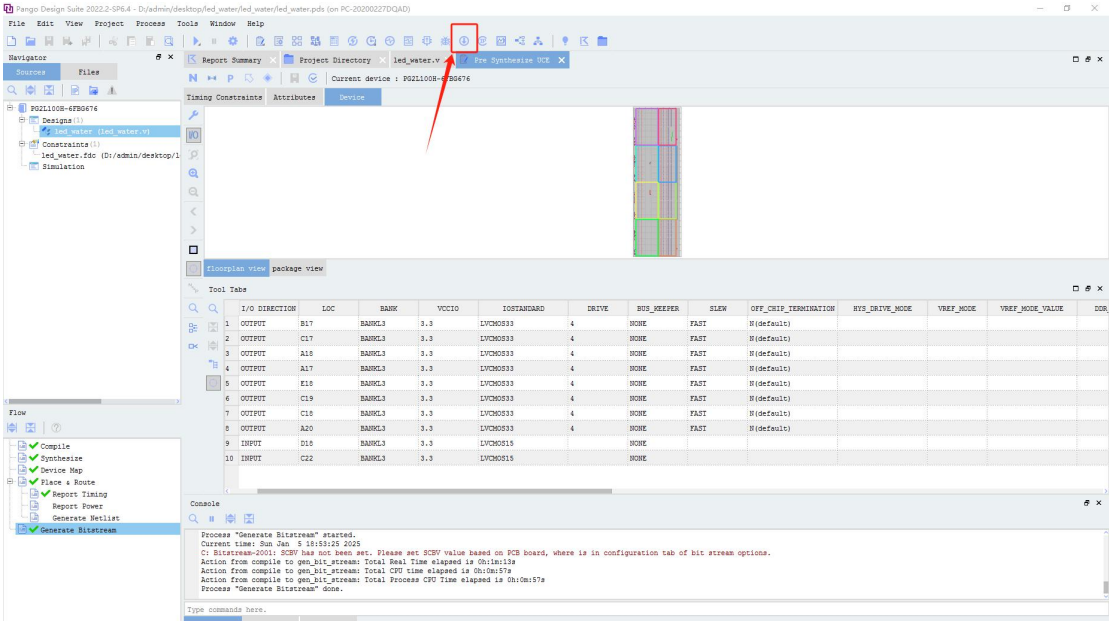


图 7.7-30

上图为工具栏 Configuration 图标。

打开 Configuration 后直接选择 Scan Device 直接进行扫描 Jtag 链操作，初始化链成功，会将链上扫描到的所有器件显示于工作区内，并在器件属性窗口显示当前器件的器件信息，并弹出对话显示能够为器件添加的配置文件；

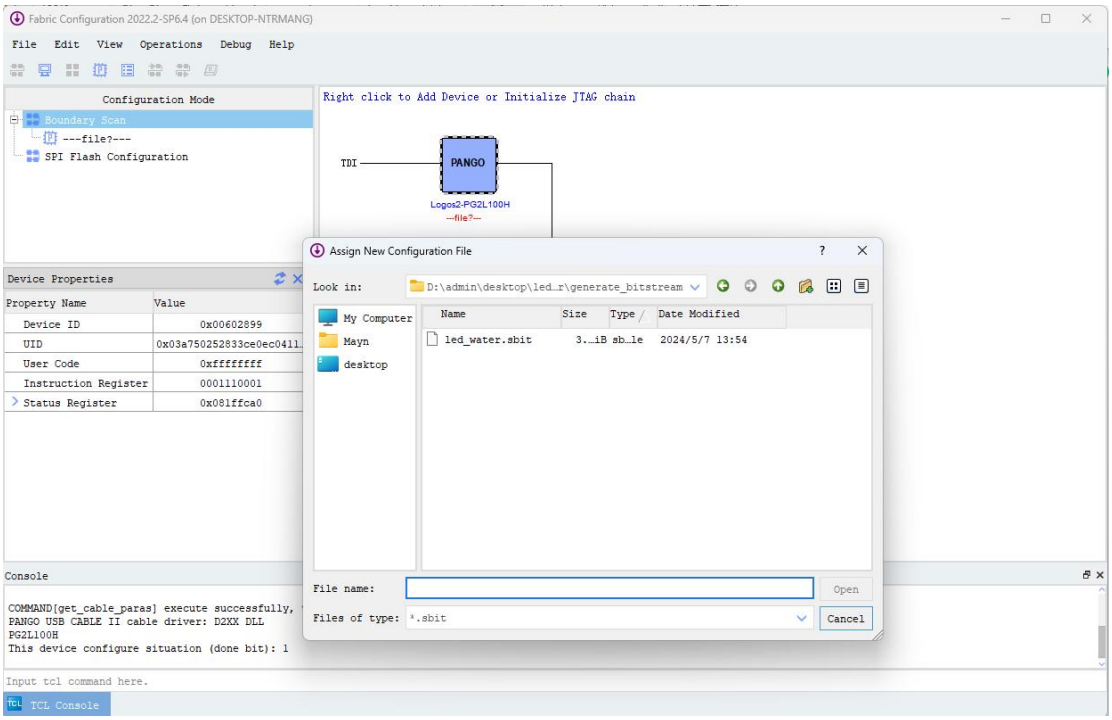


图 7.7-31

在对话框中选择位流文件, 添加该配置文件, 提示所载入文件的绝对路径并在信息栏中显示, 右键后点击 program 下载位流文件如下图所示:

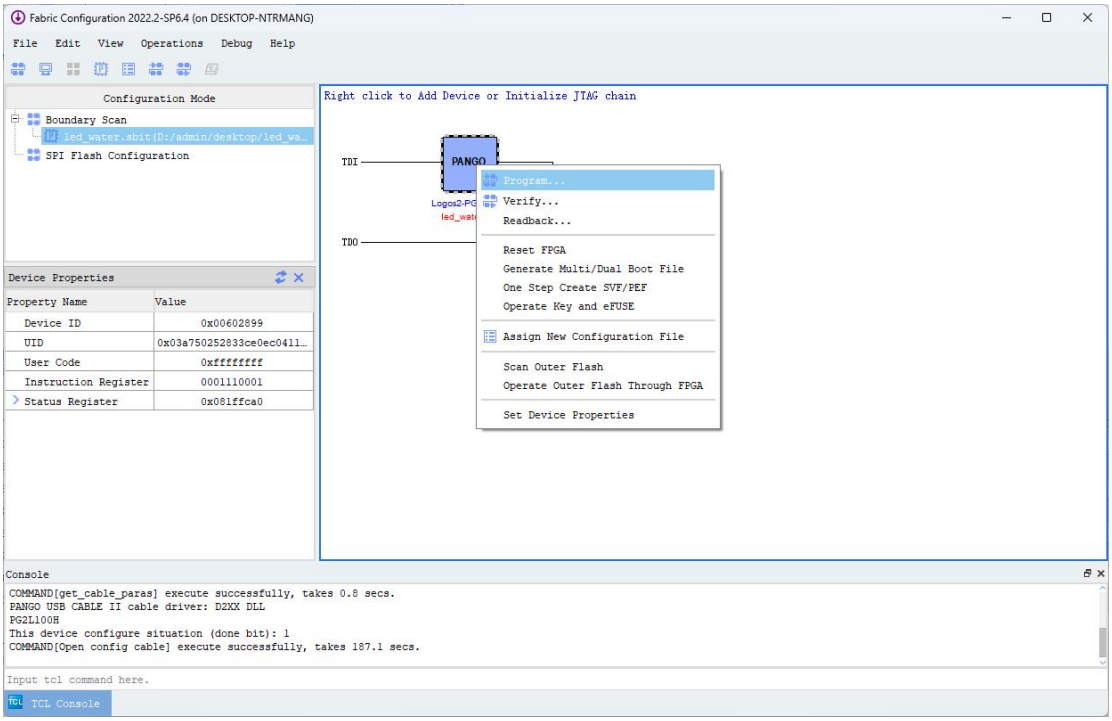


图 7.7-32

下载位流文件成功如下图所示:

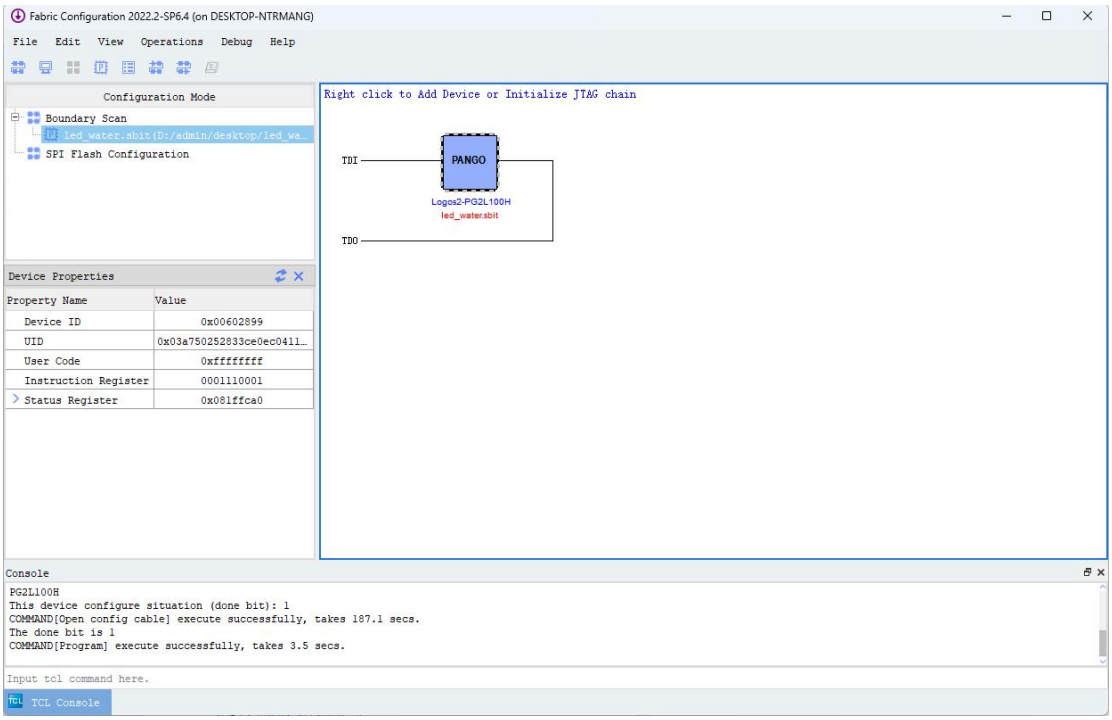


图 7.7-33

PG2L100H 板卡为 PG2L100H 的 FPGA 配置了一个外部 flash, 其中, 若需要将程序固化到板卡上需要将尾流文件转化为.sfc 文件。

首先点击 Configuration 页面的 Operations 选项的 Covert File 选项;

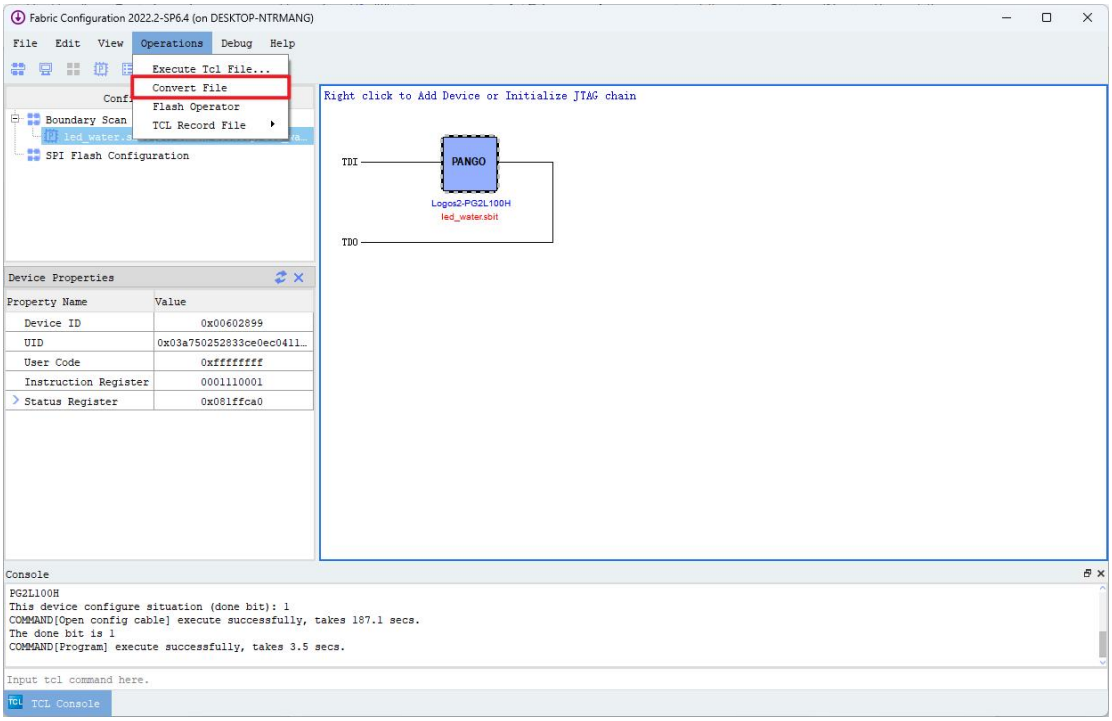


图 7.7-34

点击后会出现如下画面, 在 Generate Flash Programing File 页面选择对应的 Flash 器件的厂商名、型号、再在 BitStreamFile 位置选择位流文件的路径, 点击 OK。(若使用的 flash 器件不在可选的 flash 列表中, 需手

动添加对应 flash 型号, 操作步骤请参考开发板下载与固化相说明);

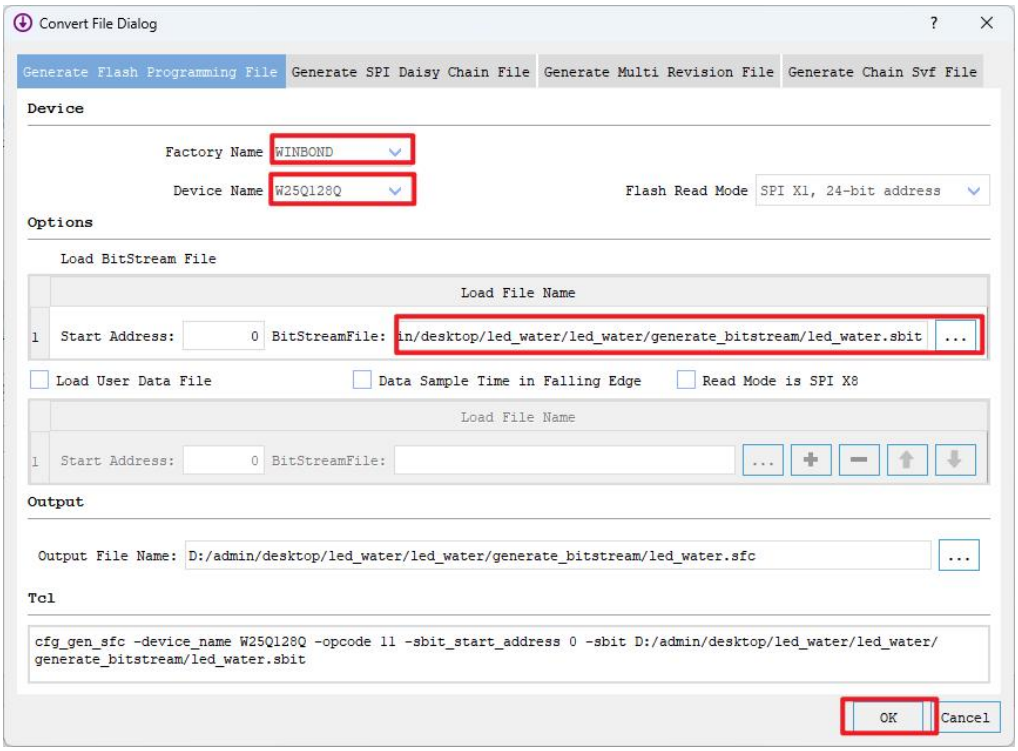


图 7.7-35

转化.sfc 文件成功后, 页面会如下图所示, 点击 OK;

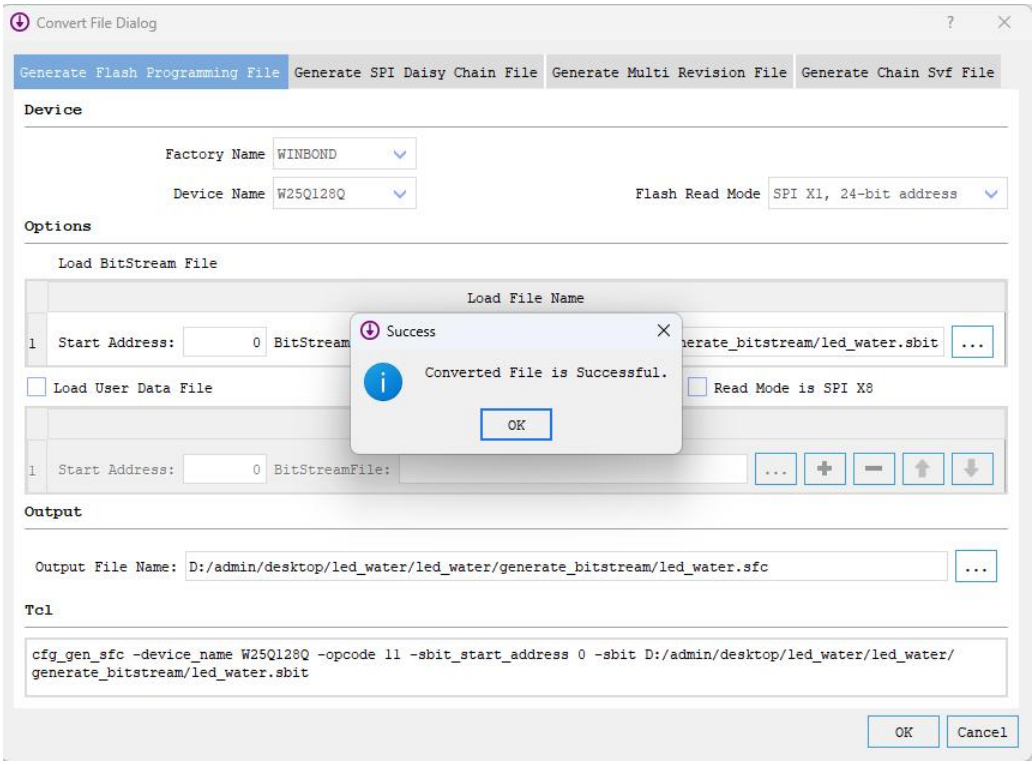


图 7.7-36

用户可通过右键下图位置, 点击 Scan Outer Flash;

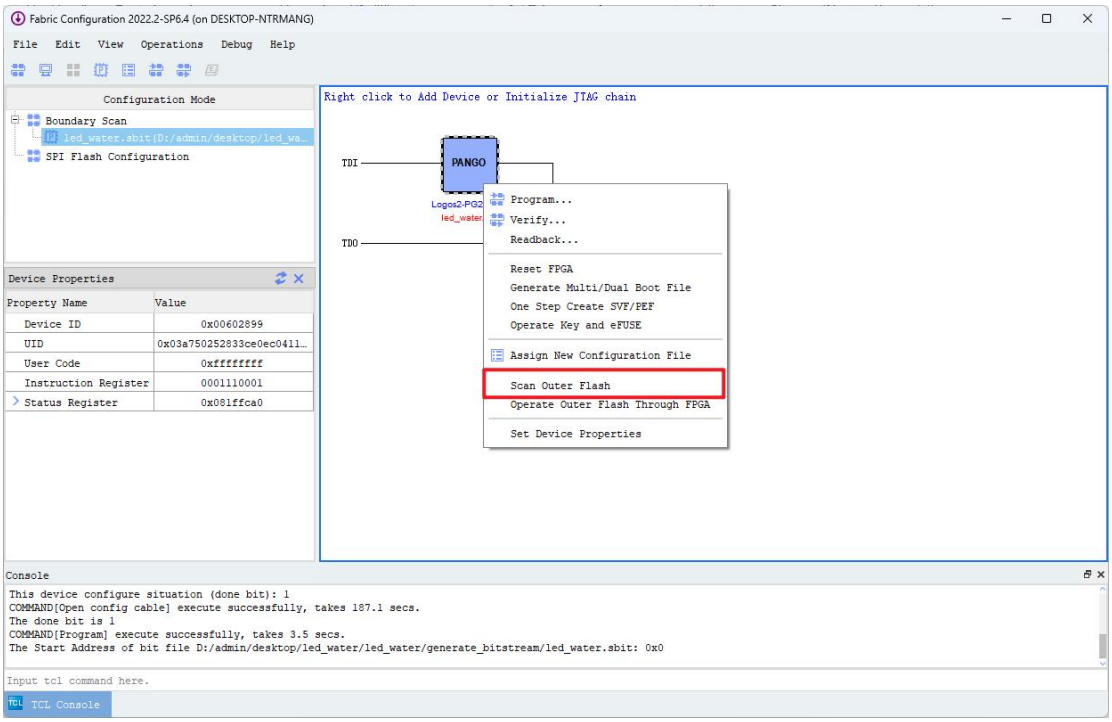


图 7.7-37

页面会显示板卡搭载的 Flash 的型号, 点击.sfc 文件, 点击 OPEN;

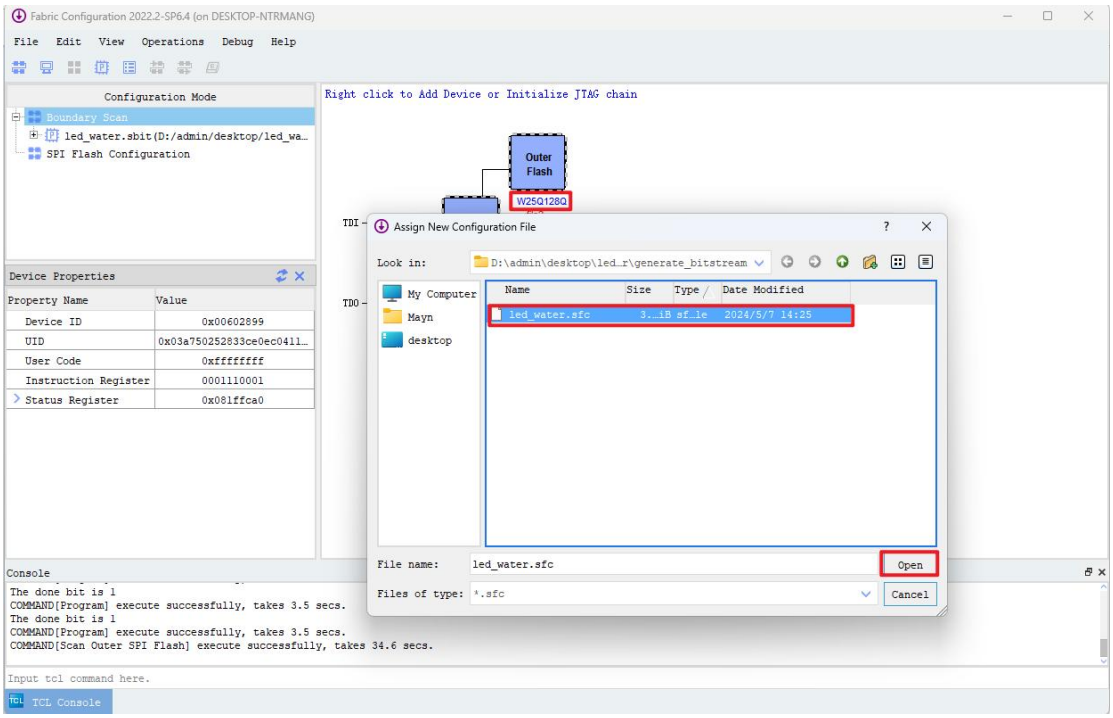


图 7.7-38

在下图位置点击鼠标右键后, 点击 Program;

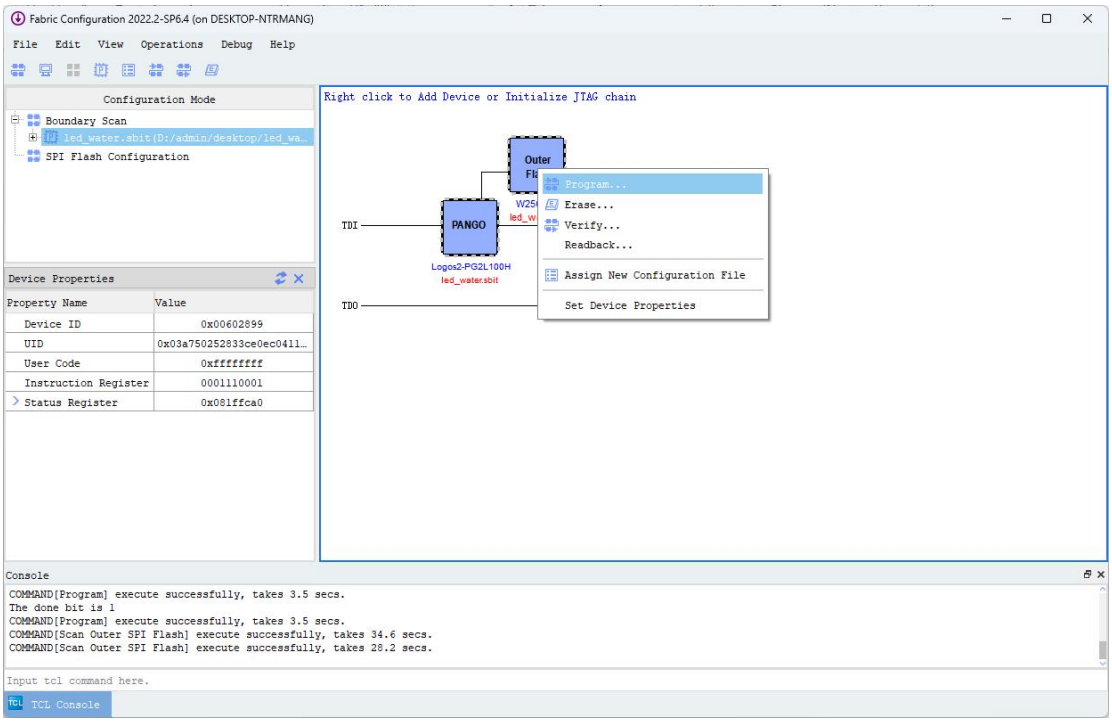


图 7.7-39

固化 Flash 成功如下图所示:

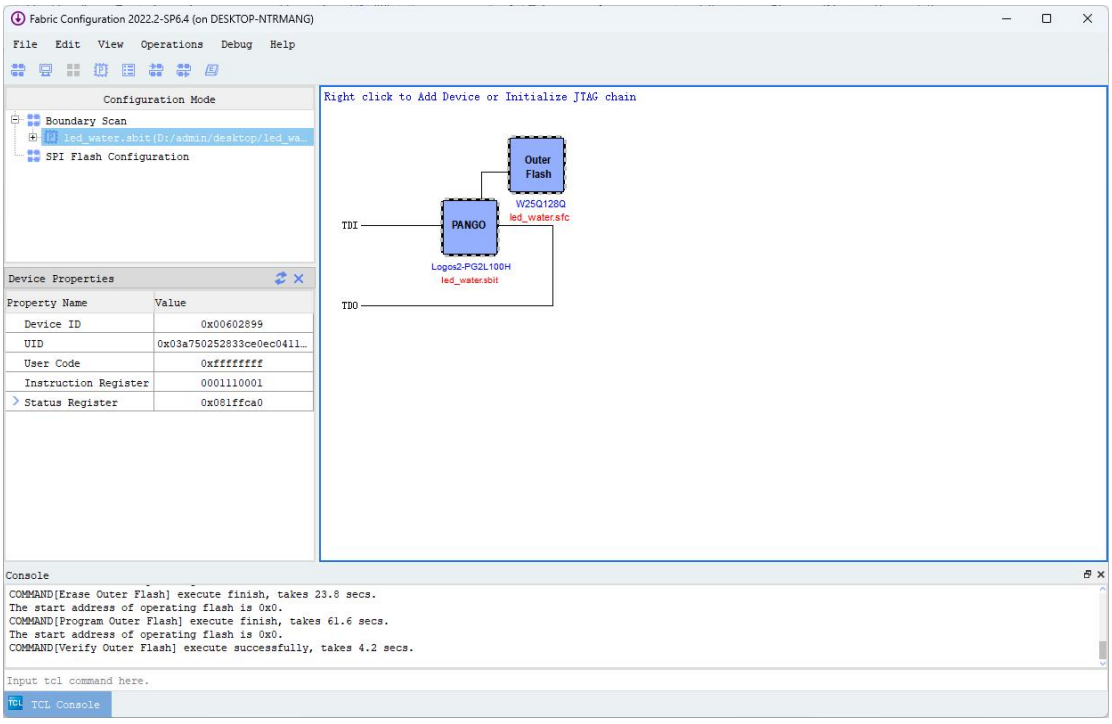


图 7.7-40

7.7.10. 上板验证

连接好电源和 jtag, 然后打开板卡上的电源;

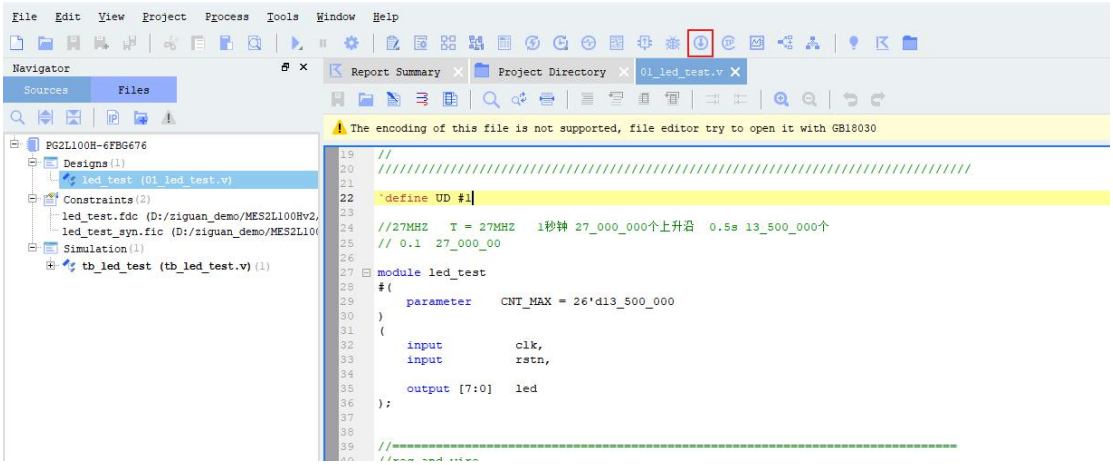


图 7.7-41

点击 PDS 软件上方的下载按钮, 红框所示部分:

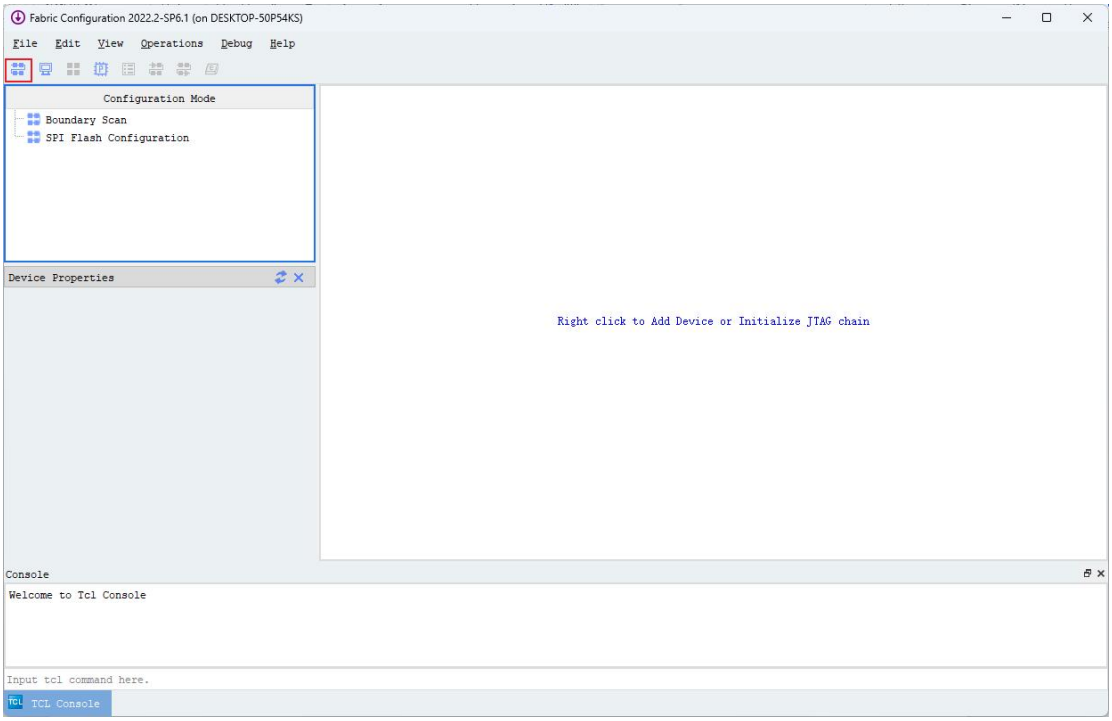


图 7.7-42

在弹出来的界面再点击如图红框部分:

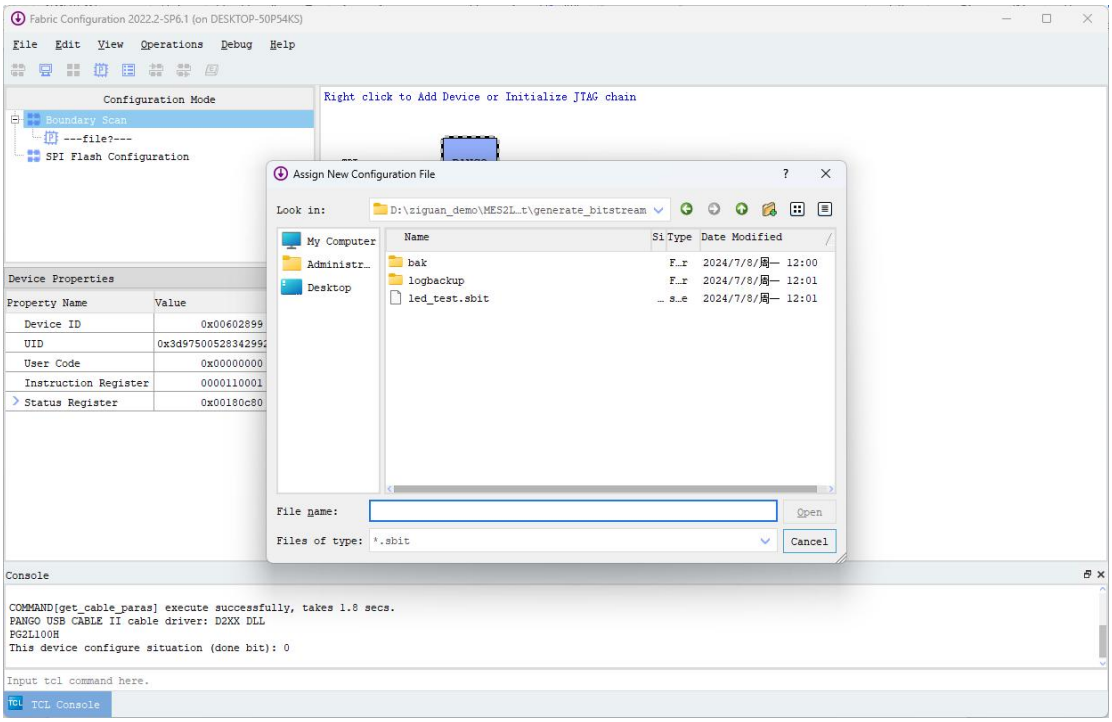


图 7.7-43

点击后会自动搜索设备, 如果连线没问题的话可以看到自动弹出一个界面来选择 sbit。然后选择 led_test.sbit;

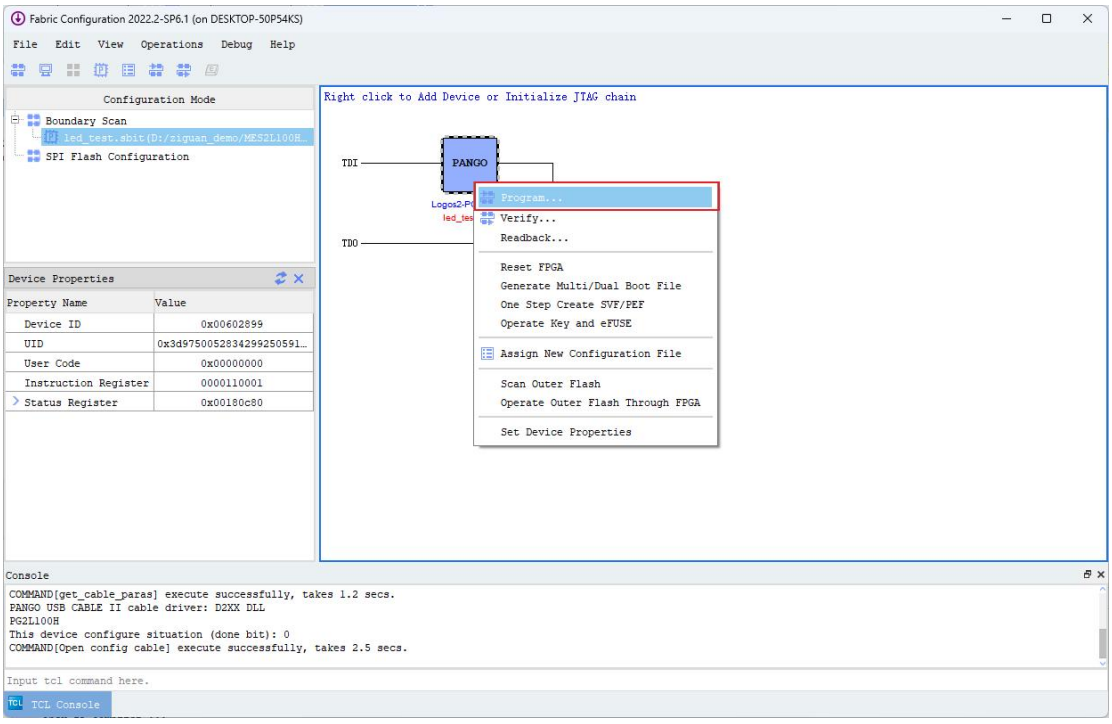


图 7.7-44

右键“芯片”, 然后选择 program, 等待烧录完成即可。接下来观察上板现象。

8. 基于紫光 FPGA 的 UART 串口通信

8.1. 实验简介

实验目的:

实现 FPGA 与 PC 之间的串口通信, 并使用 PDS 软件内置的在线调试工具验证收发数据的准确性。并根据接收到的不同数据去点亮开发板上的 LED 灯。

同时掌握在线 Debugger 工具的使用。

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

8.2. 实验原理

8.2.1.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:

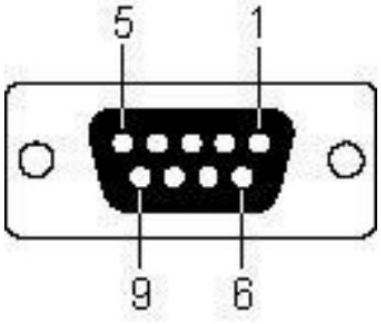


图 8.2-1

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:

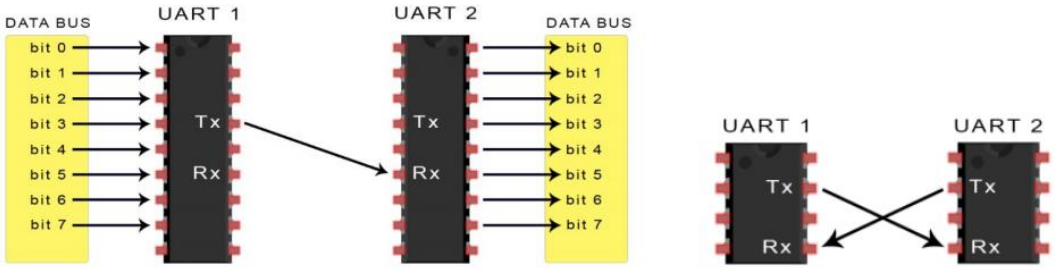


图 8.2-2

串口传输协议如下:

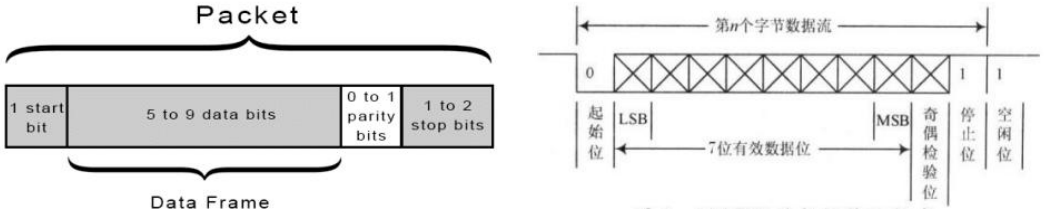


图 8.2-3

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:

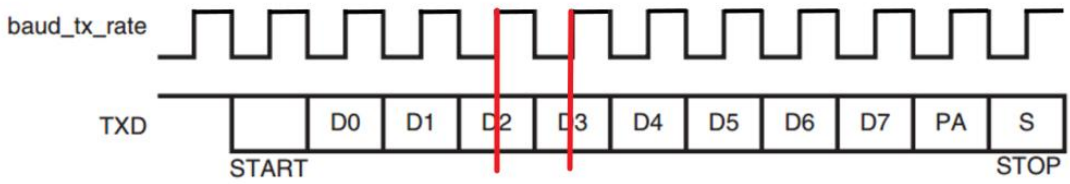


图 8.2-4

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。

8.2.2.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
```

图 8.2-5

8.2.3.串口接收数据点灯

收到的数据最终转为 8bit 的并行数据, rx_data, 而开发板上刚好也有 8 个 led 灯。1 表示亮, 0 表示灭, 那么 8 个 LED 灯就等同于 8bit 的变量。可以用二进制来表示, 例如要全亮, 那就是 8’b1111_1111, 因此我们往 FPGA 发送串口数据时, 可以通过 16 进制的方式来发送, 例如我要点亮 LED1 和 LED5, 那就发送 8’b0001_0001, 转为 16 进制就是 8’h11。所以可以根据该规律去点亮开发板上的每一个灯。

8. 3. 接口列表

uart_top.v 模块是整个工程的顶层模块

端口	I/O	位宽	描述
clk	input	1	系统时钟, 27MHZ
uart_rx	input	1	串口接收总线
led	output	8	led 灯信号
uart_tx	output	1	串口发送总线

uart_data_gen.v 模块 是生成要发送的串口数据。

端口	I/O	位宽	描述
clk	input	1	系统时钟, 27MHZ
read_data	input	8	预留的 8bit 串口接收数据信号,可作串口回环。
tx_busy	input	1	1: 表示正在发送 0: 表示空闲
write_max_num	input	8	发送的字符的个数
write_data	output	8	待发送的数据
write_en	output	1	串口数据有效信号

uart_tx.v 模块, 主要完成串口发送功能。将要发送到 PC 的 8bit 并行数据转为一个波特周期发送一个 bit。

端口	I/O	位宽	描述
BPS_NUM	**/**	**16**	**波特率分频系数**
clk	input	1	系统时钟, 27MHZ
tx_data	input	8	待串口发送的数据
tx_pluse	input	1	发送模块的触发信号, 一个脉冲信号
uart_tx	output	1	串口发送总线
tx_busy	output	1	1:表示正在发送 0:表示空闲

uart_rx.v 模块, 主要完成串口接收功能, 将 PC 发送到 FPGA 的串行的 8bit 数据转为并行的 8bit 数据给到用户使用。

端口	I/O	位宽	描述
BPS_NUM	**/**	**16**	**波特率分频系数**
clk	input	1	系统时钟, 27MHZ
uart_rx	input	1	串口接收总线
rx_data	output	8	接收到的 8bit 并行数据
rx_en	output	1	接收数据使能信号
rx_finish	output	1	接收完成信号

8.4. 工程说明

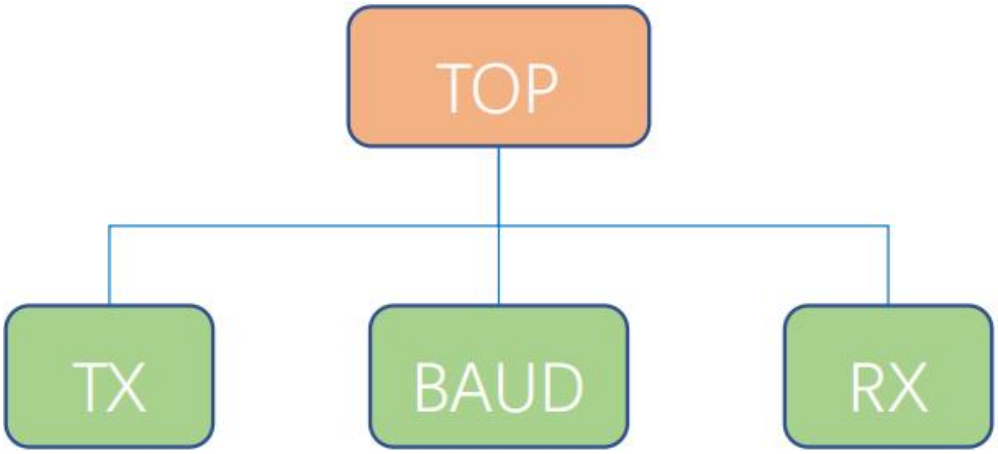


图 8.4-1

实验的架构如图 8.4-1 所示，分为 TX(发送模块)、BAUD(波特率计算模块)、RX(接收模块)。而 TOP 模块是顶层，例化了这三个模块。我们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，但是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。

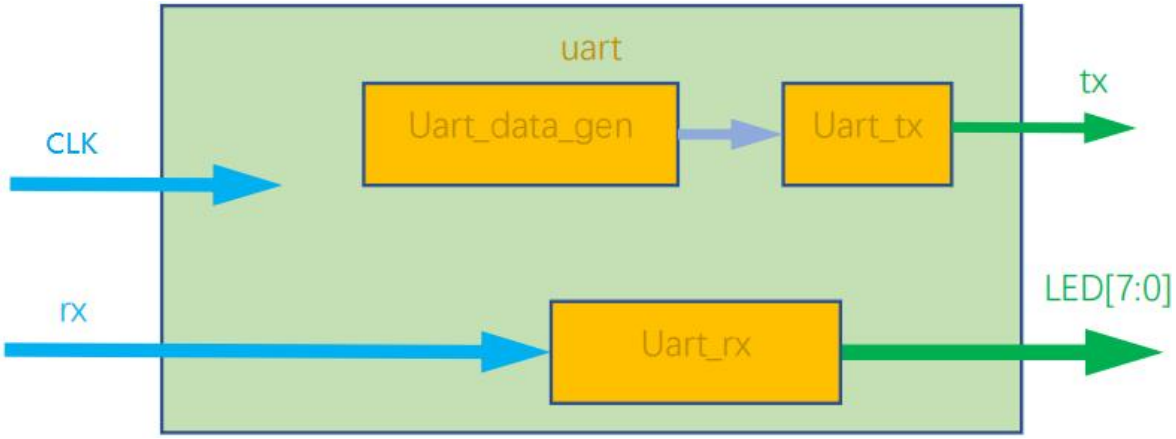


图 8.4-2

图 8.4-2 所示的 Uart_data_gen 模块是用来生成要发送的数据。即“www.meyesemi.com”这一串字符。生成后的数据给到 uart_tx 模块让其发送到 PC。Uart_rx 模块负责接收 PC 发来的数据，并做解析，然后点亮开发板上的 LED 灯。

8.5. 代码模块说明

实现功能: 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）；

这里主要讲解代码的重点设计部分，完整源码大家请参考工程源码，然后结合文档一起理解比较方便。

代码的设计主要是通过 bit 计数与 baud 计数控制状态跳转，在状态机中输出；

8.5.1.串口发送模块

```
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 tx_data[0];
         3'h1 : uart_tx <= `UD tx_data[1];
         3'h2 : uart_tx <= `UD tx_data[2];
         3'h3 : uart_tx <= `UD tx_data[3];
         3'h4 : uart_tx <= `UD tx_data[4];
         3'h5 : uart_tx <= `UD tx_data[5];
         3'h6 : uart_tx <= `UD tx_data[6];
         3'h7 : uart_tx <= `UD tx_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
```

在代码的第 5 行中有个 tx_state 的变量, 一共有四个状态 IDLE(空闲状态)、SEND_START(开始发送状态)、SEND_DATA(发送数据状态)、SEND_STOP(发送停止状态)。根据我们的串口发送协议。在 IDLE 状态下就直接保持 uart_tx 总线为高电平即可。在

SEND_START 状态下发送起始为, 一个波特周期的低电平, 之后跳转到 SEND_DATA 状态,

发送 8bit 数据, 通过 tx_bit_cnt 来计数一共发了多少个 bit, tx_bit_cnt 从 0 计数到 7, 一共发送 8bit 数据, 需要注意的时, 发送时先把 tx_data 的低位发出去。发送完 8bit 数据后跳转到 SEND_STOP 状态, 输出一个波特周期的高电平, 表示停止。至此就完成了一次串口数据的发送。

下面的这两个 always 块主要完成波特率的生成, 在串口接收模块里也有相同的代码, 故在接收模块里不再讲解。

```
always @ (posedge clk)
begin
  if(!tx_en)
    tx_bit_cnt   <= `UD 3'h0;
  else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
    tx_bit_cnt   <= `UD 3'h0;
  else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
    tx_bit_cnt   <= `UD tx_bit_cnt + 3'h1;
  else
    tx_bit_cnt   <= `UD tx_bit_cnt;
end
```

```
always @ (posedge clk)
begin
  if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
    clk_div_cnt  <= `UD 16'h0;
  else
    clk_div_cnt  <= `UD clk_div_cnt + 16'h1;
end
```

代码 1-11 行完成 tx_bit_cnt 的计数, 代码 12-18 行, 完成波特周期的计数,clk_div_cnt 是用来对我们输入进来的 50MHZ 的时钟进行分频, 以此来得到波特周期。其中 BPS_NUM 的值根据不同的波特率会有不同的结果, 默认是 115200, 所以 BPS_NUM=50MHZ/115200≈434。计算公式为 BPS_NUM = CLK/BAUD_RATE。需要注意的是, 第 14 行的判断条件, clk_div_cnt 的清 0 除了每次计数到 BSP_NUM 后需要清 0 之外, 当发送使能的上升沿到达来到时, 我们也需要清 0。tx_pluse 实际是发送使能信号, tx_pluse_reg 是 tx_pluse 打一拍后的信号, ~tx_pluse_reg & tx_pluse 表示是获取 tx_pluse 的上升沿。所以, 该条件表示我们要开始发送数据了, 为了能够正常计数波特周期, 故要把 clk_div_cnt 清 0, 让其重新计数。

代码第三行, 当 tx_en 为低电平时, 表示不处于发送状态, 所以一直置 0。代码第 5 行, 当 tx_bit_cnt==7 且 clk_div_cnt== BPS_NUM 时, 把 tx_bit_cnt 清空, 因为此时已经发送了 8 个 bit 的数据。代码第 7 行, 在发送状态下, 每到来一个波特周期就让 tx_bit_cnt 不断+1。其余情况下, tx_bit_cnt 保持不变。需要注意的是, 波特率在发送模块和接收模块里都是一样的, 故接收模块将不讲解这一部分。

8.5.2.串口接收模块

串口接收模块是发送模块的逆过程, 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, 在前面讲发送的时候都是在波特周期开始的位置变更数据, 但是接收数据在提取时需要在 rx 稳定时刻取数, 也就是在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;

串口接收代码的重点部分如下:

```
//状态机状态跳转条件及跳转规律
  always @ (*)
  begin
   case(rx_state)
     IDLE     :
     begin
       if(start)                          //监测到 start 信号到来, 下一状态跳转到 start 状态
         rx_state_n = RECEIV_START;
       else
         rx_state_n = rx_state;
     end
     RECEIV_START   :
     begin
       if(clk_div_cnt == BPS_NUM)               //已完成接收 start 标志信号
         rx_state_n = RECEIV_DATA;
       else
         rx_state_n = rx_state;
     end
     RECEIV_DATA   :
     begin
       if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM) //已完成 8bit 数据的传输
         rx_state_n = RECEIV_STOP;
       else
         rx_state_n = rx_state;
     end
     RECEIV_STOP   :
     begin
       if(clk_div_cnt == BPS_NUM)                //已完成接收 stop 标志信号
         rx_state_n = RECEIV_END;
       else
         rx_state_n = rx_state;
     end
     RECEIV_END   :
```

```
        begin
          if(!uart_rx_1d)    //数据线重新被拉低, 表示新数据传输又发送 start 标志信号, 需要跳转到 start 状态
            rx_state_n = RECEIV_START;
          else                          //没有其他状况出现时, 回到空闲状态, 等待 start 信号的到来
            rx_state_n = IDLE;
        end
        default   : rx_state_n = IDLE;
      endcase
    end

    // 状态机输出
    always @ (posedge clk)
    begin
      case(rx_state)
        IDLE       ,
        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;
        begin
          rx_en <= `UD 1'b0;
          rx_data_reg <= `UD 8'h0;
        end
        RECEIV_DATA :
        begin
          if(clk_div_cnt == BPS_NUM[15:1])      //在一个波特周期的中间位置取数据线上传输的数据;
            rx_data_reg  <= `UD {uart_rx , rx_data_reg[7:1]};  //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）
        end
        RECEIV_STOP :
        begin
          rx_en   <= `UD 1'b1;              // 输出使能信号, 表示最新的数据输出有效
          rx_data <= `UD rx_data_reg;          // 将缓冲寄存器的值赋值给输出寄存器
        end
        RECEIV_END   :
        begin
          rx_data_reg <= `UD 8'h0;
        end
```

```
        default:    rx_en <= `UD 1'b0;
    endcase
end
```

代码的 1-42 行主要完成 rx_state 的状态跳转。一共有 5 个状态。IDLE(空闲状态)、RECEIV_START(接收开始状态)、RECEIV_DATA(接收数据状态)、RECEIV_stop(停止接收状态)、RECEIV_END(接收结束状态)。在 IDLE 状态下, 等待 start 信号到来, 一旦 start 信号拉高, 跳转到 RECEIV_START 状态。在 RECEIV_START 状态下经过一个波特周期后就跳转到 RECEIV_DATA 状态, 之后经过 8 个波特周期, 即接收 8bit 数据后再跳转到 RECEIV_STOP 状态。在 RECEIV_STOP 下, 等待一个波特周期后跳转到 RECEIV_END 状态, 整个接收流程到此结束。

接下来看代码的 45-70 行, 主要执行 rx_state 在不同状态下的逻辑操作。在 IDLE 和 RECEIV_START 状态下都将接收使能和 rx_data_reg(接收数据缓存)置 0, 等待接收数据到达。在 RECEIV_DATA 状态下, 注意判断条件是 clk_div_cnt==BPS_NUM[15:1],在前面说过,我们要在中心时刻采样数据才是最稳定的,所以 BPS_NUM[15:1]其实就是 BPS_NUM>>1, 即左移 1 位, 就是除以 2。所以该状态就是每次到达波特周期的中心时刻就把数据给到 rx_data_reg 进行缓存, 而 rx_data_reg <= {uart_rx,rx_data_reg[7:1]}则是将进来的 uart 数据不断右移, 最后的结果就是最先进来的数据会放在低位。因为我们发送的时候也是先把最低位的数据先发出去, 所以这里就是把最先收到的数据放到低位, 这样就和发送的数据一一对应。在 RECEIV_STOP 状态下, 拉高 rd_en 信号, 表示接收的数据有效,并将 rx_data_reg 的值赋值给 rx_data(输出寄存器),所以这个时刻下,rx_data 和 rd_en 是同步的, 因此读者要正确使用接收的数据话可以用 rd_en 作为有效条件。在 RECEIV_END 状态下, 将 rx_data_reg 清空。

而波特率生成模块都集成在模块内部, 并且和发送模块的是一模一样, 故不再讲解。

8.5.3.工程及现象

通过使用在线 Debugger 工具来抓取收到的串口数据,并进行对比,比较收到的和我们发送的是否一致。

打开 PDS 上方工具栏的 Insert 工具,如图所示:

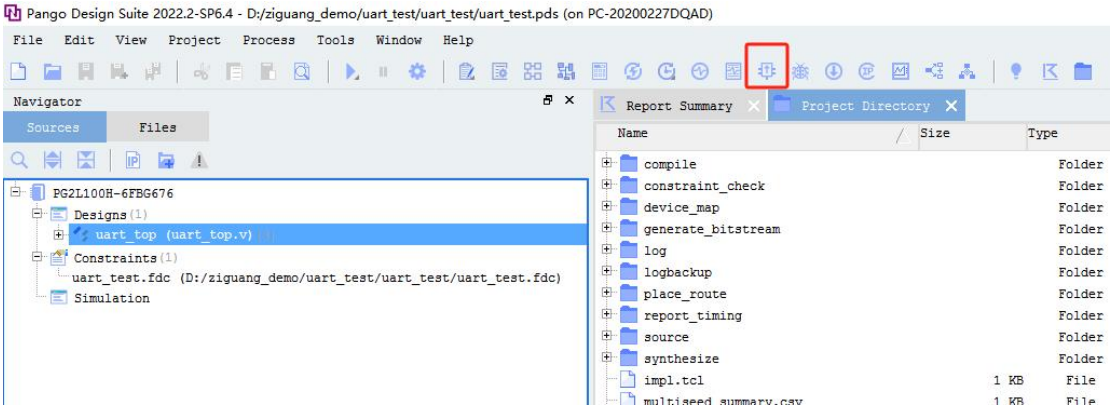


图 8.5-1

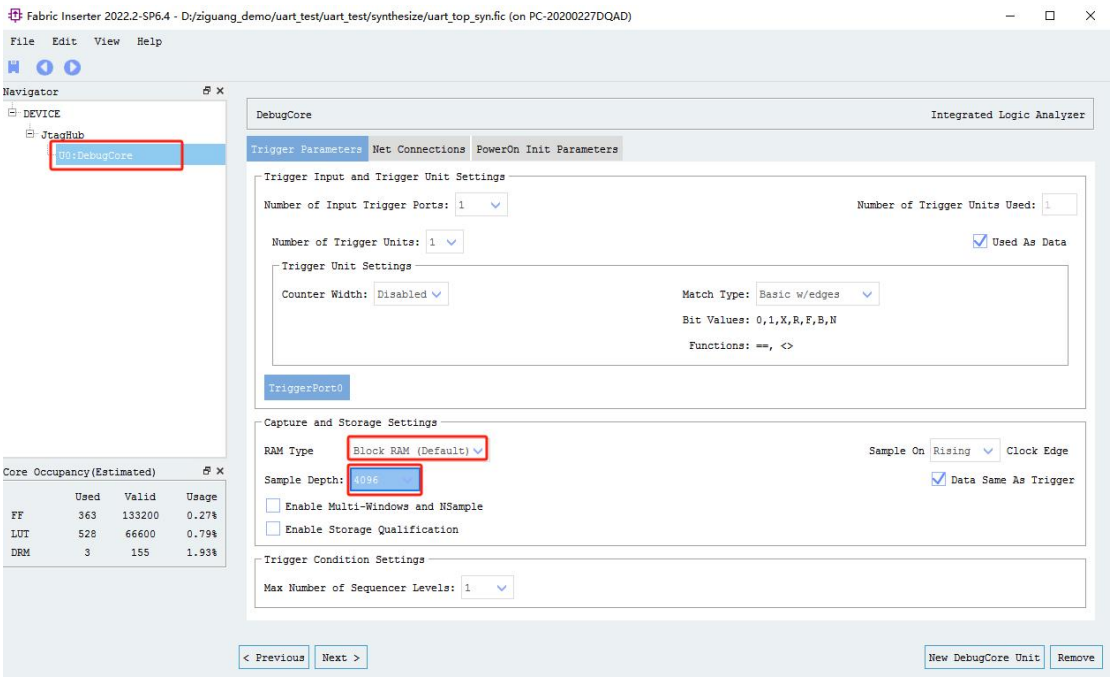


图 8.5-2

RAM Type 默认使用 Block RAM,采样深度选择 4096 即可,不用太大,添加采样时钟,这里的时钟最好是和我们要采样的数据是同个时钟源,或者频率比数据时钟源高也行,但是不能低,否则可能会采样不到。

添加信号的操作流程如下,添加完成然后保存。

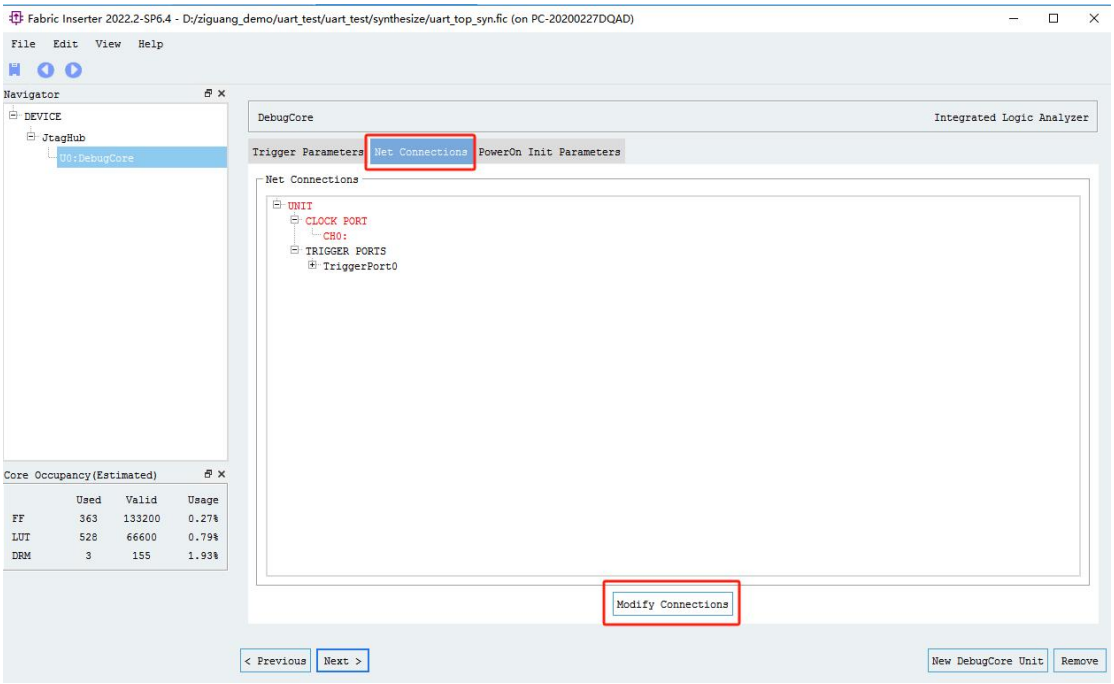


图 8.5-3

首先点击 Modify Connections;

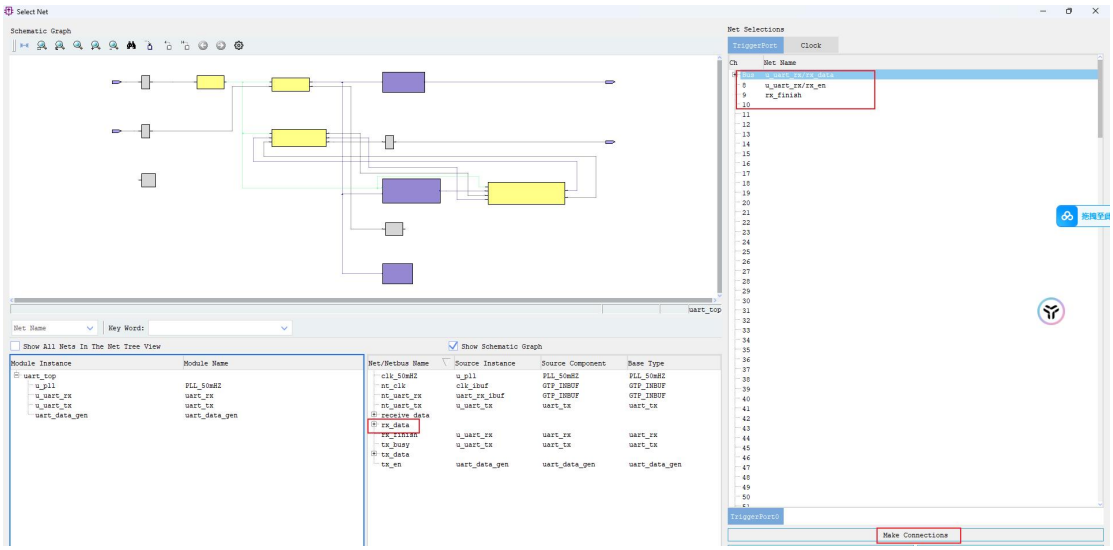


图 8.5-4

之后点击要抓取的信号, 然后点击下方的 Make Connection 即可。

图 8.5-5

添加上图所示的全部信号并保存即可。

8.6. 在线 Debugger 工具的使用

8.6.1.Fabric 工具说明

PDS 在线调试工具主要用于对 FPGA 内部实际运行中信号的捕获, 便于调试者进行分析和定位问题, 在调试过程中可以很大程度上可以替代逻辑分析仪, 避免繁琐的连线工作和节省成本。本 demo 通过 Fabric Inserter 和 Fabric Debugger 两个工具的使用实现在线调试。

Fabric Inserter 工具将 DebugCore 自动插入用户的设计网表中生成新的设计网表, 从而使用户不需要手工在 HDL 代码中例化。

Fabric Debugger 是一款界面化 FPGA 芯片调试工具, 直接与 JtagHub 和 DebugCore 交互, 可实时配置目标 FPGA、设置触发条件并观测目标信号捕获结果。

实现在线调试流程中, inserter 工具和 debugger 工具的作用如下图所示

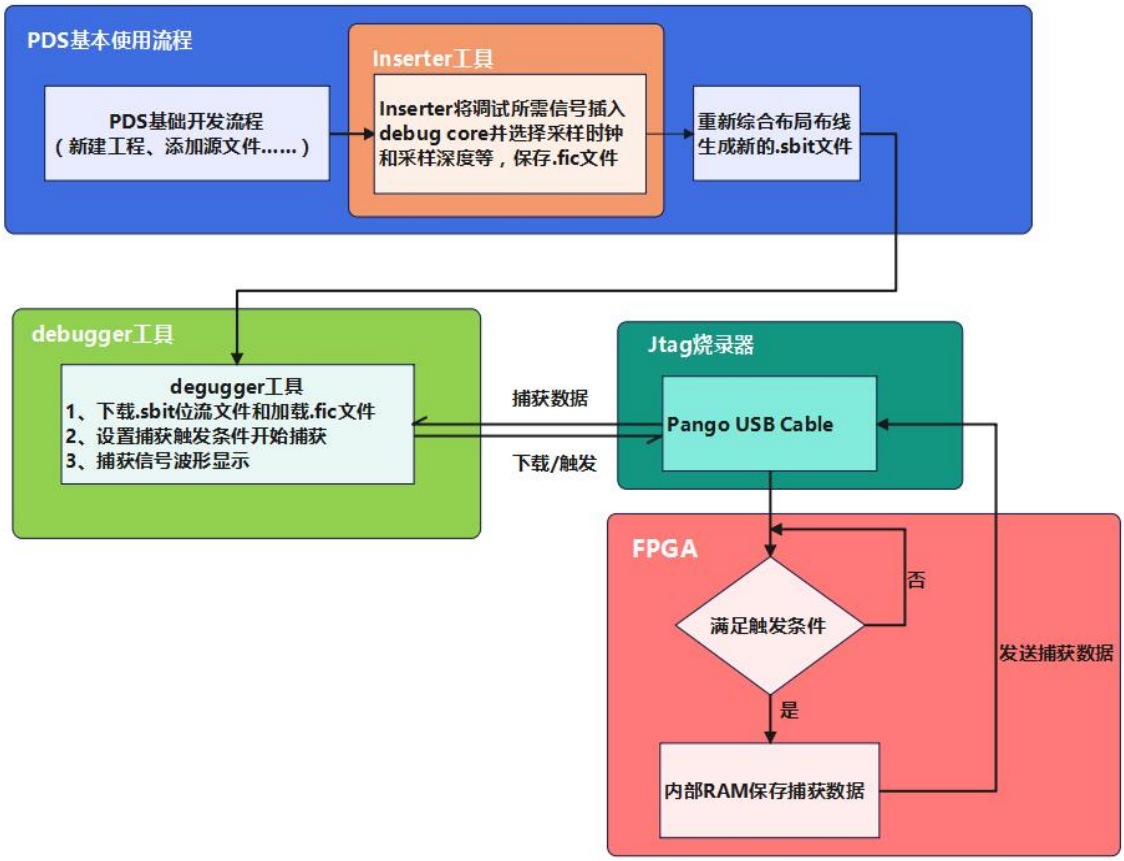


图 8.6-1 inserter 和 debugger 作用

8.6.2.Fabric 工具使用

点击 Insert 旁边的像瓢虫的图标” Debugger”。

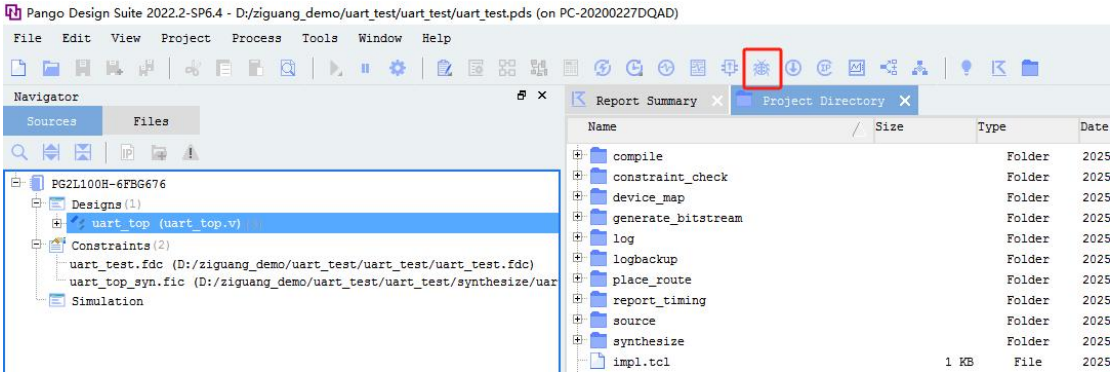


图 8.6-2

在弹出的界面点击左上角的图标寻找 JTAG 设备, 如下所示:

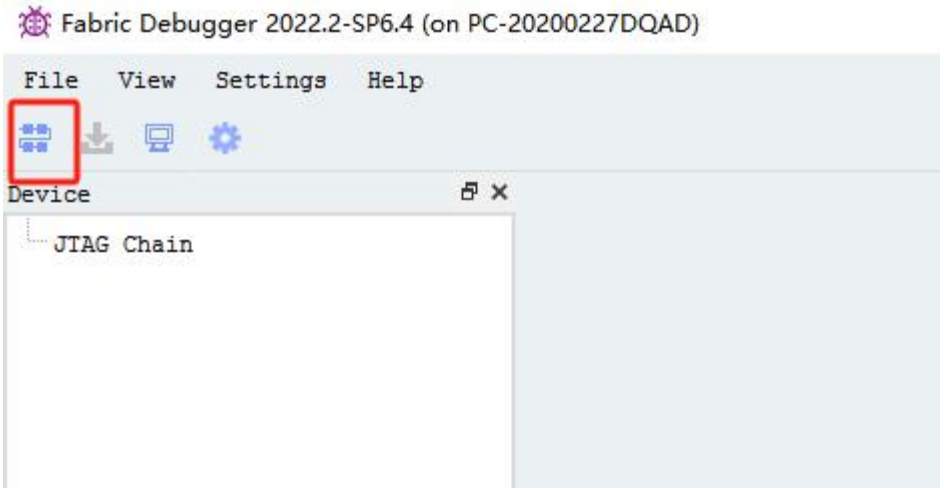


图 8.6-3

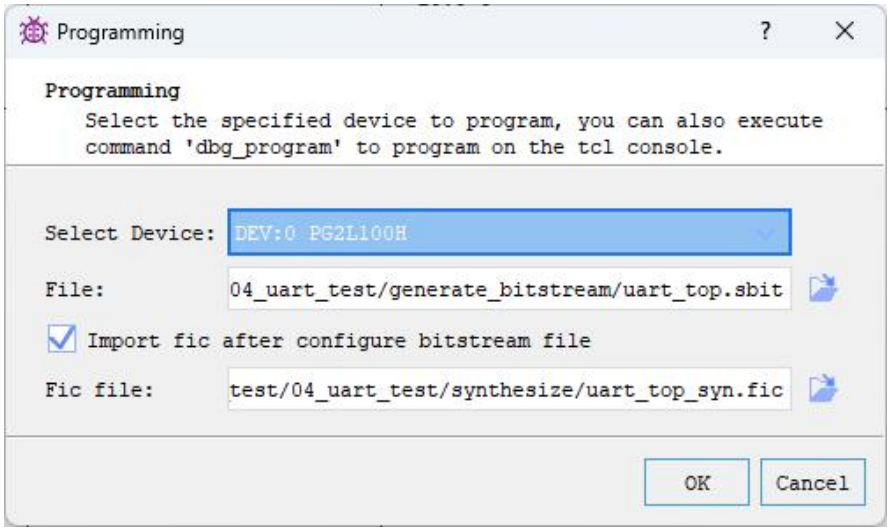


图 8.6-4

点击 OK, 下载 sbit 文件以及导入 fic 文件。

Trigger Unit	Function	Value	Radix
TU0(TriggerPort0)	==	XR_XXXX_XXXX	Bin
u_uart_rx/rx_data		XXXX_XXXX	
u_uart_rx/rx_en		R	
rx_finish		X	

图 8.6-5

将 rx_en 设置为上升沿触发。

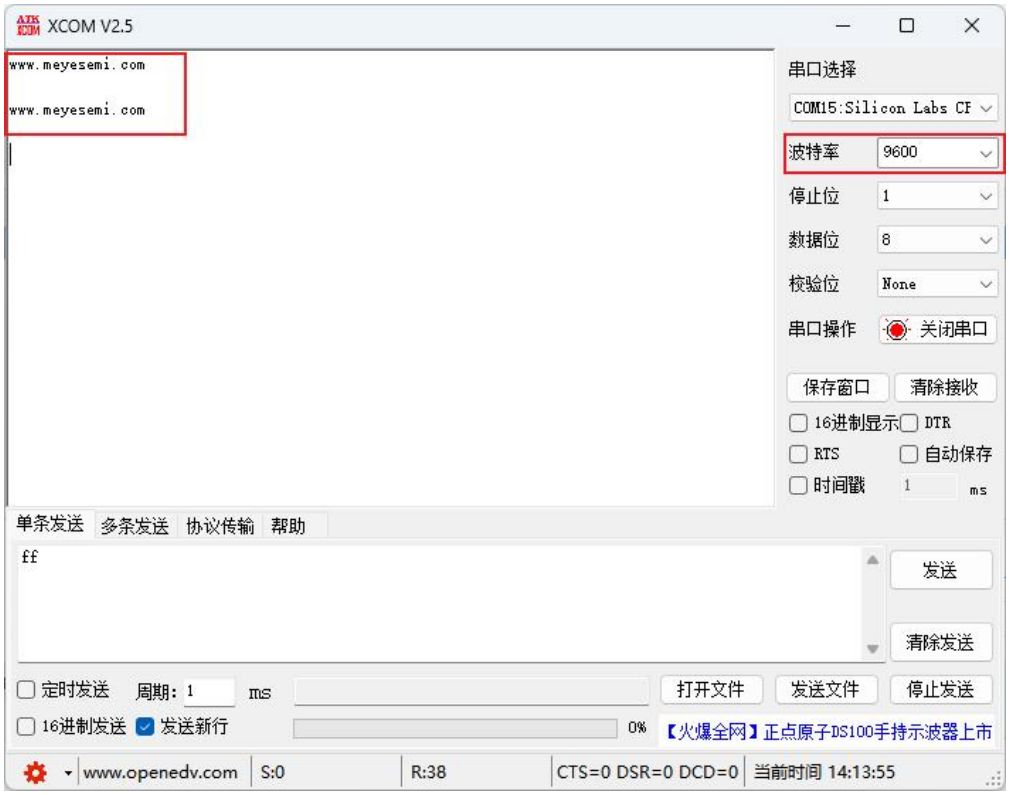


图 8.6-6

打开串口助手, 波特率设置为 9600, 然后打开端口, 我的电脑是 COM15 端口, 可以看到每隔 1s, 串口助手会收到 www,meyesemi.com 的字符串, 具体端口号大家可以打开设备管理器进行查看, 如下所示:

图 8.6-7

接下来回到 Debugger 界面, 运行。

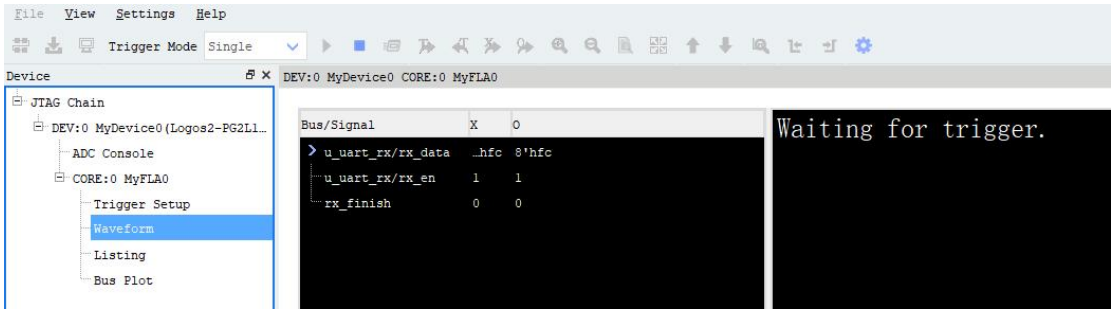


图 8.6-8

等待触发, 此时串口助手发送 ff,注意勾选 16 进制发送。

图 8.6-9

图 8.6-10

可以看到 rd_en 拉高, 同时 rx_data 的值也为 0xff, 和我们 PC 发送的数据一致。同时观察开发板上的 8 个 led 灯, 可以看到 8 个 LED 灯全亮。ff 对应二进制 1111_1111, 所以会亮 8 个灯。以此类推, 1a 的二进制是 0001_1001, 所以会亮 LED1、LED4、LED5 三个灯, 大家可以自行尝试。

9. DDR3 读写实验例程

9.1. 实验简介

实验目的:

完成 DDR3 的读写测试。

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

9.2. 实验原理

MES2L100HP 开发板集成两颗 4Gbit(512MB)DDR3 芯片,型号为 MT41K256M16。DDR3 的总线宽度共为 32bit。DDR3 SDRAM 的最高数据速率 1066Mbps。

9.2.1.DDR3 控制器简介

PG2L100H 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

支持 x8、x16 Memory Device

最大位宽支持 32 bit

支持精简的 AXI4 总线协议

一个 AXI4 256 bit Host Port

支持 Self_refresh, Power down

支持 Bypass DDRC

支持 DDR3 Write Leveling 和 DQS Gate Training

DDR3 最快速率达 1066 Mbps

9.3. 工程说明

PDS 安装后, 需手动添加 DDR3 IP, 请按以下步骤完成:

DDR3 IP 文件: PG2L_IP\PG2L_IP\DDR3\ips2l_hmic_s_v1_10.iar

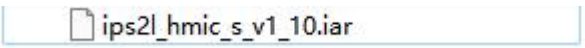


图 9.3-1

IP 安装步骤: 查看 “工具使用篇\03_IP 核安装与查看用户指南”

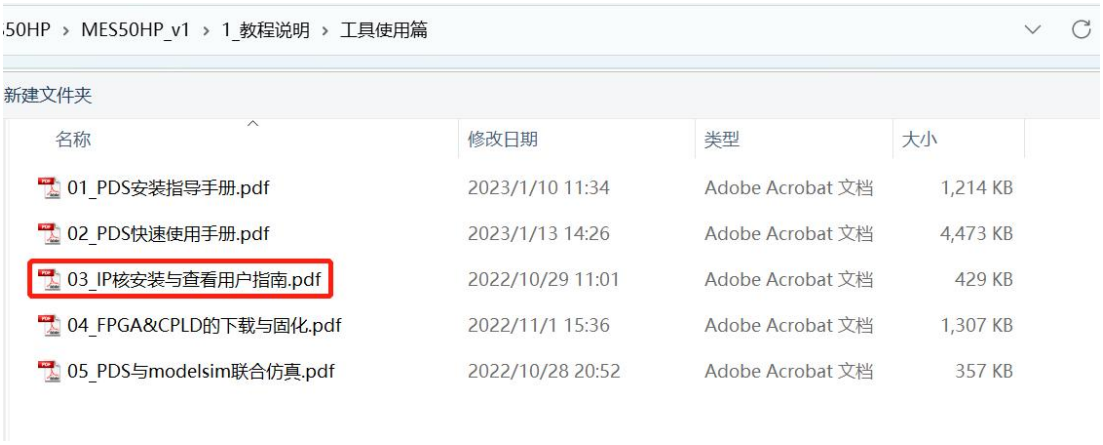


图 9.3-2

9.3.1. DDR3 读写 Example 工程

打开 PDS 软件, 新建工程 ddr3_test, 点开如下图标, 打开 IP Compiler;

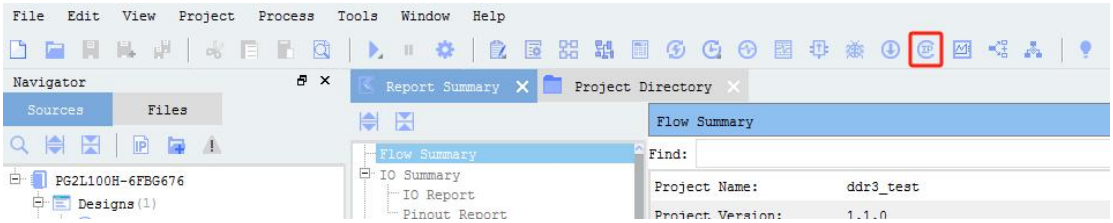


图 9.3-3

选择 DDR3 IP, 取名, 然后点击 Customize;

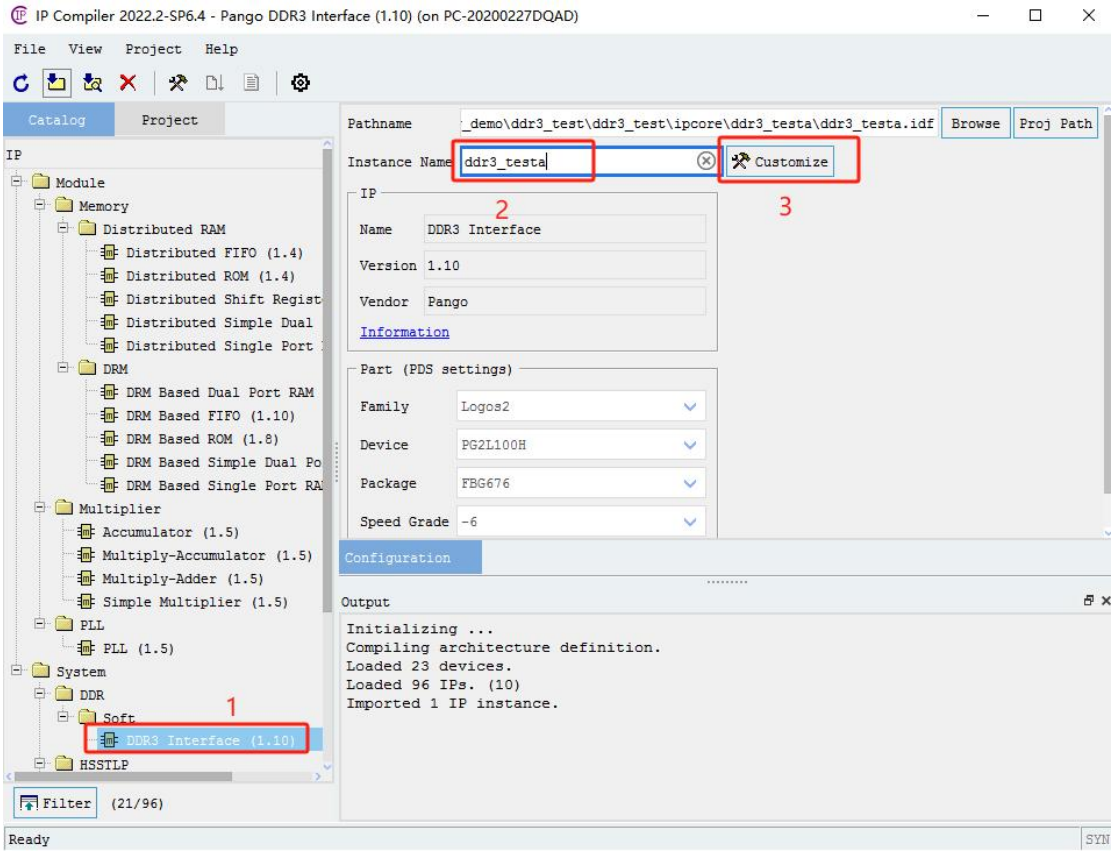


图 9.3-4

在 DDR3 设置界面中 Step1 按照如下设置:

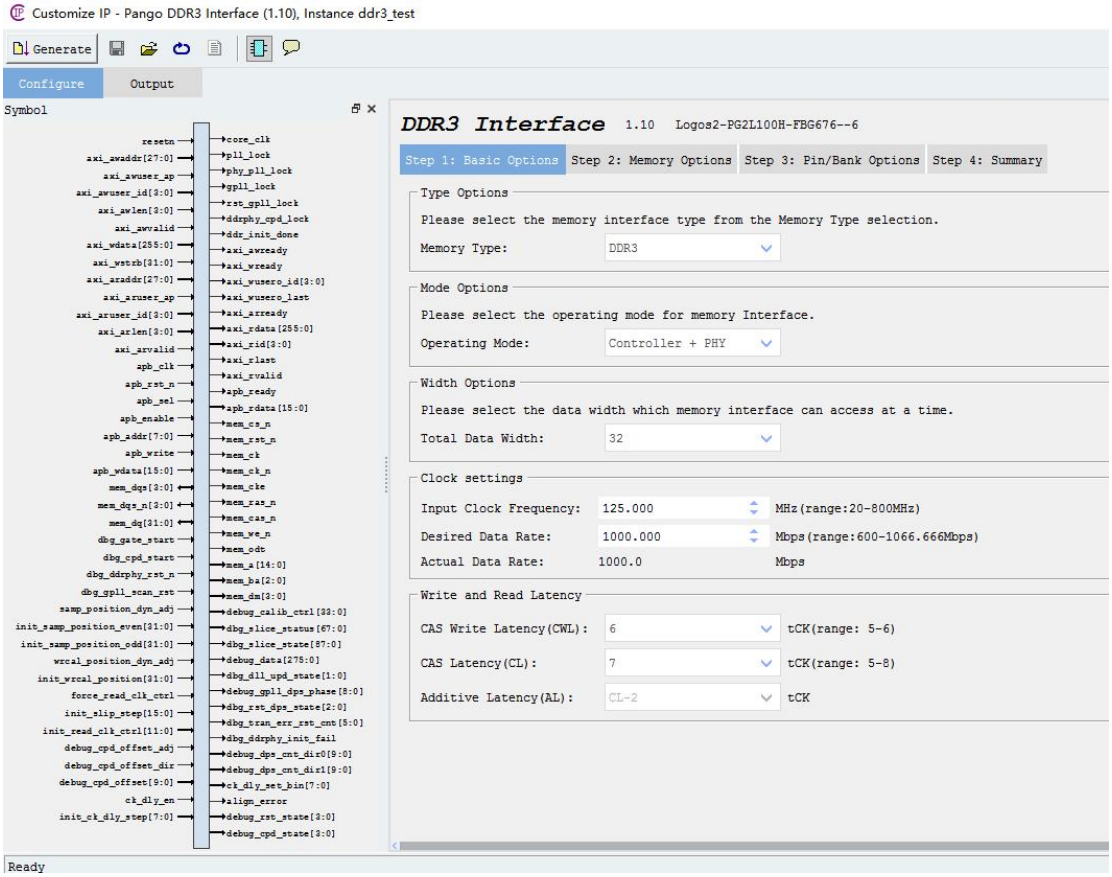


图 9.3-5

Step2 按照如下设置:

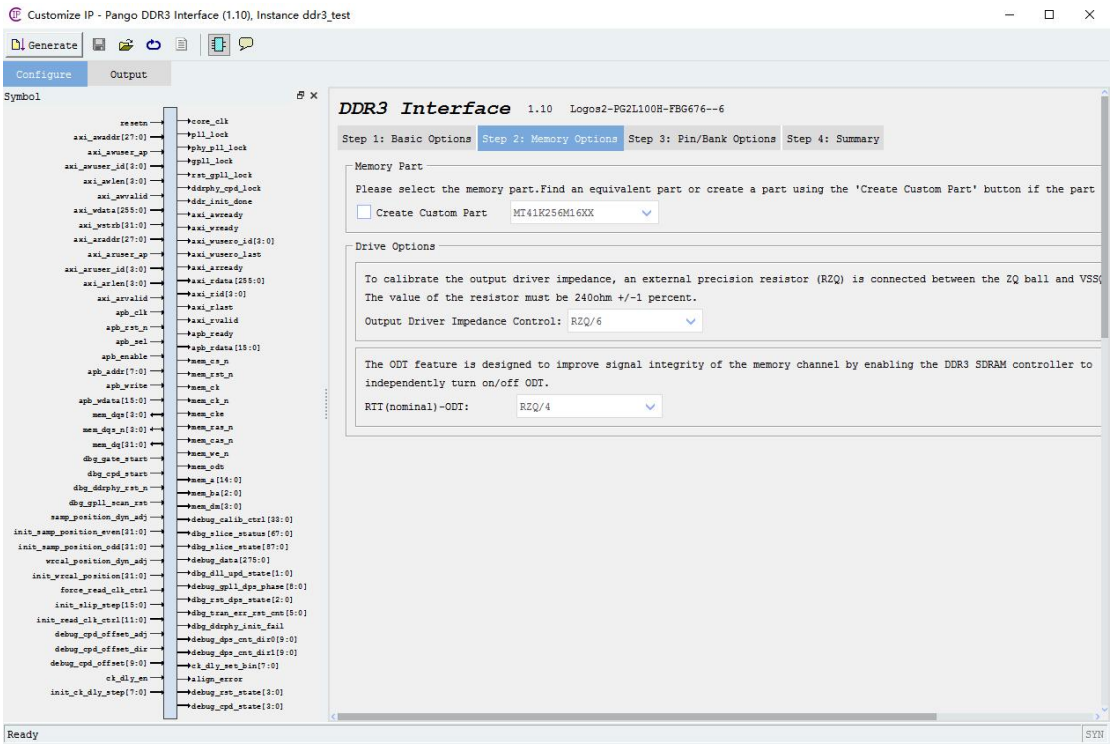


图 9.3-6

Step3 按照如下设置，勾选 Custom Control/Address Group，管脚约束参考原理图：

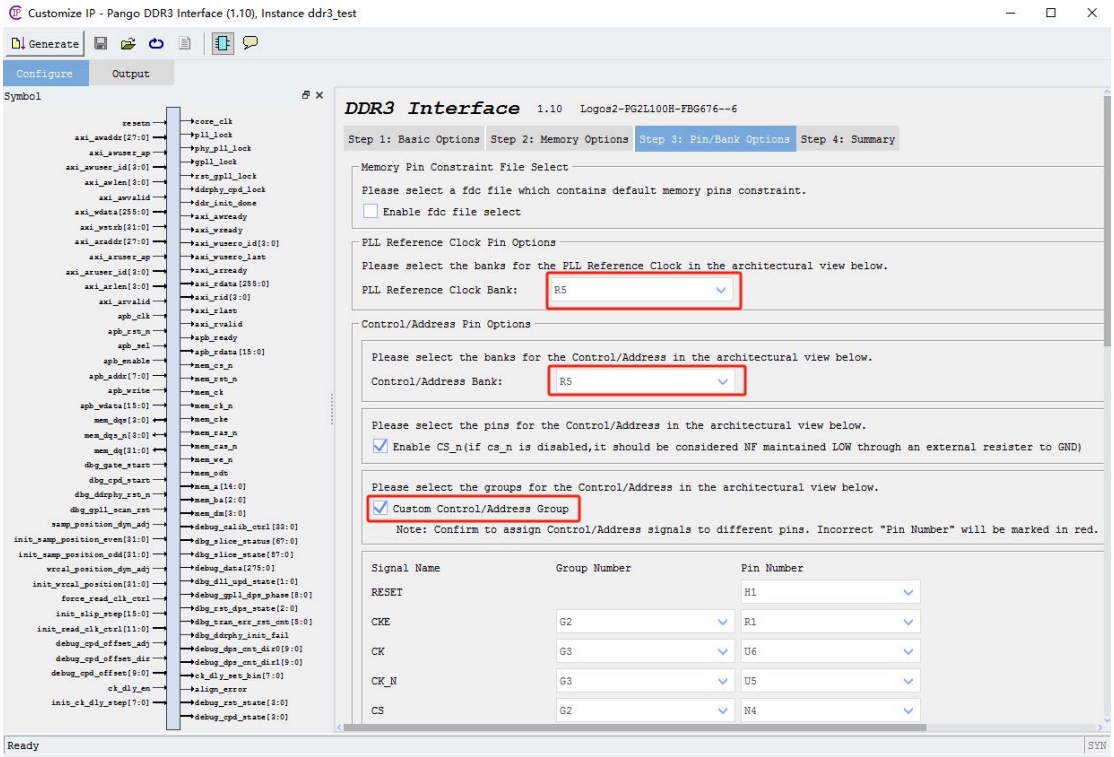


图 9.3-7

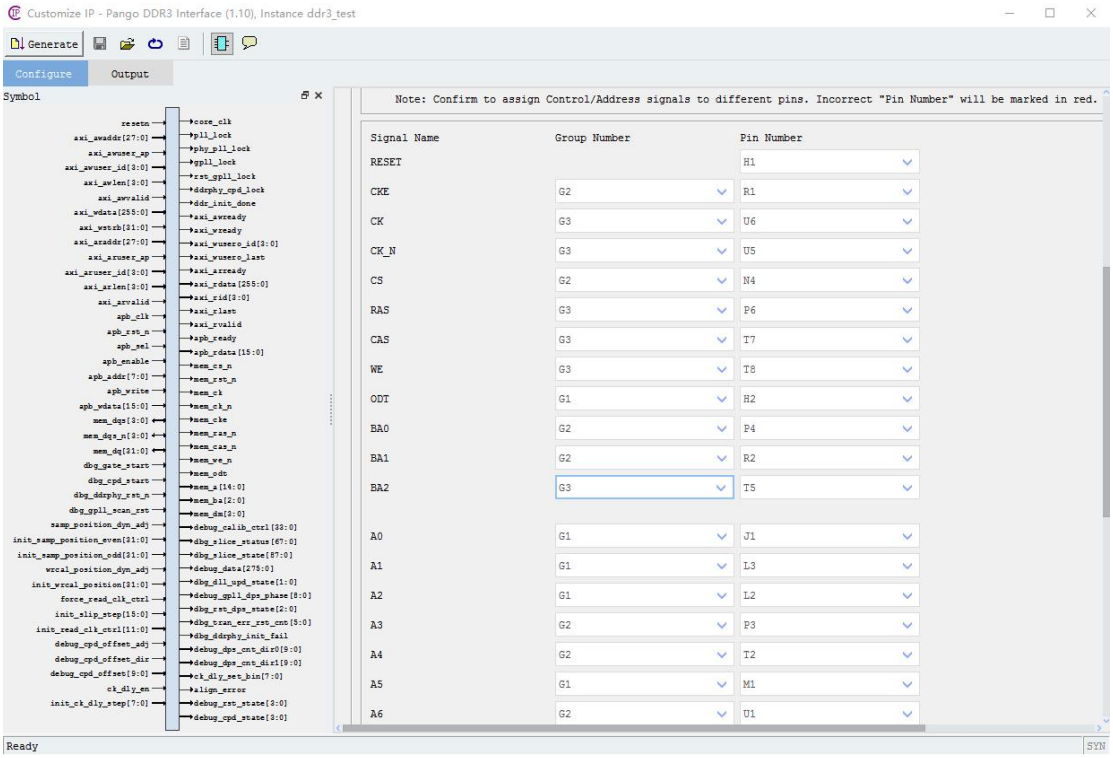


图 9.3-8

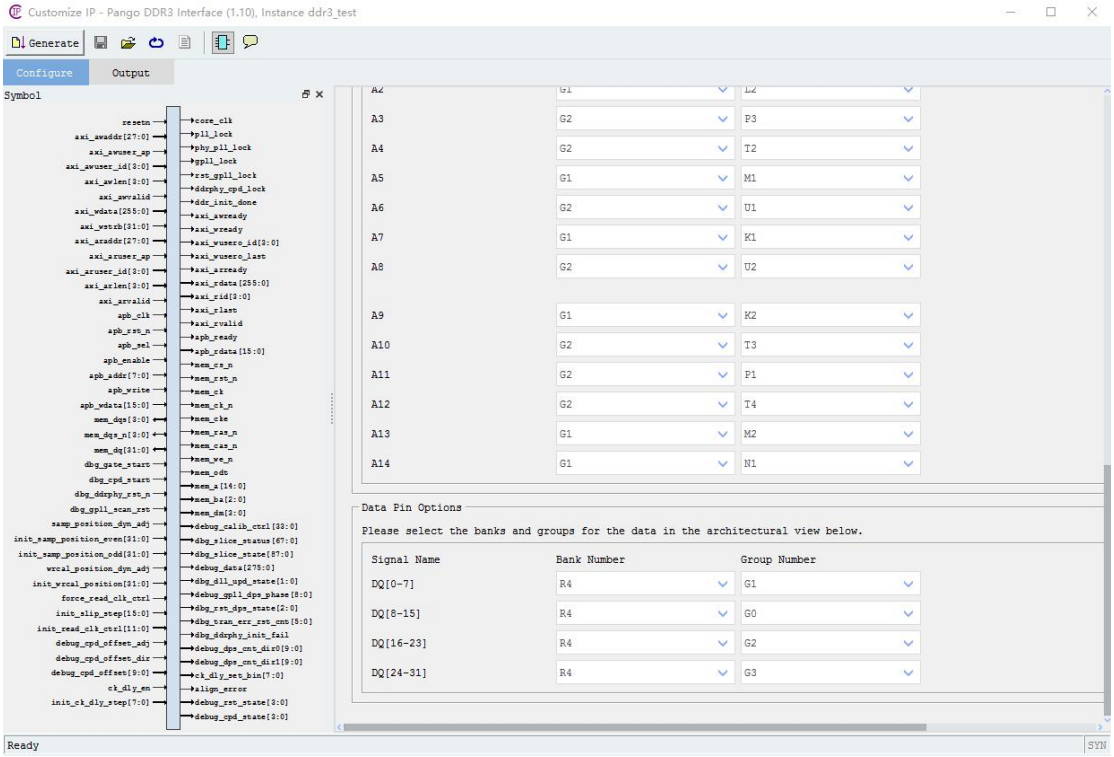


图 9.3-9

提醒:

在设置 IP 核时, step 3: pin/bank options 中, 管脚设置中的 Group Number 与原理图的对应关系如下图

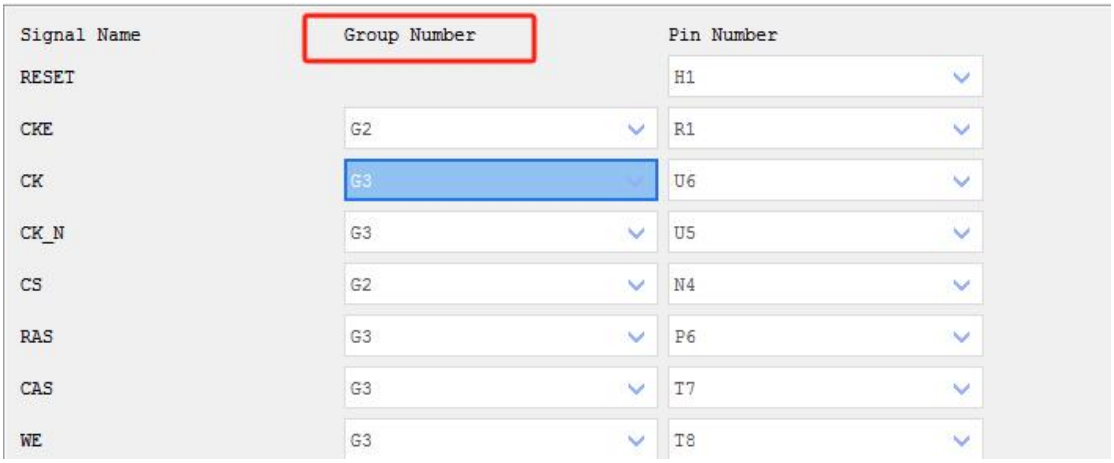


图 9.3-10

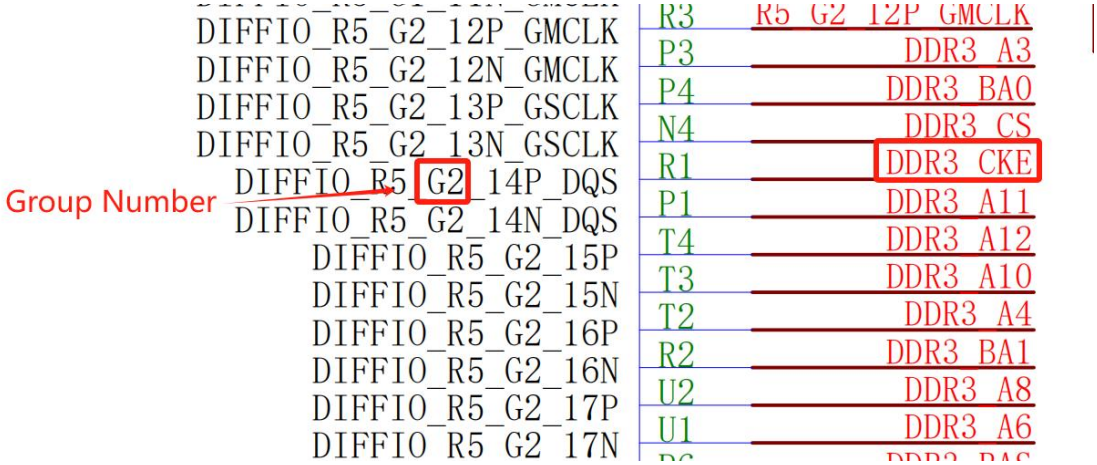


图 9.3-11

Step4 为概要, 点击 Generate 可生成 DDR3 IP;

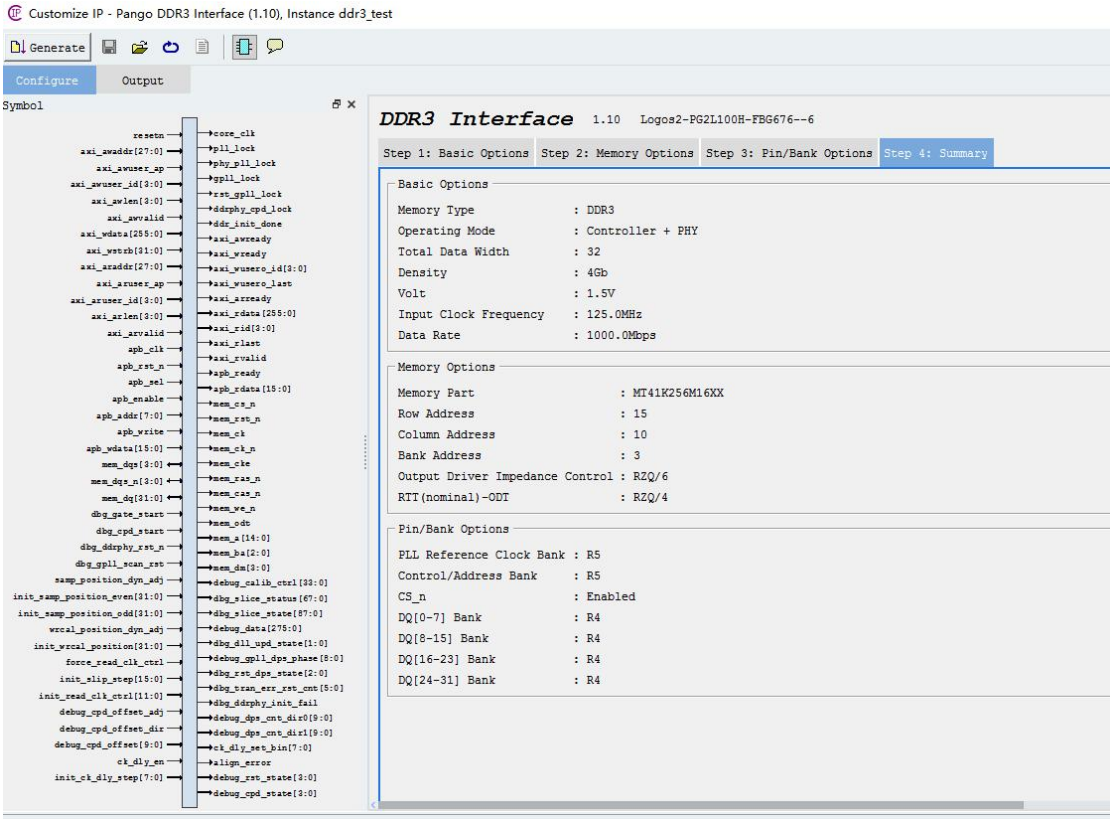

Customize IP - Pango DDR3 Interface (1.10), Instance ddr3_test
Generate
Configure
Output
Symbol
DDR3 Interface 1.10 Logos2-PG2L100H-FBG676--6
Step 1: Basic Options
Step 2: Memory Options
Step 3: Pin/Bank Options
Step 4: Summary
Basic Options
Memory Type : DDR3
Operating Mode : Controller + PHY
Total Data Width : 32
Density : 4Gb
Volt : 1.5V
Input Clock Frequency : 125.0MHz
Data Rate : 1000.0Mbps
Memory Options
Memory Part : MT41K256M16XX
Row Address : 15
Column Address : 10
Bank Address : 3
Output Driver Impedance Control : RZQ/6
RTT(nominal)-ODT : RZQ/4
Pin/Bank Options
PLL Reference Clock Bank : R5
Control/Address Bank : R5
CS_n : Enabled
DQ[0-7] Bank : R4
DQ[8-15] Bank : R4
DQ[16-23] Bank : R4
DQ[24-31] Bank : R4

图 9.3-12

关闭本工程, 在 IP 保存路径下打开 IP Example 工程:

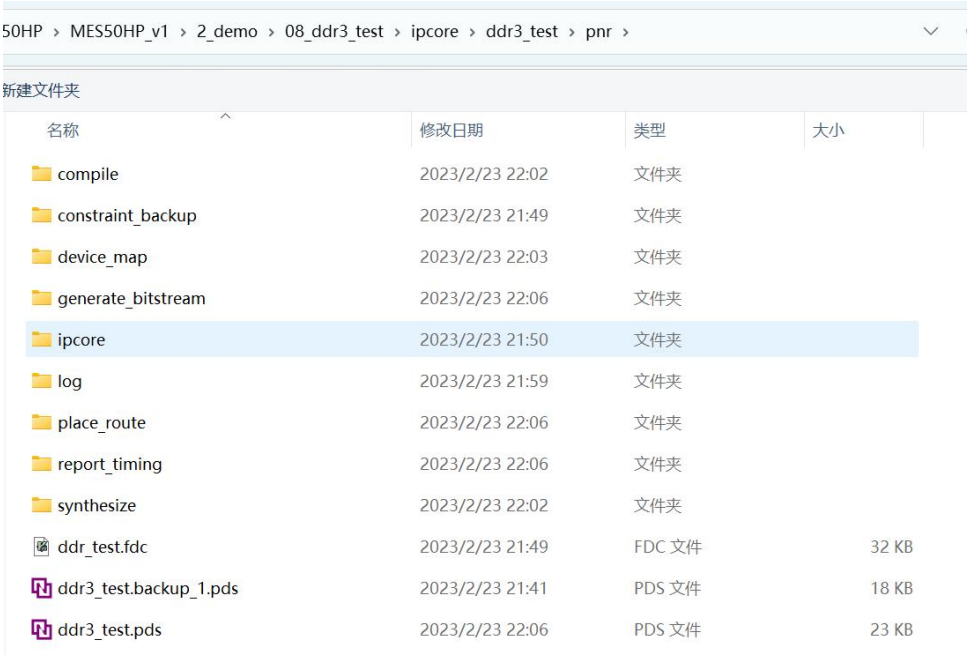


图 9.3-13

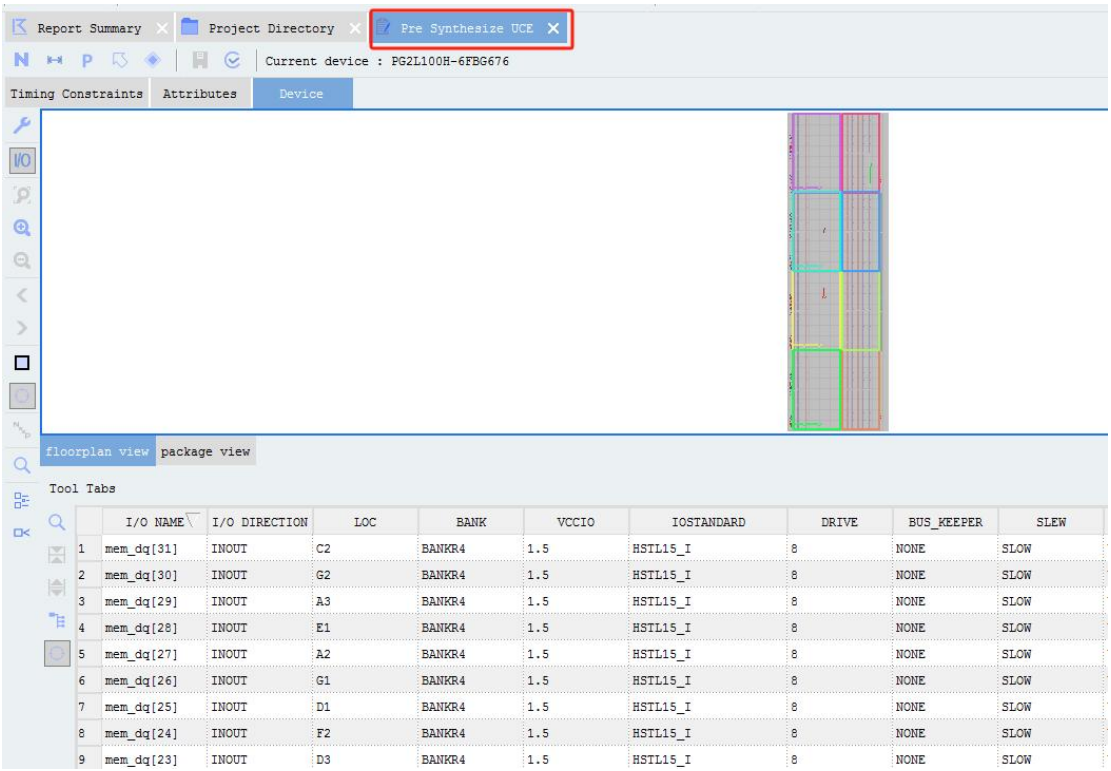
打开顶层文件 free_clk、ref_clk 可使用同一时钟源:

test ddr.v

```
19    parameter MEM_DM_WIDTH          = MEM_DQ_WIDTH/8,
20    parameter MEM_DQS_WIDTH         = MEM_DQ_WIDTH/8,
21    parameter CTRL_ADDR_WIDTH       = MEM_ROW_ADDR_WIDTH + MEM_BADDR_WIDTH + MEM_COL_
22  )(
23    input                                   ref_clk         ,
24    //input                                   free_clk        ,
25    input                                   rst_board       ,
26    output                                  pll_lock        ,
27    output                                  ddr_init_done   ,
28
29    //uart
30    input                                   uart_rxd        ,
31    output                                  uart_txd        ,
32
33    output                                  mem_rst_n       ,
34    output                                  mem_ck          ,
35    output                                  mem_ck_n        ,
36    output                                  mem_cke         ,
37
38    output                                  mem_cs_n        ,
39
40    output                                  mem_ras_n       ,
41    output                                  mem_cas_n       ,
42    output                                  mem_we_n        ,
43    output                                  mem_odt         ,
44    output      [MEM_ROW_ADDR_WIDTH-1:0]    mem_a           ,
45    output      [MEM_BADDR_WIDTH-1:0]       mem_ba          ,
46    inout       [MEM_DQS_WIDTH-1:0]         mem_dqs         ,
47    inout       [MEM_DQS_WIDTH-1:0]         mem_dqs_n       ,
48    inout       [MEM_DQ_WIDTH-1:0]          mem_dq          ,
49    output      [MEM_DM_WIDTH-1:0]          mem_dm          ,
50    output reg                              heart_beat_led  ,
51    output                                  err_flag_led
52
53
54  );
55  assign free_clk = ref_clk  ;
56
```

图 9.3-14

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:



	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD	DRIVE	BUS_KEEPER	SLEW
1	mem_dq[31]	INOUT	C2	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
2	mem_dq[30]	INOUT	G2	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
3	mem_dq[29]	INOUT	A3	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
4	mem_dq[28]	INOUT	E1	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
5	mem_dq[27]	INOUT	A2	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
6	mem_dq[26]	INOUT	G1	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
7	mem_dq[25]	INOUT	D1	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
8	mem_dq[24]	INOUT	F2	BANKR4	1.5	HSTL15_I	8	NONE	SLOW
9	mem_dq[23]	INOUT	D3	BANKR4	1.5	HSTL15_I	8	NONE	SLOW

图 9.3-15

以下管脚可约束在 LED, 方便观察实验现象;

ddr_init_done	OUTPUT	F13
err_flag_led	OUTPUT	E13
heart_beat_led	OUTPUT	E14
pll_lock	OUTPUT	E16

可按以下方式查看 IP 核的用户指南, 了解 Example 模块组成;

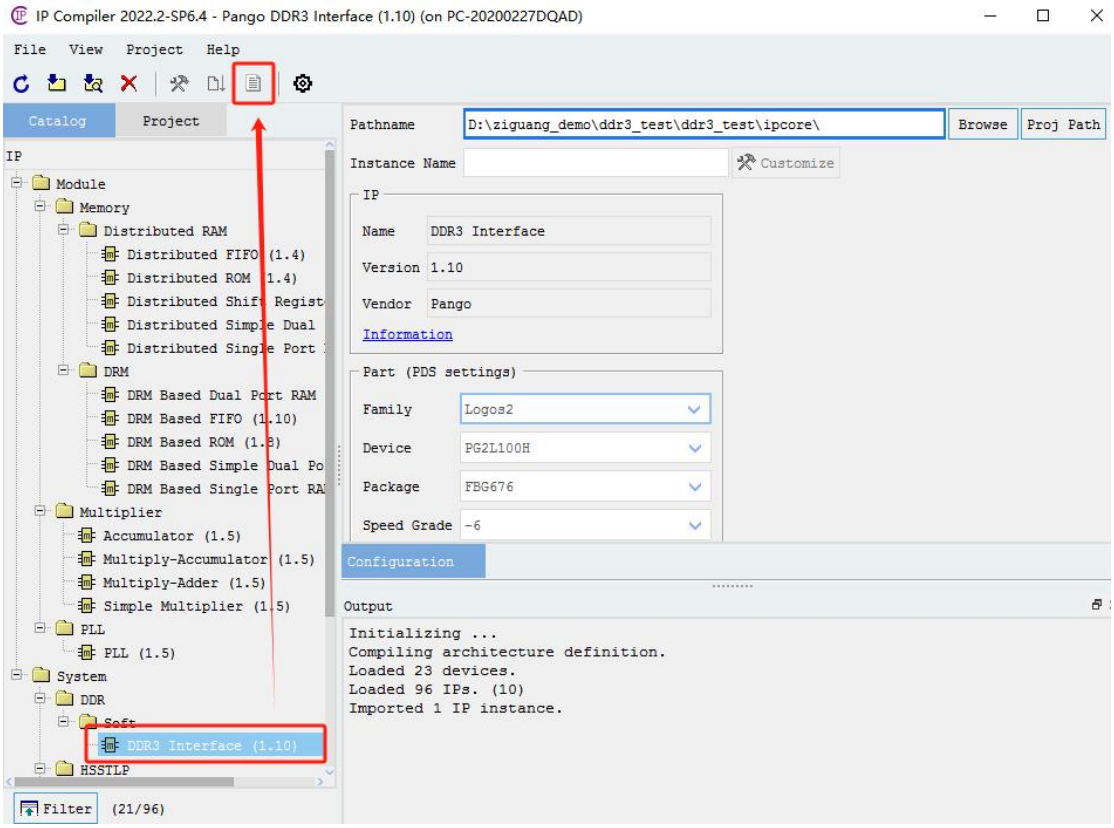

IP Compiler 2022.2-SP6.4 - Pango DDR3 Interface (1.10) (on PC-20200227DQAD)
File View Project Help
Catalog Project
IP
Module
Memory
Distributed RAM
Distributed FIFO (1.4)
Distributed ROM (1.4)
DRM
DRM Based Dual Port RAM
DRM Based FIFO (1.10)
Multiplier
Accumulator (1.5)
Multiply-Accumulator (1.5)
Multiply-Adder (1.5)
Simple Multiplier (1.5)
PLL
PLL (1.5)
System
DDR
Soft
DDR3 Interface (1.10)
HSSTLP
Filter (21/96)
Pathname D:\ziguang_demo\ddr3_test\ddr3_test\ipcore\
Browse
Proj Path
Instance Name
Customize
IP
Name DDR3 Interface
Version 1.10
Vendor Pango
Information
Part (PDS settings)
Family Logos2
Device PG2L100H
Package FBG676
Speed Grade -6
Configuration
Output
Initializing ...
Compiling architecture definition.
Loaded 23 devices.
Loaded 96 IPs. (10)
Imported 1 IP instance.

图 9.3-16

9.4. 实验现象

注: 例程位置: ddr3_test\ipcore\ddr3_test\pnr

下载程序, 可以看到 LED1 常亮, LED3 闪烁， LED4 闪烁, LED5 常亮;

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	3
heart_beat_led	心跳信号	4
pll_lock	Pll锁定指示	5

提醒:

err_flag_led 信号闪烁说明数据检测无错误。可在 IP 核数据手册中找到。

err_flag_led	O	1	数据检测错误信号。 闪烁：bist 正常运行，数据监测无错误。 1：数据检测出现错误，排除错误后需要手动清除。 0：bist 未能正常运行。 （Example Design 中产生的信号）

图 9.4-1

管脚约束过程需仔细核对每个信号对应的管脚。

10. HDMI 回环实验

10.1. 实验简介

实验目的:

完成 HDMI 回环实验

实验环境:

Window11

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

10.2. 实验原理

HDMI 输入接口采用宏晶微 MS7200 HMDI 接收芯片,HDMI 输出接口采用宏晶微 MS7210 HMDI 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。

MES2L100HP 开发板上将 MS7200 的 SA 管脚下拉到地, 故 IIC 的 ID 地址为 0x56, 将 MS7210 的 SA 管脚上拉到电源电压, 故 IIC 的 ID 地址为 0xB2。

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。

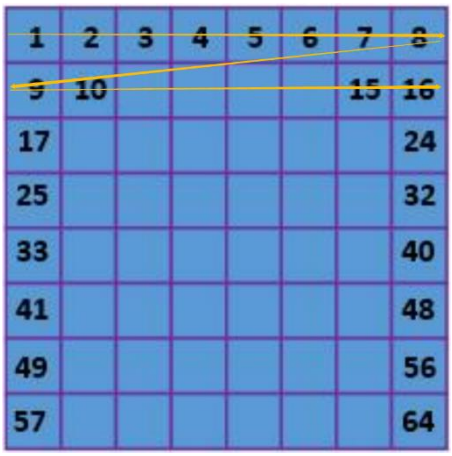


图 10.2-1

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 28=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。

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。

10.2.2. HDMI_PHY 配置

MS7200 为 HMDI 接收芯片, MS7210 为 HMDI 发送芯片, 芯片的 IIC 配置接口已与 FPGA 的 IO 相连, 芯片正常使用需要通过 FPGA 的对芯片进行初始化和配置操作。寄存器配置切忌修改。具体配置可以参考 demo。

10.3. 接口列表

顶层模块接口如下所示:

端口	I/O	位宽	描述
sys_clk	input	1	系统时钟
rstn_out	output	1	Hdmi_phy 复位信号
iic_scl	output	1	MS7200 SCL 信号
iic_sda	inout	1	MS7200 SDA 信号
iic_tx_scl	output	1	MS7210 SCL 信号
iic_tx_sda	inout	1	MS7210 SDA 信号
pixclk_in	input	1	HDMI_IN 像素时钟
vs_in	input	1	HDMI_IN 场信号
hs_in	input	1	HDMI_IN 行信号
de_in	input	1	HDMI_IN 像素有效信号
r_in	input	8	HDMI_IN 红色分量
g_in	input	8	HDMI_IN 绿色分量
b_in	input	8	HDMI_IN 蓝色分量
pixclk_out	output	8	HDMI_OUT 像素时钟输出
vs_ovut	output	1	HDMI_OUT 场信号
hs_out	output	1	HDMI_OUT 行信号
de_out	output	1	HDMI_OUT 像素有效信号
r_out	output	8	HDMI_OUT 红色分量
g_out	output	8	HDMI_OUT 绿色分量
b_out	output	8	HDMI_OUT 蓝色分量
led_int	output	1	Hdmi_phy 初始化完成信号

10.4. 工程说明

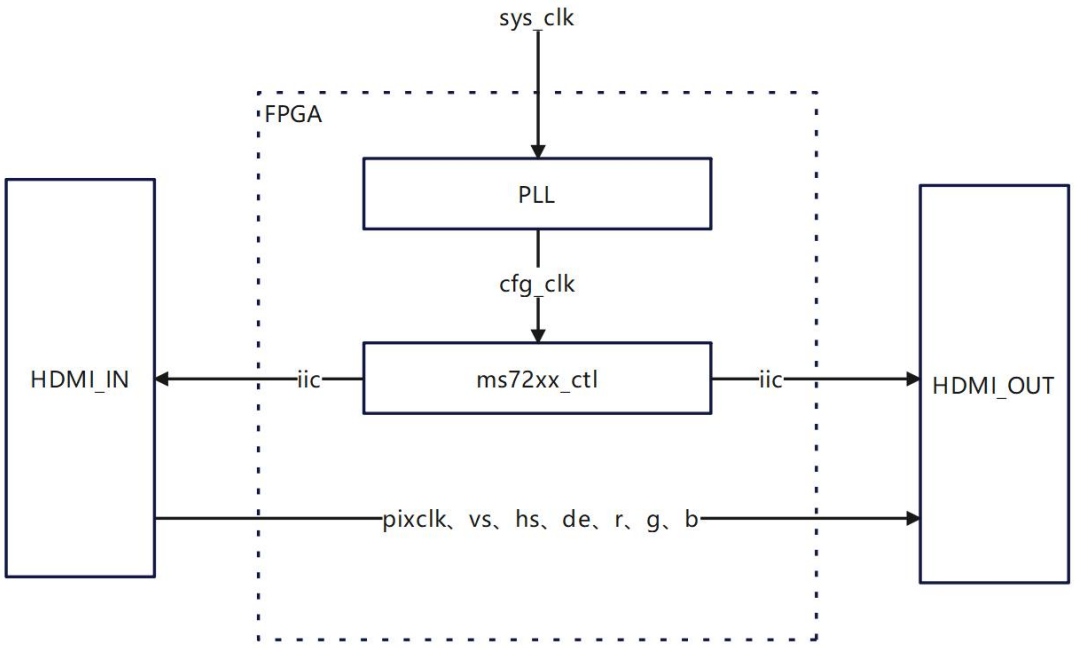


图 10.4-1

系统时钟 sys_clk 通过 pll 锁相环 IP 分出 cfg_clk 给到 ms72xx_ctl 模块，该模块通过 IIC 配置 MS7200 与 MS7210 芯片。HDMI 输入的数据经过 MS7200 芯片解码后转成标准 RGB 格式的数据输入到 FPGA 中。再将该数据输出到 MS7210 芯片。MS7210 将 RGB 格式的数据编码成 TMDS 信号对外输出。

10.5. 代码模块说明

Ms72xx_ctl.v 模块

```
module ms72xx_ctl(
    input       clk,
    input       rst_n,

    output      init_over,
    output      iic_tx_scl,
    inout       iic_tx_sda,
    output      iic_scl,
    inout       iic_sda
);
    reg rstn_temp1,rstn_temp2;
    reg rstn;
    always @(posedge clk or negedge rst_n)
    begin
        if(!rst_n)
            rstn_temp1 <= 1'b0;
        else
            rstn_temp1 <= rst_n;
    end

    always @(posedge clk)
    begin
        rstn_temp2 <= rstn_temp1;
        rstn <= rstn_temp2;
    end

    wire        init_over_rx;
    wire   [7:0] device_id_rx;
    wire        iic_trig_rx ;
    wire        w_r_rx      ;
    wire  [15:0] addr_rx     /*synthesis PAP_MARK_DEBUG="true"*/;
    wire  [ 7:0] data_in_rx  ;
    wire        busy_rx     ;
    wire  [ 7:0] data_out_rx ;
```

```
wire        byte_over_rx;

wire   [7:0] device_id_tx;
wire        iic_trig_tx ;
wire        w_r_tx      ;
wire  [15:0] addr_tx     /*synthesis PAP_MARK_DEBUG="true"*/;
wire  [ 7:0] data_in_tx  ;
wire        busy_tx     ;
wire  [ 7:0] data_out_tx ;
wire        byte_over_tx;



ms7200_ctl ms7200_ctl(
    .clk          (  clk         ),//input
    .rstn         (  rstn        ),//input

    .init_over    (  init_over_rx  ),//output reg
    .device_id    (  device_id_rx  ),//output      [7:0]
    .iic_trig     (  iic_trig_rx   ),//output reg
    .w_r          (  w_r_rx        ),//output reg
    .addr         (  addr_rx       ),//output reg   [15:0]
    .data_in      (  data_in_rx    ),//output reg   [ 7:0]
    .busy         (  busy_rx       ),//input
    .data_out     (  data_out_rx   ),//input       [ 7:0]
    .byte_over    (  byte_over_rx  ) //input
);

ms7210_ctl ms7210_ctl(
    .clk          (  clk         ),//input
    .rstn         (  init_over_rx  ),//input    rstn),//

    .init_over    (  init_over     ),//output reg
    .device_id    (  device_id_tx  ),//output      [7:0]
    .iic_trig     (  iic_trig_tx   ),//output reg
    .w_r          (  w_r_tx        ),//output reg
```

```
    .addr         ( addr_tx     ),//output reg  [15:0]
    .data_in      ( data_in_tx  ),//output reg  [ 7:0]
    .busy         ( busy_tx     ),//input
    .data_out     ( data_out_tx ),//input      [ 7:0]
    .byte_over    ( byte_over_tx ) //input
);


wire       sda_in/*synthesis PAP_MARK_DEBUG="true"*/;
wire       sda_out/*synthesis PAP_MARK_DEBUG="true"*/;
wire       sda_out_en/*synthesis PAP_MARK_DEBUG="true"*/;
iic_dri #(
    .CLK_FRE      ( 27'd10_000_000 ),//parameter        CLK_FRE   = 27'd50_000_000,//system clock frequency
    .IIC_FREQ     ( 20'd400_000    ),//parameter        IIC_FREQ  = 20'd400_000,   //I2c clock frequency
    .T_WR         ( 10'd1          ),//parameter        T_WR      = 10'd5,         //I2c transmit delay ms
    .ADDR_BYTE    ( 2'd2           ),//parameter        ADDR_BYTE = 2'd1,          //I2C addr byte number
    .LEN_WIDTH    ( 8'd3           ),//parameter        LEN_WIDTH = 8'd3,          //I2C transmit byte width
    .DATA_BYTE    ( 2'd1           ) //parameter        DATA_BYTE = 2'd1           //I2C data byte number
)iic_dri_rx(
    .clk          ( clk          ),//input            clk,
    .rstn         ( rstn         ),//input            rstn,
    .device_id    ( device_id_rx ),//input            device_id,
    .pluse        ( iic_trig_rx  ),//input            pluse,              //I2C transmit trigger
    .w_r          ( w_r_rx       ),//input            w_r,                //I2C transmit direction 1:send  0:receive
    .byte_len     ( 4'd1         ),//input  [LEN_WIDTH:0] byte_len,       //I2C transmit data byte length of once trigger

    .addr         ( addr_rx      ),//input  [7:0]     addr,               //I2C transmit addr
    .data_in      ( data_in_rx   ),//input  [7:0]     data_in,            //I2C send data
```

```
    .busy         ( busy_rx       ),//output reg         busy=0,              //I2C bus status

    .byte_over    ( byte_over_rx  ),//output reg         byte_over=0,         //I2C byte transmit o
ver flag
    .data_out     ( data_out_rx   ),//output reg[7:0]    data_out,            //I2C receive data

    .scl          ( iic_scl       ),//output             scl,
    .sda_in       ( sda_in        ),//input              sda_in,
    .sda_out      ( sda_out       ),//output   reg       sda_out=1'b1,
    .sda_out_en   ( sda_out_en    ) //output             sda_out_en
  );

  assign iic_sda = sda_out_en ? sda_out : 1'bz;
  assign sda_in = iic_sda;

  wire       sda_tx_in/*synthesis PAP_MARK_DEBUG="true"*/;
  wire       sda_tx_out/*synthesis PAP_MARK_DEBUG="true"*/;
  wire       sda_tx_out_en/*synthesis PAP_MARK_DEBUG="true"*/;
  iic_dri #(
    .CLK_FRE      ( 27'd10_000_000 ),//parameter        CLK_FRE   = 27'd50_000_000,//system cl
ock frequency
    .IIC_FREQ     ( 20'd400_000    ),//parameter        IIC_FREQ  = 20'd400_000,   //I2c clock freq
uency
    .T_WR         ( 10'd1          ),//parameter        T_WR      = 10'd5,         //I2c transmit delay ms
    .ADDR_BYTE    ( 2'd2           ),//parameter        ADDR_BYTE = 2'd1,          //I2C addr byte nu
mber
    .LEN_WIDTH    ( 8'd3           ),//parameter        LEN_WIDTH = 8'd3,          //I2C transmit byte
width
    .DATA_BYTE    ( 2'd1           ) //parameter        DATA_BYTE = 2'd1           //I2C data byte num
ber
  )iic_dri_tx(
    .clk          ( clk            ),//input            clk,
    .rstn         ( rstn           ),//input            rstn,
    .device_id    ( device_id_tx   ),//input            device_id,
    .pluse        ( iic_trig_tx    ),//input            pluse,               //I2C transmit trigger
```

```
    .w_r         ( w_r_tx       ),//input          w_r,              //I2C transmit direction 1:send  0:
receive
    .byte_len    ( 4'd1         ),//input  [LEN_WIDTH:0] byte_len,          //I2C transmit data byt
e length of once trigger

    .addr        ( addr_tx      ),//input  [7:0]       addr,           //I2C transmit addr
    .data_in     ( data_in_tx   ),//input  [7:0]       data_in,        //I2C send data

    .busy        ( busy_tx      ),//output reg         busy=0,         //I2C bus status

    .byte_over   ( byte_over_tx ),//output reg         byte_over=0,    //I2C byte transmit o
ver flag
    .data_out    ( data_out_tx  ),//output reg[7:0]    data_out,       //I2C receive data

    .scl         ( iic_tx_scl   ),//output          scl,
    .sda_in      ( sda_tx_in    ),//input           sda_in,
    .sda_out     ( sda_tx_out   ),//output  reg     sda_out=1'b1,
    .sda_out_en  ( sda_tx_out_en ) //output         sda_out_en
  );

  assign iic_tx_sda = sda_tx_out_en ? sda_tx_out : 1'bz;
  assign sda_tx_in = iic_tx_sda;


endmodule
```

该模块主要完成对 ms7210 和 ms7200 芯片的 IIC 配置。具体代码不详细分析，按照 IIC 协议去完成即可，然后就是配置 ms7210 和 ms7200 的寄存器即可。以下给出 IIC 协议的时序说明，

如下图所示:

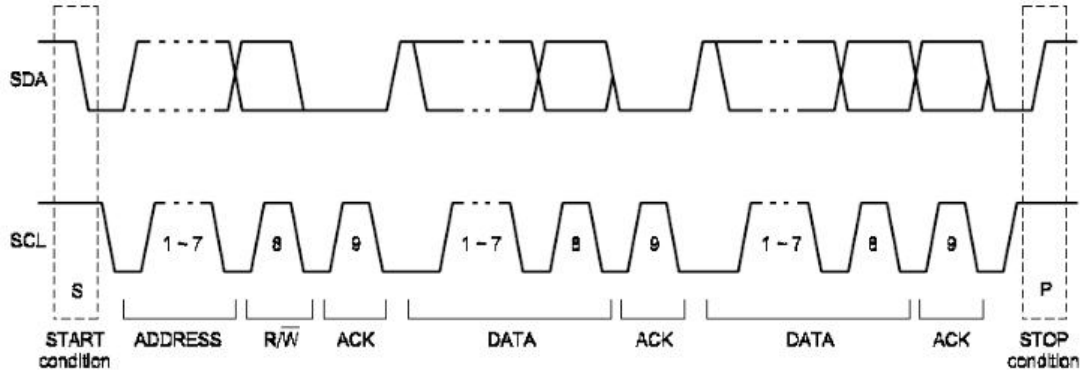


图 10.5-1

当 SDA 和 SCL 都处于高电平时, 表示空闲状态, 因此在硬件设计时其实也要通过上拉电阻将信号拉高, 表示空闲。当 SCL 保持高电平时, SDA 拉低时表示 IIC 总线启动, 标志一次数据传输的开始, 在开始前, IIC 总线必须处于空闲状态。在数据传输过程中, SDA 上逐位串行传输每一位数据, 在 SCL 高电平期间, SDA 必须保持稳定。当 SCL 为低电平时, SDA 才允许改变。

IIC 总线每次传输一个字节, 也就是传输 8 个 bit, 每 8bit 后要收到一个来自接收方反馈的 ACK(应答信号), 该应答信号为高电平时, 表示接收方接收字节失败。为低电平时, 表示有效。之后, 在 SCL 高电平期间, SDA 重新拉高, 则表示传输结束, IIC 总线回到空闲状态。

10.6. 实验现象

连接好 MES2L100HP-MINI 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。

图 10.6-2

图 10.6-3

上图为 FPGA 的 HDMI_OUT 输出的图像。

注意事项:

需要插上 HDMI 接口, 板子上电时才会对 HDMI 芯片进行配置, 否则将配置失败。

我们对 HDMI 芯片配置时使用的是 RGB888 协议, 如果视频源的颜色格式不是 RGB888, 会出现一些问题。如上图, 视频源的颜色格式是 YCbCr444, 回环实验中, 在显示器上显示视频源的画面, 显示器上的画面颜色会偏绿。

11. 基于 UDP 的以太网传输实验例程

11.1. 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

PDS2022.2-SP6.4

硬件环境:

MES2L676-100HP-MINI

11.2. 开发板以太网接口简介

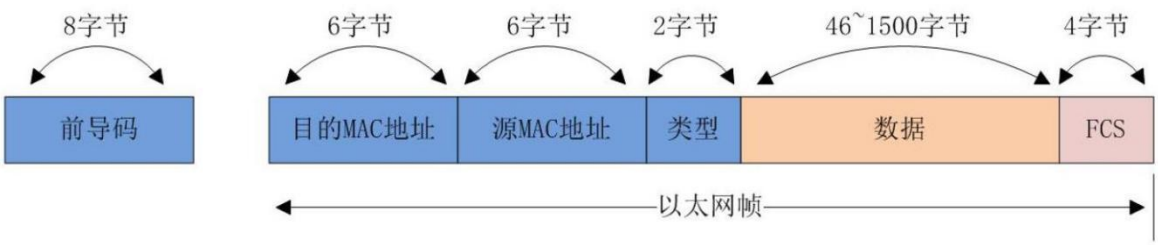
MES2L676-100HP-MINI-MINI 开发板使用 Realtek RTL8211F PHY 实现了一个 10/100/1000 以太网端口, 用于网络连接。该器件工作电压为支持 2.5V、3.3V, 通过 RGMII 接口连接到 PG2L100H。RJ-45 连接器是 HFJ11-1G01E-L12RL, 具有集成的自动缠绕磁性元件, 可提高性能, 质量和可靠性。 RJ-45 有两个状态指示灯 LED, 用于指示流量和有效链路状态 (详情请查看“MES50HP 开发板硬件使用手册”)。

11.3. 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。

11.4. 以太网协议简介

11.4.1. 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;

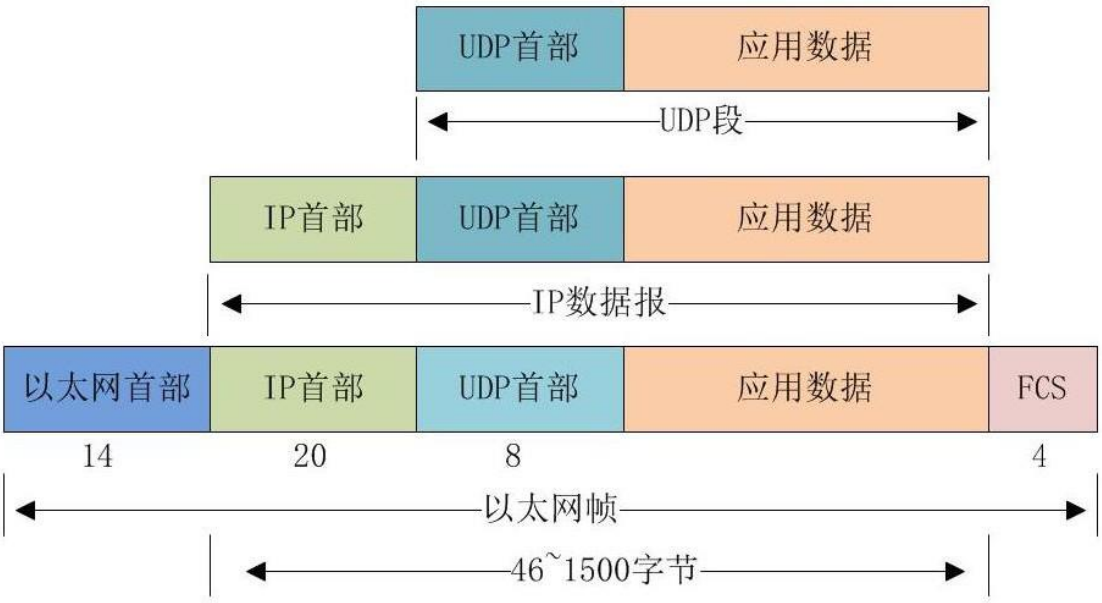
源 MAC 地址: 6 字节, 存放发送端设备的物理地址;

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 ARP 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。

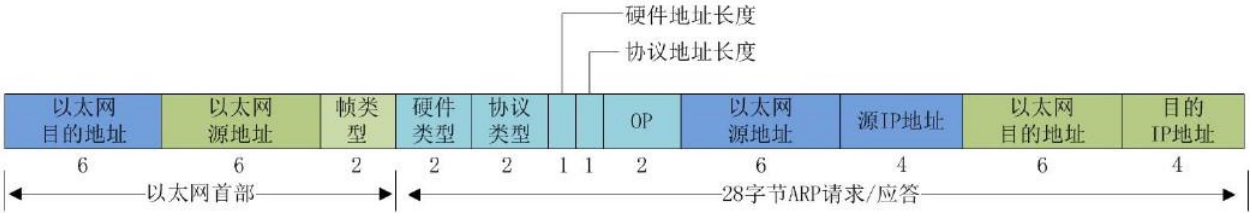
FCS: 帧尾, 4 字节, 称为帧校验序列, 采用 32 位 CRC 校验, 对目的 MAC 地址字段到数据字段进行校验。

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。



11.4.2. 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。

11.4.3. 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;

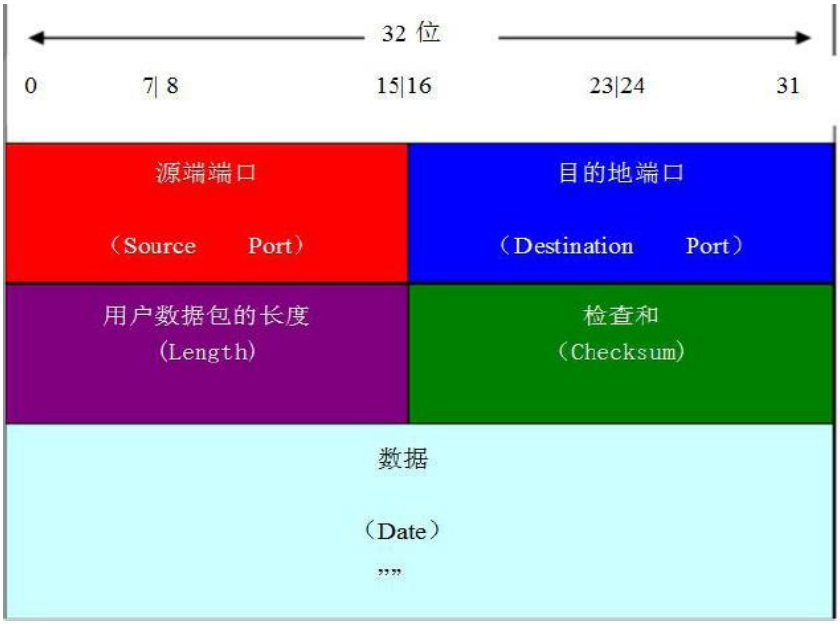
源地址和目的地址:都各占 4 字节,分别记录源地址和目的地址;

11.4.4. 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

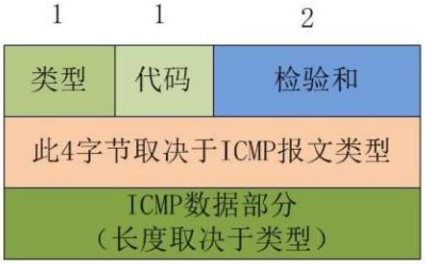
算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些

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。

11.4.5. 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

11.5. 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 ManagementInterface),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

11.5.1. 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

11.5.2. 读时序

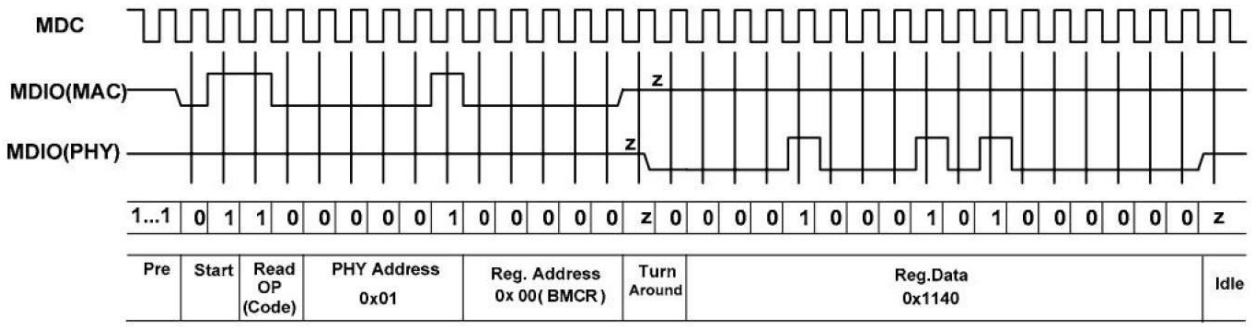


Figure 8. MDC/MDIO Read Timing

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。

11.5.3. 写时序

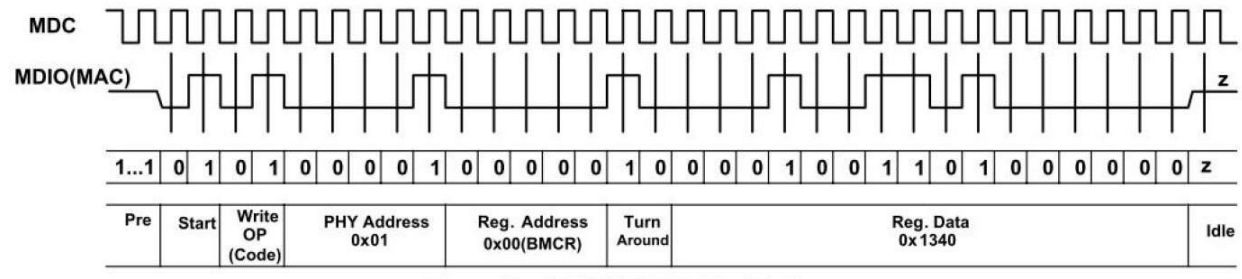


Figure 9. MDC/MDIO Write Timing

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。

11. 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。

11.6.1. 发送部分

11.6.1.1. 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 CRC32 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

11.6.1.2. MAC 发送模式

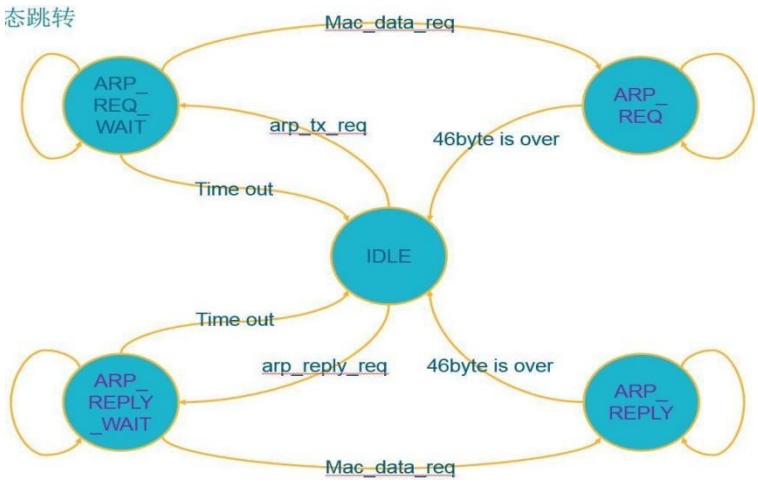
工程中的 mac_tx_mode.v 为发送模式选择, 根据发送模式是 IP 或 ARP 选择相应的信号与数据。

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

11.6.1.3. 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

11.6.1.4. 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

11.6.1.5. 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

11.6.1.6. 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)

11.6.2. 接收部分

11.6.2.1. 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

11.6.2.2. 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

11.6.2.3. 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_HEADER1 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

11.6.2.4. 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

11.6.3. 其他部分

11.6.3.1. 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

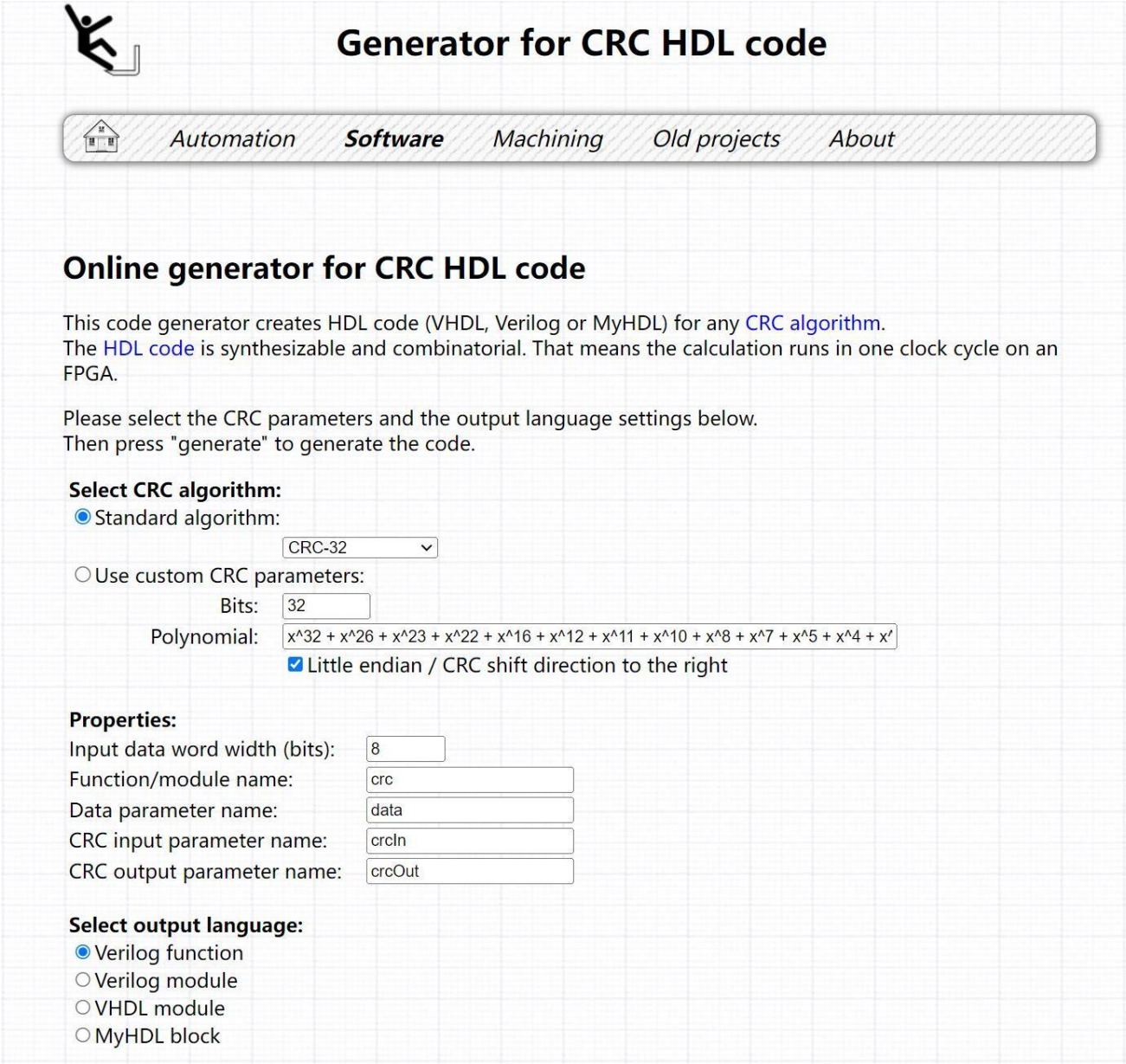
11.6.3.2. ARP 缓存

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

11.6.3.3. 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



11.6.4. 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:

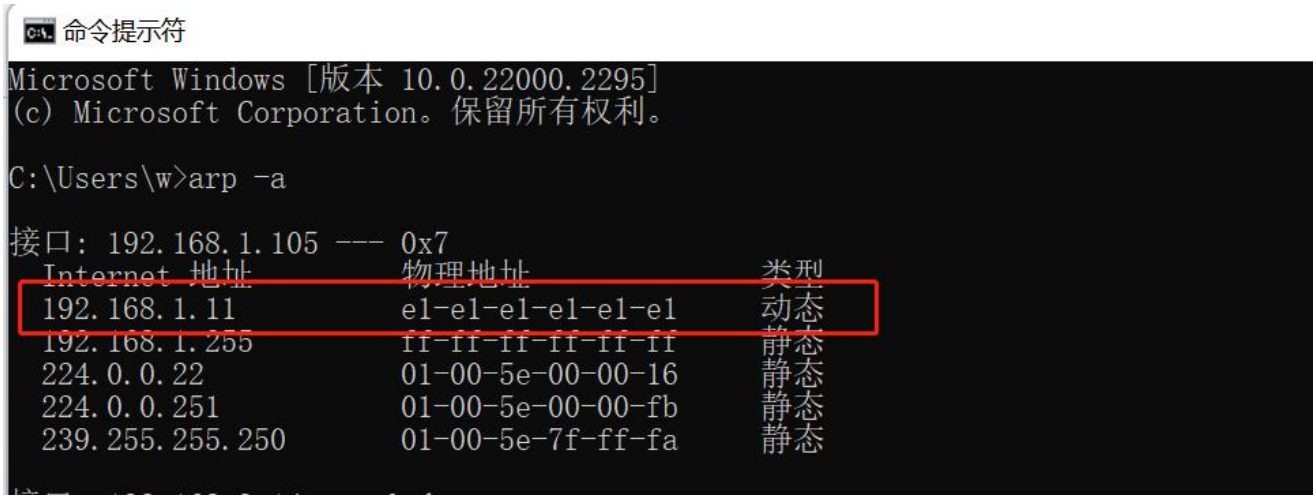
```
module ethernet1_test#(
    parameter       LOCAL_MAC = 48'he1_e1_e1_e1_e1_e1,
    parameter       LOCAL_IP  = 32'hC0_A8_01_0B,//192.168.1.11
    parameter       LOCL_PORT = 16'h1F90,

    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:

	LED 编号		参考说明
	1		闪烁

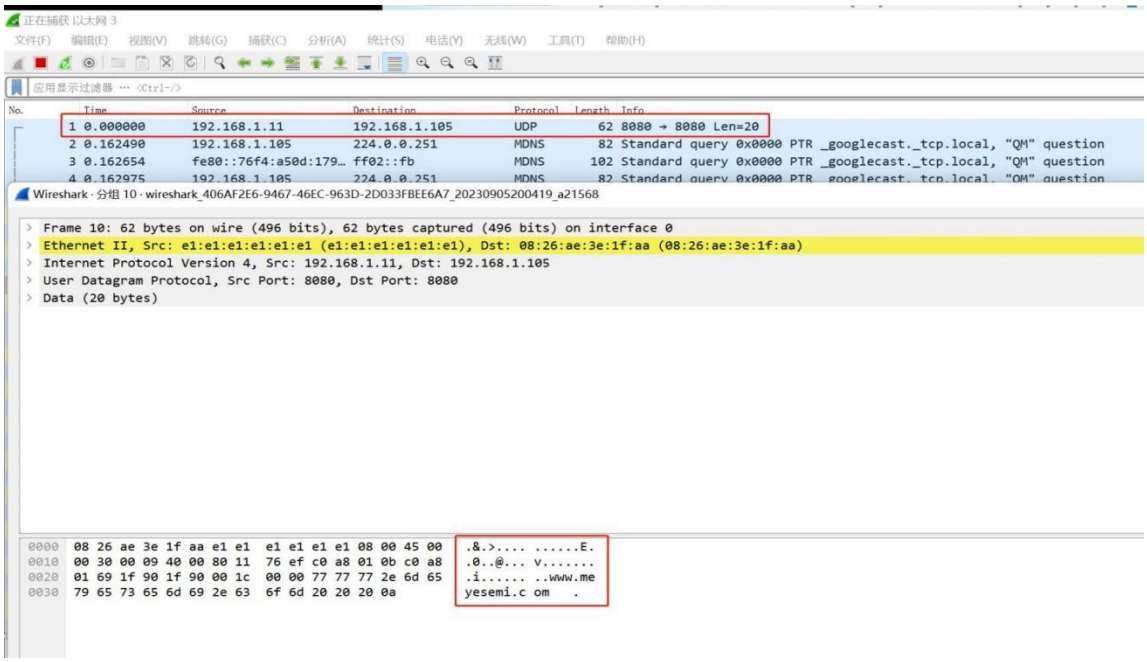
通过命令提示符, 输入 arp -a, 可以查到 IP: 192.168.1.11 MAC: e1_e1_e1_e1_e1_e1;



通过 Wireshark 软件抓包验证数据链路是否正常连接以及数据传输是否正常。资料包中 Wireshark 安装包目录如下:

MES50HP \5_Software\网络调试助手\Wireshark-win32-2.4.1.0.exe

PC 端打开 Wireshark 软件, 烧录重新后进行捕获数据报可以看到如下所示的交互过程。



成功建立连接后会持续发送数据报“www.meyesemi.com”。