小眼睛科技

提供一站式 FPGA&嵌入式解决方案

Kosmo 之 PA 实验指导手册

Kosmo 系列（PG2K100-6IMBG400）

开发板实验教程

紫光同创 Kosmo 系列开发平台

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： 442106123QQ3 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/
技术论坛

*扫码注册开源技术论

*扫一扫关注官微

* 官方旗舰

目录

1. FPGA 开发工具使用 7
1.1. 实验简介 7
1.2. 实验原理 7
1.2.1. PDS 软件的安装 7
1.2.1.1. 软件简介 7
1.2.1.2. 支持平台 7
1.2.1.3. 软件安装 7
1.2.1.4. 注意事项 7
1.2.1.5. 安装程序 8
1.2.1.6. License 关联，环境变量设置 10
1.2.2. PDS 工具的使用 14
2. Modelsim 的使用和 do 文件编写 14
2.1.1.1. 实验简介 14
2.1.1.2. 实验原理 14
2.1.1.3. 接口列表 14
2.1.2. Testbench 文件的编写 15
2.1.3. Modelsim 的使用 16
2.1.3.1. 文件的编写 27
2.1.3.2. 基本命令介绍 27
2.1.3.3. 文件示例 28
3. Pango 与 Modelsim 的联合仿真 32
3.1. 实验简介 32
3.2. 实验原理 32
3.2.1. 编译仿真库 32
3.2.2. 设置仿真路径 34
3.2.3. 启动联合仿真 36
4. 紫光同创 IP core 的使用及添加 38
4.1. 实验简介 38
4.2. 实验原理 38
4.2.1. IP 的安装 38
4.2.2. 例化 IP 及查看 IP 手册 41
5. Pango 的时钟资源——锁相环 46
5.1. 实验目的 46
5.2. 实验原理 46
5.2.1. PLL 介绍 46
5.2.2. IP 配置 46

5.2.3. 代码设计.....49
5.2.4. PDS 与 Modelsim 联合仿真.....51
5.2.5. 实验现象.....52
6. Pango 的 ROM、RAM、FIFO 的使用.....54
6.1. 实验简介.....54
6.2. 实验原理.....54
6.2.1. RAM 介绍.....54
6.2.1.1. RAM 的读写时序.....56
6.2.1.2. ROM 的读时序.....61
6.2.2. FIFO 介绍.....61
6.2.2.1. 的读写时序.....64
6.3. 接口列表.....66
6.4. 工程说明.....67
6.5. 代码仿真说明.....67
6.5.1. RAM 仿真测试.....67
6.5.2. ROM 仿真测试.....70
6.5.3. FIFO 仿真测试.....71
7. 基于紫光 FPGA 的键控 LED 流水灯.....75
7.1. 实验简介.....75
7.2. 实验原理.....75
7.3. 接口列表.....76
7.4. 工程说明.....76
7.4.1. 代码模块说明.....76
7.4.2. 代码仿真.....78
7.5. 实验步骤.....79
7.5.1. 打开 PDS 软件，创建工程.....79
7.5.2. 添加设计文件.....84
7.5.3. 编译.....88
7.5.4. 工程约束.....89
7.5.4.1. 时钟约束.....90
7.5.4.2. 物理约束.....91
7.5.5. 综合.....91
7.5.6. Device Map.....92
7.5.7. Place & Route.....93
7.5.8. Generate Bitstream.....94
7.5.9. 下载生成的位流文件.....94
8. HDMI 回环实验.....97
8.1. 实验简介.....97

8.2. 实验原理.....97
8.2.1. 显示原理.....97
8.3. 接口列表.....99
8.4. 工程说明.....100
8.5. 代码模块说明.....101
8.6. 实验现象.....107
9. 音频回环实验.....110
9.1. 实验简介.....110
9.2. 实验原理.....110
9.2.1. 音频原理.....111
9.3. 接口列表.....112
9.4. 工程说明.....113
9.5. 代码模块说明.....114
9.6. 实验现象.....117

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。

实验环境：

Window11

PDS2025.2

硬件环境：

PG2K100-6IMBG400

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_2025.2 版本可将软件安装在 C:\pango\PDS_2025.2（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 Suite2025；在程序菜单 Pango Design Suite2025.2 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 2025.2 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.注意事项

软件安装过程中有如下注意事项：

1）软件建议安装我们推荐的版本，否则版本差异会导致一些奇怪的问题；

2）开始安装软件前，关闭电脑的杀毒软件以及防火墙；

3）开始安装软件前，检查个人电脑账户名称是否是中文或特殊字符，如果含有这两个其中任意的一个，可能会导致软件安装失败或者安装完成也不能正常工作；

4）安装软件的路径可以自定义，但是自定义的路径不能出现中文还有特殊字符，否则会导致软件安装失败（建议使用默认安装否则会有部分步骤操作与文档不同）；

5）FPGA 开发工具安装时比较耗时，需要耐心等待，中间不能出现强行终止或者意外中断，否则只能重新进行安装。

1.2.1.5.安装程序

将压缩包解压出来（注意：解压目录的路径名称只能够包含字母、数字、下划线，不要出现中文与特殊字符，否则安装程序有可能出问题），为避免在安装过程中出错，在开始安装之前，请先关闭安全或杀毒软件。双击解压出来的文件夹下的“setup.exe”，开始安装 PDS 软件，如下图所示：

首先右击解压安装包，打开解压后安装包文件夹，双击安装包中的安装程序 Setup.exe，启动安装程序：

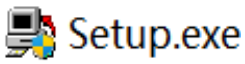


图 1.2-1

启动安装程序后的界面如下：

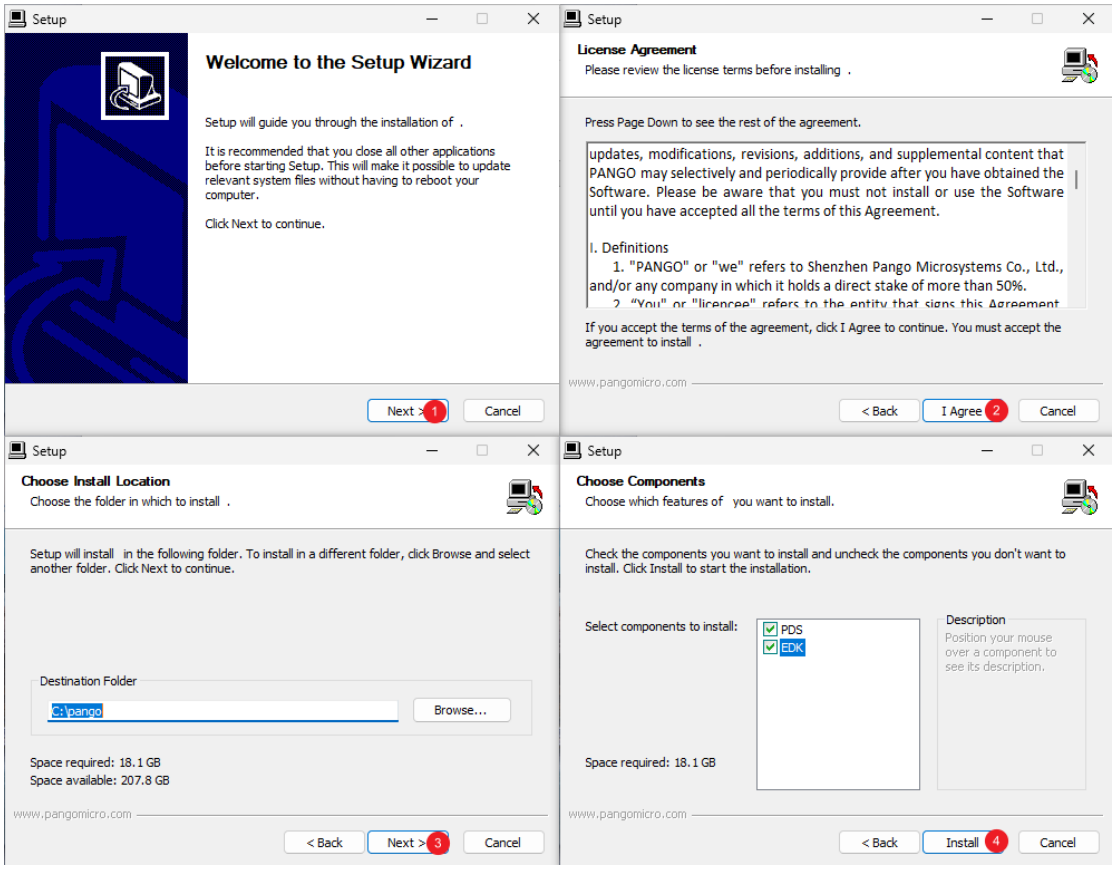


图 1.2-2

如图所示，默认安装路径为 C:\pango\PDS_2025.2，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。注意如果使用裸机开发需要勾选 EDK 然后点击“Install”。

点击“Install”后则跳转到安装界面。

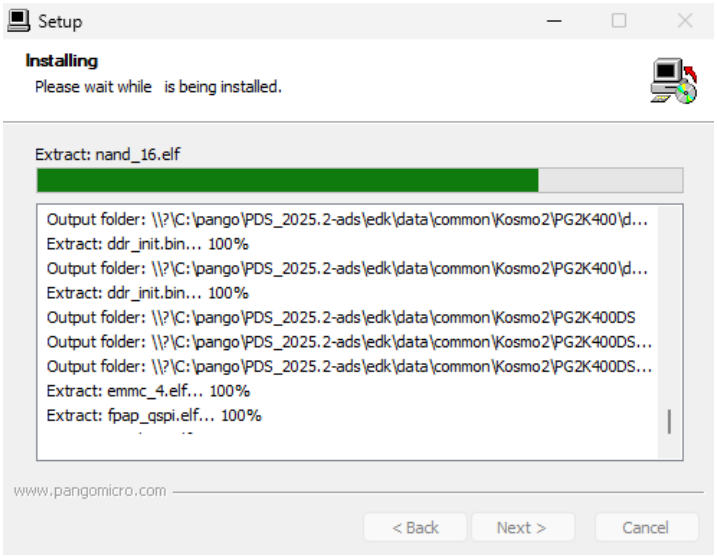


图 1.2-6

耐心等待至完成全部安装过程，点击“Finish”，安装完毕。

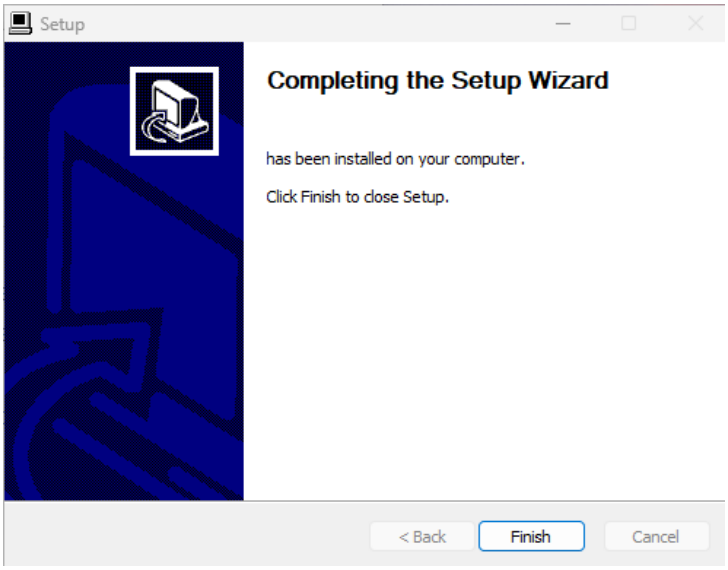


图 1.2-7 点击 Finish

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 1.2-11

结束安装。在桌面上看到如下图标：

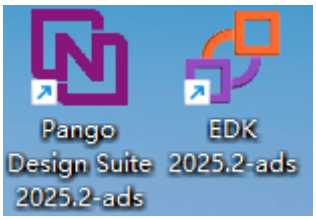


图 1.2-3

1.2.1.6.License 关联，环境变量设置

软件完成安装后，PangoDesignSuite 需要 License 文件才能正常使用，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。（教学中以 C:\pango\PDS_2025.2-ads\lic 为 license 存放文件夹）

如果使用客服获取提供的通用 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：

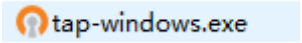


图 1.2-4

双击运行，全部保持默认。

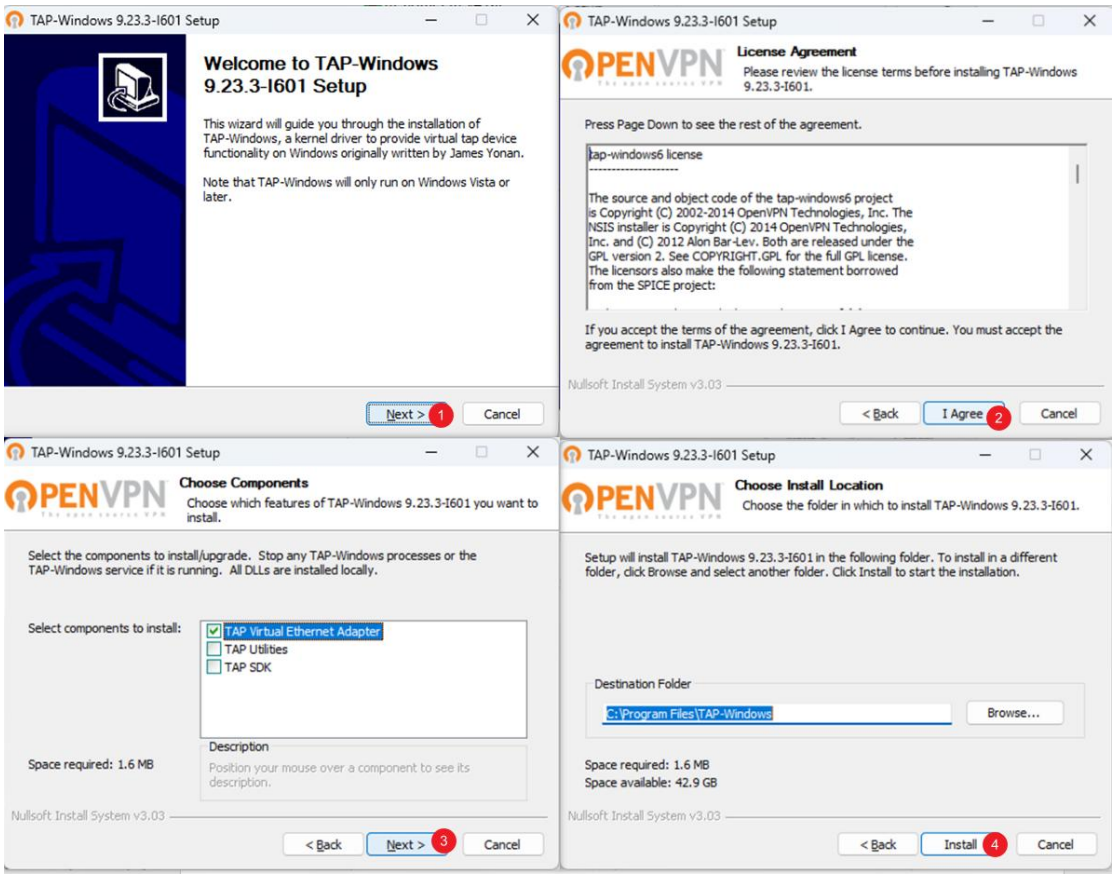


图 1.2-5

安装路径保持默认，点击 Install，等待安装完成。

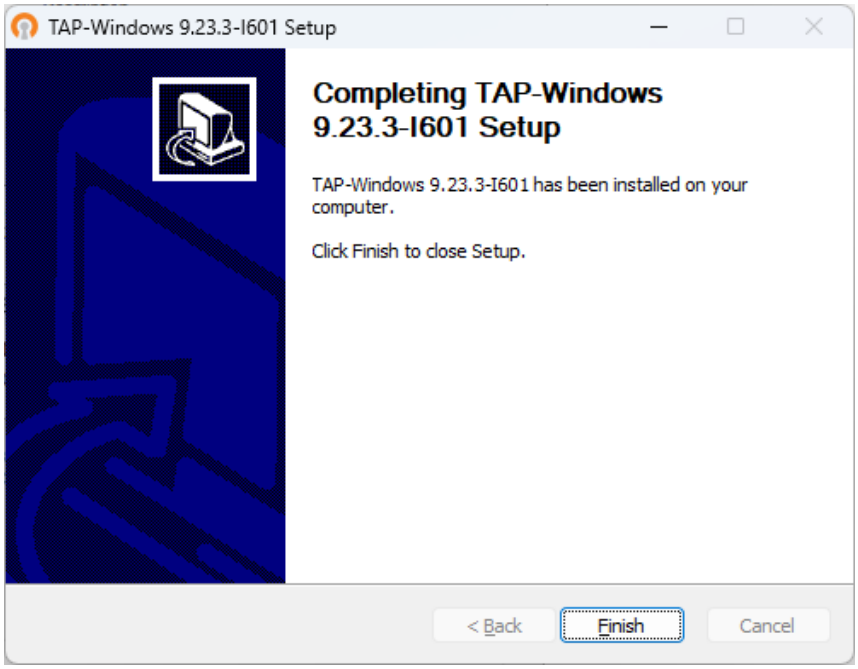
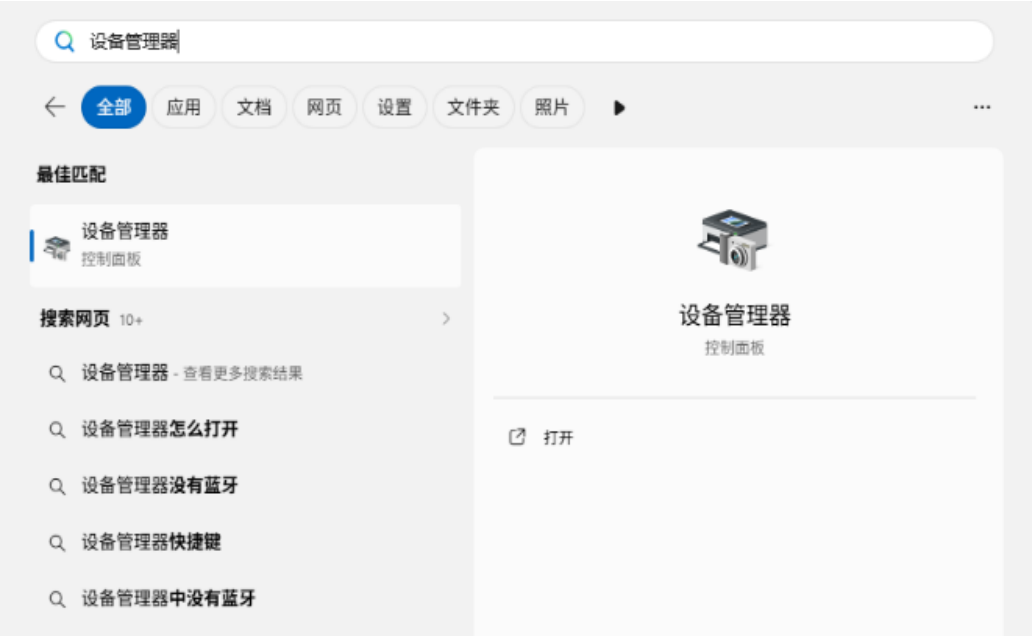


图 1.2-6 点击 Finish 安装完成。

安装 tap-windows 软件完成后仍需要对 TAP 虚拟网卡做配置修改，点击 Windows 按钮搜索设备管理器



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

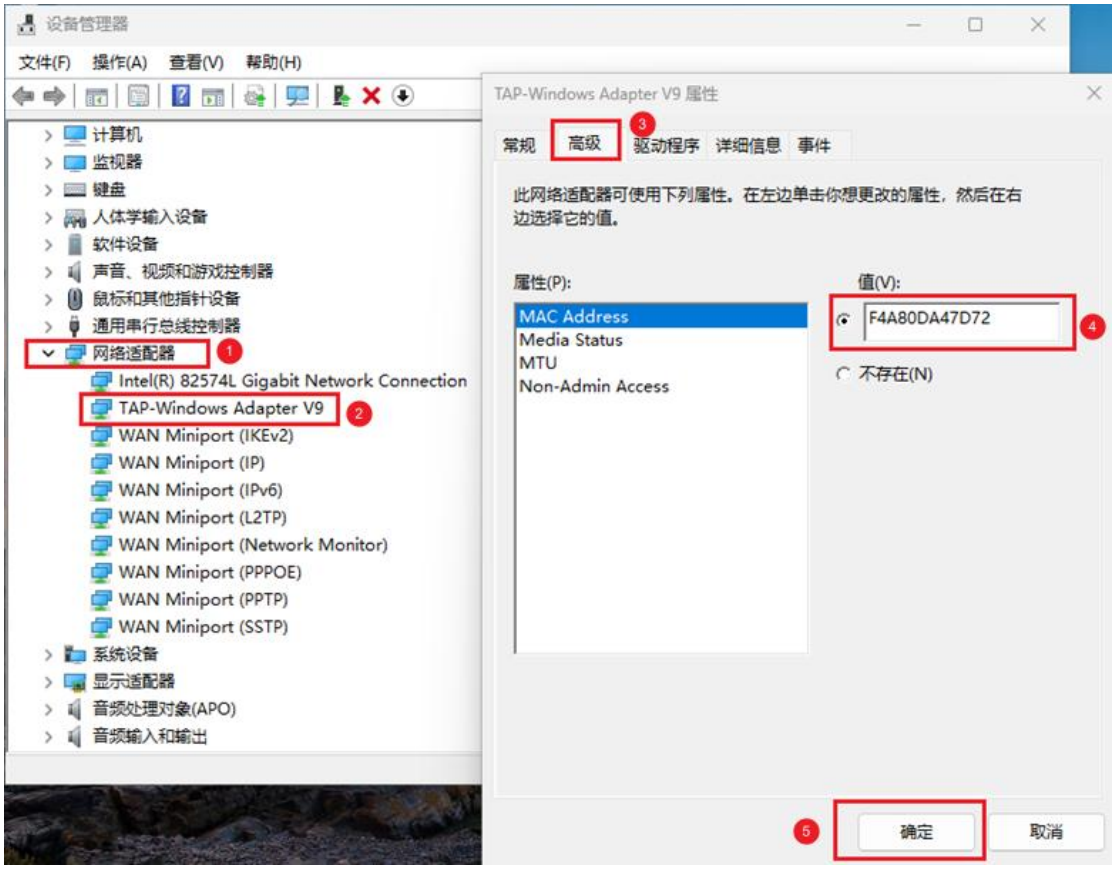
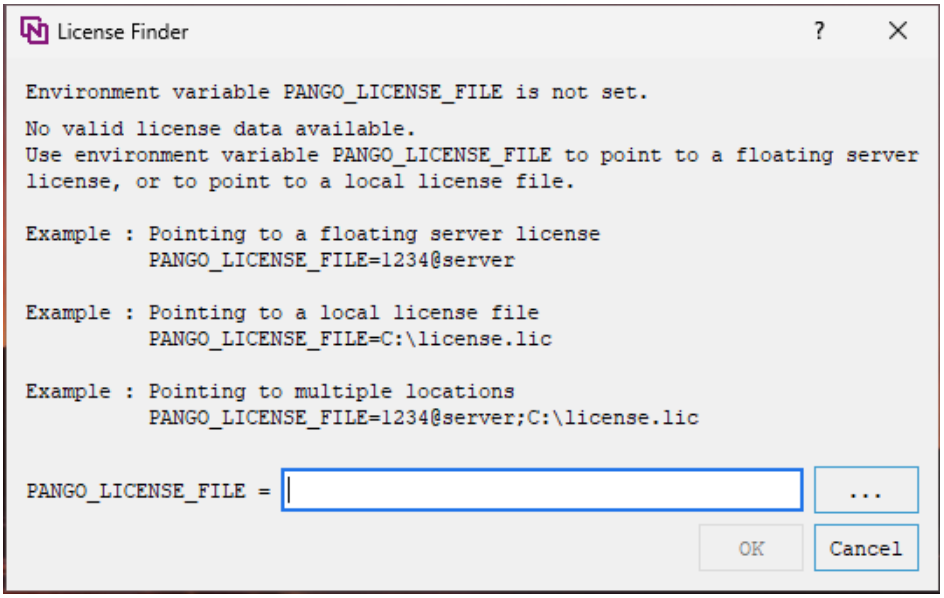


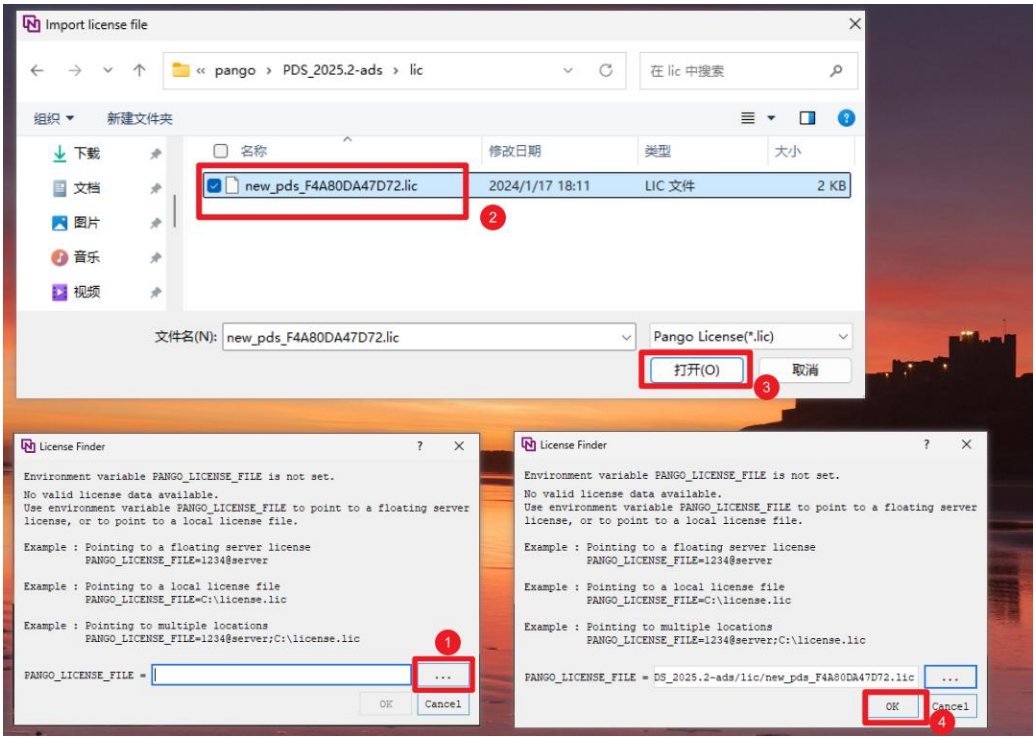
图 1.2-7

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（F4A80DA47D72）。

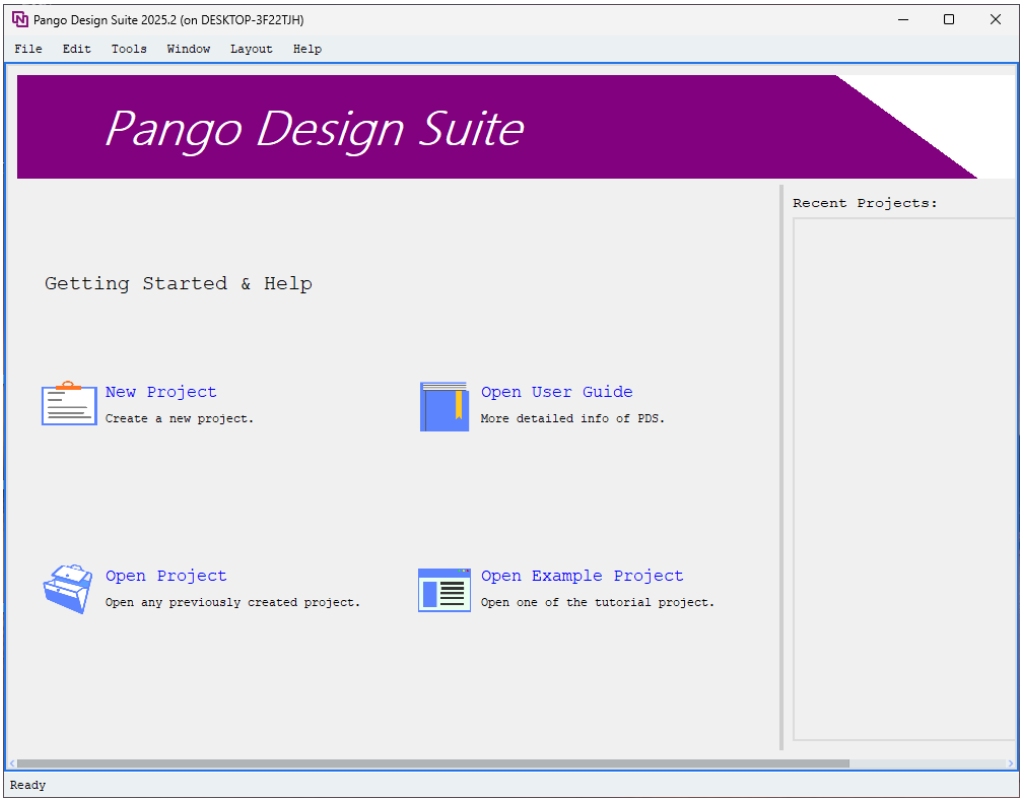
打开 Pango Design Suite 软件，提示需要输入 Lincese 路径。



选择 license 文件，以 C:\pango\PDS_2025.2-ads\lic 为示例作 license 存放文件夹。注意 license 文件以及 PDS 软件的路径均不能有中文以及中文字符



打开 license 文件后点击 ok，出现 PDS 软件界面。



至此软件安装完成。

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

2.1.1.1.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

PDS2025.2

Modelsim10.6c

2.1.1.2.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。

2.1.1.3.接口列表

暂无

2.1.2.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。

2.1.3.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：

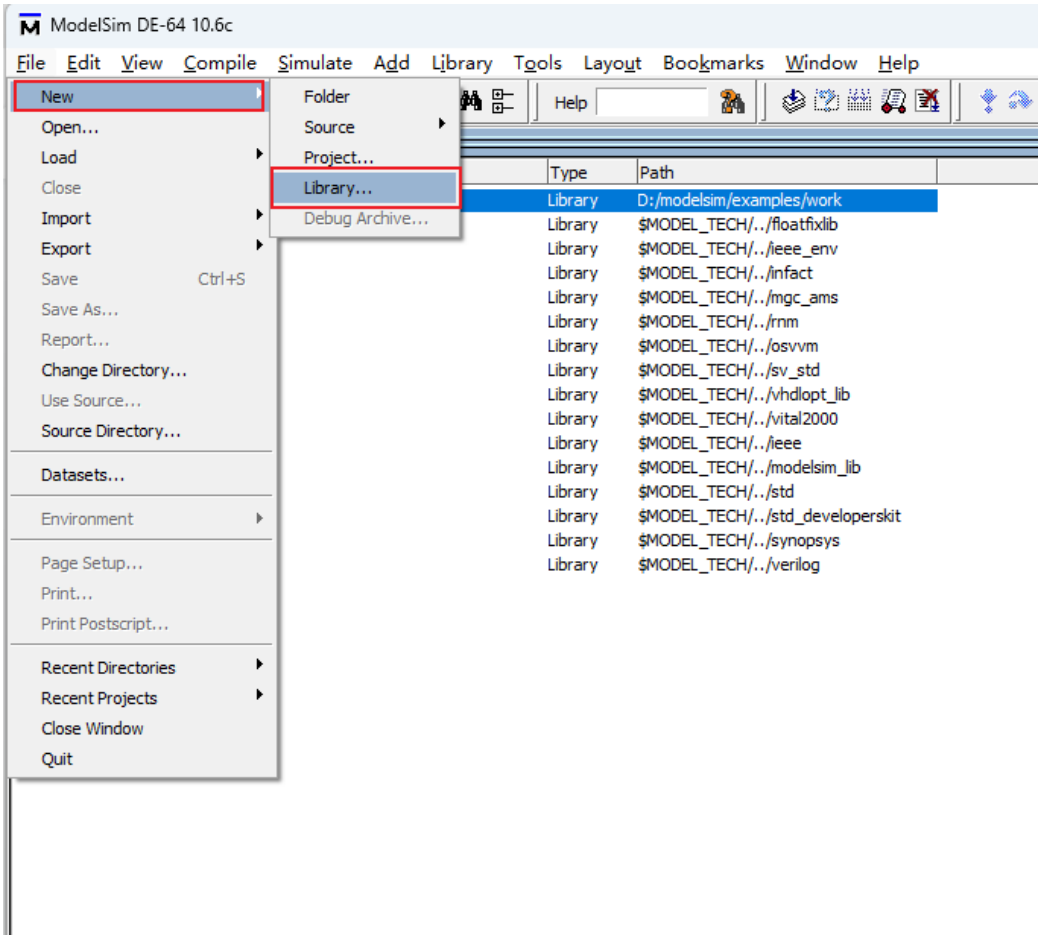


图 1.2-1

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。

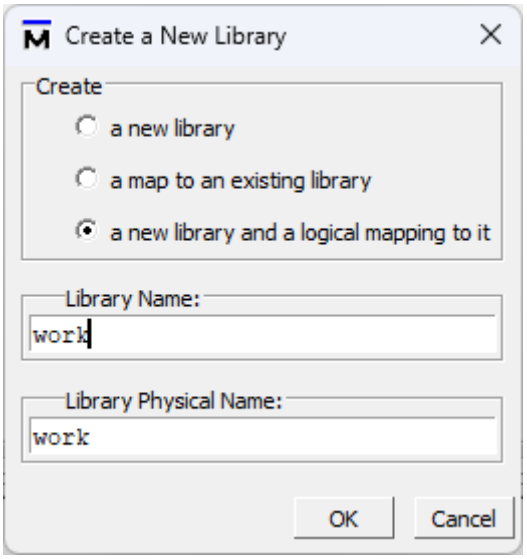


图 1.2-2

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。

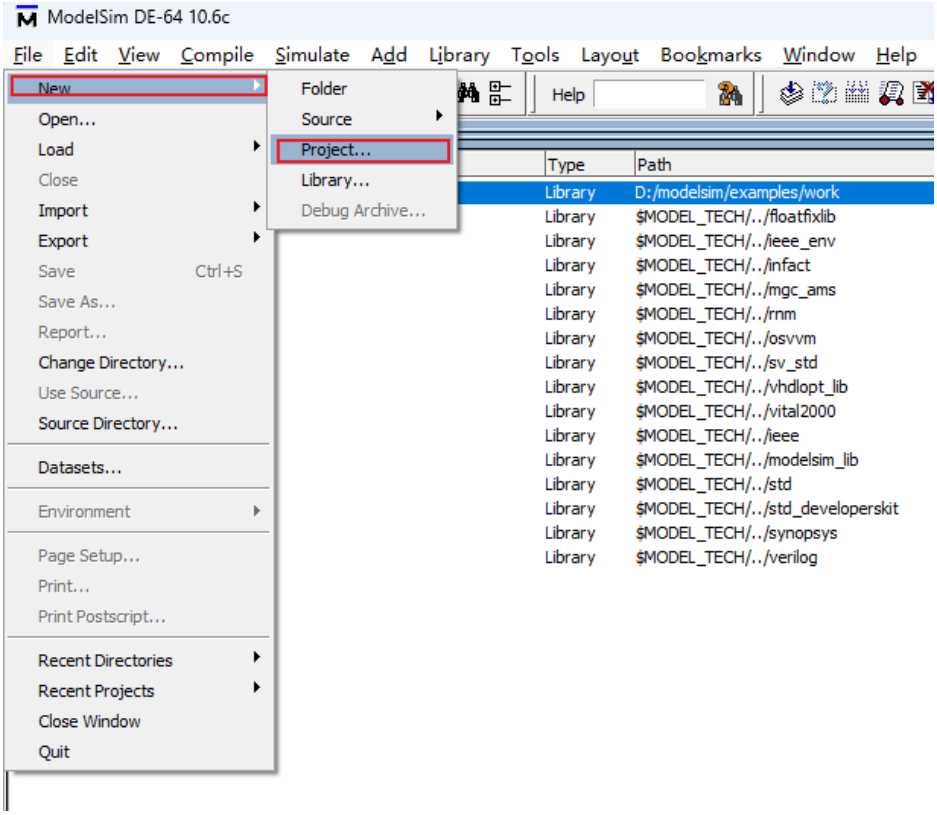


图 1.2-3

左上角 File->New->Project，新建工程。

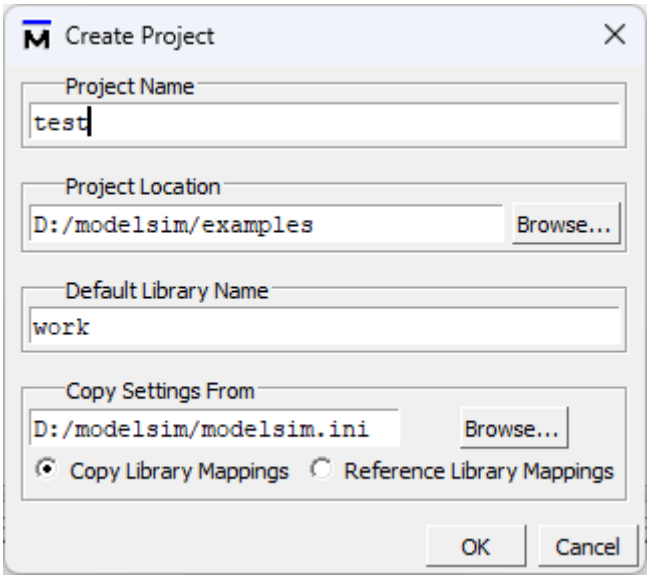


图 1.2-4

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。

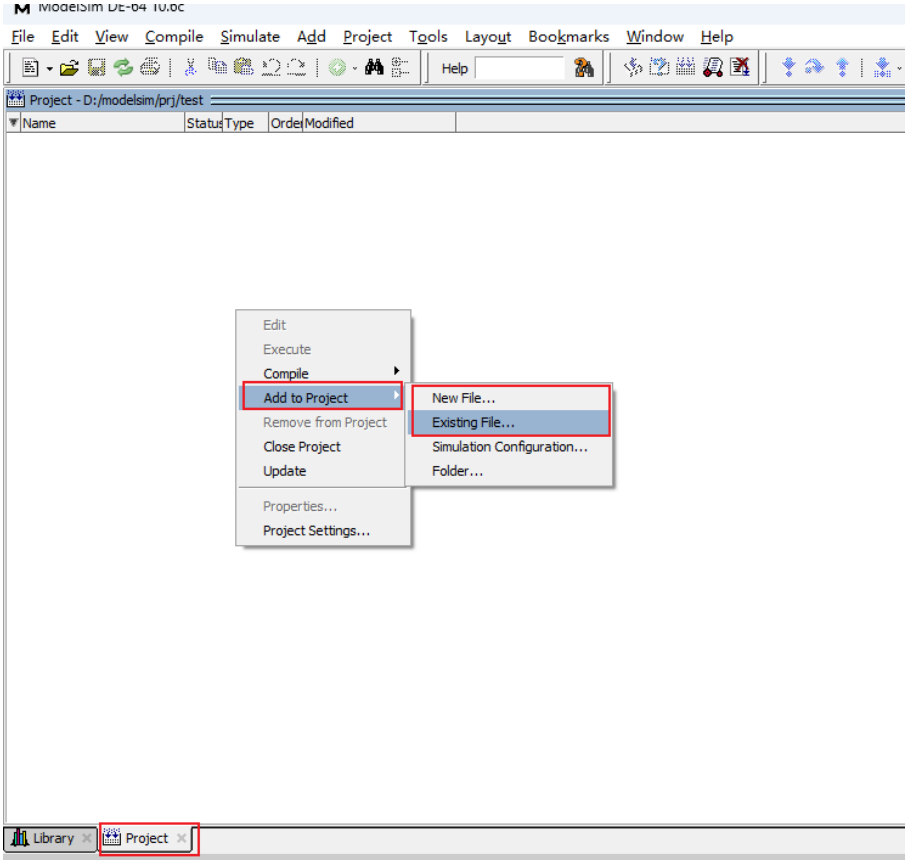


图 1.2-5

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。

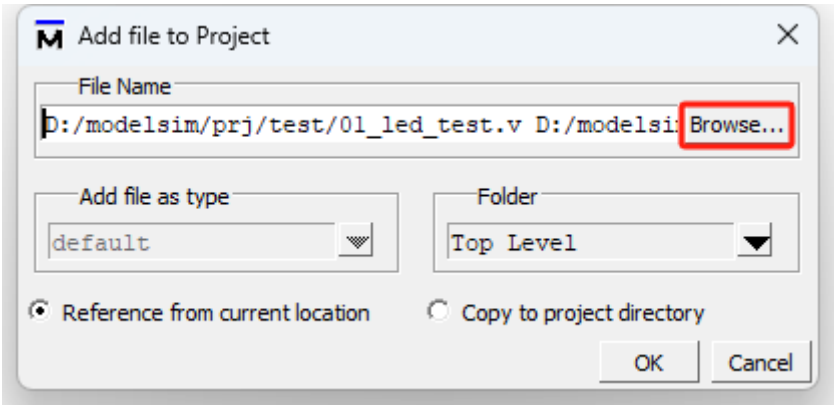


图 1.2-6

点击 Browse，添加要仿真的文件。

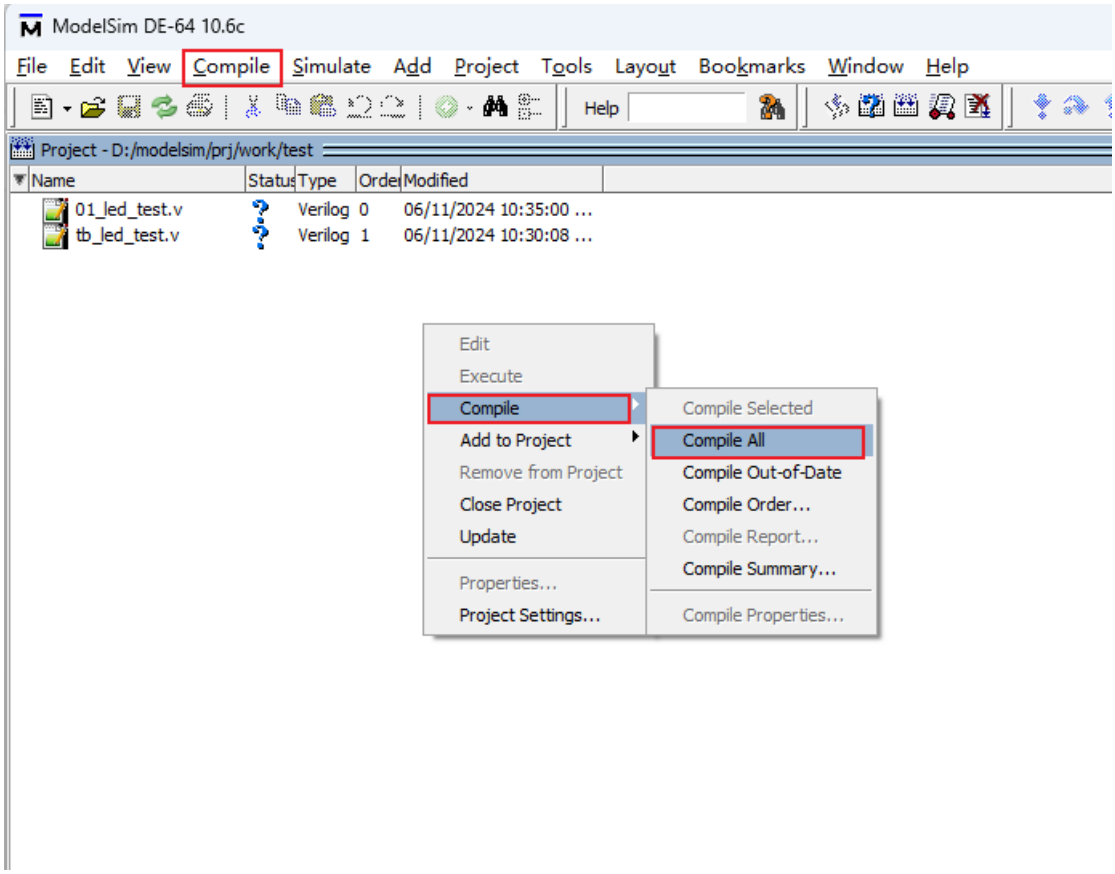
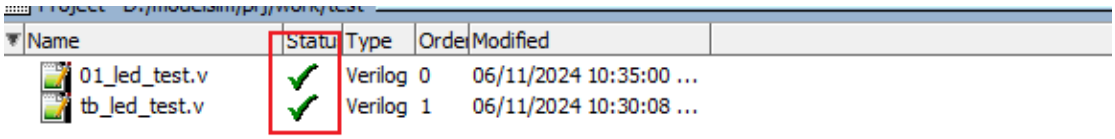


图 1.2-7

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。



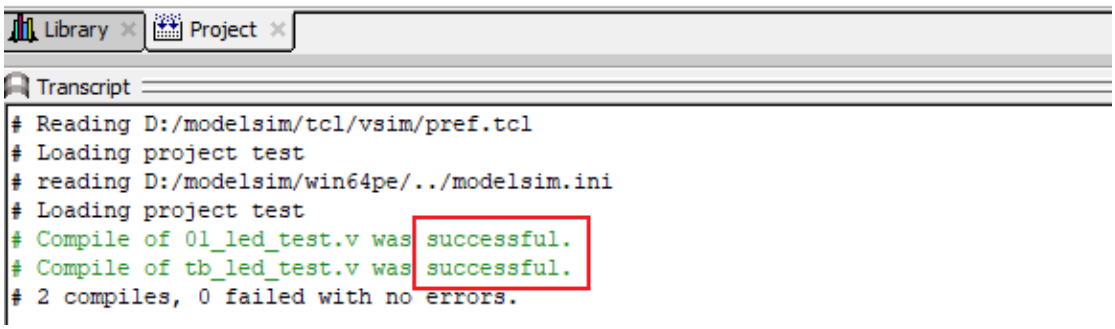


图 1.2-8

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。

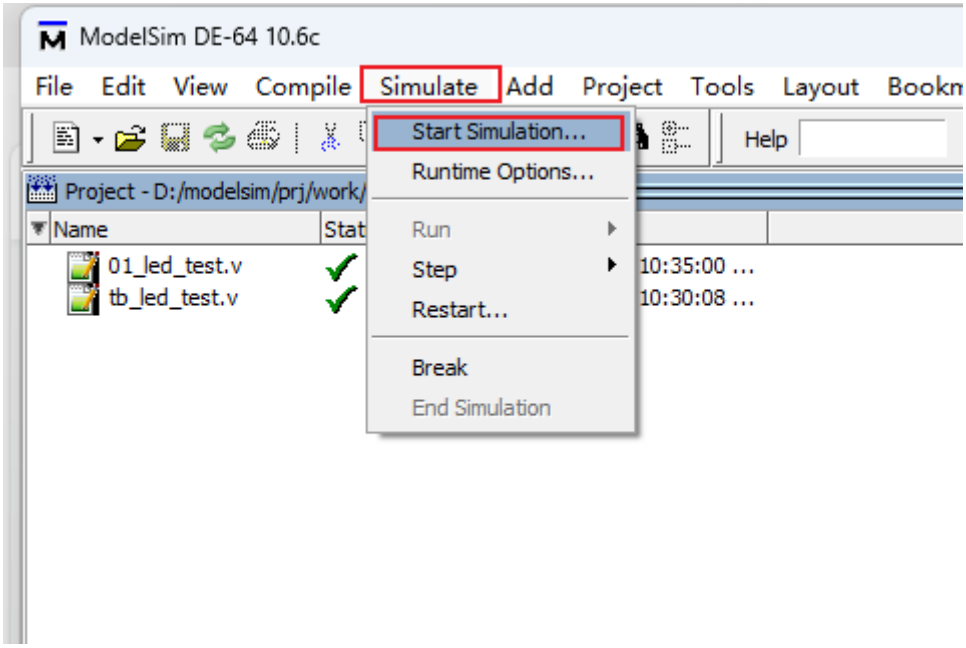


图 1.2-9

选择上方 Simulate->Start Simulation，之后会弹出如图 1.2-10 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。

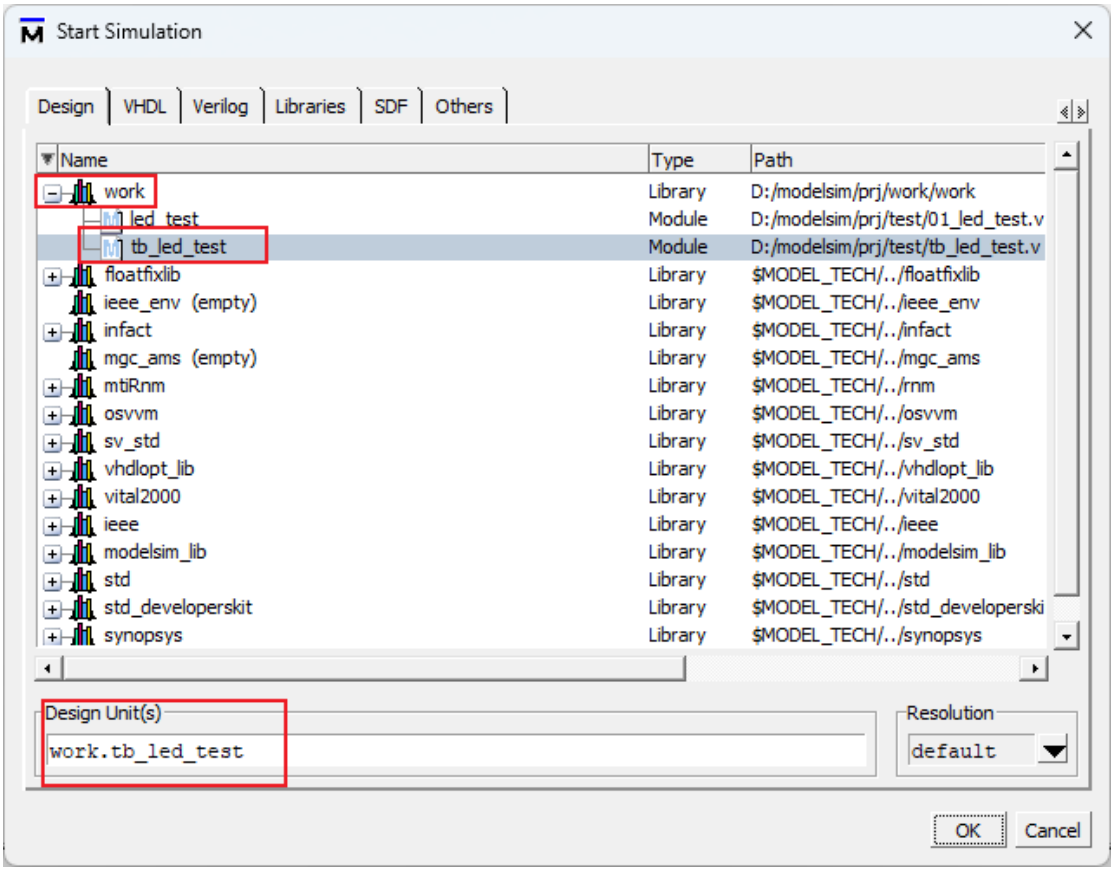


图 1.2-10

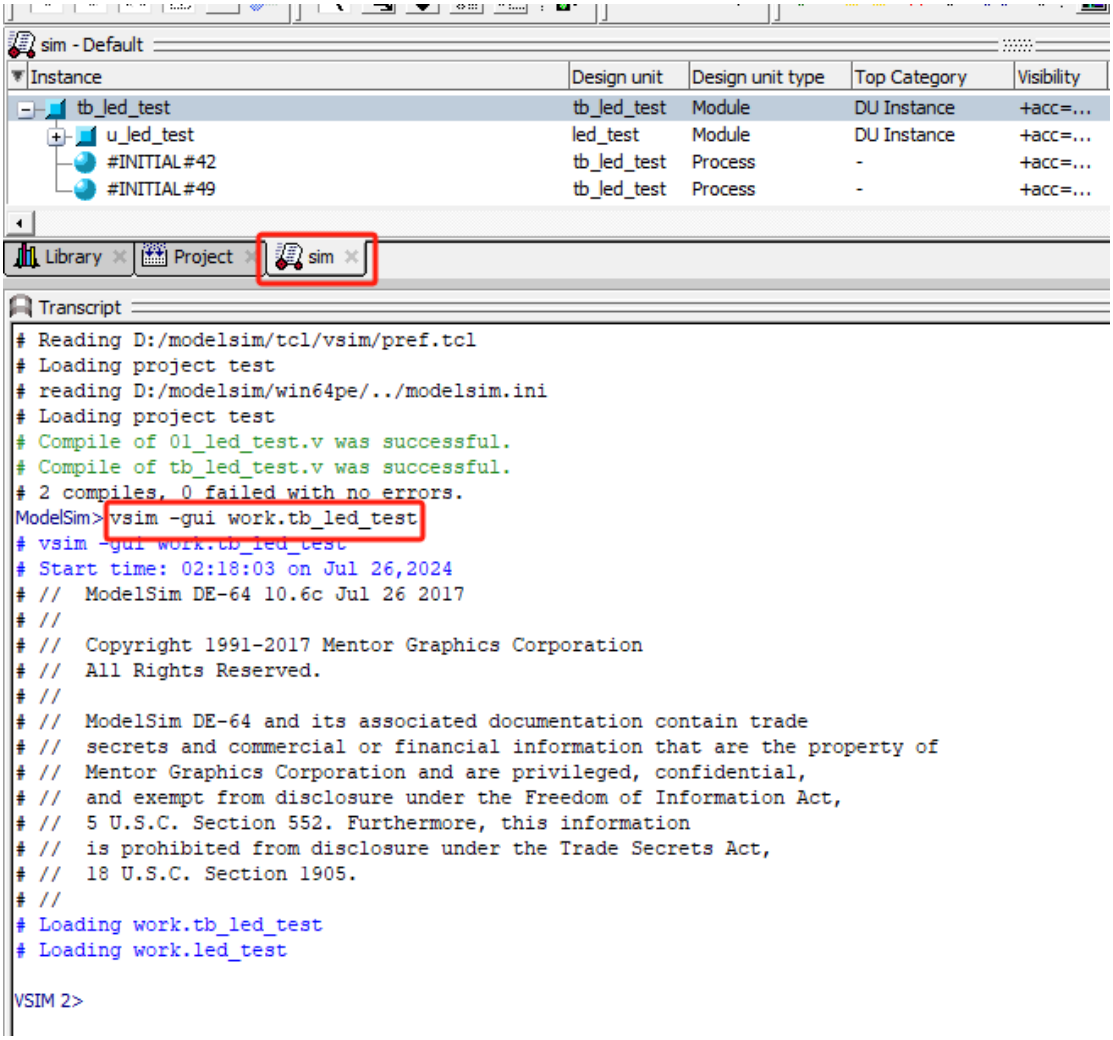


图 1.2-11

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。

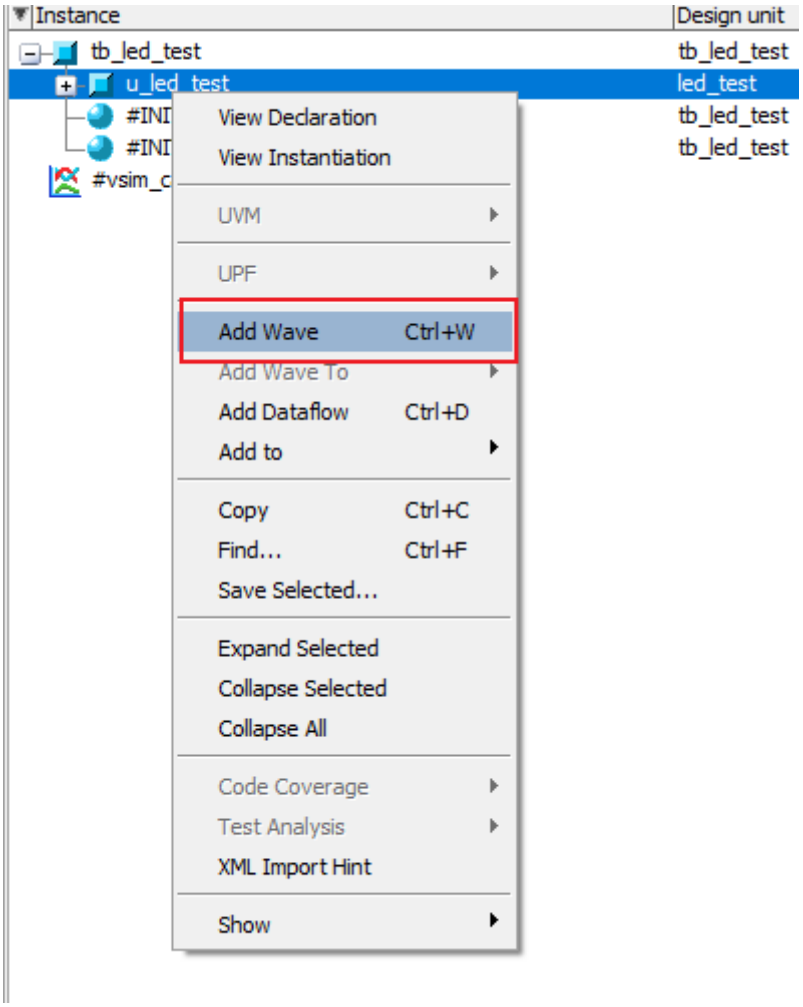


图 1.2-12

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。

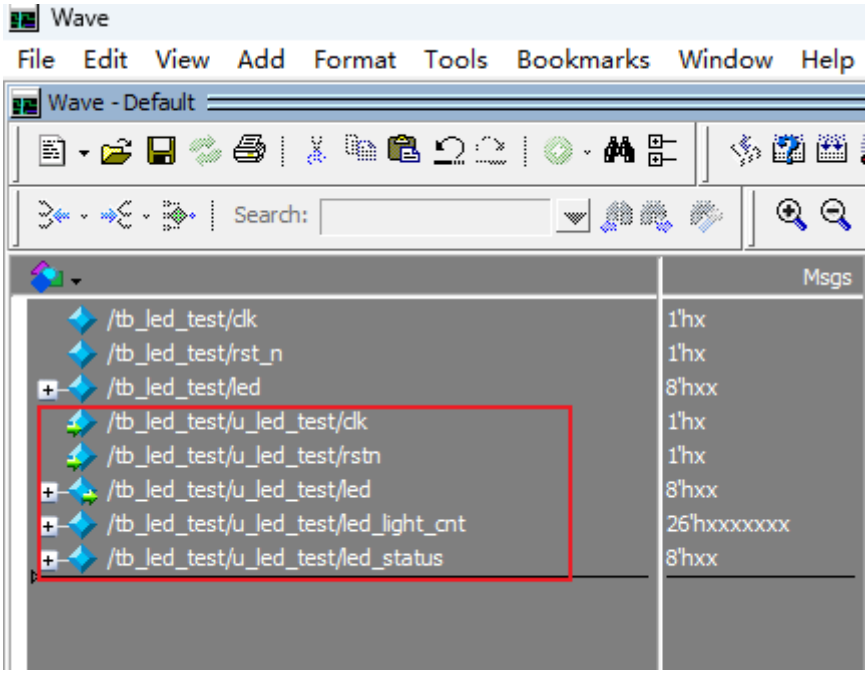


图 1.2-13

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 1.2-14 所示：

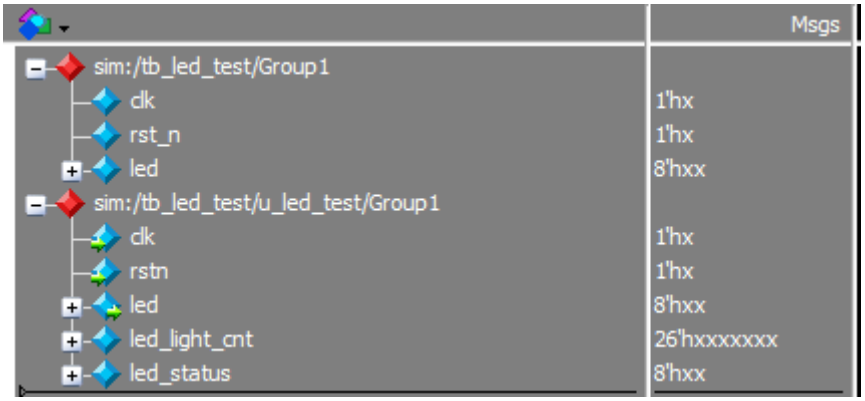


图 1.2-14

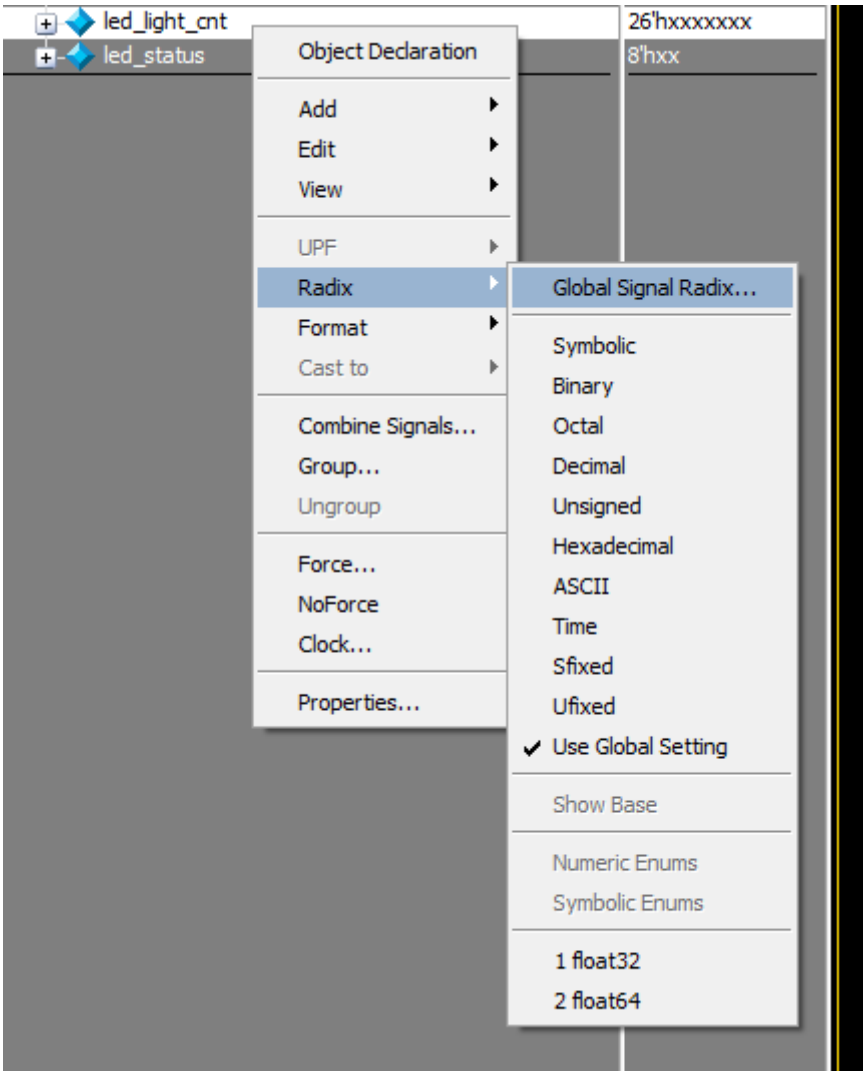


图 1.2-15

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。

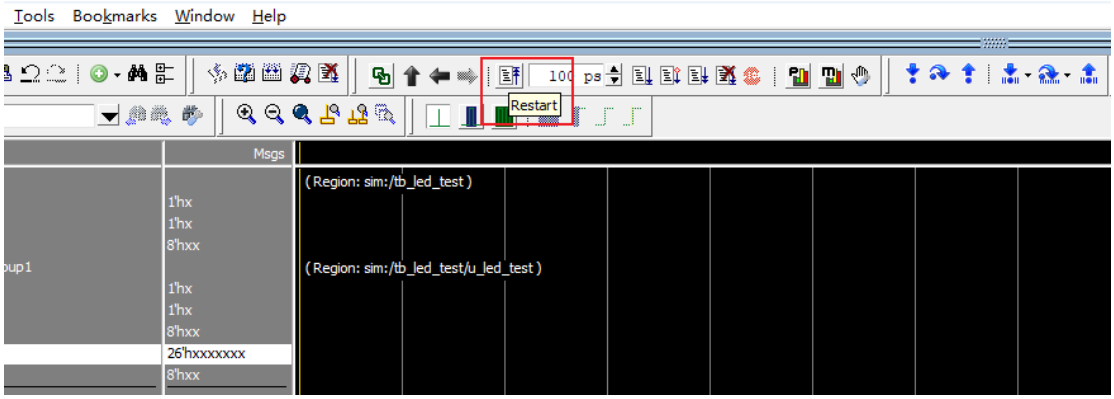


图 1.2-16

点击上方这个地方，对信号全部进行 Restart 复位。

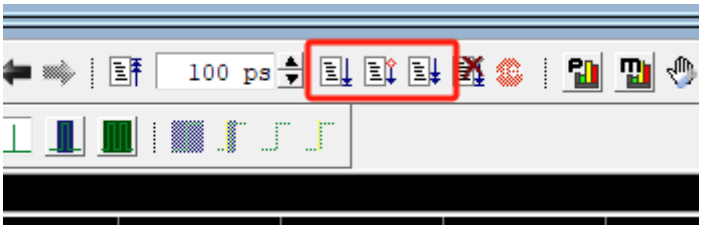


图 1.2-17

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 1.2-18 所示：

图 1.2-18

按下后，当用户点击 stop，仿真才会停止。

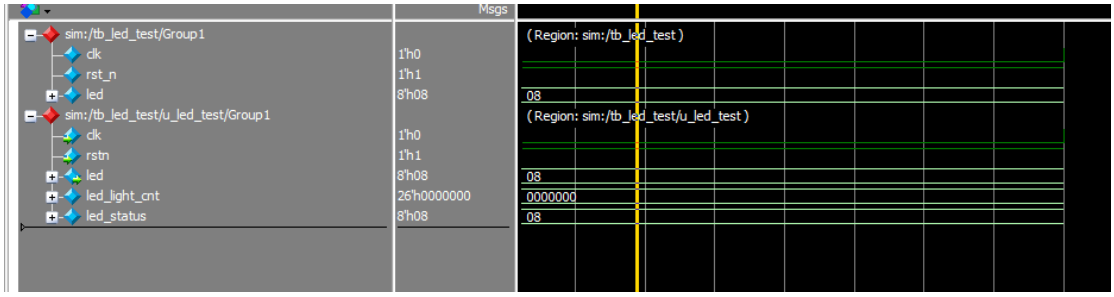


图 1.2-19

图 1.2-19 为操作后显示出来的波形。

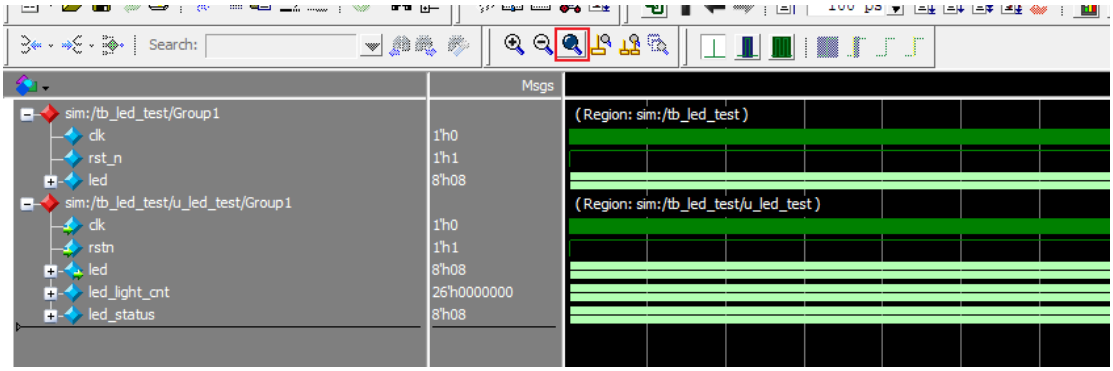


图 1.2-20

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

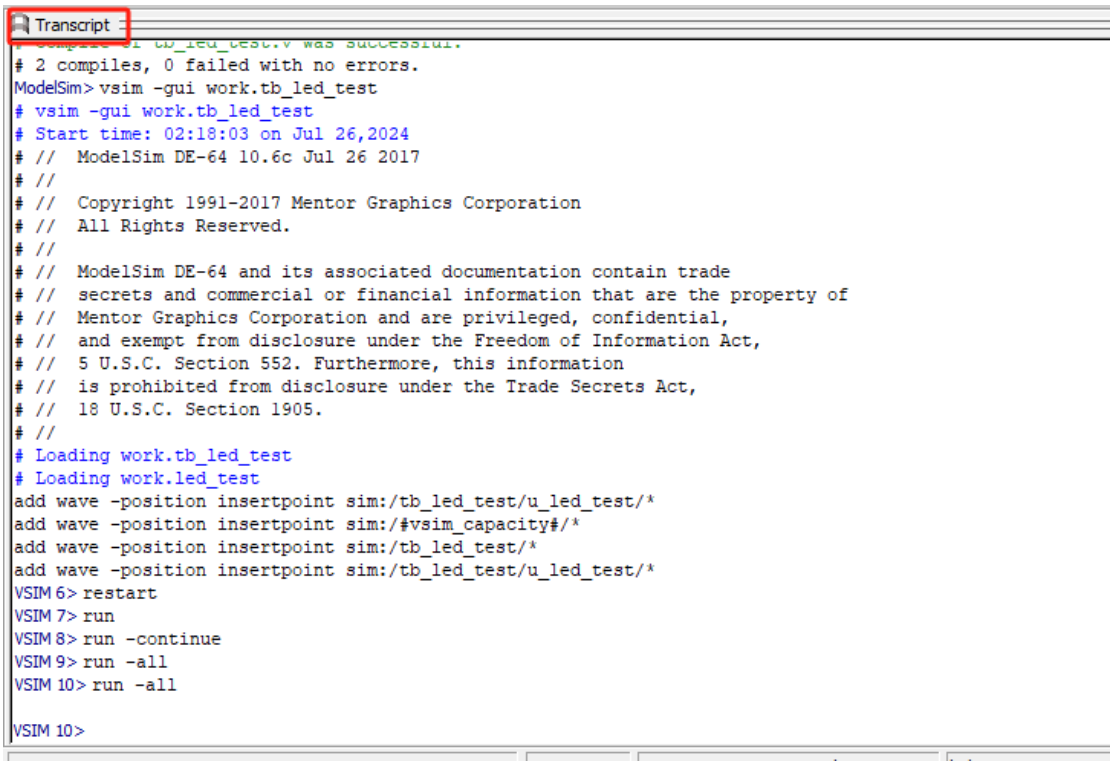


图 1.2-21

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 1.2-22 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。

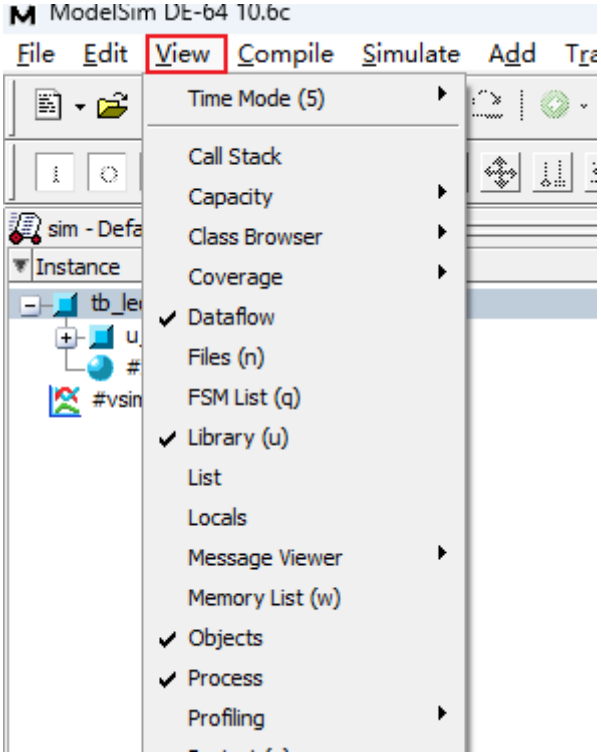


图 1.2-22

2.1.3.1.文件的编写

2.1.3.2.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）

2.1.3.3.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 2-23 所示：

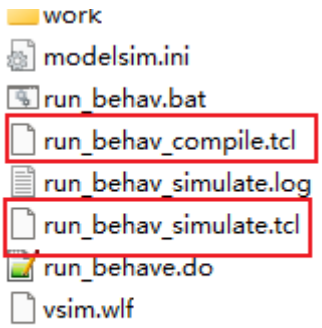


图 2-23

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

PDS2025.2

硬件环境：

PG2K100-6IMBG400

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：

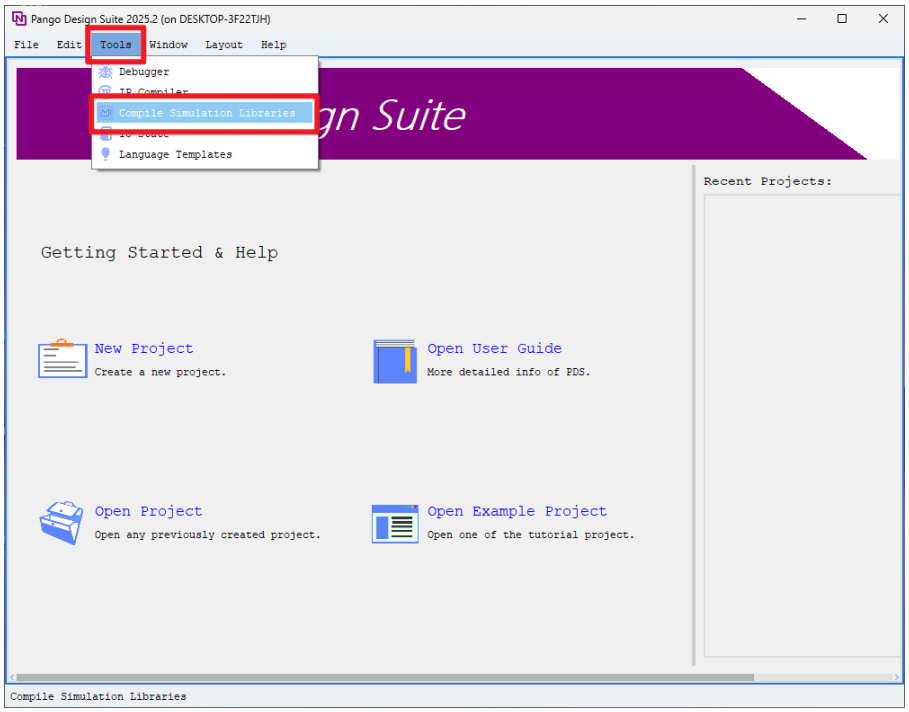


图 3.2-1

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;

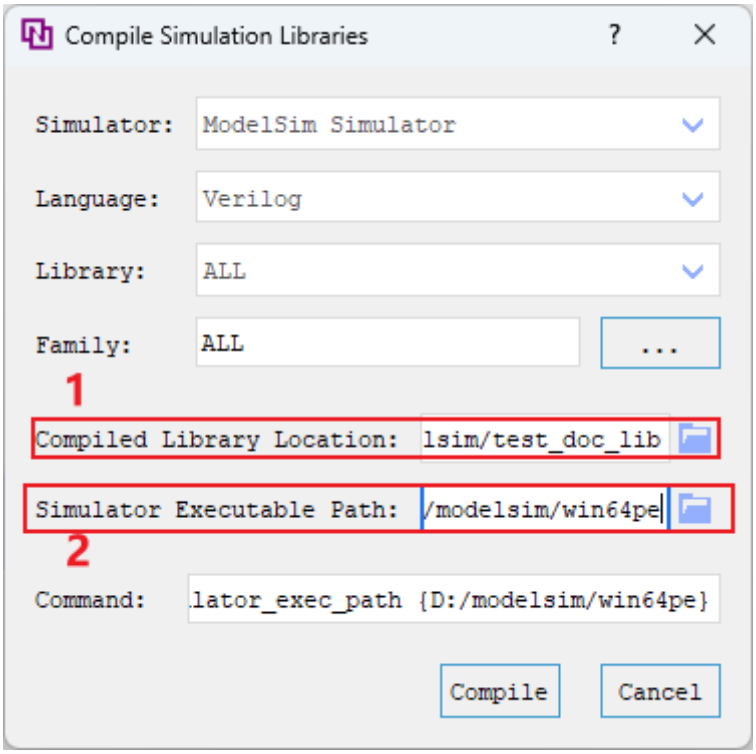


图 3.2-2

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。

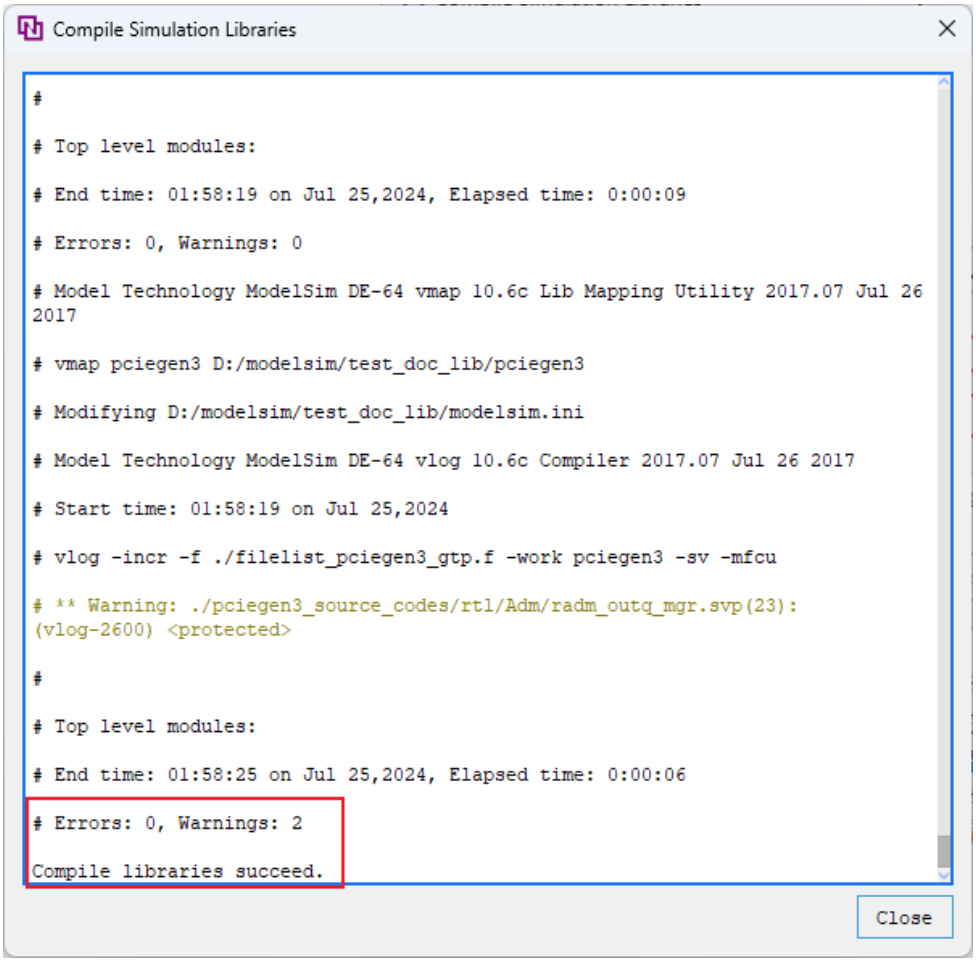


图 3.2-3

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。

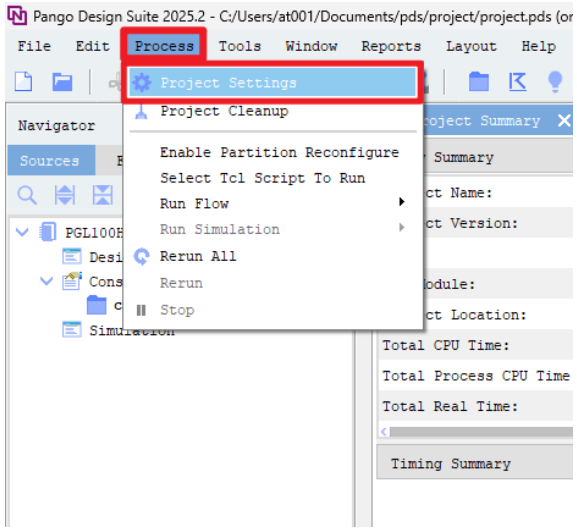


图 3.2-4

打开工程后，选择 Project->Project Settings；

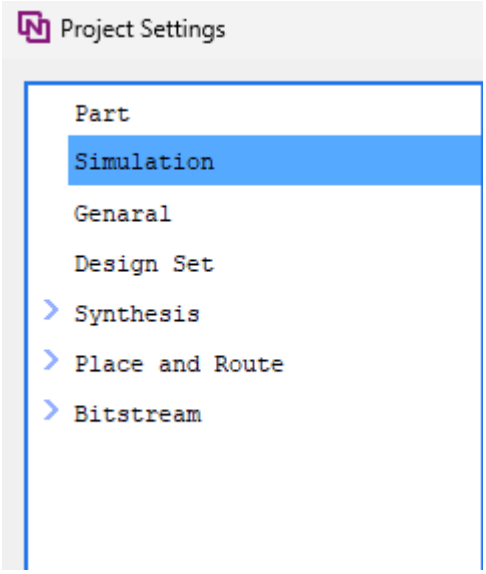


图 3.2-5

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

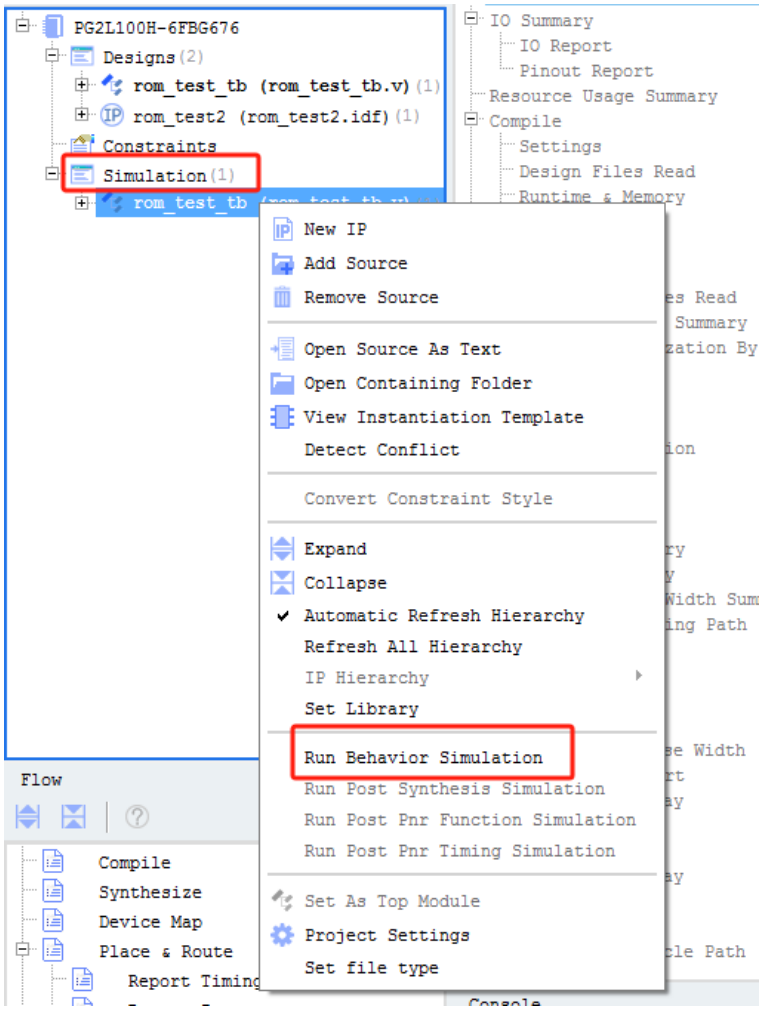


图 3.2-7

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：

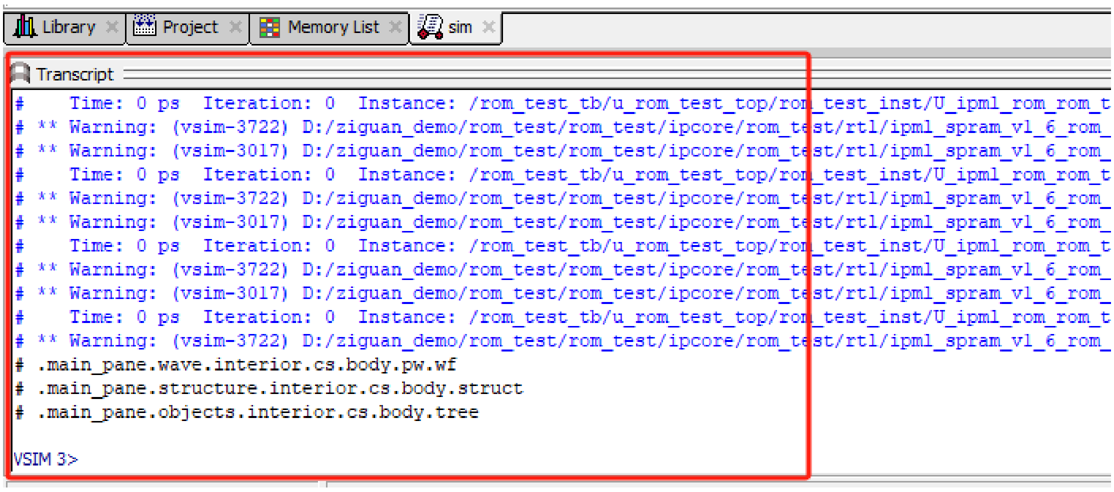


图 3.2-8

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

PDS2025.2

硬件环境：

PG2K100-6IMBG400

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。

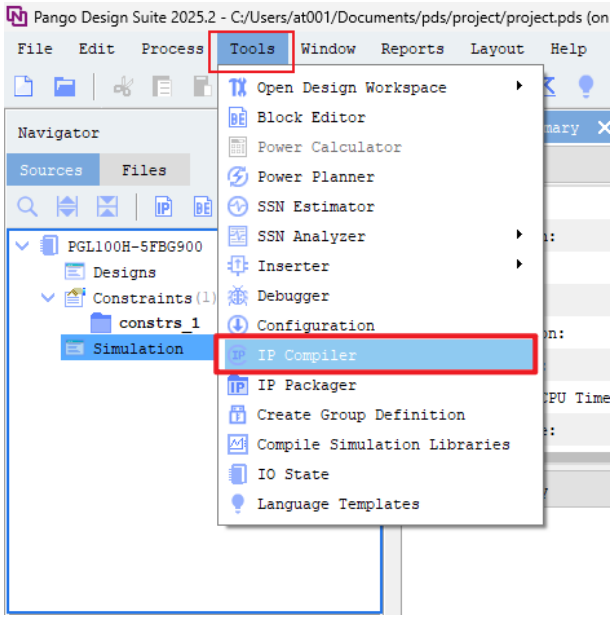


图 4.2-1

打开 PDS 后，点击图 4.2-1 里红框部分的 IP 图标。

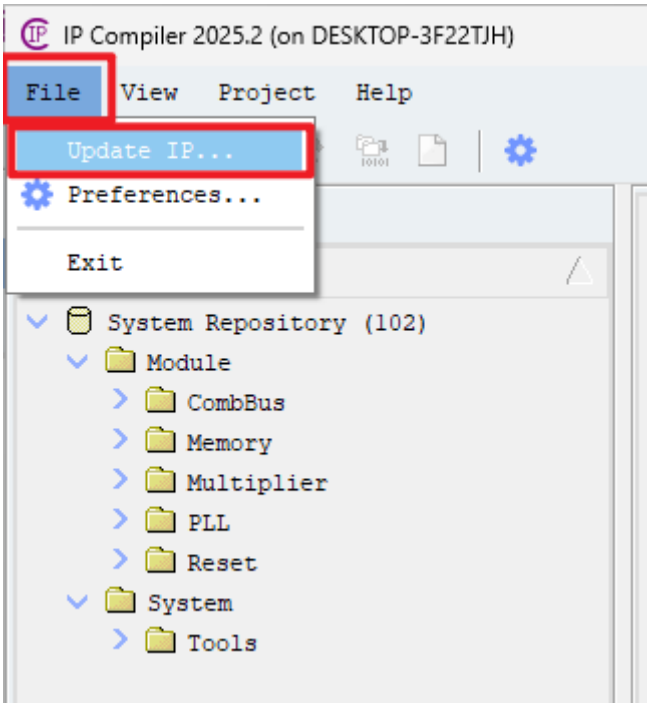


图 4.2-2

之后在弹出的选项卡的左上角点击 File->Update...

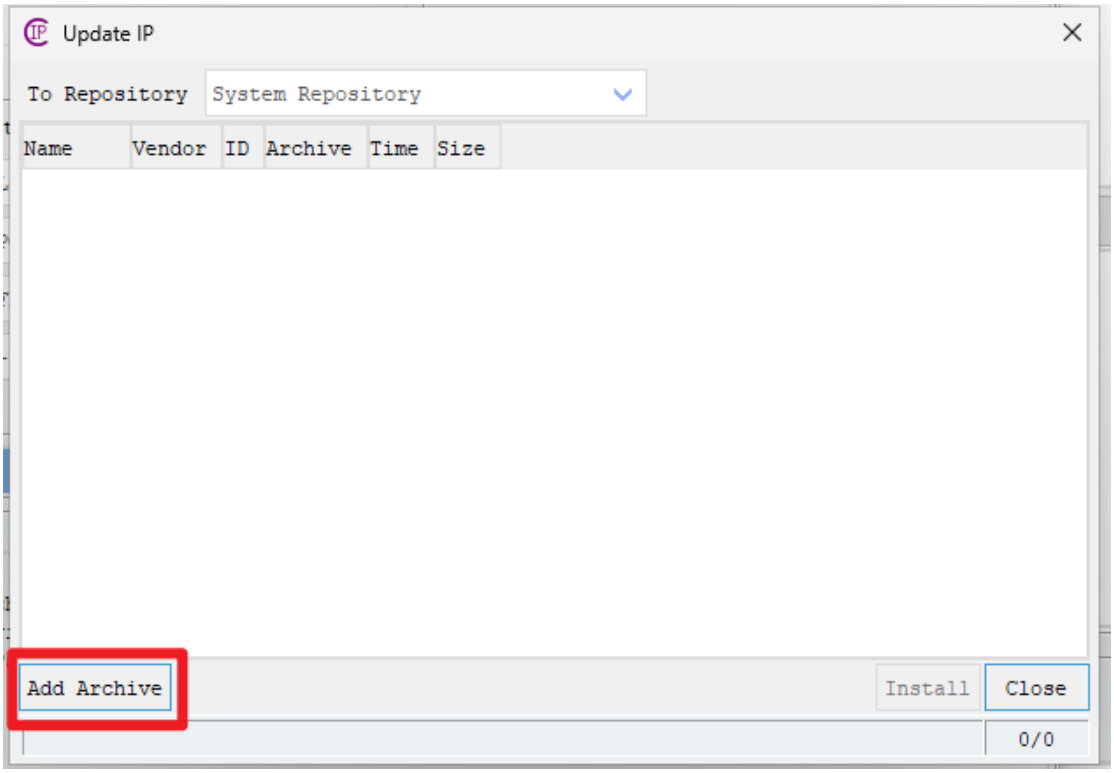


图 4.2-3

点击左上角 Add Package。

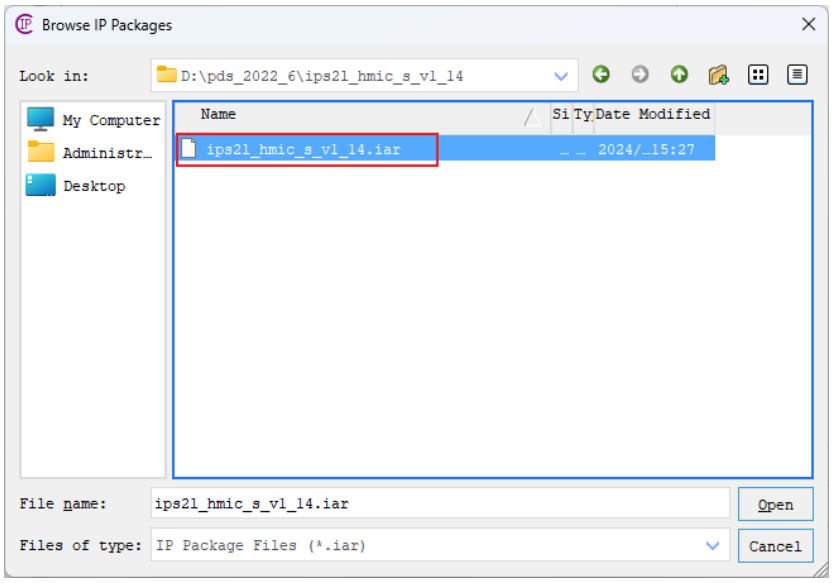


图 4.2-4

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。

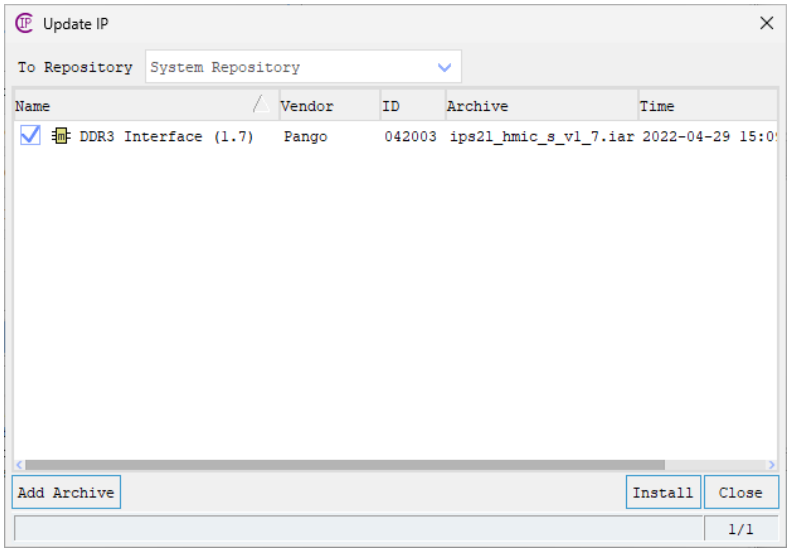


图 4.2-5

之后勾选上前面的√，点击 Install 即可。

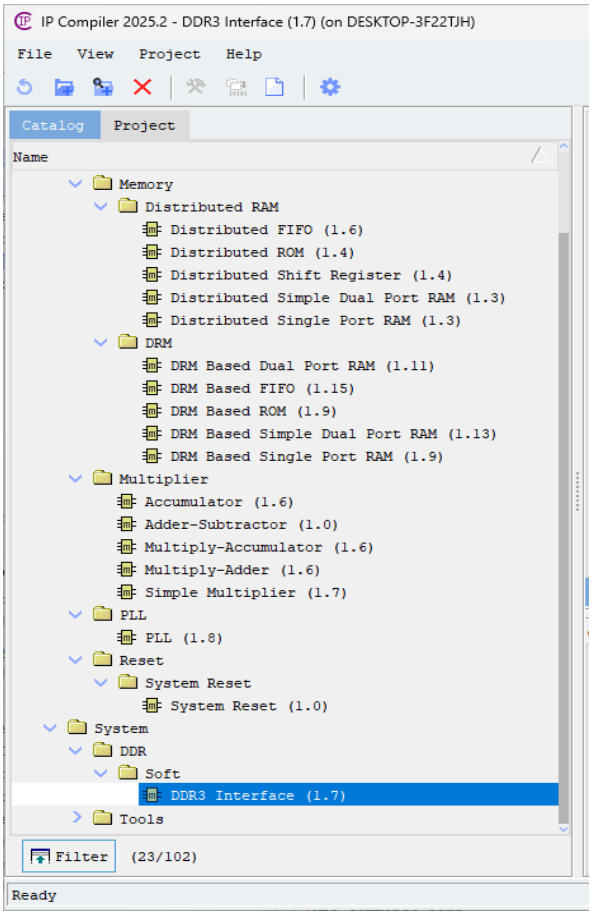


图 4.2-6

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

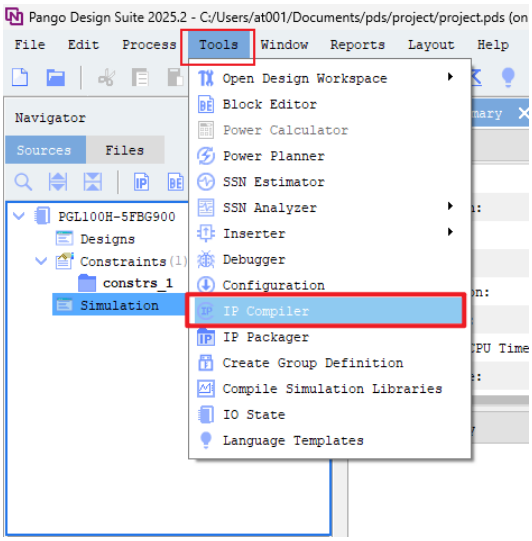


图 4.2-7

继续点击图 4.2-7 所示红框的图标。

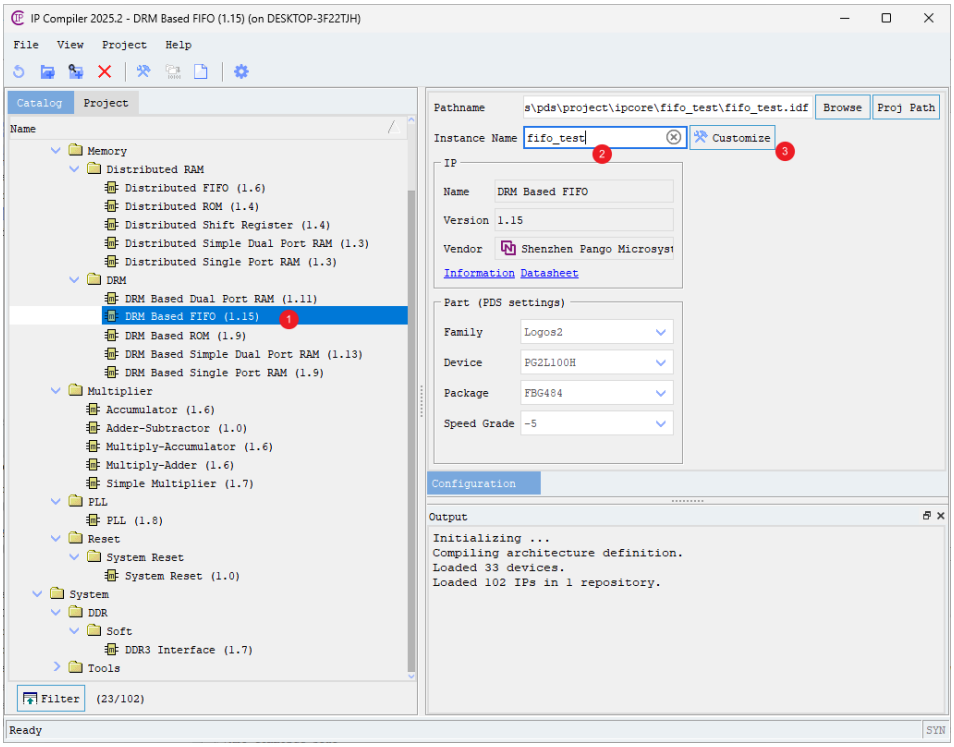


图 4.2-8

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：

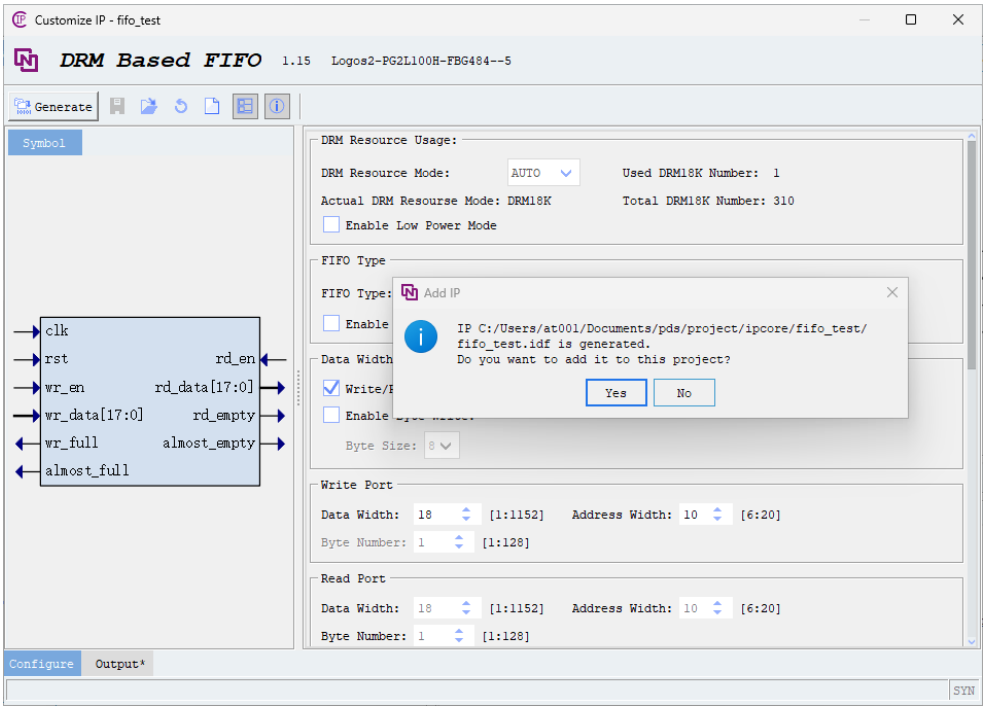


图 4.2-9

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：

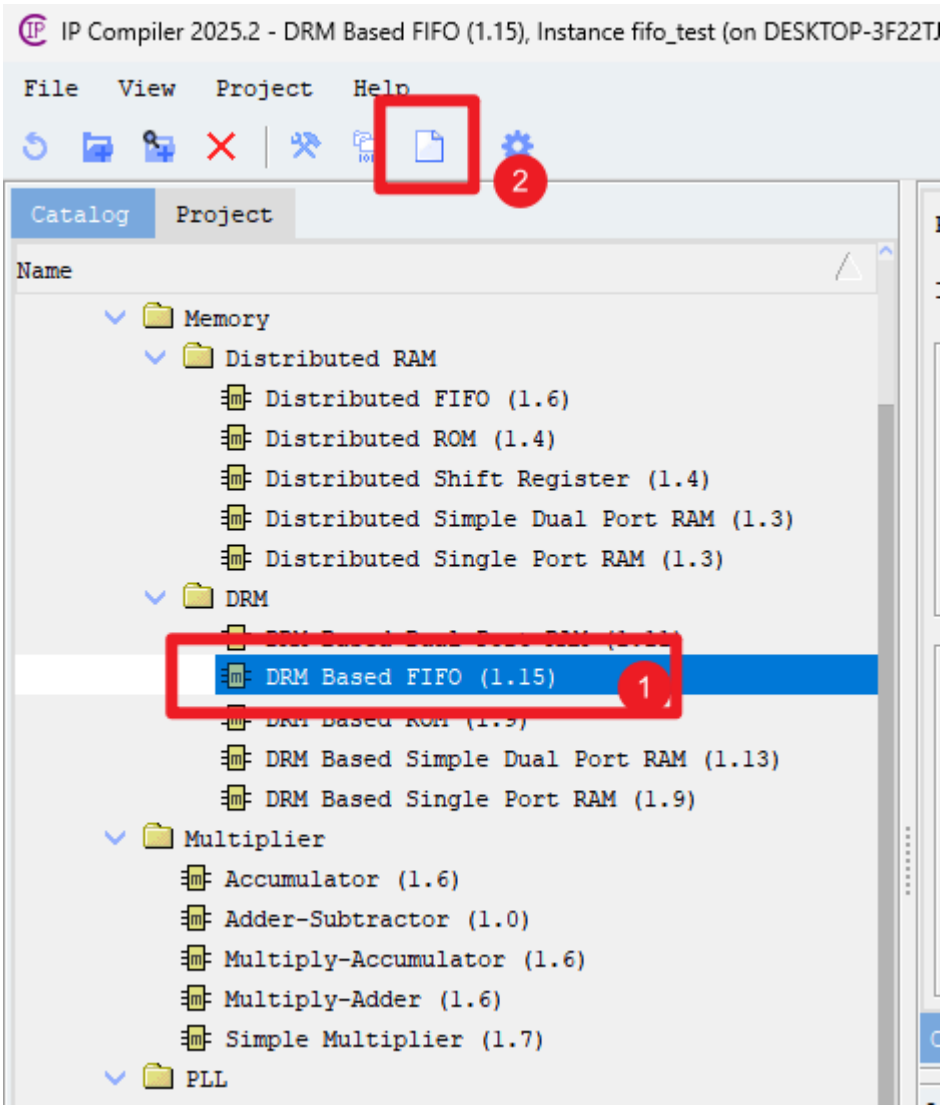


图 4.2-10

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。

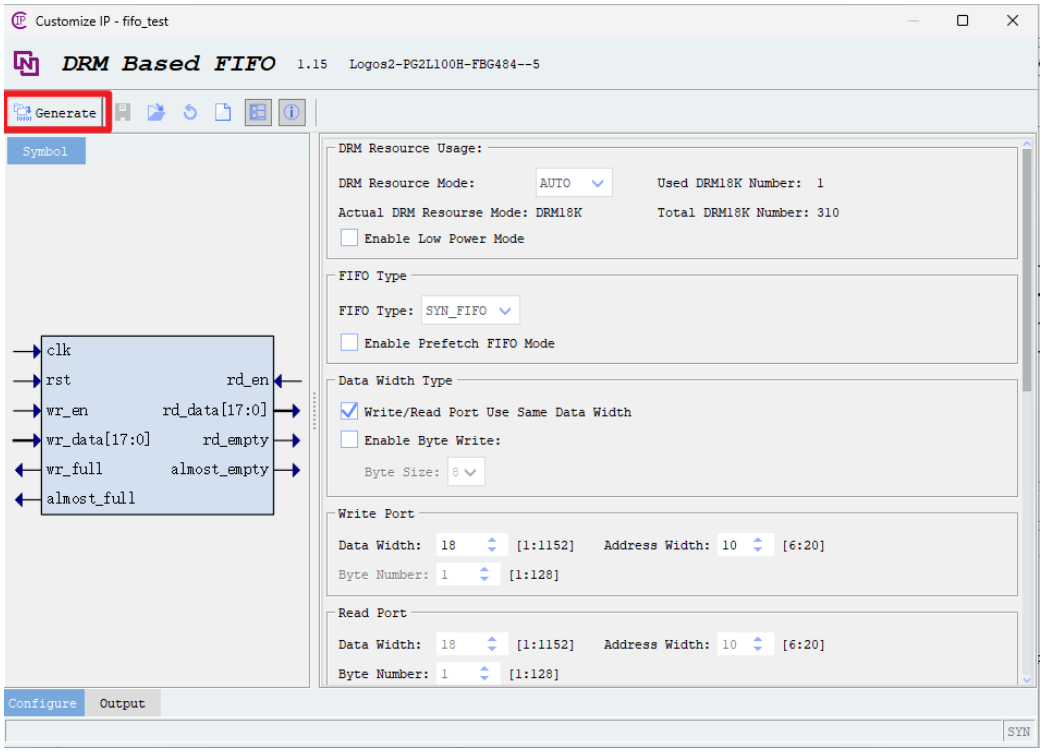


图 4.2-12

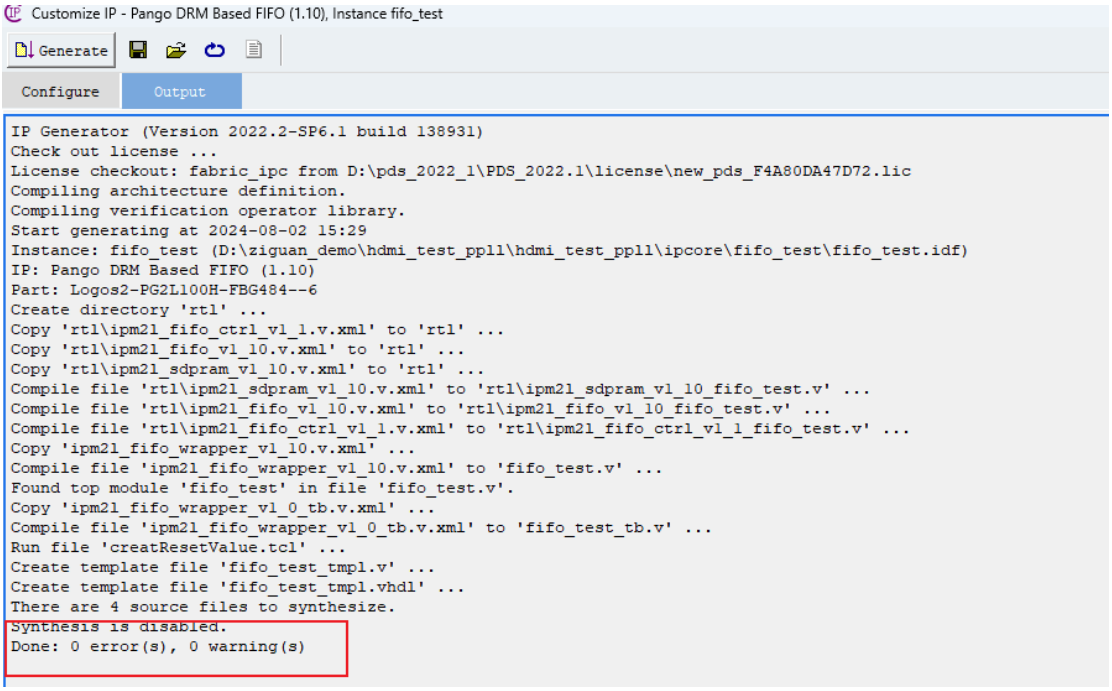


图 4.2-13

没有任何错误测表示生成成功。

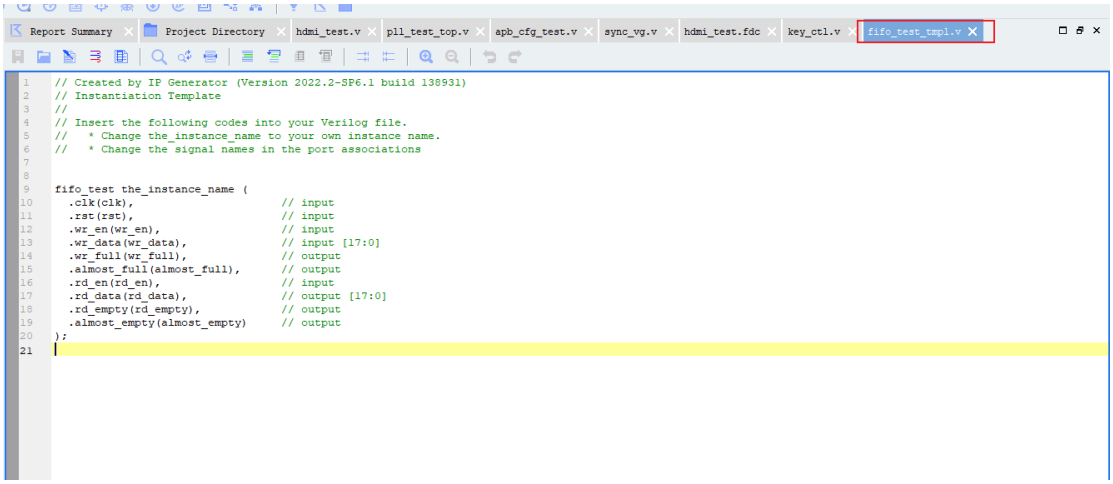


图 4.2-14

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

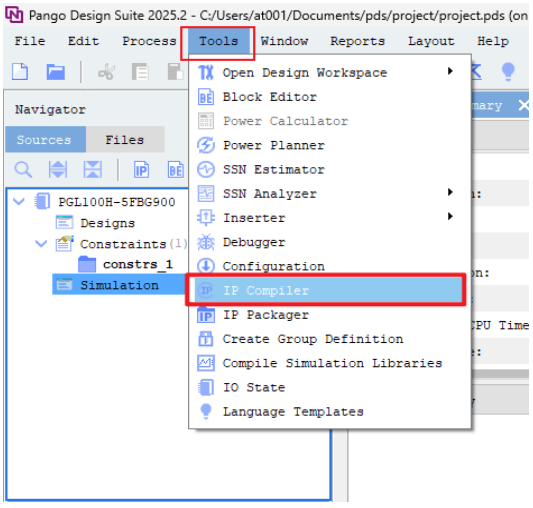


图 5.2-1 “IP”图标示意图

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：

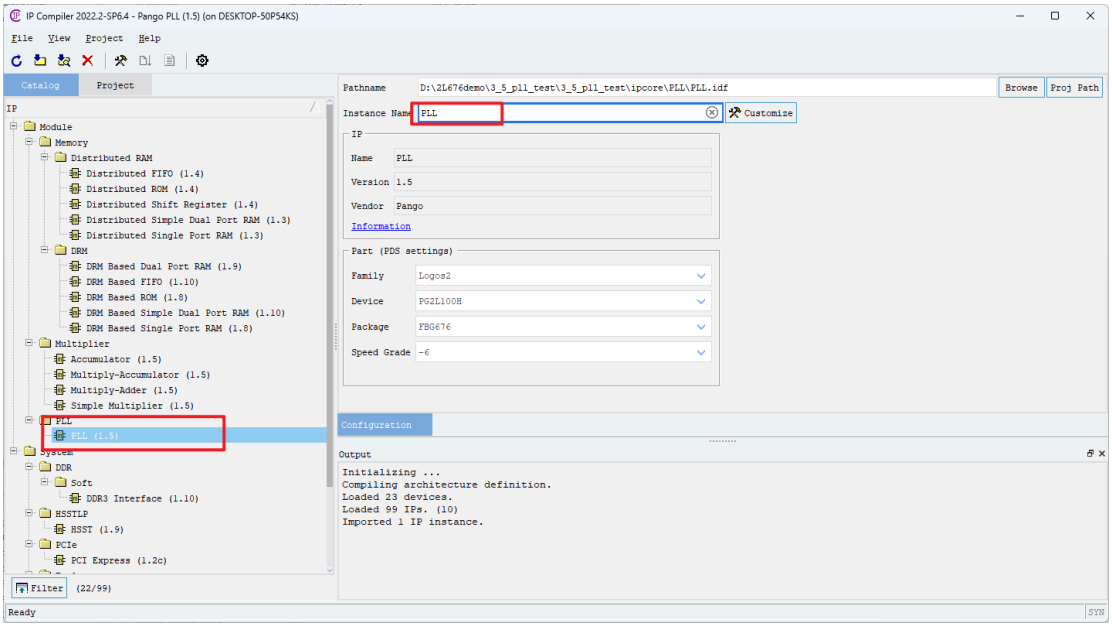


图 5.2-2 IP 配置流程图 1

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。

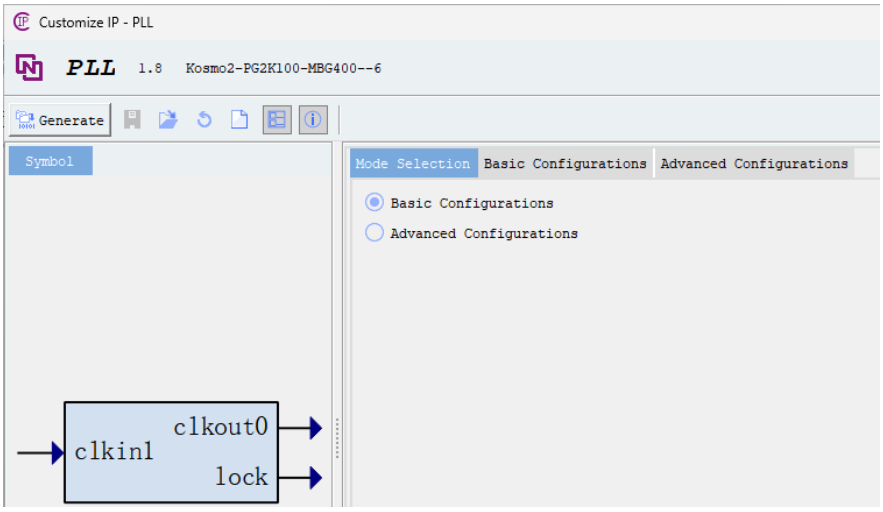


图 5.2-3 IP 配置流程图 2

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：

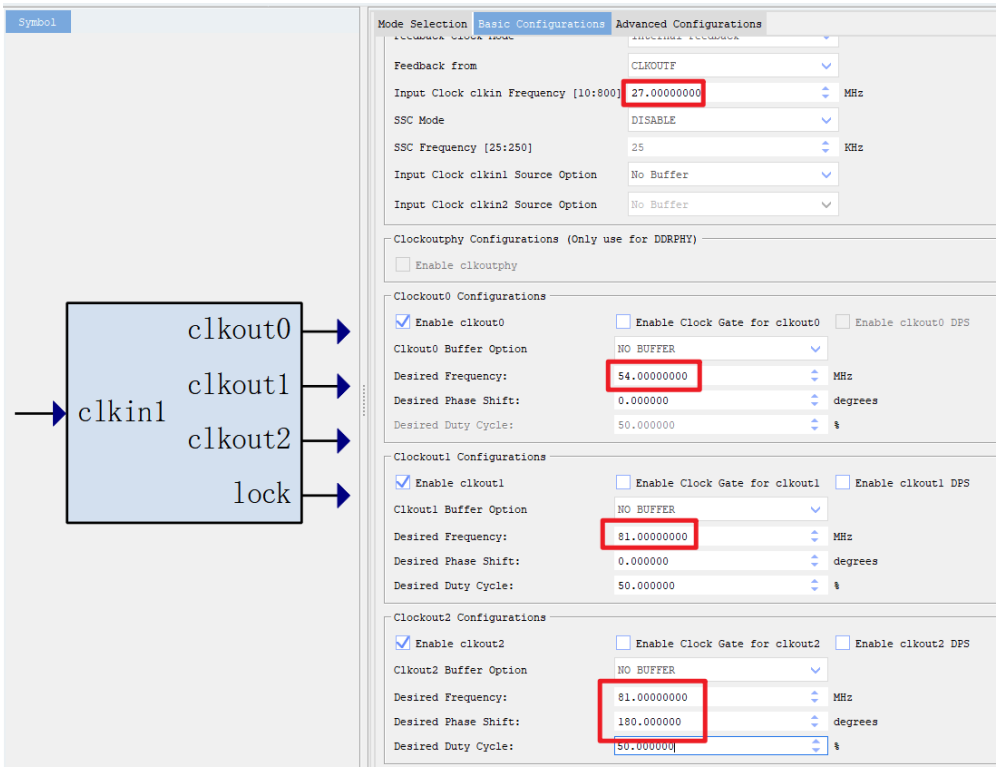


图 5.2-4 IP 配置流程图 3

点击左上角 generate 生成 IP。

5.2.3.代码设计

模块接口列表如下所示：

表 5.2-1 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：

```
module PLL_TEST(
    input           sys_clk     ,
    output          clkout0     ,
    output          clkout1     ,
    output          clkout2     ,
    output          lock
  );

PLL PLL_U0 (
    .clkout0        (clkout0        ),// output
    .clkout1        (clkout1        ),// output
    .clkout2        (clkout2        ),// output
    .lock           (lock           ),// output
    .clkin1         (sys_clk        ) // input
);


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：

```
timescale 1ns / 1ps

module    PLL_tb();
    reg             sys_clk     ;
    wire            clkout0     ;
```

```
24.         wire                                    clkout1                 ;
25.         wire                                    clkout2                 ;
26.         wire                                    lock                    ;
27.
28.
29.
30.         initial
31.           begin
32.             #2
33.                 sys_clk <= 0        ;
34.           end
35.
36.         parameter     CLK_FREQ = 27;//Mhz
37.         always # ( 1000/CLK_FREQ/2 ) sys_clk = ~sys_clk ;
38.
39.
40.       PLL_TEST u_PLL_TEST(
41.         .sys_clk                                (sys_clk                ),
42.         .clkout0                                (clkout0                ),
43.         .clkout1                                (clkout1                ),
44.         .clkout2                                (clkout2                ),
45.         .lock                                   (lock                   )
46.       );
47.
48.
49.       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。

5.2.4.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。

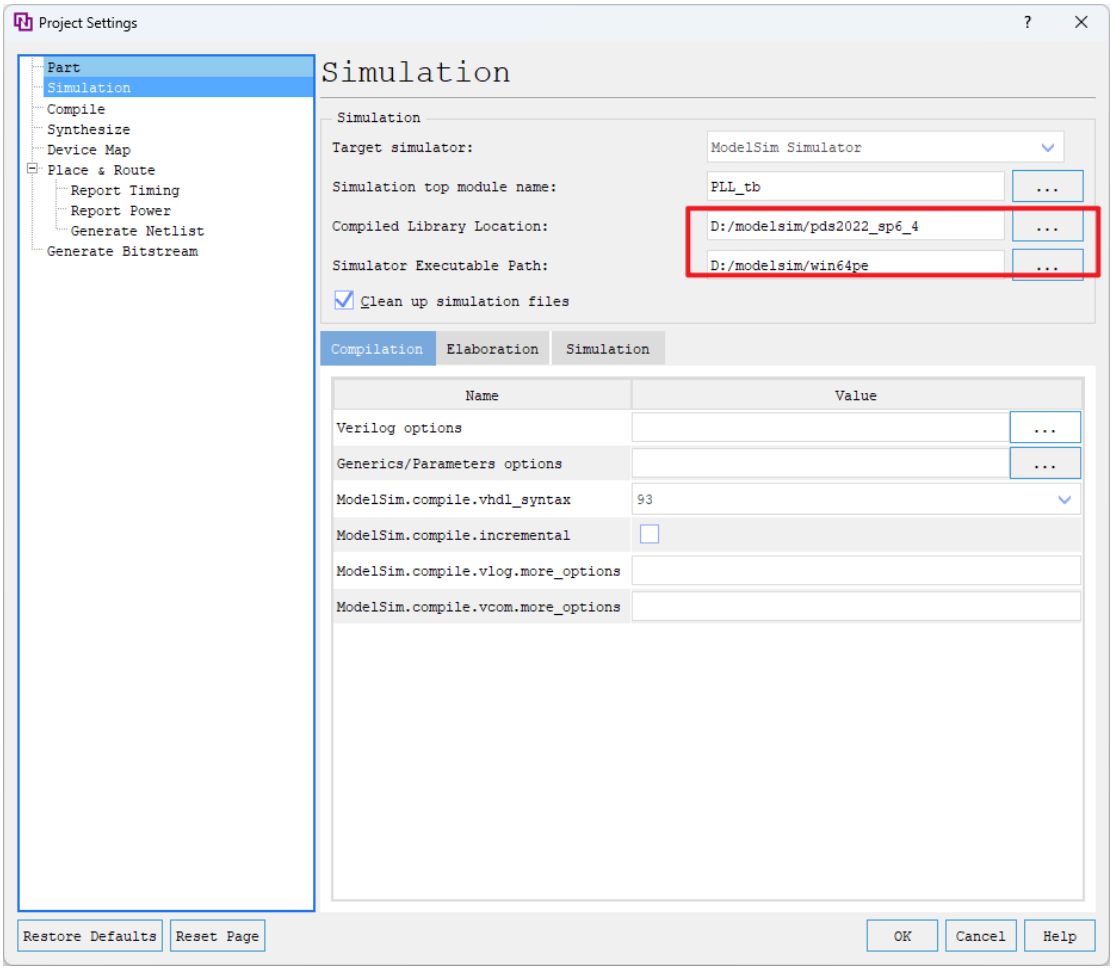


图 5.2-5 PDS 和 Modelsim 联合仿真流程图 4

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。

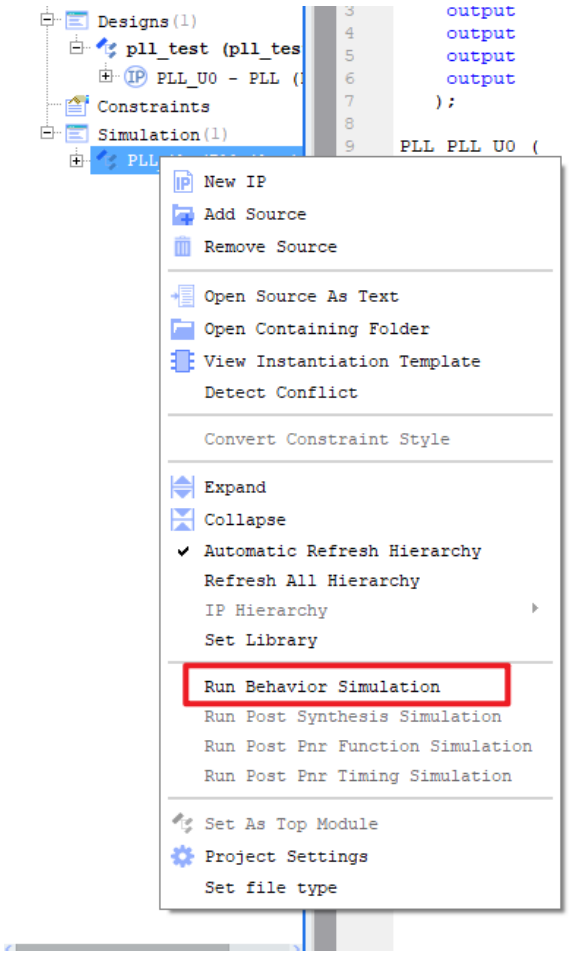


图 5.2-6 PDS 和 Modelsim 联合仿真流程图 5

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。

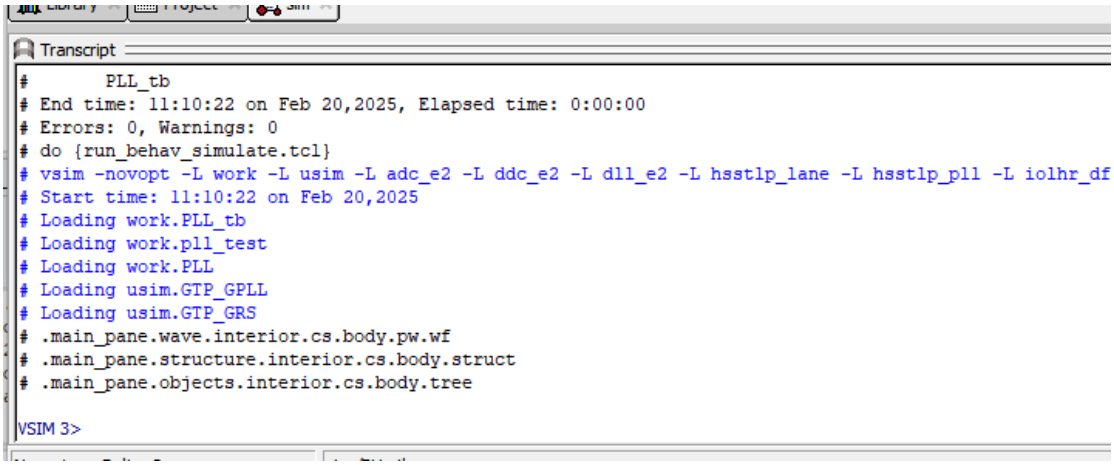


图 5.2-7 PDS 和 Modelsim 联合仿真流程图 6

5.2.5.实验现象

点击 Wave 观察 PLL 输出信号：

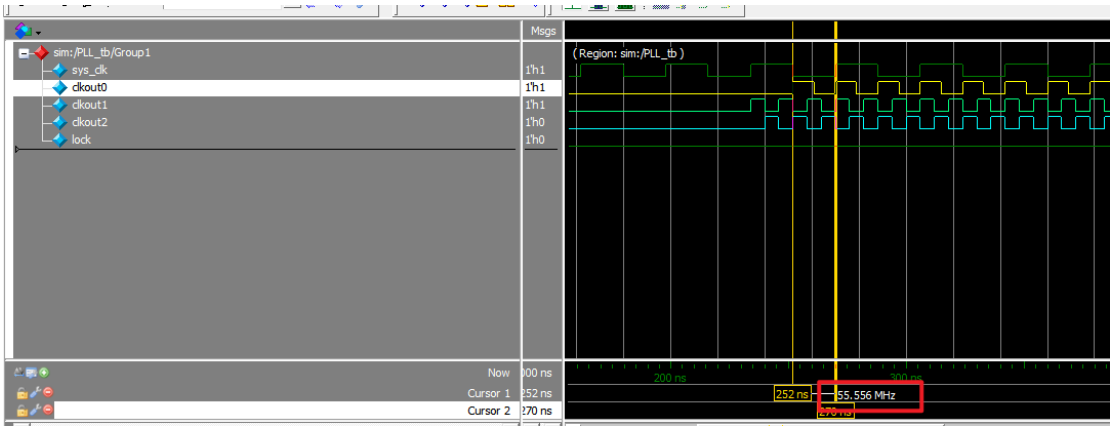


图 5.2-8 PLL IP 使用实验结果波形图 1

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。

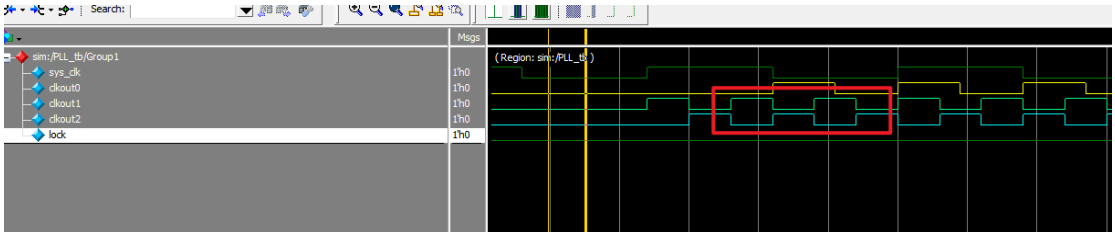


图 5.2-9 PLL IP 使用实验结果波形图 2

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

PDS2025.2

硬件环境：

PG2K100-6IMBG400

6.2.实验原理

了解 RAM、ROM、FIFO 是通用 IP 使用方法。

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：

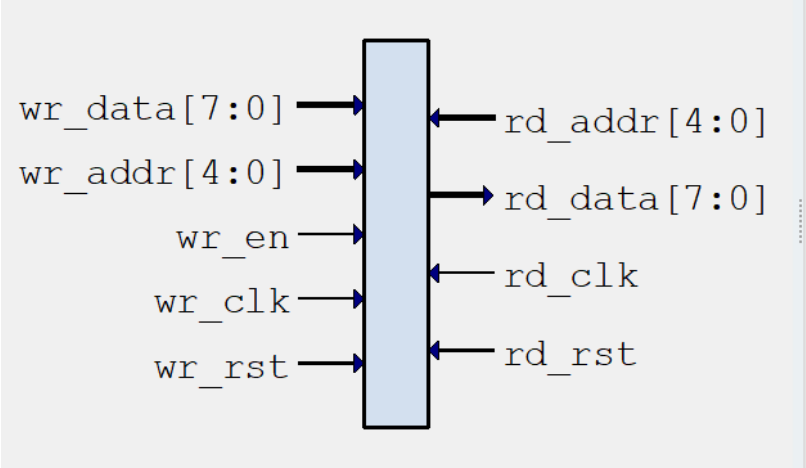


图 6.2-1

图 6.2-2 为 IP 配置：

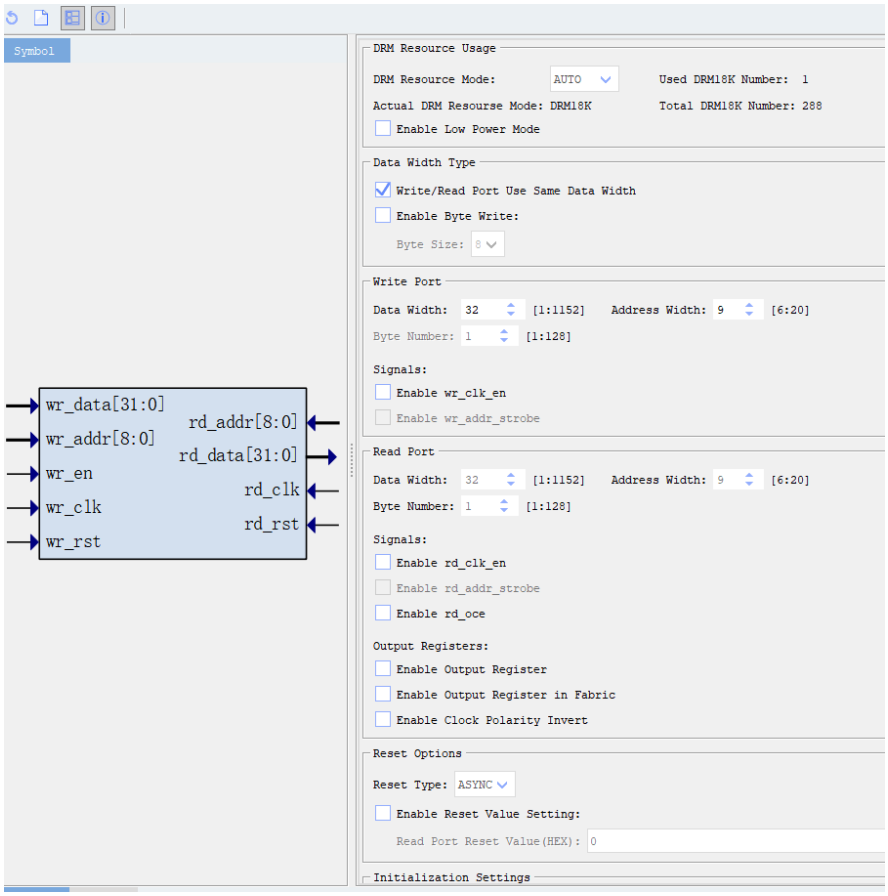


图 6.2-2

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。

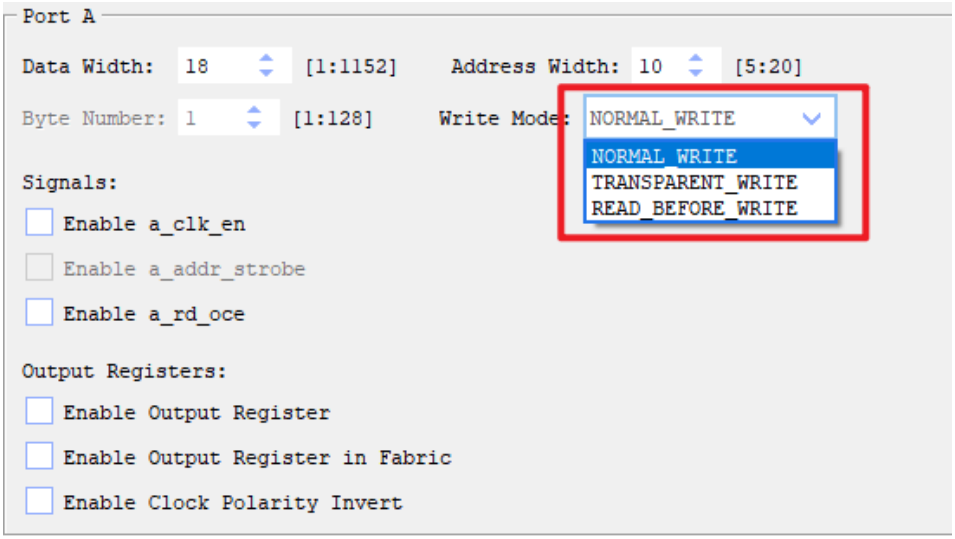


图 6.2-4

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。

NORMAL_WRITE

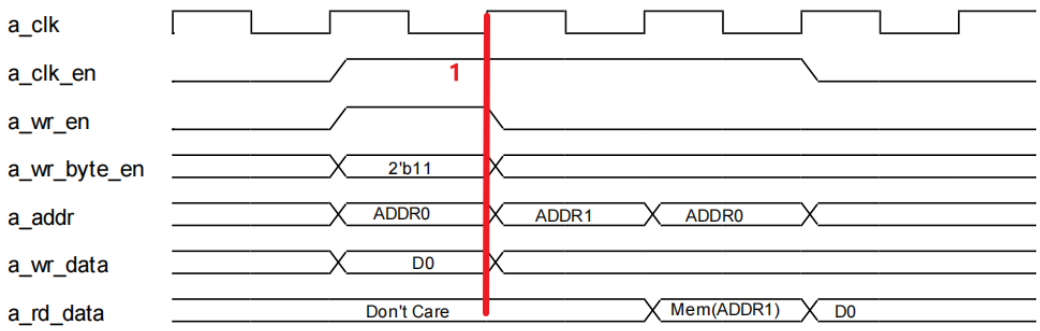


图 6.2-5

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。

6.2.1.1.1.READ_BEFORE_WRITE

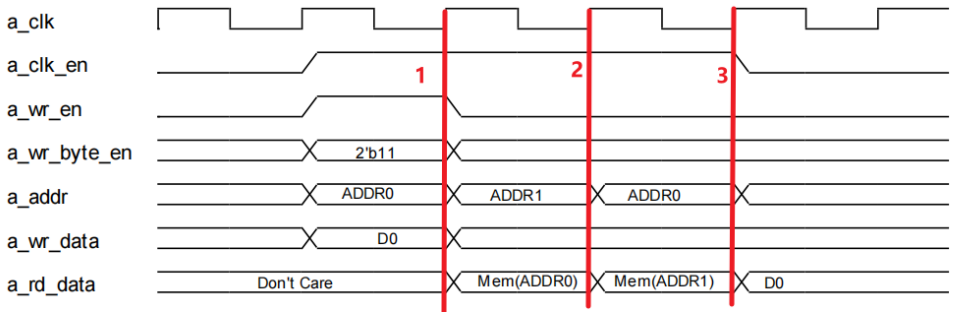


图 6.2-6

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。

6.2.1.1.2.Transparent_Write

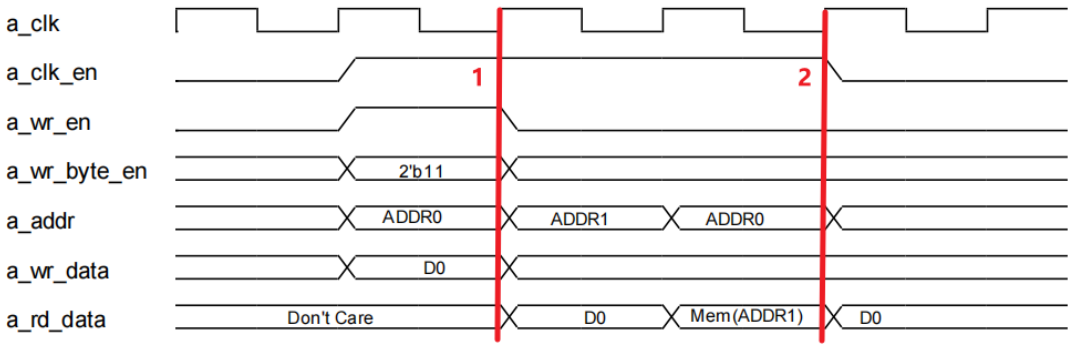


图 6.2-7

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。

6.2.1.1.3.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：

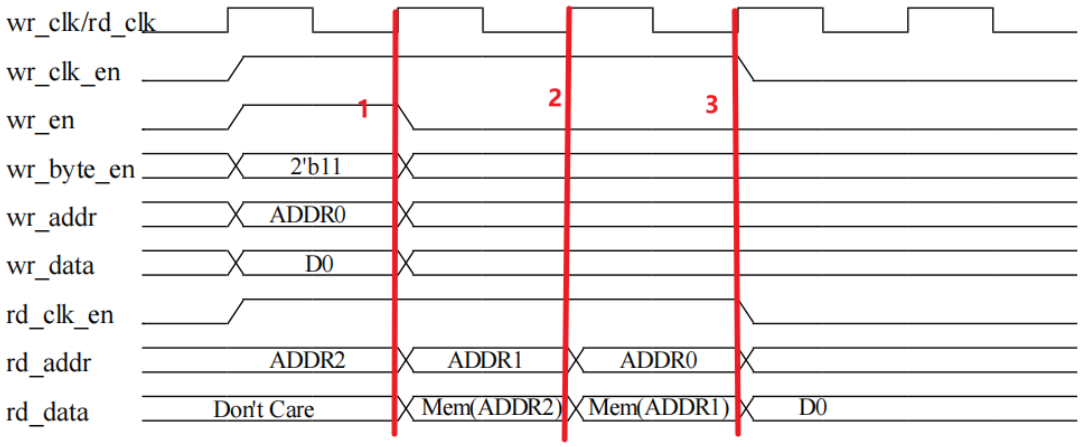


图 6.2-8

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。

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：

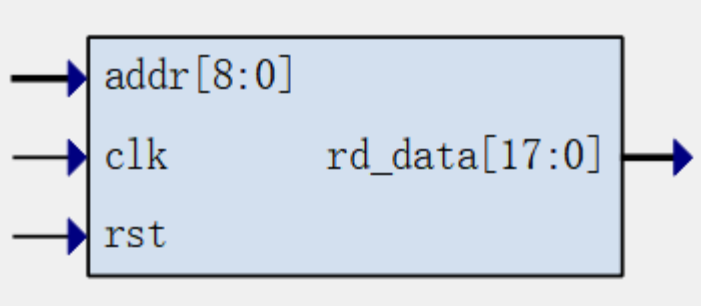


图 6.2-9

下图为 IP 配置：

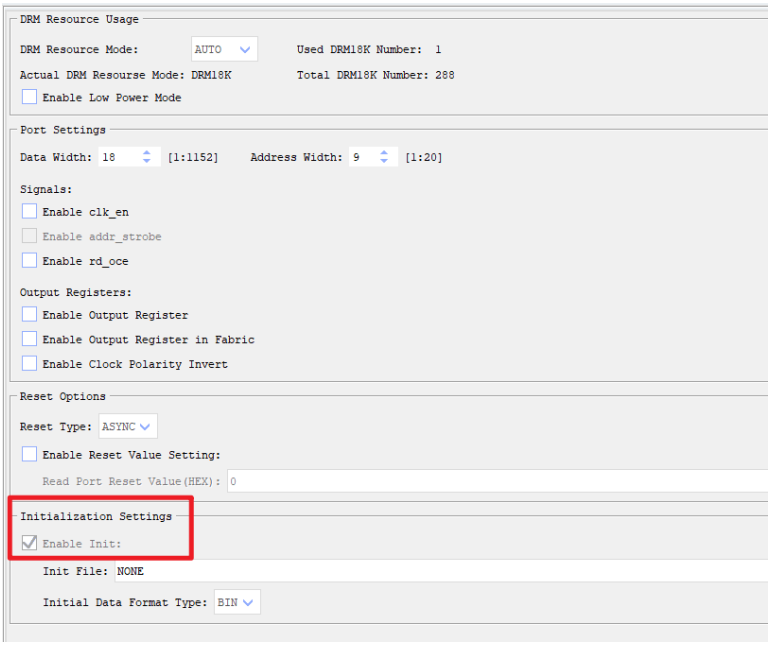


图 6.2-10

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。

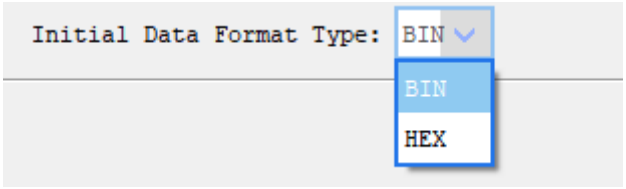


图 6.2-11

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

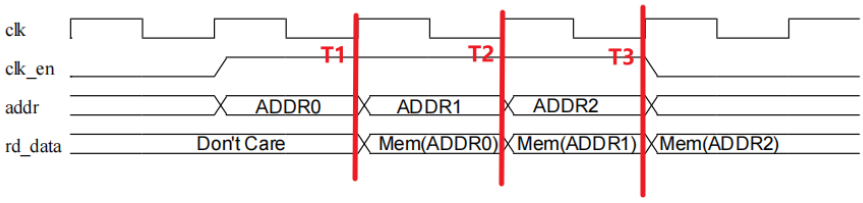


图 6.2-13

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。

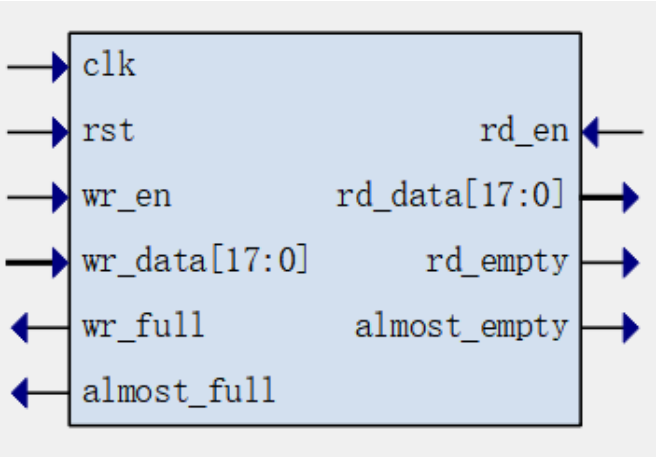


图 6.2-14

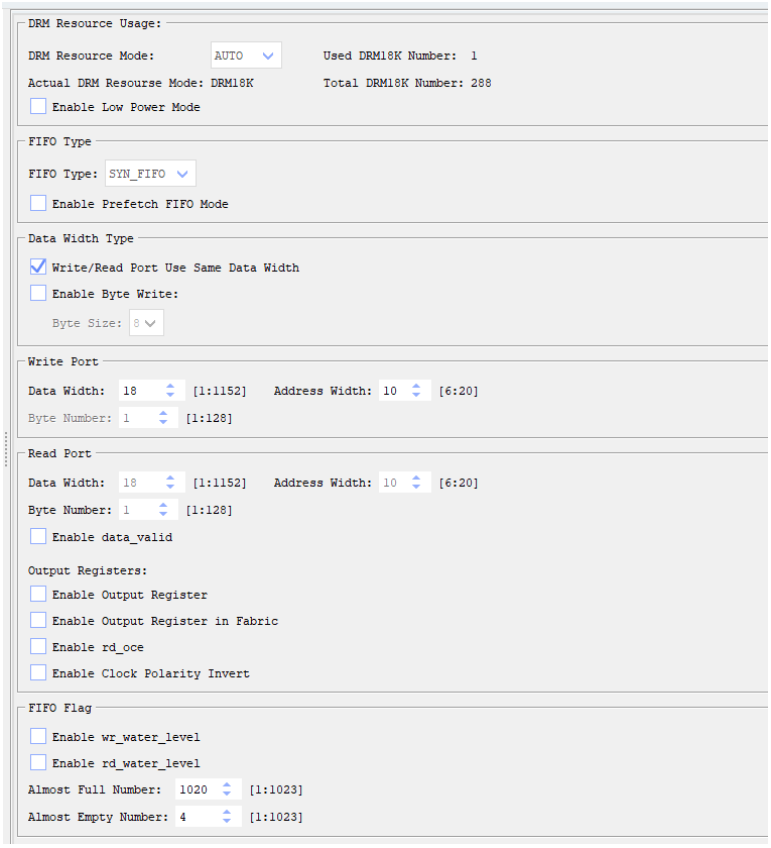


图 6.2-15

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。

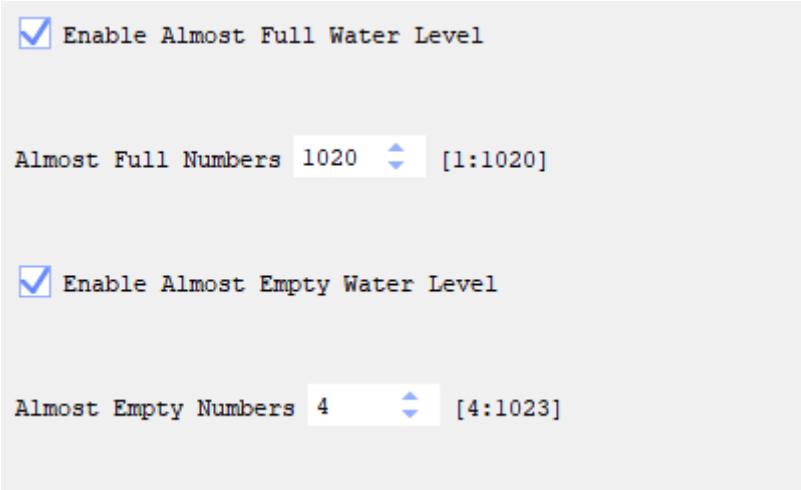


图 6.2-17

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

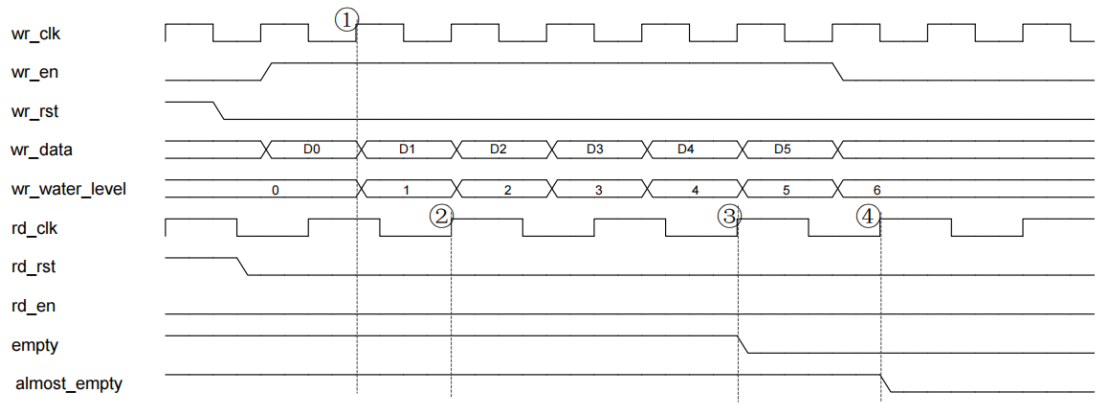


图 6.2-18

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

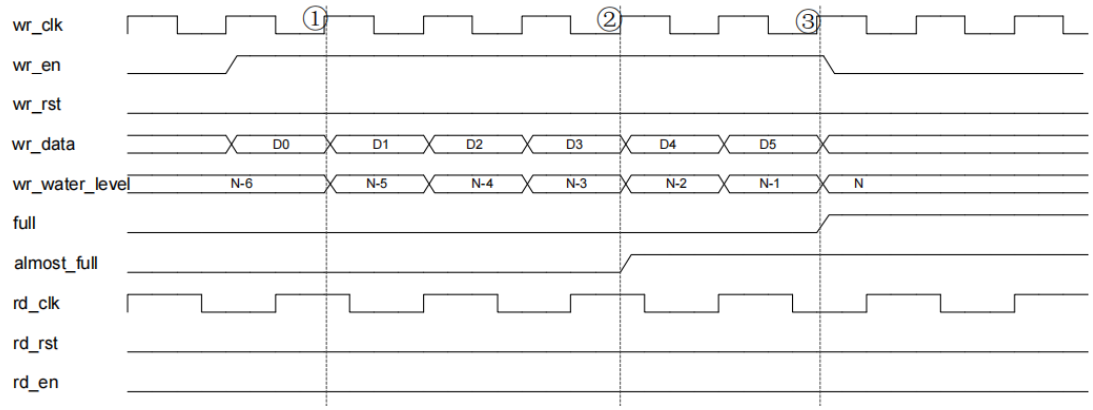


图 6.2-19

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

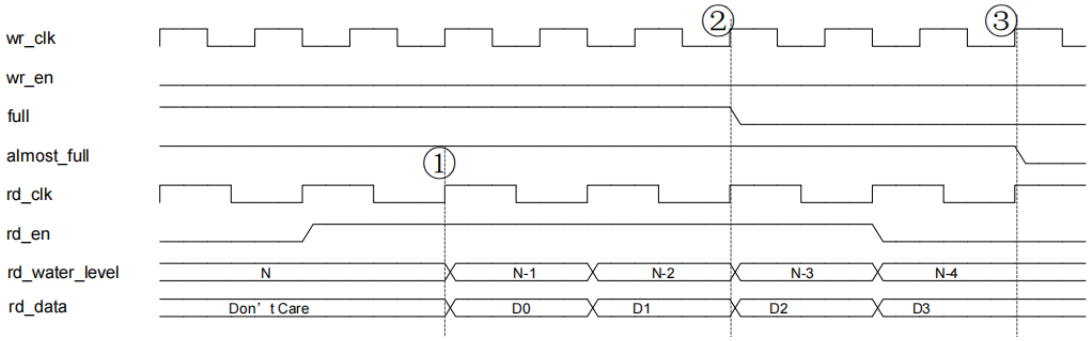


图 6.2-20

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

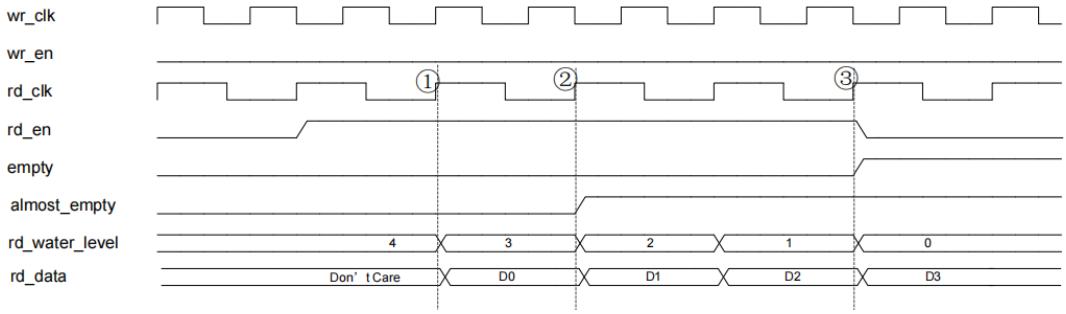


图 6.2-21

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 LED 灯按顺序依次点亮和熄灭。

实验环境：

Window11

PDS2025.2

硬件环境：

PG2K100-6IMBG400

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）。

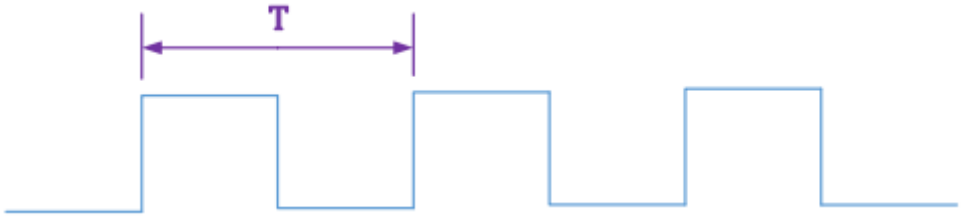


图 7.2-1

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。

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）

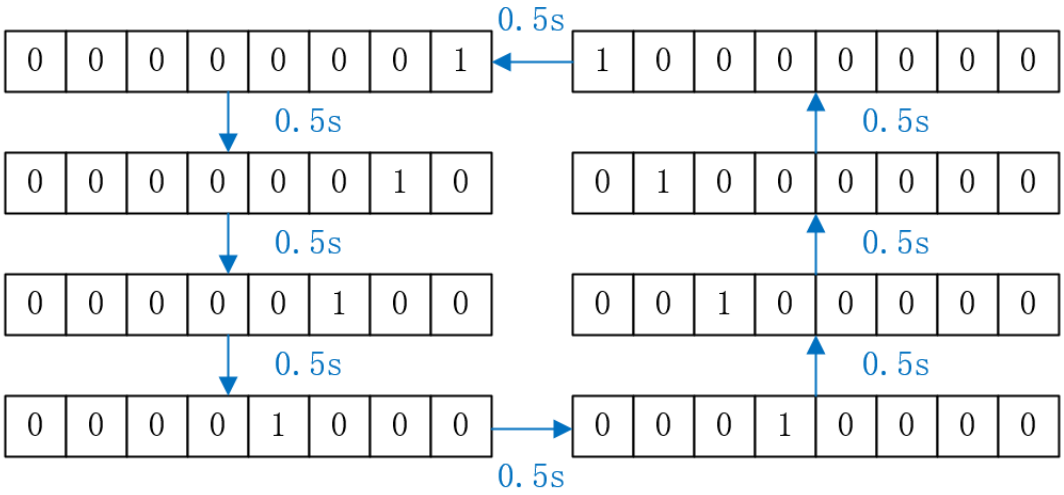


图 7.2-2

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

7.4.1.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 1350000，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。

7.4.2.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 1350 0000 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。

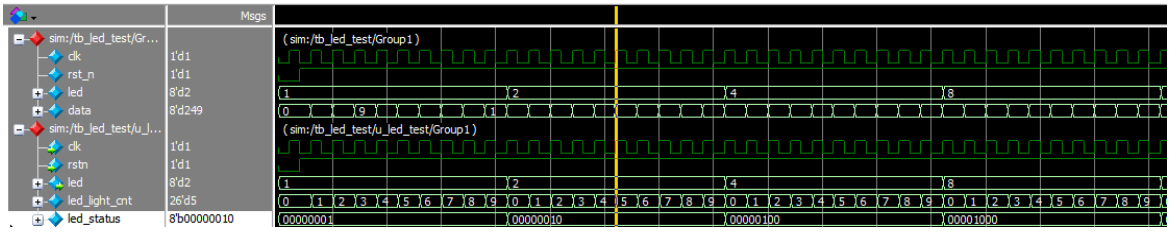


图 7.4-1

图 7.4-1 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。

7.5.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。

7.5.1.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；

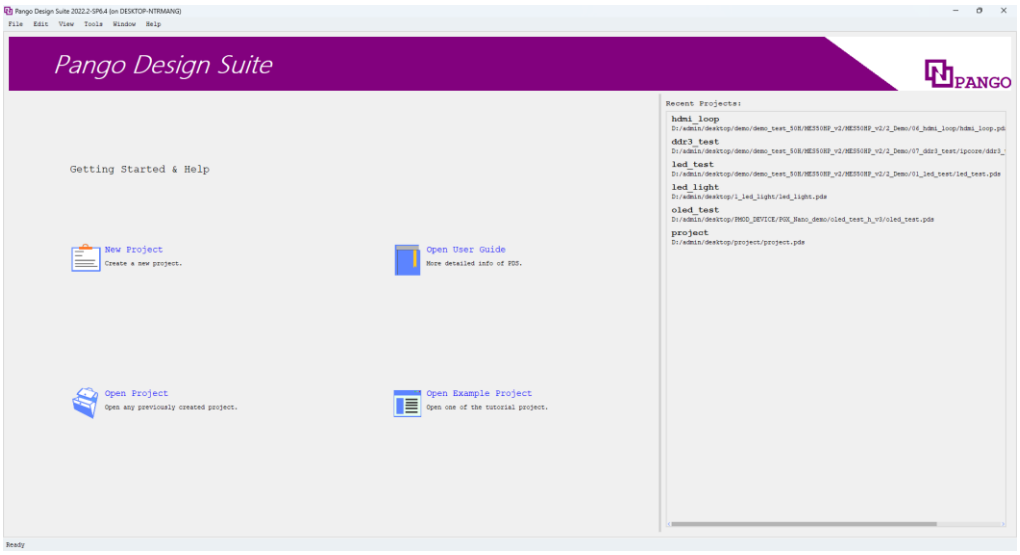


图 7.5-1

Step2：单击 NEXT；

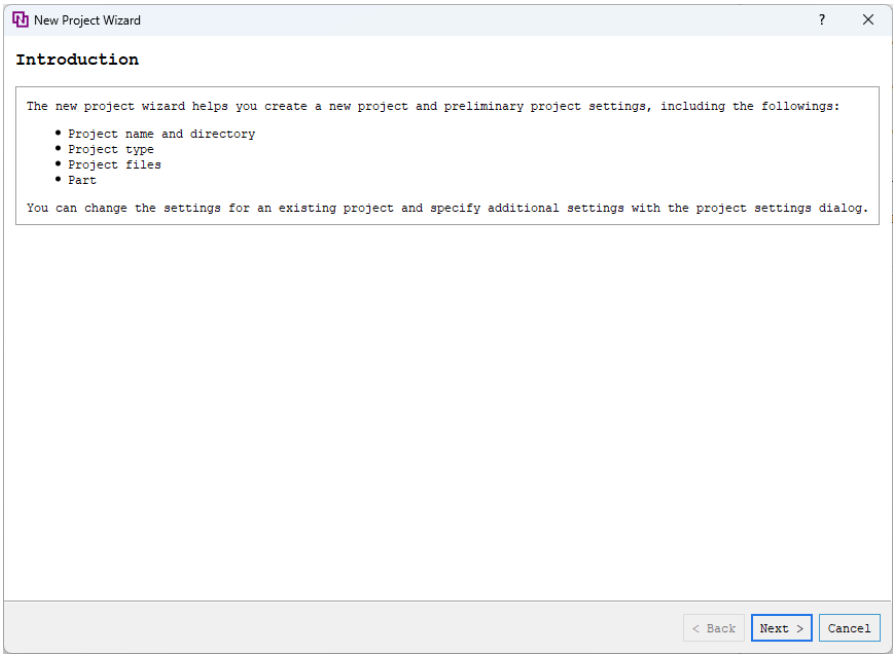


图 7.5-2

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。

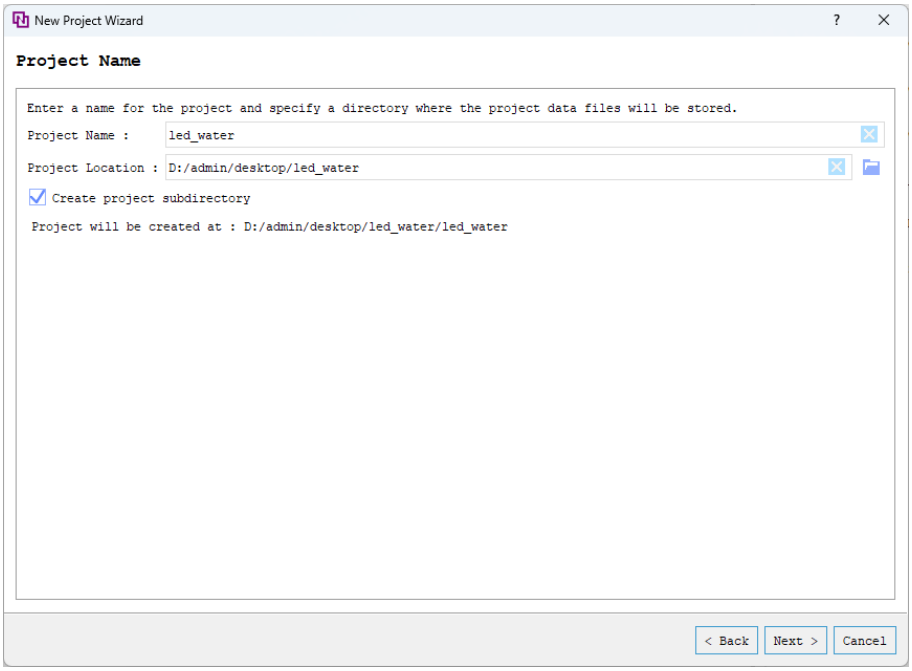


图 7.5-3

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。

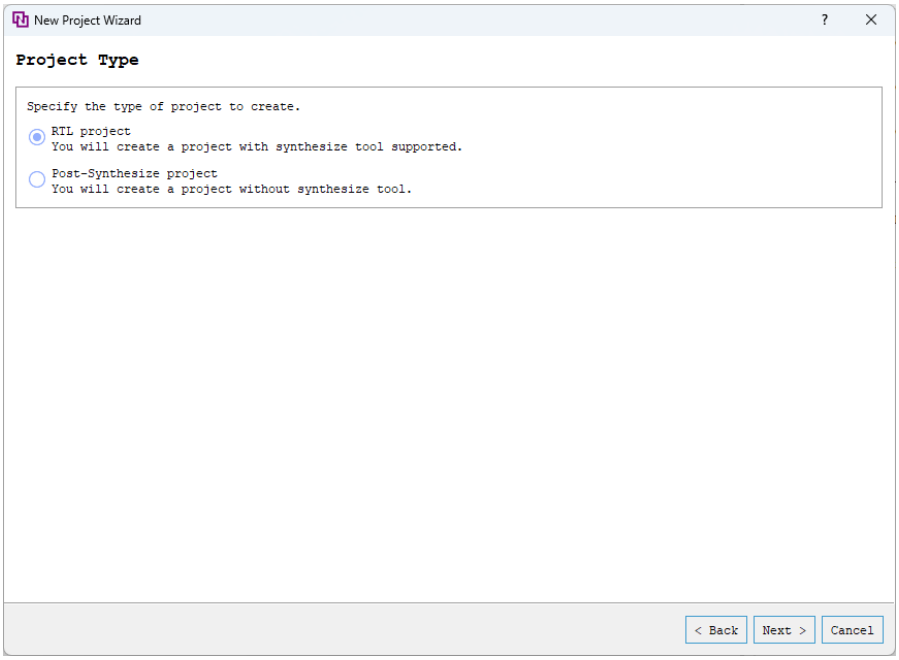


图 7.5-4

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。

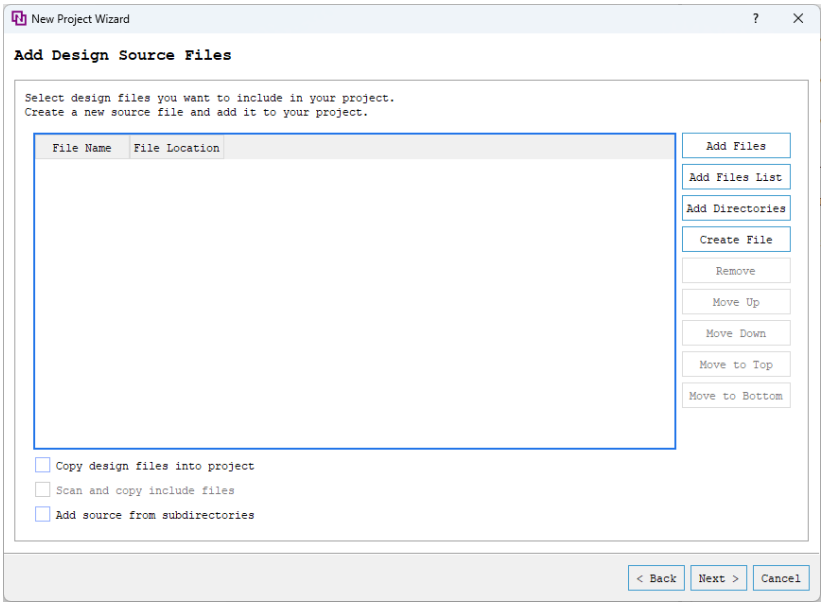


图 7.5-5

Step6：单击 Next；

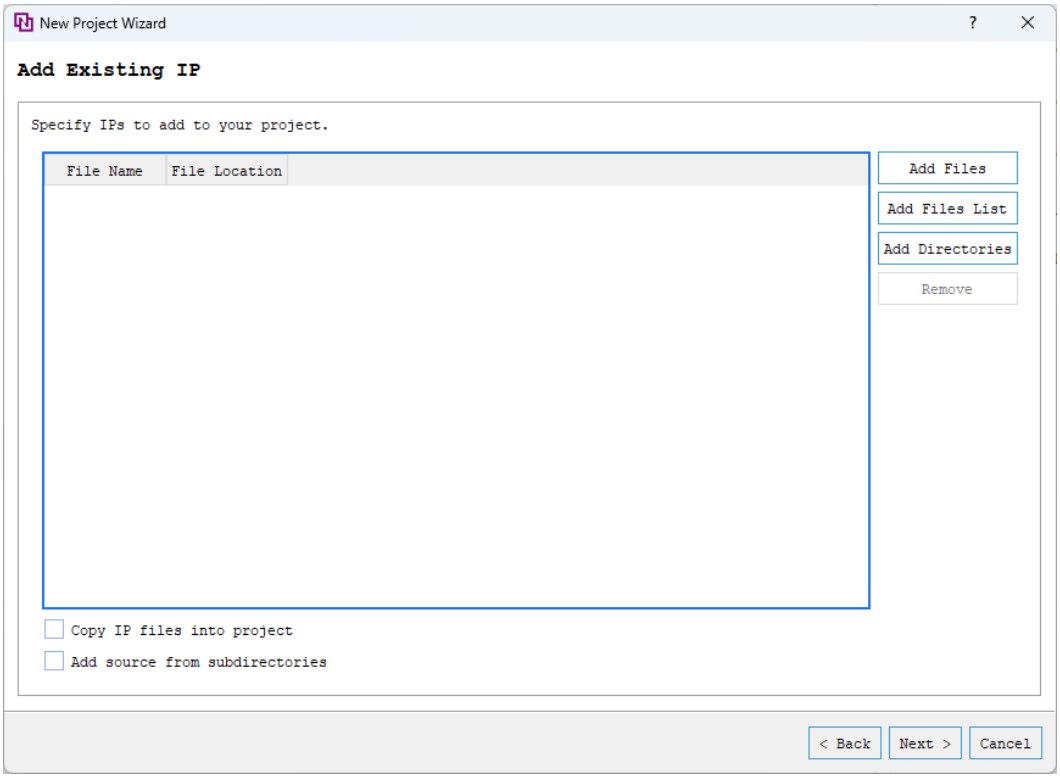


图 7.5-6

Step7：单击 Next；

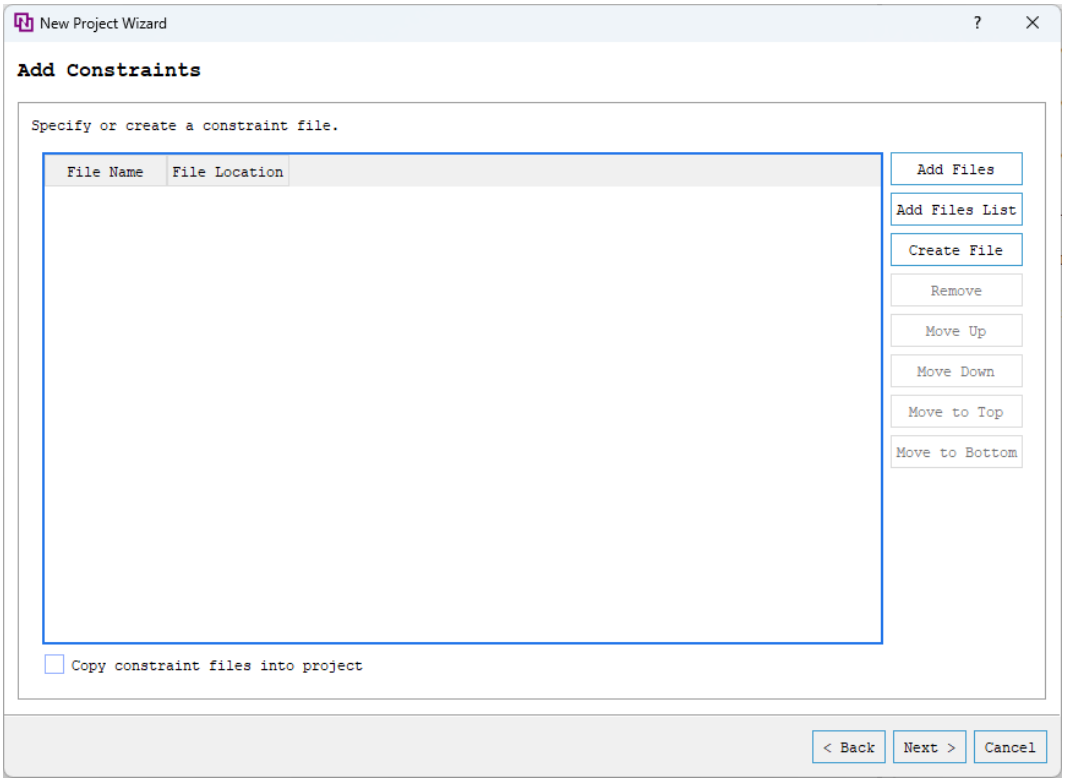


图 7.5-7

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。

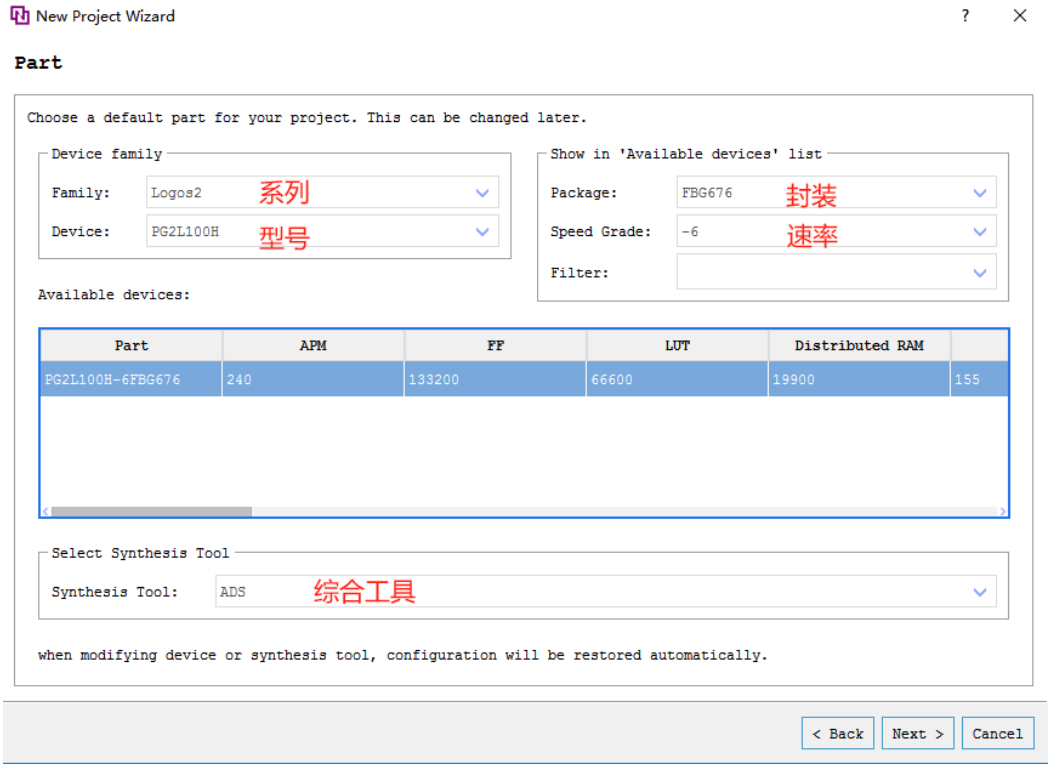


图 7.5-8

Step9：在 summary 单击 Finish，完成工程的创建；

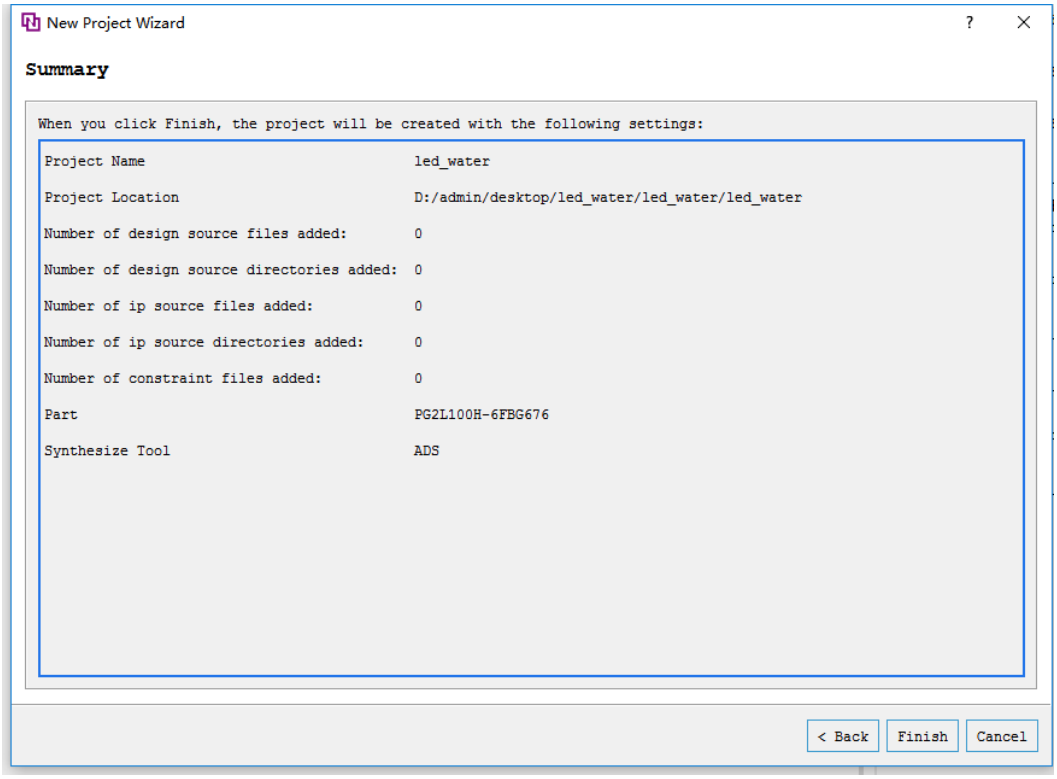


图 7.5-9

7.5.2.添加设计文件

PDS 软件界面如下图：

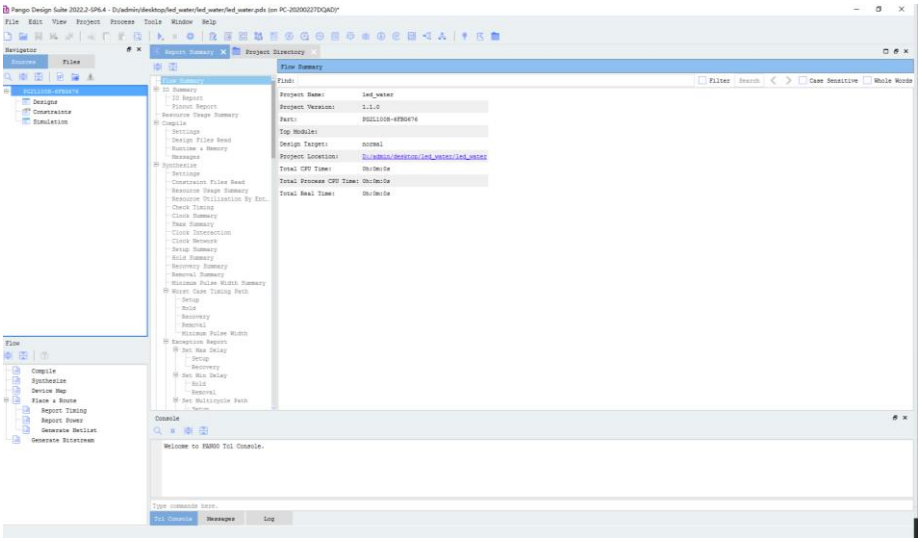


图 7.5-10

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；

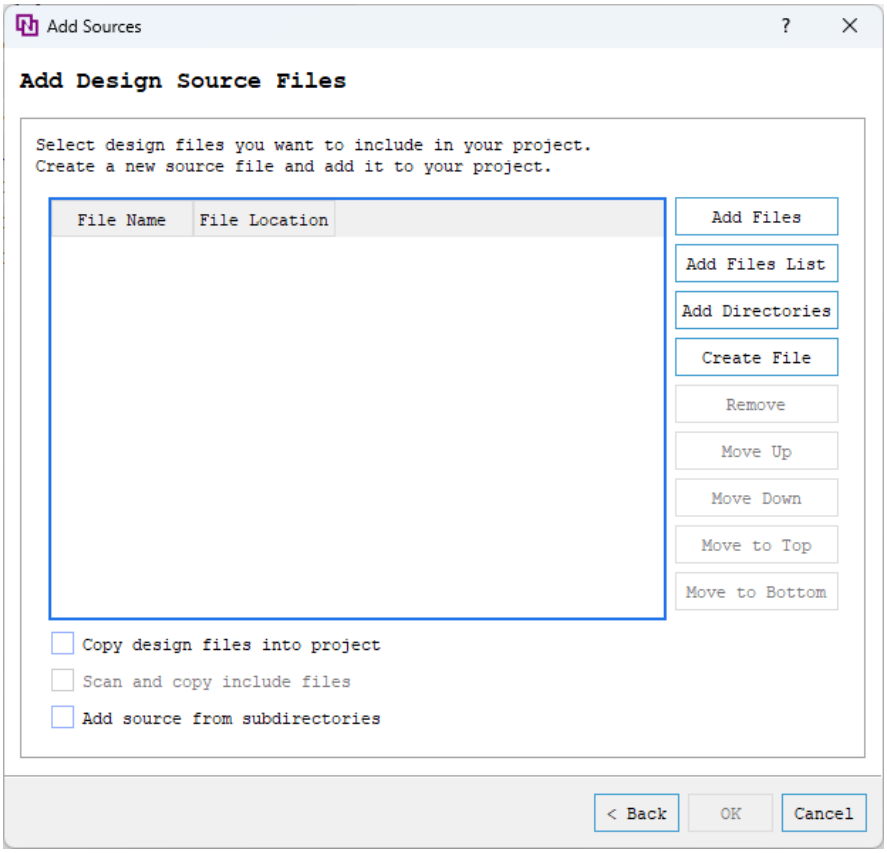


图 7.5-11

添加文件到工程：

在窗口中点击 Add Files，选择添加文件到工程；

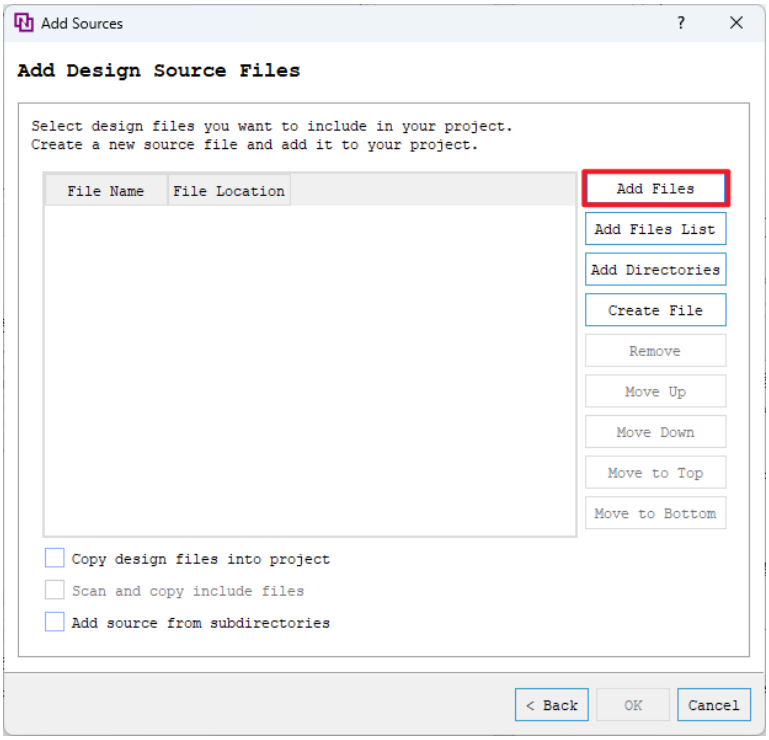


图 7.5-12

新建文件到工程：

1）在窗口中点击 Create File；

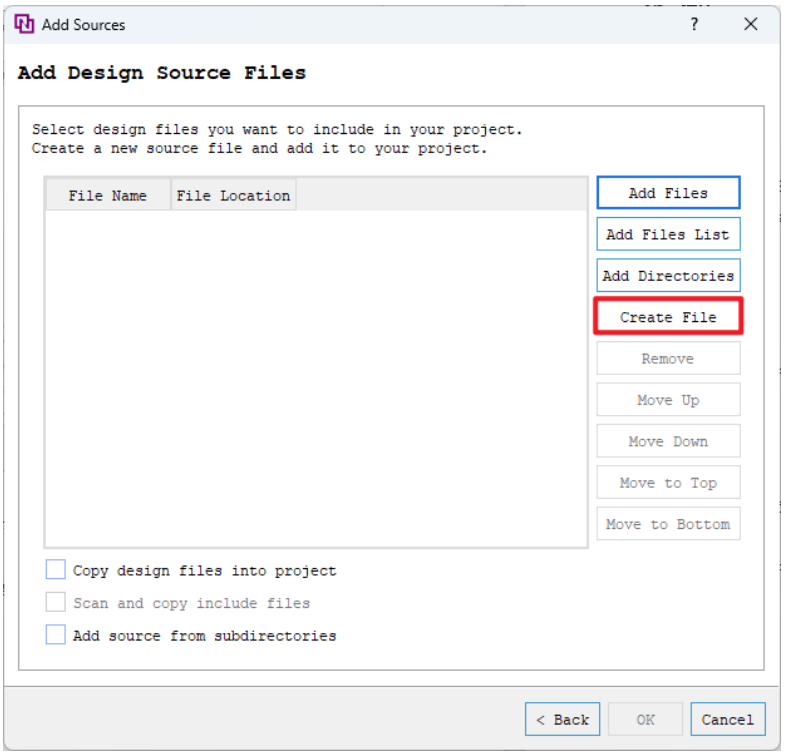


图 7.5-13

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

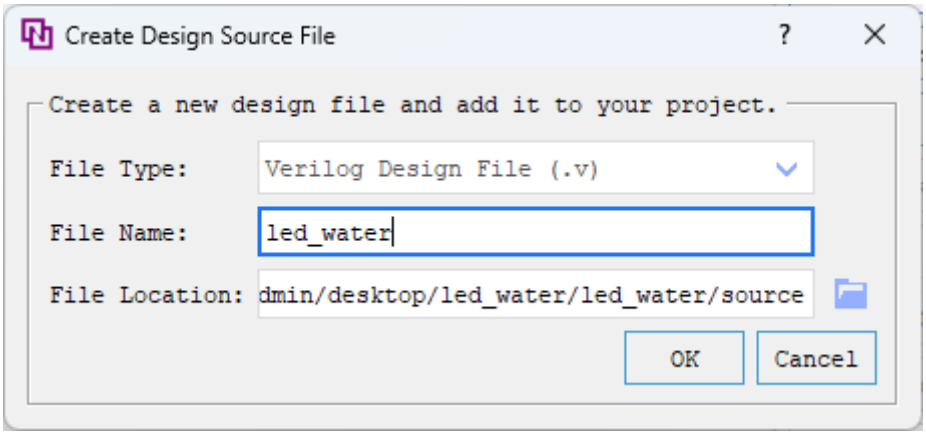


图 7.5-14

点击 OK；

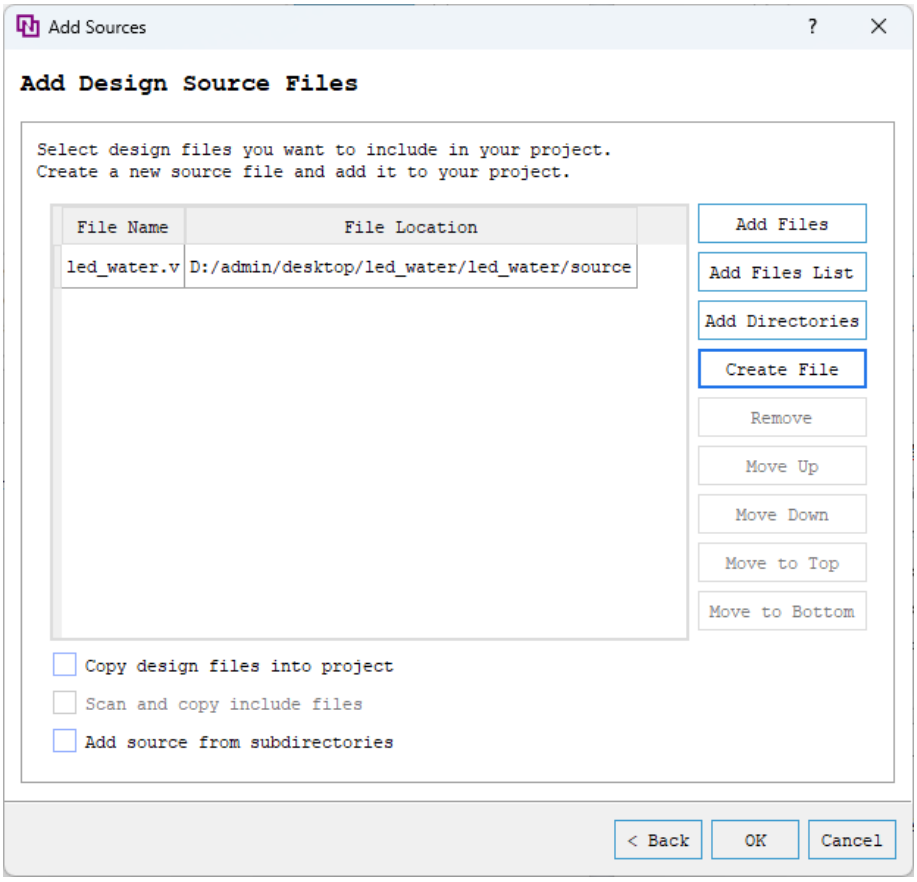


图 7.5-15

4）点击 Cancel；

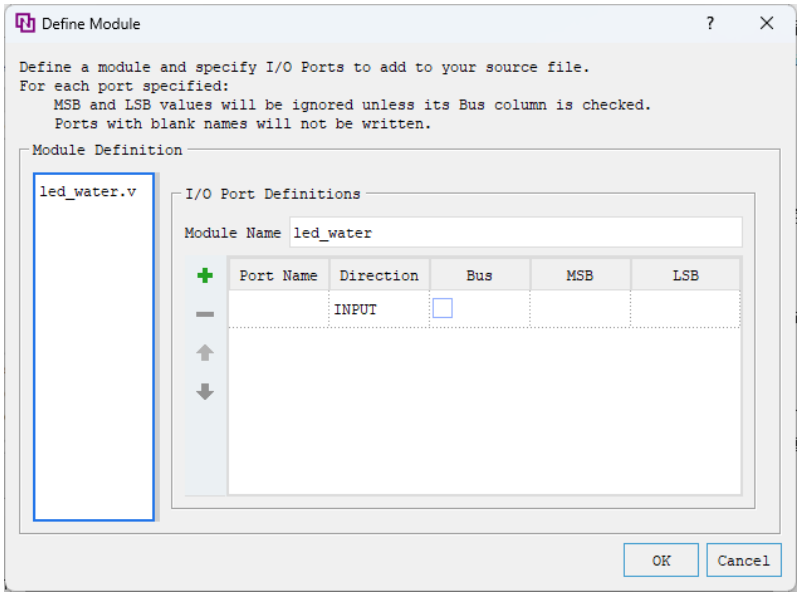


图 7.5-16

5）默认打开新建文件，将前面设计的 module 复制进去；

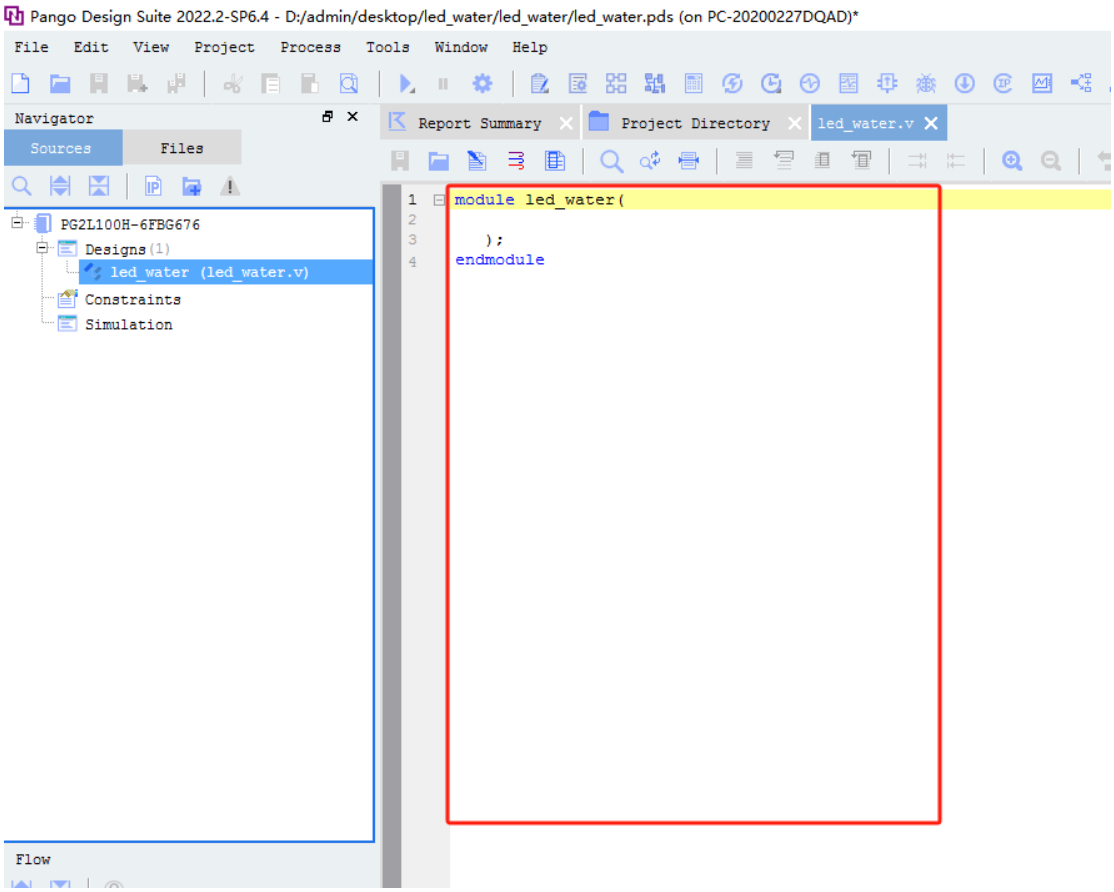


图 7.5-17

点击保存，新建文件完成。

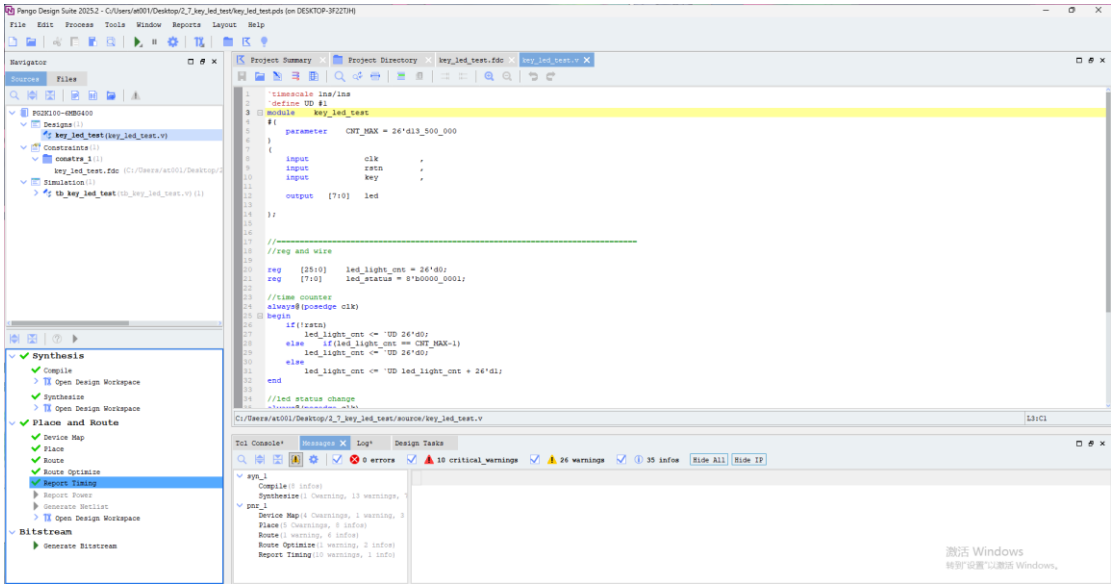


图 7.5-18

7.5.3.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；

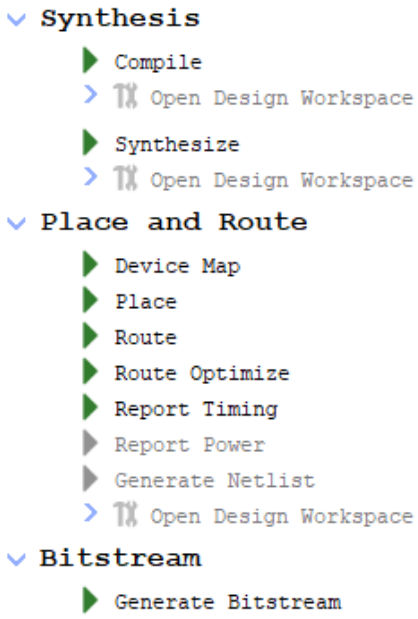


图 7.5-19

7.5.4.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：

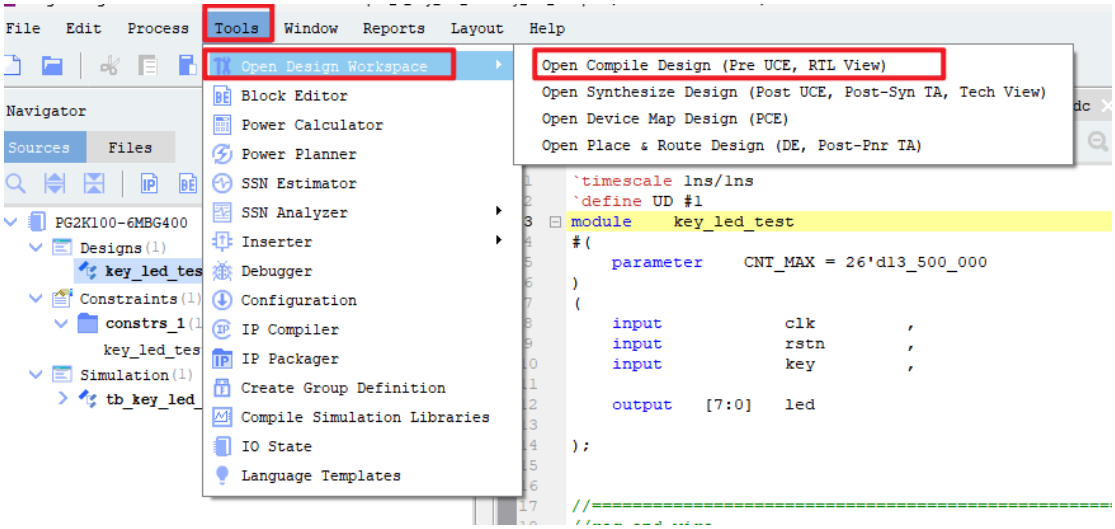


图 7.5-20

Tools 下的 User Constraint Editor(Timing and Logic)。

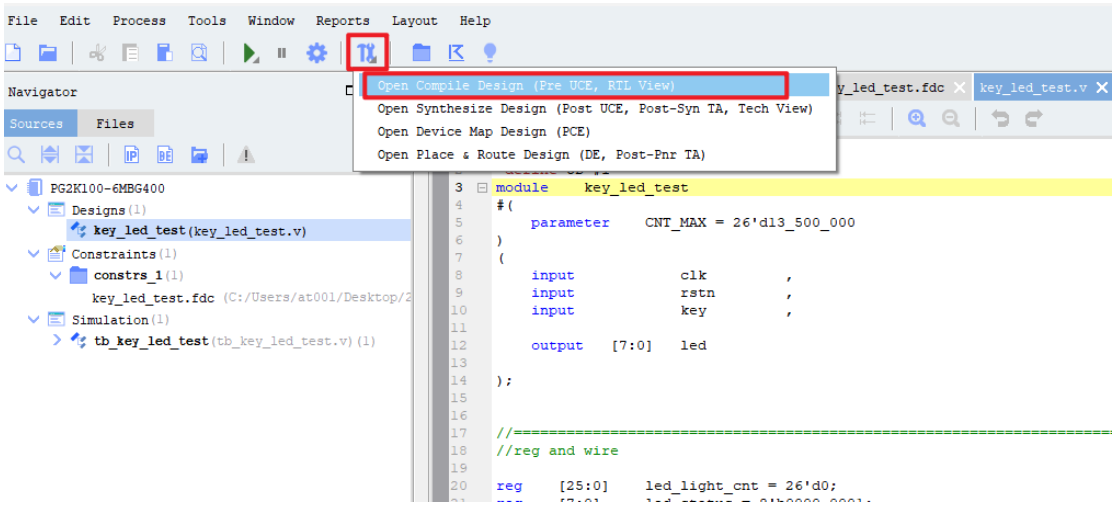


图 7.5-21

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

7.5.4.1.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。

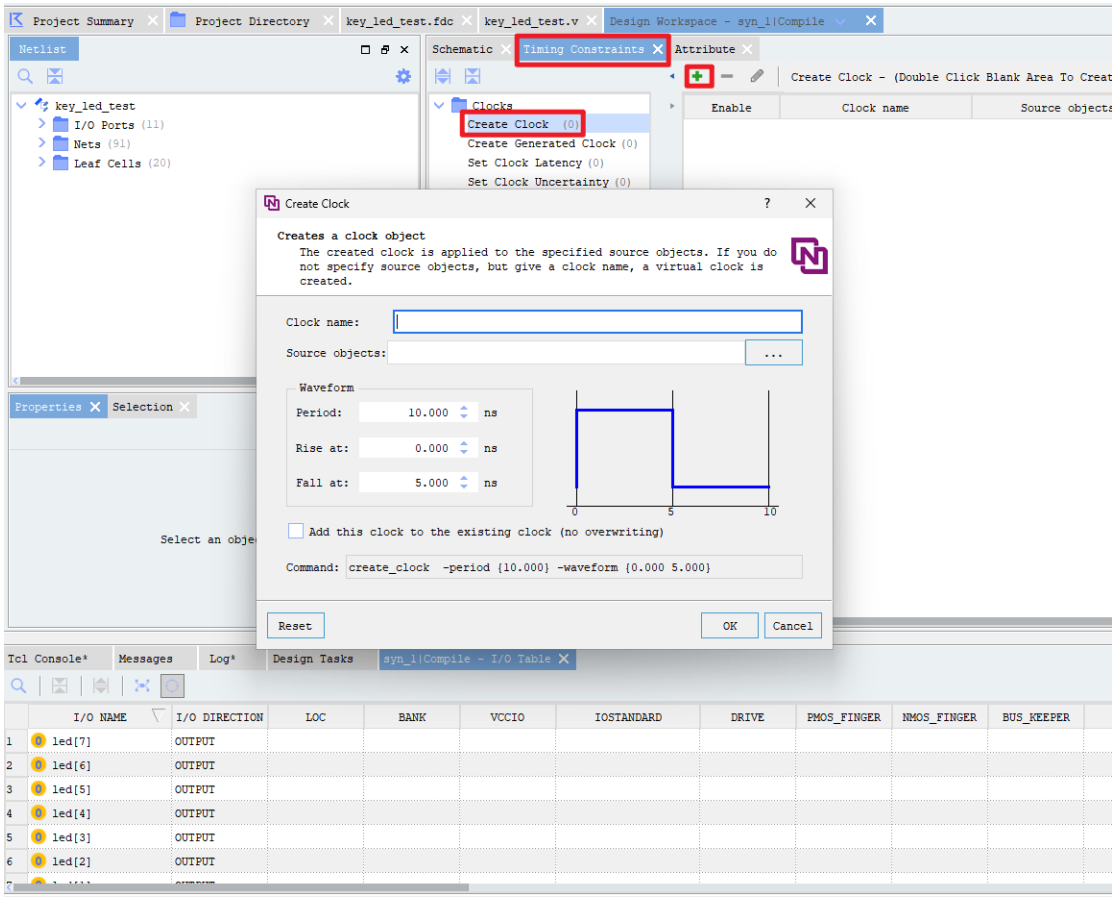


图 7.5-22

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：

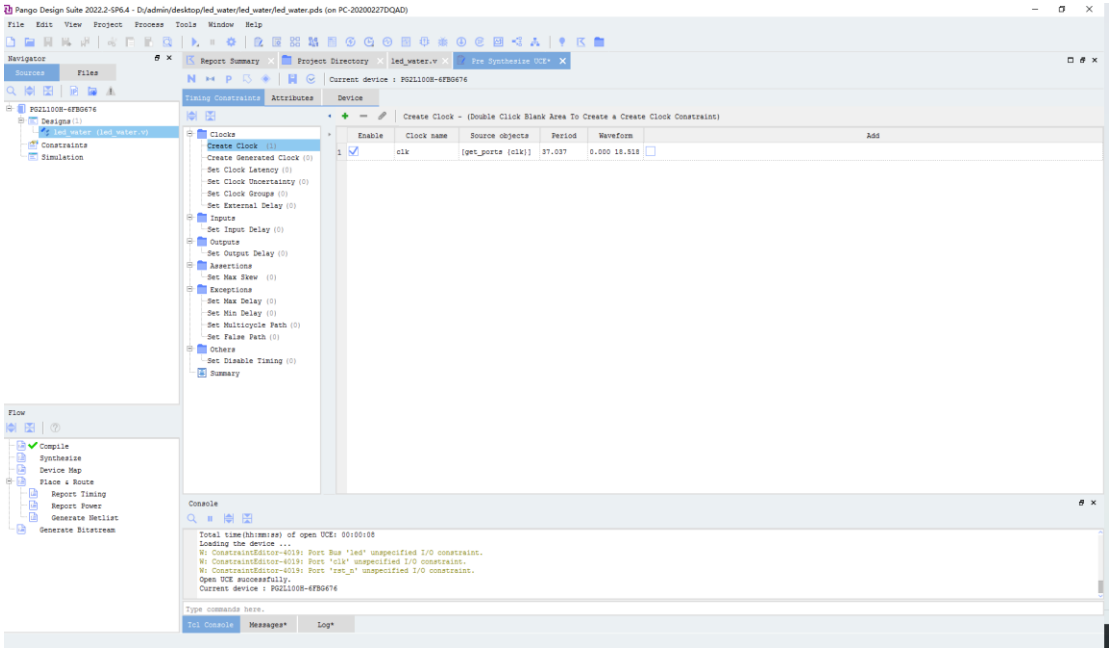


图 7.5-23

7.5.4.2.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。

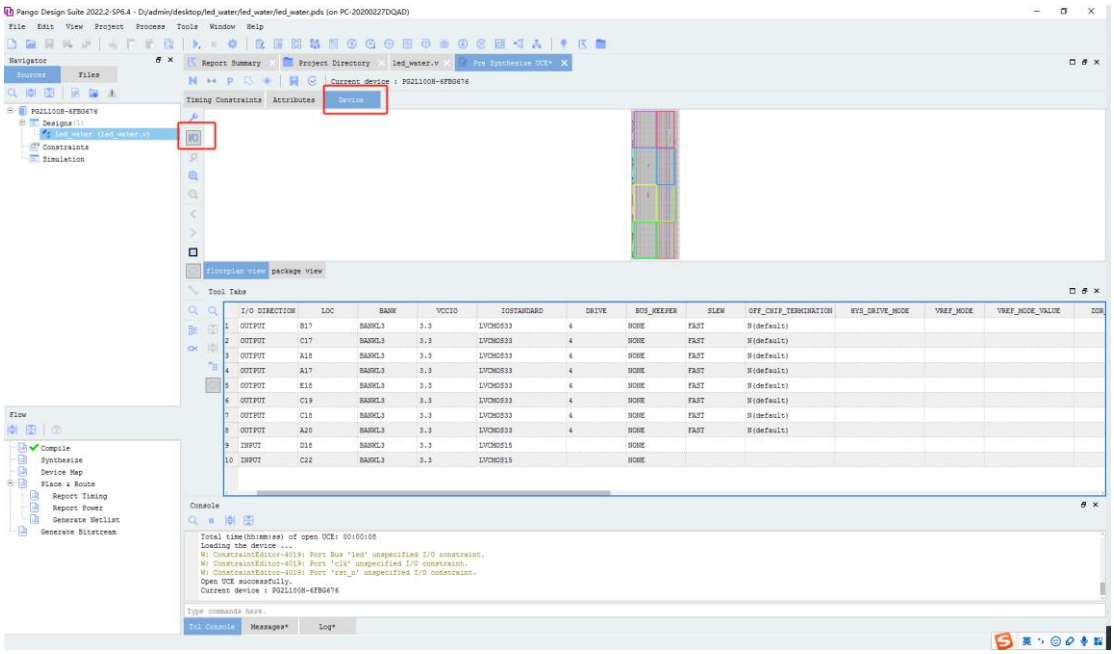


图 7.5-24

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。

7.5.5.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：

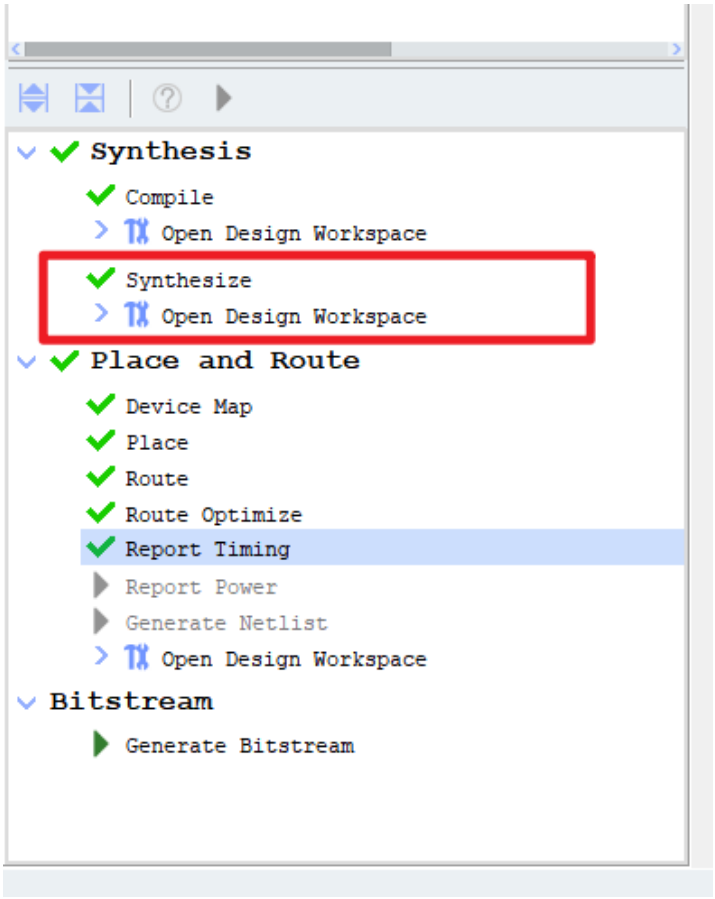


图 7.5-25

7.5.6.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：

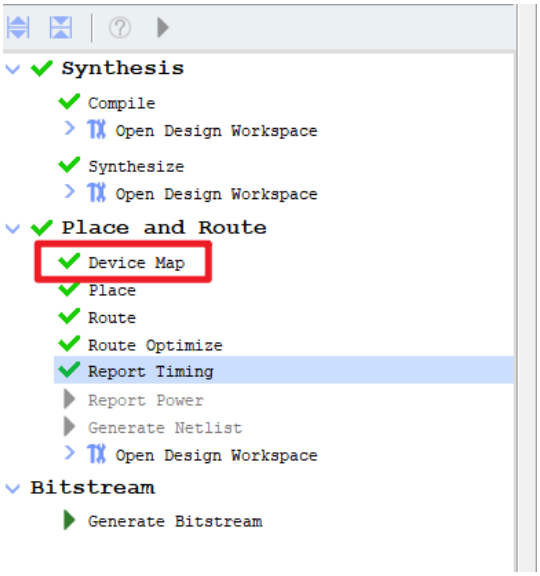


图 7.5-26

7.5.7.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：

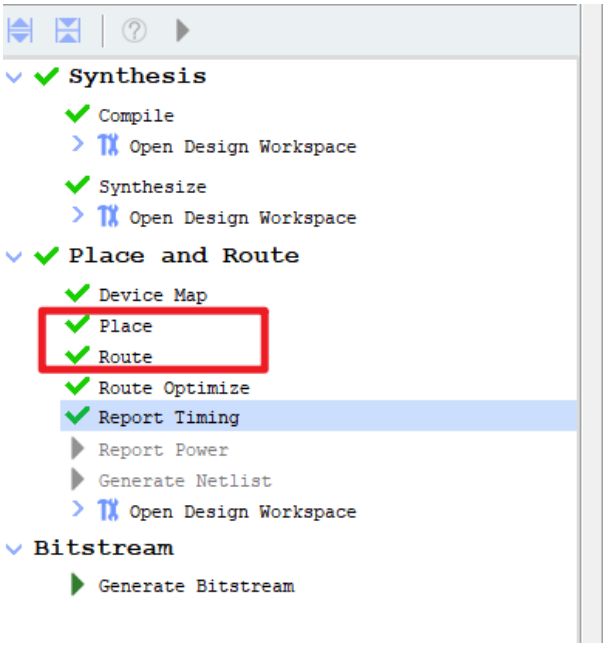


图 7.5-27

7.5.8.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：

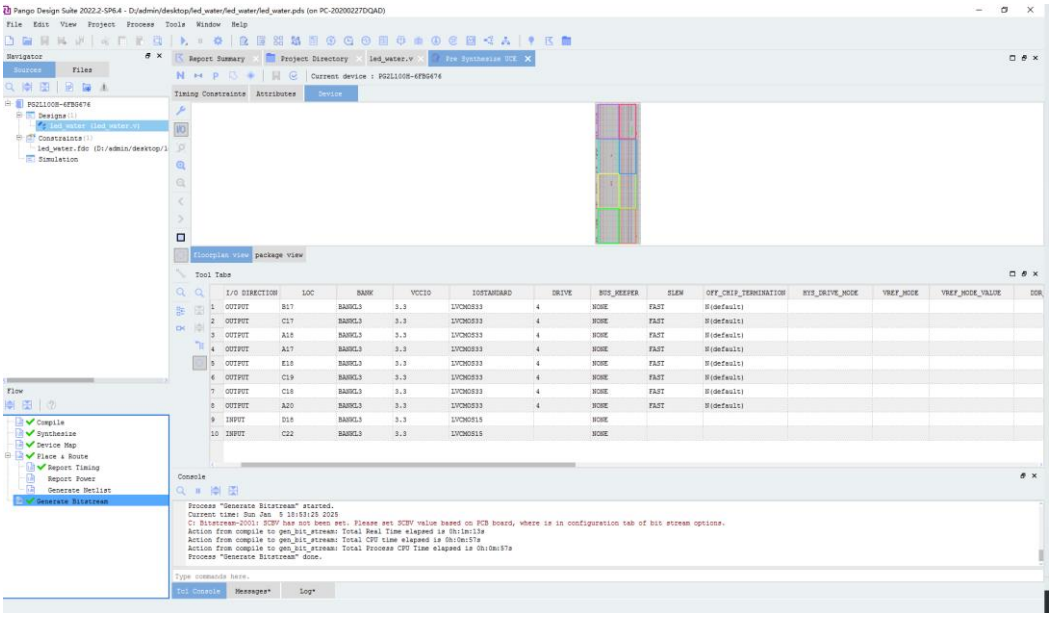


图 7.5-28

7.5.9.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：

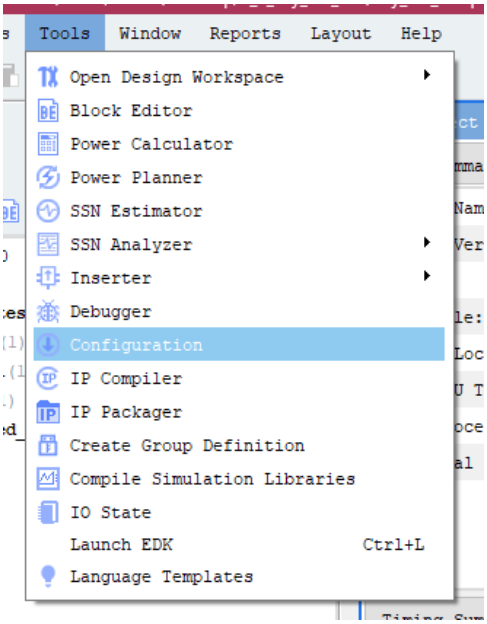


图 7.5-29

上图为 Tools 下的 Configuration 选项。

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；

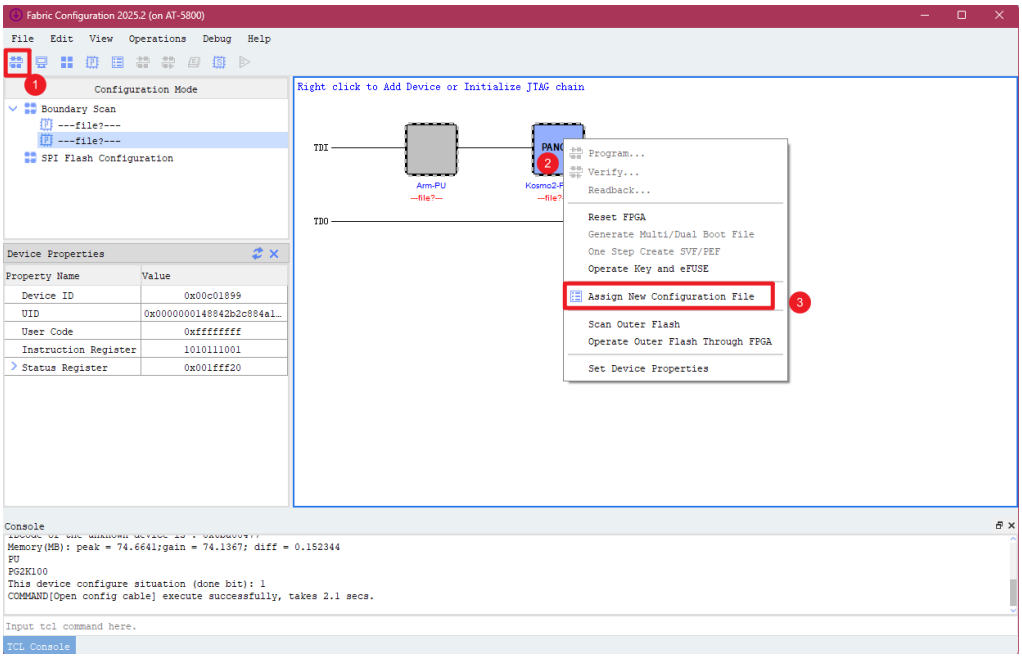


图 7.5-30

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：

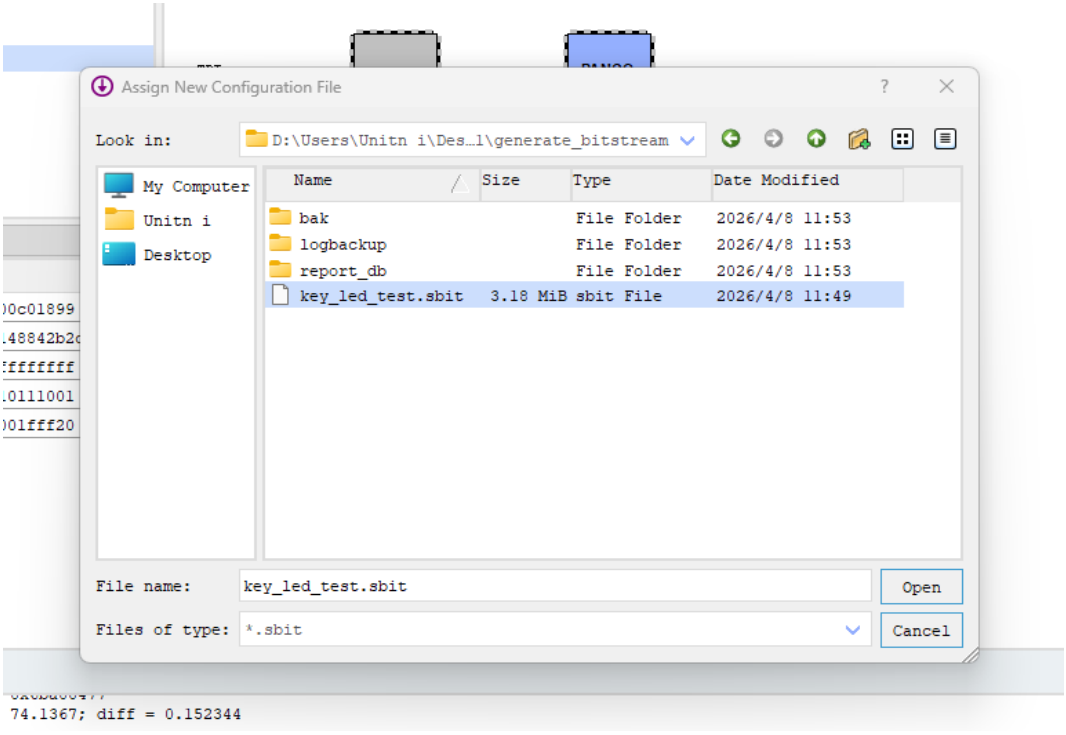


图 7.5-31

下载位流文件成功如下图所示：

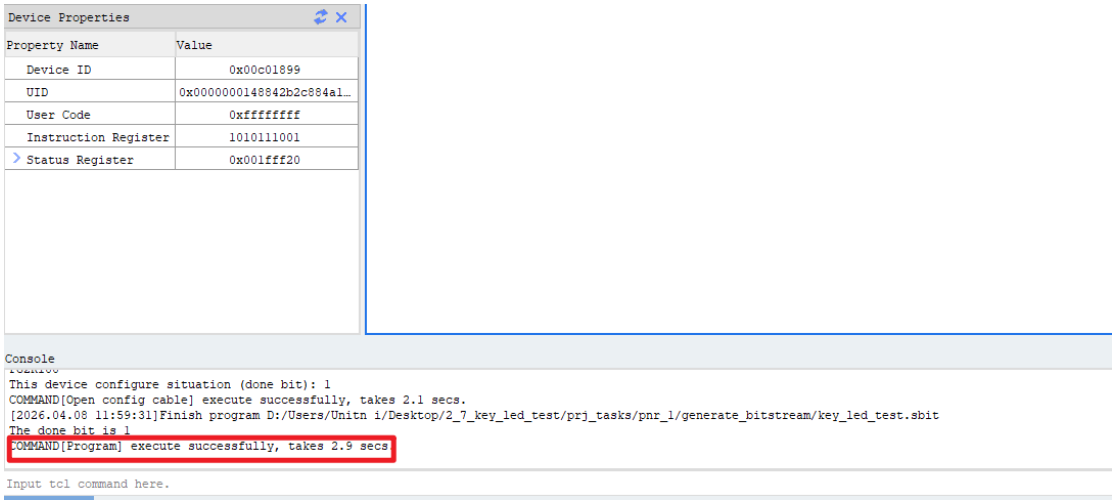


图 7.5-32

8.HDMI 回环实验

8.1.实验简介

实验目的：

完成 HDMI 回环实验

实验环境：

Window11

PDS2025.2

硬件环境：

PG2K100-6IMBG400

8.2.实验原理

HDMI 是一种数字化视频和音频接口技术，能够以极高的带宽同时传输未经压缩的高清视频数据和多声道音频信号。在物理连接层面，标准的 HDMI 链路主要由四对高速差分线组成，其中包含三个数据通道（Data Lanes）以及一个专用的时钟通道（Clock Lane）。这些通道共同协作，确保像素数据能够在发送端（如电脑、播放器）与接收端（如显示器、FPGA 开发板）之间保持精准的同步，而诸如 DDC（显示数据通道）和 HPD（热插拔检测）等辅助信号则负责处理显示设备的身份识别与连接状态感知。

HDMI 底层的信号传输协议，实际上采用的是名为 TMDS（最小化传输差分信号）的技术。TMDS 的核心逻辑在于通过特定的算法将 8 位原始数据编码为 10 位逻辑电平，这种做法的主要目的是最大限度地减少信号翻转次数，从而降低电磁干扰（EMI），并维持直流平衡，确保信号在长距离传输后仍能被准确还原。

LVDS（低电压差分信号）作为一种高性能的物理层接口标准，是理解本次实验原理的关键。它的工作原理基于在两根平行的信号线上传输相位相反的电流信号，接收端通过读取这两根线之间的电压差来判断逻辑状态。这种差分传输模式具备极强的抗干扰能力，因为环境中产生的共模噪声会同时作用于两根导线，而在接收端的减法运算中会被自然抵消。正因如此，LVDS 能够在极低的电压摆幅下实现每秒数吉比特（Gbps）的数据传输速率，同时保持极低的功耗和辐射，完美适配了高清视频对高带宽和稳定性的严苛要求。

8.2.1.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。

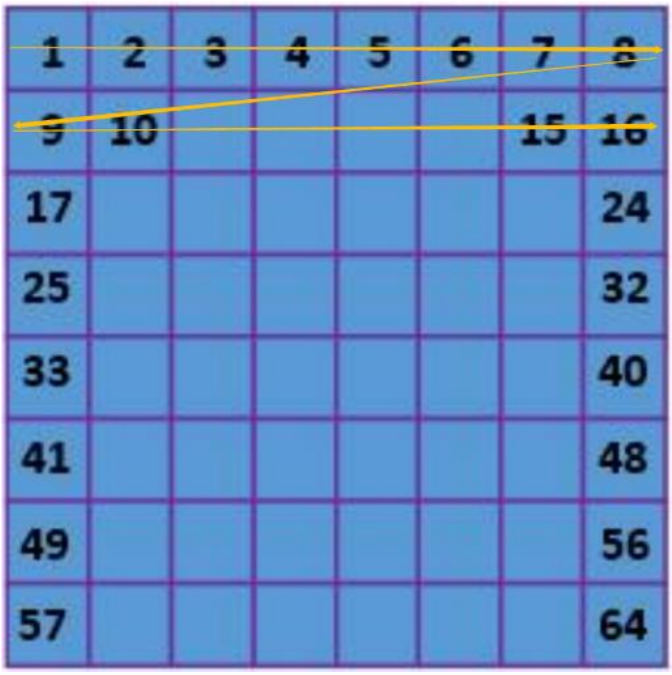


图 8.2-1

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。

8.3.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
i_rx_tmds_clk_n	input	1	HDMI 差分输入时钟 N 端
i_rx_tmds_clk_p	input	1	HDMI 差分输入时钟 P 端
[2:0] i_rx_tmds_data_n	input	3	HDMI_IN 输入差分数据 N 端
[2:0] i_rx_tmds_data_p	input	3	HDMI_IN 输入差分数据 P 端
hdmi_rx_hpd	output	8	HDMI_IN HPD 信号
i2c_scl	inout	1	IIC CLK 信号
i2c_sda	inout	1	IIC SDA 信号
hdmi_oen	output	1	HDMI_TX 输出使能信号
tmds_clk_n	output	1	HDMI 差分输出时钟 N 端
tmds_clk_p	output	1	HDMI 差分输出时钟 P 端
tmds_data_n	output	3	HDMI_IN 输出差分数据 N 端
tmds_data_p	output	3	HDMI_IN 输出差分数据 P 端
[4:0] led	output	5	HDMI_EDID 状态

8.4.工程说明

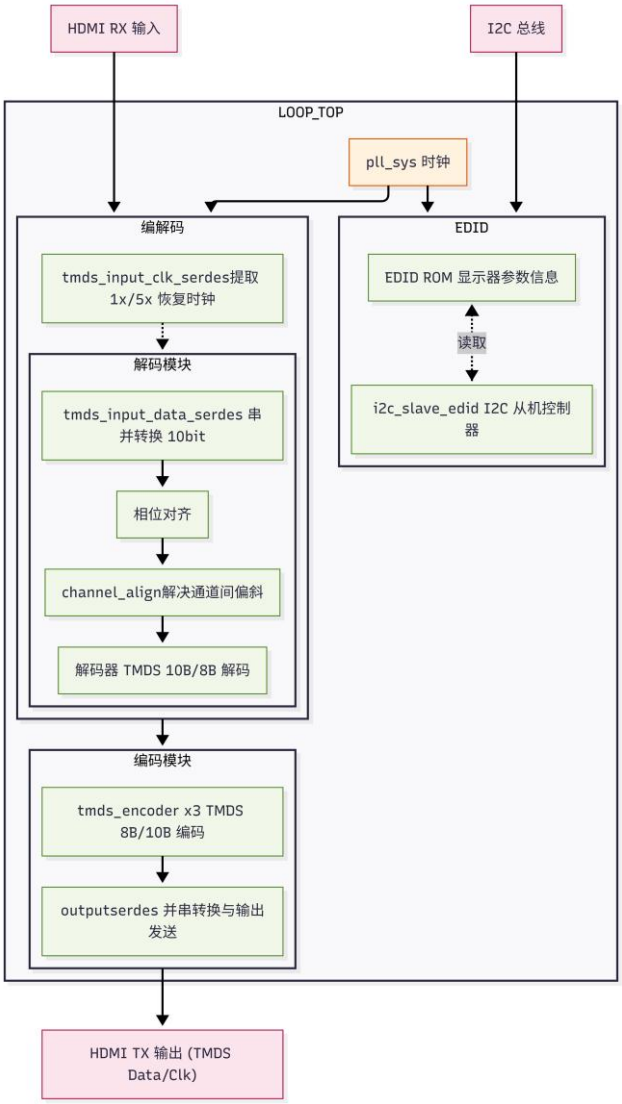


图 8.4-1

数据流概览，输入阶段 HDMI 端口的 TMDS 差分信号进入 tmds_input_clk_serdes，产生像素时钟 pixel_clk（74.25 MHz）和高速串行时钟 serial_clk（371.25 MHz）。phasealign 对每条 10-bit 串行数据执行 bitslip 与延迟校准，确保在像素时钟边沿捕获到完整的 10-bit 码字。对齐后的 10-bit 数据送入 tmds2rgb（decoder），得到 24-bit RGB 像素以及同步/使能信号。channel_fifo_10to10 可在像素时钟域提供 FIFO，缓冲时序抖动。rgb2tmds 将 RGB 像素重新编码为 10-bit TMDS 并行数据，内部使用 tmds_encoder 完成 8b/10b 编码。outputserdes 将并行的 10-bit 数据在 serial_clk 上串行化，驱动 TMDS 输出引脚形成 HDMI 输出信号。i2c_slave_edid 通过 I²C 总线向上位机提供 EDID ROM，决定 HDMI 源设备的分辨率与时序，独立于视频数据路径。

8.5.代码模块说明

edid 模块

```
`timescale 1ns / 1ps

module i2c_slave_edid(
    input  wire clk,        // System clock (e.g. 50MHz ~ 200MHz)
    input  wire rst_n,      // Active low reset
    inout  wire i2c_scl,    // I2C Clock input
    inout  wire i2c_sda,    // I2C Data inout
    output wire [4:0] dbg_led // Debug LEDs
);

    // ================================================================
    // I2C Signal Debouncer & Edge Detection (CRITICAL for slow I2C rise times)
    // ================================================================
    reg [4:0] scl_filter;
    reg [4:0] sda_filter;
    reg scl_clean, sda_clean;

    always @(posedge clk or negedge rst_n) begin
        if(!rst_n) begin
            scl_filter <= 5'b11111;
            sda_filter <= 5'b11111;
            scl_clean <= 1'b1;
            sda_clean <= 1'b1;
        end else begin
            scl_filter <= {scl_filter[3:0], i2c_scl};
            sda_filter <= {sda_filter[3:0], i2c_sda};

            // 连读 5 拍 (100ns) 稳定才认为电平翻转，完美过滤缓慢爬坡导致的毛刺
            if(scl_filter == 5'b11111) scl_clean <= 1'b1;
            else if(scl_filter == 5'b00000) scl_clean <= 1'b0;

            if(sda_filter == 5'b11111) sda_clean <= 1'b1;
            else if(sda_filter == 5'b00000) sda_clean <= 1'b0;
        end
```

```
end

reg [2:0] scl_r;
reg [2:0] sda_r;
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        scl_r <= 3'b111;
        sda_r <= 3'b111;
    end else begin
        scl_r <= {scl_r[1:0], scl_clean};
        sda_r <= {sda_r[1:0], sda_clean};
    end
end

wire scl_rise = (scl_r[2:1] == 2'b01);
wire scl_fall = (scl_r[2:1] == 2'b10);
wire sda_fall = (sda_r[2:1] == 2'b10);
wire sda_rise = (sda_r[2:1] == 2'b01);

// I2C Start and Stop condition detection
wire start_bit = sda_fall && (scl_r[1] == 1'b1);
wire stop_bit  = sda_rise && (scl_r[1] == 1'b1);

// State Machine encoding
localparam S_IDLE     = 4'd0;
localparam S_DEV_ADDR = 4'd1;
localparam S_ACK1     = 4'd2;
localparam S_REG_ADDR = 4'd3;
localparam S_ACK2     = 4'd4;
localparam S_WRITE    = 4'd5;
localparam S_ACK4     = 4'd6;
localparam S_READ     = 4'd7;
localparam S_ACK3     = 4'd8;

reg [3:0] state;
reg [3:0] bit_cnt;
reg [7:0] shift_reg;
reg [7:0] addr_reg;
```

```
reg          is_read;

wire [7:0] rd_data;

// ========================================
//   ROM IP 例化
// ========================================
EDID EDID_inst (
   .addr({2'b00, addr_reg}), // input [9:0] (高位补 0 凑齐 10 位)
   .clk(clk),                 // input
   .rst(~rst_n),              // input 高有效复位
   .rd_data(rd_data)          // output [7:0]
);

reg sda_en_reg;
reg sda_out_reg;

// 数据读取状态时，动态非阻塞输出 ROM 中的当前比特位
// SCL 变低瞬间地址改变，ROM 的数据会有 1 cycle(20ns) 的等待时间
// 这个 20ns 期间 I2C 时钟处于低电平(主设备不会采样)，完美契合
wire sda_out_val = (state == S_READ) ? rd_data[7 - bit_cnt[2:0]] : sda_out_reg;

// I2C 必须是开漏(Open-Drain)输出，只能驱动 0，靠外部上拉电阻拉高 1
assign i2c_sda = (sda_en_reg && (sda_out_val == 1'b0)) ? 1'b0 : 1'bz;

always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        state <= S_IDLE;
        bit_cnt <= 0;
        sda_en_reg <= 0;
        sda_out_reg <= 1;
        addr_reg <= 0;
        is_read <= 0;
        shift_reg <= 0;
    end else begin
        if(start_bit) begin
    state <= S_DEV_ADDR;
    bit_cnt <= 0;
```

```
            sda_en_reg <= 0;
        end else if(stop_bit) begin
            state <= S_IDLE;
            sda_en_reg <= 0;
        end else if(scl_rise) begin
    if(state == S_DEV_ADDR || state == S_REG_ADDR || state == S_WRITE) begin
            // sample on rising edge for receive
            shift_reg <= {shift_reg[6:0], sda_r[1]};
      end else if(state == S_ACK3) begin
            is_read <= sda_r[1]; // SCL rise on ACK3 (master ACK/NACK)
        end
end else if(scl_fall) begin
        case(state)
            S_IDLE: begin
                sda_en_reg <= 0;
            end
            S_DEV_ADDR: begin
                if(bit_cnt == 8) begin
                    if(shift_reg[7:1] == 7'h50) begin // 0xA0 寻址
                        state <= S_ACK1;
                        is_read <= shift_reg[0];
                        sda_out_reg <= 1'b0;
                        sda_en_reg <= 1'b1; // 本机 ACK
                    end else begin
                        state <= S_IDLE;
                        sda_en_reg <= 1'b0;
                    end
                    bit_cnt <= 0;
                end else begin
                    bit_cnt <= bit_cnt + 1;
                end
            end
            S_ACK1: begin
                if(is_read) begin
                state <= S_READ;
                bit_cnt <= 0;
                sda_en_reg <= 1'b1; // 后续由 sda_out_val 连接发数据
               end else begin
```

```
                state <= S_REG_ADDR;
                bit_cnt <= 0;
                sda_en_reg <= 1'b0;
            end
        end
        S_REG_ADDR: begin
            if(bit_cnt == 7) begin
                addr_reg <= shift_reg; // Safely use the shifted byte
                state <= S_ACK2;
                sda_out_reg <= 1'b0;
                sda_en_reg <= 1'b1; // 本机 ACK
                bit_cnt <= 0;
            end else begin
                bit_cnt <= bit_cnt + 1;
            end
        end
        S_ACK2: begin
            state <= S_WRITE;
            bit_cnt <= 0;
            sda_en_reg <= 1'b0;
        end
        S_WRITE: begin
            if(bit_cnt == 7) begin
                state <= S_ACK4;
                addr_reg <= addr_reg + 1;
                sda_out_reg <= 1'b0;
                sda_en_reg <= 1'b1; // 本机 ACK (可写操作)
                bit_cnt <= 0;
            end else begin
                bit_cnt <= bit_cnt + 1;
            end
        end
        S_ACK4: begin
            state <= S_WRITE;
            bit_cnt <= 0;
            sda_en_reg <= 1'b0;
        end
        S_READ: begin
```

```
                sda_en_reg <= 1'b1;
                if(bit_cnt == 7) begin
                    state <= S_ACK3;
                    sda_en_reg <= 1'b0; // 释放 SDA 等待主机回应 ACK
                    bit_cnt <= 0;
                end else begin
                    bit_cnt <= bit_cnt + 1;
                end
            end
            S_ACK3: begin
                if(is_read == 1'b0) begin // 主机发出了 ACK
                    state <= S_READ;
                    addr_reg <= addr_reg + 1;
                     sda_en_reg <= 1'b1;
                     bit_cnt <= 0;
                end else begin
                    state <= S_IDLE; // 主机 NACK
                    sda_en_reg <= 1'b0;
                end
            end
                endcase
            end
        end
    end
    // ============================================================
    // I2C Hardware Debugging via LEDs
    // ============================================================
    reg has_start = 0;
    reg has_addr = 0;
    reg has_read = 0;
    reg has_ack = 0;
    always @(posedge clk) begin
        if(!rst_n) begin
            has_start <= 0; has_addr <= 0; has_read <= 0; has_ack <= 0;
        end else begin
            if(start_bit) has_start <= 1;
            if(state == S_ACK1) has_addr <= 1;
            if(state == S_READ) has_read <= 1;
```

```
                if(state == S_ACK3 && is_read == 0) has_ack <= 1;
            end
        end
        // led[4]=SDA level, led[3]=Host-ACK, led[2]=Read, led[1]=AddrMatch, led[0]=Sta
rtDetect
        assign dbg_led = {sda_clean, has_ack, has_read, has_addr, has_start};


endmodule
```

该模块主要完成对 EDID 的 IIC 配置。具体代码不详细分析，按照 IIC 协议去完成即可。以下给出 IIC 协议的时序说明，

如下图所示：

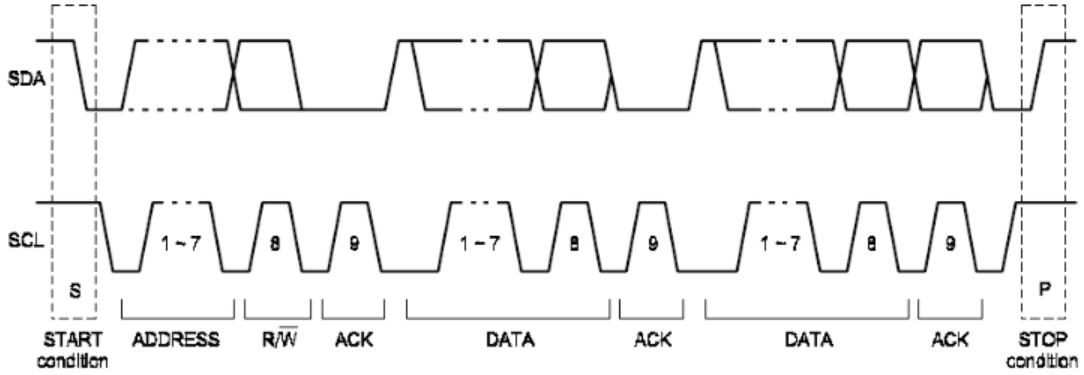


图 8.5-1

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。

8.6.实验现象

连接好开发板、视频源和显示器；

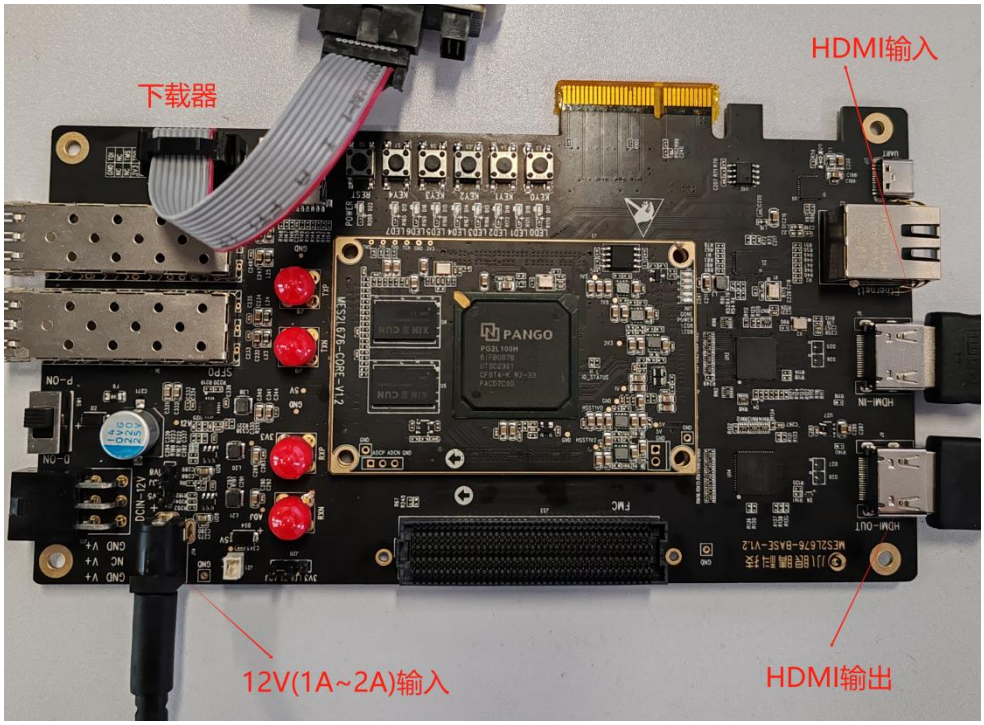


图 8.6-1

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 8.6-2

图 8.6-3

上图为 FPGA 的 HDMI_OUT 输出的图像。

9.音频回环实验

9.1.实验简介

实验目的：

完成 HDMI 回环实验

实验环境：

Window11

PDS2025.2

硬件环境：

PG2K100-6IMBG400

9.2.实验原理

本实验旨在利用 FPGA 实现对 ES8389 高性能音频编解码器 (Codec) 的完全控制。通过构建基于 I2C 的寄存器配置逻辑和基于 I2S 的高速音频数据通路，实现模拟音频信号的实时采集（ADC）与高质量回放（DAC）。

音频信号在自然界中是以连续变化的模拟波形存在的，而 FPGA 作为数字逻辑器件，无法直接处理这些连续的电压变化。为了让机器“听见”声音，必须经过采样、量化和编码三个核心步骤。采样频率决定了捕捉声音波形的频率，通常 48kHz 的采样率足以覆盖人耳的听觉范围；量化位数则决定了振幅描述的精度，本实验采用的 ES8389 能够极大地还原声音的动态范围。这些离散的采样点最终组合成数字信号，成为 FPGA 内部算法可以运算的数学序列。

ES8389 是一款高性能、低功耗立体声音频编解码器（Codec），它在系统中扮演着模拟世界与数字世界“翻译官”的角色。该芯片内部集成了多路高性能的模数转换器（ADC）和数模转换器（DAC），支持宽范围采样率。除了基础的转换功能，ES8389 还内置了可编程增益放大器（PGA）和低噪声麦克风偏置电路，能够对微弱的麦克风信号进行初步放大。其灵活的电源管理和多种数字接口模式，使其成为 FPGA 嵌入式音频系统设计中的理想选择。

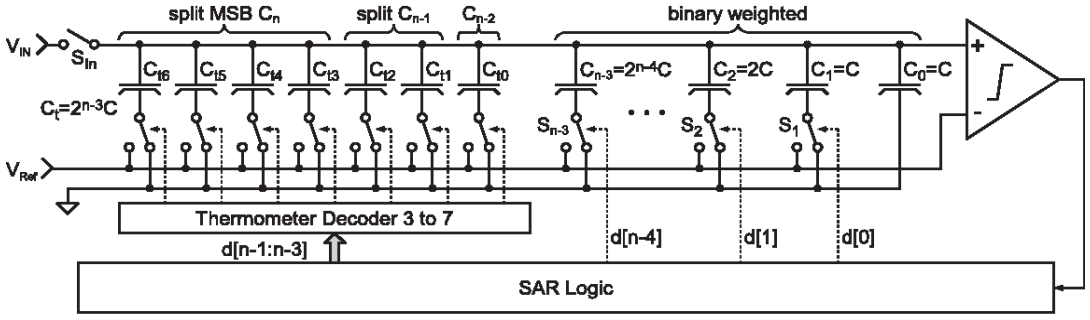
FPGA 与 ES8389 之间的通信由“控制通道”和“数据通道”两部分组成，两者分工明确。控制通道采用 I2C 协议，FPGA 作为主机通过该总线访问芯片内部的寄存器，完成采样率设置、音量调节和路径切换等初始化工作。数据通道则采用 I2S（Inter-IC Sound）标准总线，这是一种专门用于数字音频流传输的同步串行接口。在 I2S 协议下，MCLK 提供主时钟参考，BCLK 驱动位数据传输，而 LRCK 则指示当前传输的是左声道还是右声道数据。FPGA 必须精确控制这些时钟的时序关系，才能确保音频数据流不失真地在两者之间循环。

音频回环（Loopback）实验是基本掌握音频系统最有效的方法，其过程涵盖了从模拟输入到数字处理再到模拟输出的全路径验证。在实验中，外部音源输入的模拟信号首先进入 ES8389 的 ADC 转换为数字流，FPGA 实时接收这些串行数据并将其还原为并行信号。在 FPGA 内部，开发者可以对信号进行直连转发或加入简单的增益处理算法，随后通过 I2S 发送端将数据重新串行化送回 ES8389 的 DAC 模块。最后，经过转换的模拟信

号驱动耳机或音箱发声。通过对比输入与输出的声音质量，开发者可以直观地验证 I2C 配置的正确性、I2S 时序的稳定性以及 PCB 信号完整性的设计水平。

9.2.1.音频原理

音频信号是自然界的连续物理量要想将音频信号在数字世界进行变换必须要经过 ADC 以及 DAC 的参与。ADC 通过采样、保持、量化和编码，将连续的模拟信号转换为数字信号，实现“模拟世界 → 数字世界”的桥梁。DAC，即数模转换器（Digital-to-Analog Converter），是一种将离散的数字信号（由 0 和 1 组成的二进制代码）转换为连续的模拟信号（如电压、电流）的电子器件。而量化（Quantization）是将模拟信号中连续变化的幅度值映射为有限个离散数值的过程，它是实现模数转换（ADC）的核心步骤。在模拟世界中，信号的电压值在一定范围内是无穷尽的，但数字系统只能处理有限位数的二进制数据。量化器的作用就是将整个输入电压范围划分为若干个微小的“阶梯”，当模拟电压落入某个阶梯范围内时，就用该阶梯对应的唯一数字编码来表示。这种从“无限连续”到“有限离散”的转换，本质上是一种舍入或截断操作，决定了数字音频信号对原始声音幅度的还原精度。



图表 8.2-1 位深（Bit Depth）

量化的精度主要由“位深（Bit Depth）”决定，这直接影响了阶梯的密集程度。以你实验中使用的 n-bit 音频芯片为例，它将电压范围分割成了2^n个等级。每一级代表的最小电压单位被称为“量化阶距”或 LSB（最低有效位）。位深越高，量化阶距就越小，数字信号就能更细腻地捕捉模拟波形的微小起伏。如果位深过低，连续的波形就会变成明显的“锯齿状”，导致声音听起来粗糙且缺乏细节。

在量化过程中，由于模拟电压通常不会正好等于某个阶梯的标称值，实际电压与量化后的数字值之间必然存在微小的偏差，这被称为“量化误差（Quantization Error）”。在时域上看，这种误差表现为随机的波动；在频域上看，它演变为一种背景噪声，即“量化噪声”。量化位数每增加 1 位，信噪比（SNR）理论上会提高约 6dB。因此，高性能音频芯片通过增加量化位数和优化量化算法，能够将这种噪声压制到极低的水平，从而实现高保真的音频效果。

在数模转换（DAC）端，量化的原理则体现为一种“重构”过程。DAC 根据输入的数字编码，在内部产生对应电平的阶梯电压。由于量化数据本身是离散的，重构出来的模拟信号最初呈现为台阶状的折线，包含了大量的高频量化谐波。为了还原平滑的原始波形，通常在 DAC 的输出端会配合低通滤波器（LPF）或过采样技术。

9.3.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
key	input	1	音频使能信号
es8389_scl	inout	1	I2C 串行时钟线（SCL）
es8389_sda	inout	1	I2C 串行数据线（SDA）
es0_mclk	output	1	I2S 系统主频时钟（MCLK）
sclk	output	1	I2S 位时钟（BCLK）
lrck	output	1	I2S 声道切分时钟
es1_sdin	input	1	I2S ADC 数据输入接口
es1_sdout	output	1	I2S DAC 数据输出接口
adc_dac_int	output	1	复位释放信号兼状态引脚。

9.4.工程说明

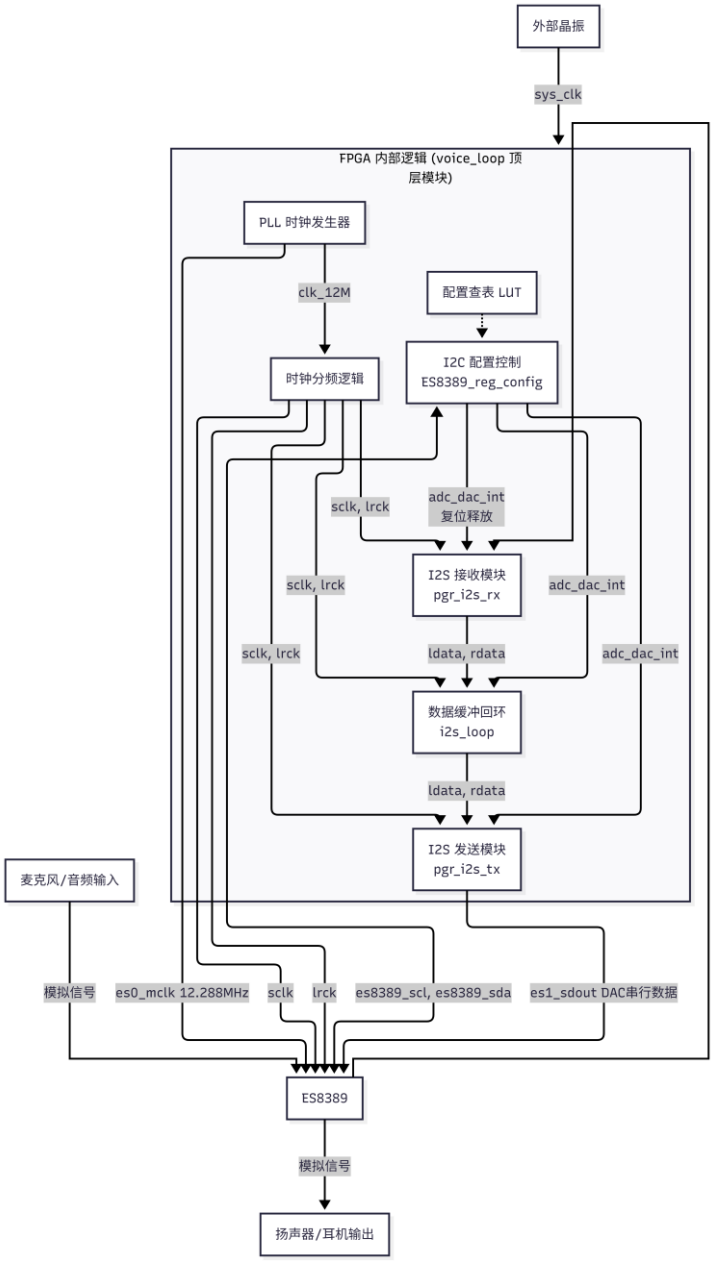


图 9.4-1

外部的模拟声音片段被输入至麦克风，由主板上的 ES8389 芯片内部的 ADC 采样并转换为数字音频信号。接着串数字流在位时钟（sclk）和声道分界时钟（lrck）的并排同步下，遵循标准的 I2S 通信协议，经由硬件引脚 es1_sdin 以串行比特的形式汇入 FPGA。进入 FPGA 后，pgr_i2s_rx 接收模块首先将其串行捕获，并解组转换为左右声道相分离的 16 位并行数据；紧接着，i2s_loop 缓冲模块作为核心回环节点同步承接该并行数据，完成纯数字域的高速数据透明中继。最后，pgr_i2s_tx 发送模块将这套完整的并行音频重新序列化为 I2S 标准串行流，通过引脚 es1_sdout 送回 ES8389 芯片。在终端流中，音频芯片的内部 DAC 会将数字信号顺位解码重造成模拟波形，直接驱动受控扬声器或耳机播放。

9.5.代码模块说明

voice_loop.v 模块

```
module voice_loop(
    input           sys_clk,
    input           key,
    inout           es8389_scl,
    inout           es8389_sda,
    output          es0_mclk,
    output          sclk,
    output          lrck,
    input           es1_sdin,
    output          es1_sdout,
    output          adc_dac_int
);

    wire locked, es8389_init, clk_12M, clock_i2c, test_key;
    wire es1_dsclk, es1_dlrc;
    wire [15:0] ldata, rdata, rx_data;
    wire rx_l_vld, rx_r_vld;
    reg [7:0] i2s_div;

    assign es0_mclk     = clk_12M;
    assign test_key     = locked & ~key;
    assign adc_dac_int  = es8389_init;
    assign sclk         = i2s_div[1];
    assign lrck         = i2s_div[7];
    assign es1_dsclk    = i2s_div[1];
    assign es1_dlrc     = i2s_div[7];

    PLL u_pll(
        .clkin1   (sys_clk),
        .lock     (locked ),
        .clkout0  (clk_12M)
    );

    always @(posedge clk_12M) begin
        if(!locked) i2s_div <= 8'h0;
```

```
        else            i2s_div <= i2s_div + 1'b1;
    end

    ES8389_reg_config  ES8389_reg_config(
        .clk_12M          (sys_clk      ),
        .rstn             (test_key     ),
        .i2c_sclk         (es8389_scl   ),
        .i2c_sdat         (es8389_sda   ),
        .reg_conf_done    (es8389_init),
        .clock_i2c        (clock_i2c    )
    );

    pgr_i2s_rx  #(16)  ES8389_i2s_rx(
        .rst_n  (adc_dac_int),
        .sck    (es1_dsclk    ),
        .ws     (es1_dlrc     ),
        .sda    (es1_sdin     ),
        .data   (rx_data      ),
        .l_vld  (rx_l_vld     ),
        .r_vld  (rx_r_vld     )
    );

    i2s_loop  #(16)  ES8389_i2s_loop(
        .rst_n  (adc_dac_int),
        .sck    (es1_dsclk    ),
        .data   (rx_data      ),
        .l_vld  (rx_l_vld     ),
        .r_vld  (rx_r_vld     ),
        .ldata  (ldata        ),
        .rdata  (rdata        )
    );

    pgr_i2s_tx  #(16)  ES8389_i2s_tx(
        .rst_n  (adc_dac_int),
        .sck    (es1_dsclk    ),
        .ws     (es1_dlrc     ),
        .sda    (es1_sdout    ),
        .ldata  (ldata        ),
```

```
        .l_req  (                ),
        .rdata  (rdata           ),
        .r_req  (                )
    );
endmodule
```

顶层模块 voice_loop.v 是整个数字音频回环工程的核心枢纽。它的主要作用是将系统时钟、复位逻辑、I2C 寄存器控制总线以及 I2S 音频的数据流收发链路整合拼接在一起，以实现硬件级的麦克风声音采集、中继处理再到耳机/扬声器播放。

PLL 锁相环 (生成 MCLK)：首先通过内部 PLL IP 模块把外部传入的系统时钟 (sys_clk) 转换成音频专属基线频—— clk_12M (12.288MHz)。这个时钟会通过 es0_mclk 管脚发送给 ES8389 音频芯片内部做主驱动时钟。使用一个 8 位的累加计数器 i2s_div 以 12.288MHz 为基准不断计数，i2s_div[1] 实现了 4 分频，产生频率为 3.072MHz 的位时钟（对应 sclk/es1_dsclk），用于在 I2S 每比特音频数据发送时进行精确移位打拍。i2s_div[7] 实现了 256 分频，产生频率为 48kHz 的帧声道时钟（对应 lrck/es1_dlrc）。48kHz 对应数字音频中常说的高保真 48kHz 采样率，并用于区分左声道耳和右声道耳。初始化信号锁与按键：assign test_key = locked & ~key; 利用了 PLL 正常启动后的 locked（锁定信号）和板卡上的 key 给 I2C 控制模块提供安全的复位驱动。I2S 收发的就绪释放：assign adc_dac_int = es8389_init; 外部音频芯片 ES8389 需要几十个 I2C 寄存器指令写入后才能正常进入工作模式。因此顶层逻辑将 I2C 配置模块(ES8389_reg_config)输出的完成标志 es8389_init 连给了 adc_dac_int。 值得注意的是，后续的 pgr_i2s_rx、i2s_loop 和 pgr_i2s_tx 模块全都将 adc_dac_int 作为他们的复位引脚（rst_n）。

9.6.实验现象

连接好开发板、耳机以及麦克风（或其他输入源）；

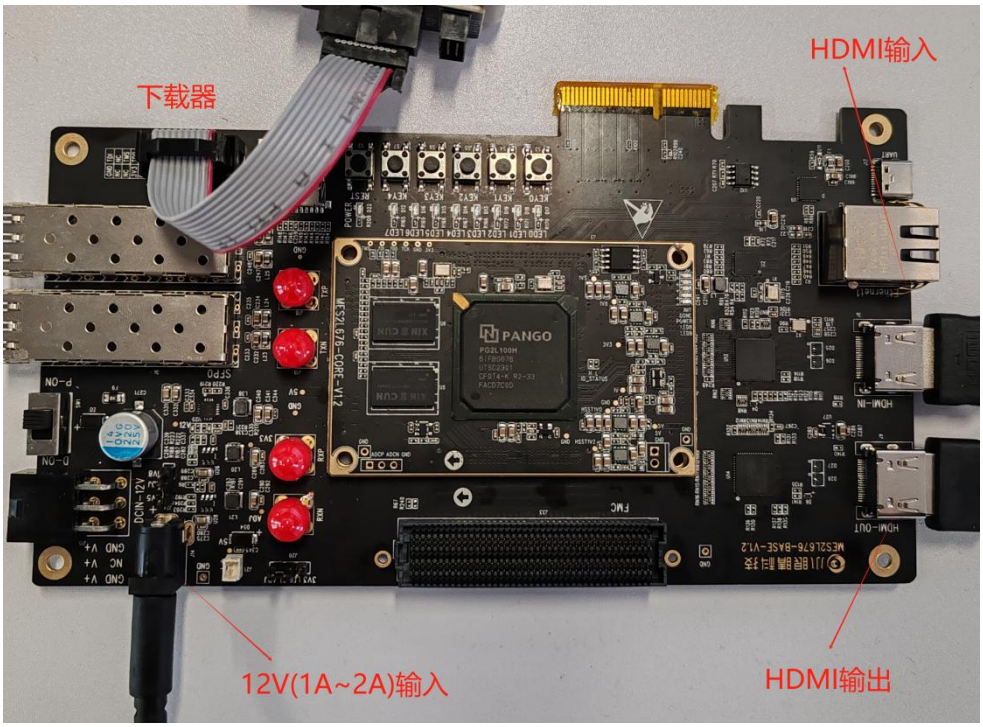


图 9.6-1

下载程序从输出端就能听到麦克风输入音频